Aim: Implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search

Theory:

Graphs: A graph consists of vertices (or nodes) connected by edges. Graphs can be either directed or undirected, and they can also have weighted or unweighted edges.

BFS (Breadth-First Search):

BFS is a traversal technique that explores vertices in layers. It starts from a source vertex, visits all its neighbours, then moves to the neighbour's neighbours, continuing layer by layer.

Applications: BFS is used to find the shortest path in unweighted graphs, discover connected components, and more.

DFS (Depth-First Search):

DFS is a traversal technique that explores as far down a path as possible before backtracking. Starting from a source vertex, it follows a path until it can't go further, then backtracks to explore other paths.

Applications: DFS is used for pathfinding, cycle detection, topological sorting, and connected component discovery.

Procedure:

1. BFS (Breadth-First Search)
Function: BFS(source)
Description: Traverses the graph layer by layer starting from the source vertex.
Algorithm:
Initialize a queue and a visited list:
Enqueue the source vertex and mark it as visited.
While the queue is not empty:
Dequeue a vertex from the front of the queue.
Process (or print) the dequeued vertex.
For each unvisited neighbor of the dequeued vertex:
Mark the neighbor as visited and enqueue it.

Code:

a) Depth First Search

```
#include <stdio.h>
 #include <stdlib.h>
 #include <stdbool.h>
```

```
// Structure for a node in the adjacency list
struct Node {
int data;
struct Node* next;
};
// Structure for the adjacency list
struct List {
struct Node* head;
};
// Structure for the graph
struct Graph {
int vertices;
struct List* array;
};
// Function to create a new node
struct Node* createNode(int data) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->next = NULL;
return newNode;
}
// Function to create a graph with a given number of vertices
struct Graph* createGraph(int vertices) {
struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
graph->vertices = vertices;
graph->array = (struct List*)malloc(vertices * sizeof(struct
List));
for (int i = 0; i < vertices; i++) {
graph->array[i].head = NULL;
}
return graph;
}
// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
struct Node* newNode = createNode(dest);
newNode->next = graph->array[src].head;
graph->array[src].head = newNode;
// Uncomment the following code to make the graph undirected
/*
newNode = createNode(src);
newNode->next = graph->array[dest].head;
graph->array[dest].head = newNode;
```

```
*/
}
// Function to perform Depth First Search (DFS) from a given vertex
void DFS(struct Graph* graph, int vertex, bool visited[]) {
visited[vertex] = true;
printf("%d ", vertex);
struct Node* currentNode = graph->array[vertex].head;
while (currentNode) {
int adjacentVertex = currentNode->data;
if (!visited[adjacentVertex]) {
DFS(graph, adjacentVertex, visited);
}
currentNode = currentNode->next;
}
}
// Function to perform DFS traversal from a given vertex in a
specified order
void DFSTraversal(struct Graph* graph, int* order, int orderSize) {
bool* visited = (bool*)malloc(graph->vertices * sizeof(bool));
for (int i = 0; i < graph->vertices; i++) {
visited[i] = false;
}
for (int i = 0; i < orderSize; i++) {
if (!visited[order[i]]) {
DFS(graph, order[i], visited);
}
}
free(visited);
}
int main() {
int vertices = 4;
struct Graph* graph = createGraph(vertices);
addEdge(graph, 2, 0);
addEdge(graph, 0, 2);
addEdge(graph, 1, 2);
addEdge(graph, 0, 1);
addEdge(graph, 3, 3);
addEdge(graph, 1, 3);
int order[] = {2, 0, 1, 3};
int orderSize = sizeof(order) / sizeof(order[0]);
printf("Following is Depth First Traversal (starting from vertex
2):\n");
```

```
 DFSTraversal(graph, order, orderSize);
 return 0;
 }
```

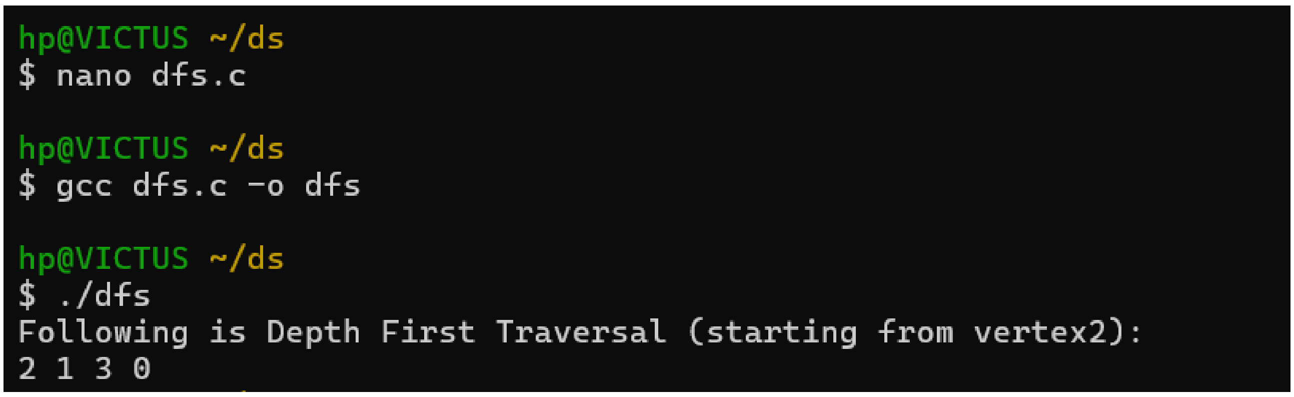
b) Breadth First Search

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 100
// BFS function from a given source s
void bfs(int adj[MAX][MAX], int V, int s) {
   // Queue for BFS
   int q[MAX], front = 0, rear = 0;
   // Array to mark all vertices as not visited initially
   bool visited[MAX] = { false };

   // Mark the source node as visited and enqueue it
   visited[s] = true;
   q[rear++] = s;
   printf("BFS Traversal starting from node %d: ", s);
   // Iterate over the queue
   while (front < rear) {
      // Dequeue a vertex and print it
      int curr = q[front++];
      printf("%d ", curr);

      // Get all adjacent vertices of the dequeued vertex
      for (int i = 0; i < V; i++) {
         if (adj[curr][i] == 1 && !visited[i]) {
            visited[i] = true;
            q[rear++] = i;
         }
      }
   }
   printf("\n");
}
// Function to add an edge to the graph
void addEdge(int adj[MAX][MAX], int u, int v) {
   adj[u][v] = 1;
   adj[v][u] = 1; // For undirected graph
}
```

```
int main() {
    // Number of vertices in the graph
    int V = 5;
    // Adjacency matrix representation of the graph
    int adj[MAX][MAX] = {0};
    // Add edges to the graph
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 4);

    // Perform BFS traversal from node 0
    bfs(adj, V, 0);
    return 0;
}
```

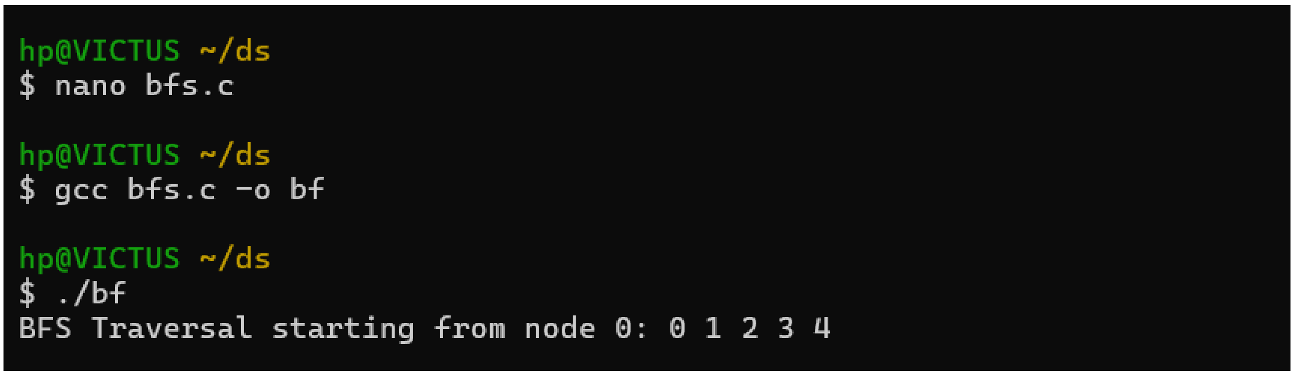
Output:

a) Depth First Search



```
hp@VICTUS ~/ds
$ nano dfs.c

hp@VICTUS ~/ds
$ gcc dfs.c -o dfs

hp@VICTUS ~/ds
$ ./dfs
Following is Depth First Traversal (starting from vertex2):
2 1 3 0
```

b) Breadth First Search



```
hp@VICTUS ~/ds
$ nano bfs.c

hp@VICTUS ~/ds
$ gcc bfs.c -o bf

hp@VICTUS ~/ds
$ ./bf
BFS Traversal starting from node 0: 0 1 2 3 4
```

Conclusion:

Implement graph traversal techniques, specifically Depth First Search and Breadth First Search, using a linked list. This experiment delves into exploring graphs, which are data structures representing relationships between nodes, using linked lists for node connections. Depth First Search traverses the graph starting from a root node and going as deep as possible along each branch before backtracking, while Breadth First Search visits all nodes at a particular level before going to the next level