ESO211 (Data Structures and Algorithms) Lectures 28 to 31

Shashank K Mehta

In these lectures we will describe the concept of *greedy algorithms*. Divide and conquer technique and the dynamic programming technique are based on reducing a given instance into one or more smaller instances if the same problem and finding a way to compute the solution of the given instance using the solutions of the smaller instances. These techniques are top-down and the bottom-up methods to exploit this relation of a larger instance with the smaller instances.

In greedy paradigm we search (or build) a solution by make one decision at a time based on the current knowledge. Further, these decisions are final, i.e., we do not revert back and change the decision because we learnt new truths at a later time. To explain this point, imagine that you are search a way out of a maze. You are a spot S where there are more than one ways which could be potential exits. You select one of those ways, say W1, based on some partial information. After a while you realize that this way is not taking you out and you return to the same point. Now you eliminate W1 and choose one of the remaining ways to get out. This is an example of changing an older decision because you became "wiser" after some more experience. In greedy algorithms we never trace back. Hence in general we do not expect to achieve correct solution using greedy method. But there are several problems where greedy paradigm give optimal solution. Usually, the greedy algorithm is simple but the proof of correctness is not trivial. This contrasts with the divide and conquer method and dynamic programming where correctness is established as soon as we establish the relation between the solution of the larger instance with those of the smaller instances.

Activity Selection Problem

Given n activities with a fixed starting and finishing times, s_i and f_i respectively, for $i = 1, 2 \ldots, n$. We denote the activities by labels $a_1, a_2, \ldots, a_n$. A subset of activities is said to be schedulable if no two activities in the subset overlap. The objective is to compute a maximum cardinality schedulable subset.

```
Sort the activities such that their finishing time is in non-decreasing order;
/* with loss of generality assume that the resulting sequence is $a_1, a_2, \ldots a_n$ */
$A := \{a_1\}$;
$j := 1$;
for $i := 2$ to $n$ do
    if $s_i \geq f_j$ then
        $A := A \cup \{a_i\}$; $j := i$;
    end
end
return $A$
```

To prove the correctness we first show that A is a schedulable set. Suppose the activities selected are $a_{j_1}, a_{j_2}, \ldots, a_{j_k}$ in order of their selection. From the condition of the loop we know that $f_{j_r} \leq$

$s_{j_{r+1}}$ for all r. Hence each activity in the sequence ends before the next starts. Hence this is a schedulable set.

Next we will show that the result is optimum. Consider any optimal schedulable set B. Write this set as a subsequence of $a_1, a_2, \ldots, a_n$. Suppose first r items are same as in A. So let the B sequence be $a_{j_1}, \ldots a_{j_r}, a_{t_{r+1}}, \ldots a_{t_{r+m}}$. That is, the first r items are same in the two solutions and $r + 1$-st items differ. Observe that B being a schedulable set so the starting times of each activity must occur at or after the finishing time of the previous activity, i.e., $s_{r_1} < f_{r_1} \le s_{r_2} < f_{r_2} \le \cdots \le s_{j_r} < f_{j_r} \le s_{t_{r+1}} < f_{t_{r+1}} \le \cdots$.

Note: Here I will put the argument given in the class slightly differently. The way we argued in the class was not how a mathematician will argue.

There may be more than one optimal solutions. Each will give it value of r. Assume that we have selected that optimal solution which gives the maximum value of r.

Our argument has two parts: (i) We will show that $r = k$, (2) next we will show that $m = 0$. Together these will imply that the chosen optimal solution is A itself.

To prove the first claim assume that $r < k$. We will show that this leads to a contradiction. We chose $a_{j_{r+1}}$ because it has the least finishing time among all the activities that do not overlap $a_{j_1}, a_{j_2}, \ldots, a_{j_r}$. Clearly $a_{t_{r+1}}$ does not overlap any of these activities. Hence $f_{j_{r+1}} \le f_{t_{r+1}}$. So $B' = (B \setminus \{a_{t_{r+1}}\}) \cup \{a_{j_{r+1}}\}$ must also be a schedulable set and $|B'| = |B|$ So B' is also an optimal solution. But if we write B' as a subsequence of $a_1, a_2, \ldots, a_n$, then first $r + 1$ activities will match with the sequence of A. But that is impossible because we had chosen that optimal solution which gave the longest initially matching sequence. This proves that $r = k$.

So we see that B is $a_{j_1}, \ldots a_{j_k}, a_{t_{k+1}}, \ldots a_{t_{k+m}}$. In our algorithm we stopped after k iteration because at that time all the remaining activities overlapped with already selected activities, namely, $a_{j_1}, \ldots, a_{j_k}$. But if $m > 0$, then $a_{t_{k+1}}$ must not interfere with $a_{j_1}, \ldots, a_{j_k}$. This is a contradiction. So we conclude that $m = 0$. Hence we prove that $B = A$, i.e., A is optimal.

Minimum Spanning Tree: Kruskal's algorithm

Given an edge-weighted graph $G = (V, E, w)$ where w denotes the weight function. Compute a minimum weight spanning tree of G.

```
Sort $E$ in non-decreasing order of their weights;
/* Assume that the sorted list is $e_1, e_2, \ldots, e_m$                    */
$A := \emptyset$;
for $j := 1$ to $m$ do
    if $A \cup \{e_j\}$ is cycle-free then
        $A := A \cup \{e_j\}$;
    end
end
return $T = (V, A)$
```

For the analysis, denote the value of the set variable A after i rounds by A_i. So A_{n-1} will denote the final value of A, i.e., $T = (V, A_{n-1})$. First we show that the T is a spanning tree of G. Clearly, by construction, T does not contain any cycle. Hence all we need to show that T is connected.

Assume that T is not connected. So there is a partition of vertices, V_1 and V_2, such that there is no edge in T between any vertex of V_1 and that of V_2. But G is connected so there must exist at least one edge between these sets in G. Let it be e_j. Clearly $e_j \notin A_{n-1}$ so $A_j = A_{j-1}$. This implies that e_j was considered after $j - 1$ rounds and it was rejected. That means $A_{j-1} \cup \{e_j\}$ must be containing a cycle. That, in turn, implies that $A_{n-1} \cup \{e_j\}$ must also contain a cycle. But that is impossible because A_{n-1} has no cycle and it also does not have any edge between V_1 and V_2, while e_j is an edge between these sets. This is absurd. Hence the assumption must be false. So $T = (V, A_{n-1})$ is a tree over the entire set of vertices of G, i.e., T is a spanning tree of G.

Next we will show that T is a minimum weight spanning tree. To simplify the proof first we assume that all edges have different weights. We will describe the general case later.

Suppose the final set A_{n-1} contains $e_{j_1}, e_{j_2}, \ldots, e_{j_{n-1}}$ in order of their selection. Consider an optimum solution (V, B). Let e_x be the maximum weight edge in A_{n-1} such that all edges of A_{n-1} with lesser weight than e_x also belong to B, but $e_x \notin B$. Note that some edges of same weight as e_x may also belong to B. Call $w(e_x)$ the critical weight with respect to (V, B). Without loss of generality let us assume that among various optimal solutions we chose the optimal spanning tree in such a way that the number of edges having weight less than or equal to $w(e_x)$ and also belong to B, is maximum.

Claim: The weights of the edges of $B \setminus A_{n-1}$ is greater than or equal to $w(e_x)$.

Proof of Claim: Assume that there is an edge $e' \in B \setminus A_{n-1}$ and $w(e') < w(e_x)$. Then the algorithm must have considered e' before e_x. But e' was not included in A so we can conclude that inclusion of e' must be forming a cycle. But a that time already present edges in A had lesser weight than $w(e_x)$. So all those edges are also in B. This we see that $(A \cap B) \cup \{e'\}$ must contain a cycle. This implies that B contains a cycle because $e' \in B$. But this is absurd since (V, B) is a tree. Hence the assumption must be wrong. So the claim holds.

Consider the graph $H = (V, B \cup \{e_x\})$, i.e., add the edge e_x in (V, B). If $e_x = \{u, v\}$, then there is not a cycle in H because there was a path between u and v in (V, B) and now we added the edge $e_x = \{u, v\}$. Suppose the edges in this cycle are $e_x, e'_1, e'_2, \ldots, e'_p$. If all the edges of this cycle are also in A_{n-1}, then the entire cycle must be present in (V, A_{n-1}). But that is impossible since it is a tree. So without loss of generality assume that $e'_1 \in B \setminus A_{n-1}$. From the above claim $w(e'_1) \geq w(e_x)$. Consider $B' = (B \setminus \{e'_1\}) \cup \{e_x\}$.

Claim: (V, B') is a spanning tree.

Proof of the Claim: Exercise.

Since $w(e'_1) \leq w(e_x)$, (V, B') has no more than the weight of (V, B). But the latter is optimum so the former is also optimum. Observe that the critical weight with respect to (V, B') greater than or equal to $w(e_x)$ because all lesser weight edges are also in B'. But we see that the total number of edges in $A_{n-1} \cap B'$ of weight $\leq w(e_x)$ is one more than those in $A_{n-1} \cap B$. This contradicts the choice of B. Hence we conclude that $A_{n-1} = B$. Hence (V, A_{n-1}) is a minimum weight spanning tree.

0.0.1 Implementation Details and Time Complexity

The only step in the algorithm for which we need to explain the details, is checking whether a new edge is creating a cycle with already selected edges. Observe that at any arbitrary sage in this algorithm we have a set of trees such that each vertex of the graph belongs to exactly one of these trees. Hence we can view this as a **partition** of the vertex set. Thus we can view this as a collection of disjoint subsets (i.e., no vertex belongs to more than one subset) of vertices where each vertex belongs to some subset (i.e., each vertex belongs to exactly one subset). When we want to decide whether edge $e = \{u, v\}$ creates a cycle, we only need to check whether u and v belong to same subset. Observe that inclusion of e in A will create a cycle if and only if u and v belong to the same subset. If they do not belong to the same subset, then we will include e in A and replace their subsets by their union. Initially there will be $n - 1$ subsets, each containing one vertex. The process will terminate after there is a single set containing all the vertices.

In the next set of lectures we will discuss a data structure for storing a disjoint collection of sets in which we can do two operations: Find (find the set which contains the given element) and Union (replace given two sets by a single set which is their union. We will show that if there are p Find operations and q Union operations on this data structure, then it will cost $(p + n \log n)$. Observe that in this setting there can be at most $n - 1$ unions.

Suppose m is the number of edges in the graph and n is the number of vertices. Above mentioned datastructure will allow us to implement Kruskal's algorithm in $(m+n \log n)$ because we will perform m Find operations and $n - 1$ union operations.

Minimum Spanning Tree: Prim's algorithm

Prim's algorithm is very similar to Kruskal's algorithm but in this case at any intermediate stage we will create only one tree. The algorithm is described below. In the algorithm $\delta(S)$ is the set of edges which have one vertex in S and the other vertex in $V \setminus S$, for any subset S of V.

```
$S := v_1$;
/* $v_1$ is any arbitrary vertex                                  */
$A := \emptyset$;
for $i := 1$ to $n - 1$ do
    Select the minimum weight edge $e = \{u, v\} \in \delta(S)$;
    $A := A \cup \{e\}$;
    $S := S \cup \{v\}$;
    /* assuming that $u$ already belongs to $S$                   */
end
Return $S$;
```

Suppose the algorithm includes the edges in to A in order $e_1, e_2, \ldots, e_{n-1}$. Let $A_i = \{e_1, \ldots, e_i\}$. To prove the correctness let us assume that $T = (V, B)$ is a minimum cost spanning tree. Assume that (V, A) is not an optimal solution. Hence $A \neq B$. Suppose j_0 is the minimum index such that $e_i \in B$ for $i < j_0$ and $e_{j_0} \notin B$. Without loss of generality let us assume that B is that optimal solution for which j is the largest.

Suppose the value of set-variable S after $j_0 - 1$ iteration is S'. Consider the graph $(V, B \cup \{e_{j_0}\})$. It must contain a cycle which passes through e_{j_0}. This cycle goes from S' to $V \setminus S'$ when it passes through e_{j_0}. Hence there must be an edge e_k through which it returns to S'. Hence $e_k \in \delta(S')$. Since e_{j_0} is the first edge of A which connects one vertex of S' with a vertex out of it. Although e_k may or may not be in A, but it is clear that $w(e_{j_0}) \leq e_k$. Let $B' = (B \setminus \{e_k\}) \cup \{e_{j_0}\}$. The observe that $T' = (V, B')$ is a spanning tree and its weight cannot be less than that of T so T' is also optimal. But first j_0 edges of A are common with B'. This contradicts that B has first $j_0 - 1$ edges are common with B and this number is largest among all optimal solutions. Hence the assumption must be false and (V, A) must be optimal.

0.0.2 Implementation Details and Time Complexity

The implementation issue that we need to highlight is how to select the minimum cost edge from $\delta(S)$. One way to deal with this problem is to use a min-heap datastructure to store all the edges of $\delta(S)$. It will take at most $O(\log m)$ to extract the minimum cost edge from it (note that structure may have at most m edges in it). Now let us see how to update this datastructure after including the vertex v into S. Observe that we must delete all the edges from the heap which are between

v and the other members of S. and we need to include all the remaining edges incident on v into the datastructure. It takes $O(\log m)$ time to insert or delete an edge from the heap. Each edge gets included in S at most once. Hence it will be deleted from it at most once. Thus the total cost is $O(m \log m)$.

Fractional Knap Sack Problem

You are given a sack of V volume and a choice of products. Say the product volumes are $V_1, V_2, \ldots, V_n$ and their respective prices are $P_1, P_2, \ldots, P_n$. We can choose any fraction of any product and put in the sack. Our goal is to pack a selection of these products (so total volume must not exceed V and ensure that total value of the packed product is maximum possible.

The algorithm is to sort the products in non-decreasing order of $R_i = P_i/V_i$. Now pack as much of product with largest R-value as possible. If there is room in the sack, then pack the product with the next R-value, so on.

Prove that this algorithm gives optimum result.

A Variation of this problem is called zero-one knap-sack problem. In this case we must either pack the entire amount of a product or none at all. Give an example in which above algorithm will not give an optimum result for this variation.