1 Greedy Algorithms - Introduction

Greedy Algorithms choose some slice at each step, usually the largest in some sense, until they can't take any more slices. Often, such algorithms will result in an optimal or near-optimal solution. Examples we've seen before are Prim's Algorithm and Kruskal's Algorithm.

However, this is not always the case.

Example Suppose we wish to make 61 cents of change from quarters, dimes, nickels and pennies, using the smallest number of coins possible. If we take the largest possible coin at each stage, we will use two quarters, one dime and one penny. It's not hard to see that it's impossible with three coins, so this is the best possible.

Non-Example Suppose instead that our coins have denominations $1, 15$ and 25. If we try to make change for 61 cents using the same greedy algorithm, we will use two 25 cent coins and 11 one cent coins. However, this time the solution is not optimal - taking four 15 cent coins and one 1 cent coin is better.

Exercise. *Give an amount of change and a set of coin denominations such that*
(a) The greedy algorithm uses at least 124 *coins, while it's possible to make change with fewer than* 5 *coins.*
(b) The greedy algorithm fails to find any way to make change, even though it is possible.

Solution.
(a) The greedy algorithm will use 124 coins when making change for 248 cents with denominations $\{1, 124, 125\}$, while we could make change with just two 124 cent coins.
(b) It's possible to make change for 41 cents with denominations $\{4, 15, 25\}$, but the greedy algorithm will first choose 25 cents, then 15, then get stuck.

Exercise. *Show that if our denominations are* $\{1, c^2, ..., c^k\}$ *for some* $c, k \geq 2$*, then the greedy algorithm will always find the optimal solution.*

Solution.
Note that any optimal solution uses at most $c - 1$ coins of each denomination. Suppose for the sake of contradiction that the problem statement is false. Then there is a positive integer r and a point at which we need to make change for $n \geq c^r$ cents, and the optimal solution uses only coins of denominations in the set $\{1, c, c^2, ..., c^{r-1}\}$. However, by our note above, any optimal solution can make change for at most $\sum_{i=0}^{r-1}(c-1)c^i = c^r - 1 < n$ cents using only coins of these denominations, a contradiction.

2 Minimum Vertex Cover

We are given an undirected graph G, and would like to find the smallest set of vertices such that every edge is incident with at least one vertex. Unfortunately for us, this problem is NP-complete. However, it turns out that it is possible to find a decent approximation using a greedy algorithm.

Attempt 1 While there is at least one uncovered edge, take a vertex incident with that edge.

Unfortunately, this fails spectacularly for a graph with one node v connected to k nodes $v_1, ..., v_k$ - we might choose all of $v_1, ..., v_k$, while the minimum vertex cover is just $\{v\}$.

Attempt 2 While there is at least one uncovered edge, take the vertex incident with the greatest number of uncovered edges.

This is better, but it's possible to construct graphs for which it is off by a factor of $\Omega(\log n)$.[1]

Attempt 3 While there is at least one uncovered edge, take *both* its endpoints.

This gives us a 2-approximation - if we take the endpoints of k edges, these k edges must be disjoint, so the minimum vertex cover has size at least k, while we return a vertex cover of size $2k$.

This is actually very good - it's known that a 1.361-approximation is NP-complete, and it's not known whether or not there is an efficient 1.99-approximation algorithm.

3 Huffman Coding

We're given an alphabet of symbols $A = \{a_1, ..., a_n\}$ together with a set of weights $W = \{w_1, ..., w_n\}$ (think of these as probabilities/counts/frequencies). The goal is to output a set C of *prefix-free* binary codewords such that the *average* length of a codeword (with respect to the probabilities/counts) is as small as possible.

Huffman coding creates a leaf node for each symbol, and then adds it to a min min-heap. While the heap is non-empty, the two symbols with the smallest weights are greedily (abstemiously?) extracted. A node is created with these two symbols as its children, and weight equal to the sum of the two smallest weights. Finally, this node is added to the min-heap.

This creates a binary tree. The path from the root to a leaf node is a sequence of left/right movements, and this naturally corresponds to a binary string.

Exercise. *I have balls of colours* $1, 2, ...8$*. There are* k *balls of the* k^{th} *colour. I put all* 36 *balls in a bag, and draw one uniformly at random. You may ask me a series of yes/no questions to determine the colour of the ball. How many questions must you ask (a) in the worst case, and (b) on average?*

Solution.
(a) A simple binary search uses 3 questions in the worst case.
(b) Treat colour k as a symbol with weight k, and ask whether the colour is left/right of the current node in the Huffman code tree. This uses $\frac{17}{6}$ questions on average.

[1] Consider a set U of k vertices, and sets $V_1, ..., V_k$, with V_i containing $\lfloor k/i \rfloor$ vertices. For each i, connect each vertex in V_i with i vertices in U, such that no two vertices in V_i share a neighbour.