图论算法Kruskal、Prim与Dijkstra详细解析

Kruskal 算法：贪心选边构建最小生成树

实现步骤详解

1. **边的排序**：将图中所有的边按照权值进行升序排序。这一步是整个算法的基础，通过排序，我们可以方便地按照权值从小到大的顺序依次考察每条边。例如，假设有一个包含多个边的图，边的权值分别为 5、3、8、2 等，排序后就会得到 2、3、5、8 这样的顺序，便于后续的选择操作

2. **并查集初始化**：并查集是一种用于管理元素分组的数据结构，在这里，我们初始化并查集，让每个节点都单独成为一个集合。这样做的目的是为了方便判断两个节点是否属于同一个连通分量，在后续选择边的过程中，若一条边的两个端点不在同一个集合中，就说明加入这条边不会形成环。比如，有节点 A、B、C，初始化后它们分别在各自的集合 {A}、{B}、{C} 中。

3. **边的遍历与选择**：从排序后的边集合中开始遍历。对于每一条边，检查它的两个端点是否在同一个集合中。如果不在同一个集合，说明这条边不会导致环的形成，那么就将这条边加入到最小生成树中，并执行合并操作（union (u, v)），将这两个端点所在的集合合并成一个集合；如果两个端点在同一个集合中，就跳过这条边，因为加入它会形成环。例如，遍历到一条边连接节点 A 和 B，若 A 和 B 不在同一个集合，就将这条边加入最小生成树，然后合并 A 和 B 所在的集合。

4. **结束条件**：当遍历完所有的边，或者已经选中了 n - 1 条边（n 为图中顶点的数量）时，算法结束。因为对于一个具有 n 个顶点的连通图，其最小生成树恰好包含 n - 1 条边，当收集到足够数量的边时，最小生成树就构建完成了。

Java 代码实现

```
import java.util.Arrays;
import java.util.Comparator;

// 定义边类
class Edge {
    int start;
    int end;
    int weight;

    public Edge(int start, int end, int weight) {
        this.start = start;
        this.end = end;
        this.weight = weight;
    }
}

// 定义并查集类
class DisjointSet {
    int[] parent;
    int[] rank;
```

```
    public DisjointSet(int size) {
        parent = new int[size];
        rank = new int[size];
        for (int i = 0; i < size; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    // 查找节点的根节点
    public int find(int u) {
        if (parent[u] != u) {
            parent[u] = find(parent[u]);
        }
        return parent[u];
    }

    // 合并两个集合
    public void union(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);

        if (rootU != rootV) {
            if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            } else if (rank[rootU] < rank[rootV]) {
                parent[rootU] = rootV;
            } else {
                parent[rootV] = rootU;
                rank[rootU]++;
            }
        }
    }
}

// 定义Kruskal算法类
class Kruskal {
    public void kruskalMST(Edge[] edges, int vertexCount) {
        // 按权重对边进行排序
        Arrays.sort(edges, Comparator.comparingInt(e -> e.weight));

        DisjointSet ds = new DisjointSet(vertexCount);
        Edge[] result = new Edge[vertexCount - 1];
        int e = 0; // 计数选中的边
        int i = 0; // 迭代边的索引

        while (e < vertexCount - 1 && i < edges.length) {
            Edge nextEdge = edges[i++];
            int x = ds.find(nextEdge.start);
            int y = ds.find(nextEdge.end);

            if (x != y) {
                result[e++] = nextEdge;
                ds.union(x, y);
```

```
            }
        }

        // 打印结果
        System.out.println("最小生成树的边：");
        for (int j = 0; j < e; j++) {
            System.out.println(result[j].start + " -- " + result[j].end + "  权重:
" + result[j].weight);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Edge[] edges = {
                new Edge(0, 1, 10),
                new Edge(0, 2, 6),
                new Edge(0, 3, 5),
                new Edge(1, 3, 15),
                new Edge(2, 3, 4)
        };
        int vertexCount = 4;

        Kruskal kruskal = new Kruskal();
        kruskal.kruskalMST(edges, vertexCount);
    }
}
```

复杂度分析

1. **排序边**：使用常见的排序算法（如快速排序、Java 中的 Arrays.sort () ）对边进行排序，时间复杂度为 O (E log E)，其中 E 是图中边的数量。这是因为排序算法的时间复杂度通常与元素数量和元素之间的比较次数有关，对于 E 条边，比较次数大致为 E log E。

2. **并查集操作**：在 Kruskal 算法中，需要对每条边进行两次查找操作（find）和可能的一次合并操作（union）。由于并查集采用了路径压缩和按秩合并的优化策略，每次查找和合并操作的时间复杂度接近常数时间，所以对 E 条边进行并查集操作的总时间复杂度为 O (E α(V))，其中 α(V) 是阿克曼函数的反函数，在实际应用中，α(V) 增长极其缓慢，通常可以近似认为是常数，所以这部分时间复杂度近似为 O (E)。

3. **总复杂度**：综合排序边和并查集操作，Kruskal 算法的总时间复杂度为 O (E log E)。因为在图论中，边数 E 和顶点数 V 存在关系 E <= V^2，所以 log E 和 log V 是同数量级的，有时也可以表示为 O (E log V)。

特点总结

1. **对边稀疏的图效率高**：在边稀疏的图中，边的数量 E 相对较小，Kruskal 算法主要的时间消耗在边的排序上，而并查集操作的时间复杂度近似为线性，所以整体效率较高。相比其他一些算法（如 Prim 算法），在这种情况下更具优势。

2. **实现简单**：Kruskal 算法的实现思路相对直观，主要包括边的排序和并查集操作两个核心部分，代码实现相对简洁，易于理解和掌握。

考试易错点提醒

1. **忘记判环**：在选择边加入最小生成树时，必须要判断加入这条边是否会形成环。如果忘记判环，可能会导致生成的不是最小生成树，而是包含多余边的连通图。在考试中，这是一个常见的错误点，需要特别注意使用并查集等方法进行判环操作。

2. **忘记 MST 只需 n - 1 条边**：对于一个具有 n 个顶点的连通图，其最小生成树恰好包含 n - 1 条边。在实现算法时，如果没有正确处理这个结束条件，可能会继续遍历边，导致不必要的计算，甚至可能因为超出数组边界等问题导致程序出错 。在考试中，要明确这个条件，当收集到足够数量的边时，及时结束算法。

Prim 算法：从点出发的最小生成树构建法

核心思想解读

Prim 算法同样是用于求解无向连通图最小生成树的经典算法。它的核心思想可以理解为从某个节点开始 “生长” 最小生成树。就像在一片区域里建立通信基站网络，我们先选择一个起始位置建立第一个基站，然后每次都选择离已建立基站最近（权值最小）的未建站位置去建立新基站 ，直到所有位置都被基站覆盖，这样就构建出了一个成本最低（总边权最小）的通信网络，这个网络就是最小生成树。具体来说，Prim 算法从任意一个节点出发，将其加入到最小生成树的顶点集合中，然后不断从连接已在最小生成树中的顶点和未在最小生成树中的顶点的边中，选择权值最小的边，并将这条边连接的未在最小生成树中的顶点加入到最小生成树的顶点集合中，如此反复，直到所有顶点都被加入到最小生成树中。这种基于贪心的策略，每次都选择当前状态下的最优解，逐步构建出全局最优的最小生成树。

实现步骤拆解

1. **起点选取**：首先，我们可以任意选取图中的一个节点作为起始点。比如在一个包含节点 A、B、C、D 的图中，我们可以选择节点 A 作为起点，将其加入到最小生成树（MST）的顶点集合中。这一步就像是在搭建积木时，先确定第一块积木的位置。

2. **数据结构准备**：建立一个最小堆（优先队列）或者数组来存储当前最小生成树与未加入节点之间的最小边权。若使用数组，我们需要初始化数组，将除起点外的其他节点到最小生成树的距离设为无穷大（在 Java 中可以用`Integer.MAX_VALUE`表示）。例如，对于上述图，若以 A 为起点，那么我们初始化数组，记录 B、C、D 到最小生成树（此时只有 A）的距离为无穷大。如果使用最小堆，我们将起点的所有邻接边加入最小堆，堆中的元素按照边权值从小到大排序。

3. **节点扩展**：每次从最小堆（或数组）中选取最小权边。假设当前最小堆中最小权边连接的是节点 A 和节点 B，且 B 未在最小生成树中，那么我们将节点 B 加入到最小生成树中。然后，对于节点 B 的所有邻接边，检查这些边连接的节点是否在最小生成树中，如果不在，并且这些边的权值小于之前记录的该节点到最小生成树的距离，就更新距离数组（或最小堆）。比如，B 有邻接边连接 C 和 D，若边 BC 的权值小于之前记录的 C 到最小生成树的距离，就更新 C 到最小生成树的距离为边 BC 的权值，并将边 BC 加入最小堆（若使用最小堆）。

4. **结束条件**：重复上述步骤，直到所有节点都被加入到最小生成树中。因为一个具有 n 个顶点的连通图的最小生成树包含 n - 1 条边，所以当我们已经加入了 n - 1 条边时，也意味着所有节点都已加入，此时算法结束。

Java 代码实现

```java
import java.util.PriorityQueue;

class Edge implements Comparable<Edge> {
    int to;
    int weight;

    public Edge(int to, int weight) {
        this.to = to;
        this.weight = weight;
    }

    @Override
    public int compareTo(Edge other) {
        return this.weight - other.weight;
    }
}

class Graph {
    int vertexCount;
    PriorityQueue<Edge>[] adj;

    public Graph(int vertexCount) {
        this.vertexCount = vertexCount;
        adj = new PriorityQueue[vertexCount];
        for (int i = 0; i < vertexCount; i++) {
            adj[i] = new PriorityQueue<>();
        }
    }

    public void addEdge(int from, int to, int weight) {
        adj[from].add(new Edge(to, weight));
        adj[to].add(new Edge(from, weight));
    }
}

public class Prim {
    public void primMST(Graph graph) {
        int[] key = new int[graph.vertexCount];
        boolean[] mstSet = new boolean[graph.vertexCount];
        int[] parent = new int[graph.vertexCount];

        for (int i = 0; i < graph.vertexCount; i++) {
            key[i] = Integer.MAX_VALUE;
            mstSet[i] = false;
            parent[i] = -1;
        }

        key[0] = 0;
        PriorityQueue<Edge> pq = new PriorityQueue<>();
        pq.add(new Edge(0, 0));

        while (!pq.isEmpty()) {
```

```
            Edge current = pq.poll();
            int u = current.to;
            mstSet[u] = true;

            for (Edge edge : graph.adj[u]) {
                int v = edge.to;
                int weight = edge.weight;
                if (!mstSet[v] && weight < key[v]) {
                    key[v] = weight;
                    parent[v] = u;
                    pq.add(new Edge(v, weight));
                }
            }
        }

        System.out.println("最小生成树的边：");
        for (int i = 1; i < graph.vertexCount; i++) {
            System.out.println(parent[i] + " -- " + i + "  权重：" + key[i]);
        }
    }

    public static void main(String[] args) {
        Graph graph = new Graph(4);
        graph.addEdge(0, 1, 10);
        graph.addEdge(0, 2, 6);
        graph.addEdge(0, 3, 5);
        graph.addEdge(1, 3, 15);
        graph.addEdge(2, 3, 4);

        Prim prim = new Prim();
        prim.primMST(graph);
    }
}
```

复杂度分析

1. **使用邻接矩阵**：在每次迭代中，需要遍历所有顶点来找到最小权边，每次查找的时间复杂度为 O (V)，总共需要进行 V - 1 次迭代，所以时间复杂度为 O (V^2)，其中 V 是图中顶点的数量。这种方式适合稠密图，因为在稠密图中，边的数量接近 V^2，邻接矩阵可以快速地访问任意两个顶点之间的边权。

2. **使用最小堆 + 邻接表**：初始化最小堆的时间复杂度为 O (E)，因为需要将所有边加入堆中。每次从堆中取出最小边的操作时间复杂度为 O (log E)，需要执行 V - 1 次。对于每个新加入的顶点，更新堆中邻接边的操作最多为 E 次（因为每个顶点最多有 E 条邻接边），每次更新操作时间复杂度为 O (log E)。由于在连通图中，E 和 V 存在关系 E >= V - 1，所以时间复杂度可以近似表示为 O (E log V)。这种方式适合稀疏图，因为在稀疏图中，边的数量远小于 V^2，使用邻接表存储图可以节省空间，并且最小堆可以高效地找到最小权边。

3. **总复杂度**：对于稠密图，使用邻接矩阵的 Prim 算法总复杂度为 O (V^2)；对于稀疏图，使用最小堆和邻接表优化的 Prim 算法总复杂度为 O (E log V)。

特点概括

1. **每次扩展最近节点**：Prim 算法的每一步都是选择离当前最小生成树最近的节点加入，这种方式使得最小生成树能够以一种局部最优的方式逐步扩展，最终得到全局最优的最小生成树。

2. **增量式**：与 Kruskal 算法不同，Kruskal 算法是先对所有边进行排序，然后全局地选择权值最小的边来构建最小生成树；而 Prim 算法是从一个节点开始，逐步增量式地扩展最小生成树，每一步都基于当前已构建的最小生成树来选择下一个节点和边 。

考试易错点梳理

1. **起点可以任意**：在考试中，容易忘记 Prim 算法的起点可以是图中的任意一个节点，而错误地认为必须从特定节点开始。实际上，无论选择哪个节点作为起点，最终都能得到相同权值的最小生成树（虽然树的形态可能不同）。

2. **必须更新最小边权**：在加入新节点后，一定要对新节点的邻接边进行检查和更新最小边权。如果忘记更新，可能会导致最终得到的不是最小生成树，因为没有考虑到新加入节点可能带来的更优连接方式。

3. **注意稠密图和稀疏图复杂度不同**：在分析时间复杂度时，要根据图的类型（稠密图或稀疏图）正确选择对应的复杂度分析方式。如果在考试中不区分情况，错误地使用复杂度公式，可能会导致丢分。比如，在稀疏图中使用邻接矩阵的 Prim 算法复杂度分析，就会得到错误的结果。

Dijkstra 算法：寻找单源最短路径

核心思想阐释

Dijkstra 算法是一种用于求解图中单个源点到其他所有点的最短路径的经典算法，适用于非负权图。其核心思想融合了贪心策略和松弛操作。贪心策略体现在每次都选择当前距离源点最近的未访问节点进行扩展，就像在地图上规划路线时，总是优先选择离当前位置最近的下一个目的地 。而松弛操作则是该算法的关键步骤，它通过不断尝试更新节点到源点的距离，来找到真正的最短路径。例如，假设有节点 A、B、C，已知从源点到 A 的距离为 5，A 到 B 的距离为 3，若之前记录从源点到 B 的距离为 10，通过松弛操作，发现从源点经过 A 到 B 的距离为 5 + 3 = 8，小于 10，就更新从源点到 B 的距离为 8。这种局部最优的选择和不断优化的过程，使得算法能够逐步构建出从源点到所有节点的最短路径。

实现步骤详述

1. **初始化操作**：首先，创建两个重要的数组。距离数组`dist[]`，用于存储源点到每个节点的最短距离，初始时将源点的距离设为 0，其余节点的距离设为无穷大（在 Java 中可以用`Integer.MAX_VALUE`表示）。例如，在一个包含节点 A、B、C 的图中，若以 A 为源点，那么`dist[A] = 0`，`dist[B] = Integer.MAX_VALUE`，`dist[C] = Integer.MAX_VALUE`。标记数组`visited[]`，用于记录每个节点是否已经被访问过，初始时所有节点都标记为未访问，即`visited[i] = false`，其中`i`表示各个节点。

2. **节点选取与松弛**：在每一步中，从所有未访问的节点中选取距离最小的节点`u`。例如，在上述图中，若当前未访问节点 B 和 C 的距离分别为 10 和 15，那么就选取节点 B。然后，对节点`u`的所有邻居节点`v`进行松弛操作。计算从源点经过节点`u`到达邻居节点`v`的距离`dist[u] + w(u, v)`（其中`w(u, v)`表示节点`u`到`v`的边权），如果这个距离小于当前记录的`dist[v]`，则更新`dist[v]`为`dist[u] + w(u, v)`。比如，节点 B 的邻居节点是 C，B 到 C 的边权为 5，若当前`dist[C]`为 15，而`dist[B] + 5 = 10 + 5 = 15`，不小于`dist[C]`，则不更新；若`dist[C]`为 20，则更新`dist[C]`为 15。最后，将节点`u`标记为已访问，即`visited[u] = true`。

3. **结束条件**：重复上述选取节点和松弛操作的步骤，直到所有节点都被访问完。因为当所有节点都被访问时，意味着从源点到所有节点的最短路径都已经确定，此时算法结束，`dist[]`数组中存储的就是从源点

到各个节点的最短路径长度。

Java 代码实现

```
import java.util.PriorityQueue;

class Node implements Comparable<Node> {
    int vertex;
    int distance;

    public Node(int vertex, int distance) {
        this.vertex = vertex;
        this.distance = distance;
    }

    @Override
    public int compareTo(Node other) {
        return this.distance - other.distance;
    }
}

class Graph {
    int vertexCount;
    PriorityQueue<Node>[] adj;

    public Graph(int vertexCount) {
        this.vertexCount = vertexCount;
        adj = new PriorityQueue[vertexCount];
        for (int i = 0; i < vertexCount; i++) {
            adj[i] = new PriorityQueue<>();
        }
    }

    public void addEdge(int from, int to, int weight) {
        adj[from].add(new Node(to, weight));
    }
}

public class Dijkstra {
    public int[] dijkstra(Graph graph, int source) {
        int[] dist = new int[graph.vertexCount];
        boolean[] visited = new boolean[graph.vertexCount];

        for (int i = 0; i < graph.vertexCount; i++) {
            dist[i] = Integer.MAX_VALUE;
            visited[i] = false;
        }

        dist[source] = 0;
        PriorityQueue<Node> pq = new PriorityQueue<>();
        pq.add(new Node(source, 0));
```

```
        while (!pq.isEmpty()) {
            Node current = pq.poll();
            int u = current.vertex;
            visited[u] = true;

            for (Node neighbor : graph.adj[u]) {
                int v = neighbor.vertex;
                int weight = neighbor.distance;
                if (!visited[v] && dist[u] != Integer.MAX_VALUE && dist[u] +
weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.add(new Node(v, dist[v]));
                }
            }
        }

        return dist;
    }

    public static void main(String[] args) {
        Graph graph = new Graph(4);
        graph.addEdge(0, 1, 10);
        graph.addEdge(0, 2, 6);
        graph.addEdge(0, 3, 5);
        graph.addEdge(1, 3, 15);
        graph.addEdge(2, 3, 4);

        Dijkstra dijkstra = new Dijkstra();
        int[] distances = dijkstra.dijkstra(graph, 0);

        System.out.println("从源点0到其他节点的最短距离：");
        for (int i = 0; i < distances.length; i++) {
            System.out.println("到节点 " + i + " 的距离： " + distances[i]);
        }
    }
}
```

复杂度分析

1. **使用数组**：在每次迭代中，需要遍历所有未访问的节点来找到距离最小的节点，这一步的时间复杂度为 O (V)，其中 V 是图中顶点的数量。总共需要进行 V - 1 次迭代，所以总的时间复杂度为 O (V^2)。这种方式适用于稠密图，因为在稠密图中，边的数量接近 V^2，使用数组存储和操作距离信息相对简单直接。

2. **使用优先队列（堆）**：初始化优先队列时，将源点及其邻接节点加入队列，时间复杂度为 O (E)（因为需要遍历源点的所有邻接边）。每次从优先队列中取出最小距离节点的操作时间复杂度为 O (log V)，总共需要进行 V 次这样的操作；对于每个节点，更新其邻接节点距离时，由于每条边最多被更新一次，所以更新操作的时间复杂度为 O (E log V)。因此，使用优先队列优化后的时间复杂度为 O ((V + E) log V)。这种方式适合稀疏图，因为在稀疏图中，边的数量远小于 V^2，优先队列可以高效地管理距离信息，减少查找最小距离节点的时间开销。

3. **总复杂度**：对于稠密图，使用数组实现的 Dijkstra 算法总复杂度为 O (V^2)；对于稀疏图，使用优先队列优化的 Dijkstra 算法总复杂度为 O ((V + E) log V)。

特点归纳

1. **只能处理非负权图**：Dijkstra 算法的一个重要限制是只能处理边权非负的图。这是因为其贪心策略基于当前距离最小的节点进行扩展，如果存在负权边，可能会导致之前认为的最短路径在经过负权边后变得更短，从而破坏了算法的正确性。例如，在一个图中，从节点 A 到 B 的路径原本通过节点 C，距离为 5，若存在一条从 B 到 C 的负权边，权值为 - 3，那么从 A 直接到 B 再到 C 的路径距离就变为 5 - 3 = 2，小于之前认为的最短路径，这与 Dijkstra 算法的假设和操作方式相冲突。

2. **松弛操作是核心**：松弛操作是 Dijkstra 算法的核心步骤，通过不断尝试更新节点到源点的距离，使得算法能够逐步逼近并最终找到最短路径。每一次松弛操作都可能发现更优的路径，这种逐步优化的过程是算法能够正确求解最短路径的关键 。

考试易错点提示

1. **忘记松弛**：在实现 Dijkstra 算法时，忘记对邻居节点进行松弛操作是一个常见的错误。如果不进行松弛操作，就无法更新节点到源点的距离，也就无法找到真正的最短路径，导致算法结果错误。

2. **忘记标记已访问**：忘记将访问过的节点标记为已访问，会导致同一个节点被多次处理，不仅增加了计算量，还可能导致错误的结果。因为算法依赖于对已访问节点的判断来确定是否继续扩展和更新距离。

3. **图中有负权边**：如果在考试中遇到的图包含负权边，直接使用 Dijkstra 算法会得到错误的结果。此时需要注意题目条件，若图中有负权边，应使用专门处理负权边的算法，如 Bellman - Ford 算法，而不是 Dijkstra 算法。