数据结构与算法
作业报告

第 四 次

姓名	曹家豪
班级	软件 2204 班
学号	2226114017
电话	13572763245
Email	caojiahao@xjtu.stu.edu.cn
日期	2024-2-28

目录

实验 1

题目

该任务的主要内容是完成用图数据结构对迷宫的表示实现，成功表示之后，可以使用如下两个策略生成迷宫：

策略 1：随机擦除 70%的图中的边

策略 2：随机擦除 50%的图中的边

针对不同的策略，使用 Dijkstra 算法检测所生成的迷宫从入口（左上角）到出口（右下角）是 否有路径可达。对每一个策略可以执行 100 次，给出成功生成迷宫的概率（成功即代表从入口到出 口一定有路径可达）。

数据设计

Graph 类

vertices：表示图中顶点的数量，对应于迷宫中的单元格总数。在一个 20*20 的迷宫中，顶点总数即为 20*20=400。

List<List<Edge>> adjList;：邻接表用于存储图中每个顶点的相邻顶点（或称边）。每个顶点对应一个边的列表，这些边直接连接到其他顶点。

Graph(int vertices)：构造函数，用于初始化图。它接收顶点数量作为参数，初始化邻接表，为每个顶点创建一个空的边列表。

addEdge(int src, int dest)：用于在图中添加一条边。接受源顶点 src 和目标顶点 dest 作为参数，然后在它们的相应邻接列表中添加一个新的 Edge 对象。

removeEdge(int src, int dest)：用于从图中移除一条边。通过遍历源顶点和目标顶点的邻接列表并删除相应的边来实现。

getAllEdges()：此方法用于获取图中所有的边。它遍历每个顶点的邻接列表，收集并返回所有边的列表。

removeEdgesRandomly(double percentage)：根据指定的百分比随机移除边，用于生成迷宫。

isPathExists(int source, int destination)：使用 Dijkstra 算法检测从源顶点到目标顶点是否存在路径。

Edge 类：内部类，用于表示图中的边。包含源顶点和目标顶点，可以表示迷宫中单元格之间的连接关系。

MazeGenerator 类（负责实际的迷宫生成逻辑）：

initializeFullyConnectedGraph(Graph graph, int rows, int cols)：初始化一个完全连通的迷宫图。遍历迷宫的每个单元格，为每个单元格添加与其相邻（上、下、左、右）单元格的边，确保图的初始状态为完全连通。

calculateSuccessRate(int trials, double percentage)：用于计算给定策略成功生成迷宫的概率。它重复执行迷宫生成和路径检测的过程指定次数，记录成功生成可解迷宫的次数，并计算成功率。

算法设计

Graph 类：

addEdge 方法（为图添加一条连接两个顶点的无向边）：

方法接收两个参数：源顶点和目标顶点。为每个顶点在其邻接列表中添加一个新的 Edge 对象，表示边的双向性质。因此，这个方法实际上会在 adjList 中两个位置添加边信息，一次在 src 的列表中添加指向 dest 的边，另一次在 dest 的列表中添加指向 src 的边。

removeEdge 方法（从图中移除连接两个顶点的边）：

通过 src 和 dest 参数标识要移除的边。遍历这两个顶点的邻接列表，使用 removeIf 方法移除对应的边。确保从两个顶点的邻接列表中都移除了边信息。

removeEdgesRandomly 方法（随机移除图中一定比例的边，用于生成迷宫）：

首先，通过 getAllEdges 方法获取图中所有的边，并将它们随机打乱。基于输入的百分比 percentage 计算要移除的边的总数。遍历打乱后的边列表，根据计算出的数量移除边。由于每条边在邻接表中以两个顶点的形式存在，实际移除时需要调用 removeEdge 方法移除边。

isPathExists 方法（使用 Dijkstra 算法检测从源点到目标点是否存在路径）

首先初始化一个距离数组 distances，所有元素初始值为 Integer.MAX_VALUE，表示无限远，除了源点的距离初始化为 0。使用一个优先队列`pq`（基于距离排序），首先加入源点。然后，不断从

队列中取出距离最小的顶点，更新其相邻顶点的距离。如果目标顶点被取出，说明存在路径，返回 true。如果队列为空，说明没有路径，返回 false。

MazeGenerator 类：

initializeFullyConnectedGraph 方法（初始化一个完全连通的图，每个单元格作为一个顶点，单元格间的上下左右相邻关系通过边来表示）：

遍历 rows * cols 的迷宫格，对于每个单元格，计算其在图中的顶点索引，并根据其位置（非边缘）添加与其相邻单元格（上、下、左、右）的边。确保初始迷宫的图是完全连通的。

calculateSuccessRate 方法（计算给定边移除策略下，生成的迷宫中从入口到出口存在路径的概率）：

执行指定次数的试验，每次试验都从一个完全连通的迷宫图开始，使用 removeEdgesRandomly 方法移除一定比例的边，然后使用 isPathExists`方法检查从入口到出口是否存在路径。记录成功的次数，并根据总的试验次数计算成功率。

接下来在主程序中调用 removeEdgesRandomly 方法传入相应参数即可。

主干代码说明

```
1.
2. public class Graph {
3.     int vertices; // 图中顶点的数量
4.     List<List<Edge>> adjList; // 邻接表，用于存储每个顶点的相
   邻边
5.
6.     public Graph(int vertices) {
7.         this.vertices = vertices; // 初始化顶点数量
```

```
8.          adjList = new ArrayList<>();
9.          for (int i = 0; i < vertices; i++) {
10.             adjList.add(new ArrayList<>()); // 为每个顶点初始化一个边的列表
11.         }
12.     }
13.
14.     public void addEdge(int src, int dest) {
15.         adjList.get(src).add(new Edge(src, dest)); // 向源顶点的邻接列表中添加一条边
16.         adjList.get(dest).add(new Edge(dest, src)); // 向目标顶点的邻接列表中也添加一条边，因为是无向图
17.     }
18.
19.     public void removeEdge(int src, int dest) {
20.         adjList.get(src).removeIf(edge -> edge.dest == dest); // 从源顶点的邻接列表中移除指定的边
21.         adjList.get(dest).removeIf(edge -> edge.src == src); // 从目标顶点的邻接列表中也移除这条边
22.     }
23.
24.     public List<Edge> getAllEdges() {
25.         List<Edge> allEdges = new ArrayList<>();
26.         for (List<Edge> edges : adjList) {
27.             allEdges.addAll(edges); // 收集图中的所有边
28.         }
29.         return allEdges;
30.     }
31.
32.     public void removeEdgesRandomly(double percentage) {
33.         List<Edge> allEdges = getAllEdges(); // 获取所有边
34.         Collections.shuffle(allEdges); // 随机打乱边的顺序
35.         int edgesToRemove = (int) (allEdges.size() * percentage / 2); // 计算要移除的边的数量
36.         for (int i = 0; i < edgesToRemove; i++) {
37.             Edge edge = allEdges.get(i);
38.             removeEdge(edge.src, edge.dest); // 移除选定的边
39.         }
40.     }
41.
42.     public boolean isPathExists(int source, int destination) {
43.         int[] distances = new int[vertices]; // 存储从源点到每个顶点的距离
```

```
        boolean[] visited = new boolean[vertices]; // 标记顶点是否已被访问
        PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> distances[a] - distances[b]); // 优先队列，按距离排序

        for (int i = 0; i < vertices; i++) {
            distances[i] = Integer.MAX_VALUE; // 初始化所有距离为无限大
        }
        distances[source] = 0; // 源点到自己的距离为0
        pq.add(source); // 将源点加入优先队列

        while (!pq.isEmpty()) {
            int u = pq.poll(); // 获取并移除队列中距离最小的顶点
            if (u == destination) {
                return true; // 如果这个顶点是目标顶点，则存在路径
            }
            if (visited[u]) {
                continue; // 如果这个顶点已被访问，则跳过
            }
            visited[u] = true; // 标记顶点为已访问

            for (Edge edge : adjList.get(u)) {
                if (!visited[edge.dest] && distances[edge.dest] > distances[u] + 1) {
                    distances[edge.dest] = distances[u] + 1; // 更新到达相邻顶点的距离
                    pq.add(edge.dest); // 将这个顶点加入优先队列
                }
            }
        }
        return false; // 队列为空，未找到到达目标顶点的路径
    }

    class Edge {
        int src, dest; // 边的源顶点和目标顶点

        Edge(int src, int dest) {
            this.src = src;
            this.dest = dest; // 初始化边的起点和终点
        }
    }
```

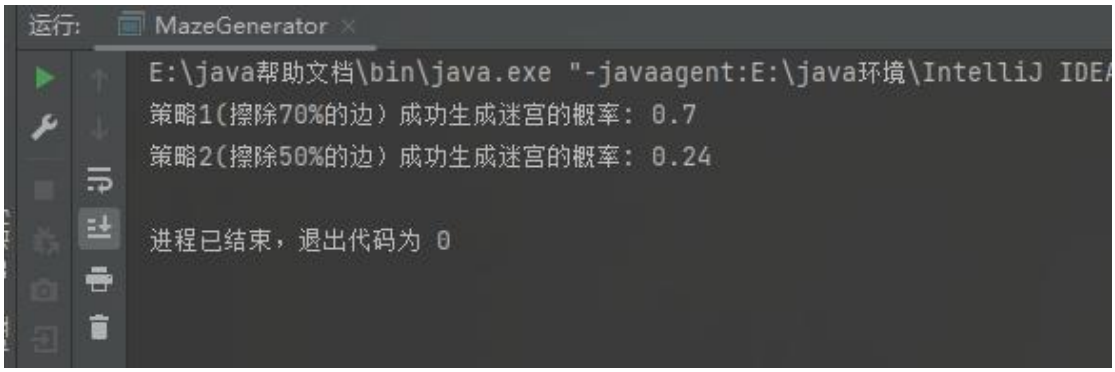
```
}
```

Ctrl+M

```
public class MazeGenerator {
    private static void initializeFullyConnectedGraph(Graph graph, int rows, int cols) {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int src = i * cols + j; // 计算当前单元格在图中的顶点索引
                if (i > 0) graph.addEdge(src, src - cols); // 添加与上方单元格的边
                if (j > 0) graph.addEdge(src, src - 1); // 添加与左侧单元格的边
                if (i < rows - 1) graph.addEdge(src, src + cols); // 添加与下方单元格的边
                if (j < cols - 1) graph.addEdge(src, src + 1); // 添加与右侧单元格的边
            }
        }
    }

    private static double calculateSuccessRate(int trials, double percentage) {
        int successCount = 0; // 记录成功生成可解迷宫的次数
        int size = 40; // 迷宫的大小，这里假设为40x40
        for (int t = 0; t < trials; t++) {
            Graph graph = new Graph(size * size); // 创建一个新的图实例
            initializeFullyConnectedGraph(graph, size, size); // 初始化为完全连通图
            graph.removeEdgesRandomly(percentage); // 随机移除一定比例的边
            if (graph.isPathExists(0, size * size - 1)) {
                successCount++; // 如果存在从入口到出口的路径，计入成功次数
            }
        }
        return (double) successCount / trials; // 计算并返回成功率

}
```

Ctrl+M

运行结果展示



总结和收获

(1) 了解了如何利用图论算法来解决实际问题。

(2) 学会实现了图的数据结构

(3) 学习了 Dijkstra 的具体实现与应用

实验 2

题目

为了确保每次迷宫的生成都是成功的，将不再采用任务 1 中的随机擦除边的生成方式，而是采 用 Kruskal 最小支撑树的算法来实现迷宫生成。具体的执行步骤如下：

1. 为任务 1 中用来表示迷宫的图中的每一条边都随机生成一个权值(此时的图是一个带权图，所以在表示时可能会和任务 1 的表示有出入，请注意这个细节)；

2. 利用 Kruskal 算法对步骤 1 中的图生成最小支撑树 T；

3. 将 T 中的每一条边相对应在迷宫中的边擦除掉，此时迷宫就生成了，这样生成的迷宫一定是成功的。

数据设计

NewGraph 类

在 **NewGraph** 类中，对迷宫的表示进行了扩展，以支持 Kruskal 最小支撑树算法的实现：

- **addEdge(int src, int dest, int weight)**：此方法用于向图中添加一条边，边包含两个顶点 **src** 和 **dest** 以及边的权重 **weight**。
- **List<Edge> kruskalMST()**：该方法实现 Kruskal 算法，用于从当前图中生成最小支撑树。该算法首先将所有边按权重排序，然后逐一检查每条边，使用并查集（**UnionFind** 类）来避免环的产生，并逐步构建最小支撑树。

Edge 类

Edge 类设计用于表示图中的一条边，包含以下成员：

- **src, dest;**：边的起点和终点，对应于迷宫中的两个单元格。
- **weight;**：边的权重，为 Kruskal 算法中边选择的依据。
- **compareTo(Edge compareEdge)**：实现 **Comparable** 接口，使边可以根据权重进行排序，为 Kruskal 算法中边的排序提供依据。

算法设计

1. **初始化带权图**：首先，构建一个完全连通的带权图，每个顶点代表迷宫的一个单元格，每条边代表单元格之间的连接，边的权重随机生成。
2. **应用 Kruskal 算法**：使用 Kruskal 算法对带权图生成最小支撑

树，确保没有环路，同时保持图的连通性。

通过以下步骤在图中生成最小支撑树：

排序：所有的边按照权重从小到大排序。

选择：按排序后的顺序遍历每条边，使用并查集检查当前边的两个顶点是否已经在同一连通分量中。

合并：如果两个顶点不在同一连通分量中，则将这条边加入最小支撑树中，并在并查集中合并这两个顶点的连通分量。

3. **生成迷宫布局**：根据最小支撑树中的边生成迷宫布局，最小支撑树中的边表示迷宫中的通道，不在树中的边表示墙壁。
4. **打印迷宫**：遍历迷宫布局数组，使用不同的字符打印通道和墙壁，将迷宫可视化。

主要方法的实现逻辑

NewGraph 类

- addEdge(int src, int dest, int weight)：与原 Graph 类中的 addEdge 不同，这里添加的边包含权重信息，用于 Kruskal 算法中根据边的权重进行排序和选择。
- kruskalMST()：新增方法，实现 Kruskal 算法。首先对所有边按权重排序，然后遍历边，使用并查集判断加入这条边是否会形成环。如果不会，则将边加入最小支撑树中，通过并查集合并两个顶点。

NewMazeGenerator 类

- initializeGraphWithWeights()：与 MazeGenerator 中的

initializeFullyConnectedGraph 类似，此方法初始化一个完全连通的带权图。不同之处在于，这里为图的每条边随机分配权重。

- generateMazeLayout()：根据 Kruskal 算法生成的最小支撑树确定迷宫的通道和墙壁。在最小支撑树中的边对应迷宫中的通道，不在树中的边对应墙壁。
- printMaze()：打印迷宫的方法与 MazeGenerator 中类似，但在这里是基于最小支撑树生成的布局打印迷宫，而不是基于随机擦除边的结果。

主干代码说明

```
1. public class NewGraph {
2.     int vertices; // 图中顶点的数量
3.     List<Edge> edges; // 存储图中所有边的列表
4.
5.     public NewGraph(int vertices) {
6.         this.vertices = vertices; // 初始化顶点数量
7.         edges = new ArrayList<>(); // 初始化边的列表
8.     }
9.
10.    public void addEdge(int src, int dest, int weight) {
11.        edges.add(new Edge(src, dest, weight)); // 向图中添加一条带权重的边
12.    }
13.
14.    public List<Edge> kruskalMST() {
15.        Collections.sort(edges); // 将边按权重排序
16.        UnionFind unionFind = new UnionFind(vertices); // 初始化并查集
17.        List<Edge> result = new ArrayList<>(); // 存储最小生成树中的边
18.
19.        for (Edge edge : edges) {
20.            int x = unionFind.find(edge.src); // 查找源顶点的根
```

```
            int y = unionFind.find(edge.dest); // 查找目标顶点的根
            if (x != y) {
                result.add(edge); // 如果根不同，添加这条边到结果中
                unionFind.union(x, y); // 合并两个顶点所在的集合
            }
        }
        return result; // 返回最小生成树的边集
    }

    class Edge implements Comparable<Edge> {
        int src, dest, weight; // 边的起点、终点和权重

        Edge(int src, int dest, int weight) {
            this.src = src;
            this.dest = dest;
            this.weight = weight; // 初始化边
        }

        @Override
        public int compareTo(Edge compareEdge) {
            return this.weight - compareEdge.weight; // 按权重比较边
        }
    }
}
```

Ctrl+M

```
public class NewMazeGenerator {
    private static final int size = 40; // 迷宫的大小

    public static void main(String[] args) {
        NewGraph graph = new NewGraph(size * size); // 创建一个新的图实例
        initializeGraphWithWeights(graph, size, size); // 初始化带权重的图
        List<NewGraph.Edge> mst = graph.kruskalMST(); // 生成最小生成树
        boolean[][] maze = generateMazeLayout(mst, size); // 生成迷宫布局
        printMaze(maze, size); // 打印迷宫
    }

```

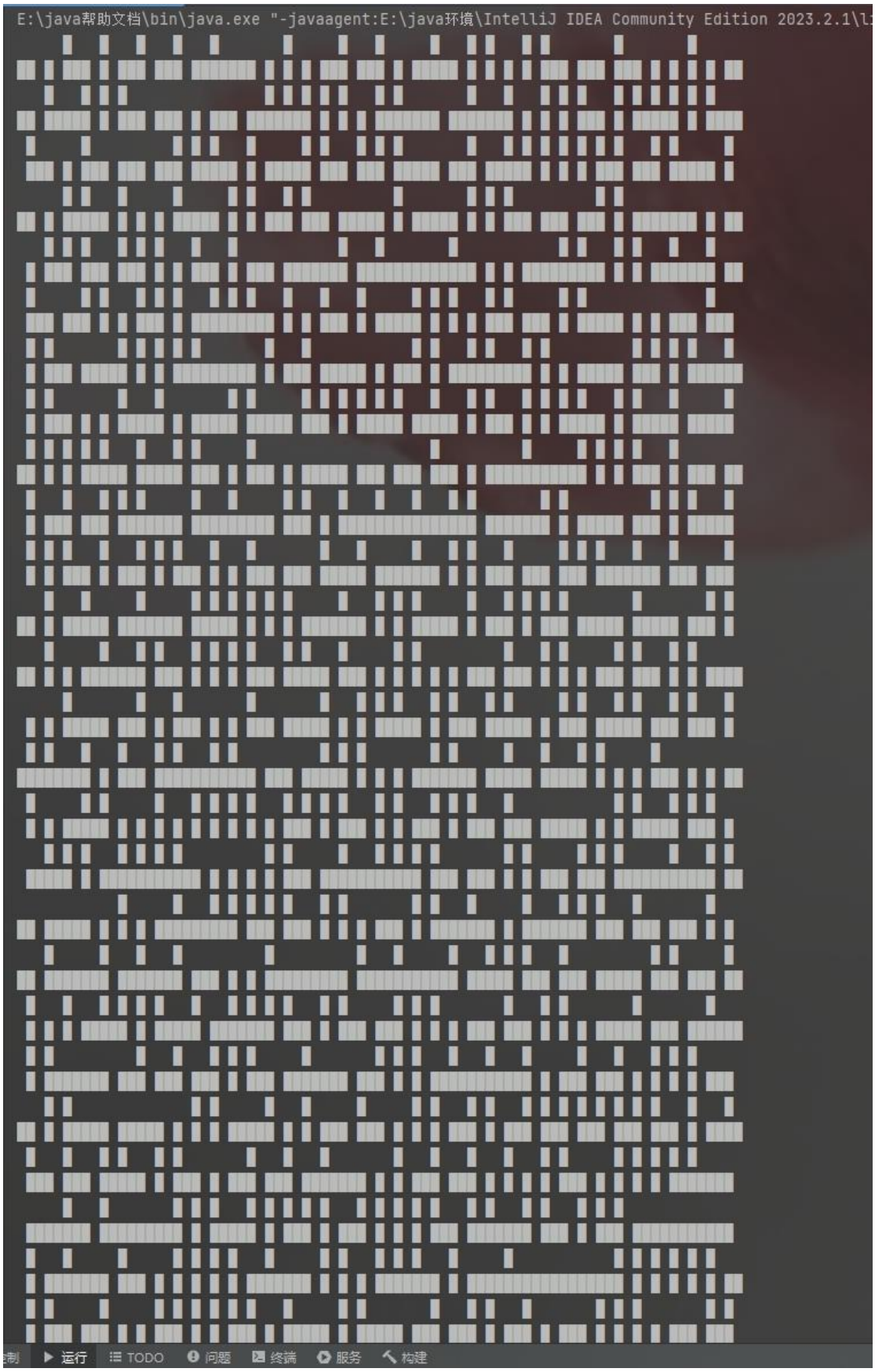
```
12.     private static void initializeGraphWithWeights(NewGraph graph, int rows, int cols) {
13.         Random rand = new Random(); // 随机数生成器
14.         for (int i = 0; i < rows; i++) {
15.             for (int j = 0; j < cols; j++) {
16.                 int src = i * cols + j; // 计算顶点索引
17.                 if (i < rows - 1) graph.addEdge(src, src + cols, rand.nextInt(100) + 1); // 向下连接
18.                 if (j < cols - 1) graph.addEdge(src, src + 1, rand.nextInt(100) + 1); // 向右连接
19.             }
20.         }
21.     }
22.
23.     private static boolean[][] generateMazeLayout(List<NewGraph.Edge> mst, int size) {
24.         boolean[][] maze = new boolean[size * 2 - 1][size * 2 - 1]; // 创建迷宫布局数组
25.         for (NewGraph.Edge edge : mst) {
26.             int x1 = edge.src / size, y1 = edge.src % size; // 计算源顶点的行列
27.             int x2 = edge.dest / size, y2 = edge.dest % size; // 计算目标顶点的行列
28.             maze[x1 * 2][y1 * 2] = true; // 标记源顶点位置
29.             maze[x2 * 2][y2 * 2] = true; // 标记目标顶点位置
30.             if (x1 == x2) maze[x1 * 2][y1 * 2 + 1] = true; // 如果在同一行，标记中间的单元格
31.             if (y1 == y2) maze[x1 * 2 + 1][y1 * 2] = true; // 如果在同一列，标记中间的单元格
32.         }
33.         return maze; // 返回迷宫布局
34.     }
35.
36.     private static void printMaze(boolean[][] maze, int size) {
37.         for (boolean[] row : maze) {
38.             for (boolean cell : row) {
39.                 System.out.print(cell ? " " : "\u2588"); // 打印迷宫，空白或墙
40.             }
41.             System.out.println();
42.         }
43.     }
44. }
```

```
    private int[] parent; // 存储每个元素的父节点
    private int[] rank; // 存储树的“秩”

    public UnionFind(int size) {
        parent = new int[size];
        rank = new int[size];
        for (int i = 0; i < size; i++) {
            parent[i] = i; // 初始时，每个元素的父节点是它自己
            rank[i] = 0; // 初始时，每棵树的秩都是0
        }
    }

    public int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // 路径压缩
        }
        return parent[x]; // 返回x的根节点
    }

    public void union(int x, int y) {
        int rootX = find(x); // 查找x的根节点
        int rootY = find(y); // 查找y的根节点
        if (rootX != rootY) { // 如果不在同一个集合中
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY; // 将秩较小的树的根节点连接到秩较大的树上
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX; // 如果秩相同，任意连接一个，并增加秩
                rank[rootX]++;
            }
        }
    }

```

运行结果展示

将生成迷宫可视化，如下所示（空格表示通路，可以经过。黑色方块

表示有边进行堵塞）：


E:\java帮助文档\bin\java.exe "-javaagent:E:\java环境\IntelliJ IDEA Community Edition 2023.2.1\li
运行
TODO
问题
终端
服务
构建

（上下图紧邻，无重复部分）

经手算验证，确实可以通过。

总结和收获

（1）加深了对图论算法及其在实际问题中应用的理解。

（2）自定义了并查集，感受其在算法中的应用，尤其是在管理元素分组和合并时的高效性。

（3）应用 Kruskal 算法进行最小支撑数的生成，对图论相关算法的实现更加熟悉。

附录

```
1. import java.util.ArrayList;
2. import java.util.Collections;
3. import java.util.List;
4. import java.util.PriorityQueue;
5. public class Graph {
6.     int vertices;
7.     List<List<Edge>> adjList;
8.
9.     public Graph(int vertices) {
10.        this.vertices = vertices;
11.        adjList = new ArrayList<>();
12.        for (int i = 0; i < vertices; i++) {
13.            adjList.add(new ArrayList<>());
14.        }
15.    }
16.
17.    public void addEdge(int src, int dest) {
18.        adjList.get(src).add(new Edge(src, dest));
19.        adjList.get(dest).add(new Edge(dest, src));
20.    }
21.
22.    public void removeEdge(int src, int dest) {
23.        adjList.get(src).removeIf(edge -> edge.dest == dest
   );
24.        adjList.get(dest).removeIf(edge -> edge.src == src)
   ;
25.    }
26.
27.    public List<Edge> getAllEdges() {
28.        List<Edge> allEdges = new ArrayList<>();
29.        for (List<Edge> edges : adjList) {
30.            allEdges.addAll(edges);
31.        }
32.        return allEdges;
33.    }
34.
35.    public void removeEdgesRandomly(double percentage) {
36.        List<Edge> allEdges = getAllEdges();
37.        Collections.shuffle(allEdges);
38.        int edgesToRemove = (int) (allEdges.size() * percen
   tage / 2);
39.        for (int i = 0; i < edgesToRemove; i++) {
```

```
            Edge edge = allEdges.get(i);
            removeEdge(edge.src, edge.dest);
        }
    }

    public boolean isPathExists(int source, int destination) {
        int[] distances = new int[vertices];
        boolean[] visited = new boolean[vertices];
        PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> distances[a] - distances[b]);

        for (int i = 0; i < vertices; i++) {
            distances[i] = Integer.MAX_VALUE;
        }
        distances[source] = 0;
        pq.add(source);

        while (!pq.isEmpty()) {
            int u = pq.poll();
            if (u == destination) {
                return true;
            }
            if (visited[u]) {
                continue;
            }
            visited[u] = true;

            for (Edge edge : adjList.get(u)) {
                if (!visited[edge.dest] && distances[edge.dest] > distances[u] + 1) {
                    distances[edge.dest] = distances[u] + 1;
                    pq.add(edge.dest);
                }
            }
        }
        return false;
    }

    class Edge {
        int src, dest;

        Edge(int src, int dest) {
            this.src = src;
```

```
            this.dest = dest;
        }
    }
}
```

Ctrl+M

```
public class MazeGenerator {
    private static void initializeFullyConnectedGraph(Graph graph, int rows, int cols) {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int src = i * cols + j;
                if (i > 0) graph.addEdge(src, src - cols); // 上
                if (j > 0) graph.addEdge(src, src - 1);   // 左
                if (i < rows - 1) graph.addEdge(src, src + cols); // 下
                if (j < cols - 1) graph.addEdge(src, src + 1);   // 右
            }
        }
    }

    private static double calculateSuccessRate(int trials, double percentage) {
        int successCount = 0;
        int size = 40; // 迷宫大小为40x40
        for (int t = 0; t < trials; t++) {
            Graph graph = new Graph(size * size);
            initializeFullyConnectedGraph(graph, size, size);
            graph.removeEdgesRandomly(percentage);
            if (graph.isPathExists(0, size * size - 1)) {
                successCount++;
            }
        }
        return (double) successCount / trials;
    }

    public static void main(String[] args) {
        int trials = 100; // 测试次数
        double strategy1SuccessRate = calculateSuccessRate(trials, 0.7);
```

```
        double strategy2SuccessRate = calculateSuccessRate(trials, 0.5);

        System.out.println("策略 1 成功生成迷宫的概率: " + strategy1SuccessRate);
        System.out.println("策略 2 成功生成迷宫的概率: " + strategy2SuccessRate);
    }
}
```

Ctrl+M

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

public class NewGraph {
    int vertices;
    List<Edge> edges;

    public NewGraph(int vertices) {
        this.vertices = vertices;
        edges = new ArrayList<>();
    }

    public void addEdge(int src, int dest, int weight) {
        edges.add(new Edge(src, dest, weight));
    }

    public List<Edge> kruskalMST() {
        Collections.sort(edges);
        UnionFind unionFind = new UnionFind(vertices);
        List<Edge> result = new ArrayList<>();

        for (Edge edge : edges) {
            int x = unionFind.find(edge.src);
            int y = unionFind.find(edge.dest);
            if (x != y) {
                result.add(edge);
                unionFind.union(x, y);
            }
        }
        return result;
    }

```

```
35.     class Edge implements Comparable<Edge> {
36.         int src, dest, weight;
37.
38.         Edge(int src, int dest, int weight) {
39.             this.src = src;
40.             this.dest = dest;
41.             this.weight = weight;
42.         }
43.
44.         @Override
45.         public int compareTo(Edge compareEdge) {
46.             return this.weight - compareEdge.weight;
47.         }
48.     }
49. }
```

Ctrl+M

```
1. import java.util.List;
2. import java.util.Random;
3.
4. public class NewMazeGenerator {
5.     private static final int size = 40; // 迷宫大小
6.
7.     public static void main(String[] args) {
8.         NewGraph graph = new NewGraph(size * size);
9.         initializeGraphWithWeights(graph, size, size);
10.         List<NewGraph.Edge> mst = graph.kruskalMST();
11.         boolean[][] maze = generateMazeLayout(mst, size);
12.         printMaze(maze, size);
13.     }
14.
15.     private static void initializeGraphWithWeights(NewGraph
    graph, int rows, int cols) {
16.         Random rand = new Random();
17.         for (int i = 0; i < rows; i++) {
18.             for (int j = 0; j < cols; j++) {
19.                 int src = i * cols + j;
20.                 if (i < rows - 1) graph.addEdge(src, src +
   cols, rand.nextInt(100) + 1); // 下
21.                 if (j < cols - 1) graph.addEdge(src, src +
   1, rand.nextInt(100) + 1);   // 右
22.             }
23.         }
24.     }
25.
```

```
26.     private static boolean[][] generateMazeLayout(List<NewGraph.Edge> mst, int size) {
27.         boolean[][] maze = new boolean[size * 2 - 1][size * 2 - 1];
28.         for (NewGraph.Edge edge : mst) {
29.             int x1 = edge.src / size, y1 = edge.src % size;
30.             int x2 = edge.dest / size, y2 = edge.dest % size;
31.             maze[x1 * 2][y1 * 2] = true;
32.             maze[x2 * 2][y2 * 2] = true;
33.             if (x1 == x2) maze[x1 * 2][y1 * 2 + 1] = true; // 水平相邻
34.             if (y1 == y2) maze[x1 * 2 + 1][y1 * 2] = true; // 垂直相邻
35.         }
36.         return maze;
37.     }
38.
39.     private static void printMaze(boolean[][] maze, int size) {
40.         for (boolean[] row : maze) {
41.             for (boolean cell : row) {
42.                 System.out.print(cell ? " " : "");
43.             }
44.             System.out.println();
45.         }
46.     }
47. }
```

Ctrl+M

```
1. public class UnionFind {
2.     private int[] parent;
3.     private int[] rank;
4.
5.     public UnionFind(int size) {
6.         parent = new int[size];
7.         rank = new int[size];
8.         for (int i = 0; i < size; i++) {
9.             parent[i] = i;
10.             rank[i] = 0;
11.         }
12.     }
13.
14.     public int find(int x) {
15.         if (parent[x] != x) {
```

```
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
}
```

Ctrl+M