CS3230 Chapter 6 - Greedy Algorithms

Based on lectures by Chang Ee-Chien
Notes taken by Andrew Tan
AY18/19 Semester 1

1 Greedy algorithms

For many optimization problems, we need not use dynamic programming to determine the best choices; simpler and more efficient algorithms will do. A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

In most instances, greedy algorithms can only find the local optimal. In other problems, however, these algorithms may find the global optimal solution.

Developing a greedy algorithm is not unlike developing a dynamic programming procedure. We design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes it the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Essentially, greedy algorithms have two key properties: optimal substructure and the greedy-choice property. From here, we can see that greedy algorithms have this general form:

1. Let A be the empty set
2. Repeat
   (a) Choose the "best" c such that $A + \{c\}$ is a *feasible* solution
   (b) Insert c into A

   Until A is a solution.

We denote c to be a *candidate*, and if a set of candidates A could be extended to a solution, then we say that A is *feasible*. Furthermore, a feasible set is *promising* if it could be extended to the optimal solution.

In this chapter, we will explore several examples where greedy algorithms may help us obtain optimal solutions to certain problems.

2 Making change

Given an integer S, we define a *solution* to be a set of coins such that the total is equal to S. These coins are drawn with replacement from $\{1, 5, 10, 25\}$ cent coins. From here, we wish to output the solution which has the minimum number of coins.

We can easily develop a greedy algorithm as follows:

1. Let A be the empty set.
2. Repeat
   (a) Find c, the highest valued coin such that $c + \text{total\_amount}(A) \leq S$.
   (b) Insert c into A.

   Until $\text{total\_amount}(A) = S$.

However, we will need to prove the correctness of this algorithm. Hence, we first make a claim:

In the greedy algorithm, the set A is always promising.

We will sketch a proof by induction:

First note that A is always feasible. This is because we can always add any number of 1-cent coins to make A feasible.
Base case: Trivially, $A = \emptyset$ is promising.
Inductive step: Assume A is promising, then $A + \{c\}$ is promising, where c is the largest candidate. We will show by contradiction that $A + \{c\}$ is also promising. If $A + \{c\}$ is not promising and A is promising, then there exists a set B of coins such that:

(a) $A + B$ is promising

(b) all coins in B is smaller than c, and

(c) the total value of B is larger or equal to c.

Note that since $A + B$ is promising (i.e. is a subset of an optimal solution), then we are unable to replace a subset in B by a fewer number of coins of the same total value.

Now suppose $c = 5$, then from (b) and (c), B can only consist of 1-cent coins, and must be at least 5 1-cent coins in B. We can replace the 5 1-cent coins by a 5-cent coin, which contradicts (a). Likewise, we can arrive at similar contradictions for the cases $c = 10$ and $c = 25$.

As all cases lead to a contradiction, $A + \{c\}$ is promising.

It is noteworthy to point out that not all sets of coins allow for an optimal greedy algorithm. (e.g. in the case of having coins of values $\{1, 10, 25\}$

3 Huffman Code

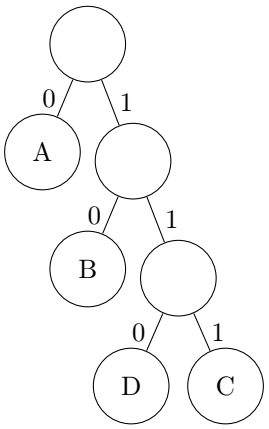
In computer systems, characters are represented as binary strings. The association between the characters to the binary sequences is denoted as a "codebook", and the binary sequence itself a "codeword". Take ASCII for example, ASCII is a fixed-length code - every character is represented by a specific 8-bit sequence. If we want to represent a text file S containing n characters and represented using ASCII, then the total number of bits used is simply $8n$.

We can use codebooks with a variable-length code such that the codeword of more freqent characters are shorter, and less frequent characters are longer, to allow us to compress S to a fewer number of bits.

3.1 Prefix code

A prefix code is a codebook where any codeword is not the prefix of other codewords. For example, this is a prefix code: $(A = 0, B = 10, D = 110, C = 111)$. If our string is represented using a prefix code, we can easily decode the string by scanning from left to right.

We can represent our prefix code as a binary tree, where each leaf represents a character. Taking our prefix code $(A = 0, B = 10, D = 110, C = 111)$, we obtain:



3.2 Finding prefix codes

Given a string S of length m, with a character set of n characters $A = a_1, a_2, \ldots, a_n$, we with to find a prefix code C for A such that S can be represented using the minimum number of bits, this optimal prefix code C is also known as the **Huffman code**.

Let f_i be the number of occurences of the character a_i and c_i be the length of the codeword for a_i in C. The length of S is $c_1 f_1 + c_2 f_2 + \cdots + c_n f_n$. This length term is the *cost* in this optimization problem.

We shall sketch an algorithm that outputs the optimal prefix code as a full binary tree. Our input shall be the set $F = \langle (a_1, f_1), (a_2, f_2), \ldots, (a_n, f_n) \rangle$.

1. Let P be the empty graph. Insert the vertices $a_1, a_2, \ldots, a_n$ into P.
2. Repeat
   (a) Find i, j where f_i, f_j are the two smallest values in F.
   (b) Create a new alphabet, and call it z.
   (c) Remove (a_i, f_i) and (a_j, f_j) from F and insert $(z, f_i + f_j)$ into F.
   (d) Add the vertex z into P.
   (e) Add two edges into P so that a_i and a_j are the children of z.

   Until F contains only 1 element.
3. Output P.

Note that this is a greedy algorithm. At each iteration, we find the best candidate and insert it into the solution.

Let us now prove its correctness:
Let P be the graph at before the next iteration of the loop, and P' be the graph after the next iteration is complete (i.e. P' is obtained from P by replacing the two smallest elements). Let F and F' be the frequency counts at these two steps respectively. Our goal is to show that if P is promising, then P' is also promising.

If P is promising, then there exists an optimal tree T that contains P. Let us treat the internal nodes in T that appears in F as leaves. Now, given a set of characters and their frequencies $F = \langle (a_1, f_1), (a_2, f_2), \ldots, (a_n, f_n) \rangle$, there exists an optimal prefix code tree P where the two smallest frequencies has the same parent (if there isn't, we can always swap between characters and that does not increase the cost). Hence, there exists an optimal tree such that the two smallest leaves have the same parent. This optimal tree contains P', and thus P' is promising.