5 Greedy Algorithms

The philosophy of being greedy is shortsightedness. Always go for the seemingly best next thing, always optimize the presence, without any regard for the future, and never change your mind about the past. The greedy paradigm is typically applied to optimization problems. In this section, we first consider a scheduling problem and second the construction of optimal codes.

A scheduling problem. Consider a set of activities $1, 2, \ldots, n$. Activity i starts at time s_i and finishes at time $f_i > s_i$. Two activities i and j *overlap* if $[s_i, f_i] \cap [s_j, f_j] \neq \emptyset$. The objective is to select a maximum number of pairwise non-overlapping activities. An example is shown in Figure 6. The largest number of activities can be scheduled by choosing activities with early finish times first. We first sort and reindex such that $i < j$ implies $f_i \leq f_j$.

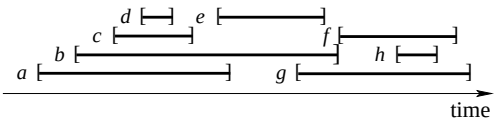

Figure 6: A best schedule is c, e, f, but there are also others of the same size.

$S = \{1\}$; $last = 1$;
`for` $i = 2$ `to` n `do`
  `if` $f_{last} < s_i$ `then`
    $S = S \cup \{i\}$; $last = i$
  `endif`
`endfor`.

The running time is O($n \log n$) for sorting plus O(n) for the greedy collection of activities.

It is often difficult to determine how close to the optimum the solutions found by a greedy algorithm really are. However, for the above scheduling problem the greedy algorithm always finds an optimum. For the proof let $1 = i_1 < i_2 < \ldots < i_k$ be the greedy schedule constructed by the algorithm. Let $j_1 < j_2 < \ldots < j_\ell$ be any other feasible schedule. Since $i_1 = 1$ has the earliest finish time of any activity, we have $f_{i_1} \leq f_{j_1}$. We can therefore add i_1 to the feasible schedule and remove at most one activity, namely j_1. Among the activities that do not overlap i_1, i_2 has the earliest finish time, hence $f_{i_2} \leq f_{j_2}$. We can again add i_2 to the feasible schedule and remove at most one activity, namely j_2 (or possibly j_1 if it was not removed before). Eventually, we replace the entire feasible schedule by the greedy schedule without decreasing the number of activities. Since we could have started with a maximum feasible schedule, we conclude that the greedy schedule is also maximum.

Binary codes. Next we consider the problem of encoding a text using a string of 0s and 1s. A *binary code* maps each letter in the alphabet of the text to a unique string of 0s and 1s. Suppose for example that the letter 't' is encoded as '001', 'h' is encoded as '101', and 'e' is encoded as '01'. Then the word 'the' would be encoded as the concatenation of codewords: '00110101'. This particular encoding is unambiguous because the code is *prefix-free*: no codeword is prefix of another codeword. There is a one-to-one correspondence between prefix-free binary codes and binary trees where each leaf is a letter and the corresponding codeword is the path from the root to that leaf. Figure 7 illustrates the correspondence for the above 3-letter code. Being prefix-free corresponds to leaves not having children. The tree in Figure 7 is not full because three of its internal nodes have only one child. This is an indication of waste. The code can be improved by replacing each node with one child by its child. This changes the above code to '00' for 't', '1' for 'h', and '01' for 'e'.

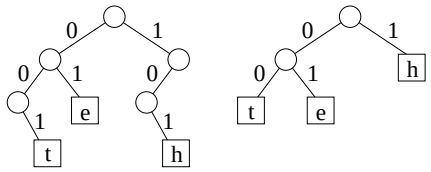

Figure 7: Letters correspond to leaves and codewords correspond to maximal paths. A left edge is read as '0' and a right edge as '1'. The tree to the right is full and improves the code.

Huffman trees. Let w_i be the frequency of the letter c_i in the given text. It will be convenient to refer to w_i as the *weight* of c_i or of its external node. To get an efficient code, we choose short codewords for common letters. Suppose δ_i is the length of the codeword for c_i. Then the number of bits for encoding the entire text is

$$P = \sum_i w_i \cdot \delta_i.$$

Since δ_i is the depth of the leaf c_i, P is also known as the *weighted external path length* of the corresponding tree.

The *Huffman tree* for the c_i minimizes the weighted external path length. To construct this tree, we start with n nodes, one for each letter. At each stage of the algorithm, we greedily pick the two nodes with smallest weights and make them the children of a new node with weight equal to the sum of two weights. We repeat until only one node remains. The resulting tree for a collection of nine letters with displayed weights is shown in Figure 8. Ties that

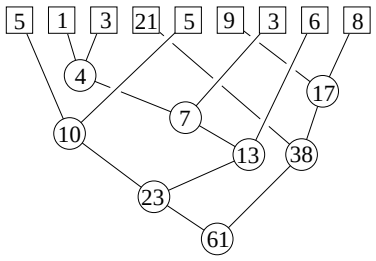

Figure 8: The numbers in the external nodes (squares) are the weights of the corresponding letters, and the ones in the internal nodes (circles) are the weights of these nodes. The Huffman tree is full by construction.

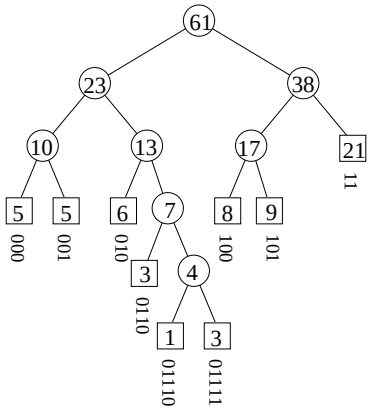

Figure 9: The weighted external path length is $15 + 15 + 18 + 12 + 5 + 15 + 24 + 27 + 42 = 173$.

arise during the algorithm are broken arbitrarily. We redraw the tree and order the children of a node as left and right child arbitrarily, as shown in Figure 9.

The algorithm works with a collection N of nodes which are the roots of the trees constructed so far. Initially, each leaf is a tree by itself. We denote the weight of a node by $w(\mu)$ and use a function EXTRACTMIN that returns the node with the smallest weight and, at the same time, removes this node from the collection.

```
Tree HUFFMAN
  loop μ = EXTRACTMIN(N);
       if N = ∅ then return μ endif;
       ν = EXTRACTMIN(N);
       create node κ with children μ and ν
          and weight w(κ) = w(μ) + w(ν);
       add κ to N
  forever.
```

Straightforward implementations use an array or a linked list and take time O(n) for each operation involving N. There are fewer than $2n$ extractions of the minimum and fewer than n additions, which implies that the total running time is O(n^2). We will see later that there are better ways to implement N leading to running time O($n \log n$).

An inequality. We prepare the proof that the Huffman tree indeed minimizes the weighted external path length. Let T be a full binary tree with weighted external path length $P(T)$. Let $\Lambda(T)$ be the set of leaves and let μ and ν be any two leaves with smallest weights. Then we can construct a new tree T' with

(1) set of leaves $\Lambda(T') = (\Lambda(T) - \{\mu, \nu\}) \dot\cup \{\kappa\}$,

(2) $w(\kappa) = w(\mu) + w(\nu)$,

(3) $P(T') \leq P(T) - w(\mu) - w(\nu)$, with equality if μ and ν are siblings.

We now argue that T' really exists. If μ and ν are siblings then we construct T' from T by removing μ and ν and declaring their parent, κ, as the new leaf. Then

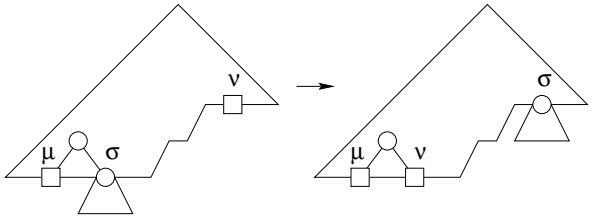

Figure 10: The increase in the depth of ν is compensated by the decrease in depth of the leaves in the subtree of σ.

$$\begin{aligned} P(T') &= P(T) - w(\mu)\delta - w(\nu)\delta + w(\kappa)(\delta - 1) \\ &= P(T) - w(\mu) - w(\nu), \end{aligned}$$

where $\delta = \delta(\mu) = \delta(\nu) = \delta(\kappa) + 1$ is the common depth of μ and ν. Otherwise, assume $\delta(\mu) \geq \delta(\nu)$ and let σ be

the sibling of μ, which may or may not be a leaf. Exchange ν and σ. Since the length of the path from the root to σ is at least as long as the path to μ, the weighted external path length can only decrease; see Figure 10. Then do the same as in the other case.

Proof of optimality. The optimality of the Huffman tree can now be proved by induction.

HUFFMAN TREE THEOREM. Let T be the Huffman tree and X another tree with the same set of leaves and weights. Then $P(T) \leq P(X)$.

PROOF. If there are only two leaves then the claim is obvious. Otherwise, let μ and ν be the two leaves selected by the algorithm. Construct trees T' and X' with

$$\begin{aligned} P(T') &= P(T) - w(\mu) - w(\nu), \\ P(X') &\leq P(X) - w(\mu) - w(\nu). \end{aligned}$$

T' is the Huffman tree for $n-1$ leaves so we can use the inductive assumption and get $P(T') \leq P(X')$. It follows that

$$\begin{aligned} P(T) &= P(T') + w(\mu) + w(\nu) \\ &\leq P(X') + w(\mu) + w(\nu) \\ &\leq P(X). \end{aligned}$$

□

Huffman codes are binary codes that correspond to Huffman trees as described. They are commonly used to compress text and other information. Although Huffman codes are optimal in the sense defined above, there are other codes that are also sensitive to the frequency of sequences of letters and this way outperform Huffman codes for general text.

Summary. The greedy algorithm for constructing Huffman trees works bottom-up by stepwise merging, rather than top-down by stepwise partitioning. If we run the greedy algorithm backwards, it becomes very similar to dynamic programming, except that it pursues only one of many possible partitions. Often this implies that it leads to suboptimal solutions. Nevertheless, there are problems that exhibit enough structure that the greedy algorithm succeeds in finding an optimum, and the scheduling and coding problems described above are two such examples.