A Fast Algorithm for Optimal Length-Limited Huffman Codes

Lawrence L. Larmore[†‡] and Daniel S. Hirschberg[‡]

Abstract

An $O(nL)$-time algorithm is introduced for constructing an optimal Huffman code for a weighted alphabet of size n, where each code string must have length no greater than L. The algorithm uses $O(n)$ space.

[†] Department of Computer Science, California State University, Dominguez Hills
[‡] Department of Information and Computer Science, University of California, Irvine, CA 92717.

1. Introduction

Given an alphabet $\Sigma = \{a_1, \ldots a_n\}$, where a_i occurs with frequency w_i, the *Huffman coding problem* is to find a prefix-free binary code† for Σ which minimizes the weighted length of a code string, defined to be $\sum_{i=1}^{n} w_i l_i$, where l_i is the length of the code for a_i. For example, if $n = 3$, and if $w_1 = 2$, $w_2 = 5$, and $w_3 = 3$, then the code

$$a_1 \to 00$$
$$a_2 \to 1$$
$$a_3 \to 01$$

is optimal, with weighted length 15. Huffman's algorithm [5] finds such an optimal code in time $O(n \log n)$, and can be implemented to run in $O(n)$ time if the w_i are already sorted [9].

Restricted length. A related problem is to find a prefix-free code which has minimum weighted length, subject to the restriction that, for all i, $l_i \le L$, where L is a given constant. Hu and Tan give an exponential time algorithm for finding such a code [3]. Garey, using a different approach, gives an algorithm that requires $O(n^2 L)$ time and space [1]. A hybrid algorithm, combining the methods of both Hu-Tan and Garey, runs in $O(n^{3/2} L \log^{1/2} n)$ time and requires $O(n^{3/2} L \log^{-1/2} n)$ space [9]. In this paper, we present an $O(nL)$-time algorithm which requires only linear space.

Binary trees. The Huffman coding problem is equivalent to the following problem: Given a list of weights, $w_1, \ldots w_n$, sorted into non-decreasing order, find a full binary tree‡ T with n leaves for which the *weighted path length* $= \sum_{i=1}^{n} w_i l_i$ is minimized, where l_i is the depth of the i^{th} leaf of T. (We write WPL(T) for the weighted path length of T.) The restricted length coding problem is then equivalent to that same minimum weighted path length problem, but with the restriction that the height of T cannot exceed L. (See, for example, [1].)

Methods. Huffman's original algorithm uses a greedy method. The items are first sorted by weight, and each item is considered to be a tree of just one node. A "combine" step is then

----

† A code is *prefix-free* if no code string is a prefix of any other. The advantage of a prefix-free code is that code strings can differ in length, yet any coded message can be decoded unambiguously.

‡ A *full* binary tree is a rooted tree in which every non-leaf has precisely two sons.

executed $n-1$ times. Each "combine" step deletes the two trees of smallest weight from the sorted list, combines them to form one tree (making the two smaller trees the subtrees of that new tree) and then inserts that new tree, whose weight is the sum of the weights of the subtrees, in the proper position in the sorted list. After $n-1$ iterations, the list contains just one tree, which is the Huffman tree. Hu and Tan's algorithm uses a dynamic programming approach, working across the tree from left to right. The items are first sorted by weight. For each integer $j \in [1,n]$ and each q which is an integral multiple of 2^{-L} in the range [0,1], Hu and Tan's algorithm dynamically computes the smallest possible total $\sum_{i=1}^{j} w_i l_i$, subject to the condition that $\sum_{i=1}^{j} 2^{-l_i} = q$. The best sequence $l_1, \ldots l_n$ for $j = n$, $q = 1$, is the sequence of leaves for the optimal tree. Garey's algorithm also uses dynamic programming, building optimal subtrees, starting with the smallest possible subtrees, and ending with the entire tree, analogous to Knuth's algorithm [7] for constructing optimal binary search trees. Larmore's algorithm [9] uses a hybrid of those last two methods, running the subtree algorithm for subtrees up to a certain size, then switching to the Hu-Tan left-to-right method.

In this paper, we introduce a new problem which we call the *Coin Collector's* problem, a version of the Knapsack problem. Suppose a coin collector has m coins of various denominations (face values) and various numismatic values. Since the country he lives in has binary coinage, the denomination of each coin is an integral power of 2. The collector is obliged to spend X dollars to buy groceries, but the grocer (rather unimaginatively) refuses to accept any coin at other than its face value. How can the coin collector choose a set of coins of minimum total numismatic value whose total face value is X?

We give a linear time algorithm, which we call the *Package-Merge* algorithm, for solving the Coin Collector's problem. We reduce the restricted length Huffman coding problem to an instance of the Coin Collector's problem of size nL. The Package-Merge algorithm then gives an optimal restricted length code in $O(nL)$ time.

Space complexity. The algorithm in its simple form takes $O(nL)$ space, but can be modified to take only linear space, using a technique similar to that introduced by Hirschberg [2]. The time complexity remains $O(nL)$.

2. The Package-Merge Algorithm

An instance (I,X) of the Coin Collector's problem of size m is defined by:

(a) A set I of m items, each of which has a *width* and a *weight*, such that each width is a (possibly negative) integral power of 2, and each weight is a real number. (Think of *width* as being face value of a coin, and *weight* as being numismatic value.)

(b) A non-negative real number X, which we call *total width*.

A *solution* to such an instance is defined to be a subset S of I whose widths sum to X, and an *optimal solution* is a solution of minimum total weight. We write $Opt\_Sol(I,X)$ to be such an optimal solution. If X is not diadic (a *diadic* real number is one that can be written as a fraction whose denominator is a power of 2) then no solution exists.

We now give a recursive description of the Package-Merge algorithm. Assume X is diadic, which implies that X can be written as a finite sum of distinct integral powers of 2, including possibly negative powers. If $X > 0$, write *minwidth* for the smallest of those powers of 2.

Basis: If $X = 0$, then $Opt\_Sol(I,X)$ is the empty set. If $X > 0$ and I is empty, then no solution exists.

Recursion: Let r be the smallest width of any item in I. We consider four cases.

Case 1: $r >$ *minwidth*. No solution exists.

Case 2: $r =$ *minwidth*. Let $a \in I$ be the smallest weight item of width r. Then $Opt\_Sol(I,X) = Opt\_Sol(I-\{a\},X-r) \cup \{a\}$.

Case 3: $r <$ *minwidth*, and there is just one item $a \in I$ of width r. Then $Opt\_Sol(I,X) = Opt\_Sol(I - \{a\},X)$.

Case 4: $r <$ *minwidth*, and there are at least two items in I of width r. Let $a, a' \in I$ be the two least weight items of width r, and let b be a new item, which we call a *package*, formed by combining a and a'. The width of b is $2r$, and its weight is $weight(a) + weight(a')$. Let $S' = Opt\_Sol(I-\{a,a'\}\cup\{b\},X)$. If $b \in S'$ then $Opt\_Sol(I,X) = S'-\{b\}\cup\{a,a'\}$ otherwise $Opt\_Sol(I,X) = S'$.

Correctness. We show that the Package-Merge algorithm produces an optimal solution by induction on the depth of the recursion. The basis is trivially correct, so we can assume that I is non-empty and $X > 0$. The inductive hypothesis is that the algorithm is correct for any problem instance that requires fewer recursive calls than the instance (I,X).

In Case 1, there is no solution since the width of every subset of I must be an integral multiple of r, and X is not an integral multiple of r. In Case 2, any solution must contain an odd number of items of width $minwidth = r$. The optimal solution must contain the item of that width of minimum weight, since otherwise its one item could be exchanged for that minimum weight item, causing an improvement. The remaining items must then be the optimal solution to the reduced problem. In Case 3, the one item of width r could not possibly be part of any solution, hence can be discarded. In Case 4, any solution must have total width an even multiple of r, hence must contain an even number of items of width r. If this number is 0, neither a nor a' will be in the solution, while if this number is 2 or more, both will be in any optimal solution. Thus, the two items a and a' can be “packaged" together, it being later decided whether they are both in or both out of the optimal solution. Replacing a and a' by the combined item (package) b and then recursively applying the algorithm accomplishes this.

Implementation. The Package-Merge algorithm can be implemented in $O(m)$ time provided the items are presorted, as in our application. (If not, an $O(m \log m)$-time sorting step can be included.) The space requirement is $O(m)$. Although the algorithm is described above recursively, for ease of proof, the implementation given here is non-recursive.

Let L_d be the list of items of width 2^d, sorted in order of increasing weight. By a slight abuse of notation, we shall not distinguish between an *item* and the singleton *set* of items whose sole member is that item. We refer to the *diadic expansion* of X as its representation as powers of 2. (For example, the diadic expansion of 5.625 is $2^2 + 2^0 + 2^{-1} + 2^{-3}$.)

Package Merge Algorithm(I,X)

$S \leftarrow \varnothing$
for all d, $L_d \leftarrow$ list of items having width 2^d, sorted by weight
while $X > 0$ **loop**

```
        minwidth = the smallest term in the diadic expansion of X
        if I = ∅ then
                return "No solution."
        else
                d ← the minimum such that L_d is not empty
                r ← 2^d
                if r > minwidth then
                        return "No solution."
                else if r = minwidth then
                        Delete the minimum weight item from L_d and insert it into S
                        X ← X − minwidth
                end if
                P_{d+1} ← PACKAGE(L_d)
                discard L_d
                L_{d+1} ← MERGE(P_{d+1},L_{d+1})
        end if
end loop
return "S is the optimal solution."
```

The step PACKAGE. The list P_{d+1} is formed from L_d by combining items in consecutive pairs, starting from the lightest. *I.e.*, the k^{th} item of P_{d+1} is the package formed by combining items $(2k-1)$ and $2k$ of L_d. If L_d is of odd length, its heaviest item is simply discarded. The MERGE step is just the usual merging of two sorted lists.

Time Analysis. Merging of two sorted lists takes time which is linear in the sum of the lengths of the lists, while the package step takes time which is linear in the length of the list. The following amortization argument shows that the entire algorithm takes linear time. Place three credits on each original item. Invariably, there are three credits on each item of any list L_d which consists solely of original items, two credits on each item of any list L_d which was formed by a MERGE step, and three credits on each item of P_d. The PACKAGE step combines two items which have two or three credits each into one item which has three credits, one credit paying for the operation. The MERGE step takes time which is linear in the sum of the lengths of the lists. One credit from each item (they have three each) pays for the MERGE, leaving each item with two credits.

Space Analysis. Each package can be represented as a binary tree, where the leaves are original items. The space requirement is $O(m)$.

3. The Length-Limited Huffman Coding Problem

In this section, we show how to reduce the restricted length Huffman coding problem to the Coin Collector's problem. The Package-Merge algorithm can then be applied to solve the original problem in $O(nL)$ time and $O(nL)$ space.

We assume that the input weights are non-negative. The input weights can be sorted within the stated complexity bounds, since $\log n = O(L)$ and hence we assume that the weights are presented in sorted order.

We begin with the nodeset representation of binary trees, which was introduced in [10]. Fix $n \geq 1$ and $L \geq \log_2 n$. We are only interested in full binary trees with n leaves whose height does not exceed L.

Nodeset representation. Define a *node* to be an ordered pair (i,l) such that $i \in [1,n]$, which is called the *index* of the node, and $l \in [1,L]$, which is called the *level* of the node. Any set of nodes we call a *nodeset*. If T is a tree, define

$$nodeset(T) = \{(i,l) \mid 1 \leq l \leq l_i\}$$

where l_i is the depth of the i^{th} leaf of T. For example, Figure 1 shows $nodeset(T)$ for a tree T of 7 leaves, with $L = 4$.

Width and weight. If (i,l) is any node, define $width(i,l) = 2^{-l}$, and $weight(i,l) = w_i$. If A is a nodeset, $width(A)$ and $weight(A)$ will be the sums of the widths and weights, respectively, of its constituent nodes. We make the following two observations.

1. If T is a tree, then $weight(Nodeset(T)) = \mathrm{WPL}(T)$. This follows directly from the definition of weighted path length.

2. If T is a tree with n leaves, then $width(Nodeset(T)) = n-1$. This can be proved easily by induction. The basis is n=1. This tree has one leaf at level 0 and $width(Nodeset(T)) = 0$. For the inductive step, consider T, a tree with n>1 leaves. Let a and b be two leaves that are siblings (there must be such a pair) and let f be their father at level $l \geq 0$. Let T' be T with a and b deleted. T' has $n-1$ leaves (a and b are no longer leaves and f is now a leaf) and, by the inductive

hypothesis, $width(Nodeset(T'))=n-2$. To obtain $width(Nodeset(T))$ we must subtract the contributions of $\{(f,j)\mid 1\le j\le l\}$ and add the contributions of $\{(a,j),(b,j)\mid 1\le j\le l+1\}$. That is, we must subtract $(1-2^{-l})$ and we must add $2(1-2^{-(l+1)})$. The net result is that $width(Nodeset(T))=width(Nodeset(T'))+1=(n-2)+1=n-1$.

For convenience, we assume strict monotonicity of the weights, i.e., $w_i > w_{i+1}$. No loss of generality is incurred by this assumption, since an infinitesimal value can be added to weights to force tie-breaking in the correct direction. We can also assume $w_i > 0$ for all i, since $w_i \ge 0$ and we could add an infinitesimal value to the zero weights.

Monotonicity. We say that a nodeset A is *monotone* if the following two conditions hold:

(a) for $i < n$, $(i,l) \in A$ $(i+1,l) \in A$

(b) for $l > 1$, $(i,l) \in A$ $(i,l-1) \in A$

Lemma 1. Suppose that A is a nodeset of width $I(2^{-l})+r$ where I is an integer and $0 < r < 2^{-l}$. Then A has a subset B whose width is exactly r.

Proof. By induction on the cardinality (number of nodes) of A. If A has just one node, its width must be r, and we can simply let $B = A$. If A has cardinality greater than 1, we assume the inductive hypothesis, namely that the lemma holds for all nodesets of cardinalities less than that of A. Let p be the node of A of smallest width, say 2^{-k}. If $k \le l$, we have a contradiction, since A would then have width a multiple of 2^{-k}, and hence a multiple of 2^{-l}. Thus $k > l$. Since $width(A)$ and 2^{-l} are both multiples of 2^{-k}, r must also be a multiple of 2^{-k}, hence $r \ge 2^{-k}$. If $r = 2^{-k}$, let $B = \{p\}$. Otherwise, let $A' = A-\{p\}$, let B' be the subset of A' of width $r - 2^{-k}$, obtained by the inductive hypothesis, and let $B = B'\cup\{p\}$.

Lemma 2. If $X < n$ is an integer, the minimum weight nodeset of width X is monotone.

Proof. Let A be the minimum weight nodeset of width X. If $(i,l) \in A$ and $(i+1,l) \notin A$, then $A\cup\{(i+1,l)\}-\{(i,l)\}$ has the same width as A and smaller weight, a contradiction. If $(i,l) \in A$ and $(i,l-1) \notin A$, let $A' = A\cup\{(i,l-1)\}-\{(i,l)\}$, which has the same weight as A, but width which is 2^{-l}

larger. Thus the width of A' is $X + 2^{-l}$, with X integer. By Lemma 1, there exists a nodeset $B \subseteq A'$ of width 2^{-l}. $A' - B$ has width X and weight less than A, a contradiction.

Lemma 3. If $l_1, \ldots l_n$ is a list of integers in the range $[1,L]$, and A is the nodeset $\{ (i,l) \mid 1 \le i \le n,\ 1 \le l \le l_i \}$, then $width(A) = n - \sum_{i=1}^{n} 2^{-l_i}$.

Proof. For each i, let $A_i \subseteq A$ be the set of all nodes in A of index i, i.e., $A_i = \{(i,1),\ldots(i,l_i)\}$. Then $width(A_i) = 2^{-1} + 2^{-2} + \ldots + 2^{-l_i} = 1 - 2^{-l_i}$. Summing over all i yields the result.

Lemma 4. If $w = (l_1, l_2, \ldots)$ is a monotone increasing list of non-negative integers whose width is 1, then w is the list of leaf depths of a tree.

Proof. This follows as an immediate corollary from Lemma 2.3 in [9], p.1117. For completeness, we give a proof here.

The proof is by induction on the length of w. If $|w| = 1$, then $w = (0)$ which is the list of leaf depths of a tree consisting of one leaf. Suppose $|w| = n > 1$. Define

$$x_0 = 0$$
$$x_i = x_{i-1} + 2^{-l_i}, \text{ for all } i \in [1,n].$$

Note that $(x_0, x_1, \ldots, x_n)$ is a monotone strictly increasing sequence, and that $x_n = 1$. Let k be the smallest index such that $x_k \ge \frac{1}{2}$. If $x_k > \frac{1}{2}$, we obtain a contradiction, as follows. Since w is monotone increasing, 2^{-l_i} is an integral multiple of 2^{-l_k} for all $i < k$. Thus both x_{k-1} and x_k are both multiples of 2^{-l_k}, and in fact are consecutive multiples of that quantity. But $\frac{1}{2}$, which is also a multiple of 2^{-l_i}, lies strictly between them, a contradiction. Thus $x_k = \frac{1}{2}$. Let $u = (l_1 - 1, \ldots l_k - 1)$ and let $v = (l_{k+1} - 1, \ldots l_n - 1)$. Both u and v are lists of length less than n of width 1. By the inductive hypothesis, u and v are the lists of leaf depths of trees L and R, respectively. Let T be the tree whose left and right subtrees are L and R. The list of leaf depths of T will be w.

To apply the Package-Merge algorithm, we need the following theorem.

Main Theorem. If the w_i are distinct (i.e., $w_i > w_{i+1}$ for all i) then any nodeset A that has minimum weight among all nodesets of width $n-1$ is the nodeset of a tree T that is an optimal solution to the restricted length Huffman coding problem.

Proof. Let A be the minimum weight nodeset of width $n-1$. For each i, let l_i be the largest index such that $(i,l_i) \in A$. By Lemma 2, A is monotone, hence $l_i \le l_{i+1}$. Since A is monotone and has width $n-1$, $\sum_{i=1}^{n} 2^{-l_i} = 1$ by Lemma 3. Therefore, by Lemma 4, $\{l_i\}$ is the list of leaf depths of a binary tree T, and hence $A = nodeset(T)$. If there were a tree of smaller weighted path length, the weight of its nodeset would be less than that of A and A would then not be the least weight nodeset of width $n-1$. Thus, T is optimal.

The reduction. We can find an optimal Huffman tree of depth no more than L as follows. Let each node in the nodeset be an item, each of which has width less than 1. Apply the Package-Merge algorithm to the set of all those nodes to find a minimal weight nodeset of width $n-1$. For each $l \in [1,L]$, the list of nodes of width 2^{-l} is initialized as $((n,l), (n-1,l), \ldots (1,l))$. Note that sorting of the nodes is unnecessary, since the w_i are already sorted. Ties are broken as if w_i were infinitesimally greater than w_{i+1}, so that the Main Theorem applies. We construct the optimal tree from the resulting nodeset as in the proof of the Main Theorem.

Time and space. The algorithm takes $O(nL)$ time and $O(nL)$ space. In the next section, we show how the space can be reduced to $O(n)$, while multiplying the time by only a constant factor.

4. A Linear Space Algorithm

In this section, we show how the algorithm of the previous section can be modified to solve the restricted length Huffman coding problem in $O(n)$ space, while still taking only $O(nL)$ time.

In the previous section, the restricted length Huffman coding problem was reduced to the Coin Collector's problem, where each node (coin) was an ordered pair in $[1,n]\times[1,L]$. During the course of the algorithm, “packages" were formed, each of which is a set of nodes, which could be represented as (for example) a binary tree. Each such package has a width and a weight, being the sums of the widths and weights of its constituent nodes.

At any given point in the algorithm, the number of packages that has to be remembered is fewer than $2n$ — fewer than n packages formed at the previous level plus n nodes at the current level. But these packages could have, as their members, most of the nL original nodes. Thus, $O(nL)$ space is required to keep track of everything. We propose, instead, to keep track of a very limited portion of this information, that portion being sufficient to divide the problem into two subproblems that can be worked recursively. Each stage of the recursion will require only $O(n)$ space. The size of the original Coin Collector’s problem is nL, and the total of the sizes of the Coin Collector’s problems at each stage of the recursive descent does not exceed half the size of the total at the previous stage. Thus, the total work is roughly twice that of the work during the first stage, i.e., still $O(nL)$.

It is important to note that the linear space algorithm is guaranteed to calculate the same nodeset S as the original algorithm. Each recursive call calculates the least weight nodeset of a given width within a given sub-rectangle R of $[1,n]\times[1,L]$. That nodeset will be $S\cap R$. If the recursion returned any nodeset A other than $S\cap R$, it would contradict the fact that S is the lowest weight nodeset of width $n-1$, since S could be improved by removing $S\cap R$ and replacing it with A.

We now explain in detail the first stage of the algorithm, which is illustrated by Figure 2. Let $l_{\mathbf{mid}} = \lfloor (L+1)/2 \rfloor$. The basic idea is to run the package-merge algorithm once, using only linear space, retaining only enough information to be able to break the problem into two subproblems whose total complexity does not exceed half the complexity of the original problem. Our goal is to determine the set of leaves. We can do so in linear time if we know the number of nodes at each level. As we execute the algorithm, we keep track of only the following four values for each package. Other information, such as the full set of members of a package, is discarded. The values we keep are:

$weight(p)$ = the sum of the weights of nodes in p

$width(p)$ = the total width of all nodes in p

$midct(p)$ = the number of nodes in p of level $l_{\mathbf{mid}}$

$hiwidth(p)$ = the total width of all nodes in p whose levels exceed $l_{\mathbf{mid}}$

In addition, these same values are maintained for S, which will be the optimal nodeset by the end of the algorithm. S satisfies the following two monotonicity properties (see Lemma 2).

(a) for $i < n$, $(i,l) \in A \quad (i+1,l) \in A$

(b) for $l > 1$, $(i,l) \in A \quad (i,l-1) \in A$

Let m be the number of nodes of level $l_{\mathbf{mid}}$ in S. $m = midct(S)$, which is remembered by the algorithm. We note that S can be written as the disjoint union of four sets, namely

A = nodes in S whose levels are $< l_{\mathbf{mid}}$ with indices in $[1,n-m]$

B = nodes in S whose levels are $< l_{\mathbf{mid}}$ with indices in $[n-m+1,n]$

C = nodes in S whose levels are $= l_{\mathbf{mid}}$

D = nodes in S whose levels are $> l_{\mathbf{mid}}$

Figure 2 illustrates the partition of S into A, B, C, and D.

By the monotonicity of S, the nodes in C are $(n-m+1,l_{\mathbf{mid}})$, ... $(n,l_{\mathbf{mid}})$ and the nodes in B are $[n-m+1,n]\times[1,l_{\mathbf{mid}}-1]$. Thus, we know the number of nodes in B and C at each level. We can determine the width of the four sets as follows. The width of C is $m2^{-l_{\mathrm{mid}}}$ and the width of B is $m(1-2^{-(l_{\mathrm{mid}}-1)})$. $D \subseteq [n-m+1,n]\times[l_{\mathbf{mid}}+1,L]$ and therefore the width of D is $hiwidth(S)$, which is remembered by the algorithm. The width of A is $width(S)-width(B)-width(C)-width(D)$.

Finally, A and D are (respectively) the minimum weight subsets of $[1,n-m]\times[1,l_{\mathbf{mid}}-1]$ and $[n-m+1,n]\times[l_{\mathbf{mid}}+1,L]$, of their respective widths. Thus, the number of nodes at each level of A and D can be found by recursive calls to the algorithm. (Although, in the recursive calls, the total width needed is no longer $n-1$.)

The recurrence relation we obtain is the following. Let $f(n,L)$ be the worst case time to find the minimum weight subset S of $[1,n]\times[1,L]$ of a given width X, under the assumption that S satisfies the two monotonicity properties. Then,

for $L < 3$, $\quad f(n,L) \le c_1 n$

for $L \ge 3$, $\quad f(n,L) \le c_2 nL + f(n_1,L_1) + f(n_2,L_2)$

where $L_1 = l_{\mathbf{mid}}-1 \le \lfloor L/2 \rfloor$, $L_2 = L - L_1 - 1 \le \lfloor L/2 \rfloor$, and the adversary chooses $n_1 + n_2 = n$. To obtain an upper bound on the complexity of $f(n,L)$, we note that $f(n,L) = O(g(n,L))$, where g is any function that satisfies the recurrence

for $L < 3$, $\quad g(n,L) \ge c_1 nL$

for $L \ge 3$, $\quad g(n,L) \ge c_2 nL + g(n_1,L/2) + g(n-$v'0.4m'1$,L/2)$

It is seen that $g(n,L) = (c_1+2c_2)nL$ satisfies the recurrence relation. Thus $f(n,L) = O(nL)$.

5. Additional Questions

Alphabetic Codes. A prefix free code is said to be *alphabetic* if the lexical order of the code strings corresponds to a given order of the original weighted alphabet. Itai and Wessner [6], [12] present algorithms to find an optimal length-limited alphabetic code in $O(n^2L)$ time. By using the nodeset representation of code trees, an $O(nL\log n)$ algorithm for this problem has been developed [11]. Can the complexity be reduced to $O(nL)$?

Dynamic Trees. Consider the case in which an encoding of a stream of symbols is transmitted, and the code (based on symbol frequencies) is updated after each symbol. If there are no length limitations, the optimal tree can be updated in $O(l)$ time, where l is the length of the codeword whose frequency was incremented [8]. Can the optimal length-limited code tree be updated in $O(l)$ time?

References

[1] Garey, M.R., "Optimal Binary Search Trees with Restricted Maximal Depth," *SIAM J Comp* **3** (1974) pp. 101-110.

[2] Hirschberg, D.S., "A linear space algorithm for computing maximal common subsequences," *Comm. ACM* **18** (1975), pp. 341-343.

[3] Hu, T.C. and K.C. Tan, "Path length of binary search trees," *SIAM J Applied Math* **22**

(1972) pp. 225-234.

[4] Hu, T.C. and A.C. Tucker, “Optimal computer search trees and variable length alphabetic codes," *SIAM J Applied Math* **21** (1971) pp. 514-532.

[5] Huffman, D.A., “A Method for the construction of minimum redundancy codes," *Proc. Inst. Radio Engineers* **40** (1952) pp. 1098-1101.

[6] Itai, Alon, “Optimal alphabetic trees," *SIAM J Comp* **5** (1976), pp. 9-18.

[7] Knuth, D.E. “Optimal Binary Search Trees," *Acta Informatica* **1** (1971), pp. 14-25.

[8] Knuth, D.E. “Dynamic Huffman coding," *J Algorithms* **6**, 2 (1985) pp. 163-180.

[9] Larmore, L.L., “Height-restricted optimal binary trees," *SIAM J Comp* **16** (1987) pp. 1115-1123.

[10] Larmore, L.L., “Minimum delay codes," *SIAM J Comp* **18** (1989), pp. 82-94.

[11] Larmore, L.L., “Optimal Length-Restricted Codes," Colloquium, A.T.&T. Bell Labs, Murray Hill, NJ, January 6, 1988.

[12] Wessner, Rusell L., “Optimal alphabetic search trees with restricted maximal height," *Information Processing Letters* **4** (1976), pp. 90-94.

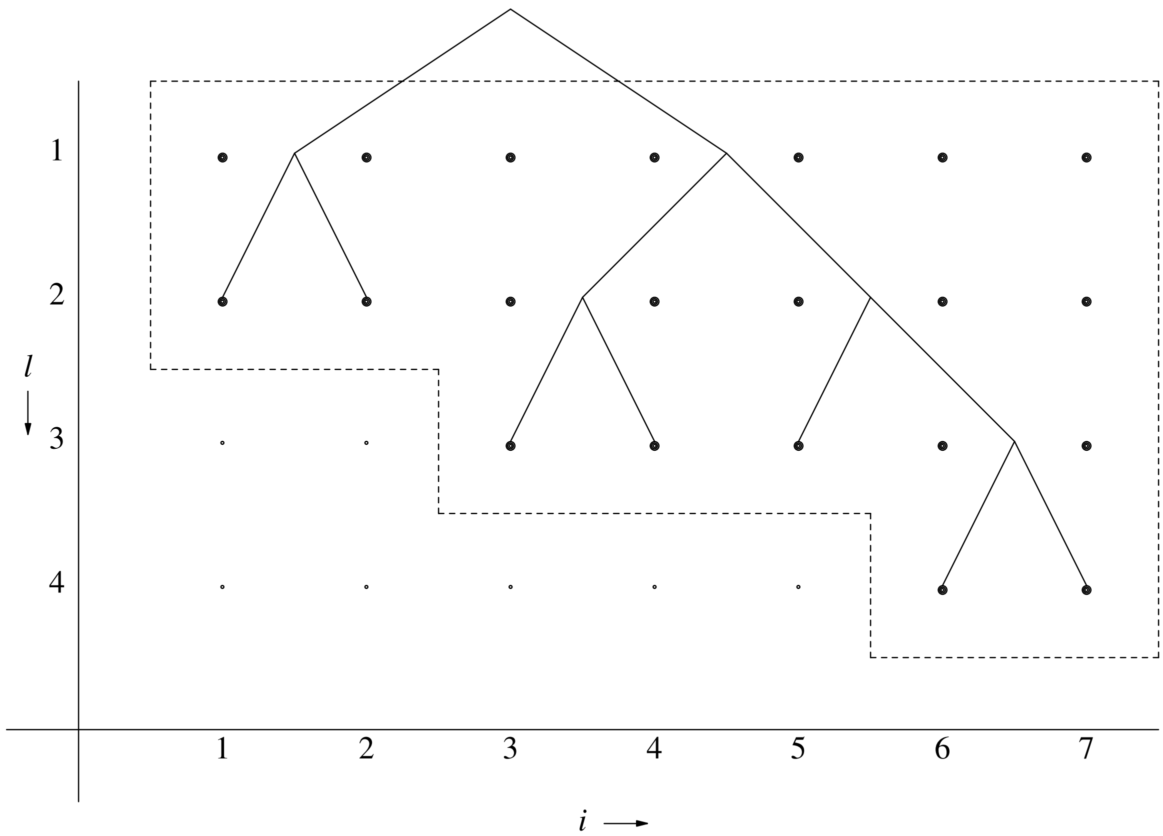

Figure 1. *nodeset(T)*

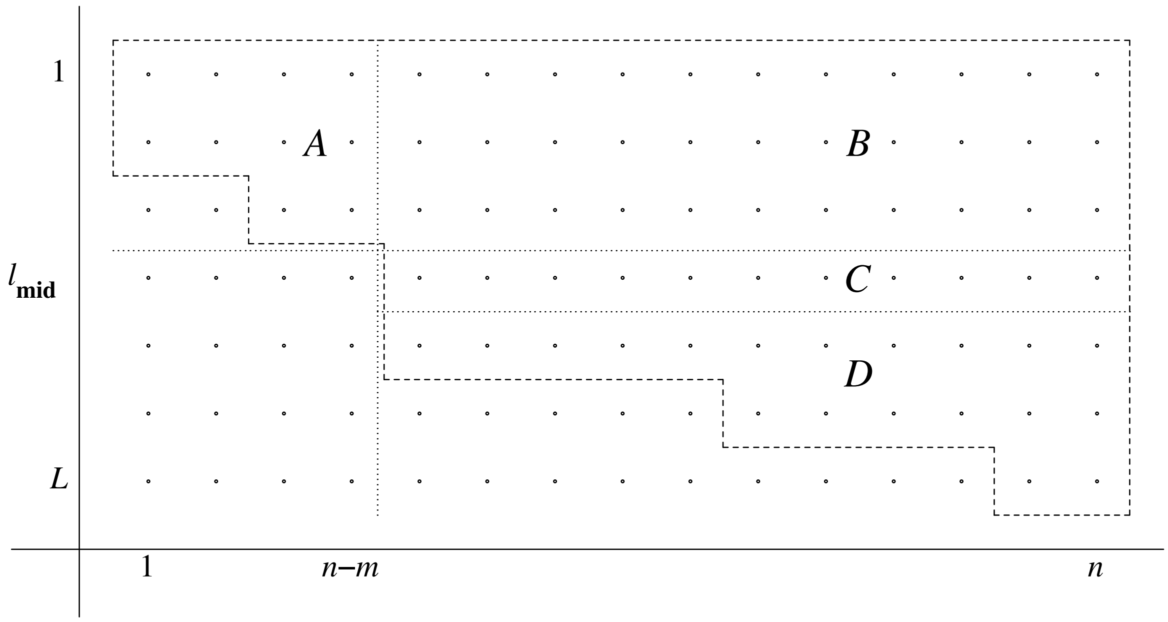

Figure 2. *Sets A, B, C, D*