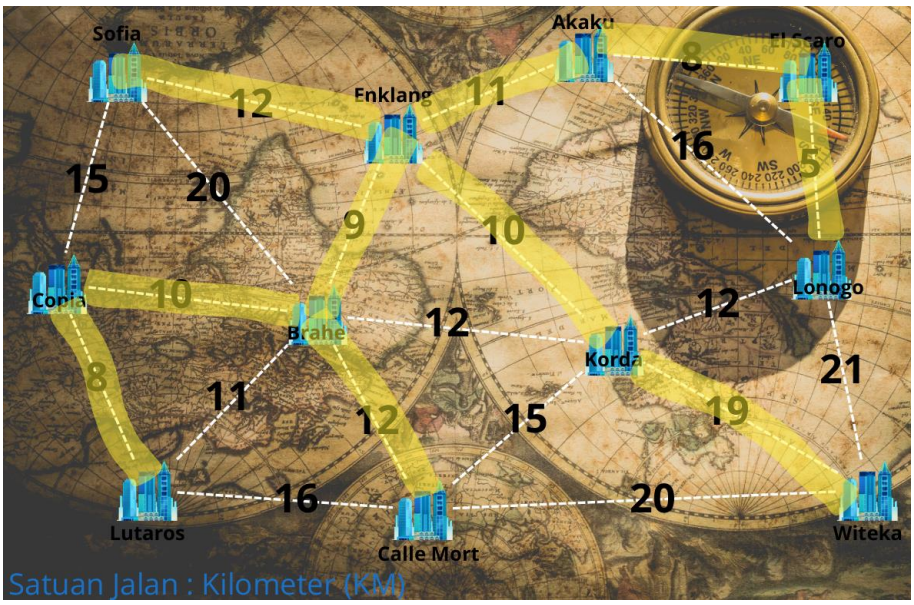
MODUL V

GRAPH

5.1 PERMASALAHAN

5.1.1 EASY LEVEL

Pak ilmah adalah seorang bupati di kabupaten Jogot, dalam masa jabatannya beliau ingin membangun jalan tol untuk kota kota yang yang ada di kabupaten tersebut, tapi karena anggaran kabupaten terbatas pak ilmah membuat strategi dimana daripada membuat jalan baru pak ilmah bisa meng-upgrade jalan jalan yang sudah ada, dan kemudian pak ilmah akan memilih jalan jalan tertentu yang dimana setidaknya seluruh kota mendapat jalan tol yang menhubungkannya dengan biaya paling minimum



Buatkan sebuah kodingan yang dimana semua kota dan jalan di gabungkan dalam sebuah graph, kemudian bantulah pak ilmah untuk menentukan jalan mana saja harus di ubah menjadi jalan tol sehingga total jarak jalan tol yang di bangun adalah paling sedikit dengan ketentuan setidak semua kota di lewatin jalan tol.

```
Korda ke Witeka : 19
El Scaro ke Lonogo : 5
Sofia ke Enklang : 12
Copia ke Lutaros : 8
Enklang ke Brahe : 9
Brahe ke Calle Mort : 12
Akaku ke El Scaro : 8
Enklang ke Korda : 10
Dengan Total :83
```

Urutan jalan bebas

5.2 HASIL PERCOBAAN

1. Algoritma
    a. Start program.
    b. Mempersiapkan kelas dan variabel, Membuat kelas Node untuk menyimpan informasi entitas graf, termasuk atribut nama (value), daftar edge yang terhubung (edgeList), parent node untuk Union-Find (parent), dan rank untuk optimisasi Union-Find. Membuat kelas Edge untuk menyimpan informasi edge dalam graf, termasuk node asal (from), node tujuan (to), bobot edge (weight), serta pointer ke edge berikutnya dalam linked list (nextEdge). Membuat kelas Graph untuk mengelola struktur graf dengan atribut utama berupa linked list dari node-node yang ada.
    c. Menambahkan node ke dalam graf menggunakan metode addNode pada kelas Graph untuk menambahkan node baru. Setiap node baru akan disisipkan di awal linked list yang merepresentasikan graf. Node ini nantinya dapat dihubungkan dengan node lain melalui edge.
    d. Menambahkan edge ke dalam graf menggunakan metode `addEdge` untuk menambahkan edge satu arah dengan bobot tertentu. Jika edge dua arah (undirected) diperlukan, metode `addUndirectedEdge` dipanggil untuk menambahkan edge ke dua arah. Setiap edge akan ditambahkan ke daftar edge milik node asal.
    e. Mengumpulkan semua edge, iterasi melalui setiap node di graf untuk mengumpulkan semua edge yang terhubung ke dalam linked list global allEdges. Setiap edge disalin ke linked list ini untuk diproses lebih lanjut.
    f. Mengurutkan edge berdasarkan bobot, memanggil metode `sortEdges` untuk mengurutkan linked list edge berdasarkan bobot (ascending). Pengurutan dilakukan menggunakan algoritma bubble sort sederhana.
    g. Menyiapkan struktur Union-Find menggunakan metode `find` untuk mendapatkan representasi set suatu node dengan path compression untuk mengoptimalkan pencarian. Menggunakan metode `union` untuk menggabungkan dua set berdasarkan rank, memastikan efisiensi proses penggabungan.
    h. Menjalankan algoritma Kruskal untuk MST, iterasi melalui edge yang telah diurutkan. Untuk setiap edge, periksa apakah dua node yang dihubungkan berada dalam set yang sama menggunakan metode `find`. Jika tidak, tambahkan edge tersebut ke MST dan gabungkan kedua set menggunakan metode `union`.

i. Menampilkan MST, setelah semua edge yang valid ditambahkan ke MST, daftar edge yang termasuk dalam MST dicetak beserta bobotnya. Selain itu, total bobot MST juga ditampilkan sebagai hasil akhir.

j. Selesai program.

2. *Source Code*

```
class Node {
    String value;
    Edge edgeList = null;
    Node nextNode = null;
    Node parent;
    int rank;
    public Node(String value) {
        this.value = value;
        this.parent = this;
        this.rank = 0;
    }
}

class Edge {
    Node from;
    Node to;
    int weight;
    Edge nextEdge = null;

    public Edge(Node from, Node to, int weight) {
        this.from = from;
        this.to = to;
        this.weight = weight;
    }
}
class Graph {
    Node head = null;

    public void addNode(String value) {
        Node newNode = new Node(value);
        newNode.nextNode = head;
        head = newNode;
    }
    public void addEdge(String from, String to, int weight) {
        Node fromNode = findNode(from);
        Node toNode = findNode(to);

        if (fromNode == null || toNode == null) {
            System.out.println("Node tidak ditemukan!");
            return;
        }
        Edge newEdge = new Edge(fromNode, toNode, weight);
        newEdge.nextEdge = fromNode.edgeList;
        fromNode.edgeList = newEdge;
    }
     public void addUndirectedEdge(String node1, String node2, int
weight) {
        addEdge(node1, node2, weight);
        addEdge(node2, node1, weight);
    }
    private Node findNode(String value) {
        Node current = head;
```

```
        while (current != null) {
            if (current.value == value) {
                return current;
            }
            current = current.nextNode;
        }
        return null;
    }
    private Node find(Node node) {
        if (node.parent != node) {
            node.parent = find(node.parent);
        }
        return node.parent;
    }
    private void union(Node node1, Node node2) {
        Node root1 = find(node1);
        Node root2 = find(node2);

        if (root1 != root2) {
            if (root1.rank > root2.rank) {
                root2.parent = root1;
            } else if (root1.rank < root2.rank) {
                root1.parent = root2;
            } else {
                root2.parent = root1;
                root1.rank++;
            }
        }
    }
    public Edge sortEdges(Edge edges) {
        if (edges == null || edges.nextEdge == null) {
            return edges;
        }

        boolean swapped;
        do {
            swapped = false;
            Edge current = edges;
            while (current != null && current.nextEdge != null) {
                if (current.weight > current.nextEdge.weight) {
                    int tempWeight = current.weight;
                    Node tempFrom = current.from;
                    Node tempTo = current.to;
                    current.weight = current.nextEdge.weight;
                    current.from = current.nextEdge.from;
                    current.to = current.nextEdge.to;
                    current.nextEdge.weight = tempWeight;
                    current.nextEdge.from = tempFrom;
                    current.nextEdge.to = tempTo;
                    swapped = true;
                }
                current = current.nextEdge;
            }
        } while (swapped);

        return edges;
    }

    public void kruskalMST() {
        Edge allEdges = null;
        Node currentNode = head;
```

```
        while (currentNode != null) {
            Edge currentEdge = currentNode.edgeList;
            while (currentEdge != null) {
                        Edge  newEdge  =  new  Edge(currentEdge.from,
currentEdge.to, currentEdge.weight);
                newEdge.nextEdge = allEdges;
                allEdges = newEdge;
                currentEdge = currentEdge.nextEdge;
            }
            currentNode = currentNode.nextNode;
        }

        allEdges = sortEdges(allEdges);

        Edge mstEdges = null;
        Edge currentEdge = allEdges;
        int totalWeight = 0;
        while (currentEdge != null) {
            Node fromNode = currentEdge.from;
            Node toNode = currentEdge.to;

            if (find(fromNode) != find(toNode)) {
                union(fromNode, toNode);
                totalWeight += currentEdge.weight;
                      Edge  mstEdge  =  new  Edge(fromNode,  toNode,
currentEdge.weight);
                mstEdge.nextEdge = mstEdges;
                mstEdges = mstEdge;
            }

            currentEdge = currentEdge.nextEdge;
        }

        System.out.println("Edges in the Minimum Spanning Tree:");
        currentEdge = mstEdges;
        while (currentEdge != null) {
              System.out.println(currentEdge.from.value  +  "  -  "  +
currentEdge.to.value + " - " + currentEdge.weight );
            currentEdge = currentEdge.nextEdge;
        }
        System.out.println(" Dengan Total: " + totalWeight);
    }
}

public class Main {
    public static void main(String[] args) {
        Graph graph = new Graph();

        // Menambahkan node
        graph.addNode("Sofia");
        graph.addNode("Enklang");
        graph.addNode("Akaku");
        graph.addNode("El Scaro");
        graph.addNode("Copia");
        graph.addNode("Brahe");
        graph.addNode("Korda");
        graph.addNode("Lonogo");
        graph.addNode("Lutaros");
        graph.addNode("Calle Mort");
        graph.addNode("Witeka");
```

```
        // Menambahkan edge dengan bobot
        graph.addUndirectedEdge("Sofia", "Brahe", 20);
        graph.addUndirectedEdge("Sofia", "Copia", 15);
        graph.addUndirectedEdge("Sofia", "Enklang", 12);

        graph.addUndirectedEdge("Enklang", "Brahe", 9);
        graph.addUndirectedEdge("Enklang", "Korda", 10);
        graph.addUndirectedEdge("Enklang", "Akaku", 11);

        graph.addUndirectedEdge("Akaku", "El Scaro", 8);
        graph.addUndirectedEdge("Akaku", "Lonogo", 16);
        // graph.addUndirectedEdge("Akaku", "El Scaro", 8);

        graph.addUndirectedEdge("El Scaro", "Lonogo", 5);

        graph.addUndirectedEdge("Lonogo", "Witeka", 21);
        graph.addUndirectedEdge("Lonogo", "Korda", 12);

        graph.addUndirectedEdge("Korda", "Witeka", 19);
        graph.addUndirectedEdge("Korda", "Calle Mort", 15);
        graph.addUndirectedEdge("Korda", "Brahe", 12);

        graph.addUndirectedEdge("Brahe", "Calle Mort", 12);
        graph.addUndirectedEdge("Brahe", "Korda", 12);
        graph.addUndirectedEdge("Brahe", "Lutaros", 11);
        graph.addUndirectedEdge("Brahe", "Copia", 10);

        graph.addUndirectedEdge("Lutaros", "Calle Mort", 16);
        graph.addUndirectedEdge("Witeka", "Calle Mort", 20);

        graph.addUndirectedEdge("Copia", "Lutaros", 8);

        // Menjalankan Kruskal untuk MST
        graph.kruskalMST();
    }
}
```

3. Hasil Program

```
Korda - Witeka - 19
Brahe - Calle Mort - 12
Sofia - Enklang - 12
Enklang - Akaku - 11
Copia - Brahe - 10
Enklang - Korda - 10
Enklang - Brahe - 9
Copia - Lutaros - 8
Akaku - El Scaro - 8
El Scaro - Lonogo - 5
Dengan Total    : 104
```

Gambar 3.1 Hasil *Run* 1

3.1 ANALISIS DATA

CLASS NODE

```
class Node {
    String value;
    Edge edgeList = null;
    Node nextNode = null;
    Node parent;
    int rank;

    public Node(String value) {
        this.value = value;
        this.parent = this;
        this.rank = 0;
    }
}
```

Kelas Node di atas adalah representasi dari simpul dalam sebuah graf, yang memiliki beberapa atribut untuk mendukung operasi graf dan algoritma seperti Union-Find. Atribut utama dari Node meliputi value, yang menyimpan nilai unik sebagai identitas dari simpul tersebut. Atribut edgeList adalah pointer ke daftar *linked list* yang merepresentasikan semua *edge* (sisi) yang terhubung ke simpul ini. Atribut nextNode digunakan untuk menunjuk ke simpul berikutnya dalam graf, memungkinkan graf direpresentasikan sebagai *linked list* dari simpul. Selain itu, kelas ini juga mendukung operasi Union-Find dengan atribut parent, yang pada awalnya menunjuk ke dirinya sendiri sebagai indikator bahwa simpul adalah pemimpin dari set-nya sendiri, dan atribut rank, yang digunakan untuk mengelola efisiensi penggabungan set dalam Union-Find. Konstruktor kelas ini menginisialisasi atribut-atribut tersebut, memastikan setiap simpul baru siap digunakan dalam operasi graf.

CLASS EDGE

```
class Edge {
    Node from;
    Node to;
    int weight;
    Edge nextEdge = null;

    public Edge(Node from, Node to, int weight) {
        this.from = from;
        this.to = to;
```

```
        this.weight = weight;
    }
}
```

Kelas Edge merupakan struktur yang digunakan untuk merepresentasikan sisi (edge) dalam sebuah graf berbobot. Setiap objek Edge memiliki atribut from, yaitu referensi ke simpul asal (Node) dari sisi tersebut, to sebagai referensi ke simpul tujuan (Node), dan weight yang menunjukkan bobot atau nilai yang terkait dengan sisi tersebut. Selain itu, atribut nextEdge digunakan sebagai pointer ke objek Edge berikutnya dalam daftar linked list, mendukung representasi graf berbasis daftar ketetanggaan. Konstruktor pada kelas ini menginisialisasi atribut from, to, dan weight berdasarkan parameter yang diberikan, sehingga setiap sisi dapat direpresentasikan secara lengkap dalam graf.

CLASS GRAPH

```
public class Graph {
    Node head = null;

    public void addNode(String value) {
        Node newNode = new Node(value);
        newNode.nextNode = head;
        head = newNode;
    }
```

Kode di atas merupakan bagian dari kelas Graph yang merepresentasikan graf sebagai kumpulan simpul menggunakan struktur *linked list*. Atribut head digunakan untuk menyimpan referensi ke simpul pertama dalam graf, memungkinkan traversal dari satu simpul ke simpul berikutnya melalui atribut nextNode dari setiap simpul. Pada saat sebuah objek Graph diinisialisasi melalui konstruktornya, atribut head diatur ke null, menunjukkan bahwa graf masih kosong tanpa simpul apa pun.

Metode addNode bertanggung jawab untuk menambahkan simpul baru ke graf. Ketika metode ini dipanggil dengan sebuah nilai, sebuah objek Node baru dibuat menggunakan nilai tersebut. Simpul baru ini kemudian ditambahkan ke awal daftar *linked list* dengan cara mengatur atribut nextNode dari simpul baru untuk menunjuk ke simpul yang saat ini berada di posisi head. Setelah itu, atribut head diperbarui untuk menunjuk ke simpul baru, menjadikannya simpul pertama dalam graf. Proses ini memungkinkan graf untuk menambah simpul secara efisien di awal daftar.

```
public void addEdge(String from, String to, int weight) {
        Node fromNode = findNode(from);
```

```
        Node toNode = findNode(to);

        if (fromNode == null || toNode == null) {
            System.out.println("Node tidak ditemukan!");
            return;
        }

        Edge newEdge = new Edge(fromNode, toNode, weight);
        newEdge.nextEdge = fromNode.edgeList;
        fromNode.edgeList = newEdge;
    }
```

Metode addEdge pada kode di atas berfungsi untuk menambahkan sebuah sisi (*edge*) dengan bobot tertentu antara dua simpul (*node*) dalam graf. Metode ini pertama-tama mencari simpul asal (fromNode) dan simpul tujuan (toNode) menggunakan metode findNode. Jika salah satu dari simpul tersebut tidak ditemukan dalam graf, pesan "Node tidak ditemukan!" akan ditampilkan, dan proses penambahan sisi dihentikan dengan menggunakan perintah return. Jika kedua simpul ditemukan, sebuah objek Edge baru dibuat untuk merepresentasikan sisi yang menghubungkan simpul asal ke simpul tujuan, beserta nilai bobotnya. Objek Edge baru ini kemudian ditambahkan ke awal daftar *linked list* dari sisi-sisi yang terhubung ke simpul asal (fromNode.edgeList). Dengan mengatur atribut nextEdge dari sisi baru untuk menunjuk ke daftar sisi yang sudah ada, metode ini memastikan bahwa semua sisi yang terhubung ke simpul asal tetap terhubung, sementara sisi baru menjadi elemen pertama dalam daftar. Pendekatan ini membuat proses penambahan sisi menjadi efisien dan terintegrasi dengan representasi graf berbasis daftar ketetanggaan.

```
public void addUndirectedEdge(String node1, String node2, int weight) {
        addEdge(node1, node2, weight);
        addEdge(node2, node1, weight);
    }

    private Node findNode(String value) {
        Node current = head;
        while (current != null) {
            if (current.value == value) {
                return current;
            }
            current = current.nextNode;
        }
        return null;
    }
```

Metode addUndirectedEdge digunakan untuk menambahkan sisi (*edge*) tidak berarah antara dua simpul dalam graf. Untuk melakukannya, metode ini memanfaatkan dua

pemanggilan metode addEdge, pertama untuk menambahkan sisi dari node1 ke node2, dan kedua untuk menambahkan sisi dari node2 ke node1. Dengan demikian, hubungan dua arah antara kedua simpul direpresentasikan sebagai dua sisi terpisah dalam graf, menciptakan hubungan simetris yang merepresentasikan sisi tidak berarah dengan bobot yang sama.

Metode findNode berfungsi untuk mencari dan mengembalikan referensi simpul dalam graf berdasarkan nilai tertentu. Metode ini menggunakan iterasi linear melalui daftar *linked list* simpul-simpul dalam graf, dimulai dari simpul pertama yang direferensikan oleh atribut head. Jika simpul dengan nilai yang dicari ditemukan, metode akan mengembalikan objek Node tersebut. Jika tidak, metode ini akan terus melanjutkan pencarian hingga mencapai akhir daftar, dan jika simpul dengan nilai yang dicari tetap tidak ditemukan, metode akan mengembalikan null. Pendekatan ini memungkinkan operasi pencarian simpul yang sederhana namun efektif dalam struktur graf berbasis *linked list*.

```
private Node find(Node node) {
        if (node.parent != node) {
            node.parent = find(node.parent);
        }
        return node.parent;
    }

     private void union(Node node1, Node node2) {
        Node root1 = find(node1);
        Node root2 = find(node2);

        if (root1 != root2) {
            // Union berdasarkan rank
            if (root1.rank > root2.rank) {
                root2.parent = root1;
            } else if (root1.rank < root2.rank) {
                root1.parent = root2;
            } else {
                root2.parent = root1;
                root1.rank++;
            }
        }
    }
```

Metode find dan union di atas adalah implementasi dari struktur Union-Find yang digunakan untuk mengelola himpunan-himpunan yang saling terpisah (*disjoint sets*). Metode find bertugas untuk menemukan representasi (*root parent*) dari himpunan tempat suatu simpul berada. Metode ini menggunakan teknik *path compression*, di mana setiap simpul yang dilewati selama pencarian langsung diperbarui untuk menunjuk ke simpul akar (*root*).

Teknik ini meningkatkan efisiensi operasi pencarian dengan meratakan struktur pohon, sehingga pencarian di masa depan menjadi lebih cepat.

Metode union bertugas untuk menggabungkan dua himpunan yang berbeda menjadi satu. Metode ini dimulai dengan menemukan *root parent* dari kedua simpul yang diberikan menggunakan metode find. Jika kedua simpul berasal dari himpunan yang sama (memiliki *root parent* yang sama), penggabungan tidak dilakukan karena keduanya sudah terhubung. Jika *root parent* berbeda, penggabungan dilakukan berdasarkan atribut rank untuk menjaga pohon tetap seimbang. Simpul dengan rank lebih tinggi menjadi *parent* dari simpul dengan rank lebih rendah. Jika kedua simpul memiliki rank yang sama, salah satu simpul dipilih sebagai *parent*, dan rank dari *root parent* baru ini dinaikkan satu. Pendekatan ini memastikan bahwa pohon tetap memiliki kedalaman yang minimum, meningkatkan efisiensi operasi di masa depan.

```
public Edge sortEdges(Edge edges) {
        if (edges == null || edges.nextEdge == null) {
            return edges;
        }
        boolean swapped;
        do {
            swapped = false;
            Edge current = edges;
            while (current != null && current.nextEdge != null) {
                if (current.weight > current.nextEdge.weight) {
                    // Tukar bobot dan node
                    int tempWeight = current.weight;
                    Node tempFrom = current.from;
                    Node tempTo = current.to;
                    current.weight = current.nextEdge.weight;
                    current.from = current.nextEdge.from;
                    current.to = current.nextEdge.to;
                    current.nextEdge.weight = tempWeight;
                    current.nextEdge.from = tempFrom;
                    current.nextEdge.to = tempTo;
                    swapped = true;
                }
                current = current.nextEdge;
            }
        } while (swapped);
        return edges;
    }
```

Metode sortEdges bertujuan untuk mengurutkan daftar sisi (*edges*) berdasarkan bobotnya menggunakan algoritma pengurutan *bubble sort*. Jika daftar sisi kosong atau hanya

memiliki satu elemen (yaitu edges adalah null atau hanya memiliki satu sisi), metode ini langsung mengembalikan daftar sisi tanpa melakukan perubahan.

Jika daftar sisi memiliki lebih dari satu elemen, algoritma *bubble sort* diterapkan untuk membandingkan setiap sisi yang berurutan. Selama iterasi, sisi yang memiliki bobot lebih besar dibandingkan sisi berikutnya akan ditukar posisinya. Proses pertukaran ini melibatkan pertukaran bobot (weight), simpul asal (from), dan simpul tujuan (to) untuk kedua sisi yang dibandingkan. Proses ini terus diulang hingga tidak ada lagi pertukaran yang terjadi, yang menandakan bahwa daftar sisi sudah terurut dengan benar. Metode ini mengembalikan daftar sisi yang telah diurutkan berdasarkan bobotnya. Algoritma *bubble sort* digunakan di sini meskipun bukan yang paling efisien untuk jumlah data besar, tetapi mudah diimplementasikan pada struktur data seperti ini.

```
public void kruskalMST() {
        Edge allEdges = null;
        Node currentNode = head;

        while (currentNode != null) {
            Edge currentEdge = currentNode.edgeList;
            while (currentEdge != null) {
               Edge newEdge = new Edge(currentEdge.from, currentEdge.to,
currentEdge.weight);
                newEdge.nextEdge = allEdges;
                allEdges = newEdge;
                currentEdge = currentEdge.nextEdge;
            }
            currentNode = currentNode.nextNode;
        }
```

Metode kruskalMST dimulai dengan menginisialisasi variabel allEdges yang akan menampung semua sisi yang ada dalam graf. Proses ini dilakukan dengan menggunakan *linked list* dari sisi-sisi yang terhubung ke setiap simpul dalam graf. Dimulai dari simpul pertama yang direferensikan oleh atribut head, metode ini menelusuri semua simpul dalam graf. Untuk setiap simpul, metode mengakses daftar sisi yang terhubung ke simpul tersebut melalui atribut edgeList. Setiap sisi yang ditemukan kemudian disalin ke dalam *linked list* baru (allEdges), yang memungkinkan pengumpulan semua sisi dalam satu daftar.

Untuk setiap sisi, sebuah objek Edge baru dibuat untuk memastikan salinan dari sisi asli dibuat, dan sisi tersebut ditambahkan ke awal daftar allEdges dengan cara menghubungkan sisi yang baru ke sisi sebelumnya. Proses ini dilakukan secara berurutan untuk setiap simpul dan sisi yang ada dalam graf hingga semua sisi dari semua simpul dimasukkan ke dalam daftar allEdges. Dengan cara ini, seluruh graf direpresentasikan dalam satu daftar sisi yang

akan diproses lebih lanjut dalam algoritma Kruskal untuk mencari Minimum Spanning Tree (MST).

```
allEdges = sortEdges(allEdges);

         Edge mstEdges = null;
        Edge currentEdge = allEdges;
        int totalWeight = 0;
        while (currentEdge != null) {
            Node fromNode = currentEdge.from;
            Node toNode = currentEdge.to;

            if (find(fromNode) != find(toNode)) {
                union(fromNode, toNode);
                totalWeight += currentEdge.weight;
                        Edge mstEdge = new Edge(fromNode, toNode,
currentEdge.weight);
                mstEdge.nextEdge = mstEdges;
                mstEdges = mstEdge;
            }

            currentEdge = currentEdge.nextEdge;
        }
```

Metode kruskalMST dimulai dengan mengurutkan semua sisi dari graf berdasarkan bobotnya menggunakan metode sortEdges. Hal ini memastikan bahwa sisi dengan bobot terkecil diproses terlebih dahulu. Setelah itu, sebuah variabel mstEdges dibuat untuk menyimpan sisi-sisi yang akan membentuk *Minimum Spanning Tree* (MST). Proses ini dimulai dengan mengiterasi melalui setiap sisi yang sudah diurutkan. Untuk setiap sisi, algoritma memeriksa apakah simpul asal dan simpul tujuan dari sisi tersebut sudah tergabung dalam satu himpunan yang sama menggunakan metode find. Jika belum, artinya sisi tersebut menghubungkan dua himpunan yang berbeda, sehingga sisi itu dapat ditambahkan ke MST.

Setelah menambahkan sisi tersebut, kedua himpunan yang terhubung oleh sisi ini digabungkan menggunakan metode union, untuk memastikan mereka tetap terhubung dalam MST dan tidak membentuk siklus. Bobot dari sisi yang ditambahkan juga ditambahkan ke total bobot MST. Setiap sisi yang ditambahkan ke MST dimasukkan ke dalam daftar mstEdges, dengan sisi terbaru menjadi elemen pertama dalam daftar tersebut. Proses ini terus berlanjut hingga semua sisi yang mungkin ditambahkan telah diproses, dan hasil akhirnya adalah sebuah MST yang menghubungkan semua simpul dengan bobot total yang minimal.

```
 System.out.println("Edges in the Minimum Spanning Tree:");
        currentEdge = mstEdges;
        while (currentEdge != null) {
```

```
                System.out.println(currentEdge.from.value + " ke " +
currentEdge.to.value + " : " + currentEdge.weight );
            currentEdge = currentEdge.nextEdge;
        }
        System.out.println("Total Weight of MST: " + totalWeight);
    }
}
```

Bagian kode ini bertanggung jawab untuk menampilkan hasil dari *Minimum Spanning Tree* (MST) yang telah dibangun oleh algoritma Kruskal. Setelah proses pembuatan MST selesai, kode ini mencetak setiap sisi yang ada dalam MST, dimulai dengan mencetak pesan "Edges in the Minimum Spanning Tree:". Kemudian, dengan menggunakan variabel currentEdge yang menunjuk pada sisi pertama dalam daftar mstEdges, kode ini mengiterasi melalui setiap sisi yang ada dalam MST. Untuk setiap sisi, informasi tentang simpul asal (from.value), simpul tujuan (to.value), dan bobot sisi (weight) ditampilkan di konsol dalam format yang mudah dibaca, misalnya "A ke B : 10".

Setelah mencetak semua sisi dalam MST, kode ini melanjutkan dengan mencetak total bobot dari MST, yang disimpan dalam variabel totalWeight. Dengan cara ini, pengguna dapat melihat semua sisi yang membentuk MST beserta total bobot yang diperlukan untuk menghubungkan semua simpul dalam graf tanpa membentuk siklus.

CLASS MAIN

```
public class Main {
    public static void main(String[] args) {
        Graph graph = new Graph();

        graph.addNode("Sofia");
        graph.addNode("Enklang");
        graph.addNode("Akaku");
        graph.addNode("El Scaro");
        graph.addNode("Copia");
        graph.addNode("Brahe");
        graph.addNode("Korda");
        graph.addNode("Lonogo");
        graph.addNode("Lutaros");
        graph.addNode("Calle Mort");
        graph.addNode("Witeka");

        graph.addUndirectedEdge("Sofia", "Brahe", 20);
        graph.addUndirectedEdge("Sofia", "Copia", 15);
        graph.addUndirectedEdge("Sofia", "Enklang", 12);

        graph.addUndirectedEdge("Enklang", "Brahe", 9);
```

```
        graph.addUndirectedEdge("Enklang", "Korda", 10);
        graph.addUndirectedEdge("Enklang", "Akaku", 11);

        graph.addUndirectedEdge("Akaku", "El Scaro", 8);
        graph.addUndirectedEdge("Akaku", "Lonogo", 16);

        graph.addUndirectedEdge("El Scaro", "Lonogo", 5);

        graph.addUndirectedEdge("Lonogo", "Witeka", 21);
        graph.addUndirectedEdge("Lonogo", "Korda", 12);

        graph.addUndirectedEdge("Korda", "Witeka", 19);
        graph.addUndirectedEdge("Korda", "Calle Mort", 15);
        graph.addUndirectedEdge("Korda", "Brahe", 12);

        graph.addUndirectedEdge("Brahe", "Calle Mort", 12);
        graph.addUndirectedEdge("Brahe", "Korda", 12);
        graph.addUndirectedEdge("Brahe", "Lutaros", 11);
        graph.addUndirectedEdge("Brahe", "Copia", 10);

        graph.addUndirectedEdge("Lutaros", "Calle Mort", 16);
        graph.addUndirectedEdge("Witeka", "Calle Mort", 20);

        graph.addUndirectedEdge("Copia", "Lutaros", 8);

        graph.kruskalMST();
    }
}
```

Kode di atas mendefinisikan kelas Main yang berfungsi untuk menjalankan program pembuatan *Minimum Spanning Tree* (MST) dengan menggunakan algoritma Kruskal pada graf yang dibangun. Pertama, objek Graph baru dibuat dengan nama graph, yang mewakili sebuah graf kosong. Kemudian, beberapa node (simpul) ditambahkan ke dalam graf menggunakan metode addNode, di mana setiap node diberi nama sebagai representasi dari kota atau lokasi, seperti "Sofia", "Enklang", "Akaku", dan seterusnya.

Selanjutnya, kode ini menambahkan beberapa edge (sisi) yang menghubungkan node-node tersebut dengan bobot tertentu menggunakan metode addUndirectedEdge. Metode ini menambahkan sisi dua arah antara dua node, yang berarti jika ada edge dari "Sofia" ke "Brahe" dengan bobot 20, maka ada juga edge dari "Brahe" ke "Sofia" dengan bobot yang sama. Proses ini dilakukan untuk seluruh node dan edge yang ada dalam graf untuk menciptakan hubungan antara berbagai kota.

Setelah graf terbentuk dengan sejumlah node dan edge, algoritma Kruskal dijalankan untuk menemukan MST dari graf tersebut melalui pemanggilan metode kruskalMST.

Metode ini mengurutkan semua edge berdasarkan bobotnya dan menggunakan teknik *union-find* untuk membangun MST dengan memilih edge-egde yang tidak membentuk siklus, sehingga menghubungkan semua node dengan total bobot yang minimal. Hasil akhirnya adalah MST yang dapat dilihat melalui pencetakan sisi-sisi yang ada dalam MST beserta total bobotnya di konsol.