Experiment No. : 3

Title: Implement Dijkstra`s Algorithm using Greedy approach

Batch: B-4 **Roll No.: 16010422234** **Name: Chandana Ramesh Galgali**

Experiment No.: 03

Aim: To implement and analyze time complexity Dijkstra's Algorithm to find Shortest path.

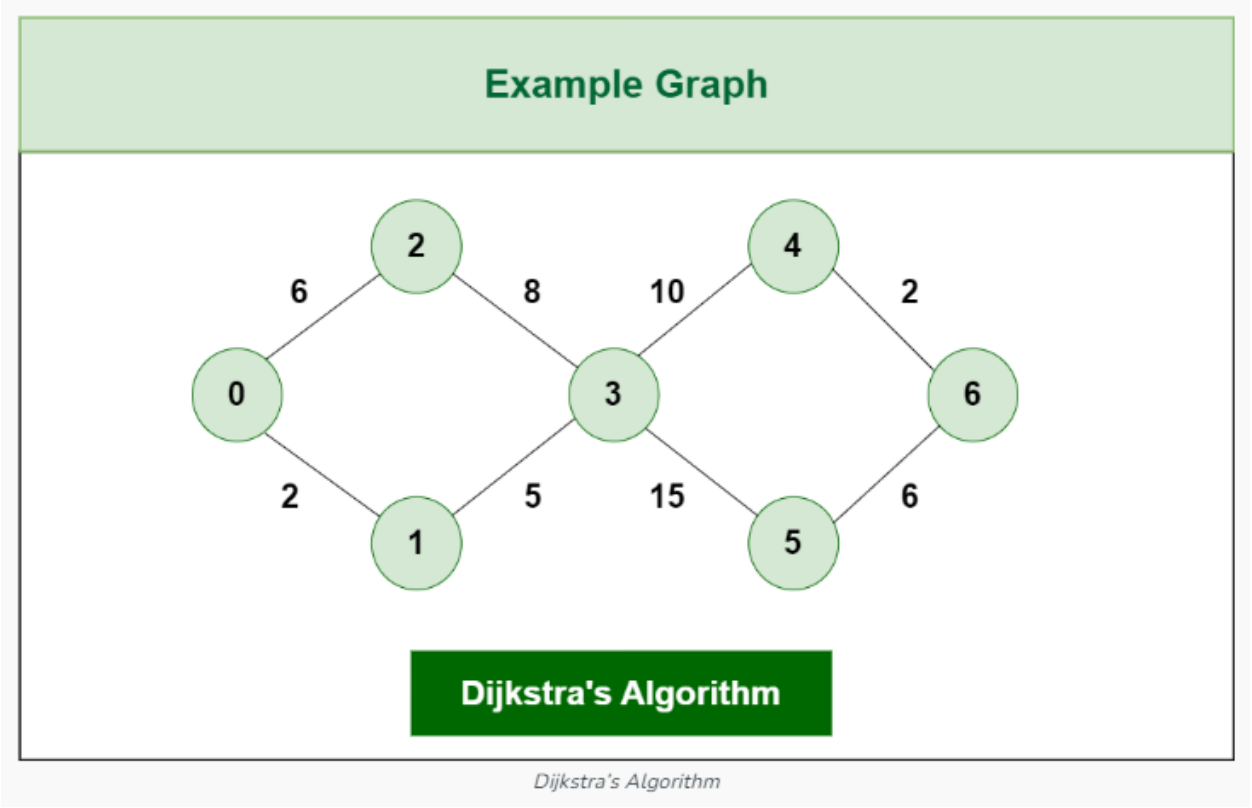
Algorithm of Dijkstra Algorithm:

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes that are unvisited.

Explanation and Working of Dijkstra Algorithm:

Let's see how Dijkstra's Algorithm works with an example given below:
Dijkstra's Algorithm will generate the shortest path from Node 0 to all other Nodes in the graph.
Consider the below graph:

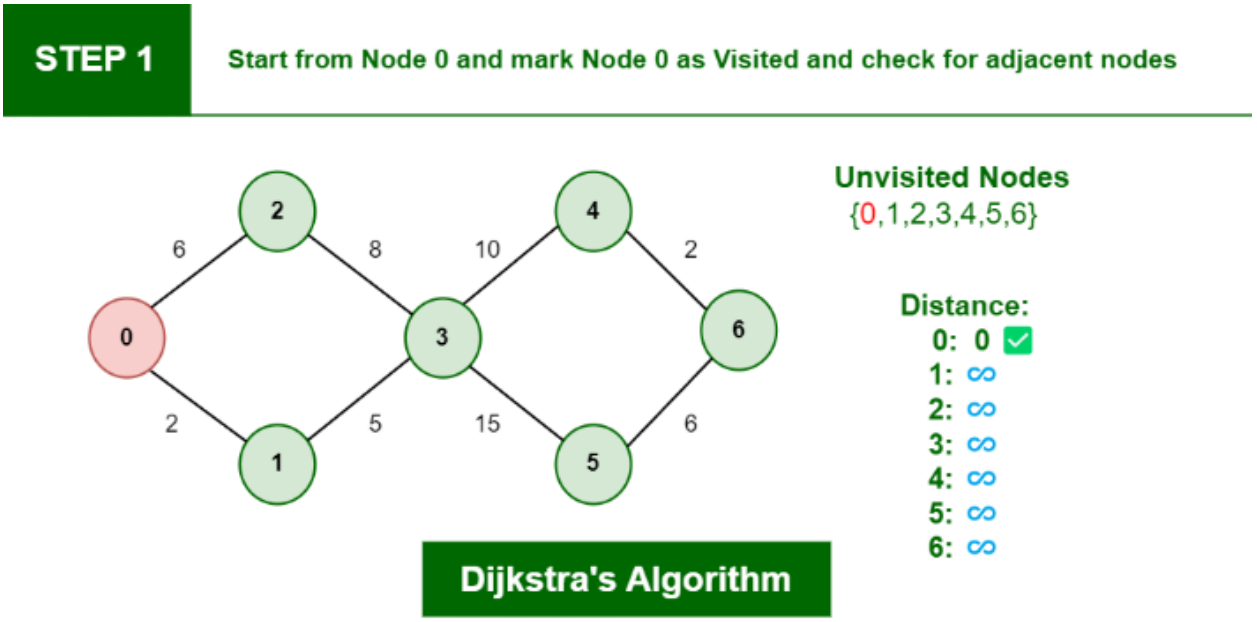


Dijkstra's Algorithm

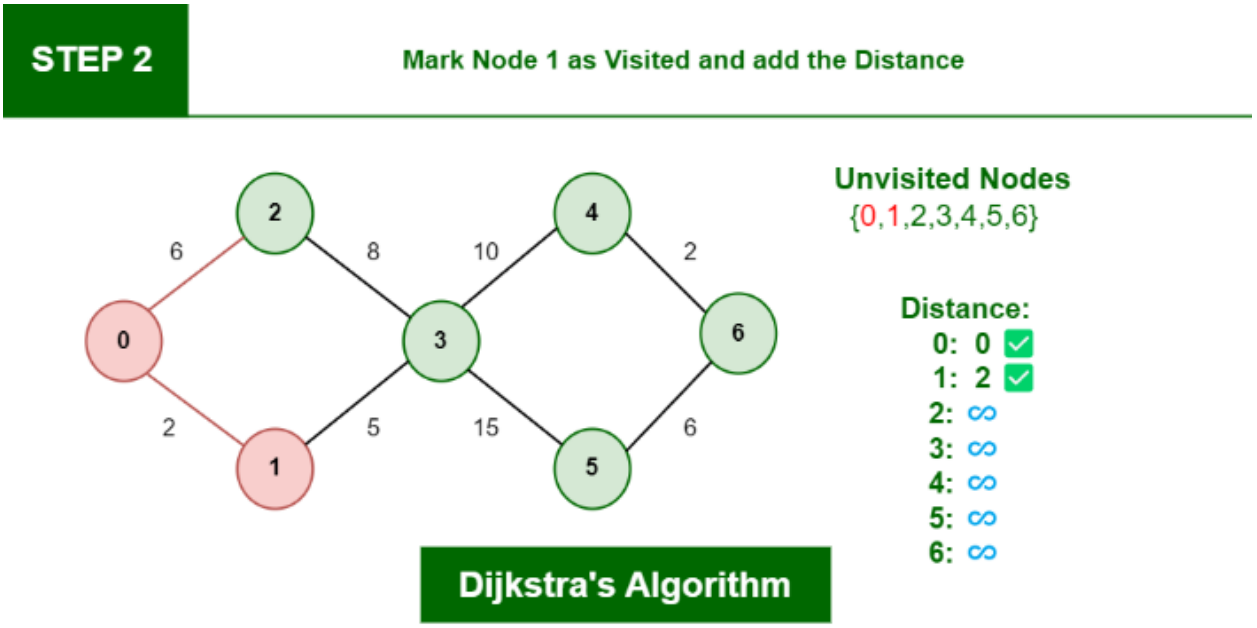
The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.
For this graph, we will assume that the weight of the edges represents the distance between two nodes.

As, we can see we have the shortest path from,
Node 0 to Node 1, from
Node 0 to Node 2, from
Node 0 to Node 3, from
Node 0 to Node 4, from
Node 0 to Node 6.
Initially we have a set of resources given below :
The Distance from the source node to itself is 0. In this example the source node is 0.
The distance from the source node to all other node is unknown so we mark all of them as infinity.
Example: $0 \ -> \ 0, 1 \ -> \ \infty, 2 \ -> \ \infty, 3 \ -> \ \infty, 4 \ -> \ \infty, 5 \ -> \ \infty, 6 \ -> \ \infty.$
We'll also have an array of unvisited elements that will keep track of unvisited or unmarked Nodes.
Algorithm will complete when all the nodes marked as visited and the distance between them are added to the path. Unvisited Nodes:- 0 1 2 3 4 5 6.

Step 1: Start from Node 0 and mark Node as visited as you can check in below image visited Node is marked red.

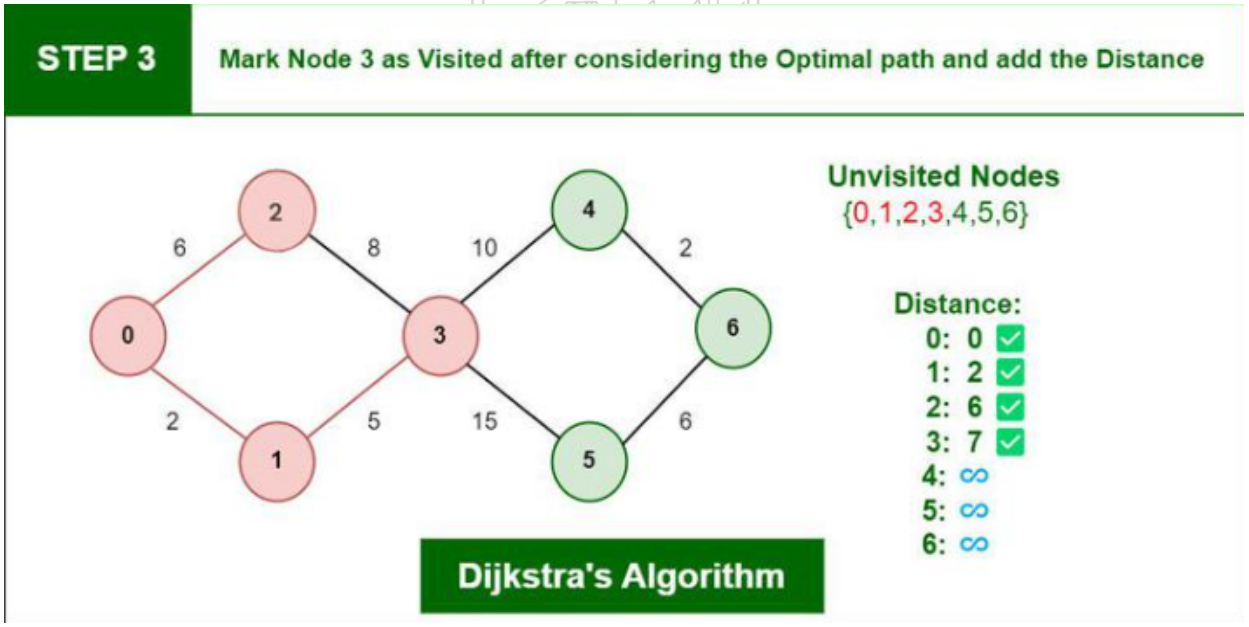


Step 2: Check for adjacent Nodes, Now we have to choices (Either choose Node1 with distance 2 or either choose Node 2 with distance 6) and choose Node with minimum distance. In this step Node 1 is Minimum distance adjacent Node, so marked it as visited and add up the distance.
Distance: $Node \ 0 \ -> \ Node \ 1 \ = \ 2$



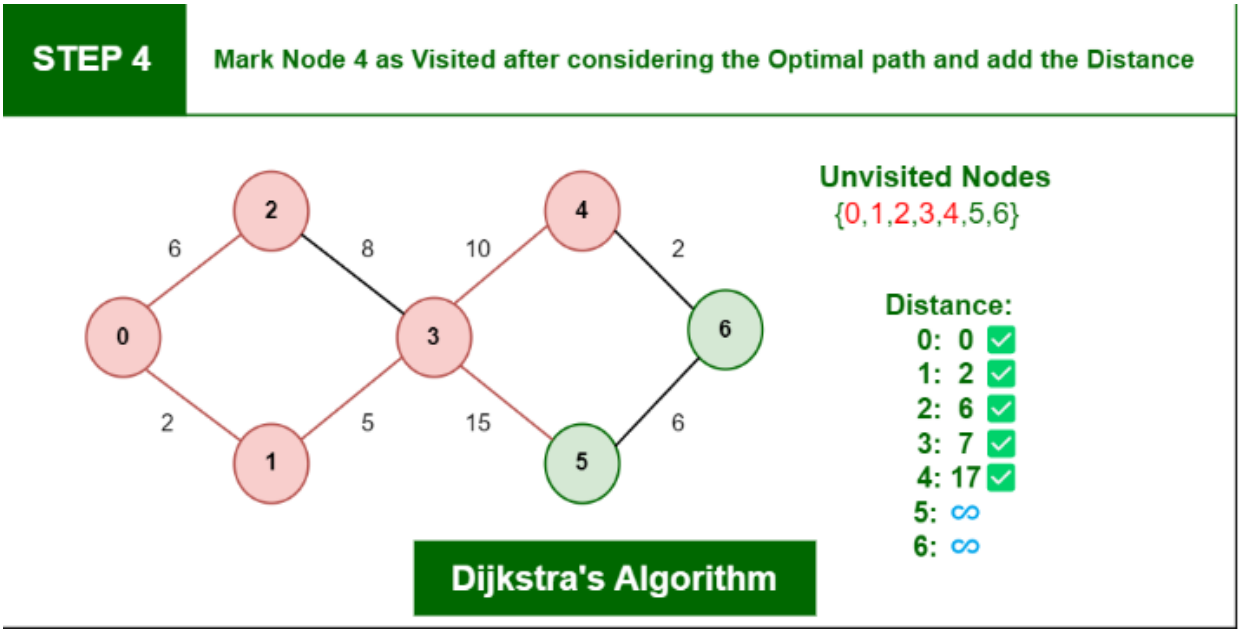
Step 3: Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:

Distance: $Node\ 0\ -> Node\ 1\ -> Node\ 3 = 2 + 5 = 7$



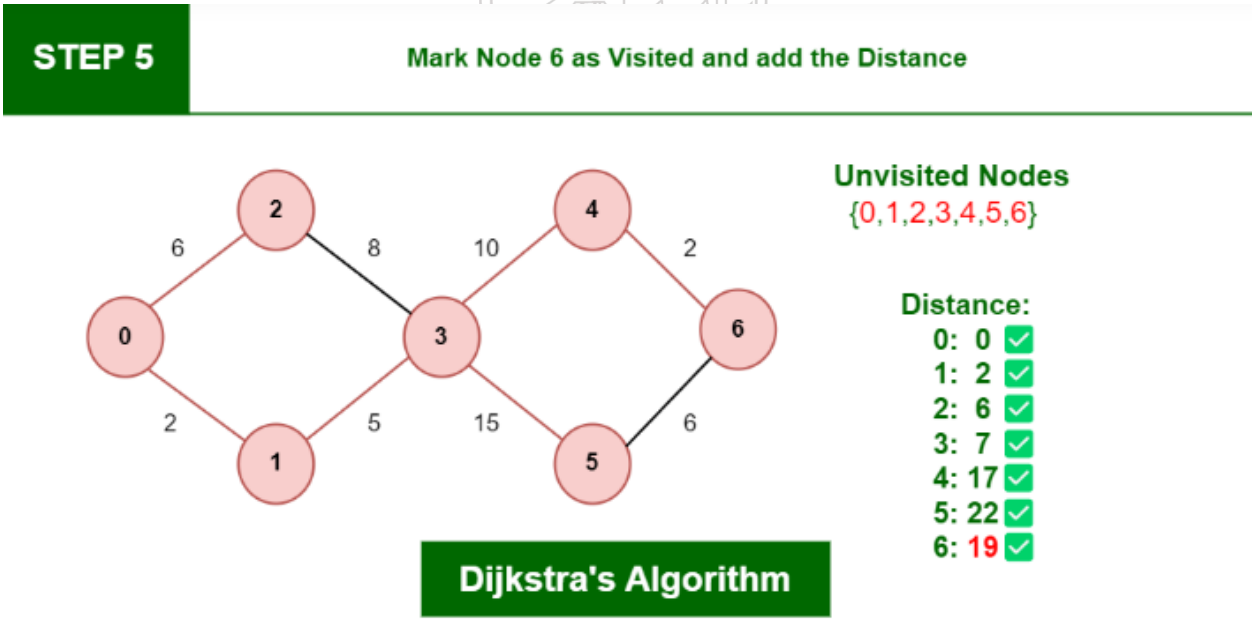
Step 4: Again we have two choices for adjacent Nodes (Either we can choose Node 4 with distance 10 or either we can choose Node 5 with distance 15) so choose Node with minimum distance. In this step Node 4 is Minimum distance adjacent Node, so marked it as visited and add up the distance.

Distance: $Node\ 0\ -> Node\ 1\ -> Node\ 3\ -> Node\ 4 = 2 + 5 + 10 = 17$



Step 5: Again, Move Forward and check for adjacent Node which is Node 6, so marked it as visited and add up the distance, Now the distance will be:
Distance:

$$Node\ 0 \ -> \ Node\ 1 \ -> Node\ 3 \ -> Node\ 4 \ -> \ Node\ 6 = 2 + 5 \ + 10 \ + 2 \ = 19$$



So, the Shortest Distance from the Source Vertex is 19 which is optimal one

Time complexity and derivation of Dijkstra Algorithm:

The time complexity of Dijkstra's algorithm can be derived based on its steps:

1. Initialization: Setting the distance of all nodes to infinity and marking the source node's distance as 0 can be done in $O(V)$ time, where V is the number of vertices in the graph.

2. Finding the node with the smallest current distance: This operation can be performed efficiently using a priority queue or a min-heap. Extracting the minimum element from a priority

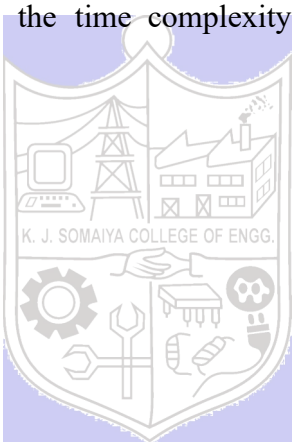
queue takes $O(log\ V)$ time, and since there can be at most V iterations (each node is processed once), this step takes $O(V\ log\ V)$ time.

3. Relaxation: For each neighbor of the current node, updating the distance if a shorter path is found involves constant time operations (comparisons and addition), which results in $O(1)$ time per edge. Since each edge is examined exactly once, the total time complexity for this step is $O(E)$, where E is the number of edges in the graph.

4. Marking a node as visited: This can be done in constant time.

5. Checking for unvisited nodes: This involves checking if there are any unvisited nodes left. In the worst case, this check might take $O(V)$ time if unvisited nodes are not tracked efficiently.

Overall, the time complexity of Dijkstra's algorithm depends on the implementation and the data structures used. If a priority queue or a min-heap is used to efficiently find the node with the smallest current distance, the dominant factor in the time complexity is usually the $O(V\ log\ V)$ term from step 2. Hence, the overall time complexity of Dijkstra's algorithm with a proper implementation is $O(V\ log\ V\ +\ E)$. However, if a simple array or list is used to find the node with the smallest current distance, the time complexity can be higher, reaching $O(V^2)$ or $O(V^2\ +\ E)$.



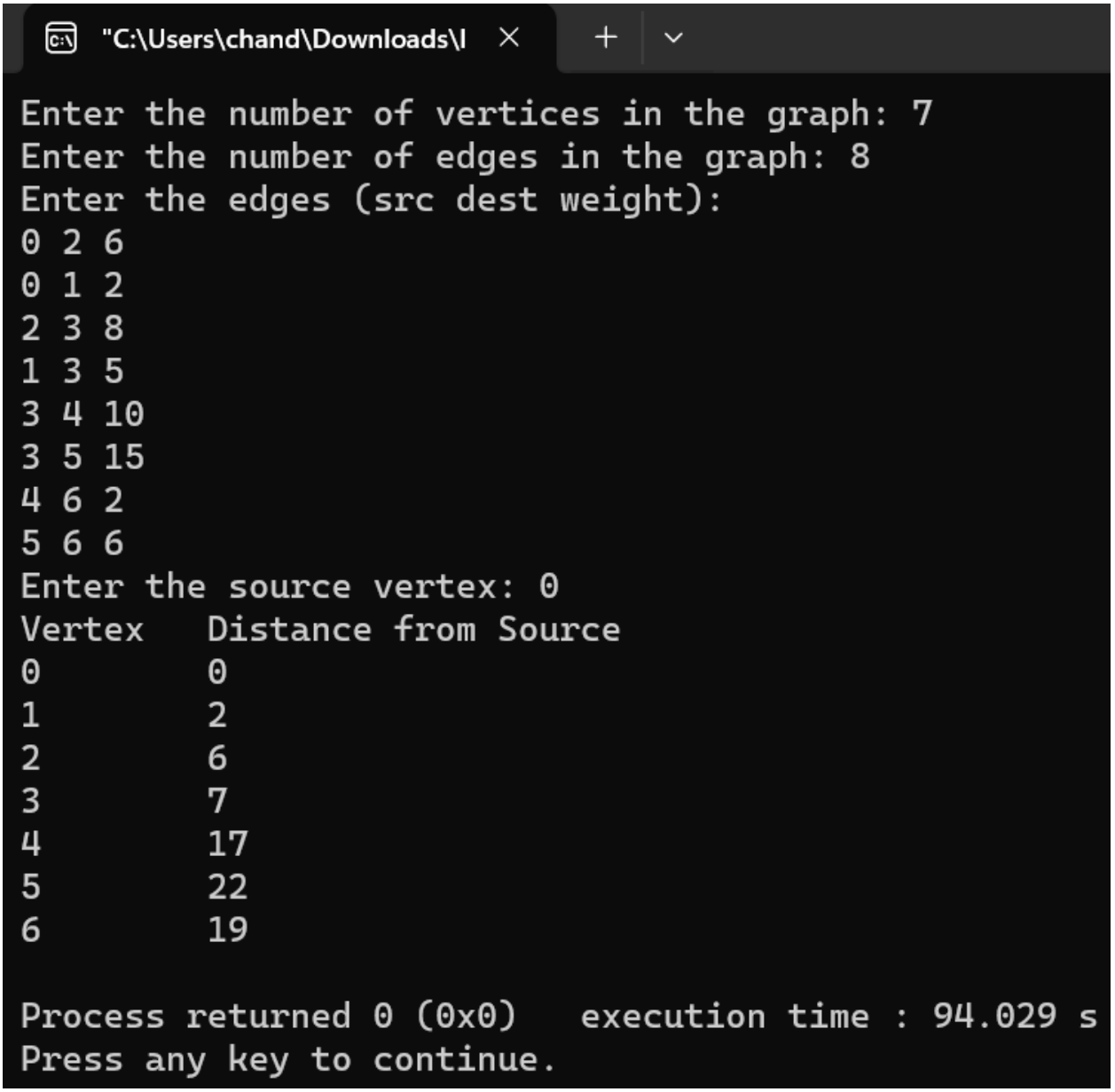
Program(s) of Dijkstra Algorithm:

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
struct AdjListNode {
   int dest;
   int weight;
   struct AdjListNode* next;
};
struct AdjList {
   struct AdjListNode* head;
};
struct Graph {
   int V; // Number of vertices
   struct AdjList* array;
};
struct AdjListNode* newAdjListNode(int dest, int weight) {
   struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
   newNode->dest = dest;
   newNode->weight = weight;
   newNode->next = NULL;
   return newNode;
}
struct Graph* createGraph(int V) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->V = V;
   graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));
   for (int i = 0; i < V; ++i)
```

```
        graph->array[i].head = NULL;
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest, int weight) {
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}
int minDistance(int dist[], bool sptSet[], int n) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < n; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}
void printSolution(int dist[], int n) {
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < n; i++)
        printf("%d \t %d\n", i, dist[i]);
}
void dijkstra(struct Graph* graph, int src) {
    int n = graph->V;
    int dist[n];
    bool sptSet[n];
    for (int i = 0; i < n; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
    dist[src] = 0;
    for (int count = 0; count < n - 1; count++) {
        int u = minDistance(dist, sptSet, n);
        sptSet[u] = true;
        struct AdjListNode* pCrawl = graph->array[u].head;
        while (pCrawl != NULL) {
            int v = pCrawl->dest;
            if (!sptSet[v] && dist[u] != INT_MAX && dist[u] + pCrawl->weight < dist[v])
                dist[v] = dist[u] + pCrawl->weight;
            pCrawl = pCrawl->next;
        }
    }
    printSolution(dist, n);
}
int main() {
    int n; // Number of vertices in the graph
    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &n);
    struct Graph* graph = createGraph(n);
    int m; // Number of edges in the graph
    printf("Enter the number of edges in the graph: ");
```

```
    scanf("%d", &m);
    printf("Enter the edges (src dest weight):\n");
    for (int i = 0; i < m; i++) {
        int src, dest, weight;
        scanf("%d %d %d", &src, &dest, &weight);
        addEdge(graph, src, dest, weight);
    }
    int src; // Source vertex
    printf("Enter the source vertex: ");
    scanf("%d", &src);
    dijkstra(graph, src);
    return 0;
}
```

Output(o) of Dijkstra Algorithm:



Conclusion: (Based on the observations):
Implementing and analyzing Dijkstra's algorithm provides valuable insights into its efficiency, scalability, and suitability for various graph-related applications, contributing to a deeper understanding of algorithmic concepts and their practical implications.

Outcome: Implement Greedy and Dynamic 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.