Traveling Salesman Problem (TSP) Solver with GUI for Egyptian Cities

---

Project Title: TSP Solver with GUI for Egyptian Cities

Course: Algorithm

Team Members and Responsibilities:

Abanoub Amir George (2301001) - Dynamic Programming (Held-Karp) Algorithm Implementation

Mazen ahmed mohamed (2301178) - GUI Development (Qt Interface)

Mohamed ALaa Mohamed (2301205) - GUI Development (Qt Interface)

Mohamed Ahmed Khadre Alkassas (2301327) - Implementation of the Brute Force Algorithm

Mahmoud Abdelghany Rabea (2301226) - Animation Logic and Visual Route Display

Ahmed Atef Elpery (2301019) - Haversine Distance Calculation and City Management

Ahmed Kamel Hassanin (2301030) - Greedy Approximation Algorithm and Integration

ahmed mohamed ahmed aboughazal (2301034) – PowerPoint Presentation

Ahmed Mohmed Ebrahem (2301031) – The Report

---

1. Requirement Elicitation and Analysis

Functional Requirements:

- Add/Edit/Delete cities with latitude and longitude.
- Display cities on a 2D map.
- Select algorithm (Brute Force, DP, Greedy).
- Visualize and animate the TSP solving process.
- Display total distance and final route.

Non-Functional Requirements:

- User-friendly GUI using Qt.
- Efficient computation for reasonable number of cities.
- Accurate distance calculation using Haversine formula.

Input Requirements:

- City name
- Latitude and Longitude coordinates

Output Requirements:

- Path order of cities
- Total distance of the route

Algorithmic Requirements:

- Brute Force: Compute all permutations of cities.
- Dynamic Programming: Held-Karp algorithm for optimized TSP.
- Greedy: Nearest Neighbor algorithm.

---

2. Design Overview

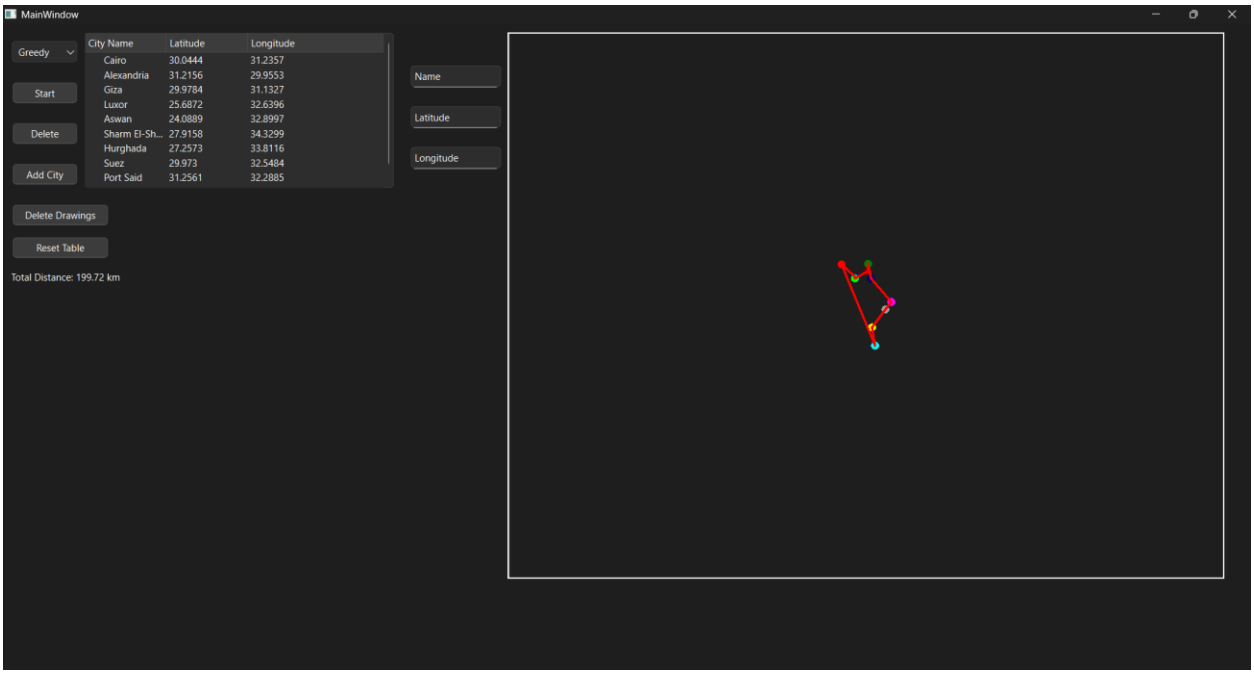
Architecture:

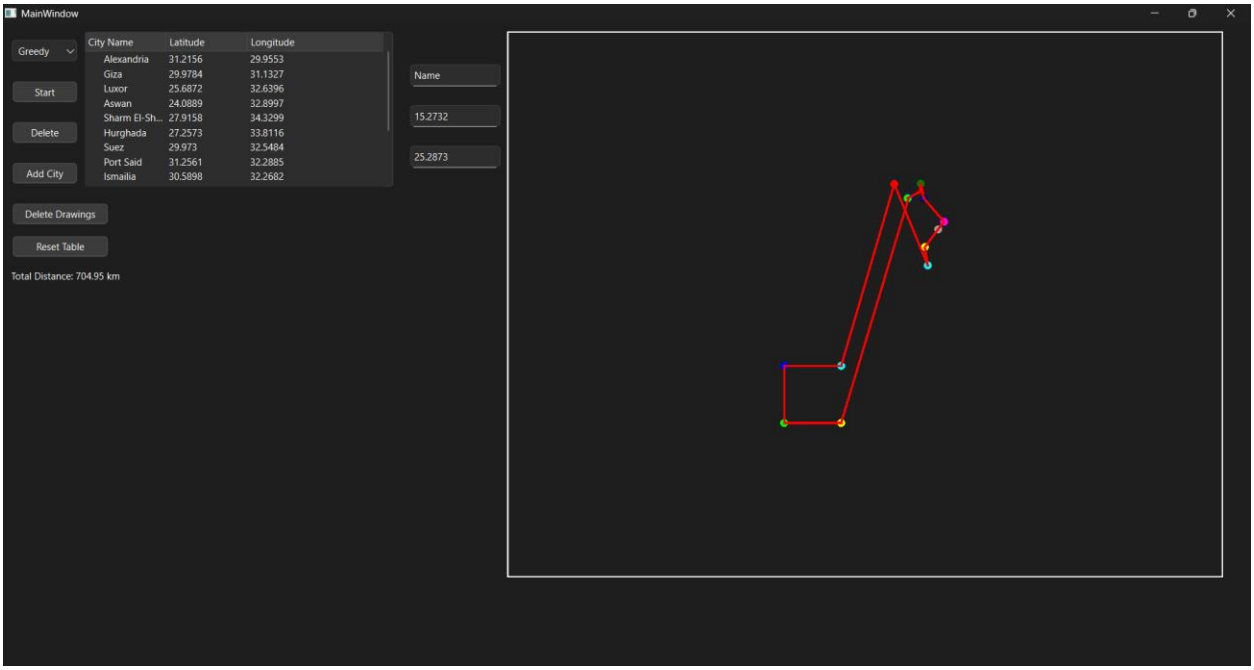
- Modular structure with clear separation between UI, algorithms, and data models.
- MVC-style layout using Qt for GUI and logic separation.

GUI Layout:

- **Left Panel**: Algorithm selector and control buttons (Start, Reset)
- **Top Panel**: Display total distance and current status
- **Main Panel**: Map with city points, table of cities with edit/delete, and form to add new cities

3. Screenshots from the App:





---

5. Code Snippets

Greedy Algorithm (Nearest Neighbor):

```
QVector<QPointF> MainWindow::runBruteForce(const QVector<QPointF>& cities) {
    QVector<QPointF> path;
    int n = cities.size();
    QVector<int> permutation(n);
    std::iota(permutation.begin(), permutation.end(), 0);

    double minDistance = std::numeric_limits<double>::max();
    QVector<int> bestPermutation;

    do {
        double totalDistance = 0;
        for (int i = 0; i < n - 1; ++i) {
            totalDistance += calculateDistance(cities[permutation[i]], cities[permutation[i + 1]]);
        }
        totalDistance += calculateDistance(cities[permutation[n - 1]], cities[permutation[0]]); // Return to the start city

        if (totalDistance < minDistance) {
            minDistance = totalDistance;
            bestPermutation = permutation;
        }

    } while (std::next_permutation(permutation.begin(), permutation.end()));

    // Construct the path using the best permutation
    for (int idx : bestPermutation) {                    ⚠ c++11 range-loop might detach Qt container (QList) [clazy-range-loop-detach]
        path.append(cities[idx]);
    }

    return path;
}
```

Brute Force Algorithm:

```
QVector<QPointF> MainWindow::runGreedyAlgorithm(const QVector<QPointF>& cities) {
    QVector<QPointF> path;
    if (cities.isEmpty()) return path;

    QVector<bool> visited(cities.size(), false);
    path.append(cities[0]);
    visited[0] = true;

    for (int i = 1; i < cities.size(); ++i) {
        double minDistance = std::numeric_limits<double>::max();
        int nextCityIndex = -1;
        const QPointF& currentCity = path.last();

        for (int j = 0; j < cities.size(); ++j) {
            if (!visited[j]) {
                double distance = calculateDistance(currentCity, cities[j]);
                if (distance < minDistance) {
                    minDistance = distance;
                    nextCityIndex = j;
                }
            }
        }

        if (nextCityIndex >= 0) {
            visited[nextCityIndex] = true;
            path.append(cities[nextCityIndex]);
        }
    }

    return path;
}
```

Dynamic Programming (Held-Karp Algorithm):

```
QVector<QPointF> MainWindow::runDynamicProgramming(const QVector<QPointF>& cities) {
    int n = cities.size();
    QVector<QVector<double>> dist(n, QVector<double>(n));

    // Precompute distances between all pairs of cities
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            dist[i][j] = calculateDistance(cities[i], cities[j]);
        }
    }

    // DP table, where dp[mask][i] is the shortest path to visit all cities in 'mask' ending at city 'i'
    QVector<QVector<double>> dp(1 << n, QVector<double>(n, std::numeric_limits<double>::max()));
    dp[1][0] = 0; // Start at city 0

    // DP to calculate the minimum cost
    for (int mask = 1; mask < (1 << n); ++mask) {
        for (int u = 0; u < n; ++u) {
            if ((mask & (1 << u)) == 0) continue;
            for (int v = 0; v < n; ++v) {
                if (mask & (1 << v)) continue;
                dp[mask | (1 << v)][v] = std::min(dp[mask | (1 << v)][v], dp[mask][u] + dist[u][v]);
            }
        }
    }

    // Reconstruct the path from the DP table
    int mask = (1 << n) - 1;
    int last = 0;
    QVector<QPointF> path;
    for (int i = 1; i < n; ++i) {
        if (dp[mask][i] < dp[mask][last]) {
            last = i;
        }
    }
```

---

4. Implementation Summary

Languages & Libraries:

- C++
- Qt for GUI
- Standard Template Library (STL)

Algorithms:

- **Brute Force**: Generates all permutations and selects the one with minimum total distance.
- **Held-Karp (DP)**: Uses bitmasking and memoization to reduce complexity to O(n^2 * 2^n).
- **Greedy**: Starts from a city and always visits the nearest unvisited city.

Distance Calculation:

- Haversine formula was used for accurate geographical distance between latitude and longitude.

Visualization:

- Cities represented as colored nodes
- Paths animated using QGraphicsView

---

5. Conclusion

This application successfully solves the TSP for a given set of Egyptian cities using three different algorithms. It offers a user-friendly GUI and allows users to visualize and compare algorithm performance interactively. The modular code structure ensures extensibility and clarity.