Top-k Shortest Paths in Directed Labeled Multigraphs

Sven Hertling[1], Markus Schröder[1], Christian Jilek[1], and Andreas Dengel[1,2]

[1] Knowledge Management Group, German Research Center for Artificial Intelligence (DFKI) GmbH, Trippstadter Straße 122, Kaiserslautern, Germany
[2] Knowledge-Based Systems Group, Department of Computer Science, University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern, Germany
{sven.hertling, markus.schroeder, christian.jilek, andreas.dengel}@dfki.de

Abstract. A top-k shortest path algorithm finds the k shortest paths of a given graph ordered by length. Interpreting graphs as RDF may lead to additional constraints, such as special loop restrictions or path patterns. Thus, traditional algorithms such as the ones by Dijkstra, Yen or Eppstein cannot be applied without further ado. We therefore implemented a solution method based on Eppstein's algorithm which is thoroughly discussed in this paper. Using this method we were able to solve all tasks of the ESWC 2016 Top-k Shortest Path Challenge while achieving only moderate overhead compared to the original version. However, we also identified some potential for improvements. Additionally, a concept for embedding our algorithm into a SPARQL endpoint is provided.

Keywords: top-k shortest paths, loop restrictions, Eppstein's algorithm

1 Introduction

A common approach for representing knowledge is the usage of a knowledge graph. Such graphs are well-known for their ability to describe relationships between concepts. Usually, a shortest path explains an obvious link between them, whereas subsequent shortest paths could reveal even more interesting relationships. A general way to compute such paths is the usage of a top-k shortest path algorithm which outputs shortest paths ordered by weight or length, respectively. In the area of semantic web, in which knowledge graphs are typically represented using RDF, such algorithms were already applied [3, 7]. Interpreting graphs as RDF may lead to additional constraints, such as special loop restrictions or path patterns. In our case we will conceive an RDF graph as a directed labeled multigraph having RDF resources as vertices and RDF properties as edges. We further want to disallow paths including the same RDF triple (vertex - edge - vertex) multiple times. Apart from that, we tolerate multiply visited vertices and edges. That is why common algorithms cannot be applied without further ado. Additionally, even suitable modifications may result in unacceptable time and memory consumption in practice. For example, a slightly modified Dijkstra algorithm is not practical due to the large size of typical RDF graphs. Considering the constraints mentioned above, this paper proposes a solution method based on Eppstein's top-k shortest path algorithm [2].

The paper is structured as follows: In Section 2 other papers mainly about finding shortest paths are presented. Section 3 explains our approach which is evaluated and discussed in Sections 4 and 5. Our algorithm's results on the evaluation sets of the ESWC 2016 Top-k Shortest Path Challenge are separately addressed in Section 6. Last, we shortly introduce our SPARQL endpoint for shortest paths (Section 7) and give an outlook on possible future work (Section 8).

2 Related Work

Approaches of finding top-k shortest paths can be divided in two categories based on whether their resulting paths are loopless or not. The latter is a less complex task.

One well-known representative of this category is Eppstein's algorithm [2]. Its core is a bounded-degree graph which outputs k paths using breadth-first search. Given a graph with n vertices and m edges this results in a time complexity of $O(m + n \log n + k)$. There are various modified versions: The algorithm by Jiménez et al. [5] lazily builds data structures for practical improvements whereas Aljazzar et al. [1] allow the usage of a heuristic function to guide through the search space.

Yen's algorithm [12] is an example of the other category finding loopless paths. It has a time complexity of $O(kn(m + n \log n))$ which results from calling Dijkstra's algorithm multiple times to compute shortest paths.

Examining the complexity of evaluating property paths defined in SPARQL [6, 8] and retrieving paths from large datasets [3, 11] are other related research areas of interest.

More details will be given in the next section which is about our approach.

3 Approach

While Eppstein's top-k shortest path algorithm [2] finds paths with loops, our solution ensures that every path additionally complies to a unique triple condition. In particular, a path and the unique triple condition are defined as follows:

Definition 1 *A path is a sequence of alternating RDF resources and properties* $(r_0, p_1, r_1, \ldots, p_n, r_n)$ *where* $n > 0$.

Definition 2 *The unique triple condition for a path* $(r_0, p_1, r_1, \ldots, p_n, r_n)$ *is true if and only if* $\forall\ 0 < i \leq n : \exists!(r_{i-1}, p_i, r_i) \in p$.

A path is valid if (and only if) the unique triple condition is true.

There are paths which comply to this condition but contain multiple visited vertices. Yen's algorithm [12] would consider these paths to contain loops. Thus, we decided against softening the loop restriction on the one hand and inserting the unique triple condition on the other.

Our contribution is based on Eppstein's algorithm [2] which is shortly explained in the following. (For a more detailed introduction we kindly refer the reader to the original paper.) The algorithm uses a bounded-degree graph $P(G)$ built in $O(m + n \log n)$ that outputs k paths in linear time using breadth-first search. For building $P(G)$ a single source shortest path tree T for target vertex t has to be computed. All edges which are not part of T are called sidetrack edges, thus $G - T$ is a sidetrack graph. $P(G)$ is built with the help of two kinds of heaps: (1) $H_T(v)$ forms a heap of all minimal sidetracks on the shortest path from v to t. (2) $H_{out}(v)$ forms a heap of all outgoing sidetracks of v except the minimal sidetrack. Merging $H_T(v)$ and $H_{out}(v)$ results in $H_G(v)$. All H_G are connected by so-called cross edges to form $P(G)$. A breadth-first search on $P(G)$ with a priority queue outputs the top-k shortest paths ordered by weight in $O(k)$.

Our approach is depicted as pseudocode in Algorithm 1. In general, the given RDF dataset is interpreted as an edge-label directed multigraph (line 1). Before running the algorithm's main part, all statements containing a literal are removed in a preprocessing step[3]. Lines 2 - 13 are directly adopted from Eppstein: Like stated above, the shortest path tree T, the sidetrack graph $G-T$, the edge weights $\delta(e)$ and the heaps H_T and H_{out} are computed and necessary data structures are initialized.

Three modifications to the original Eppstein algorithm are made: (a) Every path has to be built and/or checked in order to immediately check its validity (line 16), (b) only valid paths are added to the result list R (lines 17-19) and (c) $P(G)$ is pruned whenever p is invalid due to multiple sidetracks (line 20).

There are cases in which a path does not comply to the unique triple condition: It may contain multiply used (1) sidetracks and/or (2) shortest path edges.

To item (1): Using $P(G)$ Eppstein's algorithm keeps track of activated sidetracks, thus multiply used ones are easily detectable. In this case we can stop adding a cross edge, because all further paths would become invalid. Additionally, if k is greater than the number of possible valid paths in G, this pruning guarantees our algorithm's termination. However, the computation does not stop until every sidetrack combination is generated and checked which may result in significant overhead.

To item (2): Since $P(G)$ only contains sidetracks, each path has to be built in order to detect multiply used shortest path edges. In that case we have to extend $P(G)$ and add a cross edge, since there could be a sidetrack in $P(G)$ which allows bypassing this problematic shortest path edge in the future. It's

[3] This is done since providing a literal as a target node could be ambiguous (e.g. several persons having the same first name). Thus, to apply our algorithm in such use cases, this ambiguity has to be resolved first. For example, one could first search all resources associated with the given literal and then choose the one that is actually meant as the target node (or subsequently run the algorithm on all of them).

not possible to remove these nodes in $H_G(v)$, since the algorithm would then miss cross edges to potential valid paths.

A path is built with a (possibly empty) queue of sidetracks S and the start node s (see (a) above and line 16). If S already contains duplicates (case 1, see above) building the path is obsolete. If no duplicates are detected, the system initiates the building process in which sidetracks (if suitable) are preferred over shortest path edges. While building the path, duplicate shortest path edges can be detected by keeping track of used edges. There are special cases in which the given end node t is visited multiple times. Thus, the building path algorithm stops at t only if all given sidetracks in the queue are used.

In the given algorithm, $P(G)$ is only extended and $H_G(v)$ only constructed if necessary (lines 21 - 23). This is possible due to the pruning condition (line 20). Next, a breadth-first traversal step is performed on $P(G)$ and weights are updated (lines 25 - 29). Finally, all found and valid paths in R are returned (line 32).

For more complex problems we would also like to consider a special property path condition which we state as follows:

Definition 3 *The special property path condition for a path* $(r_0, p_1, r_1, \ldots, p_n, r_n)$ *and a given property* P *is true if and only if* $p_1 \equiv P \vee p_n \equiv P$.

A naive approach would be to simply delete all $(s, \neg P, r)$ edges from G. However, this would prevent valid paths like $(s, P, \ldots, s, \neg P, \ldots)$ from being found. That's why our approach (see Algorithm 2) introduces a dummy vertex v which ensures that all considered paths contain P as the first property. First, the algorithm collects all P-edges that start in s (we denote this set as A). If none is found the algorithm terminates, leaving only P-edges which end in t to be checked. By inserting the dummy vertex v and connecting it to all *head*(e) of A we ensure that all paths from v to t contain P as the first property. After each found path the algorithm has to substitute v with s which is valid, since there actually exist outgoing P-edges from s.

Finding paths containing P-edges that end in t can be reduced to the previously solved problem. We therefore invert all edges in G and call Algorithm 2 with interchanged s and t. The resulting paths have to be reversed. Finally, both result lists are merged and ordered by length also removing duplicates.

In order to apply this approach, the previously introduced build path algorithm had to be modified. Since every path starts with $(v, P, r, \ldots)$ the occurrence of (s, P, r) would not be recognized as a duplicate. This is solved by artificially adding (s, P, r) to the activated sidetrack queue S and to the set of already used edges.

How our method performs in practical scenarios is discussed in the next section.

```
Data: start node $s$, end node $t$, required number of paths $k$, dataset $D$
Result: $k$ paths between $s$ and $t$ in $D$ ordered by length complying to unique
        triple condition
interpret $D$ as an edge-labeled directed multigraph $G$;
compute shortest path tree $T$ starting from $t$ using Dijkstra;
compute sidetrack graph $\widetilde{G} = G - T$;
for $e \in \widetilde{G}$ do
    compute $\delta(e) = l(e) + d(head(e), t) - d(tail(e), t)$;
end
for $v \in T$ do
    compute $H_T(v)$ and $H_{out}(v)$;
end
initialize initial edge $init = (\emptyset, \emptyset, s)$;
initialize heap $H =< init >$;
initialize graph $P(G) = (\{init\}, \emptyset)$;
initialize result list $R =<>$;
while $H$ is not empty $\wedge |R| < k$ do
    remove minimal element $m$ from H;
    build path $p$ using $m$;
    if $p$ is valid then
        add $p$ to $R$;
    end
    if $p$ is valid $\vee$ $p$ contains multiple shortest path edges then
        compute $H_G(head(m))$ and add to $P(G)$;
        let $r = root(H_G(head(m)))$;
        add cross edge $(m, r)$ with weight $\delta(r)$ to $P(G)$;
    end
    for $e \in out(m)$ do
        let $n = head(e)$;
        compute $\delta(n) = \delta(m) + l(e)$;
        add $n$ to $H$;
    end
    remove $m$ from $P(G)$;
end
return $R$;
```

Algorithm 1: Modified Eppstein Algorithm

```
    Data: start node $s$, end node $t$, required number of paths $k$, property $P$,
          dataset $D$
    Result: $k$ paths between $s$ and $t$ in $D$ ordered by length complying to
            unique triple and special property path condition with $P$
 1  interpret $D$ as a edge-labeled multigraph $G$;
 2  let $A = \{e \in G : \mathit{tail}(e) \equiv s \wedge \mathit{label}(e) \equiv P\}$;
 3  if $A$ is empty then
 4      return $\emptyset$;
 5  end
 6  construct an arbitrary vertex $v \notin G$;
 7  add $v$ to $G$;
 8  for $e \in A$ do
 9      add edge $(v, \mathit{head}(e))$ to $G$;
10  end
11  run modified Eppstein with $s' = v$, $t' = t$, $k' = k$;
```

Algorithm 2: Reuse of modified Eppstein to retrieve paths complying to the special property path condition

4 Evaluation

We evaluated our approach on the training set of the ESWC 2016 Top-k Shortest Path Challenge. This challenge consists of two tasks: Besides finding the top-k shortest paths complying to the unique triple condition (see Definition 2), the second one additionally introduces a special property path restriction (see Definition 3).

The training set corresponds to a 10% dataset generated from the DBpedia knowledge base [9,10]. Since the data is not generated artificially like the LUBM dataset [4], it is very heterogeneous and there are no well structured classes. The dataset is processed so that there are no blank nodes, untyped classes or unparsable triples[4]. As a consequence, each resource r is not a blank node and has at least got one statement (r rdf:type *type*) after the processing.

In total the training set consists of 9,996,907 triples, 7,598,913 of them being literal statements. After their removal the resulting set contains 2,397,994 triples. There are 181,702 duplicate statements which do not provide additional information. In fact, there are 13 statements which appear 4 times in the dataset. Our algorithm has to deal with multigraphs since there are 394,085 multiple edges (also including duplicate triples). Moreover, there are 407 reflexive edges which means that also smallest loops have to be handled. The most important number is the average out degree. In this set, the mean value is 6.03 with a standard deviation of 4.51. The vertices' number of outgoing edges ranges from 0 to 106.

[4] In particular, we found unparsable datatypes like <*http://dbpedia.org/datatype/brake horsepower*>. Our first solution removed them, in a later version they were fixed by URL-encoding the blank.

In this section we analyze our approach in terms of performance, space and overhead compared to the original Eppstein algorithm. Table 1 shows our results of Task 1 in which four queries with increasing values of k were given. For each value we recorded six different measures which are explained in more detail in the following:

The first one is the number of iterations needed by our algorithm, i.e. the number of executions of the *while* loop in line 14. Additional iterations in comparison to Eppstein's algorithm are characterized by the relative overhead to k: $\frac{(iterations-k)}{k}$. Next, we captured time in milliseconds from the computation of the shortest path tree T (line 2) till the return of R (line 32). All experiments were conducted on a typical end user notebook with an Intel 4 Core i7-4712MQ 2.30 GHz CPU, 16 GB RAM and Java 1.8 installed. The next column contains the number of times $P(G)$ was not extended or in other words pruned. This is the case when multiple sidetracks are detected, thus the *if* condition in line 20 is skipped. To give an impression of our algorithm's memory consumption we additionally record $|V_{P(G)}|$ and $|H|$. $|V_{P(G)}|$ is the number of vertices in $P(G)$ (line 12) whereas $|H|$ is the number of entries in heap H (line 11) after termination.

Like stated before, Task 2 introduced the special property path condition (see Definition 3). This means all paths need to have P as their first (i.e. $(s, P, \ldots)$) or last (i.e. $(\ldots, P, t)$) property. The latter case does not appear in queries 1, 3 and 4. For query 2 we separated the cases accordingly. All measures are found in Table 2.

The relation between k and the overhead of all queries is depicted in Figure 1. k is measured on a logarithmic scale whereas the overhead is given in percent on a linear scale. Additionally, we plotted a trendline obtained by applying linear regression on all query results.

Results are further discussed in the next section.

5 Discussion

Since the expected results of the challenge were provided beforehand we could verify that our algorithm performed correctly for all queries. The individual results of our evaluation (see Tables 1, 2 and Figure 1) will be discussed in the following, starting with Table 1.

Obviously, a high k results in a high overhead since more potentially invalid paths are found. This effect is mitigated by pruning of $P(G)$. The time column shows that 5 to 6 seconds are required to initialize the necessary data structures of the training set (see Algorithm 1, lines 2 - 13). Additional time is needed to discover the shortest paths which is done in the while loop. For a maximum k query 4 needs about 40 seconds for this task having a rather low overhead of 10%. Please note that all time measurements were done using a non-optimized version of our algorithm. During implementation we identified some potential for improvements which was not exploited for time reasons. Moreover, we discovered that the number of vertices in $P(G)$ is proportional to k.

Query	k	Iterations	Overhead	Time (ms)	Pruned	$\lvert V_{P(G)}\rvert$	$\lvert H\rvert$
1	8	8	0.00%	6531	0	83	15
1	344	362	5.23%	6663	6	4401	715
1	1068	1128	5.61%	6833	12	16408	2296
1	20152	21663	7.49%	12404	407	293527	43701
2	3	3	0.00%	5528	0	64	6
2	4	4	0.00%	5662	0	80	8
2	79	91	15.18%	5783	4	1702	174
2	154	169	9.74%	5908	4	3193	336
3	36	36	0.00%	5785	0	64	6
3	336	336	0.00%	5932	0	80	8
3	4866	5034	3.45%	8147	4	1702	174
4	2	2	0.00%	6137	0	78	4
4	16	16	0.00%	6230	0	524	34
4	250	254	1.60%	6416	0	7389	515
4	1906	1980	3.88%	8393	34	58426	3984
4	20224	21858	8.07%	13980	678	619359	42879
4	175560	192367	9.57%	45785	7592	5628758	380839

Table 1. Results of Task 1

Query	k	Iterations	Overhead	Time (ms)	Pruned	$\lvert V_{P(G)}\rvert$	$\lvert H\rvert$
1	32	32	0.00%	6912	0	391	63
1	98	98	0.00%	8353	0	1410	200
1	1914	1990	3.97%	9119	28	27063	4030
1	16632	16999	2.20%	12929	119	247948	34909
1	212988	220639	3.59%	45968	2319	3178408	451439
2 $(s, P, \ldots)$	3	3	0.00%	6418	0	61	6
2 $(\ldots, P, t)$	3	3	0.00%	5297	0	99	5
2 $(s, P, \ldots)$	4	4	0.00%	5519	0	77	8
2 $(\ldots, P, t)$	4	4	0.00%	4560	0	127	8
2 $(s, P, \ldots)$	76	88	15.78%	5701	4	1635	167
2 $(\ldots, P, t)$	76	84	10.52%	4589	3	2818	170
2 $(s, P, \ldots)$	151	167	10.59%	5686	5	3149	328
2 $(\ldots, P, t)$	151	169	11.92%	4616	5	5572	338
2 $(s, P, \ldots)$	2311	2848	23.23%	7787	232	52309	5297
2 $(\ldots, P, t)$	2311	2519	9.00%	7988	83	89060	5362
3	12	12	0.00%	5652	0	295	24
3	76	76	0.00%	5554	0	1677	151
3	1440	1488	3.33%	7259	16	32991	2869
3	8088	8377	3.57%	10122	109	173570	16281
4	1	1	0.00%	7827	0	34	1
4	6	6	0.00%	8493	0	218	12
4	72	74	2.77%	7664	0	2166	146
4	614	641	4.39%	8025	12	19182	1280
4	5483	6018	9.75%	10960	198	172989	11788
4	52649	58671	11.43%	20077	2640	1707021	115534
4	471199	540815	14.77%	71884	26962	15997816	1065488

Table 2. Results of Task 2

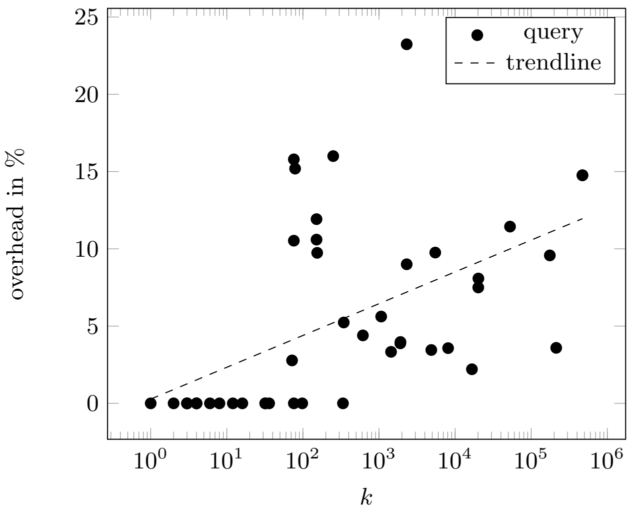

Fig. 1. Relation of k and corresponding overhead

Most findings also apply to Table 2. Although query 2 has high prune values the overhead is the highest in comparison to all other queries. Our assumption is that the execution of query 2 involves a high number of loops from s to t. That is why it has got the highest overhead of 23% for a maximum k of 2311.

Figure 1 gives a summarizing overview of the relation between k and the overhead. Considering that k is plotted on a logarithmic scale we see that the overhead only grows slowly. In particular, there are several queries having a k less than 10^2 which do not cause any overhead. Most queries have an overhead between 0% to 15% even if k approaches 500,000. However, for values greater than 10^2 there are some outliers above 10% though not reaching 25%.

Thus, our algorithm achieves a time complexity which is moderately above the one of Eppstein's algorithm for the given training set. In general, our algorithm's overhead increases with the number of loops in the graph.

6 ESWC 2016 Challenge Evaluation

We also ran our algorithm on the evaluation set of the ESWC 2016 Top-k Shortest Path Challenge. It contains 1,551,041 URIs and about 13.6 million distinct triples, which is about 5.7 times the size of the training set used before. This led to memory problems with our first implementation. We therefore optimized our algorithm's memory efficiency by indexing all triples and storing them as an integer array during runtime. However, we still had to use a more powerful machine compared to the one used in Section 4 in order to complete all four evaluation tasks, in particular the fourth one. We used a virtual machine having

48 cores à 2.8 GHz[5] and 492 GiB of memory running openSUSE Linux 42.1 (64-bit) and Java OpenJDK 8. The achieved results, which are principally quite similar to those of the training set (see Section 4), are given in Table 3.

Query	k	Iterations	Overhead	Time (ms)	Pruned	$\lvert V_{P(G)} \rvert$	$\lvert H \rvert$
1.1	377	389	3.18%	31918	4	21115	739
1.2	53008	56231	6.08%	35154	1589	2979808	106252
2.1 $(s, P, \ldots)$	374	386	3.20%	21283	4	20934	732
2.1 $(\ldots, P, t)$	374	386	3.20%	23349	2	48356	725
2.2 $(s, P, \ldots)$	52664	55877	6.10%	41206	1586	2961258	105579
2.2 $(\ldots, P, t)$	52664	52766	0.19%	271662	23	95446490	151819

Table 3. Results of evaluation

Again, the overhead compared to the original Eppstein algorithm was moderate. Tasks 1.1 to 2.1 needed less than 10 GiB of RAM, whereas up to 80 GiB were required in order to complete Task 2.2. This results from Task 2.2 being quite complex: in the second part of the query $((\ldots, P, t))$, the algorithm was only able to prune for 23 times. Thus, P(G) had to be extended several times resulting in very high memory usage. A low overhead of 0.2% indicates that this is not caused by our extensions but would also have been present when using the original Eppstein alone. Concerning runtime performance there was a noticeable overhead induced by the virtualization, especially also related to memory allocation. Please note that we therefore omitted freeing memory during the calculation for improved runtime performance.

To better use the algorithm in our daily work (e.g. our research about explanation aware computing), we embedded it into a SPARQL endpoint which is explained in the next section.

7 SPARQL Endpoint for Shortest Paths

In this section a modification to the SPARQL evaluation is provided to use the proposed algorithm for calculating k shortest paths. We extended the property path semantics to also include the additional shortest paths while keeping the SPARQL query language syntax and expressiveness. Therefore a `SELECT` query is used to retrieve the path components. The idea is to create a suitable number of hidden variables containing resources and properties alternately. They are numbered consecutively starting from zero. The path length is assigned to the `?length` variable. Our approach is conform to SPARQL 1.1, i.e. `FILTER` and `LIMIT` instructions work as expected. We used the property path syntax to en-

[5] Our algorithm is still implemented as a single thread application.

code a shortest path semantic[6]. As an example, the first predicate can also be restricted to always be `dbp:after` with the following query:

```
SELECT *
WHERE {
    dbr:Felipe_Massa !:* dbr:Red_Bull.
    FILTER(?r1 = dbp:after).
}
LIMIT 3
```

?length	?r0	?r1	?r2	?r3	?r4	?r5	?r6
7	dbr:Felipe _Massa	dbp:after	dbr:Robert _Kubica	dbp:first Win	dbr:2008 _Canadian _Grand _Prix	dbp:third Team	dbr:Red _Bull
7	dbr:Felipe _Massa	dbp:after	dbr:Robert _Kubica	dbo:first Win	dbr:2008 _Canadian _Grand _Prix	dbp:third Team	dbr:Red _Bull
7	dbr:Felipe _Massa	dbp:after	dbr:Robert _Kubica	dbo:first Win	dbr:2008 _Canadian _Grand _Prix	dbo:third Team	dbr:Red _Bull

Table 4. Example SPARQL response

A possible solution is depicted in Table 4. The downside of this approach is that all paths have to be computed to calculate the maximum amount of necessary variables. Another possibility is to name the variables directly in the query. With the help of the variable length, one can check if the amount of variables given in the query is enough. If this is not the case, the requester can create a new query with more variables. In the following, a brief overview of our SPARQL endpoint's implementation is shown. It uses the proposed path algorithm and responds to `SELECT` queries like in the given example above. We used FUSEKI SPARQL server based on Jena introducing a new Jena subsystem called `GraphQuery`. The `QueryEngineMain` of Jena is extended with an `OpExecutor`[7], making it possible to replace the evaluation of a specific operation. In this use case only the path operation is changed and all other operations are executed as usual. If the subject as well as the object are not variables, the proposed algorithm will be executed. We had to modify the evaluation of the query to prevent

[6] Given two resources A and B, A as the source and B as the target, the corresponding SPARQL query is `A !:* B`. Technically, this searches for zero or more occurrences of properties (indicated by "*") between A and B not ("!") matching an introduced fake IRI ":".

[7] `https://jena.apache.org/documentation/query/arq-query-eval.html`

the removal of equalities in `FILTER` elements. Otherwise all variables which are not explicitly defined like our hidden ones, would be removed. As a consequence the `FILTER` statement would not have any effect. Creating or updating the actual dataset triggers the creation of the corresponding graph index. To create a running FUSEKI instance for it, we used the Jena Assembler specification.

In summary, we now have a running FUSEKI server which is able to retrieve all shortest path for a given SPARQL property path query.

8 Conclusion and Future Work

In this paper we presented several approaches for finding top-k shortest paths and pointed out why they are not directly applicable for our specified use case. Based on Eppstein's algorithm we therefore implemented our own solution method which induces only moderate overhead. We were able to solve all queries given in the ESWC 2016 Top-K Shortest Path Challenge and additionally provided further details on our algorithm's performance. One general problem was the overhead resulting from loops in the graph.

During the implementation we already identified some potential for improvements concerning time and memory consumption which was not exploited due to time reasons. For future work we could lazily build the different heaps and try to predict invalid paths earlier. Since our algorithm would not need to follow these paths the resulting overhead is far less.

Besides, a concept for embedding our algorithm into a SPARQL endpoint was provided.

Acknowledgement

This work was partially funded by the BMBF project Multimedia Opinion Mining (MOM: 01WI15002).

References

1. H. Aljazzar and S. Leue. K*: A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence*, 175(18):2129–2154, 2011.
2. D. Eppstein. Finding the k shortest paths. *SIAM Journal on computing*, 28(2):652–673, 1998.
3. A. Gubichev and T. Neumann. Path query processing on very large rdf graphs. In *WebDB*. Citeseer, 2011.
4. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3), 2011.
5. V. M. Jiménez and A. Marzal. A lazy version of eppstein's k shortest paths algorithm. In *Experimental and efficient algorithms*, pages 179–191. Springer, 2003.
6. E. V. Kostylev, J. L. Reutter, M. Romero, and D. Vrgoč. Sparql with property paths. In *The Semantic Web-ISWC 2015*, pages 3–18. Springer, 2015.

7. J. Lehmann, J. Schüppel, and S. Auer. Discovering unknown connections-the dbpedia relationship finder. *CSSW*, 113:99–110, 2007.
8. K. Losemann and W. Martens. The complexity of evaluating path expressions in sparql. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 101–112. ACM, 2012.
9. M. Morsey, J. Lehmann, S. Auer, and A.-C. Ngonga Ngomo. DBpedia SPARQL Benchmark—Performance Assessment with Real Queries on Real Data. In *ISWC 2011*, 2011.
10. M. Morsey, J. Lehmann, S. Auer, and A.-C. Ngonga Ngomo. Usage-Centric Benchmarking of RDF Triple Stores. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI 2012)*, 2012.
11. M. Przyjaciel-Zablocki, A. Schätzle, T. Hornung, and G. Lausen. Rdfpath: Path query processing on large rdf graphs with mapreduce. In *The Semantic Web: ESWC 2011 Workshops*, pages 50–64. Springer, 2011.
12. J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.