1. Algorithm

Task A: Adapt an Existing Algorithm

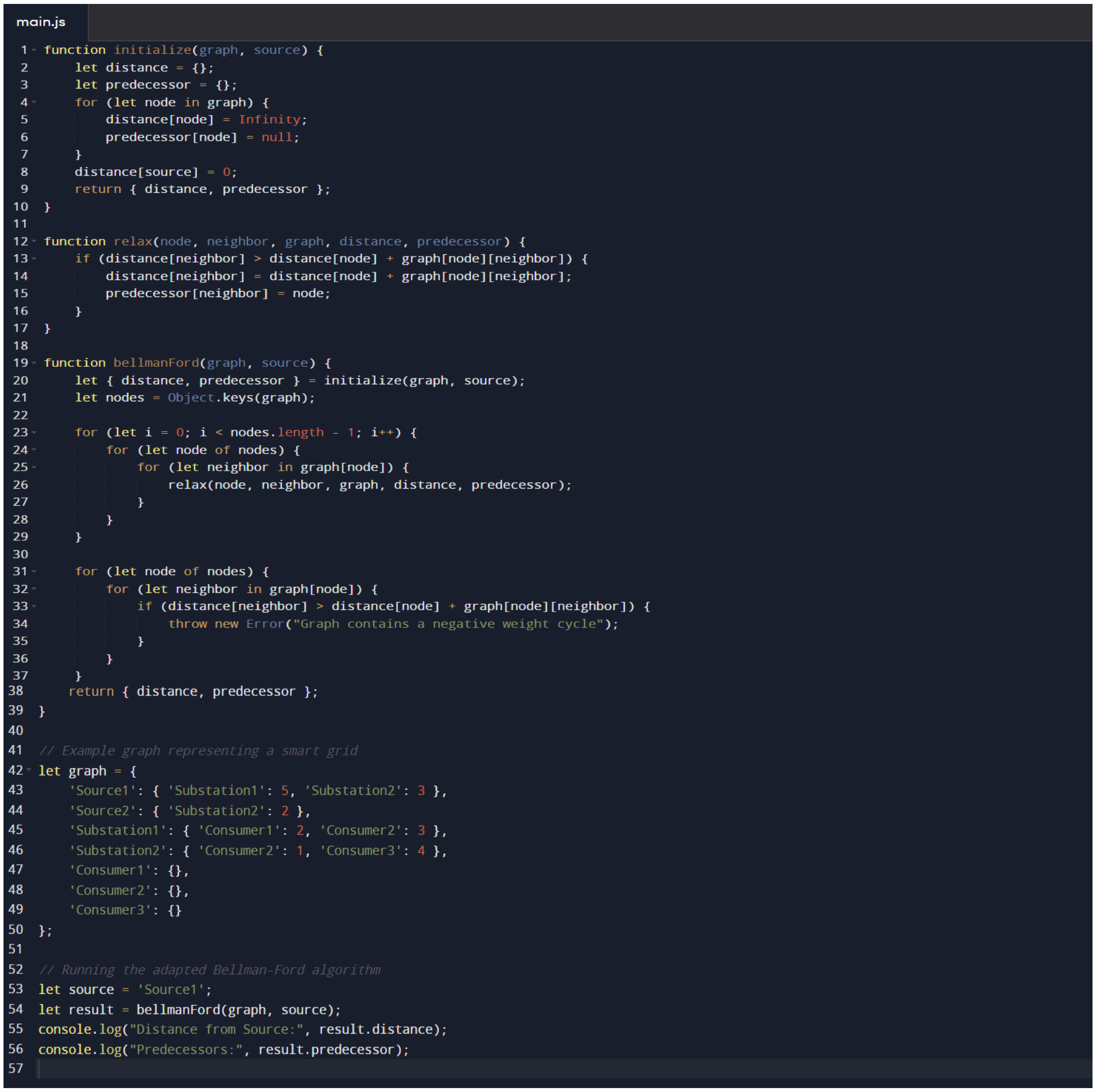
Objective: To optimize energy consumption in smart grids by adapting the Bellman-Ford algorithm, which is traditionally used for finding the shortest paths in weighted graphs, to manage the dynamic and complex nature of energy distribution in smart grid systems.

Algorithm Overview: The Bellman-Ford algorithm is used to find the shortest path from one starting point (node) to all other points in a network (graph). It can handle graphs where some connections have negative weights, making it suitable for scenarios with energy losses. Here, the algorithm is adapted to optimize energy distribution in a smart grid.

Key Steps in the Algorithm:

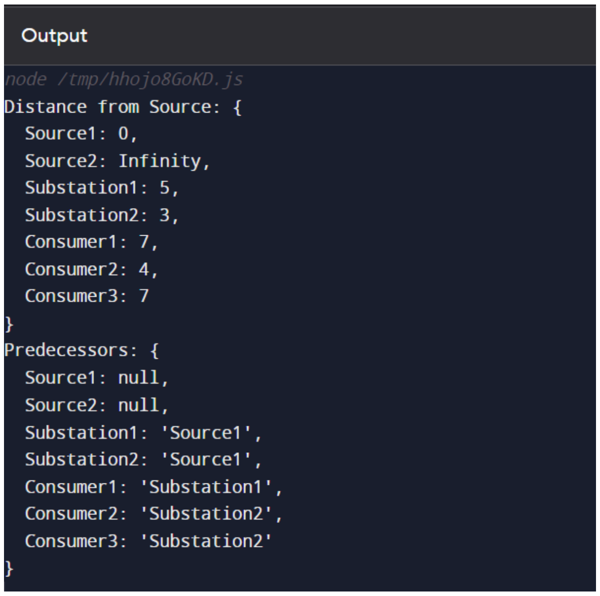
1. **Initialization:**
    - **Purpose:** Set up the starting conditions.
    - **Process:**
        - Set the distance to all nodes as infinity (`Infinity`), except for the starting node which is set to zero.
        - Create a map to keep track of the previous node for each node.
    - **Explanation:** This step ensures that initially, we assume the maximum possible distance to each node, except the starting node, which is zero because it is the starting point.
2. **Relaxation:**
    - **Purpose: Update the shortest path estimates for each node.**
    - **Process:**
        - For each connection, check if the known distance to the destination node can be shortened by taking the connection from the source node. If yes, update the distance and the previous node.
    - **Explanation:** This step repeatedly checks all connections to find shorter paths to each node and updates the shortest paths accordingly.
3. **Detection of Negative Weight Cycles:**
    - **Purpose: Ensure the graph does not contain any negative weight cycles.**
    - **Process:**
        - After relaxing all connections $|V|-1$|V| - 1$|V|-1$ times (where $|V|$|V|$|V|$ is the number of nodes), check for any further distance improvements. If any are found, it indicates a negative weight cycle.
    - **Explanation:** Negative weight cycles mean that the total path length can keep decreasing indefinitely, which is not possible for finding the shortest path.

CODE IN JAVASCRIPT



main.js

```javascript
function initialize(graph, source) {
    let distance = {};
    let predecessor = {};
    for (let node in graph) {
        distance[node] = Infinity;
        predecessor[node] = null;
    }
    distance[source] = 0;
    return { distance, predecessor };
}

function relax(node, neighbor, graph, distance, predecessor) {
    if (distance[neighbor] > distance[node] + graph[node][neighbor]) {
        distance[neighbor] = distance[node] + graph[node][neighbor];
        predecessor[neighbor] = node;
    }
}

function bellmanFord(graph, source) {
    let { distance, predecessor } = initialize(graph, source);
    let nodes = Object.keys(graph);

    for (let i = 0; i < nodes.length - 1; i++) {
        for (let node of nodes) {
            for (let neighbor in graph[node]) {
                relax(node, neighbor, graph, distance, predecessor);
            }
        }
    }

    for (let node of nodes) {
        for (let neighbor in graph[node]) {
            if (distance[neighbor] > distance[node] + graph[node][neighbor]) {
                throw new Error("Graph contains a negative weight cycle");
            }
        }
    }
    return { distance, predecessor };
}

// Example graph representing a smart grid
let graph = {
    'Source1': { 'Substation1': 5, 'Substation2': 3 },
    'Source2': { 'Substation2': 2 },
    'Substation1': { 'Consumer1': 2, 'Consumer2': 3 },
    'Substation2': { 'Consumer2': 1, 'Consumer3': 4 },
    'Consumer1': {},
    'Consumer2': {},
    'Consumer3': {}
};

// Running the adapted Bellman-Ford algorithm
let source = 'Source1';
let result = bellmanFord(graph, source);
console.log("Distance from Source:", result.distance);
console.log("Predecessors:", result.predecessor);
```



Output

```
node /tmp/hhojo8GoKD.js
Distance from Source: {
  Source1: 0,
  Source2: Infinity,
  Substation1: 5,
  Substation2: 3,
  Consumer1: 7,
  Consumer2: 4,
  Consumer3: 7
}
Predecessors: {
  Source1: null,
  Source2: null,
  Substation1: 'Source1',
  Substation2: 'Source1',
  Consumer1: 'Substation1',
  Consumer2: 'Substation2',
  Consumer3: 'Substation2'
}
```