ELSEVIER

Computer Communications 22 (1999) 1415–1422

computer communications

Fast IP routing lookups for high performance routers

T. Kijkanjanarat, H.J. Chao*

Department of Electrical Engineering, Polytechnic University, 6 Metrotech Center, Brooklyn, NY 11201, USA

Received 23 December 1998; received in revised form 30 April 1999; accepted 30 April 1999

Abstract

The key to the success of the next generation IP networks to provide good services relies on the deployment of high performance routers to do fast IP routing lookups. In this paper, we propose a new algorithm for fast IP lookups using a so-called two-trie structure. The two-trie structure provides the advantages in that less memory space is required for representing a routing table than the standard trie while it still provides fast IP lookups. Based on the simulation result, the memory space can be saved around 27% over the standard trie while a lookup operation takes 1.6 memory accesses in the average case and 8 memory accesses in the worst case. Also, the structure is not based on any assumptions about the distribution of the prefix lengths in routing tables. Thus, increasing the lengths from 32 to 128 bit (from IPv4 to IPv6) does not affect the main structure.

Keywords: IP address lookup; Longest matching prefix; Trie

1. Introduction

The emergence of new multimedia networking applications and the growth of the Internet due to the exponentially increasing number of users, hosts, and domains have led to a situation where network capacity is becoming a scarce resource. In order to maintain the good services, three key factors have been considered in designing IP networks: large bandwidth links, high router data throughput, and high packet forwarding rates [12]. With the advent of fiber optics which provides fast links and the current switching technology which is applied for moving packets from the input interface of a router to the corresponding output interface at gigabit speeds, the first two factors can be readily solved. Therefore, the key to the success of the next generation IP networks relies on the deployment of high performance routers to forward the packets at the high speeds.

There are many tasks to be performed in packet forwarding such as packet header encapsulation and decapsulation, updating the TTL field in each packet header, classifying the packets into queues for specific service classes, etc. The major task, which seems to dominate the processing time of the incoming packet, is searching for the next hop information of the appropriate prefix matching the destination address from a routing table. This is also called an *IP route lookup*.

As the Internet has evolved and grown over in recent years, it has been proved that the IP address space which is divided into classes A, B, and C is inflexible and wasteful. Class C, with a maximum of 254 host addresses, is too small while class B, which allows up to 65,534 addresses, is too large for most organizations. The lack of a network class of a size which is appropriate for mid-sized organization results in exhaustion of the class B network address space. In order to use the address space efficiently, bundles of class C addresses were given out instead of class B addresses. This also causes massive growth of routing table entries.

To reduce the number of routing table entries, classless inter-domain routing (CIDR) [4] was introduced, to allow for arbitrary aggregation of networks. A network that has the identical routing information for all subnets except a single one requires only two entries in the routing table: one for the specific subnet (which has preference if it is matched) and the other for the remaining subnets. This decreases the size of the routing table and results in better usage of the available address space while, on the other hand, the efficient mechanism to do IP route lookups is required.

In the way of CIDR, a routing table consists of a set of IP routes. Each IP route is represented by a ⟨route prefix/prefix length⟩ pair. The prefix length indicates the number of significant bits in the route prefix. Searching is done in a longest matching manner. For example, a routing table may

* Corresponding author. Tel.: + 1-718-260-3302; fax: + 1-718-260-3906.

E-mail addresses: tws@kings.poly.edu (T. Kijkanjanarat), chao@antioch.poly.edu (H.J. Chao)

0140-3664/99/$ - see front matter
PII: S0140-3664(99)00099-7

Table 1
An example of expanding the route prefixes by using 2-bit trie structure

Routing table	
Original	Expanded
P1 = 0*	01* (P1)
P2 = 1*	00* (P1)
P3 = 10*	10* (P3)
P4 = 111*	11* (P2)
P5 = 1000*	1110* (P4)
P6 = 11001*	1111* (P4)
	1000* (P5)
	110010* (P6)
	110011* (P6)

have the prefix routes ⟨12.0.54.8/32⟩, ⟨12.0.54.0/24⟩, and ⟨12.0.0.0/16⟩. If a packet has the destination address ⟨12.0.54.2⟩, the second prefix route is matched and its next hop is retrieved and used for forwarding the packet.

Recently, there have been a number of techniques proposed to provide fast IP lookups [2,3,5,7,9–12]. One possible solution is using trie-based structure. A trie structure is a multi-way tree in which each node contains zero or more pointers to its children nodes. In the 1-bit trie structure [6], each node contains two pointers, the 0-pointer and the 1-pointer. A node *X* at the level *h* represents the set of all route prefixes that have the same *h* bits as their first bits. Depending on the value of the $(h + 1)$th bit, 0 or 1, each pointer of the node *X* points to the corresponding subtree (if it exists) which represents the set of all route prefixes that have the same $(h + 1)$ bits as their first bits. Each IP lookup starts at the root node of the trie. Based on the value of each bit of the destination address of the packet, the lookup algorithm determines the next node to be visited. The next hop of the longer matching prefix found along the path is maintained while the trie is traversed. Using the 1-bit trie structure for the IP route lookups provides the drawback in that the number of memory accesses in the worst case could be 32 for IPv4.

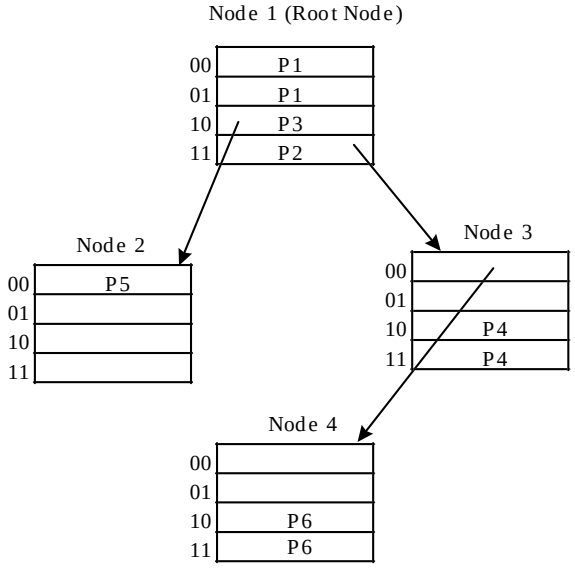

Fig. 1. The 2-bit trie structure corresponding to the routing table in Table 1.

The performance of lookups can be substantially improved by using a multi-bit trie structure. In the multi-bit, say *K*-bit, trie structure, each node contains 2^K pointers. Each route prefix in a routing table is expanded to be the new ones whose prefix lengths are the smallest numbers which are greater than or equal to the original length and also multiples of *K* bits. For example, a routing table, shown in the left column of Table 1, contains route prefixes (P1–P6). The asterisk in each route prefix indicates that the remaining bits are insignificant. By using 2-bit trie structure, the expanded version of the routing table can be obtained in the right column of Table 1. Note that the prefix P2 = 1* can be expanded into 10* and 11* but 10* is not used since it overlaps P3 which is the longer matching prefix. The corresponding 2-bit trie structure is also shown in Fig. 1.

The traversal to the node at each level of the *K*-bit trie is determined based on each *K*-bit part of the destination address of the packet. The advantage of the *K*-bit trie structure is that it provides fast lookups while the disadvantage is that large memory space is required for nodes of the structure since prefixes share the nodes representing their front parts while the ones representing their rear parts are rarely shared. In Fig. 1, for example, the root node (node 1) is used to represent the front part (the first two bits) of all prefixes (P1–P6) while the nodes 2, 3, and 4 are used to represent the remaining parts of the prefixes P5, P4, and P6, respectively. Each node is used by only one prefix.

In this paper, we propose a new way to do IP route lookups based on a so-called *two-trie structure*. In the two-trie structure, the nodes representing the front and rear parts of the prefixes are shared so that the resulting number of nodes in the structure can be reduced. Moreover, it still provides fast lookups. The original two-trie structure was proposed by Aoe et al. [1]. Their algorithms can be applied for various applications as long as searching is done in the exact match manner. In this paper, we present a new version of the two-trie structure and its related algorithms which are intentionally used for doing IP route lookups. Specifically, our lookup algorithm does the longest-prefix match while our updating algorithms have the property in maintaining the next hop information of the longest prefixes in the structure when the prefix addition or deletion is performed.

The paper is organized as follows. Section 2 provides the architecture of a generic router. Section 3 describes the model of the two-trie structure and some relevant terminology used throughout the paper. Section 4 explains the way to do IP route lookups. In Section 5, we present the updating algorithms for prefix addition and deletion. The performance study based on experimental results and a potential improvement are discussed in Section 6. The conclusion is provided in Section 7.

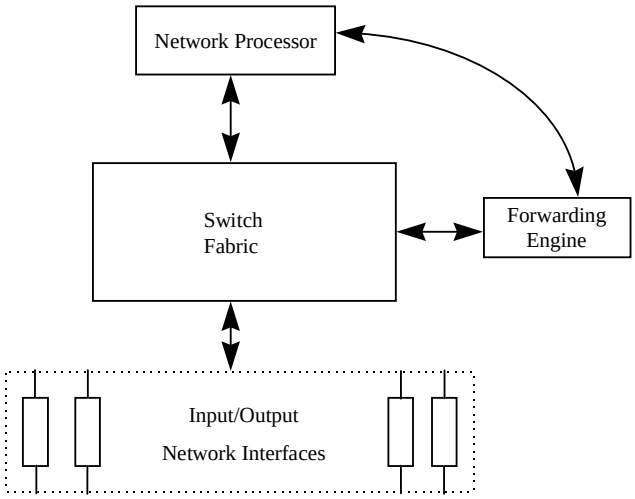

Fig. 2. Architecture of a generic router.

2. Architecture of a generic router

The architecture of a generic router is schematically shown in Fig. 2. A generic router contains the following components: input/output network interfaces, a forwarding engine, a network processor and a switch fabric. The input/output network interfaces are the points for incoming/outgoing packets. The network processor participates in routing protocols such as RIP or OSPF to create a routing table that is used in packet forwarding. It also performs some other tasks such as resource reservation, handling packets that need extra attention, and other administrative duties [2]. When a packet comes in, the forwarding engine uses a forwarding table, a local version of the routing table downloaded from the network processor, to make a routing decision. Then, the packet is transferred to the next hop through the switch fabric. It is not necessary to update the forwarding table for each routing update since the routing protocol needs time in the order of minutes to converge. It is the responsibility of the network processor to maintain the dynamic routing table for the next generation of the forwarding table while the forwarding table, on the other hand, can be optimized for performing fast lookups and needs only be updated once per few seconds.

3. A two-trie structure

A K-bit two-trie structure consists of two K-bit tries, the *front trie* and the *rear trie*, joining leaf nodes together in the middle. A node is called a front (rear) node if it is on the front (rear) trie. Both tries are allowed to be traversed in both directions. The direction outgoing from the root node of each trie to its own children nodes is called the *forward direction* while the opposite one is called the *backward direction*.

Throughout this paper, we assume that any route prefix (or IP address) X of length Y bits is represented in the dotted decimal form of $\langle x(0).x(1)..\ldots x(N)\rangle$, where $x(i)$ is a K-bit wide integer of the prefix X, $0 \le i \le (N-1)$, $x(N)$ is a '#' special symbol, and $N = \lceil Y/K \rceil$. For example, given $K = 8$, a prefix $X = 128.238.0.0$ of length $Y = 16$ bit would be represented as $\langle 128.238.\# \rangle$. A '#' special symbol

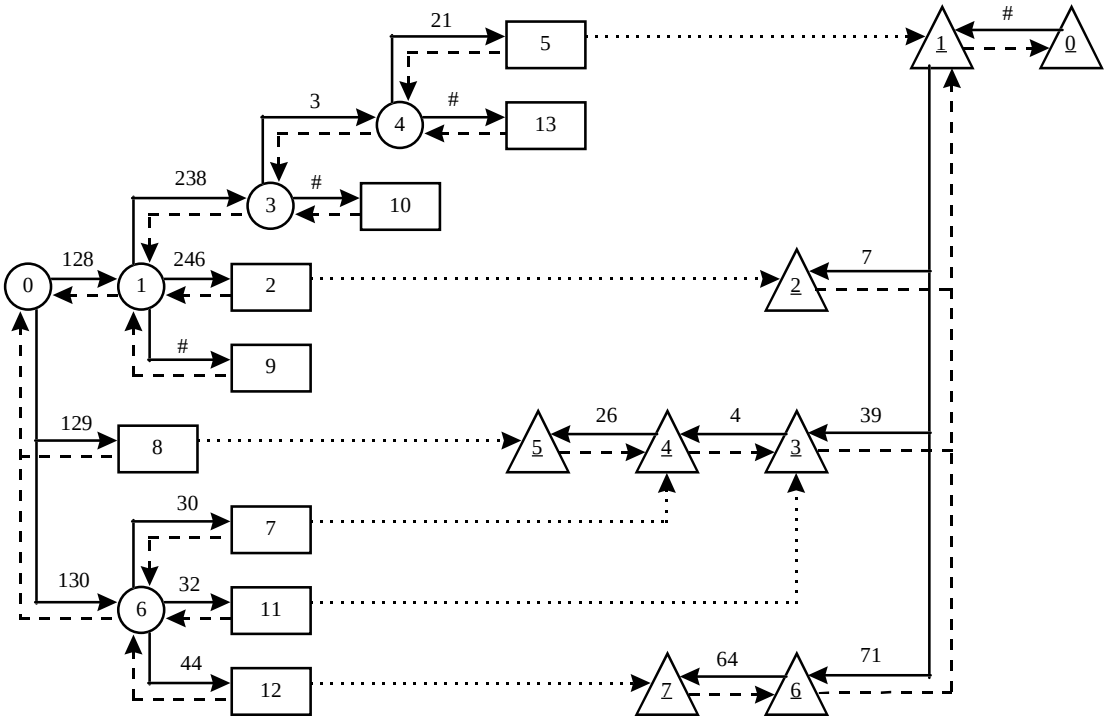

Fig. 3. An example of the two-trie structure when $K = 8$.

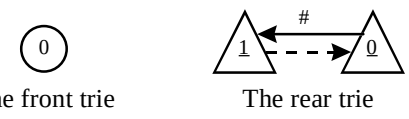

Fig. 4. Initialization of the front and rear tries before adding any prefix.

indicates the end of the prefix and is used to distinguish, for example, between ⟨128.238.#⟩ and ⟨128.238.3.#⟩ so that a prefix is not allowed in any other prefixes. If Y is not a multiple of K-bits, for example, given $K = 8$, $X =$ 128.238.32.0 and $Y = 20$, then X would be transformed as a set of prefixes ⟨128.238.j.#⟩, where $32 \leq j \leq 47$.

Fig. 3 shows an example of the two-trie structure when $K = 8$. The nodes of both tries are represented by the node numbers, where each root node is represented by the node number 0. Since we use the same node numbers on both tries, the node numbers of the rear trie are distinguished by an underline. The forward direction is illustrated by a solid line while the backward one is illustrated by a dashed line. Each leaf node on the front trie, the last node on the front trie in each path when traversing the front trie in the forward direction, is called a *separate node*. Note that the separate nodes are the nodes that differentiate a prefix from any other prefixes. The next hop of each prefix is stored at each separate node. In Fig. 3, separate nodes (nodes 2, 5, 7, 8, 9, 10, 11, 12, 13) are represented by using rectangles while other front nodes (nodes 0, 1, 3, 4, 6) are represented by using circles. Rear nodes (nodes $\underline{0}, \underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \underline{6}, \underline{7}$) are represented by using triangles.

4. IP route lookup algorithm

In this section, we will illustrate how to do IP lookups. The IPLookup(X) algorithm is shown. The input of the function is the IP destination address X of an incoming packet which is in the dotted decimal form. The function returns the next hop of the longest matching prefix found in the two-trie structure.

Algorithm (IPlookup(X)).

1. Let Z be the variable that stores the next hop of the longest matching prefix. Initially Z is the default next hop.
2. Start to do an IP lookup from the root node of the front trie by matching each K-bit part of the destination address X of the packet with prefixes in the two-trie structure.
3. If there is a match, the traversal is moved to the child node at the next level of the front trie.
4. Whenever a new front node is arrived, the algorithm first looks for its child node corresponding to the '#' symbol (which must be the separate node). If the node is found, it means that the two-trie structure contains the longer matching prefix so the variable Z is updated with the next hop value of this prefix retrieved from the separate node.
5. When the separate node is reached, matching continues to the rear trie by using a pointer at the separate node (shown as a dotted line in Fig. 3). Matching on the rear trie is done in the backward direction.
6. The algorithm stops whenever

   (a) a mismatch is detected somewhere in the structure. In such a case, the current value of Z is returned as the next hop, or
   (b) the traversal reaches the root node of the rear trie (no mismatch is detected). This means that the destination address X of the packet is actually stored as a prefix in the structure. The variable Z is updated with the next hop value of the prefix stored at the separate node we previously visited and returned as the output of the function.

Example 1. Suppose that we would like to look up the destination address $X = ⟨130.44.64.71.\#⟩$ from the two-trie structure shown in Fig. 3. The lookup starts from the root node of the front trie and proceeds along the front trie by moving to the next nodes 6 and 12. Then, the traversal is transferred to the rear trie and continues through the rear nodes $\underline{7}, \underline{6}$, and $\underline{1}$ until the root node of the rear trie is reached. The algorithm stops and the next hop stored at the separate node 12 is assigned to the variable Z and returned as the output. Note that the algorithm can be improved by keeping track of the number of parts of the address X we already matched. Instead of traversing until reaching the root node of the rear trie, in this example the algorithm can stop when '71' which is the last part of X is matched at the rear node $\underline{6}$.

Now let us change the destination address X to be ⟨130.44.64.72.#⟩ and do the lookup again. When the traversal is at the rear node $\underline{6}$, there is no match with '72'. The algorithm stops and returns the current value Z which is the default next hop as the output.

5. Prefix update algorithms

In this section, we will introduce the AddPrefix(X, Y, Z) and DelPrefix(X, Y) algorithms which perform the addition and deletion of a prefix X of length Y bit with the next hop Z. Note that when Y is not a multiple of K-bits, the prefix X is first transformed to a set of the new prefixes and then the addition and deletion are performed for each prefix which is just generated.

Before we start to build up a routing table by adding any prefix, initially the front trie has only the root node, the node

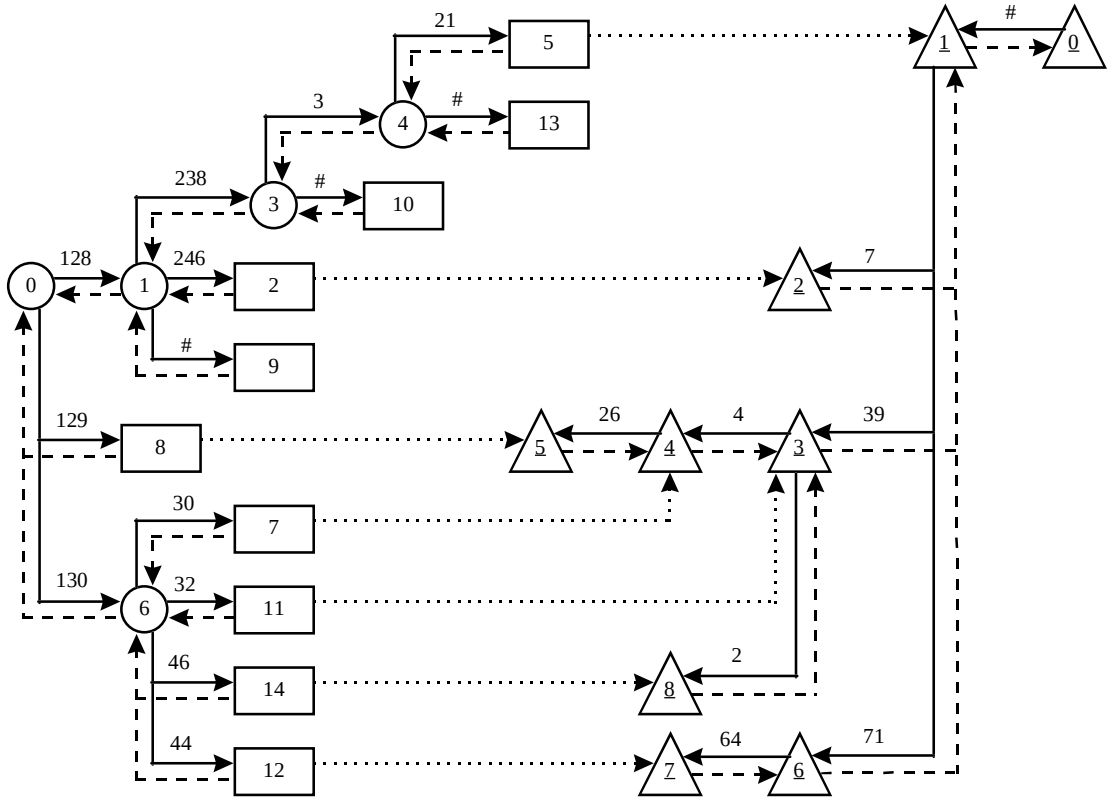

Fig. 5. The result of adding the prefix $X = \langle 130.46.2.39.\# \rangle$ into the two-trie structure in Fig. 3.

0, while the rear trie has two nodes, the nodes 0 and 1, and pointers corresponding to the '#' symbol, as shown in Fig. 4.

Algorithm (AddPrefix(X, Y, Z)).

1. Start by traversing nodes from the root node of the front trie to the root node of the rear trie to check whether the prefix X is in the structure.
2. If the prefix X is in the structure, the traversal will finally reach the root node of the rear trie. The algorithm retrieves the current length of X from the separate node we just visited along the path and compares with the length Y. If Y is greater than or equal to the current length of X, the algorithm will update the length and the next hop of X, which are stored at the separate node along the path we just visited, with the values Y and Z, respectively. Then the algorithm stops.
3. If the prefix X is a new one, a mismatch is detected somewhere in the structure. There are two possible cases:

   (a) The mismatch is detected on the front trie.

   (i) Create a new separate node for matching the K-bit part of the prefix X which starts to be different from any other prefixes in the two-trie structure. Store the length Y and the next hop Z at this node.
   (ii) Adding the remaining K-bit parts of the prefix X to the rear trie. Since the remaining parts of X can be shared with those of other prefixes in the structure, the algorithm traverses the rear trie from the root node in the forward direction and matches the remaining parts of X from the end. Some new rear nodes may be created for the parts of X that have no matches.
   (iii) Set a linkage to connect the separate node in (i) with the rear trie in (ii).

   (b) The mismatch is detected on the rear trie.

   (i) At the rear node the mismatch is detected, let the variable M be the K-bit part of the prefix in the structure which differs from the one of the prefix X. Note that M is the value that causes the occurrence of the mismatch.
   (ii) Retrieve the length and the next hop at the separate node we just visited and copy into the variables L and N. Change this separate node into a front node.
   (iii) Create new front nodes on the front trie for matching the parts of the prefix X the algorithm already visited on the rear trie from the first rear node we visited to the rear node where the mismatch is detected.
   (iv) Create a new separate node next to the front states in (iii) for matching the value M. Set a linkage of this node with the rear trie. Store the values of L and N into this separate node.
   (v) Add the necessary parts of the new prefix X and its length and next hop into the two-trie structure by doing the same steps as the ones from (i) to (iii) of (a).
   (vi) Remove all unused nodes from the rear trie.

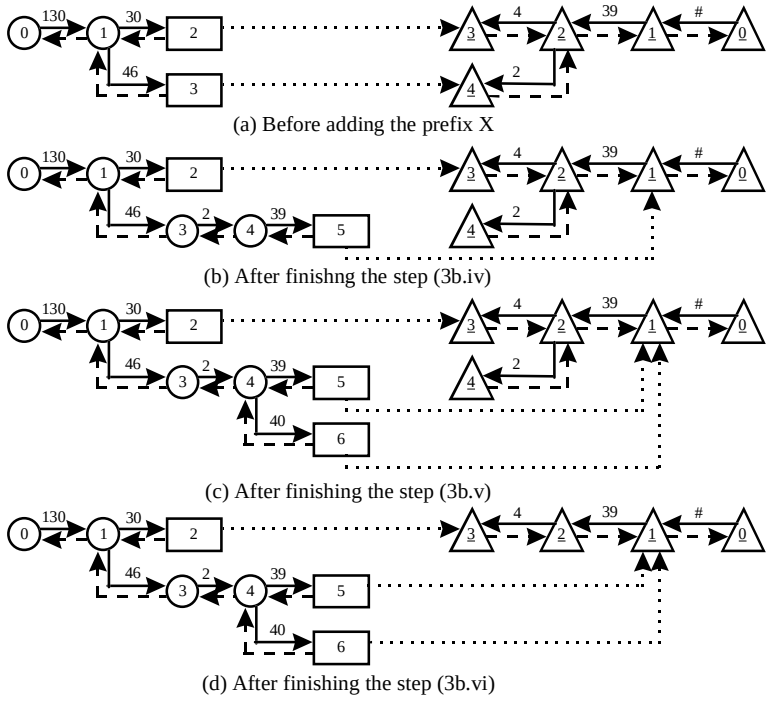

Fig. 6. Illustration of adding of the prefix X in Example 3.

Example 2.

Suppose that we would like to add a prefix $X = \langle 130.46.2.39.\# \rangle$ of length $Y = 32$ bit and the next hop Z into the structure in Fig. 3. The traversal starts from the root node of the front trie, matches '130', and moves to the node 6. At the node 6, a mismatch is detected on the front trie. The function creates a new separate node 14 for matching '46' and stores the length Y and the next hop Z at this node. The remaining part of X which is $\langle 2.39.\# \rangle$ is checked with the rear trie. From the root node of the rear trie, the traversal passes the nodes 1 and 3 for matching '#' and '39'. At the node 3, a new rear node 8 is created for matching '2'. Finally, a linkage is set to connect the separate node 14 with the rear node 8. The final result of the algorithm is illustrated in Fig. 5.

Example 3.

Suppose that we would like to add a prefix $X = \langle 130.46.2.40.\# \rangle$ of length $Y = 32$ bit and the next hop Z into the structure in Fig. 6(a). The traversal starts from the root node of the front trie and finally a mismatch is detected at the rear node 2. Copy the value '39' into M. The length and the next hop of the prefix $\langle 130.46.2.39.\# \rangle$ stored at the separate node 3 is copied into L and N. Change the status of the node 3 from the separate node into a front node. Now create the front node 4 and the separate node 5 for matching '2' and M (which is '39'). Store the values L and N into the separate node 5 and set a linkage to connect the node 5 with the rear node 1. The result is shown in Fig. 6(b). Add the necessary parts of X by creating a new separate node 6 for matching '40', traversing the rear trie from the root node for matching '#', stopping at the node 1, setting a linkage to connect the separate node 6 and the rear node 1. The result is shown in Fig. 6(c). Remove the unused node 4 from the rear trie. The final result of the function is shown in Fig. 6(d).

Algorithm (DelPrefix(X, Y)).

1. Start by traversing nodes from the root node of the front trie to the root node of the rear trie to check whether the prefix X is in the structure.
2. If X is not in the structure, the algorithm stops.
3. If X is in the structure, the traversal finally reaches the root node of the rear trie. The current length of X retrieved from the separate node we just visited along the path is compared with the length Y. If they are both equal, the algorithm tries to search for the second longest prefix from the structure and replaces the length and the next hop of X with the ones of this prefix. If we cannot find the second longest prefix, we do the prefix deletion by performing the following steps:

   (a) Remove the separate node we just visited during the traversal and destroy the linkage which connects with the rear trie.
   (b) Remove unused nodes from the rear trie.
   (c) Rearrange the two-trie structure to maintain the property that the separate nodes are the nodes that differentiate a prefix from any other prefixes.

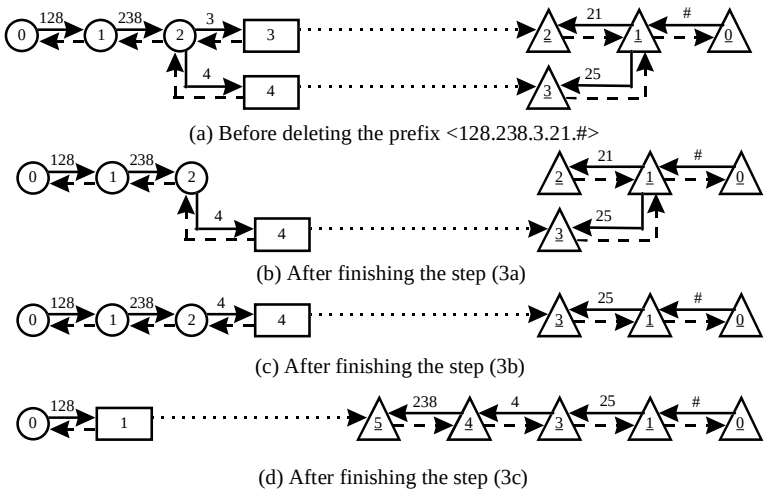

Fig. 7. Illustration of deleting the prefix ⟨128.238.3.21 #⟩ in Example 4.

Example 4.

Suppose that we would like to delete the prefix ⟨128.238.3.21 #⟩ from the structure shown in Fig. 7(a). We do the deletion operation by removing the separate node 3 and the linkage connecting the node 2, as shown in Fig. 7(b). The unused node 2 is removed from the rear trie, as in Fig. 7(c). Then the two-trie structure is rearranged in that the prefix ⟨128.238.4.25 #⟩ should be differentiated by using the node 1 instead of the node 4. The final result of the operation is shown in Fig. 7(d).

6. Performance study

In this section, we will discuss the performance of our scheme based on program simulation. For performance measurement, we used a practical routing table with over 38,000 prefixes obtained from [8]. Two performance issues are considered: (1) the number of memory accesses taken for the lookup operation and (2) the memory space required for the two-trie structure.

6.1. The number of memory accesses for lookup operation

The worst case of the lookup operation for the 8-bit two-trie structure occurs when, for example, the IP destination address ⟨128.238.3.21 #⟩ is searched for the next hop in Fig. 3. The lookup operation takes one memory access at the root node to transfer to the node 1. At the nodes 1, 3, and 4, it takes three memory accesses for each level. Two for updating the next hop value to be one of the longer prefix and one for transferring to the next level. At the separate node 5, the algorithm reaches the last part of the address which is '21'. It takes one more memory access to retrieve the next hop value. Thus, the total number of memory accesses in the worst case is 11.

Table 2
The number of memory accesses for the lookup operation

Structures	Bits for each level	Average case/worst case
Two-trie	8, 8, 8, and 8	3.6/11
Two-trie	16, 8, and 8	1.6/8
Trie	8, 8, 8, and 8	2.1/4

The performance of the lookup operation can be, however, improved by modifying the value of K at the first level of the two-trie structure so that the root node of the front trie covers 16 bit of the prefixes. In such a case, the number of levels at the front trie is reduced by one and the total number of memory accesses in the worst case is reduced to be 8.

To find the IP lookup performance on the average, assuming that the IP addresses are equally probable, we generated 100 million addresses and performed the lookup operations. For the sake of comparison, we used the 8-bit trie structure. The result is shown in Table 2.

6.2. Memory requirement for the structure

A typical implementation can be used to implement the two-trie structure. The front or the rear node contains one pointer to its parent node and the entries providing the associated pointers to the nodes at the next level. The separate node stores the length and the next hop information of each prefix. It also stores two pointers for each direction. Additionally, each node in the structure contains other necessary node information which would be used while performing the prefix addition and deletion.

To build up the two-trie structure, we read each IP route from the practical routing table in the random order and performed the prefix addition. The result is shown in Table 3.

By using the two-trie structure to implement the routing table, we can save the memory by around 27% when comparing with the standard trie.

Table 3
Memory requirements for the structures after creating the routing table with 38,367 entries

Structures	Bits for each level	Memory requirements (MB)
Two-trie	8, 8, 8, and 8	11.6
Two-trie	16, 8, and 8	11.6
Trie	8, 8, 8, and 8	16.0

7. Conclusion

We have presented a new way to represent IP routing tables by using a so-called two-trie structure. The two-trie structure provides the advantages in that less memory space is required for representing a routing table than the standard trie while it still provides fast IP lookups. We have also proposed the lookup algorithm which provides fast lookups and the updating algorithms which allow the insertion and deletion to be done dynamically. Our results indicate that the memory can be saved around 27% while it takes 1.6 memory accesses for lookup operation in the average case and 8 memory accesses in the worst case. Also, the structure is not based on any assumptions about the distribution of the prefix lengths in routing tables. Thus, increasing the lengths from 32 to 128 bit (from IPv4 to IPv6) does not affect the main structure at all. With small modifications, it can be applied to the larger addresses of IPv6.

Acknowledgements

The authors would like to thank Xiaolei Guo for his detailed and insightful suggestions. The authors would also like to thank the anonymous reviewers for their helpful comments.

References

[1] J. Aoe, K. Morimoto, M. Shishibori, K. Park, A trie compaction algorithm for a large set of keys, IEEE Transaction on Knowledge and Data Engineering 8 (3) (1996) 476–490.
[2] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, ACM SIGCOMM (1997) 3–14.
[3] W. Doeringer, G. Karjoth, M. Nassehi, Routing on longest-matching prefixes, IEEE/ACM Transaction on Networking 4 (1) (1996) 86–97.
[4] V. Fuller, T. Li, J. Yu, K. Varadhan, Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, RFC 1519, September 1993.
[5] P. Gupta, S. Lin, N. McKeown, Routing lookups in hardware at memory access speeds, IEEE INFOCOM (1998) 1240–1247.
[6] D. Knuth, Fundamental Algorithms. Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.
[7] B. Lampson, V. Srinivasan, G. Varghese, Lookups using multiway and multicolumn search, IEEE INFOCOM (1998) 1248–1256.
[8] Merit Network, Inc., Routing Snapshot on October 30, 1997 at the Mae-East NAP. URL: http://www.merit.edu/ipma.
[9] S. Nilsson, G. Karlsson, Fast address lookup for internet routers, IFIP Fourth International Conference on Broadband Communication, 1998, pp. 11–12.
[10] V. Srinivasan, G. Varghese, Faster IP lookups using controlled prefix expansion, ACM SIGMATICS (1998) 1–10.
[11] H.-Y. Tzeng, Longest prefix search using compressed trees, IEEE GLOBECOM (1998).
[12] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, Scalable high-speed IP routing lookups, ACM SIGCOMM (1997) 25–36.