WORLD AIRLINE ROUTE SEARCH SYSTEM

TASMAIYA TAMBOLI MICHAEL OLUWOLE FATEMA KHATUN

tasmaiyatamboli@my.unt.edu michaeloluwole@my.unt.edu fatemakhatun@my.unt.edu

S M MAZHARUL HOQUE CHOWDHURY GOPINATH REDDY GANGIREDDYGARI

smmazharulhoquechowdhury@my.unt.edu gopinathreddygangireddygari@my.unt.edu

April 27, 2025

Introduction

In this project, we focus on the design and implementation of algorithms for efficient storage and querying of graph-structured data, specifically in the context of airline route networks. World Airline (WA) is a hypothetical airline that operates flights to various global destinations including Moscow, Seoul, Tokyo, Hong Kong, and London. The connectivity data for these flights is provided via a synthetically generated file, `flight.txt`, which describes a directed graph where each line indicates a source city and its directly reachable destination cities.

The primary objective of this project is to create an efficient system to answer real-world-like route queries using this flight graph. To achieve this, we utilize graph algorithms and data structures that enable us to model airline connectivity and search for routes with specific constraints. The dataset is generated using a C++ program, `graphGen.cpp`, which allows flexible creation of graphs with varying sizes and complexity for robust algorithm testing.

We address the following key queries:

Q1. Given cities "A" and "B", can a passenger travel from "A" to "B" with fewer than `x` connections? If so, return the route with the smallest number of connections.

Q2. What is the shortest route (in terms of connections) from "A" to "D" that passes through both "B" and "C" (in any order)?

Q3. Given three individuals starting from cities "A", "B", and "C" respectively, determine a fourth city "D" (not equal to A, B, or C) where they can meet such that the total number of connections required is minimized. Return the optimal meeting city and routes for each person.

The output for each query follows a specified format and is printed to standard output. Testing and evaluation will be conducted on various graphs generated using the `graphGen.cpp` tool. Performance and scalability are key considerations, particularly when the dataset scales to a large number of cities and connections.

1 Design Issues and Solutions

There were choices around graph representation that were critical to making the system efficient in terms of graph traversal. One had to weigh memory use versus the speed of doing operations upon that structure. Also key was the design of the algorithms themselves–minimal connection counts meant a guarantee of performance on queries to find routes. The last challenge that came in here was the handling of very large graphs; naive approaches were not going to work due to time constraints or memory. Optimization for scale gives preference to the adjacency list with a BFS approach, allowing the system to handle big data but still perform.

1.0.1 Graph Representation

The flight data is stored in a directed graph format, where each city plays a node role and the edges stand for direct connections between cities. Two different ways of doing this were considered: There exist two applicable representations of the graph within the World Airline Route Search System with respect to the different application scenarios. The Adjacency Matrix, as defined in WA.cpp, is a structure that is directly applied, particularly in dense graphs, but space inefficient in sparse graphs since it uses $O(V^2)$ memory. This makes it very easy to use with small datasets, such as the fixed number of cities like 140-for example.

The **Adjacency List**, as defined in `routeSearch_GUI.cpp`, serves sparse graphs much better and is a more space-efficient structure that uses $\mathcal{O}(V + E)$ memory. This representation is also more advantageous during traversal operations such as BFS and DFS, which have a time complexity of $\mathcal{O}(V + E)$, compared to $\mathcal{O}(V^2)$ when using the

adjacency matrix. As a result, the adjacency list is a superior choice, especially in terms of scalability and performance during real-world route search operations.

Another reference to this graph representation is seen in `routeSearch_GUI.cpp`, where the adjacency list is implemented for its benefits in sparse graph scenarios. With a memory complexity of $\mathcal{O}(V + E)$, it significantly reduces storage requirements. Additionally, graph traversal techniques like BFS and DFS perform more efficiently with this structure, operating in $\mathcal{O}(V + E)$ time as opposed to the $\mathcal{O}(V^2)$ complexity required by the adjacency matrix. Therefore, the adjacency list remains the optimal choice for large-scale and complex airline route systems.

2 Data Structures

3 Data Structures for Efficient City and Route Management

Very effective management of cities along with their routes is done with several core data structures present to allow fast look-up and proper traversal through the graphs.

- `map<string, int>` **(City-to-ID Mapping)**:
  It is a data structure which maps the name of each city (string) to its unique number (integer). The `map` container is based on a balanced binary search tree (most likely a red-black tree), giving $O(\log N)$ time complexity with regard to insertion, deletion, and lookup. This way, the system will quickly map a name to an ID if it matters for large dataset processing where cities are referred to by index and not string. Thus, it will become much easier to look up the ID number of a city every time it is mentioned by name.

- `vector<string>` **(ID-to-City Mapping)**:
  The `vector<string>` maps each unique city ID back to its corresponding city name. This structure provides $O(1)$ constant time access, meaning that once the ID is known, it is immediately available without any computation. This is the most efficient way to facilitate conversion from IDs based on integers in a graph to string representations of cities that are human-readable.

- `vector<vector<int»` **(Adjacency List)**:
  The adjacency list is a normal representation of a graph, where a city (node) has a list of neighboring cities (edges) to which it is directly linked. This adjacency list could be stored in a `vector<vector<int»` where each inner `vector<int>` corresponds to a list of the neighboring cities for a particular city. For example, `adj[cityID]` holds all cities that can be directly reached from the city with ID `cityID`. The adjacency list is well-suited for sparse graphs; it uses memory proportional to the number of vertices plus the number of edges $O(V + E)$, and it is more space-efficient than representations like the adjacency matrix. In addition to that, it makes traversing the graph (such as BFS/DFS) optimized because all the neighbors of a city are kept in a direct list, thus making it very convenient to iterate through them.

4 Algorithms

Among the significant core operations for the World Airline Route Search System are managed by traversing graph algorithms. Task 1, requiring a shortest route return with a limit on the number of connections, a functionality provided by the Breadth-First Search, guarantees that the matched path is the best in $\mathcal{O}(V + E)$ time complexity because it deals with unweighed graphs. When a traverse path becomes longer than allowed, a search is terminated early considering there is a connection limit. To carry out any of the two countries, Task 2 deals with finding a route through two specified intermediary cities. This is done by using BFS with permutations, thus considering both likely orders (A→B→C→D and A→C→B→D) in the evaluation of each permutations being $\mathcal{O}(2(V + E))$. It has been assigned for Task 3 for determining the meeting city optimum out of three located at different starting positions. This distance it calculates in $\mathcal{O}(3(V + E))$, which precomputes every distance into each of the three cities to all others by running BFS from each of these. The meeting city is then chosen according to the minimum cumulative number of connections from all three origins, which takes $\mathcal{O}(V)$ time. These algorithmic strategies provide efficiency and scalability in route planning operations.

5 Main Logic Used in the Algorithms

5.1 Breadth-First Search (BFS) for Shortest Path (Task 1)

BFS guarantees the path with the least number of hops from the starting node to any other node in an unweighted graph. It does this by traversing all the nodes in levels and thus maintains that whenever a node is reached for the first time, the path traversed is the shortest path.

Pseudocode

```
vector<int> bfs(int start, int end, int limit = -1) {
    queue<vector<int>> q;
    set<int> visited;
    q.push({start});

    while (!q.empty()) {
        vector<int> path = q.front(); q.pop();
        int current = path.back();

        if (limit != -1 && path.size() - 1 > limit) continue;
        if (current == end) return path;
        if (visited.count(current)) continue;

        visited.insert(current);

        for (int neighbor : adj[current]) {
            if (find(path.begin(), path.end(), neighbor) == path.end()) {
                vector<int> newPath = path;
                newPath.push_back(neighbor);
                q.push(newPath);
            }
        }
    }
    return {}; // No path found
}
```

5.2 Finding a Path Through Two Intermediate Cities (Task 2)

In general, we want to find the shortest valid combination of A → B → C → D or A → C → B → D given the condition that the path A → D must pass through B and C in any order. Algorithm: We apply Breadth-First Search to calculate two independent paths for each segment, that is, first from A to B to C to D, and second from A to C to B to D. After both alternatives are computed, we compare to get the shortest valid one. Key Logic: The algorithm checks for both possible permutations of the intermediate cities (A → B → C → D and A → C → B → D), concatenating both valid segments while removing any duplicate cities found in both paths so that what remains is the shortest path.

Pseudocode

```
vector<int> findThroughPath(int start, int mid1, int mid2, int end) {
    vector<vector<int>> options;
    vector<vector<int>> orders = { {mid1, mid2}, {mid2, mid1} };

    for (auto& order : orders) {
        vector<int> path1 = bfs(start, order[0]);
        vector<int> path2 = bfs(order[0], order[1]);
        vector<int> path3 = bfs(order[1], end);

        if (!path1.empty() && !path2.empty() && !path3.empty()) {
            path1.pop_back(); // Remove duplicate node
            path2.pop_back(); // Remove duplicate node
            path1.insert(path1.end(), path2.begin(), path2.end());
            path1.insert(path1.end(), path3.begin(), path3.end());
            options.push_back(path1);
        }
    }

    if (options.empty()) return {};
    return *min_element(options.begin(), options.end(),
    [](const vector<int>& a, const vector<int>& b) {
        return a.size() < b.size();
    });
}
```

5.3 Finding the Optimal Meeting City (Task 3)

Problem Breakdown: Three persons located in A, B, and C want to meet at some other place D with the minimum cost to all in terms of connecting with one another. Given this situation, the object is to find that city D which

minimizes the total distance of travel for all three. Algorithm Choice: To do this, BFS is done from all three cities A, B, and C towards every possible meeting city D (excluding A, B, and C). For every possible city D, the shortest path from A→D, B→D, and C→ D is calculated. The sum of these will then be represented and the city with the minimal total will be determined as the best meeting point. Optimization: Early termination is employed to bypass any city D that is unreachable from one or more of the cities A, B, or C. Also suggested is memoization to cache the results from BFS carried out for each city in order to further speed up the algorithm by avoiding duplicate computations.

Pseudocode

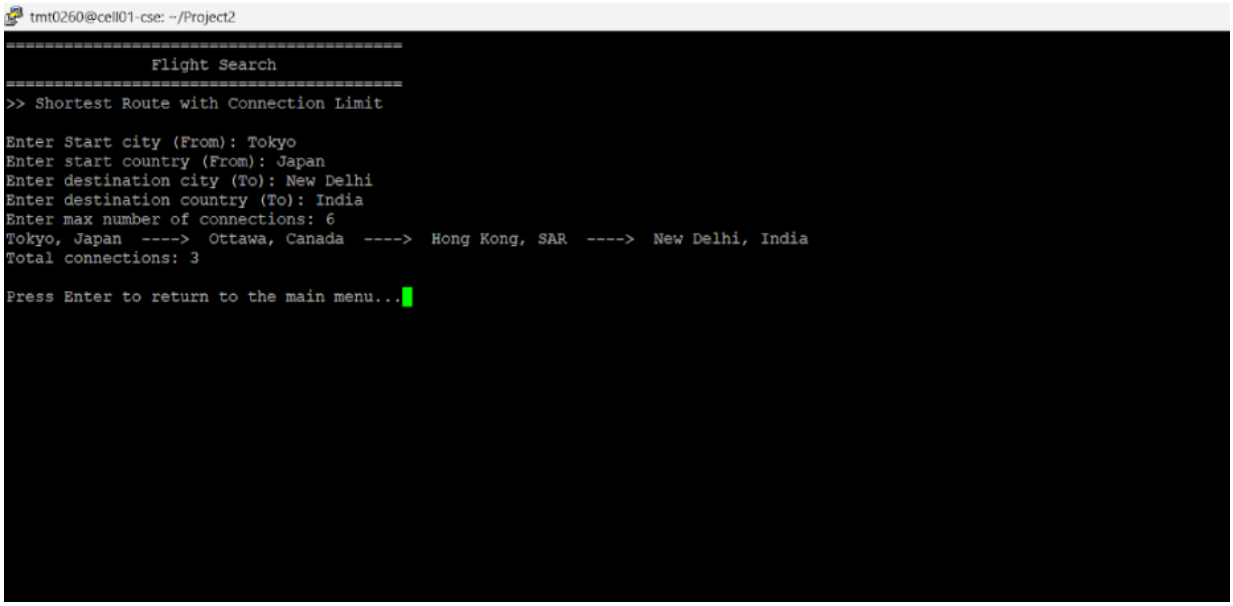
```
void findMeetingCity(int a, int b, int c, int n) {
    int minTotal = INT_MAX, bestCity = -1;

    for (int city = 0; city < n; city++) {
        if (city == a || city == b || city == c) continue;

        vector<int> pa = bfs(a, city);
        vector<int> pb = bfs(b, city);
        vector<int> pc = bfs(c, city);

        if (!pa.empty() && !pb.empty() && !pc.empty()) {
            int total = (pa.size() - 1) + (pb.size() - 1) + (pc.size() - 1);
            if (total < minTotal) {
                minTotal = total;
                bestCity = city;
            }
        }
    }

    if (bestCity == -1) {
        cout << "No meeting city found.\n";
        return;
    }

    cout << "Meet at: " << idToCity[bestCity] << "\n";
    cout << "Person A: "; printPath(bfs(a, bestCity));
    cout << "Person B: "; printPath(bfs(b, bestCity));
    cout << "Person C: "; printPath(bfs(c, bestCity));
    cout << "Total connections: " << minTotal << "\n";
}
```

6 Running Time Analysis

The system executes the routing calculation with an efficient road map of BFS, running in $\mathcal{O}(V+E)$ time per search, allowing it to perform excellently in large graphs. For multiple source domains, such as the meeting point of three travelers, a multi-source BFS is performed with a combined complexity $\mathcal{O}(3(V+E))$, which is still linear and scalable. In these attributes, the system handles very large graphs with thousands of cities and connected flight routes without compromising efficiency.

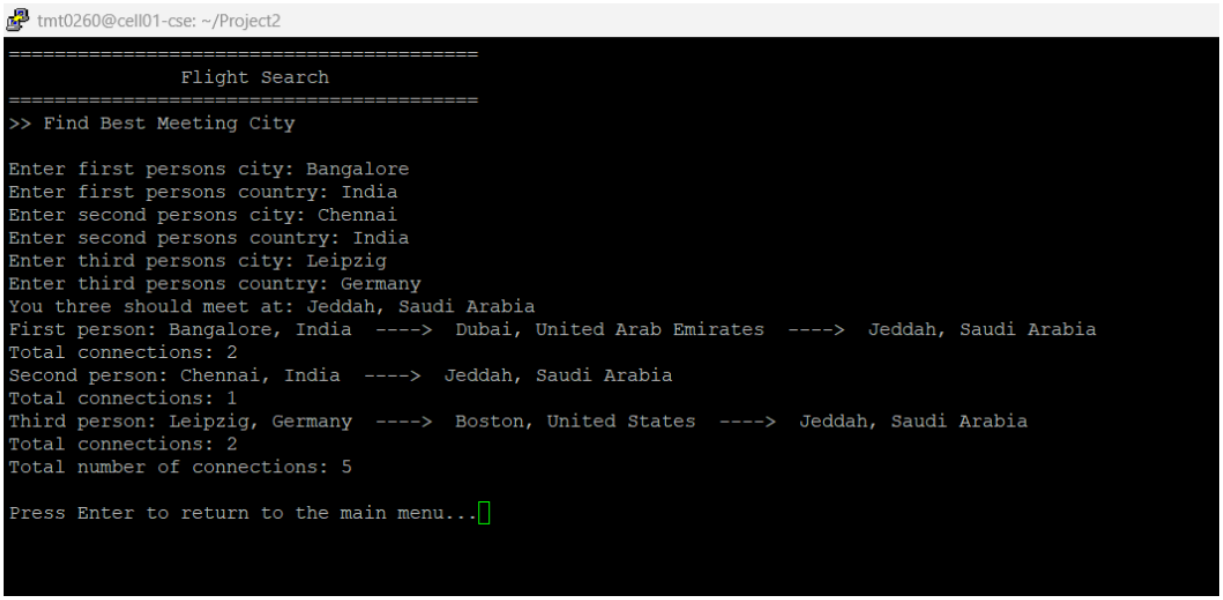
7 Optimization Issues

The system incorporated a number of key optimizations to enhance performance. First, in BFS, early termination can stop the exploration of further paths when the connection limitation is overridden, thus aborting unnecessary computation. Even though it is not currently implemented, we recommend future enhancement in memoization, where BFS results can become cached to make repeat queries, thereby offering great efficiency gains. Another area of potential improvement would be parallel BFS, ideally for task 3: running searches simultaneously from cities A, B, and C could speed up the search for an optimal rendezvous point. From a scalability perspective, it uses an adjacency list that allows it to work efficiently on very large graphs—for example, those with several million cities and routes. That's not without drawbacks; the BFS may experience performance bottlenecks on extremely large graphs due to its $\mathcal{O}(V)$ memory usage for queue allocation. Therefore, smarter techniques like bidirectional BFS or the A-star algorithm could be considered for weighted graph cases



(a) Example output for Query 1: Limited Connections

(b) Example output for Query 2: Path through two cities



(c) Example output for Query 3: Optimal meeting city

Figure 1: Screenshots of All Queries in the Route Search System

8 Conclusion

The system efficiently solves the three routing problems using BFS and adjacency lists, ensuring scalability and quick pathfinding. The use of BFS allows for optimal performance in finding the shortest path while leveraging adjacency lists to handle large graphs efficiently. Future improvements to the system include adding edge weights to incorporate flight durations, which would enable the use of more advanced algorithms like Dijkstra's or A* for weighted graphs. Additionally, parallelizing BFS for large-scale graphs is a potential enhancement that could significantly reduce computation time and improve the system's scalability in real-world scenarios.