Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 MST practice

Let $G = (V, E)$ be an undirected, connected graph.

(a) Prove that there is a unique MST if all edge weights are distinct.

(b) True or False? If G has more than $|V| - 1$ edges, and there is a unique heaviest edge, then this edge cannot be part of a MST.

(c) True or False? If the lightest edge in G is unique, then it must be a part of every MST.

Solution:

(a) Suppose the graph has two different MSTs T_1 and T_2. Let e be the lightest edge present in just one of the trees. Without loss of generality, say $e \in T_1$. Thus, adding e to T_2 gives a cycle. However, this cycle must contain an edge e' not in T_1 which connects the two components of $T_1 - e$. Since e is the lightest edge in just one of the trees, e is lighter than e'. Thus, we can add e to T_2 and remove e' to get a better MST. This contradicts our claim of two different MSTs. Thus, there is only one unique MST in this graph.

(b) False; consider the case when removing the heaviest edge disconnects the graph. Then any spanning tree must include the heaviest edge in the tree.

(c) True; if the lightest edge e is not part of some MST, there exists a cycle connecting the two endpoints of e, so adding e and removing another edge of the cycle produces a lighter tree, a contradiction.

2 Finding Counterexamples

In this problem, we give example greedy algorithms for various problems, and your goal is to find a counterexample where they do not find the best solution.

(a) In the travelling salesman problem, we have a weighted undirected graph $G(V, E)$ with all possible edges. Our goal is to find the cycle that visits all the vertices exactly once with minimum length.

One greedy algorithm is: Build the cycle starting from an arbitrary start point s, and initialize the set of visited vertices to just s. At each step, if we are currently at vertex u and our cycle has not visited all the vertices yet, add the shortest edge from u to an unvisited vertex v to the cycle, and then move to v and mark v as visited. Otherwise, add an edge from the current vertex to s to the cycle, and return the now complete cycle.

(b) In the maximum matching problem, we have an undirected graph $G(V, E)$ and our goal is to find the largest matching E' in E, i.e. the largest subset E' of E such that no two edges in E' share an endpoint.

One greedy algorithm is: While there is an edge $e = (u, v)$ in E such that neither u or v is already an endpoint of an edge in E', add any such edge to E'. (Challenge: Can you prove that this algorithm still finds a solution whose size is at least half the size of the best solution?)

Solution:
Note: For each part, there are many counterexamples.

(a) One counterexample is to have a four vertex graph with vertices a, b, c, d, where the edges $(a, b), (b, c), (c, d)$ cost 1, the edge (a, d) costs 100, and all other edges cost 2. If the greedy algorithm starts at vertex a, it will add the edges $(a, b), (b, c), (c, d)$ to the cycle, and then be forced to add the very expensive edge (a, d) at the end to find a cycle of cost 103. The optimal cycle is $(a, b), (b, d), (d, c), (c, a)$ which costs 6. The key idea here is that by using a path of low-weight edges, we forced the algorithm into a position where it had to pick a high-weight edge to complete its solution.

(b) The simplest example is a path graph with three edges. The greedy algorithm may pick the middle edge, the optimal solution is to pick the two outer edges.

To show this algorithm always finds a matching at least half the size of the best solution, let the size of the solution found by the algorithm be m. Any edge in the best solution shares an endpoint with one of the edges in the algorithm's solution (otherwise, the greedy algorithm could have added it). None of the edges in the best solution can share an endpoint, and the m edges in the algorithm's solution have $2m$ endpoints, so the best solution must have at most $2m$ edges.

3 Longest Huffman Tree

Under a Huffman encoding of n symbols with frequencies $f_1, f_2, \ldots, f_n$, what is the longest a codeword could possibly be? Give an example set of frequencies that would produce this case, and argue that it is the longest possible.

Solution: The longest codeword can be of length $n-1$. An encoding of n symbols with $n-2$ of them having probabilities $1/2, 1/4, \ldots, 1/2^{n-2}$ and two of them having probability $1/2^{n-1}$ achieves this value. No codeword can ever by longer than length $n-1$. To see why, we consider a prefix tree of the code. If a codeword has length n or greater, then the prefix tree would have height n or greater, so it would have at least $n+1$ leaves. Our alphabet is of size n, so the prefix tree has exactly n leaves.

4 Activity Selection

Assume there are n activities each with its own start time a_i and end time b_i such that $a_i < b_i$. All these activities share a common resource (think computers trying to use the same printer). A feasible schedule of the activities is one such that no two activities are using the common resource simultaneously. Mathematically, the time intervals are disjoint: $(a_i, b_i) \cap (a_j, b_j) = \emptyset$. The goal is to find a feasible schedule that maximizes the number of activities k.

Here are two potential greedy algorithms for the problem.

Algorithm A: Select the shortest-duration activity that doesn't conflict with those already selected until no more can be selected.

Algorithm B: Select the earliest-ending activity that doesn't conflict with those already selected until no more can be selected.

(a) Show that Algorithm A can fail to produce an optimal output.

(b) Show that Algorithm B will always produce an optimal output. (Hint: To prove correctness, show how to take any other solution S and repeatedly swap one of the activities used by Algorithm B into S while maintaining that S has no overlaps)

(c) **Challenge Problem:** Show that Algorithm A will always produce an output at least half as large as the optimal output.

Solution:

(a) There are many examples that work, but here is a simple one, easiest explained pictorally (each line represents an activity):

The optimal strategy is to pick the two longer activities, but strategy A picks the shorter activity, and then cannot pick another.

(b) Consider any optimal schedule S, and suppose the first i activities in S are the same as the first i activities chosen by Algorithm B (i might be zero). If both schedules have i activities, Algorithm B's solution is the same as S, so the proof is done. Otherwise, both Algorithm B's solution and the optimal solution must have a $(i+1)$st activity: If Algorithm B's solution is a subset of S, it would have added one of the remaining activities in S, and if Algorithm B's solution has more than i activities but S only has i activites, S is not optimal. In either case, we have a contradiction.

Then, consider swapping the $(i+1)$st activity in the optimal solution with the $(i+1)$st activity in our solution. By definition of Algorithm B, its $(i+1)$st activity ends at least as early as the $(i+1)$st activity in S. This means it can't overlap with later activities in S, so the feasibility and size of S are maintained by this swap. This means that given any optimal schedule where the first i activities are the same as in Algorithm B's solution, we can find an optimal schedule where the first $i+1$ activities are the same as in Algorithm B's solution. Starting from any optimal schedule and applying this repeatedly, we find an optimal schedule which is exactly the same as in Algorithm B's solution.

(c) Let I_{opt} be any optimal set of activities and I be the activities chosen by the shortest-duration greedy strategy.

We show the following:

(a) Every activity in I_{opt} overlaps at least 1 activity in I.

Suppose there exists activities in I_{opt} that does not overlap with any activity in I. Let $x \in I_{opt}$ be such an activity of shortest duration. If x had shorter duration than any other activities in I, it would have been added before the others were; contradiction. Otherwise, x would have been added just after all those in I are added; contradiction.

(b) Any activity in I can overlap at most 2 activities in I_{opt}.

Let I_k be the set of activities added by the greedy strategy just after the kth iteration. We show by induction that any activity in I_k overlaps with at most 2 activities in I_{opt}.

- Base case: $I_0 = \emptyset$; vacuously true.
- Inductive case: Assume true for I_k and let x be the activity added during the $k+1$th iteration. If $x \in I_{opt}$, we are done. Otherwise, suppose for the purpose of contradiction that x overlapped with strictly more than two activities in I_{opt}. Then it must be that one such activity begins and ends while x is still active; contradiction.

The original claim follows immediately as a corollary.

The two claims above imply $|I_{opt}| \leq 2|I|$.

5 Doctor

A doctor's office has n customers, labeled $1, 2, \ldots, n$, waiting to be seen. They are all present right now and will wait until the doctor can see them. The doctor can see one customer at a time, and we can predict exactly how much time each customer will need with the doctor: customer i will take $t(i)$ minutes.

(a) We want to minimize the average waiting time (the average of the amount of time each customer waits before they are seen, not counting the time they spend with the doctor). What order should we use? You do not need to justify your answer for this part. (Hint: sort the customers by ____)

(b) Let $x_1, x_2, \ldots, x_n$ denote an ordering of the customers (so we see customer x_1 first, then customer x_2, and so on). Prove that the following modification, if applied to any order, will never increase the average waiting time:

- If $i < j$ and $t(x_i) \geq t(x_j)$, swap customer i with customer j.

(For example, if the order of customers is $3, 1, 4, 2$ and $t(3) \geq t(4)$, then applying this rule with $i = 1$ and $j = 3$ gives us the new order $4, 1, 3, 2$.)

(c) Let u be the ordering of customers you selected in part (a), and x be any other ordering. Prove that the average waiting time of u is no larger than the average waiting time of x—and therefore your answer in part (a) is optimal.

Hint: Let i be the smallest index such that $u_i \neq x_i$. Use what you learned in part (b). Then, use proof by induction (maybe backwards, in the order $i = n, n-1, n-2, \ldots, 1$, or in some other way).

Solution:

(a) Sort the customers by $t(i)$, starting with the smallest $t(i)$.

(b) First observe that swapping x_i and x_j does not affect the waiting time customers $x_1, x_2, \ldots, x_i$ or customers $x_{j+1}, x_{j+1}, \ldots, x_n$ (i.e., for customers x_k where $k \leq i$ or $k > j$). Therefore we only have to deal with customers $x_{i+1}, \ldots, x_j$, i.e., for customer k, where $i < k \leq j$. For customer x_k, the waiting time before the swap is

$$T_k = \sum_{1 \leq l < k} t(x_l),$$

and the waiting time after the swap is

$$T'_k = \sum_{1 \leq l < i} t(x_l) + t(x_j) + \sum_{i < l < k} t(x_l) = T_k - t(x_i) + t(x_j).$$

Since $t(x_i) \geq t(x_j)$, $T'_k \leq T_k$, so the waiting time is never increased for customers $x_{i+1}, \ldots, x_j$, hence the average waiting time for all the customers will not increase after the swap.

(c) Let u be the ordering in part (a), and x be any other ordering. Let i be the smallest index such that $u_i \neq x_i$. Let j be the index of x_i in u, i.e, $x_i = u_j$ and k be the index of u_i in x. It's easy to see that $j > i$. By the construction of x, we have $T(x_i) = T(u_j) \geq T(u_i) = T(x_k)$, therefore by swapping x_i and x_k, we will not increase the average waiting time. If we keep doing this, eventually we will transform x into u. Since we never increase the average waiting time throughout the process, u is the optimal ordering.