

ESO207 – Theoretical Assignment #3

Name: Sumath Rengasami V

Roll No: 241054

Date: October 30, 2025

Question 1: Restoring Galactic Communication Network

The Galactic Federation maintains a communication network consisting of n planets, numbered from 1 to n , connected by m bidirectional communication channels. After a severe solar storm, parts of the network have been damaged, leaving several planets disconnected.

To restore communication, engineers can construct new channels between any two planets i and j . However, the construction cost is proportional to the square of the difference in their identification numbers:

$$\text{Cost}(i, j) = (i - j)^2$$

The goal is to ensure that Planet 1 (the command center) and Planet n (the outer colony) are connected—either directly or indirectly—by adding at most two new channels. The objective is to determine the minimum total cost required to achieve this connectivity.

- (a) Design an efficient algorithm to compute the minimum total cost needed to make Planet 1 and Planet n connected by adding at most two new edges. Clearly describe the main steps of your approach and justify any data structures used.
- (b) Discuss why the algorithm guarantees the minimum possible cost and provide pseudo code of your approach from part (a).
- (c) Analyze the time and space complexity of your code from part (b).

Solution.

- (a) The purpose of this algorithm is to find the smallest possible cost needed to reconnect Planet 1 (the command center) and Planet n (the outer colony) after a solar storm damages the communication network. Each new communication channel between two planets i and j costs $(i - j)^2$, and we are allowed to add at most two new channels.

Main Idea: After the storm, the network might break into multiple disconnected parts. If Planet 1 and Planet n already lie in the same component, they are already connected, and the cost is zero. If not, we need to decide whether to connect them directly with one edge or indirectly through another component using two edges.

Step 1: Finding Components We first run a Depth-First Search (DFS) to identify the connected components of the graph. Each planet is assigned a unique component ID, stored in the array $comp[i]$.

Step 2: Identifying Important Components Let C_1 be the component containing Planet 1, and C_n be the component containing Planet n . Two boolean arrays, inC_1 and inC_n , are used to mark which planets belong to these special components.

Step 3: Finding the Closest Planets Since planet numbers are already in increasing order from 1 to n , we can find how close every planet is to the nearest planet in C_1 and C_n without sorting. We do this using two passes across the list of planets: a left-to-right pass and a right-to-left pass. These sweeps compute:

$$dist1[i] = \min_{p \in C_1} |i - p|, \quad distn[i] = \min_{q \in C_n} |i - q|.$$

Each sweep takes linear time, $O(n)$.

Step 4: Computing Component Distances For each component C , we determine:

$$d_1(C) = \min_{i \in C} dist1[i], \quad d_n(C) = \min_{i \in C} distn[i].$$

These values represent how close each component is to the components of Planet 1 and Planet n .

Step 5: Calculating Possible Costs We consider two possibilities:

- (a) **Direct connection:** Add one edge connecting a planet in C_1 directly to a planet in C_n . The minimal cost is:

$$oneEdgeCost = \left(\min_{i \in C_n} dist1[i] \right)^2.$$

- (b) **Indirect connection:** Add two edges through an intermediate component C_k , one connecting C_1 to C_k and another connecting C_k to C_n . The total cost is:

$$twoEdgeCost = \min_{C_k} (d_1(C_k)^2 + d_n(C_k)^2).$$

Step 6: Choosing the Minimum Cost The algorithm compares the two options and returns the smaller of the two:

$$Answer = \min(oneEdgeCost, twoEdgeCost).$$

This guarantees that the network is restored with the lowest possible cost while using no more than two additional edges.

We use arrays to store the different distances and an adjacency list to store the graph

(b) We are taking G as the input graph network after the solar flare.

Why the Algorithm Guarantees the Minimum Cost: Correctness:

- The cost of constructing a new edge between two planets i and j is $\text{Cost}(i, j) = (i - j)^2$. Since this function is monotonic in $|i - j|$, minimizing the cost is equivalent to minimizing the absolute difference $|i - j|$ between connected planets.
- The algorithm first computes the connected components of the damaged network. If Planets 1 and n already lie in the same component, no new edges are required and the minimum cost is 0.
- If they are disconnected, two possible strategies can reconnect them:
 - (a) **One-edge connection:** Directly link a planet in C_1 (the component containing Planet 1) to a planet in C_n (the component containing Planet n). The algorithm finds the pair (i, j) minimizing $|i - j|$, which yields the least possible $(i - j)^2$ cost among all single-edge options.
 - (b) **Two-edge connection:** Connect C_1 and C_n through an intermediate component C_k . The total cost for a two-edge solution is $(\min_{x \in C_1, y \in C_k} |x - y|)^2 + (\min_{p \in C_k, q \in C_n} |p - q|)^2$. The algorithm evaluates this value for every component C_k and selects the smallest sum, ensuring the minimal total cost among all possible two-edge constructions.
- Finally, the algorithm compares the best one-edge cost and the best two-edge cost:

$$\text{Answer} = \min(\text{oneEdgeCost}, \text{twoEdgeCost}).$$

This guarantees that the final result is the minimum possible cost under the problem constraint of adding at most two new edges.

Conclusion: Because each step systematically computes the smallest possible distance between relevant components and the cost function strictly increases with distance, the algorithm always produces the global minimum total cost required to reconnect Planet 1 and Planet n .

Algorithm 1: Assigning Component IDs using DFS

Input: An undirected graph $G(V, E)$
Output: Array $comp[1..n]$ where $comp[v]$ stores the component ID of vertex v

```

1 Initialize arrays visited, comp
2 for  $i \leftarrow 1$  to  $n$  do
3   | visited[i]  $\leftarrow$  false
4   | comp[i]  $\leftarrow$  0
5 end for
6 componentID  $\leftarrow$  0
7 for  $v \leftarrow 1$  to  $n$  do
8   | if visited[v] = false then
9     |   componentID  $\leftarrow$  componentID + 1
10    |   DFS( $v$ , componentID) // DFS Code is below (Line 14)
11    | end if
12 end for
13 return comp
14 DFS( $v$ , componentID):
15 visited[v]  $\leftarrow$  true
16 comp[v]  $\leftarrow$  componentID
17 foreach  $u$  in which is a neighbour of  $v$  do
18   | if visited[u] = false then
19     |   DFS( $u$ , componentID)
20   | end if
21 end foreach
```

Algorithm 2: Minimum Cost to Connect Planet 1 and Planet n

```

1 Run the above DFS on  $G$  to label each planet with a component ID. After
  running the above DFS let  $comp[i]$  be the component ID of the  $i$  planet.
2 Let  $C_1$  = component ID of Planet 1, and  $C_n$  = component ID of Planet  $n$ .
3 if  $C_1 = C_n$  then
4   | return 0 // As they are already connected
5 end if
6 Create boolean arrays  $inC1[1..n]$  and  $inCn[1..n]$ :
7 for  $i = 1$  to  $n$  do
8   |  $inC1[i] \leftarrow (comp[i] = C_1)$ ; //  $inC1[i]$  is true if  $comp[i] = C_1$ 
9   |  $inCn[i] \leftarrow (comp[i] = C_n)$ ; //  $inCn[i]$  is true if  $comp[i] = C_n$ 
10 end for
```

```

1 Initialize arrays  $dist1[1..n]$  and  $distn[1..n]$  with  $\infty$ .
2 (a) Distance to Component(1):
3  $last \leftarrow -1$ 
4 for  $i = 1$  to  $n$  do
5   if  $inC1[i]$  then
6      $last \leftarrow i$ ;
7   if  $last \neq -1$  then
8      $dist1[i] \leftarrow i - last$ ;
9  $last \leftarrow -1$ 
10 for  $i = n$  to  $1$  do
11   if  $inC1[i]$  then
12      $last \leftarrow i$ ;
13   if  $last \neq -1$  then
14      $dist1[i] \leftarrow \min(dist1[i], last - i)$ ;
15 (b) Distance to Component(n):
16  $last \leftarrow -1$ 
17 for  $i = 1$  to  $n$  do
18   if  $inCn[i]$  then
19      $last \leftarrow i$ ;
20   if  $last \neq -1$  then
21      $distn[i] \leftarrow i - last$ ;
22  $last \leftarrow -1$ 
23 for  $i = n$  to  $1$  do
24   if  $inCn[i]$  then
25      $last \leftarrow i$ ;
26   if  $last \neq -1$  then
27      $distn[i] \leftarrow \min(distn[i], last - i)$ ;
28 For each component  $C$ , set:
29  $d_1(C) = \min_{x \in C} dist1[x]$ 
   // we know which component  $x$  belongs to because of compID
30  $d_n(C) = \min_{x \in C} distn[x]$ 
31  $oneEdgeCost = (\min_{x \in C_n} dist1[x])^2$ 
32  $twoEdgeCost = \min_C (d_1(C)^2 + d_n(C)^2)$ 
33  $answer = \min(oneEdgeCost, twoEdgeCost)$ 
34 return  $answer$ 

```

- (c) The network is a graph which can be represented as $G = (V, E)$, where $|V| = n$ is the number of vertices and $|E| = m$ is the number of edges.

Initialization: Marking all vertices as unvisited and assigning initial component IDs takes

$$O(n)$$

time and space, since each vertex is initialized exactly once.

DFS Traversal and Assigning Component IDs: Each vertex is visited exactly once during the DFS, and each edge is explored at most twice (once from each endpoint). Hence, the total time complexity for all DFS calls is

$$O(n + m)$$

This is also the space complexity as the graph requires $O(n + m)$ space.

Boolean array: To assign the values of both the boolean arrays it takes $O(n)$ time (because we visit all vertex) and since each array stores all the vertex it also requires $O(n)$ space.

Dist1 and Distn array: To assign the values of both the distance arrays it takes $O(n)$ time because (we visit all vertex twice once during forward and the other during backward to store minimum distance) and since each array stores all the vertex it also requires $O(n)$ space.

$d_1(C)$ and $d_2(C)$ array: To assign the values of both the distance arrays for each component it takes $O(n)$ time because we visit all vertex once (we visit all vertex of all components) and since each array stores the minimum of all components it also requires $O(n)$ space as maximum number of components is n .

One edge cost: To define one edge cost we go through all $dist1$ of vertex in the component containing n so it takes $O(n)$ time and $O(1)$ space

Two edge cost: To define two edge cost we go through all d_1 and d_n of all component so it takes $O(n)$ (as maximum number of components is n) time and $O(1)$ space

Total Time Complexity: Combining all the above steps, the overall time complexity is:

$$T(n, m) = O(n) + O(n + m) = O(n + m)$$

Space Complexity: The algorithm uses the following data structures:

- The adjacency list representation of the graph, which takes $O(n + m)$ space.
- Two arrays, `visited[1..n]` and `comp[1..n]`, which each take $O(n)$ space.
- The recursion stack for DFS, which in the worst case can store $O(n)$ vertices.

Therefore, the overall space complexity is:

$$S(n, m) = O(n + m)$$

Question 2: The Postal Routes of Valoria

In the ancient kingdom of Valoria, there are n towns connected by m two-way trade routes. Each route connects two distinct towns and can be traveled in both directions.

The King wishes to establish a grand postal route for royal messengers. The rules for the route are as follows:

- Every trade route must be used exactly once during the journey.
 - A messenger may travel along the routes in any direction, but cannot reuse any route.
 - The postal routes starts and ends in different towns.
- (a) Design an efficient algorithm to decide whether a valid royal postal route exists in the kingdom of Valoria. If it exists, also provide such a route. Explain your approach.
- (b) The royal messengers have now requested you to find a route that starts and ends in the same town. Mention the changes you need to make in your algorithm in part (a).
- (c) Analyze the time and space complexity of your algorithm for both the parts (a) and (b).

Solution.

- (a) We have to find out a Eulerian path in an undirected graph $G(V, E)$.

For a connected undirected graph, an Eulerian path exists if and only if exactly two vertex have odd degree. This is because, whenever the messenger (or walker) enters a vertex via one edge, there must be another unused edge to leave it, except for the starting and ending vertex where the walk starts/ends.

Step-by-Step Approach:

- (a) **Connectivity and Degree Check:**

First, verify that the graph is connected and that exactly two vertex have odd degree. If these conditions are not met, an Eulerian path cannot exist.

- (b) **Initialization:**

Select any vertex say s with odd degree > 0 as the starting point. Maintain:

- a **stack** to track the current path,
- an **adjacency list** to represent edges,
- and a boolean array **used[]** to mark edges that have already been traversed.

- (c) **Building the Path:**

- Push the starting vertex onto the stack.
- While the stack is not empty:

- i. Let v be the top of the stack.
- ii. If v has an unused adjacent edge (v, u) :
 - Mark this edge as used.
 - Push u onto the stack.
- iii. Otherwise, pop v from the stack and add it to the final path.

(d) Result:

When all edges have been used, the list of popped vertices represents the Eulerian path.

Correctness: Since the algorithm:

- traverses every edge exactly once,
- ensures all edges are connected into a single cycle,
- and uses the even-degree condition to guarantee exit from every vertex(except start and end vertex),

it correctly constructs a valid Eulerian path whenever one exists.

Algorithm 3: Finding Eulerian Path

Input: An undirected graph $G(V, E)$ representing towns and trade routes.

Output: A valid Eulerian trail, if it exists.

```

1 Compute  $\deg(v)$  for each vertex  $v$ . Count how many have odd degree.
2 if number of odd-degree vertices  $\neq 2$  then
3   | return NO TRAIL EXISTS
4 end if
5 Run a DFS from any town with degree to check if all towns are reachable ( $G$  is
   connected). If componentID  $> 1$  then  $G$  is not connected
6 if  $G$  is disconnected then
7   | return NO TRAIL EXISTS
8 end if
9 Let  $s$  and  $t$  be the two odd-degree towns. Start at  $s$ .
10 Maintain a stack  $S$  and an empty list  $path$ . Push  $s$  into the stack
11 while  $S$  is not empty do
12   | Let  $v$  be the top of the stack.
13   | if  $v$  has any unused adjacent edge then
14     | Choose an unused edge  $(v, u)$ , mark it used, and push  $u$  onto  $S$ .
15   | end if
16   | else
17     | Pop  $v$  from  $S$  and append it to  $path$ .
18   | end if
19 end while
20 return  $path$ .
```

Algorithm 4: Counting the Number of Connected Components using DFS

Input: An undirected graph $G(V, E)$

```

1 Initialize visited array
2 for  $i \leftarrow 1$  to  $n$  do
3   | visited[i]  $\leftarrow$  false
4   | comp[i]  $\leftarrow$  0
5 end for
6 componentID  $\leftarrow$  0
7 for  $v \leftarrow 1$  to  $n$  do
8   | if visited[v] = false then
9     |   componentID  $\leftarrow$  componentID + 1
10    |   DFS(v)
11   | end if
12 end for
13 return componentID

14 Procedure DFS( $v$ ):
15 visited[v]  $\leftarrow$  true
16 foreach  $u$  which is a neighbour of  $v$  do
17   | if visited[u] = false then
18     |   DFS(u)
19   | end if
20 end foreach
```

- (b) Since the path starts and ends at different vertex it is an Eulerian path. This is different from an Eulerian circuit as it needs to start and end at the same vertex while visiting every edge exactly once unlike the Eulerian path where we can start and end at different vertex.

Two things will change in the algorithm :

- All vertex must have even degree (instead of exactly two vertex should have odd degree in case of part (a))
- We can start at any vertex (instead of starting at vertex with odd degree like in case of art (a))

Everything else in the algorithm remains the same.

Algorithm 5: Finding Eulerian Circuit

Input: An undirected graph $G(V, E)$ representing towns and trade routes.

Output: A valid Eulerian trail, if it exists.

```

1 Compute  $\deg(v)$  for each vertex  $v$ . Count how many have odd degree.
2 if number of odd-degree vertices  $\neq 0$  then
3   | return NO TRAIL EXISTS
4 end if
5 Run a DFS from any town with degree to check if all towns are reachable ( $G$  is
   connected). If componentID  $> 1$  then  $G$  is not connected
6 if  $G$  is disconnected then
7   | return NO TRAIL EXISTS
8 end if
9 Start at any vertex  $s$ .
10 Maintain a stack  $S$  and an empty list path. Push  $s$  into the stack
11 while  $S$  is not empty do
12   | Let  $v$  be the top of the stack.
13   | if  $v$  has any unused adjacent edge then
14     | Choose an unused edge  $(v, u)$ , mark it used, and push  $u$  onto  $S$ .
15   | end if
16   | else
17     | Pop  $v$  from  $S$  and append it to path.
18   | end if
19 end while
20 return path.
```

(c) Let the graph be $G = (V, E)$, where $|V| = n$ is the number of vertices and $|E| = m$ is the number of edges.

1. Degree and Connectivity Check:

- Computing the degree of each vertex requires scanning all adjacency lists once, taking $O(n + m)$ time [For both algorithms in part (a) and (b)].
- Checking connectivity using DFS also takes $O(n + m)$ time, since each vertex and edge is explored once.[For both algorithms in part (a) and (b)]

Thus, the total cost is:

$$O(n + m)$$

2. Constructing the Eulerian Path :

- Each edge is traversed exactly once when the algorithm moves from one vertex to another and marked as used.
- Every edge is also inspected once in the adjacency list while skipping used edges. Hence, the total time spent is proportional to m .
- Each vertex is pushed and popped from the stack at most once per edge incident on it, which contributes $O(n + m)$ operations in total.

Therefore, the main construction step also runs in:

$$O(n + m)$$

[For both algorithms in part (a) and (b)]

Total Time Complexity : Combining all steps, the overall time complexity is:

$$T(n, m) = O(n + m)$$

Space Complexity :

- The adjacency list representation of the graph takes $O(n + m)$ space.
- Arrays such as `visitedEdges[]` to mark edges require $O(m)$ space.
- The **stack** used may hold up to $O(m)$ vertices in the worst case.
- The final Eulerian path list also stores $O(m)$ vertices for algorithm in part (a) and $O(m + 1)$ for algorithm in part (b).

Thus, the total space complexity is:

$$S(n, m) = O(n + m)$$

[For both algorithms in part (a) and (b)]

Therefore,

$T(n, m) = O(n + m), \quad S(n, m) = O(n + m)$
--

for both algorithms in part (a) and (b)

Question 3: The Sky Trams of Aetheria

In the floating city of Aetheria, hundreds of sky trams glide endlessly between levitating islands. Each tram follows a fixed circular route, repeating the same sequence of islands forever. There are n tram routes, where the i^{th} route is represented by a list of at most m distinct island identifiers that the tram visits in order. Assume all island identifiers are unique integers lying between 1 and m (both inclusive). For example, if $route[0] = [1, 2, 3, 4]$, it means that the first tram travels as

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \dots$$

A messenger begins their journey at island source and wishes to reach island target. The messenger may board or exit a tram only at an island that the tram visits. Two trams are said to intersect if they both stop at the same island, allowing the messenger to switch from one tram to another.

Your task is to determine the smallest number of trams the messenger must use to travel from source to target. If it is impossible to reach the target island, report that as well.

- Formulate the problem using a graph. Explain your formulation and mention what do the nodes and edges represent in your graph.
- Design an efficient algorithm to compute the minimum number of trams required to travel from source to target. Describe your approach.
- Provide a pseudocode to implement your approach.
- Analyze the time and space complexity of your algorithm.

Solution.

- We define a bipartite graph $G = (X \cup Y, E)$, where:

- X represents the set of islands.
- Y represents the set of tram routes.
- An edge $(i, r) \in E$ exists if island i belongs to tram route r .

Graph Traversal:

- Moving from an island to a tram node represents boarding that tram.
- Moving from a tram to an island node represents traveling along the tram to that stop.

The goal is to find the shortest sequence of tram nodes connecting the source island to the target island. This ensures that traversing from one island to another counts only the number of distinct tram routes used.

(b) Let the newly constructed bipartite graph be G . Algorithm approach :

- (a) If `source == target`, return 0 (no tram needed).
- (b) To represent the new bipartite graph efficiently, we construct an adjacency list that connects every island to the tram routes that contains it. This adjacency list is implemented as a mapping named `stops_to_route`, where:

$$\text{stops_to_route}[i] = \{r_j \mid \text{tram route } r_j \text{ visits island } i\}.$$

Construction Process:

- (a) Initialize an empty list for each island identifier from 1 to m .
- (b) Iterate through all tram routes. For each route r_j :
 - Examine its list of islands $[v_1, v_2, \dots, v_k]$.
 - For each island v_i in this list, append the current route index r_j to the list `stops_to_route` $[v_i]$.
- (c) After this step, every island's entry in `stops_to_route` contains all routes that pass through that island.
- (a) Initialize a distance array and initialize all values to -1 . Then run BFS to find distance of target from source.
- (b) If BFS ends without finding `target` (i.e, distance of target after BFS is -1), return `IMPOSSIBLE`. Else return the value of $\text{distance}[\text{target}]/2$.

We divide the BFS distance by 2 because the raw distance represents the total number of edges traversed in the bipartite graph. Since the graph alternates between islands and tram routes, every two edges correspond to a single tram ride (boarding and leaving a tram). As the path starts and ends at island nodes, half the total number of edges equals the number of trams required to reach the target.

(c) Pseudocode for the above algorithm :

Algorithm 6: Minimum Trams via Bipartite BFS (divide distance by 2)

Input: Number of islands m ; number of routes n ; array `route[1..n]` where each `route[r]` lists the islands that route r visits; integers `source`, `target`

Output: Minimum number of trams required, or IMPOSSIBLE if unreachable

```

1 if source = target then
2   | return 0
3 end if
4 for  $r \leftarrow 1$  to  $n$  do
5   | foreach island  $i$  in route[r] do
6   |   | append  $r$  to stops_to_route[i] //Let  $H$  be the newly constructed
7   |   |                               bipartite graph whose adjacency
8   |   |                               list is stops_to_route
9   | end foreach
10 end for
11 for  $i \leftarrow 1$  to  $m$  do
12   | dist_island[i]  $\leftarrow -1$ 
13 end for
14 for  $r \leftarrow 1$  to  $n$  do
15   | dist_route[r]  $\leftarrow -1$ 
16 end for
17 queue  $\leftarrow$  empty queue of pairs (type, id)
18 dist_island[source]  $\leftarrow 0$ ; enqueue(queue, (island, source))
19 while queue not empty do
20   | (type, id)  $\leftarrow$  dequeue(queue)
21   | if type = island then
22   |   |  $u \leftarrow id$ 
23   |   | foreach route  $r$  in stops_to_route[u] do
24   |   |   | if dist_route[r] = -1 then
25   |   |   |   | dist_route[r]  $\leftarrow$  dist_island[u] +1
26   |   |   |   | enqueue(queue, (route,  $r$ ))
27   |   |   | end if
28   |   | end foreach
29   | end if
30   | else // type = route
31   |   |  $r \leftarrow id$ 
32   |   | foreach island  $v$  in route[r] do
33   |   |   | if dist_island[v] = -1 then
34   |   |   |   | dist_island[v]  $\leftarrow$  dist_route[r] +1
35   |   |   |   | enqueue(queue, (island,  $v$ ))
36   |   |   | end if
37   |   | end foreach
38   | end if
39 end while
40 if dist_island[target] = -1 then
41   | return IMPOSSIBLE
42 end if
43 else
44   | return dist_island[target] /2
45 end if

```

(d) We know that :

$$n = \text{number of routes}, \quad m = \text{number of islands}, \quad K = \sum_{r=1}^n |\text{route}[r]|$$

be the total number of *(route, island)* incidences (i.e., the total number of stops across all routes). Note that $K \leq nm$.

Building the incidence lists (stops_to_route): The nested loop that appends route r into `stops_to_route[i]` for every island $i \in \text{route}[r]$ touches each incidence once. Hence the construction time is

$$T_{\text{build}} = O\left(\sum_{r=1}^n |\text{route}[r]|\right) = O(K) = O(nm).$$

Distance-array initialization : Initializing `dist_island[1..m]` and `dist_route[1..n]` is $O(m) + O(n)$.

Time Complexity of BFS: In general, BFS runs in $O(V + E)$ time, where V is the total number of vertices and E is the total number of edges in the graph. For our bipartite graph:

$$V = m + n, \quad E = K = \sum_{r=1}^n |\text{route}[r]| \leq mn.$$

Therefore, the time complexity of BFS is:

$$O(V + E) = O((m + n) + K) = O(m + n + mn) = O(mn)$$

in the worst case, when every route passes through every island.

Total time. Summing the stages:

$$T(n, m) = T_{\text{build}} + O(n) + O(m) + O(nm) = O(nm)$$

Since $K \leq nm$, the worst-case bound is $O(nm)$, and the analysis is tight in terms of the actual input size K .

Space Complexity Analysis

Input storage. The input array `route[1..n]` already stores all incidences once; its size is

$$S_{\text{input}} = O(K) = O(nm).$$

Constructed adjacency list (island \rightarrow routes). The algorithm builds `stops_to_route[1..m]`. Each incidence contributes one entry to exactly one island list, hence

$$S_{\text{adj}} = O(K) = O(nm).$$

Distance/visited arrays. Two arrays store BFS discovery levels (doubling as visited flags):

$$\text{dist_island}[1..m], \text{dist_route}[1..n] \Rightarrow S_{\text{dist}} = O(m + n).$$

BFS queue. The queue holds pairs $(type, id)$ for islands and routes. In the worst case it can contain $O(V) = O(m + n)$ elements:

$$S_{\text{queue}} = O(m + n).$$

Total space. Excluding the given input `route[]` (size $\Theta(K)$), the additional working space is

$$S_{\text{aux}} = S_{\text{adj}} + S_{\text{dist}} + S_{\text{queue}} = O(nm) + O(m + n) + O(m + n) = O(nm + m + n).$$

Including the input, the overall space is

$$S_{\text{total}} = O(nm) + O(nm + m + n) = O(nm + m + n) = O(nm).$$

Summary. The algorithm uses space in the size of :

$$\boxed{S = O(nm)}.$$

End of Solutions.