Lecture 9: Greedy algorithm

First, the basic concept of greedy algorithm is finding a greedy strategy such that after making the greedy choice, the remaining sub-problem still admits an optimal solution. That is, the greedy choices made at each step belong to some optimal solution. This is called optimal substructure property.

Now minimum spanning tree (ch. 23). For a connected (undirected) graph $G$, find a tree that connects all the nodes such that the total edge costs are the smallest. The well-known Kruskal’s algorithm is a greedy algorithm: first sort the edges by edge weights; then for each edge, add edge if not cycles is created by this edge. We discuss in class to claim this algorithm will always lead to a spanning tree (see your notes/textbook). Question: is this optimal? Yes, and we ow show each added edge is safe to add: safe means we are not making mistakes at that step (the edges we picked so far belong to some MST) at each step.

We first assume all edge costs are distinct. The cut property is a fundamental property of graph cut: let $S$, $V - S$ be partitions the nodes, and edge $e = (v, w)$ is the cheapest edge across $S$ and $V - S$ (say $v \in S$). Claim: every MST contains $e$. Proof: consider spanning tree $T$ does not contain $e$. Will prove $T$ is not MST. by the exchange argument (used several times before fro greedy algorithm): will find an edge $e'$ of $T$ that is most costly than $e$ and we will replace $e'$ with $e$ to get a new cheaper $T'$. Now, since $T$ does not have $e = (v, w)$. There is a path in $T$ from $v$ to $w$. Since $v$ and $w$ on different partition, we follow this path from $v$ to $w$ and we will leave $S$ at node $v'$ and enter $V - S$ at $w'$ (for the first time). And $e' = (v', w')$ belongs to $T$. Now, we remove $e'$ and include $e$ instead. Claim: what we get is a spanning tree. First it is tree: still connected: each pair of node can still reachable (now going through $e$ instead of $e'$). It is also not hard to show what we have is acyclic. The changed tree $T$ is less costly.

Now we use this property to show Kruskal’s algorithm is correct (meaning optimal). Consider one edge $e = (v, w)$ added by the algorithm. We let $S$ be the nodes reachable from $v$ by the selected edges by the algorithm so far. In class, we show this $S$ works: first $v$ in $V$ but not $w$ (why?). Also, $e$ must be the first edge between $S$ and $V - S$ (why?). Thus, $e$ is the cheapest between $S$ and $V - S$. So $e$ belongs to every MST according to the “fundamental property of cut”. Moreover, although I did not explicitly show that the selected edges by the Kruskal’s algorithm must form a spanning tree, this is in fact easy to see. Suppose the selected edges do not form a spanning tree, then suppose these edges will connect nodes in two (the case of having three or more components is similar) disconnected components $C_1$ and $C_2$. Let $S$ be the nodes in $C_1$ and the rest nodes are in $C_2$. Then there must be some edge crossing the cut of $(C_1, C_2)$ (why?). In this case, Kruskal’s algorithm will select some edges crossing this cut since such edge will not lead to cycles. This is a contradiction to that $C_1$ and $C_2$ are disconnected.

Finally, I will show that we can make Kruskal’s algorithm work even when the edge weights are not distinct. In this case, as I explain in class, we can disturb the edges in $G$ by very small amount s.t. the resulting graph $G'$ has all distinct edge weight. Then we can find an MST $T$ from $G'$ using Kruskal’s algorithm. Now I claim $T$ is also an MST for the original (unperturbed) graph $G$, if the perturbation of edge weights is small enough. This can be proved in a rigorous way.

The second algorithm for MST is the Prim’s algorithm. The idea is to grow a MST from a node $s$, and each time, add the cheapest edge to connect a new node from the partial tree (and thus each step is a tree). The Prim’s algorithm maintains a set $S$ be the nodes connected by the current spanning tree so far. Initially, $S = \{s\}$. While the size of $S$ is less than $n$, we find a node $v \notin S$. s.t. $\text{cost}(u, v)$ is minimized for some $u$ in $S$. We add $v$ to $S$. The correctness follows the cut property. The selection of new node is done using a priority queue which represents for each un-connected node, the cost of linking to the partial tree. Time analysis: suppose use binary heap to implement a priority queue. We need $O(n)$ to initialize. There are $O(n)$ iterations, each for a new node. Selecting this new node takes
log(n) time using the priority queue. Then for each node, need to update their neighbors, which takes total \( O(m) \) since the total length of the adjacency lists is \( 2m \). One update operation will take \( O(\log n) \) each. So total time: \( O(n \log n + m \log n) = O(m \log n) \). Note this can be improved using more advanced data structure. As comparison, the Kruskal’s algorithm will run slightly slower. See the textbook for its time analysis.

**Lecture 10: Greedy algorithm: Huffman coding**

We now switch our attention to the Huffman encoding, which is a widely used data compression method. Suppose we have a text from some alphabet with A, B, C, D. We need to convert the text to a binary encoded string. The straightforward approach is to use 2 bits per symbol: A is encoded as 00, B is 01, C is 10, and D is 11. Suppose the text is 100M long, and we need 100M*2=200 MBits. One observation is that symbol frequency is quite different. Thus, to reduce the encoded string length, we want to encode more frequent words with shorter codes. For example, let us suppose the number of As in text is 60M, Bs of 5M, Cs of 15M, and Ds of 20M. We will use variable length codewords. To avoid ambiguity for decoding, we require that no code word is a prefix of other code words (such codewords are called prefix code). One possible prefix code for our example is: A:0, B:110, C:111 and D:10. This encoding leads total size of \( 60 \times 1 + 5 \times 3 + 15 \times 3 + 20 \times 2 = 160 \) MBits, which is savings of 20 percent.

Now here is the problem formulation: suppose symbol frequencies are known, \( f_1, f_2, \ldots, f_n \) for the \( n \) symbols in the alphabet. Our goal is to design codewords that minimize total encoded length. That is, we want minimize \( \sum_i f_i \cdot |w_i| \), where \( w_i \) is the code word for the \( i \)-th symbol. A nice property of prefix code is that prefix code can be represented as a binary tree. In this binary tree, a node is either a leaf or has two children. That is, a node with a single child is not allowed (Why do we have this full binary tree property?). Each branch of the tree is labeled by a single digit, 0 or 1. Look at Figure 16.4 for an example of this tree. We call the depth of a node as the number of bits traversed from root to that node. Thus, the total length of the encoded string is: \( \sum_i f_i \cdot \text{depth}(\text{leaf}_i) \), noting leaves correspond to codewords (Why?).

An alternative formulation for expressing the total encoded length is to given internal nodes assigned frequencies as the summation of frequencies of the leaves under this node. We claim that total encode length is equal to summation of frequencies of all the nodes, including the internal nodes, in the tree (except the root). Why? Intuitively, we distribute the frequency of a leaf times its depth to each node on the path from root to the leaf.

We now study the properties of the complete binary tree constructed in Huffman encoding. The key observation is that the two lowest frequency symbols must be the two siblings with largest depth. To see this, suppose this is not true and the least frequent symbol is not assigned to the leaf with largest depth. Then we simply swap the symbol with the largest depth with the least frequent symbol, and this swapping will make the text shorter. Also note that there must exist a symbol as the sibling of the deepest symbol.

This implies that we can choose two lowest frequency symbols and create a subtree. From the second formulation, we know we can assign the frequency of this newly created internal node (called \( w \), with two least frequent symbols \( w_1 \) and \( w_2 \) as children) as the summation of the frequency of \( w_1 \) and \( w_2 \). This allows us to shrink in this problem: get rid of \( w_1 \) and \( w_2 \), and add a new symbol \( w \) into the list of symbols to process. Then we repeat the process. This leads to a simple algorithm: at each iteration, pick the two symbols with lowest frequencies and form a subtree with these two symbols; remove these two symbols and create a new symbol with the summation of the these removed symbols as its frequency. Continue until there is only one symbol left. Try to run some examples yourself to make sure you understand this algorithm.

But why does this work? That is, why the algorithm gives optimal solution? Recall that we can choose two lowest frequency symbols and create a subtree. From the second formulation, we know we can assign the frequency of this newly created internal node (called \( w \), with two least frequent symbols \( w_1 \) and \( w_2 \) as children) as the summation of the frequency of \( w_1 \) and \( w_2 \). This allows us to shrink in
this problem: get rid of $w_1$ and $w_2$, and add a new symbol $w$ into the list of symbols to process. Then we repeat the process. But why does this work?

It is useful to think of picking formulation 2: our goal is to find a tree where each node has a frequency (for internal node, the frequency must equal to the total frequencies of the leaves under this node). Recall that we now have $n - 1$ symbols in the smaller problem. After getting rid of $w_1$ and $w_2$ but keep their parent node $w$, the tree is still a valid Huffman encoding tree (verify this yourself). The only difference is that we have a new leaf $w$ with frequency $f(w_1) + f(w_2)$. To minimize the total encoding length of the original tree, of course we should also minimize the total length of this new tree. Due to the equivalence of the first and second formulations, we are really seeking the optimal Huffman encoding tree for the $n - 1$ symbols. And this means we can re-apply the rule of pairing up two smallest frequencies among the $n - 1$ symbols and continue.

This leads to a simple algorithm: at each iteration, pick the two symbols with lowest frequencies and form a subtree with these two symbols; remove these two symbols and create a new symbol with the summation of the these removed symbols as its frequency. Continue until there is only one symbol left. This algorithm needs frequent operations to obtain (and extract) the symbols with lowest frequency. This can be done by a heap, which makes each iteration to take $O(\log n)$ time for $n$ symbols. So the total running time is $O(n \log n)$. Try to run some examples yourself to make sure you understand this algorithm.

Now let us think about what properties of a good greedy strategy has. For greedy algorithm to work, we should be able to make locally optimal choices at the moment, and after making the choice, we need to prove we still always get optimal solution. Second, we need the optimal substructure property: after making a choice, we now have a new (and smaller) problem; the key to greedy algorithm to work is that we can solve the smaller problem and then combine this solution with the decision made at the greedy choice to obtain the final solution. Read our textbook for more detailed explanation.