Minage de règles rapide, exact et exhaustif dans de larges bases de connaissances

Jonathan Lajus

Fast, Exact, and Exhaustive Rule Mining in Large Knowledge Bases

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626
École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Informatique, données, IA

Thèse présentée et soutenue à Palaiseau, le 16 février 2021, par

JONATHAN LAJUS

Composition du Jury :

Thomas Bonald Professor, Télécom Paris	Président
Paolo Papotti Associate Professor, EURECOM	Rapporteur
Heiko Paulheim Professor, University of Mannheim	Rapporteur
Hannah Bast Professor, University of Freiburg	Examinatrice
Luis Galàrraga Researcher, INRIA Rennes	Examinateur
Fatiha Saïs Professor, Paris Saclay University (LRI)	Examinatrice
Fabian Suchanek Professor, Télécom Paris	Directeur de thèse

To my parents, my family and Minou.

Acknowlegements

Une thèse est un circuit de longue haleine. Tout a commencé il y a cinq ans quand Fabian a accepté de me faire rejoindre l'équipe DIG. A l'époque, j'ai commencé par me mettre dans la roue de Luis durant un stage de 6 mois. Puis après son échappée, il m'a fallu prendre le vent.

Heureusement, je pu faire partie d'un formidable peloton, comprenant Luis, Thomas, Thomas, Julien, Ned, Antoine, Marc, Marie, Pierre-Alexandre, Jean-Louis, Mauro et tant d'autres avec qui j'ai pu partager cette aventure. Je souhaite remercier particulièrement Fabian, qui en bon capitaine de route a toujours su être à l'écoute. Je remercie également Bruno Defude, Bernd Amann et Arnaud Soulet pour leur conseils en cours de route.

Le parcours était bien escarpé, avec des hauts et des bas, parfois en danseuse mais souvent à mouliner. Je ne peux que remercier mes parents, toute ma famille qui m'ont soutenu mais aussi mes colocataires qui m'ont supporté. Sans oublier les Nadine, Samy, Sarah, Simon, Émilie, Romaing, Joris, Seya, Doc, Guillaume, Malfoy et Camille pour les japs et les soirées jeux qui m'ont permis de sortir la tête du guidon.

Bien que les rêves de maillot jaune ont disparu, j'ai pu au moins finir la course. Il a fallu changer de braquet pour cette dernière étape de 153 pages. Je remercie tous les membres du jury de donner de leur temps pour considérer son homologation.

A thesis is a long ride. Everything started, five years ago, when Fabian made me join the team. I first sat in the wheels of Luis during my master thesis. Then, after he made a breakaway, I had to set my own pace.

Fortunately, I was part of a terrific peloton, including Luis, Thomas, Thomas, Julien, Ned, Antoine, Marc, Marie, Pierre-Alexandre, Jean-Louis, Mauro and many others, with who I could share this adventure. I would like to particularly thank Fabian, who made a great raod captain, always receptive and responsive. I would also like to thank Bruno Defude, Bernd Amann and Arnaud Soulet for their valuable input along the road.

The journey was steep, with ups and downs. I have to thank my parents and my whole family for their support, and my roommates who were able to put up with me. I also thank the Nadine, Samy, Sarah, Simon, Émilie, Romaing, Joris, Seya, Doc, Guillaume, Malfoy et Camille for dining out and the game night that made me keep my head over the water.

If the dreams of the yellow jersey have faded away, I did at least finish the race. I had to pick up an higher pace for this last stage of 153 pages. I thank all the member of the jury for the time they spend as they have to decide whether they certify this run.

Contents

Chapter 1

Introduction

1.1 Motivation

When we send a query to Google or Bing, we obtain a set of Web pages. However, in some cases, we also get more information. For example, when we ask "When was Steve Jobs born?", the search engine replies directly with "February 24, 1955". When we ask just for "Steve Jobs", we obtain a short biography, his birth date, quotes, and spouse. All of this is possible because the search engine has a huge repository of knowledge about people of common interest. This knowledge takes the form of a knowledge base (KB).

The KBs used in such search engines are entity-centric: they know individual entities (such as *Steve Jobs*, the *United States*, the *Kilimanjaro*, or the *Max Planck Society*), their semantic classes (such as *SteveJobs is-a computerPioneer, SteveJobs is-a entrepreneur*), relationships between entities (e.g., *SteveJobs founded AppleInc, SteveJobs hasInvented iPhone, SteveJobs hasWonPrize NationalMedalOfTechnology*, etc.) as well as their validity times (e.g., *SteveJobs wasCEOof Pixar [1986,2006]*).

The idea of such KBs is not new. It goes back to seminal work in Artificial Intelligence on universal knowledge bases in the 1980s and 1990s, most notably, the Cyc project [47] at MCC in Austin and the WordNet project [24] at Princeton University. These knowledge collections were hand-crafted and manually curated. In the last ten years, in contrast, KBs are often built automatically by extracting information from the Web or from text documents. Salient projects with publicly available resources include KnowItAll (UW Seattle, [22]), ConceptNet (MIT, [49]), DBpedia (FU Berlin, U Mannheim, & U Leipzig, [46]), NELL (CMU, [12]), BabelNet (La Sapienza, [67]), Wikidata (Wikimedia Foundation, [84]), and YAGO (Telecom Paris & Max Planck Institute, [78]). Commercial interest in KBs has been strongly growing, with projects such as the Google Knowledge Graph [19] (including Freebase [9]), Microsoft's Satori, Amazon's Evi, LinkedIn's Knowledge Graph, and the IBM Watson KB [26]. These KBs contain many millions of entities, organized in hundreds to hundred thousands of semantic classes, and hundred millions of relational facts between entities. Many public KBs are interlinked, forming the Web of Linked Open Data [8].

This large interlinked Knowledge can itself be mined for information. The pattern or correlation found are expressed as rules such as:

$$married(x, y) \wedge livesIn(x, z) \Rightarrow livesIn(y, z)$$

This rule expresses that "If X and Y are married, and X lives in Z, then Y also lives in Z", or in other words, that two married people usually live in the same place. Such rules usually come with confidence scores that express to what degree a rule holds.

Rule mining is the task of automatically finding logical rules in a given KB. It usually focus on outputting the rules of highest quality, i.e the rules that have a confidence score superior to a given user-defined parameter.

The rules can serve several purposes: First, they serve to complete the KB. If we do not know the place of residence of a person, we can propose that the person lives where their spouse lives. Second, they can serve to debug the KB. If the spouse of someone lives in a different city, then this can indicate a problem. Finally, rules are useful in downstream applications such as fact checking [2], ontology alignment [28] or predicting completeness [29].

The difficulty in finding such rules lies in the exponential size of the search space: every relation can potentially be combined with every other relation in a rule. This is why early approaches (such as AMIE [31]) were unable to run on large KBs such as Wikidata in less than a day. Since then, several approaches have resorted to sampling or approximate confidence calculations [30, 94, 63, 14]. The more the approach samples, the faster it becomes, but the less accurate the results will be. Another common technique [63, 58, 94, 66] (from standard inductive logic programming) is to mine not all rules, but only enough rules to cover the positive examples. This, likewise, speeds up the computation, but does not mine all rules that hold in the KB.

1.2 Contribution

The main goal of this thesis is to study optimizations and novel approaches to develop efficient rule mining algorithms that scale to large KBs, compute the quality measures exactly, and are exhaustive with regard to user-defined parameters. Such algorithms can serve as a baseline for other approximate rule mining algorithms, and to produce a gold standard of mined rules.

Preliminaries. In Chapter 2, we define the main notions of entity-centric Knowledge Bases and explain their principal characteristics. Then we introduce the task of Rule Mining as an Inductive Logic Programming problem and describe the multiple rule mining algorithms that apply on Knowledge Bases. This chapter is based on our following tutorial paper:

- Fabian M Suchanek, Jonathan Lajus, Armand Boschin, and Gerhard Weikum. Knowledge representation and rule mining in entity-centric knowledge bases. In *Reasoning Web. Explainable Artificial Intelligence*, pages 110–152. Springer, 2019

AMIE 3. In Chapter 3, we present AMIE 3, an improved version of the rule mining algorithm AMIE+ [30]. In this new version we introduce new optimizations on the computation of the quality measures, allowing AMIE to once again become an exact and exhaustive rule mining algorithm. This new version is also more efficient, allowing AMIE to scale to large KBs that were previously beyond reach. This chapter is based on our following full paper:

- Jonathan Lajus, Luis Galárraga, and Fabian Suchanek. Fast and exact rule mining with AMIE 3. In *European Semantic Web Conference*, pages 36–52. Springer, 2020

Pruning the search space. In Chapter 4, we further address the problem of the exponential search that prevents AMIE from efficiently mining more complex rules, for example rules of size 4 on large KBs. In particular, we study how the decomposition of the rules into smaller entity-centric patterns, the "star patterns", can be used to identify and prune unsatisfiable rules before any costly computation. This chapter is based on unpublished work.

Path rule mining. In Chapter 5, we restrict our study of efficient rule mining to a certain type of rules, the path rules. In particular, we show that all closed path rules can be generated without duplicates using a specific refinement operator, and that the star patterns introduced in Chapter 4 can be used constructively to efficiently prune the search space of path rules. From this, we deduce the "Pathfinder vanilla" algorithm, an exact and exhaustive path rule mining algorithm.

In Chapter 6, we study the consequences of a fundamental dependency in the Pathfinder vanilla algorithm: every rule comes from a unique parent rule. In particular, we show that a parent rule can pass information to its child rules and we use this idea to improve the computation of the quality measures and perform early pruning. Finally, we present the complete "Pathfinder" algorithm, a refined version of the Pathfinder vanilla algorithm which outperforms AMIE on path rules on most datasets, and by multiple orders of magnitudes when mining longer rules. This part is also based on unpublished work.

Rule mining about the real world. In Chapter 7 we present work that can automatically determine the obligatory attributes of a class. For example, our algorithm mines that every singer sings in the real world, even if our KB is largely incomplete. In particular, we introduce a statistical modelization of the incompleteness of the KB and deduce a metric to identify obligatory attributes. This chapter is based on our following full paper:

- Jonathan Lajus and Fabian M. Suchanek. Are all people married? determining obligatory attributes in knowledge bases. In *WWW*, pages 1115–1124. International World Wide Web Conferences Steering Committee, 2018

Conclusion. Chapter 8 concludes this thesis and present further prospects.

Chapter 2

Introduction to Knowledge Bases and Rule Mining

2.1 Introduction

The field of knowledge representation has a long history, and goes back to the early days of Artificial Intelligence. It has developed numerous knowledge representation models, from frames and KL-ONE to recent variants of description logics. The reader is referred to survey works for comprehensive overviews of historical and classical models [73, 76]. In the first part of this chapter, we discuss the knowledge representation that has emerged as a pragmatic consensus in the research community of entity-centric knowledge bases.

In the second part of this chapter, we discuss logical rules on knowledge bases. A logical rule can tell us, e.g., that if two people are married, then they (usually) live in the same city. Such rules can be mined automatically from the knowledge base, and they can serve to correct the data or fill in missing information. We discuss first classical Inductive Logic Programming approaches, and then show how these can be applied to the case of knowledge bases.

2.2 Knowledge Representation

2.2.1 Entities

Entities of Interest

The most basic element of a KB is an *entity*. An entity is any abstract or concrete object of fiction or reality, or, as Bertrand Russell puts it in his

Principles of Mathematics [86]:

Definition 2.2.1 (Entity)**.** An entity is whatever may be an object of thought.

This definition is completely all-embracing. Steve Jobs, the Declaration of Independence of the United States, the Theory of Relativity, and a molecule of water are all entities. Events (such as the French Revolution), are entities, too. An entity does not even have to exist: Harry Potter, e.g., is a fictional entity. Phlogiston was presumed to be the substance that makes up heat. It turned out to not exist – but it is still an entity.

KBs model a part of reality. This means that they choose certain entities of interest, give them names, and put them into a structure. Thus, a KB is a structured view on a selected part of the world. KBs typically model only distinct entities. This cuts out a large portion of the world that consists of variations, flows and transitions between entities. Drops of rain, for instance, fall down, join in a puddle and may be splattered by a passing car to form new drops [75]. KBs will typically not model these phenomena. This choice to model only discrete entities is a projection of reality; it is a grid through which we see only distinct things. Many entities consist of several different entities. A car, for example, consists of wheels, a bodywork, an engine, and many other pieces. The engine consists of the pistons, the valves, and the spark plug. The valves consist again of several parts, and so on, until we ultimately arrive at the level of atoms or below. Each of these components is an entity. However, KBs will typically not be concerned with the lower levels of granularity. A KB might model a car, possibly its engine and its wheels, but most likely not its atoms. In all of the following, we will only be concerned with the entities that a KB models.

Entities in the real world can change gradually. For example, the Greek philosopher Eubilides asks: If one takes away one molecule of an object, will there still be the same object? If it is still the same object, this invites one to take away more molecules until the object disappears. If it is another object, this forces one to accept that two distinct objects occupy the same spatio-temporal location: The whole and the whole without the molecule. A related problem is the question of identity. The ancient philosopher Theseus uses the example of a ship: Its old planks are constantly being substituted. One day, the whole ship has been replaced and Theseus asks, "Is it still the same ship?". To cope with these problems, KBs typically model only atomic entities. In a KB, entities can only be created and destroyed as wholes.

Identifiers and Labels

In computer systems (as well as in writing of any form), we refer to entities by *identifiers*.

Definition 2.2.2 (Identifier)**.** An identifier for an entity is a string of characters that represents the entity in a computer system.

Typically, these identifiers take a human-readable form, such as *Elvis-Presley* for the singer Elvis Presley. However, some KBs use abstract identifiers. Wikidata, e.g., refers to Elvis Presley by the identifier *Q303*, and Freebase by */m/02jq1*. This choice was made so as to be language-independent, and so as to provide an identifier that is stable in time. If, e.g., Elvis Presley reincarnates in the future, then *Q303* will always refer to the original Elvis Presley. It is typically assumed that there exists exactly one identifier per entity in a KB. For what follows, we will not distinguish identifiers from entities, and just talk of entities instead.

Entities have names. For example, the city of New York can be called "city of New York", "Big Apple", or "Nueva York". As we see, one entity can have several names. Vice versa, the same name can refer to several entities. "Paris", e.g., can refer to the city in France, to a city of that name in Texas, or to a hero of Greek mythology. Hence, we need to carefully distinguish *names* – single words or entire phrases – from their *senses* – the entities that they denote. This is done by using labels.

Definition 2.2.3 (Label)**.** A label for an entity is a human-readable string that names the entity.

If an entity has several labels, the labels are called synonymous. If the same label refers to several entities, the label is polysemous. Not all entities have labels. For example, your kitchen chair is clearly an entity, but it probably does not have any particular label. An entity that has a label is called a *named entity*. KBs typically model mainly named entities. There is one other type of entities that appears in KBs: literals.

Definition 2.2.4 (Literal)**.** A literal is a fixed value that takes the form of a string of characters.

Literals can be pieces of text, but also numbers, quantities, or timestamps. For example, the label "Big Apple" for the city of New York is a literal, as is the number of its inhabitants (8,175,133).

2.2.2 Classes

Classes and Instances

KBs model entities of the world. They usually group entities together to form a *class*:

Definition 2.2.5 (Class)**.** A class (also: concept, type) is a named set of entities that share a common trait. An element of that set is called an instance of the class.

Under this definition, the following are classes: The class of singers (i.e., the set of all people who sing professionally), the class of historical events in Latin America, and the class of cities in Germany. Some instances of these classes are, respectively, Elvis Presley, the independence of Argentina, and Berlin. Since everything is an entity, a class is also an entity. It has (by definition) an identifier and a label.

Theoretically, KBs can form classes based on arbitrary traits. We can, e.g., construct the class of singers whose concerts were the first to be broadcast by satellite. This class has only one instance (Elvis Presley). We can also construct the class of left-handed guitar players of Scottish origin, or of pieces of music that the Queen of England likes. There are several theories as to whether humans actually build and use classes, too [51]. Points of discussion are whether humans form crisp concepts, and whether all elements of a concept have the same degree of membership. For the purpose of KBs, however, classes are just sets of entities.

It is not always easy to decide whether something should be modeled as an instance or as a class. We could construct, e.g., for every instance a singleton class that contains just this instance (e.g., the class of all Elvis Presleys). Some things of the world can be modeled both as instances and as classes. A typical example is *iPhone*. If we want to designate the type of smartphone, we can model it as an instance of the class of smartphone brands. However, if we are interested in the iPhones owned by different people and want to capture them individually, then *iPhone* should be modeled as a class. A similar observation holds for abstract entities such as *love*. Love can be modeled as an instance of the class *emotion*, where it resides together with the emotions of *anger*, *fear*, and *joy*. However, when we want to model individual feelings of love, then *love* would be a class. Its instances are the different feelings of love that different people have. It is our choice how we wish to model reality.

Some KBs do not make the distinction between classes and instances (e.g., the SKOS vocabulary, [89]). In these KBs, everything is an entity. There is, however, usually a "is more general than" link between a more special entity and a more general entity. Such a KB may contain, e.g., the knowledge that *iPhone* is more special than *smartphone*, without worrying whether one of them is a class. The distinction between classes and instances adds a layer of granularity. This granularity is used, e.g., to define the domains and ranges of relations, as we shall see in Section 2.2.3.

Taxonomies

Definition 2.2.6 (Subsumption)**.** Class A is a subclass of class B if A is a subset of B.

For example, the class of singers is a subclass of the class of persons, because every singer is a person. We also say that the class of singers is a *specialization* of the class of persons, or that singer *is subsumed by* or *included in* person. Vice versa, we say that person is a *superclass* or a *generalization* of the class of singers. Technically speaking, two equivalent classes are subclasses of each other. This is the way the RDFS standard models subclasses [88]. We say that a class is a *proper subclass* of another class, if the second contains more entities than the first. We use the notion of subclass here to refer to *proper* subclasses only.

It is important not to confuse class inclusion with the relationship between parts and wholes. For example, an arm is a part of the human body. That does not mean, however, that every arm is a human body. Hence, *arm* is not a subclass of *body*. In a similar manner, New York is a part of the US. That does not mean that New York would be a subclass of the US. Neither New York nor the US are classes, so they cannot be subclasses of each other.

Class inclusion is transitive: If A is a subclass of B, and B is a subclass of C, then A is a subclass of C. For example, *viper* is a subclass of *snake*, and *snake* is a subclass of *reptile*. Hence, by transitivity, *viper* is also a subclass of *reptile*. We say that a class is a *direct subclass* of another class, if there is no class in the KB that is a superclass of the former and a subclass of the latter. When we talk about subclasses, we usually mean only direct subclasses. The other subclasses are *transitive subclasses*. Since classes can be included in other classes, they can form an inclusion hierarchy – a taxonomy.

Definition 2.2.7 (Taxonomy)**.** A taxonomy is a directed graph, where the nodes are classes and there is an edge from class X to class Y if X is a proper direct subclass of Y.

The notion of taxonomy is known from biology. Zoological or botanic species form a taxonomy: *tiger* is a subclass of *cat*. *cat* is a subclass of *mammal*, and so on. This principle carries over to all other types of classes. We say, e.g., that *internetCompany* is a subclass of *company*, and that *company* is a subclass of *organization*, etc. Since a taxonomy models proper inclusion, it follows that the taxonomic graph is acyclic: If a class is the subclass of another class, then the latter cannot be a subclass of the former. Thus, a taxonomy is a directed acyclic graph. A taxonomy does not show the transitive subclass edges. If the graph contains transitive edges, we can always remove them. Given a finite directed acyclic graph with transitive edges, the set of direct edges is unique [4].

Transitivity is often essential in applications. For example, consider a question-answering system where a user asks for artists that are married to actors. If the KB only knew about Elvis Presley and Priscilla Presley being in the classes *rockSinger* and *americanActress*, the question could not be answered. However, by reasoning that *rockSingers* are also *singers*, who in turn are *artists* and *americanActresses* being *actresses*, it becomes possible to give this correct answer.

Usually (but not necessarily), taxonomies are connected graphs: Every node in the graph is, directly or indirectly, linked to every other node. Usually, the taxonomies have a single root, i.e., a single node that has no outgoing edges. This node identifies the most general class, of which every other class is a subclass. In zoological KBs, this may be class *animal*. In a person database, it may be the class *person*. In a general-purpose KB, this class has to be the most general possible class. In YAGO and Wordnet, the class is *entity*. In the RDF standard, it is called *resource* [87]. In the OWL standard [90], the highest class that does not include literals is called *thing*.

Some taxonomies have at most one outgoing edge per node. Then, the taxonomy forms a tree. The biological taxonomy, e.g., forms a tree, as does the Java class hierarchy. However, there can be taxonomies where a class has two distinct direct superclasses. For example, if we have the class *singer* and the classes of *woman* and *man*, then the class *femaleSinger* has two superclasses: *singer* and *woman*. Note that it would be wrong to make *singer* a subclass of *man* and *woman* (as if to say that singers can be men or women). This would actually mean that all singers are at the same time men and women.

When a taxonomy includes a "combination class" such as *FrenchFemaleSingers*, then this class can have several superclasses. *FrenchFemaleSingers*, e.g., can have as direct superclasses *FrenchPeople*, *Women*, and *Singers*. In a similar manner, one entity can be an instance of several classes. Albert Einstein, e.g., is an instance of the classes *physicist*, *vegetarian*, and *violinPlayer*.

When we populate a KB with new instances, we usually try to assign them to the most specific suitable class. For example, when we want to place *Bob*

Dylan in our taxonomy, we would put him in the class *americanBluesSinger*, if we have such a class, instead of in the class *person*. However, if we lack more specific information about the instance, then we might be forced to put it into a general class. Some named entity recognizers, e.g., distinguish only between organizations, locations, and people, which means that it is hard to populate more specific classes. It may also happen that our taxonomy is not specific enough at the leaf level. For example, we may encounter a musician who plays the Arabic oud, but our taxonomy does not have any class like *oudPlayer*. Therefore, a class may contain more instances than the union of its subclasses. That is, for a class C with subclasses $C_1, \ldots, C_k$, the invariant is $\cup_{i=1..k} C_k \subseteq C$, but $\cup_{i=1..k} C_k = C$ is often false.

2.2.3 Relations

Relations and Statements

KBs model also *relationships* between entities:

Definition 2.2.8 (Relation)**.** A relationship (also: relation) over the classes $C_1, ..., C_n$ is a named subset of the Cartesian product $C_1 \times ... \times C_n$.

For example, if we have the classes *person*, *city*, and *year*, we may construct the *birth* relationship as a subset of the cartesian product *person*×*city*×*year*. It will contain tuples of a person, their city of birth, and their year of birth. For example, $\langle \mathit{ElvisPresley}, \mathit{Tupelo}, 1935 \rangle \in \mathit{birth}$. In a similar manner, we can construct *tradeAgreement* as a subset of *country*×*country*×*commodity*. This relation can contain tuples of countries that made a trade agreement concerning a commodity. Such relationships correspond to classical relations in algebra or databases.

As always in matters of knowledge representation (or, indeed, informatics in general), the identifier of a relationship is completely arbitrary. We could, e.g., call the *birth* relationship *k42*, or, for that matter, *death*. Nothing hinders us to populate the *birth* relationship with tuples of a person, and the time and place where that person ate an ice cream. However, most KBs aim to model reality, and thus use identifiers and tuples that correspond to real-world relationships.

If $\langle x_1, ..., x_n \rangle \in R$ for a relationship R, we also write $R(x_1, ..., x_n)$. In the example, we write $\mathit{birth}(\mathit{ElvisPresley}, \mathit{Tupelo}, 1935)$. The classes of R are called the *domains* of R. The number of classes n is called the *arity* of R. $\langle x_1, ..., x_n \rangle$ is a *tuple of R*. $R(x_1, ..., x_n)$ is called a *statement*, *fact*, or *record*. The elements $x_1, ..., x_n$ are called the *arguments* of the facts. Finally, a *knowledge base*, in its simplest form, is a set of statements. For example, a KB can contain the relations *birth*, *death* and *marriage*, and thus model some of the aspects of people's lives.

Binary Relations

Definition 2.2.9 (Binary Relation)**.** A binary relation is a relation of arity 2. We note $\mathcal{R}$ the set of binary relations of a KB.

Examples of binary relations are *birthPlace, friendOf*, or *marriedTo.* The first argument of a binary fact is called the *subject*, and the second argument is called the *object* of the fact. The relationships are sometimes called *properties.* Relationships that have literals as objects, and that have at most one object per subject are sometimes called *attributes.* Examples are *hasBirthDate* or *hasISBN.* The *domain* of a binary relation $R \subset A \times B$ is A, i.e., the class from which the subjects are taken. B is called the *range* of R. For example, the domain of *birthPlace* is *person*, and its range is *city.* The *inverse* of a binary relation R is a relation R^{-1}, such that $R^{-1}(y, x)$ iff $R(x, y)$. For example, the inverse relation of *hasNationality* (between a person and a country) is *hasNationality*$^{-1}$ (between a country and a person) – which we could also call *hasCitizen.*

Definition 2.2.10 (Signed relation)**.** The set of signed relations $\widetilde{\mathcal{R}}$ of a KB contains every relation $R \in \mathcal{R}$ of the KB and its respective inverse. R^{-1}

Any n-ary relation R with $n > 2$ can be split into n binary relations. This works as follows. Assume that there is one argument position i that is a *key*, i.e., every fact $R(x_1, ..., x_n)$ has a different value for x_i. In the previously introduced 3-ary *birth* relationship, which contains the person, the birth place, and the birth date, the person is the key: every person is born only once at one place. Without loss of generality, let the key be at position $i = 1$. We introduce binary relationships $R_2, ..., R_n$. In the example, we introduce *birthPlace* for the relation between the person and the birth place, and *birthDate* for the relation between the person and the birth year. Every fact $R(x_1, ..., x_n)$ gets rewritten as $R_2(x_1, x_2), R_3(x_1, x_3), R_4(x_1, x_4), ..., R_n(x_1, x_n)$. In the example, the fact *birth(Elvis,Tupelo,1935)* gets rewritten as *birthPlace(Elvis,Tupelo)* and *birthDate(Elvis,1935).* Now assume that a relation R has no key. As an example, consider again the *tradeAgreement* relationship. Obviously, there is no key in this relationship, because any country can make any number of trade-agreements on any commodity. We introduce binary relationships $R_1, ...R_n$ for every argument position of R. For *tradeAgreement*, these could be *country1*, *country2* and *tradeCommodity.* For each fact of R, we introduce a new entity, an *event entity.* For example, if the US and Brazil make a trade-agreement on coffee, *tradeAgreement(Brazil,US,Coffee)*, then we create *coffeeAgrBrUs*. This entity represents the fact that these two countries made this agreement. In general, every fact $R(x_1, ..., x_n)$ gives rise to an event entity $e_{x1,...,xn}$. Then, every fact $R(x_1, ..., x_n)$ is rewritten as $R_1(e_{x1,...,xn}, x_1), R_2(e_{x1,...,xn}, x_2), ..., R_n(e_{x1,...,xn}, x_n)$. In the example, *country1(coffeeAgrBrUs, Brazil), country2(coffeeAgrBrUs, US), tradeCommodity(coffeeAgrBrUs, Coffee)*. This way, any n-ary relationship with $n > 2$ can be represented as binary relationships. For $n = 1$, we can always invent a binary relation *hasProperty*, and use the relation as an additional argument. For example, instead of *male(Elvis)*, we can say *hasProperty(Elvis, male)*.

The advantage of binary relationships is that they can express facts even if one of the arguments is missing. If, e.g., we know only the birth year of Steve Jobs, but not his birth place, then we cannot make a fact with the 3-ary relation $birth \subset person \times city \times year$. We have to fill the missing arguments, e.g., with *null* values. If the relationship has a large arity, many of its arguments may have to be null values. In the case of binary relationships, in contrast, we can easily state *birthDate(SteveJobs, 1955)*, and omit the *birthPlace* fact. Another disadvantage of n-ary relationships is that they do not allow adding new pieces of information a posteriori. If, e.g., we forgot to declare the astrological ascendant as an argument to the 3-ary relation *birth*, then we cannot add the ascendant for Steve Job's birth without modifying the relationship. In the binary world, in contrast, we can always add a new relationship *birthAscendant*. Thus, binary relationships offer more flexibility. This flexibility can be a disadvantage, because it allows adding incomplete information (e.g., a birth place without a birth date). However, since knowledge bases are often inherently incomplete, binary relationships are usually the method of choice.

Functions

Definition 2.2.11 (Function)**.** A function is a binary relation that has for each subject at most one object.

Typical examples for functions are *birthPlace* and *hasLength*: Every person has at most one birth place and every river has at most one length. The relation *ownsCar*, in contrast, is not a function, because a (rich) person can own multiple cars. In our terminology, we call a relation a function also if it has no objects for certain subjects, i.e., we include partial functions (such as *deathDate*).

Some relations are *functions in time*. This means that the relation can have several objects, but at each point of time, only one object is valid. A typical example is *isMarriedTo*. A person can go through several marriages, but can only have one spouse at a time (in most systems). Another example is *hasNumberOfInhabitants* for cities. A city can grow over time, but at any point of time, it has only a single number of inhabitants. Every function is a function in time.

A binary relation is an *inverse function*, if its inverse is a function. Typical examples are *hasCitizen* (if we do not allow double nationality) or *hasEmail-Address* (if we talk only about personal email addresses that belong to a single person). Some relations are both functions and inverse functions. These are identifiers for objects, such as the social security number. A person has exactly one social security number, and a social security number belongs to exactly one person. Functions and inverse functions play a crucial role in entity matching: If two KBs talk about the same entity with different names, then one indication for this is that both entities share the same object of an inverse function. For example, if two people share an email address in a KB about customers, then the two entities must be identical.

Some relations are "nearly functions", in the sense that very few subjects have more than one object. For example, most people have only one *nation-ality*, but some may have several. This idea is formalized by the notion of *functionality* [79]. The functionality of a relation r in a KB is the number of subjects, divided by the number of facts with that relation:

$$fun(r) := \frac{|\{x : \exists y : r(x, y)\}|}{|\{x, y : r(x, y)\}|}$$

The functionality is always a value between 0 and 1, and it is 1 if r is a function. It is undefined for an empty relation.

We usually have the choice between using a relation and its inverse relation. For example, we can either have a relationship *isCitizenOf* (between a person and their country) or a relationship *hasCitizen* (between a country and its citizens). Both are valid choices. In general, KBs tend to choose the relation with the higher functionality, i.e., where the subject has fewer objects. In the example, the choice would probably be *isCitizenOf*, because people have fewer citizenships than countries have citizens. The intuition is that the facts should be "facts about the subject". For example, the fact that the author of this thesis is a citizen of France is clearly an important property for this author.Vice versa, the fact that France is fortunate enough to count this author among its citizens is a much less important property of France (it does not appear on the Wikipedia page of France).

2.2.4 Completeness and Correctness

Knowledge bases model only a part of the world. In order to make this explicit, one imagines a complete knowledge base $\mathcal{W}$ that contains all entities and facts of the real world in the domain of interest. A given KB $\mathcal{K}$ is *correct*, if $\mathcal{K} \subseteq \mathcal{W}$. Usually, KBs aim to be correct. In real life, however, large KBs tend to contain also erroneous statements. YAGO, e.g., has an accuracy of 95%, meaning that 95% of its statements are in $\mathcal{W}$ (or, rather, in Wikipedia, which is used as an approximation of $\mathcal{W}$). This means that YAGO still contains hundreds of thousands of wrong statements. For most other KBs, the degree of correctness is not even known.

A knowledge base is *complete*, if $\mathcal{W} \subseteq \mathcal{K}$ (always staying within the domain of interest). The *closed world assumption* (CWA) is the assumption that the KB at hand is complete. Thus, the CWA says that any statement that is not in the KB is not in $\mathcal{W}$ either. In reality, however, KBs are hardly ever complete. Therefore, KBs typically operate under the *open world assumption* (OWA), which says that if a statement is not in the KB, then this statement can be either true or false in the real world.

KBs usually do not model negative information. They may say that Caltrain serves the city of San Francisco, but they will not say that this train does not serve the city of Moscow. While incompleteness tells us that some facts may be missing, the lack of negative information prevents us from specifying which facts are missing because they are false. This poses considerable problems, because the absence of a statement does not allow any conclusion about the real world [71].

2.2.5 The Semantic Web

The common exchange format for knowledge bases is RDF/RDFS [87]. It specifies a syntax for writing down statements with binary relations. Most notably, it prescribes URIs as identifiers, which means that entities can be identified in a globally unique way. To query such RDF knowledge bases, one can use the query language SPARQL [91]. SPARQL borrows its syntax from SQL, and allows the user to specify graph patterns, i.e., triples where some components are replaced by variables. For example, we can ask for the birth date of Elvis by saying "SELECT ?birthdate WHERE { ⟨Elvis⟩ ⟨bornOnDate⟩ ?birthdate }".

To define semantic constraints on the data, RDF is extended by OWL [90]. This language allows specifying constraints such as functions or disjointness of classes, as well as more complex axioms. The formal semantics of these axioms is given by Description Logics [7]. These logics distinguish facts about instances from facts about classes and axioms. The facts about instances are called the *A-Box* ("Assertions"), and the class facts and axioms are called the *T-Box* ("Theory"). Sometimes, the term *ontology* is used to mean roughly the same as *T-Box*. Description Logics allow for automated reasoning on the data.

Many KBs are publicly available online. They form what is known as the *Semantic Web*. Some of these KBs talk about the same entities – with different identifiers. The Linked Open Data project [8] aims to establish links between equivalent identifiers, thus weaving all public KBs together into one giant knowledge graph.

Knowledge can also be represented in the form of rules. For example, if a doctoral student is advised by a professor, then the university of graduation will be the employer of the professor. Again, this type of knowledge representation is well covered in classical works [73, 47], and recent approaches have turned to using it for KBs [31, 30, 14]. This type of intensional knowledge is what we will now discuss in the next section.

2.3 Rule Mining

2.3.1 Rules

Once we have a knowledge base, it is interesting to look out for *patterns* in the data. For example, we could notice that if some person A is married to some person B, then usually B is also married to A (symmetry of marriage). Or we could notice that, if, in addition, A is the parent of some child, then B is usually also a parent of that child (although not always).

We usually write such rules using the syntax of first-order logic. For example, we would write the previous rules as:

$$\mathit{marriedTo}(x, y) \Rightarrow \mathit{marriedTo}(y, x)$$

$$\mathit{marriedTo}(x, y) \wedge \mathit{hasChild}(x, z) \Rightarrow \mathit{hasChild}(y, z)$$

Such rules have several applications: First, they can help us complete the KB. If, e.g., we know that Elvis Presley is married to Priscilla Presley, then we can deduce that Priscilla is also married to Elvis – if the fact was missing. Second, the rules can help us disambiguate entities and correct errors. For example, if Elvis has a child Lisa, and Priscilla has a different child Lisa, then our rule could help find out that the two Lisa's are actually a single entity. Finally, those frequent rules give us insight about our data, biases in the data, or biases in the real world. For example, we may find that European presidents are usually male or that Ancient Romans are usually dead. These two rules are examples of rules that have not just variables, but also entities:

$$\mathit{type}(x, \mathit{AncientRoman}) \Rightarrow \mathit{dead}(x)$$

We are now interested in discovering such rules automatically in the data. This process is called Rule Mining. Let us start with some definitions. The components of a rule are called atoms:

Definition 2.3.1 (Atom)**.** An atom is of the form $r(t_1, \ldots, t_n)$, where r is a relation of arity n (for KBs, usually $n = 2$) and $t_1, \ldots t_n$ are either variables or entities.

In our example, $\mathit{marriedTo}(x, y)$ is an atom, as is $\mathit{marriedTo}(\mathit{Elvis}, y)$. We say that an atom is *instantiated*, if it contains at least one entity. We say that it is *grounded*, if it contains only entities and no variables. A *conjunction* is a set of atoms, which we write as $\vec{A} = A_1 \wedge ... \wedge A_n$. We are now ready to combine atoms to rules:

Definition 2.3.2 (Rule)**.** A Horn rule (rule, for short) is a formula of the form $\vec{B} \Rightarrow h$, where $\vec{B}$ is a conjunction of atoms, and h is an atom. $\vec{B}$ is called the body of the rule, and h its head.

For example, $\mathit{marriedTo}(x, y) \Rightarrow \mathit{marriedTo}(y, x)$ is a rule. Such a rule is usually read as "If x is married to y, then y is married to x". In order to apply such a rule to specific entities, we need the notion of a *substitution*:

Definition 2.3.3 (Substitution)**.** A substitution is a function that maps variables to entities or to other variables.

For example, a substitution σ can map $\sigma(x) = \mathit{Elvis}$ and $\sigma(y) = z$ – but not $\sigma(\mathit{Elvis}) = z$. A substitution can be generalized straightforwardly to atoms, sets of atoms, and rules: if $\sigma(x) = \mathit{Elvis}$, then $\sigma(\mathit{marriedTo}(\mathit{Priscilla}, x)) = \mathit{marriedTo}(\mathit{Priscilla, Elvis})$. With this, an *instantiation of a rule* is a variant of the rule where all variables have been substituted by entities (so that all atoms are grounded). If we substitute $x = \mathit{Elvis}$ and $y = \mathit{Priscilla}$ in our example rule, we obtain the following instantiation:

$$\mathit{marriedTo(Elvis,\ Priscilla)} \Rightarrow \mathit{marriedTo(Priscilla,\ Elvis)}$$

Thus, an instantiation of a rule is an application of the rule to one concrete case. Let us now see what rules can predict:

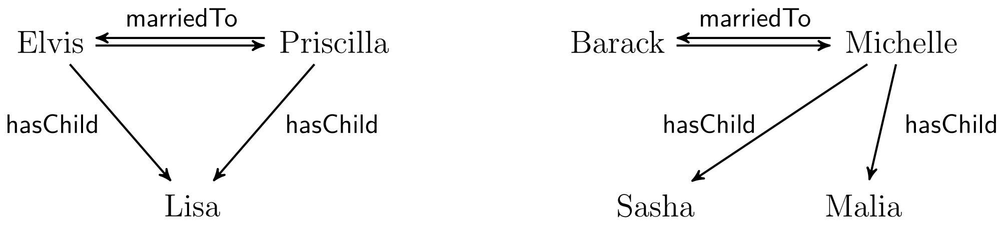

Figure 2.1 – Example KB

Definition 2.3.4 (Prediction of a rule)**.** The predictions P of a rule $\vec{B} \Rightarrow h$ in a KB $\mathcal{K}$ are the head atoms of all instantiations of the rule where the body atoms appear in $\mathcal{K}$. We write $\mathcal{K} \wedge (\vec{B} \Rightarrow h) \models P$. The predictions of a set of rules are the union of the predictions of each rule.

For example, consider the KB in Figure 2.1. The predictions of the rule $\mathit{marriedTo}(x, y) \wedge \mathit{hasChild}(y, z) \Rightarrow \mathit{hasChild}(x, z)$ are *hasChild(Priscilla, Lisa), hasChild(Elvis, Lisa), hasChild(Barack, Sasha), hasChild(Barack, Malia), hasChild(Michelle, Sasha), hasChild(Michelle, Malia).* This is useful, because two of these facts are not yet in the KB.

Logic. From a logical perspective, all variables in a rule are implicitly universally quantified (over every entity defined in the KB). Thus, our example rule is more explicitly written as

$$\forall x, y, z : \mathit{marriedTo}(x, y) \wedge \mathit{hasChild}(y, z) \Rightarrow \mathit{hasChild}(x, z)$$

It can be easily verified that such a rule is equivalent to the following disjunction:

$$\forall x, y, z : \neg \mathit{marriedTo}(x, y) \vee \neg \mathit{hasChild}(y, z) \vee \mathit{hasChild}(x, z)$$

While every Horn rule corresponds to a disjunction with universally quantified variables, not every such disjunction corresponds to a Horn rule. Only those disjunctions with exactly one positive atom correspond to Horn rules. In principle, we could mine arbitrary disjunctions, and not just those that correspond to Horn rules. We could even mine arbitrary first-order expressions, such as $\forall x : \mathit{person}(x) \Rightarrow \neg(\mathit{underage}(x) \wedge \mathit{adult}(x))$. For simplicity, we stay with Horn rules in what follows, and point out when an approach can be generalized to disjunctions or arbitrary formulae.

2.3.2 Rule Mining

Inductive Logic Programming

We now turn to mining rules automatically from a KB. This endeavor is based on *Inductive Reasoning.* To reason by induction is to expect that events that always appeared together in the past will always appear together in the future. For example, inductive reasoning could tell us: "All life forms we have seen so far need water. Therefore, all life forms in general need water.". This is the fundamental principle of empirical science: the generalization of past experiences to a scientific theory. Of course, inductive reasoning can never deliver the logical certitude of deductive reasoning. This is illustrated by Bertrand Russel's analogy of the turkey [72]: The turkey is fed every day by its owner, and so it comes to believe that the owner will always feed the turkey – which is true only until Christmas day. The validity and limitations of modeling the reality using inductive reasoning are a debated topic in philosophy of science. For more perspectives on the philosophical discussions, we refer the reader to [36] and [39]. In the setting of KBs, inductive reasoning is formalized as *Inductive Logic Programming* [65, 74, 60]:

Definition 2.3.5 (Inductive Logic Programming)**.** Given a background knowledge $\mathcal{B}$ (in general, any first order logic expression; in our case: a KB), a set of positive example facts E^+, and a set of negative example facts E^-, Inductive Logic Programming (ILP) is the task of finding an hypothesis $\mathfrak{h}$ (in general, a set of first order logic expressions; in our case: a set of rules) such that $\forall e^+ \in E^+ : \mathcal{B} \wedge \mathfrak{h} \models e^+$ and $\forall e^- \in E^- : \mathcal{B} \wedge \mathfrak{h} \not\models e^-$.

This means that the rules we seek have to predict all positive examples (they have to be *complete*), and they may not predict a negative example (they have to be *correct*). For example, consider again the KB from Figure 2.1 as background knowledge, and let the sets of examples be:

$$E^+ = \{ \text{ } \textit{isMarriedTo(Elvis, Priscilla), isMarriedTo(Priscilla, Elvis),} \\ \textit{isMarriedTo(Barack, Michelle), isMarriedTo(Michelle, Barack)}\}$$

$$E^- = \{ \text{ } \textit{isMarriedTo(Elvis, Michelle), isMarriedTo(Lisa, Barack),} \\ \textit{isMarriedTo(Sasha, Malia)}\}$$

Now consider the following hypothesis:

$$\mathfrak{h} = \{\textit{isMarriedTo}(x, y) \Rightarrow \textit{isMarriedTo}(y, x)\}$$

This hypothesis is complete, as every positive example is a prediction of the rule, and it is correct, as no negative example is predicted.

The attentive reader will notice that the difficulty is now to correctly determine the sets of positive and negative examples. In the ideal case the positive examples should contain any fact that is true in the real world and the negative examples contain any other fact. Thus, in a correct KB, every fact is a positive example.

Definition 2.3.6 (Rule Mining)**.** Given a KB, Rule Mining is the ILP task with the KB as background knowledge, and every single atom of the KB as a positive example.

This means that the rule mining will find several rules, in order to explain all facts of the KB. Three problems remain: First, we have to define the set of negative examples. Second, we have to define what types of rules we are interested in. Finally, we have to adapt our mining to cases where the rule does not always hold.

The Set of Negative Examples

Rule mining needs negative examples (also called *counter-examples*). The problem is that KBs usually do not contain negative information. We can think of different ways to generate negative examples.

Closed World Assumption. The Closed World Assumption (CWA) says that any statement that is not in the KB is wrong (Section 2.2.4). Thus, under the Closed-World Assumption, any fact that is not in the KB can serve as a negative example. The problem is that these may be exactly the facts that we want to predict. In our example KB from Figure 2.1, we may want to learn the rule $\mathit{marriedTo}(x, y) \wedge \mathit{hasChild}(y, z) \Rightarrow \mathit{hasChild}(x, z)$. For this rule, the fact *hasChild(Barack, Malia)* is a counter-example. However, this fact is exactly what we want to predict, and so it would be a counter-productive counter-example.

Open World Assumption. Under the Open-World Assumption (OWA), any fact that is not in the KB can be considered either a negative or a positive example (see again Section 2.2.4). Thus the OWA does not help in establishing counter-examples. Without counter-examples, we can learn any rule. For example, in our KB, the rule *type(x, person)* $\Rightarrow$ *marriedTo(x, Barack)* has a single positive example (for $x =$ *Michelle*), and no counter-examples under the Open World Assumption. Therefore, we could deduce that everyone is married to Barack.

Partial Completeness Assumption. Another strategy to generate negative examples is to assume that entities are complete for the relations they already have. For example, if we know that Michelle has the children Sasha and Malia, then we assume (much like Barack) that Michelle has no other children. If, in contrast, Barack does not have any children in the KB, then we do not conclude anything. This idea is called the Partial-Completeness Assumption (PCA) or the Local Closed World Assumption [31]. It holds trivially for functions (such as *hasBirthDate*), and usually [30] for relations with a high functionality (such as *hasNationality*). The rationale is that if the KB curators took the care to enter some objects for the relation, then they will most likely have entered all of them, if there are few of them. In contrast, the assumption does usually not hold for relations with low functionality (such as *starsInMovie*). Fortunately, relations usually have a higher functionality than their inverses (see Section 2.2.3). If that is not the case, we can apply the PCA to the object of the relation instead.

Random Examples. Another strategy to find counter-examples is to generate random statements [59]. Such random statements are unlikely to be correct, and can thus serve as counter-examples. This is one of the methods used by DL-Learner [38]. It is not easy to generate helpful random counter-examples. If, e.g., we generate the random negative example *marriedTo(Barack,USA)*, then it is unlikely that a rule will try to predict this example. Thus, the example does not actually help in filtering out any rule. The challenge is hence to choose counter-examples that are false, but still reasonable. The authors of [63] describe a method to sample negative statements about semantically connected entities by help of the PCA.

The Language Bias

After solving the problem of negative examples, the next question is what kind of rules we should consider. This choice is called the *language bias*, because it restricts the "language" of the hypothesis. We have already limited ourselves to Horn Rules, and in practice we even restrict ourselves to connected and closed rules.

Definition 2.3.7 (Connected rules)**.** Two atoms are connected if they share a variable, and a rule is connected if every non-ground atom is transitively connected to one another.

For example, the rule *presidentOf(x, America)* $\Rightarrow$ *hasChild(Elvis, y)* is not connected. It is an uninteresting and most likely wrong rule, because it makes a prediction about arbitrary y.

Definition 2.3.8 (Closed rules)**.** A rule is closed if every variable appears in at least two atoms.

For example the rule *marriedTo(x, y)* $\wedge$ *worksAt(x, z)* $\Rightarrow$ *marriedTo(y, x)* is not closed. It has a "dangling edge" that imposes that x works somewhere. While such rules are perfectly valid, they are usually less interesting than the more general rule without the dangling edge.

Finally, one usually imposes a limit on the number of atoms in the rule. Rules with too many atoms tend to be very convoluted [30]. That said, mining rules without such restrictions is an interesting field of research, and we will come back to it in Chapters 4 to 6.

Support and Confidence

One problem with classical ILP approaches is that they will find rules that apply to very few entities, such as *marriedTo(x, Elvis)* $\Rightarrow$ *hasChild(x, Lisa)*. To avoid this type of rules, we define the *support* of a rule:

Definition 2.3.9 (Support)**.** The support of a rule in a KB is the number of positive examples predicted by the rule.

Usually, we are interested only in rules that have a support higher than a given threshold (say, 100). Alternatively, we can define a relative version of support, the *head coverage* [31], which is the number of positive examples predicted by the rule divided by the number of all positive examples with the same relation. Another problem with classical ILP approaches is that they will not find rules if there is a single counter-example. To mitigate this problem, we define the *confidence*:

Definition 2.3.10 (Confidence)**.** The confidence of a rule is the number of positive examples predicted by the rule (i.e., the support of the rule), divided by the number of examples predicted by the rule.

This notion depends on how we choose our negative examples. For instance, under the CWA, the rule $\mathit{marriedTo}(x, y) \wedge \mathit{hasChild}(y, z) \Rightarrow \mathit{hasChild}(x, z)$ has a confidence of 4/6 in Figure 2.1. We call this value the *standard confidence*. Under the PCA, in contrast, the confidence for the example rule is 4/4. We call this value the *PCA confidence*. While the standard confidence tends to "punish" rules that predict many unknown statements, the PCA confidence will permit more such rules. We present in Appendix A the exact mathematical formula of these measures.

In general, the support of a rule quantifies its completeness, and the confidence quantifies its correctness. A rule with low support and high confidence indicates a conservative hypothesis and may be overfitting, i.e. it will not generalize to new positive examples. A rule with high support and low confidence, in contrast, indicates a more general hypothesis and may be overgeneralizing, i.e., it does not generalize to new negative examples. In order to avoid these effects we are looking for a trade-off between support and confidence.

Definition 2.3.11 (Frequent Rule Mining)**.** Given a KB $\mathcal{K}$, a set of positive examples (usually $\mathcal{K}$), a set of negative examples (usually according to an assumption above) and a language of rules, Frequent rule mining is the task of finding all rules in the language with a support and a level of confidence superior to given thresholds.

2.3.3 Rule Mining Approaches

Using substitutions (see Definition 2.3.3), we can define a syntactical order on rules:

Definition 2.3.12 (Rule order)**.** A rule $R \equiv (\vec{B} \Rightarrow h)$ subsumes a rule $R' \equiv (\vec{B'} \Rightarrow h')$, or R is "more general than" R', or R' "is more specific than" R, if there is a substitution σ such that $\sigma(\vec{B}) \subseteq \vec{B'}$ and $\sigma(h) = h'$. If both rules subsume each other, the rules are called equivalent.

For example, consider the following rules:

$$\begin{cases} hasChild(x,y) \Rightarrow hasChild(z,y) & (R_0) \\ hasChild(Elvis,y) \Rightarrow hasChild(Priscilla,y) & (R_1) \\ hasChild(x,y) \Rightarrow hasChild(z,Lisa) & (R_2) \\ hasChild(x,y) \wedge marriedTo(x,z) \Rightarrow hasChild(z,y) & (R_3) \\ marriedTo(v_1,v_2) \wedge hasChild(v_1,v_3) \Rightarrow hasChild(v_2,v_3) & (R_4) \\ hasChild(x,y) \wedge marriedTo(z,x) \Rightarrow hasChild(z,y) & (R_5) \end{cases}$$

The rule R_0 is more general than the rule R_1, because we can rewrite the variables x and z to *Elvis* and *Priscilla* respectively. However R_0 and R_2 are incomparable as we cannot choose to bind only one y and not the other in R_0. The rules R_3, R_4 and R_5 are more specific than R_0. Finally R_3 is equivalent to R_4 but not to R_5.

Proposition 2.3.13 (Prediction inclusion)**.** If a rule R is more general than a rule R', then the predictions of R' on a KB are a subset of the predictions of R. As a corollary, R' cannot have a higher support than R.

This observation gives us two families of rule mining algorithms: top-down rule mining starts from very general rules and specializes them until they become too specific (i.e., no longer meet the support threshold).

Bottom-up rule mining, in contrast, starts from multiple ground rules and generalizes them until the rules become too general (i.e., too many negative examples are predicted).

Top-Down Rule Mining

The concept of specializing a general rule to more specific rules can be traced back to [74] in the context of an exact ILP task (under the CWA). Such approaches usually employ a *refinement operator*, i.e. a function that takes a rule (or a set of rules) as input and returns a set of more specific rules. For example, a refinement operator could take the rule $\mathit{hasChild}(y, z) \Rightarrow \mathit{hasChild}(x, z)$ and produce the more specific rule $\mathit{marriedTo}(x, y) \wedge \mathit{hasChild}(y, z) \Rightarrow \mathit{hasChild}(x, z)$. This process is iterated, and creates a set of rules that we call the *search space* of the rule mining algorithm. On the one hand, the search space should contain every rule of a given rule mining task, so as to be complete. On the other hand, the smaller the search space is, the more efficient the algorithm is.

Usually, the search space is *pruned*, i.e., less promising areas of the search space are cut away. For example, if a rule does not have enough support, then any refinement of it will have even lower support (Proposition 2.3.13). Hence, there is no use refining this rule.

AMIE. AMIE [31] is a top-down rule mining algorithm that aims to mine any connected rule composed of binary atoms for a given support and minimum level of confidence in a KB. AMIE starts with rules composed of only a head atom for all possible head atoms (e.g., ⇒ *marriedTo(x, y)*). It uses three refinement operators, each of which adds a new atom to the body of the rule.

The first refinement operator, `addDanglingAtom`, adds an atom composed of a variable already present in the input rule and a new variable.

$$\begin{array}{lrll}
\text{Some refinements of:} & & \Rightarrow hasChild(z, y) & (R_h) \\
\text{are:} & \left\{\begin{array}{r} hasChild(x, y) \\ marriedTo(x, z) \\ marriedTo(z, x) \end{array}\right. & \begin{array}{l} \Rightarrow hasChild(z, y) \\ \Rightarrow hasChild(z, y) \\ \Rightarrow hasChild(z, y) \end{array} & \begin{array}{l} (R_0) \\ (R_a) \\ (R_b) \end{array}
\end{array}$$

The second operator, `addInstantiatedAtom`, adds an atom composed of a variable already present in the input rule and an entity of the KB.

$$\begin{array}{lrll}
\text{Some refinements of:} & & \Rightarrow hasChild(Priscilla, y) & (R'_h) \\
\text{are:} & \left\{\begin{array}{r} hasChild(Elvis, y) \\ hasChild(Priscilla, y) \\ marriedTo(Barack, y) \end{array}\right. & \begin{array}{l} \Rightarrow hasChild(Priscilla, y) \\ \Rightarrow hasChild(Priscilla, y) \\ \Rightarrow hasChild(Priscilla, y) \end{array} & \begin{array}{l} (R_1) \\ (R_\top) \\ (R_\bot) \end{array}
\end{array}$$

The final refinement operator, `addClosingAtom`, adds an atom composed of two variables already present in the input rule.

$$\begin{array}{lrll}
\text{Some refinements of:} & marriedTo(x,z) & \Rightarrow hasChild(z,y) & (R_a) \\
\text{are:} \quad \left\{\begin{array}{l} \\ \\ \\ \end{array}\right. & \begin{array}{r} hasChild(x,y) \wedge marriedTo(x,z) \\ marriedTo(z,y) \wedge marriedTo(x,z) \\ marriedTo(x,z) \wedge marriedTo(x,z) \end{array} & \begin{array}{l} \Rightarrow hasChild(z,y) \\ \Rightarrow hasChild(z,y) \\ \Rightarrow hasChild(z,y) \end{array} & \begin{array}{l} (R_3) \\ (R_\alpha) \\ (R_a^2) \end{array}
\end{array}$$

As every new atom added by an operator contains at least a variable present in the input rule, the generated rules are connected. The last operator is used to close the rules (for example R_3), although it may have to be applied several times to actually produce a closed rule (cf. Rules R_α or R_a^2).

The AMIE algorithm works on a queue of rules. Initially, the queue contains one rule of a single head atom for each relation in the KB. At each step, AMIE dequeues the first rule, and applies all three refinement operators. The resulting rules are then pruned: First, any rule with low support (such as $R_\perp$) is discarded. Second, different refinements may generate equivalent rules (using the closing operator on R_0 or R_a, e.g., generates among others two equivalent "versions" of R_3). AMIE prunes out these equivalent versions. AMIE+ [30] also detects equivalent atoms as in $R_\top$ or R_a^2 and rewrites or removes those rules. There are a number of other, more sophisticated pruning strategies that estimate bounds on the support or confidence. The rules that survive this pruning process are added to the queue. If one of the rules is a closed rule with a high confidence, it is also output as a result. In this way, AMIE enumerates the entire search space.

The top-down rule mining method is generic, but its result depends on the initial rules and on the refinement operators. The operators directly impact the language of rules we can mine (see Section 2.3.2) and the performance of the method. We can change the refinement operators to mine a completely different language of rules. For example, if we don't use the `addInstantiatedAtom` operator, we restrict our search to any rule without instantiated atoms, which also drastically reduce the size of the search space[1].

Apriori Algorithm. There is an analogy between top-down rule mining and the Apriori algorithm [1]. The Apriori algorithm considers a set of transactions (sales, products bought in a supermarket), each of which is a set of items (items bought together, in the supermarket analogy). The goal of the Apriori algorithm is to find a set of items that are frequently bought together.

[1]Let $|\mathcal{K}|$ be the number of facts and $|r(\mathcal{K})|$ the number of relations in a KB $\mathcal{K}$. Let d be the maximal length of a rule. The size of the search space is reduced from $O(|\mathcal{K}|^d)$ to $O(|r(\mathcal{K})|^d)$ when we remove the `addInstantiatedAtom` operator.

These are frequent patterns of the form $\vec{P} \equiv I_1(x) \wedge \cdots \wedge I_n(x)$, where $I(t)$ is in our transaction database if the item I has been bought in the transaction t. Written as the set (called an "itemset") $\vec{P} \equiv \{I_1, \ldots, I_n\}$, any subset of $\vec{P}$ forms a "more general" itemset than $\vec{P}$, which is at least as frequent as P. The Apriori algorithm uses the dual view of the support pruning strategy: Necessarily, all patterns more general than $\vec{P}$ must be frequent for $\vec{P}$ to be frequent[2]. The refinement operator of the Apriori algorithm takes as input all frequent itemsets of size n and generate all itemsets of size $n + 1$ such that any subset of size n is a frequent itemset. Thus, Apriori can be seen as a top-down rule mining algorithm over a very specific language where all atoms are unary predicates.

The WARMR algorithm [17], an ancestor of AMIE, was the first to adapt the Apriori algorithm to rule mining over multiple (multidimensional) relations.

Ontological Pathfinding. AMIE (and its successor AMIE+) [31, 30] was the first approach to explicitly target large KBs. While AMIE+ is at least 3 orders of magnitude faster than the first-generation systems, it can still take hours, even days, to find rules in very large KBs such as Wikidata. On these grounds, more recent approaches [13, 14] have proposed new strategies (parallelism, approximations, etc.) to speed up rule mining on the largest KBs. The Ontological Pathfinding method (OP) [13, 14] resorts to a highly concurrent architecture based on Spark[3] to calculate the support and the confidence of a set of candidate rules. The candidates are computed by enumerating all conjunctions of atoms that are allowed by the schema. Like AMIE, OP calculates the exact scores of the rules and supports both the CWA and the PCA for the generation of counter-evidence. At the same time, the system supports only path rules of up to 3 atoms. Other types of rules require the user to implement a new mining procedure.

RudiK. RudiK [63] is a recent rule mining method that applies the PCA to generate explicit counter-examples that are semantically related. For example, when generating counter-facts for the relation *hasChild* and a given person x, RudiK will sample among the non-children of x who are children of someone else ($x' \neq x$). RudiK's strategy is to find all rules that are necessary to predict the positive examples, based on a greedy heuristic that at each step adds the most promising rule (in terms of coverage of the examples) to the output set. Thus, differently from exhaustive rule mining approaches [13, 31,

[2]instead of: if a rule is not frequent, none of its refinements can be frequent

[3]`https://spark.apache.org`

30, 34], RudiK aims to find rules that make good predictions, not all rules above a given confidence threshold. This non-exhaustivity endows RudiK with comparable performance to AMIE+ and OP. Note that RudiK is also capable of mining negative rules, rules that predicts the absence of a fact, a novel prospect that have not yet been studied by the other approaches.

Bottom-Up Rule Mining

As the opposite of a refinement operator, one can define a generalization operator that considers several specific rules, and outputs a rule that is more general than the input rules. For this purpose, we will make use of the observation from Section 2.3.1 that a rule $b_1 \wedge ... \wedge b_n \Rightarrow h$ is equivalent to the disjunction $\neg b_1 \vee \cdots \vee \neg b_n \vee h$. The disjunction, in turn, can be written as a set $\{\neg b_1, \ldots, \neg b_n, h\}$ – which we call a *clause.* For example, the rule $\mathit{marriedTo}(x, y) \wedge \mathit{hasChild}(y, z) \Rightarrow \mathit{hasChild}(x, z)$ can be written as the clause $\{\neg \mathit{marriedTo}(x, y), \neg \mathit{hasChild}(y, z), \mathit{hasChild}(x, z)\}$. Bottom-up rule mining approaches work on clauses. Thus, they work on universally quantified disjunctions – which are more general than Horn rules. Two clauses can be combined to a more general clause using the "least general generalization" operator [65]:

Definition 2.3.14 (Least general generalization)**.** The least general generalization (lgg) of two clauses is computed in the following recursive manner:

- The lgg of two terms (i.e., either entities or variables) t and t' is t if $t = t'$ and a new variable $x_{t/t'}$ otherwise.
- The lgg of two negated atoms is the negation of their lgg.
- The lgg of $r(t_1, \ldots, t_n)$ and $r(t'_1, \ldots, t'_n)$ is $r(lgg(t_1, t'_1), \ldots, lgg(t_n, t'_n))$.
- The lgg of a negated atom with a positive atom is undefined.
- Likewise, the lgg of two atoms with different relations is undefined.
- The lgg of two clauses R and R' is the set of defined pair-wise generalizations:

$$lgg(R, R') = \{lgg(l_i, l'_j) \ : \ l_i \in R, \ l'_j \in R', \text{ and } lgg(l_i, l'_j) \text{ is defined}\}$$

For example, let us consider the following two rules:

$$\begin{aligned} hasChild(Michelle, Sasha) \ &\wedge marriedTo(Michelle, Barack) \\ &\qquad \Rightarrow hasChild(Barack, Sasha) \qquad (R) \\ hasChild(Michelle, Malia) \ &\wedge marriedTo(Michelle, x) \\ &\qquad \Rightarrow hasChild(x, Malia) \qquad (R') \end{aligned}$$

In the form of clauses, these are

$$\begin{array}{lll}\{\neg hasChild(Michelle, Sasha), & \neg marriedTo(Michelle, Barack), & \\ & \quad hasChild(Barack, Sasha)\} & (R) \\ \{\neg hasChild(Michelle, Malia), & \neg marriedTo(Michelle, x), & \\ & \quad hasChild(x, Malia)\} & (R')\end{array}$$

Now, we have to compute the lgg of every atom of the first clause with every atom of the second clause. As it turns out, there are only 3 pairs where the lgg is defined:

$$\begin{aligned} & lgg(\neg hasChild(Michelle, Sasha), \neg hasChild(Michelle, Malia)) \\ & \quad = \neg lgg(hasChild(Michelle, Sasha), hasChild(Michelle, Malia)) \\ & \quad = \neg hasChild(lgg(Michelle, Michelle), lgg(Sasha, Malia)) \\ & \quad = \neg hasChild(Michelle, x_{Sasha/Malia}) \end{aligned}$$

$$\begin{aligned} & lgg(\neg marriedTo(Michelle, Barack), \neg marriedTo(Michelle, x)) \\ & \quad = \neg marriedTo(Michelle, x_{Barack/x}) \end{aligned}$$

$$\begin{aligned} & lgg(hasChild(Barack, Sasha), hasChild(x, Malia)) \\ & \quad = hasChild(x_{Barack/x}, x_{Sasha/Malia}) \end{aligned}$$

This yields the clause

$$\begin{array}{ll}\{\neg hasChild(Michelle, x_{Sasha/Malia}), & \neg marriedTo(Michelle, x_{Barack/x}), \\ & \quad hasChild(x_{Barack/x}, x_{Sasha/Malia})\}\end{array}$$

This clause is equivalent to the rule

$$hasChild(Michelle, y) \wedge \ marriedTo(Michelle, x) \Rightarrow hasChild(x, y)$$

Note that the generalization of two different terms in an atom should result in the same variable as the generalization of these terms in another atom. In our example, we obtain only two new variables $x_{Sasha/Malia}$ and $x_{Barack/x}$. In this way, we have generalized the two initial rules to a more general rule. This can be done systematically with an algorithm called *GOLEM*.

GOLEM. The GOLEM/RLGG algorithm [60] creates, for each positive example $e \in E^+$, the rule $\mathcal{B} \Rightarrow e$, where $\mathcal{B}$ is the background knowledge. In our case, $\mathcal{B}$ is the entire KB, and so a very long conjunction of facts. The algorithm will then generalize these rules to shorter rules. More precisely, the *relative lgg* (rlgg) of a tuple of ground atoms $(e_1, \ldots, e_n)$ is the rule obtained by computing the lgg of the rules $\mathcal{B} \Rightarrow e_1$, ..., $\mathcal{B} \Rightarrow e_n$. We will call a rlgg *valid* if it is defined and does not predict any negative example.

The algorithm starts with a randomly sampled pair of positive examples (e_1, e_2) and selects the pair for which the rlgg is valid and predicts ("covers") the most positive examples. It will then greedily add positive examples, chosen among a sample of "not yet covered positive examples", to the tuple – as long as the corresponding rlgg is valid and covers more positive examples. The resulting rule will still contain ground atoms from B. These are removed, and the rule is output. Then the process starts over to find other rules for uncovered positive examples.

Progol and others. More recent ILP algorithms such as Progol [57], HAIL [68], Imparo [41] and others [92, 40] use inverse entailment to compute the hypothesis more efficiently. This idea is based on the observation that a hypothesis $\mathfrak{h}$ that satisfies $\mathcal{B} \wedge \mathfrak{h} \models E^+$ should equivalently satisfy $\mathcal{B} \wedge \neg E^+ \models \neg\mathfrak{h}$ (by logical contraposition). The algorithms work in two steps: they will first construct an intermediate theory F such that $\mathcal{B} \wedge \neg E^+ \models F$ and then generalize its negation $\neg F$ to the hypothesis $\mathfrak{h}$ using inverse entailment.

2.3.4 Related Approaches

This chapter cannot give a full review of the field of rule mining. However, it is interesting to point out some other approaches in other domains that deal with similar problems:

OWL. OWL is a Description logic language designed to define rules and constraints on the KB. For example, an OWL rule can say that every person must have a single birth date. Such constraints are usually defined upfront by domain experts and KB architects when they design the KB. They are then used for automatic reasoning and consistency checks. Thus, constraints *prescribe* the shape of the data, while the rules we mine *describe* the shape of the data. In other words, constraints are used deductively – instead of being found inductively. As such, they should suffer no exception. However, rule mining can provide candidate constraints to experts when they want to augment their theory [38].

Probabilistic ILP. As an extension of the classic ILP problem, Probabilistic ILP [16] aims to find the logical hypothesis $\mathfrak{h}$ that, given probabilistic background knowledge, maximizes the probability to observe a positive example, and minimizes the probability to observe a negative example. In our case, it would require a probabilistic model of the real world. Such models have been proposed for some specific use cases [44, 95], but they remain an ongoing subject of research.

Graph Mining and Subgraph Discovery. Subgraph discovery is a well studied problem in the graph database community (see [32] Part 8 for a quick overview). Given a set of graphs, the task is to mine a subgraph that appears in most of them. Rule mining, in contrast, is looking for patterns that are frequent in the same graph. This difference may look marginal, but the state-of-the-art algorithms are very different and further work would be needed to determine how to translate one problem to the other.

Link Prediction. Rules can be used for link prediction, i.e., to predict whether a relation links two entities. This task can also be seen as a classification problem ([32] Part 7): given two entities, predict whether there is a relation between them. A notable work that unites both views [45] uses every conjunction of atoms (a possible body for a rule, which they call a "path") as a feature dimension for this classification problem.

Representation learning, another link prediction approach based on machine learning, aims to represent entities and relations between those entities in some vector space $\mathbb{R}^d$. For example, given a fact $r(s, o)$ in our KB, the TransE model [10] will attempt to map r, s and o to vectors $\boldsymbol{r}$, $\boldsymbol{s}$ and $\boldsymbol{o}$ in $\mathbb{R}^d$ such that $\boldsymbol{s} + \boldsymbol{r} \approx \boldsymbol{o}$. For example, in the Example KB Figure 2.1, we would have:

$$\boldsymbol{Elvis} + \boldsymbol{marriedTo} \approx \boldsymbol{Priscilla}$$
$$\boldsymbol{Elvis} + \boldsymbol{hasChild} \approx \boldsymbol{Lisa}$$
$$\boldsymbol{Barack} + \boldsymbol{marriedTo} \approx \boldsymbol{Michelle}$$

Here, the task of fact prediction can be seen as finding the entity o such that:

$$\boldsymbol{Barack} + \boldsymbol{hasChild} \approx \boldsymbol{o}$$

The "embeddings" $\boldsymbol{s}$, $\boldsymbol{r}$ and $\boldsymbol{o}$ are learned by maximizing the scoring function:

$$f_r(s, o) = ||\boldsymbol{s} + \boldsymbol{r} - \boldsymbol{o}||_{1 \ or \ 2}$$

for any positive example $r(s, o)$ of the KB, and minimizing the same scoring function for any negative example (cf. the discussion Section 2.3.2 on how to choose those negative examples).

The main problem of the TransE model is that, on 1-to-N relationships, several entities will be mapped to the same vectors:

$$\boldsymbol{Michelle} + \boldsymbol{hasChild} \approx \boldsymbol{Malia} \approx \boldsymbol{Sasha}$$

Several other models such as DistMult [93], ComplEX [82] or ConvE [18], have been proposed to address these issues. A comprehensive survey on the domain of KB embeddings techniques can be found in [85].

Recent studies have shown that KB embeddings approaches do not perform significantly better than the rule mining approaches on the task of fact prediction [52, 5].

2.4 Conclusion

In this chapter, we have investigated how entities, relations, and facts in a knowledge base can be represented. We have seen the standard knowledge representation model of instances and classes. We have also seen how to represent more complex relationships between the entities in the form of rules and presented classical and contemporary rule mining approaches.

However, many challenges persist. First, as the KBs grow larger and larger, the scalability of the rule mining approaches remains a permanent problem. Second, new approaches use custom quality metrics or new pruning strategies to discover more productive rules for specialized purposes (fact prediction notably). In terms of usability, bridging the gap between general ILP-based and purpose-oriented rule mining approaches would be profitable for both approaches.

Chapter 3

AMIE 3: Fast Computation of Quality Measures

3.1 Introduction

In order to be able to process larger and larger Knowledge Bases, the previous version of AMIE, AMIE+ [30], resorts to use a sophisticated pruning strategy called the functionality heuristic. As a result, AMIE+ is way more performant than the original AMIE algorithm but this pruning heuristic is not exact and may prune perfectly legitimate rules. For example, on DBPedia 3.8, AMIE+ will wrongfully prune around 1.6% of the rules. Other recent approaches, such as Ontological Pathfinding [13], use similar pruning strategies, producing incomplete results.

Scaling is important but it should not be at the expense of the generality of the result. Without any guarantee on the results, a rule mining algorithm is just a tool that mines some rules, among many others. An exact and exhaustive rule mining algorithm would be the rule mining algorithm par excellence, as it provides a gold standard the other rule mining algorithms can be compared against.

A first difficulty in mining exhaustively the rules from a KB lies in the exponential size of the search space: every relation can potentially be combined with every other relation in a rule. The second difficulty comes from the exact computation of the quality measures, as the algorithm must consider more predictions the larger the KB is. For example, to compute exactly the confidence of the rule:

$$livesIn(x, Paris) \wedge livesIn(y, Paris) \Rightarrow marriedTo(x, y)$$

the algorithm must consider every pair of person living in Paris, which amounts to roughly $10^6 \times 10^6$ pairs for a complete KB.

In this chapter, we present AMIE 3, a successor of AMIE [31] and AMIE+ [30]. Our system employs a number of sophisticated strategies to speed up rule mining: pruning strategies, parallelization, and a lazy computation of confidence scores. This allows our system to scale effortlessly to large KBs. At the same time, the system still computes the exact confidence and support values for each rule, without resorting to approximations. Furthermore, unlike its predecessor [30] and other systems, AMIE 3 exhaustively computes all rules that hold in the KB for a given confidence and support threshold.

Our experiments show that AMIE 3 beats the state of the art by a factor of 15 in terms of runtime. We believe that the techniques that we have discovered can be of use for other systems as well — no matter whether they compute the exhaustive set of rules or not.

3.2 AMIE 3

In this section, we first recap the original AMIE algorithm [31] (Section 3.2.1). Then we present a series of optimizations that give rise to AMIE 3 (Section 3.2.2). Finally, we show different quality metrics that AMIE 3 can compute (Section 3.2.3).

3.2.1 The AMIE Approach

The AMIE algorithm [31, 30] is a method to mine closed Horn rules on large KBs. AMIE (Algorithm 1) takes as input a knowledge base $\mathcal{K}$, and thresholds l for the maximal number of atoms per rule, *minHC* for the minimum head coverage, and *minC* for the minimum PCA confidence. AMIE uses a classical breadth-first search: Line 1 initializes a queue with all possible rules of size 1, i.e., rules with an empty body. The search strategy then dequeues a rule R at a time and adds it to the output list (Line 6) if it meets certain criteria (Line 5), namely, (i) the rule is closed, (ii) its PCA confidence is higher than *minC*, and (iii) its PCA confidence is higher than the confidence of all previously mined rules with the same head atom as R and a subset of its body atoms. If the rule R has less than l atoms and its confidence can still be improved (Line 7), AMIE refines it. The refinement operator *refine* (Line 8) derives new rules from R by considering all possible atoms that can be added to the body of the rule, and creating one new rule for each of them, as described in section 2.3.3.

AMIE iterates over all the non-duplicate refinements of rule R and adds those with enough head coverage (Lines 10-11). The routine finishes when the queue runs out of rules. The AMIE algorithm has been implemented in Java with multi-threading. By default, AMIE sets *minHC*=0.01, *minC*=0.1, and $l = 3$. AMIE+ [30] optimized this algorithm by a number of pruning strategies, but did not change the main procedure.

Algorithm 1: AMIE

```
Input: a KB: $\mathcal{K}$, maximum rule length: $l$, head coverage threshold:
       $minHC$, confidence threshold: $minC$
Output: set of Horn rules: $rules$
1  $q = [\top \Rightarrow r_1(x, y), \top \Rightarrow r_2(x, y) \dots \top \Rightarrow r_m(x, y)]$
2  $rules = \langle\rangle$
3  while $|q| > 0$ do
4      $R = q.dequeue()$
5      if $closed(R) \wedge pca\text{-}conf(R) \geq minC \wedge betterThanParents(R, rules)$
        then
6          $rules.add(r)$
7      if $length(R) < l \wedge pca\text{-}conf(R_c) < 1.0$ then
8          for each rule $R_c \in refine(R)$ do
9              if $hc(R_c) \geq minHC \wedge R_c \notin q$ then
10                 $q.enqueue(r_c)$
11 return $rules$
```

3.2.2 AMIE 3

We now present the optimizations of Algorithm 1 that constitute AMIE 3, the successor of AMIE+.

Existential Variable Detection. In order to decide whether to output a rule, AMIE has to compute its confidence (Lines 5 and 7 of Algorithm 1), i.e., it has to evaluate Equation A.1. If the PCA confidence is used, this equation becomes (cf. Appendix A):

$$pca\text{-}conf(\vec{B} \Rightarrow r(x, y)) = \frac{support(\vec{B} \Rightarrow r(x, y))}{|\{(x, y) : \exists y' : \vec{B} \wedge r(x, y')\}|}. \tag{3.1}$$

This is for the case where $fun(r) \geq fun(r^-)$. If $fun(r) < fun(r^-)$, the denominator becomes $|\{(x, y) : \exists x' : \vec{B} \wedge r(x', y)\}|$. To evaluate this denominator, AMIE first finds every possible value of x. This is the purpose of Algorithm 2: We find the most restrictive atom in the query, i.e., the atom A^* with the relation with the least number of facts. If x appears in this atom, we select the possible instantiation of x in the atom for which the rest of the query is satisfiable (Lines 3 and 4). Otherwise, we recursively find the values of x for each instantiation of this most restrictive atom and add them to the result set $\mathcal{X}$. Once AMIE has found the set of possible values for x with Algorithm 2, it determines, for each value of x, the possible values of y — again by Algorithm 2. This is necessary because we cannot keep in memory all values of y encountered when we computed the values of x, because this would lead to a quadratic memory consumption.

This method can be improved as follows: Assume that our rule is simply $r_1(x,z) \wedge r_2(z,y) \Rightarrow r_h(x,y)$. Then AMIE will compute the number of distinct pairs (x,y) for the following query (the denominator of Equation 3.1):

$$r_1(x,z) \wedge r_2(z,y) \wedge r_h(x,y')$$

AMIE will use Algorithm 2 to select the possible values of x. Assume that the most restrictive atom is $r_2(z,y)$. Then AMIE will use all possible instantiations $\sigma : \{z \leftarrow Z, y \leftarrow Y\}$ of this atom, and find the possible values of x for the following query (Lines 5 and 6 of Algorithm 2):

$$r_1(x,Z) \wedge r_2(Z,Y) \wedge r_h(x,y') \tag{3.2}$$

However, we do not have to try out all possible values of y, because for a fixed instantiation $z \leftarrow Z$ all assignments $y \leftarrow Y$ lead to the same value for x. Rather, y can be treated as an existential variable: once there is a single Y with $r_2(Z,Y)$, we do not need to try out the others. Thus, we can improve Algorithm 2 as follows: If a variable y of $A^* = r(x,y)$ does not appear elsewhere in q, then Line 5 iterates only over the possible values of x in A^*.

Lazy Evaluation. The calculation of the denominator of Equation 3.1 can be computationally expensive, most notably for "bad" rules such as:

$$R : \mathit{directed}(x,z) \wedge \mathit{hasActor}(z,y) \Rightarrow \mathit{marriedTo}(x,y). \tag{3.3}$$

In such cases, AMIE spends a lot of time computing the exact confidence, only to find that the rule will be pruned away by the confidence threshold. This can be improved as follows: Instead of computing first the set of values for x, and then for each value of x the possible values of y, we compute for each value of x directly the possible values of y —and only then consider the

Algorithm 2: DistinctValues

```
Input:  variable $x$, query $q = A_1 \wedge ... \wedge A_n$, KB $\mathcal{K}$,
Output: set of values $\mathcal{X}$
1 $\mathcal{X} := \emptyset$
2 $A^* := argmin_A(|\{(x, y) : A = r(x, y), A \in q\}|)$
3 if $x$ appears in $A^*$ then
4     return $\{x : x \in \sigma(A^*) \wedge \sigma(q \setminus A^*)$ is satisfiable$\}$
5 for each $\sigma : \sigma(A^*) \in \mathcal{K}$ do
6     $\mathcal{X} := \mathcal{X} \cup DistinctValues(x, \sigma(q \setminus A^*), \mathcal{K})$
7 return $\mathcal{X}$
```

next value of x. Following the principle "If you know something is bad, do not spend time to figure out how bad exactly it is", we stop this computation as soon as the set size reaches the value $support(R) \times minC^{-1}$. If this occurs, we know that $pca\text{-}conf(R) < minC$, and hence the rule will be pruned in Line 5 of Algorithm 1.

Variable Order. To compute the PCA confidence (Equation 3.1), we have to count the instantiations of pairs of variables x, y. AMIE counts these asymmetrically: It finds the values of x and then, for each value of x, the values of y. We could as well choose to start with y instead. The number of pairs is the same, but we found that the choice impacts the runtime: Once one variable is fixed, the computation of the other variable happens on a rule that has fewer degrees of freedom than the original rule, i.e., it has fewer instantiations. Thus, one has an interest in fixing first the variable that appears in as many selective atoms as possible. Alas, it is very intricate to determine which variable restricts more efficiently the set of instantations, because the variables appear in several atoms, and each instantiation of the first variable may entail a different number of instantiations of the second variable. Therefore, estimating the exact complexity is unpractical.

We use the following heuristic: Between x and y, we choose to start with the variable that appears in the head atom of the rule in the denominator of Equation 3.1. The reason is that this variable appears in at least two atoms already, whereas the other variable appears only in at least one atom. We show in our experiments that this method improves the runtime by several orders of magnitude for some rules.

Parallel Computation for Overlap Tables. AMIE implements an approximation of Equation 3.1. This approximation misses only a small percentage of rules (maximally 5% according to [30]), but speeds up the calculation drastically. In AMIE 3, this feature can be switched off (to have exact results) or on (to have faster results). Here, we show how to further speed up this heuristic. The method finds an efficient approximation of the denominator of Equation 3.1 for a rule R. This approximation uses the join structure of the query in combination with the functionality scores and the overlaps of the different relations to estimate the total number of examples (both positive and negative) of a rule. The exact formula and the rationale behind it can be found in [30]. The functionality, domain and overlaps with other relations are pre-computed for all relations. This pre-calculation can be significant for large KBs with many predicates. In our experiments with DBpedia, e.g., precomputing all overlaps takes twice as much time as the mining. In AMIE 3, we exploit the fact that this task is easy parallelizable, and start as many threads as possible in parallel, each treating one pair of relations. This reduces the precomputation time linearly with the number of threads (by a factor of 40 in our experiments).

Integer-based in-memory database. AMIE uses an in-memory database to store the entire KB. Each fact is indexed by subject, by object, by relation, and by pairs of relation/subject and relation/object. In order to be able to load also large KBs into memory, AMIE compresses strings into custom-made *ByteStrings*, where each character takes only 8 bits. AMIE makes sure that ByteString variables holding equivalent ByteStrings point to the same physical object (i.e., the ByteString exists only once). This not just saves space, but also makes hashing and equality tests trivial. Still, we incur high costs of managing these objects and the indexes: ByteStrings have to be first created, and then checked for duplicity; unused ByteStrings have to be garbage-collected; equality checks still require casting checks; and HashMaps create a large memory overhead. Built-in strings suffer from the same problems. Therefore, we migrated the in-memory database to an integer-based system, where entities and relations are mapped to an integer space and represented by the primitive datatype *int*. This is in compliance with most RDF engines and popular serialization formats such as [25]. We use the fastutil library[1] to store the indexes. This avoids the overhead of standard HashMaps. It also reduces the number of objects that the garbage collector has to treat, leading to a significant speedup.

[1]`http://fastutil.di.unimi.it/`

3.2.3 Quality Metrics

AMIE is a generic exhaustive rule miner, and thus its output consists of *rules*. These rules can serve as input to other applications, for example, to approaches that predict *facts* [20, 63]. By default, AMIE uses the PCA confidence to assess the quality of a rule, because it has been shown to rank rules closer to the quality of their predictions than classical metrics such as the CWA confidence [31]. However, the PCA cannot generate counter-examples for functional relations, which is problematic for rules such as:

$$hasChild(z, x) \land politicianOf(z, y) \Rightarrow politicianOf(x, y)$$

This rule has a high PCA confidence as the children of a politician that are not politician are not counted as a counter-examples of the rule[2]. Thus the rule actually represents the fact that every child of a politician *who is politician of some country* is politician of the same country as their parents but the same rule would be used to predict that every child of a politician is a politician. We specifically give directions to address this problem in Section 8.2.

AMIE 3 is not limited to the PCA and can compute any of the following quality metrics that can be enabled by command lines switches:

Support & head coverage. Support is a standard quality metric that indicates the significance of a rule. Due to the anti-monotonicity property, most approaches use support to prune the search space of rules. AMIE [31, 30] uses by default the head coverage (the relative variant of support) for pruning.

PCA Confidence. The default confidence metric.

CWA confidence. This confidence is used in OP [13, 14]. Many link prediction methods are evaluated under the closed world assumption as well [80].

GPRO confidence. The work of [20] noted that the PCA confidence can underestimate the likelihood of a prediction in the presence of non-injective mappings. Therefore, the authors propose a refinement of the PCA confidence, the GPRO confidence, which excludes instances coming from non-injective mappings in the confidence computation. To judge the quality of a predicted fact, the approach needs the GPRO confidence both on the first and second variable of the head atom. AMIE is not designed to judge the quality of a predicted fact, but can compute the GPRO confidence on both variables.

[2]as they are not subject of a $politicianOf$ relation.

GRANK confidence. This refinement of the GPRO metric is proposed by [20] in order to take into account the number of instances of the variables of the rule that are not in the head atom.

The rules mined by AMIE can be used to support task such as data cleaning or reasoning but can also be added to an open rule repository as proposed in [3] to facilitate sharing and comparison between different approaches and quality metrics.

3.3 Experiments

We conducted two series of experiments to evaluate AMIE 3: In the first series we study the impact of our optimizations on the system's runtime. In the second series, we compare AMIE 3 with two scalable state-of-the-art approaches, namely RudiK [63] and Ontological Pathfinding (OP) [13, 14] (also known as ScaleKB) on 6 different datasets.

3.3.1 Experimental Setup

Data. We evaluated AMIE 3 and its competitors on YAGO (2 and 2s), DBpedia (2.0 and 3.8) and a dump of Wikipedia from December 2014. These datasets were used in evaluations of AMIE+ [30], OP [13] and Rudik [63]. In addition, we used a recent dump of Wikidata from July 1st, 2019[3]. Table 3.1 shows the numbers of facts, relations, and entities of our experimental datasets.

Configurations. All experiments were run on a Ubuntu 18.04.3 LTS with 40 processing cores (Intel Xeon CPU E5-2660 v3 at 2.60GHz) and 500Go of RAM. AMIE 3 and RudiK are implemented in Java 1.8. AMIE 3 uses its own in-memory database to store the KB, whereas RudiK relies on Virtuoso Open Source 06.01.3127, accessed via a local endpoint. OP was implemented in Scala 2.11.12 and Spark 2.3.4.

Unless otherwise noted, the experiments were run using the default settings of AMIE: We used the PCA confidence, computed lazily with a threshold of 0.1, with all the lossless optimizations (no approximations). The threshold on the head coverage is 0.01 and the maximal rule length is 3 [31].

[3]Selecting only facts between two Wikidata entities, and excluding literals.

Table 3.1 – Experimental datasets

Dataset	Facts	Relations	Entities
Yago2	948 358	36	834 750
Yago2s	4 484 914	37	2 137 469
DBpedia 2.0	6 601 014	1 595	2 275 327
DBpedia 3.8	11 024 066	650	3 102 999
Wikidata 12-2014	8 397 936	430	3 085 248
Wikidata 07-2019	386 156 557	1 188	57 963 264

Table 3.2 – Loading time and memory used. *Ov. tables* is the time needed to compute the overlap tables.

Dataset	Loading	Ov. tables	Memory used	
			Integer	ByteString
Yago2	7s	0.2s	6Go	9Go
Yago2s	45s	2.4s	16Go	19Go
DBpedia 2.0	55s	23.5s	29Go	32Go
DBpedia 3.8	1min 20s	15.2s	40Go	42Go
Wikidata 2014	59s	12s	27Go	54Go
Wikidata 2019	42min 53s	41.4s	479Go	N/A

3.3.2 Effect of our optimizations

In-memory database. Table 3.3 shows the performance with the new integer-based in-memory database and the old ByteString database. The change reduces the memory footprint by around 3 GB in most cases, and by 50% in Wikidata. Moreover, the new database is consistently faster, up to 8-fold for the larger KBs such as DBpedia 3.8.

Laziness. As explained in Section 3.2.2, AMIE can invest a lot of time in calculating the PCA confidence of low-confident rules. The lazy evaluation targets exactly this problem. Table 3.4 shows that this strategy can reduce the runtime by a factor of 4. We also show the impact of laziness when the PCA confidence approximation is switched on. We observe that the parallel calculation of the overlap tables reduces drastically the contribution of this phase to the total runtime when compared to AMIE+ —where it could take

Table 3.3 – Old ByteString database vs. the new integer-based database.

Dataset	Integer	ByteString
Yago2	26.40s	29.69s
Yago2s	1min 55s	4min 10s
DBpedia 2.0	7min 32s	34min 06s
DBpedia 3.8	7min 49s	52min 10s
Wikidata 2014	5min 44s	6min 01s

Table 3.4 – Impact of laziness and of switching on the confidence approximation.

Dataset	Conf. Approx. off		Conf. Approx. on	
	Non-lazy	Lazy	Non-lazy	Lazy
Yago2	24.12s	26.40s	24.39s	21.41s
Yago2s	4min 28s	1min 55s	1min 42s	2min 03s
DBpedia 2.0	10min 14s	7min 32s	7min 42s	8min 13s
DBpedia 3.8	14min 50s	7min 49s	11min 07s	10min 18s
Wikidata 2014	19min 27s	5min 44s	5min 45s	4min 36s
Wikidata 2019	$>$ 48h	16h 43min	17h 06min	16h 31min

longer than the mining itself. We also note that the residual impact of the confidence approximation is small, so that this feature is now dispensable: We can mine rules exhaustively.

Count variable order. To measure the impact of the count variable order, we ran AMIE 3 (with the lazy evaluation activated) on Yago2s and looked at the runtimes when counting with the variable that appears in the head atom versus the runtime when counting with the other variable. For every rule with three atoms and a support superior to 100, we timed the computation of the PCA confidence denominator (Equation 3.1) in each case. The y-axis of Figure 3.1 shows the runtime when we first instantiate the variable that occurs in the head atom, whereas the x-axis shows the runtime when using the other variable.

We see that every query can be run in under 10 seconds and that most of the queries would run equally fast independently of the order of the variables. However, for some rules, instantiating first the variable that does not

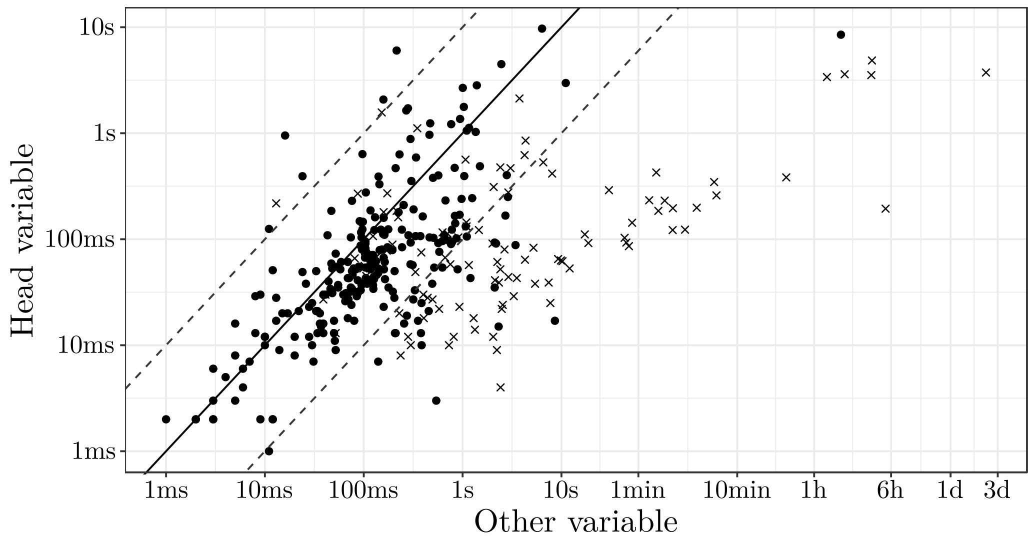

Figure 3.1 – Impact of the variable order on Yago2s. Each point is a rule. Cross points: pruned by the confidence approximation. Plain line: same performance. Dashed lines: relative speedup of 10×.

appear in the head atom can be worse than the contrary by several orders of magnitude. Some queries would take hours (days in one case) to compute, even with lazy evaluation. In Yago2s, these rules happen to be pruned away by the AMIE+ confidence upper bound (a lossless optimization), but this may not be the case for all KBs. The problematic rules all have bodies of the following shape:

$$\begin{cases} \mathit{hasGender}(x, g) \wedge \mathit{hasGender}(y, g) \\ \mathit{isLocatedIn}(x, l) \wedge \mathit{isLocatedIn}(y, l) \end{cases}$$

Both *hasGender* and *isLocatedIn* are very large relations as they apply to any person and location, respectively. While early pruning of those "hard rules" is the purpose of the confidence approximations and upper bounds of AMIE+, these strategies may fail in a few cases, leading to the execution of expensive queries. Finally, we show the overall impact of the count variable order heuristic in Table 3.5. The results suggest that our heuristic generally yields lower runtimes.

Impact of existential variable detection. Last but not least, the on-the-fly detection of existential variables reduces the number of recursive calls made to Algorithm 2. Table 3.6 shows the performances of AMIE 3 with and without this optimization. This optimization is critical for AMIE 3 on most

Table 3.5 – Impact of the variable order: variable that appears in the head atom (new AMIE 3 heuristic); variable that does not appear in the head atom; variable that appears first in the head atom of the original rule (old AMIE method).

Dataset	Head	Non-head	Always first
Yago2	26.40s	25.64s	23.59s
Yago2s	1min 55s	4min 32s	4min 30s
DBpedia 2.0	7min 32s	12min 46s	6min 36s
DBpedia 3.8	7min 49s	21min 12s	8min 53s
Wikidata 2014	5min 44s	36min 09s	9min 50s

Table 3.6 – Existential variable detection (ED)

Dataset	AMIE 3	No ED
Yago2	26.40s	24.84s
Yago2s	1min 55s	> 2h
DBpedia 2.0	7min 32s	9min 10s
DBpedia 3.8	7min 49s	> 2h
Wikidata 2014	5min 44s	> 2h

datasets. This is less important for DBpedia 2.0 as it contains mostly small relations.

Metrics. Table 3.7 shows the impact of different quality metrics on the runtime, with iPCA being the PCA with injective mappings. The metrics run slower than the PCA confidence, because we cannot use the PCA upper bound optimization. The GRank metric, in particular, is very sensitive to the number of facts per relation, which explains its performance on Yago2s and DBpedia 3.8. For all other metrics, however, the numbers are very reasonable.

3.3.3 Comparative Experiments

In this section, we compare the performance of AMIE 3 with two main state-of-the-art algorithms for rule mining in large KBs, RuDiK and OP.

Table 3.7 – Different metrics (Section 3.2.3)

CWA	iPCA	GPro	GRank
22.54s	38.42s	37.47s	33.36s
1min 56s	3min 30s	2min 45s	> 2h
7min 26s	12min 31s	11min 53s	1h 16min
6min 49s	15min 22s	23min 31s	> 2h
5min 48s	7min 04s	11min 50s	> 2h

AMIE 3. We ran AMIE 3 in its default settings. In order to compare the improvements to previous benchmarks of AMIE, we had AMIE compute the standard CWA confidence for each rule, in addition to the PCA confidence (except for Wikidata 2019, where no such previous benchmark exists).

RuDiK. We set the number of positive and negative examples to 500, as advised on the project's github page[4]. We tried to run the system in parallel for different head relations. However, the graph generation phase of the algorithm already runs in parallel and executes a lot of very selective SPARQL queries in parallel. Hence, the additional parallelization flooded the SPARQL endpoint, which rejected any new connection at some point. For this reason, we mined the rules for every possible relation sequentially, using only the original parallelization mechanism. RuDiK also benefits from information on the taxonomic types of the variables. While the built-in method to detect the types of the relations works out-of-the-box for DBpedia (which has a flat taxonomy), it overgeneralizes on the other datasets, inverting the expected benefits. Therefore, we ran RuDiK without the type information on the other datasets.

Ontological Pathfinding. This system first builds a list of candidate rules (Part 5.1 of [14]). Unfortunately, the implementation of this phase of the algorithm is not publicly available. Hence, we had to generate candidate rules ourselves. The goal is to create all rules that are "reasonable", i.e., to avoid rules with empty joins such as $\mathit{birthPlace}(x, y) \wedge \mathit{hasCapital}(x, z)$. The original algorithm discards all rules where the domain and range of joining relations do not match. However, it does not take into account the fact that an entity can be an instance of multiple classes. Thus, if the domain of *actedIn* is *Actor*, and the domain of *directed* is *Director*, the original algorithm

[4] `https://github.com/stefano-ortona/rudik`

Table 3.8 – Performances and output of Ontological Pathfinding (OP), RuDiK and AMIE 3. *: rules with support ≥ 100 and CWA confidence ≥ 0.1.

Dataset	System	Rules	Runtime
Yago2s	OP (their candidates)	429 (52*)	18min 50s
	OP (our candidates)	1 348 (96*)	3h 20min
	RuDiK	17	37min 30s
	AMIE 3	97	**1min 50s**
	AMIE 3 (support=1)	1 596	7min 6s
DBpedia 3.8	OP (our candidates)	7 714 (220*)	> 45h
	RuDiK	650	12h 10min
	RuDiK + types	650	11h 52min
	AMIE 3	5 084	**7min 52s**
	AMIE 3 (support=1)	132 958	32min 57s
Wikidata 2019	OP (our candidates)	15 999 (326*)	> 48h
	RuDiK	1 145	23h
	AMIE 3	8 662	**16h 43min**

would discard any rule that contains $actedIn(x, y) \wedge directed(x, z)$ —even though it may have a non-empty support. Hence, we generated all candidate rules where the join between two connected atoms is not empty in the KB. This produces more candidate rules than the original algorithm (around 10 times more for Yago2s, i.e., 29762), but in return OP can potentially mine all rules that the other systems mine.

Results

It is not easy to compare the performance of OP, AMIE 3, and Rudik, because the systems serve different purposes, have different prerequisites, and mine different rules. Therefore, we ran all systems in their default configurations, and discuss the results (Table 3.8) qualitatively in detail.

Ontological Pathfinding. We ran OP both with a domain-based candidate generation (which finds fewer rules) and with our candidate generation. In general, OP has the longest running times, but the largest number of rules. This is inherent to the approach: OP will prune candidate rules using a heuristic [13] that is similar to the confidence approximation of AMIE+. After this step, it will compute the support and the exact CWA confidence of any remaining candidate. However, it offers no way of pruning rules upfront by support and confidence. This has two effects: First, the vast majority ($> 90\%$) of rules found by OP have very low confidence ($< 10\%$) or very low support (< 100). Second, most of the time will be spent computing the confidence of these low-confidence rules, because the exact confidence is harder to compute for a rule with low confidence.

To reproduce the result of OP with AMIE, we ran AMIE 3 with a support threshold of 100 and a CWA confidence threshold of 10%. This reproduces the rules of OP (and 8 more because AMIE does not use the OP functionality heuristics) in less than two minutes. If we set our support threshold to 1, and our minimal CWA confidence to 10^{-5}, then we mine more rules than OP on Yago2s (as shown in Table 3.8) in less time (factor $25\times$). If we mine rules with AMIE's default parameters, we mine rules in less than two minutes (factor $90\times$).

The large search space is even more critical for OP on DBpedia 3.8 and Wikidata 2019, as the number of candidate rules grows cubically with the number of relations. We generated around 9 million candidate rules for DBpedia and around 114 million candidates for Wikidata. In both cases, OP mined all rules of size 2 in 1h 20min ($\approx 21k$ candidates) and 14 hours ($\approx 100k$ candidates) respectively. However, it failed to mine any rule of size 3 in the remaining time. If we set the minimal support again to 1 and the CWA confidence threshold to 10^{-5}, AMIE can mine twice as many rules as OP on DBpedia 3.8 in 33 minutes.

RuDiK. For RuDiK, we found that the original parallelization mechanism does not scale well to 40 cores. The load average of our system, Virtuoso included, never exceeded 5 cores used. This explains the similar results between our benchmark and RuDiK's original experiments on Yago2s with fewer cores. On DBpedia, we could run the system also with type information —although this did not impact the runtime significantly. The loss of performance during the execution of the SPARQL queries is more noticeable due to the multitude of small relations in DBpedia compared to Yago. In comparison, AMIE was more than $20\times$ faster on both datasets. This means that, even if RuDiK were to make full use of the 40 cores, and speed up 4-fold, it would still be 5 times slower. AMIE also found more rules than RuDiK. Among these are all rules that RuDiK found, except two (which were clearly wrong rules; one had a confidence of 0.001).

In our experiment, RuDiK mined rules in Wikidata in 23 hours. However, RuDiK was not able to mine rules for 22 of the relations as Virtuoso was not able to compute any of the positive or the negative examples RuDiK requires to operate. This is because RuDiK would timeout any SPARQL query after 20 seconds of execution[5]. Virtuoso failed to compute the examples during this time frame on the 22 relations, which are the largest ones in our Wikidata dataset: They cover 84% of the facts. Interestingly, RuDiK did also not find rules that contain these relations in the body (except one, which covered 0.5% of the KB).

In comparison, AMIE mined 1703 rules with at least one of these relations, computing the support, confidence and PCA confidence exactly on these huge relations —in less time. For example, it found the rule $\mathit{inRegion}(x, y) \wedge \mathit{inCountry}(y, z) \Rightarrow \mathit{inCountry}(x, z)$, which is not considered by RuDiK, but has a support of over 7 million and a PCA confidence of over 99%.

AMIE 3. outperformed both OP and RuDiK in terms of runtime and the number of rules. Moreover, it has the advantage of being exact and complete. Then again, the comparisons have to be seen in context: RuDiK, e.g., is designed to run on a small machine. For this, it uses a disk-based database and sampling. AMIE, in contrast, loads all data into memory, and thus has a large memory footprint (the 500GB were nearly used up for the Wikidata experiment). In return, it computes all rules exactly and is fast.

3.4 Conclusion

We have presented AMIE 3, the newest version of the rule mining system AMIE (available at `https://github.com/lajus/amie/`). The new system uses a range of optimization and pruning strategies, which allow scaling to large KBs that were previously beyond reach. In particular, AMIE 3 can exhaustively mine all rules above given thresholds on support and confidence, without resorting to sampling or approximations.

[5]Increasing the timeout parameter is not necessarily a good solution for two reasons: First, we cannot predict the optimal value so that all queries finish. Second, it would increase the runtime of queries succeeding with partial results thanks to Virtuoso's Anytime Query capability. This would largely increase RuDiK's runtime with no guarantee to solve the issue.

Chapter 4

Star patterns: Reducing the Search Space

4.1 Introduction

The improvements of Chapter 3 were focused on improving the performances of the computation of the support and confidence measures of a specific rule. AMIE 3 still relies on the original base algorithm for exploring the search space. This algorithm, however, shows clear limitations when we want to mine more complex rules on large Knowledge Bases. For example, AMIE 3 is neither able to mine every rule of size 4 nor the rules of size 3 with constants within 12 hours on Yago2s.

Instantiations are critical to handle properly some relations and recover valuable rules. For example, using instantiations on the *type* relation allows to get schema information about the rule, i.e the types of the variables used in the rule. In contrast, without the instantiations, AMIE is able to mine only that a variable "has a type" (and eventually, that two variables have the same type). As for longer rules, they can model more complex relationships and in practice the interest for such rules is appreciable among AMIE's userbase, according to the recent exchanges we had on AMIE's project page.

To mine longer rules and rules with constants efficiently, we must address how AMIE handles the search space directly. More precisely, for any rule, AMIE computes the support of any refinements of this rule and potentially prune it if it is a duplicate or if its support does not meet the required threshold. In both cases, the computation of the support is, a posteriori, superfluous. We propose in this chapter a method to discard, a priori, an unwanted rule, without computing the support of the rule at all. Such an early pruning process should speed up AMIE exploration significantly.

Contribution. In particular, we will decompose the rules into simpler patterns, the star patterns, that will be used to identify, a priori, unsatisfiable rules.

This chapter is structured as follows: In Section 4.2 we define the star patterns and introduce the necessary condition used to prune the rules early on. Then, in Section 4.3 we show how frequent pattern mining can be used to extract from the KB all the star patterns that satisfy the necessary condition. Finally, in Section 4.4 we present a method to combine the star patterns into rules that satisfy the necessary condition, thus achieving the construction of our pruned search space. However, we also show in this last section that the naive generation of all rules is not tractable in the general case.

In Chapter 5, we will show that in the specific case of path rules, the necessary condition can be used constructively to efficiently prune the search space.

4.2 Star patterns

4.2.1 Patterns

A Knowledge Base can be seen as a directed graph with labelled edges between entities. The task of rule mining can thus be seen as the task of finding frequent closed subgraphs in this huge graph. We can represent these subgraphs using patterns, a graph representation of a rule with variables in place of vertices. For example the pattern in Figure 4.1a represents the rule

$$marriedTo(y, x) \Rightarrow marriedTo(x, y)$$

In the Elvis KB (represented as a labeled graph in Figure 2.1), we can find four couples of values for (x, y) that satisfy this rule: (Elvis, Priscilla), (Priscilla, Elvis), (Barack, Michelle) and (Michelle, Barack). Finding those couples is equivalent to finding where the pattern matches on the labeled graph representation of the KB, printing the pattern over a tracing paper and trying to find matches on the KB graph.

The patterns are just representations of the rules, and there are just as expressive. Some apply on a KB and others do not. Some are simple and others are ridiculously complex. But even a more complex pattern can be broken into smaller patterns, simpler to find, to help localize where the bigger pattern applies. It is as if you cut the tracing paper into smaller pieces.

To match the pattern of Figure 4.1a[1], AMIE also breaks it into smaller

[1]to find the set of pair of entities (x, y) of the KB satisfying $marriedTo(y, x) \wedge marriedTo(x, y)$

$$\begin{tikzcd} x \arrow[r, shift right, "\mathsf{marriedTo}"'] & y \arrow[l, shift right, "\mathsf{marriedTo}"'] \end{tikzcd}$$

(a) Pattern for $marriedTo(y, x) \wedge marriedTo(x, y)$

$$\begin{tikzcd} x \arrow[r, "\mathsf{marriedTo}"'] & y \end{tikzcd} \qquad \begin{tikzcd} x & y \arrow[l, "\mathsf{marriedTo}"'] \end{tikzcd}$$

(b) Relational sub-patterns for $marriedTo(y, x) \wedge marriedTo(x, y)$

$$\begin{tikzcd} ? \arrow[dr, "\mathsf{marriedTo}"'] & & ? \\ & x \arrow[ur, "\mathsf{marriedTo}"'] & \end{tikzcd} \qquad \begin{tikzcd} ? \arrow[dr, "\mathsf{marriedTo}"'] & & ? \\ & y \arrow[ur, "\mathsf{marriedTo}"'] & \end{tikzcd}$$

(c) Star (sub-)patterns for $marriedTo(y, x) \wedge marriedTo(x, y)$

Figure 4.1 – Patterns and sub-patterns on a simple example

patterns. These patterns are centered around the relation, as AMIE will try out all the pairs (x, y) satisfying $marriedTo(y, x)$ against the condition $marriedTo(x, y)$ (or the opposite depending on the most restrictive relation). Thus it will consider the sub-patterns represented in Figure 4.1b.

In order to find a necessary condition, we will consider the other set of smaller pieces: the star patterns, i.e the patterns centered on a single variable as in Figure 4.1c.

4.2.2 Star patterns

Definition 4.2.1 (Star pattern)**.** For a rule R and any variable x of this rule, we can construct the star patterns of x in the rule R, noted $sp(x, R)$ as follows:

- First we identify in the rule the relations $[r_{i_1}, \dots r_{i_n}]$ where x appears in the subject position and the relations $[r_{j_1}, \dots r_{j_m}]$ where x appears in the object position.
- Then we pose:

$$sp(x, R) = \bigwedge_{k=1}^{n} \exists y_k . r_{i_k}(x, y_k) \wedge \bigwedge_{k=1}^{m} \exists z_k . r_{j_k}(z_k, x)$$

As such, a star pattern is a conjunctive query where only one variable is open and the other variables are existentially quantified.

For example, in the rule $R = marriedTo(y, x) \Rightarrow marriedTo(x, y)$, x is in the subject position of the relation *marriedTo* and in the object position of another *marriedTo* atom. Thus we have:

$$sp(x, R) = \exists a.marriedTo(x, a) \wedge \exists b.marriedTo(b, x)$$

The star patterns capture the pattern of relations around a single variable. Represented as in Figure 4.1c, we can directly identify that the variable x must be in the subject position and in the object position of the *marriedTo* relation.

Set notation. To simplify the notation, we denote every atom of the form $\exists y.r(x, y)$ by the item r and represent a conjunction of such atoms as a set of items. Finally, we use signed relations to represent the atoms where x appears in the object position. Using this notation, we have:

$$sp(x, R) = \{marriedTo, marriedTo^{-1}\}$$

Definition 4.2.2 (Support and Satisfiability of a star pattern)**.** By substitution, we can identify the entities that satisfy a star pattern (seen as a simple conjunctive query) in a KB $\mathcal{K}$. The number of such entities in $\mathcal{K}$ is called the support of the star pattern, and we say that a star pattern is satisfiable if its support is at least one.

Proposition 4.2.3 (Necessary condition)**.** In a KB $\mathcal{K}$ an entity E can be a valid substitution for x in the rule R only if E satisfies $sp(x, R)$ in $\mathcal{K}$.

As a consequence, a rule is satisfiable in a KB $\mathcal{K}$ only if every one of its star patterns is satisfiable in this KB.

Proof. Syntactically, $sp(x, R)$ is more general than R. $\square$

Proposition 4.2.3 gives us a necessary condition for a rule to be satisfiable. In Section 4.2.4, we will introduce the language of star patterns considered and a systematic way to decompose a rule into star patterns of this language.

The first observation of Proposition 4.2.3 will also be used in order to restrict the set of possible instantiations for any variable x and speed up the computation of the quality measures of a rule in Section 6.2.5.

4.2.3 Related work

Star patterns have already been introduced in [62] under the name of Characteristic sets in order to compute cardinality estimations of a Sparql query.

$$\begin{tikzcd}
\text{Person} & x \arrow[l, "\mathsf{type}"'] \arrow[rr, "\mathsf{marriedTo}"] \arrow[dr, "\mathsf{hasChild}"'] & & y \arrow[dl, "\mathsf{hasChild}"] \\
& & z &
\end{tikzcd}$$

(a) The example rule R

$$\begin{tikzcd}
? & x \arrow[l, "\mathsf{type}"'] \arrow[r, "\mathsf{marriedTo}"] \arrow[dr, "\mathsf{hasChild}"'] & ? \\
& & ?
\end{tikzcd}$$

(b) The star pattern $sp(x, R)$

$$\begin{tikzcd}
& ? \arrow[r, "\mathsf{marriedTo}"] & y \arrow[dl, "\mathsf{hasChild}"] \\
& ? &
\end{tikzcd}$$

(c) The star pattern $sp(y, R)$

$$\begin{tikzcd}
? \arrow[dr, "\mathsf{hasChild}"'] & & ? \arrow[dl, "\mathsf{hasChild}"] \\
& z &
\end{tikzcd}$$

(d) The star pattern $sp(z, R)$

Figure 4.2 – The example rule R and its star patterns

Further works have been proposed, focusing on query optimization [55, 53], data profiling [33] or KB sampling [37].

To the best of our knowledge, those characteristic sets have never been used in the context of rule mining, or in an attempt to compute the complete set of satisfiable queries as we are doing here.

In the following, we will keep our notion of "star patterns" for what has been called "characteristic sets" in the literature because the graphical representation we introduced is more intuitive for the purpose of rule recombination.

4.2.4 Languages of star patterns

The definition 4.2.1 of a star pattern has two main limitations. In this section we introduce these limitations via the study of the example rule:

$$R = type(x, Person) \wedge marriedTo(x, y) \wedge hasChild(x, z) \Rightarrow hasChild(y, z)$$

and we define different languages of star patterns in order to cope with those limitations.

The example rule R and its star patterns are depicted in Figure 4.2. The logical formulae corresponding to these star patterns are:

(4.2b) $sp(x, R) = \exists c.type(x, c) \wedge \exists a.marriedTo(x, a) \wedge \exists b.hasChild(x, b)$

(4.2c) $sp(y, R) = \exists c.hasChild(y, c) \wedge \exists d.marriedTo(d, y)$

(4.2d) $sp(z, R) = \exists e.hasChild(e, z) \wedge \exists f.hasChild(f, z)$

Multiplicity. The first limitation comes from the logical representation of $sp(z, R)$. The pattern depicted in Figure 4.2d represents the fact that z should have at least two parents. However, the multiplicity of the relation is lost when we use the logical formula:

$$sp(z, R) = \exists e.hasChild(e, z) \wedge \exists f.hasChild(f, z)$$

which is equivalent to $\exists e.hasChild(e, z)$. In order to take the multiplicity into account, we define:

$$sp^{\text{MULT}}(z, R) = \exists e, f.\ (e \neq f \wedge hasChild(e, z) \wedge hasChild(f, z))$$

In general, given the list of relation $[r_{i_1}, \ldots, r_{i_n}]$ where x appears in the subject position and given a relation r_i appearing m times in the list, $sp^{\text{MULT}}(x, R)$ contains the term $\exists y_{i_1} ... \exists y_{i_m} \bigwedge_{1 \leq k < j \leq m} y_{i_k} \neq y_{i_j} \bigwedge_{k=1}^{m} r_i(x, y_{i_k})$.

Instantiations. The second limitation comes from the handling of constants. In our example rule R, the type of x is clearly identified. However, this value is lost when we consider the star-pattern $sp(x, R)$. As our purpose is to recombine star patterns into rules, this effect will eventually prevent us from mining rules with constants. Thus we define:

$$sp^{\text{INST}}(x, R) = type(x, Person) \wedge \exists a.marriedTo(x, a) \wedge \exists b.hasChild(x, b)$$

In general, $sp^{\text{INST}}(x, R)$ can be computed by computing $sp(x, R)$ without considering the atoms with a constant where x appears and by adding these atoms directly to the result.

Note that we can use both languages, with multiplicity and instantiations, together.

Set notation. We extend the set notation we introduced in Section 4.2.2 to take into account multiplicity and instantiations. For this we define new types of items as shown in Table 4.1.

Item	Logical equivalent	Language considered
r	$\exists a.r(x,a)$	All
$r(E)$	$r(x,E)$ with E any entity of the KB	With instantiations
r^n	$\exists y_1 ... \exists y_n \bigwedge_{1\leq i<j\leq n} y_i \neq y_j \bigwedge_{i=1}^{n} r(x,y_i)$	With multiplicity

Table 4.1 – Item notation and its logical equivalent for a signed relation r

Example. We recall here our example rule:

$$type(x, Person) \wedge marriedTo(x,y) \wedge hasChild(x,z) \Rightarrow hasChild(y,z)$$

Using the items defined in Table 4.1, we can transform the star patterns of this rule in their equivalent set notation:

(x) $sp^{\text{INST}}(x,R) = \{type(Person), marriedTo, hasChild\}$

(y) $sp(y,R) = \{hasChild, marriedTo^{-1}\}$

(z) $sp^{\text{MULT}}(z,R) = \{hasChild^{-2}\}$

Note that we also simplified the item $(hasChild^{-1})^2$ to $hasChild^{-2}$.

This way the star patterns can be represented as itemsets, i.e set of items as defined in Table 4.1. The size of a star pattern is defined as the sum of the multiplicity of the different items (the power of the items).

4.3 Mining star patterns

Our goal here is to mine the satisfiable star patterns from an input KB, because these patterns are the necessary conditions for a rule to hold.

In the following, we fix an input KB $\mathcal{K}$ and we note $\mathcal{E}$ the set of entities of $\mathcal{K}$ and $\mathcal{I}$ the set of all syntactically possible items of $\mathcal{K}$ (using instantiations and multiplicity). The star patterns that are satisfiable in $\mathcal{K}$ can be mined by translating $\mathcal{K}$ into a transactional database over $\mathcal{I}$ and using the Apriori algorithm.

Definition 4.3.1 (Itemset of an entity)**.** The itemset of an entity E in a KB $\mathcal{K}$ is the maximal star pattern (for set inclusion) $sp(E) \in 2^{\mathcal{I}}$ that E satisfies in $\mathcal{K}$.

Definition 4.3.2 (The KB as a transactional database)**.** We define the transactional database $sp(\mathcal{K}) \in 2^{2^{\mathcal{I}}}$ as:

$$sp(\mathcal{K}) = \{sp(E) : E \in \mathcal{E}\}$$

Definition 4.3.3 ($Star_\theta$)**.** Given a transactional database $sp(\mathcal{K})$ and a positive integer parameter θ, we define the set of star patterns $Star_\theta(\mathcal{K})$ as:

$$Star_\theta(\mathcal{K}) = \{\sigma \in 2^{\mathcal{I}} : support(\sigma) \geq \theta\}$$

By definition,

$$\forall \theta, \theta' \geq 1, \quad \theta > \theta' \Rightarrow Star_\theta \subseteq Star_{\theta'}$$

And the set of satisfiable star patterns of $\mathcal{K}$ is $Star_1(\mathcal{K})$.

In an actual KB, star patterns are naturally selective: it means that if you take a star pattern at random, such as $\{birthPlace^{-1}, marriedTo\}$ or $\{type(Dog), marriedTo\}$, it is unlikely that it will be satisfiable. This is essentially due to the heterogeneity of the data and of the relations that are present in a KB. Indeed, most items are incompatible in the sense that they never actually co-occur: we don't expect to find a birth place married to someone nor a married dog.

However, the satisfiability of a star pattern is very sensible to errors or outliers[2] as a single (eventually erroneous) entity is sufficient to allow new possible star patterns. The θ parameter allows us to measure the impact of these outliers in the size of $Star_\theta(\mathcal{K})$, in the tables below.

But for the purpose of rule mining, we will have to consider the entire set of satisfiable star patterns, i.e. $Star_1(\mathcal{K})$ for our approach to be exhaustive. Indeed, the following valid rule[3] in YAGO:

$$hasGender(x, g) \wedge hasChild(x, y) \Rightarrow hasGender(y, g)$$

possesses the satisfiable star pattern $\{hasGender^{-2}\}$ (for g) but the support of this star pattern in our KB is only two[4]. This means that we cannot find this rule by combining star patterns of $Star_3(YAGO)$.

Star pattern selectivity and schema constraints. The natural selectivity of the star patterns is a consequence of the schema of the relations of a KB (cf. Definition 2.2.3), as most classes, e.g Dog or Person, do not intersect. But instead of relying on the schema information provided by the KBs, which is often unreliable or too general, the star patterns capture from the actual data the compatibility or the incompatibility of the items. In this sense, star patterns are a generalization of the schema constraints.

[2]as married dogs in fact: https://fxn.ws/2SxIkgH

[3]according to AMIE with the default parameters

[4]only *Male* and *Female* are genders in YAGO.

Table 4.2 – Number of items in Yago2

Type of items	$Star_1(\mathcal{K})$	$Star_{10}(\mathcal{K})$	$Star_{50}(\mathcal{K})$
p	68	68	65
p^n $(n > 1)$	23 969	4 410	1 388
$p(E)$	3 838 590	1 377 867	24 806
Total	3 862 627	1 382 345	26 259

The star patterns on an actual Knowledge Base

Table 4.2 shows the impact of θ on the number of items extracted from the Yago2 Knowledge Base using the facts and the transitive typing of the entities. Note that the number of instantiated items is huge largely because of the very precise typing of Yago. The precise distribution of the number of items per relation (multiplicity or instantiated items) is detailed in Appendix B.

We have computed the star patterns in Table 4.3 using the fpgrowth algorithm [35, 11] with a timeout of 2hours and 300Go of available memory. The experiment shows that for $Star_1$ and $Star_{10}$ with instantiations, fpgrowth was not able to compute the star patterns at all.

In the next section, we will discuss how to combine the star patterns found into new rules. Fortunately, we will see that the instantiations do not play any role in this process. Thus, we can restrict our analysis to star patterns without instantiations at first. However, we will also see that the process of combining star patterns into rules is also computationally challenging in itself.

4.4 Combining the star patterns into rules

In Section 4.3, we discussed the methodology to mine the set of satisfiable star patterns from the KB, i.e $Star_1(\mathcal{K})$. In this section, we consider we are given as input a set of star patterns Σ and try to construct every possible rule from it. Obviously, not every set of star patterns $\Omega \subseteq \Sigma$ will form a syntactically well-defined rule. Hence, we will first define a condition for a set of star patterns Ω to define a closed rule. Then, we will discuss the complexity of finding every subset Ω of Σ that defines a closed rule.

4.4.1 Compatibility of the star patterns

A set of star patterns is said to be "compatible" if it defines a closed rule.

Table 4.3 – Number of star patterns in Yago2. TO = Timeout, OOM = Out of Memory.

(a) Without instantiated items

Star patterns	$Star_1(\mathcal{K})$	$Star_{10}(\mathcal{K})$	$Star_{50}(\mathcal{K})$
Size 2	807	578	453
Size 3	4 710	2 691	1 592
Maximal	TO	33 411	13 135

(b) With instantiated items

Star patterns	$Star_1(\mathcal{K})$	$Star_{10}(\mathcal{K})$	$Star_{50}(\mathcal{K})$
Size 2	OOM	OOM	299 729
Size 3	TO	TO	2 327 815
Maximal	TO	TO	199 948

Existential definition of the compatibility

Given a set of star patterns $\Omega \subseteq \Sigma$, we have to decide whether this set can be combined into a rule R we would want to mine in AMIE, i.e a closed connected rule without existential quantifiers.

Definition 4.4.1 (Compatibility condition)**.** A set of star patterns Ω can be combined into a rule R if there exists a surjective function:

$$\begin{aligned} \rho : \ \mathcal{V}(R) &\to \Omega \\ x_i &\mapsto sp^{\text{MULT+INST}}(x_i, R) \end{aligned}$$

Here, $\mathcal{V}(R)$ is the set of variables of the rule R.

A set Ω is said to be compatible if it can be combined into a rule R that is closed, connected and that does not use any existentially quantified variables.

In this definition, ρ is a function, which means that the star pattern of any variable of R must be in Ω. Moreover ρ is surjective, which means that every star pattern of Ω must describe at least one variable.

For example $\Omega_0 = \{\{marriedTo, marriedTo^{-1}\}\}$ can be combined into the rule:

$$marriedTo(y, x) \Rightarrow marriedTo(x, y)$$

as both variables have the star pattern $\{marriedTo, marriedTo^{-1}\}$.

Alternative definition. In order to decide whether Ω is compatible, we can equivalently look for a *bijective* function ρ, while considering at the same time Ω as a multiset. In this setting, every star pattern of Ω in the input will be mapped to exactly one variable of the combined rule. This greatly simplifies the design of an algorithm that check the compatibility of a set Ω.

Using this alternative definition Ω_0 is no longer compatible but the following multiset is:

$$L_0 = \{\{marriedTo, marriedTo^{-1}\}, \{marriedTo, marriedTo^{-1}\}\}$$

It is easy to verify if a rule R is described by a set of star patterns Ω, but the problem here is to find, given a set Ω, a proper rule R, which is not trivial.

An effective method to find a compatible rule

In order to find a compatible rule, we proceed by analysis-synthesis: we enumerate every necessary condition the rule must meet and deduce a method to construct such a rule.

Closure. A rule is closed if every variable appears at least twice and a variable appears twice if and only if the size of its star pattern is at least two. Thus we can ignore every star pattern of size one.

Connectivity. There is no way to know if the generated rule will be connected just from the star patterns. We will filter out the non connected rules after the generation.

Removal of the existential quantifiers. One cannot simply remove the quantifiers, as the previously existentially quantified variable must be replaced by another variable in our rule. Thus we need to have another star pattern in Ω that can describe this variable.

This last condition is very restrictive and allows us to constructively check the compatibility of a set of star patterns.

Definition 4.4.2 (Multiplicity of a signed relation in a star pattern)**.** Given a star pattern σ, the multiplicity of the signed relation p in σ is defined as:

$$mult(p, \sigma) = \begin{cases} 0 & \text{if } \forall n > 0 \; p^n \notin \sigma \\ max(n > 0 : p^n \in \sigma) & \text{otherwise.} \end{cases}$$

Proposition 4.4.3 (Compatibility condition)**.** A multiset of star patterns $L = \{\sigma_1, \ldots, \sigma_n\}$ is compatible if and only if for every signed relation p, there is a mapping $M_p : L \to L$ such that:

- $\forall i \in \{1, n\}$, σ_i has exactly $mult(p, \sigma_i)$ images by M_p.
- $\forall i \in \{1, n\}$, σ_i has exactly $mult(p^{-1}, \sigma_i)$ antecedents by M_p.

Such a mapping defines exactly the atoms with the p relation of the generated rule. More precisely, if $(\sigma_i, \sigma_j) \in M_p$ then the atom $p(\rho^{-1}(\sigma_i), \rho^{-1}(\sigma_j))$ will be in the constructed rule.

Example 1. Given the following set of star patterns:

$$\Omega_1 = \left\{ \begin{array}{l} \{type(Person), marriedTo, hasChild\} \\ \{hasChild, marriedTo^{-1}\} \\ \{hasChild^{-2}\} \end{array} \right\}$$

We first associate a variable symbol to each star pattern of this set:

(x) $\sigma_x = \{type(Person), marriedTo, hasChild\}$

(y) $\sigma_y = \{hasChild, marriedTo^{-1}\}$

(z) $\sigma_z = \{hasChild^{-2}\}$

As the variable x is subject of one *marriedTo* relation, we must find another variable *in* Ω_1 to play the role of the object of this relation. As only y can be an object of the *marriedTo* relation, we must have the atom $marriedTo(x, y)$ in the combined rule. Here $M_{marriedTo} = \{(\sigma_x, \sigma_y)\}$.

As the variable z is object of two *hasChild* relations, we must find two different variables in Ω_1 to play the role of the subject of this relation. Here $M_{hasChild} = \{(\sigma_x, \sigma_z), (\sigma_y, \sigma_z)\}$. From this we recover the combined rule[5]:

$$type(x, Person) \wedge marriedTo(x, y) \wedge hasChild(x, z) \Rightarrow hasChild(y, z)$$

Note that the instantiated atom does not play any role in the process.

Example 2. Now consider the following two sets of star patterns:

$$\Omega_2 = \left\{ \begin{array}{l} \{marriedTo, hasGender\} \\ \{hasChild, marriedTo^{-1}\} \end{array} \right\} \quad \Omega_3 = \left\{ \begin{array}{l} \{hasChild^{2}\} \\ \{hasChild^{-2}\} \end{array} \right\}$$

[5]In practice we only recover a conjunctive query where any atom may be used as a head atom.

In Ω_2, we can deduce that both variables are married together, but we can associate neither the gender of the first variable nor the child of the second variable to a star pattern of the set. Ω_2 is thus incompatible.

In Ω_3, the multiplicity enforces that the first variable must have two *different* children and the second variable must have two *different* parents. Thus even if we make the first variable the parent of the second, we cannot find a second child nor a second parent in our set. Thus Ω_3 is incompatible.

In Appendix C, we present an efficient[6] algorithm that decides constructively the compatibility of a set of star patterns and we prove its correctness.

Remark 1. A star pattern may contain a relation and its inverse (as $\{marriedTo, marriedTo^{-1}\}$). This means the mapping may map a star pattern to itself and those mappings would produce reflexive atoms in the corresponding rule.

Remark 2. Compatibility requires the existence of a mapping for every relation. However, there may exist multiple valid mappings for a relation. As such, every combination of valid mappings across the different relations generates a new candidate rule. Moreover, nothing prevents multiple combinations of mappings to generate equivalent rules.

4.4.2 Selection of compatible star patterns

Even if we can decide efficiently if a set of star patterns is compatible, the enumeration of all compatible subsets of Σ may remain intractable.

Proposition 4.4.4 (Necessary condition for the compatibility)**.** A list of star patterns $(\sigma_1, \dots, \sigma_n)$ is compatible only if for every relation p:

$$\sum_{i=1}^{n} mult(p, \sigma_i) = \sum_{j=1}^{n} mult(p^{-1}, \sigma_j)$$

If we index every unsigned relation of our KB as $(p_1, \dots, p_{|\mathcal{R}|})$ we can map every star pattern $\sigma \in \Sigma$ into a multidimensional integer space, using the function:

$$\begin{array}{rcl} f : \Sigma & \rightarrow & \mathbb{Z}^{|\mathcal{R}|} \\ \sigma & \mapsto & \begin{pmatrix} mult(r_1, sp_i) - mult(r_1^{-1}, sp_i) \\ \vdots \\ mult(r_{|\mathcal{R}|}, sp_i) - mult(r_{|\mathcal{R}|}^{-1}, sp_i) \end{pmatrix} \end{array}$$

[6] $O(n \times log(n))$ where n is the sum of the sizes of the different star patterns.

Doing so, we can recognize an instance of the zero sum subset problem in a multidimensional setting, which is NP-Complete.

Proposition 4.4.5 (As a zero sum subset problem)**.** A set Σ of star patterns contains a compatible subset only if:

$$\exists\{\sigma_{i_1}, \ldots, \sigma_{i_m}\} \subseteq \Sigma : \sum_{k=1}^{m} f(\sigma_{i_k}) = 0_{\mathbb{Z}^{|\mathcal{R}|}}$$

This proposition does not constitute a proof of complexity, but it strongly suggests that our problem is hard and given the size of our input (cf Table 4.3), probably intractable.

4.5 Conclusion

In this chapter, we defined the star patterns: simple patterns that can be extracted from a rule. We showed how these star patterns, that can be mined with relative efficiency using frequent itemset mining algorithms, can then be used to prune unsatisfiable rules.

Then, we introduced the compatibility condition that decides if a set of star patterns can describe a rule or not and presented a polynomial algorithm that decides the compatibility in Appendix C. However, we also showed that different compatible sets of star patterns can represent duplicate rules and that finding all compatible sets of star patterns is intractable. Thus, we cannot use this approach to generate uniquely all satisfiable rules in the general case.

That said, we will see in the next chapter how star patterns can be used to efficiently explore, and prune, the search space of rules of a specific shape: the path rules.

Chapter 5

Star patterns and path rules

5.1 Introduction

We showed in the previous chapter that generating the entire search space of rules using star patterns is not tractable in the general case. This is because the general case is too general and considers all satisfiable star patterns independently of their sizes. In practice, the bigger star patterns represent rules with a large number of atoms, which are rarely usable rules. Thus, in this chapter, we study simpler star patterns, those of size 2, and the simpler type of rules they generate: the path rules. The goal here is still to propose a method that will discard a priori unwanted rules, but this time working only on path rules.

Efficient path rule mining should greatly benefit the general task of rule mining for two reasons. First, the path rules are the hardest type of queries when it comes to computing the support and the confidence [30]. Second, as we will see in Section 5.4, a divide-and-conquer approach can be implemented as any rule can be decomposed into multiple conjunctive path queries (paths in a connected graph). Moreover, path rules constitute most of the output of AMIE in practice. For example on Yago2s, only 1 over the 97 output rules is not a path rule[1]. This is why multiple works in the literature also address specifically the problem of path rule mining [45, 13].

Contribution. In this chapter, we identify the two instrumental properties of path rules. First, we can produce all path rules without generating any duplicate if we enforce a simple constraint on the refinement operator. Sec-

[1]This is also mainly due to the skyline pruning strategy [31] that prevents a rule to be output if it does not have a better confidence than its output parents, the parent being a more general rule.

ond, given the simplicity of the star patterns of the path rules, the necessary condition and the compatibility condition of those star patterns can be used efficiently and constructively to generate a pruned search space. These two properties allow us to devise the "Pathfinder vanilla" algorithm that explores incrementally and efficiently the pruned search space of path rules, without duplicates to eliminate.

This chapter is structured as follows: In Section 5.2 we introduce path rules and describe our refinement operator that does not introduce any duplicate rule. From this, we derive the Pathfinder vanilla algorithm. In Section 5.3 we present the bipattern graph that represents the compatibility condition of the satisfiable star patterns of size 2. Then we demonstrate how it can be used a priori, to generate all satisfiable path rules, or incrementally, in the Pathfinder vanilla algorithm. Finally, in Section 5.4 we discuss how we could generalize the usage of the bipattern graph to other types of rules in future work.

In Chapter 6 we will further enhance the Pathfinder vanilla algorithm and experimentally compare the performance of this new approach with AMIE.

5.2 Path rules

Definition 5.2.1 (Path rules and path queries)**.** Path rules are connected rules where every variable appears at most twice. Path queries are connected conjunctive queries where every variable appears at most twice.

5.2.1 Path queries

Theorem 5.2.2 (Forms of path queries)**.** *Any closed path query has one of the following forms:*

$$\begin{aligned} R_c[r_1,\dots,r_n] &:= \; r_1(x_0,x_1)\wedge r_2(x_1,x_2)\wedge\cdots\wedge r_n(x_{n-1},x_0) \quad (\textit{close})\\ R_i(E_1,E_2)[r_1,\dots,r_n] &:= \; r_1(E_1,x_1)\wedge r_2(x_1,x_2)\wedge\cdots\wedge r_n(x_{n-1},E_2) \quad (\textit{inst}) \end{aligned}$$

Here $r_1,\dots r_n$ are signed relations, $x_0,\dots,x_n$ are variables and E_1 and E_2 are entities of the KB. They can be seen as refinements of an open path query, which has the form:

$$R_o[r_1,\dots,r_n] := r_1(x_0,x_1)\wedge r_2(x_1,x_2)\wedge\cdots\wedge r_n(x_{n-1},x_n) \quad (\textit{open})$$

Proof. We can transform any path query into a graph where the vertices are the variables of the rule (or the entities E_1 and E_2) and where the labeled directed edges between the vertices represent the atoms. As this graph represents a path query, it must be connected and the degree of each vertice is at most 2. Thus it is either a simple path or a simple cycle.

Reading one simple path from one end to another and using the signed relations to enforce the increasing order of the variables, we recover the open path queries. We proceed similarly with the simple cycles, choosing any vertice for the role of the extremity of the path, recovering the closed path queries. □

Closed variant. The path query $R_c[r_1, \dots, r_n]$ will also be called the closed variant of the path query $R_o[r_1, \dots, r_n]$.

Remark 3. There is actually a second form for the open path query, a half instantiated form but we will not consider this edge case here:

$$R_{oi}(E_1)[r_1, \dots, r_n] := r_1(E_1, x_1) \wedge r_2(x_1, x_2) \wedge \cdots \wedge r_n(x_{n-1}, x_n) \quad (oi)$$

Corollary 5.2.3 (Reduced notation)**.** *All path queries can be represented as $R_c[r_1, \dots, r_n]$, $R_i(E_1, E_2)[r_1, \dots, r_n]$ or $R_o[r_1, \dots, r_n]$. In other words, the form (R_c, $R_i(E_1, E_2)$ and R_o) and the list of signed relation used $[r_1, \dots, r_n]$ totally describes the path query.*

Examples. The open path query:

$$hasChild(x, y) \wedge hasChild(z, w) \wedge marriedTo(x, z)$$

has two extremities, y and w. We first rename the variables choosing one extremity, here y, as x_0 and rename the following variable linked to y, here x, as x_1, the other variable linked to x, here z, as x_2 and so on. Then we reorder the atoms and the variables in the atoms using the signed relations to recover the desired order on the variables $((x_0, x_1), (x_1, x_2), \dots, (x_{n-1}, x_n))$:

$$\begin{array}{rl} hasChild(x, y) \wedge hasChild(z, w) \wedge marriedTo(x, z) & \\ hasChild(x_1, x_0) \wedge hasChild(x_2, x_3) \wedge marriedTo(x_1, x_2) & (rename) \\ hasChild(x_1, x_0) \wedge marriedTo(x_1, x_2) \wedge hasChild(x_2, x_3) & (reorder) \\ hasChild^{-1}(x_0, x_1) \wedge marriedTo(x_1, x_2) \wedge hasChild(x_2, x_3) & (switch) \\ R_o[\ hasChild^{-1},\ marriedTo,\ hasChild\ ] & (as\_list) \end{array}$$

For the closed path query:

$$hasChild(x, y) \wedge hasChild(z, y) \wedge marriedTo(x, z)$$

we proceed similarly, here also choosing y as the "extremity" x_0:

$$\begin{aligned}
hasChild(x,y) \wedge hasChild(z,y) \wedge marriedTo(x,z) & \\
hasChild(x_1,x_0) \wedge hasChild(x_2,x_0) \wedge marriedTo(x_1,x_2) & \quad (rename) \\
hasChild(x_1,x_0) \wedge marriedTo(x_1,x_2) \wedge hasChild(x_2,x_0) & \quad (reorder) \\
hasChild^{-1}(x_0,x_1) \wedge marriedTo(x_1,x_2) \wedge hasChild(x_2,x_0) & \quad (switch) \\
R_c[\ hasChild^{-1},\ marriedTo,\ hasChild\ ] & \quad (as\_list)
\end{aligned}$$

Theorem 5.2.4 (Equivalent path queries)**.** *For any list of signed relations $[r_1,\dots,r_n]$ and any circular permutation ς, the following path queries are syntactically equivalent:*

$$\begin{aligned}
&R_c[r_1,\dots,r_n] \equiv R_c[r_n^{-1},\dots,r_1^{-1}] \\
&R_c[r_1,\dots,r_n] \equiv R_c[\ \varsigma(r_1,\dots,r_n)\ ] \\
&R_o[r_1,\dots,r_n] \equiv R_o[r_n^{-1},\dots,r_1^{-1}] \\
&R_i(E_1,E_2)[r_1,\dots,r_n] \equiv R_i(E_2,E_1)[r_n^{-1},\dots,r_1^{-1}]
\end{aligned}$$

Moreover, this list is exhaustive*. Two path queries are not syntactically equivalent if they are not equivalent under these transformations*[2]*.*

Proof. The choice of the extremity in the proof of Theorem 5.2.2 is not enforced and the different possibilities still represent the same initial rule.

The exhaustivity comes from the fact that a path query, or any of its equivalent rewritings, will be represented by the same graph in the construction of the proof of Theorem 5.2.2. □

Examples. Consider the previous example query:

$$hasChild(x,y) \wedge hasChild(z,w) \wedge marriedTo(x,z)$$

Using w as the extremity we recover the equivalent query:

$$R_o[\ hasChild^{-1},\ marriedTo^{-1},\ hasChild\ ]$$

Consider the previous closed query:

$$hasChild(x,y) \wedge hasChild(z,y) \wedge marriedTo(x,z)$$

We can still choose the extremity y but now choosing z as x_1 which gives us the equivalent rule $R_c[\ hasChild^{-1},\ marriedTo^{-1},\ hasChild\ ]$. Choosing z as extremity (and x as x_1) gives us the equivalent rule $R_c[\ marriedTo^{-1},\ hasChild,\ hasChild^{-1}\ ]$.

[2]or for the closed path queries, under any combination of the two transformations.

5.2.2 Path rules

Any path rule is one of those path queries where we have chosen one atom to take the role of the head. Thus a conjunctive query of length n can describe n different rules. We extend the notation introduced above to make the chosen head atom apparent:

Definition 5.2.5 (Path rule notation)**.** We write $R_o[r_1, \dots, \boldsymbol{r_i}, \dots, r_n]$ for the path rule $R_o[r_1, \dots, r_n]$ using the atom $r_i(x_{i-1}, x_i)$ as the head.

We write $R_c[r_1, \dots, \boldsymbol{r_i}, \dots, r_n]$ for the path rule $R_c[r_1, \dots, r_n]$ using the atom $r_i(x_{i-1}, x_i)$ as the head ($r_n(x_{n-1}, x_0)$ if $i = n$).

Using this notation, we can deduce the following proposition, which is actually a corollary of Theorem 5.2.4.

Proposition 5.2.6 (Unicity of closed path rules)**.** All closed path rules can be written univocally as $R_c[\boldsymbol{r_1}, \dots, r_n]$ where r_1 is an *unsigned* relation.

Algorithm 3: Pathfinder (vanilla)

```
   Input: maximum rule length: $l$, support threshold: $minS$,
          confidence threshold: $minC$
   Output: set of path rules: $rules$
 1 $q = $ Queue()
 2 $rules = \langle\rangle$
 3 for any unsigned relation $r_i$ do
 4     $q.push( [\boldsymbol{r_i}] )$
 5 while $|q| > 0$ do
 6     $[\boldsymbol{r_1}, \dots, r_n] = q.pop()$
 7     for any possible refinement $r_{n+1}$ of the rule $[\boldsymbol{r_1}, \dots, r_n]$ do
 8         if $support(R_o[\boldsymbol{r_1}, \dots, r_n, r_{n+1}]) > minS$ then
 9             if $n + 1 < l$ then
10                 $q.push([\boldsymbol{r_1}, \dots, r_n, r_{n+1}])$
11             if $support(R_c[\boldsymbol{r_1}, \dots, r_n, r_{n+1}]) > minS$
12               and $confidence(R_c[\boldsymbol{r_1}, \dots, r_n, r_{n+1}]) > minC$ then
13                 $rules.add(R_c[\boldsymbol{r_1}, \dots, r_n, r_{n+1}])$
14 return $rules$
```

Algorithm. We present the Pathfinder vanilla algorithm in Algorithm 3. In this algorithm, every rule is represented as a list of signed relations and refining a rule only consists of adding a signed relation to the end of this list. The queue of rules is initialized with all the rules containing only the head atom, using every possible unsigned relation. This way, according to Proposition 5.2.6, every closed path rule will be considered at most once, i.e no duplicate rules are generated.

5.2.3 Other path rules

In its current form, the Pathfinder vanilla algorithm only outputs the *closed* path rules satisfying the given thresholds *minS* and *minC*. On one hand, it is straightforward to adapt this algorithm to mine the path rules having the head atom as an extremity, i.e the rules of the form $R_o[\boldsymbol{r_1}, \dots, r_n]$ or $R_i(E_1, E_2)[\boldsymbol{r_1}, \dots, r_n]$[3], and it should be implemented in the near future.

On the other hand, mining the other path rules, i.e rules of the form $R_o[r_1, \dots, \boldsymbol{r_i}, \dots, r_n]$ or $R_i(E_1, E_2)[r_1, \dots, \boldsymbol{r_i}, \dots, r_n]$ is not easy with this algorithm. However, we should also point out that these rules can be considered less interesting than the former, in particular for fact prediction [5]. Indeed, those rules are "product rules" and would predict the head fact based on two *independent* conditions on the head variables. For example, the rule $R_o[r_1, \dots, \boldsymbol{r_i}, \dots, r_n]$ stands for:

$$r_i(x_{i-1}, x_i) \Leftarrow \underbrace{\bigwedge_{j=1}^{i-1} r_j(x_{j-1}, x_j)}_{\text{independent condition on } x_{i-1}} \quad \wedge \quad \underbrace{\bigwedge_{j=i+1}^{n} r_j(x_{j-1}, x_j)}_{\text{independent condition on } x_i}$$

5.3 Pruning the search space

Proposition 5.2.6 ensures that every possible closed path rule is considered at most once by Algorithm 3 (i.e there are no duplicates). Thus, we can use any signed relation as a "possible refinement r_{n+1}" (Line 7 of the algorithm) to ensure that all closed path rules are considered exactly once.

In this section, we will see how we can further restrict the "possible refinements r_{n+1}" using the star patterns to prune the search space before the computation of the support of the rules while still outputting every satisfiable closed path rule.

[3]On the specific case of instantiated rules, refining the rules of the form $R_{oi}(E_1)[\boldsymbol{r_1}]$ instead of the more general rules $R_o[\boldsymbol{r_1}]$ is also a promising strategy.

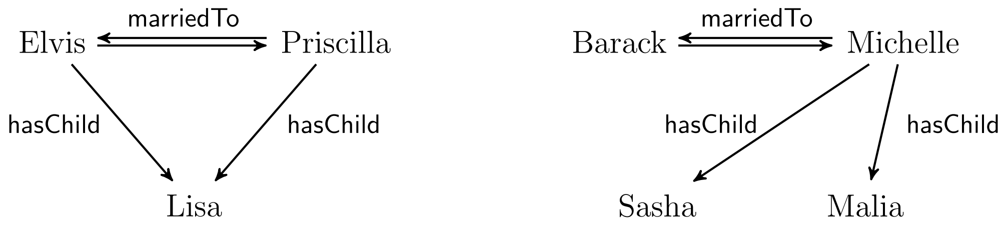

Figure 2.1 – Example KB (Reproduced here)

5.3.1 Star patterns and path rules

Proposition 5.3.1 (Star patterns of a path rules)**.** The star patterns of the variables of a path rule are:

$$\begin{aligned} \forall i \in \{1, n-1\}, \quad sp(x_i, R_c[r_1, \ldots, r_n]) &= sp(x_i, R_o[r_1, \ldots, r_n]) \\ &= \{r_i^{-1}, r_{i+1}\} \\ \text{and} \quad sp(x_0, R_c[r_1, \ldots, r_n]) &= \{r_n^{-1}, r_0\} \end{aligned}$$

For example, the star patterns of the path rule:

$$R_c[\ hasChild^{-1},\ marriedTo,\ hasChild\ ]$$

which stands for:

$$R : hasChild^{-1}(x_0, x_1) \wedge marriedTo(x_1, x_2) \wedge hasChild(x_2, x_0)$$

are:

$$\begin{aligned} sp(x_0, R) &= \{hasChild^{-2}\} \\ sp(x_1, R) &= \{marriedTo, hasChild\} \\ sp(x_2, R) &= \{marriedTo^{-1}, hasChild\} \end{aligned}$$

This proposition suggests some kind of cyclic structure that we describe below.

Definition 5.3.2 (The bipattern graph)**.** The "bipattern graph" of a KB $\mathcal{K}$ is the directed graph $(\widetilde{R}, E)$ containing a directed edge (r_i, r_j) (resp. an edge (r^{-1}, r)) if and only if the star pattern $\{r_i^{-1}, r_j\}$ (resp. $\{r^2\}$) is satisfiable in $\mathcal{K}$.

The star patterns of size 2 of the Example KB (Figure 2.1) are depicted in Table 5.1a with the entities satisfaying those star patterns. Each star pattern of size 2 $\{r_i^{-1}, r_j\}$ will generate two edges in the bipattern graph: an edge (r_i, r_j) and a dual edge (r_j^{-1}, r_i^{-1}). For example, the star pattern $\{marriedTo^{-1}, hasChild\}$ will give us two edges in the bipattern graph: $(marriedTo, hasChild)$ and $(hasChild^{-1}, marriedTo^{-1})$. The resulting bipattern graph is depicted in Figure 5.1b.

Star patterns of size 2	Entities
$\{marriedTo, marriedTo^{-1}\}$	$\{Elvis, Priscilla, Barack, Michelle\}$
$\{marriedTo, hasChild\}$	$\{Elvis, Priscilla, Michelle\}$
$\{marriedTo^{-1}, hasChild\}$	$\{Elvis, Priscilla, Michelle\}$
$\{hasChild^{-2}\}$	$\{Lisa\}$

(a) Star patterns of size 2

$$\begin{tikzcd}
 & hasChild \arrow[dd] & \\
marriedTo \arrow[loop above] \arrow[ur] & & marriedTo^{-1} \arrow[loop above] \arrow[ul] \\
 & hasChild^{-1} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

(b) Bipattern graph

Figure 5.1 – Transformation of the Example KB (Figure 2.1)

Theorem 5.3.3 (Bipattern graph and path rules)**.** *$R_o[r_1, \dots, r_n]$ (resp. $R_c[r_1, \dots, r_n]$) is satisfiable in a KB $\mathcal{K}$ only if the bipattern graph contains the path $r_1 \to \dots \to r_n$ (resp. the cycle $r_1 \to \dots \to r_n \to r_1$).*

This means that each cycle in the bipattern graph represents a satisfiable closed path query. For example the cycle:

$$hasChild^{-1} \to marriedTo \to hasChild \to hasChild^{-1}$$

represents the satisfiable path query:

$$R_c[\ hasChild^{-1},\ marriedTo,\ hasChild\ ]$$

and the cycle:

$$marriedTo \to marriedTo \to marriedTo$$

represents the path query $R_c[\ marriedTo,\ marriedTo\ ]$, i.e:

$$marriedTo(x, y) \wedge marriedTo(y, x)$$

Note that a path (resp. cycle) $r_1 \to \dots \to r_n$ represents the path query $R_o[r_1, \dots, r_n]$ (resp. $R_c[r_1, \dots, r_n]$) but its dual $r_1^{-1} \leftarrow \dots \leftarrow r_n^{-1}$ also represents the equivalent path query $R_o[r_n^{-1}, \dots, r_1^{-1}]$ (resp. $R_c[r_n^{-1}, \dots, r_1^{-1}]$).

From the star pattern perspective, an edge in the bipattern graph represents a star pattern of size 2 from our KB and the cyclic structure is the compatibility condition of a set of star patterns. Indeed, a path $r_{i-1} \to r_i \to r_{i+1}$ resolves the assignments for r_i as it necessarily introduces a bipattern containing r_i^{-1} (the edge $r_i \to r_{i+1}$) and another one containing r_i (the edge $r_{i-1} \to r_i$). As the compatibility condition must also be solved for the relation r_{i+1} we must follow a new edge from r_{i+1} and so on. Only cycles ensure compatibility for every pattern.

From a KB perspective, computing the possible bipatterns amounts to computing which relation range or domain effectively intersects with another relation range or domain, which is already done by AMIE+.

5.3.2 Generating all path rules

As sought out in Chapter 4, we can use the bipattern graph to generate, a priori, all satisfiable path rules (open or closed). More precisely, we can extract every path or cycle of the bipattern graph to find every satisfiable path query. Then, we just need to distinguish a query from its dual and choose a head atom as in Section 5.2.2.

We used this approach in the AMIE 3 experiments (Section 3.3) to generate the set of all possible closed path rules on Yago, DBPedia and Wikidata, generating "our" set of candidate rules for the Ontological Pathfinding algorithm. Originally, the Ontological Pathfinding algorithm uses the schema constraints to compute the set of possible rules. "Our" set of candidate rules is generated using the star patterns instead, which is more general than these schema constraints (cf. discussion in Section 4.3).

In Section 6.5, we will also define in our benchmark the "Star pattern" approach as a generate-and-test rule mining algorithm, that will generate all closed path rules using this approach and test their support and confidence using AMIE's methods.

5.3.3 Incremental generation of candidates

In practice, we propose three different methods to generate the "possible refinements r_{n+1}" in Line 7 of the Pathfinder vanilla algorithm (Algorithm 3). The method can be chosen using a command line option. The possible refinements of $R_o[\boldsymbol{r_1}, \dots, r_n]$ are, depending on the method:

Naive (ALL): Any signed relation is used as a possible refinement.

BiPattern existence (BP): Any signed relation r_{n+1} such that the edge (r_n, r_{n+1}) exists in the bipattern graph.

BiPattern Path existence (BPP): Given a maximal rule length l, any signed relation r_{n+1} such that there exist an edge (r_n, r_{n+1}) and a path of length at most $l-n$ between r_{n+1} and r_1 in the bipattern graph.

In addition to the star pattern condition, the bipattern path existence method captures whether a rule $R_o[\boldsymbol{r_1}, \ldots, r_n, r_{n+1}]$ is "closable" within $l-n$ refinements or not and prunes the rule in the latter case. As the distance (up to l) between the nodes of the bipattern graph can be precomputed, this last method is the most efficient in practice.

The exploration of the search space is still guaranteed to be exhaustive using any of these increasingly restrictive methods.

5.4 The bipattern graph and other rules

As stated in the introduction, any rule can be decomposed into multiple path rules. Thus, for a rule to be satisfiable, all of the path rules that compose it must be satisfiable. In particular, all of these path rules must meet the conditions of Theorem 5.3.3 and appear in the bipattern graph. As a consequence they create together a complex, but recognizable, pattern in the bipattern graph.

Method. We can do a case by case study: given rules of a specific shape, we give the pattern that must appear in the bipattern graph for the rules to be satisfiable.

In the following, we illustrate this approach on two representative examples.

Example 1. Any rule with 3 atoms and 2 variables

$$r_1(x, y) \Leftarrow r_2(x, y) \wedge r_3(x, y)$$

can be decomposed into 3 path rules/conjunctive queries:

$$\begin{gathered} r_1(x, y) \Leftarrow r_2(x, y) \\ r_1(x, y) \Leftarrow r_3(x, y) \\ r_2(x, y) \wedge r_3(x, y) \end{gathered}$$

These can be represented as 3 cycles and 3 dual cycles in the bipattern graph of the KB, resulting in the pattern depicted in Figure 5.2.

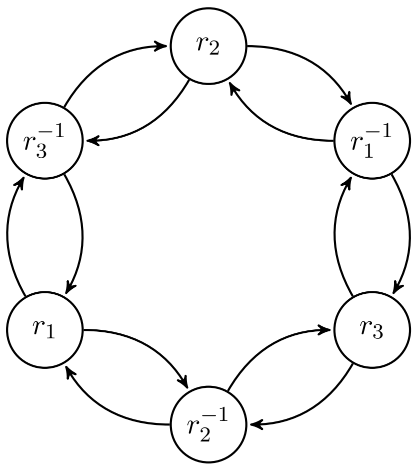

Figure 5.2 – Pattern in the BP Graph for a rule with 3 atoms and 2 variables

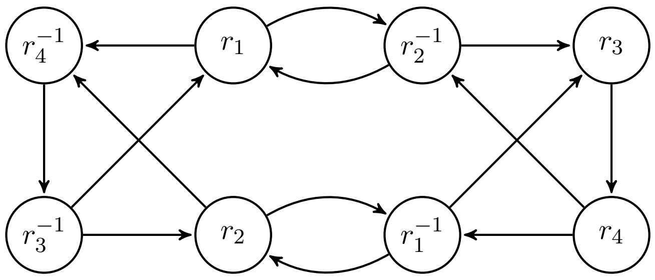

Figure 5.3 – Pattern in the BP Graph for Example 2

Example 2. Any rule with 4 atoms and 3 variables

$$r_1(x, y) \Leftarrow r_2(x, y) \wedge r_3(x, z) \wedge r_4(z, y)$$

can be decomposed into two path rules/queries of size 3:

$$\begin{array}{l} r_1^{-1}(y, x) \Leftarrow r_3(x, z) \wedge r_4(z, y) \\ r_2^{-1}(y, x) \wedge r_3(x, z) \wedge r_4(z, y) \end{array}$$

and a path rule of size 2:

$$r_1(x, y) \Leftarrow r_2(x, y)$$

These can be represented as 2 cycles of size 3, one cycle of size 2 and their respective dual cycles in the bipattern graph, resulting in the pattern depicted in Figure 5.3.

Note that the bipattern graph cannot represent completely the star patterns of the rules above as it only represents their subsets of size 2. For example the star pattern of x in the rule of Example 1 is $\{r_1, r_2, r_3\}$ but the pattern in the bipattern graph only ensures that $\{r_1, r_2\}$, $\{r_1, r_3\}$ and $\{r_2, r_3\}$ are satisfiable which is a weaker necessary condition.

5.5 Conclusion

In this chapter we showed how the shape of the path rules allows us to devise a lean generation method that considers every path rule only once, getting rid of the need to eliminate duplicates. In addition, we demonstrated how the star patterns can be used to efficiently prune the search space.

Unfortunately, we will see in the experimental section of the next chapter (section 6.5) that this fine-tuned exploration method alone is not quite enough to make Pathfinder Vanilla significantly better than AMIE. However, we will see in the next chapter how some improvements on the computation of the quality measures can make this algorithm faster. These improvements are possible only through the Pathfinder vanilla approach that generates each candidate rule independently.

Chapter 6

Pathfinder: Efficient Path Rule Mining

6.1 Introduction

The Pathfinder Vanilla algorithm that we have seen in the last chapter is not consistently faster than AMIE on path rules. This is because AMIE performs the computation of the support of all the refinements of a rule as a whole and thus benefits from some data reusage. In Pathfinder Vanilla, there are less possible refinements to consider but the algorithm recomputes the support of each of them independently.

However, in a rule mining process the rules considered are not independent. Exploiting the dependencies between the rules can significantly speed up the computation of the different quality measures, as AMIE does for the computation of the support of the refinements of a rule. In particular, reusing information between related rules can not only helps computing the support of a rule but also the confidence of this rule or the support of its refinements. Moreover, the reused data can also provide quantitative bounds on the value of the different quality metrics, avoiding unnecessary and costly computations.

Instead of using the dependencies between the different refinements of a rule only once, as AMIE does during the bulk computation of the support, we propose to go even further: we will carry over the information used in the computation of the quality metrics of a rule to all the refinements of this rule. This way a refined rule will inherit data from its parent that will be reused to speed up the computation, or bound the value, of its own quality measures. In this sense, data can be passed on to the next generation. As a consequence, the longer a rule is, the more it should benefit from this approach.

Contribution. In this chapter we define three different type of heritable information from a rule to its refinements: the set of possible values of the variables, the iterated provenance and some bounds on the size of the former. The first two allow us to design more efficient measure computation strategies, which use the locally stored information to speed up the computation of the quality measures of the rules. The bounds on the size of the iterated provenance give us bounds on the value of the quality measures of the rules, allowing us to bypass the exact computation of those measures (a very costly operation) when it is not necessary. Then, using these optimizations, we develop the full Pathfinder algorithm, which consistently outperforms AMIE and its precursor Pathfinder Vanilla, by multiple orders of magnitude on longer rules.

This chapter is structured as follows: First, in Section 6.2 we describe how to efficiently compute the set of possible values of the variables and the iterated provenance from a path rule and we present the different measure computation strategies that make use of this data. Then, in Section 6.3 we introduce the local bounds used to speed up the Pathfinder algorithm, an algorithm that we extensively describe in Section 6.4 and experimentally evaluate in Section 6.5.

6.2 Heritable information

6.2.1 Notation

Here we define the functions *image* and *prov* in a KB $\mathcal{K}$ and introduce the equivalent "diamond" notation (that uses the symbol $\diamond$). Given any signed relation r, we define the image of an entity x by r as:

$$image(x, r) \equiv \{y \in \mathcal{K} : r(x, y)\} \equiv r(x, \diamond)$$

We extend this function and notation to a set X of entities:

$$image(X, r) \equiv \{y \in \mathcal{K} : \exists x \in X,\ r(x, y)\} \equiv r(X, \diamond)$$

As such the domain and range of r can also be written as:

$$\begin{aligned} dom(r) &= r(\diamond, \mathcal{K}) = image(\mathcal{K}, r^{-1}) \\ rng(r) &= r(\mathcal{K}, \diamond) = image(\mathcal{K}, r) \end{aligned}$$

The provenance of $Y \subseteq image(X, r)$ is defined as the subset of X that generated Y:

$$prov(X, r, Y) \equiv \{x \in X : \exists y \in Y,\ r(x, y)\} = r(\diamond, Y) \cap X$$

When we use this notation, we always consider we use a known KB $\mathcal{K}$ given as input to the program.

6.2.2 The sets of possible values

The AMIE and Pathfinder Vanilla algorithms always consider every rule independently and never remember more than the value of the quality metrics for each given rule. This means that, for each new rule considered, they must recompute those quality metrics by testing possible instantiations of each variable.

But as stated in the introduction, rules are not created independently of each other and those algorithms do not exploit the core dependency between a rule and its further generation: every variable of a specialized rule can only take its value among the values the same variable took in the parent rule.

Example. Consider a rule and one of its refinements:

$$hasChild(x, y) \Leftarrow marriedTo(x, z)$$
$$hasChild(x, y) \Leftarrow marriedTo(x, z) \wedge hasChild(z, v)$$

In order to compute the support of the first rule, AMIE first determines the set of possible value of the variable x, and for every possible value of the this variable, it counts the number of corresponding values of the variable y. For the PCA body size, it counts the number of corresponding values of the variable z.

For the second rule, it will also compute the set of possible value of the variable x and count the corresponding y. But the possible values of x (or y or z) in the second rule must also be a solution of the less restrictive first rule to be solution of the second. We can reduce the strain of looking for candidate in the whole KB in the second rule by looking only among the candidates of the first rule.

Propagation. For any path rule

$$R_o[\boldsymbol{r_1}, \ldots, r_n] = \cdots \wedge r_{n-1}(x_{n-2}, x_{n-1}) \wedge r_n(x_{n-1}, x_n)$$

we denote by $X_0, \ldots, X_n$ the sets of possible values of the variables $x_0, \ldots, x_n$. During the refinement of this rule, i.e when the atom $r_n(x_{n-1}, x_n)$ is added,

Rule	Local Set	Provenance
$R_o[\boldsymbol{r1}]$	$X_1 \leftarrow r_1(\mathcal{K}, \diamond)$	$X_0 \leftarrow prov(\mathcal{K}, r_1, X_1)$
$R_o[\boldsymbol{r1}, r_2]$	$X_2 \leftarrow r_2(X_1, \diamond)$	$X_1 \leftarrow prov(X_1, r_2, X_2)$
	$\dots$	
$R_o[\boldsymbol{r1}, \dots, r_n]$	$X_n \leftarrow r_n(X_{n-1}, \diamond)$	$X_{n-1} \leftarrow prov(X_{n-1}, r_n, X_n)$

Table 6.1 – Computation of the local sets for $R_o[\boldsymbol{r1}]$ and its refinements

Pathfinder computes the set of possible values for x_n in $R_o[\boldsymbol{r_1}, \dots, r_n]$ as the image of X_{n-1} by r_n: $X_n = r_n(X_{n-1}, \diamond)$. Moreover, it also restricts further the values of X_{n-1} to the provenance of X_n: $prov(X_{n-1}, r_n, X_n)) = X_{n-1} \cap r_n(\diamond, X_n)$ in the refined rule. This process is summarized in Table 6.1 and illustrated in Figure 6.1, where the ellipses represent the different sets of entities and the arcs between ellipses represent the computation of the image of the different sets by a relation.

For example, in order to form the rule:

$$R_o[\ \boldsymbol{hasChild^{-1}},\ marriedTo,\ hasChild\ ]$$

on the Example KB (Figure 2.1), the Pathfinder algorithm will first consider the rule $R_o[\ \boldsymbol{hasChild^{-1}}\ ]$[1] and compute the sets:

$$X_1 = hasChild(\diamond, \mathcal{K}) = \{Elvis, Priscila, Michelle\}$$
$$X_0 = prov(\mathcal{K}, hasChild^{-1}, X_1) = \{Lisa, Malia, Sasha\}$$

Then, for the refined rule $R_o[\ \boldsymbol{hasChild^{-1}},\ marriedTo\ ]$, it will compute the set:

$$X_2 = marriedTo(X_1, \diamond) = \{Elvis, Priscilla, Barack\}$$

As every entity of X_1 is subject of the *marriedTo* relation, the set X_1 is unchanged. Finally, for the path rule

$$R_o[\ \boldsymbol{hasChild^{-1}},\ marriedTo,\ hasChild\ ]$$

Pathfinder computes the sets:

$$X_3 = hasChild(X_2, \diamond) = \{Lisa\}$$
$$X_2 = prov(X_2, hasChild, X_3) = \{Elvis, Priscilla\}$$

[1]the rule $hasChild(x_1, x_0) \Leftarrow$.

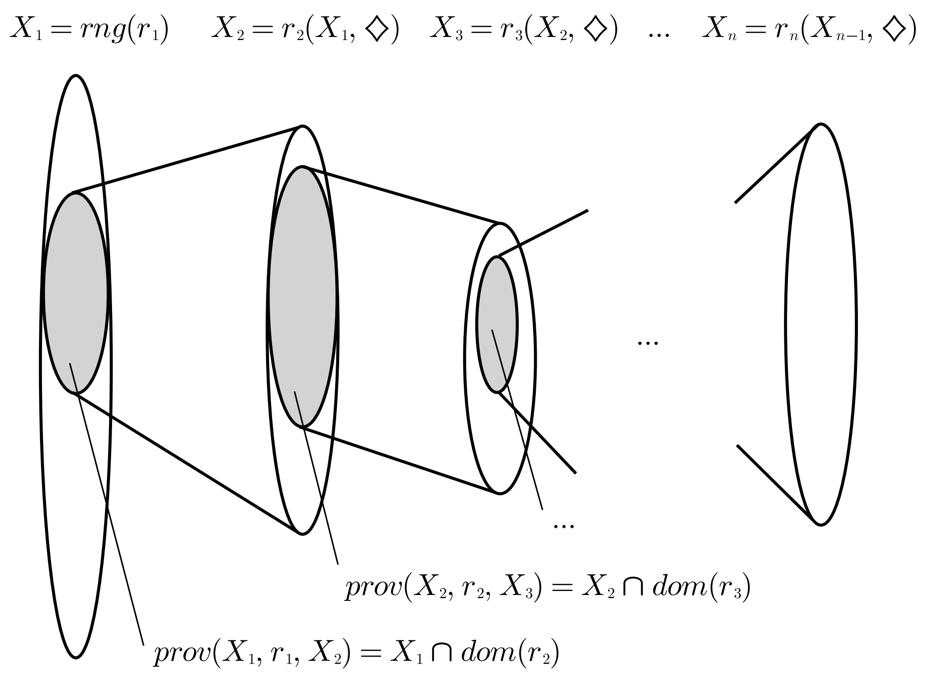
$X_1 = rng(r_1)$
$X_2 = r_2(X_1, \diamondsuit)$
$X_3 = r_3(X_2, \diamondsuit)$
...
$X_n = r_n(X_{n-1}, \diamondsuit)$
...
...
$prov(X_2, r_2, X_3) = X_2 \cap dom(r_3)$
$prov(X_1, r_1, X_2) = X_1 \cap dom(r_2)$

Figure 6.1 – Illustration of the computation of candidate sets

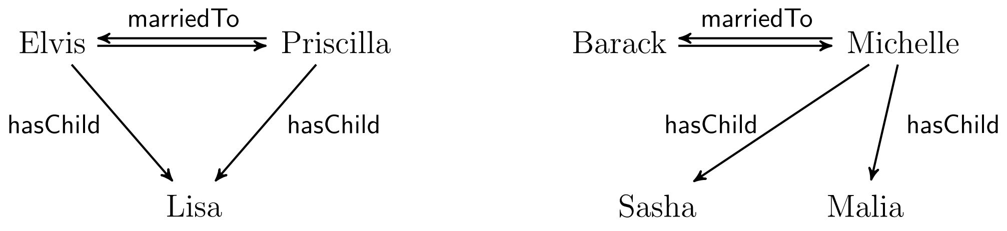

Figure 2.1 – Example KB (Reproduced here)

To conclude, the possible values of the variables of the last rule are, for the Example KB:

$$\begin{aligned} X_0 &= \{Lisa, Malia, Sasha\} \\ X_1 &= \{Elvis, Priscilla, Michelle\} \\ X_2 &= \{Elvis, Priscilla\} \\ X_3 &= \{Lisa\} \end{aligned}$$

First, these sets can be used for early pruning because if a set X_i is empty then the rule is unsatisfiable. Second, they will be used in Section 6.2.5 to speed up the computation of the quality measures of this rule: using these sets, we know we do not need to consider *Barack* as a possible value for x_2 during the computation of the quality measures for example. In particular, for the closed rule

$$R_c[\ \boldsymbol{hasChild^{-1}},\ marriedTo,\ hasChild\ ]$$

the possible values of x_0 are further reduced to $X_0 \cap X_3$.

Note that Pathfinder needs to know only the set X_{n-1} to compute the new set X_n and that the same set X_{n-1} can be used for any refinement of the rule $R_o[\boldsymbol{r_1}, \dots, r_{n-1}]$. In our previous example, the rule $R_o[\ \boldsymbol{hasChild^{-1}},\ marriedTo\ ]$ can also be refined as

$$R_o[\ \boldsymbol{hasChild^{-1}},\ marriedTo,\ marriedTo\ ]$$

Knowing that we had $X_2 = \{Elvis, Priscilla, Barack\}$, the set X_3 for this new refinement would be $marriedTo(X_2, \diamond)$.

Proposition 6.2.1 (Path existence)**.** Given a rule $R_o[\boldsymbol{r1}, \dots, r_n]$, the set X_n and the provenance sets $X_0, \dots, X_{n-1}$ computed as in Table 6.1, X_n is exactly the set of entities x_n such that:

$$\exists x_0, \dots, x_{n-1} \in \mathcal{K} : r_1(x_0, x_1) \wedge r_2(x_1, x_2) \wedge \dots \wedge r_n(x_{n-1}, x_n)$$

And for any entities $x_0, \dots, x_{n-1}$ satisfying this relation, we have in particular: $x_0 \in X_0, \dots, x_{n-1} \in X_{n-1}$. We say that "there exists a path from a x_1 (in $rng(r_1)$) to x_n" or that "x_1 generated x_n".

Note that the possible values of the other variables $x_0, \dots, x_{n-2}$ are also impacted by the addition of the new atom $r_n(x_{n-1}, x_n)$, i.e the sets of possible values $X_0, \dots, X_{n-2}$ are not tight for the refined rule. However, recomputing those stricter restrictions would result in a prohibitive overhead in used memory and computation time with no guarantee those restrictions would be significantly tighter.

Overhead. Even computing just the new image X_{n+1} can be a high cost operation, and there is no guarantee that the computed set would be of any usage at all, for example if the generated rule is pruned right away.

In terms of space, those sets cannot be larger than the size of the domain (or range) of the relation the variable appears in, which is usually relatively small. However, we need to maintain one set per atom for each rule that is yet to be refined or yet to be considered for output, i.e for any rule in the working queue. That is the main reason we use a stack for q in Algorithm 4 instead of a queue. The depth-first search ensures these local sets are freed as soon as possible.

The sets of possible values can be used to reduce the scope of the count queries, but the count queries still have to be performed. In the next section, we introduce the iterated provenance which contains enough information in itself to compute the confidence metrics without querying the KB.

6.2.3 The iterated provenance

In this section, we show that if we know all the *pairs* (x_1, x_n) that satisfy the query: $r_1(x_0, x_1) \wedge \dots \wedge r_n(x_{n-1}, x_n)$ then we can compute the quality metrics of $R_o[\boldsymbol{r_1}, \dots, r_n]$ and $R_c[\boldsymbol{r_1}, \dots, r_n]$ directly (Proposition 6.2.4). We will also show that this information, which we call the iterated provenance of x_n, is heritable.

Definition 6.2.2 (Iterated provenance)**.** We define the iterated provenance of a rule $R_o[\boldsymbol{r_1}, \dots, r_n]$, given the sets $X_2, \dots, X_n$ computed as in Table 6.1, as:

$$\begin{array}{rl} \forall x_2 \in X_2, & \mathit{prov}_2(x_2) = \mathit{prov}(X_1, r_2, \{x_2\}) \\ \forall i \in \{3, \dots, n\}, \quad \forall x_i \in X_i, & \mathit{prov}_i(x_i) = \displaystyle\bigcup_{x_{i-1} \in \mathit{prov}(X_{i-1}, r_i, \{x_i\})} \mathit{prov}_{i-1}(x_{i-1}) \end{array}$$

Propagation. For a rule $R_o[\boldsymbol{r_1}, \dots, r_n]$ we compute locally and incrementally the sets:

$$\begin{aligned} X_2^{prov} &= \{(x_2, \mathit{prov}_2(x_2)) : x_2 \in X_2\} \\ &\dots \\ X_n^{prov} &= \{(x_n, \mathit{prov}_n(x_n)) : x_n \in X_n\} \end{aligned}$$

by computing unions of the iterated provenance of the entities of the previous generation as in the formula of Definition 6.2.2.
For example, we have for the rule $R_o[\, \boldsymbol{hasChild^{-1}}, \ \mathit{marriedTo}, \ \mathit{hasChild}\,]$:

$$X_2^{prov} = \left\{ \begin{array}{l} (\mathit{Elvis}, \mathit{prov}(X_1, \mathit{marriedTo}, \{\mathit{Elvis}\}) = \{\mathit{Priscilla}\}), \\ (\mathit{Priscilla}, \mathit{prov}(X_1, \mathit{marriedTo}, \{\mathit{Priscilla}\}) = \{\mathit{Elvis}\}), \\ (\mathit{Barack}, \mathit{prov}(X_1, \mathit{marriedTo}, \{\mathit{Barack}\}) = \{\mathit{Michelle}\}) \end{array} \right\}$$

$$X_3^{prov} = \{(\mathit{Lisa}, \{\mathit{Priscilla}\} \cup \{\mathit{Elvis}\})\}$$

as $\mathit{prov}(X_2, \mathit{hasChild}, \{\mathit{Lisa}\}) = \{\mathit{Elvis}, \mathit{Priscilla}\}$ and $\{\mathit{Priscilla}\}$ is the provenance of *Elvis* in X_2^{prov} and $\{\mathit{Elvis}\}$ is the provenance of *Priscilla* in X_2^{prov}.

Proposition 6.2.3 (Iterated provenance and path existence)**.** Given a rule $R_o[\boldsymbol{r_1}, \dots, r_n]$, the "iterated provenance" $prov_i(x_i)$ of any element $x_i \in X_i$ is the subset of X_1 that generated x_i, i.e all the entities $x_1 \in X_1$ for which there exists a path $r_1(x_0, x_1) \wedge \cdots \wedge r_i(x_{i-1}, x_i)$.

This means that there are two pairs of values for the variables (x_1, x_3) such that we have:

$$hasChild^{-1}(x_0, x_1) \wedge marriedTo(x_1, x_2) \wedge hasChild(x_2, x_3)$$

in the Example KB. These pairs are: $\{(Elvis, Lisa), (Priscilla, Lisa)\}$

Proposition 6.2.4 (Iterated provenance and quality measures)**.** The iterated provenance is sufficient to compute directly the support and confidence of a rule $R_o[\boldsymbol{r_1}, \dots, r_n]$ and its closed variant, as we have:

$$support(R_o[\boldsymbol{r_1}, \dots, r_n]) = \sum_{x_1 \in \bigcup prov_n(x_n)} \|r_1(\diamond, x_1)\|$$

$$support(R_c[\boldsymbol{r_1}, \dots, r_n]) = \sum_{x_n \in X_n \cap X_0} \|prov_n(x_n) \cap r_1(x_n, \diamond)\|$$

$$pca\text{-}body(R_o[\boldsymbol{r_1}, \dots, r_n]) = \sum_{x_n \in X_n} \|prov_n(x_n)\|$$

In our example, knowing that $X_3^{prov} = \{(Lisa, \{Priscilla\} \cup \{Elvis\})\}$, we can compute:

$$\begin{aligned} support(R_o[\ \boldsymbol{hasChild^{-1}},\ \ & marriedTo,\ hasChild\ ]) \\ &= \sum_{x_1 \in \{Elvis, Priscilla\}} \|hasChild(x_1, \diamond)\| = 2 \end{aligned}$$

$$\begin{aligned} support(R_c[\ \boldsymbol{hasChild^{-1}},\ \ & marriedTo,\ hasChild\ ]) \\ &= \|prov_3(Lisa) \cap hasChild(\diamond, Lisa)\| = 2 \end{aligned}$$

$$\begin{aligned} pca\text{-}body(R_o[\ \boldsymbol{hasChild^{-1}},\ \ & marriedTo,\ hasChild\ ]) \\ &= \|prov_3(Lisa)\| = 2 \end{aligned}$$

Overhead. If we consider the size of the sets $prov_i(x_i)$ that need to be computed and maintained locally, we notice that they are, taken together, as large as the PCA Body Size (cf. the formula to compute the PCA Body size above).

This value can be huge and the whole point of the lazy strategy introduced in Chapter 3 was to bypass the exact computation of this value when it grows too large. Here we have to compute all the sets $prov_i(x_i)$ exactly, which is costly and cannot be lazy.

6.2.4 Backtracking the iterated provenance

Since computing and storing the iterated provenance a priori may be too costly, we can also recompute the sets $\mathit{prov}_i(x_i)$ on the rules of interest, a posteriori. This way, it will prevent unnecessary computations while still taking advantage of Proposition 6.2.4.

Proposition 6.2.5 (Iterated provenance backtracking)**.** We define recursively the function *back-prov*$_i$ such that:

$$\begin{aligned} \forall i \geq 3,\quad \forall A_i \subseteq X_i,\quad & \mathit{back\text{-}prov}_i(A_i) = \mathit{back\text{-}prov}_{i-1}(\mathit{prov}(X_{i-1}, r_i, A_i)) \\ \forall A_2 \subseteq X_2,\quad & \mathit{back\text{-}prov}_2(A_2) = \mathit{prov}(X_1, r_2, A_2) \end{aligned}$$

This way we have: $\forall i \geq 2,\ \forall x_i \in X_i,\ \mathit{prov}_i(x_i) = \mathit{back\text{-}prov}_i(\{x_i\})$

In the previous example:

$$\begin{aligned} \mathit{back\text{-}prov}_3(\quad & \{\mathit{Lisa}\}) \\ & = \mathit{back\text{-}prov}_2(\mathit{prov}(\{\mathit{Elvis}, \mathit{Priscilla}\}, \mathit{hasChild}, \{\mathit{Lisa}\})) \\ & = \mathit{back\text{-}prov}_2(\{\mathit{Elvis}, \mathit{Priscilla}\}) \\ & = \mathit{prov}(\{\mathit{Elvis}, \mathit{Priscilla}\}, \mathit{marriedTo}, \{\mathit{Elvis}, \mathit{Priscilla}\}) \\ & = \{\mathit{Elvis}, \mathit{Priscilla}\} \end{aligned}$$

To compute the iterated provenance of *Lisa*, we need to compute the iterated provenance of its provenance, i.e $\{\mathit{Elvis}, \mathit{Priscilla}\}$. This set, and only this set, is required to compute the iterated provenance of *Lisa*: the provenance of a subset (for example $\{\mathit{Elvis}\}$) may return a the partial result (here $\{\mathit{Priscilla}\}$) while the provenance of a superset (for example $\{\mathit{Elvis}, \mathit{Priscilla}, \mathit{Barack}\}$) may return a wrong result (here the superset $\{\mathit{Elvis}, \mathit{Priscilla}, \mathit{Michelle}\}$). In this sense this backtracking method is valid and tight[2].

Thus we can compute the quality measures of any rule using Proposition 6.2.4 by backtracking the iterated provenance when necessary. Using this method, we no longer need to propagate the iterated provenance as in Section 6.2.3, which has a large overhead, and we can even compute PCA body size lazily.

Mixed strategy. However, computing the support of the closed rule using backtracking underperforms compared to the other approaches due to the additional join. Indeed, computing $\mathit{prov}_n(x_n) \cap r_1(x_n, \diamond)$ is not the optimal

[2]this is due to the fact that we consider for the provenance only the entities that contributed to a path in the first place, cf Proposition 6.2.1.

way to compute the subset of entities of X_1 that contributes to the support. This is why we introduce in the next section a mixed strategy which uses backtracking to compute all the quality measures of a rule except for the support of the closed rule.

6.2.5 Measure computation strategies

In our proposed algorithm Pathfinder, described in Section 6.4, the following measure computation strategies are available, via command line options:

Legacy: The legacy strategy uses the methods of AMIE 3 (Algorithm 2) to perform every count query, without using any of the local information computed above.

Restricted: The restricted strategy also uses the methods of AMIE 3. As Algorithm 2 performs recursively on the different instantiations of the rule, we restrict the values of the instantiations used to the possible values of each variables, limiting this way the number of recursive calls.

Restricted (Vanilla setting): We can also use the restricted strategy with the Pathfinder Vanilla algorithm (that does not compute any local set) using the star patterns to restrict the possible values of the variables (as in Proposition 4.2.3).

FullSet: The FullSet strategy computes and stores locally the iterated provenance for each rule and computes the different quality metrics using the formulae of Proposition 6.2.4.

Backtrack: The Backtrack strategy computes the quality measures using the formulae of Proposition 6.2.4, backtracking the iterated provenance when necessary.

Mixed Restricted: It uses the Restricted strategy to compute the support of the closed rule and the Backtrack strategy for the other measures.

The Legacy and Restricted computation strategies can also be used with the option "Laziest". Using this option, the algorithm will compute the support of the open rule lazily, up to the required support threshold.

6.3 The bounds on the iterated provenance

Approach. For a rule $R_o[\boldsymbol{r_1}, \ldots, r_n]$, we compute two weights for every entity $x_n \in X_n$:

1. the lower bound w_n^1 on the number of entities x_1 of X_1 that generated x_n (i.e the entities x_1 for which there exists a path $r_1(x_0, x_1) \wedge \cdots \wedge r_n(x_{n-1}, x_n)$ in the KB);
2. the lower bound w_n^y on the number of pairs $(x_0, x_1) \in X_0 \times X_1$ such that x_1 generated x_n and $r_1(x_0, x_1) \in \mathcal{K}$.

We will see how these weights can be used to bound the value of the quality measures of $R_o[\boldsymbol{r_1}, \ldots, r_n]$ and $R_n[\boldsymbol{r_1}, \ldots, r_n]$ in Section 6.3.2.

6.3.1 Computing the lower bounds

Definition 6.3.1 (The weights w_i^1 and w_i^y)**.** During the computation of the local sets $X_1, \ldots, X_n$ of a rule $R_o[\boldsymbol{r_1}, \ldots, r_n]$ (as in Section 6.2.2), we additionally compute the following weights:

$$\begin{aligned} \forall x_1 \in X_1, \quad & w_1^1(x_1) = 1 \\ & w_1^y(x_1) = \|r_1(\diamond, x_1)\| \end{aligned}$$

and we use the following propagation formula for both weights w^1 and w^y:

$$\forall i \geq 2, \forall x_i \in X_i, w_i(x_i) = \sum_{x_{i-1} \in prov(X_{i-1}, r_i, \{x_i\})} \frac{w_{i-1}(x_{i-1})}{\|r_i(x_{i-1}, \diamond)\|}$$

Propagation. In the same manner as for the iterated provenance, for a rule $R_o[\boldsymbol{r_1}, \ldots, r_n]$ we compute locally and iteratively the sets:

$$\begin{aligned} X_1^w &= \{(x_1, w_1^1(x_1), w_1^y(x_1) : x_1 \in X_1\} \\ &\cdots \\ X_n^w &= \{(x_n, w_n^1(x_n), w_n^y(x_n) : x_n \in X_n\} \end{aligned}$$

computing the weights of a generation using the propagation formula of Definition 6.3.1 on the weights of the previous generation. In Algorithm 4 below, we will actually store a map, mapping an entity to its corresponding weights.

Now, we will prove that these weights actually are the desired lower bounds, for this we introduce the following theorem:

Theorem 6.3.2 (Bound on the weights)**.** *The sum of a weight over a set of entities is bounded by the sum of its weight on its provenance, i.e for each weight:*

$$\forall Y \subseteq X_i, \sum_{x_i \in Y} w_i(x_i) \leq \sum_{x_{i-1} \in prov(X_{i-1}, r_i, Y)} w_{i-1}(x_{i-1})$$

with equality if $Y = X_i$.

Proof. By construction, every element of $prov(X_{i-1}, r_i, Y)$ and only those elements will contribute to the weights of the elements of Y.

For any element x_{i-1} of $prov(X_{i-1}, r_i, Y)$, Y contains at most (if $Y = X_i$, exactly) $\|r_i(x_{i-1}, \diamond)\|$ entities that are the images of x_{i-1} by r_i. Each of these will receive a contribution of $\frac{w_{i-1}(x_{i-1})}{\|r_i(x_{i-1},\diamond)\|}$ from x_{i-1} to their respective weight. Thus the total contribution of x_{i-1} to the sum of weights of Y is at most (if $Y = X_i$, exactly) $w_{i-1}(x_{i-1})$. □

Corollary 6.3.3 (Lower bounds)**.** *We have the desired lower bounds:*

$$\forall Y \subseteq X_i, \sum_{x_i \in Y} w_i^1(x_i) \leq \|\{x_1 : x_i \in Y,\ r_1(x_0, x_1) \wedge \cdots \wedge r_i(x_{i-1}, x_i)\}\|$$

$$\forall Y \subseteq X_i, \sum_{x_i \in Y} w_i^y(x_i) \leq \|\{(x_0, x_1) : x_i \in Y,\ r_1(x_0, x_1) \wedge \cdots \wedge r_i(x_{i-1}, x_i)\}\|$$

Proof. With $prov(X_{i-1}, r_i, Y)$ taking the role of Y, we can recursively bound the right term in Theorem 6.3.2 to deduce:

$$\forall Y \subseteq X_i,\ \sum_{x_i \in Y} w_i(x_i) \leq \sum_{x_1 \in \textit{back-prov}_i(Y)} w_1(x_1)$$

□

Overhead. Those weights can be computed directly when we compute the image of a local set X_i as above. Thus this computation does not change the computational complexity of our approach.

6.3.2 Bounding the value of the quality measures

Support of the open rule. The second result of Corollary 6.3.3 gives us a lower bound on the support of any path rule:

$$\sum_{x_n \in X_n} w_n^y(x_n) \leq \textit{support}(R_o[\boldsymbol{r_1}, \ldots, r_n])$$

If the lower bound is superior to the support threshold then we can directly proceed and refine the rule. Otherwise, we need to actually compute the support in order to decide whether to prune the rule or not.

Support of the closed rule. We deduce this bound directly from Proposition 6.2.4 which states:

$$support(R_c[\boldsymbol{r_1}, \ldots, r_n]) = \sum_{x_n \in X_n \cap r_1(\diamond, X_1)} \|prov_n(x_n) \cap r_1(x_n, \diamond)\|$$

Thus we have in particular the following weaker upper bound:

$$support(R_c[\boldsymbol{r_1}, \ldots, r_n]) \leq \sum_{x_n \in X_n \cap r_1(\diamond, X_1)} \|r_1(x_n, \diamond)\|$$

Then we can discard the rule if the computed upper bound does not meet the support threshold.

PCA confidence. The PCA body size is the number of pairs (x_1, x_n) such that there exists a path from x_1 to x_n. We can take as lower bound either the number of x_n or the number of x_1 for which such a path exists. The first number is simply the size of X_n and the second is, according to Corollary 6.3.3, superior to $\sum_{x_n \in X_n} w_n^1(x_n)$:

$$max(|X_n|, \sum_{x_n \in X_n} w_n^1(x_n)) \leq pca\text{-}body(R_o[\boldsymbol{r_1}, \ldots, r_n])$$

This gives us an upper bound of the PCA confidence, allowing us to discard the rules which do not pass the confidence threshold before computing the exact value of the confidence.

6.4 The complete Pathfinder algorithm

The Pathfinder algorithm presented as Algorithm 4 is based on the Pathfinder Vanilla algorithm (Algorithm 3). As such, its refinement method (Line 23 of Algorithm 4) can also be implemented using the different strategies presented in Chapter 5 (ALL, BP, or BPP).

The main additions compared to the vanilla version are the computation of the local information and the usage of the local weights for early pruning.

The sets of possible values of the variables are stored in the *Atom* structure defined Lines 1 to 5 of our algorithm. Any rule R is a list of such atoms. At each step of the exploration, the function *computeWeightedImage* (Line 26) computes the image of the last atom of the rule by a signed relation r_i and stores the computed sets X_i^w and $prov(X_{i-1}, r_i, X_i)$ in a new *Atom*. This atom is then pushed to the end of the list of atoms of the initial rule

Algorithm 4: Pathfinder (full)

```
Input: maximum rule length: $l$, support threshold: $minS$,
       confidence threshold: $minC$
Output: set of path rules: $rules$
struct {
    $Relation$ r;
    $Set\langle Entity\rangle$ prov;
    $Map\langle Entity, double \times double\rangle$ X;
} $Atom$;
// A rule is represented as a list of such Atoms
$S = Stack()$
for $each\ unsigned\ relation\ r_i$ do
    $S.push([Atom(\boldsymbol{r_i}, \mathcal{K}, \{(x, (1, |r_i(\diamond, x)|)) : x \in r_i(\mathcal{K}, \diamond)\}])$
$rules = \langle\rangle$
while $|S| > 0$ do
    $R = S.pop()$
    $lastAtom = R.last();\ \ r_n = lastAtom.r;$
    $headAtom = R.head();\ \ \boldsymbol{r_1} = headAtom.r;$
    if $(R.size() > 1$ and $BPGraph.hasEdge(r_n, \boldsymbol{r_1})$
      and $\sum_{y \in lastAtom.X \cap \boldsymbol{r_1}(\diamond, headAtom.X)} |\boldsymbol{r_1}(y, \diamond)| > minS$
      and $(sup = computeCloseSupport(R)) > minS)$ then
        $maxBody = (sup/minC) + 1$
        if $(|lastAtom.X| < maxBody$
          and $\sum_{x \in lastAtom.X} w^1(x) < maxBody$
          and $computeBodyUpTo(R, maxBody) < maxBody)$ then
            $rules.add(R)$
    if $R.size() < l$ then
        for $any\ possible\ refinement\ r_{n+1}\ of\ R$ do
            if $\sum_{x \in r_{n+1}(\diamond, \mathcal{K}) \cap lastAtom.X} w^y(x) > minS$
              or $computeOpenSupport(R + [r_{n+1}]) > minS$ then
                $nX, nProv =$
                  $computeWeightedImage(\mathcal{K}, lastAtom.X, r_{n+1})$
                $R_c = R + [Atom(r_{n+1}, nProv, nX)]$
                $S.push(R_c)$
return $rules$
```

Algorithm 5: computeWeightedImage

```
Input: a KB: $\mathcal{K}$, weighted map of entity: $X_{in}$, relation $r$
Output: weighted image of $X_{in}$ by $r$: $X_{out}$, provenance
        $prov(X_{in}, r, X_{out})$: $prov$
1  $X_{out} = Map()$
2  $prov = \langle\rangle$
3  for $x \in X_{in}$ do
4      $Y = compute(r(x, \diamond))$
5      if $|Y| > 0$ then
6          $prov.add(x)$
7          $(w^1(x), w^y(x)) = X_{in}.get(x)$ ;      // weights to propagate
8          for $y \in Y$ do
9              $(w^1(y), w^y(y)) = X_{out}.getOrDefault(y, (0, 0))$
10             $w^1(y) = w^1(y) + (w^1(x)/|Y|)$
11             $w^y(y) = w^y(y) + (w^y(x)/|Y|)$
12             $X_{out}.put(y, (w^1(y), w^y(y)))$
13 return $X_{out}, prov$
```

R to form the refined rule R_c. The actual computation of the image by the function *computeWeightedImage* is described in Algorithm 5 and is the main source of overhead of our algorithm compared to the vanilla version.

The bounds presented in Section 6.3.2 are used as follows: the support of the closed rule upper bound is used in Line 15, the confidence upper bounds in Lines 18 and 19 and the final bound, the lower bound of the support of the rule, is used in Line 24.

It is important to note that the weights are computed only once, by the function *computeWeightedImage*, and then stored in the *Atom* structure. Thus the instructions in Lines 19 and 24 access only the value of the weights stored in *lastAtom.X* and do not recompute them.

Finally, the implementation of the functions that computes the exact value of the quality measures (Lines 16, 20 and 25) depends on the measure computation strategy chosen among those presented in Section 6.2.5. The restricted and the backtrack approaches use the provenance sets stored in the *Atom* structure to restrict the mappings, the legacy method uses no additional information and the FullSet approach uses additional local information bundled in the *Atom* structure (not depicted here).

Dataset	Facts	Relations	Entities
Yago2	948 358	36	834 750
Yago2s	4 484 914	37	2 137 469
DBPedia 2.0	6 601 014	1 595	2 275 327
DBPedia 3.8	11 024 066	650	3 102 999
Wikidata 12-2014	8 397 936	430	3 085 248
Wikidata 07-2019	386 156 557	1 188	57 963 264

Table 3.1: Datasets

6.5 Experiments

In this section we will compare the performance of the full Pathfinder algorithm and the Pathfinder Vanilla algorithm, using AMIE 3 as a baseline. We will first perform an analysis of the performance of the different candidate generation methods. Then we will focus on the impact of the different measure strategies. Finally, we will study the scaling capabilities of the different algorithms.

6.5.1 Experimental Setup

For the sake of comparability, we used the exact same experimental setup as in our previous AMIE 3 experiments (section 3.3). We recall this setup briefly here and specify precisely the baselines used for comparison.

Data. We evaluated our approaches on YAGO (2 and 2s), DBPedia (2.0 and 3.8) and Wikidata 2014. For the scaling experiment, we used a recent dump of Wikidata from July 1st 2019. Table 3.1 recalls the numbers of facts, relations and entities of our experimental datasets.

System settings. All experiments were run on a Ubuntu 18.04.3 LTS with 40 processing cores (Intel Xeon CPU E5-2660 v3 at 2.60GHz) and 500Go of RAM. AMIE 3 and Pathfinder are implemented in Java 1.8 and use the same integer-based in-memory database to store the KB (using the fastutils library[3]).

[3]`http://fastutil.di.unimi.it/`

Baseline: AMIE-Path. We want to use AMIE 3 as a baseline. However, as AMIE 3 computes exhaustively all rules given some quality thresholds, it seems unfair to compare it with an algorithm that reports only a subset of these rules: the path rules. Thus, we use as baseline a modified version of AMIE 3, "AMIE-Path", that refines only the open rules[4]. In this way, AMIE-Path produces only path rules.

6.5.2 Pathfinder generation methods

Configurations

We use here the legacy methods to compute the quality measures for every algorithm and we report the total running time (wall time) taken to mine every closed path rule with the default parameters (maximal length of 3, minimal head coverage of 0.01, minimal PCA confidence of 0.1).

AMIE-Path. As AMIE, AMIE-Path computes the support of every rule of a new generation using bulk computation (cf Section 6.1), selects the rules above the support thresholds and performs duplicate elimination.

Star pattern. Here we first use the bipattern graph to generate all the possible path rules in our KB. Then we use AMIE to compute the support and the confidence of each rule, one-by-one. We expect this approach to underperform compared to an incremental approach.

Pathfinder-vanilla. We compare the performance of our three generation methods: ALL, BP and BPP. As those methods are increasingly selective, we expect our algorithm to be increasingly faster (BP faster than ALL and BPP faster than BP). It will also allow us to measure the tradeoff between the bulk computation of the support in AMIE and the independent computation of the support on smartly selected candidate relations of Pathfinder vanilla.

Pathfinder. We compare the performances of our three generation methods and expect similar speedup between the different methods than the vanilla algorithm. The variation of performances between the vanilla and the full version will come exclusively from the difference between the overhead of the computation of the weighted images and the speedup of using the bounds on the quality measures.

[4]We add the condition $\neg closed(R)$ to the test line 7 in Algorithm 1

Dataset	Pathfinder	Generation method		
		ALL	BP	BPP
Yago2	Vanilla	14.37s	10.31s	8.97s
	Full	13.50s	11.34s	8.18s
Yago2s	Vanilla	1min 20s	43.42s	38.98s
	Full	1min 05s	45.34s	44.46s
DBPedia 2.0	Vanilla	> 2h	31min 07s	6min 18s
	Full	> 2h	15min 43s	5min 59s
DBPedia 3.8	Vanilla	54min 48s	14min 53s	17min 02s
	Full	41min 02s	20min 54s	10min 31s
Wikidata 2014	Vanilla	6min 53s	4min 02s	3min 59s
	Full	12min 06s	3min 47s	2min 55s

Table 6.2 – Impact of the generation method on Pathfinder

Dataset	AMIE-Path	Star Pattern	P.-vanilla	Pathfinder
Yago2	23.34s	19.05s	8.97s	8.18s
Yago2s	1min 40s	56min 36s	38.98s	44.46s
DBPedia 2.0	6min 21s	> 2h	6min 18s	5min 59s
DBPedia 3.8	10min 00s	> 2h	17min 02s	10min 31s
Wikidata 2014	4min 38s	> 2h	3min 59s	2min 55s

Table 6.3 – Impact of the generation algorithm

Results

Table 6.2 shows the performances of Pathfinder using the different generation methods.

First, we observe the expected hierarchy of performances on most datasets: BPP is faster than BP, which is faster than ALL except on DBPedia 3.8 (Vanilla). The speedup between the approaches is more pronounced on DBPedia. As it has more relations, the search space is larger, thus the pruning coming from the BP graph has a more noticeable effect.

Second, if we compare Pathfinder and Pathfinder vanilla with the same generation method on the different datasets, we observe that the tradeoff between computing the local information and the speedup of the measure bounds is really dependent on the case considered. There is no clear-cut winner. However, if we compare the performances on the best performing generation method, then Pathfinder is faster than Pathfinder-vanilla on all datasets except Yago2s.

In Table 6.3, we confront the performances of our Pathfinder algorithms (using BPP) with the performances of our baseline (AMIE-Path) and of the Star Pattern approach. This last approach does not scale to larger KBs and is clearly outperformed by any algorithm using incremental generation of candidate. The performances of our baseline (using bulk computation) and Pathfinder (using BPP) are comparable, with a speedup of 2× on datasets with a small number of relations (Yago and Wikidata) and identical or slightly worse on datasets with more relations (DBPedia). Note that the bulk computation is way faster than the Naive approach (Pathfinder ALL), but the usage of BPP counteracts this effect overall.

To conclude, if we consider the different generation strategies: bulk computation vs BPP, Pathfinder vs Pathfinder-vanilla, they offer relatively similar performances for rule mining. However this comparison only stands using the legacy measure computation strategy on rules of size 3 and as such, does not make full use of the local information computed by Pathfinder.

6.5.3 Pathfinder measure computation strategies

The previous experiment used the “Legacy” measure computation strategy[5] for each algorithm. In this section, we present the performance of the other measure computation strategies, introduced in Section 6.2.5, available for the Pathfinder and Pathdinder Vanilla algorithms. We use the BPP generation method for both versions of the Pathfinder algorithm.

Results. In Table 6.4, we report the running time of our different measure strategies. In general, the usage of the Mixed Restricted computation method allows the Pathfinder algorithm to consistently outperform the other algorithms.

The Pathfinder Vanilla algorithm (using the restricted laziest strategy) competes with the Pathfinder algorithm on the Yago and the Wikidata datasets but underperforms on the DBPedia datasets. We believe this is due to the prolixity of small relations of DBPedia, increasing the selectivity of the the set of possible values of the variables and as such improving the beneficial impact of using these sets in Pathfinder.

As discussed in Section 6.2.4, these results confirm that the lazy approach of the Backtrack strategy performs better than the FullSet strategy on most

[5]i.e AMIE 3 base methods

Algorithm	Measure strategy	Running time				
		Yago2	Yago2s	DBPedia 2.0	DBPedia 3.8	Wikidata 2014
AMIE-Path	Legacy	25.02s	2min 12s	4min 34s	10min 22s	4min 18s
P.-vanilla	Legacy	8.97s	38.98s	6min 18s	17min 02s	3min 59s
	Restricted	7.79s	52.02s	2min 31s	5min 30s	1min 48s
	Rest. Laziest	**6.10s**	37.36s	2min 43s	2min 55s	**1min 16s**
Pathfinder	Legacy	8.18s	44.46s	5min 59s	10min 31s	2min 55s
	Restricted	6.46s	44.02s	2min 19s	4min 04s	1min 53s
	Rest. Laziest	6.86s	45.27s	2min 19s	2min 29s	1min 46s
	FullSet	8.40s	44.73s	8min 13s	10min 02s	2min 41s
	Backtrack	8.68s	44.94s	5min 46s	8min 23s	3min 17s
	Mixed Rest.	**6.14s**	**37.14s**	**1min 53s**	**1min 20s**	**1min 16s**

Table 6.4 – Impact of the measure strategy

	AMIE-Path	P.-vanilla	Pathfinder
Yago2	25.02s	**6.10s**	**6.14s**
Yago2s	2min 12s	**37.36s**	**37.14s**
DBPedia 2.0	4min 34s	2min 43s	**1min 53s**
DBPedia 3.8	10min 22s	2min 55s	**1min 20s**
Wikidata 2014	4min 18s	**1min 16s**	**1min 16s**
Wikidata 2019	**16h 43min**	> 48h	36h 12min

Table 6.5 – Running time on increasingly large datasets

	AMIE-Path	P.-vanilla	Pathfinder
Yago2	3min 55s	1min 15s	**10.77s**
Yago2s	> 6h	7min 00s	**1min 53s**
DBPedia 2.0	> 6h	2h 35min	**47min 31s**
DBPedia 3.8	> 6h	3h 20min	**23min 22s**
Wikidata 2014	> 6h	> 6h	**14min 20s**

Table 6.6 – Rule mining of rules with maximal length of 4

datasets and that the computation of the support of the closed rule is indeed the bottleneck of both approaches. This is why the mixed strategy consistently outperforms both of these approaches.

6.5.4 Scaling experiments

We distinguish two distinct scaling factors: the size of the Knowledge Base and the size of the search space. This last factor is mostly impacted by the number of relations of the KB and the length of the rules we aim to mine. Both have a significant impact on the performances of AMIE and of our proposed Pathfinder algorithms as they are exact and exhaustive rule mining algorithms.

In particular, we expect that an increasing number of facts per relation would have a negative impact on the performances of Pathfinder as it increases significantly the overhead of computing the image of the sets. However, the extensive reusage of the data between a rule and its refinement should greatly benefit the Pathfinder algorithm while mining longer rules and allow it to scale in the size of the search space as no other algorithms has achieved before.

Results. In Table 6.5, we present the performance of our approaches on increasingly large datasets. The Pathfinder algorithm outperforms AMIE-Path on all datasets except the bigger one Wikidata 2019, which has a large number of facts per relation. Interestingly, the Pathfinder algorithm performs better than the Pathfinder Vanilla algorithm on this dataset, which means that computing the sets of possible values of the variables is actually beneficial overall and that the main overhead comes from the independent computation of the support of the refined rules (in comparison to AMIE-Path).

On a larger search space however, the Pathfinder algorithm achieves to compute all the path rules of length 4 on all datasets within less than hour, which is unprecedented. In comparison, we show in Table 6.6 that AMIE-Path cannot complete in less than 6 hours on all datasets except Yago2 and the Pathfinder Vanilla algorithm cannot complete in less than 1 hour an all datasets except the Yago datasets. This demonstrates the impact of our optimizations in the Pathfinder algorithm: the lower bounds on the iterated provenance, the restricted computation of the support of the closed rule and the backtracking of the iterated provenance for the computation of the other metrics.

Finally, the Pathfinder algorithm can scale to even longer rules: for example it mines all path rules of length at most 5 in 2 minutes and 30 seconds on Yago2 and in less than 37 minutes on Yago2s.

6.6 Conclusion

In this chapter presented the algorithm Pathfinder which mines exactly and exhaustively all the path rules from a KB. It uses the unique generation process and the pruning capabilities of the bipattern graph presented in the last chapter and exploits a data reusage technique introduced in this chapter to deduce an optimal measure computation strategy and local bounds on the quality measures. This allows Pathfinder to outperform other rule mining approaches on most datasets, by orders of magnitude on longer rules.

Future works. The direct followup of this work would be to use the Pathfinder algorithm to mine path rules with instantiatied extremities (rules of the form $R_i(E_1, E_2)[\boldsymbol{r_1}, \dots, r_n]$). We expect an even more significant speedup on these rules than on longer rules.

However, more work is required to make Pathfinder scale to Knowledge Bases with a large number of facts per relations such as Wikidata 2019. Multiple directions can be considered: the usage of caching during the backtracking process or the implementation of an hybrid algorithm that computes the sets of possible values only if those sets are selective enough compared to the star patterns.

Chapter 7

Identifying obligatory attributes in a KB

In this chapter, we use the incomplete KB to mine rules about the real-world. This chapter is based on our following full paper:

- Jonathan Lajus and Fabian M. Suchanek. Are all people married? determining obligatory attributes in knowledge bases. In *WWW*, pages 1115–1124. International World Wide Web Conferences Steering Committee, 2018

7.1 Introduction

Knowledge Bases find applications in information retrieval, machine translation, and question answering. The usefulness of these applications depends on the data quality of the knowledge base. One important dimension of quality is the correctness of the data. But there is another important dimension: the completeness of the data – i.e., whether or not a statement about an entity is missing from the KB. Data completeness affects queries about cardinalities, about existence, and about top-ranked entities. For example, if the population of Tokyo is missing from the KB, then a query about the top-10 most populous cities in the world will return a factually wrong result.

If we knew that every city has to have a population, we could know that the reason for Tokyo's missing population is not that Tokyo does not have a population in the real world, but that the number was not added to the KB. We could thus alert the user that the data on which the query is computed is known to be incomplete. We say that the population is an *obligatory attribute* for the class *city*. Not all attributes are obligatory. For example, not every city has to be the capital of a region. The same goes for other classes: Every person has to have a birth date, but not every person has to be married.

If we were able to distinguish obligatory attributes from optional ones, we could easily identify where information is missing in the KB. This, in turn, could help the designers of the knowledge base focus their effort on completion on this particular data. For example, collaborative knowledge bases such as Wikidata could ask contributors specifically for the obligatory attributes of a new entity. Moreover, the obligatory attributes can give semantics to classes. For example, the characteristics of actors is that they act in a movie. Such information can help decide whether an entity belongs to a class or not, it can guide the process of taxonomy design, and it can help define schema constraints [42]. We note that even obligatory attributes with a few counter-examples would be helpful for these goals. For example, it is good to know that people generally have a nationality – even if there are some people who do not have one. Our goal is to find the rule rather than the exception.

It is not easy to determine whether an attribute is obligatory or not. Today's KBs contain not just people and cities, but literally hundreds of thousands of other classes. They also contain hundreds, if not thousands of attributes. It is thus infeasible to specify the obligatory attributes manually. It is also hard to find them automatically: In YAGO, e.g., 2% of soccer players have a club – and that is an obligatory attribute for professional soccer players. At the same time, 2% of people have a spouse – and that is an optional attribute. Using the available data to determine obligatory attributes thus amounts to generalizing from a few instances to all instances of a class. This is a very difficult endeavor – even for humans. The case of KBs is even more intricate, because most KBs do not explicitly say that a statement does not hold in reality. For example, the KBs do not say that Pope Francis is *not married*. Rather, they operate under the Open World Assumption: A statement may be missing from the KB either because it was not added, or because it does not hold in reality. Thus, we find ourselves with the task of generalizing from a few instances in the absence of counter-examples.

In this chapter, we present methods that can detect obligatory attributes automatically. Our key idea is to use the class hierarchy: Most modern KBs contain extensive class hierarchies (YAGO, e.g., contains 650,000 classes; DBpedia and Wikidata have manually designed taxonomies). And yet, the KBs use the class hierarchy mainly to specify domain and range constraints. They do not exploit the semantics of the hierarchy any further. Our idea is to make use of the classes to determine obligatory attributes. More precisely, our contributions are as follows:

- a formal definition of the problem of obligatory attributes
- a probabilistic model for the incompleteness of a KB
- an algorithm that can determine obligatory attributes automatically
- extensive experiments on different datasets with different competitors, showing that obligatory attributes can be detected with a precision of up to 90%.

We first present our approach discuss related work in Section 7.2 and the preliminaries in Section 7.3. Then we present a formal definition of our problem, describe our approach and define the probabilistic model it is based upon in Section 7.4. Finally we present our algorithm in section 7.5 and how it performs on actual datasets in section 7.6.

7.2 Related Work

Query Completeness. Much recent work [56, 48, 70] has investigated the completeness of queries when the completeness of the data is known. These approaches are orthogonal to our work, which aims to establish whether the data is complete in the first place.

Measuring Incompleteness. Several studies have confirmed that KBs are incomplete. A watermarking study [77] reports that 69%–99% of instances in YAGO and DBpedia lack at least one property that other entities in the same class have. In Freebase, 71% of people have no known place of birth, and 75% have no known nationality [19]. Wikidata is aware of the problem of incompleteness, and has developed tools to specify completeness [21], as well as tools to manually add completeness information [15]. Unlike our work, these approaches do not aim at determining completeness automatically.

Determining Incompleteness. Closest to our work, several approaches have recently taken to measure the incompleteness in knowledge bases [71, 29]. However, these works determine whether a particular subject (such as Emmanuel Macron) is incomplete with respect to a particular attribute (such as *birthDate*). Our work, in contrast, aims at determining whether an attribute is obligatory or not for a given class. It thus operates on the schema level.

Schema Mining. Other work has investigated the more general problem of schema mining [64, 83, 38, 6, 31]. The work of [64] mines domain and range constraints for relations. Our work, in contrast, mines relations that are obligatory for classes. [38] uses machine learning to find OWL class descriptions. However, they rely on negative facts given by the user, or on prior knowledge about the schema – while we require none of these. [83] mines Horn rules on a KB. However, this approach is not targeted towards sparse obligatory attributes. We use it as a baseline in our experiments. [31] also mines Horn rules, and can deal with sparse data. At the same time, it cannot mine rules with existential variables in the head. Any such rule would trivially have a confidence of 100% in their model, because the model makes the Partial Completeness Assumption. Another work [6] mines definitions of classes. This approach comes closer to our goal, but is not exactly targeted towards obligatory attributes. We use such an approach as a baseline in our experiments.

7.3 Preliminaries

Notations. Given a Knowledge Base $\mathcal{K}$ and a signed relation p, we note $p_{\mathcal{K}}$ the set of entities of $\mathcal{K}$ that satisfies the star pattern $\{p\}$ in $\mathcal{K}$. For example, $\mathit{presidentOf}_{YAGO}$ represents the set of subjects of the *presidentOf* relation in $YAGO$, containing notably the entities $Macron$ or *Obama*. A class $C_{\mathcal{K}}$[1] of $\mathcal{K}$ is the set of entities satisfying the star pattern $\{type(C)\}$ in $\mathcal{K}$. For example $Macron$ and *Obama* are members of the classes $Person_{YAGO}$ or $President_{YAGO}$. We note $\mathcal{C}_{\mathcal{K}}$ the set of classes of the KB $\mathcal{K}$.

Ideal KB. As in Section 2.2.4, we consider a (hypothetical) ideal KB $\mathcal{W}$, which contains all facts of the real world. With this, our work is in line with the other work in the area [56, 48, 70, 71, 29], which also assumes an ideal KB. The problems of determining how such a KB could look are discussed in [71].

Generalization Rules. A *generalization rule* for a KB $\mathcal{K}$ is a formula of the form $A \subseteq B$, where A and B are classes of $\mathcal{K}$, signed relations of $\mathcal{K}$, or intersections thereof. For example, if $President_{\mathcal{W}}$ is the class of presidents in the real world, then the following generalization rule says that all presidents are presidents of some country:

$$President_{\mathcal{W}} \subseteq \mathit{presidentOf}_{\mathcal{W}}$$

[1]always capitalized to distinguish it from the signed relations.

With this, we can already make a simple observation:

Proposition 7.3.1 (Heredity)**.** In any KB $\mathcal{K}$, for every class $C_{\mathcal{K}} \in \mathcal{C}_{\mathcal{K}}$, any subclass $D_{\mathcal{K}} \subseteq C_{\mathcal{K}}$, and every property p: if $C_{\mathcal{K}} \subseteq p_{\mathcal{K}}$ then $D_{\mathcal{K}} \subseteq p_{\mathcal{K}}$.

This proposition tells us that if a generalization rule holds for a class, it also holds for all subclasses. The *confidence* of a generalization rule is defined as

$$\mathit{conf}(A \subseteq B) = \frac{|A \cap B|}{|A|}$$

Finally, we can make a second simple observation:

Proposition 7.3.2 (Separation)**.** If $C_{\mathcal{K}} \subseteq p_{\mathcal{K}}$ for some class $C_{\mathcal{K}}$ of some KB $\mathcal{K}$ and some property p, then the following holds for any class $D_{\mathcal{K}}$ of $\mathcal{K}$:

$$\mathit{conf}(C_{\mathcal{K}} \cap p_{\mathcal{K}} \subseteq D_{\mathcal{K}}) = \mathit{conf}(C_{\mathcal{K}} \subseteq D_{\mathcal{K}})$$

7.4 Model

7.4.1 Problem Definition

Goal. In this chapter, we aim to find generalization rules of the form $C_{\mathcal{W}} \subseteq p_{\mathcal{W}}$. Such a rule says that every instance of the class C in the real world must have the property p in the real world. We call p an *obligatory attribute* of the class C. For example, we aim to mine

$$\mathit{Film}_{\mathcal{W}} \subseteq \mathit{directed}_{\mathcal{W}}^{-1}$$

Here, $\mathit{directed}^{-1}$ is an obligatory attribute for the class *Film*, i.e., every film has to have a director. The difficulty is to find such a rule in $\mathcal{W}$ by looking only at the data of a given KB $\mathcal{K}$. In the following, we write $C_{\mathcal{W}}$ for the class C in the real world, and $C_{\mathcal{K}}$ for the corresponding class in $\mathcal{K}$.

Baseline 1. One way to find obligatory attributes is assume that the KB is complete (Closed World Assumption) and correct. Under these assumptions, we can predict that an attribute p is obligatory for a class C if and only if all instances of C have p in the KB $\mathcal{K}$:

$$(C_{\mathcal{K}} \subseteq p_{\mathcal{K}}) \quad \overset{?}{\Rightarrow} \quad (C_{\mathcal{W}} \subseteq p_{\mathcal{W}})$$

This is how a rule mining system under the Closed World Assumption would proceed [83], if applied naively to our problem. In practice, however, KBs are rarely complete. They operate under the Open World Assumption. There will be hardly any property p that all instances of class C have.

Baseline 2. Another method would be to predict that an attribute p is obligatory for a class C, if the corresponding generalization rule has a confidence above a threshold θ in the KB $\mathcal{K}$:

$$\mathit{conf}(C_{\mathcal{K}} \subseteq p_{\mathcal{K}}) \geq \theta \quad \overset{?}{\Rightarrow} \quad (C_{\mathcal{W}} \subseteq p_{\mathcal{W}})$$

For example, if more than 90% of presidents in $\mathcal{K}$ have the property *presidentOf*, then we would predict that *presidentOf* is an obligatory attribute for the class of presidents. The problem is that an attribute may be very prevalent without being obligatory. For example, many film directors also acted in movies – but acting in a movie is not an obligatory attribute for film directors.

Baseline 3. Yet another idea (inspired by [6]) is to make use of the taxonomy. Given a property p and a KB $\mathcal{K}$, we can find the lowest class $C_{\mathcal{K}}$ of the taxonomy such that nearly all instances with p fall into that class. This is motivated by the contraposition of Proposition 7.3.2. Formally, the method predicts that, for any property p, for any class $C_{\mathcal{K}}$ of a KB $\mathcal{K}$, and for a threshold θ:

$$\begin{array}{ll} \mathit{conf}(p_{\mathcal{K}} \subseteq C_{\mathcal{K}}) \geq \theta \quad \wedge & \\ \forall D_{\mathcal{K}} \subset C_{\mathcal{K}} : \mathit{conf}(p_{\mathcal{K}} \subseteq D_{\mathcal{K}}) < \theta & \end{array} \overset{?}{\Rightarrow} \; (C_{\mathcal{W}} \subseteq p_{\mathcal{W}})$$

This approach will work well for properties whose domain is a class, such as the property *presidentOf* with the class *President*. However, it will work less well if the attribute applies to only a subset of the class. For example, every person x with $\exists y : \mathit{hasSpouse}(x, y) \in \mathcal{K}$ belongs to the class *Person*. Thus, the above confidence will be 1 for *hasSpouse* and *Person*, and the method will conclude that every person is married.

7.4.2 Our Approach

Our approach is based on the assumption that the incompleteness of the KB is distributed equally across all classes of the KB. The implications of such a hypothesis are better grasped on an example.

Consider the whole living French population. As humans, we know that around 40% of the population is married. We represent this in figure 7.1 by the rings uniting 20 couples among 100 french people. In our knowledge base however, only 24% of French people are married. This is due to the incompleteness of our KB that we model given the following process:

1. every French is represented by an entity in our KB
2. we have access to all marriage certificates of French people

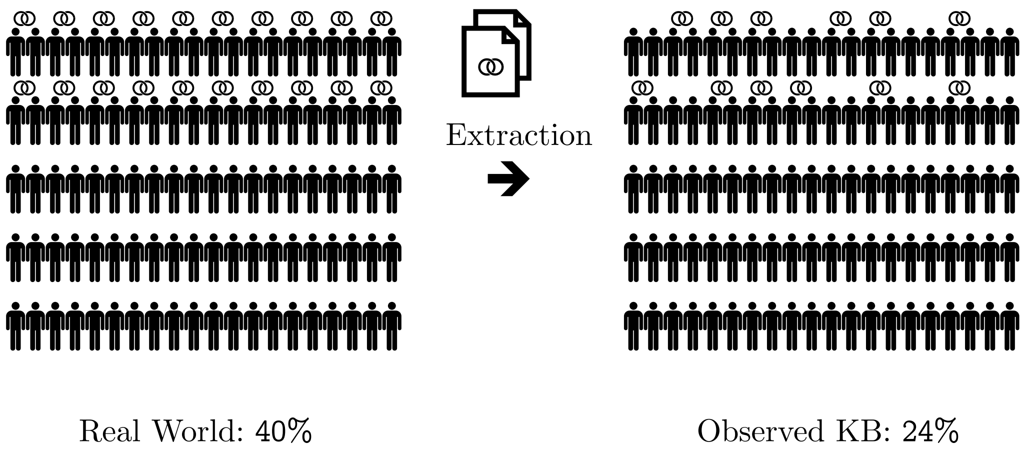

Figure 7.1 – Marriage example: Random sampling

Given this information, one would expect the KB to know about every marriage made in France. However, processing those marriage certificate is not an easy task. In order to be precise, the information extraction algorithm may take a conservative approach and extract a marriage fact only if it recognize the entities indubitably. The question is now, given only the extracted data, are all people married in the real world?

Random Sampling. The information extraction algorithm achieves to extract the marriage information from a sample of the marriage certificates. Thus, in the knowledge base, the married couples observed are a sample of the *married couples* of the real world. For example, in Figure 7.1, the information extraction algorithm achieved to identify only 60% of the facts from the marriage certificates, resulting in a KB with only 12 couples on the original 20.

In the general case, we assume that we know neither the percentage of married people in the real world nor the sensitivity of the information extraction algorithm (the percentage of successful extraction among the documents). However, the observed married couples are a sample of a very specific population, the population of married people of the real world. This means that the opposite population, the single people (in the real world), still induces a statistical bias in the observed distribution of marriages in our Knowledge Base that can provide valuable information.

Spread. The "spread" is an indicator of non-correlation of two different features. For example, the residence of an individual is usually uncorrelated

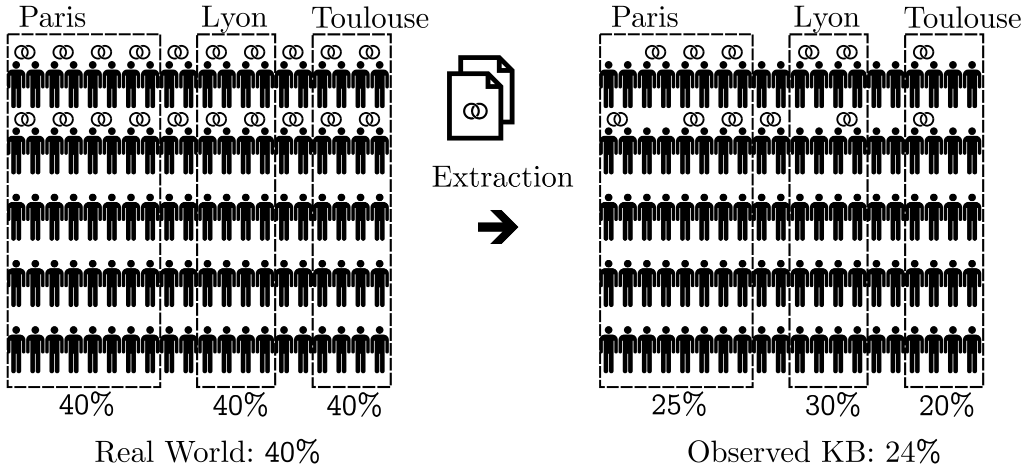

Figure 7.2 – Marriage example: Spread

with its marital status: the proportion of married Parisians should also be roughly 40% of all Parisians, the same for natives of Marseille or Toulouse. But as the observed marital status is a random sample of the actual marital status, the observed proportion of married Parisians should be similar to the observed proportion of married French, married Marseillais and married Toulousain, as in Figure 7.2. The observed proportion of married people "spreads" across the different place of residence, indicating that the latter and the marital status are uncorrelated.

Amplification. In contrast, when two features are correlated, for example the marital status and the age, we can observe a relative increase (or decrease) in the relative representation of married people in the different age group, as in Figure 7.3. As the proportion of married french teenagers is really low compared to the whole population, it is even lower in the observed sample, but more importantly, lower than the observed proportion in the population (0% versus 24%). On the contrary, the observed proportion of married adults is larger than in the whole population (40% versus 24%), we observe an "amplification" of the density.

To sum up, under the random sampling assumption, the observed relative proportion of married people in different classes follows the actual proportion of married people in that class. In particular, if the attribute is obligatory in a class, this proportion must be maximal: there should be an amplification compared to any superclass and a spread towards any subclass.

In our example, the spread of the "marriages" across all places of residence is a signal that being married may be obligatory for all french, more

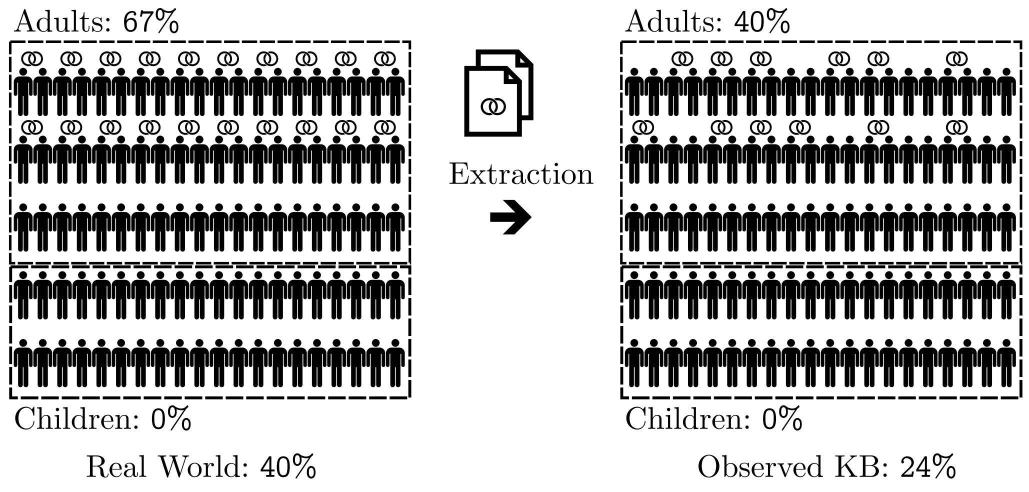

Figure 7.3 – Marriage example: Amplification

precisely it tells us it is not not obligatory for all french people. However, the amplification (and the inverse phenomenon in the opposite class) of the married attribute relatively to the age is a strong signal that not all french are married.

We will now show how to formalize this idea, and under which conditions we can guarantee that an attribute is not obligatory for a class.

7.4.3 Assumptions

In order to deduce formal statements about obligatory attributes in the real world from our KB $\mathcal{K}$, we have to make a number of assumptions about $\mathcal{K}$.

Assumption 1 (Correctness of the KB $\mathcal{K}$)**.** Every fact that appears in the KB $\mathcal{K}$ also appears in the ideal KB $\mathcal{W}$: $\mathcal{K} \subseteq \mathcal{W}$.

This assumption basically says that the KB does not contain wrong statements. This is a strong assumption, which may not hold in practice [78, 23]. However, we use it here mainly for our theoretical model. Our experiments will show that our method works even if there is some amount of noise in the data. We make a second assumption:

Assumption 2 (Class Hierarchy of the KB $\mathcal{K}$)**.** The classes of the KB $\mathcal{K}$ are correct and complete, i.e., $\mathcal{C}_{\mathcal{K}} = \mathcal{C}_{\mathcal{W}}$.

Again, this is a strong assumption that we use mainly for our theoretical model. In practice, three types of problems can appear. First, an instance can belong to a wrong class in the knowledge base. Second, an instance may be tagged with a too general class (e.g., *Macron* belongs to *Person*, but not to *President*). Finally, a class may be missing altogether (such as *Sciences-PoAlumni* for *Macron*). These problems impact our method, as we discuss in Section 7.6.5. However, for Wikidata, the class system that we use appears sufficiently complete and correct to make our method work. For YAGO, the data is known to be highly accurate [78], and furthermore, the Wikipedia categories are included in the class hierarchy. This makes the hierarchy sufficiently complete for our method to work. In DBpedia, in contrast, each instance is tagged with only one class. This results in so much incompleteness that our method cannot work. For example, in DBpedia, the proportion of singers who wrote a song is higher than the proportion of song-writers who wrote a song. This indicates that many singers should actually (also) be tagged as song-writers – which they are not.

Assumption 2 allows us to omit the subscript from the classes from now on. With Assumptions 1 and 2, we can already show:

Proposition 7.4.1 (Upper bound for Confidence)**.** Under Assumptions 1 and 2,

$$\mathit{conf}(C \subseteq p_{\mathcal{K}}) \leq \mathit{conf}(C \subseteq p_{\mathcal{W}})$$

for any KB $\mathcal{K}$, any class C, and any property p.

This proposition holds because Assumption 1 tells us that $x \in p_{\mathcal{K}}$ implies $x \in p_{\mathcal{W}}$. Furthermore, Assumption 2 tells us that the classes of $\mathcal{K}$ are the classes of $\mathcal{W}$.

7.4.4 Random sampling model

Our method assumes that the incompleteness of the KB is evenly distributed. More formally, let us consider the space of all possible KBs under Assumption 1. These are $\Omega = 2^{\mathcal{W}}$. We assume a probability distribution $\mathbb{P}(\cdot)$ over this space. Given a property p and instances s, o, the statement $p(s,o) \in \mathcal{K}$ becomes a boolean random variable defined on a KB $\mathcal{K}$, and we denote it by $p(s,o)$. In the same way, the expression $|p_{\mathcal{K}} \cap C|$ becomes a numerical random variable defined on a KB $\mathcal{K}$, and we denote it by $|p \cap C|$. Likewise, $\mathit{conf}(C \subseteq p_{\mathcal{K}})$ becomes a numerical random variable, and we denote it by $\mathit{conf}(C \subseteq p)$. We constrain $\mathbb{P}(\cdot)$ by the following assumption:

Assumption 3 (Random sampling)**.** On the space of all KBs in $\Omega = 2^{\mathcal{W}}$, there exists a probability l_p for each property p such that:

$$\forall x, y. \mathbb{P}(p(x,y)) = \begin{cases} l_p, & \text{if } p(x,y) \in \mathcal{W} \\ 0, & \text{otherwise} \end{cases}$$

The second case follows from Assumption 1. The first case states that facts with the property p in our KB come from a uniform random sampling of all true facts with property p in the real-world.

Several factors can thwart this assumption. First, the KB may be biased towards popular instances. For example, Wikipedia contains more information about American actors than about Polish actors, and people magazines are more concerned about the extra-marital affairs of actors than about the affairs of an architect. Thus, any KB that extracts from these sources will be biased. Second, the information extraction itself may have a bias. For example, several information extraction methods feed from the Wikipedia infoboxes. These infoboxes come in a number of pre-defined templates, and these templates define the properties. This entails that the presence or absence of a property in the KB depends on whether the instance happens to belong to an infobox template that defines this property or not. That said, making such simplifying assumptions about the probability distribution of facts is not unusual [19, 45, 61]. Our experiments will show that our model works also in cases where this assumption is violated to some degree.

We constrain $\mathbb{P}(\cdot)$ further by adding in the PCA (Equation A.2).

Assumption 4 (PCA)**.** On the space of all KBs in $\Omega = 2^{\mathcal{W}}$, $\mathbb{P}(\mathcal{K}) = 0$ if there exists a property p (which is not an inverse) and instances x, y, y' with $p(x,y) \in \mathcal{K}, p(x,y') \notin \mathcal{K}$ and $p(x,y') \in \mathcal{W}$.

The PCA is a common assumption for the KBs we consider [31, 19]. It has been experimentally shown to be correct in a large number of cases [30]. Again, we need the PCA mainly for our model. Our experiments will show that our method gracefully translates to scenarios where the PCA does not hold for all properties. In particular, our method is robust enough to work also with the inverses of properties, for which the PCA usually does not hold. In the appendix, we prove:

Theorem 7.4.2 (Random sampling under PCA)**.** Under Assumptions 3 and 4, for each property p with probability l_p (as given by Assumption 3),

$$\forall x : \mathbb{P}(\exists y : p(x,y)) = \begin{cases} l_p, & \text{if } x \in p_{\mathcal{W}} \\ 0, & \text{otherwise} \end{cases}$$

Theorem 7.4.2 tells us that the truth value of $\exists y : p(x,y) \in \mathcal{K}$ for an instance x in a KB $\mathcal{K}$ can be seen as a random draw of a Bernoulli variable with a parameter l_p. This allows us to derive

$$|p \cap C| \sim \sum_{x \in c, x \in p_{\mathcal{W}}} \textsc{Bernoulli}(l_p) = \textsc{Binom}(|p_{\mathcal{W}} \cap C|, l_p)$$

This allows for the following proposition.

Proposition 7.4.3 (Biased estimator)**.** The confidence of a generalization rule in Ω follows a binomial distribution divided by a constant:

$$conf(C \subseteq p) \sim \frac{\textsc{Binom}(|C \cap p_{\mathcal{W}}|, l_p)}{|C|}$$

Hence, the expected confidence of the rule in Ω is a biased estimator for the confidence of the rule in $\mathcal{W}$:

$$\mathbb{E}[conf(C \subseteq p)] = l_p \times conf(C \subseteq p_{\mathcal{W}})$$

This proposition confirms that, in our model, the confidence of $C \subseteq p_{\mathcal{W}}$ cannot be estimated from the data in our KB alone, as long as l_p remains unknown. The proposition also allows us to predict the behavior of Baseline 2 with parameter θ (see again Section 7.4.1). For a predicate p, if $\theta > l_p$, then the baseline is less likely to find all the correct classes for the predicate p, but the classes it finds have a high probability of being correct. We show in the appendix:

Proposition 7.4.4 (Unbiased estimator)**.** Given two classes C, D and a property p, the expected confidence of $C \cap p \subseteq D$ in Ω is an unbiased estimator for the confidence in $\mathcal{W}$:

$$\mathbb{E}[conf(C \cap p \subseteq D)] = conf(C \cap p_{\mathcal{W}} \subseteq D)$$

This proposition finally establishes a link between the (expected) observed confidence in our KB and the confidence in the real world.

7.5 Algorithm

In this section, we first define our main indicator score for obligatory attributes. We then present our algorithm and propose some variations of this algorithm.

7.5.1 Confidence Ratio

Our main indicator score for obligatory attributes is defined as follows:

Definition 7.5.1 (Confidence ratio)**.** Given a KB K, a property p, and two classes C and C' with $|C \cap C'| \neq 0$ and $|C \setminus C'| \neq 0$, the confidence ratio is

$$s_p^K(C, C') = \frac{conf(C \setminus C' \subseteq p_K)}{conf(C \cap C' \subseteq p_K)}$$

This expression compares the ratio of instances with p in $C \cap C'$ to the ratio of instances with p in $C \setminus C'$. It represents the influence of being in the class C' for the instances of a class C on p. This ratio is similar to the relative risk that is used in clinical tests. We can now make the following observation (which we prove in the appendix):

Proposition 7.5.2 (Main observation)**.** If p is an obligatory attribute for some class C, then for every class C' with $|C \cap C'| \neq 0$ and $|C \setminus C'| \neq 0$,

$$\mathbb{E}[s_p(C, C')] = 1$$

Here, $s_p(C, C')$ is a random variable, and hence does not carry the K. The observation tells us that the density of p in C should not be influenced by C' if p is obligatory for C. The probability of a KB where C' influences p is low in our probability space.

Measure instability. Our confidence ratio estimate will suffer from instability when the expected number of instances with a property p in an intersection is inferior to 1. In that case, there might be no instance with the property p in the intersection, and the confidence ratio will be infinite. In practice, this happens in small intersections for highly incomplete properties. Therefore, we decide to consider only *stable classes*. Given a class C and a property p in a KB K, an intersecting class C' is stable if either the expected number of instances ($conf(C \subseteq p_K) \times |C \cap C'|$) or the actual number ($|C \cap C' \cap p_K|$) is at least 1. The same has to hold for class differences.

7.5.2 Algorithm

Proposition 7.5.2 allows us to make statements about a generalization rule $C \subseteq p_{\mathcal{W}}$ in the real world purely by observing an incomplete knowledge base K. All we have to do is to check the classes C' that intersect with the class C. If the ratio of instances of p_K in $C \cap C'$ is very different from the ratio in $C \setminus C'$, then it is very unlikely that $C \subseteq p_{\mathcal{W}}$ holds. Furthermore, if $s_p^K(C', C) \gg 1$, then p cannot be obligatory for $C \cap C'$. Thus, it cannot be obligatory for C and C'.

These considerations give us Algorithm 6. This algorithm takes as input a KB K, a class C, and a property p. The algorithm also uses two thresholds: θ is the margin that we allow s_p^K to deviate from 1. The larger the threshold, the more obligatory attributes the algorithm will find – and the more likely it is that some of them will be wrong. The threshold θ' is the minimum support allowed for the rule $C \subseteq p$ to be considered. In practice, we set θ' to 100, as in AMIE [31]. Our algorithm returns *false* if the generalization rule $C \subseteq p_{\mathcal{W}}$

should be rejected – either because the support is too small (Lines 1-2), or because there is a stable intersecting class C' with $s_p^K(C, C') \neq 1$ (Lines 4-5), or $s_p^K(C', C) \gg 1$ (Lines 6-7). If neither is the case, the algorithm returns *true*.

Caveat. Our algorithm will return *true* if it finds no reason to reject a class. This, however, does not necessarily mean that the attribute is obligatory in this class. In particular, our algorithm may perform poorly if there is no class where the attribute is obligatory. However, our experiments show that despite this caveat, the method works well in practice.

Algorithm 6: ObligatoryAttribute

```
Input: KB $K$, class $C$, property $p$, threshold $\theta$,
       threshold $\theta' = 100$
Output: true if $C \subseteq p_{\mathcal{W}}$ is predicted
1 if $|C \cap p_K| < \theta'$ then
2     return false
3 for stable class $C'$ do
4     if $|log(s_p^K(C, C'))| > log(\theta)$ then
5         return false
6     if $log(s_p^K(C', C)) > log(\theta)$ then
7         return false
8 return true
```

Example. Consider again the example in Figure 7.3. On this example, we obtain:

$$s_{marriedTo}(All, Children) = +\infty$$

Thus, our algorithm will reject the hypothesis that *marriedTo* is be obligatory for the class *Children* or for the entire KB. Now consider the exemple in Figure 7.2, we have:

$$s_{marriedTo}(All, Paris) = 0.8$$

Since this value is closer to 1, we understand that to be married hardly depends on the fact of being Parisian or not. But when we compare the inhabitants of Paris with the Adults:

$$s_{marriedTo}(Paris, Adult) = 0$$

We deduce that not all Parisians are married. We need to consider every other intersection in order to conclude for the class *Adult*, for example we have:

$$s_{marriedTo}(Adult, Paris) = 0.66$$

7.5.3 Variations

Relaxation. In practice, classes in a KB intersect only in small areas. Thus, when $s^K_\phi(C, C') \gg 1$, we decided to reject only C'. In this relaxed variant, the condition in Line 4 of Algorithm 6 becomes $log(s^K_p(C, C')) < -log(\theta)$.

Fisher's Exact Test. We also experimented the Fisher's Exact Test [27] instead of the confidence ratio. We replace the logarithm of the confidence ratio s in Line 4 of Algorithm 6 by the probability that $C \cap C'$ has higher values, and in Line 6 with the probability that it has lower values over the set of possible contingency tables with fixed marginals.

7.6 Experiments

In this section, we evaluate our approach experimentally on large real-world KBs. We first evaluate our approach on YAGO, a KB for which we know that our Assumptions 1 and 2 hold by and large. Then, we submit our approach to a stress test: We run it on Wikidata, where less is known about our assumptions. Finally, we investigate how our approach could be generalized to composite classes.

7.6.1 Datasets

YAGO. We chose the YAGO3 knowledge base [50] for our experiments, because the data is of good quality (Assumption 1) and the taxonomy is extensive (Assumption 2). We use the facts of all instances, the full taxonomy, and the transitive closure of types. With this, our dataset contains more than 5 million instances and around 54,000 classes with more than 50 (direct or indirect) instances.

Wikidata. As a stress-test, we also evaluated our approach in Wikidata, where less is known about our Assumptions 1-4. We used the version from 2017-06-07, which contains more than 16,000 properties. This makes a manual evaluation impractical. Hence, we reduced the dataset to only people.

However, all people are in only one class: *Human*. Therefore, we used the *occupation* property (P106) to define classes. For example, in Wikidata, Elvis Presley (Q303) has the occupations *FilmActor*, *Actor*, *Singer*, *Screenwriter*, *Guitarist* and *Soldier*. These occupations form their own hierarchy, which we use as class hierarchy. For example, *FilmActor* is a sub-occupation of *Actor*, and thus becomes a subclass of it. This subset of Wikidata contains 1023 classes, around 1.6 million instances, and 2569 properties.

7.6.2 Gold Standard

Since our problem is novel, there is no previously published gold standard for it. Therefore, we had to construct a gold standard manually. For YAGO, we considered 68 properties (37 properties and their inverses), we determined the classes where more than 100 instances have the property, and we manually evaluated whether the attribute is obligatory or not. For Wikidata, we randomly selected 100 properties, and evaluated the output of each method manually. Our manual evaluation gives us an estimate for precision. Since Baseline 3 has a recall of 100% at maximal θ, we can use it to estimate our recall.

It is not always easy to determine manually whether an attribute is obligatory. For example, consider the attribute *isAffiliatedTo*. Is it obligatory for an artist to be affiliated to a museum, for a football player to be affiliated to a football club, or for people in general to be affiliated to their relatives? For our gold standard, we restricted ourselves to cases where we could clearly establish whether an attribute is obligatory or not, and removed all other cases.

Another problem arises for classes where the huge majority of instances have a particular attribute. For example, should we discard *hasNationality* as an obligatory attribute for *Person* because there exist stateless people? In such cases, we decided that the absence of the attribute is an exception to the rule that our method should not predict. Hence, we considered *hasNationality* obligatory. A related problem is that an attribute may not necessarily be obligatory for a class, but that de facto all instances have it. For example, we expect all instances of the class *RomanEmperor* to be dead by now, but what if a renewed Roman empire arises in the future? In such cases, we considered an attribute obligatory if de facto all known instances have it.

We constructed our gold standard according to these principles, and refer the reader to [71] for a more detailed discussion of such evaluations. All our datasets, as well as the gold standard and the evaluation results, are available at `https://suchanek.name/work/publications/www-2018-data`.

7.6.3 Evaluation Metric

The most intuitive way to evaluate the prediction of obligatory attributes would be to consider each predicted pair of a class and an attribute, and to compare this set to the gold standard. However, this comparison would not take into account the size of the class. For example, it is more important to predict that all organizations have a headquarters than that all Qatari ski champions have a gender, because there are many more of the former than of the latter. However, weighting each class by the number of instances causes another problem. Consider, e.g., the classes *Man* and *Woman*, which partition the class *Person* in our data[2]. If we predict that an attribute is obligatory for *Man* and for *Woman*, but not for *Person*, we would obtain a recall of only 50% – even though we predicted the attribute correctly for all instances.

To mitigate this problem, we compare, for each class C and for each property p, the actual set of predicted instances with the instances in the gold standard, i.e., we compare

$$P_p = \{x \in c | c \subseteq p_W \text{ predicted by our algorithm}\}$$

with

$$G_p = \{x \in c | c \subseteq p_W \text{ in the gold standard}\}$$

The true positives are the instances in the intersection of these sets. Then we compute the precision and recall as follows:

$$\mathit{precision} = \frac{\sum_p |P_p \cap G_p|}{\sum_p |P_p|}$$

$$\mathit{recall} = \frac{\sum_p |P_p \cap G_p|}{\sum_p |G_p|}$$

The F1-measure is computed as the harmonic mean of these.

7.6.4 YAGO Experiment

We ran all three baselines (Section 7.4) as well as our approaches (Section 7.5) on our YAGO dataset. Figure 7.4 shows the recall over the precision for each approach, with varying threshold θ.

[2]We are talking about a property of our data, not about genders in the real world.

Baseline 1. Recall that this baseline (inspired by [83]) labels an attribute as obligatory in a class, if all instances of the class have this attribute in the KB. This baseline performs like Baseline 2 at $\theta = 1$. Unsurprisingly, it has a very good precision, but a very bad recall: Only very few attributes (such as *label*) appear on all instances.

Baseline 2. This baseline relaxes Baseline 1 by labeling an attribute as obligatory if it is very prevalent in the class. For smaller θ, this method has a better recall than Baseline 1. However, it cannot exceed an F1 value of 45%. This is because there is no global threshold θ that would work well for all attributes. The baseline will work better if the KB is more complete. At the same time, the more complete the KB is, the less novel information there is to predict.

Baseline 3. This baseline (inspired by [6]) considers an attribute obligatory for a class if the vast majority of instances with that attribute fall in that class. The somewhat unusual curve comes from the fact that the baseline chooses the deepest class in the taxonomy where the target rule holds. While the method achieves slightly better F1 values (55%), its precision never exceeds 42%.

Confidence Ratio (Strict). This is our approach, based on the ratio of an attribute in a class and its intersections with the other classes (Algorithm 6). Different from Baseline 2, it delivers a very high precision (always $> 80\%$) – at the expense of somewhat lower recall. The best F1 measure is 37%.

Confidence Ratio (Relaxed). The relaxed variant of our method is less conservative. It trades off precision for higher recall. Indeed, we see that recall increases steadily with growing θ, while precision decreases gently. This allows for very good trade-offs between the two, with the maximum F1 value easily surpassing 55%. It is thus our method of choice.

Fisher's Test. This variation of our approach aims to make the Confidence Ratio less vulnerable to small data sizes. This is indeed what happens. However, the method errs on the side of caution: it has a very good precision (always $> 90\%$), but a mediocre recall. Hence, the best F1 value is quite low (12%). To increase this recall, the significance level of this test would have to be increased by a factor of several orders of magnitude, which would defy its purpose. The method should thus be seen as a stable, but inherently precision-oriented method.

Comparison. Figure 7.5 plots precision and recall for each of the methods across the spectrum of parameter values.[3] Baseline 3 achieves the highest recall. However, its precision never exceeds 42%, which makes the method unusable in practice. On the other side of the spectrum, Baseline 2 offers very good precision – but it cannot achieve good recall. Our relaxed confidence ratio occupies a sweet spot between the two: a precision between 75% and 95%, at a recall of 45% and 10%, respectively. It thus dominates the other methods in the mid-range between good recall and good precision.
Completeness of the attributes. By identifying classes in which an attribute is obligatory, our method identifies the entities that should have this attribute. If we compute the proportion of these entities that actually have the attribute in the data, we get an approximation of the completeness of the data. Table 7.1 shows the estimated completeness of the data according to different methods: the gold standard, Baseline 2 at different thresholds, and our method at different thresholds. We show 3 attributes that are obligatory in certain classes, and the deviation from the gold standard across all attributes. The small deviation for our method shows that we can approximate the real completeness quite well.

We can now also algorithmically answer the question raised in the introduction of our approach: No, not all people are married (in the real world). Our method finds that *isMarriedTo* is an optional attribute for the class *Person*. However, marriage is obligatory for the classes *Spouse* and *Royal-Consort*.

Table 7.1 – Approximation of completeness of attributes

		Baseline 2		CR (Relaxed)	
Attribute	Gold Standard	0.5	0.9	1.5	3
hasGender	0.58	0.51	0.91	0.79	0.58
wasBornIn	0.14	0.47	0.93	0.46	0.25
isMarriedTo	0.57	0.51	0.93	0.59	0.23
Avg-Squared error to GS (all p)		0.21	0.59	0.17	0.08

7.6.5 Wikidata Experiment

As a stress test, we also evaluated our method on Wikidata, where less is known about our assumptions. Figure 7.6 shows our results. We first note

[3]Different values for θ can give the same combination of precision and recall, whence the "loop" of Baseline 3.

that all methods exhibit a similar behavior to the YAGO experiment. Baseline 3 has high recall, low precision ($< 55\%$) and remains unstable. Baseline 2 performs well, with a precision of 97% and a recall of 33% for threshold $\theta = 0.7$. This indicates that some of the properties in our data are already highly complete. Our method performs similarly to Baseline 2 in the precision range of 97%. However, in precision range of 93%, it has a higher recall than Baseline 2.

7.6.6 Artificial Classes

In the following two experiments, we investigate how our algorithm performs on artificially constructed classes. For this purpose, we constructed classes that depend on the facts in our KB. Since the facts are incomplete, these classes are incomplete, too, and Assumption 2 no longer holds.

Life Expectancy. We construct artificial classes for all people born before a certain decade t:

$$C_t = \{x | \exists y : \mathit{birthDate}(x, y) \wedge y < t\}$$

These classes form a taxonomy:

$$C_t \subseteq C_{t+10}$$

In this way, we generated the classes C_t for $t = 1700, 1710, ..., 2020$ in YAGO. We can now mine obligatory attributes also on these artificial classes. In particular we mine the generalization rule

$$C_t \subseteq \mathit{deathDate}_{\mathcal{W}}$$

Table 7.2 shows the t for which the rule holds, according to our relaxed algorithm. We see that for a conservative $\theta < 3$ (which delivered high precision also in the previous experiments), we get again very good estimates for t. As θ increases, our method starts to believe that all people (even younger ones) should have a death date – as expected. This experiment shows that our approach has the potential to mine obligatory attributes even on intensionally defined classes.

Cardinality experiment. To expand even more on the effect of using intensionally defined classes, we ran our algorithm while considering every star pattern of the form $\{p^n\}$ as a class. We use the fact that $\{p^{n+1}\} \Rightarrow \{p^n\}$ to define an induced taxonomy.

Table 7.2 – Some results of the Life expectancy experiment

θ up to:	1.3	2.5	5.0	9.5	10	20	30
t mined:	1920	1930	1940	1950	1960	1970	1980

Table 7.3 – Results of the Cardinality experiment

Attribute	is obligatory for ...
$wasBornIn$	$created^{80}$
	$playsFor^{14}$
$isMarriedTo$	$hasChild^{8}$
	$actedIn^{49}$
$hasChild$	$isMarriedTo^{3}$
	$actedIn^{24}$

We added these classes to YAGO and we ran our algorithm with a small modification: for an attribute p, we never considered any class p^n for the intersections. In the end, our algorithm with threshold $\theta = \log(3)$ outputs 248 rules with those items.

As a result, the additional classes exacerbated the tendency our approach has to overfit (exemplified in Table 7.3). Some examples look reasonable, for example someone married thrice can be expected to have a child (compared to newly weds for example) but other results look odd, for example someone should be born if he created at least 80 items. This is because, to output this rule, our algorithm have rejected the rule that every person that created less than 79 items must have been born as well. Overfitting is a double-edged sword, allowing our algorithm to be precise, but at the expense of the recall when the classes are too specific.

7.7 Conclusion

In this chapter, we have introduced the general problem of mining obligatory attributes from knowledge bases. This is the problem of determining whether all instances of a given class have a given attribute in the real world – while all we have at our disposal is an incomplete KB. We have developed a new way to model the incompleteness of a KB statistically. From this model, we were able to derive the necessary conditions for obligatory attributes. Based on this, we have proposed an algorithm that can mine such attributes with a precision of up to 92%.

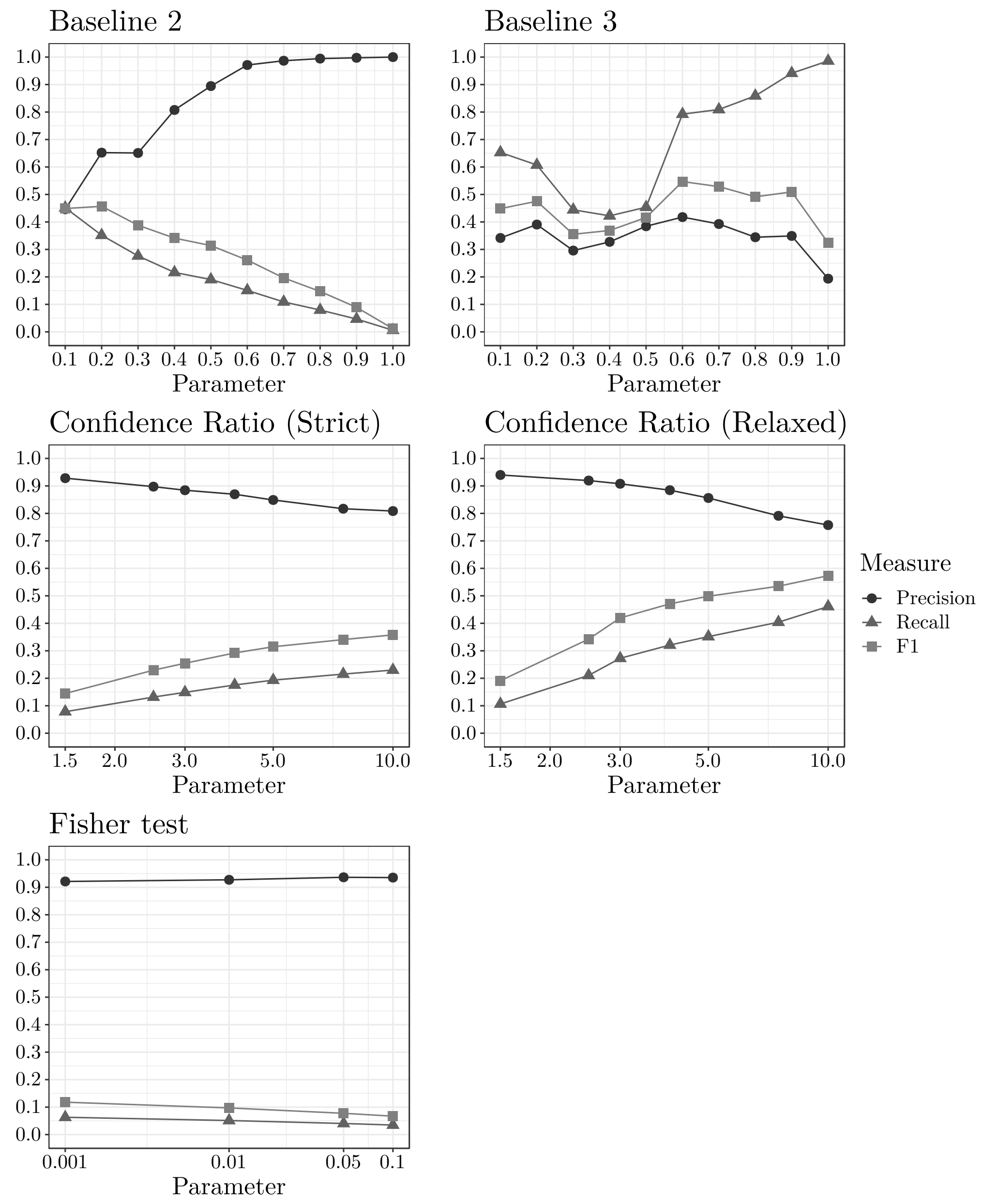

Figure 7.4 – Influence of θ on precision, recall and F1 for the different classifiers. Baseline 1 is Baseline 2 for parameter $\theta = 1$.

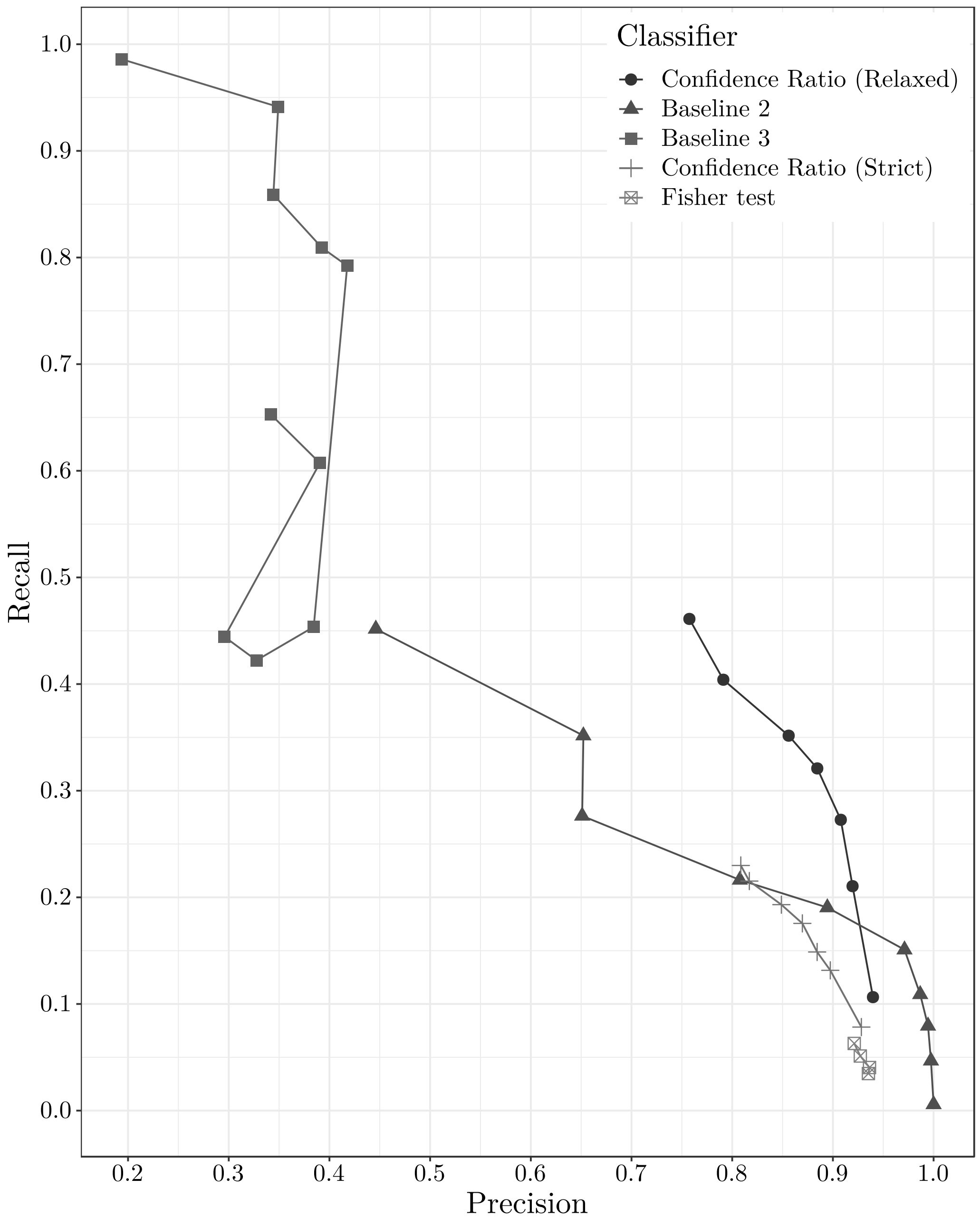

Figure 7.5 – Precision and Recall on YAGO

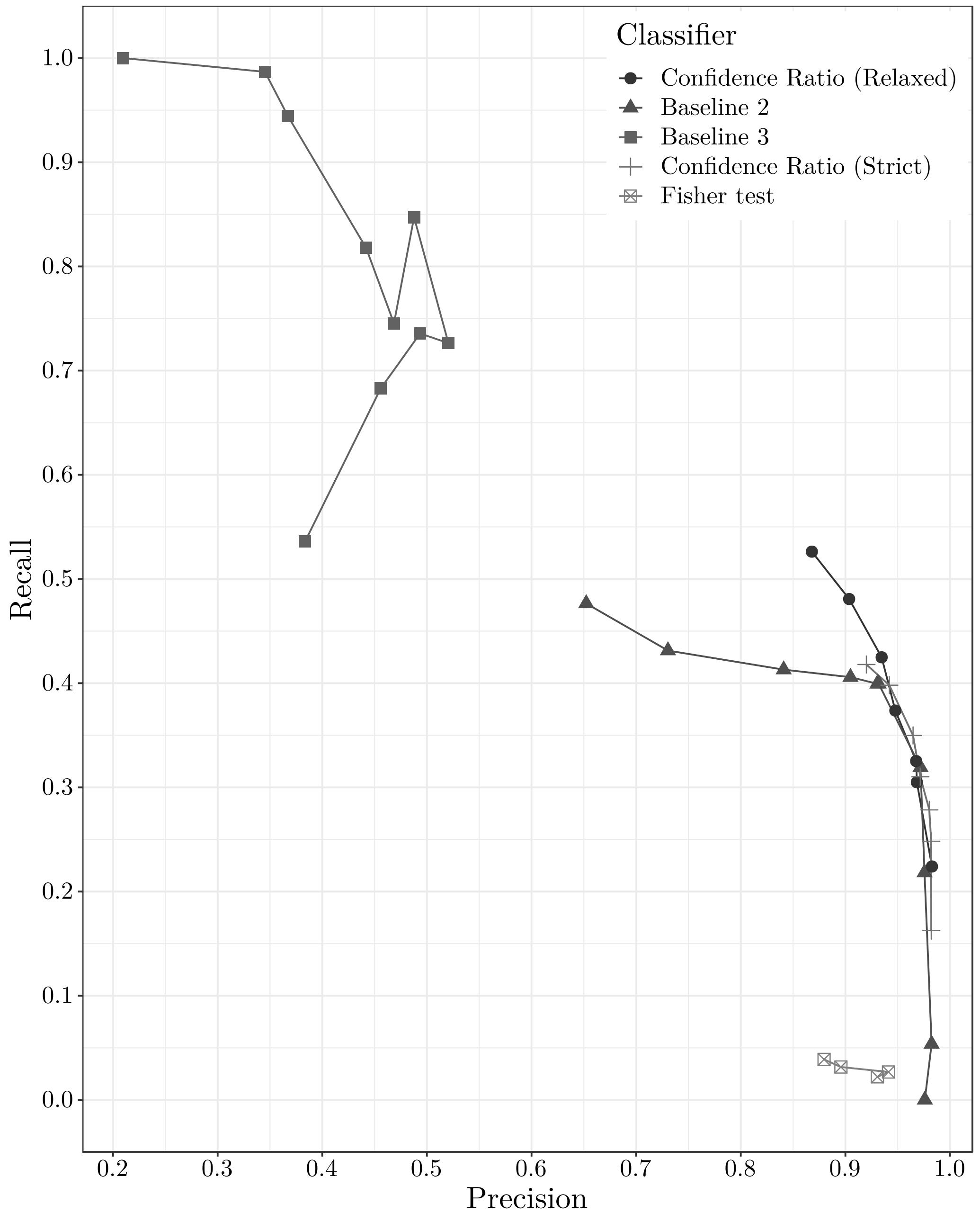

Figure 7.6 – Stress test: Precision and Recall on Wikidata

Chapter 8

Conclusion

8.1 Summary

In this thesis, we introduced novel approaches and optimizations designed to speed up the process of rule mining on large Knowledge Bases. We developed and presented two algorithms to demonstrate the efficiency of those optimizations on real datasets: the AMIE 3 algorithm (the successor of the exact rule mining algorithm AMIE+) and the Pathfinder algorithm, a novel algorithm specialized in mining path rules. These two algorithms are *exhaustive* with regard to the parameters provided by the user, compute the quality measures of each rule *exactly* and *scale* to large KB and longer rules.

Rule Mining. In Chapter 2 we presented the task of rule mining as an Inductive Logic Programming problem and described multiple rule mining algorithms that apply on Knowledge Bases. Among them was the exact rule mining algorithm AMIE+, which use a pruning heuristic to speed up the processing of large KBs at the cost of the completeness of the result.

In Chapter 3, we improved the method that AMIE uses to compute the quality metrics of the rules. We introduced the lazy strategy, some query optimizations and more efficient data structures. These optimizations make the pruning heuristic obsolete and make AMIE 3 exhaustive once again. In our experiments, AMIE 3 is around 50× faster than its predecessor AMIE+ on some datasets (Yago2s or DBPedia 3.8) and scales to large KBs that were previously beyond reach.

Pruning the search space. In Chapter 4, we decomposed the rules into star patterns and outlined that every star pattern of a rule must be satisfiable for the rule to be satisfiable. We discussed how to extract all satisfiable star patterns of a KB and introduced the compatibility condition that decides if a set of star patterns can be combined into a rule. However, we also showed that the combinatorial problem of enumerating every compatible set of star patterns can be seen as a zero sum subset problem, which is NP-complete.

Path Rule Mining. In Chapters 5 and 6, we aimed to mine every closed path rule from a KB. We introduced two increasingly refined algorithms to solve this problem: the Pathfinder Vanilla algorithm and the (Full) Pathfinder algorithm.

In Chapter 5, we showed that the shape of the path rules and of the of star patterns (introduced in Chapter 4) allows us to design a refinement operator for both algorithms that does not produce any duplicate rule and that efficiently prunes unsatisfiable rules early.

In Chapter 6, we discussed how data can be reused between a rule and its refinements to speed up the computation of the quality measures or avoid unnecessary computations. In particular, the Pathfinder algorithm maintains the set of possible values of the variables of a rule and the bounds on the iterated provenance[1] between a rule and its refinements. In our experiments, we showed that the overhead of computing the additional information is usually compensated by the efficient pruning of the propagated bounds and the improved computation of the quality measures, making the Pathfinder algorithm faster than AMIE on most datasets. This is particularly noticeable when mining longer rules: the Pathfinder algorithm is capable of mining every rule of length 4 within one hour on datasets where this task was previously intractable, and it can scale even further and mine even longer rules at incredible speed.

Mining rules about the real world. In Chapter 7, we presented a novel approach to mine rules about the real world from an incomplete KB. In particular, we addressed the problem of mining automatically which attribute should be obligatory for a class, such as “every singer should sing”. We introduced a statistical model of incompleteness, the random sampling model and presented two signals in this model: the “spread” and the “amplification”. We translated these signals in a new metric, the confidence ratio, used to discriminate if an attribute is obligatory in a class or not. In our experiments, the confidence ratio is capable of identifying obligatory attributes with a precision of up to 90% in Yago and Wikidata.

[1]or the iterated provenance itself depending on the strategy used

8.2 Outlook

We describe here an outlook of broader directions in continuation of this thesis.

Query optimization. Most of the optimizations on the computation of the quality measures we proposed in this thesis rely on some particularities of the rule mining process (laziness, data reusage between multiple rules). However, it would be interesting to study how the optimizations and subtleties introduced in this thesis can carry over to other databases, other systems or more generally to the field of query optimization.

Rule and Oracle. In Section 3.2.3, we described a limitation of the PCA Confidence when it comes to make prediction with a rule having a functional head relation. For example, the rule:

$$hasChild(z,x) \wedge politicianOf(z,y) \Rightarrow politicianOf(x,y)$$

would predict that every child of a politician is a politician. A solution of this problem would be to make a prediction only when we know that the child of a politician is also politician. We model this using an oracle capable of deciding whether any child x should have the attribute *politicianOf* or not, i.e an oracle deciding for every entity E of the KB if this entity E has the attribute *politicianOf* in the real world or not.

Using this oracle, we would be able to use the rules to make predictions only on incomplete portions of the KB and as such this approach solves the problem of the PCA confidence. However, the problem is now on how to design such an oracle. In an unpublished experiment, we built an oracle using the obligatory attributes mined in Chapter 7, but the results where inconclusive as the the obligatory condition is too restrictive for the task.

The completeness information mined in [29, 81, 54, 69] or a model based on KB embeddings could be used to built such an oracle. Note that the idea of using completeness information to reduce the scope of the predictions is already used in [29] and introduced in [81]. However, it has never been properly formalized, as we propose here, using two distinct objects: a rule and an oracle. We believe that this representation is clearer and more flexible.

Comparative Rule Mining. In order to mine obligatory attributes in a KB, we used in Chapter 7 a comparative approach that can be described as: "what characterizes best people who sing: being a singer or being a guitarist?". This comparative analysis allows to discriminate mere correlations (the guitarist who sings because he is also a singer) from interesting rules. We investigated, during this thesis, how to push this analysis further, using more complex tools such as decision trees. This approach was not conclusive as the more selective the approach was, the harder it became to distinguish the features from the noise.

An approach to overcome this obstacle would be to construct some sort of "null model" of the KB to compare to. In other words, to generate a random KB that shares some statistical properties with the original KB. These statistical properties are yet to be defined, but we believe that the study of KB sampling techniques might provide some directions.

8.3 Conclusion

The *Semantic Web* has quickly become a constellation of large and interconnected entity-centric Knowledge Bases. These KBs contain domain-specific knowledge that can be used for multiple applications such as question answering or automatic reasoning. But in order to take full advantage of this data, it is essential to understand the schema and the patterns of the KB. A simple and expressive manner to describe the dependencies in a KB is to use rules. Thus it is crucial to be able to perform rule mining at scale.

With this thesis, we have made a step towards this objective. We proposed novel approaches to improve rule mining and new algorithms that efficiently process large KBs without resorting to sampling or approximations. With this, we provide tools for data scientists to understand, exploit and complete this interconnected knowledge. We hope this work will help analyze complex and momentous datasets, such as KG-Covid19, carry over to connected fields such as query optimization, and give rise to new prospects in the study of Knowledge Bases and their completion.

Appendix A

Computation of Support and Confidence

Notation. Given a logical formula ϕ with some free variables $x_1, \ldots, x_n$, all other variables being by default *existentially* quantified, we define:

$$\#(x_1, \ldots, x_n) : \phi \quad := \quad |\{(x_1, \ldots, x_n) \; : \; \phi(x_1, \ldots, x_n) \text{ is } \textit{true}\}|$$

We remind the reader of the two following definitions:

Definition 2.3.4 (Prediction of a rule)**.** The predictions P of a rule $\vec{B} \Rightarrow h$ in a KB $\mathcal{K}$ are the head atoms of all instantiations of the rule where the body atoms appear in $\mathcal{K}$. We write $\mathcal{K} \wedge (\vec{B} \Rightarrow h) \models P$.

Definition 2.3.9 (Support)**.** The support of a rule in a KB is the number of positive examples predicted by the rule.

A prediction of a rule is a positive example if and only if it is in the KB. This observation gives rise to the following property:

Proposition A.1 (Support in practice)**.** The support of a rule $\vec{B} \Rightarrow h$ is the number of instantiations of the head variables that satisfy the query $\vec{B} \wedge h$. This value can be written as:

$$\text{support}(\vec{B} \Rightarrow h(x, y)) = \#(x, y) : \vec{B} \wedge h(x, y)$$

Definition 2.3.10 (Confidence)**.** The confidence of a rule is the number of positive examples predicted by the rule (the support of the rule), divided by the number of examples predicted by the rule.

Under the CWA, all the predicted examples are either positive examples or negative examples. Thus, the standard confidence of a rule is the support of the rule divided by the number of prediction of the rule, written:

$$\textit{std-conf}(\vec{B} \Rightarrow h(x,y)) = \frac{\#(x,y) : \vec{B} \wedge h(x,y)}{\#(x,y) : \vec{B}} \tag{A.1}$$

Assume h is more functional than inverse functional. Under the PCA, a predicted negative example is a prediction $h(x,y)$ that is not in the KB, such that, for this x there exists another entity y' such that $h(x,y')$ is in the KB. When we add the predicted positive examples, the denominator of the PCA confidence becomes:

$$\#(x,y) : (\vec{B} \wedge h(x,y)) \vee (\vec{B} \wedge \neg h(x,y) \wedge \exists y'.h(x,y'))$$

We can simplify this logical formula to deduce the following formula for computing the PCA confidence:

$$\textit{pca-conf}(\vec{B} \Rightarrow h(x,y)) = \frac{\#(x,y) : \vec{B} \wedge h(x,y)}{\#(x,y) : \vec{B} \wedge \exists y'.h(x,y')} \tag{A.2}$$

Appendix B

Number of items per relation

Table B.1 – Maximal multiplicity of relations in Yago2

Relation	$Star_1(\mathcal{K})$	$Star_{10}(\mathcal{K})$	$Star_{50}(\mathcal{K})$
actedIn	246	116	62
$actedIn^{-1}$	22	18	14
created	325	196	101
$created^{-1}$	22	14	9
dealsWith	12	8	6
$dealsWith^{-1}$	67	19	2
diedIn	1	1	1
$diedIn^{-1}$	951	227	59
graduatedFrom	1	1	1
$graduatedFrom^{-1}$	372	110	42
hasAcademicAdvisor	2	2	2
$hasAcademicAdvisor^{-1}$	15	10	5
hasMusicalRole	11	9	6
$hasMusicalRole^{-1}$	2 646	251	12
isAffiliatedTo	3	1	0
$isAffiliatedTo^{-1}$	6	1	0
isLocatedIn	2	2	2
$isLocatedIn^{-1}$	8 652	1 074	476
isMarriedTo	7	5	3
$isMarriedTo^{-1}$	9	5	3
livesIn	7	5	4
$livesIn^{-1}$	297	135	40
...	...	...	...
Number of multiplicity items	23 969	4 410	1 388

Table B.2 – Number of instantiated items per relation in Yago2

Relation	$Star_1(\mathcal{K})$	$Star_{10}(\mathcal{K})$	$Star_{50}(\mathcal{K})$
actedIn	33 844	530	0
$actedIn^{-1}$	22 984	2 720	94
created	156 244	44	0
$created^{-1}$	46 156	4 757	283
dealsWith	107	17	1
$dealsWith^{-1}$	122	4	0
diedIn	3 234	366	66
$diedIn^{-1}$	22 156	0	0
graduatedFrom	1 717	274	36
$graduatedFrom^{-1}$	11 875	0	0
hasAcademicAdvisor	1 118	10	0
$hasAcademicAdvisor^{-1}$	1 881	0	0
hasMusicalRole	151	53	27
$hasMusicalRole^{-1}$	5 667	5	0
isAffiliatedTo	15	0	0
$isAffiliatedTo^{-1}$	20	0	0
isLocatedIn	46 142	1941	549
$isLocatedIn^{-1}$	211 178	0	0
isMarriedTo	11 004	0	0
$isMarriedTo^{-1}$	10 547	0	0
livesIn	4 409	240	31
$livesIn^{-1}$	10 930	0	0
rdf:type	280 979	92 697	22 208
$rdf{:}type^{-1}$	2 467 341	1 270 526	1 024
...	...	...	...
Number of instantiated items	3 838 590	1 377 867	24 806

Appendix C

Compatibility of a set of star patterns: Algorithm

We recall the compatibility condition:

Definition 4.4.2 (Multiplicity of a signed relation in a star pattern)**.** Given a star pattern σ, the multiplicity of the signed relation p in σ is defined as:

$$mult(p,\sigma) = \begin{cases} 0 & \text{if } \forall n > 0 \ p^n \notin \sigma \\ max(n > 0 : p^n \in \sigma) & \text{otherwise.} \end{cases}$$

Proposition 4.4.3 (Compatibility condition)**.** A multiset of star patterns $L = \{\sigma_1, \dots, \sigma_n\}$ is compatible if and only if for every signed relation p, there is a mapping $M_p : L \to L$ such that:

- $\forall i \in \{1, n\}$, σ_i has exactly $mult(p, \sigma_i)$ images by M_p.
- $\forall i \in \{1, n\}$, σ_i has exactly $mult(p^{-1}, \sigma_i)$ antecedents by M_p.

Such a mapping defines exactly the atoms with the p relation of the generated rule. More precisely, if $(\sigma_i, \sigma_j) \in M_p$ then the atom $p(\rho(\sigma_i), \rho(\sigma_j))$[1] will be in the constructed rule.

Using Proposition 4.4.3, this problem can be seen as finding a directed graph (L, E) given some conditions on the input and output degrees (δ_{in} and δ_{out}) of each vertices as represented in Figure C.1. An equivalent problem is to find a bipartite graph given those restrictions as represented in Figure C.2.

Algorithm 7 is a greedy algorithm that decides the existence of such bipartite graph for a relation p and a list L of star patterns.

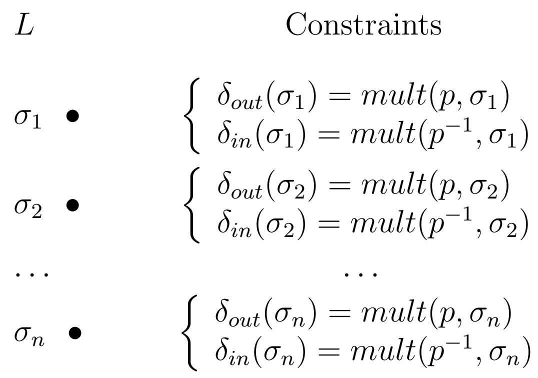
L
Constraints
σ_1
$\delta_{out}(\sigma_1) = mult(p, \sigma_1)$
$\delta_{in}(\sigma_1) = mult(p^{-1}, \sigma_1)$
σ_2
$\delta_{out}(\sigma_2) = mult(p, \sigma_2)$
$\delta_{in}(\sigma_2) = mult(p^{-1}, \sigma_2)$
...
...
σ_n
$\delta_{out}(\sigma_n) = mult(p, \sigma_n)$
$\delta_{in}(\sigma_n) = mult(p^{-1}, \sigma_n)$

Figure C.1 – Deciding compatibility as a graph existence problem

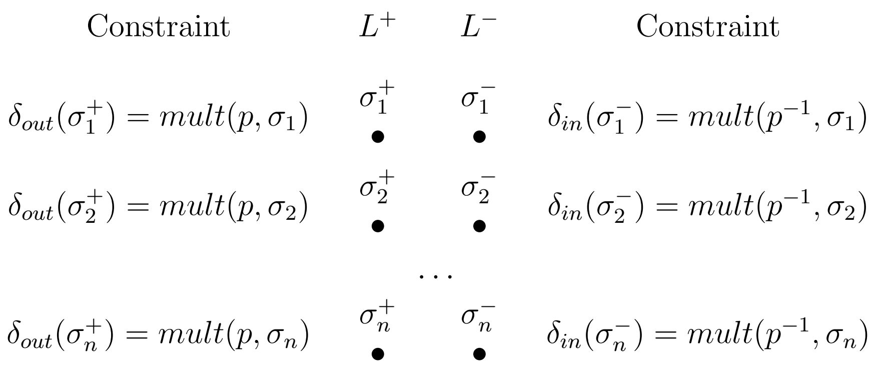
Constraint
L^+
L^-
Constraint
$\delta_{out}(\sigma_1^+) = mult(p, \sigma_1)$
σ_1^+
σ_1^-
$\delta_{in}(\sigma_1^-) = mult(p^{-1}, \sigma_1)$
$\delta_{out}(\sigma_2^+) = mult(p, \sigma_2)$
σ_2^+
σ_2^-
$\delta_{in}(\sigma_2^-) = mult(p^{-1}, \sigma_2)$
...
$\delta_{out}(\sigma_n^+) = mult(p, \sigma_n)$
σ_n^+
σ_n^-
$\delta_{in}(\sigma_n^-) = mult(p^{-1}, \sigma_n)$

Figure C.2 – Deciding compatibility as a bipartite graph existence problem

Correctness and completeness of algorithm 7. Algorithm 7 returns True if and only if L is compatible for the relation p.

Proof. **If:** We must prove that G is a bipartite graph satisfying the constraints on the input and output degrees of each nodes. First we prove the following loop invariant:

Lemma C.1. At the beginning of the loop line 8, the priority p of every node σ_j^- in L^- is:

$$mult(p^{-1}, \sigma_j) - \delta_{in}(\sigma_j^-)$$

δ_{in} being here the incoming degree of the edges in G at the beginning of the loop.

Proof of Lemma C.1. On the first execution of the loop, G is empty ($\delta_{in}(\sigma_j^-) = 0$) and the priority of each node is $mult(p^{-1}, \sigma_j)$.

Suppose the invariant verified at the beginning of the loop, Lines 14 and 15 we add a new edge (σ_i^+, σ_j^-) for each node σ_j^- in t. Every σ_j^- considered in the loop Line 11 are different (as they are pulled from the queue L^-) and the edges added in G are new as we consider a different σ_i^+ every loop iteration (pulled from the queue L^+). Thus, at the end of the loop Lines 11 to 15, t contains all the elements σ_j^- for which we added exactly one new incoming edge in G. At the end of the second loop, the priority of every element in L^- will be:

$$mult(p^{-1}, \sigma_j) - \hat{\delta}_{in}(\sigma_j^-)$$

with:

$$\hat{\delta}_{in}(\sigma_j^-) = \begin{cases} \delta_{in}(\sigma_j^-) + 1 & \text{if one edge towards } \sigma_j^- \text{ was added to } G \\ \delta_{in}(\sigma_j^-) & \text{otherwise} \end{cases}$$

■

Suppose our algorithm returned True. First, every node σ_i^+ in G has exactly $mult(p, \sigma_i)$ outgoing edge as each node is considered exactly once by the loop Line 8 and the loop Line 11 added exactly m (different) outgoing edges of σ_i^+ to G, m being the priority of σ_i^+ initialized to $mult(p, \sigma_i)$ and unchanged afterwards.

Second, the test Line 20 ensures that the maximal priority of the elements in L^- is 0 and the test Line 13 ensures that the priority of the elements in L^- remain non-negative. Thus, before returning True, the priority of every node σ_j^- in L^- is 0 and using the Lemma C.1 we have:

$$\forall j \in \{1, n\},\ \delta_{in}(\sigma_j^-) = mult(p^{-1}, \sigma_j)$$

Only if: We want to prove that our algorithm will answer True if a mapping M_p satisfaying Proposition 4.4.3 exists. In order to prove the correctness of our greedy approach, we need to prove an additional lemma:

Lemma C.2. If a (non-empty) mapping M_p satisfaying Proposition 4.4.3 exists, then there is mapping M'_p satisfaying Proposition 4.4.3 that maps the star pattern with maximal multiplicity in p to the star pattern of maximal multiplicity in p^{-1}.

Proof of Lemma C.2. Let σ_i^+ and σ_j^- be the star patterns with maximal multiplicity in p and p^{-1} respectively. Suppose there is a non-empty mapping M_p satisfaying Proposition 4.4.3 such that $(\sigma_i^+, \sigma_j^-) \notin M_p$. σ_j^- has at least one antecedent σ_k^+ by M_p. As σ_i^+ has the highest multiplicity in p, there is an element $\sigma_{k'}^-$ in the image of σ_i^+ by M_p that is not in the image of σ_k^+ by M_p. Written formally:

$$\begin{cases} |M_p(\sigma_i^+)| \geq |M_p(\sigma_k^+)| \\ (\sigma_i^+, \sigma_j^-) \notin M_p \\ (\sigma_k^+, \sigma_j^-) \in M_p \end{cases} \Rightarrow^2 \exists \sigma_{k'}^- \begin{cases} (\sigma_i^+, \sigma_{k'}^-) \in M_p \\ (\sigma_k^+, \sigma_{k'}^-) \notin M_p \end{cases}$$

Then we pose:

$$M_p' = M_p \setminus \{(\sigma_k^+, \sigma_j^-), (\sigma_i^+, \sigma_{k'}^-)\} \cup \{(\sigma_i^+, \sigma_j^-), (\sigma_k^+, \sigma_{k'}^-)\}$$

M_p' satisfies Proposition 4.4.3 as overall the number of images and antecedents of each star pattern is unchanged compared to M_p and M_p' maps σ_i^+ to σ_j^-. ■

Using the same reasoning, we can find a map M_p' satisfaying Proposition 4.4.3 that maps σ_i^+ with the $mult(p, \sigma)$ elements with maximal multiplicity in p^{-1}. Once we have mapped σ_i^+, we end up with the equivalent problem of finding a mapping $M_p' : L \setminus \{\sigma_i^+\} \rightarrow L$ such that $\forall i \in \{1, n\}$, σ_i has exactly $mult(p^{-1}, \sigma_i) - \delta_{in}(\sigma_i)$ antecedents by M_p'. We can use the same strategy for this sub-problem. □

[2] $|A| \geq |B| \wedge |B \setminus A| > 0 \Rightarrow A \not\subseteq B \Rightarrow |A \setminus B| > 0$

Algorithm 7: isCompatible

```
Input: List $L = \{\sigma_1, \dots, \sigma_n\}$, relation $p$
Output: True iff $L$ is compatible for the relation $p$
$L^+$ = MaxPriorityQueue()
$L^-$ = MaxPriorityQueue()
$G = \langle\rangle$
for $i = 1$ to $n$ do
    if $mult(p, \sigma_i) > n$ or $mult(p^{-1}, \sigma_i) > n$ then return False;
    $L^+$.insert-with-priority($i$, $mult(p, \sigma_i)$)
    $L^-$.insert-with-priority($i$, $mult(p^{-1}, \sigma_i)$)
while $L^+ \neq \emptyset$ do
    $(i, m) = L^+$.pull()
    $t$ = List()
    for $k = 0$ to $m$ do
        $(j, p) = L^-$.pull()
        if $p = 0$ then return False;
        $G$.add($(\sigma_i^+, \sigma_j^-)$)
        // The atom $p(\rho(\sigma_i), \rho(\sigma_j))$ is generated
        $t$.add($(j, p)$)
    for $k = 0$ to $m$ do
        $(j, p) = t$.pop()
        $L^-$.insert-with-priority($j$, $p - 1$)
$(j, p) = L^-$.pull()
if $p = 0$ then
    return True;
else
    return False;
```

Appendix D

Résumé de la thèse en Français

D.1 Position du problème

Lorsqu'on effectue une recherche sur Google ou sur Bing, on obtient une liste de sites webs en réponse. Mais dorénavant, dans certains cas, on obtient une réponse structurée. Par exemple, lorqu'on pose la question “Quand est-ce que Steve Jobs est né ?”, le moteur de recherche nous répond directement le “24 février 1955”. Si on recherche seulement “Steve Jobs”, on obtient une petite biographie, sa date de naissance, de mort, un lien vers son épouse et sa famille. Tout cela est possible parce que le moteur de recherche dispose de données spécifiques sur les personnes les plus connues. Ces données sont notamment stockées dans des Bases de Connaissances.

Les bases de connaissances utilisées par les moteurs de recherche utilisent un modèle entités/relations. Les individus (*Steve Jobs*), ou plus généralement n'importe quel objet réel ou imaginable (les *Etats-Unis*, le *Kilimanjaro*, *Télécom Paris*) sont représentés en tant qu'entités, et les relations entre les différentes entités sont stockées sous la forme de “faits”. Ces faits sont des déclarations sujet-verbe-complément simples (telles que : *SteveJobs founded AppleInc*, *SteveJobs hasWonPrize NationalMedalOfTechnology*, etc.). Les entités sont aussi rangées dans différents catégories appelées “classes sémantiques”. Par exemple on a les relations : *SteveJobs is-a person, SteveJobs is-a entrepreneur*, etc.

Cette représentation des connaissances n'est pas nouvelle. Elle s'inspire des avancées des années 80-90 dans le domaine de l'intelligence artificielle, plus particulièrement, le projet Cyc [47] ou le projet WordNet [24]. À l'époque, ces bases de connaissances étaient crées et maintenues manuellement, mais durant la dernière décennie, de nouvelles bases de connaissances ont pu se développer grâce à l'extraction automatique de données.

Des projets tels que KnowItAll [22], ConceptNet [49], DBPedia [46], NELL [12], BabelNet [67], Wikidata [84] et YAGO [78] contiennent des connaissances diverses et variées, en libre accès, maintenus automatiquement ou de manière collaborative. Les entreprises privées ont ensuite emboité le pas, avec le développement du Google Knowledge Graph [19] (contenant Freebase [9]), de Satori (Microsoft), Evi (Amazon), LinkedIn's Knowledge Graph, et IBM Watson [26]. Au total, ces bases de connaissances contiennent plusieurs millions d'entités, organisées dans des centaines de milliers de classes, et des centaines de millions de faits reliant ces entités. De plus les données de multiples bases sont reliées entre elles, permettant le croisement d'informations, et forment ainsi ce qu'on appelle le Web Sémantique [8].

Ces immenses rassemblements de connaissances inter-connectés peuvent alors être eux-même analysés. Les motifs récurrents et corrélations peuvent être représentées sous la forme de règles logiques simples. Par exemple, la règle :

$$married(x, y) \wedge livesIn(x, z) \Rightarrow livesIn(y, z)$$

se lisant “Si X et Y sont mariés, et que X vit dans un lieu Z, alors Y vit aussi dans Z. En d'autres termes, cette règle exprime le fait que deux personnes mariées vivent généralement au même endroit. À chaque règle est associé un indice de confiance, un score, permettant de jauger la validité de cette règle.

Les bases de connaissances peuvent donc être analysées pour trouver automatiquement l'ensemble des règles logiques s'appliquant à cette base. C'est ce qu'on appelle l'extraction de règles. Plus précisément, les algorithmes d'extraction de règles veulent extraire les règles de meilleure qualité, c'est-à-dire les règles ayant un indice de confiance élevé, ou supérieur à une valeur définie par l'utilisateur.

Les règles trouvées ont ensuite plusieurs utilités : Premièrement, elles peuvent permettrent de compléter les données manquantes d'une base de connaissances. Par exemple, si on ne connaît le lieu de résidence d'une personne, le lieu de résidence de son époux peut être une proposition intéressante. Deuxièmement, elles peuvent permettre d'identifier des données erronées ou incomplètes. Si deux époux vivent dans différentes villes, cela peut indiquer un problème. Enfin, ces règles peuvent être utiles pour d'autres tâches sous-jacentes, telles que la vérification de faits [2], l'alignement d'ontologies [28] ou pour déterminer si les données sont complètes ou non [29].

Néanmoins, la difficulté du problème d'extraction de règle réside dans la taille exponentielle de l'espace de recherche : chaque relation peut être combinée avec n'importe quelle relation pour former une règle, de qualité variable. C'est pourquoi les premières approches (par exemple AMIE [31]) étaient incapables d'analyser de grandes bases de connaissances telle que Wikidata en moins d'une journée. Depuis, les bases de connaissances ont continuées à s'agrandir, et les approches d'extraction de règles se sont multipliées. Certaines approches utilisent l'échantillonnage ou diverses approximations pour estimer la qualité d'une règle [30, 94, 63, 14]. Plus une approche approxime, plus elle est rapide, mais moins précis sont ses résultats. Une autre stratégie couramment employée [63, 58, 94, 66] est de limiter la recherche à un sous-ensemble de règles qui réussisent à prédire un ensemble de faits donné. Cela permet aussi d'accélérer la recherche, mais ne permet pas de trouver toutes les règles ayant un indice de confiance élevé de notre base de connaissances.

D.2 Contribution

L'objectif pricipal de cette thèse a été d'étudier et de développer de nouvelles approches permettant d'extraire efficacement les règles de qualité depuis d'énormes bases de connaissances. En particulier, nous avons œuvré à concevoir un algorithme d'extraction de règles qui calcule de manière *exacte* la qualité d'une règle et qui soit *exhaustif*, c'est-à-dire qui soit capable d'extraire toutes les règles de qualité de nos données. Un tel algorithme, générant un ensemble complet et exact de règles, deviendrait alors un référentiel pour d'autres algorithmes d'extraction de règles.

Préliminaires. Dans le chapitre 2 de cette thèse, nous définissons la notion de base de connaissances et expliquons leur principales caractéristiques. Ensuite, nous introduisons formellement le problème d'extraction de règles et décrivons les différents algorithmes d'extractions s'appliquant aux bases de connaissances. Ce chapitre se base notre article de présentation ("tutorial paper") :

- Fabian M Suchanek, Jonathan Lajus, Armand Boschin, and Gerhard Weikum. Knowledge representation and rule mining in entity-centric knowledge bases. In *Reasoning Web. Explainable Artificial Intelligence*, pages 110–152. Springer, 2019

AMIE 3. Dans le chapitre 3, nous présentons AMIE 3, une version améliorée de AMIE+ [30], un algorithme d'extraction de règles. Dans cette version, nous introduisons de multiples améliorations dans la façon de calculer la qualité d'une règle, permettant à AMIE de redevenir un algorithme *exact* et *exhaustif* d'extractions de règles. Cette nouvelle version est aussi plus rapide et peut maintenant analyser des bases de connaissances trente fois plus grande. Ce chapitre est basé sur notre article de recherche :

- Jonathan Lajus, Luis Galárraga, and Fabian Suchanek. Fast and exact rule mining with AMIE 3. In *European Semantic Web Conference*, pages 36–52. Springer, 2020

Réduire l'espace de recherche. La taille exponentielle de l'espace de recherche demeure un problème lorsque nous voulons utiliser AMIE pour extraire des règles plus complexes, telle que des règles possédant 4 atomes. Dans le chapitre 4, nous étudions comment la décomposition de règles complexes en motifs simples pourrait nous permettre d'identifier et d'écarter des règles impossibles sans avoir à effectuer de longs calculs.

Extraction de règles chaînées. Dans le chapitre 5, nous nous concentrons sur l'extraction de règles d'une forme particulière, les règles chaînées. Nous montrons que ces règles peuvent être générées de manière unique via une méthode particulière et que les motifs simples introduits au chapitre 4 peuvent être utilisés pour efficacement réduire l'espace de recherche des règles chaînées. Á partir de ses résultats, nous construisons l'algorithme "Pathfinder Vanilla", un algorithme, exact et exhaustif, d'extraction de règles chaînées.

Cependant, ce nouvel algorithme calcule la qualité de chaque règle indépendamment et la plupart des calculs sont donc redondants. Mais dans un algorithme d'extraction de règles, chaque règle est dérivée d'une règle parente et les données nécessaires au calcul de la qualité de la règle parente peuvent être réutilisées pour le calcul d'une règle dérivée. Dans le chapitre 6, nous utilisons cette stratégie pour accélérer le calcul des mesures de qualité de toutes nos règles, ainsi que pour déterminer statistiquement des bornes exactes permettant d'identifier et d'écarter précocément de mauvaises règles. Enfin, nous présentons l'algorithme "Pathfinder", une version améliorée de l'algorithme Pathfinder Vanilla, qui extrait les règles chaînées d'une base de connaissances de manière exacte et exhaustive, plus efficacement que AMIE sur la plupart des jeux de données. Notre algorithme est d'autant plus efficace que les règles à extraire sont longues. Ces chapitres sont des travaux non publiés.

Au-delà des données... Les bases de connaissances représentent nos connaissances du monde réel mais sont souvent incomplètes. Cela entraîne des biais conséquents lors de l'extraction de règles : il est difficile de savoir si une règle extraite est tout le temps vraie ou s'il s'agit d'une simple corrélation. Néanmoins, ces biais peuvent être analysés statistiquement. Dans le chapitre 7, nous introduisons un modèle statistique, et un algorithme correspondant, permettant de déterminer les relations obligatoires d'une classe.

Par exemple, notre algorithme est capable de déterminer que tous les chanteurs doivent chanter, à partir de nos données, même si celles-ci sont très incomplètes. Ce chapitre est basé sur notre article de recherche :

- Jonathan Lajus and Fabian M. Suchanek. Are all people married? determining obligatory attributes in knowledge bases. In *WWW*, pages 1115–1124. International World Wide Web Conferences Steering Committee, 2018

D.3 Conclusion

Au fil des ans, le Web Sémantique s'est agrandi pour regrouper une constellation d'énormes bases de connaissances interconnectées. Ces bases répertorient nos connaissances du monde sous la forme de faits structurés et sont utilisées pour la réponse automatique de questions ainsi que pour le raisonnement automatique. Mais pour tirer pleinement avantage de ce vivier d'informations, il est essentiel de comprendre le schéma et les interdépendances intrinsèques à ces données. En particulier, les dépendances fonctionnelles entre les différentes relations peuvent être représentées sous la forme de règles simples. Il est donc crucial de pouvoir extraire ces règles efficacement à partir de nos données.

Dans cette thèse, nous avons fait un pas de plus vers cet objectif. Nous avons proposé des approches novatrices permettant d'améliorer l'extraction de règles et des nouveaux algorithmes qui traitent efficacement de larges quantités de données sans avoir recours à l'échantillonage ou l'approximation. Ainsi, nous fournissons aux analystes des outils pour comprendre, exploiter et compléter cette représentation structurée de nos connaissances actuelles. Nous espérons que ces travaux aideront à l'analyse de jeux de données complexes et actuels (tel que KG-Covid19, répertoriant nos connaissances liées à l'épidémie de Covid-19), qu'ils permettront des avancées dans des domaines de recherche voisins (tel que l'optimisation de requêtes), et ouvriront des perspectives nouvelles dans l'étude des bases de connaissances et de leur complétion.

Bibliography

[1] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *VLDB*, volume 1215, 1994.

[2] Naser Ahmadi, Joohyung Lee, Paolo Papotti, and Mohammed Saeed. Explainable fact checking with probabilistic answer set programming. In *Conference for Truth and Trust online*, 2019.

[3] Naser Ahmadi, Thi-Thuy-Duyen Truong, Le-Hong-Mai Dao, Stefano Ortona, and Paolo Papotti. Rulehub: A public corpus of rules for knowledge graphs. *Journal of Data and Information Quality (JDIQ)*, 12(4):1–22, 2020.

[4] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2), 1972.

[5] Farahnaz Akrami, Mohammed Samiul Saeef, Qingheng Zhang, Wei Hu, and Chengkai Li. Realistic re-evaluation of knowledge graph completion methods: An experimental study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1995–2010, 2020.

[6] Mehwish Alam, Aleksey Buzmakov, Victor Codocedo, and Amedeo Napoli. Mining definitions from RDF annotations using formal concept analysis. In *IJCAI*, 2015.

[7] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook*. Springer, 2003.

[8] Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. Linked data on the Web. In *WWW*, 2008.

[9] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*. ACM, 2008.

[10] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, pages 2787–2795, 2013.

[11] Christian Borgelt. An implementation of the FP-growth algorithm. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, pages 1–5, 2005.

[12] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R Hruschka, and T. M Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.

[13] Yang Chen, Sean Goldberg, Daisy Zhe Wang, and Soumitra Siddharth Johri. Ontological Pathfinding. In *SIGMOD*, 2016.

[14] Yang Chen, Daisy Zhe Wang, and Sean Goldberg. ScaLeKB: Scalable learning and inference over large knowledge bases. *VLDB Journal*, 25(6), 2016.

[15] F. Darari, S. Razniewski, R. Prasojo, and W. Nutt. Enabling fine-grained RDF data completeness assessment. In *ICWE*, 2016.

[16] Luc De Raedt and Kristian Kersting. Probabilistic inductive logic programming. In *Probabilistic Inductive Logic Programming*. Springer, 2008.

[17] Luc Dehaspe and Luc De Raedt. Mining association rules in multiple relations. In *International Conference on Inductive Logic Programming*, 1997.

[18] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[19] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *SIGKDD*, 2014.

[20] Takuma Ebisu and Ryutaro Ichise. Graph pattern entity ranking model for knowledge graph completion. In *NAACL-HLT*, 2019.

[21] F. Erxleben, M. Günther, M. Krötzsch, J. Mendez, and D. Vrandecic. Introducing Wikidata to the linked data web. In *ISWC*, 2014.

[22] Oren Etzioni, Michael Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Web-scale information extraction in knowitall: (preliminary results). In *WWW*, 2004.

[23] Michael Färber, Frederic Bartscherer, Carsten Menne, and Achim Rettinger. Linked data quality of DBpedia, Freebase, Opencyc, Wikidata, and Yago. *Semantic Web*, 2016.

[24] C. Fellbaum, editor. *WordNet: An Electronic Lexical Database.* MIT Press, 1998.

[25] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary RDF Representation (HDT). *Web Semantics*, 19, 2013.

[26] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A Kalyanpur, Adam Lally, J William Murdock, Eric Nyberg, John Prager, et al. Building Watson: An overview of the DeepQA project. *AI magazine*, 31(3), 2010.

[27] R. A. Fisher. On the interpretation of chi square from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, 85(1), Jan 1922.

[28] Luis Galárraga, Nicoleta Preda, and Fabian M Suchanek. Mining rules to align knowledge bases. In *AKBC*, 2013.

[29] Luis Galárraga, Simon Razniewski, Antoine Amarilli, and Fabian M Suchanek. Predicting completeness in knowledge bases. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 375–383, 2017.

[30] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. Fast rule mining in ontological knowledge bases with AMIE+. In *VLDBJ*, 2015.

[31] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. AMIE: Association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*, 2013.

[32] Lise Getoor and Christopher P Diehl. Link mining: a survey. *ACM SIGKDD Explorations Newsletter*, 7(2), 2005.

[33] François Goasdoué, Paweł Guzewicz, and Ioana Manolescu. Incremental structural summarization of RDF graphs. 2019.

[34] Bart Goethals and Jan Van den Bussche. Relational Association Rules: Getting WARMER. In *Pattern Detection and Discovery*, volume 2447. Springer Berlin / Heidelberg, 2002.

[35] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2):1–12, 2000.

[36] James Hawthorne. Inductive logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2018.

[37] Lars Heling and Maribel Acosta. Estimating characteristic sets for RDF dataset profiles based on sampling. In *European Semantic Web Conference*, pages 157–175. Springer, 2020.

[38] Sebastian Hellmann, Jens Lehmann, and Sören Auer. Learning of OWL class descriptions on very large knowledge bases. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):25–48, 2009.

[39] Leah Henderson. The problem of induction. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2019.

[40] Katsumi Inoue. Induction as consequence finding. *Machine Learning*, 55(2), 2004.

[41] Tim Kimber, Krysia Broda, and Alessandra Russo. Induction on failure: Learning connected horn theories. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 2009.

[42] Holger Knublauch and Dimitris Kontokostas. Shapes constraint language (SHACL). W3C recommendation, W3C, July 2017.

[43] Jonathan Lajus, Luis Galárraga, and Fabian Suchanek. Fast and exact rule mining with AMIE 3. In *European Semantic Web Conference*, pages 36–52. Springer, 2020.

[44] Jonathan Lajus and Fabian M. Suchanek. Are all people married? determining obligatory attributes in knowledge bases. In *WWW*, pages 1115–1124. International World Wide Web Conferences Steering Committee, 2018.

[45] Ni Lao, Tom Mitchell, and William W Cohen. Random walk inference and learning in a large scale knowledge base. In *EMNLP*, 2011.

[46] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web Journal*, 6(2), 2015.

[47] Douglas B Lenat and Ramanathan V Guha. *Building large knowledge-based systems; representation and inference in the Cyc project.* Addison-Wesley Longman Publishing Co., Inc., 1989.

[48] A. Y. Levy. Obtaining complete answers from incomplete databases. In *VLDB*, 1996.

[49] Hugo Liu and Push Singh. ConceptNet—a practical commonsense reasoning tool-kit. *BT technology journal*, 22(4):211–226, 2004.

[50] Farzaneh Mahdisoltani, Joanna Asia Biega, and Fabian M. Suchanek. Yago3: A knowledge base from multilingual Wikipedias. In *CIDR*, 2015.

[51] Eric Margolis and Stephen Laurence. Concepts. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Stanford, 2014.

[52] Christian Meilicke, Manuel Fink, Yanjie Wang, Daniel Ruffinelli, Rainer Gemulla, and Heiner Stuckenschmidt. Fine-grained evaluation of rule- and embedding-based systems for knowledge graph completion. In *International Semantic Web Conference*, pages 3–20. Springer, 2018.

[53] Marios Meimaris, George Papastefanatos, Nikos Mamoulis, and Ioannis Anagnostopoulos. Extended characteristic sets: graph indexing for SPARQL query optimization. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 497–508. IEEE, 2017.

[54] Paramita Mirza, Simon Razniewski, Fariz Darari, and Gerhard Weikum. Enriching knowledge bases with counting quantifiers. In *International Semantic Web Conference*, pages 179–197. Springer, 2018.

[55] Gabriela Montoya, Hala Skaf-Molli, and Katja Hose. The Odyssey approach for optimizing federated SPARQL queries. In *International Semantic Web Conference*, pages 471–489. Springer, 2017.

[56] A. Motro. Integrity = Validity + Completeness. *TODS*, 1989.

[57] Stephen Muggleton. Inverse entailment and Progol. *New generation computing*, 13(3-4), 1995.

[58] Stephen Muggleton. Learning from positive data. In *ILP*, 1997.

[59] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19, 1994.

[60] Stephen Muggleton and Cao Feng. *Efficient induction of logic programs.* 1990.

[61] Ndapandula Nakashole, Gerhard Weikum, and Fabian M. Suchanek. PATTY: A taxonomy of relational patterns with semantic types. In *EMNLP*, 2012.

[62] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*, pages 984–994. IEEE, 2011.

[63] Stefano Ortona, Venkata Vamsikrishna Meduri, and Paolo Papotti. Robust Discovery of Positive and Negative Rules in Knowledge Bases. In *ICDE*, 2018.

[64] Heiko Paulheim and Christian Bizer. Type inference on noisy RDF data. In *ISWC*, 2013.

[65] Gordon Plotkin. Automatic methods of inductive inference. 1972.

[66] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3), Aug 1990.

[67] S. Ponzetto R. Navigli. BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence*, 193, 2012.

[68] Oliver Ray, Krysia Broda, and Alessandra Russo. Hybrid abductive inductive learning: A generalisation of Progol. In *International Conference on Inductive Logic Programming.* Springer, 2003.

[69] Simon Razniewski, Nitisha Jain, Paramita Mirza, and Gerhard Weikum. Coverage of information extraction from sentences and paragraphs. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5775–5780, 2019.

[70] Simon Razniewski, Flip Korn, Werner Nutt, and Divesh Srivastava. Identifying the extent of completeness of query answers over partially complete databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 561–576, 2015.

[71] Simon Razniewski, Fabian M. Suchanek, and Werner Nutt. But what do we actually know? In *AKBC workshop*, 2016.

[72] Bertrand Russell. *The Problems of Philosophy.* Barnes & Noble, 1912.

[73] S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach.* Prentice Hall, 2002.

[74] Ehud Y Shapiro. *Inductive inference of theories from facts.* Yale University, Department of Computer Science, 1981.

[75] J. F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* Brooks/Cole, 2000.

[76] Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies.* International Handbooks on Information Systems. Springer, 2004.

[77] F. M. Suchanek, D. Gross-Amblard, and S. Abiteboul. Watermarking for Ontologies. In *ISWC*, 2011.

[78] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[79] Fabian M Suchanek, Serge Abiteboul, and Pierre Senellart. PARIS: Probabilistic alignment of relations, instances, and schema. *Proceedings of the VLDB Endowment*, 5(3), 2011.

[80] Fabian M Suchanek, Jonathan Lajus, Armand Boschin, and Gerhard Weikum. Knowledge representation and rule mining in entity-centric knowledge bases. In *Reasoning Web. Explainable Artificial Intelligence*, pages 110–152. Springer, 2019.

[81] Thomas Pellissier Tanon, Daria Stepanova, Simon Razniewski, Paramita Mirza, and Gerhard Weikum. Completeness-aware rule learning from knowledge graphs. In *International Semantic Web Conference*, pages 507–525. Springer, 2017.

[82] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. International Conference on Machine Learning (ICML), 2016.

[83] Johanna Völker and Mathias Niepert. Statistical schema induction. In *ESWC*, 2011.

[84] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10), 2014.

[85] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, 2017.

[86] Alfred North Whitehead and Bertrand Russell. *Principia mathematica.* 1913.

[87] Word Wide Web Consortium. RDF Primer, 2004.

[88] Word Wide Web Consortium. RDF Vocabulary Description Language 1.0: RDF Schema, 2004.

[89] Word Wide Web Consortium. SKOS Simple Knowledge Organization System, 2009.

[90] Word Wide Web Consortium. OWL 2 Web Ontology Language, 2012.

[91] Word Wide Web Consortium. SPARQL 1.1 Query Language, 2013.

[92] Akihiro Yamamoto. Hypothesis finding based on upward refinement of residue hypotheses. *Theoretical Computer Science*, 298(1), 2003.

[93] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. 2015.

[94] Qiang Zeng, Jignesh M. Patel, and David Page. QuickFOIL: Scalable Inductive Logic Programming. *VLDB*, 8(3), November 2014.

[95] Kaja Zupanc and Jesse Davis. Estimating rule quality for knowledge base completion with the relationship between coverage assumption. In *WWW*, 2018.

ECOLE DOCTORALE

Titre : Extraction de Règles Rapide, Exact et Exhaustif dans de Larges Bases de Connaissances

Mots clés : Bases de Connaissances, Web Sémantique, Ontologie, Extraction de Règles

Résumé : Au fil des ans, le Web Sémantique s'est agrandi pour regrouper une constellation d'énormes Bases de Connaissances interconnectées. Ces bases répertorient nos connaissances du monde sous la forme de faits structurés et sont utilisées pour la réponse automatique de questions ainsi que pour le raisonnement automatique. Mais pour tirer pleinement avantage de ce vivier d'informations, il est essentiel de comprendre le schéma et les interdépendances intrinsèques à ces données. En particulier, les dépendances fonctionnelles entre les différentes relations peuvent être représentées sous la forme de règles simples. Il est donc crucial de pouvoir extraire ces règles efficacement à partir de nos données.
Dans cette thèse, on introduit de nouvelles approches et optimisations pour accélérer l'extraction de règles dans de larges Bases de Connaissances. On présente deux nouveaux algorithmes implémentant ces optimisations: AMIE 3 (le successeur de l'algorithme exact AMIE+) et Pathfinder, un nouvel algorithme spécialisé dans l'extraction de règles chaînées. Ces deux algorithmes sont exhaustifs, ils calculent la qualité des règles de manière exacte et passent à l'échelle de manière efficace sur un plus grand volume de données et sur des règles plus complexes.

Title : Fast, Exact, and Exhaustive Rule Mining in Large Knowledge Bases

Keywords : Knowledge Bases, Semantic Web, Ontology, Rule Mining

Abstract : The *Semantic Web* has quickly become a constellation of large and interconnected entity-centric Knowledge Bases. These KBs contain domain-specific knowledge that can be used for multiple application such as question answering or automatic reasoning. But in order to take full advantage of this data, it is essential to understand the schema and the patterns of the KB. A simple and expressive manner to describe the dependencies in a KB is to use rules. Thus it is crucial to be able to perform rule mining at scale.
In this thesis, we introduce novel approaches and optimizations designed to speed up the process of rule mining on large Knowledge Bases. We present two algorithms that implements these optimizations: the AMIE 3 algorithm (the successor of the exact rule mining algorithm AMIE+) and the Pathfinder algorithm, a novel algorithm specialized in mining path rules. These two algorithms are *exhaustive* with regard to the parameters provided by the user, they compute the quality measures of each rule *exactly* and efficiently *scale* to large KB and longer rules.

Institut Polytechnique de Paris
91120 Palaiseau, France