Adjusted Weight-Balanced Binary Tree for IP Routing Table Lookup*

Tsern-Huei Lee† and Pau-Chuan Ting‡

†Institute of Communication Engineering
‡Institute of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan 300
Republic of China
†thlee@atm.cm.nctu.edu.tw, ‡bcding@csie.nctu.edu.tw

Abstract

Various algorithms have recently been proposed for Internet routers to improve the performance of routing table lookup. However, most of previous research effort does not utilize the access frequencies of route entries. Since Internet traffic has significant temporal localities [9]-[11], one could largely improve the performance of routing table lookup by taking advantage of access frequencies. An example that aims at minimizing the average lookup time can be found in [12]. Since global optimization is computationally prohibited, the authors suggested to construct a binary tree based on a weight-balancing heuristic. We present in this paper that the resulted weight-balanced binary tree can be adjusted with a simple algorithm to further improve the performance. The adjustment algorithm is so simple that it basically does not increase the complexity in constructing the binary tree. Through analysis, we found that the improvement can be as large as 25% for a tree with four leaf nodes. For a tree of arbitrary size, the improvement could be larger than 12.5%.

Keywords — Routing table lookup, Weight-balanced tree, temporal locality.

1. Introduction

Because of the explosive growth of Internet traffic, the performance of IP routing table lookup is becoming critical for high-speed routers to provide satisfactory services. As such, many researches developed in the past few years new algorithms to accomplish high-speed routing table lookup [1]-[10]. Some of the algorithms compress the routing table with sophisticated data structures so that a processor can perform routing table lookup in its cache [1]-[7] and some others use simple data structures with special hardware to assist in routing table lookup [8]-[10]. In general, sophisticated data structure renders difficulty in table update and simple data structure requires a large amount of memory.

Unfortunately, most of previous research effort does not consider access frequencies of

*This work was supported in part by the Industrial Technology Research Institute in Taiwan, under contract T2-91034-4.

route entries. Recent studies of Internet traffic [9]-[11] demonstrate the existence of significant temporal locality in the packet streams. In other words, the average performance of IP routing table lookup can be significantly improved if an algorithm is developed with a data structure that considers access frequencies. The authors of [12] argued that Huffman encoding [13] is not applicable to routing table lookup and proposed to use a weigh-balancing procedure to generate a binary tree to represent the routing table. It was shown that the average table lookup time could be largely reduced with the weight-balanced binary tree.

In this paper, we present an adjustment scheme that can be applied to the weight-balanced binary tree to further improve the average performance. The adjustment is simple and thus basically it does not increase the complexity in constructing the binary tree. We show through analysis that the improvement could be larger than 12.5%. For a tree with only four leaf nodes, the maximal improvement is 25%.

The rest of this paper is organized as follows. In Section 2, we briefly review the concept of weight-balanced binary tree proposed in [12]. In Section 3, we present our adjustment algorithm. Section 4 contains analysis of the maximal achievable gain with the adjustment algorithm. Finally, we draw conclusion in Section 5.

2. The Weight-Balanced Binary Tree

The IP routing lookup problem can be described as follows. Given an incoming packet's destination IP address, find the longest matching prefix among a set of route prefixes. Since every prefix determines an interval, one can simply use binary search to solve the IP routing table lookup problem. In fact, binary search is deemed an efficient solution for routing table lookup. As an example, consider the prefixes shown in Table 1. In this example, we assume 4-bit addresses and the interval determined by prefix k is denoted by I_k. Notice that two intervals may contain common addresses. However, if two intervals I_a and I_b contain common addresses, then one is fully contained in the other, i.e., either $I_a \subset I_b$ or $I_b \subset I_a$. Since each prefix has two end points, one can use all the end points to divide the entire address space into elementary intervals (see Fig. 1(a)). Clearly, under the longest matching prefix criterion, an elementary interval uniquely determines a prefix. For example, prefix 000∗ is matched if a destination address falls in elementary interval [0000, 0001]. Having the elementary intervals, one can easily construct a binary tree for table lookup. Figure 1(b) illustrates the binary tree for the prefixes listed in Table 1. In this binary tree, each leaf node corresponds to an elementary interval. It is clear that the binary tree shown in Fig. 1(b) gives good worst-case performance. However, the average performance can be improved if access frequencies are taken into consideration. For example, Gupta, *et al.* [12] suggested to use the weight-balanced binary tree to take advantage of access frequencies to reduce the average lookup time. The weight-balanced binary tree is described below.

	Prefix	Start	End
P_1	*	0000	1111
P_2	0*	0000	0111
P_3	000*	0000	0001
P_4	0101	0101	0101
P_5	1*	1000	1111
P_6	110*	1100	1101

I_1: 0000~1111
I_2: 0000~0111
I_3: 0000~0001
I_4: 0101
I_5: 1000~1111
I_6: 1100~1101

Table 1. Routing table example with 4-bit prefixes and the corresponding intervals $I_1 \ldots I_6$.

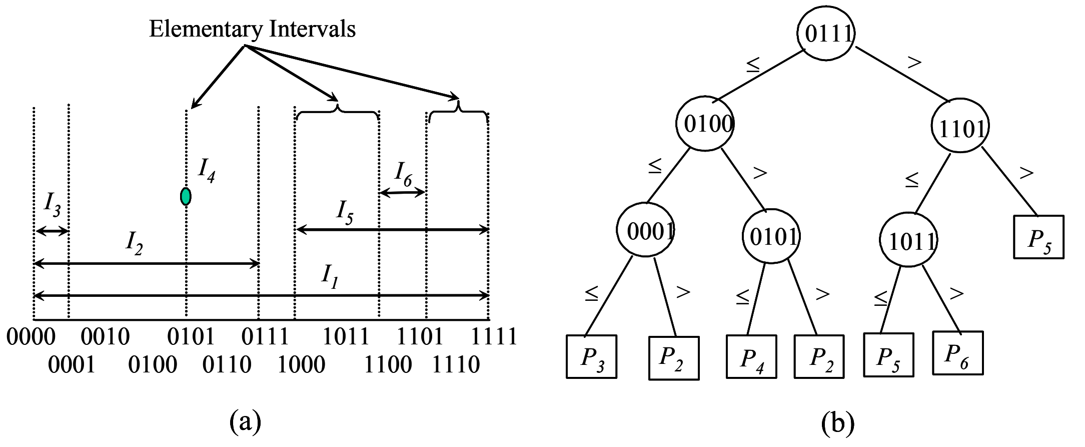


Figure 1. (a) Elementary Intervals and (b) Corresponding binary tree for the prefixes listed in Table 1.

Assume that there are l elementary intervals. Let w_i denote the access frequency (or weight) of elementary interval i and $[w_1\ w_2\ \ldots\ w_l]$ the set of weights for all elementary intervals. The left sub-tree of the weight-balanced binary tree contains elementary intervals 1, 2, …, n (and the right sub-tree contains elementary intervals n+1, n+2, …, l) if and only if n satisfies

$$\left|\sum_{i=1}^{n} w_i - \sum_{i=n+1}^{l} w_i\right| = \min_{1\le k<l}\left|\sum_{i=1}^{k} w_i - \sum_{i=k+1}^{l} w_i\right|. \tag{1}$$

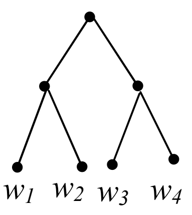
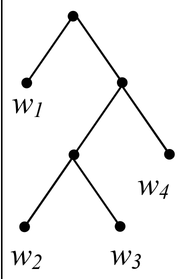
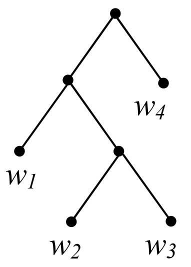
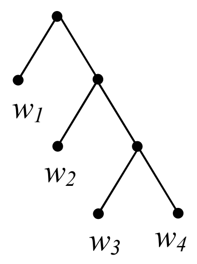
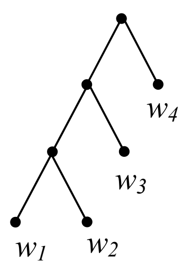
A tie can be broken arbitrarily. The same procedure applies recursively to both the left and the right sub-trees until every sub-tree contains a single elementary interval. For convenience, the procedure for generating the weight-balanced binary tree is referred to as the weight-balancing procedure. Note that the weight-balanced binary tree is easy to construct. Besides, compared with a tree which does not take into account access frequencies, the average search time could be significantly reduced. However, we found that the average performance can be further improved with simple adjustment of the weight-balanced binary tree. The adjustment algorithm is presented in the following section.

3. Adjusted Weight-Balanced Tree

For ease of description, consider a binary tree consisting of four leaf nodes with weight vector [w_1 w_2 w_3 w_4]. There are five possible binary tree patterns with four leaf nodes as shown in Table 2. Let $l_T(i)$ denote the distance of leaf node i from the root on tree T. For comparison purpose, we define the total lookup time of tree T as $W(T) = \sum_{i=1}^{4} w_i l_T(i)$. Notice that $W(T) \Big/ \sum_{i=1}^{4} w_i$ represents the average lookup time. As listed in Table 2, the total lookup time for tree pattern *a* is equal to $2(w_1+w_2+w_3+w_4)$ and that for pattern *b* is equal to $w_1+2(w_2+w_3)+w_4$. Thus, pattern *b* is better than pattern *a* in terms of average lookup time if and only if $w_1 > w_2+w_3$. In Table 2, we also provide the criteria for each tree pattern to be the optimum (i.e., least total lookup time) among the five tree patterns.

The weight-balancing procedure, unfortunately, does not always results in the optimal tree pattern. For example, assume that the weight vector is [23 5 2 27]. Figures 2(a) and 2(b) show, respectively, the weight-balanced binary tree and the optimal binary tree. They are different and the gain, which is defined as the difference of total lookup times, of the optimal binary tree is 20. From a different viewpoint, the average lookup time of the weight-balanced binary tree is 114/57 = 2 and that of the optimal tree is 94/57 = 1.6491. The

Table 2. The possible tree patterns, the total lookup time required, and the criteria for optimality over the complete binary tree of depth 2.

	a	***b***	***c***	***d***	***e***
Tree Pattern					
Total Lookup Time	$2(w_1+w_2+w_3+w_4)$	$w_1+2w_4+3(w_2+w_3)$	$2w_1+w_4+3(w_2+w_3)$	$w_1+2w_2+3(w_3+w_4)$	$w_4+2w_3+3(w_1+w_2)$
Criteria for optimality	$\begin{cases} w_1 < w_2+w_3 \\ w_4 < w_2+w_3 \\ w_1 < w_3+w_4 \\ w_4 < w_1+w_2 \end{cases}$	$\begin{cases} w_2+w_3 < w_1 \\ w_2 < w_4 \le w_1 \end{cases}$	$\begin{cases} w_2+w_3 < w_4 \\ w_3 < w_1 \le w_4 \end{cases}$	$\begin{cases} w_3+w_4 < w_1 \\ w_4 < w_2 \end{cases}$	$\begin{cases} w_1+w_2 < w_4 \\ w_1 < w_3 \end{cases}$

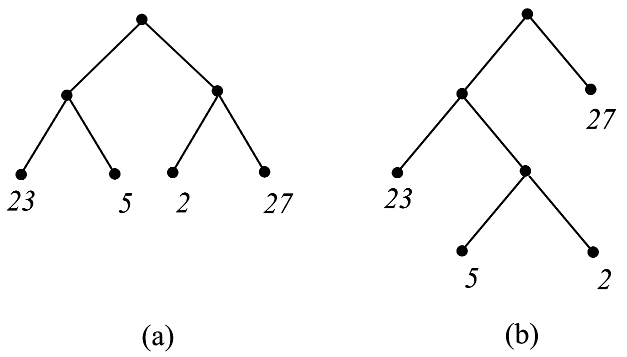


Figure 2. (a) A weight balanced binary tree and (b) the corresponding optimal binary tree.

following Proposition shows that, if tree pattern *a*, *d* or *e* is optimal, then the weight-balanced binary tree is identical to the optimal pattern.

Proposition 1. If tree pattern *a*, *d* or *e* is optimal, then the weight-balancing procedure results in the optimal pattern.

Proof: For tree pattern *a* to be optimal, it must hold that

$$\begin{cases} w_1 < w_2 + w_3 \text{ (pattern } a \text{ is better than pattern } b), \\ w_4 < w_2 + w_3 \text{ (pattern } a \text{ is better than pattern } c), \\ w_1 < w_3 + w_4 \text{ (pattern } a \text{ is better than pattern } d), \text{ and} \\ w_4 < w_1 + w_2 \text{ (pattern } a \text{ is better than pattern } e). \end{cases}$$

The third inequality, i.e., $w_1 < w_3 + w_4$, implies that the left sub-tree of the weight-balanced binary tree contains at least two leaf nodes. Similarly, the fourth inequality implies that the right sub-tree contains at least two leaf nodes. As a result, both the left sub-tree and the right sub-tree contain exactly two leaf nodes. In other words, the weight-balanced binary tree is identical to pattern *a*.

For tree *d* to be the optimal pattern, it must hold that

$$\begin{cases} w_3 + w_4 < w_1 \text{ (pattern } d \text{ is better than pattern } a), \\ w_4 < w_2 \qquad \text{(pattern } d \text{ is better than pattern } b), \\ 2w_4 < w_1 + w_2 \text{ (pattern } d \text{ is better than pattern } c), \text{ and} \\ w_3 + 2w_4 < 2w_1 + w_2 \text{ (pattern } d \text{ is better than pattern } e). \end{cases}$$

The first inequality, i.e., $w_3 + w_4 < w_1$, implies that the weight-balanced binary tree is either pattern *b* or *d*. It has to be pattern *d* because of the inequality $w_4 < w_2$. The Proof for pattern *e* is similar. This completes the proof of Proposition 1. □

Proposition 2 below states that the weight-balancing procedure might result in pattern *a* if the optimal pattern is either *b* or *c*. As a result, one may have adjustment gain by altering pattern *a* to the optimal pattern. Besides, according to Proposition 1, this is the only possibility for adjustment gain. Since the criteria for optimality is quite simple, adjustment can be easily done without much effort.

Proposition 2. If pattern *b* (or pattern *c*) is optimal, then the weight-balancing procedure results in pattern *a* or *b* (pattern *a* or *c*).

Proof: For pattern *b* to be optimal, it must hold that

$$\begin{cases} w_2 + w_3 < w_1 & \text{(pattern } b \text{ is better than pattern } a\text{),} \\ w_4 < w_1 & \text{(pattern } b \text{ is better than pattern } c\text{),} \\ w_2 < w_4 & \text{(pattern } b \text{ is better than pattern } d\text{), and} \\ w_3 + w_4 < 2w_1 & \text{(pattern } b \text{ is better than pattern } e\text{).} \end{cases}$$

The first inequality, i.e., $w_2 + w_3 < w_1$, implies that the weight-balanced binary tree is either pattern *a*, *b* or *d*. Since $w_2 < w_4$, pattern *d* will not be generated by the weight-balancing procedure. To show that the weight-balancing procedure may result in pattern *a*, it suffices by giving an example. Consider a weight vector $[x\ 1\ 1\ x\text{-}1]$ with $x > 2$. It is clear that the corresponding optimum tree is pattern *b*. However, the weight-balancing procedure generates pattern *a*. Therefore, the weight-balancing procedure could result in tree pattern *a* or *b* if *b* is the optimal pattern. The Proof for pattern *c* is similar. This completes the proof of Proposition 2. □

One can generalize the concept of adjustment to a tree with an arbitrary number of leaf nodes. However, the number of possible tree patterns grows explosively as the number of leaf nodes becomes large. Let n_i denote the number of possible tree patterns when there are *i* leaf nodes. It is not difficult to show that $n_i = \sum_{j=1}^{i-1} n_j n_{i-j}$ with $n_1 = n_2 = 1$. As an example, we have $n_8 = 429$. Because of the complexity, we will focus on adjusting trees with four leaf nodes. In other words, a tree or a sub-tree is adjusted after the weight-balancing procedure is executed for every two steps. Proposition 3 below proves that the adjusted tree always gives a better average performance than the non-adjusted tree.

Proposition 3. Given a weight balanced binary tree *T*. Let $\hat{T}$ be the adjusted weight-balanced binary tree. We have $W(\hat{T}) \le W(T)$.

Proof: Define the weight of an internal node as the sum of the weights of all leaf nodes under it. Recall that the weight-balancing procedure is recursively applied to derive the weight-balanced tree. As a result, before the weight-balancing procedure terminates, there are "leaf nodes" that can be further expanded. For convenience, we call such a tree a partially expanded tree. We will prove that, for all "corresponding" partially expanded trees, the total lookup time with adjustment is smaller than or equal to that without adjustment.

According to Propositions 1 and 2, after the first two steps of expansion, the total lookup time for the partially expanded tree with adjustment is smaller than or equal to that for the partially expanded tree without adjustment. The proof is completed if there is no leaf node of the partially expanded tree that can be further expanded, i.e., if the partially expanded tree is actually fully expanded. So, consider the case that there is at least one leaf node that can be further expanded. Let T_1, T_2, T_3, and T_4 denote the four leaf nodes of the partially expanded tree without adjustment. Similarly, let $\hat{T}_1$, $\hat{T}_2$, $\hat{T}_3$, and $\hat{T}_4$ denote the four leaf nodes of the partially expanded tree with adjustment. A key point for the proof is that the adjustment algorithm only alters tree pattern. It does not change the weight of any (internal) node. In other words, T_i is identical to $\hat{T}_i$, except that they might be in different levels. Let $l(T_i)$ and $l(\hat{T}_i)$ denote the levels of nodes T_i and $\hat{T}_i$, respectively. Further, let W_{T_i} and $W_{\hat{T}_i}$ represent the total weight of leaf nodes under nodes T_i and $\hat{T}_i$, respectively. As mentioned above, we have $W_{T_i} = W_{\hat{T}_i}$, for all i.

Without loss of generality, assume that the leftmost leaf node, i.e., T_1 (or $\hat{T}_1$), can be further expanded. To avoid trivial cases, we assume that the node can be further expanded by at least two steps. According to the adjustment algorithm, the tree is adjusted after two steps of expansion. Let T_{11}, T_{12}, T_{13}, and T_{14} denote the leaf nodes under T_1 after two steps of expansions. Also, let $l(T_{1i})$ denote the distance from node T_{1i} to node T_1. As a result, for the new partially expanded tree without adjustment, the total lookup time becomes $\sum_{i=2}^{4} W_{T_i} l(T_i) + \sum_{j=1}^{4} W_{T_{1j}} \left(l(T_{1j}) + l(T_1) \right)$. Similarly, let $l(\hat{T}_{1i})$ denote the distance of node $\hat{T}_{1i}$ to node $\hat{T}_1$. The total lookup time of the new partially expanded tree with adjustment becomes $\sum_{i=2}^{4} W_{\hat{T}_i} l(\hat{T}_i) + \sum_{j=1}^{4} W_{\hat{T}_{1j}} \left(l(\hat{T}_{1j}) + l(\hat{T}_1) \right)$. Since the adjustment algorithm guarantees that $\sum_{j=1}^{4} W_{\hat{T}_{1j}} l(\hat{T}_{1j}) \leq \sum_{j=1}^{4} W_{\hat{T}_{1j}} l(\hat{T}_{1j})$. Thus, we have

$$\sum_{i=2}^{4} W_{\hat{T}_i} l(\hat{T}_i) + \sum_{j=1}^{4} W_{\hat{T}_{1j}} \left(l(\hat{T}_{1j}) + l(\hat{T}_1) \right) \leq \sum_{i=2}^{4} W_{T_i} l(T_i) + \sum_{j=1}^{4} W_{T_{1j}} \left(l(T_{1j}) + l(T_1) \right).$$

In other words, the total lookup time of the partially expanded tree with adjustment is smaller than or equal to that of the partially expanded tree without adjustment. The same arguments can be applied repeatedly until the tree is fully expanded. This completes the proof of Proposition 3. □

4. Adjustment Gain

Consider a tree T with n leaf nodes. Let $[w_1\ w_2\ \ldots w_n]$ be the weight vector. Further, let $G_T(\hat{T})$ denote the gain obtained by altering tree T to tree $\hat{T}$ and is defined as

$$G_T(\hat{T}) = \sum_{i=1}^{n} w_i l_T(i) - \sum_{i=1}^{n} w_i l_{\hat{T}}(i) = W(T) - W(\hat{T}), \tag{2}$$

The ratio of $G_T(\hat{T})$ to $W(T)$, called the gain ratio, represents the percentage of improvement in average lookup time.

Based on Proposition 1 we know that, for a tree with four leaf nodes, the only possible adjustment is to alter from pattern *a* to either pattern *b* or *c*. Since patterns *b* and *c* are symmetric, we can focus on altering from pattern *a* to pattern *b*.

For $n = 4$, the maximum gain ratio is given by $(x-2)/2(2x+1)$, where x is a non-negative integer greater than or equal to 2. It is achieved by altering tree pattern *a* to tree pattern *b* when the weight vector is $[x\ 1\ 1\ x-1]$. This can be derived as follows. When a tree is altered from pattern *a* to pattern *b*, the gain ratio is given by $\left(w_1 - (w_2 + w_3)\right) \Big/ 2\sum_{i=1}^{4} w_i$. To maximize the gain ratio, w_2 and w_3 have to be as small as possible. That is, we should have $w_2 = w_3 = 1$. Since the weight-balanced binary tree must be of pattern *a*, w_1 and w_4 cannot differ by more than 1. Therefore, the weight vector which maximizes the gain ratio is $[x\ 1\ 1\ x-1]$. It is obvious that as x increases, the maximum gain ratio increases and the limiting value is 1/4. Another weight vector which also results in a limiting gain ratio of 1/4 is $[x\ 1\ 1\ x]$. For ease of description, we will use this vector for the rest of this section.

A tree is said to be a k-level tree if the longest distance from root to any leaf node is k. For examples, in Table 2, pattern *a* is a 2-level tree and all the other patterns are *3*-level trees. One can easily show that pattern *a* is the only 2-level tree that can be adjusted for performance improvement. We call a k-level tree which results in the largest gain ratio the worst pattern of level k. For example, pattern *a* in Table 2 is the worst pattern of level 2.

Let us consider the maximum adjustment gain for a 3-level tree. The left sub-tree of a 3-level tree is either a level-1 or a level-2 tree. Therefore, the left sub-tree must be the worst pattern of level 2 to achieve the maximum adjustment gain. Similarly, in order to achieve the maximum adjustment gain, the right sub-tree is also the worst pattern of level 2. As a consequence, the worst patterns of level 3 must have eight leaf nodes and its left sub-tree and right sub-tree are the worst patterns of level 2. Figure 3 shows the worst pattern of level 3 and its corresponding tree after adjustment. Note that, for the weight-balancing procedure to generate the complete binary tree as shown in Figure 3, the total weight under the left sub-tree equals that under the right sub-tree. That is, the weight vector for the worst pattern of level 3

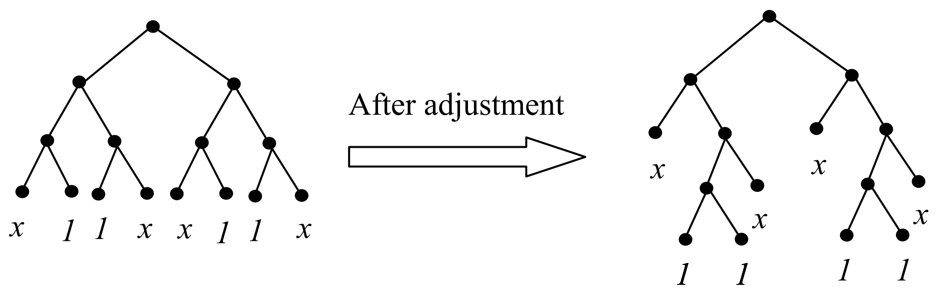


Figure 3. The worst pattern of level 3 and its corresponding tree after adjustment.

should be [x 1 1 x x 1 1 x]. The maximum gain ratio for a 3-level tree is thus given by (x-2)/(2(x+1)).

Consider a 4-level tree now. One might guess that the maximum gain ratio is achieved for a complete 4-level tree where the left and the right sub-trees are the worst patterns of level 3. This is not correct. Notice that the maximum gain ratio for the worst pattern of level 2 is larger than that for the worst pattern of level 3. Therefore, to achieve the maximum gain ratio, one of the two sub-trees should be the worst pattern of level 2 and the other the worst pattern of level 3, as illustrated in Figure 4. Again, for the weight-balancing procedure to generate the tree shown in Figure 4, we have to assign appropriate weights. Basically, the total weight of the left sub-tree is equal to that of the right sub-tree. As a result, the weight vector should be [2x+1 1 1 2x+1 x 1 1 x x 1 1 x] and the maximum gain ratio is equal to (4x-5)/(28x+28).

Similarly, to achieve the maximum gain ratio for a 5-level tree, the tree should have a worst pattern of level 2 as a sub-tree and a worst pattern of level 4 as another sub-tree. Again, the weights have to be appropriately assigned. After some calculations, it can be determined to be [4x+3 1 1 4x+3 2x+1 1 1 2x+1 x 1 1 x x 1 1 x] and thus the maximum gain ratio is equal to (8x-4)/(60x+60).

Let us consider now k-level trees. Without loss of generality, we assume that the left sub-tree is always shorter than the right sub-tree. From the above discussions, we know that,

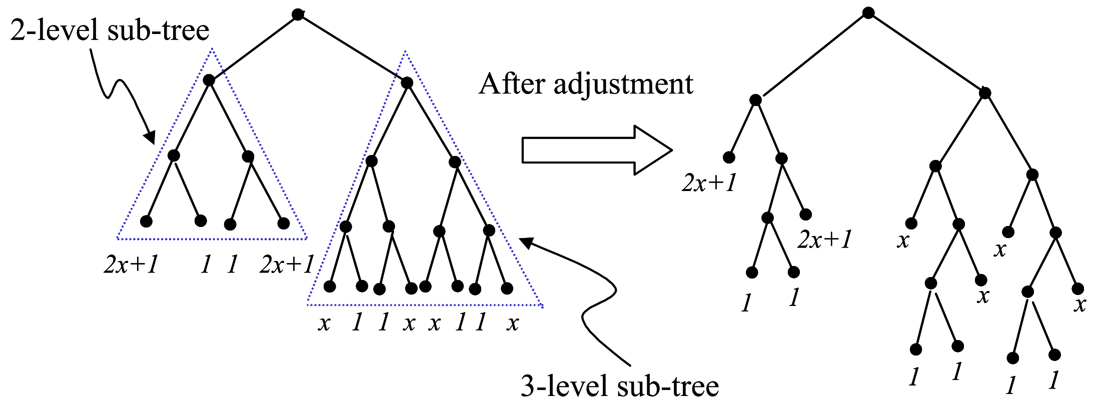


Figure 4. An example of 4-level binary tree that results in the maximum gain ratio.

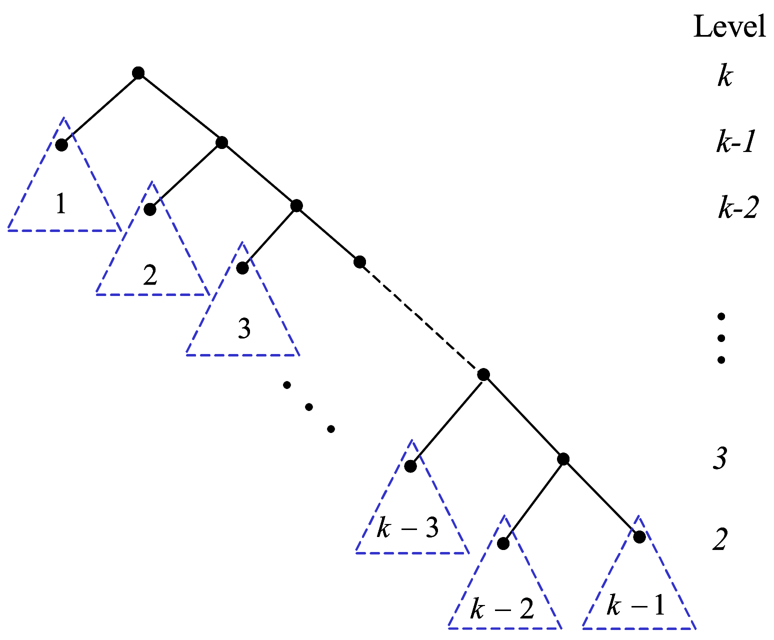


Figure 5. The *k*-level worst pattern tree.

to achieve the maximum gain ratio, it should have the 2-level worst pattern tree as the left sub-tree and the (*k*-1)-level worst pattern tree as the right sub-tree. Figure 5 shows the recursive construction of the *k*-level worst pattern tree. Clearly, the *k*-level worst pattern tree consists of *k*-1 2-level worst pattern sub-trees (see Figure 4 for an example of *k* = 4). Let us

number these 2-level worst pattern sub-trees from left to right. After some calculations, for a *k*-level worst pattern tree, the weight vector of the *n*-th 2-level worst pattern sub-tree is [$2^{k-n-2}(x+1)-1$ 1 1 $2^{k-n-2}(x+1)-1$]. Hence the maximum gain ratio is given by

$$\frac{\sum_{n=1}^{k-2}\left(\left(2^{k-n-2}(x+1)-1\right)-2\right)+(x-2)}{\sum_{n=1}^{k-2}2^{k-n-1}(x+1)(n+2)+2(x+1)k}=\frac{2^{k-2}(x+1)-3(k-1)}{\left(2^{k+1}-4\right)(x+1)}. \qquad (3)$$

As x goes to infinity, the maximum gain ratio is equal to $\frac{1}{\left(8-2^{-(k-4)}\right)}$.

5. Conclusion

We have presented in this paper a simple adjustment algorithm for the weight-balanced binary tree to improve the average performance in IP routing table lookup. Through analysis, we showed that the improvement could be as large as 25% for a 2-level tree or larger than 12.5% for an arbitrary *k*-level tree. Since the gain decreases as the level of a tree increases, in practice one should focus on adjusting the leaf nodes that are within a short distance (say, 8) from the root. One possible further research topic, which is currently under investigation, is to apply the weight-balancing procedure with adjustment to multi-field packet classification.

6. References

[1]. M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," in *Proceedings of ACM SIGCOMM'97*, 1997.

[2]. P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," in *Proceedings of ACM SIGCOMM'99*, pp. 147-160, 1999.

[3]. V. Srinivasan and G. Varghese, "Fast IP Lookups Using Controlled Prefix Expansion," in *ACM TOCS*, vol. 17, pp. 1-40, Feb. 1999.

[4]. B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," in *IEEE/ACM Transactions on Networking*, Vol. 7, No. 3, pp. 324-334, June 1999.

[5]. P. Gupta, S. Lin, and N. Mckeown, "Routing Lookups in hardware at Memory Access Speeds," in *Proceedings of INFOCOM'98*.

[6]. S. Sikka and G. Varghese, "Memory-Efficient State Lookups with Fast Updates," in *Proceedings of SIGCOMM'00*, 2000.

[7]. S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," in *IEEE Journal on Selected Areas in Communications*, Vol. 17, pp. 1083-1092, June 1999.

[8]. C. Pratridge, "Locality and Route Caches," in *Proceedings of NSF Workshop on Internet Statistics Measurement and Analysis*, 1996. (http://www.caida.org/ISMA/Positions/partridge.html)

[9]. T. Chiueh and P. Pradhan, "High Performance IP Routing Table Lookup Using CPU Caching," in *Proceedings of IEEE INFOCOM*, April 1999.

[10]. T. Chiueh and P. Pradhan, "Cache Memory Design for Network Processors," in *Proceedings of High-Performance Computer Architecture*, 2000, pp. 409-418.

[11]. G. Cheung and S. McCanne, "Optimal Routing Table Design for IP Address Lookups under Memory Constraints," in *Proceedings of IEEE INFOCOM*, 1999.

[12]. P. Gupta, B. Prabhakar, and S. Boyd, "Near-Optimal Routing Lookups with Bounded Worst Case Performance," in *Proceedings of IEEE INFOCOM*, 2000.

[13]. C. L. Liu, *Elements of Discrete Mathematics*, McGraw-Hill, 1986.

[14]. T-Y.C. Woo, "A Modular Approach to Packet Classification: Algorithms and Results," in *Proceedings of IEEE INFOCOM*, 2000, pp. 1213-1222.

[15]. V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification Using Tuple Space Search," in *ACM Computer Communication Review*, 1999.

[16]. T.V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-Dimensional Range Matching," in *Proceedings of ACM SIGCOMM'98*, 1998.