Greedy Algorithms I

Greedy Algorithms form an important class of algorithmic techniques. We start the course by applying this technique to two problems: the simplified Knapsack Problem, and Minimum Cost Spanning Trees. We show that the "natural" greedy method does not always yield the optimal solution when applied to the Knapsack Problem, but it always yields the optimal solution when applied to the Minimum Cost Spanning Tree Problem.

Simple Knapsack Problem

Informally, the problem is that we have a knapsack that can only hold weight C, and have a bunch of items that we wish to put in the knapsack; each item has a specified weight, and the total weight of all the items typically exceeds C. We want to put as much weight as possible in the knapsack without going over the capacity.

Let $w_1, \ldots, w_d \in \mathbb{N}$ be weights and for each $S \subseteq \{1, \ldots, d\}$ let $K(S) = \sum_{i \in S} w_i$. Find:

$$M = \max_{S \subseteq \{1,\ldots,d\}} \{K(S) | K(S) \leq C\}$$

for some positive C, normally less than the sum of all weights.

Brute force search is not feasible because there are 2^d subsets of $\{1, \ldots, d\}$.

We can estimate M using the Greedy method: sort the weights in decreasing (or rather non-increasing order)

$$w_1 \geq w_2 \geq \ldots \geq w_d$$

Try the weights one at a time, adding each if there is room. Call this resulting estimate $\bar{M}$.

It is easy to find examples for which this greedy algorithm does not give the optimal solution; for example weights $\{501, 500, 500\}$ with $C = 1000$. Then $\bar{M} = 501$ but $M = 1000$. However, this is just about the worst case:

Lemma 1 Either $\bar{M} = M$ or $\bar{M} > \frac{1}{2}C \geq \frac{1}{2}M$.

Proof: Exercise.

The notion of a **polynomial time** algorithm is basic to complexity theory, and to this course (see definition below). Roughly speaking, we regard an algorithm as *feasible* (or *tractable*) if and only if it runs in polynomial time. The above greedy algorithm runs in polynomial time (see below) and is feasible to execute for values of d in the thousands or even millions. On the other hand, the blind search algorithm takes more than 2^d steps, is not polynomial time, and will never run on any physical computer (now or in the future) for values of d as large as 200 (the solar system will expire first).

Unfortunately the greedy algorithm does not necessarily yield an optimal solution. This brings up the question: is there any polynomial time algorithm that is guaranteed to find the optimal solution to the simple knapsack problem? This question will be studied in the next part of the course. The answer is that this knapsack problem is "NP hard" (assuming that the weights are given using binary or decimal notation), and hence is very unlikely to be solvable by a polynomial time algorithm.

Definition 1 An algorithm is *polynomial time* iff $\exists k$ such that the running time $T(n)$ of the algorithm satisfies $T(n) \in O(n^k)$, where $n =$ input "size".

Example 1 The running time (i.e. $T(d)$) for the knapsack problem with Greedy is $O(d \log d)$, because first we have to sort the weights, and then go through the d weights to see which ones can be added.

Lemma 2 For any constant $c > 0$ and positive integer k, $n^k \in O(2^{cn})$ but $2^{cn} \notin O(n^k)$.

This can be proved using L'Hospital's rule. The derivative of 2^{cx} with respect to x is $(c \log_e 2)2^{cx}$. So if we differentiate any number of times, the resulting function still approaches ∞ as x approaches ∞. On the other hand, if we differentiate x^k just k times, the result is the constant $k!$.

Minimum Cost Spanning Trees

We start by giving several definitions:

An *undirected graph* G is a pair (V, E) where V is a set (of vertices or nodes), and $E \subseteq V \times V$ and $(u, v) \in E$ iff $(v, u) \in E$, and $(u, u) \notin E$. The *degree* of a vertex v is the number of edges touching v. A *path* in G between v_1 and v_k is a sequence $v_1, v_2, \ldots, v_k$ such that each $(v_i, v_{i+1}) \in E$. G is *connected* if between every pair of distinct nodes there is a path. A *cycle* is a simply closed path $v_1, \ldots, v_k, v_1$ with $v_1, ..., v_k$ all distinct, and $k \geq 3$. A graph is *acyclic* if it has no cycles. A *tree* is a connected acyclic graph. A *spanning tree* of a connected graph G is a subset $T \subseteq E$ of the edges such that (V, T) is a tree. (In other words, the edges in T must connect all nodes of G and contain no cycles.)

If G has a cycle, then there is more than one spanning tree for G, and in general G may have many spanning trees, but each spanning tree has the same number of edges.

Lemma 3 Every tree with n nodes has exactly $n - 1$ edges.

Proof: Exercise (induction on n, using the fact that every finite tree has a *leaf*, i.e. a node of degree one).

We are interested in finding a minimum cost spanning tree for G, assuming that each edge e is assigned a cost $c(e)$. (Assume for now that the cost $c(e)$ is a nonnegative real number.) In this case, the cost $c(T)$ is the sum of the costs of the edges in T. We say that T is a *minimum cost spanning tree* for G if T is a spanning tree for G and given any spanning tree T' for G, $c(T) \leq c(T')$.

Given a graph $G = (V, E)$, where $c(e_i) =$ "cost of edge e_i", we want to find a minimum cost spanning tree. It turns out (miraculously) that in this case, an obvious greedy algorithm (Kruskal's algorithm) always works. Kruskal's algorithm is the following: sort the edges in non-decreasing order of costs, so that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$, add an edge one at a time, but throw the edge out if it forms a cycle.

Kruskal's Algorithm

Sort the edges: $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
$T \longleftarrow \emptyset$
for $i : 1..m$
$(*)$ if $T \cup \{e_i\}$ has no cycle then
        $T \longleftarrow T \cup \{e_i\}$
    end if
end for

But how do we test for a cycle (i.e. execute $(*)$)? After each execution of the loop, the set T of edges divides the vertices V into a collection $V_1 \ldots V_k$ of *connected components*. Thus V is the disjoint union of $V_1 \ldots V_k$, each V_i forms a connected graph using edges from T, and no edge in T connects V_i and V_j, if $i \neq j$. A simple way to keep track of $V_1, \ldots, V_k$ is to use an array $D[i]$ where $D[i] = j$ if vertex $i \in V_j$. Initialize D as follows:

for $i : 1..n$
    $D[i] \longleftarrow i$
end for

To check whether $e_i = (r, s)$ forms a cycle with T, check whether $D[r] = D[s]$. If e_i does not form a cycle with T, we update: $T \longleftarrow T \cup \{(r, s)\}$, and we merge the component $D[r]$ with $D[s]$ as follows:

$k \longleftarrow D[r]$
$l \longleftarrow D[s]$
for $j : 1..n = \#$ of vertices
    if $D[j] = l$ then $D[j] \longleftarrow k$
    end if
end for

Exercise: Given that the edges can be ordered in $m \log m$ steps, what is the running time of this algorithm?

Correctness of Kruskal's Algorithm

It is not immediately clear that Kruskal's algorithm yields a spanning tree at all, let alone a minimum cost spanning tree.

To see that the resulting collection T of edges is a spanning tree for G (assuming that G is connected), we must show that (V, T) is connected and acyclic. It is obvious that T is acyclic, because we never add an edge that results in a cycle. To show that (V, T) is connected, we reason as follows. Let u and v be two distinct nodes in V. Since G is connected, there is a path p connecting u and v in G. The algorithm considers each edge e_i of G in turn, and puts e_i in T *unless* $T \cup \{e_i\}$ forms a cycle. But in the latter case, there must be a path in T connecting the end points of e_i, so deleting e_i does not disconnect the graph.

This argument can be formalized by showing that the following statement is an invariant of the loop in Kruskal's algorithm:

$(*)$ *The edge set* $T \cup \{e_{i+1}, ..., e_m\}$ *connects all nodes in* V

Exercise: Prove, by induction, that $(*)$ is a loop invariant.

To show that the spanning tree resulting from the algorithm is in fact a minimum cost spanning tree, we reason that after each execution of the loop, the set T of edges can be expanded to a minimum cost spanning tree using edges that have not yet been considered. Hence after termination, all edges have been considered, so T must itself be a minimum cost spanning tree.

We can formalize this reasoning as follows:

Definition 2 A set T of edges of G is *promising* if T can be expanded to a minimum cost spanning tree for G.

Lemma 4 The following is a loop invariant for Kruskal's algorithm:

T is promising

Notice that the edges used to expand T to a spanning tree must come from those not yet considered, because the edges that have been considered are either in T already, or have been rejected because they form a cycle.

To prove the lemma, we argue by induction:

Basis Case: Initially T is the empty set, which is obviously promising.

Induction Step: We assume that T is promising, and show that T continues to be promising after one more execution of the loop.

case(I): e_i is rejected. T remains unchanged, and so it is still promising.

case(II): e_i is accepted. We must show that $T \cup \{e_i\}$ is still promising. Since T is promising, there is a minimum cost spanning tree T_1 such that $T \subseteq T_1$.

subcase(a): $e_i \in T_1$. Then obviously $T \cup \{e_i\}$ is promising.

subcase(b): $e_i \notin T_1$. Then, according to the Exchange Lemma below, there is an edge e_j in $T_1 - T_2$ (where T_2 is the spanning tree resulting from the algorithm) such that $T_3 = T_1 \cup \{e_i\} - \{e_j\}$ is a spanning tree. Notice that $i < j$, since otherwise e_j would have been rejected from T and thus would form a cycle in T and so also in T_1. Therefore $c(e_i) \leq c(e_j)$, so $c(T_3) \leq c(T_1)$, so T_3 must also be a minimum cost spanning tree. Since $T \cup \{e_i\} \subseteq T_3$, it follows that $T \cup \{e_i\}$ is promising.

This completes the proof of the above lemma, except for the Exchange Lemma.

Lemma 5 (Exchange Lemma) Let G be a connected graph, and let T_1 and T_2 be any two spanning trees for G. For every edge e in $T_2 - T_1$ there is an edge e' in $T_1 - T_2$ such that $T_1 \cup \{e\} - \{e'\}$ is a spanning tree for G.

Proof: Exercise (Hint: let e be an edge in $T_1 - T_2$. Then $T_2 \cup \{e\}$ contains a cycle – can all the edges in this cycle belong to T_1?).