www.combinatorics.ir

Transactions on Combinatorics
ISSN (print): 2251-8657, ISSN (on-line): 2251-8665
Vol. 5 No. 3 (2016), pp. 23-31.

www.ui.ac.ir

A NEW $O(m + kn \log \overline{d})$ ALGORITHM TO FIND THE k SHORTEST PATHS IN ACYCLIC DIGRAPHS

MEHDI KADIVAR

Communicated by Manouchehr Zaker

Abstract. We give an algorithm, called T*, for finding the k shortest simple paths connecting a certain pair of nodes, s and t, in a acyclic digraph. First the nodes of the graph are labeled according to the topological ordering. Then for node i an ordered list of simple $s - i$ paths is created. The length of the list is at most k and it is created by using tournament trees. We prove the correctness of T* and show that its worst-case complexity is $O(m + kn \log \overline{d})$ in which n is the number of nodes and m is the number of arcs and $\overline{d}$ is the mean degree of the graph. The algorithm has a space complexity of $O(kn)$ which entails an important improvement in space complexity. An experimental evaluation of T* is presented which confirms the advantage of our algorithm compared to the most efficient k shortest paths algorithms known so far.

1. Introduction

Let s and t be two fixed nodes of a directed network $G = (N, A)$ where N and A are the node and arc sets of G respectively and c_{ij} is the length of the arc $(i, j) \in A$. In the k-shortest path problem KSP we intend to determine k paths $p_1, \ldots, p_k$ in G, between two fixed nodes, the initial node s and the terminal node t, such that p_i have length greater or equal than p_{i-1} when $1 < i \leq k$, and the remainder paths between the fixed nodes should have length at least equal to p_k. Several important variations of the KSP problem have been studied in the literature. Usually every solution path is acceptable but in some works arc- or node-disjoint paths are considered. In other works an special value of k or loop less paths are considered. In this paper we consider an acyclic directed network an find k simple shortest paths i.e. no node can be repeated.

MSC(2010): Primary: 90C27; Secondary: 68Q25.

Keywords: k shortest path problem, complexity of algorithms, ranking of paths.

Received: 25 July 2015, Accepted: 20 January 2016.

The application range of the KSP problem is wide. For example it is usable in scheduling, sequence alignment, networking and many other application areas in which optimization problems need to be solved [5].

When the time complexity of algorithms are compared, implicitly it is assumed that enough main memories are accessible in the machine. So we must consider the space that is needed for execution of the algorithms. In this paper, we introduce a new algorithm to solve KSP which is not only faster than the best known algorithms but also it contains an important improvement in the space complexity. We show that for large scale problems, fast algorithms with unsuitable space complexity have low performance.

1.1. **Related works.** This problem has been studied very well and several algorithms have been presented to solve the problem. There are several kind of the KSP problem in which the solution paths should satisfy certain constraints. Usually every solution path is well come but in some works arc- or node-disjoint paths are considered. In other works, the value of k is a fixed number or loop less paths are considered [1, 5].

The problem was originally examined by Hoffman and Pavley [11], and then all attempts to solve it led to exponential time algorithms [16]. Several papers [2, 9, 14] consider the version of the k shortest paths problem in which repeated nodes are allowed, and it is this version that we also study. One can also make a restriction that the paths found be arc disjoint or node disjoint [15], or include capacities on the arcs [4].

For the unconstrained KSP problem, which allows a path to contain cycles, the theoretically fastest algorithm to date is Eppstein's algorithm [5] with a time complexity of $O(m + n\log n + k\log k)$. The constrained KSP problem, for which only simple paths are allowed, is much harder. In undirected graphs, the best algorithm attributed to Katoh et al. [13] has a complexity of $O(k(m + n\log n))$, and in directed graphs, the best result is an algorithm proposed by Gotthilf and Lewenstein [10], which has a complexity of $O(k(mn + n2loglogn))$.

Aljazzar et. al. present a heuristic directed search algorithm, called K*, to find the k shortest paths in directed weighted graphs which operates on-the-fly and it runs in $O(m + n\log n + k)$ time [1].

The algorithm presented in [6] has a complexity of $O(kn(m+n\log n))$ and follows the same process as Yen's deviation algorithm.

1.2. **Preliminaries.** Let $G = (V, A)$ be a flow network by a set $V = \{1, 2, 3, \ldots, n\}$ of n nodes and a set A of m directed arcs.

- *Node degree*: Let d_i^- and d_i^+ denote the in- and out-degree of a node i and $d_i = d_i^- + d_i^+$. Also the mean degree of the nodes is $\overline{d} = \frac{\sum_{i\in N} d_i}{n}$.

- *Adjacency List*: The out-node adjacency list $A^+(i)$ of a node i is the ordered list of nodes which are adjacent to that node, that is $A^+(i) = \{j \in N|(i,j) \in A\}$. Similarly the in-arc adjacency list $A^-(i)$ of i is the ordered list of nodes that i is adjacent to them, that is $A^-(i) = \{j \in N|(j,i) \in A\}$

1.3. **Eppstein's algorithm.** Now we introduce Eppstein's algorithm (EA) and a lazy variant of it [12] for acyclic digraphs, and we will compare our contribution with them in the next sections. For the unconstrained KSP problem, which allows a path to contain cycles, the theoretically fastest algorithm to date is the basic algorithm presented by Eppstein [5] with a time complexity of $O(m+n\log n+k)$, where n is the number of nodes and m is the number of edges. Some data structure implementation techniques can improve the algorithm to $O(m+n+k)$. However these techniques are too complicated from a practical point of view [5, 12]. Let $G = (V, A, s, t)$ be a acyclic digraph in which s and t are the source and sink nodes, respectively. EA contains the following three main steps in the algorithm:

- The first step is the Dijkstra's search on G in reverse in order to compute a shortest path tree rooted at t. This step requires $2m + n = O(m + n)$ runtime.
- The second step is the construction of the path graph $P(G)$ which takes $m+11n = O(m+n)$ using some complex data structure techniques described in [8].
- The third step is extracting the k shortest paths from the path graph $P(G)$. This step can be performed using Frederickson's search on $P(G)$ [7], which requires a runtime of $O(k)$.

The time and space required to build $P(G)$ is fairly large. To avoids this expensive step, a lazy variant of Eppstein's algorithm (LVEA) in which only the parts of $P(G)$ which are necessary for the selection of the k shortest paths are built [12]. However LVEA and EA have the same asymptotic worst-case complexity in terms of both time and space, it has a significant performance advantage over EA in practice [1, 12].

2. The T* algorithm

It is assumed that the nodes of input network are labeled based on topological ordering. We refer to $P[i] = \{P_1^i, P_2^i, \ldots, P_{k_i}^i\}$ as the set of k_i shortest paths from s to node i. We assume that $L(P_1^i) \leq L(P_2^i) \leq \cdots L(P_{k_i}^i)$ which $L(P)$ denotes the length of path P. Let $(i,j) \in A$, because of topological ordering and since $order(j)$ is larger than the other nodes in all $s-i$ paths, $P_l^i \cup \{(i,j)\}$ is a loop less path for each $l \in \{1,2,\ldots,k_i\}$. Therefore if we define $P[s] = \emptyset$ then $P[i] = \bigcup_{j\in A^-(i)} \bigcup_{P\in P[j]} P \cup \{(j,i)\}$. According to the above descriptions, we introduce the following algorithm. The main idea of the algorithm is easy to to describe, start with the source node s with $P[s] = \emptyset$ and then select a node i with the smallest label (with respect to the topological ordering) and compute $P[i]$. Repeat this process until $P[t]$ is computed. To compute $P[i]$ we construct a tournament tree [7] namely, first we select the smallest element from each $P[j]$ in which $j \in A^-(i)$ and we build a min-priority queue (using a min-heap) out of these elements. Then we repeat the following steps: we extract the minimum from the min-priority queue and this will be the next element in $P[i]$. From the original

sorted list where this element came from we remove the next smallest element (if it exists) and insert it to the min-priority queue. We are done when the queue becomes empty or $P[i]$ contains at most k elements.

Algorithm 1 T*: Topological ordering based k shortest path algorithm.

1: $next = order(s)$;
2: **while** $next < order(t)$ **do**
3: Update($P[next]$ by $P[j]$, $j \in A^{-}(next)$);
4: $next = next + 1$;
5: **end while**
6: −
7: Update($P[i]$ by $P[j]$, $j \in A^{-}(i)$)
8: Make min-priority queue $MQ(i)$ by $P_j^1 \in P[j]$ for $j \in A^{-}(i)$ according to $L(P_j^1) + c_{ji}$ values.
9: **while** $MQ(i) \neq \emptyset$ and $|P[i]| < k$ **do**
10: Move P at root of $MQ(i)$ to $P[i]$ and replace the moved item by its neighbor in the source list from which it came and update the nodes values travering up in $MQ(i)$ along the path of the moved item.
11: **end while**

3. Correctness and complexity of the T* algorithm

In this section we analytically study the properties of the T* algorithm. First we prove the correctness the algorithm and then its time and space complexity is computed.

Theorem 3.1. *$P[t]$ determines at most k shortest $s-t$ paths.*

Proof. Without loss of generality it is assumed that number i is the topological ordering label of node i. We use induction to show that $P[t]$ determines at most k shortest $s-t$ paths. Since the T* algorithm starts with the node s and it is assumed that networks are loop less, we have $P[s] = \emptyset$. When the algorithm computes $P[j]$ for a node j, all $P[i]$s are computed for each i where $i < j$ (see the lines 1-6 of the algorithm). Also, because of topological ordering, each directed path from s to j contains only the nodes with smaller order than the order of j. Therefore we claim that at the end of the iteration of T* for node j, $P[j]$ contains at most k shortest paths from s to the node j. According to the above elucidations, the set $\bigcup_{i \in A^{-}(j)} P[i]$ contains all possible paths that can connect s to j by adding an arc (i_0, j) to them. Since $MQ(j)$ is constructed by $\bigcup_{i \in A^{-}(j)} P[i]$ and $P[j]$ is updated by minimum elements of $MQ(j)$ (see line 9 of T*), therefore it contains the smallest possible path. Also the condition of the loop in the algorithm keeps the length of $P[j]$ less than k. □

Theorem 3.2. *The T* runs in $O(m + nk \log \overline{d})$ time in which $\overline{d}$ is the mean degree in digraph G. Also, The asymptotic space complexity of the algorithm is $O(kn)$.*

Proof. In the topological ordering algorithm for a node i, all in-arcs are examined in $|A^-(i)|$ times. Overall, the algorithm performs this operation $\sum_{i\in N}|A^-(i)| = m$ times.

In T* algorithm, to compute $P[i]$, first we select the smallest element from each $P[j]$ in which $j \in A^-(i)$ and we build a min-priority queue (using a min-heap) out of these elements in $O(|A^-(i)|) = O(d_i)$ time. Then we repeat the following steps: we extract the minimum from the min-priority queue (in $O(\log d_i)$ time) and this will be the next element in $P[i]$. From the original sorted list where this element came from we remove the next smallest element (if it exists) and insert it to the min-priority queue (in $O(\log d_i)$ time). We are done when the queue becomes empty or $P[i]$ contains at most k elements. Therefor $P[i]$ is computed in $O(d_i + k\log d_i)$ time. The total running time is $O(m) + O(\sum_{i\in N}(d_i + k\log d_i)) = O(m) + O(m) + O(k\sum_{i\in N}\log d_i)$. Since $\sqrt[n]{\prod_{i\in N} d_i} \le \frac{\sum_{i\in N} d_i}{n}$ and $\sum_{i\in N}\log d_i = \log\prod_{i\in N} d_i$, overall, the algorithm performs this operation in $O(m+kn\log\overline{d})$ in which $\overline{d}$ is the mean degree of the graph.

Space complexity: In each iteration for a node i, at most $2k$ unit of space is needed to save the list of i's neighbors $p[i]$ and the space consumed by the binary tree $MQ(i)$ is $O(\Delta)$. Whenever $p[i]$ have been computed, we can free the space used for the tree. Altogether, we get obtain a space complexity of $O(\Delta) + \sum_{i=1}^{n} 2k = O(kn)$ □

Although $O(m+n+k)$ is the best known worst-case time complexity, but for some values of k, the complexity of T* is better for DAGs. The time complexities of K* and T* contain terms $4m$ [1] and $2m$ (see the proof of Theorem 2.2), respectively. Therefore, we can consider $O(4m + 2n\log n + k)$ as the worst case complexity of K* and also $O(2m + kn\log\overline{d})$ as the worst case complexity of T*. The inequality $2m + kn\log\overline{d} < 4m + 2n\log n + k$ implies that for $k < (2m + 2n\log n)/(n\log\overline{d} - 1)$ the complexity of T* is better. If $m = O(n)$ then we have $\log\overline{d} = O(1)$ and the inequality is change to $k < 2(1 + \log n)$. Also, the exact times for EA is $3m + 12n + k$ [5]. Therefore T* outperforms EA when $k < (m + 12n)/(n\log\overline{d} - 1)$.

Example 3.3. *We now give a numerical example. Consider the network G shown in Figure 3.3. The arc lengths are written on each arc in the figure. Let nodes 1 and 6 are the source and destination nodes respectively and $k = 2$. Therefore $P[i]$ contains at most 2 paths for every node i.*

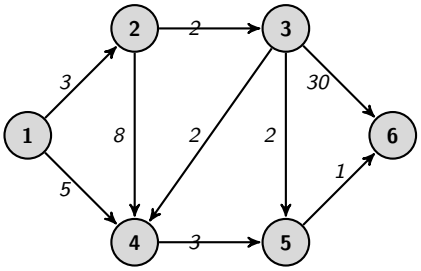

FIGURE 1. A directed Graph with arc length.

The results in Table 1 show $P[\cdot]$. First for node 1 as the source node we have $P[1] = \emptyset$ and then we have $P[2] = \{1{-}2\}$ and $P[3] = \{1{-}2{-}3\}$. Initially $P[4]$ is updated by $P[1]$ and we have $P[4] = \{1{-}4\}$

then $P[4]$ *is updated by* $P[2]$ *and we have* $P[4] = \{1-4, 1-2-4\}$ *and finally it is updated by* $P[3]$ *and science the path 1-2-4 has a larger length than path 1-2-3-4, it is replaced by 1-2-3-4 and we have* $P[4] = \{1-4, 1-2-3-4\}$. *Initially* $P[5]$ *is updated by* $P[3]$ *and* $P[5] = \{1-2-3-5\}$ *and next it is updated by* $P[4]$ *and we have* $P[5] = \{1-2-3-5, 1-4-5\}$. *similarly for 6 we have* $P[6] = \{1-2-3-5-6, 1-4-5-6\}$.

next	*All possible paths*	$P[next]$
1	$\emptyset$	$\emptyset$
2	*1-2*	*1-2*
3	*1-2-3*	*1-2-3*
4	*1-4, 1-2-4 , 1-2-3-4*	*1-4, 1-2-3-4*
5	*1-4-5, 1-2-3-5, 1-2-4-5, 1-2-3-4-5*	*1-4-5, 1-2-3-5*
6	*1-2-3-6, 1-4-5-6, 1-2-3-5-6, 1-2-4-5-6, 1-2-3-4-5-6*	*1-4-5-6, 1-2-3-5-6*

TABLE 1. T* process to find 2 shortest paths between the nodes 1 and 6.

4. **Experimental results**

In this section, our computational experiments are presented. We compare the cpu time of our algorithm versus EA, LVEA, the Yen (Yen's algorithm) [16] and Fng (the hybrid algorithm) algorithms [6] and K* [1]. We use the versions of EA, LVEA and K* which designed for DAGs. These algorithms are coded in C++ with the same style and functions and performed on 2.66 GHz Intel(R) Core (TM) i5 machines with 2GB RAM. Our test instances include random networks containing up to thousands of nodes and arcs. We use a network generator in our experiments:

- The MAFRANGEN generator: We develop the MAFRANGEN generator based on the design presented in [3] to generate random acyclic digarphs. It is a generator that produces random instances of network optimization problems. MAFRANGEN uses parameters $d = m/n(n-1)$ and m/n to specify the density of the network. To made a acyclic networks, we remove cycles from the produced instances.

Each CPU time value in the following tables and figures is the average values over at least 10 random instances for each combination of parameters.

The elapsed time for dense families of MAFRANGEN generator is shown in Table 2. In these instances $n = 4000$, $m = 9 * 10^6$ and $\overline{d} \approx 2000$. Our results show that T* is up to 4, 300, 260, 10 and 4 times faster than K^*, Yen, Fng, EA and LEVA, respectively. In average, these results are 2.8, 190.9, 171.6, 5 and 2.8 respectively.

k	T*	K*	Yen	Fng	EA	LEVA
2	0.31	1.53	27.4	23.7	3.2	1.2
5	0.31	1.51	30.2	26.2	3.2	1.8
10	0.35	1.6	46.21	41.31	3.2	1.2
20	0.41	1.7	58.54	58.6	3.2	1.36
30	0.6	1.7	69.5	71.16	3.2	1.5
40	0.76	1.7	93.52	113.4	3.3	1.5
50	0.84	1.73	104.5	131.8	3.3	1.57

TABLE 2. CPU times of T*, K*, Yen, Feng, EA and LEVA algorithms for relatively dense DGAs.

The result presented in Table 3 shows the cpu time in millisecond. In this case we put $n = 4000$ and $m = 100000$. The simulation time for K*, EA and LEVA are approximately invariant but this time for the others increase. This results show that the running time of K*, EA and LEVA have low dependency to the value of k.

k	T*	K*	Yen	Fng	EA	LEVA
2	4.1	6.2	17670.0	16066.1	7.2	7.1
5	5.0	7.9	44101.2	40164.5	7.2	7.0
10	6.5	7.33	88576.5	80123.1	7.2	7.0
15	8.1	7.23	130545.6	120411.2	7.2	7.0
50	19.0	8.7	440231.2	401089.1	7.3	7.01

TABLE 3. CPU times of T*, K*, Yen and Feng algorithms for dense DGAs.

Now we compare the algorithms with respect to the size of the problem. we use four similar computers to evaluate the algorithms. Auxiliary memories are used whenever overflow occurs. Table 4 shows the cpu time versus the size of the digraphs when $k = 50$. The results show that T* is computationally superior to EA, LVEA and K* algorithms. For $n = 10^6, m = 2 * 10^{11}$ and $n = 5 * 10^6, m = 5 * 10^{12}$ overflow occurs during the execution of EA and LEVA and K*. So we use auxiliary memory to save the heaps and $P(G)$ in theses algorithms which is time-consuming. For $n = 10^6, m = 2.5 * 10^{10}$ only T* solves the KSP problem and the other programs were not stopped after 600 minutes elapsed.

n	m	T*	K*	LEVA	EA
10^4	$2*10^9$	5 sec.	10 sec.	9.6 sec.	22.8 sec.
10^6	$2*10^{11}$	8.3 min.	279.6 min.	286.6 min.	Over flow
$5*10^6$	$5*10^{12}$	208.1 min.	$>$ 600 min.	$>$ 600 min.	Over flow

TABLE 4. CPU times of T*, K*, Yen and Feng algorithms VS the size of DAGs.

Above results show that for dense graphs, the run time of T* is smaller than the other algorithms. Also for sparse graphs and some values of k, T* is at least 2.8 times faster than the others.

5. **Conclusion**

A new algorithm T* is presented in this paper for finding k shortest simple paths in a acyclic weighted digraph. Its worst time complexity is $O(m + kn \log \overline{d})$ in which $\overline{d}$ is the mean degree of the graph. We show that for $k < (m + 12n)/(n \log \overline{d} - 1)$ our algorithm is faster that the best known algorithms. In practice, simulation results show that T* is at least 2.8 times faster than the K*, Yen's, Feng's, EA and LVEA algorithms. An other advantage of the presented algorithm is that it does not require the graph to be explicitly available and stored in main memory. In T*, at most kn unit of memory is required. This nice feature is very useful to solve moderate and large scale problems. When the time complexity of algorithms are compared, it is assumed that enough main memories are accessible in the machine. Some times this assumption is not correct. For example our experimental results show that because of a lack in space in RAM, we have to save $P(G)$ and the heaps, used in EA or K*, in our hard disk. Since access time to a hard disk is time consuming, the runtime of the algorithms have a significant increase.

Acknowledgments

The author wish to thank the referee for helpful comments. This work was supported in part by the research council of Shahrekord University.

References

[1] H. Aljazzar and S. Leue, K*: a heuristic search algorithm for finding the k shortest paths, *Artificial Intelligence*, **175** no. 18 (2011) 2129–2154.

[2] J. A. Azevedo, M. E. O. Santos Costa, J. J. E. R. Silvestre Madeira and E. Q. V. Martins, An algorithm for the ranking of shortest paths, *European J. Oper. Res.*, **69** (1993) 97–106.

[3] C. Caliskan, A computational study of the capacity scaling algorithm for the maximum flow problem, *Comput. Oper. Res.*, **39** no. 11 (2012) 2742–2747.

[4] Y. L. Chen, Finding thekquickest simple paths in a network, *Information Processing Letters*, **50** (1994) 89–92.

[5] D. Eppstein, Finding the k shortest paths, *SIAM J. Comput.*, **28** (1999) 652–673.

[6] G. Feng, Finding k Shortest Simple Paths in Directed Graphs: A Node Classification Algorithm, *Networks*, **64** (2014) 6–17.

[7] G. N. Frederickson, An optimal algorithm for selection in a min-heap, *Inform. and Comput.*, **104** (1993) 197–214.

[8] G. N. Frederickson, Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees, in: 32nd Annual Symposium on Foundations of Computer Science, *IEEE Comput. Soc. Press*, Los Alamitos, CA, 1991 632–641.

[9] B. L. Fox, k-th shortest paths and applications to the probabilistic networks, ORSA/TIMS Joint National Mtg., **23** (1975) p. B263.

[10] Z. Gotthilf and M. Lewenstein, Improved algorithms for the k simple shortest paths and the replacement paths problems, *Inform. Process. Lett.* , **109** (2009) 352–355.

[11] W. Hoffman and R. Pavley, A method for the solution of the N th best path problem, *J. Assoc. Comput. Mach.*, **6** (1959) 506–514.

[12] V. M. Jimenez and A. Marzal, A lazy *version of Eppstein's shortest paths algorithm*, in: 2nd International Workshop on Experimental and Efficient Algorithms, in: Lecture Notes in Computer Science, Springe, 2003 179–190.

[13] N. Katoh, T. Ibaraki and H. Mine, An efficient algorithm for k shortest simple paths, *Networks*, **12** (1982) 411–427.

[14] S. P. Miaou and S. M. Chin, Computing k-shortest path for nuclear spent fuel highway transportation, *Eur. J. Operational Research*, **53** (1991) 64–80.

[15] J. W. Suurballe, Disjoint paths in a network, *Networks*, **4** (1974) 125–145.

[16] J. Y. Yen, Finding the K shortest loopless paths in a network, *Management Sci.*, **17** (1971) 712–716.

Mehdi Kadivar

Department of Mathematical Sciences, University of Shahrekord, P.O.Box 115, Shahrekord, Iran

Email: m_kadivar@aut.ac.ir