Homework 3 — Greedy Algorithms

Due: November 3rd

A few of these questions may appear on or help you prepare for the quiz on **November 3rd**.
Note that these selected solutions and comments do not necessarily represent complete answers.

1. *Pruning* removes elements from an instance of a problem if they cannot possibly be part of some desired solution. Consider the "reverse delete" variation of Kruskal's algorithm, which greedily prunes edges:

   ReverseDelete($G = (V, E)$)

   Input: A connected, weighted graph G
   Output: A minimum spanning tree of G

   ```
   let A be E, sorted in decreasing order by weight
   for all e ∈ A do
       if removing e from E would not disconnect G then
           Remove e from E
   return G
   ```

   Using the Cut Property and the Cycle Property, briefly justify the correctness of this algorithm.

   ReverseDelete removes the first edge considered along each cycle. Since edges are considered in descending order by weight, such edges must be among the heaviest in a cycle; by the Cycle Property, they ought to be excluded from an MST.

   ReverseDelete keeps the last edge considered in each cut set. Since edges are considered in descending order by weight, such edges must be among the lightest in a cut set; by the Cut Property, they ought to be included in an MST.

2. Consider the following statement:

   Proposition: Let $G = (V, E)$ be a connected, weighted graph. If all edge weights are distinct, then G has a unique minimum spanning tree.

   Give a proof of or a counterexample to this statement.

   Proof:

   - *Let $G = (V, E)$ be a connected, weighted graph such that all edge weights are distinct. Suppose, for contradiction, that G has two distinct MSTs, T_1 and T_2.*
   - *Then there must exist an edge e that is in T_1 but not in T_2. Since T_2 is a spanning tree, adding e to T_2 creates a cycle.*
   - *Since all edge weights are distinct, one of the edges traversed by that cycle must be strictly heavier than the others. By the Cycle Property, that heaviest edge cannot be in any MST, however, it must be in either T_1 or T_2. This contradicts the assumption that T_1 and T_2 were MSTs.*
   - *Therefore, by contradiction, if all edge weights are distinct, then G has a unique MST.* □

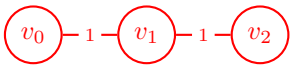
3. Consider the following statement:

   Proposition: Let $G = (V, E)$ be a connected, weighted graph. If G has a unique minimum spanning tree, then all edge weights are distinct.

   Give a proof of or a counterexample to this statement.

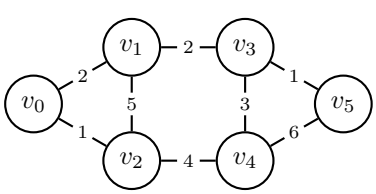
   Counterexample:

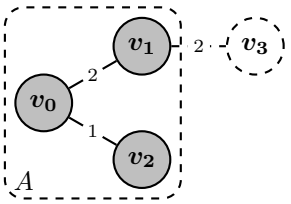
   - *Consider the following graph:*

   - *Then there exists a unique MST, namely G itself, but there are two edges of weight 1.*

4. Recall that Dijkstra's algorithm greedily traverses the next lightest path leading to an unexplored vertex. For example, given $s = v_0$ and:

   …after 3 iterations, ShortestPath has computed distances to $A = (v_0, v_2, v_1)$, in that order, and is about to traverse the edge (v_1, v_3) in order to compute the distance to v_3.

   Lemma: Suppose ShortestPath returns distances $\text{dist}(v)$ and there exist optimal distances $\text{Opt}(v)$. Further suppose that $A = (a_0, a_1, \ldots, a_{n-1})$ is V, sorted in the order that distances were computed by ShortestPath. For all $i < n$, $\text{dist}(a_i) = \text{Opt}(a_i)$.

   Prove the above "greedy stays ahead" lemma, supposing that all weights are non-negative.

   Proof:

   - *Let $G = (V, E)$ be a weighted graph such that all weights are non-negative. Further, let $A = (a_0, a_1, \ldots, a_{n-1})$ be V, sorted in the order that distances were computed by ShortestPath.*
   - Basis Step:
     - *Let $i = 0$. Since $a_0 = s$, by definition, $\text{dist}(a_0) = \text{Opt}(a_0) = 0$.*
   - Inductive Hypothesis:
     - *Suppose that, for all $0 \leq i \leq k$, $\text{dist}(a_i) = \text{Opt}(a_i)$.*
   - Inductive Step:
     - *Let $i = k + 1$, and let $(a_0, \ldots, a_j, a_i)$ be the path found by ShortestPath. Consider any alternative path $(a_0, \ldots, a_x, a_y, \ldots, a_i)$, where $x < i$ and $y \geq i$. That is, ShortestPath found paths $(a_0, \ldots, a_j)$ and $(a_0, \ldots, a_x)$, and could have found a path to a_y next instead of to a_i.*
     - *Since ShortestPath considers paths in increasing order by length, it must be the case that $\text{dist}(a_j) + w_{ji} \leq \text{dist}(a_x) + w_{xy}$, where $j \leq k$ and $x \leq k$. By the hypothesis, $\text{dist}(a_j) = \text{Opt}(a_j)$ and $\text{dist}(a_x) = \text{Opt}(a_x)$, thus, $\text{Opt}(a_j) + w_{ji} \leq \text{Opt}(a_x) + w_{xy}$.*
     - *That is, the path $(a_0, \ldots, a_j, a_i)$ is no longer than the path $(a_0, \ldots, a_x, a_y)$. Since all edge weights are non-negative, the path $(a_0, \ldots, a_j, a_i)$ cannot possibly be longer than any alternative path $(a_0, \ldots, a_x, a_y, \ldots, a_i)$. Thus, $\text{Opt}(a_i) = \text{Opt}(a_j) + w_{ji} = \text{dist}(a_j) + w_{ji} = \text{dist}(a_i)$.*
   - *Therefore, by the Principle of Mathematical Induction, for all $i < n$, $\text{dist}(a_i) = \text{Opt}(a_i)$.* □

5. Consider the following statement:

   Proposition: Let S be a set of activities, each taking place between a start time and a finish time, $[s_i, f_i)$. If a_i is the activity with the strictly earliest finish time in S, then a_i must be in every maximum subset of non-conflicting activities.

   Give a proof of or a counterexample to this statement.

   Consider: just because something is always an *optimal choice does not necessarily mean that it is always* the *only optimal choice. If a_i finishes strictly earlier than some a_j, but no other activity starts between f_i and f_j, then a_j could be an equally optimal choice.*

6. Consider the following statement:

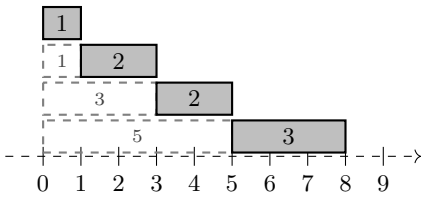
   Proposition: Let S be a set of activities, each taking place between a start time and a finish time, $[s_i, f_i)$. If a_i is the activity with the strictly latest finish time in S, then a_i cannot be in any maximum subset of non-conflicting activities.

   Give a proof of or a counterexample to this statement.

   Consider: just because one choice would not be optimal compared to another choice does not necessarily mean that there exists another choice at all. If a_i finishes strictly later than any a_j, but all f_j are before s_i, then there is no alternative choice to a_i.

7. Suppose that you are given a finite set of n customers $S = \{t_0, t_1, \ldots, t_{n-1}\}$, where each customer will require t_i time to serve. Further suppose that no two customers can be served simultaneously, thus, customer t_i must wait $\sum_{j=0}^{i-1} t_j$ time: the time required to serve all of the customers served previously.

   For example, given $S = \{3, 1, 2, 2\}$:

   …serving the customers in the order $(1, 2, 2, 3)$ causes them to collectively wait $1 + 3 + 5 = 9$ time.

   (a) Give pseudocode for a greedy algorithm that finds the minimum total waiting time of all customers.

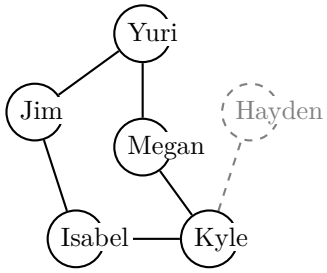
   (b) Briefly explain your algorithm's greedy assumption and why it is safe to make.

   Note that serving faster customers first causes the remaining customers to wait less. In contrast, serving slower customers first causes the remaining customers to wait more.

   (c) Give the time complexity of your algorithm.

8. Suppose that you are given a finite set of n people and you wish to host a party, to which you can invite some or all of the n people. Further suppose that you would like every partygoer to meet at least 2 people whom they do know and at least 2 people whom they do not.

   For example, given:



   …Hayden cannot be invited, since they will not meet 2 people whom they already know.

   (a) Give pseudocode for a greedy algorithm that finds the maximum subset of invitable people.

   (b) Briefly explain your algorithm's greedy assumption and why it is safe to make.

   Note that a person who knows too few people can safely be uninvited, as uninviting others cannot cause that person to know enough invitees. In contrast, a person who knows enough people cannot be safely invited, as uninviting others might cause that person to know too few invitees.

   (c) Give the time complexity of your algorithm.