CS 170 Homework 10 (OPTIONAL)

Due **11/6/2023, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

2 A Cohort of Secret Agents

A cohort of k secret agents residing in a certain country needs escape routes in case of an emergency. They will be travelling using the railway system which we can think of as a directed graph $G = (V, E)$ with V being the cities and E being the railways. Each secret agent i has a starting point $s_i \in V$, and all s_i's are distinct. Every secret agent needs to reach the consulate of a friendly nation; these consulates are in a known set of cities $T \subseteq V$. In order to move undetected, the secret agents agree that at most c of them should ever pass through any one city. Our goal is to find a set of paths, one for each of the secret agents (or detect that the requirements cannot be met).

Model this problem as a flow network. Specify the vertices, edges, and capacities; show that a maximum flow in your network can be transformed into an optimal solution for the original problem. You do not need to explain how to solve the max-flow instance itself.

Solution: We can think of each secret agent i as a unit of flow that we want to move from s_i to any vertex $\in T$. To do so, we can model the graph as a flow network by setting the capacity of each edge to ∞, adding a new vertex t and adding an edge (t', t) for each $t' \in T$. We can add a source s and edges of capacity 1 from s to s_i. By doing so, we restrict the maximum flow to be k.

Lastly, we need to ensure that no more than c secret agents are in a city. We want to add vertex capacities of c to each vertex. To implement it, we do the following: for each vertex v that we want add constraint to, we create 2 vertices v_{in} and v_{out}. v_{in} has all the incoming edges of v and v_{out} has all the outcoming edges. We also put a directed edge from v_{in} to v_{out} with edge capacity constraint c.

If the max flow is indeed k, then as every capacity is an integer, the Ford-Fulkerson algorithm for computing max flow will output an integral flow (one with all flows being integers). Therefore, we can incrementally starting at each secret agent i follow a path in the flow from s_i to any vertex in T. We iterate through all of secret agents to reach a solution.

3 Running a Gym

You are running a gym and need to hire trainers for it. Here are the constraints:

- There are D days in total.
- There are n instructors for hire. The i^{th} instructor is available for hire from day a_i to day b_i. The numbers $\{a_i, b_i\}$ for $i = 1 \dots n$ are given as input.
- Each instructor can be hired for their entire period of availability, i.e. if we hire the i^{th} instructor, then they would be at the gym every day from day a_i to day b_i.
- There should be **exactly** k instructors at the gym every day.

Your goal is to determine which instructors to hire so that the above constraints are met. Devise an efficient algorithm that solves the problem by constructing a directed graph G with a source s and sink t, and computing the maximum flow from s to t. Proof of Correctness is not required.

Solution:

Solution 1: Using $D + 2$ vertices.

Algorithm Description: We create one vertex for each day $d = 1 \dots D$. We also create a start node s and an end node t, which we will respectively index as 0 and $D + 1$ for simplicity of the arguments below.

Then, create an edge $u \to v$ for $u, v \in \{0, \dots, D\}$ if there is some instructor who is available from day $v + 1$ to day u, and set its weight to the number of instructors with this availability range. Note that we take vertex s to have index 0 for the purpose of these calculations. Additionally, add an edge from vertex D to the end node t with weight k.

Proof of correctness: To check whether there is a valid scheduling for a setup, check whether the max flow from s to t is equal to k. To prove this reduction is correct, we must show that this happens if and only if there is a valid scheduling.

Valid scheduling implies max flow of k: Given a valid scheduling, construct a max flow with value k as follows. For each instructor chosen in the scheduling, add a unit of flow along the edge corresponding to their availability range. Then add k units of flow from D to t. Note that the flow into a vertex is equal to the number of chosen instructors whose range ends on that day, and (except for in the case of vertex D) the flow out is equal to the number of instructors whose range ends the next day. Because these quantities must always be equal for a valid scheduling, conservation constraints are met for vertices $1, ..., D$. The flow out of vertex D is k by construction, and the flow into it must be D as well because all valid assignments have k chosen instructors with ranges ending on the last day, so its conservation constraint is also met. The flow also clearly has value k because k units of flow enter t.

Max flow of k implies valid scheduling: If there is a max flow with size k, then

there must be an integral max flow of the same value by integrality of the capacities. Choose such a flow, and construct a valid scheduling as follows. For each unit of flow along an edge u, v, select an instructor with availability range $u+1, v$. Note that number of instructors scheduled for day d is the number of selected instructors who start on/before d and end after d, which is equal to the value of the flow across the cut separating vertices with indices less than d from those with indices at least d. The value of all such s/t cuts must be k, so there are exactly k instructors scheduled for each day.

Solution 2: Using $n+2$ vertices.

Algorithm Description: We create one vertex for each instructor $i = 1 \dots n$, plus s and t. Then, connect vertices i and j with a unit capacity edge if $a_j = b_i + 1$. Connect all vertices with $a_i = 0$ to s and $b_i = D$ to t.

Proof of Correctness: Similar to the $D+2$ solution, paths from s to t correspond to sequences of instructors covering all d days.

Note: The $n+2$ solution is preferable to the other solution because D is only given as a parameter (whereas we are given a list of n instructors), so the $D+2$-vertex solution is actually pseudo-polynomial time.

4 Weighted Rock-Paper-Scissors

You and your friend used to play rock-paper-scissors, and have the loser pay the winner 1 dollar. However, you then learned in CS170 that the best strategy is to pick each move uniformly at random, which took all the fun out of the game.

Your friend, trying to make the game interesting again, suggests playing the following variant: If you win by beating rock with paper, you get 2 dollars from your opponent. If you win by beating scissors with rock, you get 1 dollars. If you win by beating paper with scissors, you get 4 dollar.

Feel free to use an online LP solver to solve your LPs in this problem. Here is an example of an online solver that you can use: `https://online-optimizer.appspot.com/`.

(a) Draw the payoff matrix for this game. Assume that you are the maximizer, and your friend is the minimizer.

(b) Write an LP to find the optimal strategy in your perspective.

Your friend now wants to make the game even more interesting and suggests that you assign points based on the following payoff matrix:

		Your friend:		
		rock	paper	scissors
You:	rock	-10	3	3
	paper	4	-1	-3
	scissors	6	-9	2

(c) Write an LP to find the optimal strategy for yourself. What is the optimal strategy and expected payoff?

(d) Now do the same for your friend. What is the optimal strategy and expected payoff? How does the expected payoff compare to the answer you get in part (c)?

Solution:

(a)

		Your Friend:		
		rock	paper	scissors
You:	rock	0	-2	1
	paper	2	0	-4
	scissors	-1	4	0

(b) Let r, p, s be the probabilities that you play rock, paper, scissors respectively. Let z stand for the expected payoff, if your opponent plays optimally as well.

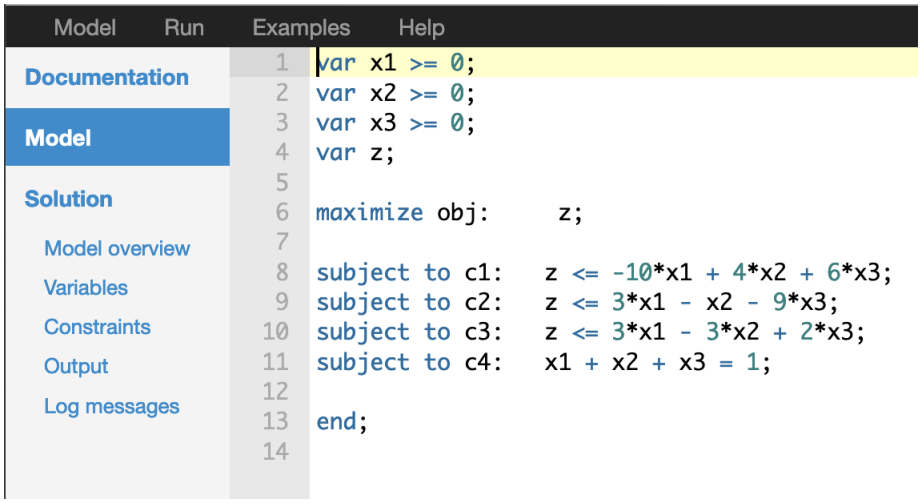
$$
\begin{aligned}
\max \quad & z & \\
2p - s &\geq z & \text{(Opponent chooses rock)} \\
4s - 2r &\geq z & \text{(Opponent chooses paper)} \\
r - 4p &\geq z & \text{(Opponent chooses scissors)} \\
r + p + s &= 1 & \\
r, p, s &\geq 0 &
\end{aligned}
$$

(c)

$$
\begin{aligned}
\max \quad & z & \\
-10r + 4p + 6s &\geq z & \text{(Opponent chooses rock)} \\
3r - p - 9s &\geq z & \text{(Opponent chooses paper)} \\
3r - 3p + 2s &\geq z & \text{(Opponent chooses scissors)} \\
r + p + s &= 1 & \\
r, p, s &\geq 0 &
\end{aligned}
$$

The optimal strategy is $r = 0.3346,\ p = 0.5630,\ s = 0.1024$ for an optimal payoff of -0.48.

If you are using the website suggested in this problem, here is what you should put in the model tab:

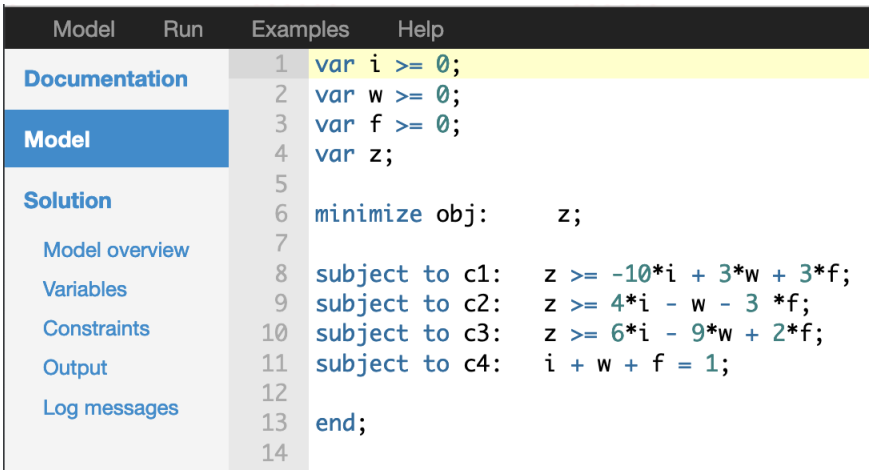


(d)

$$\begin{aligned} \min \quad & z & \\ -10r + 3p + 3s &\leq z & \text{(You choose rock)} \\ 4r - p - 3s &\leq z & \text{(You choose paper)} \\ 6r - 9p + 2s &\leq z & \text{(You choose scissors)} \\ r + p + s &= 1 & \\ r, p, s &\geq 0 & \end{aligned}$$

Your friend's optimal strategy is $r = 0.2677$, $p = 0.3228$, $s = 0.4094$. The value for this is -0.48, which is the payoff for you. The payoff for your friend is the negative of your payoff, i.e. 0.48, since the game is zero-sum.

If you are using the website suggested in this problem, here is what you should put in the model tab:



(Note for grading: Equivalent LPs are of course fine. It is fine for part (d) to maximize your friend's payoff instead of minimizing yours. For the strategies, fractions or decimals close to the solutions are fine, as long as the LP is correct.)

5 Domination

In this problem, we explore a concept called *dominated strategies*. Consider a zero-sum game with the following payoff matrix for the row player:

		Column:		
		A	B	C
Row:	D	1	2	-3
	E	3	2	-2
	F	-1	-2	2

(a) If the row player plays optimally, can you find the probability that they pick D without directly solving for the optimal strategy? Justify your answer.

Hint: How do the payoffs for the row player picking D compare to their payoffs for picking E?

(b) Given the answer to part a, if the both players play optimally, what is the probability that the column player picks A? Justify your answer.

(c) Given the answers to part a and b, what are both players' optimal strategies?

Note: All parts of this problem can be solved without using an LP solver or solving a system of linear equations.

Solution:

(a) 0. Regardless of what option the column player chooses, the row player always gets a higher payoff picking E than D, so any strategy that involves a non-zero probability of picking D can be improved by instead picking E.

(b) 0. We know that the row player is never going to pick D, i.e. will always pick either E or F. But in this case, picking B is always better for the column player than picking A (A is only better if the row player picks D). That is, conditioned on the row player playing optimally, B dominates A.

(c) Based on the previous two parts, we only have to consider the probabilities the row player picks E or F and the column player picks B or C. Looking at the 2-by-2 submatrix corresponding to these options, it follows that the optimal strategy for the row player is to pick E and F with probability $1/2$, and similarly the column player should pick B, C with probability $1/2$.

6 Decision vs. Search vs. Optimization

Recall that a vertex cover is a set of vertices in a graph such that every edge is adjacent to at least one vertex in this set.

The following are three formulations of the VERTEX COVER problem:

- As a *decision problem*: Given a graph G, return TRUE if it has a vertex cover of size at most b, and FALSE otherwise.
- As a *search problem*: Given a graph G, find a vertex cover of size at most b (that is, return the actual vertices), or report that none exists.
- As an *optimization problem*: Given a graph G, find a minimum vertex cover.

At first glance, it may seem that search should be harder than decision, and that optimization should be even harder. We will show that if any one can be solved in polynomial time, so can the others.

(a) Suppose you are handed a black box that solves VERTEX COVER (DECISION) in polynomial time. Give an algorithm that solves VERTEX COVER (SEARCH) in polynomial time.

(b) Similarly, suppose we know how to solve VERTEX COVER (SEARCH) in polynomial time. Give an algorithm that solves VERTEX COVER (OPTIMIZATION) in polynomial time.

Solution:

(a) If given a graph G and budget b, we first run the DECISION algorithm on instance (G, b). If it returns "FALSE", then report "no solution".

If it comes up "TRUE", then there is a solution and we find it as follows:

- Pick any node $v \in G$ and remove it, along with any incident edges.
- Run DECISION on the instance $(G \setminus \{v\}, b-1)$; if it says "TRUE", add v to the vertex cover. Otherwise, put v and its edges back into G.
- Repeat until G is empty.

Correctness: If there is no solution, obviously we report as such. If there is, then our algorithm tests individual nodes to see if they are in any vertex cover of size b (there may be multiple). If and only if it is, the subgraph $G \setminus \{v\}$ must have a vertex cover no larger than $b - 1$. Apply this argument inductively.

Running time: We may test each vertex once before finding a v that is part of the b-vertex cover and recursing. Thus we call the DECISION procedure $O(n^2)$ times. This can be tightened to $O(n)$ by not considering any vertex twice. Since a call to DECISION costs polynomial time, we have polynomial complexity overall.

Note: this reduction can be thought of as a greedy algorithm, in which we discover (or eliminate) one vertex at a time.

(b) Binary search on the size, b, of the vertex cover.

Correctness: This algorithm is correct for the same reason as binary search.

Running time: The minimum vertex cover is certainly of size at least 1 (for a nonempty graph) and at most $|V|$, so the SEARCH black box will be called $O(\log |V|)$ times, giving polynomial complexity overall.

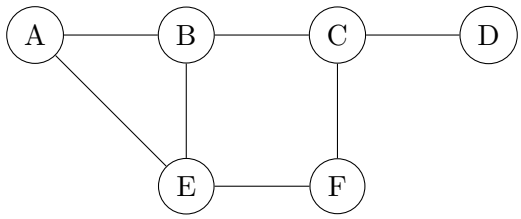
Finally, since solving the optimization problem allows us to answer the decision problem (think about why), we see that all three reduce to one another!

7 Vertex Cover to Set Cover

Vertex Cover: given an undirected unweighted graph $G = (V, E)$, a vertex cover C_V of G is a subset of vertices such that for every edge $e = (u, v) \in E$, at least one of u or v must be in the vertex cover C_V.

Set Cover: given a universe of elements U and a collection $\mathcal{S} = \{S_1, \ldots, S_m\}$, a set cover is any (sub)collection of sets C_S whose union equals U.

In the *minimum vertex cover problem*, we are given an undirected unweighted graph $G = (V, E)$, and are asked to find the smallest vertex cover. For example, in the following graph, $\{A, E, C, D\}$ is a vertex cover, but not a minimum vertex cover. The minimum vertex covers are $\{B, E, C\}$ and $\{A, E, C\}$.



Then, recall in the *minimum set cover problem*, we are given a set U and a collection $S_1, \ldots, S_m$ of subsets of U, and are asked to find the smallest set cover. For example, given $U := \{a, b, c, d\}$, $S_1 := \{a, b, c\}$, $S_2 := \{b, c\}$, and $S_3 := \{c, d\}$, a solution to the problem is $C_S = \{S_1, S_3\}$.

Give an efficient reduction from the minimum vertex cover problem to the minimum set cover problem. Proof of correctness is not required.

Solution:

Algorithm Description: Let $G = (V, E)$ be an instance of the minimum vertex cover (MVC) problem. Create an instance of the minimum set cover (MSC) problem where $U = E$ and for each $u \in V$, the set S_u contains all edges incident to u.

Let $C_S = \{S_{u_1}, S_{u_2}, \ldots, S_{u_k}\}$ be a set cover, where $k = |C_S|$. Then our corresponding vertex cover will be $C_V = u_1, u_2, \ldots, u_k$.

Proof of Correctness: To see that V_C is a vertex cover, take any $(u, v) \in E$. Since $(u, v) \in U$, there is some set S_{u_i} containing (u, v), so u_i equals u or v and (u, v) is covered in the vertex cover. Thus, every vertex cover in G corresponds to a set cover in $U, \mathcal{S}$.

Now take any vertex cover $u_1, \ldots, u_k$. To see that $S_{u_1}, \ldots, S_{u_k}$ is a set cover, take any $(u, v) \in E$. By the definition of vertex cover, there is an i such that either $u = u_i$ or $v = u_i$. So $(u, v) \in S_{u_i}$, so $S_{u_1}, \ldots, S_{u_k}$ is a set cover. Thus, every set cover in $U, \mathcal{S}$ corresponds to a vertex cover in G.

Since every vertex cover has a corresponding set cover (and vice-versa) and minimizing set cover minimizes the corresponding vertex cover, the reduction holds.

8 Min Cost Flow

In the max flow problem, we just wanted to see how much flow we could send between a source and a sink. But in general, we would like to model the fact that shipping flow takes money. More precisely, we are given a directed graph G with source s, sink t, costs l_e, capacities c_e, and a flow value F. We want to find a nonnegative flow f with minimum cost, that is $\sum_e l_e f_e$, that respects the capacities and ships F units of flow from s to t.

(a) Show that the minimum cost flow problem can be solved in polynomial time.

(b) Describe a reduction from the shortest path problem to the minimum cost flow problem. Proof of correctness is not required.

(c) Describe a reduction from the maximum flow problem to the minimum cost flow problem. Proof of correctness is not required.

Solution:

(a) Write min-cost flow as a linear program. One formulation is as follows:

$$\begin{aligned} &\min \sum_e l_e f_e \\ &\text{subject to } 0 \le f_e \le c_e && \forall e \in E \\ &\sum_{e \text{ incoming to } v} f_e = \sum_{e \text{ outgoing from } v} f_e && \forall v \in V, v \neq s, t \\ F + &\sum_{e \text{ incoming to } s} f_e = \sum_{e \text{ outgoing from } s} f_e \end{aligned}$$

(b) **Algorithm Description:** Consider a shortest path instance on a graph G with start s, end t, and edge weights l_e. Let $F = 1$ and let $c_e = 1$ for all edges e. In the min cost flow f—the subgraph H of G consisting of edges with nonzero flow—all directed paths from s to t have length equal to the shortest path from s to t (proof shown below). Therefore, to find a shortest path from s to t, it is enough to use a DFS or BFS to find any path between s to t in H.

Proof of Correctness (not required): Suppose that each $l_e > 0$ (we can contract any edge with $l_e = 0$). First, we show that H is a DAG. If H is not a DAG, it must contain a cycle. Let $\delta > 0$ be the minimum flow along any edge of this cycle. Subtracting δ from the flow of all edges in this cycle results in a flow that still ships 1 unit of flow from s to t. This flow, though, has smaller cost than before, a contradiction to the fact that f is a min-cost flow. Therefore, H is a DAG.

Consider any path P between s and t in H. Let δ be the minimum flow on any edge of this path. One can write the flow f as a linear combination of paths including P by writing

$$f = \delta p + \delta_1 q_1 + \ldots + \delta_k q_k$$

where p and q_i are the unit flows along the paths P and Q_i respectively, δ_i is the minimum amount of flow on some arbitrary path Q_i after subtracting flow from $P, Q_1, Q_2, \ldots, Q_{i-1}$. Since subtracting δ_i units of flow decreased the flow from s to t by δ_i units, $\delta + \sum_i \delta_i = 1$. Since the objective function for min-cost flow is linear, we can write

$$\sum_e l_e f_e = \delta \sum_e l_e p_e + \sum_i \delta_i \sum_e l_e q_{ie} = \delta \sum_{e \in P} l_e + \sum_i \delta_i \sum_{e \in P_i} l_e$$

In particular, the cost of the flow f is the average length of the paths $P, Q_1, \ldots, Q_k$ according to the weights $\delta, \delta_1, \ldots, \delta_k$. Therefore, if P or some Q_i is not a shortest path from s to t, one can decrease the cost of f by reassigning the flow along that path to a shortest path between s and t. This means that if f has minimum cost, P and all Q_is must be shortest paths between s and t. Since P was chosen to be an arbitrary path from s to t in H, all paths in H must be shortest paths between s and t.

(c) We can use binary search to find the true max flow value. Consider a max flow instance on a graph G with capacities c_e where we wish to ship flow from s to t. Create a min-cost flow instance as follows. Set the capacities to be equal to c_e and let the lengths be arbitrary (say $l_e = 1$ for all edges). We know that the max flow value $F_{max} \leq \sum_e c_e$. If the capacities are integers, F_{max} is also an integer. Therefore, we can binary search to find its true value. For an arbitrary F, we can find out if there is a flow that ships more than F units of flow by querying our min-cost flow instance with value at least F. If there is a flow with value at least F, it will return a flow with finite cost. Otherwise, the program is infeasible.