EXPERIMENT 9

Aim:

1. For a given graph $G = (V, E)$, study and implement the Breadth First Search (or traversal) i.e., BFS. Also, perform complexity analysis of this algorithm in-terms of time and space.
2. For a given graph G=(V,E), study and implement the Depth First Search (or traversal) i.e., DFS. Also, perform complexity analysis of this algorithm in-terms of time and space.

Theory (Breadth First Search [BFS]):

BFS explores a graph layer by layer, starting from a source node, and visiting all neighboring nodes before moving on to nodes at the next distance level. It uses a **queue** data structure to keep track of nodes to visit next. BFS marks each node as visited when enqueued to prevent re-processing. BFS is typically used to find the shortest path in an unweighted graph.

1. Enqueue the starting node and mark it as visited.
2. While the queue is not empty:
    a. Dequeue a node, process it (e.g., print or store it).
    b. Enqueue all unvisited adjacent nodes of the dequeued node and mark them as visited.
3. Repeat until the queue is empty, ensuring all nodes at each level are visited before advancing.

Time Complexity:

1. **Adjacency List**: $O(V + E)$, where V is the number of vertices and E is the number of edges. Each node and each edge is processed once.
2. **Adjacency Matrix**: $O(V^2)$, as we check all pairs of nodes for edges.

Space Complexity:

1. **Adjacency List**: $O(V + E)$, storing nodes and edges in the list.
2. **Queue**: $O(V)$, as the queue holds all nodes in the worst case.

Program:

```
// status array

// status
// 1- ready
// 2- waiting
// 3 - processed

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

#define MAX 10

int queue[MAX];
int front = -1;
int rear = -1;

// int adj[MAX][MAX];
// int status[MAX];
int status[9];

bool isEmpty();
bool isFull();
```

```
void enqueue(int ele);
int dequeue();
void printQueue();

void bfs(int start, int n, int adj[9][9]);

int main(void)
{
    int adj[9][9] = {
        {0,1,1,1,0,0,0,0,0},
        {1,0,1,0,1,0,0,0,0},
        {1,1,0,1,1,1,1,0,0},
        {1,0,1,0,0,0,1,0,0},
        {0,1,1,0,0,0,0,0,1},
        {0,0,1,0,0,0,1,1,1},
        {0,0,1,1,0,1,0,1,0},
        {0,0,0,0,0,1,1,0,1},
        {0,0,0,0,1,1,0,1,0}
    };

    for (int i = 0; i < 9; i++)
    {
        status[i] = 1;
    }

    bfs(0, 9, adj);

    return 0;
}

void bfs(int start, int n, int adj[9][9])
{
    printf("BFS Traversal: ");

    enqueue(start);
    status[start] = 2;

    while (!isEmpty())
    {
        int v = dequeue();
        printf("%d ", v);
        status[v] = 3;

        for (int i = 0; i < n; i++)
        {
            if (adj[v][i] == 1 && status[i] == 1)
            {
                enqueue(i);
                status[i] = 2;
            }
        }
```

```
    }
    printf("\n");

}

bool isEmpty()
{
    return (front == -1);
}

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

void enqueue(int ele)
{
    if (isFull())
    {
        printf("Queue is FULL!\n");
        return;
    }
    if (isEmpty())
    {
        front = rear = 0;
    }
    else
    {
        rear = (rear + 1) % MAX;
    }
    queue[rear] = ele;
    // printf("Inserted: %d\n", ele);
}

int dequeue()
{
    if (isEmpty())
    {
        printf("Queue is EMPTY!\n");
        return -1;
    }

    int dequeued = queue[front];
    // printf("Deleted: %d\n", queue[front]);
    if (front == rear)
    {
        front = rear = -1;
    }
    else
    {
        front = (front + 1) % MAX;
```

```
    }

    return dequeued;
}

void printQueue()
{
    if (isEmpty())
    {
        printf("Queue is EMPTY!\n");
        return;
    }
    printf("Queue: ");
    int i = front;
    while (i != rear)
    {
        printf("%d ", queue[i]);
        i = (i + 1) % MAX;
    }
    printf("%d\n", queue[rear]);
}
```

Output:

```
PS B:\sem3\ds\23bcp153_dsa\lab9> gcc bfsmat.c -o bfsmat
PS B:\sem3\ds\23bcp153_dsa\lab9> ./bfsmat
BFS Traversal: 0 1 2 3 4 5 6 8 7
PS B:\sem3\ds\23bcp153_dsa\lab9>
```

Theory (Depth First Search [DFS]):

DFS explores as far as possible down one branch before backtracking, utilizing a **stack** (either explicitly or through recursive function calls). Starting from a source node, DFS goes to an unvisited adjacent node, then continues exploring deeper until reaching a dead end. After backtracking, DFS continues from the next unvisited node of the previously visited nodes. DFS is helpful in applications like detecting cycles, topological sorting, and finding connected components.

1. Push the starting node onto the stack and mark it as visited.
2. While the stack is not empty:
    a. Pop the top node, process it.
    b. Push all unvisited adjacent nodes of the popped node onto the stack and mark them as visited.
3. Repeat until the stack is empty, allowing deep exploration before moving to other branches.

Time Complexity:

1. **Adjacency List**: $O(V + E)$, as we process each vertex and each edge once.
2. **Adjacency Matrix**: $O(V^2)$, checking all pairs for connectivity.

Space Complexity:

1. **Adjacency List**: $O(V + E)$, to store vertices and edges.
2. **Stack**: $O(V)$ for the stack space, as each vertex is pushed onto the stack at most once.

Program:

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

// stack functions are included in stackarrlib.c/.h

#define MAX 10

typedef struct Stack
{
   int top;
   int capacity;
   int *array;
} Stack;

void dfs(int start, int n, int adj[9][9]);

// int adj[MAX][MAX];
// int status[MAX];

int status[9];

Stack *createStack(int n);
char pop(Stack *s);
void push(Stack *s, int value);
int isFull(Stack *s);
int isEmpty(Stack *s);
int peek(Stack *s);

int main(void)
{
   int adj[9][9] = {
      {0,1,1,1,0,0,0,0,0},
      {1,0,1,0,1,0,0,0,0},
      {1,1,0,1,1,1,1,0,0},
      {1,0,1,0,0,0,1,0,0},
      {0,1,1,0,0,0,0,0,1},
      {0,0,1,0,0,0,1,1,1},
      {0,0,1,1,0,1,0,1,0},
      {0,0,0,0,0,1,1,0,1},
      {0,0,0,0,1,1,0,1,0}
   };

   for (int i = 0; i < 9; i++)
   {
      status[i] = 1;
```

```
    }

    dfs(0, 9, adj);

    return 0;
}

void dfs(int start, int n, int adj[9][9])
{
    Stack *mystack = createStack(9);

    printf("DFS Traversal: ");
    push(mystack, start);
    status[start] = 2;

    while(!isEmpty(mystack))
    {
        int v = pop(mystack);
        printf("%d ", v);
        status[v] = 3;

        for (int i = n - 1; i >= 0; i--)
        {
            if (adj[v][i] == 1 && status[i] == 1)
            {
                push(mystack, i);
                status[i] = 2;
            }
        }
    }
    printf("\n");
}

Stack *createStack(int n)
{
    Stack *s = (Stack *)malloc(sizeof(Stack));
    s->capacity = n;
    s->top = -1;
    s->array = (int *)malloc(s->capacity * sizeof(int));
    return s;
}

char pop(Stack *s)
{
    if (isEmpty(s))
    {
        // Underflow
        printf("Stack is Empty\n");
        return -1;
    }
    char popped = s->array[s->top];
```

```
    s->top--;
    return popped;
}

void push(Stack *s, int value)
{
    // Overflow (as we do in algo in class)
    if (isFull(s))
    {
        printf("Stack is Full!\n");
        return;
    }

    s->top++;
    s->array[s->top] = value;
    return;
}

int isFull(Stack *s)
{
    return s->top == s->capacity - 1;
}

int isEmpty(Stack *s)
{
    return s->top == -1;
}

int peek(Stack *s)
{
    if (isEmpty(s))
    {
        return -1;
    }
    return s->array[s->top];
}
```

Output:

```
PS B:\sem3\ds\23bcp153_dsa\lab9> gcc dfsmat.c -o dfsmat
PS B:\sem3\ds\23bcp153_dsa\lab9> ./dfsmat
DFS Traversal: 0 1 4 8 5 6 7 2 3
PS B:\sem3\ds\23bcp153_dsa\lab9>
```