Automatic IP Address Assignment on Network Topologies

Jonathon Duerig[†] Robert Ricci[†] John Byers[‡] Jay Lepreau[†]
[†]University of Utah {duerig,ricci,lepreau}@cs.utah.edu
[‡]Boston University byers@cs.bu.edu

Flux Technical Note FTN–2006–02 February 11, 2006

Abstract

We consider the problem of automating the assignment of IP addresses to nodes in a network. An effective assignment exploits the natural hierarchy in the network and assigns addresses in such a way as to minimize the sizes of routing tables on the nodes. Automated IP address assignment benefits simulators and emulators, where scale precludes manual assignment, large routing tables can limit network size, and realism can matter. It benefits enterprise networks, where large routing tables can overburden the legacy routers frequently found in such networks.

We formalize the problem and point to several practical considerations that distinguish our problem from related theoretical work. We then describe several of the algorithmic directions and metrics we have explored, some based on previous graph partitioning work and others based on our own methods. We present a comparative assessment of our algorithms on a variety of real and automatically generated router-level Internet topologies. Our two best algorithms, yielding the highest quality namings, can assign addresses to networks of 5000 routers, comparable to today's largest single-owner networks, in 2.4 and 58 seconds.

1 Introduction

Minimizing the size of routing tables on network hosts and routers is a basic problem in networking. For each possible destination, a routing table must store an entry that specifies the first hop to that destination from the current node. Without route aggregation, each of the n nodes in a network would need to store a routing table entry for all $n-1$ routes. By defining an aggregation scheme that allows multiple routes with the same first hop to be combined, the size of these routing tables can be reduced by orders of magnitude. A good example of this is Classless Inter-Domain Routing (CIDR), the routing scheme used in the Internet today. In CIDR, a route for an entire IP prefix can be specified with a single table entry.

In most Internet settings, names, in the form of IP addresses, have already been assigned to hosts, so once routes are computed, minimizing the size of a routing table amounts to compressing the table to correctly specify routes with a minimum of table entries. However, in several other important settings, addresses have not been assigned in advance. This gives us the opportunity to assign addresses in such a way as to minimize the ultimate size of routing tables. Simulation and emulation, for example, typically use generated topologies which do not come annotated with address assignments. Some overlay networks choose to use a virtual IP address space to name their members, and there are increasingly more enabling technologies [34, 28] and reasons [2] for deploying such virtualized networks. Occasionally, even operators of real networks re-design their address assignment scheme. In fact, one of the authors' institutions recently completed a project to re-assign addresses to their entire 20,000+ node network because the old scheme had led to unacceptably large routing tables. Many enterprises use memory-constrained legacy routers [29], which can be overburdened by routing tables due to poor address assignment. In all of these cases, optimized address assignment ultimately leads to more efficient networks.

Fundamentally, a "good" address assignment is one that reflects the underlying hierarchy of the network. Because CIDR itself is inherently hierarchical, with more specific prefixes overriding more general ones, an address assignment that exploits the hierarchy in the topology leads to smaller routing tables. A significant caveat is that real topologies are not strictly hierarchical, and thus the computational challenges of identifying a suitable hierarchical embedding of the topology come to the fore. This direction—inferring hierarchy in this practical setting—is the focus of our work.

Given a topology, we seek to produce an IP address assignment that minimizes the sizes of the routing tables on the nodes in the network. Underlying this problem are elegant graph-theoretical questions, such as compactness of a graph, and the degree to which vertices are equivalent, from the perspective of routing, as viewed by other vertices. Our work connects to these theoretical questions, but stems from a problem formulation that captures the complexities and nuances of assigning IP addresses to network interfaces in an environment where CIDR aggregation is used.

The areas in which our work is of primary importance are network simulation and emulation. Network simulators such as *ns* [5] typically emphasize certain details of network modeling, such as topology and queuing, and abstract away others. Specifically, many use an abstract notion of node identity, and not real IP addresses and CIDR routing. Largely as a result of this, work on topology generation has concentrated on producing representative topologies, but not on represen-

tative address assignment. Current popular topology generators, such as GT-ITM [13] and BRITE [26], do not annotate their topologies with addresses. Lack of work on address assignment, in turn, discourages new simulators from using IP addresses. Rather than address this shortfall by modifying a particular topology generator, we instead aim to provide general algorithms that can assign IP addresses to any router-level topology, regardless of its source or the model used to generate it.

Recently, network emulation environments such as Emulab [39] and Modelnet [36] have become popular. These environments create topologies using real nodes and network hardware. They emulate topologies similar to the ones used by simulators; in fact, the same topology generators can be used for both. Since they run real IP stacks, rather than simulations of stacks, IP addresses are a requirement. Currently, address assignment is typically done manually or in an ad hoc manner. Assigning addresses manually is tedious and surprisingly error-prone. Ad hoc schemes, such as random assignment, may produce consistent addresses, but they do not reflect the factors that go into assignment on real networks.

Providing more sensibly-structured IP addresses is a requirement for accurate experimental evaluation in areas such as dynamic routing algorithms, firewall design and placement, and worm propagation. At a more basic level, most simulators are fundamentally unrealistic in that they name network nodes instead of network interfaces. In some cases the distinction can be important in specification and evaluation, as related work has found [35, 7].

Address assignments in real networks are influenced by the hierarchy present in the topology of the network, and the policies and organic growth of the organizations that own them. It is not clear to what extent policy can be modeled, given that is often ad-hoc and organization-dependent. There is other work [1] that attempts to model the ways in which networks grow. Thus, we concentrate on the hierarchical properties of networks, attempting to assign addresses that, by minimizing routing table sizes, match the natural hierarchy of the network.

This paper makes the following contributions.

- We build upon a theoretical formulation of interval routing to formulate the IP address assignment problem, and believe we help to open this general area of study.
- We devise a general metric, "Routing Equivalence Sets," that quantifies the extent to which routes to sets of destinations can be aggregated.
- We develop three classes of algorithms to optimize IP naming, each with a fundamentally different approach to the problem.
- We devise a pre-processing step that improves the running times of several of our algorithms by orders of magnitude without sacrificing solution quality.
- We implement the algorithms and evaluate them on a number of topologies. We find two of them, recursive partitioning and tournament RES, to be particularly effective and efficient enough to run on topologies as large as today's largest single-owner networks [3], in a few seconds and a minute, respectively.

The rest of the paper is organized as follows. In the next section we start by defining a clean theoretical version of the problem, then outline some of the practical issues that complicate it. Section 3 details the algorithms we have developed, while Section 4 evaluates their effectiveness. Finally, we discuss related work, and conclude.

2 Problem Statement

This work seeks to produce a global address assignment automatically, i.e., an assignment in which IP addresses are assigned to each network interface in a network. Our primary aim in computing an assignment is to minimize the total space consumption of routing tables, which in turn helps to minimize packet processing time at routers. In practice, IP address assignment directly impacts the sizes of routing tables, since a set of destinations with contiguous IP addresses that share the same first hop can be captured as a single routing table entry. It is also essential to name hosts from a compact namespace, as the available address space is typically sharply limited. As we later discuss in detail, it is also important to consider the running time of an assignment algorithm in evaluating its effectiveness. While our work explicitly focuses on shortest-path intradomain routing, another possible consideration is that of producing a naming scheme that yields routes with small stretch, i.e., routes used are at worst a constant stretch factor longer than shortest-path routes. We formulate our assignment problem first using the clean conceptual notion of *interval routing*, widely used in theoretical studies, and then describe the additional constraints that CIDR prefixes and CIDR aggregation impose on the problem.

As an example of interval routing, consider the network depicted in Figure 1, in which nodes are assigned addresses from $\{1, \ldots, 7\}$. Interval routing table entries are shown in Figure 2 for the outbound interfaces of nodes 1, 2, and 7. Node 7 can express its shortest-path routes with two disjoint intervals, one per interface, which corresponds to a routing table of size two. With the given address assignment, node 1 must use a total of three disjoint intervals to exactly specify the routes on its outbound interfaces. Note that in this example, ties between shortest-path routes can be exploited to minimize routing table size. For example, the routing table at node 7 elected to group node 3 on the same interface as nodes 1, 2, and 4 to save two table entries.

For a formal definition of interval routing, consider an n-node undirected graph $G = (V, E)$, where we will refer to vertices as hosts, and an edge (u, v) as a pair of interfaces (one at vertex u and one at vertex v). An address assign-

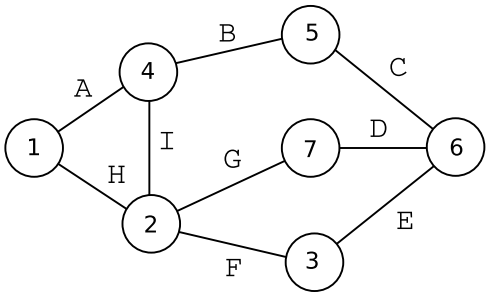

Figure 1: A 7-node network. Vertices are labeled with numbers and edges are labeled with letters.

Node 1		Node 2		Node 7	
Range	First Hop	Range	First Hop	Range	First Hop
2–3	H	1	H	1–4	G
4–6	A	3	F	5–6	D
7	H	4–5	I		
		6	F		
		7	G		

Figure 2: Selected interval routing tables from the above network. The first hop is designated by the label of the first edge to traverse.

ment $\mathcal{A}$ assigns each vertex in V a unique label from the namespace of integers $\{1, \ldots n\}$. The routing table of vertex u associates each label of every vertex with one edge (u, v) (the next hop). In this manner, a subset of labels in $\mathcal{A}$ is associated with each edge. Interval routing compacts the routing table by expressing each of these subsets of labels in $\mathcal{A}$ as a set of intervals of integers.

On a node using interval routing, the size of the minimal set of intervals is the routing table size, or the *compactness* of the routing table. We denote the number of entries in the routing table of vertex u by k_u. The theory literature has considered questions such as determining the minimum value of k for which an assignment results in routing tables all of size smaller than k [37, 18, 15]. For a given graph, this value of k is defined to be the *compactness* of the graph. We are primarily concerned with the *average* routing table size, so we work with the following objective function:

Objective: For a graph G, generate an address assignment $\mathcal{A}$ that minimizes $\sum_{u \in V} k_u$.

It is well known that search and decision problems of this form are NP-complete, and several heuristics and approximation algorithms are known [18]. Our focus is on the practical considerations that cause CIDR routing to be a significantly different problem than interval routing; we discuss these differences and our approach next.

Practical Considerations

There are three main differences between the theoretical approach to compact addressing that we have described so far and the actual addressing problem that must be solved in emulation and simulation environments. First, although interval routing is intuitively appealing and elegant, routing table aggregation in practice is performed using the set of classless inter-domain routing (CIDR) rules [17], adding significant complexity. Second, in IP addressing, each individual interface (outbound edge) of a node is assigned an address(label), not each vertex, adding subtleties to the naming process. Finally, widely used local-area network technologies such as Ethernet provide all-to-all connectivity, and these networks are best described by *hypergraphs* [6] (described below), not generic graphs. We next discuss these three practical considerations and their consequences for the problem formulation.

CIDR specifies aggregation rules that change the problem in the following ways. A CIDR address is an aggregate that is specified as a prefix of address bits of arbitrary length. It encompasses all the IP addresses that match that prefix. This implies that a CIDR address can express only those intervals of IP addresses that are a power of two in size and that start on an address that is a multiple of that same power of two. In other words the interval must be of the form $[c * 2^y, (c + 1) * 2^y)$ for integers c and y. This more restrictive aggregation scheme means that an IP assignment must be carefully aligned in order to fully take advantage of aggregation. In practice, dealing with this alignment challenge consumes many bits of the namespace, and *address space exhaustion* becomes an issue even when the number of interfaces is much smaller than the set of available names. A key aspect of our work, covered in Section 3.3, are methods to address the problem of address space exhaustion by compactly orienting blocks of contiguous address space. Note that interval routing runs into no such difficulty. A second difference between interval routing and CIDR aggregation arises because CIDR routing tables accommodate a *longest matching prefix* rule. With longest matching prefix, the intervals delimited by CIDR routing table entries may overlap, but the longest (and consequently most specific) matching interval is used for routing. The potential for use of overlapping intervals is advantageous for CIDR, as it admits more flexibility than basic interval routing.

When IP addresses are assigned, they are assigned to network interfaces, not hosts. For single-homed hosts this is a distinction without a difference, but for hosts with multiple interfaces, such as network routers, this distinction can materially impact address assignment. These multi-homed hosts may be associated with multiple addresses, which complicates the problem. Within a single AS (Autonomous System) using shortest-path routing, when a packet is sent to any one of a host's addresses, it is typically expected to take the shortest path to any interface on the host. Thus, it is valuable to be able to aggregate all addresses assigned to a host. This means that we must not only be concerned with how links aggregate with each other, but also with how the interfaces on a host aggregate as well.

The networks we consider in simulation and emulation environments are best represented as *hypergraphs*, since they often contain local-area networks such as Ethernet, which enable all-pairs connectivity among a set of nodes rather than connectivity between a single pair of nodes. A hypergraph

captures this, since it is a generalized graph in which each edge is associated with a set of vertices rather than a pair of vertices. As before, when assigning addresses to a hypergraph, we wish to assign addresses to network interfaces. With the hypergraph representation, this becomes more difficult to reason about, since each network edge may be associated with a set of vertices of arbitrary size.

For convenience, we work instead with the dual hypergraph [6] (see Figure 4); to find the dual hypergraph of a given topology, we create a hypergraph with vertices that correspond to links in the original topology, and hyperedges that correspond to hosts in the original topology. Each vertex in the dual hypergraph is incident on the edges that represent the corresponding hosts in the original graph. Thus, by labeling vertices of the dual hypergraph, we are labeling the network LANs and links in the original topology. We label the vertices with IP subnets, and then assign an address to each interface from the subnet of its adjacent LAN.

3 Algorithmic Contributions

We now describe our methods for producing a labeling that strives to generate CIDR routing tables of minimum average size, and subject to the additional practical constraints enumerated above.

We first decompose solutions to the problem into three logical steps:

1. Graph Preprocessing: Since the running times of our algorithms is dependent upon the size of the graph, we provide methods to reduce the size of the input topology by identifying and removing subgraphs whose addresses can be assigned optimally using only local information.

2. Trie Embedding: We then embed the vertices of the graph into the leaves of a binary trie, where each internal node represents a logical subnet of its associated leaves and encompasses the interval of IP addresses of its children. This step is the linchpin of our approach, and we contrast several different methods that we have experimented with.

3. Address Compaction: To minimize the impact of address space exhaustion, we devise a post-processing step that reorients the tree to minimize its height.

The methods for the steps above constitute the main technical contributions of this paper. We describe these algorithms in this section, and then in Section 4, we evaluate their effectiveness in practice.

3.1 Graph Preprocessing

Most of the algorithms that we propose for the key step of Trie Embedding have superlinear time complexity in the size of the graph, which limits their scalability on large topologies. To achieve scaling we have devised a prepass phase which meets two goals: (1) identify and remove subgraphs for which locally optimal address assignment is possible, and (2) decompose the remaining input topology into subgraphs to which addresses can be independently assigned.

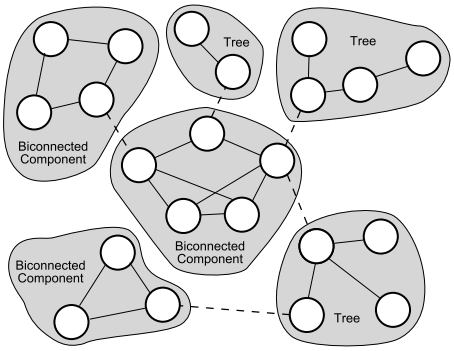

Figure 3: The prepass partitions the graph into trees and biconnected components. Bridges are shown as dashed lines.

To achieve the first goal, we use the fact that there are some structures for which there are simple optimal algorithms for address assignment, like trees. Such structures are relatively common in some types of networks, such as at the periphery of enterprise and campus networks. They are also seen frequently in the synthetic topologies used in simulation and emulation, and thus it is worth optimizing these common cases. To achieve the second goal, we take advantage of the fact that any singly connected component, i.e., a subgraph where removal of a single edge called a *bridge* breaks the component in two, is also amenable to preprocessing. Here, address labelings for the subgraphs on either side of the bridge can be generated independently with a minimal impact on the overall quality of the approximation (discussed below). The property we exploit is this: if each biconnected component [10] is assigned a unique prefix, then the internal assignment of addresses within a component does not change the number of routes of any host outside of that component. By identifying trees and bridge edges (both linear time), the prepass phase naturally decomposes the graph into a set of smaller biconnected components and trees, as shown in Figure 3.

While the preprocessing step has obvious benefits, there are also some less obvious costs. First, there are some technicalities introduced by our need to work with the dual hypergraph. These are discussed in the Appendix. Second, the partitioning performed in the prepass typically leads to some increase in routing table sizes.

For example, suppose the partitioning elects to separate biconnected components A and B by cutting a bridge edge (a, b) for some vertices $a \in A$ and $b \in B$. Our methods will then (naturally) assign a an address in the space assigned to subnet A, allowing all vertices in B to use a single routing table entry to reach all of A. But consider the non-intuitive assignment of giving a an address in subnet B instead. This has the likely effect of complicating routing tables of hosts within B. However, all tables in A stand to gain, as the hosts can route to all of $B \bigcup \{a\}$ with a single entry. It turns out

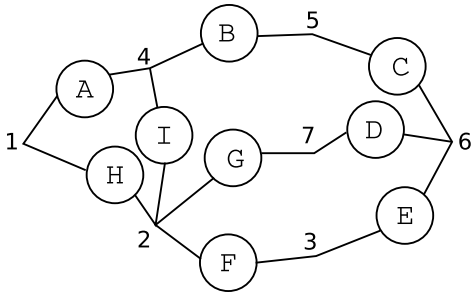

Figure 4: The dual hypergraph of Figure 1. Each host is taken to be an hyperedge and each link is taken to be a vertex. Taking the dual of the original topology is necessary because IP addresses are assigned to interfaces and links rather than hosts.

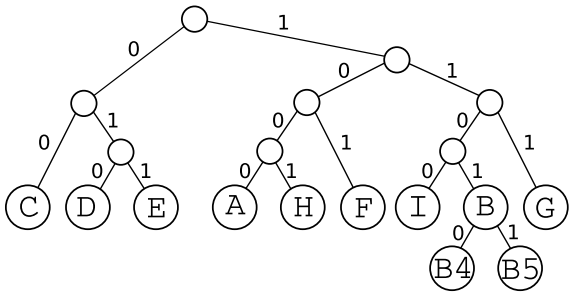

Figure 5: A sample embedding of the vertices in the graph of Figure 4.

Label	Prefi x	First Hop Link
a	*	D
b	1*	G
c	11011	D
Table Size: 3		

Figure 6: Routing table for Host 7 given the embedding in Figure 5. Each label corresponds to a node in the trie (indicated by a box).

that the best choice of address assignment on the cut boundary depends on the relative sizes of A and B and their internal topologies. The assignment we use gives away this edge, thus the prepass comes at some cost. In Section 4.3 we evaluate the tradeoffs that the prepass imposes on routing table sizes for generated and Internet topologies.

3.2 Trie Embedding

We next explore several methods for embedding the vertices of the graph into a binary trie. Some are of our own devising, and some leverage work from the graph partitioning community.

Background We begin with a basic description of the trie, employing the following terminology in the remainder of this section. 'Host' refers to an actual host or router in the original topology. 'Link' refers to a link or LAN in the original topology. 'Nodes' and 'leaves' refer to nodes in the trie. Since addresses need to be assigned to interfaces and therefore to links, the embedding algorithms operate on the dual of the original graph. The dual of the graph in Figure 1 is depicted in Figure 4. A dual 'vertex' refers to a link in the original graph, while a dual 'edge' is a host in the original topology. These dual vertices are embedded into the leaves of the trie.

The goal of each of these algorithms is to build a binary trie, with nodes in the trie corresponding to IP subnets. A binary trie is a special case of a binary tree in which left branches are labeled with 0, right branches are labeled with 1, and nodes are labeled with the concatenation of labels on edges along the root to node path. Using the trie we build, each leaf, representing a vertex, is given an IP subnet corresponding to its trie label, appended with zeroes to fill out the address in the event the label length is smaller than the desired address length. Similarly, each internal node represents a subnet spanning the IP addresses corresponds to its trie label followed by a *. In this manner, proximate nodes in the trie correspond to proximate addresses in the IP assignment, so this is a natural representation of IP addresses (also extensively used in methods for fast IP lookup).

For example, consider the trie depicted in Figure 5 that embeds the vertices of the dual graph in Figure 4 (all host and link labels in this section come from this figure). Each link in the original network is assigned a subnet based on its path from the root of the trie to the node in the trie that represents the link. For example, the interfaces in link H are in the subnet 1001_b. Since each interface in a link shares a subnet, the interface addresses are implicitly named by assigning a prefix to a link. We show this explicitly for link B where the subnet for the interface at link B on host 4 is 11010_b.

The routing table in Figure 6 is prefix-based, with longer prefixes taking precedence over shorter prefixes. The route to 1_b affects links A, H, F, I, B, and G. The more specific route 11011_b overrides the route to the interface at link B on host 5. The choice of embedding determines the smallest routing table size that can be achieved at a given host; different embeddings can clearly have significant impact.

We now describe our methods. Each of our three algorithms for trie embedding approaches the problem from a different angle. Our first algorithm is a top-down approximation using graph partitioning methods. Second, we describe an algorithm that uses a bottom-up greedy tournament to create a binary trie. The third approach decomposes the IP address assignment problem into two simpler subproblems: 1) identifying an appropriate ordering the vertices, and 2) embedding that ordering into a trie.

3.2.1 Recursive Graph Partitioning

A natural approach to trie-building is to perform a top-down decomposition of the graph. For a variety of applications, especially those relating to scientific computing, hierarchical decomposition of a graph is a useful preprocessing step that facilitates parallel processing. One widely studied approach is recursive graph partitioning, where at each step, an input topology is recursively partitioned into a set of subtopologies. Typically, a partition is constrained by having each of the subtopologies attain some minimum size, and feasible partitions are scored by a metric, such as the size of the edge cut set induced by the partition. While optimal graph

partitioning is NP-hard in general, the problem is well studied and there are a number of libraries which provide good heuristic approximations. METIS [23] is among the most widely used graph partitioners. We chose METIS because it supported the metric described above (minimizing edge cut on k equally-sized partitions) among many others that have potential applicability to our problem domain.

We use METIS to perform a full recursive decomposition of the graph down to the vertex level. Such a decomposition naturally creates a tree (not necessarily binary) in which an internal node represents a subgraph and its children reflect the one-round recursive decomposition of this node. Since METIS performs a round of partitioning in time linear in the size of the graph, a full recursive decomposition has overall time complexity of $O(n \cdot \log(n))$ (where n is the number of vertices), assuming that each partitioning round reduces the size of the largest subgraph by a constant factor.

One characteristic of METIS and graph partitioners in general is that they require the user to specify the number of partitions to create at each round. METIS, like many k-way partitioners, finds k partitions by recursive bisection. Since our trie-building algorithm recursively calls METIS, we can bypass the problem of selecting the number of partitions by using METIS exclusively for bisection (two partitions). We tested this intuition by experimenting with search-based approaches to choosing the number of partitions, at each round partitioning the graph into 2–20 subgraphs, scoring each different partitioning, and selecting the partitioning with the best score. We tried several scoring metrics for evaluating the number of partitions and found two promising ones: conductance [22] and ratio cut [38]. Comparing these two search-based approaches with one another and with bisection, we empirically determined that the number of partitions per level had relatively little impact on the final routing table size. For simplicity and runtime performance, we settled on partitioning the graph into two subgraphs, as it immediately produces a binary trie.

A final component of a top-down algorithm is an appropriate termination condition. The obvious termination condition is when the partition is trivial (size 1). But another important case is when the address bitspace is exhausted, a problem we address in more detail in Section 3.3.

3.2.2 Bottom-Up Tree Building

The second approach we describe leverages the intuition that a natural way to assign addresses is bottom-up, i.e., to identify groups of vertices that can be aggregated into a logical subnet while causing minimal increases of routing table sizes throughout the hosts of the network. In terms of the trie that is constructed, the grouping operation corresponds to producing a rooted trie for the entire subnet, where the children are the vertices or groups of vertices that were present prior to the coalescence operation. Indeed, if we have an appropriate function $\mathrm{benefit}(S,T)$ that quantifies the benefit of merging arbitrary sets of vertices S and T, then we can readily construct a binary trie via the following greedy tournament:

Greedy Tournament

$X = \{\{v_1\}, \{v_2\}, \{v_3\}, \dots\}$
repeat until $|X| = 1$
  For all $S, T \in X$, compute $\mathrm{benefit}(S,T)$
  For maximizing $\mathrm{benefit}(S,T)$, create $U = S \bigcup T$
  Delete S and T from X.
  Add U to X.
end repeat

There are two key challenges to realizing this approach: defining an appropriate benefit function, and avoiding the time complexity embodied in naive direct implementation of this approach, which involves $O(n^2)$ computations of $\mathrm{benefit}(S,T)$ in each of $n-1$ rounds.

Routing Equivalence Sets To motivate the derivation of what we believe to be an appropriate benefit function in the scenario above, we consider two sets of vertices that constitute logical subnets S_1 and S_2, and perform the thought experiment: is S_1 a good candidate for aggregation with S_2? To quantify the benefit, consider that of the vertices in $V\ (S_1 \bigcup S_2)$, there will be some set of vertices which use the same first hop to all vertices in $(S_1 \bigcup S_2)$, and thus could express them in a single routing table entry if we give all vertices in $(S_1 \bigcup S_2)$ addresses that allow aggregation. Some vertices will require different first hops to reach the vertices in $(S_1 \bigcup S_2)$, and thus cannot aggregate routes to them. Therefore, the benefit of aggregating S_1 and S_2 is proportional to the size of the first set, or those external vertices who can save a routing table table entry.

Following this intuition, we have devised Routing Equivalence Sets (RES) as a way to characterize the benefit of aggregating the addresses of sets of vertices. It is an elegant and useful way to reason about the effects of aggregation on routing table size.

For a set of vertices D, those vertices whose first-hop routes to *all* vertices in D are identical are said to be in the Routing Equivalence Set of D, $\mathrm{res}(D)$. Equivalently, if vertices of D are assigned IP addresses in the same subnet, then a routing table in any member of $\mathrm{res}(D)$ can store all routes to D with a single routing table entry. Formally, let V be the set of vertices in a graph. Let D be a set of destination vertices. Let $H_x[y]$ be the first hop from source vertex x to destination vertex y. Then we define $\mathrm{res}(D)$ as:

$$\mathrm{res}(D) = \{v \in V : \forall d, e \in D, H_v[d] = H_v[e]\}.$$

Figures 7 and 8 show a concrete example of the RES function. In each, the first-hop routes from each of the vertices not in D to each vertex in D are shown. Vertices that have a single outbound arrow, such as vertex 4, use the same first hop to every vertex in D, and are thus members of $\mathrm{res}(D)$. Vertexes with multiple outbound arrows, such as vertex 5,

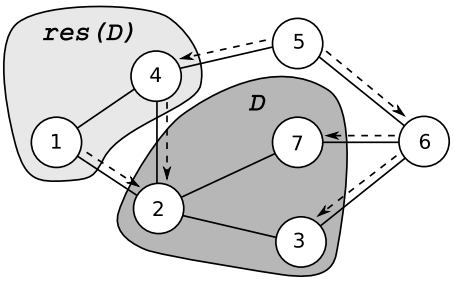

Figure 7: $\mathrm{res}(D)$ for set $D = \{2, 3, 7\}$. First-hop routes from vertices outside D to the members of D are shown as arrows.

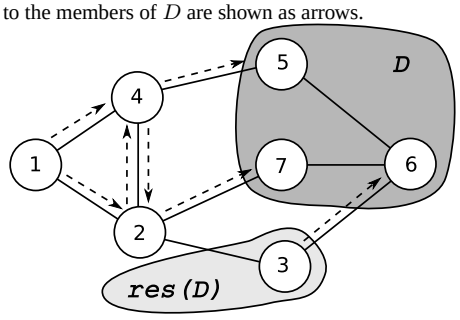

Figure 8: $\mathrm{res}(D)$ for set $D = \{5, 6, 7\}$.

must store multiple first hops to reach all of D along shortest paths, and are thus not members of $\mathrm{res}(D)$.

We use RES to measure the impact on routing table sizes of aggregating sets of vertices. Returning to our example in Figures 7 and 8, since $|\mathrm{res}(\{2, 3, 7\}| > |\mathrm{res}(\{5, 6, 7\})|$, it is more advantageous to aggregate the former set than the latter. Indeed, we can use the magnitude of the RES set as the measure in performing pairwise comparisons: the maximum benefit merger in *Greedy Tournament* is the one where the RES of the resulting aggregated subnet is maximized.

One potential issue is that computing a RES set directly from this definition is expensive: computing res(D) has time complexity $O(n|D|^2)$. However, we can prove (and do so in the Appendix) that res(D) has a clean recursive decomposition that is amenable to much more efficient computation with the following lemma.

Lemma 1 *For any sets D and E, and given $\mathrm{res}(D)$ and $\mathrm{res}(E)$, $\mathrm{res}(D \cup E)$ can be computed in time O(n).*

Efficient Tournament Design Using the RES metric, we now use the greedy tournament algorithm to build a binary address assignment trie from the vertices in the graph. We determine the cost of merging and the order of the coalescence operations by setting $\mathrm{benefit}(S, T)$ to $\mathrm{res}(S, T)$. Next we demonstrate how to improve upon the running time of the tournament. Let the n be the number of vertices in the graph. A straightforward implementation of the RES tournament takes $n - 1$ rounds, and must find the max of a set of $O(n^2)$ RES sets, each of which takes $O(n)$ time to construct in the worst case. This leads to a running time of $O(n^4)$. Although n^2 pairwise combinations must be considered for each of $n - 1$ rounds, there is an optimization available that cuts the running time by a factor of n. In the first round, we must compute the RES metric for all n singletons, and all n^2 possible combinations for singletons. In subsequent rounds, however, the RES values of most of the possible n^2 combinations have not changed—in fact, the only ones that have changed are those pairs in which S or T were one of the combined vertices. There are at most $2n$ such combinations. So, in fact, we can store all possible combinations in a priority queue, and only update those entries that changed based upon the winners of a given round. Scoring a pairwise combination of two sets with RES can be done in $O(n)$ time as proven in Lemma 1. Thus, all rounds after the first take $O(n^2)$ time, and there are $n - 1$ of them, leading to an overall time complexity of $O(n^3)$ for the tournament. Storing the values of combinations under consideration in a priority queue requires $O(n^2)$ space.

3.2.3 Holistic Orderings

A final approach we consider are two-phase methods that first produce an order on the vertices in the graph (but do not lend themselves to a natural binary trie embedding), and then embed this ordering into a trie.

Our main motivation is twofold: first, we speculated that the use of spectral methods, and in particular, use of the Laplacian, might well provide an ordering of the vertices that could be leveraged to produce a good binary tree embedding. Second, we are often given an ordering of the vertices when the test topology is specified, and we noticed that this (non-random) default ordering often captures some interesting locality in the graph. We were curious as to the quality that an embedding of this default ordering would provide. Experimental results are provided in the subsequent section; here we focus on the mechanics of our use of spectral methods and of converting an ordering into an embedding.

The Laplacian Ordering Our starting point is a standard technique from graph theory, that of obtaining an ordering using the Laplacian matrix [14] of the graph and the eigenvector associated with the second-smallest eigenvalue of the matrix [27, 21]. We refer to the second-smallest eigenvalue by λ_2 and the associated eigenvector by $\vec{v_2}$. The Laplacian matrix is essentially a representation of the adjacency of vertices in the graph, and thus it contains only *local* information about each node. The vector $\vec{v_2}$ contains a value for each vertex in the graph. These values can be used to generate an approximation for a minimum-cut bipartitioning of the graph. The characteristic value for each vertex relates to its distance from the cut, with vertices in the first partition having negative values, and those in the second partition having positive values. By sorting the vertices by their characteristic values, we obtain a spectral ordering.

DRE Ordering A limitation of using only the second-smallest eigenvector of the Laplacian is that this captures

notions of adjacency, but does not necessarily capture the notions of similarity between vertices from the perspective of routing. We therefore considered an alternative Laplacian-like graph that goes beyond 0/1 adjacency values and instead incorporates real-valued coefficients that reflect the degree of similarity between a pair of vertices. To do so, we devised a new metric, called Degree of Routing Equivalence (DRE). DRE is defined for a pair of vertices $i, j \in V$, as:

$$\mathrm{dre}(i,j) = |\,\mathrm{res}(\{i,j\})|$$

We then construct an n by n matrix containing the DRE for every pair of vertices. In essence, what we have created is similar to the Laplacian of a fully-connected graph, with weights on the edges such that the higher the edge weight, the more benefit is derived from placing two vertices together. This more directly captures the routing properties of vertices than the standard Laplacian. As with the Laplacian, we then take a characteristic valuation of the matrix to obtain an ordering.

From Ordering to Trie Embedding After an ordering has been generated, constructing an appropriate trie embedding is relatively straightforward. A tournament similar to *Greedy Tournament* can be run, except that not all pairs of vertices need be considered in each round, but only those remaining vertices that are adjacent in the original ordering. This reduces the tournament running time by another factor of n, since there are now only O(n) such pairs in a given round, not O(n^2). Finally, by the same trick used in the original tournament, only two new combinations need be scored in rounds subsequent to the first, so the total time complexity of the tournament is $O(n^2)$. In the event that the ordering has done an effective job in grouping vertices that are similar from a routing perspective, then the intuition is that the tournament algorithm will still produce a good assignment tree.

3.3 Address Compaction

As noted earlier, a key practical limitation is that the IP address space is limited. In IP version 4, each address is limited to 32 bits. In our domains, the networks to label are much smaller than the Internet, and thus smaller address space limits can be required or desired. For example, emulated networks may need to smoothly interact with the Internet, so must be assigned within a smaller subnet; simulations may be more accurate with realistic address densities; and enterprises own limited IP space.

The trie-building algorithms must be designed to gracefully deal with bitspace exhaustion. The number of bits allotted to the topology represents the maximum depth of trie we can build. We have developed and implemented bitspace compaction algorithms for our two best algorithms, tournament RES and recursive partitioning.

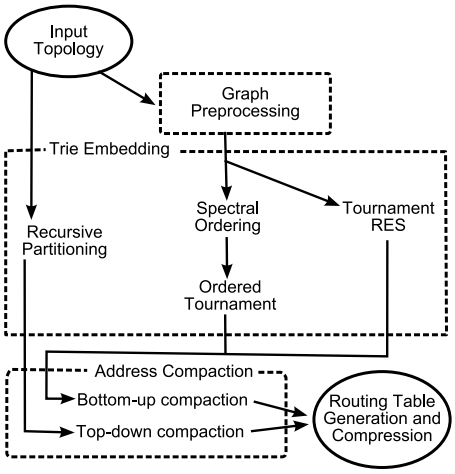

Figure 9: A flowchart showing, at a high level, how the different algorithms in this section are combined. The dashed boxes correspond to the first three subsections of this section.

3.3.1 Bottom-Up Compaction

The basic tournament trie building algorithm operates on a forest of subtrees, combining pairs of subtrees to build a single trie bottom-up. Given a particular set of subtrees and this combining operation, there is a minimum-depth tree that can be formed from the subtrees. We incrementally calculate the depth of this tree in constant time as subtrees are combined. After a combination, if the minimum-depth tree is equal to the number of bits allocated to the topology, the algorithm halts and yields the minimum-depth tree.

3.3.2 Top-Down Compaction

The basic recursive partitioning algorithm operates from the top down. At each stage, the partitioner checks to see if a partitioning would result in a subgraph that is larger than the subnet size assigned to it. If such a situation occurs, sequential addresses are assigned to all of the LANs in the subgraph.

These two algorithms represent two extreme trade-offs. When there is plenty of bitspace, both algorithms ignore the bitspace limitations. They continue until there is just enough bitspace to squeeze the remaining hierarchy into. At that point, the sole objective becomes minimizing bitspace consumption. The bottom-up algorithm produces poorer quality assignments at the top of the trie, while top-down produces them at the bottom.

3.4 Putting It All Together

Figure 9 shows how the methods detailed in the previous subsections logically fit together. All of the trie embedding schemes go through the prepass except for the recursive partitioner, which is sufficiently fast that it does not see a speed improvement from the prepass. After the prepass, each component of the graph is processed separately to obtain a local naming. We then re-combine the components, and run them through the bottom-up address compaction algorithm to produce a global naming. Here again, the recursive parti-

tioner uses its own compaction algorithm. Finally, we compute the routing tables for all nodes and compress them with ORTC [12], which, for a given naming, produces provably optimal routing tables. It is the number of routes in these compressed tables that we report in the next section.

4 Experimental Results

We now move on to evaluate the algorithms presented in the last section by using them on a variety of topologies, and comparing their resulting routing table sizes and runtimes.

4.1 Methodology

We ran experiments on topologies from three sources: two popular router-level topology generators, and the Internet. Our primary interest is the generated topologies because such topologies are prevalent in simulation and emulation. The real Internet topologies serve two purposes. First, they give us insight into the applicability of our methods on ISP and enterprise networks. The second is that new research in topology models and generators [1, 25] is improving the degree to which they are representative of the real Internet—thus, these topologies give us a sense of how well our methods will operate on future generations of topology generators.

The first set of topologies are generated by the BRITE [26] topology generator, using the GLP [8] model proposed by Bu and Towsley. These topologies are intended to model the topology within a single organization, ISP, or AS. The second set of topologies are generated by the GT-ITM [13] topology generator. They model topologies that include several different administrative domains; each topology consists of some number of transit networks and stub networks. Thus, they contain some level of inherent hierarchy. Finally, we use some real-world topologies gathered by the Rocketfuel topology mapping engine [32]. These are maps of real ISPs, created from traceroute data of packets passing through the ISP for a large set of $(source, destination)$ pairs. All Rocketfuel graphs are of individual transit ASes. Although they are annotated with some IP addresses, there are not enough to reconstruct probable routing tables, so we ignore the annotations.

For each topology we report the number of interfaces, rather than the number of nodes. Since it is interfaces that must be named, this gives a more accurate view of the complexity of the assignment problem. Our test topologies, like those generated by most topology generators, contain links and not LANs. Thus, the number of interfaces is twice the number of links in the topology. All topologies are router-level topologies—they contain no end hosts. End hosts do not significantly impact the complexity of the assignment problem, because they tend to be organized into relatively large LANs, which can be assigned in a single step.

After generating IP addresses, we generate routing tables and compress them with ORTC [12], which, for a given address assignment, produces a provably optimal routing table.

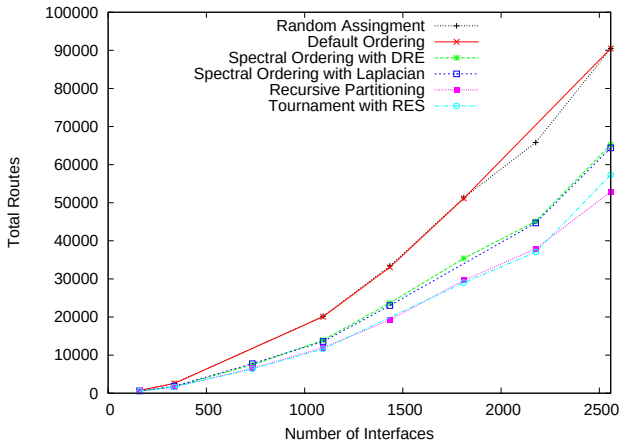

Figure 10: Global number of routes for a variety of assignment algorithms. Topologies are from the BRITE topology set. The lines for recursive partitioning and tournament RES overlap at the bottom of the graph.

We also run the generated routing tables through a consistency checker to verify their correctness. The run times reported are for the assignment of addresses, as routing table generation and compression need to be performed no matter what the assignment algorithm is.

All of our experiments were run on Pentium IV processors running at 3.0 GHz, with 1 MB of L2 cache and 2 GB of main memory on an 800 MHz front side bus.

4.2 Full-Graph Algorithms

We begin by comparing results for the set of assignment algorithms without using the prepass stage. Due to the high time and memory complexity of many of these methods, we are limited in the size of topologies we can compare.

BRITE Topologies

Figure 10 shows the global aggregate routing table size produced by each method for the BRITE topology set. We see that all algorithms do significantly better than random assignment, with the best (recursive partitioning) saving 42% of routes over random, at around 2500 interfaces. For this topology generator, the assignment derived from its generated ordering is indistinguishable from a random assignment.

The two spectral methods perform very similarly to each other, with 29% fewer routes than random. This similarity is surprising, because the Laplacian matrix contains only local information, while the DRE matrix contains global routing-specific information. This suggests that global knowledge is not as useful in this situation as it might seem. Finally, we see that recursive partitioning and tournament RES also perform similarly, producing the smallest routing tables among the methods we studied.

GT-ITM Topologies

Figure 11 shows results from the GT-ITM topology set. This time, the route savings are more pronounced—the best improvement we see over random assignment is 70% fewer routes. However, the various assignment algorithms are far

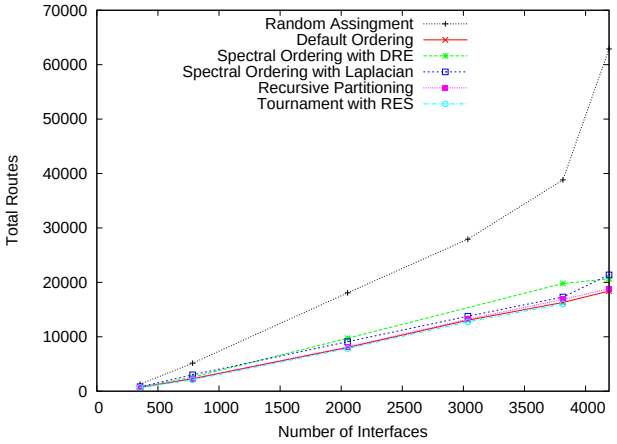

Figure 11: Global number of routes for a variety of assignment algorithms. Topologies are from the GT-ITM topology set.

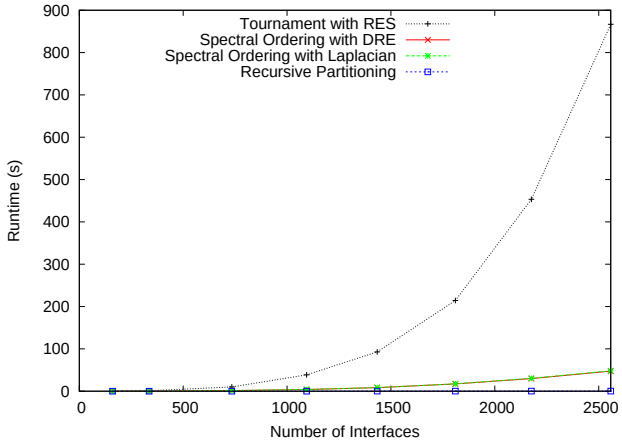

Figure 12: Run times in seconds for a variety of assignment algorithms, on the BRITE topology set. The lines for the two spectral orderings overlap; they are in the middle of the graph. The recursive partitioning line is nearly coincident with the X axis.

Algorithm	EBONE	Tiscali
Random	27031	33104
Default Ordering	15904	18891
Spectral Laplacian	18128	22761
Spectral DRE	15581	20579
Recursive Partitioning	11630	16802
Tournament RES	11427	16354

Table 1: Number of routes generated for the Rocketfuel topologies.

more clustered: most result in similar routing table sizes. It is interesting to note that this time, the default ordering is competitive with the more sophisticated orderings. We believe this to be a side-effect of how GT-ITM generates its topologies.

Rocketfuel Topologies

Rocketfuel graphs provide some idea of how the algorithms compare on real topologies. One disadvantage of using Rocketfuel is that the graphs it generates are often disconnected. To make our results more representative, we use two of the Rocketfuel graphs which are connected. The Rocketfuel results are shown in Table 1 for two European networks, EBONE and Tiscali. Like the BRITE topology set, the tournament RES and recursive partitioning algorithms perform similarly, with a slight advantage going to tournament RES. The default ordering again achieves remarkable improvement over random—in fact, it is better than the spectral methods.

Run Time Comparison and Summary

Finally, Figure 12 shows the run times for the BRITE topology set for our various assignment algorithms. Here, recursive partitioning is the clear winner. Its run time appears near linear, while all other methods show quadratic behavior in their run time. In summary, this set of experiments clearly shows that recursive partitioning is the most efficient method among those evaluated. In terms of quality, assessed by number of routes, it is essentially a toss-up between recursive partitioning and tournament RES.

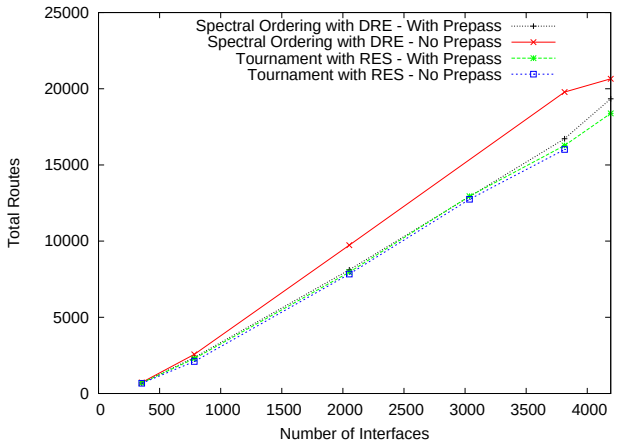

Figure 13: Routing table sizes with and without the prepass.

4.3 Prepass Effects

We now evaluate the prepass, its effects on address assignment and runtime, and a rough characterization of the subgraphs it generated. For two representative algorithms above, DRE spectral ordering and tournament RES, we study the effects of the prepass on their behavior, using GT-ITM topologies. (Due to the similar results we found above for DRE and Laplacian orderings, we do not consider the latter here, while recursive partitioning never uses the prepass.)

We expect to see three effects. First, the prepass finds tree-like structures and assigns addresses to them optimally, which will tend to improve results. Second, by virtue of making decisions using a greedy local algorithm, we will clearly make some sub-optimal global decisions. Third, we expect to see a significant reduction in overall running time by reducing the effective input sizes of the subgraphs.

Routing Table Size Figure 13 shows that, for these graphs, the positive and negative effects of the prepass can balance, in terms of solution quality. The table sizes of tournament

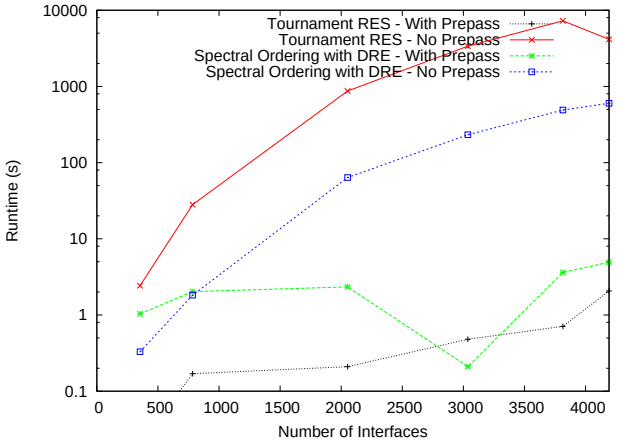

Figure 14: Run times in seconds, with and without the prepass. The y-axis is on a logarithmic scale.

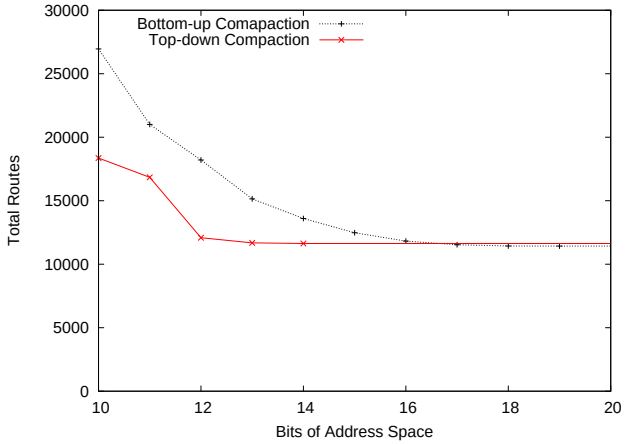

Figure 15: Total routes resulting from limiting the bitspace available to the compaction algorithms. Smaller routing tables are better. No improvement occurs past 20 bits for either algorithm. These results are from the EBONE topology.

Component size	GT-ITM	BRITE	Rocketfuel
1	4	15	29
2–10	49	22	23
11–20	1	4	1
21–30	1	4	
91-100	1		
321-330		1	
420-430			1
# of components	56	46	54
# of links	392	547	543

Table 2: Histogram of prepass component sizes, in links, for three graphs.

RES are almost identical, while DRE spectral ordering obtains 10–15% improvement from the prepass.

Run Time Benefit Figure 14 shows the run time improvement due to the prepass. Note that the improvement is so dramatic that the y-axis is shown in a logarithmic scale. At 3800 interfaces, tournament RES runtime drops by 3 orders of magnitude, from over 2 hours to less than a second, while spectral DRE improves by over 2 orders of magnitude, to about 2 seconds. It is remarkable that we can obtain this level of speedup without losing quality. But by appropriate splitting of the graph into smaller, simpler pieces, the prepass is not only able to bring the run times down to a practical level, but is able to improve route quality by optimally assigning some of the components.

Component Characterization Lastly, we obtain a quantitative and qualitative feel for the subgraph components themselves. For this part of the study, we chose a representative topology from each of the three topology sets (GT-ITM, BRITE, Rocketfuel); we chose those with the most similar link counts. Table 2 shows a histogram of the sizes of the components into which the prepass divides these input topologies. The smaller the largest component, the better the running time of the quadratic algorithms. We can see from this table that the prepass has varying levels of effectiveness for the different topology types. For the GT-ITM topology, the largest component is roughly one-fourth of the topology size, while on the other topologies, the largest components are three-fifths and four-fifths of the size of the original topology. In all topologies, the majority of the components are smaller than 10 nodes. Manual examination reveals that in all three topologies, the largest components are biconnected. The smaller components are almost always trees.

4.4 Address Compaction

We compared the routing tables resulting from the bottom-up (as implemented in the RES tournament) and top-down (as implemented in the recursive partitioner) compaction algorithms. We started by limiting the bitspace to 10 bits; this is the smallest address space the topology will fit in, since there are 543 links. We then relaxed the bitspace constraints until both methods converged to their minimum size routing tables. Figure 15 shows the results of this test.

In both cases, limiting the bitspace results in more routes. This is expected, because dense use of the address space packs together sets of nodes that aggregate poorly at the expense of sets that aggregate well. The top-down compaction handles small bitspaces more gracefully than bottom-up does. When nodes are more tightly packed into the address space, it produces a routing table only 57% larger than when it has unlimited bit space. In comparison, the bottom-up method produces routing tables 135% larger. The clear conclusion is that top-down compaction is much more suitable for limited bit space.

The fact that the two converge at different points illustrates a difference in the bitspace used by the tournament RES algorithm when compared to recursive partitioning. When bitspace is not constrained, the former uses 19 bits of address space, which is why it converges at this point in the graph. The latter converges at only 14 bits. In general, the RES tournament makes sparser use of the address space than recursive partitioning.

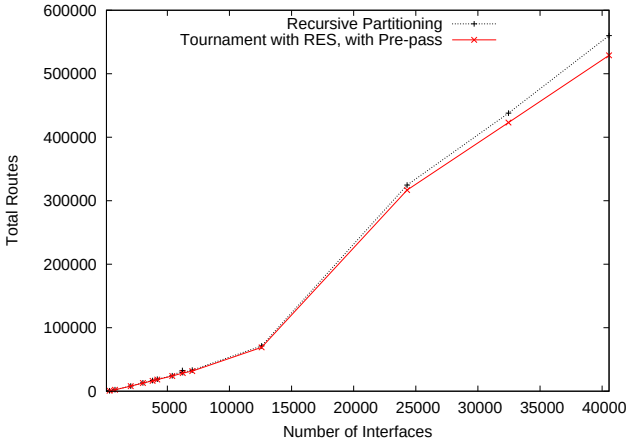

Figure 16: Total number of routes on large graphs.

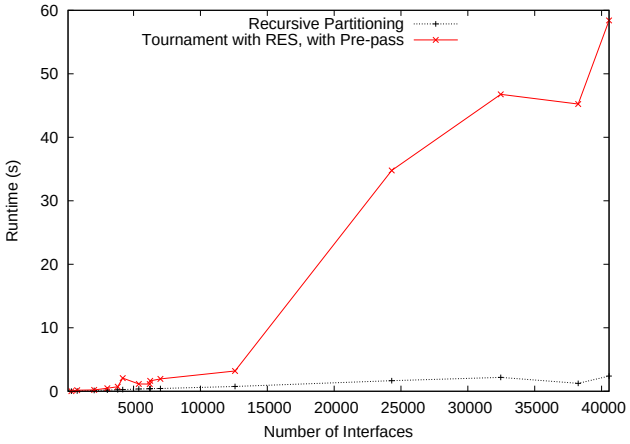

Figure 17: Run times on large graphs, in seconds.

4.5 Large Graphs

Finally, we compare the two best algorithms, recursive partitioning and tournament RES with the prepass. Figure 16 shows the number of routes for these experiments, and Figure 17 shows the run times. The two algorithms produce a nearly equal number of routes, with the slight advantage again going to tournament RES. For graphs under 12,500 interfaces, their run times are comparable and very low, but for the largest graphs, recursive partitioning shows much better scaling. Recursive partitioning is preferable for time-sensitive applications, but tournament RES provides slightly better results and still completes in under a minute on even the largest topology. The largest graph in this test set has 5,000 router nodes, apparently as large as today's largest single-owner networks [3].

4.6 Summary of Experimental Results

Figure 18 summarizes the results across all algorithms, showing the number of routes for one large, representative topology from each generator and for both Rocketfuel graphs. The number of routes for each topology is normalized to the number of routes for the random address assignment. There are two clear patterns in these results: First, in most topologies the spectral orderings perform worse than the default ordering. Second, the recursive partitioning and RES tournament methods consistently yield the fewest routes, with RES usually having a slight edge. The clear conclusion is that the latter two methods are superior.

It is important to keep in mind that all topologies are router-level and do not contain end hosts. Thus, they are most representative of ISP networks. In an enterprise or campus network, if we conservatively estimate 5 to 10 end hosts per router, then the largest topologies in our results represent networks of 25,000 to 50,000 nodes. Our best algorithms are clearly efficient enough to scale to very large networks.

5 Related Work

Methods for optimizing an assignment of names to hosts in a network to minimize routing table size date back to the mid-1980s [37, 16]. In 1987, van Leeuwen and Tan formulated the notion of interval routing [37]; their work and subsequent work studied the problem of computing bounds on the *compactness* of graphs, a measure of the space complexity of a graph's shortest-path routing tables using interval routing [18, 15]. Their work is similar in direction to our problem; however, their work emphasizes worst-case routing table size for specific families of graphs and uses the idealized interval routing approach instead of CIDR.

A more recent direction, mostly pursued in the theoretical literature, is compact routing [4, 11, 33]. By relaxing the requirement of obtaining true shortest paths, compact routing enables much smaller routing tables at the expense of routes with *stretch*: the ratio between the routed source-destination path and the shortest source-destination path. Although these methods appear potentially promising for realistic Internet topologies [24], routing tables that support true shortest-path routes are still the norm in simulated, emulated, and real-world environments.

A different direction related to our work is that of designing routing table compression schemes for network simulators and emulators, to avoid the $O(n^2)$ memory required for precomputing all-pairs shortest-path routes. For example, NIx-Vector-related designs [30, 31] replace static tables with on-demand dynamic route computation and caching. Each packet contains a compact representation of the remaining hops to its destination. This source routing means that routing at each individual node is simple and fast. Depending on the traffic pattern, the size of this global cache can be much smaller than the memory required to pre-calculate all of the routes.

Another practical alternative uses spanning trees [9]. Several spanning trees are calculated, covering most of the edges in the topology. These spanning trees cover most of the shortest-path routes in the topology and a small cache is kept for the remainder. The spanning trees and cache can be stored in a linear amount of memory. While this is a novel routing method, it assumes global information, since routing

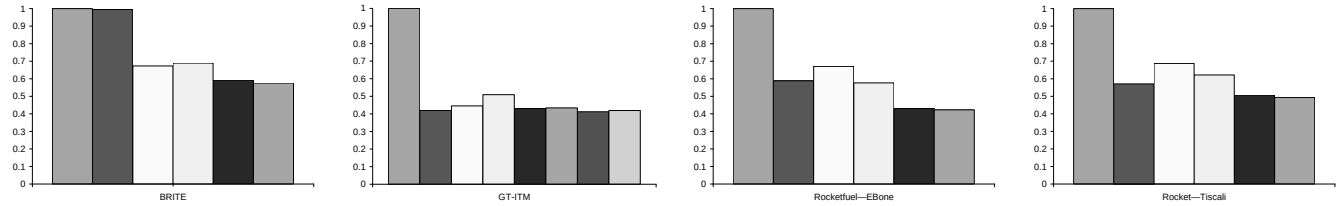

Figure 18: Summary comparison of global routing table sizes resulting from different algorithms. The results are normalized to the largest table size, which results from *random assignment*. From left to right, the bars illustrate the results of *random assignment*, *default ordering*, *spectral ordering with Laplacian*, *spectral ordering with DRE*, *recursive partitioning* and *tournament RES*. For GT-ITM, in the last two bars we also show the results of *tournament RES with prepass*, and *spectral ordering with DRE and prepass*.

requires access to the global spanning trees and cache, a potential bottleneck for distributed simulations and emulations.

Finally, there has been work on optimizing Internet routing tables. First, a number of guidelines for CIDR addressing have been proposed to facilitate manual assignment of IP addresses [19, 20] to take advantage of CIDR routing. Second, a method is known which minimizes the number of table entries for a *given* set of routes and IP addresses. The Optimal Routing Table Construction (ORTC) [12] technique optimally compresses IP routing tables using CIDR rules. ORTC takes a routing table as input and produces a compressed routing table with the same functional behavior, taking advantage of CIDR rules to aggregate routes where possible. For any IP address assignment, ORTC optimally minimizes the number of routes in the routing table. We employ ORTC as a post-processing step in our work.

6 Discussion And Conclusion

We have investigated challenges associated with annotating an Internet-like topology with IP addresses in such a way as to minimize the size of routing tables on hosts and routers. While there is considerable related work, especially for interval routing, none of it adequately handles the complexities of CIDR aggregation: longest prefix matching, the need to name network interfaces instead of hosts, and the nuances of addressing hosts on LANs with all-pairs connectivity. We argue that these factors must be considered in realistic simulation and emulation environments, and our experimental results indicate that they impose a challenging set of constraints beyond those imposed by a more basic interval routing problem.

Our prepass phase captures a fundamental aspect of route-aware address assignment. By partitioning the topology at critical points, it has a dramatic impact on the runtimes of several otherwise infeasible algorithms, making them practical, while making minimal sacrifices in assignment quality.

The methods that we proposed and implemented for address assignment attacked the problem from a number of angles. All of our methods produce routing tables that are far better than those that result from naive, randomly chosen assignments, but two consistently show the best results. Recursive partitioning consistently runs the fastest, and produces very small routing tables. The RES concept leads to the best solutions, and the greedy tournament we use it with can likely be improved upon, either by a faster or more optimal algorithm. Beyond that, we feel that RES makes an important theoretical contribution, providing a clean and flexible way of quantifying "routing similarity".

References

[1] D. Alderson, J. Doyle, R. Govindan, and W. Willinger. Toward an optimization-driven framework for designing and generating realistic internet topologies. *ACM SIGCOMM Computer Communication Review*, Jan. 2003.

[2] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the internet impasse through virtualization. *IEEE Computer*, Apr. 2005.

[3] Anonymized for double-blind submission. Personal communication, 2006.

[4] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Improved routing strategies with succinct tables. *J. of Algorithms*, 11(3):307–341, 1990.

[5] S. Bajaj et al. Improving simulation for network research. Technical Report 99–702b, University of Southern California, March 1999. revised September 1999.

[6] C. Berge. *Graphs and Hypergraphs*, volume 6, pages 389–390. North-Holland, Amsterdam, second edition, 1976.

[7] R. Braden. RFC 1122: Requirements for Internet hosts—communication layers. Technical report, IETF, 1989.

[8] T. Bu and D. Towsley. On distinguishing between Internet power law topology generators. In *Proc. INFOCOM*, pages 1587–1596, July 2002.

[9] J. Chen, D. Gupta, K. V. Vishwanath, A. C. Snoeren, and A. Vahdat. Routing in an Internet-scale network emulator. In *MASCOTS*, pages 275–283, 2004.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, Cambridge, Mass., 1990.

[11] L. Cowen. Compact routing with minimum stretch. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 255–260, 1999.

[12] R. Draves, C. King, S. Venkatachary, and B. Zill. Constructing optimal IP routing tables. In *Proc. of IEEE INFOCOM*, pages 88–97, 1999.

[13] K. C. E. Zegura and S. Bhattacharjee. How to model an internetwork. In *Proc. of IEEE INFOCOM*, pages 594–602, Mar. 1996.

[14] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23(98):298–305, 1973.

[15] M. Flammini, J. van Leeuwen, and A. Marchetti-Spaccamela. The complexity of interval routing on random graphs. *The Computer Journal*, 41(1):16–25, 1998.

[16] G. N. Frederickson and R. Janardan. Designing networks with compact routing tables. *Algorithmica*, 3:171–190, 1988.

[17] V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC 1519: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, Sept. 1993.

[18] C. Gavoille and D. Peleg. The compactness of interval routing. *SIAM J. on Discrete Mathematics*, 12(4):459–473, 1999.

[19] E. Gerich. RFC 1466: Guidelines for management of IP address space, May 1993.

[20] K. Hubbard, M. Kosters, D. Conrad, D. Karrenberg, and J. Postel. RFC 2050: Internet Registry IP Allocation Guidelines, Nov. 1996.

[21] M. Juvan and B. Mohar. Optimal linear labelings and eigenvalues of graphs. *Discrete Applied Mathematics*, 36(2):153–168, 1992.
[22] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM*, 51(3):497–515, May 2004.
[23] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. on Scientific Computing*, 20(1):359–392, 1999.
[24] D. Krioukov, K. Fall, and X. Yang. Compact routing on Internet-like graphs. In *Proc. IEEE INFOCOM*, pages 209–219, 2004.
[25] P. Mahadevan, D. Krioukov, M. Fomenkov, B. Huffaker, X. Dimitripoulos, kc claffy, , and A. Vahdat. The internet as-level topology: Three data sources and one defi nitive metric. In *ACM SIGCOMM Computer Communications Review (CCR)*, Jan. 2006.
[26] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An approach to universal topology generation. In *MASCOTS 2001*, pages 346–356, Aug. 2001.
[27] B. Mohar. The Laplacian spectrum of graphs. In Y. Alavi, G. Chartrand, O. Ollermann, and A. Schwenk, editors, *Graph theory, combinatorics, and applications*, volume 2, pages 871–898, New-York, 1991. John Wiley and Sons, Inc.
[28] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.
[29] J. Rexford. Personal communication, May 2005.
[30] G. F. Riley, M. H. Ammar, and R. Fujimoto. Stateless routing in network simulations. In *MASCOTS 2000*, pages 524–531, 2000.
[31] G. F. Riley and D. Reddy. Simulating realistic packet routing without routing protocols. In *PADS'05: Proc. of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 151–158, Washington, DC, 2005. IEEE Computer Society.
[32] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *SIGCOMM 2002*, pages 2–16, Pittsburgh, PA, 2002.
[33] M. Thorup and U. Zwick. Compact routing schemes. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 1–10, 2001.
[34] J. Touch. Dynamic Internet overlay deployment and management using the X-Bone. *Computer Networks*, pages 117–135, July 2001.
[35] J. Touch, Y.-S. Wang, L. Eggert, and G. Finn. A virtual internet architecture. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA 2003)*, Karlsruhe, Germany, August 2003.
[36] 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. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 271–284, Boston, MA, Dec. 2002.
[37] J. van Leeuwen and R. Tan. Interval routing. *The Computer Journal*, 30:298–307, 1987.
[38] Y. Wei and C. Cheng. Ratio cut partitioning for hierarchical designs. *IEEE Trans. on Computer-Aided Design*, 10(7):911–921, July 1997.
[39] 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. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002.

Appendix

Hypergraph Biconnectivity and Hypertrees

In Section 2, we described why our methods label the *dual hypergraph* of the input topology (also a hypergraph). To perform the prepass described in Section 3.1, we must extend the definitions of biconnectivity and trees into the domain of hypergraphs. There are a number of alternative definitions which potentially apply; we use the following one which best fits our purposes.

For every path p, the function $\text{edges}\,(p)$ is the set of edges or hyperedges along that path. From here forward, we will use the term 'edge' in a general sense to denote either an edge or a hyperedge. A pair of vertices u and v is said to be *edge-biconnected* if and only if there exist two paths p and q between u and v such that:

$$\text{edges}\,(p) \cap \text{edges}\,(q) = \emptyset$$

Similarly, an *edge-biconnected component* is a set of vertices V such that for all u, v in V, u and v are edge-biconnected. An *edge-biconnected partitioning* of a graph G is a partitioning of the vertices of G into partitions $G_1, G_2, \ldots G_n$ such that for all i, G_i is a maximal edge-biconnected component.

Using similar notions, we define a *hypertree* to be a connected subgraph of a hypergraph that contains no cycles. As with trees on regular graphs, it is straightforward to optimally assign IP addresses to a hypertree of a hypergraph.

Using these definitions, our pre-processing step partitions the hypergraph into edge-biconnected components and hypertrees. Fast algorithms for computing such a decomposition on regular graphs are known; by maintaining some additional information about vertices incident to each hyperedge, these methods can be extended to apply to hypergraphs. Once the decomposition is complete, addresses on the hypertrees are assigned optimally by a special tree-assignment procedure; addresses are assigned on the edge-biconnected components by the procedures described earlier. The supergraph of partitions is created and can be used to label the partitions themselves.

Proof of Lemma 1

Lemma 1 *For any sets D and E, and given res(D) and res(E),* $\text{res}(D \cup E)$ *can be computed in time O(n).*

Proof: First note that by transitivity, for any v and for all $a, b, c \in V$:

$$(H_v[a] = H_v[b] \wedge H_v[b] = H_v[c]) \rightarrow (H_v[a] = H_v[c])$$

Further, the definition of RES and transitivity imply that for destination set D, and specializing to $v \in \text{res}\,(D)$ and $d, e \in D$,

$$H_v[a] = H_v[d] \rightarrow H_v[a] = H_v[e]$$

Which means that $\forall v \in V, \forall d \in D$:

$$\text{res}\,(D \cup \{v\}) = \text{res}\,(D) \cap \text{res}\,(\{v, d\})$$

Therefore, given two destination sets D and E, we can select *any* $d \in D$ and $e \in E$ to give the recurrence:

$$\text{res}\,(D \cup E) = \text{res}\,(D) \cap \text{res}\,(E) \cap \text{res}\,(\{d, e\}).$$

Since $\text{res}(\{d, e\})$ can be computed from the definition in O(n) time, and assuming use of standard set representations that allow intersection in linear time, the lemma follows. ■