Algorithms for Variable Length Subnet Address Assignment

Mikhail J. Atallah*(Fellow)
COAST Laboratory and
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
U.S.A.
mja@cs.purdue.edu

Douglas E. Comer
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
U.S.A.
dec@cs.purdue.edu

Abstract

In a computer network that consists of M subnetworks, the L-bit address of a machine consists of two parts: A prefix s_i that contains the address of the subnetwork to which the machine belongs, and a suffix (of length $L - |s_i|$) containing the address of that particular machine within its subnetwork. In fixed-length subnetwork addressing, $|s_i|$ is independent of i, whereas in variable-length subnetwork addressing, $|s_i|$ varies from one subnetwork to another. To avoid ambiguity when decoding addresses, there is a requirement that no s_i be a prefix of another s_j. The practical problem is how to find a suitable set of s_i's in order to maximize the total number of addressable machines, when the ith subnetwork contains n_i machines. Not all of the n_i machines of a subnetwork i need be addressable in a solution: If $n_i > 2^{L-|s_i|}$ then only $2^{L-|s_i|}$ machines of that subnetwork are addressable (none is addressable if the solution assigns no address s_i to that subnetwork). The abstract problem implied by this formulation is: Given an integer L, and given M (not necessarily distinct) positive integers $n_1, \cdots, n_M$, find M binary strings $s_1, \cdots, s_M$ (some of which may be empty) such that (i) no nonempty string s_i is prefix of another string s_j, (ii) no s_i is more than L bits long, and (iii) the quantity $\sum_{|s_k| \neq 0} \min\{n_k, 2^{L-|s_k|}\}$ is maximized. We generalize the algorithm to the case where each n_i also has a priority p_i associated with it and there is an additional constraint involving priorities: Some subnetworks are then more important than others and are treated preferentially when assigning addresses. The algorithms can be used to solve the case when L itself is a variable; that is, when the input no longer specifies L but rather gives a target integer γ for the number of addressable machines, and the goal is to find the smallest L whose corresponding optimal solution results in at least γ addressable machines.

Index Terms — Addressing, algorithms, computer networks, prefix codes.

*Portions of this work were supported by sponsors of the COAST Laboratory.

1 Introduction

This introduction discusses the connection between computer networking and the abstract problems for which algorithms are subsequently given. It also introduces some terminology.

In a computer network that consists of M subnetworks, the L-bit address of a machine consists of two parts: A prefix that contains the address of the subnetwork to which the machine belongs, and a suffix containing the address of that particular machine within its subnetwork. In the case where the various subnetworks contain roughly the same number of machines, a *fixed* partition of the L bits into a t-bit prefix, $t = \lceil \log M \rceil$, and an $(L-t)$-bit suffix, works well in practice: Each subnetwork can then contain up to 2^{L-t} addressable machines; if it contains more, then only 2^{L-t} of them will have an address and the remaining ones will be *unsatisfied*, in the sense that they will have no address. If, in a fixed length partition scheme, some machines are unsatisfied, then the only way to satisfy them is to increase the value of L. However, a fixed length scheme can be wasteful if the M subnetworks consist of (or will eventually consist of) different numbers of machines, say, n_i machines for the ith subnetwork. In such a case, the fixed scheme can leave many machines unsatisfied (for that particular value of L) even though the *variable* length partition scheme that we describe next could satisfy all of them without having to increase L.

In a variable partition scheme, the length of the prefix containing the subnetwork's address varies from one subnetwork to another. In other words, if we let s_i be the prefix that is the address of the ith subnetwork, then we now can have $|s_i| \neq |s_j|$. However, to avoid ambiguity (or having to store and transmit $|s_i|$), there is a requirement that no s_i be a prefix of another s_j. Variable length subnetwork addressing is easily shown to satisfy a larger total number of addressable machines than the fixed length scheme: There are examples where fixed length subnetwork addressing cannot satisfy all of the $N = n_1 + \cdots + n_M$ machines, whereas variable length subnetwork addressing can. Furthermore, we are also interested in the cases where even variable length addressing cannot satisfy all of the N machines: In such cases we want to use the L bits available as effectively as possible, i.e., in order to satisfy as many machines as possible. Of course an optimal solution might then leave unsatisfied all the machines of, say, the ith subnetwork; this translates into s_i being the empty string, i.e., $|s_i| = 0$. An optimal solution therefore consists of determining binary strings $s_1, \cdots, s_M$ that maximize the sum

$$\sum_{|s_k| \neq 0} \min\{n_k, 2^{L-|s_k|}\}.$$

A solution *completely satisfies* the ith subnetwork if it satisfies all of the machines of that subnetwork, i.e., if $|s_i| > 0$ and $n_i \leq 2^{L-|s_i|}$. If $|s_i| = 0$ then no machine of the ith subnetwork is satisfied, and we then say that the ith network is *completely unsatisfied.* If the solution satisfies some but not all the machines of the ith subnetwork, then that subnetwork is *partially satisfied*; this happens when $n_i > 2^{L-|s_i|}$, in which case only $2^{L-|s_i|}$ of the machines of that subnetwork are satisfied. An optimal solution can leave some of the subnetworks completely satisfied, others completely unsatisfied, and others partially satisfied.

The prioritized version of the problem models the situation where some subnetworks are more important than others. We use the following priority policy.

Priority Policy: "The number of satisfied machines of a subnetwork is the same as if all lower-priority subnetworks did not exist."

The next section proves some useful properties for a subset of the optimal solutions. We assume the unprioritized case, and leave the prioritized case until the end of the paper.

Before proceeding with the technical details of our approach, we should stress that in the above we have provided only enough background and motivation to make this paper self-contained. The reader interested in more background than we provided can find, in references [11, 8, 9, 10, 6, 4, 12], the specifications for standard subnet addressing, and other related topics. For a more general discussion of hierarchical addressing, its benefits in large networks, and various lookup solution methods (e.g., digital trees), see [7, 5]. Finally, what follows assumes the reader is familiar with basic techniques and terminology from the text algorithms and data structures literature — we refer the reader to, for example, the references [1, 2, 3].

2 Preliminaries

The following definitions and observations will be useful later on. We assume, without loss of generality, that $n_1 \geq \cdots \geq n_M$. Since the case when $n_1 \geq 2^L$ admits a trivial solution (2^L machines are satisfied, all from subnetwork 1), from now on we assume that $n_1 < 2^L$. Throughout, all logarithms are to the base 2.

Lemma 1 *Let S be any solution (not necessarily optimal). Then there exists a solution S' that satisfies the same number of machines as S, uses the same set of subnetwork addresses as S, and in which the completely unsatisfied subnetworks (if there are any) are those that have the k lowest n_i values for some integer k. In other words, $|s_i| = 0$, $M - k + 1 \leq i \leq M$.*

Proof: Among all such solutions that satisfy the same number of machines as S, consider one that has the smallest number of *offending pairs* i, j, defined as pairs i, j for which $n_i > n_j$, i is completely unsatisfied, and j is not completely unsatisfied. We claim that the number of such pairs is zero: Otherwise interchanging the roles of subnetworks i and j in that solution does not decrease the total number of satisfied machines, a contradiction since the resulting solution has at least one fewer offending pair. $\square$

On the other hand, there does *not* necessarily exist an S' of equal value to S and in which all of the (say, k) completely satisfied subnetworks are those that have the k highest n_i values. If, in the optimal solution we seek, we go through the selected subnetworks by decreasing n_i values, then we initially encounter a mixture of completely satisfied and partially satisfied subnetworks, but once we get to a completely unsatisfied one then (by the above lemma) all the remaining ones are completely unsatisfied.

Lemma 2 *Let S be any solution (not necessarily optimal). There exists a solution S' that satisfies as many machines as S, uses the same set of subnetwork addresses as S, and is such that $|s_i| > |s_j| > 0$ implies that $n_i \leq n_j$.*

Proof: Among all such solutions that satisfy the same number of machines as S, consider one which has the smallest number of *offending pairs* i, j, defined as pairs i, j such that $|s_i| > |s_j| > 0$ and $n_i > n_j$. We claim that the number of such pairs is zero: Otherwise interchanging the roles of subnetworks i and j in that solution does not decrease the total number of satisfied machines, a contradiction since the resulting solution has at least one fewer offending pair. $\square$

Let T be a full binary tree of height L, i.e., T has 2^L leaves and $2^L - 1$ internal nodes. For any solution S, one can map each nonempty s_i to a node of T in the obvious way: The node v_i of T corresponding to subnetwork i is obtained by starting at the root of T and going down as dictated by the bits of the string s_i (where a 0 means "go to the left child" and a 1 means "go to the right child"). Note that the depth of v_i in T (its distance from the root) is $|s_i|$, and that no v_i is ancestor of another v_j in T (because of the requirement that no nonempty s_i is a prefix of another s_j). For any node w in T, we use $parent(w)$ to denote the parent of w in T, and we use $l(w)$ to denote the number of leaves of T that are in the subtree of w; hence $l(v_i) = 2^{L-|s_i|}$. Observe that solution S completely satisfies subnetwork i iff $l(v_i) \geq n_i$, in which case we can extend our terminology by saying that "node v_i is completely satisfied by S" rather than the more accurate "the subnetwork i corresponding to node v_i is completely satisfied by S."

Lemma 3 *Let $S = (v_1, \ldots, v_k)$ be any solution that satisfies Lemmas 1 and 2. Then there is a solution $S' = (v'_1, \ldots, v'_k)$ that, for each subnetwork i ($1 \leq i \leq k$), has v'_i at the same depth as v_i,* *and is such that $i < j$ implies that v'_i has smaller preorder number in T than v'_j (which is equivalent to saying that s'_i is lexicographically smaller than s'_j).*

Proof: S' can be obtained from S by a sequence of "interchanges" of various subtrees of T, as follows. Set $i = 1$, let T' be initially a copy of T, and repeat the following until $i = k$:

1. Perform an "interchange" in T' of the subtree rooted at node v_i with the subtree rooted at the leftmost node of T' having same depth as v_i; v'_i is simply the new position occupied by v_i after this "interchange".

2. Delete from T' the subtree rooted at v'_i, and set $i = i + 1$.

Performing in T the interchanges done on T' gives a new T where the v'_i's have the desired property. □

The "interchange" operations used to prove the above lemma will not be actually performed by our algorithm – their only use is for the proof of the lemma.

Lemma 4 *Let S be any solution (not necessarily optimal) that satisfies the properties of Lemmas 1–3. There exists a solution S' that satisfies as many machines as S, that also satisfies the properties of Lemmas 1–3, and is such that any v_i that is not the root of T has $l(parent(v_i)) > n_i$. Furthermore, the nonempty s_i's of such an S' are a subset of the nonempty s_i's of S.*

Proof: Among all solutions that satisfy the same number of machines as S, let $S' = (v_1, \ldots, v_k)$ be one that maximizes the integer i $(1 \le i \le k)$ for which all of $v_1, \ldots, v_i$ satisfy the lemma's property, i.e., they have $l(parent(v_j)) > n_j$ for all $1 \le j \le i$. We claim that $i = k$, i.e., that such an S' already satisfies the lemma. Suppose to the contrary that $i < k$, i.e., that $l(parent(v_{i+1})) \le n_{i+1}$. Node v_{i+1} cannot be completely satisfied since that would imply that $l(v_{i+1}) \ge n_{i+1}$, and hence $l(parent(v_{i+1})) = 2l(v_{i+1}) > n_{i+1}$. Hence v_{i+1} is only partially satisfied, i.e., $l(v_{i+1}) < n_{i+1}$. Let z be the parent of v_{i+1} and y be the sibling of v_{i+1} in T; y must be to the right of v_{i+1} since otherwise v_i is at y and v_i too has $l(parent(v_i)) < n_i$, which contradicts the definition of i. Also note that the fact that $l(z) \le n_{i+1}$ implies that $n_{i+1} - l(v_{i+1}) \ge l(y)$, i.e., the number of unsatisfied machines in subnetwork $i + 1$ is $\ge l(y)$. Now imagine *promoting* v_{i+1} by "moving it to its parent", one level up the tree T, thus (i) replacing the old s_{i+1} by a new (shorter) one obtained by dropping the rightmost bit of the old s_{i+1}, and (ii) deleting from S' all of the s_j that now have the new s_{i+1} as a prefix. Note that, for each s_j so removed, its corresponding v_j was in the subtree of y, hence the removal of these s_j's results in at most $l(y)$ machines becoming unsatisfied, but that is compensated for by $l(y)$ machines of subnetwork $i+1$ that have become newly satisfied as a result of v_{i+1}'s promotion, implying that the new solution S'' has value that is no less than that of S'. However, a v_j so deleted from the subtree of y can cause S'' to no longer satisfy the property of Lemma 1 because of a surviving v_t to the right of z having an $n_t < n_j$. We next describe how to modify S'' so it does satisfy Lemma 1. In the rest of the proof S' refers to the solution we started with, before v_{i+1} was moved up by one level, and S'' refers to the solution after v_{i+1} was moved.

Let $(v_{i+2}, \ldots, v_{i+2+l})$ $(0 \le l \le k-i-2)$ denote the set of the deleted v_j's (who were in y's subtree in the original S' but are not in S''). If $i+2+l < k$, then $(v_{i+3+l}, \ldots, v_k)$ are in S'' and are to the right of z, hence we need to "repair" S'' to restore the property of Lemma 1 (if on the other hand $i+2+l = k$ then no such repair is needed). This is done as follows. Simultaneously for each of the elements of the sequence $(v_{i+2}, \cdots, v_k)$, do the following: In the tree T, place the element considered (say, v_j) at the place previously (in the original S') occupied by v_{j+l+1} (if $j+l+1 > k$ then that v_j cannot be placed and the new solution leaves v_j completely unsatisfied). The S'' so modified satisfies the same number of machines as the original one, still satisfies Lemmas 1–3, but has "moved" v_{i+1} one level up the tree T. This can be repeated until v_{i+1} is high enough that $l(parent(v_{i+1}) > n_{i+1}$, but that is a contradiction to the definition of integer i. Hence it must be the case that S' has $i = k$. $\square$

Lemma 5 *There exists an optimal solution S that satisfies the properties of Lemma 4 and in which every subnetwork i has an s_i of length equal to either $L - \lceil \log n_i \rceil$ or $L - \lceil \log n_i \rceil + 1$.*

Proof: Let S be an optimal solution satisfying Lemma 4. First, we claim that there is such an S in which every s_i satisfies $|s_i| \ge L - \lceil \log n_i \rceil$. Suppose to the contrary that, in S, some s_i has length less than $L - \lceil \log n_i \rceil$. Then moving v_i from its current position, say node y in T, to a descendant of y whose depth equals $L - \lceil \log n_i \rceil$, would leave subnetwork i completely satisfied without affecting the other subnetworks. Repeating this for all i gives a solution in which every s_i has length $\ge L - \lceil \log n_i \rceil$. Of course moving a v_i down to (say) y's left subtree leaves a "hole" in y's right subtree in the sense that the right subtree of y is unulitilized in the new solution. The resulting S might have many such unutilized subtrees of T: It is easy to "move them to the right" so that they all lie to the right of the utilized subtrees of T (the details are easy and are omitted). Hence we can assume that S is such that $|s_i| \ge L - \lceil \log n_i \rceil$. (Note that the above does not introduce any violation of the properties of Lemma 4.)

To complete the proof we must show that $|s_i| \le L - \lceil \log n_i \rceil + 1$. Lemma 4 implies that

$$n_i < l(parent(v_i)) = 2l(v_i) = 2 \cdot 2^{L-|s_i|}.$$

Taking logarithms on both sides gives:

$$\lceil \log n_i \rceil \leq 1 + L - |s_i|,$$

which completes the proof. □

The observations we made so far are enough to easily solve in $O(M \log M)$ time the following (easier) version of the problem: Either completely satisfy all M subnetworks, or report that it is not possible to do so. It clearly suffices to find a v_i in T for each subnetwork i (since the v_i's uniquely determine the s_i's). This is done in $O(M \log M)$ time by the following `greedy` algorithm, which operates on only that portion of T that is above the v_i's:

1. Sort the n_i's in decreasing order, say $n_1 \geq \cdots \geq n_M$. Time: $O(M \log M)$ (the $\log M$ factor goes away if the n_i's can be sorted in linear time, e.g., if they are integers smaller than $M^{O(1)}$).

2. For each n_i, compute the depth d_i of v_i in T: $d_i = L - \lceil \log n_i \rceil$. Time: $O(M)$.

3. Repeat the following for $i = 1, \cdots, M$: Place v_i on the leftmost node of T that is at depth d_i and has none of $v_1, \cdots, v_{i-1}$ as ancestor (if no such node exists then stop and output "No Solution Exists"). Time: $O(M)$ by implementing this step as a construction and (simultaneously) preorder traversal of the relevant portion of T — call it T'; i.e., we start at the root and stop at the first preorder node of depth d_1, label it v_1 and consider it a leaf of T', then resume until the preorder traversal reaches another node of depth d_2, which is labeled v_2 and considered to be another leaf of T', etc. Note that in the end the leaves of T' are the v_i's in left to right order.

Theorem 1 *Algorithm* `greedy` *solves the problem of finding an assignment of addresses that completely satisfies all subnetworks when such an assignment exists. Its time complexity is $O(M)$ if the n_i's are given in sorted order, $O(M \log M)$ if it has to sort the n_i's.*

Proof: The time complexity was argued in the exposition of the algorithm. Correctness of the algorithm follows immediately from Lemmas 1–5. □

Theorem 2 *An assignment that completely satisfies all subnetworks exists if and only if*

$$L \geq \lceil \log(\sum_{i=1}^{M} 2^{\lceil \log n_i \rceil}) \rceil.$$

Proof: Observe that algorithm `greedy` succeeds in satisfying all subnetworks if and only if the inequality is satisfied. □

Corollary 1 *Whether there is an assignment that completely satisfies all subnetworks can be determined in $O(M)$ time, even if the n_i's are not given in sorted order.*

Proof: The right-hand side of the inequality in the previous theorem can be computed in $O(M)$ time. □

Would the greedy algorithm solve the problem of satisfying the largest number of machines when it cannot satisfy all of them? That is, when it cannot assign a v_i to a node (in Step 3), instead of saying "No Solution Exists", can it accurately claim that the solution produced so far is optimal? The answer is no, as can be seen from the simple example of $L = 3$, $M = 2$, and $n_1 = 5$, $n_2 = 3$ (for this example the greedy algorithm satisfies 5 machines whereas it is possible to satisfy 7 machines). However, the following holds.

Observation 1 *The solution returned by the greedy algorithm satisfies a number of machines that is no less than half the number satisfied by an optimal solution.*

Proof: Let m be the number of subnetworks completely satisfied by `greedy`. Observe that $n_i > l(v_i)/2$, since if we had $n_i \le l(v_i)/2$ then `greedy` would have put v_i at a greater depth than its current position. Therefore an optimal solution could, compared to `greedy`, satisfy no more than an additional $\sum_{i=1}^{m} l(v_i)/2$ machines, which is less than $\sum_{i=1}^{m} n_i =$ the number satisfied by `greedy`. □

However, we need not resort to approximating an optimal solution, since the next section will give an algorithm for finding an optimal solution.

3 Algorithm for the Unprioritized Case

We assume throughout this section that the greedy algorithm described earlier has failed to satisfy all the machines. The goal then is to satisfy as many machines as possible.

We call *level* ℓ the 2^ℓ nodes of T whose depth (distance from the root) is ℓ. We number the nodes of level ℓ as follows: $(\ell, 1), (\ell, 2), \cdots, (\ell, 2^\ell)$, where (ℓ, k) is the kth leftmost node of level ℓ.

Lemma 5 says that v_i is either at a depth of d_i or of $d_i + 1$, where $d_i = L - \lceil \log n_i \rceil$. This limits the number of choices for where to place v_i to 2^{d_i} choices at depth d_i, and 2^{d_i+1} choices at depth $d_i + 1$. For every i, j pair where $1 \le i \le M$ and $1 \le j \le 2^{d_i}$, we define $C(i, j)$ to be the maximum number of machines of subnetworks $1, \ldots, i$ that can be satisfied by using only the portion of T having preorder numbers $\le$ the preorder number of (d_i, j), and subject to the constraint that v_i is placed at node (d_i, j). $C'(i, j)$ is defined analogously but with $(d_i + 1, j)$ playing the role that

(d_i, j) played in the definition of $C(i,j)$. The $C(i,j)$'s and $C'(i,j)$'s will play an important role in the algorithm: Clearly, if we had these quantities for all i,j pairs then we could easily obtain the number of machines satisfied by an optimal solution, simply by choosing the maximum among them:

$$\max_{1\le i\le M}\{\max_{1\le j\le 2^{d_i}} C(i,j), \max_{1\le j\le 2^{d_i+1}} C'(i,j)\}.$$

Another notion used by the algorithm is that of the *ℓ-predecessor* of a node v of T, where ℓ is an integer no greater than v's depth: It is the node of T at level ℓ that is immediately to the left of the ancestor of v at level ℓ (if no such node exists then v has no ℓ-predecessor). In other words, if w is the ancestor of v at level ℓ (possibly $w = v$), then the ℓ-predecessor of v is the rightmost node to the left of w at level ℓ. The algorithms will implicitly make use of the fact that the ℓ-predecessor of a given node v can be obtained in constant time: If v is represented as a pair (a,b) where a is v's depth and b is the left-to-right rank of b at that depth (i.e., v is the bth leftmost node at depth a), then the ℓ-predecessor of (a,b) is (ℓ,c) where $c = \lceil b2^{\ell-a}\rceil - 1$.

The following algorithm `preliminary` will later be modified into a better algorithm. The input to the algorithm is L and the n_i's. The output is a placement of the v_i's in T; recall that this is equivalent to computing the s_i's because the s_i's can easily obtained from the v_i's (in fact each s_i can be obtained from v_i in constant time, as will be pointed out later). We assume that a preprocessing step has already computed the d_i's. We use $pred(\ell, v)$ or $pred(\ell, a, b)$ interchangeably, to denote the ℓ-predecessor of a node $v = (a,b)$, with the convention that $pred(\ell,a,b)$ is $(-1,-1)$ when it is undefined, i.e., when $\ell > a$ or (a,b) has no ℓ-predecessor.

1. For $i = 1$ to M in turn, do the following:

   (a) For $b = 1$ to 2^{d_i} compute

   $$C(i,b) = \max\{C(i-1, pred(d_{i-1}, d_i, b)), C'(i-1, pred(d_{i-1}+1, d_i, b))\} + \min\{n_i, 2^{L-d_i}\}$$

   with the convention that $C(i-1,-1,-1)$ and $C'(i-1,-1,-1)$ are 0.
   Let $f(i,b)$ be the node of T that "gives $C(i,b)$ its value" in the above maximization, that is, $f(i,b)$ is
   $= pred(d_{i-1}, d_i, b)$ if $C(i-1, pred(d_{i-1}, d_i, b)) > C'(i-1, pred(d_{i-1}+1, d_i, b))$,
   $= pred(d_{i-1}+1, d_i, b)$ if $C(i-1, pred(d_{i-1}, d_i, b)) \le C'(i-1, pred(d_{i-1}+1, d_i, b))$.

   (b) For $b = 1$ to 2^{d_i+1} compute

   $$C'(i,b) = \max\{C(i-1, pred(d_{i-1}, d_i+1, b)), C'(i-1, pred(d_{i-1}+1, d_i+1, b))\} + \min\{n_i, 2^{L-d_i-1}\}$$

with the convention that $C(i-1,-1,-1)$ and $C'(i-1,-1,-1)$ are 0.

Let $f'(i,b)$ be the node of T that "gives $C'(i,b)$ its value" in the above maximization, that is, $f'(i,b)$ is

$= pred(d_{i-1}, d_i+1, b)$ if $C(i-1, pred(d_{i-1}, d_i+1, b)) > C'(i-1, pred(d_{i-1}+1, d_i+1, b)$,

$= pred(d_{i-1}+1, d_i+1, b)$ if $C(i-1, pred(d_{i-1}, d_i+1, b)) \leq C'(i-1, pred(d_{i-1}+1, d_i+1, b)$.

2. Find the largest, over all i and b, of the $C(i,b)$'s and $C'(i,b)$'s computed in the previous step: Suppose it is $C(k,b)$ (respectively, $C'(k,b)$). Then $C(k,b)$ (respectively, $C'(k,b)$) is the maximum possible number of machines that are satisfied by an optimal solution $v_1, \ldots, v_k$. To generate a set of assignments that correspond to that optimal solution (rather than just its value), we use the f and f' functions obtained in the previous step: Starting at node (d_k, b) (respectively, (d_k+1, b)), we "trace back" from there, and output the nodes of the optimal solution as we go along (in the order $v_k, v_{k-1}, \ldots, v_1$). The details of this "tracing back" are as follows:

   (a) Set $i = k$. If the largest of the $C(i,b)$'s and $C'(i,b)$'s computed in the previous step was $C(k,b)$ (respectively, $C'(k,b)$) then set (α, β) equal to (d_k, b) (respectively, (d_k+1, b)). Then repeat the following until $i = 1$.

   (b) Output "$v_i = (\alpha, \beta)$ " then set (α, β) equal to either $f(i, \beta)$ (in case $\alpha = d_i$) or to $f'(i, \beta)$ (in case $\alpha = d_i + 1$).

   Note. To output the string s_i corresponding to a v_i node, rather than the (d_i, j) or (d_i+1, j) pair describing that v_i, we modify the above Step 2(b) as follows: If $v_i = (a, b)$ then s_i is the binary string consisting of the rightmost a digits in the binary representation of the integer $2^a + b - 1$ (note that $2^a + b - 1$ is the breadth-first number of the node (a,b), and that an empty string corresponds to the root since $2^0 + 1 - 1 = 1$). This implies that s_i can be computed from the pair (a,b) in constant time.

Correctness of the above algorithm `preliminary` follows from Lemmas 1 – 5.

The time complexity of `preliminary` is unsatisfactory because it can depend on the size of T as well as M, making the worst case take $O(M2^L)$ time. However, the following simple modification results in an $O(M^2)$ time algorithm. In Steps 1(a) and (respectively) 1(b), replace "For $b = 1$" by "For $b = \max\{1, 2^{d_i} - M\}$" and (respectively) "For $b = \max\{1, 2^{d_i+1} - M\}$" (the upper iteration bounds for b remain unchanged, at 2^{d_i} for 1(a) and 2^{d_i+1} for 1(b)). Before arguing the correctness of this modified algorithm, we observe that its time complexity is $O(M^2)$, since we now iterate over only M^2 distinct i, b pairs. (*Implementation note:* The relevant $C(i,b)$'s need not be explicitly initialized, they can implicitly be assumed to be zero initially; this works because of the particular order in which Step 1 computes them.) Correctness follows from the claim (to be proved next) that there is an optimal solution that, of the 2^a nodes of any level a, does not use any of the leftmost $2^a - M$ nodes of that level. Let S be an optimal solution that has the smallest possible number (call it t) of violations of the claim, i.e., the smallest number of nodes (a,b) where $b < 2^a - M$ and some v_i is at (a,b). We prove that $t = 0$ by contradiction: Suppose that $t > 0$, and let a be the smallest depth at which the claim is violated. Let (a,b) be a node of level a that violates the claim, i.e., $b < 2^a - M$ and some v_i is placed at (a,b) by optimal solution S. Since there are more than M nodes to the right of v_i at level a, the value of S would surely not decrease if we were to modify S by re-positioning all of $v_i, v_{i+1}, \ldots, v_M$ in the subtrees of the rightmost $M - i + 1$ nodes of level a (without changing their depth). Such a modification, however, would decrease t, contradicting the definition of S. Hence t must be zero, and the claim holds.

The following summarizes the result of this section.

Theorem 3 *The unprioritized case can be solved in $O(M^2)$ time.*

4 Algorithm for the Prioritized Case

Let the priorities be $p_{k_1} > p_{k_2} > \cdots > p_{k_M}$ where p_{k_i} is the priority of subnetwork k_i. In the rest of this section we assume that L is not large enough to completely satisfy all of the M subnetworks (because in the other case, where L is large enough, the priorities do not play a role and Theorem 1 applies).

Use `greedy` (or, alternatively, Corollary 1) in a binary search for the largest i (call it $\hat{i}$) such that the subnetworks $k_1, \cdots, k_i$ can be completely satisfied; each "comparison" in the binary search corresponds to a call to `greedy` (or, alternatively, to Corollary 1) – of course it ignores the priorities of the subnetworks $k_1, \cdots, k_i$. This takes total time $O(M \log M)$ even though we may use `greedy` a logarithmic number of times, because we sort by decreasing n_j's only once, which makes each subsequent execution of `greedy` cost $O(M)$ time rather than $O(M \log M)$. Let S be the solution, returned by `greedy`, in which all of subnetworks $k_1, \cdots, k_{\hat{i}}$ are completely satisfied. By the definition of $\hat{i}$, it is impossible to completely satisfy all of subnetworks $k_1, \cdots, k_{\hat{i}+1}$. Our task is to modify S so as to satisfy as many of the machines of subnetworks $k_{\hat{i}+1}, \ldots k_M$ as possible without violating the priority policy (hence keeping subnetworks $k_1, \cdots, k_{\hat{i}}$ completely satisfied).

This is done as follows:

1. Set $j = \hat{i} + 1$, and set the depth of each k_i, $1 \leq i \leq j-1$, to be $\lceil \log n_{k_i} \rceil$.

2. Use `greedy` $\log \log n_{k_j}$ times to binary search for the smallest depth (call it d) at which v_{k_j} can be placed without resulting in the infeasibility (as tested by `greedy`) of (i) placing all of subnetworks $k_1, \ldots, k_{j-1}$ at their previously fixed depths and (ii) placing k_j at depth d (there are $\log n_{k_j}$ possible values for d, which implies the $\log \log n_{k_j}$ iterations of the binary search). If no such d exists (i.e., if any placement of k_j prevents the required placement of $k_1, \ldots, k_{j-1}$) then proceed to Step 3. If the binary search finds such a d then fix the depth of v_j to be d (it stays d in all future iterations), set $j = j + 1$, and repeat Step 2.

3. The solution is described by the current depths of $k_1, \ldots, k_{j-1}$. These fixed depths are then used by a preorder traversal of (part of) T to position $v_{k_1}, \ldots, v_{k_{j-1}}$ in T.

That the above algorithm respects the priority policy follows from the way we fix the depth of subnetwork k_j: Subnetworks of lower priority do not interfere with it (because they are considered later in the iteration). The time complexity is easily seen to be $O(M^2 \log L)$, since $n_{k_j} < 2^L$.

The following summarizes the result of this section.

Theorem 4 *The prioritized case can be solved in $O(M^2 \log L)$ time.*

5 Further Remarks

What if L itself is a variable ? That is, consider the situation where instead of specifying L the input specifies a target integer γ for the number of addressable machines; the goal is then to find the smallest L that is capable of satisfying at least γ machines. The algorithms we gave earlier (and that assume a fixed L) can be used as subroutines in a "forward" binary search for the optimal (i.e., smallest) value of L (call it $\hat{L}$) that satisfies at least γ machines: We can use them $\log \hat{L}$ times in a "forward" binary search for $\hat{L}$. So it looks like there is an extra multiplicative $\log \hat{L}$ time factor if L is itself a variable that we seek to minimize, as opposed to the version of the problem that fixes L ahead of time. However, Theorem 2 implies that there is no such $\log \hat{L}$ factor time penalty in the important case where $\gamma = n_1 + \cdots + n_M$, i.e., where we seek the smallest L that satisfies all the machines: This version of the problem can be solved just as fast as the one where L is fixed and we seek to check whether it can completely satisfy all M subnetworks.

Acknowledgement. The authors are grateful to three anonymous referees for their helpful comments on an earlier version of this paper.

References

[1] A. Apostolico and Z. Galil (Eds), *Combinatorial Algorithms on Words,* Springer, 1985.

[2] T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[3] M. Crochemore and W. Rytter, *Text Algorithms*, Oxford University Press, 1994.

[4] Internet Assigned Numbers Authority (IANA), "Class A Subnet Experiment", RFC 1797, 04/25/1995.

[5] D. Knox and S. Panchanathan, "Parallel searching techniques for routing table lookup," *Proceedings of the 12th Annual Joint Conference of the IEEE Computer and Communications Societies - IEEE INFOCOM '93*, San Francisco, CA, v 3, 1993, pp. 1400-1405.

[6] B. Manning, "Class A Subnet Experiment Results and Recommendations", RFC 1879, 01/15/1996.

[7] A.J. McAuley and P.J. Francis, "Fast routing table lookup using CAMs," *Proceedings of the 12th Annual Joint Conference of the IEEE Computer and Communications Societies - IEEE INFOCOM '93*, San Francisco, CA, v 3, 1993, pp. 1382-1891.

[8] J. Mogul and J. Postel, "Internet standard subnetting procedure", RFC 0950, 08/01/1985.

[9] J. Mogul, "Broadcasting Internet datagrams in the presence of subnets", RFC 0922, 10/01/1984.

[10] J. Mogul, "Internet subnets", RFC 0917, 10/01/1984.

[11] T. Pummill and B. Manning, "Variable Length Subnet Table For IPv4", RFC 1878, 12/26/1995.

[12] P. Tsuchiya, "On the Assignment of Subnet Numbers", RFC 1219, 04/16/1991.

6 Author Biographies

6.1 Mikhail J. Atallah

Mikhail J. Atallah received a BE degree in electrical engineering from the American University, Beirut, Lebanon, in 1975, and MS and Ph.D. degrees in electrical engineering and computer science from Johns Hopkins University, Baltimore, Maryland, in 1980 and 1982, respectively. In 1982, Dr. Atallah joined the Purdue University faculty in West Lafayette, Indiana; he is currently a professor in the computer science department. In 1985, he received an NSF Presidential Young Investigator Award from the U.S. National Science Foundation. His research interests include the design and analysis of algorithms, in particular for the application areas of computer security and computational geometry.

Dr. Atallah is a Fellow of the IEEE, and serves or has served on the editorial boards of *SIAM J. on Computing, J. of Parallel and Distributed Computing, Information Processing Letters, Computational Geometry: Theory & Applications, Int. J. of Computational Geometry & Applications, Parallel Processing Letters, Methods of Logic in Computer Science.* He was Guest Editor for a Special Issue of *Algorithmica* on Computational Geometry, has served as Editor of the *Handbook of Parallel and Distributed Computing* (McGraw-Hill), as Editorial Advisor for the *Handbook of Computer Science and Engineering* (CRC Press), and serves as Editor in Chief for the *Handbook of Algorithms and Theory of Computation* (CRC Press). He has also served on many conference program committees, and state and federal panels.

6.2 Douglas E. Comer

Douglas E. Comer received his B.S. from Houghton College in 1971 and earned his Ph.D from Pennsylvania State University in 1976. He joined the Purdue faculty in 1976, and is currently a professor in the computer science department. Dr. Comer is an internationally recognized expert on TCP/IP, who gives lectures at various network meetings and as a consultant to private industry. He has authored many well-known textbooks, which include the titles: Operating System Design: The Xinu Approach; Operating System Design, Vol.II Internetworking with Xinu; Internetworking with TCP/IP (three volumes); The Internet Book; and Computer Networks and Internets (all Prentice Hall books). He is Editor of *Software-Practice* *and Experience* (John Wiley) and Editor-in-Chief of *Internetworking, Research and Experience* (John Wiley). Dr. Comer is on leave of absence for two years from Purdue, and is serving as Dean of the Interop Graduate Institute for Softbank Corp. He has lectured on TCP/IP and Networking at the NETWORLD+INTEROP conference workshops worldwide. He is the former chairman of the DARPA Distributed Systems Architecture Board and the CSNET Technical Committee, and member of the Internet Activities Board. He is a member of Sigma Xi and Upsilon Pi Epsilon honoraries.