Assignment Title:

Individual Coursework

Module Name:

ST5003CEM ADVANCED ALGORITHMS

Submission Date:

30th January 2026

Submitted By:

Prabin Babu Kattel

Coventry ID: 15370743

Submitted To:

Mr. Hikmat Saud

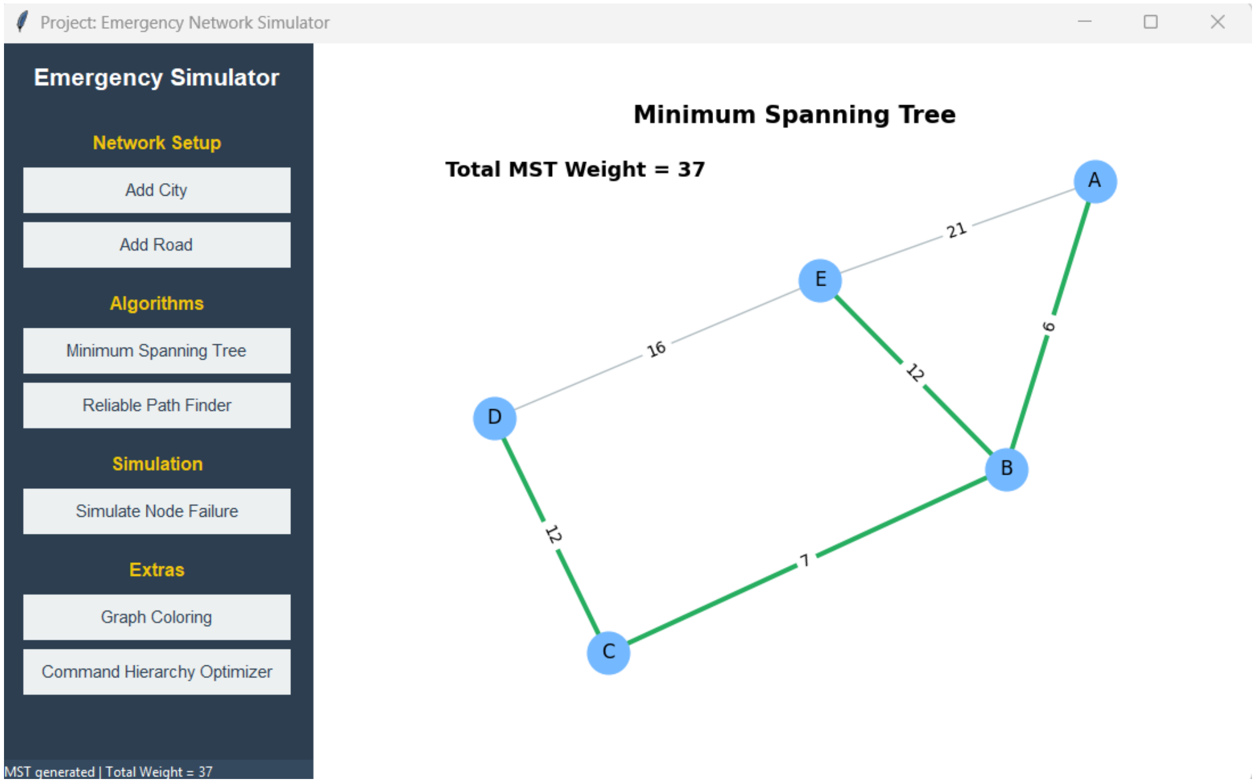
GitHub Link :

https://github.com/dailydoseofgithub/advancedalgorithm

QUESTION NO 5

1. Dynamic MST Visualization

Here, we can see how the MST visualization is on the work.



Total MST Weight = 37

MST generated | Total Weight = 37

Figure 1 MST Visualization 1

Auto Update MST when new node is added.

We can add nodes however we like, lets add a new node now.

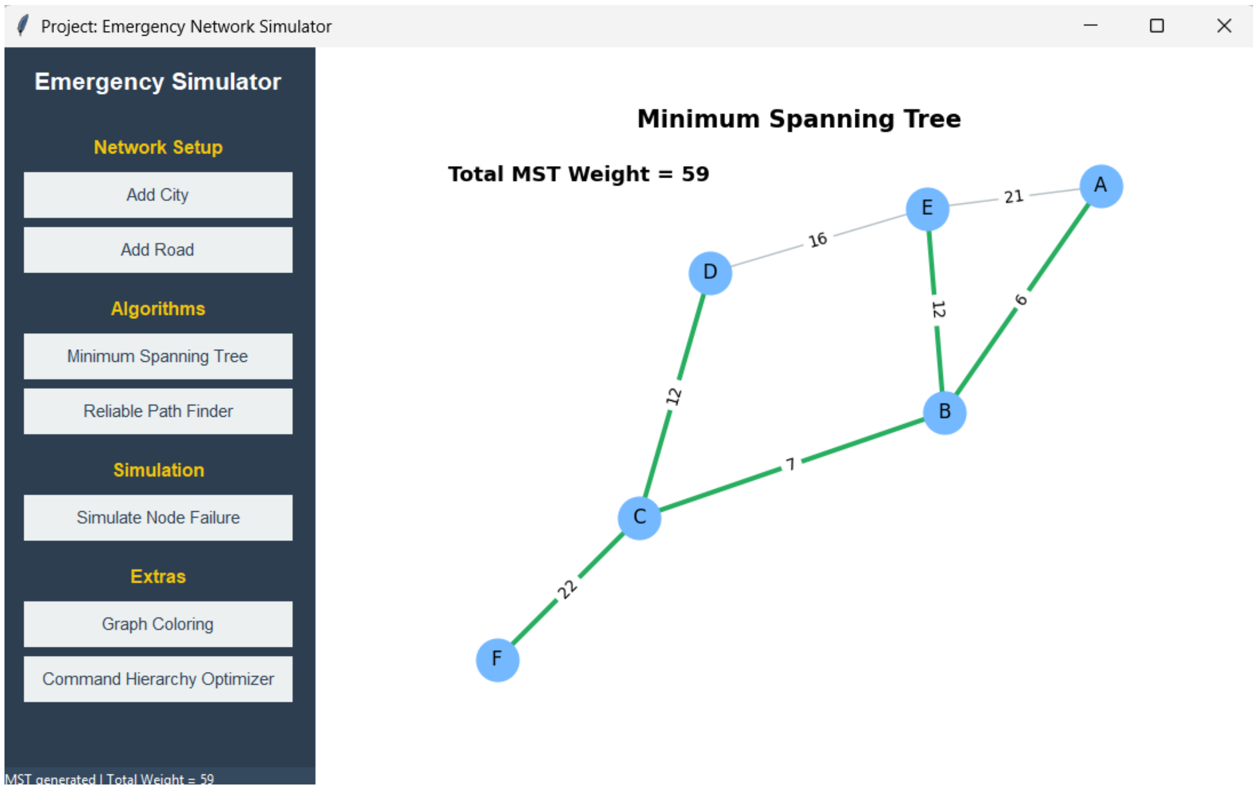


Figure 2 MST update after new node

Algorithm Used:

The algorithm that we used is **Kruskal's Algorithm.**

Why Kruskal's Algorithm is Suitable Here

- It works efficiently on sparse graphs (typical for city networks)
- Easy to update when edges are added or removed
- Ensures minimum total communication / infrastructure cost
- Avoids cycles automatically

This makes it ideal for dynamic emergency networks.

Step-by-Step Backend Logic

Collect all roads (edges) with their weights
Each road represents communication cost or travel time

Sort edges in ascending order of weight

Initialize disjoint sets (Union–Find)
Each city starts in its own component

Iteratively select edges

- Pick the smallest edge
- Add it **only if it does not create a cycle**
- Merge the connected components

Stop when (V − 1) edges are selected
V = number of cities

After this,

The code then:

- Extracts MST edges
- Computes total MST weight
- Visualizes the result

Code:

```
total_weight = sum(self.graph[u][v]['weight'] for u, v in mst.edges())
```

Time Complexity Analysis

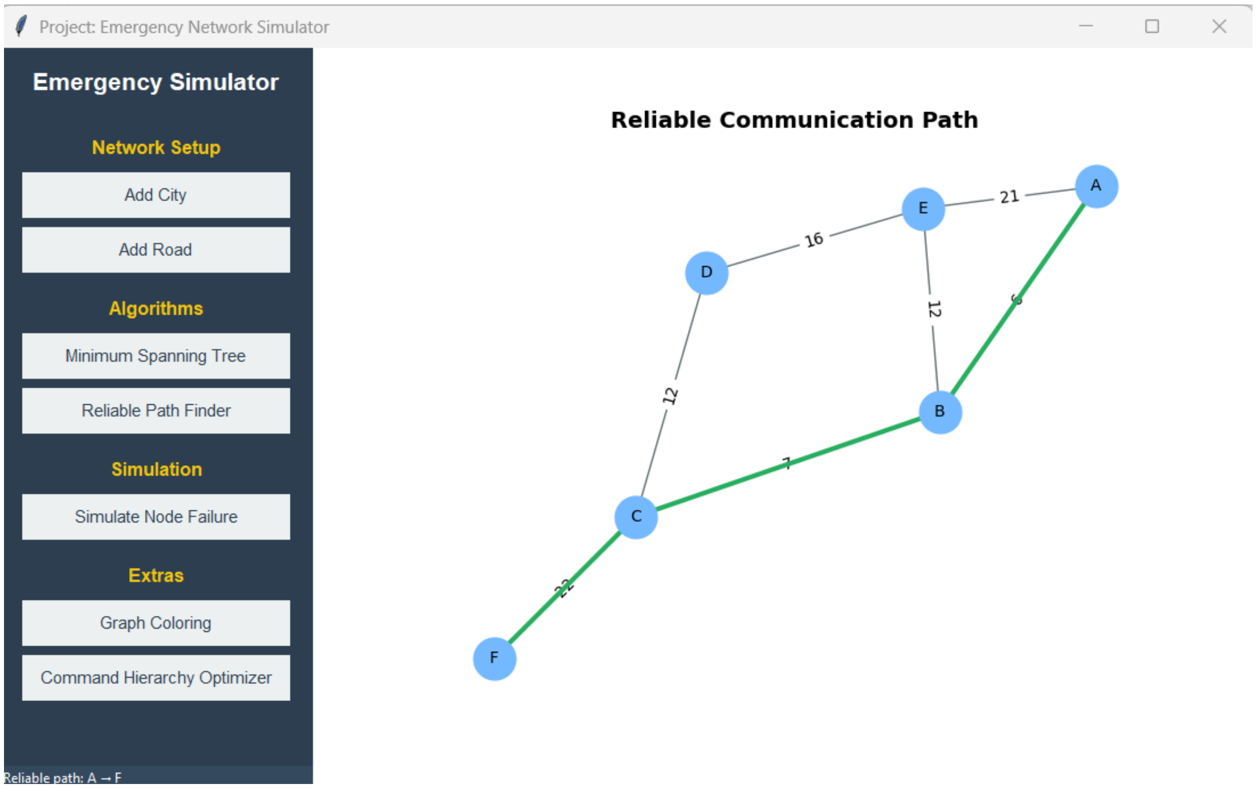
Suppose:

- V = number of cities (nodes)
- E = number of roads (edges)

Step	**Time Complexity**
Sorting edges	O(E log E)
Union-Find operations	O(E α(V)) ≈ O(E)
Total	O(E log E)

Final Time Complexity : O(E log E)

2. Reliable Path Finder with GUI Controls



Reliable path: A → F

Figure 3 Reliable Path

The Reliable Path Finder uses Dijkstra's Shortest Path Algorithm

- It optimizes minimum total weight
- Weight represents distance / time / reliability cost
- Guarantees the most reliable communication path

Backend Logic:

Code:

```
path = nx.shortest_path(self.graph, src, dst, weight='weight')
```

Backend Processing

First the user clicks the “Reliable Path Finder” button, then the GUI prompts user for

- Source City
- Destination City

Once inputs are received:

- The algorithm searches for the shortest weighted path
- Uses current graph state
- The program automatically takes into account for:
  - Failed nodes
  - Removed edges
  - Network changes

Then the node path is converted into edges.

```
edges = list(zip(path, path[1:]))
```

This steps acts like a bridge of algorithm output to GUI visualization.

After this, the program gives visual feedback to user

```
self.draw_graph(edges, "Reliable Communication Path")
```

3. Command Hierarchy Optimizer

The Command Hierarchy Optimizer allows the user to view the current binary command tree. User can click optimize to rebalance the tree and minimize the longest communication path from HQ. User can easily compare between the older and optimized versions.

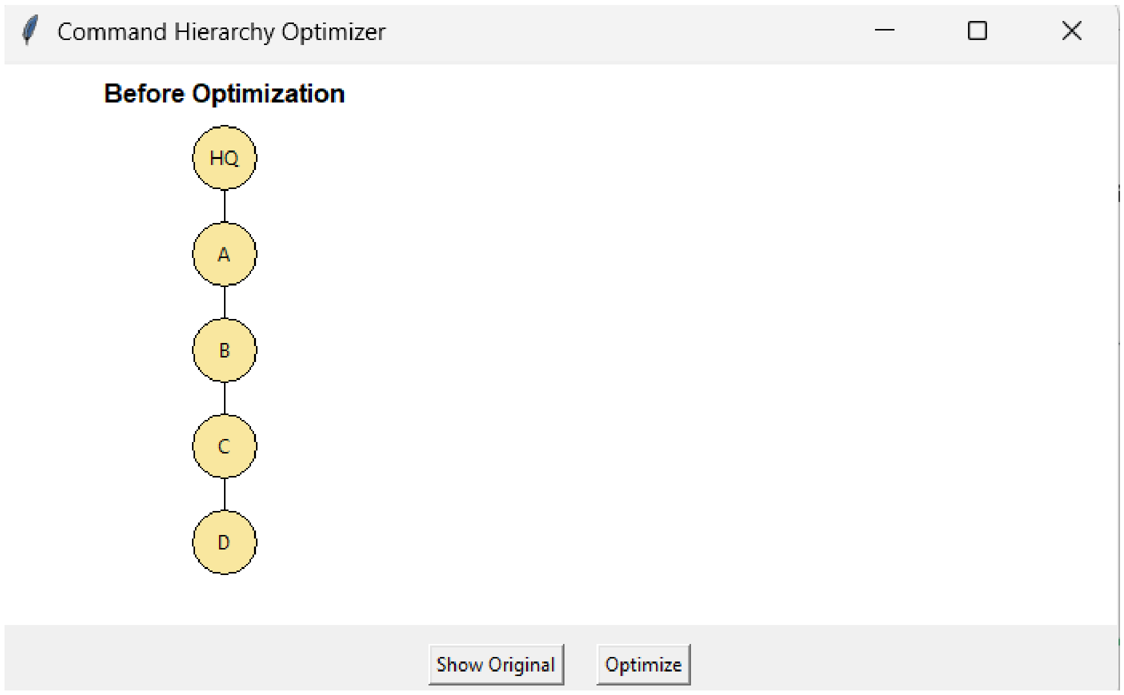


Figure 4 Before Optimization of Command Hierarchy

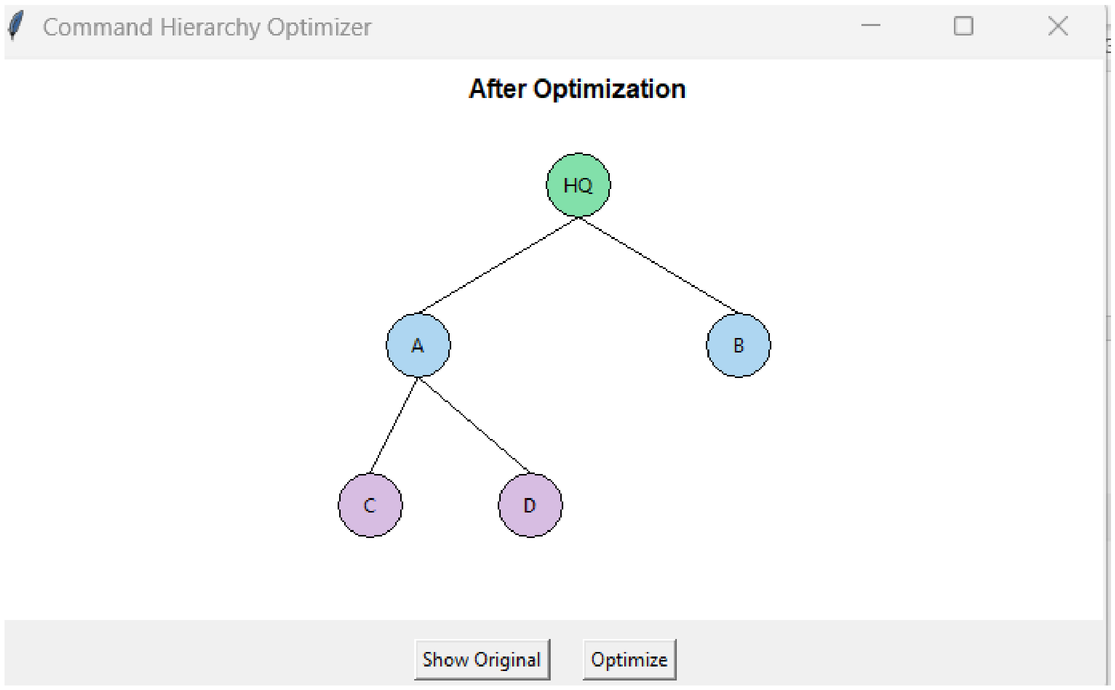
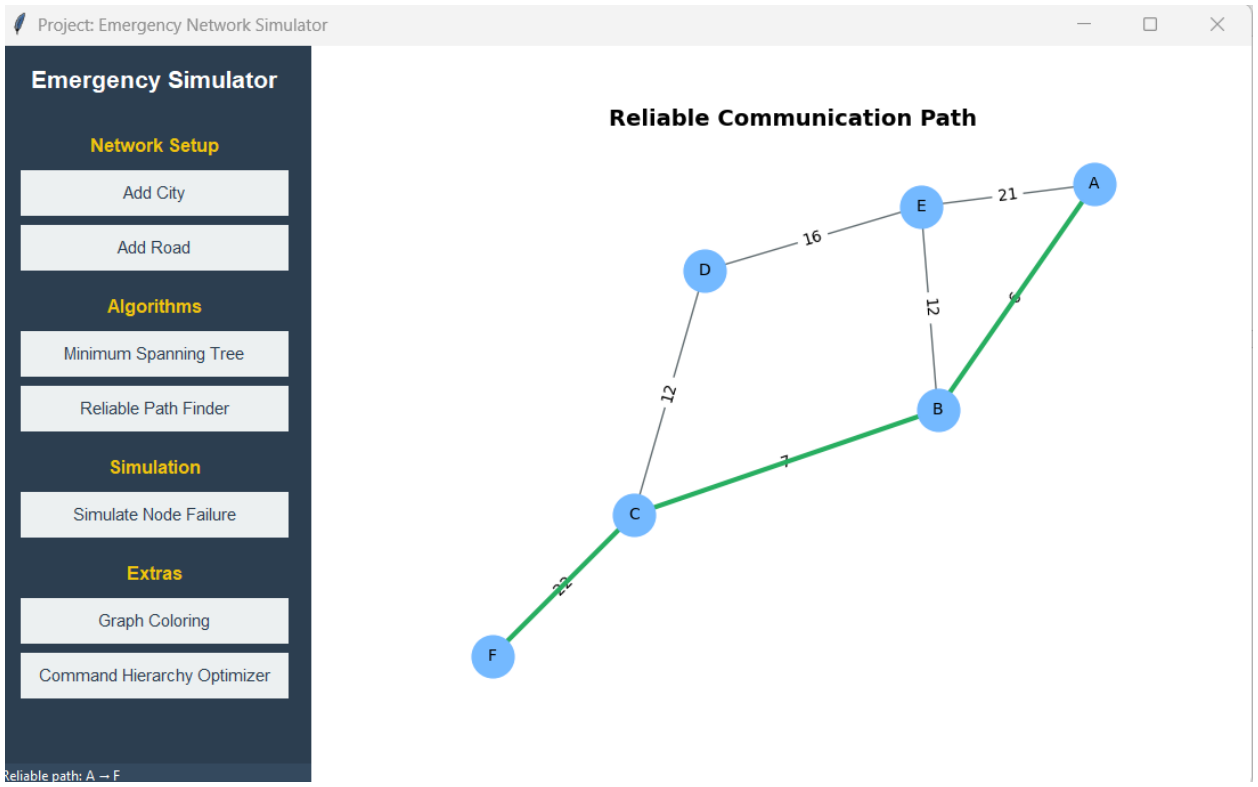


Figure 5 After Optimization of Command Hierarchy

4. Failure Simulation & Rerouting Module

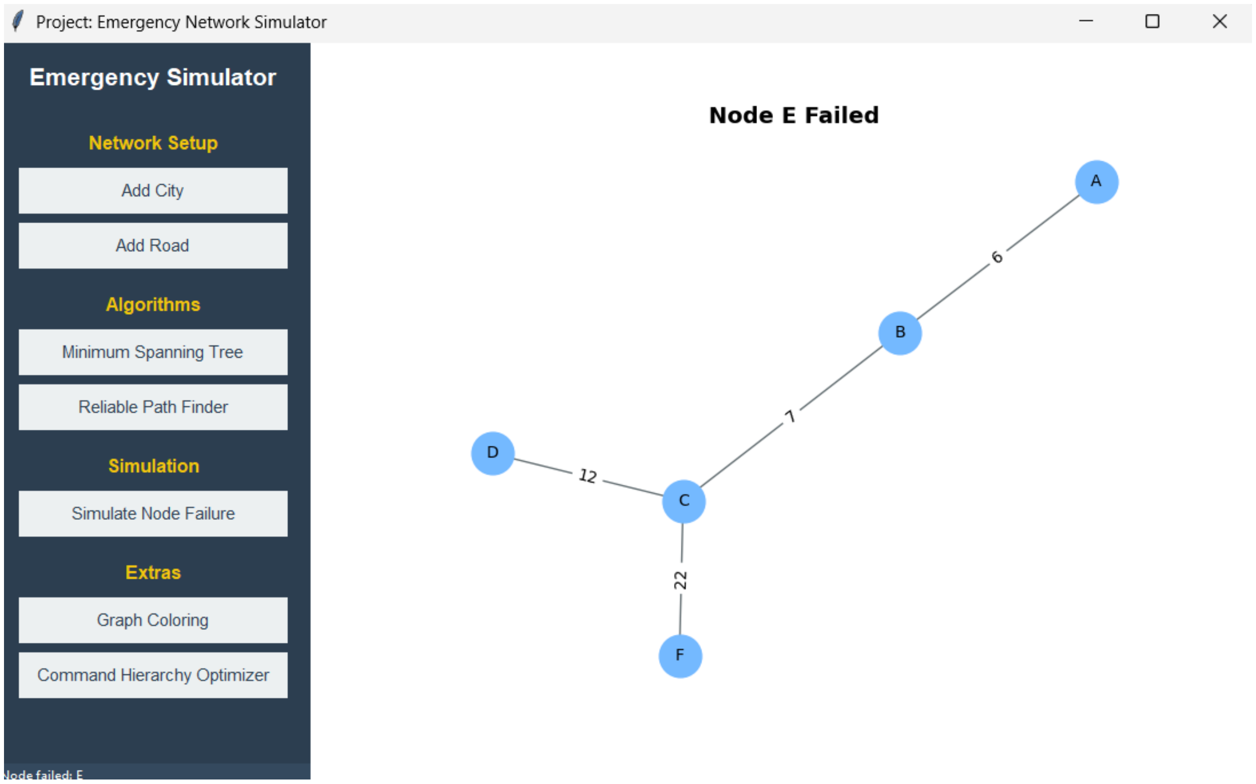
The GUI allows user to simulate node failure also, and based on the path finding techniques reroutes the path.



Reliable path: A → F

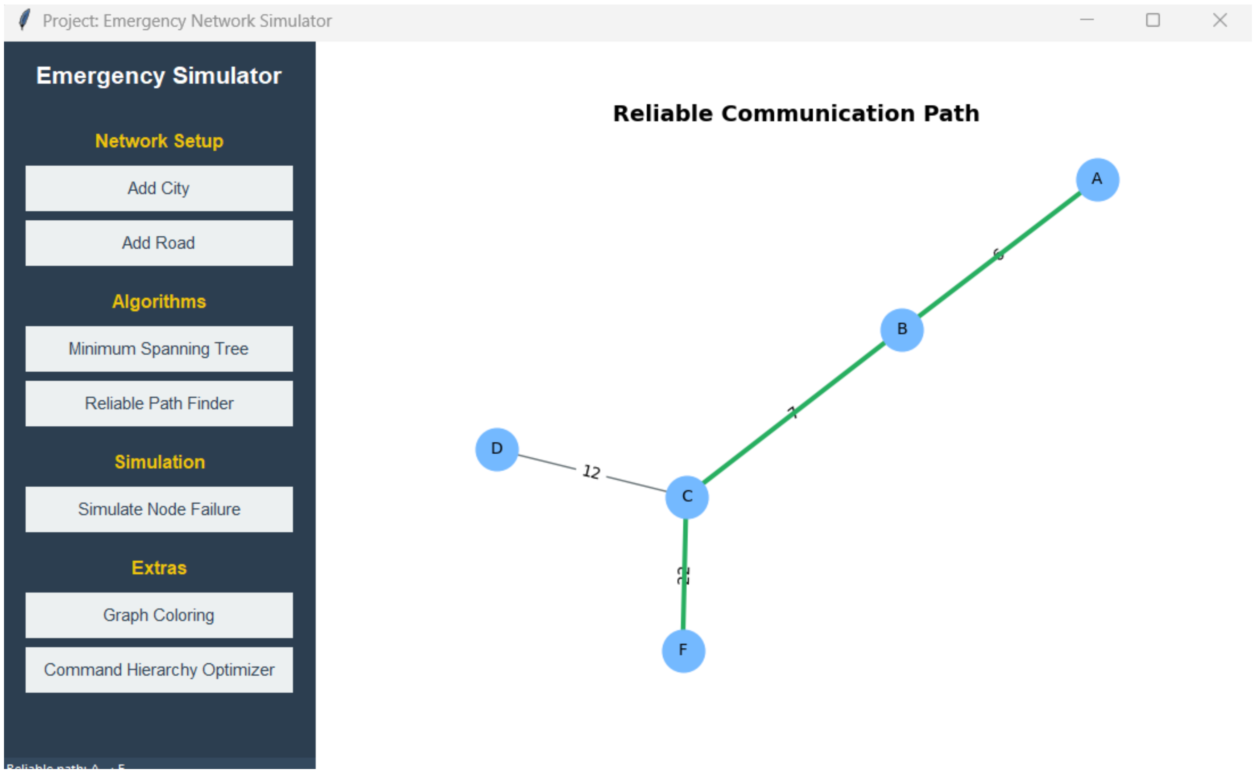
Figure 6 Before Node Failure Simulation

Let’s disable a node now, to simulate node failure and see the results.\



Node failed: E

Figure 7 After Simulating Node Failure



Reliable path: A → F

Figure 8 Reliable Path After Node Failure

The core idea is:

Remove the failed node then recompute connectivity after that update communication paths.

Graph Coloring

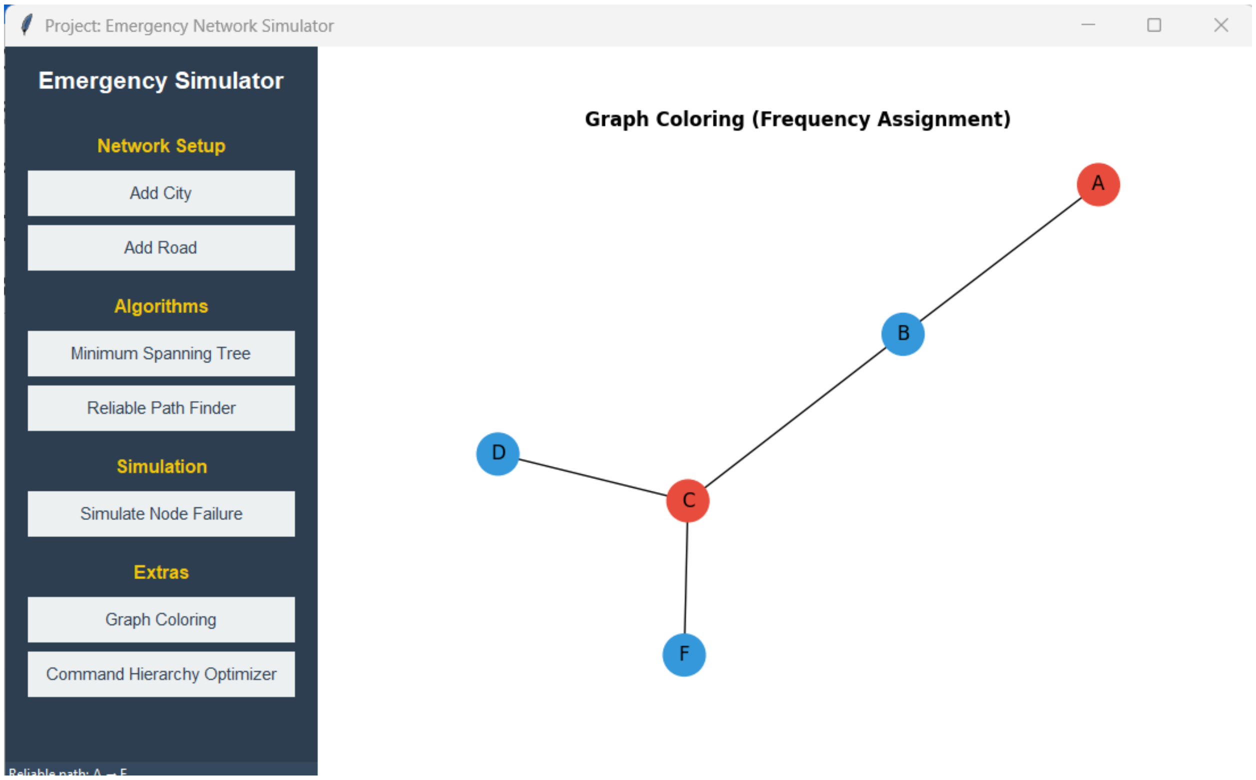


Figure 9 Graph Coloring

Backend Logic

Code:

colors = nx.coloring.greedy_color(self.graph)

This uses a greedy graph coloring algorithm.

What it does:

1. Takes one node at a time
2. Assigns the lowest available color
3. Ensures no neighbor has the same color

QUESTION NO 6

1. Based on the provided diagram map (a), build the state space.

Answer:

The delivery problem can be modeled as a state-space search problem.

1) States
   Here, each state represents the robot being located in one city.
   That means: State = Current city

   So, list of states or cities are:

   Glogow, Leszno, Wroclaw, Opole, Katowice, Czestochowa, Kalisz, Poznan, Bydgoszcz, Wloclawek, Konin, Lodz, Warsaw, Radom, Kielce, Krakow, Plock

2) Initial State:
   The robot starts at blue node:
   So, Initial State = Glogow

3) Goal State:
   The delivery destination is red node:
   So, Goal State = Plock

There will be actions (move from one city to another), transition model and path cost function associated also.

Each road has an associated distance (weight) shown in diagram (a).

The path cost is the sum of distances traveled.

Example

Cost (Glogow to Leszno to Poznan)= 45 + 90 = 135

So, State Space Structure

- The state space is a weighted, undirected graph
- Nodes = cities

- Edges = roads
- Edge weights = distances

The state space includes all the cities represented in the map. The state is defined as the robot's current city. The initial state is Glogow, and the goal state is Plock. The cost for moving from one state to another is the total distance traveled from one city to another via the roads. The transition from one state to another is deterministic, as each action will take us to a single state. An action takes us from one state to another if the cities are directly connected.

a. **Solve the above problem using a depth-first-search (DFS) algorithm. Use open and closed containers to explain the algorithm.**

b. **Solve the above problem using a breadth-first-search (BFS) algorithm. Use open and close containers to explain the algorithm.**

Answer to both questions 1)a) and 1)b) are:

What is OPEN Container and Closed Container?

OPEN Container

OPEN = nodes that are discovered but not yet explored. These are cities the robot knows about but hasn't visited yet. It is also called frontier.

CLOSED Container (Explored Set)

CLOSED = nodes/cities that are already explored or visited and will not be visited again. It prevents infinite loops. A list of all cities that were explored during the search process.

Key points

- Includes every visited city
- Order shows search behavior
- May contain cities not in final path

Why do we need OPEN and CLOSED containers?

- Avoid revisiting the same city
- Track progress of the algorithm
- Clearly explain algorithm steps
- Prevent infinite loops in graphs

Where DFS and BFS differ

Both DFS and BFS use “open” and “closed” containers, but they are not the same in how they are used. It has

- Same concept
- Different data structure
- Different order of expansion
- Different behavior and result

1. Depth First Search (DFS)

DFS Implementation Idea

- Go as deep as possible before backtracking
- Uses a stack
- Does not guarantee shortest path
- Can get stuck exploring long paths first

DFS using Open & Closed containers

Open container in DFS

- It is implemented as a STACK (LIFO)
- Last inserted node is expanded first

DFS behavior

- Goes deep along one path
- Backtracks only when no further moves are possible

DFS – How it works in algorithm

Initial state

- Open = [Glogow]
- Closed = []

Step 1

- Pop Glogow from Open
- Add to Closed
- Push its neighbors (e.g., Leszno, Wroclaw) to Open

Step 2

- Pop the *last added* city (say Leszno)
- Continue expanding deeply

DFS may reach the goal without exploring all shallow nodes.

In DFS, the open container is treated as a stack, causing the algorithm to explore one path deeply before backtracking.

2. Breadth First Search (BFS)

- Explores level by level
- Uses a queue
- Always finds the shortest path (in number of steps)
- Memory expensive

BFS using Open & Closed containers

Open container in BFS

- Implemented as a QUEUE (FIFO)
- First inserted node is expanded first

BFS behavior

- Explores level by level
- Finds the shortest path in terms of number of edges

BFS – How it works in algorithm

Initial state

- Open = [Glogow]
- Closed = []

Step 1

- Dequeue Glogow
- Add to Closed
- Enqueue neighbors (Leszno, Wroclaw)

Step 2

- Dequeue Leszno
- Then Wroclaw
- Continue level-wise expansion

BFS explores all cities at depth 1 before depth 2.

In BFS, the open container is treated as a queue, ensuring nodes are expanded in order of increasing depth.

Conclusion

Although DFS and BFS both use open and closed containers, the key difference lies in how the open container is organized. DFS uses a stack (LIFO) to explore paths deeply, while BFS uses a queue (FIFO) to explore nodes level by level. This difference results in distinct search behaviors and solution properties.

Basically, the containers are conceptually the same, but their internal ordering (stack vs queue) makes DFS and BFS fundamentally different algorithms.

2. Based on the provided diagram map (b), design the heuristic function (10 marks). Solve the above problem using the A* algorithm.

Answer:

Heuristic Function Design (Straight-line Distance)

Heuristic definition

From diagram (b), the numbers shown are straight-line (Euclidean) distances between cities.

We define the heuristic function as:

h(n)= straight line distance from city n to the goal city (Plock)

The heuristic function h(n) estimates the cost from the current node n to the goal node.

So, from diagram (b), we get heuristic values as such:

```
heuristic = {
  "Glogow": 40,
  "Leszno": 67,
  "Poznan": 108,
  "Bydgoszcz": 90,
  "Wloclawek": 44,
  "Konin": 102,
  "Kalisz": 95,
  "Lodz": 118,
  "Warsaw": 95,
  "Radom": 91,
  "Krakow":102,
  "Plock": 0

}
```

Why this heuristic is valid

- It is admissible.
  Straight-line distance never overestimates the true shortest road distance.
- It is consistent.
  Triangle inequality holds for Euclidean distances.

Therefore, A* using this heuristic is optimal.

It will be used as:

A* Evaluation Function $f(n) = g(n) + h(n)$

Where:

$g(n)$ = cost from start

$h(n)$ = heuristic estimate to goal

3. **Discuss the advantages and disadvantages of BFS, DFS and A* search algorithms using the problem you analyzed and the resulting solutions/paths as the context for your discussion.**

Answer:

Discussion: BFS, DFS and A* in the Parcel Delivery Problem

In this problem, a robot is required to deliver parcels from Glogow (start city) to Plock (goal city) using a map of Polish cities. Different search algorithms were applied to find a route, and each algorithm showed distinct strengths and weaknesses based on how it explores the state space.

1. Depth First Search (DFS)

DFS explores one path deeply before backtracking. In our problem, DFS started from Glogow and followed one branch of cities until it eventually reached Plock.

Advantages

- DFS uses less memory because it only stores the current path and a small number of unexplored nodes
- It was able to find a valid path from Glogow to Plock without needing additional information such as distances or heuristics.
- Simple to implement and easy to understand.

Disadvantages

- DFS does not guarantee the shortest path. The path it found may contain unnecessary detours.
- The algorithm may explore irrelevant cities before reaching Plock.
- In larger maps, DFS can waste time by going deep into poor routes.

Observation from our result:

DFS found a solution, but there was no guarantee that the route was optimal for parcel delivery.

2. Breadth First Search (BFS)

BFS explores all cities level by level, starting from the nearest cities to Glogow.

Advantages

- BFS guarantees the shortest path in terms of number of cities visited.
- It is reliable and systematic, ensuring that Plock is reached using the minimum number of steps.
-
- Useful when all actions have equal cost.

Disadvantages

- BFS requires large memory, as it stores all frontier nodes at each level.
- It does not consider actual road distances, only the number of edges.
- Many cities were explored even when they were not part of the final path.

Observation from our result:

BFS produced a correct and shortest-step route, but it explored many unnecessary cities, making it inefficient for large maps.

3. A* Search Algorithm

A* combines actual path cost and heuristic information to guide the search toward the goal.

In this problem, the heuristic was the straight-line distance to Plock (from diagram b).

Advantages

A* found the optimal path efficiently.

- It explored fewer cities compared to BFS because it was guided by geographical information.
- The algorithm is both complete and optimal when the heuristic is admissible.
- Very suitable for real-world navigation problems like parcel delivery.

Disadvantages

- Requires a well-designed heuristic, which may not always be available.
- More complex to implement than BFS and DFS.
- Slightly higher memory usage than DFS.

Observation from our result:

A* reached Plock faster and more efficiently than BFS and DFS by prioritizing cities closer to the goal.

Comparison: DFS vs BFS vs A*

In This Problem:

- DFS found a solution but without optimality guarantee
- BFS found shortest path but explored many nodes
- A* found the best path faster, using straight-line distances

Algorithm	Advantages	Disadvantages
DFS	Low memory usage, simple	Not optimal, may take long path
BFS	Guaranteed shortest path	High memory usage
A*	Optimal + efficient, goal-directed	Needs good heuristic

Conclusion

For the parcel delivery robot:

- DFS is not reliable
- BFS is correct but inefficient
- A* is the best choice, as it uses geographical knowledge