Near-Optimal Routing Lookups with Bounded Worst Case Performance

Pankaj Gupta *Balaji Prabhakar* *Stephen Boyd*
Departments of Electrical Engineering and Computer Science
Stanford University, CA 94305.
pankaj@stanford.edu, balaji@isl.stanford.edu, boyd@stanford.edu

Abstract—**The problem of route address lookup has received much attention recently and several algorithms and data structures for performing address lookups at high speeds have been proposed. In this paper we consider one such data structure – a binary search tree built on the intervals created by the routing table prefixes. We wish to exploit the difference in the probabilities with which the various leaves of the tree (where the intervals are stored) are accessed by incoming packets in order to speedup the lookup process. More precisely, we seek an answer to the question "How can the search tree be drawn so as to minimize the average packet lookup time while keeping the worst-case lookup time within a fixed bound?" We use ideas from information theory to derive efficient algorithms for computing near-optimal routing lookup trees. Finally, we consider the practicality of our algorithms through analysis and simulation.**

Keywords—**Routing lookups, Prefix matching, Optimality, Fast routing.**

I. Introduction

The explosive growth of the Internet has placed huge demands on its infrastructure. While advances in optical technologies such as DWDM have increased link speeds to beyond tens of gigabits per second, Internet backbone routers – the processing nodes interconnecting these links, have been lagging behind. One main reason for this is the relatively complex packet processing required at a router — for every incoming packet, the router has to lookup a forwarding table (also called a routing table) to determine the packet's next hop destination. This process of looking up a packet's next hop is aggravated with a steady increase in routing table sizes [1]. As a result, the routing lookup problem has received considerable attention, both in academia and in the industry.

The adoption of classless inter-domain routing (CIDR) [2] in 1993 means that a router now performs a "longest prefix match" to determine the next hop of a packet. A router maintains a set of destination address prefixes in a routing table. Given a packet, the lookup operation consists of finding the longest prefix in the routing table that matches the first few bits of the destination address of the packet. Several solutions including innovative data structures and algorithms for solving this problem have appeared in the recent literature (see, for example, [3], [4], [5], [6], [7], [8]). The early work [4], [5], [6], [7], [8] focused on the development of data structures and algorithms for minimizing the lookup time given a routing table and some memory space constraints. For example, the work of Srinivasan and Varghese [8] aims to minimize the worst-case lookup time of a packet given memory space constraints. Also given space constraints, a more recent paper by Cheung and McCanne [3] considers the frequency with which a certain prefix is accessed to improve the average time taken to look up an address. Both these papers [3], [8] consider a trie data structure (or its variant) while doing the minimization.

This paper also assumes that the frequency with which prefixes are accessed is known. Given this, the aim is to design a routing lookup scheme for minimizing the average lookup time. In this respect our formulation is similar to that in [3]. However, since our data structure is different (we use a binary search tree instead of a radix trie), our methods and the constraints imposed upon us are different. For example, redrawing a trie typically entails compressing it by increasing the degree of some of its internal nodes. 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 our formulation.

But the use of a binary tree data structure brings up a different problem: Observe that it is now possible for the worst-case depth of the binary tree to be very large depending on the distribution of the access probabilities[1]. This can lead to prohibitively long lookup times for some prefixes. It is therefore important to constrain the maximum depth of the binary tree to some small prespecified number.

We approach the problem of finding good depth-constrained binary search trees using information theoretic ideas and convex optimization techniques. To this end, we set up the problem as an average lookup time minimization problem subject to a maximum lookup time constraint. There exists an algorithm to solve this problem due to Larmore and Przytycka [10] with a preprocessing time complexity of $O(nD \log n)$, where n is the number of prefixes and D is the worst case number of memory accesses allowed. Despite its optimality the algorithm is complicated and difficult to implement. The algorithm obtained in this paper is nearly optimal (to within two memory accesses) and has a preprocessing time complexity of $O(n \log n)$. More importantly, it is easy to implement.

A useful by-product of our approach is that the resulting data structure is easily parallelizable. That is, if several processing engines were available to carry out the lookup operation, then each engine will be sharing the total load almost equally. This load balancing feature when used in parallel hardware designs reduces the average lookup time by a multiplicative factor.

In this paper the routing lookup problem motivates us to find good depth-constrained binary search trees (also called alphabetic trees). Finding good depth-constrained alphabetic and Huffman trees are problems of independent interest, e.g. in computationally efficient compression. The general approach of this paper, although developed for alphabetic trees, turns out to be

[1] There are about 65000 prefixes in a large routing table today [9]. This can lead to a binary search tree with a maximum depth of several hundreds or thousands.

	Prefix	*StartPoint*	*EndPoint*
P_1	*	0000	1111
P_2	00*	0000	0011
P_3	1*	1000	1111
P_4	1101	1101	1101
P_5	001*	0010	0011

TABLE I

An example routing table with 4-bit prefixes and endpoints of their induced intervals on the number line $[0000, 1111]$.

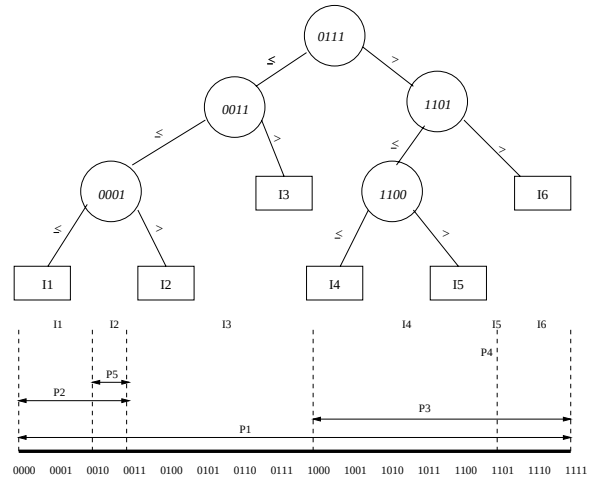

Fig. 1. *The binary search tree corresponding to the routing table in Table I.*

equally applicable for finding depth-constrained Huffman trees, and compares favorably to recent work on this topic (see, for example, Schieber [11] and Mildiu and Laber [12]).

The paper is organized as follows: Section 2 describes the problem of routing lookups and its relationship with Information Theory. Section 3 sets up the optimization problem and describes our proposed solution. Section 4 provides simulation results of the proposed algorithms on some large publicly available routing tables and a large packet trace. And Section 5 concludes.

II. Routing Lookups

A routing table consists of a series of tuples of the form ⟨*prefix, nextHop*⟩, where *prefix* represents the aggregation of several 32-bit destination IP addresses and *nextHop* is the IP address of the corresponding next hop router. Given an incoming packet's destination address, the routing lookup problem is to find the longest (i.e. the most specific) of all the prefixes matching the first few bits of the incoming packet's destination address.

Each prefix can be viewed as an interval on the number line $[0, 2^{32})$, referred to here as the *IP number line*. Lampson, Srinivasan and Varghese [6] propose a data structure in which the end points of this interval, sorted in increasing order after eliminating duplicates to give $\{P_1, P_2, \ldots, P_m\}$, are stored in the external nodes (leaves) of a binary tree and the corresponding next hop addresses are precomputed for each of the disjoint basic intervals, $[P_i, P_{i+1})$. The internal nodes contain suitably chosen values to guide the search process to one of their two children. An example of a routing table with 4-bit prefixes is shown in Table I. The corresponding subdivision of the IP number line and an example binary search tree is shown in Figure 1. Clearly, with n prefixes, there are no more than $m = 2n$ such endpoints and thus the resulting binary search tree has a size which is linear in the number of routing table prefixes.

While this binary tree structure is very good if all intervals are accessed uniformly, the search time can be improved considerably by making use of the frequency with which a certain routing table entry is accessed. We note that today's routers already maintain such per-prefix statistics. Hence, minimizing routing lookup times by making use of this information comes at no extra data collection cost. Given this, a natural question to ask is "What is the best tree data structure given the frequency of access of prefix intervals?" Viewing it this way, the problem is readily recognized to be one of minimizing the average weighted length of a binary tree whose leaves are weighted by the probabilities associated with the intervals represented by the leaves. The minimization is to be carried over all possible binary trees with the given intervals and corresponding weights at the leaves.

In the language of Information Theory, the problem stated above translates to: Find a *minimum average length alphabetic code (or tree)* for an m-letter alphabet (where each letter corresponds to an interval). An alphabetic code is one in which the m letters are ordered lexicographically on the leaves of the resulting binary tree. That is, if the letter A appears before the letter B in the alphabet then the code word associated with the letter A has a smaller binary value than the code word associated with the letter B. For the example in Figure 1, the code word associated with Interval I1 is 000 and that associated with Interval I5 is 101, where a bit in the codeword is 0 (resp. 1) for the left (resp. right) branch at the corresponding node.

As an example, if the intervals *I1* through *I6* are accessed with probabilities 1/2, 1/4, 1/8, 1/16, 1/32 and 1/32 respectively, then the optimal alphabetic tree corresponding to these probabilities (or weights) is shown in Figure 2(a). The codeword for I1 is now 0 and that of I5 is 11110.

III. Algorithms

The average length of a general prefix code for a given set of probabilities can be minimized using the well-known Huffman coding algorithm [13]. However, a Huffman solution is not guaranteed to 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 search. 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 itself, it is not feasible to use a Huffman solution.

Apart from the alphabetic constraint, it is necessary to bound the maximum code word length (or the maximum depth of the tree) to make the solution useful in practice. This is because an optimal alphabetic tree for n letters can have a maximum depth of $n - 1$, see for instance, Figure 2(a) (the root is assumed to

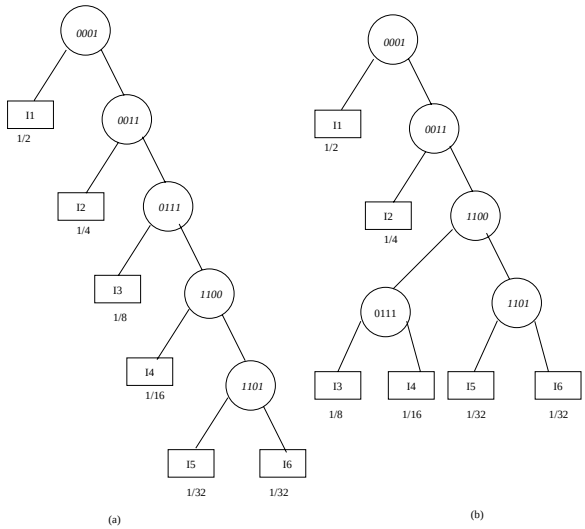

Fig. 2. *Optimal alphabetic tree corresponding to the tree in Figure 1 with leaf probabilities as shown: (a) unconstrained depth, (b) depth constrained to 4.*

be at depth 0). This is unacceptable in practice where the typical value of n is around 65000, since the access of a deep leaf can cause the router to slow down considerably. Further, 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 very desirable to have a small upper bound on the maximum depth of the alphabetic tree. An upper bound on the worst case lookup time also simplifies the hardware design of a router. Thus, well-known algorithms for finding an optimal alphabetic tree such as those in [14], [15], [16] which do not incorporate a maximum depth constraint cannot be used in our setting.

To understand this last point better, consider the alphabetic tree of Figure 2(a) which is optimal if the intervals *I1* through *I6* shown in the binary tree of Figure 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 (equal to $1 \cdot 1/2 + 2 \cdot 1/4 + 3 \cdot 1/8 + 4 \cdot 1/16 + 5 \cdot 1/32 + 5 \cdot 1/32$), while the maximum depth is 5. If we impose a maximum depth constraint of 4, then we need to redraw the tree and obtain another tree as shown in Figure 2(b) where the average lookup time has increased to 2.

In general, we are interested in the following minimization problem:

$$\underset{\{l_i\}}{\text{minimize}} \quad C = \sum_{i=1}^{i=n} l_i \cdot p_i \quad \text{s.t. } l_i \leq D \ \forall i \qquad (1)$$

for n intervals with access probabilities p_i, l_i being the number of comparisons required to lookup a packet in the ith interval. From the previous discussion, we also require that the l_i's obtained as a result of solving the above minimization problem give rise to an alphabetic tree. Yeung [17] gives a necessary and sufficient condition that the l_i's must satisfy in order for the resulting tree to be alphabetic. This condition is stated below as Lemma 1.

The smallest possible value of C is the entropy, $H(p)$, of the set of probabilities $\{p_i\}$, where $H(p) = -\sum p_i \log p_i$.[2] And for depth constrained alphabetic trees C will be bigger than $H(p)$ more often than not[3]. 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 [18], [12], [19], almost 40 years after the original Huffman algorithm. Imposing the alphabetic constraint renders the problem even harder [20], [21], [22], [23]. Larmore and Przytycka [10] have proposed an optimum algorithm which runs in time $O(nD \log n)$ and finds a depth constrained alphabetic tree. However, their algorithm is very complicated to implement.

In light of this, we attempt to find a practical and provably good solution to the problem of computing optimal depth-constrained alphabetic trees. A good approximate solution to the routing lookup problem is of value because: (1) It is much simpler to find than an optimum solution. It is also much simpler to implement. (2) As the probabilities associated with the intervals induced by routing prefixes change frequently and are not known exactly, it does not seem to make much sense to solve the problem exactly for an optimum solution. (3) Our approximate solution can be theoretically proved to be requiring no more than two extra comparisons per lookup when compared to the optimum solution. In practice, the discrepancy is very often found to be less than two.

A. Algorithm MINDPQ

We first state two results from [17] as lemmas that we will use to develop our algorithm. The first lemma states a necessary and sufficient condition for the existence of an alphabetic code with specified code word lengths, and the second prescribes a method for constructing good, near-optimal trees which are not depth-constrained.

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

Proof: For a complete proof, see [17]. 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 the letter i to be 0010. If $l_{i+1} = 4$, then the codeword for the 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 [17]) 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 paper.)

Lemma 2: The minimum average length, C_{min}, of an alphabetic code on n letters, where the ith letter occurs with probability p_i satisfies: $H(p) \leq C_{min} \leq H(p) + 2 - p_1 - p_n$.

[2] All logarithms in this paper are to the base 2.

[3] The lower bound of entropy is achieved in general when there are no alphabetic or maximum depth constraints.

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:

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

The proof in [17] verifies that these lengths satisfy the characteristic inequality, 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 optimal depth-constrained alphabetic trees. 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 2 could yield a tree where the maximum depth is bigger than D. To work around this problem, we transform the given probabilities $\{p_k\}$ into another set of probabilities $\{q_k\}$ such that $q_{min} = \min_k q_k \geq 2^{-D}$. This allows us to apply the following variant of the scheme in Lemma 2 to obtain a near-optimal depth-constrained alphabetic tree with leaf probabilities $\{q_k\}$.

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

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

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 1 it suffices to show that the $\{l_k^*\}$ satisfy the characteristic inequality. We defer this verification to the Appendix.

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,

$$\begin{aligned} \sum_k p_k l_k^* &\leq \sum_k p_k \log \frac{1}{q_k} + 2 \\ &= \sum_k p_k \log \frac{p_k}{q_k} - \sum_k p_k \log p_k + 2 \\ &= \mathcal{D}(p||q) + H(p) + 2, \end{aligned} \quad (3)$$

where $\mathcal{D}(p||q)$ is the "relative entropy" between the distributions p and q and $H(p)$ is the entropy of the distribution p. In order to minimize $\sum_k p_k l_k^*$, we must therefore choose $\{q_i^*\} = \{q_i\}$ so as to minimize $\mathcal{D}(p||q)$.

A.1 The minimization problem

We are thus led to the following optimization problem:

$$\begin{array}{ll} \text{minimize} & DPQ = \mathcal{D}(p||q) = \sum_i p_i \log(p_i/q_i) \\ \text{subject to} & \sum_i q_i = 1 \quad \text{and} \quad q_i \geq Q = 2^{-D} \ \forall i \end{array} \quad (4)$$

Observe that the cost function $\mathcal{D}(p||q)$ is convex in (p, q) (see Page 30 of Cover and Thomas [24]). Further, the constraint set is convex and compact. In fact, 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 [25]).

Accordingly, define

$$\begin{aligned} \mathcal{L}(q, \overline{\lambda}, \mu) &= \sum p_i \log(p_i/q_i) + \\ &\quad \sum \lambda_i (Q - q_i) + \mu(\sum q_i - 1) \end{aligned}$$

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

$$\frac{\partial \mathcal{L}}{\partial q_i} = 0 \Rightarrow q_i^* = \frac{p_i}{\mu - \lambda_i} \quad (5)$$

Putting this back in $\mathcal{L}(q, \overline{\lambda}, \mu)$, we get the dual

$$\mathcal{G}(\overline{\lambda}, \mu) = \sum_i (p_i \log(\mu - \lambda_i) + \lambda_i Q) + (1 - \mu)$$

Now minimizing $\mathcal{G}(\overline{\lambda}, \mu)$ subject to $\lambda_i \geq 0$ and $\mu > \lambda_i \ \forall i$ gives

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

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

which combined with the constraint that $\lambda_i \geq 0$ gives us $\lambda_i^* = \max(0, \mu - p_i/Q)$. Substituting this in Equation (5), we get

$$q_i^* = \max(p_i/\mu, Q) \quad (6)$$

To finish, we need to solve Equation (6) for $\mu = \mu^*$ under the constraint that $\sum_{i=1}^{i=n} q_i^* = 1$. $\{q_i^*\}$ will then be the desired transformed probability distribution. It turns out that we can find a closed form expression for μ^*, using which we can solve Equation (6) by an $O(n \log n)$ time and $O(n)$ space algorithm. The 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 $\mathcal{F}(\mu^*) = 0$ where

$$\begin{aligned} \mathcal{F}(\mu) &= \sum_{i=1}^{i=n} \hat{q_i^*} - 1 \\ &= \sum_{i=1}^{i=k_\mu} \frac{\hat{p_i}}{\mu} + (n - k_\mu)Q - 1 \end{aligned} \quad (7)$$

where the second equality follows from Equation 6, and k_μ is the number of letters with probability greater than μQ. Figure 3 shows the relationship between μ and k_μ. For all letters to the left of μ in Figure 3, $\hat{q_i^*} = Q$ and for others, $\hat{q_i^*} = \hat{p_i}/\mu$.

Lemma 3: $\mathcal{F}(\mu)$ is a monotonically decreasing function of μ.

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

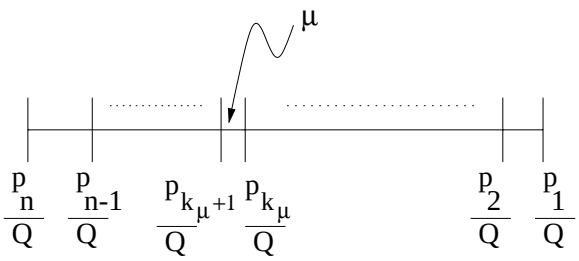

Fig. 3. *Showing the position of μ and k_μ*

Lemma 3 implies that we can do a binary search for finding a suitable value of r such that $\mu \in [p_r/Q, p_{r-1}/Q)$ and $\mathcal{F}(p_r/Q) \geq 0$ and $\mathcal{F}(p_{r-1}/Q) < 0$. This will take us only $O(\log n)$ time. Once we know that μ belongs to this half-closed interval, we know the exact value of $k_\mu = K$ and we can then directly solve for μ^* using Equation (7) to get $\mu^* = (\sum_{i=1}^{i=K} \hat{p_i})/(1-(n-K)Q)$. Putting this value of μ^* in Equation (6) will then give us the transformed set of probabilities $\{\hat{q_i^*}\}$. [4] Given such $\{\hat{q_i}\}$, the algorithm then constructs a canonical alphabetic coding tree as in [17] with the codeword lengths l_k^* as chosen in Equation (2). This tree then clearly has a maximum depth of no more than D, and its average weighted length is worse than the optimum algorithm by no more than 2 bits. To see this, let us refer to the code lengths in the optimum tree as $\{l_k^{opt}\}$. Then $C_{opt} = \sum_k p_k l_k = H(p) + \mathcal{D}(p||2^{-l_k^{opt}})$. As we have chosen q^* to be such that $\mathcal{D}(p||q^*) \leq \mathcal{D}(p||q)$ for all probability distributions q, it follows from Equation (3) that $C_{mindpq} \leq C_{opt} + 2$. We have thus proved the following main theorem of the paper:

Theorem 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 away from the optimum depth-constrained alphabetic tree. Further, if the probabilities are given in sorted order, such a tree can be constructed in linear time.

B. Depth-Constrained Weight Balanced Tree (DCWBT)

In this subsection, we present a heuristic algorithm to generate near-optimal depth constrained alphabetic trees. This heuristic is similar to the weight balancing heuristic proposed by Horibe [26] with the modification that the maximum depth constraint is never violated. The trees generated by this heuristic algorithm have been observed to have lower average weighted length than those generated by algorithm MINDPQ. Also, implementation of this algorithm turns out to be even simpler. Despite its simplicity, it is unfortunately hard to prove any optimality properties of this algorithm.

We proceed to describe the normal weight balancing heuristic and then describe the modification needed to incorporate the constraint of maximum depth. 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 as well as of its root is $\sum_{i=r}^{i=t} p_i$. The root node of this subtree is said to represent the probabilities $\{p_i\}_{i=r}^{i=t}$. Thus, the root node of an alphabetic tree has weight 1 and represents the probability distribution $\{p_i\}_{i=1}^{i=n}$.

In the normal weight balancing heuristic of Horibe [26], 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 $\{p_r \ldots p_t\}$, we take the left and right children as representing the probabilities $\{p_r \ldots p_s\}$ and $\{p_{s+1} \ldots p_t\}$ if s is such that

$$\begin{aligned}\Delta(r,t) &= \left|\sum_{i=r}^{i=s} p_i - \sum_{i=s+1}^{i=t} p_i\right|,\ r \leq s < t \\ &= \min_{\forall u: r \leq u < t} \left|\sum_{i=r}^{i=u} p_i - \sum_{i=u+1}^{i=t} p_i\right|\end{aligned}$$

This "top-down" algorithm clearly produces an alphabetic tree. As an example, the weight-balanced tree corresponding to Figure 1 is the tree shown in Figure 2(a) for the shown probabilities. Horibe 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 which has an unbounded maximum depth. For instance, a distribution $\{p_i\}$ such that $p_n = 2^{-(n-1)}$ and $p_i = 2^{-i}\ \forall 1 \leq i \leq n-1$, will produce a highly skewed tree of maximum depth $(n-1)$. Figure 2(a) is an instance of a tree on such a distribution and so is highly skewed. Here is a simple modification we propose to respect the depth constraint — we follow Horibe's weight balancing heuristic constructing the tree in the normal top-down weight balancing manner until we reach a node such that if we were to split the weight of the node further in the most balanced manner, the depth constraint would be violated. Instead, we split the node maintaining as much balance as we can while respecting the depth constraint. If this happens at a node at depth d representing the probabilities $\{p_r \ldots p_t\}$ we take the left and right children as representing the probabilities $\{p_r \ldots p_s\}$ and $\{p_{s+1} \ldots p_t\}$ if s is such that

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

where $a = t - 2^{D-d-1}$ 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. Intuitively, any node where we are unable to use the heuristic is expected to be very deep down in the tree. This would mean that the total weight of this node is small enough so that approximating the weight balancing heuristic does not cause any substantial effect to the average path length. For instance, Figure 4 shows the depth-constrained weight balanced tree for a maximum depth constraint of 4 for the tree in Figure 1(a).

[4] Note that we will have to spend $O(n)$ time in the calculation of $\sum_{i=1}^{i=k_\mu} \hat{p_i}$ anyway, so if we want we can simply implement a linear search instead of a binary search to find the interval $[p_r/Q, p_{r-1}/Q)$.

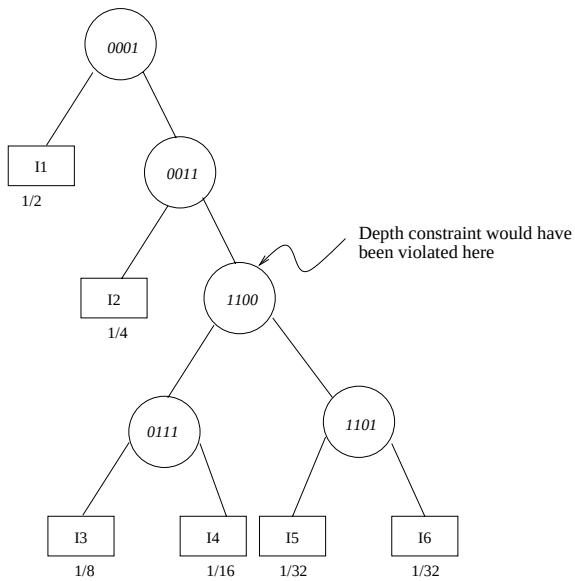

Fig. 4. *Weight balanced tree for Figure 1 with a depth constraint of 4.*

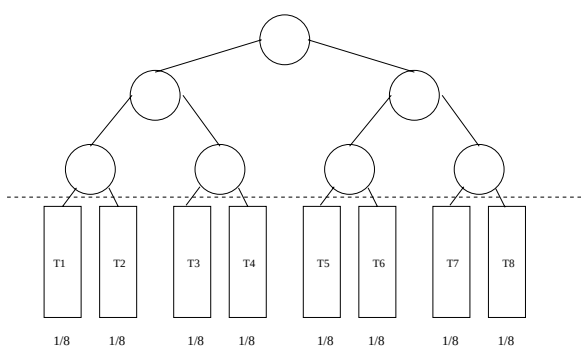

Fig. 5. *Showing 8-way parallelism in the constructed alphabetic tree.*

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

Lemma 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[5]. 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.□

C. Load Balancing

From the manner in which the DCWBT is constructed, it is easy to see that the two children subtrees of any internal node are nearly as weight-balanced as possible. Even in the alphabetic tree constructed by the MINDPQ algorithm, the two subtrees of an internal node are almost equally balanced. This implies that such a tree data structure can be efficiently parallelized. Suppose that we had two separate lookup engines for traversing a binary tree. Then we can assign the left-subtree of the root node to one of these engines and the right-subtree to the other engine. Since, the work load is expected to be balanced among the two engines, we can get twice the lookup rate that is possible with one engine. Such an architecture is attractive in parallelizable designs. This "near-perfect load-balancing" thus helps achieve linear speedup with the number of lookup engines. This is a nice scalability property — for instance, if the routing tables were to increase 8 times in size, the same lookup rate could be achieved by having 8 subtrees, each being traversed by a separate lookup engine, as shown in Figure 5.

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

IV. Simulation Results

A plot at the CAIDA web site [27] shows that the amount of traffic per prefix length is very non-uniformly distributed. 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. For the purpose of detailed simulation to further demonstrate this claim, we took two large default-free routing tables publicly available at IPMA [9] and another smaller table at VBNS [28].

To make an accurate evaluation of the advantages of the optimization algorithms proposed in this paper, a knowledge of the access probabilities of the routing table entries is crucial. However, there are no publicly available packet traffic traces with unencrypted destination addresses that access these tables. Fortunately, we were able to find one trace of about 2.14 million packet destination addresses at NLANR [29] – this trace has been taken from *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 simulations, where the probability of accessing a particular prefix is proportional to its size, i.e. an 8-bit long prefix has a probability of access twice that of a 9-bit long prefix.

Table II shows the sizes of the three routing tables considered in our simulation, 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, which simply is $\lceil \log(\#Intervals) \rceil$.

In Figures 6 and 7, we plot 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 $\lceil \log_2 n \rceil$ to higher values, the average lookup time quickly approaches the entropy of the corresponding distribution (see Table II). An interesting observation from the plots 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.

Routing Table	# Prefixes	# Intervals	Entropy (Uniform)	Entropy (Trace)	Unopt_srch
VBNS [28]	1307	2243	4.41	6.63	12
MAE_WEST [9]	24681	39277	6.61	7.89	16
MAE_EAST [9]	43435	65330	6.89	8.02	16

TABLE II

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

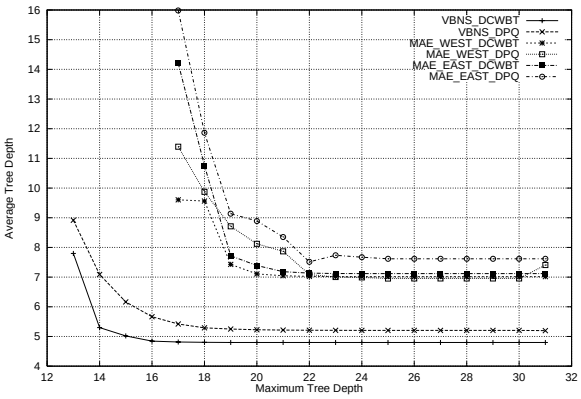

Fig. 6. *Showing how the average lookup time decreases when the worst case depth constraint is relaxed, for the "uniform" probability distribution. X_Y in the legend means that the plot relates to algorithm Y when applied to the routing table X.*

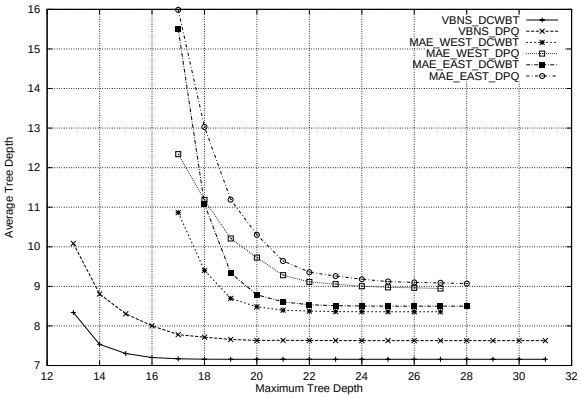

Fig. 7. *Showing how the average lookup time decreases when the worst case depth constraint is relaxed, for the probability distribution derived by the two million packet trace available from NLANR.X_Y in the legend means that the plot relates to algorithm Y when applied to the routing table X.*

A. Tree Reconfigurability

Because the routing tables and the 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 simulations show that even for the biggest 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.[6] The build times

[6]These simulations were carried out by implementing the algorithms in C and running the simulation as a user-level process under Linux on a 333MHz Pentium-II processor with 96MB of memory.

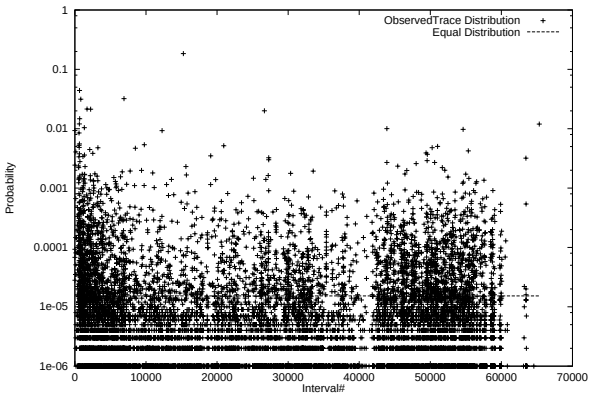

Fig. 8. *Showing the probability distribution as derived from the packet trace on the the MAE_EAST routing table. Note that the "Equal" Distribution corresponds to a horizontal line at y=1.5e-5.*

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. 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. We can simply batch several updates to the routing table and run the tree computation algorithm periodically. Changes in the routing table structure can therefore be easily managed. 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 reality, we expect the global access pattern to 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 length (which is the expected value of the average lookup time if the packets were obeying the probability distribution) by more than some settable threshold amount. The tree could also be recomputed at fixed intervals regardless of the changes. In summary, we believe that the tree build times are small enough to make the algorithms of practical use.

To investigate tree reconfigurability in more detail, we simulated the packet trace with the MAE_EAST routing table. For simplicity, we divided the 2.14 million packet destination ad-

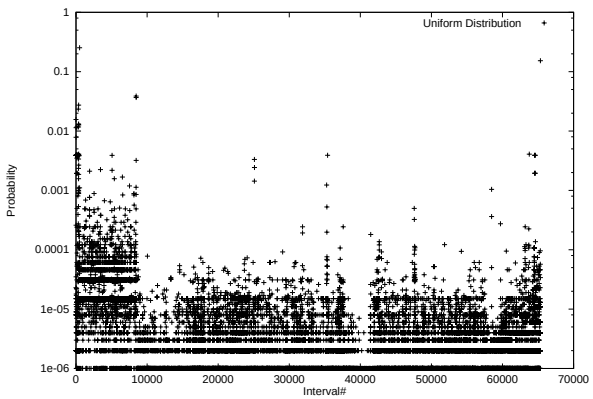

Fig. 9. *Showing the "uniform" probability distribution as derived from the MAE_EAST routing table. In this distribution, the probability of accessing an interval is proportional to its length.*

PktNum	luWst	luAvg	luSd	Entropy	WtLen
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

TABLE III

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. The maximum lookup time is constrained to be 22 memory accesses. All times/lengths are specified in terms of the number of memory accesses to reach the leaf of the tree storing the interval. luWst is the worst case lookup time, luAvg is the average look up time and luSd is the standard deviation. WtLen is the average weighted length of the tree.

dresses in the trace into groups, each group consisting of 0.5M packets. We fed the addresses one at a time to the simulation and simulated the effects of updating the tree after seeing the last packet in every group. We started out with the "equal" distribution, i.e. every tree leaf, which corresponds to a prefix interval, is equally likely to be accessed by an incoming packet. Thus our initial tree is simply the complete tree of depth $\lceil \log_2 n \rceil$. The simulation showed 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 8 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 9. We found the distribution derived from the trace to be relatively unchanging from one group to another, and therefore only one of the groups is shown in Figure 8. This is also borne out in the tree statistics for the MINDPQ trees computed for every group as shown in Table III for (an arbitrarily chosen) maximum lookup time constraint of 22 memory accesses. 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 a nearly 42% improvement in the lookup rate. This improvement is expected to be greater if the depth constraint were to be relaxed further.

While it is not possible to make a direct comparison with the other proposed schemes [3] [8] for optimizing the routing table data structures because of the different nature of problems being solved; we can make a comparison of the complexity of computation of the "optimal" data structures. The complexity of the data structure in [8] is stated to be $O(HnB)$ where $H = 32$, $n =$ number of prefixes, $B =$ constant believed to be around 3 [8]. This is thus around $96n$. The complexity of the algorithm in [3] is stated to be $O(HnB)$ where B is a constant around 10, 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 $Cn \log n$, where C is a constant no more than 3. Thus even for very large values of n, say $2^{16} = 64K$, the complexity of these algorithms is no worse than approximately $48n$.

V. Conclusions

This paper proposed two near-optimal algorithms for doing route lookups using a binary search tree data structure with a constraint on the maximum depth. The complexity, performance and optimality properties of the algorithms were explored. Simulations performed on data taken from routing tables in the backbone of the Internet show that the algorithms provide a significant gain in performance.

References

[1] "BGP Table," http://telstra.net/ops/bgptable.html.

[2] Y. Rekhter and T. Li, "An Architecture for IP Address Allocation with CIDR," RFC 1518, 1993.

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

[4] A. Brodnik, S. Carlsson, M. Degermark, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proceedings of ACM SIGCOMM*, Mar. 1997, pp. 3–13.

[5] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proceedings of INFOCOM*, 1998, pp. 1240–1247.

[6] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups using Multiway and Multicolumn Search," in *Proceedings of INFOCOM*, 1998, pp. 1248–1256.

[7] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," *To appear in IEEE Journal on Selected Areas in Communications*, 1999.

[8] V. Srinivasan and G. Varghese, "Faster IP Lookups using Controlled Prefix Expansion," in *ACM Sigmetrics*, 1998.

[9] "IPMA Project," http://www.merit.edu/ipma.

[10] L. L. Larmore and T. M. Przytycka, "A fast algorithm for optimum height-limited alphabetic binary trees," *SIAM Journal on Computing*, vol. 23, no. 6, pp. 1283–1312, Dec. 1994.

[11] B. Schieber, "Computing a minimum-weight k-link path in graphs with the concave mong e property," in *Proc. 6th ACM IEEE Symposium on Discrete Algorithms*, Jan. 1995, pp. 405–411.

[12] R. L. Milidiu and E. S. Laber, "Warm-up algorithm: A lagrangean construction of length restricted huffman codes," , no. 15, Jan. 1996.

[13] D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. Inst. Radio Engineers*, vol. 40, no. 10, pp. 1098–1101, Sept. 1952.

[14] A. M. Garsia and M. L. Waschs, "A new algorithm for minimal binary search trees," *SIAM Journal on Computing*, vol. 6, pp. 622–642, 1977.

[15] T. C. Hu, *Combinatorial Algorithms*, Addison-Wesley, 1982.

[16] T. C. Hu and A. C. Tucker, "Optimal computer search trees and variable length alphabetic codes," *SIAM Journal on Applied Mathematics*, vol. 21, pp. 514–532, 1971.
[17] R. W. Yeung, "Alphabetic codes revisited," *IEEE Transactions on Information Theory*, vol. 37, no. 3, pp. 564–572, 1991.
[18] R. L. Mildiu and E. S. Laber, "Improved bounds on the inefficiency of length-restricted prefix codes," *unpublished*.
[19] R. L. Milidiu, A. A. Pessoa, and E. S. Laber, "Efficient implementation of the WARM-UP algorithm for the construction of length-restricted prefix codes," in *Proceedings of the ALENEX*, Baltimore, Maryland, USA, January 1999, vol. 1619 of *Lecture Notes in Computer Science*, Springer.
[20] M. R. Garey, "Optimal binary search trees with restricted maximal depth," *SIAM Journal on Computing*, vol. 3, pp. 101–110, 1974.
[21] E. N. Gilbert, "Codes based on inaccurate source probabilities," *IEEE Transactions on Information Theory*, vol. 17, pp. 304–314, 1971.
[22] A. Itai, "Optimal alphabetic trees," *SIAM Journal on Computing*, vol. 5, pp. 9–18, 1976.
[23] R. L. Wessner, "Optimal alphabetic search trees with restricted maximal height," *Information Processing Letters*, vol. 4, pp. 90–94, 1976.
[24] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, Wiley Series in Telecommunications, 1995.
[25] D. P. Bertsekas, *Nonlinear Programming*, Athena Scientific, 1995.
[26] Y. Horibe, "An improved bound for weight-balanced tree," *Information and Control*, vol. 34, pp. 148–151, 1977.
[27] KC Claffy, "Caida Internet Measurement Presentation," http://www.caida.org/Presentations/Soa9905/mgp00026.html.
[28] "All vBNS routes snapshot," http://www.vbns.net/route/Allsnap.rt.html.
[29] "NLANR Network Analysis Infrastructure," http://moat.nlanr.net.

APPENDIX

Lemma 5: For a depth constrained alphabetic tree with maximum depth D, the lengths of n letters when chosen as follows satisfy the characteristic inequality of Lemma 1, i.e. $s_n \leq 1$ where $s_k = c(s_{k-1}, 2^{-l_k}) + 2^{-l_k}$, $s_0 = 0$ and c is defined by $c(a,b) = \lceil a/b \rceil b$; when the code length l_k of the kth letter occurring with probability q_k ($q_k \geq 2^{-D} \ \ \forall k$) is given by:

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

Proof: We first prove by induction that

$$s_i \leq \sum_{k=1}^{i} q_k \quad \forall 1 \leq i \leq n-1$$

For the base case, $s_1 = 2^{-l_1} \leq q_1$ by the definition of l_1. For the induction step, assume the hypothesis is true for $i-1$. By definition, $s_i = 2^{-l_i} + c(s_{i-1}, 2^{-l_i})$. Now there are two possible cases:

1. $\lceil -\log q_i \rceil + 1 \leq D$, and therefore $2^{-(l_i - 1)} \leq q_i$. Using the fact that $\lceil a/b \rceil < a/b + 1$ or $c(a,b) < a + b$ for all nonzero real numbers a and b, we get that $s_i \leq 2^{-l_i} + s_{i-1} + 2^{-l_i}$. Now using the inductive hypothesis, we get

$$s_i \leq 2^{-(l_i - 1)} + \sum_{k=1}^{k=i-1} q_k \leq q_i + \sum_{k=1}^{k=i-1} q_k = \sum_{k=1}^{k=i} q_k$$

2. $\lceil -\log q_i \rceil + 1 > D$. This implies that $l_i = D$ and hence $q_i \geq 2^{-l_i}$. Also, as s_j is an integral multiple of $2^{-D} \ \ \forall j$, $c(s_{i-1}, 2^{-l_i}) = s_{i-1} \leq \sum_{k=1}^{i-1} q_k$ and thus

$$s_i = 2^{-l_i} + c(s_{i-1}, 2^{-l_i}) \leq q_i + \sum_{k=1}^{k=i-1} q_k = \sum_{k=1}^{k=i} q_k$$

Therefore, $s_{n-1} \leq \sum_{i=1}^{i=n-1} q_i = 1 - q_n \leq 1 - 2^{-l_n}$. And $s_n = 2^{-l_n} + c(s_{n-1}, 2^{-l_n}) \leq c(1 - 2^{-l_n}, 2^{-l_n}) + 2^{-l_n} = 1 - 2^{-l_n} + 2^{-l_n} = 1$. This completes the proof that these lengths satisfy the characteristic inequality.□