Data Structures : Algorithms and Applications(Lab)

EXPERIMENT No: 11

AIM: Implementation of BFS and DFS on a directed graph using an adjacency matrix.

THEORY:

Breadth-First Search (BFS) is a graph traversal algorithm that explores vertices in layers, starting from a given source vertex and visiting all of its neighboring vertices before moving on to the next level of vertices. BFS employs a queue data structure to keep track of vertices that need to be explored. This ensures that the algorithm processes vertices in the order they are discovered, making it particularly useful for finding the shortest path in unweighted graphs.

Key Concepts:

- Graph Representation: In the provided implementation, the graph is represented using an adjacency matrix, where adjMatrix[i][j] is set to 1 if there is an edge between vertices i and j, and 0 otherwise.
- Vertex Structure: Each vertex is represented by a structure that contains its label and a visited flag, indicating whether it has been processed.
- Queue for Traversal: A circular queue is utilized for managing the vertices to be explored, allowing efficient addition and removal of elements.
- Traversal Process:
    1. Start from a specified source vertex and mark it as visited.
    2. Enqueue the source vertex.
    3. While the queue is not empty, dequeue a vertex and explore its adjacent unvisited vertices, marking them as visited and enqueueing them for later exploration.
    4. This process continues until all reachable vertices from the source have been visited.

Depth First Search (DFS) is an algorithm used for traversing or searching tree or graph data structures. The algorithm starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

In this program, we implement DFS using an adjacency matrix to represent the graph and a stack to manage the traversal process. The adjacency matrix is a 2D array where each element adjMatrix[i][j] is 1 if there is an edge between vertex i and vertex j, otherwise it's 0.

Key Concepts:

1. Vertices and Edges: Vertices are stored as a list of structures, where each structure holds the vertex label and a boolean flag to indicate if it has been visited. The edges between the vertices are represented in the adjacency matrix.

2. Stack: DFS uses a stack to keep track of the vertices as it traverses the graph. The last pushed vertex is explored first (LIFO).
3. DFS Process:
   - Start by marking the first vertex as visited and push it onto the stack.
   - Then, explore its unvisited adjacent vertices, marking them as visited and pushing them onto the stack.
   - Continue this process until all reachable vertices have been visited.
   - If there are no more unvisited adjacent vertices for a vertex, pop it from the stack and backtrack.
   - Once the stack is empty, the DFS traversal is complete.

CODE:
Implementation of BFS using adjacency matrix.

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

#define MAX 5

void addVertex(char);
void addEdge(int, int);
void displayVertex(int);
void depthFirstSearch();
int getAdjUnvisitedVertex(int);

struct Vertex {
  char label;
  bool visited;
};

//stack variables
int stack[MAX];
int top = -1;

//graph variables
//array of vertices
struct Vertex * lstVertices[MAX];
//adjacency matrix
int adjMatrix[MAX][MAX];
//vertex count
int vertexCount = 0;

//stack functions
void push(int item) {
  stack[++top] = item;
}
```

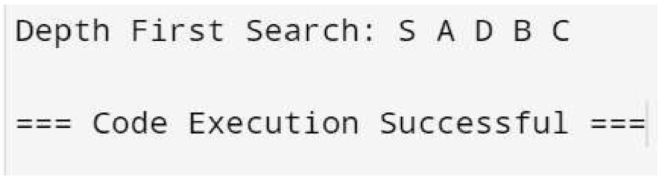
```
int pop() {
  return stack[top--];
}

int peek() {
  return stack[top];
}

bool isStackEmpty() {
  return top == -1;
}

//graph functions
//add vertex to the vertex list
void addVertex(char label) {
  struct Vertex * vertex = (struct Vertex * ) malloc(sizeof(struct Vertex));
  vertex -> label = label;
  vertex -> visited = false;
  lstVertices[vertexCount++] = vertex;
}

//add edge to edge array
void addEdge(int start, int end) {
  adjMatrix[start][end] = 1;
  adjMatrix[end][start] = 1;
}

//display the vertex
void displayVertex(int vertexIndex) {
  printf("%c ", lstVertices[vertexIndex] -> label);
}

//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
  int i;
  for (i = 0; i < vertexCount; i++) {
    if (adjMatrix[vertexIndex][i] == 1 && lstVertices[i] -> visited == false) {
      return i;
    }
  }
  return -1;
}

void depthFirstSearch() {
  int i;

  //mark first node as visited
  lstVertices[0] -> visited = true;

  //display the vertex
  displayVertex(0);

  //push vertex index in stack
  push(0);

  while (!isStackEmpty()) {
    //get the unvisited vertex of vertex which is at top of the stack
    int unvisitedVertex = getAdjUnvisitedVertex(peek());
    //no adjacent vertex found
    if (unvisitedVertex == -1) {
      pop();
    } else {
      lstVertices[unvisitedVertex] -> visited = true;
      displayVertex(unvisitedVertex);
      push(unvisitedVertex);
    }
  }

  //stack is empty, search is complete, reset the visited flag
  for (i = 0; i < vertexCount; i++) {
    lstVertices[i] -> visited = false;
  }
}

int main() {
  int i, j;
  for (i = 0; i < MAX; i++) // set adjacency {
    for (j = 0; j < MAX; j++) // matrix to 0
      adjMatrix[i][j] = 0;

  addVertex('S'); // 0
  addVertex('A'); // 1
  addVertex('B'); // 2
  addVertex('C'); // 3
  addVertex('D'); // 4

  addEdge(0, 1); // S - A
  addEdge(0, 2); // S - B
  addEdge(0, 3); // S - C
  addEdge(1, 4); // A - D
  addEdge(2, 4); // B - D
  addEdge(3, 4); // C - D

  printf("Depth First Search: ");
  depthFirstSearch();

  return 0;
}
```

Implementation of DFS using adjacency matrix.

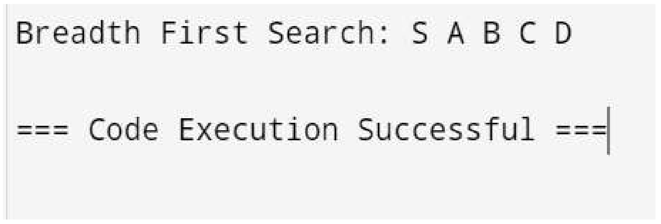
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 5

struct Vertex {
  char label;
  bool visited;
};

// Graph variables
struct Vertex *lstVertices[MAX];
int adjMatrix[MAX][MAX];
int vertexCount = 0;

// Queue variables
int queue[MAX], front = 0, rear = -1;

// Function declarations
void addVertex(char label);
void addEdge(int start, int end);
void displayVertex(int vertexIndex);
void breadthFirstSearch();
bool isQueueEmpty();
void enqueue(int item);
int dequeue();

void addVertex(char label) {
  struct Vertex *vertex = (struct Vertex *)malloc(sizeof(struct Vertex));
  vertex->label = label;
  vertex->visited = false;
  lstVertices[vertexCount++] = vertex;
}

void addEdge(int start, int end) {
  adjMatrix[start][end] = 1;
  adjMatrix[end][start] = 1; // Undirected graph
}

void displayVertex(int vertexIndex) {
  printf("%c ", lstVertices[vertexIndex]->label);
}

bool isQueueEmpty() {
  return front == (rear + 1) % MAX;
}
```

```
void enqueue(int item) {
   rear = (rear + 1) % MAX;
   queue[rear] = item;
}

int dequeue() {
   int data = queue[front];
   front = (front + 1) % MAX;
   return data;
}

void breadthFirstSearch() {
   lstVertices[0]->visited = true;
   displayVertex(0);
   enqueue(0);

   while (!isQueueEmpty()) {
      int currentVertex = dequeue();
      for (int i = 0; i < vertexCount; i++) {
         if (adjMatrix[currentVertex][i] == 1 && !lstVertices[i]->visited) {
            lstVertices[i]->visited = true;
            displayVertex(i);
            enqueue(i);
         }
      }
   }

   // Reset visited flags
   for (int i = 0; i < vertexCount; i++) {
      lstVertices[i]->visited = false;
   }
}

int main() {
   for (int i = 0; i < MAX; i++)
      for (int j = 0; j < MAX; j++)
         adjMatrix[i][j] = 0;

   addVertex('S'); // 0
   addVertex('A'); // 1
   addVertex('B'); // 2
   addVertex('C'); // 3
   addVertex('D'); // 4

   addEdge(0, 1); // S - A
   addEdge(0, 2); // S - B
   addEdge(0, 3); // S - C
   addEdge(1, 4); // A - D
   addEdge(2, 4); // B - D
   addEdge(3, 4); // C - D
   printf("Breadth First Search: ");
   breadthFirstSearch();

   return 0;
}
```

OUTPUT:



```
Depth First Search: S A D B C

=== Code Execution Successful ===
```



```
Breadth First Search: S A B C D

=== Code Execution Successful ===
```

CONCLUSION:

BFS and DFS are key graph traversal algorithms. BFS uses a queue to explore nodes level by level, making it suitable for finding the shortest path in unweighted graphs, while DFS uses a stack for deep exploration, ideal for exhaustive searches. The choice between them depends on specific problem requirements.