Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-98-13

1998-01-01

Routing Table Compression Using Binary Tree Collapse

Jonathan Turner, Qiyong Bian, and Marcel Waldvogel

This paper describes an algorithm which can roughly halve the size of the current Internet routing tables. This algorithm is based on the radix trie representation of routing tables, which was firstly used in the BSD Unix distributions. The binary tree representation, which is a simplified case of radix tree, does well at showing the relationships among all routing table entries and provides us a way to build a collapse algorithm based on its internal structure. The binary tree collapse algorithm consists of three techniques, with the first two quite intuitive while the third is a bit more elaborate.... Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

 Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
Turner, Jonathan; Bian, Qiyong; and Waldvogel, Marcel, "Routing Table Compression Using Binary Tree Collapse" Report Number: WUCS-98-13 (1998). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/468

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/468

Routing Table Compression Using Binary Tree Collapse

Jonathan Turner, Qiyong Bian, and Marcel Waldvogel

Complete Abstract:

This paper describes an algorithm which can roughly halve the size of the current Internet routing tables. This algorithm is based on the radix trie representation of routing tables, which was firstly used in the BSD Unix distributions. The binary tree representation, which is a simplified case of radix tree, does well at showing the relationships among all routing table entries and provides us a way to build a collapse algorithm based on its internal structure. The binary tree collapse algorithm consists of three techniques, with the first two quite intuitive while the third is a bit more elaborate. All techniques used in this algorithm are discussed and their effects on reducing the size of the routing table are listed and compared.

Routing Table Compression Using Binary Tree Collapse

Jonathan Turner, Qiyong Bian, Marcel Waldvogel

WUCS-98-13

May 98

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

This paper describes an algorithm which can roughly halve the size of the current Internet routing tables. This algorithm is based on the radix trie representation of routing tables, which was firstly used in the BSD Unix distributions. The binary tree representation, which is a simplified case of radix trie, does well at showing the relationships among all routing table entries and provides us a way to build a collapse algorithm based on its internal structure. The binary tree collapse algorithm consists of three techniques, with the first two quite intuitive while the third is a bit more elaborate. All techniques used in this algorithm are discussed and their effects on reducing the size of the routing table are listed and compared.

Routing Table Compression Using Binary Tree Collapse

Jonathan Turner *jst@arl.wustl.edu*
Qiyong Bian *bian@dworkin.wustl.edu*
Marcel Waldvogel *mwa@tik.ee.ethz.ch*

1. Introduction

A routing table sits at the core of every router that interconnects computer networks. Generally, each router has a number of links that are connected to other routers or hosts, and to route an incoming packet from any link, the router must consults the routing table to decide which output link the packet is supposed to be sent to. There are a variety of algorithms that compute the topology of the network, and routing table is built as a result of such topology computations. Each routing table entry usually consists of an address prefix as the key, along with a next hop address. So given an incoming packet, the destination address of the packet is extracted and is compared to all the entries in the routing table. The number of matches may range from zero to many. For the "many" case, the best matching, i.e., the entry with the longest matching prefix wins and the next hop address of the winning entry is used as the outgoing link. If there is no such matching, an error message will be sent to the originating link. The above procedure will also be repeated at the next hop but the routing table might be different, as in many times a router is free to choose a routing algorithm and the algorithm does not need to be the same as its neighbor's.

One of the many things that are affected by the rapid explosion of the Internet in recent years, is the size of the routing table. When the Internet was first created, there were just a dozen supercomputer centers that were interconnected together. At those days routing packets among these hosts was not complicated, each host only needs to know the topology of the whole network and creates a routing table enumerating next-hops for any possible incoming packets. Since the topology was simple and there were not too many hosts altogether, the routing table was consequently quite simple and the size of such tables was relatively small.

Such conditions no longer hold, of course. With the exponential growth of the Internet in recent years, we have seen the same thing happened on those routing tables. Even though the routing are now done hierarchically, the growth is still there on each level of the routing table, especially those sitting on the network backbones. As an example, table 1 shows the number of entries, sorted by Class A, B, C and Swamp, of the Mae-East backbone router taken from a day of January 1997 and January 1998, respectively. As a note for the Swamp class, it was what was first (before CIDR and hierarchical address assignment) assigned when anyone requested a Class C address(i.e. 192.*). Because of the limitation of the original Class C(each Class C address has only up to 255 individual host addresses), it was very under-utilized and was the reason why CIDR came along. So strictly speaking, Swamp is the original Class C while the current "Class C" is the addresses greater than 193.* and is assigned hierarchically under the CIDR scheme. Detailed discussion of the CIDR scheme can be found in [1]. From here we can see the size of the routing table is ever growing. Therefore,

Mae-east Routing Table	Class A	Class B	Class C	Swamp
January, 1997	45	4,637	22,979	5,087
January, 1998	145	4,232	28,694	5,474

Table 1: Routing table data from a major ISP

how to effectively reduce the size of the Internet routing tables without affecting the integrity of the infrastructure of the Internet becomes an interesting problem.

This paper gives an algorithm which can roughly compress the current Internet routing table to half of its original size. It is based on the radix trie representation of routing table entries which was first used in the BSD Unix distributions. The binary tree representation discussed in this paper is one of the possible ways to graphically represent a trie. While the binary tree collapse is based on the trie structure organization of the routing table, it can also be applicable to other routing schemes.

The rest of the paper is organized as follows: Section 2 explains the basic data structure of the binary tree and its nodes, with each field of the internal structure discussed in detail; Sections 3, 4 and 5 explain the three techniques employed in the algorithm, respectively; Section 6 shows the results from our experiments and Section 7 summarize the results and gives an outlook.

2. The Basic Binary Tree Structure in The Collapsing Algorithm

Given a routing table, the first step is to build a corresponding binary tree. The routing table entry comes with the format

```
destination address/prefix length      next hop address
```

The reason that the key consists of both the destination address and a prefix length is because usually a router does not route traffic the way as from a particular *host* to another *host*; building an routing table entry for every possible IP address would make the routing table too large(up to 2^{32} entries). Instead, the router routes packets from a *network* to another *network*, so packets with addresses of the same network could share a single routing table entry, and this would make the routing table much smaller and managable. How such information is obtained is beyond the scope of this paper and interested readers may consult routing algorithms for details, but the basic idea is to route the traffic on a network basis rather than a host basis; in fact, the goal of the Binary Tree Collapse is to further minimize the number of networks in a routing table without the loss of any useful information.

There are different ways to represent a network address, and the prefix approach is used throughout this paper and the related algorithms. The prefix approach works as follows: given an IPv4 address, if the prefix length is 32, it is a host address, otherwise it is a network address, and the network address is simply the prefix of the address concatenated with 0s. With the prefix approach, it is now possible to define a match given the incoming packet and the routing table database: first the prefix of the destination address of the incoming packet is obtained and the prefix length is the one indicated in the routing table entry; next 0s are concatenated to the prefix to form a IPv4 network address; finally if the new network address matches the destination address of the routing

table entry, a match is called. It would be tedious if we have to find the match one by one, and there are actually fast algorithms[2] to accomplish the match. The prefix length of a routing table can be arbitrarily long, and this often results multiple matches. Note that only the match with the longest prefix is used. The next hop address associated with that match is taken and the packet is sent to the line connected to that next hop.

To build a binary tree we start with a pre-built root node. Adding an entry is to simply follow the path of the binary tree, and create new nodes on the way if necessary. Each node can be represented using a C structure which is defined as follows:

```
struct Node {
  Node *lc;
  Node *rc;
  Node *parent;
  unsigned int nextHop;
  int terminal;
  int range;
};
```

`lc`, `rc` and `parent` are left child pointer, right child pointer and parent pointer of the current node. During the creation of the binary tree, we will need to create many nodes which are on the path of the routing table entries, and they serve just as links between the root node and the final matching nodes. We call such nodes as non-terminal nodes, as oppose to the leave nodes corresponding to routing table entries, which in that case are named as terminal nodes. The field `terminal` specifies if the node is terminal or non-terminal, and it serves just as a flag. For those terminal nodes, the `nextHop` field specifies the next hop the packet is routed to, and it is type of IPv4(32-bit). For those non-terminal nodes, the `nextHop` field still has meaning and this will be discussed in Section 4. The `range` field also serves as a flag, and its meaning will be discussed in Section 5. This flag is actually ignored when performing the first two techniques.

After the binary tree is successfully built, we will begin to traverse the binary tree and use some techniques on it. We use post-order when traversing the binary tree and the reason becomes obvious when we proceed to the next section. A total of two passes of the binary tree are performed where during the first pass, the first two techniques are used and the third technique is used in the second pass. The result of this algorithm is still a binary tree but with less nodes marked terminal. Note that each terminal node corresponds to a routing table entry, so less terminal nodes means a reduction of the size of the routing table. A number of things can benefit from such a reduction; there is less storage space needed to keep all the routing information; routing algorithm periodically transmitting its routing tables to its neighbors will have less information to send, and this may lead savings on communication bandwidth; since the speed of certain routing table lookup algorithms have a correlation with the size of the routing table, with a smaller routing table, the speed of such algorithms can further go up.

Now let us begin with the easiest technique.

3. Technique One: Same Children Property Collapse

This technique is the most obvious one and is illustrated in figure 1.

The rule of this technique is that if a node has both left child and right child, and both children are terminal nodes with the same outlink, we can collapse these two children nodes and put their

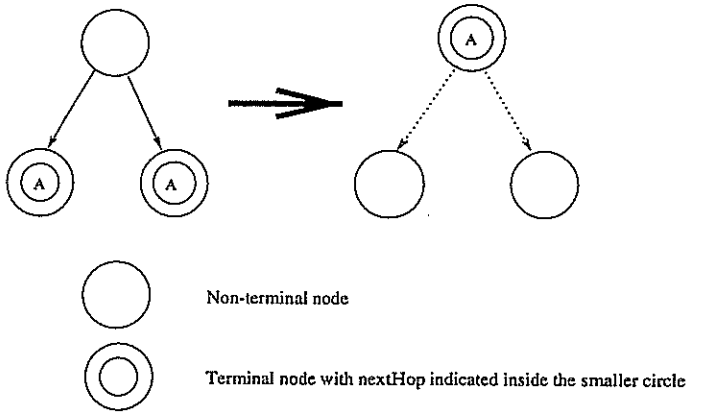

Figure 1: binary tree structure where first technique can be applied

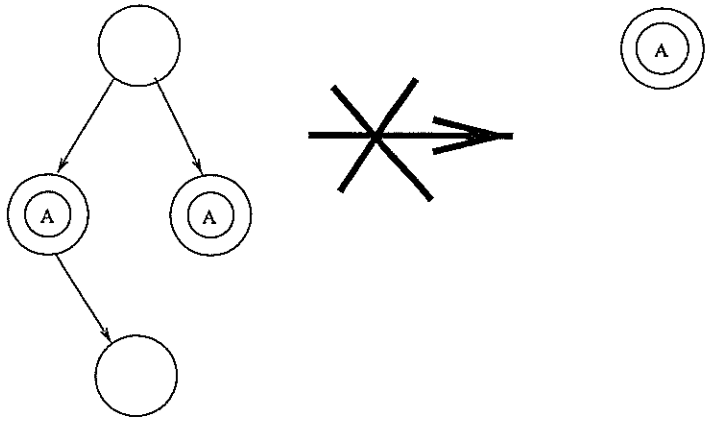

Figure 2: Why we cannot delete the two children nodes when doing the collapse

routing information at their parent node. Here we do not actually collapse in the sense of deleting two children, we just mark them as non-terminal nodes and mark the parent node as terminal node with the next hop address being their children's. The reason that an actual collapse may garble the routing database can be found in figure 2. Here deleting the two children will lose all the subtree of the parent node and will generate incorrect routing information. In the case of both children nodes are leaf nodes, an actual collapse can be done but again, the resulting routing table is the same as the one without doing so. The pseudo-code of the first technique is given as follows:

```
For each node in the binary tree
  if node has both left and right child then
    if node->lc->terminal = 1 and node->rc->terminal = 1
       and node->lc->nextHop = node->rc->nextHop then
      node->terminal := 1;
      node->nextHop := node->lc->nextHop;
      node->lc->terminal := node->rc->terminal := 0;
    end if;
  end if;
end for;
```

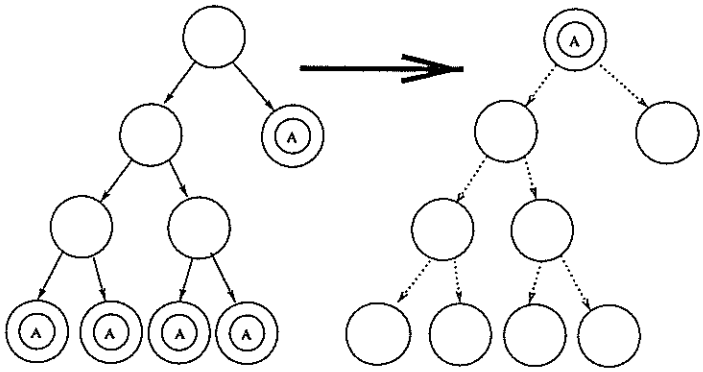

Figure 3: Why we need post-order traversal

Now it should become obvious why we use post-order to traverse the binary tree. Collapsing the binary tree is a recursive process and to continuously using technique 1, we must make sure that the higher level of nodes are visited last. Only post-order traversal can continuously collapse binary trees such as the one illustrated in figure 3.

4. Technique Two: Hierarchical Collapse

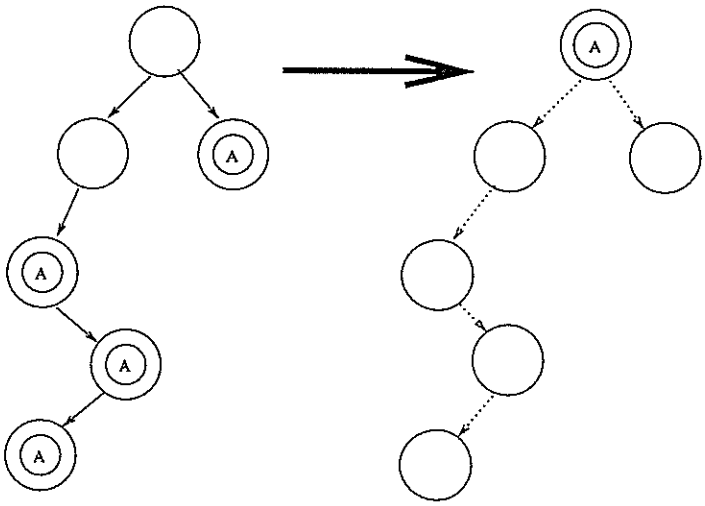

Figure 4: Simple case of hierarchical collapse

By observing the original routing table, we can always find many entries with the structure such as the one in figure 4 in the binary tree representation.

Obviously we can collapse the binary tree to a single node which is the root node at the binary tree in this example. It is easy to come up with a rule such as "collapse if the node has a single child which is a terminal node, mark the child node as non-terminal and put the routing information in the parent node." Unfortunately, this simple rule sometimes fails. A counter example is given in figure 5, which is just a bit altered structure of figure 4.

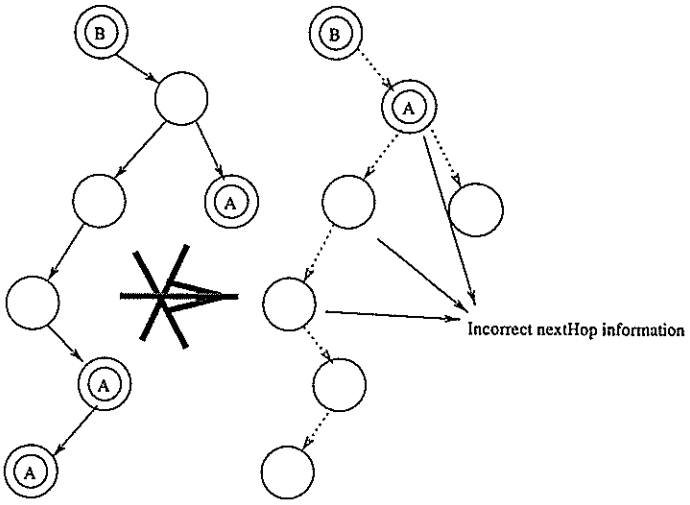

Figure 5: Incorrect use of the second technique

If we collapse all the way up to the top, the three intermediate nodes would have incorrect next hop addresses, which should still be B but now are As.

In fact, all we need to do is to change the above rule a little so that it can recognize situations such as the one shown in figure 5. When we discussed the data structure of the node in previous section, we mentioned that `nextHop` field has its meaning even for non-terminal nodes. The fact is that during the building of the binary tree, for each node we created, if it is a non-terminal node, we copy its parent's `nextHop` to this field. The root's `nextHop` address is initialized to a special value marking that no route to this address is known, or the default next hop address if there is one. The result is, when we try to match a packet's destination to its corresponding node in the binary tree, during the search at each node, we can always tell the current nearest match by checking the `nextHop` address. Therefore, to prevent the incorrect collapse of the above case, we change the rule to "collapse" if the node has a single child which is a terminal node, and both have the same next hop address. Mark the child node as non-terminal and put the routing information in the parent node. The pseudo-code of the second technique is given as follows:

```
For each node in the binary tree
  if node has a left child then
    if node->lc->terminal = 1 and node->lc->nextHop = node->nextHop then
      node->terminal := 1;
      node->lc->terminal :=0;
    end if;
  end if;
  repeat the above procedure for the right child;
end for;
```

5. Technique Three: Range Collapse

This technique is more complicated than the first two and can be best explained by an example. In one of the routing tables we have examined, we found the following entries:

```
24.48.0.0/20    B
24.48.33.0/24   A
24.48.34.0/23   A
24.48.36.0/22   A
24.48.40.0/21   A
24.48.48.0/22   A
24.48.52.0/23   A
24.48.54.0/24   A
```

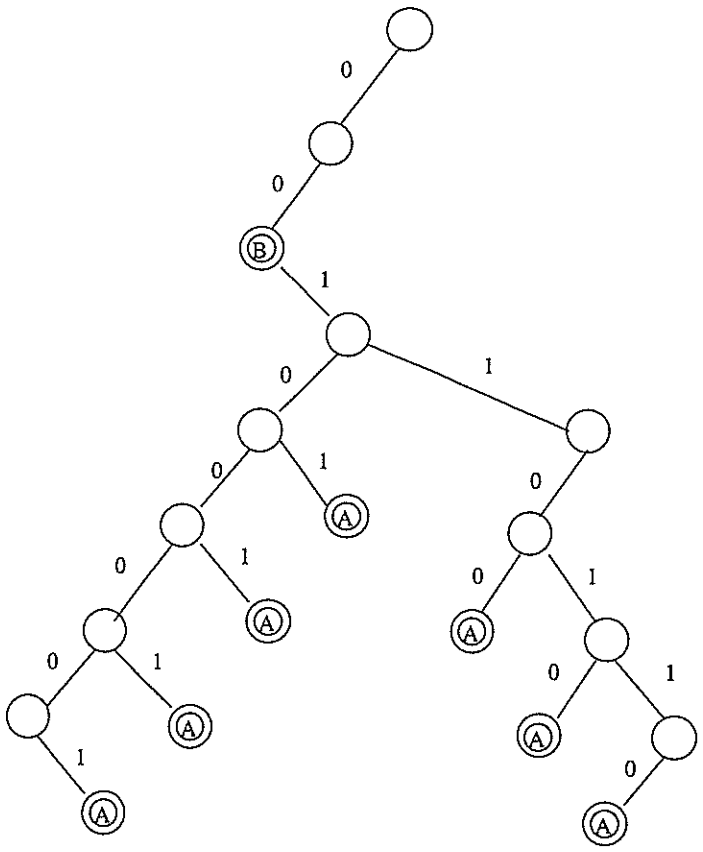

Figure 6: binary tree structure that can be applied by technique 3

The corresponding binary tree structure is shown in figure 6. By using the previous techniques on this sub-binary tree, we will get no compression at all, because none of those techniques can be applied here. However, by observing this structure carefully, we can find a range of addresses sharing the same next hop address. The problem preventing us from doing the collapse is that the binary tree is not a complete binary tree, in other words, the range of the addresses does not start from the leftmost leaf node and does not end at the rightmost leaf node. A collapse in this senario would alter the logical structure of the binary tree and would make the routing of those particular leaf nodes incorrect. Nevertheless, by first explicitly adding those leaf nodes, and mark them as terminal nodes with the next hop addresses being the ones they should be, we will then have a complete binary tree and can then start doing the collapse. The collapse is shown in figure 7 and 8, where figure 7 is the intermediate step.

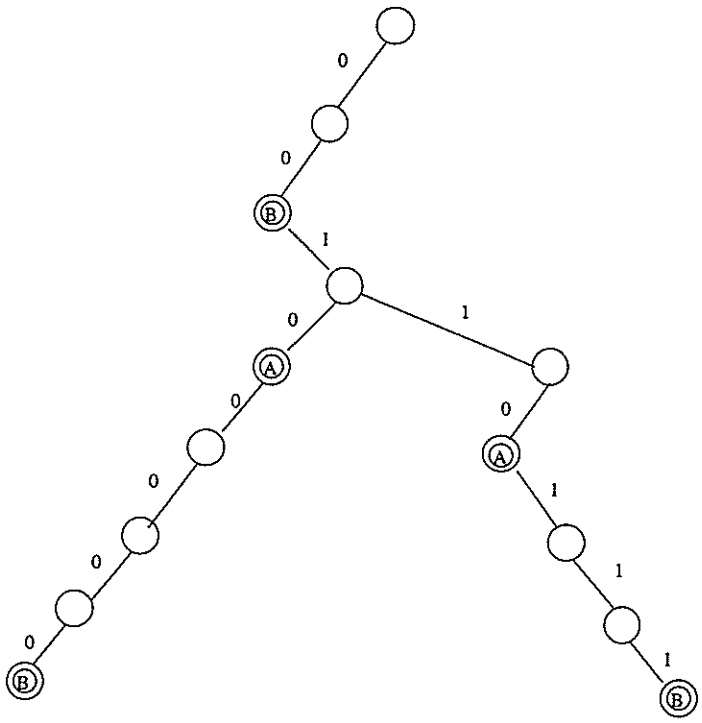

Figure 7: Intermediate step of applying technique 3

Here we call the explicitly added nodes as the "expansion nodes", and we call this technique as "range collapse".

Further observation tells us that this technique can also be used on structures such as the one shown in figure 9. Without loss of generality, the applied rule here is: for any node with a left child which is a terminal node with next hop address being equal to the node's left sibling(if it has one), and its left sibling is also a terminal node, then we can do a range collapse. If the node does not have a right child, create one(the expanded node), mark it terminal with next hop address being equal to its parent node's next hop address. In the case that there exists a right child node and the right child node is non-terminal, repeat the above procedure minus the node creation step. If the right child is a terminal node, nothing will be done. Now for the left child node, mark it non-terminal. Finally, mark the current node itself as being processed by setting the `range` flag of the node structure. Obviously, the above rule is symmetric so by changing the "left" to "right" and "right" to "left", we will be able to handle the other half of possibilities.

The `range` flag merely indicates that the node fits the above rules. This rule is yet to be complete as it does not say when to stop. So we need to add this: when the above rule does not hold and the left node's `range` flag is set, mark the right child non-terminal, mark the node itself terminal with next hop address being its right child's next hop address. Again, the stop rule is also symmetric. The psedo-code for the third technique is given as follows:

```
For each node in the binary tree, do the following in the post-order manner
  if node->lc->terminal = 1 and
     node->lsib->terminal = 1 and
     node->lc->nextHop = node->lsib->nextHop then
```

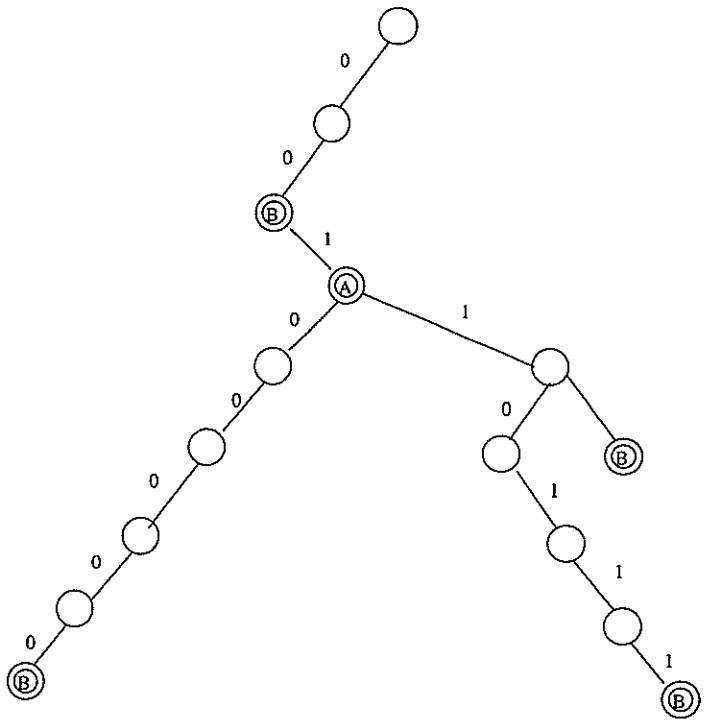

Figure 8: Final result of applying technique 3

```
    if node has no right child then
      create right child;
      node->rc->terminal := 1;
      node->rc->nextHop := node->nextHop;
    end if;

    node->lc->terminal := 0;
    node->range := 1;
  end if;
  repeat the above procedure for the other side;
end for;
```

Please note that in the above pseudo-code, the left sibling and right sibling pointers(`lsib` and `rsib`) are used to simply the code, while there is no such pointers defined in the actual node structure.

Now we have discussed all three techniques that we have devised for the binary tree collapse algorithm. The next section gives us some experiment results we have obtained by using the three techniques.

6. Experiment Results

We used the routing tables taken from Mae-East of January 16, 1998 as the test data and applied the algorithm to the routing tables. The original routing tables can be obtained from the World Wide Web at `http://www.merit.edu/ipma/routing_table/`, and the entries are categorized as class A,

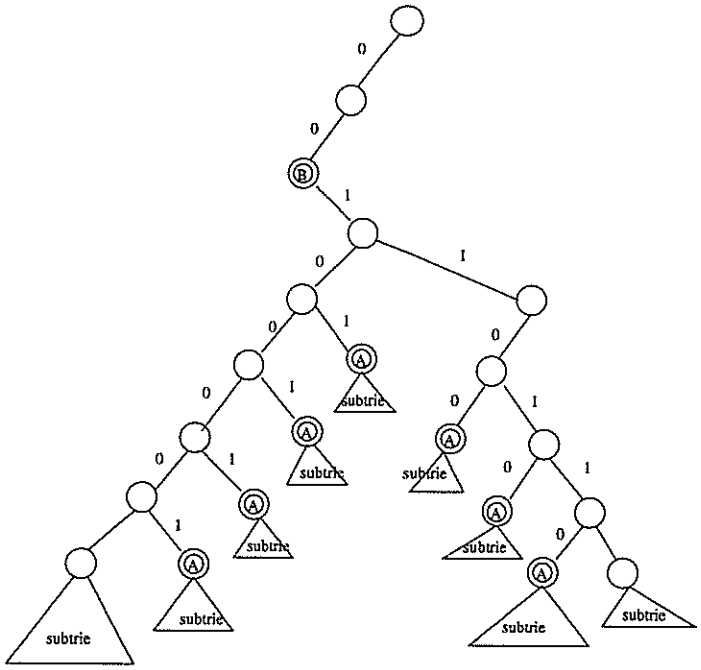

Figure 9: Variation of figure 6

B, C and Swamp. We did not merge all the routing tables and instead we applied the algorithm to the four routing tables individually. This actually helps us to find the different results we would obtain on different kind of routing tables. As stated before, the algorithm has two passes, with the first pass combining the first and second techniques and the second pass implementing the third technique. The results are shown in figure table 2. To show how effective each technique is, we have given the intermediate results also.

	Class A	Class B	Class C	Swamp D
Before Algorithm	145	4,232	28,694	5,474
After technique 1 and 2	105	3,636	17,157	4,593
Final results	105	3,586	16,910	4,467

Table 2: Routing table data from a major ISP

From the results we can clearly see that the algorithm has the biggest impact on class C, where the original routing table size is almost halved. It has less effects on class A, B and Swamp. This does make sense because the available address space of class C is smaller than class A and B, however, the number of addresses in use is quite large. These two factors make the routing table entries rather congregated and naturally we should be able obtain better compression ratio. We can also optimistically predict that as the Internet routing table becomes larger in size, we will find still better results from this algorithm.

As to the performance of the algorithm, it is very effcient. It takes a mid-range Pentium machine only a few seconds to compress a backbone routing table with fourty thousands entires. This makes it quite favorable to be integrated in the routing algorithm without any noticable performance loss, considering the routing table is usually updated at a much slower pace. Also note that to

accomodate routing table entry insertion and deletion, only very small changes need to be made to the original algorithm. For insertion, there is actually no change at all, the entry can be inserted in the straightforward way, and a collapse may occur later when running the algorithm. For routing table deletion, since we do not actually delete any routing table entry in our collapse process, the entry will be there and can be marked non-terminal. The thing that needs to be modified however, is to let all subsequent nodes on the path inherit possibly a new next hop address, if the next hop address of the deleted entry and its parent's next hop address are different. There is no further change needed and another collapse later may again modify the structure of the routing table binary tree.

Another interesting thing is technique one and two contribute most of the compression. The third technique, while quite elaborate, is not very effective at collapsing the binary tree. This reflects that the current Internet routing tables do not have too many occurrence of the structures as what we have shown in Section 5.

7. Summary and Future Work

This paper has introduced an algorithm which can be used to compress the Internet routing tables. It is based on the binary tree representation of the routing table and is most effective on the backbone class C routing tables, which has a big number of entries and all entries are tightly congregated. The algorithm consists of three techniques, with each collapsing a certain type of binary tree structure. The effects of these techniques are given and compared.

We believe that this algorithm can be easily integrated into general routing algorithms and can speed existing routing process while providing savings on storage and communication bandwidth.

References

[1] Andrew S. Tanenbaum, *Computer Networks, 3rd edition,* Prentice Hall, Inc.

[2] Marcel Waldvogel, George Varghese, Jon Turner, Bernhard Plattner, *Scalable High Speed IP Routing Lookups* , SIGCOM 97.

[3] Gary R. Wright, W. Richard Stevens, *TCP/IP Illustrated, Volume 2,* Addison Wesley, Inc. 1995.