

Evaluating navigational RDF queries over the Web

Jorge Baier, Dietrich Daroch, Juan L. Reutter, Domagoj Vrgoč
Pontificia Universidad Católica de Chile and Center for Semantic Web Research

ABSTRACT

Semantic Web, and its underlying data format RDF, lend themselves naturally to navigational querying due to their graph-like structure. This is particularly evident when considering RDF data on the Web, where various separately published datasets reference each other and form a giant graph known as the Web of Linked Data. And while navigational queries over singular RDF datasets are supported through SPARQL property paths, not much is known about evaluating them over Linked Data. In this paper we propose a method for evaluating property path queries over the Web based on the classical AI search algorithm A^* , show its optimality in the open world setting of the Web, and test it using real world queries which access a variety of RDF datasets available online and that are not necessarily known in advance.

ACM Reference format:

Jorge Baier, Dietrich Daroch, Juan L. Reutter, Domagoj Vrgoč. 2016. Evaluating navigational RDF queries over the Web. In *Proceedings of ACM Hypertext 2017*, (HYPERTEXT'17), 10 pages. DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Resource Description Framework (RDF) [30] is the World Wide Web consortium (W3C) standard for representing Semantic Web data. In essence, an RDF graph is a set of triples of internationalised resource identifiers (IRIs), where the first and last of them represent entity resources, and the middle one relates these resources, just as is done in graph databases [4]. The official query language for RDF databases is SPARQL [15].

To answer the need for including navigational features into SPARQL, the latest version of the language includes *property paths*, a set of queries that can be seen as the analogues of established graph database languages such as regular path queries and two-way regular path queries [11]. Consequently, property paths are already supported by the vast majority of existing SPARQL engines (e.g., [10, 24, 37]). The inclusion of navigational queries is also present in most other graph database models (see e.g. [4, 7]).

Besides the traditional approach where one issues a query over a (set of) graph databases, the community has further raised the need for a fundamentally different way of querying RDF data: to obtain answers of queries over the whole corpus of RDF data present on the Web and linked together into what is known as the *Web of Linked Data*, in a distributed way and without assuming any mediation nor centralised organisation in control of the data, following the *Linked Data Principles* [9].

The fundamental property of RDF data that makes this querying possible is that the IRIs in RDF documents published online should be *dereferenceable*. This basically means that by accessing any given IRI, we obtain a new RDF document describing its neighbourhood

(or a part of it) in the Linked Data graph. Let us explain how this works using the online RDF documents published by DBLP, one of the simplest datasets now forming part of the Web of Linked Data. In the RDF representation of DBLP, each researcher is given a unique IRI, as well as each paper. The authorship relation indicating that an author A wrote a paper P is then represented by the triple $\{P, dc:creator, A\}$. The IRI for each author, then serves as a good starting point for investigating the DBLP dataset, as dereferencing their IRI will intuitively give us all the papers written by this author. For example, if we dereference the IRI `M.Stonebraker`, representing Michael Stonebraker, we obtain a document containing, amongst other things, the following triples

```
M.Stonebraker      foaf:name  "M. Stonebraker"
inTods:StonebrakerWKH76  dc:creator M.Stonebraker
inSigmod:PavloPRADMS09  dc:creator M.Stonebraker
```

These triples indicate that M.Stonebraker is the author of the papers represented by IRIs `inTods:StonebrakerWKH76` and `inSigmod:PavloPRADMS09`, and that the name of the entity represented by M.Stonebraker is indeed "Michael Stonebraker". Suppose now that we need to retrieve the names of all the co-authors of Michael Stonebraker. It is very easy to do this using the linked data infrastructure: We first dereference the IRI `M.Stonebraker`, obtaining an RDF document that contains, in particular, a triple $\{P, dc:creator, M.Stonebraker\}$ for each paper P authored by M.Stonebraker. Then we just need to dereference each of the IRIs of these papers: dereferencing each of these IRIs P gives us triples of the form $\{P, dc:creator, A\}$, and now we know that A is a co-author of M.Stonebraker. The last step is to further dereference the IRI of each of these researchers, to look for a triple $\{A, foaf:name, N\}$ that indicates the name of the researcher (in this case N).

Of course, the query looking for co-authors of Michael Stonebraker can be seen as a fixed pattern: namely, it is a path of length two, starting in the IRI `M.Stonebraker` and traversing the edge `dc:creator` backwards (thus reaching a paper written by Michael Stonebraker), and then traversing the `dc:creator` edge forwards to reach one of his co-authors. But what happens when we want to generalise this query and obtain the collaboration reach of Michael Stonebraker, that is, his co-authors, the co-authors of his co-authors, their co-authors, etc? This is similar to the popular notion of Erdős number, but this time starting with a different author. To answer such a query a fixed length path will no longer suffice, since we do not know the distance between the starting node and the ending node in advance. We therefore need to use property paths; in this case this would be done using the query

```
M.Stonebraker  (^dc:creator/dc:creator)*  ?x,
```

which repeats the simple path from one author to a paper (using `^dc:creator` to follow an edge labelled `dc:creator` in a reverse direction) and then to another author (using `dc:creator`) an arbitrary number of times, as signified by the star operator `*`. The idea is as before, but now once a co-author is retrieved, search does not stop, but continues with this (co-)author as the starting node.

When evaluating these queries we have only dereferenced and fetched the documents that we needed in order to answer the query, and thus we are taking full advantage of the nature of Linked Data. There is another fundamental advantage of this approach: we can cross between different domains without any effort by using the infrastructure of the Web, which happens when a dereferenced IRI links to another IRI residing on a different server. This is in contrast with, for instance, issuing a single distributed query to a centralised endpoint, since we can access an arbitrary number of different sources. Furthermore, we can access data that is not published on dedicated endpoints; all that we need is data published on the standard Web architecture. Up to our best knowledge, this framework of distributed, decentralised and ungoverned querying has not been considered before the advent of Linked Data.

The advantages of these approaches have led the Semantic Web community to investigate the fundamentals of querying over the Web [18], and developing algorithms for answering SPARQL queries over Linked Data [19, 35]. Unfortunately, despite the potential that property paths could have in Web querying, most of the algorithms developed in this context focus on the pattern matching features of SPARQL, and do not consider property paths. Indeed, the majority of studies about property paths only consider how they work over a single centralised dataset [5, 14, 26, 38]. And while the need for understanding how property paths might work over the Web has repeatedly been raised by the research community [8, 20, 21], previous studies have mostly focused on understanding appropriate semantics and/or proposing new languages to help users navigate the Web, instead of describing the algorithms computing these answers. The only exception is [13], suggesting a basic depth-first search algorithm in the context of NautiLOD queries: a language proposal that extends property paths. Therefore, the main objective of this paper is to answer the question: *How can one efficiently evaluate property path queries over the Web of Linked Data?*

Contributions. Our main contribution is an algorithm for efficiently retrieving answers to property path queries over Linked Data. Our solution is based on the observation that evaluating property paths can be seen as a search problem over an initially unknown graph. Indeed, in the examples above we start from one known IRI (M. Stonebraker) and begin exploring its neighbours guided by the query we are trying to answer. But this problem has been well studied by the Artificial Intelligence community, and it is generally agreed that the most appropriate solution here is an heuristic-search algorithm such as A* [16, 31]. In this paper we propose a variant of A* for the setting of Linked Data by using the property path we are trying to answer as a heuristic to guide our search. The main advantages of this approach are the following:

- It allows to overcome shortcomings of basic graph traversal algorithms such as depth-first search (DFS) and breadth-first search (BFS). In fact, we show that A* dominates BFS and DFS, and that it is optimal with respect to the part of the graph that became available during the search. This, in some sense, is the best we can hope for in the open-world setting of the Web.
- It does not only allow to find pairs of nodes connected by a property path, but it can also return (one of the) shortest paths which witness this connection: a feature that existing SPARQL engines are currently lacking.

- It is very robust when evaluating property paths live over the Web infrastructure, and can often answer queries which fail even on SPARQL implementations executed over a local dataset.

Apart from describing the basic implementation of the A* algorithm and proving its optimality, we also develop several optimisations geared towards query answering in the Linked Data setting. Most notably, we show that dereferencing multiple IRIs in parallel can speed up the computation of property paths significantly. Finally, we describe how our implementation runs over the Web of Linked data using a number of real-world queries which utilise different RDF datasets. We compare our approach to BFS and DFS-based algorithms and their parallel versions, showing that A* is superior when it comes to querying over the Web.

Outline. We formalise Linked Data and property paths in Section 2. In Section 3 we describe how DFS and BFS can be used to answer property path queries and what are their shortcomings. In Section 4 we introduce the A* algorithm and show its optimality. Optimisations are presented in Section 5, and real-world experiments in Section 6. We conclude in Section 7.

2 PRELIMINARIES

RDF graphs. Let I and \mathcal{L} be countably infinite disjoint sets of IRIs and literals, respectively. An *RDF triple* is a triple (s, p, o) from $(I \cup \mathcal{L}) \times I \times (I \cup \mathcal{L})$, where s is called *subject*, p *predicate*, and o *object*. An *(RDF) graph* is a finite set of RDF triples. For simplicity we only deal with RDF documents that do not contain blank nodes.

Linked Data. We are interested in computing navigational queries over the wide body of RDF documents published on the Web that comprise what is known as the Web of Linked Data. As customary in the literature (see e.g. [3, 17]), we treat this corpus of documents as a tuple $W = (\mathcal{G}, adoc)$, where \mathcal{G} is a set of RDF graphs and $adoc : I \rightarrow \mathcal{G} \cup \{\emptyset\}$ is a function that assigns graphs in \mathcal{G} to some IRIs, and the empty graph to the rest of the IRIs. Note that previous work (e.g. [17]) 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.

The intuition behind this definition is that \mathcal{G} represents the set of documents on the Web of Linked data, and $adoc$ captures dereferencing; that is, $adoc(u)$ gives us the neighbours of u in \mathcal{G} . Note that \mathcal{G} is usually not available and has to be retrieved by looking up IRIs with $adoc$.

Example 2.1. We can now formalise the operations performed in the introduction over the linked data architecture of DBLP. Starting with the IRI M. Stonebraker, we can invoke $adoc$ on this IRI to fetch its associated graph

$$adoc(M.Stonebraker) = \begin{cases} M.Stonebraker & foaf:name & "M. Stonebraker" \\ inSigmod:PavloPRADMS09 & dc:creator & M. Stonebraker \\ \vdots & \vdots & \vdots \end{cases}$$

When looking for the coauthors of M. Stonebraker, we might want to fetch $adoc(inSigmod:PavloPRADMS09)$, which will give us a graph containing, amongst other things, triples of the form $(inSigmod:PavloPRADMS09, dc:creator, A)$, with A being the IRI

of the authors of the paper. To get the name of A we fetch the graph $adoc(A)$ and look for the triple with `foaf:name` as the predicate.

Property Paths. Navigational queries over graph databases commonly ask for paths that satisfy certain properties. The most simple of them correspond to *regular path queries*, or RPQs [2, 12], which select pairs of nodes connected by a path conforming to a regular expression, and *2-way regular path queries*, or 2RPQs [11], which extend RPQ with the ability to traverse an edge backwards. SPARQL features a class of navigational queries known as *property paths*, which are themselves an extension of the well known class of 2RPQs. For readability we assume we deal only with 2RPQs, adopting the formalisation in [26]. Note however that our algorithms (and our implementation) work for all property path expressions.

Formally, we define property paths by the grammar

$$e := u \mid e^- \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^* \mid e?,$$

where u is an IRI in \mathcal{I} . The semantics of property paths, denoted by $\llbracket e \rrbracket_G$, for a property path e and an RDF graph G , is shown below.

$$\begin{aligned} \llbracket a \rrbracket_G &= \{(s, o) \mid (s, a, o) \in G\}, \\ \llbracket e^- \rrbracket_G &= \{(s, o) \mid (o, s) \in \llbracket e \rrbracket_G\}, \\ \llbracket e_1 \cdot e_2 \rrbracket_G &= \llbracket e_1 \rrbracket_G \circ \llbracket e_2 \rrbracket_G, \\ \llbracket e_1 + e_2 \rrbracket_G &= \llbracket e_1 \rrbracket_G \cup \llbracket e_2 \rrbracket_G, \\ \llbracket e^* \rrbracket_G &= \bigcup_{i \geq 1} \llbracket e^i \rrbracket_G \cup \{(a, a) \mid a \text{ is a term in } G\}, \\ \llbracket e? \rrbracket_G &= \llbracket e \rrbracket_G \cup \{(a, a) \mid a \text{ is a term in } G\}. \end{aligned}$$

Here \circ is the usual composition of binary relations, and e^i is the concatenation of i copies of e .

2.1 Evaluating Property Paths via Automata

As in the case of the query computing the coauthor reach of M. Stonebraker, one is usually interested in computing all the IRIs that can be reached from a starting IRI u by means of a property path expression. Formally, we study the following problem.

Problem:	PPCOMPUTATION
Input:	Property Path e , RDF graph G , starting IRI u
Output:	All IRIs v such that $(u, v) \in \llbracket e \rrbracket_G$

Alternatively, one may wish to compute the full evaluation $\llbracket e \rrbracket_G$ of pairs connected via a path conforming to e . However, this operation is seldom used in practice: it is not an intuitive query to ask, and when using property paths in SPARQL one usually obtains starting points from other patterns or joins of patterns. Also, computing the full $\llbracket e \rrbracket_G$ is not even supported in all SPARQL systems (for instance Virtuoso allows only property paths with a starting point). Furthermore, as we will see in the following sections, in the open world setting of Linked Data it is only natural to have a starting point for our search, since it is unrealistic to expect the computation to traverse and manipulate the entire Web graph. This is why we chose to focus on PPCOMPUTATION.

To solve the PPCOMPUTATION problem, the theoretical literature proposed a simple algorithm based on automata theory. To present this algorithm, note first that our property paths are nothing more than regular expressions over the alphabet $\mathcal{I}^\pm = \mathcal{I} \cup \{u^- \mid u \in \mathcal{I}\}$ that contains all IRIs and their inverses. Thus, for each property path e we can construct a nondeterministic finite state automaton

(NFA) A_e over \mathcal{I}^\pm that accepts the same language as e , when viewed as a regular expression. We can now show:

PROPOSITION 2.2 ([11, 12, 26]). PPCOMPUTATION can be solved in $O(|G| \cdot |e|)$ (thus linear in both the size of the graph and the query).

The idea is as follows. Let G be an RDF graph, e a property path expression and u an IRI. First, we construct the automaton $A_e = (Q_e, \mathcal{I}^\pm, q_e^0, F_e, \delta_e)$ equivalent to the query e , where Q_e is the set of states, q_e^0 is the initial state, F_e is the set of final states and $\delta_e \subseteq Q_e \times \mathcal{I}^\pm \times Q_e$ is the transition relation. Next, from G and A_e we construct the labelled product graph $G \times A_e$ whose nodes come from $\mathcal{I} \times Q_e$, and there is an edge from a node (u_1, q_1) to a node (u_2, q_2) labelled with $a \in \mathcal{I}$ if and only if (i) G contains a triple (u_1, a, u_2) and (ii) the transition relation δ_e contains the triple (q_1, a, q_2) , that is, if in A_e one can advance from q_1 to q_2 while reading a . Similarly, there is an edge between (u_1, q_1) and (u_2, q_2) labelled with $a^- \in \mathcal{I}^-$ if (i) $(u_2, a, u_1) \in G$ and (ii) $(q_1, a^-, q_2) \in \delta_e$. It is now not difficult to show the following property:

LEMMA 2.3 ([12]). A pair (u, v) belongs to $\llbracket e \rrbracket_G$ if and only if there is a path from (u, q_e^0) to (v, q_e^f) in the labelled graph $G \times A_e$, where $q_e^f \in F_e$ is a final state of A_e .

We can now solve the PPCOMPUTATION problem by traversing the product graph $G \times A_e$ starting in (u, q_e^0) and returning all the IRIs v such that we encounter a node (v, q_e^f) , with $q_e^f \in F_e$, during our traversal. Thus, in a sense, one can recast the problem of query computation (in a single graph) as the problem of searching for all connected final nodes in the product graph. This duality between evaluation and search is a crucial component of our approach for querying multiple graphs on the Web of Linked Data.

3 COMPUTING PROPERTY PATHS OVER THE WEB

When computing the answer of a property path over the Web, we cannot simply rely on the algorithm outlined in Section 2.1, because this assumes that we have our entire graph in memory, which is not a feasible option for the case of the Web. Having a starting IRI u comes in handy here, as we can emulate the algorithm from Section 2.1 by dereferencing u , retrieving its neighbours in $adoc(u)$, and continuing from there, thus building a local copy of a portion of the Web graph needed to answer the query.

To formalise this, let us define the *Web graph* G_{Web} as the RDF graph consisting of the union $\bigcup_{u \in \mathcal{I}} adoc(u)$ of all the graphs resulting by dereferencing an IRI in \mathcal{I} (i.e. the complete Web of Linked Data). From here onwards we assume that $adoc(u)$ gives us the neighbours of u in the Web graph (see Section 5.2 for a discussion of how to deal with the shortcomings of the current Linked Data infrastructure). The problem we are now interested in is solving PPCOMPUTATION above for the graph G_{Web} , i.e. the problem:

Problem:	PP_OVER_THE_WEB
Input:	Property Path e , starting IRI u
Output:	All IRIs v such that $(u, v) \in \llbracket e \rrbracket_{G_{\text{Web}}}$

Now, although the approach of Section 2.1 would require us to do our search over $G_{\text{Web}} \times A_e$, we can recast PP_OVER_THE_WEB as finding paths inside a subgraph $G_P \subset G_{\text{Web}} \times A_e$ which is

constructed dynamically by dereferencing IRIs starting at u . And although the graph G_P might be much smaller (in fact, we can stop constructing it when we desire), selecting the best algorithm for producing this graph and doing path searching over it is not an obvious task, due to the following issues not occurring in the classical path-finding setting.

First, path-finding algorithms are designed to work with graphs that can be either stored in memory or generated efficiently. In contrast, graph G_P is generated by dereferencing IRIs which involves resolving a number of HTTP requests. The time required to complete a request dominates significantly the time required to carry out any operation performed in memory. Efficient algorithms for this problem should therefore aim at reducing network requests, a factor that is usually not considered when solving path-finding problems. The second issue is that here we are interested in more than one solution. As such, an algorithm that returns answers incrementally seems to be a more sensible option than one that computes *all* answers prior to returning any.

Next, we discuss how classical path-finding algorithms can be modified to return answers to property path queries over the Web and pinpoint some of their shortcomings in this setting.

3.1 Depth-First Search

Depth-First Search (DFS) is an easy-to-implement path-finding algorithm that can be used to solve the PP_OVER_THE_WEB problem. On input a starting IRI u and an automaton A_e over \mathcal{I}^\pm , the algorithm begins a search over the graph $G_{\text{Web}} \times A_e$ starting with the node $init = (u, q_e^0)$, where q_e^0 is the initial node of A_e . The goal of the algorithm is to look for nodes of the form (v, q_f) , with v an IRI and q_f a final state of A_e ; this is commonly known as the *goal* condition of the algorithm. At every moment during execution, the algorithm maintains a *search frontier* (or *Open* list) implemented as a stack. At initialisation, the frontier is set to only contain the start node $init$. In the main loop, a node s is extracted from the frontier and *expanded* by computing its neighbours in $G_{\text{Web}} \times A_e$, by means of the function *Neighbours*. All neighbouring goal nodes are returned, and then all neighbours that have not been previously added to the frontier are now inserted at the top of the frontier. The algorithm terminates unsuccessfully if the frontier empties.

A pseudo code for DFS is presented in Algorithm 1. Note that we need *Open* to be a stack for DFS (Line 7). Observe additionally that the algorithm does not return a path but rather a node from which a path can be obtained by following the so-called parent pointers (set in Line 11). Finally, observe that in the context of navigational query answering, computing the neighbours of a node (function *Neighbours*) needs IRI dereferencing (set in Line 16) which in turn requires network communication, an operation that may take significantly more time than others carried out by the algorithm, such as data management.

There are three properties of DFS that are important for query answering. First, DFS can be easily modified to return paths incrementally instead of only one path. Indeed, instead of returning in Line 12, the node just found to be a goal node can be added to a list of solutions. In the same spirit, one can easily adapt DFS to return the first k solutions by introducing k as an additional parameter. Second, DFS is complete for finite graphs: if a goal node

Algorithm 1: Breadth/Depth-First Search

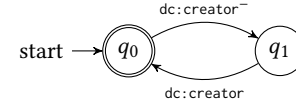
```

1 function Search( $u, A_e$ )
2    $init \leftarrow (u, q_e^0)$ 
3    $init.parent \leftarrow null$ 
4   if  $q_e^0 \in F_e$  then return  $init$  or add  $init$  to solutions
5   Initialise  $Open$  as an empty stack (DFS) or queue (BFS)
6   Initialise  $Seen$  as an empty set
7   Insert  $init$  into both  $Open$  and  $Seen$ 
8   while  $Open$  is not empty do
9     Extract node  $s = (v, q)$  from  $Open$  and compute  $Neighbours(s)$ 
10    for each  $t = (v', q')$  in  $Neighbours(s)$  that is not in  $Seen$  do
11       $t.parent \leftarrow s$ 
12      if  $q' \in F_e$  then return  $t$  or add  $t$  to solutions
13      Insert  $t$  into both  $Open$  and  $Seen$ 
14 function Neighbours( $v, q$ )
15   Initialise  $Succ$  as an empty set and RDF graph  $G_{temp}$  as an empty graph
16    $G_{temp} \leftarrow adoc(v)$ 
17   for each IRI  $a \in \mathcal{I}$  and state  $q'$  s.t.  $(q, a, q')$  is in  $\delta_e$  do
18     for each triple  $(v, a, v')$  in  $G_{temp}$  do Insert  $(q', v')$  into  $Succ$ 
19   for each IRI  $a^- \in \mathcal{I}^-$  and state  $q'$  s.t.  $(q, a^-, q')$  is in  $\delta_e$  do
20     for each triple  $(v', a, v)$  in  $G_{temp}$  do Insert  $(q', v')$  into  $Succ$ 
21   return  $Succ$ 

```

is reachable from $init$ then the algorithm eventually retrieves this node. This is important because it guarantees that all solutions to a query are eventually returned. Third, the memory footprint of DFS is relatively low. Actually, if the depth of the node on top of the stack is k and the maximum branching factor (number of neighbours of a node) is b , then the size of *Open* is $O(kb)$.

To see how DFS works when solving PP_OVER_THE_WEB, let us consider the first steps taken when processing the property path $(dc:creator^- \cdot dc:creator)^*$, with the starting IRI M. Stonebraker, which was presented in the introduction. First, let A_e be the following NFA:



As explained in Lemma 2.3, the starting node for our search is $(M.\text{Stonebraker}, q_0)$. In the first iteration we extract this node from *Open*, dereference the IRI M.Stonebraker, obtaining, amongst others, the triple $(inTods:StonebrakerWKH76, dc:creator, M.Stonebraker)$. This will allow us to add to our frontier the node $(inTods:StonebrakerWKH76, q_1)$ and we proceed similarly for other triples in $adoc(M.Stonebraker)$. For the second iteration, let us assume $(inTods:StonebrakerWKH76, q_1)$ is at the top of the stack. When expanded, DFS will lookup $adoc(inTods:StonebrakerWKH76)$ and retrieve, amongst other things, all nodes connected to $inTods:StonebrakerWKH76$ by means of a label $dc:creator$. This, in particular, yields all 4 authors of this paper, but $(M.Stonebraker, q_0)$ is not added to *Open* because it was already in *Seen*. For the next iteration DFS takes one of these nodes, say $(G.Held, q_0)$, expands them again, obtaining all papers of G. Held; the next iteration expands one of these papers, adds all the authors to the list of answers; and so on.

Algorithm 1 implements a *loop detection* by preventing the insertion of a previously seen node to *Open*. This is important to guarantee that the algorithm terminates over a finite graph and that the answers are complete. In our case this implies that we are looking for simple paths, albeit not in the RDF graph but in

the product graph $G_{\text{Web}} \times A_e$. In practice this implies that our algorithm looks for paths where the same IRI may be repeated at most a number of times equivalent to the states of the expression automata A_e . Completeness of DFS in our context follows from a simple pumping argument and the fact that property paths are regular expressions over \mathcal{I}^\pm .

The most notable drawback of DFS is that there is no guarantee on solution quality, and solutions with much shortest paths may be missed. For instance, in the query above it will return the co-authors of G. Held, which are at distance two or more from M. Stonebraker, before returning the other authors of the paper inTods:StonebrakerWKH76. In practice this means that we would need many more HTTP requests to retrieve subsequent solutions, which in turn means more time to compute answers.

3.2 Breadth-First Search

To alleviate the drawbacks of DFS, one could consider instead using *Breadth-First Search* (BFS), another complete search algorithm that is guaranteed to find shortest paths. BFS is similar to DFS in most aspects: it keeps a search frontier (i.e. the *Open* list) during execution and in each iteration it extracts a node from the frontier and then expands it. The most important difference is that BFS, instead of always expanding the deepest node in the frontier, it always expands the shallowest one. At the algorithmic level, BFS can be obtained from DFS by simply changing underlying data structure for *Open* to a FIFO queue instead of a stack. As such, successors of a node are always added at the end of the queue, and therefore a shallow node is always selected for expansion.

However, BFS also suffers from an important drawback in our context: BFS has the potential of needing many iterations to find a first solution to the problem. Indeed, assume once again that a node has at most b neighbours, and imagine that the shortest path in the search graph has k edges. Then, *all* nodes that are reachable in less than k edges are added to *Open* which means that $O(b^k)$ iterations are needed before such a path is found.

3.3 Issues with BFS and DFS

Both BFS and DFS have issues with some queries. Consider for example the following query, starting with M. Stonebraker:

`(dc:creator- · dc:creator)* · dc:creator- · rdfs:label,`

that is, intuitively we want to retrieve the papers written by a co-author of M. Stonebraker, or by a co-author of some of his co-authors, and so on. Furthermore, take the realistic assumption that there are hundreds of IRIs connected via `dc:creator-` with M. Stonebraker (indeed, Stonebraker's DBLP entry, as of the writing of this paper, contains 298 papers).

Let us now focus on what BFS does with this query. It will first dereference the IRI for Stonebraker, adding the IRI of each of his 298 papers to *Open*. Then, it will dereference each of these IRIs, which requires 298 requests over the network. When each of these IRIs are expanded, we add to *Open* the co-authors of Stonebraker. Only after all the IRIs of Stonebraker's papers are expanded, it will expand the IRI of one Stonebraker's coauthors, and, immediately will find a solution path.

Waiting for 298 HTTP requests before obtaining the first answer is not sensible: in this case only three requests are needed to find

the first answer. Indeed, starting from Stonebraker's IRI, we just choose the IRI for *one* of Stonebraker's papers, we expand such an IRI, from where we choose the IRI of *one* of his co-author's. After dereferencing the latter IRI we find the first solution.

DFS has different yet important issue with this very same query. To find a first solution, DFS actually does the minimum amount of effort, dereferencing the minimum number of IRIs, as described above. The issue appears when looking for the answers that follow the first. Because the focus of DFS is depth, when executed over DBLP, the 5th answer of our query has length 6, the next 4 answers have length 12, and the following ones 32 and up. This implies that DFS will incur in more computation time to retrieve these answers, as well as more http requests. Moreover, returning these lengthy paths first does not seem intuitively right, as we normally want to display simpler, shorter paths first. Indeed, it is not hard to contrive examples in which the length of solutions increases much faster than in our examples, even when many shorter solutions exist.

What we need is a good balance between execution time and solution quality. In our example, a sensible way to proceed would be to take the IRI for the first paper, look at its authors, list them, and then proceed likewise with the second paper. This balance has been studied in the area of Heuristic Search, for many years, producing algorithms that are guided by a heuristic function h , that is such that $h(s)$ estimates the cost of a path from s to a goal node. Expansions are significantly (usually, exponentially) reduced as one improves the "quality" of h . Next we discuss the challenges of using of using heuristic search over Linked Data.

4 AI SEARCH TO THE RESCUE

A^* is one of the most simple and well-studied heuristic algorithms capable of solving path search problems like the one we described in the previous sections. In this section we study how to apply it to the problem of answering property paths over the Web.

The main difference between A^* and the algorithms described earlier is that the search frontier is a priority queue where the priority is given by $f(s)$, a function that estimates the cost of a solution that passes through s [16]. A high-level description of A^* is as follows. At initialisation, the initial node is added to the *Open* queue. A^* now repeats the following loop: first, it extracts a node with the highest priority from *Open*. It returns s if it is a goal state; otherwise, it expands s to obtain its neighbours, adds them to *Open* and continues execution. Next we give a formal description of A^* .

The search graph of A^* is *implicitly* described by (1) a start node s_{start} ; (2) a set of actions *Act*; (3) a partial successor function *Succ*, such that *Succ*(a, s), if defined, returns a set S of successor nodes; (4) a goal condition, which is a boolean function over nodes—*goal nodes* are those nodes for which this function returns true; (5) a non-negative cost function c between successor nodes. The objective of the algorithm is to find a path from s_{start} to a goal node.

An additional argument required by A^* is a heuristic function h , which is a non-negative function over nodes such that $h(s)$ is an estimate of the cost of a path that starts in s and reaches a goal node. The heuristic is key to the performance of A^* . An empirically well-known fact is that as h is more accurate, time savings can be very big because expansions are significantly reduced. It can be proven that when h is *admissible*, that is, for every s it holds that

Algorithm 2: The A* Algorithm

```

1 procedure A*
2   Closed ← empty set
3   Open ← empty priority queue ordered by f attribute
4    $g(s_{start}) \leftarrow 0$ 
5    $f(s_{start}) \leftarrow h(s_{start})$ 
6   Insert  $s_{start}$  into Open
7   while Open ≠ ∅ do
8     Extract s from Open
9     if s is a goal node then
10      return s or add s to list of solutions
11    Expand(s)
12 procedure Expand(s)
13   Insert s into Closed
14   for each a in Act such that Succ(a, s) is defined do
15     for each s' in Succ(a, s) do
16        $t \leftarrow s'$ 
17       if t is not in Seen then
18         Add t to Seen
19          $g(t) \leftarrow \infty$ 
20        $cost \leftarrow g(s) + c(s, t)$ 
21       if  $cost < g(t)$  then
22          $g(t) \leftarrow cost$ 
23          $f(t) \leftarrow g(t) + h(t)$ 
24          $parent(t) \leftarrow \langle s, a \rangle$ 
25         if t is a goal and  $f(t) \leq f(top(Open))$  then
26           return t or add t to list of solutions
27         if  $t \notin Open$  then Insert t in Open
28         else Update priority of t in Open

```

$h(s)$ does not overestimate the cost of any path from s to a goal node, then A* finds a minimum-cost path from s_{start} to a goal node.

Algorithm 2 shows a pseudo-code for A*. The priority function is defined as $f(s) = g(s) + h(s)$, where h is the heuristic function defined above and $g(s)$ is the cost of the best path found so far towards s . In an implementation of A*, a hash table is used to store nodes that have been generated in an expansion (cf. Line 18), and *parent*-, *g*-, *h*-, and *f*- values are stored as properties of s .

A final and important observation is that A* can be easily modified to return a sequence of answers, instead of a single one. In this case, we simply modify the return statement in Line 10 by something that adds s to a list.

Using A* for computing property paths. Let us show how we use A* to solve PP.OVER.THE.WEB. That is, given as inputs a property path e and a starting IRI u , we look for all v such that $(u, v) \in \llbracket e \rrbracket_{G_{Web}}$. Let $A_e = (Q_e, I^\pm, q_e^0, F_e, \delta_e)$ be the automata over I^\pm that is equivalent to e . Recall that (see Lemma 2.3 and Section 3) we can reduce this problem to searching for all nodes (v, q_f) , for an IRI v and a state $q_f \in F_e$, over the graph $G_P \subset G_{Web} \times A_e$ such that there is a path from (u, q_e^0) to (v, q_f) . In turn, this problem can be seen as an A* description where (1) the start node s_{start} is (u, q_e^0) ; (2) the set Act of actions corresponds to IRIs in I^\pm ; (3) the partial successor function Succ corresponds to the edges of $G_{Web} \times A_e$, that is, if $s = (u, q)$, we say $(u', q') \in Succ(a, s)$ if both $(u, a, u') \in adoc(u)$, and (q, a, q') is a transition in A_e , or if both $(u', a, u) \in adoc(u)$, and (q, a^-, q') is a transition in A_e ; (4) a node $s = (v, q)$ is a goal if $q \in F_e$; and (5) the cost function is 1 for each pair of nodes connected by Succ.

There is an important subtlety that distinguishes our algorithm from classical A* applications. Just as in the case of BFS and DFS,

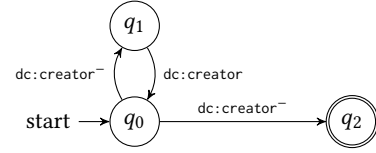


Figure 1: An automaton finding papers of the co-authors of M. Stonebraker.

the successors of (u, q) must be obtained by dereferencing an IRI (using, for example, the function *Neighbours* from Algorithm 1). This again means that the most costly operation is the expansion of new successor nodes, and as such any implementation of A* must try their best to find a way of reducing this bottleneck. We explain how to do this in Section 5.1. But before, let us see how to choose a good heuristic function in our scenario.

4.1 A Heuristic for Navigational Queries

Heuristic functions are essential for the performance of A*. We also want A* to be optimal, so our heuristic must be admissible, that is, it should not overestimate the cost of path to a goal node.

Let A be an automaton over I^\pm . Our heuristic for this problem is defined as follows: for all nodes (v, q) over $I \times Q_e$, where Q_e are the states of A_e , we define $h((v, q))$ as the minimum distance from q to a final state of A_e (and as ∞ if no path from q to a final state exists). To illustrate our heuristic consider Figure 1, corresponding to the automaton of the query for papers of the coauthor reach of M. Stonebraker introduced in Section 3.3. Then we define $h(u, q_1) = 2$, $h(u, q_0) = 1$, and $h(u, q_2) = 0$, for every $u \in I$. Usually $h(u, q)$ is implemented as a simple lookup in a table. Given an automaton A we denote the heuristic defined as described above by h_A .

Our heuristic h_{A_e} is admissible for each property path e , as long as A_e is the minimum NFA for e . To see this, note that the minimum number of actions required to reach a goal node from node (u, q) cannot exceed the number of edges of a shortest path between the automaton state q and a final state. This is because each successor (u', q') of node (u, q) must be such that there is an edge between q and q' in the automaton's graph.

4.2 Theoretical Guarantees

A well-known property of A* is that it finds cost-optimal (i.e., shortest) paths. Here we provide an optimality result of the same sort. Now, because the function $adoc(u)$ is not necessarily guaranteed to return all triples containing u , we cannot show optimality over the entire Web, but rather only over the graph we have already discovered, that we denote by G_{A^*} .

Formally, given an execution of A*, the labelled graph $G_{A^*} \subset G_{Web} \times A_e$ contains the edge (s, a, s') iff (1) $s \in Closed$, and (2) $s' \in Succ(a, s)$. Notice that this correspond to the product of A_e with the graph that contains all triples present in any of the documents that have been retrieved so far in our computation. The following is an optimality result both for BFS and A* run with our property-path heuristic.

THEOREM 4.1. *Let G_{A^*} be defined as above from a run of A* that has returned N answers with either $h = h_{A_e}$ or $h = 0$. Let π_k be path found to the k -th solution found by A*, for any $k \in \{1, \dots, N\}$.*

Furthermore, let c_k be the length of the k -th shortest path from s_{start} to any goal state over G_{A^*} . Then the cost of π_k is c_k .

SKETCH. Let $G_{A^*}^i$ denote G_{A^*} right after the i -th solution has been returned. We prove by induction that the i -th solution found by A^* would be the first solution found by A^* if we were to mark as non-goals all solutions found prior to the i -th solution. Now we use the fact that the heuristic is admissible and thus the solution found is the i -th optimal over $G_{A^*}^i$. \square

In practice, as we see later on, BFS runs slower than A^* with our heuristic. Interestingly, we can prove that A^* is better in the sense that BFS has to expand at least as many nodes as A^* .

THEOREM 4.2. *Let (u, e) be an IRI and a property path. Then every node expanded by A^* , used with h_{A_e} , is also expanded by BFS.*

PROOF. We observe that $h_{A_e}(s) > 0$ for every non-goal state s . The result now follows from Theorem 7 in [31]. \square

5 OPTIMISING QUERY EXECUTION

In this section we provide several optimisations to the base algorithms presented in Section 3 and Section 4. We start by describing how parallel expansions can be used in order to reduce the execution times of our search algorithms. We then explain what are the current shortcomings of the Linked Data infrastructure and propose a way to overcome them using endpoints. Finally, we discuss a way of tweaking the heuristic used in A^* in order to both avoid unnecessary network requests and find answers sooner.

5.1 Parallel Expansions

All of our search algorithms function in such a way that they select a set of nodes which will serve as the starting point in the next iteration, and then start the search from these nodes one by one. An issue with this is that a request over the network—which on average takes more than a second—is needed per each dereference.

Instead of expanding one node at a time, our algorithms can benefit greatly from expanding multiple ones in parallel. More specifically, we modify the algorithm to extract up to k of nodes that could be at the top of the *Open*, and expand them in parallel. k is now a parameter of the algorithms which can be understood as a *degree of parallelism*. To obtain k -BFS and k -DFS, we modify Algorithm 1 such that Line 16 deals with up to k top-valued nodes from *Open*, and neighbours are computed for them in parallel. Similarly, k - A^* is obtained by extracting up to k nodes with the highest f -values from *Open* in Line 8 of Algorithm 2, and expanding them all in parallel in line 11. In all 3 algorithms, after all successors are computed, we add them all together to *Open*, in the same order that we would have, had the nodes been expanded sequentially. It is then not hard to see that optimality (Theorem 4.1) still holds for k - A^* (and k -BFS).

In Section 6 we show that, depending on the degree of parallelism, the computation is sped up tenfold in some instances.

5.2 Using The Endpoint Infrastructure

The evaluation algorithms presented in previous sections rely on the dereferencing mechanism of Linked Data and work under the assumption that when a specific IRI is dereferenced, we obtain all the triples mentioning such an IRI which reside on the server

we are using. Unfortunately, it was shown repeatedly that this is generally not the case when working with Linked Data [22, 23], which can lead to incomplete answers since many triples containing the dereferenced IRI might not be returned. This is particularly problematic when working with inverse links, as it is estimated that publishers include only about a half of the triples where the requested IRI appears as the object [23].

Many Linked Data providers also set up public SPARQL endpoints where users can query the dataset, so we can partially alleviate the lack of Linked Data infrastructure by relying on public SPARQL endpoints together with Linked Data. When evaluating property paths over Linked Data, we combine the two approaches and, each time we dereference an IRI, we also query the appropriate endpoint in order to obtain the triples mentioning the said IRI. Furthermore, we query the endpoint only asking for links in the appropriate direction. For instance, if our property paths needs to traverse the author edge forwards starting from an IRI $start$, we ask the query `SELECT ?x WHERE {start author ?x}` to the appropriate endpoint and similarly for the backwards edges.

5.3 Minimising Network Requests

We have argued above that dereferencing is an expensive operation. When A^* is modified to find multiple answers, as we proposed above (i.e., by simply adding solutions to a list), some expansions may be carried out sooner than we would want, leading to unnecessary dereferencing. Indeed, our heuristic assigns the value 0 to any node of the form (u, q_f) with q_f a final state (because the distance to a final state is 0). Assuming q_f has outgoing transitions, the standard A^* algorithm would prioritise the expansions of those nodes over any other node with the same f value, an operation that intuitively would take us farther from the goal.

We can postpone these expansions by using a slightly different heuristic, defined as follows. Let $A = (Q, \Sigma, q_o, F, \delta)$ be an NFA. The *pathmax distance* $\hat{d}(q, q')$ between two states is defined as $\hat{d}(q, q') = 1 + \min_q |\delta(q, a)$ is defined $d(q, q')$, if $\delta(q, a)$ is defined for at least some $a \in \Sigma$, or $\hat{d}(q, q') = \infty$ otherwise; and where d is the usual graph distance between q and q' . Then the pathmax heuristic \hat{h}_A with respect to A is defined as $\hat{h}_A((u, q)) = \min_{q_f \in F} \hat{d}(q, q_f)$, that is, the minimum pathmax distance from q to any final state of A . This is a standard technique used by search algorithms in which a node may have to be re-expanded [25, 33]. Interestingly, one can see that the pathmax distance \hat{d} coincides with the usual distance in all states of the automata except for the final states. We avoid early re-expansion of nodes with final states because the heuristic for them now corresponds to 1 plus the minimum of the heuristic value of the neighbours of this state.

6 EXPERIMENTAL EVALUATION

In this section we evaluate how an implementation of A^* algorithm performs when executing property path queries over a real Web environment. To establish a baseline, we also compare the performance of A^* with BFS and DFS, the only other algorithms proposed so far in the literature. We begin by presenting our experimental setup, the queries, and then compare how A^* fares against BFS and DFS, showing that A^* outperforms the other two algorithms on a regular basis. Next we investigate the impact of parallel requests on

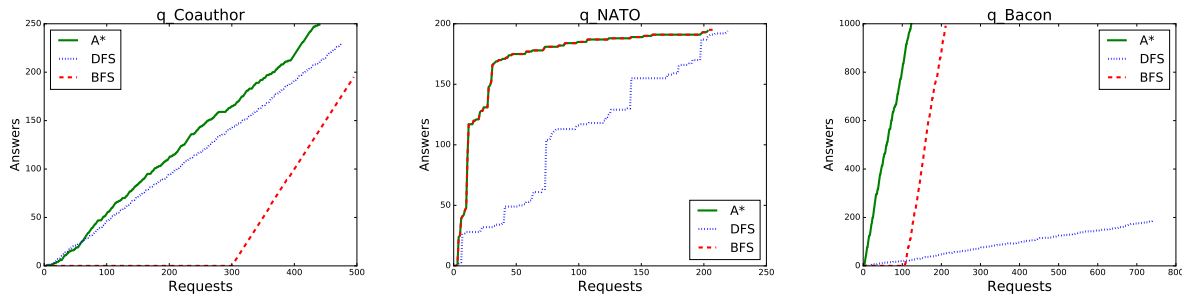


Figure 2: A* minimises the requests needed to obtain answers of $q_Coauthor$, q_NATO and q_Bacon

our algorithms. As we see, adding parallelism reduces total runtime for all three algorithms, with A* remaining the most consistent. Interestingly, all algorithms tend to look more alike as more and more parallel requests are allowed. Finally, we discuss some real-world examples of the paths retrieved by our algorithms.

6.1 Experimental Setup

We selected 11 navigational queries that are inspired by previous benchmarks (see e.g. [14, 32]). These queries are representative of several different features of property paths, ranging from easy, fixed-depth ones to queries using multiple star operations which are much harder to evaluate. Our queries target one or more of the following Linked Data domains: YAGO, a huge knowledge base extracting data from Wikipedia and various other sources [29]; DBPedia [6], one of the central datasets of the Linked Data initiative that also originates from Wikipedia; Linked Movie Database, the best known semantic database for movie information [27]; and the Linked Data domain of DBLP. Our implementations will always use the optimisation techniques presented in Section 5.2 and Section 5.3, while we assess the benefits of parallelism (Section 5.1) separately. As an example, below are 3 of the 11 queries we use. For complete details of the queries, results of all our runs, and implementation of our algorithms, please refer to our online appendix [1].

q_Coauthor: The property path $(coauthor^- \cdot coauthor)^*$ in the DBLP dataset, starting from the IRI of M. Stonebraker. This property path looks for the IRI of all authors that are related to M. Stonebraker on DBLP, by a co-authorship path of arbitrary length.

q_NATO: A property path that selects all places that host an entity dealing with a NATO member state, according to YAGO.

q_Bacon: A property path that looks for the IRIs of actors having a finite Bacon-number¹, and that navigates using links and/or IRIs present in any of YAGO, DBPedia or Linked Movie Database. This is an interesting query, as currently the only way to evaluate it is by means of our Linked Data approach (see [8] for a discussion).

To assess our algorithm we use two indicators: the number of HTTP requests made to compute a fraction of the answers, and the time needed to compute them. In both cases we want to minimise the number of requests, or the amount of time needed to produce the answers. We note that the number of requests is a much better indicator on how the algorithm works: because HTTP requests take considerably more time than all the other operations, the total

time of computing our queries is essentially given by the number of requests performed by the algorithm. This also rules out the dependence on parameters which we have no control over, such as the Internet traffic, or the availability of servers providing us with data. Thus, by focusing on requests we ignore latency differences that may persist even after taking several runs of the same query.

All experiments were run without an access to the data locally, relying solely on the Web infrastructure to retrieve the data needed at each step of the computation. The experiments were run on a Manjaro Linux machine with a i5-4670 quad-core processor and 4GB of RAM. To avoid flooding servers with requests we only ran our search until we either found 1000 answers, retrieved more than 100 000 triples from the server, or reached a 10-minute time limit. Each experiment was ran 10 times, and since the results were largely equivalent, we report the numbers of the latest execution. The source code for running the experiments is available at [1].

6.2 Heuristic Search Against BFS and DFS

The general conclusion of our experiments is that A* both requires fewer requests and is faster than BFS and DFS. Before reporting our results in full, let us examine the runs of queries $q_Coauthor$, q_Bacon and q_NATO presented above. Figure 2 shows the number of requests needed to compute a fraction of the total answers available for these queries. In particular, we see that both A* and DFS are the best choice for the query $q_Coauthor$, because they produce more answers using fewer HTTP requests (even though the quality of the answers produced by A* is arguably better – see below). On the other hand, BFS requires around 300 expansions to start producing answers, which results in a much slower throughput altogether. Next, both A* and BFS are the best choice for the query q_NATO . This is again expected, because this query requires less navigation and more shallow exploration. It is interesting to see that in this case A* really simulates the optimal BFS search. On the other hand, DFS wastes a lot of time exploring long paths and obtaining “deep” answers. Finally, in the case of q_Bacon , A* is shown to strictly beat both BFS and DFS. In the case of BFS, this is mostly because A*’s heuristic allows a finer control on which links to explore, and the main detractor for DFS is that it starts exploring initial links which often require many requests before encountering a solution.

Full results. For reasons of space, we cannot report the remaining 8 experiments with the same detail, so instead we do the following. For each query, we analyse the complete behaviour of the answers

¹An actress has Bacon number 1 if she acted in the same movie as Kevin Bacon, and Bacon number n if she acted with someone with Bacon number $n - 1$.

vs. request and answers vs. time curves. We say that an algorithm *dominates* the others if it is such that it returns at least as many answers as any other for 80% of the range of requests (or time) for which we evaluate them. For example, in Figure 2 we see that A^* dominates the other algorithms for queries **q_Coauthor** and **q_Bacon**, while in the case of **q_NATO** both A^* and BFS dominate. The total number of times each algorithm dominates (out of 11 queries) is shown below, for both the answers vs. request and answers vs. time curves (full details are found in our online appendix [1]). Once again, A^* remains the most consistent option.

Measure	A^*	BFS	DFS
Requests v/s Answers	11	3	4
Time v/s Answers	11	3	4

6.3 The Effect of Parallel Requests

Next, we test the effect of allowing parallel requests in our algorithms, as presented in Section 5.1. This optimisation goes a long way into tackling the slow latency of Web requests, one of the main problems of querying over the HTTP protocol. Indeed, HTTP requests are such an important bottleneck in our algorithm that allowing parallel requests essentially means parallelising the entire algorithm. Issuing parallel requests also soften up high latency pockets or temporary network problems. Moreover, we can also expect the algorithms to be accelerated even further when the number of allowed requests is increased, simply because more requests essentially means more parallel instances of our algorithm and even more softening power. The other interesting observation is that, as we allow more parallel requests in our algorithms, all of A^* , BFS and DFS start to look alike, and in fact it is easy to see that all three algorithms are essentially equivalent in the limit where we issue an infinite number of requests at the same time.

To empirically test these observations, we issued new live runs of the 11 queries described in the previous sections, but this time using parallel versions of A^* , BFS and DFS. To see the impact on the number of parallel threads allowed, we report experiments with a maximum of 10, 20, and 40 parallel threads. Before reporting the full results, let us start with comparing the results of the algorithm with no parallelism against the one with 20 parallel threads. Figure 3 shows the time needed to compute the answers of query **q_Coauthor**, for all three algorithms on their non-parallel version and on their parallel version with a maximum of 20 threads. As we see, the time needed to compute the same amount of answers decreases by almost tenfold in all three cases. Moreover, the parallel version of BFS now behaves almost as A^* and DFS when computing the first 300 answers (it then reaches a stalemate because all shallow answers have already been discovered).

Full results. As expected, the time taken to compute answers decreases drastically (the behaviour is the same as for **q_Coauthor**). Perhaps more interestingly, we focus on how algorithms change when more parallel threads are allowed. In order to do this, we repeat the same reports made in the previous subsection, but this time for different levels of parallelism. More precisely, for each of our 11 queries and 4 different thread counts we report which of A^* , BFS or DFS dominates in the time needed to compute the answers. As we see as more parallelism is allowed into the algorithms, both BFS and DFS start becoming more competitive compared to A^* .

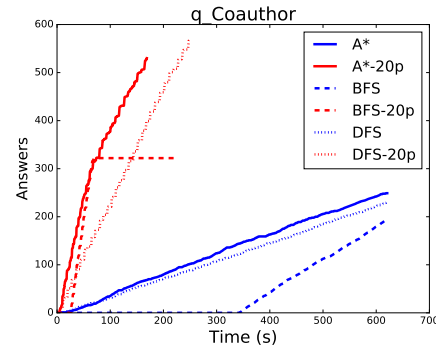


Figure 3: Answers over time on **q_Coauthor**. The parallel versions (in red) are much faster than the non-parallel ones.

```

dblpAuthor:Michael_Stonebraker
^dc:creator dblpPub:conf/acm/MuthuswamyKZSPJ85
dc:creator dblpAuthor:Matthias_Jarke
rdfs:label "Matthias Jarke"

dblpAuthor:Michael_Stonebraker
^dc:creator dblpPub:conf/dbvis/AikenCLSSW95
dc:creator dblpAuthor:Mybrid_Spalding
rdfs:label "Mybrid Spalding"

dblpAuthor:Michael_Stonebraker
^dc:creator dblpPub:conf/vldb/StonebrakerABCCFLMOORTZ05
dc:creator dblpAuthors:Adam_Batkin
rdfs:label "Adam Batkin"

```

Figure 4: Paths for the 10th, 50th, and 200th answers found by A^* on **q_Coauthor**.

Max parallel calls	A^*	BFS	DFS
1	11	3	4
10	7	3	3
20	7	3	3
40	6	4	5

6.4 Returning paths

So far we have only talked about finding pairs of nodes that form the answer of a property path query, as dictated by the SPARQL standard. However, our search algorithms can also be used to compute the entire *path* between two nodes, and in fact we can get them at a very marginal cost: we already need to keep track of all the expansions, so we can produce paths simply by returning the IRIs corresponding to each of the requests made by our algorithm.

Paths can be used as a justification for the answers, or to continue extracting more information afterwards. In the case of queries over Linked Data, we can even use the paths of queries to gather information about the structure of the Web itself. For these reasons returning paths is a very sought-after functionality of graph query languages, and is present for example in the popular Neo4j engine [34]. Unfortunately, a language capable of returning (all) paths, or even (all) simple paths, is bound to be very complicated to evaluate [5, 28], and this is the reason why the SPARQL standard does not include such a functionality. In our case we have a natural workaround for this issue, as our search focuses on shortest paths, which are known to be easier to evaluate than simple paths.

As an example of the usefulness of paths, it was by analysing paths that we inferred that A^* normally produces better answers than DFS (because the paths are shorter). As an illustration, Figure 4

presents paths witnessing the answers 10, 50, and 200 of a run of the query **q_Coauthor** with A^* . From the query itself all that we can say is that these three researchers are connected to M. Stonebraker by a coauthorship path of arbitrary length. However, by looking at the paths we now know that they are direct coauthors. On the other hand, the length of paths retrieved by DFS are going to be much higher. For one run of **q_Coauthor** with DFS the lengths of the answers 10, 50 and 200 were respectively 14, 74, 312.

7 CONCLUSIONS AND FUTURE WORK

This paper presents the first fundamental study of the problem of computing property paths over the Web. We showed how to cast query answering as an AI search problem, and provided an optimal algorithm based on the classical A^* algorithm. We provide strong theoretical and practical evidence that A^* is a better alternative than both BFS and DFS in the context of Linked Data, and this can be sped up even further by allowing parallel execution threads.

In terms of future work, we identify three main challenges we plan on tackling.

Using triple pattern fragments. As noted in Section 5, there are some issues with the Linked Data infrastructure; most notably, it does not provide all the information one would expect when dereferencing IRIs. While it is possible to alleviate this issue by using endpoints, since their uptime can be erratic, it was recently suggested that a more lightweight infrastructure of triple pattern fragments [36] would be more appropriate for the task. In the future we plan to test how using triple pattern fragments affects the performance and accuracy of our algorithms when compared to the standard endpoint infrastructure.

Answering NautiLOD and LDQL queries with A^* . NautiLOD [13] is a traversal-based language proposed as an option to SPARQL when querying Linked Data, in which one has more finer control on how is the Web going to be traversed. In the same spirit, LDQL [20] is another language aimed at controlling how data is to be retrieved, albeit much less powerful than NautiLOD. The interesting observation is that we can also cast the query evaluation problem for these languages as a search problem, and thus A^* should also provide optimal query answering algorithms. In fact, the algorithm proposed in [13] is essentially what we define here as k -DFS, so one can naturally suspect that A^* should provide a better behaviour.

A^* in local computations. Although we based our investigation in the context of Linked Data, there is some evidence that our approach might have potential in the classical setting where data is available locally. The main reason is the fact that the currently available property path evaluation algorithms demand a lot of resources, especially when dealing with property paths that use the Kleene star operator, and current systems cannot easily cope with these requirements [8]. On the other hand, we have seen that the memory usage of an A^* -based algorithm is directly dependant on the amount of answers that need to be computed, and each answer requires an almost negligible amount of additional memory. This suggests that, in those cases when we do not need all the answers, an approach based on A^* might be a better option.

REFERENCES

- [1] 2016. <http://dvrqoc.ing.puc.cl/navigation/>. (2016).
- [2] S. Abiteboul, P. Buneman, and D. Suciu. 1999. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufman.
- [3] Serge Abiteboul and Victor Vianu. 1997. Queries and Computation on the Web. In *ICDT '97*. 262–275.
- [4] Renzo Angles and Claudio Gutiérrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1 (2008).
- [5] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. 2012. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *WWW*.
- [6] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. *Dbpedia: A nucleus for a web of open data*. Springer.
- [7] Pablo Barceló Baeza. 2013. Querying graph databases. In *PODS 2013*. 175–188.
- [8] Jorge Baier, Dietrich Daroch, Juan L. Reutter, and Domagoj Vrgoč. 2016. Property Paths over Linked Data: Can It Be Done and How To Start?. In *COLD@ISWC 2016*.
- [9] Tim Berners-Lee, Christian Bizer, and Tom Heath. 2009. Linked data-the story so far. *Int. Journal on Semantic Web and Information Systems* 5, 3 (2009), 1–22.
- [10] Blazegraph 2016. Blazegraph. <https://www.blazegraph.com/>. (2016).
- [11] D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. 2000. Containment of conjunctive regular path queries with inverse. In *KR 2000*. 176–185.
- [12] I. Cruz, A.O. Mendelzon, and P. Wood. 1987. A graphical query language supporting recursion. In *SIGMOD '87*. 323–330.
- [13] Valeria Fionda, Giuseppe Pirrò, and Claudio Gutierrez. 2015. NautiLOD: A Formal Language for the Web of Data Graph. *TWEB* 9, 1 (2015), 5:1–5:43.
- [14] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. 2013. Sparqling kleene: fast property paths in RDF-3X. In *GRADES 2013*. 14.
- [15] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 query language. *W3C* (2013).
- [16] Peter E. Hart, Nils Nilsson, and B. Raphael. 1968. A formal basis for the heuristic determination of minimal cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968).
- [17] Olaf Hartig. 2012. SPARQL for a Web of Linked Data: Semantics and computability. In *The Semantic Web: Research and Applications*. Springer, 8–23.
- [18] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. 2009. *Executing SPARQL queries over the web of linked data*. Springer.
- [19] Olaf Hartig and M. Tamer Özsu. 2016. Walking Without a Map: Ranking-Based Traversal for Querying Linked Data. In *ISWC 2016*. 305–324.
- [20] Olaf Hartig and Jorge Pérez. 2015. LDQL: A Query Language for the Web of Linked Data. In *ISWC 2015*. Springer, 73–91.
- [21] Olaf Hartig and Giuseppe Pirrò. 2015. A Context-Based Semantics for SPARQL Property Paths Over the Web. In *ESWC 2015*. 71–87.
- [22] Aidan Hogan and Claudio Gutierrez. 2014. Paths towards the Sustainable Consumption of Semantic Data on the Web. In *AMW 2014*.
- [23] Aidan Hogan, Jürgen Umbrich, Andreas Harth, Richard Cyganiak, Axel Polleres, and Stefan Decker. 2012. An empirical survey of Linked Data conformance. *J. Web Sem.* 14 (2012), 14–44.
- [24] Jena 2015. Apache Jena Manual. <http://jena.apache.org>. (2015).
- [25] Richard E. Korf. 1993. Linear-Space Best-First Search. *Artificial Intelligence* 62, 1 (1993), 41–78. DOI : [http://dx.doi.org/10.1016/0004-3702\(93\)90045-D](http://dx.doi.org/10.1016/0004-3702(93)90045-D)
- [26] Egor V Kostylev, Juan L Reutter, Miguel Romero, and Domagoj Vrgoč. 2015. SPARQL with Property Paths. In *ISWC 2015*. Springer, 3–18.
- [27] LMDb. Linked movie database. <http://linkedmdb.org/>. (????).
- [28] Katja Losemann and Wim Martens. 2012. The complexity of evaluating path expressions in SPARQL. In *PODS 2012*. 101–112.
- [29] Farzaneh Mahdisoltani, Joanna Biega, and Fabian Suchanek. 2014. Yago3: A knowledge base from multilingual wikipeas. In *CIDR*.
- [30] Frank Manola, Eric Miller, and Brian McBride. 2014. RDF 1.1 primer. <https://www.w3.org/TR/rdf11-primer/>, *W3C* (2014).
- [31] Judea Pearl. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [32] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. 2015. Recursion in SPARQL. In *ISWC 2015*.
- [33] Stuart J. Russell. 1992. Efficient Memory-Bounded Search Methods. In *ECAI '92*. 1–5.
- [34] The Neo4j Team. 2016. The Neo4j Manual v3.0. <http://neo4j.com>. (2016).
- [35] Jürgen Umbrich, Aidan Hogan, Axel Polleres, and Stefan Decker. 2015. Link traversal querying for a diverse Web of Data. *Semantic Web* 6, 6 (2015), 585–624.
- [36] Ruben Verborgh, Olaf Hartig, Ben De Meester, Gerald Haesendonck, Laurens De Vocht, Miel Vander Sande, Richard Cyganiak, Pieter Colpaert, Erik Mannens, and Rik Van de Walle. 2014. Querying Datasets on the Web with High Availability. In *ISWC 2014*. 180–196.
- [37] Virtuoso 2015. Open Link Virtuoso. <http://virtuoso.openlinksw.com/>. (2015).
- [38] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query Planning for Evaluating SPARQL Property Paths. In *SIGMOD 2016*. 1875–1889.