**Application: RouteTrixs**

**Developed Date:  22/10/2024**

**Developer By: Oluropo Akinbolade-Jones**

**Email: ropojones@gmail.com**

**Github: https://github.com/ropojones**

**RouteTrix on Github: https://github.com/ropojones/RouterTrixs**

**Overview**

The program was designed to showcase my understanding of the fundamentals of building a modern applications that scales to requirement. Though a technical assignment, I tried to demonstrate the understanding of the key concepts by ensuring the program follows clean architecture principles, including Dependency Injection (DI) to manage services, ensuring that the logic is modular, testable, and maintainable. It also highlights a deep consideration for user experience. Though a console application, I tried to ensure that it is interactive, intuitive and instructive enough to guide the users in navigating through the application. I have cultivated a habit of paying attention to detail and I hope this spotlights my skill as a Software Developer and lands me the job.

The code implements a clean architecture-based system to compute various operations related to academy routes in different regions, such as distance calculations, route counting, and finding the shortest route.

**1. Program Structure**

The architecture separates concerns into different parts:

**Interfaces (IRouteService):** Defines contracts or abstract behaviors (e.g., calculating distances, counting routes). This allows different implementations to be swapped out without changing the rest of the code.

Implementation (**RouteService**): Provides the actual logic to implement the defined interface methods.

Dependency Injection (Program.cs): Centralized configuration to inject dependencies, making the services reusable and testable without hardcoded instantiation.

**2. Main Entities**

**IRouteService**

The **IRouteService** interface defines the following methods:

**CalculateRouteDistance(string route)**: Given a sequence of academies (e.g., A-B-C), it calculates the total distance between them.

**CountRoutesWithMaxStops(string start, string end, int maxStops)**: Counts the number of routes between two academies with no more than a specified number of stops.

**CountRoutesWithExactStops(string start, string end, int exactStops)**: Counts routes with exactly the given number of stops.

**FindShortestRoute(string start, string end):** Finds the shortest distance between two academies.

**CountRoutesWithMaxDistance(string start, string end, int maxDistance)**: Counts the number of different routes between two academies where the total distance is below a given threshold.

Each method addresses a specific problem related to route calculation. These methods are designed in a way that makes it easy to implement various algorithms.

**RouteService**

**RouteService** implements the logic defined in the **IRouteService**. It takes a list of tuples as input, where each tuple represents a directed route between two academies and their corresponding distance.

The data structure used to store routes is a graph: A dictionary of dictionaries (\_routeMap). This allows us to store the routes as adjacency lists, where each node (academy) points to its connected nodes (neighboring academies) along with the distance.

private readonly Dictionary<(string, string), int> \_routeMap = new Dictionary<(string, string), int>();

        public RouteService(IEnumerable<(string, string, int)> routes)

        {

            foreach (var route in routes)

            {

                \_routeMap[(route.Item1, route.Item2)] = route.Item3;

            }

        }

This structure is useful for fast lookups and traversal in algorithms like **Breadth-First Search (BFS)** or **Dijkstra's algorithm** for shortest path finding.

**3. Algorithms and Logic**

**3.1. CalculateRouteDistance**

This method computes the total distance for a specific route (e.g., A-B-C):

1. Parse the route sequence.
2. Traverse the graph and accumulate the distances between consecutive academies.
3. If any part of the route doesn’t exist, return a message like NO SUCH ROUTE.

 public string CalculateDistance(params string[] academies)

        {

            int totalDistance = 0;

            for (int i = 0; i < academies.Length - 1; i++)

            {

                var key = (academies[i], academies[i + 1]);

                if (\_routeMap.ContainsKey(key))

                {

                    totalDistance += \_routeMap[key];

                }

                else

                {

                    return "NO SUCH ROUTE";

                }

            }

            return totalDistance.ToString();

        }

**Algorithm:**

* Traverse the graph using the route's academies. Time complexity is O(n) where n is the number of route stops.

**3.2. FindShortestRoute**

To find the shortest route between two academies, I used the **Breadth-First Search (BFS)** since the graph is small.

  private int FindShortestRoute(string current, string end, HashSet<string> visited, int currentDistance)

        {

            if (current == end && visited.Count > 0)

            {

                return currentDistance;

            }

            visited.Add(current);

            int shortest = int.MaxValue;

            foreach (var route in \_routeMap.Where(r => r.Key.Item1 == current))

            {

                if (!visited.Contains(route.Key.Item2) || route.Key.Item2 == end)

                {

                    int distance = FindShortestRoute(route.Key.Item2, end, new HashSet<string>(visited), currentDistance + route.Value);

                    if (distance < shortest)

                    {

                        shortest = distance;

                    }

                }

            }

            return shortest;

        }

**Algorithm:**

* Initializes all distances as "infinite."
* Uses BFS/DFS to explore routes and update the shortest known distance to each neighboring node.
* Time complexity is O(E + V log V) (if using priority queue for Dijkstra's). E is the number of edges, V is the number of vertices.

**3.3. CountRoutesWithMaxStops, CountRoutesWithExactStops, CountRoutesWithMaxDistance**

This algorithm uses **Depth-First Search (DFS)** or **Backtracking** to explore all possible paths between two academies and count the ones that satisfy the condition of having up to a certain number of stops (Max or Exact or MaxStopsLessThan).

        public int CountTripsWithMaxStops(string start, string end, int maxStops)

        {

            return CountTrips(start, end, 0, maxStops);

        }

        public int CountTripsWithExactStops(string start, string end, int exactStops)

        {

            return CountTrips(start, end, 0, exactStops, exactStops);

        }

        private int CountTrips(string current, string end, int stops, int maxStops, int? exactStops = null)

        {

            if (stops > maxStops) return 0;

            if (exactStops.HasValue && stops == exactStops && current == end) return 1;

            if (!exactStops.HasValue && stops <= maxStops && current == end && stops > 0) return 1;

            int count = 0;

            foreach (var route in \_routeMap.Where(r => r.Key.Item1 == current))

            {

                count += CountTrips(route.Key.Item2, end, stops + 1, maxStops, exactStops);

            }

            return count;

        }

        public int CountRoutesWithMaxDistance(string start, string end, int maxDistance)

        {

            return CountRoutesWithMaxDistanceHelper(start, end, 0, maxDistance);

        }

        private int CountRoutesWithMaxDistanceHelper(string current, string end, int currentDistance, int maxDistance)

        {

            if (currentDistance >= maxDistance) return 0;

            int count = 0;

            if (current == end && currentDistance > 0)

            {

                count++;

            }

            foreach (var route in \_routeMap.Where(r => r.Key.Item1 == current))

            {

                count += CountRoutesWithMaxDistanceHelper(route.Key.Item2, end, currentDistance + route.Value, maxDistance);

            }

            return count;

        }

**Algorithm:**

* Recursively explore routes starting from the current node.
* If the number of stops exceeds maxStops, stop that branch of exploration.
* Time complexity depends on the number of routes possible, which can grow exponentially based on the number of routes.

**4. Dependency Injection Pattern**

In the Program.cs, the **HostBuilder** pattern is used to configure Dependency Injection. This is where the service (like RouteService) is registered to be injected into different parts of the application.

// Setting up a HostBuilder to configure Dependency Injection

            var host = Host.CreateDefaultBuilder()

                .ConfigureServices((context, services) =>

                {

        // Registering the IRouteService and RouteService

         services.AddSingleton<IRouteService, RouteService>();

                })

                .Build();

        // Resolves and runs the main application

        var routeService = host.Services.GetRequiredService<IRouteService>();

* **services.AddSingleton<IRouteService, RouteService>()**: This registers RouteService as a singleton instance for IRouteService. This ensures that any time the application needs an IRouteService, it will receive the same instance of RouteService.
* **host.Services.GetRequiredService<IRouteService>()**: The DI container is queried to get an instance of IRouteService, which is then used in the application logic.

**My Summary:**

* **Graph-based traversal**: Uses dictionaries to represent a graph where the nodes are academies and the edges are routes with distances.
* **Shortest route calculation**: Uses BFS/Dijkstra to find the shortest distance between two academies.
* **Route counting**: Uses DFS/Backtracking to explore all possible routes within specific conditions (e.g., maximum stops or maximum distance).
* **Dependency Injection**: A central pattern in clean architecture to decouple service instantiation from the logic using it. This is implemented using Microsoft.Extensions.DependencyInjection.

This structure ensures flexibility, testability, and scalability of the codebase.