Computing Property Paths over Linked Data Using AI Search

Jorge Baier, Dietrich Daroch, Juan L. Reutter, and Domagoj Vrgoč

PUC Chile and Center for Semantic Web Research

Abstract. The evaluation of SPARQL queries over the Web of Linked Data poses significant challenges not seen when querying local RDF datasets. This is particularly evident when processing *property paths*, which are arguably one of the more important SPARQL features in the context of Linked Data. Although semantics for property paths over linked data have emerged, algorithms are currently lacking. In this paper we show that the evaluation of a property path query can be reduced to finding the solution of a single-agent deterministic search task, a problem which has been studied by the Artificial Intelligence (AI) community for decades. By formulating the problem in terms of automata we are able to define a variant of the A* search algorithm which can be used to compute answers to property path queries over the Web of Linked Data.

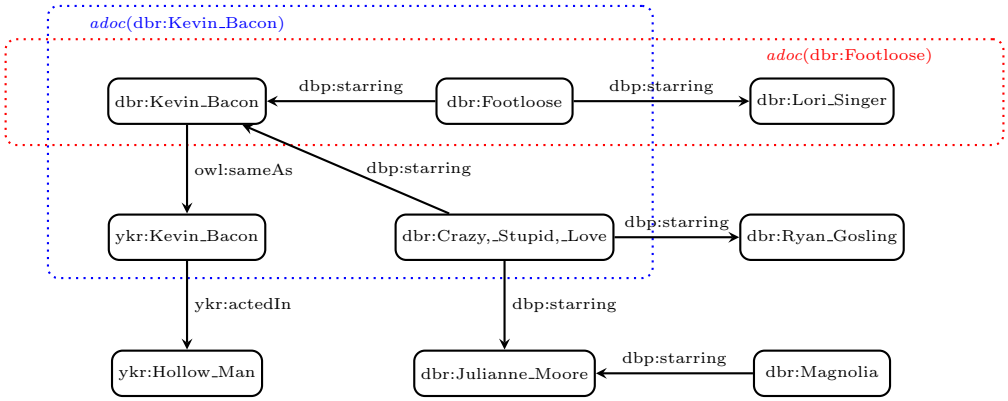
1 Introduction

The *Web of Linked Data* comprises a wide variety of datasets that have been published under a set of best practices and standards that aim to improve the interconnection of these datasets and allow computers to search for information the way humans do it with webpages (see e.g. [5]). The adoption of the linked data standard and the creation of this new web has brought up several challenges, one of the most important being how to query the Web of Linked Data.

From an algorithmic perspective, we define this problem as follows. We assume that data in the Web is stored as RDF graphs, and that every IRI in these graphs is *dereferenceable*, which in our context means that from any given IRI u one should be able to obtain the set $adoc(u)$ of hopefully all RDF triples in the Web that mention u, or at least some of them. The main challenge is the sheer size of the Web: it is not feasible to pose queries as if it was a single database. Instead, the idea is to leverage dereferencing to navigate the Web of Linked Data, extracting only the information that is needed to process a given query.

Example 1. In order to retrieve actors with a finite *Bacon* number [1], we start with the simpler task of computing all actors that have acted in a movie with Kevin Bacon. We begin by searching DBPedia's IRI for Kevin Bacon, which corresponds to `dbr:Kevin_Bacon` (we omit prefixes for readability). We dereference

[1] Actors have Bacon number 1 if they have acted in a movie with Kevin Bacon, and Bacon number n if they have acted together with an actor with Bacon number $n-1$.



(a) Graph Representation of RDF triples

dbr:Kevin_Bacon	*owl:sameAs*	*ykr:Kevin_Bacon*
dbr:Footloose	*dbp:starring*	*dbr:Kevin_Bacon*
dbr:Crazy,_Stupid_Love	*dbp:starring*	*dbr:Kevin_Bacon*

(b) RDF document corresponding to *adoc*(dbr:Kevin_Bacon)

dbr:Footloose	*dbp:starring*	*dbr:Kevin_Bacon*
dbr:Footloose	*dbp:starring*	*dbr:Lori_Singer*

(c) Document corresponding to *adoc*(dbr:Footloose)

Fig. 1. A portion of the Web of Linked Data, depicting triples from DBPedia and Yago

this IRI to obtain the RDF document *adoc*(`dbr:Kevin_Bacon`) shown in Figure 1. This document, in turn, contains a number of movies (we know an IRI I is a movie when it is part of a triple of form I `dbp:starring dbr:Kevin_Bacon`). To finish our query we just need to dereference each of these movies, since they will contain information about all the actors starring in them. For example, if we dereference the IRI for Footloose we obtain the document *adoc*(`dbr:Footloose`), which contains the information that Lori Singer also starred in Footloose.

Readers may note that in the particular example above we could have queried the DBPedia endpoint to obtain this information. There are, however, advantages in focusing on the more general problem of querying Linked Data. First, knowing how to query the Web of Linked Data allows us to extract information even when a suitable SPARQL endpoint does not exists, or just gives partial information. Second, the Linked Data approach gives us a way to query the information without knowing the structure beforehand, but rather obtaining the structure as we navigate the Web. And Finally, as we show in this paper, abstracting to the more general Linked Data approach may sometimes even allow us to propose algorithms that are better suited to process some SPARQL queries, even in the context of a local database.

In order to express queries, the recommendation is to use SPARQL, the default language for querying RDF datasets. Unfortunately, the official semantics

of SPARQL assumes we are dealing with a single dataset, and there is still no standard semantics for SPARQL queries over the Web of Linked Data. The main problem is that the open-world nature of the Web does not couple well with some fragments of SPARQL, as the answers for some queries may be invalidated when dereferencing additional tuples. For this reason, most previous work has focused on simple, monotonic fragments of SPARQL (see e.g. [6, 14, 13]).

In this paper we focus instead on Property Paths, a navigational feature of SPARQL 1.1 [11] that allow users to traverse through RDF documents, navigating between two different entities. Given that traversing through nodes (by dereferencing) is the most basic operation in linked data, it is conceivable that property paths should have a major role when posing SPARQL queries over this infrastructure. Unfortunately, this topic has remained mostly unexplored. There has been some work about navigation within Linked Data (see e.g. [15, 9, 8]), but the first proposal for semantics was published less than a year ago [16], and it is still not clear how to actually compute the answers to property paths in the linked data context. To understand the challenges we face, consider the following property path pattern, designed to compute all actors within DBPedia [3] with a finite Bacon number.

```
{ dbr:Kevin_Bacon  (^dbo:starring/dbo:starring)+ ?x }
```

But how is one supposed to process this query? Recall that the structure of the Web is not known, but must be learnt as we dereference new IRIs. If we think of the Web of Linked Data as a huge graph, then a natural starting point for algorithms to consider are Depth-first search (DFS) or Breadth-first search (BFS). The DFS approach is clearly not the best option here, as the query above allows for paths of arbitrary length and therefore defining when to stop the depth search is not clear. BFS seemed to work with small graphs such as the one from Figure 1. But plain BFS is not the best alternative either (and especially if we are only interested in just a few answers): with BFS one would need to retrieve all of Kevin Bacon's movies before retrieving any answers. This algorithmic issue becomes more and more involved when we deal with increasingly complex property paths.

Our proposal is to specify the problem of evaluating property paths as a search problem, and solve it using a variation of the classical A* algorithm. More precisely, the following are our contributions.

- We show that answering property path queries over the Web of Linked Data can be cast as what is known as a *search* problem, which are normally solved with A* variants. This leads to the proposal of the first algorithm (up to our best knowledge) for computing answers of property paths over the Web of Linked data, that is also in line with previously proposed semantics [16].
- We show that this algorithm can be naturally extended to deal with more expressive purely navigational queries such as pSPARQL [1], nested regular expressions [21], NautiLOD [9] or LDQL [15], and show how to extend it so that we return not only the nodes connected by a property path, but also their witnessing paths of triples.

– We test the feasibility of the algorithm by running it over property paths queries which have been considered in previous literature. We also compare the algorithm to a BFS-based implementation and demonstrate that in the worst case has comparable performance, and at best requires significantly less resources to obtain query answers. structural proposals such as the Linked Data Fragments.

2 Preliminaries

RDF graphs and linked data. Let $\mathcal{I}$, $\mathcal{L}$, and $\mathcal{B}$ be countably infinite disjoint sets of *IRIs*, *literals*, and *blank nodes*, respectively. The set of *RDF terms* $\mathcal{T}$ is $\mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$. An *RDF triple* is a triple (s, p, o) from $\mathcal{T} \times \mathcal{I} \times \mathcal{T}$, where s is called *subject*, p *predicate*, and o *object*. An *(RDF) graph* is a finite set of RDF triples.

To formalize the notion of Linked Data we rely on the model introduced in [13]. A *Web of Linked Data* is a tuple $W = (\mathcal{G}, adoc)$, where $\mathcal{G}$ is a set of RDF graphs and $adoc : \mathcal{I} \to \mathcal{G} \cup \{\emptyset\}$ is a function that assigns graphs in $\mathcal{G}$ to some IRIs, and the empty graph to the resto of the IRIs [2].

The intuition behind this definition is that $\mathcal{G}$ represents the set of documents in the Web of Linked data, and *adoc* captures IRI-based retrieval of documents (for example, through dereference). Note that the tuple $(\mathcal{G}, adoc)$ is usually not available and has to be retrieved by looking up IRIs with *adoc*.

Property Path Expressions. We adopt the formalisation in [19], but note

$$
\begin{aligned}
[\![a]\!]_G &= \{(s,o) \mid (s,a,o) \in G\},\\
[\![e^-]\!]_G &= \{(s,o) \mid (o,s) \in [\![e]\!]_G\},\\
[\![e_1 \cdot e_2]\!]_G &= [\![e_1]\!]_G \circ [\![e_2]\!]_G,\\
[\![e_1 + e_2]\!]_G &= [\![e_1]\!]_G \cup [\![e_2]\!]_G,\\
[\![e^+]\!]_G &= \textstyle\bigcup_{i>1} [\![e^i]\!]_G,\\
[\![e^*]\!]_G &= [\![e^+]\!]_G \cup \{(a,a) \mid a \text{ is a term in } G\},\\
[\![e?]\!]_G &= [\![e]\!]_G \cup \{(a,a) \mid a \text{ is a term in } G\},\\
[\![!\{a_1,\ldots,a_k\}]\!]_G &= \{(s,o) \mid \exists a \text{ with } (s,a,o) \in G \text{ and } a \notin \{a_1,\ldots,a_k\}\},\\
[\![!\{a_1^-,\ldots,a_k^-\}]\!]_G &= \{(s,o) \mid (o,s) \in [\![!\{a_1,\ldots,a_k\}]\!]_G\},
\end{aligned}
$$

Table 1. The *evaluation* $[\![e]\!]_G$ of a property path expression e over an RDF graph G. Here $\circ$ is the usual composition of binary relations, and e^i is the concatenation $e \cdot \ldots \cdot e$ of i copies of e relations.

that the standard sometimes uses different symbols for operators; for example, inverse paths e^- and alternative paths $e_1 + e_2$ from our definition are denoted there by $\hat{}e$ and $e_1 \mid e_2$, respectively.

[2] Previous work such as [13] usually defines *adoc* as a partial function. We adopt instead the convention that $adoc(u) = \emptyset$ whenever *adoc* is not defined for u, as it simplifies the presentation of this paper

Definition 1. Property path expressions *are defined by the grammar*

$$e \;:=\; a \mid e^- \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^+ \mid e^* \mid e? \mid !\{a_1, \ldots, a_k\} \mid !\{a_1^-, \ldots, a_k^-\},$$

where $a, a_1, \ldots, a_k$ *are IRIs in* $\mathcal{I}$*. Expressions starting with* ! *are called* negated property sets*. We denote the set of all property path expressions by* **PP***.*

The normative semantics of Property Path expressions is given in terms of the *evaluation* $[\![e]\!]_G$ of a property path expression e over an RDF graph G, and is shown in Table 1. As is usual with navigational languages for graph databases, the problem of deciding whether a pair of IRIs belongs to the evaluation of a Property Path expression e in a graph G is rather easy in terms of computational complexity. For example, Kostlyev et al. provide an algorithm that works in time $O(|G| \cdot |e|)$ that is based on a simple reachability argument [19].

Let $\mathcal{V}$ be an infinite set of variables disjoint from $\mathcal{T}$. We denote elements of $\mathcal{V}$ by $?x, ?y$, etc. A *property path pattern* is a triple in $(\mathcal{T} \cup \mathcal{V}) \times \mathbf{PP} \times (\mathcal{T} \cup \mathcal{V})$. For instance, the expression $(\texttt{dbr:Kevin\_Bacon}, (\texttt{dbo:starring}^-\texttt{dbo:starring})^+, ?x)$ is a property path pattern. Property path patterns are added to SPARQL as as usual graph patterns (see [19] for more details). The set of variables of a property path pattern P, denoted by $Var(P)$, is simply the set containing all elements from $\mathcal{V}$ appearing in P. Semantics of a property path pattern is defined via mappings in the same way as for usual SPARQL queries [11]. The evaluation of a property path pattern $P = (u, e, v)$ over a graph G is then formally defined as the set of all mappings $\{\mu : Var(P) \to \mathcal{T} \mid (\mu^*(u), \mu^*(v)) \in [\![e]\!]_G\}$, where $\mu^*(u) = u$, if $u \in \mathcal{T}$, and $\mu^*(u) = \mu(u)$ if $u \in \mathcal{V}$. Note that we adopt set semantics: we only focus on the mappings retrieved by a pattern and not their cardinality.

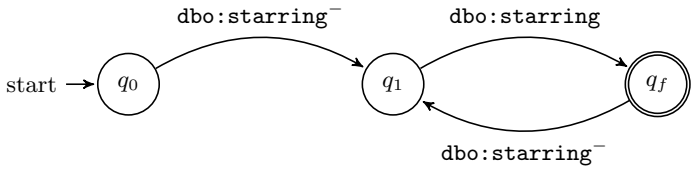
3 Property Paths over Linked Data

In this section we formally define the problem of computing property paths over linked data. Since our goal is to solve this problem using AI search techniques, it is best if we adopt an automata theoretic approach to define the answers of a property path. We thus start by introducing the notion of property automata, and then show how to use these automata to define the answers of property path patterns. We finish with a comparison with the semantics introduced in [16].

3.1 Automata for Property Paths

Let Σ be a set of IRIs, and define $\Sigma^- = \{a^- \mid a \in \Sigma\}$ and $\Sigma^{\pm!} = \Sigma \cup \Sigma^- \cup \{!S \mid S \subseteq \Sigma \text{ or } S \subseteq \Sigma^-\}$. A property automaton over Σ is just an NFA over the extended alphabet $\Sigma^{\pm!}$. That is, a tuple $\mathcal{A} = \{Q, \Sigma, q_0, F, \delta\}$, where Q is the set of states, q_0 is an initial state, F is the set of final states and $\delta \subseteq Q \times \Sigma^{\pm!} \times Q$ is the transition relation.

Example 2. Consider the following simple automata, which intuitively corresponds to the property path $(\texttt{dbo:starring}^-/\texttt{dbo:starring})^+$ from the introduction.



As we can see, the idea is to represent each property path with an automaton, in the same way as it is done with classical regular expressions. The difference in syntax is that some transitions can be labelled with inverses, and others with sets of IRIs or inverses of IRIs. The semantics, however, is completely different from regular NFAs.

Semantics. In order to give semantics for property automata, we fix a Web of Linked Data $W = (\mathcal{G}, adoc)$ and define the function $neighbors_a$, parameterised by the symbol $a \in \Sigma^{\pm!}$, that takes an IRI u as input and gives all those IRIs that can be *traversed* from u by means of a. Formally, for each $a \in \Sigma^{\pm!}$ we define the function $neighbors_a(u) : \mathcal{I} \to 2^{\mathcal{T}}$ as follows:

- $neighbors_a(u) = \{u' \mid adoc(u)$ contains the triple $(u, a, u')\}$, if $a \in \Sigma$;
- $neighbors_a(u) = \{u' \mid adoc(u)$ contains the triple $(u', a, u)\}$, if $a \in \Sigma^-$; [3]
- $neighbors_a(u) = \{u' \mid u' \in neighbors_b(u)$ for $b \notin S\}$, if $a = !S$ and $S \subseteq \Sigma$;
- $neighbors_a(u) = \{u' \mid u' \in neighbors_{b^-}(u)$ for $b^- \notin S\}$, if $a = !S$ and $S \subseteq \Sigma^-$.

Semantics is given by means of configurations. A configuration of a property automaton $\mathcal{A}$ is a tuple (q, u), where q is a state in Q and u is a term in $\mathcal{T}$. Intuitively, a configuration represents a piece of the navigation that is defined by $\mathcal{A}$: we are positioned in u while parsing $\mathcal{A}$ in state q.

A run of a property automaton is a sequence of configurations $c_0, \ldots, c_k$ such that c_0 is of the form (q_0, u), c_k is of the form (q_f, u') for some state $q_f \in F$, and for each $c_i = (q_i, u_i)$ and $c_{i+1} = (q_{i+1}, u_{i+1})$ we have that:

1. There is a symbol $a \in \Sigma^{\pm!}$ such that q_{i+1} belongs to $\delta(q_i, a)$,
2. The IRI u_{i+1} belongs to $neighbors_a(u_i)$

Note that, unlike regular NFAs, the runs for property automata are defined with respect to a starting term and an ending term. In this case we say that the run starts in u and ends in u'.

Example 3. Let us consider a run of the automata in Example 2 that starts in `dbr:Kevin_Bacon` and ends in `dbr:Julianne_Moore`. The run starts with $(q_0, \texttt{dbr:Kevin\_Bacon})$, which corresponds to the initial configuration for Kevin Bacon. Next, given that $adoc(\texttt{dbr:Kevin\_Bacon})$ contains the triple

$$(\texttt{dbr:Crazy\_Stupid\_Love}, \texttt{dbo:starring}, \texttt{dbr:Kevin\_Bacon}),$$

[3] Note that we are implicitly assuming that $adoc(u)$ contains at least some triples where u is not the subject, but the object. While some servers do register in $adoc(u)$ all triples where u is present, others only register those triples where u is a subject. We discuss how to deal with this issue in Section 3.3.

then dbr:Crazy_Stupid_Love is in $neighbors_{\texttt{dbo:starring}^-}(\texttt{dbr:Kevin\_Bacon})$, which allows us to advance to the configuration $(q_1, \texttt{dbr:Crazy\_Stupid\_Love})$. The final configuration, $(q_f, \texttt{dbr:Julianne\_Moore})$, is reached using a similar argument.

3.2 Using automata to compute SPARQL queries

As it is customary in formal languages, to each property path e we can associate a property automaton $\mathcal{A}_e$. We obtain this automaton in two steps:

- First we normalize e into an equivalent property path e' where all the inverses are applied only to IRIs (see e.g. [4] for a precise algortithm).
- Treating e' as a standard regular expression over $\Sigma^{\pm!}$ (we can do this because e' is already normalized), we construct an NFA (over $\Sigma^{\pm!}$) for e' in the usual way. The property automata $\mathcal{A}_e$ is an exact copy of this NFA, except $\mathcal{A}_e$ runs over Σ and is understood as a property automaton.

The following shows that automata capture the correct notion of computing links for property paths. Recall that we are working with a fixed Web of Linked Data $W = (\mathcal{G}, adoc)$. We can then define the graph $\mathbf{\Omega}$ of W as $\mathbf{\Omega} = \bigcup_{G \in \mathcal{G}} G$.

Proposition 1. *Let e be a property path and $\mathcal{A}_e$ the corresponding property automata, and let u, u' be IRIs. If there is a run for $\mathcal{A}_e$ that starts in u and ends in u', then $(u, u') \in [\![e]\!]_{\mathbf{\Omega}}$.*

With this proposition we can now turn to the evaluation of SPARQL property path patterns. We say that an IRI u' can be discovered from an IRI u using an automaton $\mathcal{A}$ if there is a run that starts in u and ends in u'. Using this definition, we can specify an alternative semantics for the evaluation of SPARQL property path patterns over graphs.

Definition 2 (Automata-based semantics). *Given a property path pattern P and a Web of Linked Data W, the result of evaluating P over W under automata-based semantics is denoted $[\![P]\!]^{\mathcal{A}}_W$. The definition is given in Figure 2.*

Patterns of the form $\{?x\ e\ ?y\}$ are especially problematic, as both the start and end are existentially quantified and therefore there is nowhere to draw IRIs from. We define the semantics assuming there is already a graph G where we can draw IRIs from, for example, the graph that we have explored in previous computations or in the evaluation of a different pattern in the same computation.

3.3 Inverses and comparison with previous semantics

As argued by Hartig and Pirrò [16], the semantics as defined in the standard is not directly applicable to linked data that is distributed on the Web, as it is defined assuming a single, centralised graph. To remediate the situation they propose a context-based semantics for property paths, in line with the context-based semantics for SPARQL graph patterns proposed in [14, 13].

$[\![u\ e\ u']\!]^{\mathcal{A}}_W =$ [] if and only if u' can be discovered from u using $\mathcal{A}_e$.
$[\![u\ e\ ?x]\!]^{\mathcal{A}}_W =$ $\{[?x \to u']$ such that u' can be discovered from u using $\mathcal{A}_e\}$.
$[\![?x\ e\ u]\!]^{\mathcal{A}}_W =$ $\{[?x \to u']$ such that u' can be discovered from u using $\mathcal{A}_{e^-}\}$.
$[\![?x\ e\ ?y]\!]^{\mathcal{A}}_W = \{[?x \to u, ?y \to u']$ s.t. either $u \in G$ and u' can be discovered from u using $\mathcal{A}_e$, or $u' \in G$ and u can be discovered from u' using $\mathcal{A}_{e^-}\}$.

Fig. 2. Alternative semantics $[\![]\!]^{\mathcal{A}}_W$ for the evaluation of property path patterns. Here u and u' are IRIs, e is a property path, and G is an RDF graph. We use [] to denote the empty mapping and $[?x \to u]$ and $[?x \to u, ?y \to u']$ to denote the mapping that only maps $?x$ to u, and $?x$ to u and $?y$ to u', respectively.

Specifically, Hartig and Pirrò distinguish two types of semantics: a Full-Web semantics and the aforementioned Context-based semantics. Full-Web semantics for a property path e is just $[\![e]\!]_{\mathbf{\Omega}}$, which corresponds to the evaluation of e over the complete graph of the Web. But, as we mentioned, this semantics is not feasible in practice. This is why the alternative Context-based semantics was proposed, which can be seen as an approximation of Full Web semantics: since there is no way to obtain all possible answers, we just take what the context-based semantics gives us.

The idea of context-based semantics is that the next triples that should follow the evaluation of a property path can only be selected within the document that we are currently exploring. Assume that we are evaluating a property path e of the form a^*b, for some $a, b \in \mathcal{I}$. Moreover, assume that we have a sequence of triples $(u_0, a, u_1), (u_1, a, u_2), \ldots (u_{n-1}, a, u_n)$. Now, in order to continue from u_n with this path, we need to find a triple of the form (u_n, a, u_{n+1}) or (u_n, b, u_{n+1}) for some $u_{n+1} \in \mathcal{T}$. The question is, where do we look for this triple? Full Web semantics tells us that we should look for this triple in the entire Web, which is not very practical as we have no efficient way of doing this. Context-based semantics restricts this search over $adoc(u_n)$, the document retrieved from u_n.

It turns out that we can define Context-based semantics in our terms, albeit a little bit of extra work is needed when property paths use inverses. The first observation is that both semantics coincide for the case of property paths that do not use inverse paths. This is expected, since our automata are also based on the idea of looking for connections only in those triples that are dereferenced.

Proposition 2. *For every property path pattern $(u, e, ?x)$, for $?x$ in $\mathcal{V}$ and $u \in \mathcal{T}$, and where e does not use inverses, the set $[\![(u, e, ?x)]\!]^{\mathcal{A}}_W$ coincides with the set of mappings retrieved by the context-based semantics of [16].*

The comparison becomes more complicated when considering property paths with inverses or patterns of the form $(?x, e, u)$ (even if now e does not use inverse). To see this, consider the pattern $(?x, \texttt{knows}, \texttt{Tim})$, which under our semantics is equivalent to $(\texttt{Tim}, \texttt{knows}^-, ?x)$. To compute answers for this query we need to retrieve IRIs u' such that there is a triple of the form $(u', \texttt{knows}, \texttt{Tim})$. Since our focus is on the algorithmic side, we have assumed for now that we look for those triples in $adoc(\texttt{Tim})$. In contrast, to state Hartig and Pirrò's context-based semantics in our term we need to define $neighbors_{a^-}(u)$ as the set of all IRIs

$u' \in \mathcal{I}$ such that $adoc(u')$ now contains the triple (u', a, u). In other words, we now search for *all possible IRI's* that, when dereferenced, produce the triple in question. However, even they acknowledge that there is no feasible way to compute these answers.

Computing reverse links in practice. As we have mentioned, our algorithms rely on the assumption that $adoc(u)$ contains some triples where u is not the subject, but the object. Having these types of reverse links is not uncommon in Linked Data, and for example this is the case for DBPedia. Nevertheless, to cope with the cases when $adoc(u)$ does not contain reverse links, we maintain a list of SPARQL endpoints in our system that can be used in case $adoc(u)$ does not contain inverse links. In this case we look whether the domain of u corresponds to the domain of any of our endpoints. If it is so, then we set $neighbors_{a^-}(u)$ as the set of all answers to SPARQL query `SELECT ?s WHERE {?s a u}`, posed in the endpoint corresponding to u. This practical definition of $neighbors_{a^-}(u)$ is clearly less general that what Hartig and Pirrò define, so even with this hack we can only guarantee that our algorithm will deliver just a subset of the answers defined by their context-based semantics.

We would like to note that the community is already building an infrastructure that would eliminate this mismatch: the *Linked Data Fragments* initiative [24], which aims to study different ways of publishing linked data on the web. Specifically, one of the proposals of this initiative is to build an infrastructure that can support the answer of any SPARQL triple pattern. This infrastructure is called *Triple Pattern Fragments* [23] and, if present, would be enough to process runs of our automata, since both $neighbors_a(u)$ and $neighbors_{a^-}(u)$ can be obtained with the triple pattern queries `SELECT ?x WHERE {u a ?x}` and `SELECT ?x WHERE {?x a u}`, respectively.

4 Property Automata Querying as AI Search

Heuristic Search is a well-established area of Artificial Intelligence (AI) that aims at solving hard search problems, with applications ranging from puzzle solving (e.g., [18]) to computational biology (e.g., [17]). A deterministic, single-agent search problem intuitively corresponds to finding a path in a graph connecting an *initial state* with a *goal state*. More precisely, a search problem is a tuple $(s_0, Act, Succ, \varphi_G)$, where:

- s_{start}, Act, and $Succ$ define an implicit *search graph* in the following way. s_{start} is the *initial state*. Act is a set of *action operations* (or simply actions), and $Succ$ is a partial function that given an action and a state returns a state. We denote actions by the letter o, and states by s (with subscripts is needed). When $Succ(o, s)$ is defined we say that o is applicable in state s. We say s' is a successor of s if for some $o \in Act$, $Succ(o, s) = s'$. Slightly abusing notation, $Succ(s)$ denotes the set containing all the successors of s. The nodes of the search graph are s_{start} and all states resulting from successive application of actions over s_{start}. An arc between s and s' exists if and only if s' is a successor of s.

– φ_G is a *goal condition*, which is a Boolean function that returns true for goal states.

A *solution* to a search problem is a sequence of action operations $o_0 o_1 o_2 \dots o_n$ whose successive application over s_0 leads to a state that satisfies φ_G. To define this formally we extend the definition of *Succ* to sequences of actions: if α is a sequence of actions and $o \in Act$, then $Succ(o\alpha, s)$ is equal to $Succ(o, Succ(\alpha, s))$ if $Succ(\alpha, s)$ is defined, and is undefined otherwise. In addition $Succ(\alpha, s) = s$ if α is the empty sequence. A state trace $\rho = s_0 s_1 \dots s_{n+1}$ is *induced by the execution of a sequence of actions* $o_0 o_1 \dots o_n$ iff $s_0 = s_{start}$ and $Succ(o_0 \dots o_i, s_{start}) = s_{i+1}$, for every $i \in \{0, \dots, n\}$. A sequence of actions α is a *solution* to the search problem iff $Succ(\alpha, s_{start})$ satisfies φ_G.

The problem of computing the run for a given automaton (i.e., finding the answer to a property path query), can be reduced to that of finding a solution to a search problem. Indeed, given a property automaton $\mathcal{A} = \{Q, \Sigma, q_0, F, \delta\}$, and an initial IRI u_0, we define a search problem $P = (s_0, Act, Succ, \varphi_G)$, where

– $s_0 = (q_0, u_0)$, thus the initial state is an ordered pair containing an IRI and the initial state of $\mathcal{A}$.
– $Act = \Sigma^{\pm !} \times Q$.
– $Succ((a, p), (q, u))$ is defined as (q', u') if $p = q'$, $neighbors_a(u)$ contains u', and $\delta(q, a) = q'$, and it is undefined otherwise.
– $\varphi_G(u, q)$ is defined as $q \in F$.

Intuitively, an action (a, q') tells us that by reading a letter a we move to the state q' of $\mathcal{A}$. We need both the state and the symbol because $\mathcal{A}$ is nondeterministic. Our states are configurations of the property path automaton $\mathcal{A}$ (over a Web of linked data), so an action (a, q') is applicable to (q, u) with the result (q, u'), if $\delta(q, a) = q'$, and u' is an a-neighbor of u. The following result establishes that a run of the property automaton $\mathcal{A}$ can be obtained by solving P.

Proposition 3. *If α is a solution to P then the state trace ρ induced by α on s_0 is a run of $\mathcal{A}$.*

4.1 A Search Algorithm for Property Path Queries

A* [12] is one of the most well-known algorithms for solving search problems. Besides a search problem, A* receives a *heuristic function*, h, as a parameter. This function, which is key to A*'s performance is an estimate of cost-to-go; in other words, $h(s)$ estimates the cost of a path from s to a goal state. If h is *admissible*, i.e., it is such that for every state s, $h(s)$ never overestimates the cost of any path from s to a goal state, then A* is guaranteed to find an optimal solution; i.e., a shortest path between the initial state and a goal.

In a nutshell A* works as follows. At every moment it keeps a list of states, *Open*, in which states are ranked using a function $f(s) = g(s) + h(s)$, where h is the heurisitic function and $g(s)$ is the cost of the best path towards s that has been found so far by A*. In each iteration it performs the following steps:

(1) the lowest f-value state, s_{best}, is extracted from *Open*; (2) if s_{best} is a goal, it returns; (3) s_{best} is *expanded*, which means that each successor s' of s_{best} is computed and added to *Open* unless s' has been previously found by the current run via a better path.[4]

While A* can be applied directly to the problem of answering a path property query, here we propose to use variant of k-Best-First Search (k-BFS) [7]. k-BFS is a generalization of A* that expands k nodes of least f-value instead of simply one (like A* does). k-BFS is A* when k equals 1. k-BFS is also optimal when h is admissible.

The reason why we use k-BFS instead of A* is that expansions are computationally expensive in query answering (we elaborate on this later when we describe implementation aspects). The variant we use simply expands those k successors in parallel. As successors are computed, we add them to *Open*. After all successors have been computed (and added to *Open*), we continue with the next iteration. Algorithm 1 shows a pseudocode of k-BFS$_{\text{PE}}$. The main differences between this and a standard A* pseudocode are as follows. k-BFS$_{\text{PE}}$ extracts, in one shot, k nodes from the *Open* list, unlike A* which only extracts one state from *Open*. In addition, k-BFS$_{\text{PE}}$ expands several states in parallel (Line 9).

Guiding Search with a Heuristic. Above we mentioned that heuristic functions are essential for the performance of search algorithms. It turns out that for this class of problems it is possible to obtain an admissible heuristic from the property automaton in a rather straightforward way. Indeed, assume a problem P has been constructed using the property path automaton $\mathcal{A}$. Observe that the minimum number of actions required to reach a goal state from state (u, q) cannot exceed the number of edges of a shortest path between the automaton state q and a final state in the graph for automaton $\mathcal{A}$. This is because each time an action is applied to (u, q) the successor state (u', q') must be such that there is an edge between q and q' in the automaton's graph. Prior to search, we compute the shortest path of each state q in $\mathcal{A}$ to some final state, and store this value as an attribute of q. During search, $h(u, q)$ is a simple lookup that takes constant time.

A Discussion of Implementation Aspects. We implemented Algorithm 1 in Python. Consistent with standard AI implementations, the *Open* list is implemented with a priority queue, and generated states (Line 13) are stored in a hash table. The following aspects however are not standard in AI applications and are very particular to the application of A* over Linked Data.

- The computation of $Succ((a, q'), (q, u))$ needs an HTTP request to IRI u (we implement this using the RDFlib library). Therefore the time needed for this operation is extremely high relative to other operations of the search algorithm. This is the main reason we chose to run expansions in parallel. If many of the IRIs being expanded reside on the same server, this could

[4] In fact, if s' is already in *Open* and has been re-discovered via a better path most implementations will decrease the key of s' in *Open* instead of adding s' again to *Open*.

Algorithm 1: k-BFS$_{\text{PE}}$ with Parallel Expansions

```
1  procedure k-BFS_PE
2      Let Open be a priority queue ordered by f attribute
3      g(s_start) ← 0
4      f(s_start) ← h(s_start)
5      Insert s_start into Open
6      while Open ≠ ∅ do
7          Initialize B with an empty list
8          Extract min{k, |Open|} states from Open, and insert them in B
9          parallel for each b in B do
10             Expand(b)
11 procedure Expand (s)
12     for each o in Act such that Succ(o, s) is defined do
13         t ← Succ(o, s)
14         if t is a goal state then
15             Output t
16         if t has not been seen before then
17             Store t in memory
18             g(t) ← ∞
19         cost ← g(s) + 1
20         if cost < g(t) then
21             g(t) = cost
22             f(t) = g(t) + h(t)
23             parent(t) = ⟨s, a⟩
24             if t ∉ Open then
25                 Insert t in Open
26             else
27                 Update priority of t in Open
```

lead to delays or closed connections. Thus the optimal parameter k may be query- and database-specific.

- Our implementation keeps a local RDF graph which is formed by every triple (u, a, u') that has ever been fetched. This allows speeding up state re-expansions, but it also allows another interesting feature: because all previously seen triples are in memory, the implementation may compute a more complete version of the *adoc* function when it is looking for inverses. Indeed, if (u, a, u') has been fetched from the document u (when expanding a state of the form (q, u)), the algorithm "remembers" that u is related inversely to u' via a. If, later on, a state of the form (q', u') is expanded, then (u, a, u') can be considered as being part of $adoc(u')$ even if such a triple is not in it.
- As stated previously, computing "inverse" neighbors of u can be particularly problematic in practice, since most servers do not even publish inverse triples available on that very server. Because of this our implementation sometimes uses endpoints to obtain inverse edges, as described in Section 3.3.

4.2 Getting more than property paths

While the idea of computing the paths that explain a particular property path expression is not new, it has been classified as a difficult problem, since the number of those paths can be arbitrarily large. Previous attempts of restricting the structure of paths have been also deemed infeasible (see e.g. [2, 20]), so there

is still not a clear consensus on what types of paths may or could be retrieved from the evaluation of property path patterns over RDF graphs.

Another huge advantage of using our search algorithms is that we can actually retrieve the shortest path of triples that serve as an explanation for the answers of a property path expression. We formalise this as follows.

Let $t_1, \ldots, t_n$ be a sequence of triples, where each t_i is (s_i, p_i, o_i), and consider a run $\pi = c_1, \ldots, c_{n+1}$ for a property automaton $\mathcal{A}$, where each c_i is (q_i, u_i). We say that the sequence $t_1, \ldots, t_n$ witnesses the run π if c_1 is either (q_0, s_1) or (q_0, o_1), and for each $1 \leq i \leq n$, one of the following holds.

- We have that $u_i = s_i$, state q_{i+1} is in $\delta(q_i, p_i)$ and $u_{i+1} = o_i$, or
- it holds that $u_i = o_i$, state q_{i+1} is in $\delta(q_i, p_i^-)$ and $u_{i+1} = s_i$, or
- we have that $u_i = s_i$, state q_{i+1} is in $\delta(q_i, !S)$ for some $S \subseteq \Sigma$ such that p_i is not in S and $u_{i+1} = o_i$; or
- we have that $u_i = o_i$, state q_{i+1} is in $\delta(q_i, !S)$ for some $S \subseteq \Sigma^-$ such that p_i^- is not in S and $u_{i+1} = s_i$.

From this definition and Proposition 1, it is immediate to see that the property automata $\mathcal{A}_e$ of a property path e has a run that starts in u and ends in u' if and only if this run is witnessed by a sequence $t_1, \ldots, t_n$ of triples in the Web of Linked Data. This sequence can be understood as the path that explains why (u, u') belongs to the evaluation of e. Coming back to Example 3, the corresponding sequence of triples that witness the run given in this example is simply:
(`dbr:Crazy_Stupid_Love`, `dbo:starring`, `dbr:Kevin_Bacon`),
(`dbr:Crazy_Stupid_Love`, `dbo:starring`, `dbr:Julianne_Moore`).

On input u and $\mathcal{A}$, our implementation already computes a state trace ρ and a set α of actions that correspond to a run of $\mathcal{A}$ that starts in u. However, one could easily modify Algorithm 1 so that we also retrieve a sequence of triples that witnesses such run, simply by storing the triple we use to verify that $Succ(o, s)$ is defined. by the properties of our algorithm, one can actually show that, for each pair (u, u') we actually retrieve the shortest possible sequence of triples, amongst all the possible witnessing sequences that are formed with triples in the documents that we discover.

5 Experimental Evaluation

Next, we show how the algorithm from Section 4 fares when computing property paths over the Web of linked data. The objective of this study is twofold: first, we will show that our implementation can compute a significant amount of answers to property path queries which fail on SPARQL endpoints (even when asked for a single result); and second, we compare our algorithm to the most obvious way to implement property paths, namely, breadth-first-search (BFS). As input we take the property path (which is transformed into an appropriate automaton), and the start IRI, which allows us to search for the required data on the Web. We would again like to stress that at the beginning of the evaluation we *do not* have a local dataset which can aid us in the search. For our study we selected

five property path queries appearing in the previous literature [10, 22], and designated an appropriate start IRI for their evaluation. None of the five queries returns any answers when ran over the appropriate public SPARQL endpoint, due to exceeding the allotted memory quota, even when limited to a single answer. The queries are as follows (see `http://dvrgoc.ing.puc.cl/Planning` for more details):

- **Q1:** This query finds the (transitive) co-authors of Jorge A. Baier, starting in the DBLP dataset;
- **Q2** and **Q3:** Here we return actors with a finite Bacon number, while starting in DBPedia (Q2) and YAGO (Q3), respectively;
- **Q4:** The query Q4 finds places located in a geographical entity which has something to do with some European capital (starting in YAGO);
- **Q5:** This query searches the YAGO dataset in order to find places connected to an airport in the Netherlands.

Experiment Setup. Since our goal is to illustrate that A* variants can be useful when a partial answer to the query suffices, we test how much resources are required in order to compute the first one, ten, fifty and hundred answers. As users are usually not interested in seeing all the answers, having an algorithm capable of returning a fixed quantity of answers can provide very useful, particularly taking into account that our approach also has the ability to return path witnessing the answer to a property path query virtually for free.

Of course, when running queries over the Web, the execution time is not a good indicator of the efficiency of an algorithm, since it depends on many factors outside of our control (for instance, in one of the queries the first run took 2 seconds, and the next one 160, although both produced precisely the same sequence of requests). For this reason we measure the efficiency of our evaluation with respect to the number of server requests. To ensure some sort of a fair-usage policy, we also limited the total number of triples which can be obtained from the server to 100000, in order not to bomb the server with limitless requests. Since the queries Q1 through Q5 had trouble executing on the endpoints they were designed for due to exceeding the memory limit, we will also report the total amount of (system) memory our implementation used to compute the last answer to the query. As a baseline we test the algorithm with no parallelism (that is, 1-BFS, which in our context is essentially equivalent to classical A*). When parallel calls are allowed, the only efficiency measure that changes is the execution time, while the total number of requests, triples obtained, and the memory used is roughly the same.

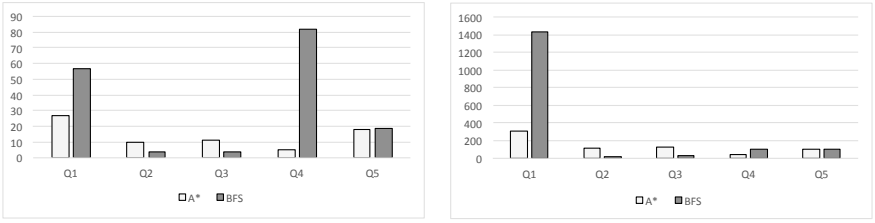
Results. Table 2 shows the result of our runs. We see that for a small set of answers we do not require huge amount of requests to the server, and the memory required to compute the answers is generally pretty low.

Although the numbers we provide are indicative that A*-based implementations of property paths are feasible in practice, as a sanity-check, we would also like to compare our implementation to the most obvious approach to computing property paths: using breadth-first-search (BFS). Note that BFS can be viewed

	Requests used to compute i answers				
	`LIMIT 1`	`LIMIT 10`	`LIMIT 50`	`LIMIT 100`	Memory
Q1	4	27	162	305	42 MB
Q2	3	10	63	118	44.5 MB
Q3	3	11	59	123	39.7 MB
Q4	5	5	N/A	N/A	46.8 MB
Q5	4	11	97	N/A	32 MB

Table 2. Number of requests needed to compute i-answers (i is the number in `LIMIT`). We also report the total memory used to compute the final answer (for Q4 and Q5 this is 20 and 50, respectively). All of the queries exceed the memory limit when ran on the corresponding endpoint and did not return any answers.

as a special case of A*, where the heuristic forces each node to be expanded completely, before getting a step closer to the final state. Here we test how our implementation fares when compared to BFS in terms of the number of server requests (the other measures, such as the number of triples, or memory, scale accordingly). Figure 3 shows the result for the runs not utilizing parallelism, however, analogous set of numbers is obtained when considering parallel runs. As one would expect, the results depend heavily on the shape of the queried data. In particular, for queries Q2, Q3 and Q5, we obtain many answers by doing just one or two joins, thus resulting in a comparable number of requests for BFS and A*. In case of queries Q1 and Q4, however, we can see that the number of requests and used triples is significantly lower for A*, mainly due to the fact that most answers are not obtainable by expanding the first node (or just a few nodes), but the paths one must follow in order to answer the query are much longer. To further illustrate his point, we also modified Q2, in order to find actors at distance 2, 3, and 4 from Kevin Bacon. The results here show that for distance 3 and 4, BFS can not even compute a single answer, due to surpassing the amount of permitted triples (100000), while A* finds first 100 answers using only 124 and 126 requests, in case of distance 3 and 4, respectively. When we require actors at a distance 2, BFS can compute the answers, however, it uses almost triple the amount of requests that A* needs (297 vs. 122).



(a) Requests needed for 10 answers (b) Requests needed for 100 answers

Fig. 3. Comparison of A* and BFS-based runs for queries Q1 through Q5.

Overall, we can conclude that using A* is a viable approach to implementing property paths when we do not want to compute the entire answer, but want to obtain the first few answers to the query rather quickly, and using relatively few server requests. What is encouraging is that we can show that A* is at its worst comparable to BFS, but when the data is more spread out, it can really speed up the computation. We would also like to highlight that this implementation also uses a very limited amount of working memory, and since all of the endpoints struggled with evaluating property paths due to a high memory demand, it might be a good alternative to evaluating property path queries over public endpoints.

6 Conclusions and Future Work

References

1. Alkhateeb, F., Baget, J., Euzenat, J.: Extending SPARQL with regular expression patterns (for querying RDF). J. Web Sem. 7(2), 57–73 (2009)
2. Arenas, M., Conca, S., Pérez, J.: Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In: WWW 2012 (2012)
3. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: Dbpedia: A nucleus for a web of open data. Springer (2007)
4. Barceló, P., Pérez, J., Reutter, J.L.: Relative Expressiveness of Nested Regular Expressions. In: AMW. pp. 180–195 (2012)
5. Berners-Lee, T., Bizer, C., Heath, T.: Linked data-the story so far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)
6. Berners-Lee, T., Chen, Y., Chilton, L., Connolly, D., Dhanaraj, R., Hollenbach, J., Lerer, A., Sheets, D.: Tabulator: Exploring and analyzing linked data on the semantic web. In: SWUI Workshop (2006)
7. Felner, A., Kraus, S., Korf, R.E.: KBFS: k-best-first search. Annals of Mathematics and Artificial Intelligence 39(1-2), 19–39 (2003)
8. Fionda, V., Gutierrez, C., Pirrò, G.: The swget portal: Navigating and acting on the web of linked data. J. Web Sem. 26, 29–35 (2014)
9. Fionda, V., Pirrò, G., Gutierrez, C.: NautiLOD: A Formal Language for the Web of Data Graph. TWEB 9(1), 5:1–5:43 (2015)
10. Gubichev, A., Bedathur, S.J., Seufert, S.: Sparqling kleene: fast property paths in RDF-3X. In: GRADES 2013. p. 14 (2013)
11. Harris, S., Seaborne, A.: SPARQL 1.1 query language. W3C (2013)
12. Hart, P.E., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimal cost paths. IEEE Transactions on Systems Science and Cybernetics 4(2) (1968)
13. Hartig, O.: Sparql for a web of linked data: Semantics and computability. In: The Semantic Web: Research and Applications, pp. 8–23. Springer (2012)
14. Hartig, O., Bizer, C., Freytag, J.C.: Executing SPARQL queries over the web of linked data. Springer (2009)
15. Hartig, O., Pérez, J.: Ldql: A query language for the web of linked data. In: The Semantic Web-ISWC 2015, pp. 73–91. Springer (2015)
16. Hartig, O., Pirrò, G.: A context-based semantics for SPARQL property paths over the web. In: ESWC 2015. pp. 71–87 (2015)

17. Hatem, M., Ruml, W.: External memory best-first search for multiple sequence alignment. In: AAAI 2013. AAAI Press (2013)
18. Korf, R.E.: Finding optimal solutions to rubik's cube using pattern databases. In: AAAI 1997. pp. 700–705. AAAI Press / The MIT Press (1997)
19. Kostylev, E.V., Reutter, J.L., Romero, M., Vrgoč, D.: Sparql with property paths. In: The Semantic Web–ISWC 2015, pp. 3–18. Springer (2015)
20. Losemann, K., Martens, W.: The complexity of evaluating path expressions in sparql. In: PODS 2012. pp. 101–112 (2012)
21. Pérez, J., Arenas, M., Gutierrez, C.: nSPARQL: A navigational language for RDF. J. Web Sem. 8(4), 255–270 (2010)
22. Reutter, J.L., Soto, A., Vrgoč, D.: Recursion in SPARQL. In: ISWC 2015 (2015)
23. Verborgh, R., Hartig, O., Meester, B.D., Haesendonck, G., Vocht, L.D., Sande, M.V., Cyganiak, R., Colpaert, P., Mannens, E., de Walle, R.V.: Querying datasets on the web with high availability. In: ISWC 2014. pp. 180–196 (2014)
24. Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Mannens, E., Van de Walle, R.: Web-scale querying through linked data fragments. In: LDOW (2014)

Appendix

A Queries used in Section 5

Here we provide the full code of the five queries used to test the feasibility of implementing property paths using the A* algorithm. Note that some of the property path queries from [10] have been modified in order to write them as a single property path.

Q1. The first query, Q1, finds the co-authors of Jorge A. Baier in the DBLP dataset.

```
NEED DIETRICH TO TELL ME THE EXACT QUERY

select * where {Jorge A. Baier (^akt:has-author/akt:has-author)* ?x }
```

Q2. The second query, Q2, finds people with a finite Bacon number in DBPedia.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>

select * where {dbr:Kevin_Bacon (^dbo:starring/dbo:starring)* ?x }
```

Q3. The third query, Q3, finds people with a finite Bacon number in YAGO.

```
PREFIX yago: <http://yago-knowledge.org/resource/>

select * where {?x (yago:actedIn/^yago:actedIn)* yago:Kevin_Bacon }
```

Q4. The fourth query, Q4, finds places located in a geographical entity which has something to do with some European capital (starting in YAGO).

```
PREFIX yago: <http://yago-knowledge.org/resource/>

select * where { yago:wikicat_Capitals_in_Europe
                     ^rdf:type/yago:isLocatedIn*/yago:dealsWith ?area }
```

Q5. The fifth query, Q5, finds the places connected to an airport in the Netherlands inside the YAGO dataset.

```
PREFIX yago: <http://yago-knowledge.org/resource/>

select * where { yago:wikicat_Airports_in_the_Netherlands
                                  ^a/yago:isConnectedTo* ?x }
```