1: Unique Minimum Spanning Trees

(a) Let G be a weighted, undirected graph with all edge weights being distinct integers. Prove that there is a unique minimum spanning tree.

Proof by contradiction: Assume there are two distinct MSTs T_1 and T_2. Let e_1 be the edge with the smallest weight that appears in T_1 but not T_2 or in T_2 but not T_1 (assume $e_1 \in T_1$ for the purpose of the proof).

Add e_1 to T_2. By the property of trees, if any edge is added to the tree a simple cycle C is formed.

We know C must contain at least one edge that is not in T_1 as T_1 contains e_1 and if it contained all the other edges in C, T_1 itself would have a cycle (which it can't it's a MST). Let this edge be e_2.

Using the fact $e_1 < e_2$ (e_1 is the smallest weighted edge that appears is only one of T_1 or T_2) if we remove e_2 we will create a spanning tree with a smaller weight than T_2. This is a contradiction as we stated T_2 is a MST.

Proving there is a unique minimum spanning tree if all edge weights are distinct.

(b) Algorithm: Recall that Kruskal's algorithm starts with each vertex in G being a disjoint

tree and all of the edges are sorted by weight. The lightest edge is removed and if it joins two disjoint trees into one tree it is added, otherwise it's discarded.

Run Kruskal's algorithm on G to find a minimum spanning tree. As each edge is added to the MST, mark this edge as visited in the original graph G.

Run Kruskal's algorithm on G a second time with a slight modification. Each time the lightest valid edge is found, check to see if there are edges with the same weight that have not been marked as visited. One at a time, attempt to add the unvisited edge(s) to the MST. If they don't form a cycle and can be added to the MST, we have discovered there is not a unique MST. Otherwise discard them as they don't belong in the MST and repeat this process for the next lightest edge.

If Kruskal's algorithm finishes and all the unvisited duplicate weight edges are invalid for the MST, we know there is a unique MST.

Correctness: We know that Kruskal's algorithm is guaranteed to find a MST. It accomplishes this by making greedy choices considering the lightest edge not already in the MST.

In the our algorithm, we know the first pass of Kruskal's will produce a valid MST. In the second pass of Kruskal's, if there are multiple lightest edges, we are free to choose which lightest edge to consider first and still produce an MST. Thus if it is valid to include an edge in the second pass, that weren't used in the first pass, there exists more than one MST.

If there are no valid edges to include on the second pass, that weren't used in the first pass, there is a unique MST.

Running time: Kruskal's algorithm runs in $O(E \log V)$ time on the first pass. The modified Krusakal's algorithm for the second pass requires us to look forward for duplicate weighted edges. But while checking these edges, if they can't be added to the MST, they are discarded. Therefore, we still only touch each edge once. Thus the runtime is two passes of Kruskal's algorithm or $O(E \log V)$.

$\boxed{O(E \log V)}$

2: Max Flow Basics

(a) Proof by induction

Basis: Before the first iteration of the Ford-Fulkerson algorithm, no augmenting paths have been found from s to t. It follows then that the flow from s to t is initially 0 which is an integer.

Inductive step: Assume the kth iteration of the Ford-Fulkerson algorithm resulted in an integral flow. Given this, all residual capacities for all edges must be integral as adding or subtracting integers results in an integer.

On iteration $k+1$ of the Ford-Fulkerson algorithm the found augmenting path from s to t will contain a bottleneck edge with a integral capacity. Given a flow with an integral capacity as a bottleneck, the flow will be integral.

Thus maintaining an integral flow on every edge. Proving that for any s, t there exists a maximum flow between s, t in which every edge has an integral flow on it.

(b) Algorithm:

Increase one edge's capacity by 1

When the capacity of one edge is increased 1, the max $s-t$ flow **might** increase. This is because there might now exist a new augmenting path from s to t. To update the max $s-t$ flow, one iteration of the Ford-Fulkerson algorithm is ran. That is, a depth first search is ran in the residual graph in an attempt to find a new augmenting path.

We know that Ford-Fulkerson will increase the $s-t$ flow by at least one, if possible, each iteration. As only 1 edge gained 1 capacity, the most the max $s-t$ flow can increase by is 1. Therefore, one iteration of the Ford-Fulkerson algorithm is sufficient.

Decrease one edge's capacity by 1

There are two cases when decreasing the capacity of an edge by 1.

case 1: The edge has a capacity greater than 0 before reducing it by 1. If this is the case, none of the flows between edges change, no other capacities of edges change and no back edges change. Since the flow of none of the augmenting paths from s to t change, max $s-t$ flow doesn't change. Therefore nothing has to be done other than reducing the capacity of the given edge by 1.

case 2: The edge has a remaining capacity 0 before reducing it by 1. If this is the case, this edge is the, or one of, the bottle neck edges of each augmenting path it belongs to. We must find **one** of the augmenting paths it is the bottleneck of and reduce the flow of each edge in this augmenting path by 1 (and update the capacities and back edges). This is only required for one augmenting path as the updated edge at most reduced the max $s-t$ flow by 1.

We have reduced a flow from s to t by 1, overall reducing the max $s-t$ flow by 1. But, there might be a new route for the reduced augmenting path that doesn't need the edge that lost 1 capacity. To determine this, one iteration of Ford-Fulkerson is ran. This will update the max $s-t$ flow by at least one and will properly update the new max $s-t$ flow.

Correctness:

Increase one edge's capacity by 1

Given 1 extra capacity in an edge, the max $s-t$ flow can be greater. This is a result of the edge could be the only bottle neck in an augmenting path from s to t or create a new augmenting path from s to t. Ford-Fulkerson is guaranteed to increase the max $s-t$ flow by at least one each iteration if possible. Therefore we know the change will be found.

Decrease one edge's capacity by 1

case 1: Once the max $s-t$ is found, there can exist remaining capacities of edges in the residual graph that are not used. Decreasing one these remaining capacities by 1 won't change any of the augmenting paths from s to t thus the max $s-t$ flow will not change.

case 2: A remaining capacity of 0 of an edge in the residual graph indicates that a set of augmenting paths from s to t contain this edge. Therefore, decreasing the capacity by 1 means one of these augmenting paths will lose 1 flow. Ford-Fulkerson is guaranteed to increase the max $s-t$ flow by at least one each iteration if possible. Thus we know the lost flow can be recovered if a different augmenting path from s to t exists.

Running time: One iteration of the Ford-Fulkerson algorithm consists of running a search of the graph. This can be done with time complexity $O(|V|+|E|)$. In the case of reducing the capacity by 1 when the remaining capacity is already 0, we must search for an augmenting path. This requires traversing some $O(|E|)$ back edges. Thus in all this algorithm has a $O(|V|+|E|)$ time complexity.

$\boxed{O(|V|+|E|)}$

(c)

Proof *there exists k edge disjoint paths from $s-t \Rightarrow$ there exists k edge disjoint paths from $t-s$.*

We know all edges have capacity 1. There exists k edge disjoint paths from $t-s$ iff the max $t-s$ flow is k. That is, there are k augmenting paths from t to s, each with a flow of 1.

The *in-degree* of every vertex is equal to its *out-degree*. Therefore, for every augmenting path from $t-s$ there is an augmenting path from $s-t$. We know that there are k augmenting paths from $t-s$ so it follows there are k augmenting paths from $s-t$.

Therefore, if there exists k edge disjoint paths from $t-s$, there also exists k edge disjoint paths from $s-t$.

Proof *there exists k edge disjoint paths from $t-s \Rightarrow$ there exists k edge disjoint paths from $s-t$.*

Proving the iff statement from right-to-left follows the exact logic from proving the iff statement from left-to-right.

3: More Reductions to Flow

(a) Constructing the graph

step 1: Starting with a source s, create an edge with capacity $\frac{m}{3}$ to each of three vertices representing the ranks *assistant*, *associate*, and *full* professors.

step 2: Create m vertices to represent each of the faculty members. Create an edge of capacity 1 from their rank to the vertex that represents them.

step 3: Create n vertices to represent each of the departments. Create an edge of capacity 1 from each faculty member to **each** department they work for.

step 4: Finally, create a sink t and add an edge of capacity 1 from each department to the sink.

Determining if a committee exists

Run the Ford-Fulkerson algorithm to find the max flow of the constructed graph. If the max flow is equal to n (the number of departments) we can conclude a committee is possible. Otherwise we can conclude it is impossible to find such a committee. Note that a committee with equal representation from all ranks of facility isn't possible if the number departments isn't a multiple of 3.

Finding the committee

Finding the committee if it exists is simple. Consider each flow from a rank vertex to a faculty member vertex. If the flow is equal to one, that faculty member belongs on the committee.

Correctness: We know that the Ford-Fulkerson algorithm will find the maximum flow. We limit the flow from s by $\frac{m}{3}$ for each of the ranks, upholding the restriction of equal representation for the committee. Additionally, we will have at most n flow as there is n possible flow into the sink t. Finally, each faculty member can only represent one department as they only have at most 1 flow flowing through their vertex.

Running time: Constructing the graph with be linear with respect to the number of departments and faculty with time complexity $O(n+m)$. The edges of the flow graph have an integer capacity so Ford-Fulkerson is bounded by $O(E*M)$ where E is the number of edges and M is the maximum flow. The maximum flow will be n and size of E will be of the order $O(n+m)$. Thus the total runtime will be $O((n+m)n)$.

$\boxed{O((n+m)n)}$

(b) Constructing the graph

step 1: Create a vertex representing each black square that exists on the checkerboard and create an edge with capacity 1 from a source s to each black vertex.

step 2: Create a vertex representing each white square that exists on the checkerboard and create an edge with capacity 1 from each white vertex to a sink t.

step 3: Finally, for each adjacent neighbor a black square has (up, down, left, and right of the square - the neighbors will always be white squares) add an edge of capacity 1 from the black vertex that represents that square to it's white vertex neighbor.

Determining if the board can be tiled

Run the Ford-Fulkerson algorithm to find the max flow of the constructed graph. If the max flow is equal to half the number of squares on the checkerboard, the board can be tiled. Any max flow that is odd or less than half the number of squares indicates tiling the board is impossible.

Correctness: We know each domino must occupy exactly two adjacent squares of the checkerboard (a pair of adjacent squares will always be one white and one black square). Additionally, each square may only be occupied once.

In the constructed flow graph, an augmenting path from $s-t$ represents placing a domino on two

squares and contributes 1 to the total flow from $s - t$. These squares may not be used in any other augmenting paths as the edges have capacity 1. If the checkerboard has s squares, we know we have a solution when exactly $s/2$ dominos have been placed. This is identical to having a max-flow of $n/2$.

Running time: Constructing the graph will be linear with respect to the number of squares on the checkerboard with time complexity $O(s)$ where s is the number of squares. The edges of the flow graph have integer capacities so Ford-Fulkerson is bounded by $O(E * M)$ where E is the number of edges and M is the maximum flow. The maximum flow will be $s/2$ and size of E will be of the order $O(s)$. Thus the total runtime will be $O(s^2)$.

$\boxed{O(s^2)}$

(c) Constructing the graph

step 1: For each vertex $v_i \in V$, create two vertices in the flow graph: $from_i$ and to_i.

step 2: Create a source s and add an edge with capacity 1 from s to each vertex $from_i$.

step 3: For each directed edge(u, v) in G, add an edge to the flow graph with capacity 1 between $from_u$ and to_v.

step 4: Create a sink t and add an edge with capacity 1 from each vertex $from_i$ to t.

Determine if a cycle cover exists

Run the Ford-Fulkerson algorithm to find the max flow of the constructed graph. If the max flow is equal to $|V|$, a cycle cover exists. Otherwise, it's impossible.

Find the cycle cover

If the max flow of the constructed flow graph is equal to $|V|$, a cycle cover exists. Each augmenting path from $s - t$ in the residual network of the flow graph pass through two vertices that represent an edge in the original graph G. For example, vertices $from_1$ and to_2 represents a directed edge from vertex 1 to vertex 2.

The collection of augmenting paths from $s - t$ in the residual network that constitute the flow represent the edges that form a *cycle cover* of G. This collection of edges forms a set of vertex-disjoint cycles in the graph G that covers all the vertices.

Correctness: A *cycle cover* of a directed graph G is a set of vertex-disjoint cycles in the graph that covers all the vertices. Each vertex may only be visited once and all vertices must be visited.

In our constructed flow graph, we applied the constraint each vertex can only contribute a flow of 1 to the total from from $s - t$ by giving the edges out of s and the edges into t a capacity of 1. Therefore, the only way to achieve a max flow of $|V|$ is if every vertex is visited.

Running time: Constructing the graph will be linear with respect to the number of vertices in G with time complexity $O(m + n)$ where m is the number of vertices in G and n is the number of edges in G. The edges of the flow graph have integer capacities so Ford-Fulkerson is bounded by $O(E * M)$ where E is the number of edges and M is the maximum flow. The maximum flow will be m and the size of E will be of order $O(m + n)$. Thus the total runtime will be $O((m + n)m)$.

$\boxed{O((m + n)m)}$

(d) Proof

Consider the graph G which is a perfect matching bipartite graph. That is, there exists a matching M where M is a set of edges from G s.t. no two edges share a common vertex. M is a perfect matching meaning it matches every vertex in G such that every vertex in G is incident to exactly one edge in M.

Bob's winning strategy: In this game Alice goes first and names an actress. For Bob to guarantee victory, Bob must answer with the actor matched with Alice's actress in the perfect matching. That is a $m \in M$ is an edge that connects an actress with an actor. It is a perfect matching so we know that for every actress Alice can name, there exists a matching actor that hasn't been named yet.

Bob wins this game as he goes second. In the worst case, Bob can simply respond with the matching actor from M until Alice and Bob have named every eligible actress and actor. At which point it's Alice's turn and she has no actress to name and loses the game.

(d bonus) Proof Now let's consider a graph G s.t. there is not a perfect matching. Recall a perfect matching M is a set of edges from G s.t. no two edges share a common vertex and it matches every vertex in G is incident to exactly one edge in M.

If a bipartite graph doesn't have a perfect matching it must contain a set of connected vertices that is preventing a perfect matching. That is, there is a set of vertices on the right hand side of the bipartite graph (the actor side) that are only neighbors with a set of vertices on the left hand side of the bipartite graph (the actress side). Additionally, the left hand side set contains more vertices than the right hand side set, or vice versa. We know they aren't equal as an unequal pair of sets must exist since there doesn't exist a perfect matching.

Put in terms of this question, if G doesn't have a perfect matching there will be pairs of set(s) of actors and actresses of this form. Where for each pair, one of the two sets is bigger than the other.

Alice's winning strategy: There are three cases that can occur as a result and Alice can leverage both of them to win the game.

case 1: G only contains pairs of sets s.t. the actor set is larger than the actress set. If this is the case, there will exist an actress s.t. there is no actor she has co-appeared with. Alice can simply pick this actress, Bob will have no response and Alice wins.

case 2: Otherwise G contains a pair of sets s.t. the actress set is larger than the actor set. In this case, Alice continues to pick valid actresses from this set depending on Bob's response. Since Alice's set is larger, Bob will run out of responses before Alice and Alice will win.

case 3: No pairs of sets could exist in G. Again, this would lead to Alice having an actress not connected to any actors and gives her an easy win.

Thus if G doesn't contain a perfect matching, Alice will always have a winning move and will win the game.

4: Unexpected Reductions to Flow/Matchings

(a) No Answer

(b) Algorithm: We know there exists an algorithm that solves a weighted version of the

maximum bipartite matching problem in polynomial time. We will construct a weighted bipartite graph and use the algorithm to maximize the amount of floor space saved thus minimizing the amount of total floor space used.

Constructing the weighted bipartite graph

step 1: For each of the tiles, create a vertex u_i and a vertex v_i.

step 2: Compare each of the tiles to every other tile. Each tile has the dimensions a_i x b_i. If either ($a_j \leq a_i$ *and* $b_j \leq b_i$) or ($a_j \leq b_i$ *and* $b_j \leq a_i$) is true then tile j can be placed on top of tile i. For each case where tile j can be placed on top of tile i, place a directed edge from u_j to v_i. The weight of this edge is the area of tile j, that is $a_j * b_j$.

Finding the optimal stacking

Once the weighted bipartite graph G is constructed, we run it through the weighted maximum bipartite matching algorithm. Let's assume a matching M is produced, which is a set of edges that constitute the maximum matching.

To produce the optimal stacking we consider each edge in M. An edge(u_i, v_j) signifies that panel i is placed on panel j in the optimal stacking. For each edge in M perform the stacking following this method and we produce the optimal stacking.

Correctness: We construct a bipartite graph representing all possible panels that can be stacked on top of another panel where the weight of the edge connecting these two panels is the amount of floor space saved by stacking them.

We give this graph to the maximum bipartite matching algorithm and are given a matching M. We know that in a matching, no two edges share a common vertex. This enforces each panel may only have one directly stacked on top of it and the algorithm will choose the panel that saves the maximum amount of floor space. Therefore, this set of stackings is valid and will minimize the amount of total floor space used.

Running time: When constructing the graph G, we just consider n^2 stackings of two tiles where n is the number of tiles when creating edges. There will $2n$ vertices in G. Giving a polynomial size graph that takes $O(n^2)$ time to create.

The maximum bipartite matching algorithm runs in some unknown polynomial time. Thus the runtime of the total algorithm is whichever is more complex: the creation of the graph or the bipartite matching algorithm.