Miles to Go: Smart Road Trip Planner

Group: 82

Name: Yilin.Li 2255705

Rui.Sang 2251576

Date：2025.5.18

Xi'an Jiaotong-Liverpool University
西交利物浦大学

COURSE PRESENTATION TEMPLATE

Part A
Design

Task A – System Design for Route Planning Application

Project Goal：

Build a program to plan optimized road trips in the USA

Users input:

A starting city and destination city

Zero or more attractions they want to visit

Find a route that:

Starts at the origin city

Visits all selected attractions

Ends at the destination

Minimizes the total driving distance

Core Components：

Graph construction using real map data

Shortest path algorithms (Dijkstra, A*)

Attraction ordering (TSP, Nearest Neighbor + 2-opt)

Object-oriented program design

Task A – System Architecture

Xi'an Jiaotong-Liverpool University
西交利物浦大学

The application follows a modular architecture:

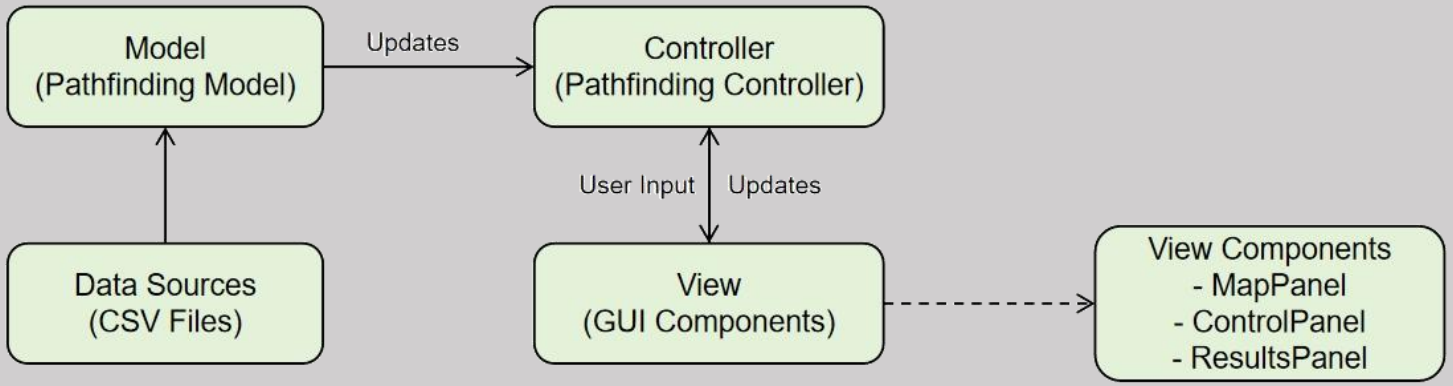


Figure.8 MVC Architecture of the interaction

Input Layer

Reads data from attractions.csv and roads.csv

Core Logic Layer:

Handles graph creation, route calculation, and attraction ordering

Output Layer

Displays route details and test results

Task A – Data Structure Design

attractions.csv

Stored as: *HashMap<String, String>*
→ (Attraction Name → City Name)

O(1) lookup speed
Simple and memory-efficient
No need to create complex objects

roads.csv

Stored as: *List<DataLoader.RoadConnection>*
RoadConnection includes: *cityA, cityB, distance*

Easy to load line-by-line and transfer to the graph

Graph Representation

Stored as: *HashMap<String, Map<String, Integer>>*
→ (City → Neighbor → Distance)

Efficient for sparse graphs (US road network)
Fast neighbor access: O(1)
Easy to scale and update

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
Edge List	O(E)	O(E)	O(E)	medium

Task A – Class Structure and Interactions

Model Layer

PathfindingModel: Facade for loading data and running pathfinding

Graph: Adjacency *list Map<String, Map<String, Integer>>;* supports fast neighbor access

PathPlanner: Manages algorithm calls; uses caching for TSP

PathResult: Stores result path, total distance, and stats (immutable)

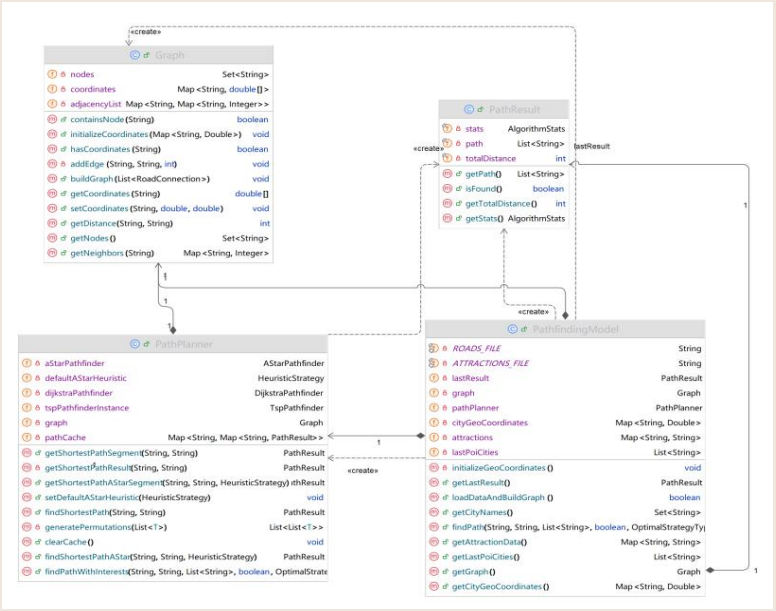


Figure 2: UML Class Diagram of the Model Components

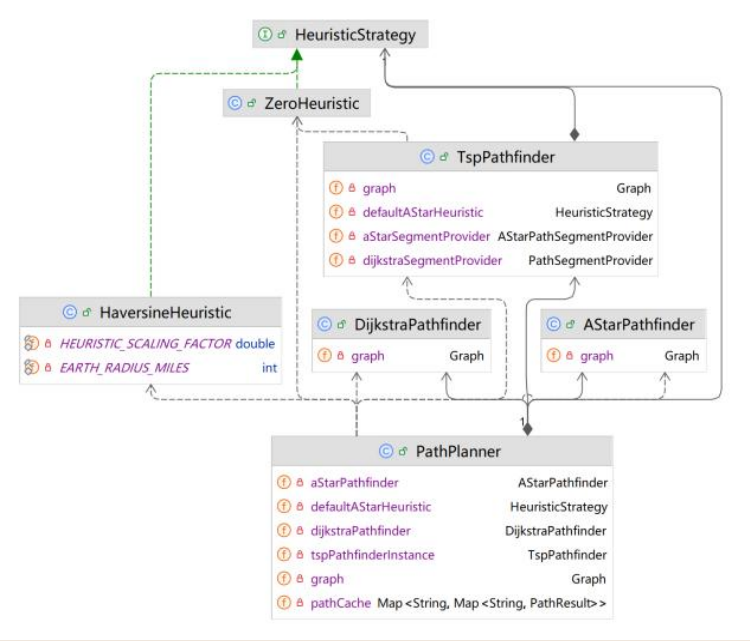


Figure 3: UML Class Diagram of the Algorithm Implementations

Algorithm Layer

DijkstraPathfinder / AStarPathfinder: Find shortest path

TspPathfinder:

Exact (permutations)

Approximate (NN + 2-opt)

HeuristicStrategy: Interface (e.g., Haversine, Zero)

Task A – Class Structure and Interactions

View Layer

MapPanel: Renders cities, roads, and paths

ControlPanel: Receives user input (city/POI selection)

ResultsPanel: Displays route + performance metrics

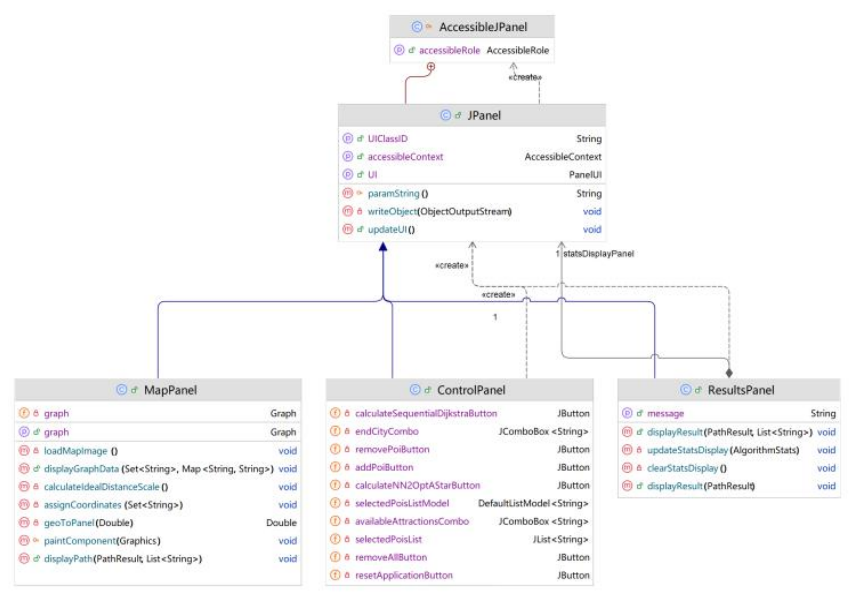
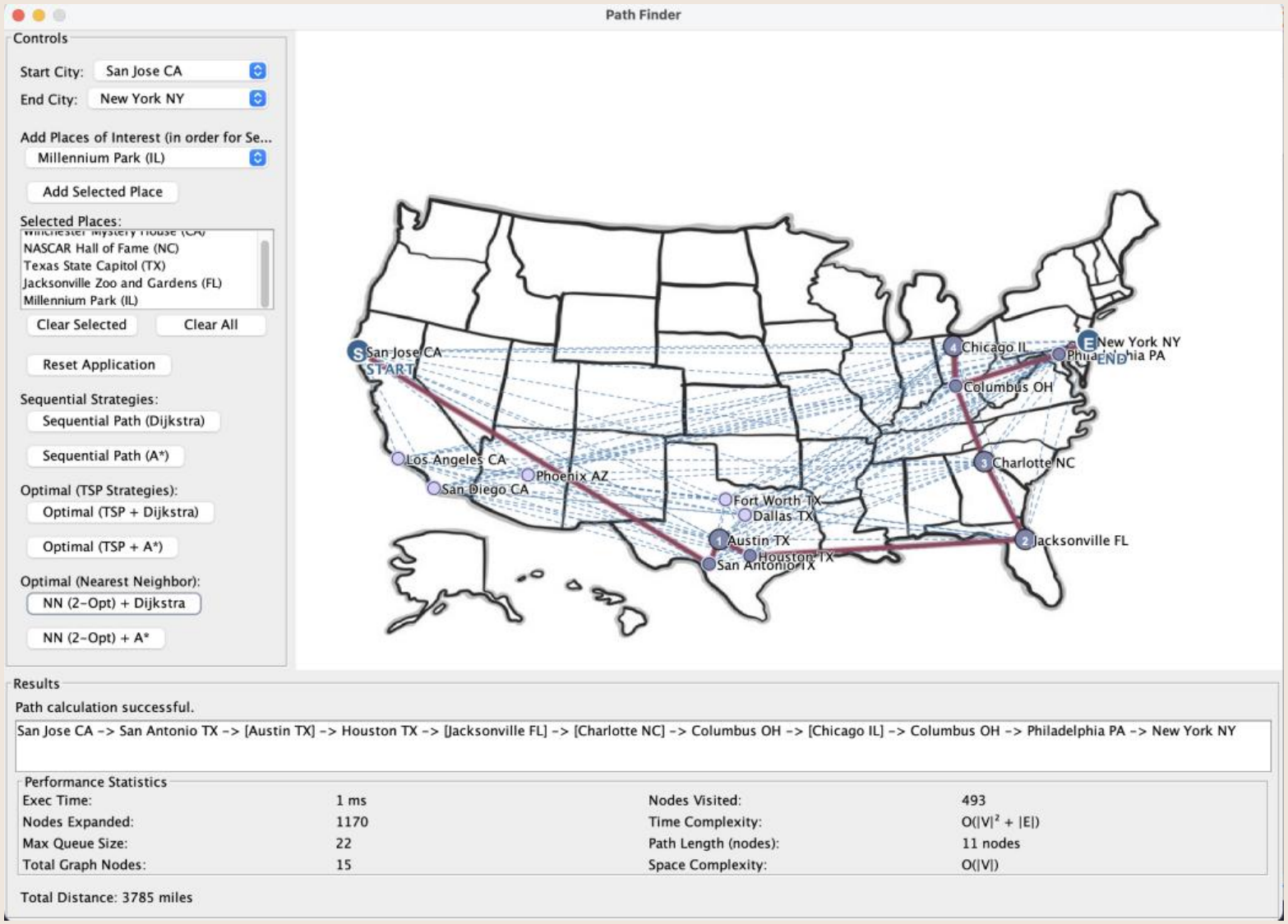


Figure 4: Class Diagram of the View Components



Task A – Application of Object-Oriented Programming (OOP) Principles

Encapsulation

Each algorithm has its own class (e.g., Dijkstra, A*)
Internal logic hidden; only public methods exposed
PathResult is immutable and safely shared

```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

Abstraction

Interfaces simplify algorithm use
PathResult hides internal logic, only shows results

```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

Inheritance

Heuristics share the HeuristicStrategy interface
UI panels extend JPanel to reuse rendering

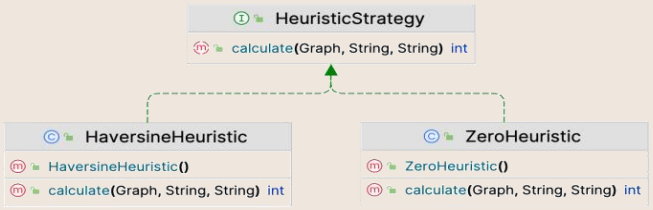


Figure 11 Inheritance hierarchy diagram for Heuristic Strategies

Polymorphism

Different algorithms/heuristics used with the same interface
Controller handles them without knowing implementation

Xi'an Jiaotong-Liverpool University
西交利物浦大学

COURSE PRESENTATION TEMPLATE

Part B
Evaluation

Task B – Algorithm Evaluation: Graph

Path Planning: Two Subproblems

Find the best order to visit all points of interest (POIs)

Find the shortest path between each pair of locations

1.Best Visiting Order

Optimal Visiting Order

TSP (Permutation-Based)

Tries all POI orders → guaranteed optimal

Time complexity: O(n!)

Suitable for ≤10 POIs

Nearest Neighbor + 2-opt

Greedy path + local optimization

Time complexity: $O(n^2)$

Scales better, good approximation

2. Shortest Path Between Two Cities

Dijkstra's Algorithm

Guarantees shortest path

Time: O((V + E) log V)

Uses priority queue

A Algorithm*

Adds heuristic (Haversine distance)

Same worst-case as Dijkstra

Faster in practice due to goal direction

Task B – Demonstration of Results

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

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

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

```
Houston TX -> Columbus OH -> Philadelphia PA
```

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

Task B – Demonstration of Results

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

Task B – Demonstration of Results

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

Task B – Demonstration of Results

Case 5 : 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:

```
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

Task B – Algorithm Complexity

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Time Complexity

Shortest Path Algorithms

A* typically visits fewer nodes than Dijkstra's algorithm for the same problem.

Dijkstra's:

O((V + E) log V)

Searches entire graph with priority queue

A*:

O((V + E) log V) (worst case)

Faster in practice using Haversine heuristic

Finding the Optimal Visiting Order

TSP algorithm:

O(n!) **only for a small number of POIs**

TSP algorithm:

O(n2) for the Nearest Neighbor and O(n2) for2-opt improvement

much more efficient for larger sets of POIs

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

Each distance computation in NN2-opt requires a shortest path calculation using either Dijkstra or A* than TSP.

Space Complexity

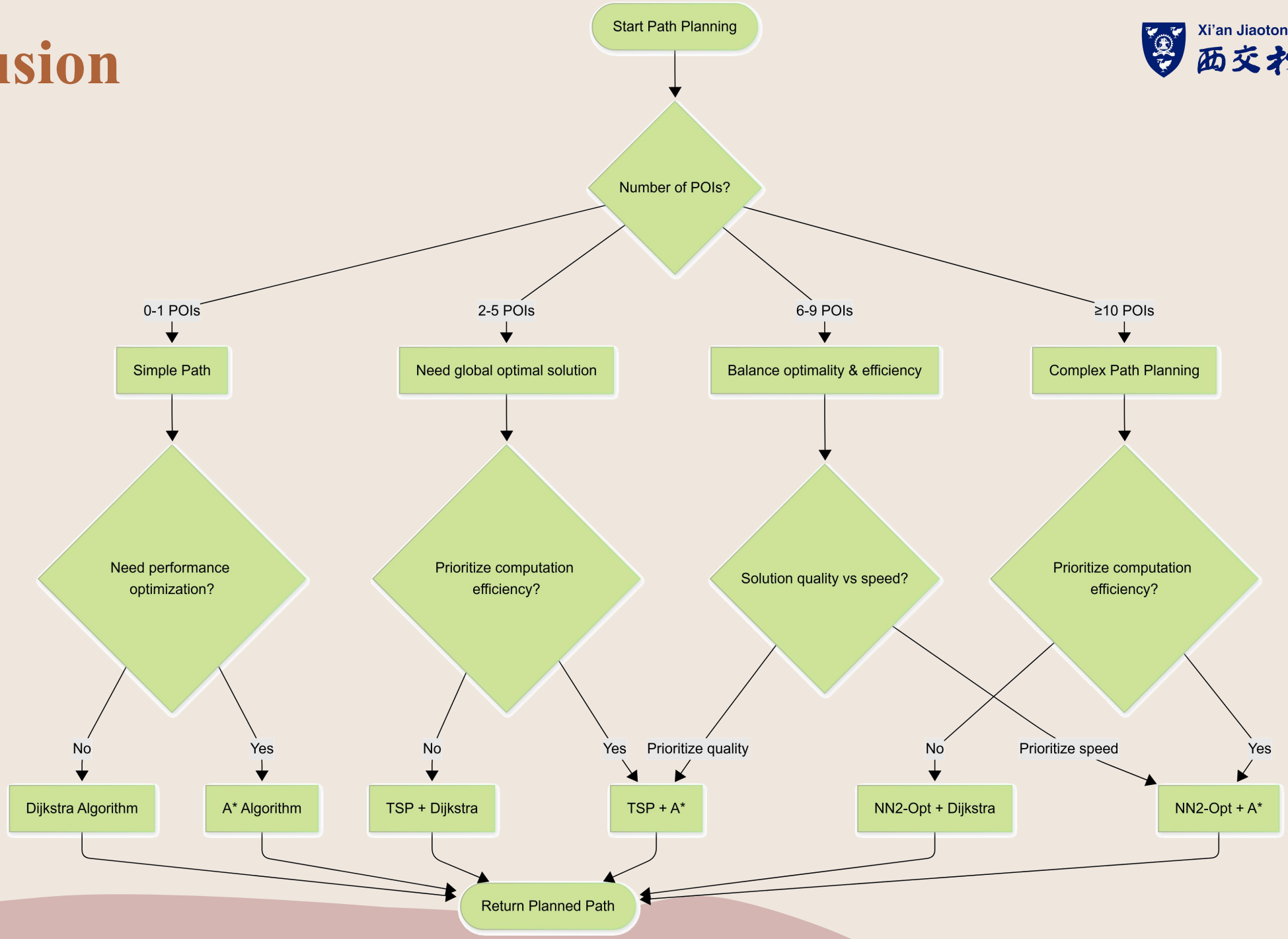
Graph Representation

Adjacency List: O(V + E)

Nodes set + Coordinates map: O(V)

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

Conclusion


Start Path Planning
Number of POIs?
0-1 POIs
2-5 POIs
6-9 POIs
≥10 POIs
Simple Path
Need global optimal solution
Balance optimality & efficiency
Complex Path Planning
Need performance optimization?
Prioritize computation efficiency?
Solution quality vs speed?
Prioritize computation efficiency?
No
Yes
No
Yes
Prioritize quality
No
Prioritize speed
Yes
Dijkstra Algorithm
A* Algorithm
TSP + Dijkstra
TSP + A*
NN2-Opt + Dijkstra
NN2-Opt + A*
Return Planned Path

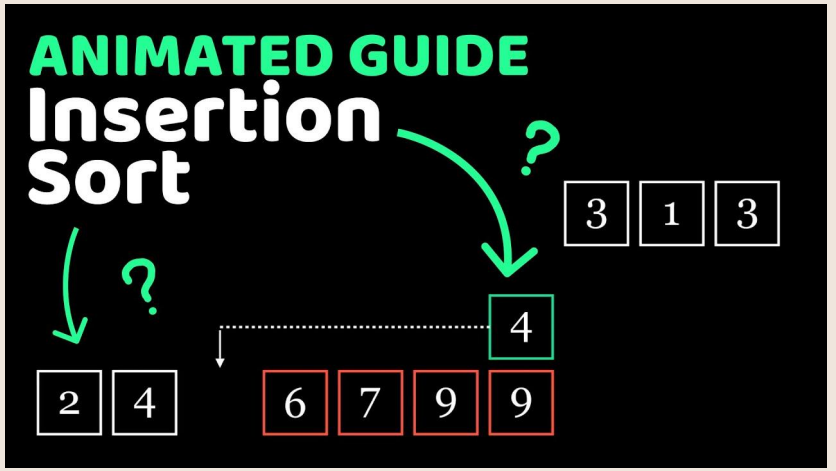
Xi'an Jiaotong-Liverpool University
西交利物浦大学

COURSE PRESENTATION TEMPLATE

Part C
Sorting

TaskC-Comparing 3 Sorting Method

Xi'an Jiaotong-Liverpool University
西交利物浦大学



Insertion Sort: This algorithm goes through the array from left to right. For each element, it places it into the correct position among the elements before it.
It works like sorting playing cards in your hand.
It is done in-place, and best for small or nearly sorted data.

Quick Sort : It picks one element as a **pivot** and splits the array into two parts: smaller and larger than the pivot. Then it sorts both parts using the same method.

Merge Sort: It keeps splitting the array into **halves** until each part has one item.
Then it starts merging those small parts back together in the correct order.
It always gives good performance, but it uses **extra memory.**

TaskC-Testing Result

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 1. Original execution time (ms) before pivot optimization

Strange!

Degradation!

$O(n \log n)$

Worse

$O(n^2)$

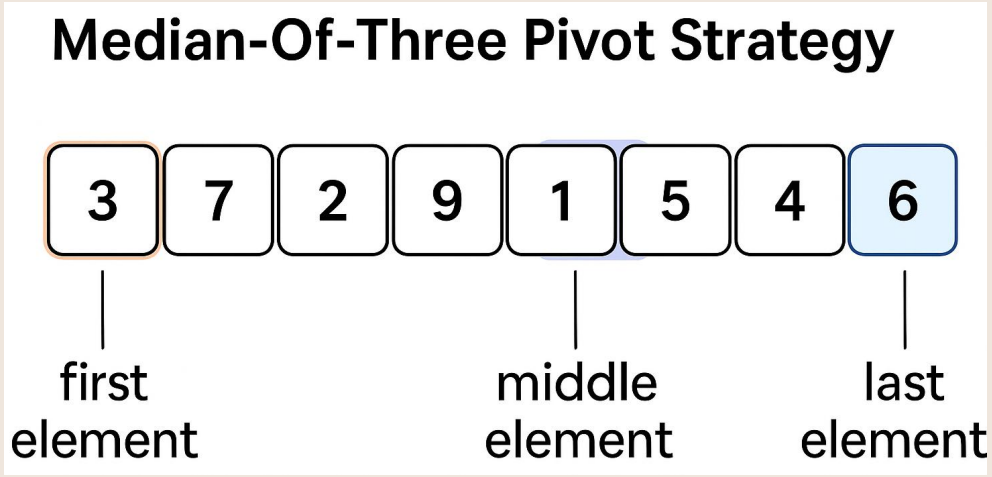
TaskC- Improvement in Quick Sort

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Median-Of-Three Pivot Strategy

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

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



TaskC-Impact of Input Order on Sorting Performance

Insertion Sort

Best case: Runs in O(n) when data is already sorted (e.g., 1000places_sorted).
Worst case: $O(n^2)$ for reversed or random input due to many shifts.
Highly sensitive to input order.
Efficient for partially sorted real-world data (e.g., logs, user activity).

Quick Sort

Generally stable in our tests after pivot optimization.
Classic Quick Sort can degrade to $O(n^2)$ on sorted input (if pivot is always last).
Solution: Median-of-three pivot strategy helps prevent this.

Merge Sort

Always splits and merges regardless of input order.
Runs consistently in O(n log n) for both sorted and random inputs.
Good predictability, but uses more memory.

TaskC-Impact of Input Size on Sorting Performance

Insertion Sort

Time Complexity: $O(n^2)$

Random Input:

1000 → 1.70 ms 10000 → 82.05 ms

Sorted Input:

1000 → 0.11 ms 10000 → 0.20 ms

Highly sensitive to input size when disordered

Quick Sort

Time Complexity: O(n log n)

Stable growth:

Random: 0.39 ms → 2.75 ms Sorted: 0.30 ms → 1.68 ms

Consistent performance, predictable scaling

Merge Sort

Time Complexity: Average O(n log n)

Random:0.25 ms → 1.93 ms Sorted:0.22 ms → 1.50 ms

Optimization: Used median-of-three pivot to prevent worst-case $O(n^2)$

TaskC-Sorting Performance with Duplicate Values

Test Setup :

Original datasets had no duplicate values
Created synthetic datasets with only three place names
Dataset sizes: 1000 and 10000 elements

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

Merge Sort is the best choice for datasets with many duplicates
Insertion Sort suffers from unnecessary shifting
Quick Sort needs improved handling for equal elements
Merge Sort’s consistency and stability make it ideal in such cases

TaskC-Memory Usage Comparison – Method & Results

Test Method：

Tracked **extra memory** for Merge Sort
Estimated stack usage for Quick Sort via **recursion depth**
Insertion Sort needs **no extra memory**

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： No extra memory **(O(1))**
Quick Sort： Improved by median-of-three pivot Worst-case (not shown): up to **624 KB stack**
Merge Sort： Always allocates extra arrays **(O(n))** **Not ideal** for devices with **limited memory**

Xi'an Jiaotong-Liverpool University
西交利物浦大学

COURSE PRESENTATION TEMPLATE

Part D
Conclusion

Task D – AI-Assisted Planning and Collaboration

Tools Used:

Used **Trello** for weekly task assignment, progress tracking, and code management
Though not AI-powered, features like checklists, timelines, reminders kept us on track

Team Feedback Loop:

Regularly commented and suggested changes on Trello cards
Example: Initial sorting chart was hard to read → revised based on team input
This iterative feedback helped us improve both design and collaboration

Task D – Equality, Diversity, and Inclusion (EDI)

Equality: Same content and layout for all users, regardless of region

Diversity: Aiming to support various languages, color-blind modes, senior-friendly displays

Inclusion: Simple interface so anyone—especially elderly or users with disabilities—can use core features

Challenges: Added development cost, design complexity, and the need for real-user testing

Task D – Reflection & Skills Gained

Improved in **algorithm design**, Java GUI, and performance testing

Learned to **break down problems**, work independently, and collaborate in a team setting

Gained experience in real-world **development workflows** and **inclusive design thinking**

THANKS for watching