Mining Expressive Rules in Knowledge Graphs

NASER AHMADI and VIET-PHI HUYNH, Eurecom, France
VAMSI MEDURI, Arizona State University, USA
STEFANO ORTONA, Meltwater, UK
PAOLO PAPOTTI, Eurecom, France

We describe RuDiK, an algorithm and a system for mining declarative rules over RDF knowledge graphs (KGs). RuDiK can discover rules expressing both *positive* relationships between KG elements, e.g., "if two persons share at least one parent, they are likely to be siblings," and *negative* patterns identifying data contradictions, e.g., "if two persons are married, one cannot be the child of the other" or "the birth year for a person cannot be bigger than her graduation year." While the first kind of rules identify new facts in the KG, the second kind enables the detection of incorrect triples and the generation of (training) negative examples for learning algorithms. High-quality rules are also critical for any reasoning task involving the KGs.

Our approach increases the *expressive power* of the supported rule language w.r.t. the existing systems. RuDiK discovers rules containing (i) comparisons among literal values and (ii) selection conditions with constants. Richer rules increase the accuracy and the coverage over the facts in the KG for the task at hand. This is achieved with aggressive pruning of the search space and with disk-based algorithms, which enable the execution of the system in commodity machines. Also, RuDiK is robust to errors and missing data in the input graph. It discovers *approximate* rules with a measure of support that is aware of the quality issues. Our experimental evaluation with real-world KGs shows that RuDiK does better than existing solutions in terms of scalability and that it can identify effective rules for different target applications.

CCS Concepts: • **Information systems** → **Data mining**; **Data cleaning**;

Additional Key Words and Phrases: Rule mining, knowledge graphs, graph dependencies

ACM Reference format:
Naser Ahmadi, Viet-Phi Huynh, Vamsi Meduri, Stefano Ortona, and Paolo Papotti. 2020. Mining Expressive Rules in Knowledge Graphs. *J. Data and Information Quality* 12, 2, Article 8 (May 2020), 27 pages.
https://doi.org/10.1145/3371315

1 INTRODUCTION

The creation of RDF knowledge graphs (KGs) is an important activity in modern information systems. KGs organize information in the form of triples with *predicate* expressing a binary relation between a *subject* and an *object*. KG triples, or *facts*, represent information about real-world entities, their properties, and their relationships, such as "Larry Page is the founder of Google." Research has been conducted on KG creation both in the research community (NELL [13], DBPedia [8], Yago

Authors' addresses: N. Ahmadi, V.-P. Huynh, and P. Papotti, Eurecom, France, emails: {Naser.Ahmadi, viet-phi.huynh, papotti}@eurecom.fr; V. Meduri, Arizona State University; email: vmeduri@asu.edu; S. Ortona, Meltwater, UK; email: stefano.ortona@meltwater.com.

1936-1955/2020/05-ART8 $15.00
https://doi.org/10.1145/3371315

[39], Wikidata [43], FreeBase [9], DeepDive [38]) and in the industry (Google [19], Wal-Mart [18]). A major challenge with KGs is the quality of their data. This is because of the processes involved in their creation and update. The structured data in the KGs is usually extracted from multiple sources, possibly from the Web, without human validation. This raises two main issues. The first problem is factual mistakes. Incorrect or outdated data can be transmitted from the sources to the KGs, or noise can originate from the automatic extractors [12, 19]. The second issue is incompleteness. As a graph is rarely complete in practice, *closed world assumption* (CWA) does not hold in KGs [19, 25], i.e., it is not possible to conclude that a missing fact is false. We therefore follow the *open world assumption* (OWA) and consider it as *unknown*. Also, in many cases the KG schema is not fixed, i.e., the set of predicates changes over time and new facts can be inserted without integrity checks.

Because of these issues, the quantity of incompleteness and errors in KGs can be large, with up to 30% errors reported for data extracted from Web sources [1, 40]. KGs can have lots of data, e.g., Wikidata has over a billion facts and millions of entities, therefore it is not feasible to manually verify triples to find mistakes and add missing facts. Tools are needed to assist humans in the KG curation. A natural approach in this direction is to mine *declarative rules*. Once validated, these can be run at scale on the KG to increase the quality of its data [2, 4, 14, 18, 25]. We study the mining and the applications for two kinds of rules: (i) *positive rules*, which can be executed to add new facts to the KG with the aim of increasing its coverage of the reality; (ii) *negative rules*, which can be used to identify logical inconsistencies in the KG, ultimately detecting incorrect triples.

Example 1. Assume a given KG containing people and facts for child and parent connections. A simple positive rule is expressed with the formula:

$$r_1 : \texttt{hasParent}(b, a) \Rightarrow \texttt{hasChild}(a, b),$$

which states that for any person a reported as parent of a person b, there should also be a fact reporting that b is a child of a. If the KG has the fact "Scott Eastwood has a parent named Clint Eastwood," then we can infer the fact that "Clint Eastwood has a child named Scott Eastwood." A negative rule is similar in shape, but expresses different semantics. In the following rule, predicate birthYear stands for "Year of birth" for a person:

$$r_2 : \texttt{birthYear}(a, v_0) \land \texttt{birthYear}(b, v_i) \land v_0 > v_i \land \texttt{hasChild}(a, b) \Rightarrow \bot.$$

The rule states that if person b was born before person a, then it cannot be that b is a child of a. By running the rule on the `hasChild` triples in the KG, it is possible to identify incorrect facts.

While intuitive to humans, the above rules must be manually crafted to be executed over KGs. This process can be difficult, because the domain experts, who know the semantics for the dataset at hand, may miss the CS background to formally express rules. Moreover, the task of writing a set of rules is time-consuming, as there can be thousands of them that are needed to achieve good quality results [41]. In this context, a system to mine rules is important to assist the users in data curation, as well as in any task involving reasoning over the KG [7, 24]. Unfortunately, three main issues make the mining of rules from KGs challenging.

Quality of the Data. Most rule mining algorithms assume that input data have very small amounts of errors [3, 5, 15, 28], but KGs are incomplete and can have high percentages of mistakes.

Open World Assumption. Some methods assume that positive and negative examples are available [17, 32]. However, KGs contain only positive statements, and it is not possible to assume CWA, as there is no obvious solution to derive negative facts acting as counter examples.

Data Volume. Several existing algorithms for rule mining exploit the premise that the input graph can fit entirely in the main central memory [4, 14, 23, 25]. As KGs can have a very large size, existing methods limit the size of the search space by restricting the mining to a simple rule language, which can miss some of the important patterns in the data.

We address the above challenges by introducing declarative rule mining algorithms over noisy and incomplete KGs. We implement such algorithms in RuDiK, a system that exploits disk-based solutions to enable the mining of a larger search space over KGs. The ability to mine with a more expressive language leads to the discovery of rules with comparisons across elements beyond equality and the mining for graph predicates involving literal values (e.g., birthDate). More patterns enable the identification of a bigger number of both errors and new facts with high accuracy. Such improvements are obtained by building our system around four main contributions.

(i) Approximate Rule Mining. We specify the problem of robust rule mining over incomplete and erroneous KGs. The input of the mining is a KG predicate and positive and negative examples for it. In terms of output, in contrast to the long lists of rules ranked on a measure of support [17, 25, 37], in Section 3, we give a definition that identifies a subset of *approximate* rules, i.e., patterns that may not hold over all examples because of missing and incorrect triples in KGs. The solution is then the smallest set of rules covering most of the positive examples and few negative examples.

(ii) Generation of the Examples. The input examples for a predicate strongly affect the quality of the output rules in our approach. While positive examples are available from the graph, manually creating negative examples is an expensive task. Section 4 introduces example generation techniques that are aware of the issues due to inconsistencies and missing data in the KG. Our generated examples enable the discovery of better rules than the examples from alternative techniques.

(iii) A Greedy Algorithm for Rule Mining. We describe a $\log(k)$-approximation algorithm for the rule mining problem, where k is the maximum number of positive examples covered by a rule. The disk-based algorithm, described in Section 5, incrementally materializes the KG in the mining process by traversing only paths that can generate promising rules. The resulting low memory footprint allows the mining with a more expressive rule language.

(iv) Conditional Rules. In many cases, rules are correct when defined for a specific geographical area, in a certain time, or for specific types of entities. To capture these more subtle patterns, in Section 6, we extend the core mining algorithms to discover *conditional rules*, i.e., rules that apply only for a subset of the data, identified by a selection with a constant over the values of an entity or of a type. By exploiting clustering of the entities, we are able to identify subsets of the data for which the mining leads to new rules that are not identified with the general algorithm.

We test our system experimentally on three popular and widely used KGs (DBPedia [8], Yago [39], and Wikidata [43]). Results in Section 7 demonstrate that RuDiK discovers rules of high quality, with a relative increase in average precision by 45% w.r.t. the existing systems. The system can be executed successfully over large KGs on a commodity laptop. We report results showing that negative rules mined by the system generate training examples of high quality comparable to manually crafted examples for learning algorithms. In Section 8, we describe our system and results in the context of the related literature; in Section 9, we conclude with open directions for research in this topic. We open-sourced the code of the system online, together with the experimental results and discovered rules at https://github.com/stefano-ortona/rudik.

This article extends the original RuDiK system [33, 34] with the support for the discovery of rules for object properties, such as *birthDate (person, date)*; the discovery of type conditional rules holding only for a subset of the elements in the KG, such as politicians or artists; and a more extensive experimental evaluation of our solution.

2 BACKGROUND AND DEFINITIONS

We study how to mine declarative rules from RDF KGs. An RDF KG is a database representing information with triples (or *facts*) $\langle s, p, o\rangle$, where a *predicate* p connects a *subject* s and an *object* o. For example, the fact that Hillary Clinton has a child named Chelsea Clinton is expressed with a triple $\langle \mathit{Hillary\_Clinton}, \mathit{hasChild}, \mathit{Chelsea\_Clinton}\rangle$. In a triple, the subject is an *entity*, i.e., a real-world concept; the object is either an entity or a *literal*, i.e., primitive type such as number, date, and string; and the triple predicate specifies a relationship between subject and object.

Several KGs do not limit the allowed instances with a fixed schema, i.e., triples can be inserted for any new predicate. While this operation allows flexibility, it increases the likelihood of introducing mistakes compared to traditional (relational) databases guided by schema. KGs offer constructs to express *TBox* statements defining classes, domain types for predicates, and relationships among classes. These constructs can be used for integrity checks, but most graphs do not come with such information. For this reason, we focus our attention on the triples describing instance data, i.e., *ABox* statements.

2.1 Rule Language

We aim at the automatic discovery of first-order logical formulas in KGs. Our focus is on mining *Horn Rules* with universally quantified variables only. A Horn Rule is a disjunction of *atoms* with at most one unnegated atom. In the implication form, they have the following format:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \Rightarrow B,$$

where $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ is the *body* of the rule (a conjunction of atoms) and B is the *head* of the rule (a single atom). It is possible to express the rule in a logically equivalent form by rewriting the atom in the head of the rule in its negated form in the body to emphasize contradictions:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge \neg B \Rightarrow \bot.$$

We therefore distinguish two kinds of rules. *Positive rules*, or *definite clauses*, determine new candidate triples, such as r_1 in Example 1. *Negative rules*, such as r_2 in Example 1, identify contradicting triples, with a role similar to denial constraints in tabular data [15]. An atom is a predicate connecting two variables, two elements of the graph (entities or literal values), or an element and a variable. We use the notation $\texttt{rel}(a, b)$ to write an atom, where $\texttt{rel}$ is a KG predicate and a, b are either variables or elements. Given a rule r, for the body and the head of the rule, we use r_{body} and r_{head}, respectively, and name the variables in the head of the rule as *target variables*, e.g., a and b in r_1.

Our algorithm is designed to mine rules with one atom in the head. This means that for negative rule discovery, the problem is defined as the mining of rules for an atom in its *negated form* in the head. It can be interpreted as a formula generating false facts. Negative rule r_2 in Example 1 is the result of rewriting the notHasChild atom in the body of the rule:

$$r_2' : \texttt{birthYear}(a, v_0) \wedge \texttt{birthYear}(b, v_i) \wedge v_0 > v_i \Rightarrow \texttt{notHasChild}(a, b).$$

The negative rule contains an example of the *literal comparisons* allowed in the rule language. A literal comparison is expressed as an atom $\texttt{rel}(a, b)$, with $\texttt{rel} \in \{<, \leq, \neq, >, \geq\}$, and a and b that can only be assigned to literal values except if $\texttt{rel}$ is equal to $\neq$, i.e., inequality comparisons for entities are allowed. We may use the form a $\texttt{rel}$ b in some examples for clarity.

Given KG kg and atom $A = \texttt{rel}(a, b)$ where a and b are two KG elements, A *holds* over kg iff $\langle a, \texttt{rel}, b\rangle \in kg$. Given an atom $A = \texttt{rel}(a, b)$ with at least one variable, A can be *instantiated* over kg if there exists at least one element from kg for each variable in A s.t. if we substitute all variables in A with these elements, A holds over kg. Transitively, r_{body} can be instantiated over kg if every atom (with elements) in r_{body} can be instantiated and every literal comparison is logically true.

We do not discover rules with negated atoms in the body. We define a rule *valid* iff every variable in it appears at least twice [14, 25]. This restriction avoids mining rules that we consider not interesting, such as those with unrelated atoms and cross-products, e.g., $\texttt{hasChild}(a, v_0) \wedge \texttt{hasChild}(b, v_i) \Rightarrow \texttt{spouse}(a, b)$.

2.2 Rule Coverage

Given a pair of elements (x, y) from a KG kg and a Horn Rule r, we say that r_{body} *covers* (x, y) if $(x, y) \models r_{body}$. More specifically, given a rule $r : r_{body} \Rightarrow \texttt{r}(a, b)$, r_{body} covers a pair of elements $(x, y) \in kg$ iff we can substitute a with x, b with y, and the rest of the body can be instantiated over kg. Given a set of pair of elements $E = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ and a rule r, we denote by $C_r(E)$ the *coverage* of r_{body} over E as the set of elements in E covered by r_{body}: $C_r(E) = \{(x, y) \in E | (x, y) \models r_{body}\}$.

Given the body r_{body} of a rule r, we denote by r^*_{body} the *unbounded body* of r. The unbounded body of a rule is obtained by keeping only atoms that contain a target variable and substituting such atoms with new atoms where the target variable is paired with a new unique variable. As an example, given $r_{body} = \texttt{rel}_1(a, v_0) \wedge \texttt{rel}_2(v_0, b)$ where a and b are the target variables, $r^*_{body} = \texttt{rel}_1(a, v_i) \wedge \texttt{rel}_2(v_{ii}, b)$. While in r_{body} the target variables are bounded to be connected by variable v_0, in r^*_{body} they are unbounded. Given a set of pair of elements $E = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ and a rule r, we denote by $U_r(E)$ the *unbounded coverage* of r^*_{body} over E as the set of elements in E covered by r^*_{body}: $U_r(E) = \{(x, y) \in E | (x, y) \models r^*_{body}\}$. Given a set E, $C_r(E) \subseteq U_r(E)$.

Example 2. Let E be the set of all possible element pairs in kg. The coverage of r_2 of Example 1 over E ($C_r(E)$) is the set of all pairs of entities $(x, y) \in kg$ s.t. both x and y are the subject for a birth year triple and the year for x is higher. The unbounded coverage of r_2 over E ($U_r(E)$) is the set of all pairs of entities (x, y) s.t. both x and y are the subject for a birth year triple without any condition on the relationships between such year values.

Unbounded coverage plays a crucial role in distinguishing inconsistent and missing data. Given entities (x, y) in a pair, if the birth year is missing for at least one of them, it is not possible to state who was born first. Only if both entities have their birth years *and* x is born before y, we conclude that r_2 does not cover (x, y). Given that KGs are largely incomplete [31], the ability to discriminate missing and conflicting information is of paramount importance. We extend next the definitions for coverage and unbounded coverage to a set of rules $R = \{r_1, r_2, \ldots, r_n\}$ as the union of individual coverages:

$$C_R(E) = \bigcup_{r \in R} C_r(E) \qquad U_R(E) = \bigcup_{r \in R} U_r(E).$$

3 RULE DISCOVERY FOR NOISY KGS

For the sake of simplicity, we define the discovery problem for a given *target predicate.* To obtain all rules for a KG, we compute rules for every predicate in it. We characterize a predicate with two sets of pairs of elements, where an element is either an entity or a literal value. The *generation set* G contains examples for the predicate, while the *validation set* V contains counter examples for the same. Consider the discovery of positive rules for the `hasChild` predicate between entities; G contains true pairs of parents and children and V contains pairs of people who *are not* in such relation. If we want to identify errors (negative rules), then the sets of examples are the same, but they switch roles. To discover negative rules for `hasChild`, G contains pairs of people not in a child relation and V contains pairs of entities in a child relation. As another example, consider predicate `birthDate`, which has a literal value as object. For mining positive rules, G contains true pairs with subject entities and their birth dates and V contains pairs with subjects and literal values that are not their birth dates. Again, for identifying negative rules, we switch the role of the sets.

We formalize next the *exact discovery problem*. In the following definitions, we assume for the sake of simplicity that all possible valid rules and the sets of examples have been already generated; we detail in the rest of the article how they are efficiently obtained from the KG.

Definition 1. Given a KG kg, two sets of pairs of elements G and V from kg with $G \cap V = \emptyset$, and all the valid Horn Rules R for kg, a solution for the *exact discovery problem* is a subset R' of R s.t.:

$$\underset{R'}{\operatorname{argmin}}(size(R')|(C_{R'}(G) = G) \wedge (C_{R'}(V) = \emptyset)).$$

The minimal set of rules covering all pairs in G and none of the pairs in V forms the exact solution. It minimizes the number of rules in the output ($size(R')$) to favor generic rules that can affect large portions of the graph. In fact, given a pair of elements (x, y), there is always an overfitting rule whose body covers only pair (x, y) by assigning target variables to literal values x and y.

Example 3. Assume we are mining positive rules for the predicate `couple` by using examples from the people forming the Obama family. We are given only two couple examples. The positive one is the pair (Barack,Michelle) and the negative one contains their daughters (Natasha,Malia). Consider now the three rules below:

$$r_3 : \texttt{livesIn}(a, v_0) \wedge \texttt{livesIn}(b, v_0) \Rightarrow \texttt{couple}(a, b),$$

$$r_4 : \texttt{hasChild}(a, v_i) \wedge \texttt{hasChild}(b, v_i) \Rightarrow \texttt{couple}(a, b),$$

$$r_5 : \texttt{hasChild}(Michelle, Natasha) \wedge \texttt{hasChild}(Barack, Natasha) \Rightarrow \texttt{couple}(Barack, Michelle).$$

Positive rule r_3 states that two individuals are likely to be a couple if they live in the same location, while rule r_4 states that this is the case when they have at least one child in common. Assuming triples for predicates `livesIn` and `hasChild` are in the KG, both r_3 and r_4 cover the positive example. While r_4 is an exact solution (not covering the negative example), r_3 is not, as the daughters also reside in the same location. Rule r_5 explicitly mentions entity values (constants) in its head and body. It is also an exact solution, but, in contrast to r_3, it only applies for the given positive example.

If any `hasChild` triple is missing in G, then the exact discovery finds only r_5 as a solution. This example demonstrates why the exact discovery is not robust to data quality issues in KGs. Even in cases in which a valid rule exists, missing or incorrect triples associated to the examples in G and V lead to misleading coverage. In the worst case, the exact solution may collapse to a set of rules where every one of them covers only one example in G, effectively discovering rules with no coverage outside of the examples when executed on the graph.

3.1 Weight Function

Being aware of errors and missing data in the graph, we remove the requirement of (not) covering (V) G exactly with the rules. We instead identify rules that hold for most of the data (soft-constraints) to be robust w.r.t. incompleteness and noise. Coverage still guides our assessment of the rule quality: Good rules should cover as many example pairs in G as possible, while covering very few to zero pairs in V. We implement these ideas with a *weight* computed for every rule.

Definition 2. Given a KG kg, two sets of element pairs G and V from kg with $G \cap V = \emptyset$, and a Horn Rule r, the *weight of r* is defined as follows:

$$w(r) = \alpha \cdot \left(1 - \frac{\mid C_r(G) \mid}{\mid G \mid}\right) + \beta \cdot \left(\frac{\mid C_r(V) \mid}{\mid U_r(V) \mid}\right), \tag{1}$$

with $\alpha, \beta \in [0, 1]$ and $\alpha + \beta = 1$, thus $w(r) \in [0, 1]$.

The weight models rule quality w.r.t. G and V—a rule covering all generation elements of G and none of the V validation elements has a weight of 0. Therefore, the better the rule, the smaller its weight. The weight has two parts normalized by parameters α and β. The first part quantifies the coverage over generation set G—the ratio between the coverage of r over G and the size of G. If r covers all elements in G, then this component becomes 0. The second part measures the coverage of r over the unbounded coverage of the rule over V, instead of the total elements in V. Since some elements in V might not satisfy the predicates in r_{body}, we restrict V with unbounded coverage to validate on "qualifying" example pairs that have the information tested by the rule's body.

Parameters α and β set the relevance of each part. A high value for β guides the mining towards rules with high precision by penalizing the ones that cover V pairs, while a high value for α favors the recall by championing rules that cover more G pairs.

Example 4. We use rule r_2 from Example 1. Assume two sets of element pairs G and V from a KG kg. The first part of $w(r_2)$ corresponds to 1 minus the number of examples $(x, y) \in G$ where x is born after y divided by the size of G. The second part is the number of examples $(x, y) \in V$ with x born after y divided by the number of examples $(x, y) \in V$ having birth date values for both x and y in kg, i.e., $U_{r_2}(V)$ does not contain pairs without birth date values.

Definition 3. Given a KG kg, two sets of element pairs G and V from kg with $G \cap V = \emptyset$, and a set of rules R, the *weight for R* is:

$$w(R) = \alpha \cdot \left(1 - \frac{\mid C_R(G) \mid}{\mid G \mid}\right) + \beta \cdot \left(\frac{\mid C_R(V) \mid}{\mid U_R(V) \mid}\right).$$

Weights model the presence of errors in KGs. Consider the case of negative rule discovery, where V contains positive examples from the graph. We report in the experimental evaluation several negative rules with significant coverage over V, which corresponds to errors in the KG.

3.2 Problem Definition

We can now state the approximate version of the problem.

Definition 4. Given a KG kg, two sets of element pairs G and V from kg with $G \cap V = \emptyset$, all the valid Horn Rules R for kg, and a weight function w for R, a solution for the *robust discovery problem* is a subset R' of R such that:

$$\underset{R'}{\operatorname{argmin}}(w(R') | C_{R'}(G) = G).$$

The *robust* version of the discovery problem aims at identifying rules covering all elements in G and as few elements as possible in V. As we want to avoid overfitting rules in R, we do not allow in the solution rules with constants for both target variables.

Our problem definition can be mapped to the NP-complete *weighted set cover problem* [16]. The reduction follows from the following mapping: the set of elements (universe) corresponds to the generation pairs in G, the input sets are identified by the rules in R (where each rule covers a subset of G), the non-negative weight function $w : r \rightarrow \mathbb{R}$ is $w(r)$ (Definition 2), and the cost of R is defined to be its total weight (Definition 3).

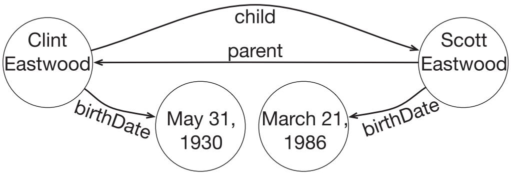

Fig. 1. Four DBpedia facts in the graph representation.

4 GENERATION OF RULES AND EXAMPLES

We start by discussing the generation of the universe of all possible rules. We first assume that examples in G and V are available and then discuss methods to obtain them automatically. Our solution does not depend on how examples are obtained. We explicitly generate them in our approach, but the same rule mining algorithms apply if humans provide pairs for G and V.

We describe the mining of *positive* rules with correct facts in G and incorrect ones in V. When mining *negative* rules, our rule generation and mining algorithms are the same: It is enough to swap the role of generation set G (therefore containing false facts) and validation set V (in this case, containing true facts). The example generation algorithm also applies in both scenarios.

4.1 Rule Generation

We represent a KG kg as a directed graph: elements (entities and literals) form the nodes with a directed edge from node a to node b for each fact $\langle a, rel, b\rangle \in kg$. The relation rel that connects subject to object is used as label for the corresponding edge. We report four facts in Figure 1.

If we allow navigation of edges independently of the edge direction, then we can navigate kg as an *undirected graph*. The body of a rule can be seen as a path in the undirected graph. In Figure 1, the body $\texttt{child}(a,b) \wedge \texttt{parent}(b,a)$ corresponds to the path *Clint Eastwood* – *Scott Eastwood* – *Clint Eastwood*. As defined in Section 2.1, a valid body contains target variables a and b at least once, every other variable at least twice, and atoms are transitively connected. Given a pair of elements (x, y), a *valid body* is a valid path p on the undirected graph s.t.: (i) p starts at the node x; (ii) p covers y at least once; (iii) p ends in x, in y, or in a different node that has been already visited. Given the body of a rule r_{body}, r_{body} covers a pair of elements (x, y) iff there exists a valid path on the graph that corresponds to r_{body}. This implies that for a pair of elements (x, y), we can generate bodies of all possible valid rules by computing all valid paths that start from x with a breadth-first search. The ability to navigate each edge in any direction by turning the original directed graph into an undirected one is key for the generation task, but we must preserve information about the original direction of the edges. This is used when translating paths to rule bodies. In fact, an edge directed from a to b produces the atom $\texttt{rel}(a,b)$, while b to a produces $\texttt{rel}(b,a)$, according to the direction in the original directed graph.

Since every node can be traversed multiple times, for two elements x and y there might exist a large number of valid paths starting from x. This number is constrained with a *maxPathLen* parameter that limits the search space by determining the maximum number of edges in the path, i.e., the maximum number of atoms in the body of the corresponding rule.

We are now ready to describe the generation of all possible rules. Every rule in R (universe of all possible rules) must cover at least one example in the generation set G. The universe of all possible rules is therefore created by analyzing G elements.

(i) Path Generation. Given an example with elements (x, y), we retrieve all nodes from x or y at distance $maxPathLen - 1$ or less. We collect also the edges in this navigation. The retrieval is a recursive exploration with a queue of elements. The queue is initialized with x and y. For each node e in the queue, we run a SPARQL query against the graph to get nodes (and edges) at distance 1 from e (*single hop queries*). At the nth step, we add the new nodes to the queue iff they are at a distance less than $(maxPathLen - n)$ from x or y. Given the graph for every (x, y), we compute all valid paths that start from every x with a simple DFS exploration.

(ii) Path Evaluation. Computing paths for every example in G is related to the computation of the rule coverage. The *coverage* of a rule r corresponds to the number of G pairs for which there exists a path corresponding to r_{body}. Given that the universe of rules and their coverage over G have been computed, the unbounded coverage is computed over V with a simple execution for each rule of two SPARQL queries over the KG.

Example 5. Consider a scenario where we mine the `sibling` predicate for positive rules. Starting from a positive example with a pair (Natasha, Malia), one path can traverse three nodes such as: Natasha–Barack–Malia. If several pairs in G have this path (siblings have one common parent), then the corresponding rule would be $\texttt{hasParent}(b, a) \wedge \texttt{hasParent}(c, a) \Rightarrow \texttt{sibling}(b, c)$.

In the example, the shared variable a between two `hasParent` atoms corresponds to a join, therefore a comparison based on value equality. Since one of our goals is to mine rich rules, we make use of more atom types during the path generation, as discussed next.

Literal comparison. To discover rich rules with comparisons beyond equalities, the graph should have edges connecting literals with a symbol from $\{<, \leq, \neq, >, \geq\}$, e.g., Figure 1 would contain an edge "<" from node "*March 31, 1930*" to node "*March 21, 1986.*" Unfortunately, the original KG does not have this information explicitly, and materializing such edges among all literals is infeasible.

However, in our algorithm, we discover paths for a pair of elements from G in isolation. The graph for a pair of elements has a size that is orders of magnitude smaller than the KG. The very small number of nodes for a single example graph enables us to represent all the literal pairwise comparisons across them. Besides equality comparisons expressed in joins, we add explicit relationships with comparisons ">," "≥," "<," "≤" between numbers and dates, and ≠ between all literals. These comparisons are added to the graph as normal edges (atoms in the formula): $x \geq y$ leads to $\texttt{rel}(x, y)$, where `rel` is ≥. Once we add comparison edges for every input example graph, the discovery of literal comparisons is equivalent to the discovery with predicate atoms.

Not equal entities. We introduce "not equal" comparison for literal values, but this comparison is useful also for entities. For example, a negative rule may state that if a person a was born in a location *different* from b, then such person cannot be the president of b ($\texttt{bornIn}(a, x) \wedge x \neq b \wedge \texttt{president}(a, b) \Rightarrow \bot$). One way to introduce inequalities among entities in the graph is to add edges among all entity pairs. It is clear that this strategy is inefficient and may actually introduce obvious edges, e.g., persons are different from locations. To limit the search space while aiming at meaningful rules, we therefore exploit the `rdf:type` triples associated to elements, which is usually available in KGs. We add a new inequality edge in the input example graph only between pairs of elements of the same type. In the example, x and b are compared, as they are both locations.

4.2 Negative Examples Generation

Given a KG kg and a predicate $rel \in kg$, generation set G and validation set V are generated in our approach. For positive rule mining, we follow the traditional silver standard creation and set G equals to the facts for predicate rel [35], i.e., all pairs of elements (x, y) such that $\langle x, rel, y\rangle \in kg$. These are going to be mostly correct, depending on the quality of kg. Set V contains counter examples for rel. These must be generated because of the open world assumption in KGs. Differently from mining in relational databases, it is not possible to make the assumption that anything not in a KG is false (closed world assumption); thus, in our graphs, missing information is *unknown*.

However, it is unlikely that two randomly selected elements are a true positive example for any predicate. This intuition leads to a simple way of creating counter examples for any predicate by randomly selecting pairs from the cross-product of the elements [32].

While the simple generation "at random" can generate negative examples with high accuracy, a very small percentage of these element pairs will be *semantically related*. In other terms, the elements will be unrelated in time, space, and topic. This data aspect is of crucial importance when mining negative rules. In fact, two unrelated elements have a smaller number of paths between them than semantically related ones; this is reflected by a lower number of rules that can be mined when counter examples are in G. As we show experimentally, unrelated elements are less likely to lead to meaningful patterns in the KG, and therefore the rules are of lower quality.

It is important to notice that there is no such issue when G contains correct examples, i.e., mining positive rules. In fact, at least one semantic connection is present for any example by generation, since pairs in G are obtained from the triples of the predicate of interest. To generate negative examples that are likely to be correct (truly false facts) and that are semantically related, we generate queries over the graph that identify elements that satisfy these requirements.

First, we exploit the notion of *Local-Closed World Assumption* (*LCWA*) [19, 25] to identify the elements that are likely to be completely described in the KG. The assumption states that if for a given subject and predicate the KG contains at least one object value, then it contains all possible object values; e.g., if a graph has one spouse for Barack Obama, then we assume that all his spouses are in the graph. While this is always the case for *functional* predicates (e.g., `capital`), it might not hold for non-functional ones (e.g., `hasChild`). Under this assumption, we identify elements that are likely to be complete and create negative examples by taking the union of elements satisfying the LCWA. For a predicate rel, a negative example is a pair (x, y) where either x is the subject of at least one triple $\langle x, rel, y' \rangle$ with $y \neq y'$, or y is the object of one or more triples $\langle x', rel, y \rangle$ with $x \neq x'$. For example, if rel = `hasChild`, a query would identify as negative example any pair (x, y) s.t. x has some children in the graph who are not y, or y is the child of someone who is not x.

Second, for a candidate negative example over elements (x, y), x must be connected to y via a predicate that is different from the target predicate. In other words, given a KG kg and a target predicate rel, (x, y) is a negative example if $\langle x, rel', y \rangle \in kg$, with $rel' \neq rel$. These restrictions make the size of V of the same order of magnitude as G and guarantee that, for every $(x, y) \in V$, x and y are semantically related by at least one predicate.

Example 6. A negative example (x, y) for the target predicate `hasChild` has the following characteristics: (i) x and y are not connected by the `hasChild` predicate; (ii) either x has one or more children (different from y) or y has one or more parents (different from x); (iii) x and y are connected by a predicate that is different from `hasChild` (e.g., `colleague`).

Similarly, for a negative example (x, y) for predicate `birthDate`: (i) x and y are not in a `birthDate` relationship; (ii) x has a birth date (different from y) or y appears in a birth date relation with a subject (different from x); (iii) x and y are connected by a predicate different from `birthDate`.

To enhance the quality of the input examples and avoid cases of mixed types, we require that for every example pair (x, y), either in G or V, all the x (y) occurrences have the same *type*.

5 DISCOVERY ALGORITHM

We introduce a greedy approach to solve the approximate discovery problem (Section 3.2). Since the number of possible rules can be very large, we introduce an algorithm that generates only promising rules from the KG, while preserving the quality guaranteed by the exhaustive generation.

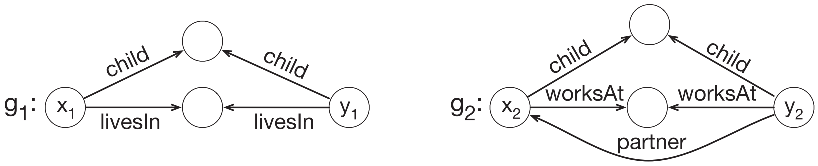

Fig. 2. Two positive examples.

5.1 A Greedy Algorithm Based on Marginal Weight

Our goal is to discover a set of rules to produce a weighted set cover for the given examples. We therefore follow the intuition behind the greedy algorithm for weighted set cover by defining a *marginal weight* for rules that are not yet included in the solution [16].

Definition 5. Given a set of rules R and a rule r such that $r \notin R$, the *marginal weight* of r w.r.t. R is defined as:

$$w_m(r) = w(R \cup \{r\}) - w(R).$$

The marginal weight quantifies the weight increase by adding r to R. It measures the contribution of r to R in terms of new elements covered in G and V. Since we aim at minimizing the total weight, a rule is not added to the solution if its marginal weight is greater than or equal to 0.

When all rules have been generated, the algorithm for greedy rule selection is simple: Given G, V, and the universe of rules R, select the rule r with minimum marginal weight, add it to solution R', and iterate over $R \setminus r$. Algorithm stops when at least one of the three termination conditions is met: (1) all R rules are in the solution; (2) R' covers all elements of G; (3) among the remaining rules in R, none of them has a negative marginal weight.

The greedy algorithm has quadratic complexity over the number of rules and guarantees a $\texttt{log}(k)$ approximation to the optimal solution [16], where k is the largest number of elements covered in G by a rule in R. If the optimal solution is made of rules that cover disjoint sets over G, then the greedy solution coincides with the optimal one.

5.2 Graph Traversal with A^* Search

The greedy algorithm for weighted set cover is based on the fact that the universe of rules R is available. We could generate R by traversing all valid paths from a node x to a node y, for each pair $(x, y) \in G$. However, generating all these paths is not actually needed for every example.

Example 7. Suppose we discover positive rules for predicate spouse. The generation set G includes entities g_1 and g_2 shown as graphs in Figure 2. Assume that all rules in R have the same coverage and unbounded coverage over the validation set V. One possible rule is $r : \texttt{child}(x, v_0) \wedge \texttt{child}(y, v_0) \Rightarrow \texttt{spouse}(x, y)$, stating that entities x and y with a child in common are also married. In the graph, r covers both g_1 and g_2. Since all rules have the same coverage and unbounded coverage over V, any other rule will not bring any benefit to the current solution. In fact, any other candidate will not cover new elements in G, therefore their marginal weights will be negative. Therefore, the exploration of edges livesIn in g_1, worksAt in g_2, and partner in g_2 is superfluous.

Based on the above observation, we avoid generating the entire universe R, but rather follow at each iteration the most promising path on the graph. For each example $(x, y) \in G$, we start the exploration from x. Inspired by the A^* graph traversal algorithm [26], we use a queue of invalid paths (which would lead to invalid rules) and, at each iteration, we pick the path with the minimum marginal weight so far, which corresponds to several paths in the graphs. We expand the path by following the edges and add new paths to the back of the queue. Unlike A^*, we do not stop when a path reaches the target node y, i.e., becomes valid. The algorithm adds the valid path (which would turn into a valid rule) to the solution and keeps looking for other valid paths until one of the termination conditions of the greedy set cover algorithm is met.

To guarantee the optimality of the solution in A^*, the estimation function must be *admissible* [26], i.e., the estimated cost must be less than or equal to the actual cost. We define an admissible estimation of the marginal weight for an invalid path that can still be expanded.

Definition 6. Given a rule $r : A_1 \wedge A_2 \cdots A_n \Rightarrow B$, we say that a rule r' is an *expansion* of r iff r' has the form $A_1 \wedge A_2 \cdots A_n \wedge A_{n+1} \Rightarrow B$.

In the graph exploration, expanding r means traversing one further edge on the path constructed by r_{body}. To guarantee optimality, the estimated marginal weight for a rule r that is invalid must be less than or equal to the actual weight of any valid rule that is generated by means of expanding r. Given a rule and some expansions of it, we can derive the following:

LEMMA 1. *Given a rule r and a set of pair of elements E, then for each r' expansion of r, $C_{r'}(E) \subseteq C_r(E)$ and $U_{r'}(E) \subseteq U_r(E)$.*

The above lemma states that the coverage and unbounded coverage of an expansion r' of r are contained in the coverage and unbounded coverage of r, respectively, and directly derives from the augmentation inference rule for functional dependencies [5], i.e., expanding a rule with a new atom to its body makes the rule more selective.

The only positive contribution to marginal weights is given by $|C_{R\cup\{r\}}(V)|$, which is equivalent to $|C_R(V)| + |C_r(V) \setminus C_R(V)|$. If we set $|C_r(V) \setminus C_R(V)| = 0$ for any invalid r, then we guarantee an admissible estimation of the marginal weight. We estimate the coverage over the validation set to be 0 for any rule that can be further expanded, since expanding it may bring the coverage to 0.

Definition 7. Given an *invalid* rule r and a set of rules R, we define the *estimated marginal weight* of r as:

$$w^*_m(r) = -\alpha \cdot \frac{|C_r(G) \setminus C_R(G)|}{|G|} + \beta \cdot \left(\frac{|C_R(V)|}{|U_{R\cup\{r\}}(V)|} - \frac{|C_R(V)|}{|U_R(V)|} \right).$$

The estimated marginal weight for a valid rule is equal to the actual marginal weight from Definition 5. Given Lemma 1, we can easily see that $w^*_m(r) \leq w^*_m(r')$, for any r' expansion of r. Thus our marginal weight estimation is admissible.

We now introduce Algorithm 1, a modified set cover procedure including the A^*-like rule expansion. For simplicity, we blur the difference between paths and rules in the description; however, the search is on the undirected graph. For a rule r, we call *frontier nodes*, $N_f(r)$, the last visited nodes in the paths that correspond to r_{body} from every example graph covered by r. Expanding r means navigating a single edge from any of the frontier nodes. The set of frontier nodes is initialized with starting nodes x, for every $(x, y) \in G$ (Line 2). The algorithm keeps a queue of rules Q_r, from which it picks at each iteration the rule with minimum estimated weight. The function `expandFrontiers` selects all nodes (along with edges) at distance 1 from frontier nodes and returns the set of all rules constructed with this expansion. Q_r is thus initialized with all rules of length 1 starting at x (Line 3). In the main loop, the algorithm checks if the current best rule r is valid or not. If r is valid, then it is added to the output and it is not further expanded (Line 8). If r is invalid, then it is expanded iff the length of its body is less than *maxPathLen* (Line 10). The termination conditions and the last part of the algorithm are the same as the greedy set cover algorithm.

The combined generation and selection of the rules has two main advantages. First, the algorithm does not materialize the entire graph for every G pair. The algorithm gradually materializes

ALGORITHM 1: RuDiK Rule Discovery.

```
    input: $G$ (generation set), $V$ (validation set), $maxPathLen$ (maximum rule body length)
    output: $R_{opt}$ – union of rules in the solution
 1  $R_{opt} \leftarrow \emptyset$;
 2  $N_f \leftarrow \{x|(x, y) \in G\}$;
 3  $Q_r \leftarrow$ expandFrontiers($N_f$);
 4  $r \leftarrow \underset{r \in Q_r}{\text{argmin}}(w_m^*(r))$;
 5  repeat
 6      $Q_r \leftarrow Q_r \setminus \{r\}$;
 7      if isValid($r$) then
 8          $R_{opt} \leftarrow R_{opt} \cup \{r\}$;
 9      else
            // rules expansion
10          if length($r_{body}$) $< maxPathLen$ then
11              $N_f \leftarrow$ frontiers($r$);
12              $Q_r \leftarrow Q_r \cup$ expandFrontiers($N_f$);
13      $r \leftarrow \underset{r \in Q_r}{\text{argmin}}(w_m^*(r))$;
14  until $Q_r = \emptyset \lor C_{R_{opt}}(G) = G \lor w_m^*(r) \geq 0$;
15  if $C_{R_{opt}}(G) \neq G$ then
16      $R_{opt} \leftarrow R_{opt} \cup$ singleInstanceRule($G \setminus C_{R_{opt}}(G)$);
17  return $R_{opt}$
```

parts of the graph whenever they are needed for navigation (Lines 3 and 12). Second, the weight estimation enables the pruning of unpromising rules that do not cover neither new G nor new V pairs.

5.3 Algorithm Analysis

Complexity. Each iteration of the algorithm picks the next best rule r according to the marginal weight and then expands r by (possibly) generating new rules. The procedure of expanding a rule is the following: For each example e in G covered by r, it takes the frontier node of e and navigates the outgoing and incoming edges of the node in the graph. Each navigated edge might lead to a new rule to be added to the queue Q_r, and for each new discovered rule, we compute its marginal weight by issuing a single query against the KG to compute its coverage over the validation set. If we consider the cost of the query as constant, then the asymptotic complexity of the expansion step is $O(2 \cdot p \cdot |G|)$, where p is the total number of different predicates in the KG, and the multiplier 2 is given by the fact that each predicate can generate two new rules, one for each navigation direction. The expansion step is repeated at most k times, where k is the number of all possible rules we can generate from G. Because each atom in the body of the rule can be either a predicate from the KG or one of the six literal comparisons (Section 4.1), the total number of rules is at most $(2p + 6)^1$ (rule with one atom) $+(2p + 6)^2$ (rule with two atoms) $+ \cdots + (2p + 6)^{maxPathLen}$. Therefore, the asymptotic runtime complexity of Algorithm 1 is $O\left(p^{maxPathLen} \cdot |G|\right)$. As we will outline in the experimental section, the *maxPathLen* parameter plays a crucial role in the runtime of the algorithm.

Optimality. A^* algorithm is guaranteed to return the cheapest path from start to goal if the heuristic function is *admissible*, meaning the heuristic function will never overestimate the actual cost [26]. In our settings, our heuristic is admissible iff the estimated marginal weight for a rule r is always less or equal to the actual marginal weight of r. Because in the estimated weight we set the coverage of r over the validation set to 0 (Definition 7), and since the coverage over the validation set is the only positive contribution to the actual marginal weight, then for every possible rule the estimated marginal weight will always be less or equal to the actual marginal weight. This guarantees that the output of Algorithm 1 will always be equal to the output of the greedy set cover algorithm applied on the universe of all possible rules (Section 5.1).

6 CONDITIONAL RULES

In the previous section, we described an algorithm that discovers rules with only universal variables in the head of the rule. Let us denote them as *generic* rules. These rules are the most applicable, as they can be satisfied by a large number of facts in the KGs. However, an important problem with generic rules is that several of them do not hold semantically in all cases.

This problem is reflected by low accuracy for some of the tasks involving the generic rules, regardless of the mining algorithm. Stating that two persons in a child relationship cannot be in a spouse relationship is correct in most of the cases ($\texttt{child}(x,y) \wedge \texttt{spouse}(x,y) \Rightarrow \perp$), but there are triples in KGs for which this generic rule does not hold, such as facts whose entities are fictional characters. Indeed, the rule becomes more accurate if we restrict its scope by constraining the type of the entities, such as $\texttt{child}(x,y) \wedge \texttt{spouse}(x,y) \wedge \texttt{type}(x, \textit{Royalty}) \wedge \texttt{type}(y, \textit{Royalty}) \Rightarrow \perp$. This observation leads to the idea of improving the quality of the discovered rules by extending them with type selection.

A similar reasoning applies for rules that apply only for a certain entity. For example, the rule we discussed above for presidents and place of birth does not apply for all countries. The correct rule should be discovered for specific countries, such as the one stating that only someone born in the U.S.A. can become an American president ($\texttt{bornIn}(a,b) \wedge b \neq \textit{U.S.A.} \Rightarrow \neg\texttt{president}(a, \textit{U.S.A.})$).

There is an important trade-off with conditional rules. On the one hand, a more specific context decreases the rule applicability, but, on the other hand, the rule becomes immediately more accurate. We therefore extend the rule mining algorithm to discover *conditional rules.*

6.1 Type Condition

We start by using constraints on the element types, such as *Date* for literals and *Organization* for entities. The idea is to group the subject and the object instances by their type and run the mining on each combination of groups. As every element can have one or more types associated, the same entity or literal can belong to different groups.

We rely on the following algorithm to identify the subject and object types for the triples involved in a predicate and then group them according to the different subject-object type combinations:

- *(i)* Extract the types for subject and object elements in the triples for a predicate;
- *(ii)* Create subject and object type groups;
- *(iii)* Create all combinations between type groups, each combination has a type assigned for subject and a (possibly equal) type for object;
- *(iv)* For each combination, generate positive and negative examples (G and V sets), drop the combination if the number of generated examples is smaller than a threshold;
- *(v)* Run generic rule mining for the examples in every remaining combination.

The grouping process leads to discovery of rules that do not apply in general, such as:

$$\texttt{party}(x, v0) \wedge \texttt{party}(y, v1) \wedge v0 \neq v1 \wedge \texttt{type}(x, \textit{Politician}) \wedge \texttt{type}(y, \textit{OfficeHolder}) \\ \wedge\ \texttt{spouse}(x,y) \Rightarrow \perp.$$

It is much rarer to have politicians from different parties who are married, while it is more common for people (typed only with a generic *Person*) who are just registered with different parties.

While the method above is intuitive and already leads to accurate rules with type selection, there are pre-processing steps that better characterize the group types for the triples in the KG. One approach is to use *entity embeddings* [11]. Embeddings encode entities in the KG into a low-dimensional vector space while preserving structural information about the graph [10]. A relationship in the graph can then be interpreted as a translation from subject entity to object entity in such space.

Embeddings characterize entities with hundreds of (learned) features and lead to clusters that are of high quality. Specifically for our application, the clusters can be used to obtain groups that are characterized beyond type information. In clusters from embeddings, entities with popular types, such as *Person* and *Agent*, can be spread across multiple clusters, but types with finer granularity, such as *Politician* and *OfficeHolder*, are grouped together. At the same time, clusters from embeddings are more uniform in other properties that are not captured by type information, such as the structural properties of the entities when expressed as nodes in the graph.

Embeddings can therefore be used to improve the group creation steps (i) and (ii) above: We start by clustering triples with the embeddings and then find subject and object types for each cluster.

6.2 Entity Condition

We allow the discovery of rules with constant selections over the entities. Suppose that for a rule r that states that a person cannot be president of a given country if she was not born in it, all examples of r are people born in the "*U.S.A.*," and there is at least one country for which this rule is not valid. According to our problem statement, the right rule is therefore:

$$\texttt{bornIn}(a, b) \wedge b \neq U.S.A. \Rightarrow \neg\texttt{president}(a, U.S.A.).$$

To discover such atoms, we introduce a refinement of the rule generation. For a given rule r, we promote a variable v in a given rule r to an element e iff for every $(x, y) \in G$ covered by r, v can always be instantiated with the same value e. We allow the substitution of every variable indiscriminately, including variables in the head of the rule. This could potentially lead to rules having only constants, but we do not accept them in our solutions.

7 EXPERIMENTS

The discussed techniques have been implemented in RuDiK [34], our rule mining system (https://github.com/stefano-ortona/rudik). We group the results for our system evaluation into five parts: (i) showing the accuracy of our discovered rules; (ii) comparing our techniques with related systems; (iii) demonstrating how discovered rules create valid training examples for learning; (iv) discussing examples and quality of conditional rules; (v) evaluating the effects of RuDiK parameters.

Setup. We run our evaluation on a desktop with a quad-core i5 CPU at 2.80 GHz and 16 GB RAM. We deployed on the same machine a SPARQL endpoint through OpenLink Virtuoso, optimized for 8 GB RAM of available memory. Whenever not reported explicitly, we use weight parameters $\alpha = 0.3$ ($\beta = 0.7$) for positive rules, $\alpha = 0.4$ ($\beta = 0.6$) for negative rules, and allow at most 3 atoms in the body of a rule ($maxPathLen = 3$). We discuss in detail these parameters in Section 7.5.

Metrics. We evaluated the quality of the discovered rules as the main metric to judge the effectiveness of the mining. For each KG, we started by sorting predicates w.r.t. their popularity, i.e., the number of facts for the predicate. We then chose the top three predicates for which we knew there

Table 1. Dataset Characteristics

KG	*Version*	*Size*	*#Triples*	*#Predicates*
DBPedia	3.7	10.06 GB	68,364,605	1,424
Yago 3	3.0.2	7.82 GB	88,360,244	74
Wikidata	20160229	12.32 GB	272,129,814	4,108

Table 2. RuDiK Rule Precision

	Positive Rules			Negative Rules			
	Avg Exec.	*Avg Precision Predicates*	*#*	*Avg Exec.*	*# Detected*	*Avg Precision*	*#*
KG	*Time*	*with Rules (All)*	*Labels*	*Time*	*Errors*	*All Predicates*	*Labels*
DBPedia	35 mins	**97.14%** (63.99%)	139	19 mins	499	92.38%	84
Yago 3	59 mins	**84.44%** (62.86%)	150	10 mins	2,237	90.61%	90
Wikidata	141 mins	**98.95%** (73.33%)	180	65 mins	1,776	73.99%	105

existed at least one significant rule and another 2 top predicates for which we were not aware of the existence of significant rules.

We carried out the evaluation of the discovered rules according to the best practice for rule evaluation [25]. If for a rule it was possible to declare it semantically correct in all cases, then we marked all the triples coming from the application of the rule over the KG as true. Similarly for rules that were clearly incorrect, all its output triples would be marked as false. If a rule correctness was not clear just by reading its formula, then we randomly sampled 30 triples from the application of the rule, which would be either new facts (positive rules) or detected errors (negative rules), and manually checked each of the facts. The *precision* of a rule is then computed as the ratio of correct triples to all of its output triples.

7.1 Quality of Generic Rules Discovered by RuDiK

The first set of experiments computed the accuracy of output rules over three widely used KGs: DBPedia, Yago, and Wikidata. Table 1 shows the characteristics of these KGs.

The size of a KG matters, as loading all the triples in memory requires to either use powerful machines [14, 23] or to drastically shrink it by eliminating literal values [25]. Given the small memory footprint of our algorithm, we are able to discover rules with commodity HW resources without dropping the literals, which are crucial for obtaining meaningful rules. While RuDiK works with a target predicate at a time, we can use it to discover rules on the entire KG by using each predicate as input. Next, results are discussed for subsets of predicates, since the manual annotation of the computed new facts and errors is an expensive process. However, when RuDiK is executed on all the predicates of a KG, results are consistent in terms of size of the output and execution times. For example, for roughly 600 predicates in DBPedia, we mined about 3K positive rules, with at most 26 rules per predicate, and 4K negative rules, with at most 32 rules per predicate.

Positive Rules. We evaluate the precision for the discovered positive rules on the top five predicates for each KG. The number of new triples varies significantly across rules. To avoid the overall precision to be dominated by rules with big output, we first compute the precision for each rule and then average values over all discovered rules. Table 2 shows precision values, along with average running time (per predicate), and the number of manually annotated facts. We distinguish predicates for which there existed at least one meaningful rule (in bold) and all predicates.

Table 3. Datasets for the Comparison with AMIE

KG	*Size*	*# Triples*	*# Predicates*	*#* `rdf:type`
DBPedia-NL	551M	7M	10,342	22.2M
Yago 2-NL	48M	948.3K	38	77.9M

For predicates such as `academicAdvisor`, `child`, and `spouse`, the output rules show a precision above 95% in all KGs. However, when we consider other predicates, average values are brought down by few cases, such as `founder`, where valid positive rules probably do not exist at all. When a valid rule existed, the system was able to find it, but it did not recognize all cases where no positive rules existed. In our experience, it was sufficient to read the rules to identify semantically incorrect rules. Also, results show that the more accurate is the KG, the better is the quality of the positive rules. Wikidata contains few errors, since it is manually curated, while DBPedia and Yago are automatically generated by information extractors, hence their quality is lower.

The size of the KG and the target predicate at hand have impact on the run time. The more the connections of a node (KG element), the more paths we traverse in the graph. Some elements have a large number of incoming edges, e.g., entity "*China*" in Wikidata has more than 600K. When the generation set includes such popular elements, the traversal of the graph becomes slower. Parameter *maxPathLen* also has a big impact on the run time (as we discuss in Section 7.5).

Negative Rules. Negative rules are evaluated as the percentage of actual incorrect triples over all discovered triples. Table 2 reports, for every KG, the total number of potential erroneous facts discovered with the output rules, whereas the precision is given as the percentage of actual errors among all the triples identified as possibly incorrect.

Generic negative rules have better accuracy than positive ones when averaging all predicates, as there are more in reality. While the quality of the rules is good, especially on the noisy KGs, we also discover rules that are supported by the vast majority of the data but do not hold semantically. As an example, RuDiK identifies the rule that two people cannot be married if they have the same gender both in Yago and Wikidata. Such rule has a precision of 94% in Yago and of 57% in Wikidata. Literals' role is bigger in negative rules, compared to the positive case. In fact, several valid negative rules rely on temporal aspects where something cannot take place before/after something else. Temporal data are usually encoded through dates' and years' literal values.

Mining negative rules is usually faster than mining positive ones , because of the different nature of the examples covered by validation queries. Whenever we identify a potentially valid rule, we translate its body to a SPARQL query against the KG to obtain its coverage over V. Such queries are faster for the negative case, because V has only entities connected by one predicate, whereas this is not the case in the mining of positive rules.

7.2 Comparison to Existing Systems

We evaluated our techniques against AMIE [25], a rule mining system for KGs. AMIE is based on the assumption that the target KG fits into memory and mines positive rules for every predicate in the KG. It then ranks rules according to a support function.

Given its in-memory implementation, AMIE could not compute a solution on the KGs of Table 1. Therefore, we used their reduced versions, which do not include literals and `rdf:type` triples [25]: DBPedia-NL, Yago 2-NL. Without literals and `rdf:type` facts, the size of the KG drops drastically. We run AMIE on its original datasets. Since our method needs type information (for the generation of G and V and inequality atoms exploration), for our case, we used the reduced, NL versions plus `rdf:type` triples. Last column of Table 3 shows the number of such triples.

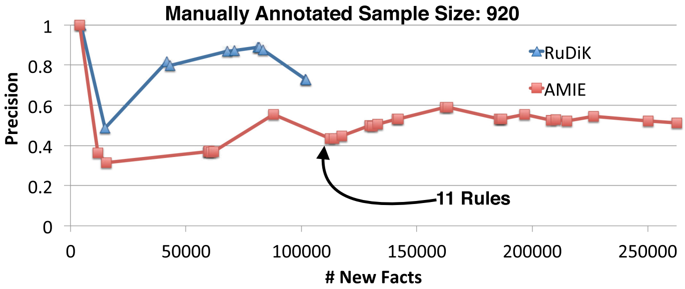

Fig. 3. Precision for new triples obtained by running rules in descending AMIE's score on YAGO 2-NL.

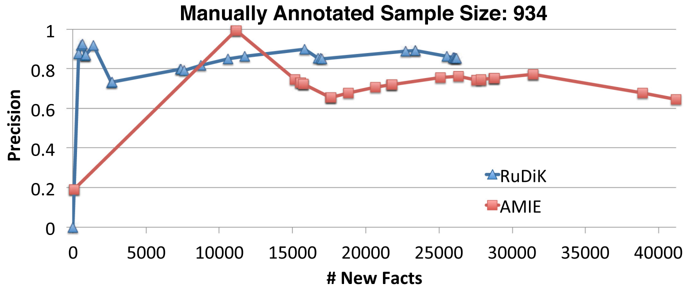

Fig. 4. Precision for new triples obtained by running rules in descending score on DBPEDIA-NL.

Positive Rules. We first compare against AMIE on positive rule discovery. In this setting, we ran RuDiK with $maxPathLen = 2$ (AMIE default setting).

AMIE's output consisted of 75 rules in YAGO and 6,090 in DBPEDIA. Following their experiments, we picked the top 30 rules according to their score. We then chose the rules produced by our method involving the same head predicates of the top 30 rules produced by AMIE.

Figures 3 and 4 report the results on YAGO-NL and DBPEDIA-NL, respectively. We plot the total cumulative number of new unique triples (x-axis) versus the averaged precision (y-axis) when incrementally including in the output the rules according to their descending (AMIE's) score. Overall, rules from AMIE output more facts, but with significantly lower accuracy in both KGs. This is because many valid rules are preceded by meaningless ones in the ranking, and it is not easy to set a proper k to get the best output. RuDiK relies on a coverage function that identifies inherently meaningful rules with enough support. As a consequence, RuDiK outputs only 11 rules for 8 predicates on the entire YAGO—no rules with enough support are discovered for the remaining predicates. If we limit the output of AMIE to the top 11 rules in YAGO (same size as our approach), then its final accuracy is still 29% below our approach, with just 10K more facts predicted.

Negative Rules. Even though AMIE is not designed to mine negative rules, we created a baseline solution on top of it. First, we generated negative examples (Section 4.2) for each predicate in the top-5. For each negative pair, we added a triple to the KG by connecting the two elements with the *negation* of the predicate. For example, we injected a triple for the notSpouse predicate for every pair of not married people. We then executed AMIE on these negated predicates.

Table 4 reports that RuDiK outperforms AMIE in both KGs with an overall precision gain of roughly 20% (41%–49% relative). The drop in quality for RuDiK w.r.t. the results in Section 7.1 is due to the absence of literal values. When RuDiK exploits literals, its benefit in accuracy is more than 50% absolute. Numbers in brackets show the number of facts manually annotated for the evaluation.

Table 4. Quality of the Negative Rules in the Comparison vs. AMIE

	AMIE		RuDiK (no literals)	
KG	*# Errors*	*Precision*	*# Errors*	*Precision*
DBPEDIA-NL	457 (157)	38.85%	148 (73)	**57.76%**
YAGO 2-NL	633 (100)	48.81%	550 (35)	**68.73%**

Table 5. Comparison of the Total Execution Times

KG	*# Predicates*	AMIE	RuDiK	*Types*
YAGO 2-NL	20	30s	18 m,15 s	12 s
YAGO 2s-NL	26 (38)	>8h	47 m,10 s	11 s
DBPEDIA 2.0-NL	904 (10,342)	>10h	7 h,12 m	77 s
DBPEDIA 3.8-NL	237 (649)	>15h	8 h,10 m	37 s
WIKIDATA-NL	118 (430)	>25h	8 h,2 m	11 s
YAGO 3	72	-	2 h,35 m	128 s

Execution Time. On our machine, AMIE could complete its mining only on YAGO 2-NL, while for other KGs it got stuck after some hours. For these cases, we stopped the mining if there were no changes in the interactive output for more than two hours. Running times for AMIE differ from Reference [25], where the computation was done on a 48 GB RAM server.

Table 5 reports the running time on several KGs. Five KGs have been removed of the literal values (NL), while YAGO 3 is complete. The second column reports the total number of predicates for which AMIE computed at least one rule before getting stuck, while in brackets is the total number of predicates in the KG. The third and fourth columns show the total running time of the two techniques. Despite being disk-based, RuDiK successfully completes the mining faster than AMIE in all cases, with the exception of YAGO 2. This is due to the small size of this KG, which fits in memory, but complete KGs (such as YAGO 3) could not be loaded because of out-of-memory errors. The last column shows the running time to collect `rdf:type` triples for all predicates.

Other Systems. There are other available systems to discover rules in KGs. In Reference [4], the approach discovers rules that are more specific than our technique; on YAGO 2, it discovers 2K new triples with a precision lower than 70%, while the first rule we discover on YAGO 2 already discovers more than 4K triples with a 100% precision. Another system [14] implements AMIE algorithm with a focus on distribution and scalability. The mining part is not modified, hence the output is the same as AMIE. We did not compare against classic Inductive Logic Programming systems [17, 42], as these are already significantly outperformed by AMIE both in accuracy and run time.

7.3 Using the Rules in a Learning Algorithm

The focus of these experiments is to demonstrate the applicability of our approach in providing Machine Learning (ML) algorithms meaningful training examples. We picked DeepDive [38], an ML framework for relation extraction. DeepDive extracts entities and relations from free text articles using distant supervision. The main idea behind distant supervision is to use an external source of truth (e.g., a KG) to provide training examples for supervised algorithms. For example, DeepDive can extract mentions of married people from text documents. It first uses DBPEDIA to label some pairs of entities as *true* positive (pairs of married couples found in DBPEDIA). Unfortunately, KGs can provide positive examples only. We compared the accuracy of DeepDive on the

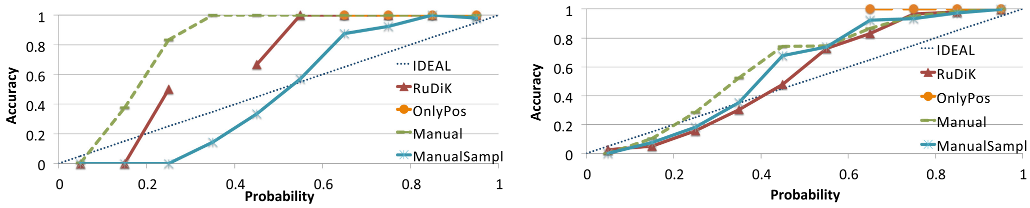

Fig. 5. DeepDive accuracy results with different training examples—1K (left) and 1M (right) articles.

spouse example trained with three different negative examples sets over two datasets. To evaluate our approach, we created negative examples with the rules obtained on DBPedia with RuDiK.

The plot in the left-hand side of Figure 5 reports DeepDive accuracy plot run on the first dataset (1K documents). The plot shows the fraction of correct positive predictions over total predictions (y-axis), for each probability value (x-axis). The dotted blue line marks the perfect algorithm, which would predict all new facts with 100% probability and zero facts with 0%. The less a technique deflects from the dotted line, the better is its accuracy. We report four approaches: RuDiK represents DeepDive trained with negative examples generated by our rules; `OnlyPos` leverages only on positive examples from DBPedia; `Manual` uses positive examples from DBPedia and manually crafted rules to generate negative examples; `ManualSampl` randomly samples a subset of the negative examples generated manually equal in size to the positive set. `OnlyPos` and `Manual` cannot provide valid training, as they either have only positive examples and thus label everything as true, or have way more negative examples than positive and label everything as false. `ManualSampl` is the clear winner, while RuDiK suffers from the absence of data: Over the given 1K articles, we found only 20 positive and 15 negative examples from DBPedia. Missing points for RuDiK are also caused by the lack of evidence in the training data, with no predictions in the 25%–45% range.

The plot in the right-hand side of Figure 5 shows how with the second dataset (1M articles), all methods except `OnlyPos` successfully steer DeepDive in the training, with RuDiK leading to the best result. This is due to the quality of the negative examples: Our rules identify diverse enough examples that enable DeepDive to understand distinctive features. We can use our technique to simulate user behavior and automatically produce qualitative negative rules.

7.4 Conditional Rules

We now compare the results for the discovery of generic rules with the RuDiK, the conditional rules coming from the type group analysis, and the conditional rules coming from clustering with embeddings followed by type grouping. The experiment has been executed on the full DBPedia for the six largest clusters obtained with *TransE* [10]; we then kept the type groups with at least 20 subject (object) elements.

From the rules reported in Table 6, it is easy to see that mining with grouping leads to meaningful rules for a subset of the entities. For example, a positive one stating that two artists in the same band are likely to be its co-founders (e_5), or the negative rule stating that two politicians in different parties are unlikely to be married (e_8). Similarly for the rules coming from clustering and grouping, where rules for artists are identified both for `spouse` and `foundedBy` (e_9, e_{12}). It is interesting to see that different pre-processing algorithms lead to different sets of rules. As there is no clear winner, we argue that all methods should be executed to get the richest possible set of rules.

Evaluating the Quality of Conditional Rules. We evaluate the quality of conditional rules versus generic rules by comparing the outcome of the rules for two DBPedia predicates: `spouse` and `foundedBy`. For each predicate, we consider both positive and negative rules. Evaluation of the

Table 6. Sample of Discovered Rules from DBPedia

Generic Rules
$e_1 : \texttt{foundedBy}(v0, y) \land \texttt{foundedBy}(v0, v1) \land \texttt{foundedBy}(x, v1) \Rightarrow \textbf{foundedBy}(x, y)$
$e_2 : \texttt{birthDate}(y, v0) \land v0 > v1 \land \texttt{foundingYear}(x, v1) \land \textbf{foundedBy}(x, y) \Rightarrow \bot$
$e_3 : \texttt{parent}(v0, x) \land \texttt{parent}(v0, y) \Rightarrow \textbf{spouse}(x, y)$
$e_4 : \texttt{spouse}(y, v0) \land \texttt{parent}(x, v0) \land \textbf{spouse}(x, y) \Rightarrow \bot$

Conditional Rules from Type Grouping
$e_5 : \texttt{foundedBy}(x, v0) \land \texttt{associatedBand}(v0, v1) \land \texttt{associatedBand}(y, v1) \land \texttt{type}(x, \mathit{Company}) \land \texttt{type}(y, \mathit{Artist}) \Rightarrow \textbf{foundedBy}(x, y)$
$e_6 : \texttt{parentCompany}(y, v0) \land \texttt{owningCompany}(v0, v1) \land \texttt{owningCompany}(x, v1) \land \texttt{type}(x, \mathit{Company}) \land \texttt{type}(y, \mathit{Organisation}) \land \textbf{foundedBy}(x, y) \Rightarrow \bot$
$e_7 : \texttt{parent}(v0, x) \land \texttt{parent}(v0, y) \land \texttt{type}(x, \mathit{BritishRoyalty}) \land \texttt{type}(y, \mathit{Royalty}) \Rightarrow \textbf{spouse}(x, y)$
$e_8 : \texttt{party}(y, v0) \land \texttt{party}(x, v1) \land v0 \neq v1 \land \texttt{type}(x, \mathit{Politician}) \land \texttt{type}(y, \mathit{OfficeHolder}) \land \textbf{spouse}(x, y) \Rightarrow \bot$

Conditional Rules from Entity Clustering and Type Grouping
$e_9 : \texttt{occupation}(y, v0) \land \texttt{occupation}(v1, v0) \land \texttt{foundedBy}(x, v1) \land \texttt{type}(x, \mathit{Organisation}) \land \texttt{type}(y, \mathit{Artist}) \Rightarrow \textbf{foundedBy}(x, y)$
$e_{10} : \texttt{foundingYear}(x, v0) \land v0 < v1 \land \texttt{foundingYear}(y, v1) \land \texttt{type}(y, \mathit{Organisation}) \land \texttt{type}(x, \mathit{Organisation}) \land \textbf{foundedBy}(x, y) \Rightarrow \bot$
$e_{11} : \texttt{parent}(v0, x) \land \texttt{parent}(v0, y) \land \texttt{type}(x, \mathit{Royalty}) \land \texttt{type}(y, \mathit{Royalty}) \Rightarrow \textbf{spouse}(x, y)$
$e_{12} : \texttt{activeYearStartYear}(x, v0) \land v0 > v1 \land \texttt{deathDate}(y, v1) \land \texttt{type}(x, \mathit{Artist}) \land \texttt{type}(y, \mathit{Person}) \land \textbf{spouse}(x, y) \Rightarrow \bot$

Table 7. Comparison between Conditional and Generic Rules

	Conditional Rules			**Generic Rules**		
Predicate	*#Rules*	*Avg Precision*	*Avg #Triples*	*#Rules*	*Avg Precision*	*Avg #Triples*
spouse	25	**81.8%**	2,785	4	77.5%	57,314
negSpouse	78	**99.7%**	4,977	13	98.5%	13,602
foundedBy	22	**57.0%**	902	5	38.0%	15,071
negFoundedBy	31	95.8%	786	6	95.8%	13,146

quality of rules is based on three different metrics: number of rules extracted, average of manually estimated precision for the rules, and average number of triples in the output of the rules. For computing the estimated precision of a given rule r, we randomly choose 20 triples from DBPedia that satisfy r and then check to see how many of those triples are true (or errors in the case r is a negative rule). The estimated precision is the number of correct instances divided by 20.

From the results reported in Table 7, it is clear that we identify a larger number of rules with our proposed extension for a given predicate. For grouping and clustering rules, the precision is higher than generic ones, but their scope is obviously lower. For all predicates, except negFoundedBy, the precision of conditional rules is higher. As expected, an absolute average precision improvement of 6.1% comes at the cost of a lower number of triples in the output, e.g., 70K vs. 229K for spouse.

7.5 In-depth Evaluation

We study the impact of individual internal components and methods in RuDiK.

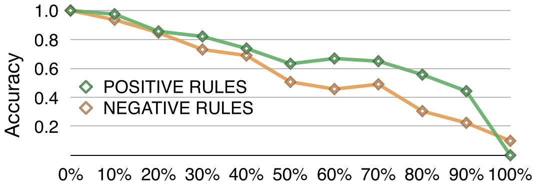

Fig. 6. Impact of KG noise on rules quality.

Table 8. Effect of Negative Examples Generation Strategy on Rule Quality

Strategy	*# Potential Errors*	*Precision*
Random	247	**95.95%**
LCWA	263	95.82%
RuDiK	**499**	92.38%

KG Noise. In terms of KG quality, the percentages of erroneous facts discovered by our methods are 0.23% for Wikidata, 0.26% for DBPedia, and 0.6% for Yago. To evaluate the impact of noise in the KG, we first manually removed errors from the top five predicates in DBPedia to generate clean examples. We mined rules from such examples and fixed the result as the best outcome. We then injected errors by switching positive and negative facts between the sets and did the mining. Figure 6 reports the degradation in accuracy averaged over predicates (y-axis) from 0% to 100% injected noise (x-axis). The accuracy decreases with increasing amount of noise, but RuDiK is robust enough to compute mostly valid rules until 40% of errors, followed then by a significant drop in accuracy. RuDiK is able to mine some valid rules even with 90% of errors.

Local Completeness Assumption. We study the impact of the negative examples generation on rules' quality. For a predicate p, we compare three generation techniques: RuDiK strategy (as explained in Section 4.2), Random (randomly select k pairs (x, y) from the Cartesian product s.t. $\langle x, p, y\rangle \notin kg$), and LCWA (RuDiK strategy without the constraint that x and y must be connected by a predicate not equal to p). Table 8 reports the quality results for negative rules. *Random* and *LCWA* have similar accuracy, with slightly better results than RuDiK. This is because whenever we pick examples from the Cartesian product, the likelihood of picking entities from different time periods is very high, and negative rules relying on time features are usually correct. Instead, if we force x and y to be connected with a different predicate, the two elements are semantically related and lead to different types of rules. A rule such as $\texttt{parent}(a, b) \Rightarrow \texttt{notSpouse}(a, b)$ is not generated with random strategies, since the likelihood of choosing two persons in a parent relation is extremely low. The technique in RuDiK enables the mining of different types of rules.

Literals. Since previous rule discovery approaches exclude literals from the mining [4, 14, 25], we quantify the impact of literal rules. Table 9 shows the precision obtained by running RuDiK including and excluding the use of literal comparisons. Enabling literal values has a tangible impact on accuracy, both for positive and negative cases. Negative rules without literals find less than half potentially erroneous facts (numbers in brackets) with a lower accuracy. Predicate `founder` is the most evident example: 79 potential errors with a 95% accuracy are discovered with literal values, while 0 are computed without using literals. Another prominent example is `birthDate`, for which almost all negative rules included literal comparisons (e.g., $\texttt{activeYearsStartDate}(x, v0) \land \texttt{birthDate}(x, y) \land y > v0 \Rightarrow \bot$). The pruning effect of the A^* search also reduces

Table 9. Impact of Literals on DBPedia

	With Literals		**Without Literals**	
Rules	*Run Time*	*Precision*	*Run Time*	*Precision*
Pos.	~35min	**63.99**%	~54min	60.49%
Neg.	~19min	**92.38**% (**499**)	~25min	84.85% (235)

Table 10. Impact of the *maxPathLen* Parameter on DBPedia

	MaxPathLen = **2**		*MaxPathLen* = **3**		*MaxPathLen* = **4**	
Rules	*Run Time*	*Precision*	*Run Time*	*Precision*	*Run Time*	*Precision*
Pos.	~**3min**	49.2%	~35min	64.0%	~>24h	**66.0**%
Neg.	~**56sec**	90,0%	~20min	**92.4**%	~>24h	92.7%

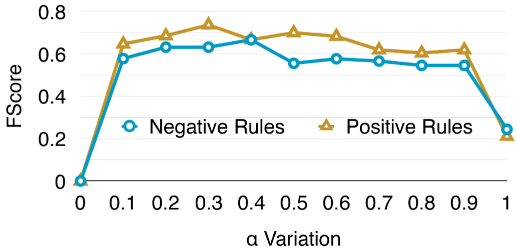

Fig. 7. Impact of α on quality performance.

the running time with literal values: RuDiK can find valid rules earlier and hence stop the computation.

Rules Length. The *maxPathLen* parameter fixes the maximum number of atoms in the body of a generic rule. Low values for *maxPathLen* may exclude from the search space meaningful rules, while high values exponentially increase the search space and consequently the running time. We show in Table 10 that with $maxPathLen = 2$, the run time significantly improves, at the expense of losing several meaningful rules and a drop of precision to 49% for positive rules and 90% for negative rules. Specifically, rules with literal comparisons are lost, as these require at least three atoms in the body. On the other side of the spectrum, with $maxPathLen = 4$RuDiK could not finish the computation within 24 hours for any predicate. The accuracy of rules discovered in 24 hours of computation is comparable to the one obtained with $maxPathLen = 3$, with only a small increase for positive rules. Generic rules with 4 atoms oftentimes return an empty result when executed over the KG, i.e., they are less useful, as they have a narrower scope. We identified $maxPathLen = 3$ as a good compromise between running time and rule accuracy. However, we also discover conditional rules with up to 5 atoms.

Weight Parameters. We validated the choice of the values for α and β. Given a manually crafted set of rules, Figure 7 shows that for positive rules, the best assignment is $\alpha = 0.3$ ($\beta = 0.7$), while for negative rules is $\alpha = 0.4$ ($\beta = 0.6$). Since discovering valid positive rules is more difficult than the negative case, giving more weight to precision over recall produces the best result; whereas, for negative rules, we maximize F-measure by favoring the recall. In both cases, the variation in quality for $\alpha \in [0.1, 0.9]$ is small ($\leq$12%), proving the ability of the set cover formulation to provide high-quality rules in most settings.

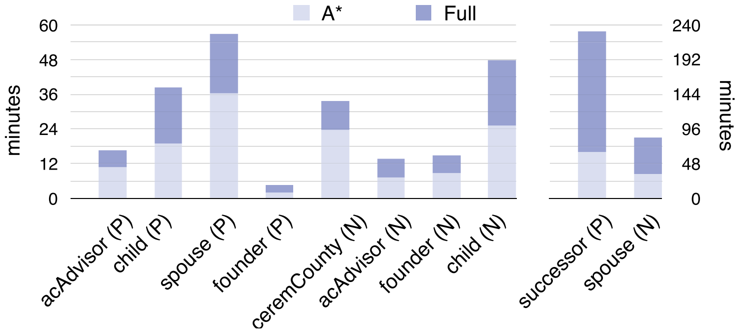

Fig. 8. Runtime for Rule Generation.

Greedy Search. We show the impact of A^* algorithm in the rule generation step. Figure 8 reports the running time, for each predicate, of the A^* algorithm (light-colored bars) against a different version that first computes the universe of all possible rules and then applies the greedy set cover on it (dark-colored bars). The last two columns refer to the y-axis on the right-hand side, as they have significantly different run times. (P) refers to positive rules, while (N) to negative ones. The A^* navigation allows the pruning of several unpromising paths and avoids the exploration of such paths, thus saving upon querying the corresponding RDF entities from disk. This is reflected in the running times with an average of 50% improvement.

Ranking vs. Set Cover. The set cover problem formulation leads to a compact set of rules, which is preferable to the large output obtained via a ranking-based solution. To support our argument, we observe that oftentimes valid rules are hidden low in the ranking, and we report cases where the correct rules were below the 100th position. For the mining of negative rules for predicate `founder`, there was only one valid rule in the output, i.e., a person cannot be the founder of a company if the company was founded before the person was born. The rule appeared at position 127 in the ranking-based version of RuDiK, whereas it was included in output of the classic variant of RuDiK.

8 RELATED WORK

For *relational data*, dependencies are discovered over the attributes of a given schema and encoded into formalisms, such as Functional Dependencies [3, 5, 28] and Denial Constraints [15, 20]. However, these techniques cannot be applied to KGs for three main reasons: (i) the schema-less nature of RDF data and the open world assumption; (ii) traditional approaches rely on the assumption that data are either clean or have a small amount of errors, which is not the case with KGs; (iii) even when the algorithms are designed to support more errors [1, 30], there are scalability issues on large RDF datasets: a direct application of relational database techniques on the graph requires the materialization of all possible predicate combinations into relational tables. Recently, Fan et al. [22] laid the theoretical foundations of Functional Dependencies on Graphs (GFDs) and have introduced an algorithm for their discovery [21]. However, their language does not include general literal comparisons, which we have shown to be useful when detecting errors in KGs.

RuDiK is the first approach that is generic enough to mine both positive and negative rules in RDF KGs. Rule mining approaches designed for positive rule discovery in KGs, such as AMIE [25] and OP algorithm [14], load the entire graph into memory prior to the traversal step. This is a constraint for their applicability over large KGs, and neither of these two approaches can afford value comparison. In contrast to them, by generating the graph on-demand, RuDiK discovers rules on a small fraction of the graph. This makes it scalable, and the low memory footprint enables a bigger search space with rules that can have literal comparisons. We showed in the experimental section how RuDiK outperforms AMIE both in final accuracy and running time. Finally, Reference [4] recommends new facts by using association rule mining techniques. Their rules are made only of constants and are therefore less general than the rules generated by RuDiK.

ILP systems such as WARMR [17] work under the CWA and require the definition of error-free positive and negative examples. These assumptions do not hold in KGs, and AMIE outperforms these two systems [25]. Sherlock [37] is an ILP system that extracts first-order Horn Rules from Web text. While extending RuDiK to free text is an interesting future work, the statistical significance estimate needs a threshold to discover meaningful rules. Error detection in KGs has also been studied by using ILP methods to discover axioms concerning properties' domain and range restrictions that identify contradictions [42]. Another approach identifies outliers after grouping subjects by type [44]. For example, for "population" it groups by "city" and "state" and then detects anomalies. Both methods are orthogonal and complementary to our negative rules. Finally, the generation of new facts in a graph is related to the task of *link prediction* [10].

9 CONCLUSION

We introduced a rule mining system that discovers generic and conditional declarative rules from noisy and incomplete KGs. The system discovers both positive rules, which suggest new potential facts for the KG, and negative rules, which identify inconsistent triples. Negative rules not only identify potential errors in KGs, but also generate representative training data for ML algorithms. We showed experimentally that our system generates concise sets of precise rules and can scale nicely with the size of existing KGs.

Given the recent efforts in the interactive discovery of the rules for relational data [27, 36], open questions are related to similar tasks in KGs. The size of G has a direct impact on the search space and hence on the running time. Since we generate all valid rules for each example in G, the search space grows roughly linearly with its size. If we could identify a subset of examples that lead to the generation of all valid rules, then we could use only those few examples. However, it is not clear how to sample examples while not compromising on the quality of the mined rules. A second promising research direction is the expressiveness of the language. We aim at mining even richer rules that better exploit spatial and temporal information through a smart analysis of literal values' distributions and correlations [1], e.g., "if two persons have age difference greater than 100 years, then they cannot be married." Finally, we are exploring how to use rules generated by our system in application beyond data curation, such as fact checking of textual content [6, 29].

REFERENCES

[1] Ziawasch Abedjan, Cuneyt G. Akcora, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2015. Temporal rules discovery for web data cleaning. *PVLDB* 9, 4 (2015), 336–347.

[2] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting data errors: Where are we and what needs to be done? *PVLDB* 9, 12 (2016), 993–1004.

[3] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2017. Data profiling: A tutorial. In *Proceedings of the SIGMOD.* 1747–1751.

[4] Ziawasch Abedjan and Felix Naumann. 2014. Amending RDF entities with new facts. In *Proceedings of the ESWC.* 131–143.

[5] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[6] Naser Ahmadi, Joohyung Lee, Paolo Papotti, and Mohammed Saeed. 2019. Explainable fact checking with probabilistic answer set programming. In *Proceedings of the TTO.*

[7] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog system: Datalog-based reasoning for knowledge graphs. *PVLDB* 11, 9 (2018), 975–987.

[8] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. 2009. DBpedia-A crystallization point for the web of data. *Web Sem.: Sci. Serv. Agents WWW* 7, 3 (2009), 154–165.

[9] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the SIGMOD*. 1247–1250.

[10] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Proceedings of the NIPS*.

[11] Antoine Bordes, Jason Weston, Ronan Collobert, and Yoshua Bengio. 2011. Learning structured embeddings of knowledge bases. In *Proceedings of the AAAI*.

[12] Mirko Bronzi, Valter Crescenzi, Paolo Merialdo, and Paolo Papotti. 2013. Extraction and integration of partially overlapping web sources. *PVLDB* 6, 10 (2013), 805–816.

[13] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr, and Tom M. Mitchell. 2010. Toward an architecture for never-ending language learning. In *Proceedings of the AAAI*. 1306–1313.

[14] Yang Chen, Sean Goldberg, Daisy Zhe Wang, and Soumitra Siddharth Johri. 2016. Ontological pathfinding. In *Proceedings of the SIGMOD*. 835–846.

[15] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering denial constraints. *PVLDB* 6, 13 (2013), 1498–1509.

[16] Vasek Chvatal. 1979. A greedy heuristic for the set-covering problem. *Math. Op. Res.* 4, 3 (1979), 233–235.

[17] Luc Dehaspe and Hannu Toivonen. 1999. Discovery of frequent datalog patterns. *Data Min. Knowl. Disc.* 3, 1 (1999), 7–36.

[18] Omkar Deshpande, Digvijay S. Lamba, Michel Tourn, Sanjib Das, Sri Subramaniam, Anand Rajaraman, Venky Harinarayan, and AnHai Doan. 2013. Building, maintaining, and using knowledge bases: A report from the trenches. In *Proceedings of the SIGMOD*. 1209–1220.

[19] Xin Luna Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Kevin Murphy, Shaohua Sun, and Wei Zhang. 2014. From data fusion to knowledge fusion. *PVLDB* 7, 10 (2014), 881–892.

[20] Wenfei Fan and Floris Geerts. 2012. *Foundations of Data Quality Management*. Morgan & Claypool Publishers. DOI: https://doi.org/10.2200/S00439ED1V01Y201207DTM030

[21] Wenfei Fan, Chunming Hu, Xueli Liu, and Ping Lu. 2018. Discovering graph functional dependencies. In *Proceedings of the SIGMOD*. 427–439.

[22] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional dependencies for graphs. In *Proceedings of the SIGMOD*. 1843–1857.

[23] Mina H. Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. 2016. CLAMS: Bringing quality to data lakes. In *Proceedings of the SIGMOD*. 2089–2092.

[24] Mohamed H. Gad-Elrab, Daria Stepanova, Jacopo Urbani, and Gerhard Weikum. 2019. ExFaKT: A framework for explaining facts over knowledge graphs and text. In *Proceedings of the WSDM*. 87–95.

[25] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. 2015. Fast rule mining in ontological knowledge bases with AMIE+. *VLDB J.* 24, 6 (2015), 707–730.

[26] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cyber.* 4, 2 (1968), 100–107.

[27] Jian He, Enzo Veltri, Donatello Santoro, Guoliang Li, Giansalvatore Mecca, Paolo Papotti, and Nan Tang. 2016. Interactive and deterministic data cleaning. In *Proceedings of the SIGMOD*. 893–907.

[28] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.* 42, 2 (1999), 100–111.

[29] Viet-Phi Huynh and Paolo Papotti. 2019. Buckle: Evaluating fact checking algorithms built on knowledge bases. *PVLDB* 12, 12 (2019), 1798–1801.

[30] Sebastian Kruse and Felix Naumann. 2018. Efficient discovery of approximate dependencies. *PVLDB* 11, 7 (2018), 759–772.

[31] Bonan Min, Ralph Grishman, Li Wan, Chang Wang, and David Gondek. 2013. Distant supervision for relation extraction with an incomplete knowledge base. In *Proceedings of the HLT-NAACL*. 777–782.

[32] Stephen Muggleton and Luc De Raedt. 1994. Inductive logic programming: Theory and methods. *J. Logic Prog.* 19 (1994), 629–679.

[33] Stefano Ortona, Venkata Vamsikrishna Meduri, and Paolo Papotti. 2018. Robust discovery of positive and negative rules in knowledge bases. In *Proceedings of the ICDE*. 1168–1179.

[34] Stefano Ortona, Venkata Vamsikrishna Meduri, and Paolo Papotti. 2018. RuDiK: Rule discovery in knowledge bases. *PVLDB* 11, 12 (2018), 1946–1949.

[35] Heiko Paulheim. 2017. Knowledge graph refinement: A survey of approaches and evaluation methods. *Sem. Web* 8, 3 (2017), 489–508.

[36] Joeri Rammelaere and Floris Geerts. 2018. Explaining repaired data with CFDs. *PVLDB* 11, 11 (2018), 1387–1399.

[37] Stefan Schoenmackers, Oren Etzioni, Daniel S. Weld, and Jesse Davis. 2010. Learning first-order horn clauses from web text. In *Proceedings of the EMNLP*. 1088–1098.

[38] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. 2015. Incremental knowledge base construction using DeepDive. *PVLDB* 8, 11 (2015), 1310–1321.

[39] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. YAGO: A core of semantic knowledge unifying Wordnet and Wikipedia. In *Proceedings of the WWW*. 697–706.

[40] Fabian M. Suchanek, Mauro Sozio, and Gerhard Weikum. 2009. SOFIE: A self-organizing framework for information extraction. In *Proceedings of the WWW*. 631–640.

[41] Paul Suganthan G. C., Chong Sun, Haojun Zhang, Frank Yang, Narasimhan Rampalli, Shishir Prasad, Esteban Arcaute, Ganesh Krishnan et al. 2015. Why big data industrial systems need rules and what we can do about it. In *Proceedings of the SIGMOD*. 265–276.

[42] Gerald Töpper, Magnus Knuth, and Harald Sack. 2012. DBpedia ontology enrichment for inconsistency detection. In *Proceedings of the I-SEMANTICS*. 33–40.

[43] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: A free collaborative knowledge base. *Commun. ACM* 57, 10 (2014), 78–85.

[44] Dominik Wienand and Heiko Paulheim. 2014. Detecting incorrect numerical data in DBpedia. In *Proceedings of the ESWC*.

Received April 2019; revised September 2019; accepted October 2019