Finding a Simple Path with Multiple Must-include Nodes

Hars Vardhan*, Shreejith Billenahalli*, Wanjun Huang*, Miguel Razo*, Arularasi Sivasankaran*, Limin Tang*, Paolo Monti†, Marco Tacca* and Andrea Fumagalli*
*The University of Texas at Dallas, TX, USA
{hxv071000, sxb071100, wxh063000, mrazora, axs075200, lxt064000, mtacca, andreaf}@utdallas.edu
†The Royal Institute of Technology, Kista, Sweden
pmonti@kth.se

***Abstract*—This paper presents an algorithm to find a simple path in the given network with multiple must-include nodes in the path. The problem of finding a path with must-include node(s) can be easily found in some special cases. However, in general, including multiple nodes in the simple path has been shown to be *NP-Complete*. This problem may arise in network areas such as forcing the route to go through particular nodes, which have wavelength converter (optical), have monitoring provision (telecom), have gateway functions (in OSPF) or are base stations (in MANET). In this paper, a heuristic algorithm is described that follows *divide and conquer* approach, by dividing the problem in two subproblems. It is shown that the algorithm does not grow exponentially in this application and initial re-ordering of the given sequence of must-include nodes can improve the result. The experimental results demonstrate that the algorithm successfully computes near optimal path in reasonable time.**

I. Introduction

Network standards [1] [2] allow loose definition of routing by requiring one or more nodes to be in the route of Link State Packet. This problem may arise in various networking areas such as optical networks, OSPF (Open Shortest Path First) protocol, telecommunication networks, and MANET (Mobile ad-hoc networks).

The problem of including node(s) in the path is polynomial time solvable only in some scenarios described in Section II. The shortest path with multiple must-include nodes can be seen as a more general case of the well known *traveling salesman path* (TSP) problem, which is *NP-complete*. Instead of traveling all nodes as in the original TSP, this problem requires to travel only a subset of the nodes from source to destination. Akin to the TSP problem, the problem of finding shortest path with multiple must-include nodes is shown to be *NP-Complete* [3].

In this paper, a heuristic algorithm is proposed to compute a simple path which contains a given ordered set of *must-include* nodes, i.e., set I. The algorithm's primary objective is to find at least one simple path, which satisfies the must-include constraint. However, the nature of the algorithm itself leads to finding near shortest path solutions. The algorithm comprises two main steps: **(1)** considering each pair of consecutive nodes in I to represent a segment of the entire end-to-end path, and then computing candidate paths for the segment; **(2)** concatenate segments' candidate paths, one from each segment, in order to make a simple path from source to destination. The *max-flow* [4] approach is used to find candidate paths, which yields maximum number of edge-disjoint paths for individual segments. The time complexity of step (1) is $O(k|V||E|^2)$, where $|V|$ and $|E|$ are the number of nodes and edges in the network and k is the size of set I. Step (2) uses backtracking algorithm for combining segment paths. The worse case complexity of the backtracking algorithm is $O(\lambda^k)$, where λ is the maximum degree of the node in the network. However, in practice, the run time of this second step is affected by the number of pairs of candidate paths across segments, which are disjoint. The number of such disjoint path pairs decreases with increasing value of $\frac{k}{\lambda}$, as explained in Section II. Consequently, the run time of step (2) stays reasonably low even at large values of k and it has minimal effect on the algorithm's run time.

As intuition suggests, the order of the nodes in set I may significantly affect the algorithm outcome. Two cases are considered. In one case, the node order in I is randomly given, and cannot be changed. In the other case, the nodes in I can be re-ordered prior to running the algorithm. Experimental results presented in the paper indicate that the proposed algorithm is able to find a simple path with *must-include* nodes and requires reasonable run time.

II. Algorithm Description

Given a directed graph $G = (V, E)$ and a set $I \subset V$, the objective is to find a simple path $\mathcal{P}$ from source $s \in V$ to destination $t \in V$. Further, the set of must-include nodes I may be given as an ordered set. Let $k = |I|$ and $u_i \in I$, where $i = 1, 2, 3..., k$, be the nodes in I. Also, let $\pi(x)$ denote the index of node x in $\mathcal{P}$, then $\mathcal{P}$ must have the following properties:

$$\forall u_i \in I \implies u_i \in \mathcal{P}; \quad i = 1, 2, 3...., k \tag{1}$$

$$\forall u_i, u_j \in I \wedge i < j \implies \pi(u_i) < \pi(u_j) \tag{2}$$

Simple solutions to this problem can be found in two special scenarios. If $k = 1$ and the given graph is *undirected*, the problem can be solved by applying *max-flow* algorithm once. And, if the constraint of simple path is removed, then the problem is reduced to computing shortest path between every pair of nodes (or *segment*) along $\mathcal{P}$. However, in general, i.e., for directed graph, when $k \geq 2$ and the path must be simple, the problem is *NP-Complete*. A straightforward solution to the problem can be obtained by directly using *K-shortest path* algorithm [5] results in shortest (optimal) path, but the required value of K may be large in most cases, thus making the solution impractical.

Algorithm 1 *incNodePaths(s, t, I, allPaths)*

```
1:  $j \leftarrow 0$, $seg_j \leftarrow (s, I_j)$
2:  for $j = 1$ to $j < I.size$ do
3:      $seg_j \leftarrow (I_{j-1}, I_j)$
4:  end for
5:  $seg_j \leftarrow (I_j, t)$
6:  $\forall e : e \in E$, capacity$(e) \leftarrow 1$
7:  for $j = 0$ to $j <= I.size$ do
8:      $v_1 \leftarrow seg_j.first$, $v_2 \leftarrow seg_j.second$
9:      for all $u_i \in I :\ u_i \neq v_1 \ \wedge u_i \neq v_2$ do
10:         $\forall e$ : $e$ passing through node $u_i$; capacity$(e) \leftarrow 0$
11:     end for
12:     compute flows $F = \bigcup f(i,j)\ \forall\ u_i, u_j \in V$ in order
        to maximize the flow between $v_1$ and $v_2$
13:     $m \leftarrow 0$
14:     while $\exists i, j : f(i,j) > 0$ do
15:         Trace path $P_{jm}$ from $v_1$ to $v_2$
16:         $m \leftarrow m + 1$
17:     end while
18: end for
19: loop
20:     $\mathcal{P} \leftarrow \phi$
21:     $ret \leftarrow$ combinePaths(0, $P$, $\mathcal{P}$)
22:     if $ret = -1$ then
23:         Break
24:     end if
25:     allPaths $\leftarrow \mathcal{P}$, $P \leftarrow P - \mathcal{P}$
26: end loop
```

First, the problem is addressed with both of the constraints given in “Eqn. (1)” and “Eqn. (2)”. The algorithm follows *divide and conquer* approach. Firstly, it computes multiple candidate paths for each segment. Then, combining paths, one from each segment such that they do not form a loop, gives the solution. The algorithm is formally described in Algorithm 1 and Algorithm 2.

Let $I = \{u_1, u_2, \ldots.., u_k\}$ and segments seg_i is defined as follows: $seg_0 = (s, u_1)$, $seg_1 = (u_1, u_2)$, ..., $seg_k = (u_k, t)$. Conversely, we can say $s = seg_0.first$, $u_1 = seg_0.second = seg_1.first$, ..., $u_k = seg_k.first = seg_{k-1}.second$. Procedure *incNodePaths(s, t, I, allPaths)* first computes all edge-disjoint paths P_i for seg_i then, procedure *combinePaths(m, P, $\mathcal{P}$)*, described in Algorithm 2, is called to compute a simple path by combining paths, one from each of the segments.

Steps $7-18$ of Algorithm 1 use *max-flow* approach to compute all possible edge-disjoint paths for each of the segments. So, the time complexity of the steps $7-18$ of Algorithm 1 is $O(k|V||E|^2)$. Algorithm 2 is a *backtracking*

Algorithm 2 *combinePaths(m, P, $\mathcal{P}$)*

```
1:  if $m > k$ then
2:      return 1
3:  end if
4:  $ret \leftarrow -1$
5:  for all path $p \in P_m$ do
6:      if $\exists\ p :\ p \cup \mathcal{P}$ not making a loop then
7:          $\mathcal{P}.push\_back(p)$
8:          $ret \leftarrow$ combinePaths($m+1$, $P$, $\mathcal{P}$)
9:          if $ret = -1$ then
10:             $\mathcal{P}.pop\_back()$   // {Undone last step}
11:         end if
12:     end if
13: end for
14: return $ret$
```

algorithm that iterates through all possible cases until it finds a solution which turns into k multiplications of the number of edge-disjoint paths in each of the segments. Here, the number of paths for each segment is bounded by λ. Hence the worse case complexity of Algorithm 2 is $O(\lambda^k)$. However, careful inspection reveals that complexity is in practice a polynomial function of k, as explained next. The number of candidate paths in each of the segments is an increasing function of λ. In the presence of a large number of segments (large value of k), the candidate paths in each of the segments are more likely to (be incompatible) contain some node that is also contained in paths of other segments. Algorithm 2 is designed to efficiently prune further iterations which may lead to loop. On the other hand, if the number of candidate paths per segment, which are not making loop with others, is high then the backtracking algorithm terminates early as it finds with ease a simple path from s to t. So, effectively the run time of Algorithm 2 does not grow exponentially with k.

The *max-flow* computation uses shortest path for augmenting paths [6]. Also, we sort and index all paths in P according to the *hop-count*. Hence, nature of the algorithm results in near-shortest possible path.

Further, if the constraint given in “Eqn. (2)” can be relaxed, the outcome of the Algorithm 1 can be improved by re-ordering the nodes of I. The re-ordering of I can be done using simple algorithm such as *depth first traversal* and the reordered set I' of must-include nodes can be used instead of I in Algorithm 1.

III. Experimental Results

Simulation is carried out on several instances of input parameters to verify the effectiveness of the proposed algorithm. The input parameters are the topology layout and the list of traffic requests, each defined by both (s,t) pair and set I. Network topologies are generated randomly keeping the number of nodes at $n = 50$ and varying λ. s and t of each request are chosen randomly. Similarly, k must-include nodes are chosen randomly in the topology, excluding both s and t as possible choices. For each tuple (n, λ, k), 100 requests are created and their average results are presented. The performance parameters computed in the experiments are: 1) N_{succ} - Number of times at least one simple path is found, 2) T_{exp} - Total experiment run time. The suffix (W) and (R) denotes the result of the experiment without reordering of I and with reordering of I, respectively. All of the experiments are conducted on the same hardware-software platform.

First, parameter N_{succ} is analyzed with respect to λ as well as with respect to k. The study of parameter N_{succ} is shown in Figs. 1-2. As λ increases, N_{succ} increases if k

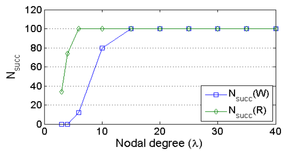

Fig. 1. N_{succ} vs λ: $k = 20$

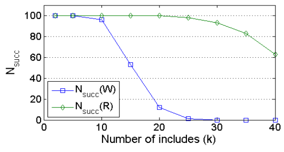

Fig. 2. N_{succ} vs k: $\lambda = 6$

remains constant (Fig. 1). N_{succ} decreases with increasing values of k (Fig. 2). These plots support the earlier claim that the availability of more disjoint candidate paths for every segment favors the success rate of the algorithm. Indeed, if we increase the value of λ, *max-flow* algorithm tends to find more disjoint candidate paths for each segment, hence, results tend to improve. Also, increasing the ratio $\frac{k}{\lambda}$ (by increasing k), candidate paths computed for different segments are more likely to share some nodes in set $V - I$. Consequently, it becomes increasingly difficult for the algorithm to find a loop-less path from s to t.

In order to study parameter T_{exp}, two additional parameters are defined: 1) T_{flow} - Total computation time by steps $7 - 18$ of Algorithm 1, 2) T_{comb} - Total computation time by Algorithm 2, where $T_{exp} = T_{flow} + T_{comb}$. The study of parameter T_{exp} is shown in Figs. 4 - 5. As expected, T_{exp} increases polynomially with λ (Fig. 5), keeping k constant, as the computation time of the candidate paths for every segment is $O(V|E|^2)$, which is proportional to λ^2. Parameters T_{flow} and T_{comb} versus k are plotted in Fig. 3. T_{flow} is a function of $k(n-k)$, which is shown by experiment in Fig. 3. Considering T_{comb}, when k becomes relatively high (e.g., $> \frac{n}{2}$), the vast majority of candidate paths computed for the segments are not disjoint, thus allowing Algorithm 2 to quickly determine that a solution may not exist and terminate swiftly. Also, T_{comb} is relatively very low as compared to T_{flow} as shown in Fig. 3. The combined result for $T_{exp}(W)$ as well as for $T_{exp}(R)$ is shown in Fig. 4. The increasing vertical distance between $T_{exp}(W)$ and $T_{exp}(R)$ in Fig. 4 shows that the re-ordering of I results in more number of candidate paths and hence, higher value of $T_{flow}(R)$. These results support the earlier claim that Algorithm 1 does not grow exponentially in practice.

Overall, the reordering of set I is shown to improve the outcome of N_{succ} (Figs. 1-2). The *depth first traversal* approach is used to re-order the nodes in I which is applied k times. The computation time of this step is directly proportional to k. As shown in Fig. 5 the reordering step has a limited impact on T_{exp}.

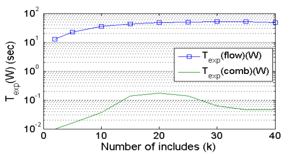

Fig. 3. $T_{flow}(W)$ vs k: $\lambda = 6$

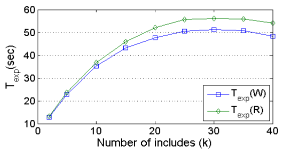

Fig. 4. T_{exp} vs k:$\lambda = 6$

Last, T_{exp} required by *KSP approach* is compared with that of *flow approach* described in Algorithm 1, when allowing reordering of set I. For comparison, the *K-shortest path* algorithm is used to exhaustively find the shortest path from s to t, which contains all nodes of I. This is computationally costly, and applicable only to small size networks, e.g., $n = 25$, $\lambda = 4$, $k = 4$. In Fig. 6, T_{exp} is plotted for each of the LSP requests. It can be seen that T_{exp} for *flow approach* varies within small range as oppose to T_{exp} for *KSP approach*. In most cases, *KSP-approach* needs much longer time than what is required by Algorithm 1.

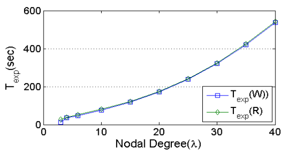

Fig. 5. T_{exp} vs nodal degree (λ): $n = 50$, $k = 20$

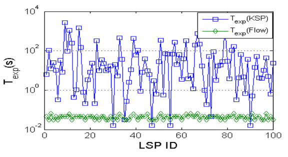

Fig. 6. T_{exp}(Flow) vs T_{exp}(KSP) $n = 25,\ \lambda = 4,\ k = 4$

IV. Conclusion

The requirement of including multiple nodes in the computation of end-to-end routing paths may find many applications in today's networks, e.g., optical, Ethernet, and mobile networks. The problem of including node(s) in the path is easily solvable in some cases. However, in general, including 2 or more nodes can be shown to be *NP-complete*. In this paper, a heuristic algorithm is presented to compute a simple path with multiple must-include nodes. The heuristic algorithm follows the *divide and conquer* approach, by dividing the problem into two subproblems. The experimental results show that our algorithm computes near-shortest path containing all of the include nodes in reasonable time.

References

[1] D. Awduche, L. Berger, D. Gan, V. S. T. Li, and G. Swallow, "RFC 3209 - RSVP-TE: Extensions to RSVP for LSP Tunnels," December 2001.

[2] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus, "RFC 2702 - Requirements for Traffic Engineering Over MPLS," September 1999.

[3] "Finding a simple path with multiple must-include nodes," The University of Texas at Dallas, TX, USA, Tech. Rep., 2009. [Online]. Available: http://opnear.utdallas.edu/publications/reports/UTD-EE-2-2009.pdf

[4] J. Edmonds and R. M. Karp, *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems*, New York, NY, USA, 1972.

[5] E. de Queirs Vieira Martins and M. M. B. Pascoal, "A new implementation of yen's ranking loopless paths algorithm." *4OR*, vol. 1, no. 2, pp. 121–133, 2003.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.