Routing in an Internet-Scale Network Emulator

Jay Chen, Diwaker Gupta, Kashi V. Vishwanath, Alex C. Snoeren and Amin Vahdat
University of California, San Diego, CA 92093 USA
j2chen@ucsd.edu, {dgupta, kvishwanath, snoeren, vahdat}@cs.ucsd.edu

Abstract

One of the primary challenges facing scalable network emulation and simulation is the overhead of storing network-wide routing tables or computing appropriate routes on a per-packet basis. We present an approach to routing table calculation and storage based on spanning tree construction that provides an order of magnitude reduction in routing table size for Internet-like topologies. In our approach, we maintain a variable number of spanning trees for a given topology and choose the path between two hosts in each tree, choosing the shortest. We also populate offline a negative cache of actual shortest paths for source-destination pairs—typically a few percent of the total—where the lookups result in sub-optimal routes. We have implemented our technique in a popular network emulator, ModelNet, and show that our enhanced version can emulate Internet topologies 10–100 times larger than previously possible.

1. Introduction

Developing and implementing next-generation, robust, large-scale networked systems requires a deep understanding of system behavior under a wide variety of conditions. In addition to evaluation of live systems, network simulation [5, 11] and real-time emulation [13] have become essential tools to understanding complex system interactions. A primary challenge facing the designer of any simulation or emulation environment is managing the trade-off between accuracy and scalability. For instance, it is impossible to capture the characteristics of the entire Internet in any simulation; only a subset of network characteristics can be accurately captured without undue impact on overall system scalability. The key question then becomes: What is the largest system that can be accurately simulated or emulated on the target hardware platform?

We have been conducting research into building a scalable and accurate network emulation environment, called ModelNet [10]. Briefly, ModelNet subjects the inter-node packet communication of unmodified applications running on unmodified operating systems/hardware to the hop-by-hop characteristics of a target large-scale network topology. The first step in this real-time emulation is looking up the path that a packet would take through the target topology. Currently, ModelNet assumes static routing and stores all-pairs shortest path information in a pre-computed $O(n^2)$ table, where n is the number of communicating end hosts.

The current obstacle to scaling ModelNet beyond a few thousand communicating hosts on a commodity cluster is the size of the routing table that nodes must store. The goal of this work is to develop techniques to remove this obstacle. The memory requirements for routing have received a great deal of attention, but, traditionally, researchers have studied how to distribute routing information amongst the nodes of the network such that a packet can be appropriately forwarded at each node. In this context, Peleg has shown that a lower bound of $\Omega(n^{1+1/6})$ bits of information are required in the network [7]. Gavoille and Pérennès further show that, for any routing scheme, there exists a network that requires $\Theta(n^2 \log d)$ bits of storage, where d is the degree of the network [3]. Unfortunately, routing schemes approaching these limits often make use of considerable space in the packets themselves (in the form of packet routing headers) and, more importantly, the route lookup operation at each node can be expensive. In our environment, entire paths are computed ahead of time, using global knowledge of network topology; no routing header is used beyond standard IP headers, and, to ensure scalable emulation, route lookup needs to be efficient.

Thus, we present the design and implementation of techniques that enable ModelNet to accurately select the shortest path between a pair of hosts without needing to store $O(n^2)$ routing table state. While our implementation and evaluation is specific to ModelNet, we believe that our techniques are applicable to a broad range of network simulation and emulation environments. Essentially, our approach is to trade additional per-packet computation for reduced memory overhead. We store k spanning trees across a target large-scale topology. For each packet, we determine the path from the specified source to the specified destination in each of the k trees, choosing the shortest.

In this paper, we describe: i) techniques to appropriately choose both the number of trees and the roots of the trees to balance lookup costs and space savings, ii) the use of a small negative cache to maintain shortest path routing for the source destination pairs that do not result in shortest paths based on the spanning tree calculations, iii) a simulation-based evaluation of the potential memory savings for a variety of both realistic Internet topologies and synthetically generated power-law networks, iv) a positive route cache to eliminate virtually all of the computational overhead associated with route lookup in the common case, and v) a complete integration of our proposed techniques into the ModelNet emulation environment with a quantification of the associated performance degradation that results from slower route lookup operations. Overall, our techniques enable ModelNet to scale to tens of thousands of end hosts from a routing perspective, effectively pushing scalability concerns to other system components.

2. Approach

Our approach uses two characteristics of network emulation as points of leverage. First, a network emulator, unlike a real network, stores all information for routes centrally, giving us global information of all emulated nodes and their adjoining connections. Second, Internet-like networks are typically not dense, in the sense that the Internet is structured without separate links from each node to all other nodes. The intuition is that if we generate a spanning tree for such graph, a relatively large percentage of shortest paths between vertices will be contained within the tree. As we create more spanning trees for the graph, each one will contain a different subset of the shortest paths between vertices. Thus, as the number of spanning trees increases, so does fraction of shortest paths that are in the set. We also compute a negative cache of the shortest paths that are not included in any of the spanning trees.

For a path lookup between any two vertices we can guarantee that it will be found either in the tree or the negative cache. This can potentially result in vast improvement: As long as a sufficiently small number of trees can cover a sufficiently large percentage of shortest paths, we can eliminate the need to record $O(n^2)$ shortest-paths (one for each source destination pair in a directed graph). Also, since all of this information is static and available before run-time it can be precomputed. At run-time, we first check the negative cache for the shortest path. If not present, we calculate the path between source and destination in each of k spanning trees, choosing the shortest. The negative cache ensures the resulting route is in fact the shortest available.

Available space savings depends on how well our networks are actually connected, how many trees we choose, and how we choose them. For networks with the same number of vertices, as the average node degree increases, we expect our savings to decrease since the spanning trees will contain smaller percentages of the shortest paths. If the number of vertices in the network increases, we can pick more spanning trees and still save the same proportion of space. For any topology if we pick more trees, then space savings will increase until the space being consumed by each new tree outweighs the savings we get from a smaller negative cache. The two extremes are if we pick zero trees or n trees. Using zero trees results in a negative cache that is exactly the same as all-pairs-shortest-path; n trees could could potentially remove the need for the cache, but the size of our trees would be $O(n^2)$.

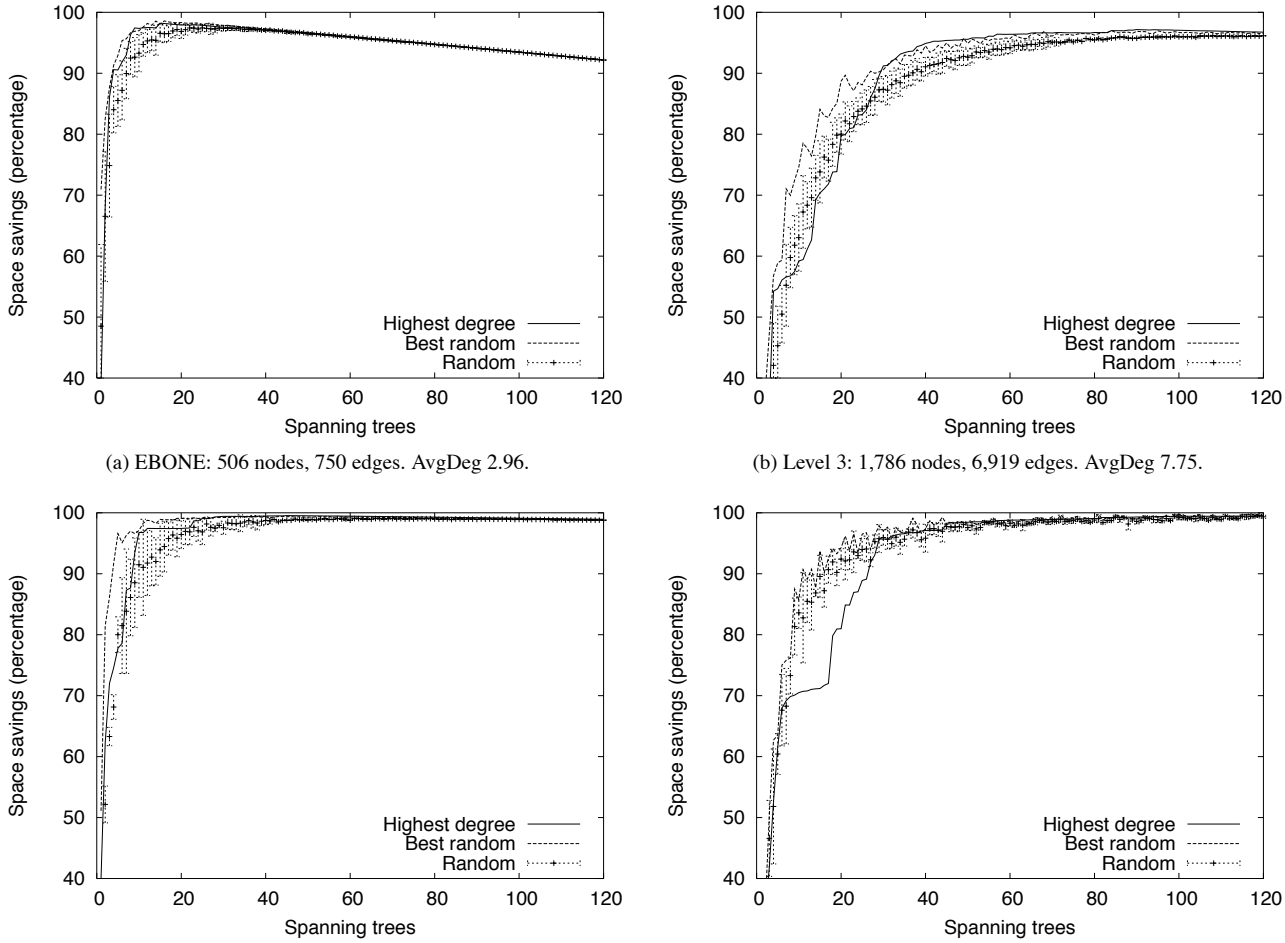
2.1. Implementation

There are two key issues when considering implementing this approach: how many trees should be generated, and where should they be rooted. While we have not yet formulated a concrete answer to the first question, we study the performance of varying numbers of trees in the following sections, and prescribe a simple approach for determining an appropriate number of trees at runtime. Intuitively, short, fat spanning trees would incorporate a larger percentage of shortest paths within them, so trees should be rooted at nodes of high degree. In order to validate this assumption, we implemented both this degree-based approach and a simple random selection, that picks a new, distinct root for each new tree uniformly at random. Our spanning trees were generated using a simple breadth first search algorithm [2, Chap. 22] to minimize their height. Then, iterating through each node, we use BFS again to obtain the single-source shortest paths from the current node to every other node. We check to see if the shortest path is contained in any of the spanning trees, and count only paths between nodes that have not been previously checked with the source and destination reversed to avoid redundancy.

The average runtime of the components of our implementation is $O(V + E)$ for initialization, $O(T(V + E))$ for tree generation and $O(V^2(T \log V + (V + E))$ for finding all of the shortest paths paths (necessary in order to construct the negative cache); where V is the number of vertices, E is the number of edges and T is the number of spanning trees chosen. The dominating runtime factor is the search for shortest paths so overall our average runtime is $O(V^2(T \log V + (V + E))$. The storage requirement is $O(T \cdot V)$ for the trees, and $O(D)$ for the deltas where D is the total size of the negative cache.

3. Validation

Since the efficacy of our approach depends on the characteristics of the target network, we begin by considering



(a) EBONE: 506 nodes, 750 edges. AvgDeg 2.96.

(b) Level 3: 1,786 nodes, 6,919 edges. AvgDeg 7.75.

(c) Telestra: 3,515 nodes, 4,325 edges. AvgDeg 2.46.

(d) AT&T: 11,745 nodes, 14,264 edges.. AvgDeg 2.42.

Figure 1. Space savings as a function of the number of spanning trees. For each topology, we plot the results of two different methods of selecting spanning tree roots: deterministically in order of node degree (starting with the highest), and randomly. For the random case, we plot both the average of 31 runs (with error bars showing the standard deviation) and the best set of trees encountered. Due to time constraints, the AT&T results are currently the average of only four runs.

the potential space savings for actual Internet AS topologies. We compare the memory requirements of our tree-based approach against a naive all-pairs shortest path routing table similar to that currently implemented in ModelNet. We consider the length of each path to be the number of nodes along the path including the source and destination. The negative cache size is then the sum of the lengths of the shortest paths not included in any of the trees.

To calculate the total amount of space consumed, we add the size of all the spanning trees to the size of our negative cache. The size of the routing table without modifications is computed as the sum of the lengths of the all-pairs shortest paths divided by two. Finally, we determine the percentage of space saved by dividing the difference between the original size of the routing table and the size of our spanning-tree based implementation by the size of the original table.

3.1. ISP topologies

Figure 1 presents the results for four representative autonomous systems (ASes): EBONE (AS#1755), Level 3 (3356), Telestra (1221), and AT&T (7018). We do not have access to actual AS topologies; instead, we used the topologies published by the RocketFuel [8] project.[1] Space savings initially increases quickly with the number of spanning

1 In cases when the published RocketFuel topology was not fully connected, we considered only the largest connected sub-component.

trees used; the percentage savings then slows and eventually falls off gradually. This is expected since, initially, each additional spanning tree is contributing a large number of new shortest paths. This effect slows as we find fewer new shortest paths with each new spanning tree. Finally, the savings begins to decline at a certain threshold when the number of new shortest paths gained by adding a tree is outweighed by the storage size of the additional tree.

Focusing on the individual topologies, the first, EBONE, is on the smaller side of the topologies we looked at. The space savings rises rapidly, and then, compared to the others, tails off markedly. It is interesting to note that the standard deviation of the percentage efficiency is much larger than for the other, larger networks. The second graph, Level 3, is in the middle range of size, but it stood out because of its high average degree (Level 3 uses MPLS to increase the perceived IP-level connectivity of its backbone). The net effect is a slightly slower rise in the percentage of space saved peaking at a lower point than the other graphs of the same number of nodes. Interestingly, choosing the highest degree nodes as roots of the tree performs better than the average random root choices. Unlike the other topologies the two methods do not converge as the number of trees increases. AT&T was the largest of our available topologies. It behaves much as would be expected: the graph peaks at a higher percentage space saved and falls off very slowly. The Telestra topology is a good example of an ideal graph that behaves similarly to the synthetic topologies below.

3.2. Generated topologies

Next, we consider how performance varies as a function of graph topology. Clearly, it is possible to construct pathological topologies that would achieve little to no space savings, but the target topologies for this algorithm are ones relating to the Internet and other "naturally occurring" networks. In order to better understand the sensitivity of our approach to various topology parameters, we used a synthetic topology generator, BRITE [6]. We generate topologies with 100 to 1,000 nodes in increments of 100. In an attempt to accurately model real Internet AS topologies, we employed the ASBarabasi model [1] with an average degree of four. We ran 31 distinct iterations, each with a different set of random roots, and one additional time using only highest degree roots.

3.2.1. Graph size. We begin by considering the effect of graph size. Figure 2 shows the average maximum space savings achievable by our algorithm using random node selection, for an optimal number of trees. To calculate this value, we first generate a topology of the indicated size. Then, we generate 31 distinct sets of n random trees, for $1 \leq n \leq 100$. We find the maximum percentage space saved in each of the 31 iterations and average them together

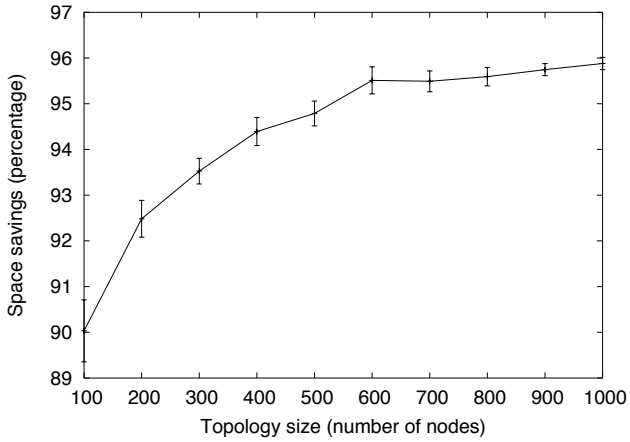


Figure 2. Space savings as a function of graph size. Topologies generated using the Barabasi model with an average node degree four. Error bars are one standard deviation.

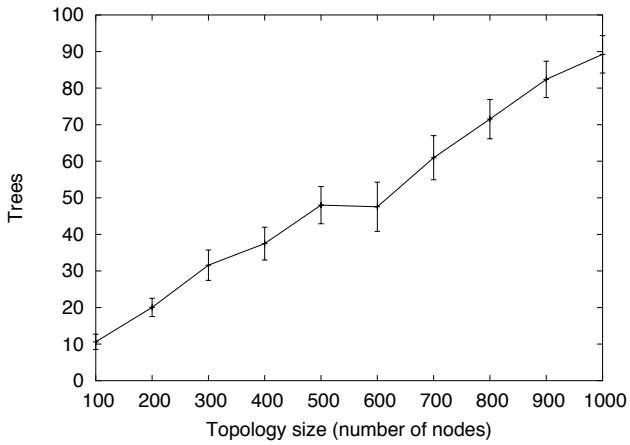


Figure 3. Minimum number of trees necessary to achieve the space savings reported in Figure 2.

and calculate standard deviation. The efficiency of the algorithm increases as the size of the topology increases, since the marginal benefit of having spanning trees and negative caches increase with the size of the topology.

Figure 3 expands on the previous result, showing the number of trees necessary to achieve the space savings shown in Figure 2. We determine this value by inspection: For each iteration, We identify the maximum space savings and then find the minimum number of trees that result in space savings within 1% of highest achieved for that iteration. The overall pattern is clear: the larger the graph, the more trees that are necessary.

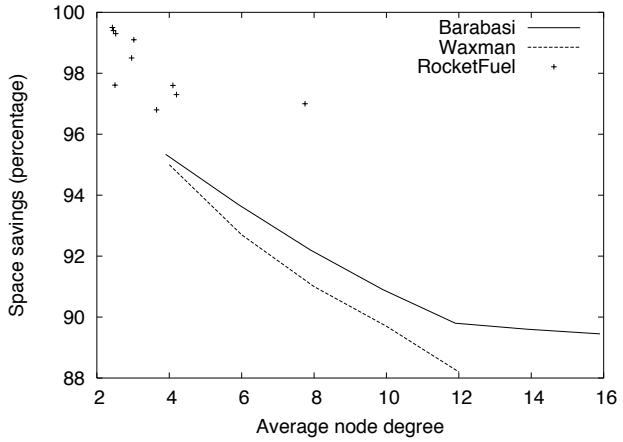


Figure 4. Space savings as a function of average node degree. Topology size is fixed at 1,000 nodes for Barabasi and Waxman, but varies between 226 and 11,745 nodes for the RocketFuel topologies.

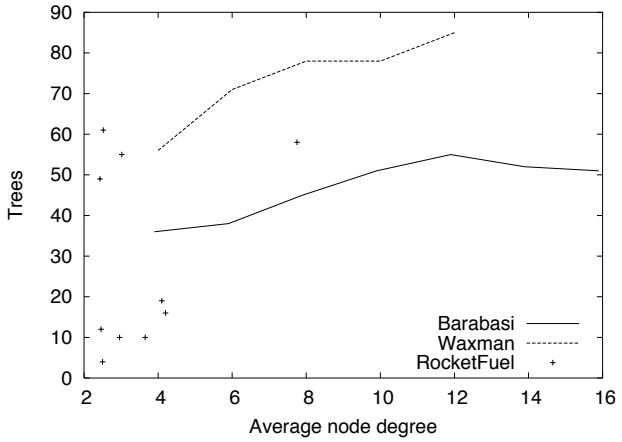


Figure 5. The number of spanning trees necessary to achieve the savings reported in Figure 4.

3.2.2. Node degree. Intuitively, the effectiveness of our algorithm depends on the connectivity of the graph. One concise metric of graph connectivity is average node degree. Here, we generate topologies of fixed size, but with varying node degree. In an attempt to distinguish between the effects of average degree and the particular degree distribution, we report results for two graph types: Barabasi, as before, and Waxman [12]. For context, we also include the various RocketFuel topologies, but they cannot be directly compared due to their varying size.

Figure 4 plots these results, varying the average degree from 4 to 16 while keeping the number of nodes fixed at 1,000 and using the Barabasi model to generate the topologies. Again, we ran 31 random iterations up to 200 trees and plot the largest savings observed for each degree. The maximum space savings is inversely proportional to the average degree. However, Figure 5 shows that the number of trees necessary to attain the reported savings increases with average degree. (The method used to compute the number of trees necessary is the same as in Figure 3.) This is expected; as the average degree rises, there is an increasing number of shortest paths that share fewer edges, thus the set of spanning trees is less likely to cover as many shortest paths as the number of edges in any spanning tree is fixed (at $n-1$) and does not increase with node degree.

3.2.3. Degree distribution. As can be seen by comparing Figure 6 with Figure 1, there are two significant differences between the Barabasi/RocketFuel and the Waxman topologies. First, in comparison to the random root choice, the highest average degree algorithm performs slightly better

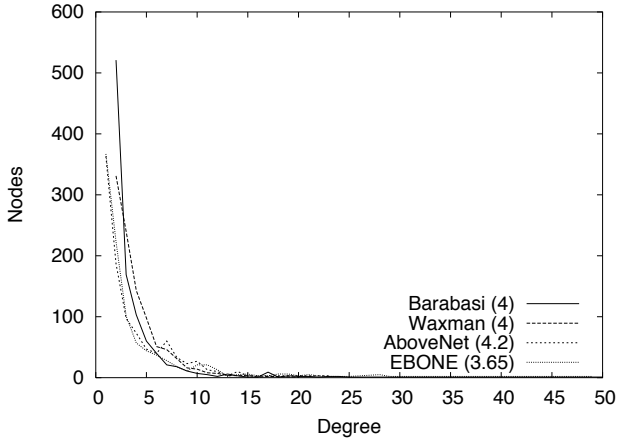
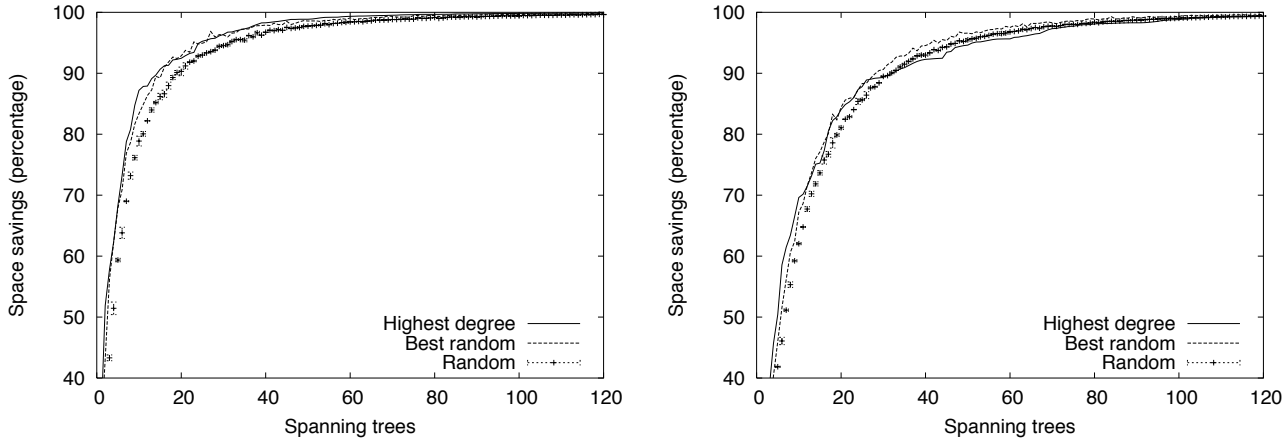


Figure 7. Comparison of node degree distributions between generated and RocketFuel topologies (normalized to 1,000 nodes).

for RocketFuel, nearly as well for Barabasi and much more poorly in the Waxman. This difference in performance of using highest degree roots is due to the difference in distribution of node degree (as shown in Figure 7). The Barabasi and nearly all of the RocketFuel topologies exhibit power-law properties for the frequency of node degree versus degree. However, the Barabasi models node degree distribution is shifted on both ends toward the average degree resulting in a non-hierarchical and more densely connected clusters of nodes in the graph. This causes the intuitive advantages of choosing root nodes of highest degree to generate shorter trees to be weakened. The second difference is that, for the Waxman model, the space savings increases



(a) Barabasi: 1,000 nodes, 1997 edges. AvgDeg 4.0.

(b) Waxman: 1,000 nodes, 2000 edges. AvgDeg 4.0.

Figure 6. Space savings as a function of the number of spanning trees for synthetic graphs.

at a significantly slower rate. This is also due to the difference in distribution of node degree; the decrease in the proportion of nodes in the network with one or two edges causes the spanning trees we generate to be less likely to contain the shortest path between the root and the leaves of the tree. Lastly, an artifact of the BRITE topology generation and conversion is that each topology originally contained directed links. Converting to undirected graph caused the minimum degree to increase to two for each node. This means the BRITE generated results were actually slightly more pessimistic than if we had been able to distribute edges in a manner similar to the RocketFuel topologies.

Though the percentage space saved in the Waxman model is worse than those in Barabasi/RocketFuel models, the space saved still peaks at similar values for comparable average degrees. Therefore, only the rate of increase of space saved versus number of spanning trees chosen is affected by the unfavorable distributions of the Waxman model topologies.

4. Performance

To investigate how our approach performs in a real Internet scale emulator, we integrated our route lookup techniques into ModelNet. We perform benchmarks here for actual space savings and lookup degradation versus the unmodified ModelNet route lookup scheme.

4.1. ModelNet integration

ModelNet was designed to be modular; as a consequence, the route lookup functionality is cleanly separated from the rest of the system. This enables easy integration and effectively "drop-in" replacement with alternate routing schemes. The process is straightforward: We simply wrote a new lookup function which uses the spanning trees and negative caches for route lookups instead of the original n^2 routing table. The spanning trees are computed offline as a pre-processing stage and stored in a text file. This text file is read into the kernel at module-load time and the data structures are initialized using this file.

We measured the lookup cost by averaging over a large number of random lookups. However, this is highly unlikely to occur in practice. A real experiment will seldom send packets at random; on the contrary, most experiments typically consist of packet flows. This led us to implement a small *positive cache* to speed up frequently accessed routes. Since the amount of memory devoted to the cache is configurable, it does need not impact the memory savings. The cache is associative: each cache line is a linked list of pointers to routes that have been computed earlier. The maximum sizes of both the cache and the individual lines can be configured. The worst case lookup involves a constant time indexing into the array, and $O(line\ size)$ traversal of the linked list. Our current hash implementation is naive, and open to optimizations in the data structure and choice of hash function. Despite this, it provides significant performance benefits.

For our experiments, we use the same ISP topologies as in the simulations. However, since ModelNet has slightly different semantics and notation for describing topologies, there are minor differences:

- ModelNet can assign "roles" to nodes in the topology (transit node, stub node, or client node). The ModelNet routing table is indexed only by the client nodes (also called virtual nodes). Since our experiments perform routing lookups on arbitrary source/destination pairs, we augmented the original topology by attaching a client node to each node in the ISP topology.

- Our simulations are on on undirected graphs. However, ModelNet uses bi-directional edges to emulate asymmetrical links. We therefore augment our original topologies with bi-directional links.

The end result of these modifications is that the topologies used by ModelNet are twice as large as those used in simulation. We account for this difference when analyzing our results in the following subsections.

We use a heuristic to determine the number of trees to generate for a given target topology. The number of trees, N, is given by:

$$N = 2 \times 0.418325 \times n^{0.61535}$$

where n is the number of vertices in the topology. To arrive at this formula, we took the data from our analysis of topologies of various sizes (from 100 to 5,000 nodes) in the previous section, and determined the minimum number of trees needed to reach within 1% of the maximum space savings possible. We then computed a best-fit curve using regression analysis, which gave us the constants 0.418325 and 0.61535. The additional factor of two ensures that we reach the peak of our space savings curve. We do not consider the average node degree since its influence is small compared to the topology size and distribution.

4.2. Results

4.2.1. Space savings. To measure space savings, we instrumented ModelNet by defining separate kernel memory heaps for MTree and ModelNet's original routing table. After the data structures have been created and initialized, we simply record the memory allocated by these heaps. Note that this is not entirely accurate since there is some granularity in memory allocation involved—for instance, when asking to allocate memory for five integers, the heap might receive an entire page instead (making subsequent allocations faster).

Figure 8 shows the results of this experiment. For each ISP topology, we use a fixed number of trees and compare it with the predicted savings (from simulations) using the same number of trees. Note that the predicted savings take into account the inflation in topology size inside ModelNet as described earlier. The actual savings closely match the predicted values. The minor extra memory consumption in the actual case stems from the use of some auxiliary data structures used in the implementation and the granularity of memory measurement.

4.2.2. Pre-computation time. Table 1 shows the time required to compute the routing tables used in Figure 8 on a 2.8-GHz Intel Pentium 4, as well as several larger synthetic, mesh-like topologies used in later experiments. Due to the regular, well-connected nature of the mesh topologies, constructing trees and deltas is relatively faster.

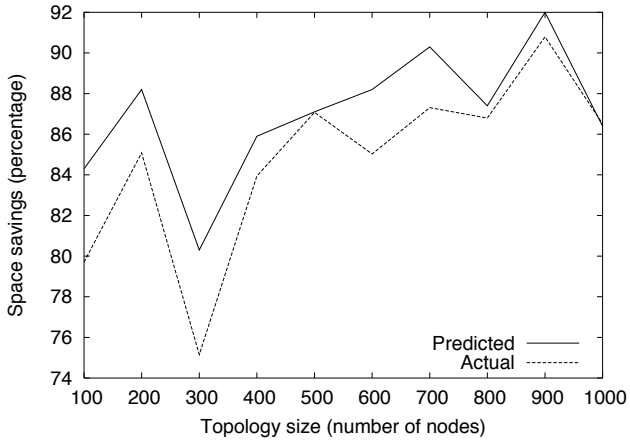


Figure 8. Space savings as a function of graph size. Each topology is generated using the Barabasi model with an average node degree of four. No positive cache was used.

Num. nodes	Num. trees	Time (seconds)
100	22	1
200	26	1
300	38	6
400	40	7
500	50	12
1,000	72	79
2,000	100	552
3,000	177	2,066
5,000	242	9,811
10,400	248	4,186
15,600	318	9,345
22,320	397	11,733

Table 1. Pre-computation time for a subset of the Barabasi topologies used in Figures 2 and 3. Sizes 10,400 and larger are generated using artificial mesh-like topologies.

4.2.3. Lookup cost. Here we evaluate the degradation in the *per-route* lookup time using the MTree approach as compared to ModelNet's original route lookup. For our experiments, we performed 10,000 random route lookups for each topology. Figure 9 shows the results on the Barabasi topologies use for the previous experiments. ModelNet's original route lookup cost is independent of the topology size since it is simply the cost of a function call that returns a memory reference. For MTree, the lookup cost increases as the size of the topology increases. This is expected since the lookup now has to go through larger trees.

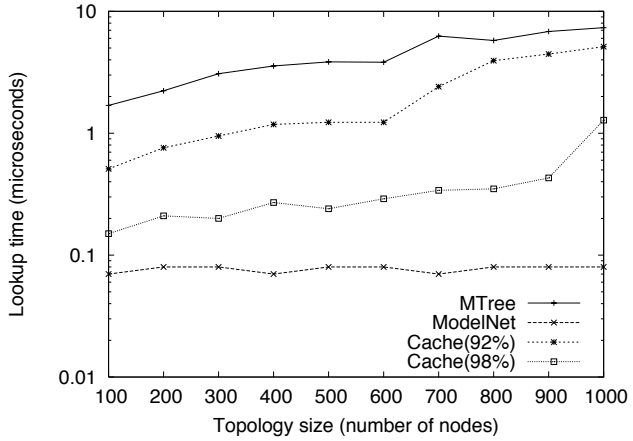


Figure 9. ModelNet route lookup latency. Note the log scale on the Y axis. Barabasi topologies were used. The positive cache was restricted to 1,000 routes (line size 10).

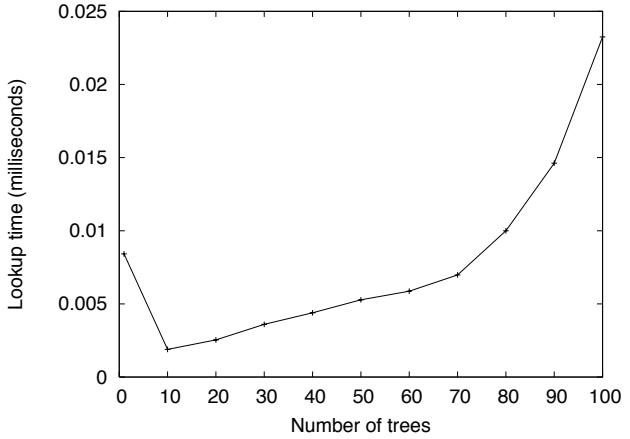


Figure 10. Lookup cost as a function of number of trees for a 200-node Barabasi graph.

While the lookup cost for MTree is several orders of magnitude higher than that of the original ModelNet, there is significant performance improvement with the introduction of the positive cache. At high hit rates (we plot 92% and 98%), the lookup cost comes down significantly, since most of the routes are available in the cache.

We will see in the following section that with the positive cache in place, the emulation time still dominates the lookup cost; there is no significant degradation in the end-to-end performance. Previously, ModelNet's scalability was limited by the size of the routing table that would fit into the memory. For instance, ModelNet was not able to load the routing tables for a 2,000-node Barabasi topology in our setup. On the other hand, MTree is able to load an artificial, mesh-like topology with 22,320 nodes and 84,902 edges. This topology requires 108 MB of memory (including a 1,000-entry positive cache) for the routing data, resulting in 0.56-ms lookup times (assuming a 92% hit rate).

The lookup cost also varies with the number of trees used and the size of the resulting negative cache. Figure 10 shows the variation of lookup cost for a 200-node Barabasi topology as a function of the number of trees. With a single tree, the size of the negative cache is huge, making the negative cache almost as big as an n^2 routing table. As the number of trees increases to ten, the lookup cost falls sharply since a significant number of routes are now included in the trees. On further increase in the number of trees, the lookup cost increases slowly at first and rises sharply as the number of trees becomes very large.

4.2.4. End-to-end performance. This section investigates the degradation in end-to-end performance in

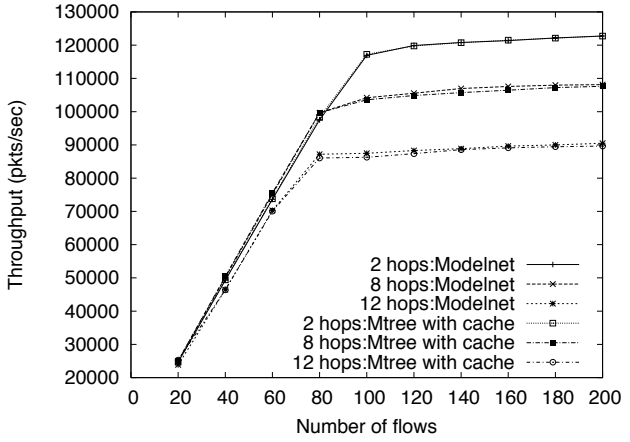


Figure 11. Capacity of ModelNet core: Unmodified ModelNet vs. MTree. Artificially generated grid like topology of size $maxnumflows \times numhops$.

ModelNet resulting from the increased lookup costs of MTree. Our experiment consisted of running concurrent independent flows and measuring the throughput delivered by the system. We used grid-like topologies for this experiment—the "height" is the number of flows, and the "width" is the number of hops. We manipulate the load on the core by varying the number of hops, as each hop requires additional processing.

The flows correspond to a "row" in the grid, with one end point being the source and the other being the sink. These flows are *parallel* in the sense that the hops in each path are mutually exclusive; therefore, along any path, there is no

contention for bandwidth. The access links (the hops coming out of the sources and going into the sinks) are restricted to 10 Mbps; all the other hops in the topology have 1 Gbps of bandwidth. In the ideal case, therefore, each flow should be getting a bandwidth of 10 Mbps with no packet drops.

Intuitively, we would expect the system to deliver increasing throughput with increasing number of flows, until the system saturates. Beyond this point, the system should deliver a sustained constant throughput irrespective of the number of flows. Further, we would expect that the system performance will decrease with increasing number of hops. Of course, the critical issue here is the performance deterioration incurred as a result of using the MTree approach instead of the usual n^2 routing table. For this experiment, we were using a 100-line cache, with a line size of 10.

Figure 11 shows the results of this experiment. To measure system throughput, we instrumented the core to record the number of packets going through it every second. So, for each experiment we gathered a "time-line" data, and computed the 95th percentile of the packets/sec data, which was then plotted here. The flows were run for sufficiently long periods (30 seconds) to make sure that we avoided any boundary effects (flows starting up or dying down) and to give us a comfortably long measurement window where all flows were indeed running in parallel.

There is no distinguishable performance difference between the unmodified ModelNet and the MTree approach. Thus, putting a small positive cache in front of the spanning tree lookup can give tremendous savings, especially for long flows. The system also degrades gracefully under load, maintaining a sustained throughput of around 90,000 packets/sec at peak load (200 flows, 12 hops).

5. Conclusion and future work

This paper presents an approach to routing table calculation and storage based on multiple spanning trees rooted at various nodes in a target topology. We find that this results in an order of magnitude reduction in routing table size for Internet-like topologies. Off-line, we also populate a negative cache of shortest paths that do not appear in any of the trees. We integrated our approach into ModelNet, an Internet-scale network emulator, and demonstrated that the actual space savings closely match the values predicted in simulations. We also investigated the increase in lookup and its affect on end to end performance. Using a small positive cache, we were able to demonstrate that the performance of MTree is commensurate with that of unmodified ModelNet for flow based experiments. It is likely that the performance can be improved further with clever data structures and better choice of hash functions.

Work on so-called compact routing [9] has developed extremely space-efficient routing mechanisms for many classes of networks. These schemes typically do not compute optimal, shortest paths; instead, they compute routes that are within a factor k of optimal, referred to as a *stretch* factor of k. Due to our need for shortest, or stretch-1, paths, these approaches did not initially seem applicable. Recent work has shown that for Internet-like topologies, however, compact-routing schemes often produce stretch-1 paths, and the average stretch factor of all paths computed using compact routing on Internet-like topologies is close to one [4]. Hence, we are interested in exploring the applicability of these schemes to the ModelNet environment.

Acknowledgments

The authors would like to thank David Becker, John Byers, Priya Mahadevan and Ken Yocum for valuable insights and comments. Special thanks to Marvin McNett for help with the experimental setup and cluster administration.

References

[1] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, Oct. 1999.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Prentice Hall, 1990.

[3] C. Gavoille and S. Pérennès. Memory requirements for routing in distributed networks. In *Proc. ACM PODC*, pages 125–133, May 1996.

[4] D. Krioukov, K. Fall, and X. Yang. Compact routing on Internet-Like graphs. In *Proc. IEEE INFOCOM*, Mar. 2004.

[5] X. Liu and A. A. Chien. Realistic large-scale online network simulation. In *Proc. ACM Supercomputing*, Nov. 2004.

[6] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An approach to universal topology generation. In *Proc. MASCOTS*, Aug. 2001.

[7] D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, July 1989.

[8] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, pages 133–145, Aug. 2002.

[9] M. Thorup and U. Zwick. Compact routing schemes. In *Proc. ACM SPAA*, pages 183–192, 2001.

[10] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proc. USENIX OSDI*, Dec. 2002.

[11] K. Walsh and E. G. Sirer. Staged simulation: A general technique for improving simulation scale and performance. *ACM TMACS*, 14(2):170–195, Apr. 2004.

[12] B. M. Waxman. Routing of multipoint connections. *IEEE J-SAC*, 6(9):1617–1622, Dec. 1998.

[13] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. USENIX OSDI*, pages 255–270, Dec. 2002.