

# Project2-MapNavigation System

## 1.实验目的

为了更好地理解图数据结构，例如最短路径算法和最小生成树算法。

## 2.实验代码思路及算法复杂度分析

**Graph类**：记录图的顶点，边等信息

```
//GraphNode
record GraphNode(int v, double weight) {
}

//Edge
record GraphEdge(int s, int d, double weight) {
}

public class Graph {
    protected static final int NODE_COUNT = 26;
    private final List<List<GraphNode>> adjacencyList = new ArrayList<>();
    private final double[][] adjacencyMatrix;

    public Graph() {
        for (int i = 0; i < NODE_COUNT; i++) {
            adjacencyList.addLast(new ArrayList<>()); //初始化邻接链表
        }

        adjacencyMatrix = new double[NODE_COUNT][NODE_COUNT];
        for (int i = 0; i < NODE_COUNT; i++) {
            Arrays.fill(adjacencyMatrix[i], Double.MAX_VALUE);
            adjacencyMatrix[i][i] = 0; // Distance to self is 0
        }
    }

    //添加边
    public void addEdge(int start, int end, double weight) {
        adjacencyList.get(start).add(new GraphNode(end, weight));
        adjacencyList.get(end).add(new GraphNode(start, weight));
        adjacencyMatrix[start][end] = weight;
        adjacencyMatrix[end][start] = weight;
    }
}
```

**记录路径信息函数collectPaths**： `collectPaths` 方法通过前驱节点信息递归地构建最短路径，在最坏情况下，可能需要遍历图中的大部分节点来构建完整路径，其时间复杂度取决于图的结构和最短路径的分布情况，其时间复杂度为  $O(V)$ ，因为最多也就是遍历所有节点一次来构建路径。

```
//路径信息
```

```

private void collectPaths(ArrayList<ArrayList<Integer>> record, int temp,
int next, int x, Deque<Integer> stack, double weight, List<String> paths) {
    //向前寻找路径信息
    while (record.get(temp).getFirst() != x) {
        if (temp < 0 || temp >= NODE_COUNT) {
            return;
        }
        int m = record.get(temp).size();
        if (m > 1) { //如果在同一级中有多个节点，就复制相关信息进入递归
            for (int i = 1; i < m; i++) {
                int v = record.get(temp).get(i);
                Deque<Integer> newStack = new LinkedList<>(stack);
                newStack.push(v);
                collectPaths(record, v, v, x, newStack, weight +
adjacencyMatrix[v][next], paths);
            }
        }
        int v = record.get(temp).getFirst();
        if (v < 0 || v >= NODE_COUNT) {
            break;
        }
        stack.push(v);
        temp = v;
        weight += adjacencyMatrix[temp][next];
        next = temp;
    }
    weight += adjacencyMatrix[x][next];

    StringBuilder path = new StringBuilder();
    path.append((char) (x + 'A'));
    while (!stack.isEmpty()) {
        int index = stack.pop();
        path.append("->").append((char) ('A' + index));
    }
    path.append("\n最短距离为 ").append(String.format("%.2f",
weight)).append(" km.\n");
    paths.add(path.toString());
}

```

## (1) Given two locations, show the shortest path from one to the other on the map and its length.

功能一：采用Dijkstra与Bellman-Ford算法。

操作代码如下：

```

button1.setOnAction(event -> {
    outputArea.clear();
    outputArea.appendText("Operation 1:\n");
    String start = location1.getText();
    String end = location2.getText();
    if (start.length() == 1 && end.length() == 1 &&
        (start.charAt(0) >= 'A' && start.charAt(0) <= 'Z') &&
        (end.charAt(0) >= 'A' && end.charAt(0) <= 'Z')) {
        outputArea.appendText("Following is Dijkstra Alogorithm.\n");
        // 使用Dijkstra算法求两点间最短路径
    }
}

```

```

        List<String> resultD =
graph.findShortestPathDijkstra(start.charAt(0), end.charAt(0));
        StringBuilder s = new StringBuilder();
        for (int i = 0; i < resultD.size() - 1; i++) {
            s.append(resultD.get(i));
        }
        s.append(resultD.get(resultD.size() - 1));
        outputArea.appendText("从 " + start + " 到 " + end + " 路线: " +
s.toString() + "\n");
        outputArea.appendText("\nFollowing is Bellman-Ford
Algorithm.\n");
        // 使用Bellman-Ford算法求两点间最短路径（作为另一种解法示例，可按需选用或
扩展更多算法对比）
        List<String> resultBF =
graph.findShortestPathBellmanFord(start.charAt(0), end.charAt(0));
        StringBuilder s2 = new StringBuilder();
        for (int i = 0; i < resultD.size() - 1; i++) {
            s2.append(resultBF.get(i));
        }
        s2.append(resultBF.get(resultBF.size() - 1));
        outputArea.appendText("从 " + start + " 到 " + end + " 路线: " +
s2.toString() + "\n");
    } else {
        outputArea.appendText("输入地点错误，请重新输入。 \n");
    }
}
});

```

## Dijkstra算法:

```

//Dijkstra算法 单源最短路径
public List<String> findShortestPathDijkstra(char s, char d) {
    int start = s - 'A', end = d - 'A';
    double[] distance = new double[NODE_COUNT]; //距离
    ArrayList<ArrayList<Integer>> lists = new ArrayList<>(); //记录前一个节点
    for (int i = 0; i < NODE_COUNT; i++) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(-1); // 初始化
        lists.add(list);
    }
    boolean[] visited = new boolean[NODE_COUNT]; //记录是否已经访问过了

    Arrays.fill(distance, Integer.MAX_VALUE);
    distance[start] = 0;

    PriorityQueue<Integer> heap = new PriorityQueue<>
(Comparator.comparingDouble(a -> distance[a])); //最小堆
    heap.add(start);

    while (!heap.isEmpty()) {
        int index = heap.poll();
        if (visited[index]) continue;
        visited[index] = true;
        int l = adjacencyList.get(index).size();
        for (int i = 0; i < l; i++) {
            int ver = adjacencyList.get(index).get(i).v(); //提取相邻的节
点信息
            double weight = adjacencyList.get(index).get(i).weight();

```

```

        if (visited[ver]) continue;

        if (distance[index] + weight < distance[ver]) {
            distance[ver] = distance[index] + weight;
            lists.remove(ver);
            ArrayList<Integer> al = new ArrayList<>();
            al.add(index);
            lists.add(ver, al);
            heap.add(ver);
        } else if (!visited[ver] && distance[index] + weight ==
distance[ver]) {
            if (lists.get(ver).getFirst() == -1) {
                lists.get(ver).removeFirst();
                lists.get(ver).add(index);
            } else {
                lists.get(ver).add(index);
            }
        }
    }
}

Deque<Integer> stack = new LinkedList<>();
stack.add(end);
List<String> paths = new ArrayList<>();
collectPaths(lists, end, end, start, stack, 0, paths);
return paths;
}

```

### 时间复杂度分析:

#### 1. 初始化:

- 初始化 `lists` 用于记录前驱节点, 时间复杂度为 $O(V)$ 。
- 初始化 `visited`, `d` 数组, 时间复杂度为 $O(V)$ 。

#### 2. 优先队列的初始化:

- 使用最小堆 (即 `PriorityQueue`) 来存储未访问的节点, 初始时间复杂度为 $O(\log V)$ 。

#### 3. 主循环和更新:

- 外层循环执行, 直到 `heap` 为空。最坏情况下, 如果图是稀疏的, Dijkstra 将会遍历所有的节点, 即 $V$ 次。

在每次迭代中, 主要操作包括:

- 从 `heap` 中取出一个节点, 时间复杂度为 $O(\log V)$ 。
- 更新相邻节点的距离:
  - 对每个相邻节点, 检查是否被访问, 如果没有, 则更新距离和前驱节点, 这个相邻节点的处理可能会遍历所有与该节点相邻的边。

假设  $E$  是图中的边数, 对于每个节点, 最坏情况下, 可能会遍历所有边, 时间复杂度在这一部分是  $O(E \log V)$ 。

综上所述:

稀疏图( $E \approx V$ ): 时间复杂度为  $O(V \log V + E \log V) = O(V \log V)$ 。

稠密图( $E \approx V^2$ ): 时间复杂度为  $O(E \log V) = O((V^2) \log V)$ 。

总结: 时间复杂度主要由堆操作和邻接链表遍历决定, 表示为  $O(E \log V)$ 。

## Bellman-Ford算法:

```
//Bellman-Ford算法
public List<String> findShortestPathBellmanFord(char s, char d) {
    int x = s - 'A', y = d - 'A';

    double[] distance = new double[NODE_COUNT]; //初始化距离数组
    Arrays.fill(distance, Integer.MAX_VALUE);
    distance[x] = 0;

    ArrayList<ArrayList<Integer>> record = new ArrayList<>(); //记录前一个节点
    是哪一个
    for (int i = 0; i < NODE_COUNT; i++) {
        ArrayList<Integer> a1 = new ArrayList<>();
        a1.add(-1);
        record.add(a1);
    }

    for (int i = 0; i < NODE_COUNT - 1; i++) { //循环次数为顶点数减1
        boolean[] visited = new boolean[NODE_COUNT]; //记录每次循环中节点是否被
        访问过
        for (int j = 0; j < NODE_COUNT; j++) { //遍历所有边
            int m = adjacencyList.get(j).size();
            for (int k = 0; k < m; k++) {
                int ver = adjacencyList.get(j).get(k).v();
                double weight = adjacencyList.get(j).get(k).weight();

                if (distance[j] != Integer.MAX_VALUE && distance[j] + weight
                    < distance[ver]) { //成功松弛
                    distance[ver] = distance[j] + weight;
                    record.remove(ver);
                    ArrayList<Integer> a1 = new ArrayList<>();
                    a1.add(j);
                    record.add(ver, a1);
                    visited[ver] = true;
                } else if (distance[j] != Integer.MAX_VALUE && distance[j] +
                    weight == distance[ver]) {
                    if (!visited[ver]) {
                        record.remove(ver);
                        ArrayList<Integer> a1 = new ArrayList<>();
                        a1.add(j);
                        record.add(ver, a1);
                    } else {
                        record.get(ver).add(j);
                    }
                    visited[ver] = true;
                }
            }
        }
    }

    Deque<Integer> stack = new LinkedList<>();
    stack.add(y);
    List<String> paths = new ArrayList<>();
    collectPaths(record, y, y, x, stack, 0, paths);
    return paths;
}
```

## 时间复杂度分析：

### 1. 初始化操作：

- 初始化距离数组 `distance`，需要遍历包含 `NODE_COUNT` 个元素的数组并赋值，时间复杂度为  $O(V)$ （这里  $|V|$  表示图中节点数量，等同于 `NODE_COUNT`）。
- 初始化记录前驱节点的列表 `record`，同样需要遍历 `NODE_COUNT` 次来创建内部的每个 `ArrayList` 并初始化，其时间复杂度也是  $O(V)$ 。
- 所以初始化这部分总的复杂度为  $O(V)$ 。

### 2. 主循环部分（Bellman - Ford 算法的核心迭代过程）：

- 主循环会执行 `NODE_COUNT - 1` 次，也就是与图中节点数量  $|V|$  相关的固定次数，其时间复杂度可表示为  $O(V)$ 。
- 在每一轮循环中：
  - 首先会初始化一个长度为 `NODE_COUNT` 的布尔数组 `visited` 来记录节点访问情况，这一步的时间复杂度为  $O(V)$ 。
  - 接着需要遍历图中的所有边，遍历方式是先遍历所有节点（通过 `for (int j = 0; j < NODE_COUNT; j++)`），对于每个节点再遍历其邻接边（通过 `int m = adjacencyList.get(j).size(); for (int k = 0; k < m; k++)`）。假设图中边的数量为  $|E|$ ，所有节点邻接边数量之和就是  $|E|$ ，所以遍历所有边的操作时间复杂度为  $O(E)$ 。
  - 在遍历每条边时，进行距离松弛判断以及相应的前驱节点记录更新操作，这些操作本身的时间复杂度是常数级别，可忽略不计，主要时间消耗还是在遍历边的过程中。
- 综合来看，主循环部分每一轮的时间复杂度是  $O(E)$ ，由于一共执行  $|V| - 1$  轮，所以这部分整体的时间复杂度为  $O(V * E)$ 。

综合上述各部分的时间复杂度分析，`findShortestPathBellmanFord` 方法总的复杂度为  $O(V * E)$ ，其中  $|V|$  表示图中节点的数量， $|E|$  表示图中边的数量。需要注意的是，Bellman - Ford 算法在处理边数相对较少的稀疏图时，时间复杂度相较于一些优化后的算法（如 Dijkstra 算法在稀疏图下的时间复杂度表现）可能相对较高。

## (2) Given one location, show the shortest paths from all locations on the map to this one and their length.

功能二：采用Dijkstra与Bellman-Ford算法。

操作代码如下：

```
button2.setOnAction(event -> {
    outputArea.clear();
    outputArea.appendText("Operation 2:\n");
    String target = location1.getText();
    if (target.length() == 1 && (target.charAt(0) >= 'A' &&
    target.charAt(0) <= 'Z')) {
        outputArea.appendText("到 " + target + " 的最短路径:\n");
        outputArea.appendText("Following is Dijkstra Alogorithm.\n");
        for (int i = 0; i < Graph.NODE_COUNT; i++) {
            List<String> resultAllD =
graph.findShortestPathDijkstra((char) ('A' + i), target.charAt(0));
            StringBuilder s = new StringBuilder();
            for (int j = 0; j < resultAllD.size() - 1; j++) {
                s.append(resultAllD.get(i));
            }
            s.append(resultAllD.get(resultAllD.size() - 1));
```

```

        outputArea.appendText("从 " + (char) ('A' + i) + " 到 " +
target.charAt(0) + " 路线: " + s.toString() + "\n");
    }
    outputArea.appendText("\nFollowing is Bellman-Ford
Algorithm.\n");
    for (int i = 0; i < Graph.NODE_COUNT; i++) {
        List<String> resultAllBF =
graph.findShortestPathBellmanFord((char) ('A' + i), target.charAt(0));
        StringBuilder s2 = new StringBuilder();
        for (int j = 0; j < resultAllBF.size() - 1; j++) {
            s2.append(resultAllBF.get(j));
        }
        s2.append(resultAllBF.get(resultAllBF.size() - 1));
        outputArea.appendText("从 " + (char) ('A' + i) + " 到 " +
target.charAt(0) + " 路线: " + s2.toString() + "\n");
    }
    } else {
        outputArea.appendText("输入地点错误, 请重新输入.\n");
    }
}
});

```

#### 时间复杂度分析:

分别使用 Dijkstra 算法和 Bellman - Ford 算法来计算从每个节点 (总共  $V$  个节点) 到目标节点的最短路径, 并将结果展示在 outputArea 中。对于每种算法的结果处理, 是先获取从每个节点出发到目标节点的最短路径列表 (通过调用对应的算法方法), 然后将列表中的字符串元素拼接起来, 最终把从各节点到目标节点的完整路线信息添加到 outputArea 中显示。

因此对应 Dijkstra 时间复杂度即为  $O(V \cdot E \cdot \log V)$ , Bellman-Ford 时间复杂度即为  $O(E \cdot V^2)$ 。

### (3) You need to provide the path and distance for designing the subway routes based on the current road , which needs to meet these conditions:

a. All locations are included in this route.

b. The selected route has the shortest distance among all possible routes.

功能三: 使用 Kruskal 与 Prim 算法。

操作代码如下:

```

button3.setOnAction(event -> {
    outputArea.clear();
    outputArea.appendText("Operation 3:\n");
    // 设计地铁路线
    outputArea.appendText("Following is Kruskal Algorithm.\n");
    // Kruskal 算法
    List<String> resultK = graph.Kruskal_subway();
    StringBuilder s = new StringBuilder();
    for (int i = 0; i < resultK.size() - 1; i++) {
        s.append(resultK.get(i));
    }
    s.append(resultK.get(resultK.size() - 1));
    outputArea.appendText(s.toString() + "\n");
}

```

```

        outputArea.appendText("Following is Prim Alogorithm.\n");
        // Prim算法
        List<String> resultP = graph.Prim_subway();
        StringBuilder s2 = new StringBuilder();
        for (int i = 0; i < resultP.size() - 1; i++) {
            s2.append(resultP.get(i));
        }
        s2.append(resultP.get(resultP.size() - 1));
        outputArea.appendText(s2.toString() + "\n");
    });
}

```

## Kruskal算法:

```

public List<String> Kruskal_subway() {
    int[] father = new int[NODE_COUNT]; // 并查集初始化
    for (int i = 0; i < NODE_COUNT; i++) father[i] = i;
    ArrayList<GraphEdge> edges = new ArrayList<>();
    for (int i = 0; i < NODE_COUNT; i++) {
        for (int j = i + 1; j < NODE_COUNT; j++) {
            if (adjacencyMatrix[i][j] != Double.MAX_VALUE)
                edges.add(new GraphEdge(i, j, adjacencyMatrix[i][j]));
        }
    }
    // 按边权重从小到大排序
    edges.sort(Comparator.comparingDouble(GraphEdge::weight));
    List<String> result = new ArrayList<>();
    double totalWeight = 0;
    // 选择边构建最小生成树
    for (GraphEdge edge : edges) {
        if (find(father, edge.s()) != find(father, edge.d())) {
            union(father, edge.s(), edge.d());
            result.add("(" + (char) (edge.s() + 'A') + " , " + (char)
(edge.d() + 'A') + " , " + String.format("%.2f", edge.weight()) + " km)\n");
            totalWeight += edge.weight();
        }
    }
    // 检查是否所有节点都被连接
    int root = find(father, 0);
    for (int i = 1; i < NODE_COUNT; i++) {
        if (find(father, i) != root) {
            return List.of("无法构建连通图, 无法满足条件!");
        }
    }
    result.add("最短路径总长度为: " + String.format("%.2f", totalWeight) + "
km.\n");
    return result;
}

```

## 时间复杂度分析:

1. **初始化并查集**: 时间复杂度为  $O(V)$ 。
2. **收集边信息**: 时间复杂度为  $O(V^2)$ , 是双层循环遍历邻接矩阵的结果, 这部分操作相对比较耗时, 尤其在节点数量较多的图中。
3. **边排序**: 时间复杂度为  $O(E \cdot \log E)$ , 排序操作的复杂度取决于边的数量以及排序算法本身的特性, 通常是影响整体复杂度的重要部分, 尤其是边数较多时。



4. **构建最小生成树（遍历边并利用并查集操作）**：时间复杂度近似为  $O(E)$ ，主要时间消耗在对每条边进行判断和合并操作上，虽然单次 `find` 和 `union` 操作可能有优化空间，但整体循环次数与边数相关。
5. **检查连通性**：时间复杂度为  $O(V)$ ，相对来说这部分的时间复杂度量级在整体中通常不是主导因素。

综合各部分的时间复杂度情况，在 Kruskal 算法中，边排序操作的时间复杂度  $O(E \cdot \log E)$  与 收集边信息的  $O(V^2)$  通常是主导因素。

所以，`Kruskal_subway` 方法总的时间复杂度为  $O(V^2 + E \cdot \log E)$ 。

## Prim算法

```
public List<String> Prim_subway() {
    boolean[] visited = new boolean[NODE_COUNT];
    List<String> result = new ArrayList<>();
    PriorityQueue<GraphEdge> heap = new PriorityQueue<>
(Comparator.comparingDouble(GraphEdge::weight));
    for (GraphNode node : adjacencyList.getFirst()) {
        heap.add(new GraphEdge(0, node.v(), node.weight()));
    }
    visited[0] = true;

    double weight = 0;
    while (!heap.isEmpty()) {
        GraphEdge edge = heap.poll();
        int v = edge.d();
        if (visited[v]) continue;
        weight += edge.weight();
        result.add("( " + (char) (edge.s() + 'A') + " , " + (char) (edge.d()
+ 'A') + " , " + String.format("%.2f", edge.weight()) + " km)\n");
        visited[v] = true;
        for (GraphNode node : adjacencyList.get(v)) {
            heap.add(new GraphEdge(v, node.v(), node.weight()));
        }
    }
    result.add("最短距离为 " + String.format("%.2f", weight) + " km.\n");
    return result;
}
```

### 时间复杂度分析：

1. **初始化操作**：初始化 `visited` 数组时间复杂度为  $O(V)$ ，创建其他数据结构（列表和优先队列）时间复杂度可看作常数级别，这部分整体时间复杂度主要由 `visited` 数组初始化决定，为  $O(V)$ 。
2. **初始边加入优先队列**：遍历起始节点的邻接边并加入优先队列，时间复杂度为  $O(E)$ ，此部分在整个算法执行初期执行一次，是整体复杂度的一部分。
3. **主循环部分**：
  - 主循环最多执行  $|E|$  次，每次循环中从堆中取边的操作时间复杂度为  $O(\log E)$ ，其他如判断节点访问状态、更新权重、添加结果以及标记节点访问等操作时间复杂度都是常数级别，可忽略不计，主要的时间消耗在于取边操作以及每次可能添加邻接边到堆中的操作。所以主循环部分整体时间复杂度为  $O(E \cdot \log E)$ 。

综合来看，`Prim_subway` 方法总的时间复杂度为  $O(V + E \log E)$ ，即为  $O(E \cdot \log E)$ 。

(4) Given one location, such as A, you need to provide the path and distance for designing bus routes starting from A, and ensure that the bus route is the shortest. In addition:

a. The shortest path from all other points to A through this bus route remains the same length as before.

b. Any two points can be reached through the designed bus route.

具体要求是在最短路径组合成的图中，若有多种可能，取其总路径长度最短。

操作代码如下：

```
button4.setOnAction(event -> {
    outputArea.clear();
    outputArea.appendText("Operation 4:\n");
    String start = location1.getText();
    if (start.length() == 1 && (start.charAt(0) >= 'A' &&
start.charAt(0) <= 'Z')) {
        // 求从指定点出发的公交路线
        List<String> busRoutes = graph.bus_Dijkstra(start.charAt(0));
        StringBuilder s = new StringBuilder();
        for (int i = 0; i < busRoutes.size() - 1; i++) {
            s.append(busRoutes.get(i));
        }
        s.append(busRoutes.get(busRoutes.size() - 1));
        outputArea.appendText(s.toString() + "\n");
    } else {
        outputArea.appendText("输入地点错误，请重新输入。 \n");
    }
});
```

## bus算法

```
public List<String> bus_Dijkstra(char s) {
    int x = s - 'A'; // 起点

    double[] distance = new double[NODE_COUNT]; // 起点到其他各点的距离
    Arrays.fill(distance, Double.MAX_VALUE);
    distance[x] = 0;

    ArrayList<ArrayList<Integer>> record = new ArrayList<>(); // 记录前一个节
点
    for (int i = 0; i < NODE_COUNT; i++) {
        ArrayList<Integer> a1 = new ArrayList<>();
        a1.add(-1);
        record.add(a1);
    }

    boolean[] visited = new boolean[NODE_COUNT]; // 是否访问过
    PriorityQueue<Integer> heap = new PriorityQueue<>
(Comparator.comparingDouble(a -> distance[a])); // 最小堆
    heap.add(x);

    // 使用 Dijkstra 算法计算从起点到所有点的最短路径
```

```

while (!heap.isEmpty()) {
    int index = heap.poll();
    if (visited[index]) continue;
    visited[index] = true;
    for (GraphNode neighbor : adjacencyList.get(index)) {
        int ver = neighbor.v();
        double weight = neighbor.weight();

        if (visited[ver]) continue;

        if (distance[index] + weight < distance[ver]) { // 松弛操作
            distance[ver] = distance[index] + weight;
            record.set(ver, new ArrayList<>(List.of(index)));
            heap.add(ver);
        } else if (distance[index] + weight == distance[ver]) { // 记录等
            record.get(ver).add(index);
        }
    }
}

// 构建公交线路图
Graph busGraph = new Graph();
for (int y = 0; y < NODE_COUNT; y++) {
    if (y == x) continue;
    Deque<Integer> stack = new LinkedList<>();
    stack.add(y);
    bus_Recursive(record, y, y, x, stack, busGraph);
}

// 使用 Prim 算法生成最小生成树
return busGraph.Prim_subway();
}

private void bus_Recursive(ArrayList<ArrayList<Integer>> record, int temp,
int next, int x, Deque<Integer> stack, Graph g) {
    // 向前追踪路径信息
    while (record.get(temp).get(0) != x) {
        int m = record.get(temp).size();
        if (m > 1) { // 如果同一级有多个节点，递归处理
            for (int i = 1; i < m; i++) {
                int v = record.get(temp).get(i);
                Deque<Integer> newStack = new LinkedList<>(stack);
                newStack.push(v);
                bus_Recursive(record, v, v, x, newStack, g);
            }
        }
        int v = record.get(temp).get(0);
        stack.push(v);
        temp = v;
        next = temp;
    }

    // 添加边到公交线路图
    g.addEdge(x, stack.peek(), adjacencyMatrix[x][stack.peek()]);
    while (!stack.isEmpty()) {
        int index = stack.pop();

```

```
        if (!stack.isEmpty()) g.addEdge(index, stack.peek(),
adjacencyMatrix[index][stack.peek()]);
    }
}
```

时间复杂度分析：

bus\_Dijkstra 方法整体流程回顾

### (1) 初始化部分

- 首先进行了一系列的数据结构初始化操作：
  - 初始化距离数组 `distance`，将其所有元素初始化为 `Double.MAX_VALUE`，并将起点对应的距离设为 0，这一操作遍历 `NODE_COUNT`（设图中节点数量为  $|V|$ ）个元素，时间复杂度为  $O(V)$ 。
  - 初始化记录前驱节点的 `record` 列表，其包含  $|V|$  个 `ArrayList`，每个内部列表初始化为包含一个 `-1` 元素，整体初始化时间复杂度同样为  $O(V)$ 。
  - 初始化用于标记节点是否访问过的布尔数组 `visited`，时间复杂度也是  $O(V)$ 。
  - 创建并初始化一个最小堆 `heap`，向其中添加起始节点，添加操作时间复杂度为  $O(1)$ ，所以初始化这部分总的时间复杂度为  $O(V)$ 。

### (2) Dijkstra 算法核心循环部分

- 通过while循环执行 Dijkstra 算法，循环终止条件是堆heap为空，在最坏情况下会执行V次（即遍历所有节点）。
  - 在每次循环中：
    - 从堆中取出节点（`heap.poll()`）的操作时间复杂度为 $O(\log V)$ ，因为最小堆的取出操作时间复杂度基于元素个数的对数级别。
    - 遍历当前取出节点的邻接节点列表（通过 `adjacencyList.get(index)` 获取邻接节点，假设平均每个节点的邻接边数量为  $|E| / |V|$ ，这里  $|E|$  表示图中边的数量），这一步的时间复杂度近似看作为  $O(E)$ （在整体图规模考量下）。
    - 对于每个邻接节点，进行距离松弛判断以及更新前驱节点列表和向堆中添加节点等操作，这些操作的时间复杂度都是常数级别，主要时间消耗在于遍历邻接节点和堆操作上。
  - 所以这部分整体的时间复杂度为 $O(V \cdot \log V + E)$ 。

### (3) 构建公交路线图部分

- 通过 `for` 循环遍历所有节点（除起点外），对于每个节点调用 `bus_Recursive` 方法来构建公交路线图，循环执行  $|V| - 1$  次。每次调用 `bus_Recursive` 方法会根据前驱节点信息构建路径并添加边到新的图 `busGraph` 中，其时间复杂度与图结构和路径情况相关，暂记为  $T$ ，所以这部分总的时间复杂度为  $O(V \cdot T)$ 。

### (4) 生成最小生成树部分

- 调用 `busGraph.Prim_subway` 方法，根据之前对 `Prim_subway` 方法的时间复杂度分析可知，其时间复杂度为  $O(V + E \cdot \log E)$ ，这里  $|V|$  和  $|E|$  是构建的公交路线图 `busGraph` 中的节点数量和边数量。

## 2. bus\_Recursive 方法时间复杂度分析

- 该方法用于根据前驱节点信息递归地构建公交路线图的边信息，主要操作如下：

- 通过 `while` 循环向前追踪路径信息，循环条件基于前驱节点信息判断，在最坏情况下可能需要遍历较多的节点，但最多也就是遍历到起点，所以循环次数与图中节点数量  $|V|$  相关，可看作最多执行  $|V|$  次。
- 在循环中：
  - 首先判断同一级是否有多个前驱节点，如果有则进行递归调用，递归调用的次数取决于同一级的前驱节点数量（最多可能与  $|V|$  相关，但通常会小于  $|V|$ ），每次递归调用又会执行同样的逻辑，整体递归的时间复杂度较难精确分析，但大致上与图的结构以及路径的分支情况相关，其复杂度不会超过  $O(V^2)$ （极端情况下考虑所有节点都存在多个前驱且层层递归），通常会远小于这个量级。
  - 其他操作如将节点入栈、更新临时节点索引等操作时间复杂度都是常数级别，可忽略不计。
- 循环结束后，添加边到图 `g` 的操作时间复杂度也是常数级别（每次添加两条边，忽略具体添加边的底层复杂度）。
- 综合来看，`bus_Recursive` 方法的时间复杂度较难精确给出一个简洁表达式，但大致可以认为其时间复杂度在  $O(V)$  到  $O(V^2)$  之间，取决于图的具体结构和路径分支情况，为了后续整体分析方便，暂记为  $T$ （前面已提及）。

### `bus_Dijkstra` 方法总体时间复杂度

综合上述各部分分析，`bus_Dijkstra` 方法总的时间复杂度为各部分时间复杂度之和，化简为  $O(E \log E + V * P)$ ， $P$  为路径分支数。

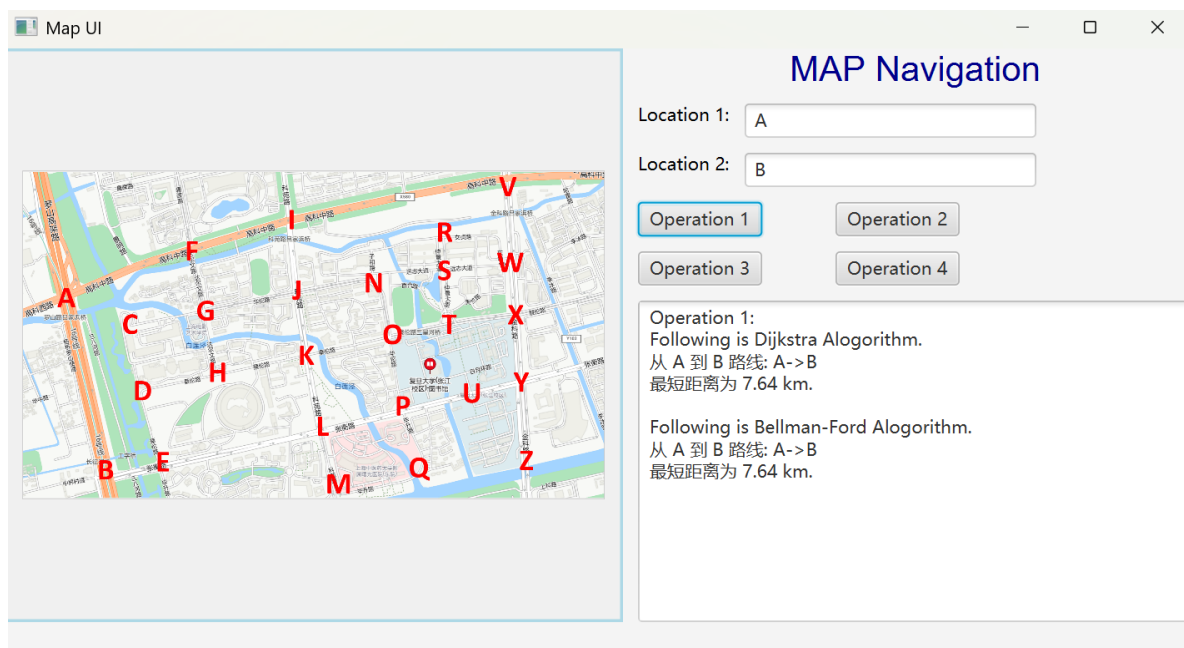
## 3.Map系统操作指南

operation1-4分别对应四个功能。

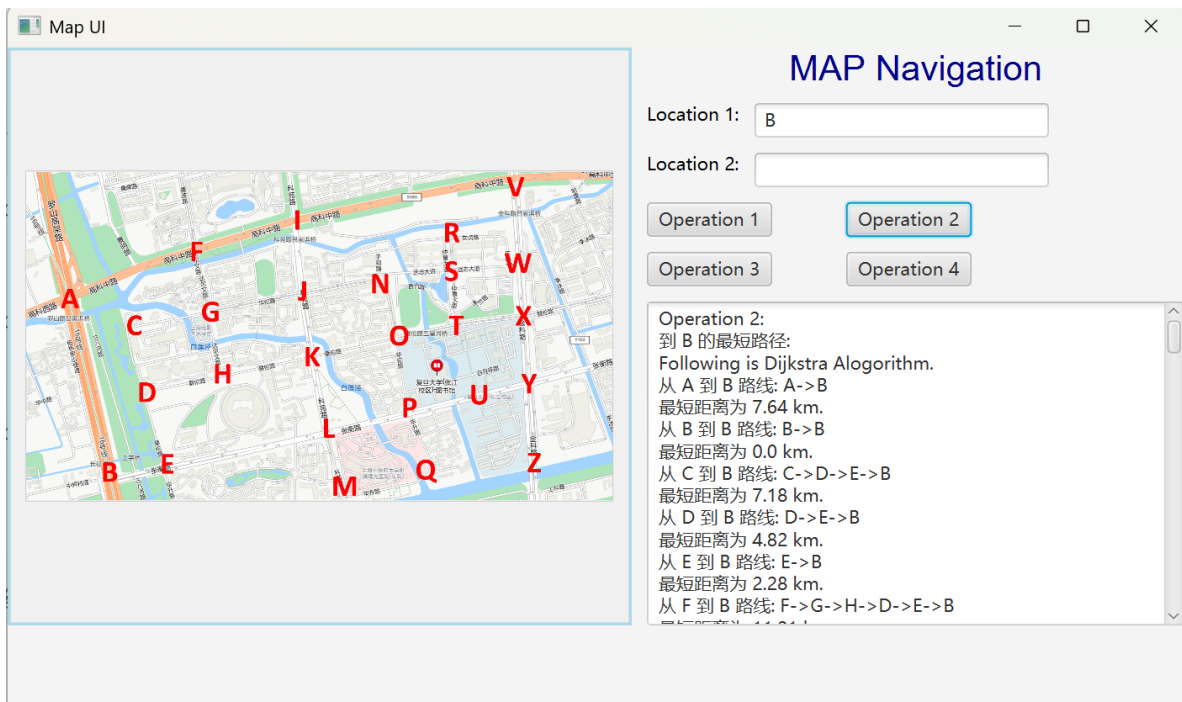
- operation1：需要location 1与location 2，再点击按钮operation1，下方文本框即可输出答案。
- operation2：仅需要输入location 1，再点击按钮operation2，下方文本框即可输出答案。
- operation3：点击按钮operation3，下方文本框即可输出答案。
- operation4：仅需要输入location 1，再点击按钮operation4，下方文本框即可输出答案。

操作结果实例：

operation1：



operation2:(文本框可以上下滑动)



文本框完整内容：

#### Operation 2:

到 B 的最短路径:

Following is Dijkstra Alogorithm.

从 A 到 B 路线: A->B

最短距离为 7.64 km.

从 B 到 B 路线: B->B

最短距离为 0.00 km.

从 C 到 B 路线: C->D->E->B

最短距离为 7.18 km.

从 D 到 B 路线: D->E->B

最短距离为 4.82 km.

从 E 到 B 路线: E->B

最短距离为 2.28 km.

从 F 到 B 路线: F->G->H->D->E->B

最短距离为 11.21 km.

从 G 到 B 路线: G->H->D->E->B

最短距离为 9.71 km.

从 H 到 B 路线: H->D->E->B

最短距离为 7.70 km.

从 I 到 B 路线: I->F->G->H->D->E->B

最短距离为 15.60 km.

从 J 到 B 路线: J->G->H->D->E->B

最短距离为 13.65 km.

从 K 到 B 路线: K->H->D->E->B

最短距离为 11.64 km.

从 L 到 B 路线: L->E->B

最短距离为 10.45 km.

从 M 到 B 路线: M->L->E->B

最短距离为 12.38 km.

从 N 到 B 路线: N->O->K->H->D->E->B

最短距离为 16.86 km.

从 O 到 B 路线: O->K->H->D->E->B

最短距离为 15.46 km.

从 P 到 B 路线: P->L->E->B

最短距离为 14.14 km.

从 Q 到 B 路线: Q->M->L->E->B

最短距离为 15.46 km.

从 R 到 B 路线: R->S->T->O->K->H->D->E->B

最短距离为 19.52 km.

从 S 到 B 路线: S->T->O->K->H->D->E->B

最短距离为 18.68 km.

从 T 到 B 路线: T->O->K->H->D->E->B

最短距离为 17.25 km.

从 U 到 B 路线: U->P->L->E->B

最短距离为 16.61 km.

从 V 到 B 路线: V->W->S->T->O->K->H->D->E->B

最短距离为 23.90 km.

从 W 到 B 路线: W->S->T->O->K->H->D->E->B

最短距离为 21.14 km.

从 X 到 B 路线: X->T->O->K->H->D->E->B

最短距离为 20.02 km.

从 Y 到 B 路线: Y->U->P->L->E->B

最短距离为 18.28 km.

从 Z 到 B 路线: Z->Y->U->P->L->E->B

最短距离为 20.80 km.

Following is Bellman-Ford Alogorithm.

从 A 到 B 路线: A->B

最短距离为 7.64 km.

从 B 到 B 路线: B->B

最短距离为 0.00 km.

从 C 到 B 路线: C->D->E->B

最短距离为 7.18 km.

从 D 到 B 路线: D->E->B

最短距离为 4.82 km.

从 E 到 B 路线: E->B

最短距离为 2.28 km.

从 F 到 B 路线: F->G->H->D->E->B

最短距离为 11.21 km.

从 G 到 B 路线: G->H->D->E->B

最短距离为 9.71 km.

从 H 到 B 路线: H->D->E->B

最短距离为 7.70 km.

从 I 到 B 路线: I->F->G->H->D->E->B

最短距离为 15.60 km.

从 J 到 B 路线: J->G->H->D->E->B

最短距离为 13.65 km.

从 K 到 B 路线: K->H->D->E->B

最短距离为 11.64 km.

从 L 到 B 路线: L->E->B

最短距离为 10.45 km.

从 M 到 B 路线: M->L->E->B

最短距离为 12.38 km.

从 N 到 B 路线: N->O->K->H->D->E->B

最短距离为 16.86 km.

从 O 到 B 路线: O->K->H->D->E->B

最短距离为 15.46 km.

从 P 到 B 路线: P->L->E->B

最短距离为 14.14 km.

从 Q 到 B 路线: Q->M->L->E->B

最短距离为 15.46 km.

从 R 到 B 路线: R->S->T->O->K->H->D->E->B

最短距离为 19.52 km.

从 S 到 B 路线: S->T->O->K->H->D->E->B

最短距离为 18.68 km.

从 T 到 B 路线: T->O->K->H->D->E->B

最短距离为 17.25 km.

从 U 到 B 路线: U->P->L->E->B

最短距离为 16.61 km.

从 V 到 B 路线: V->W->S->T->O->K->H->D->E->B

最短距离为 23.90 km.

从 W 到 B 路线: W->S->T->O->K->H->D->E->B

最短距离为 21.14 km.



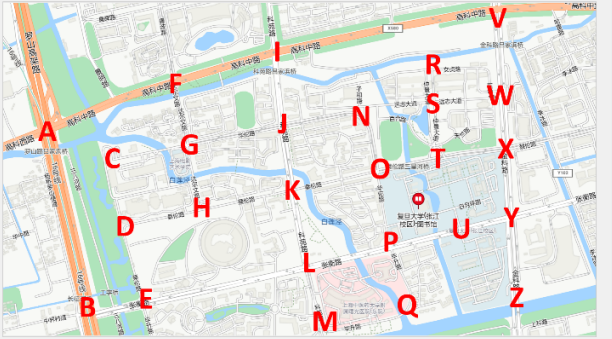
从 X 到 B 路线：X->T->O->K->H->D->E->B  
最短距离为 20.02 km.

从 Y 到 B 路线：Y->U->P->L->E->B  
最短距离为 18.28 km.

从 Z 到 B 路线：Z->Y->U->P->L->E->B  
最短距离为 20.80 km.

operation3:(文本框可以上下滑动)

Map UI



MAP Navigation

Location 1:

Location 2:

Operation 1

Operation 2

Operation 3

Operation 4

Operation 3:

Following is Kruskal Alogorithm.

( R , S , 0.84 km)

( N , O , 1.40 km)

( S , T , 1.43 km)

( W , X , 1.43 km)

( F , G , 1.50 km)

( U , Y , 1.67 km)

( O , T , 1.79 km)

( L , M , 1.93 km)

( G , H , 2.01 km)

( X , Y , 2.04 km)

( O , P , 2.13 km)

( T , U , 2.18 km)

( P , Q , 2.20 km)

( B , E , 2.28 km)

( I , J , 2.32 km)

( J , K , 2.32 km)

( K , L , 2.32 km)

( C , D , 2.36 km)

( Y , Z , 2.52 km)

( D , E , 2.54 km)

( V , W , 2.76 km)

( D , H , 2.88 km)

文本框完整内容：

Operation 3:  
Following is kruskal Alogorithm.  
( R , S , 0.84 km)  
( N , O , 1.40 km)  
( S , T , 1.43 km)  
( W , X , 1.43 km)  
( F , G , 1.50 km)  
( U , Y , 1.67 km)  
( O , T , 1.79 km)  
( L , M , 1.93 km)  
( G , H , 2.01 km)  
( X , Y , 2.04 km)  
( O , P , 2.13 km)  
( T , U , 2.18 km)  
( P , Q , 2.20 km)  
( B , E , 2.28 km)  
( I , J , 2.32 km)  
( J , K , 2.32 km)  
( K , L , 2.32 km)  
( C , D , 2.36 km)  
( Y , Z , 2.52 km)  
( D , E , 2.54 km)  
( V , W , 2.76 km)  
( D , H , 2.88 km)

( M , Q , 3.08 km)  
( G , J , 3.94 km)  
( A , F , 5.43 km)  
最短路径总长度为: 57.30 km.

Following is Prim Alogorithm.

( A , F , 5.43 km)  
( F , G , 1.50 km)  
( G , H , 2.01 km)  
( H , D , 2.88 km)  
( D , C , 2.36 km)  
( D , E , 2.54 km)  
( E , B , 2.28 km)  
( H , K , 3.94 km)  
( K , J , 2.32 km)  
( K , L , 2.32 km)  
( L , M , 1.93 km)  
( J , I , 2.32 km)  
( M , Q , 3.08 km)  
( Q , P , 2.20 km)  
( P , O , 2.13 km)  
( O , N , 1.40 km)  
( O , T , 1.79 km)  
( T , S , 1.43 km)  
( S , R , 0.84 km)  
( T , U , 2.18 km)  
( U , Y , 1.67 km)  
( Y , X , 2.04 km)  
( X , W , 1.43 km)  
( Y , Z , 2.52 km)  
( W , V , 2.76 km)

最短距离为 57.30 km.

operation4:(文本框可以上下滑动)

Map UI

MAP Navigation

Location 1:

Location 2:

Operation 1

Operation 2

Operation 3

Operation 4

Operation 4:  
( A , F , 5.43 km)  
( F , G , 1.50 km)  
( G , H , 2.01 km)  
( H , D , 2.88 km)  
( G , C , 3.19 km)  
( G , J , 3.94 km)  
( J , N , 3.23 km)  
( N , O , 1.40 km)  
( O , T , 1.79 km)  
( O , P , 2.13 km)  
( T , U , 2.18 km)  
( U , Y , 1.67 km)  
( P , O , 2.20 km)

文本框完整内容:

Operation 4:  
( A , F , 5.43 km)

( F , G , 1.50 km)  
( G , H , 2.01 km)  
( H , D , 2.88 km)  
( G , C , 3.19 km)  
( G , J , 3.94 km)  
( J , N , 3.23 km)  
( N , O , 1.40 km)  
( O , T , 1.79 km)  
( O , P , 2.13 km)  
( T , U , 2.18 km)  
( U , Y , 1.67 km)  
( P , Q , 2.20 km)  
( Y , Z , 2.52 km)  
( T , X , 2.77 km)  
( N , S , 2.91 km)  
( S , R , 0.84 km)  
( S , W , 2.46 km)  
( H , K , 3.94 km)  
( K , L , 2.32 km)  
( L , M , 1.93 km)  
( F , I , 4.39 km)  
( A , B , 7.64 km)  
( B , E , 2.28 km)  
( I , V , 11.04 km)

最短距离为 78.59 km.