Constructing Optimal IP Routing Tables

Richard P. Draves
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Christopher King
Department of Mathematics
Northeastern University
Boston, MA 02115

Srinivasan Venkatachary
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Brian D. Zill
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Abstract—**The Border Gateway Protocol (BGP) populates Internet backbone routers with *routes* or *prefixes*. We present an algorithm to locally compute (without any modification to BGP) equivalent forwarding tables that provably contain the minimal number of prefixes. For large backbone routers, the Optimal Routing Table Constructor (ORTC) algorithm that we present produces routing tables with roughly 60% of the original number of prefixes. The publicly available MaeEast database with 41315 prefixes reduces to 23007 prefixes when ORTC is applied. We present performance measurements on four publicly available databases and a formal proof that ORTC does produce the optimal set of routes.**

I. Introduction

As the Internet grows to fill every corner of the world, the demands on the Internet backbone routers keep increasing. One of the major problems facing the backbone routers today is the increasing number of routing entries or prefixes that they have to handle. The number of routes in the Internet backbone has been growing by 10,000 per year for the last several years [11].

In this paper we present an algorithm for constructing a routing table that has the least possible number of entries, while still providing the same routing information. More precisely, given a routing table that provides forwarding information for IP addresses using longest prefix match, the algorithm produces a new routing table that a) has the same forwarding behavior and b) has the least possible number of entries. The table is constructed by applying subnetting and supernetting to the original table. We call our algorithm ORTC (Optimal Routing Table Constructor). ORTC can be easily and efficiently implemented, and reduces the number of prefixes in a large backbone router by around 40%. While for our experiments we use IPv4 prefixes, ORTC is applicable to any longest-matching-prefix database.

Routers acquire these routes using a distributed algorithm, the Border Gateway Protocol (BGP) [13]. Because of the way BGP operates, routers can get redundant routes. Modifying BGP and other routing protocols to produce optimal routing tables at each router is a daunting task. In contrast, ORTC is a simple mechanism by which each router can locally compute an optimal set of routes.

This reduction in the number of prefixes while retaining equivalent forwarding information helps improve performance. Having fewer prefixes reduces the size of the forwarding data structure. If one uses customized hardware for forwarding, it might have a limited amount of memory and it would be worthwhile to increase the effective size of the routing tables that it could hold. If one uses a general purpose processor with cache memory to do forwarding, a reduced forwarding structure size means a larger fraction of the structure can fit into the cache, improving average case performance.

The paper is organized as follows. We discuss related work in section II. In section III we describe ORTC and provide an example using a small routing table. We also discuss the intuition behind ORTC and state the optimal compression result. In section IV we evaluate the performance of ORTC on publicly available backbone routing tables. Section V contains discussion and conclusions. An appendix presents a mathematical formulation of ORTC and a formal optimality proof.

II. Background

Initial work on routing for large networks [1] established that hierarchical routing produces routing tables logarithmic in the number of network hosts with negligible increase in message path lengths. This is important for achieving scalability as network sizes increase. Internet routing today takes advantage of this principle.

Internet address lookup would be simple if we could lookup a 32-bit IP destination address in a table that lists the output link for each assigned Internet address. However, each router would have to keep an entry for every Internet host—millions of entries. To reduce database size and routing update traffic, an Internet router database consists of a much smaller set of *prefixes*. This reduces database size, but at the cost of requiring a more complex lookup called *longest matching prefix*. Each prefix P has an associated *next hop* or *output link* information, which specifies where a packet is to be forwarded if its longest matching prefix is P.

We will write prefixes as bit strings of up to 32 bits followed by a '*'. For example, the prefix 01* matches any address that begins with the bits 01. The prefix * matches every address. Thus if the destination address begins with 01000 and we had only two prefix entries (01* $\rightarrow$ 1; 0100* $\rightarrow$ 2), the longest-matching-prefix would be 0100* and so the packet would be directed to next-hop 2.

The Internet initially used a simple hierarchy in which 32-bit addresses were divided into a network prefix and a host number, so that routers would only store entries for networks. For flexible address allocation, the network prefixes came in three sizes: Class A (8 bits), Class B (16 bits), and Class C (24 bits). Organizations that required more than 256 hosts were given class A or B addresses; these organizations could further structure their addresses for internal routing with *subnetting* [3]. For example, if P1=00* is a network, then P2=0011* is a *subnet* under P1. However, the Class A and B spaces did not scale to handle the Internet's growth. This led to the invention of Classless Inter-Domain Routing (CIDR) [6]. CIDR can give organizations multiple contiguous network prefixes that can still be aggregated by a common prefix, which reduces backbone router table size. *Supernetting* denotes the aggregation of adjacent prefixes that have the same next-hop information. For example, supernetting can replace P1=00* $\rightarrow$ 1 and P2=01* $\rightarrow$ 1 with one prefix,

0-7803-5417-6/99/$10.00 ©1999 IEEE.

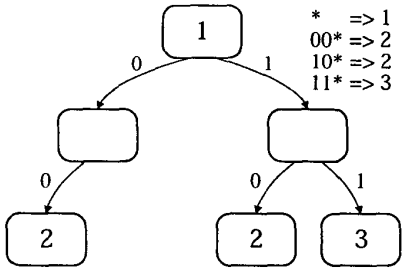

Fig. 1. Binary tree representation of a routing table.

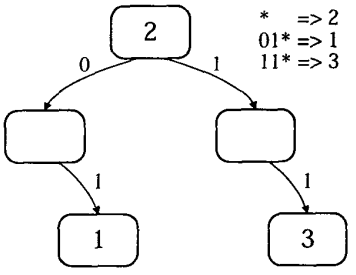

Fig. 2. Example routing table after ORTC.

P=0* → 1.

In the past year, there have been several new algorithms that provide very fast IP lookups in software [9], [8], [10], [12]. Note that the work presented in this paper can be used to improve the memory requirements of any of these schemes. In particular, section IV presents the reduction in size that we achieved using ORTC in conjunction with the data structure described in [9].

There is some prior work in the area of reducing routing table sizes while preserving equivalent forwarding behavior. Most notably, [7] defines Binary Tree Collapse (BTC). BTC consists of three transformations, each of which recognizes a common opportunity for eliminating redundant routing table entries. The BTC algorithm performs a single post-order traversal of a binary tree built from the routing table, looking for opportunities to apply its three transformations. BTC does not handle multiple next hops per prefix. In contrast, an ORTC implementation uses two passes but produces provably minimal table sizes. ORTC can either preserve multiple next-hop information, or take advantage of it to improve compression. On an actual Internet backbone routing table, ORTC produces an optimized routing table that is 43% smaller while BTC produces a routing table that is 35% smaller.

III. Constructing Optimal Routing Tables

Our algorithm for reducing the number of entries in a routing table generalizes the subnetting and supernetting techniques. In this section we first give an intuitive explanation of the algorithm's operation. We then provide a detailed description, in terms of operations on a tree representation of the routing table. To simplify the explanation, the initial description relies on having a single next hop for a prefix and having a default route for the null prefix. We remove these restrictions in the final subsection.

To graphically depict a set of prefixes, we use a binary tree representation. Each successive bit in a prefix corresponds to a link to a child node in the tree, with a 0 corresponding to the left child and a 1 corresponding to the right child. Note that the binary tree generally contains more nodes than there are prefixes, since every successive bit in the prefix produces a node. We label nodes with next-hop information, typically a small integer or a set of small integers. Figure 1 shows an example with four routes. For instance, the root node in the tree represents the null prefix, with a default route to next-hop 1. The lower left tree node represents the prefix 00*, and the next hop associated with this prefix is 2.

Figure 2 shows the output of our algorithm on this example. By changing the default route at the root of the tree from 1 to 2, ORTC reduces the number of routes from four to three. Note that the optimized routing table encodes forwarding behavior equivalent to the original routing table. Using the longest-matching-prefix algorithm, both routing tables forward 00 to next-hop 2, 01 to next-hop 1, 10 to next-hop 2, and 11 to next-hop 3.

A. Intuition

Several key observations let ORTC gain maximal advantage from supernetting and subnetting and produce an optimally compressed routing table equivalent to the original routing table. The first observation, a generalization of supernetting, is that the shorter prefixes, close to the root of the tree, should route to the most popular or prevalent next hops. Longer prefixes, near the leaves of the tree, should route to less prevalent next hops. Then subnetting will prune the maximal number of routes from the tree. Finally, these ideas should be applied recursively, within every sub-tree.

In Figures 1 and 2, note that next-hop 2 is most prevalent in the original routing table: it accounts for half of the possible destinations. Hence ORTC produces a smaller routing table by moving it to the root of the tree and using longer prefixes to represent routes to next-hops 1 and 3.

B. Description of the Algorithm

In its simplest form, ORTC optimizes a routing table using three passes over the binary tree representation. (The next subsection describes optimizations to the basic algorithm, including combining the first two passes.) The first pass propagates routing information down to the tree's leaves. The second pass finds the most prevalent next hops, by percolating information (sets of next hops) from the leaves back up towards the root. Finally, a third pass moves down the tree, choosing a next hop from the set of possibilities for a prefix and eliminating redundant routes.

This description of the algorithm makes two simplifying assumptions about routing tables, which do not hold for real backbone routing tables. First, we assume that every routing table has a default route, or equivalently, that the null prefix at the root of the tree has a next hop. Second, we assume that every prefix in the routing table has only a single next hop. The next subsection shows how these assumptions can be relaxed so that ORTC can be applied to real routing tables.

We use the routing table from Figure 1 as an example throughout this section, to demonstrate the operation of the algorithm. Section III-C restates this section's informal description in a more precise form using pseudo-code.

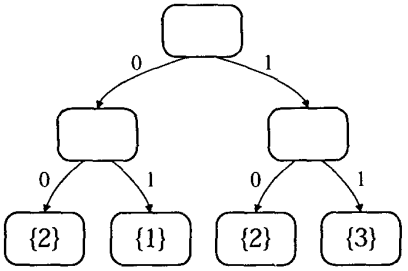

Fig. 3. Example routing table after pass 1.

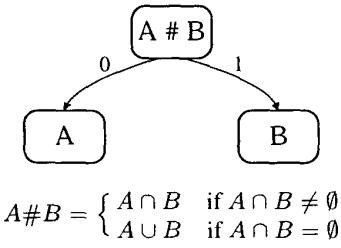

Fig. 4. Percolating sets of next hops up the tree.

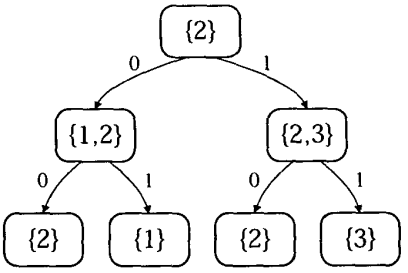

Fig. 5. Example routing table after pass 2.

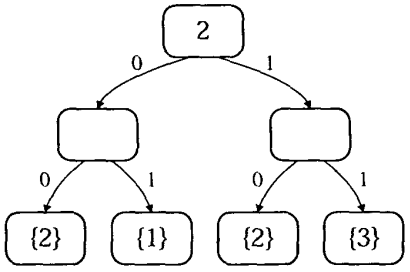

Fig. 6. Example routing table during pass 3.

B.1 Pass One

The first pass "normalizes" the binary tree representation of the routing table, in preparation for the second and third passes. It enlarges the tree so that every node has either zero or two children. It does this by creating new leaf nodes and initializing the next hop for a new node with the next hop that the new node inherits from its nearest ancestor that has a next hop. Once the tree is fully populated with leaf nodes, the next-hop information for interior nodes is no longer needed and may be discarded. In preparation for the second pass, which uses sets of next hops, the first pass converts the next hop for each prefix to a singleton set. Figure 3 shows the result of pass 1 processing on the example routing table.

An implementation of the first pass might use a pre-order traversal of the binary tree or a traversal by levels from the root down. In either case, the traversal pushes next-hop information down to child nodes that do not have a next hop, creating new child nodes when a parent node has only one child.

B.2 Pass Two

The second pass calculates the most prevalent next hops at every level of the routing table by percolating sets of next hops up the tree. An implementation of the second pass could use a post-order traversal of the tree or a traversal by levels from the bottom up towards the root. At each parent node visited in the traversal, a set of next hops is calculated as shown in Figure 4. If there are any next hops in common between the two child nodes, then they are the next hops that are most prevalent at the level of the parent node. Otherwise all the next hops from the children's level are carried up to the parent node.

When the second pass is complete, every node in the tree is labeled with a set of potential next hops. Figure 5 shows the result of pass 2 processing on the example routing table.

B.3 Pass Three

The third pass moves down the tree selecting next hops for prefixes and eliminating redundant routes via subnetting. An implementation could use either a pre-order traversal of the tree or a traversal by levels from the root down. Each node visited will have a set of possible next hops, computed in the second pass. Except for the root node, the node will inherit a next hop from the closest ancestor node that has a next hop. If this inherited next hop is a member of the node's set of potential next hops, then the node does not need a next hop of its own: it is inheriting an appropriate next hop. However, if the inherited next hop is not a member of the node's set of potential next hops, then the node does need a next hop. Any member of the node's set of potential next hops may be chosen as the node's next hop.

Figure 6 and Figure 7 demonstrate this phase of ORTC on the example routing table, using a traversal by levels. After the second pass, the root node is labeled with a singleton next-hop set $\{2\}$, so the third pass selects the next hop 2 for the root. Because next-hop 2 is a member of the sets labeling the two children of the root, the root's children have their next hops removed. Figure 6 depicts this intermediate point in the third pass. Figure 7 depicts

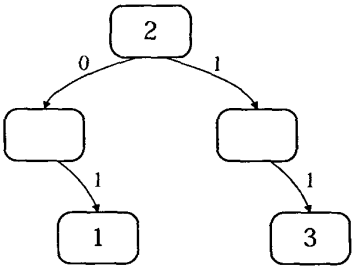

Fig. 7. Example routing table after pass 3.

the final output of the algorithm, after the traversal visits the four nodes at the bottom of the tree. Two of those nodes do not need next-hop information because they inherit an appropriate next hop from the root.

C. ORTC Definition

Our definition of ORTC uses pseudo-code to render more precisely the previous section's informal description.

The pseudo-code algorithm operates on a binary tree. The symbol N denotes a node in the tree. $nexthops(N)$ denotes a set of next hops associated with the node N. If the routing table does not assign next hops to N, then $nexthops(N)$ is defined to be the empty set $\emptyset$. We assume $nexthops(root) \neq \emptyset$. For nodes with children, $left(N)$ and $right(N)$ denote the left and right child nodes. Similarly, we define $parent(N)$ for all nodes except the root. The operation $choose(A)$ picks an element from the non-empty set A.

As in Figure 4, we define the operation $A\#B$ on two sets of next hops:

$$A\#B = \begin{cases} A \cap B & \text{if } A \cap B \neq \emptyset \\ A \cup B & \text{if } A \cap B = \emptyset \end{cases}$$

We define the function $inherited(N)$ on nodes other than the root:

$$inherited(N) = \begin{cases} nexthops(parent(N)) & \text{if} \neq \emptyset \\ inherited(parent(N)) & \text{otherwise} \end{cases}$$

The first and third passes perform a traversal from the tree's root down to its leaves. This can be either a pre-order traversal or a traversal by levels. Similarly, the second pass performs a traversal from the leaves up to the root, using either a post-order traversal or a traversal by levels.

Pass One.

```
for each node N (root to leaves) {
    if N has exactly one child node,
        create the missing child node
    if nexthops(N) = ∅,
        nexthops(N) ← inherited(N)
}
```

Pass Two.

```
for each node N (leaves to root) {
    if N is a parent node,
        nexthops(N) ←
            nexthops(left(N))#nexthops(right(N))
}
```

Pass Three.

```
for each node N (root to leaves) {
    if N is not the root and
      inherited(N) ∈ nexthops(N)
        nexthops(N) ← ∅
    else
        nexthops(N) ← choose(nexthops(N))
}
```

D. Optimality Theorem

Although ORTC is very simple in operation, it always yields a routing table that is optimal, in the sense that the output routing table has the smallest number of prefixes possible while still maintaining the same forwarding behavior. In the third pass of the algorithm, the algorithm may choose a next hop from a set of potential next hops. This means that the algorithm may produce many different output routing tables for a given input table. ORTC guarantees that all of these possible output routing tables are optimal, and hence they are all the same size.

The appendix contains a mathematical formulation of ORTC and a complete proof of its optimality. The proof proceeds via induction on the levels of the tree corresponding to the routing table.

E. Improvements

The basic ORTC algorithm just presented neglects several factors important in the real world. These include performance and stability, as well as the two assumptions that the input routing table contains a default route (a next hop for the null prefix) and that the input table contains a single next hop for its prefixes. This subsection discusses these issues and presents enhancements to the basic algorithm.

E.1 Improving Performance

There are several ways to reduce the number of steps in ORTC and hence improve performance.

It's possible to skip the first pass. When the second pass (now being done first) comes across a parent node with only one child node, then at that time it can create the new child node and assign an inherited next hop to the new child. This is an example of lazy evaluation; the work of the first pass is delayed until it is really necessary.

Another performance improvement saves some work in the third pass by anticipating it in the second pass. In the second pass, when a parent node is assigned the intersection of its child nodes' sets of potential next hops, then the next-hop information for the two child nodes can be deleted. This immediately prunes those prefixes from the routing table. This is a safe optimization because a member of the intersection of two sets is by definition a member of both sets. In the third pass, that parent node will be assigned a next hop from the intersection (or it will inherit such a next hop). If the third pass processed the child nodes, it would see that they inherit a next hop that is a member of their potential set, and prune them at that time.

The complexity of the algorithm is linear in the number of nodes in the tree. The number of tree nodes is $O(wN)$ where w is the maximum number of bits in the prefixes and N is the number of prefixes in the input routing table. For IPv4, $w <= 32$ and for IPv6, $w <= 128$. Implementing the algorithm using path-compressed tries [14] would reduce the number of tree nodes to $O(N)$, speeding up the algorithm's operation. ORTC's space complexity is the same as its time complexity.

E.2 Improving Stability

In some situations it may be advantageous to compress a routing table without changing it "unnecessarily." In particular, if

a routing table is already optimally compressed, one might like the compression algorithm to produce an output table identical to the input table.

ORTC as given above does not have this property. For example, consider the routing table with two entries: $0^* \rightarrow 1$ and $1^* \rightarrow 2$. (Ignore for a moment the fact that this routing table has no default route; the next subsection removes that restriction.) ORTC will produce as output a different table with two entries, either $^* \rightarrow 1$ and $1^* \rightarrow 2$ or $^* \rightarrow 2$ and $0^* \rightarrow 1$ depending on the choice made at the root node in the third pass.

Two small modifications to the selection of next hops in the third pass improve ORTC's stability. We conjecture that these modifications guarantee that ORTC will not change an already-optimal routing table.

During the third pass, suppose that ORTC must choose a next hop for a parent node from a set X of potential next hops. If this nodes' prefix had a next hop in the input routing table, and that next hop is a member of X, then it is the logical choice. This improves stability because this prefix's next hop will be preserved.

If ORTC must choose a next hop for a parent node but its prefix did not have a next hop in the input routing table, then to improve stability one would like to remove the prefix from the optimized routing table instead of assigning it a next hop. This is a safe modification to the algorithm if the parent node's set of potential next hops was formed by union of its two child nodes' sets. Using the original description of the third pass would result in the parent node and one of its two child nodes generating routes in the output routing table. With this modification, the parent node does not generate a route but instead both child nodes generate routes in the output routing table. Either way, the output routing table contains the same number of routes.

E.3 Removing the Default Route Assumption

As formulated above, ORTC assumes that the input routing table assigns a next hop to the null prefix (a default route). The algorithm's first pass creates new leaf nodes and assigns them inherited next hops, and the presence of the default route ensures that every new child node inherits a next hop. However, in the real world one often encounters "default-free" routing tables. In particular, the backbone routers in the Internet use default-free routing tables.

There are two ways to remove this restriction in ORTC. First, with a more complex definition for the three passes it is possible to cope with the absence of a default route. However, this reduces the effectiveness of supernetting. For example, then it is not possible to optimize the routing table with $00^* \rightarrow 1$, $010^* \rightarrow 1$, $1^* \rightarrow 1$.

A better approach is to introduce a default route to a dummy next hop 0, at the beginning of the first pass. At the end of the third pass, if the dummy route at the root is present in the output table it may be removed. Note that the output table may contain routes to next hop 0. Forwarding to next-hop 0 should be taken as an error, just as if no matching prefix were found. With this improvement, the above routing table with three entries optimizes to yield a routing table with two entries: $^* \rightarrow 1$, $011^* \rightarrow 0$.

E.4 Removing the Single Next Hop Assumption

Our presentation of ORTC assumed that each prefix in the initial routing table has a single next hop, while in real routing tables multiple next hops are common. There are several ways to overcome this limitation in ORTC.

First, it is possible to choose the best next hop for each prefix, by some metric, before applying ORTC to optimize the resulting routing table. This is an appropriate method if the metric can pick a single best next hop from the set of next hops for a prefix.

If several next hops tie for best, then ORTC can use the flexibility it gets by having multiple next hops from which to choose to achieve better compression. In this technique, we allow the input table to ORTC to contain multiple next hops. This requires a small modification to the first pass—a new child node may inherit multiple next hops from its ancestor. The operation of the second and third passes is not affected. This approach allows ORTC to achieve greater compression because it has a better chance of finding prevalent next hops at higher levels of the tree. We call this variation ORTC-1. ORTC-1 does not preserve the multiple next-hop information that existed in the input routing table, so one can not for example round-robin among different next hops when forwarding.

If it is important to preserve the sets of next hops in the input routing table, then it is still possible to apply ORTC. The only modification required is to create "virtual" next hops, where each virtual next hop represents a different set of next hops found in the input routing table. ORTC then optimizes using the virtual next hops, so instead of manipulating sets of next hops it is really manipulating sets of sets of next hops. We call this variation ORTC-m.

IV. Performance

We analyze the performance of ORTC using publicly available routing table databases [11]. We first compare our results with the Binary Tree Collapse scheme [7], showing that ORTC achieves significantly better compression. We then examine ORTC's performance on four large Internet backbone routing tables. For the two largest routing tables, ORTC reduces the routing tables to roughly 60% of their original size. If ORTC is constrained to preserve multiple next hops in the output routing table, then the compressed tables are roughly 70% of their original size. We also examine ORTC's impact on the size of fast forwarding data structures built from the routing tables and look at the ORTC's stability, or how much it changes real routing tables while optimizing them.

Table I compares Binary Tree Collapse (BTC) [7] and ORTC using the same input routing table, the MaeEast table of January 16, 1998. The data here for BTC is taken from [7]. The routing table is segmented into four sections, corresponding to the old IP address classes. For this routing table, optimizing the four sections separately achieves the same result as optimizing them together as one routing table. Overall, BTC reduces the routing table to 65% of its original size and ORTC reduces it to 57% of its original size. In the MaeEast database, each route maps to a set of next hops. ORTC-1 refers to the case where we want to keep only one of these next hops in the optimized routing table and we allow ORTC to pick the one that achieves the best

TABLE I

COMPARISON OF BINARY TREE COLLAPSE (BTC) AND OPTIMAL ROUTING TABLE CONSTRUCTOR (ORTC).

	Class A	Class B	Class C	Swamp	TOTAL
Initial	145	4232	28694	5474	38545
BTC	105	3636	17157	4593	25068 (65%)
ORTC-1	104	2945	14533	4213	21795 (57%)

TABLE II

ORTC PERFORMANCE ON INTERNET BACKBONE ROUTING TABLES.

	Initial	ORTC-1	Time (ms)
MaeEast	41315	23007 (56%)	400
AADS	24418	14964 (61%)	259
MaeWest	18968	13750 (73%)	227
Paix	3020	2593 (85%)	51

compression. When multiple next hops for a prefix are available, BTC just uses the first one and ignores the others.

In Table II we present the reduction in the number of prefixes in four Internet backbone routing tables [11] from July 7, 1998. Entries in these routing tables often have multiple next hops. ORTC compresses the two largest routing tables to roughly 60% of their original size.

In addition, Table II presents ORTC's runtime performance on this data. The running times shown here were measured with a preliminary, non-optimized implementation of ORTC. For the largest routing table, our implementation of ORTC took less than 0.5 seconds.

We also examined ORTC's performance when it is constrained to preserve all the multiple next-hop information in the input routing tables. ORTC-m refers to the case where each different set of next hops is treated as a unique virtual next hop. Table III lists the results obtained with ORTC-m. To our surprise, we found that ORTC-m achieves good compression. It reduces the two largest routing tables to roughly 70% of their original size. ORTC-m is nearly as effective as ORTC-1 because in practice there are not very many unique sets of next hops in use, although the potential number of different sets is enormous. For each routing table, Table III also shows the number of distinct next hops, the number of different sets of next hops, and the maximum and mean number of next hops assigned to routing table entries.

In Table IV, we present the reduction in forwarding structure size for the MaeEast database, using the fast IP forwarding structure presented in [9]. Because ORTC reduces the size of the forwarding structure, a greater fraction of the forwarding structure can fit in cache memory, improving average case forwarding performance. The structure defined in [9] uses multi-level tries, and allows the number of levels used to be varied. Using more

TABLE III

ORTC PERFORMANCE ON INTERNET BACKBONE ROUTING TABLES, WHEN PRESERVING MULTIPLE NEXT-HOP INFORMATION.

	Initial	ORTC-m	Distinct	Virtual	Max	Mean
MaeEast	41315	29995 (72%)	76	842	10	1.7
AADS	24418	16764 (68%)	32	143	4	1.15
MaeWest	18968	15146 (80%)	68	470	8	1.84
Paix	3020	2616 (86%)	16	30	3	1.04

TABLE IV

REDUCTION IN SIZE OF THE MULTIBIT TRIE BASED IP LOOKUP STRUCTURE.

	Before ORTC	After ORTC
Prefixes	41315	23007
2 level trie	1028 KB	670 KB
3 level trie	496 KB	334 KB
4 level trie	402 KB	275 KB
5 level trie	379 KB	260 KB

levels decreases the data structure's memory requirement. Ignoring cache effects, the worst-case lookup time is proportional to the number of levels. For example, using a router configured with 384 KB of fast memory available for forwarding, ORTC allows a 3-level trie to be used instead of a 5-level trie, improving performance.

Finally, we measured how much of the original routing table ORTC preserves in the output compressed table. For the 41315-prefix MaeEast database, which compresses to 23007 prefixes, 15174 (66%) of the prefixes were the same as those in the original database. After improving ORTC's stability as described in section III-E.2, 15772 (69%) of the output prefixes were the same as those in the original database.

V. CONCLUSIONS

We have presented an algorithm (ORTC) for constructing optimal routing tables. We have shown that in typical backbone routers, equivalent forwarding behavior can be obtained with routing tables containing roughly 40% fewer prefixes than the routing tables in use today. We find that a large 41315-prefix table (from MaeEast) can be reduced to an equivalent table with only 23007 prefixes.

We find reductions of up to 30% in the size of multibit trie-based forwarding structures built from the optimized routing tables. In future work, we are considering changing the optimality criteria to be the size of the forwarding structure instead of the number of prefixes.

In practice routers see and generate many incremental routing updates. While we can batch updates and rerun ORTC periodically, in future work we hope to find more efficient ways of handling incremental updates.

Finally, we would like to note that this is a step in the direction of self-configuring networks, which automatically assign network prefixes. When each router has optimized its routing table, the only way to reduce the number of routing table entries further is to renumber the prefixes assigned to networks. From the optimal routing tables, we might be able to get information as to what kind of renumbering will help in further reduction. We plan to pursue this in future work.

ACKNOWLEDGMENT

C. K. thanks C. Borgs, J. T. Chayes and the Microsoft Theory Group for their support and hospitality. We thank Qiyong Bian and Jonathan Turner for giving us the MAE-East data that they used in their study. We thank Allison Mankin and Bill Bolosky for their comments.

References

[1] L. Kleinrock, F. Kamoun. *Hierarchical Routing for Large Networks.* Computer Networks 1:155-174, 1977.

[2] J. Postel, editor. *Internet Protocol.* Internet RFC 791, September 1981.

[3] J. Mogul, J. Postel. *Internet Standard Subnetting Procedure.* Internet RFC 950, August 1985.

[4] V. Fuller, T. Li, J. Yu, K. Varadhan. *Supernetting: an Address Assignment and Aggregation Strategy.* Internet RFC 1338, June 1992.

[5] Y. Rekhter, T. Li. *An Architecture for IP Address Allocation with CIDR.* Internet RFC 1518, September 1993.

[6] V. Fuller, T. Li, J. Yu, and K. Varadhan. *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy.* Internet RFC 1519, September 1993.

[7] J. Turner, Q. Bian and M. Waldvogel. *Routing Table Compression Using Binary Tree Collapse.* Technical Report WUCS-98-13, Washington University in St. Louis, May 1998.

[8] M. Waldvogel, G. Varghese, J. Turner and B. Plattner. *Scalable High Speed IP Routing Lookups.* Computer Communications Review, October 1997.

[9] S. Venkatachary and G. Varghese. *Faster IP Lookups using Controlled Prefix Expansion.* Proceedings of ACM Sigmetrics, June 1998.

[10] M. Degermark, A. Brodnik, S. Carlsson and S. Pink. *Small Forwarding Tables for Fast Routing Lookups.* Computer Communications Review, October 1997.

[11] Merit Inc. *Routing table snapshots.* www.merit.edu.

[12] S. Nilsson and G. Karlsson. *Fast Address Look-up for Internet Routers.* Proceedings of IEEE Broadband Communications, April 1998.

[13] Y. Rechter and T. Li. *A Border Gateway Protocol 4 (BGP-4).* Internet RFC 1771, March 1995.

[14] D. Knuth. *Fundamental Algorithms vol 3: Sorting and Searching.* Addison-Wesley, 1973.

Appendix

Translation, notation and definitions.

We will prove a result, stated in the Theorem below, that implies the optimality of ORTC. In order to state the theorem and prove it, we must formulate a mathematical description of the IP forwarding protocol. We do this next, and then define the algorithm using this notation. Then we state the Theorem and provide the proof. For generality we consider N-bit IP addresses for any integer N. It is convenient to work in a unified setting for all N-bit routing tables, so we embed all routing tables in the full N-bit binary tree with root. This means that we consider a routing table as a map from the full N-bit binary tree into sets of next hops, where the map assigns the empty set to any node on the tree that does not appear in the routing table.

The theorem is proved by induction on N, that is we assume its validity for all k-bit routing tables with $k \leq N$, and then establish the result for any $N+1$-bit routing table. For $N = 1$ the result is proved explicitly. The statement of our Theorem is the induction hypothesis, and this has two parts. The first part says that every routing table has at least as many entries as the compressed tables produced by the algorithm. The second part says that unless the root entry of a routing table (if it is non-empty) is in a certain subset of the set of next hops, then that routing table is strictly larger than the minimal size. This stronger hypothesis is necessary for the proof of the induction step.

The information in a routing table is equivalent to a *forwarding map*, which we define now. Let $\mathcal{H}$ be the set of all possible next hops for the router. We write $2^{\mathcal{H}}$ for the set of subsets of $\mathcal{H}$. Let $\mathcal{A}_N$ be the set of all binary numbers of length N, so $\mathcal{A}_N$ has 2^N elements. Then a *forwarding map* is any map

$$F : \mathcal{A}_N \to 2^{\mathcal{H}} \setminus \emptyset \tag{1}$$

That is, F assigns a non-empty subset of $\mathcal{H}$ to every N-bit IP address. The routing table contains this information in condensed form. Let $\mathcal{B}_N$ be the full binary tree whose vertices are all prefix/mask pairs with mask lengths from 0 up to N, and edges between every parent and child. So $\mathcal{B}_N$ contains $2^{N+1} - 1$ vertices, including the root which we denote r_N. A routing table assigns a non-empty subset of $2^{\mathcal{H}}$ to some vertices of $\mathcal{B}_N$. We extend this by assigning the empty set to all remaining vertices. So we define a *routing map* to be any map

$$R : \mathcal{B}_N \to 2^{\mathcal{H}} \tag{2}$$

For convenience we drop the 'mask' notation for vertices in $\mathcal{B}_N$. So a vertex x/k in $\mathcal{B}_N$ will be denoted by the k-bit binary x, and we write $|x| = k$ to denote its level or length. We call $R(x)$ the *state* of x, and say that x is *occupied* if $R(x) \neq \emptyset$. Also $|R|$ denotes the number of occupied vertices in the tree.

Given a routing map R we can try to define a forwarding map F_R, via the longest prefix match, as follows. Note that there is a unique path from the root r_N to every vertex x in $\mathcal{B}_N$. On this path, either a) there is a unique occupied vertex which is closest to x, but not equal to x, call it $\mathrm{Anc}(x)$ (short for ancestor of x), or b) there are no occupied vertices. We define the *inherited state* of x to be

$$\mathrm{Inh}[x; R] = \begin{cases} R(\mathrm{Anc}(x)) & \text{in case (a)} \\ \emptyset & \text{in case (b)} \end{cases} \tag{3}$$

If $\mathrm{Inh}[x; R] \neq \emptyset$ for all $x \in \mathcal{A}_N$, that is all inherited states at the N-bit level are non-empty, then we say that the routing map R is *complete*. For a complete routing map R we define the forwarding map F_R for every $x \in \mathcal{A}_N$ by

$$F_R(x) = \begin{cases} R(x) & \text{if } R(x) \neq \emptyset \\ \mathrm{Inh}[x; R] & \text{if } R(x) = \emptyset \end{cases} \tag{4}$$

Given a complete routing table R and a forwarding map F, we say that R *covers* F if

$$F_R(x) \subset F(x) \quad \text{all } x \in \mathcal{A}_N \tag{5}$$

In other words, for every IP address the routing table R provides forwarding information that is a subset of the forwarding information provided by F.

Finally, we introduce the two following (non-standard) operations on sets.

a). For a set A, we denote by $\mathrm{Ran}[A]$ a randomly chosen element of A.

b). For any two subsets A and B, define

$$A \# B = \begin{cases} A \cap B & \text{if } A \cap B \neq \emptyset \\ A \cup B & \text{if } A \cap B = \emptyset \end{cases} \tag{6}$$

The algorithm.

The algorithm presented below is equivalent to ORTC. The input is a forwarding map F. In practice this will come from a complete routing map obtained from a routing table, by pulling the routing information all the way down to the 2^N leaves of the full binary tree. However it is not necessary to assume that F arises in this way. The output is a collection of compressed routing tables $\{R_{F;i}\}$ $(i \in I)$, where I is some finite index set whose size depends on F. As we prove later, these are the optimal compressed tables.

The algorithm proceeds in two steps. In the first step the routing information in the leaves is pushed up the tree level by level, until all nodes in the binary tree (including the root) are occupied. In the second step the nodes are successively pruned until only the minimal number of entries remains. Given a vertex x we write $x0$ and $x1$ for the two children vertices.

The input.

Let F be any forwarding map on $\mathcal{A}_N$.

Step 1.

Define inductively a sequence of routing maps $\left(R^{(N)}, R^{(N-1)}, \ldots, R^{(1)}, R^{(0)}\right)$ by

$$R^{(N)}(x) = \begin{cases} \emptyset & \text{if } x \notin \mathcal{A}_N \\ F(x) & \text{if } x \in \mathcal{A}_N, \end{cases} \tag{7}$$

and for $1 \leq k \leq N$

$$R^{(k-1)}(x) = \begin{cases} R^{(k)}(x) & \text{if } x \notin \mathcal{A}_{k-1} \\ R^{(k)}(x0) \# R^{(k)}(x1) & \text{if } x \in \mathcal{A}_{k-1} \end{cases} \tag{8}$$

Step 2.

Construct inductively a sequence of routing maps $\left(T^{(0)}, T^{(1)}, \ldots, T^{(N)}\right)$ by

$$T^{(0)}(x) = \begin{cases} R^{(0)}(x) & \text{if } x \neq r_N \\ \mathrm{Ran}[R^{(0)}(r_N)] & \text{if } x = r_N, \end{cases} \tag{9}$$

and for $1 \leq k \leq N$

$$T^{(k)}(x) = \begin{cases} T^{(k-1)}(x) & \text{if } x \notin \mathcal{A}_k \\ \emptyset & \begin{cases} \text{if } x \in \mathcal{A}_k \text{ and} \\ \mathrm{Inh}[x; T^{(k-1)}] \in T^{(k-1)}(x) \end{cases} \\ \mathrm{Ran}[T^{(k-1)}(x)] & \text{otherwise} \end{cases} \tag{10}$$

The output.

The output of the algorithm is the routing map $T^{(N)}$ constructed at the end of Step 2. Since many choices are made in Step 2, there are many possible results. We denote them by $R_{F;i}$, where i is an index that distinguishes between them. The collection of all indices is a finite set I, so the possible results of the algorithm are the routing maps $\{R_{F;i}\}$ $(i \in I)$.

Comments.

a) For every k, $R^{(k)}(x)$ is the empty set if $|x| < k$, and is non-empty if $|x| \geq k$. So for example the routing map $R^{(0)}$ has an entry for every vertex in $\mathcal{B}_N$, including the root r_N.

b) It is easy to see that the output $T^{(N)}$ covers the input F. Indeed, if $x \in \mathcal{A}_N$, then $T^{(N-1)}(x) = F(x)$, so either $T^{(N)}(x) = \mathrm{Ran}[T^{(N-1)}(x)] \in F(x)$, or else $T^{(N)}(x) = \emptyset$, in which case $\mathrm{Inh}[x; T^{(N)}] = \mathrm{Inh}[x; T^{(N-1)}] \in T^{(N-1)}(x) = F(x)$.

The theorem

Recall that, given a forwarding map F, at the end of Step 1 the algorithm assigns a non-empty set to the root in the routing map R^0. We will denote this set by M_F, that is

$$M_F = R^{(0)}(r_N) \tag{11}$$

Theorem

Let F be a forwarding map on $\mathcal{A}_N$. Let R' be any routing map that covers F. Then

a) $|R'| \geq |R_{F;i}|$ for all $i \in I$.

b) If in addition $R'(r_N)$ is not a subset of M_F, then $|R'| \geq 1 + |R_{F;i}|$ for all $i \in I$.

Comments

a) The Theorem implies that $|R_{F;i}| = |R_{F;j}|$ for all $i, j \in I$, that is all the routing tables constructed by the algorithm have the same size. The Theorem also implies that $|R_{F;i}|$ is the smallest possible size for a routing table that covers F. That is, these tables achieve the optimal compression. So the Theorem implies the optimality of ORTC.

b) The algorithm described above is wildly inefficient, since at the end of Step 1 it constructs a routing table with $2^{N+1} - 1$ entries. The implementation presented in section III does not have this deficiency; see section III-E.1.

c) In order to retain all multiple next-hop information in the original table, define $\Omega = 2^{\mathcal{H}} \setminus \emptyset$. If the algorithm is run with $\mathcal{H}$ replaced everywhere by Ω, then the result will be a family of maximally compressed tables that retain all multiple next-hop information. One could imagine retaining partial next-hop information by using another set in place of Ω, but we do not pursue this question here.

Merging and splitting

The proof of the theorem relies on two operations that we call *merging* and *splitting* of routing tables. For the merging operation we take two routing tables R_0, R_1 on the N-bit tree and join them to form a routing table $R_0 * R_1$ on the $(N+1)$-bit tree. Note that the root node of the $(N+1)$-bit tree is not occupied in $R_0 * R_1$. For the splitting operation we take a routing table R on the N-bit tree and produce a new routing table $\mathrm{Push}[R]$ on the N-bit tree that can be written (uniquely) in the form $\mathrm{Push}[R] = R_0 * R_1$ for some $(N-1)$-bit trees R_0 and R_1. The notation "Push$[R]$" is used to indicate that we push down the routing information from the root to its two children – this is a necessary step before we can split the table into two parts. After doing this, the root entry is irrelevant and can be discarded.

First we define the merging operation. A word on notation: if x, y are k, n-bit binary numbers respectively, we denote by xy the $(k+n)$-bit binary obtained by appending y to x. Let R_0, R_1 be routing maps on $\mathcal{B}_N$. Define the routing map $R_0 * R_1$ on $\mathcal{B}_{N+1}$ by

$$R_0 * R_1(x) = \begin{cases} \emptyset & \text{if } x = r_{N+1} \\ R_0(y) & \text{if } x = 0y \\ R_1(y) & \text{if } x = 1y \end{cases} \tag{12}$$

where as usual r_{N+1} is the root of $\mathcal{B}_{N+1}$.

It will be convenient to define a similar operation for forwarding maps. Let F_0, F_1 be forwarding maps on $\mathcal{A}_N$. We define the forwarding map $F_0 * F_1$ on $\mathcal{A}_{N+1}$ by

$$F_0 * F_1(x) = \begin{cases} F_0(y) & \text{if } x = 0y \\ F_1(y) & \text{if } x = 1y \end{cases} \tag{13}$$

Every forwarding map F on $\mathcal{A}_{N+1}$ can be written uniquely in the form $F = F_0 * F_1$ for some F_0, F_1.

Next we define the splitting operation. Let R be a routing map on $\mathcal{B}_N$. Define a new routing map Push$[R]$ on $\mathcal{B}_N$ as follows:

$$\begin{aligned}
\text{Push}[R](r_N) &= \emptyset \\
\text{Push}[R](0) &= \begin{cases} R(r_N) & \text{if } R(0) = \emptyset \\ R(0) & \text{otherwise} \end{cases} \\
\text{Push}[R](1) &= \begin{cases} R(r_N) & \text{if } R(1) = \emptyset \\ R(1) & \text{otherwise} \end{cases} \\
\text{Push}[R](x) &= R(x) \quad \text{if } |x| \geq 2
\end{aligned} \tag{14}$$

Note that Push$[R]$ has no routing information at the root of the N-bit tree. Hence there are unique routing maps R_0, R_1 on $\mathcal{B}_{N-1}$ such that Push$[R] = R_0 * R_1$.

Next we list some facts concerning Push$[R]$: these all follow easily from the definitions.

1). If R is complete, then Push$[R]$ is also complete, and $F_R = F_{\text{Push}[R]}$.
2). For all R, $|R| - |\text{Push}[R]| \in \{-1, 0, 1\}$. That is, the splitting operation changes the number of entries in R by at most one.
3). For a routing table $R_{F;i}$ produced by the algorithm, $|R_{F;i}| - |\text{Push}[R_{F;i}]| \in \{-1, 0\}$. This is because the algorithm cannot produce a table with entries at the root and at both of its child nodes, and this is the only situation where Push$[R]$ has fewer entries than R.
4). Let F be a forwarding map on $\mathcal{A}_N$, and define forwarding maps F_0, F_1 on $\mathcal{A}_{N-1}$ by $F = F_0 * F_1$. Let R be a routing map on $\mathcal{B}_N$ that covers F. Then there are unique routing maps R_0, R_1 on $\mathcal{B}_{N-1}$ such that Push$[R] = R_0 * R_1$, and such that R_0 covers F_0 and R_1 covers F_1.
5). Again let $F = F_0 * F_1$ be a forwarding map on $\mathcal{A}_N$, and let I_F, I_{F_0}, I_{F_1} be the index sets listing routing maps produced by the algorithm for the forwarding maps F, F_0, F_1 respectively. Then $M_F = M_{F_0} \# M_{F_1}$. Furthermore, for every $i \in I_F$, there exist unique $j \in I_{F_0}$ and $l \in I_{F_1}$ such that Push$[R_{F;i}] = R_{F_0;j} * R_{F_1;l}$. This is the key property of our algorithm – every compressed table constructed on the N-bit tree is equivalent to two compressed tables on the $N-1$-bit subtrees, and these can be recovered using the splitting operation.

The proof

The proof of the Theorem is by induction on N, the number of levels in the tree. First we prove the result for $N = 1$. In this case the tree $\mathcal{B}_1$ has three vertices, namely the root r_1 and its two children 0 and 1. Let $F(0) = A$ and $F(1) = B$. Then $M_F = A \# B$. Let R' be any routing table that covers F, and let $R'(r_1) = G$ (the set G may be empty). Let $R_{F;i}$ be any routing table produced by the algorithm and let

$$\Delta = |R'| - |R_{F;i}|$$

We must prove that $\Delta \geq 0$, and that if G is not a subset of M_F, then $\Delta \geq 1$. Write

$$\Delta = |R'| - |\text{Push}[R']| + |\text{Push}[R_{F;i}]| - |R_{F;i}|$$

(we have used $|\text{Push}[R']| = |\text{Push}[R_{F;i}]| = 2$ since both cover the forwarding map F). There are three cases to consider:

Case 1. $|R'| - |\text{Push}[R']| = 1$.

Recall that $|\text{Push}[R_{F;i}]| = 2 \geq |R_{F;i}|$, hence $\Delta \geq 1$, so we are done.

Case 2. $|R'| - |\text{Push}[R']| = 0$.

Hence $\Delta = |\text{Push}[R_{F;i}]| - |R_{F;i}| \geq 0$. If $|R_{F;i}| = 1$ then $\Delta \geq 1$ so we are done. So assume that $|R_{F;i}| = 2$, in which case by Step 2 of the algorithm $A \cap B = \emptyset$. Also since $|R'| = 2$, then either $G = \emptyset$, in which case $G \subset M_F$ and we are done, or else Push$[R']$ assigns the state G to at least one of the vertices $0, 1$. Since R' covers F, and hence $G \subset F(0)$ or $G \subset F(1)$, so $G \subset F(0) \cup F(1) = A \cup B$. But since $A \cap B = \emptyset$, so $M_F = A \cup B$, hence $G \subset M_F$ and we are done.

Case 3. $|R'| - |\text{Push}[R']| = -1$.

Hence $|R'| = 1$, so G is non-empty. Also Push$[R']$ assigns the state G to both vertices 0 and 1, so $G \subset F(0)$ and $G \subset F(1)$. Hence $G \subset F(0) \cap F(1) = A \cap B$, so $M_F = A \cap B$. Hence $G \subset M_F$, and $|R_{F;i}| = 1$, so $\Delta = 0$.

Now we prove the induction step, namely we assume the result for all integers less than or equal to N, and prove it for $N + 1$. So F is a forwarding map on $\mathcal{A}_{N+1}$, and so can be written as $F = F_0 * F_1$ for unique forwarding maps F_0, F_1 on $\mathcal{A}_N$. Let R' be any routing map on $\mathcal{B}_{N+1}$ that covers F. Define $G = R'(r_{N+1})$. Let $R_{F;i}$ be any routing map constructed from the algorithm. Define

$$\begin{aligned}
\Delta &= |R'| - |R_{F;i}| \\
&= |R'| - |\text{Push}[R']| + |\text{Push}[R']| \\
&\quad - |\text{Push}[R_{F;i}]| + |\text{Push}[R_{F;i}]| - |R_{F;i}|
\end{aligned} \tag{15}$$

We must prove that $\Delta \geq 0$, and that $\Delta \geq 1$ unless $G \subset M_F$. As observed before, there are unique routing maps R'_0, R'_1, and unique indices j, l such that

$$\begin{aligned}
\text{Push}[R'] &= R'_0 * R'_1 \\
\text{Push}[R_{F;i}] &= R_{F_0;j} * R_{F_1;l}
\end{aligned} \tag{16}$$

Define

$$\begin{aligned}
\rho_0 &= |R'_0| - |R_{F_0;j}| \\
\rho_1 &= |R'_1| - |R_{F_1;l}|
\end{aligned} \tag{17}$$

Since R'_0 covers F_0, by the induction hypothesis $\rho_0 \geq 0$, and if $R'_0(r_N)$ is not a subset of M_{F_0} then also $\rho_0 \geq 1$. Similarly for ρ_1. Furthermore since $|R * S| = |R| + |S|$ for any maps R, S, we get

$$\Delta = |R'| - |\text{Push}[R']| + \rho_0 + \rho_1 + |\text{Push}[R_{F;i}]| - |R_{F;i}|$$

Again there are three cases.

Case 1. $|R'| - |\text{Push}[R']| = 1$.

Since $|\text{Push}[R_{F;i}]| \geq |R_{F;i}|$ we deduce that $\Delta \geq 1$, so we are done.

Case 2. $|R'| - |\text{Push}[R']| = 0$.

Hence $\Delta \geq 0$. If $|\text{Push}[R_{F;i}]| = 1 + |R_{F;i}|$ then $\Delta \geq 1$ and we are done. So assume that $|\text{Push}[R_{F;i}]| = |R_{F;i}|$. By the

construction of the map $T^{(1)}$ in Step 2 of the algorithm, it follows that $M_{F_0} \cap M_{F_1} = \emptyset$, and hence $M_F = M_{F_0} \cup M_{F_1}$. Also since $|R'| = |\text{Push}[R']|$, either $G = \emptyset$, in which case $G \subset M_F$ and we are done, or else $G = R'_0(r_N)$ or $G = R'_1(r_N)$, and hence $G \subset R'_0(r_N) \cup R'_1(r_N)$. Now with these assumptions $\Delta \geq \rho_0 + \rho_1$. If $\rho_0 + \rho_1 \geq 1$ then we are done, so assume that $\rho_0 = \rho_1 = 0$. Then by the induction hypothesis, $R'_0(r_N) \subset M_{F_0}$ and $R'_1(r_N) \subset M_{F_1}$. Hence

$$G \subset R'_0(r_N) \cup R'_1(r_N) \subset M_{F_0} \cup M_{F_1} = M_F$$

Case 3. $|R'| - |\text{Push}[R']| = -1$.
Hence

$$\Delta \geq -1 + \rho_0 + \rho_1 + |\text{Push}[R_{F;i}]| - |R_{F;i}|$$

Also G must be non-empty, and $G = R'_0(r_N) = R'_1(r_N)$. If moreover $\rho_0 + \rho_1 \geq 1$ and also $|\text{Push}[R_{F;i}]| - |R_{F;i}| \geq 1$ then $\Delta \geq 1$ and we are done. So consider the two remaining subcases.

Subcase 3.1 $\rho_0 = \rho_1 = 0$.

Then $R'_0(r_N) \subset M_{F_0}$, and $R'_1(r_N) \subset M_{F_1}$, hence $G \subset M_{F_0} \cap M_{F_1}$. Since G is non-empty, this implies that $M_F = M_{F_0} \cap M_{F_1}$, that $G \subset M_F$ and that $\Delta \geq 0$, so we are done.

Subcase 3.2 $|\text{Push}[R_{F;i}]| = |R_{F;i}|$.

Then $M_{F_0} \cap M_{F_1} = \emptyset$. Hence if $G = R'_0(r_N) \subset M_{F_0}$, then $G = R'_1(r_N)$ is not a subset of M_{F_1}, and vice versa. Hence $\rho_0 + \rho_1 \geq 1$, so $\Delta \geq 0$. Also in this case $M_F = M_{F_0} \cup M_{F_1}$, so if G is not a subset of M_F then $R'_0(r_N)$ is not a subset of M_{F_0} and $R'_1(r_N)$ is not a subset of M_{F_1}, hence $\rho_0 + \rho_1 \geq 2$ and $\Delta \geq 1$.
QED