**ACADEMIC YEAR: 2024-25**

**Course:** Analysis of Algorithm Lab

**Course code:** CSL401

**Year/Sem:** SE/IV

| |
|---|
| **Experiment No.:** 04 |
| **Aim:** To implement single source shortest path – Dijkstra's algorithm using greedy method approach. |
| **Name:** SOHAM HEMENDRA RAUT |
| **Roll Number:** 24 |
| **Date of Performance:** 06/02/2025 |
| **Date of Submission:** 13/02/2025 |

**Evaluation**

| Performance Indicator | Max. Marks | Marks Obtained |
|---|---|---|
| Performance | 5 | |
| Understanding | 5 | |
| Journal work and timely submission. | 10 | |
| **Total** | **20** | |

| Performance Indicator | Exceed Expectations (EE) | Meet Expectations (ME) | Below Expectations (BE) |
|---|---|---|---|
| Performance | 5 | 3 | 2 |
| Understanding | 5 | 3 | 2 |
| Journal work and timely submission. | 10 | 8 | 4 |

**Checked by**

**Name of Faculty** : Mrs. Komal Champanerkar

**Signature** : **Date** :

- **Aim: To implement single source shortest path – Dijkstra's algorithm using greedy method approach.**
- **Theory:**

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex v to all other vertices in the graph.

Dijkstras Algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph

theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Input: Weighted graph G={E,V} and source vertex v∈V, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex v∈V to all other vertices

- **Approach:**

• The algorithm computes for each vertex u the distance to u from the start vertex v, that is, the weight of a shortest path between v and u.

• the algorithm keeps track of the set of vertices for which the distance has been computed, called the cloud C

• Every vertex has a label D associated with it. For any vertex u, D[u] stores an approximation of the distance between v and u. The algorithm will update a D[u] value when it finds a shorter path from v to u.

• When a vertex u is added to the cloud, its label D[u] is equal to the actual (final) distance between the starting vertex v and vertex u.

- **Algorithm:**

**Step 1:** Initialization

- Set the distance to the source node as `0` and all other nodes as infinity (∞).

- Push the source node into a priority queue with a distance of `0`.

**Step 2:** Process Nodes

- While the priority queue is not empty:
  - Extract the node `u` with the smallest distance.
  - For each neighbor `v` of `u`:

    If `v` is not visited and the distance via `u` is shorter than the current distance to `v`, update `dist[v]` and push `v` into the queue.

**Step 3:** Termination

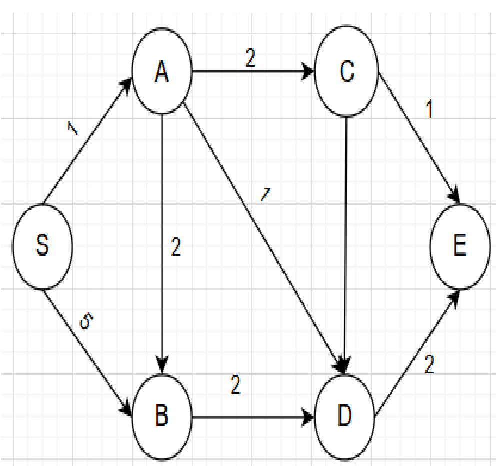
- Repeat until all nodes are processed.
- The distance array will contain the shortest path from the source to all nodes.

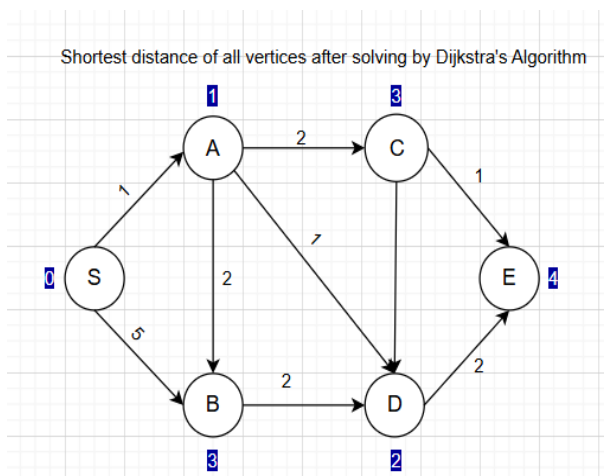
❖ **Program:**

```
import heapq

def dijkstra(graph, start):

    pq = [(0, start)]

    distances = {node: float('inf') for node in graph}    # Dictionary to store the shortest distance to each node

    distances[start] = 0

    visited = set()



    while pq:

        current_distance, current_node = heapq.heappop(pq)    # select the node with the smallest distance
```

```
        if current_node in visited:
            continue
        visited.add(current_node)    # Mark the node as visited

        for neighbor, weight in graph[current_node].items():   # Update the distances to the neighbors
            if neighbor not in visited:
                new_distance = current_distance + weight
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance
                    heapq.heappush(pq, (new_distance, neighbor))
    return distances
def get_input():
    graph = {}
    num_nodes = int(input("Enter the number of nodes: "))    # number of nodes
    for _ in range(num_nodes):   # node and its edges
        node = input("Enter node name: ").strip()
        graph[node] = {}

        num_edges = int(input(f"Enter the number of edges for node {node}: "))
```

```
        for _ in range(num_edges):     # Input for each edge (neighbor, weight)
            neighbor, weight = input(f"Enter neighbor and weight for edge (e.g., 'B 4'): ").split()
            graph[node][neighbor] = int(weight)
    start_node = input("Enter the source node: ").strip()
    return graph, start_node
if __name__ == "__main__":
    graph, start_node = get_input()
    shortest_paths = dijkstra(graph, start_node)
    print(f"\nShortest distances from {start_node}:")  # Print the shortest distance
    for node, distance in shortest_paths.items():
        print(f"{node}: {distance}")
```



❖ **Output:**

```
PS C:\Users\SOHAM> & C:/Users/SOHAM/anaconda3/python.exe c:/Users/SOHAM/.conda/dijkstra.py
Enter the number of nodes: 6
Enter node name: s
Enter the number of edges for node s: 2
Enter neighbor and weight for edge (e.g., 'B 4'): a 1
Enter neighbor and weight for edge (e.g., 'B 4'): b 5
Enter node name: a
Enter the number of edges for node a: 3
Enter neighbor and weight for edge (e.g., 'B 4'): b 2
Enter neighbor and weight for edge (e.g., 'B 4'): c 2
Enter neighbor and weight for edge (e.g., 'B 4'): d 1
Enter node name: b
Enter the number of edges for node b: 1
Enter neighbor and weight for edge (e.g., 'B 4'): d 2
Enter node name: c
Enter the number of edges for node c: 2
Enter neighbor and weight for edge (e.g., 'B 4'): d 3
Enter neighbor and weight for edge (e.g., 'B 4'): e 1
Enter node name: d
Enter the number of edges for node d: 1
Enter neighbor and weight for edge (e.g., 'B 4'): e 2
Enter node name: e
Enter the number of edges for node e: 0
Enter the source node: s

Shortest distances from s:
s: 0
a: 1
b: 3
c: 3
d: 2
e: 4
```

❖ **Conclusion:**

Dijkstra's algorithm has a time complexity of O(V log V) in the best case, which occurs in sparse graphs. In the worst and average cases, particularly in dense graphs or when many edges must be processed, the complexity is O((V + E) log V). The algorithm utilizes a priority queue (min-heap) to efficiently select the node with the smallest distance and processes each node and edge only once. The space complexity is O(V + E) due to the storage of distances and the graph's adjacency list. Overall, the efficiency of Dijkstra's algorithm is influenced by the density and structure of the graph.