Kaan Tonta – 200218036

Arda Varol – 200218702

Mertcan Gürbüz-200212031

1. Introduction

1.1 Background

The increasing complexity of network infrastructures demands robust pathfinding algorithms that can optimize communication routes while adhering to specific constraints. This project aims to address this need by implementing and analyzing different algorithms for finding the shortest path in a network graph.

1.2 Development Environment

- We used python programming language.
- We used Visual Studio Code.

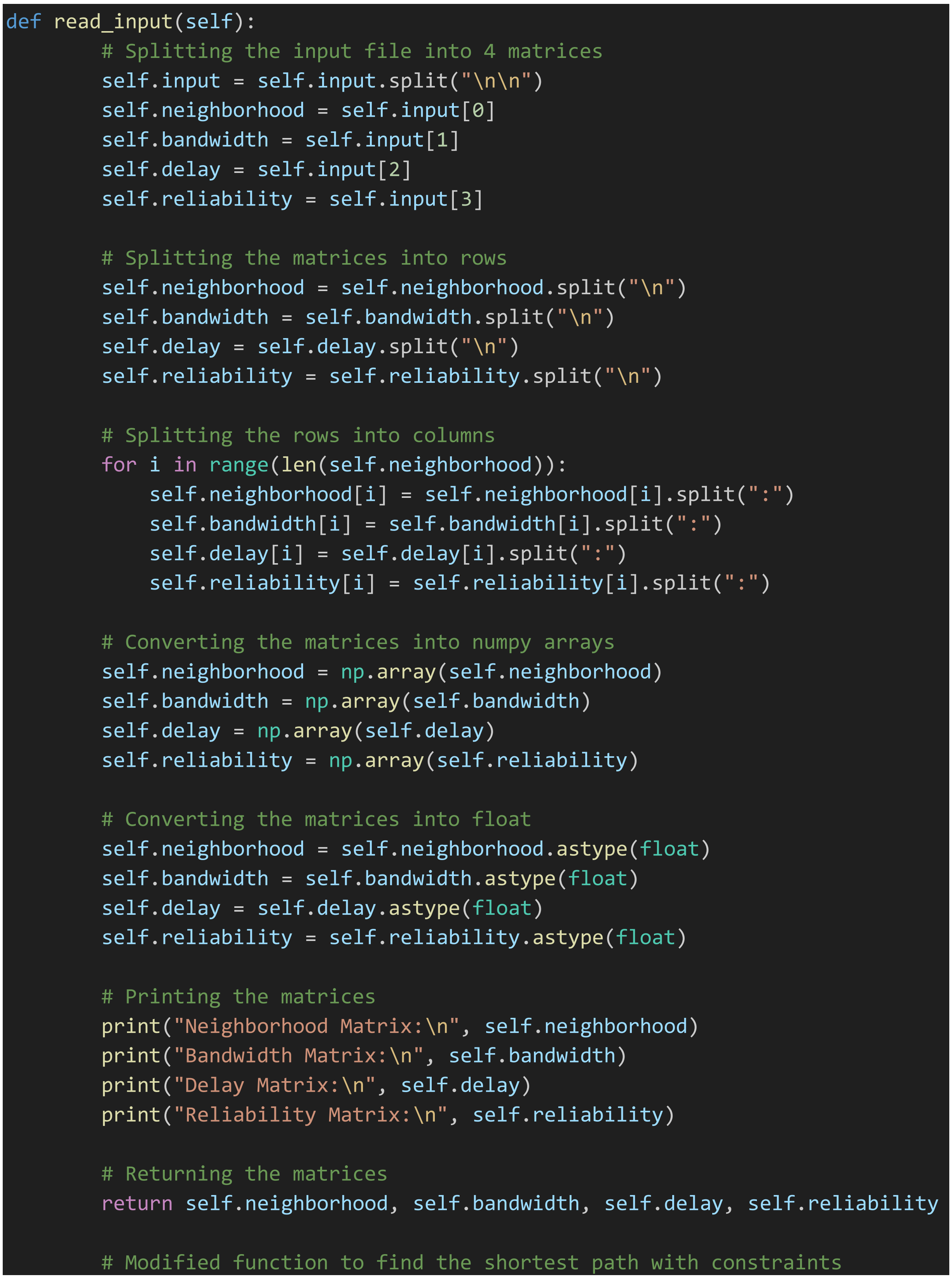
1.3 Objectives

- Implement Dijkstra's algorithm for finding the shortest path with constraints.
- Extend the algorithm to consider additional constraints such as bandwidth, delay, and reliability.
- Implement Bellman-Ford algorithm with objective functions to optimize paths.
- Explore the A* algorithm adapted for network pathfinding with constraints.

2. Methodology

2.1 Input Processing

The project starts with reading an input file containing adjacency matrices representing the network's neighborhood, bandwidth, delay, and reliability.



```python
def read_input(self):
        # Splitting the input file into 4 matrices
        self.input = self.input.split("\n\n")
        self.neighborhood = self.input[0]
        self.bandwidth = self.input[1]
        self.delay = self.input[2]
        self.reliability = self.input[3]

        # Splitting the matrices into rows
        self.neighborhood = self.neighborhood.split("\n")
        self.bandwidth = self.bandwidth.split("\n")
        self.delay = self.delay.split("\n")
        self.reliability = self.reliability.split("\n")

        # Splitting the rows into columns
        for i in range(len(self.neighborhood)):
            self.neighborhood[i] = self.neighborhood[i].split(":")
            self.bandwidth[i] = self.bandwidth[i].split(":")
            self.delay[i] = self.delay[i].split(":")
            self.reliability[i] = self.reliability[i].split(":")

        # Converting the matrices into numpy arrays
        self.neighborhood = np.array(self.neighborhood)
        self.bandwidth = np.array(self.bandwidth)
        self.delay = np.array(self.delay)
        self.reliability = np.array(self.reliability)

        # Converting the matrices into float
        self.neighborhood = self.neighborhood.astype(float)
        self.bandwidth = self.bandwidth.astype(float)
        self.delay = self.delay.astype(float)
        self.reliability = self.reliability.astype(float)

        # Printing the matrices
        print("Neighborhood Matrix:\n", self.neighborhood)
        print("Bandwidth Matrix:\n", self.bandwidth)
        print("Delay Matrix:\n", self.delay)
        print("Reliability Matrix:\n", self.reliability)

        # Returning the matrices
        return self.neighborhood, self.bandwidth, self.delay, self.reliability

        # Modified function to find the shortest path with constraints
```

2.2 Dijkstra's Algorithm

A modified version of Dijkstra's algorithm is implemented to find the shortest path while considering user-specified constraints.

```
def shortest_path_with_objective(self, source, destination, min_bandwidth,
max_delay, min_reliability,
                                     bandwidth_demand):
        # Create a graph with additional properties
        self.graph = nx.Graph()
        for i in range(len(self.neighborhood)):
            for j in range(len(self.neighborhood[i])):
                if self.neighborhood[i][j] != 0:
                    self.graph.add_edge(i, j, weight=self.neighborhood[i][j],
                                        bandwidth=self.bandwidth[i][j],
                                        delay=self.delay[i][j],
                                        reliability=self.reliability[i][j])
```

2.3 Bellman-Ford Algorithm

The Bellman-Ford algorithm is modified to include objective functions, providing optimized paths.

```
def bellman_ford_with_objective(self, source, bandwidth_demand):
        # Initialize distances and predecessors
        distances = {v: float('infinity') for v in self.graph.nodes()}
        distances[source] = 0

        # Relax edges repeatedly
        for _ in range(len(self.graph.nodes()) - 1):
            for u, v, data in self.graph.edges(data=True):
                new_cost = distances[u] + (data['weight'] * bandwidth_demand)
                if new_cost < distances[v]:
                    distances[v] = new_cost

        # Check for negative weight cycles
        for u, v, data in self.graph.edges(data=True):
            if distances[u] + (data['weight'] * bandwidth_demand) <
distances[v]:
```

```
            raise ValueError("Graph contains a negative weight cycle")

    return distances
```

2.4 A* Algorithm

The A* algorithm is adapted to incorporate constraints and objective functions for efficient pathfinding.

```
def a_star_with_objective(self, start, goal, bandwidth_demand):
    open_set = [(0, start, 0, [])]  # (f-score, node, g-score, path)
    closed_set = set()

    while open_set:
        _, current, g, path = heapq.heappop(open_set)
        path = path + [current]

        if current == goal:
            return path

        closed_set.add(current)

        for neighbor, data in self.graph[current].items():
            if neighbor in closed_set:
                continue

            g_score = g + (data['weight'] * bandwidth_demand)
            f_score = g_score + self.heuristic(neighbor, goal)

            heapq.heappush(open_set, (f_score, neighbor, g_score, path))

    return None
```

3. Results

3.1 Shortest Path with Dijkstra's Algorithm

The algorithm successfully finds the shortest path considering user-specified constraints on bandwidth, delay, and reliability.

```
Dijkstra's Algorithm Path: [0, 1, 2, 4] Distance: 8.0
```

3.2 Bellman-Ford Algorithm Results

Objective functions in the Bellman-Ford algorithm lead to optimized paths based on bandwidth demand

```
Bellman-Ford Algorithm Distances: {0: 0, 1: 15.0, 5: 5.0, 2: 30.0, 3: 55.0, 4: 40.0, 6: 20.0, 7: 25.0, 8: 20.0, 10: 15.0, 9: 30.0, 11: 20.0, 12: 35.0, 13: 45.0, 14: 35.0, 18: 25.0, 15: 30.0, 16: 50.0, 17: 70.0, 19: 40.0, 20: 45.0, 21: 40.0, 22: 55.0, 23: 70.0}
```

3 A* Algorithm Results

The A* algorithm efficiently finds paths by considering constraints and objective functions.

```
A* Algorithm Path: [0, 1, 2, 4]
```

4. Conclusion

In conclusion, the implemented algorithms successfully address the challenge of network pathfinding with constraints. The adaptability of Dijkstra's algorithm, Bellman-Ford algorithm, and A* algorithm provides flexibility in handling different scenarios and optimizing communication routes in network graphs.

```
Shortest path from node 0 to node 4: [0, 1, 2, 4]
Total distance: 3.0
```