Homework Assignment 3

Submission Guidelines.

- Latex submissions are preferred but not mandatory. However, please write your solutions on unruled paper and upload a clearly scanned copy. Please do not use poorly lit photographs. Keep your solutions brief and to the point. Meaningless rambles fetch negative credits.
- Collaboration is allowed in teams of up to two students, but each student must write and submit their own individual solutions in their own words. Also mention the name of your collaborator.
- "Bibles" such as textbooks, lecture notes from other courses, or online forums (e.g., Stack Exchange) may not be used as references for your write-up.
- Limited use of large language models (LLMs) is permitted, but you may not directly copy-paste text from them; use them only for clarification and then produce your own reasoning and explanations.

Problem 1. (10 points) This problem concerns running time optimizations to the Hungarian algorithm for computing minimum-cost perfect bipartite matchings. Recall the $O(mn^2)$ running time analysis from lecture: there are at most n augmentation steps, at most n price update steps between two augmentation steps, and each iteration can be implemented in $O(m)$ time.

(a) By a phase, we mean a maximal sequence of price update iterations (between two augmentation iterations). The naive implementation in lecture regrows the search tree from scratch after each price update in a phase, spending $O(m)$ time on this for each of up to n iterations. Show how to reuse work from previous iterations so that the total amount of work done searching for good paths, in total over all iterations in the phase, is only $O(m)$.

Solution. Goal. After a price update, do *not* restart BFS. Continue the existing alternating BFS/tree and only extend it using the newly tight edges.

State kept between price updates.

- A single FIFO queue Q holding the current BFS frontier.
- A parent pointer for each discovered vertex which will be used to retrieve the path

What is done.

- Run BFS normally over *currently tight* edges by popping Q and expanding each vertex once (scan all its incident edges when it is first visited).

- When Q becomes empty, compute the price increment δ and apply it. (This is taken care of in part (b))
- After applying δ, the only edges that matter are those whose reduced cost has just reached 0 (i.e., edges that became tight). For each such edge, if an endpoint is unvisited, mark it visited, set its parent, and push it into Q.
- Resume BFS from Q. Repeat until an augmenting path is found.

Why this is correct.

- Price updates only decrease reduced costs; previously discovered reachability cannot be lost. New tight edges can only enlarge the reachable set of vertices.
- Therefore reusing the visited set and enqueuing endpoints of newly tight edges yields exactly the same reachable set that a full restart of BFS would produce.

Why the work per phase is $O(m)$.

- Each vertex is visited and expanded at most once per phase, so its incident edges are scanned at most once.
- Each edge is considered at most once as "became tight" (when its reduced cost hits 0), and is otherwise only encountered when scanning its visited endpoint. Thus each edge causes $O(1)$ work in the phase.
- Summing over all edges gives $O(m)$ total work per phase.

Rubric: $+2$ for maintaining the correct pointers (especially the parent or something analogous) , $+3$ for the correct algorithm and some explanation.

(b) The other non-trivial work in a price update phase is computing the value of δ (the minimum magnitude by which the prices can be updated without violating any of the invariants). This is easy to do in $O(m)$ time per iteration. Explain how to maintain a heap data structure so that the total time spent computing δ over all iterations in the phase is only $O(m \log n)$. Be sure to explain what heap operations you perform while growing the search tree and when executing a price update.

Solution.

(b) Computing all the δ's in $O(m \log n)$ using a heap. We maintain a min-heap whose purpose is to quickly identify the edge that determines

$$\delta \;=\; \min_{(v,w):\, v\in S,\, w\notin N(S)} \big(c_{vw} - p(v) - p(w)\big).$$

(1) What the heap stores. At any moment in a phase, the heap H contains exactly the edges (v, w) such that

$$v \in S, \qquad w \notin N(S),$$

and for each such edge we store its *initial* reduced slack

$$\text{slack}_0(v, w) \;=\; c_{vw} - p(v) - p(w),$$

together with a global offset Γ such that the *current* slack is interpreted as

$$\text{slack}(v, w) \;=\; \text{slack}_0(v, w) - \Gamma.$$

A single offset suffices because every price update decreases the slack of *all* such edges by the same amount.

(2) When edges are inserted into or removed from the heap. The heap is updated incrementally as the search tree grows:

- When a vertex v enters S (an even-level vertex is discovered), then for every incident edge (v, w):
  - If $w \notin N(S)$, insert (v, w) into H with key $\text{slack}_0(v, w)$.
  - Otherwise ignore the edge.
- When a vertex w enters $N(S)$ (an odd-level vertex is discovered), then for every incident edge (u, w):
  - If $u \in S$, remove (u, w) from H, since it is no longer of the form $v \in S$, $w \notin N(S)$.
  - Otherwise ignore the edge.

Each edge is inserted at most once (when its S-endpoint appears) and removed at most once (when its other endpoint joins $N(S)$). Thus there are $O(m)$ heap operations per phase, each costing $O(\log n)$.

(3) Computing δ and updating the offset. To obtain δ, we simply take the minimum key of the heap:

$$\delta \;=\; \min_{(v,w)\in H} \text{slack}(v, w) \;=\; \Big(\min_{(v,w)\in H} \text{slack}_0(v, w)\Big) - \Gamma.$$

Extracting the minimum key of H takes $O(1)$ time.

After computing δ, the algorithm performs the price update:

$$p(v) \leftarrow p(v) + \delta \quad (v \in S), \qquad p(w) \leftarrow p(w) - \delta \quad (w \in N(S)).$$

For every candidate edge (v, w) with $v \in S$, $w \notin N(S)$, this reduces the true slack by exactly δ. Instead of updating each heap key individually, we simply update the global offset:

$$\Gamma \;\leftarrow\; \Gamma + \delta,$$

so that $\text{slack}(v, w) = \text{slack}_0(v, w) - \Gamma$ remains valid for all edges. Heap ordering is unaffected, since all slacks shift uniformly.

Final running time. There are $O(m)$ heap insertions/deletions per phase, each in $O(\log n)$ time. Minimum-slack queries take $O(1)$ time. Combined with the $O(m)$ search-tree work from part (a), the total running time per phase is

$$O(m \log n).$$

Rubric.: I do not expect you to write the solution exactly as above or in this detail. However, a few things are very important - (a) The correct data-structure (+1 for this) (b) Explaining how an edge enters the heap and with what key (+1) (c) Using some idea to avoid updating the keys (+3, this is very crucial to ensure the runtime of $m \log n$ per phase)

[This yields an $O(mn \log n)$ time implementation of the Hungarian algorithm.]

Problem 2. (10 points) Consider the following linear programming problem:

$$\begin{aligned} \text{minimize} \quad & cx \\ \text{subject to} \quad & x_1 + x_2 \geq 1, \\ & x_1 + 2x_2 \leq 3, \\ & x_1 \geq 0, \\ & x_2 \geq 0, \\ & x_3 \geq 0. \end{aligned}$$

For each of the following objectives c, give the optimum value and the set of optimum solutions:

(a) $c = (1, 0, 0)$

(b) $c = (0, 1, 0)$

Justify your answers with a sentence or two. This means convince me mathematically that your answer is correct.

Solution.

Problem 2. Consider the linear program

$$\begin{aligned} \text{minimize} \quad & c_1x_1 + c_2x_2 + c_3x_3 \\ \text{subject to} \quad & x_1 + x_2 \geq 1, \\ & x_1 + 2x_2 \leq 3, \\ & x_1, x_2, x_3 \geq 0. \end{aligned}$$

Since x_3 does not appear in any constraint, at optimum we always set $x_3 = 0$.

(a) Objective $c = (1, 0, 0)$: minimize x_1.

We want to minimize x_1. From $x_1 + x_2 \geq 1$ we have $x_1 \geq 1 - x_2$, so to make x_1 small we choose x_2 as large as possible. Setting $x_1 = 0$ gives $x_2 \geq 1$, and the second constraint

$$0 + 2x_2 \leq 3$$

implies $x_2 \leq 1.5$. Thus $(x_1, x_2) = (0, x_2)$ is feasible for all $x_2 \in [1, 1.5]$.

Hence the optimal value is 0.

(b) Objective $c = (0, 1, 0)$: minimize x_2.

We now minimize x_2. Try $x_2 = 0$. Then $x_1 + x_2 \geq 1$ implies $x_1 \geq 1$, and $x_1 + 2x_2 \leq 3$ implies $x_1 \leq 3$. Thus $x_2 = 0$ is feasible with any $x_1 \in [1, 3]$.

Therefore the optimal value is 0.

Disclaimer : The above solution is completely GPT generated. **Rubric** : $+2.5$ for correct solution, $+2.5$ for a correct explanation for each.

Problem 3. (10 points) You work for the Short-Term Capital Management company and start the day with D dollars. Your goal is to convert them to yen through a series of currency trades involving assorted currencies, so as to maximize the amount of yen you end up with.

You are given a list of pending orders: client i is willing to convert up to u_i units of currency a_i into currency b_i at a rate of r_i (that is, he will give you r_i units of currency b_i for each unit of currency a_i).

You may also borrow an arbitrary amount of any currency, with zero interest, provided you pay it back in the same currency by the end of the day. Assume that around any directed cycle of trades, $\prod r_i < 1$; that is, there is no opportunity to make a profit by arbitrage.

Formulate a linear program for maximizing the amount of yen you have at the end of trading.

Solution.

Let x_i be the variable that denotes the amount of money we wish to trade with client i. We want to maximize the amount of yen we make, so we want to maximize the sum of the x_i's that trade from anything to yen, which we denote by y, minus the amount of yen we trade away. Making sure we optimize for the final amount of yen is important because otherwise we would simply optimize for a maximum amount of yen we could achieve over time, which could mean trading away all the yen we got into pesos and trading it back into yen again, increasing the LP's optimal value without actually ending with more yen.

We denote by d the dollar currency. When we write $i = (a_i, b_i)$, we mean the client i that trades currency a_i for b_i. We can write the objective as

$$\max \left(\sum_{i:(a_i,y)} r_i x_i \; - \sum_{i:(b_i,y)} x_i \right).$$

We subject the variables x_i to a few constraints. First, we cannot trade more than u_i units with client i:

$$0 \;\le\; x_i \;\le\; u_i.$$

Lastly, for every currency c, we cannot trade more of that currency than we have:

$$\sum_{i:(a_i,c)} x_i \;\le\; \sum_{i:(b_i,c)} r_i x_i.$$

Also, we cannot trade more than D dollars, as that is our initial amount of money, plus whatever we trade *into* dollars:

$$\sum_{i:(a_i=d)} x_i \;\le\; D + \sum_{i:(b_i=d)} r_i x_i.$$

Rubric : $+2$ for each of the constraints and $+2$ for the objective. Each of the constraints are important and hence they carry equal weigtage.

Problem 4. (10 points) Consider the following linear program with respect to a graph $G = (V, E)$

with non-negative edge costs c_e.

$$\begin{aligned} \min \quad & \sum_{e\in E} c_e x_e \\ \text{subject to} \quad & \sum_{e\in\delta^+(S)} x_e \geq 1, \quad \forall\, S\subseteq V \text{ with } s\in S,\, t\notin S, \\ & x_e \geq 0, \qquad\qquad \forall\, e\in E. \end{aligned}$$

(a) Which graph problem you know about is captured by the linear program ? Prove formally your claim. In other words, show that all feasible solutions to the problem of your choice corresponds to an integer solution of the above LP.

Solution. This LP is the standard cut-based relaxation of the *shortest s–t path problem* (or equivalently, the minimum s–t cut in an uncapacitated directed graph). We show that the integer feasible solutions of the LP correspond exactly to directed s–t paths.

($\Rightarrow$) Any s–t path is an integer feasible solution. Let P be a directed path from s to t. Define

$$x_e = \begin{cases} 1, & e\in P, \\ 0, & e\notin P. \end{cases}$$

For any S with $s\in S$ and $t\notin S$, the path P must leave S at least once; hence $\delta^+(S)\cap P\neq\emptyset$, and therefore $\sum_{e\in\delta^+(S)} x_e \geq 1$. Thus x is feasible.

($\Leftarrow$) Any integer feasible x corresponds to an s–t path. Let x be feasible and integer. Let

$$R = \{v\in V : \text{there exists a directed } s\text{–}v \text{ path using only edges with } x_e = 1\}.$$

If $t\notin R$, take $S = R$. Then $s\in S$ and $t\notin S$, but by definition no edge in $\delta^+(S)$ has $x_e = 1$. Hence

$$\sum_{e\in\delta^+(S)} x_e = 0,$$

violating the constraint for S. Thus $t\in R$, so there is a directed s–t path using only edges with $x_e = 1$, proving the claim.

Therefore, the integer feasible solutions of the LP are exactly the incidence vectors of directed s–t paths, and the LP captures the shortest s–t path problem.

Rub : +1 for correct identification of the problem and +2 for each direction.

(b) Note that the LP has exponentially many constraints. In spite of that, give a *polynomial* time algorithm to solve the following - given any candidate solution x, either return that x is feasible or return a constraint which is violated by x.

Solution. Although the LP has exponentially many constraints, we can test feasibility of any candidate solution x in polynomial time.

A constraint corresponding to a set S is violated exactly when

$$\sum_{e\in\delta^+(S)} x_e < 1.$$

Thus we seek an s–t cut of minimum x-capacity. Construct a directed graph with edge capacities x_e, and compute a minimum s–t cut $(S, \bar{S})$ using any max-flow algorithm. Let

$$\lambda = \sum_{e \in \delta^+(S)} x_e.$$

- If $\lambda < 1$, then the cut S violates its constraint, and we output this set.
- If $\lambda \geq 1$, then every s–t cut satisfies its constraint, so x is feasible.

Since minimum s–t cut can be computed in polynomial time, this gives a polynomial-time separation oracle for the LP.

Thus, despite exponentially many constraints, feasibility of a candidate x can be checked efficiently.

Rubric : This is subjective , $+5$ for any correct solution.