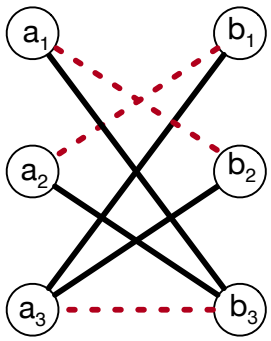
Problem 1. Matching. [25 points]

Consider the following problem `Bipartite-Perfect-Matching`:

Input: Undirected *bipartite* graph $G = (V, E)$ with bipartition $V = L \cup R$ where $|L| = |R| = n$.

Output: A perfect matching if one exists, and NO if none exist in G.

A perfect matching is a subset S of edges where every vertex $v \in V$ is incident exactly one edge in S. Here is an example of a bipartite graph with a perfect matching in dashed edges:



Explain how to reduce the `Bipartite-Perfect-Matching` problem to the max-flow problem.

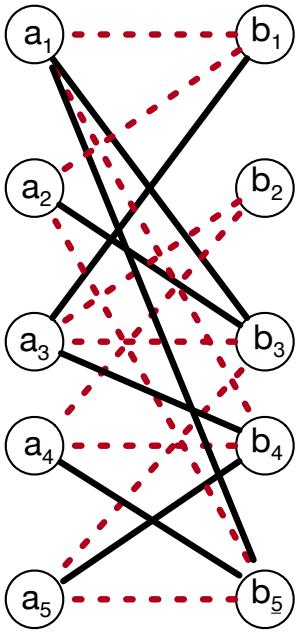
In other words, explain how to solve the bipartite perfect matching problem using the algorithm for solving max-flow as a black-box.

Part (a): Given an input $G = (V, E)$ to the `Bipartite-Perfect-Matching` problem, explain how you create the input to the max-flow problem (specify the graph G' and the edge capacities in this new graph). Do not do it for the above example, do it in general.

Part (b): Given a max-flow f^* for the flow network that you defined in part (a), explain how you use f^* to determine if G has a perfect matching.

Part (c): What is the running time of your algorithm in terms of the original graph G where $n = |V|$ and $m = |E|$. State whether you are using the Ford-Fulkerson or Edmonds-Karp algorithm (only consider these two which we saw in class); faster is better.

Part (d): A 2-perfect subgraph is a subset S of edges where every vertex has degree exactly 2 in S. So a perfect matching corresponds to a 1-perfect subgraph. Below is an example graph with a 2-perfect subgraph marked by red/dashed lines. Explain what needs to change in part (a) to check if a bipartite graph G has a 2-perfect subgraph.


a1
b1
a2
b2
a3
b3
a4
b4
a5
b5

Problem 2. MST. [20 points]

Standard disclaimer: use the algorithms from class, such as DFS, Explore, BFS, Dijkstra's (using min-heaps), connected components, etc., as a black-box subroutine for your algorithm. If you attempt to modify one of these algorithms you will not receive full credit, even if it is correct. Make sure to explain your algorithm in words, no pseudocode. Faster – and correct – is worth more credit.

You are given a tree T which is a MST of a graph $G = (V, E)$. For one particular edge e^* which is not in T, its edge weight is decreased (all other edges stay the same).
Specifically the weight of e^* changed from a to b where $a > b > 0$.

Find a MST T' for this new graph using the MST T for the old graph.

The set T is given as a list of edges or in adjacency list representation (whichever is convenient for you). You can access the function $w()$ for the weight of each edge. You are given one particular edge $e^* = (y, z)$ where $e^* \notin T$ and you are given the old weight a and new weight b for this edge e^*.

Part (a): Explain your algorithm for finding a MST T' for these new weights.
Part (b): State and explain the running time in terms of $n = |V|$ and $m = |E|$. Faster and correct is worth more credit. Your algorithm needs to be faster than constructing a MST from scratch to receive credit.

Problem 3. Verifying Max-Flow. [20 points]

Standard disclaimer: use the algorithms from class, such as DFS, Explore, BFS, Dijkstra's (using min-heaps), connected components, etc., as a black-box subroutine for your algorithm. If you attempt to modify one of these algorithms you will not receive full credit, even if it is correct. Make sure to explain your algorithm in words, no pseudocode. Faster – and correct – is worth more credit.

For a flow network $G = (V, E)$ with specified $s, t \in V$ and capacities $c_e > 0$ for $e \in E$, you are given a flow f (you are given f_e for every $e \in E$).
Note: For both parts, **faster** – and correct – is worth more credit. And if you're algorithm takes the same time as computing a max-flow from scratch then you'll get 0 points.
Part (a): Explain an algorithm to *verify/check* whether or not f is a **valid** flow. Be sure to state and explain the running time of your algorithm.
Part (b): It is claimed that f is a maximum flow. Explain an algorithm to *verify/check* whether or not f is a **maximum** flow. Be sure to state and explain the running time of your algorithm.

Problem 4. Distances. [20 points]

Standard disclaimer: use the algorithms from class, such as DFS, Explore, BFS, Dijkstra's (using min-heaps), connected components, etc., as a black-box subroutine for your algorithm. If you attempt to modify one of these algorithms you will not receive full credit, even if it is correct. Make sure to explain your algorithm in words, no pseudocode. Faster – and correct – is worth more credit.

You are given a **directed** graph $G = (V, E)$ with positive weights $w(e) > 0$ on edges $e \in E$. For each pair of vertices x, y let

$$\text{dist}(x, y) = \text{shortest path distance } \textbf{from } x \textbf{ to } y.$$

You are given as input $G = (V, E)$ and you are also given a specific vertex z^*. Give an algorithm to find all vertices v where:

$$\text{dist}(v, z^*) > 2\text{dist}(z^*, v).$$

In words, you want to output the list of vertices v whose distance *from* v *to* z^* is more than twice as long as the distance *from* z^* *to* v.
Be sure to state/explain the running time of your algorithm (faster – and correct – is worth more credit).