1: Unique Minimum Spanning Tree

(a) We can prove this using the Kruskal's algorithm. In every iteration we look for the edge with the smallest weight and collapse the two end vertices of that edge. We also perform the necessary clean-up. If we run the algorithm for the graph G that has edge weights as distinct integers, then the first iteration will pickup the smallest edge. We don't have any choice in this regard. After collapsing and clean-up we again have a new G' with distinct edges. We pick the smallest edge in this graph and continue to G'' and so on.

Kruskal's algorithm is guaranteed to return a MST. In addition, for G, there is only one possible execution of the algorithm (discussed above) which will give only one MST. Therefore this is the only MST for G and hence is unique.

(b) Algorithm: Let's say the original graph is $G = (V, E)$. We first find out a MST using

Kruskal's or Prim's algorithm. Let's say the MST is $M = (V, F)$. Next pick an edge from the set $E \backslash F$ (the edges which are not part of the MST we found). Let's say such an edge is (u, v) and it's weight is w. Since MST is a tree, there must be a path in M between u and v. Find that path and check if the minimum weight on this path is w. If there is, then we can say that the G does not have a unique MST, otherwise repeat this process for a different edge in $E \backslash F$. If we can't satisfy the above condition for the entire set $E \backslash F$, then return that G has a unique MST.

Correctness: We first have an MST. The algorithm tries to remove an edge in the MST and insert another edge of same weight, so that we get another MST. The edges in $E \backslash F$ are not part of M. We look for an edge e with weight m in the path $u \to v$ in M. If there is such an edge, then adding the edge (u, v) will create a cycle. But we can remove e and insert (u, v), still keeping the graph connected and maintaining the same total weight. In this way we get a different MST.

Running time: Kruskal's or Prim's algorithm takes polynomial time to compute an MST. This MST will have exaclty $n - 1$ edges, therefore, the set $E \backslash F$ will have $(m - n + 1)$ edges. We go through each of these edges and try to find a corresponding path in the MST which can be done polynomial time using depth-first algorithm. This process is repeated for $(m - n + 1)$ times, bounding the whole process to a polynomial runtime.

2: Max Flow Basics

(a) This can be proved by showing that the Ford's algorithm execution has integer flows at every iteration and the algorithm ultimately terminates. We can prove the first part by induction. At the very first iteration (base case), all the edges have integer capacities and flows are 0 (integer). Let's assume that at i^{th} iteration, there are integer flows on every edge. Then for the $i + 1$ iteration, the bottleneck for an augmenting path can only be a integer (as bottleneck is the difference of f_e and c_e of some edge e at iteration i). For all the forward paths on this augmenting path, the flow is increased by this bottleneck capacity (which is again an integer). For all the backward paths, decrease the flow by the bottleneck value (which is again an integer). Therefore, at every step of the execution, the flows on every edge is an integer.

Let's say that the total capacity of all the edges coming out of the source is X. The while loop in

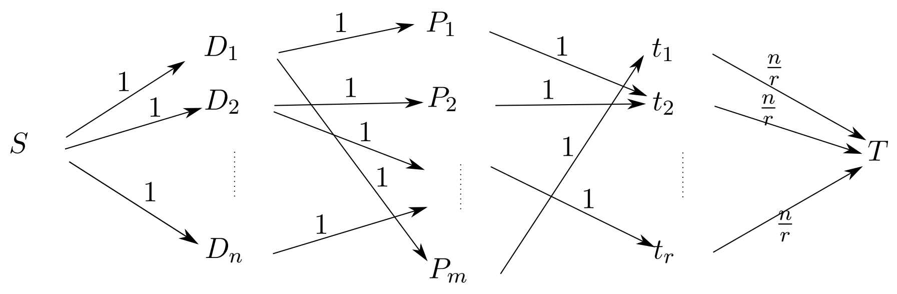

Figure 1: Faculty Committee

the Ford's algorithm is guaranteed to terminate after X iterations, if at each step the total flow increases by 1. Using these two proofs, we can say that when the algorithm terminates, all the flows will have integer values and this flow will be maximum.

(b) Increase capacity by 1: Algorithm: We try to find an augmenting path in G' (the graph with the edge e whose capacity has been increased by 1). If there exists such a path, return max-flow as $C+1$ for G', where C was max-flow of initial graph G. If we cannot find such a path, then return C as max-flow of G'.

Correctness: The flow of the original graph was C, therefore the value of min-cut for that graph was also C. Now this increase in the capacity of the edge e can cause min-cut to change to another one with value atmost $C+1$ (note that if e does not become the part of a new min-cut, the previous min-cut still has the same value). So the flow can either remain the same or increase by atmost 1. Now if an augmenting path exist in the G', it will go through e, thereby increasing the flow to $C+1$. If such a path doesn't exist, the flow is still the same as C.

Running time: We just need to find a new augmenting path in G' which is a $O(n+m)$ operation (n is the nodes, m is the vertices).

Decrease capacity by 1: Algorithm: Let's say the capacity of edge e (u,v) in G is reduced by 1. If the flow through this edge was smaller than the original capacity, then decreasing its capacity does not affect the max flow; return the original flow C for G'. Else if there is a path from $u \to v$ such that we can increase the flow along this path by 1, we can return the original flow C as flow for G'. Otherwise if no such path can be found, then increase the capacity of the backward edge (v,u) by 1 and then find the path p_1, from u to s, path p_2, from t to v (s and t are source and sink respectively). Send one unit of flow back to source from sink along the path p_2, $v \to u$, p_1. By performing this operation we get a graph with flow $C-1$ and with the modified edge. Next try to find an augmenting path from s to t. If such a path is found then return flow as C, otherwise $C-1$.

Correctness: If the flow of e in the original graph was less than its capacity, it means that this edge was not the part of the min-cut and therefore, decreasing its capacity will not make any difference on the value of min-cut (and hence max-flow). For the second case, we have a violation on the edge e where its flow now becomes greater than its capacity. So either we can redirect this flow along some path from u to v; in which case the net flow remains the same and we have a legal residual graph. But if even that's not possible, we should nullify this violation. This is acheived by sending one unit of flow from t to s. The paths p_1 and p_2 must exist because we had a positive flow from s to t via e. Now we have a graph with a legal flow of $C - 1$. Since we can atmost have a flow of C for G', we find just one augmenting path in this graph. If found, it means we are able to void the affect of decresed capacity of e by some path otherwise $C - 1$ is the new capacity

Running time: We are doing two path searches from t to v, u to s which can be done by depth-first search algorithm. We also need to search for an augmenting path which can again be done by depth-first search algorithm. So in total atmost three depth-first search is required and the process is bounded by $O(m + n)$.

(c) Let's assign capacity 1 to each one of the edges in the graph. Choose any pair (s, t) and run the Ford' algorithm to find max-flow from s to t, considering only the outgoing edges from s and only the incoming edges for t. Let's say this flow is m. Since we assigned evey edge a capacity of 1, there must be m edge-disjoint paths from s to t. Similarly find max-flow from t to s, and let's say that this flow is k, indicating that there are k edge-disjoint paths from t to s. Now let's consider the three cases, $m > k$, $m = k$, and $m < k$. For our solution we will try to prove that the first and the last cases can't exist and in fact will reduce to case 2.

Consider case 1, where the incoming edges for t must be atleast m (as there are m edge-disjoint paths from s to t). Also the outgoing edges from t must be k (which enters s). So there are atleast $m - k$ edges for t that should not enter s (due to max-flow). In a similar fashion there are $m - k$ edges which are incoming to s but must not be from t. We will use contradiction to say that these outgoing $m - k$ edges from t must enter the source s. Let's say that the $m - k$ outgoing nodes from t enter a set of nodes T and the $m - k$ edges incoming into s are coming from a set S. Our graph cannot have sources and sinks as indegree of each node is equal to its outgoing degree. So if the set S is not receiving these $m - k$ nodes from T, it means that there are some nodes with indegree $\neq$ outdegree. Therefore, there must be $m - k$ paths from T to S. These paths should be disjoint otherwise their count is less $m - k$. Hence, m must be equal to k.

We don't need to prove anything for case 2. Case 3 can be proved by using the above reasoning just in the opposite direction.

3: More Reductions to Flow

(a) Algorithm: Let's say the departments are $D_1, D_2, \ldots, D_n$, faculty members are

$P_1, P_2, \ldots, P_m$, and there are r ranks. First check if $m \geq n$, if it is not return such a committe is not possible (as there is not even n faculty members). For the other case, we can reduce this problem to that of finding max-flow in a graph shown in Figure 1. Create nodes $D_1, D_2, \ldots, D_n$ corresponding to the departments, $P_1, P_2, \ldots, P_m$ corresponding to the faculty members, and $t_1, t_2, \ldots, t_r$ corresponding to the the ranks. Also include two extra nodes S and T as source and sink respectively. Connect S to each of $D_1, D_2, \ldots, D_n$ with capacity of each node 1. Connect D_i to P_j if P_j has an appointment at D_i, with capacity of each edge as 1. Next connect P_i to t_j if the rank of P_i is t_j, with capacity of each edge as 1. Finally, connect $t_1, t_2, \ldots, t_r$ to T, , with capacity of each edge as $\frac{n}{r}$. Find the max flow for this graph, and if it is n then return the solution as the set of faculty members whose corresponding $P_i \to t_j$ has a flow of 1. If the max flow is not n, then return such a committee is not possible.

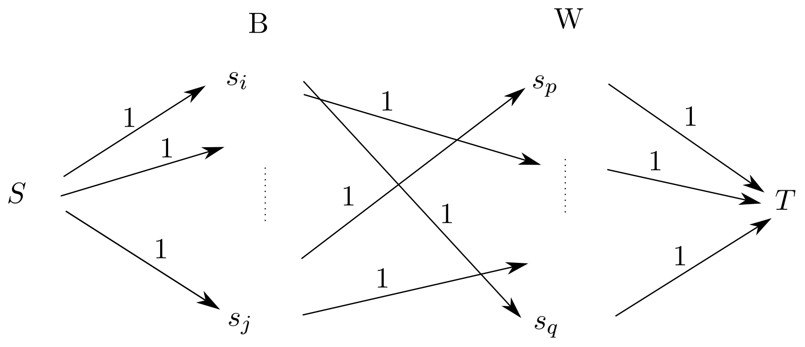

Figure 2: Chessboard Problem

Correctness: The sum of capacities of edges from $S \rightarrow D_i$ is n, therefore max-flow cannot be greater than n. If max-flow $= n$, it means that all the edges $t_i \rightarrow T$ have flow $\frac{n}{r}$. Therefore, each t_i must have exactly $\frac{n}{r}$ edges with flow 1. Also notice that each of these edges will be coming in from only one faculty as a faculty can have only one rank. Therefore, for all ranks t_i there will be exactly n number of P_j that have flow 1 (which is our solution). These P_j will have inflow 1. Each department has inflow 1 (as for max-flow all $S \rightarrow D_i$ edges have flow $= 1$). Therefore, only one edge coming out of a department can have flow $= 1$. This will ascertain that only one distinct faculty is selected for each department. However, if the max-flow is less than n, then it means there are less than n edges with flow 1 from P_j to the set of t_i. This in turn means we don't have a selection of n faculties.

Running time: We only need to setup the graph which can be done in $O(mn)$ time (time to connect $D_i \rightarrow P_j$). The max-flow can be computed in polynomial time using Ford's algorithm (any poly-time algo will work). From the residual graph we can easily pick the $P_i \rightarrow t_j$ edges with flow 1 which can be done in $O(m)$ time. Therefore, overall the algorithm is polynomial time.

(b) Algorithm: Let's say that there are m squares left in the chessboard, and we name them as

$s_1, s_2, \ldots, s_m$. First we perform a check if m is odd. If it is, we return such tiling is not feasible. Else, this problem can be solved by reducing to finding max-flow in a graph shown in Figure 2. Create nodes $s_1, s_2, \ldots, s_m$ corresponding to the m squares. Also include two more nodes S and T as source ans sink respectively. Connect S to all the s_i that are black in color and connect all the s_j that are white in color to T. A particular black square s_i can atmost have four possible white square neighbours for tiling (vertically up and down, horizontally left and right). Connect all the black squares s_i to all possible white square neighbours with an edge. The capacity of each edge is 1. Find the max-flow of this graph. If the max-flow $= \frac{m}{2}$, then return such a tiling is possible, else return such a tiling is not possible.

Correctness: Since we have already established that there are even no. of squares, there will be exactly $\frac{m}{2}$ black and white squares. Therefore, the sum of edges $S \rightarrow s_i$(black squares s_i) is exactly $\frac{m}{2}$ and so is the sum of edges $s_j \rightarrow T$ (white squares s_j). The sum of edges (black)$s_i \rightarrow s_j$(white) is atleast $\frac{m}{2}$ as each square has atleast one neighbour. Therefore, max flow

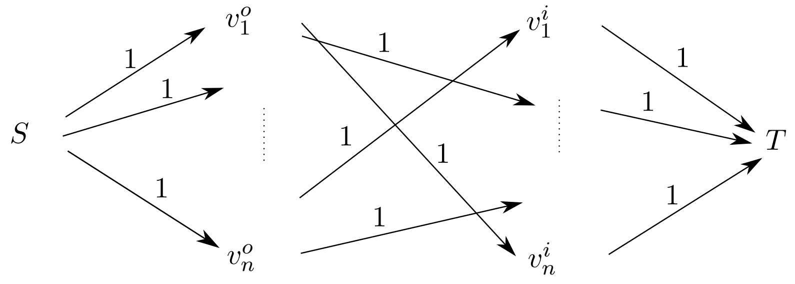

Figure 3: Cycle Cover

cannot exceed $\frac{m}{2}$. When the max-flow is indeed $\frac{m}{2}$, all the outgoing edges from S and incoming edges into T have flow 1. This signifies that all the squares are covered. The legality of this covering is established by the fact that of all the outgoing edges from a black square, only one can have a flow 1. This edge will also be the only edge with flow 1 for a white square. These two squares can be covered by a single tile. There will not be any overlapping as for each white square there will only one and distinct black square. If the max flow is not $\frac{m}{2}$, then it means atleast one black square and one white square does not have inflow and outflow as 1 and it means they can't be covered.

Running time: We can setup this graph in $O(m^2)$ time. Solving the max-flow is polynomial. The solution of the max-flow is directly translated to the solution of our problem if max flow is $\frac{m}{2}$ or not.

(c) Algorithm: First check if $|V| \leq |E|$ ($n \leq m$), if it is not then return such a cycle cover is not possible (as there must be atleast n edges to have a cycle that can cover all nodes). For the other case, we split each of the $\{v_1, v_2, \ldots, v_n\}$ nodes into $\{v_1^o, v_1^i, v_2^o, v_2^i, \ldots, v_n^o, v_n^i\}$. This problem can be solved by reducing to finding max-flow in a graph shown in Figure 3. Create two nodes S and T as source ans sink respectively. Connect S to each one of v_k^o nodes and connect each one of v_k^i to T. Connect nodes v_k^o to v_j^i if there was a directed edge in the original graph from v_k to v_j. Compute the max flow for this graph. If the max-flow is n, then such a cycle cover exists. In the residual graph for the max-flow, there will be exaclty n edges of type $v_k^o \rightarrow v_j^i$. Return the corresponding set of edges $v_k \rightarrow v_j$ as the solution (if $v_k = v_j$, it implies a self loop). If the max-flow is not equal to n, then such a cycle cover does not exist.

Correctness: The sum of edges $S \rightarrow v_k^o$ is n and so is the sum of edges $v_j^i \rightarrow T$, therefore, we cannot have a flow more than n. When we have a flow of n, it means each of the edges $S \rightarrow v_k^o$ and $v_j^i \rightarrow T$ have a flow of 1 (this guarantees each node is selected in our solution). This implies that only one of outgoing and incoming edges for the nodes v_k^o and v_j^i respectively can have a flow of 1. As a result we will have n edges of type $v_k^o \rightarrow v_j^i$. Now consider only the corresponding edges $v_k \rightarrow v_j$ in the original graph. We have n edges and each vertex has exactly one edge coming out and one edge coming in. This is only possible if this set of edges form some cycles and two cycles can't have a same node because each node is visited exactly once. This is exactly our solution. If max-flow is less than n, then atleast one path $S \rightarrow v_k^o \rightarrow v_j^i \rightarrow T$ has a flow 0 and it means that

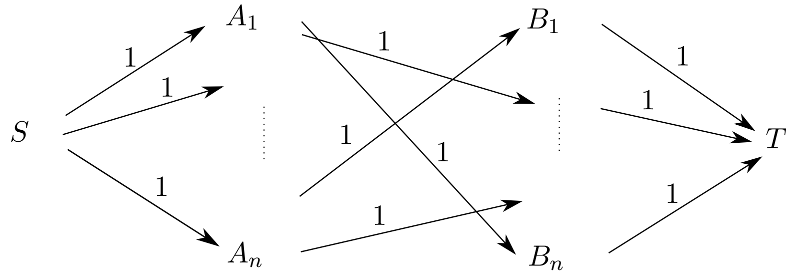

Figure 4: Perfect matching Actresses and Actors

one node only has an incoming edge and another node only has an outgoing edge and they are not part of a cycle.

Running time: We are creating a new graph with $2n + 2$ nodes and atmost $2n + m$ edges which are polynomial in n and m (where n and m were the nodes and edges in original graph). The max-flow algorithm takes polynomial time to compute. The algorithm also produces the residual graph, and we can pick the edges $v_k^o \to v_j^i$ with flow 1 as our solution (change them to $v_k \to v_j$).

(d) Let's say there are n actresses and n actors. We have the graph as specified in the problem statement shown in Figure 4 with two extra nodes S and T acting as source and sink. There is a directed edge from A_i to B_k if they have co-appeared in a movie. The problem of perfect matching is equivalent to having a flow of n (which is max-flow for this graph) in this graph where the capacity of each edge is 1. Suppose, we find a flow of n in this graph. It means all the edges $S \to A_i$ and $B_j \to T$ have flow 1. This implies that exactly one of the outgoing edges from A_i and one of the incoming edges into B_j can have a flow 1. In other words, there are exaclty n edges of type $A_i \to B_j$ that have a flow 1. So Bob can create such a graph and check for max-flow $= n$. If it is, then for a particular actress A_i named by Alice, he should reply with a actor B_j, such that the edge $A_i \to B_j$ had a flow of 1 in the final residual graph. This way Bob has answer to every actress Alice names. Since Alice started the game, at some point the whole list of actresses would be exhausted and she won't be able to reply to last actor named by Bob.

Bonus: One possible case for not having a perfect matching (or max flow $= n$) is that there is some m actors who have incoming edges from atmost a set of $m - 1$ actresses. Since there are equal number of actresses and actors, there must be atleast actress who has not co-appeared with any actor. In that case, Alice can start with her and win immediately. The only other case is when there are m actresses who have outgoing edges to a set of $m - 1$ actors. Alice should start with an actress from this set and will win as Bob will not be able to reply to m^{th} actress.

4: Unexpected Reductions to Flow/Matchings

(a) If we consider a node in the graph and analyze the difference of the edges coming in from s and going into t, it gives us $d_i - 2\lambda$. Now if this value is ≥ 0, it means that this node has atleast 2λ nodes in the whole graph. For nodes where this value is less than 0, it means that this node is connected to less than 2λ nodes. Now for the first case, the inflow from s will be greater than the outflow to t and the other way round for the second case. Therefore, for the nodes of the first case, if send in the max-possible flow from s, we can distribute it to some other nodes of second case. Therefore, after running the Ford's algorithm the nodes whose outgoing edge to t have flow equal to its capacity, can be a part of S. The reasoning is that there are some nodes with edges more than 2λ and some with less than 2λ, but the average of 2λ is maintained.

On this possibly feasible S, we can then find out the connected components and see if the number of edges in a connected component is $\geq \lambda S$. If it is, we can return this as the solution, otherwise if we cannot find any such connected component, return such a set is infeasible.

We need to run Ford's algorithm, which is polynomial. The possibly feasible set S can atmost have n nodes. The connected components in this set can also be computed in polynomial time.

(b) Algorithm: The problem wants to minimize the total floor space used. Let's say initially we

laid out every tile on the floor without stacking. For this configuration, we say that we have 0 area savings. If we stack t_1 over t_2, then we have a saving equal to the area of t_1. Let's create a graph from these (assumingly) n tiles. There are two nodes for each tile t_i and t'_i. We will only have edges between nodes of type t_i and t'_j, where $i \neq j$, when t_j can be placed over t_i (simply or by rotating). The weight of an edge of this type will be the area of t'_j, which signifies the saving we make if we stack t_j over t_i. For the example given in the question, let's say the nodes are $A, B, C, D, A', B', C', D'$. The edges will be $(A, B', weight = 150), (A, C', weight = 75), (A, D', weight = 100), (B, C', weight = 75), (B, D', weight = 100)$. So we have a bipartite graph with t_i on one side and t'_i on other side. Solve this graph for weighted maximum bipartite matching. The solution will be a set of type (t_i, t'_j). Now pick up one pair, say (t_1, t'_2), it means that t_2 can be stacked over t_1. Next search for a pair that has first element t_2. If it exists, it means that the second element can be stacked over t_2. If it does not, it means that nothing needs to be stacked over t_2. So for every pair keep creating a list of the type $t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_k$. All the nodes that do not appear either as first or second element in the pairs cannot be stacked over anyone or cannot accomodate another tile. They just need to be kept on the floor without being part of any stack.

Correctness: The purpose for any stacking is to reduce space in optimal way. The algorithm works by maximizing the savings acheived from stacking. By building this graph we enumerate all the possible stackings and the savings offered by them. The graph is bipartite as there will not be edges between nodes of type t_i and t_j or t'_i and t'_j. The weighted bipartite matching will now try to maximize the saving, which is our objective. The set of edges returned by maximum matching algorithm can be used build the stack order.

Running time: For n tiles we have $2n$ nodes and atmost n^2 nodes. The bipartite matching algorithm is polynomial in n and n^2 which is again polynomial. The algorithm will return atmost $\frac{2n}{2} = n$ edges. In the case of perfect matching, we will have to search a possible stacking for each n node in n edges which is bounded by $O(n^2)$. Hence the total runtime of the algorithm is polynomial in the number of tiles.