Counting to k or how SPARQL1.1 Property Paths Can Be Extended to Top-k Path Queries*

Vadim Savenkov
Vienna University of Economics and Business
Welthandelsplatz 1
Vienna, Austria 1020
vadim.savenkov@wu.ac.at

Qaiser Mehmood
Insight Centre for Data Analytics, NUI Galway
P.O. Box 1212
Galway, Ireland 43017-6221
qaiser.mehmood@insight-centre.org

Jürgen Umbrich
Vienna University of Economics and Business
Welthandelsplatz 1
Vienna, Austria 1020
juergen.umbrich@wu.ac.at

Axel Polleres
Vienna University of Economics and Business and
Complexity Hub Vienna
Vienna, Austria
axel.polleres@wu.ac.at

ABSTRACT

While the volume of graph data available on the Web in RDF is steadily growing, SPARQL, as the standard query language for RDF still remains effectively unusable for the basic task of finding paths through the graph between selected nodes. Property Paths, as introduced in SPARQL 1.1 are unfit for this purpose, as they can only be used to test path existence. More expressive features, such as counting distinct paths between two nodes, have been shown highly intractable in the worst case, in particular in graphs with high degree of cyclicity. Still, practical use cases demand a solution for path retrieval even when the total number of paths is prohibitively large. A common approach is to ask not for all, but only for the k shortest paths. In this paper, we extend SPARQL 1.1 property paths in a manner that allows to compute and return the k shortest paths matching a property path expression between two nodes. For RDF graphs in the compact HDT format, we evaluate or algorithm for top k shortest paths showing that a relatively simple approach works (in fact, more efficiently than other, more complex algorithms in the literature) in practical use cases.

CCS CONCEPTS

•Theory of computation →Shortest paths; •Information systems →Network data models; Resource Description Framework (RDF);

KEYWORDS

k shortest paths, SPARQL, querying RDF data, routing

*we thank Marcelo Arenas et al. [4] for inspiring our title.

ACM Reference format:
Vadim Savenkov, Qaiser Mehmood, Jürgen Umbrich, and Axel Polleres. 2017. Counting to k or how SPARQL1.1 Property Paths Can Be Extended to Top-k Path Queries. In *Proceedings of Semantics2017, Amsterdam, Netherlands, September 11–14, 2017,* 7 pages.
DOI: 10.1145/3132218.3132239

1 INTRODUCTION

RDF data on the Web is starting to form a constantly growing, labelled graph, sometimes called Linked Data, sometimes called the "Web of Data", but anyway justifying the claim that a tangible portion of a "Semantic Web" has become a reality. In order to query RDF and Linked Data graphs, SPARQL as the standard query language is typically the tool of choice, but several omissions still make it far from perfectly fit for the task. For instance, while SPARQL 1.1 introduced queries testing path existence between two nodes via a feature called property paths, the language still falls short in both counting and retrieving the paths. In fact, there is a history behind this, as the seminal paper by Perez et al. in 2012, warned us not to "count beyond the Yottabyte", i.e. the paper showed not only that – at that time current – SPARQL engines implemented property paths in an inefficient manner, but also that a query language feature that allowed to return (the number of) *all* property paths would easily become infeasible even in relatively small graphs due to potentially double-exponential number of solutions.

The existential semantics which the authors proposed to the rescue (and which became eventually a part of the final SPARQL 1.1 standard), has its limitations in practice where not only the existence but the paths themselves are of interest. For instance, in professional social networks, typically everybody is (somehow) transitively connected to anyone, but we are interested in the k most promising connections to get introduced to some peer.

The SPARQL property path query

Listing 1: Property Path query in SPARQL1.1

```
SELECT * WHERE {:me foaf:knows+ :bob}
```

would not help here, in fact, it would simply return an empty binding. Likewise, for most practical routing applications the mere reachability test is not sufficient. On the other hand, enumerating all paths between given points is seldom required either: rather, only the top most relevant (e.g., shortest) paths need to be found.

A particular interest for path queries comes from the bioinformatics domain where the volume of semantic data is constantly growing [5]. For instance, in the area of *cancer genomics* experts often need to discover relevant associations between biological and genetic entities such as diseases, drugs, genes, pathways etc. requiring efficient querying mechanisms [15, 16]. It is typical in medical research that multiple genetic features, their effects like diseases, and treatments to those diseases are studied together, often in a larger context such as medical history. One of the key challenges in cancer genomics – a cornerstone of precision medicine – is to discover gene-disease-drug associations which might be relevant for developing new treatment methods. Such associations essentially correspond to paths in semantic databases. It is exactly the paths that represent the associations and that need to be discovered, however SPARQL 1.1 does not provide adequate means for returning paths, only for testing reachability.

The lack of support of path enumeration in SPARQL has long been recognized as an issue. There have been several surges of interest to the topic of implementing path queries in the context of SPARQL in the past decade which we briefly survey in Section 6. In particular, the European Semantic Web Conference in 2016 defined a challenge where the goal was exeactly to find the "Top-K Shortest Path in Large Typed RDF Graphs"[21]. Only a handful of entries [12, 14, 23] were submitted where the winners approached the problem with tailored versions special purpose graph algorithms. What the short papers describing these approaches left out, was a systematic extension of SPARQL. It is surprising and unsatisfactory that still, even with numerous open source triple stores like Virtuoso and Jena, there is still no simple open source solution or extension library for top k-paths problems which one could use and extend.

In this paper, we aim to close exactly this gap: building on our prior work [9] we introduce an extension of SPARQL which allows us to find the top k shortest paths compliant with the property path expression. Using our syntax, the three most promising connections could be obtained with the following query:

Listing 2: Query using the :topk function

```
SELECT ?path WHERE {
?path ppath:topk (:me :bob  3 "foaf:knows+")}
```

Technically, our solution uses a built-in extension mechanism of Jena ARQ and works out of the box with the Jena API [1] without a need to recompile the Jena code or modify its syntax.

In the remainder of this paper, after presenting the preliminaries (Section 2) we will discuss the syntax and semantics of the topk function (Section 3), and provide a simple but functionally complete evaluation strategy based on an efficient indexing with HDT and on the bidirectional breadth-first search (Section 4) with the support of path restrictions via regular expressions. As we can show in our evaluation Section 5, our proposed solution of path computation on a HDT backend offers a very promising performance tackling graphs with tens of millions of triples. In particular, as we mention in a related work survey in Section 6, it significantly outperforms the approaches in [12, 14, 23] and — in terms of property path evaluation — also the popular systems Virtuoso, Blazegraph and Stardog. Concluding remarks and a note on future work are offered in Section 7.

[1]https://jena.apache.org/documentation/query/library-propfunc.html

2 PRELIMINARIES

We assume a simplified RDF model representing graph data as a set of *subject-predicate-object* (spo) triples $\mathcal{I} \cup \mathcal{B} \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$ where $\mathcal{I}$ is a set of globally unique resource identifiers (IRIs), $\mathcal{L}$ is the set of data values, or literals, and $\mathcal{B}$ is the set of placeholders known as *blank nodes*. The triples form a labeled directed graph G. Specifically, each edge of G is a triple (s, p, o) where s is a subject or a *source node*, p is a predicate or an *edge label* and o is an object or a *target node*. We define a *path* in G as an ordered sequence of edges $(e_1, \ldots, e_n)$ such that (i) all edges in the path are *unique* i.e., $e_k \neq e_m$ for all integer $k, m \leq n$, and (ii) adjacent edges have a common incident node: that is, for every i the target node of e_i equals the source node of e_{i+1} if both edges are in a path. The source node of p is the source node of the first edge in it, and the target node of p is the target node t of the last edge in p, in which case p is called a *path from s to t*. If the source and the target node of a path coincide, it is called a *cycle*. According to our definition, paths can contain cycles: the same node can occur multiple times, but repeating edges are not allowed. Using standard graph terminology, our graph G is a *multigraph*, and every cycle in a path needs to be a *trail*, that is a cycle *without repeated edges*. The *length* of the path p is a number of edges in it. By $P_G(s, t)$ we denote the set of all paths from s to t in G, and by $\mathcal{P}_G^{asc}(s, t)$ we denote a sequence of all elements of $P_G(s, t)$ sorted in the order of non-decreasing lengths.

Our definition of path expressions is close to the SPARQL1.1 specification of property paths, up to inverse properties which we currently do not support.[2] Specifically, the following syntax is supported:

$$\begin{aligned} P &:= Q^* \mid Q^+ \mid Q^? \\ Q &:= a \mid ![a_1, \ldots, a_k] \mid (P/P) \mid (P|P) \end{aligned}$$

Here Q denotes an expression without occurrence restrictions, the unary quantifiers $*, +$ and ? respectively denote an unrestricted number of occurrences, at least a single occurrence and at most a single occurrence of a respective pattern Q. $a \in \mathcal{I}$ is an IRI representing a property, the set negation $![a_1, \ldots, a_k]$, for $a_1, \ldots, a_k \in \mathcal{I}$ is satisfied by any single property $b \in \mathcal{I} \setminus \{a_1, \ldots, a_k\}$. Finally, (P/P) defines a sequence of path expressions and $(P|P)$ stipulates that only one of the expressions on the left and on the right of $|$ need to be satisfied. Both binary operators are associative, so we will omit parentheses in sequences of repeated binary operators of the same kind. The top k shortest paths problem is defined below:

Given an RDF graph G, $s \in \mathcal{I}$, $t \in \mathcal{I} \cup \mathcal{L}$, an integer k, and a regular path expression p.
Compute first k elements of $\mathcal{P}_G^{asc}(s, t)$ satisfying p.

For our computation, we will rely on a compressed index representation for RDF graphs called *HDT*: HDT[8] is a compact representation of RDF triples encoding verbose textual IRIs, literals and blank nodes as integers in an optimal way from the information

[2]As a workaround, appropriately named inverse properties could be added to the graph, resulting in at most twice the number of edges.

theoretic point of view (i.e., assigning smallest values to most frequent items in the dataset). HDT supports the three-way indexing of triples, which is crucial for the performance of our algorithm.

3 SPARQL SYNTAX EXTENSION AND IMPLEMENTATION IN JENA

Our goal is to provide an implementation that can be embedded into the existing open source SPARQL engines, in particular Jena ARQ[3] without a need of changing any legacy code. Although not part of any official specification, Jena SPARQL extension interfaces can be seen as a de facto standard which we make use of.

Our implementation of the basic, but robust solution for the k shortest path problem is based on bidirectional search: both the source and the target node of the paths need to be specified like in most routing tasks. The number of desired paths k defaults to 1, and P defaults to the path pattern $(!\ :)^*$. [4] These arguments can either be omitted or must be bound to constants of type integer (k) and string (path expression). The Jena ARQ framework allows SPARQL variables to occur at the input argument positions: our function requires those variables to also occur elsewhere in the query in order to ensure that at the moment the topk function is called, all its input parameters are bound (analogous to the notion of safety in Datalog [22]). Upon each call (for each binding of s and t), the topk function queries the RDF graph again and computes the shortest paths between s and t as explained in the next section. For each input tuple, there are up to k output values each representing a path. In our approach, we opted to encode the found path collections as strings which can be then parsed by the calling applications to extract the properties and resources involved.

The type of Jena ARQ extension mechanism catering for all above desiderata is known as *property function*. Syntactically property functions use infix notation, appearing in the WHERE clause of SPARQL query as triples: the function is in the predicate position (predicates are often called properties, hence the name "property function").

Both the values in the subject and in the object positions are passed to the function as arguments, whereby one of the two is meant to be the output and hence needs to contain an ubound variable. To pass multipe arguments, RDF lists need to be used. The predicate denoting functions can be distinguished by a dedicated namespace. The first option is to directly instruct ARQ which Java class to instantiate using a pseudo-url "java:⟨java.namespace.⟩". The SPARQL function name needs to coincide with the name of the Java class implementing it in this case, which is not always convenient. The other way is to use a special registry of IRIs that resolve to property functions (such as ppath:topk) in our example. Such a registry is provided by ARQ. The downside of this approach is a slight increase of boilerplate code and, most importantly, the necessity to rebuild the calling Java program, which rules out this option for existing applications with dynamic queries like Jena Fuseki[5]. When the matching between the special predicate name and the Java function is established by ARQ, the function is called for each tuple of constants binding the variables occurring in the triple that represent the property function call. The function `topk` is put into action in the small example in Listing 2.

4 ALGORITHM

The core of our solution[6] is the implementation of the topk function itself, for which we follow a relatively simple approach based on the bidirectional breath first search (BBFS). Our prior work [9] showed that BBFS can perform surprisingly well on semantic graphs in the compact and efficient HDT format.

Herein, we extend the algorithm from [9] with path pruning based on path expressions, provide an implementation which is easy to adapt to arbitrary graph models, and incorporates the path search in full SPARQL via the extension function mentioned in the previous section. The listing of the extended path search algorithm can be found below as algorithm 1.

The bidirectional breadth-first search (BFS) algorithm maintains the sets f_f and f_b of resources (RDF nodes) called *frontiers*: before the i-th iteration of the search procedure, f_f contains references to resources reachable from the source in exactly $\lfloor i/2 \rfloor$ steps, and f_b refers to resources reachable from the target node in exactly $\lfloor (i-1)/2 \rfloor$ steps. At each iteration, either the forward frontier f_f (odd i) or the backward frontier f_b (even i) is advanced. A resource α referenced by both frontiers before the iteration i belongs to a path of length $i-1$. Since BFS is used, all paths of the specified length are identified at the respective iteration of the algorithm. The finding of [9] is the way of maintaining the set of *paths* from the two terminal nodes to the respective frontiers using linked lists, so that if two paths have common prefix, this prefix is only represented once in the memory. Thus, the actual data items stored in frontiers are *traversal edges* (n, e, pr, γ) where n denotes the node, e is the incedent edge via which this node has been reached, pr is the reference to the preceding traversal edge $(n_p, e_p, pr_p, \gamma_p)$ constructed at the previous advance step in the same direction, that is, on the one before previous iteration (the forward and the backward frontiers are advanced interchangeably). The meaning of γ is explained below.

To account for the property path pattern P in the process of search, we convert it into a nondeterministic finite automaton (NFA) using the library dk.brics.automaton[19] by Anders Møller. The implementation is based on character strings. Thus, in a preprocessing step (not shown in algorithm 1), we map each property mentioned in the path expression P to a unique character. Furthermore, a special character $\perp$ is reserved to represent properties not used in P: such properties are not distinguished by P, therefore for the admissibility w.r.t. P, all such properties can be represented by one and the same symbol. The overall size of P is limited by the number of Unicode characters, which is perfectly sufficient in practice.

To cater for path checking also in the backward search, the second automaton based on the *inverse* of the path expression P is used: to invert P in our property path language it suffices to recursively reverse all sequences occurring in P: that is, replace every sequence (P_1/P_2) with (P_2/P_1). Longer sequences $(P_1/\ldots/P_k)$ (which we allow by virtue of associativity of $/$) are inverted as $(P_k/\ldots/P_1)$. Two NFAs A_f and A_b are obtained respectively from the path expression

[3] https://jena.apache.org/

[4] We assume that the default namespace URI ':' with an empty name does not occur as a term in the dataset.

[5] https://jena.apache.org/documentation/serving_data/

[6] Available online at https://bitbucket.org/vadim_savenkov/topk-pfn

Algorithm 1 Bidirectional BFS with Pattern Enforcement via NFA

```
procedure BidirectionalBFS($G$, $start$, $target$, $k$, $P$)
    $sol \leftarrow \emptyset$                                  ▷ Solutions: shortest paths
    $A_f \leftarrow$ Automaton($P$)                           ▷ RegExp $P$ to NFA
    $A_b \leftarrow$ Automaton(inverse($P$))                  ▷ Inverse of $P$ to NFA
    $f_f \leftarrow (start, \text{null}, \text{null}, \gamma_{\text{init}}^{A_f})$          ▷ Forward frontier
    $f_b \leftarrow (target, \text{null}, \text{null}, \gamma_{\text{init}}^{A_b})$         ▷ Backward frontier
    $(f_{act}, A_{act}) \leftarrow (f_f, A_f)$                ▷ $f_f$ is the active frontier
    $(f_{pass}, A_{pass}) \leftarrow (f_b, A_b)$
    while $|sol| \leq k$ and not both $f_f$, $f_b$ stable do
        if Advance($f_{act}, A_{act}$) then
            $sol \leftarrow sol \cup$ filter($P$, join($f_f, f_b$))
        else
            Mark $f_{act}$ as stable.
        end if
        swap $(f_{act}, A_{act}) \leftrightarrow (f_{pass}, A_{pass})$
    end while
    return $sol$
end procedure

procedure Advance($f$, $A$)
    $f' \leftarrow \emptyset$
    $\text{inc} = \begin{cases} \text{successor} & \text{if } f \text{ is the forward frontier} \\ \text{predecessor} & \text{otherwise} \end{cases}$
    for $(n, e, pr, \gamma) \in f$ do
        for $(e', n') \in \text{inc}(n)$ do
            $\gamma' \leftarrow$ nextstate($A, \gamma, e'$)
            if $\gamma' \neq$ reject then
                $f' \leftarrow f' \cup \{(n', e', (n, e, pr, \gamma), \gamma')\}$
            end if
        end for
    end for
    if $f \neq \emptyset$ then
        Update the active frontier: $f \leftarrow f'$
    else
        return fail
    end if
end procedure

function join($f_f$, $f_b$)
    $res \leftarrow \emptyset$
    for $(n, e_1, pr_1, \gamma_1) \in f_f$, $(n, e_2, pr_2, \gamma_2) \in f_b$ do
        $res \leftarrow res \cup$ trace($n, e_1, pr_1$)$\cdot$trace($n, e_2, pr_2$)
    end for
    return $res$
end function
```

P and its inverse P'. At the frontier advancement step, for a node n in a frontier tuple (n, e, pr, γ), we only follow those incident edges e of n which are not rejected by the respective automaton A in its step γ (e is an outgoing edge for the forward frontier and incoming for the backward one).

	Triples	Subjects	Pred's	Objects	Shared
0.1DB	9 264 609	313 036	13 114	3 482 820	58 535
1DB	46 275 619	1 457 983	21 875	13 751 780	462 478

Table 1: Evaluation dataset statistics

ID	Source node (dbr:)	Target node (dbr:)	Property (dbp:)
Q1	Felipe_Massa	Red_Bull	firstWin
Q2	1952_Winter_Olympics	Elliot_Richardson	after
Q3	Karl_W._Hofmann	Elliot_Richardson	predecessor
Q4	Karl_K._Polk	Felix_Grundy	president

Table 2: Input parameters

5 EVALUATION

We experimented with two datasets from the DBpedia SPARQL Benchmark [20] used in "Top-k Shortest Paths in large typed RDF Datasets Challenge" [7] which was part of the 13th European Semantic Web Conference in 2016. Our hardware setup was a desktop machine with 4x3,2 GHz Intel i5 processor and 16GB RAM. The used datasets correspond to the 10% sample and to the full benchmark dataset, which we respectively denote 0.1DB and 1DB. The data is freed from blank and untyped nodes. Table 1 lists the total number of triples, distinct subjects (IRIs), predicates, and objects, as well as the number of IRIs that occur both in the subject and the object position in the dataset. It is clear from the dataset statistics that the number of such shared objects is relatively small, within one promille of the number of triples.

Our experiment is based on the queries of the ESWC '16 challenge, which were four distinct pairs of source and target IRIs and two parameters, namely k and an additional path condition present in half of the cases, stipulating that a certain label must be present either as the first or the last edge of the path, captured by the path pattern (⟨property⟩/(!:)* | (!:)*/⟨property⟩). All combinations of input parameters used in the challenge are given in Table 2.

Listing 3 presents the query that computes required paths for one of the tasks using our property function ppf:topk.

```
PREFIX  : <http://dbpedia.org/property/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX ppf: <java:at.ac.wu.arqext.path.>
SELECT ?path WHERE {?path ppf:topk (dbr:Felipe_Massa dbr:Red_Bull 10
                                    ":firstWin/((!:)*)|(!:)*/:firstWin")}
```

Listing 3: Implementation of the query Q1 with 10 paths

We summarize the results of this experiment graphically in the following three figures. The first figure shows the dependency between the running time and the values of k used in different tasks. Note that not all tasks have the same values of k therefore the chart is not monotone, however one can see the trend of the running time increase with the growth of k. The two lines in Figure 1 show the average performance of the queries based on Table 2, for the tasks with and without path expressions. The dependency is not strictly monotone since not all queries are evaluated with the same values of k in the challenge. However, one can see that (i) the impact of path expressions on the performance in this particular

[7] cf. https://bitbucket.org/ipapadakis/eswc2016-challenge/downloads/

case is negligible (2) the dependency on k is linear (both axes in the figure are logarithmic).

Figure 2 shows the performance of exactly the same set of queries run against 0.1DB and the full DBpedia Benchmark dataset 1DB. It is noteworthy that an intuition that query performance should degrade monotonically as size of the input increases does not hold for top k paths queries. The reason is that the k-th shortest path computed on sampled data tends to be *longer* than if computed on an extended dataset, degree of nodes in which may grow. For instance, the longest path for the sample is of length 10, and of length 9 for the full case. For 10 out of 38 queries of the challenge, the maximal path length on 0.1DB was exceeding the maximal path length on 1DB by 20 to 25%. As a consequence, the search for k may take longer and thus the performance is sometimes worse on 0.1DB as Figure 2 shows.

In all cases, and on the HDT backend, our algorithm was able to compute every single path in the maximum of 9 and 10 seconds respectively for both databases 0.1DB and 1DB. The median running time is 46 ms, the average for the 0.1DB is 652 ms and for 1DB 929 ms. Each task with k smaller than 100 took at most 112 ms to solve.

We compared the performance of k shortest paths queries over HDT with the SPARQL 1.1 property paths queries using the same source and target nodes, and the same path pattern. This allowed us to benchmark against popular SPARQL engines Blazegraph[8] (open source), Virtuoso Community Edition[9] (open source) and Stardog[10] (closed source, commercial). It has to be stressed that this is by no means a real benchmark, since our setup on top of an HDT store is a read-only system highly optimized for querying, whereas the other three systems are full SPARQL engines with the read and write functionality. Furthermore, we made no performance optimizations beyond ensuring that the systems can use all available memory. The results of the comparison on 0.1DB are summarized in Table 3.

Query	$k = 1$	$k = 100$	Stardog (reachability)	Blazegraph, Virtuoso
Q1 (!:*)	24	94	57	N/A
Q1	125	6 138	19,732	N/A
Q2 (!:*)	15	132	45	N/A
Q2	19	2 568	34 707	N/A

Table 3: Performance of top-k / reachability queries (in ms)

Neither Blazegraph nor Virtuoso were able to cope with the queries on 0.1DB within our 16Gb main memory limit. Stardog was performing very well for unrestricted paths, however a path expression fixing any of the first or the last path edge resulted in a significant performance degradation. This suggests that an efficient implementation of regular expressions may be an important performance factor. One can see that for the value $k = 100$ our top k paths function stands well in terms of performance against the reachability test implementation of Stardog. Moreover, as Figure1 suggests, no clear performance penalty is associated with the necessity to enforce path patterns, due to our use of the BSD-licensed

[8]https://www.blazegraph.com/
[9]https://virtuoso.openlinksw.com/
[10]https://www.stardog.com/

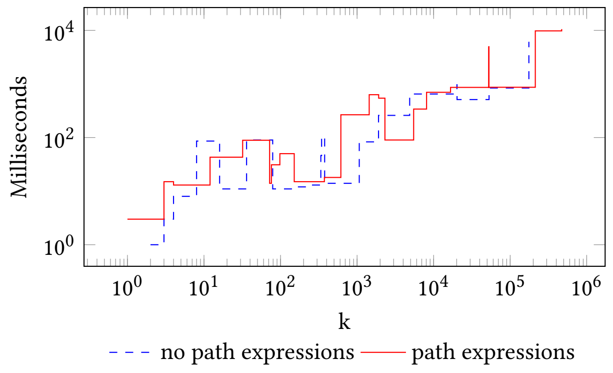

Figure 1: Performance with and without path expressions

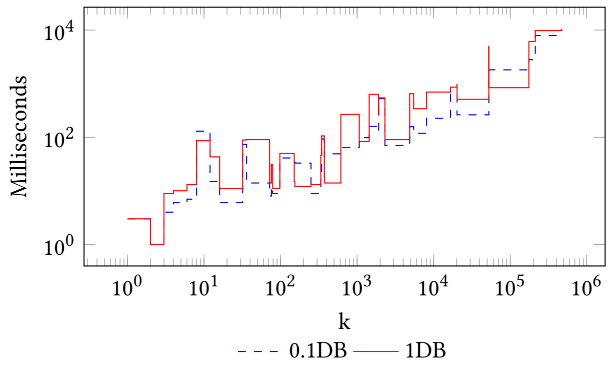

Figure 2: Performance for 1DB and its 10% sample 0.1DB

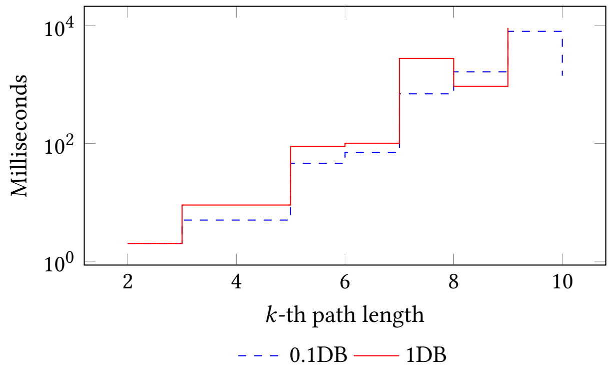

Figure 3: Performance vs. the path length

dk.brics.automaton library [19]. The comparison also shows that property paths support in popular open source systems can be easily improved by using bidirectional search.

In the last experiment we searched for top k paths between nodes occurring at the ends of a random walk of a given length, which we varied in the range 5..20, in order to obtain the upper bound on the shortest path length. To also find long enough paths we were increasing the value of k. For 0.1DB, this approach typically allowed us to find paths of maximal length up to 14, in

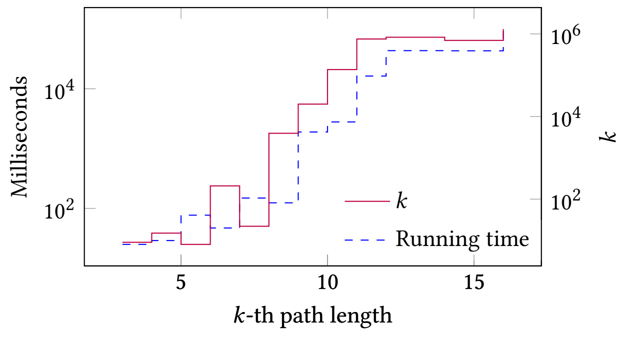

Figure 4: Performance vs. random path length

which case k was often close or even exceeding the value 10^6, since most of DBpedia paths belong to cycles, and have alternative paths in the graph (in particular, most properties with the http://dbpeida.ort/propery namespace are also present in the namespace http://dbpeida.ort/ontology which alone causes a combinatorial explosion of the number of paths).

Figure 4 shows that both k and the query evaluation time are exponential in the length of the top k-th path length.

6 RELATED WORK

Although path search is one of the most studied problems in Computer Science and also central in the Semantic Web area, the top k shortest path problem has not received sufficient attention in the Semantic Web context so far. In fact, to the best of our knowledge the three ESWC'16 Challenge submissions [13], [14], [6] were the first to deal with the top k path computation specifically. The winner approach by Herlting et al. [14] applied the Eppstein routing algorithm [7], which was designed for weighted graphs. In [6] the authors focus on decentralized computation and [13] developed an extension of the algebraic algorithm based on matrix multiplication and a special ρ-index structure. The primary benefit of our solution compared with the mentioned works is performance: due to the use of HDT-based indexing, for all reported queries our algorithm is at least the order of magnitude faster than any of the aforementioned solutions. The second benefit of our system is the possibility of using it with the standard systems out of the box. Having tested it with the HDT backend, we based our implementation on the standard Jena ARQ extension specification which makes it applicable to every system supporting this API.

Beyond the scope of top k path queries, the literature on property path computation in the SPARQL context is broad. One of the earliest published accounts on addressing path queries was SPARQLeR [17] and a comprehensive path query processor SPARQ2L [3]. Both approaches extend SPARQL with *path variables* (prepended by % or ?? instead of the usual single ?) instantiated by paths in queries such as `SELECT %path WHERE {⟨r⟩ %path ⟨s⟩}`. Additional possibilities of extracting individual resources from the path, filtering paths using path expressions and length restrictions (in the `FILTER` operator) and comparing paths, e.g. testing them for equality is supported. Neither of the two systems support top k path queries: to limit the number of retrieved results, one needs to restrict the path length in the filter condition. Ten years past the publication date, neither SPARQ2L nor SPARQLeR system seem to be in use, freely available online for download or can be combined with the main open source SPARQL engines such as Jena ARQ or Virtuoso.

A path extension has been also reported for the efficient RDF3X engine [11]. With the syntax close to the previous two systems, RDF3X_path only finds a single shortest path, not an arbitrary k ones. The extension is tightly incorporated into the RDF3X codebase and cannot be used independently. One of the most comprehensive syntactical extensions of SPARQL has been undertaken by the CPSPARQL engine [1], where regular path expressions are extended with constraints on resources (nodes) occurring within paths. Again, top k path queries are not supported in this system, and although an implementation of the system is available, the focus of research is on the flexible and expressive language itself rather than on efficacy of query evaluation. No performance results on large graphs have been reported for CPSPARQL to the best of our knowledge.[11]

The creators of Stardog have recently announced a plan to support path retrieval via a SPARQL extension Pathfinder with an expressive syntax supporting complex path expressions, somewhat similar to CPSPARQL.[12] At the moment of publication of the present paper the Pathfinder extension has not yet been released. It should also be noted that Stardog is a closed source system, and that it is hard to predict whether the new syntax will be adopted by other implementations and will eventually become part of the standard.

Blazegraph also provides two extensions connected through the service interface, one called ALP and another GAS. ALP stands for Arbitrary Length Paths, the respective service is meant to support more complex path expressions (e.g. checking a filter expression for each property or node in the path) via a rewriting technique. ALP mandates that the lower and upper bounds on the path length are set. Performance-wise the service does not introduce significant traversal optimizations to the implementation of Blazegraph. GAS service is named after the Gather-Apply-Scatter framework [10], which generalizes the "think like a vertex" approach of graph processing pioneered by Google's Pregel system [18] for big distributed graphs. Several GAS algorithms are implemented, including the bidirectional traversal and single-source shortest path algorithm. Path expressions are not supported by the currently available GAS algorithms. Furthermore, the GAS approach is quite different from SPARQL, results typically require post-processing, and for a basic task of path enumeration on a locally stored graph this solution might be too complex.

An increasingly popular approach to evaluating graph queries is based on the graph traversal language Gremlin and the graph processing framework Tinkerpop, both belonging to the Apache technology stack.[13] Gremlin is a concise graph traversal language which is however very different from SPARQL. Both Stardog and Blazegraph expose their RDF graphs to Gremlin via Gremlin's own query console. Comparison with such essentially non-SPARQL tools is out of the scope of the present paper, where our focus is on

[11] A predecessor of CPSPARQL, the PSPARQL system by the same authors [2], apparently sharing large part of the code base with it, has been applied in the "Yottabyte" context by [4] to advocate the existential semantics of path expressions. However, input data sizes in [4] remain in the range of several kilobytes.
[12] http://www.stardog.com/blog/a-path-of-our-own/
[13] http://tinkerpop.apache.org/

a minimalistic and simple SPARQL extension, that should fit into existing open source engines without a need for changing their code base.

7 CONCLUSIONS

We presented an efficient solution for the *k* shortest path problem in the context of SPARQL. Our function is based on the adaptation of the bidirectional breadth first search to top *k* paths computation [9], extending it with path expressions and embedding into Jena ARQ via the extension mechanism of property functions. On the indexed HDT backend, our implementation demonstrates very promising performance even without complex optimizations. Employing reachability indexes to deal with very large and dense graphs is left for future work, as well as adaptation to further SPARQL engines, in particular Sesame and Blazegraph. The source code of our implementation is openly available.

ACKNOWLEDGMENTS

This work has been supported by the Austrian Ministry for Transport, Innovation and Technology (bmvit) and the Austrian Research Promotion Agency (FFG) under the project GraphSense (grant no. 849906) and Open Data for Local Communities (grant no. 855407). Qaiser Mehmood was supported by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289.

REFERENCES

[1] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. 2008. Constrained regular expressions in SPARQL. In *International conference on semantic web and web services-SWWS 2008*. CSREA Press, 91–99.

[2] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. 2009. Extending SPARQL with regular expression patterns (for querying RDF). *Web Semantics: Science, Services and Agents on the World Wide Web* 7, 2 (2009), 57–73.

[3] Kemafor Anyanwu, Angela Maduko, and Amit P. Sheth. 2007. SPARQ2L: towards support for subgraph extraction queries in rdf databases. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. 797–806. DOI: http://dx.doi.org/10.1145/1242572.1242680

[4] 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 *Proceedings of the 21st World Wide Web Conference 2012 (WWW 2012)*, Alain Mille, Fabien L. Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab (Eds.). ACM, Lyon, France, 629–638.

[5] Charles E Cook, Mary Todd Bergman, Robert D Finn, Guy Cochrane, Ewan Birney, and Rolf Apweiler. 2016. The European Bioinformatics Institute in 2016: data growth and integration. *Nucleic acids research* 44, D1 (2016), D20–D26.

[6] Laurens De Vocht, Ruben Verborgh, and Erik Mannens. 2016. *Using Triple Pattern Fragments to Enable Streaming of Top-k Shortest Paths via the Web*. Springer, Cham, 228–240. DOI: http://dx.doi.org/10.1007/978-3-319-46565-4_18

[7] David Eppstein. 1999. Finding the K Shortest Paths. *SIAM J. Comput.* 28, 2 (Feb. 1999), 652–673. DOI: http://dx.doi.org/10.1137/S0097539795290477

[8] Javier D Fernández, Miguel A Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. 2013. Binary RDF representation for publication and exchange (HDT). *Web Semantics: Science, Services and Agents on the World Wide Web* 19 (2013), 22–41.

[9] Erwin Filtz, Vadim Savenkov, and Jürgen Umbrich. 2016. On finding the k shortest paths in RDF data. In *Intelligent Exploration of Semantic Data (IESD 2016) – A Workshop at the International Semantic Web Conference (ISWC 2016)*.

[10] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[11] Andrey Gubichev and Thomas Neumann. 2011. Path Query Processing on Very Large RDF Graphs. In *Proceedings of the 14th International Workshop on the Web and Databases 2011, WebDB 2011, Athens, Greece, June 12, 2011*. ACM. http://webdb2011.rutgers.edu/papers/Paper21/pathwebdb.pdf

[12] Zohaib Hassan, Mohammad Abdul Qadir, Muhammad Arshad Islam, Umer Shahzad, and Nadeem Akhter. 2016. Modified MinG Algorithm to Find Top-K Shortest Paths from large RDF Graphs. In *Semantic Web Challenges - Third SemWebEval Challenge at ESWC 2016, Revised Selected Papers (Communications in Computer and Information Science)*, Harald Sack, Stefan Dietze, Anna Tordai, and Christoph Lange (Eds.), Vol. 641. Springer, Heraklion, Crete, Greece, 213–227.

[13] Zohaib Hassan, Mohammad Abdul Qadir, Muhammad Arshad Islam, Umer Shahzad, and Nadeem Akhter. 2016. *Modified MinG Algorithm to Find Top-K Shortest Paths from large RDF Graphs*. Springer, Cham, 213–227. DOI: http://dx.doi.org/10.1007/978-3-319-46565-4_17

[14] Sven Hertling, Markus Schröder, Christian Jilek, and Andreas Dengel. 2016. Top-k Shortest Paths in Directed Labeled Multigraphs. In *Semantic Web Challenges - Third SemWebEval Challenge at ESWC 2016, Revised Selected Papers (Communications in Computer and Information Science)*, Harald Sack, Stefan Dietze, Anna Tordai, and Christoph Lange (Eds.), Vol. 641. Springer, Heraklion, Crete, Greece, 200–212.

[15] Wei Hu, Honglei Qiu, and Michel Dumontier. 2015. Link Analysis of Life Science Linked Data. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 9367. Springer, 446–462. DOI: http://dx.doi.org/10.1007/978-3-319-25010-6_29

[16] Bevan E Huang, Widya Mulyasasmita, and Gunaretnam Rajagopal. 2016. The path from big data to precision medicine. *Expert Review of Precision Medicine and Drug Development* 1, 2 (2016), 129–143. DOI: http://dx.doi.org/10.1080/23808993.2016.1157686 arXiv:http://dx.doi.org/10.1080/23808993.2016.1157686

[17] Krys Kochut and Maciej Janik. 2007. SPARQLeR: Extended SPARQL for semantic association discovery. *The Semantic Web: Research and Applications* (2007), 145–159.

[18] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

[19] Anders Møller. 2010. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. (2010). `http://www.brics.dk/automaton/`.

[20] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. 2011. DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*. 454–469. DOI: http://dx.doi.org/10.1007/978-3-642-25073-6_29

[21] Ioannis Papadakis, Michalis Stefanidakis, Phivos Mylonas, Brigitte Endres-Niggemeyer, and Spyridon Kazanas. 2016. Top-K Shortest Paths in Large Typed RDF Datasets Challenge. In *Semantic Web Challenges - Third SemWebEval Challenge at ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Revised Selected Papers*. Springer, 191–199. DOI: http://dx.doi.org/10.1007/978-3-319-46565-4_15

[22] Jeffrey D. Ullman. 1989. *Principles of Database and Knowledge Base Systems*. Computer Science Press.

[23] Laurens De Vocht, Ruben Verborgh, and Erik Mannens. 2016. Using Triple Pattern Fragments to Enable Streaming of Top-k Shortest Paths via the Web. In *Semantic Web Challenges - Third SemWebEval Challenge at ESWC 2016, Revised Selected Papers (Communications in Computer and Information Science)*, Harald Sack, Stefan Dietze, Anna Tordai, and Christoph Lange (Eds.), Vol. 641. Springer, Heraklion, Crete, Greece, 228–240.