Erwin Filtz and Vadim Savenkov and Jürgen Umbrich

On finding the k shortest paths in RDF data

Article (Published)
(Refereed)

On finding the k shortest paths in RDF data

Erwin Filtz, Vadim Savenkov, Jürgen Umbrich

Vienna University of Economics and Business
Institute for Information Business
{firstname.lastname}@wu.ac.at

Abstract. Finding relationships between entities in RDF data is in the heart of many exploration tasks. General path enumeration algorithms are typically used for computing such relationships, finding top k shortest paths being of special interest. The k shortest paths problem has been thoroughly studied for the weighted graph case, the two most popular generic algorithms are due to Eppstein and to Yen. Along with the Dijkstra's shortest path algorithm upon which they build, these two algorithms are available in most libraries and graph databases. In the RDF context, however, the graph is unlabeled but can have multi-edges, and the found paths can contain cycles, so applying the mentioned algorithms is either impossible (Yen's) or suboptimal. It is a folklore knowledge that the traditional breadth first search (BFS) can be easily adapted to compute the k shortest paths. However, for dense graphs and large k, both time and memory consumption become critical. We discuss two BFS adaptations which are easy to implement and substantially boost performance when solving the k shortest paths problem.

1 Introduction

The size of online linked datasets can be very large [3], and even when the source domain of a dataset is known in advance and a general meta-data such as provenance and recency is available, a detailed exploration and accurate characterization of a dataset with respect to a given task in a limited time remains a complex and challenging task. As long as graph-oriented data such as RDF is concerned, enumerating complex relationships between entities is a cornerstone exploration routine (e.g., [15]). This routine can be cast as the *k shortest paths* or *k shortest paths routing* problem.

The importance of routing in graphs certainly goes far beyond data exploration, it is perhaps one of the most ubiquitous problems in computer science with applications ranging from networking to artificial intelligence. In many cases, routes in weighted graphs need to be found, where weights might refer to bandwidth, travel time or a rank. In the simplest setting, unit edge weights are assumed. The algorithm by Dijkstra [10] and its extension A* [14] are used most frequently for finding optimal paths, and the best known extensions for producing the top k paths are the algorithms by Eppstein [11] and Yen [20]. Many popular graph libraries, e.g. JGraph [1] and graph databases (e.g. Neo4J [2], including custom extensions [15]) only support these general algorithms capable of

routing in weighted graphs for solving routing problems. Notably, in the recent ESWC 2016 challenge, none of the finalists has used the basic BFS algorithm for the k shortest paths task. The greater flexibility comes at a certain price however, which can be negligible for smaller datasets but becomes significant for larger ones. In this paper we experiment with k shortest paths algorithms specifically tailored for unweighted labeled graphs, which are useful for exploring and querying RDF datasets.

The k shortest path problem has not received much attention in the literature in this more specific setting. Probably, one reason for that was a folklore knowledge that one can adapt the breadth first search (BFS) for computing both the shortest path and the top k shortest paths alike, and that such an algorithm is fairly easy to implement. However, when graphs are large and dense, one has to be careful about the BFS based procedures, as the number of intermediate results that need to kept in memory becomes large. We take a closer look on the BFS for k shortest paths and explore several optimizations significantly improving a naïve approach, allowing it to outperform the general Eppstein's algorithm even on unlabeled graphs with unit edge weights. There are two main components of our solution. The first component is the use of a disk based graph index for coping with large graphs. Our main use case being linked data processing, we opted for the HDT library [12] which is a natural choice for the RDF data. HDT also compresses the data, by replacing verbose RDF strings with integers in a principled way, the graph is then loaded in memory incrementally. The second component is the adaptation of BFS tailored specifically towards supporting the k shortest paths computation. One improvement here is the tree based data structure for maintaining candidate paths efficiently, another improvement is bidirectional search. Altogether this results in a very simple algorithm which outperforms the state of the art solutions presented at the ESWC 2016 competition.

Related work. Finding paths in graphs is one of the most well studied topics in computer science. Many algorithms have been developed solve the single-source-shortest path problem (e.g. [10, 11, 20]) as well as the all-pair-shortest path problem (e.g. [6, 13, 18]).

As far as the k shortest paths problem (or k shortest path routing) is concerned, one of the most famous algorithms is by Yen [20] that builds upon Dikstra's algorithm to find next best *acyclic* paths, and a more general algorithm by Eppstein [11] that also supports cyclic paths. Numerous modifications to these algorithms have been proposed. For instance, the Lawler's variant of Yen's algorithm [19] avoids duplicate path calculation [7]. and Dijkstra's algorithm [10] with improvements [4], or using heuristics [5, 9]. All mentioned algorithms are general enough to solve the k shortest path routing problem in directed weighted graphs, where "length" of the path refers to the sum of the edge weights in it.

RDF graphs have certain peculiarities relevant for the choice of routing algorithms. In particular, they are *labeled multigraphs*, i.e. multiple edges with different labels, corresponding to the RDF predicate names and can connect the same pair of nodes. Such labels are typically important for the graph ex-

Algorithm 1 Baseline BFS for k shortest paths

```
1: procedure BaselineBFS(G, start, target, k)
2:     solutions ← ∅
3:     q ← [start]
4:     while q not empty do
5:         p ← q.poll()          ▷ The first element p of the queue q is removed from q
6:         n ← p.lastNode()                ▷ The the last node in the path p is n
7:         for edges e(n, n′) ∈ G do
8:             p′ ← p.append(e(n, n′))
9:             if n′ == target then
10:                solutions ← solutions.append(p′)
11:                if solutions.size ≥ k then
12:                    return solutions
13:                end if
14:            end if
15:            q ← q.append(p′)
16:        end for
17:    end while
18:    return solutions
19: end procedure
```

ploration, paths consisting of different sequences of properties capture different relationships between concepts and thus need to be distinguished (see, e.g. the case of relationship visualization[1]). At the same time, weights of predicates are rarely defined: in this case, path length is determined by a number of edges. This generally makes algorithms for weighted graphs suboptimal for the RDF domain. Yet, Eppstein's algorithm is quite often used for RDF graph exploration deriving the shortest path from the shortest paths tree containing only the necessary parts of the graph calculated by Eppstein's algorithm [17]. An algorithm to find the top-k shortest paths in directed labeled multi-edge graphs is proposed at the ESWC top-k shortest paths in large typed RDF graph challenge[2] with immediate validity check for path query restrictions and no triple is allowed to be repeated [16].

Contribution and outline Our contribution is the assessment of the BFS based approaches for the exploration of RDF graphs under the HDT compression. We experiment with the DBpedia RDF graphs stemming from the ESWC 2016 challenge.

The remainder of this paper is structured as follows: after setting out the prelimirary definitions in Section 2, in Section 3 we present our algorithms and then report on the evaluation results in Section 4. Section 5 offers concluding remarks.

[1] `http://www.visualdataweb.org/relfinder.php`

[2] `http://2016.eswc-conferences.org/top-k-shortest-path-large-typed-rdf-graphs-challenge`

2 Preliminaries

We consider a graph G to consist of a set of nodes (vertices) V, edge labels L and directed edges labeled by values from L: $G = (V, L, E)$, where every edge in E is a triple (x, ℓ, y) with x called *source node*, y called *target node* $(x, y \in V)$ and $\ell \in L$ is an edge label. We define a *path* in $G = (V, L, E)$ to be a sequence $(e_1, \ldots, e_n)$ of *unique edges* from E — i.e., $e_k \neq e_m$ for all integer $k, m \leq n$, in which edges e_i and e_{i+1} are adjacent if and only if the target node of e_i equals the source node of e_{i+1}. 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 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 as subsequences, as long as no edge is visited twice. Using the 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 $P(s, t)$, $s, t \in V$ we denote the set of all paths from s to t in G, and by $P_{asc}(s, t)$ we denote a sequence of all elements of $P(s, t)$ in the order of ascending lengths. For a graph G defined as above, the k shortest paths problem takes the nodes s, t from V as input and outputs the first k elements of a sequence $P_{asc}(s, t)$. Note that the solution is non-deterministic in general, since several paths from s to t of any given length can exist.

Our use case will be RDF graphs $G = (V, L, E)$ in which V is a subset of the set V_{RDF} of resources and L is contained in the set L_{RDF} of predicate names represented by *international resource identifiers* (IRIs). The set E is called a set of *RDF triples* in the RDF terminology. Note that we do not treat data values or blank nodes in any special way here: set V_{RDF} can be defined to contain these types of values as well.

3 Algorithms

A baseline algorithm for the k shortest paths problem using a breadth first search approach (BFS) is given in Fig. 1. The queue q stores all paths starting at node *start* ordered by length. Every path p, extracted from the top of q, gives rise to a set of paths, each extending p by a single edge. Every such extended path p' is again queued in q. Whenever the added edge leads to the node *target*, p' is also appended to the list *solutions*. The procedure terminates as soon as k solutions have been found or all nodes in the graph have been explored.

Our first observation here is that copying the prefix p into each new path p' should be avoided: instead of paths, q could store *tuples with references* of the form (n, e, pr). Here, n is a graph node, e an incoming edge of n, and pr is a reference to the tuple (n', e', pr') representing the graph node n' which is the starting node of the incoming edge e. The link pr points to the tuple that represents a preceding node in the path, and several tuples can link to the same predecessor tuple, avoiding the duplication of path prefixes. Thus, references pr organize the tuples into a tree with the root $(start, \mathsf{null}, \mathsf{null})$. The tree can be

Algorithm 2 BFS for k shortest paths

```
 1: procedure KShortestBFS(G, start, target, k)
 2:     solutions ← ∅
 3:     q ← (start, null, null)
 4:     while q not empty do
 5:         (n, e, pr) ← q.poll()
 6:         p ← trace((n, e, pr))          ▷ Walk down the pr links, collect edges & nodes
 7:         for edges e′(n, n′) ∈ G do
 8:             p′ ← p.append(e′(n, n′))
 9:             if n′ == target then
10:                 solutions ← solutions.append(p′)
11:                 if solutions.size ≥ k then
12:                     return solutions
13:                 end if
14:             end if
15:             pr′ ← referenceTo((n, e, pr))
16:             q ← q.append((n′, e′, pr′))
17:         end for
18:     end while
19:     return solutions
20: end procedure
```

traversed from leaves to the root, each traversal corresponding to a candidate path in the graph. These leaf nodes (and not complete paths like in the baseline Algorithm 1) are now stored in the queue q, see Algorithm 2.

Our second improvement over the baseline algorithm is the bidirectional search, listed as Algorithm 3. The paths are now explored not node at a time, but rather *hop at a time*: at iteration i, all paths of length i with the start node *start* are found and placed in a list called *frontier*. Two frontiers, *forward and backward*, are maintained by the algorithm. At each iteration of the algorithm, every path in a frontier is replaced by all paths extending it by a single edge. This operation is called *frontier advancement*. Paths in the forward frontier can be traced back to the *start* node, and the paths in the backward frontier can be traced to the *target* node, following the *pr* links. By joining the two frontiers on graph nodes after each frontier advancement, the BidirectionalBFS procedure discovers all paths from *start* to *target* in the natural order.

Last but not least (although not directly related to the search algorithm), we use the RDF compression approach HDT [12] to reduce the required size of the datasets. HDT internally represents graph nodes and edge labels as integers rather than strings in a way that assigns shorter values to frequently used values. It also allows for incremental loading of the graph into memory, thus facilitating the exploration of large graphs.

Algorithm 3 Bidirectional BFS

```
1: procedure BidirectionalBFS(G, start, target, k)
2:     solutions ← ∅
3:     f_f ← (start, null, null)
4:     f_b ← (target, null, null)
5:     f_act ← f_f; f_pass ← f_b                    ▷ Aliases marking one frontier as active
6:     while any of f_f, f_b not empty and solutions.size ≥ k do
7:         advance(f_act)                           ▷ Advance all paths in the active frontier
8:         solutions ← solutions.appendAll(join(f_f, f_b))
9:         swap(f_act, f_pass)
10:    end while
11:    return solutions
12: end procedure

13: function join(f_f, f_b)
14:    result ← ∅
15:    for (n, e_1, pr_1) ∈ f_f, (n, e_2, pr_2) ∈ f_b do
16:        ▷ Concatenate traces obtained by traversing the respective links pr_i:
17:        result ← result.appendAll(trace((n, e_1, pr_1))·trace((n, e_2, pr_2)))
18:    end for
19:    return result
20: end function
```

4 Evaluation

We evaluate the KShortestBFS and BidirectionalBFS algorithms from Section 3 in two experiments. The first experiment focuses on studying the algorithms in term of runtime and memory usage for different top-k shortest path tasks and varying size datasets. The second experiment compares our approach to state-of-the-art algorithms and reported runtimes. In both experiments we are using the tasks and datasets provided by the ESWC 2016 top-k shortest path challenge.[3]

4.1 Datasets and Evaluation Setup

Table 1 shows our two evaluation datasets which are available online.[4] The datasets are provided in *nt*-format, differ in size (2 GB vs 19 GB raw size) and in the average degree of the nodes. Note that the diameter (d) of the large data set grows slower than the number of the nodes and edges, and the density (D) of the large graph is actually lower than in the small graph.

Tasks: The tasks for our evaluation are also taken from the ESWC 2016 challenge homepage and are divided into two groups. In task 1, k paths between two given nodes need to be found in an ascending order of their length. Task 2 adds an

[3] `http://2016.eswc-conferences.org/top-k-shortest-path-large-typed-rdf-graphs-challenge`
[4] `https://bitbucket.org/ipapadakis/eswc2016-challenge`

Table 1. DBpedia dataset properties

Data set	#Triples	Raw size	HDT size	$\|V\|$	Avg. deg.	D	d
Small	9,996,907	2 GB	532 MB	186679	6.12	1.64×10^{-5}	138
Large	110,621,287	19 GB	1.9 GB	1187306	10.40	4.38×10^{-6}	201

edge restriction to path, which means that the first or the last edge in a path must meet a given criteria, hence a specific edge label.

Experiments: We run two experiments: in the first one, we study the impact of the optimisations outlined in Section 3. All 17 queries — that is, triples $(source, target, k)$ defining the instances of the k shortest paths problem — belong to the first task of the ESWC 2016 challenge, asking for paths with no constraints besides suppressing repeated edges, with the values of k ranging from 2 to 175,560. Filtering out solutions with repeated edges (which we do not count as paths according to the definitions in Section 2) is not shown in the listings of Algorithms 1–3 for the sake of simplicity. Besides better alignment with the requirements of the ESWC 2016 challenge, disallowing repeated edges allows for obtaining self contained results in the following sense: acyclic paths (that is, paths without repeated nodes) can be obtained by further filtering the solutions, and paths with repeated edges can be computed by iterating the cycles in the solutions. In both cases, the graph does not need to be searched again.

In the second experiment, we compare to the Eppstein's algorithm implemented by Brandon Smock [5] run on a version of the graph *without edge labels*, and also to the results of the ESWC 2016 challenge finalists whose results we could find in the respective competition reports. For the latter comparison, we also use *the constrained version* of the k shortest paths problem (Task 2 of the ESWC 2016 challenge) in which paths are required to start or end with an edge having specific label. For all queries there are four different start / target node combinations with three to six different increasing k queries.

Evaluation metrics & implementation We measure for each tasks the time elapsed to compute the k shortest path and the maximum memory usage. To measure the memory, we run a background thread which measures every 5 ms the current memory allocation and returns the maximum allocation. All our algorithms are implemented in Java 1.7. For each run, we always start a new JVM instance to avoid hotspot compiler optimisations in JAVA and also perform aggressive garbage collection.

All algorithms and evaluation data are available online for reproducibility.[6]

[5] `https://github.com/bsmock/k-shortest-paths`
[6] `https://bitbucket.org/vadim_savenkov/k-shortest-paths`

4.2 Evaluation results

The results for the first experiments are depict in Figure 1 (runtime) and Figure 2 (memory usage) and show the average over 10 runs for the two datasets (small and large) and the two algorithms KShortestBFS (bfs) and BidirectionalBFS (bibfs). The x-axis shows the tasks, ordered by their start and end node and parameter k. The y-axis shows in logscale the required time and memory usage, respectively. The labelling is aligned with the task name of the ESWC 2016 challenge, e.g. q1_8 indicates query1 of task 1 which asks for the top-8 path. Please note, that the reported runtimes do not include the time needed to output the solution paths, that is, converting the internal HDT ids to IRIs.

The second important aspect for an algorithm is the memory consumption. Even though memory is not as expensive as some years ago, we still strive for a low memory usage and not to store more information as required. Regarding the computation time (cf. Figure 1), the results show that the BidirectionalBFS algorithm outperforms the KShortestBFS algorithm for both datasets in all tasks by up to 3 orders of magnitudes. While in general the runtime increases with the parameter k, we can observe some interesting results for $q2$ for which the runtime stays more or less constant (cf. q2_79 and q2_154). One reason for this observation might be the graph structure and that the shortest path are sharing many sub path. A similar behaviour can be observed for the task q1_344 and q1_1068 with the BidirectionalBFS algorithm.

Regarding the maximum memory consumption (cf. Figure 2), our experiment again shows that the bidirectional search algorithms outperforms the one directional search by up to 1 order of magnitude. We measured a maximum memory allocation of $\sim$10Gb for large k. An interesting observation is that the memory consumption for the q2 tasks seems to be fairly constant independent of k, while the task q1 requires more memory with increasing k. This again indicates a strong influence of the graph structure and the performance, while we also see that the BidirectionalBFS algorithm is less affected by the underlying structure.

Overall, our results clearly show that for the tasks and datasets of the ESWC challenge a bidirectional top-k shortest path search algorithms outperforms the straight forward breadth-first search algorithm in terms of computation time and memory consumption. As expected, the structure of the graph which is traversed from the start or target node directly influence the evaluation measures. A bidirectional search seems to be less influenced by the structure.

Next, we compare our winning BidirectionalBFS algorithm with the reported results of the participants of the ESWC challenge and compare also to an implementation of the Eppstin algorithm. The ESWC 2016 challenge published for the larger dataset two queries each for the two tasks: all four resulting queries have the same start and target node. The unconstrained (Task 1 of the challenge) queries have k set to 337 (*T1Q1*) and 53008 (*T1Q2*), the queries with edge restriction (Task 2 of the challenge) have k 374 (*T2Q2*) and 52664 (*T2Q2*). The edge restriction of the "T2"-queries limits the first or the last predicate in the property path be `http://dbpedia.org/property/after`. The Eppstein al-

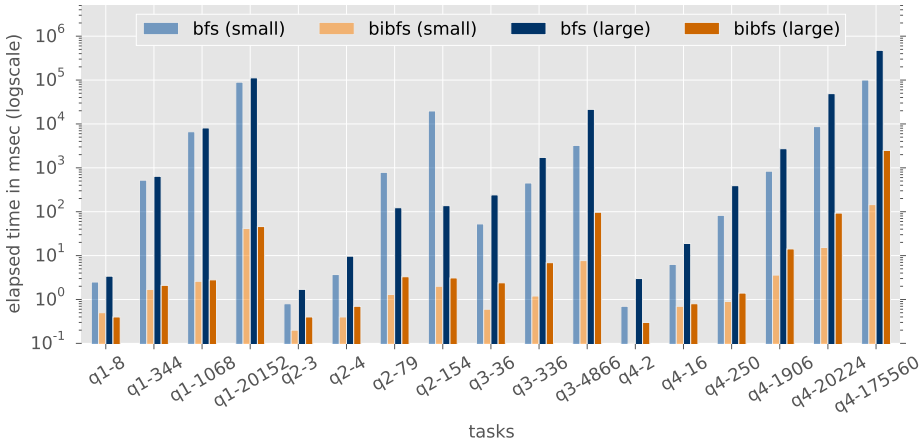

Fig. 1. Avg. elapsed time over 10 runs

gorithm has been run on a version of the graph in which IRIs are replaced by integers and with no multiedges (i.e., replaced by a single edge with weight 1). This results in a smaller, less complex dataset for the case of the Eppstein's algorithm (whereas the BidirectionalBFS runs on the original multigraph). The two queries with edge restrictions are not evaluated for the Eppstein algorithm, as the version we used for comparison does not support multiedges. Please note, that for a better comparison, we also included in the reported runtimes the output of the results, on contrast to the first experiment.

The challenge submission by Hertling et.al. [16] uses a modified version of Eppstein's algorithm to solve challenge queries (labelled as "Eppstein modified"). The modifications made are such that every computed path is immediately checked for its validity (no recurring node - edge - node triple in the path) and only valid paths are added to the result list. Furthermore, paths with multiple sidetracks are pruned. The second challenger uses triple pattern fragments to solve the challenge queries with a streaming algorithm [8], labelled "TPF". We added the reported runtimes from the challenge submission as reference values to our figure.

The evaluation results are depict in Figure 3 and show that the BidirectionalBFS algorithm outperforms the algorithms proposed by the other challengers and the standard Eppstein algorithm by 3 orders of magnitude. The implementation of the standard Eppstein algorithms shows in general the slowest performance. We assume that the slow retrieval times of the TPF is due to establishing http connections and the transfer of the result path over the network which adds some overhead. We also see that the optimisations of the modified Eppstein algorithms slightly outperforms the original algorithm. In contrast, our approach seems to benefit from the compression and the fast lookups in HDT and that the approach only loads the relevant parts of the graph into memory.

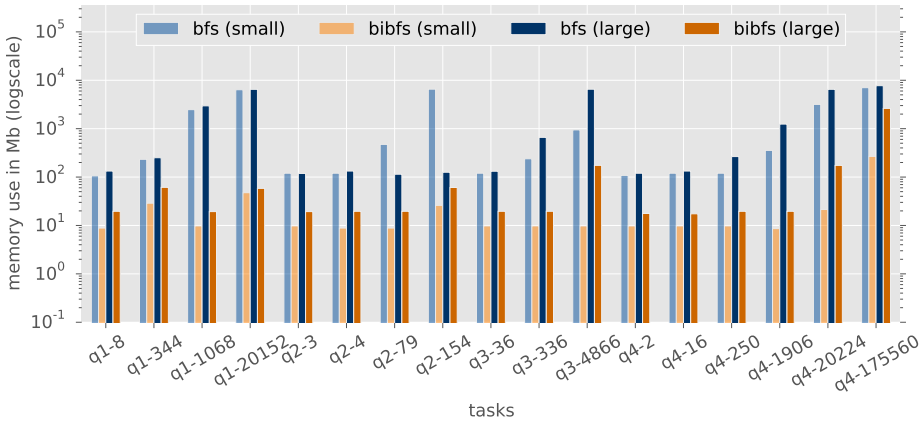

Fig. 2. Avg. max memory usage over 10 runs

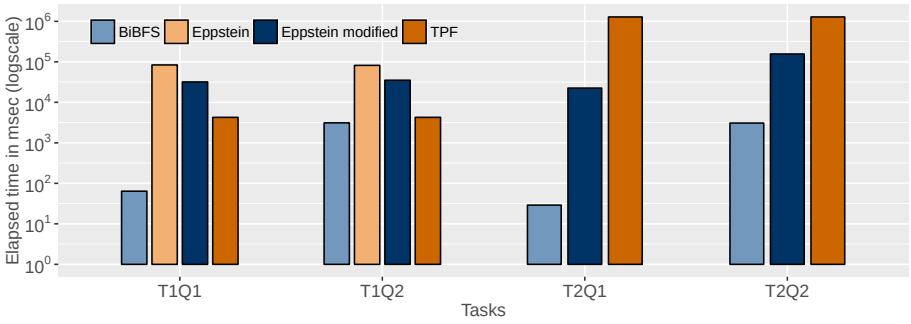

Fig. 3. Experiment 2: Performance of BIDIRECTIONALBFS against other algorithms

5 Discussion and Conclusion

The task of finding paths in a graph has been of interest since decades. The constantly growing size of datasets cannot be fully absorbed by the increase in computational power over the last years, which motivates the research of time and memory efficient algorithms for processing large amounts of data.

In this paper we designed and experimented with a bidirectional breadth-first search algorithm to find the k shortest paths in a graph and applied it to the DBpedia dataset stemming from the ESWC 2016 challenge. The training dataset of the challenge (a small dataset of Table 1) for which complete results have been published by the challenge organizers, allowed us, in particular, to validate the correctness of our implementation and to put it in the context of other approaches. Our experiments reported in Section 4 demonstrate that a memory intensive BFS approach can be adopted for solving the k shortest paths problem in a time and memory efficient way, outperforming implementations

using other top-k algorithms. In addition, our results again confirmed that the underlying graph structure influences the runtime and memory usage.

Our future work can be grouped into two directions. The first direction is to further optimise our algorithm. We will study in more detail the correlation between graph structure and performance of the algorithm and research optimisations for particular graph characteristics. Another challenge we plan to address is to detect/approximate if and how many paths exists between two nodes and as such avoiding to load the whole graph into memory in the worst case.

The second direction in future work is to applied and study the bidirectional BFS-based search algorithm to graphs of different domains. We plan to apply it for exploring transaction networks of virtual currencies like Bitcoin to find relations between addresses. Several different paths between the same addresses might be an indicator for criminal activities where users try to obfuscate the flow of Bitcoins. Supporting path constraints based on regular expressions will be an important next step as well, making the algorithm usable in the graph database domain (e.g., as a Neo4J plugin or in SPARQL processors for the path query evaluation).

6 Acknowledgement

This work was funded by the Austrian research funding association (FFG) under the scope of the ICT of the Future program (contract # 849906).

References

1. JGraphT, https://jgrapht.org/
2. Neo4j–The World's Leading Graph Database, https://neo4j.com/
3. Rdf dumps, https://www.w3.org/wiki/DataSetRDFDumps
4. Ahuja, R.K., Mehlhorn, K., Orlin, J., Tarjan, R.E.: Faster algorithms for the shortest path problem. Journal of the ACM (JACM) 37(2), 213–223 (1990)
5. Aljazzar, H., Leue, S.: K∗: A heuristic search algorithm for finding the k shortest paths. Artif. Intell. 175(18), 2129–2154 (2011), http://dx.doi.org/10.1016/j.artint.2011.07.003
6. Bellman, R.: On a routing problem. Tech. rep., DTIC Document (1956)
7. Brander, A.W., Sinclair, M.C.: A Comparative Study of k-Shortest Path Algorithms, pp. 370–379. Springer London, London (1996), http://dx.doi.org/10.1007/978-1-4471-1007-1_25
8. De Vocht, L., Verborgh, R., Mannens, E., Van de Walle, R.: using Triple Pattern Fragments to Enable Streaming of Top-k Shortest Paths via the Web. Tech. rep., Ghent University, iMinds - Data Science Lab (2016)
9. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation, chap. Engineering Route Planning Algorithms, pp. 117–139. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02094-0_7
10. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1(1), 269–271 (1959), http://dx.doi.org/10.1007/BF01386390

11. Eppstein, D.: Finding the k shortest paths. SIAM J. Comput. 28(2), 652–673 (Feb 1999), http://dx.doi.org/10.1137/S0097539795290477
12. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary rdf representation for publication and exchange (hdt). Web Semantics: Science, Services and Agents on the World Wide Web 19, 22–41 (2013), http://www.websemanticsjournal.org/index.php/ps/article/view/328
13. Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM 5(6), 345–345 (Jun 1962), http://doi.acm.org/10.1145/367766.368168
14. Hart, P.E., Nilsson, N.J., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics 4(2), 100–107 (July 1968)
15. Heim, P., Hellmann, S., Lehmann, J., Lohmann, S., Stegemann, T.: Semantic Multimedia: 4th International Conference on Semantic and Digital Media Technologies, SAMT 2009 Graz, Austria, December 2-4, 2009 Proceedings, chap. RelFinder: Revealing Relationships in RDF Knowledge Bases, pp. 182–187. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-10543-2_21
16. Hertling, S., Schröder, M., Jilek, C., Dengel, A.: Top-k Shortest Paths in Directed Labeled Multigraphs. Tech. rep., Knowledge Management Group, German Research Center for Artifical Intelligence (DFKI) GmbH (2016)
17. Jiménez, V.M., Marzal, A.: Experimental and Efficient Algorithms: Second International Workshop, WEA 2003, Ascona, Switzerland, May 26–28, 2003 Proceedings, chap. A Lazy Version of Eppstein's K Shortest Paths Algorithm, pp. 179–191. Springer Berlin Heidelberg, Berlin, Heidelberg (2003), http://dx.doi.org/10.1007/3-540-44867-5_14
18. Johnson, D.B.: Efficient algorithms for shortest paths in sparse networks. J. ACM 24(1), 1–13 (Jan 1977), http://doi.acm.org/10.1145/321992.321993
19. Lawler, E.L.: A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. Management Science 18(7), 401–405 (1972), http://dx.doi.org/10.1287/mnsc.18.7.401
20. Yen, J.Y.: Finding the k shortest loopless paths in a network. Management Science 17(11), 712–716 (1971), http://dx.doi.org/10.1287/mnsc.17.11.712