实验三——地图路由

学号：22009200894 姓名：王越洋

1.实验背景与目的

（1）实验背景

最短路径问题是图论中重要的经典问题之一，广泛应用于交通网络、路由选择、物流调度等场景。Dijkstra 算法是求解单源最短路径的著名算法之一，通常用于加权有向图且权重为非负数的情况。该算法的效率在很大程度上依赖于优先队列的性能，因此在不同实现的优先队列下，算法的时间和空间复杂度也有所不同。

（2）实验目的

比较在不同优先队列（Binary Heap、Multiway Heap）实现下 Dijkstra 算法的性能差异。本次实验研究不同维度的多路堆对算法效率的影响，从而分析选择合适优先队列的意义。通过多次实验测量运行时间和内存占用情况，验证并评估不同实现的实际表现。

2.实验内容

（1）实现 Dijkstra 算法

基于加权有向图数据结构，编写 Dijkstra 算法的实现类 DijkstraSP，使用优先队列管理候选节点并执行放松操作。

（2）实现多种优先队列

Binary Heap（IndexMinPQ）：二叉堆的优先队列实现。

Multiway Heap（IndexMultiwayMinPQ）：支持多个子节点的多路堆优先队列，并允许自定义维度 d。

（3）实验设计与运行

使用不同优先队列类型（Binary、Multiway（d=3）、Multiway（d=4））运行 Dijkstra 算法。

测量每种配置下的运行时间和内存使用。

结果分析与总结：对实验结果进行分析，包括各配置下的运行效率和资源占用情况。

3. 代码实现与原理

3.1 DijkstraSP 类

（1）原理

DijkstraSP 类实现了 Dijkstra 算法的核心逻辑。算法的核心思想是通过维护源点到各节点的最短路径长度，不断从未访问节点集合中选择当前最短路径的节点进行扩展。

（2）关键代码说明

1.初始化源节点：将源节点距离初始化为 0，其他节点距离为正无穷。

2.优先队列操作：优先队列的主要操作包括：

- insert(s, distTo[s])：将源节点 s 插入优先队列。

- delMin()：取出当前距离源节点最近的节点。
- decreaseKey()：若发现更短路径，更新节点优先级。

3.放松操作：检查是否可以通过某条边找到更短路径并更新邻接节点的路径长度。

```java
public class DijkstraSP {
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    private IndexMinPQ<Double> pq; // 使用二叉堆优先队列

    public DijkstraSP(EdgeWeightedDigraph G, int s) {
        distTo[s] = 0.0;
        pq.insert(s, distTo[s]);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }

    private void relax(DirectedEdge e) {
        int v = e.from(), w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else pq.insert(w, distTo[w]);
        }
    }
}
```

3.2 IndexMultiwayMinPQ 类

（1）原理

IndexMultiwayMinPQ 实现了多路堆优先队列，允许设定堆的维度 d，使得每个节点最多拥有 d 个子节点。理论上，多路堆可以减少堆的高度，从而降低插入和删除最小值的操作复杂度。

（2）代码片段

```java
public class IndexMultiwayMinPQ<Key> {
    private final int d;      // 堆的维度
    private int[] pq;         // 堆数组
    private int[] qp;         // 逆数组
    private Key[] keys;       // 优先级数组

```

```
    public IndexMultiwayMinPQ(int N, int D) {
        this.d = D;
        pq = new int[N + D];
        qp = new int[N + D];
        keys = (Key[]) new Comparable[N + D];
    }

    private void swim(int i) {
        while (i > 0 && greater((i - 1) / d, i)) {
            exch(i, (i - 1) / d);
            i = (i - 1) / d;
        }
    }

    private void sink(int i) {
        while (d * i + 1 < n) {
            int j = minChild(i);
            if (!greater(i, j)) break;
            exch(i, j);
            i = j;
        }
    }
}
```

3.3 EdgeWeightedDigraph 类

EdgeWeightedDigraph 管理了图的顶点和边以及每条边的权重。该类使用邻接表存储各顶点的邻接边，从而使边的管理和遍历更加高效。以下是代码的逻辑和关键代码的简述：

（1）构造方法

EdgeWeightedDigraph(int V)：初始化一个包含 V 个顶点且没有边的空图。
EdgeWeightedDigraph(int V, int E)：创建包含 V 个顶点、E 条随机边的图。
EdgeWeightedDigraph(In in)：从输入流中读取顶点和边的定义并构建图。
EdgeWeightedDigraph(EdgeWeightedDigraph G)：通过深拷贝另一个图 G 来创建新的图对象。

（2）添加和获取方法

addEdge(DirectedEdge e)：将有向边 e 添加到图中，并更新邻接表和入度数组。

adj(int v)：返回与顶点 v 相连的所有有向边。

indegree(int v) 和 outdegree(int v)：分别返回顶点 v 的入度和出度。
edges()：返回图中的所有有向边集合。

```
public Iterable<DirectedEdge> edges() {
```

```java
        Bag<DirectedEdge> list = new Bag<DirectedEdge>();
        for (int v = 0; v < V; v++) {
            for (DirectedEdge e : adj(v)) {
                list.add(e);    // 将每条边添加到集合中
            }
        }
        return list;
}
```

3.4 DijkstraSPMap 类

（1）从文件读取数据并初始化图

1.文件读取和顶点、边信息解析：函数从文件中读取顶点和边的数量，初始化一个带权有向图 EdgeWeightedDigraph。

- 时间复杂度：假设文件中包含 V 个顶点和 E 条边，则读取数据的时间复杂度约为 O(V+E)。
- 空间复杂度：图结构和数据存储使用 O(V+E) 的空间，顶点数和边数均会影响内存需求。

2.存储顶点的坐标：使用 HashMap 将每个顶点的编号与其坐标（x 和 y）关联，以便计算边的权重（距离）。

- 时间复杂度：使用 HashMap 插入 V 个顶点坐标，时间复杂度约为 O(V)。
- 空间复杂度：HashMap 中存储了 V 个顶点的坐标信息，空间复杂度为 O(V)。

3.构建边和计算边权重：读取边信息，并根据顶点坐标计算每条边的权重（使用欧几里得距离）。边的权重计算完毕后，通过 addEdge 方法将边加入到图中。

- 时间复杂度：对于每条边进行一次距离计算和插入操作，总时间复杂度为 O(E)。
- 空间复杂度：邻接表存储 E 条边，空间复杂度为 O(E)。

（2）执行 Dijkstra 算法

1.选择优先队列类型：根据输入参数 pqType，函数选择使用 binary 或 multiway（多路堆）作为优先队列实现来执行 Dijkstra 算法。

- 二叉堆：当 pqType 为 binary 时，选择 D=2 的多路堆，相当于一个二叉堆。
- 多路堆：当 pqType 为 multiway 时，函数使用指定的 D 值作为多路堆的维度。

2.Dijkstra 算法的执行：基于选择的优先队列类型，创建 DijkstraSP 对象并计算最短路径。

- 时间复杂度：Dijkstra 算法的复杂度主要依赖于优先队列操作。对于二叉堆实现，时间复杂度是 O((V+E)logV)；对于多路堆，时间复杂度与 D 相关，较大的 D 可以减少树的高度，但单次插入和删除操作的时间会增大。
- 空间复杂度：优先队列存储了图中顶点和边的路径信息，其空间复杂度约为 O(V+E)。

（3）记录结束时间和内存使用

executionTime：算法总执行时间，计算方法为 endTime - startTime，单

位为毫秒。

memoryUsed：算法执行过程中实际使用的内存，通过 endMemory - startMemory 计算（单位为 MB）。这里将字节转换为 MB 方便展示。

4.结果分析

（1）数据

对 binary 和 multiway (d=3 和 d=4) 队列类型下的Dijkstra算法分别进行了运行，记录了它们的执行时间和内存使用情况。结果如下：

优先队列类型	执行时间（毫秒）	内存使用（MB）
Binary Heap	699	10
Multiway Heap (d=3)	567	11
Multiway Heap (d=4)	298	11

（2）分析

1.执行时间：实验结果显示，multiway 优先队列的执行时间随维度 d 的增加而减小。这是因为随着 d 的增大，堆的高度降低了，使得 insert 和 delMin 操作变得更快。

2.内存使用：d=3 的多路堆使用了最多的内存，而 d=4 内存使用最少。这可能是由于更大的维度导致了更高的内存利用率，尤其是在降低堆高度和优化数组访问时，使堆结构更紧凑。

3.比较：多路堆在执行时间方面有显著优势，尤其是在更大维度的设置下，但内存消耗也需要权衡。因此，如果对内存使用要求较高，d=4 是较优的选择。

（3）复杂度

插入、减少键值、获取最小键复杂度：$O(d \cdot \log_d(n))$。

删除最小键复杂度：$O(d \cdot \log_d(n))$。

删除键、增加键复杂度：$O(d \cdot \log_d(n))$。

随着 d 增加，堆的层数减少，使得插入和删除的复杂度下降，但由于每层比较的子节点增多，删除操作会有额外的开销。

5.总结

本实验展示了不同优先队列实现对 Dijkstra 算法的性能影响。通过实验结果可以得出以下结论：

1.在执行时间方面，multiway 优先队列优于 binary 优先队列，且多路堆的维度越大，性能提升越显著。

2.在内存消耗方面，multiway 的内存需求在 d=3 时显著高于 d=4，更大的维度反而有助于降低内存开销。

3.实际应用中应根据具体需求选择合适的优先队列。对于需要频繁操作大量节点的场景，多路堆（d=4）是一种较优选择，可以在性能和内存占用之间取得平衡。

6. 收获与反思

（1）收获

通过本实验，我掌握了如何在 Dijkstra 算法中使用优先队列优化性能。深入理解了多路堆的结构与操作的细节，进一步加深了对数据结构效率和算法实现

之间关系的理解。同时，学习到如何通过实验数据分析来验证和总结算法的效率。

（2）反思

在实验过程中，我发现提高维度虽然可以减少堆高度，但会增加每层的子节点数，这在实际场景中需要动态调整以平衡时间和内存的需求。