Greedy Algorithms

(read Section 16 on Greedy Algorithms in Cormen, Leiserson, Rivest, Stein)

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 study the construction of optimal codes.

A scheduling problem. Consider a set of activities, \( S = \{1, 2, \ldots, n\} \). Activity \( i \) has start time \( s_i \) and finish time \( f_i \). 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 46. The largest number of activities can be scheduled by choosing activities with early finishing times first. We first sort and reindex such that \( 0/1 \) implies \( 324 \).

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 we can show that 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 \( j_1 < j_2 < \ldots < j_k \) 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 \( i_1 \). 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 \( i_2 \) (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 0’s and 1’s. A binary code maps each letter in the alphabet of the text to a unique string of 0’s and 1’s. 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: each leaf is a letter and the corresponding codeword is the path from the root to that leaf. Figure 47 illustrates the correspondence for the above 3-letter code. Being prefix-free corresponds to leaves not having children. The tree in Figure 47 is not full because three of its internal nodes have only one child. This is wasteful. The code can be improved by replacing each node that has only one child by its child. This changes the above code.
Huffman trees. In the situation we consider, each letter $c_i$ has a weight or frequency $f_i$ measuring, for example, the relative occurrence in a typical English text. To get an efficient code, we choose short codewords for common letters. Suppose $\delta_i$ is the length of the codeword for $c_i$. Then the number of bits for encoding the entire text is $P = \sum_i f_i \cdot \delta_i$. Since $\delta_i$ is also the depth of the leaf $c_i$, $P$ is the weighted external path length of the corresponding tree.

A Huffman tree for $n$ letters minimizes the weighted external path length. To construct such a 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 various frequencies is shown in Figure 48. Ties that 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 49.

Proof of optimality. 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 $\mu$ and $\nu$ be any two leaves with smallest weights. Then we can construct a new tree $T'$ with

1. set of leaves $\Lambda(T') = (\Lambda(T) - \{\mu, \nu\}) \cup \{\kappa\}$,
2. $f(\kappa) = f(\mu) + f(\nu)$,
3. $P(T') \leq P(T) - f(\mu) - f(\nu)$, with equality if $\mu$ and $\nu$ are siblings.

We now argue that $T'$ really exists. If $\mu$ and $\nu$ are siblings then we construct $T'$ from $T$ by removing $\mu$ and $\nu$ and

![Figure 47](image1)

Figure 47: 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.

![Figure 48](image2)

Figure 48: The numbers in the external square nodes are frequencies of the corresponding letters, and the ones in the round internal nodes are the weights of these nodes. The Huffman tree is full by construction.

![Figure 49](image3)

Figure 49: The weighted external path length is $15 + 15 + 18 + 12 + 5 + 15 + 24 + 27 + 42 = 173$.

The construction of the Huffman tree is facilitated by a priority queue that maintains the set 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 $\mu$ by $f(\mu)$.
declaring their parent, $\kappa$, as the new leaf. Then
\[
P(T') = P(T) - \delta f(\mu) + f(\nu) + (\delta - 1) f(\kappa)
\]
\[
= P(T) - f(\mu) - f(\nu),
\]
where $\delta = \delta(\mu) = \delta(\nu) = \delta(\kappa) + 1$ is the common depth of $\mu$ and $\nu$. Otherwise, assume $\delta(\mu) \geq \delta(\nu)$ and let $\sigma$ be the sibling of $\mu$, which may or may not be a leaf. Exchange $\nu$ and $\sigma$. As illustrated in Figure 50, the weighted external path length can only decrease. Then do the same as in the other case.

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 $\mu$ and $\nu$ be the two leaves selected by the algorithm. Construct trees $T'$ and $X'$ with
\[
P(T') = P(T) - f(\mu) - f(\nu),
\]
\[
P(X') \leq P(X) - f(\mu) - f(\nu).
\]
$T'$ 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
\[
P(T) = P(T') + f(\mu) + f(\nu)
\]
\[
\leq P(X') + f(\mu) + f(\nu)
\]
\[
\leq P(X).
\]

Huffman codes are binary codes that correspond to Huffman trees as described. They are commonly used to compress text and other information. Although Huffman

**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 to find an optimum, and the scheduling and coding problems described above are two such examples.