Midterm 2

- **The exam has 4 questions, is worth 100 points, and will last 120 minutes.**
- The exam has three parts. In part A solve all questions. In both parts B and C, you'll need to solve exactly one out of two options. For each part, please indicate which option you chose.
- We indicated how points are allocated to different parts of questions. Not all parts of a problem are weighted equally.
- Read the instructions and the questions carefully first.
- Begin each problem on a new page.
- Be precise and concise.
- The questions start with true/false, then short answer, then algorithm design. The problems may **not** necessarily follow the order of increasing difficulty.
- Good luck!

Part A (55 pts) – Answer all of the following three questions:

0 Logistics and Self-Reflect (1 pt)

(0.5 pt) Are you recording the midterm? Did you provide a link so that staff can login to the room if needed?
(0.5 pt) How well did you do in the exam? Please write your estimate of your total score.

1 True/False + short justification: MSTs + Max-Flow (24 pts)

(3 pts per item) The first six items involve minimum spanning trees (MSTs). Given a weighted undirected connected graph $G = (V, E)$, are the following **true** or **false**? **If true, provide a brief justification (1-3 sentences); if false, provide a counterexample**. In the following, a **non-trivial cut** would refer to a partition $(S, V \setminus S)$ of the vertices, where both S and $V \setminus S$ are **non-empty**.

(a) For any spanning tree T and any non-trivial cut $(S, V \setminus S)$, T contains *at least* one edge crossing this cut.

True. Otherwise, vertices in S can't be connected to vertices in $V \setminus S$.

(b) For any spanning tree T and any non-trivial cut $(S, V \setminus S)$, T contains *at most* one edge crossing this cut.

False. e.g. consider the three-vertex path graph a-b-c. The cut $(\{b\}, \{a, c\})$ has two edges, both in the only spanning tree.

(c) For any non-trivial cut $(S, V \setminus S)$, suppose e crosses this cut and has strictly smaller weight than every other edge crossing this cut. Then any minimum spanning tree must contain e.

True. This is just the cut property. Let's spell it out: Suppose not. Then, there exists a minimum spanning tree T not containing e. We can add e to this tree. This creates a cycle containing another edge e' besides e across this cut that can be removed leaving $T' = T \cup \{e\} \setminus \{e'\}$ connected. But then T' is a spanning tree with smaller weight – a contradiction.

(d) For any non-trivial cut $(S, V \setminus S)$, suppose e crosses this cut and there is another edge with strictly smaller weight than e crossing the cut. Then no minimum spanning tree contains e.

False. e.g. consider again the three-vertex path graph a-b-c, where (a, b) has weight 1 and (b, c) has weight 2. (b, c) is heavier than (a, b) and both cross the cut $(\{b\}, \{a, c\})$, but (b, c) is in the only spanning tree.

(e) For any cycle in the graph, suppose e is in this cycle and has strictly larger weight than every other edge in this cycle. Then no minimum spanning tree contains e.

True. Consider any spanning tree containing e. We can delete e, and some other edge in this cycle can be added to reconnect the tree (but reduce its cost).

(f) For any two vertices s, t, the shortest path from s to t is contained in every minimum spanning tree.

False. Consider a complete three-vertex graph with edge weights $101, 102, 103$. The edge of weight 103 is a shortest path between two vertices but isn't in the minimum spanning tree.

The next couple of items involve Min-Cut-Max-Flow. Again, for each statement, answer whether this statement is true or false, and provide a **short justification/counterexample**. Let $G = (V, E)$ be a network with capacities c and with s-t max-flow with value f. Let $(S, V \setminus S)$ be a minimal s-t cut in the network, and let e be an edge crossing this cut.

(g) If we **increase** the capacity of e by 1 then we **increase** the maximal flow of the network by 1. False. Take a path of length 2 from s to t, $s \to u \to t$ with capacities 1 on both edges. The cut $(\{s\}, \{u, t\})$ is an s-t-min-cut but increasing the capacity of the edge (s, u) by 1 would not increase the max-flow since there's another cut $(\{s, u\}, \{t\})$ with the same capacity.

(h) If we **decrease** the capacity of e by 1 then we **decrease** the maximal flow of the network by 1. True. This cut's value decreases to $f - 1$, so the max flow value must decrease by 1 as well.

2 Max Flow, Zero-sum Games & Linear Programming (30 pts)

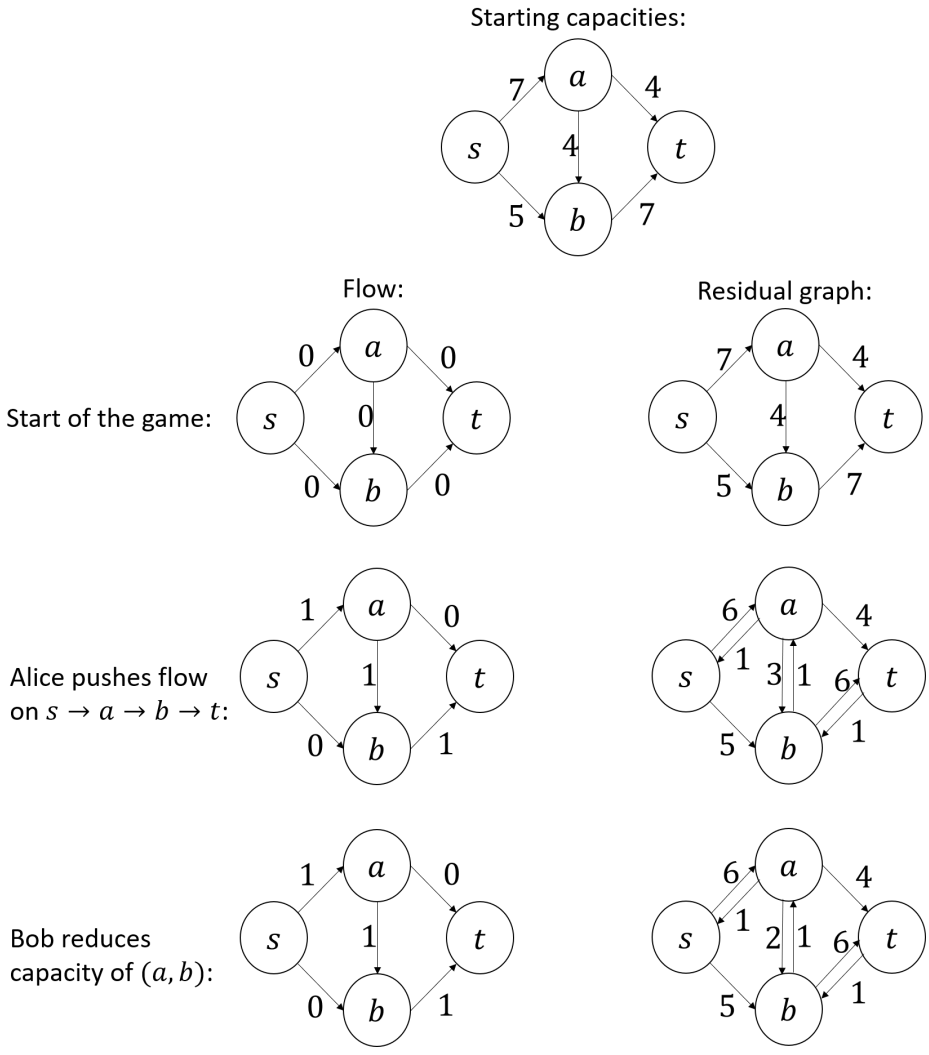
(a) Max Flow (10 points):

Alice and Bob are playing a game. They start with a directed graph G with capacities c_e, for which F is the value of the maximum s-t flow in G. All capacities are integers.

Before the first round of the game, no flow has been pushed. In each round: Alice chooses an s-t path in the residual graph, pushes one unit of flow on this path, and updates the residual graph. Then, Bob chooses an edge in the original graph whose capacity is larger than the current flow through it, decreases its capacity by 1, and updates the residual graph.

The game ends when Alice cannot choose a path in the residual graph, or Bob cannot choose an edge that is not at capacity.

For example, Alice and Bob's first turns might look like this:



If Alice plays perfectly, what is the minimum flow she is guaranteed to push? If Bob plays perfectly, what is the maximum flow he can limit Alice to pushing? Justify your answer.

Bob's optimal strategy is to select a min-cut and in each step reduce the capacity of some edge in that cut by 1. Alice's optimal strategy is to pick an arbitrary path in the residual graph, and push a unit of flow. This pair of strategies will meet in the middle, with Alice guaranteed to push $\lceil F/2 \rceil$ units of flow, and Bob guaranteeing that Alice pushes no more than this.

To see this, notice that Alice can push at most k units of flow in k steps, and in that time Bob's strategy reduces the min cut from F to $F - k$. Therefore Bob can guarantee that Alice pushes no more than $\lceil F/2 \rceil$ units of flow. Conversely, Bob can reduce the min cut by at most 1 in each step, and Alice can push a unit of flow as long as the min cut does not become 0.

Note: If you cannot solve this problem, for **half** of the points solve the following problem: Suppose someone presents you with a solution to a s-t max-flow problem on some network. Give a linear time ($O(|V| + |E|)$) algorithm to determine whether the solution does indeed give a maximum flow.

Recall that a flow is a max-flow iff there is no path from s to t in the residual graph. Computing the residual graph, given the flow, can be carried out in linear time (just process each edge of the graph). To test whether there is a path from s to t just perform a DFS from s. This takes linear time.

(b) Zero-sum games + LP (20 points):

Alice and Bob are playing a two-person zero-sum game described by the payoff matrix:

$$\begin{bmatrix} +1 & -3 \\ -2 & +4 \end{bmatrix}$$

Here Alice is the row player and Bob the column player, and the entry of the matrix corresponds to the payoff for Alice. For this problem, you may denote Alice and Bob's strategies as $(p, 1-p)$ and $(q, 1-q)$, respectively, where p denotes the probability that Alice plays the first row, etc. You are encouraged to justify your solution to both parts to maximize your chances of getting partial credit.

i) Write down an LP for finding Alice's optimal strategy. Hint: you might find it helpful to first write down the max-min condition satisfied by Alice's optimal strategy.

$$\begin{aligned} \max z \quad & \text{s.t.} \\ & z \le p - 2(1-p) \\ & z \le -3p + 4(1-p) \\ & 0 \le p \le 1 \\ & z \in \mathbb{R} \end{aligned}$$

ii) Write down the dual to the LP from part i).

Approach 1: Note that the dual is just the optimization problem from Bob's perspective:

$$\begin{aligned} \min w \quad & \text{s.t.} \\ & w \ge q - 3(1-q) \\ & w \ge -2q + 4(1-q) \\ & 0 \le q \le 1 \end{aligned}$$

Approach 2: One can find the dual out manually. We can rewrite Alice's LP as:

$$\begin{aligned} \max z \quad & \text{s.t.} \\ & z - 3p \le -2 \\ & z + 7p \le 4 \\ & p \le 1 \\ & p \ge 0 \end{aligned}$$

Let a, b, c be the variables for the first, second, third constraint. Our objective is $\min -2a + 4b + c$. The constraint for z is $a + b = 1$ (approach 3 implies this as follows: since z is not restricted to non-negative values. This can be obtained by setting $z = z^+ - z^-$, and writing the dual for the resulting LP, which yields the constraints $a + b \ge 1$ and $-a - b \ge -1$ which is the same as $a + b \le 1$). The constraint for p is $-3a + 7b + c \ge 0$. So we get:

$$\begin{aligned} \min -2a+4b + c \quad & \text{s.t.} \\ & a + b = 1 \\ & -3a + 7b + c \ge 0 \\ & a, b, c \ge 0 \end{aligned}$$

Additional Remark: With some extra work (substituting $c = \max\{0, 3a - 7b\}$ and $b = 1 - a$) one can show this is equivalent to the LP we get from Approach 1, but this is not necessary for full credit.

Approach 3: This is the same as Approach 2, but we force all the variables to be non-negative before taking the dual. Letting $z = z^+ - z^-$ where $z^+, z^- \geq 0$, we can rewrite Alice's LP as:

$$\begin{aligned} \max z^+ - z^- \quad & \text{s.t.} \\ z^+ - z^- - 3p &\leq -2 \\ z^+ - z^- + 7p &\leq 4 \\ p &\leq 1 \\ p, z^+, z^- &\geq 0 \end{aligned}$$

Let a, b, c be the variables for the first, second, third constraint. Our objective is min $-2a + 4b + c$. The constraint for z^+ is $a + b \geq 1$, and the constraint for z^- is $-a - b \geq -1$. So just like in Approach 2, we get:

$$\begin{aligned} \min -2a + 4b + c \quad & \text{s.t.} \\ a + b &\geq 1 \\ -a - b &\geq -1 \\ -3a + 7b + c &\geq 0 \\ a, b, c &\geq 0 \end{aligned}$$

Which is identical to the LP we get from Approach 2.

Part B (20 pts): Greedy Algorithms – Solve one of the following two problems:

3 Greedy Option A: Fractional Knapsack

A thief has broken into a vault and has a knapsack in which to carry the loot. There are n items in the vault, where the i-th item weighs w_i and has value v_i. Assume that the knapsack can hold a total weight of at most W, and that for each i the thief can choose to carry any fraction $0 \leq a_i \leq 1$ of the i-th item, in which case it contributes weight $a_i w_i$ and value $a_i v_i$ to the knapsack. Help the thief design an efficient *greedy* algorithm to get away with the largest value in loot. Use an exchange argument to justify the correctness of your algorithm, and bound the running time.

Algorithm: Sort the items in decreasing order of v_i/w_i (this is the value per unit weight). Greedily pack your knapsack with items in decreasing order of value per unit weight, until you get to the first item that doesn't entirely fit — choose whatever fraction of that item that fits without exceeding total weight W.

Runtime: $O(n \log n)$. It takes this much time to sort the items, and then the rest of the algorithm can be run in linear time if we keep track of the total weight we've used so far.

Correctness: Consider an optimal solution OPT. Assume WLOG that the i-th item has the ith-highest value per unit weight v_i/w_i, and there are no ties (if there are ties we can treat all the items with same value per unit weight as one item with higher weight and same value per unit weight). If OPT is not identical to our solution, let i be the first item such that our algorithm uses a larger fraction of this item than OPT. Suppose our greedy solution uses fraction a_i, while OPT uses fraction $b_i < a_i$. Add $a_i - b_i$ amount of item i to OPT and reduce an equal weight of items numbered $i+1$ or greater from OPT. Since the removed items have smaller value per unit weight, the added value is larger than the removed value, while the total weight remains the same. This means we created a solution that is better than OPT. Contradiction.

3 Greedy Option B: Combining Ropes

You have n ropes with lengths $f_1, f_2 \ldots f_n$. You need to connect all these ropes into one rope. At each step, you may connect two ropes, at the cost of the sum of their lengths. Your total cost is the sum of the costs of all the steps resulting in a single rope.

Give a greedy algorithm that finds the lowest-cost strategy for combining ropes. Prove the optimality of your strategy using the optimality of Huffman coding (alternatively, you can prove optimality directly, but we recommend relying on Huffman's optimality). No runtime analysis needed.

An example: if we are given three ropes of lengths 1, 2, 4, then we can connect the three ropes as follows:

- First, connect ropes with length 1 and 2. This costs 3, and leaves you with two ropes of lengths 3, 4.
- Then, connect ropes with length 3 and 4. This costs 7, and leaves you with one rope of length 7.
- The overall cost of the above solution is 3+7 = 10.

The greedy strategy is to combine the two shortest ropes into one rope, and repeat.

This combining strategy is the same as for Huffman coding, so it can be represented by a binary tree, in the same way. Moreover, if rope i, or any rope that rope i is a part of, get combined a total of d_i times, then rope i contributes $f_i d_i$ to the final cost. So our task is to find a binary tree with the n ropes at the n leaves, such that the total cost $\sum_{i=1}^{n} f_i d_i$ is minimized, where d_i is the length of the path from leaf i to the root. This is exactly the the quantity that Huffman's encoding minimizes. Huffman's solution chooses to merge at each point the two nodes with smallest frequency – this correspond to connecting the two ropes with smallest length into one rope, and then repeating.

Part C (25 pts): Dynamic Programming – Solve one of the following two problems:

For whichever option you choose in this part, your solution should include the following:

- Identifying the subproblems (Option A only).
- A recurrence relation.
- Base cases for your recurrence relation.
- The order in which to solve the subproblems.
- The final output of the algorithm.
- A **justification** for your recurrence relation (A full proof by induction is not needed).
- Runtime analysis.

4 DP Option A: Counting Paths

You are given an unweighted **DAG** (Directed Acyclic Graph) G, along with a start node s and a target node t. Design a linear time (i.e., runtime $O(|V| + |E|)$) dynamic programming algorithm for computing the number of all paths (not necessarily shortest) from s to t.

For each $v \in V$ we define n_v to be the number of paths from s to v.

Our recurrence relation is $n_v = \sum_{u:(u,v)\in E} n_u$, since paths that end up in v must visit some node u with $(u, v) \in E$ prior to visiting v. Each path reaching v is counted exactly once since the last vertex before v in the path partitions all paths reaching v into disjoint sets.

The base case is $n_s = 1$, since there's a single path, i.e., the empty path, starting at s and ending up at s. This is due to the fact that the graph is acyclic.

We solve the subproblems in the topological order of the DAG. In this way, when we apply the recurrence to compute n_v the values of n_u for $u : (u, v) \in E$ are already computed.

The final output is n_t.

The runtime for solving all subproblems is $O(|V| + |E|)$. This is because we perform an addition at each vertex of the values on all incoming edges — i.e. constant amount of work for each vertex and each edge.

(Observe that this solution is similar to shortest paths in a DAG, where min is replaced with sum).

One can also define n_v to be the number of paths from v to t. In this case, the solution is functionally identical: The base case is $n_t = 1$, and the recurrence is $n_v = \sum_{u:(v,u)\in E} n_u$. The final answer is n_s.

4 DP Option B: All Pairs Shortest Paths

We have a directed weighted graph $G(V, E)$. The weight $w(e)$ of each edge can be any integer, including negative integers, but there are no negative-weight cycles. Our goal is to find the shortest path length between every pair of vertices. One option is to run Bellman-Ford from every vertex. However, this could take $O(|V|^4)$ time if the graph is a complete graph. In this problem we will design a **faster** algorithm.

Denote by $d(u, v, i)$ the length of the shortest u-v path using at most 2^i edges, for $i = 0, 1, \ldots, \lceil \log |V| \rceil$ (if no path using at most 2^i edges exists then $d(u, v, i) = \infty$). Design an efficient dynamic programming algorithm for computing all shortest path lengths between every pair of vertices, using the subproblems $d(u, v, i)$.

The recurrence relation is $d(u, v, i) = \min_{w\in V} [d(u, w, i - 1) + d(w, v, i - 1)]$. The shortest path from u to v using at most 2^i edges can be split into two paths of length at most 2^{i-1}: one from u to some w, and one from the same w to v, and these two paths must also be the shortest path between their endpoints (since otherwise we could replace them with shorter paths to get a shorter path from u to v).

Base cases: For $u \neq v$, the shortest u-v path using at most 1 edge is just the edge (u, v) if it exists. So $d(u, v, 0) = w_{u,v}$ if (u, v) exists, and ∞ otherwise. We also have the case $d(u, u, 0) = 0$.

We compute the subproblems in increasing order of i. That is, compute all $d(u, v, 0)$, then all $d(u, v, 1)$, then all $d(u, v, 2)$ etc. Since our recurrence for $d(u, v, i)$ only depends on subproblems $d(x, y, i-1)$, the order of pairs of vertices doesn't matter.

Our final output is for all u, v pairs to output $d(u, v, \lceil \log |V| \rceil)$ as the shortest path length (since $2^{\lceil \log |V| \rceil} \geq |V|$, and the number of edges on any shortest path is at most $|V|$).

There are $O(|V|^2 \log |V|)$ subproblems, and each takes $O(|V|)$ time to solve, so the runtime is $O(|V|^3 \log |V|)$.