ART
– Allotment Routing Table –
A Fast Free Multibit Trie Based Routing Table

Yoichi Hariguchi

Cisco Systems
170 West Tasman Dr.
San Jose, CA 95134
yoichi@cisco.com
2002/04/12 at 12:44

Abstract

After introduction of the Classless Inter-Domain Routing, several high speed longest prefix matching route lookup algorithms were proposed. Among them, it is known that multibit trie based routing tables, particularly the techniques called Controlled Prefix Expansion (CPE) and Smart Multi-Array Routing Table (SMART) have a low and deterministic search cost of modestly higher memory consumption. Their search cost is typically 3 to 4 routing table memory accesses for IPv4. However, no one can use them without licensing since they are patented or patent pending. This paper presents a new freely reusable multibit trie based routing table called Allotment Routing Table (ART) which is as fast as both the CPE and SMART. The IPv4 performance of one ART configuration is 10M lookups/sec for search, 800K routes/sec for insertion, and 2.04M routes/sec for deletion on a Pentium III 1GHz PC. This is 16 times in search, 50% in insertion, 4 times in deletion as fast as the BSD radix with 3 times more memory consumption; another ART configuration has the performance of more than 8 times in search, 2 times in inseartion, 50% in deletion as fast as the BSD radix with less memory consumption. This paper also describes the path compression technique. The algorithms of ART are in the public domain. The author's ART implementations and an ART implementation in the KAME IPv6 stack for the BSD kernel are also freely reusable.

1 Introduction

It is important to develop fast and scalable routing table algorithms because the size of the Internet routing table is growing rapidly [1] even after the introduction of the Classless Inter-Domain Routing (CIDR) [2]. The number of routes in a core router is almost 100,000 [3] at the time of this writing. It is known that the multibit trie based [4] routing tables have a very low and deterministic search cost. In particular, techniques called Controlled Prefix Expansion (CPE) [6] and Smart Multi-Array Routing Table (SMART) [7] are easy to implement as software since their algorithms are pretty simple. The problem is that the CPE is patented [5] and the SMART is patent pending so that no one can use them without licensing.

This paper describes a new routing table algorithms called Allotment Routing Table (ART). The ART is a freely usable multibit trie based routing table.

This work was performed while the author was with MAYAN Networks, Corp., San Jose, CA

2 Algorithms

2.1 Single Level

Assume the address length is 4 bits. We need a single level array to make an ART routing table in this case. It is easy to extend the single level algorithms to the multi-level algorithms that support arbitrary address length. Section 2.2 describes the multi-level algorithms.

Table 1 shows all the possible 31 prefixes in this case.

Table 1: 4-BIT ADDRESS LENGTH PREFIXES

0/0	0/4	3/4	5/4	8/1	9/4	12/2	14/3
0/1	1/4	4/2	6/3	8/2	10/3	12/3	14/4
0/2	2/3	4/3	6/4	8/3	10/4	12/4	15/4
0/3	2/4	4/4	7/4	8/4	11/4	13/4	

Here, the left hand side of '/' represents the destination address and the right hand side of '/' represents the prefix length, respectively. Note that some destination addresses can have different prefix lengths. For example, destination address 8 can have prefix length 1, 2, 3, or 4.

Now let us apply the following mapping function to each prefix:

baseIndex(w, a, l)
    return $(a \gg (w - l)) + (1 \ll l)$

Here, w is the address length in bit, a is the address, and l is the corresponding prefix length, respectively. For example,

w = 4, a = 8, and l = 1 for prefix 8/1. Let us call the return value of *baseIndex*() base index. Figure 1 illustrates the operation of *baseIndex*().

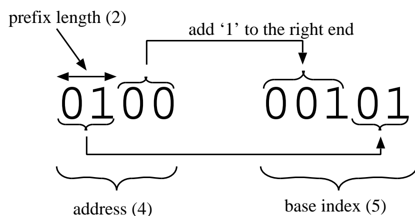


Figure 1: Operation of *baseIndex*()

Table 2 shows all the prefixes and their base indices.

Table 2: PREFIX (P) AND BASE INDEX (B)

P	B	P	B	P	B	P	B
0/0	1	2/3	9	1/4	17	9/4	25
0/1	2	4/3	10	2/4	18	10/4	26
8/1	3	6/3	11	3/4	19	11/4	27
0/2	4	8/3	12	4/4	20	12/4	28
4/2	5	10/3	13	5/4	21	13/4	29
8/2	6	12/3	14	6/4	22	14/4	30
12/2	7	14/3	15	7/4	23	15/4	31
0/3	8	0/4	16	8/4	24		

baseIndex() is one of two keys of the ART. Table 2 shows that all the possible route pointers can be stored in an array using a base index as an array index. At the same time, this array also forms a complete binary tree as shown in Figure 2. In other words, *baseIndex*() maps all the possible prefixes into a complete binary tree [8].

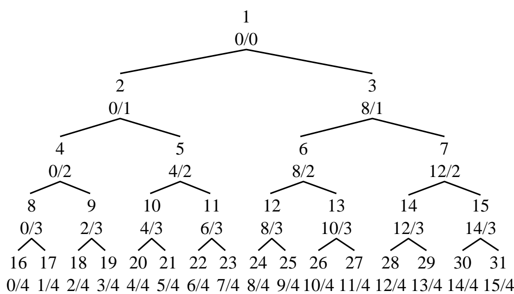


Figure 2: All prefixes mapped into complete binary tree

Let X be an array pointer and b a base index, respectively. It is important to note that $X[b \gg 1]$ always points to the next most specific route of $X[b]$ if it exists in Figure 2. This characteristics contributes to the deletion performance of ART.

Let r be a route pointer, $r \rightarrow a$ be the destination address of the route, $r \rightarrow l$ be the corresponding prefix length, respectively. Note that the bottom indices of the complete binary tree represent all the possible host routes in the address space. Let us call the bottom indices of the complete binary tree fringe indices. For example, indices 16..31 are fringe indices in Figure 2. When address a (0..15) is given, the corresponding fringe index is obtained by the following function:

fringeIndex(w, a)
    return baseIndex(w, a, w).

Here, w is the address length ($w = 4$ in this example). All the addresses (0..15) have the one-to-one mapping relationship with the corresponding fringe indices.

Inserting a route to the ART is equal to allotting a new route pointer to the corresponding base index and all of its child indices that do not have route pointers to more specific routes (see Fig. 3). As the result of insertion, the new route pointer is also alloted to the fringe indices corresponding to all the possible addresses of the new route as long as those fringe indices do not point to more specific routes than the new route.

That is why the route lookup function of the single level ART is as simple as follows:

lookup_s(X, w, a)
    return X[fringeIndex(w, a)].

Deleting a route pointed to by route pointer r from the ART is equal to allotting the next most specific route pointer to the indices whose value is r.

Function *allot*() in Algorithm 1 is another key of the ART. It allots a new route pointer r to base index b and all the child indices of b that have route pointer q.

Note that function *allot*() does not visit further child indices when the value of an index is not equal to q since it means that there is at least one more specific route than the route pointed to by q. This feature prevents redundant checking and increases the insertion and deletion performance.

Algorithm 1: Allotting route r (recursive)
Input: array pointer: X, smallest fringe index in X: t, base index b, old route pointer: q, new route pointer: r
Output:
allot(X, t, b, q, r)
(1) **if** $X[b] = q$ **then** $X[b] = r$ **else return**
(2) **if** $b \geq t$ **then return** /* b is a fringe index */
(3) $b \leftarrow b \ll 1$
(4) allot(X, t, b, q, r) /* Allot r to left children */
(5) $++b$
(6) allot(X, t, b, q, r) /* Allot r to right children */

Section 2.1.1, 2.1.2, and 2.1.3 give the examples and the algorithms of insertion, search, and deletion, respectively.

2.1.1 Insertion

Algorithm 2 shows the insertion algorithm for the single level ART.

Algorithm 2: Insertion algorithm (single level)
Input: array pointer: X, address length: w, address: a, prefix length: l, route pointer: r
Output: **true** if successful, **false** otherwise
insert_s(X, w, a, l, r)
(1) $b \leftarrow$ baseIndex(w, a, l)
(2) **if** $r \rightarrow a = X[b] \rightarrow a$ **and** $r \rightarrow l = X[b] \rightarrow l$ **then**
(3) **return false** /* Already occupied */
(4) **endif**
(5) allot(X, $1 \ll w$, b, $X[b]$, r)
(6) **return true**

Let us see how the ART insertion algorithm works with examples. Assume there is no routes in the ART and we insert a route to prefix 12/2. The insertion process is as follows:

1. *insert_s*() is called as *insert_s*$(X, 4, 12, 2, 12/2)$.
2. *insert_s*() calls *allot*() as *allot*$(X, 16, 7, \Lambda, 12/2)$.
3. *allot*() allots route pointer 12/2 to index 7 and all of its child indices (14, 15, and 28..31).

Here, Λ means NULL pointer. Figure 3-1 shows the ART after the route to prefix 12/2 is inserted. Now assume we insert a route to prefix 14/3. The insertion process is as follows:

1. *insert_s*() is called as *insert_s*$(X, 4, 14, 3, 14/3)$
2. *insert_s*() calls *allot*() as *allot*$(X, 16, 15, 12/2, 14/3)$
3. *allot*() sets $X[15]$ to 14/3 since $X[15] = 12/2$.
4. *allot*() visits the left child of index 15, which means that *allot*() calls itself recursively as *allot*$(X, 16, 30, 12/2, 14/3)$. The recursive call sets $X[30]$ to 14/3 and returns since index 30 is a fringe index.
5. *allot*() visits the right child of index 15, which means that *allot*() calls itself recursively as *allot*$(X, 16, 31, 12/2, 14/3)$. The recursive call sets $X[31]$ to 14/3 and returns since index 31 is a fringe index.

Figure 3-2 shows the ART after the route to prefix 14/3 is inserted. Now assume we insert a route to prefix 8/1. The insertion process is as follows:

1. *insert_s*() is called as *insert_s*$(X, 4, 8, 1, 8/1)$
2. *insert_s*() calls *allot*() as *allot*$(X, 16, 3, \Lambda, 8/1)$
3. *allot*() sets $X[3]$ to 8/1 since $X[3] = \Lambda$.
4. *allot*() visits the left child of index 3, which means that *allot*() calls itself recursively as *allot*$(X, 16, 6, \Lambda, 8/1)$. The recursive call eventually allots route pointer 8/1 to index 6 and all of its child indices (12, 13, 24..17) since their value is Λ.
5. *allot*() visits the right child of index 3, which means that *allot*() calls itself recursively as *allot*$(X, 16, 7, \Lambda, 8/1)$. The recursive call immediately returns since the value of $X[7]$ is not equal to Λ(which means that no child indices of index 7 has value Λ).

Figure 3-3 shows the ART after inserting a route to 8/1.

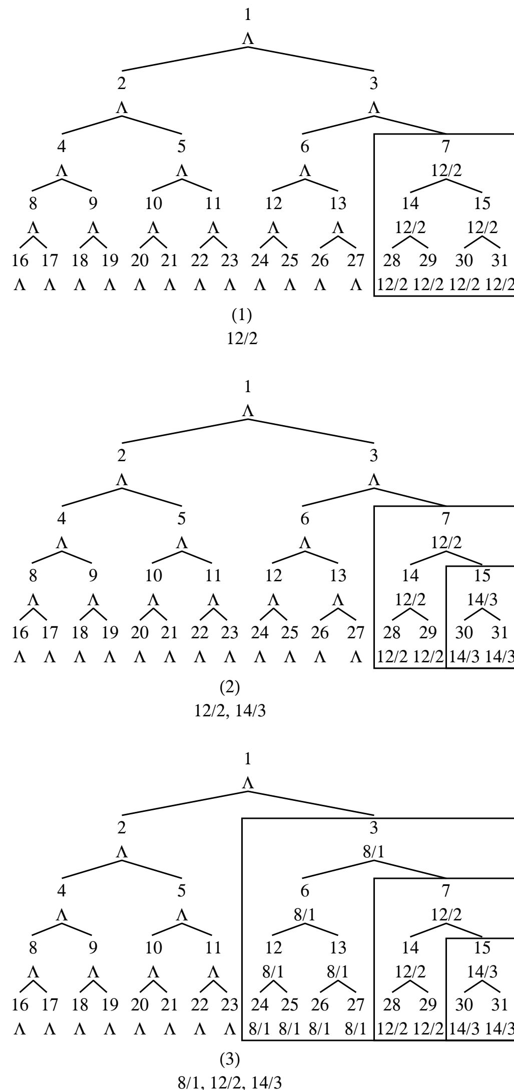


Figure 3: ART route insertion example

2.1.2 Search

Algorithm 3 shows the search algorithm for the single level ART.

Algorithm 3: Search algorithm (single level)
Input: array pointer: X, address length: w, address a
Output: Matched route pointer
lookup_s(X, w, a)
(1) **return** X[fringeIndex(w, a)]

In this example, Algorithm 3 becomes as follows:

lookup_s(X, 4, a)
    return $X[16 + a]$.

That is why *lookup_s*() returns Λwhen a is 0..7, 8/1 when a is 8..11, 12/2 when a is 12..13, and 14/3 when a is 14..15, respectively.

2.1.3 Deletion

Algorithm 4 shows the deletion algorithm for the single level ART.

Algorithm 4: Deletion algorithm (single level)
Input: array pointer: X, address length: w, address: a, prefix length: l
Output: Deleted route pointer if successful, Λotherwise
delete_s(X, w, a, l)
(1) $b \leftarrow$ baseIndex(w, a, l)
(2) **if** $X[b] = \Lambda$ **then**
(3)   **return** Λ /* No such route */
(4) **endif**
(5) allot(X, $1 \ll w$, b, $X[b]$, $X[b \gg 1]$)
(6) **return** r

When route pointer r (whose associated base index is b) is deleted, the value of indices in which r is stored must be replaced with the route pointer that points to the next most specific route. This process is necessary for all the multibit trie based routing tables. In the ART, the next most specific route can be always obtained with one memory access since it is stored in $X[b \gg 1]$. In addition, deleting the route pointed to by r from the ART is equal to allotting $X[b \gg 1]$ to the indices whose value is r.

Let us see how the ART deletion algorithm works with an example. Assume the route to prefix 12/2 is deleted from the ART in Figure 3-3. The deletion process is as follows:

1. *delete_s*() is called as *delete_s*($X, 4, 12, 2$).
2. *delete_s*() calls *allot*() as *allot*($X, 16, 7, 12/2, 8/1$).
3. *allot*() sets $X[7]$ to 8/1 since $X[7] = 12/2$.
4. *allot*() visits the left child of index 7, which means that *allot*() calls itself recursively as *allot*($X, 16, 14, 12/2, 8/1$). The recursive call eventually allots route pointer 8/1 to index 14 and all of its child indices (28, 29) since their value is 12/2.
5. *allot*() visits the right child of index 7, which means that *allot*() calls itself recursively as *allot*($X, 16, 15, 12/2, 8/1$). The recursive call immediately returns since the value of $X[15]$ is not equal to 12/2 (which means that no child indices of index 15 has value 12/2).

After the route to 12/2 is deleted, the ART returns to Figure 3-2.

2.2 Multiple Levels

The single level ART described in Section 2.1 requires an array that has 2^{w+1} indices to support address length w. For example, an array that has 2^{32+1} indices is necessary to build a single level ART for IPv4, which is not feasible. This problem can be solved by splitting the single address into multiple short addresses. Let us call the split addresses strides. This operation converts a single array into a multibit trie wherein a stride becomes a search key at each level. Let s_i be the stride length at level i (i.e., $w = \sum s_i$).

Now let us see how it works for IPv4 with an example. An IP address can be split into 4 strides each of whose stride length is 8 bits. Figure 4 shows a multi-level ART with routes to 10/14, 10.1/16, 10.1.2/23, and 11.1.2.2/31.

In the multi-level ART, each array element has a child array pointer (say pn) in addition to a route pointer (say pr). Let $X_n[i]$ be index i of array X at level n. If $X_n[i].pn$ is not equal to Λ, a child array is connected to index i and there are one or multiple of more specific routes in the descendent array(s). Function *allot*(), *baseIndex*(), and *fringeIndex*() described in Section 2.1 are applicable to the multi-level ART with no change. The multi-level ART insertion algorithm allocates new arrays if necessary and calls *insert_s*(); likewise the deletion algorithm calls *delete_s*() and frees arrays if necessary. There is one thing to consider when extending the single level search to support multiple levels; there may be the longest matching prefix even when the value of pr at a fringe index is Λ. Assume IPv4 address 10.1.4.5 is given to search the ART in Figure 4. The search ends at index 256+4 in the level 2 array that has route pointers to 10.1.2/23. The value of index 256+4 is Λ, but the longest matching prefix to 10.1.4.5 is 10.1/16. That is why the multi-level lookup function must remember the value of pr unless it is Λ each time it visits a child array. Algorithm 5, 6, and 7 show the insertion, deletion, and search algorithms for the multi-level ART. In Algorithm 5, 6, index 0 is used as a reference counter that contains the number of pr and pn in the array since the ART does not

use index 0. Note that the multi-level ART does not use index 1, either. This is because a route associated with index 1 is stored in the parent array.

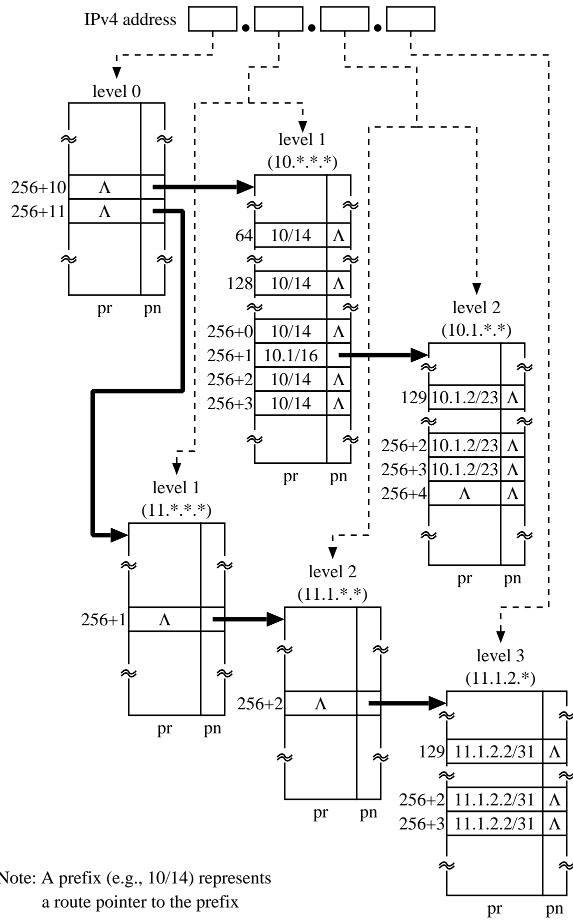


Figure 4: Multiple level ART for IPv4

Algorithm 5: Insertion algorithm (multi-level)
Input: level 0 array pointer: X_0, address length: w, stride length array pointer: sl, route pointer: r
Output: true if successful, **false** otherwise
insert(X_0, w, sl, r)
(1) Int $l \leftarrow 0$ /* level */
(2) Int i /* array index */
(3) Int s /* stride */
(4) Int $ss \leftarrow 0$ /* stride length summation */
(5) Array $X \leftarrow X_0$ /* level 0 array */
(6)
(7) **if** $r \rightarrow a = 0$ **and** $r \rightarrow l = 0$ **then**
(8) **if** $X[1] \neq \Lambda$ **then return false**
(9) $X[1] \leftarrow r$ /* default route */
(10) **return true**
(11) **endif**
(12) **while true**
(13) $ss \leftarrow ss + sl[l]$
(14) $s \leftarrow (r \rightarrow a \gg (w - ss)) \& ((1 \ll sl[l]) - 1)$
(15) **if** $r \rightarrow l \leq ss$ **then break**
(16) i = fringeIndex($sl[l]$, s)
(17) **if** $X[i].pn = \Lambda$ **then**
(18) $X[i].pn \leftarrow$ New Array /* array allocation */
(19) $X[0].pn \leftarrow X[0].pn + 1$ /* ref. counter */
(20) **endif**
(21) $X \leftarrow X[i].pn$
(22) $l \leftarrow l + 1$
(23) **endwhile**
(24)
(25) $ss \leftarrow ss - sl[l]$
(26) **if** insert_s(X, $sl[l]$, s, $r \rightarrow l - ss$, r) = **true then**
(27) $X[0].pn \leftarrow X[0].pn + 1$ /* new route entry */
(28) **return true**
(29) **endif**
(30) **return false**

Algorithm 6: Deletion algorithm (multi-level)
Input: level 0 array pointer: X_0, address length: w, stride length array pointer: sl, destination address: a, corresponding prefix length: pl
Output: true if successful, **false** otherwise
delete(X_0, w, sl, a, pl)
(1) Array $X \leftarrow X_0$ /* level 0 array */
(2) Array $Xsv[0] \leftarrow X$ /* parent array pointers */
(3) Int $ss \leftarrow 0$ /* stride length summation */
(4) Int s /* stride */
(5) Int $l \leftarrow 0$ /* level */
(6) Int $i \leftarrow 0$ /* index */
(7) Int $isv[]$ /* parent indices */
(8) RoutePointer r
(9)
(10) **if** $a = 0$ **and** $pl = 0$ **then**
(11) **if** $X_0[1].pr = \Lambda$ **then return false**
(12) $X_0[1].pr \leftarrow \Lambda$
(13) Return $X_0[1].pr$
(14) **endif**
(15)
(16) **while true**
(17) $ss \leftarrow ss + sl[l]$
(18) $s \leftarrow (r \rightarrow a \gg (w - ss)) \& ((1 \ll sl[l]) - 1)$
(19) **if** $pl \leq ss$ **then break**
(20) i = fringeIndex($sl[l]$, s)
(21) $isv[l] = i$
(22) **if** $X[i].pn = \Lambda$ **then return false**
(23) $Xsv[l] = X$
(24) $X \leftarrow X[i].pn$
(25) $l \leftarrow l + 1$
(26) **endwhile**

```
(27)ss ← ss − sl[l]
(28)r ← delete_s(X, sl[l], s, pl − ss)
(29)if r = Λ then return false
(30)
(31)/* Free array(s) if necessary */
(32)X[0].pn ← X[0].pn − 1
(33)if l > 0  and X[0].pn = 0 then
(34)     while true
(35)          Free X /* free current array */
(36)          l ← l − 1 /* get parent level */
(37)          X ← Xsv[l] /* get parent array pointer */
(38)          /* child array is deleted */
(39)          X[0].pn ← X[0].pn − 1
(40)          if l ≤ 0  or X[0].pn > 0 then
(41)               return r
(42)          endif
(43)     endwhile
(44)endif
(45)return r
```

Algorithm 7: Search algorithm (multi-level)
Input: level 0 array pointer: X_0, address length: w, stride length array pointer: sl, search key address: a
Output: longest prefix matching route pointer or Λ

```
search(X0, w, sl, a)
(1)  RoutePointer lmr ← X0[1].pr
(2)  Array X ← X0 /* level 0 array */
(3)  Int ss ← 0 /* stride length summation */
(4)  Int s /* stride */
(5)  Int l ← 0 /* level */
(6)  Int i ← 0 /* index */
(7)
(8)  while true
(9)       ss ← ss + sl[l]
(10)      s ← (a ≫ (w − ss))&((1 ≪ sl[l]) − 1)
(11)      i = fringeIndex(sl[l], s)
(12)      if X[i].pn ≠ Λ then
(13)           /* update current longest matching route */
(14)           if X[i].pr ≠ Λ then lmr = X[i].pr
(15)           X ← X[i].pn
(16)           l ← l + 1
(17)      else if X[i].pr ≠ Λ then
(18)           return X[i].pr
(19)      else
(20)           return lmr /* pr = pn = Λ */
(21)      endif
(22)endwhile
```

3 Optimizations

This section discusses some optimization techniques to reduce memory usage and to increase insertion/deletion performance that the author's ART implementation [11] uses. These techniques are generic enough to apply to any multibit tries.

3.1 Element Consolidation

The data structure described in Section 2.2 has two pointers (*pr* and *pn*) per array element. These pointers can be stored in a single location as shown in Figure 5.

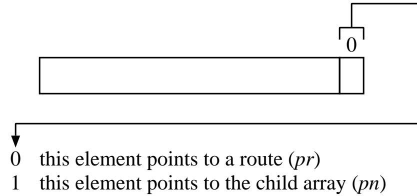


Figure 5: Consolidated element

The data structure in Figure 5 is based on the assumption that neither a route entry nor an array starts at an odd address. This technique reduces the array size to half. It is possible that both *pr* and *pn* exist in the same index. For example, index 256+1 of a level 1 array in Figure 4 has route pointer to 10.1/16 and child array pointer. In such a case, *pr* is stored in index 1 of the child array (Figure 6). Remember index 1 is not used.

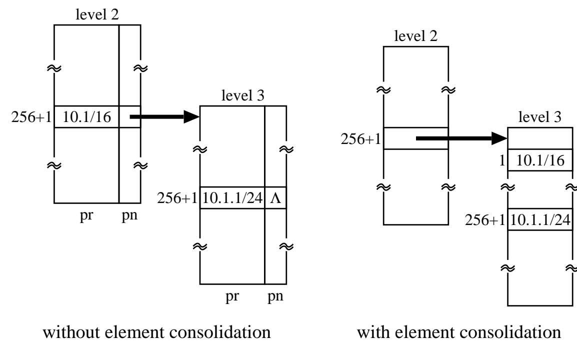


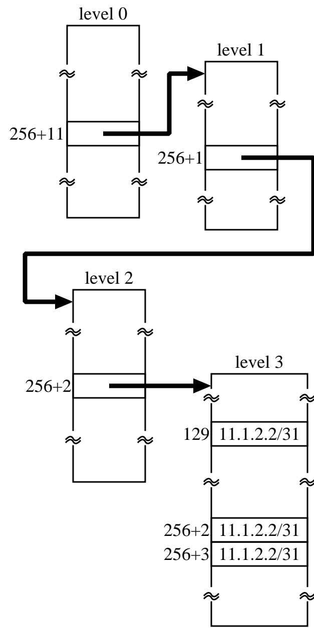
Figure 6: Routes to 10.1/16 and 10.1.1/24 in ART

The algorithms that support element consolidation are shown in Appendix A.

3.2 Path Compression

Path compression is a well-known trie compression technique. The idea is to remove arrays that have only one child array pointer (Figure 7). Let us call such an array transit array. A path compressed binary trie is often referred to as Patricia trie [9].

Path Compression reduces memory usage and increases search performance in some cases, but not always. It depends on the prefix length distribution, address length, and



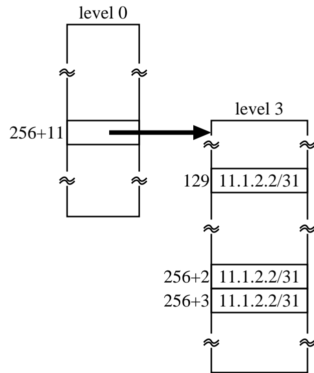


Figure 7: Path compression

the path length. Another drawback of path compression is that it is necessary to compare the search key with the key in the matched entry after a match is found. Assume IPv4 address 11.20.20.3 is given to search the path compressed ART in Figure 7. The search successfully ends at index 256+3 of the level 3 array since both level 0 and 3 fringe indices of 11.20.20.3 are the same as those of 11.1.2.2/31. However, 11.1.2.2/31 is not the correct destination prefix to 11.20.20.3. That is why it is necessary to compare two addresses after a match is found. This overhead becomes negligible if a path is long enough and there are many transit arrays in the path. On the other hand, this overhead slows down the search performance when a path is short or the number of transit arrays in the path is small. Section 4 shows how much path compression is effective for IPv4 and IPv6.

The ART algorithms with path compression is not shown in this paper for the space reason, but the entire source code can be obtained from [11].

3.3 Avoiding Recursion

Function *allot*() is one of two keys of the ART and it uses recursion. Changing the *allot*() algorithm from recursion to loop increases insertion and deletion performance. Algorithm 8 shows a non-recursive version of *allot*() with element consolidation.

Algorithm 8: Allotting route r (non-recursive)
Input: array pointer: X, smallest fringe index in X: t, base index: b, old route pointer: q, new route pointer: r
Output:
allot(X, t, b, q, r)
(1) ArrayPointer Y
(2) Int $j \leftarrow b$
(3)
(4) **if** $j \geq t$ **then**
(5) **if** $(X[b]\&1) = 1$ **then**
(6) $Y \leftarrow X[b]\&(\sim 1)$
(7) **if** $Y[1] = q$ **then** $Y[1] \leftarrow r$
(8) **else if** $X[b] = q$ **then**
(9) $X[b] \leftarrow r$
(10) **endif**
(11) **return**
(12)**endif**
(13)
(14) startChange:
(15) $j \leftarrow j \ll 1$
(16) **if** $j < t$ **then goto** nonFringe
(17) /* Handle fringe indices */
(18) **while true**
(19) **if** $(X[b]\&1) = 1$ **then**
(20) $Y \leftarrow X[b]\&(\sim 1)$
(21) **if** $Y[1] = q$ **then** $Y[1] \leftarrow r$
(22) **else if** $X[b] = q$ **then**
(23) $X[b] \leftarrow r$
(24) **endif**
(25) **if** $(j\&1) = 1$ **then goto** moveUp
(26) $j \leftarrow j + 1$
(27) **endwhile**
(28)
(29) nonFringe:
(30) **if** $X[j] = q$ **then goto** startChange

(31) moveOn:
(32) **if** $(j\&1) = 1$ **then goto** moveUp
(33) $j \leftarrow j + 1$
(34) **goto** nonFringe
(35) moveUp:
(36) $j \leftarrow j >> 1$
(37) $X[j] \leftarrow r$ /* Handle non-fringe node */
(38) **if** $j \neq b$ **then goto** moveOn

4 Measurements and Comparison

This section describes the performance of the ART by simulations. The simulations are performed on a Pentium III 1GHz CPU running Linux 2.1.14. The routing table used for IPv4 simulations is a combination of the MAE East routing table (42,366 routes on Aug. 17, 1999) [10] and 4,000 randomly generated routes whose prefix lengths are longer than 24. This is because the longest prefix length for inter-AS routes like stored in the MAE East routing table is 24 (Figure 8). However, the routing tables in large ISPs have both inter-AS routes and intra-AS routes. Hence, it is better to use the routing table that has both types of routes.

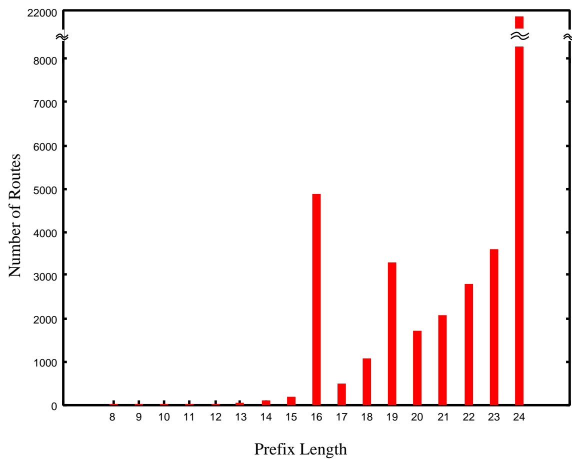


Figure 8: Route distribution of MAE East (8/17/1999)

The routing table used for IPv6 simulations is generated from the MAE East routing table as follows:

1. TLA ID: a 13-bit random number
2. NLA ID: leading 24-bit of an IPv4 address
3. prefix length: 24 + prefix length of the corresponding IPv4 address

This is the hardest condition from the point of view of the size of routing table. In IPv6, the leading 48 bits are used for routing among and within ISPs [12]. That is why there is no need to look up more than 48 bits.

The source code and prefix data used in this section can be obtained from [11].

4.1 IPv4 Performance

Table 3 shows the IPv4 performance comparison among the ART that supports single stride length distribution which is 16 bits for level 0, 8 bits for level 1 and 2 (say 16-8-8), the ART that supports arbitrary stride length distribution (and configured to the 16-8-8 stride length distribution), SMART, CPE, and BSD radix [13]. Both ART implementations use element consolidation.

Table 3: PERFORMANCE COMPARISON

	Insertion (K routes/s)	Deletion (K routes/s)	Search (M lookups/s)
ART-16-8-8	800	2041	10.00
ART	794	758	6.25
SMART	900	943	6.25
CPE	943	676	5.55
BSD Radix	544	515	0.63

All the routing tables except BSD Radix has the 16-8-8 stride length distribution. ART-16-8-8 supports only 16-8-8 stride length distribution. ART, SMART, and CPE support arbitrary stride length distribution and configured to 16-8-8. Random IP addresses are used for search after 46,366 routes are inserted.

ART-16-8-8 has the best performance since it is specially tuned for the fixed stride length distribution so that it does not have any loop. The search performance of ART is comparable to SMART and CPE. The reason ART has lower insertion performance compared to SMART and CPE is that the ART insertion algorithm requires more number of memory writes to an array compared to the other two. CPE has the least deletion performance among three because of its overhead of finding the next most specific route. SMART has better deletion performance than ART because the SMART deletion algorithm requires less number of memory writes than that the ART deletion algorithm.

Table 4: MEMORY CONSUMPTION (MB)

ART-16-8-8	ART	SMART	CPE	BSD Radix
17.07	17.07	17.42	17.67	5.54

CPE uses almost the same amount of memory as ART and SMART even though it theoretically requires half amount of memory compared to ART and SMART. This is because CPE does not use element consolidation (paper [6] does not mention it either).

4.2 Stride Length Distribution and Performance

4.2.1 IPv6

Figures 9 to 12 show the simulation results about stride length distribution and performance for IPv6. The theoretical worst case cost of insertion and deletion is mainly controlled by the maximum stride length. The actual performance however depends on the prefix length distribution of the routing table. As for the IPv6 routing table used in this paper, more than 60% of the routes in the routing table has prefix length 48.

The insertion performance increases as the stride length decreases up to some point (20-4x7 for simple trie, 16-4x8 for path compressed trie) and decreases after that. It means that the stride length for the /48 prefixes is the main factor at first, then the cost of following a path becomes the main factor. The insertion performance of a path compressed trie is higher than that of a simple trie, particularly when the path length becomes longer. This is because a path compressed trie usually allocates only one array at insertion. On the other hand, simple trie may have to allocate multiple arrays at insertion.

The deletion performance of a simple trie sharply decreases as the number of strides grows. This suggests that the number of routing table accesses at deletion increases as the number of strides grows, which means that the prefix length within an array becomes longer as the number of strides grows. The deletion performance of a path compressed trie also decreases as the number of strides grows, but the degree of decrease is modest. This suggests that the path length factor also contributes to the deletion performance.

The search performance of a path compressed trie is relatively independent from the number of strides. The search performance of a simple trie is also relatively independent from the number of strides up to 24-4x8, but it decreases as the number of strides grows after that. These results are understandable as the characteristics of path compressed trie and simple trie.

The memory consumption of both tries exponentially decreases as the number of stride length increases. One exception is 24-4x6 because level 0 array of 24-4x6 requires 128MB of memory. The memory consumption of a pass compressed trie is half as large as that of a simple trie when the stride length of level 0 array is not 24.

The simulation results show that a path compressed trie is better than a simple trie in all aspects for IPv6.

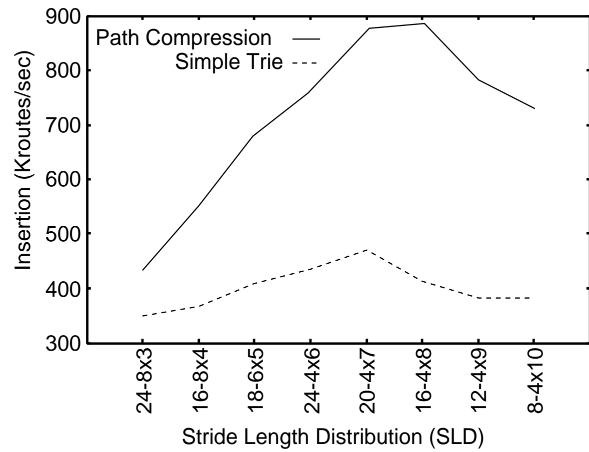


Figure 9: SLD vs. performance (insertion)

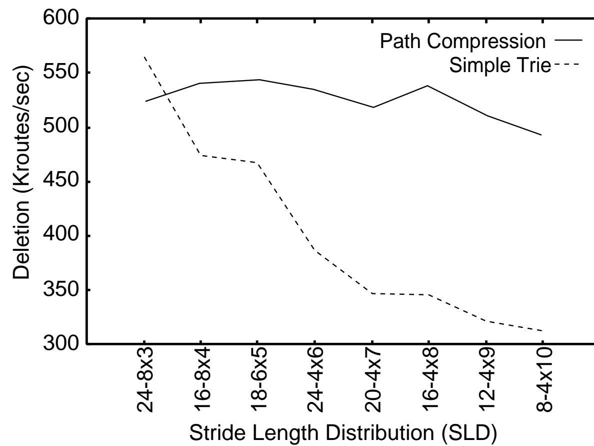


Figure 10: SLD vs. performance (deletion)

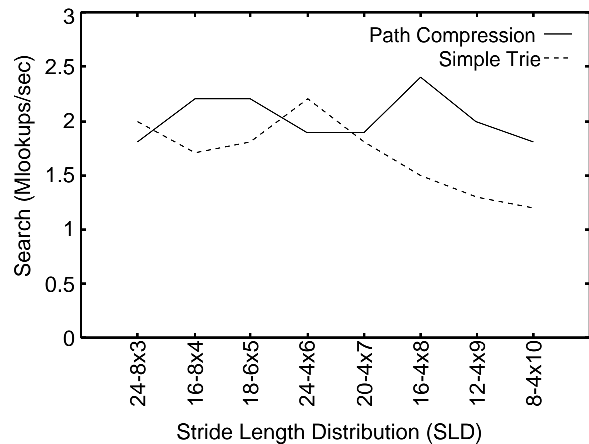


Figure 11: SLD vs. performance (search)

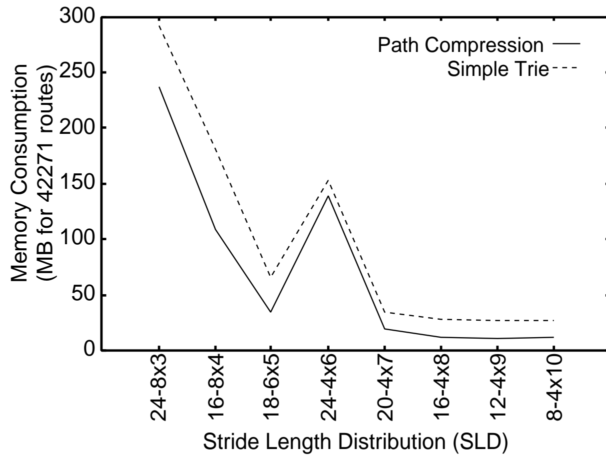


Figure 12: SLD vs. memory consumption

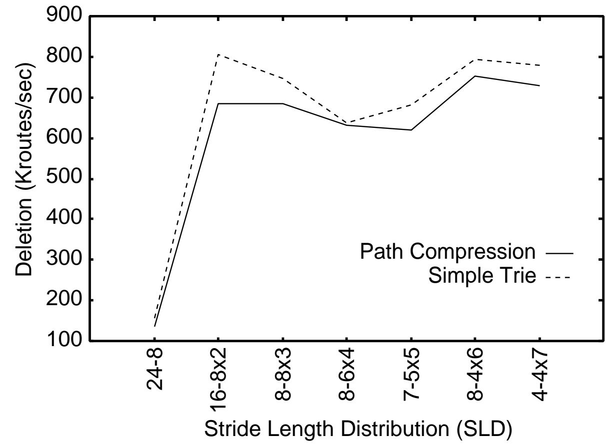


Figure 14: SLD vs. performance (deletion)

4.2.2 IPv4

Figures 13 to 16 show the simulation results about stride length distribution and performance for IPv4. The simulation results show that a path compressed trie does not have any advantage to a simple trie in all aspects for IPv4. Actually the simple trie shows better performance in some cases. It means that the effect of pass compression highly depends on the address length.

The memory consumption of both tries exponentially decreases as the number of stride length increases in IPv4 as well as IPv6. Please note that the simple trie with the 8-4x6 stride length distribution still has 3.8 Mlookups/sec search performance, which is 6 times faster than BSD radix, with less memory consumption.

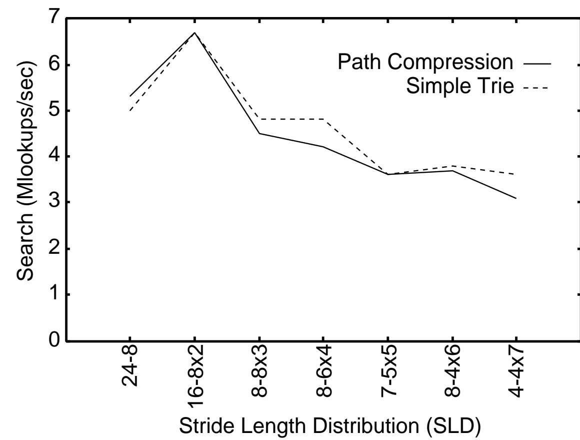


Figure 15: SLD vs. performance (search)

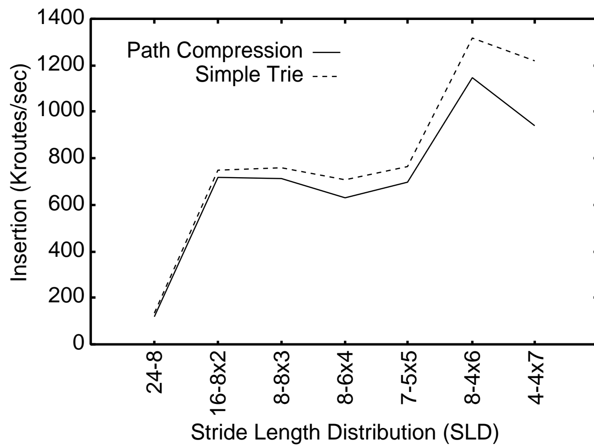


Figure 13: SLD vs. performance (insertion)

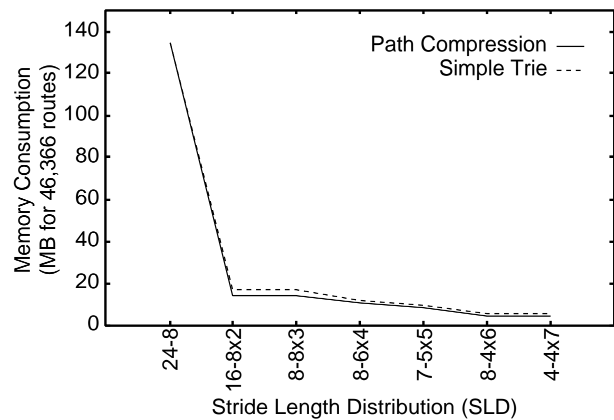


Figure 16: SLD vs. memory consumption

5 Conclusion

This paper presented a multibit trie based routing table called ART. This paper showed that the ART has good performance for all three routing table operations with both theory and simulation. This paper also showed that for IPv6, path compression reduces memory consumption and improves the performance of all three routing table operations, and for IPv4, it does not have any advantages to all three routing table operations. The ART algorithms are freely usable. There are two free ART implementations available at the time of this writing. The one is used for the simulations in this paper. It supports arbitrary address length and prefix length distribution. The other is used in the KAME IPv6 protocol stack for BSD kernels.

Acknowledgement

The ART was invented by Donald E. Knuth while he was reviewing author's paper [7]. The author would like to thank Professor Knuth for allowing the author to write a paper about his invention. The author would also like to thank Dr. Steve Deering for his suggestion to add path compression, Immanuel Rahardja, Dr. Jun-ichiro "itojun" Hagino, and Kenji Rikitake for their useful comments on this paper. Lastly, the author would like to thank Cisco Systems, particularly Dr. John Wakerly who allowed the author to publish this paper.

References

[1] Scott Marcus, *IPv4 Address Space Allocation and Usage Trends*, http://www.nanog.org/mtg-0105/ppt/marcus.ppt.

[2] V. Fuller, T. Li, J. Yu, and K. Varadhan, *Classless Inter-Domain Routing (CIDR)*, RFC1519, September 1993.

[3] Merit Networks, Inc., *Internet Routing Trends* http://www.telstra.net/ops/bgptable.html

[4] D. Knuth, *The Art of Computer Programming Vol.3*, Addison Wesley, 492-494

[5] V. Srinivasan and George Varghese, *Method and apparatus for fast hierarchical address lookup using controlled expansion of prefixes*, U.S. Patent 6,011,79

[6] V. Srinivasan and George Varghese, *Faster IP Lookups using Controlled Prefix Expansion*, Proceedings of ACM Sigmetrics, September 98 and ACM TOCS 99.

[7] Yoichi Hariguchi, *Smart Multi-Array Routing Table*, Proceedings of INET2001, June 2001. http://www.isoc.org/inet2001/CD_proceedings/7/smart.pdf

[8] D. Knuth, *The Art of Computer Programming Vol.3*, Addison Wesley, 144-149.

[9] D. R. Morrison, *PATRICIA – practical algorithm to retrieve information coded in alphanumeric.*, Journal of the ACM 15, 1968, 514–534.

[10] Internet Performance Measurement and AnalysisProject, *Internet Routing Table Statistics*, http://www.merit.edu/ipma/routing_table/

[11] Yoichi Hariguchi, *ART – Allotment Routing Table –*, http://www.yottanet.com:8080/art/

[12] R. Hinden, M. O'Dell, S. Deering, *An IPv6 Aggregatable Global Unicast Address Format*, RFC2374, July 1998.

[13] FreeBSD 2.2.2, /usr/src/sys/net/radix.[ch].

Appendix

A Algorithms with Element Consolidation

Algorithm 9: Insertion algorithm (single level)
Input: array pointer: X, address length: w, address: a, prefix length: l, route pointer: r
Output: **true** if successful, **false** otherwise
insert_s(X, w, a, l, r)
(1) RoutePointer q
(2)
(3) $b \leftarrow$ baseIndex(w, a, l)
(4) **if** $X[b]\&1 = 0$ **then**
(5) $\quad q \leftarrow X[b]$
(6) **else**
(7) $\quad q \leftarrow (X[b]\&(\sim 1))[1]$
(8) **endif**
(9) **if** $r{\rightarrow}a = q{\rightarrow}a$ **and** $r{\rightarrow}l = q{\rightarrow}l$ **then**
(10) $\quad$ **return false** /* Already occupied */
(11) **endif**
(12) allot(X, $1 \ll w$, b, q, r)
(13) **return true**

Algorithm 10: Deletion algorithm (single level)
Input: array pointer: X, address length: w, address: a, prefix length: l
Output: Deleted route pointer if successful, Λotherwise
delete_s(X, w, a, l)
(1) RoutePointer r
(2)
(3) $b \leftarrow$ baseIndex(w, a, l)
(4) **if** $X[b]\&1 = 0$ **then**
(5) $\quad r \leftarrow X[b]$
(6) **else**
(7) $\quad r \leftarrow (X[b]\&(\sim 1))[1]$
(8) **endif**
(9) **if** $r = \Lambda$ **then**
(10) $\quad$ **return** Λ/* No such route */
(11) **endif**
(12)
(13) allot(X, t, b, r, $X[b \gg 1]$)
(14) **return** r

Algorithm 11: Insertion algorithm (multi-level)
Input: level 0 array pointer: X_0, address length: w, stride length array pointer: sl, route pointer: r
Output: **true** if successful, **false** otherwise
insert(X_0, w, sl, r)
(1) RoutePointer q
(2) Int i /* array index */
(3) Int s /* stride */
(4) Int $l \leftarrow 0$ /* level */
(5) Int $ss \leftarrow 0$ /* stride length summation */
(6) Array $X \leftarrow X_0$ /* level 0 array */
(7)
(8) **if** $r{\rightarrow}a = 0$ **and** $r{\rightarrow}l = 0$ **then**
(9) $\quad$ **if** $X[1] \neq \Lambda$ **then return false**
(10) $\quad X[1] = r$ /* default route */
(11) $\quad$ **return true**
(12) **endif**
(13)
(14) **while true**
(15) $\quad ss \leftarrow ss + sl[l]$
(16) $\quad s \leftarrow (r{\rightarrow}a \gg (w - ss))\&((1 \ll sl[l]) - 1)$
(17) $\quad$ **if** $r{\rightarrow}l \leq ss$ **then break**
(18) $\quad i =$ fringeIndex($sl[l]$, s)
(19) $\quad$ **if** $X[i]\&1 = 0$ **then**
(20) $\quad\quad q = X[i]\&(\sim 1)$ /* save route pointer */
(21) $\quad\quad X[i] \leftarrow$ New Array |1 /* array allocation */
(22) $\quad\quad (X[i]\&(\sim 1))[1] = q$
(23) $\quad\quad X[0] \leftarrow X[0] + 1$ /* ref. counter */
(24) $\quad$ **endif**
(25) $\quad X \leftarrow X[i]\&(\sim 1)$
(26) $\quad l \leftarrow l + 1$
(27) **endwhile**
(28)
(29) $ss \leftarrow ss - sl[l]$
(30) **if** insert_s(X, $sl[l]$, s, $r{\rightarrow}l - ss$, r) = **true then**
(31) $\quad X[0] \leftarrow X[0] + 1$ /* new route entry */
(32) $\quad$ **return true**
(33) **endif**
(34) **return false**

Algorithm 12: Deletion algorithm (multi-level)
Input: level 0 array pointer: X_0, address length: w, stride length array pointer: sl, destination address: a, corresponding prefix length: pl
Output: **true** if successful, **false** otherwise
delete(X_0, w, sl, a, pl)
(1) Array $X \leftarrow X_0$ /* level 0 array */
(2) Array $Xsv[0] \leftarrow X$ /* parent array pointers */
(3) Int $ss \leftarrow 0$ /* stride length summation */
(4) Int s /* stride */
(5) Int $l \leftarrow 0$ /* level */
(6) Int $i \leftarrow 0$ /* index */
(7) Int $isv[]$ /* parent indices */

```
(8)  RoutePointer r
(9)
(10) if a = 0  and pl = 0 then
(11)     if X_0[1] = Λ then return false
(12)     X_0[1] ← Λ
(13)     Return X_0[1]
(14) endif
(15)
(16) while true
(17)     ss ← ss + sl[l]
(18)     s ← (r→a ≫ (w − ss))&((1 ≪ sl[l]) − 1)
(19)     if pl ≤ ss then break
(20)     i = fringeIndex(sl[l], s)
(21)     isv[l] = i
(22)     if (X[i]&1 = 0 then return false
(23)     Xsv[l] = X[i]&(∼1)
(24)     X ← Xsv[l]
(25)     l ← l + 1
(26) endwhile
(27)
(28) ss ← ss − sl[l]
(29) r ← delete_s(X, sl[l], s,pl - ss)
(30) if r = Λ then return false
(31)
(32) /* Free route entry and arrays */
(33) X[0] ← X[0] − 1
(34) if l > 0  and X[0] = 0 then
(35)     while true
(36)         Free X /* free current array */
(37)         l ← l − 1 /* get parent level */
(38)         X ← Xsv[l] /* get parent array pointer */
(39)         /* child array is deleted */
(40)         X[0] ← X[0] − 1
(41)         if l ≤ 0  or X[0] > 0 then
(42)             return r
(43)         endif
(44)     endwhile
(45) endif
(46) return r
```

Algorithm 13: Search algorithm
Input: level 0 array pointer: X_0, address length: w, stride length array pointer: sl, search key address: a
Output: longest prefix matching route pointer or Λ
search(X_0, w, sl, a)

```
(1)  RoutePointer lmr ← X_0[1]
(2)  Array X ← X_0 /* level 0 array */
(3)  Int ss ← 0 /* stride length summation */
(4)  Int s /* stride */
(5)  Int l ← 0 /* level */
(6)  Int i ← 0 /* index */
(7)  while true
(8)      ss ← ss + sl[l]
(9)      s ← (a ≫ (w − ss))&((1 ≪ sl[l]) − 1)
(10)     i = fringeIndex(sl[l], s)
(11)     if X[i]&1 = 1 then
(12)         /* update current longest matching route */
(13)         r ← (X[i]&(∼1))[1]
(14)         if r ≠ Λ then lmr = r
(15)         X ← X[i]&(∼1)
(16)         l ← l + 1
(17)     else if X[i]&(∼1) = 0 then
(18)         return lmr
(19)     else
(20)         return X[i]
(21)     endif
(22) endwhile
```

B Analysis of Algorithms

The complexity of all three routing table operations is independent on the number of routes in the ART. Instead, the complexity of all three depends on the stride length distribution, which is the maximum stride length (say s_{max}) and the number of strides (say N_s). The insertion and deletion time mostly depends on the amount of memory access in *allot*(). That is why both operations are $O(2^{s_{max}})$. The search time is proportional to the number of strides. That is why the search is $O(N_s)$. Table 5 shows the maximum number of routing table memory accesses for insertion, deletion, and search.

Table 5: COMPLEXITY OF ART

Insertion	Deletion	Search
$\sum_{i=0}^{s_{max}-1} 2^i (= 2^{s_{max}} - 1)$	$\sum_{i=0}^{s_{max}-1} 2^i + 1 (= 2^{s_{max}})$	$2N_s$

No arrays have the routes whose array-local prefix length is 0 in the multi-level ART as described in Section 2.2. That is why the summation ends at $s_{max} - 1$. Deletion requires one more routing table memory access to obtain the next most specific route pointer in addition to the same number of memory accesses as insertion. Search requires two routing memory accesses per level; one is the route pointer access, the other is the current longest matching route pointer access.