XI'AN JIAOTONG-LIVERPOOL UNIVERSITY

西交利物浦大学

COURSEWORK SUBMISSION COVER Page

Group Number	Group 82					
Students' ID Number	2251576	2255705				
Module Code	CPT204					
Assignment Title	CPT204-Coursework 3					
Submission Deadline	2025.05.18					

By uploading this coursework submission with this cover page, we certify the following:

- ❖ We have read and understood the definitions of collusion, copying, plagiarism, and dishonest use of data as outlined in the Academic Integrity Policy of Xi'an Jiaotong-Liverpool University.
- ❖ This work is **entirely** our own, original work produced specifically for this assignment. It does not misrepresent the work of another group or institution as our own.
- ❖ This work is not the product of unauthorized collaboration between ourselves and others.
- ❖ This work has not been shared wholly or in part outside of the group.
- ❖ It is a submission that has not been previously published, or submitted to any modules.
- ❖ Students who would like to submit the same or similar work from previous years to the current module or other modules must receive written permission from all instructors involved in advance of the assignment due date.
- ❖ **All** group members are **equally** and **collectively** responsible for the **entire** submission. Violations of academic integrity including failure to monitor group member contribution originality constitutes negligence.
- ❖ Unreported suspicious academic misconduct by one group member will be attributed to ALL members in terms of penalties. **ALL members share the responsibilities**.
- ❖ Use of generative AI is strictly prohibited for all assessments involved in this module.

We understand collusion, plagiarism, dishonest use of data, and submission of procured work are serious academic misconducts. **All** group members are held **jointly accountable** for the integrity of the **entire** submission. By uploading or submitting this cover page, we acknowledge that we are jointly subject to penalties and disciplinary actions if we are found to have committed such acts.

~~~~~~

We acknowledge that the university late submission policy will be applied if applicable.

~~~~~~

Please list the ID number of any group member NOT contributing to the submission:

____

Please indicate whether based on your individual contributions you meet the learning outcomes covered by this coursework and confirm your submission meets with all above requirements by signing your name, handwritten in ink, in English (Foreign students) or PINYIN (local students):

Student ID	✓(tick) box to confirm you contributed to all cw tasks	✓(tick) box to confirm you meet all LOs covered by this cw	Signed
2251576	☑	☑	Rui.Sang
2255705	☑	☑	Yilin.Li
	☐	☐	
	☐	☐	

Date: 2025.05.18

1. Chapter 1: Task A - Program Design

1.1 System Architecture Overview

Our USA Road Trip Planner system follows a layered architecture with clear separation of concerns, organized into three main components:

1. **View Layer :** The graphical user interface that handles user interactions and displays results. This layer includes classes like `PathFinderGUI`, `MapPanel`, `ControlPanel` and `ResultsPanel`.
2. **Model Layer :** The core business logic, containing the pathfinding algorithms and data structures. This includes domain components like `PathfindingModel`, `PathPlanner`,`Graph` `PathResult`,control layer :`PathFinderController` and data layer for parsing and storing data from files and build graph.
3. **Algorithm implementations Layer :** All the algorithms involved are implemented here, two basic algorithms for calculating the shortest path between two points(`DijkstraPathfinder&AStarPathfinder)`, and two algorithms for finding the optimal path sequence(`NN2optpathfinder` and `TspPathfinder`).

The *data flow* through the system can be visualized using the *sequence diagram*:

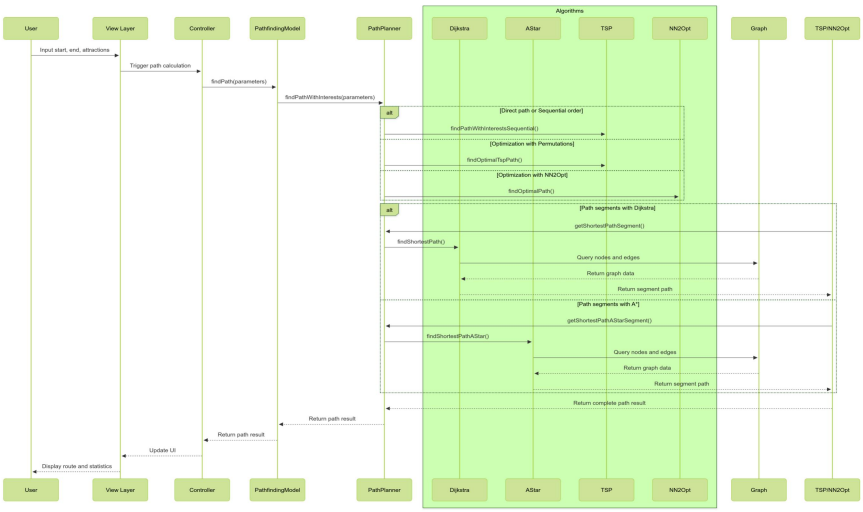


Figure.1 Sequence diagram of the data flow in this

This architecture ensures that changes to one layer can be made without impacting other layers, promoting maintainability and extensibility.

1.2 Data Structure Selection and Justification

1.2.1 `attractions.csv` Data Representation

For attraction data we use a combination of HashMap structures to facilitate efficient lookups. The primary structure is a `HashMap<String, String>` that maps attraction names to their corresponding city names(Fig 2.(a)

```
public static Map<String, String> loadAttractionData(String filePath) {
    Map<String, String> attractions = new HashMap<>();

    try (BufferedReader br = new BufferedReader(new
FileReader(filePath))) {
        // Skip header
        String line = br.readLine();

        while ((line = br.readLine()) != null) {
            String[] parts = line.split(",");
            if (parts.length == 2) {
                String attraction = parts[0];
                String location = parts[1];
                attractions.put(attraction, location);
            }
        }
    } catch (IOException e) {
        System.err.println("Error reading attraction data: " +
e.getMessage());
    }

    return attractions;
}
```

(a) . Hashmap for 'Attraction'

```
public static List<RoadConnection> loadRoadData(String filePath) {
    List<RoadConnection> roads = new ArrayList<>();

    try (BufferedReader br = new BufferedReader(new
FileReader(filePath))) {
        // Skip header
        String line = br.readLine();

        while ((line = br.readLine()) != null) {
            String[] parts = line.split(",");
            if (parts.length == 3) {
                String cityA = parts[0];
                String cityB = parts[1];
                int distance = Integer.parseInt(parts[2]);
                roads.add(new RoadConnection(cityA, cityB, distance));
            }
        }
    } catch (IOException e) {
        System.err.println("Error reading road data: " + e.getMessage());
    }

    return roads;
}
```

(b) . Arraylist for 'Attraction'

```
public static class RoadConnection {
    private String cityA;
    private String cityB;
    private int distance;

    public RoadConnection(String cityA, String cityB, int
distance) {
        this.cityA = cityA;
        this.cityB = cityB;
        this.distance = distance;
    }

    public String getCityA() {
        return cityA;
    }

    public String getCityB() {
        return cityB;
    }

    public int getDistance() {
        return distance;
    }
```

(c) . Inner class for 'Road connection'

Figure.2 Data structure selection from CSV file

Following are our consideations:

· **Lookup Efficiency :** Finding the city associated with an attraction name is an O (1)operation, which is crucial for processing user-selected attractions

· **Simplicity :** For the purposes of this application, we only need to know which city contains a given attraction, making a direct mapping sufficient.

· **Memory Efficiency :** A simple mapping uses less memory than creating complex objects for each attraction when only the location information is needed.

1.2.2 `roads.csv` Data Representation

The data structure for reading is a `List<DataLoader.RoadConnection>`.(Fig 2.(b)) Our `DataLoader` class use an `ArrayList` to store road information read line-by-line from a CSV file. A static inner class, `RoadConnection`, is defined to encapsulate each row's data(Fig 2.(c)) . This class includes private fields for `cityA(String)`, `cityB(String)`, and `distance(int)`, along with their corresponding getter methods, facilitating data transfer to the `Graph` class for road network construction.

1.2.3 `Graph.class` Representation

The road network is represented by an `adjacency list` through the Graph class. This structure uses a `HashMap<String, Map<String, Integer>>` where:

The outer map's key is a city name and the value is another map where keys are neighboring city names and values are the distances.Following are our considerations:

· **Sparse Graph Efficiency :** The US road network is a sparse graph (each city connects to all others), making an adjacency list more space-efficient than an adjacency matrix(Table.1).

Representation	Space Complexity	Neighbor Access	Edge Lookup	Memory Usage
Adjacency List	O(V+E)	O(1)	O(deg(v))	low
Adjacency Matrix	$O(V^2)$	O(V)	O(1)	high

Table.1 The comparison between Adjacency List and Adjacency matrix

· **Neighbor Access Performance :** Pathfinding algorithms frequently need to access a node's neighbors, which is an O(1) operation in this structure.

· **Edge Weight Representation :** The integer values naturally represent road distances, essential for calculating the shortest path.

· **Dynamic Nature :** The adjacency list easily accommodates adding new cities and roads without restructuring the entire graph.

1.3 Class Structure and Interactions

Our USA Road Trip Planner system follows a modular design organized around the Model-View-Controller (MVC) pattern. We will present the key classes in each module and explain how they interact to provide the system' s functionality.

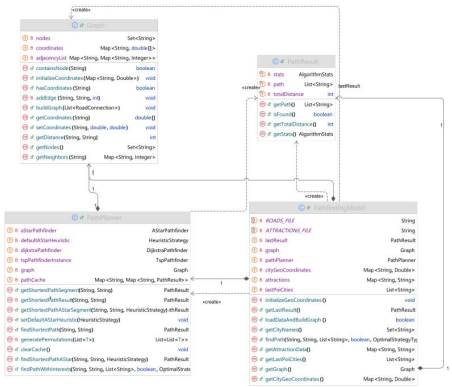


Figure.3 Class Diagram of the Model Components

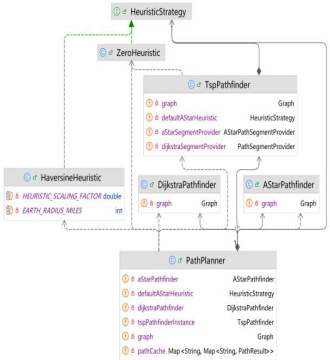


Figure 4:Class Diagram of the Algorithm Implementations

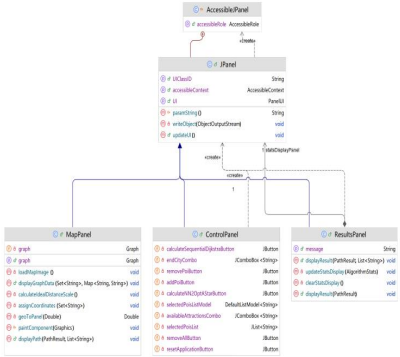


Figure 5: UML Class Diagram of the View Components

1.3.1 Model Classes

The Model module encapsulates the data structure and business logic of our road trip planning system(Fig.3). The key classes in this module include:

`1.PathfindingModel` : serves as the facade for all model operations, coordinating interactions through:

- `loadDataAndBuildGraph()`: Initializes the graph from CSV files

- `findPath(...)`: Coordinates pathfinding with configurable algorithm parameters
- Various `getter` methods that provide data access to the Controller

2.Graph: represents the road network as an adjacency list, with cities as nodes and roads as weighted edges. Its core functionality is exposed through:

- `buildGraph(roads)`: Constructs the graph from a list of road connections.
- `getNeighbors(node)`: Returns adjacent cities with their distances.
- `getDistance(source, destination)`: Returns the distance between two directly connected cities.

3.PathPlanner: mediates between model and specific algorithm implementations:

- `findPathWithInterests(...)`: Unified method for finding paths through multiple points of interest.
- `getShortestPathSegment(from, to)`: Provides Dijkstra-calculated path segments.
- `getShortestPathAStarSegment(from, to, heuristic)`: Provides A-calculated path segments.

PathPlanner employs a *caching mechanism* to avoid redundant calculations when evaluating different route permutations.

4.PathResult: encapsulates the outcome of a path calculation, including the city sequence, total distance, and algorithm statistics. Its simple getter methods provide access to these results while *hiding* the implementation details.

1.3.2 Algorithm Implementation Classes

This moudle contains classes that implement various pathfinding algorithms, from basic shortest path finding to optimal access order finding(Fig.4).

1.DijkstraPathfinder: implements the classic Dijkstras algorithm for finding the shortest path between two cities. Its primary method findShortestPath(start,end) uses a `priority queue` to efficiently track nodes by current known distance:

2.AStarPathfinder: extends Dijkstra ' s approach with heuristics through findShortestPathASta end, heuristicStrategy), improving search efficiency by guiding

toward the goal.

3.TspPathfinder: implements an exact solution for the Traveling Salesperson Problem aspect of finding optimal routes through several points of interest. :

- `findOptimalTspPath(...)` and `findOptimalTspPathAStar(...)`: Implement the exact solution using permutations of all possible routes, guaranteeing the globally optimal path.
- `findPathWithInterestsSequential(...)` and `findPathWithInterestsSequentialAStar(...)`: Implement the direct sequential visiting approach without optimization.
- `findPathWithInterestsSequential(...)` and `findPathWithInterestsSequentialAStar`: Implement the direct sequential visiting approach.

4.HeuristicStrategy: defines the interface for *A* heuristics*, with implementations like HaversineHeuristic (calculates geographic distance) and ZeroHeuristic (reduces A* to Dijkstra).

1.3.3 View Classes

The View module contains classes that handle the user interface(Fig.5) and visualization aspects of the system(Fig.6).

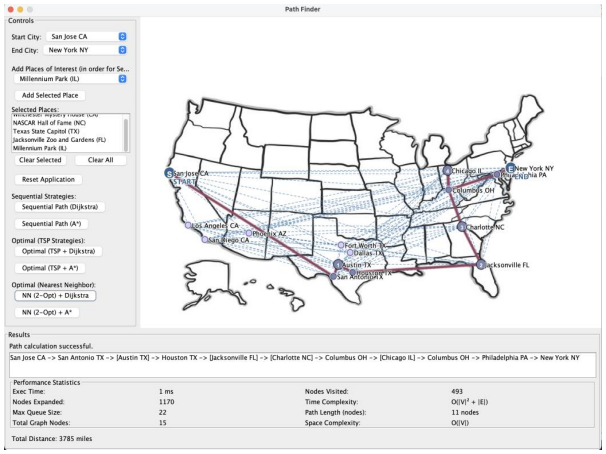


Figure.6 System UI panel and path

1.MapPanel visualizes the road network and calculated paths on a geographic map. Its key methods include:

- `setGraph(graph)`: Initializes the panel with a graph to display.
- `displayPath(result, poiCities)`: Updates the display with a calculated path and highlights points of interest.
- `paintComponent(g)`: Overrides JPanels painting method to render the map with

cities, roads, and the current path.

2.`ControlPanel` provides UI elements for user input:

- `populateCities(cities)` and `populateAttractions(attractions)`:Initialize UI components
- Various `getter` methods allow controller to access UI and attach listeners.

3.`ResultsPanel` ResultsPanel displays calculation results and statistics:

- `displayResult(result, poiCities)`: Formats and presents the path with POI highlighting.
- `updateStatsDisplay(stats)`:Shows algorithm performance metrics

A key feature is the POI highlighting in path display(Fig.7):

```
// Highlight POIs (not start/end points) in square brackets
if (poiCities.contains(city) && i > 0 && i < path.size() - 1) {
sb.append("[").append(city).append("]");
} else {
sb.append(city);
}
```

Figure.7 The feature display of UI

1.3.4 Controller and Component Interactions

The `PathfindingController` bridges Model and View components through:

- `initializeViewData()`: Populates UI with model data
- `attachListeners()`: Connects UI events to handling methods
- `handlePathCalculationRequest(...)`: Processes user inputs

The typical model interaction proceeds as follows:

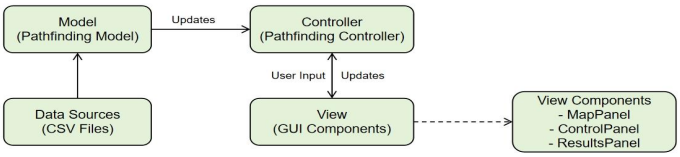


Figure.8 MVC Architecture of the interaction

This MVC design provides clear separation of concerns: the Model manages data and algorithms, the View handles presentation, and the Controller coordinates their interaction. This modular approach allows for independent development of components and facilitates future extensions.

2.1. Application of Object-Oriented Programming (OOP) Principles

2.1.1. Encapsulation

Encapsulation is the principle of hiding internal state and requiring all interaction to occur through well-defined interfaces. Here, encapsulation is evident in several ways:

```java
public class Graph {
    private Map<String, Map<String, Integer>>
adjacencyList;
    private Set<String> nodes;
    private Map<String, double[]> coordinates; // Store
coordinates for each city

    //Constructor initializes an empty graph
    public Graph() {
        adjacencyList = new HashMap<>();
        nodes = new HashSet<>();
        coordinates = new HashMap<>();
    }
. . .
}
```

(a) . Private fileds example in `Graph`

```java
public void buildGraph(List<RoadConnection> roads) {
    for (RoadConnection road : roads) {
        String cityA = road.getCityA();
        String cityB = road.getCityB();
        int distance = road.getDistance();
        // ...
        // Add edges in the undirected graph
        addEdge(cityA, cityB, distance);
        addEdge(cityB, cityA, distance);
    }
}

private void addEdge(String source, String destination, int
distance) {
    adjacencyList.computeIfAbsent(source, k -> new
HashMap<>()).put(destination, distance);
}
```

(b) . Information hiding example in `Graph`

```java
public class PathResult {
    private final List<String> path;
    private final int totalDistance;
    private final AlgorithmStats stats;

public PathResult(List<String> path, int totalDistance,
AlgorithmStats stats) {
    // Ensure path is unmodifiable and a defensive copy is
made
    this.path = (path != null) ?
Collections.unmodifiableList(new ArrayList<>(path)) :
Collections.emptyList();
    this.totalDistance = totalDistance;
    this.stats = stats;
  }
. . .
}
```

(c) . Immutable object example in `PathResult`

Figure 9 Code example of the application of Encapsulation

1. **Algorithm separation :** I have designed a total of four algorithms and implemented their own algorithm logics in the four classes of `DijkstraPathfinder`, `AStarPathfinder`, `NN2OptPathfinder` and `TspPathfinder` respectively. They can be independently modified, which is convenient for testing and debugging.
2. **Private Fields with Public Methods :** Classes like `Graph` and `AlgorithmStats` keep their data members private and expose functionality through public methods, preventing direct manipulate internal state(Fig 9.(a)).
3. **Information Hiding :** The internal implementation of the graph structure is hidden within the `Graph` class, The `addEdge` method is declared as private because it is an internal implementation detail. Externally, only the `buildGraph` method can construct a graph, and edges cannot be directly added(Fig 9.(b)) .
4. **Immutable object :** In the `PathResult` class, I designed the encapsulation of immutable objects: Statement for the final by all fields, and in the constructor to use defensive copying `(Collections. UnmodifiableList (new ArrayList < > (path)))` to create an immutable copy of the path, ensures that the internal state cannot be modified externally(Fig 9.(c)).

2.1.2. Abstraction

```java
public class TspPathfinder {
    @FunctionalInterface
    public interface PathSegmentProvider {
        PathResult getShortestPathSegment(String from, String
to);
    }

    @FunctionalInterface
    public interface AStarPathSegmentProvider {
        PathResult getShortestPathAStarSegment(String from,
String to, HeuristicStrategy strategy);
    }
    ...
}
```

(a) . Functional Interface example in `TspPathfinder`

```java
public class DijkstraPathfinder {
...
    public PathResult findShortestPath(String start,
String end) {
    return new PathResult(path, distance, stats);
    }
    ...
```

```java
public class AStarPathfinder {
...
    public PathResult findShortestPath(String start,
String end) {
    return new PathResult(path, distance, stats);
    }
    ...
```

```java
public PathResult findShortestPath(String start, String end) {
    return dijkstraPathfinder.findShortestPath(start, end);
}
public PathResult findShortestPathAStar(String start, String end, HeuristicStrategy heuristicStrategy) {
    return aStarPathfinder.findShortestPathAStar(start, end, heuristicStrategy);
}
```

(b) . Abstraction application in `PathResult`

Figure.10 Code example of the application of Abstraction

Abstraction involves simplifying complex reality by modeling classes that represent the essential aspects of a domain. The application demonstrates in several ways:

1.**Functional Interface :** The `PathSegmentProvider` and `AStarPathSegmentProvider` functional interfaces in the `TspPathfinder` class are excellent examples of abstraction, providing a way to decouple the TSP logic(Fig.10(a)) .

2.**PathResult Class :** This class abstracts the result of a pathfinding operation,providing a clean way to access the path, distance, and performance statistics without exposing implementation details(Fig.10(b)) .

2.1.3. Inheritance

Inheritance allows a class to inherit properties and behavior from another class. In the application, inheritance is used judiciously:

1. **Heuristic Strategies :** Both `ZeroHeuristic` and `HaversineHeuristic` implement the `HeuristicStrategy` interface, inheriting its contract and providing specific implementations(Fig.11).

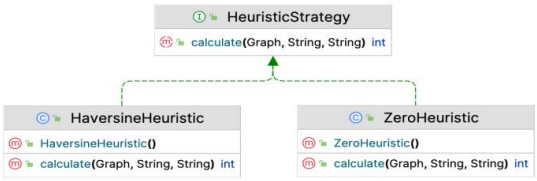


Figure 11 Inheritance hierarchy diagram for Heuristic Strategies

2. **UI Components Classes :** like `MapPanel` and `ResultsPanel` extend Swing components like `JPanel`, inheriting rendering capabilities and extending them with application-specific functionality

2.1.4. Polymorphism

Polymorphism allows objects of different types to be treated uniformly through a common interface.

1. **HeuristicStrategy:** Different implementations (Haversine, Zero) can be used interchangeably in A*.

2. `PathSegmentProvider` and `AStarPathSegmentProvider`: PathPlanner implements both interfaces, allowing it to be used in different contexts.

3. **Optimal strategy types**: Different strategies to find the best sequence can be selected at runtime.

2.1.5. Importance of OOP Principles

The application of OOP principles in this project offers several significant benefits:

1. **Modularity :** The clear separation of responsibilities into distinct classes makes the codebase easier to understand and maintain.
2. **Reusability :** Components like the `Graph` representation and `pathfinding` algorithms are designed to be reusable in different contexts.
3. **Extensibility :** Adding new features can be done with minimal changes to existing code thanks to the use of interfaces and *polymorphism*.
4. **Maintainability :** Encapsulation ensures that changes to internal implementations don't cascade throughout the system.
5. **Testability :** The separation of concerns and well-defined interfaces make unit testing more straightforward, as components can be tested in isolation.

These benefits collectively contribute to a robust, scalable application architecture that can evolve to meet changing requirements while maintaining code quality and reliability.

2. Chapter 2: Task B - Algorithm Evaluation: Graph

2.1. Algorithm Selection and Implementation

The path planning problem with points of interest can be broken down into two main subproblems:

1. Finding the **optimal order** to visit multiple points of interest (POIs)
2. Finding **the shortest path** between each consecutive pair of pointsfind

2.1.1. Finding the Optimal Visiting Order

Algorithm 1 Optimal Path by Permutation

```
function FINDOPTIMALPATHBYPERMUTATION(start, end, POIs, useAStar, heuristic)
    bestPath ← null
    shortestDistance ← ∞
    permutations ← GENERATEPERMUTATIONS(POIs)
    for each permutation in permutations do
        currentDistance ← 0
        valid ← true
        currentLocation ← start
        routeOrder ← [permutation_1, permutation_2, ..., permutation_n, end]
        for each nextDestination in routeOrder do
            segment ← GETPATHSEGMENT(currentLocation, nextDestination, useAStar, heuristic)
            if segment is not valid then
                valid ← false
                break
            end if
            currentDistance ← currentDistance + segment.distance
            currentLocation ← nextDestination
        end for
    end for
    if bestPath ≠ null then
        return reconstructFullPath(start, end, bestPath, useAStar, heuristic)
    else
        return pathNotFoundResult()
    end if
end function
```

Algorithm 2 Nearest Neighbor with 2-opt Optimization

```
function FINDOPTIMALPATHBYNN2OPT(start, end, POIs, useAStar, heuristic)
    // Step 1: Construct initial tour using Nearest Neighbor
    tour ← [start]
    remainingPOIs ← copy of POIs
    while remainingPOIs is not empty do
        current ← last city in tour
        nearest ← find nearest POI to current in remainingPOIs
        add nearest to tour
        remove nearest from remainingPOIs
    end while
    add end to tour
    // Step 2: Apply 2-opt improvement
    improved ← true
    while improved do
        improved ← false
        for i ← 1 to tour.length − 3 do
            for j ← i + 1 to tour.length − 2 do
                if swapping edges (tour[i],tour[i+1]) and (tour[j],tour[j+1]) reduces distance then
                    reverse the portion of tour between i + 1 and j
                    improved ← true
                end if
            end for
        end for
    end while
    return constructPathFromTour(tour, useAStar, heuristic)
end function
```

Figure 12 Pseudo-code for TSP and NN+2-opt

(A) .Permutation-based Approach (TSP Implementation):

For finding the optimal order to visit all points of interest, we implement a permutation-based approach as a Traveling Salesperson Problem(TSP) which guarantees finding the optimal solution by evaluating all possible orderings of POIs. However, its time complexity is O(n!) where n is the number of POIs, making it suitable only for a small number of POIs.

(B) K-Nearest Neighbors + 2-optimization

For scenarios with a larger number of POIs, we implement a heuristic approach that combines the Nearest Neighbor algorithm with 2-opt optimization which has a time complexity of $O(n^2)$ for the Nearest Neighbor phase and $O(n^2)$ for the 2-opt improvement phase, making it much more efficient for larger sets of POIs. While it doesn't guarantee the optimal solution, it typically produces good approximations.However, experiments have proved that with the data volume of fifteen cities like ours, this algorithm can almost achieve the optimal solution

2.1.2. The shortest path between two points

Algorithm 3 Dijkstra's Algorithm

```
function FINDSHORTESTPATH(start, end)
    Initialize distance[v] ← ∞ for all vertices v
    distance[start] ← 0
    Initialize priority queue Q with (start, 0)
    Initialize predecessor[v] ← null for all vertices v
    while Q is not empty do
        (u, dist) ← extract minimum from Q
        if u = end then
            return reconstruct path using predecessor
        end if
        for each neighbor v of u do
            alt ← distance[u] + weight(u, v)
            if alt < distance[v] then
                distance[v] ← alt
                predecessor[v] ← u
                add or update (v, alt) in Q
            end if
        end for
    end while
    return "No path found"
end function
```

Algorithm 4 A* Algorithm

```
function FINDSHORTESTPATHASTAR(start, end, heuristic)
    Initialize gScore[v] ← ∞ for all vertices v
    gScore[start] ← 0
    Initialize priority queue openSet with (start, 0) ordered by fScore
    Initialize cameFrom[v] ← null for all vertices v
    while openSet is not empty do
        current ← extract node with minimum fScore from openSet
        if current = end then
            return reconstruct path using cameFrom
        end if
        for each neighbor neighbor of current do
            tentativeGScore ← gScore[current] + distance(current, neighbor)
            if tentativeGScore < gScore[neighbor] then
                cameFrom[neighbor] ← current
                gScore[neighbor] ← tentativeGScore
                fScore[neighbor] ← gScore[neighbor] + heuristic(neighbor, end)
                add or update neighbor in openSet with priority fScore[neighbor]
            end if
        end for
    end while
    return "No path found"
end function
```

Figure 12 Pseudo-code for Dijkstra's Algorithm and A* Algorithm

(A) .Dijkstra's Algorithm Implementation:

Dijkstras'algorithm is used to find the shortest path between two cities in the road network which guarantees finding the shortest path in a graph with non-negative edge weights. Its time complexity is O((V + E) log V) using a binary heap priority queue, where V is the number of vertices cities and E is the number of (roads.

(B) A* Algorithm Implementation:

A* extends Dijkstra's algorithm by using a *heuristic function* to guide the search toward the goal.The A* algorithm uses a *heuristic function* to estimate the distance from the current node to the goal. In our implementation, we use the Haversine

distance (great-circle distance) as the heuristic for geographic coordinates.

The time complexity of *A** is the same as *Dijkstra's* in the worst case, O((V + E) log V), but it can be significantly faster in practice due to the heuristic guiding the search more directly toward the goal.

2.2. Demonstration of Results (Test Cases)

Case 1 : Houston TX to Philadelphia PA (No attractions)

For this direct route, the application used Dijkstra's algorithm and A* algorithm to find the shortest path:

Houston TX -> Columbus OH -> Philadelphia PA

Component	Total Distance (miles)	Execute Time (ms)	Nodes Expanded	Max Queue Size	Nodes Visited
Sequential + Dijkstra	1350	2	20	15	12
Sequential + A*	1350	0	17	14	5

Case 2 : Philadelphia PA to San Antonio TX via Hollywood Sign

This case required finding the city where the Hollywood Sign is located Los Angeles CA and calculating the optimal route through this intermediate point:

Philadelphia PA -> Columbus OH -> [Los Angeles CA] -> Phoenix AZ -> San Antonio TX

Component	Total Distance (miles)	Execute Time (ms)	Nodes Expanded	Max Queue Size	Nodes Visited
Sequential + Dijkstra	3700	0	52	23	28
Sequential + A*	3700	0	**48**	**14**	**7**
TSP + Dijkstra	3700	0	105	21	56
TSP + A*	3700	1	97	14	14
NN + Dijkstra	3700	1	79	21	50
NN + A*	3700	0	72	14	10

Case 3 : San Jose CA to Phoenix AZ via Liberty Bell and Millennium Park

This complex case required determining the optimal order to visit two attractions(Liberty Bell in Philadelphia PA and Millennium Park in Chicago IL):

San Jose CA -> Columbus OH -> [Philadelphia PA] -> Columbus OH -> [Chicago IL] -> Phoenix AZ

Component	Total Distance (miles)	Execute Time (ms)	Nodes Expanded	Max Queue Size	Nodes Visited
Sequential + Dijkstra	4620	1	72	21	41
Sequential + A*	4620	0	63	14	13
TSP + Dijkstra	4620	1	218	21	119
TSP + A*	4620	1	190	14	39
NN + Dijkstra	4620	2	416	23	251
NN + A*	4620	1	346	14	75

Case 4: Custom Test with Multiple Attractions with 6 attractions

To demonstrate the scalability of our algorithms, we tested a scenario with 6

attractions between New York and Los Angeles:

San Diego CA -> [San Jose CA] -> [Houston TX] -> [Dallas TX] -> [Chicago IL] -> [Columbus OH] -> [Charlotte NC] -> Philadelphia PA -> New York NY

Component	Total Distance (miles)	Execute Time (ms)	Nodes Expanded	Max Queue Size	Nodes Visited
Sequential + Dijkstra	7865	0	142	21	74
Sequential + A*	7865	0	123	14	22
TSP + Dijkstra	**4215**	1	105491	21	52117
TSP + A*	4215	2	90232	14	16579
NN + Dijkstra	**4255**	0	3262	21	1210
NN + A*	4255	1	**2778**	**14**	**507**

San Diego CA -> [San Jose CA] -> [Dallas TX] -> [Houston TX] -> [Charlotte NC] -> [Columbus OH] -> [Chicago IL] -> [Columbus OH] -> Philadelphia PA -> New York NY

Case 5: Custom Test with Multiple Attractions with 10 attractions

To compare the different algorithm we also tested a scenario with 10 attractions between New York and Los Angeles

New York NY -> [Philadelphia PA] -> [Columbus OH] -> [Chicago IL] -> [Columbus OH] -> [Charlotte NC] -> [Jacksonville FL] -> [Houston TX] -> [Fort Worth TX] -> [Phoenix AZ] -> [San Diego CA] -> [Los Angeles CA] -> San Jose CA

Component	Total Distance (miles)	Execute Time (ms)	Nodes Expanded	Max Queue Size	Nodes Visited
Sequential + Dijkstra	13305	1	253	31	126
Sequential + A*	13415	0	226	14	47
TSP + Dijkstra	4620	1327	912280512	31	463034924
TSP + A*	4620	3058	804505140	14	162207388
NN + Dijkstra	4620	1	5194	31	2387
NN + A*	**4620**	2	**4649**	**14**	**947**

2.3. Algorithm Complexity Analysis

2.3.1. Time Complexity

The time complexity of our path planning system depends on both the shortest path algorithm used and the strategy for finding the optimal order of multiple attractions.

1.Shortest Path Algorithms

- *Dijkstra' s Algorithm*: O((V + E) log V), where V is the number of cities and E is the number of roads. With 15 cities and 105 road connections, each search operation typically explores a significant portion of the graph.The logarithmic factor comes from the priority queue operations.
- *A* Algorithm*: O((V + E) log V) in the worst case, but with a good heuristic, the practical performance is often better. Our implementation uses the Haversine distance as a heuristic, which is admissible (never overestimates) and consistent (satisfies the triangle inequality).As shown in our test cases, ***A* typically visits fewer nodes than Dijkstra's algorithm for the same problem***.

2.Ordering Strategies

- *TSP Approach*: O(k! • (V + E) log V),where k is thenumber of attractions. This

approach generates all k! permutations of attractions and for each permutation, computes k + 1 shortest paths using either Dijkstra or A. The factorial growth makes this approach impractical for more than 5-6 attractions.

• Nearest Neighbor with 2-opt: O(k2 • (V + E) log V) The Nearest Neighbor phase is $O(k^2)$ as it involves finding the nearest unvisited point k times, each requiring k distance computations. The 2-opt improvement phase is also $O(k^2)$ as it considers all pairs of edges for potential swaps. ***Each distance computation requires a shortest path calculation using either Dijkstra or A*.***

Number of Attractions(k)	Permutation	**NN+2-opt**
3	6	9
5	120	25
7	5040	49
10	3628800	100

2.3.2. Space Complexity

1.Graph Representation

The space complexity of our graph representation is O(V +E), where V is city number and E is road number.Here:The *adjacency list* requires O(V + E) space;The *set of nodes* requires O(V) space;The *coordinates map* requires O(V) space.

2.Shortest Path Algorithms

• *Dijkstras Algorithm*: O(V) The algorithm maintains distance values, predecessor pointers, and a priority queue, all of which are O(V) in size.

• *A* Algorithm*: O(V) Similar to Dijkstra, A* requires O(V) space for the open set, g-scores, and predecessor pointers.

3.Ordering Strategies

• *Permutation-based Approach*: O(k! + V) Generating all permutations requires O(k!) space to store them. For large k, this becomes the dominant factor.

• *Nearest Neighbor with 2-opt*: O(k + V) The algorithm only needs to store the current tour and some additional state variables, requiring O(k) space beyond the graph representation.

Component	Time Complexity	Space Complexity
Graph Representation	-	O(V + E)
Dijkstra's Algorithm	O((V + E) log V)	O(V)
A* Algorithm	O((V + E) log V)	O(V)
TSP Strategy	O(k! · (V + E) log V)	O(k! + V)
NN+2-opt Strategy	$O(k^2 \cdot (V + E) \log V)$	O(k! + V)

3. Chapter 3: Sorting Algorithm (Task C)

3.1. Performance Testing and Visualization Overview

To evaluate the performance of different sorting algorithms, I conducted tests using four datasets: 1000places_sorted, 1000places_random, 10000places_sorted, and 10000places_random. Each dataset was processed using three classical sorting algorithms—Insertion Sort, Quick Sort, and Merge Sort.

For accurate and consistent results, the initial run was disregarded to eliminate overhead from Just-In-Time (JIT) compilation, memory allocation, and potential JVM class loading. Each sorting algorithm was then executed 10 times, and the average execution time was computed to minimize random variations and enhance measurement stability.

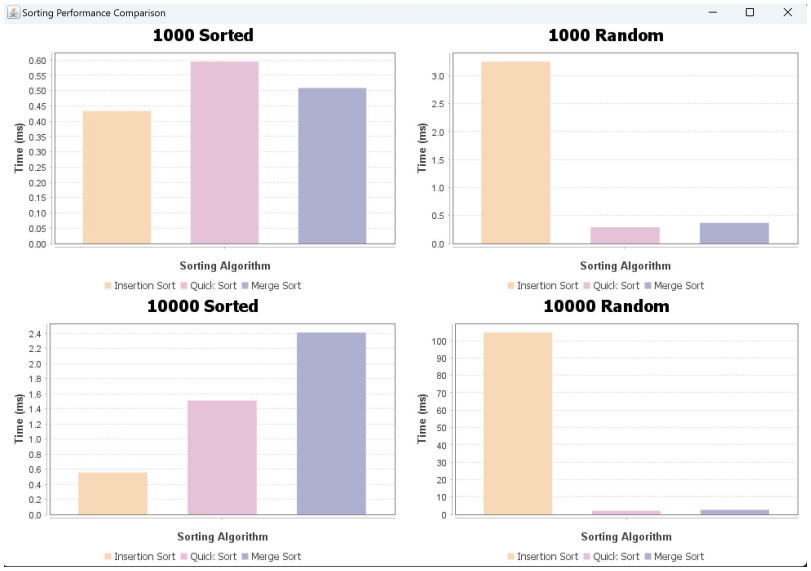


Figure.13 Execution time comparison of three sorting algorithms

Beyond numerical analysis, a GUI with color-coded bar charts offered intuitive visual comparison of sorting methods and datasets, making performance trends easily interpretable.

Unexpectedly, Quick Sort performed worse on a sorted 10,000-element dataset than on a random one, degrading to $O(n^2)$ complexity. This occurred because using the last element as the pivot on sorted data created highly unbalanced partitions.

Implementing a median-of-three pivot strategy resolved this by ensuring more

balanced partitions, restoring Quick Sort's expected O(n log n) performance and highlighting the critical impact of pivot selection on efficiency, particularly with ordered data.

Table 1. *Original execution time (ms) before pivot optimization*

Dataset	Insertion Sort (ms)	Quick Sort (ms)	Merge Sort (ms)
1000places_sorted	0.0834	2.9082	0.3293
1000places_random	1.9874	0.2885	0.3610
10000places_sorted	0.1591	191.1221	1.6657
10000places_random	77.1717	1.9228	2.7346

Table 2. *Average execution time (ms) of three sorting algorithms across different datasets (after optimizations)*

Datasets	Insertion (ms)	Quick (ms)	Merge (ms)
1000places_sorted	0.1061	0.2180	0.3005
1000places_random.csv	1.7002	0.2497	0.3860
10000places_sorted	0.2039	1.5032	1.6807
10000places_random.csv	82.0535	1.9301	2.7548

3.2. Impact of Input Order on Sorting Performance

Test results clearly show that input order (sorted vs. random) significantly affects the performance of certain sorting algorithms, notably Insertion Sort. For instance, Insertion Sort was considerably faster on pre-sorted datasets (like 1000places_sorted.csv) due to its $O(n)$ best-case complexity, where minimal element shifting occurs. Conversely, its $O(n^2)$ worst-case performance arises with disordered or reverse-sorted inputs. This sensitivity makes Insertion Sort particularly efficient for real-world, partially sorted data, such as logs or user activity sequences.

Quick Sort, on the other hand, showed relatively stable performance across both sorted and random inputs in our implementation, although the classic version is known to degrade to $O(n^2)$ on sorted data if the pivot selection is poor (e.g., always picking the last element). However, modern variations often include randomized or median-of-three pivot strategies to mitigate this.

Merge Sort appeared largely unaffected by input order. Its divide-and-conquer approach always splits the data and merges results in O(n log n) time, regardless of initial element order. This makes it a reliable choice for predictable performance,

though it comes with higher memory overhead.

3.3. Impact of Input Size on Sorting Algorithm Performance

The performance of sorting algorithms is notably affected by the size of the input, especially for those with higher time complexity. Our experiments with datasets of 1000 and 10000 elements highlight these differences clearly.

Insertion Sort, with its $O(n^2)$ worst-case complexity, showed the steepest increase in runtime. On random input, it rose from **1.70** ms (1000 items) to **82.05** ms (10000 items), a substantial escalation. This aligns with its quadratic growth. However, on sorted input, its runtime remained minimal—**0.11** ms to **0.20** ms—demonstrating its optimal behavior when the input is already ordered.

Merge Sort, which consistently runs in O(n log n), scaled predictably. Runtime increased from **0.39** ms to **2.75** ms on random input and from **0.30** ms to **1.68** ms on sorted data, reflecting stable performance regardless of input order.

Quick Sort displayed relatively balanced scaling on random input—from **0.25** ms to **1.93** ms—but initially suffered on sorted input. The unoptimized version jumped from **0.22** ms to **1.50** ms, which suggests improved handling of sorted data likely due to our use of the median-of-three pivot strategy.

3.4. Sorting Duplicate Values

Sorting datasets with a high number of duplicate values presents a unique challenge for certain algorithms. Since the original datasets provided did not include duplicates, we implemented a custom test class to generate synthetic datasets containing repeated values—cycling through only three place names across arrays of 1000 and 10000 elements.

Table 3. *Sorting Performance on Datasets with High Duplicate Values*

Dataset Size	Insertion Sort	Quick Sort	Merge Sort
1000 items	5.54 ms	4.85 ms	0.82 ms
10000 items	49.54 ms	82.67 ms	2.96 ms

Insertion Sort performs element-wise comparisons and shifts to place each item in its correct position. While it is efficient for sorted or nearly sorted data, its

performance deteriorates when handling duplicate values. This is because although the comparisons might be fast due to equality, the algorithm still needs to perform unnecessary shifts, especially in larger arrays. In our duplicate dataset tests, Insertion Sort required **5.54** ms for 1000 items and **49.54** ms for 10000 items, showing noticeable scaling issues. The presence of many equal elements does not mitigate the cost of shifting, leading to increased runtime with input size.

Quick Sort demonstrated the most significant performance degradation in the presence of duplicate values. When a poor pivot strategy is used—such as picking the last element—repeated values result in unbalanced partitions and deep recursive calls. In our tests, Quick Sort took **4.85** ms on the 1000-item duplicate dataset and a much higher **82.67** ms on 10000 items, making it the least efficient among the three. This reflects Quick Sort's sensitivity to input distribution and its tendency to recurse excessively when equal elements are not handled properly during partitioning.

Merge Sort, by contrast, maintained consistent and efficient performance across both input sizes. It divides the data evenly and merges in a way that is unaffected by the number of duplicate elements. Since it doesn't rely on comparisons that are skewed by value repetition, its runtime remains close to its theoretical O(n log n) complexity. For 1000 and 10000 duplicate values, Merge Sort completed in **0.82** ms and **2.96** ms respectively—the best among the tested algorithms. This consistency and robustness make Merge Sort the most suitable choice for sorting large datasets with many repeated entries.

3.5. Memory Usage Analysis Based on Algorithm Principles

To evaluate the memory efficiency of sorting algorithms in constrained environments, we initially considered measuring peak memory usage. However, we found this approach unreliable, as memory consumption varied significantly with input order—particularly for Quick Sort. We therefore refined our method by tracking total auxiliary memory allocations for Merge Sort and estimating stack usage through recursion depth for Quick Sort. This

provided a more consistent and meaningful basis for comparison across different input scenarios.

Table 4. *Memory Usage Comparison Across Sorting Algorithms*

Dataset	Insertion Sort (Extra Memory)	Quick Sort (Stack Usage)	Recursion Depth	Merge Sort (Extra Memory)
1000 Sorted	0 KB	0 KB	9	77 KB
1000 Random	0 KB	1 KB	17	77 KB
10000 Sorted	0 KB	0 KB	13	1043 KB
10000 Random	0 KB	1 KB	22	1043 KB

Insertion Sort

Insertion Sort uses no additional memory during the sorting process. It sorts the input array in place by repeatedly shifting elements and inserting the current value into its correct position. In our tests, regardless of input size or whether the data is sorted or random, Insertion Sort consistently used **0 KB** of extra memory. This aligns with its space complexity of **O(1)**. The fact that no recursive calls or helper arrays are needed makes it especially suitable for memory-constrained systems like embedded devices. However, this memory efficiency comes at the cost of slower performance on large or unsorted datasets.

Quick Sort is also an in-place algorithm, meaning it does not require extra memory for data storage. However, it uses stack memory for each recursive call. Our results show that Quick Sort's memory usage varies depending on input order. For example, sorting 10000 elements of random data used only **1 KB of stack memory** with a recursion depth of **22**. On sorted input, the recursion depth dropped to **13**, using **0 KB** of stack. These results reflect the effect of using the **median-of-three pivot strategy**, which helps maintain balanced partitions and avoids deep recursion.

Earlier in the project, when we used the default pivot strategy (selecting the last element), Quick Sort showed severe performance degradation on sorted input. In that case, the recursion depth reached 9999, and the estimated stack usage was

624 KB, highlighting the importance of proper pivot selection for memory safety and performance stability.

Merge Sort

Merge Sort always allocates additional space because it creates temporary arrays during the merging process. This makes its space complexity O(n). In our measurements, Merge Sort used **77** KB for 1000 elements and **1043** KB for 10000 elements, regardless of whether the data was sorted or random. This predictable growth in memory usage reflects the algorithm's structure, which always splits and merges arrays recursively. While Merge Sort is stable and avoids deep recursion problems, its memory footprint is too large for most embedded systems or constrained devices.

4. Chapter4: AI-Assisted Planning and Collaboration(Task D)

During this project, our team used Trello to help with planning and task assignment. We split the development work into weekly parts. We used Trello boards to give out tasks, follow progress, and manage code updates. Trello is not a strong AI tool, but its checklists, timelines, and reminders helped us stay organized and not miss important work.

The work went well because we gave each other feedback often. We looked at each other's tasks on Trello, wrote comments under each task, and gave suggestions. For example, when we tested the sorting algorithms, we saw that the first version of the chart was hard to understand. After some feedback on Trello, we changed the way it looked and made the chart easier to read. This way of working helped us work better and get better results.

4.1. Recognition of Equality, Diversity, and Inclusion (EDI)

I think equality in software means all users should get the same information and services. It should not matter where they are from or what culture they have. In our app, we made sure everyone saw the same interface. We did not change content based on location.

I see diversity as making the design work for many kinds of users. This includes travelers from different places, who speak different languages, and who are in different age groups. Later, I want to add features like support for many languages, color settings for people with color blindness, and display options for older users or people with low vision. We also want to let users plan from their home region and connect their trip to other places. They could also choose two cities they must visit, and the app would help find the best way to plan the rest of the trip around them.

I think inclusion means keeping things simple. The interface should be easy to use. People who are not good with technology, including older adults or people with disabilities, should still be able to use the main features without problems.

Some problems we may face include higher development costs and harder design work when we try to support many user types. It is also hard to keep the design simple when adding more options. We also need to test with real users who have special needs, which takes time and resources.

Conclusion: Skills and Knowledge Gained

Completing this project has significantly improved both my **technical skills** and my understanding of full-cycle software development. I deepened my knowledge of **algorithm implementation**, especially in analyzing sorting performance under different dataset conditions. I also gained hands-on experience with **Java GUI design**, performance measurement, and code modularization.

Furthermore, I learned how to break down complex problems independently and troubleshoot issues during development. Working with a team taught me how real-world collaborative workflows operate—especially how to coordinate tasks, manage shared responsibilities, and conduct incremental improvements based on team feedback. These skills are essential for any aspiring software developer and have prepared me to handle larger-scale, more inclusive, and user-centered development challenges in the future.

5. Chapter 5: Program Code

TaskA_PathPlanner

AStarPathfinder.java

```
package algorithm;

import path_planning.Graph;
import path_planning.PathResult;
import path_planning.AlgorithmStats;
import path_planning.NodeDistance;
import path_planning.heuristics.HeuristicStrategy;

import java.util.*;

public class AStarPathfinder {

    private Graph graph;

    public AStarPathfinder(Graph graph) {
        this.graph = graph;
    }

    public PathResult findShortestPathAStar(String start, String end, HeuristicStrategy heuristicStrategy) {
        long startTime = System.currentTimeMillis();
        int nodesVisited = 0;
        int nodesExpanded = 0;
        int maxQueueSize = 0;
        int totalNodes = graph.getNodes().size(); // Get total node count for complexity estimation

        if (!graph.containsNode(start) || !graph.containsNode(end)) {
            System.err.println("Error: Start or end node not found in graph (A*).");
            return new PathResult(Collections.emptyList(), -1, new AlgorithmStats(System.currentTimeMillis()
- startTime, 0, 0, 0, 0, totalNodes));
        }

        if (start.equals(end)) {
            return new PathResult(Collections.singletonList(start), 0, new
AlgorithmStats(System.currentTimeMillis() - startTime, 1, 0, 1, 1, totalNodes));
        }

        System.out.println("Calculating path using A* algorithm (AStarPathfinder) with heuristic: " +
heuristicStrategy.getClass().getSimpleName());

        PriorityQueue<NodeDistance> openSet = new PriorityQueue<>(Comparator.comparingInt(nd ->
nd.getFCost(heuristicStrategy, end, graph)));
        Map<String, String> cameFrom = new HashMap<>();
        Map<String, Integer> gScore = new HashMap<>();

        for (String node : graph.getNodes()) {
            gScore.put(node, Integer.MAX_VALUE);
        }

        gScore.put(start, 0);
        openSet.add(new NodeDistance(start, 0));
        maxQueueSize = 1;
        nodesExpanded = 1;

        while (!openSet.isEmpty()) {
            NodeDistance currentWrapper = openSet.poll();
            String current = currentWrapper.getNode();
            int currentGScore = currentWrapper.getGCost(); // Use getGCost for clarity
```

```
                nodesVisited++;

                if (current.equals(end)) {
                    List<String> path = reconstructPath(cameFrom, current);
                    int distance = calculatePathDistance(path);
                    long endTime = System.currentTimeMillis();
                    AlgorithmStats stats = new AlgorithmStats(endTime - startTime, nodesVisited,
nodesExpanded, maxQueueSize, path.size(), totalNodes);
                    return new PathResult(path, distance, stats);
                }

                if (currentGScore > gScore.getOrDefault(current, Integer.MAX_VALUE)) {
                    continue;
                }

                for (Map.Entry<String, Integer> neighborEntry : graph.getNeighbors(current).entrySet()) {
                    String neighbor = neighborEntry.getKey();
                    int edgeWeight = neighborEntry.getValue();
                    if (currentGScore == Integer.MAX_VALUE) continue; // Avoid overflow if currentGScore is
MAX_VALUE

                    int tentativeGScore = currentGScore + edgeWeight;

                    if (tentativeGScore < gScore.getOrDefault(neighbor, Integer.MAX_VALUE)) {
                        cameFrom.put(neighbor, current);
                        gScore.put(neighbor, tentativeGScore);
                        NodeDistance neighborNodeDistance = new NodeDistance(neighbor, tentativeGScore);

                        // Standard A* with re-adding to priority queue (Java's PQ doesn't support
decrease-key efficiently)
                        openSet.remove(neighborNodeDistance); // Remove if present (uses equals based on
node ID)
                        openSet.add(neighborNodeDistance);      // Add (or re-add with new GScore)
                        nodesExpanded++; // Count as expansion

                        if (openSet.size() > maxQueueSize) {
                            maxQueueSize = openSet.size();
                        }
                    }
                }
            }

            long endTime = System.currentTimeMillis();
            AlgorithmStats stats = new AlgorithmStats(endTime - startTime, nodesVisited, nodesExpanded,
maxQueueSize, 0, totalNodes);
            return new PathResult(Collections.emptyList(), -1, stats);
        }

        private List<String> reconstructPath(Map<String, String> cameFrom, String current) {
            List<String> totalPath = new ArrayList<>();
            totalPath.add(current);
            while (cameFrom.containsKey(current)) {
                current = cameFrom.get(current);
                totalPath.add(0, current);
            }
            return totalPath;
        }

        private int calculatePathDistance(List<String> path) {
            if (path == null || path.size() < 2) {
                return (path != null && path.size() == 1) ? 0 : -1;
            }
            int distance = 0;
            for (int i = 0; i < path.size() - 1; i++) {
```

```
                String current = path.get(i);
                String next = path.get(i + 1);
                int segmentDistance = graph.getDistance(current, next);
                if (segmentDistance < 0) {
                    System.err.println("Error calculating distance (A*): Segment " + current + " -> " + next + " not
found in graph.");
                    return -1;
                }
                distance += segmentDistance;
            }
            return distance;
        }
\
```

DijkstraPathfinder.java

```
package algorithm;

import path_planning.Graph;
import path_planning.PathResult;
import path_planning.AlgorithmStats;
import path_planning.NodeDistance; // Assuming NodeDistance is now a top-level class

import java.util.*;

public class DijkstraPathfinder {

    private Graph graph;

    public DijkstraPathfinder(Graph graph) {
        this.graph = graph;
    }

    public PathResult findShortestPath(String start, String end) {
        long startTime = System.currentTimeMillis();
        int nodesVisited = 0;
        int nodesExpanded = 0;
        int maxQueueSize = 0;
        int totalNodes = graph.getNodes().size(); // Get total node count for complexity estimation

        if (!graph.containsNode(start) || !graph.containsNode(end)) {
            System.err.println("Error: Start or end node not found in graph (Dijkstra).");
            return new PathResult(Collections.emptyList(), -1, new AlgorithmStats(System.currentTimeMillis()
- startTime, 0, 0, 0, 0, totalNodes));
        }

        if (start.equals(end)) {
            AlgorithmStats stats = new AlgorithmStats(System.currentTimeMillis() - startTime, 1, 0, 1, 1,
totalNodes);
            return new PathResult(Collections.singletonList(start), 0, stats);
        }

        System.out.println("Calculating path using Dijkstra algorithm (DijkstraPathfinder)...");

        PriorityQueue<NodeDistance> pq = new
PriorityQueue<>(Comparator.comparingInt(NodeDistance::getDistance));
        Map<String, Integer> distances = new HashMap<>();
        Map<String, String> previous = new HashMap<>();

        for (String node : graph.getNodes()) {
            distances.put(node, Integer.MAX_VALUE);
        }

        distances.put(start, 0);
```

```
        pq.add(new NodeDistance(start, 0));
        maxQueueSize = 1;
        nodesExpanded = 1;

        while (!pq.isEmpty()) {
            NodeDistance current = pq.poll();
            String currentNode = current.getNode();
            int currentDistance = current.getDistance();
            nodesVisited++;

            if (currentNode.equals(end)) {
                break;
            }

            if (currentDistance > distances.get(currentNode) && distances.get(currentNode) !=
Integer.MAX_VALUE) {
                continue;
            }

            for (Map.Entry<String, Integer> neighbor : graph.getNeighbors(currentNode).entrySet()) {
                String neighborNode = neighbor.getKey();
                int edgeWeight = neighbor.getValue();
                if (currentDistance == Integer.MAX_VALUE) continue;

                int newDistance = currentDistance + edgeWeight;
                if (newDistance < distances.getOrDefault(neighborNode, Integer.MAX_VALUE)) {
                    distances.put(neighborNode, newDistance);
                    previous.put(neighborNode, currentNode);
                    pq.add(new NodeDistance(neighborNode, newDistance));
                    nodesExpanded++;
                    if (pq.size() > maxQueueSize) {
                        maxQueueSize = pq.size();
                    }
                }
            }
        }

        List<String> path = buildPath(start, end, previous, distances);
        int distance = calculatePathDistance(path);
        long endTime = System.currentTimeMillis();

        if (path.isEmpty() || distance < 0) {
            AlgorithmStats stats = new AlgorithmStats(endTime - startTime, nodesVisited, nodesExpanded,
maxQueueSize, 0, totalNodes);
            return new PathResult(Collections.emptyList(), -1, stats);
        }

        AlgorithmStats stats = new AlgorithmStats(endTime - startTime, nodesVisited, nodesExpanded,
maxQueueSize, path.size(), totalNodes);
        return new PathResult(path, distance, stats);
    }

    private List<String> buildPath(String start, String end, Map<String, String> previous, Map<String, Integer>
distances) {
        LinkedList<String> path = new LinkedList<>();
        if (distances.getOrDefault(end, Integer.MAX_VALUE) == Integer.MAX_VALUE) {
            return Collections.emptyList();
        }
        String current = end;
        while (current != null) {
            path.addFirst(current);
            if (current.equals(start)) break;
            current = previous.get(current);
            if (current == null && !path.getFirst().equals(start)) {
```

```
                System.err.println("Path reconstruction error (Dijkstra)!");
                return Collections.emptyList();
            }
        }
        if (path.isEmpty() && start.equals(end)) {
            path.add(start);
        } else if (path.isEmpty() || !path.getFirst().equals(start)) {
            return Collections.emptyList();
        }
        return path;
    }

    private int calculatePathDistance(List<String> path) {
        if (path == null || path.size() < 2) {
            return (path != null && path.size() == 1) ? 0 : -1;
        }
        int distance = 0;
        for (int i = 0; i < path.size() - 1; i++) {
            String current = path.get(i);
            String next = path.get(i + 1);
            int segmentDistance = graph.getDistance(current, next);
            if (segmentDistance < 0) {
                System.err.println("Error calculating distance (Dijkstra): Segment " + current + " -> " + next +
" not found in graph.");
                return -1;
            }
            distance += segmentDistance;
        }
        return distance;
    }
}
```

NN2OptPathfinder.java

```
package algorithm;

import path_planning.Graph;
import path_planning.PathResult;
import path_planning.AlgorithmStats;
import path_planning.NodeDistance; // Assuming NodeDistance is now a top-level class

import java.util.*;

public class DijkstraPathfinder {

    private Graph graph;

    public DijkstraPathfinder(Graph graph) {
        this.graph = graph;
    }

    public PathResult findShortestPath(String start, String end) {
        long startTime = System.currentTimeMillis();
        int nodesVisited = 0;
        int nodesExpanded = 0;
        int maxQueueSize = 0;
        int totalNodes = graph.getNodes().size(); // Get total node count for complexity estimation

        if (!graph.containsNode(start) || !graph.containsNode(end)) {
            System.err.println("Error: Start or end node not found in graph (Dijkstra).");
            return new PathResult(Collections.emptyList(), -1, new AlgorithmStats(System.currentTimeMillis()
- startTime, 0, 0, 0, 0, totalNodes));
        }

        if (start.equals(end)) {
```

```
            AlgorithmStats stats = new AlgorithmStats(System.currentTimeMillis() - startTime, 1, 0, 1, 1,
totalNodes);
            return new PathResult(Collections.singletonList(start), 0, stats);
        }

        System.out.println("Calculating path using Dijkstra algorithm (DijkstraPathfinder)...");

        PriorityQueue<NodeDistance> pq = new
PriorityQueue<>(Comparator.comparingInt(NodeDistance::getDistance));
        Map<String, Integer> distances = new HashMap<>();
        Map<String, String> previous = new HashMap<>();

        for (String node : graph.getNodes()) {
            distances.put(node, Integer.MAX_VALUE);
        }

        distances.put(start, 0);
        pq.add(new NodeDistance(start, 0));
        maxQueueSize = 1;
        nodesExpanded = 1;

        while (!pq.isEmpty()) {
            NodeDistance current = pq.poll();
            String currentNode = current.getNode();
            int currentDistance = current.getDistance();
            nodesVisited++;

            if (currentNode.equals(end)) {
                break;
            }

            if (currentDistance > distances.get(currentNode) && distances.get(currentNode) !=
Integer.MAX_VALUE) {
                continue;
            }

            for (Map.Entry<String, Integer> neighbor : graph.getNeighbors(currentNode).entrySet()) {
                String neighborNode = neighbor.getKey();
                int edgeWeight = neighbor.getValue();
                if (currentDistance == Integer.MAX_VALUE) continue;

                int newDistance = currentDistance + edgeWeight;
                if (newDistance < distances.getOrDefault(neighborNode, Integer.MAX_VALUE)) {
                    distances.put(neighborNode, newDistance);
                    previous.put(neighborNode, currentNode);
                    pq.add(new NodeDistance(neighborNode, newDistance));
                    nodesExpanded++;
                    if (pq.size() > maxQueueSize) {
                        maxQueueSize = pq.size();
                    }
                }
            }
        }

        List<String> path = buildPath(start, end, previous, distances);
        int distance = calculatePathDistance(path);
        long endTime = System.currentTimeMillis();

        if (path.isEmpty() || distance < 0) {
            AlgorithmStats stats = new AlgorithmStats(endTime - startTime, nodesVisited, nodesExpanded,
maxQueueSize, 0, totalNodes);
            return new PathResult(Collections.emptyList(), -1, stats);
        }
```

```
        AlgorithmStats stats = new AlgorithmStats(endTime - startTime, nodesVisited, nodesExpanded,
maxQueueSize, path.size(), totalNodes);
        return new PathResult(path, distance, stats);
    }

    private List<String> buildPath(String start, String end, Map<String, String> previous, Map<String, Integer>
distances) {
        LinkedList<String> path = new LinkedList<>();
        if (distances.getOrDefault(end, Integer.MAX_VALUE) == Integer.MAX_VALUE) {
            return Collections.emptyList();
        }
        String current = end;
        while (current != null) {
            path.addFirst(current);
            if (current.equals(start)) break;
            current = previous.get(current);
            if (current == null && !path.getFirst().equals(start)) {
                System.err.println("Path reconstruction error (Dijkstra)!");
                return Collections.emptyList();
            }
        }
        if (path.isEmpty() && start.equals(end)) {
            path.add(start);
        } else if (path.isEmpty() || !path.getFirst().equals(start)) {
            return Collections.emptyList();
        }
        return path;
    }

    private int calculatePathDistance(List<String> path) {
        if (path == null || path.size() < 2) {
            return (path != null && path.size() == 1) ? 0 : -1;
        }
        int distance = 0;
        for (int i = 0; i < path.size() - 1; i++) {
            String current = path.get(i);
            String next = path.get(i + 1);
            int segmentDistance = graph.getDistance(current, next);
            if (segmentDistance < 0) {
                System.err.println("Error calculating distance (Dijkstra): Segment " + current + " -> " + next +
" not found in graph.");
                return -1;
            }
            distance += segmentDistance;
        }
        return distance;
    }
}
```

TspPathfinder.java

```
package algorithm;

import path_planning.Graph;
import path_planning.PathResult;
import path_planning.AlgorithmStats;
import path_planning.heuristics.HeuristicStrategy;
import path_planning.heuristics.ZeroHeuristic; // For fallback

import java.util.*;
import java.util.stream.Collectors;

public class TspPathfinder {

    // Functional interface to allow TspPathfinder to request path segments
    // from an external provider (e.g., DijkstraPathfinder or AStarPathfinder via PathPlanner).
```

```
    @FunctionalInterface
    public interface PathSegmentProvider {
        PathResult getShortestPathSegment(String from, String to);
    }

    // If A* segments are to be used, a similar provider for A* would be needed, or the existing one enhanced.
    @FunctionalInterface
    public interface AStarPathSegmentProvider {
        PathResult getShortestPathAStarSegment(String from, String to, HeuristicStrategy strategy);
    }

    private Graph graph; // Keep for graph.getNodes().size() for stats, or pass totalNodes count in
    private PathSegmentProvider dijkstraSegmentProvider;      // For Dijkstra-based segments
    private AStarPathSegmentProvider aStarSegmentProvider;        // For A*-based segments
    private HeuristicStrategy defaultAStarHeuristic;          // Default heuristic if A* is chosen and none is
provided

    // Constructor for when Dijkstra segments are primary
    public TspPathfinder(Graph graph, PathSegmentProvider dijkstraSegmentProvider) {
        this.graph = graph;
        this.dijkstraSegmentProvider = dijkstraSegmentProvider;
        this.defaultAStarHeuristic = new ZeroHeuristic(); // A sensible default, though not used if only dijkstra
provider is set
    }

    // Constructor for when A* segments might be used (can also do Dijkstra if aStarSegmentProvider also
implements PathSegmentProvider or by specific calls)
    public TspPathfinder(Graph graph, AStarPathSegmentProvider aStarSegmentProvider, HeuristicStrategy
defaultHeuristic) {
        this.graph = graph;
        this.aStarSegmentProvider = aStarSegmentProvider;
        this.defaultAStarHeuristic = (defaultHeuristic != null) ? defaultHeuristic : new ZeroHeuristic();
    }

    // Constructor that takes both, allowing flexibility or if PathPlanner is both
    public TspPathfinder(Graph graph, PathSegmentProvider dijkstraProvider, AStarPathSegmentProvider
aStarProvider, HeuristicStrategy defaultHeuristic) {
        this.graph = graph;
        this.dijkstraSegmentProvider = dijkstraProvider;
        this.aStarSegmentProvider = aStarProvider;
        this.defaultAStarHeuristic = (defaultHeuristic != null) ? defaultHeuristic : new ZeroHeuristic();
    }

    // Sequential path using Dijkstra segments
    public PathResult findPathWithInterestsSequential(String start, String end, List<String> placesOfInterest) {
        if (this.dijkstraSegmentProvider == null) {
            System.err.println("Error: TspPathfinder (Sequential Dijkstra) requires a Dijkstra
PathSegmentProvider.");
            return new PathResult(Collections.emptyList(), -1, new AlgorithmStats(0,0,0,0,0));
        }
        return calculatePathInOrder(start, end, placesOfInterest, false, null);
    }

    // Sequential path using A* segments
    public PathResult findPathWithInterestsSequentialAStar(String start, String end, List<String>
placesOfInterest, HeuristicStrategy strategy) {
        if (this.aStarSegmentProvider == null) {
            System.err.println("Error: TspPathfinder (Sequential A*) requires an AStarPathSegmentProvider.");
            return new PathResult(Collections.emptyList(), -1, new AlgorithmStats(0,0,0,0,0));
        }
        HeuristicStrategy heuristicToUse = (strategy != null) ? strategy : this.defaultAStarHeuristic;
        return calculatePathInOrder(start, end, placesOfInterest, true, heuristicToUse);
    }
```

```
    // Optimal TSP path using Dijkstra segments
    public PathResult findOptimalTspPath(String start, String end, List<String> placesOfInterest) {
        if (this.dijkstraSegmentProvider == null) {
            System.err.println("Error: TspPathfinder (Optimal Dijkstra) requires a Dijkstra
PathSegmentProvider.");
            return new PathResult(Collections.emptyList(), -1, new AlgorithmStats(0,0,0,0,0));
        }
        return findOptimalPathByPermutation(start, end, placesOfInterest, false, null);
    }

    // Optimal TSP path using A* segments
    public PathResult findOptimalTspPathAStar(String start, String end, List<String> placesOfInterest,
HeuristicStrategy strategy) {
        if (this.aStarSegmentProvider == null) {
            System.err.println("Error: TspPathfinder (Optimal A*) requires an AStarPathSegmentProvider.");
            return new PathResult(Collections.emptyList(), -1, new AlgorithmStats(0,0,0,0,0));
        }
        HeuristicStrategy heuristicToUse = (strategy != null) ? strategy : this.defaultAStarHeuristic;
        return findOptimalPathByPermutation(start, end, placesOfInterest, true, heuristicToUse);
    }

    // Common private method for sequential path calculation
    private PathResult calculatePathInOrder(String start, String end, List<String> placesOfInterest,
                                            boolean useAStar, HeuristicStrategy heuristic) {
        long startTime = System.currentTimeMillis();
        int totalNodesVisited = 0;
        int totalNodesExpanded = 0;
        int maxQueueSize = 0;
        int totalGraphNodes = (graph != null && graph.getNodes() != null) ? graph.getNodes().size() : 0;

        List<String> fullPath = new ArrayList<>();
        int totalDistance = 0;
        String currentLocation = start;

        List<String> pois = (placesOfInterest == null) ? new ArrayList<>() : new ArrayList<>(placesOfInterest);
        List<String> pointsToVisitInOrder = new ArrayList<>(pois);
        pointsToVisitInOrder.add(end);

        for (String nextDestination : pointsToVisitInOrder) {
            PathResult segmentResult;
            if (useAStar) {
                segmentResult = aStarSegmentProvider.getShortestPathAStarSegment(currentLocation,
nextDestination, heuristic);
            } else {
                segmentResult = dijkstraSegmentProvider.getShortestPathSegment(currentLocation,
nextDestination);
            }

            if (segmentResult.getStats() != null) {
                totalNodesVisited += segmentResult.getStats().getNodesVisited();
                totalNodesExpanded += segmentResult.getStats().getNodesExpanded();
                maxQueueSize = Math.max(maxQueueSize, segmentResult.getStats().getMaxQueueSize());
            }
            if (!segmentResult.isFound() || segmentResult.getTotalDistance() == Integer.MAX_VALUE) {
                System.err.println("Sequential Path: Cannot find path segment from " + currentLocation + " to
" + nextDestination);
                return new PathResult(Collections.emptyList(), -1, new
AlgorithmStats(System.currentTimeMillis() - startTime, totalNodesVisited, totalNodesExpanded,
maxQueueSize, 0, totalGraphNodes));
            }
            if (fullPath.isEmpty()) {
                fullPath.addAll(segmentResult.getPath());
            } else if (!segmentResult.getPath().isEmpty()){
                fullPath.addAll(segmentResult.getPath().subList(1, segmentResult.getPath().size()));
```

```
                }
                totalDistance += segmentResult.getTotalDistance();
                currentLocation = nextDestination;
            }
            AlgorithmStats stats = new AlgorithmStats(System.currentTimeMillis() – startTime, totalNodesVisited,
totalNodesExpanded, maxQueueSize, fullPath.size(), totalGraphNodes);
            return new PathResult(fullPath, totalDistance, stats);
        }

        // Common private method for optimal path calculation using permutations
        private PathResult findOptimalPathByPermutation(String start, String end, List<String> placesOfInterest,
                                                        boolean useAStar, HeuristicStrategy heuristic) {
            long startTime = System.currentTimeMillis();
            int permsNodesVisited = 0; // Stats specifically for the permutation evaluation phase
            int permsNodesExpanded = 0;
            int permsMaxQueueSize = 0;
            int totalGraphNodes = (graph != null && graph.getNodes() != null) ? graph.getNodes().size() : 0;

            List<String> bestPathConstruction = null;
            int shortestDistanceSoFar = Integer.MAX_VALUE;
            List<String> pois = (placesOfInterest == null) ? new ArrayList<>() : new ArrayList<>(placesOfInterest);

            if (pois.isEmpty()) { // Direct path if no POIs
                return calculatePathInOrder(start, end, Collections.emptyList(), useAStar, heuristic);
            }

            List<List<String>> permutations = generatePermutations(pois);
            System.out.println("TSP Optimal: Generated " + permutations.size() + " permutations for " + pois.size()
+ " POIs.");
            permsNodesExpanded += permutations.size(); // Count permutations as expanded states

            for (List<String> permutation : permutations) {
                int currentPermutationDistance = 0;
                boolean currentPermutationValid = true;
                String currentLocation = start;

                // Temporarily gather segment stats for this permutation
                int tempVisited = 0;
                int tempExpanded = 0;
                int tempQueue = 0;

                // Path for this permutation: start -> perm_poi_1 -> ... -> perm_poi_n -> end
                List<String> currentRouteOrder = new ArrayList<>(permutation);
                currentRouteOrder.add(end); // Add the final destination to the list of points to visit

                for (String nextDestination : currentRouteOrder) {
                    PathResult segmentResult;
                    if (useAStar) {
                        segmentResult = aStarSegmentProvider.getShortestPathAStarSegment(currentLocation,
nextDestination, heuristic);
                    } else {
                        segmentResult = dijkstraSegmentProvider.getShortestPathSegment(currentLocation,
nextDestination);
                    }

                    if (segmentResult.getStats() != null) {
                        tempVisited += segmentResult.getStats().getNodesVisited();
                        tempExpanded += segmentResult.getStats().getNodesExpanded();
                        tempQueue = Math.max(tempQueue, segmentResult.getStats().getMaxQueueSize());
                    }

                    if (!segmentResult.isFound() || segmentResult.getTotalDistance() == Integer.MAX_VALUE) {
                        currentPermutationValid = false;
                        System.err.println("TSP Optimal: Cannot find path segment from " + currentLocation + "
```

```
to " + nextDestination);
                        break; // Skip this permutation entirely, it has an invalid segment
                    }

                    currentPermutationDistance += segmentResult.getTotalDistance();
                    currentLocation = nextDestination;

                    if (currentPermutationDistance >= shortestDistanceSoFar) {
                        // Early termination for this permutation - already worse than our best so far
                        currentPermutationValid = false;
                        break;
                    }
                }

                if (currentPermutationValid && currentPermutationDistance < shortestDistanceSoFar) {
                    shortestDistanceSoFar = currentPermutationDistance;
                    // Construct the best path later using the best permutation to save memory if paths are long
                    // For now, we'll just store the order and reconstruct once at the end.
                    // To properly sum AlgorithmStats, we would need to sum them from the *actual* segments
of the chosen path.
                    // So, we find the best order first, then reconstruct and sum stats.
                    bestPathConstruction = new ArrayList<>(permutation); // Store the order of POIs
                }
                // Aggregate stats from exploring this permutation (rough estimate)
                permsNodesVisited += tempVisited; // Or simply count permutations * POIs as visited states
                permsNodesExpanded += tempExpanded;
                permsMaxQueueSize = Math.max(permsMaxQueueSize, tempQueue);
            }

            if (bestPathConstruction == null) {
                System.err.println("TSP Optimal: No valid path found that visits all POIs.");
                return new PathResult(Collections.emptyList(), -1, new AlgorithmStats(System.currentTimeMillis()
- startTime, permsNodesVisited, permsNodesExpanded, permsMaxQueueSize, 0, totalGraphNodes));
            }

            // Reconstruct the best path and sum its accurate AlgorithmStats
            List<String> finalFullPath = new ArrayList<>();
            int finalTotalDistance = 0;
            String currentFinalLocation = start;
            List<String> finalRouteOrder = new ArrayList<>(bestPathConstruction);
            finalRouteOrder.add(end);

            int finalNodesVisited = 0;
            int finalNodesExpanded = 0;
            int finalMaxQueueSize = 0;
            int finalPathSize = 0;

            for (String nextDestination : finalRouteOrder) {
                PathResult segment;
                if (useAStar) {
                    segment = aStarSegmentProvider.getShortestPathAStarSegment(currentFinalLocation,
nextDestination, heuristic);
                } else {
                    segment = dijkstraSegmentProvider.getShortestPathSegment(currentFinalLocation,
nextDestination);
                }

                if (segment.getStats() != null) {
                    finalNodesVisited += segment.getStats().getNodesVisited();
                    finalNodesExpanded += segment.getStats().getNodesExpanded();
                    finalMaxQueueSize = Math.max(finalMaxQueueSize, segment.getStats().getMaxQueueSize());
                }

                if (finalFullPath.isEmpty()) {
```

```
                finalFullPath.addAll(segment.getPath());
            } else {
                finalFullPath.addAll(segment.getPath().subList(1, segment.getPath().size()));
            }
            finalTotalDistance += segment.getTotalDistance();
            finalPathSize = finalFullPath.size(); // Path size for stats
            currentFinalLocation = nextDestination;
        }

        // Total stats = permutation exploration stats + final path reconstruction stats
        AlgorithmStats combinedStats = new AlgorithmStats(
            System.currentTimeMillis() - startTime,
            permsNodesVisited + finalNodesVisited,
            permsNodesExpanded + finalNodesExpanded,
            Math.max(permsMaxQueueSize, finalMaxQueueSize),
            finalPathSize,
            totalGraphNodes
        );

        return new PathResult(finalFullPath, finalTotalDistance, combinedStats);
    }

    private <T> List<List<T>> generatePermutations(List<T> original) {
        if (original.isEmpty()) {
            List<List<T>> result = new ArrayList<>();
            result.add(new ArrayList<>());
            return result;
        }
        T firstElement = original.remove(0);
        List<List<T>> returnValue = new ArrayList<>();
        List<List<T>> permutations = generatePermutations(original);
        for (List<T> smallerPermutated : permutations) {
            for (int index = 0; index <= smallerPermutated.size(); index++) {
                List<T> temp = new ArrayList<>(smallerPermutated);
                temp.add(index, firstElement);
                returnValue.add(temp);
            }
        }
        return returnValue;
    }

    private PathResult getSegment(String from, String to, boolean useAStar, HeuristicStrategy heuristic) {
        if (useAStar) {
            if (aStarSegmentProvider == null) throw new IllegalStateException("AStarSegmentProvider not set
for A* segment.");
            return aStarSegmentProvider.getShortestPathAStarSegment(from, to, heuristic);
        } else {
            if (dijkstraSegmentProvider == null) throw new IllegalStateException("DijkstraSegmentProvider not
set for Dijkstra segment.");
            return dijkstraSegmentProvider.getShortestPathSegment(from, to);
        }
    }

    private PathResult reconstructAndStatPath(String start, String end, List<String> poiOrder,
                                              boolean useAStar, HeuristicStrategy heuristic,
                                              long overallStartTime, AlgorithmStats initialStats) {
        List<String> fullPath = new ArrayList<>();
        int totalDistance = 0;
        String currentLocation = start;
        AlgorithmStats pathConstructionStats = new AlgorithmStats(0,0,0,0,0, initialStats.getTotalNodes());

        List<String> pointsToVisit = new ArrayList<>(poiOrder);
        pointsToVisit.add(end);
```

```
        for (String nextDestination : pointsToVisit) {
            if (currentLocation.equals(nextDestination)) continue;
            PathResult segmentResult = getSegment(currentLocation, nextDestination, useAStar, heuristic);
            if (segmentResult.getStats() != null) pathConstructionStats.accumulate(segmentResult.getStats());

            if (!segmentResult.isFound()) {
                System.err.println("Reconstruction Error: Cannot find path segment from " + currentLocation
+ " to " + nextDestination);
                AlgorithmStats errorStats = new AlgorithmStats(System.currentTimeMillis() -
overallStartTime,
                                                               initialStats.getNodesVisited(),
                                                               initialStats.getNodesExpanded(),
                                                               initialStats.getMaxQueueSize(),
                                                               0,
                                                               initialStats.getTotalNodes());
                return new PathResult(Collections.emptyList(), -1, errorStats);
            }

            if (fullPath.isEmpty()) {
                fullPath.addAll(segmentResult.getPath());
            } else {
                fullPath.addAll(segmentResult.getPath().subList(1, segmentResult.getPath().size()));
            }
            totalDistance += segmentResult.getTotalDistance();
            currentLocation = nextDestination;
        }

        // Add stats from path construction to initial stats
        AlgorithmStats combinedStats = new AlgorithmStats(
            System.currentTimeMillis() - overallStartTime,
            initialStats.getNodesVisited() + pathConstructionStats.getNodesVisited(),
            initialStats.getNodesExpanded() + pathConstructionStats.getNodesExpanded(),
            Math.max(initialStats.getMaxQueueSize(), pathConstructionStats.getMaxQueueSize()),
            fullPath.size(),
            initialStats.getTotalNodes()
        );

        return new PathResult(fullPath, totalDistance, combinedStats);
    }
}
```

HaversineHeuristic.java

```
package path_planning.heuristics;

import path_planning.Graph;

/**
 * A heuristic strategy that calculates the Haversine distance
 * (straight-line distance on a sphere) between two nodes
 * using their geographic coordinates.
 */
public class HaversineHeuristic implements HeuristicStrategy {

    private static final int EARTH_RADIUS_MILES = 3963;
    // Heuristic scaling factor for admissibility in A*.
    // This might need tuning based on the specific graph and edge weights.
    // A value <= 1 helps maintain admissibility if edge costs are actual distances.
    private static final double HEURISTIC_SCALING_FACTOR = 0.8;

    @Override
    public int calculate(Graph graph, String fromNodeId, String toNodeId) {
        if (graph.hasCoordinates(fromNodeId) && graph.hasCoordinates(toNodeId)) {
            double[] fromCoord = graph.getCoordinates(fromNodeId);
            double[] toCoord = graph.getCoordinates(toNodeId);
```

```
            if (fromCoord != null && toCoord != null && fromCoord.length >= 2 && toCoord.length >= 2) {
                double fromLat = fromCoord[0];
                double fromLon = fromCoord[1];
                double toLat = toCoord[0];
                double toLon = toCoord[1];

                double dLat = Math.toRadians(toLat - fromLat);
                double dLon = Math.toRadians(toLon - fromLon);

                double a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
                        Math.cos(Math.toRadians(fromLat)) * Math.cos(Math.toRadians(toLat)) *
                        Math.sin(dLon / 2) * Math.sin(dLon / 2);
                double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
                double distance = EARTH_RADIUS_MILES * c;

                return (int) (distance * HEURISTIC_SCALING_FACTOR);
            }
        }
        // Fallback to zero if coordinates are not available or invalid
        return 0;
    }
}
```

HeuristicStrategy.java

```
package path_planning.heuristics;

import path_planning.Graph;

/**
 * A heuristic strategy that calculates the Haversine distance
 * (straight-line distance on a sphere) between two nodes
 * using their geographic coordinates.
 */
public class HaversineHeuristic implements HeuristicStrategy {

    private static final int EARTH_RADIUS_MILES = 3963;
    // Heuristic scaling factor for admissibility in A*.
    // This might need tuning based on the specific graph and edge weights.
    // A value <= 1 helps maintain admissibility if edge costs are actual distances.
    private static final double HEURISTIC_SCALING_FACTOR = 0.8;

    @Override
    public int calculate(Graph graph, String fromNodeId, String toNodeId) {
        if (graph.hasCoordinates(fromNodeId) && graph.hasCoordinates(toNodeId)) {
            double[] fromCoord = graph.getCoordinates(fromNodeId);
            double[] toCoord = graph.getCoordinates(toNodeId);

            if (fromCoord != null && toCoord != null && fromCoord.length >= 2 && toCoord.length >= 2) {
                double fromLat = fromCoord[0];
                double fromLon = fromCoord[1];
                double toLat = toCoord[0];
                double toLon = toCoord[1];

                double dLat = Math.toRadians(toLat - fromLat);
                double dLon = Math.toRadians(toLon - fromLon);

                double a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
                        Math.cos(Math.toRadians(fromLat)) * Math.cos(Math.toRadians(toLat)) *
                        Math.sin(dLon / 2) * Math.sin(dLon / 2);
                double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
                double distance = EARTH_RADIUS_MILES * c;
```

```
                return (int) (distance * HEURISTIC_SCALING_FACTOR);
            }
        }
        // Fallback to zero if coordinates are not available or invalid
        return 0;
    }
}
```

ZeroHeuristic.java

```
package path_planning.heuristics;

import path_planning.Graph;

/**
 * A heuristic strategy that always returns 0.
 * This makes A* behave like Dijkstra's algorithm.
 */
public class ZeroHeuristic implements HeuristicStrategy {
    @Override
    public int calculate(Graph graph, String fromNodeId, String toNodeId) {
        // The graph and node IDs are not strictly needed for this heuristic,
        // but are included to conform to the interface.
        return 0;
    }
}
```

AlgorithmStats.java

```
package path_planning;

/**
 * Static class to represent algorithm performance statistics
 */
public class AlgorithmStats {
    private long executionTimeMs;          // Execution time (ms)
    private int nodesVisited;              // Number of nodes visited
    private int nodesExpanded;             // Number of nodes expanded
    private int maxQueueSize;              // Maximum queue size
    private int pathLength;                // Path length (nodes)
    private int totalNodes;                // Total nodes in graph (for complexity estimation)

    public AlgorithmStats(long executionTimeMs, int nodesVisited, int nodesExpanded, int maxQueueSize, int
pathLength) {
        this(executionTimeMs, nodesVisited, nodesExpanded, maxQueueSize, pathLength, -1);
    }

    public AlgorithmStats(long executionTimeMs, int nodesVisited, int nodesExpanded, int maxQueueSize, int
pathLength, int totalNodes) {
        this.executionTimeMs = executionTimeMs;
        this.nodesVisited = nodesVisited;
        this.nodesExpanded = nodesExpanded;
        this.maxQueueSize = maxQueueSize;
        this.pathLength = pathLength;
        this.totalNodes = (totalNodes >= 0) ? totalNodes : this.totalNodes; // Preserve if new totalNodes is
invalid
    }

    public long getExecutionTimeMs() { return executionTimeMs; }
    public int getNodesVisited() { return nodesVisited; }
    public int getNodesExpanded() { return nodesExpanded; }
    public int getMaxQueueSize() { return maxQueueSize; }
```

```
    public int getPathLength() { return pathLength; }
    public int getTotalNodes() { return totalNodes; }

    public void setExecutionTimeMs(long executionTimeMs) {
        this.executionTimeMs = executionTimeMs;
    }

    public void setPathLength(int pathLength) {
        this.pathLength = pathLength;
    }

    // Setter for totalNodes if needed, or ensure constructor logic is sufficient
    public void setTotalNodes(int totalNodes) {
        this.totalNodes = totalNodes;
    }

    public void accumulate(AlgorithmStats other) {
        if (other == null) {
            return;
        }
        this.executionTimeMs += other.executionTimeMs;
        this.nodesVisited += other.nodesVisited;
        this.nodesExpanded += other.nodesExpanded;
        this.maxQueueSize = Math.max(this.maxQueueSize, other.maxQueueSize);
        // Path length is usually set for the final path, not accumulated directly unless meaningful.
        // If pathLength is for segments, this might need different logic.
        // For now, let's assume pathLength will be set explicitly at the end for the overall path.
        // this.pathLength += other.pathLength;

        // Total nodes should ideally be set once from the graph and not accumulated.
        // If other.totalNodes is more representative (e.g. set later), update.
        if (other.totalNodes > this.totalNodes) { // Or some other logic to determine the correct totalNodes
            this.totalNodes = other.totalNodes;
        }
    }

    /**
     * Estimates the time complexity based on collected statistics
     * @return String representation of time complexity
     */
    public String getTimeComplexityEstimate() {
        // For pathfinding algorithms like Dijkstra and A*
        double visitRatio = (totalNodes > 0) ? (double) nodesVisited / totalNodes : 0.0; // Avoid division by
zero if totalNodes isn't set

        if (visitRatio > 0.8) {
            return "O(|V|² + |E|)"; // Close to examining all nodes - like Dijkstra without priority queue
        } else if (visitRatio > 0.5) {
            return "O(|E| + |V|log|V|)"; // Examined many nodes - like Dijkstra with priority queue
        } else if (visitRatio > 0.2) {
            return "O(|E| + |V|log|V|)"; // Examined some nodes - like A* with decent heuristic
        } else {
            return "O(|E| + |V|log|V|)"; // Examined few nodes - like A* with good heuristic
        }
    }

    /**
     * Estimates the space complexity based on collected statistics
     * @return String representation of space complexity
     */
    public String getSpaceComplexityEstimate() {
        // For pathfinding algorithms, space complexity is typically related to max queue size
        double queueRatio = (totalNodes > 0) ? (double) maxQueueSize / totalNodes : 0.0; // Avoid division by
zero
```

```
            if (queueRatio > 0.7) {
                return "O(|V|)"; // Large queue – storing most vertices
            } else if (queueRatio > 0.3) {
                return "O(|V|)"; // Medium queue – storing many vertices
            } else {
                return "O(|V|)"; // Small queue – but worst case is still O(|V|)
            }
        }

        @Override
        public String toString() {
            StringBuilder sb = new StringBuilder("Algorithm Performance Statistics:\n");
            sb.append("– Execution Time: ").append(executionTimeMs).append(" ms\n");
            sb.append("– Nodes Visited: ").append(nodesVisited).append("\n");
            sb.append("– Nodes Expanded: ").append(nodesExpanded).append("\n");
            sb.append("– Max Queue Size: ").append(maxQueueSize).append("\n");
            sb.append("– Path Length: ").append(pathLength).append(" nodes\n");

            // Add time and space complexity estimates if total nodes is available
            if (totalNodes > 0) { // Check totalNodes > 0 before using for complexity
                sb.append("– Total Graph Nodes: ").append(totalNodes).append("\n"); // Also print total nodes
                sb.append("– Time Complexity Estimate: ").append(getTimeComplexityEstimate()).append("\n");
                sb.append("– Space Complexity Estimate: ").append(getSpaceComplexityEstimate());
            }

            return sb.toString();
        }
    }
```

Graph.java

```
package path_planning;

import java.util.*;
import utils.DataLoader.RoadConnection;
import java.awt.geom.Point2D;

/**
 * Graph class to represent the road network
 */
public class Graph {
    private Map<String, Map<String, Integer>> adjacencyList;
    private Set<String> nodes;
    private Map<String, double[]> coordinates; // Store coordinates for each city

    /**
     * Constructor initializes an empty graph
     */
    public Graph() {
        adjacencyList = new HashMap<>();
        nodes = new HashSet<>();
        coordinates = new HashMap<>();
    }

    /**
     * Builds the graph from road connections
     * @param roads List of road connections with distances
     */
    public void buildGraph(List<RoadConnection> roads) {
        for (RoadConnection road : roads) {
            String cityA = road.getCityA();
            String cityB = road.getCityB();
```

```
            int distance = road.getDistance();

            // Add nodes to the set
            nodes.add(cityA);
            nodes.add(cityB);

            // Add edges (undirected graph)
            addEdge(cityA, cityB, distance);
            addEdge(cityB, cityA, distance);
        }
    }

    /**
     * Add an edge to the graph
     * @param source Source node
     * @param destination Destination node
     * @param distance Distance between nodes
     */
    private void addEdge(String source, String destination, int distance) {
        adjacencyList.computeIfAbsent(source, k -> new HashMap<>()).put(destination, distance);
    }

    /**
     * Get all nodes in the graph
     * @return Set of all nodes
     */
    public Set<String> getNodes() {
        return nodes;
    }

    /**
     * Get all neighbors of a node with their distances
     * @param node The node to get neighbors for
     * @return Map of neighbor nodes to distances
     */
    public Map<String, Integer> getNeighbors(String node) {
        return adjacencyList.getOrDefault(node, Collections.emptyMap());
    }

    /**
     * Check if the graph contains a node
     * @param node Node to check
     * @return true if the node exists, false otherwise
     */
    public boolean containsNode(String node) {
        return nodes.contains(node);
    }

    /**
     * Get distance between two adjacent nodes
     * @param source Source node
     * @param destination Destination node
     * @return Distance between nodes, or -1 if not connected
     */
    public int getDistance(String source, String destination) {
        if (adjacencyList.containsKey(source) && adjacencyList.get(source).containsKey(destination)) {
            return adjacencyList.get(source).get(destination);
        }
        return -1; // Not directly connected
    }

    /**
     * Set coordinates for a city/node
     * @param node The node/city name
```

```
     * @param lat Latitude
     * @param lon Longitude
     */
    public void setCoordinates(String node, double lat, double lon) {
        coordinates.put(node, new double[] {lat, lon});
    }

    /**
     * Check if coordinates are available for a node
     * @param node The node to check
     * @return true if coordinates exist, false otherwise
     */
    public boolean hasCoordinates(String node) {
        return coordinates.containsKey(node);
    }

    /**
     * Get coordinates for a node
     * @param node The node to get coordinates for
     * @return Array with [latitude, longitude] or null if not found
     */
    public double[] getCoordinates(String node) {
        return coordinates.get(node);
    }

    /**
     * Initialize city coordinates from a map of geographic points
     * @param geoCoordinates Map of city names to Point2D objects with lon/lat
     */
    public void initializeCoordinates(Map<String, Point2D.Double> geoCoordinates) {
        for (Map.Entry<String, Point2D.Double> entry : geoCoordinates.entrySet()) {
            String city = entry.getKey();
            Point2D.Double point = entry.getValue();
            // Point2D has x=longitude, y=latitude
            setCoordinates(city, point.getY(), point.getX());
        }
    }
}
```

NodeDistance.java

```
package path_planning;

import path_planning.heuristics.HeuristicStrategy;
import java.util.Objects;

/**
 * Inner class to represent a node with its distance in the priority queue
 */
public class NodeDistance {
    private String node;
    private int gCost; // Renamed from 'distance' to 'gCost' for clarity in A*

    public NodeDistance(String node, int gCost) {
        this.node = node;
        this.gCost = gCost;
    }

    public String getNode() { return node; }
    public int getDistance() { return gCost; } // Keep getDistance for compatibility if used elsewhere, but it's
gCost
    public int getGCost() { return gCost; }      // New getter for clarity

    // Calculate fCost = gCost + hCost. Heuristic is passed to avoid storing it in NodeDistance
    public int getFCost(HeuristicStrategy heuristic, String targetNode, Graph graph) {
```

```
        if (gCost == Integer.MAX_VALUE) return Integer.MAX_VALUE; // Unreachable
        // Ensure graph is not null before calling methods on it
        if (graph == null) {
            // Or handle this case as an error, throw exception, or return a specific value
            return Integer.MAX_VALUE;
        }
        int hCost = heuristic.calculate(graph, this.node, targetNode);
        if (hCost == Integer.MAX_VALUE) return Integer.MAX_VALUE; // Heuristic indicates unreachable

        // Check for potential overflow if gCost and hCost are large
        if (gCost > Integer.MAX_VALUE - hCost) { // Avoids gCost + hCost > Integer.MAX_VALUE
            return Integer.MAX_VALUE;
        }
        return gCost + hCost;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        NodeDistance that = (NodeDistance) o;
        // Compare only node ID for PriorityQueue contains/remove operations,
        // as gCost might differ for the same node if a shorter path is found.
        return Objects.equals(node, that.node);
    }

    @Override
    public int hashCode() {
        // Hash only node ID for consistency with equals()
        return Objects.hash(node);
    }
}
```

OptimalStrategyType.java

```
package path_planning;

import path_planning.heuristics.HeuristicStrategy;
import java.util.Objects;

/**
 * Inner class to represent a node with its distance in the priority queue
 */
public class NodeDistance {
    private String node;
    private int gCost; // Renamed from 'distance' to 'gCost' for clarity in A*

    public NodeDistance(String node, int gCost) {
        this.node = node;
        this.gCost = gCost;
    }

    public String getNode() { return node; }
    public int getDistance() { return gCost; } // Keep getDistance for compatibility if used elsewhere, but it's
gCost
    public int getGCost() { return gCost; }      // New getter for clarity

    // Calculate fCost = gCost + hCost. Heuristic is passed to avoid storing it in NodeDistance
    public int getFCost(HeuristicStrategy heuristic, String targetNode, Graph graph) {
        if (gCost == Integer.MAX_VALUE) return Integer.MAX_VALUE; // Unreachable
        // Ensure graph is not null before calling methods on it
        if (graph == null) {
            // Or handle this case as an error, throw exception, or return a specific value
            return Integer.MAX_VALUE;
        }
```

```
            int hCost = heuristic.calculate(graph, this.node, targetNode);
            if (hCost == Integer.MAX_VALUE) return Integer.MAX_VALUE; // Heuristic indicates unreachable

            // Check for potential overflow if gCost and hCost are large
            if (gCost > Integer.MAX_VALUE - hCost) { // Avoids gCost + hCost > Integer.MAX_VALUE
                return Integer.MAX_VALUE;
            }
            return gCost + hCost;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;
            NodeDistance that = (NodeDistance) o;
            // Compare only node ID for PriorityQueue contains/remove operations,
            // as gCost might differ for the same node if a shorter path is found.
            return Objects.equals(node, that.node);
        }

        @Override
        public int hashCode() {
            // Hash only node ID for consistency with equals()
            return Objects.hash(node);
        }
    }
```

PathfindingController.java

```
package path_planning; // Correct package

import javax.swing.*;
import java.util.List;
import java.util.ArrayList;
import java.util.stream.Collectors;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.DefaultListModel;
import javax.swing.SwingWorker;

// Remove incorrect default package imports if they existed
// import ControlPanel;
// import MapPanel;
// import ResultsPanel;

// Keep correct imports for View components
import view.ControlPanel;
import view.MapPanel;
import view.ResultsPanel;

// Ensure model and PathResult are imported (PathResult is inner class)
import path_planning.PathfindingModel;
import path_planning.PathResult; // New import for top-level class
import path_planning.OptimalStrategyType; // Import the new enum
import path_planning.heuristics.HaversineHeuristic;
import path_planning.heuristics.HeuristicStrategy;
import path_planning.heuristics.ZeroHeuristic;


public class PathfindingController {

    private PathfindingModel model;
    private ControlPanel controlPanel;
    private MapPanel mapPanel;
    private ResultsPanel resultsPanel;
```

```
    public PathfindingController(PathfindingModel model, ControlPanel controlPanel, MapPanel mapPanel,
ResultsPanel resultsPanel) {
        this.model = model;
        this.controlPanel = controlPanel;
        this.mapPanel = mapPanel;
        this.resultsPanel = resultsPanel;

        // Add action listeners from Controller, not View
        attachListeners();
    }

    private void attachListeners() {
        // Sequential strategies
        controlPanel.getCalculateSequentialDijkstraButton().addActionListener(e ->
handlePathCalculationRequest(false, OptimalStrategyType.PERMUTATIONS, false)); // OptimalStrategyType
is irrelevant for sequential
        controlPanel.getCalculateSequentialAStarButton().addActionListener(e ->
handlePathCalculationRequest(false, OptimalStrategyType.PERMUTATIONS, true));   // OptimalStrategyType
is irrelevant for sequential

        // Optimal (Permutations) strategies
        controlPanel.getCalculateOptimalDijkstraButton().addActionListener(e ->
handlePathCalculationRequest(true, OptimalStrategyType.PERMUTATIONS, false));
        controlPanel.getCalculateOptimalAStarButton().addActionListener(e ->
handlePathCalculationRequest(true, OptimalStrategyType.PERMUTATIONS, true));

        // Optimal (Nearest Neighbor + 2-Opt) strategies
        controlPanel.getCalculateNN2OptDijkstraButton().addActionListener(e ->
handlePathCalculationRequest(true, OptimalStrategyType.NEAREST_NEIGHBOR_2OPT, false));
        controlPanel.getCalculateNN2OptAStarButton().addActionListener(e ->
handlePathCalculationRequest(true, OptimalStrategyType.NEAREST_NEIGHBOR_2OPT, true));

        controlPanel.getResetApplicationButton().addActionListener(e -> resetAll());
    }

    // Reset all selections and path display to default state
    private void resetAll() {
        // Clear the selected POIs
        controlPanel.getSelectedPoisListModel().clear();

        // Reset the dropdowns to first item if available
        if (controlPanel.getStartCityCombo().getItemCount() > 0) {
            controlPanel.getStartCityCombo().setSelectedIndex(0);
        }
        if (controlPanel.getEndCityCombo().getItemCount() > 0) {
            controlPanel.getEndCityCombo().setSelectedIndex(0);
        }
        if (controlPanel.getAvailableAttractionsCombo().getItemCount() > 0) {
            controlPanel.getAvailableAttractionsCombo().setSelectedIndex(0);
        }

        // Clear any displayed path
        mapPanel.displayPath(null, null);

        // Reset the results panel
        resultsPanel.setMessage("All selections have been reset.");
    }

    public void initializeViewData() {
        System.out.println("Controller: Initializing view data...");
        if (model != null) {
            // Set Graph reference FIRST
            if(mapPanel != null) {
                mapPanel.setGraph(model.getGraph()); // Set graph before displaying data
```

```
            }
            // Now populate view components
            controlPanel.populateCities(model.getCityNames());
            controlPanel.populateAttractions(model.getAttractionData());
            // Display initial graph state (nodes/edges might be drawn now)
            mapPanel.displayGraphData(model.getCityNames(), model.getAttractionData());
            System.out.println("Controller: View data initialized.");
        } else {
             System.err.println("Controller: Model is null during view initialization.");
        }
    }

    // Unified handler for all path calculation requests
    private void handlePathCalculationRequest(boolean findOptimalOrder, OptimalStrategyType
optimalStrategy, boolean useAStar) {
        String startCity = (String) controlPanel.getStartCityCombo().getSelectedItem();
        String endCity = (String) controlPanel.getEndCityCombo().getSelectedItem();

        if (startCity == null || endCity == null) {
            resultsPanel.setMessage("Please select start and end cities.");
            return;
        }

        List<String> poiNames = new ArrayList<>();
        DefaultListModel<String> listModel = controlPanel.getSelectedPoisListModel();
        for (int i = 0; i < listModel.getSize(); i++) {
            String displayName = listModel.getElementAt(i);
            String attractionName = extractAttractionName(displayName);
            poiNames.add(attractionName);
        }

        if (startCity.equals(endCity) && poiNames.isEmpty()) {
            resultsPanel.setMessage("Start and end cities are the same, and no places of interest selected.");
            return;
        }

        List<String> poiCities = new ArrayList<>(); // These are the cities for the POIs
        for (String poiName : poiNames) {
            String poiCity = model.getAttractionData().get(poiName);
            if (poiCity != null) {
                poiCities.add(poiCity);
            } else {
                // Check if the poiName itself is a city (selected directly as a city POI)
                if (model.getCityNames() != null && model.getCityNames().contains(poiName)){
                    poiCities.add(poiName); // Add the city name directly
                } else {
                    resultsPanel.setMessage("Error: Location not found for: " + poiName);
                    return;
                }
            }
        }

        HeuristicStrategy heuristic = useAStar ? new HaversineHeuristic() : null;
        String algorithmName = useAStar ? "A*" : "Dijkstra";
        String orderDescription;
        if (findOptimalOrder) {
            orderDescription = optimalStrategy == OptimalStrategyType.NEAREST_NEIGHBOR_2OPT ?
                "Optimal (NN + 2-Opt)" : "Optimal (Permutations)";
        } else {
            orderDescription = "Sequential Order";
        }

        String statusMessage = "Calculating path with " + orderDescription + " using " + algorithmName + "...";
```

```
        executeCalculation(startCity, endCity, poiCities, findOptimalOrder, optimalStrategy, useAStar,
heuristic, statusMessage);
    }

    /**
     * Extract the actual attraction name from display name with city abbreviation
     * @param displayName Format: "Attraction Name (XX)"
     * @return Original attraction name
     */
    private String extractAttractionName(String displayName) {
        if (displayName == null) return "";
        int parenIndex = displayName.lastIndexOf(" (");
        if (parenIndex > 0) {
            return displayName.substring(0, parenIndex);
        }
        return displayName; // Return as is if no parentheses found
    }

    private void executeCalculation(String startCity, String endCity, List<String> poiWaypoints,
                                    boolean findOptimalOrder, OptimalStrategyType optimalStrategy,
                                    boolean useAStar, HeuristicStrategy heuristic, String statusMessage) {

        resultsPanel.setMessage(statusMessage);
        mapPanel.displayPath(null, null);

        final List<String> originalPoiWaypoints = new ArrayList<>(poiWaypoints);

        SwingWorker<PathResult, Void> worker = new SwingWorker<PathResult, Void>() {
            @Override
            protected PathResult doInBackground() throws Exception {
                String optimalStrategyName = findOptimalOrder ? optimalStrategy.toString() : "SEQUENTIAL";
                System.out.println("Controller (Worker): Background calculation started. Strategy: " +
                                   optimalStrategyName + ", Algorithm: " + (useAStar ? "A*" : "Dijkstra") +
".");
                return model.findPath(startCity, endCity, poiWaypoints, findOptimalOrder, optimalStrategy,
useAStar, heuristic);
            }

            @Override
            protected void done() {
                try {
                    PathResult result = model.getLastResult(); // Assumes model stores the last result
                    System.out.println("Controller (Worker): Background calculation finished.");
                    updateView(result, originalPoiWaypoints); // Pass original POIs for display consistency
                } catch (Exception ex) {
                    System.err.println("Controller (Worker): Error during path calculation: " +
ex.getMessage());
                    ex.printStackTrace();
                    resultsPanel.setMessage("Error calculating path: " + ex.getMessage());
                    mapPanel.displayPath(null, null);
                }
            }
        };
        worker.execute();
    }

    // Updates the view components after calculation
    private void updateView(PathResult result, List<String> displayPoiCities) {
        // The displayPoiCities are the actual city names corresponding to selected POIs, used for
highlighting in results
        resultsPanel.displayResult(result, displayPoiCities);
        mapPanel.displayPath(result, displayPoiCities);
    }
}
```

PathfindingModel.java

```
package path_planning; // Place in appropriate package

import utils.DataLoader;
import utils.DataLoader.RoadConnection;
import path_planning.heuristics.HeuristicStrategy;
import path_planning.heuristics.ZeroHeuristic; // Default for Dijkstra via A*
import path_planning.OptimalStrategyType; // Import the enum

import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.awt.geom.Point2D;
import java.util.HashMap;

// --- CHANGED Import for PathResult ---
// import path_planning.PathPlanner.PathResult; // Old import
import path_planning.PathResult; // New import for top-level class

public class PathfindingModel {

    private Graph graph;
    private PathPlanner pathPlanner;
    private Map<String, String> attractions;
    // private List<RoadConnection> roads; // This is loaded but not directly used elsewhere in model, can be kept or removed if only for graph building

    private PathResult lastResult;
    private List<String> lastPoiCities; // Stores the original list of POI *city names* for display purposes

    // Data file paths (could be passed in constructor or configured)
    private static final String ROADS_FILE = "CW3_Data_Files/roads.csv";
    private static final String ATTRACTIONS_FILE = "CW3_Data_Files/attractions.csv";

    // US major cities geographic coordinates
    private Map<String, Point2D.Double> cityGeoCoordinates;

    public PathfindingModel() {
        initializeGeoCoordinates();
    }

    /**
     * Initialize geographic coordinates for major US cities
     */
    private void initializeGeoCoordinates() {
        cityGeoCoordinates = new HashMap<>();

        // Initialize with the same coordinates used in MapPanel
        // Northeast cities
        cityGeoCoordinates.put("New York NY", new Point2D.Double(-70.0, 39.7));
        cityGeoCoordinates.put("Philadelphia PA", new Point2D.Double(-72.2, 39.2));

        // Midwest cities
        cityGeoCoordinates.put("Chicago IL", new Point2D.Double(-80.1, 39.5));
        cityGeoCoordinates.put("Columbus OH", new Point2D.Double(-80.0, 38.0));

        // Southern cities
        cityGeoCoordinates.put("Charlotte NC", new Point2D.Double(-77.8, 35.2));
        cityGeoCoordinates.put("Jacksonville FL", new Point2D.Double(-74.7, 32.3));
        cityGeoCoordinates.put("Dallas TX", new Point2D.Double(-95.8, 33.2));
        cityGeoCoordinates.put("Fort Worth TX", new Point2D.Double(-97.3, 33.8));
        cityGeoCoordinates.put("Houston TX", new Point2D.Double(-95.4, 31.7));
        cityGeoCoordinates.put("San Antonio TX", new Point2D.Double(-98.5, 31.4));
```

```
        cityGeoCoordinates.put("Austin TX", new Point2D.Double(-97.7, 32.3));

        // Western cities
        cityGeoCoordinates.put("Phoenix AZ", new Point2D.Double(-112.1, 34.7));
        cityGeoCoordinates.put("Los Angeles CA", new Point2D.Double(-121.9, 35.3));
        cityGeoCoordinates.put("San Diego CA", new Point2D.Double(-119.2, 34.2));
        cityGeoCoordinates.put("San Jose CA", new Point2D.Double(-124.9, 39.3));
    }

    public boolean loadDataAndBuildGraph() {
        System.out.println("Model: Loading data...");
        List<RoadConnection> roadData = DataLoader.loadRoadData(ROADS_FILE); // Renamed for clarity
        attractions = DataLoader.loadAttractionData(ATTRACTIONS_FILE);

        if (roadData.isEmpty() || attractions.isEmpty()) {
            System.err.println("Model: Error loading data files.");
            return false;
        }
        System.out.println("Model: Data loaded successfully.");

        graph = new Graph();
        graph.buildGraph(roadData);

        // Initialize city coordinates for A* algorithm
        graph.initializeCoordinates(cityGeoCoordinates);

        pathPlanner = new PathPlanner(graph);
        // Configure PathPlanner with a default heuristic for A* if Dijkstra is to be simulated via A*
        // Or ensure PathPlanner can use Dijkstra directly.
        // For now, assuming PathPlanner.findShortestPathWithInterests handles Dijkstra when no heuristic
is given / A* flag is false.
        // And PathPlanner.findShortestPathAStarWithInterests handles A*.
        System.out.println("Model: Graph built and PathPlanner initialized.");
        return true;
    }

    // --- Getters for initial UI setup ---

    public Set<String> getCityNames() {
        return graph != null ? graph.getNodes() : Collections.emptySet();
    }

    public Map<String, String> getAttractionData() {
        return attractions != null ? Collections.unmodifiableMap(attractions) : Collections.emptyMap();
    }

    // --- Pathfinding Execution (called by Controller's SwingWorker) ---

    /**
     * Unified method to find a path, accommodating Dijkstra/A* and Sequential/Optimal POI order.
     *
     * @param startCity           The starting city name.
     * @param endCity             The ending city name.
     * @param poiWaypoints        A list of city names representing the waypoints to visit.
     * @param findOptimalOrder    If true, finds the optimal order for visiting POIs (TSP-like).
     *                            If false, visits POIs in the provided sequential order.
     * @param optimalStrategy     The optimal strategy type to use if optimal order is selected.
     * @param useAStar            If true, uses A* algorithm for path segments.
     *                            If false, uses Dijkstra algorithm for path segments.
     * @param heuristic           The heuristic strategy to use if A* is selected (can be null if useAStar is
false).
     * @return PathResult object containing the path and distance.
     */
    public PathResult findPath(String startCity, String endCity, List<String> poiWaypoints,
```

```
                                    boolean findOptimalOrder, OptimalStrategyType optimalStrategy,
                                    boolean useAStar, HeuristicStrategy heuristic) {
        if (pathPlanner == null) {
            System.err.println("Model: PathPlanner not initialized.");
            this.lastResult = new PathResult(Collections.emptyList(), -1);
            this.lastPoiCities = poiWaypoints; // Store even if error
            return this.lastResult;
        }
        this.lastPoiCities = poiWaypoints;

        // Call the unified method in PathPlanner
        this.lastResult = pathPlanner.findPathWithInterests(startCity, endCity, poiWaypoints,
                                                            findOptimalOrder, optimalStrategy,
                                                            useAStar, heuristic);
        return this.lastResult;
    }

    // Remove or comment out old specific findPath, findPathAStar, findPathAStarWithInterests methods
    // as they are now consolidated into the new findPath method.
    /*
    public PathResult findPath(String start, String end, List<String> poiCities, boolean findOptimal) { ... }
    public PathResult findPathAStar(String start, String end, HeuristicStrategy strategy) { ... }
    public PathResult findPathAStarWithInterests(String start, String end, List<String> poiCities, boolean
findOptimal, HeuristicStrategy strategy) { ... }
    */

    // --- Getters for results ---

    public PathResult getLastResult() {
        return lastResult;
    }

    public List<String> getLastPoiCities() {
        return lastPoiCities;
    }

    // --- ADDED: Getter for the Graph object ---
    public Graph getGraph() {
        return graph;
    }

    // --- ADDED: Getter for city geographic coordinates ---
    public Map<String, Point2D.Double> getCityGeoCoordinates() {
        return Collections.unmodifiableMap(cityGeoCoordinates);
    }
}
```

PathPlanner.java

```
package path_planning;

import path_planning.heuristics.HaversineHeuristic;
import path_planning.heuristics.HeuristicStrategy;
import path_planning.heuristics.ZeroHeuristic;
import algorithm.DijkstraPathfinder;
import algorithm.AStarPathfinder;
import algorithm.TspPathfinder;
import algorithm.NN2OptPathfinder;
import path_planning.OptimalStrategyType;

import java.util.*;

/**
 * PathPlanner class for finding shortest paths in a graph
```

```
 */
public class PathPlanner implements TspPathfinder.PathSegmentProvider,
TspPathfinder.AStarPathSegmentProvider,
                                    NN2OptPathfinder.PathSegmentProvider,
NN2OptPathfinder.AStarPathSegmentProvider {
    private Graph graph;
    private DijkstraPathfinder dijkstraPathfinder;
    private AStarPathfinder aStarPathfinder;
    private TspPathfinder tspPathfinderInstance; // Single instance, configured as needed
    private NN2OptPathfinder nn2OptPathfinderInstance; // New instance for NN2Opt algorithm

    // Cache for computed shortest paths between nodes to avoid re-computation in TSP
    private Map<String, Map<String, PathResult>> pathCache = new HashMap<>();

    // Default heuristic strategy, can be changed or set via controller
    private HeuristicStrategy defaultAStarHeuristic;

    /**
     * Constructor
     * @param graph The graph representing the road network
     */
    public PathPlanner(Graph graph) {
        this.graph = graph;
        this.dijkstraPathfinder = new DijkstraPathfinder(graph);
        this.aStarPathfinder = new AStarPathfinder(graph);

        // Default heuristic for A* segments within TSP, if no other is specified.
        this.defaultAStarHeuristic = new HaversineHeuristic();

        // Initialize TspPathfinder with both capabilities, as PathPlanner implements both interfaces
        this.tspPathfinderInstance = new TspPathfinder(graph, this, this, this.defaultAStarHeuristic);

        // Initialize NN2OptPathfinder with both capabilities
        this.nn2OptPathfinderInstance = new NN2OptPathfinder(graph, this, this, this.defaultAStarHeuristic);
    }

    /**
     * Sets the default heuristic strategy for A*.
     * @param strategy The heuristic strategy to use.
     */
    public void setDefaultAStarHeuristic(HeuristicStrategy strategy) {
        this.defaultAStarHeuristic = (strategy != null) ? strategy : new HaversineHeuristic();
        // Re-configure the Pathfinder instances with the new default heuristic
        this.tspPathfinderInstance = new TspPathfinder(graph, this, this, this.defaultAStarHeuristic);
        this.nn2OptPathfinderInstance = new NN2OptPathfinder(graph, this, this, this.defaultAStarHeuristic);
    }

    /**
     * Clears the internal path cache. Call this if the graph changes.
     */
    public void clearCache() {
        pathCache.clear();
    }

    /**
     * Helper method to get or compute the shortest path between two nodes.
     * Uses a cache to store results.
     * @param from Start node
     * @param to End node
     * @return PathResult containing the shortest path and distance, or empty result if no path.
     */
    private PathResult getShortestPathResult(String from, String to) {
        // Check cache first
        pathCache.computeIfAbsent(from, k -> new HashMap<>());
```

```
        if (pathCache.get(from).containsKey(to)) {
            return pathCache.get(from).get(to);
        }

        // If start and end are the same
        if (from.equals(to)) {
            PathResult sameNodeResult = new PathResult(Collections.singletonList(from), 0, new
AlgorithmStats(0,1,0,1,1));
            pathCache.get(from).put(to, sameNodeResult);
            return sameNodeResult;
        }

        // Compute using findShortestPath if not in cache
        PathResult result = findShortestPath(from, to);

        // Handle no path found
        if (!result.isFound()) {
            PathResult noPathResult = new PathResult(Collections.emptyList(), Integer.MAX_VALUE,
result.getStats());
            pathCache.get(from).put(to, noPathResult);
            return noPathResult;
        }

        pathCache.get(from).put(to, result); // Store in cache
        return result;
    }

    /**
     * Main method to find path with interests, directing to specific TSP or sequential logic.
     * This will replace the older findShortestPathWithInterests and findShortestPathAStarWithInterests.
     */
    public PathResult findPathWithInterests(String start, String end, List<String> placesOfInterest,
                                            boolean findOptimalOrder, OptimalStrategyType
optimalStrategy,
                                            boolean useAStarForSegments, HeuristicStrategy
heuristicForSegments) {
        clearCache();

        HeuristicStrategy effectiveHeuristic = useAStarForSegments ?
            ((heuristicForSegments != null) ? heuristicForSegments : this.defaultAStarHeuristic)
            : null;

        if (effectiveHeuristic == null && useAStarForSegments) {
            System.err.println("Warning: A* selected but no heuristic provided/defaulted. Using
ZeroHeuristic.");
            effectiveHeuristic = new ZeroHeuristic();
        }

        if (!findOptimalOrder || placesOfInterest == null || placesOfInterest.isEmpty()) {
            // Sequential order or direct path (0 POIs)
            if (useAStarForSegments) {
                return tspPathfinderInstance.findPathWithInterestsSequentialAStar(start, end,
placesOfInterest, effectiveHeuristic);
            } else {
                return tspPathfinderInstance.findPathWithInterestsSequential(start, end, placesOfInterest);
            }
        } else {
            // Optimal order for POIs
            switch (optimalStrategy) {
                case PERMUTATIONS:
                    if (useAStarForSegments) {
                        return tspPathfinderInstance.findOptimalTspPathAStar(start, end, placesOfInterest,
effectiveHeuristic);
                    } else {
```

```
                        return tspPathfinderInstance.findOptimalTspPath(start, end, placesOfInterest);
                    }
                case NEAREST_NEIGHBOR_2OPT:
                    if (useAStarForSegments) {
                        return nn2OptPathfinderInstance.findOptimalPathAStar(start, end, placesOfInterest,
effectiveHeuristic);
                    } else {
                        return nn2OptPathfinderInstance.findOptimalPath(start, end, placesOfInterest);
                    }
                default:
                    System.err.println("Error: Unknown optimal strategy type: " + optimalStrategy);
                    // Fallback to sequential Dijkstra as a safe default if strategy is unknown
                    return tspPathfinderInstance.findPathWithInterestsSequential(start, end,
placesOfInterest);
            }
        }
    }

    // This method remains for direct two-point Dijkstra, PathSegmentProvider calls this.
    public PathResult findShortestPath(String start, String end) {
        return dijkstraPathfinder.findShortestPath(start, end);
    }

    // This method remains for direct two-point A*, AStarPathSegmentProvider calls this.
    public PathResult findShortestPathAStar(String start, String end, HeuristicStrategy heuristicStrategy) {
        if (heuristicStrategy == null) {
            System.err.println("PathPlanner (Direct A*): A* called without heuristic, using default or
ZeroHeuristic.");
            heuristicStrategy = this.defaultAStarHeuristic != null ? this.defaultAStarHeuristic : new
ZeroHeuristic();
        }
        return aStarPathfinder.findShortestPathAStar(start, end, heuristicStrategy);
    }

    // Implementation for TspPathfinder.PathSegmentProvider and NN2OptPathfinder.PathSegmentProvider
    @Override
    public PathResult getShortestPathSegment(String from, String to) {
        pathCache.computeIfAbsent(from, k -> new HashMap<>());
        if (pathCache.get(from).containsKey(to)) {
            return pathCache.get(from).get(to);
        }
        if (from.equals(to)) {
            PathResult sameNodeResult = new PathResult(Collections.singletonList(from), 0, new
AlgorithmStats(0,1,0,1,1));
            pathCache.get(from).put(to, sameNodeResult);
            return sameNodeResult;
        }
        PathResult result = dijkstraPathfinder.findShortestPath(from, to);
        if (!result.isFound()) {
            result = new PathResult(Collections.emptyList(), Integer.MAX_VALUE, result.getStats());
        }
        pathCache.get(from).put(to, result);
        return result;
    }

    // Implementation for TspPathfinder.AStarPathSegmentProvider and
NN2OptPathfinder.AStarPathSegmentProvider
    @Override
    public PathResult getShortestPathAStarSegment(String from, String to, HeuristicStrategy strategy) {
        HeuristicStrategy heuristicToUse = (strategy != null) ? strategy : this.defaultAStarHeuristic;
        if (heuristicToUse == null) {
            System.err.println("PathPlanner (A* Segment): Heuristic is null, using ZeroHeuristic.");
            heuristicToUse = new ZeroHeuristic();
        }
```

```
        // Cache key for A* includes heuristic type for uniqueness
        String heuristicName = heuristicToUse.getClass().getSimpleName();
        String aStarCacheKey = to + "_A*_" + heuristicName;
        pathCache.computeIfAbsent(from, k -> new HashMap<>());
        if (pathCache.get(from).containsKey(aStarCacheKey)) {
            return pathCache.get(from).get(aStarCacheKey);
        }

        if (from.equals(to)) {
            PathResult sameNodeResult = new PathResult(Collections.singletonList(from), 0, new
AlgorithmStats(0,1,0,1,1));
            pathCache.get(from).put(aStarCacheKey, sameNodeResult);
            return sameNodeResult;
        }

        PathResult result = aStarPathfinder.findShortestPathAStar(from, to, heuristicToUse);
        if (!result.isFound()) {
            result = new PathResult(Collections.emptyList(), Integer.MAX_VALUE, result.getStats());
        }
        pathCache.get(from).put(aStarCacheKey, result);
        return result;
    }

    // generatePermutations and other private helpers can remain as they are, if still used by TspPathfinder
internally or directly.
    // If TspPathfinder now handles its own permutations, this can be removed from PathPlanner.
    // For now, assuming TspPathfinder might still rely on this or a similar utility if it doesn't have its own.
    private <T> List<List<T>> generatePermutations(List<T> original) {
        if (original.isEmpty()) {
            List<List<T>> result = new ArrayList<>();
            result.add(new ArrayList<>());
            return result;
        }
        T firstElement = original.remove(0);
        List<List<T>> returnValue = new ArrayList<>();
        List<List<T>> permutations = generatePermutations(original);
        for (List<T> smallerPermutated : permutations) {
            for (int index = 0; index <= smallerPermutated.size(); index++) {
                List<T> temp = new ArrayList<>(smallerPermutated);
                temp.add(index, firstElement);
                returnValue.add(temp);
            }
        }
        return returnValue;
    }
}
```

PathResult.java

```
package path_planning;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * Static class to represent the result of a path calculation
 */
public class PathResult {
    private final List<String> path;
    private final int totalDistance; // Use Integer.MAX_VALUE to indicate no path found in TSP context
    private final AlgorithmStats stats; // 新增算法统计信息
```

```
    public PathResult(List<String> path, int totalDistance) {
        // Provide default performance statistics data for backward compatibility
        this(path, totalDistance, new AlgorithmStats(0, 0, 0, 0, path != null ? path.size() : 0));
    }

    public PathResult(List<String> path, int totalDistance, AlgorithmStats stats) {
        // Ensure path is unmodifiable and a defensive copy is made
        this.path = (path != null) ? Collections.unmodifiableList(new ArrayList<>(path)) :
Collections.emptyList();
        this.totalDistance = totalDistance;
        this.stats = stats;
    }

    public List<String> getPath() {
        return path;
    }

    public int getTotalDistance() {
        return totalDistance;
    }

    public AlgorithmStats getStats() {
        return stats;
    }

    public boolean isFound() {
        return path != null && !path.isEmpty() && totalDistance != Integer.MAX_VALUE && totalDistance >= 0;
    }
}
```

DataLoader.java

```
package utils;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Utility class to load data from CSV files
 */
public class DataLoader {

    /**
     * Loads road data from CSV file
     * @param filePath path to the roads.csv file
     * @return List of road connections with distances
     */
    public static List<RoadConnection> loadRoadData(String filePath) {
        List<RoadConnection> roads = new ArrayList<>();

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            // Skip header
            String line = br.readLine();

            while ((line = br.readLine()) != null) {
                String[] parts = line.split(",");
                if (parts.length == 3) {
                    String cityA = parts[0];
                    String cityB = parts[1];
                    int distance = Integer.parseInt(parts[2]);
```

```
                    roads.add(new RoadConnection(cityA, cityB, distance));
                }
            }
        } catch (IOException e) {
            System.err.println("Error reading road data: " + e.getMessage());
        }

        return roads;
    }

    /**
     * Loads attraction data from CSV file
     * @param filePath path to the attractions.csv file
     * @return Map of attractions to their locations
     */
    public static Map<String, String> loadAttractionData(String filePath) {
        Map<String, String> attractions = new HashMap<>();

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            // Skip header
            String line = br.readLine();

            while ((line = br.readLine()) != null) {
                String[] parts = line.split(",");
                if (parts.length == 2) {
                    String attraction = parts[0];
                    String location = parts[1];
                    attractions.put(attraction, location);
                }
            }
        } catch (IOException e) {
            System.err.println("Error reading attraction data: " + e.getMessage());
        }

        return attractions;
    }

    /**
     * Loads place names from CSV file for sorting tasks
     * @param filePath path to the places CSV file
     * @return List of place names
     */
    public static List<String> loadPlaceNames(String filePath) {
        List<String> places = new ArrayList<>();

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;
            // Assuming no header in places files
            while ((line = br.readLine()) != null) {
                if (!line.isEmpty()) {
                    places.add(line);
                }
            }
        } catch (IOException e) {
            System.err.println("Error reading place names: " + e.getMessage());
        }

        return places;
    }

    /**
     * Inner class to represent a road connection between two cities
     */
    public static class RoadConnection {
```

```
        private String cityA;
        private String cityB;
        private int distance;

        public RoadConnection(String cityA, String cityB, int distance) {
            this.cityA = cityA;
            this.cityB = cityB;
            this.distance = distance;
        }

        public String getCityA() {
            return cityA;
        }

        public String getCityB() {
            return cityB;
        }

        public int getDistance() {
            return distance;
        }
    }
}
```

MapGenerator.java

```
import javax.imageio.ImageIO;
import java.awt.*;
import java.awt.geom.Path2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

/**
 * Simple US Map Generator, creates a basic US map background image
 */
public class MapGenerator {

    // USA map approximate outline coordinates (simplified version)
    private static final int[][] US_OUTLINE_POINTS = {
        {124, 36}, {124, 49}, // West coast northern
        {116, 49}, {110, 45}, {105, 45}, {102, 43}, {95, 49}, // Northern border
        {83, 46}, {81, 42}, {77, 45}, {67, 45}, {67, 42}, // Northeast
        {79, 34}, {80, 31}, {85, 31}, {88, 30}, {83, 25}, // Southeast
        {98, 26}, {107, 31}, {115, 32}, {124, 34} // South and Southwest
    };

    // US Great Lakes approximate outline
    private static final int[][] GREAT_LAKES_POINTS = {
        {82, 42}, {84, 43}, {87, 45}, {85, 47}, {82, 47}, {80, 44}
    };

    public static void main(String[] args) {
        // Create image, width 900 height 750
        BufferedImage usMap = new BufferedImage(900, 750, BufferedImage.TYPE_INT_ARGB);
        Graphics2D g2d = usMap.createGraphics();

        // Enable anti-aliasing
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);

        // Create background
        GradientPaint bgGradient = new GradientPaint(
            0, 0, new Color(220, 240, 255),
            900, 750, new Color(200, 220, 255)
```

```
        );
        g2d.setPaint(bgGradient);
        g2d.fillRect(0, 0, 900, 750);

        // Draw grid lines
        g2d.setColor(new Color(210, 230, 245));
        int gridSize = 50;
        for (int x = 0; x < 900; x += gridSize) {
            g2d.drawLine(x, 0, x, 750);
        }
        for (int y = 0; y < 750; y += gridSize) {
            g2d.drawLine(0, y, 900, y);
        }

        // Scaling factor to convert geographic coordinates to image coordinates
        int xOffset = 250; // Horizontal offset
        int yOffset = 150; // Vertical offset
        int scale = 5;     // Scale factor

        // Create US outline path
        Path2D usOutline = new Path2D.Double();
        usOutline.moveTo(US_OUTLINE_POINTS[0][0] * scale + xOffset,
                         US_OUTLINE_POINTS[0][1] * scale + yOffset);

        for (int i = 1; i < US_OUTLINE_POINTS.length; i++) {
            usOutline.lineTo(US_OUTLINE_POINTS[i][0] * scale + xOffset,
                             US_OUTLINE_POINTS[i][1] * scale + yOffset);
        }
        usOutline.closePath();

        // Create Great Lakes area path
        Path2D greatLakes = new Path2D.Double();
        greatLakes.moveTo(GREAT_LAKES_POINTS[0][0] * scale + xOffset,
                          GREAT_LAKES_POINTS[0][1] * scale + yOffset);

        for (int i = 1; i < GREAT_LAKES_POINTS.length; i++) {
            greatLakes.lineTo(GREAT_LAKES_POINTS[i][0] * scale + xOffset,
                              GREAT_LAKES_POINTS[i][1] * scale + yOffset);
        }
        greatLakes.closePath();

        // Fill US outline
        g2d.setColor(new Color(245, 245, 245));
        g2d.fill(usOutline);
        g2d.setColor(new Color(220, 220, 220));
        g2d.setStroke(new BasicStroke(1.5f));
        g2d.draw(usOutline);

        // Fill Great Lakes area
        g2d.setColor(new Color(200, 225, 255));
        g2d.fill(greatLakes);
        g2d.setColor(new Color(180, 210, 240));
        g2d.draw(greatLakes);

        // Add major city markers (simple implementation)
        g2d.setColor(new Color(100, 100, 100));
        g2d.setFont(new Font("Arial", Font.PLAIN, 10));

        // Processing completed
        g2d.dispose();

        // Save the image
        try {
            File outputDir = new File("CW3_Data_Files/resources");
```

```
            if (!outputDir.exists()) {
                outputDir.mkdirs();
            }

            File outputFile = new File("CW3_Data_Files/resources/us_map.png");
            ImageIO.write(usMap, "png", outputFile);
            System.out.println("US map background generated: " + outputFile.getAbsolutePath());
        } catch (IOException e) {
            System.err.println("Error saving map image: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

PathFinderGUI.java

```
import javax.imageio.ImageIO;
import java.awt.*;
import java.awt.geom.Path2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

/**
 * Simple US Map Generator, creates a basic US map background image
 */
public class MapGenerator {

    // USA map approximate outline coordinates (simplified version)
    private static final int[][] US_OUTLINE_POINTS = {
        {124, 36}, {124, 49}, // West coast northern
        {116, 49}, {110, 45}, {105, 45}, {102, 43}, {95, 49}, // Northern border
        {83, 46}, {81, 42}, {77, 45}, {67, 45}, {67, 42}, // Northeast
        {79, 34}, {80, 31}, {85, 31}, {88, 30}, {83, 25}, // Southeast
        {98, 26}, {107, 31}, {115, 32}, {124, 34} // South and Southwest
    };

    // US Great Lakes approximate outline
    private static final int[][] GREAT_LAKES_POINTS = {
        {82, 42}, {84, 43}, {87, 45}, {85, 47}, {82, 47}, {80, 44}
    };

    public static void main(String[] args) {
        // Create image, width 900 height 750
        BufferedImage usMap = new BufferedImage(900, 750, BufferedImage.TYPE_INT_ARGB);
        Graphics2D g2d = usMap.createGraphics();

        // Enable anti-aliasing
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);

        // Create background
        GradientPaint bgGradient = new GradientPaint(
            0, 0, new Color(220, 240, 255),
            900, 750, new Color(200, 220, 255)
        );
        g2d.setPaint(bgGradient);
        g2d.fillRect(0, 0, 900, 750);

        // Draw grid lines
        g2d.setColor(new Color(210, 230, 245));
        int gridSize = 50;
        for (int x = 0; x < 900; x += gridSize) {
            g2d.drawLine(x, 0, x, 750);
        }
        for (int y = 0; y < 750; y += gridSize) {
```

```
        g2d.drawLine(0, y, 900, y);
    }

    // Scaling factor to convert geographic coordinates to image coordinates
    int xOffset = 250; // Horizontal offset
    int yOffset = 150; // Vertical offset
    int scale = 5;     // Scale factor

    // Create US outline path
    Path2D usOutline = new Path2D.Double();
    usOutline.moveTo(US_OUTLINE_POINTS[0][0] * scale + xOffset,
                     US_OUTLINE_POINTS[0][1] * scale + yOffset);

    for (int i = 1; i < US_OUTLINE_POINTS.length; i++) {
        usOutline.lineTo(US_OUTLINE_POINTS[i][0] * scale + xOffset,
                         US_OUTLINE_POINTS[i][1] * scale + yOffset);
    }
    usOutline.closePath();

    // Create Great Lakes area path
    Path2D greatLakes = new Path2D.Double();
    greatLakes.moveTo(GREAT_LAKES_POINTS[0][0] * scale + xOffset,
                      GREAT_LAKES_POINTS[0][1] * scale + yOffset);

    for (int i = 1; i < GREAT_LAKES_POINTS.length; i++) {
        greatLakes.lineTo(GREAT_LAKES_POINTS[i][0] * scale + xOffset,
                          GREAT_LAKES_POINTS[i][1] * scale + yOffset);
    }
    greatLakes.closePath();

    // Fill US outline
    g2d.setColor(new Color(245, 245, 245));
    g2d.fill(usOutline);
    g2d.setColor(new Color(220, 220, 220));
    g2d.setStroke(new BasicStroke(1.5f));
    g2d.draw(usOutline);

    // Fill Great Lakes area
    g2d.setColor(new Color(200, 225, 255));
    g2d.fill(greatLakes);
    g2d.setColor(new Color(180, 210, 240));
    g2d.draw(greatLakes);

    // Add major city markers (simple implementation)
    g2d.setColor(new Color(100, 100, 100));
    g2d.setFont(new Font("Arial", Font.PLAIN, 10));

    // Processing completed
    g2d.dispose();

    // Save the image
    try {
        File outputDir = new File("CW3_Data_Files/resources");
        if (!outputDir.exists()) {
            outputDir.mkdirs();
        }

        File outputFile = new File("CW3_Data_Files/resources/us_map.png");
        ImageIO.write(usMap, "png", outputFile);
        System.out.println("US map background generated: " + outputFile.getAbsolutePath());
    } catch (IOException e) {
        System.err.println("Error saving map image: " + e.getMessage());
        e.printStackTrace();
    }
```

```
    }
}
```

ResultsPanel.java

```
import javax.imageio.ImageIO;
import java.awt.*;
import java.awt.geom.Path2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

/**
 * Simple US Map Generator, creates a basic US map background image
 */
public class MapGenerator {

    // USA map approximate outline coordinates (simplified version)
    private static final int[][] US_OUTLINE_POINTS = {
        {124, 36}, {124, 49}, // West coast northern
        {116, 49}, {110, 45}, {105, 45}, {102, 43}, {95, 49}, // Northern border
        {83, 46}, {81, 42}, {77, 45}, {67, 45}, {67, 42}, // Northeast
        {79, 34}, {80, 31}, {85, 31}, {88, 30}, {83, 25}, // Southeast
        {98, 26}, {107, 31}, {115, 32}, {124, 34} // South and Southwest
    };

    // US Great Lakes approximate outline
    private static final int[][] GREAT_LAKES_POINTS = {
        {82, 42}, {84, 43}, {87, 45}, {85, 47}, {82, 47}, {80, 44}
    };

    public static void main(String[] args) {
        // Create image, width 900 height 750
        BufferedImage usMap = new BufferedImage(900, 750, BufferedImage.TYPE_INT_ARGB);
        Graphics2D g2d = usMap.createGraphics();

        // Enable anti-aliasing
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);

        // Create background
        GradientPaint bgGradient = new GradientPaint(
            0, 0, new Color(220, 240, 255),
            900, 750, new Color(200, 220, 255)
        );
        g2d.setPaint(bgGradient);
        g2d.fillRect(0, 0, 900, 750);

        // Draw grid lines
        g2d.setColor(new Color(210, 230, 245));
        int gridSize = 50;
        for (int x = 0; x < 900; x += gridSize) {
            g2d.drawLine(x, 0, x, 750);
        }
        for (int y = 0; y < 750; y += gridSize) {
            g2d.drawLine(0, y, 900, y);
        }

        // Scaling factor to convert geographic coordinates to image coordinates
        int xOffset = 250; // Horizontal offset
        int yOffset = 150; // Vertical offset
        int scale = 5;     // Scale factor

        // Create US outline path
        Path2D usOutline = new Path2D.Double();
        usOutline.moveTo(US_OUTLINE_POINTS[0][0] * scale + xOffset,
                         US_OUTLINE_POINTS[0][1] * scale + yOffset);
```

```
        for (int i = 1; i < US_OUTLINE_POINTS.length; i++) {
            usOutline.lineTo(US_OUTLINE_POINTS[i][0] * scale + xOffset,
                             US_OUTLINE_POINTS[i][1] * scale + yOffset);
        }
        usOutline.closePath();

        // Create Great Lakes area path
        Path2D greatLakes = new Path2D.Double();
        greatLakes.moveTo(GREAT_LAKES_POINTS[0][0] * scale + xOffset,
                          GREAT_LAKES_POINTS[0][1] * scale + yOffset);

        for (int i = 1; i < GREAT_LAKES_POINTS.length; i++) {
            greatLakes.lineTo(GREAT_LAKES_POINTS[i][0] * scale + xOffset,
                              GREAT_LAKES_POINTS[i][1] * scale + yOffset);
        }
        greatLakes.closePath();

        // Fill US outline
        g2d.setColor(new Color(245, 245, 245));
        g2d.fill(usOutline);
        g2d.setColor(new Color(220, 220, 220));
        g2d.setStroke(new BasicStroke(1.5f));
        g2d.draw(usOutline);

        // Fill Great Lakes area
        g2d.setColor(new Color(200, 225, 255));
        g2d.fill(greatLakes);
        g2d.setColor(new Color(180, 210, 240));
        g2d.draw(greatLakes);

        // Add major city markers (simple implementation)
        g2d.setColor(new Color(100, 100, 100));
        g2d.setFont(new Font("Arial", Font.PLAIN, 10));

        // Processing completed
        g2d.dispose();

        // Save the image
        try {
            File outputDir = new File("CW3_Data_Files/resources");
            if (!outputDir.exists()) {
                outputDir.mkdirs();
            }

            File outputFile = new File("CW3_Data_Files/resources/us_map.png");
            ImageIO.write(usMap, "png", outputFile);
            System.out.println("US map background generated: " + outputFile.getAbsolutePath());
        } catch (IOException e) {
            System.err.println("Error saving map image: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

TestClient.java

```
import algorithm.NN2OptPathfinder;
import algorithm.TspPathfinder;
import path_planning.*;
import path_planning.heuristics.HaversineHeuristic;
import path_planning.heuristics.HeuristicStrategy;

import java.util.ArrayList;
import java.util.List;
```

```java
/**
 * Simple test client to verify that both TSP and NN2Opt implementations work correctly after refactoring.
 */
public class TestClient {
    public static void main(String[] args) {
        // Initialize model and data
        PathfindingModel model = new PathfindingModel();
        boolean dataLoaded = model.loadDataAndBuildGraph();

        if (!dataLoaded) {
            System.err.println("Error: Failed to load data");
            System.exit(1);
        }

        // Test parameters
        String startCity = "Houston TX";
        String endCity = "Philadelphia PA";
        List<String> pois = new ArrayList<>();
        pois.add("Los Angeles CA");   // Stand-in for an attraction in LA
        pois.add("Chicago IL");       // Stand-in for an attraction in Chicago

        HeuristicStrategy heuristic = new HaversineHeuristic();

        System.out.println("Starting test with:");
        System.out.println("Start: " + startCity);
        System.out.println("End: " + endCity);
        System.out.println("POIs: " + pois);
        System.out.println();

        // Test Dijkstra with Permutations
        System.out.println("=== Testing Dijkstra with Permutations (TSP) ===");
        PathResult result1 = model.findPath(startCity, endCity, pois, true,
OptimalStrategyType.PERMUTATIONS, false, null);
        printResult(result1);

        // Test A* with Permutations
        System.out.println("\n=== Testing A* with Permutations (TSP) ===");
        PathResult result2 = model.findPath(startCity, endCity, pois, true,
OptimalStrategyType.PERMUTATIONS, true, heuristic);
        printResult(result2);

        // Test Dijkstra with NN2Opt
        System.out.println("\n=== Testing Dijkstra with NN2Opt ===");
        PathResult result3 = model.findPath(startCity, endCity, pois, true,
OptimalStrategyType.NEAREST_NEIGHBOR_2OPT, false, null);
        printResult(result3);

        // Test A* with NN2Opt
        System.out.println("\n=== Testing A* with NN2Opt ===");
        PathResult result4 = model.findPath(startCity, endCity, pois, true,
OptimalStrategyType.NEAREST_NEIGHBOR_2OPT, true, heuristic);
        printResult(result4);

        // Compare results
        System.out.println("\n=== Results Comparison ===");
        System.out.println("Permutation-based TSP (Dijkstra): " + result1.getTotalDistance());
        System.out.println("Permutation-based TSP (A*): " + result2.getTotalDistance());
        System.out.println("NN2Opt (Dijkstra): " + result3.getTotalDistance());
        System.out.println("NN2Opt (A*): " + result4.getTotalDistance());
    }

    private static void printResult(PathResult result) {
        if (result == null) {
```

```
            System.out.println("Null result");
            return;
        }

        if (!result.isFound()) {
            System.out.println("No path found");
            return;
        }

        System.out.println("Path found with distance: " + result.getTotalDistance());
        System.out.println("Path length: " + result.getPath().size() + " nodes");

        // Print stats
        AlgorithmStats stats = result.getStats();
        if (stats != null) {
            System.out.println("Stats: ");
            System.out.println("   Execution time: " + stats.getExecutionTimeMs() + " ms");
            System.out.println("   Nodes visited: " + stats.getNodesVisited());
            System.out.println("   Nodes expanded: " + stats.getNodesExpanded());
            System.out.println("   Max queue size: " + stats.getMaxQueueSize());
        }

        // Print a few nodes from the path for verification
        List<String> path = result.getPath();
        System.out.println("Path preview:");
        System.out.println("   Start: " + path.get(0));
        System.out.println("   Middle: " + path.get(path.size() / 2));
        System.out.println("   End: " + path.get(path.size() - 1));
    }
}
```

TaskC_Sorting/src/CSVReader.java

```
import java.io.*;
import java.util.*;

public class CSVReader {
    public static String[] readPlacesFromFile(String filePath) {
        List<String> places = new ArrayList<>();
        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = br.readLine()) != null) {
                places.add(line.trim());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return places.toArray(new String[0]);
    }
}
```

src/DuplicateValueSortTest.java

```
public class DuplicateValueSortTest {

    // Generate duplicated Data
    public static String[] generateDuplicateData(int size) {
        String[] data = new String[size];
        for (int i = 0; i < size; i++) {
            if (i % 3 == 0) {
                data[i] = "Springfield";
            } else if (i % 3 == 1) {
                data[i] = "Austin";
```

```
            } else {
                data[i] = "Boston";
            }
        }
        return data;
    }

    // Test and print sorting time-consuming
    public static void testAndPrint(String[] data, String sortName) {
        String[] clone = data.clone();
        long start = System.nanoTime();

        switch (sortName) {
            case "insertion":
                Sorter.insertionSort(clone);
                break;
            case "quick":
                Sorter.quickSort(clone, 0, clone.length - 1);
                break;
            case "merge":
                Sorter.mergeSort(clone, 0, clone.length - 1);
                break;
            default:
                System.out.println("Unknown sort: " + sortName);
        }

        long end = System.nanoTime();
        double timeInMs = (end - start) / 1e6;
        System.out.printf("%s Sort Time: %.4f ms%n", sortName, timeInMs);
    }

    public static void main(String[] args) {
        int[] testSizes = {1000, 10000};

        for (int size : testSizes) {
            System.out.println("=== Duplicate Dataset Test (" + size + " items) ===");
            String[] duplicateData = generateDuplicateData(size);
            testAndPrint(duplicateData, "insertion");
            testAndPrint(duplicateData, "quick");
            testAndPrint(duplicateData, "merge");
            System.out.println();
        }
    }
}
```

src/Main.java

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        String[] sorted1000 = CSVReader.readPlacesFromFile("src/data/1000places_sorted.csv");
        String[] random1000 = CSVReader.readPlacesFromFile("src/data/1000places_random.csv");
        String[] sorted10000 = CSVReader.readPlacesFromFile("src/data/10000places_sorted.csv");
        String[] random10000 = CSVReader.readPlacesFromFile("src/data/10000places_random.csv");


        testAllAlgorithms("1000 Sorted", sorted1000);
        testAllAlgorithms("1000 Random", random1000);
        testAllAlgorithms("10000 Sorted", sorted10000);
        testAllAlgorithms("10000 Random", random10000);
    }

    public static void testAllAlgorithms(String label, String[] data) {
        int runs = 10;
```

```
        System.out.println("Processing file: " + label);
        System.out.println("Data size: " + data.length);

        // Insertion Sort
        warmUp(() -> Sorter.insertionSort(Arrays.copyOf(data, data.length)));
        long insertionTotal = 0;
        for (int i = 0; i < runs; i++) {
            String[] input = Arrays.copyOf(data, data.length);
            long start = System.nanoTime();
            Sorter.insertionSort(input);
            long end = System.nanoTime();
            insertionTotal += (end - start);
        }
        double insertionAvg = insertionTotal / runs / 1_000_000.0;
        System.out.printf("   Insertion Sort       : %.4f ms%n", insertionAvg);

        // Quick Sort
        warmUp(() -> Sorter.quickSort(Arrays.copyOf(data, data.length), 0, data.length - 1));
        long quickTotal = 0;
        for (int i = 0; i < runs; i++) {
            String[] input = Arrays.copyOf(data, data.length);
            long start = System.nanoTime();
            Sorter.quickSort(input, 0, input.length - 1);
            long end = System.nanoTime();
            quickTotal += (end - start);
        }
        double quickAvg = quickTotal / runs / 1_000_000.0;
        System.out.printf("   Quick Sort           : %.4f ms%n", quickAvg);

        // Merge Sort
        warmUp(() -> Sorter.mergeSort(Arrays.copyOf(data, data.length), 0, data.length - 1));
        long mergeTotal = 0;
        for (int i = 0; i < runs; i++) {
            String[] input = Arrays.copyOf(data, data.length);
            long start = System.nanoTime();
            Sorter.mergeSort(input, 0, input.length - 1);
            long end = System.nanoTime();
            mergeTotal += (end - start);
        }
        double mergeAvg = mergeTotal / runs / 1_000_000.0;
        System.out.printf("   Merge Sort           : %.4f ms%n", mergeAvg);

        System.out.println();
    }


    // Warm Up
    private static void warmUp(Runnable sortingTask) {
        sortingTask.run(); // Run only once
    }
}
```

Sorter.java

```
public class Sorter {
    //
    /** mergeSort Total bytes allocated for all auxiliary arrays */
    private static long extraMemoryBytes = 0;
    /** quickSort Current recursion depth */
    private static int currentRecursionDepth = 0;
    /** quickSort Maximum recursion depth reached */
    private static int maxRecursionDepth = 0;

    // —— 统计方法 ——
```

```
    /** Reset all statistics */
    public static void resetMemoryStats() {
        extraMemoryBytes = 0;
        currentRecursionDepth = 0;
        maxRecursionDepth = 0;
    }

    public static long getExtraMemoryBytes() {
        return extraMemoryBytes;
    }

    public static int getMaxRecursionDepth() {
        return maxRecursionDepth;
    }

    public static long getQuickStackBytes() {
        return (long) maxRecursionDepth * 64;
    }

    // Inserting Sort
    public static void insertionSort(String[] arr) {
        for (int i = 1; i < arr.length; i++) {
            String key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j].compareTo(key) > 0) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }

    // Quick sort
    public static void quickSort(String[] arr, int low, int high) {
        if (low < high) {
            currentRecursionDepth++;
            if (currentRecursionDepth > maxRecursionDepth) {
                maxRecursionDepth = currentRecursionDepth;
            }

            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);

            currentRecursionDepth--;
        }
    }

    private static int partition(String[] arr, int low, int high) {
        // choose the median of the low, mid, and high positions as the pivot.
        int mid = low + (high - low) / 2;
        String a = arr[low];
        String b = arr[mid];
        String c = arr[high];

        int pivotIndex;

        if (a.compareTo(b) < 0) {
            if (b.compareTo(c) < 0) {
                pivotIndex = mid; // a < b < c
            } else if (a.compareTo(c) < 0) {
                pivotIndex = high; // a < c < b
            } else {
                pivotIndex = low; // c < a < b
```

```
            }
        } else {
            if (a.compareTo(c) < 0) {
                pivotIndex = low; // b < a < c
            } else if (b.compareTo(c) < 0) {
                pivotIndex = high; // b < c < a
            } else {
                pivotIndex = mid; // c < b < a
            }
        }

        // Swap the median value to the end (high position) as pivot
        String temp = arr[pivotIndex];
        arr[pivotIndex] = arr[high];
        arr[high] = temp;


        String pivot = arr[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j].compareTo(pivot) <= 0) {
                i++;
                String t = arr[i];
                arr[i] = arr[j];
                arr[j] = t;
            }
        }

        String t = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = t;

        return i + 1;
    }


    // Merge Sort
    public static void mergeSort(String[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    private static void merge(String[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;
        String[] L = new String[n1];
        String[] R = new String[n2];
        extraMemoryBytes += (long)(n1 + n2) * 8;

        System.arraycopy(arr, left, L, 0, n1);
        System.arraycopy(arr, mid + 1, R, 0, n2);

        int i = 0, j = 0, k = left;
        while (i < n1 && j < n2) {
            if (L[i].compareTo(R[j]) <= 0) {
                arr[k++] = L[i++];
            } else {
                arr[k++] = R[j++];
            }
        }
```

```
        while (i < n1) arr[k++] = L[i++];
        while (j < n2) arr[k++] = R[j++];
    }
}
```

SortingTrendVisualizer.java

```
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.CategoryAxis;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.CategoryPlot;
import org.jfree.chart.renderer.category.BarRenderer;
import org.jfree.data.category.DefaultCategoryDataset;

import javax.swing.*;
import java.awt.*;
import java.util.Arrays;

public class SortingTrendVisualizer extends JFrame {

    public SortingTrendVisualizer() {
        super("Sorting Performance Comparison");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new GridLayout(2, 2));   // 2x2 图表

        // Read four datasets
        String[] sorted1000 = CSVReader.readPlacesFromFile("src/data/1000places_sorted.csv");
        String[] random1000 = CSVReader.readPlacesFromFile("src/data/1000places_random.csv");
        String[] sorted10000 = CSVReader.readPlacesFromFile("src/data/10000places_sorted.csv");
        String[] random10000 = CSVReader.readPlacesFromFile("src/data/10000places_random.csv");

        add(createBarChartPanel(sorted1000, "1000 Sorted"));
        add(createBarChartPanel(random1000, "1000 Random"));
        add(createBarChartPanel(sorted10000, "10000 Sorted"));
        add(createBarChartPanel(random10000, "10000 Random"));

        setSize(1000, 700);
        setLocationRelativeTo(null);
    }

    private ChartPanel createBarChartPanel(String[] data, String title) {
        DefaultCategoryDataset dataset = new DefaultCategoryDataset();

        // Measurement of overall sorting time-consuming
        double tIns = measure(() -> Sorter.insertionSort(Arrays.copyOf(data, data.length)));
        double tQuick = measure(() -> Sorter.quickSort(Arrays.copyOf(data, data.length), 0, data.length - 1));
        double tMerge = measure(() -> Sorter.mergeSort(Arrays.copyOf(data, data.length), 0, data.length - 1));

        dataset.addValue(tIns, "Insertion Sort", "");
        dataset.addValue(tQuick, "Quick Sort", "");
        dataset.addValue(tMerge, "Merge Sort", "");

        JFreeChart chart = ChartFactory.createBarChart(
                title,
                "Sorting Algorithm",
                "Time (ms)",
                dataset
        );

        CategoryPlot plot = chart.getCategoryPlot();
        plot.setBackgroundPaint(Color.WHITE);
        plot.setRangeGridlinePaint(Color.LIGHT_GRAY);
```

```
        BarRenderer renderer = (BarRenderer) plot.getRenderer();
        renderer.setBarPainter(new org.jfree.chart.renderer.category.StandardBarPainter()); // 平面风格
        renderer.setMaximumBarWidth(0.2);
        renderer.setItemMargin(0.1);

        Color insertionColor = new Color(247, 216, 183); // 奶油米黄
        Color quickColor     = new Color(231, 193, 215); // 柔粉紫
        Color mergeColor     = new Color(175, 176, 208); // 雾霾蓝

        renderer.setSeriesPaint(0, insertionColor);
        renderer.setSeriesPaint(1, quickColor);
        renderer.setSeriesPaint(2, mergeColor);


        CategoryAxis domain = plot.getDomainAxis();
        domain.setTickLabelFont(new Font("SansSerif", Font.PLAIN, 11));
        domain.setLabelFont(new Font("SansSerif", Font.BOLD, 13));

        NumberAxis range = (NumberAxis) plot.getRangeAxis();
        range.setTickLabelFont(new Font("SansSerif", Font.PLAIN, 11));
        range.setLabelFont(new Font("SansSerif", Font.BOLD, 13));

        return new ChartPanel(chart);
    }

    private double measure(Runnable task) {
        long total = 0;
        int runs = 5;
        for (int i = 0; i < runs; i++) {
            long start = System.nanoTime();
            task.run();
            total += System.nanoTime() - start;
        }
        return total / (runs * 1_000_000.0);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            SortingTrendVisualizer frame = new SortingTrendVisualizer();
            frame.setVisible(true);
        });
    }
}
```

src/SortMemoryTest.java

```
import java.util.Arrays;

public class SortMemoryTest {
    public static void main(String[] args) {
        String[] sorted1000  = CSVReader.readPlacesFromFile("src/data/1000places_sorted.csv");
        String[] random1000  = CSVReader.readPlacesFromFile("src/data/1000places_random.csv");
        String[] sorted10000 = CSVReader.readPlacesFromFile("src/data/10000places_sorted.csv");
        String[] random10000 = CSVReader.readPlacesFromFile("src/data/10000places_random.csv");

        test("1000 Sorted",   sorted1000);
        test("1000 Random",   random1000);
        test("10000 Sorted", sorted10000);
        test("10000 Random", random10000);
    }

    private static void test(String label, String[] data) {
        System.out.println("=== " + label + " (n=" + data.length + ") ===");

        // Insertion
```

```
        Sorter.resetMemoryStats();
        String[] copy1 = Arrays.copyOf(data, data.length);
        Sorter.insertionSort(copy1);
        System.out.printf("Insertion Sort extra memory: %d KB%n",
                Sorter.getExtraMemoryBytes()/1024);

        // Quick
        Sorter.resetMemoryStats();
        String[] copy2 = Arrays.copyOf(data, data.length);
        Sorter.quickSort(copy2, 0, copy2.length - 1);
        System.out.printf("Quick Sort stack usage: %d KB (depth=%d)%n",
                Sorter.getQuickStackBytes()/1024,
                Sorter.getMaxRecursionDepth());

        // Merge
        Sorter.resetMemoryStats();
        String[] copy3 = Arrays.copyOf(data, data.length);
        Sorter.mergeSort(copy3, 0, copy3.length - 1);
        System.out.printf("Merge Sort extra memory: %d KB%n%n",
                Sorter.getExtraMemoryBytes()/1024);
    }
}
```

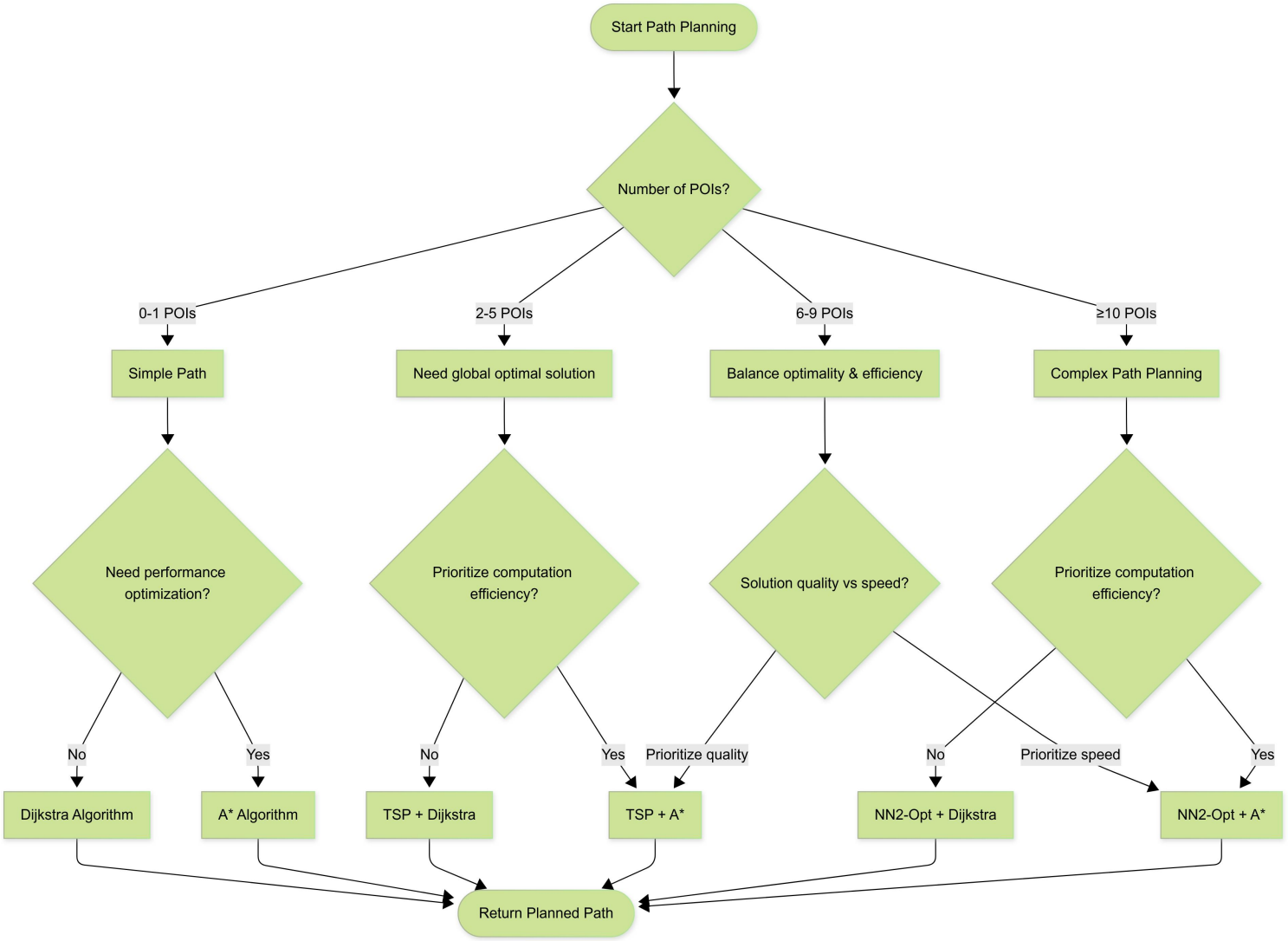
6. Chapter 6: Contribution

Student ID	Contribution(%)
2251576	50%
2255705	50%

7. Chapter 7: Appendix

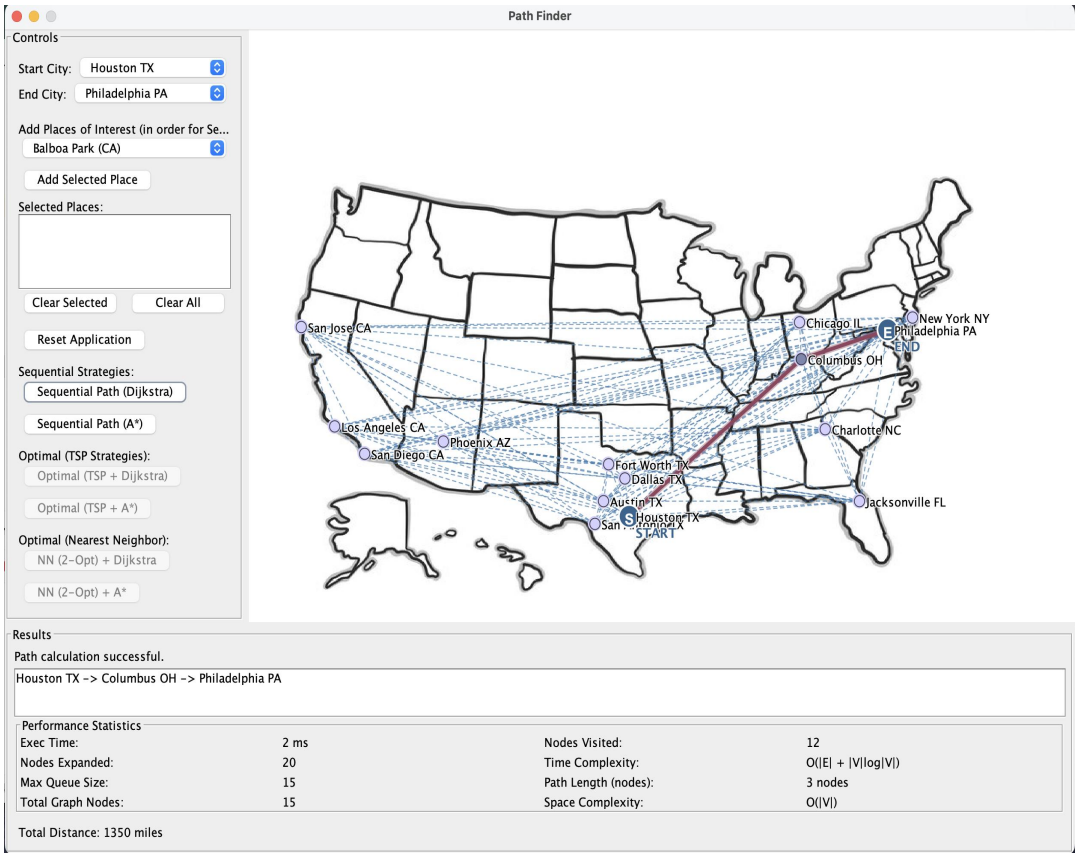
Appendix A: Output of Task A/B

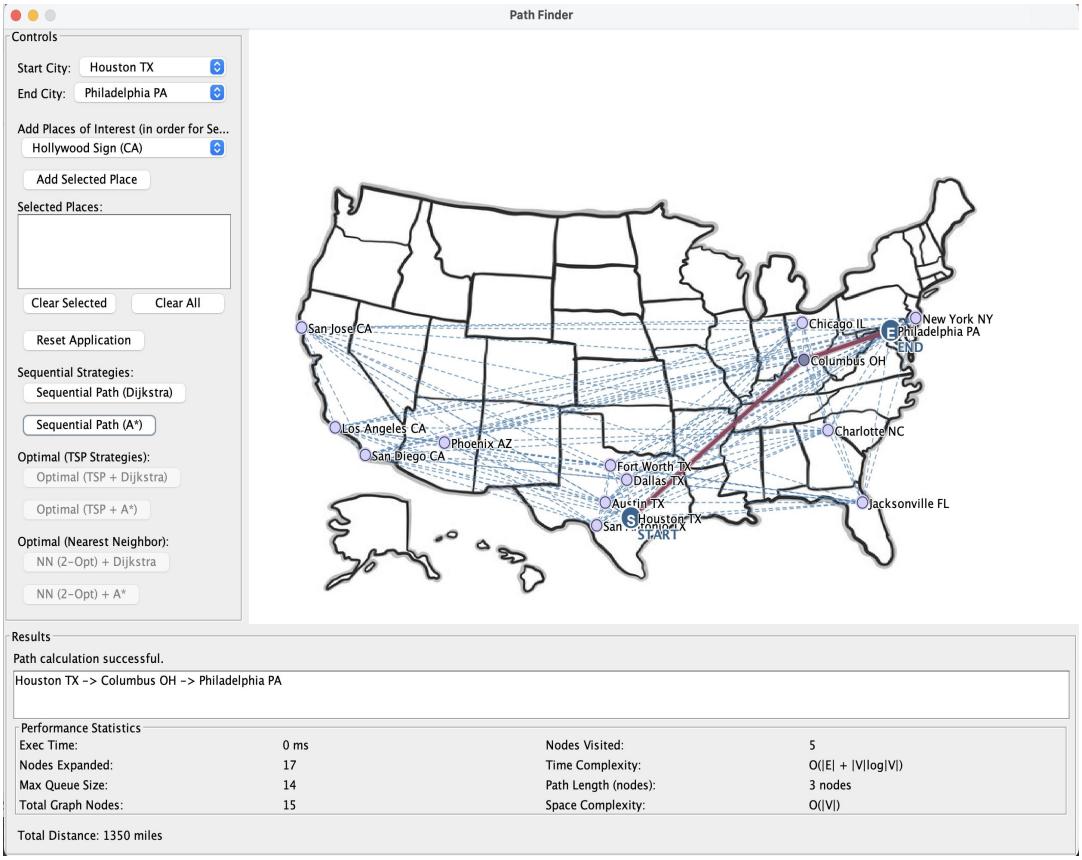
A.Final conclusion and Thinking



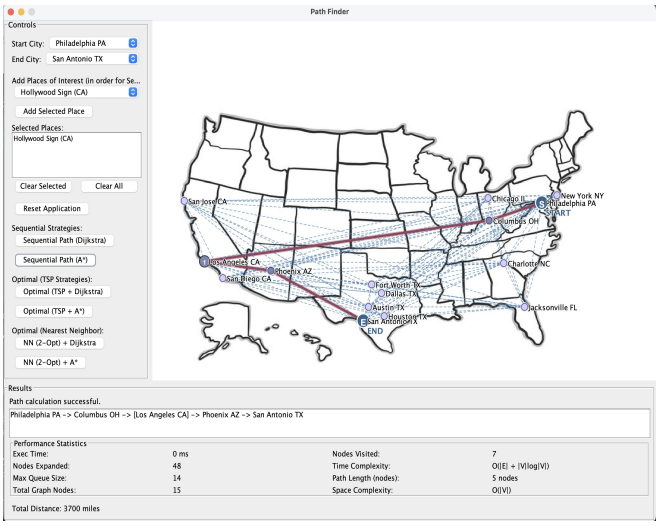
B.Three test cases：

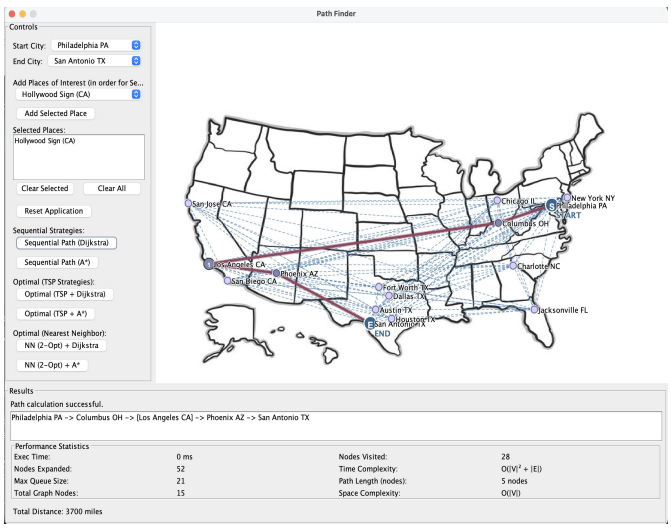
1. Case 1

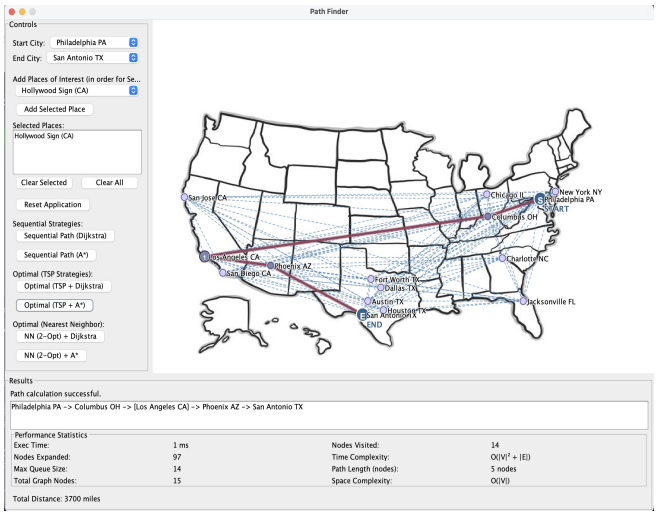


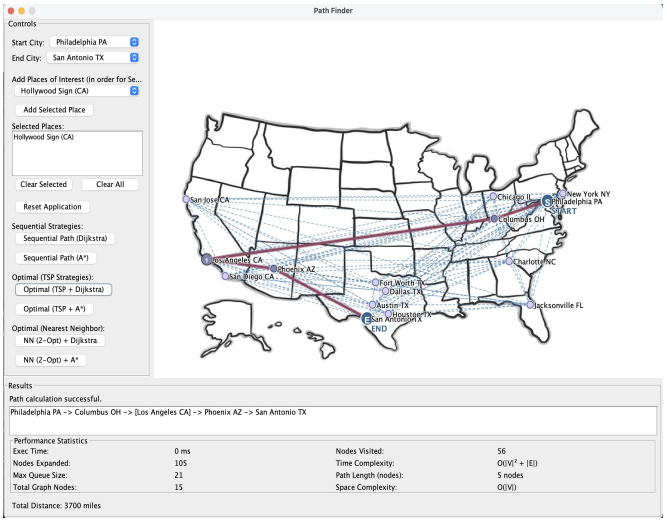


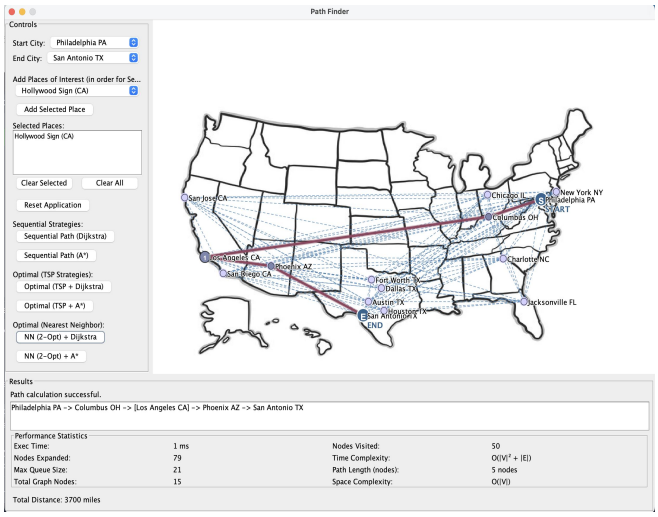
2. Case 2

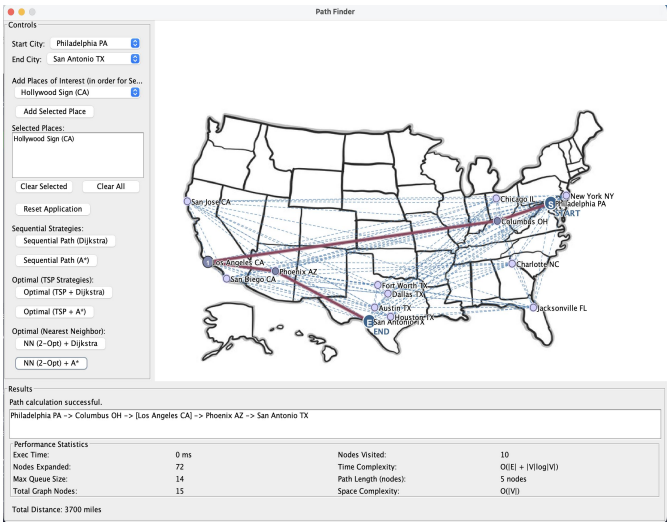

Path Finder
Controls
Start City: Philadelphia PA
End City: San Antonio TX
Add Places of Interest (in order for Se...
Hollywood Sign (CA)
Add Selected Place
Selected Places:
Hollywood Sign (CA)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
Philadelphia PA -> Columbus OH -> [Los Angeles CA] -> Phoenix AZ -> San Antonio TX
Performance Statistics
Exec Time: 0 ms
Nodes Visited: 7
Nodes Expanded: 48
Time Complexity: O(|E| + |V|log|V|)
Max Queue Size: 14
Path Length (nodes): 5 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 3700 miles
San Jose CA
Chicago IL
New York NY
Philadelphia PA
START
Columbus OH
Los Angeles CA
Phoenix AZ
San Diego CA
Charlotte NC
Fort Worth TX
Dallas TX
Austin TX
Houston TX
San Antonio TX
END
Jacksonville FL


Path Finder
Controls
Start City: Philadelphia PA
End City: San Antonio TX
Add Places of Interest (in order for Se...
Hollywood Sign (CA)
Add Selected Place
Selected Places:
Hollywood Sign (CA)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
Philadelphia PA -> Columbus OH -> [Los Angeles CA] -> Phoenix AZ -> San Antonio TX
Performance Statistics
Exec Time: 0 ms
Nodes Visited: 28
Nodes Expanded: 52
Time Complexity: O(|V|² + |E|)
Max Queue Size: 21
Path Length (nodes): 5 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 3700 miles
San Jose CA
Chicago IL
New York NY
Philadelphia PA
START
Columbus OH
Los Angeles CA
Phoenix AZ
San Diego CA
Charlotte NC
Fort Worth TX
Dallas TX
Austin TX
Houston TX
San Antonio TX
END
Jacksonville FL

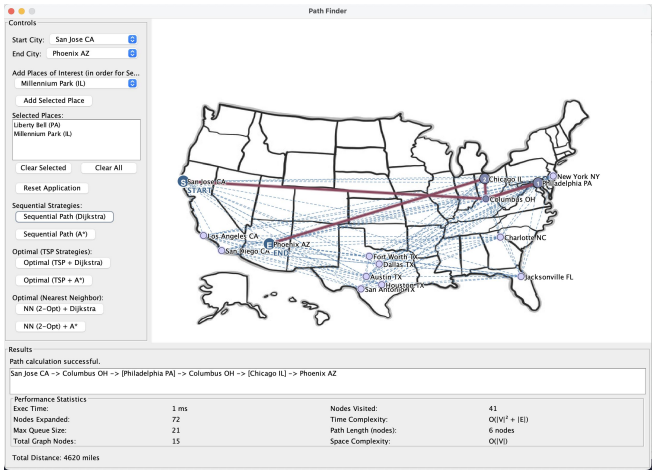

Path Finder
Controls
Start City: Philadelphia PA
End City: San Antonio TX
Add Places of Interest (in order for Se...
Hollywood Sign (CA)
Add Selected Place
Selected Places:
Hollywood Sign (CA)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
Philadelphia PA -> Columbus OH -> [Los Angeles CA] -> Phoenix AZ -> San Antonio TX
Performance Statistics
Exec Time: 1 ms
Nodes Visited: 14
Nodes Expanded: 97
Time Complexity: O(|V|² + |E|)
Max Queue Size: 14
Path Length (nodes): 5 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 3700 miles
San Jose CA
Chicago IL
New York NY
Philadelphia PA
START
Columbus OH
Los Angeles CA
Phoenix AZ
San Diego CA
Charlotte NC
Fort Worth TX
Dallas TX
Austin TX
Houston TX
San Antonio TX
END
Jacksonville FL

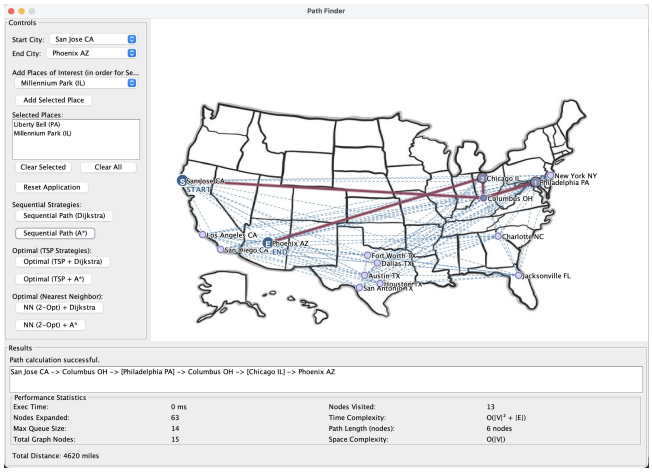

Path Finder
Controls
Start City: Philadelphia PA
End City: San Antonio TX
Add Places of Interest (in order for Se...
Hollywood Sign (CA)
Add Selected Place
Selected Places:
Hollywood Sign (CA)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
Philadelphia PA -> Columbus OH -> [Los Angeles CA] -> Phoenix AZ -> San Antonio TX
Performance Statistics
Exec Time: 0 ms
Nodes Visited: 56
Nodes Expanded: 105
Time Complexity: O(|V|² + |E|)
Max Queue Size: 21
Path Length (nodes): 5 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 3700 miles
San Jose CA
Chicago IL
New York NY
Philadelphia PA
START
Columbus OH
Los Angeles CA
Phoenix AZ
San Diego CA
Charlotte NC
Fort Worth TX
Dallas TX
Austin TX
Houston TX
San Antonio TX
END
Jacksonville FL

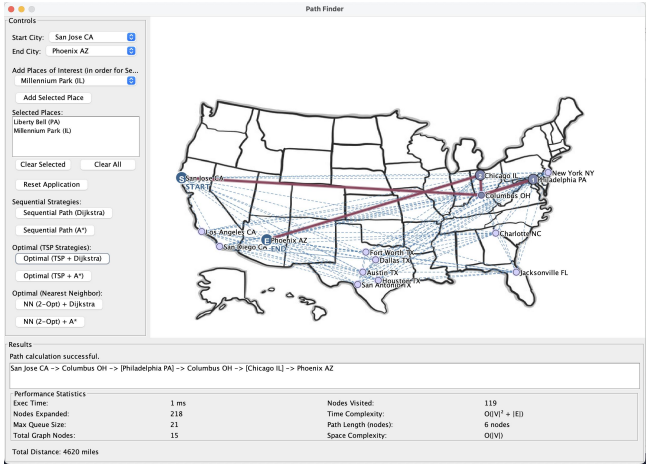

Path Finder
Controls
Start City: Philadelphia PA
End City: San Antonio TX
Add Places of Interest (in order for Se...
Hollywood Sign (CA)
Add Selected Place
Selected Places:
Hollywood Sign (CA)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
Philadelphia PA -> Columbus OH -> [Los Angeles CA] -> Phoenix AZ -> San Antonio TX
Performance Statistics
Exec Time: 1 ms
Nodes Visited: 50
Nodes Expanded: 79
Time Complexity: O(|V|² + |E|)
Max Queue Size: 21
Path Length (nodes): 5 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 3700 miles
San Jose CA
Chicago IL
New York NY
Philadelphia PA
START
Columbus OH
Los Angeles CA
Phoenix AZ
San Diego CA
Charlotte NC
Fort Worth TX
Dallas TX
Austin TX
Houston TX
San Antonio TX
END
Jacksonville FL

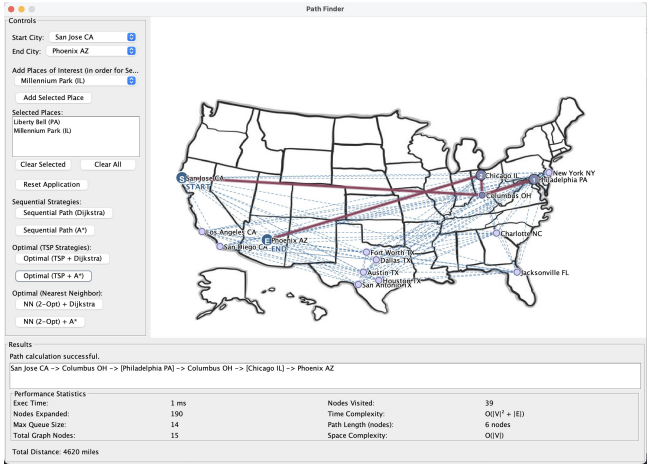

Path Finder
Controls
Start City: Philadelphia PA
End City: San Antonio TX
Add Places of Interest (in order for Se...
Hollywood Sign (CA)
Add Selected Place
Selected Places:
Hollywood Sign (CA)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
Philadelphia PA -> Columbus OH -> [Los Angeles CA] -> Phoenix AZ -> San Antonio TX
Performance Statistics
Exec Time: 0 ms
Nodes Visited: 10
Nodes Expanded: 72
Time Complexity: O(|E| + |V|log|V|)
Max Queue Size: 14
Path Length (nodes): 5 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 3700 miles
San Jose CA
Chicago IL
New York NY
Philadelphia PA
START
Columbus OH
Los Angeles CA
Phoenix AZ
San Diego CA
Charlotte NC
Fort Worth TX
Dallas TX
Austin TX
Houston TX
San Antonio TX
END
Jacksonville FL

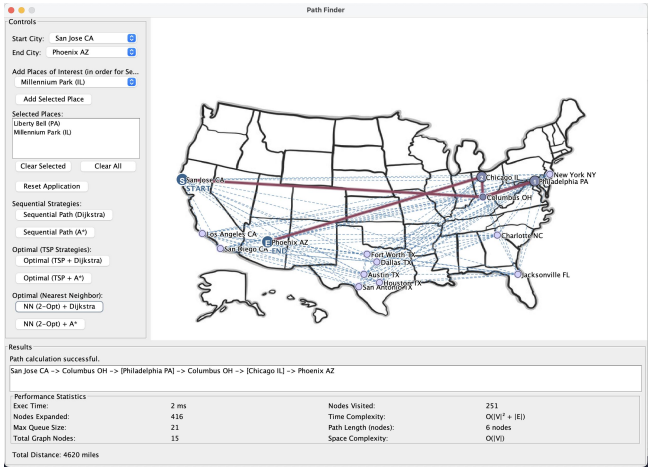
3. Case 3

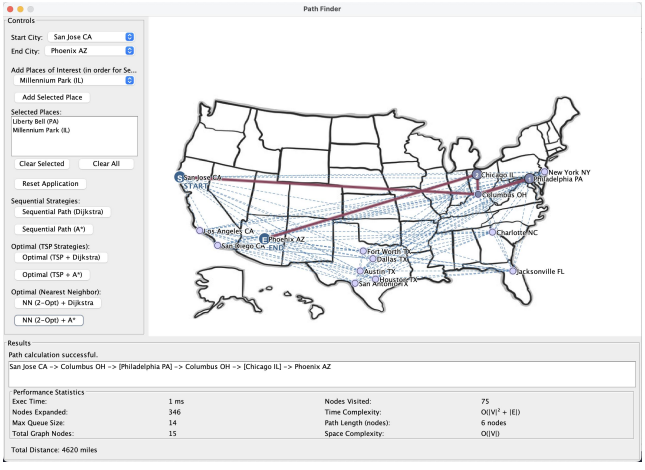

Path Finder
Controls
Start City: San Jose CA
End City: Phoenix AZ
Add Places of Interest (in order for Se...
Millennium Park (IL)
Add Selected Place
Selected Places:
Liberty Bell (PA)
Millennium Park (IL)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
San Jose CA -> Columbus OH -> [Philadelphia PA] -> Columbus OH -> [Chicago IL] -> Phoenix AZ
Performance Statistics
Exec Time: 1 ms
Nodes Visited: 41
Nodes Expanded: 72
Time Complexity: O(|V|² + |E|)
Max Queue Size: 21
Path Length (nodes): 6 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 4620 miles


Path Finder
Controls
Start City: San Jose CA
End City: Phoenix AZ
Add Places of Interest (in order for Se...
Millennium Park (IL)
Add Selected Place
Selected Places:
Liberty Bell (PA)
Millennium Park (IL)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
San Jose CA -> Columbus OH -> [Philadelphia PA] -> Columbus OH -> [Chicago IL] -> Phoenix AZ
Performance Statistics
Exec Time: 0 ms
Nodes Visited: 13
Nodes Expanded: 63
Time Complexity: O(|V|² + |E|)
Max Queue Size: 14
Path Length (nodes): 6 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 4620 miles


Path Finder
Controls
Start City: San Jose CA
End City: Phoenix AZ
Add Places of Interest (in order for Se...
Millennium Park (IL)
Add Selected Place
Selected Places:
Liberty Bell (PA)
Millennium Park (IL)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
San Jose CA -> Columbus OH -> [Philadelphia PA] -> Columbus OH -> [Chicago IL] -> Phoenix AZ
Performance Statistics
Exec Time: 1 ms
Nodes Visited: 119
Nodes Expanded: 218
Time Complexity: O(|V|² + |E|)
Max Queue Size: 21
Path Length (nodes): 6 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 4620 miles


Path Finder
Controls
Start City: San Jose CA
End City: Phoenix AZ
Add Places of Interest (in order for Se...
Millennium Park (IL)
Add Selected Place
Selected Places:
Liberty Bell (PA)
Millennium Park (IL)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
San Jose CA -> Columbus OH -> [Philadelphia PA] -> Columbus OH -> [Chicago IL] -> Phoenix AZ
Performance Statistics
Exec Time: 1 ms
Nodes Visited: 39
Nodes Expanded: 190
Time Complexity: O(|V|² + |E|)
Max Queue Size: 14
Path Length (nodes): 6 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 4620 miles


Path Finder
Controls
Start City: San Jose CA
End City: Phoenix AZ
Add Places of Interest (in order for Se...
Millennium Park (IL)
Add Selected Place
Selected Places:
Liberty Bell (PA)
Millennium Park (IL)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
San Jose CA -> Columbus OH -> [Philadelphia PA] -> Columbus OH -> [Chicago IL] -> Phoenix AZ
Performance Statistics
Exec Time: 2 ms
Nodes Visited: 251
Nodes Expanded: 416
Time Complexity: O(|V|² + |E|)
Max Queue Size: 21
Path Length (nodes): 6 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 4620 miles


Path Finder
Controls
Start City: San Jose CA
End City: Phoenix AZ
Add Places of Interest (in order for Se...
Millennium Park (IL)
Add Selected Place
Selected Places:
Liberty Bell (PA)
Millennium Park (IL)
Clear Selected
Clear All
Reset Application
Sequential Strategies:
Sequential Path (Dijkstra)
Sequential Path (A*)
Optimal (TSP Strategies):
Optimal (TSP + Dijkstra)
Optimal (TSP + A*)
Optimal (Nearest Neighbor):
NN (2-Opt) + Dijkstra
NN (2-Opt) + A*
Results
Path calculation successful.
San Jose CA -> Columbus OH -> [Philadelphia PA] -> Columbus OH -> [Chicago IL] -> Phoenix AZ
Performance Statistics
Exec Time: 1 ms
Nodes Visited: 75
Nodes Expanded: 346
Time Complexity: O(|V|² + |E|)
Max Queue Size: 14
Path Length (nodes): 6 nodes
Total Graph Nodes: 15
Space Complexity: O(|V|)
Total Distance: 4620 miles

Appendix B: Output of Task C

1. Output:

```
Processing file: 1000 Sorted
Data size: 1000
  Insertion Sort     : 0.2289 ms
  Quick Sort         : 0.2142 ms
  Merge Sort         : 0.1349 ms

Processing file: 1000 Random
Data size: 1000
  Insertion Sort     : 1.2794 ms
  Quick Sort         : 0.2290 ms
  Merge Sort         : 0.3690 ms

Processing file: 10000 Sorted
Data size: 10000
  Insertion Sort     : 0.0752 ms
  Quick Sort         : 0.7118 ms
  Merge Sort         : 1.3013 ms

Processing file: 10000 Random
Data size: 10000
  Insertion Sort     : 95.8226 ms
  Quick Sort         : 1.5353 ms
  Merge Sort         : 1.9183 ms


Process finished with exit code 0
```