Algorithms Assignment -8

Question 1:

Approach -1:
We should create a graph for the Jumanji jungle S.T. all the land marks in Jumanji jungle will form the nodes of the graph and the path between them form the edges with the distance as the cost of the edges.

In order to solve this question, we can use Floyd-Warshall's algorithm which uses dynamic approach to find the shortest path between all pairs of vertices by comparing all possible paths that exists between the given pair of vertices.

By using the map provided we can construct a distance matrix "distanceMatrix" which gives us the details of the distance between land marks in the Jumanji world if there exists a direct path between the two land marks(we can even use a matrix which gives details of the time taken to travel from one landmark to another) and an HashMap "deathTrapMap" which keeps track of the details of whether a land-mark is a death trap or not.

We can have another matrix which is set to the distance matrix that will be given as input(entries with 0 is set to infinity), Let's call that "costMatrix" , this matrix is updated with the shortest route between any pair of the landmarks by FloydWarshall's algorithm.

Another matrix to keep track of the total number of death traps along the path between two land marks in the graph. Let's call this "trapMatrix". This matrix is initialized to 2 if both u and v are traps, 1 if either is a trap and 0 if none are traps.

We update this to the (total traps along the path +1) each time go through a landmark which has death trap: (trapMatrix[i][j] = trapMatrix[i][k] + trapMatrix[k][j] +1).

In order to make floyd-warshall's algorithm work for this we need to make changes to the standard algorithm. i.e. we need to check for an additional condition i.e. if the total death traps along the path from i to j is lesser than or equal to the total number of lives smolder has. Let that be L. only if this condition is satisfied then we need to update the costMatrix with the shortest distance computed so far.

Let G be the graph which is formed as described above
Let N be the number of land marks in Jumaji jungle. All the matrices are of size N*N
Let E be the edges in the graph G.
Let V be all the nodes in G

Algorithm:

Given: "distMatrix " a matrix which has the distance between two land marks in the Jumanji Jungle and an "sourceIndx", from which we need to start the journey and "destinationIndx" the point at which jaguar eye needs to be restored. E is the edges of the graph G, V set of all nodes in G, deathTrapMap which has all the death trap nodes in hashmap

Returns: an integer value which represents the shortest distance to reach Jaguar's eye alive.

```
ShortestPathToJaguarEye ( distMatrix, L,sourceIndx, destinationIndx, E , V, deathTrapMap)
{
        Let costMatrix be a N* N matrix initialized to INFINITY
        Let next be a N * N the matrix which keeps track of the next node in the path

        // set cost matrix with the values of distance matrix initally if edge exists
        for each (u,v) in E
                if distMatrix[u][v] >0
                        costMatrix[u][v] = distMatrix[u][v]
                // set next node in the path to v for all edges (u,v)
                set next[u][v] to v
```

```
    // initialize trap matrix to 0 intially
    for each (u,v) in E
            trapMatrix[u][v] to trap counts i.e. 2 if both u and v are traps, 1 if either of it a trap and 0 if
            neither of them are traps or if no edge (u,v) exists


    // initialize distance matrix to 0 for all paths from node to itself
    for each v in V
       distMatrix[v][v]   = 0



    for k from 1 to |V|
            for i from 1 to |V|
                 for  j from 1 to |V|
                     //compute the total trap count between i and j through k
                     totalTrapCount = (trapMatrix[i][k] + trapMatrix [k][j] )

                     // if life count  L is greater than the totalTrapCount then we update the optimal
                     distance // value
                     if  ( (totalTrapCount < L)  && (costMatrix[i][j] >   costMatrix[i][k] +
                     costMatrix[k][j]) )
                     {
                             costMatrix[i][j] = costMatrix[i][k] + costMatrix[k][j]
                             trapMatrix [i][j] = totalTrapCount
                             next[i][j] = next[i][k]
                     }

    // we can get the path by backtracking the next and costmatrix gives the total distance by backtrack
    return costMatrix[][], next[][];

}
```

We can compute using this code

```
Start = u
While( Start ! = v)
{
        Start = next[u][v]
        //prints path
        print( u )

}
```

Approach-2 :

This is brute force approach where we can use bellman -ford algorithm to discover all possible paths in the graph from source node to the destination. Keep track of the count of the number of landmarks in the path which are death traps. The principles of relaxation for the bellman ford algorithm will be modified a little i.e Approximation to the correct distance is gradually replaced by more accurate values until we reach the optimal solution if and only if the count of death traps along the path is less than the total number of lives which ensures that the Sheldon and Smolder will reach the destination i.e. Jaguar’s eye alive.

We should create a graph for this S.T. all the land marks in Jumanji jungle will form the nodes of the graph and the path between them form the edges with the distance as the cost of the edges. Let each node be associated with the Boolean field "isDeathTrap". If this field is set to true then the node is a death trap.

Algorithm :

Let V be the nodes of the graph and E represent the edges of the graph G which is created as stated above.Each edge is associated with cost or weight w

```
 Given:  Edges ( E ), Vertices (V), source from which we need to start the journey,
        and L is the total life count
ShortestPathToJaguarEye(E, V, G, source,L)
{

        // Initialize distance to all the nodes to infinity initially
        for each v in V
                set dist[v]  to Infinity
                set predecessor[v] to undefined
                // keeps track of death trap count along the path traversed
                Set trap[v] to 0

        // Set distance of source to zero so that it gets picked first
        dist[source] = 0


        for i=1 to V.length-1
                // for each of the edge perform relaxation
                for each (u,v) in E with weight w
                        // If the current node is death trap increment the death trap count
                        // along the path
                        if(v.isDeathTrap)
                                        totalTrapCount = trap[u]+ 1
                        // else set it to the count of the no of traps that we have along the path to v
                        else
                                        totalTrapCount = trap[u];

                        if (  ( totalTrapCount <L) and   (dist[u] + w  < dist[v]))
                                dist[v] = dist[u] + w;
                                predecessor[v] = u;
                                trap[v]  =  totalTrapCount;



        // In case of negative cycles return not possible
        for each edge (u, v) with weight w in edges
                if dist[u] + w  < dist[v]
                        error " negative cycle exists"

        // return the distance array and predecessor  which can be backtracked to get the shortest path
        return distance[],  predecessor[]
}
```

Time complexity for this algorithm will O(|E|*|V|) (Bellman-ford algorithm complexity) Where E is the total no of edges in Graph G and V is the total number of vertices in G

Question -2 :

In order to solve this problem we can use max flow network. We need to identify all the cities to which students belong and get an idea of how many students belong to each city. All the cities and tables are represented as nodes. Include a source node and a sink node. Connect each city with source node using edges whose capacity is equal to the total number of students from each city attending the event and connect each table with sink node using edges whose capacity is equal to the total number of students each table can accommodate.

Each city node is connected with the table node with edges whose max capacity is equal to 3, as we don't want more than three students from a particular city to be seated on the same table.

Initially we need to check if the total number of seats available on all the table is equal to or greater than the total number of students attending the event else we need to return "not possible" as we can allot place for all the people being invited for the event.

On running max network flow algorithm on this we can get to know the maximum number of students that can be accommodated on the table such that not more than three students from same city are allotted the same table.

If the maximum flow value obtained by the algorithm is greater or equal to the total number of students attending the event from all the cities then the above mentioned arrangement is possible else not possible.

After running Ford Fulkerson's max flow algorithm, if arrangement is possible then we need to return the details of all the edges in graph G connecting city to table with flow count greater than 0 which gives all possible arrangement details.

Let S be the source and T be the sink or terminal of the network flow graph constructed.

Let a1, a2, a3, a4…ai be the count of students belonging to any city i and b1, b2, b3, b4… bj be the total number of students any table j can accommodate.

Let 'n' be the total number of cities (sum of all ai's for all i) and 'm' be the total number of seats available in the event hall(sum of bj's for j)

We can use Ford-Fulkerson algorithm to solve max-flow problem

Let Cn be the capacity of edge in general which includes all ai, bj and 3

Algorithm:

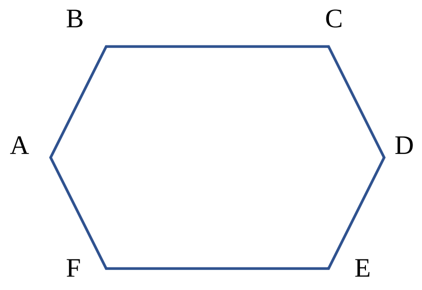
```
Input:    Graph G(V,E) which Is constructed as described above S. T. V = C + T + 2
          where C is total number of different cities to which students belong
          and T is total number of tables in the event hall and 2 for source and sink. Each edge has a total
          capacity of ai or bj or 3 as explained above.
          Total count of students(studentCount)

Output: 2D array which has city index u , table index v and number of students from city u that can be
          accommodated in table v, "not possible " if arrangement not possible

          // To record the maximum number of people that can be accommodated in the
          // hall such that no more than three people is allotted the same table.
          Set maxPeople to 0 initially.

          // Flow value is set to 0 intially for all edges e in E
          Let fn be set to 0

          While path p exists from S to T in G S.T. Cn(u,v) >0 for all edges belonging to path p

                    // Find the bottle neck in the path p
                    Let bottleneck = min( Cn(u,v) : (u, v) ∈ p)
                    For ean edge (u,v)∈ p
                              // forward edges
                              f(u,v) = f(u,v) + bottleneck
                              // Backward edges to return the flow
                              f(v,u) = f(v,u) – bottleneck


                    maxPeople = maxPeople + bottleneck;

          // If maximum people that can be accommodated is greater than the total number //of students
          attending the event then the arrangement is possible else not possible
          If(maxPeople >=  studentCount)

                    For all the edges (u, v)  in graph G  connecting **city to table**
                              If(f(u,v) >0)
                                        //Keep track of u, v and f(u, v) in an 2D array "result"
                                        result.add(u,v, f(u,v))

                              // result will have the city(u), table(v) and count of students from city u that can be
                              // accommodated in table v
                    return result
          else
                   return "not possible"
```

Question -3 :
In general, if a graph G is connected, undirected and not a tree, then the graph G should contain a cycle. For any graph G with v vertices, the largest cycle possible has v edges in it. i.e largest possible cycle will be connecting all n vertices with n edges.

Diameter of such a graph will be (n/2) if the number of edges in even else it will be (n/2)+1 for odd number of edges case. This is because the minimum no of edges to reach the last node from the first one will be (n-1) edges which is longest amongst all other vertex pairs for the graph with largest cycle. So any graph which is connected , undirected and not a tree will have a cycle whose length can at most be 2 * diam(G) + 1

This can also be proved used an example as shown below:

Let G(V,E) be a graph with 6 vertices and 6 edges as shown below. Adding any additional edges will create a cycle smaller than the one shown below.



The distance for each pair of vertices will be as shown below:

A – B   -> 1
A – C   -> 2
A – D -> 3
A – E   -> 2
A – F   -> 1
B – C -> 1

B – E -> 3
B – F -> 2
C – D -> 1
C – E -> 2
C – F -> 3
D – E -> 1
D – F -> 2
E – F -> 1

From the above value, we can notice that the $diam(G) = 3$, for graph with even number of edges, $diam(G) <= V / 2$ and for a graph with odd number of edges, $diam(G) <= (V / 2) + 1$
Where V is the total number of nodes or vertices in graph G.

When V is even => $diam(G) <= V / 2$ Equation 1
When V is odd => $diam(G) <= (V / 2) + 1$ Equation 2

From above equations:

$Diam(G) <= (V / 2)$

⇨ $V = 2 * Diam(G)$

Therefore there exists a cycle in every graph G of length at most $2 * diam(G) + 1$ as we can not a cycle whose length is greater than V in any graph G