Problem 1 (10 points).

Given a directed graph $G = (V, E)$, with a source vertex $s \in V$ and a sink vertex $t \in V$, we define two paths from s to t are *vertex-disjoint*, if these two paths share no vertex except s and t. Describe a polynomial-time algorithm that finds the maximum number of vertex-disjoint paths from s to t.

Solution. We transform this problem into the max-flow problem. We first build network $G' = (V', E', s', t', c')$ from the given graph $G = (V, E)$ as follows. For each original vertex $v \in V$ we add two new vertices v_1 and v_2 to V', and then connect them with a new edge (v_1, v_2) added to E'. For each original edge $e = (u, v) \in E$ we add edge (u_2, v_1) to E'. The source s' of G' will be s_1 and the sink t' of G' will be t_2. All edges in G' has a capacity of 1, i.e., $c'(e) = 1$ for any $e \in E'$.

We now run any max-flow algorithm on G' to find a max-flow f^*. Since all capacities of G' are integers (i.e., 1), we can assume that $f^*(e) \in \{0, 1\}$ for any $e \in E'$. Hence, $|f^*|$ must be integer as well.

We now prove that $|f^*|$ equals to the maximum number of vertex-disjoint paths from s to t in the original graph G. In fact, edges in G' with $f^*(e) = 1$ form $|f^*|$ *vertex-disjoint* paths in G' from s' to t', because each vertex in G' is either have in-degree of 1 or out-degree of 1; these $|f^*|$ paths therefore correspond to $|f^*|$ vertex-disjoint paths in G, by simply concatenating v_1 and v_2 into v. On the other side, if there exists k vertex-disjoint s-t paths in G, then they can be transformed into k vertex-disjoint s'-t' paths in G'. Combined, we conclude that $|f^*|$ equals to the maximum number of vertex-disjoint s-t paths in G (and these paths can be fetched following edges with $f^*(e) = 1$).

The running time of above algorithm consists of building network G', which takes $O(|V| + |E|)$ time, and running one max-flow algorithm, which takes polynomial-time.

Problem 2 (10 points).

You are given a directed graph $G = (V, E)$ with positive integral capacity $c(e)$ on $e \in E$, a source $s \in V$, and a sink $t \in V$. You are also given an integral maximum s-t flow f of G. Now we pick one edge $e^* \in E$ and reduce its capacity by 1. Show how to find a maximum flow in the resulting network in time $O(|E| + |V|)$.

Solution. Let $e^* = (u, v)$. Denote by G' the resulting network after reducing the capacity of edge e^* by 1. Clearly the value of the maximum flow of G' is at most $|f|$.

Since f is an integral max-flow of G, we have only two cases: either $f(e^*) \leq c(e^*) - 1$ or $f(e^*) = c(e^*)$. If it is the first case, i.e., $f(e^*) \leq c(e^*) - 1$, then f is still a flow of G', and hence a maximum flow of G'.

Now let's consider the case that $f(e^*) = c(e^*)$. In this case f is not a (valid) flow of G' anymore (the capacity constraint on edge e^* is broken). We use the following algorithm to find the maximum flow of G'. We run DFS (or BFS) to find a path p_1 from s to u, and a path p_2 from v to t, that only use edges $e \in E$ with $f(e) \geq 1$. Such paths p_1 and p_2 must exist, simply because $f(e^*) = c(e^*) \geq 1$. We then consider following two cases.

Suppose that p_1 and p_2 are vertex-disjoint. Then we have a *simple* path p from s to t: p_1 followed by edge (u, v) followed by p_2. Let f' be the flow through decreasing $f(e)$ by 1 for all edges on path p: we set $f'(e) = f(e) - 1$ for edges $e \in p$, and $f'(e) = f(e)$ for edges $e \notin p$. Clearly, $f'(e)$ is a (valid) flow of G', and $|f'| = |f| - 1$. We then build the residual graph $G'_{f'}$ of G' w.r.t. f', and use BFS to decide whether there exists an augmenting path q from s to t in $G'_{f'}$. If such path q can be found, we can then augment f' using q to get a new flow f'' with $|f''| \geq |f'| + 1 = |f|$. Notice that we must have $|f''| = |f|$ and f'' is a maximum flow of G'. If such augmenting path cannot be found, then this proves that f' is a maximum flow of G'.

Suppose that p_1 and p_2 are not vertex-disjoint. Let w be a vertex that is in p_1 and in p_2. Then we can find a cycle C: $w \to u \to v \to w$. Recall that $f(e) \geq 1$ for all edges $e \in C$. Therefore, we build a new flow f' with $f'(e) = f(e) - 1$ for all $e \in C$ and $f'(e) = f(e)$ for all edges $e \notin C$. Clearly, f' is a flow of G' and $|f'| = |f|$. Hence, f' is a maximum flow of G'.

The above algorithm uses a constant number of BFS (or DFS) and therefore runs in $O(|E|+|V|)$ time.

Problem 3 (20 points).

Given s-t network $G=(V,E)$, design a polynomial-time algorithm to compute $\cap_{C=(S,T)\in\mathcal{C}} S$ and $\cup_{C=(S,T)\in\mathcal{C}} S$, where $\mathcal{C}$ denotes the set of *all* minimum-cut of G. Prove that your algorithm is correct.

Solution. The algorithm to compute $\cap_{C=(S,T)\in\mathcal{C}} S$ and $\cup_{C\in\mathcal{C}} S$ is quite simple. We use any max-flow algorithm to compute one max-flow f and its corresponding residual graph G_f. Let A be the vertices that can be reached from s in G_f, and let B be the vertices that can reach t in G_f. A and B can be computed by BFS on G_f. Then we have that $\cap_{C=(S,T)\in\mathcal{C}} S = A$ and $\cup_{C=(S,T)\in\mathcal{C}} S = V-B$. The running time of this algorithm is dominated by the running time of the max-flow algorithm.

We first prove that $\cap_{C=(S,T)\in\mathcal{C}} S = A$, and the other part can be proved symmetrically. First, we show that if $v \in \cap_{C=(S,T)\in\mathcal{C}} S$ then we have $v \in A$. In fact, $(A, V-A)$ is a minimum s-t cut (proved in class). Thus, we have that $v \in A$, as v belongs to the s-side of *every* minimum s-t cut. Second, we prove that if $v \in A$ then we have $v \in \cap_{C=(S,T)\in\mathcal{C}} S$. We prove this by contradiction. Assume that $v \notin \cap_{C=(S,T)\in\mathcal{C}} S$. Then there exists one minimum s-t cut $C' = (S', T')$ such that $v \in T'$. Since $v \in A$, there must be one path in G_f from s to v. This path must go through some cut-edge e of C' since v is in the t-side of C'. This means that e is not saturated by f, which is a contradiction with the fact that C' is one minimum s-t cut.

Remark: This statement implies that the minimum-cut is unique if and only if $A \cup B = V$.

Problem 4 (20 points).

There are n guests, m different white wines and m different red wines in one party. Each guest has a preference list that specifies the wines he/she likes. A guest is *well served* if he/she is served both white wine and red wine, and both of them are in his/her preference list. Each kind of wine can be served to at most b guests. Devise a polynomial-time algorithm to compute the maximum number of guests that can be well served.

Solution. This problem can be reduced to a network flow problem. The network G has one vertex w_i for the ith white wine, one vertex r_j for the jth red wine, $1 \le i, j \le m$, two vertices u_k and v_k for the kth guest, $1 \le k \le n$, and source s and sink t. The edges of the network are (s, w_i) with capacity b, (w_i, u_k) with capacity 1 if ith white wine is in the preference list of kth guest, (u_k, v_k) with capacity 1, (v_k, r_j) with capacity 1 if jth red wine is in the preference list of the kth guest, and (r_j, t) with capacity b.

We run any algorithm to compute the maximum flow f^* of the above network $G=(V,E)$. Since all capacities of G are integers, $|f^*|$ must be an integer and $f^*(e)$ must be an integer too for any $e \in E$. We now prove that $|f^*|$ is the maximum number of the guests that can be well served. This is rather straightforward. On one side, suppose that k guests can be well served. Then we can construct a flow f of G with value $|f| = k$, simply setting $f(w_i, u_k) = 1$, $f(u_k, v_k) = 1$, and $f(v_k, r_j)$ if kth guest is well-served and served with ith white wine and jth red wine, and setting $f(s, w_i)$ and $f(r_j, t)$ to the proper value to balance w_i and r_j. This proves that the maximum number of guests can be well served is a lower bound of $|f^*|$. On the other side, we can find $|f^*|$ guests that can be well served following f^*: those guests with $f^*(u_k, v_k) = 1$ can be well served and the corresponding wines can be fetched following $f^*(w_i, u_k) = 1$ and $f^*(v_k, r_j) = 1$. This proves that the maximum number of guests can be well served is an upper bound of $|f^*|$. Combined, $|f^*|$ equals to the maximum number of the guests that can be well served.

The network has $(2m+2n+2)$ vertices. Thus, the above algorithm runs in polynomial-time (in n and m).

Problem 5 (20 points).

You are given an $n \times n$ binary matrix M, unlimited number of $k \times 1$ column vectors and $1 \times k$ rows vectors for all $1 \le k \le n$. Each length k column/row vector can be used to cover continuous k elements of one

column/row of M. Devise a polynomial-time algorithm to compute the minimum number of vectors such that each 0 is covered by at least one vector and that each 1 is not covered by any vector.

Solution. First, we find all possible locations of those vectors, which are continuous 0s in M, since 1 is not allowed to be covered. Clearly, we only need to consider those longest continuous 0s, since each 0 can be covered many times. Now we build a bipartite graph $G = (X \cup Y, E)$. Each vertex in X corresponds to one vertical continuous longest 0s, and each vertex in Y corresponds to one horizontal longest 0s. Two vertices are connected if the corresponding column segment and row segment share one element.

The minimum vertex cover of the bipartite graph gives the minimum vectors needed. This is because there exists a one-to-one correspondence between a vertex cover of G and how to select vectors to cover all 0s in the matrix. The running time of above algorithm is dominated by finding a minimum vertex cover of G, which can be done in polynomial-time.

Problem 6 (20 points).

Let $G = (V, E)$ be a directed graph. Let $A \subset V$ and $B \subset V$ be two *disjoint* subsets of vertices. We say there exists a *migration* from A to B if there exists a set of paths P such that (1) each vertex in A is the first vertex of one path in P, (2) the last vertex of each path in P must in B, and (3) paths in P do not share any edges.

1. Given G, A and B, design an algorithm to decide whether there exists a migration from A to B.
2. Replace the condition (3) as: paths in P do not share any *vertex*; design an algorithm to decide whether there exists a migration from A to B.

Solution.

1. We transform this problem into a max-flow problem. We construct a network $G_1 = (V_1, E_1, s, t, c)$, where $V_1 = V \cup \{s, t\}$. We add all edges in E to E_1 with $c(e) = 1$ for any $e \in E$. We add a new edge (s, u) to E_1 with capacity 1 for all $u \in A$, and add a new edge (v, t) with capacity of infinity (or a big number such as $|V|$) for all $v \in B$. We then run any max-flow algorithm to find the maximum flow f^* of G_1. Clearly, there exists a migration from A to B if and only if $|f^*| = |A|$. The running time of this algorithm is dominated by the max-flow algorithm.
2. (Here we use the same technique with Problem 1.) We construct a network $G_2 = (V_2, E_2, s, t, c)$. For each vertex $v \in V$, we add two vertices v_1 and v_2 to V_2. We further add source s and sink t to V_2. For any vertex $v \in V$ we add new edge (v_1, v_2) to E_2 with capacity 1. For any edge $e = (u, v) \in E$ we add new edge (u_2, v_1) to E_2 with capacity 1. We add a new edge (s, u_1) to E_1 with capacity 1 for all $u \in A$, and add a new edge (v_2, t) with capacity of infinity (or a big number such as $2 \cdot |V|$) for all $v \in B$. We then run any max-flow algorithm to find the maximum flow f^* of G_2. Clearly, there exists a migration (vertex-disjoint) from A to B if and only if $|f^*| = |A|$. The running time of this algorithm is again dominated by the max-flow algorithm.

Problem 7 (20 points).

Suppose that a single entrance, single exit maze can be represented by a directed graph $G = (V, E)$ with the entrance being $s \in V$ and exit being $t \in V$. There are K people at the entrance want to escape (at time 0). Every hour, each person can traverse an edge to a neighboring vertex or stay where they are. However, the edge $e \in E$ has an associated capacity $c(e)$, meaning that at most $c(e)$ people can traverse the edge during the same hour. Design an algorithm to calculate the minimum number of hours for all people to escape and analyze your algorithm's running time.

Solution. We first define a sub-problem: in t hours what is the maximum number of people can escape the maze. We transform this subproblem into a max-flow problem. We construct a network $G' = (V', E')$ where

$V' = \{(v,i) \mid v \in V, 0 \leq i \leq t\}$. For all $0 \leq i < t$, we add an edge to E' from (v,i) to $(v,i+1)$ with capacity of infinity; this indicates that people can stay at vertex v in one hour. For every edge $e = (u,v) \in E$ in the original graph, we add an edge to E' from (u,i) to $(v,i+1)$ with capacity $c(e)$; this indicates that people can travel to next vertex in one hour. Let the source of G' be $(s,0)$ and let the sink of G' be (v,t). We use any max-flow algorithm to find the maximum flow f^* of G'. Clearly, $|f^*|$ gives, in t hours, the maximum number of people who can escape. Running time: there are $t \cdot |V|$ vertices and $t \cdot (|V| + |E|)$ edges; if we use push-relabel algorithm, the running time of this algorithm is $O(t^3 \cdot |V|^2 \cdot (|V| + |E|))$.

Algorithm for the entire problem: notice that the maximum number of hours is bounded by $(|E| + K)$. This is because in $|E|$ hours, one people must be able to escape, and therefore all K people can escape in $|E| + K$ hours following the same route one by one. We can do binary search between 0 and $(|E| + K)$ to find the minimum t. Thus, we can solve the whole problem by running above algorithm for solving the subproblem $O(\log(|E| + K))$ times.