Finding a Simple Path with Multiple Must-include Nodes

Technical Report UTD/EE/2/2009

June 2009

Hars Vardhan*, Shreejith Billenahalli*, Wanjun Huang*, Miguel Razo*, Arularasi Sivasankaran*, Limin Tang*, Paolo Monti†, Marco Tacca*, and Andrea Fumagalli*
*Open Networking Advance Research (OpNeAR) Lab
Erik Jonsson School of Engineering and Computer Science
The University of Texas at Dallas, Richardson, TX, USA
andreaf@utdallas.edu
†Next Generation Optical Network (NeGONet) Group
School of Information and Communication Technology, ICT-FMI
The Royal Institute of Technology, Kista, Sweden
pmonti@kth.se

Abstract

This document presents an algorithm to find a simple path in the given network with multiple must-include nodes. The problem of finding a simple path with only one must-include node can be solved in polynomial time using *lower bound max-flow* approach. However, including multiple nodes in the path has been shown to be a *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). Also, network standards allow loose definition of routing by requiring one or more nodes to be in the routing of Link State Packet. In this document, a heuristic algorithm is described to find a simple path between a pair of terminals, which has constraint to pass through a certain set of other nodes.

The algorithm is comprised into two main steps: (1) considering a pair of nodes in sequence from source to destination as a segment and then computing candidate paths between each segment, and (2) combining paths, one from each segment, in order to make simple path from source to destination. The *max-flow* approach is used to find candidate paths, a which provides maximum number of edge disjoint paths for individual segments. The second step of the algorithm uses backtracking algorithm for combining paths. The time complexity of the first step of the algorithm is $O(k|V||E|^2)$, where k is the number of must-include nodes. The time complexity of step (2) depends upon total number of candidate paths which are not touching any one of the candidates of other segments. So, the worse case time complexity of step (2) is $O(\lambda^k)$, where λ is the maximum nodal degree of the network. However, we show that step (2) has minimal effect on the algorithm and it does not grow exponentially with k in this application. Later, we also show that initial re-ordering of the given sequence of must-include nodes can improve the result. The experimental results show that the algorithm is successful in computing near optimal path in reasonable time.
keywords: constrained path computation, graph theory, heuristic algorithm, max flow, network route.

I. INTRODUCTION

Network standards [1] [2] [3] 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). For example, the optical network routing may require the route to include some specific nodes, which are wavelength converter or amplifier/regenerator enabled. OSPF network may have some of the nodes to act as gateways across the subnetwork, which must be included in the path when source and destination are multiple subnet apart. In telecommunication network, routing of the traffic may be forced to go through specific nodes that have traffic monitoring capability. MANET often require routes through some designated nodes, which are directly connected to a base station.

The problem of including only *one node* in the computation of a simple path is polynomial time solvable using, for example *lower bound max-flow* [4], as briefly described next. The lower bound max-flow algorithm computes a flow, which must include all the edges that have a positive (> 0) lower bound value. If only one edge has a positive lower bound, the flow computed by the algorithm can be used to build a simple path, which includes such edge. The "edge" instance of the problem can be transformed into its "node" equivalent by first splitting the include-node into two half-nodes, i.e., n_1 and n_2. All incoming edges to the original node converge to n_1. All outgoing edges of the original node emerge from n_2. A directed edge e_l from n_1 to n_2 is added with a positive lower bound value. All other (original) edges of the network have zero lower bound. Now, applying the *lower bound max-flow* [5] algorithm to this flow network, a feasible flow from source to destination can be found such that e_l is included in at least one of the (flow) augmenting paths. Then on merging n_1 and n_2 the simple path can be traced. The lower bound max-flow technique cannot be successfully applied to 2 or more include nodes, as there is not a guarantee that the same augmenting path in the flow algorithm contains all of them at once. Hence, it is not possible to trace out a simple path with all of the must-include nodes.

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*[1]. 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. This problem can be solved by *brute-force* method of complete enumeration of all the possible paths and then choosing the best path which contains all must-include nodes. Akin to the TSP problem, the number of enumerations in the problem is exceedingly large [8]. Given a network of $n(= |V|)$ nodes, the number of possible such tours is exponential, or more precisely $O(n!)$. On the other hand, if a path is given from source to destination then it can be verified in polynomial time whether it includes all the nodes of I. Given a path P, presence of any loop in P can be checked

[1]The TSP problem and its variations have been addressed by a number of papers in the literature, e.g., [6] [7].

in $O(|V|)$ time. Simultaneously, it can also be verified that P has all must-include nodes in it. Hence, this problem belongs to the class of *NP-complete* problems.

In this document, 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 routing 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 from source to destination, and then computing candidate paths for each segment; (2) Concatenate segment paths, one from each segment, in order to make a simple path from source to destination. We use the *max-flow* [9] approach to find candidate paths for each segment, which yields the maximum number of edge-disjoint paths for each individual segment. The time complexity of this first step of the algorithm 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. The second step of the algorithm uses a backtracking algorithm for combining segment paths and may even find more than one end-to-end edge-disjoint paths, which contain all of the nodes in I. 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 not disjoint[2] . Though the backtracking algorithm is exponential in terms of k, the number of such disjoint path pairs decreases with increasing value of $\frac{k}{n}$. Consequently, step (2) has minimal effect on the algorithm's run time and it stays reasonably low even at large values of k.

As intuition suggests, the order of the nodes in set I may significantly affect the algorithm outcome. Two cases are considered in this paper. 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 based on *depth first traversal* before running the proposed algorithm. Experimental results presented in the document indicate that the proposed algorithm is able to find a simple path with *must-include* nodes and requires reasonable run time in most cases. In addition, we quantify the performance gain obtainable when set I can be re-ordered based on *depth first traversal*.

II. Algorithm Description

In this section, the formal definition of the routing problem constrained to set I — the set of must-include nodes — is given, followed by the detailed description of the algorithm proposed to solve the problem. Finally, a simple algorithm is described, which favorably re-orders the set of nodes in I to improve the final outcome.

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 $t \in V$. The nodes in set I must be present in $\mathcal{P}$. Let $k = |I|$ be the number of nodes in set I. Let

[2]Two paths are disjoint to each other if together, they are not making any loop

$u_i \in I$, where $i = 1, 2, 3...k$, be the nodes in I. Then, $\mathcal{P}$ must have the following property:

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

Simple solutions to this problem can be found in two special cases. If $k = 1$, the problem can be solved using *lower bound max-flow* algorithm [4]. If the constraint of finding simple path is removed, then the problem is reduced to computing shortest path between every pair of nodes (or *segment*) $(s, u_1), (u_1, u_2) \ldots, (u_k, t)$ along $\mathcal{P}$. The concatenation of the segment shortest paths forms an end-to-end path which may not be simple, i.e., it may contain loops. However, in general, i.e., when $k \geq 2$ and the path must be simple, the problem is NP-Complete. In order to prove this, the proof of NP-completeness of TSP problem can be extended which is shown in Section I.

One can obtain a straightforward solution to the problem by directly using *K-shortest path* algorithm [10]. The K-shortest path algorithm can be called repeatedly by gradually increasing the value of K starting from $K = k$, till "equation (1)" is satisfied. Though this approach results in shortest (optimal) path, the required value of K may be large in most cases, thus making the solution impractical. (In Section III the *K-shortest path* approach is applied to a small network and a small size of I to offer an optimum result to compare our heuristic against in terms of hop count.)

Further, the set of must-include nodes I may be given as an ordered set. In this case, $\mathcal{P}$ must have the following additional property. Let $\pi(x)$ denote the index of node x in $\mathcal{P}$, then

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

Adding constraint given by "equation (2)" does not change the hardness of the problem.

We first address the problem with both of the constraints given in "equation (1)" and "equation (2)". The algorithm follows *divide and conquer* approach. Firstly, it computes multiple candidate paths for each individual segment. Then, combining paths, one from each segments such that they do not form a loop, gives the solution. Let $I = \{u_1, u_2, u_3.....u_k\}$, then we define segments seg_i, $i = 0, 1, 2..k$ as follows: $seg_0 = (s, u_1)$, $seg_1 = (u_1, u_2)$, $seg_2 = (u_2, u_3)$ .... $seg_k = (u_k, t)$. Conversely, we can say $s = seg_0.first$, $u_1 = seg_0.second = seg_1.first$, ... $u_k = seg_k.first = seg_{k-1}.second$. So, given k must-include nodes, there are $k + 1$ segments. Also, P_i is defined as a collection of all edge-disjoint candidate paths for seg_i. P_{ij} represents j^{th} path (starting from index 0) of P_i. The algorithm uses two procedures: *findDisjointPaths(seg_i)* and *combinePaths(m, P, $\mathcal{P}$)*. The procedure *findDisjointPaths(seg_i)* computes all edge-disjoint paths P_i for seg_i. Edge-disjoint paths for each segment are computed by setting the capacity of all edges in the graph $G(V, E)$ to 1. Then, maximum flow from v_1 to v_2 is computed using *max-flow* algorithm, where $v_1 = seg_i.first$, $v_2 = seg_i.second$. Now, by using depth-first search

(DFS) of nodes from v_1 to v_2 along only those edges which has flow > 0, all disjoint paths of P_i for seg_i can be traced one by one. The procedure *findDisjointPaths(seg_l)* is formally described in Algorithm 1 which returns set of paths P_i for seg_i. P_i is sorted according to ascending order of the length of the path.

Algorithm 1 : *findDisjointPaths(seg_l)*

1: $v_1 \leftarrow seg_l.first$
2: $v_2 \leftarrow seg_l.second$
3: **for all** $u_i \in I: \ u_i \neq v_1 \ \wedge u_i \neq v_2$ **do**
4: $\quad \forall e$: edge e passing through node u_i; capacity$(e) = 0$
5: **end for**
6: 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
7: $m \leftarrow 0$
8: **while** $\exists i, j : f(i,j) > 0$ **do**
9: $\quad u_x \leftarrow v_1$
10: $\quad P_{lm} \leftarrow \{u_x\}$ // $\{P_{lm}$ is a vector of nodes$\}$
11: $\quad$ **repeat**
12: $\quad\quad$ **if** $\exists u_y : f(x,y) > 0$ **then**
13: $\quad\quad\quad P_{lm} \leftarrow P_{lm} + \{u_y\}$ // {push-back an element}
14: $\quad\quad\quad u_x \leftarrow u_y$
15: $\quad\quad\quad f(x,y) \leftarrow f(x,y) - 1$
16: $\quad\quad$ **end if**
17: $\quad$ **until** $u_x = v_2$
18: $\quad m \leftarrow m + 1$
19: **end while**
20: **return** P_l

This procedure is called for all segments $(seg_i; i = 0, 1, 2...k)$ in order to compute candidate paths P. Since *max-flow* algorithm is used for path computation, all paths within P_i are edge-disjoint. The step 3-5 of the Algorithm 1 ensures that the paths for seg_i do not touch any other include-node that is not belonging to seg_i. A candidate path which traverses some include-node other than nodes of its corresponding segment will never lead to a simple path. By removing those nodes (temporarily) each time before running *max-flow* algorithm, such possibilities can be avoided. Additional effect of temporarily removing nodes results in a network graph of $n - k$ nodes. So, the *max-flow* algorithm runs on the network of only $n - k$ nodes instead of n. Step 6 of the Algorithm 1 computes maximum flow between the segment, and step 8-19 trace all edge disjoint paths for the segment.

The another procedure *combinePaths(m, P, $\mathcal{P}$)* uses output P of the Algorithm 1 in order to compute desired path from s to t. The procedure *combinePaths(m, P, $\mathcal{P}$)* computes a simple path by combining paths, one from each of the segments. The procedure is defined formally in Algorithm 2. The resultant path $\mathcal{P}$ is a path (sequence of nodes) in the order it is added.

Initially, $\mathcal{P}$ is empty. Also, the information about current segment is passed as first parameter of the procedure. This is a recursive procedure which should be initially invoked like *combinePaths(0,P,ϕ)*. The procedure combines paths

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

from each of the segments to make a simple path. Step 1-3 ensures the successful termination of the Algorithm 2. Step 7-8 selects and concatenate a possible sub-path (not making loop with previously selected sub-paths) from current segment and proceed toward next segment. In steps 9-10, previous concatenation of a sub-path is undone and the *for-loop* (step 5) looks for another possible sub-path for the current segment. On success, Algorithm 2 returns 1, and -1 otherwise.

It is possible that there are more than one possible paths from s to t, all of them are satisfying the constraints given in "equation(1)" and "equation(2)". On removing all paths from candidate set, which are successfully combined in the previous steps, process of combining paths can be repeated until no further simple path is found. For computing more than one such edge-disjoint paths, the procedure *combinePaths(m, P, $\mathcal{P}$)* should be called repeatedly after removing all sub-paths (P_{ij}) used in previous rounds from P.

Algorithm 1 is using *max-flow* approach to compute all possible edge-disjoint paths for each of the segments. The *max-flow* algorithm is called for each segment. So, the total time complexity of the Algorithm 1 is $O(k|V||E|^2)$, where $k = |I|$ is the number of include nodes in the given network graph $G(V, E)$. The *max-flow* algorithm used in our implementation is given by Edmonds-Karp [9] which has the time complexity of $O(|V||E|^2)$. However, our algorithm is independent of a particular algorithm used to compute the maximum flow between a pair of terminals.

The next procedure described in Algorithm 2 is a *backtracking* algorithm that iterates through all possible cases until it finds a solution which turns into k multiplications of number of edge-disjoint paths in each of the segments. Here, number of paths for each segment is bounded by the maximum nodal degree(λ) of the graph G. Hence the worse case complexity of Algorithm 2 is $O(\lambda^k)$.

The procedure in Algorithm 2 can be shown to be a *k-clique* problem. In order to show that, we will create a

TABLE I
INDEPENDENT PATH LIST

path	***Disjoint paths of other segments***
p_0	$p_4,\ p_5,\ p_7,\ p_8,\ p_9,\ p_{10}$
p_1	$p_4,\ p_6,\ p_7,\ p_9,\ p_{10},\ p_{11}$
p_2	$p_3,\ p_5,\ p_6,\ p_8,\ p_9,\ p_{10}$
p_3	$p_2,\ p_8,\ p_{10},\ p_{11}$
p_4	$p_0, p_1,\ p_7,\ p_8,\ p_9,\ p_{11}$
p_5	$p_0,\ p_2,\ p_7,\ p_8,\ p_9,\ p_{11}$
p_6	$p_1,\ p_2\ p_7,\ p_9,\ p_{10},\ p_{11}$
p_7	$p_0,\ p_1,\ p_4,\ p_5,\ p_6,\ p_{10},\ p_{11}$
p_8	$p_0,\ p_2,\ p_3,\ p_4,\ p_5,\ p_9,\ p_{10}$
p_9	$p_0,\ p_1,\ p_2,\ p_4,\ p_5,\ p_6,\ p_8$
p_{10}	$p_0,\ p_1,\ p_2,\ p_3,\ p_6,\ p_7,\ p_8$
p_{11}	$p_1,\ p_3,\ p_4,\ p_5,\ p_6,\ p_7$

graph G' using P, and will demonstrate that combining paths in G is equivalent to find a *k-clique* in the created graph G'. Let $G' = (V', E')$ is created where each vertex $v' \in V'$ corresponds to a path P_{ij} in P. Also, two vertices u' and v' are connected if and only if corresponding paths in P -

(a) belong to different segments.

(b) together, they are not making a loop.

To illustrate this construction, consider one example where P is computed for the given network and segments are $seg_0,\ seg_1,\ seg_2,\ seg_3$. Suppose, P is computed such that: P_0 contains three paths $\{p_0, p_1, p_2\}$ for seg_0, P_1 contains four paths $\{p_3, p_4, p_5, p_6\}$ for seg_1, P_2 contains two paths $\{p_7, p_8\}$ for seg_2 and P_3 contains three paths $\{p_9, p_{10}, p_{11}\}$ for seg_3. Since, there is a vertex in graph G' corresponding to each path p_x, so $V' = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$.

Now, the list of paths p_j which are not making a loop with path $p_i,\ i \neq j$ and belong to different segments are given in the TABLE I. With the help of this TABLE I, a graph $G' = (V', E')$ can be constructed which is shown in Fig. 1. The alike nodes, belong to the same segment. In the Fig. 1, node {0,1,2} belong to seg_0, {3,4,5,6} belong to seg_1, {7,8} belong to seg_2 and {9,10,11} belong to seg_3 segment. No two nodes of same type are connected with an edge. A node i is connected to node j with an edge iff corresponding row p_i of the TABLE I contains p_j in its right column. The construction of the graph G' ensures that no two nodes in V' are connected which represent paths belonging to the same segment in G.

Now, suppose if we have a *k-clique* in the graph G', then it must be having following properties:

- The size of clique k implies that the combined path, represented by these nodes, has k nodes.
- None of the two nodes in the clique represent the path are belonging to the same segment. So, all must-include nodes are included in the clique.

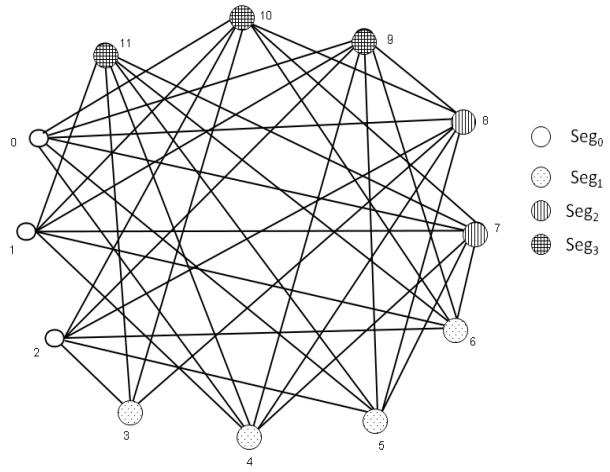

Fig. 1. The Graph G' formed by using P

- Since each node in G' is connected to $k-1$ nodes, so each of the corresponding paths all together can make a desired simple path.

These properties are required and sufficient to combine paths one from each segment to make desired loop less path. Hence, the problem of combining path is equivalent to the problem of finding *k-clique* in the induced graph. Now, we can easily combine all paths represented by nodes in the clique found in order to make a simple path from source to destination.

The worse case complexity of combining paths depends upon two parameters: λ-maximum nodal degree and k-number of must-include nodes. However, careful inspection reveals that it depends on the number of candidate paths in each of the segments. The number of candidate paths for each segment depends upon the ratio $\frac{\lambda}{k}$. Here, the upper bound on number of paths is λ. More the value of ratio $\frac{\lambda}{k}$ is, more number of independent candidate paths can be found. The reason is, if more number of segments are created, the *max-flow* will try to maximize the number (up to λ) of disjoint paths, and so paths belonging to two different segments may be forced to traverse one or more common nodes. The backtracking algorithm used in Algorithm 2 has capability to prune further iteration which may lead a path containing two such nodes in G'. On the other hand, if the number of candidate paths in segments, which are not making loop with others, is very high then the backtracking algorithm terminates early as it has found the simple path from source to destination. So, effectively the running time of the Algorithm 2 does not grow exponentially on an average.

Though, the algorithm does not guarantee that the path obtained is the shortest (optimal), the nature of the

algorithm results in near shortest possible paths which satisfy the constraint of including given set of nodes. The *max-flow* computation uses shortest path for computing augmenting paths [11]. Also, we sort the paths in P according to the *hop-count*. Further, the Algorithm 2 traverses nodes of G', formed by using P, in increasing order of index of nodes. Hence, the near-shortest possible path will be found first. If the Algorithm 2 is called second time after removal of all nodes in G' which belong to previous found paths, the next possible path between s and t will not be shorter than the previous ones. In order to support this claim, we compared our result for smaller networks with the results obtained by running *K-shortest path* algorithm repeatedly with incrementing the value of K until desired path is obtained. Since the value of K required by *K-shortest path* algorithm is very large, it is computationally not possible to get the optimal results for larger networks.

The complete algorithm is formally described in Algorithm 3. The algorithm first make *segments* from set I and computes candidate paths using Algorithm 1. The last step of the procedure calls Algorithm 2 repeatedly till all possible edge-disjoint paths are obtained.

Algorithm 3 *incNodePaths(s, t, I,allPaths)*

Require: $s \neq t$

1: $j \leftarrow 0$
2: **for** $j = 0$ *to* $j < I.size()$ **do**
3: **if** $j = 0$ **then**
4: $seg_j \leftarrow (s, I_j)$
5: **else**
6: $seg_j \leftarrow (I_{j-1}, I_j)$
7: **end if**
8: **end for**
9: $seg_j \leftarrow (I_j, t)$
10: **for** $j = 0$ *to* $j <= I.size()$ **do**
11: $P_j \leftarrow$ *findDisjointPaths(*seg_j*)*
12: **end for**
13: **loop**
14: $\mathcal{P} \leftarrow \phi$
15: $ret \leftarrow$ *combinePaths(*0, P,$\mathcal{P}$*)*
16: **if** $ret = 1$ **then**
17: *allPaths* $\leftarrow \mathcal{P}$
18: $P \leftarrow P - \mathcal{P}$
19: **else**
20: Break
21: **end if**
22: **end loop**

Further, if the ordering of the must-include given in "equation (2)" can be relaxed, the result of the Algorithm 3 can be improved by re-ordering the nodes in the set I. The re-ordering of nodes in I does not change any part of the Algorithm 3 or any of the procedures defined in Algorithm 1 and Algorithm 2. It only possibly change the

relative ordering of the include nodes in the desired path. This pre-computation step can be done before calling the Algorithm 3.

The re-ordering of include nodes can be done using simple algorithm such as *depth first traversal*. Let the set I' is the outcome of this process. Starting *depth first traversal* from s, suppose the node that is traversed first among I is x_1, then it should be placed first in say I'. Now, restarting *depth first traversal* from x_1, let the node x_2 is traversed first among $I - I'$ then x_2 should be placed next in I'. Repeating the process until $I - I' = \phi$ results in I' which contains all element of I but may be in different order. The new set I' of must-include nodes can be used instead of I in Algorithm 3.

The motivation of Algorithm 1 is to find maximum number of candidates for each of the segments such that each of the candidates is sharing the least number (0) of intermediate nodes with other candidates. And, the idea behind re-arranging the order of must-include nodes using *depth first traversal* is to avoid making a segment with nodes that are far apart. If the two nodes of the segment is nearer, the number of hops in the candidate paths for the segment is lesser. Hence, the possibility of interfering of a candidate path with other candidates is decreased and path combining process in Algorithm 2 becomes more successful.

The effect of re-ordering the nodes in I is illustrated with a simple example. Let, for a given source s and destination t in network graph G, the set of include nodes $I = \{I_1, I_2\}$. In Fig. 2, s, t, I_1 and I_2 are shown in bigger circle along with other nodes in smaller circle. Now, segments are formed by using set I, which are

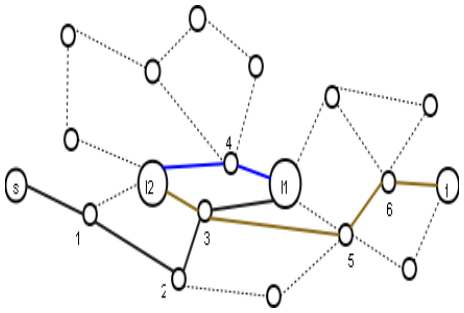

Fig. 2. Candidate paths without re-ordering I in G.

$seg_0 = (s, I_1)$, $seg_1 = (I_1, I_2)$ and $seg_2 = (I_2, t)$. The computation of candidate paths for each segment results into following:

$P_{00} = [s, 1, 2, 3, I_1]$

$P_{10} = [I_1, 4, I_2]$

$P_{20} = [I_2, 3, 5, 6, t]$.

It can be seen that $node$ 3 is forced to be used in P_{00} and P_{20} which are candidate paths of seg_0 and seg_2 correspondingly and therefore, they can not make a simple path together. On the other hand, if we consider re-

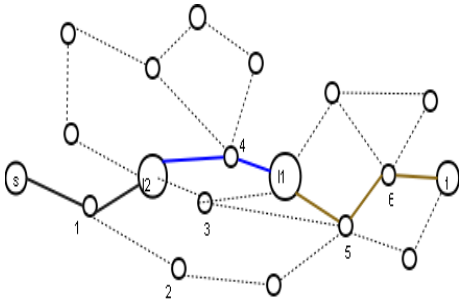

Fig. 3. Candidate paths after re-ordering I.

arranged elements of set $I = \{I_2, I_1\}$, then candidate paths shown in Fig. 3 would be as following:

$P_{00} = [s, 1, I_2]$

$P_{10} = [I_2, 4, I_1]$

$P_{20} = [I_1, 5, 6, t]$.

In this case, paths P_{00}, P_{10} and P_{20} can be combined in order to find a simple path from s to t, which increases the overall performance of the proposed algorithm.

Experimental studies are done for both of the above scenarios individually and results are discussed in the next section.

III. Experimental Results

Simulation experiments are 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 source-destination pair and set I (the set of must-include nodes). Network topologies are generated randomly keeping the number of nodes at $n = 50$ [3] and varying the average nodal degree (λ). The nodes are placed within a defined square area. An edge between two nodes is added with a probability that is inversely proportional to the geographical (euclidean) distance between the two nodes[4]. In practice, the distance between any two edge-connected nodes is bounded from above by some value. The source and destination of each request (s and t) are chosen randomly in the topology. 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:

- N_{succ}: The number of times the experiment finds at least one simple path.

[3]For scalability test, we have run additional experiments for larger networks up to $n = 1000$ and up to $\lambda = 100$ with similar results.

[4]There are many physical properties of network devices and equipments that may limit the maximum link length, e.g., loss of signal strength, signal distortion.

- D_{avg} : The average number of edge disjoint paths found per request.
- T_{exp} : Total experiment run time.

N_{succ} represents the number of instances out of 100 for which the algorithm is able to find at least one path from s to t that satisfies the must-include node constraint given in "equation (1)". D_{avg} represents the average number of edge-disjoint paths that are found per request, accounting for all 100 experiments. T_{exp} represents the average execution time of the experiment for all 100 requests. T_{exp} is the sum of the run time required by both Algorithm 1 and Algorithm 2. The suffix (W) denotes the result of the experiment without the reordering of I. The suffix (R) denotes the result of the experiment with the reordering of I. All of the experiments are conducted on the same hardware and software platform. The complete result is shown in the table II. In the following paragraphs, each of the parameters are discussed one by one.

First, parameter N_{succ} is analyzed with respect to the average nodal degree (λ) as well as with respect to the number of include nodes (k). The study of parameter N_{succ} is shown in Figs. 4 and 5. As λ increases, N_{succ} increases

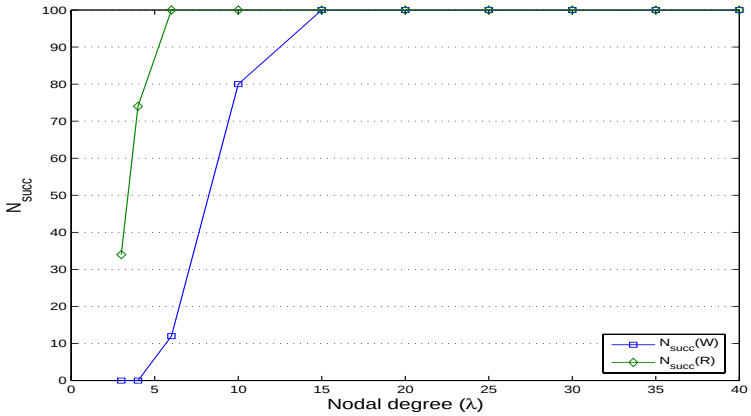

Fig. 4. N_{succ} vs nodal degree (λ): $n = 50$, $k = 20$

if k remains constant (Figure 4). N_{succ} decreases with increasing values of k (Figure 5). 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, keeping all other parameters constant, *max-flow* algorithm tends to find more disjoint candidate paths for each segment. Hence, results tend to improve with increasing values of λ. Also, increasing the ratio $\frac{k}{n}$ (by increasing k in this document), candidate paths computed for different segments are more likely to share some nodes in set $N - I$. Consequently, it becomes increasingly difficult for the algorithm to find a loop-less path from s to t. Dependency of D_{avg} on λ (k) is shown in Figs. 6 and 7. Parameter D_{avg} follows the same trend of parameter N_{succ}. In Figure 6 the gap between (W) and (R) decreases for increasing values of nodal degree, as the relative large number of include nodes ($k = 20$) limits the probability of finding additional simple paths even when reordering set I .

TABLE II
EXPERIMENT RESULTS

N	λ	k	$N_{succ}(W)$	$D_{avg}(W)$	$T_{exp}(W)$	$N_{succ}(R)$	$D_{avg}(R)$	$T_{exp}(R)$
50	3	20	0	0	13	34	0.34	15
50	4	20	0	0	18	74	0.74	21
50	6	20	12	0.12	23	100	1	29
50	10	20	80	0.8	37	100	1.15	43
50	15	20	100	1	58	100	1.78	65
50	20	20	100	1.51	87	100	1.97	95
50	25	20	100	1.83	119	100	2.13	136
50	30	20	100	1.96	160	100	2.23	194
50	35	20	100	2.12	211	100	2.35	264
50	40	20	100	2.2	272	100	2.36	312
50	6	2	100	2.71	6	100	3.12	7
50	6	5	100	1.74	12	100	2.13	14
50	6	10	96	1.03	17	100	1.43	20
50	6	15	53	0.53	21	100	1.02	26
50	6	20	12	0.12	23	100	1	29
50	6	25	1	0.01	25	98	0.98	32
50	6	30	0	0	26	93	0.93	31
50	6	35	0	0	24	83	0.83	30
50	6	40	0	0	23	63	0.63	29
250	6	10	100	1.63	79	100	2.28	79
250	10	10	100	2.7	117	100	3.78	130
250	15	10	100	3.87	162	100	5.91	180
250	25	10	100	5.31	269	100	8.34	302
250	6	25	94	1	92	100	1.43	93
250	10	25	100	1.32	249	100	2.13	281
250	15	25	100	2	347	100	3.04	376
250	25	25	100	3.22	572	100	4.56	621
250	50	25	100	5.74	1377	100	7.09	1402
250	50	50	100	3	2333	100	3.88	2410
250	100	50	100	4	7603	100	4.73	8012
500	10	10	100	3.02	233	100	3.83	431
500	10	25	100	1.87	519	100	2.78	827
500	15	10	100	4.66	313	100	5.34	461
500	15	25	100	2.81	705	100	3.89	898
500	25	100	100	1	3502	100	2	3787
500	25	10	100	7.25	484	100	7.76	682
500	25	25	100	4.39	1100	100	4.87	1289
500	25	50	100	2.53	2044	100	3.23	2296
500	50	100	100	2.16	8661	100	3.1	8899
500	50	200	100	1	12273	100	2	13225
500	50	25	100	7.56	2500	100	8.12	2760
500	50	50	100	4.53	4659	100	5.34	4950
500	6	10	100	1.9	163	100	2	310
500	6	25	99	1	368	100	1.68	588
500	75	150	100	2	20476	100	3	20913
500	75	300	85	1	25708	100	1.74	34749
500	75	50	100	6.11	8773	100	6.58	8474

TABLE III
EXPERIMENT RESULTS FOR LARGE NETWORKS

N	λ	k	$N_{succ}(W)$	$D_{avg}(W)$	$T_{exp}(W)$	$N_{succ}(R)$	$D_{avg}(R)$	$T_{exp}(R)$
1000	5	10	100	1.89	257	100	2	670.953
1000	10	20	100	2.94	963	100	3.2	1321.67
1000	25	50	100	3.92	4851	100	4.5	5231.58
1000	50	50	100	6.85	8645	100	7.2	10221.4
1000	100	100	100	6.14	70032	100	6.8	60232.7

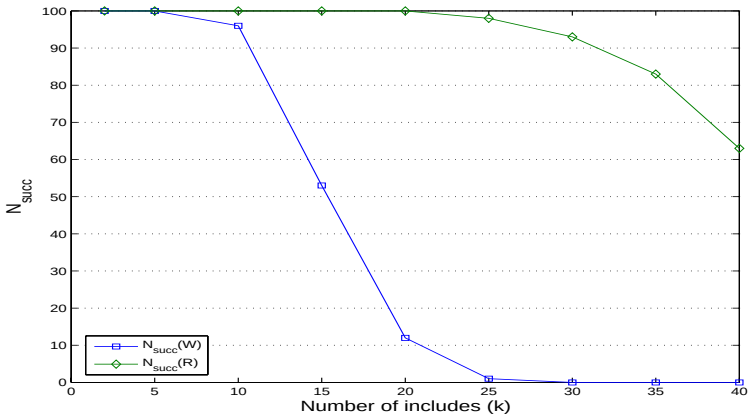

Fig. 5. N_{succ} vs *number of include* nodes (k): $n = 50$, $\lambda = 6$

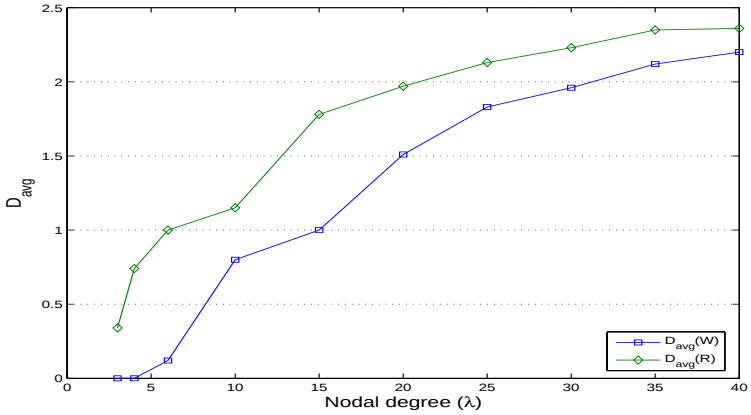

Fig. 6. D_{avg} vs nodal degree (λ): $n = 50$, $k = 20$

In Figure 7, on the other hand, a larger number of nodes in I severely limits the ability of the algorithm to find even one simple path, unless I is reordered.

The study of parameter T_{exp} is shown in Figs. 8 - 13. As expected, T_{exp} increases polynomially in λ (Figure 8)

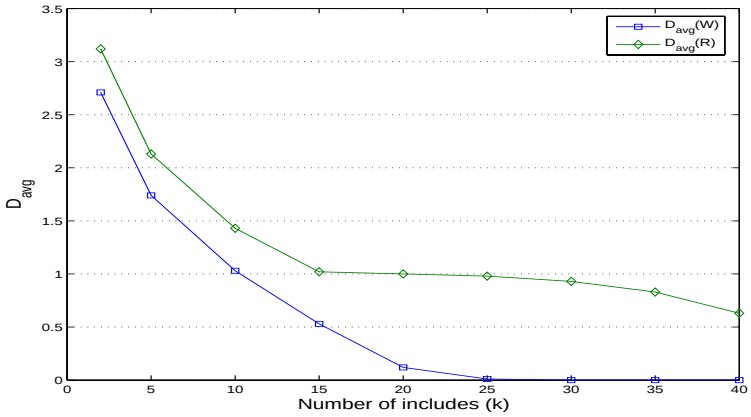

Fig. 7. D_{avg} vs *number of include* nodes (k): $n = 50$, $\lambda = 6$

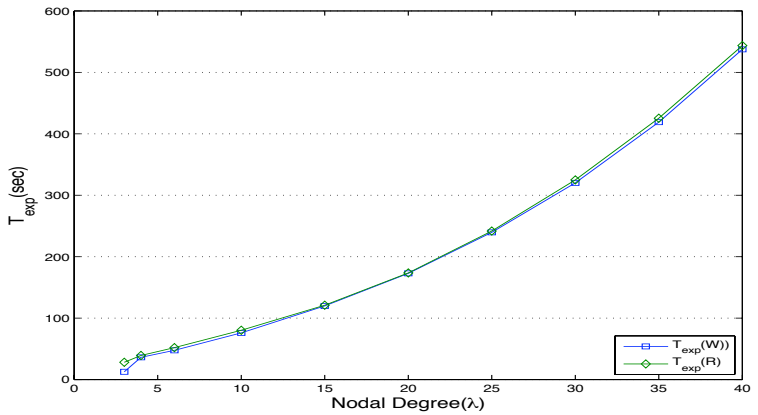

Fig. 8. T_{exp} vs nodal degree (λ): $n = 50$, $k = 20$

as the computation time of the candidate paths for every segment is $O((V)|E|^2)$, which is proportional to λ^2. In order to study parameter T_{exp} versus k, two additional parameters are defined –

- T_{flow} : Total computation time by Algorithm 1
- T_{comb} : Total computation time by Algorithm 2,

where $T_{exp}(W)$ is the sum of T_{flow} and T_{comb}. Parameters T_{flow} and T_{comb} are plotted in Fig. 9 and Fig. 10 respectively. Given a network of n nodes, Algorithm 1 runs the *max-flow* algorithm on the graph of only $n - k$ nodes. So, T_{flow} is a function of $k(n-k)$, which is shown by experiment in Fig. 9. The increasing vertical distance between $T_{exp}(W)$ and $T_{exp}(R)$ in Fig. 9 shows that the re-ordering I results in more number of candidate paths and hence, higher value of $T_{flow}(R)$. Further, when k becomes relatively high (e.g., $> \frac{n}{2}$, shown in Fig. 10), 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. 11 and Fig. 12. The combined result of Figs. 9 - 12 is shown in Fig. 13. These results support the earlier claim that in this problem, Algorithm 3 does not grow exponentially in practice.

Overall, the reordering of the must-include nodes (set I) is shown to improve the results of N_{succ} and D_{avg}. The *depth first traversal* approach is used to re-order the nodes in I. As *depth first search* (DFS) algorithm is applied k times, the computation time of this step is directly proportional to k. As shown in Fig. 8 the reordering step has a limited impact on T_{exp}.

Last, the outcome of Algorithm 3 is analyzed in terms of *number of hops* in the computed simple path, when allowing reordering of set I. For comparison, the *K-shortest path* algorithm is used to exhaustively find the shortest path from source to destination, which contains all nodes in I. The latter is a brute force technique that is computationally costly, and applicable only to small size problems, e.g., $n = 25$, $\lambda = 4$, $k = 4$. The complete

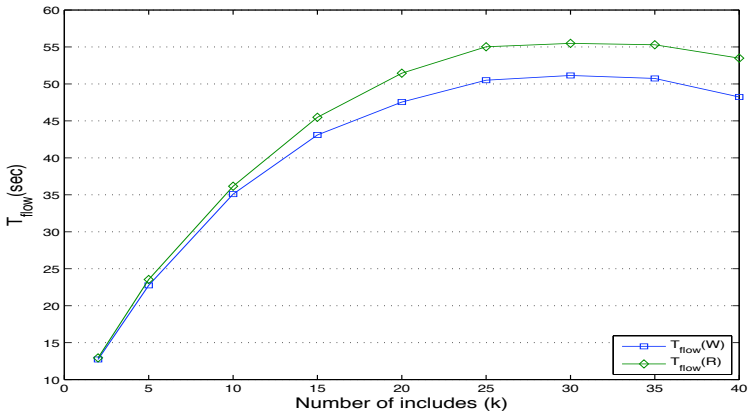

Fig. 9. T_{flow} vs *number of include* nodes (k): $n = 50$, $\lambda = 6$

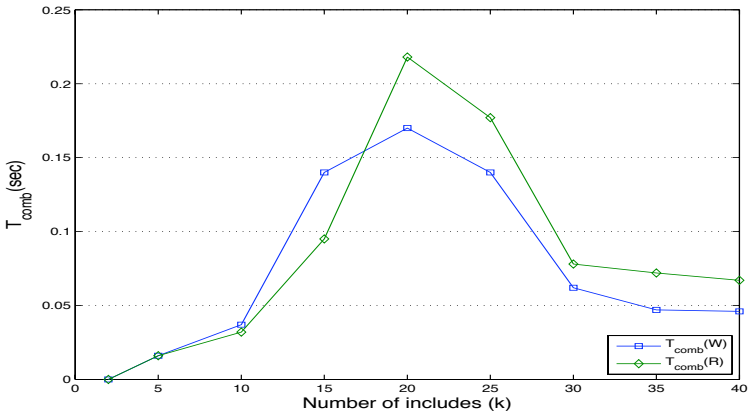

Fig. 10. T_{comb} vs *number of include* nodes (k): $n = 50$, $\lambda = 6$

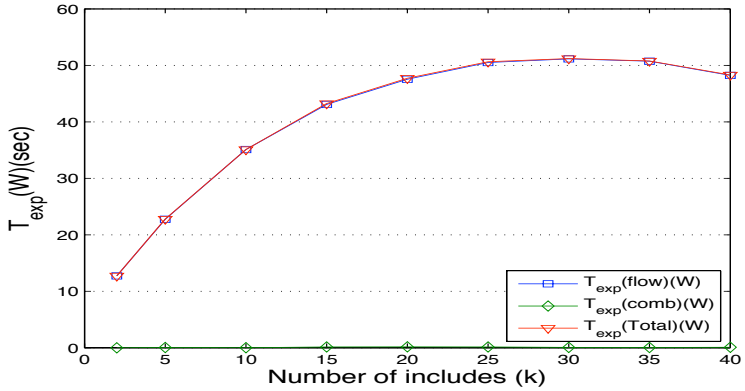

Fig. 11. $T_{flow}(W)$, $T_{comb}(W)$, $T_{total}(W)$ vs k: $n = 50$, $\lambda = 6$

TABLE IV
COMPARING RESULTS WITH K-SHORTEST PATHS APPROACH. $N = 25$, $\lambda = 4$, $k = 4$

ID	src	dest	K	$l(KSP)$	$l(W)$	$l(R)$	**ID**	src	dest	K	$l(KSP)$	$l(W)$	$l(R)$
1	20	3	388	9	13	9	51	17	2	2310	11	12	11
2	1	16	1306	10	11	10	52	17	3	119	8	8	8
3	2	21	509	10	15	10	53	17	15	1141	11	11	11
4	1	16	742	10	10	10	54	24	11	1157	11	11	13
5	11	12	485	9	12	9	55	1	20	20	6	6	6
6	23	21	92	7	8	7	56	8	12	1157	11	15	14
7	19	13	840	10	13	10	57	10	8	2087	10	12	10
8	8	2	442	9	11	11	58	14	21	672	10	13	10
9	16	2	4909	12	14	12	59	9	0	1116	10	14	10
10	9	17	1338	11	14	12	60	18	15	291	8	12	8
11	2	15	3529	10	13	10	61	19	11	907	10	14	10
12	19	13	1988	9	13	9	62	8	0	2192	11	11	11
13	3	18	106	7	11	7	63	21	2	292	9	9	9
14	15	3	160	8	12	10	64	22	19	353	9	12	10
15	0	3	3946	11	11	11	65	21	22	179	8	8	8
16	15	8	1613	10	13	10	66	18	19	1758	10	14	10
17	13	15	137	7	9	7	67	21	3	3050	10	10	10
18	6	3	77	7	12	7	68	18	24	1882	11	0	12
19	13	5	294	9	9	9	69	6	16	220	8	8	8
20	19	22	454	7	11	7	70	15	5	163	8	12	10
21	5	21	728	10	10	10	71	3	1	78	7	7	7
22	2	13	72	7	7	7	72	10	2	1041	10	12	12
23	23	2	2246	10	12	10	73	13	14	31	6	10	6
24	10	23	463	9	14	9	74	12	2	1333	10	14	10
25	6	0	834	10	11	10	75	7	21	375	9	12	12
26	6	17	724	9	9	9	76	13	11	617	9	13	9
27	15	19	458	9	13	10	77	0	16	649	10	11	11
28	10	2	333	8	9	8	78	20	3	777	10	12	12
29	17	20	623	9	13	12	79	16	1	3994	11	15	11
30	19	0	9	6	6	6	80	9	16	806	10	15	12
31	13	23	669	10	14	11	81	13	24	298	9	11	11
32	15	7	359	9	9	12	82	12	15	1380	9	12	9
33	3	6	2587	11	11	11	83	17	24	1662	11	18	13
34	11	2	906	9	10	9	84	7	14	93	8	11	11
35	21	5	26	6	6	6	85	12	23	312	9	9	9
36	10	7	86	7	10	7	86	18	24	1107	10	13	10
37	6	3	262	8	10	8	87	11	22	347	8	10	8
38	19	16	500	9	13	9	88	23	15	128	7	7	7
39	16	3	2499	11	14	11	89	11	1	230	8	8	8
40	13	14	1082	10	15	10	90	16	8	19	5	8	5
41	0	19	1389	11	16	16	91	16	4	116	8	11	9
42	1	4	153	8	8	8	92	8	9	580	9	12	9
43	17	11	446	9	11	11	93	16	12	1017	10	11	11
44	4	23	894	10	11	11	94	2	22	331	8	8	8
45	22	14	475	9	13	9	95	0	10	306	9	15	10
46	7	6	389	9	11	11	96	14	2	490	9	10	10
47	22	9	10	6	6	6	97	16	7	958	10	14	13
48	14	8	260	9	9	9	98	4	9	358	8	10	12
49	3	5	131	7	8	8	99	9	7	122	8	11	11
50	12	2	805	9	0	9	100	22	0	306	8	14	8

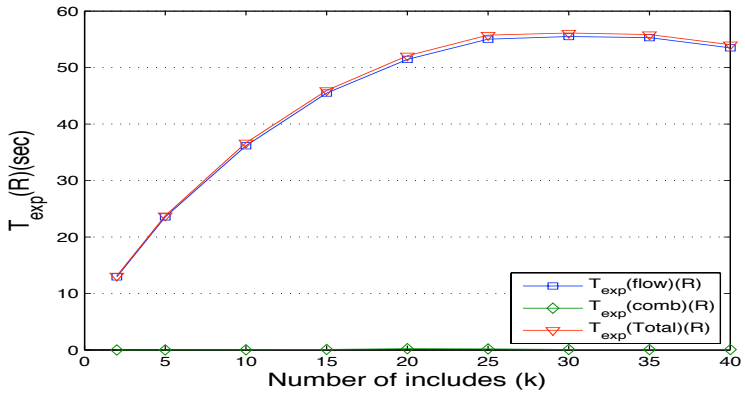

Fig. 12. $T_{flow}(R), T_{comb}(R),\ T_{total}(R)$ vs k: $n = 50$, $\lambda = 6$

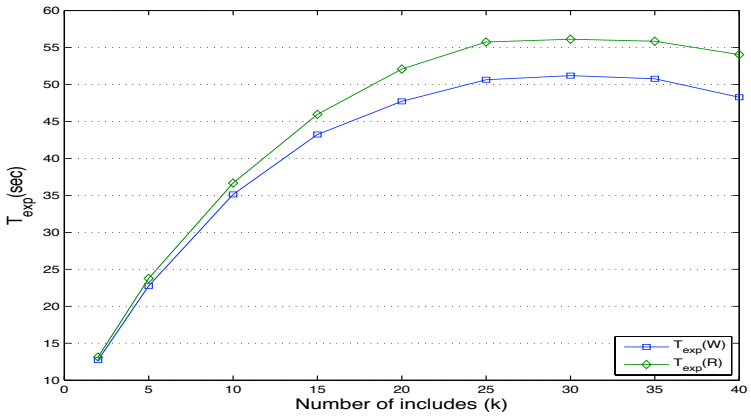

Fig. 13. T_{exp} vs *number of include* nodes (k): $n = 50$, $\lambda = 6$

result is shown in TABLE IV. Fig. 14 reports the results of the all 100 experiments, comparing the number of hops in the two simple paths computed by K-shortest path algorithm (labeled as KSP Approach) and Algorithm 3 with re-ordering I (labeled as FLOW Approach), respectively. In many cases, the simple path found by Algorithm 3 has the same number of hops of the shortest simple path found by K-shortest path algorithm. The later algorithm, however, may require a considerably large value of K to find the simple path that contains all the nodes in I, as reported in Fig. 15. Further, T_{exp} required by *KSP approach* is compared with that of *Flow approach* described in Algorithm 3. In Fig. 14, T_{exp} is plotted in *log scale* for each of the 100 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 of the cases, *KSP-approach* needs much longer time than time required by Algorithm 3. Hence, Algorithm 3 is able to compute near shortest paths that include all the nodes in I in reasonable time.

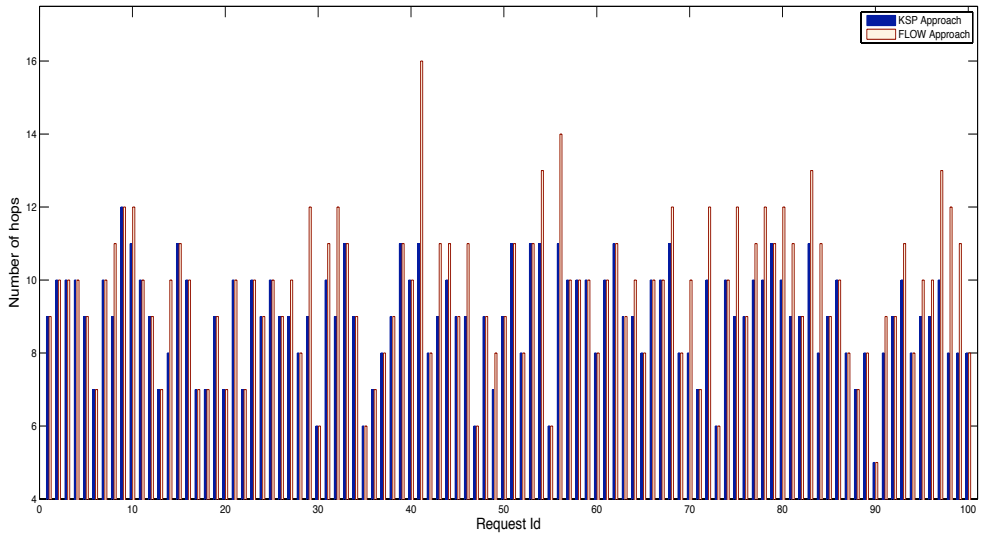

Fig. 14. *K-shortest path* vs Algorithm 3 with reordering

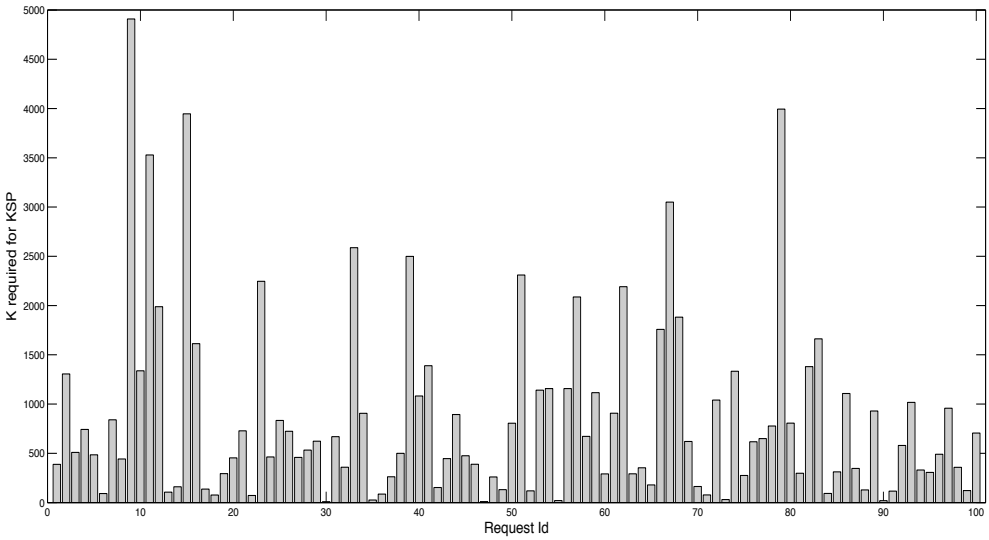

Fig. 15. *K required for KSP*

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 only one node in the path is polynomially solvable. However, including 2 or more nodes is shown to be *NP-complete*. In this paper, we presented a heuristic algorithm to compute a simple path with multiple must-include nodes. Our heuristic algorithm follows the *divide and conquer* approach, by dividing the problem in two subproblems. The two sequential

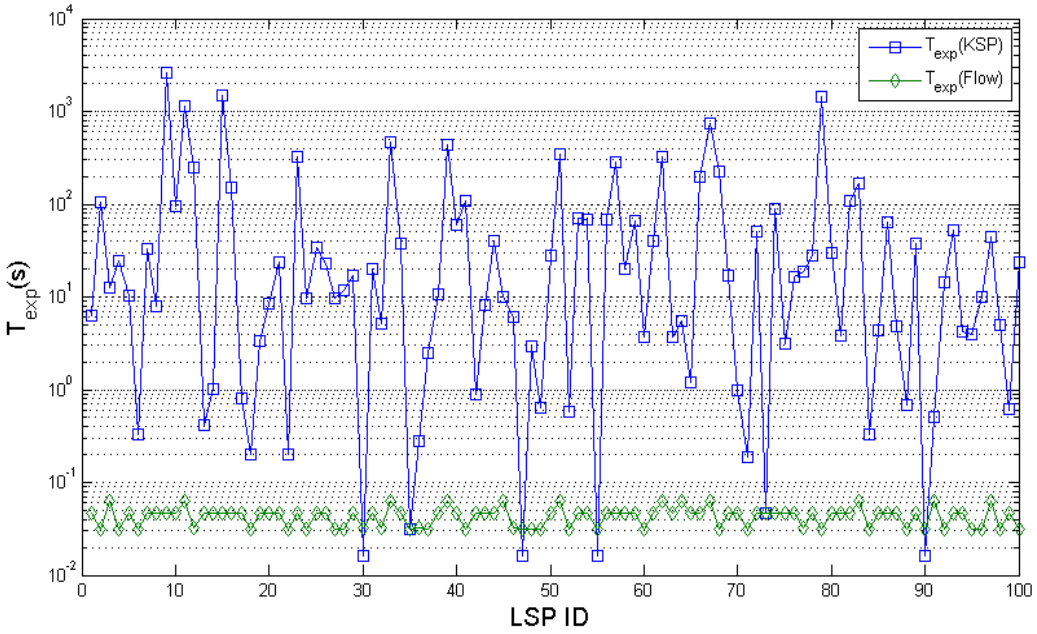

Fig. 16. T_{exp} (Flow-approach vs KSP-approach); $n = 25,\ \lambda = 4,\ k = 4$

subproblems are (i) computing candidate paths for individual segments of the end-to-end route and (ii) merging them to form an end-to-end simple path for a given pair of terminals. The *max-flow* algorithm is used to compute the maximum number of edge-disjoint candidate paths for each segment. The *backtracking* algorithm is used to choose and combine candidate paths to form the end-to-end route, as its run time does not grow exponentially in this application. 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] L. Berger, "RFC 3473 - Generalized Multi-Protocol Label Switching (GMPLS) Signaling Resource ReserVation Protocol-Traffic Engineering (RSVP-TE) Extensions," January 2003.

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

[4] M. Gondran, M. Minoux, and S. Vajda, *Graphs and algorithms*. Wiley-Interscience Series in Discrete Mathematics.Wiley (Chichester [West Sussex], New York), 1984.

[5] T. Yi and K. G. Murthy, "Finding maximum flows in networks with nonzero lower bounds using preflow methods," Department of Indusrtial and Operaion Engineering The University of Michigan Ann Arbor, MI 48109-2117, Tech. Rep., July 1991.

[6] R. Burkard, V. Deineko, R. van Dal, J. van der Veen, and G. Woeginger, "Well-solvable special cases of the tsp: A survey," 1995. [Online]. Available: citeseer.ist.psu.edu/burkard95wellsolvable.html

[7] E. L. Gilmore, P.C. and D. Shmoys, "Well-solved special cases," 1985, in E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy-Kan and D.B. Shmoys (eds.), Chapter 4 in The Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization. Wiley: Chichester, pp. 87-144.

[8] H. J. Miller and S.-L. Shaw, *Geographic Information Systems for Transportation: Principles and Applications*. Oxford University Press US, 2001.

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

[10] 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.

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