A Hybrid B-Tree and Bloom Filter IPv6 Routing Lookup Algorithm Postprint

Authors: Yao Ming, Zhao Jingjing, Xingya He, Yang Yun

Date: 2018-05-24T00:00:00+00:00

Abstract

To improve IPv6 routing lookup efficiency and address the problem of non-uniform distribution of IPv6 routing prefixes, a hybrid IPv6 routing lookup algorithm based on B-tree and Bloom filter (BTBF) is proposed. BTBF consists of two parts: B-tree lookup and Bloom filter lookup. First, the B-tree is utilized to search for the first 16-bit value of the routing prefix. Then, through the mapping of the bit vector in the B-tree node, the next step is linked to the Bloom filter. Finally, the next hop is extracted using the value mapping of the Bloom filter bit array. Experimental results demonstrate that, compared with other tree-based and Bloom filter-based algorithms, the BTBF algorithm effectively reduces space and time overhead, and can maintain stable lookup performance even when the number of routing table entries varies significantly.

Full Text

Preamble

A Fast IPv6 Routing Lookup Algorithm Combining B-Tree and Bloom Filter

Yao Ming, Zhao Jingjing, He Xingya, Yang Yun
(College of Information Engineering, Yangzhou University, Yangzhou, Jiangsu 225005, China)

Abstract: To improve IPv6 routing lookup efficiency and address the problem of non-uniform IPv6 routing prefix distribution, this paper proposes a B-Tree and Bloom filter-based IPv6 routing lookup algorithm (BTBF). BTBF consists of two components: B-Tree lookup and Bloom filter lookup. The algorithm first uses a B-Tree to search for the first 16-bit value of the routing prefix, then maps to a Bloom filter through the bit vector in the B-Tree node, and finally extracts the next hop using the value mapping of the Bloom filter bit array. Experimental results demonstrate that compared with other tree-based

and Bloom filter-based algorithms, BTBF effectively reduces space and time overhead while maintaining stable lookup performance even with substantial changes in routing table entries.

Keywords: routing lookup algorithm; B-tree; Bloom filter; IPv6 backbone routing table

0 Introduction

IPv6 has garnered widespread attention and recognition from both industry and academia. China' s Next Generation Internet (CNGI) initiative and U.S. federal agencies have already deployed IPv6 in advance [1]. The IPv6 address length has increased from 32 bits in IPv4 to 128 bits, creating significant performance bottlenecks for backbone routers when performing table lookups. Consequently, research into reliable and fast lookup mechanisms specifically designed for IPv6 routing tables has become essential [2][3].

IP routing lookup algorithms can be categorized into prefix-length-based and prefix-value-based approaches. Prefix-length-based algorithms typically employ tree data structures; however, when extended to IPv6 lookup, the fourfold increase in address length results in greater tree depth and reduced lookup efficiency. Prefix-value-based algorithms can avoid the problem of excessive tree depth. Zhong [4] proposed a recursive balanced multi-path range tree (RBMRTs) data structure for longest prefix matching, achieving O(log N) complexity for both lookup and update operations. However, RBMRTs stores numerous duplicate endpoints in internal nodes, increasing time and memory requirements during route updates. The BSRPS algorithm [5] groups adjacent prefix value intervals into a single node to reduce redundant empty and duplicate nodes in the search tree, but it still generates many empty nodes when facing IPv6' s massive address space, substantially increasing memory access 次数.

Dharmapurikar et al. proposed a longest prefix matching algorithm based on Bloom filters [6], which stores IP address prefixes of different lengths in separate Bloom filters and queries them in parallel during lookup. This approach 容易产生过多的误判, and the vastly different scales of routing tables corresponding to different prefix lengths make it difficult to find a robust Bloom filter configuration. The TCAM-based efficient floating keyword matching algorithm [7] employs prefix expansion to reduce the number of Bloom filters, but this simultaneously expands the routing table and decreases lookup speed. The fast routing lookup method based on Bloom filters [8] uses an index table to probe the first byte to reduce Bloom filter queries, but it does not specify the first byte length, making the filtering efficiency unclear. Moreover, the index table adds lookup time on top of IPv6' s massive address space.

To address these issues, this paper proposes an IPv6 routing lookup algorithm based on the combination of B-trees and Bloom filters. Since B-trees produce

many duplicate nodes when querying large routing tables, our approach uses the B-tree only to lookup the first 16-bit prefix value to index into the Bloom filter, which then performs further routing table lookup. The shallow depth of B-trees effectively avoids empty sets, and under B-tree indexing, the scale of routing table entries that the Bloom filter needs to search, along with its false positive rate and lookup time, are all reduced.

---

1 IPv6 Address Structure and Backbone Routing Table Characteristics

1.1 IPv6 Address Structure

The latest IPv6 global unicast address structure specified in RFC3587 consists of three parts: an n-bit global routing prefix, a (64-n)-bit subnet identifier, and a 64-bit interface identifier [Figure 1: see original paper]. The global routing prefix identifies a site, the subnet identifier marks a subnet within the site, and the interface identifier (generated in IEEE EUI-64 format) represents a host within the subnet. The maximum prefix length used for routing is 64 bits. Statistical analysis of backbone routers reveals that the minimum IPv6 prefix length is 19 bits.

1.2 Backbone Routing Table Characteristics

Statistical analysis of over 80,000 routing table entries from the potaroo [9] autonomous system AS131072 reveals several key characteristics of current backbone router IPv6 addresses:

a) As shown in [Figure 2: see original paper], statistics on the first 16-bit values indicate that no routing table entries have prefix lengths shorter than 16 bits. There are only 39 distinct values in the first 16 bits, with non-uniform distribution and significant quantity differences. Prefix-value-based lookup methods cannot effectively match prefixes in smaller sets. When performing IPv6 routing lookup, it is necessary to both increase the probability of finding sets with more routing entries and reduce the depth of the prefix value search tree to locate the target set more quickly. The BTBF algorithm employs B-trees to achieve lower search tree depth.

b) According to potaroo official data, IPv6 routing tables show substantial distribution differences across prefix lengths from /16 to /64. Further statistical analysis of the AS131072 routing table yields [Figure 3: see original paper], where each line represents a set of first 16-bit values and the variation in routing prefix set counts across different prefix lengths. Although the sets formed by first 16-bit prefix values differ significantly in size, their distribution patterns are fundamentally similar. By using B-trees and prefix length values, the entire routing table can be partitioned into many smaller sets. Each peak in the figure corresponds to a

partitioned set that the Bloom filter needs to search. To simplify algorithm design, a threshold is established to classify these sets. The dashed line at position 1000 in [Figure 3: see original paper] divides the sets into thousand-level and hundred-level categories. This allows the Bloom filter to obtain optimal hash function counts and bit array sizes for each classified level, reducing error rates while improving lookup efficiency.

---

2 Routing Lookup Algorithm Based on B-Tree and Bloom Filter

IPv6 routing tables have few distinct values in the first 16 bits, concentrated primarily on 2001, 2600, and 2804, with fewer entries for other values. BTBF first uses a B-tree to query the first 16-bit value, classifying each corresponding routing table by prefix length. For example, potaroo backbone routers can be divided into 46 groups ranging from /19 to /64. Each group corresponds to a Bloom filter, with different numbers of hash functions and bit array sizes for different-scale groups to reduce invalid probes.

2.1 Data Structures

2.1.1 B-Tree Structure for First 16-bit Values B-tree operations (search, insertion, and deletion) complete in "logarithmic time." As a self-balancing tree, B-trees do not require frequent rebalancing like other balanced search trees. Given the small number of distinct first 16-bit values and their non-uniform distribution, the B-tree child node bounds are set to 2 and 3, creating what is commonly called a 2-3 tree. This tree has three key characteristics: each non-empty node contains between 1 and 2 data items; each node has at most 3 child nodes; and the tree is self-balancing. During updates, 4-nodes are converted to 3-nodes or 2-nodes in constant time.

2.1.2 B-Tree Node Data Structure B-tree nodes are classified as leaf or non-leaf nodes, each storing 1-2 data items that hold the first 16-bit prefix values. Node-to-Bloom-filter forwarding utilizes two m-bit vectors: VS[h(/19), h(/20), h(/21),···, h(/64)] and VL[h(/19), h(/20), h(/21),···, h(/64)]. These vectors store mapping information for set elements, where hash function h(/n) maps sets to vector V based on prefix length n. The B-tree node data structure is shown in [Figure 4: see original paper].

Each node contains two data items S and L (the smaller and larger prefix values, respectively, each 16 bits), a flag CNUMBER, three pointers (LNode, MNode, RNode), and two vectors VS and VL. CNUMBER indicates the node type (2-node, 3-node, or leaf) with 2 bits taking values 11, 10, or 00. When CNUMBER is 11, the node is a 3-node with up to 3 children, storing all data items, child pointers, and vectors. When CNUMBER is 10, the node is a 2-node with at most two children, retaining only data item S and vector VS. When CNUMBER

is 00, the node is a leaf with no children. The vectors map routing table entry sets where the first 16-bit prefix value equals S or L. According to current IANA IPv6 allocation policies, backbone routers contain no routing entries with prefixes shorter than /19, so vector elements are stored sequentially from h(/19) to h(/64). The two-stage lookup process uses VS and VL: when a B-tree node matches, the algorithm selects the vector corresponding to data item S or L, reads the routing set address stored in the vector element, and forwards to the Bloom filter for querying.

2.1.3 Bloom Filter Data Structure As shown in [Figure 5: see original paper], a Bloom filter initializes with an m-bit array where all elements are 0 and defines k different hash functions, each randomly mapping input elements to a bit position. Without false positives, a deterministic input yields a deterministic bit array value. Standard Bloom filters cannot perform deletion operations, so a counters array replaces the original bit array. When inserting an element, the k corresponding counter array elements are incremented by 1; when deleting, they are decremented by 1.

The Bloom filter probes the routing entry set forwarded from the B-tree matching node. The process reads non-zero elements in vector V[h(/19), h(/20), h(/21),⋯,h(/n)] to obtain addresses for m sets (1 < m < n-18). Each set corresponds to a Bloom filter, resulting in m parallel Bloom filters. The destination IP address is input in parallel to these m Bloom filters. Based on the matched Bloom filter' s counter array values, the next-hop address is extracted.

In the final step, the Bloom filter only detects whether a matching longest prefix exists for the target IP address; it does not store the routing table itself. The specific next-hop address requires further routing table search. Even though backbone routing tables are classified by prefix length, some length-based sets remain large. Traditional search and extraction impacts lookup performance, so a Bloom filter mapping method enables rapid target routing entry extraction. When inserting a new prefix X into the Bloom filter, the counters bit array yields value Cn. As shown in [Figure 5: see original paper], X's value Cx = 0100010000 01000 = 0x2208. An address mapping fx(Cn) is established through Cn' s value, representing the specific location of the target IP address prefix in the routing table. This enables fast lookup and forwarding of IP packets. [Figure 6: see original paper] illustrates the routing entry storage and address relationships.

2.2 Routing Lookup

The routing lookup pseudocode is as follows:

```
input: dstIP; // destination IP address
output: next hop;
search BTBF(dstIP){
  1. BMP = next hop of default route;
  2. key = the 1-16bit of dstIP;
```

```
  3. node = search B-Tree of order 3(key);
  4. if(node.CNUMBER == 2) node.S = node.L;
  5. if(key != node.S || key != node.L){
  6.   if(key < node.S ) node = node.LNode;
  7.   if(key > node.S && key < node.L) node = node.MNode;
  8.   if(key > node.L) node = node.RNode;
  9.   search BTBF(dstIP);
  10. }else{
  11.   if(node.CNUMBER == 0) return BMP;
  12.   if(key = node.S) return Vs[];
  13.   if(key = node.L) return VL[];
  14. }
  15. i = the length of route prefix;
  16. counters = BinaryArray of Bloom Filter;
  17. for (i = 64 down to 19) search dstIP%i;
  18. if (V[h(/i)] = 1) return counters[i];
  19. return BMP = fx(counters[i]) -> nexthop;
}
```

The algorithm first stores the router' s default next hop in BMP (Best Matching Prefix) (line 1), then extracts the first 16 bits of the destination IP address for B-tree lookup (lines 2-14). The specific search process determines whether the current node is a 2-node or 3-node. For 2-nodes, data item S is assigned to L (line 4). The 16-bit key value is compared with data items; if no match occurs, the algorithm forwards to LNode, MNode, or RNode based on conditional expressions (lines 6-9). If no matching node is found and the current node is a leaf, the routing table contains no matching prefix and the packet is forwarded to the default next hop (line 11). If a matching node exists, the corresponding vector V is returned (lines 12-14).

Finally, the Bloom filter searches for the prefix length corresponding to the destination IP address. If found, the counters array value is returned and used for address extraction; if not found, the packet is forwarded to the default route next hop (lines 15-19).

[Figure 7: see original paper] shows the algorithm flowchart, which comprises two main parts: the B-tree search process (left) and the Bloom filter search process (right). The B-tree search proceeds as follows: (a) extract the first 16 bits of the IP address to query, assign its value to variable key, and set node to the B-tree root; (b) traverse the B-tree, first checking if key equals node S. If equal, the match succeeds and the algorithm enters the Bloom filter query for the routing set mapped by vector VS. If not equal, it checks against node L. If equal, it enters the Bloom filter query for the set mapped by vector VL. (c) If key matches neither S nor L, the algorithm proceeds to the next node. Based on CNUMBER, it determines the number of child nodes. If CNUMBER is 00, the node is a leaf and the match defaults to the next hop. If CNUMBER is 01, the node has two children; key $>$ S enters left child LNode, key $<$ S enters right

child RNode. If CNUMBER is 11, the node has three children; key < S enters LNode, S < key < L enters MNode, key > L enters RNode. The found child node is assigned to node and the loop returns to step (b).

When the first 16 bits of the destination IP address successfully match in the B-tree, the algorithm obtains the routing entry set mapped by the corresponding vector V. Within this set, the routing table is further divided into 46 subsets by prefix length (/19-/64), each corresponding to a Bloom filter. The Bloom filter query phase then proceeds: (a) the destination IP address is input in parallel from /19 to /64 into the corresponding Bloom filters; (b) if no match is verified, the default next hop is used. If a match is verified, the next hop is obtained through the prefix address mapping in the Bloom filter, completing the lookup process.

2.3.1 Route Update Process

BTBF supports incremental updates, requiring only local data structure modifications when the routing table changes rather than complete reconstruction. When adding a new routing entry, B-tree insertion is performed followed by counter array increment operations in the Bloom filter.

a) B-tree insertion: Obtain the first 16-bit prefix value and prefix length n of the route entry, search for the value in the B-tree to find the matching node. If the node' s S or L value matches, forward to the Bloom filter mapped by vector V[h(/n)] for step (b) update. If no match exists, insert the value into the B-tree node. The B-tree (a 2-3 tree in this algorithm) automatically rotates to maintain balance. Finally, update vector element V[h(/n)] and forward to the Bloom filter mapped by h(/n) for step (b) update.

b) Bloom filter insertion: Increment by 1 the values of counters array elements detected in step (a)' s mapped Bloom filter.

Deletion operations are similar: perform the B-tree lookup from step (a), but decrement counters array element values by 1 in step (b). When a Bloom filter' s counters array value Cn reaches 0, indicating no routing prefixes remain in the set, the algorithm backtracks to the B-tree node to set vector element V[h(/n)] to 0 and releases the address mapping. If all elements in vector V are 0, the algorithm backtracks to the B-tree to delete the corresponding data item from the node and performs automatic balancing rotation.

2.3.2 Algorithm Performance Analysis B-tree height determines lookup efficiency. BTBF uses a 2-3 tree structure. In the worst case (all 2-nodes), lookup efficiency is lg N. In the best case (all 3-nodes), lookup efficiency is log N 0.631lgN. For a 2-3 tree with 1 million nodes, height ranges between 12-20; for 1 billion nodes, height ranges between 18-30. Insertion only requires modifying nodes associated with the target node without checking other nodes, so its efficiency matches lookup.

The BTBF B-tree only searches the first 16-bit prefix value, of which current backbone routing tables contain only a few dozen distinct values, forming an efficient tree structure. Thus, time complexity is approximately O(log N), whereas the RBMRTs algorithm, which uses only B-trees for routing lookup, achieves O(logN). BTBF nodes store at least two routing matches, requiring N/3-N/2 nodes in the best case, while RBMRTs requires at least 2N nodes.

Bloom filter insertion/query time is constant O(k), where k is the number of hash functions. Bloom filter usage must consider its false positive rate p, determined by:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-kn/m}\right)^{k}$$

where m is bit array size and n is the number of elements in the Bloom filter set. The formula shows that error rates decrease as m increases but increase with more inserted elements n. For a given error rate p, the optimal number of hash functions k is:

$$k = \frac{m}{n} \ln 2$$

The optimal bit array size m is determined by:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

Based on these formulas, the Bloom filter' s optimal bit array size and hash function count can be determined, significantly reducing error rates. Assuming a target error rate around 0.001 and using backbone router statistics, the most substantial 2001:: segment can be classified into thousand-level and hundred-level sets (e.g., /48 length contains 2,970 entries; 2620:: segment /32 length contains 1,362 entries). For thousand-level sets using n = 2,500, the optimal bit array size is only m 35Kb with k 10.06 hash functions (rounded to k = 10). Similarly, hundred-level sets using median n = 200 yield optimal bit array size of 2.8Kb with k = 10 hash functions. Compared with Bloom filter-based routing algorithms by Sarang D [6] and J. H. Mun [10], BTBF' s two-stage set partitioning drastically reduces both set element counts and hash function quantities. Conventional Bloom filters must search entire routing tables with over 13 hash functions, increasing false positive rates and lookup time while reducing efficiency.

3 Experimental Validation

The experimental phase collected routing prefixes from IPv6 backbone router autonomous systems AS6447, AS131072, and AS3356. Test code was implemented in C++ on the Visual Studio platform. First, routing table entries from the three autonomous domains were statistically analyzed. Based on these results and the formulas for optimal Bloom filter array size m and hash function count k, sets classified by the B-tree were processed. The classification yielded two levels (thousand and hundred) with parameters (m = 35Kb, k = 10) and (m = 2.8Kb, k = 10), respectively, corresponding to a 0.001 error rate.

To reflect real-world conditions, active routing entries from each autonomous domain were used. shows prefix statistics across the three domains.

** Prefix Statistics Across Three Autonomous Domains**

Routing Table Entries	AS6447	AS3356	AS131072
First 16 bits: 2001			
First 16 bits: 2600			
Total Routing Entries			

The implementation uses simple, high-performance folded XOR hash functions for Bloom filters. One hundred thousand packets were generated for testing: 85% selected from routing table entries to ensure most packets have matching entries, and 15% randomly generated to verify matching to either routing entries or the default next hop. To verify Bloom filter error rate stability, error counts were statistically analyzed for the 2001:: and 2600:: segments across the three autonomous domains. Results are shown in [Figure 8: see original paper], demonstrating that experimental results align with algorithm design, maintaining error rates around 0.001. In practice, each set can be customized with an optimal Bloom filter for even better error rate control.

Memory consumption primarily involves the B-tree, Bloom filter bit arrays, and routing table loading. Using the AS131072 BGP routing table, BTBF implementation memory usage remains at 2,700 KB, with the vast majority used for routing table loading. compares memory consumption results.

** Memory Usage Comparison Using Real Routing Tables**

Algorithm	Memory Usage
TSB	3.16M
Hash and CAM-based IPv6 routing	15.33M
BSRPS	
Fast routing lookup based on Bloom filter	
BTBF	2.7M

Theoretical BTBF time complexity is at most O(5). Lookup performance experiments used the same 100,000 packets, comparing BTBF with RBMRTs, BSRPS, and Bloom filter routing lookup algorithms. RBMRTs stores many duplicate endpoints in internal nodes, causing performance instability as routing tables grow. BSRPS uses a set tree based on prefix value intervals, offering stable performance but limited by IPv6' s massive address space, preventing the set tree depth from being controlled at a low level. The Bloom filter-based algorithm uses first-byte indexing to exclude some non-matching IP addresses, reducing Bloom filter probes. However, when the index quantity for a queried address' s first byte is large, the advantage over parallel queries diminishes, and reduced filter count increases error rates and lookup time. [Figure 9: see original paper] compares these algorithms' performance under identical conditions.

The results show that as table entries increase, BTBF' s lookup speed remains stable and high. This stability stems from BTBF' s two-stage classification (B-tree and prefix length), both of which adapt well to large data volume changes, keeping Bloom filter lookup sets at a stable size.

---

4 Conclusion

This paper analyzed IPv6 address structure, allocation policies, and IPv6 backbone routing table characteristics, proposing the BTBF algorithm that combines B-trees with Bloom filters. Compared with existing algorithms, BTBF offers fast lookup speed, small memory footprint, good scalability, and supports incremental updates. Experimental results demonstrate that BTBF maintains excellent lookup performance while adapting to routing table size changes, remaining consistently stable. This benefits from fully leveraging the B-tree' s low lookup depth, ensuring the algorithm' s correctness and reliability—validated by experimental results.

References

[1] Li Zhenqiang, Zheng Dongqu, Ma Yan. TSB: a multi-stage algorithm for IPv6 routing table lookup [J]. Acta Electronica Sinica, 2007, 35 (10): 1859-1864.

[2] Hsiao Yimao, Chu Yuansun, Lee Jengfarn, et al. A high-throughput and high-capacity IPv6 routing lookup system [J]. Computer Networks the International Journal of Computer & Telecommunications Networking, 2013, 57 (3): 782-794.

[3] Bando M, Chao H J. Flashtrie: hash-based prefix-compressed trie for IP route lookup beyond 100Gbps [C]// Proc of IEEE INFOCOM. 2010: 1-9.

[4] Zhong Pingfeng. An IPv6 address lookup algorithm based on recursive balanced multi-way range trees with efficient search and update [C]// Proc of International Conference on Computer Science and Service System. 2011.

[5] Cui Yu, Tian Zhihong, Zhang Hongli, et al. Binary search on range of IPv6 prefix sets [J]. Journal on Communications, 2013, 34 (06): 29-37+48.

[6] Dharmapurikar S. Longest prefix matching using bloom filters [J]. IEEE//ACM Transactions on Networking, 2006, 14 (2): 397-409.

[7] Li Kunpeng, Lan Julong. Efficient unfixed keywords matching algorithm based on TCAM [J]. Computer Engineering, 2012, 38 (4): 269 274.

[8] Yu Ming, Wang Zhen' an, Wang Dongju. A fast method for IP lookups based on Bloom filters [J]. Journal of Harbin Engineering University, 2014, 35 (10): 1247-1252.

[9] http://bgp.potaroo.net [EB/OL]. 2017.

[10] Ju H M, Lim H. New approach for efficient IP address lookup using a Bloom filter in trie-based algorithms [J]. IEEE Trans on Computers 2016, 65 (5).

Note: Figure translations are in progress. See original paper for figures.