Theoretical Assignment - 3

ESO207

Name: Aaryaman Aggarwal
Roll No.: 230020

October 30, 2025

1) Galactic Federation

a)

In the given question we can face the following cases:

- Planet 1 is connected to Planet n: In this case there is no need to construct bridges between any two planets
- There are only two connected components in the graph, in which case only 1 connection between the two connected components will be required. The cost in this case will always be 1 as if there are only two connected components, we can always find a pair (i, j) such that $|i - j| = 1$.
- There are three or more connected components in the graph and we can categorize them as follows:
  - Connected to 1: Let's call this A
  - Connected to n: Let's call this B
  - Connected to neither: Let's call this C which contains a list of all the planets that are not connected to either 1 or n. It is not necessary that these are connected to each either as well.

To solve the given question we take the following steps:

- We would begin by conducting a DFS to obtain the connected components of the graph which would allow us to construct the lists A, B and C of the connected planets from 1, n and neither respectively
- We would check if n is already connected to 1 and then simply return 0 and no new edges are required

- If after the DFS runs, C is empty i.e $|A| + |B| = n$ or simply put, every planet is connected to either 1 or n, then we return 1 as the cost is $|i - j|^2$ and there will always be a pair $i \in A$ and $j \in B$ such that $|i-j| = 1$. This is simply because since every planet belongs to either A or B, there will be at least 1 index in A such that one of it's neighbors will be in B. Suppose this wasn't true, then all the indexes between 1 and n would belong to A and that would lead to case(I).
- If after all DFS runs we have all three lists to be non-empty. We do the following for this:
  - Sort the numbers corresponding to the planets connected in each list.
  - In a simple two-scan between A and B we can find the indexes (i, j) such that the cost $(i - j)^2$ is minimized
  - Further, it is easy to see that if there is a k such that $i < k < j$ or $j < k < i$ then the new cost $(i-k)^2+(j-k)^2$ will be lesser than the original cost $(i-j)^2$. Hence we scan through C to find an ideal candidate which minimizes the cost

Note: To avoid sorting in $\mathcal{O}(n \log n)$, we can do it in the following method

- Create an array label[n] which stores the values as follows:
  - 1: If the element is connected to 1 and visited in DFS(1)
  - n: If the element is connected to n and visited in DFS(n)
  - -1: If the element is connected to neither and not visited in either.
- Once label[n] is ready, for any i based on label[i] it is appended to the lists A,B and C accordingly. This ensures that the lists A, B and C are sorted as each planet is added in an increasing manner

DFS(u) is modified such that each element v it visits stores u in label[v].

(b) Pseudocode

The algorithm given below guarantees the minimum possible cost because we find the 3 closest points that have no edges between them, by minimizing the cost over all lists A, B and C and connecting them to form a path between 1 and n.

(c) Time complexity analysis:

The time complexity is as follows:

- The DFS traversal is of time complexity $\mathcal{O}(n + m)$

Algorithm 1 Galatic Federation

```
1: function MINIMUM COST(n)
2:     Create array label[1, 2 ··· n] and each Label[i] = -1
3:     DFS(1)
4:     DFS(n)
5:     Create lists A, B, C
6:     x ← 0, y ← 0, z ← 0
7:     for i ← 1 to n do                                  ▷ Filling lists A, B, C
8:         if label[i] = 1 then
9:             A[x] ← i
10:            x ← x + 1
11:        end if
12:        if label[i] = n then
13:            B[y] ← i
14:            y ← y + 1
15:        else
16:            C[z] ← i
17:            z ← z + 1
18:        end if
19:    end for
20:    if A[x − 1] = B[y − 1] then
21:        return 0
22:        //This is the case where 1 is connected to n
23:    end if
24:    if k == 0 (//C is empty ⟹ k is never incremented) then
25:        return 1
26:    end if
27:
28:    i ← 0, j← 0
29:    cost ← (n − 1)^2
30:    p ← 0, q ← 0
31:    while (i < x) and (j < y) do
32:        if ((A[i] − B[j])^2 < cost) then
33:            p ← A[i]
34:            q ← B[j]
35:            cost ← (A[i] − B[j])^2
36:        end if
37:        if A[i] < B[j] then
38:            i ← i + 1
39:        else
40:            j ← j + 1
41:        end if
42:    end while
43:
44:    for i ← 0 to |C| − 1 do
45:        if (p − C[i])^2 + (q − C[i])^2 < cost then
46:            cost ← (p − C[i])^2 + (q − C[i])^2
47:        end if
48:    end for
49:    return cost
50: end function
```

- Creating the lists A, B, and C are a single sweep of the array label[n] and hence are of time complexity $\mathcal{O}(n)$
- The for loop in lines 25 through 36 is of order $\mathcal{O}(n)$ as each element in both arrays, A and B is visited once. Each array is of size less than n and hence the while loop is bounded by $\mathcal{O}(n)$.
- Similarly, in the subsequent loop, each element in list C is visited exactly once and the size of list C is less than n, hence this is also bounded by $\mathcal{O}(n)$.

Therefore the total time complexity by combining above elemnent is:

$$T(m, n) = \mathcal{O}(n + m)$$

Space Complexity:
There are fours lists that when combined are of size n. Hence the space complexity is bounded by $\mathcal{O}(n)$.

2) Postal Routes of Valoria

(a)

A route in the connection of towns which uses each edge only once and also starts and ends in different towns in a Eulerian trail. The necessary conditions for it to exist are:

- There is only one connected component in the graph. We want to ensure that all n-towns are connected
- Exactly two have an odd-degree. These vertices would be the endpoints.

These conditions are sufficient and if they fail we can automatically say that there will not be a valid grand route.
To produce the given route, which starts and ends in different towns, we first add a pseudo-edge between the start town and end town
We start from any vertex which has unused edges (unused degree > 0). We follow each edge randomly but continuously and mark each one as used and continue storing the path to *circuit* as we go on until we reach the ending vertex. If all the edges of some vertex, say u haven't been used after the above step, we repeat this process to form a new cycle which can be added to the earlier trail. We essentially merge it into the earlier trail . We can continue this process until every edge is used.
In the end we remove the pseudo-edge added between the start town and end-town and this is the Eulerian trail required.
This works because if a vertex has an even degree, we can leave it every time we enter the vertex while using a new-edge which is what the above algorithm does. If there are any cycles we remove it layer by layer.

(b). Eulerian Circuit

To ensure that the route starts and ends at the same town, every single vertex must have an even non-zero degree. This condition is sufficient to ensure the guarantee an Eulerian circuit. The changes needed in the algorithm above are:

- The condition to check that exactly two vertices have an odd degree is now changed to ensure that **No vertex has an odd degree**
- The process of adding a pseduo-edge and later deleting it is no longer needed as the Hierholzer algorithm returns an Eulerian Circuit automatically.

(c) Time Complexity

Both algorithms in parts(a) and (b) are almost identical except for the single step of adding an edge between s and t. In both algorithms the following parts are there:

Algorithm 2 Royal Routes (Eulerian trail via Hierholzer)

```
1: function CONSTRUCT GRAPH(n, Edges[m])
2:     //Edges[m] = list of edges between (u, v), n = number of vertices
3:     for i ← 0 to n do
4:         adj[i] ← empty list
5:         degree[i] ← 0
6:     end for
7:     for i ← 0 to m do
8:         u ← Edge[i][0];
9:         v ← Edge[i][1];
10:        degree[u] ← degree[u] + 1
11:        degree[v] ← degree[v] + 1
12:    end for
13: end function
14: function ROYALROUTES(n)
15:    odd ← 0
16:    s ← −1, t ← −1                                  ▷ start/end candidates
17:    for i ← 0 to n do
18:        if degree[i] mod 2 = 1 then
19:            odd ← odd +1
20:            if s = −1 then
21:                s ← i
22:            else
23:                t ← i
24:            end if
25:        end if
26:        if degree[i] = 0 then
27:            return Not Valid                        ▷ some town is isolated
28:        end if
29:    end for
30:    if odd ≠ 2 then
31:        return Not Valid                            ▷ necessary condition violated
32:    end if
33:    add an edge between (s, t)                      ▷ make graph Eulerian
34:    create empty stack S create empty list C        ▷ the circuit/trail
35:    u ← s
36:    while (not S.EMPTY()) or (|adj[u]| > 0) do
37:        if |adj[u]| = 0 then
38:            append u to C
39:            u ← S.POP()
40:        else
41:            S.PUSH(u)
42:            choose v ∈ adj[u] through an unused edge (u, v)
43:            remove edge (u, v) from adj[u] and adj[v]
44:            u ← v
45:        end if
46:    end while
47:    append u to C
48:    remove the edge (s, t) from C to obtain the trail
49:    return C
50: end function
```

- Visit each vertex exactly once during the construction of the Adjacency lists and computations of degree. This gives us a time complexity of $\mathcal{O}(n+m)$.
- A single scan of the vertexes to verify the sufficient conditions: $T(n) = \mathcal{O}(n)$
- The algorithm to produce the Eulerian circuit visits each edge exactly once as per the constraint in the question and as described in (a) and each vertex is pushed and popped in the stack exactly once. Hence, this loop creates a time complexity of $\mathcal{O}(m+n)$

Hence, both algorithms in part(a) and (b) give the following time complexity:

$$T(m,n) = \mathcal{O}(m+n)$$

Space Complexity

The above algorithm uses space in the following manners:

- Adjacency lists of size $\mathcal{O}(m+n)$
- An array degree[n] which is also of size n and hence gives a space complexity of $\mathcal{O}(n)$
- The stack can only have a maximum size of $\mathcal{O}(n)$ since n vertices are pushed and popped exactly once

$$\textbf{Space Complexity} = \mathcal{O}(m+n)$$

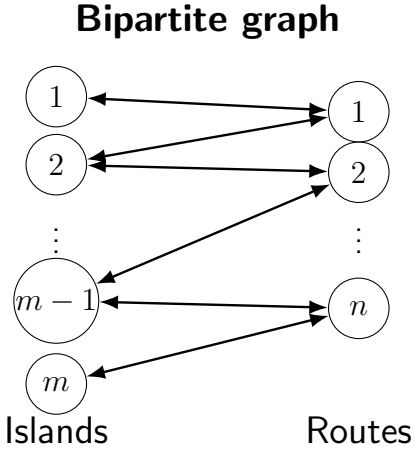
3) Sky Trams

(a) Formulation

We can formulate the problem statement as a Bi-partite graph with sets I and T.

- **Nodes:** The set of all islands(I) and trams(T). Total number of nodes = $n + m$
- **Edges:** Each edge is between an island and a route. The route will have an edge to each island that it visits. There will be no edges between two routes or between two islands in this formulation of the graph. The number of edges can be calculated as follows: Each node in the set I has size(T) = n options to connect to. Therefore each node in set I can have a maximum of n edges. Applying the same for each m islands there can be a total of mn edges at max.

Now that we have a graph representing a map of connections between the routes and islands, we simply have to check connection between the start island and finish island. Our main problem is now to count the number of trams that lie on the shortest path to the finish island.



(b)Algorithm

To calculate the minimum number of trams required we do the follows:

- Conduct a BFS starting at the source island on the constructed above graph which would guarantee the shortest path from source to target.
- Maintain a variable *count* to keep track of the number of layers from which we can calculate the number of trams as given below

Using a BFS we can guarantee that we find the shortest path and due to the method we have formulated the graph we ensure that we use only valid routes that cycle between their respective islands. It is also due to the bi-partite nature that the BFS will alternatively process a set of islands and trams.
In our Bi-partite graph which will start with the node source, which is an island:

- All the even layers [0, 2, 4, ...] would be islands.
- All the odd layers [1, 3, 5, ...] would be the routes/trams.

Since we terminate the BFS when we reach our target island which is an even layer, then if there are a total of L layers, $L/2$ trams would have been taken to get there. Hence, returning $\frac{\text{number of layers}}{2}$ is the correct number of trams.

(C) Pseudocode

The pseudocode is as follows:

Algorithm 3 Minimum Number of Trams from Source to Target

```
1: function TRAMS(Routes[n], source, target)
2:     if source = target then
3:         return 0
4:     end if
5:     Create adjacency list representation of the graph graph[m + n]
6:     for i ← 0 to n − 1 do
7:         for each island j in routes[i] do
8:             append i to graph[j]
9:             append j to graph[i]
10:        end for
11:    end for                          ▷ The routes are indexed from m+1, m+2... n
12:    Start a BFS from source on graph
13:    Create a queue Q with source in it
14:    Create an array visited[m+n] = 0 for all indexes
15:    visited[start] = 1
16:    count ← 0
17:    while Q not empty do count ← count + 1 size ← size(Q)
18:        for i ← 1 to size do
19:            u ← dequeue(Q)
20:            if u = source then
21:                return count
22:            end if
23:            for v in graph[u] do
24:                if visited[v] = 0 then
25:                    visited[v] = 1
26:                    Q.enqueue(v)                    ▷ Add v to the queue
27:                end if
28:            end for
29:        end for
30:    end while
31:    return -1                        ▷ The target island wasn't connected
32: end function
```

(d) Time and Space Complexity

Space Complexity

The space taken by the graph approach is due to the following:

- $graph[m+n]$ which as discussed in (a) has a maximum of mn edges and hence it has a space complexity of $\mathcal{O}(mn)$
- Any other lists, arrays or queues in the algorithm use a maximum space for $\mathcal{O}(m+n)$

Time Complexity

The time complexity of this algorithm is mainly due to running a BFS traversal on $graph$. The time complexity of this is $\mathcal{O}$(number of nodes in graph+number of edges in graph = $\mathcal{O}(m+n+mn)$. In this expression, the term mn dominates and hence the time complexity of the given algorithm is $\mathcal{O}(mn)$.
Therefore, the complexities are:

$$\textbf{Space Complexity} = \mathcal{O}(mn)$$
$$\textbf{Time Complexity} = \mathcal{O}(mn)$$