FAST NUCES
KARACHI

02/12/2024

GT Project Report

Presented By:
22K-4172 Abdul Wasey
22K-4141 Saad Arshad
22K-4523 Muhammad Hamza

Shortest Path Algorithms and Steiner Tree Optimization for Emergency Response

PREPARED FOR:

Miss Urooj

Abstract

This report evaluates the implementation of Dijkstra's and Steiner Tree algorithms in C# and Python for solving shortest path and network optimization problems in emergency response scenarios. The efficiency analysis compares execution times and memory usage across varying grid sizes and targets, providing insights into the strengths and weaknesses of each language. The findings offer a practical guide for choosing the appropriate language for similar computational tasks.

Introduction

- ## Problem Statement:

  Efficient path planning is critical in emergencies to minimize response time and fuel consumption. This project focuses on implementing graph algorithms to optimize paths between multiple emergency sites.

- ## Objectives:

  - Implement Dijkstra's algorithm for shortest paths.
  - Implement the Steiner tree algorithm for multi-target optimization.
  - Analyze performance in terms of execution time and memory usage.

- ## Overview:

  The report details the algorithmic implementations, test scenarios, efficiency analysis, and insights into the suitability of C# and Python for the problem.

Methodology

- ## Graph Algorithms:

  - **Dijkstra's Algorithm**: Finds the shortest path from a source node to one or more target nodes in a weighted graph. Prioritize paths with the least cumulative weight using a priority queue for efficiency.
  - **Steiner Tree Algorithm**: Connects a subset of nodes (called terminals or targets) in a graph with the shortest possible combined edge weight. Aims to optimize routes when multiple targets need to be visited.

- ## Implementation:

  - A graph is considered with nodes representing locations and edges connecting those locations with weights as the distance between those locations.
  - Both C# and Python implements Dijkstra's algorithm to find the shortest paths from the starting node to all the target nodes and implements the Steiner tree approximation after constructing the minimum spanning tree.
  - The city grid is read from one of the 5 files randomly generated beforehand. The format of the input files is as follows:
    - Grid Dimension: Specifies the grid dimension (e.g., 5 5 indicates 5x5 grid).
    - Graph edges with weights: X1,Y1 X2,Y2 Weight (e.g., 0,0 1,1 2 indicates the edge between coordinates 0,0 and 1,1 is weighted 2).
    - START: Identifies the starting point (e.g., START 2,4 indicates that 2,4 is the starting point).

- TARGETS: Identifies the number of target locations and their coordinates (e.g., TARGETS 2 0,1 4,4 indicates that there are 2 targets, and their coordinates are 0,1 and 4,4)

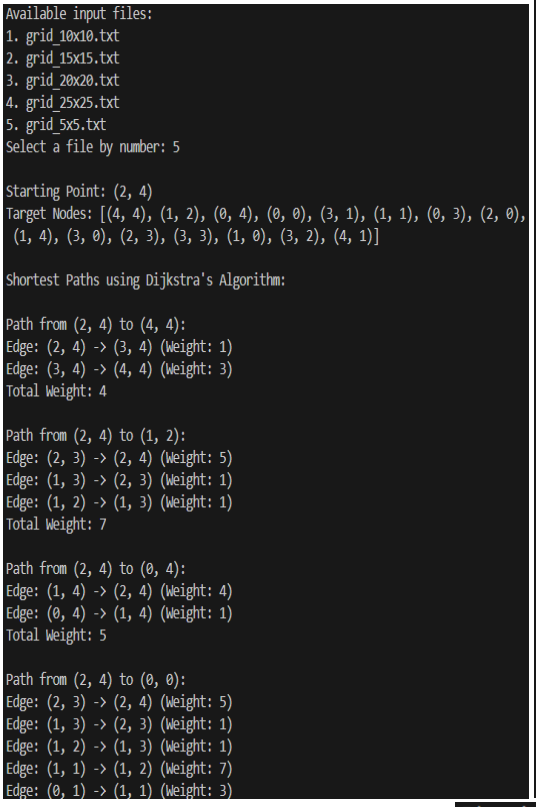
- Performance Metrics:
  - Execution time: measured in milliseconds(ms). Indicates the time taken to execute the algorithms for a specific input size and target count.
  - Memory Usage: measured in bytes. Records the memory allocated during the execution of the algorithms.
  - Example Metrics Table:

Metric	C#	Python
5x5 Grid	129 ms, 1,367,984 bytes	282 ms, 155,648 bytes
10x10 Grid	159 ms, 769,608 bytes	74 ms, 270,336 bytes

Test Cases and Results:

Test Case 1: 5x5 grid with 15 targets.

Result of C#:



Result of Python:

```
Available input files:
1. grid_10x10.txt
2. grid_15x15.txt
3. grid_20x20.txt
4. grid_25x25.txt
5. grid_5x5.txt
Select a file by number: 5

Starting Point: (2, 4)
Target Nodes: [(4, 4), (1, 2), (0, 4), (0, 0), (3, 1), (1, 1), (0, 3), (2, 0),
 (1, 4), (3, 0), (2, 3), (3, 3), (1, 0), (3, 2), (4, 1)]

Shortest Paths using Dijkstra's Algorithm:
```

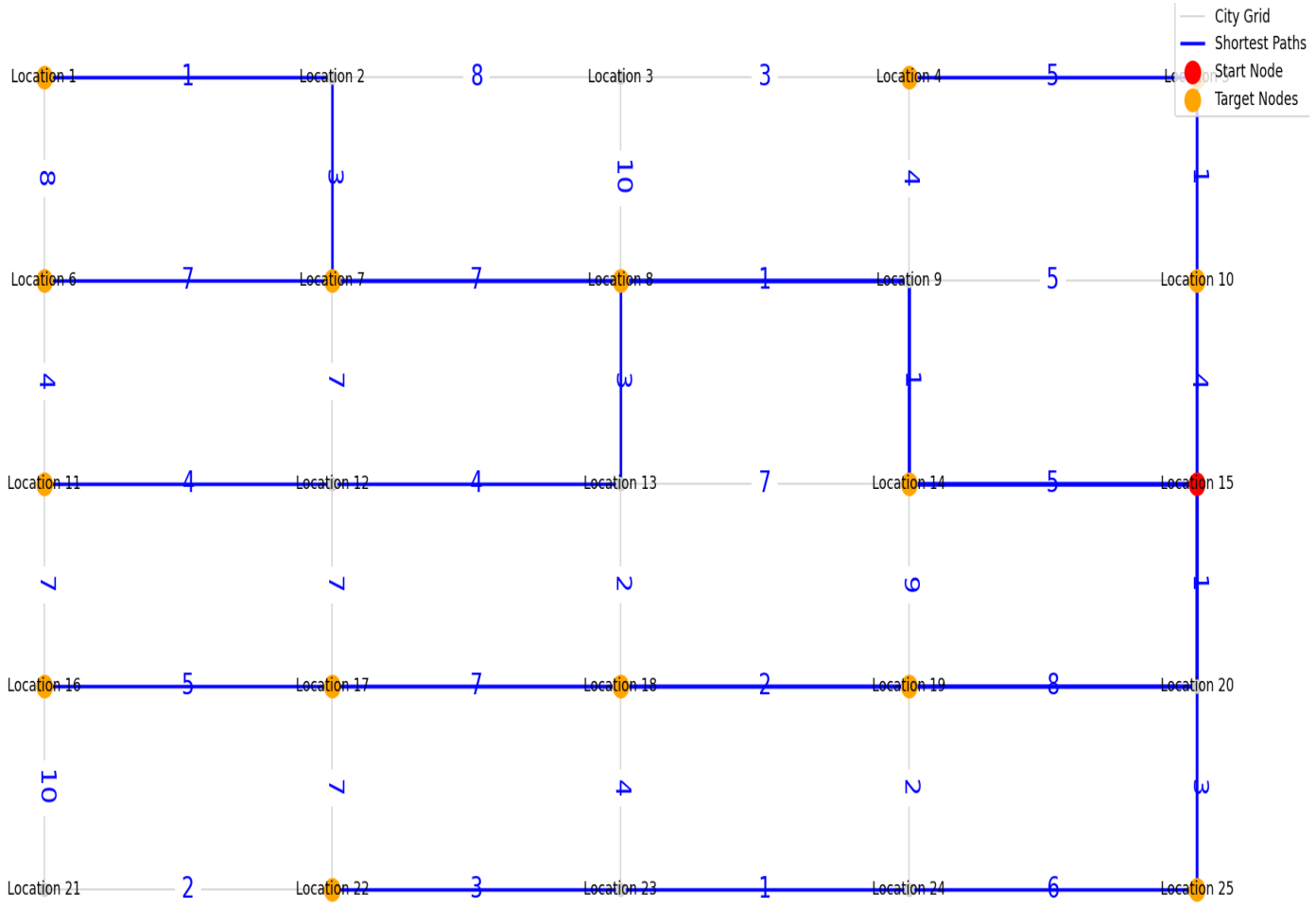
```
Shortest Paths using Dijkstra's Algorithm:

Path from (2, 4) to (4, 4):
Edge: (2, 4) -> (3, 4) (Weight: 1), Accumulated Total Weight: 1
Edge: (3, 4) -> (4, 4) (Weight: 3), Accumulated Total Weight: 4

Path from (2, 4) to (1, 2):
Edge: (2, 4) -> (2, 3) (Weight: 5), Accumulated Total Weight: 5
Edge: (2, 3) -> (1, 3) (Weight: 1), Accumulated Total Weight: 6
Edge: (1, 3) -> (1, 2) (Weight: 1), Accumulated Total Weight: 7

Path from (2, 4) to (0, 4):
Edge: (2, 4) -> (1, 4) (Weight: 4), Accumulated Total Weight: 4
Edge: (1, 4) -> (0, 4) (Weight: 1), Accumulated Total Weight: 5

Path from (2, 4) to (0, 0):
Edge: (2, 4) -> (2, 3) (Weight: 5), Accumulated Total Weight: 5
Edge: (2, 3) -> (1, 3) (Weight: 1), Accumulated Total Weight: 6
Edge: (1, 3) -> (1, 2) (Weight: 1), Accumulated Total Weight: 7
Edge: (1, 2) -> (1, 1) (Weight: 7), Accumulated Total Weight: 14
Edge: (1, 1) -> (0, 1) (Weight: 3), Accumulated Total Weight: 17
Edge: (0, 1) -> (0, 0) (Weight: 1), Accumulated Total Weight: 18

Path from (2, 4) to (3, 1):
Edge: (2, 4) -> (3, 4) (Weight: 1), Accumulated Total Weight: 1
Edge: (3, 4) -> (3, 3) (Weight: 8), Accumulated Total Weight: 9
Edge: (3, 3) -> (3, 2) (Weight: 2), Accumulated Total Weight: 11
Edge: (3, 2) -> (3, 1) (Weight: 7), Accumulated Total Weight: 18

Path from (2, 4) to (1, 1):
Edge: (2, 4) -> (2, 3) (Weight: 5), Accumulated Total Weight: 5
Edge: (2, 3) -> (1, 3) (Weight: 1), Accumulated Total Weight: 6
Edge: (1, 3) -> (1, 2) (Weight: 1), Accumulated Total Weight: 7
Edge: (1, 2) -> (1, 1) (Weight: 7), Accumulated Total Weight: 14
```

```
Path from (2, 4) to (0, 3):
Edge: (2, 4) -> (1, 4) (Weight: 4), Accumulated Total Weight: 4
Edge: (1, 4) -> (0, 4) (Weight: 1), Accumulated Total Weight: 5
Edge: (0, 4) -> (0, 3) (Weight: 5), Accumulated Total Weight: 10

Path from (2, 4) to (2, 0):
Edge: (2, 4) -> (2, 3) (Weight: 5), Accumulated Total Weight: 5
Edge: (2, 3) -> (1, 3) (Weight: 1), Accumulated Total Weight: 6
Edge: (1, 3) -> (1, 2) (Weight: 1), Accumulated Total Weight: 7
Edge: (1, 2) -> (2, 2) (Weight: 3), Accumulated Total Weight: 10
Edge: (2, 2) -> (2, 1) (Weight: 4), Accumulated Total Weight: 14
Edge: (2, 1) -> (2, 0) (Weight: 4), Accumulated Total Weight: 18

Path from (2, 4) to (1, 4):
Edge: (2, 4) -> (1, 4) (Weight: 4), Accumulated Total Weight: 4

Path from (2, 4) to (3, 0):
Edge: (2, 4) -> (3, 4) (Weight: 1), Accumulated Total Weight: 1
Edge: (3, 4) -> (3, 3) (Weight: 8), Accumulated Total Weight: 9
Edge: (3, 3) -> (3, 2) (Weight: 2), Accumulated Total Weight: 11
Edge: (3, 2) -> (3, 1) (Weight: 7), Accumulated Total Weight: 18
Edge: (3, 1) -> (3, 0) (Weight: 5), Accumulated Total Weight: 23

Path from (2, 4) to (2, 3):
Edge: (2, 4) -> (2, 3) (Weight: 5), Accumulated Total Weight: 5

Path from (2, 4) to (3, 3):
Edge: (2, 4) -> (3, 4) (Weight: 1), Accumulated Total Weight: 1
Edge: (3, 4) -> (3, 3) (Weight: 8), Accumulated Total Weight: 9
```

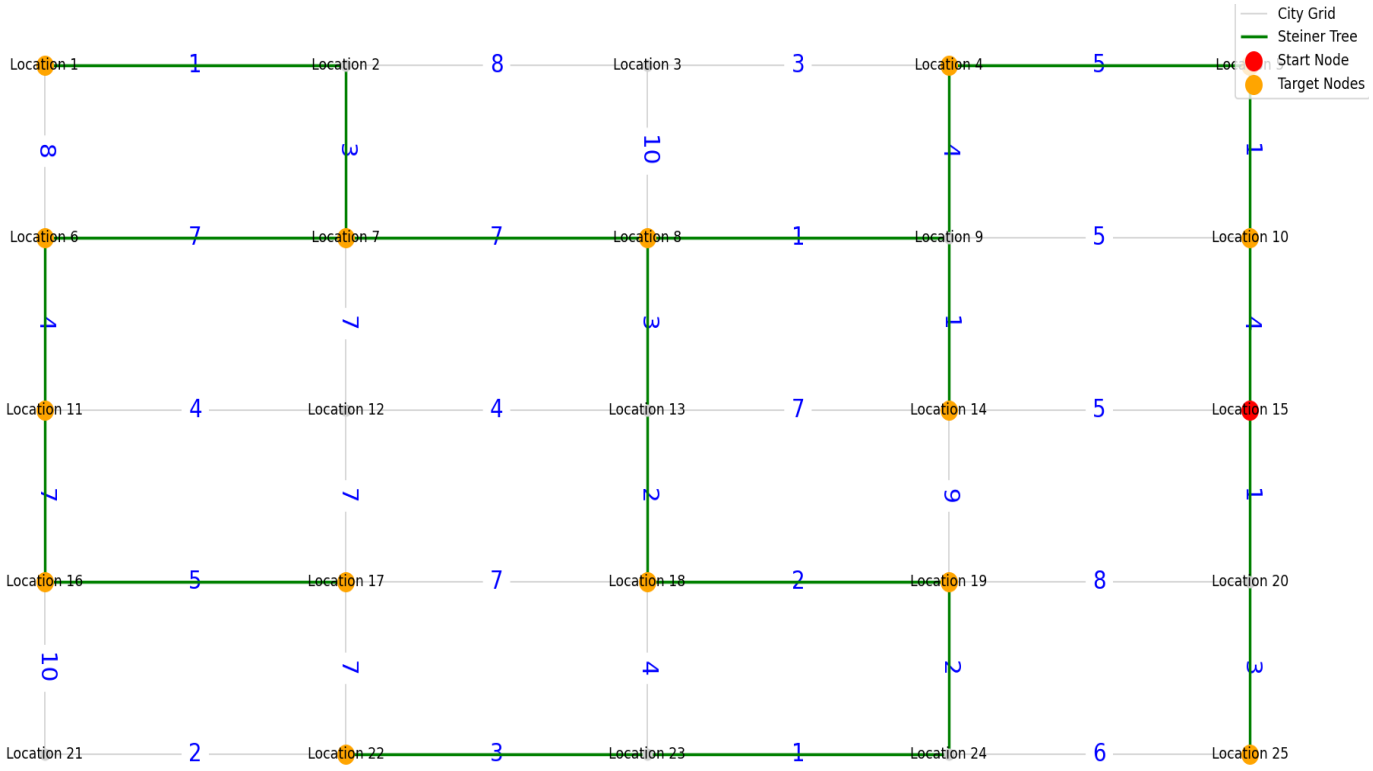
```
Path from (2, 4) to (1, 0):
Edge: (2, 4) -> (2, 3) (Weight: 5), Accumulated Total Weight: 5
Edge: (2, 3) -> (1, 3) (Weight: 1), Accumulated Total Weight: 6
Edge: (1, 3) -> (1, 2) (Weight: 1), Accumulated Total Weight: 7
Edge: (1, 2) -> (1, 1) (Weight: 7), Accumulated Total Weight: 14
Edge: (1, 1) -> (1, 0) (Weight: 7), Accumulated Total Weight: 21

Path from (2, 4) to (3, 2):
Edge: (2, 4) -> (3, 4) (Weight: 1), Accumulated Total Weight: 1
Edge: (3, 4) -> (3, 3) (Weight: 8), Accumulated Total Weight: 9
Edge: (3, 3) -> (3, 2) (Weight: 2), Accumulated Total Weight: 11

Path from (2, 4) to (4, 1):
Edge: (2, 4) -> (3, 4) (Weight: 1), Accumulated Total Weight: 1
Edge: (3, 4) -> (4, 4) (Weight: 3), Accumulated Total Weight: 4
Edge: (4, 4) -> (4, 3) (Weight: 6), Accumulated Total Weight: 10
Edge: (4, 3) -> (4, 2) (Weight: 1), Accumulated Total Weight: 11
Edge: (4, 2) -> (4, 1) (Weight: 3), Accumulated Total Weight: 14
```

```
Edges in the Steiner Tree:
Edge: (2, 4) -> (1, 4) (Weight: 4), Accumulated Total Weight: 4
Edge: (2, 4) -> (3, 4) (Weight: 1), Accumulated Total Weight: 5
Edge: (3, 4) -> (4, 4) (Weight: 3), Accumulated Total Weight: 8
Edge: (1, 4) -> (0, 4) (Weight: 1), Accumulated Total Weight: 9
Edge: (0, 4) -> (0, 3) (Weight: 5), Accumulated Total Weight: 14
Edge: (0, 3) -> (1, 3) (Weight: 4), Accumulated Total Weight: 18
Edge: (1, 3) -> (1, 2) (Weight: 1), Accumulated Total Weight: 19
Edge: (1, 3) -> (2, 3) (Weight: 1), Accumulated Total Weight: 20
Edge: (1, 2) -> (2, 2) (Weight: 3), Accumulated Total Weight: 23
Edge: (2, 2) -> (3, 2) (Weight: 2), Accumulated Total Weight: 25
Edge: (1, 2) -> (1, 1) (Weight: 7), Accumulated Total Weight: 32
Edge: (3, 2) -> (3, 3) (Weight: 2), Accumulated Total Weight: 34
Edge: (3, 3) -> (4, 3) (Weight: 2), Accumulated Total Weight: 36
Edge: (4, 3) -> (4, 2) (Weight: 1), Accumulated Total Weight: 37
Edge: (4, 2) -> (4, 1) (Weight: 3), Accumulated Total Weight: 40
Edge: (1, 1) -> (0, 1) (Weight: 3), Accumulated Total Weight: 43
Edge: (0, 1) -> (0, 0) (Weight: 1), Accumulated Total Weight: 44
Edge: (1, 1) -> (1, 0) (Weight: 7), Accumulated Total Weight: 51
Edge: (1, 0) -> (2, 0) (Weight: 4), Accumulated Total Weight: 55
Edge: (2, 0) -> (3, 0) (Weight: 7), Accumulated Total Weight: 62
Edge: (3, 0) -> (3, 1) (Weight: 5), Accumulated Total Weight: 67

Execution Time: 37.2748374938964 84 ms


Memory Usage: 159744 bytes
```

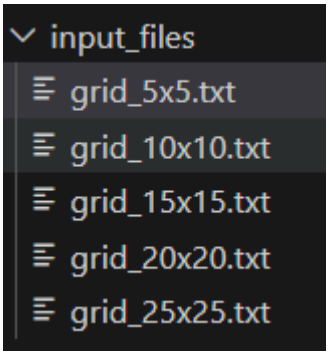
Shortest Path Visualization:



Steiner Tree Visualization:



Other Test Cases:



Efficiency Analysis:

Execution Time Comparison:

Grid Size (Targets)	C# Execution Time (ms)	Python Execution Time (ms)	Winner
5x5 (15 targets)	129	282.25	**C# (faster)**
10x10 (9 targets)	159	74.82	**Python (faster)**
15x15 (14 targets)	472	292.22	**Python (faster)**
20x20 (6 targets)	264	192.48	**Python (faster)**
25x25 (7 targets)	324	228.86	**Python (faster)**

Observations on Execution Time:

- C# performs better on small grid sizes with many targets (e.g., the 5x5 grid).
- Python outperforms C# for larger grid sizes with fewer targets, suggesting better handling of certain data manipulations and optimizations.
- Python's overhead in smaller grids might be due to its interpreted nature, while its libraries (e.g., NumPy) excel in optimized operations for larger, complex data structures.

Memory Usage Comparison:

Grid Size (Targets)	C# Memory Usage (bytes)	Python Memory Usage (bytes)	Winner
5x5 (15 targets)	1,367,984	155,648	Python(efficient)
10x10 (9 targets)	769,608	270,336	Python(efficient)
15x15 (14 targets)	1,011,000	581,632	Python(efficient)
20x20 (6 targets)	1,003,096	831,488	Python(efficient)
25x25 (7 targets)	1,203,592	1,404,928	C# (efficient)

Observations on Memory Usage:

- Python consistently uses less memory for smaller grid sizes with fewer targets.
- C# overtakes Python in efficiency for larger grids with increased complexity, likely due to its static typing and memory management mechanisms.
- Python's increased memory usage on larger grids may result from dynamic typing and object-oriented overhead.

Conclusion:

The analysis demonstrates that while Python is more suitable for quick development and academic use, C# provides better performance and memory efficiency for production-scale applications. The choice of language should align with the project's requirements, emphasizing speed or flexibility as needed.

THANK YOU!