COL351: Analysis and Design of Algorithms **Fall 2024**

Tutorial Sheet 7

Announced on: Sept 07 (Sat)

Problems marked with ($\star$) will not be asked in the tutorial quiz.

1. This problem describes the *cut property* of minimum spanning trees (MSTs).

   Let $G = (V, E)$ be a connected and weighted graph where the edge costs may not be distinct. For any subset $S \subseteq V$ of the vertices, let $\bar{S} := V \setminus S$ denote the vertices not in S. Let $E(S, \bar{S})$ denote the set of all edges with one endpoint each in S and $\bar{S}$. Show that if there exists a unique edge $e \in E(S, \bar{S})$ with the smallest weight among the edges in $E(S, \bar{S})$, then e must belong to every MST of graph G.

   Proof by Contradiction: Assume that there exists a subset $S \in V$ such that there exists a unique edge $e \in E(S, \bar{S})$ with the smallest weight among the edges in $E(S, \bar{S})$ and e is not part of an MST T of G.
   Now, if we add e to the MST, a unique cycle C gets created. By double crossing lemma, there is another edge $f \in C$ such that $f \in E(S, \bar{S})$. Since e was the unique edge of minimum weight crossing the cut $(S, \bar{S})$, we know that weight of f is more than weight of e. So, we can remove f and keep e which gives a lower weight spanning tree. But this is a contradiction because we supposedly started with a MST, and now we have a collection of edges which is a spanning tree that weighs less. Thus, the original MST was not actually minimal.

2. Consider the following alleged proof of the cut property from Problem 1.

   Let T be a minimum spanning tree that does not contain e. Since T is a spanning tree, it must contain an edge f with one end in S and the other end in $\overline{S}$. Since e is the cheapest edge with this property, we have $c_e < c_f$, and hence $T \setminus \{f\} \cup \{e\}$ is a spanning tree that is cheaper than T.

   Is this proof correct? If not, then explain the error.

   The proof is incorrect.

   Intuitive Idea: On a high level, as mentioned in the Proof of Ques. 1, while adding the edge e, we removed the edge which was a crossing edge of the cut $(S, \bar{S})$ as well as part of the cycle induced in $T \cup \{e\}$. Whereas in the proof given in Ques. 2, we remove any edge crossing the cut $(S, \bar{S})$, not necessarily part of a cycle. Because of this, the resulting graph might not be a spanning tree.

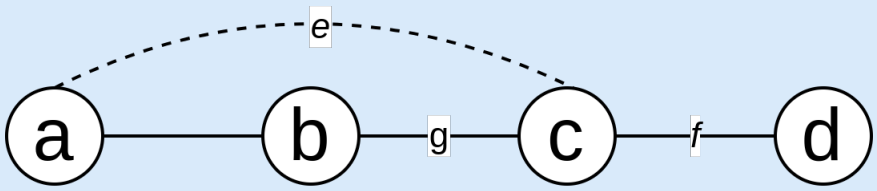

Figure 1: Counter example graph, the solid edges denote the edges of the spanning tree T, and dashed edges denote the edges of the graph not present in T.

As a counterexample, consider the (weighted) graph shown. consider the $S = \{a, b, d\}$ and $\bar{S} = V \setminus S = \{c\}$. The edge weights are such that the edge e is the unique minimum weight edge across the cut $(S, \bar{S})$ Now, f is an edge with one end in S and other edge $\bar{S}$. But, it can be seen that $T \setminus \{f\} \cup \{e\}$ is not a spanning tree and hence, the proof is flawed.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Claiming that the proof is incorrect - 1 point
  - Mentioning the error that f also needs to be part of the cycle formed in $T \cup \{e\}$ (just giving a counterexample for the proof also works) - 2 points

3. Suppose you are given a connected weighted graph $G = (V, E)$ with a distinguished vertex s where all edge costs are positive and distinct. Is it possible for a tree of shortest paths from s and a minimum spanning tree in G to not share any edges? If so, give an example. If not, give a reason.

Consider the shortest path tree T_D generated by taking this distinguished vertex s as the source node. Let e be the minimum weight outgoing edge in T_D from s to some neighboring vertex, say t.

Claim 1. *The edge e is part of every MST of G.*

Proof. When e is removed from T_D, it gives rise to two connected components. Let S and $\bar{S}$ denote the mutually exclusive non-empty vertex sets of these connected components. Thus, e becomes the minimum cost crossing edge between S and $\bar{S}$. As discussed in Question 1, edge e should be part of every MST of G.

□

So, we have shown that at least one edge is common between T_D and any MST of G.

4. Consider a connected undirected graph G with distinct edge costs. Design a linear-time algorithm to find a *minimum bottleneck spanning tree*, which is a spanning tree T that minimizes the maximum edge cost $\max_{e \in T} c_e$. You can assume access to a subroutine `FindMedian` that, given as input an unsorted list of n numbers, computes the median in $\mathcal{O}(n)$ time.

   For the tutorial quiz, you can provide a plain English description of the algorithm as your solution. You don't need to write a complete pseudocode, but you are welcome to include it in addition to or instead of the English description. Additionally, please provide a brief justification (one or two sentences) for the correctness and running time of the algorithm.

   Note: For simplicity, we assume that all edge costs are distinct. The same arguments will apply even when edge costs are not distinct.

   Informal Idea: We use a divide and conquer style algorithm. We partition the edges into two equal halves, such that every edge of right half has larger weight than every edge of left half. Now, we check if we can make a spanning tree from edges of just the left half. If so, we recurse on the edges of the left half, If not, the MBST will contain at least one edge from right half and hence, including any edge from left half does not increase the cost of MBST and hence, all edges of left half can be joined together to form a supernode and we find an MBST in the new graph.

   ALGORITHM 1: Algorithm for calculating the Minimum Bottleneck Spanning Tree (MBST)

   Input: Undirected graph G

   Function `MBST(`G`)`**:**
   - $E \leftarrow$ set of edges of G
   - **if** $|E| = 1$ **then**
     - **return** E
   - **else**
     - $e_m \leftarrow$ Edge of E with median cost
     - $E_> \leftarrow$ Set of edges with cost greater than c_{e_m}
     - $E_\leqslant \leftarrow E - E_>$ ▷ `Edges with cost at most` c_{e_m}
     - **if** $G(E_\leqslant)$ *is connected* **then**
       - **return** $MBST(G(E_\leqslant))$
     - **else**
       - $F \leftarrow$ Set of edges such that each connected component of $G(E_\leqslant)$ is a tree
       - **return** $F \cup MBST(G(E_>)_\eta)$ ▷ $G(E_>)_\eta$ `is the graph with edge set` $E_>$ `and each connected component of` $G(E_\leqslant)$ `is replaced by a super vertex in` $G(E_>)_\eta$

   Time Complexity:

   - Finding the median and dividing the edges into two sets: $O(|E|)$
   - Finding the edge set F: $O(|E|)$
   - At every iteration, the number of edges are halved (assuming distinct edge costs): $T(|E|) = T(\frac{|E|}{2}) + O(|E|)$. By Master Theorem, the overall time complexity is $O(|E|)$.

Proof of Correctness: We will prove by *strong induction* on the number of edges in G:
Inductive Hypothesis $P(n)$: For a connected graph G with $\leqslant n$ edges, the above algorithm outputs the edges of *minimum bottleneck spanning tree*.
Base Case $P(1)$: It can be seen that the base case is correct, since there is only one edge.
Inductive Case $P(n) \implies P(n+1)$:
To show this, we make the following claims:
Claim 2. *If $G(E_{\leqslant})$ is connected then MBST of $G(E_{\leqslant})$ is an MBST of G.*

Proof. The proof is simple, as there exists a spanning tree of G using just the edges of $E_{\leqslant}$ and hence, MBST of G does not use any edge from $E_{>}$. □

Claim 3. *If $G(E_{\leqslant})$ is not connected then union of F and MBST of $(G(E_{>})_{\eta})$ is an MBST of G.*

Proof. This case is also simple, as any spanning tree of G requires at least one edge from $E_{>}$ and hence, MBST of G can contain any edge from $E_{\leqslant}$ without increasing the cost of MBST of G. □

From the above two claims, the inductive case is easy to show (**Try it yourself!**).
Try it yourself: Prove that every Minimum Spanning Tree is an MBST.

- Written "I do not know how to approach this problem" - 0.6 points
- **Note:** No marks will be deducted for assuming have distinct edge costs
  - Algorithm Description - 2 points
    - Finding the edge with median cost - 0.5 points
    - Handling the case if $G(E_{\leqslant})$ is connected - 0.5 points
    - Handling the *else* part - 1 point
  - Justification for correctness - 0.5 points
  - Mentioning the recursive relation to justify $O(E)$ running time - 0.5 points

5. Suppose we are given the minimum spanning tree T of a given graph $G = (V, E)$ with distinct edge costs, and a new edge $e = (u, v)$ of cost c that we will add to G. Design an $\mathcal{O}(|V|)$ algorithm to find the minimum spanning tree of the graph $G + e$. Clearly write the pseudocode and briefly justify (in one or two sentences) why the algorithm is correct and has the desired running time.

ALGORITHM 2: Update Minimum Spanning Tree with New Edge

Data: T: The current MST of graph $G = (V, E)$
$e = (u, v)$: The new edge with cost c
Result: Updated MST

```
Function UpdateMST(T, e):
    T ← T ∪ {e}
    Use DFS/BFS to find the unique cycle C in T
    e' ← Heaviest edge in C
    T ← T \ {e'}
    return T
```

We are given the Minimum Spanning Tree (MST) T of a graph $G = (V, E)$ with distinct edge costs, and a new edge $e = (u, v)$ with cost c is added. The goal is to efficiently compute the new MST for $G + e$ by detecting the cycle formed and removing the heaviest edge if necessary.

We prove this short lemma as a helper:

Lemma 1 (Cycle Property)**.** *For any cycle C in the graph, if the weight of an edge e of C is larger than the weights of all other edges of C, then this edge cannot belong to any Minimum Spanning Tree (MST).*

Proof. Assume the contrary, i.e., that e belongs to an MST T_1. Deleting e will break T_1 into two subtrees, with the endpoints of e in different subtrees. The remainder of C reconnects the subtrees, hence there is an edge f in C with endpoints in different subtrees. This edge f reconnects the subtrees into a new tree T_2 with a weight less than that of T_1 because the weight of f is less than the weight of e.
This contradicts the assumption that T_1 is an MST, thus proving that the edge e cannot belong to an MST. □

The above lemma suggests an alternate algorithm to find an MST of G:

- Initialise $T = G$
- While there exists a cycle in T: Remove the largest edge of the cycle from T.
- return T

Proof Idea: To compute MST of $G + e$, suppose we use the above algorithm and first keep removing largest edges from the cycles **not containing the edge** e. Then, it can be seen that we will be left with the subgraph $T + e$. Now, to obtain the MST of $G + e$, we remove the heaviest edge from the only cycle of $T + e$.

Time Complexity: The algorithm runs in $O(|V|)$ (since number of edges is also $|V|$-1) time since the cycle detection and finding the heaviest edge can both be done using DFS or BFS, which are linear-time operations for a tree.

- Written "I do not know how to approach this problem" - 0.6 points
- **Note:** No points will be deducted for assuming that cost of e is distinct from existing edge costs.
  - Adding e to T and removing the heaviest edge of C - 1 point
  - Claiming that the heaviest edge of a cycle does not belong to any MST - 0.5 points
  - Justifying why can there not exist another spanning tree of $G + e$ which does not have lesser cost than the spanning tree returned by our algorithm - 1 point
  - Justification for $O(V)$ running time - 0.5 points

6. You wish to drive from point A to point B along a highway minimizing the time that you are stopped for gas. You are told beforehand the capacity C of your gas tank in liters, your rate F of fuel consumption in liters/kilometer, the rate r in liters/minute at which you can fill your tank at a gas station, and the locations $A = x_1, x_2, \ldots, x_{n-1}, x_n = B$ of the gas stations along the highway. So, if you stop to fill your tank from 2 liters to 8 liters, you would have to stop for $6/r$ minutes. Consider the following two algorithms:

   a) Stop at every gas station, and fill the tank with just enough gas to make it to the next gas station.

   b) Stop if and only if you don't have enough gas to make it to the next gas station and if you stop fill the tank up all the way.

   For each algorithm, either prove or disprove that the algorithm correctly solves the problem. Your proof of correctness must use an exchange argument.

   a) Let G represent the greedy algorithm defined in the problem, and let OPT denote an optimal algorithm. Assume, for the sake of contradiction, that the greedy algorithm G is not optimal, which implies $t_G > t_{OPT}$, where t_G and t_{OPT} are the total times taken by G and OPT, respectively.

   In our greedy solution G, the car stops at all gas stations $x_1, x_2, \ldots, x_n$, refueling v_j liters at each station x_j. In contrast, the optimal solution OPT stops at fewer gas stations: $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$, where $m < n$, and refuels u_j liters at each x_{i_j}.

   The case $m = n$ is not possible, as it would imply that OPT is identical to G (Hint: compare u_i and v_i for any i).

   Now, consider the case where $m < n$, meaning OPT stops at fewer stations than G. Let k be the first index where $(x_{i_k}, u_k) \neq (x_k, v_k)$. At this index, $x_{i_k} = x_k$ (Hint: Use the fact that $(x_{i_{k-1}}, u_{k-1}) = (x_{k-1}, v_{k-1})$). Additionally, we must have $u_k > v_k$, because if $u_k \leqslant v_k$, the car would not have enough fuel to reach x_{k+1}, as

v_k represents the minimum fuel required to make it to x_{k+1}.

We can now modify the optimal solution OPT by replacing the stop (x_{i_k}, u_k) in OPT with the pair of stops (x_k, v_k) and $(x_{k+1}, u_k - v_k)$. This adjustment does not increase the total time of the solution. If $x_{i_{k+1}} = x_{k+1}$ in OPT, we can further combine the stops $(x_{k+1}, u_k - v_k)$ and $(x_{i_{k+1}}, u_{k+1})$ into a single stop at x_{k+1}, refueling $u_k - v_k + u_{k+1}$ liters.

By making this transformation, we reduce the number of differences between OPT and G by one, without increasing the time required by OPT. Repeating this process iteratively for all remaining differences will eventually transform OPT into G, implying $t_{OPT} = t_G$, which contradicts our initial assumption that $t_G > t_{OPT}$.

Therefore, the greedy algorithm G is indeed optimal.

b) Consider the following counterexample involving three gas stations, x_1, x_2, x_3, located along a highway:

$$\begin{aligned} C &= 150 \text{ liters}, \\ F &= 1 \text{ liter/kilometer}, \\ r &= 1 \text{ liter/minute}, \\ x_1 &= 0 \text{ km}, \\ x_2 &= 100 \text{ km}, \\ x_3 &= 200 \text{ km}. \end{aligned}$$

If we stop at x_2 and fill the tank completely with 100 liters (to refill from the current level of 50 liters), the time required to refuel will be:

$$t = \frac{100}{r} = 100 \text{ minutes}.$$

However, if we instead refill only the amount needed to reach x_3 (i.e., 50 liters), the refueling time would be:

$$t' = \frac{50}{r} = 50 \text{ minutes}.$$

Since $t' = 50$ minutes $< t = 100$ minutes, this demonstrates that the algorithm which always fills the tank completely is not optimal, as refueling only what is necessary can reduce the total time spent.

7. ($\star$) In tutorial sheet 6, you saw an example where the "top down" divide-and-conquer strategy for the optimal prefix-free code problem is suboptimal. You have also seen in class that the Huffman coding algorithm is optimal.

However, you love divide-and-conquer algorithms and can't stop thinking about them. So instead, you devise the following proposal: Define a *median* symbol whose frequency, along with the frequencies of all other symbols with higher (respectively, lower) frequency, adds up to at least 50% of the total. That is, if the frequencies in ascending order are $p_1, p_2, \ldots, p_n$, then p_ℓ is the frequency of the median symbol if

$$\sum_{i=1}^{\ell} p_i \geqslant \frac{1}{2}\sum_{i=1}^{n} p_i \text{ and } \sum_{i=\ell}^{n} p_i \geqslant \frac{1}{2}\sum_{i=1}^{n} p_i.$$

At each step, your algorithm divides the current set of symbols into two parts, namely, those that are more (respectively, less) frequent than the median, and assigns the median to the "underdog" part, i.e., the part with the smaller total (breaking ties arbitrarily).

Prove or disprove: The above algorithm returns an optimal prefix-free code.

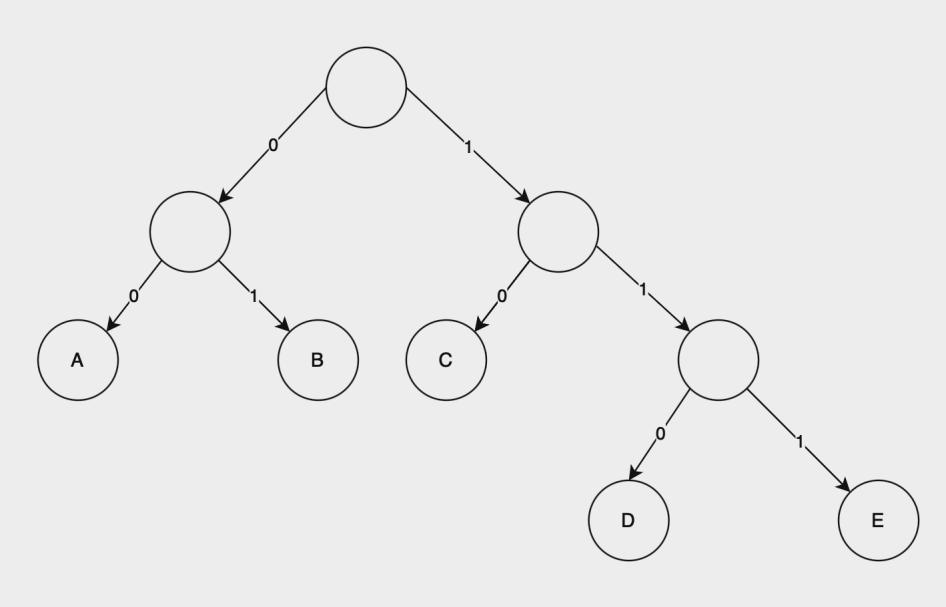

Figure 2: Σ-tree for above algorithm

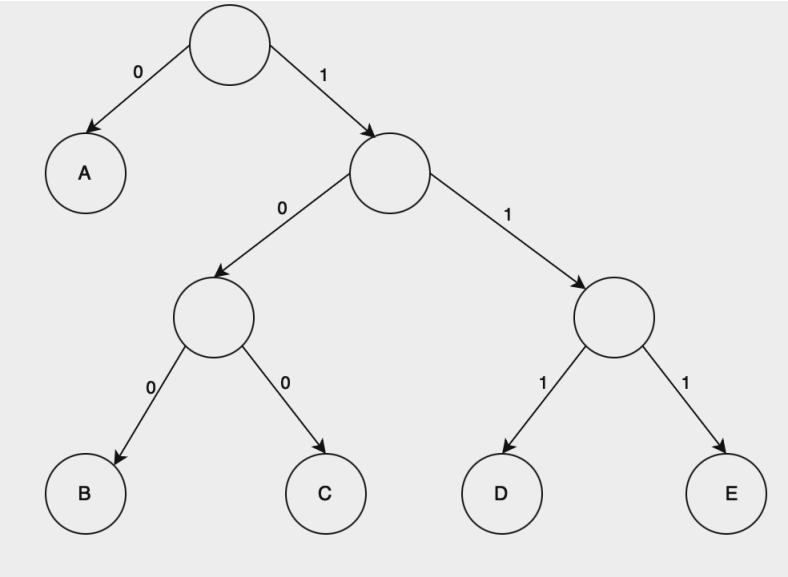

Figure 3: Σ-tree for Huffman

Consider $\Sigma = \{A, B, C, D, E\}$ with frequencies $\{\frac{7}{17}, \frac{3}{17}, \frac{3}{17}, \frac{3}{17}, \frac{1}{17}\}$. The average leaf depth of the constructed Σ tree using the above algorithm is:

$$\frac{1}{17} \times 3 + \frac{3}{17} \times 3 + \frac{3}{17} \times 2 + \frac{3}{17} \times 2 + \frac{7}{17} \times 2 = \frac{38}{17}$$

The average leaf depth of the constructed Σ tree using the huffman algorithm is:

$$\frac{1}{17} \times 3 + \frac{3}{17} \times 3 + \frac{3}{17} \times 3 + \frac{3}{17} \times 3 + \frac{7}{17} \times 1 = \frac{35}{17}.$$

Hence, the above algorithm is not optimal.

8. ($\star$) Suppose you are given a connected graph $G = (V, E)$ with a cost c_e on each edge e. In class, we saw that when all edge costs are distinct, G has a unique minimum spanning tree. However, G may have many minimum spanning trees when the edge costs are not all distinct. Here we formulate the question: Can Kruskal's Algorithm be made to find *all* the minimum spanning trees of G?

   Recall that Kruskal's Algorithm sorted the edges in order of increasing cost, then greedily processed the edges one by one, adding an edge e as long as it did not form a cycle. When some edges have the same cost, the phrase "in order of increasing cost" has to be specified a little more carefully: We'll say that an ordering of the edges is *valid* if the corresponding sequence of edge costs is nondecreasing. We'll say that a *valid execution* of Kruskal's Algorithm is one that begins with a valid ordering of the edges of G.

For any graph G, and any minimum spanning tree T of G, is there a valid execution of Kruskal's Algorithm on G that produces T as output? Give a proof or a counterexample.

Intuition: To obtain an MST T from Kruskal's Algorithm, we construct an ordering in which we prefer edges present in T. Then we can prove that this ordering would give us the tree T by using induction.

Formally, for an MST T if we want the following condition to hold: For every two edges e_1 and e_2, if they have the same costs and e_1 is in T while e_2 is not in T, then e_1 should appear before e_2 in the ordering. If we run Kruskal's Algorithm on a valid ordering which satisfies the above criteria, we claim that we would get the MST T.

Notation:

- We denote a valid ordering $O = \{e_1, e_2, \ldots, e_m\}$, with $O[i] = e_i$ as the indexing operation.
- Let O_T be a valid ordering which satisfies the condition stated above for an MST T.
- Let index_till(c, O) denote the index i in a valid ordering O such that $\forall_{j<i}.\, c_{O[j]} < c$ and $\forall_{j \geqslant i}.\, c_{O[j]} \geqslant c$

Induction Hypothesis: During the run of Kruskal on an ordering O_T, when we are at the index index_till(c, O_T), then we would have added exactly the edges to the tree which have cost less than c and appear in T.

Base case: For $c = \min_{i \in m}(c_{e_i})$, index_till$(c, O_T) = 0$, there are no edges with cost less than c and we wouldn't have added any edges to the tree either (because we are at the start of the algorithm), hence the hypothesis is satisfied.

Inductive Case: Suppose the hypothesis holds for all values of c less than c^*. We claim that after processing all the edges in O_T with weight c^* we would end up with exactly the edges which have weight $\leqslant c^*$ and are also in T.

Lets consider the case where there exists at least one vertex in E with weight c^*, as the other case is not interesting. Now let $i =$ index_till(c^*, O_T), which means $c_{O_T[i]} = c^*$.

a) Case 1: If there doesn't exist an edge in T with weight c^* then we prove that adding any edge with the weight c^* to the current tree will create a cycle. This is true because of our induction hypothesis saying that the MST created so far is the same as T, so if we can add an edge with weight c^* to it then we can replace an edge in T with weight larger than c^* with this edge, and thus contradicting the fact that T is an MST.

b) Case 2: Let j be the largest index such that $O_T[j] \in T$ and $c_{O_T[j]} = c^*$. According to the constraints on O_T, we know that all edges $O_T[i], O_T[i+1], \ldots, O_T[j]$ are

in T. We claim that all of these edges can be added to the current tree without causing a cycle, as if the cycle exits in this tree then the same cycle will exist in T. Also all edges in O_T after j with weight c^* will not be added to the tree by Kruskal's algorithm. This is because after index j if we can add another edge with weight c^* without causing a cycle then we can replace an edge in T with larger weight than c^*, and thus contradict it being an MST.

Hence we prove that the induction hypothesis holds, which means that at the end of the algorithm we would obtain exactly the same tree T. This proves that for every MST T there exists an ordering which would make Kruskal's algorithm output T.