CS6150 - Homework/Assignment-4

Arnab Das(u1014840)

November 4, 2016

1: Minimum Spanning Tree

(a) Let G be weighted directed graph with all edge weights being distinct. **Prove:** G has an unique spanning tree:
Proof: Consider G has two spanning trees, T_1 and T_2. Let e_1 be the edge with the minimum weight in the graph that is contained in T_1 and not in T_2. Then $T_2 \cup e_1$ results in a cycle, say C. Then one of the other edges of C, say e_2 is not in T_1, else T_1 would have had a cycle. Then, if we replace e_2 by e_1 in T_2, that would result in a spanning tree with a smaller weight than T_2. Then T_2 is not a spanning tree. Hence T_1 is a unique spanning tree. (Proved)

(b) Even if the weights of the graph are not distinct, it is still possible to get a unique spanning tree. We describe below an algorithm , that is a slight modification of the Kruskal algorithm. Given a weighted graph $G = (V, E, W)$, where V denotes the set of vertices, E denotes the set of edges and W denotes the edge weights. Consider initially a setting of all the nodes without any of the edges, each is a separate tree. Then we gradually build up the minimum spanning tree(mst) by union of these disjoint trees. We always start with the edge of minimal weight at every step, and if the end points of that edge lies in two different trees, then that edge acts as the union of these two trees resulting in a bigger tree. If , the end-points of the edge lie in the same tree, that edge, if added, will contribute to a cycle in that tree. In Kruskal's we ignore this edge. However, we can use this edge to detect non-uniqueness. Once a cycle is detected, then in that tree, we look for another edge that has the same weight as out current edge. If there exists another such an edge, that leads to the conclusion that it is not unique, since the other edge could have been replaced with the current edge, still yielding the minimum spanning tree.
PseudoCode:

```
KRUSKAL(G(V,E,W)):
    // Graph represented as triples(u,v,w), where u,v are the edge end-points and w the edge weight w(u,v).
    H=BuildHeap(G(V,E,W))
    foreach u ∈ V
        Make-Set(u)
    T = NULL // This corresponds to the spanning set.
    while (|T| = n-1) // if still spanning
        (u,v,w) = ExtractMin(H)
        Uset = Find(u)
        Vset = Find(v)
        if (Uset ≠ Vset)
            T = T ∪ (u, v)
            Merge(Uset, Vset)
        else // indicates part of same tree, hence contributes to a cycle
            bool x = search(w, Uset) // search for the same weight value in the Uset
            if (x=true) return MST_NOT_UNIQUE
    return MST_UNIQUE
```

Correctness: The correctness is same as Kruskal's algorithm. Consider that Kruskal's algorithm gives a minimum spanning tree, M=(V,F), where F is the set of the edges in the MST. Then, if there exists an edge, e, in E-F, such that adding e to the M yields a cycle, C, and if weight(e)=m is the lowest edge weight in $F \cap C$, then we can swap that edge of $F \cap C$ with e, yoielding another spanning tree. This is exactly the part executed by 'search' and hence guaranteed correctness. We searches for same weight edges in the Uset corresponding to the edge of the current edge when a cycle is detected, and if found we say the MST is not unique, else the MST is unique. Thus, we have modified the Kruskal's algorithm to dynamically find the MST uniqueness.

Running time: We use the heap structure to store the data, and use the weight of the edges as the keys for creating the heap. The heap creation takes time $O(|E|)$. The Kruskal's algorithm takes $(|E| + |V|)log|V|)$. Additionally , for the search operation, we need to perform a breadth first search on the subtree where a cycle is found. Breadth first search is $O(|E| + |V|)$, and atmost we do that for $(V - 1)$ times. So the overall bound comes to be polynomial.

2: Max Flow Basics . . .

(a) Given G is a directed graph with edges having **integer** capacity.
To Prove: For any s,t there exists a maximum flow between s,t in which every edge has an integeral flow on it.
Proof: The proof can be broken down into two parts. First we show that at every iteration iteration , the flow value on every edge f(u,v) and hence the flow of the augmenting paths are integers, and second that the iteration terminates.
First: Goal: At every intermediate step of the Ford-Fulkerson algorithm, the values f(u,v) and the residual capacities are integers.

At iteration 0, this is true by construction of the graph that the given flow capacities of every edge are integer values and the intialized flow capacities are zero. Now, suppose it is true after 'j' iterations. Then, if G_f is the residual graph after the j'th iteration, the bottleneck evaluated in the (j+1) iteration will also be integer since all the residual capacities in G_f and the flow values are integers. Now, since the capacities and the residual capacities are integers, then the flow on every edge must also be an integer as for a forward edge e,Residual Capacity = c_e - f(e), and for a backward edge of e, Residual Capacity = f(e), where c_e is the capacity of the edge e and f(e) is the flow value of edge e in j'th iteration. Then, in the (j+1) iteration, in the new flow over an augmenting path, we either increase f(e) for forward edges by the bottleneck amount which is an integer addition resulting in an integer, and decrease f(e) for backedges by the bottleneck amount which is an integer subtraction resulting in an integer. Thus the new flow and all the flow values on the edges will have integer values in the (j+1) iteration. This also leads to the fact, that the residual capacities we evaluate to build the residual graph will also be integer values since they are evaluated as the difference of the current capacities and the flow values which we proved to be integers. Hence, by induction, at every intermediate step, the flow values and the residual capacities are integers.
Second: Goal: The algorithm terminates.
Since, the flow value at every step is increased by the bottleneck amount, which is a positive integer quantity, and it cannot go beyond the out flow from the source, this leads to the fact that the while loop is guaranteed to terminate in atmost C steps, where C is the total outflow from the source.
Combining the **first** and **second** proofs, we can say, that the step at which it terminates, the result would have yielded an integer maximum flow and the flows on every edge will have integer values.

(b) Case-1: Increase Capacity of an edge by 1
Algorithm: As input, we are given the maximum feasible flow f in G, v(f) the value of f,an edge 'e' whose capacity is to be increased by 1, and we try to find a single augmenting path in G' which contains the edge 'e' whose capacity is increased by 1. We increase the capacity of the edge 'e' in the residual graph and find an augmenting path in the residual graph. If an augmenting path is found, it will have a bottleneck of '1' due to this increased capacity of 'e' and hence the flow value inctreases by 1. If an augmenting path is not found, then the flow value remains the same.

Correctness: Since, we have already reached the Maximum flow in G, thus there exists a s-t cut(A,B), such that all the paths crossing the cut are saturated and cannot carry any flow. Note that the maximum flow in G' is atleast v(f), since f is still a valid flow of the network as we haven't decreased the value of any capacity. All the edges across the cut have the same capacity in G' as in G with only exception of 'e'(if and only if e crosses the cut). In that case 'e' can provide a unit path across (A,B) and the flow can increase

atmost by 1, else the flow remains the same. This guarantees the correctness of the algorithm.

Running time: We only need to find an augmenting path in the residual graph, which is $O(m+n)$, where m is the number of edges and n is the number of nodes in the network. This is significantly smaller than restarting the maximum flow evaluation from scratch.

Case-2: Decrease Capacity of an edge by 1
Algorithm: As input, we are given the maximum feasible flow f in G, v(f) the value of f, an edge 'e' whose capacity is to be decreased by 1. If the current capacity of 'e' is strictly greater than its flow value, return the current maxflow. Else,in the residual graph, we decrease the capacity of 'e' by 1 and increase the capacity of the backedge of 'e' by 1. Then attempt to find an augmenting path from sink(t) to the source(s). If an augmenting path is found, return 1 unit of flow from sink to source to balance the reduction in capacity. Next, run the augmenting path subroutine again from source to sink, to find a new augmentng path and recompute the flow. This new flow value is the maximum flow with the the capacity of 'e' reduced.

Correctness: If the edge 'e' had not reached maximum capacity, that is, its capacity value was still greater than its flow value when the termination happended, then by decreasing its capacity it might atmost reach its full capacity in which case the current flow still remains valid. So, in that case we can say the maxflow in the modified capacity is still the same v(f). However, if the flow value of 'e' had reached its maximum capacity, then if we reduce its capacity, it means that we have transferred one additional unit from the source to the sink with respect to the new graph of modified capacities. Hence, somehow we need to take out this 1 unit of flow from the sink to the source to balance out and then recompute. Let e=(u,v) and source(s) and sink(t). Note that, since the edge 'e' had reached its capacity(assuming positive, else there would be no meaning of reducing it), that flow must have travelled through from v to t through some path, and hence there might be backedges that can provide a path from t to v that can flow atleast 1 unit. Similarly, since there was a non zero flow across 'e', that flow must have reached 'e' through some path from s to u. Then, there will be some backedges connecting from u to s that can atleast carry one unit of flow. If we can find such augmenting paths from t to v and from u to s, then we can tranfer this one unit of flow on 'e' back using the path t-v-u-s. The unit flow we aim to return can be modeled as an additional unit on the backedge of e, that (v,u). Finding the augmenting path from t to s, exactly serves this purpose. Once , we do this and return the flow back, we have established the network to a situation where the capacity of 'e' is reduced by 1, and the flow would have been v(f)-1. Now we try finding a forward augmenting path from s to t, and if found, that will increase the flow by atmost 1, or will remain the same(since flow never decreases over iterations and cannot exceed v(f), as v(f) was the max flow of a graph with highre capacity). Hence, the max-flow of this graph will be either v(f) or v(f)-1. This, guarantees the correctness of our algorithm.

(c) We use the following proposition to complete the final proof:
Proposition Let D be a directed graph such that every node v in D has indegree equal to its outdegree. Then there exists cycles $C_1, C_2, \ldots, C_k$ so that every edge appears in exactly one of the cycles.
Proof of the proposition: First choose a maximal list of the cycles, $C_1, C_2, \ldots, C_k$ so that every edge appears in atmost one. Suppose(**for contradiction**) that there is an edge not included in any cycle C_i and let H be the component $D - \cup_{i=1}^{k} E(C_i)$ which contains an edge. Now, every vertex $v \in V(H)$, where V(H) is the vertex set of H, satisfies indegree(v)=outdegree(v)$\neq 0$. Then for n nodes, the degree sum is more than n, which will result in a cycle $C \subseteq H$. But then C may be appended to the List of cycles. This contradiction completes the proof.
Proof Goal: For any two vertices s,t and an integer $k \geq 1$ in directed graph G of edge capacities 1, there exists k edge disjoint paths from s to t, iff there exists k edge disjoint paths from t to s.
Proof Since, there exists cycles that covers every edge as per the proposition, the cycle C_i covering the edges in the i'th edge disjoint path from s to t will require a path i' from t to s to complete the cycle. Since, the k paths are edge disjoint, they do not share any edge and hence they cannot be mutually part of the same cycle. Which means for the k edge disjoint paths from s to t , there exists atleast k cycles, and each of

these k edge disjoint paths from s to t, covers their corresponding cycles from s to t. Then to complete the cycle, each of them will have a remaining path from t to s, and these paths will also be edge disjoint since the cycles cannot intersect else same edge will get covered in two cycles. Hence, in the reverse path from t to s, there also must exist k-edge disjoint paths if there needs to exists k edge disjoint paths from s to t.(Proved).

3: More Reductions to Flows . . .

(a) Algorithm: Consider the n departments set be labeled as $D = D_1, D_2, \ldots, D_n$ and the set of faculties be labeled as $F = F_1, F_2, \ldots, F_m$. Also, suppose that there are 'r' ranks of professors. We can set up the problem as a maximum matching problem as shown in the figure-1:

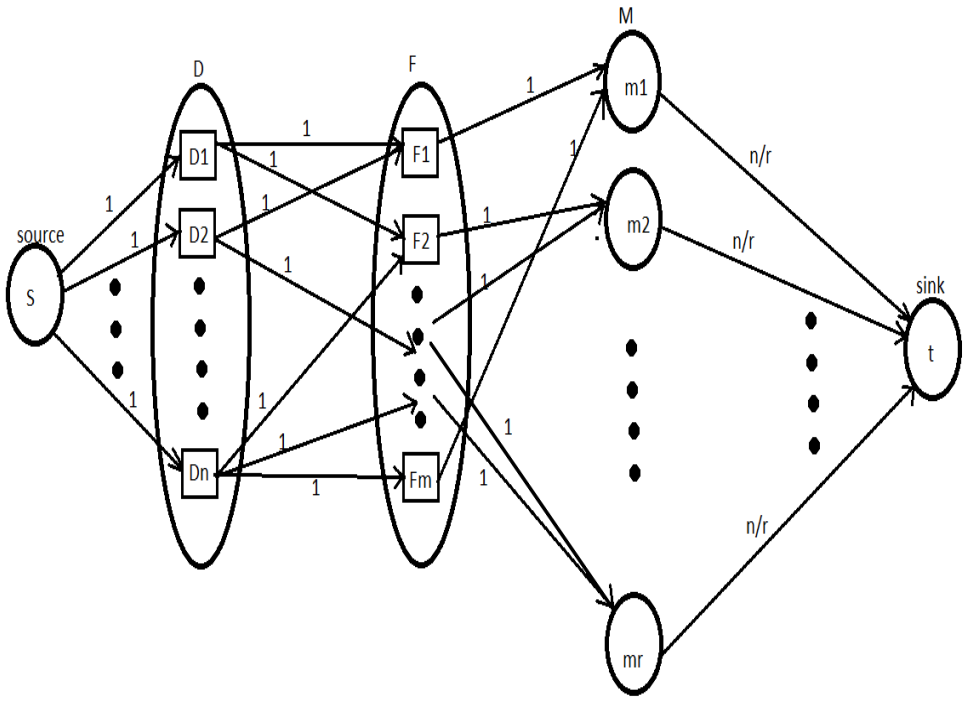

Figure 1: Max Matching for committee formation

We set up a graph from set D to set F, and every edge from $d_i \in D$ to $f_j \in F$ indicates that faculty F_j has appointment with department D_i. The weights of these edges are set to 1. The sets of $M = M_1, M_2, \ldots, M_r$ indicates the set of ranks. Thus an edge from F_i to M_j indicates that the faculty F_i belongs to rank M_j.

The outflow edges from the rank nodes to the sink(t) are given weights of $\frac{n}{r}$, such that every rank has equal representation to the final committee. The inflow edges from to the D_i from the source is set to 1, such that the max outflow is n and every department is allowed to pick one faculty. Now we run ford-fulkerson algorithm and if the maximum flow of this network is equal to the max possible outflow from the source,that is n, , then we have a maximum matching solution. Then, we say that a committee can be formed with equal representation from all ranks of faculties, and one faculty for each different, and **return** the set of faculties for whom the flow value to their corresponding rank, that is, $F_i -> mj$ is 1. If the max flow is not n, the return that such a committee cannot be formed.

Correctness: The only way we can get a max flow of n, is if the flow from each rank is equal to $\frac{n}{r}$, since the capacities of these edges is limited to n/r . Each rank can contribute n/r if and only if, it gets contributions from (n/r) distinct faculties since the edge capacities between faculties to ranks is 1 and no faculty can have edge to multiple ranks. This leads to the fact that n distinct faculties are contributing to the entire rank set. Since, the edge weights from departments to faculties is also 1, the maximum matching solution ensures that one department gets flow 1 with exactly 1 faculty and no two departments gets match with the same faculty. Since, we have n faculties with flow 1 to the rank set, hence each of the n department must have an edge 1 faculty to conserve the flow. Thus, overall we ensure that each department has 1 representative faculty with no overlap across departments and that each rank of faculties have equal distribution in the final committee, if we get the max matching solution. This ensures correctness of our algorithm.

Running time: To set-up the graph, we have to setup at maximum E=(n + mn + m + n) edges, hence this setup will be bounded in O(mn). The outflow from the source is n, so the Ford-Fulkersion is bounded by $O(n \times E) = O(n(2n + mn + n)) = O(mn^2)$, which is poly-time.

(b) Algorithm: Given an $n \times n$ checkerboard with some squares deleted. In a checkerboard, the neighbours of a black box are 4 white boxes and the neighbours of a white box are 4 black boxes. So, never will a white box have a white box as a neighbour and a black box will never have a black box as a neighbour. This directly leads us to a bipartite matching scenario. Note that in our case, we might not have the 4 neighbours since some squares will be deleted, but that doesn't affects the bipartiteness of the problem. So, we consider a set of white squares/boxes(W) and a set of black squares/boxes(B) and a source node S and a sink node t. An edge from $w_i \in W$ to an edge $b_i \in B$ indicates that in the given checkerboard they are valid neighbours and has the possibility of a domino being placed over them, and these valid edges are weighted as 1, since each square can atmost be part of a single domino. Each source node to w_i edge is weighted as 1 so that the w_i can have an incoming flow of 1 so that it can have an outgoing flow of 1 towards one black box. Similarly , each b_i has an outgoing flow of 1 towards the sink. The setup is shown in Figure-2

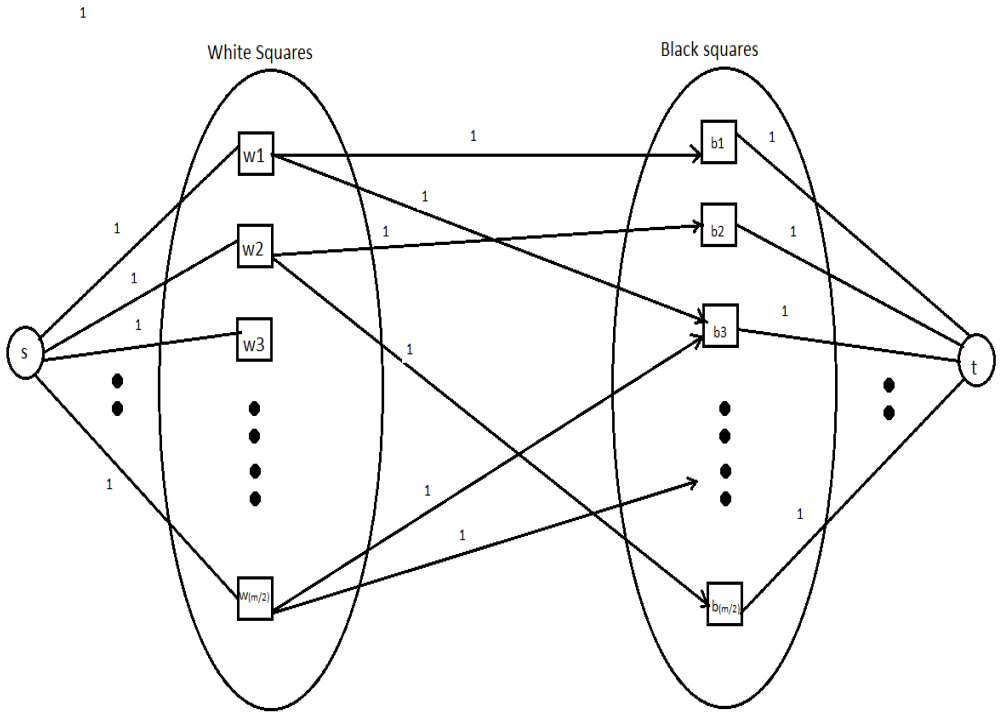

Figure 2: Checkerboard bipartite setup

If the number of available squares is odd, then we return that the tiling is not possible, since a domino is placed on two squares without an overlap, it requires an even number of squares. If the number of squares are even, two have the solution to this problem, the number of black and white squares must also be the same and equal to $m/2$, and if not we return tiling is not possible. When the cardinality of both white and black squares is even and m/2, we run the Ford-Fulkerson algorithm on this setup. If the maximum flow is m/2 , we say the domino tiling is possible and return the solution as set of edges $\{w_i, b_j\}$, which forms the exact combinations of the black and white squares that should be tiled together.

Correctness: On this setup if we get the maximum flow, then our claim is that we have got a solution for the tiling of the dominos. Since the out flows from the black boxes to the sink are weighted 1, a max flow of m/2 means that each b_i contributes 1 unit. For each b_i to contribute 1 unit, they must have received exact 1 unit flow from exact one of the w_j . Since, the number of white and black boxes are equal, this means that each white box w_j has been paired to a distinct b_i by contributing a flow of one. These distinct pairs of w_j, b_i are the positions where we place the dominos. This guarantees the correctness of the solution.

Running time: The setup of the graph can be done in $O(m^2)$ time . The outflow from the source is m/2. Hence, the runtime bound will be $(m^2 \times \frac{m}{2}) = O(m^3)$, which is polynomial.

(c) Algorithm: Given a graph G=(V,E) where V are the set of vertices and E are the set of edges, and V

$= \{v_1, v_2, \ldots, v_n\}$. If $|V| \geq |E|$, then we do not have a cycle cover, since a cycle cover requires $|E| \geq |V|$. W break each vertex, v_j as two vertices $v_{j_{in}}$ and $v_{j_{out}}$, where the former forms the set of vertices that receive incoming edges and the latter form the set of edges that has outgoing edges. We setup the bipartite sets of these vertices such that any edge ,e, going from $v_{j_{out}}$ to $v_{k_{in}}$ indicates a directed edge from node v_j to v_k in the original graph. We set up the source S and connect each $v_{j_{out}}$ to the source with edge weight 1, and connect each $v_{j_{in}}$ to the sink node with edge weight 1. The valid edges between $v_{j_{out}}$ to $v_{k_{in}}$ also has an edge weight of 1. Then we compute the max flow on this graph by running the Ford-Fulkerson algorithm. If the max flow is equal to n, then we have a cycle cobver for this graph and return the solution set comprising of the pairs of edges ($v_{j_{out}}$,$v_{k_{in}}$) that have a flow value of 1.

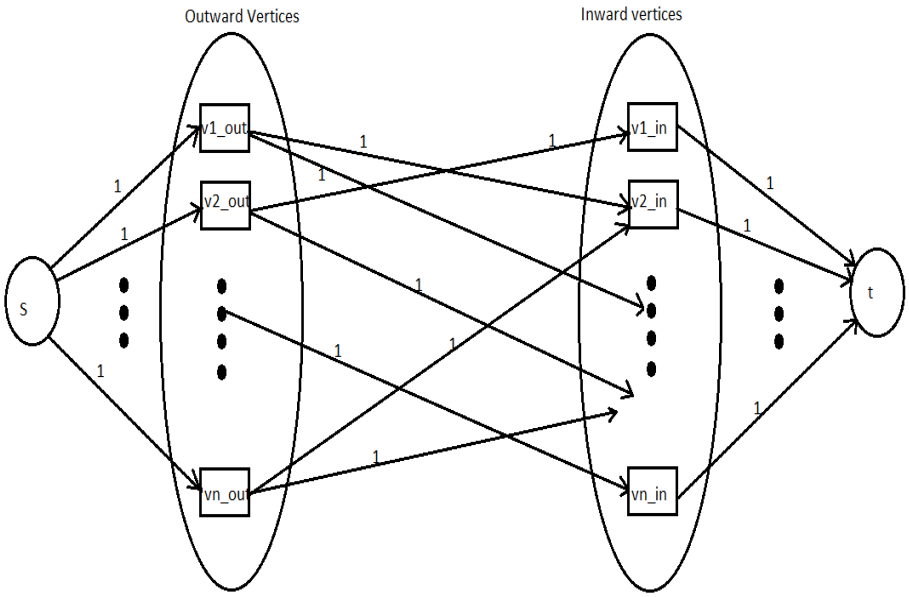

Figure 3: Cycle Cover

Correctness: We claim that if the maximum flow is equal to n , then we have found a cycle cover. If max flow is n, then each of the $v_{k_{in}}$ has contributed 1 unit to that, which means each $v_{k_{in}}$ has received exactly 1 unit from exactly one $v_{j_{out}}$. Thus, when you have maximum matching, it indicates that each node, is having one incoming edge and one outgoing edge. This is only possible if these edges form cycles and two cycles cannot have the same node, since each node is visited exactly once and there are n edges. Suppose of these n nodes and n edges, m of them form a cycle, using m edges. Then the remaining (n-m) nodes will be completely disjoint from these m nodes since the m nodes have already been visited. Since the (n-m) nodes also has (n-m) edges available, they can again form another cycle or many disjoint smaller cycles. Hence, the solution will be a cycle cover. This guarantees the correctness of the solution.

Running time: The set-up of the graph requires (m + 2n) edges where m are the edges of the original graph and n are the nodes of the original graph. The runtime of FordFulkerson is bound by $O((m+2n)n)$, which is polynomial in m and n.

(d) Algorithm: Given there are equal number n of actresses,$[A_i, A_2, \ldots, A_n]$ and actors,$[B_1, B_2, \ldots, B_n]$.

From the given database who compose a bipartite graph of the pairing of the actresses and actors, where an edge from A_i to B_j indicates that actress A_i has appeared in a movie with actor B_j. We define two additional nodes source,s, and the sink,t, such that there is an edge from source to each actress of weight 1 and there is an edge from each actor to the sink of weight 1. The edges between the actresses and actors are also 1. With this setup, if we run ford-fulkerson and get a maximum flow equal to n, that means we have received a perfect matching because, each edge terminating to the sink, contributes 1 unit of flow which further means that each actor B_j received 1 unit of flow from exactly one A_i. Under these scenario, we see that each actress has a 1 unit contribution to exactly one actor, and since the actor flows exactly one unit, it must have received from exactly one actress. Thus there is a perfect matching. If Bob has this perfect matching solution, then for every actress Alice names uniquely, say A_i, Bob will have an unique answer,B_j, and ultimately after n trials, since Alice cannot repeat names, she will exhaust the list of the actresses and fail to produce a new actress.Hence **there exists a wining strategy for Bob**. (Proved)

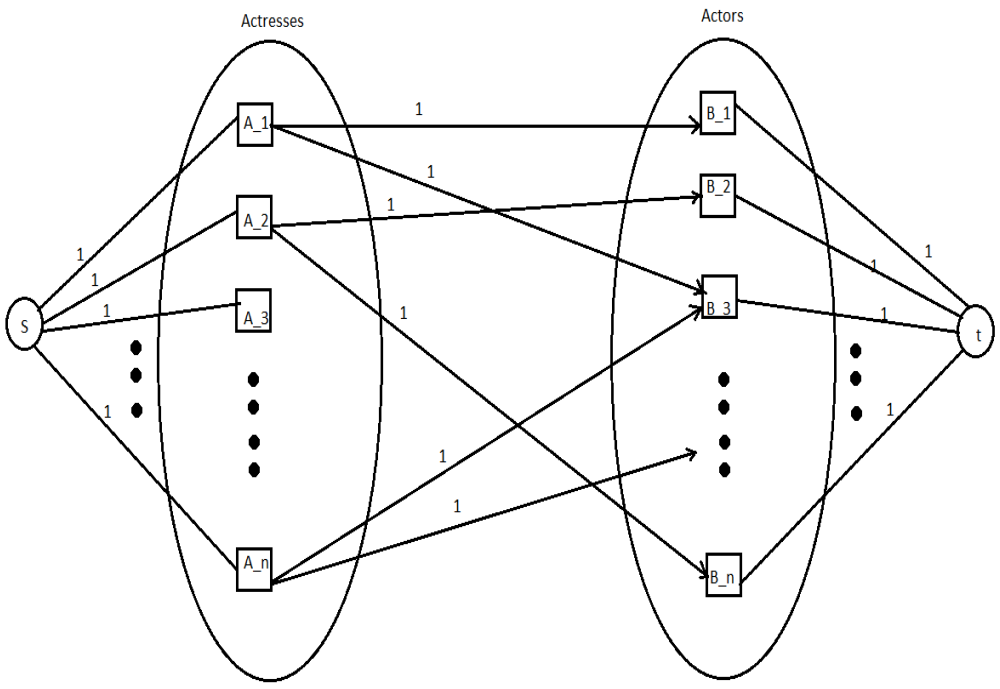

Figure 4: Actors and Actresses

(d.bonus) If the max flow is less than n, that means, not all actors could be paired with an actress edge

of flow 1. This leads to the fact, that there is atleast one actress, A_i, whose flow could not be transferred to one of the actors which means there does not exist a perfect matching. This can happen in two cases, either the actors she was paired with had edges already saturated, which means there exists atleast another actor who doesn't have any pairing or she doesn't have a pair. In the first case, suppose A_i had paired with some B_j's, who had other pairs with A_k's and the flow happended from these other A_k's to the B_j's leaving A_i out. Then, if Alice names these A_k's first, Bob will be able to answer with the corresponding B_j's, and then she can put up A_i, whose pairings have already been exhausted by Bob. So Bob looses and Alice gets a wining strategy. The other case is the A_i doesn't have a pairing, in which case Alice starts with A_i and Bob looses in the first round itself.

4: Unexpected Reductions to Flow/Matchings ...

(a)

(b) Algorithm: Given n rectangular tiles where the dimension of the i'th tile $a_i \times b_i$, and a tile j can be placed on top of tile i if either $(a_j \leq a_i and b_j \leq b_i) or (a_j \leq b_i and b_j \leq a_i)$. The motivation is to stack the tiles on top of each other such that the remaining area is maximized. So, we begin with considering that the all the tiles are laid out on the floor and we have the least area saving under this condition such that without loss of generality we can consider the situation here to be 0 area saving. We will strive to pick up a tile one by one and place on top of the other and in this way maximize the remaining space.
We create a bipartite graph of set A and set B, such that set A contains all n tiles $[t_1, t_2, \ldots, t_n]$ and set B also contains another version of these n tiles $[k_1, k_2, \ldots, k_n]$, such that, the an edge from t_i to k_j, indicates that tile k_j can be placed on top of tile t_i by satisfying the dimensionality constraints. The edges between t_i and k_j, denotes how much area we were able to save by placing k_j on top of t_i, which will be equal to the area of k_j, since we have removed k_j from the floor thus saving the space occupied by k_j. An obvious constraint on the graph is that no edge should exists between t_i to k_i, although the constraints were satisfied, since a tile cannot be placed on itself. Thus we have a graph for **weighted bipartite graph**. Given that we have a solver,S, for weighted maximum bipartite matching, we send this weighted bipartite graph with a call to this solver. The solution will be a set of pairs $\{t_i, k_j\}$. Of the given set of solution, search for tiles that only appear on the left-side of the 2-tuple. These are tiles that cannot be placed on top of any other tiles and needs to be kept on the floor. Suppose (t_i, k_j) is such a set. Then search for a tuple with t_j on the left, such as, (t_j, k_m)). Then we place tile m on tile j which in turn is placed on tile i. One stack completes the second element of the tuple does not belongs to any other tuple's first element. Thenagain look for the next base tiles and create the next stack. Unique tuples whose both first and second elements do not occur in any other tuple will results in stacks of length 1. So for every tuples from the solution set, keep creating such stacks until the solution list is exhausted. The final orientation of the stacks on the fllor is the best orientation that maximizes the remaining area.

Correctness: The motivation behind stacking is to minimize the amount of total floor space, which can be looked as a maximization problem from the point of view of the remaining fllor space. Thus, we always look to maximize the remaining floor space, and this is done by enumerating the edge weightts between tiles that can be palced on top of the other as the amount of area it saves by by performing this stacking. Since, the maximum weighted bipartite matching strives to find the best matching for maximizing the sum of the weights in the solution, we are bguaranteed to get the best solution for the maximum possible remaining area, or in turn minimum amount of total floor space used bu the tiles.

Running time: The tiles correspond to the nodes. Since we have a duplicate set as the other set , the total number of nodes in the graph is n^2 . The maximum number of edges will be $O(n^2)$, and hence the time taken taken to construct the graph is polynomial time. Furthermore, assuming that the solver for the

maximum weighted bipartite matching is polynomial time, we can further build up the stacks in polynomial time using polynomial time searches over the solution set of tuples comprising pairs of tiles. Hence, the algorithm is polynomial time.