Computing Alternate Multicast Trees with Maximum Latency Guarantee

Limin Tang*, Wanjun Huang*, Miguel Razo*, Arularasi Sivasankaran*,
Paolo Monti†, Marco Tacca*, and Andrea Fumagalli*
* OpNeAR Lab, Erik Jonsson School of Engineering and Computer Science
The University of Texas at Dallas, Richardson, TX, USA
{lxt064000, wxh063000, mrazora, axs075200, mtacca, andreaf}@utdallas.edu
† NeGONet Group, School of Information and Communication Technology, ICT-FMI
Royal Institute of Technology (KTH), Kista, Sweden
{pmonti}@kth.se

***Abstract*—The growing demand for online media content delivery and multi-player gaming is expected to increase the amount of multicast service requests in both public and private networks. Careful traffic engineering of multicast service requests is becoming increasingly essential, as establishing the lowest cost tree, e.g., shortest path tree, in the network for every individual multicast request does not always ensure a global optimization.**

In this paper, the authors investigate the use of alternate tree routing, according to which multiple sub-optimal tree candidates are computed for each multicast request. When performing global routing optimization, each request is then assigned one of the tree candidates, in a way that yields the desired result, e.g., to minimize the number of (active) links that are required in the network to bear the entire set of requests. A key component of the investigated approach is the timely construction of the set of alternate trees for every given root and destination set.

An algorithm is proposed to compute multiple sub-optimal tree candidates with a guaranteed upper bound on the maximum end-to-end transmission latency. The algorithm builds on the widely used *K* shortest path algorithm, generalizing the technique of computing *K* candidate shortest paths to obtain a number of candidate sub-optimal trees with desirable properties. Simulation experiments obtained for a multi-protocol label switching (MPLS) traffic engineering network are discussed to investigate the effectiveness, performance, and scalability of the proposed algorithm.

I. Introduction

When dealing with unicast traffic in connection oriented network (MPLS for example), the choice of shortest path is optimal for the individual label switch path (LSP) request. However, when looking at the network as a whole, shortest path routing may often lead to uneven resource utilization, e.g., some links are underutilized, others are congested. *Alternate routing* is a way to address this problem by allowing each LSP request to be assigned one of the many possible sub-optimal candidate paths. At the cost of a controlled degradation of routing optimality for every individual LSP, a global network optimization can thus be performed by searching across the available candidate paths and choosing the most suitable one to achieve the desired goal. One simple approach to compute multiple candidate paths for a given LSP is to find the K shortest paths, a technique that has been used extensively to solve a number of routing optimization problems [1],[2],[3],[4]. Various algorithms are available to compute the K shortest paths from a source to a destination node [5],[6],[7],[8]. Every newly added candidate path is guaranteed to be the next best path available to the LSP, thus offering a controlled degradation of LSP's routing optimality as the search space for global optimality is augmented. A similar approach to handling multicast (as opposed to unicast LSP) requests, i.e. compute multiple candidate trees for each request, is not available.

In this paper the authors propose a time efficient algorithm to compute multiple sub-optimal candidate trees from a root to a set of destinations. Each candidate tree is guaranteed to have an upper bound on the maximum hop count from the root to every destination. Complexity, correctness, and a practical implementation of the proposed algorithm are discussed, assuming that every node in the network has unlimited multicast capability. Simulation results are used to estimate the benefit of applying alternate routing to computing traffic engineering for multicast requests in a number of network and traffic scenarios.

II. Algorithm for Computing Multiple Candidate Trees

This section contains a description of the proposed algorithm, along with the analysis of its complexity and proof of its correctness. The algorithm objective is to compute a number of candidate trees in a graph for a given multicast request, specified as one root and a set of destination vertices. The algorithm first computes the K shortest paths for each root-destination pair of the multicast request. Multiple trees are then constructed for the multicast request from the K paths, while guaranteeing an upper bound on the maximum hop-count from the root to all destinations. The following notation is used:

- N: number of vertices in the network;
- M: number of edges in the network;
- s: root of the multicast request;
- D: set of destinations of the request;
- $d_i \in D$: the ith destination of the request;
- n: number of destinations in D;
- P: set of ordered loopless paths from s to all $d_i \in D$;
- P_i: set of ordered loopless paths from s to d_i;
- p_{ij}: jth shortest path from s to d_i;

- $t(V, E)$: a multicast tree with V as the set of vertices and E as the set of edges;
- T: set of multicast candidate trees;
- $h(p)$: hop-count of path p;
- $h_i(t)$: hop count from s to d_i in tree t.

A. Algorithm Description

The algorithm comprises two procedures:

Procedure 1 creates set P_i, i.e., it computes a set of K ordered loopless shortest paths for every pair (s, d_i), $d_i \in D$, using hop count as metric, which often is the dominant factor responsible for transmission latency in MPLS networks.

Procedure 2 makes use of the paths in set P_i to compute a number of trees to be added to T. T is an empty set before running this procedure. The procedure works iteratively, adding one tree at a time. At each iteration, a new tree $t(V, E)$ is computed as follows. First, initialize V and E to be empty sets. Then, vertices and edges are increasingly added to $t(V, E)$ till a satisfactory tree is obtained. One at a time, all destinations $d_i \in D$ are considered as follows: choose a path $p_{ij} \in P_i$ that was not used in any previous iteration and add all the vertices and edges in this path to $t(V, E)$. Then for each of the remaining destinations $d_{i'}$, choose the shortest path from s to $d_{i'}$ in set $P_{i'}$, i.e., $p_{i'1}$. Vertices and edges in this path are added to $t(V, E)$, starting from $d_{i'}$ and traveling backward along the path, until one edge that has a node already in $t(V, E)$ is added. Then, move to the next destination node $d_{i'}$ till all destinations in D are considered.

Pseudocode of Procedure 2:

```
$T \leftarrow \emptyset$
For ($P_i \in P$)
   For ($p_{ij} \in P_i$)
      create a new tree $t(V, E)$, $V \leftarrow \emptyset$, $E \leftarrow \emptyset$
      For vertex $v \in p_{ij}$
         $V = V \cup v$
      EndFor
      For edge $e \in p_{ij}$
         $E = E \cup e$
      EndFor
      For $d_{i'} \in D$ and $i' \neq i$
         $v \leftarrow d_{i'}$
         $e \leftarrow$last edge of $p_{i'1}$
         While ($v \notin V$)
            $V = V \cup v$
            $E = E \cup e$
            $v = v$'s upstream vertex on $p_{i'1}$
            $e = e$'s upstream edge on $p_{i'1}$
         EndWhile
      EndFor
      If $t(V, E) \notin T$
         $T = T \cup t(V, E)$
      EndIf
   EndFor
EndFor
```

An example of how the algorithm runs on the NSFNet topology (Fig. 1) is given next. Consider the multicast request with root $s = 0$ and destinations $D = \{2, 6, 11\}$. Let $K = 2$. The following 2 shortest paths from s to every $d_i \in D$ are computed by the algorithm:

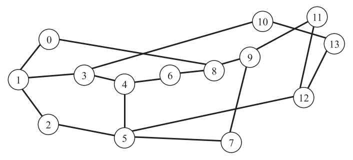
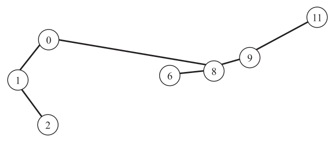


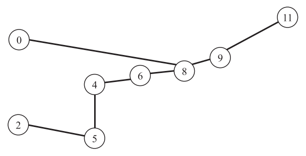
Fig. 1: NSFNet Topology.

- P_1
  p_{11}: $0 \rightarrow 1 \rightarrow 2$
  p_{12}: $0 \rightarrow 8 \rightarrow 6 \rightarrow 4 \rightarrow 5 \rightarrow 2$
- P_2
  p_{21}: $0 \rightarrow 8 \rightarrow 6$
  p_{22}: $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 6$
- P_3
  p_{31}: $0 \rightarrow 8 \rightarrow 9 \rightarrow 11$
  p_{32}: $0 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 12 \rightarrow 11$

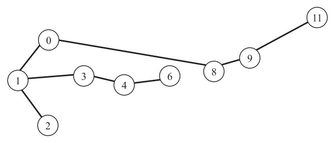
The first tree is built by using p_{11}, p_{21} and p_{31}, which is the shortest path tree (SPT) (Fig. 2a). The second tree is built as follows: first all vertices and edges in p_{12} are added to the tree; then vertices and edges in p_{21} are added sequentially to the tree, beginning from vertex 6. Note that since vertex 6 is already on the tree, no action is required and the algorithm moves to the next step. Path p_{31} is considered next, and vertices and edges in p_{31} are added to the tree, beginning with vertex 11. The adding stops once vertex 8 (already in the tree) is reached. The computed tree is shown in Fig. 2b. Similarly, two more trees (shown in Figs. 2c and 2d) can be built based on paths (p_{11}, p_{22} and p_{31}), and paths (p_{11}, p_{21} and p_{32}), respectively.



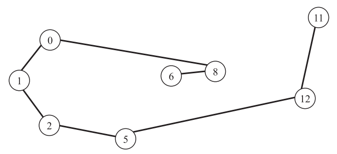
(a) Tree 1 (SPT)



(b) Tree 2



(c) Tree 3



(d) Tree 4

Fig. 2: Multiple trees built by the algorithm for the multicast request with $s = 0$, $D = \{2, 6, 11\}$ on the NSFNet topology.

The number of trees computed by the algorithm for a given multicast request is potentially large (as shown in Section II-C). This property offers great flexibility when routing multicast requests, which is a unique feature of the algorithm. The downside is that both heavy computation and large memory consumption may be required to compute and store all the possible candidate trees upfront. A way to circumvent this drawback while retaining the ability to explore a large number of candidate trees during optimization is to make use of a light-weight implementation of the algorithm as follows.

Only the K shortest paths for each root-destination pair are precomputed and stored in memory. The computation of one or more candidate trees is performed only when alternate routes of a multicast request are needed. When and how many such routes are to be computed are both decisions to be made by the combinatorial optimization algorithm that is searching for the most optimal multicast request-to-candidate tree assignment.

B. Algorithm Complexity

This section evaluates the complexity of the algorithm's two procedures.

Procedure 1 computes the K shortest paths from source to all destinations of the multicast request. Since computing K shortest paths for a pair of vertices has complexity $O(KN(M+N\log N))$ [8], the complexity of Procedure 1 is $O(nKN(M+N\log N))$.

Procedure 2 has at most nK iterations, given that there are K shortest paths from every s to all $d_i \in D$ pair. Each iteration comprises three steps:

1) add vertices and edges of the jth shortest path from s to d_i to the tree;
2) add vertices and edges of the shortest path from s to $d_{i'}(i' \neq i)$ to the tree;
3) check whether the computed tree already exists in T or not.

Since every path can have at most M edges and at most $M+1$ vertices, step 1 has complexity $O(M)$. Similarly, step 2 has complexity $O((n-1)M)$. Step 3 compares a newly computed tree with all trees in T. Hence, in the worse case it takes $O(M)$ to determine whether two trees are distinct and there can be at most $n(K-1)$ existing trees in T (see Section II-C). Consequently, step 3 has complexity $O((nK-1)M) = O(nKM)$. In conclusion, the complexity of Procedure 2 is $nK(O(M)+O((n-1)M)+O(KnM)) = nK(O((K+1)nM) = O(n^2K^2M)$.

Combining both procedures, the maximum complexity of the algorithm is $O(nKN(M+N\log N)) + O(n^2K^2M) = O(nK((N+nK)M+N^2\log N)))$.

C. Proof of Correctness

In this section proof is given that the algorithm can compute at least K distinct candidate trees for the multicast request, given that K distinct shortest paths can be found for at least one root-destination pair s-d_j. For the proof of correctness of Procedure 1, see [5] and [8]. The proof of correctness of Procedure 2 comprises two parts. First, it is proven that $t(V,E)$ created in line 4-20 of Procedure 2 is a tree (Proof 1), then it is proven that in the inner loop of Procedure 2 (line 3-24), every $p_{ij} \in P_i$ yields a distinct and unique tree (Proof 2).

Proof 1: $t(V,E)$ is a tree. At initialization, $t(V,E)$ only contains vertices and edges of path p_{ij}. At this time $t(V,E)$ is therefore a loopless tree without branches. Recall that for destination $d_{i'}(i' \neq i)$, edges and vertices of $p_{i'1}$ (shortest path from s to $d_{i'}$) are added sequentially to $t(V,E)$, beginning from the last vertex ($d_{i'}$) of $p_{i'1}$, and proceeding backward towards the root, till a vertex is found that is already in V. From this iterative step, it can be observed that:

- since $p_{i'1}$ is a loopless path and its edges and vertices are added till a vertex in the tree is reached, $t(V,E)$ is still loopless;
- since there exists at least one vertex that is in both $p_{i'1}$ and $t(V,E)$, namely s, the stop condition is always reached.

Based on these two observations it is clear that after every step, $t(V,E)$ remains a tree.

Proof 2: in the inner loop of Procedure 2 (line 3-24), every $p_{ij} \in P_i$ yields a distinct and unique tree. Let t_j and $t_{j'}$ be two candidate trees computed from p_{ij} and $p_{ij'}$, respectively, and let $j \neq j'$. Note that all edges and vertices in $p_{i'1}$ ($i' \neq i$) added to t_j and $t_{j'}$ belong to SPT; when $j=1$, the computed tree is SPT. Then it is observed that two different paths with same source and same destination must depart from each other at one node and must join each other at another node, hence there must be two edges that belong to these two paths respectively having different sources and same destination. In other words for any two paths p_{ij} and $p_{ij'}$, there are at least two edges $e(v_1,v_2) \in p_{ij}$ and $e'(v'_1,v'_2) \in p_{ij'}$ that satisfy $v_1 \neq v'_1$ and $v_2 = v'_2$. Obviously, $e \notin p_{ij'}$ and $e' \notin p_{ij}$. Also, e and e' cannot both be in SPT, otherwise there would be two shortest paths from s to v_2, which is not allowed. Thus, either e or e' is not in SPT. Without loss of generality, assume that e is not in SPT. Then e cannot appear in any path of $p_{i'1}$. Combined with $e \notin p_{ij'}$, it is clear that $e \notin t_{j'}$. Since $e \in p_{ij}$ and therefore $e \in t_j$, t_j and t'_j must be two distinct trees.

Both Proof 1 and 2 ensure that a tree computed from p_{ij} is distinct from a tree computed from $p_{ij'}$, when $j \neq j'$. Since $j \in \{1,\ldots,K\}$ (i.e., at least K shortest paths from s to d_i can be found), K distinct trees can be computed by the inner loop of Procedure 2 (line 3-24).

The maximum number of distinct trees that can be built by the algorithm can also be computed. There are n iterations of the inner loop of Procedure 2. If K shortest paths from s to d_i can be found for every destination $d_i \in D$, Procedure 2 will compute a total of nK trees. Note that SPT is computed n times in this case, hence if all other trees are distinct, the number of distinct candidate trees computed by Procedure 2 has upper bound $nK-(n-1) = n(K-1)+1$.

D. Proof of Maximum Hop Count

In this section it is demonstrated that for a multicast tree $t(V,E)$ computed from p_{ij} (line 3 of Procedure 2) and $p_{i'1}$ ($i' \neq i$) (line 13 of Procedure 2), the hop count from s to $d_{i'} \in D$ in $t(V,E)$ can not be greater than the hop count from s to $d_{i'} \in D$ using the shortest path tree (SPT) plus x_{ij}, where $x_{ij} = h(p_{ij}) - h(p_{i1})$:

$$h_{i'}(t) \leq h_{i'}(\text{SPT}) + x_{ij} \;\forall\; i' \in [1,n] \tag{1}$$

First note that $h(p_{i'1}) = h_{i'}(\text{SPT})$ always holds. Since all the edges of p_{ij} are in t, $h_i(t) = h(p_{ij})$, i.e. $x_{ij} = h_i(t) - h_i(\text{SPT})$, which means for $i' = i$, Eq. (1) holds. For $i' \neq i$, if $p_{i'1}$ and p_{ij} do not have any common vertex except s, then path from s to $d_{i'}$ in t is the shortest path from s to $d_{i'}$, i.e., $h_{i'}(t) = h_{i'}(\text{SPT})$, thus Eq. (1) holds. Otherwise, path from s to $d_{i'}$ in t contains edges either in $p_{i'1}$ or in p_{1j}, and the number of hop counts from s to $d_{i'}$ in t can be at most $h(p_{i'1}) + h(p_{ij}) - h(p_{i1}) = h_{i'}(\text{SPT}) + x_{ij}$. Hence, $h_{i'1}(t) \leq h_{i'1}(\text{SPT}) + x_{ij}$, Eq. (1) still holds. Thus, the claim in Eq. (1) is always correct.

Since transmission latency in MPLS networks is usually dependent on the hop count from source to destination, and the hop count from s to any $d_i \in D$ in a tree built by the algorithm has an upper bound (as shown in Eq. (1)), it is expected that the maximum transmission latency in the built tree has a guaranteed upper bound, too.

E. A Relaxed Version of the Algorithm

The original algorithm is quite strict in that, when paths from source to destinations are chosen to build the tree, only one of them can be a non-shortest path (p_{ij}), all other paths must be shortest paths ($p_{i'1}$). This condition can be relaxed to let all these paths be chosen from any of the computed K shortest paths, i.e., any $p_{i'j} \in P_{i'}$, then up to K^n distinct trees can be built for each multicast request and hence providing a much larger pool of candidate trees. Performance differences between the original algorithm and the relaxed one are compared in Section III-B.

One thing to notice is that since the relaxed algorithm computes potentially many more trees than the original one, the light-weighted implementation presented in Section II-A becomes even more valuable in the former case.

III. Experiments and Simulation Results

Two experiments are carried out to study the effectiveness of the proposed algorithm. In Experiment I, the focus is on the effect of K on the whole network optimization and average hop count of the chosen trees. In Experiment II, the original and relaxed algorithms are compared.

A. Experiment I

Three MPLS network topologies are randomly generated: network 1 has 10 vertices and 60 unidirectional edges; network 2 has 50 vertices and 300 unidirectional edges; and network 3 has 100 vertices and 600 unidirectional edges. Any edge can be equipped with one transmission link with maximum transmission capacity C. A number of multicast requests — with number of destinations ranging from 2 to 8 and bandwidth request $C/100$ — are generated uniformly for each network.

The simulation experiments are carried via a network design tool [9]. The tool uses simulated annealing (SA) algorithm to minimize the number of edges that are equipped with a C capacity link, in order to support the set of multicast requests. At each iteration the SA algorithm randomly picks one multicast request and randomly chooses one candidate tree for that request, using the algorithms described in Section II. Annealing temperature is gradually decreased till it is not possible to accept any different candidate tree other than the one currently assigned to every request. As already mentioned, the purpose of these experiments is to determine the degree of optimization of the whole network as a function of the number of candidate trees that is available to each request (i.e., value of K).

Numbers of required links in the three networks are shown in Figs. 3, 4, and 5, respectively. In all three networks the number of required links is less when more candidate trees are allowed for each request (i.e., K is larger). The reduction gain is significant especially when the total number of requests is relatively small. This result is reasonable, since when the network load is low, there is more room for optimization.

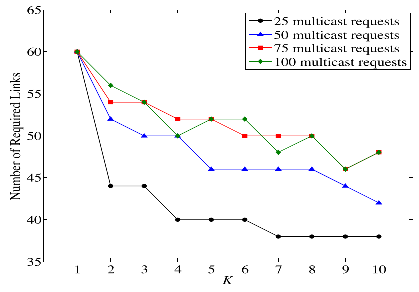


Fig. 3: Effect of K on the number of required links for a set of multicast requests, the network has 10 vertices and 60 unidirectional edges.

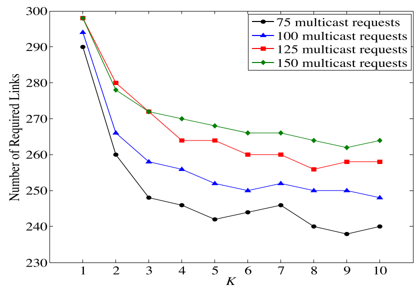


Fig. 4: Effect of K on number of required links for a set of multicast requests, the network has 50 vertices and 300 unidirectional edges.

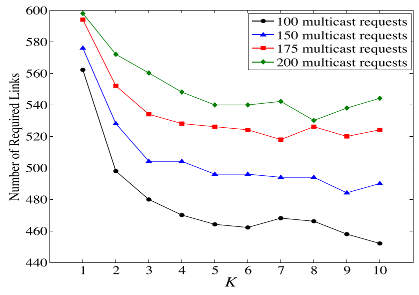


Fig. 5: Effect of K on number of required links for a set of multicast requests, the network has 100 vertices and 600 unidirectional edges.

It can also be seen that increments of K yield substantial gain when K is small. This is natural since when the number of available trees is small, one more alternate tree can increase the system flexibility a lot. However, if the number of available trees is large, improvement of one more alternate tree for optimization is relatively minimal and can even cause an adverse effect in some cases, as complexity of the optimization problem increases more than polynomially and the SA algorithm may not be able to efficiently find a sub-optimal solution in this case.

The *average maximum hop count* (AMHC) in each experiment is computed as follows: take the maximum hop count of each multicast request (i.e., hop count of the longest source-destination path in the tree assigned to the request) and compute the average over all requests. The AMHC results are shown in Figs. 6, 7, and 8, respectively. As K increases, AMHC generally also increases. This relationship is intuitive since longer paths are computed when a larger K is used. Notice that when $K = 1$, SPT is assigned to every multicast request. This is the case with minimal AMHC. Combined with Figs. 3, 4, and 5, trade-offs between the overall network cost (i.e., number of required links) and hop count for the multicast requests is clearly visible in all three networks.

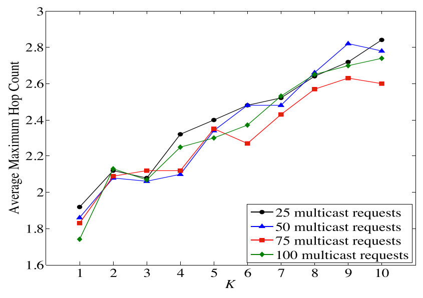


Fig. 6: Effect of K on AMHC for a set of multicast requests, the network has 10 vertices and 60 unidirectional edges.

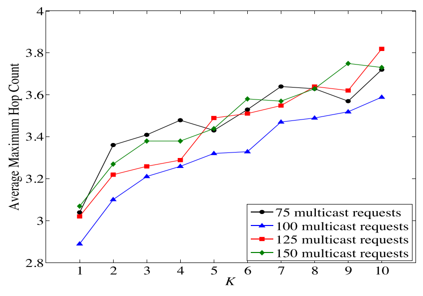


Fig. 7: Effect of K on AMHC for a set of multicast requests, the network has 50 vertices and 300 unidirectional edges.

B. Experiment II

The following experiments are designed to compare the original against the relaxed algorithm in terms of both the number of required links and AMHC. The network topologies and set of multicast requests are the same as in Experiment I.

The required number of links is reported in Figs. 9, 10 and 11, respectively. The average hop count is reported in Figs. 12,

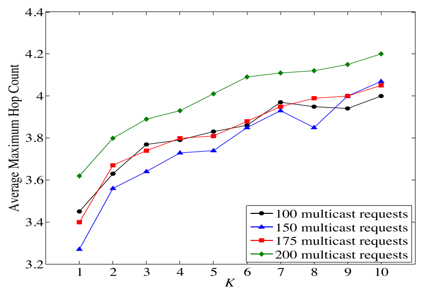


Fig. 8: Effect of K on AMHC for a set of multicast requests, the network has 100 vertices and 600 unidirectional edges.

13 and 14, respectively. As expected, relaxing the hop count constraint yields a reduced number of required links in the network, at the price of an increased AMHC of trees assigned to requests. It is also worth to note that when K is small, the relaxed algorithm performs worse than the original one in both network cost optimization and average transmission latency. The reason for this behavior is that when K is small, the number of candidate trees for each request that can be found by using the relaxed algorithm is not significantly larger compared to the number of candidate trees found by using the original algorithm.

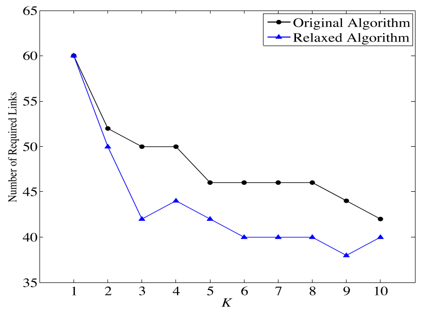


Fig. 9: Effects of original and relaxed algorithms on number of required links, the network has 10 vertices and 60 unidirectional edges and 50 multicast requests.

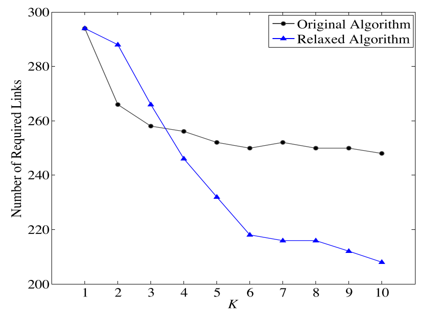


Fig. 10: Effects of original and relaxed algorithms on number of required links, the network has 50 vertices and 300 unidirectional edges and 100 multicast requests.

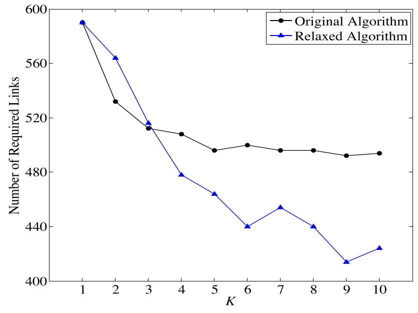


Fig. 11: Effects of original and relaxed algorithms on number of required links, the network has 100 vertices and 600 unidirectional edges and 125 multicast requests.

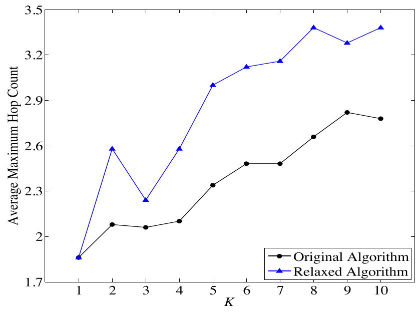


Fig. 12: Effects of original and relaxed algorithms on AMHC, the network has 10 vertices and 60 unidirectional edges and 50 multicast requests.

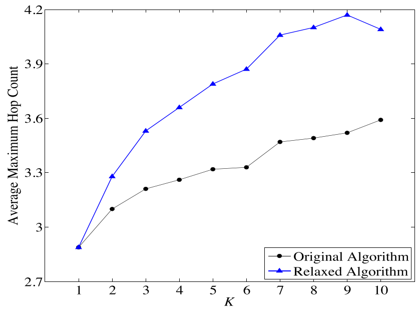


Fig. 13: Effects of original and relaxed algorithms on AMHC, the network has 50 vertices and 300 unidirectional edges and 100 multicast requests.

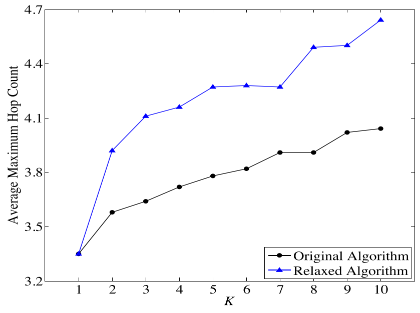


Fig. 14: Effects of original and relaxed algorithms on AMHC, the network has 100 vertices and 600 unidirectional edges and 125 multicast requests.

IV. Conclusion

The study in this paper is motivated by the increasing demand for multicast services in connection oriented (MPLS) networks, such as online media content delivery and multi-player gaming. These services are expected to constitute a significant portion of the overall network traffic in the years ahead, and their solution for traffic engineering in the network is becoming increasingly important.

The traffic engineering solution chosen in this paper is to compute multiple multicast candidate trees for each multicast request, and then solve a combinatorial problem to assign one of the candidate trees to every request. Although some of the candidate trees may be sub-optimal choices for the individual request, global network optimization becomes possible by virtue of solving the combinatorial problem. The set of candidate trees is computed by combining K shortest paths — computed for the root destination pairs — in such a way that an upper bound is guaranteed for the maximum hop-count, hence maximum transmission latency, from root to every destination of the request.

The ability of the proposed algorithm to compute an effective set of multiple candidate trees is assessed indirectly by running a series of simulation experiments. The experiments seem to suggest that the proposed technique yields noticeable network cost reduction (or number of required links) across a number of cases, regardless of the network size and number of multicast requests. It is also noted that the algorithm offers good scalability property and may find application to many practical scenarios.

V. Acknowledgements

This research was supported in part by NSF Grant No. CNS-0435393.

References

[1] C.-F. Hsu, J. Y. Hui, C. feng Hsu, and J. Y. Hui, “Load-balanced k-shortest path routing for circuit-switched networks,” *In Proceedings of IEEE NY/NJ Regional Control Conference*, 1992.

[2] A. K. Yashar Ganjali, “Load balancing in ad hoc networks: Single-path routing vs. multi-path routing,” *IEEE Infocom*, 2004.

[3] Z. Jia and P. Varaiya, “Heuristic methods for delay constrained least cost routing using k-shortest-paths,” *IEEE Transactions on Automatic Control*, vol. 51, no. 4, 2006.

[4] A. Esfahani and M. Analoui, “Widest k-shortest paths q-routing: A new qos routing algorithm in telecommunication networks,” *Computer Science and Software Engineering, International Conference on*, vol. 4, pp. 1032–1035, 2008.

[5] J. Y. Yen, “Finding the k shortest loopless paths in a network,” *Management Science*, vol. 17, p. 712, 1971.

[6] D. Eppstein, “Finding the k shortest paths,” *SIAM J. Comput.*, vol. 28, no. 2, pp. 652–673, 1999.

[7] V. M. Jiménez and A. Marzal, “Computing the k shortest paths: A new algorithm and an experimental comparison,” in *WAE ’99: Proceedings of the 3rd International Workshop on Algorithm Engineering*. London, UK: Springer-Verlag, 1999, pp. 15–29.

[8] E. Q. Martins and M. M. Pascoal, “A new implementation of Yen’s ranking loopless paths algorithm,” *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, vol. 1, p. 121, 2003.

[9] M. Razo, A. Litovsky, W. Huang, A. Sivasankaran, L. Tang, H. Vardhan, P. Monti, M. Tacca, and A. Fumagalli, “The planet-PTN module: a single layer design tool for packet transport networks,” *the 14th IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks*, 2009.