A network flow instance is a graph \((V, E)\) with two special nodes \(r, s \in V\) and non-negative edge capacities, \(u_{ij}\) and \(u_{ji}\) for each \((i, j) \in E\).

**Problem 1.** Given is a network flow instance. A min-capacity of a path \((i_0, i_1, \cdots, i_k)\) is 
\[
\min_{j=1}^{k} u_{i_{j-1}i_j}.
\]
Describe an efficient algorithm, similar to Dijkstra’s that finds a path from \(r\) to \(s\) with the maximum min-capacity.

You can use a pseudocode from Lecture 30, but you need three changes: (1) The algorithm should return "NO PATH" if there is no path from \(r\) to \(s\), (2) it should return min-capacity and a list of nodes on such a path if it does exists, (3) of course, a small change is needed so you compute a min-capacity path rather than a shortest path.

**Answer.** For each node \(i\) we will have the previous node on the best path found so far (if any), \(\text{Prev}[i]\), and min-capacity of that path, \(\text{MinCap}[i]\). Because min-capacity, the quality measure for our paths, is better for larger values, we will be using a priority queue with \(\text{DeleteMax}(Q)\) operator, which in queue \(Q\) removes and return a node with the maximum value of \(\text{MinCap}[i]\).

```plaintext
for (every node \(i\))
    \(\text{MinCap}[i] = 0;\)
    \(\text{MinCap}[r] = \infty;\)
initialize \(Q\) with all nodes;
while (\(Q\) is not empty) {
    \(i = \text{DeleteMax}(Q);\)
    for (every edge \((i, j)\)) {
        \(q = \text{minimum(\text{the capacity of } (i, j), \text{MinCap}[i])};\)
        \(\text{if } (q > \text{MinCap}[j])\)
            \(\text{MinCap}[j] = q,\)
            \(\text{Prev}[j] = i;\)
    }
}
if (\(\text{MinCap}[s] == 0\))
    no path to \(s;\)
else
    follow \(\text{Prev}[i]\) from \(s\) to \(r;\)
```
Problem 2. Show an example of a network flow instance such that after
(a) finding an augmenting path from \( r \) to \( s \) with the maximum min-capacity,
(b) changing the capacities according to the flow sent on that augmenting path,
(c) finding again an augmenting path from \( r \) to \( s \) with the maximum min-capacity,
the path found in (c) has a larger min-capacity than the path found in (a).

Answer. Our network has 4 nodes, \( r, s, a, b \). Edges \((r, a), (a, b), (b, s)\) have capacity 2. Edges \((r, b), (a, s)\) have capacity 3. Edge \((b, a)\) has capacity 1.

Initially, we cannot reach \( s \) from \( r \) using edges of capacity 3, and we can find a path with edges of capacity at least 2. We can choose path \((r, a, b, s)\).

Afterwards, the capacities of \((r, b), (a, s)\) are still 3, and the residual capacity of \((b, a)\) increases to 3. Thus \((r, b, a, s)\) becomes a path with min-capacity 3.

Problem 3. We consider a network flow instance and shortest augmenting paths, i.e. paths from \( r \) to \( s \) with positive min-capacity and the minimum number of edges, say \( k \).

a. Suppose that \((i, j)\) is an edge on a shortest augmenting path and \( u_{ji} = 0 \). Show that even if we increase \( u_{ji} \) to a positive value, \((j, i)\) cannot belong to an augmenting path of length \( k \) or less.

Answer. Suppose that shortest paths from \( r \) to \( s \) with all edges having positive capacities have \( L \) edges, \((i, j)\) belongs to such a path, and the shortest path from \( r \) to \( i \) has \( K \) edges. Then the shortest path from \( r \) to \( j \) has \( K + 1 \) edges, and the shortest path from \( i \) to \( s \) has \( L - K \) edges. Consequently, if we increase the capacity of \((j, i)\) to positive, the shortest path from \( r \) to \( s \) that uses this edge will first go to \( j \), using \( K + 1 \) edges, then \((j, i)\), and then from \( i \) to \( s \) using \( L - K \) edges, for the total of \( L + 2 \) edges.

b. Show how to find all edges that belong to shortest augmenting paths in time \( O(|E|) \).

Answer. First we use breadth first search to find \( Dist[i] \) for every node, the length of a shortest path from \( r \) to \( i \). Next, we use breadth first search in the graph with all edges reversed to find \( RDist[i] \) for every node, the length of a shortest path from \( i \) to \( s \) (or from \( s \) to \( i \) using the reversed edges). Then we have a test if an edge \((i, j)\) belongs to a shortest path from \( r \) to \( s \): it is so if \( Dist[i] + RDist[j] = Dist[s] - 1 \).

c. Assume that all positive capacities are equal to 1. Show how in time \( O(|E|) \) one can find shortest augmenting paths \( P_1, P_2, \ldots \) such that after changing capacities according to the flow sent through these paths there is no augmenting path of length \( k \) or less. (Perhaps it would be simpler to say that in time \( O(|E|) \) you should find a maximal set of shortest augmenting paths.)

Answer. First we find the distance of \( s \) from \( r \), say \( L \), and using the method of \( b \) we find the edges that are on paths from \( r \) to \( s \) of length \( L \). Let \( E_{short} \) be the set of these edges.

Next we find paths from \( r \) to \( s \) in \((V, E_{short})\), and we delete all edges encountered during the search. In this way, we inspect each edge once. We can achieve it using a variant of depth first search that starts at \( r \) and which is interrupted when it visits \( r \). If a call \( dfs(j) \) made from \( dfs(i) \) does not visit \( s \) then there is no path from \( j \) to \( s \) in \((V, E_{short})\)
— because of the edge deletions made before — so it is valid to delete all edges encountered during that call, as well as the edge \((i, j)\). If we reach \(s\) we interrupt the search, we add the path to the collection of shortest path and we delete its edges.

This would not be valid in an arbitrary flow network, but when all capacities are equal to 1, sending a flow of 1 over an edge reduces its capacity to 0, so we can delete that edge. Later we can use the reverse of this edge, but not in a path of length at most \(L\).

d. Use the result of c. to show that given an instance \((V, E)\) of a bipartite matching problem in time \(O(\sqrt{|V||E|})\) we can find a matching \(M\) such that the shortest augmenting path for \(M\) has at least \(\sqrt{|V|}\) edges.

e. If \(M^*\) is a maximum matching, and for matching \(M\) there are no augmenting paths with less than \(\sqrt{|V|}\) edges, then \(|M^*| - |M| < \sqrt{|V|}\). Use this implication, and the result of d. to show that a maximum matching in a bipartite graph can be found in time \(O(\sqrt{|V||E|})\).

**Answer.** We create a network that corresponds to the maximum matching: edges from \(r\) to \(A\) with capacity 1, edges from \(B\) to \(s\) with capacity 1, and for an edge \(\{i, j\} \in E, i \in A, j \in B\) we create edge \((i, j)\) with capacity 1 and \((j, i)\) with capacity 0.

Now we can apply the method of b and c to find all augmenting path of the minimum length in time \(|E|\). When we repeat it, the minimum length of the augmenting path is larger. After we repeated it \(\sqrt{|V|}\) times, the minimum length of augmenting paths is more than \(\sqrt{|V|}\). Until then, we use \(O(\sqrt{|V||E|})\) time. Afterwards, it suffices to search for augmenting paths using breadth first search, and can succeed at most \(\sqrt{|V|}\) times, so again, we use \(O(\sqrt{|V||E|})\) time.
Problem 4. Find a maximum matching and a minimum cover in the bipartite graph below.

Problem 5. In a tournament \( n \) players, 1, \( \ldots \), \( n \), play each other once, and there are no ties. Let \( w_i \) be the number of wins of player \( i \). Describe and analyze an algorithm that determines if a vector \((w_1, \ldots, w_n)\) is a possible outcome of such a tournament. The algorithm should run in time that is polynomial in \( n \).

Answer. First we check an easy condition: altogether, players play \( n(n - 1)/2 \) times, so the total number of wins is \( n(n - 1)/2 \), so we check if \( \sum_{i=1}^{n} w_i \) is equal to \( n(n - 1)/2 \). If no, we can answer in the negative.

Otherwise we persevere. We create a network flow instance such that the vector \((w_1, \ldots, w_n)\) is a possible outcome if and only that network has a flow with value \( n(n - 1)/2 \). We have 4 “layers” of nodes in the network: (a) source \( r \), (b) game nodes \( g_{ij} \) where \( 1 \leq i < j \leq n \); (c) player nodes \( p_i \) where \( 1 \leq i \leq n \), (d) sink node \( s \). Edges \((r, g_{ij})\) have capacity 1; edges \((p_i, s)\) have capacity \( v_i \), and from \( g_{ij} \) we have two edges of capacity 1: \((g_{ij}, p_i)\) and \((g_{ij}, p_j)\).

The idea is that the flow is an accounting of game results. Because capacities are integer, a maximum flow has integer values. We must have an inflow of 1 to every game node. Then the outflow from a game node goes to the winning player. The number of wins that player accumulates is the inflow of his node, and the outflow has to be equal to \( v_i \).