**Experiment No. : 3**

# Title: Implement Dijkstra`s Algorithm using Greedy approach

**Batch: A1 Roll No.: 16010422013**
**Experiment No.: 3**

**Aim:** To implement and analyse time complexity Dijkstra`s Algorithm to find Shortest path.

**Algorithm of Dijkstra Algorithm:**

```
function Dijkstra(Graph, start):
   create priority queue Q

   for each vertex v in Graph:
      dist[v] = INFINITY
      previous[v] = UNDEFINED
      add v to Q

   dist[start] = 0

   while Q is not empty:
      u = vertex in Q with smallest distance in dist[]
      remove u from Q

      for each neighbor v of u:
         alt = dist[u] + length(u, v)
         if alt < dist[v]:
            dist[v] = alt
            previous[v] = u

   return dist[], previous[]
```

K. J. SOMAIYA COLLEGE OF ENGG.

**Explanation and Working of Dijkstra Algorithm:**

**Dijkstra's algorithm is a graph search algorithm used to find the shortest path from a source node to all other nodes in a graph with non-negative edge weights. The algorithm initializes with setting tentative distances to infinity for all nodes except the source, which is set to 0. It then iteratively selects the unvisited node with the smallest tentative distance, evaluates its neighbors, and updates their tentative distances if a shorter path is found. The selected node is marked as visited, and the process repeats until all nodes are visited or the destination is reached. Dijkstra's algorithm is greedy, always choosing the locally optimal solution at each step. The result is a distance array and previous node information, which can be used to reconstruct the shortest paths from the source to all other nodes.**

**Time complexity and derivation of Dijkstra Algorithm:**

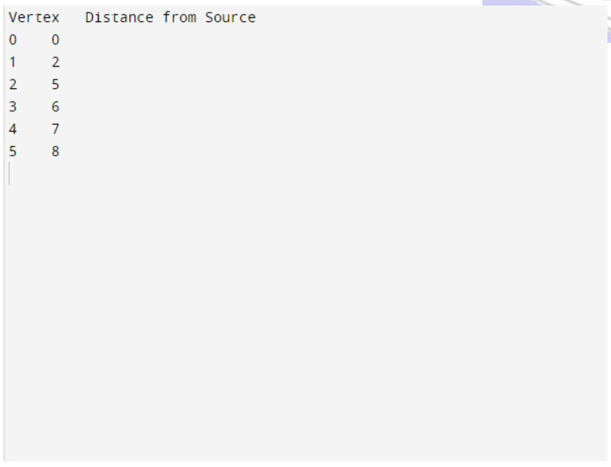
**Dijkstra's algorithm is used for finding the shortest paths between nodes in a graph. The time complexity depends on the data structures used. The algorithm involves initializing a priority queue, processing vertices by extracting the one with the minimum distance, and updating distances for neighboring vertices. The time complexity is O((V + E) * log(V)), where V is the number of vertices and E is the number of edges, assuming a binary heap is used for the priority queue. The algorithm terminates when all vertices are visited or the destination is reached. Note that Dijkstra's algorithm assumes non-negative edge weights.**

**Program(s) of Dijkstra Algorithm:**

```
#include <stdio.h>
#include <limits.h>

#define V 6
struct HeapNode {
   int vertex;
   int distance;
};

void swap(struct HeapNode* a, struct HeapNode* b) {
   struct HeapNode temp = *a;
   *a = *b;
   *b = temp;
}


void minHeapify(struct HeapNode heap[], int size, int idx) {
   int smallest = idx;
   int left = 2 * idx + 1;
   int right = 2 * idx + 2;

   if (left < size && heap[left].distance < heap[smallest].distance)
      smallest = left;

   if (right < size && heap[right].distance < heap[smallest].distance)
      smallest = right;

   if (smallest != idx) {
      swap(&heap[idx], &heap[smallest]);
      minHeapify(heap, size, smallest);
   }
}
```

```
struct HeapNode extractMin(struct HeapNode heap[], int* size) {
    struct HeapNode minNode = heap[0];
    heap[0] = heap[*size - 1];
    (*size)--;
    minHeapify(heap, *size, 0);
    return minNode;
}


void decreaseKey(struct HeapNode heap[], int v, int distance) {
    int i;
    for (i = 0; i < V; i++) {
        if (heap[i].vertex == v)
            break;
    }

    heap[i].distance = distance;

    while (i != 0 && heap[(i - 1) / 2].distance > heap[i].distance) {
        swap(&heap[i], &heap[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

void printSolution(int dist[]) {
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t %d\n", i, dist[i]);
}

void dijkstra(int graph[V][V], int src) {
    int dist[V];
    struct HeapNode heap[V];
    int heapSize = V;

    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        heap[i].vertex = i;
        heap[i].distance = INT_MAX;
    }

    heap[src].distance = 0;
    dist[src] = 0;

    while (heapSize != 0) {
        struct HeapNode u = extractMin(heap, &heapSize);
        int uVertex = u.vertex;

        for (int v = 0; v < V; v++) {
            if (graph[uVertex][v] && dist[uVertex] != INT_MAX &&
                dist[uVertex] + graph[uVertex][v] < dist[v]) {

                dist[v] = dist[uVertex] + graph[uVertex][v];
                decreaseKey(heap, v, dist[v]);
            }
        }
    }

    printSolution(dist);
}

int main() {
    int graph[V][V] = {{0, 2, 0, 6, 0, 0},
                       {2, 0, 3, 8, 5, 0},
                       {0, 3, 0, 0, 7, 0},
                       {6, 8, 0, 0, 9, 4},
                       {0, 5, 7, 9, 0, 1},
                       {0, 0, 0, 4, 1, 0}};


    dijkstra(graph, 0);

    return 0;
}
```

**Output(o) of Dijkstra Algorithm:**



```
Vertex   Distance from Source
0     0
1     2
2     5
3     6
4     7
5     8
```

**Conclusion: (Based on the observations):**

Dijkstra's algorithm is a widely used method for determining the shortest paths between nodes in a graph. It begins by initializing a priority queue and setting distances to the source vertex. The algorithm then processes vertices, extracting the one with the minimum distance and updating distances for its neighbors. The time complexity hinges on the implementation of the priority queue, typically O((V + E) * log(V)) for a binary heap. The algorithm terminates when all vertices are visited or the destination is reached. Crucially, it assumes non-negative edge weights for accurate results. While efficient for non-negative weights, Dijkstra's algorithm is less suitable for graphs with negative weights, where Bellman-Ford might be a better choice. Its widespread applicability makes it a fundamental tool in network routing and optimization problems. Careful consideration of the underlying data structures ensures optimal performance. In summary, Dijkstra's algorithm provides a robust solution for finding shortest paths in graphs with non-negative edge weights, addressing key challenges in network analysis and optimization.

**Outcome: Implement Greedy Programming Algorithms.**

**References:**

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.