EXPERIMENT NO 6

Title: Implement single Source Shortest Path using Dijkstra's algorithm by applying greedy approach and analyze its complexity.

Aim: Implement single source shortest paths to other vertices using Dijkstra's algorithm.

Lab Outcomes : CSL401.2: Ability to implement the algorithm using Greedy approach.

Theory:
The Dijkstra algorithm is also called the single source shortest path algorithm. It is based on greedy technique. The algorithm maintains a list visited[ ] of vertices, whose shortest distance from the source is already known. If visited[1], equals 1, then the shortest distance of vertex i is already known. Initially, visited[i] is marked as, for source vertex. At each step, we mark visited[v] as 1. Vertex v is a vertex at the shortest distance from the source vertex. At each step of the algorithm, the shortest distance of each vertex is stored in an array distance[ ].

Algorithm:

```
INITIALIZE-SINGLE-SOURCE(G,s)
1.  for each vertex v ∈ G.V
2.      v.d = ∞
3.      v.pi = NIL
4.  s.d = 0
```

```
RELAX(u,v,w)
1.  if v.d > u.d + w(u,v)
2.      v.d = u.d + w(u,v)
3.      v.pi = u
```

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Dijkstra’s using adjacency matrix

1. Create cost matrix C[ ][ ] from adjacency matrix adj[ ][ ]. C[i][j] is the cost of going from vertex i to vertex j. If there is no edge between vertices i and j then C[i][j] is infinite.
2. Array visited[ ] is initialized to zero.

for(i=0;i<n;i++)
visited[i]=0;

3. If the vertex 0 is the source vertex then visited[0] is marked as 1.
4. Create the distance matrix, by storing the cost of vertices from vertex no. 0 to n-1 from the source vertex 0.

for(i=1;i<n;i++)
distance[i]=cost[0][i];

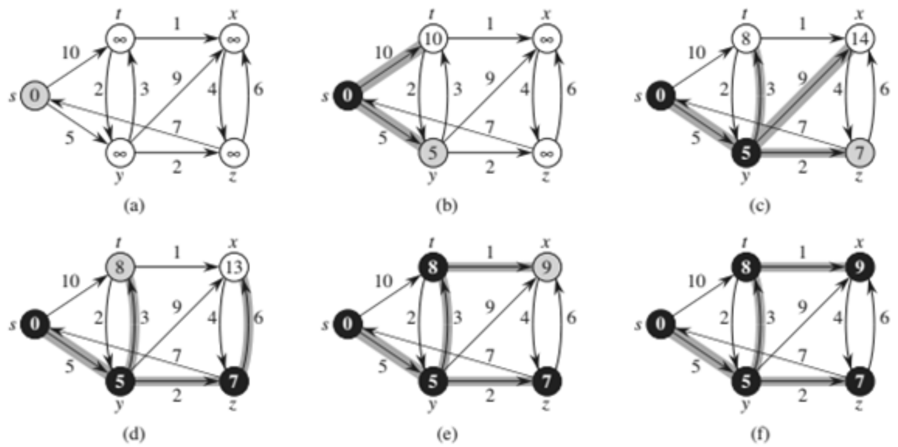
Initially, distance of source vertex is taken as 0. i.e. distance[0]=0;

5. for(i=1;i<n;i++)

– Choose a vertex w, such that distance[w] is minimum and visited[w] is 0. Mark visited[w] as 1.
– Recalculate the shortest distance of remaining vertices from the source.
– Only, the vertices not marked as 1 in array visited[ ] should be considered for recalculation of distance. i.e. for each vertex v

```
if(visited[v]==0)
        distance[v]=min(distance[v],
        distance[w]+cost[w][v])
```

Example:



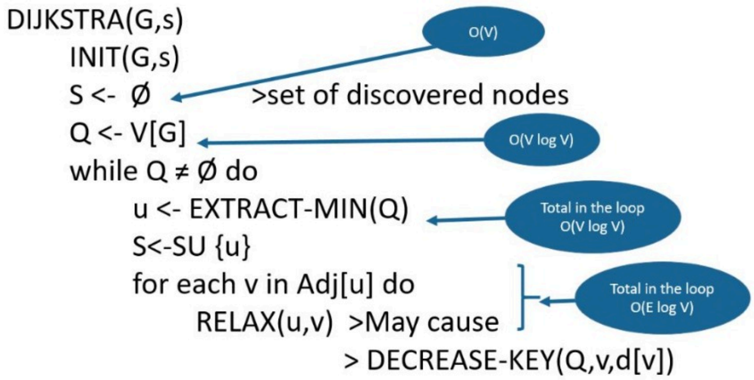
Time Complexity

The time complexity of the given code/algorithm looks O(V^2) as there are two nested while loops. If we take a closer look, we can observe that the statements in the inner loop are executed O(V+E) times (similar to BFS). The inner loop has decreaseKey() operation which takes O(LogV) time. So overall time complexity is O(E+V)*O(LogV) which is O((E+V)*LogV) = O(ELogV) Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to O(E + VLogV) using Fibonacci Heap. The reason is, Fibonacci Heap takes O(1) time for decrease-key operation while Binary Heap takes O(Log n) time.

Best and Worst Cases for Dijkstra's Algorithm

The worst case (where log denotes the binary logarithm log 2) for connected graphs this time bound can be simplified to Θ (|E|log |V|) . The Fibonacci heap improves this to O(|E| + |V| * log |V|).
When using binary heaps, the average case time complexity is lower than the worst-case: assuming edge costs are drawn independently from a common probability distribution, the expected number of decrease-key operations is bounded by O(|V| * log(|E| / |V|)), giving a total running time of O(|E| + |V| * log |E| / |V| * log |V|).

Time complexity of Dijkstra's Algorithm



Conclusion: Thus we have implemented single source shortest paths to other vertices using Dijkatra's algorithm.