More Greed

Greedy Algorithm for Activity Selection

We have a set of activities to choose from, but some of them overlap. We want to choose the largest possible set of non-overlapping activities.

A greedy algorithm always starts by sorting the objects. In class we experimented with different criteria for sorting:

1. Sort the activities by start time. Then repeatedly choose the next activity that doesn't overlap with the ones already chosen.

This algorithm fails on this example

<table>
<thead>
<tr>
<th>Task</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Time</td>
<td>8:00</td>
<td>8:01</td>
<td>8:05</td>
</tr>
<tr>
<td>Finish Time</td>
<td>9:00</td>
<td>8:02</td>
<td>8:06</td>
</tr>
</tbody>
</table>

2. Sort the activities by length, shortest first. Then repeatedly choose the next activity that doesn't overlap with the ones already chosen.

This algorithm fails on this example

<table>
<thead>
<tr>
<th>Task</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Time</td>
<td>8:58</td>
<td>8:55</td>
<td>9:01</td>
</tr>
<tr>
<td>Finish Time</td>
<td>9:02</td>
<td>8:59</td>
<td>9:05</td>
</tr>
</tbody>
</table>
3. Sort the activities by length, longest first. Then repeatedly choose the next activity that doesn't overlap with the ones already chosen.

This algorithm fails on this example

<table>
<thead>
<tr>
<th>Task</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Time</td>
<td>8:00</td>
<td>8:01</td>
<td>8:05</td>
</tr>
<tr>
<td>Finish Time</td>
<td>9:00</td>
<td>8:02</td>
<td>8:06</td>
</tr>
</tbody>
</table>

None of these work ... but undaunted by our repeated failures, we finally hit upon another alternative:

4. Sort the activities by finish time, earliest first. Then repeatedly choose the next activity that doesn't overlap with the ones already chosen.

Proof of correctness:

Base case: Let \( n = 0 \). Clearly the algorithm will choose the only activity, thus finding the optimal solution.

Assume the algorithm always finds the optimal solution when there are \( \leq n \) activities, where \( n \) is some integer \( \geq 0 \).

Let the number of activities be \( n+1 \). Apply the algorithm (sort by finish time, and select as above). Let the Greedy Algorithm solution be \( A = \{a_1, a_2, ..., a_t\} \). We need to show that there is an optimal solution containing \( a_1 \). Let \( O \) be any optimal solution. Sort the elements of \( O \) by finish time. Let \( O = \{o_1, o_2, ..., o_s\} \). We know that \( \text{start_time}(o_2) \geq \text{finish_time}(o_1) \geq \text{finish_time}(a_1) \). Thus \( O' = \{a_1, o_2, ..., o_s\} \) is a feasible solution, and \( |O'| = |O| \), so \( O' \) is an optimal solution that contains \( a_1 \).

By the inductive assumption, \( \{a_2, a_3, ..., a_t\} \) is an optimal solution to the problem of choosing presentations that start no earlier than \( \text{finish_time}(a_1) \). Let \( O^* \) be an optimal solution that starts with \( a_1 \), \( O^* = \{a_1, o^*_2, ..., o^*_k\} \). Observe that \( \{o^*_2, o^*_3, ..., o^*_k\} \) is also a solution to the problem of choosing presentations that start no earlier than \( \text{finish_time}(a_1) \). Thus \( t \leq k \).

Now if \( t < k \), \( |A| < |O^*| \), which contradicts the optimality of \( S \). Thus \( t = k \), and \( |A| = |O^*| \). Thus \( A \) is an optimal solution, and the Greedy Algorithm finds the optimal solution whenever there are \( n+1 \) activities.

Therefore the Greedy Algorithm finds the optimal solution for all sets of activities.
**Huffman Coding**

Data compression is an essential component of modern computing and communications. The Huffman Coding algorithm, discovered in 1952, is still in use as the “back-end” for modern forms of compression such as JPEG and MP3. Huffman Coding is provably optimal for lossless compression in which each encoded symbol is represented by a fixed bitstring.

Let $S$ be a set of symbols (eg letters, number, etc), with $S = \{s_1, s_2, \ldots, s_n\}$

We will define a Code for $S$ to be a set of unique bitstrings, one for each symbol in $S$, so that $C = \{c_1, c_2, \ldots, c_n\}$. We will use the notation $C(s_i) = c_i$ to indicate that symbol $s_i$ is encoded by $c_i$.

Given a string $D$ containing only symbols in $S$, we encode $D$ with $C$ by replacing each symbol by its associated bitstring.

Example: Let $S = \{a, b, c, d\}$, $C = \{00, 01, 10, 11\}$ and $D = “acccdba”$, $D$ would be encoded by the bitstring 00101010110100

Our interest is in **minimizing** the length of the encoding of $D$.

In the example above, each bitstring in $C$ had the same length. However, this is not a necessary requirement for codes.

Example: Let $C_2 = \{0, 00, 100, 101\}$ - each symbol in $S$ has an unique bitstring associated with it .... but this won’t work! If $D_1 = “aac”$ and $D_2 = “bc”$, they would both be encoded by 00100 which means that the task of correctly decoding would be impossible.

So if we want to use codes in which the bitstrings are of different lengths (and we do!) then we have to require that the codes satisfy the **Prefix Property**: no bitstring in the code can be the first part (ie the prefix) of any other bitstring in the code.

Example: Let $C_2 = \{0, 10, 110, 111\}$, and $D_1$ and $D_2$ as above. Now $D_1$ encodes as 00110 and $D_2$ encodes as 10110

If a code satisfies the Prefix Property, then decoding an encoded string is very easy: just start at the first bit and keep going until you have completed one of the bitstrings in the code. Then start over from the next bit, etc. Because of the prefix property there is never any ambiguity.
For example, decoding $D_1$ goes like this:

First bit is 0 – that’s “a”
Second bit is 0 – that’s “a”
Third bit is 1 – don’t know yet
Fourth bit is 1 – so far we have 11, but we still don’t know what the letter is
Fifth bit is 0 – so we have 110, and that’s “c”

Note that codes in which all bitstrings have the same length automatically satisfy the Prefix Property.

Tracing the decoding process for a prefix-property code is exactly equivalent to traversing a binary tree in which the edges are labelled 0 and 1. The symbols in $S$ are placed at the leaves of the tree, and the sequence of 0s and 1s that lead to $s_i$ are exactly $c_i$. For any prefix-property code there is exactly one binary tree with the edges labelled 0 and 1 and with $n$ leaves labelled with $S$, and for any binary tree with the edges labelled 0 and 1 and with $n$ leaves labelled with $S$, there is exactly one prefix-property code. We will use this equivalence between prefix-property codes and binary trees when we prove the optimality of Huffman Coding.

Now we can define an objective function for our problem: for a given string $D$ and code $C$, let $F(C) =$ the total number of bits in the encoded form of $D$.

We can compute $F(C)$ if we know the frequency of each $s_i$ in $D$. Let $f(s_i) =$ the number of times $s_i$ occurs in $D$. Then

$$F(C) = \sum_{s_i \in S} f(s_i) \cdot |C(s_i)|$$

Finally, let’s look at the Huffman Coding Algorithm:

initialize $C(s_i)$ to “” \quad \forall i
assume $f(s_i)$ has been computed for all $s_i$
let $W = S$ \quad # $W$ is a working copy of $S$
let $R(w_i) = \{s_i\}$ \quad \forall i
# Each element of $W$ starts out representing
# a single symbol. As the algorithm
# progresses the symbols are grouped
# together so elements in $W$ represent
# groups of symbols
while $|W| > 1$:
    let $w_i$ and $w_j$ be the symbols in $W$ with the lowest frequencies
    prepend “0” to the codes for all symbols in $R(w_i)$
    prepend “1” to the code for all symbols in $R(w_j)$
    create a new symbol $w_{i,j}$ with $f(w_{i,j}) = f(w_i) + f(w_j)$, and let
    $R(w_{i,j}) = R(w_i) \cup R(w_j)$
    $W = W - \{w_i, w_j\} + \{ w_{i,j} \}$
Expressed that way, the algorithm is well-defined but potentially confusing. It’s much easier to understand if we think of building the binary tree that represents the code.

1. Assume \( f(s_i) \) has been computed for all \( s_i \).
2. Create \( V \) = a set of vertices, labelled with the elements of \( S \).
3. While \(|V| > 1\):
   1. Let \( v_i \) and \( v_j \) be the vertices in \( V \) with the lowest frequencies.
   2. Create a new vertex \( v_{i,j} \).
   3. Make \( v_i \) the left child of \( v_{i,j} \) and label the edge “0”.
   4. Make \( v_j \) the right child of \( v_{i,j} \) and label the edge “1”.
   5. Let \( f(v_{i,j}) = f(v_i) + f(v_j) \).
   6. \( V = V - \{v_i, v_j\} + \{v_{i,j}\} \).

The codes for all the symbols in \( S \) can be found by tracing down from the root of the tree to the leaves.

**Example:** Let \( S = \{a,b,c,d,e\} \) and let \( D \) be a string in which the symbols occur with these frequencies:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>12</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

We start with five vertices:

\[
\begin{align*}
a:12 & \quad b:10 \\
c:9 & \quad d:8 \\
e:7 &
\end{align*}
\]

Since \( d \) and \( e \) have the lowest frequencies, we give them a parent, labelled “\( d,e \)”:

\[
\begin{align*}
a:12 & \quad b:10 \\
c:9 & \quad d \quad e \\
\end{align*}
\]

\[
\begin{align*}
\downarrow & \quad \downarrow \\
d,e : 15 &
\end{align*}
\]

Now \( b \) and \( c \) have the lowest frequencies:

\[
\begin{align*}
a:12 & \quad b \quad c \quad d \quad e \\
\downarrow & \quad \downarrow \\
\downarrow & \quad \downarrow \\
b,c : 19 & \quad d,e : 15
\end{align*}
\]
Now a and d,e have the lowest frequencies:

```
  a   d   e   b   c
  \   \   \   /   /
  \    d,e     b,c : 19
  \          /   /
     a,d,e: 27
```

(I moved things around a bit to make the drawing easier)
Finally a,d,e and b,c have the lowest frequencies:

```
  a   d   e   b   c
  \   \   \   /   /
  \    d,e     b,c
  \          /   /
     a,d,e   /   /
        a,d,e,b,c: 46
```

Assuming (without loss of generality) that edges going up to the left are “0” and edges going up the right are “1”, we get the following code C:

```
a: 00
b: 10
c: 11
d: 010
e: 011
```

For this code, we can see that \( F(C) = 12*2 + 10*2 + 9*2 + 8*3 + 7*3 = 107 \)

If we had used a fixed-length code we would have needed at least 3 bits for each symbol, and the total encoded length would be at least 138.

So we can see that in this instance Huffman encoding paid off. The question is, why should we believe that it always will?
Proof of Optimality for Huffman Coding

Theorem: Let C be the code created by the Huffman Coding algorithm to encode a string D. Then \( F(C) \leq F(X) \) where X is any other prefix-property code for D.

Proof: as with virtually all greedy algorithms, we use induction.

Base case: If \(|S| = 2\), there is only one solution and the algorithm finds it: represent one symbol by “0” and the other by “1”

IH: Assume that for all instances of the problem with \( n \) symbols (where \( n \) is some integer with \( n \geq 1 \)) the algorithm finds an optimal solution.

Suppose \(|S| = n+1\)

Let A be the code selected by the algorithm. We know that \( s_i \) and \( s_j \), the two symbols with the lowest frequencies, must be siblings in the tree for code A

Let O be an optimal code for this problem (ie. \( F(O) \leq F(X) \) \( \forall \) other codes X)

I claim that Without Loss of Generality, \( s_i \) and \( s_j \) are siblings in O, on the lowest level of the tree (ie. the level furthest from the root). This is because if either \( s_i \) or \( s_j \) is not in this position, we can swap it in the tree for whatever symbol does occupy that position without increasing the F() value, which means that the newly derived code is also optimal.

(It is also useful to note that in the tree representing O, all internal (non-leaf) vertices must have two children – if not, we can shorten a branch of the tree, which will decrease F(). Since O is optimal, there can be no code with a lower F() value.)

Thus the first iteration of the algorithm (giving \( s_i \) and \( s_j \) a shared parent, and combining their frequencies) is contained in an optimal solution O. This reduces the size of the set of symbols to \( n \), and by our inductive assumption the algorithm finds an optimal code for this reduced problem – call this reduced problem code A*

Let \( T_A \) be the tree representing A, and let \( T_{A*} \) be the tree representing A*. Note that \( T_{A*} \) is exactly the tree we get by removing \( w_i \) and \( w_j \) from \( T_A \), and assigning the sum of their frequencies to their parent.

Now if we let \( T_O \) be the tree representing O, we see that doing the same thing to it (emoving \( w_i \) and \( w_j \) from \( T_O \), and assigning the sum of their frequencies to their parent) we get a reduced tree ... call it \( T_{O*} \) ... that represents a solution ... call it O* ... to exactly the same
problem that \( A^* \) solves.

One more thing to observe: We can compute \( F(A) \) directly from \( T_A \): each edge of \( T_A \) contributes “points” to \( F(S) \) determined by the leaves below it and their frequencies. The only difference between \( T_A \) and \( T_{A^*} \) is the two edges joining \( w_{i,j} \) to \( w_i \) and \( w_j \), and those two edges contribute \( f(s_i) \) and \( f(s_j) \) respectively to \( F(A) \). So

\[
F(A) = F(A^*) + f(s_i) + f(s_j)
\]

and the same relationship holds for \( O \) and \( O^* \) : \( F(O) = F(O^*) + f(s_i) + f(s_j) \)

And now it all comes together (at last!)

By our inductive assumption, \( F(A^*) \leq F(O^*) \)

So \( F(A^*) + f(s_i) + f(s_j) \leq F(O^*) + f(s_i) + f(s_j) \)

So \( F(A) \leq F(O) \)

Therefore \( A \) is optimal.

Therefore Huffman Coding finds the optimal prefix-property code in all cases.

WOOT!