A Simple Survey on Top K Paths Algorithms on FST

Xixuan Feng
University of Wisconsin-Madison
Madison, WI 53706, USA
xfeng@cs.wisc.edu

ABSTRACT

This article provides a survey on top k paths algorithms, especially focusing on how to apply these algorithms to finite-state transducers (FST). I would compare 3 algorithms that relate to the tree of paths, and 2 of them are implemented.

1. PROBLEM DEFINITION

Our task is to enumerate top k shortest paths of a given FST as fast as possible. Before we actually look at the algorithms that find top k paths on a given graph, we should be aware of that an FST is a directed acyclic graph (DAG) with extra input and output labels attached to each edge, and it also has multiple edges between 2 nodes. Given this, we should not pay too much attention to algorithms that do more work to avoid loops in the output paths, which are presumably slower than the ones that do not consider loops at all. Therefore, in this article, our problem comes down to enumerate top k paths in a given DAG (allowing multiple edges between 2 nodes) efficiently.

2. ALGORITHMS

The introduction section of [4] is a rather comprehensive description of some top-k-path algorithms.

Besides the theoretical results of algorithms, I also discuss the details when implementing them and the time-space tradeoff if some indexes are built.

2.1 Terminology

For a given FST G, let n be the number of states(nodes) in G, d be the maximum number of out degree of any nodes in G, and m be the number of edges in G. We have $m = O(nd)$. s and t are source and sink nodes of G, respectively. Let k denote the k in the kth-shortest path problem, and A_k be the kth-shortest path. The 3 following algorithms are discussed in presence of the shortest path tree T from all the nodes to the sink node, which can be computed using Dijkstra's algorithm with time complexity $O(m + n \log n)$.

2.2 Yen's Algorithm without Guaranteed Loopless

2.2.1 Theoretical Analysis

Yen in [5] purposes an algorithm to enumerate k shortest loopless paths. The time complexity is $O(kn^4)$.

[2] revises Yen's algorithm that it does not consider loopless, reducing the time complexity to be $O(km)$.

2.2.2 Implementation on FST

The simplest desciption of this implementation is: if A_k is available, find all the possible candidates of A_{k+1}, making sure that A_{k+1} would not be the same as $A_j, j = 1, 2 \cdots k$ and put them in a heap, pop out the shortest one to be A_{k+1}. There are 2 steps to accomplish this: 1) find n_1 so that any nodes before the n_1th node in A_k won't be used to generate candidates; 2) let $n_2 = n - n_1$, and iterate all edges from each nodes from n_1th to t and generate one candidate for each node, and then put them into the heap.

At the beginning, I did not build the tree structure T_k to represent all $A_j, j = 1, 2 \cdots k$, so it takes $O(kn_1 \log d))$ to do step 1; otherwise only $O(1)$ needed, with $O(n)$ time to maintain T_k and construct A_k.

Before using T_k, step 2 takes $O(n_2 d(n + \log(n_2 k)))$. It can be optimized as $O(n_2 d \log(n_2 k))$ if T_k is maintained. As for the $\log(n_2 k)$ term, a fibonacci heap can be used to get rid of it, making the step 2 complexity as $O(n_2 d)$ and the total complexity as $O(n) + O(n_2 d) = O(nd) = O(m)$ for computing A_{k+1}, same as the theoretical result.

2.3 Using Reduced Cost

2.3.1 Theoretical Analysis

[2] makes use of the structure of reduced cost in [3] to slight improve the above Yen's algorithm without guaranteed loopless. They call this MPS algorithm. The reduced cost structure can be used to sort edge in time $O(m \log m)$. Having this, the time complexity of enumeration is $O(kn)$.

2.3.2 Implementation on FST

Once T is computed, we can sort edges grouped by their tails using reduced cost. Time complexity is $O(nd \log d)$.

During the enumeration process, it is very similar to Yen's algorithm above, having 2 steps. And the first step is exactly the same. Time complexity is $O(n)$.

Step 2 is a lot faster after the sorting. $O(n \log(nk)$ with heap or balanced search tree, and $O(n)$ with fibonacci heap.

2.3.3 Do Some Work Offline

In this algorithm, there are two things we need to do before actually enumerating paths. One is to build shortest path tree T, and the other is to sort the edges using reduced cost.

If no projection is considered, both of the above can be done offline. T takes $O(n(d+\log n))$ time and $O(n)$ space. Sorting takes $O(nd \log d)$ time and no space.

If we have to consider projection, I would guess sorting is not doable offline. Let's discuss an example. In Figure 1, edge

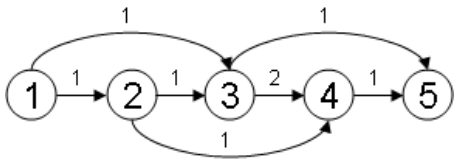

Figure 1: FST

$(1, 3)$ is ordered before $(1, 2)$ because it generates a shorter path from node 1 to node 5. But if we perform a projection between node 1 and node 4, the order should be changed. In Figure 2, edge $(1, 2)$ should be ordered before edge $(1, 3)$,

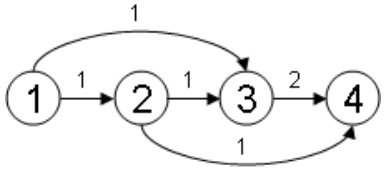

Figure 2: FST after Projection

which means we cannot do the sorting offline if we don't know which portion of the FST projections may take.

Because a shortest path tree is rooted by the sink of the DAG, projections that choose a different source but the same sink should be easily handled by simply removing those nodes. We have to store more information to support projections that choose a different sink as long as our FSTs have any fan-outs in the shortest path tree. For instance, the shortest path tree of Figure 1 should contain no path from node 3 to node 4. Once we do the projection and the FST becomes Figure 2, we need more information to build the new shortest path tree. However, I notice that all the paths in one shortest path tree are shortest paths between those two nodes. Therefore, if our FSTs does not have too many fan-outs, it is possible to store information in $O(n)$ space for constructing shortest path trees for any projections.

2.4 Implicit Path Representation

2.4.1 Theoretical Analysis

On computing top k paths, Eppstein [3] presents the asymptotically fastest algorithm. After using $O(m + n)$ to build some data structre representing all paths, we can get the implicit representation of the kth path in time $O(\log k)$. Of course, if we need to construct the kth path, we need $O(n)$ time, which is the same as the one uses reduced cost to improve Yen's algorithm.

3. EVALUATION

3.1 OpenFST

The ShortestPath operation in OpenFST [1] takes minutes to enumerate top 100 paths, while all the implementations discussed above run less than 100 milliseconds.

3.2 MPS v.s. YEN

I have implemented both Yen's algorithm without considering loopless and MPS algorithm. The following figures give time comparisons of two implementations on a FST with about 100 states.

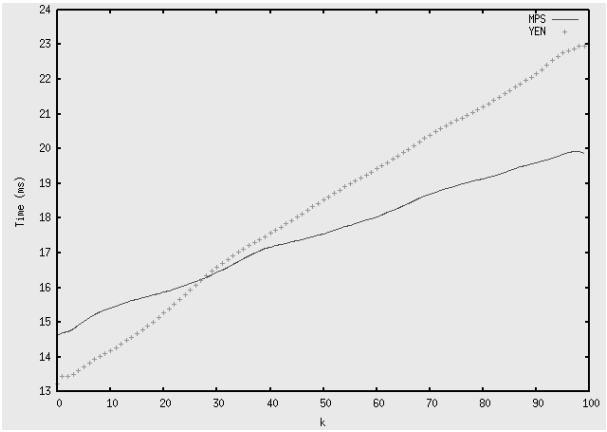

Figure 3: MPS v.s. YEN

Figure 3 shows MPS takes more time to build up the data structure as we analyzed, and it runs faster to enumerate each path. Although I have not used fibonacci heap, the running time curves look like linear because the heap insertion in step 2 is with a small constant factor compared to step 1.

4. CONCLUSION

I would suggest we just take the YEN's algorithm if we do less than tens of enumerations on each FST. If we figure out how to store the shortest path trees offline, top 10 paths can be enumerated within 1 millisecond.

Otherwise, MPS offers a faster total running time on enumerating more than 100 paths. Also, we may consider to guess the k in advance, and then pick the better one.

5. REFERENCES

[1] Openfst library.

[2] E. de QueirÃşs Vieira Martins, E. Queir, J. L. E. dos Santos, V. Martins, M. Margarida, M. M. B. Pascoal, J. Luis, and E. Santos. The k shortest paths problem, 1998.

[3] D. Eppstein. Finding the k shortest paths. *SIAM J. Comput*, 28(2):652–673, 1998.

[4] V. M. Jiménez and A. Marzal. Computing the K shortest paths: A new algorithm and an experimental comparison. In J. S. Vitter and C. D. Zaroliagis, editors, *Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 1999.

[5] J. Y. Yen. Finding the K shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.