

**Project Presentation of CSE 206**

**Course Title:  Algorithms Lab**

**Course Code: CSE 206**

**Section: PC DA**

**Submitted to   
Dr. Shah Murtaza Rashid Al Masud**

**Assistant Professor**

**Dept. of CSE**

**Green University of Bangladesh**

**Submitted by:**

**Mohammad Nazmul Hossain**

**ID:193902031**

**Dept. of CSE**

**Dijkstra's algorithm** is an algorithm for finding the shortest paths between nodes in a weighted graph. In Dijkstra’s algorithm, we always extract the node with the lowest cost. We can prove the correctness of this approach in the case of non-negative edges.

* we start from a source node and initialize its distance by zero.
* we push the source node to a priority queue with a cost equal to zero.
* The complexity of Dijkstra’s algorithm is O(V+E.log(V)).

**Code:-**

djikstraAlgorithm(startNode) {

let distances = {};

// Stores the reference to previous nodes

let prev = {};

let pq = new PriorityQueue(this.nodes.length \* this.nodes.length);

// Set distances to all nodes to be infinite except startNode

distances[startNode] = 0;

pq.enqueue(startNode, 0);

this.nodes.forEach(node => {

if (node !== startNode) distances[node] = Infinity;

prev[node] = null;

});

while (!pq.isEmpty()) {

let minNode = pq.dequeue();

let currNode = minNode.data;

let weight = minNode.priority;

this.edges[currNode].forEach(neighbor => {

let alt = distances[currNode] + neighbor.weight;

if (alt < distances[neighbor.node]) {

distances[neighbor.node] = alt;

prev[neighbor.node] = currNode;

pq.enqueue(neighbor.node, distances[neighbor.node]);

}

});

}

return distances;

}

let g = new Graph();

g.addNode("A");

g.addNode("B");

g.addNode("C");

g.addNode("D");

g.addNode("E");

g.addNode("F");

g.addNode("G");

g.addDirectedEdge("A", "C", 100);

g.addDirectedEdge("A", "B", 3);

g.addDirectedEdge("A", "D", 4);

g.addDirectedEdge("D", "C", 3);

g.addDirectedEdge("D", "E", 8);

g.addDirectedEdge("E", "F", 10);

g.addDirectedEdge("B", "G", 9);

g.addDirectedEdge("E", "G", 50);

console.log(g.djikstraAlgorithm("A"));

**Output:**

This will give the output −

**{ A: 0, B: 3, C: 7, D: 4, E: 12, F: 22, G: 12 }**

**Bellman-Ford Algorithm**The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.

* we start from a source node and initialize its distance by zero
* we push the source node to a priority queue with a cost equal to zero.
* The complexity of Dijkstra’s algorithm is O(V+E.log(V))

import GraphVertex from '../../../../data-structures/graph/GraphVertex';

import GraphEdge from '../../../../data-structures/graph/GraphEdge';

import Graph from '../../../../data-structures/graph/Graph';

import bellmanFord from '../bellmanFord';

describe('bellmanFord', () => {

it('should find minimum paths to all vertices for undirected graph', () => {

const vertexA = new GraphVertex('A');

const vertexB = new GraphVertex('B');

const vertexC = new GraphVertex('C');

const vertexD = new GraphVertex('D');

const vertexE = new GraphVertex('E');

const vertexF = new GraphVertex('F');

const vertexG = new GraphVertex('G');

const vertexH = new GraphVertex('H');

const edgeAB = new GraphEdge(vertexA, vertexB, 4);

const edgeAE = new GraphEdge(vertexA, vertexE, 7);

const edgeAC = new GraphEdge(vertexA, vertexC, 3);

const edgeBC = new GraphEdge(vertexB, vertexC, 6);

const edgeBD = new GraphEdge(vertexB, vertexD, 5);

const edgeEC = new GraphEdge(vertexE, vertexC, 8);

const edgeED = new GraphEdge(vertexE, vertexD, 2);

const edgeDC = new GraphEdge(vertexD, vertexC, 11);

const edgeDG = new GraphEdge(vertexD, vertexG, 10);

const edgeDF = new GraphEdge(vertexD, vertexF, 2);

const edgeFG = new GraphEdge(vertexF, vertexG, 3);

const edgeEG = new GraphEdge(vertexE, vertexG, 5);

const graph = new Graph();

graph

.addVertex(vertexH)

.addEdge(edgeAB)

.addEdge(edgeAE)

.addEdge(edgeAC)

.addEdge(edgeBC)

.addEdge(edgeBD)

.addEdge(edgeEC)

.addEdge(edgeED)

.addEdge(edgeDC)

.addEdge(edgeDG)

.addEdge(edgeDF)

.addEdge(edgeFG)

.addEdge(edgeEG);

const { distances, previousVertices } = bellmanFord(graph, vertexA);

expect(distances).toEqual({

H: Infinity,

A: 0,

B: 4,

E: 7,

C: 3,

D: 9,

G: 12,

F: 11,

});

expect(previousVertices.F.getKey()).toBe('D');

expect(previousVertices.D.getKey()).toBe('B');

expect(previousVertices.B.getKey()).toBe('A');

expect(previousVertices.G.getKey()).toBe('E');

expect(previousVertices.C.getKey()).toBe('A');

expect(previousVertices.A).toBeNull();

expect(previousVertices.H).toBeNull();

});

it('should find minimum paths to all vertices for directed graph with negative edge weights', () => {

const vertexS = new GraphVertex('S');

const vertexE = new GraphVertex('E');

const vertexA = new GraphVertex('A');

const vertexD = new GraphVertex('D');

const vertexB = new GraphVertex('B');

const vertexC = new GraphVertex('C');

const vertexH = new GraphVertex('H');

const edgeSE = new GraphEdge(vertexS, vertexE, 8);

const edgeSA = new GraphEdge(vertexS, vertexA, 10);

const edgeED = new GraphEdge(vertexE, vertexD, 1);

const edgeDA = new GraphEdge(vertexD, vertexA, -4);

const edgeDC = new GraphEdge(vertexD, vertexC, -1);

const edgeAC = new GraphEdge(vertexA, vertexC, 2);

const edgeCB = new GraphEdge(vertexC, vertexB, -2);

const edgeBA = new GraphEdge(vertexB, vertexA, 1);

const graph = new Graph(true);

graph

.addVertex(vertexH)

.addEdge(edgeSE)

.addEdge(edgeSA)

.addEdge(edgeED)

.addEdge(edgeDA)

.addEdge(edgeDC)

.addEdge(edgeAC)

.addEdge(edgeCB)

.addEdge(edgeBA);

const { distances, previousVertices } = bellmanFord(graph, vertexS);

expect(distances).toEqual({

H: Infinity,

S: 0,

A: 5,

B: 5,

C: 7,

D: 9,

E: 8,

});

expect(previousVertices.H).toBeNull();

expect(previousVertices.S).toBeNull();

expect(previousVertices.B.getKey()).toBe('C');

expect(previousVertices.C.getKey()).toBe('A');

expect(previousVertices.A.getKey()).toBe('D');

expect(previousVertices.D.getKey()).toBe('E');

});

});

/\*\*

\* @param {Graph} graph

\* @param {GraphVertex} startVertex

\* @return {{distances, previousVertices}}

\*/

export default function bellmanFord(graph, startVertex) {

const distances = {};

const previousVertices = {};

// Init all distances with infinity assuming that currently we can't reach

// any of the vertices except start one.

distances[startVertex.getKey()] = 0;

graph.getAllVertices().forEach((vertex) => {

previousVertices[vertex.getKey()] = null;

if (vertex.getKey() !== startVertex.getKey()) {

distances[vertex.getKey()] = Infinity;

}

});

// We need (|V| - 1) iterations.

for (let iteration = 0; iteration < (graph.getAllVertices().length - 1); iteration += 1) {

// During each iteration go through all vertices.

Object.keys(distances).forEach((vertexKey) => {

const vertex = graph.getVertexByKey(vertexKey);

// Go through all vertex edges.

graph.getNeighbors(vertex).forEach((neighbor) => {

const edge = graph.findEdge(vertex, neighbor);

// Find out if the distance to the neighbor is shorter in this iteration

// then in the previous one.

const distanceToVertex = distances[vertex.getKey()];

const distanceToNeighbor = distanceToVertex + edge.weight;

if (distanceToNeighbor < distances[neighbor.getKey()]) {

distances[neighbor.getKey()] = distanceToNeighbor;

previousVertices[neighbor.getKey()] = vertex;

}

});

});

}

return {

distances,

previousVertices,

};

}