CHAPTER 3

Minimum average and bounded worst-case routing lookup time on binary search trees

1 Introduction

Most work on routing lookups [17][31][69][93] has focused on the development of data structures and algorithms for minimizing the worst-case lookup time, given a forwarding table and some storage space constraints. Minimizing the worst-case lookup time is attractive because it does not require packets to be queued before lookup. This enables simplicity and helps bound the delay of the packet through the router. However, it suffices to minimize the average lookup time for some types of traffic, such as “best-effort” traffic.[1] This presents opportunities for higher overall lookup performance because an average case constraint is less stringent than the worst-case constraint. This chapter presents two such algorithms for minimizing the average lookup time — in particular, lookup algorithms that adapt their binary search tree data structure based on the observed statistical

1. Best-effort traffic comprises the highest proportion of Internet traffic today. This is generally expected to continue to remain true in the near future.

properties of recent lookup results in order to achieve higher performance. The exact amount of performance improvement obtained using the proposed algorithms depends on the forwarding table and the traffic patterns. For example, experiments using one set of parameters show a reduction of 42% in the average number of memory accesses per lookup than those obtained by worst-case lookup time minimization algorithms. Another benefit of these algorithms is the "near-perfect" load balancing property of the resulting tree data structures. This enables, for example, doubling the lookup speed by replicating only the root node of the tree, and assigning one lookup engine each to the left and right subtrees.

As we saw in Chapter 2, most lookup algorithms use a tree-based data structure. A natural question to ask is: "What is the best tree data structure for a given forwarding table?". This chapter considers this question in the context of binary search trees as constructed by the lookup algorithm discussed in Section 2.2.6 of Chapter 2. The two algorithms proposed in this chapter adapt the shape of the binary search tree constructed by the lookup algorithm of Section 2.2.6 of Chapter 2. The tree is redrawn based on the statistics gathered on the number of accesses to prefixes in the forwarding table, with the aim of minimizing the average lookup time. However, the use of a binary search tree data structure brings up a problem — depending on the distribution of prefix access probabilities, it is possible for the worst-case depth of a redrawn binary search tree to be as large as $2m - 1$, where m is the total number of forwarding table entries, and is close to 98,000 [136] at the time of writing. The worst-case lookup time can not be completely neglected — if it takes very long to lookup even one incoming packet, a large number of packets arriving shortly thereafter must be queued until the packet has completed its lookup. Practical router design considerations (such as silicon and board real-estate resources) limit the maximum size of this queue, and hence make bounding the worst-case lookup time highly desirable. Bounding the worst-case performance also enables bounding packet delay in the router

and hence in the network. It is the objective of this chapter to devise algorithms on binary search trees that *minimize the average* lookup time while *keeping the worst-case* lookup time *smaller* than a pre-specified maximum.

The approach taken in this chapter has a limitation that it cannot be used in some hardware-based designs where the designer desires a fixed routing lookup time for all packets. The approach of this chapter can only be used when the router designer wants to minimize the average, subject to a maximum lookup time. Thus, the designer should be willing to buffer incoming packets before sending them to the lookup engine in order to absorb the variability in the lookup times of different packets.

1.1 Organization of the chapter

Section 2 sets up the formal minimization problem. Sections 3 and 4 describe the two proposed algorithms and analyze their performance. Section 5 discusses the load balancing characteristics of these algorithms, and Section 6 provides experimental results on publicly available routing tables and a packet trace. Section 7 discusses related work, and Section 8 concludes with a summary and contributions of this chapter.

2 Problem statement

Recall that the binary search algorithm [49], discussed in Section 2.2.6 of Chapter 2, views each prefix as an interval on the IP number line. The union of the end points of these intervals partitions the number line into a set of disjoint intervals, called basic intervals (see, for example, Figure 2.7 of Chapter 2). The algorithm precomputes the longest prefix for every basic interval in the partition, and associates every basic interval with its left end-point. The distinct number of end-points for m prefixes is at most $n = 2m$. These end-points are kept in a sorted list. Given a point, P, on the number line representing an incoming packet, the longest prefix matching problem is solved by using binary search on

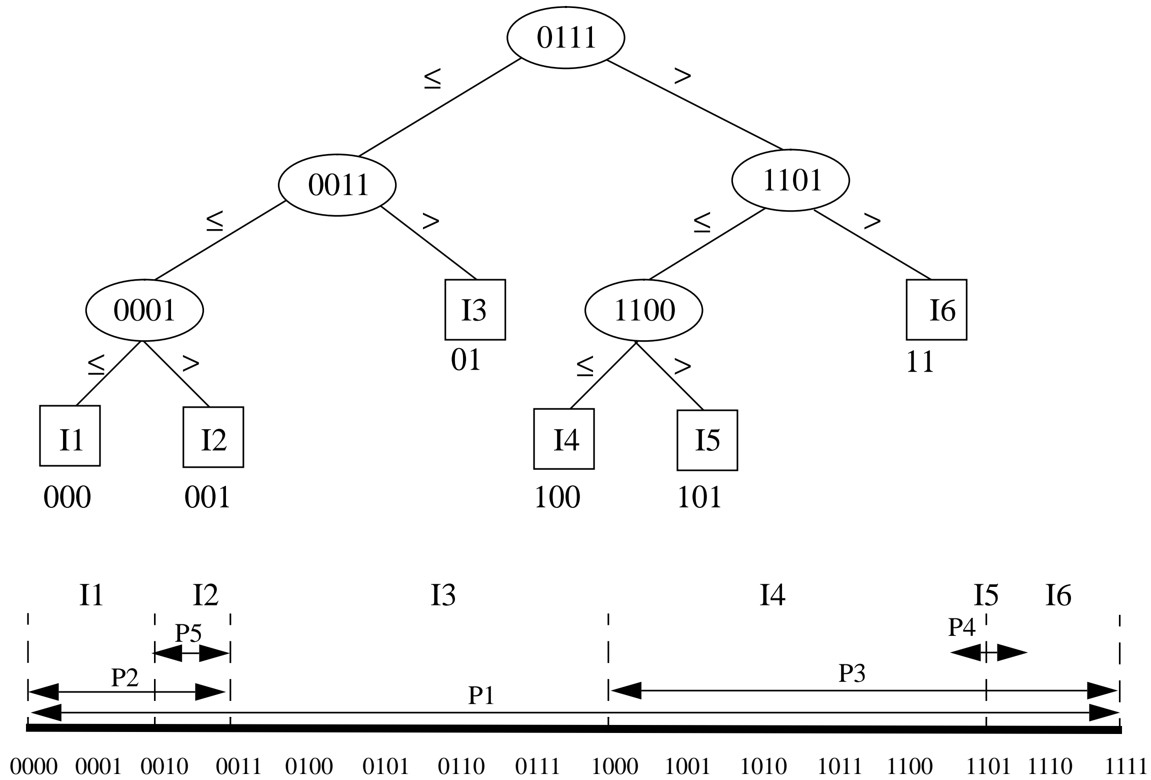

Figure 3.1 The binary search tree corresponding to the forwarding table in Table 3.1. The bit-strings in bold are the binary codes of the leaves.

the sorted list to find the end-point in the list that is closest to, but not greater than P. Binary search is performed by the following binary tree data structure: the leaves (external nodes) of the tree store the left end-points in order from left to right, and the internal nodes of the tree contain suitably chosen values to guide the search process to the appropriate child node. This binary search tree for m prefixes takes $O(m)$ storage space and has a maximum depth of $O(\log(2m))$.[1]

Example 3.1: An example of a forwarding table with 4-bit prefixes is shown in Table 3.1, and the corresponding partition of the IP number line and the binary search tree is shown in Figure 3.1.

1. All logarithms in this chapter are to the base 2.

TABLE 3.1. An example forwarding table.

	Prefix	Interval start-point	Interval end-point
P1	*	0000	1111
P2	00*	0000	0011
P3	1*	1000	1111
P4	1101	1101	1101
P5	001*	0010	0011

The key idea used in this chapter is that the average lookup time in the binary search tree data structure can be decreased by making use of the frequency with which a certain forwarding table entry is accessed in the router. We note that most routers already maintain such per-entry statistics. Hence, minimizing routing lookup times by making use of this information comes at no extra data collection cost. A natural question to ask is: 'Given the frequency with which the leaves of a tree are accessed, what is the best binary search tree — i.e., the tree with the minimum average depth?' Viewing it this way, the problem is readily recognized to be one of minimizing the average weighted depth of a binary tree whose leaves are weighted by the probabilities associated with the basic intervals represented by the leaves. The minimization is to be carried over all possible binary trees that can be constructed with the given number and weights of the leaves.

This problem is analogous to the design of efficient codes (see Chapter 5 of Cover and Thomas [14]), and so we briefly explain here the relationship between the two problems. A binary search tree is referred to as an *alphabetic tree*, and the leaves of the tree the *letters* of that alphabet. Each leaf is assigned a binary codeword depending on its position in the tree. The length of the codeword of a symbol is equal to the depth of the corresponding leaf in the tree. For the example in Figure 3.1, the codeword associated with interval *I1* is

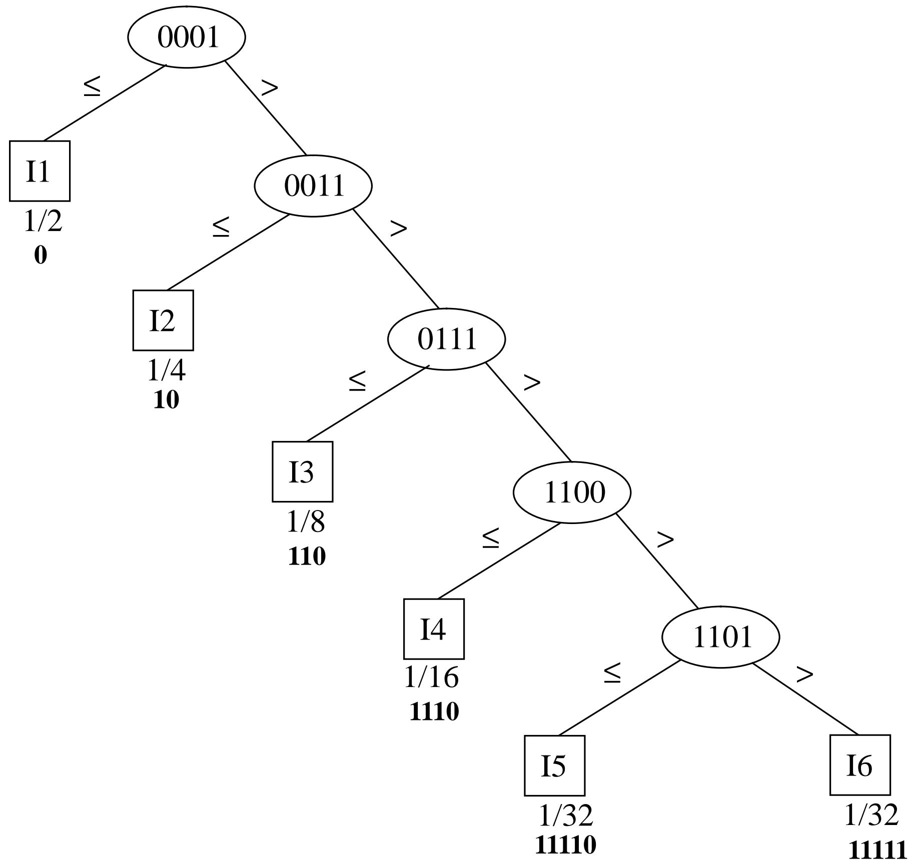

Figure 3.2 The optimal binary search tree (i.e., one with the minimum average weighted depth) corresponding to the tree in Figure 3.1 when leaf probabilities are as shown. The binary codewords are shown in bold.

000 and that associated with interval *I5* is 101, where a bit in the codeword is 0 (respectively 1) for the left (respectively right) branch at the corresponding node.

A *prefix* code satisfies the property that no two codes are prefixes of each other. An *alphabetic* code is a prefix code in which the n letters are ordered lexicographically on the leaves of the resulting binary tree. In other words, if letter A appears before letter B in the alphabet, then the codeword associated with letter A has a value of smaller magnitude than the codeword associated with letter B. Designing a code for an alphabet is equivalent to constructing a tree for the letters of the alphabet. With a letter corresponding to an inter-

val, the lookup problem translates to: "Find a minimum average length alphabetic prefix code (or tree) for an n-letter alphabet."

Example 3.2: If the intervals *I1* through *I6* in Figure 3.1 are accessed with probabilities 1/2, 1/4, 1/8, 1/16, 1/32 and 1/32 respectively, then the best (i.e., optimal) alphabetic tree corresponding to these probabilities (or weights) is shown in Figure 3.2. The codeword for *I1* is now 0 and that of *I5* is 11110. Since *I1* is accessed with a greater probability than *I5*, it has been placed higher up in the tree, and thus has a shorter codeword

The average length of a general prefix code for a given set of probabilities can be minimized using the Huffman coding algorithm [39]. However, Huffman's algorithm does not necessarily maintain the alphabetic order of the input data set. This causes implementational problems, as simple comparison queries are not possible at internal nodes to guide the binary search algorithm. Instead, at an internal node of a Huffman tree, one needs to ask for memberships in arbitrary subsets of the alphabet to proceed to the next level. Because this is as hard as the original search problem, it is not feasible to use Huffman's algorithm.

As mentioned previously, we wish to bound the maximum codeword length (i.e., the maximum depth of the tree) to make the solution useful in practice. This can now be better understood: an optimal alphabetic tree for n letters can have a maximum depth (the root is assumed to be at depth 0) of $n-1$ (see, for instance, Figure 3.2 with $n = 6$). This is unacceptable in practice because we have seen that $n = 2m$, and the value of m, the size of the forwarding table, could be as high as 98,000 [136]. Furthermore, any change in the network topology or in the distribution of incoming packet addresses can lead to a large increase in the access frequency of a deep leaf. It is therefore highly desirable to have a small upper bound on the maximum depth of the alphabetic tree. Therefore, well-known algorithms for finding an optimal alphabetic tree such as those in [27][36][37] which do

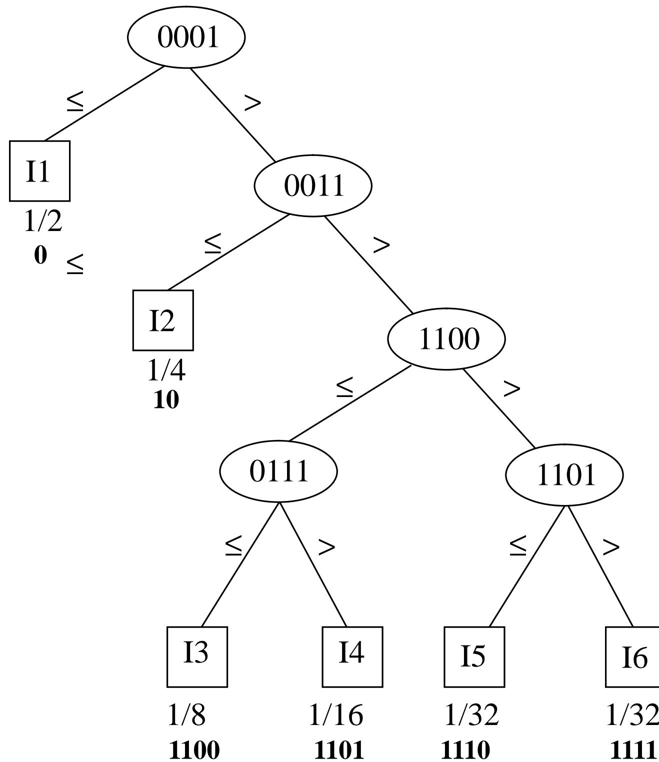

Figure 3.3 The optimal binary search tree with a depth-constraint of 4, corresponding to the tree in Figure 3.1.

not incorporate a maximum depth-constraint cannot be used in this chapter's setting. Here is an example to understand this last point better.

Example 3.3: The alphabetic tree in Figure 3.2 is optimal if the intervals I1 through I6 shown in the binary tree of Figure 3.1 are accessed with probabilities {1/2, 1/4, 1/8, 1/16, 1/32, 1/32} respectively. For these probabilities, the average lookup time is 1.9375,[1] while the maximum depth is 5. If we impose a maximum depth-constraint of 4, then we need to redraw the tree to obtain the optimal tree that has minimum average weighted depth and has maximum depth no greater than 4. This tree is shown in Figure 3.3 where the average lookup time is calculated to be 2.

The general minimization problem can now be stated as follows:

1. $1.9375 = 1 \cdot (1/2) + 2 \cdot (1/4) + 3 \cdot (1/8) + 4 \cdot (1/16) + 5 \cdot (1/32) + 5 \cdot (1/32)$

Choose $\{l_i\}_{i=1}^{n}$ in order to minimize $C = \sum_{i=1}^{n} l_i \cdot p_i$, such that $l_i \le D \ \forall i$, and $\{l_i\}_{i=1}^{n}$ gives rise to an alphabetic tree for n intervals where p_i is the access probability of the ith interval, and l_i is the length of its codeword, i.e., the number of comparisons required to lookup a packet in the ith interval.

The smallest possible value of C is the entropy [14], $H(p)$, of the set of probabilities $\{p_i\}$, where $H(p) = -\sum_i p_i \log p_i$. It is usually the case that C is larger than $H(p)$ for depth-constrained alphabetic trees.[1] Finding fast algorithms for computing optimal depth-constrained binary trees (without the alphabetic constraint) is known to be a hard problem, and good approximate solutions are appearing only now [59][60][61], almost 40 years after the original Huffman algorithm [39]. Imposing the alphabetic constraint renders the problem harder [27][28][35][109]. Still, an optimal algorithm, proposed by Larmore and Przytycka [50], finds the best depth-constrained alphabetic tree in $O(nD\log n)$ time. Despite its optimality, the algorithm is complicated and difficult to implement.[2]

In light of this, our goal is to find a practical and provably good approximate solution to the problem of computing optimal depth-constrained alphabetic trees. Such a solution should be simpler to find than an optimal solution. More importantly, it should be much simpler to implement. Also, as the probabilities associated with the intervals induced by routing prefixes change and are not known exactly, it does not seem to make much sense to solve the problem exactly for an optimal solution. As we will see later, one of the two near-optimal algorithms proposed in this chapter can be analytically proved to be requiring no more than two extra comparisons per lookup when compared to the optimal solution. In practice, this discrepancy has been found to be less than two (for both of the approximate algorithms). Hence, we refer to them as algorithms for *near-optimal depth-constrained alphabetic trees*, and describe them next.

---

1. The lower bound of entropy is achieved in general when there are no alphabetic or maximum depth-constraints.

2. The complexity formula $O(nD\log n)$ has large constant factors, as the implementation requires using a list of mergeable priority queues with priority queue operations such as *delete_min, merge, find* etc.

3 Algorithm MINDPQ

We first state two results from Yeung [114] as lemmas that we will use to develop algorithm MINDPQ. The first lemma states a necessary and sufficient condition for the existence of an alphabetic code with specified codeword lengths, and the second prescribes a method for constructing good, near-optimal trees (which are not depth-constrained).

Lemma 3.1 *(The Characteristic Inequality)*: There exists an alphabetic code with codeword lengths l_k if and only if $s_n \le 1$, where $s_k = c(s_{k-1}, 2^{-l_k}) + 2^{-l_k}$, $s_0 = 0$, and c is defined by $c(a,b) = \lceil a/b \rceil b$.

Proof: For a complete proof, see [114]. The basic idea is to construct a *canonical* coding tree, a tree in which the codewords are chosen lexicographically using the lengths l_i. For instance, suppose that $l_i = 4$ for some i, and in drawing the canonical tree we find the codeword corresponding to letter i to be 0010. If $l_{i+1} = 4$, then the codeword for letter $i+1$ will be chosen to be 0011; if $l_{i+1} = 3$, the codeword for letter $i+1$ is chosen to be 010; and if $l_{i+1} = 5$, the codeword for letter $i+1$ is chosen to be 00110. Clearly, the resulting tree will be alphabetic and Yeung's result verifies that this is possible if and only if the characteristic inequality defined above is satisfied by the lengths l_i.

The next lemma (also from [114]) considers the construction of good, near-optimal codes. Note that it does not produce alphabetic trees with prescribed maximum depths. That is the subject of this chapter.

Lemma 3.2 The minimum average length, C_{min}, of an alphabetic code on n letters, where the ith letter occurs with probability p_i satisfies: $H(p) \le C_{min} \le H(p) + 2 - p_1 - p_{min}$. Therefore, there exists an alphabetic tree on n letters with average code length within 2 bits of the entropy of the probability distribution of the letters.

Proof: The lower bound, $H(p)$, is obvious. For the upper bound, the code length l_k of the kth letter occurring with probability p_k is chosen to be:

$$l_k = \begin{cases} \lceil -\log p_k \rceil & k = 1, n \\ \lceil -\log p_k \rceil + 1 & 2 \le k \le n-1 \end{cases}$$

The proof in [114] verifies that these lengths satisfy the characteristic inequality of Lemma 3.1, and shows that a canonical coding tree constructed with these lengths has an average depth satisfying the upper bound.

We now return to our original problem of finding near-optimal depth-constrained alphabetic trees. Let D be the maximum allowed depth. Since the given set of probabilities $\{p_k\}$ might be such that $p_{min} = min_k\{p_k\} < 2^{-D}$, a direct application of Lemma 3.2 could yield a tree where the maximum depth is higher than D. To work around this problem, we transform the given probabilities p_k into another set of probabilities q_k such that $q_{min} = min_k\{q_k\} \ge 2^{-D}$. This allows us to apply the following variant of the scheme in Lemma 3.2 to obtain a near-optimal depth-constrained alphabetic tree with leaf probabilities q_k.

Given a probability vector q_k such that $q_{min} \ge 2^{-D}$, we construct a canonical alphabetic coding tree with the codeword length assignment to the kth letter given by:

$$l_k^* = \begin{cases} min(\lceil -\log q_k \rceil, D) & k = 1, n \\ min(\lceil -\log q_k \rceil + 1, D) & 2 \le k \le n-1 \end{cases} \quad (3.1)$$

Each codeword is clearly at most D bits long and the tree thus generated has a maximum depth of D. It remains to be shown that these codeword lengths yield an alphabetic tree. By Lemma 3.1 it suffices to show that the $\{l_k^*\}$ satisfy the characteristic inequality. This verification is deferred to Appendix B later in the thesis.

Proceeding, if the codeword lengths are given by $\{l_k^*\}$, the resulting alphabetic tree has an average length of $\sum_k p_k l_k^*$. Now,

$$\sum_k p_k l_k^* \le \sum_k p_k \log \frac{1}{q_k} + 2 = \sum_k p_k \log \frac{p_k}{q_k} - \sum_k p_k \log p_k + 2 = D(p \| q) + H(p) + 2 \tag{3.2}$$

where $D(p \| q)$ is the 'relative entropy' (see page 22 of [14]) between the probability distributions p and q, and $H(p)$ is the entropy of the probability distribution p. In order to minimize $\sum_k p_k l_k^*$, we must therefore choose $\{q_i\}$, given $\{p_i\}$, so as to minimize $D(p \| q)$.

3.1 The minimization problem

We are thus led to the following optimization problem:

Given $\{p_i\}$, choose $\{q_i\}$ in order to minimize $DPQ = D(p \| q) = \sum_i p_i \log (p_i / q_i)$ subject to $\sum_i q_i = 1, q_i \ge Q = 2^{-D} \forall i$.

Observe that the cost function $D(p \| q)$ is convex in (p, q) (see page 30 of [14]). Further, the constraint set is convex and compact. In fact, we note that the constraint set is defined by *linear* inequalities. Minimizing convex cost functions with linear constraints is a standard problem in optimization theory and is easily solved by using Lagrange multiplier methods (see, for example, Section 3.4 of Bertsekas [5]).

Accordingly, define the Lagrangean

$$L(q, \bar{\lambda}, \mu) = \sum p_i \log (p_i / q_i) + \sum \lambda_i (Q - q_i) + \mu\left(\sum q_i - 1\right)$$

Setting the partial derivatives with respect to q_i to zero at q_i^*, we get:

$$\left(\frac{\partial L}{\partial q_i} = 0\right) \Rightarrow q_i^* = \frac{p_i}{\mu - \lambda_i} \tag{3.3}$$

Putting this back in $L(q, \bar{\lambda}, \mu)$, we get the dual: $G(\bar{\lambda}, \mu) = \sum_i (p_i \log(\mu - \lambda_i) + \lambda_i Q) + (1 - \mu)$. Now minimizing $G(\bar{\lambda}, \mu)$ subject to $\lambda_i \geq 0$ and $\mu > \lambda_i \forall i$ gives:

$$\frac{\partial G}{\partial \mu} = 0 \Rightarrow \sum_i \frac{p_i}{\mu - \lambda_i} = 1$$

$$\frac{\partial G}{\partial \lambda_i} = 0 \forall i \Rightarrow Q = \frac{p_i}{\mu - \lambda_i},$$

which combined with the constraint that $\lambda_i \geq 0$ gives us $\lambda_i^* = max(0, \mu - p_i / Q)$. Substituting this in Equation 3.3, we get

$$q_i^* = max(p_i / \mu, Q) \tag{3.4}$$

To finish, we need to solve Equation 3.4 for $\mu = \mu^*$ under the constraint that $\sum_{i=1}^{n} q_i^* = 1$. The desired probability distribution is then $\{q_i^*\}$. It turns out that we can find an explicit solution for μ^*, using which we can solve Equation 3.4 by an algorithm that takes $O(n \log n)$ time and $O(n)$ storage space. This algorithm first sorts the original probabilities $\{p_i\}$ to get $\{\hat{p}_i\}$ such that $\{\hat{p}_1\}$ is the largest and $\{\hat{p}_n\}$ the smallest probability. Call the transformed (sorted) probability distribution $\{\hat{q}_i^*\}$. Then the algorithm solves for μ^* such that $F(\mu^*) = 0$ where:

$$F(\mu) = \sum_{i=1}^{n} \hat{q}_i^* - 1 = \sum_{i=1}^{k_\mu} \frac{\hat{p}_i}{\mu} + (n - k_\mu) Q - 1 \tag{3.5}$$

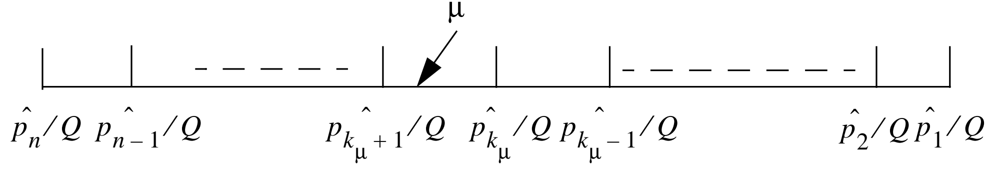

Figure 3.4 Showing the position of μ and k_μ.

Here, k_μ is the number of letters with probability greater than $(\mu \cdot Q)$, and the second equality follows from Equation 3.4. Figure 3.4 shows the relationship between μ and k_μ. For all letters to the left of μ in Figure 3.4, $\hat{q}_i^* = Q$ and for others, $\hat{q}_i^* = \hat{p}_i/\mu$.

Lemma 3.3 $F(\mu)$ is a monotonically decreasing function of μ.

Proof: First, it is easy to see that if μ increases in the interval $\left[\hat{p}_{r+1}/Q, \hat{p}_r/Q\right)$, i.e., such that k_μ does not change, $F(\mu)$ decreases monotonically. Similarly, if μ increases from $\hat{p}_r/Q - \varepsilon$ to $\hat{p}_r/Q + \varepsilon$ so that k_μ decreases by 1, it is easy to verify that $F(\mu)$ decreases.

The algorithm uses Lemma 3.3 to do a binary search (in $O(\log n)$ time) for finding the half-closed interval that contains μ, i.e., a suitable value of r such that $\mu \in \left[\hat{p}_r/Q, \hat{p}_{r-1}/Q\right)$ and $F(\hat{p}_r/Q) \geq 0$ and $F(\hat{p}_{r-1}/Q) < 0$.[1] The algorithm then knows the exact value of $k_\mu = K$ and can directly solve for μ^* using Equation 3.5 to get an explicit formula to calculate $\mu^* = \left(\sum_{i=1}^{K} \hat{p}_i\right) / (1 - (n-K)Q)$. Putting this value of μ^* in Equation 3.4 then gives the transformed set of probabilities $\{\hat{q}_i^*\}$. Given such $\{\hat{q}_i\}$, the algorithm then constructs a canonical alphabetic coding tree as in [114] with the codeword lengths l_k^* as chosen in Equation 3.1. This tree clearly has a maximum depth of no more than D, and its average weighted depth is worse than the optimal algorithm by no more

1. Note that $O(n)$ time is spent by the algorithm in the calculation of $\sum_{i=1}^{k_\mu} \hat{p}_i$ anyway, so a simple linear search can be implemented to find the interval $\left[\hat{p}_r/Q, \hat{p}_{r-1}/Q\right)$.

than 2 bits. To see this, let us refer to the codeword lengths in the optimal tree as $\{l_k^{opt}\}$. Then $C_{opt} = \sum_k p_k l_k = H(p) + D(p \| 2^{-l_k^{opt}})$. As q^* has been chosen to be such that $D(p \| q^*) \le D(p \| q)$ for all probability distributions q in the set $\{q_i : q_i \ge Q\}$, it follows from Equation 3.2 that $C_{mindpq} \le H(p) + D(p \| q^*) + 2 \le C_{opt} + 2$. This proves the following main theorem of this chapter:

Theorem 3.1 Given a set of n probabilities $\{p_i\}$ in a specified order, an alphabetic tree with a depth-constraint D can be constructed in $O(n \log n)$ time and $O(n)$ space such that the average codeword length is at most 2 bits more than that of the optimal depth-constrained alphabetic tree. Further, if the probabilities are given in sorted order, such a tree can be constructed in linear time.

4 Depth-constrained weight balanced tree (DCWBT)

This section presents a heuristic algorithm to generate near-optimal depth-constrained alphabetic trees. This heuristic is similar to the weight balancing heuristic proposed by Horibe [35] with the modification that the maximum depth-constraint is never violated. The trees generated by this heuristic algorithm have been observed to have even lower average weighted depth than those generated by algorithm MINDPQ. Also, the implementation of this algorithm turns out to be even simpler. Despite its simplicity, it is unfortunately hard to prove optimality properties of this algorithm.

We proceed to describe the normal weight balancing heuristic of Horibe, and then describe the modification needed to incorporate the constraint of maximum depth. First, we need some terminology. In a tree, suppose the leaves of a particular subtree correspond to letters numbered r through t — we say that the weight of the subtree is $\sum_{i=r}^{t} p_i$. The root node of this subtree is said to represent the probabilities $\{p_r, p_{r+1}, \ldots, p_t\}$; denoted by $\{p_i\}_{i=r}^{t}$. Thus, the root node of an alphabetic tree has weight 1 and represents the probability distribution $\{p_i\}_{i=1}^{n}$.

In the normal weight balancing heuristic of Horibe [35], one constructs a tree such that the weight of the root node is split into two parts representing the weights of its two children in the most balanced manner possible. The weights of the two children nodes are then split recursively in a similar manner. In general, at an internal node representing the probabilities $\{p_r \ldots p_t\}$, the left and right children are taken as representing the probabilities $\{p_r \ldots p_s\}$ and $\{p_{s+1} \ldots p_t\}$, $r \le s < t$, if s is such that

$$\Delta(r, t) = \left| \sum_{i=r}^{s} p_i - \sum_{i=s+1}^{t} p_i \right| = min_{\forall u\,(r \le u < t)} \left| \sum_{i=r}^{u} p_i - \sum_{i=u+1}^{t} p_i \right|$$

This ‘top-down’ algorithm clearly produces an alphabetic tree. As an example, the weight-balanced tree corresponding to Figure 3.1 is the tree shown in Figure 3.2. Horibe [35] proves that the average depth of such a weight-balanced tree is greater than the entropy of the underlying probability distribution $\{p_i\}$ by no more than $2 - (n+2)p_{min}$, where p_{min} is the minimum probability in the distribution.

Again this simple weight balancing heuristic can produce a tree of unbounded maximum depth. For instance, a distribution $\{p_i\}$ such that $p_n = 2^{-(n-1)}$ and $p_i = 2^{-i}$ $\forall 1 \le i \le n-1$, will produce a highly skewed tree of maximum depth $n-1$. Figure 3.2 is an instance of a highly skewed tree on such a distribution. We now propose a simple modification to account for the depth constraint. The modified algorithm follows Horibe's weight balancing heuristic, constructing the tree in the normal top-down weight balancing manner until it reaches a node such that if the algorithm were to split the weight of the node further in the most balanced manner, the depth-constraint would be violated. Instead, the algorithm splits the node maintaining as much balance as it can while respecting the depth-constraint. In other words, if this node is at depth d representing the proba-

bilities $\{p_r \dots p_t\}$, the algorithm takes the left and right children as representing the probabilities $\{p_r \dots p_s\}$ and $\{p_{s+1} \dots p_t\}$, $a \le s < b$, if s is such that

$$\Delta(r, t) = \left|\sum_{i=r}^{s} p_i - \sum_{i=s+1}^{t} p_i\right| = min_{\forall u\,(a \le u < b)} \left|\sum_{i=r}^{u} p_i - \sum_{i=u+1}^{t} p_i\right|,$$

and $a = t - 2^{D-d-1}$ and $b = r + 2^{D-d-1}$. Therefore, the idea is to use the weight balancing heuristic as far down into the tree as possible. This implies that any node where the modified algorithm is unable to use the original heuristic would be deep down in the tree. Hence, the total weight of this node would be small enough so that approximating the weight balancing heuristic does not cause any substantial effect to the average path length. For instance, Figure 3.5 shows the depth-constrained weight balanced tree for a maximum depth-constraint of 4 for the tree in Figure 3.1.

As mentioned above, we have been unable to come up with a provably good bound on the distance of this heuristic from the optimal solution, but its conceptual and implementational simplicity along with the experimental results (see next section) suggest its usefulness.

Lemma 3.4 A depth-constrained weight balanced tree (DCWBT) for n leaves can be constructed in $O(n\log n)$ time and $O(n)$ space.

Proof: At an internal node, the signed difference in the weights between its two subtrees is a monotonically increasing function of the difference in the number of nodes in the left and right subtrees. Thus a suitable split may be found by binary search in $O(\log n)$ time at every internal node.[1] Since there are $n-1$ internal nodes in a binary tree with n leaves, the total time complexity is $O(n\log n)$. The space complexity is the complexity of storing the binary tree and is thus linear.

1. Note that we may need access to $\sum_{i=r}^{s} p_i, \forall 1 \le r, s \le n$. This can be obtained by precomputing $\sum_{i=1}^{s} p_i, \forall 1 \le s \le n$ in linear time and space.

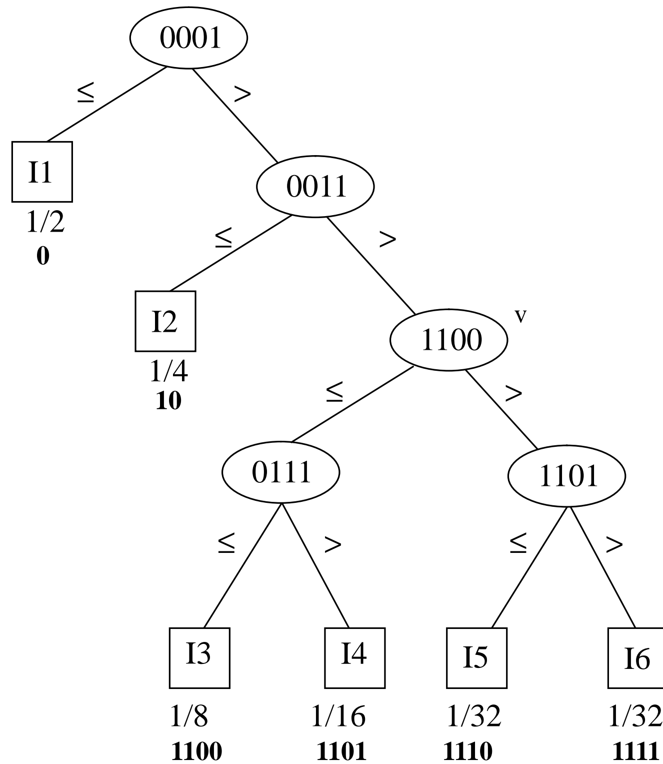

Figure 3.5 Weight balanced tree for Figure 3.1 with a depth-constraint of 4. The DCWBT heuristic is applied in this example at node v (labeled 1100).

5 Load balancing

Both of the algorithms MINDPQ and DCWBT produce a binary search tree that is fairly weight-balanced. This implies that such a tree data structure can be efficiently parallelized. For instance, if two separate lookup engines for traversing a binary tree were available, one engine can be assigned to the left-subtree of the root node and the second to the right-subtree. Since the work load is expected to be balanced among the two engines, we can get twice the average lookup rate that is possible with one engine. This 'near-perfect load-balancing' helps achieve speedup linear in the number of lookup engines, a feature attractive in parallelizable designs. The scalability property can be extended — for instance, the average lookup rate could be made 8 times higher by having 8 subtrees, each being traversed by a separate lookup engine running at 1/8th the aggregate lookup rate

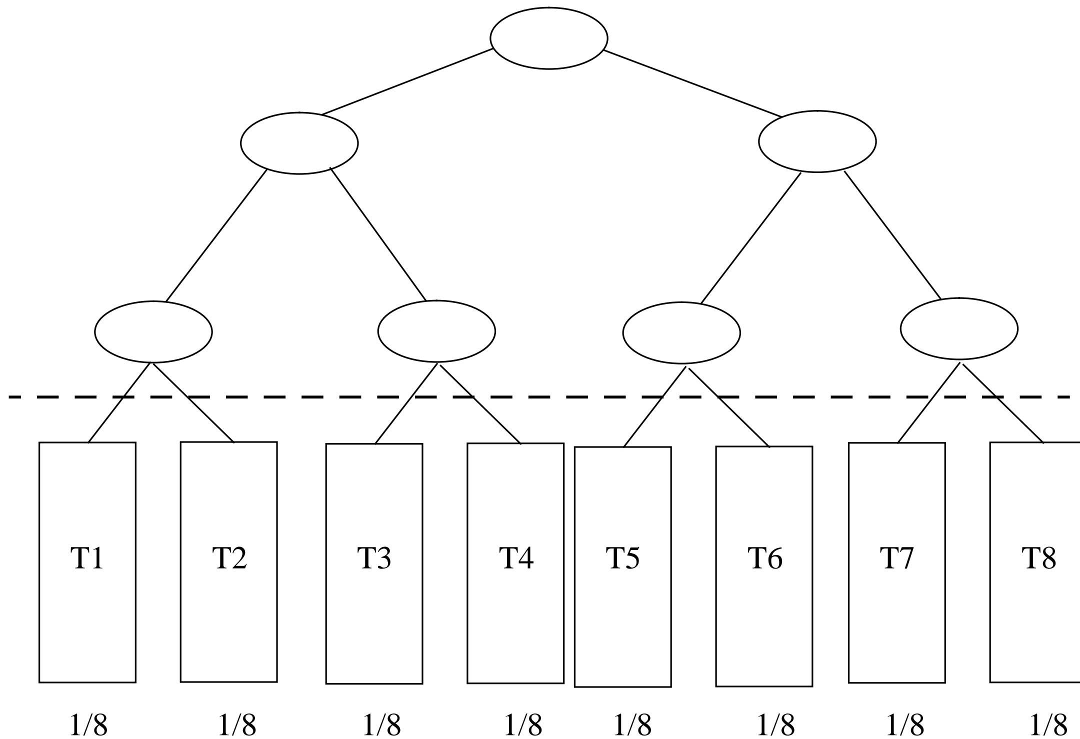

Figure 3.6 Showing 8-way parallelism achievable in an alphabetic tree constructed using algorithm MINDPQ or DCWBT.

(see Figure 3.6). It is to be remembered, however, that only the *average* lookup rate is balanced among the different engines, and hence, a buffer is required to absorb short-term bursts to one particular engine in such a parallel architecture.

6 Experimental results

A plot at CAIDA [12] shows that over 80% of the traffic is destined to less than 10% of the autonomous systems — hence, the amount of traffic is very non-uniformly distributed over prefixes. This provides some real-life evidence of the possible benefits to be gained by optimizing the routing table lookup data structure based on the access frequency of the table entries. To demonstrate this claim, we performed experiments using two large default-free routing tables that are publicly available at IPMA [124], and another smaller table available at VBNS [118].

A knowledge of the access probabilities of the routing table entries is crucial to make an accurate evaluation of the advantages of the optimization algorithms proposed in this chapter. However, there are no publicly available packet traffic traces with non-encrypted destination addresses that access these tables. Fortunately, we were able to find one trace of about 2.14 million packet destination addresses at NLANR [134]. This trace has been taken from a different network location (*Fix-West)* and thus does not access the same routing tables as obtained from IPMA. Still, as the default-free routing tables should not be too different from each other, the use of this trace should give us valuable insights into the advantages of the proposed algorithms. In addition, we also consider the 'uniform' distribution in our experiments, where the probability of accessing a particular prefix is proportional to the size of its interval, i.e., an 8-bit long prefix has a probability of access twice that of a 9-bit long prefix.

Table 3.2 shows the sizes of the three routing tables considered in our experiments, along with the entropy values of the uniform probability distribution and the probability distribution obtained from the trace. Also shown is the number of memory accesses required in an unoptimized binary search (denoted as "Unopt_srch"), which simply is $\lceil \log(\#\text{Intervals}) \rceil$.

TABLE 3.2. Routing tables considered in experiments. Unopt_srch is the number of memory accesses required in a naive, unoptimized binary search tree.

Routing table	Number of prefixes	Number of intervals	Entropy (uniform)	Entropy (trace)	Unopt_srch
VBNS [118]	1307	2243	4.41	6.63	12
MAE_WEST [124]	24681	39277	6.61	7.89	16
MAE_EAST [124]	43435	65330	6.89	8.02	16

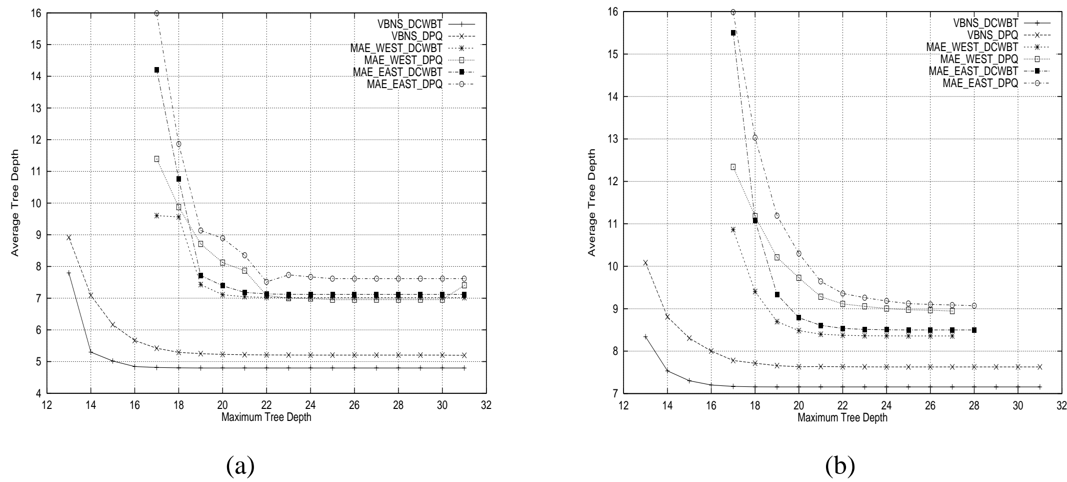

Figure 3.7 Showing how the average lookup time decreases when the worst-case depth-constraint is relaxed: (a) for the “uniform” probability distribution, (b) for the probability distribution derived by the 2.14 million packet trace available from NLANR. X_Y in the legend means that the plot relates to algorithm Y when applied to routing table X.

Figure 3.7 plots the average lookup query time (measured in terms of the number of memory accesses) versus the maximum depth-constraint value for the two different probability distributions. These figures show that as the maximum depth-constraint is relaxed from $\lceil \log m \rceil$ to higher values, the average lookup time falls quickly, and approaches the entropy of the corresponding distribution (see Table 3.2). An interesting observation from the plots (that we have not been able to explain) is that the simple weight-balancing heuristic DCWBT almost always performs better than the near-optimal MINDPQ algorithm, especially at higher values of maximum depth-constraint.

6.1 Tree reconfigurability

Because routing tables and prefix access patterns are not static, the data-structure build time is an important consideration. This is the amount of time required to compute the optimized tree data structure. Our experiments show that even for the bigger routing table at MAE_EAST, the MINDPQ algorithm takes about 0.96 seconds to compute a new tree, while the DCWBT algorithm takes about 0.40 seconds.[1] The build times for the smaller

VBNS routing table are only 0.033 and 0.011 seconds for the MINDPQ and DCWBT algorithms respectively.

Computation of a new tree could be needed because of two reasons: (1) change in the routing table, or (2) change in the access pattern of the routing table entries. As mentioned in Chapter 2, the average frequency of routing updates in the Internet today is of the order of a few updates per second, even though the peak value can be up to a few hundred updates per second. Changes in the routing table structure can be managed by batching several updates to the routing table and running the tree computation algorithm periodically. The change in access patterns is harder to predict, but there is no reason to believe that it should happen at a very high rate. Indeed, if it does, there is no benefit to optimizing the tree anyway. In practice, we expect that the long term access pattern will not change a lot, while a small change in the probability distribution is expected over shorter time scales. Hence, an obvious way for updating the tree would be to keep track of the current average lookup time as measured by the last few packet lookups in the router, and do a new tree computation whenever this differs from the tree's average weighted depth (which is the expected value of the average lookup time if the packets were obeying the probability distribution) by more than some configurable threshold amount. The tree could also be recomputed at fixed intervals regardless of the changes.

To investigate tree reconfigurability in more detail, the packet trace was simulated with the MAE_EAST routing table. For simplicity, we divided the 2.14 million packet destination addresses in the trace into groups, each group consisting of 0.5M packets. The addresses were fed one at a time to the simulation and the effects of updating the tree simulated after seeing the last packet in every group. The assumed initial condition was the 'equal' distribution, i.e., every tree leaf, which corresponds to a prefix interval, is equally

---

1. These experiments were carried out by implementing the algorithms in C and running as a user-level process under Linux on a 333 MHz Pentium-II processor with 96 Mbytes of memory.

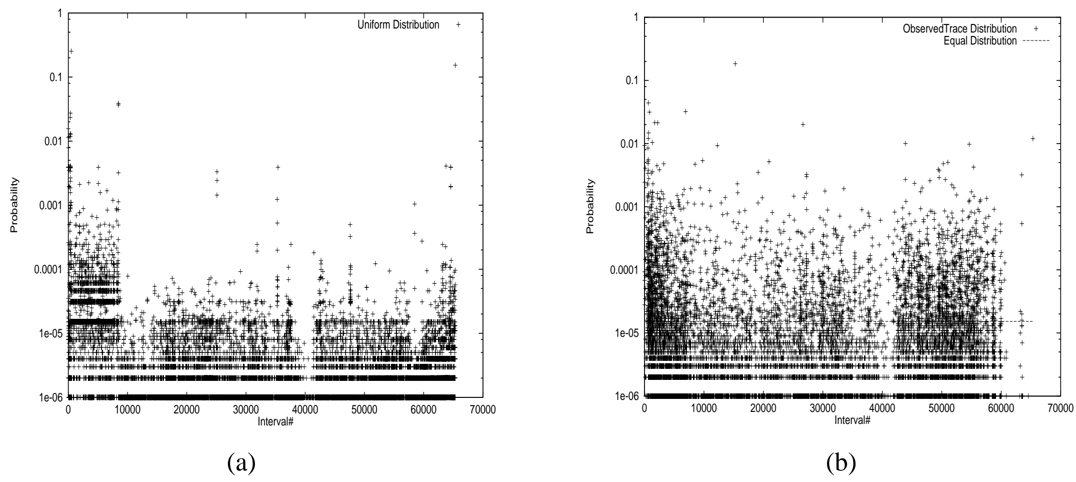

Figure 3.8 Showing the probability distribution on the MAE_EAST routing table: (a) “Uniform” probability distribution, i.e., the probability of accessing an interval is proportional to its length, (b) As derived from the packet trace. Note that the “Equal” Distribution corresponds to a horizontal line at y=1.5e-5.

likely to be accessed by an incoming packet. Thus the initial tree is simply the complete tree of depth $\lceil \log m \rceil$. The tree statistics for the MINDPQ trees computed for every group are shown in Table 3.3 for (an arbitrarily chosen) maximum lookup time constraint of 22 memory accesses.

TABLE 3.3. Statistics for the MINDPQ tree constructed at the end of every 0.5 million packets in the 2.14 million packet trace for the MAE_EAST routing table. All times/lengths are specified in terms of the number of memory accesses to reach the leaf of the tree storing the interval. The worst-case lookup time is denoted by luWorst, the average look up time by luAvg, the standard deviation by luSd. and the average weighted depth of the tree by WtDepth.

PktNum	luWorst	luAvg	luSd	Entropy	WtDepth
0-0.5M	16	15.94	0.54	15.99	15.99
0.5-1.0M	22	9.26	4.09	7.88	9.07
1.0-1.5M	22	9.24	4.11	7.88	9.11
1.5-2.0M	22	9.55	4.29	7.89	9.37
2.0-2.14M	22	9.38	4.14	7.92	9.31

The table shows how computing a new tree at the end of the first group brings down the average lookup time from 15.94 to 9.26 memory accesses providing an improvement

in the lookup rate by a factor of 1.72. This improvement is expected to be greater if the depth-constraint were to be relaxed further. The statistics show that once the first tree update (at the end of the last packet of the first group) is done, the average lookup time decreases significantly and the other subsequent tree updates do not considerably alter this lookup time. In other words, the access pattern changes only slightly across groups. Figure 3.8(b) shows the probability distribution derived from the trace, and also plots the 'equal' distribution (which is just a straight line parallel to the x-axis). Also shown for comparison is the 'uniform' distribution in Figure 3.8(a). Experimental results showed that the distribution derived from the trace was relatively unchanging from one group to another, and therefore only one of the groups is shown in Figure 3.8(b).

7 Related work

Early attempts at using statistical properties comprised caching recently seen destination addresses and their lookup results (discussed in Chapter 2). The algorithms considered in this chapter adapt the lookup data structure based on statistical properties of the forwarding table itself, i.e., the frequency with which each forwarding table entry has been accessed in the past. Intuitively, we expect that these algorithms should perform better than caching recently looked up addresses because of two reasons. First, the statistical properties of accesses on a forwarding table are relatively more static (as we saw in Section 6.1) because these properties relate to prefixes that are aggregates of destination addresses, rather than the addresses themselves. Second, caching provides only two discrete levels of performance (good or bad) for all packets depending on whether they take the slow or the fast path. Hence, caching performs poorly when only a few packets take the fast path. In contrast, an algorithm that adapts the lookup data structure itself provides a more continuous level of performance for incoming packets, from the fastest to the slowest, and hence can provide a higher average lookup rate.

Since most previous work on routing lookups has focussed on minimizing worst-case lookup time, the only paper with a similar formulation as ours is by Cheung and McCanne [10]. Their paper also considers the frequency with which a certain prefix is accessed to improve the average time taken to lookup an address. However, reference [10] uses a trie data structure answering the question of "how to redraw a trie to minimize the average lookup time under a given storage space constraint," while the algorithms described here use a binary search tree data structure based on the binary search algorithm [49] discussed in Section 2.2.6 of Chapter 2. Thus, the methods and the constraints imposed in this chapter are different. For example, redrawing a trie typically entails compressing it by increasing the degree of some of its internal nodes. As seen in Chapter 2, this can alter its space consumption. In contrast, it is possible to redraw a binary search tree without changing the amount of space consumed by it, and hence space consumption is not a constraint in this chapter's formulation.

While it is not possible to make a direct comparison with [10] because of the different nature of the problems being solved, we can make a comparison of the complexity of computation of the data structures. The complexity of the algorithm in [10] is stated to be $O(DnB)$ where B is a constant around 10, and $D = 32$, which makes it about $320n$. In contrast, both the MINDPQ and the DCWBT algorithms are of complexity $O(n \log n)$ for n prefixes, which, strictly speaking, is worse than $O(n)$. However, including the constants in calculations, these algorithms have complexity $O(Cn \log n)$, where the constant factor C is no more than 3. Thus even for very large values of n, say $2^{17} = 128K$, the complexity of these algorithms is no more than $96n$.

8 Conclusions and summary of contributions

This chapter motivates and defines a new problem — that of minimizing the average routing lookup time while still keeping the worst case lookup time bounded — and pro-

poses two near-optimal algorithms for this problem using a binary search tree data structure. This chapter explores the complexity, performance and optimality properties of the algorithms. Experiments performed on data taken from routing tables in the backbone of the Internet show that the algorithms provide a performance gain up to a factor of about 1.7. Higher lookup rates can be achieved with low overhead by parallelizing the data structure using its "near-perfect" load balancing property.

Finding good depth-constrained alphabetic and Huffman trees are problems of independent interest, e.g., in computationally efficient compression and coding. The general approach of this chapter, although developed for alphabetic trees for the application of routing lookups, turns out to be equally applicable for solving related problems of independent interest in Information theory — such as finding depth-constrained Huffman trees, and compares favorably to recent work on this topic (for example, Mildiu and Laber [59][60][61] and Schieber [85]). Since this extension is tangential to the subject of this chapter, it is not discussed here.