Greedy Algorithms[1]

Simple Knapsack Problem

"Greedy Algorithms" form an important class of algorithmic techniques. We illustrate the idea by applying it to a simplified version of the "Knapsack Problem". Informally, the problem is that we have a knapsack that can only hold weight C, and we 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 exceeds C; we want to put items in the knapsack so as to come as close as possible to weight C, without going over. More formally, we can express the problem as follows.

Let $w_1, \ldots, w_d \in \mathbb{N}$ be weights, and let $C \in \mathbb{N}$ be a weight. For each $S \subseteq \{1, \ldots, d\}$ let $K(S) = \sum_{i \in S} w_i$. (Note that $K(\emptyset) = 0$.)
Find:

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

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

We can estimate M using the Greedy method:
We first sort the weights in decreasing (or rather nonincreasing order)

$$w_1 \geq w_2 \geq ... \geq w_d$$

We then try the weights one at a time, adding each if there is room. Call this resulting estimate $\overline{M}$.

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 $\overline{M} = 501$ but $M = 1000$. However, this is just about the worst case:

Lemma 1 $\overline{M} \geq \frac{1}{2} M$

[1] These notes are based on lecture notes by Stephen Cook and Michael Soltys in CSC 364, University of Toronto, Spring 2003

Proof:
We first show that if $\overline{M} \neq M$, then $\overline{M} > \frac{1}{2}C$; this is left as an exercise. Since $C \geq M$, the Lemma follows. $\square$

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 an optimal solution to the simple knapsack problem?

However, a positive answer to this question would show that NP=P, since we can show that Knapsack problem is NP-hard. Let us consider a decision version of Knapsack, in which there is one more parameter B; now, rather than trying to get S with maximal possible sum, we are just asking if there exists an $S \subseteq \{1, \ldots, n\}$ with $K(S) \geq B$

Lemma 2 *The decision version of Simple Knapsack is NP-complete.*

Proof:
The valid instances are of the form $\{w_1, \ldots, w_n, C, B\}$, and an instance is in the language if there exists an $S \subseteq \{1, \ldots, n\}$ such that $B \leq \Sigma_{i \in S} w_i \leq C$.

To see that this problem is in NP, note that given a certificate S it takes polynomial time to compute the sum of elements in S and compare it to two numbers B and C.

To see that it is NP-hard, we will use a reduction from $SubsetSum$ problem; that is, we will show $SubsetSum \leq_p DecisionSimpleKnapsack$. Recall that instances of SubsetSum are of the form $\{a_1, \ldots, a_n, t\}$, and an instance is in the language iff there is an $S \subseteq \{1, \ldots, n\}$ such that $\Sigma_{i \in S} a_i = t$. Now, $f(\{a_1, \ldots, a_n, t\}) = \{a_1, \ldots, a_n, t, t\}$. That is, set $w_i = a_i$ for $i \in \{1, \ldots, n\}$ and set both B and C equal to t. Now, if there is a set S' such that the sum of elements in it is both $\leq$ and $\geq$ to t, then it must equal to t. Similarly for the other direction, a set S with sum of its elements equal to t satisfies both $\geq B$ and $\leq C$ conditions for $B = C = t$. $\boxdot$

So unless something drastic (that is, P=NP) happens, we have no hope of getting a polynomial-time algorithm for Knapsack. However, we can change the problem in such a way that the new problem is indeed solvable in polynomial time. Let us consider a variation called FractionalKnapsack: in this

case, we will again look at the optimization (search) problem, as opposed to decision problem. Also, rather than modifying SimpleKnapsack in which we only had weights, we will now assign a weight and a profit to each element.

Let FractionalKnapsack to be defined as follows. The input is $\{(w_1, p_1), \ldots, (w_n, p_n), C\}$ where w_i are weights, p_i are profits, and C is the capacity. But now we allow fractions of elements to be put in the Knapsack. So S becomes a list of numbers $(s_1, \ldots, s_n)$, where each $0 \leq s_i \leq 1$ and s_i corresponds to the fraction of ith element in the set. Now, the capacity condition becomes $\Sigma_{i=1}^{n} w_i s_i \leq C$ and we are trying to maximize $\Sigma_{i=1}^{n} p_i s_i$.

(To make it easier to think about this problem, consider a situation when a company is trying to load a container with goods. If they are required to put each item as a whole, then it is the usual Knapsack definition. However, suppose they are transporting something that can be split – e.g., coal or grain. Now, if the whole "batch" of grain does not fit, they can just take as much of it as fits, and leave the rest).

For the case of FractionalKnapsack, the following simple greedy algorithm solves the problem optimally. The idea is to sort the elements by their unit value (that is, total profit divided by the total weight of each element). Now, start adding them in that order; when cannot put the whole element put as much as fits.

Fractional Knapsack Algorithm:

Sort the items so that: $p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$
$S = (0, 0, \ldots, 0)$
$M = 0$
for $i = 1$ to n do
    if $C - M \geq w_i$ then
        $S(i) = 1$; $M = M + w_i$
    else if $M < C$
        $S(i) = (C - M)/w_i$; $M = C$.
    else return S
    end if
end for

Minimum Spanning Trees

An *undirected graph* G is a pair (V, E); V is a set (of vertices or nodes); E is a set of (undirected) edges, where an edge is a set consisting of exactly two (distinct) vertices. For convenience, we will sometimes denote the edge between u and v by $[u, v]$, rather than by $\{u, v\}$.
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* (or *simple cycle*) is a closed path $v_1, \ldots, v_k, v_1$ with $k \geq 3$, where $v_1, ..., v_k$ are all distinct. A graph is *acyclic* if it has no cycle. 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 cycle.)

If a connected G has a cycle, then there is more than one spanning tree for G, and in general G may have exponentially 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.*

The proof is by 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 a given connected graph 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 defined to be the sum of the costs of the edges in T. We say that T is a *minimum cost spanning tree* (or an optimal 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 connected graph $G = (V, E)$ with n vertices and m edges $e_1, e_2, \ldots, e_m$, 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: first, sort the edges in increasing (or rather nondecreasing) order of costs, so that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$; then, starting with an initially empty tree T, go through the edges one at a time, putting an edge in T if it will not cause a cycle, but throwing the edge out if it would cause a cycle.

Kruskal's Algorithm:

Sort the edges so that: $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
$T \leftarrow \emptyset$
for $i : 1..m$
(*) if $T \cup \{e_i\}$ has no cycle then
    $T \leftarrow T \cup \{e_i\}$
  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 the connected components of T is to use an array $D[1..n]$ where $D[i] = D[j]$ iff vertex i is in the same component as vertex j. So our initialization becomes:

$T \leftarrow \emptyset$
for $i : 1..n$
  $D[i] \leftarrow i$
end for

To check whether $e_i = [r, s]$ forms a cycle with T, check whether $D[r] = D[s]$. If not, and we therefore want to add e_i to T, we merge the components containing r and s as follows:

$k \leftarrow D[r]$
$l \leftarrow D[s]$
for $j : 1..n$
  if $D[j] = l$ then
    $D[j] \leftarrow k$
  end if
end for

The complete program for Kruskal's algorithm then becomes as follows:

```
Sort the edges so that: c(e_1) ≤ c(e_2) ≤ ... ≤ c(e_m)
T ← ∅
for i : 1..n
    D[i] ← i
end for
for i : 1..m
    Assign to r and s the endpoints of e_i
    if D[r] ≠ D[s] then
      T ← T ∪ {e_i}
      k ← D[r]
      l ← D[s]
      for j : 1..n
          if D[j] = l then
            D[j] ← k
          end if
      end for
    end if
end for
```

We wish to analyze the running of Kruskal's algorithm, in terms of n (the number of vertices) and m (the number of edges); keep in mind that $n-1 \leq m$ (since the graph is connected) and $m \leq \binom{n}{2} < n^2$. Let us assume that the graph is input as the sequence $n, I_1, I_2, \ldots, I_m$ where n represents the vertex set $V = \{1, 2, \ldots, n\}$, and I_i is the information about edge e_i, namely the two endpoints and the cost associated with the edge. To analyze the running time, let's assume that any two cost values can be either added or compared in one step. The algorithm first sorts the m edges, and that takes $O(m \log m)$ steps. Then it initializes D, which takes time $O(n)$. Then it passes through the m edges, checking for cycles each time and possibly merging components; this takes $O(m)$ steps, plus the time to do the merging. Each merge takes $O(n)$ steps, but note that the total number of merges is the total number of edges in the final spanning tree T, namely (by the above lemma) $n - 1$. Therefore this version of Kruskal's algorithm runs in time $O(m \log m + n^2)$. Alternatively, we can say it runs in time $O(m^2)$, and we can also say it runs

in time $O(n^2 \log n)$. Since it is reasonable to view the size of the input as n, this is a polynomial-time algorithm.

This running time can be improved to $O(m \log m)$ (equivalently $O(m \log n)$) by using a more sophisticated data structure to keep track of the connected components of T; this is discussed on page 570 of CLRS (page 505 of CLR).

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. We will now prove that it does in fact produce an optimal spanning tree. To show this, we reason that after each execution of the loop, the set T of edges can be expanded to an optimal spanning tree using edges that have not yet been considered. Hence after termination, since all edges have been considered, T must itself be a minimum cost spanning tree.

We can formalize this reasoning as follows:

Definition 1 *A set T of edges of G is* promising after stage i *if T can be expanded to a optimal spanning tree for G using edges from $\{e_{i+1}, e_{i+2}, \ldots, e_m\}$. That is, T is promising after stage i if there is an optimal spanning tree T_{opt} such that $T \subseteq T_{opt} \subseteq T \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$.*

Lemma 4 *For $0 \leq i \leq m$, let T_i be the value of T after i stages, that is, after examining edges $e_1, \ldots, e_i$. Then the following predicate $P(i)$ holds for every i, $0 \leq i \leq m$:*

$P(i)$: T_i is promising after stage i.

Proof:
We will prove this by induction. $P(0)$ holds because T is initially empty. Since the graph is connected, there exists *some* optimal spanning tree T_{opt}, and
$T_0 \subseteq T_{opt} \subseteq T_0 \cup \{e_1, e_2, \ldots, e_m\}$.

For the induction step, let $0 \leq i < m$, and assume $P(i)$. We want to show $P(i+1)$. Since T_i is promising for stage i, let T_{opt} be an optimal spanning

tree such that
$T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$. If e_{i+1} is rejected, then $T_i \cup \{e_{i+1}\}$ contains a cycle and $T_{i+1} = T_i$. Since $T_i \subseteq T_{opt}$ and T_{opt} is acyclic, $e_{i+1} \notin T_{opt}$. So
$T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.

Now consider the case that $T_i \cup \{e_{i+1}\}$ does *not* contain a cycle, so we have $T_{i+1} = T_i \cup \{e_{i+1}\}$. If $e_{i+1} \in T_{opt}$, then we have $T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.
So assume that $e_{i+1} \notin T_{opt}$. Then according to the Exchange Lemma below (letting T_1 be T_{opt} and T_2 be T_{i+1}), there is an edge $e_j \in T_{opt} - T_{i+1}$ such that $T'_{opt} = T_{opt} \cup \{e_{i+1}\} - \{e_j\}$ is a spanning tree. Clearly $T_{i+1} \subseteq T'_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$. It remains to show that T'_{opt} is optimal. Since $T_{opt} \subseteq T_i \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$ and $e_j \in T_{opt} - T_{i+1}$, we have $j > i+1$. So (because we sorted the edges) $c(e_{i+1}) \leq c(e_j)$, so $c(T'_{opt}) = c(T_{opt}) + c(e_{i+1}) - c(e_j) \leq c(T_{opt})$. Since T_{opt} is optimal, we must in fact have $c(T'_{opt}) = c(T_{opt})$, and T'_{opt} is optimal.

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

Lemma 5 *(Exchange Lemma) Let G be a connected graph, let T_1 be any spanning tree of G, and let T_2 be be a set of edges not containing a cycle. Then 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 of G.*

Proof:
Let T_1 and T_2 be as in the lemma, and let $e \in T_2 - T_1$. Say that $e = [u, v]$. Since there is a path from u to v in T_1, $T_1 \cup \{e\}$ contains a cycle C, and it is easy to see that C is the only cycle in $T_1 \cup \{e\}$. Since T_2 is acyclic, there must be an edge e' on C that is not in T_2, and hence $e' \in T_1 - T_2$. Removing a single edge of C from $T_1 \cup \{e\}$ leaves the resulting graph acyclic but still connected, and hence a spanning tree. So $T_1 \cup \{e\} - \{e'\}$ is a spanning tree of G. $\square$

We have now proven Lemma 4. We therefore know that T_m is promising after stage m; that is, there is an optimal spanning tree T_{opt} such that $T_m \subseteq T_{opt} \subseteq T_m \cup \emptyset = T_m$, and so $T_m = T_{opt}$. We can therefore state:

Theorem 1 *Given any connected edge weighted graph G, Kruskals algorithm outputs a minimum spanning tree for G.*

Discussion of Greedy Algorithms

Before we give another example of a greedy algorithm, it is instructive to give an overview of how these algorithms work, and how proofs of correctness (when they exist) are constructed.

A Greedy algorithm often begins with sorting the input data in some way. The algorithm then builds up a solution to the problem, one stage at a time. At each stage, we have a partial solution to the original problem – don't think of these as solutions to subproblems (although sometimes they are). At each stage we make some decision, usually to include or exclude some particular element from our solution; we never backtrack or change our mind. It is usually not hard to see that the algorithm eventually halts with some solution to the problem. It is also usually not hard to argue about the running time of the algorithm, and when it is hard to argue about the running time it is because of issues involved in the data structures used rather than with anything involving the greedy nature of the algorithm. The key issue is whether or not the algorithm finds an *optimal* solution, that is, a solution that minimizes or maximizes whatever quantity is supposed to be minimized or maximized. We say a greedy algorithm is optimal if it is guaranteed to find an optimal solution for every input.

Most greedy algorithms are not opitmal! The method we use to show that a greedy algorithm is optimal (when it is) often proceeds as follows. At each stage i, we define our partial solution to be *promising* if it can be extended to an optimal solution by using elements that haven't been considered yet by the algorithm; that is, a partial solution is promising after stage i if there exists an optimal solution that is consistent with all the decisions made through stage i by our partial solution. We prove the algorithm is optimal by fixing the input problem, and proving by induction on $i \geq 0$ that after stage i is performed, the partial solution obtained is promising. The base case of $i = 0$ is usually completely trivial: the partial solution after stage 0 is what we start with, which is usually the empty partial solution, which of course can be extended to an optimal solution. The hard part is always the induction step, which we prove as follows. Say that stage $i + 1$ occurs, and that the

partial solution after stage i is S_i and that the partial solution after stage $i+1$ is S_{i+1}, and we know that there is an optimal solution S_{opt} that extends S_i ; we want to prove that there is an optimal solution S'_{opt} that extends S_{i+1} . S_{i+1} extends S_i by making only one decision; if S_{opt} makes the same decision, then it also extends S_{i+1}, and we can just let $S'_{opt} = S_{opt}$ and we are done. The hard part of the induction step is if S_{opt} does not extend S_{i+1}. In this case, we have to show either that S_{opt} could not have been optimal (implying that this case cannot happen), or we show how to change some parts of S_{opt} to create a solution S'_{opt} such that

- S'_{opt} extends S_{i+1}, and
- S'_{opt} has value (cost, profit, or whatever it is we're measuring) at least as good as S_{opt}, so the fact that S_{opt} is optimal implies that S'_{opt} is optimal.

For most greedy algorithms, when it ends, it has constructed a solution that cannot be extended to any solution other than itself. Therefore, if we have proven the above, we know that the solution constructed must be optimal.