ADS LAB ASSINGMENT 7

Name-Sandesh Santosh Kadam
Class- SY IT-C Batch 1
PRN-12210695
Roll no-26

Q. **Write C/C++ program for graph traversals using BFS and DFS**

Theory-

Graph Traversals: BFS (Breadth-First Search) and DFS (Depth-First Search)

Graph traversals are fundamental algorithms used to visit all the vertices and edges of a graph. They are crucial for exploring and understanding the structure of a graph.

Breadth-First Search (BFS):

1. **Overview:**
    - **BFS** explores a graph level by level, visiting all the neighbors of a vertex before moving on to the next level.
    - It uses a queue to maintain the order of vertices to be processed.
2. **Process:**
    - Start from a source vertex and enqueue it.
    - While the queue is not empty:
        - Dequeue a vertex, visit it, and enqueue all its unvisited neighbors.
    - Repeat until all vertices are visited.
3. **Applications:**
    - Shortest path finding in an unweighted graph.
    - Connected components in an undirected graph.

- Web crawling and social network analysis.

4. **Time Complexity:**
    - O(V + E), where V is the number of vertices and E is the number of edges.

Depth-First Search (DFS):

1. **Overview:**
    - **DFS** explores a graph by going as deep as possible along each branch before backtracking.
    - It uses a stack (recursion in the case of recursive DFS) to maintain the order of vertices.
2. **Process:**
    - Start from a source vertex and visit it.
    - For each unvisited neighbor, recursively apply DFS.
    - Backtrack if needed.
3. **Applications:**
    - Topological sorting of a directed acyclic graph (DAG).
    - Detecting cycles in a graph.
    - Solving puzzles and games.
4. **Time Complexity:**
    - O(V + E), where V is the number of vertices and E is the number of edges.

Comparison:

1. **Memory Usage:**
    - BFS uses more memory due to the need to store all vertices at the current level.
    - DFS is memory-efficient as it only needs to store the current path.
2. **Path Length:**

- BFS guarantees the shortest path in an unweighted graph.
- DFS does not guarantee the shortest path.

3. **Implementations:**
    - BFS is often implemented using a queue.
    - DFS can be implemented using a stack or recursion.
4. **Completeness:**
    - Both BFS and DFS are complete for connected graphs.
5. **Use Cases:**
    - Choose BFS when finding the shortest path is crucial.
    - Choose DFS when exploring all possibilities or detecting cycles is important.
6. **Order of Visitation:**
    - BFS visits vertices in layers, level by level.
    - DFS can visit vertices in a different order based on the traversal strategy.

Graph traversals play a vital role in various algorithms and applications, aiding in the analysis and understanding of graph structures. The choice between BFS and DFS depends on the specific requirements of the problem at hand.

Programming Lang. Used- C

Compiler Used- VS Code

CODE BFS-

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

struct queue
{
    int size;
    int f;
```

```
    int r;
    int* arr;
};


int isEmpty(struct queue *q){
    if(q->r==q->f){
        return 1;
    }
    return 0;
}

int isFull(struct queue *q){
    if(q->r==q->size-1){
        return 1;
    }
    return 0;
}

void enqueue(struct queue *q, int val){
    if(isFull(q)){
        printf("This Queue is full\n");
    }
    else{
        q->r++;
        q->arr[q->r] = val;
        // printf("Enqued element: %d\n", val);
    }
}

int dequeue(struct queue *q){
    int a = -1;
    if(isEmpty(q)){
        printf("This Queue is empty\n");
    }
    else{
        q->f++;
        a = q->arr[q->f];
    }
    return a;
}
int main() {
    struct queue q;
    q.size = 400;
    q.f = q.r = 0;
    q.arr = (int*) malloc(q.size*sizeof(int));

    // BFS Implementation
    int node;
```

```
    int i = 0;
    int visited[7] = {0,0,0,0,0,0,0};
    int a [7][7] = {
        {0,1,1,0,0,0,0},
        {1,0,0,1,0,0,0},
        {1,0,0,1,1,0,0},
        {0,1,1,0,0,0,0},
        {0,0,1,0,0,1,1},
        {0,0,0,0,1,0,1},
        {0,0,0,0,1,1,0}
    };
    int j;
    printf("%d ", i);
    enqueue(&q, i);
    while (!isEmpty(&q))
    {
        int node = dequeue(&q); //remove
        visited[i] = 1; //mark visited
        for (j = 0; j < 7; j++)
        {
            if(a[node][j] ==1 && visited[j] == 0){
                printf("%d ", j); //add to the bfs
                visited[j] = 1;
                enqueue(&q, j); //add neighbors
            }
        }
    }
    return 0;
}
```

OUTPUT-

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\SANDESH\Documents\psap c>  & 'c:\Users\SANDESH\.vscode\extensions\ms-vscode.cpptools-1.18.5-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-jf01bznm.3kj' '--stdout=Microsoft-MIEngine-Out-pu0iznjm.k4t' '--stderr=Microsoft-MIEngine-Error-fjzcvai2.nqj' '--pid=Microsoft-MIEngine-Pid-cj5xgmor.odg' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'
0 1 2 3 4 5 6
PS C:\Users\SANDESH\Documents\psap c>
```

CODE DFS-

```c
// DFS algorithm in C

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

struct node {
  int vertex;
  struct node* next;
};

struct node* createNode(int v);

struct Graph {
  int numVertices;
  int* visited;

  // We need int** to store a two dimensional array.
  // Similary, we need struct node** to store an array of Linked lists
  struct node** adjLists;
};

// DFS algo
void DFS(struct Graph* graph, int vertex) {
  struct node* adjList = graph->adjLists[vertex];
  struct node* temp = adjList;

  graph->visited[vertex] = 1;
  printf("Visited %d \n", vertex);

  while (temp != NULL) {
    int connectedVertex = temp->vertex;
```

```
    if (graph->visited[connectedVertex] == 0) {
      DFS(graph, connectedVertex);
    }
    temp = temp->next;
  }
}

// Create a node
struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

// Create graph
struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));

  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
  // Add edge from src to dest
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;

  // Add edge from dest to src
  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

// Print the graph
void printGraph(struct Graph* graph) {
  int v;
```

```
  for (v = 0; v < graph->numVertices; v++) {
    struct node* temp = graph->adjLists[v];
    printf("\n Adjacency list of vertex %d\n ", v);
    while (temp) {
      printf("%d -> ", temp->vertex);
      temp = temp->next;
    }
    printf("\n");
  }
}

int main() {
  struct Graph* graph = createGraph(4);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 2, 3);

  printGraph(graph);

  DFS(graph, 2);

  return 0;
}
```

OUTPUT-

```
 Adjacency list of vertex 0
 2 -> 1 ->

 Adjacency list of vertex 1
 2 -> 0 ->

 Adjacency list of vertex 2
 3 -> 1 -> 0 ->

 Adjacency list of vertex 3
 2 ->
Visited 2
Visited 3
Visited 1
Visited 0
```