CS 255 Practice Midterm

Directions:

- The exam is **in class on Thursday, March 28**.
- You have **75 minutes**.
- You are allowed to have one hand-written two-sided 8.5×11 cheat sheet.
- No phones, laptops, tablets, textbooks, lecture notes or other resources are allowed.
- I recommend you read every question first. If something is not clear, please ask for clarification as soon as possible.
- Points are listed at the beginning of each question as **(x Points)**. Subproblems might not be worth an equal amount of points.

Determine if the following statements are True or False. Provide a brief justification or example. **(2 Points Each)**: 1 point for correct answer (True/False), 1 point for justification or example.

1. Given an arbitrary flow network G, the value of *any* flow is upper bounded by the capacity of *any* cut.

True. This is weak duality discussed in lecture.

2. In a parallel algorithm, a race condition occurs when two threads need to *read* from the same memory location.

False. One thread needs to write to a location another thread reads.

3. Suppose that we have a flow network G where *all* of the capacities are integer multiples of 6. Then, the value of the max-flow in G is also an integer multiple of 6.

True. If all capacities are integer, then there exists an integer valued max-flow. In Ford-Fulkerson, the initial augmenting path will send a multiple of 6 from s to t. Then, the residual capacities will be multiples of 6, so that the next augmenting path sends a multiple of 6, and so on.

4. Chebyshev's inequality requires stronger assumption of independence than a Chernoff bound.

False. Applying Chebyshev's to a sum of independent random variables requires pairwise independence. Chernoff bounds assume full independence.

5. Consider the following multithreaded pseudocode for transposing an $n \times n$ matrix A in place:

```
P-Transpose(A):
    n ← A.rows
    parallel for j ← 2 to n
        parallel for i ← 1 to j − 1
            exchange a_ij with a_ji
```

Analyze the work, span, and parallelism of this algorithm.

Solution. $T_1(n) = \Theta(n^2)$, $T_\infty(n) = \Theta(\log n)$, and parallelism is $\Theta(n^2/\log n)$.

6. **(10 Points)** Each semester, dozens of SJSU graduate students work as teaching assistants (TA) for CS courses at the university. There are two restrictions: i) students can only be a TA for a course in which they received an A, and ii) at most one student can be the TA for each course. There are n students and m courses.

(a) Describe and analyze an efficient algorithm that either assigns each student a distinct course they can TA, or reports that such an assignment is impossible. Your input is a two-dimensional boolean array $CanTA[1 \ldots n][1 \ldots m]$, where $CanTA[i][j] =$ True if and only if student i can TA for course j. The runtime of your algorithm should be stated in terms of the original input parameters n and m.

(b). Oh, no! A mischievous professor hacked into the CS Department's system and assigned each student to course they can't TA! Since you are a clever algorithms student, you propose a strategy to fix everyone's assignments: each student will give their assigned course to another student that can actually TA it. For example, suppose

- Alice was assigned course A, but she can only TA courses C and D.
- Bob was assigned course B, but he can only TA courses A and C.
- Claire was assigned course C, but she can only TA courses A and D.
- Dan was assigned course D, but he can only TA courses B and C.

The students can fix their assignments as follows: Alice gives course A to Bob, Bob gives course B to Dan, Claire gives course C to Alice, and Dan gives course D to Claire.

Describe and analyze an efficient algorithm to compute an exchange that results in a valid assignment of students to courses. The input to your algorithm is the boolean array $CanTA$ from part (a) and a second array $TA[1 \ldots n]$, where $TA[i]$ is the index of the course originally assigned to student i. The correct output is an array $GiveTo[1 \ldots n]$, where $GiveTo[i] = j$ means student i should give their course to student j.

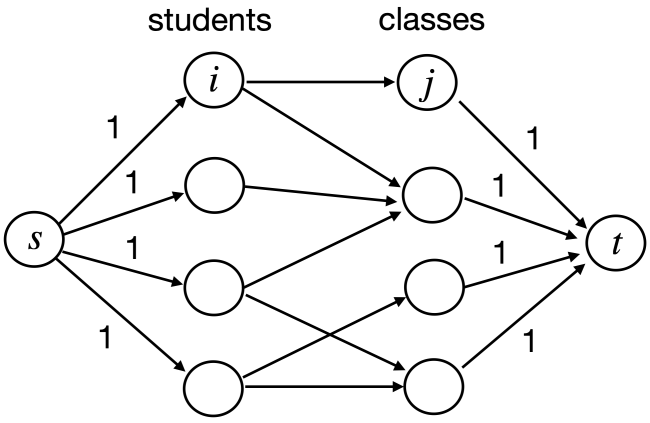
You can assume that $CanTA[i][TA[i]] =$ False for every index i (that is, no student can TA their assigned course) and that a valid exchange exists. Also assume that $m = n$.

Solution. (a) We want to match students with courses. You could just say we compute a maximum bipartite matching with students on the left and courses on the right, then give runtime. That would be full credit.

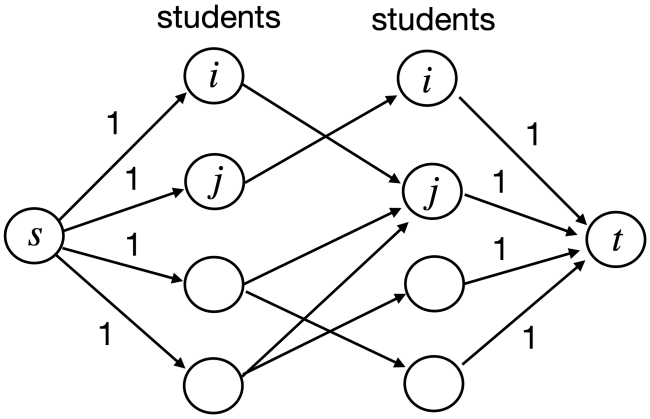
If you don't remember the matching flow network, we construct a bipartite graph as follows. We create a vertex on the left for each student i, and vertex on the right for each class j. We make an edge with capacity 1 connecting student i with class j if $CanTA[i][j] =$ True. We also add a source s and add edges (s, i) with capacity 1 to each student, and a sink t and edges (j, t) with capacity 1 from every class j.

We compute a max-flow in the network. If the max-flow saturates all the edges out of s, then each student is assigned to a course; otherwise, no assignment is possible. We can obtain an assignment by examining edges with positive flow. That is, if there is flow positive flow on the edge (i, j) then we assign student i to TA class j.

There are $|E| = O(mn)$ edges in the network. We can construct the network in $O(E)$ time. The value of the max-flow, f^*, is upper bounded by n, the number of students, where the flow saturates all edges out of s. We can compute a max-flow in $O(Ef^*) = O(n^2m)$ time using Ford-Fulkerson.



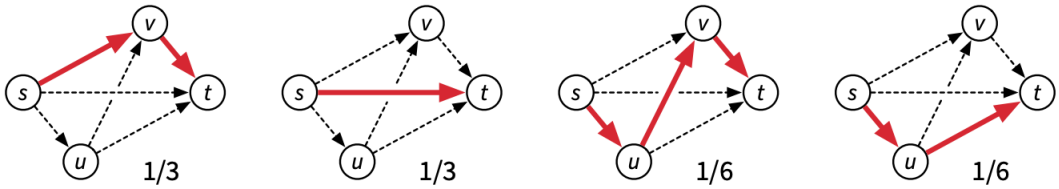
Solution. (b) We compute a another matching using a different bipartite graph. In the new graph, both left and right side vertices are students. We add an edge (i, j) if student i can give their assignment to student j. That is, if $CanTA[j][TA[i]] = \text{True}$. We complete that network by adding a source s and sink t as in part (a).



We compute a max-flow in the network. If edge (i, j) has positive flow, we set $GiveTo[i] = j$. There are $O(n)$ vertices, and $O(n^2)$ edges in the network. We can construct the network in $O(n^2)$ time. The value of the max-flow is no more than n. Therefore, we can compute the max-flow in $O(Ef^*) = O(n^3)$ time using Ford-Fulkerson.

7. Let $G = (V, E)$ be an arbitrary DAG with a unique source s and unique sink t. Suppose we compute a random walk from s to t, where at each vertex v we choose an outgoing edge $v \to w$ uniformly at random to determine the successor of v.

For example, in the following graph, there are four walks from s to t which are chosen with the indicated probabilities.



(a) Describe and analyze an algorithm to compute, for every vertex v, the probability that the random walk visits v. For example, in the graph above, a random walk visits the source s with probability 1, the bottom vertex u with probability $1/3$, the top vertex v with probability $1/2$, and the sink t with probability 1.

Solution. Since G is a DAG, any vertex v can only be reached from vertices u that are before v in topological order. Specifically

$$\begin{aligned} P(\text{visit } v) &= \sum_{(u \to v) \in E} P(\text{visit } v \text{ from } u) \\ &= \sum_{(u \to v) \in E} P(u \to v | \text{ visit } u)\ P(\text{visit } u) \\ &= \sum_{(u \to v) \in E} \frac{1}{\text{degree}(u)}\ P(\text{visit } u). \end{aligned}$$

We can calculate the desired probabilities using dynamic programming. We create a field $v.P$ for each vertex. We process the vertices in topological order, updating $v.P$ using the equation above. The base cases are $v \in \{s, t\}$, where $s.P = t.P = 1$. This algorithm runs in $O(V + E)$ time.

(b) Describe and analyze an algorithm to compute the expected number of edges in the random walk. For example, given the graph shown above, your algorithm should return the number $2 \cdot 1/3 + 1 \cdot 1/3 + 3 \cdot 1/6 + 2 \cdot 1/6 = 11/6$.

Solution. For each edge e, define the indicator random variable $X_e = 1$ if the walk traverses e, 0 otherwise. Then, $X = \sum_{e \in E} X_e$, gives the number of edges in the walk. By linearity of expectation

$$E[X] = \sum_{e \in E} E[X_e] = \sum_{e \in E} P(X_e = 1).$$

Observe that for edge $e = u \to v$, we have

$$P(X_e = 1) = P(\text{traverse } u \to v) = P(u \to v | \text{ visit } u) P(\text{visit } u) = \frac{1}{\text{degree}(u)} P(\text{visit } u),$$

which is computed by the algorithm from part (a). Therefore, given the result of part (a), we can compute $E[X]$ in $O(V + E)$ time. Overall the algorithm runs in $O(V + E)$ time.