Gottfried Wilhelm Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik

Efficient Symbolic Learning over Knowledge Graphs

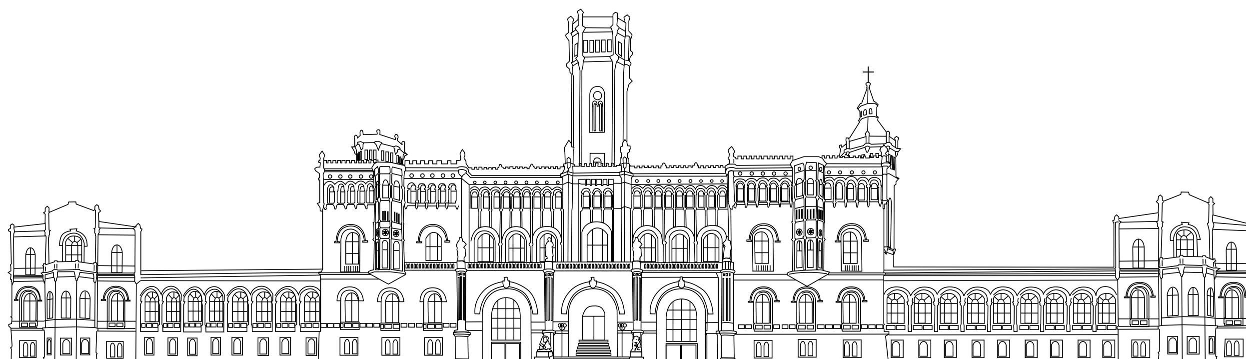
A thesis submitted in fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science

BY

Sohan Deshar
Matriculation number: 10032417
E-mail: sohan.deshar@stud.uni-hannover.de

First evaluator: Prof. Dr. Maria-Esther Vidal
Second evaluator: Prof. Dr. Sören Auer
Supervisor: M.Sc. Disha Purohit

November 14, 2023



Declaration of Authorship

I, Sohan Deshar; declare that this thesis titled, 'Efficient Symbolic Learning over Knowledge Graphs' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Sohan Deshar

Signature: ______________________________

Date: ______________________________

"It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts"

— Arthur Conan Doyle (Sherlock Holmes)

Acknowledgements

I would like to express my appreciation to my advisor Prof. Dr. Maria-Ester Vidal and my supervisor Disha Purohit, M. Sc. for their guidance and feedback.

Abstract

Knowledge Graphs (KG) are repositories of structured information. Inductive Logic Programming (ILP) can be used over these KGs to mine logical rules which can then be used to deduce new information and learn new facts from these KGs. Over the years, many algorithms have been developed for this purpose, almost all requiring the complete KG to be present in the main memory at some point of their execution. With increasing sizes of the KGs, owing to the improvement in the knowledge extraction mechanisms, the application of these algorithms is being rendered less and less feasible locally. Due to the sheer size of these KGs, many of them don't even fit in the memory of normal computing devices. These KGs can, however, also be represented in RDF making them structured and queriable using the SPARQL endpoints. And thanks to software like Openlink's Virtuoso, these queriable KGs can be hosted on a server as SPARQL endpoints. In light of this fact, an effort was undertaken to develop an algorithm that overcomes the memory bottleneck of the current logical rule mining procedures by using SPARQL endpoints. To that end, one of the state-of-the-art algorithms AMIE [5] was taken as a reference to create a new algorithm that mines logical rules over these KGs by querying the SPARQL endpoints on which they are hosted, effectively overcoming the aforementioned memory bottleneck, allowing us to mine rules (and eventually deduce new information) locally.

Keywords: Rule Mining, SPARQL, Inductive Logical Programming, AMIE

Contents

1 Introduction **1**
1.1 Motivation 2
1.2 Contributions 4
1.3 Structure of the Book 4

2 Background **6**
2.1 Knowledge Graph 6
2.1.1 Atoms and Rules 6
2.1.2 Relation and Functions 7
2.1.3 Rule Mining and some useful evaluation metrics 8
2.2 Rule Mining as ILP, and OWA, and PCA 9
2.3 RDF and SPARQL 10

3 Related Work **11**
3.1 RUDIK 11
3.2 Ontological Pathfinding 12
3.3 AnyBURL 12
3.4 AMIE, AMIE+ and AMIE3 13

4 AMIE SPARQL- Our Proposed Approach **14**
4.1 Mining Operators 14
4.2 Pruning 16
4.3 Our Proposed Algorithm for AMIE SPARQL 18

5 Implementation **21**
5.1 Parallel computing with Python 21
5.2 SPARQL Query Generation 22
5.3 Synchronized Rule Store 24

6 Empirical Evaluation **27**
6.1 Experimental Configuration 27
6.1.1 Knowledge Graphs used in Experiments 28
6.2 Results 28
6.2.1 Run-time Performance Results 28
6.2.2 Memory Performance Results 32
6.2.3 Discussion 33

7 Conclusions and Future Work **35**
7.1 Conclusions 35
7.2 Limitations and Future Work 35

Bibliography **37**

List of Figures

1.1 Motivating Example 3

6.1 Algorithm Execution Times 29
6.2 Answer traces over time, familyKG 29
6.3 Answer traces over time, frenchRoyalty 30
6.4 Continuous Efficiency comparison with diefk 30
6.5 Performance comparison with dieft 31
6.6 Memory-trace Comparison, familyKG 32
6.7 Memory-trace Comparision, frenchRoyalty 33

List of Tables

1.1 Sizes of KGs used by popular Tech companies. Data source: [10] 3
1.2 Sizes of openly available large KGs. Data source: [3] 3

List of Algorithms

4.1 The Rule-mining Algorithm 17
4.2 Refinement Algorithm 18
4.3 Dangling Operator 19
4.4 Algorithm to check if the rule should be outputted 19
4.5 Algorithm to check if the rule should be further expanded 20
5.1 Algorithm to store a rule 25
5.2 Algorithm to check if a rule exists in the store 26

Acronyms

CSV Comma-separated values

HTTP Hypertext Transfer Protocol

ILP Inductive Logic Programming

JSON JavaScript Object Notation

KG Knowledge Graph

OWA Open World Assumption

OWL Web Ontology Language

PCA Partical Completeness Assumption

RDF Resource Description Framework

RDFS Resource Description Framework Schema

SPARQL SPARQL Protocol And RDF Query Language

URI Uniform Resource Identifier

Chapter 1

Introduction

In today's world, data is one of the most important intangible resources. Raw data, on the other hand, has limited value by itself. It only becomes helpful and capable of delivering insight and comprehension when it has been processed and contextualized. This understanding and insight about a certain aspect or entity is commonly known as knowledge. Although we, humans, are great at extracting knowledge from information available to us, most of us are incapable of remembering the knowledge for a long period of time without other means. Since some of the knowledge might be relevant for years, or even decades, it becomes essential to record these somehow for look-up later in future. In light of this fact and various other advantages that a digital and physical system of archives entails, various forms of data storage have been devised over the period of human history.
Knowledge Graphs (KG) are one such method of digital information storage, which store not only information but can also capture the semantics behind the stored information, making them both human- and machine-readable and even machine-interpretable. Besides storing information, the ability to incorporate the semantics behind the stored entities makes them a powerful tool for making inferences, for gaining a better understanding of the huge knowledge structure, etc. Although the phrase *'knowledge graph'* itself has been used in literature since the early 1970s, the true potential of the tool was realized much later, after the uptake by tech giants like Google in the early 2010s. Ever since the announcement of the use of KGs in its search engine by Google in 2012 to enhance the search results [14], they have been gaining a lot of popularity in the industry, as indicated by the adoption of KGs by Yahoo and Microsoft in their search engines. Not only search engines but also the fields of machine learning, pharmacy, etc. (to name a few) have found applications of KGs for various purposes. The increase in industry adoption of KGs has led to

increased research interest in the field in recent years as well. The use of KGs for knowledge inference has been one of the areas of keen interest in academia.
The inference methods that have been devised in recent years can be divided into two categories: numeric learning and symbolic learning [7]. In numeric learning, numeric representation of entities and relationships present in KGs are created and used for the inference process, whereas in symbolic learning, the symbols of the entities are used as is for the inference. Rule mining is one such symbolic learning method, and is the central topic of exploration of this thesis. Rule mining is a process of finding patterns in the form of logical rules using the facts that are already available in the KG. As KGs store facts as *(subject, predicate, object)* triple; from presence of triples *(A, isMotherOf, B)*, *(C, isFatherOf, B)*, *(A, isSpouseOf, C)* and *(C, isSpouseOf, A)* in a KG, we can infer following logical rules:

$$(?\boldsymbol{a},\ \textit{isMotherOf},\ ?\boldsymbol{b}) \wedge (?\boldsymbol{c},\ \textit{isFatherOf},\ ?\boldsymbol{b}) \implies (?\boldsymbol{a},\ \textit{isSpouseOf},\ ?\boldsymbol{c}) \quad (1)$$

$$(?\boldsymbol{a},\ \textit{isMotherOf},\ ?\boldsymbol{b}) \wedge (?\boldsymbol{c},\ \textit{isFatherOf},\ ?\boldsymbol{b}) \implies (?\boldsymbol{c},\ \textit{isSpouseOf},\ ?\boldsymbol{a}) \quad (2)$$

Using these rules, if the fact that X is the mother of Y and Z is the father of Y is present in the KG, the fact that X and Z are spouses can be inferred with a certain confidence.

1.1 Motivation

Figure 1.1 gives a brief idea about how the current state-of-the-art mining approach[8] works and what the thesis proposes. As depicted in the figure, the current state-of-the-art algorithm starts by uploading the KG as a list of *subject predicate object* triples in the main memory. Using the uploaded KG, the algorithm starts the mining procedure. The procedure calculates the rule metrics locally, which as described in [5, 4, 8] is not a trivial task.

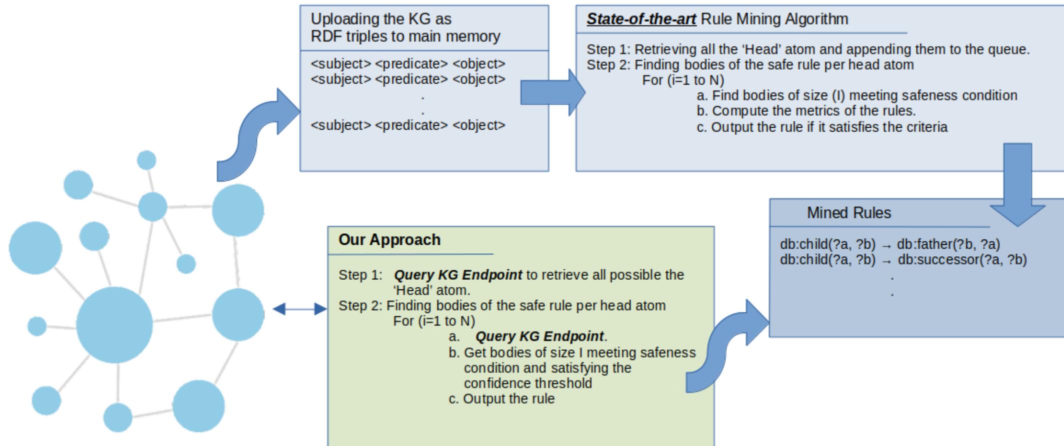


Figure 1.1: Motivating Example: Comparison of state-of-the-art mining algorithm with our approach

Maintained by	Size of the graph
Microsoft	$\tilde{2}$ billion primary entities, $\tilde{55}$ billion facts
Google	1 billion entities, 70 billion assertions
Facebook	$\tilde{50}$ million primary entities, $\tilde{500}$ million assertions
eBay	>1 triples
IBM	>100 million entities, >5 billion entities

Table 1.1: Sizes of KGs used by popular Tech companies. Data source: [10]

Knowledge Graph	Size of the graph
DBpedia	411 million triples
Freebase	3 billion triples
OpenCyc	2 million triples
WikiData	748 million triples
YAGO	1 billion triples

Table 1.2: Sizes of openly available large KGs. Data source: [3]

As Table 1.1 and Table 1.2 show, both openly available and industrial-scale KGs have triples in the hundreds of millions range. With time, these KGs tend to grow to become even larger. This trend is not only seen in these particular KGs but all KGs in general, as there is always something new to record, some new information that needs to be stored. This makes mining rule over the KGs locally a challenging task, especially because with time they might not even fit in storage capacities available

locally. Since the state-of-the-art rule mining algorithm rely on the presence of the KGs locally, in terms of local rule mining we might soon hit the limits, as far as the storage requirements are concerned. To circumvent this storage bottleneck which will unquestionably arise in the future, it is reasonable to explore other possibilities.

Since, there are already software like opensw's virtuoso[1], which allow hosting KG as SPARQL endpoint, in this work we present an algorithm that makes use of these endpoints for mining purpose. We propose a mining algorithm that adapts the current state-of-the-art algorithm to mine rules by only retrieving the required data from these endpoints.

1.2 Contributions

This thesis proposes AMIE SPARQL, an extension of the AMIE [5] rule mining algorithm to iteratively mine rules over KGs hosted as SPARQL endpoints and presents results of an early proof-of-concept implementation of the algorithm in Python. The thesis at the end provides suggestions on ways to improve the algorithm and implementation for better performance.

1.3 Structure of the Book

The thesis is structured as follows.

- **Background**: This chapter goes over the preliminaries required to understand the thesis.
- **Related Work**: The chapter presents briefly works that have been done in recent years in the field of symbolic learning.
- **AMIE SPARQL- Our Proposed Approach**: The chapter goes over the mining model presented by AMIE and presents an adaptation of it tailored for mining rules over KGs hosted as SPARQL endpoints.
- **Implementation**: This chapter goes over some of the details about the implementation of the algorithm in Python and discusses some of the caveats that arose during the implementation.

[1]https://virtuoso.openlinksw.com/

- **Empirical Evaluation**: The chapter analyzes the results of the approach presented in AMIE SPARQL- Our Proposed Approach and presents comparison of the results with the AMIE3[8], latest version of AMIE[5], in various scenarios.
- **Conclusions and Future Work**: The last chapter provides a brief look at possible future work for performance improvement before concluding the thesis.

Chapter 2

Background

This chapter introduces the main topics needed to understand the development of this thesis.

2.1 Knowledge Graph

Knowledge Graph is a graph-based structure $\mathcal{K}$, composed of vertices $\mathcal{V}$ and edges $\mathcal{P}$. The vertices symbolize a set of real-world entities E like *person, organization, country,* etc or literals L like *date, string, number,* etc. The edges $\mathcal{P}$, which are labeled and directed, represent a set of directed relationships between the vertices like *isCapitalOf, isFatherOf, isMotherOf,* etc. In this structure, a directed edge from vertex **A** to another vertex **B** represents a fact about vertex **A**. This is often represented in literature as *(s,p,o)* triple or as the relation between subject and object as $p(s, o)$ with subject $s \in E$, predicate $p \in \mathcal{P}$, and object $o \in E \cup L$. Furthermore, this structure allows for capturing formal specification of the meaning of these entities and the relationship using RDF, RDFS, OWL etc. Since, the complete structure is nothing but a collection of these triples, KGs, as presented here, can be considered as a set of facts about entities.

2.1.1 Atoms and Rules

An **Atom** is, for the purpose of the thesis, *(s,p,o)* triple *[also represented as p(s,o)]*. The subject can be either an entity or variable, $s \in E \cup V$, the object either entity, variable or literal, $o \in E \cup V \cup L$. The predicate can be either a relationship or variable, $p \in \mathcal{P} \cup V$. The **variables** can be thought of as placeholders for entities/relations in the triple. The variables are represented as lowercase italicized letters with

a question mark at the front, ***?x***. As an example: *("Berlin", "isCapitalOf", **?x**)* is an atom, with variable ***?x*** in object position, which when substituted by *"Germany"* would represent a fact in a KG. Similarly, *("Berlin", **?y**, "Germany")* is an atom, with variable ***?y*** in predicate position. This represents all the facts in KG with *"Berlin"* and *"Germany"* in subject and object positions respectively. Thus, based on occurrences of variables, an atom can represent a single fact or a set of facts in the KG. In the scope of the thesis, **Rule** refer to ***Horn Rules*** which are of the form, **Body** $\implies$ **Head**. **Body** is a conjugation of atoms and **head** is a single atom. So a rule here is of the form:

$$B_1 \wedge B_2 \wedge \ldots \wedge B_n \implies H,$$

$$\text{also abbreviated as } \overrightarrow{B} \implies H$$

$$\text{where, } H \text{ is the head of the rule}$$

$$\text{and } B_{\mathbf{n}} \text{ are atoms of the body}$$

Although a rule can be of any form, i.e., the **body** of the rule can be a permutation of any sort of atoms, this work would be considering only *connected* and *closed* rules. Two atoms are ***connected*** if they share at least one variable in their subject/object positions. A rule is *connected* if all of the atoms of the rule are transitively *connected*. And the rule is considered ***closed***, if no variable appears in the rule by itself, i.e., the variables occur more than once. These considerations are mainly to restrict the search space, as allowing any form of rules would most definitely increase the run-time of the algorithm without improving the quality. Following are examples to clarify these restrictions:

$$(\boldsymbol{?x}, predicateA, \boldsymbol{?y}) \wedge (\boldsymbol{?y}, predicateB, \boldsymbol{?z}) \implies (\boldsymbol{?x}, predicateC, \boldsymbol{?z}) \quad (3)$$

$$(\boldsymbol{?x}, predicateA, objVal) \wedge (\underline{\boldsymbol{?p}}, predicateB, \underline{\boldsymbol{?k}}) \implies (\boldsymbol{?x}, predicateC, \underline{\boldsymbol{?z}}) \quad (4)$$

The Rule 3 is closed as well as connected as all the variables of the rule appear in pairs and the atoms of rule are transitively connected, whereas in Rule 4 the variables ***?p***, ***?k*** and ***?z*** appear by themselves and the two atoms of the body are not connected, making the rule as a whole not connected and not closed.

2.1.2 Relation and Functions

In the KG, there can be many subject-object pairs connected by the same predicate, as predicates are just things that express a directional relationship between entities.

So, in a way, the predicate can be considered a binary relation. And functions are a kind of binary relations that connect at most one object to every subject, i.e., $\forall s : |o : p(s, o)| \leq 1$. Similarly, inverse functions are those relations that connect at most one subject to every object. The KGs can include multiple object entries for a subject-predicate pair, which in reality should strictly have only one entry, for example: for *'Albus Dumbledore'-father* pair, the KG could have two object entries *'Percival'* and *'Percival Dumbledore'*. Even though these both entities refer to the same object, the information that both *'Percival'* and *'Percival Dumbledore'* refer to the same, might be missing in the knowledge graph because of the way the KGs are often constructed. To get an idea about several objects that a predicate associates a single subject to, AMIE [5] uses the notion of ***Functionality*** of predicates. *Functionality* of a predicate is a value between 0 and 1. It is defined as:

$$func\,(p) = \frac{|\{s : \exists o : p(s, o) \in \mathcal{K}\}|}{|\{(s, o) : p(s, o) \in \mathcal{K}\}|} \tag{5}$$

Functionality of 1 means that the predicate associates a single subject with exactly one object. The inverse functionality of a predicate is defined as $ifunc(p) := func(p^{-1})$ where $func(p^{-1})$ is calculated using Equation 5 with subject and object reversed.

2.1.3 Rule Mining and some useful evaluation metrics

As discussed in Introduction, Rule mining is a process of finding patterns in the form of logical rules using the information that is already available in the KGs. Using the mined rules, new information that is unknown to the KG can be deduced, but therein lies one of the key problems. Just because a rule implies a new fact, doesn't necessarily mean that the new fact is true, owing to OWA (more on this later) under which KGs operate. As a result, we need some way to prove that the deduced fact is indeed true. Following are some of the statistical measures of the quality of the rule, which gives a rough idea about the quality of prediction that the rule makes.

Support

Support of a rule is a measure of the quality of the rule. As in AMIE[5], support here is also defined as the number of distinct subject-object pairs in the head of all instantiations of the variables of the rule that appear in the KG. Mathematically, this means:

$$supp(\overrightarrow{B} \Rightarrow p(s, o)) := |\{(s, p) : \exists v_1, v_2, \ldots, v_n : \overrightarrow{B} \wedge p(s, o) \in \mathcal{K}\}| \tag{6}$$

In Equation 6, v_n are the instantiations of variables present in the rule. The support of a rule: $(\boldsymbol{?a}, father, \boldsymbol{?b}) \Rightarrow (\boldsymbol{?b}, child, \boldsymbol{?a})$, would thus be the number of unique pairs of instantiations of $\boldsymbol{?a}$ and $\boldsymbol{?b}$ in the KG for which the condition $(\boldsymbol{?a}, father, \boldsymbol{?b}) \wedge (\boldsymbol{?b}, child, \boldsymbol{?a})$ holds true.

Head Coverage

Head Coverage is another measure of the quality of the rule [5]. Since support is an absolute number, it might not provide useful insight into the rule without knowing the actual size of the KG. Head Coverage provides a meaningful alternative. It normalizes the support by the number of unique instances of subject-object pairs that satisfy the head atom. Mathematically, we define head coverage as follows:

$$hc(\overrightarrow{B} \Rightarrow p(s,o)) := \frac{supp(\overrightarrow{B} \Rightarrow p(s,o))}{|\{(s,o) : p(s,o) \in \mathcal{K}\}|} \tag{7}$$

Head Coverage of a rule is a number in the range [0,1]. Head Coverage of 1 means that all the new facts implied by the rule are present in the KG and thus, the implied information is true. This signifies that the predictions made by the rule using some new facts in the body will most likely be true as well.

2.2 Rule Mining as ILP, and OWA, and PCA

Inductive Logic Programming is a process of finding a hypothesis that covers all positive examples and none of the negatives while taking into account a background theory. This is typically realized by searching a space of possible hypotheses."[13] The rule mining, as it is discussed here, can thus be considered as an ILP algorithm, as its aim aligns with that of ILP. As such, in order to evaluate the hypothesis (*rule*, in our case) of an ILP algorithm, we need an evaluation metric that takes into account the positive examples that are covered and the negative ones that aren't covered by the hypothesis. An evaluation metric to evaluate rules over KGs is, thus, not feasible as the KGs operate under the Open World Assumption (OWA). According to OWA, "the truth value of a statement may be true irrespective of whether or not it is known to be true"[11]. This means that, any new fact implied by a rule, if it is not present in the KG, cannot simply be considered as a negative example, as is the case in most other Inductive Logic Programming (ILP) problems. This is because the fact implied may actually be true but just unknown to the KG.

PCA and PCA Confidence

Under Partical Completeness Assumption (PCA), it is assumed that if for a certain subject-predicate pair an object attribute is known, then all the objects for this pair is known. This means that if a rule mining algorithm, outputs a rule which predicts an another object entity for that subject-predicate pair, this observation can be considered as a negative example. Using this assumption, AMIE [5] defined a new metric called PCA Confidence for the evaluation of the rules, which is computed as follows:

$$pcaconf(\overrightarrow{B} \Rightarrow p(s,o)) := \frac{supp(\overrightarrow{B} \Rightarrow p(s,o))}{|\{(s,o) : \exists v_1, v_2, \ldots, v_n, o' : \overrightarrow{B} \wedge p(s,o')\}|} \tag{8}$$

In Equation 8, the support of the rule is normalized by the set of facts that are known to be true in the KG, together with the facts that are assumed to be false under PCA. The Equation 8 is for those predicates p whose $func(p) \geqslant ifunc(p)$. For predicates with $ifunc(p) > func(p)$, the confidence is calculated as:

$$pcaconf(\overrightarrow{B} \Rightarrow p(s,o)) := \frac{supp(\overrightarrow{B} \Rightarrow p(s,o))}{|\{(s,o) : \exists v_1, v_2, \ldots, v_n, s' : \overrightarrow{B} \wedge \underline{p(s',o)\}}|} \tag{9}$$

2.3 RDF and SPARQL

Resource Description Framework (RDF), is a standard framework for representing information on the Web. RDF, along with its extensions Resource Description Framework Schema (RDFS), Web Ontology Language (OWL), etc., define sets of vocabularies to describe and structure resources, data, entities, and relationships on the web. This enables the linking of structured data to produce a graph-based data model using an abstract subject-predicate-object syntax called RDF Triple. This model is often referred to as an RDF Knowledge Graph or simply a Knowledge Graph, as RDF has established itself as the de-facto way to represent entities and relationships in Knowledge Graphs nowadays. SPARQL Protocol And RDF Query Language (SPARQL) is a standardized query language used to retrieve information from databases or any data source that can be mapped to RDF. It can also be used to query RDF-KGs (RDF Knowledge Graphs).

Chapter 3

Related Work

In this section, some of the works that have been done in recent years in the field of symbolic learning are discussed. These are the approaches that have shown remarkable performance in terms of run-time and output of the proposed algorithm.

3.1 RUDIK

RUDIK [12], Rule Discovery in Knowledge Bases, is a system that generates declarative horn rules over Knowledge Graphs. It starts by fetching the triples that are relevant to a given predicate from a SPARQL endpoint. Using these triples it reconstructs a smaller graph adding additional edges between literals of similar types to show the order relation between the literals. The refined graph is then used to identify paths containing the variable in the rule's head using A* algorithms to generate the rule's body. The rules thus constructed are then filtered using the positive examples and negative examples, which are generated using PCA (referred to as Local Closed World Assumption in the paper), to produce a non-exhaustive list of rules which can then be used to deduce new facts. Despite its innovative approach, AMIE3 [8] is still able to outperform RUDIK in terms of run-time while still being exhaustive. The approach outlined here in the thesis, as it is based on AMIE3, and can circumvent the memory and the computing overhead of having to download the tuples into the memory and reconstruct the graph, should allow it to compete quite well against RUDIK even though a direct comparison hasn't been made at the time of writing.

3.2 Ontological Pathfinding

Ontological Pathfinding (OP) [2], is another scalable mining algorithm, that achieves its scalability via a series of parallelization, utilizing a highly concurrent architecture based on Spark, and optimization techniques based on Schema-Graph, which is constructed with the help of semantic information about the entities and relations available in the KG. The use of concurrent architecture and the schema graph allows OP to produce semantically correct rules and calculate exact quality scores of the rules making it another contender for the "state-of-the-art". However, the system allows for the mining of rules containing only 3 atoms, requiring the user to implement the mining procedure if rules with more atoms are required. This makes OP less flexible than the algorithm presented here. And since AMIE3 [8] reports itself as being faster than OP and being able to produce more general rules, the algorithm presented here, as it is based on AMIE, should fair quite well against OP as well, although a direct comparison wasn't made at the time of writing.

3.3 AnyBURL

Anytime Bottom-Up Rule Learning (AnyBURL) [9], is yet another rule mining and KG completion technique. It starts by sampling paths from the KG which is uploaded to the main memory and generalizes these sampled path to produce rules, calculates their score and if these score satisfy criterion set, accepts them for output. This procedure of sampling, score calculation and output is repeated for a certain period of time, mining session. The rules produced in these mining sessions is used afterward for KG completion. As innovative as the approach is, it still makes a major misassumption that the complete KG can be uploaded in the main memory. Furthermore, the algorithm relies on sampling paths. This means that for smaller mining sessions, the output of the algorithm depends heavily on paths that were sampled. This also means that there is certain chance that the rules produced by the algorithm is not exhaustive. The paper itself mentions that in some KGs the results produced after 20000 seconds were no different than the results produced after 10000 seconds of mining sessions, making the approach less reliable. The approach presented here corrects the misassumption of infinite memory by using KGs hosted as SPARQL endpoints. Our approach mines rule in top-down fashion using the operators which allow us to exhaustively search for rules over the complete KG and produce results in deterministic manner, making it relatively more reliable.

3.4 AMIE, AMIE+ and AMIE3

AMIE[5], AMIE+ [4], and AMIE3[8], with AMIE3 being the current state-of-the-art algorithm in symbolic learning over Knowledge Graphs, are a group of ILP algorithms that mine rules over large KGs by loading the whole KG represented as *(subject, predicate, object)* triples in an internal data structure stored in the local memory. These works successively introduced several optimization techniques to make iterative rule mining faster and more efficient, making the latest version, AMIE3, the current "state-of-the-art" in the category. But as good as these algorithms are, they make a single major misassumption that the user using these algorithms has infinite local memory, and today's large KG containing triples in the range of millions can be uploaded into local memory. The approach presented here tries to correct this misassumption by adapting these algorithms by making use of KG hosted as a SPARQL endpoint for mining purposes. Even in smaller KG containing triples in the range of thousands, the adapted algorithm (implemented in Python), is shown to have better memory performance while producing almost the same rules at the expense of some run-time performance, which can most definitely be improved (as the current implementation is still in its early stages of development).

Chapter 4

AMIE SPARQL- Our Proposed Approach

In any ILP algorithm, in order to produce a hypothesis, there needs to be some systematic mechanism for exploring the hypothesis space. Here, in the context of mining rules over a KG, the complete KG itself is the search space that needs to be explored. For KGs with millions of triples, it thus becomes essential that the exploration mechanism is systematic and very efficient in order to keep the run-time as well as the memory requirement of the algorithm reasonable. As in AMIE [5], *Mining operators* and *pruning* together are used as the exploration mechanism in the approach, the details of which are discussed in sections below along with the algorithm itself.

4.1 Mining Operators

A rule is implemented as a list of atoms, with the first element symbolizing the head of the rule and the remaining elements representing the body. Mining rule thus implicitly means iteratively adding atoms to this list and *mining operators* are the means of exploring the search space to find possible candidate atoms to add to the list. The operators are as follows:

Dangling Operator ($\mathcal{O}_D$)

The *Dangling Operator* adds a new atom B_d to the input rule $\overrightarrow{B}$. The added atom, with predicate $p_d \in \mathcal{P}$, shares either subject or object with the preceding atom B_n of the rule, which can be either a variable or even an entity, $V \cup E$. The other

component is occupied by a *fresh variable* $z \in V_{free}$, which are the variables that haven't occurred in any other atom of the input rule.

$$\begin{aligned}
&\mathcal{O}_D : (B_0, B_1, \ldots, B_n) \longrightarrow (B_0, B_1, \ldots, B_n, B_d) \\
&\text{where:} \\
&n \in N_0, z \in V_{free}, B_n : (x, p, y) \\
&B_d \in \{(x, p_d, z), (y, p_d, z), (z, p_d, x), (z, p_d, y)\}^1 \\
&x, y \in V \cup E \text{ and } p, p_d \in \mathcal{P}
\end{aligned}$$

Closing Operator ($\mathcal{O}_C$)

The *Closing Operator* adds a new atom B_c to the input rule $\overrightarrow{B}$. Similar to the dangling operator, the added atom shares either the subject or object with the preceding atom of the rule. But unlike the dangling operator, the other component of B_c is filled by a variable $z \in V_{bound}$, which are variables that have already occurred in other atom(s) of the input rule.

$$\begin{aligned}
&\mathcal{O}_C : (B_0, B_1, \ldots, B_n) \longrightarrow (B_0, B_1, \ldots, B_n, B_c) \\
&\text{where:} \\
&n \in N_0, z \in V_{bound}, B_n : (x, p, y) \\
&B_c \in \{(x, p_d, z), (y, p_d, z), (z, p_d, x), (z, p_d, y)\}^1 \\
&x, y \in V \cup E \text{ and } p, p_d \in \mathcal{P}
\end{aligned}$$

Instantiation Operator ($\mathcal{O}_I$)

The *Instantiation Operator* also works similarly to the dangling operator, in that it adds a new atom B_i to the input rule $\overrightarrow{B}$ which shares either subject or object with the preceding atom of the rule. Contrary to the dangling operator, however, the

[1] This set represents just the pattern of atoms that the operator may add to the rule. In reality, the actual set may contain more than four atoms. But all the atoms that the operator adds follow one of these four patterns.

remaining component of B_i is filled by an entity $z \in E$.

$$\mathcal{O}_I : (B_0, B_1, \ldots, B_n) \longrightarrow (B_0, B_1, \ldots, B_n, B_i)$$
where:
$$n \in N_0, z \in E, B_n : (x, p, y)$$
$$B_c \in \{(x, p_d, z), (y, p_d, z), (z, p_d, x), (z, p_d, y)\}^1$$
$$x, y \in V \cup E \text{ and } p, p_d \in \mathcal{P}$$

On applying the above operators one after the other, the whole search space of the rules can be explored. But doing so would be very inefficient, as it would produce the entire search space of possible rules, which would include rules that might not be that useful, or rules that are not *closed* and *connected*. Not only that, the exploration would have very high run-time and memory requirements, which would defeat the whole purpose. To prevent that the *pruning* comes into play.

4.2 Pruning

Pruning refers to the process of eliminating candidate rules from further extension and/or being output. Following AMIE, the intermediate rules with head coverage less than 0.01 are excluded from further expansion, the reason being the monotonicity of the head coverage and the lower significance of such rules. Since head coverage is a monotonic measure, monotonically decreasing with an increase in the length of the rule, expansion of any rule with less than 0.01 head coverage results in rules with even lower head coverage value in most cases. We consider such rules marginal and thus exclude them from expansion.
Besides head coverage, the work also prunes the rules on PCA Confidence. For PCA Confidence a lower bound of 0.1 is set for the rules in order to qualify them for output. Any rule with PCA Confidence lower than that is excluded from being output, as the fact deduced by such a rule is statistically less likely to be true. From output are also excluded, rules with PCA confidence lower than that of the parent rule. In addition to the above, the algorithm keeps a record of rules that have already been explored for extension and the rules that have already been outputted. Every rule before being expanded further and before being output is checked against these records. This ensures that no resource is wasted in exploring the same rule multiple times and prevents outputting the same rule twice.

As in AMIE, the length of the rule is also used to prune potential candidates. AMIE observed that the rules with atoms more than 3 tend to be convoluted and less insightful and thus by default the algorithm is configured to mine rules of length 3. Similarly, our approach also mines rules with a maximum of 3 atoms by default. Candidate rules that are of length 3 but aren't *closed* and/or *connected* are thus excluded from the output. Rules that are already of maximum length aren't further expanded. Furthermore, any candidate rules that are one atom short of maximum length and that have more than 2 *free variables* are also pruned, as such rules cannot be *closed* by any of the *mining operators* and thus will not end up in output.

Algorithm 4.1 The Rule-mining Algorithm

```
 1: function MineRule(Rule, Endpoint, minPcaConf, minHC, atomsToAdd)
 2:     RetrieveFunctionalityInfo(Endpoint, Rule)
 3:     if atomsToAdd == 0 then
 4:         return
 5:     end if
 6:     if exploredRules.contains(Rule) then
 7:         return
 8:     end if
 9:     exploredRules.append(Rule)
10:     possibleExtensions ← Refine(Endpoint, Rule, minPcaConf, minHC)
11:     while ¬possibleExtensions.isEmpty() do
12:         a ← possibleExtensions.pop()
13:         rule.append(a)
14:         if acceptedForOutput(rule) then
15:             output(rule)
16:         end if
17:         if ¬shouldBeExpanded(rule, atomsToAdd − 1) then
18:             continue                                ▷ Skip rest of the loop body
19:         end if
20:         MineRule(rule, Endpoint, minConf, minHC, atomsToAdd − 1)
21:     end while
22: end function
```

4.3 Our Proposed Algorithm for AMIE SPARQL

Algorithm 4.1 is the core algorithm in pseudo-code. The mining algorithm is implemented as a recursive function, which takes 'intermediate' *Rule*, the SPARQL *Endpoint* on which the KG is hosted, the threshold for PCA confidence *minPcaConf*, the threshold for head coverage *minHC* and *atomsToAdd* as inputs. Initially, the *Rule* is just a list with a head atom, which is recursively extended on successive calls. In line 2, the *functionality score* of the rule-head is retrieved from the given *Endpoint*, which is later used for the calculation of the PCA confidence value in "REFINE" function in line 10. The function returns if no more atoms are to be added to the given rule. *exploredRules* (in line 6 and 9) is the data structure, we use to keep track of rules that are explored in each recursion. The function returns also when called with an input rule, that has already been explored (line 6 - 8). If the input rule hasn't been explored, it is added to the *exploredRules* and expanded afterwards by "REFINE" function described in pseudo-code in Algorithm 4.2.

Algorithm 4.2 Refinement Algorithm

```
1: function REFINE(Endpoint, Rule, minPcaConf, minHC)
2:     possibleExtensions ← ⟨⟩
3:     operators ← [danglingOperator, closingOperator, instantionOperator]
4:     for op ∈ operators do
5:         a ← op(Endpoint, Rule, minPcaConf, minHC)
6:         possibleExtensions.UPDATE(a)
7:     end for
8:     return possibleExtensions
9: end function
```

The "REFINE" function as shown in Algorithm 4.2 applies each of the mining operators to the input *Rule* one after the other and returns all possible candidate atoms that can be added to the input rule in order to extend it. Algorithm 4.3 presents the workings of the dangling operator in pseudo-code for clarity. . The Algorithm uses the input rule to generate a list of possible patterns of atom that dangling operator $\mathcal{O}_D$ can add. For each of these patterns, a SPARQL query is generated incorporating the PCA confidence[1] threshold (*minPcaConf*) and head coverage threshold (*minHC*) as filters in the query itself. On executing these queries

[1] Since PCA confidence makes sense only for rules that are closed and connected, the PCA confidence filters are conditionally added in the generated queries for closed and connected rules only. Checking if an intermediate rule is connected and closed is trivial task of counting variables.

against the given *Endpoint*, all the possible candidate atoms are obtained which satisfy both the PCA confidence threshold and the head coverage threshold without the need of any local computations. The other operators function in similar manner.

Algorithm 4.3 Dangling Operator

```
1: function danglingOperator(Endpoint, Rule, minPcaConf, minHC)
2:     danglers ← Rule.getDanglerPatterns()
3:     possibleExtensions ← ⟨⟩
4:     while ¬danglers.isEmpty() do
5:         atomPattern ← danglers.pop
6:         q ← generateQuery(Rule, atomPattern, minPcaConf, minHC)
7:         results ← executeQuery(Endpoint, q)
8:         possibleExtensions.update(results)
9:     end while
10:    return possibleExtensions
11: end function
```

Algorithm 4.4 and Algorithm 4.5 check if the rule should be excluded from output and further expansion respectively as discussed in Section 4.2.

Algorithm 4.4 Algorithm to check if the rule should be outputted

```
1: function acceptedForOutput(Rule)
2:     if ¬Rule.isClosed() ∧ Rule.pcaConf < Rule.parent.pcaConf then
3:         return False
4:     end if
5:     if outputtedRules.contains(Rule) then
6:         return False
7:     end if
8:     outputtedRules.append(Rule)
9:     return True
10: end function
```

Algorithm 4.5 Algorithm to check if the rule should be further expanded

```
1: function shouldBeExpanded(Rule, atomsToAdd)
2:     if atomsToAdd == 1 ∧ Rule.danglingVariables.length > 2 then
3:         return False
4:     end if
5:     if Rule.pcaConf == 1.0 then
6:         return False
7:     end if
8:     return True
9: end function
```

Chapter 5

Implementation

This chapter discusses some of the non-trivial implementation details of our algorithm presented in chapter 4. The implementation was done in Python for the simplicity of language and to circumvent the memory and run-time overhead of Java Virtual Machine (JVM), which AMIE3 has.

5.1 Parallel computing with Python

Parallel computing refers to a type of computation in which multiple processes are run simultaneously. Specifically for our purpose, by using parallel computing, we can mine rules faster by running the mining algorithm and/or parts of it in multiple processes or threads. Like any other modern programming language, python also allows two ways of parallel computing: Concurrency and Parallelism. Concurrency via the use of multiple threads is an efficient way of parallel computing as it has lesser overhead than using multiple processes (Parallelism). However, "In CPython, the global interpreter lock, or GIL, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once."[6] Since almost all the Python implementation that is in use today is CPython, what this means is that, because of GIL every multi-threaded Python application is effectively single-core, even though we may have a multi-core processor. For our use case, we had to rely on a mix of multi-threading and multi-processing features that Python offers, instead of just using multi-threading.
In our implementation, for every possible rule head, the mining algorithm, Algorithm 4.1, is started in a new worker process using the *ProcessPoolExecutor* of the *multiprocessing* package. To find possible extensions for the rule head, the mining operators, the *op* function in line 5 of Algorithm 4.2, are started by these worker

process in separate worker threads using the *ThreadPoolExecutor* of the *multithreading* package. These threads themselves start further threads to query the SPARQL endpoints, the *executeQuery* function in line 7 of Algorithm 4.3. Since executing query is an asynchronous operation with certain time delay between firing the query and getting the results, executing 10 queries in 10 threads and collecting the results ends up taking about the same amount of time as executing just 1 query in the main thread and getting the results. After the possible candidates to add to the rule are found, the mining algorithm is called again in a new worker thread, line 20 of Algorithm 4.1 for each of the extended rules. In this way, the algorithm is implemented in a highly parallelized fashion.

5.2 SPARQL Query Generation

Listing 5.1 shows the pattern of query that is generated, by *generateQuery* function in line 6 Algorithm 4.1, and executed, by *executeQuery* in line 7 Algorithm 4.1, to get the needed data from the SPARQL Endpoint. In following paragraphs, we break down the query and go over the parts:

Part 1

grouping_variables are the variables of interest. For example, in case of *dangling operator* and *closing operator*, this would be a single variable which would bind to the possible predicates for the new atom that would be added by the operators. In case of *instantiation operator*, this would be two variables which would bind to distinct combinations of possible predicate and entity in the new atom. **head_triple_pattern** and **body_triple_pattern** indicate the head atom expressed in triple pattern and body atoms expressed in triple pattern. For a rule like $predecessor(a, b) \wedge father(a, b) \Rightarrow child(b, a)$, the head_triple_pattern would be *(?b child ?a)* and the body_triple_pattern would be *(?a father ?b).(?a predecessor ?b)*. From this part, we obtain the predicate value or (predicate, entity) values that we are searching for along with the associated *support*.

Part 2

From this part, the number of distinct pairs of entities that bind to the head variables of the rule is obtained.

Listing 5.1: The SPARQL Query generated by line 6 of Algorithm 4.3

```
SELECT
    ?support
    ?head
    ((?support/?head) as ?head_coverage)
    ?pca_body_size # conditional
    ((?support/?pca_body_size) as ?pca_confidence) # conditional
    grouping_variable
WHERE {
    {
        # Part 1
        SELECT
            grouping_variable
            xsd:float(COUNT(DISTINCT *)) as ?support
        WHERE {
            SELECT
                head_variables
                grouping_variable
            WHERE {
                head_triple_pattern .
                body_triple_pattern
            }
        }
        GROUPBY grouping_variable
    }
    {
        # Part 2
        SELECT
            xsd:float(COUNT(DISTINCT *)) as ?head
        WHERE {
            head_triple_pattern
        }
    }
    {
        # Part 3
        SELECT
            grouping_variable
            xsd:float(COUNT(DISTINCT *)) as ?pca_body_size
        WHERE {
            SELECT
                pca_head_variables
                grouping_variable
            WHERE {
                pca_head_triple_pattern .
                body_triple_pattern
            }
        }
        GROUPBY grouping_variable
    }
    # Part 4
    FILTER((?support/?pca_body_size) > min_pca_conf) # conditional
    FILTER((?support/?head) > min_head_cov)
}
```

Part 3

This part, along with the lines marked with # *conditional*, are conditionally generated only for the rules that are connected and closed as explained in Section 4.3. The part gets the *PCA body size* for values that bind to *grouping_variables* of a closed and connected rule. For a closed and connected rule like $predecessor(a, b) \wedge father(a, b) \Rightarrow child(b, a)$, **pca_head_triple_pattern** would be *(?b child ?a')* and the corresponding **pca_head_variables** would be *?b ?a'*, assuming $func(child) \geq ifunc(child)$. In case of $func(child) < ifunc(child)$, these would be *(?b' child ?a)* and *?b' ?a* respectively.

Part 4

min_pca_conf and min_head_cov are the PCA confidence and the head coverage thresholds passed to the *generateQuery* function. The query generator function takes these thresholds into account and generates and adds the SPARQL *FILTER*s to the query. These are responsible for filtering out the values that bind to *grouping_variables* that don't fulfill the thresholds.

5.3 Synchronized Rule Store

The *exploredRules* in Algorithm 4.1 and the *outputedRules* in Algorithm 4.4 are the data-structures that we use to store the rules and check if a rule is already present in the data-structures. These structures are the instances of *Synchronized Rule Store*, which is a hash-based data structure for storing rules in memory. The structure allows for synchronized read and write access from multiple threads. This is needed in our case, because the algorithm we have, is a highly concurrent one, with multiple threads trying to store the rules that they have outputted or explored and to check if a rule is already explored or outputted.
The structure is implemented as a key-value map containing further key-value maps. The hash of the string representation of atoms in the rule is used as a key to access the associated value, which is yet another key-value map. Algorithm 5.1 shows in pseudo-code, how a rule is added to the store and Algorithm 5.2 shows how the existence of a rule in the store is checked. The atoms in the body of the rule are reordered according to the dictionary order of predicates in the atom beforehand. This helps in making the store efficient by making sure that only a single version of the same rules with different permutations of atoms in the rule body like $predecessor(a, b) \wedge father(a, b) \Rightarrow child(b, a)$ and $father(a, b) \wedge predecessor(a, b) \Rightarrow$

Algorithm 5.1 Algorithm to store a rule

```
 1: function storeRule(Rule, keyValueMap)
 2:     current ← keyValueMap
 3:     list ← Rule.atomsList
 4:     while ¬empty(list) do
 5:         a ← list.pop()
 6:         key ← hash(a)
 7:         if current.exists(key) then
 8:             current ← current.getValue(key)
 9:             continue
10:         end if
11:         t ← KeyValueMap <>          ▷ Setting t to an empty key-value map
12:         current.add((key, t))
13:         current ← t
14:     end while
15:     headCovList ← current.getValue("headCovList")
16:     headCovList.append(Rule.headCov)
17: end function
```

child(b, a) is stored. As seen in line 16 of Algorithm 5.1, in the last key-value map, under the key "headCovList", a list of head coverage values is kept for the rules that have atoms with the same predicates. This allows for the storing of rules with the same predicates in their atoms but different variables. The *keyValueMap*s in these algorithms are implemented as Python dictionaries. As checking the existence of a key in a Python dictionary has constant time complexity, and the rules in our case have a fixed maximum length, both adding a rule and checking the existence of the rule is done in constant time. In a KG with $\mathcal{P}$ predicates, the space complexity with max rule length 3 is at most $\mathcal{O}(\mathcal{P}^4)$, reducing memory consumption.

Algorithm 5.2 Algorithm to check if a rule exists in the store

```
 1: function storeRule(Rule, keyValueMap)
 2:     current ← keyValueMap
 3:     list ← Rule.atomsList
 4:     while ¬empty(list) do
 5:         a ← list.pop()
 6:         key ← hash(a)
 7:         if current.exists(key) then
 8:             current ← current.getValue(key)
 9:             continue
10:         end if
11:         return False
12:     end while
13:     headCovList ← current.getValue("headCovList")
14:     if headCovList.exists(Rule.headCov) then
15:         return True
16:     end if
17:     return False
18: end function
```

Chapter 6

Empirical Evaluation

The chapter discusses the experiments that were performed and the obtained results. As the algorithm presented in this work is based on AMIE, the experiments were constructed with the main goal of checking how well the algorithm performs in comparison to AMIE3.

6.1 Experimental Configuration

For the experiment, our algorithm was configured with the default configuration of AMIE3, i.e, the head coverage threshold was set to 0.01, the PCA confidence threshold was set to 0.1, the maximum length of the rule was set to 3 and only the rules with variables were mined, i.e, like AMIE3 by default, the *instantiation operator* was not used in mining procedure. All the experiments were conducted on a machine running Ubuntu 22.04 with 32 gigabytes of RAM and an Intel i7 8th gen processor with 4 cores and 8 threads.
Two set of experiments were performed, the first to check the memory and run-time performance and the second to check the soundness and completeness of the output of the presented algorithm. In the first groups of experiments, we decided to use *diefficiency metrics*[1] to compare the run-time performance of the algorithms as these metrics enable us to quantify and evaluate the efficiency of the algorithms over a period of time. The python package *diefpy*[2] was used to calculate and compare the *diefficiency metrics*. For the *diefficiency metrics* calculation, the AMIE3 had to be modified to include in the rule output, the exact time at which the algorithm made the decision to output the rule, as this was not included in its output by

[2]https://pypi.org/project/diefpy

default. In order to make the actual run-time performance comparison fair, the time it took to download the KG as a (Tab-separated values) TSV file required by AMIE3 to start the mining rule was also added up in run-time. This allowed us to compare the run-time of algorithms without any presumptions, from the very start till all possible rules were outputted and the mining procedure was finished. For the memory performance comparison, the python package *memory profiler*[3] was used to record the memory usage of the algorithms while they were mining rules. In the case of AMIE, the memory usage while downloading the KG as TSV file was also recorded, which is reasonable as the experiments were considering the rule mining procedure from start to finish, i.e., from data preparation for AMIE3 till the output of the last rule. In the second group of experiments, the output rules of AMIE3 were taken as the ground truth. The hash-based data structure presented in Section 5.3 was used to check the completeness and soundness of our proposed algorithm.

6.1.1 Knowledge Graphs used in Experiments

The experiments were conducted over two small-sized Knowledge Graphs, named *familyKG* and *frenchRoyalty*. The *familyKG* contains facts about members of a hypothetical family and *frenchRoyalty* contains facts about the members of the French Royal Family and relationships between them. The *familyKG* contains 10,741 triples and *frenchRoyalty* contains 7,048 triples. For the experiments, these KGs were hosted as SPARQL endpoints using virtuoso[4] on servers. The machine on which the experiments were conducted, was on the same LAN as these servers.

6.2 Results

6.2.1 Run-time Performance Results

Figure 6.1 shows a comparison between the execution time[5] of AMIE3 and the algorithm presented in chapter 4 (which is referred in following sections as AMIE_SPARQL). AMIE3 outperforms AMIE_SPARQL in it's current stage in terms of run-time. We see that in familyKG, AMIE3 is able to produce all of the rules under default configuration in just over 2 seconds, whereas AMIE_SPARQL takes around 7.5 seconds. In case of frenchRoyalty, AMIE3 produces all of the rules just under 2 seconds, whereas AMIE_SPARQL takes around 7 seconds.

[3]https://pypi.org/project/memory-profiler

[4]https://virtuoso.openlinksw.com

[5]The execution time of both algorithms were calculated using GNU time to ensure fairness.

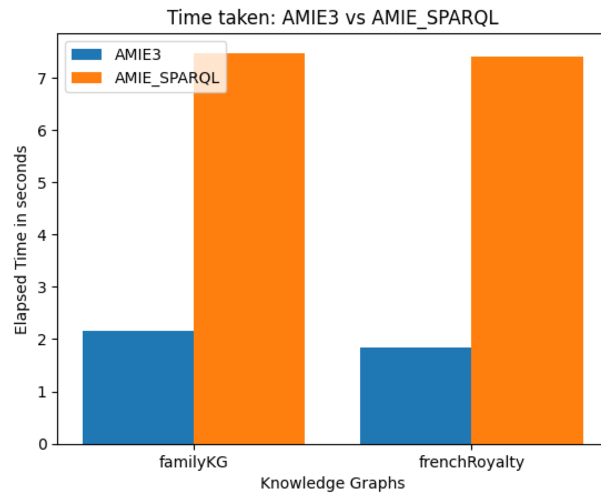


Figure 6.1: Time taken by **AMIE3** and our approach (in fig: **AMIE_SPARQL**) to mine rules over the KGs in default configuration

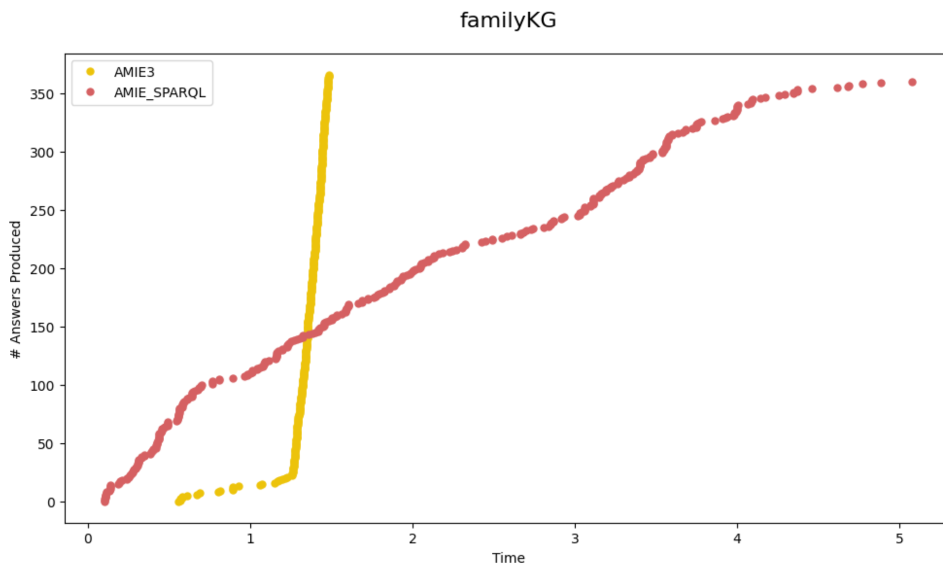


Figure 6.2: Trace of rules produced by AMIE3 and AMIE_SPARQL over time over *familyKG*

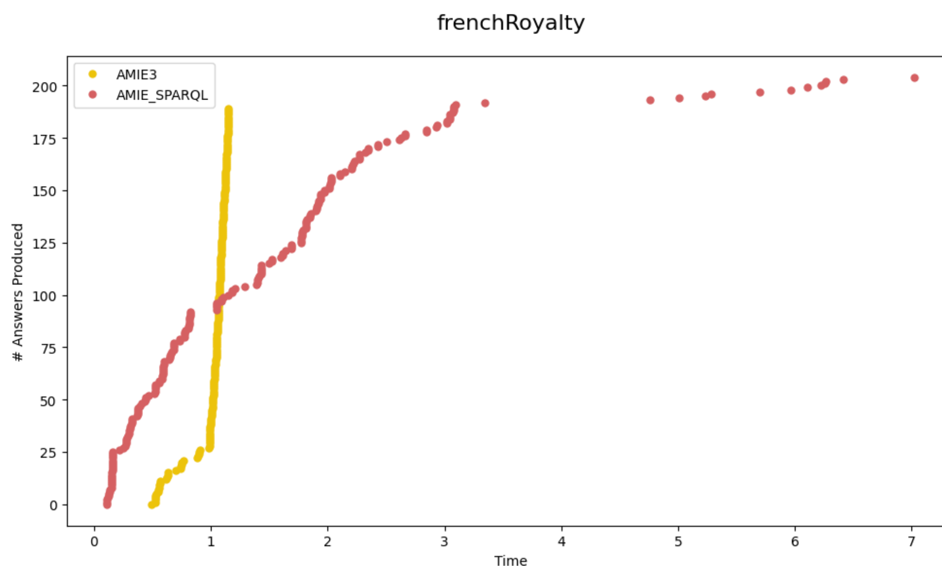
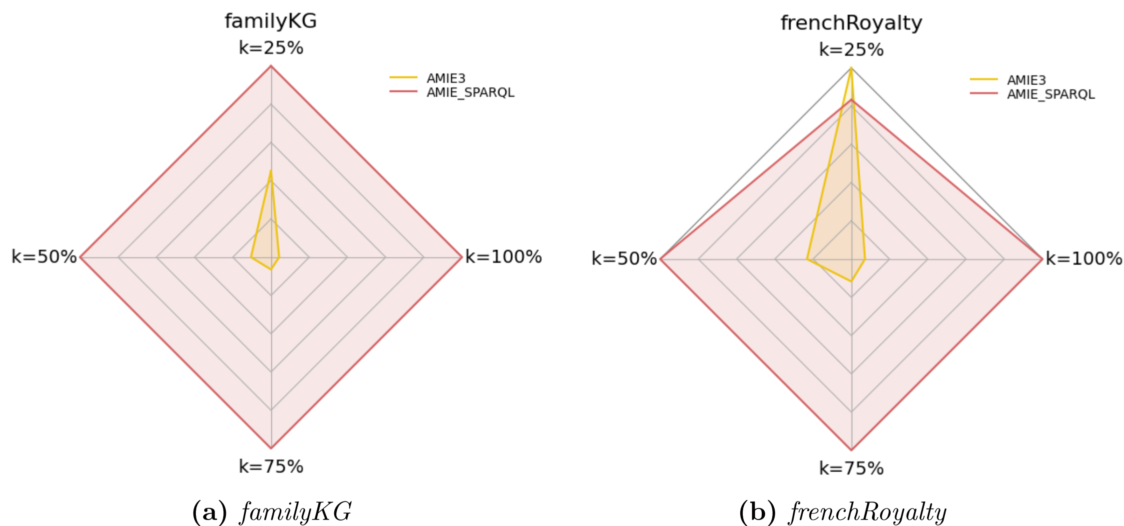


Figure 6.3: Trace of rules produced by AMIE3 and AMIE_SPARQL over time over *frenchRoyalty*

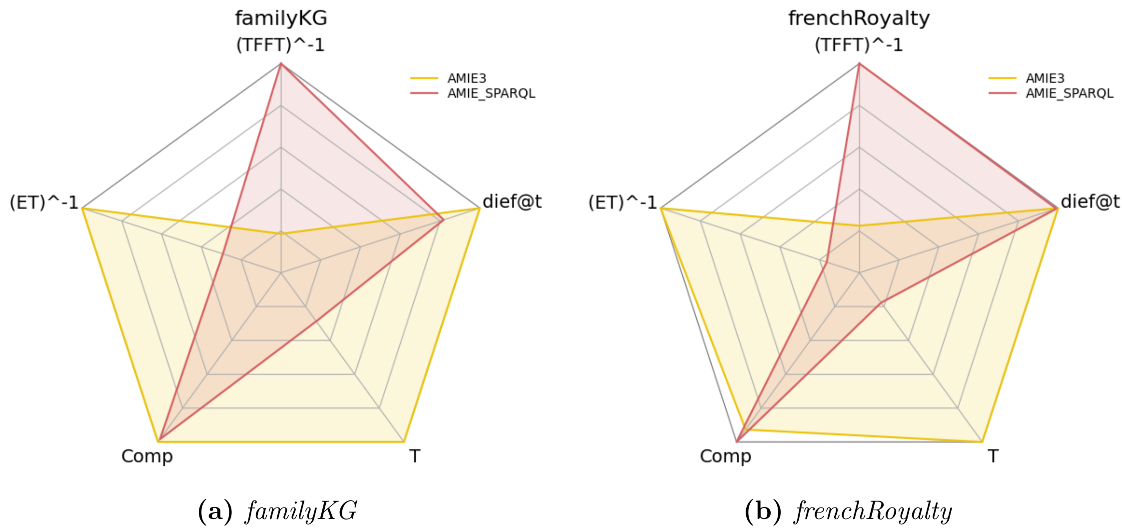


(a) *familyKG* **(b)** *frenchRoyalty*

Figure 6.4: Continuous efficiencies of AMIE3 and AMIE_SPARQL with diefk

Figure 6.4 compares the dief@k score for the algorithms when they produced k=25%, k=50%, k=75%, and k=100% of the rules when mining over the two KGs. **dief@k** score is a measure of area under the curve plotted using the number of

answers produced by an approach over a period of time, the curves in Figure 6.2 and Figure 6.3 in our case. For a fixed **k**, an approach that has a lower dief@k score is better, as a lower dief@k score means, k answers were produced in a shorter period of time, which suggests that the approach is more efficient comparatively. The Figure 6.4 suggests that the continuous efficiency of AMIE3 is better overall compared to our approach.



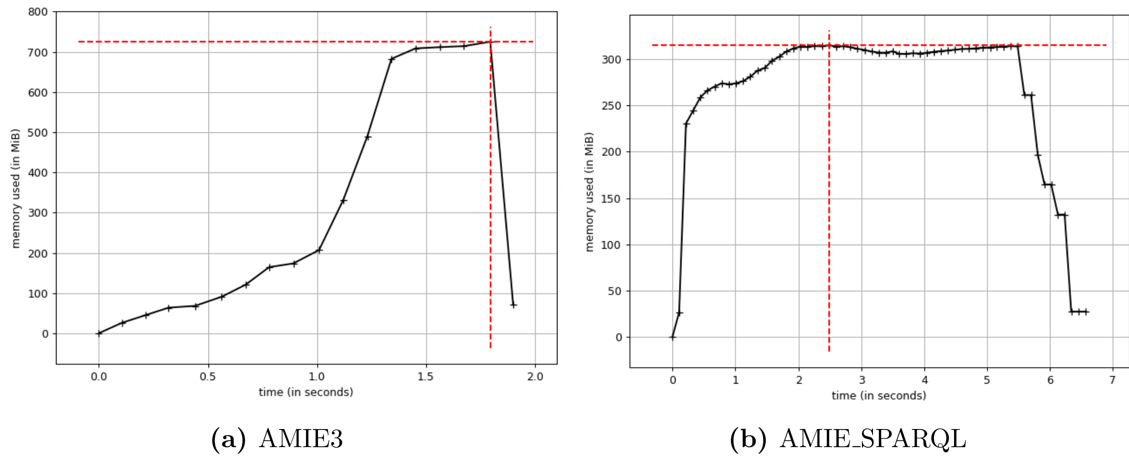
(a) *familyKG* **(b)** *frenchRoyalty*

Figure 6.5: Performances of AMIE3 and AMIE_SPARQL with dieft

Figure 6.5 allow for a more detailed comparison of run-time performance of the AMIE3 and our approach. In the figure, **TFFT**, Time for the first tuple, is the elapsed time when the first rule was mined. This means, that the higher the $TFFT^{-1}$ the faster the approach is producing the first rule. Since, our algorithm doesn't need to spend time downloading the KG and uploading the KG in the memory, in this category, we fair better than AMIE3 in both cases. **ET** is the total elapsed time for the algorithms to finish mining, and thus higher ET^{-1} is better, as the algorithm that finishes faster is better. In this regard, AMIE3 outperforms our algorithm in the experiments that were conducted. **Comp**, completeness, is the percentage of the total number of answers/rules produced by the algorithm. This is discussed in detail in Subsection 6.2.3. **T** is the throughput which is calculated by dividing the total answers produced by the execution time of the algorithms. As both AMIE3 and our approach are producing about same results, the throughput in both cases are high for AMIE3 because of the lower **ET**. **dief@t** is the area under the curve of answers produced over time **t**, where **t** is the minimum execution time of the tested

approaches. Opposite to **dief@k**, the higher the **dief@t**, the better the score, as higher score means that the approach was able to produce higher number of results within time **t**. Seeing that AMIE3 is finishing faster in both of the cases, the **dief@t** scores are calculated with **t** = **ET** of AMIE3. In *familyKG* experiment, AMIE3 is better than our approach. However, in *frenchRoyalty*, **dief@t** scores are almost same, 1138 (AMIE3), and 1127 (AMIE_SPARQL). This is most likely because of the higher slope of AMIE3's answer traces, as seen in Figure 6.3.

6.2.2 Memory Performance Results

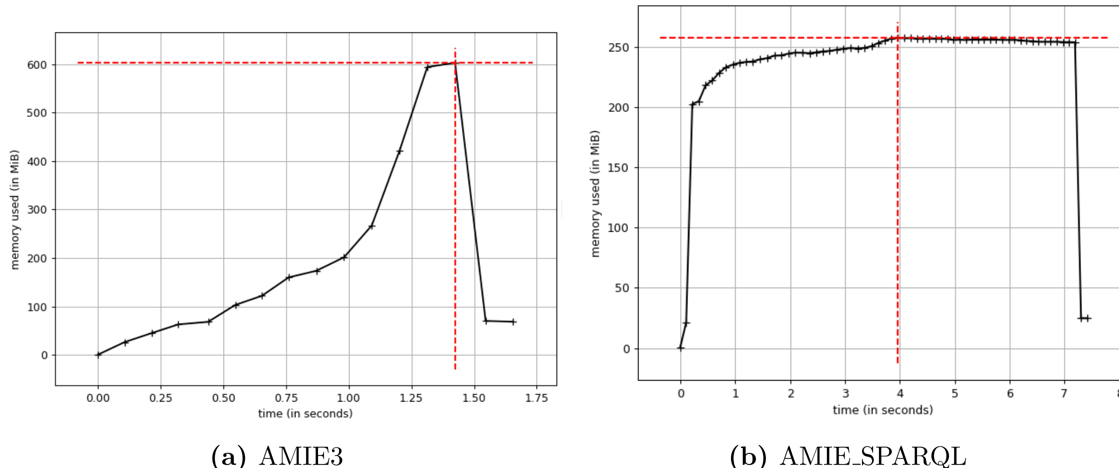


(a) AMIE3 **(b)** AMIE_SPARQL

Figure 6.6: Memory-traces of the algorithms while mining rule over *familyKG*

Figure 6.6a shows that while running AMIE3 over the *familyKG*, a maximum of around 720 MiB was used, with the area under the trace curve of around 600. AMIE_SPARQL performed, as seen in Figure 6.6b, here in this situation a lot better. The maximum memory used by AMIE_SPARQL was less than half of what AMIE3 used, however with the area under the curve of around 1700. The same kind of result could be seen while running the experiments over *frenchRoyalty* KG. The peak memory used by AMIE3 was around 600 MiB, Figure 6.7a, with area under the trace of around 300, whereas AMIE_SPARQL could work with just over 250 MiB, Figure 6.7b, but with higher area under the trace of around 1700. While the area under the trace value for AMIE_SPARQL is higher in comparison to AMIE3, it has to be noted that the implementation is still just in its early stage and no specific optimization of any kind has been done. This goes to show the potential of our

algorithm and hints at the possibility of further improvements, which could make it even faster and more memory efficient.



(a) AMIE3

(b) AMIE_SPARQL

Figure 6.7: Memory-traces of the algorithm while mining rule over *frenchRoyalty*

6.2.3 Discussion

AMIE3 produced on *frenchRoyalty* KG, a total of 190 rules, whereas AMIE_SPARQL produced a total of 206 rules. On closer inspection, it was found that AMIE_SPARQL was outputting some extra rules. This was due to the fact that when outputting rules, only the direct parent of the rule was being checked for pruning. As observed by AMIE+[4], the rule $actedIn(x, y) \wedge directed(x, y) \Rightarrow created(x, y)$ can be derived by either adding $directed(x, y)$ to $actedIn(x, y) \Rightarrow created(x, y)$ or by adding $actedIn(x, y)$ to $directed(x, y) \Rightarrow created(x, y)$. Particularly in output rules of *frenchRoyalty*, there were cases where one of the parent had higher PCA confidence scores than other. As AMIE_SPARQL checks only the direct parent of the rule, the rule $actedIn(x, y) \wedge directed(x, y) \Rightarrow created(x, y)$ is outputted even though $directed(x, y) \Rightarrow created(x, y)$ its indirect parent may have a higher PCA confidence than it. Besides these scenarios, 2 of the rules produced by AMIE3 were not being outputted by AMIE_SPARQL. These were the rules where the head predicate appeared in the rule body, as $successor(b, f) \wedge child(a, f) \Rightarrow child(a, b)$. But AMIE_SPARQL produced only the rule, $successor(b, f) \wedge child(a, b) \Rightarrow child(a, f)$ which had both higher PCA confidence score and higher head coverage. On *familyKG*, AMIE3 produced a total of 367 rules. AMIE_SPARQL on the other hand

produced 361 rules. So 6 rules were missing from AMIE_SPARQL's output. On closer inspection, AMIE_SPARQL found rules with the same predicates and head coverage as these rules, but with different variables. So if AMIE3 mined, for example, both $successor(b, f) \wedge child(a, f) \Rightarrow child(a, b)$ and $successor(b, f) \wedge child(a, b) \Rightarrow child(a, f)$ with same head coverage, AMIE_SPARQL was producing only one of these, leading to a conclusion, that the data-structure could be improved to tell these rules apart.

Chapter 7

Conclusions and Future Work

This chapter summarizes the content of the thesis, addresses the limitations of the work, and provides directions for future research in the area.

7.1 Conclusions

This thesis presents an adaptation of the algorithm proposed by AMIE, AMIE+, and AMIE3. The adaptation is made to circumvent the memory bottleneck of AMIE algorithms and to make the mining of rules over large KGs viable "locally", by using the SPARQL endpoints on which these KGs are hosted. A highly concurrent and configurable implementation of the adaptation was created which was shown experimentally in chapter 6 to improve memory usage over AMIE3, while still producing almost the same rules as mined by AMIE3. A new synchronous hash-based data structure was also implemented to allow for storing the rules and checking their existence very efficiently.

7.2 Limitations and Future Work

Although our mining algorithm could improve the memory performance in comparison drastically, the algorithm still has some issues and limitations as was observed in chapter 6. These include:

- **Use of semantic information**: Currently the algorithm generates SPARQL queries that donot consider the semantic information available in the KG. Incorporating the semantic information, for example, reasoning performed over the KGs considering *T-Box* with entailment regimes while generating the query

could improve the query execution time. This could be considered in the next steps of the work.

- **Run-time Performance**: Although, under default configuration, the algorithm was shown to mine rules in a reasonable time, the time taken was still 3 times worse, even in the small KGs on which we performed our experiments. The run-time of the algorithm at its current state can blow up exponentially when used over very large KGs. Further work has to be done to improve this, one of the ways could involve the optimization of the queries that are executed to find the extension candidates.

Bibliography

[1] Maribel Acosta, Maria-Esther Vidal, and York Sure-Vetter. "Diefficiency metrics: measuring the continuous efficiency of query processing approaches". In: *The Semantic Web–ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II 16*. Springer. 2017, pp. 3–19.

[2] Yang Chen et al. "Ontological Pathfinding: Mining First-Order Knowledge from Large Knowledge Bases". In: *Proceedings of the 2016 ACM SIGMOD international conference on Management of data*. ACM. 2016.

[3] Michael Färber and Achim Rettinger. "Which knowledge graph is best for me?" In: *arXiv preprint arXiv:1809.11099* (2018).

[4] Luis Galárraga et al. "Fast rule mining in ontological knowledge bases with AMIE+". In: *The VLDB Journal* 24.6 (2015), pp. 707–730.

[5] Luis Antonio Galárraga et al. "AMIE: association rule mining under incomplete evidence in ontological knowledge bases". In: *Proceedings of the 22nd international conference on World Wide Web*. 2013, pp. 413–422.

[6] *GlobalInterpreterLock - python-wiki*. URL: https://wiki.python.org/moin/GlobalInterpreterLock (visited on 11/09/2023).

[7] Aidan Hogan et al. *Knowledge Graphs*. English. Synthesis Lectures on Data, Semantics, and Knowledge 22. Springer, 2021. ISBN: 9783031007903. DOI: 10.2200/S01125ED1V01Y202109DSK022. URL: https://kgbook.org/.

[8] Jonathan Lajus, Luis Galárraga, and Fabian Suchanek. "Fast and exact rule mining with AMIE 3". In: *The Semantic Web: 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31–June 4, 2020, Proceedings 17*. Springer. 2020, pp. 36–52.

[9] Christian Meilicke et al. "Anytime Bottom-Up Rule Learning for Knowledge Graph Completion." In: *IJCAI*. 2019, pp. 3137–3143.

[10] Natasha Noy et al. "Industry-scale Knowledge Graphs: Lessons and Challenges: Five diverse technology companies show how it's done". In: *Queue* 17.2 (2019), pp. 48–75.

[11] *Open World Assumption*. URL: https://en.wikipedia.org/wiki/Open-world_assumption (visited on 10/05/2023).

[12] Stefano Ortona, Vamsi Meduri, and Paolo Papotti. “Robust discovery of positive and negative rules in knowledge-bases”. © EURECOM. Personal use of this material is permitted. The definitive version of this paper was published in Technical Report RR-17-333, 15 September 2017 and is available at : MA thesis. 2017.

[13] Luc De Raedt. “Inductive Logic Programming”. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 529–537. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_396. URL: https://doi.org/10.1007/978-0-387-30164-8_396.

[14] Amit Singlal. *Introducing the Knowledge Graph: things, not strings*. 2012. URL: https://blog.google/products/search/introducing-knowledge-graph-things-not/ (visited on 10/05/2023).