Bharatiya Vidya Bhavan's
Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

Computer Engineering Department &
Information Technology Engineering Department

Academic Year: 2021-2022

Class: S.Y.B.Tech **Sem.:** 4 **Course:** DAA

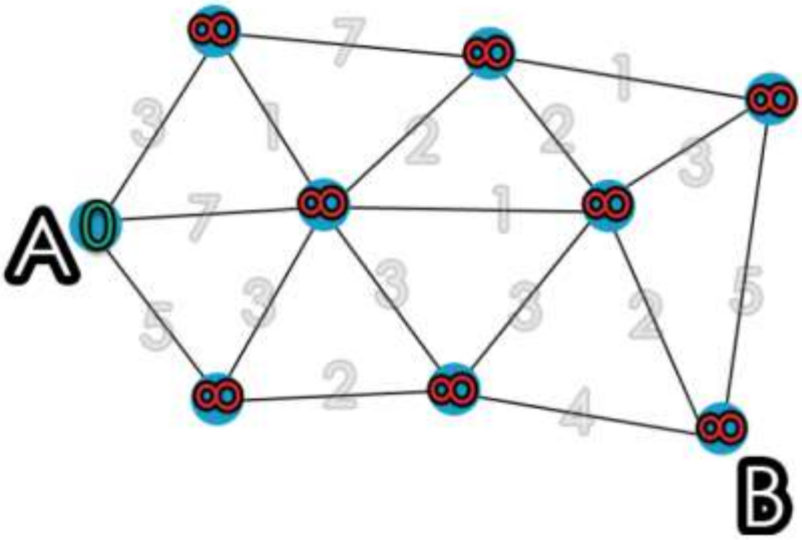
Name	**Pratik Pujari**		
UID no.	**2020300054**	**Class:**	**Comps C Batch**
Experiment No.	6		

AIM:	To implement Dijkstra algorithm
THEORY:	**What is BackTracking?** Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach. It consists of building a set of all the solutions incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed. It uses recursive calling to find a solution set by building a solution step by step, increasing levels with time. In order to find these solutions, a search tree named state-space tree is used. In a state-space tree, each branch is a variable, and each level represents a solution. A backtracking algorithm uses the depth-first search method. When it starts exploring the solutions, a bounding function is applied so that the algorithm can check if the so-far built solution satisfies the constraints. If it does, it continues searching. If it doesn't, the branch would be eliminated, and the algorithm goes back to the level before. **When to Use a Backtracking Algorithm** The backtracking algorithm is applied to some specific types of problems. For instance, we can use it to find a feasible solution to a decision problem. It was also found to be very effective for optimization problems. For some cases, a backtracking algorithm is used for the enumeration problem in order to find the set of all feasible solutions for the problem.

On the other hand, backtracking is not considered an optimized technique to solve a problem. It finds its application when the solution needed for a problem is not time-bounded.

Dijkstra's Algorithm

Dijkstra's algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.



The graph has the following:

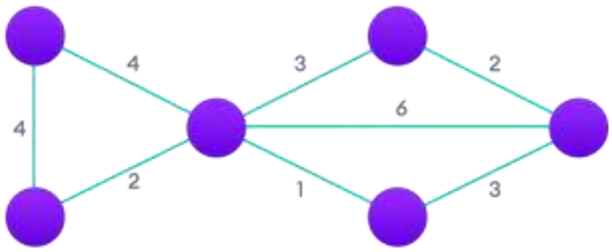
- vertices, or nodes, denoted in the algorithm by v*v* or u*u*;
- weighted edges that connect two nodes: (u,v*u*,*v*) denotes an edge, and w(u,v)*w*(*u*,*v*) denotes its weight. In the diagram on the right, the weight for each edge is written in gray.

This is done by initializing three values:

- dist*dist*, an array of distances from the source node s*s* to each node in the graph, initialized the following way: dist*dist*(s*s*) = 0; and for all other nodes v*v*, dist*dist*(v*v*) = \infty∞. This is done at the beginning because as the algorithm proceeds, the dist*dist* from the source to each node v*v* in the graph will be recalculated and finalized when the shortest distance to v*v* is found

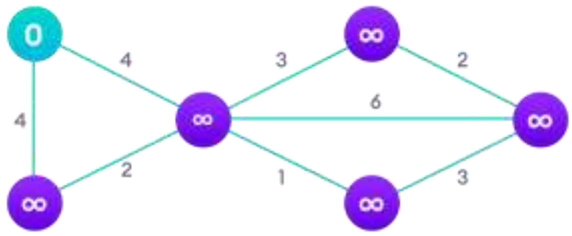
- Q*Q*, a queue of all nodes in the graph. At the end of the algorithm's progress, Q*Q* will be empty.
- S*S*, an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, S*S* will contain all the nodes of the graph.

It is easier to start with an example and then think about the algorithm.



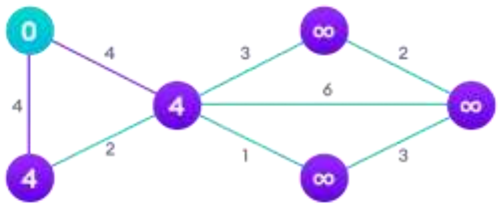
Step: 1

Start with a weighted graph



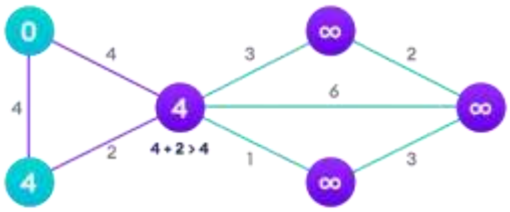
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



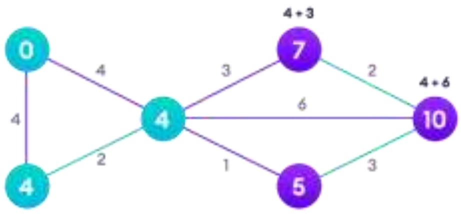
Step: 3

Go to each vertex and update its path length



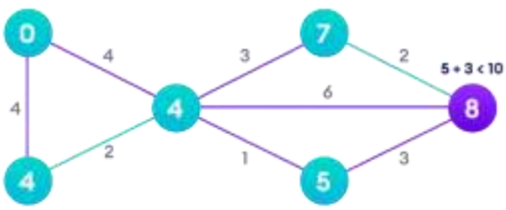
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



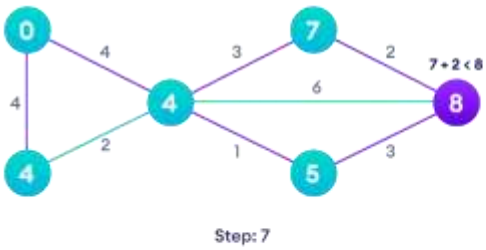
Step: 5

Avoid updating path lengths of already visited vertices

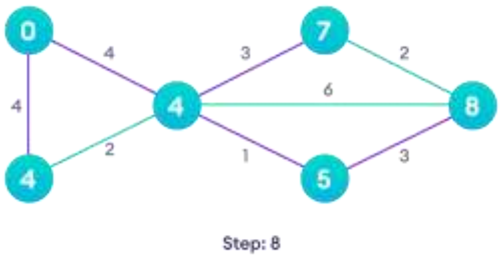


Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Notice how the rightmost vertex has its path length updated twice



Repeat until all the vertices have been visited

PSEUDOCODE:

```
function Dijkstra(Graph, source):
for each vertex v in Graph:    // Initialization
dist[v] := infinity // initial distance from source to vertex v is set to infinite
previous[v] := undefined      // Previous node in optimal path from source
dist[source] := 0 // Distance from source to source
Q := the set of all nodes in Graph   // all nodes in the graph are unoptimized - thus are in Q
while Q is not empty:    // main loop
u := node in Q with smallest dist[ ]
remove u from Q
for each neighbor v of u:       // where v has not yet been removed from Q.
alt := dist[u] + dist_between(u, v)
if alt < dist[v]      // Relax (u,v)
```

```
dist[v] := alt
previous[v] := u
return previous[ ]
```

EXPERIMENT 1

CODE:

```
import java.util.*;

public class dijkstra {

    int source;

    // The main method is where the program starts.
    void dijkstra_solve(int[][] graph) {
        int count = graph.length;
        boolean[] visited = new boolean[count];
        int[] distance = new int[count];

        for (int i = 0; i < distance.length; i++) {
            distance[i] = Integer.MAX_VALUE;
        }
        distance[source] = 0;
        for (int k = 0; k < distance.length - 1; k++) {

            int minVertex = findMin(distance, visited);
            visited[minVertex] = true;
            // explore the neighbours
            for (int i = 0; i < distance.length; i++) {
                if (graph[minVertex][i] != 0 &&
distance[minVertex] != Integer.MAX_VALUE) {
                    // checking if the there exists an edge
between the two vertices, the neighbour
                    // should not be visited
                    // adding the weight of the edge to the
distance of the min vertex
                    int newDistance = distance[minVertex] +
graph[minVertex][i]; // Relaxation
```

	`` if (newDistance < distance[i]) { // updating the distance of the vertex if the value is lesser than the // previous value of the same vertex distance[i] = newDistance; } } } } System.out.println("\nOutput\n(Vertex-> Distance):"); for (int i = 0; i < distance.length; i++) { if (distance[i] == Integer.MAX_VALUE || distance[i] == 0) { continue; } System.out.println(i + "\t" + distance[i]); } } // finding the minimum distance vertex static int findMin(int[] distance, boolean[] visited) { int minVertex = -1; // initializing the minimum vertex to -1 for (int i = 1; i < distance.length; i++) { // if the vertex is not visited and the distance is lesser than the min vertex if ((minVertex == -1 || distance[i] < distance[minVertex]) && !visited[i]) { minVertex = i; } } return minVertex; // returning the minimum vertex } public static void main(String[] args) throws Exception { try (// Driver code Scanner sc = new Scanner(System.in)) { System.out.println("----------------Dijkstra's Algorithm----------------"); ```

```
        System.out.println("\nInput(TestCases-> Vertices->
Edges-> Each edge with weights)\n");
        int testCases = sc.nextInt();
        dijkstra T = new dijkstra();
        int negativeChecker = 0;
        for (int i = 0; i < testCases; i++) {
            int v = sc.nextInt(); // vertices
            int e = sc.nextInt(); // edges
            int[][] graph = new int[1024][1024]; //
adjacency matrix
            int src = sc.nextInt();
            T.source = src;
            int dest = sc.nextInt();
            int cost = sc.nextInt(); // cost of the edge // set
to store the vertices
            // if cost is negative, then the edge is not added
            graph[src][dest] = cost;
            if (cost < 0) {
                negativeChecker = 1;
                System.out.print("\nNegative edge not
Added");
                continue;
            }
            for (int j = 0; j < e - 1; j++) {
                int p = sc.nextInt(); // source
                int q = sc.nextInt(); // destination
                cost = sc.nextInt(); // cost of the edge
                if (cost < 0) {
                    negativeChecker = 1;
                    System.out.print("\nNegative edge not
allowed");
                    break;
                }

                graph[p][q] = cost;
            }
            if (negativeChecker != 1)
                T.dijkstra_solve(graph);
        }
```

```
        sc.close();
    }
  }
}
```

OUTPUT:

```
Input(TestCases-> Vertices-> Edges-> Each e

1
3 3
5 7 1
6 5 2
7 6 3

Output
(Vertex-> Distance):
6      4
7      1
```

```
1
6 5
1 2 15
1 3 5
3 2 6
2 4 2
5 6 7

Output
(Vertex-> Distance):
2       11
3       5
4       13
```

```
1
6 7
10 20 3
20 40 -1
40 4 -2
4 80 1
4 2 -2
80 2 -4

Negative edge not allowed
```

```
1
7 7
15 3 2
15 19 5
3 14 3
19 8 2
19 10 4
14 8 1
8 16 5

Output
(Vertex-> Distance):
3       2
8       6
10      9
14      5
16      11
19      5
```

```
1
8 13
8 7 4
8 6 4
5 8 3
5 6 2
6 3 -2
6 4 4
3 7 3
3 4 -3
4 5 1
4 2 -2
1 4 -2
2 1 2

Negative edge not allowed
```

Computer Engineering Department & Information Technology Engineering Department

Academic Year: 2021-2022

Class: S.Y.B.Tech **Sem.:** 4 **Course:** DAA

```
Output
(Vertex-> Distance):
1	461
2	417
3	443
4	522
5	498
6	427
7	441
8	207
9	476
10	437
11	549
12	481
13	448
14	348
15	397
16	469
17	409
18	438
19	473
20	437
21	515
23	430
24	518
25	461
26	533
27	397
28	392
29	481
32	462
33	386
34	349
35	399
36	425
37	474
38	384
39	444
40	470
41	442
43	391
44	443
45	412
46	487
47	413
48	429
49	435
50	411
51	506
52	434
53	489
54	493
55	474
56	491
57	403
58	462
60	434
61	490
62	471
64	477
66	457
67	96
68	452
69	453
70	431
71	388
72	418
73	446
74	494
75	126
76	499
77	354
78	342
79	476
80	607
81	436
82	456
83	461
84	519
86	394
87	396
88	507
89	477
90	467
91	382
92	460
93	414
94	432
95	201
96	345
97	496
98	424
99	420
100	446
101	524
102	390
103	418
104	459
105	499
106	466
107	482
108	460
109	472
110	449
111	428
112	516
113	421
114	288
115	457
116	494
117	555
118	515
119	494
120	483
121	408
122	465
123	391
124	461
125	610
126	450
127	457
128	399
129	473
130	473
131	460
132	414
133	418
134	408
135	607
136	429
137	442
138	424
139	461
140	624
141	492
142	355
144	436
145	343
146	412
147	433
148	445
149	389
150	393
152	448
153	576
154	447
155	474
157	448
158	408
159	478
160	381
161	412
162	438
163	453
164	508
165	465
166	392
167	492
168	492
169	456
170	416
171	525
172	459
173	429
174	452
175	441
176	409
177	479
178	476
179	399
180	502
181	400
183	545
184	557
185	390
186	536
187	467
188	465
189	438
190	495
191	404
192	400
193	469
195	459
196	369
197	447
198	480
199	524
200	413
```

Bharatiya Vidya Bhavan's

Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

Computer Engineering Department &
Information Technology Engineering Department

Academic Year: 2021-2022

Class: S.Y.B.Tech **Sem.:** 4 **Course:** DAA

201	477
202	623
203	468
204	312
205	384
206	531
207	515
209	468
210	371
211	420
212	415
213	355
214	485
217	509
218	511
219	438
220	464
221	385
222	546
223	461
224	527
226	477
227	517
228	432
229	435
230	454
232	443
233	442
234	471
235	483
236	468
237	490
238	475
240	386
241	394
242	498
244	483
245	464
246	436
247	380
249	532
250	356
251	39
252	443
253	491
254	456
255	513
256	441
257	409
258	497
259	449
260	440
261	447
262	498
263	451
264	456
265	472
266	361
267	383
268	476
269	395
270	530
271	427
272	528
274	393
275	402
277	490
278	415
279	513
280	446
281	522
282	493
283	444
284	426
285	451
286	412
287	503
288	444
289	463
291	394
292	452
293	514
294	431
295	552
296	432
297	401

Bharatiya Vidya Bhavan's

Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

Computer Engineering Department & Information Technology Engineering Department

Academic Year: 2021-2022

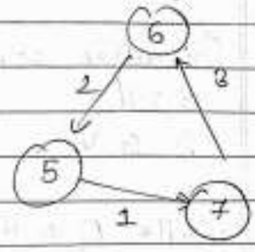
Class: S.Y.B.Tech **Sem.:** 4 **Course:** DAA

Written Sum for the Dijkstra

Pratik Pujari 2020300054

Page No.
Date

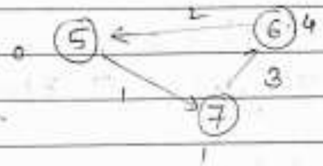
No. of vertices: 3
No. of edges: 4



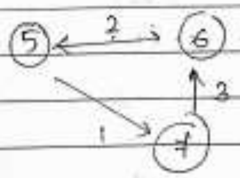
Edge 1	Edge 2	Distance
5	7	1
6	5	2
7	6	3

Source Vertex = 5
Intial dist [5] = 0

Path chosen: 5 7 (wt: 1)



Path choosen: 7, 6 (~~cost~~ wt: 3)



Distance
5 → 0
6 → 4
7 → 1

Bharatiya Vidya Bhavan's
Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

Computer Engineering Department &
Information Technology Engineering Department

Academic Year: 2021-2022

Class: S.Y.B.Tech **Sem.:** 4 **Course:** DAA

TIME COMPLEXITY:

Pratik Pujari

Page No
Date

Dijkstra time complexity (Graph, Weight, Source)

Intialize source $(G,S) \rightarrow O(V)$
$S = \phi \rightarrow O(1)$
$P = G \cdot V \rightarrow O(V)$ min heap

while $Q \neq \phi$
u = Extract min (Q) $O(\log V)$
$S = S \cup \{u\}$
for each vertex $v \in G$ Adj[u]
Relax (u,v,w)
$O(w \log v)$

Overall $TC = O(V) + O(1) + O(V) + O(V\log V) + O(E\log V)$
$= O((V+E)\log V)$

Matrix	Cost	Time
for $i=1$	c_1	V
int n = minDist()	c_2	V
visited[v] = true	c_3	V
for $v=1 \rightarrow V$	c_4	V^2
if (distance[v] + graph[u][v] < dist[v])	c_5	V^2

$T_n = c_1V + c_2V + c_3V + c_4V^2 + c_5V^2$
$= O(V^2)$

Both the time and space complexity obtained for the Dijkstra's algorithm is O(V*V) where V is the number of vertices. While if an adjacency list is used the time complexity will come out to be O(E*logV) where E is the number of edges. Since the Dijkstra's algorithm works on

	Greedy approach, it has a limitation that it doesn't work for the negative values of the distances because negative value and since Dijkstra works on the principle that once a vertex is discovered, it can't go back. Thus, it can't be used for the negative values

CONCLUSION: Learnt during the procedural programming of solving the problem

- Learnt about BackTracking and Dijsktra Algorithm
- Learnt how to use adjacency matrix in order to store the graph
- Learnt about time and space complexity of the shortest path algorithm
- Learnt about the advantages and disadvantages of dijsktra algorithm
- Learnt about the applications of a Dijsktra/ shortest path finder algorithm