DIJKSTRA'S ALGORITHM

A project report submitted in partial fulfillment of the requirement for

The award of

The degree of **bachelor's in computer science**

In the **department of computer science**

Submitted by:

Muhammad Sarim Effendi

2012304

3-E

Course Instructor: Muhammad Danish Khan

Course: CSC1201 Discrete Mathematical Structures

Submission Date: January 2022

Introduction

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

Working

- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Time and space complexity

Time complexity: Time complexity will be O (E log V) because priority queue used. Where V is Vertices and E is Edges.

Space complexity: Space: O (V + E)

Real life uses

- Digital Mapping Services in Google Maps to find the shortest path between the current location and the destination.
- It has broad applications in industry, especially in domains that require modeling networks.
- Travelling routes with the lowest cost.
- road conditions, road closures and construction
- IP routing to detect Open Shortest Path First.
- Robotic Path

Advantages

- One of the main advantages of it is its little complexity which is almost linear.

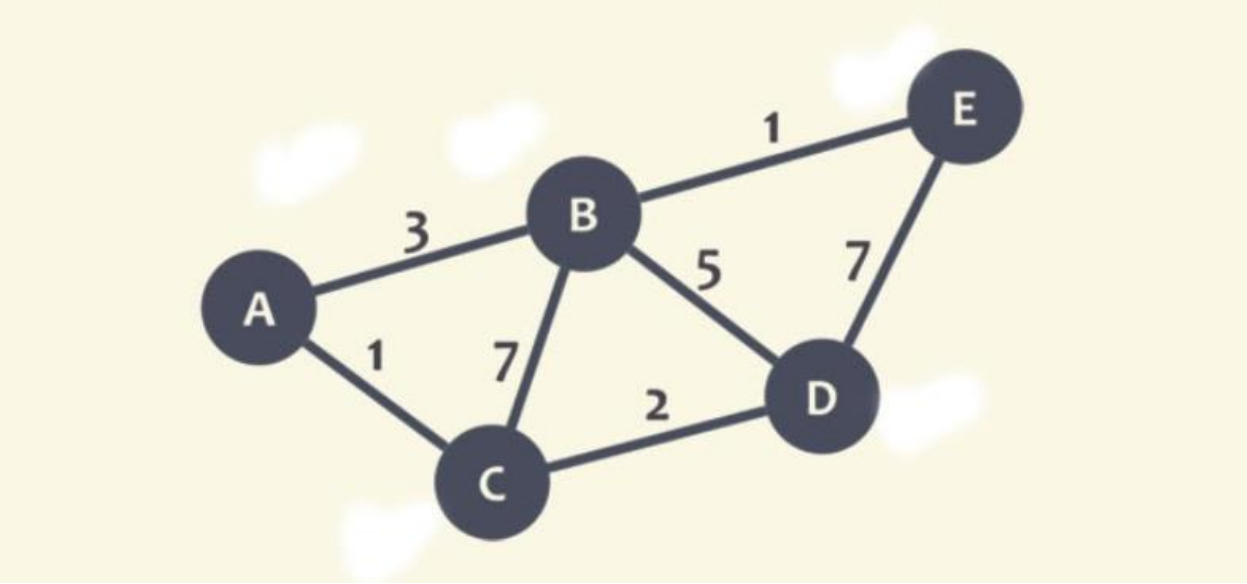
- It can be used to calculate the shortest path between a single node to all other nodes and a single source node to a single destination node by stopping the algorithm once the shortest distance is achieved for the destination node.

Disadvantages

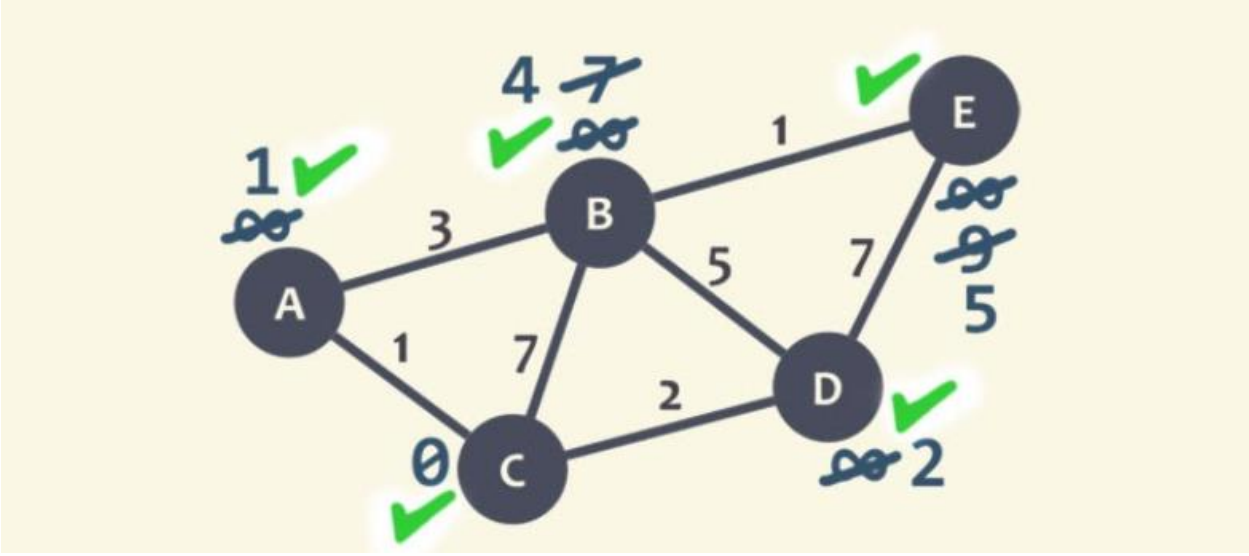
- It is unable to handle negative edges
- there is a need to maintain tracking of vertices, have been visited.

Examples of Dijkstra's algorithm

A picture of weighted graph whose shortest path needs to found



A picture of shortest path found for the source node C to every other node



Java Implementation of Dijkstra's Algorithm
For calculating the shortest path between the cities

```java
package com.company;
// Java Program to Implement Dijkstra's Algorithm
// Using Priority Queue
// Importing utility classes
import java.util.*;
// Main class DPQ
public class Main {
    // Member variables of this class
    private int dist[];
    private Set<Integer> settled;
    private PriorityQueue<Node> pq;
    // Number of vertices
    private int V;

    List<List<Node>> adj;
    // Constructor of this class

    public Main(int V) {
        // This keyword refers to current object itself
        this.V = V;
        dist = new int[V];
        settled = new HashSet<Integer>();
        pq = new PriorityQueue<Node>(V, new Node());
    }

    // Method 1
    // Dijkstra's Algorithm
    // Dijkstra's Algorithm
    public void dijkstra(List<List<Node>> adj, int src) {
        this.adj = adj;

        for (int i = 0; i < V; i++)
            dist[i] = Integer.MAX_VALUE;

        // Add source node to the priority queue
        pq.add(new Node(src, cost: 0));

        // Distance to the source is 0
        dist[src] = 0;

        while (settled.size() != V) {

            // Terminating ondition check when
            // the priority queue is empty, return
            if (pq.isEmpty())
                return;

            // Removing the minimum distance node
            // from the priority queue
            int u = pq.remove().node;

            // Adding the node whose distance is
            // finalized
            if (settled.contains(u))
```

```
            if (settled.contains(u))

                // Continue keyword skips exwcution for
                // following check
                continue;

            // We don't have to call e_Neighbors(u)
            // if u is already present in the settled set.
            settled.add(u);

            e_Neighbours(u);
        }
    }

    // Method 2
    // To process all the neighbours
    // of the passed node
    private void e_Neighbours(int u) {

        int edgeDistance = -1;
        int newDistance = -1;

        // All the neighbors of v
        for (int i = 0; i < adj.get(u).size(); i++) {
            Node v = adj.get(u).get(i);

            // If current node hasn't already been processed
            // If current node hasn't already been processed
            if (!settled.contains(v.node)) {
                edgeDistance = v.cost;
                newDistance = dist[u] + edgeDistance;

                // If new distance is cheaper in cost
                if (newDistance < dist[v.node])
                    dist[v.node] = newDistance;

                // Add the current node to the queue
                pq.add(new Node(v.node, dist[v.node]));
            }
        }
    }

    // Main driver method
    public static void main(String arg[]) {

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter number of Cities");
        int v = sc.nextInt();
        List<List<Node>> adj
                = new ArrayList<List<Node>>();

        // Initialize list for every node
        for (int i = 0; i < v; i++) {
            List<Node> item = new ArrayList<Node>();
```

```
            List<Node> item = new ArrayList<Node>();
            adj.add(item);
        }
        System.out.println("Enter the names of the " + v + " Cities");
        ArrayList<String> Cities = new ArrayList<String>();
        for (int f = 0; f < v; f++) {
            String city = sc.next();
            Cities.add(city);
            System.out.println(Cities.get(f));
            System.out.println(Cities.indexOf(city));
        }
        for (int i = 0; i < Cities.size(); i++) {
            System.out.println(Cities.get(i));
        }
        System.out.println("Enter number of Connections");
        int e = sc.nextInt();
        for (int i = 0; i < e; i++) {
            System.out.println("From");
            String city = sc.next();//edge from start

            int f = Cities.indexOf(city);
            System.out.println(f);
            System.out.println("To");
            String city2 = sc.next();//edge end
            int h = Cities.indexOf(city2);
            System.out.println(h);
            System.out.println("weight");
            int weight = sc.nextInt();//edge weight


            adj.get(f).add(new Node(h, weight));

        }
        System.out.println("Enter source city");
        String sourceCity = sc.next();
        int source = Cities.indexOf(sourceCity);
        System.out.println(source);
        // Calculating the single source shortest path
        Main dpq = new Main(v);
        dpq.dijkstra(adj, source);
        // Printing the shortest path to all the nodes
        // from the source node
        System.out.println("The shorted path from node :");

        for (int i = 0; i < dpq.dist.length; i++)
            System.out.println(Cities.get(source) + " to " + Cities.get(i) + " is "
                    + dpq.dist[i]);
    }
}
// Class 2
```

```java
// Helper class implementing Comparator interface
// Representing a node in the graph
class Node implements Comparator<Node> {

    // Member variables of this class
    public int node;
    public int cost;

    // Constructors of this class

    // Constructor 1
    public Node() {
    }

    // Constructor 2
    public Node(int node, int cost) {

        // This keyword refers to current instance itself
        this.node = node;
        this.cost = cost;
    }

    // Method 1
    @Override
    public int compare(Node node1, Node node2) {

        if (node1.cost < node2.cost)
            return -1;

        if (node1.cost > node2.cost)
            return 1;

        return 0;
    }
}
```

Output:

```
C:\Users\sarim\.jdks\openjdk-17.0.1\bin\java.exe "-javaagent:C:\Program
Enter number of Cities
7
Enter the names of the 7 Cities
karachi
karachi
0
hyd
hyd
1
sukkur
sukkur
2
larkana
larkana
3
lhr
lhr
4
multan
multan
isb
isb
6
karachi
hyd
sukkur
larkana
lhr
multan
isb
Enter number of Connections
12
From
karachi
0
To
hyd
1
weight
9
From
```

```
From
karachi
0
To
sukkur
2
weight
5
From
hyd
1
To
lhr
4
weight
13
From
hyd
1
```

```
To
larkana
3
weight
13
From
sukkur
2
To
larkana
3
weight
3
From
sukkur
2
To
multan
5
weight
7
```

```
weight
7
From
sukkur
2
To
isb
6
weight
22
From
larkana
3
To
lhr
4
weight
21
From
larkana
3
To
multan
5
weight
4
From
multan
5
To
lhr
4
weight
11
From
multan
5
To
isb
6
weight
27
```

```
weight
27
From
lhr
4
To
isb
6
weight
17
Enter source city
karachi
```

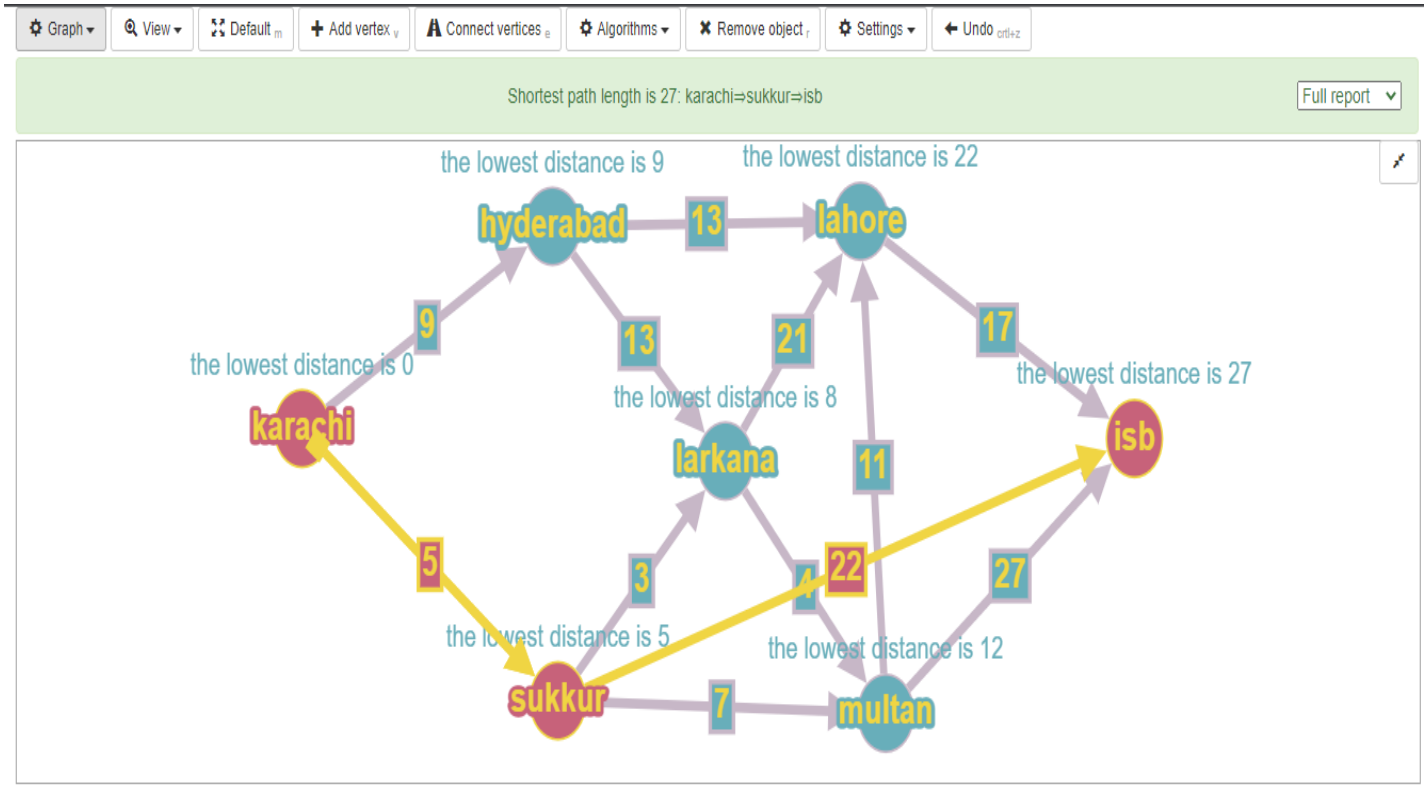
Final output:

```
To
isb
6
weight
17
Enter source city
karachi

0
The shorted path from node :
karachi to karachi is 0
karachi to hyd is 9
karachi to sukkur is 5
karachi to larkana is 8
karachi to lhr is 22
karachi to multan is 12
karachi to isb is 27

Process finished with exit code 0
```

Explanation of the program

This is a program made using Dijkstra's algorithm to find the shortest path for cities(vertices). User first have to enter the number cities(vertices) and give the names of the cities(vertices) then user have to enter the number of connections(edges) and then have to enter the connections(edges) in between the cities(vertices) and put a weight to it after entering these users have to tell the source city (source node) from which the algorithm will calculate the shortest path to every other city(vertex).

Graphical representation of the above output



References

https://graphonline.ru/en/#

https://www.analyticssteps.com

https://www.freecodecamp.org

https://www.geeksforgeeks.org