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 $f_i \leq f_j$.

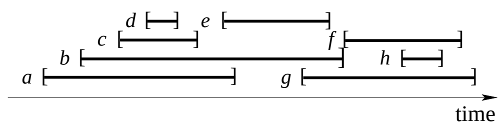

Figure 6: A best schedule is c, e, f, but there are also others of the same size.

```
$S = \{1\}; last = 1;$
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'.

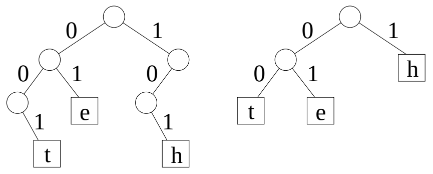

Figure 7: Letters correspond to leaves and codewords correspond to maximal paths. A left edge is read as '0' and a right edge as '1'. The tree to the right is full and improves the code.

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.