Lab 13

**Application of Graphs**

#### Depth-First Search

* We begin by visiting the start vertex v. Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated.
* When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited w adjacent to it and initiates a depth- first search from w.

##### Depth-First-Search

{

Boolean visited[n];

// initially, no vertex has been visited for (i = 0; i < n; i++)

visited[i] = FALSE;//Initially all vertices are unvisted

// start search at vertex 0 DFS (0);

}

##### DFS (v)

// Visit all previously unvisited vertices that are reachable from vertex v

{

visited[v] = TRUE;

for (each vertex w adjacent to v) if ( !visited[w])

DFS(w);

}

**Graph Representation:**

**0**

**/ \**

**1 2**

**/ \ \**

**3 4 5**

**Output of the code:** **DFS Traversal: 0 1 3 4 2 5**

#### Breadth-First Search

* In a breadth-first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are visited.
* Unvisited vertices adjacent to these newly visited vertices are then visited, and so on.

##### BFS(v)

// A BFS of the graph is carried out starting at vertex v.

// visited[i] is set to TRUE when v is visited. The algorithm uses a queue.

{

Boolean visited[n]; Queue q;

// initially, no vertex has been visited for (i = 0; i < n; i++)

visited[i] = FALSE; visited[v] = TRUE;

q.insert(v); // add vertex v to the queue while (!q.IsEmpty())

{

v = q.delete();

for (all vertex w adjacent to v)

{ if ( !visited[w])

{ q.insert(w);

visited[w] = TRUE;

}

}

}

}

**Output:**

**BFS Traversal: 0 1 2 3 4 5**

#### Dijkstra’s Shortest Path Algorithm

1. Initialize:

- Create a set `unvisited` containing all nodes in the graph.

- Create a map `distance` to store the shortest distance from the source to each node, initialized to infinity (∞) for all nodes except the source, which is set to 0.

- Create a map `previous` to store the previous node in the shortest path for each node, initialized to `null`.

2. Set the source node as `current\_node`.

3. While `unvisited` is not empty:

a. For each neighbor `neighbor` of `current\_node`:

i. Calculate the tentative distance:

`tentative\_distance = distance[current\_node] + weight(current\_node, neighbor)`

ii. If `tentative\_distance < distance[neighbor]`:

- Update `distance[neighbor]` to `tentative\_distance`.

- Update `previous[neighbor]` to `current\_node`.

b. Remove `current\_node` from `unvisited`.

c. If all remaining nodes in `unvisited` have a distance of infinity, break the loop (the graph is disconnected).

d. Select the node in `unvisited` with the smallest `distance` value as the new `current\_node`.

4. After exiting the loop:

- The `distance` map contains the shortest distances from the source to each node.

- The `previous` map can be used to reconstruct the shortest path from the source to any other node.

5. To find the shortest path to a specific destination node:

- Start from the destination node and trace back using the `previous` map to reconstruct the path.

**Lab Tasks:**

**Task 1:** Implement Breadth-First and Depth-First search algorithms for a graph.

**Task 2: Implement** the **Dijkstra’s Shortest Path Algorithm** to generate shortest paths for a directed graph.