Data Structures 2 - Bonus Lab

Shortest Paths Algorithms

Team :

Nada Mostafa Rashad (57)

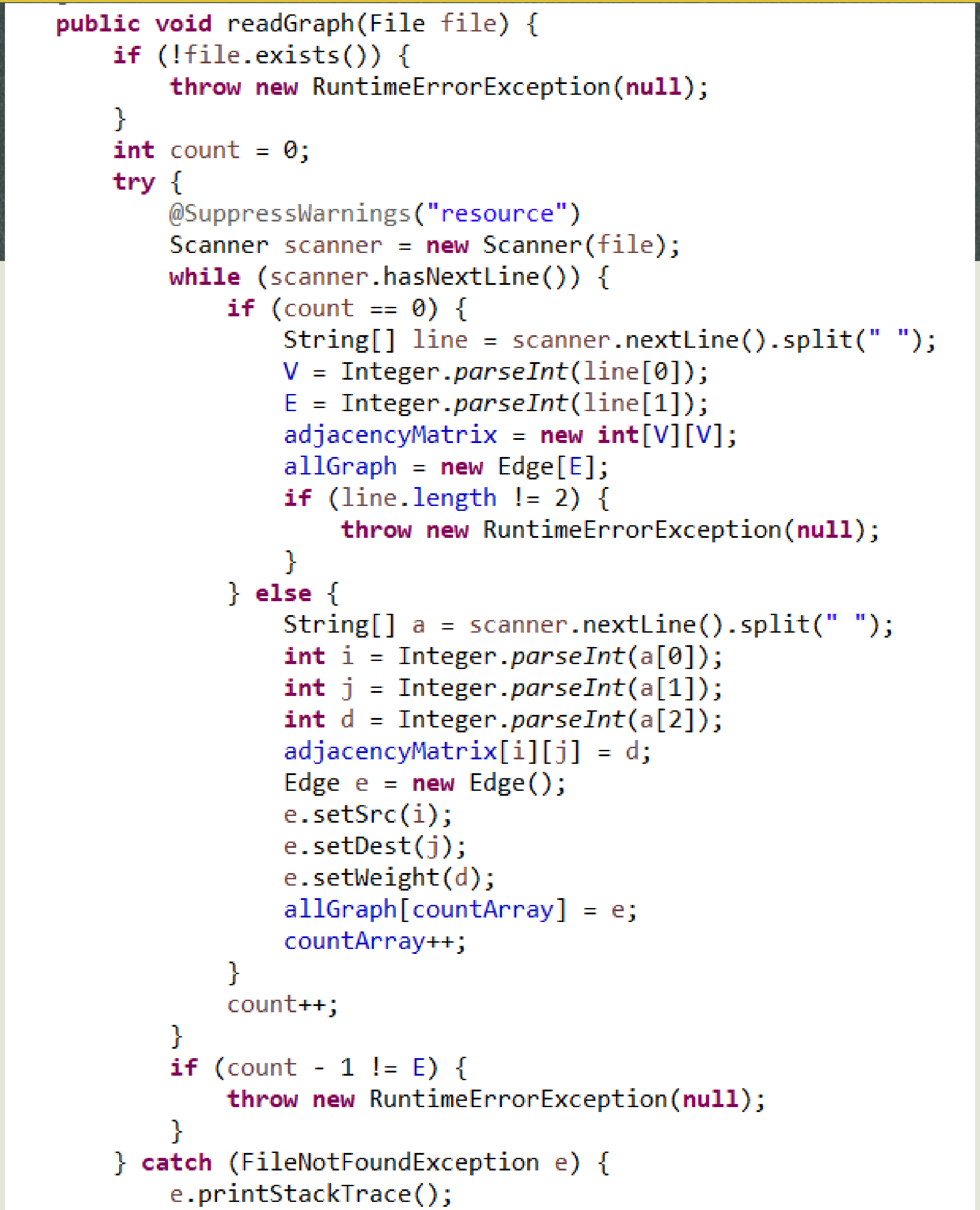
Nada Maher Mahmoud (56)

We implement two shortest path algorithms which are
-Dijkstra Algorithm
-Bellman-Ford Algorithm

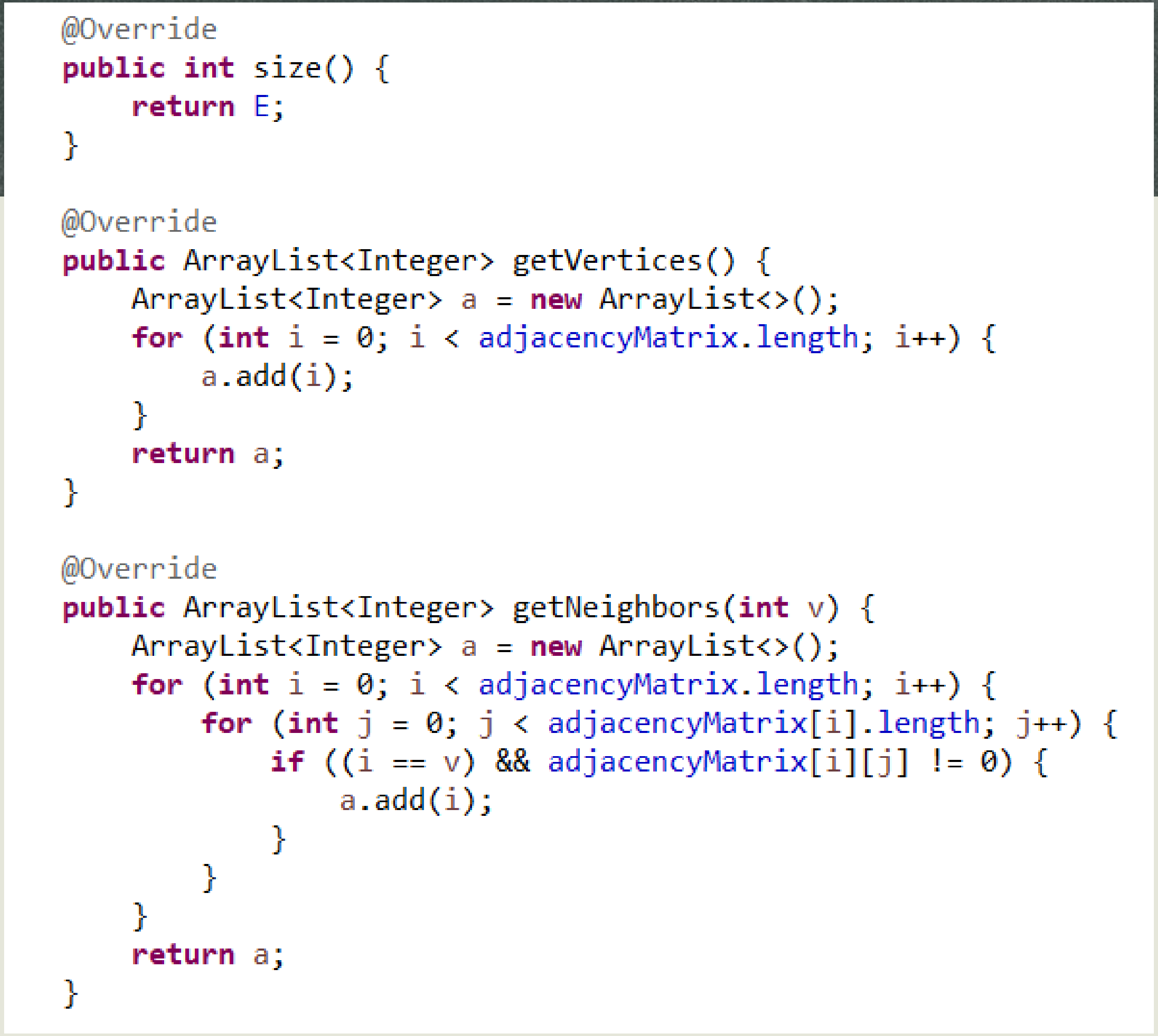
Used Classes

We formed a class called “Edge” to store the info of each entered edge; source, destination and weight.
Used Data structures:
A 2D integer array is used to form an adjacency matrix.
A 1D array of edges is used to store all edges.
A 1D integer array to store the distances.

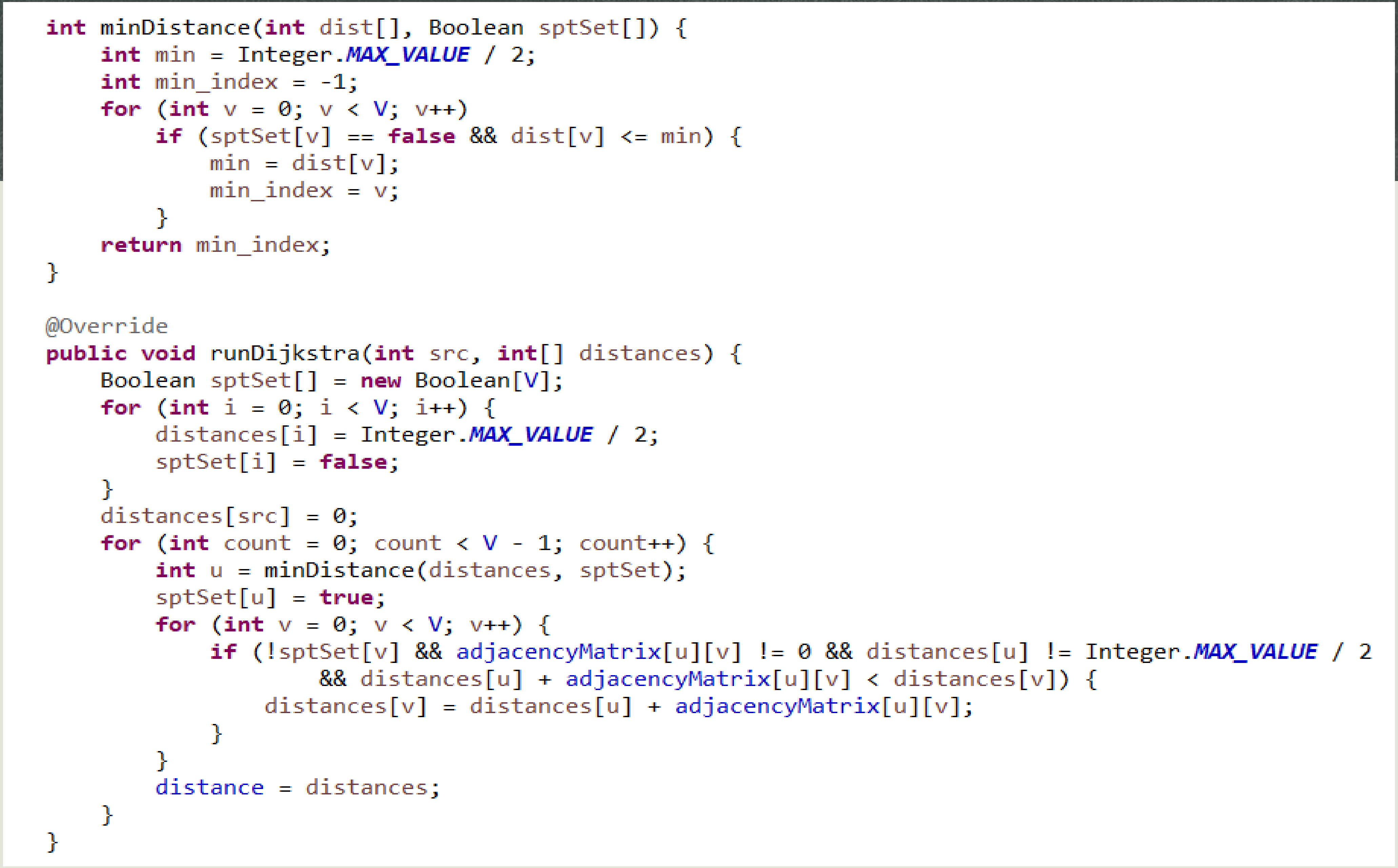
Code snapshot



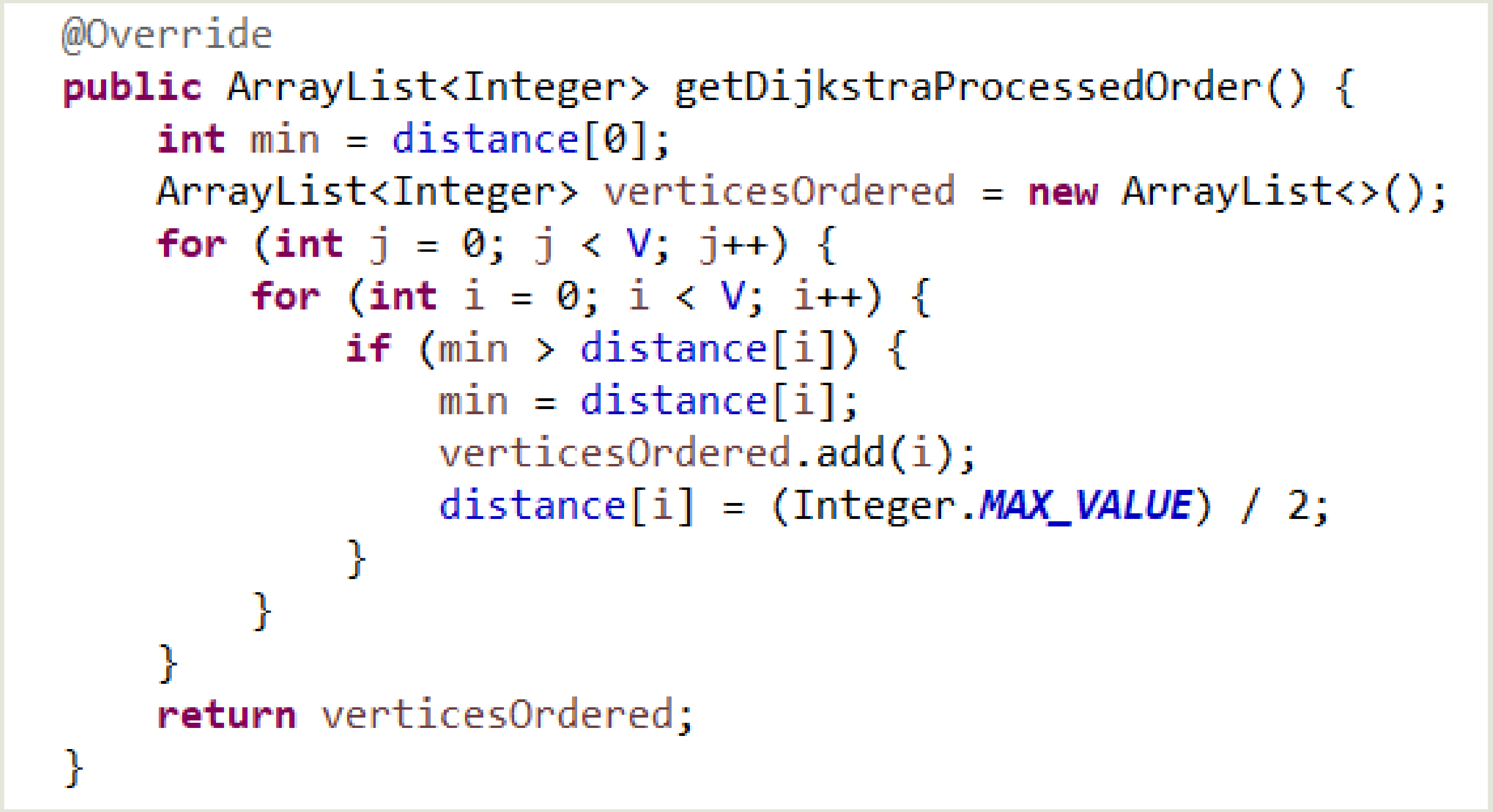
```java
public void readGraph(File file) {
    if (!file.exists()) {
        throw new RuntimeErrorException(null);
    }
    int count = 0;
    try {
        @SuppressWarnings("resource")
        Scanner scanner = new Scanner(file);
        while (scanner.hasNextLine()) {
            if (count == 0) {
                String[] line = scanner.nextLine().split(" ");
                V = Integer.parseInt(line[0]);
                E = Integer.parseInt(line[1]);
                adjacencyMatrix = new int[V][V];
                allGraph = new Edge[E];
                if (line.length != 2) {
                    throw new RuntimeErrorException(null);
                }
            } else {
                String[] a = scanner.nextLine().split(" ");
                int i = Integer.parseInt(a[0]);
                int j = Integer.parseInt(a[1]);
                int d = Integer.parseInt(a[2]);
                adjacencyMatrix[i][j] = d;
                Edge e = new Edge();
                e.setSrc(i);
                e.setDest(j);
                e.setWeight(d);
                allGraph[countArray] = e;
                countArray++;
            }
            count++;
        }
        if (count - 1 != E) {
            throw new RuntimeErrorException(null);
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
```



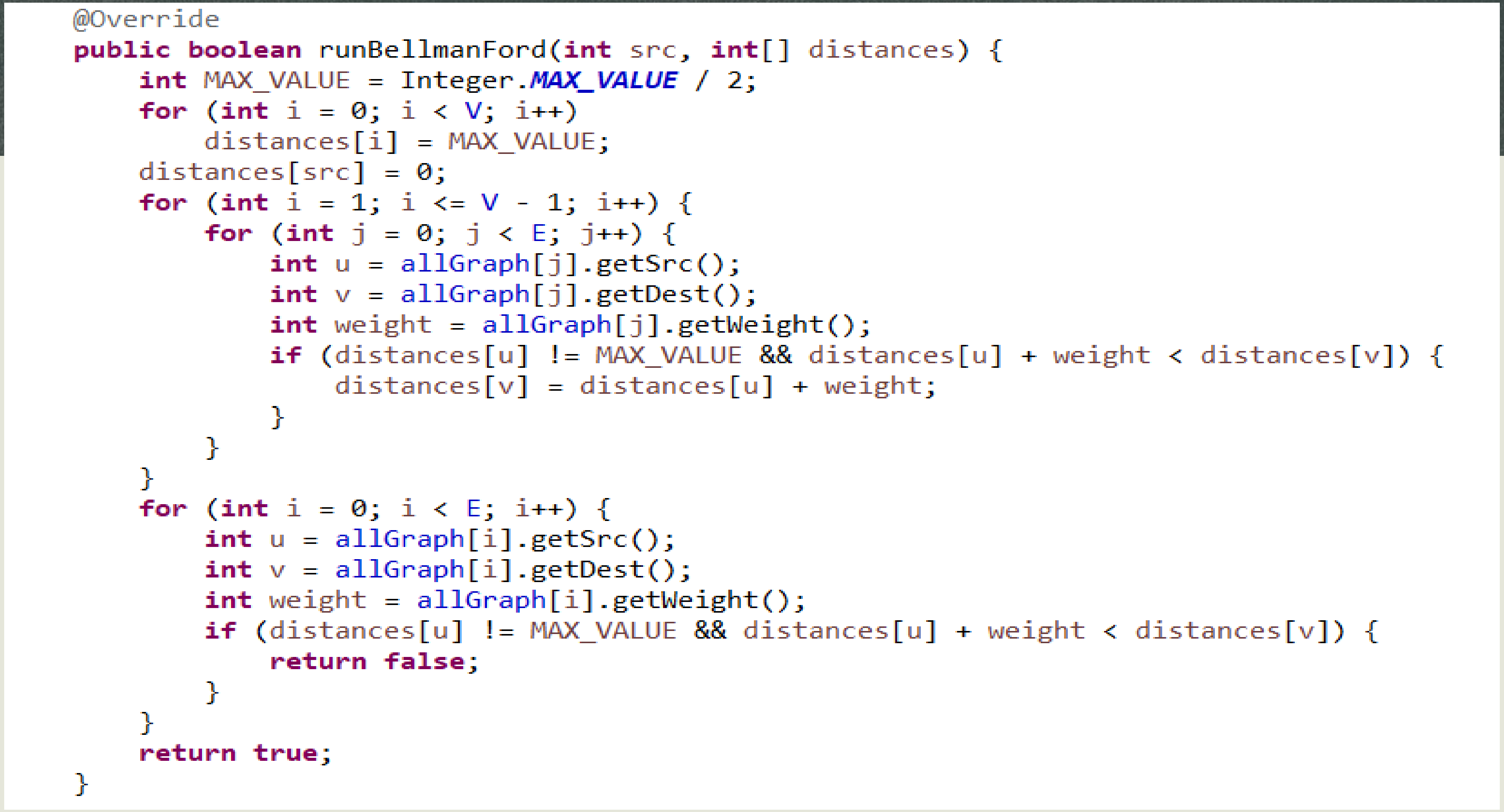
```java
@Override
public int size() {
    return E;
}

@Override
public ArrayList<Integer> getVertices() {
    ArrayList<Integer> a = new ArrayList<>();
    for (int i = 0; i < adjacencyMatrix.length; i++) {
        a.add(i);
    }
    return a;
}

@Override
public ArrayList<Integer> getNeighbors(int v) {
    ArrayList<Integer> a = new ArrayList<>();
    for (int i = 0; i < adjacencyMatrix.length; i++) {
        for (int j = 0; j < adjacencyMatrix[i].length; j++) {
            if ((i == v) && adjacencyMatrix[i][j] != 0) {
                a.add(i);
            }
        }
    }
    return a;
}
```



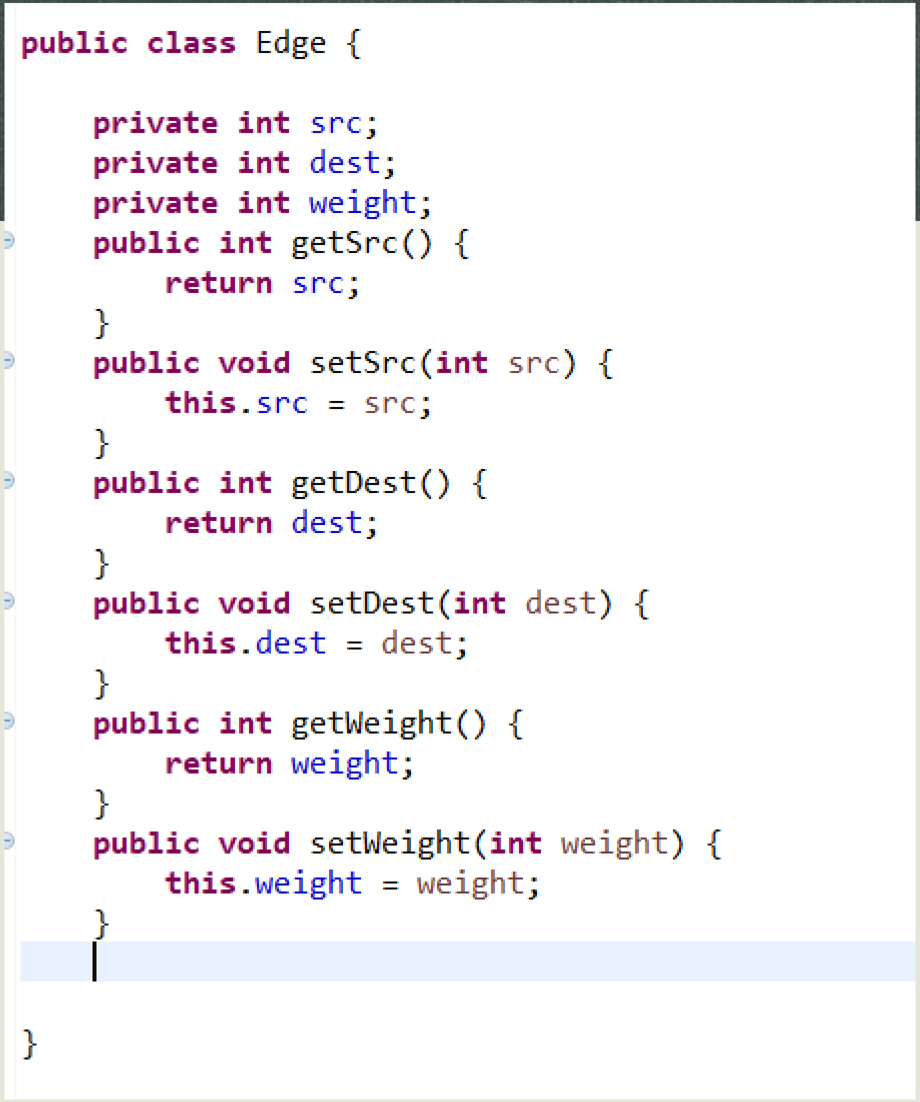
```java
int minDistance(int dist[], Boolean sptSet[]) {
    int min = Integer.MAX_VALUE / 2;
    int min_index = -1;
    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    return min_index;
}

@Override
public void runDijkstra(int src, int[] distances) {
    Boolean sptSet[] = new Boolean[V];
    for (int i = 0; i < V; i++) {
        distances[i] = Integer.MAX_VALUE / 2;
        sptSet[i] = false;
    }
    distances[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(distances, sptSet);
        sptSet[u] = true;
        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && adjacencyMatrix[u][v] != 0 && distances[u] != Integer.MAX_VALUE / 2
                    && distances[u] + adjacencyMatrix[u][v] < distances[v]) {
                distances[v] = distances[u] + adjacencyMatrix[u][v];
            }
        }
        distance = distances;
    }
}
```



```java
@Override
public ArrayList<Integer> getDijkstraProcessedOrder() {
    int min = distance[0];
    ArrayList<Integer> verticesOrdered = new ArrayList<>();
    for (int j = 0; j < V; j++) {
        for (int i = 0; i < V; i++) {
            if (min > distance[i]) {
                min = distance[i];
                verticesOrdered.add(i);
                distance[i] = (Integer.MAX_VALUE) / 2;
            }
        }
    }
    return verticesOrdered;
}
```



```java
    @Override
    public boolean runBellmanFord(int src, int[] distances) {
        int MAX_VALUE = Integer.MAX_VALUE / 2;
        for (int i = 0; i < V; i++)
            distances[i] = MAX_VALUE;
        distances[src] = 0;
        for (int i = 1; i <= V - 1; i++) {
            for (int j = 0; j < E; j++) {
                int u = allGraph[j].getSrc();
                int v = allGraph[j].getDest();
                int weight = allGraph[j].getWeight();
                if (distances[u] != MAX_VALUE && distances[u] + weight < distances[v]) {
                    distances[v] = distances[u] + weight;
                }
            }
        }
        for (int i = 0; i < E; i++) {
            int u = allGraph[i].getSrc();
            int v = allGraph[i].getDest();
            int weight = allGraph[i].getWeight();
            if (distances[u] != MAX_VALUE && distances[u] + weight < distances[v]) {
                return false;
            }
        }
        return true;
    }
```



```
public class Edge {

    private int src;
    private int dest;
    private int weight;
    public int getSrc() {
        return src;
    }
    public void setSrc(int src) {
        this.src = src;
    }
    public int getDest() {
        return dest;
    }
    public void setDest(int dest) {
        this.dest = dest;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        this.weight = weight;
    }


}
```