Fast and Exact Rule Mining with AMIE 3

Jonathan Lajus[1(✉)], Luis Galárraga[2], and Fabian Suchanek[1]

[1] Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France
jonathan.lajus@telecom-paris.fr
[2] INRIA Rennes, Rennes, France

Abstract. Given a knowledge base (KB), rule mining finds rules such as "If two people are married, then they live (most likely) in the same place". Due to the exponential search space, rule mining approaches still have difficulties to scale to today's large KBs. In this paper, we present AMIE 3, a system that employs a number of sophisticated pruning strategies and optimizations. This allows the system to mine rules on large KBs in a matter of minutes. Most importantly, we do not have to resort to approximations or sampling, but are able to compute the exact confidence and support of each rule. Our experiments on DBpedia, YAGO, and Wikidata show that AMIE 3 beats the state of the art by a factor of more than 15 in terms of runtime.

1 Introduction

Recent years have seen the rise of large knowledge bases (KBs) such as Wikidata, YAGO, DBpedia, and many others. These are large collections of knowledge about the real world in the form of entities (such as organizations, movies, people, and locations) and relations between them (such as *wasBornIn*, *actesIn*, etc.). Today's KBs contain millions of entities and facts about them. They find applications in Web search, text analysis, and chat bots.

Rule mining is the task of automatically finding logical rules in a given KB. For example, a rule mining approach can find that "If X and Y are married, and X lives in Z, then Y also lives in Z". Such rules usually come with confidence scores that express to what degree a rule holds. The rules can serve several purposes: First, they serve to complete the KB. If we do not know the place of residence of a person, we can propose that the person lives where their spouse lives. Second, they can serve to debug the KB. If the spouse of someone lives in a different city, then this can indicate a problem. Finally, rules are useful in downstream applications such as fact prediction [5,12,14,18], data and ontology alignment [7,10], fact checking [2], and error detection [1].

The difficulty in finding such rules lies in the exponential size of the search space: every relation can potentially be combined with every other relation in a rule. This is why early approaches (such as AMIE [8]) were unable to run on large KBs such as Wikidata in less than a day. Since then, several approaches have resorted to sampling or approximate confidence calculations [4,9,15,19]. The more the approach samples, the faster it becomes, but the less accurate

© Springer Nature Switzerland AG 2020
A. Harth et al. (Eds.): ESWC 2020, LNCS 12123, pp. 36–52, 2020.
https://doi.org/10.1007/978-3-030-49461-2_3

the results will be. Another common technique [13,15,16,19] (from standard inductive logic programming) is to mine not all rules, but only enough rules to cover the positive examples. This, likewise, speeds up the computation, but does not mine all rules that hold in the KB.

In this paper, we present AMIE 3, a successor of AMIE [8] and AMIE+ [9]. Our system employs a number of sophisticated strategies to speed up rule mining: pruning strategies, parallelization, and a lazy computation of confidence scores. This allows our system to scale effortlessly to large KBs. At the same time, the system still computes the exact confidence and support values for each rule, without resorting to approximations. Furthermore, unlike her predecessor [9] and other systems, AMIE 3 exhaustively computes all rules that hold in the KB for a given confidence and support threshold.

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

2 Related Work

First Generation Rule Mining. Inductive Logic Programming (ILP) is the task of learning rules from positive and negative examples. The first of these systems [11,13,16] appeared before the rise of large KBs. Hence, they are generally unsuitable for today's KBs for two reasons: (i) they were not designed to scale to millions of facts, and (ii) they do not account for the Open World Assumption (OWA) made by current KBs. For example, FOIL [16] (as well as its optimized successor [19]) cannot be applied directly to KBs because it assumes the user can provide explicit counter-examples for the rules. Alas, KBs do not store negative statements. In contrast, WARMR [11] generates negative evidence by assuming the KB is complete, i.e., by making a closed world assumption (CWA), whereas [13] uses a positives-only learning function that generates negative evidence from random facts (a similar, but more systematic mechanism is proposed in [15]). It was shown [8] that these strategies work less well on KBs than the partial completeness assumption (PCA), which was explicitly designed for KBs.

Second Generation Rule Mining. AMIE (and its successor AMIE+) [8,9] was the first approach to explicitly target large KBs. While AMIE+ is at least 3 orders of magnitude faster than the first-generation systems, it can still take hours, even days, to find rules in very large KBs such as Wikidata. On these grounds, more recent approaches [3,4,15] have proposed new strategies (parallelism, approximations, etc.) to speed up rule mining on the largest KBs. The Ontological Pathfinding method (OP) [3,4] resorts to a highly concurrent architecture based on Spark[1] to calculate the support and the confidence of a set of candidate rules. The candidates are computed by enumerating all conjunctions of atoms that are allowed by the schema. Like AMIE, OP calculates

[1] https://spark.apache.org.

the exact scores of the rules and supports both the CWA and the PCA for the generation of counter-evidence. At the same time, the system supports only path rules of up to 3 atoms. Other types of rules require the user to implement a new mining procedure. We will see in our experiments that AMIE 3 is both more general and faster than OP.

RudiK [15] is a recent rule mining method that applies the PCA to generate explicit counter-examples that are semantically related. For example, when generating counter-facts for the relation *hasChild* and a given person x, RudiK will sample among the non-children of x who are children of someone else ($x' \neq x$). Rudik's strategy is to find all rules that are necessary to predict the positive examples, based on a greedy heuristic that at each step adds the most promising rule (in terms of coverage of the examples) to the output set. Thus, differently from exhaustive rule mining approaches [3,8,9,11], Rudik aims to find rules that make good predictions, not all rules above a given confidence threshold. This non-exhaustivity endows RudiK with comparable performance to AMIE+ and OP. Nevertheless, we show that AMIE 3 outperforms RudiK in terms of runtime while still being exhaustive.

3 Preliminaries

Knowledge Bases. We assume a set $\mathcal{I}$ of entities (such as *Paris*), a set $\mathcal{P}$ of binary relations (such as *locatedIn*), and a set $\mathcal{L}$ of literal values (strings or numbers)[2]. We model a knowledge base (KB) $\mathcal{K}$ as a set of assertions $r(s, o)$, also called *facts*, with a subject $s \in \mathcal{I}$, a relation $r \in \mathcal{P}$ and an object $o \in \mathcal{I} \cup \mathcal{L}$. An example of a fact is *locatedIn*(*Paris*, *France*). Whenever $\mathcal{K}$ is clear from the context, we write $r(s, o)$ to mean $r(s, o) \in \mathcal{K}$.

Relations and Functions. The *inverse* of a relation r, denoted r^-, is the relation consisting of all the facts of the form $r^-(o, s)$ such that $r(s, o) \in \mathcal{K}$. A relation r is a *function* in $\mathcal{K}$, if r has at most one object for each subject. Some relations (e.g., *isCitizenOf*) are *quasi-functions*, i.e. they rarely associate multiple objects to a given subject. Hence, the notion of functions has been generalized to the *functionality score* [17] of a relation r:

$$fun(r) = \frac{|\{s : \exists o : r(s, o) \in \mathcal{K}\}|}{|\{(s, o) : r(s, o) \in \mathcal{K}\}|} \tag{1}$$

The functionality score is always between 0 and 1 (incl.). It is exactly 1 for strict functions such as *hasBirthPlace*, it is close to 1 for quasi-functions, and it is smaller for relations that have many objects (such as *actedInMovie*).

Atoms and Rules. An *atom* is an expression of the form $r(X, Y)$, where r is a relation and X, Y are either constants or variables. From now on, we denote variables by lowercase letters, whereas constants (entities) are always capitalized. An atom is *instantiated* if at least one of its arguments is a constant, as in

[2] In line with the other works [4,8,9,15], we do not consider blank nodes.

livesIn$(x, Berlin)$. If both arguments are constants, the atom is *grounded* and it is tantamount to a fact. We define the operator $var(A)$ so that it returns the set of variables of an atom A. A (conjunctive) *query* is a conjunction of atoms: $B_1 \wedge ... \wedge B_n$. A *substitution* σ is a partial mapping from variables to constants. Substitutions can be straightforwardly extended to atoms and conjunctions. A *result* of a query $B_1 \wedge ... \wedge B_n$ on a KB $\mathcal{K}$ is a substitution σ that (i) maps all variables and (ii) that entails $\sigma(B_i) \in \mathcal{K} \ \forall i \in \{1, ..., n\}$.

A (Horn) *rule* is a formula of the form $\boldsymbol{B} \Rightarrow H$, where the $\boldsymbol{B}$ is a query of *body atoms* $B_1, ..., B_n$, and H is the *head atom*. Two atoms A, A' are *connected* if $var(A) \cap var(A') \neq \emptyset$, i.e., they have common variables. It is common [4,8,9,15] to impose that all atoms in a rule are transitively connected and that rules are closed. A rule is *closed* if all variables appear in at least two atoms. A closed rule is always *safe*, i.e. all head variables appear also in at least one body atom.

Predictions. Given a rule $R = B_1 \wedge ... \wedge B_n \Rightarrow H$ and a substitution σ, we call $\sigma(R)$ an *instantiation* of R. If $\sigma(B_i) \in \mathcal{K} \ \forall i \in \{1, ..., n\}$, we call $\sigma(H)$ a *prediction* of R from $\mathcal{K}$, and we write $\mathcal{K} \wedge R \models \sigma(H)$. If $\sigma(H) \in \mathcal{K}$, we call $\sigma(H)$ a *true prediction*.

A *false prediction* of a rule is a prediction of a counter-example of the rule. There are different approaches to define these counter-examples: Under the *Closed World Assumption* (CWA), any assertion that is not in the KB is considered a counter-example. However, KBs are usually incomplete, and thus the CWA penalizes rules that predict new facts. Under the *Open World Assumption* (OWA), facts that are not in the KB are not necessarily wrong, and hence there are no counter-examples. This entails that a rule mining algorithm will report arbitrary rules as long as these rules make enough true predictions (such as "All people play the violin"). Therefore, AMIE [8] has proposed the *Partial Completeness Assumption* (PCA): If we have $r(s, o)$ in the KB $\mathcal{K}$, and if $fun(r) \geq fun(r^-)$, then we assume that all $r(s, o') \notin \mathcal{K}$ do not hold in the real world. If $fun(r) < fun(r^-)$, then the PCA says that all $r(s', o) \notin \mathcal{K}$ do not hold in the real world. These assertions can thus serve as counter-examples. There are a number of other approaches to generate counter-examples in the literature [18].

Support and Confidence. The *support* of a rule R in a KB $\mathcal{K}$ is the number of true predictions p (of the form $r(X, Y)$) that the rule makes in the KB:

$$support(R) = |\{p : (\mathcal{K} \wedge R \models p) \ \wedge \ p \in \mathcal{K}\}| \tag{2}$$

The *head-coverage* is the proportional variant of the support: It is the ratio of instantiations of the head atom that are predicted by the rule:

$$hc(\boldsymbol{B} \Rightarrow r(x, y)) = \frac{support(\boldsymbol{B} \Rightarrow r(x, y))}{|\{(x, y) : r(x, y) \in \mathcal{K}\}|}$$

The *confidence* of a rule R in a KB $\mathcal{K}$ is the proportion of true predictions out of the true predictions and false predictions:

$$confidence(R) = \frac{support(R)}{support(R) + |\{p : (\mathcal{K} \wedge R \models p) \wedge p \in cex(R)\}|} \tag{3}$$

Here, $cex(R)$ denotes the set of counter-examples of R. If the counter-examples are chosen by the PCA, we refer to the confidence as the *PCA confidence* and denote it by *pca-conf* (analogously for the CWA).

In general, the support of the rule quantifies its relevance, whereas the confidence quantifies its accuracy. *Rule mining* is the task of finding all rules in a KB that fulfill certain confidence and support thresholds. It is a relaxation of inductive logic programming (ILP), in the sense that it finds also rules that predict some limited number of counter-examples (see [18] for a discussion).

4 AMIE 3

In this section, we first recap the original AMIE algorithm [8] (Sect. 4.1). Then we present a series of optimizations that give rise to AMIE 3 (Sect. 4.2). Finally, we show different quality metrics that AMIE 3 can compute (Sect. 4.3).

4.1 The AMIE Approach

The AMIE algorithm [8,9] is a method to mine closed Horn rules on large KBs. AMIE (Algorithm 1) takes as input a knowledge base $\mathcal{K}$, and thresholds l for the maximal number of atoms per rule, *minHC* for the minimum head coverage, and *minC* for the minimum PCA confidence. AMIE uses a classical breadth-first search: Line 1 initializes a queue with all possible rules of size 1, i.e., rules with an empty body. The search strategy then dequeues a rule R at a time and adds it to the output list (Line 6) if it meets certain criteria (Line 5), namely, (i) the rule is closed, (ii) its PCA confidence is higher than *minC*, and (iii) its PCA confidence is higher than the confidence of all previously mined rules with the same head atom as R and a subset of its body atoms. If the rule R has less than l atoms and its confidence can still be improved (Line 7), AMIE refines it. The refinement operator *refine* (Line 8) derives new rules from R by considering all possible atoms that can be added to the body of the rule, and creating one new rule for each of them.

AMIE iterates over all the non-duplicate refinements of rule R and adds those with enough head coverage (Lines 10–11). The routine finishes when the queue runs out of rules. The AMIE algorithm has been implemented in Java with multi-threading. By default, AMIE sets $minHC = 0.01$, $minC = 0.1$, and $l = 3$. AMIE+ [9] optimized this algorithm by a number of pruning strategies, but did not change the main procedure.

4.2 AMIE 3

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

Algorithm 1: AMIE

Input: a KB: $\mathcal{K}$, maximum rule length: l, head coverage threshold: $minHC$, confidence threshold: $minC$
Output: set of Horn rules: *rules*

```
1  q = [⊤ ⇒ r1(x,y), ⊤ ⇒ r2(x,y) ... ⊤ ⇒ rm(x,y)]
2  rules = ⟨⟩
3  while |q| > 0 do
4      R = q.dequeue()
5      if closed(R) ∧ pca-conf(R) ≥ minC ∧ betterThanParents(R, rules) then
6          rules.add(r)
7      if length(R) < l ∧ pca-conf(Rc) < 1.0 then
8          for each rule Rc ∈ refine(R) do
9              if hc(Rc) ≥ minHC ∧ Rc ∉ q then
10                 q.enqueue(rc)
11 return rules
```

Existential Variable Detection. In order to decide whether to output a rule, AMIE has to compute its confidence (Lines 5 and 7 of Algorithm 1), i.e., it has to evaluate Eq. 3. If the PCA confidence is used, this equation becomes:

$$pca\text{-}conf(\boldsymbol{B} \Rightarrow r(x,y)) = \frac{support(\boldsymbol{B} \Rightarrow r(x,y))}{|\{(x,y) : \exists y' : \boldsymbol{B} \wedge r(x,y')\}|}. \quad (4)$$

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

This method can be improved as follows: Assume that our rule is simply $r_1(x,z) \wedge r_2(z,y) \Rightarrow r_h(x,y)$. Then AMIE will compute the number of distinct pairs (x,y) for the following query (the denominator of Eq. 4):

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

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

$$r_1(x, Z) \wedge r_2(Z, Y) \wedge r_h(x, y') \tag{5}$$

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

Algorithm 2: DistinctValues

Input: variable x, query $q = A_1 \wedge ... \wedge A_n$, KB $\mathcal{K}$,
Output: set of values $\mathcal{X}$

1 $\mathcal{X} := \emptyset$
2 $A^* := argmin_A(|\{(x, y) : A = r(x, y), A \in q\}|)$
3 **if** *x appears in* A^* **then**
4 **return** $\{x : x \in \sigma(A^*) \wedge \sigma(q \setminus A^*)$ is satisfiable$\}$
5 **for** ***each*** $\sigma : \sigma(A^*) \in \mathcal{K}$ **do**
6 $\mathcal{X} := \mathcal{X} \cup DistinctValues(x, \sigma(q \setminus A^*), \mathcal{K})$
7 **return** $\mathcal{X}$

Lazy Evaluation. The calculation of the denominator of Eq. 4 can be computationally expensive, most notably for "bad" rules such as:

$$R : directed(x, z) \wedge hasActor(z, y) \Rightarrow marriedTo(x, y). \tag{6}$$

In such cases, AMIE spends a lot of time computing the exact confidence, only to find that the rule will be pruned away by the confidence threshold. This can be improved as follows: Instead of computing first the set of values for x, and then for each value of x the possible values of y, we compute for each value of x directly the possible values of y—and only then consider the next value of x. Following the principle "If you know something is bad, do not spend time to figure out how bad exactly it is", we stop this computation as soon as the set size reaches the value $support(R) \times minC^{-1}$. If this occurs, we know that $pca\text{-}conf(R) < minC$, and hence the rule will be pruned in Line 5 of Algorithm 1.

Variable Order. To compute the PCA confidence (Eq. 4), we have to count the instantiations of pairs of variables x, y. AMIE counts these asymmetrically: It finds the values of x and then, for each value of x, the values of y. We could as well choose to start with y instead. The number of pairs is the same, but we found that the choice impacts the runtime: Once one variable is fixed, the computation of the other variable happens on a rule that has fewer degrees of freedom than the original rule, i.e., it has fewer instantiations. Thus, one has an interest in fixing first the variable that appears in as many selective atoms as possible. Alas, it is very intricate to determine which variable restricts more efficiently the set of instantations, because the variables appear in several atoms, and each

instantiation of the first variable may entail a different number of instantiations of the second variable. Therefore, estimating the exact complexity is unpractical.

We use the following heuristic: Between x and y, we choose to start with the variable that appears in the head atom of the rule in the denominator of Eq. 4. The reason is that this variable appears in at least two atoms already, whereas the other variable appears only in at least one atom. We show in our experiments that this method improves the runtime by several orders of magnitude for some rules.

Parallel Computation for Overlap Tables. AMIE implements an approximation of Eq. 4. This approximation misses only a small percentage of rules (maximally 5% according to [9]), but speeds up the calculation drastically. In AMIE 3, this feature can be switched off (to have exact results) or on (to have faster results). Here, we show how to further speed up this heuristic. The method finds an efficient approximation of the denominator of Eq. 4 for a rule R. This approximation uses the join structure of the query in combination with the functionality scores and the overlaps of the different relations to estimate the total number of examples (both positive and negative) of a rule. The exact formula and the rationale behind it can be found in [9]. The functionality, domain and overlaps with other relations are pre-computed for all relations. This pre-calculation can be significant for large KBs with many predicates. In our experiments with DBpedia, e.g., precomputing all overlaps takes twice as much time as the mining. In AMIE 3, we exploit the fact that this task is easy parallelizable, and start as many threads as possible in parallel, each treating one pair of relations. This reduces the precomputation time linearly with the number of threads (by a factor of 40 in our experiments).

Integer-Based In-Memory Database. AMIE uses an in-memory database to store the entire KB. Each fact is indexed by subject, by object, by relation, and by pairs of relation/subject and relation/object. In order to be able to load also large KBs into memory, AMIE compresses strings into custom-made *ByteStrings*, where each character takes only 8 bits. AMIE makes sure that ByteString variables holding equivalent ByteStrings point to the same physical object (i.e., the ByteString exists only once). This not just saves space, but also makes hashing and equality tests trivial. Still, we incur high costs of managing these objects and the indexes: ByteStrings have to be first created, and then checked for duplicity; unused ByteStrings have to be garbage-collected; equality checks still require casting checks; and HashMaps create a large memory overhead. Built-in strings suffer from the same problems. Therefore, we migrated the in-memory database to an integer-based system, where entities and relations are mapped to an integer space and represented by the primitive datatype *int*. This is in compliance with most RDF engines and popular serialization formats such as [6]. We use the fastutil library[3] to store the indexes. This avoids the overhead of standard HashMaps. It also reduces the number of objects that the garbage collector has to treat, leading to a significant speedup.

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

4.3 Quality Metrics

AMIE is a generic exhaustive rule miner, and thus its output consists of *rules*. These rules can serve as input to other applications, for example, to approaches that predict *facts* [5,15]. Such downstream applications may require different quality metrics. These can be implemented on top of AMIE, as shown here:

Support and head coverage. Support is a standard quality metric that indicates the significance of a rule. Due to the anti-monotonicity property, most approaches use support to prune the search space of rules. AMIE [8,9] uses by default the head coverage (the relative variant of support) for pruning.
PCA Confidence. By default, AMIE uses the PCA confidence to assess the quality of a rule, because it has been shown to rank rules closer to the quality of their predictions than classical metrics such as the CWA confidence [8].
CWA confidence. This confidence is used in OP [3,4]. Many link prediction methods are evaluated under the closed world assumption as well [18].
GPRO confidence. The work of [5] noted that the PCA confidence can underestimate the likelihood of a prediction in the presence of non-injective mappings. Therefore, the authors propose a refinement of the PCA confidence, the GPRO confidence, which excludes instances coming from non-injective mappings in the confidence computation. To judge the quality of a predicted fact, the approach needs the GPRO confidence both on the first and second variable of the head atom. AMIE is not designed to judge the quality of a predicted fact, but can compute the GPRO confidence on both variables.
GRANK confidence. This refinement of the GPRO metric is proposed by [5] in order to take into account the number of instances of the variables of the rule that are not in the head atom.

These metrics are implemented in AMIE 3 and can be enabled by command line switches.

5 Experiments

We conducted two series of experiments to evaluate AMIE 3: In the first series we study the impact of our optimizations on the system's runtime. In the second series, we compare AMIE 3 with two scalable state-of-the-art approaches, namely RudiK [15] and Ontological Pathfinding (OP) [3,4] (also known as ScaleKB) on 6 different datasets.

5.1 Experimental Setup

Data. We evaluated AMIE 3 and its competitors on YAGO (2 and 2s), DBpedia (2.0 and 3.8) and a dump of Wikipedia from December 2014. These datasets were used in evaluations of AMIE+ [9], OP [3] and Rudik [15]. In addition, we used

Table 1. Experimental datasets

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

Table 2. Old ByteString database vs. the new integer-based database.

Dataset	Loading time	Wall time		Memory used	
		Integer	ByteString	Integer	ByteString
Yago2	7 s	26.40 s	29.69 s	6Go	9Go
Yago2s	45 s	1 min 55 s	4 min 10 s	16Go	19Go
DBpedia 2.0	55 s	7 min 32 s	34 min 06 s	29Go	32Go
DBpedia 3.8	1 min 20 s	7 min 49 s	52 min 10 s	40Go	42Go
Wikidata 2014	59 s	5 min 44 s	6 min 01 s	27Go	54Go

a recent dump of Wikidata from July 1st, 2019[4]. Table 1 shows the numbers of facts, relations, and entities of our experimental datasets.

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

Unless otherwise noted, the experiments were run using the default settings of AMIE: We used the PCA confidence, computed lazily with a threshold of 0.1, with all the lossless optimizations (no approximations). The threshold on the head coverage is 0.01 and the maximal rule length is 3 [8].

5.2 Effect of Our Optimizations

In-Memory Database. Table 2 shows the performance with the new integer-based in-memory database and the old ByteString database. The change reduces the memory footprint by around 3 GB in most cases, and by 50% in Wikidata. Moreover, the new database is consistently faster, up to 8-fold for the larger KBs such as DBpedia 3.8.

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

Table 3. Impact of laziness and of switching on the confidence approximation. *Ov. tables* is the time needed to compute the overlap tables.

Dataset	Conf. Approx. off		Confidence approximation on		
	Non-lazy	Lazy	Non-lazy	Lazy	Ov. tables
Yago2	24.12 s	26.40 s	24.39 s	21.41 s	0.2 s
Yago2s	4 min 28 s	1 min 55 s	1 min 42 s	2 min 03 s	2.4 s
DBpedia 2.0	10 min 14 s	7 min 32 s	7 min 42 s	8 min 13 s	23.5 s
DBpedia 3.8	14 min 50 s	7 min 49 s	11 min 07 s	10 min 18 s	15.2 s
Wikidata 2014	19 min 27 s	5 min 44 s	5 min 45 s	4 min 36 s	12 s
Wikidata 2019	>48 h	16h 43 min	17h 06 min	16h 31 min	41.4 s

Laziness. As explained in Sect. 4.2, AMIE can invest a lot of time in calculating the PCA confidence of low-confident rules. The lazy evaluation targets exactly this problem. Table 3 shows that this strategy can reduce the runtime by a factor of 4. We also show the impact of laziness when the PCA confidence approximation is switched on. We observe that the parallel calculation of the overlap tables reduces drastically the contribution of this phase to the total runtime when compared to AMIE+—where it could take longer than the mining itself. We also note that the residual impact of the confidence approximation is small, so that this feature is now dispensable: We can mine rules exhaustively.

Count Variable Order. To measure the impact of the count variable order, we ran AMIE 3 (with the lazy evaluation activated) on Yago2s and looked at the runtimes when counting with the variable that appears in the head atom versus the runtime when counting with the other variable. For every rule with three atoms and a support superior to 100, we timed the computation of the PCA confidence denominator (Eq. 4) in each case. The y-axis of Fig. 1 shows the runtime when we first instantiate the variable that occurs in the head atom, whereas the x-axis shows the runtime when using the other variable.

We see that every query can be run in under 10 s and that most of the queries would run equally fast independently of the order of the variables. However, for some rules, instantiating first the variable that does not appear in the head atom can be worse than the contrary by several orders of magnitude. Some queries would take hours (days in one case) to compute, even with lazy evaluation. In Yago2s, these rules happen to be pruned away by the AMIE+ confidence upper bound (a lossless optimization), but this may not be the case for all KBs. The problematic rules all have bodies of the following shape:

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

Both *hasGender* and *isLocatedIn* are very large relations as they apply to any person and location, respectively. While early pruning of those "hard rules" is the purpose of the confidence approximations and upper bounds of AMIE+, these strategies may fail in a few cases, leading to the execution of expensive

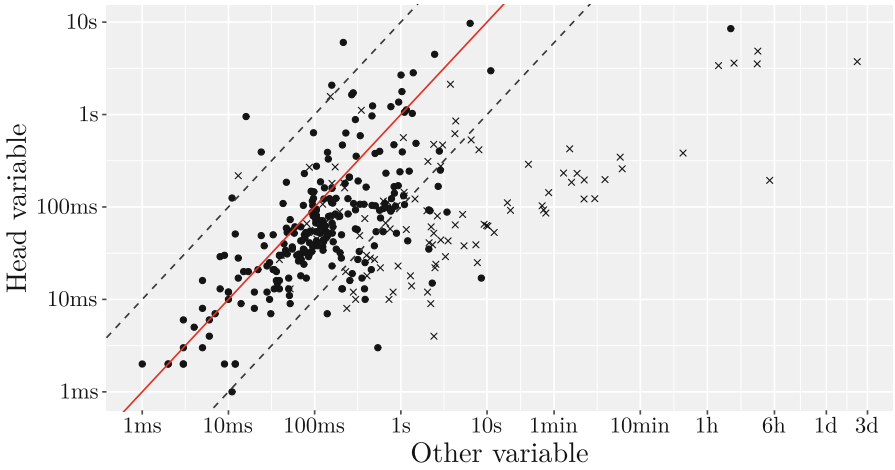

Fig. 1. Impact of the variable order on Yago2s. Each point is a rule. Cross points: pruned by the confidence approximation. Red line: same performance. Dashed lines: relative speedup of 10×.

queries. Finally, we show the overall impact of the count variable order heuristic in Table 4. The results suggest that our heuristic generally yields lower runtimes.

Impact of Existential Variable Detection. Last but not least, the on-the-fly detection of existential variables reduces the number of recursive calls made to Algorithm 2. Table 5a shows the performances of AMIE 3 with and without this optimization. This optimization is critical for AMIE 3 on most datasets. This is less important for DBpedia 2.0 as it contains mostly small relations.

Metrics. Table 5b shows the impact of different quality metrics on the runtime, with iPCA being the PCA with injective mappings. The metrics run slower than the PCA confidence, because we cannot use the PCA upper bound optimization. The GRank metric, in particular, is very sensitive to the number of facts per relation, which explains its performance on Yago2s and DBpedia 3.8. For all other metrics, however, the numbers are very reasonable.

5.3 Comparative Experiments

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

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

Table 4. Impact of the variable order: variable that appears in the head atom (new AMIE 3 heuristic); variable that does not appear in the head atom; variable that appears first in the head atom of the original rule (old AMIE method).

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

Table 5. Performance with different features.

(a) Existential variable detection (ED)

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

(b) Different metrics (Sect. 4.3)

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

RuDiK. We set the number of positive and negative examples to 500, as advised on the project's github page[5]. We tried to run the system in parallel for different head relations. However, the graph generation phase of the algorithm already runs in parallel and executes a lot of very selective SPARQL queries in parallel. Hence, the additional parallelization flooded the SPARQL endpoint, which rejected any new connection at some point. For this reason, we mined the rules for every possible relation sequentially, using only the original parallelization mechanism. RuDiK also benefits from information on the taxonomic types of the variables. While the built-in method to detect the types of the relations works out-of-the-box for DBpedia (which has a flat taxonomy), it overgeneralizes on the other datasets, inverting the expected benefits. Therefore, we ran RuDiK without the type information on the other datasets.

Ontological Pathfinding. This system first builds a list of candidate rules (Part 5.1 of [4]). Unfortunately, the implementation of this phase of the algorithm is not publicly available. Hence, we had to generate candidate rules ourselves. The goal is to create all rules that are "reasonable", i.e., to avoid rules with empty joins such as $birthPlace(x, y) \wedge hasCapital(x, z)$. The original algorithm discards all rules where the domain and range of joining relations do not match. However, it does not take into account the fact that an entity can be an instance of multiple classes. Thus, if the domain of *actedIn* is *Actor*, and the domain of *directed* is *Director*, the original algorithm would discard any rule that contains

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

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

Table 6. Performances and output of Ontological Pathfinding (OP), RuDiK and AMIE 3. *: rules with support ≥ 100 and CWA confidence ≥ 0.1.

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

Results

It is not easy to compare the performance of OP, AMIE 3, and Rudik, because the systems serve different purposes, have different prerequisites, and mine different rules. Therefore, we ran all systems in their default configurations, and discuss the results (Table 6) qualitatively in detail.

Ontological Pathfinding. We ran OP both with a domain-based candidate generation (which finds fewer rules) and with our candidate generation. In general, OP has the longest running times, but the largest number of rules. This is inherent to the approach: OP will prune candidate rules using a heuristic [3] that is similar to the confidence approximation of AMIE+. After this step, it will compute the support and the exact CWA confidence of any remaining candidate. However, it offers no way of pruning rules upfront by support and confidence. This has two effects: First, the vast majority (>90%) of rules found by OP have very low confidence (<10%) or very low support (<100). Second, most of the time will be spent computing the confidence of these low-confidence rules, because the exact confidence is harder to compute for a rule with low confidence.

To reproduce the result of OP with AMIE, we ran AMIE 3 with a support threshold of 100 and a CWA confidence threshold of 10%. This reproduces the

rules of OP (and 8 more because AMIE does not use the OP functionality heuristics) in less than two minutes. If we set our support threshold to 1, and our minimal CWA confidence to 10^{-5}, then we mine more rules than OP on Yago2s (as shown in Table 6) in less time (factor 25×). If we mine rules with AMIE's default parameters, we mine rules in less than two minutes (factor 90×).

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

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

In our experiment, RuDiK mined rules in Wikidata in 23 h. However, RuDiK was not able to mine rules for 22 of the relations as Virtuoso was not able to compute any of the positive or the negative examples RuDiK requires to operate. This is because RuDiK would timeout any SPARQL query after 20 s of execution[6]. Virtuoso failed to compute the examples during this time frame on the 22 relations, which are the largest ones in our Wikidata dataset: They cover 84% of the facts. Interestingly, RuDiK did also not find rules that contain these relations in the body (except one, which covered 0.5% of the KB).

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

AMIE 3 outperformed both OP and RuDiK in terms of runtime and the number of rules. Moreover, it has the advantage of being exact and complete. Then again,

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

the comparisons have to be seen in context: RuDiK, e.g., is designed to run on a small machine. For this, it uses a disk-based database and sampling. AMIE, in contrast, loads all data into memory, and thus has a large memory footprint (the 500 GB were nearly used up for the Wikidata experiment). In return, it computes all rules exactly and is fast.

6 Conclusion

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

Acknowledgements. Partially supported by the grant ANR-16-CE23-0007-01.

References

1. Ahmadi, N., Huynh, V.P., Meduri, V., Ortona, S., Papotti, P.: Mining expressive rules in knowledge graphs. JDIQ **12**(2), 1–27 (2019)
2. Ahmadi, N., Lee, J., Papotti, P., Saeed, M.: Explainable fact checking with probabilistic answer set programming. In: Conference for Truth and Trust online (2019)
3. Chen, Y., Goldberg, S., Wang, D.Z., Johri, S.S.: Ontological pathfinding. In: SIGMOD (2016)
4. Chen, Y., Wang, D.Z., Goldberg, S.: ScaLeKB: scalable learning and inference over large knowledge bases. VLDB J. **25**(6), 893–918 (2016). https://doi.org/10.1007/s00778-016-0444-3
5. Ebisu, T., Ichise, R.: Graph Pattern Entity Ranking Model for Knowledge Graph Completion. In: NAACL-HLT (2019)
6. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF Representation (HDT). Web Semant. **19**, 22–41 (2013)
7. Galárraga, L., Heitz, G., Murphy, K., Suchanek, F.: Canonicalizing open knowledge bases. In: CIKM (2014)
8. Galárraga, L., Teflioudi, C., Hose, K., Suchanek, F.: AMIE: association rule mining under incomplete evidence in ontological knowledge bases. In: WWW (2013)
9. Galárraga, L., Teflioudi, C., Hose, K., Suchanek, F.M.: Fast rule mining in ontological knowledge bases with AMIE+. VLDB J. **24**(6), 707–730 (2015). https://doi.org/10.1007/s00778-015-0394-1
10. Galárraga, L.A., Preda, N., Suchanek, F.M.: Mining rules to align knowledge bases. In: AKBC (2013)
11. Hand, D.J., Adams, N.M., Bolton, R.J. (eds.): Pattern Detection and Discovery. LNCS (LNAI), vol. 2447. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45728-3
12. Meng, C., Cheng, R., Maniu, S., Senellart, P., Zhang, W.: Discovering meta-paths in large heterogeneous information networks. In: WWW (2015)

13. Muggleton, S.: Learning from positive data. In: ILP (1997)
14. Niu, F., Ré, C., Doan, A., Shavlik, J.: Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. arXiv:1104.3216 (2011)
15. Ortona, S., Meduri, V.V., Papotti, P.: Robust discovery of positive and negative rules in knowledge bases. In: ICDE (2018)
16. Quinlan, J.R.: Learning logical definitions from relations. Machine Learning **5**(3), 239–266 (1990). https://doi.org/10.1007/BF00117105
17. Suchanek, F.M., Abiteboul, S., Senellart, P.: PARIS: probabilistic alignment of relations, instances, and schema. In: PVLDB, vol. 5, no. 3 (2011)
18. Suchanek, F.M., Lajus, J., Boschin, A., Weikum, G.: Knowledge representation and rule mining in entity-centric KBs. In: Reasoning Web Summer School (2019)
19. Zeng, Q., Patel, J.M., Page, D.: QuickFOIL: scalable inductive logic programming. In: VLDB, vol. 8, no.3, November 2014