CERIAS Tech Report 2005-10

ALGORITHMS FOR VARIABLE LENGTH SUBNET ADDRESS ASSIGNMENT

by Mike Atallah, Sundararaman Jeyaraman

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Algorithms for Variable Length Subnet Address Assignment

Mike Atallah, Sundararaman Jeyaraman
CERIAS and Department of Computer Sciences
Purdue University, West Lafayette, IN 47906
{mja,jsr}@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. An interesting practical problem is how to find a suitable set of s_i's in order to maximise the total number of addressable machines, when the ith subnetwork contains n_i machines. A solution might leave some subnetworks completely unsatisfied and the rest of the subnetworks completely satisfied; 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 a prefix of another string s_j, (ii) no s_i is more than L bits long (iii) the quantity $\sum_{|s_i| \neq 0} n_i$ is maximised and (iv) Every nonempty prefix completely satisfies the corresponding subnetwork - *i.e.*, $|s_i| \neq 0 \Longrightarrow 2^{L-|s_i|} \geq n_i, 1 \leq i \leq M$. We present a polynomial time algorithm for solving the aforementioned abstract problem. We also provide an algorithm to solve the case where each n_i has

a priority 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. We also make observations about the case where there is a hierarchy of subnetworks present.

1 Introduction

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

In this introduction, we provide just enough background information to make this paper self-contained. The reader interested in reading more about standard subnet addressing, variable length subnet addressing and other related IETF specifications is encouraged to peruse through [4, 12–17]. A more general discussion on hierarchical addressing, its benefits in large networks and the various IP lookup solution methods could be found in [5–7]. We also assume that the reader is familiar with basic techniques from the algorithms and data structures literature found in standard references like [1–3].

Variable length subnet address assignment is typically used for effective utilisation of the address space at the disposal of an organisation or any administrative domain — especially in the presence of subnetworks with varied demands i.e., varied number of hosts. In a mobile world, where subnetworks consisting partially or entirely of mobile nodes (possibly MANETS) are the norm, the problem of automated dynamic allocation of subnet addresses becomes pertinent. Another problem that could come to the fore is allocation of subnet addresses in the presence of constraints like a limited address space. (e.g.,) An academic department with limited IP addresses having to cope with a sudden increase in demand during a conference. Solutions based on address-configuration techniques[8][21] could be used for solving the automated dynamic allocation problem. But such solutions do not perform correctly in the presence of partitions and host mobility. A NAT based approach could be used to handle the resource constrained case. But some of the

subnetworks might contain hosts that do not wish to function behind a NAT box. Also, the NAT based approach does not work in the presence of mobile hosts that travel across administrative domains.

In this paper, we develop algorithms and insights for effective dynamic allocation of subnet addresses. We examine the scenarios when the demand is greater than the existing reseources and when constraints like priorities dictate the allocation of addresses to the different subnetworks. We show that eventhough the general allocation problem (which we define later in this section) looks deceptively similar to other resource allocation problems, most of which have been proven to be NP-complete, polynomial time solutions are possible in certain scenarios.

In a computer network consisting of M subnetworks, the L-bit address of a machine in the network is composed of two parts: A prefix that identifies the subnetwork to which the machine belongs and a suffix that contains the address of that particular machine within its subnetwork. If all the subnetworks contain roughly the same number of machines, a ***fixed*** partitioning of the address space works well in practise. In such a ***fixed*** length scheme, each subnetwork is assigned the same number of addresses – The L-bit address of any machine consists of a fixed length t-bit prefix and an $(L-t)$-bit suffix, where $t = \lceil \log M \rceil$. However, if the M subnetworks were to consist of different number of machines, say, n_i machines for the ith subnetwork, such a fixed length scheme proves to be wasteful. It could potentially leave many machines ***unsatisfied*** (i.e.,) they will have no address assigned to them and the only way to satisfy those machines is to increase the address space.

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. More importantly, we are 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, subject to certain constraints. [18]describes a polynomial time algorithm when the optimal solution is allowed to contain *partially* satisfied subnetworks. In this paper, we describe a polynomial time algorithm to find the optimal solution when it is constrained to contain only *completely* satisfied subnetworks. An optimal solution therefore consists of binary strings $s_1, \cdots, s_M$ such that $|s_i| \neq 0 \Longrightarrow 2^{L-|s_i|} \geq n_i, 1 \leq i \leq M$ and maximise the following sum

$$\sum_{|s_k| \neq 0} 2^{L-|s_k|}$$

The prioritised 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". We present a polynomial time algorithm to handle the prioritised version. Finally, we comment on the case where there is a hierarchy of networks present.

This paper is organised as follows: Section 2 reviews related work and compares and contrasts them with our work. Section 3 discusses the definitions and observations that lead to our algorithm. Section 4 introduces the unprioritised version of the algorithm. Section 5 presents the prioritised version of the algorithm. Section 6 discusses future research and finally, the conclusions are summarised in section 7.

2 Related Work

The problem of subnet address allocation is an instance of a well known general resource allocation problem in which blocks of resources are allocated from a resource "pool", based on a series of requests. Resource allocation problems arise

very frequently in a variety of contexts ranging from memory management, distribution of zip codes and telephone numbers to bandwidth allocation in computer networks. In this section, we briefly go over various approaches to solve those resource allocation problems that closely resemble the subnet address allocation problem.

`Memory Management` involves contiguous bytes of memory being allocated and de-allocated over time. While there are many memory allocation algorithms, the buddy allocation strategy [19] exhibits characteristics similar to those required to solve the subnet address allocation problem. Since memory is cheap, there is no notion of resource constraint and having to share the available resources in the most efficient manner.

`Multicast address allocation problem (Malloc).` The Any Source Multicast (ASM) requires that applications share a single, global address space. A multicast address identifies a logical group of members and any source may send data to this dynamic set of members any time. The key allocation problem here is to assign a unique address to each application from a limited globally-shared address space. [9, 10] describe the MASC address allocation architecture for dynamically allocating multicast addresses. [11] models the Malloc problem theoretically and provides complexity results for various allocation strategies. The key difference between the malloc problem and our problem is that, while a solution to our problem is restricted to using a prefix-based allocation scheme, malloc solutions are also free to use contiguous and non-contiguous address allocation schemes [11]. Figure 2 illustrates the difference between prefix-based, contiguous and non-contiguous allocation schemes.

`Subcube allocation in HyperCubes.` A hypercube is a recursive mathematical structure that served as the underlying communication network of the Intel iPSC and N-Cube parallel processors. A hypercube consists of 2^n processors where each processor is labeled with an n–bit address. Processors whose labels differ in exactly one bit position are connected. A *subcube* of a hypercube is a subset of its nodes and edges that themselves form a smaller hypercube. In a

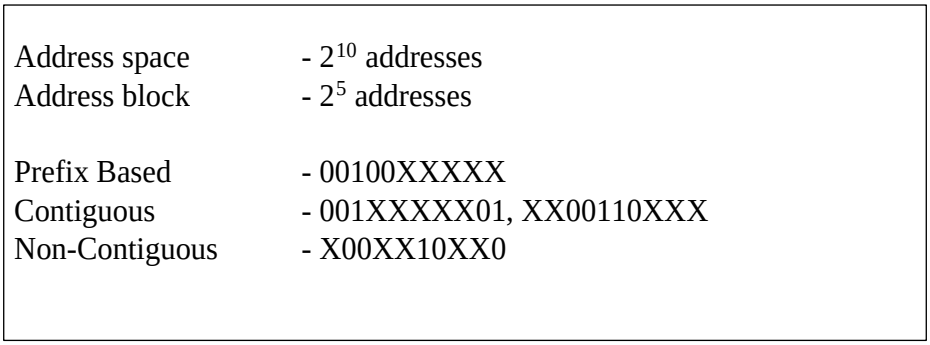

Figure 1: Examples for prefix-based, contiguous and non-contiguous allocation schemes

hypercube machine, parallel applications request subcubes, hold them for the runtime of the application, and then release the subcubes back to the operating system scheduler. Considerable research has gone into developing *subcube allocation algorithms* [22–25]. [26] proves that the malloc problem and the subcube allocation problem are infact quite similar to each other. Similarly, subcube allocation strategies do not face the constraint of having to use only prefix-based schemes as we do.

Despite the key differences between our problem and the afore-mentioned problems, we hope to benefit from the theory developed in those contexts.

3 Prelimnaries

The following definitions and observations will be useful later on. We assume, without loss of generality, that $n_1 \geq \cdots \geq n_M$. 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 1. *Let $S = (v_1, \ldots, v_k)$ be any solution (not necessarily optimal). 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 2. *Let S be a solution set (optimal or otherwise). $S = \{v_i \mid 1 \leq i \leq M, l(v_i) \geq n_i\}$. If S has more than one element, then S can be partitioned into two disjoint subsets S' and S'' such that, for some k,*

$$S' = \{v_x \mid v_x \in S, 1 \leq x \leq k\},\ S'' = \{v_y \mid v_y \in S, k+1 \leq y \leq M\},$$

Proof: Let $S = (v_1, \ldots, v_i)$ be a solution. Use lemma 3 to get a solution $S' == (v'_1, \ldots, v'_i)$ so that $i < j$ implies that $n_i \geq n_j$. Let T be the binary tree of which $v'_1, \ldots, v'_i$ are interior nodes. Let T', T'' be the left and right subtrees of the root of the binary tree T respectively. It is easy to see that there exists a v'_k such that $(v'_1, \ldots, v'_k)$ are found in T' and $(v'_k + 1, \ldots, v'_i)$ in T''. □

lemma 3. *Let S be an optimal solution. $S = \{v_i \mid 1 \leq i \leq M\}$. If S has only one element,* i.e., *$|S| = 1$, then the node v_i that corresponds to the root of the binary tree T is assigned to the subnetwork i such that*

$$\max_{1 \leq i \leq M} \{n_i | l(v_i) \geq n_i\}$$

□

4 Algorithm for the unprioritised case

4.1 A pseudo-polynomial time algorithm

Let $S = \{s_1, \ldots, s_M\}$ be an optimal solution. We define $F(i, j, \ell)$ to be the maximum number of machines satisfied by any solution using ℓ bits, if the solution set were to contain only those subnetworks in the set $i, i+1, \cdots j$ i.e., in the solution set, if for any $k, |s_k| \neq 0 \Longrightarrow i \leq k \leq j$. We use the convention that $F(i, j, \ell)$ is 0 when undefined, *i.e.,* when $i > j$. If S were to contain more than one element, according to Lemma 2, the optimal solution can be obtained in terms of optimal solutions of subproblems. It allows us to define $F(i, j, \ell)$ using the following recursive formula,

$$F(i, j, \ell) = \max_{1 \leq k \leq M} \{F(i, k, \ell - 1) + F(k + 1, j, \ell - 1)\} \tag{1}$$

If S were to contain only one element, then Lemma 3 can be used to define $F(i, j, \ell)$ with the following formula,

$$F(i, j, \ell) = \max_{1 \leq k \leq M} \{n_k, \text{if } n_k \leq 2^\ell\} \tag{2}$$

Since we do not know if S will consist of only one subnetwork, $F(i, j, \ell)$ is the maximum of the two values specifi ed in equations 1 and 2,

$$F(i,j,\ell) = \begin{cases} 0 & \text{if } j < i \\ \max \left\{ \begin{array}{l} \max_{1 \le k \le M}\{F(i,k,\ell-1) + F(k+1,j,\ell-1)\}, \\ \max_{1 \le k \le M}\{n_k, \text{if } n_k \le 2^\ell\} \end{array} \right\} & \text{otherwise} \end{cases} \quad (3)$$

Clearly, the maximum number of machines that can be satisfi ed by the optimal solution using L bits is $F(1, M, L)$.

The algorithm `CalculateOptimal` described below takes L and the n_i's as inputs and computes the entries in the $F(1 \cdots M, 1 \cdots M, 1 \cdots L)$ table. It also maintains the table $f(1 \cdots M, 1 \cdots M, 1 \cdots L)$ to help us keep track of how to construct the optimal solution. Intuitively, $f(i, j, \ell)$ points to the subnetwork k that "gives $F(i, j, \ell)$ its value". The algorithm returns the F and f tables.

The running time of the algorithm is $O(m^3 L)$, since the calculation of each table entry takes $O(M)$ time (steps $8 - 17$) and there are $2 * m^2 L$ table entries.

The f table returned by algorithm `CalculateOptimal` can be used to construct an optimal solution S as follows: For any $F(i, j, \ell)$ that resulted because of equation 1, then $f(i, j, \ell)$ corresponds to that subnetwork k that is the sole member of the solution set. In this case, we just print out the leftmost node v_i of the binary tree T that has the same depth as $L - \lceil \log_{n_k} \rceil$. On the other hand, if $F(i, j, \ell)$ had resulted because of equation 2, then $f(i, j, \ell)$ corresponds to that subnetwork k that 'splits' the solution set S into two disjoint sets S' and S'' (refer to lemma 2). In this case, we call the same procedure recursively on the two disjoint portions of the table split by k. The following recursive algorithm `PrintOptimal` prints out the optimal solution as described above.

It is easy to observe that the running time of `PrintOptimal` is $O(M)$. Observe that there can be atmost M recursive calls to `PrintOptimal`.

An astute reader would have observed that the $O(m^3 L)$ running time of algorithm `CalculateOptimal` is not polynomial time, but infact pseudo-polynomial

Algorithm 1 CalculateOptimal - Calculating the optimal solution

for $\ell = 1$ to L **do**
  for $i = 1$ to M **do**
    for $j = 1$ to M **do**
      $F(i, j, \ell) = 0$
      $f(i, j, \ell) = 0$
      if $i > j$ **then**
        $F(i, j, \ell) = 0$
        for $k = 1$ to M **do**
          $q = F(i, k, \ell - 1) + F(k + 1, j, \ell - 1)$
          if $q < n_k$ **then**
            $q = n_k$
          end if
          if $q > F(i, j, \ell)$ **then**
            $F(i, j, \ell) = q$
            $f(i, j, \ell) = k$
          end if
        end for
      end if
    end for
  end for
end for

Algorithm 2 PrintOptimal(i, j, ℓ) - Printing the optimal Solution

Let k = $f(i, j, \ell)$.
Let T' be initially a copy of the binary tree T.
if $F(i, j, \ell) = F(i, k, \ell - 1) + F(k + 1, j, \ell - 1)$ **then**
  Call `PrintOptimal`($i, k, \ell - 1$).
  Call `PrintOptimal`($k + 1, j, \ell - 1$).
else
  Print out the leftmost node v_i of T' having the same depth as $L - \lceil \log_{n_k} \rceil$.
  Delete from T' the subtree rooted at v_i.
end if

time. The size of the input is $\sum_{1 \leq i \leq M} n_i + \log L$ and the factor of L (instead of $\log L$) present in the running time makes it pseudo-polynomial. In the next section, we describe a polynomial time solution.

4.2 Polynomial time solution

In this section, we show that the optimal solution can be characterised in a way that yields a polynomial time dynamic programming solution.

We call *level* ℓ the 2^ℓ nodes of the binary tree 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. We know from our problem definition that subnetwork i is either assigned a node v_i at depth d_i, where $d_i = L - \lceil \log n_i \rceil$ or it is not assigned any node at all (i.e., $|s_i| = 0$). This limits the number of choices for where to place v_i to 2^{d_i} choices at depth d_i, if at all placed. For every i, j pair where $1 \leq i \leq M$ and $1 \leq j \leq 2^{d_i}$, we define $F(i, j)$ 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 $\leq$ the preorder number of (d_i, j). Let A be a corresponding optimal solution.

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 $c = \lceil b2^{\ell - a} \rceil - 1$. We use $lpred(\ell, v)$ or $lpred(\ell, a, b)$ interchangeably, to denote the ℓ-predecessor of a node $v = (a, b)$, with the convention that $lpred(\ell, a, b)$ is $(-1, -1)$ when it is undefined, i.e., when $\ell > a$ or (a, b) has no ℓ-predecessor.

If $d_{i-1} > d_i$, v_i can be safely placed at (d_i, j). Because of the difference in

depth, none of the $v_1, \ldots, v_{i-1}$ nodes can be placed at (d_i, j). In that case, $F(i, j)$ can be defined as

$$F(i, j) = F(i - 1, lpred(d_{i-1}, j)) + n_i \tag{4}$$

If $d_{i-1} = d_i$, the node (d_i, j) can be used to satisfy any of the subnetworks $1, \ldots, i$ having the same depth as d_i. Hence, $F(i, j)$ is defined as

$$F(i, j) = \max\{F(i - 1, j), F(i - 1, lpred(d_{i-1}, j)) + n_i\} \tag{5}$$

Since the substructure of the optimal solution A is not known beforehand, $F(i, j)$ is defined as the maximum of equation 4 and 5. Clearly, if we had $F(i, j)$'s for all i, j pairs, then the maximum number of machines satisfied by an optimal solution is obtained by choosing the maximum among them:

$$\max_{1 \le i \le M, 1 \le j \le 2^{d_i}} F(i, j) \tag{6}$$

We can avoid calculating $F(i, j)$'s for the entire range of j's from 1 to 2^{d_i} because of the following claim: 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. Hence

the maximum number of machines satisfied by an optimal solution is:

$$\max_{\substack{1\le i\le M \\ \max\{1,2^{d_i}-M\}\le j\le 2^{d_i}}} \{F(i,j)\} \tag{7}$$

The algorithm `PolyCalculateOptimal` described below calculates the entries in the $F(1\cdots M, 1\cdots 2^{d_i})$ table. It takes L and the n_i's as inputs. Inorder to help us construct the optimal solution, it also maintains the table $f(1\cdots M, 1\cdots 2^{d_i})$. $f(i,j)$ tells us if the optimal solution corresponding to $F(i,j)$ has v_i assigned to the node (d_i, j) or not.

Algorithm 3 PolyCalculateOptimal - Calculating the optimal solution

```
1:  for i = 1 to M do
2:    for j = max{1, 2^{d_i} − M } to 2^{d_i} do
3:      F(i, j) = F(i − 1, lpred(d_{i−1}, j)) + n_i
4:      f(i, j) = 1
5:      if d_{i−1} = d_i then
6:        if F(i, j) < F(i − 1, j) then
7:          F(i, j) = F(i − 1, j)
8:          f(i, j) = 0
9:        end if
10:     end if
11:   end for
12: end for
```

The time complexity of `PolyCalculateOptimal` is $O(M^2)$, since we iterate over M^2 distinct i, j pairs in the worst case and do constant work during each iteration.

The recursive algorithm `PolyPrintOptimal` prints out the optimal solution using the f and F tables returned by `PolyCalculateOptimal`. It is initially invoked with the i, j pair that produces the maximum $F(i,j)$ value.

The running time of `PolyPrintOptimal` is $O(M)$ since there are at the most M recursive calls.

The following summarizes the result of this section.

Theorem 1. *The unprioritized case can be solved in $O(M^2)$ time.*

Algorithm 4 PolyPrintOptimal - Printing the optimal solution

1: **if** $f(i,j) = 1$ **then**
2: Output the string s_i corresponding to the node (d_i, j). Calculation of s_i given the (d_i, j) pair takes constant time.
3: Call PolyPrintOptimal with $i-1, lpred(d_{i-1}, j)$
4: **else**
5: Call PolyPrintOptimal with $i-1, j$
6: **end if**

5 Algorithm for the Prioritised Case

In this section, we present an algorithm for the prioritised case. In the prioritised case, each subnetwork i 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. 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.'

In order to solve the prioritised case, we make use of the `greedy` algorithm described in [18], as a subroutine. We describe the algorithm briefly before proceeding to explain how it can be used to solved the prioritised case. The greedy algorithm solves a related (easier) version of our problem: Given M subnetworks, either completely satisfy them or report that it is not possible to do so. It is presented as follows:

Algorithm 5 Greedy Algorithm

1: Sort the n_i's corresponding to the M subnetworks in decreasing order, say $n_1 \geq \cdots \geq n_M$.
2: For each n_i, compute the depth d_i of v_i in T: $d_i = L - \lceil \log n_i \rceil$.
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").

Step 3 can be implemented 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.

The time complexity of the first step (sorting) is $O(M \log M)$. The second and the third step each take time $O(M)$. So, the time complexity of the `greedy` algorithm is $O(M \log M)$.

Now, we describe how the greedy algorithm can be used for solving our prioritised case. Let the priorities of the subnetworks be $p_{k_1}, \cdots, p_{k_M}$ where p_{k_i} is the priority of subnetwork k_i. Without loss of generality, let us assume that $p_{k_1} > p_{k_2} > \cdots > p_{k_M}$. Use `greedy` 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, i.e., if S is such a solution in which all subnetworks $k_1, \cdots, k_{\hat{i}}$ are completely satisfied, it is impossible to completely satisfy all of subnetworks $k_1, \cdots, k_{\hat{i}+1}$. Each "comparison" in the binary search corresponds to a call to `greedy`.

This takes total time $O(M \log M)$ instead of $O(M \log^2(M))$ even though we might end up calling `greedy` $\log(M)$ times in the worst case. The reason being, step 1 of `greedy` which takes $O(M \log M)$ time needs to be executed only once. Hence the first call to `greedy` costs $O(M \log M)$ and every subsequent call takes only $O(M)$ time.
The following summarizes the result of this section.

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

6 Future Research

In this paper, we have dealt with scenarios where blocks of addresses are allocated to subnetworks which contain only hosts. The allocated address blocks do not get divided further. We would like to look into scenarios of allocating addresses amongst competing networks that contain a multi-level hierarchy of subnetworks,

where the allocated address blocks get subdivided into subblocks according to the structure of the hierarchy of the networks. Intuitively, the potentially arbitrary substructure of the competing networks makes the problem appear very difficult to solve. Infact we believe that it might be NP-Complete. If the problem is proved to be NP-Complete, then developing approximation algorithms with tight bounds is an interesting direction we would like to explore.

The algorithms described in this paper help solve subnet address allocation when the requests for addresses are static and do not vary over time. This is a simplified view of reality where the requests are dynamic and vary over time. So, there is a need to develop theory and algorithms for optimal allocation of addresses for dynamic requests i.e., something on the lines of DHCP[20]. Some interesting questions in this regard are: In an online version of the problem, the dynamic nature of the requests means operating with a very fragmented address space. Does the online version become more difficult than the static version because of that? Can the solution for the static version be leveraged to solve the online version?

In the prioritised version of the problem discussed in section 5 we describe our priority policy and the constraint involving the priorities. We are interested in exploring scenarios with other types of constraints. Are polynomial time solutions possible for those constrained versions? Or do they degenerate to NP-complete optimisation problems? If they infact degenerate to NP-complete problems, we are interested in developing approximation algorithms.

We believe that, the subnet address allocation problem is an instance of a broad class of resource allocation problems. Hence cross pollination with theory developed for other problems is another potentially fruitful direction we are interested in exploring.

Finally, we are interested in implementing our algorithms and performing realistic simulations with arbitrary address demand functions to study address utilisation and the performance of our algorithms.

7 Conclusion

In this paper, we have developed a theoretical framework for the subnet address allocation problem. We have a developed a pseudo-polynomial time algorithm for the unprioritised version of the problem. We then showed that the algorithm can be improved to a polynomial time solution. We discuss about the prioritised version of the problem and show that it can be solved in polynomial time. We then proceed to discuss about the various problems that need to be worked on in the future.

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] 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.

[6] 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.

[7] V. Srinivasan , George Varghese, ”Faster IP lookups using controlled prefi x expansion,” *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pp. 1-10, June 22-26, 1998, Madison, Wisconsin, United States

[8] Nitin Vaidya, “Weak Duplicate Address Detection in Mobile Ad Hoc Networks” in *ACM MobiHoc*, June 2002.

[9] S. Kumar, P. Radoslavov, D. Thaler, C. Alaettinoglu, D.Estrin, and M. Handley, “The MASC/BGMP Architecture for Inter-domain Multicast Routing,” in *ACM SIGCOMM*, August 1998.

[10] P. Radoslavov, D. Estrin, R. Govindan, M. Handley, S. Kumar, and D. Thaler, “The Multicast Address-Set Claim (MASC) Protocol”, RFC 2909, September 2000.

[11] V. Lo, D. Zappala, C. GauthierDickey, and T. Singer, “A theoretical framework for multicast address allocation,” Tech. Rep. UO-TR-2002-01, University of Oregon, 2002.

[12] B. Manning, “Class A Subnet Experiment Results and Recommendations”, RFC 1879, 01/15/1996.

[13] J. Mogul and J. Postel, “Internet standard subnetting procedure”, RFC 0950, 08/01/1985.

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

[15] J. Mogul, “Internet subnets”, RFC 0917, 10/01/1984.

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

[17] P. Tsuchiya, “On the Assignment of Subnet Numbers”, RFC 1219, 04/16/1991.

[18] M. Atallah and D. Comer, “Algorithms for Variable Length Subnet Address Assignment”, *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 693-699, 1998.

[19] D. E. Knuth, *The Art of Computer Programming Vol I, Fundamental Algorithms 3rd Edition*, Addison Wesley, 1997.

[20] R.Droms, “Dynamic Host Configuration Protocol”, RFC 2131, March 1997.

[21] S. Thomson and T. Narten, “IPv6 Stateless Address Autoconfiguration”, RFC 1971, August 1996.

[22] S. Dutt and J. P. Hayes, “Subcube Allocation in Hypercube Computers”, *IEEE Transactions on Computers*, vol. 40, no. 3, March 1991.

[23] M. Chen and K. G. Shin, “Process Allocation in an N-Cube Multiprocessor Using Gray Code”, *IEEE Transactions on Computers*, vol. 36, no. 12, December 1987.

[24] A. AlDhelaan and B. Bose, “A new strategy for processor allocation in an n-cube multiprocessor”, *Proceedings of the International Phoenix Conference on Computers and Communication*, March 1989.

[25] V. M. Lo, W. Liu, B. Nitzberg, and K.Windisch, “Noncontiguous Processor Allocation Algorithms for Mesh-Connected Multicomputers”, *IEEE Transactions on Parallel and Distributed Systems*, July 1997.

[26] M. Livingston, V. Lo, D. Zappala, and K. Windisch, “Cyclic Block Allocation”, *First International Workshop on Networked Group Communication*, 1999.