Length-Limited Coding

Lawrence L. Larmore*
Daniel S. Hirschberg#

Abstract

An $O(nL)$-time algorithm is given for finding an optimal prefix-free binary code for a weighted alphabet of size n, with the restriction that no code string be longer than L. An $O(nL\log n)$-time algorithm is given for the corresponding alphabetic problem, which is equivalent to optimizing a dictionary of n words, implemented as a binary tree of height $h \leq L$ with all data in the leaves.

1. Introduction

Huffman's problem. Suppose Σ is an alphabet of size n, and ϕ_i is the frequency with which the i^{th} symbol of Σ is transmitted. A binary tree[1] with n leaves determines a prefix-free[2] binary code for Σ, and a tree which minimizes the *weighted depth* $T = \sum_{i=1}^{n} \phi_i l_i$ (where l_i is the depth of the leaf in T containing the i^{th} symbol of Σ) determines a code where the expected length of a code string is minimized. Figure 1 shows the correspondence between a binary tree and a binary code.

Huffman's algorithm [Huf] 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 [L].

The Alphabetic Coding problem. In [HuTu], Hu and Tucker present an $O(n\log n)$ solution to the variation of Huffman's problem in which the desired tree must satisfy the alphabetic property, *i.e.*, if the tree were to be traversed in symmetric order, the symbols (contained in the leaves) must be encountered in alphabetic order. Thus, the tree could be used as a binary search tree.

* Department of Mathematics and Computer Science, University of California, Riverside, CA 92521.
# Department of Information and Computer Science, University of California, Irvine, CA 92717.

[1] In this paper, each non-leaf node in a binary tree has exactly two children.

[2] 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.

The non-alphabetic problem can be shown to be reduced to the alphabetic problem by simply sorting the weights [HuTa].

Height-limited optimal trees. A variation on the above two problems restricts solution trees to have height at most L, where L is a given constant. These two problems are sometimes referred to as the *Length-Limited Coding* and the *Length-Limited Alphabetic Coding* problems.

Previous results. The Length-Limited (non-alphabetic) Coding problem is solved in $O(nL2^L)$ time by Hu and Tan [HuTa], in $O(n^2L)$ time by Garey [Ga], and in $O(n^{1.5}L\log^{0.5}n)$ time by Larmore [L]. This paper contains a simple $O(nL)$-time algorithm, which we call the *Package-Merge* algorithm.

The Alphabetic Coding problem can be solved in $O(n\log n)$ time by the Hu-Tucker and Garsia-Wachs algorithms [HuTu] [GaWa]. The restricted-length version is solved in $O(n^3L)$ time by Garey [Ga], and in $O(n^2L)$ time by Itai and Wessler, independently [I] [W]. In this paper we present an $O(nL\log n)$-time algorithm.

2. The Package-Merge algorithm

In this section, we introduce the Coin Collector's problem, which is a version of the Knapsack problem, and the Package-Merge algorithm which solves it in linear time. We then show how an instance of the Length-Limited Coding problem with parameters n and L can be reduced to an instance of the Coin-Collector's problem of size nL. The Package-Merge algorithm thus solves the Length-Limited Coding problem in $O(nL)$ time.

The Coin Collector's problem. A coin collector has m coins of various denominations (face values) and various numismatic values. The country he lives in has binary coinage, and so the denomination of each coin is an integral power of 2. The collector wishes to spend Q dollars (Q is an integer) 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 Q?

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

(a) A set I of m items, each of which has a *width* 2^{-d} $(d \in N)$ and a non-negative *weight.* (Think of *width* as being face value of a coin, and *weight* as being numismatic value.)

(b) An integer Q.

A *solution* to such an instance is a subset S of I of minimal weight whose widths sum to exactly Q.

The general Knapsack problem is NP-complete. However, by adding the restriction that the widths are of the form 2^{-d}, the resulting Coin-Collector's problem can be solved efficiently.

The Package-Merge algorithm. The Package-Merge algorithm maintains lists of "packages." Each package is a set of items whose total width is 2^{-d} for some $d \in N$, and each list consists of packages of all the same width, sorted in order of increasing weight. Initially, each item is a package by itself, and each list is the set of all items of a given width, sorted by weight. Each step of the algorithm combines the two smallest weight items of the smallest remaining width to form a single package of the next larger width, which is then inserted into the appropriate list. An odd package of width less than 1 is discarded. Finally, there is only one list, consisting of packages of width 1, sorted by weight. S is then taken to be the union of the first Q of these. Figure 2 illustrates the algorithm.

PACKAGE-MERGE ALGORITHM

```
Let $D$ be such that $2^{-D}$ is the smallest width of any item
$A_d$ is the list of items of width $2^{-d}$, sorted by weight
for $d \leftarrow D$ downto 1 loop
    if $A_d$ has odd length then discard its heaviest item
    Combine adjacent pairs of $A_d$ (each element has width $2^{-d}$)
        to form a list $B_d$ of packages of width $2^{-d+1}$
    Merge $B_d$ into $A_{d-1}$
end loop
Let $S$ be the union of the $Q$ least weight items of $A_0$
```

Correctness. We prove that the Package-Merge algorithm is correct by induction. If all items have width 1, correctness is trivial. Otherwise, since Q is an integer, S must contain an even number of items of the smallest width. If there is only one such item, discarding it will not affect the solution. If there are two or more items of smallest width, consider the two of smallest weight. Either both or neither of these will be in S, so combining them into a single item of larger width will not affect the solution. In either case the instance is reduced to an instance with fewer items.

Time analysis. Packaging the pairs of elements of a list takes time which is linear in the length of the list while merging two sorted lists takes time which is linear in the sum of the lengths of the lists. We begin an amortization argument by placing three credits on each original item. Invariably, there are three credits on each item of any list which consists solely of original items, two credits on each item of any list formed as a result of a merge, and three credits on each item of list B_d. Each packaging step combines two items (from A_d) which have two or three credits each into one item (placed on B_d) which has three credits, allowing at least one credit to pay 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. Therefore, the Package-Merge algorithm on m items takes linear time assuming that the items in the lists A_d are presorted by width, then by weight within each width (this will be the case in our application). Otherwise, some sorting algorithm must be applied first and the algorithm will require $O(m \log m)$ time.

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

We note that the algorithm can be modified to cover the case where Q is not an integer. Q must be some diadic rational, otherwise no solution is possible. Write Q as an integer, plus a sum of distinct powers of 2 (for example, if $Q = 3.625$, write $3 + 2^{-1} + 2^{-3}$). During the iteration of the algorithm indexed by d, if that power of 2 occurs in the sum then pluck the smallest package of width 2^{-d} from A_d before executing any other step. These plucked packages will be included in S, as will the smallest $\lfloor Q \rfloor$ packages of width 1.

The reduction (Coding problem $\rightarrow$ Coin Collector's problem). Let $\phi_1, \dots \phi_n$ be the list of frequencies (sorted into non-increasing order) for an instance of the (non-alphabetic) Length-Limited Coding problem, and let L be the maximum permitted length. We define a *node* to be an ordered pair $(i,l) \in [1,n]\times[1,L]$. Node (i,l) has *index* i, *weight* ϕ_i, *level* l, and *width* 2^{-l}. The weight (or width) of a set of nodes is the sum of the weights (or widths) of its members. If T is a binary tree, define $nodeset(T) = \{(i,l) \mid 1 \le l \le l_i\}$ where l_i is the depth of the i^{th} leaf of T. Thus $\overline{T} = \sum_{i=1}^{n} \phi_i l_i$ is exactly the weight of $nodeset(T)$, and it can be shown (by induction on n) that the width of $nodeset(T)$ is $n-1$.

The reduction to the Coin Collector's problem is to let each node be an item. Use the Package-Merge algorithm to find a set $A \subseteq [1,n]\times[1,L]$ of width $n-1$ of minimum weight. If ties (for equal weight nodes) are broken in favor of nodes of smaller index, A will be the nodeset of an optimal height-restricted tree. See Figure 3.

Time and space requirements. The Package-Merge algorithm for the Length-Limited Coding problem has a straightforward implementation that requires $O(nL)$ time and space. It can be run in linear space, using a trick similar to that used in [Hi]. The effect is to multiply the time requirements by a small constant factor (~ 2).

3. The Alphabetic Package-Merge algorithm

In this section, we give an algorithm that solves an instance of the Length-Limited Alphabetic Coding problem in time $O(nL\log n)$.

We are given weights $\phi_1, \ldots \phi_n$, not necessarily sorted, and an integer $L \ge \log_2 n$. The desired output is a binary tree T for which $\overline{T} = \sum_{i=1}^{n} \phi_i l_i$ is minimized, where l_i is the depth of the i^{th} leaf of T.

As in the previous section, we say that $(i,l) \in [1,n]\times[1,L]$ is a node of index i, of weight ϕ_i and of level l. If T is a binary tree, $nodeset(T) = \{(i,l) \mid 1 \le l \le l_i\}$ where l_i is the depth of the i^{th} leaf of T.

The weight of a set of nodes is the sum of the weights of its members. We define the index of a set of nodes to be 0.5 more than the smallest index of any of its member nodes.

As in the non-alphabetic version, this algorithm builds optimal packages at iteratively higher levels to construct the nodeset of an optimal tree. There are $L+1$ lists of nodesets (packages): $P_0, P_1, \ldots, P_L$. Initially, P_0 is empty and each list P_d ($d>0$) contains n singleton nodesets, each containing one node at level d. The algorithm proceeds in stages, indexed from L down to 1. At stage d, the algorithm iterates combining two packages at level d to form a package at level $d-1$. As in the Hu-Tucker algorithm [HuTu], packages can be combined only if they are a "compatible pair" and they have the least total weight of any compatible pair. A pair of nodesets of P_d, p_1 and p_2, whose indices are i_1 and i_2 are defined to be *compatible* if no *singleton* nodeset of P_d has index strictly between i_1 and i_2. This definition is essentially the same as that in [HuTu].

After all L stages have been executed, P_0 has $n-1$ members, each of which is a nodeset. The union of these is *nodeset*(T) for the optimal tree T.

In high-level form, the algorithm is as follows:

ALPHABETIC PACKAGE-MERGE ALGORITHM

```
$P_0 \leftarrow \emptyset$ (the empty list)
for $d \leftarrow 1$ to $L$
        $P_d \leftarrow \{(1,d)\}, \{(2,d)\}, \ldots, \{(n,d)\}$
for $d \leftarrow L$ downto 1 loop
        while $|P_d| \geq 2$ loop
                $\{p_1,p_2\} \Leftarrow P_d$ (Delete least weight compatible pair)
                $p \leftarrow p_1 \cup p_2$ (Package step)
                $P_{d-1} \Leftarrow p$ (Merge step)
        end while
end for
$S \leftarrow \cup P_0$
$T^{\text{opt}} \leftarrow$ that tree whose nodeset is $S$
```

Implementation. The Alphabetic Package-Merge algorithm can be implemented to run in $O(n \log n)$ time for each level d (and hence in $O(nL \log n)$ time altogether) using mergeable priority queues (see [AHU] for example).

If $i_1, \ldots i_m$ are the indices of the remaining singletons in P_d, let C_r be the set of remaining packages in P_d whose indices lie in the interval $[i_r, i_{r+1}]$, where we let $i_0 = 0$ and $i_{m+1} = n+1$. Singleton packages (other then possibly the first and/or last) will be in two of the $\{C_r\}$ while all other packages will be in exactly one of the $\{C_r\}$. Two packages are compatible if and only if they both lie in the same C_r. Each of the $\{C_r\}$ is represented as a mergeable priority queue, with the member packages prioritized by weight, and the sum of the weights of the lightest two members is maintained. The collection of these minimal sums is maintained as a heap, the minimum sum being at the top of the heap. Each time the least weight compatible pair is taken, the one or both copies of each package must be removed from the structure, and possibly some adjacent sets must be merged. Thus, each package-merge iteration can be performed in $O(\log n)$ time.

Any P_d can have at most $2n$ packages and so the inner loop of the algorithm iterates at most nL times. The time for the last step is also $O(nL)$. Therefore, the time for the entire algorithm is $O(nL \log n)$. The space required for the data structure is $O(nL)$.

References

[AHU] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

[GaWa] A.M. Garsia and M.L. Wachs, A New algorithm for minimal binary search trees, *SIAM J Comp* 6 (1977) pp. 622-642.

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

[Hi] Hirschberg, D.S., A linear space algorithm for computing maximal common subsequences, *Comm ACM 18* 6 (1975), pp. 341-343.

[HuTa] T.C. Hu and K.C. Tan, Path length of binary search trees, *SIAM J Applied Math* 22 (1972) pp. 225-234.

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

[Hu] T.C. Hu, *Combinatorial Algorithms*, Addison Wesley (1982).

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

[I] Itai, Alon, Optimal alphabetic trees, *SIAM Journal of Computing* 5 (1976), pp. 9-18

[L] L.L. Larmore, Height-restricted optimal binary trees, *SIAM Jour. on Comp.* 16 (1987), pp. 1115-1123.

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

$a \rightarrow 0$
$b \rightarrow 101$
$c \rightarrow 11$
$d \rightarrow 100$

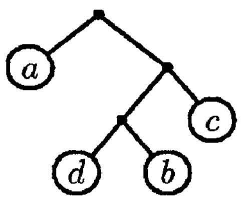

Figure 1. The binary tree corresponding to a prefix-free binary code.

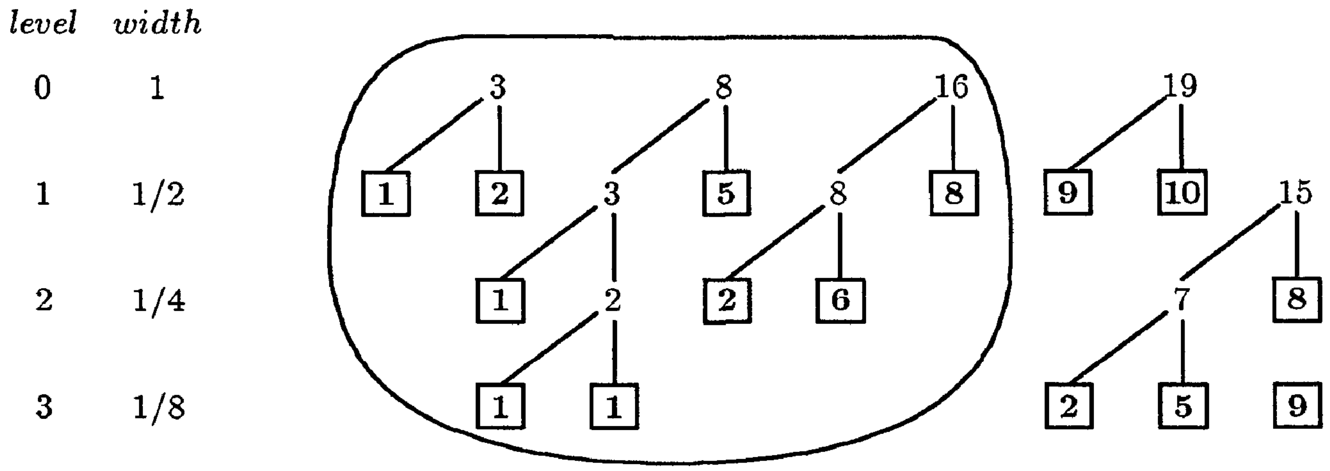

Figure 2. The Package-Merge algorithm for the Coin Collector's problem. Original items are singleton packages, indicated by square nodes. The set S of width Q=3 of minimum weight is represented by the square nodes in the enclosed region. The algorithm is worked from the bottom of the diagram up.

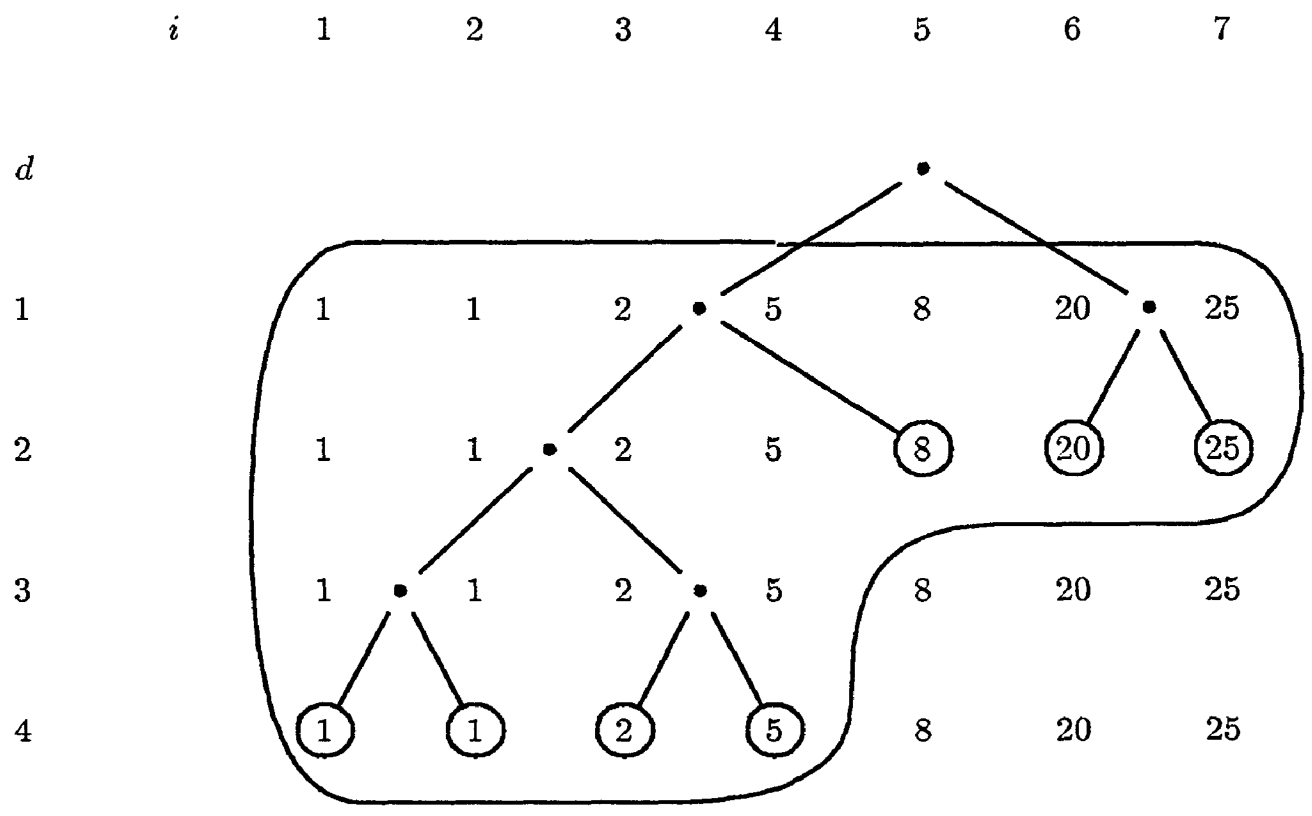

Figure 3. The minimum weight nodeset of width $n-1$
and the resulting optimal tree.
Each node is shown as a number which is its weight, ϕ_i.
Circled nodes indicate tree leaves.