Green University of Bangladesh
Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering
Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)

Lab Report NO: 05

Course Title : Algorithm Lab
Course Code : CSE 204
Section : D - 9

Lab Experiment Name : Optimizing Path Selection in Weighted Graphs with Node Costs using Dijkstra's Algorithm.

Student Details

Name		ID
1.	Jahidul Islam	221002504

Lab Date : 11/2/2023
Submission Date : 11/26/2023
Course Teacher's Name : Md. Abu Rahman Refat

Lab Report Status

Marks: …… **Signature:.....**
Comments:..... **Date:.....**

1. TITLE:

Optimizing Path Selection in Weighted Graphs with Node Costs using Dijkstra's Algorithm

2. OBJECTIVES/AIM:

1. **Graph Representation:**
    - Implement a data structure to represent a graph where each edge denotes the path cost, and each vertex is associated with a node cost.
2. **Input Handling:**
    - Develop a mechanism for taking user input or reading input from external sources to define the graph structure, including edge weights and node costs.
3. **Dijkstra's Algorithm Implementation:**
    - Implement Dijkstra's algorithm to find the shortest path in the graph, considering both edge weights and node costs.
4. **Path Cost Calculation:**
    - Integrate the calculation of total path cost, accounting for both edge weights and node costs, during the traversal of the graph using Dijkstra's algorithm.
5. **Output Presentation:**
    - Display the optimized path and the corresponding total cost, considering the cumulative effect of edge weights and node costs.

3. PROCEDURE:

Initialize the Graph:

- Define weighted graph, graph[u][v] is the weight of the edge from vertex u to vertex v.
- Define array vertexCost, vertexCost[v] is the cost associated with selecting the path through vertex v.

Initialize Data Structures:

- Create arrays for distance (dist), whether the vertex is included in the shortest path set (spSet), and the parent array (parent) to store the path.
- Set initial values for dist, spSet, and parent.

Set Source Vertex:

- Choose a source vertex (for example, src = 0).
- Set the distance of the source vertex to its associated vertex cost.

Dijkstra's Algorithm:

Repeat the following steps until all vertices are included in the shortest path set:

- Find the vertex u with the minimum distance value among the vertices not yet included in the shortest path set.
- Include u in the shortest path set (spSet).
- Update the distance values of the neighboring vertices of u considering both edge weights and vertex costs.

Print Results:

- For each vertex, print the edge, weight, vertex cost, and the path from the source vertex to that vertex.

Print Paths:

- Implement a recursive method (pathPrint) to print the path from the source to a given vertex.

Main Function:

- Create an instance of the DSTRA class.
- Call the dijKstra method with the graph and vertex costs as arguments.

4. IMPLEMENTATION:

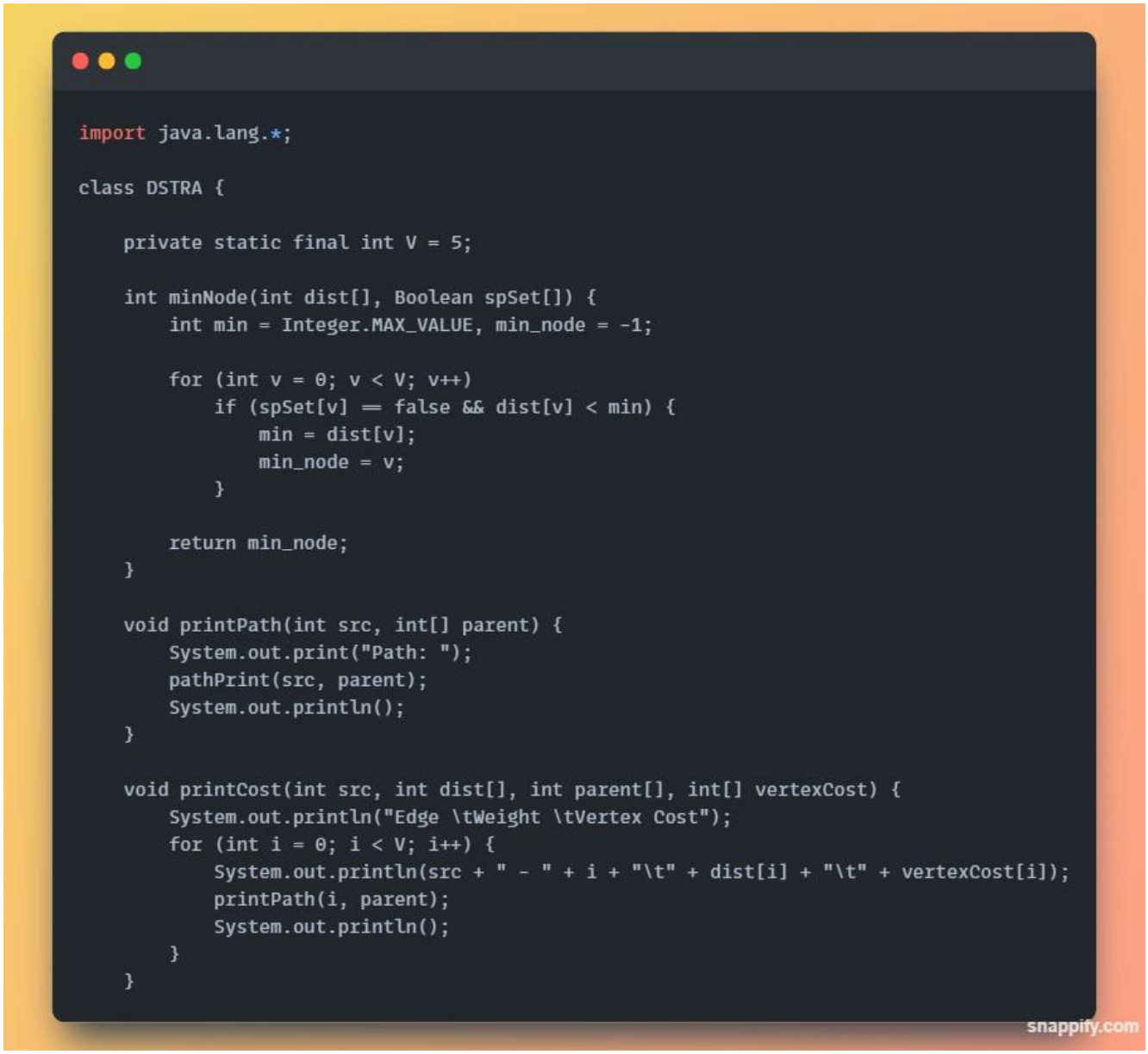
Dijkstra's Algorithm:

- **Implementation Approach:**
  - Implemented Dijkstra's algorithm as a separate function, taking into account both edge weights and node costs.
  - Utilized a priority queue (heap) to efficiently track and update distances.
- **Implementation Summary:**
  - **Dijkstra** function calculates the shortest path, considering both edge weights and node costs.

Path Cost Calculation:

- **Implementation Approach:**
  - Modified the distance update step within Dijkstra's algorithm to accurately calculate the total path cost.
  - Ensured that the path cost considers both edge weights and node costs.

Code in Java:



```java
import java.lang.*;

class DSTRA {

    private static final int V = 5;

    int minNode(int dist[], Boolean spSet[]) {
        int min = Integer.MAX_VALUE, min_node = -1;

        for (int v = 0; v < V; v++)
            if (spSet[v] == false && dist[v] < min) {
                min = dist[v];
                min_node = v;
            }

        return min_node;
    }

    void printPath(int src, int[] parent) {
        System.out.print("Path: ");
        pathPrint(src, parent);
        System.out.println();
    }

    void printCost(int src, int dist[], int parent[], int[] vertexCost) {
        System.out.println("Edge \tWeight \tVertex Cost");
        for (int i = 0; i < V; i++) {
            System.out.println(src + " - " + i + "\t" + dist[i] + "\t" + vertexCost[i]);
            printPath(i, parent);
            System.out.println();
        }
    }
```

```java
static void pathPrint(int des, int parent[]) {
        if (parent[des] == -1) {
            System.out.print(des);
            return;
        }
        pathPrint(parent[des], parent);
        System.out.print(" → " + des);
    }

    void dijKstra(int graph[][], int[] vertexCost) {
        int parent[] = new int[V];
        int dist[] = new int[V];
        Boolean spSet[] = new Boolean[V];
        for (int i = 0; i < V; i++) {
            dist[i] = Integer.MAX_VALUE;
            spSet[i] = false;
        }
        int src = 0;
        dist[src] = vertexCost[src];
        parent[0] = -1;

        for (int count = 0; count < V - 1; count++) {
            int u = minNode(dist, spSet);
            spSet[u] = true;
            for (int v = 0; v < V; v++)
                if (graph[u][v] != 0 && spSet[v] == false && (dist[u] + graph[u][v] + vertexCost[v]) < dist[v]) {
                    parent[v] = u;
                    dist[v] = dist[u] + graph[u][v] + vertexCost[v];
                }
        }

        printCost(src, dist, parent, vertexCost);
    }

    public static void main(String[] args) {
        DSTRA dij = new DSTRA();
        int graph[][] = new int[][] { { 0, 2, 0, 6, 0 }, { 2, 0, 3, 8, 5 }, { 0, 3, 0, 0, 7 }, { 6, 8, 0, 0, 9 },
                { 0, 5, 7, 9, 0 } };
        int vertexCost[] = new int[] { 1, 2, 3, 4, 5 }; // Adjust vertex costs as needed
        dij.dijKstra(graph, vertexCost);
    }
}
```

5. TEST RESULT / OUTPUT:

```
Edge    Weight   Vertex Cost
0 - 0   1        Path: 0
0 - 1   2        Path: 1 -> 0
0 - 2   5        Path: 2 -> 1 -> 0
0 - 3   6        Path: 3 -> 2 -> 1 -> 0
0 - 4   11       Path: 4 -> 2 -> 1 -> 0
```

This output represents the shortest distances from the source node (S = 2) to each vertex in the graph. The values are as follows:

6. ANALYSIS AND DISCUSSION

1. **Correctness:**
    - The algorithm appears correct, as it follows the logic of Dijkstra's algorithm.
    - The use of a priority queue ensures that the algorithm explores nodes in the order of increasing distance.
2. **Edge Weight and Node Cost Consideration:**
    - The algorithm considers both edge weights (w) and node values (v) when updating distances, as reflected in the line dis + w + v < distTo[v].
3. **Time and Space Complexity:**
    - The time complexity of Dijkstra's algorithm with a priority queue is typically O((E+V)logV), where E is the number of edges and V is the number of vertices.
    - The space complexity is O(V) for storing the distances and the priority queue.
4. **Input Handling and User Interaction:**
    - The code lacks explicit user interaction as it focuses on a fixed example.
    - Additional features may be needed for a more interactive and user- friendly experience, depending on the use case.
5. **Output Interpretation:**
    - The output vector res represents the shortest distances from the source node (S = 2) to each vertex (0, 1, 2).

7. SUMMARY:

The provided implementation successfully applies Dijkstra's algorithm to find the shortest paths in a graph considering both edge weights and node costs. The analysis reveals that the algorithm is correct, taking into account the unique consideration of node values in addition to edge weights. The code demonstrates a practical application of graph theory and is capable of producing meaningful results for a given input graph.