CS430

LECTURE NOTE NO. 5

GREEDY METHOD

In these notes we illustrate the greedy techniques in the design and analysis of algorithms. Note that the design of efficient algorithms is dependant on both the properties of the problem and the data structures which would enable efficient implementation. This will be evident in the problem of the Minimum Spanning Tree construction.

1 Minimum Spanning Trees

The Minimum Spanning Tree (MST) Problem: Given a graph $G = (V, E)$ with weights on the edges, $w : E \rightarrow \mathcal{Z}$, find a spannng tree of minimum weight.

In order to solve this problem we will first devise a critical lemma which will help characterize a Minimum Spanning Tree: We need the following definitions: *A* **cut-set** *($Cut(S, V - S)$ in a graph is a set of edges that partitions the vertices of G into at least two parts*, i.e.

$$Cut(S, V - S) = \{(u, v) | u \in S, v \in V - S\}$$

A **cut-set** *w.r.t. a set of edges A is a set of edges which do not include any edge in A and partitions the vertices into at least two parts*, i.e.

$$Cut(S, V - S, A) = Cut(S, V - S) \text{ if } Cut - Set(S, V - S) \cap A = \phi$$

A *partial MST* is a set of edges that is a subset of a MST.

Lemma 1.1. *Let A be a set of edges forming a partial MST and let e_m be the minimum weight edge in a cut-set $(S, V - S)$ w.r.t. A. Then $A \cup \{e_m\}$ is either a Minimum Spanning Tree or a partial MST.*

Proof. By contradiction. Suppose not, i.e. there does not exist any spanning tree which is an extension of the partial tree A and which contains e_m. Then there exists another edge e' in the cut-set $(S, V - S)$ which is part of a minimum spannning tree MST, T. Note that this edge is not part of set A, i.e. $e' \notin A$. In this tree suppose we add e_m. Because there is a unique path between every pair of vertices, consider the unique cycle formed on adding e_m. Replacing e' by e_m will create a new tree with lower or the same cost. And this new tree contains A and e_m and is thus an extension of the partial tree $A + e_m$. This gives us a minimum spanning tree satisfying the desired property. □

The lemma provides us with a method to add edges which are assuredly part of some minimum spanning tree, We discuss two algorithms for the minimum spanning tree.

1.1 Kruskal's Algorithm-Algorithm 1

Algo MST-Kruskal.

```
    Sort the edges in increasing order of
            weight, $e_1, e_2 \ldots e_k$;
    $T \leftarrow \Phi$;
    $i \leftarrow 1$;
    Repeat
      begin
            If $e_i$ does not form a cycle when added to $T$
            then $T \leftarrow T + \{e_i\}$;
            $i \leftarrow i + 1$;
      end;
    Until $|T| = n - 1$
```

Before we discuss the time complexity of the algorithm and discuss how to efficiently determine if e_i does not form a cycle, we, first, determine the correctness of the algorithm.

Lemma 1.2. *Algo MST-Kruskal correctly computes the minimum spanning tree.*

Proof. The proof is by induction on the number of iterations. We show that T is a partial minimum spanning tree at every step in the algorithm. Let us term the forest obtained at the ith iteration of the algorithm as T_i. Before the first execution, the forest is T_0. For the basis step note that the first edge added, termed (u, v), is a minimum cost edge in the cut $(\{u\}, V - \{u\})$. Applying the lemma 0.1 proves that T_1 is either a partial MST or a MST. At the $i + 1$st iteration of the algorithm, if edge e_j is added to the forest then it is the minimum edge in a cut-set in G w.r.t. T_i. That edge e_j is part of a cut-set w.r.t. T follows from the fact that e_i does not form a cycle when added to T_i. Furthermore, it is the minimum cost edge in the cut-set since all edges with cost less than or equal to e_i are either in T_i or form a cycle with edges in T_i. Again applying lemma 0.1 shows that T_{i+1} is a partial MST or a MST. $\square$

1.1.1 Efficient Implementation of S-MST

We consider an efficient implementation of the algorithm described above. The algorithm as described requires a subprocedure to determine if a cycle is present when an edge, say $v, w)$ is considered for addition to the current forest or partial MST. A straightforward approach to this would be to test for a path between v and w in the current forest. This is easily done by a depth-first search or a breadth-first search, which would require $O(n)$ time since there are at most $n - 2$ edges in the forest.

For a more efficient approach consider to maintain each connected component of the graph as a set of vertices. Each set has a label which is also the label of the connected component. The following operations required for maintaining the connected components in the graph translate into operations on sets.

- To determine whether a cycle is formed on the addition of an edge, it suffices to determine if the two endpoints of the edge are in the same component. This is equivalent to determining if the two vertices are contained in the same set. This is achieved by a find operation which allows us to determine the label of the set containing a given element.

- On addition of an edge into the forest, a merger of two components is equivalent to the union of two sets.

We thus need a data structure which maintains sets under the operations of UNION and FIND which are defined as follows:

(i)$UNION(S_1, S_2) \rightarrow S$ returns the union of two sets S_1 and S_2.

(ii)$FIND(x)$ returns the label of the set containing x.

A data structure which allows $UNION$ in $O(1)$ time and $FIND$ in $O(lognn)$ time is available (appears later).

In the algorithm there are $O(n)$ merges of components required and, for $O(m)$ edges it is required to determine if they form a cycle on addition to the current graph. Using the UNION-FIND data structure this requires $O(m \log n + n)$ steps. Thus the algorithm requires $O(m \log n + n)$ steps.

1.2 Prim's Algorithm

Our second algorithm maintains exactly one component S and finds the smallest edge in the cut $(S, V - S)$ thus directly exploiting the property that the smallest edge in the cut is part of the MST.

Algo Cut-MST.

```
    $S \leftarrow v_1$
    $T \leftarrow \Phi$
    Repeat
        begin
            Find smallest edge $e_m$ in $Cut - Set(S, V - S)$;
            $T \leftarrow T + \{e_m\}$;
            Let $e_m = (u, v), u \in S, v \in V - S$;
            $S \leftarrow S + \{v\}$
            Update edges in cut-set;
        end
    Until $|T| = n - 1$;
```

We can charaterize how the cut changes at each iteration. Let CS_i be the cut set $(S, V - S)$ after the addition of the ith vertex v_i to S. Then the folllowing recurrence holds,

$$CS_i = CS_{i-1} + \{(v_i, w) | w \in V - S\} - \{(u, v_i | u \in S\}$$

Initially CS_1 comprises edges between v_1 and the remaining vertices in the graph.

The proof of correctness of the algorithm is left as an exercise to the reader.

1.2.1 An Efficient Implementation

We describe an efficient implementation of the algorithm described above.

At each step we are required to find the minimum edge in maintain a cut-set. Instead of explicitly maintaining the cut-set, we consider the vertices in $R = V - S$. For each vertex $v \in R$

compute the smallest edge e_v in the cut-set (S, V_S), given by CS_i, and maintain the cost of that edge as a label, $label(v)$, on vertex v. Finding the smallest edge e_m in the cut-set $(S, V-S)$ now becomes equivalent to finding the vertex with the smallest label in R.

At each iteration the cut-set changes. Suppose vertex v is added to S. The edges which change in the cut set are incident onto vertex v. Edges of the form $(v, u), u \in R$ are required to be added into the cut-set. Edges deleted are all incident onto v. The label of each vertex in R is thus changed as follows:

$$label(u) = min\{label(u), w(v, u)\}$$

We thus require a data structure which allows us to maintain the minimum of a set where the set changes with items in the set decreasing in value. The heap forms a natural data structure where all vertices in R are stored and ordered by their labels.

Updates to the labels are performed in $O(\log n)$ steps in the heap. And the minimum can also be extracted in $O(\log n)$ steps. It remains to bound the number of heap operations. At each iteration when v is added to S, the number of updates to the heap is $O(m_v)$, the number of edges incident onto v. Summing this over all the vertices gives the total time required by the algorithm as $O(m \log n)$.

2 Huffman Encoding

Given a set of alphabets $A = \{a_1, a_2 \dots a_n\}$ used in a text, each occurring with probability p_i, we would like to construct a prefix free encoding which minimizes the expected length of the encoded text. We may assume that the coding alphabet is $\{0, 1\}$. We thus have to construct the following function $\mathcal{E} : A \rightarrow \{0, 1\}^n$. We let $|\mathcal{E}(a)|$ represent the length of the code for $a \in A$. The minimum weighted code problem requires us to minimize

$$\Sigma_{1 \leq i \leq n} |\mathcal{E}(a_i)|.p_i$$

Prefix free binary codes for an alphabet A , $\mathcal{P}(A)$, may be represented by a binary tree as follows: Let $\mathcal{T}(A)$ be a labeled binary tree with n external or leaf nodes forming the set $\mathcal{L}$. An internal node has two branches labeled with 0 and 1. With each external node, e, is associated an alphabet a_i. We let $A(e)$ represent this item. Also associated with the leaf is a weight, $w(e)$ where $w(e) = p_i$ if $A(e) = a_i$. Let $path(e)$ represent the path from the root to the leaf node. The code for the item, $A(e)$ will be the sequence of labels on $path(e)$. The length of the code, $A(e)$, is the length of the path $path(e)$ which is represented by $l(e)$. The weight of the binary tree is $wt(\mathcal{T}) = \Sigma_{p \in \mathcal{L}} l(p) w(l)$.

The following lemma is left as an exercise for the reader.

Lemma 2.1. *Let A be a set of items. $\mathcal{P}$ is a prefix free code iff $\exists$ a tree $\mathcal{T}(A)$.*

To find the minimum weighted code, it thus suffices to find the minimum weight binary tree with n external leaves. We call the minimum weighted binary tree, $\mathcal{MT}(A)$.

The following algorithm finds $\mathcal{MT}(A)$.

Algo Huffman

1. If the number of items is two, the minimum cost tree simply has a root node and two leaves.
2. Let a_1 and a_2 be the two smallest weighted items. Combine the two items into one, a', with weight $p_1 + p_2$.
3. Recursively, find $\mathcal{MT}$ for the items $A' = A - \{a_1, a_2\} + \{a'\}$.

Let $\mathcal{T}(A)$ be the tree constructed. The following equation holds

$$wt(\mathcal{T}(A)) = wt(\mathcal{T}(A')) + p_1 + p_2$$

It is easy to see that the optimum tree has the two smallest weighted items, a_1 and a_2, as siblings. Using this fact and the principle of optimality,

$$wt(\mathcal{MT}(A)) = wt(\mathcal{MT}(A')) + p_1 + p_2$$

The algorithm can be proved correct by induction.

3 Job Scheduling on a Single machine

Given a set of jobs $\mathcal{J} = (j_1, j_2 \dots j_n)$ with start and finish times $s_1, s_2 \dots s_n$ and $f_1, f_2 \dots f_n$, respectively, the *job scheduling problem* is to schedule the maximum number of jobs on a single machine.

We assume w.l.o.g. that the jobs are ordered so that $f_1 \leq f_2 \dots f_n$. We next show that the greedy method provides us with an optimum solution. We need the following lemma.

Lemma 3.1. *There exists an optimal schedule which includes j_1.*

Proof: The proof is by contradiction. Consider an optimal schedule O which does not include j_1. Let j_k be the lowest indexed job in the optimum schedule. Since $f_k \geq f_1$, the job j_k can be replaced by j_1 in O to give another scedule O' and which includes j_1. Furthermore, $|O'| \geq |O|$ since other jobs may be added to O' thus giving an optimal schedule which incudes j_1. ■

Note the similarity of this lemma with that which justifies the addition of the smallest edge in the graph into the minimum spanning tree.

We extend this lemma to enable us to design a greedy algorithm for the job-sceduling problem. Let O_p be a partial solution to the job scheduling problem. O_p can be extended into an optimal solution. Let J_p be the set of jobs $\{j_1, j_2, j_3 \dots j_k\}$, ordered by their finish times, which are feasible w.r.t. O_p, i.e. jobs in J_p can be added to O_p to give a valid schedule.

Lemma 3.2. *There exists an optimal solution which includes $O_p \cup \{j_1\}$.*

Proof: Similar to one above.

Using the above property we design an algorithm for the job scheduling problem:

```
Algorithm Job-Greedy
        begin
        $S \leftarrow \{j_1\}$
```

```
For i=2 to n do
If $S \cup j_i$ is a feasible schedule
then $S \leftarrow S \cup \{j_i\}$
end
```

The correctness of the algorithm is immediate from Lemma 4.2 The reader may convince himself that the time complexity is $O(n)$.

4 MATROIDS*

We now consider a class of problems that can be solved efficiently using the greedy method.

Consider the subset system $(S, \mathcal{I})$ where S is a set of elements and $\mathcal{I}$ is a collection of subsets of S. The sets in $\mathcal{I}$ are called *independant* sets.

A subset system is called a *matroid* if it satisfies the following properties:

- S is a finite non-empty set.
- *Heredity Property:* Let $A \subset B$. If $B \in \mathcal{I}$ then $A \in \mathcal{I}$.
- *Exchange Property:* Let $A, B \in \mathcal{I}$. $|A| < |B|$ then $\exists y \in B$ such that $A \cup \{y\} \in \mathcal{I}$.

The following interesting fact arises from the above properties:

Lemma 4.1. *Maximal independant sets have the same size.*

This fact follows immediately from the exchange property.

We next consider *Weighted Matroids.* Let w be a weighting function $w : S \rightarrow Z^+$. Furthermore, define $wt(A), A \in \mathcal{I}$ to be $\Sigma_{a \in A} w(a)$. The weighted matroid problem (WMP) is to find the maximal independant set of minimum weight.

The methodology to solve this problem is the greedy method. We need the following property:

Lemma 4.2. *Let $A, B \in \mathcal{I}$ such that $A \subseteq B, |A| \leq |B| - 1$. Let x be an element such that $A \cup \{x\} \in \mathcal{I}$. Then $\exists y \in B - A$ such that $B - \{y\} + \{x\} \in \mathcal{I}$.*

Proof: We prove the above by induction on $diff = |B| - |A|$. Consider when $diff = 1$. Let $A_1 = A \cup \{x\}$. Since $A_1 \in \mathcal{I}$ we are done. Next consider the induction step. Assume that the above is true when $diff \leq k$. Let $|B| - |A| = k + 1$. Again let $A_1 = A \cup \{x\}$. Since $|A_1| < |B|$, $\exists y' \in B$ such that $A_2 = A_1 \cup \{y'\} \in \mathcal{I}$. Note that $y' \neq x$. By induction $\exists y \in B$ such that $B - \{y\} + \{x\} \in \mathcal{I}$. This proves the result. ■

To justify the greedy method we need the following lemma. Let A be a partial solution to WMP, i.e. $\exists A', A \subseteq A'$ such that A' is a solution to WMP. Let $REM = \{x | x \in S, \{x\} \cup A \in \mathcal{I}$.

Lemma 4.3. *Let min be the least weighted element in REM. Then $\{min\} \cup A$ is a partial solution to WMP.*

Proof: The proof is by contradiction. Suppose not. Let B be an optimal solution such that $A \subset B$ and does not contain min. By Lemma 5.2 $\exists y \in B$ such that $B' = B - \{y\} + \{min\}$ is in $\mathcal{I}$. Moreover $wt(B') \leq wt(B)$. This either contradicts the optimality of B or shows that there exists an optimum solution which includs min. ■

The following greedy algorithm suggests itself now. Assume $S = \{e_1, e_2 \ldots e_n\}$ and the elements are sorted by weight.

```
Algorithm Matroid-Greedy
    begin
    $M \leftarrow \{e_1\}$
    For i=2 to n do
    If $M \cup e_i$ is $\in \mathcal{I}$
      then $M \leftarrow M \cup \{j_i\}$
    end
```

It is easy to show that M gives the least weight maximal independant set. Note the amazing similarity of the algorithm and its proof with that of the greedy MST algorithm.