Finding Top K Shortest Simple Paths with Improved Space Efficiency

Qingsong Wen[1], Ren Chen[1], Lifeng Nai[2], Li Zhou[3], Yinglong Xia[1]
[1]Huawei Research America, Santa Clara, CA 95050, USA
[2]Georgia Institute of Technology, Atlanta, GA 30332, USA
[3]The Ohio State University, Columbus, OH 43210, USA
{Qingsong.Wen,Ren.Chen,Yinglong.Xia}@huawei.com,lnai3@gatech.edu,zholi@cse.ohio-state.edu

ABSTRACT

Finding top-K shortest paths is fundamental and crucial to many graph applications, but known to be nontrivial over large graph data and large value of K. This problem becomes much more challenging when the shortest paths require to be simple (paths without loops). When searching for top-K shortest simple paths, MPS algorithm is a practically fast and efficient scheme based on the famous Yen's algorithm. In this paper, we propose an improved MPS algorithm which can significantly reduce the memory consumption and increase the execution speed compared to the original MPS algorithm. First, we design a pruning scheme during the construction of pseudo-tree, such that only the shortest path in each iteration would be added to the pseudo-tree, instead of adding all possible candidate paths as that in the original MPS algorithm. Second, we modify the pseudo-tree of shortest-path candidates with reversed order and internal ID, such that the shortest paths can be retrieved directly from the constructed pseudo-tree without explicitly storing all candidate paths. Furthermore, we evaluate the performance in terms of running time and memory consumption in both synthetic and real graphs with millions of vertices and edges. Compared to the original MPS algorithm, experimental results show that our improved MPS algorithm can bring up to 6x performance gain in both running time and memory consumption.

1 INTRODUCTION

Finding top K shortest paths (KSP) problem has been extensively studied as a generalization of the shortest path problem. Given a graph G with non-negative edge weights, a positive integer K, a source vertex s and a destination vertex t, KSP algorithm ranks the top-K shortest paths from s to t and lists them in increasing order of length. The KSP problem can be found in various real world applications including network routing, chip layout planning, wireless communications, traffic engineering and database applications [2, 7, 9, 10, 14, 16, 17]. Several variants of KSP problems have been examined in the literature [6, 7, 9, 11, 17], which can be classified into two classes including the unconstrained problem and the constrained problem. No constraints need to be considered on the path definition in the unconstrained problem, while some constraints should be satisfied in the constrained problem [10, 11, 17]. For example, some constrained KSP problems require the shortest paths to be simple (loopless), i.e., duplicate vertex in a graph should be excluded in each of the shortest paths. To solve the constrained problem of finding top-K shortest simple paths, an $O(Kn(n \log n+m)$ time algorithm has been proposed by Yen et al. [17], where $n = |V|$ vertices and $m = |E|$ edges; this algorithm is close to the best-case linear speedup predicted by theory. As a more practical and faster implementation by improving Yen's algorithm, MPS algorithm [3, 4] achieves a time complexity of $O(m \log n + Kn)$, when a worst case analysis is considered. As a highly efficient algorithm especially for finding shortest simple paths, MPS algorithm exhibits much faster speed than other top KSP algorithms [7, 9, 17] when searching for top-K shortest simple paths.

In this paper, we propose an improved MPS algorithm by exploiting the available information, such that the unnecessary memory space can be saved. Specifically, we design two novel strategies. The first is to only add the shortest path to the pseudo-tree in each iteration by a tree-pruning method. The second is to apply reversed order and internal ID in the pseudo-tree such that the storing and searching shortest paths would be much more efficient. Compared to the original MPS algorithm, the experiments show that our proposed improved MPS algorithm brings 3x~6x speedup while consuming much smaller memory sapce.

2 SHORTEST PATHS PROBLEM

2.1 Top K shortest simple paths

Finding the shortest path/paths in a network is a fundamental problem or subproblem of many practical applications, which has been extensively investigated in the literature. Given a graph, different shortest path/paths variants can be formulated according to the corresponding application scenarios. The simplest one is the single-pair shortest path problem, where a shortest path from a given source to a given destination is calculated. Another problem is to find the single-source shortest paths (SSSP), where we want to find a shortest path from a given vertex to every other vertex in a graph. Both the problems can be efficiently solved by Dijkstra's algorithm [5]. The more general problem is to find the top K ($K \geq 1$) shortest paths (KSP) for a given source-destination pair in a graph. In this paper, we focus on the general top KSP algorithms, since the performance of such algorithms has become a critical challenge in many practical applications.

Given a graph with non-negative edge weights, KSP algorithm ranks the top-K shortest paths from source vertex to destination

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GRADES'17, Chicago, IL, USA
© 2017 ACM. 978-1-4503-5038-9/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3078447.3078460

vertex and enumerates them in increasing order of length. In many real applications, the shortest paths generated by KSP algorithm are usually required to be simple, i.e., no loops containing two or more vertexes exist in each path.

2.2 MPS algorithm for top K shortest simple paths

MPS algorithm (named after the authors)[4] is a high efficient top KSP algorithm especially for finding shortest loopless paths, which exhibits much faster speed than other top KSP algorithms [7, 9, 17]. Let $G = (V, E)$ be a directed/undirected graph with $n = |V|$ vertices and $m = |E|$ edges. The main idea of MPS algorithm is to improve the shortcomings of the original Yen's algorithm [17]. Specifically, the computational complexity of Yen's algorithm was improved from $O(Kn(n \log n + m)$ to $O(m \log n + Kn)$ by MPS algorithm, when a worst case analysis is considered.

Algorithm 1 The original MPS algorithm for finding top K shortest simple paths [3, 4]

```
   Input: G = (V, E)
   Output: p_k, k = 1, 2, ..., K
 1: Compute T_t^*
 2: Compute c_ij^* for any edge (i, j) ∈ E
 3: Rearrange the set of edges of (V, E) in the sorted forward star form
 4: p_1 ← shortest path from s to t
 5: k ← 1
 6: X ← {p_k}
 7: T_k ← {p_k}
 8: while k < K and X ≠ ∅ do
 9:     p ← shortest path in X
10:     X ← X − {p_k}
11:     if p is loopless then
12:         k ← k+ 1
13:         p_k ← p
14:     end if
15:     v_k ← deviation node of p_k
16:     for each v ∈ p^k_{v_k t} do
17:         if p^k_sv is not loopless then
18:             break
19:         end if
20:         if E(v) − E_{T_k(v)} ≠ ∅ then
21:             find the first edge (v, x) in the set E(v) − E_{T_k}(v),
                such that p^k_sv ⋄ < v, (v, x), x > is loopless
22:             q ← p^k_sv ⋄ < v, (v, x), x > ⋄ p^*_xt
23:             X ← X ∪ {q}
24:             q_vt ← < v, (v, x), x > ⋄ p^*_xt
25:             T_k ← T_k ∪ {q_vt}
26:         end if
27:     end for
28: end while
```

We define the following notations for Algorithm 1, which presents the MPS algorithm for ranking paths:

- X: a set containing distinct paths which are ranked to select the shortest path in kth iteration.
- T_k: a pseudo-tree composed of shortest-path candidates.
- T_t^*: a graph having the same topology of G, where minimal cost of each vertex has been calculated using single-source shortest path algorithm.
- c_{ij}^*: pre-computed reduced cost [4] for any edge $(i, j) \in E$.
- p_k: shortest path found in the kth iteration.
- $p^k_{v_k t}$: a sub-path from v_k to t in p_k, also called as a deviation path of p_k.
- $p^*_{v_k t}$: the shortest path from v_k to t in T_t^*.
- p^k_{sv} : the path from s to v in p_k.
- $p \diamond q$: the concatenation of two paths p and q.
- (u, v) : the edge connecting a pair of vertices u and v.
- $< u, (u, v), v >$: the path containing vertex u, edge (u, v), and vertex v.
- $E(v)$: the set of edges whose tail vertex is v.
- $E_{T_k}(v)$: the set of edges in T_k whose tail vertex is v.

In Algorithm 1, set X containing path candidates for ranking shortest paths is used and initialized with the shortest path p_1. In the kth step, the shortest path candidate in X is selected and popped out as p_k. Then some new path candidates to obtain the $(k + 1)$th shortest path are generated. For this purpose, for each node v in $p^k_{v_k t}$, the shortest path $p^*_{v_k t}$ from v to t whose first edge is not an edge of $E_{T_k}(v)$, will be computed. In Algorithm 1, $p^k_{sv} \diamond p^*_{vt}$ denotes a new candidate for p_{k+1}. Note that p^*_{vt} has been pre-computed when generating T_t^*, which can be used to find the shortest path from any vertex $v \in V$ to t. When all deviation paths for node v have been determined, $E(v) - E_{T_k}(v)$ becomes an empty set.

The MPS algorithm works similarly to Yen's algorithm. Note that T_t^* can be easily computed with classic single-source shortest path algorithms by reversing the orientation of all the edges and considering t as the initial node. The total time complexity of MPS algorithm is $O(m \log n+Kn)$, where determining T_t^* takes $O(m \log n)$ time using the classic Dijistra shortest path algorithm, and ranking K shortest paths needs $O(Kn)$ time. In fact, in the worst case, no more than n different vertexes will be considered after the deviation node, when new candidate paths are being added to the set X. To produce only simple paths, a potential candidate path is examined if it is loopless when constructing the pseudo-tree (see Lines 11-14, 17-19, and 21 in Algorithm 1). Those candidate paths will be dropped if they are not loopless.

3 PROPOSED IMPROVED MPS ALGORITHM

The original MPS algorithm works well in the case of small networks with small values of K. However, the original MPS algorithm would consume excessive amount of memory in case of large networks with large values of K due to storing all candidate paths, which increases the execution time and even stalls because of running out of memory space. Therefore, we propose an improved MPS algorithm (see Algorithm 2) which significantly reduces the memory space to increase execution speed without affecting the final output paths. The main improvement comes from the following two novel designs.

3.1 Pseudo-tree with pruning

The majority of memory consumption of the MPS algorithm comes from constructing the pseudo-tree T_k of shortest-path candidates. Since most of path candidates would not be in the final top K shortest paths in large networks, we propose a novel scheme to add only one path to the pseudo-tree in an iteration, while the original MPS would add a path for each vertex on the deviation path in an iteration. To illustrate the proposed scheme, let us consider a simple network shown in Fig. 1, where the T_t^* (shortest paths of all vertices to the destination vertex t) is also provided. Based on

Algorithm 2 The improved MPS algorithm for finding top K shortest simple paths

Input: $G = (V, E)$
Output: p_k, $k = 1, 2, \ldots, K$
1: Compute T_t^*
2: Compute c_{ij}^* for any edge $(i, j) \in E$
3: Rearrange the set of edges of (V, E) in the sorted forward star form
4: $p_1 \leftarrow$ shortest path from s to t
5: $k \leftarrow 1$
6: $T_k \leftarrow \{p_k\}$
7: Insert(Q, the last vertex's internal ID of p_k)
8: **while** $k < K$ **and** $X \neq \emptyset$ **do**
9: $a \leftarrow$ Extract-min(Q)
10: $p \leftarrow$ find the path whose last vertex's internal ID is a
11: **if** the last vertex (denote as m) of p is not t **then**
12: $p \leftarrow p \diamond p_{mt}^*$
13: **end if**
14: **if** p is loopless **then**
15: $k \leftarrow k+1$
16: $p_k \leftarrow p$
17: **end if**
18: $v_k \leftarrow$ deviation node of p_k
19: **for each** $v \in p_{v_k t}^k$ **do**
20: **if** p_{sv}^k is not loopless **then**
21: break
22: **end if**
23: **if** $E(v) - E_{T_k(v)} \neq \emptyset$ **then**
24: find the first edge (v, x) in the set $E(v) - E_{T_k}(v)$ such that $p_{sv}^k \diamond < v, (v, x), x >$ is loopless
25: $T_k \leftarrow T_k \cup < v, (v, x), x >$
26: Insert(Q, the internal ID of x)
27: **end if**
28: **end for**
29: **end while**

the aforementioned MPS algorithm, we can obtain the pseudo-tree of candidate paths after the first two iterations as shown in Fig. 2. At the second iteration with $k = 2$, the MPS will add a path to destination vertex 6 for each vertex v on the deviation path (i.e., vertices 1, 2, 5). In contrast, our scheme only needs to add one path by designing a tree-pruning scheme. First, at each vertex v on the deviation path, our scheme only needs to add one vertex x instead of one path as shown in Fig. 3 (also see Line 25 of Algorithm 2), where the shaded vertices would not be added in the pseudo-tree unless they are on the top K shortest paths. Even though we only add one vertex each time instead of a candidate path, its final path cost can be obtained based on the information of T_t^*. Next, based on all path costs, the "path" with minimum cost is selected and the corresponding full path is then added in the pseudo-tree (see Lines 9-13 of Algorithm 2), which is the shortest path in current iteration. It can be seen in Fig. 3 that the proposed scheme brings reduced memory consumption. This memory reduction would be significant and brings impressive speedup in large networks, which will be demonstrated in the following experiment section.

3.2 Pseudo-tree with internal ID and reversed order

The second source of major memory consumption in MPS algorithm comes from storing the set of candidate paths (see X in Line 23 in Algorithm 1). Since all the information about the candidate paths is available in the constructed pseudo-tree, we do not need to explicitly store these paths. Here we propose a simple yet efficient way to retrieve the candidate paths without actually storing

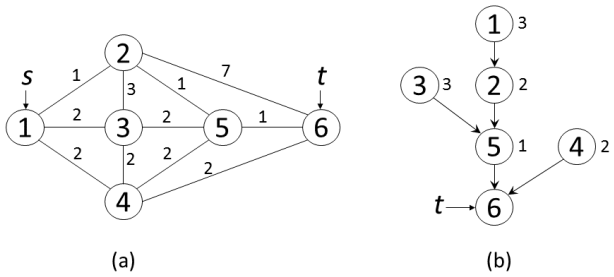

Figure 1: (a) An example network where edge cost is located beside each edge; (b) The corresponding T_t^* calculated by SSSP where the number beside each vertex denotes the minimum sum cost to the destination vertex.

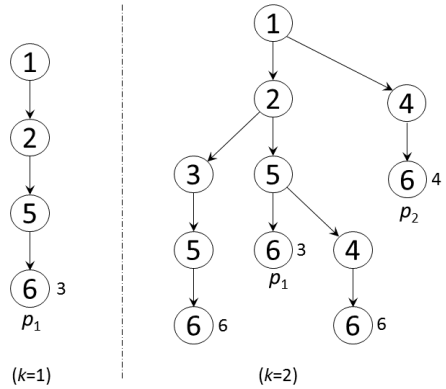

Figure 2: The constructed pseudo-tree from the first two iterations of MPS algorithm. The number beside the leaf vertex denotes the path cost. p_1 and p_2 denote the first and second shortest paths, respectively.

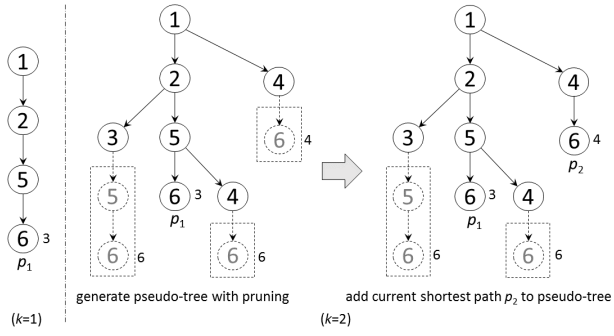

Figure 3: The constructed pseudo-tree from the first two iterations of the improved MPS algorithm with tree pruning. The number on the right hand side of the leaf vertex denotes the path cost. p_1 and p_2 denote the first and second shortest paths, respectively. The shaded vertices would not be added.

them. In the original MPS algorithm, each vertex points to its child vertex/vertices in the pseudo-tree as shown in Figure 2. We adopt

a reversed order such that each vertex points to its parent vertex. Furthermore, we add a distinct internal ID for each vertex as its property. By doing so, we can obtain the whole path from any leaf vertex's internal ID by repeated proceeding from child to parent in the pseudo-tree. This reversed-order with internal ID scheme combining with tree pruning in the pseudo-tree is depicted in Figure 4. During the construction of the pseudo-tree (with or without our proposed pruning scheme), the final cost of each added path or pruned path is also available. Therefore, we can just store the key-value pair (path cost, leaf's internal ID) of each added path or pruned path into a min-priority queue Q (see Lines 7, 26 in Algorithm 2) instead of storing the whole path as in the original MPS algorithm (see Lines 6, 23 in Algorithm 1).

Due to the adopted min-priority queue Q, the operation of finding the leaf's internal ID of the shortest path can be finished in $O(1)$ time (see Line 9 in Algorithm 2). Once the leaf's internal ID is found, the whole shortest path in current iteration can be easily retrieved from the reversed-ordered pseudo-tree as shown in Figure 4 (aslo see Lines 9-13 in Algorithm 2).

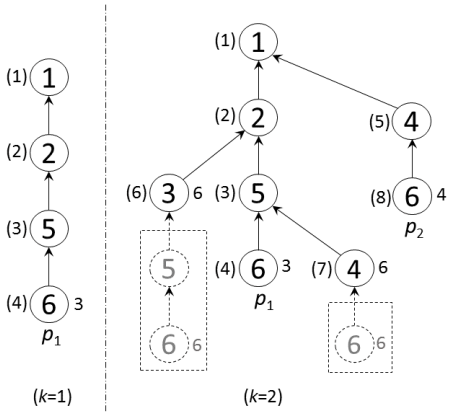

Figure 4: The constructed pseudo-tree from the first two iterations of the improved MPS algorithm with tree pruning, internal ID, and reversed order. The shaded vertices would not be added. The number inside parenthesis on the left hand side of each vertex denotes its internal ID. The internal IDs are distinct, so any path can be fetched based on its last vertex's internal ID. The number on the right hand side of the leaf vertex denotes the final path cost (the cost from s to t), even though the path is pruned.

4 EVALUATION

4.1 Evaluation methodology

In this section, we compare the performance of our improved MPS algorithm against the original MPS algorithm. To minimize the running time, we implement both algorithms in C++ on top of the GraphBIG [12]. Here GraphBIG is adopted for basic graph operations, since GraphBIG is an open-sourced efficient graph framework similar to the IBM System G library [15] and covers major graph computing types and data structures. The experiments are performed on a desktop with Intel i7-6700 CPU (3.4 GHz), 64 GB memory, and Ubuntu 16.04 operating system. Note that both algorithms do not benefit from multi-core parallel computing due to no parallel implementations at current stage. It is expect that similar results can be obtained in parallel computing since the two algorithms follow similar execution flow. The running time of each algorithm is measured between the input graph is loaded into memory and all shortest paths are written into output files.

Table 1: The number of vertices and edges for each generated grid network.

Grid Network	32×32	128×128	512×512	2048×2048
Vertex Number n	1,024	16,384	262,144	4,194,304
Edge Number m	1,984	32,512	523,264	8,384,512

4.2 Evaluation with synthetic grid networks

Without loss of generality, we consider synthetic square grid networks similar to [9] and [7] for ad hoc networks, where each vertex is connected to its four neighboring vertices with random edge weight uniformly distributed in (0, 10). Four grid networks with different sizes, from thousands to millions vertices and edges, are generated in the experiments, which are summarized in Table 1. For each experiment of searching top K shortest paths, we randomly select 50 pairs of source and destination vertices, located on the opposite sides of grid networks, to record the total running time.

Running time: First, we consider the case of finding top 10 shortest paths under different grid network configurations. The total running time of 50 pairs under different grid networks are depicted in Figure 5. In all grid networks, our improved MPS algorithm brings around 3x∼4x speedup over the original MPS algorithm. Second, we evaluate the total running time of 50 pairs of finding top K shortest paths under different K values in a 128×128 grid network, where the value of K is selected from 10 to 10,000. The results are depicted in Figure 6 by log-log scale. It can be seen that our improved MPS consistently provides 4x∼6x speedup over the original MPS algorithm.

Similar speedup can be obtained in the case of other grid network and K value configurations. However, when the network and the value of K are large to some extent, the original MPS slows down rapidly. For example, in a case of finding top-100 shortest paths of 50 pairs under 2048×2048 grid network, the improved MPS algorithm requires about 13 minutes running time while the original MPS algorithm needs over 50 hours. This is due to the excessive amount of space requirements of storing all candidate paths in the original MPS algorithm which may exceed the 64 GB memory in the desktop computer.

Memory: To measure the improvement of memory consumption for our improved MPS method, we perform experiments by utilizing the massif tool in *Valgrind* [13]. In our experiment, we compare the memory consumption of the original MPS and our improved MPS under different grid networks and different K values. As shown in Figure 7, compared to our improved MPS, the original MPS consumes up to 6.7× memory when K=100 and 2.5×

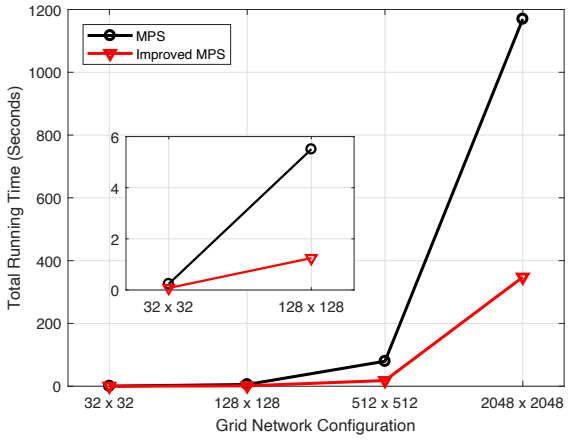

Figure 5: Total running time of 50 pairs of finding top 10 shortest paths under different grid networks.

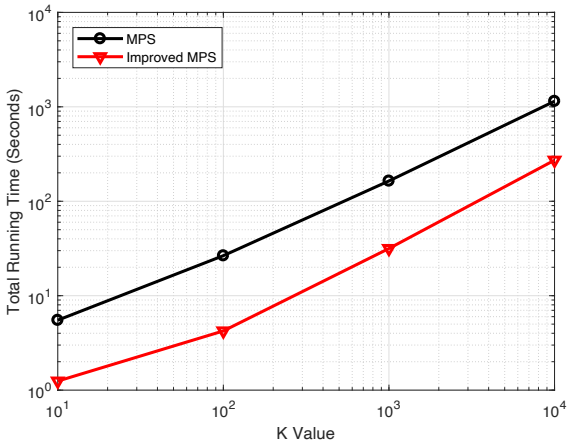

Figure 6: Total running time of 50 pairs of finding top K shortest paths under different K values in a 128 × 128 grid network.

when K=10. With the increment of network size, the memory consumption of both cases grows dramatically. For example, with the original MPS, 32×32 grid network consumes only 17MB memory when K=100. However, when the grid size gets to 2048×2048, the memory consumption of the original MPS even exceeds 23GB, while our improved MPS only consumes 6.9GB memory in this case. In real-world use cases, the memory consumption problem would be even more severe when processing larger networks. If the memory footprint exceeds the available memory capacity, we have to inevitably introduce the overhead and complexity of disk storage or distributed computing. Therefore, compared to the original MPS, our improved MPS significantly reduces memory consumption and enables the processing larger network in a single machine.

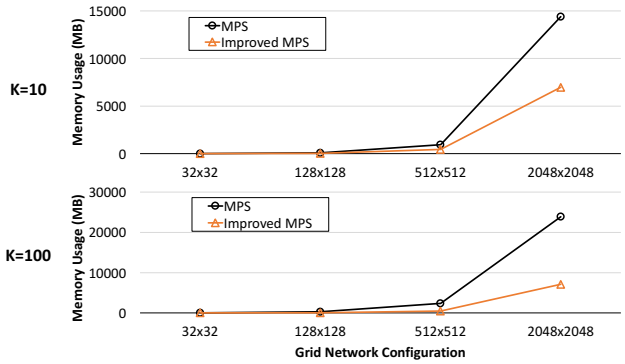

Figure 7: Memory consumption of finding top K shortest paths under different grid networks and K values.

4.3 Evaluation with real road network

In this section, we further extend our evaluation to a real-world road network in US provided by the 9th DIMACS implementation challenge for shortest path [1]. Among the 12 road networks in DIMACS, we select the Eastern USA road network (Road-E), which has 3,598,623 vertices and 8,778,114 edges. Because of the huge memory consumption of the original MPS, other larger networks in DIMACS exceed the memory capacity of our evaluation platform and therefore are infeasible for our evaluation.

Running time: In the evaluation of the Road-E dataset, we find top K shortest path with multiple randomly generated source and destination pairs and then measure the average execution time. In the experiments, we compare the original MPS and our improved MPS with K values varying from 100 to 600. As shown in Figure 8, with the real-world network, our improved MPS shows significantly performance improvement over the original MPS. For example, when K is 100, our improved MPS shows 3.5× speedup over the original MPS. With higher K values, both the original MPS and our proposed method require longer processing time. However, the speedup of our method also increases with the K value increment, and it reaches up to 4.1× when K is 600.

Memory: In addition, in the experiments of real-world datasets, we also evaluate the memory usage of both original MPS and our improved MPS method. As shown in Figure 9, with different K values, the memory usage remains relatively similar. In this case, the original MPS consumes close to 25 GB memory, while the improved MPS method consumes only 5 GB, which indicates that our improved MPS algorithm shows a 5× reduction compared to the original MPS algorithm in memory usage.

5 RELATED WORK

The KSP problem has been researched as a classical graph problem having various variants for decades [4, 7, 9, 17]. Two classes of KSP problems including the unconstrained KSP and the constrained KSP have been discussed. For the unconstrained KSP, no restriction is applied in the definition of the path in a graph. Using the classic Dijkstra's algorithm with improved priority queue data structures, the unconstrained KSP problem can be solved in $O(m + Kn \log n)$ time using the well-known approach in [8]. This algorithm is further

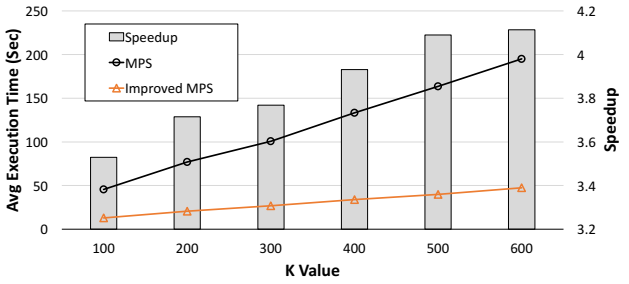

Figure 8: Average execution time of finding top K shortest paths under different K values for Road-E dataset.

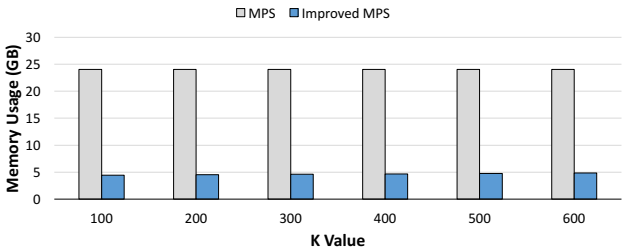

Figure 9: Memory usage of finding top K shortest paths under different K values for Road-E dataset.

improved by Eppstein et al. [6] and their approach maintains an asymptotic complexity of $O(m+n\log n+K)$ in both time and space in the worst case. Their algorithm computes an implicit representation of the paths, from which each path can be output in $O(n)$ extra time. By proposing a replacement paths algorithm, an improved implementation of Eppstein's algorithm is proposed in [9] with $O(n)$ improvement in time. Such a speedup is achieved through fast subroutine path replacement, which may fail but at a tiny probability.

The KSP problem becomes much more challenging when the K shortest paths require to be simple: in this case, any duplicate vertices should be eliminated in a path. This problem was originally examined by Hoffman [10], but most of the early attempts to solve it resulted in exponential time solutions [11, 17]. To solve this constrained KSP problem, Yen's algorithm was proposed in [17] by essentially executing $O(n)$ single-source shortest path computation for each output path. By using modern data structure, Yen's algorithm can be implemented in $O(Kn(n\log n+m))$ time and its computational upper bound increases linearly with value of K. This asymptotic worst-cased bound for enumerating K simple shortest simple paths in a directed graph has been predicted in theory [10]. Based on Yen's algorithm, several variants of KSP with heuristic improvements have been proposed [7, 9, 11]. By improving Yen's algorithm in the case of undirected graphs, Katoh et al. reduced the time complexity to $K(m+n\log n)$. To find shortest simple paths, MPS algorithm [4] overcomes the shortcomings of the original Yen's algorithm, and exhibits much faster speed than other top KSP algorithms [7, 9, 17]. By introducing the new concepts including reduced cost and deviation tree, the computational complexity of Yen's algorithm was improved from $O(Kn(n\log n+m))$ to $O(m\log n+Kn)$ by MPS algorithm in the worst case. In this paper, our proposed algorithm optimizes the MPS algorithm to enumerate shortest simple paths with optimized space efficiency. The key idea is to compress the deviation tree during the search process.

6 CONCLUSION

In this paper, we propose an improved MPS algorithm for ranking top K shortest simple paths in large graph/networks. Our algorithm overcomes the problem of excessive amount of memory consumption in the original MPS algorithm by two novel design strategies. One is to only add the shortest path to the pseudo-tree in each iteration through a tree-pruning scheme. The other is to adopt reversed order in the pseudo-tree such that the storing and searching shortest paths would be much more efficient. Extensive experiments show that our improved MPS algorithm achieves 3x~6x running-time speedup, and 2.5x~6.7x memory savings compared to the original MPS algorithm.

REFERENCES

[1] 2006. Ninth DIMACS Implementation Challenge-Shortest Paths. (2006). http://www.dis.uniroma1.it/challenge9/

[2] Ren Chen and Viktor K Prasanna. 2016. Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform. In *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 212–219.

[3] Ernesto De Queirós Vieira Martins, Marta Margarida Braz Pascoal, and José Luis Esteves Dos Santos. 1997. A new algorithm for ranking loopless paths. *Research Report, CISUC* (1997).

[4] Ernesto De Queirós Vieira Martins, Marta Margarida Braz Pascoal, and José Luis Esteves Dos Santos. 1999. Deviation algorithms for ranking shortest paths. *International Journal of Foundations of Computer Science* 10, 03 (1999), 247–261.

[5] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[6] David Eppstein. 1999. Finding the K Shortest Paths. *SIAM J. Comput.* 28, 2 (Feb. 1999).

[7] Gang Feng. 2014. Improving Space Efficiency With Path Length Prediction for Finding k Shortest Simple Paths. *IEEE Trans. Comput.* 63, 10 (2014), 2459–2472.

[8] BL Fox. 1975. K-th shortest paths and applications to probabilistic networks. In *Operations Research*, Vol. 23. B263–B263.

[9] John Hershberger, Matthew Maxel, and Subhash Suri. 2007. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms (TALG)* 3, 4 (2007), 45.

[10] Walter Hoffman and Richard Pavley. 1959. A Method for the Solution of the N th Best Path Problem. *Journal of the ACM (JACM)* 6, 4 (1959), 506–514.

[11] Naoki Katoh, Toshihide Ibaraki, and Hisashi Mine. 1982. An efficient algorithm for k shortest simple paths. *Networks* 12, 4 (1982), 411–427.

[12] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. Graphbig: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 69.

[13] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 89–100.

[14] Ajitesh Srivastava, Ren Chen, Viktor Prasanna, and Chelmis. 2015. A hybrid design for high performance large-scale sorting on FPGA. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6.

[15] Ilie Tanase, Yinglong Xia, Lifeng Nai, Yanbin Liu, Wei Tan, Jason Crawford, and Ching-Yung Lin. 2014. A highly efficient runtime and graph library for large scale graph analytics. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*. ACM, 1–6.

[16] Qingsong Wen, Qi Zhou, Chunming Zhao, and Xiaoli Ma. 2013. Fixed-point realization of lattice-reduction aided MIMO receivers with complex K-best algorithm. In *Proc. IEEE Int. Conf. Acoust., Speech and Signal Process. (ICASSP)*. Vancouver, Canada, 5031–5035.

[17] Jin Y Yen. 1971. Finding the k shortest loopless paths in a network. *management Science* 17, 11 (1971), 712–716.

View publication stats