1: unique minimum spanning trees

(a) we shall use Prim's algorithm to find all the minimum spanning trees (MST) for the given undirected graph. let us assume for now that we have got two MST's $S1$ and $S2$. given that they are two distinct MST's, at least one of the edges between them should differ. let us consider an edge with minimum weight e_{s1} and it can only be part of either $S1$ or $S2$ and let's assume it's part of $S1$.

If we add this unique edge into $S2$ it will convert the tree into a cycle. similarly there will be one more edge e_{s2} in $S2$ which is not part of $S1$ and adding this to $S1$ should convert this to a cycle as well.

Now keeping in mind the claim that all the edges in graph have distinct weights, and having considered the first edge to be the smallest $W(e_{s1}) < W(e_{s2})$, and also considering both $S1$ and $S2$ as MST's, let's construct a third MST $S3$ from $S2$ by replacing edge e_{s2} with e_{s1}. This will result in a MST whose weight is smaller than the total weight of $S2$, but this is a contradiction, since we claimed that $S2$ is a MST. Hence we can construct only one unique MST for a graph with distinct weights, thus $S1 = S2 = S3$. Also, if we are to use an efficient algorithm like Kruskal's which picks the smallest edge every time from a given graph, then we are sure to end up in a unique MST as every edge has a distinct weight, which will be a direct implementation.

(b) we shall again use Prim's algorithm and find all the MST's for the given undirected graph. For a given a graph $G = (V, E)$, let $S1 = (V, F)$ be the MST with F edges and let's assume that $F < E$.

Let us consider an edge $e_E = a, b \in E$ but not in F and let's assume that it is the edge with minimum weight $W(e_E) = M$. If we add this edge in to our MST $S1$, this will form a cycle C in $S1$ with previous F edges.

Now the claim is that if we cannot find another edge, which is of same weight M, then we have a cycle, while if we can find another edge with same weight M, we can create a second MST $S2$ by swapping that edge from $F \cap C$ with our edge e_E of weight M. Thus, we cannot have uniqueness in that case.

2: Max Flow Basics

(a) The claim puts a constraint only on the capacity of the edge being integer, logically it can also be a rational number as the final flow. we shall prove this by induction. Let's take a graph $G = (V, E)$ for which at the first iteration, all flows will be zero(base case). For this case, if the original edges have integer capacities, then residual graph resulting from augmented path will also have all flows as integer capacities. This happens because every augmenting path increases flows by an integer amount and not by rational amount. Furthermore, if there exists a integral maximum flow F_m, which essentially means that there doesn't exist any other augmenting path along $\overrightarrow{F}$. Suppose if we assume that there exists an $\overrightarrow{F}$ -augmenting path, it contradicts the claim that $\overrightarrow{F}$ has an integral flow of maximum value, because the new flow could be integral when $\overrightarrow{F}$ is integral. Hence there is no $\overrightarrow{F}$ -augmenting path, and $\overrightarrow{F}$ must be a maximum flow.

(b) From a given graph $G = (V, E)$, we have the residual graph which has maximal flow defined

from $s \mapsto t$. let's consider $u, v \in E$ as the original edge and $\overrightarrow{u}, \overrightarrow{v}$ as the compliment edge in residual graph and M as the maximal flow.

Algorithm: edge weight increase

from the residual graph

search for a maximum flow path along $\overrightarrow{u}, \overrightarrow{v}$ owing to "contraintsi"

if found:

$\overline{M} = M + 1$

else:

do nothing //retaining the original maximal flow

"constraintsi":

-if $\overrightarrow{u}, \overrightarrow{v}$ was the bottleneck in the maximal flow path

Correctness: if there is a updated maximal flow path then there must be an additional flow size of 1 as all flows are integers. if any of the "constraintsi" are not satisfied, then $\overrightarrow{u}, \overrightarrow{v}$ will not be the bottleneck and hence the maximal flow will still be the original.

Running time: only the search along residual flow paths can be done in linear time traversing from $s \mapsto t$ along all nodes and edges and hence the run time will be $O(V + E)$.

Algorithm: edge weight decrease

from the residual graph, after we decrease the unit flow, we know that there is one additional unit of flow coming into node $\overrightarrow{u}$ and one unit less flowing into node $\overrightarrow{v}$.

In residual graph search a path from $\overrightarrow{u}$ to $\overrightarrow{v}$ along which we can increase a flow by one:

if found:

redraw residual with the one additional flow coming into $\overrightarrow{u}$ diverted along new path

$\overline{M} = M$

else:

reduce flow from s to $\overrightarrow{u}$ and $\overrightarrow{v}$ to t

$\overline{M} = M - 1$

constraints:

-to check if $\overrightarrow{u}, \overrightarrow{v}$ is already not utilizing the full capacity

-if $\overrightarrow{u}, \overrightarrow{v}$ was the bottleneck in the maximal flow path

-to check if there exists a path from $\overrightarrow{u}$ to $\overrightarrow{v}$ which can increase flow by one

-finding a reverse path for flow decrement from $\overrightarrow{u}$ to s and t to $\overrightarrow{v}$.

Correctness: for the first condition where we retain the maximal flow is subject to finding a path from $\overrightarrow{u}$ to $\overrightarrow{v}$, while the second condition is a guaranteed reduction as we are sure to find a reverse flow along $\overrightarrow{u}$ to s and t to $\overrightarrow{v}$ as we already had a original flow M accounting for that additional one flow from s to t along $\overrightarrow{u}$ to $\overrightarrow{v}$ in the input residual graph.

Running time: only the search along residual flow paths can be done in linear time traversing from $s \mapsto t$ along all nodes and edges and hence the run time for this will also be $O(V + E)$.

(c) for a given graph $G = (V, E)$, let us assume these parameters.
c_e - capacity of each edge
v_i - in degree of every vertex
v_o - out degree of every vertex
$s_i = m$ - in degree of source
$t_i = n$ - in degree of sink
s_o - out degree of source
t_o - out degree of sink
m - number of edge disjoint paths from s to t
k - number of edge disjoint paths from t to s
and let's assume $m > k$.

we shall prove this by contradiction, let us assume that we already have m edge disjoint paths from $s \mapsto t$, and we are given that in-degree of every node is equal to it's out degree, thus $s_i = s_o$, and $t_i = t_o$. since there are m-edge disjoint paths from s to t and given that $m > k$, we can assume that there are additional $m - k$ paths coming out of t which are not going back to s, while k of them are going to s. Now these $m - k$ paths need to go out of t into a cluster of nodes where they need to terminate. similarly there should also be $m - k$ paths coming into s from a different emitting cluster. now when we consider these two emitting/terminating clusters, unless these clusters are self contained (independent of $s \leftrightarrow t$ network) or have a emitting/terminating nodes these cannot be considered as part of our network. ignoring the first condition, the second condition will be a direct contradiction of $v_i = v_i$ constraint. hence these clusters should merge somewhere with $m - k$ outgoing paths from t ending up in the cluster of $m - k$ requirements from s. thus total clusters going out of t will be $m - k + k$ which is m itself which is nothing but the total edge disjoint paths we considered from s to t.

3: more reductions to flow

(a)

Algorithm:

let $D_1, D_2...D_n \in D$ be the departments, while $F_1, F_2...F_m \in F$ be the faculty members and let's consider $R_1, R_2...R_r = R$ ranks among these faculty. from the given constraints, we will first check if $m \geq n$, if this condition fails, then we say that such a committee cannot be formed as no 2 departments can be represented by the same faculty. if not, we will do the following -
we shall create a tripartite graph where we will have the first partition mentioning all the departments D and the second partition mentioning all the faculty F, while the third partition will have all ranks R . Now we will have two addition nodes s before first partition and a sink node t after the third partition. now we shall draw an edge from s to all D with unit capacity, now from D, we will have an edge with unit capacity to every faculty F who are associated with the said department. Now, we shall have a unit flow edge from every faculty F to their respective ranks R (note: we can have only one edge from a faculty to a rank as he cannot hold more than one ranks). Now finally, we will have an edge with capacity $\frac{n}{r}$ from every rank node to sink t. now if we apply the max flow algorithm over this graph, we should be able to find a committee if

and only if the algorithm returns the output n which basically means there exists a path from every department to faculty to their individual ranks in the maximum flow graph, if the out put is anything less than n, then return saying that such a committee is not possible for the given input.

Correctness:

given that each department has at least one faculty representative and we need representation from every rank as well, let's compute the total capacities flowing through each partitions. we can have exactly n total flow coming into partition D, while we will have at least n capacity from D to F as each department can have multiple associated professors. furthermore, we will have exactly m flow coming into R from F as each faculty can hold only one rank, and finally we have exactly n flow coming into t from R. the idea was to make sure that we have representation from every department and every ranks without having same faculty representing two departments, hence our min-cut will be n. by adding unit flow across all partitions except for R to t, we are making sure that only one maximal flow selection is propagated till partition R. Now, if the maximum flow is less than n, then it means there are less than n edges with flow 1 from F to R, which in turn means we don't have a selection of n faculties, while having n flow essentially means that every rank of faculty with individual flow $n \div r$ was selected for representation and hence we can form a committee.

Running time: :

the graph setup will take $O(mn)$ time which is the time to connect every department with faculty. The maximum flow can be computed using any poly-time algorithm(Prim's for example). From the residual graph we can pick the faculty to rank mapping edges with flow 1 which can be done in $O(m)$ time. Hence, overall the algorithm is polynomial time.

(b)

Algorithm:

After mutilating the nxn chessboard, let's assume that we are remaining with r squares, and let's name them as $s_1, s_2, s_3..s_r$. First, we shall check if the resulting number of squares is odd or even, if in case the number is odd we revert saying that such a domino placement is not possible as the domino sizes are $2x1$. if in case the number of remaining squares is even, then we shall go ahead and solve this using bipartite max-flow graph. we will create a bipartite graph with first partition having nodes from all squares s_i which are black, similarly we will create a second partition having all nodes s_j which are white. we shall also create two additional nodes, source s and sink t. Now for a given black node, we will draw an unit flow edge into second partition across white for every node which is either on (top/bottom/right/left) side of the given black node. After connecting all black nodes this way, we will run the max-flow algorithm on this graph. If the max-flow returns $\frac{r}{2}$, then return such a domino tiling is possible, else return such a domino tiling is not possible.

Correctness:

For given even number(r) of squares remaining on a chessboard, we know that there will be exactly $\frac{r}{2}$ black and white squares. since the partitions will have equal nodes in the graph and with unit flow across partitions, the min-cut will be $\frac{r}{2}$ and hence max flow cannot exceed $\frac{r}{2}$. The upper limit of $\frac{r}{2}$ will ensure that of all the outgoing edges from a black square, only one can have a flow 1 which will also mark the same for a white square and this way it will make sure that these nodes are covered by a single domino tile and it will also avoid any overlapping as for each

white square, there will be only one and distinct black square. If the max flow is not $\frac{r}{2}$, then it means at least one black square and one white square does not have and edge between/across them which essentially means that these two squares are at positions which cannot be covered by a single tile and hence the whole board tiling will be deemed incomplete.

Running time:

We can setup this graph in $O(r^2)$ time and solving for max-flow can be done using any polynomial time algorithm. hence, the solution boils down to the point that if we can establish a max flow of $\frac{r}{2}$ or not.

(c)

Algorithm:

we need at least $V + 1$ edges to construct a cycle cover to accommodate all nodes, hence we will check if $V \leq E$(let's assume n is the total number of nodes), if not we will exit saying that we cannot create such a cycle cover, if it is, then we go ahead with the below procedure -
From the given graph G, we will create a intermediate graph where we will split each vertex into v_i^{in} and v_i^{out}.add a unit edge between every split vertices and for every other original connections, we will add an outgoing directed edge from v_i^{out} to v_j^{in} for an outgoing edge from vi from original graph, similarly we will add an incoming directed edge from v_j^{out} to v_i^{in} for every incoming edge to vi. Now, from this intermediate graph we will build a bipartite graph where every we will have first partition with all v_i^{out}'s mentioned and second partition with all v_i^{in}'s mentioned. we will add an additional source node s before first partition and connect s to all nodes in first partition with unit flow, similarly we will add a sink node t after the second partition and add a unit flow node from all of second partition nodes into t. between partitions, we shall add an unit flow edge for every directed edge from the intermediate graph. Now, this problem has been reduced into a max-flow problem and we will compute the max-flow for this graph. If the max-flow returns n, then such a cycle cover exists as n will be our min-cut (n nodes in each partitions). If the max-flow is not equal to n, then such a cycle cover does not exist.

Correctness:

since the partitions will have equal nodes in the graph and with unit flow across partitions, the min-cut will be n and hence max flow cannot exceed n. If we have a max-flow of n, it means each of the edges from first partition have flow of 1 (this guarantees each node is selected in our solution). This also implies that only one of outgoing and incoming edges within the partition can have a unit flow. As a result we will have n edges between the partitions. when contemplating the max-flow to original graph, we can see that there were originally 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 and there are n edges. If max-flow is less than n, then at least one of the paths between partitions doesn't have a unit flow which in-turn denotes that the given node only has an incoming edge and another node only has an outgoing edge and they don't result in a cycle cover.

Running time:

We can setup the new graph with $2V$ nodes and $V + E + V$ edges which are polynomial. we can compute the max-flow using any polynomial time algorithm.

(d)

We are already given a bipartite graph where we have actresses represented as n nodes on first partition (F), and actors also mapped as n nodes in second partition (S) along with source s and sink t nodes as well. There exists an unit flow edge between the nodes in partitions if and only if they have co-appeared in a movie. Let's say the graph has a perfect matching which essentially means that there is a max-flow n from the graph, which in-turn signifies that there is exactly one matching unit flow edge between the nodes in partition and Bob can fix this perfect matching by selecting a node in F_i such that there will be a predictable perfect matching for the next guess. so essentially if Bob can creates such a graph and checks for max-flow of n, then for any particular actress F_i named by Alice, he will reply with a actor S_j, such that there exists an edge from $F_i \mapsto S_j$. This way Bob will have a counter to every actress Alice names. Since Alice started the game, at some point the whole list of actresses will get exhausted and she won't be able to reply to last actor named by Bob.

Bonus
Again, let's consider the same bipartite graph, however because there is no perfect matching this time, we have two nodes which don't have a directed edge from some F_i to S_j. Now Alice's strategy would be to select the same node F_i which has no edge toward any B_j.

First, when the game starts, Alice picks a node F_i which is unmatched. Such a node will exist for sure as our graph does not have a perfect matching. Now, Bob has to choose a node S_j for which we can have one of the below two cases :

1. There exists no edge between F_1 and any of the unvisited node. In this scenario, Alice will win as the game rule says we cannot choose a node already selected.

2. If there exists such a node S_j, then it must be matched. This is because, since F_i is unmatched and if S_J is not matched, then there can exist a new matching edge towards one of F_k. Now, since S_j is matched, Alice can now choose an edge from this node i.e S_j. Now, since this will be repeated after every move of Bob, Alice can play her strategy which is mentioned above.

4: unexpected reductions

(a)

Algorithm:

For the given bipartite graph, we will run the maximum flow algorithm. for any given community, the constraint of at least 2λ nodes connected within the community is an average and should not be considered as a tight bound. now once we start analysing the graph, we will realize that any degree for a given vertex with $d_i - 2\lambda$ will be the edges going out of the community and should not be linked to any nodes within the community. once the algorithm is executed on the said graph, we will find all the edges of type $v_i \mapsto t$ that have flow equal to the capacity of edge, i.e. $f(v_i \mapsto t) = c(v_i \mapsto t)$. The set of v_i satisfying this equation is the set S.

Correctness:

for the given graph, we have $d_i - 2\lambda$ as the last flow min-cut towards sink t, hence this will be our max-flow value which will determine whether or not such a community for a given λ exists. This constraint λ will always determine the bounding box under which the algorithm runs. for every

vertex within the community that needs to be determined, we will have a edges of value $d_i - 2\lambda$ for a given i^{th} vertex which will not be part of the community.

Running time:

for setting up the new graph, we have $V + 2$ node generations, and $2V + 2E$ edge generations. we can use any polynomial time max-flow algorithm on the graph.

(b)

Algorithm:

we will create a bipartite graph for the given n tiles. while laying tiles we shall use these ground rules. 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. There are two nodes for each tile t_i and $\bar{t}_i$. We will only have edges between nodes of type t_i and $\bar{t}_j$, where $i \neq j$(not counting the same tile placements in each partition), when t_j can be placed over t_i (simply or by 90 degree rotation). The weight of an edge of this type will be the area of $\bar{t}_j$, which signifies the saving we make if we stack t_j over t_i. From the given examples, let's say the nodes are $A, B, C, D, \overline{A}, \overline{B}, \overline{C}, \overline{D}$. The edges will be ($A, \overline{B}$, weight = 150), ($A, \overline{C}$,weight = 75), ($A, \overline{D}$,weight = 100), ($B, \overline{C}$,weight = 75), ($B, \overline{D}$,weight = 100). So we have a bipartite graph with t_i on one side and $\bar{t}_i$ on other side. Solve this graph for weighted maximum bipartite matching. The solution will be a set of type (t_i to $\bar{t}_j$). Now pick up one pair, say $(t_1, \bar{t}_2)$, 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 $t_1 \mapsto t_2 \mapsto .. \mapsto t_k$. All the nodes that do not appear either as either first or second element in the pairs cannot be stacked over anyone or cannot accommodate 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 achieved from stacking. By building this graph we enumerate all the possible stacking 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 $\bar{t}_i$ and $\bar{t}_j$. 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 at most $4n^2$ nodes. The bipartite matching algorithm is polynomial in n and n^2 which is again polynomial. The algorithm will return at most $\frac{2n}{2} = n$ 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 run time of the algorithm is polynomial in the number of tiles.