Hard Transportation Hub

Date: 2025/04/20

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, the restore of the shortest path, and the identification of transportation hubs.

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.

```
function BuildGraph(n: int, m: int)
    for i in 0 to n - 1
        for j in 0 to n - 1
            if i == j then
                adj[i][j] ← 0
            else
                adj[i][j] ← $-1$
            end
        end
    for count in 1 to m
        read (u, v, length)
        adj[u][v] ← length
        adj[v][u] ← length
    end
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:

```
function Dijkstra(start: int)
    for each node i
        visited[i] ← false
        path_length[i] ← $-1$
    path_length[start] ← 0
    for each neighbor i of start
```

```
        path_length[i] ← weight(start, i)
        path_parent[i] ← [start]
    visited[start] ← true
    repeat n times
        min_node ← node with smallest path_length among unvisited
        if min_node == −1 then break
        visited[min_node] ← true
        for each neighbor j of min_node
            if not visited and edge exists
                new_length ← path_length[min_node] + weight(min_node, j)
                if path_length[j] == −1 or new_length < path_length[j] then
                    path_length[j] ← new_length
                    path_parent[j] ← [min_node]
                else if new_length == path_length[j] then
                    path_parent[j].append(min_node)
                end
            end
        end
    end
```

This modified Dijkstra lays the foundation for backtracking all shortest paths and counting node appearances to identify transportation hubs.

2.3. Backtracking All Shortest Paths

Once the shortest path lengths and their respective predecessors are computed, the next step is to reconstruct **all possible shortest paths** from the start node to a given destination. This is achieved through recursive backtracking.

Each node maintains a list of its predecessor nodes on any shortest path. To build all paths to the destination, we recursively construct all paths to its predecessors and append the destination node to each of them.

This approach ensures that every shortest route from the start node to the destination is captured.

The pseudocode below illustrates the recursive backtracking logic:

```
function GetAllShortestPaths(destination: int) → list of paths
    if destination == start then
        return [[start]]
    if not solved or path_length[destination] == −1 then
```

```
        return []
    all_paths ← []
    for each parent in path_parent[destination]
        sub_paths ← GetAllShortestPaths(parent)
        for each sub_path in sub_paths
            sub_path.append(destination)
            all_paths.append(sub_path)
        end
    end
    return all_paths
end
```

This method provides the foundation for analyzing node appearances on all shortest paths, enabling the identification of transportation hubs.

2.4. The Identification of Transportation Hubs

After all shortest paths between a pair of cities are generated, the final step is to identify which cities qualify as **transportation hubs**. According to the definition, a city is considered a hub if it appears on at least k of the shortest paths — excluding the start and end cities themselves.

To achieve this, the program counts how many times each city appears across all shortest paths. If the count reaches the given threshold k, the city is considered a hub and is included in the output for that source-destination query.

The pseudocode below outlines this identification process:

```
function FindHubs(all_paths, start, destination, k)
    count[node] ← 0 for all nodes
    for each path in all_paths
        for each node in path
            count[node] += 1
        end
    end
    hubs ← []
    for each node j
        if count[j] ≥ k and j ≠ start and j ≠ destination
            hubs.append(j)
        end
    end
    if hubs is empty
        print “None”
    else
        print hubs in ascending order
    end
end
```

This procedure ensures that only cities that serve as true intermediaries in multiple shortest paths are labeled as transportation hubs, filtering out source and destination nodes.

2.5. 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.

Below is a streamlined pseudocode version focusing only on the main orchestration logic:

```
Read n, m, k
Create graph with n nodes
Read m edges into the graph
Read T test cases
for each (start, destination) pair
    Create shortest path solver from start
    Run Dijkstra’s algorithm
    Retrieve all shortest paths to destination
    Count appearances of nodes in all paths
    Identify transportation hubs based on count
    Print results
    Reset counters for next case
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 origin 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

`4 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: 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:

```
generate random number n in [3, 500]
calculate max_roads = n * (n - 1) / 2
generate random number m in [1, max_roads]
generate random number k in [1, 5]
initialize node list from 0 to n-1
shuffle node list for randomness
initialize empty set edges and parent array for union-find
define function find(u):
    while parent[u] != u
        perform path compression and move up
    end
    return root of u
for i = 1 to n-1 do
    connect node[i-1] and node[i] to ensure connectivity
    add edge to edges
while number of edges < m
    randomly select c1 and c2 from nodes
    if c1 $\neq$ c2 and (c1, c2) not in edges
        add edge (sorted) to edges
for each edge in edges
```

```
    assign a random weight in [1, 30]
    store as (node1, node2, length)
shuffle all roads
generate T in [1, 500]
**repeat** T times
    randomly select `src` and `dst`
    store as a query pair
output n, m, k
output all roads
output T
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 Backtracking process, 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(V^2) 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. Backtracking Paths

4.2.1. Time Complexity

The time complexity of the Path Backtracking process is O(P × V), where P is the number of shortest paths from the start to the destination, and V is the number of nodes. This is because:

1. For each parent of the destination node, the function recursively generates all sub-paths to that parent. If a node has multiple parents (e.g., in a grid-like graph), this leads to combinatorial branching.
2. Appending the destination node to each sub-path takes $O(V)$ time per path.

In the worst case (e.g., a fully connected graph where every node is a parent of its successors), the number of paths P grows exponentially with V, resulting in O(2^V × V) time complexity.

4.2.2. Space Complexity

The space complexity is O(P × V) due to:
1. Storing all generated paths in memory, where each path contains up to $O(V)$ nodes.
2. The recursion stack depth can reach $O(V)$ (for paths spanning all nodes).

In scenarios with exponentially many shortest paths (e.g., graphs with redundant shortest paths), the space complexity becomes O(2^V × V). Additionally, the `path_parent` structure (precomputed in Dijkstra's algorithm) contributes $O(V^2)$ space, but this is considered part of the input and not the algorithm's working space.

4.2.3. Improvements

Maybe the process can be skiped if the number of paths is too large, and we can just count the number of paths passing each node.

Based on this conjecture, The follwing is a possible improvement to the algorithm:

Graph Theory and Linear Algebra

An unweighted directed graph can be represented as an adjacency matrix A, where each entry (i, j) is either 1 or 0:

- `1` indicates the presence of a directed edge from node `i` to node `j`.
- `0` indicates the absence of such an edge.

A more universal interpretation is that this matrix encodes the number of edges of length 1 between any pair of points.

Based on the matrix multiplication principle, the power of matrix A, denoted as A^k, reflects the number of distinct paths of length k between any two nodes in the graph. Specifically:

- The entry (i, j) in A^k represents the number of paths of exactly length k from node `i` to node `j`.
- This property can be extended to compute the total number of paths of length ≤ k by summing the powers of A from 1 to k:

$$A + A^2 + \ldots + A^k$$

- The total number of paths from node `i` to node `j` of any length can be computed by summing all powers of A:

$$A + A^2 + A^3 + \ldots + A^k + \ldots = A(I - A)^{-1} = (I - A)^{-1} - I$$

In this project, we can restore the shortest paths to a matrix A, which is much more efficient when the number of paths is large. The matrix A can be constructed as follows, then we can calculate $P = (I - A)^{-1} - I$, which gives us the number of paths between any two nodes.

The number of shortest paths which passes the node N can be calculated by the following formula:

$$P[\text{start}][N] \times P[N][\text{destination}]$$

Then compare the number of paths with the threshold k to determine if the node N is a transportation hub.

This matrix-based approach offers a more scalable alternative to recursive backtracking, especially for dense graphs or graphs with a high degree of redundancy in shortest paths.

I'm not using this algorithm for this project because linear algebra-related computations require libraries like BLAS, MKL, etc., and leverage hardware features to fully utilize their advantages. If you're interested, you can try implementing it.

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 retrieve all 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
```

```
// 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);

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

File: Graph.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) {
    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
        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
}

// Recursive helper function to fill the matrix with path lengths
voidGraphShortestPathSolution::_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
    }
}
```

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
     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
                        std::vector<std::vector<int>>   all_paths   =   solution-
>get_all_shortest_path(destination); // Get all shortest paths to the destination
node
        for (auto path : all_paths) { // Iterate through all shortest paths
            for (auto node : path) { // Iterate through each node in the path
                count[node]++; // Increment the count for the node
            }
        }
         for (int j = 0; j < n; j++) { // Check all nodes to see if they meet
the condition
             if (count[j] >= k && j != start && j != destination) { // Node must
appear at least k times and not be the start or destination
                 if (flag) { // If a node has already been printed, print a space
before the next node
                    printf(" ");
                } else {
                    flag = true; // Set the flag to true after printing the
first node
                }
                printf("%d", j); // Print the node
            }
        }
        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
    }

    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

```
#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
#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
                        std::vector<std::vector<int>>   all_paths   =   solution-
>get_all_shortest_path(destination); // Get all shortest paths to the destination
node
        for (auto path : all_paths) { // Iterate through all shortest paths
            for (auto node : path) { // Iterate through each node in the path
                count[node]++; // Increment the count for the node
            }
        }
          for (int j = 0; j < n; j++) { // Check all nodes to see if they meet
the condition
             if (count[j] >= k && j != start && j != destination) { // Node must
appear at least k times and not be the start or destination
                if (flag) { // If a node has already been printed, print a space
before the next node
                    printf(" ");
                } else {
                        flag = true; // Set the flag to true after printing the
first node
                }
                printf("%d", j); // Print the node
            }
        }
        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

```
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))

# 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
    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]}")
```

```python
# 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.