SHORTEST PATH

MADE BY:

ALISHER YEGIZOV

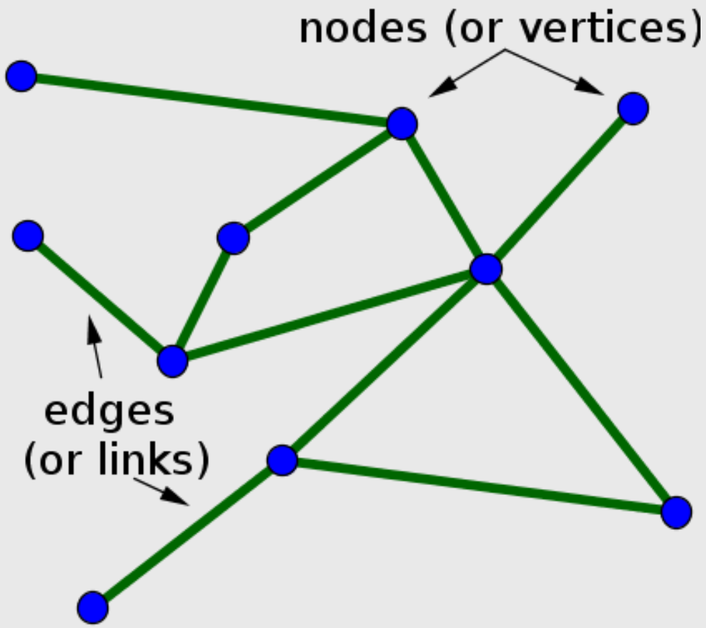
ZHUSSUPALI SALAMAT

SULTAN ZHUANDYK

YERMAKHAN KUANYSHBEKOV

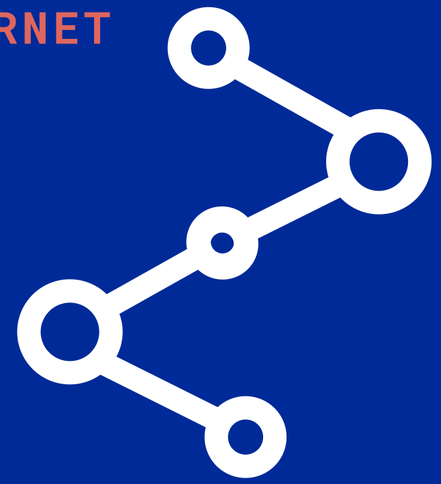
REAL-LIFE GRAPHS

GRAPH THEORY IS THE STUDY OF GRAPHS, WHICH ARE MATHEMATICAL STRUCTURES USED TO MODEL PAIRWISE RELATIONS BETWEEN OBJECTS. A GRAPH IN THIS CONTEXT IS MADE UP OF VERTICES (ALSO CALLED NODES OR POINTS) WHICH ARE CONNECTED BY EDGES.



EVERYTHING IN OUR WORLD IS LINKED:

- ALL CITIES ARE LINKED BY ROADS
- PAGES ARE LINKED BY HYPERLINKS ON THE INTERNET
- FLIGHT AND RAIL NETWORK
- MOBILE NETWORK
- COMPONENTS OF ELECTRIC CIRCUIT
- COMPONENTS OF COMPUTER CHIPS AND ETC…



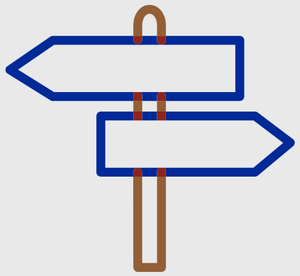
BY GRAPHS, WE CAN SIMULATE ALL THESE NETWORKS TO MAKE SOME VISUAL ANALYSIS LIKE FINDING CONNECTIONS AND SHORTEST PATHS BETWEEN NODES.

DIJKSTRA ALGORITHM

ALGORITHMS

BELLMAN-FORD ALGORITHM

ALGORITHM



WHAT IS DIJKSTRA'S SHORTEST PATH ALGORITHM?

- DIJKSTRA ALGORITHM IS A GREEDY ALGORITHM.

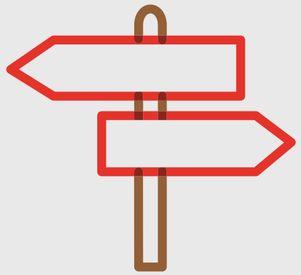
- IT FINDS A SHORTEST PATH TREE FOR A WEIGHTED UNDIRECTED GRAPH.
  - HIS MEANS IT FINDS A SHORTEST PATHS BETWEEN NODES IN A GRAPH, WHICH MAY REPRESENT, FOR EXAMPLE, ROAD NETWORKS

- FOR A GIVEN SOURCE NODE IN THE GRAPH, THE ALGORITHM FINDS THE SHORTEST PATH BETWEEN SOURCE NODE AND EVERY OTHER NODE.
  - THIS ALGORITHM ALSO USED FOR FINDING THE SHORTEST PATHS FROM A SINGLE NODE TO A SINGLE DESTINATION NODE BY STOPPING THE ALGORITHM ONCE THE SHORTEST PATH TO THE DESTINATION NODE HAS BEEN DETERMINED.

- DIJKSTRA'S ALGORITHM IS VERY SIMILAR TO PRIM'S ALGORITHM. IN PRIM'S ALGORITHM WE CREATE MINIMUM SPANNING TREE (MST) AND IN DIJKSTRA ALGORITHM WE CREATE SHORTEST PATH TREE (SPT) FROM THE GIVEN SOURCE..

ALGORITHM

WHAT IS BELLMAN-FORD'S SHORTEST PATH ALGORITHM?



THE BELLMAN-FORD ALGORITHM IS A GRAPH SEARCH ALGORITHM THAT FINDS THE SHORTEST PATH BETWEEN A GIVEN SOURCE VERTEX AND ALL OTHER VERTICES IN THE GRAPH. THIS ALGORITHM CAN BE USED ON BOTH WEIGHTED AND UNWEIGHTED GRAPHS.

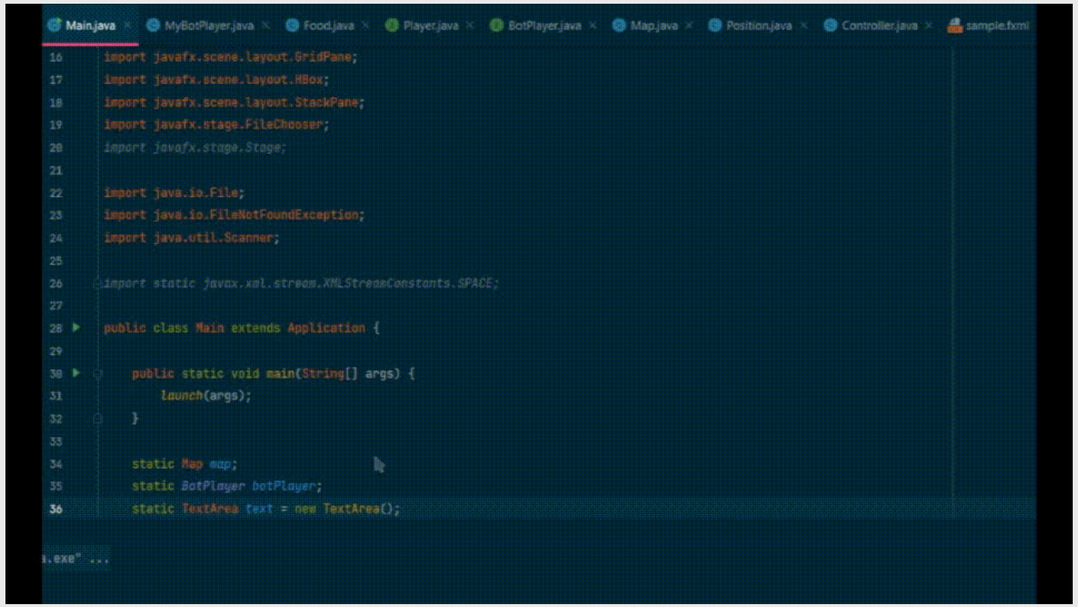
LIKE DIJKSTRA'S SHORTEST PATH ALGORITHM, THE BELLMAN-FORD ALGORITHM IS GUARANTEED TO FIND THE SHORTEST PATH IN A GRAPH. THOUGH IT IS SLOWER THAN DIJKSTRA'S ALGORITHM, BELLMAN-FORD IS CAPABLE OF HANDLING GRAPHS THAT CONTAIN NEGATIVE EDGE WEIGHTS, SO IT IS MORE VERSATILE.

IT IS WORTH NOTING THAT IF THERE EXISTS A NEGATIVE CYCLE IN THE GRAPH, THEN THERE IS NO SHORTEST PATH. GOING AROUND THE NEGATIVE CYCLE AN INFINITE NUMBER OF TIMES WOULD CONTINUE TO DECREASE THE COST OF THE PATH (EVEN THOUGH THE PATH LENGTH IS INCREASING). BECAUSE OF THIS, BELLMAN-FORD CAN ALSO DETECT NEGATIVE CYCLES WHICH IS A USEFUL FEATURE.

SHORTEST PATH APPLICATIONS

GAMES

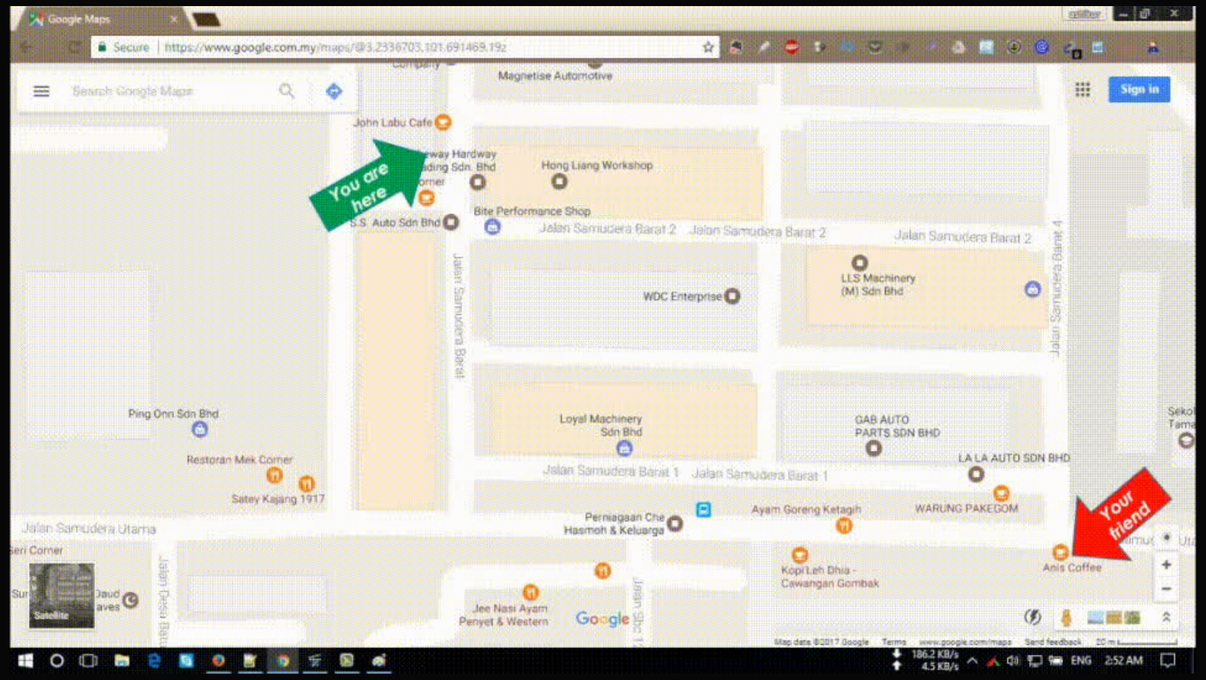
WHAT ABOUT USES OF THE SHORTEST PATH ALGORITHMS, THE VERY FIRST THING THAT CAME TO OUR MIND WAS THE PROJECT WE WORKED ON IN THE JAVA2 COURSE.



Gif: https://drive.google.com/drive/folders/1S7sY4jWE29hvo9_DolZS5-yhzcuXMTp_?usp=sharing

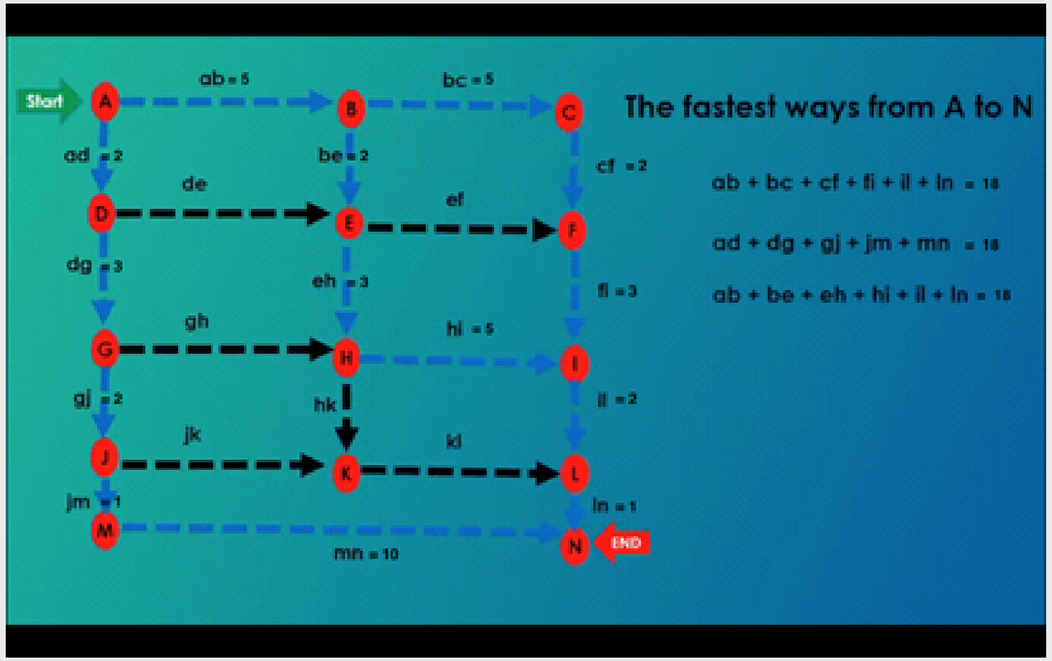
MAP SERVICES

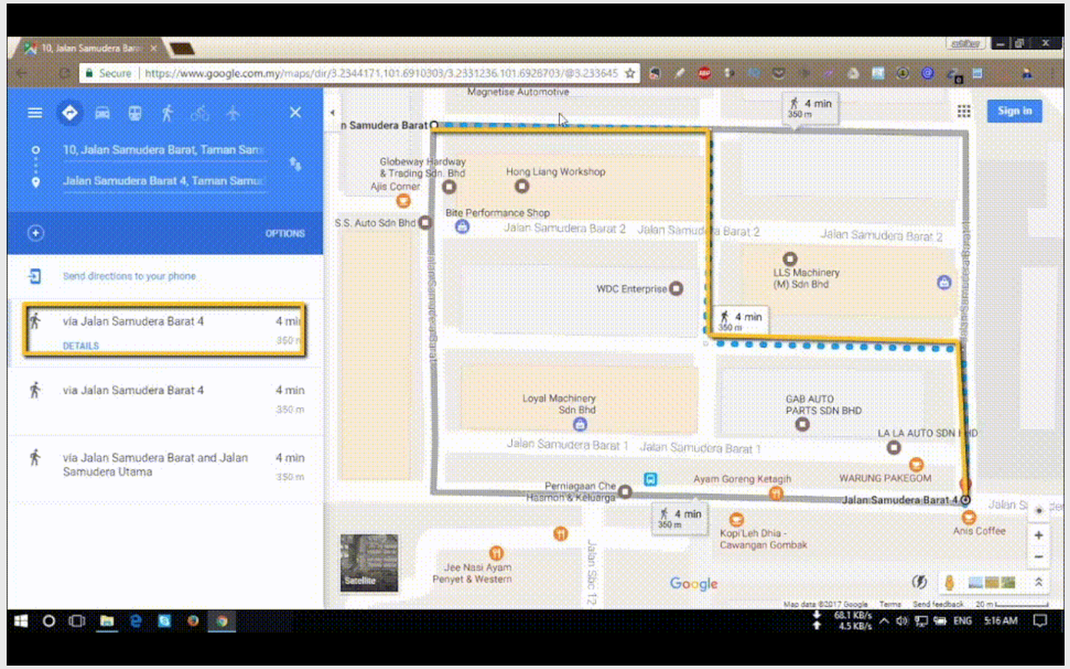
SHORTEST PATH ALGORITHMS ARE APPLIED TO AUTOMATICALLY FIND DIRECTIONS BETWEEN PHYSICAL LOCATIONS, SUCH AS DRIVING DIRECTIONS ON WEB MAPPING WEBSITES LIKE MAPQUEST OR GOOGLE MAPS.



Gif: https://drive.google.com/drive/folders/1ppboprlLA8NMHf3cf2_qMyodLfGZR3v0?usp=sharing

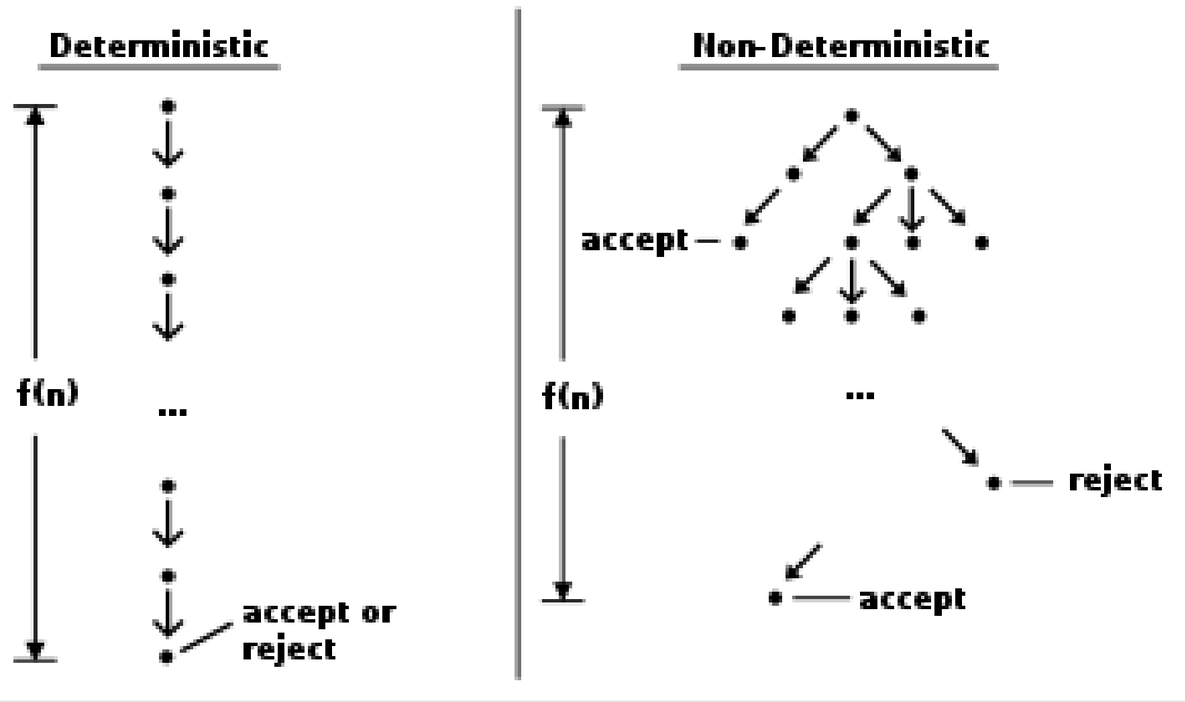
ANALYSIS AND GOOGLEMAPS RESULTS FROM GIF:



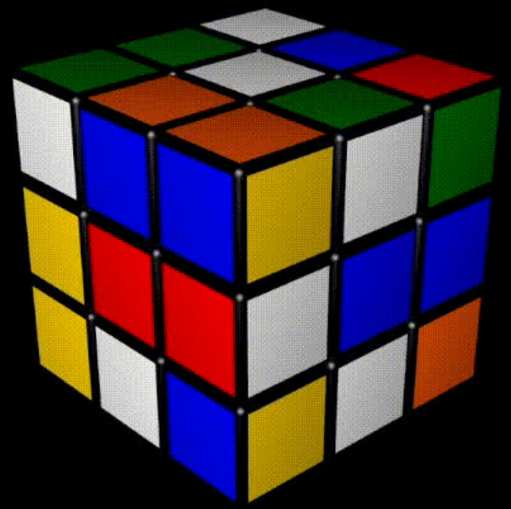


NONDETERMINISTIC MACHINE

IF ONE REPRESENTS A NONDETERMINISTIC ABSTRACT MACHINE AS A GRAPH WHERE VERTICES DESCRIBE STATES AND EDGES DESCRIBE POSSIBLE TRANSITIONS, SHORTEST PATH ALGORITHMS CAN BE USED TO FIND AN OPTIMAL SEQUENCE OF CHOICES TO REACH A CERTAIN GOAL STATE



REAL LIFE EXAMPLE:



Gif: https://drive.google.com/drive/folders/13W2GZKHRodoSSOcLbwWxK61MigXJRUrL?usp=sharing

ALL OF THESE ALGORITHMS WORK IN TWO PHASES. IN THE FIRST PHASE, THE GRAPH IS PREPROCESSED WITHOUT KNOWING THE SOURCE OR TARGET NODE. THE SECOND PHASE IS THE QUERY PHASE. IN THIS PHASE, SOURCE AND TARGET NODES ARE KNOWN. THE IDEA IS THAT THE FINAL NODE IS STATIC, SO THE PREPROCESSING PHASE CAN BE DONE ONCE AND USED FOR A LARGE NUMBER OF QUERIES ON THE SAME GRAPH.

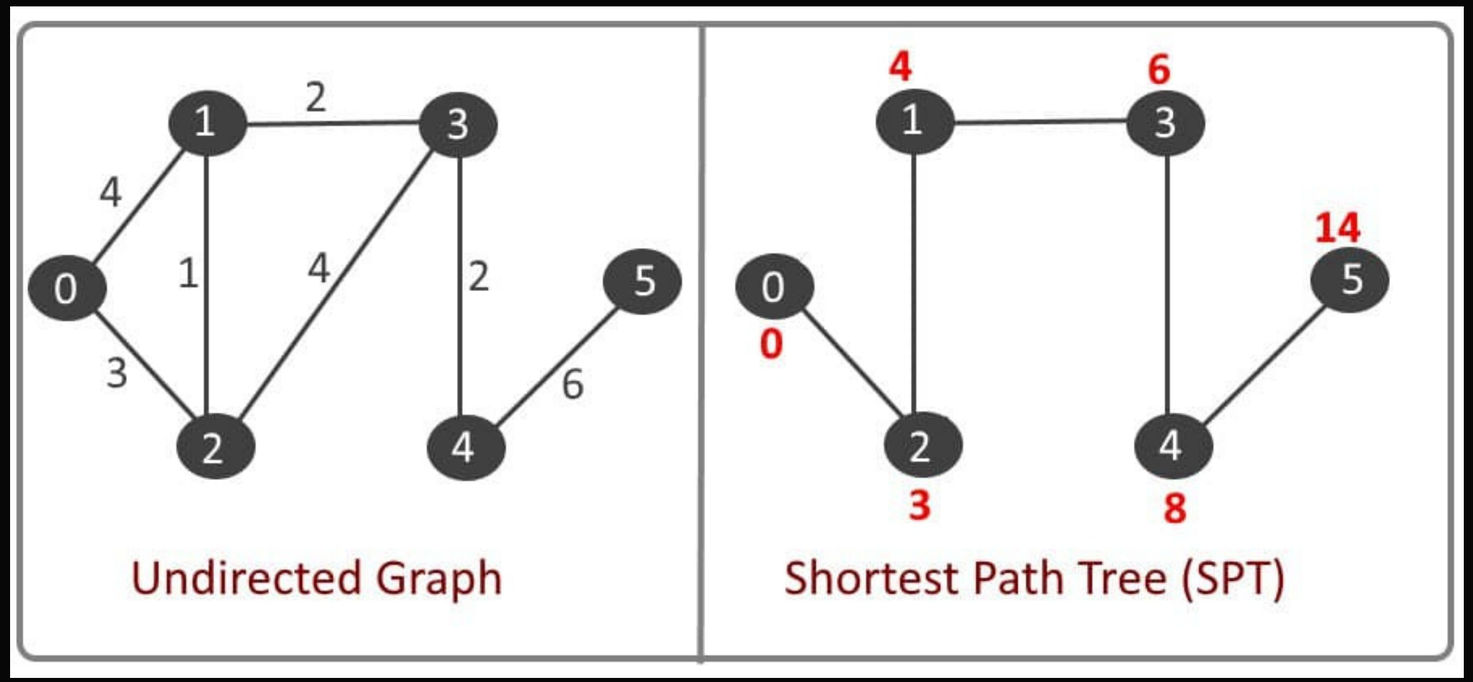
CODE

DIJKSTRA ALGORITHM

```
public class SPT {
    static class Graph {
        int vertices;
        int matrix[][];

        public Graph(int vertex) {
            this.vertices = vertex;
            matrix = new int[vertex][vertex];
        }

        public void addEdge(int v, int w, int c) {
            matrix[v][w] = c;
            matrix[w][v] = c;
        }


        int getMinVertex(boolean[] list, int[] key) {
            int minKey = Integer.MAX_VALUE;
            int vertex = -1;
            for (int i = 0; i < vertices; i++) {
                if (list[i] == false && minKey > key[i]) {
                    minKey = key[i];
                    vertex = i;
                }
            }
            return vertex;
        }

        public void getMinDist(int sourceVertex) {
            boolean[] spt = new boolean[vertices];
            int[] distance = new int[vertices];
            int INFINITY = Integer.MAX_VALUE;

            for (int i = 0; i < vertices; i++) {
                distance[i] = INFINITY;
            }

//            for (int i = 0; i < vertices; i++) {
//                for (int j = 0; j < vertices; j++) {
//                    System.out.print(matrix[i][j] + " ");
//                }
//                System.out.println();
//            }

            distance[sourceVertex] = 0;
            for (int i = 0; i < vertices; i++) {
                int vertex_U = getMinVertex(spt, distance);
//                System.out.println("Vertex from getMinVertex equals " + vertex_U);
                spt[vertex_U] = true;
```

```
System.out.print("[ ");
//                for (int j = 0; j < spt.length;j++) {
//                    System.out.print(spt[j] + " ");
//                }
//                System.out.println("]");

                for (int vertex_V = 0; vertex_V < vertices; vertex_V++) {
                    if (matrix[vertex_U][vertex_V] > 0 && spt[vertex_V] == false) {
//                        System.out.println("Matrix " + matrix[vertex_U][vertex_V]);
//                        System.out.println("Distance " + distance[vertex_U]);
                        int distanceUpdate = matrix[vertex_U][vertex_V] + distance[vertex_U];
//                        System.out.println("Sum of Matrix + Distance " + distanceUpdate);
                        if (distanceUpdate < distance[vertex_V])
                            distance[vertex_V] = distanceUpdate;
                    }
                }
            }
            //print shortest path tree
//            System.out.println( "Source Vertex " + sourceVertex);
//            for (int i = 0; i < distance.length;i++) {
//                System.out.println(i + " distance  " + distance[i]);
//            }
            printSPT(sourceVertex, distance);
        }

        public void printSPT(int sourceVertex, int[] key) {
            System.out.println("SPT Algorithm: ");
            for (int i = 0; i < vertices; i++) {
                System.out.println("Source Vertex: " + sourceVertex + " to vertex " + i +
                        " distance: " + key[i]);
            }
        }
    }

    public static void main(String[] args) {
        Graph graph = new Graph(6);
        int sourceVertex = 0;
        graph.addEdge(0, 1, 4);
        graph.addEdge(0, 2, 3);
        graph.addEdge(1, 2, 1);
        graph.addEdge(1, 3, 2);
        graph.addEdge(2, 3, 4);
        graph.addEdge(3, 4, 2);
        graph.addEdge(4, 5, 6);
        graph.getMinDist(sourceVertex);
    }
}
```

GRAPH REPRESENTATION:

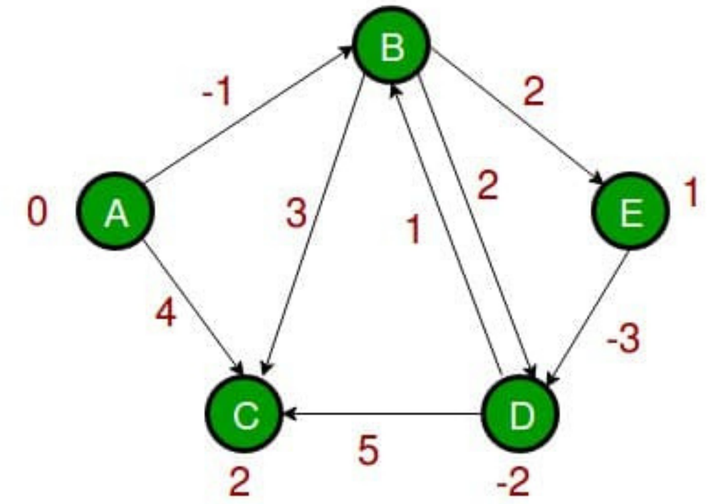


BELLMAN-FORD ALGORITHM

```
public class SPT2 {
    static void BellmanFord(int graph[][], int V, int E, int src) {
        int []dis = new int[V];
        for (int i = 0; i < V; i++) {
            dis[i] = Integer.MAX_VALUE;
        }

        dis[src] = 0;

        for (int i = 0; i < V - 1; i++)
        {

            for (int j = 0; j < E; j++)
            {
                if (dis[graph[j][0]] != Integer.MAX_VALUE && dis[graph[j][0]] + graph[j][2] <
                                dis[graph[j][1]])
                    dis[graph[j][1]] =
                    dis[graph[j][0]] + graph[j][2];
            }
        }


        for (int i = 0; i < E; i++) {
            int x = graph[i][0];
            int y = graph[i][1];
            int weight = graph[i][2];

            if (dis[x] != Integer.MAX_VALUE &&
                    dis[x] + weight < dis[y])
                System.out.println("Graph contains negative"
                        +" weight cycle");
        }

        System.out.println("Vertex Distance from Source");
        for (int i = 0; i < V; i++)
            System.out.println(i + "\t\t" + dis[i]);
    }

    public static void main(String[] args) {
        int V = 5; // Number of vertices in graph
        int E = 8; // Number of edges in graph

        int graph[][] = { { 0, 1, -1 }, { 0, 2, 4 },
                          { 1, 2, 3 }, { 1, 3, 2 },
                          { 1, 4, 2 }, { 3, 2, 5 },
                          { 3, 1, 1 }, { 4, 3, -3 } };

        BellmanFord(graph, V, E, 0);
    }
}
```

GRAPH REPRESENTATION:

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



THANK YOU FOR YOUR ATTENTION