Problem 1

Problem 1 is a modification of the perfect matching Gale Shapley algorithm (GS algorithm). I referred the GS algorithm from our textbook page 6-7 as a reference to write the code. In this question, players should not abandon their team for another team, meaning that players get to choose the coaches (players act as men and coaches act as women – players have the preference to choose)

Algorithm:

Initially all n players and m coaches are free (|n| = |m| - total number of players = total number of coaches)
While there is a player n who is free and hasn't proposed to every coach:
    Choose such a player n
    Let w be the highest-ranked coach in n's preference list to whom n has not yet proposed
    If w has a preference of 1 or 2 of the player n:
        If w is free:
            (n, w) is a match
        else w is currently engaged to another n':
            if w prefers n' to n:
                n remains free
            else:
                (n, w) is a match
                n' is free
    else:
        Nothing Happens (proposal automatically fails)
If every n has matching m:
    Return the set S of engaged matches
Else:
    Return "Impossible match"

Proof:

The algorithm is a modification of the GS algorithm. Instead of the original GS algorithm where every woman (in this case the coach) has preference of all the men (players in this case) from 1-n, coaches only have the preference of 1 or 2. In other words, even though there are 100 players and 100 coaches, every coach only prefers 2 of the players instead of having preference from 1 to 100. Therefore, when players choose a team by proposing to a coach, the coach should have the player as 1 or 2 as the coach's preference list in order for coach to accept the proposal. So, by adding the statement of – if w has a preference of 1 or 2 of the player n – whether the coach prefers the player or not is necessary line that needs to be added in the GS algorithm. Finally, the last two if statements are checking whether all n and m have a match, which is what the question asks.

Runtime:

It is proven that the GS algorithm has the runtime of O (n^2), which is a polynomial algorithm. The last if statement added is simply a constant time c, which can be ignored.

Problem 2

(a)

Algorithm:

Create n nodes

Connect the nodes in the way that a node points another node with a weight of the fare (if there is a fare, the weight equals the fare. If there is not fare from one node to another, the weight equals infinity)

Compute the distance d[v] of every vertex from the node 1 (Boston) by running the Dijkstra's algorithm and fill out the adjacency list R[v] for each vertex

By referring to the adjacency list, return the R[n] for vertex n (the node for DC)

Proof:

The question asks to find the shortest fare from Boston to DC. Each fare is given from one node to all the other nodes (either infinity or a finite number). Finding the shortest path from a directed graph with nonnegative edges (all fares are assumed to be positive – since there cannot be negative amount of money to spend) can be done by Dijkstra's algorithm. With the algorithm, we can fill out the adjacency list and simply return the smallest distance from node 1 (Boston) to node n (DC).

Runtime:

The runtime of this function is O((n+m)*log(n)) since we are using Dijkstra's algorithm. The runtime of Dijkstra's algorithm was explained in class.

(b)

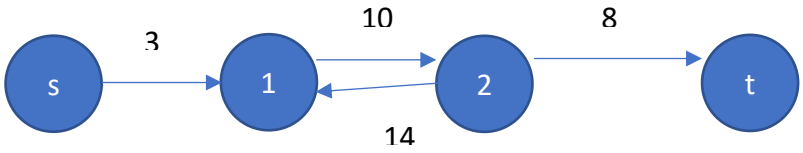
If the fare from node 2 to 3 is 1 and 3 to 2 is, therefore, automatically 1, there can be a problem with the algorithm above. If you travel from node 2 to node 3 with the distance of 1, the algorithm might recognize node 3 to node 2 as the shortest distance of 1 and might return back to node 2. If we return back to node 2, we have basically ran a cycle and only added the fare amount of 2, which is very insufficient.

We can handle this case by implementing Dijkstra's algorithm above in part (a) by indicating the previous node and if there is a direction to the previous node, simply ignore it. If the node 2 to 3 has fare of 1 and node 3 to 2 has fare of 1, we can travel from node 2 to node with the fare of 1. Then, at node 3, you have the knowledge that it traveled from node 2, and will automatically ignore the path from node 3 to node 2.

Problem 3

(a)

Below is a flow network that has directed cycle of edges with positive flow.



The directed graph has a cycle between node 1 and node 2. If the flow of 3 is sent from node s to node 1, node 1 will pass the flow of 3 to node 2. Node 2 has the option to either return back to node 1 or send to the sink node, node t. If it sends the flow of 3 back to node 1, and such act of node 1 and node 2 continuing to send the flow of 3 to each other is a directed cycle of edges with positive flow.

(b)

Every flow network has a maximum flow that could be found in polynomial time. The Ford-Fulkerson algorithm returns the maximum number of flow that could be sent from the source node (node s) to sink node (node t) in runtime O(m * f), where m is the number of nodes and f is the value of maximal flow. m * f is a polynomial algorithm. The Ford-Fulkerson algorithm of how to find the maximum flow was explained and proved in class (while there is augmenting path, continue to send flow through that path). No matter whether there is cycle or not, the flow is sent from the source node to the sink node.

Problem 4

(a)

Algorithm: (This algorithm assumes that there is only and only one trailhead parking and only one summit)

Create n spots as n nodes

Connect the n spots with an edge with weight of 1 if there is a path

Run the network flow Ford-Fulkerson algorithm that returns the maximum number of flow

If the maximum number of flow is greater than or equal to 3, create 3 trails with different color along the flow to each summit

If the maximum number of flow is less than 3, return “Impossible”

Proof:

By connecting the spots with weight of 1 if there is path, we have a directed graph from the trailhead parking and summit through various spots. The Ford-Fulkerson algorithm allows us to determine the answer: since each edge or path has only the weight of 1, we cannot go through the same path twice since if an edge was already traveled, the capacity of that edge will be 1, which is equal to the weight. Since we cannot travel the same path twice in Ford-Fulkerson algorithm in the situation described in my algorithm, every time it tries to go from trailhead parking to summit, it will travel to through different path that has not been used to travel yet. If the algorithm returns 3 or value greater than 3, it means that there is at least 3 paths from parking to summit that do not overlap (in terms of edges, nodes can overlap), which then can be colored differently. If the value is less than 3, that means there does not exist 3 completely distinct paths from parking to summit, which is why we return impossible.

Runtime:

The runtime of Ford-Fulkerson algorithm is O(m*f), where f is the value of maximal flow and m is the number of edges. We do not know the exact number of edges, but the upper bound of edges is n*n (if we consider every node is connected to other node with an edge). Therefore, the runtime of this algorithm is O(n^2*f) where n is the number of spots and f is the value of maximal flow.

(b)

If this goal of finding three distinct paths from the parking to summit is impossible, it means that either the summit is unreachable or there is a set of one or two tracks that has to be reached from every path to reach the summit. This is because if there are less than three distinct paths exist, it means that there is either 0, 1, or 2 paths from parking to summit. 0 path means that climbing up to the summit itself is impossible. If there is 1 or 2 path, it means that the 1 or 2 paths are sharing a common path that must be traveled in order to reach the summit since the Ford-Fulkerson algorithm always tries to look for distinct edge not used to travel (due to inability to travel through the capacity that is 0). If this algorithm fails and only returns 1 or 2, it means that while traveling the second or third path, respectively, it must go through an edge that was already traveled, making it impossible to go through.

Problem 5

The probability of hitting each goose is 1/n.
When the hunter fires a shot, the probability of missing each goose is therefore 1-1/n.
Since there are n geese, the probability of missing every goose is (1-1/n)^n.
This means that the probability of hitting at least one geese is 1-(1-1/n)^n.
The expected number is calculated by n * p (according to what I learned in CS237).
Since there are n hunters which the probability of 1-(1-1/n)^n, the expected number is
E(n) = n * (1-(1-1/n)^n)
= n – n(1-1/n)^n

Problem 6

The problem asks to find the partition of sets A and B given an undirected graph so that the number of edges between the set A and B are maximized. The question simply asks us to find the maximum cut, which is similar to the minimum cut problem that we covered in class (completely opposite question).

Algorithm:

While there are more than 2 nodes in the directed graph:

For every node:

Find the number of edges between two nodes and store it in E[v]

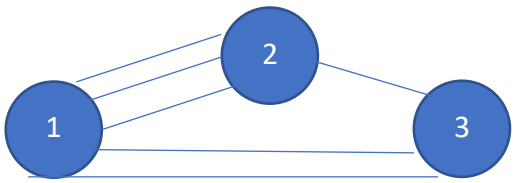
Find the lowest number inside E[v] and merge the corresponding two nodes (contract an edge(u,v)) as what we did in the slide 4_9_random.pdf by Professor Dora (when we were finding the minimum cut by using random processes)

if there are only 2 nodes left:

return the number of edges that are connected between the 2 nodes

Proof:

Instead of going to a random approach as we did in class, the question asks to find a greedy algorithm. To find the maximum cut, we must merge two nodes that has the minimum number of edges connected between them. For example, if node 1 is connected to node 2 in 3 ways, node 2 is connected to node 3 in 1 way and node 3 is connected to node 1 in 2 ways.



In this case, we are going to extract the edge that connects node 2 and node 3 to provide the maximum cut. If we do so, the cut between node 1 and node 2 becomes 5 (node 2 and node 3 merges). This is the maximum way because when we merge two nodes together, the edges that connect those two nodes disappear. This means that if we merge two nodes that have the least number of edges between them, the number of edges that are lost during the merging step until only two nodes are left will be minimized (and we are looking for the maximum number of edges in this question so the deletion of edges should be minimized). If there are only two nodes remaining, the final cut defines the maximum cut, which is what the algorithm should return.

Runtime:

The runtime of this algorithm is O(n * m) where n is the number of nodes and m is the number of edges. We have to repeat the step of merging two nodes until there is only two nodes left, requiring O(n). For every one step, we have to calculate the number of edges between two nodes to find the smallest number of edge between any two nodes, which is O(m). Multiplying these gives us O(n*m).

Number of Edges between A and B is at least half of the maximum possible:

We can show this by proving that the worst case of the algorithm returns at least half of the maximum cuts.

The worst case is when every node is connected with every other node with the same number of edges since as we merge two nodes, the number of edges lost will be maximized (since we are deleting the minimum number of edges every step, this is the worst case because if the edges were not evenly distributed between nodes, it would cut the minimum number of edges, which is always less than or equal the average number of edges).

This means that there exist m/(n*(n-1)/2) edges between any two nodes, where m is the total number of edges and n is the number of nodes. Every time the loop runs, we lose m/(n*(n-1)/2) number of edges, so we get the equation m – m/(n*(n-1)/2) * (number of loops that run). The loop runs until there is only 2 nodes left, so the number of loops that run is n-2. Therefore, the equation gives m - 2m*(n-2)/(n^2-n). Factoring out m, we get m * (1-2(n-2)/(n^2-n)), which has to be greater than 1/2m. If we cancel m and reorder the equation, we get n^2-n > 4n -8, giving n^2 – 5n + 8 > 0. Solving the quadratic gives that there is no x-intercept and the graph is heading up, which means that for any n, n^2-5n+8 is greater than 0, which explains the concept that m * (1-2(n-2)/(n^2-n)) is already greater than 1/2 * m, proving this concept.

Problem 7

Proving NP-C requires two steps:

1. Show the question is NP
2. Find another NP-C and reduce the question to it

1. Witness: The k non-overlapping sets serve as witness
   Verifier: Check whether the number of courses that every student takes is less than the k non-overlapping time slots. (Takes O(n) – going through every student and checking whether it is less than k takes O(n) time since the comparison is constant time and there are n students)
2. The scheduling of students' exam to k non-overlapping time slots can be reduced to K-colorability that we went through in class (2-colorability and 3-colorability). K-colorability is NP-C question where we are given an undirected graph and whether the nodes can be colored with k distinct colors so no adjacent nodes (nodes connected by an edge) have the same color, according to the lecture slide 4/23 by Professor Dora).

   We can reduce the scheduling problem into K-coloring problem by doing the following: the non-overlapping k slots are k distinct colors, the nodes are known to be courses, and the edges are known to be students who take or enroll the course.

   Such reduction can be made because of the following explanation: if a student takes 2 or more courses, it is connected to at least two nodes. There is a problem if those two or more courses overlap in the exam slots since a student cannot take two exams at the same time, simultaneously. If we change this to K-coloring question, since each student act as an edge, the nodes that are connected are courses that should not have same exam time periods because the same color would represent same exam period, leading at least one student to take two or more exams simultaneously. If K-coloring is possible, that means we can have students to take exams in K non-overlapping periods.

   Since K-coloring is known to be NP-C, we can find the solution of the graph and whether we can color the nodes with a color that is different with the adjacent nodes. Then, we can convert the answer of whether the K-coloring was possible and turn it into the scheduling problem. If K-coloring is possible, return yes to the scheduling problem. On the other hand, if K-coloring is impossible, return no to the scheduling problem.