Pre-tutorial questions

Do you know the basic concepts of this week's lecture content? These questions are only to test yourself. They will not be explicitly discussed in the tutorial, and no solutions will be given to them.

1. Bipartite matching.
   (a) What is a bipartite matching?
   (b) How can one use flow networks to find a maximum bipartite matching?
   (c) What does the marriage theorem state?
2. Edge disjoint paths
   (a) What does it mean that two paths in a graph are edge disjoint?
   (b) How can one use flow networks to count the maximum number of edge disjoint paths?
3. Extensions of flow networks
   (a) What is flow networks with node supplies and demands?
   (b) What is a circulation in a network with supplies and demands?
   (c) How can one use flow networks to model supply and demand?

Tutorial

Problem 1
Suppose you're in charge of managing a fleet of airplanes and you'd like to create a flight schedule for them. Here's a very simple model for this. Your market research has identified a set of m particular flight segments that that would be very lucrative if you could serve them; flight segment j is specified by four parameters: its origin airport, its destination airport, its departure time, and its arrival time. An example of six flights segments you'd like to serve with your planes over the course of a single day may include:

1. Brisbane (depart 6 am) - Sydney (arrive 7 am)
2. Sydney (depart 6:30 am) - Melbourne (arrive 8 am)
3. Sydney (depart 8 am) - Cairns (arrive 11 am)
4. Melbourne (depart 12 am) - Adelaide (arrive 2 pm)
5. Townsville (depart 2:15 pm) - Brisbane (arrive 4 pm)
6. Adelaide (depart 4:00 pm) - Perth (arrive 7 pm)

It is possible to use a single plane for a flight segment i, and then later for flight segment j, provided that:

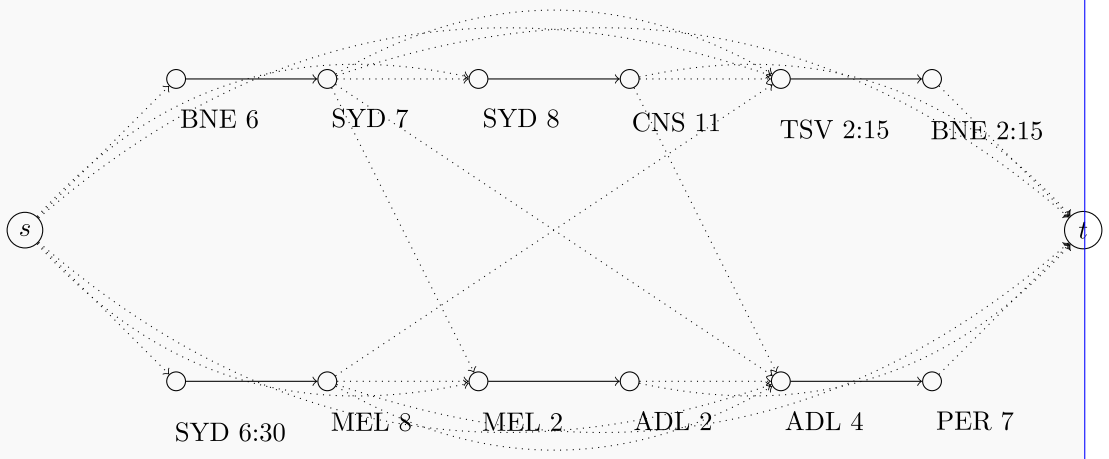
1. the destination of i is the same as the origin of j and there's enough time to perform maintenance on the plane in between; or

2. you can add a flight segment in between that gets the plane from the destination of i to the origin of j with adequate time in between.

For the example above you can group flights 1, 3, and 5 together by allocating an hour maintaince between each flight and adding an additional 1 hour flight between Cairns and Townsville for example.

Your goal based on the two conditions stated above is to determine if it is possible to service all m flights using at most k planes. In order to do this you will need to reuse planes as efficiently as possible.

Solution: The solution is based on the following idea. Units of flow will correspond to airplanes. We will have an edge for each flight, and upper and lower capacity bounds of 1 on these edges to require that exactly one unit of flow crosses this edge. In other words, each flight must be served by one of the places. If (u_i, v_i) is the edge representing flight i, and (u_j, v_j) is the edge representing flight j, and flight j is reachable from flight i, then we will have an edge from v_i to u_i with capacity 1.
For the example above we get the following $s-t$ capacitated network where it is possible to achieve a flow of 2. Although we use max flow this problem is a minimisation problem in the sense that the aim is to find the minimum max flow for the following graph where all the edges that are solid are 1.



Problem 2
Given an **undirected** graph $G = (V, E)$ and edge capacities $c : E \to \mathbb{R}_+$ and two vertices s and t, design an algorithm for computing a minimum capacity s-t cut.

Solution: Note that the graph is undirected, so we cannot compute a max-flow using the Ford-Fulkerson algorithm, which expects a directed graph as input.
Construct a new directed graph $G' = (V, E')$ with capacities c' where for each edge $(u, v) \in E$ we create two antiparallel edges (u, v) and (v, u) and set $c'(u, v) = c'(v, u) = c(u, v)$. It's easy to argue that for any cut (A, B) we have $c(A, B) = c'(A, B)$. Thus the min s-t cut with respect to G' and c' is also a minimum with respect to G and c.

Problem 3
We define a new version of the game Capture the Flag. The game is played in a maze that consists of a large number of rooms that are connected by doors. Each door connects two rooms. Each player starts in a given room in the maze (different players may start in different rooms). From each room a player can decide to move to any adjacent room, going through a door, as long as the door is open. All doors are open in the

beginning of the game, however, whenever a player crosses a door, the door will be closed and locked. That means, no other player will be able to go through that door, if they happen to find themselves in the same room at some point. So, when a player arrives in a new room, they can choose to go through any open door, but that door will be closed, and not available to anyone else from that point on. Some rooms contain a flag, and all flags are the same. Each player is trying to find a flag (any flag will do). A player can only pick up one flag. All players belong to the same team therefore they want to collect all flags if possible.

Here is the problem you need to solve. You are given the complete description of the maze, which consists of a set of rooms R, and a complete description of the set of doors D (which are pairs of rooms that are connected by a door). You are also given a set $S \subset R$, the initial k locations (rooms) of your team of k players. And you are given the set $F \subset R$, which are the k rooms that contain a flag. Your task is to check whether it is possible to find paths that your k players can follow in order to find all k flags, and no two paths cross the same door. Note that a given maze may not allow this, since doors can only be used once.

Given R,D, S and F, describe how to decide whether such a maze allows the collection of all flags or not.

Solution: A solution is possible using max flow where you can create a node for each room. Each edge between the rooms is defined by the set of doors D. Create two additional nodes s and t. Add an edge between s and each node in S and an edge between t and all nodes in F. All edges will have a weight of 1. This is to ensure that once you walk through a door another player may not follow you.
To know if their exists a winning strategy run FordFulkerson. If the max flow is k then you can collect all the flags otherwise you cannot.

Problem 4
Let G be a flow network such that, for every edge e in the network, $c(e)$ is an even integer.

1. Argue that the maximum value of a flow is an even integer.
2. Show that there is a maximum flow f such that, for every edge e, the flow over e is an even integer.

Solution: Do induction over the number of iterations made by the For-Fulkerson algorithm. After zero iterations the claims are true. Assume it's true before the ith iteration start. Note that since the flow is even and the capacities are even, it must hold that all the residual capacities are even. In the ith iteration the algorithm finds an augmenting $s-t$ path in the residual graph, if one exists. The value of the flow along this path is equla to the capacity of the capacity of the "bottleneck" edge. Since the capacity must be even the flow along this path must be even. This argument shows both the claims.

Problem 5
A d-regular graph is one where every vertex has degree d. Prove that every d-regular bipartite graph has a perfect matching.

Solution: Suppose for the sake of contradiction that there is no perfect matching. We know from the lecture (Hall's Theorem), that there is a set S on one side of the bipartition such that $|N(S)| < |S|$. Notice that $\sum_{u \in S} \deg(u) \leq \sum_{v \in N(S)} \deg(v)$ since all edge out of S go into $N(S)$. On the other hand, the degree of every vertex is k, so we get that $k|S| \leq k|N(S)|$, which contradicts our assumption that the graph does not have a perfect matching.
Note that this is not true for general graphs: A cycle on 3 vertices is 2-regular but doesn't admit a perfect matching.

Problem 6
In class we argued that Ford-Fulkerson runs for at most C iterations. If C is too large, this may take too

long. Luckily, a variant of the algorithm can be shown to run for at most $O(m \log C)$ iterations. The idea is very simple: Select an augmenting path maximizing the minimum residual capacity along the path. Show how to modify Dijkstra's algorithm to find such a path in the residual graph G^f.

Solution: (Sketch.) Recall that Dijkstra's algorithm computes a tentative distance value from s to every vertex in V; let us denote this tentative distance $d[u]$. Initially $d[s] = 0$ and $d[u] = +\infty$ for all other vertices u. The algorithm builds a priority queue for V so that the priority of $u \in V$ is $d[u]$. In each iteration it extracts the vertex u with minimum priority and updates the priority of each neighbor v of u by setting $d[v] = \min\{d[v], d[u] + \ell(u, v)\}$. The tricky part of the analysis is to show that when we remove a vertex u the shortest path distance from s to u equals $d[u]$.
Let us denote the residual capacity of an edge e with $\tilde{c}(e)$. In our modified version we again keep a tentative value $T[u]$ for each vertex. Initially $T[s] = \infty$ and $T[u] = 0$ for all vertices $u \in V - s$. We build a priority queue for V so that the priority of $u \in V$ is $T[u]$. In each iteration we extract the vertex u with maximum priority and update the priority of each neighbor v of u by setting $T[v] = \max\{T[v], \min\{T[u], \tilde{c}(u, v)\}\}$.
The trick of the analysis is to show that when we remove u from the priority queue the best bottleneck residual capacity to go from s to u equals $T[u]$. The argument is very similar to the one used for Dijkstra's algorithm, so I'll leave it up to you to fill in the details.