Data Structure Optimization of AS_PATH in BGP

Weirong Jiang

Research Institute of Information Technology, Tsinghua University, Beijing, 100084, P.R.China
jwr2000@mails.tsinghua.edu.cn

Abstract. With the fast growing size and complexity of core network, the hash based data structure of current AS_PATH implementation in BGP is facing challenges in performance, mainly caused by the static attribute of the simple hash. This paper proposed a splay tree based data structure and an optimal index generation algorithm specifically designed for AS_PATH. Exploiting the innate characteristics of AS_PATH, the proposed algorithm shows superior performance.

1 Introduction

The Border Gateway Protocol (BGP) is an inter-Autonomous System (AS) routing protocol. One of the most important attributes in BGP is AS_PATH [1]. AS_PATH serves as a powerful and versatile mechanism for policy-based routing [2]. With the rapid development of Internet and wide deployment of BGP [10], storage and comparison of AS_PATH entries become a potential performance issue to be addressed.

This paper is a forward-looking exploration on optimizing the data structure of AS_PATH. The rest of the paper is organized as follows: In Section 2, we will present the inherent problems of hash data structure of AS_PATH and propose the possible solutions briefly. In Section 3, we discuss the optimization of the AS_PATH data structure by comparative study. In Section 4, we provide results of our simulation experiments. In Section 5, we put forward our conclusion and our expectations for future work.

2 Background and Challenges

The global Internet has experienced tremendous growth over the last decade. Figure 1 shows the BGP statistics [5] from Route-Views Data with trend curves added. As shown in Figure 1 (c), the number of unique AS_PATHs is growing at nearly an exponential speed, which motivates research in optimized algorithms to provide higher performance.

In typical BGP implementations [8, 9], hash table is preferred since in early days, when number of AS_PATH is small, it is the most simple and efficient way. To deal with the collision, different AS_PATH entries with same hash value will be stored in a linked list and be distinguished through linear search by comparing the whole AS_PATH. In theory [6, 7], the time complexity to insert, lookup or delete an entry in hash table is O(1), which obviously is perfect in AS_PATH attribute update and retrieval. To reach high efficiency, nearly half the hash table should be empty, and accordingly the hash table size should double the size of the existing unique AS_PATH entries, and thus the space complexity is O(2*n*) where *n* is the number of AS_PATH entries. For instance, in [8], the table size is 32,767, almost twice as the number of unique AS_PATH entries in the global routing table.

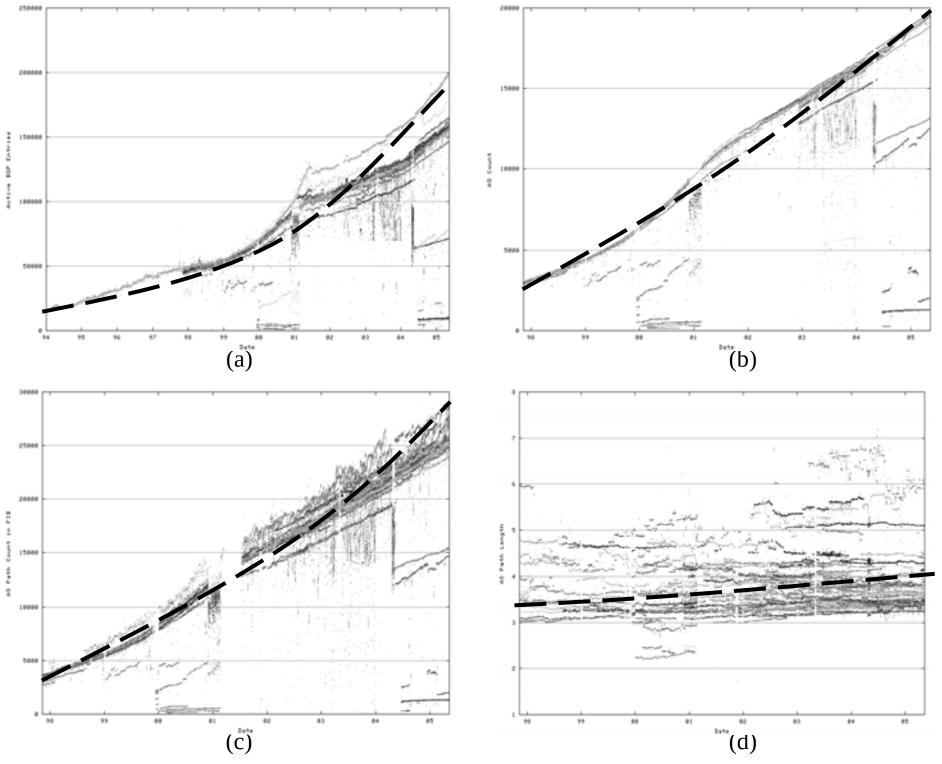

Fig. 1. BGP Statistics: (a) Active BGP entries (FIB); (b) Unique ASes; (c) Unique AS Paths used in FIB; (d) Average AS Path Length

These hash based implementations perform well nowadays in most of cases, but are expected to face severe challenges as follows.

Hash is a static data structure and the main disadvantage is the constant hash table size. The efficiency of hash will decline quickly since the hash table size will not catch up with the increasing number of AS_PATH entries. In addition, it is difficult to get it right for all situations. For example, the software including BGP routing needs to work on both a low end router with small memory and small number of routes and a high end router with large amount of memory and large number of routes. But there is no way to set a universal value for hash table size for both high and low ends. Obviously, to resolve this challenge, dynamic data structures such as binary trees could be good substitutes for hash.

The AS_PATH attribute is of various lengths and hardly can be used directly as an index. An index of constant length for each AS_PATH entry can be generated by encoding the AS_PATH attribute. Nevertheless, possible collision needs to be taken into consideration when two different AS_PATH entries are encoded into the same index. To reduce the probability of collision, folding is an easy and popular method to generate index. That is, split an AS_PATH into several equally sized sections and add all sections together. However, both splitting and adding up consume time. Since the AS_PATH is getting longer due to the rapid growth of AS number, the cost of folding is getting much more expensive. Thus there is need to find an algorithm more efficient to generate indexes.

Linking different entries with identical index is a simple solution for collision. However, increasing entries incline to cause more collisions and longer links. Then the efficiency of linked list operations (i.e. insert, lookup and delete) will also decline since entry comparison is usually expensive. One way to relieve this challenge is to construct different entries with the same index to be a secondary tree, rather than a linked list.

3 Optimizations by Exploiting the Characteristics of AS_PATH

3.1 Characteristic Observations

Table 1 shows a sample output of a BGP routing table from our test set which from real life includes more than 100,000 AS_PATH entries with 17,520 unique entries. Using this example, the following characteristics can be observed.

Characteristic 1. Many routes share one AS_PATH entry but few entries share one origin AS. In Table 1, there are at most three different AS_PATH entries originating from the same origin AS 1889. In our test set, over 80% AS_PATH entries monopolize one origin AS. Hence in most of cases a path could be specified by its distal AS indicating the origin AS of the entry.

Characteristic 2. Scanning the AS_PATH field in Table 1 from left hand side, we find that, nearer to the origin AS, two AS_PATH entries are more likely to be different. On the other hand, the nearest AS numbers, which indicate the ASes closer to local AS, are mostly the same. This can be explained that local AS usually has very few neighbors and hence left parts of most AS_PATH entries are similar.

Characteristic 3. Considering the update process, the efficiency of the data structure/algorithm is very important when it faces a burst of route updating, which might happen when one AS's state alters and all the AS_PATHs originating from it have to be updated. It requires the entry operated most frequently to be visited most promptly. This characteristic is coincident with a type of dynamic binary trees: splay tree [7].

Table 1. A sample output of a BGP routing table.

Network	Next Hop	Metric	LocPrf	Weight	AS_PATH
12.42.72.190/32	10.1.1.235	0	100	100	14207 3944 7777 i
12.43.128.0/20	10.1.1.235	0	100	100	14207 3944 2914 7018 16711 16711 16711 i
12.43.144.0/20	10.1.1.235	0	100	100	14207 3944 2914 7018 16711 i
12.65.240.0/20	10.1.1.235	0	100	100	14207 3944 2914 7018 17231 i
12.66.0.0/19	10.1.1.235	0	100	100	14207 3944 2914 7018 17231 i
12.66.32.0/20	10.1.1.235	0	100	100	14207 3944 2914 7018 17231 i
12.79.224.0/19	10.1.1.235	0	100	100	14207 3944 2914 7018 5074 i
13.13.0.0/17	10.1.1.235	0	100	100	14207 3944 2914 7018 22390 i
13.13.128.0/17	10.1.1.235	0	100	100	14207 3944 2914 4323 22390 i
13.16.0.0/16	10.1.1.235	0	100	100	14207 3944 2914 5511 5388 i
15.0.0.0/8	10.1.1.235	0	100	100	14207 3944 2914 209 71 i
15.130.192.0/20	10.1.1.235	0	100	100	14207 3944 2914 5400 1889 i
15.142.48.0/20	10.1.1.235	0	100	100	14207 3944 2914 3561 5551 1889 i
15.166.0.0/16	10.1.1.235	0	100	100	14207 3944 2914 209 71 i
15.195.176.0/20	10.1.1.235	0	100	100	14207 3944 2914 3561 1273 1889 i

3.2 Constructing Splay Trees

Similar to the means done in hash, as for a splay tree, each AS_PATH entry is stored as a whole, with an index whose value is calculated by the functions discussed in next section. The entries with identical index value are linked off the same node. Figure 2 shows the data structure of the tree node and an example process to construct a splay tree from left side to right.

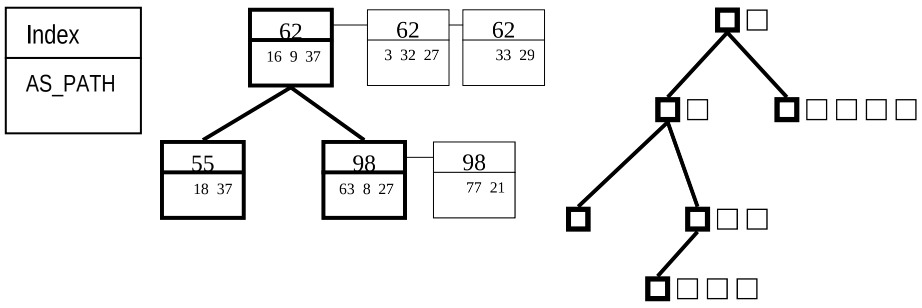

Fig. 2. Construct a Splay Tree for AS_PATH

3.3 Optimizing Index Generation

Define the k-step Golden AS in an AS_PATH Entry. We assume the golden section ratio is $\beta \approx 0.618$ and the length of an AS_PATH entry is m. The function *Position*(AS) indicates the position of an AS in the entry and its value range is {1,2,…, m }. Herein, *Position*(origin AS) = m. Then we use P_k to denote *Position*(k-step golden AS).

Definition 1. The 1-step golden AS is the one on the golden section point of the entry, that is, $P_1 = \lceil \beta m \rceil$.

Definition 2. The k-step golden AS is the one on the golden section point of the short section after last golden section, that is,

$$P_k = P_{k-1} + \lceil \beta(m - P_{k-1}) \rceil, P_k = 1, 2, \cdots, m. \qquad \textbf{(1)}$$

We impose the condition $P_k \neq P_{k-1}$, and consequently k has an upper boundary for each certain m. For our test set, $m \geq 3$, $k = 1, 2$.

Compare Different Index Generation Functions. As we have discussed, folding is expensive. According to characteristic 1, we employ the origin AS number as the index of an entry. Moreover, according to characteristic 2, we design other index generation functions whose time-consuming is on the same level. All the functions are presented as follows.

1. *Folding.* Split an AS_PATH entry into 16-bit sections and add all sections together to a 32-bit integer.
2. *Origin AS.* Directly get the rightmost AS number.
3. *Sum of rightmost two ASes.* Add the rightmost two AS numbers together.
4. *Sum of rightmost three ASes.* Add the rightmost three AS numbers together.
5. *Golden section.* Get the 1-step golden AS and add it to the origin AS.
6. *Golden section2.* Get the 2-step golden AS and add it to the origin AS.
7. *Golden section3.* Add the 1-step golden AS, the 2-step golden AS and the origin AS together.

We construct splay trees using our test set and regard the number of tree nodes and links, average length of all the tree nodes, average and maximum length of links and the time cost as the main judge of efficiency of index generation functions. Larger amount of tree nodes, less links, shorter length, and cheaper time cost, indicate the higher efficiency. The results are presented in Table 2.

According to the results, regardless of the time cost, folding seems most efficient, since it utilizes more information in an entry than any other function. However, the time cost of index generation influences much the efficiency of operations to insert, lookup and delete entries, especially when AS_PATH is getting longer. The other six types of index generation functions perform almost equal in efficiency. Hence using the origin AS as index is preferred for its simplicity.

Table 2. Efficiency of different Index Generation Functions.

Index Genera-tion	Number Of Tree Nodes	Total Average Length	Number of Links	Average Link Length	Max Link Length	Time Cost
Folding	16287	1.075705	1152	1.070313	3	O(*N*)[*]
Origin AS	13436	1.303960	2639	1.547556	12	O(1)
Sum 2	13492	1.298547	3133	1.285669	6	O(1)×2
Sum 3	14358	1.220226	2652	1.192308	5	O(1)×3
Golden 1	13077	1.339757	3366	1.319964	7	O(1)×2
Golden 2	13619	1.286438	3141	1.241961	6	O(1)×2
Golden 3	13921	1.258530	3014	1.194094	5	O(1)×3

3.4 Further Improvement

As we have discussed, when links get longer, the efficiency will decline badly for its linear data structure [6, 7]. This problem may come true soon owing to the astonishing increase of ASes and AS_PATH entries. If the link is replaced by a splay tree, our splay tree with links then alters to be a splay tree with a secondary tree, which might be called double-splay tree. We use the origin AS as index of the primary splay tree while we could use the 2-step golden AS or the second rightmost AS as index of secondary splay tree. Two different entries owning the same two indexes still have to be linked but the length of the link will be much shorter and hence the efficiency will be improved. Figure 3 shows an example process to construct a double-splay tree.

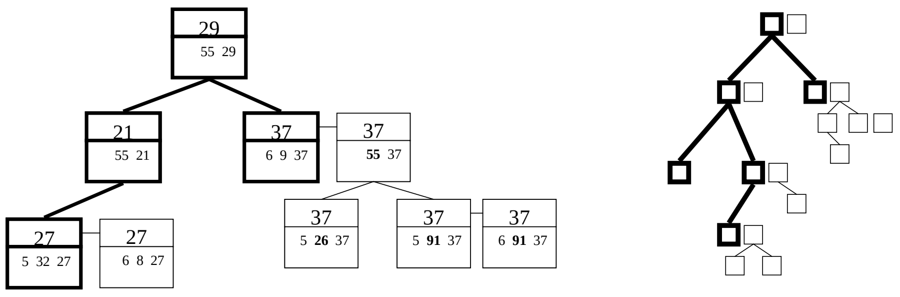

Fig. 3. Construct a Double-Splay Tree for AS_PATH

Limited by the size of test set, this improvement is not remarkable in our experiments since over 80% links are as short as just one node. We temporarily do not present the meaningless results in this paper. Nonetheless, we believe this improvement will be verified when AS_PATH entries in real life is getting much increased.

[*] *N* indicates the number of sections after splitting.

4 Simulation Experiments

4.1 Experiment Environment

For all the experiments we use a computer with a Pentium M processor running at 1.4GHz and 256 Mbytes of main memory. The host operating system is Windows XP professional with SP2. We develop and compile our program with the Microsoft Visual C++6.0 with default settings. In our program, each AS number is treated as a four-byte integer [4].

4.2 Splay Tree vs. Hash

To simulate the fact that hash is static while the number of AS_PATH entries is increasing explosively, yet limited by the condition that the number of existing AS_PATH entries is certain, we have to set the hash table size a small value (e.g. 37). We augment the size of test set from 100 to 100,000 entries, and observe the time cost to insert, lookup and delete entries. Results are shown in Figure 4(a ~ c).

Furthermore, to verify that static hash table size is not universal for both high and low end routers, we set the hash table size an appropriate value (e.g. 32,767) and experiment with small size of entries (e.g. 1,000 route entries). Figure 4 (d) reveals the memory waste for low end routers.

These results firmly demonstrate that, hash is not suitable as the data structure of AS_PATH because of its static feature. AS_PATH should be encoded into dynamic structures such as splay trees.

5 Conclusions and Future Work

According to our above discussions and experiments, hash is no longer fit for the data structure of AS_PATH for its fatal defects under the background of the explosive development of Internet. Instead, splay trees are more suitable for their dynamic attribute. To reduce collisions, we studied several functions to generate index after exploiting inherent characteristics of AS_PATH. And we suggest using the origin AS as the index. Furthermore, a novel binary tree named double-splay tree, is proposed and waiting for future's verifications.

Based on what we have done, we try to build a test bed in future to experiment with more dynamic data structures to seek more efficient data structure for AS_PATH.

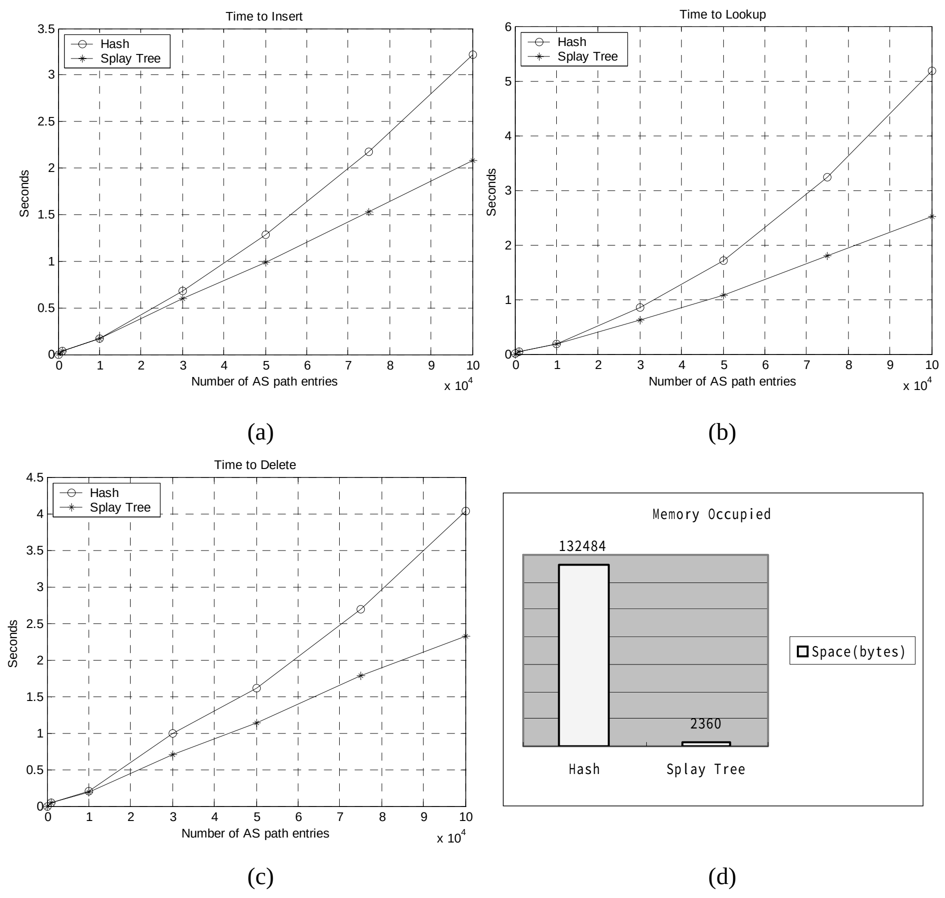

Fig. 4. Hash vs. Splay Tree using Origin AS as index

6 Acknowledgements

The author would like to thank Dr. Enke Chen at Cisco Networks for pointing out this research topic and providing much information and advice. He also would like to thank his supervisor Dr. Jun Li for his earnest enlightenment and comprehensive help. At last but not least, he would thank the Zebra community for the source code.

References

1. Rekhter, Y., and Li, T.: A Border Gateway Protocol 4 (BGP-4). IETF RFC 1771. (1995)
2. Traina, P.: BGP-4 Protocol Analysis. IETF RFC 1774. (1995)
3. Chen, E., and Yuan, J.: AS-wide Unique BGP Identifier for BGP-4. IETF draft-ietf-idr-bgp-identifier-04. (2004)
4. Vohra, Q., and Chen, E.: BGP support for four-octet AS number space. IETF draft-ietf-idr-as4bytes-08. (2004)
5. BGP Statistics from Route-Views Data. http://bgp.potaroo.net/rv-index.html. (2005)
6. Sahni, S.: Data structures, algorithms, and applications in C++. China Machine Press. (1999)
7. Shaffer, C.A.: Practical Introduction to Data Structure and Algorithm Analysis (C++ Edition). China Publishing House of Electronics Industry. (2002)
8. Zebra-0.94. http://www.zebra.org.
9. MRT-2.2.0. http://www.mrtd.net.
10. Meng, X., Xu, Z., Zhang, B., Huston, G., Lu, S., Zhang, L.: IPv4 Address Allocation and the BGP Routing Table Evolution. ACM SIGCOMM Computer Communications Review. 35(1): 71-80. (2005)