Poster

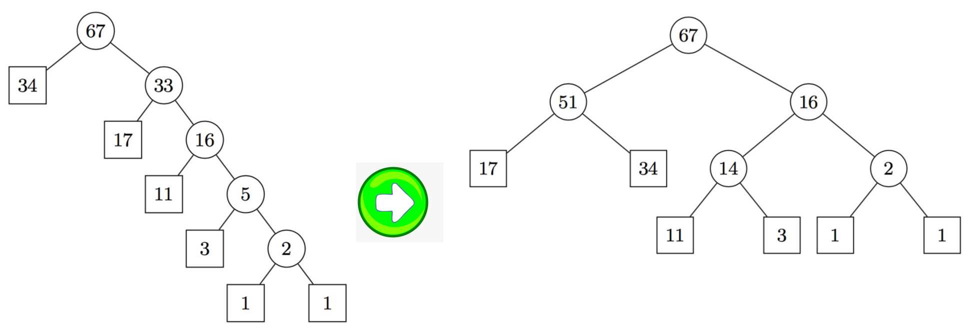
Decode-efficient prefix codes for hierarchical memory models

Model/Introduction

Model. Our scratchpad model is similar to the two-level hierarchical model proposed in [1] comprising of a limited size fast memory (scratchpad memory) and an unlimited size main memory. The cost of accessing a location in the scratchpad (main memory resp.) is 1 unit (q units resp.). Decoding the input is typically done by traversing the stored prefix tree. We consider the class of algorithms that store nodes of the prefix tree in the scratchpad – one prefix tree node in one scratchpad addressable memory location.

Problem Definition. Consider an alphabet C. For each character c in C, let $f(c)$ denote the frequency of c. Given a prefix tree T corresponding to a prefix code P for C, let $d(c)$ denote the depth of the leaf corresponding to the encoding of c in the tree T. The average code length of the encoding is given by $\ell(T) = \sum_{c \in C} f(c) \cdot d(c)$ Given a constant m (scratchpad size), we define the decode time of the encoding to be $\mathbf{dec}(T, m) = \sum_{c \in C} f(c) + q \cdot \sum_{c \in C: d(c) > \log(m)} f(c) \cdot (d(c) - \log(m))$. Given constants m and L, our goal is to find a prefix tree, T, that minimizes $\mathbf{dec}(T, m)$ subject to $\ell(T) \leq L$.

Bad example and solution (size 3 scratch pad)



Code Length is
5·1+5·1+4·3+3·11+2·17+1·34= **123**
Decode Time is (1+2q)·1+(1+2q)·1+(1+q)·3+1·11+1·17+1·34=**67+7q**

Code Length is
3·1+3·1+3·3+3·11+2·34+2·17=**150**
better Decode Time of
1·1+1·1+1·3+1·11+1·17+1·34= **67**

Approach

We present an efficient algorithm that solves the above mentioned problem optimally for a given alphabet C, a threshold codelength parameter L and a scratchpad size parameter m.

- This is based on a property of the forest outside the scratch pad. We call these as a Huffman Forest which is an intermediate step in the construction of optimal tree using huffman algorithm.
- We solve for the position of nodes which remain in the scratchpad using a Dynamic programming algorithm.

The running time of the algorithm is polynomial in the size of the fast memory (poly(m)) and near linear in the size of the alphabet (|C|log|C|).