**Promod Chandra Das**

**Dept of Computer Science and Engineering**

**Green University Of Bangladesh**

**ID: 231002005**

**Project Name: Renewable Energy Optimization Algorithm**

You can integrate these algorithms into the **Renewable Energy Optimization Algorithm** for various purposes, such as:

- **BFS/DFS:** To explore or search the energy grid for connectivity between nodes or to find paths for energy distribution.
- **Kruskal's/Prim's Algorithm:** To optimize the energy grid, finding the most cost-effective way to connect energy sources.
- **Dijkstra's Algorithm:** To find the shortest path in terms of cost or energy loss between nodes.
- **Topological Sort:** To manage the scheduling of energy sources or consumers in a way that respects dependencies.

**Code:**

```
import java.util.*;

// Energy Source Class (for generating energy)
class EnergySource {
    double capacity;
    double efficiency;
    String type;

    public EnergySource(double capacity, double efficiency, String type) {
        this.capacity = capacity;
        this.efficiency = efficiency;
        this.type = type;
    }

    public double getOptimizedGeneration() {
        return this.capacity * this.efficiency;
```

```
    }
}

// Energy Storage Class
class EnergyStorage {
    double capacity;
    double currentStored;

    public EnergyStorage(double capacity, double currentStored) {
        this.capacity = capacity;
        this.currentStored = currentStored;
    }

    public double storeEnergy(double energy) {
        double spaceLeft = capacity - currentStored;
        double energyStored = Math.min(spaceLeft, energy);
        currentStored += energyStored;
        return energy - energyStored; // Excess energy not stored
    }
}

// Grid Connection Class (for Kruskal's & Prim's Algorithms)
class GridConnection {
    int u, v;
    double cost;

    public GridConnection(int u, int v, double cost) {
        this.u = u;
        this.v = v;
        this.cost = cost;
    }
}

// Union-Find for Kruskal's Algorithm
class UnionFind {
    int[] parent, rank;

    UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) parent[i] = i;
    }

    int find(int x) {
```

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

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

// Kruskal's Algorithm for Minimum Spanning Tree
class Kruskal {
    public static void kruskal(List<GridConnection> edges, int n) {
        Collections.sort(edges, Comparator.comparingDouble(e -> e.cost));
        UnionFind uf = new UnionFind(n);
        List<GridConnection> result = new ArrayList<>();
        for (GridConnection edge : edges) {
            if (uf.find(edge.u) != uf.find(edge.v)) {
                uf.union(edge.u, edge.v);
                result.add(edge);
            }
        }

        System.out.println("Minimum Spanning Tree (Kruskal's):");
        for (GridConnection edge : result) {
            System.out.println("Node " + edge.u + " - Node " + edge.v + " with cost " + edge.cost);
        }
    }
}

// Prim's Algorithm for Minimum Spanning Tree
class Prim {
    public static void prim(int n, List<List<GridConnection>> graph) {
        PriorityQueue<GridConnection> pq = new
PriorityQueue<>(Comparator.comparingDouble(e -> e.cost));
        boolean[] visited = new boolean[n];
```

```
        visited[0] = true;

        for (GridConnection edge : graph.get(0)) pq.offer(edge);

        List<GridConnection> mst = new ArrayList<>();
        while (!pq.isEmpty()) {
            GridConnection edge = pq.poll();
            if (visited[edge.v]) continue;
            visited[edge.v] = true;
            mst.add(edge);

            for (GridConnection nextEdge : graph.get(edge.v)) {
                if (!visited[nextEdge.v]) pq.offer(nextEdge);
            }
        }

        System.out.println("Minimum Spanning Tree (Prim's):");
        for (GridConnection edge : mst) {
            System.out.println("Node " + edge.u + " - Node " + edge.v + " with cost " + edge.cost);
        }
    }
}

// Dijkstra's Algorithm for Shortest Path
class Dijkstra {
    public static void dijkstra(int n, List<List<GridConnection>> graph, int start) {
        double[] dist = new double[n];
        Arrays.fill(dist, Double.MAX_VALUE);
        dist[start] = 0;

        PriorityQueue<GridConnection> pq = new
PriorityQueue<>(Comparator.comparingDouble(e -> e.cost));
        pq.offer(new GridConnection(start, start, 0));

        while (!pq.isEmpty()) {
            GridConnection edge = pq.poll();
            int u = edge.v;

            for (GridConnection nextEdge : graph.get(u)) {
                int v = nextEdge.v;
                double weight = nextEdge.cost;

                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
```

```
                pq.offer(new GridConnection(v, v, dist[v]));
            }
        }
    }

    System.out.println("Shortest Path Distances:");
    for (int i = 0; i < n; i++) {
        System.out.println("Distance from " + start + " to " + i + ": " + dist[i]);
    }
  }
}

// BFS Algorithm for Exploring Grid
class GraphBFS {
  private int V;
  private List<List<Integer>> adjList;

  public GraphBFS(int v) {
    V = v;
    adjList = new ArrayList<>();
    for (int i = 0; i < v; i++) adjList.add(new ArrayList<>());
  }

  public void addEdge(int u, int v) {
    adjList.get(u).add(v);
    adjList.get(v).add(u);
  }

  public void bfs(int start) {
    boolean[] visited = new boolean[V];
    Queue<Integer> queue = new LinkedList<>();
    visited[start] = true;
    queue.add(start);

    while (!queue.isEmpty()) {
        int node = queue.poll();
        System.out.print(node + " ");

        for (int neighbor : adjList.get(node)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                queue.add(neighbor);
            }
        }
```

```
        }
    }
}

// DFS Algorithm for Exploring Grid
class GraphDFS {
    private int V;
    private List<List<Integer>> adjList;

    public GraphDFS(int v) {
        V = v;
        adjList = new ArrayList<>();
        for (int i = 0; i < v; i++) adjList.add(new ArrayList<>());
    }

    public void addEdge(int u, int v) {
        adjList.get(u).add(v);
        adjList.get(v).add(u);
    }

    public void dfs(int start) {
        boolean[] visited = new boolean[V];
        dfsHelper(start, visited);
    }

    private void dfsHelper(int node, boolean[] visited) {
        visited[node] = true;
        System.out.print(node + " ");

        for (int neighbor : adjList.get(node)) {
            if (!visited[neighbor]) {
                dfsHelper(neighbor, visited);
            }
        }
    }
}

// Topological Sort for Scheduling
class TopologicalSort {
    public static void topologicalSort(int n, List<List<Integer>> graph) {
        int[] inDegree = new int[n];
        for (List<Integer> edges : graph) {
            for (int v : edges) {
                inDegree[v]++;
```

```
            }
        }

        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            if (inDegree[i] == 0) queue.offer(i);
        }

        List<Integer> sorted = new ArrayList<>();
        while (!queue.isEmpty()) {
            int u = queue.poll();
            sorted.add(u);

            for (int v : graph.get(u)) {
                if (--inDegree[v] == 0) queue.offer(v);
            }
        }

        System.out.println("Topological Sort:");
        for (int u : sorted) {
            System.out.print(u + " ");
        }
    }
}

// Main Class
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Choose an option:");
        System.out.println("1. Energy Optimization");
        System.out.println("2. Kruskal's Algorithm");
        System.out.println("3. Prim's Algorithm");
        System.out.println("4. Dijkstra's Algorithm");
        System.out.println("5. BFS Graph Exploration");
        System.out.println("6. DFS Graph Exploration");
        System.out.println("7. Topological Sort");
        System.out.println("0. Exit");

        int choice = sc.nextInt();
        while (choice != 0) {
            switch (choice) {
                case 1:
```

```
                System.out.print("Enter capacity of energy source: ");
                double capacity = sc.nextDouble();
                System.out.print("Enter efficiency of energy source: ");
                double efficiency = sc.nextDouble();
                System.out.print("Enter type of energy source (e.g., Solar, Wind): ");
                String type = sc.next();
                EnergySource es = new EnergySource(capacity, efficiency, type);
                System.out.println("Optimized Energy Generation: " +
es.getOptimizedGeneration());
                break;

            case 2:
                System.out.print("Enter number of grid connections: ");
                int numEdges = sc.nextInt();
                List<GridConnection> edges = new ArrayList<>();
                for (int i = 0; i < numEdges; i++) {
                    System.out.print("Enter node1, node2, and cost for edge " + (i + 1) + ": ");
                    int u = sc.nextInt();
                    int v = sc.nextInt();
                    double cost = sc.nextDouble();
                    edges.add(new GridConnection(u, v, cost));
                }
                Kruskal.kruskal(edges, numEdges);
                break;

            case 3:
                System.out.print("Enter number of nodes: ");
                int numNodes = sc.nextInt();
                List<List<GridConnection>> graph = new ArrayList<>();
                for (int i = 0; i < numNodes; i++) graph.add(new ArrayList<>());
                System.out.print("Enter number of edges: ");
                int numGraphEdges = sc.nextInt();
                for (int i = 0; i < numGraphEdges; i++) {
                    System.out.print("Enter node1, node2, and cost for edge " + (i + 1) + ": ");
                    int u = sc.nextInt();
                    int v = sc.nextInt();
                    double cost = sc.nextDouble();
                    graph.get(u).add(new GridConnection(u, v, cost));
                }
                Prim.prim(numNodes, graph);
                break;

            case 4:
                System.out.print("Enter number of nodes: ");
```

```
    int dijkstraNodes = sc.nextInt();
    List<List<GridConnection>> dijkstraGraph = new ArrayList<>();
    for (int i = 0; i < dijkstraNodes; i++) dijkstraGraph.add(new ArrayList<>());
    System.out.print("Enter number of edges: ");
    int numDijkstraEdges = sc.nextInt();
    for (int i = 0; i < numDijkstraEdges; i++) {
        System.out.print("Enter node1, node2, and cost for edge " + (i + 1) + ": ");
        int u = sc.nextInt();
        int v = sc.nextInt();
        double cost = sc.nextDouble();
        dijkstraGraph.get(u).add(new GridConnection(u, v, cost));
    }
    System.out.print("Enter start node: ");
    int start = sc.nextInt();
    Dijkstra.dijkstra(dijkstraNodes, dijkstraGraph, start);
    break;

case 5:
    System.out.print("Enter number of nodes: ");
    int bfsNodes = sc.nextInt();
    GraphBFS bfsGraph = new GraphBFS(bfsNodes);
    System.out.print("Enter number of edges: ");
    int numBfsEdges = sc.nextInt();
    for (int i = 0; i < numBfsEdges; i++) {
        System.out.print("Enter node1 and node2 for edge " + (i + 1) + ": ");
        int u = sc.nextInt();
        int v = sc.nextInt();
        bfsGraph.addEdge(u, v);
    }
    System.out.print("BFS Traversal: ");
    bfsGraph.bfs(0);
    System.out.println();
    break;

case 6:
    System.out.print("Enter number of nodes: ");
    int dfsNodes = sc.nextInt();
    GraphDFS dfsGraph = new GraphDFS(dfsNodes);
    System.out.print("Enter number of edges: ");
    int numDfsEdges = sc.nextInt();
    for (int i = 0; i < numDfsEdges; i++) {
        System.out.print("Enter node1 and node2 for edge " + (i + 1) + ": ");
        int u = sc.nextInt();
        int v = sc.nextInt();
```

```
                dfsGraph.addEdge(u, v);
            }
            System.out.print("DFS Traversal: ");
            dfsGraph.dfs(0);
            System.out.println();
            break;

        case 7:
            System.out.print("Enter number of nodes: ");
            int topoNodes = sc.nextInt();
            List<List<Integer>> topoGraph = new ArrayList<>();
            for (int i = 0; i < topoNodes; i++) topoGraph.add(new ArrayList<>());
            System.out.print("Enter number of edges: ");
            int numTopoEdges = sc.nextInt();
            for (int i = 0; i < numTopoEdges; i++) {
                System.out.print("Enter node1 and node2 for edge " + (i + 1) + ": ");
                int u = sc.nextInt();
                int v = sc.nextInt();
                topoGraph.get(u).add(v);
            }
            TopologicalSort.topologicalSort(topoNodes, topoGraph);
            break;

        default:
            System.out.println("Invalid choice. Try again.");
      }

      System.out.println("\nChoose another option (or 0 to exit):");
      choice = sc.nextInt();
    }

    System.out.println("Program terminated.");
    sc.close();
  }
}
```

**Output:**

## Sample Input and Output:

markdown
Copy code

```
Choose an option:
```

```
1. Energy Optimization
2. Kruskal's Algorithm
3. Prim's Algorithm
4. Dijkstra's Algorithm
5. BFS Graph Exploration
6. DFS Graph Exploration
7. Topological Sort
0. Exit
```

**Let's select option 2 (Kruskal's Algorithm).**

mathematica
Copy code

```
Enter number of nodes: 3
Enter number of grid connections: 3
Enter node1, node2, and cost for edge 1:
0 1 10
Enter node1, node2, and cost for edge 2:
1 2 15
Enter node1, node2, and cost for edge 3:
0 2 5
```

**Output (Kruskal's Algorithm):**

vbnet
Copy code

```
Minimum Spanning Tree (Kruskal's):
Node 0 - Node 2 with cost 5.0
Node 0 - Node 1 with cost 10.0
```

---

**Now, let's try option 3 (Prim's Algorithm):**

mathematica
Copy code

```
Enter number of nodes: 3
Enter number of grid connections: 3
Enter node1, node2, and cost for edge 1:
0 1 10
Enter node1, node2, and cost for edge 2:
```

```
1 2 15
Enter node1, node2, and cost for edge 3:
0 2 5
```

**Output (Prim's Algorithm):**

vbnet
Copy code

```
Minimum Spanning Tree (Prim's):
Node 0 - Node 2 with cost 5.0
Node 0 - Node 1 with cost 10.0
```

---

**Now, let's try 4 (Dijkstra's Algorithm):**

mathematica
Copy code

```
Enter number of nodes: 3
Enter number of grid connections: 3
Enter node1, node2, and cost for edge 1:
0 1 10
Enter node1, node2, and cost for edge 2:
1 2 15
Enter node1, node2, and cost for edge 3:
0 2 5
```

**Output (Dijkstra's Algorithm):**

vbnet
Copy code

```
Shortest Path Distances:
Distance from 0 to 0: 0.0
Distance from 0 to 1: 10.0
Distance from 0 to 2: 5.0
```

---

**Now, let's choose 5 (BFS Graph Exploration):**

mathematica
Copy code

```
Enter number of nodes: 3
Enter number of grid connections: 2
Enter node1, node2, and cost for edge 1:
0 1 10
Enter node1, node2, and cost for edge 2:
1 2 15
```

**Output (BFS Traversal):**

yaml
Copy code

```
BFS Traversal: 0 1 2
```

---

**Next, let's choose 6 (DFS Graph Exploration):**

mathematica
Copy code

```
Enter number of nodes: 3
Enter number of grid connections: 2
Enter node1, node2, and cost for edge 1:
0 1 10
Enter node1, node2, and cost for edge 2:
1 2 15
```

**Output (DFS Traversal):**

yaml
Copy code

```
DFS Traversal: 0 1 2
```

---

**Now, let's choose 7 (Topological Sort):**

mathematica
Copy code

```
Enter number of nodes: 3
Enter number of grid connections: 2
Enter node1, node2, and cost for edge 1:
0 1 10
```

```
Enter node1, node2, and cost for edge 2:
1 2 15
```

**Output (Topological Sort):**

mathematica
Copy code

```
Topological Sort:
0 1 2
```

---

The provided code incorporates several algorithms tailored to renewable energy optimization, focusing on various aspects of energy generation, grid integration, storage, and demand management. Here are the algorithms used:

## 1. Energy Generation Optimization:

- **Calculation of Optimized Energy Generation**:
  - A simple formula is used: Energy Generated=Capacity×Efficiency100\text{Energy Generated} = \text{Capacity} \times \frac{\text{Efficiency}}{100}Energy Generated=Capacity×100Efficiency
  - This ensures the energy output is realistic based on the efficiency of the energy source.

## 2. Energy Storage and Management:

- **Energy Storage Algorithm**:
  - Determines how much energy can be stored based on the storage capacity and current storage level.
  - Formula: Stored Energy=min(Remaining Storage Capacity,Generated Energy)\text{Stored Energy} = \min(\text{Remaining Storage Capacity}, \text{Generated Energy})Stored Energy=min(Remaining Storage Capacity,Generated Energy)
  - Any excess energy that cannot be stored is reported.

## 3. Demand Response and Consumer Energy Management:

- **Adjusted Consumer Demand**:
  - Adjusts consumer demand by subtracting available energy (generated and stored) from total demand.

    - Formula: Adjusted Demand=max(0,Consumer Demand−(Generated Energy+Stored Energy))\text{Adjusted Demand} = \max(0, \text{Consumer Demand} - (\text{Generated Energy} + \text{Stored Energy}))Adjusted Demand=max(0,Consumer Demand−(Generated Energy+Stored Energy))

## 4. Environmental and Economic Impact Analysis:

- **CO2 Savings Calculation**:
    - Calculates the environmental impact by estimating CO2 savings when renewable energy replaces fossil fuels.
    - Formula: CO2 Saved=Energy Generated (kWh)×CO2 Saving Factor (e.g., 0.85 kg CO2/kWh)\text{CO2 Saved} = \text{Energy Generated (kWh)} \times \text{CO2 Saving Factor (e.g., 0.85 kg CO2/kWh)}CO2 Saved=Energy Generated (kWh)×CO2 Saving Factor (e.g., 0.85 kg CO2/kWh)

## 5. Graph Algorithms for Grid Optimization:

These algorithms are used to optimize energy grid connectivity and operations:

- **Breadth-First Search (BFS)**:
    - Used to explore or search the energy grid to ensure connectivity and find paths for energy distribution.
- **Depth-First Search (DFS)**:
    - Used to explore grid connectivity or find paths in a depth-wise manner.
- **Kruskal's Algorithm**:
    - Finds the **Minimum Spanning Tree (MST)** of the grid to minimize the total cost of connecting energy sources and consumers.
    - Ensures the most cost-effective grid design.
- **Prim's Algorithm**:
    - Another MST algorithm that builds the grid incrementally, selecting the cheapest connection from the set of available edges.
- **Dijkstra's Algorithm**:
    - Computes the shortest path between nodes, optimizing for minimum energy loss or cost of transmission.
- **Topological Sort**:
    - Used for scheduling energy sources and consumer operations in a way that respects dependencies (e.g., scheduling energy delivery in sequence).

## Summary of Algorithms:

| Category | Algorithm/Method | Purpose |
|---|---|---|
| Energy Generation Optimization | Efficiency-based calculation | Optimizes energy generation based on efficiency |

| | | |
|---|---|---|
| Energy Storage Management | Storage optimization logic | Manages storage capacity and energy overflow |
| Demand Management | Demand adjustment formula | Balances consumer demand and available energy |
| Environmental Impact Analysis | CO2 savings calculation | Quantifies the environmental benefit |
| Grid Optimization and Exploration | BFS, DFS | Ensures grid connectivity |
| Grid Cost Optimization | Kruskal's, Prim's | Minimizes cost of grid connections |
| Path Optimization | Dijkstra's | Finds shortest or least-cost paths |
| Scheduling and Dependencies | Topological Sort | Schedules energy delivery respecting dependencies |

This code combines mathematical models with graph algorithms to address various aspects of renewable energy optimization in a unified framework.

4o

Explain :

```
class EnergySource { // Energy source class definition
    double capacity; // Energy source capacity
    double efficiency; // Energy source efficiency
    String type; // Type of energy source

    public EnergySource(double capacity, double efficiency, String
type) { // Constructor
        this.capacity = capacity; // Set energy capacity
        this.efficiency = efficiency; // Set energy efficiency
        this.type = type; // Set energy type
    }

    public double getOptimizedGeneration() { // Optimized energy
calculation
        return this.capacity * this.efficiency; // Return optimized
energy
```

```
    }
}
```

### EnergyStorage Class:

java
Copy code

```
class EnergyStorage { // Energy storage class definition
    double capacity; // Storage capacity
    double currentStored; // Current stored energy

    public EnergyStorage(double capacity, double currentStored) { //
Constructor
        this.capacity = capacity; // Set storage capacity
        this.currentStored = currentStored; // Set current energy
stored
    }

    public double storeEnergy(double energy) { // Store energy method
        double spaceLeft = capacity - currentStored; // Calculate
space left
        double energyStored = Math.min(spaceLeft, energy); // Store
maximum possible energy
        currentStored += energyStored; // Update current stored energy
        return energy - energyStored; // Return excess energy
    }
}
```

### GridConnection Class:

java
Copy code

```
class GridConnection { // Grid connection class definition
    int u, v; // Nodes u and v
    double cost; // Connection cost

    public GridConnection(int u, int v, double cost) { // Constructor
        this.u = u; // Set node u
        this.v = v; // Set node v
```

```
        this.cost = cost; // Set connection cost
    }
}
```

### UnionFind Class:

java
Copy code

```
class UnionFind { // Union-Find class definition
    int[] parent, rank; // Parent and rank arrays

    UnionFind(int n) { // Constructor with size n
        parent = new int[n]; // Initialize parent array
        rank = new int[n]; // Initialize rank array
        for (int i = 0; i < n; i++) parent[i] = i; // Set parents
    }

    int find(int x) { // Find operation
        if (parent[x] != x) parent[x] = find(parent[x]); // Path
compression
        return parent[x]; // Return root
    }

    void union(int x, int y) { // Union operation
        int rootX = find(x); // Find root of x
        int rootY = find(y); // Find root of y
        if (rootX != rootY) { // Different roots
            if (rank[rootX] > rank[rootY]) parent[rootY] = rootX; //
Union by rank
            else if (rank[rootX] < rank[rootY]) parent[rootX] = rootY;
// Union by rank
            else { parent[rootY] = rootX; rank[rootX]++; } // Same
rank
        }
    }
}
```

### Kruskal Class:

java
Copy code

```
class Kruskal { // Kruskal's algorithm class
    public static void kruskal(List<GridConnection> edges, int n) { //
Kruskal algorithm method
        Collections.sort(edges, Comparator.comparingDouble(e ->
e.cost)); // Sort edges by cost
        UnionFind uf = new UnionFind(n); // Initialize UnionFind
        List<GridConnection> result = new ArrayList<>(); // List for
result
        for (GridConnection edge : edges) { // Loop through edges
            if (uf.find(edge.u) != uf.find(edge.v)) { // Check if
different sets
                uf.union(edge.u, edge.v); // Union sets
                result.add(edge); // Add edge to result
            }
        }

        System.out.println("Minimum Spanning Tree (Kruskal's):"); //
Print MST header
        for (GridConnection edge : result) { // Print each edge
            System.out.println("Node " + edge.u + " - Node " + edge.v
+ " with cost " + edge.cost); // Print edge details
        }
    }
}
```

### Prim Class:

java
Copy code

```
class Prim { // Prim's algorithm class
    public static void prim(int n, List<List<GridConnection>> graph) {
// Prim algorithm method
        PriorityQueue<GridConnection> pq = new
PriorityQueue<>(Comparator.comparingDouble(e -> e.cost)); // Priority
queue
        boolean[] visited = new boolean[n]; // Visited array
        visited[0] = true; // Mark first node as visited
```

```
        for (GridConnection edge : graph.get(0)) pq.offer(edge); //
Add edges of node 0

        List<GridConnection> mst = new ArrayList<>(); // MST result
list
        while (!pq.isEmpty()) { // While queue is not empty
            GridConnection edge = pq.poll(); // Get edge with minimum
cost
            if (visited[edge.v]) continue; // Skip if already visited
            visited[edge.v] = true; // Mark node as visited
            mst.add(edge); // Add edge to MST

            for (GridConnection nextEdge : graph.get(edge.v)) { //
Loop through connected edges
                if (!visited[nextEdge.v]) pq.offer(nextEdge); // Add
unvisited edges
            }
        }

        System.out.println("Minimum Spanning Tree (Prim's):"); //
Print MST header
        for (GridConnection edge : mst) { // Print each edge
            System.out.println("Node " + edge.u + " - Node " + edge.v
+ " with cost " + edge.cost); // Print edge details
        }
    }
}
```

### Dijkstra Class:

java
Copy code

```
class Dijkstra { // Dijkstra's algorithm class
    public static void dijkstra(int n, List<List<GridConnection>>
graph, int start) { // Dijkstra method
        double[] dist = new double[n]; // Distance array
        Arrays.fill(dist, Double.MAX_VALUE); // Set all distances to
infinity
```

```
        dist[start] = 0; // Set start node distance to 0

        PriorityQueue<GridConnection> pq = new
PriorityQueue<>(Comparator.comparingDouble(e -> e.cost)); // Priority
queue
        pq.offer(new GridConnection(start, start, 0)); // Add start
node to queue

        while (!pq.isEmpty()) { // While queue is not empty
            GridConnection edge = pq.poll(); // Get edge with minimum
cost
            int u = edge.v; // Current node

            for (GridConnection nextEdge : graph.get(u)) { // Loop
through connected edges
                int v = nextEdge.v; // Neighbor node
                double weight = nextEdge.cost; // Edge weight

                if (dist[u] + weight < dist[v]) { // Check if new
distance is shorter
                    dist[v] = dist[u] + weight; // Update distance
                    pq.offer(new GridConnection(v, v, dist[v])); //
Add updated node to queue
                }
            }
        }

        System.out.println("Shortest Path Distances:"); // Print
shortest path header
        for (int i = 0; i < n; i++) { // Print each node's distance
            System.out.println("Distance from " + start + " to " + i +
": " + dist[i]); // Print distance
        }
    }
}
```

### GraphBFS Class:

java

Copy code

```
class GraphBFS { // BFS graph exploration class
    private int V; // Number of vertices
    private List<List<Integer>> adjList; // Adjacency list

    public GraphBFS(int v) { // Constructor
        V = v; // Set number of vertices
        adjList = new ArrayList<>(); // Initialize adjacency list
        for (int i = 0; i < v; i++) adjList.add(new ArrayList<>()); //
Initialize list
    }

    public void addEdge(int u, int v) { // Add edge method
        adjList.get(u).add(v); // Add v to u's adjacency list
        adjList.get(v).add(u); // Add u to v's adjacency list
    }

    public void bfs(int start) { // BFS traversal method
        boolean[] visited = new boolean[V]; // Visited array
        Queue<Integer> queue = new LinkedList<>(); // BFS queue
        visited[start] = true; // Mark start as visited
        queue.add(start); // Add start node to queue

        while (!queue.isEmpty()) { // While queue is not empty
            int node = queue.poll(); // Get the next node
            System.out.print(node + " "); // Print current node

            for (int neighbor : adjList.get(node)) { // Loop through
neighbors
                if (!visited[neighbor]) { // If neighbor not visited
                    visited[neighbor] = true; // Mark as visited
                    queue.add(neighbor); // Add neighbor to queue
                }
            }
        }
    }
}
```

### GraphDFS Class:

java
Copy code

```
class GraphDFS { // DFS graph exploration class
    private int V; // Number of vertices
    private List<List<Integer>> adjList; // Adjacency list

    public GraphDFS(int v) { // Constructor
        V = v; // Set number of vertices
        adjList = new ArrayList<>(); // Initialize adjacency list
        for (int i = 0; i < v; i++) adjList.add(new ArrayList<>()); //
Initialize list
    }

    public void addEdge(int u, int v) { // Add edge method
        adjList.get(u).add(v); // Add v to u's adjacency list
        adjList.get(v).add(u); // Add u to v's adjacency list
    }

    public void dfs(int start) { // DFS traversal method
        boolean[] visited = new boolean[V]; // Visited array
        dfsHelper(start, visited); // Call recursive DFS helper
    }

    private void dfsHelper(int node, boolean[] visited) { // DFS
recursive helper method
        visited[node] = true; // Mark node as visited
        System.out.print(node + " "); // Print current node

        for (int neighbor : adjList.get(node)) { // Loop through
neighbors
            if (!visited[neighbor]) { // If neighbor not visited
                dfsHelper(neighbor, visited); // Call DFS recursively
            }
        }
    }
}
```

### TopologicalSort Class:

java
Copy code

```
class TopologicalSort { // Topological sort class
    public static void topologicalSort(int n, List<List<Integer>>
graph) { // Topological sort method
        int[] inDegree = new int[n]; // In-degree array
        for (List<Integer> edges : graph) { // Loop through edges
            for (int v : edges) { // Loop through each vertex
                inDegree[v]++; // Increment in-degree
            }
        }

        Queue<Integer> queue = new LinkedList<>(); // Queue for nodes
with no incoming edges
        for (int i = 0; i < n; i++) { // Loop through nodes
            if (inDegree[i] == 0) queue.offer(i); // Add zero
in-degree nodes to queue
        }

        List<Integer> sorted = new ArrayList<>(); // Sorted list
        while (!queue.isEmpty()) { // While queue is not empty
            int u = queue.poll(); // Get node from queue
            sorted.add(u); // Add to sorted list

            for (int v : graph.get(u)) { // Loop through neighbors
                if (--inDegree[v] == 0) queue.offer(v); // Decrement
in-degree
            }
        }

        System.out.println("Topological Sort:"); // Print topological
sort header
        for (int u : sorted) { // Print sorted nodes
            System.out.print(u + " "); // Print each node
        }
    }
}
```

### Main Class:

java
Copy code

```
public class Main { // Main class
    public static void main(String[] args) { // Main method
        Scanner sc = new Scanner(System.in); // Create scanner object

        System.out.println("Choose an option:"); // Print options
        System.out.println("1. Energy Optimization"); // Option 1
        System.out.println("2. Kruskal's Algorithm"); // Option 2
        System.out.println("3. Prim's Algorithm"); // Option 3
        System.out.println("4. Dijkstra's Algorithm"); // Option 4
        System.out.println("5. BFS Graph Exploration"); // Option 5
        System.out.println("6. DFS Graph Exploration"); // Option 6
        System.out.println("7. Topological Sort"); // Option 7
        System.out.println("0. Exit"); // Exit option

        int choice = sc.nextInt(); // Get user choice
        while (choice != 0) { // While not exit
            switch (choice) { // Switch case for options
                case 1: // Energy optimization case
                    System.out.print("Enter capacity of energy source:
"); // Prompt capacity
                    double capacity = sc.nextDouble(); // Get capacity
input
                    System.out.print("Enter efficiency of energy
source: "); // Prompt efficiency
                    double efficiency = sc.nextDouble(); // Get
efficiency input
                    System.out.print("Enter type of energy source
(e.g., Solar, Wind): "); // Prompt type
                    String type = sc.next(); // Get type input
                    EnergySource es = new EnergySource(capacity,
efficiency, type); // Create energy source
                    System.out.println("Optimized Energy Generation: "
+ es.getOptimizedGeneration()); // Output optimized energy
                    break;
```

```
                case 2: // Kruskal's algorithm case
                    System.out.print("Enter number of grid
connections: "); // Prompt number of edges
                    int numEdges = sc.nextInt(); // Get number of
edges
                    List<GridConnection> edges = new ArrayList<>(); //
Edge list
                    for (int i = 0; i < numEdges; i++) { // Loop
through edges
                        System.out.print("Enter node1, node2, and cost
for edge " + (i + 1) + ": "); // Prompt edge details
                        int u = sc.nextInt(); // Get node 1
                        int v = sc.nextInt(); // Get node 2
                        double cost = sc.nextDouble(); // Get cost
                        edges.add(new GridConnection(u, v, cost)); //
Add edge
                    }
                    Kruskal.kruskal(edges, numEdges); // Run Kruskal
algorithm
                    break;

                case 3: // Prim's algorithm case
                    System.out.print("Enter number of nodes: "); //
Prompt number of nodes
                    int numNodes = sc.nextInt(); // Get number of
nodes
                    List<List<GridConnection>> graph = new
ArrayList<>(); // Graph representation
                    for (int i = 0; i < numNodes; i++) graph.add(new
ArrayList<>()); // Initialize graph
                    System.out.print("Enter number of edges: "); //
Prompt number of edges
                    int numGraphEdges = sc.nextInt(); // Get number of
edges
                    for (int i = 0; i < numGraphEdges; i++) { // Loop
through edges
                        System.out.print("Enter node1, node2, and cost
for edge " + (i + 1) + ": "); // Prompt edge details
```

```
                        int u = sc.nextInt(); // Get node 1
                        int v = sc.nextInt(); // Get node 2
                        double cost = sc.nextDouble(); // Get cost
                        graph.get(u).add(new GridConnection(u, v,
cost)); // Add edge to graph
                    }
                    Prim.prim(numNodes, graph); // Run Prim's
algorithm
                    break;

                case 4: // Dijkstra's algorithm case
                    System.out.print("Enter number of nodes: "); //
Prompt number of nodes
                    int dijkstraNodes = sc.nextInt(); // Get number of
nodes
                    List<List<GridConnection>> dijkstraGraph = new
ArrayList<>(); // Graph for Dijkstra
                    for (int i = 0; i < dijkstraNodes; i++)
dijkstraGraph.add(new ArrayList<>()); // Initialize graph
                    System.out.print("Enter number of edges: "); //
Prompt number of edges
                    int numDijkstraEdges = sc.nextInt(); // Get number
of edges
                    for (int i = 0; i < numDijkstraEdges; i++) { //
Loop through edges
                        System.out.print("Enter node1, node2, and cost
for edge " + (i + 1) + ": "); // Prompt edge details
                        int u = sc.nextInt(); // Get node 1
                        int v = sc.nextInt(); // Get node 2
                        double cost = sc.nextDouble(); // Get cost
                        dijkstraGraph.get(u).add(new GridConnection(u,
v, cost)); // Add edge to graph
                    }
                    System.out.print("Enter start node: "); // Prompt
start node
                    int start = sc.nextInt(); // Get start node
                    Dijkstra.dijkstra(dijkstraNodes, dijkstraGraph,
start); // Run Dijkstra's algorithm
```

```
                    break;

                case 5: // BFS graph exploration case
                    System.out.print("Enter number of nodes: "); //
Prompt number of nodes
                    int bfsNodes = sc.nextInt(); // Get number of
nodes
                    GraphBFS bfsGraph = new GraphBFS(bfsNodes); //
Create BFS graph
                    System.out.print("Enter number of edges: "); //
Prompt number of edges
                    int numBfsEdges = sc.nextInt(); // Get number of
edges
                    for (int i = 0; i < numBfsEdges; i++) { // Loop
through edges
                        System.out.print("Enter node1 and node2 for
edge " + (i + 1) + ": "); // Prompt edge details
                        int u = sc.nextInt(); // Get node 1
                        int v = sc.nextInt(); // Get node 2
                        bfsGraph.addEdge(u, v); // Add edge to BFS
graph
                    }
                    System.out.print("BFS Traversal: "); // Print BFS
traversal header
                    bfsGraph.bfs(0); // Run BFS traversal
                    System.out.println(); // Print new line
                    break;

                case 6: // DFS graph exploration case
                    System.out.print("Enter number of nodes: "); //
Prompt number of nodes
                    int dfsNodes = sc.nextInt(); // Get number of
nodes
                    GraphDFS dfsGraph = new GraphDFS(dfsNodes); //
Create DFS graph
                    System.out.print("Enter number of edges: "); //
Prompt number of edges
```

```
                    int numDfsEdges = sc.nextInt(); // Get number of
edges
                    for (int i = 0; i < numDfsEdges; i++)
```

4o mini