Hard Transportation Hub

Date: 2025/04/21

Table of Contents

Chapter 1: Introduction

1.1. Problem Description

Given a map of cities connected by bidirectional roads, each with a positive length, we are tasked with identifying **transportation hubs** for a number of city-to-city queries. A city is considered a transportation hub if it appears on **at least k different shortest paths** between a given pair of source and destination cities — excluding the source and destination themselves.

- All roads are bidirectional and unique.
- Only intermediate cities (excluding source and destination) are considered when counting transportation hubs.
- The shortest path between a city and itself is defined to be 0.

1.2. Input Format

The input consists of the following:

1. Three integers:
   - `n`: the number of cities (indexed from 0 to n−1), where 3 ≤ n ≤ 500.
   - `m`: the number of bidirectional roads.
   - `k`: the minimum number of shortest paths on which a city must appear to be considered a transportation hub (2 ≤ k ≤ 5).

2. Followed by `m` lines, each describing a road in the form:

   ```
   c1 c2 length
   ```

   - `c1` and `c2` are the city indices.
   - `length` is the positive integer length of the road (≤ 100).

3. A positive integer `T` (≤ 500), the number of source-destination queries.

4. `T` lines follow, each containing a pair of city indices:

   ```
   source destination
   ```

1.3. Output Format

For each query:

- Output the indices of all transportation hubs, sorted in ascending order, separated by single spaces.
- If no city qualifies as a hub, output `None`.

1.4. Algorithm Background

To solve the transportation hub problem, we employ **Dijkstra's algorithm**, a well-known method for finding the shortest paths from a single source to all other nodes in a weighted graph with non-negative edge weights.

This algorithm ensures that we can efficiently compute the shortest distance between any pair of cities. By slightly extending Dijkstra's algorithm, we can also keep track of how many shortest paths pass through each city. This information is critical for identifying the cities that qualify as transportation hubs.

Due to its efficiency and accuracy in sparse graphs, Dijkstra's algorithm is well-suited for this problem.

Chapter 2: Algorithm Specification

Before sketching the main program flow, I will first introduce the main steps and algorithms in this project, which include the build of the graph, the Dijkstra algorithm, and the counting of shortest paths passing each node.

2.1. The Build of the Graph

> ### The Graph Data Structure
>
> A **graph** is a fundamental data structure consisting of:
>
> - Core Components
>   - ‣ **Vertices (Nodes)**: Represent entities/objects
>   - ‣ **Edges**: Represent relationships between vertices
>   - May be **directed** or **undirected**
>   - May have **weights** (numerical values)
>
> - Common Representations
>
>   1. **Adjacency Matrix**
>   - ‣ Square matrix where entry (i,j) represents edge between vertex i and j
>   - ‣ Efficient for dense graphs
>   - ‣ O(1) edge lookup time
>
>   2. **Adjacency List**
>   - ‣ Array of linked lists
>   - ‣ Each list stores neighbors of a vertex
>   - ‣ Memory-efficient for sparse graphs
>
> Considering that the maximum number of nodes is 500, we choose the adjacency matrix representation for simplicity and efficiency.

To represent the structure of cities and roads, the program uses an adjacency matrix to build the graph. The class `Graph` encapsulates this logic.

The graph is initialized with a fixed-size two-dimensional array `adj`, where `adj[i][j]` stores the length of the road between city `i` and city `j`. The matrix is symmetric since all roads are undirected.

At the beginning, all distances are set to `-1` to represent the absence of a road, except for the diagonal entries where `adj[i][i] = 0`, indicating zero distance from a city to itself.

Edges are then added by reading input from the standard input. Each road is defined by two city indices and a positive length. The `add_edge` function updates the adjacency matrix symmetrically to reflect the bidirectional nature of roads.

This setup allows for efficient distance lookups between any two cities, which is crucial for the shortest path calculations later in the algorithm.

```
1  function BuildGraph(n: int, m: int)
2  │  for i in 0 to n - 1
3  │  │  for j in 0 to n - 1
4  │  │  │  if i == j then
5  │  │  │  │  adj[i][j] ← 0
6  │  │  │  else
7  │  │  │  │  adj[i][j] ← −1
8  │  │  │  end
9  │  │  end
10 │  for count in 1 to m
11 │  │  read (u, v, length)
12 │  │  adj[u][v] ← length
13 │  │  adj[v][u] ← length
14 │  end
15 end
```

2.2. The Dijkstra Algorithm

To compute the shortest paths from a given source city to all others, the program uses a customized implementation of **Dijkstra's algorithm**. It also tracks multiple shortest paths by storing all valid predecessors for each node.

The algorithm starts by initializing the distance from the source to itself as 0, while all other distances are set to -1, indicating they are initially unreachable. Then, for each unvisited node, it repeatedly selects the one with the minimum known path length and updates its neighbors accordingly.

Unlike the classical Dijkstra, this version maintains a list of **predecessor nodes** for every city, which is essential for counting how many shortest paths pass through a given node.

Below is the core pseudocode that captures the algorithm's main logic:

```
1  function Dijkstra(start: int)
2  │  for each node i
3  │  │  visited[i] ← false
4  │  │  path_length[i] ← −1
5  │  path_length[start] ← 0
6  │  for each neighbor i of start
```

```
7   │ │   path_length[i] ← weight(start, i)
8   │ │   path_parent[i] ← [start]
9   │   visited[start] ← true
10  │   repeat n times
11  │ │   min_node ← node with smallest path_length among unvisited
12  │ │   if min_node == −1 then break
13  │ │   visited[min_node] ← true
14  │ │   for each neighbor j of min_node
15  │ │ │   if not visited and edge exists
16  │ │ │ │   new_length ← path_length[min_node] + weight(min_node, j)
17  │ │ │ │   if path_length[j] == −1 or new_length < path_length[j] then
18  │ │ │ │ │   path_length[j] ← new_length
19  │ │ │ │ │   path_parent[j] ← [min_node]
20  │ │ │ │   else if new_length == path_length[j] then
21  │ │ │ │ │   path_parent[j].append(min_node)
22  │ │ │ │   end
23  │ │ │   end
24  │ │   end
25  │   end
```

This modified Dijkstra lays the foundation for counting shortest paths passing each node and identifying transportation hubs.

2.3. Counting Shortest Paths Passing Each Node

To determine how many shortest paths pass through each node, the algorithm employs a two-phase dynamic programming approach that leverages the predecessor relationships captured during the Dijkstra traversal.

The core idea is to decompose any shortest path into two segments: from the source to the intermediate node, and from the intermediate node to the destination. The total number of paths passing through a node is then the product of these two independent counts.

The process can be summarized as follows:

1. Reverse Predecessor Graph: Construct a reverse graph where edges point from child nodes to their predecessors in the original shortest path tree.

2. Path Counting via Backtracking:

   - For each node, recursively count paths from the source using the original predecessor list.

- For each node, recursively count paths to the destination using the reverse predecessor graph.

3. Combination via Multiplication: Multiply the two directional counts to obtain the total paths passing through each node.

```
1  function CountPaths(source, destination)
2    for each node i
3    │ path_to_start[i] ← 0
4    path_to_start[source] ← 1
5    for nodes in Dijkstra order
6    │ │ for each predecessor p in path_parent[i] ▪ path_to_start[i] +=
       │ │   path_to_start[p]
7    │ │ end
8    │ end
9    reverse_graph ← empty adjacency list
10   for each node i
11   │ │ for each predecessor p in path_parent[i] ▪ add i to reverse_graph[p]
12   │ │ end
13   │ end
14   for each node i
15   │ path_to_dest[i] ← 0
16   path_to_dest[destination] ← 1
17   for nodes in reverse Dijkstra order
18   │ │ for each child c in reverse_graph[i] ▪ path_to_dest[i] += path_to_dest[c]
19   │ │ end
20   │ end
21   for each node j
22   │ passing_count[j] ← path_to_start[j] × path_to_dest[j]
23   end
```

This approach efficiently reuses shortest path information while avoiding explicit enumeration of all possible paths through memoization and topological ordering.

2.4. The Main Program Flow

The main function controls the complete execution: from reading input and initializing the graph, to handling multiple test cases and printing the required results. It ensures that each query is handled independently with proper resource management.

The core logic follows this sequence:

```
1  function Main()
2  |  Read graph parameters (n, m, k)
3  |  Initialize graph structure
4  |  Read T test cases
5  |  for each test case
6  |  |  Read (start, destination) pair
7  |  |  Initialize shortest path solver with source node
8  |  |  Execute Dijkstra's algorithm
9  |  |  Calculate path counts using bidirectional dynamic programming
10 |  |  Traverse all nodes to check path passing conditions ▪ if node is neither start nor
   |  |     destination ▪ and path_count ≥ threshold k ▪ Add to output list
11 |  |  end
12 |  |  if no qualifying nodes → Output "None"
13 |  |  else → Output space-separated node IDs
14 |  end
```

Chapter 3: Testing Results

3.1. Testing Infrastructure

- Test environment: Windows 11 64-bit and Debian 12 64-bit.
- Main program compiled with `g++ 14.2.0` using `-O2` optimization flag.

3.2. Testcases

3.2.1. Testcase 1: Testcase in the problem description

The input and output are provided in the problem description.

Input	Output
10 16 2 1 2 1 1 3 1 1 4 2 2 4 1 2 5 2 3 4 1 3 0 1 4 5 1 4 6 2 5 6 1 7 3 2 7 8 1 7 0 3 8 9 1 9 0 2 0 6 2 3 1 6 7 0 5 5	2 3 4 5 None None

3.2.2. Testcase 2: Unconnected Graph

The input contains a graph with unconnected components. The program should handle this gracefully and return `None` for any queries involving unconnected nodes.

Input	Output
5 2 2 1 2 1 3 4 1	None

| ```
1
1 3
``` | |

3.2.3. Testcase 3: The start node is the same as the end node

The input contains a query where the start and end nodes are the same. The program should return `None` since no transportation hub can exist in this case.

Input	Output
| ```
3 2 1
0 1 1
1 2 1
1
0 0
``` | ```
None
``` |

3.2.4. Testcase 4: Extremely large number of shortest paths

The input contains a graph with a very high number of shortest paths (larger than INT_MAX) passing through a single node. The program should efficiently count and return the correct transportation hubs.

Input	Output
| ```
132 260 5
0 1 1
0 2 1
1 3 1
1 4 1
2 3 1
2 4 1
...
128 130 1
129 131 1
130 131 1
1
0 131

(for complete sample, see:
code/test_sample/large_number_test.in)
``` | ```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81
82 83 84 85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100 101 102 103 104 105
106 107 108 109 110 111 112 113 114 115
116 117 118 119 120 121 122 123 124 125
126 127 128 129 130
``` |

3.2.5. Testcase 5: High pressure random graph

The input contains a large random graph with many nodes and edges generated by a python generator script.

The testcase generation algorithm is as follows:

1 generate random number `n` in [3, 500]

2 calculate `max_roads = n * (n - 1) / 2`

3 generate random number `m` in [1, max_roads]

4 generate random number `k` in [1, 5]

5 initialize node list from 0 to n-1

6 shuffle node list for randomness

7 initialize empty set `edges` and parent array for union-find

8 **define** function `find(u)`:

9   **while** `parent[u] != u`

10 | perform path compression and move up

11   **end**

12   **return** root of `u`

13 **for** i = 1 to n-1 **do**

14   connect node[i-1] and node[i] to ensure connectivity

15   add edge to `edges`

16 **while** number of edges < m

17   randomly select c1 and c2 from nodes

18   **if** c1 ≠ c2 and (c1, c2) not in `edges`

19 | add edge (sorted) to `edges`

20 **for each** edge in `edges`

21   assign a random weight in [1, 30]

22   store as (node1, node2, length)

23 shuffle all roads

24 generate T in [1, 500]

25 **repeat** T times

26   randomly select `src` and `dst`

27   store as a query pair

28 output n, m, k

29 output all roads

30 output T

31 output all queries

The testcase are incredibly large, so the input and output are not shown here. You can find them in the `code/test_sample/random_*.in` and `code/test_sample/random_*.out` files.

The program passes all the test cases shown above with high performance, including the random ones.

Chapter 4: Analysis and Comments

We only analyze the complexity of the Dijkstra algorithm and the counting of shortest paths passing each node, as they are the most time-consuming parts of the program.

4.1. The Dijkstra Algorithm

4.1.1. Time Complexity

The time complexity of the provided Dijkstra's algorithm implementation is $O(V^2)$, where V is the number of nodes in the graph. This is because:

1. The algorithm uses two nested loops over all nodes (V iterations each).
2. For each node, it scans all other nodes to find the unvisited node with the smallest path length (O(V) per iteration).
3. Updating neighbors also iterates over all nodes (O(V) per iteration), leading to a total of $O(V^2)$ operations.

4.1.2. Space Complexity

The space complexity is O(V²) due to:

1. The adjacency matrix `graph->adj` requiring $O(V^2)$ space.
2. The `path_parent` array storing a list of parents for each node, which could reach $O(V^2)$ in the worst case (e.g., multiple shortest paths).
3. Additional arrays like `path_length` and `visited` using $O(V)$ space.

The adjacency matrix dominates the space usage, making the overall complexity quadratic in the number of nodes.

4.1.3. Improvements

- Using a priority queue to optimize the selection of the next node to visit can reduce the time complexity to O((V + E) log V), where E is the number of edges.

4.2. Counting Shortest Paths Passing Each Node

4.2.1. Time Complexity

The overall algorithm counts the number of shortest paths passing through each node by computing:

1. The number of shortest paths from each node to the `start` node.
2. The number of shortest paths from each node to the `destination` node (via reverse traversal).
3. The final count of paths passing through a node as the product of the above two quantities.

The main operations can be broken down as follows:

1. **Recursive Path Counting (Forward and Reverse)**: Both `count_path_to_start` and `count_path_to_destination` use memoized depth-first search (DFS) to traverse the graph and count the number of paths. Each recursive traversal explores the directed acyclic graph formed by the shortest paths (stored in `path_parent`).
   - In the worst case, if the graph has E edges, and all of them are part of the shortest path DAG, then each recursive function visits each edge at most once.
   - Thus, the complexity of each counting phase is $O(V + E)$, where V is the number of nodes and E is the number of edges on shortest paths.

2. **Reverse Path Construction**: The function `get_path_parent_reverse` builds a reverse graph of shortest paths using a DFS traversal over `path_parent`. Each edge is traversed at most once, leading to a complexity of $O(V + E)$.

3. **Final Path Count Calculation**: After computing the forward and reverse path counts, the algorithm computes the number of paths passing through each node by multiplying the two counts for every node. This step takes $O(V)$ time.

Total Time Complexity: Combining all parts, the total time complexity is: $O(V + E)$

If the graph is dense and nearly all edges lie on shortest paths, the worst-case time complexity becomes: $O(V^2)$

4.2.2. Space Complexity

1. **Shortest Path DAGs** (`path_parent` and `path_parent_reverse`): These store parent relationships for each node along shortest paths. In the worst case (e.g., multiple shortest paths between many pairs), each node may have up to $O(V)$ parents, leading to $O(V^2)$ space usage.

2. **Path Count Arrays**: Arrays such as `count_path`, `visited`, and the final product array `count_path_passing_node` each require $O(V)$ space.

3. **Call Stack (Recursive DFS)**: The recursion depth is at most $O(V)$, so stack space usage is also bounded by $O(V)$.

Total Space Complexity: $O(V^2)$

4.2.3. Improvements

- Early termination when a threshold is exceeded helps mitigate performance degradation in cases with combinatorially many shortest paths.
- Path counting to start node and the reverse process can be executed in the same loop.

Appendix: Source Code (in C)

The project is written in C++ and employs relatively abstract object-oriented encapsulation, which may make it somewhat obscure.

Even though the project is small, it uses CMake for management.

5.1. CMakeLists.txt

`CMakeLists.txt` is the core configuration file for the CMake build system, defining project structure, compilation options, dependencies, and cross-platform build rules to generate native build environments (e.g., Makefiles or Visual Studio projects).

File: CMakeLists.txt

```cmake
# Set the minimum required version of CMake for this project
cmake_minimum_required(VERSION 3.10)

# Define the project name
project(FDS25_Project3)

# Include directories for header files
include_directories(.)

# Add executable targets
# Create an executable named 'main' from main.cpp and Graph.cpp
add_executable(main main.cpp Graph.cpp)
# Create an executable named 'test' from test.cpp and Graph.cpp
add_executable(test test.cpp Graph.cpp)
```

5.2. Graph.cpp / Graph.h

This module provides functionality for representing and analyzing graphs. It includes a `Graph` class to store graph data (nodes and weighted edges) and a `GraphShortestPathSolution` class to compute and count shortest paths from a given start node to any destination node in the graph. The implementation supports graphs with up to 500 nodes.

File: Graph.h

```cpp
#pragma once
#include <vector>

// Define the maximum number of nodes in the graph
#define GRAPH_MAX_N 500
```

```cpp
#define TRANSHUB_NUMBER_THRESHOLD 5 // Threshold for the number of paths to consider
a node as a transhub

// Graph class represents a graph with adjacency matrix representation
class Graph {
public:
    int n; // Number of nodes in the graph
    int adj[GRAPH_MAX_N][GRAPH_MAX_N]; // Adjacency matrix to store edge lengths

    // Constructor to initialize the graph with a given number of nodes
    Graph(int n);

    // Adds an edge between nodes u and v with a specified length
    void add_edge(int u, int v, int length);

    // Initializes the graph by reading edges from standard input
    // m represents the number of edges
    void init_from_stdin(int m);
};

// GraphShortestPathSolution class is used to solve the shortest path problem
class GraphShortestPathSolution {
public:
    Graph *graph; // Pointer to the graph object
    int start; // Starting node for the shortest path calculation
    int visited[GRAPH_MAX_N]; // Array to track visited nodes during pathfinding
    std::vector<int> path_parent[GRAPH_MAX_N]; // Stores parent nodes for each node
in the shortest path
    int path_length[GRAPH_MAX_N]; // Stores the shortest path length to each node
    bool solved; // Flag to indicate if the shortest path solution has been computed

    // Constructor to initialize the solution object with a graph and starting node
    GraphShortestPathSolution(Graph *graph, int start);

    // Solves the shortest path problem using an appropriate algorithm
    void solve();

    // Retrieves all shortest paths to a specified destination node
    // Returns a vector of paths, where each path is represented as a vector of node
    std::vector<std::vector<int>> get_all_shortest_path(int destination);

    // Converts the paths to a matrix representation
    std::vector<std::vector<int>> shortest_paths_to_matrix(int destination);

    // Counts the number of paths from each node to the start node
        std::vector<int>  count_path_to_start(int  destination,  int  threshold  =
TRANSHUB_NUMBER_THRESHOLD);

    // Retrieves the reverse paths from the start node to a given `destination` node
    std::vector<int>* get_path_parent_reverse(int destination);
```

```cpp
    // Counts the number of paths from each node to the destination node
    std::vector<int> count_path_to_destination(int destination, int threshold =
TRANSHUB_NUMBER_THRESHOLD);

private:
    // Recursive helper function to fill the matrix with path lengths
    void _shortest_paths_to_matrix(std::vector<std::vector<int>> &matrix, int
destination);

    // Recursive helper function to count the number of paths from each node to the
start node
    int _count_path_to_start(int destination, std::vector<int> &count_path, int
threshold);

    // Recursive helper function to retrieve the reverse paths from the start node
to a given `destination` node
    void _get_path_parent_reverse(int    destination,    std::vector<int>
*path_parent_reverse, std::vector<int> &visited);

    // Recursive helper function to count the number of paths from each node to the
destination node
    int _count_path_to_destination(int destination, int current, std::vector<int>
&count_path, std::vector<int> *path_parent_reverse, int threshold);
};
```

File: Graph.cpp

```cpp
#include "Graph.h"
#include <cstdio>

// Constructor for the Graph class
// Initializes a graph with `n` nodes and sets up the adjacency matrix
// `adj[i][j]` is initialized to 0 if `i == j` (self-loop), otherwise -1 (no edge)
Graph::Graph(int n) : n(n) {
    for (int i = 0; i < GRAPH_MAX_N; i++) {
        for (int j = 0; j < GRAPH_MAX_N; j++) {
            adj[i][j] = (i == j) ? 0 : -1; // Initialize adjacency matrix
        }
    }
}

// Adds an undirected edge between nodes `u` and `v` with a given `length`
void Graph::add_edge(int u, int v, int length) {
    adj[u][v] = adj[v][u] = length; // Symmetric assignment for undirected graph
}

// Reads edges from standard input and initializes the graph
// `m` is the number of edges to read
void Graph::init_from_stdin(int m) {
```

```cpp
    int c1, c2, length;
    for (int i = 0; i < m; i++) {
        scanf("%d %d %d", &c1, &c2, &length); // Read edge data
        add_edge(c1, c2, length); // Add edge to the graph
    }
}

// Constructor for the GraphShortestPathSolution class
// Initializes the shortest path solution for a given `graph` starting from `start`
// node
GraphShortestPathSolution::GraphShortestPathSolution(Graph *graph, int start)
    : graph(graph), start(start) {
    for (int i = 0; i < graph->n; i++) {
        visited[i] = 0; // Mark all nodes as unvisited
        path_length[i] = -1; // Initialize path lengths to -1 (unreachable)
    }
    path_length[start] = 0; // Distance to the start node is 0
    solved = false; // Mark the solution as unsolved
}

// Solves the shortest path problem using Dijkstra's algorithm
void GraphShortestPathSolution::solve() {
    // Initialize path lengths for direct neighbors of the start node
    for (int i = 0; i < graph->n; i++) {
        if (graph->adj[start][i] != -1) { // If there's an edge from start to `i`
            path_length[i] = graph->adj[start][i]; // Set path length
            path_parent[i].push_back(start); // Set start as the parent
        }
    }

    visited[start] = 1; // Mark the start node as visited

    // Iterate over all nodes to find the shortest paths
    for (int i = 0; i < graph->n; i++) {
        int min_length = -1; // Minimum path length found so far
        int min_node = -1; // Node corresponding to the minimum path length

        // Find the unvisited node with the smallest path length
        for (int j = 0; j < graph->n; j++) {
            if (!visited[j] && path_length[j] != -1 &&
                (min_length == -1 || path_length[j] < min_length)) {
                min_length = path_length[j];
                min_node = j;
            }
        }

        if (min_node == -1) break; // No more reachable nodes

        visited[min_node] = 1; // Mark the node as visited

        // Update path lengths for neighbors of the current node
```

```cpp
        for (int j = 0; j < graph->n; j++) {
            if (!visited[j] && graph->adj[min_node][j] != -1) { // If there's
an edge
                int new_length = path_length[min_node] + graph->adj[min_node][j]; //
Calculate new path length
                if (path_length[j] == -1 || new_length < path_length[j]) {
                    path_length[j] = new_length; // Update path length
                    path_parent[j].clear(); // Clear previous parents
                    path_parent[j].push_back(min_node); // Add new parent
                } else if (new_length == path_length[j]) {
                    path_parent[j].push_back(min_node); // Add alternative parent
                }
            }
        }
    }
    solved = true; // Mark the solution as solved
}

// Retrieves all shortest paths from the start node to a given `destination` node
std::vector<std::vector<int>>
GraphShortestPathSolution::get_all_shortest_path(int destination) {
    if (destination == start) {
        return {{start}}; // Base case: path to itself
    }
    if (!solved || path_length[destination] == -1) {
        return {}; // No solution or destination unreachable
    }
    std::vector<std::vector<int>> all_paths; // Container for all paths
    for (auto parent : path_parent[destination]) { // Iterate over all parents of
the destination
        std::vector<std::vector<int>> sub_paths = get_all_shortest_path(parent); //
Recursively get paths to the parent
        for (auto &sub_path : sub_paths) {
            sub_path.push_back(destination); // Append the destination to each
sub-path
            all_paths.push_back(sub_path); // Add the completed path to the result
        }
    }
    return all_paths; // Return all paths
}

// Converts the paths to a matrix representation
std::vector<std::vector<int>>
GraphShortestPathSolution::shortest_paths_to_matrix(int destination) {
    std::vector<std::vector<int>> matrix(graph->n, std::vector<int>(graph->n,
0)); // Initialize the matrix with 0 (no path)
    _shortest_paths_to_matrix(matrix, destination); // Fill the matrix with path
lengths
    return matrix; // Return the matrix representation of paths
}
```

```cpp
// Recursive helper function to fill the matrix with path lengths
void GraphShortestPathSolution::_shortest_paths_to_matrix(std::vector<std::vector<int>>
&matrix, int destination) {
    if (destination == start) {
        return; // Base case: path to itself
    }
    for (auto parent : path_parent[destination]) { // Iterate over all parents of
the destination
        matrix[parent][destination] = 1;
        _shortest_paths_to_matrix(matrix, parent); // Recursively fill the matrix
for each parent
    }
}


// Counts the number of paths from each node to the start node
std::vector<int> GraphShortestPathSolution::count_path_to_start(int destination,
int threshold) {
    std::vector<int> count_path(graph->n, -1); // Initialize path counts to 0
    _count_path_to_start(destination, count_path, threshold); // Count paths
recursively
    return count_path; // Return the path counts
}


// Recursive helper function to count the number of paths from each node to the
start node
int        GraphShortestPathSolution::_count_path_to_start(int        destination,
std::vector<int> &count_path, int threshold) {
    if (destination == start) {
        return 1; // Base case: path to itself
    }
    if (count_path[destination] != -1) {
        return count_path[destination]; // Return cached result if already computed
    }
    int count = 0; // Initialize path count
    for (auto parent : path_parent[destination]) { // Iterate over all parents of
the destination
        int temp = _count_path_to_start(parent, count_path, threshold); // Count
paths recursively
        if (count <= threshold) { // Check if the count exceeds the threshold
            count += temp; // Increment the count
        }
    }
    count_path[destination] = count; // Cache the result
    return count; // Return the total path count
}


// Retrieves the reverse paths from the start node to a given `destination` node
std::vector<int>*        GraphShortestPathSolution::get_path_parent_reverse(int
destination) {
    std::vector<int>* path_parent_reverse = new std::vector<int>[GRAPH_MAX_N]; //
Allocate memory for reverse paths
```

```cpp
    std::vector<int> visited = std::vector<int>(graph->n, 0); // Initialize visited
array
    _get_path_parent_reverse(destination, path_parent_reverse, visited); // Fill
the reverse paths recursively
    return path_parent_reverse; // Return the reverse paths
}

// Recursive helper function to count the number of paths from each node to the
start node
void    GraphShortestPathSolution::_get_path_parent_reverse(int    destination,
std::vector<int> *path_parent_reverse, std::vector<int> &visited) {
    if (visited[destination]) {
        return; // If already visited, return to avoid cycles
    }
    visited[destination] = 1; // Mark the destination as visited
    if (destination == start) {
        return; // Base case: path to itself
    }
    for (auto parent : path_parent[destination]) { // Iterate over all parents of
the destination
        path_parent_reverse[parent].push_back(destination); // Add the destination
to the parent's reverse path
            _get_path_parent_reverse(parent, path_parent_reverse, visited); //
Recursively fill the reverse paths for each parent
    }
}

// Counts the number of paths from each node to the destination node
std::vector<int>        GraphShortestPathSolution::count_path_to_destination(int
destination, int threshold) {
    std::vector<int> *path_parent_reverse = get_path_parent_reverse(destination); //
Get reverse paths
    std::vector<int> count_path(graph->n, -1); // Initialize path counts to 0
    _count_path_to_destination(destination, start, count_path, path_parent_reverse,
threshold); // Count paths recursively
    delete[] path_parent_reverse; // Free allocated memory for reverse paths
    return count_path; // Return the path counts
}

// Recursive helper function to count the number of paths from each node to the
destination node
int  GraphShortestPathSolution::_count_path_to_destination(int  destination,  int
current, std::vector<int> &count_path, std::vector<int> *path_parent_reverse, int
threshold) {
    if (current == destination) {
        return 1; // Base case: path to itself
    }
    if (count_path[current] != -1) {
        return count_path[current]; // Return cached result if already computed
    }
    int count = 0; // Initialize path count
```

```
      for (auto parent : path_parent_reverse[current]) { // Iterate over all parents
  of the current node
            int temp = _count_path_to_destination(destination, parent, count_path,
  path_parent_reverse, threshold); // Count paths recursively
          if (count <= threshold) { // Check if the count exceeds the threshold
              count += temp; // Increment the count
          }
      }
      count_path[current] = count; // Cache the result
      return count; // Return the total path count
  }
```

5.3. main.cpp

This module provides the main entry point for the simplification program, handling input processing, graph analysis, and result output. It reads the graph structure (nodes, weighted edges) and test cases from standard input, then computes nodes that appear in shortest paths above a specified threshold.

File: main.cpp

```
#include <cstdio> // For standard input/output functions
#include <vector> // For using the std::vector container
#include <cstring> // For memset function
#include "Graph.h" // Custom header file for Graph-related classes and functions

int main() {
    int n, m, k; // n: number of nodes, m: number of edges, k: threshold for
node appearance
    scanf("%d %d %d", &n, &m, &k); // Read the number of nodes, edges, and threshold
    Graph* graph = new Graph(n); // Create a new Graph object with n nodes
     graph->init_from_stdin(m); // Initialize the graph by reading m edges from
standard input

    int T; // Number of test cases
    bool flag = false; // Flag to track if any node meets the condition
    int start, destination; // Variables to store the start and destination nodes
for each test case
    scanf("%d", &T); // Read the number of test cases
    for (int i = 0; i < T; i++) { // Loop through each test case
        scanf("%d %d", &start, &destination); // Read the start and destination node
        GraphShortestPathSolution* solution = new GraphShortestPathSolution(graph,
start); // Create a solution object for shortest path from the start node
        solution->solve(); // Solve the shortest path problem
                        std::vector<int>   count_path_to_start   =   solution-
>count_path_to_start(destination); // Count paths to the start node
                     std::vector<int>   count_path_to_destination   =   solution-
>count_path_to_destination(destination); // Count paths to the destination node
```

```
            for (int j = 0; j < n; j++) { // Check all nodes to see if they meet
the condition
                if (count_path_to_start[j] != -1 && j != start && j != destination &&
count_path_to_start[j]*count_path_to_destination[j] >= k) {
                    if (flag) printf(" "); // Print a space if this is not the first
node meeting the condition
                    else flag = true; // Set the flag to true if this is the first node
meeting the condition
                    printf("%d", j); // Print the node number
                }
            }
            if (!flag) { // If no node meets the condition, print "None"
                printf("None");
            }
            printf("\n"); // Print a newline after processing the test case
            flag = false; // Reset the flag for the next test case
        }

        delete graph;
        return 0;
    }
```

5.4. test.cpp

This module implements an automated testing framework for graph analysis, handling input/output validation, correctness checking, and performance measurement. It extends the main program with file-based testing capabilities and cross-platform compatibility.

File: test.cpp

```cpp
#include <cstdio> // For standard input/output functions
#include <vector> // For using the std::vector container
#include <cstring> // For memset function
#include <string> // For using the std::string container
#include <fstream> // For file stream and istreambuf_iterator
#include <ctime> // For time-related functions
#include "Graph.h" // Custom header file for Graph-related classes and functions
#include <unistd.h> // For access() function to check file accessibility

// Define the TTY constant based on the operating system
#ifdef _WIN32
    #define TTY "CON"
#else
    #define TTY "/dev/tty"
#endif

// Define the maximum number of nodes in the graph
```

```c
#define MAX_OUTPUT 10000

clock_t start_time, stop_time;  /* clock_t is a built-in type for processor time
(ticks) */
double duration;        /* Records the run time (seconds) of a function */

int main(int argc, char *argv[]) {
    // Check if the number of arguments is less than 2
    if (argc != 3) {
        // Print usage information
        printf("Usage: %s <input sample> <output sample>\n", argv[0]);
        return 1;
    }

    if(access(argv[1], R_OK) == -1) { // Check if the input file is readable
        printf("Input file %s is not accessible\n", argv[1]);
        return 1;
    }

    if(access(argv[2], R_OK) == -1) { // Check if the output file is readable
        printf("Output file %s is not accessible\n", argv[2]);
        return 1;
    }

     freopen(argv[1], "r", stdin); // Redirect standard input to read from the
specified file
      freopen("test.out", "w", stdout); // Redirect standard output to a file
named "test.out"

    // Record the start time (in ticks) before the main logic begins
    start_time = clock();

     int n, m, k; // n: number of nodes, m: number of edges, k: threshold for
node appearance
    scanf("%d %d %d", &n, &m, &k); // Read the number of nodes, edges, and threshold
    Graph* graph = new Graph(n); // Create a new Graph object with n nodes
     graph->init_from_stdin(m); // Initialize the graph by reading m edges from
standard input

    int T; // Number of test cases
     int count[GRAPH_MAX_N] = {0}; // Array to count the appearance of nodes in
shortest paths
    bool flag = false; // Flag to track if any node meets the condition
    int start, destination; // Variables to store the start and destination nodes
for each test case
    scanf("%d", &T); // Read the number of test cases
    for (int i = 0; i < T; i++) { // Loop through each test case
        scanf("%d %d", &start, &destination); // Read the start and destination node
        GraphShortestPathSolution* solution = new GraphShortestPathSolution(graph,
start); // Create a solution object for shortest path from the start node
        solution->solve(); // Solve the shortest path problem
```

```cpp
                        std::vector<int> count_path_to_start = solution-
>count_path_to_start(destination); // Count paths to the start node
                   std::vector<int> count_path_to_destination = solution-
>count_path_to_destination(destination); // Count paths to the destination node
        for (int j = 0; j < n; j++) { // Check all nodes to see if they meet
the condition
            if (count_path_to_start[j] != -1 && j != start && j != destination &&
count_path_to_start[j]*count_path_to_destination[j] >= k) {
                if (flag) printf(" "); // Print a space if this is not the first
node meeting the condition
                else flag = true; // Set the flag to true if this is the first node
meeting the condition
                printf("%d", j); // Print the node number
            }
        }
        if (!flag) { // If no node meets the condition, print "None"
            printf("None");
        }
        printf("\n"); // Print a newline after processing the test case
         memset(count, 0, sizeof(count)); // Reset the count array for the next
test case
        flag = false; // Reset the flag for the next test case
    }
    // Record the stop time (in ticks) after the main logic completes
    stop_time = clock();

    freopen(TTY, "r", stdin); // Redirect standard input back to the terminal
    freopen(TTY, "w", stdout); // Redirect standard output back to the terminal


                                                              std::string
output((std::istreambuf_iterator<char>(std::ifstream("test.out").rdbuf())),
std::istreambuf_iterator<char>()); // Read the output from "test.out" into a string
                                                              std::string
output_sample((std::istreambuf_iterator<char>(std::ifstream(argv[2]).rdbuf())),
std::istreambuf_iterator<char>()); // Read the expected output from the specified
file into a string

    remove("test.out"); // Remove the temporary output file "test.out"

    // Replace all occurrences of "\r\n" with "\n" in the output and output_sample
strings
    size_t pos;
    while ((pos = output.find("\r\n")) != std::string::npos) {
        output.replace(pos, 2, "\n");
    }
    while ((pos = output_sample.find("\r\n")) != std::string::npos) {
        output_sample.replace(pos, 2, "\n");
    }

    // Remove trailing newlines from both output and output_sample
```

```
    while (!output.empty() && output.back() == '\n') {
        output.pop_back();
    }
    while (!output_sample.empty() && output_sample.back() == '\n') {
        output_sample.pop_back();
    }

    if (output == output_sample) { // Compare the output with the expected output
        printf("Correct\n"); // If they match, print "Correct"

        // Calculate the duration of the program execution in seconds
        // CLOCKS_PER_SEC is a constant representing the number of ticks per second
        duration = ((double)(stop_time - start_time)) / CLOCKS_PER_SEC;

        // Print the duration of the program execution
        printf("%lf\n", duration);
    } else {
        printf("Wrong\n"); // If they don't match, print "Wrong"
    }

    delete graph;
    return 0;
}
```

5.5. samplegen.py

This is a python script to generate a large ramdom testcase. It does not need any arguments, but should be run with Python interpreter.

File: samplegen.py

```python
import random

# Generate a random number of nodes (n) between 3 and 500
n = random.randint(3, 500)

# Calculate the maximum number of roads (edges) possible in a complete graph with
n nodes
max_roads = n * (n - 1) // 2

# Generate a random number of roads (m) between 1 and max_roads
m = random.randint(1, max_roads)

# Generate a random number of special nodes (k) between 1 and 5
k = random.randint(1, 5)

# Create a list of node indices from 0 to n-1
nodes = list(range(n))
```

```python
# Shuffle the nodes to randomize their order
random.shuffle(nodes)

# Initialize an empty set to store unique edges
edges = set()

# Initialize a parent list for union-find operations (used for ensuring connectivity
parent = list(range(n))

# Function to find the root of a node in the union-find structure
def find(u):
    while parent[u] != u:  # Traverse up the tree until the root is found
        parent[u] = parent[parent[u]]  # Path compression for optimization
        u = parent[u]
    return u

# Ensure the graph is connected by adding a spanning tree
for i in range(1, n):
    u = nodes[i-1]  # Take the previous node
    v = nodes[i]    # Take the current node
    edges.add(tuple(sorted((u, v))))   # Add an edge between them (sorted to
avoid duplicates)

# Add additional random edges until the total number of edges equals m
while len(edges) < m:
    c1 = random.randint(0, n-1)  # Randomly select the first node
    c2 = random.randint(0, n-1)  # Randomly select the second node
    if c1 != c2:  # Ensure the two nodes are not the same
        edge = tuple(sorted((c1, c2)))  # Create a sorted tuple to represent the edg
        if edge not in edges:  # Add the edge only if it doesn't already exist
            edges.add(edge)

# Create a list to store roads with their lengths
roads = []
for c1, c2 in edges:
    length = random.randint(1, 30)  # Assign a random length between 1 and 30 to
each road
    roads.append((c1, c2, length))  # Add the road as a tuple (node1, node2, length)

# Shuffle the roads to randomize their order
random.shuffle(roads)

# Generate a random number of queries (T) between 1 and 500
T = random.randint(1, 500)

# Create a list to store queries
queries = []
for _ in range(T):
    src = random.randint(0, n-1)  # Randomly select a source node
    dst = random.randint(0, n-1)  # Randomly select a destination node
```

```python
    queries.append((src, dst))  # Add the query as a tuple (source, destination)

# Print the number of nodes, edges, and special nodes
print(n, m, k)

# Print all roads in the format: node1 node2 length
for road in roads:
    print(f"{road[0]} {road[1]} {road[2]}")

# Print the number of queries
print(T)

# Print all queries in the format: source destination
for query in queries:
    print(f"{query[0]} {query[1]}")
```

5.6. README.md / README.txt / README.pdf

The README file, nothing to introduce.

Declaration

I hereby declare that all the work done in this project titled "Hard Transportation Hub" is of my independent effort.