download.png

Assignment:

Integrated Activity 2

Student's Names:

José María Soto Valenzuela (A01254831)
César Alán Silva Ramos (A01252916)
German Avelino Del Rio Guzman (A01641976)

Course Name:

Analysis and Design of Advanced Algorithms
(Group 601)

Date:

October, 2024

Contents

1 Introduction

During the year 2020, the whole world was affected by an event that no one expected: the pandemic caused by COVID-19. Health measures were taken in every country to contain the pandemic. One of these measures was to send the entire population home, moving most of the face-to-face activities to a remote model in which the ISP (Internet Service Provider) companies had a leading role. Many people moved to the remote work or home-office model, and most educational institutions decided to continue their operations under the remote model as well, greatly increasing data transmission over the Internet.

If it were up to us to improve Internet services in a small population:

- Could we decide how to wire the most important points in that town in a way that we use the least amount of fiber optic cable?
- Assuming we have several ways to connect two nodes in the town, for a person who must visit all the points in the network, what will be the optimal way to do it and return to the point of origin?
- Could we analyze the maximum amount of information that can pass from one node to another?
- Could we analyze the feasibility of connecting to the network a new point (a new location) on the map?

This report addresses these questions by implementing algorithms that provide solutions to each of the problems. The program developed in C++ covers the following requirements:

1. Determining the optimal way to wire neighborhoods with optical fiber.
2. Finding the shortest possible route for mail delivery personnel.
3. Calculating the maximum information flow between the initial and final nodes.
4. Assigning new service contracts to the geographically closest exchange.

For each part, the chosen algorithms are explained, along with the reasons for their selection and an analysis of their computational complexities. The code is dissected to elucidate how the algorithms are implemented.

2 Part 1: Optimal Wiring with Optical Fiber

2.1 Problem Description

The company needs to determine the optimal way to wire the neighborhoods with optical fiber to ensure that information can be shared between any two neighborhoods while minimizing the total length of the wiring.

2.2 Algorithm Choice and Justification

Algorithm Selected: *Kruskal's Algorithm*

Justification:

- **Suitability:** Kruskal's Algorithm is ideal for finding a Minimum Spanning Tree (MST) in a weighted, undirected graph, which is exactly the type of problem represented by the neighborhoods and their wiring distances. By systematically choosing the shortest edges that do not form a cycle, Kruskal's ensures that all nodes (neighborhoods) are connected while minimizing the total edge weight (the total wiring distance).
- **Efficiency:** Kruskal's Algorithm is efficient, particularly for sparse graphs. Its time complexity, $O(E \log E)$, is acceptable for the size of the problem.
- **Simplicity:** The algorithm is relatively simple to implement and understand, reducing complexity in coding and debugging.
- **Optimality:** It guarantees an optimal solution for connecting all neighborhoods with the least total wiring distance.

2.3 Code Explanation

Edge Structure

```cpp
// Structure to represent an edge in the graph
struct Edge {
    int from;   // Starting node of the edge
    int to;     // Ending node of the edge
    int weight; // Weight of the edge (e.g., distance)
};
```

Union-Find Disjoint Set Class

```cpp
// Union-Find Disjoint Set class for Kruskal's Algorithm
class UnionFind {
private:
    vector<int> parent; // Parent of each node
    vector<int> rank;   // Rank (approximate depth) of each tree

public:
    // Constructor to initialize the Union-Find structure
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; i++)
            parent[i] = i; // Initially, each node is its own parent
    }

    // Find operation with path compression
    int find(int x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]); // Path compression
        return parent[x];
    }

    // Union operation by rank
    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY)
            return; // They are already in the same set

        // Attach the smaller rank tree under the root of the
           higher rank tree
        if (rank[rootX] < rank[rootY])
            parent[rootX] = rootY;
        else if (rank[rootX] > rank[rootY])
            parent[rootY] = rootX;
        else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }
};
```

Kruskal's Algorithm Implementation

```
vector<Edge> kruskalMST(int numNodes, vector<vector<int>>& graph) {
    vector<Edge> edges;

    // Build the list of edges from the adjacency matrix
    for (int i = 0; i < numNodes; i++) {
        for (int j = i + 1; j < numNodes; j++) {
            if (graph[i][j] > 0) {
                edges.push_back({i, j, graph[i][j]});
            }
        }
    }

    // Sort edges by weight in ascending order
    sort(edges.begin(), edges.end(), [](Edge a, Edge b) {
        return a.weight < b.weight;
    });

    UnionFind uf(numNodes);
    vector<Edge> mst; // Store the edges included in the MST

    // Iterate over sorted edges and build MST
    for (auto& edge : edges) {
        if (uf.find(edge.from) != uf.find(edge.to)) {
            uf.unite(edge.from, edge.to);
            mst.push_back(edge);
        }
    }
    return mst;
}
```

2.4 Computational Complexity

Time Complexity Analysis:

- **Sorting Edges:** The algorithm begins by sorting all the edges in the graph based on their weights. This step has a time complexity of $O(E \log E)$, where E is the number of edges. Sorting is necessary to ensure that we consider the edges in order of increasing weight.
- **Union-Find Operations:** For each edge, we perform Union-Find operations to check if adding that edge will form a cycle. Thanks to path compression and union by rank optimizations, each Union-Find operation has an almost constant time complexity, specifically $O(\alpha(N))$, where α is the inverse Ackermann function, which grows extremely slowly and can be considered constant for all practical purposes.
- **Total Time Complexity:** Since we perform Union-Find operations for each of the E edges, the total time complexity for these operations is $O(E \cdot \alpha(N))$. However, this is dominated by the sorting step, so the overall time complexity of Kruskal's algorithm is $O(E \log E)$.

Space Complexity Analysis:

- **Edge List Storage:** We store all the edges of the graph, requiring $O(E)$ space.
- **Union-Find Data Structures:** The Union-Find data structures require $O(N)$ space to store the parent and rank arrays, where N is the number of nodes.
- **Total Space Complexity:** Overall, the space complexity is $O(E + N)$, which is acceptable for most practical applications.

2.5 Discussion and Justification

Using Kruskal's algorithm for this problem is an excellent choice due to its efficiency and suitability for finding a Minimum Spanning Tree (MST). In the context of wiring neighborhoods with optical fiber, we need to connect all neighborhoods with the minimum total length of fiber while ensuring that the network is fully connected. Kruskal's algorithm achieves this optimally.

Real-Life Example:

An example of Kruskal's algorithm in real life is in designing electrical grids or road networks where the goal is to connect multiple locations with the least amount of wiring or road length. For instance, utility companies use similar algorithms to minimize the cost of laying out power lines between substations and consumers.

Importance:

By applying Kruskal's algorithm, the company can significantly reduce costs associated with the installation of optical fiber. Minimizing the total length of fiber not only saves material costs but also reduces maintenance expenses in the long term. Additionally, an efficiently wired network improves data transmission speeds and reliability, enhancing customer satisfaction.

3 Part 2: Shortest Route for Mail Delivery Personnel Using Held-Karp Algorithm

3.1 Problem Description

The company requires a route for mail delivery personnel that visits each neighborhood exactly once and returns to the starting point, minimizing the total distance traveled.

3.2 Algorithm Choice and Justification

Algorithm Selected: *Held-Karp Algorithm (Dynamic Programming Approach to the TSP)*

Justification:

- **Efficiency Over Brute-Force:** The brute-force method for solving the Traveling Salesman Problem (TSP) has a time complexity of $O(N!)$, which becomes infeasible even for moderate values of N. The Held-Karp algorithm reduces the time complexity to $O(N^2 \times 2^N)$, which is significantly more efficient.
- **Optimal Solution:** Like the brute-force approach, the Held-Karp algorithm provides an exact solution to the TSP, guaranteeing the shortest possible route.
- **Dynamic Programming Advantages:** By storing intermediate results and avoiding redundant calculations, the Held-Karp algorithm optimizes performance and resource usage.

Why Held-Karp is Better Than Brute-Force:

- **Time Complexity Comparison:** For $N = 10$:
  - **Brute-Force:** $O(10!) = 3,628,800$ possible permutations.
  - **Held-Karp:** $O(10^2 \times 2^{10}) = 10,240$ operations.
  - **Improvement:** The Held-Karp algorithm reduces the number of computations by a factor of over 300 times.

- **Scalability:** While both algorithms are exponential, Held-Karp scales much better with increasing N, making it practical for larger problems where brute-force becomes unusable.

3.3 Code Explanation

To ensure that the comments fit within the page width, we have placed detailed explanations above the code sections.

Held-Karp Algorithm Implementation
Function Description:

- The function `tspHeldKarp` solves the Traveling Salesman Problem using dynamic programming.
- It calculates the minimum cost to visit all nodes and return to the starting point.
- It reconstructs the optimal path based on the computed costs.

```
int tspHeldKarp(int numNodes, vector<vector<int>> &graph,
   vector<int> &optimalPath) {
    const int INF = INT_MAX;
    int N = numNodes;
    int VISITED_ALL = (1 << N) - 1; // All nodes have been visited
       when the bitmask is all ones

    // dp[mask][i]: minimum cost to reach node i with visited nodes
       represented by mask
    vector<vector<int>> dp(1 << N, vector<int>(N, INF));

    // Base case: starting at node 0, cost is 0
    dp[1 << 0][0] = 0;

    // Iterate over all subsets of nodes (represented by masks)
    for (int mask = 0; mask < (1 << N); mask++) {
        for (int u = 0; u < N; u++) {
            // If u is included in the current subset represented
               by mask
            if (mask & (1 << u)) {
                // Try to find the minimum cost to reach each node
                   v from u
                for (int v = 0; v < N; v++) {
                    // Skip if v is already in the mask or u and v
                       are the same
                    if ((mask & (1 << v)) || u == v)
                        continue;

                    // Update dp[mask | (1 << v)][v] if a better
                       cost is found
                    if (dp[mask][u] != INF && dp[mask][u] +
                       graph[u][v] < dp[mask | (1 << v)][v]) {
                        dp[mask | (1 << v)][v] = dp[mask][u] +
                           graph[u][v];
                    }
                }
            }
        }
    }

    // Reconstruct the optimal path
    int mask = VISITED_ALL;
    int last = 0;
    int min_cost = INF;

    // Find the ending node that gives minimum cost
    for (int i = 1; i < N; i++) {
        if (dp[mask][i] != INF && dp[mask][i] + graph[i][0] <
           min_cost) {
            min_cost = dp[mask][i] + graph[i][0];
            last = i;
        }
    }

    // Backtracking to find the optimal path
    optimalPath.push_back(0); // Start from node 0
    int current_mask = mask;
    int current_node = last;

    vector<int> path;
    path.push_back(last);

    while (current_mask != (1 << 0)) {
        int prev_mask = current_mask ^ (1 << current_node);
        for (int i = 0; i < N; i++) {
            if ((prev_mask & (1 << i)) && dp[prev_mask][i] != INF &&
                dp[prev_mask][i] + graph[i][current_node] ==
                   dp[current_mask][current_node]) {
                path.push_back(i);
                current_node = i;
                current_mask = prev_mask;
                break;
            }
        }
    }

    // Reverse the path and add to optimalPath
    for (int i = path.size() - 1; i >= 0; i--) {
        optimalPath.push_back(path[i]);
    }

    optimalPath.push_back(0); // Return to starting node

    return min_cost;
}
```

3.4 Computational Complexity

Time Complexity Analysis:

- The Held-Karp algorithm has a time complexity of $O(N^2 \times 2^N)$, where N is the number of nodes (neighborhoods).
- This complexity arises because for each subset of nodes (there are 2^N subsets), we compute the shortest paths ending at each node (there are N nodes), and each such computation can involve $O(N)$ operations.

Space Complexity Analysis:

- The algorithm requires $O(N \times 2^N)$ space to store the dynamic programming table (`dp[mask][i]`), which holds the minimum cost to reach node i with the subset of nodes represented by `mask`.

3.5 Discussion and Justification

While the Held-Karp algorithm significantly reduces the computational complexity compared to the brute-force method ($O(N!)$), it is still exponential and becomes impractical for very large N. However, for small to medium-sized problems (up to around $N = 20$), it is feasible and provides exact solutions.

In our context, if the number of neighborhoods is within a manageable range, the Held-Karp algorithm is a good choice because it guarantees the shortest possible route, which is critical for optimizing mail delivery efficiency.

Real-Life Example:

The Traveling Salesman Problem (TSP) models many real-life scenarios beyond mail delivery, such as route planning for delivery trucks, circuit board manufacturing (where the path of the drill must be optimized), and even DNA sequencing.

For instance, a logistics company planning the daily routes for its delivery trucks can use TSP algorithms to minimize fuel consumption and delivery times, leading to cost savings and improved customer service.

Importance:

Optimizing the mail delivery route ensures that the mail personnel spend less time and resources covering the neighborhoods. This efficiency translates into reduced operational costs and faster mail delivery, enhancing service quality.

4 Part 3: Maximum Information Flow Using Ford-Fulkerson

4.1 Problem Description

The company wants to determine the maximum information flow from the initial node to the final node, given the capacities between neighborhoods and considering potential electromagnetic interference.

4.2 Algorithm Choice and Justification

Algorithm Selected: *Ford-Fulkerson Algorithm with Depth-First Search (DFS)*
Justification:

- **Maximum Flow Problem:** The scenario is a classic maximum flow problem, where we want to find the maximum amount of flow possible from a source to a sink in a network.
- **Ford-Fulkerson Method:** Ford-Fulkerson provides a simple and intuitive method for computing maximum flow by repeatedly finding augmenting paths and increasing the flow until no more paths exist.
- **Simplicity:** The DFS approach in Ford-Fulkerson is straightforward to implement and understand, making it suitable for this problem.
- **Applicability:** For moderate-sized graphs, the algorithm performs efficiently and yields exact results.

4.3 Code Explanation

To keep comments within the page width, explanations are provided above the code.
DFS Function for Ford-Fulkerson Algorithm
Function Description:

- The `dfs` function performs a Depth-First Search on the residual graph.
- It attempts to find an augmenting path from the source to the sink.
- The `parent` array keeps track of the path for flow updates.

```
bool dfs(vector<vector<int>>& residualGraph, int current, int sink,
    vector<int>& parent) {
    if (current == sink)
        return true; // Reached sink node

    for (int next = 0; next < residualGraph.size(); ++next) {
        // If there is available capacity and next node is unvisited
        if (residualGraph[current][next] > 0 && parent[next] == -1)
           {
            parent[next] = current; // Set parent for path
               reconstruction
            if (dfs(residualGraph, next, sink, parent))
                return true; // Path to sink found
        }
    }
    return false; // No path found from current node
}
```

Maximum Flow Function
Function Description:

- The `maxFlow` function calculates the maximum flow from source to sink.
- It uses the `dfs` function to find augmenting paths.
- Residual capacities are updated after each augmentation.

```
int maxFlow(vector<vector<int>>& graph, int source, int sink) {
    int totalFlow = 0;                              // Initialize
       total flow to zero
    vector<vector<int>> residualGraph = graph;      // Create
       residual graph
    vector<int> parent(graph.size());               // To store
       augmenting path

    while (true) {
        fill(parent.begin(), parent.end(), -1);     // Reset parent
           array
        parent[source] = -2;                        // Mark source
           node as visited

        // Use DFS to find an augmenting path
        if (!dfs(residualGraph, source, sink, parent))
            break; // No more augmenting paths, algorithm terminates

        // Find the bottleneck (minimum residual capacity) along
           the path found
        int flow = INT_MAX;
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            flow = min(flow, residualGraph[u][v]);
        }

        // Update residual capacities along the path
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            residualGraph[u][v] -= flow;  // Subtract flow from
               forward edge
            residualGraph[v][u] += flow;  // Add flow to reverse
               edge
        }

        totalFlow += flow; // Add flow to total flow
    }
    return totalFlow;
}
```

4.4 Computational Complexity

Time Complexity Analysis:

- The time complexity of the Ford-Fulkerson algorithm depends on the method used to find augmenting paths and the capacities in the network.
- In the worst case, the time complexity is $O(E \times F)$, where E is the number of edges, and F is the maximum flow value.
- When capacities are integers, this can lead to a pseudo-polynomial time complexity, which may be inefficient for large capacities.
- However, with the use of more efficient methods like the Edmonds-Karp algorithm (which uses BFS for finding shortest augmenting paths), the time complexity improves to $O(V \times E^2)$.

Space Complexity Analysis:

- The algorithm requires $O(V^2)$ space to store the residual graph, where V is the number of vertices.
- Additional space is needed for the parent array and other auxiliary data structures.

4.5 Discussion and Justification

Using the Ford-Fulkerson algorithm with DFS is acceptable for smaller networks or when the capacities are not too large. However, for larger networks or networks with high capacities, the algorithm may be inefficient due to its time complexity.

In such cases, using algorithms like Edmonds-Karp or Dinic's algorithm, which have better time complexities ($O(V \times E^2)$ and $O(E \times \sqrt{V})$, respectively), may be more appropriate.

In our scenario, if the network size is moderate, the Ford-Fulkerson algorithm with DFS provides a simple and effective solution for computing the maximum flow.

Real-Life Example:

Max-flow algorithms are widely used in various fields such as network traffic management, where the goal is to maximize the data transfer between two points in a network.

Another example is in project planning, where resources must flow through a network of tasks, and the goal is to find the bottleneck that limits the overall throughput.

Importance:

Determining the maximum information flow between nodes allows the company to identify potential bottlenecks in the network. By knowing the maximum capacity, the company can plan for upgrades, balance loads, and ensure that the network can handle the required data transmission demands.

5 Part 4: Assigning New Contracts to the Closest Exchange

5.1 Problem Description

Given the geographic locations of exchanges (central offices), the company needs to determine the exchange closest to a new service contract. To enhance accuracy and decision-making efficiency, a Voronoi diagram approach has been chosen to assign contracts based on proximity.

5.2 Algorithm Choice and Justification

Algorithm Selected: *Voronoi Diagram with CGAL Library*

Justification:

- **Precision:** Voronoi diagrams allow for partitioning based on proximity, creating regions around each exchange that automatically assign any given point (contract) to its nearest exchange.
- **Efficiency:** The CGAL library provides efficient algorithms to construct Voronoi cells, making the assignment process scalable and computationally feasible for large datasets.
- **Direct Application:** By leveraging Voronoi cells, contracts are assigned based on a geometric approach, accurately modeling real-world proximity relationships.

5.3 Code Explanation

Generating Voronoi Cells

Process Description:

- Insert exchange coordinates into Delaunay triangulation.
- Generate Voronoi cells corresponding to each exchange.
- Map each exchange to its Voronoi polygon.

```cpp
Delaunay delaunay;
vector<Point> points;

// Insert exchange coordinates into Delaunay triangulation
for (const auto& coord : coordinates)
    points.emplace_back(coord.first, coord.second);
delaunay.insert(points.begin(), points.end());

// Generate Voronoi cells for each exchange
map<Point, vector<pair<double, double>>> voronoi_polygons;
for (auto vertex = delaunay.finite_vertices_begin(); vertex !=
   delaunay.finite_vertices_end(); ++vertex) {
    vector<pair<double, double>> polygon;
    Delaunay::Face_circulator circulator =
       delaunay.incident_faces(vertex), done = circulator;
    do {
        Face_handle face = circulator;
        if (!delaunay.is_infinite(face)) {
            Point voronoi_vertex = delaunay.dual(face);
            polygon.emplace_back(voronoi_vertex.x(),
               voronoi_vertex.y());
        }
        ++circulator;
    } while (circulator != done);
    voronoi_polygons[vertex->point()] = polygon;
}
```

5.4 Computational Complexity

Time Complexity Analysis:

- The construction of the Voronoi diagram involves computing the Delaunay triangulation of the exchange points, which has an expected time complexity of $O(n \log n)$, where n is the number of exchange points.
- Querying the Voronoi diagram to find the closest exchange to a new contract location can be done efficiently, often in $O(\log n)$ time per query, depending on the data structures used.

Space Complexity Analysis:

- Storing the Delaunay triangulation and the associated Voronoi diagram requires $O(n)$ space.
- Additional space is needed for any indexing structures used to speed up queries.

5.5 Discussion and Justification

Using Voronoi diagrams is an excellent choice for this problem because it naturally partitions the space based on proximity to the given points (exchanges). This method is efficient and scalable, handling large numbers of exchanges and queries efficiently.

The CGAL library provides robust and optimized implementations of these geometric algorithms, ensuring that the solutions are accurate and reliable.

Real-Life Example:

Voronoi diagrams are used in many fields, such as cell phone network design, where each cell tower's coverage area can be approximated by its Voronoi cell.

In urban planning, Voronoi diagrams help in allocating resources like hospitals or schools by dividing regions according to the closest facility.

Importance:

By efficiently assigning new contracts to the closest exchange, the company ensures optimal resource utilization and service quality. Customers are connected to the nearest exchange, reducing latency and improving connection speeds.

6 Conclusion

In this integrated activity, we successfully addressed the company's needs by implementing appropriate algorithms for each problem:

1. **Minimum Spanning Tree (Kruskal's Algorithm):** Provided the optimal wiring plan with minimal total distance.

2. **Traveling Salesman Problem (Held-Karp Algorithm):** Determined the shortest possible route for mail delivery personnel using a more efficient approach.

3. **Maximum Flow (Ford-Fulkerson Algorithm):** Calculated the maximum information flow between the initial and final neighborhoods.

4. **Voronoi Diagrams:** Developed a method to efficiently assign new contracts to the nearest exchange based on geographic proximity.

Each algorithm was carefully chosen based on the problem requirements, dataset size, and computational efficiency. The complexities were analyzed to ensure that the solutions are practical and scalable for the company's context.

7 GitHub Repository

The full source code and additional materials for this integrated activity can be found in the GitHub repository:

`https://github.com/Gorchon/Advanced_Algorithms/tree/main/Integrity_Activity_2`

This repository contains all the code implementations, test cases, and supplementary documentation related to the algorithms discussed in this report.

8 Test Inputs

To thoroughly test the program and ensure its robustness, several input files were created, each designed to challenge different aspects of the code.

8.1 Input File 1: Basic Functionality Test (`input1.txt`)

Content:

```
4
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
0 16 13 10
16 0 12 14
13 12 0 18
10 14 18 0
(100,200)
(200,100)
(300,200)
(400,300)
```

Explanation:
This input tests the basic functionality of the program with a small graph of 4 nodes. It includes:

- A symmetric distance matrix with positive weights.
- A capacity matrix with varied capacities.
- Coordinates for each node.

Purpose:
To verify that the program correctly computes the MST, solves the TSP, calculates the maximum flow, and processes the coordinates.

8.2 Input File 2: Medium-Sized Graph with Zero Entries (`input2.txt`)

Content:

```
6
0 7 0 0 0 14
7 0 8 9 0 0
0 8 0 0 0 2
0 9 0 0 15 0
0 0 0 15 0 9
```

```
14 0 2 0 9 0
0 10 0 0 0 20
10 0 12 13 0 0
0 12 0 0 0 5
0 13 0 0 8 0
0 0 0 8 0 11
20 0 5 0 11 0
(50,50)
(150,100)
(200,200)
(250,150)
(300,300)
(350,250)
```

Explanation:
This input introduces a medium-sized graph with 6 nodes and includes zero entries to represent no direct connections between certain nodes.

Purpose:
To test how the program handles graphs where some nodes are not directly connected and to ensure that the algorithms still function correctly in such scenarios.

8.3 Input File 3: Larger Graph for Scalability (`input3.txt`)

Content:

```
8
0 2 0 6 0 0 0 0
2 0 3 8 5 0 0 0
0 3 0 0 7 0 0 0
6 8 0 0 9 0 0 0
0 5 7 9 0 4 0 0
0 0 0 0 4 0 5 6
0 0 0 0 0 5 0 1
0 0 0 0 0 6 1 0
0 10 0 12 0 0 0 0
10 0 8 0 9 0 0 0
0 8 0 0 7 0 0 0
12 0 0 0 11 0 0 0
0 9 7 11 0 6 0 0
0 0 0 0 6 0 7 8
0 0 0 0 0 7 0 3
0 0 0 0 0 8 3 0
(0,0)
(100,0)
(100,100)
```

```
(0,100)
(50,50)
(150,50)
(150,150)
(50,150)
```

Explanation:

A larger graph with 8 nodes is used to test the scalability of the algorithms, especially the TSP, which is computationally intensive.

Purpose:

To evaluate the program's performance with more complex data and to identify any potential issues with larger inputs.

8.4 Input File 4: Graph with Disconnected Components (`input4.txt`)

Content:

```
5
0 0 0 0 0
0 0 10 0 0
0 10 0 0 0
0 0 0 0 5
0 0 0 5 0
0 0 0 0 0
0 0 15 0 0
0 15 0 0 0
0 0 0 0 10
0 0 0 10 0
(10,10)
(20,20)
(30,30)
(40,40)
(50,50)
```

Explanation:

This input includes a graph where some nodes are not connected to the rest, creating disconnected components.

8.5 Input File 5: Large Dataset for Performance Testing (`input5.txt`)

Content:

```
20
0 34 55 2 28 51 49 17 19 76 70 45 96 20 35 58 4 8 80 86
34 0 3 61 14 99 67 29 42 31 88 95 24 54 73 11 65 13 77 90
```

```
55 3 0 39 22 81 9 26 47 62 84 78 30 69 15 18 53 7 82 93
2 61 39 0 50 44 57 27 14 85 91 36 97 6 60 71 12 21 74 89
28 14 22 50 0 68 32 38 40 59 83 87 25 46 66 16 64 5 79 92
51 99 81 44 68 0 13 70 23 60 72 94 33 55 61 10 52 9 75 88
49 67 9 57 32 13 0 41 48 63 86 79 31 68 12 19 56 6 81 94
17 29 26 27 38 70 41 0 24 82 90 37 98 5 59 72 1 22 73 87
19 42 47 14 40 23 48 24 0 84 92 35 99 7 62 70 2 20 76 85
76 31 62 85 59 60 63 82 84 0 4 89 16 47 71 13 66 15 78 91
70 88 84 91 83 72 86 90 92 4 0 93 18 49 65 14 67 16 79 95
45 95 78 36 87 94 79 37 35 89 93 0 100 9 57 73 3 23 72 84
96 24 30 97 25 33 31 98 99 16 18 100 0 52 74 17 68 12 83 96
20 54 69 6 46 55 68 5 7 47 49 9 52 0 63 69 8 24 71 83
35 73 15 60 66 61 12 59 62 71 65 57 74 63 0 21 61 17 84 98
58 11 18 71 16 10 19 72 70 13 14 73 17 69 21 0 55 14 80 91
4 65 53 12 64 52 56 1 2 66 67 3 68 8 61 55 0 25 69 88
8 13 7 21 5 9 6 22 20 15 16 23 12 24 17 14 25 0 76 90
80 77 82 74 79 75 81 73 76 78 79 72 83 71 84 80 69 76 0 5
86 90 93 89 92 88 94 87 85 91 95 84 96 83 98 91 88 90 5 0
0 16 49 24 29 35 44 10 15 70 67 40 90 22 31 53 3 7 78 81
16 0 5 57 11 94 62 25 39 27 83 88 21 50 69 8 61 9 75 87
49 5 0 35 19 76 7 22 44 58 79 71 26 64 12 15 49 4 80 92
24 57 35 0 46 39 52 22 12 80 86 33 95 6 56 67 9 18 72 84
29 11 19 46 0 64 28 34 37 54 80 82 23 42 59 13 58 5 76 88
35 94 76 39 64 0 10 65 18 55 69 89 30 51 57 9 48 8 74 86
44 62 7 52 28 10 0 38 45 60 83 75 27 63 11 17 51 6 79 93
10 25 22 22 34 65 38 0 21 78 87 34 96 5 60 68 2 19 70 85
15 39 44 12 37 18 45 21 0 79 89 32 97 7 64 66 1 16 73 82
70 27 58 80 54 55 60 78 79 0 3 85 14 44 68 12 62 13 76 89
67 83 79 86 80 69 83 87 89 3 0 90 16 46 66 13 63 14 77 94
40 88 71 33 82 89 75 34 32 85 90 0 99 8 58 70 4 21 71 83
90 21 26 95 23 30 27 96 97 14 16 99 0 49 70 16 66 11 81 95
22 50 64 6 42 51 63 5 7 44 46 8 49 0 65 71 9 22 69 82
31 69 12 56 59 57 11 60 64 68 66 58 70 65 0 20 59 15 83 97
53 8 15 67 13 9 17 68 66 12 13 70 16 71 20 0 54 12 79 90
3 61 49 9 58 48 51 2 1 62 63 4 66 9 59 54 0 23 67 86
7 9 4 18 5 8 6 19 16 13 14 21 11 22 15 12 23 0 74 88
78 75 80 72 76 74 79 70 73 76 77 71 81 69 83 79 67 74 0 6
81 87 92 84 88 86 93 85 82 89 94 83 95 82 97 90 86 88 6 0
(817,857)
(734,695)
(581,369)
(418,412)
(960,269)
(347,795)
(599,532)
```

```
(142,846)
(267,289)
(915,623)
(655,654)
(720,778)
(877,905)
(199,715)
(820,433)
(364,254)
(101,388)
(455,142)
(543,960)
(639,991)
```

Explanation:

This input file contains a large dataset with 20 nodes, designed to test the performance and scalability of the program. It includes:

- **Distance Matrix:** A 20x20 symmetric matrix with distances randomly generated between 1 and 100. The diagonal elements are zero, representing zero distance from a node to itself.
- **Capacity Matrix:** A 20x20 symmetric matrix with capacities randomly assigned between 5 and 50. The diagonal elements are zero, indicating no self-loop capacities.
- **Coordinates:** Each node is assigned coordinates within a 1000x1000 grid, simulating a large geographical area.

Purpose:

This input is selected to evaluate the program's ability to handle larger datasets efficiently. By increasing the number of nodes and the complexity of the data, we can:

- **Assess Algorithm Performance:** Test how the algorithms scale with larger input sizes, particularly for the Traveling Salesman Problem (TSP) and maximum flow computations, which are computationally intensive.
- **Identify Bottlenecks:** Detect any performance bottlenecks or memory issues that may arise with increased data volume.
- **Validate Correctness Under Load:** Ensure that the program maintains accuracy and reliability even when processing larger datasets.

Using this large input file helps to simulate real-world scenarios where networks consist of many nodes, and the efficiency of the algorithms becomes crucial.

Purpose:

To test the program's ability to handle graphs that are not fully connected and to ensure it handles such cases gracefully, possibly by indicating that a full MST is not possible.

9 References

- **Algorithms Textbooks:**
  - Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- **Online Resources:**
  - GeeksforGeeks. (2012, October 30). *Kruskal's Minimum Spanning Tree (MST) Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
  - GeeksforGeeks. (2013, November 3). *Travelling Salesman Problem using Dynamic Programming*. GeeksforGeeks. https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/
  - GeeksforGeeks. (2013, July 3). *Ford-Fulkerson Algorithm for Maximum Flow Problem*. GeeksforGeeks. https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/