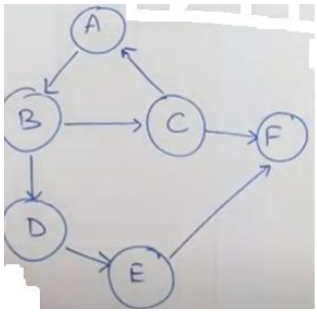
Assignment 9: Graph Traversal and its Applications

The cityADT consists of adjacency matrix that represents the connection between the cities. Adjacency matrix has an entry 1, if there is a connection between the cities. Implement the following methods. [CO2, K3]

- void create(cityADT *C) – will create the graph using adjacency matrix
- void disp(cityADT *C) – display the adjacency matrix
- void BFS(cityADT *C) – provides the output of visiting the cities by following breadth first
- void DFS(cityADT *C) – provides the output of visiting the cities by following depth first

1. Demonstrate the ADT with the following Graph



Enter the no. of vertices: 6
Enter the no. of edges: 7
AB, BC, BD, CA, CF, DE, EF

Adjacency Matrix

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	1	0	0
C	1	0	0	0	0	1
D	0	0	0	0	1	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

BFS Output: ABCDFE for Start vertex A

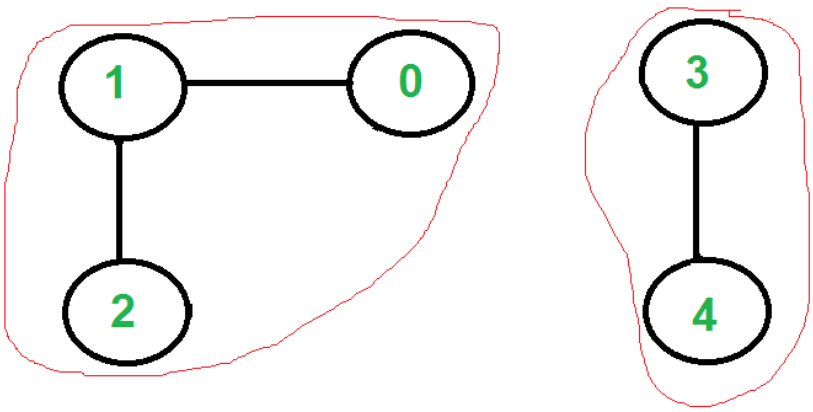
DFS Output: ABCFDE for Start vertex A

2. Write an application to utilize traversals to do the following:

a. Given the source and destination cities, find whether there is a path from source to destination
b. Find the connected components in a given graph

Test the application with the following
Input:
Source: D
Destination: F
Output:
Path exists
Input:
Source: F
Destination: B
Output:
Path not exists

Input:



There are two connected components in above undirected graph
0 1 2
3 4

1.CODE:

Queue.c

```
#include <stdio.h>
#include <stdlib.h>
struct queue{
```

```
        int a[100];
        int f,r,size;
};


void createQueue(struct queue *q, int s){
        q->size=s;
        q->f = q->r = -1;
}

void enqueue(struct queue *q, int x){
        if (q->f==-1 && q->r==-1){
                q->f++;
                q->r++;
                q->a[q->r] = x;
        }
        else if ((q->r+1)%(q->size) == q->f){
                printf("The queue is full. Cannot push\n");
        }
        else{
                q->r=(q->r+1)%(q->size);
                q->a[q->r] = x;
        }
}

int dequeue(struct queue *q){
        if (q->f == -1 && q->r == -1){
                return -1;
        }
        else if(q->f == q->r){
                int d;
                d=q->a[q->f];
                q->f=-1;
                q->r=-1;
                return d;
        }
        else{
                int d;
                d=q->a[q->f];
                q->f = (q->f +1)%(q->size);
                return d;
        }
}

int queueIsEmpty(struct queue *q){
        if (q->f==-1 && q->r==-1){
```

```
                return 1;
        }
        return 0;
}

void displayQueue(struct queue *q){
        printf("Queue elements : ");
        if (q->r < q->f){
                for (int i=q->f; i<q->size; i++){
                        printf("%d ",q->a[i]);
                }
                for (int i=0; i<q->r; i++){
                        printf("%d ",q->a[i]);
                }
        }
        else{
                for (int k=q->f; k<q->r+1; k++){
                        printf("%d ",q->a[k]);
                }
        }
        printf("\n");
}
```

Stack.c

```
#include <stdio.h>
#include <stdlib.h>
struct stack {
int top;
int data[100];
int size;
};
void init(struct stack *s, int limit) {
s->size = limit;
s->top = -1;
}
int isFull(struct stack *s) {
return s->top == s->size - 1;
}
int isEmpty(struct stack *s) {
return s->top == -1;
}
void push(struct stack *s, int d) {
if (isFull(s)) {
printf("Stack is full. %d.\n", d);
} else {
```

```
s->data[++s->top] = d;
//printf("\nPushed: %d", d);
}
}
void pop(struct stack *s) {
if (isEmpty(s)) {
printf("Stack is empty.\n");
} else {
//printf("Popped: %d\n", s->data[s->top]);
s->top--;
}
}
int peek(struct stack *s) {
if (isEmpty(s)) {
printf("Stack is empty.\n");
return -1;
} else {

return s->data[s->top];
}
}
```

Main.c

```
#include<stdio.h>
#include<stdlib.h>
#include"graph.c"
int main() {
    struct graph *g;
    int v, e;
    g = (struct graph *)malloc(sizeof(struct graph));
    struct edgePair edges[100];

    printf("Enter the number of vertices: ");
    scanf("%d", &v);
    printf("Enter the number of edges: ");
    scanf("%d", &e);

    for (int i = 0; i < e; i++) {
        printf("Enter the edge pair: ");
        scanf(" %c %c", &edges[i].from, &edges[i].to); // Adjusted to read characters
    }

    create(g, v, e, edges);
    display(g);

    char start_vertex;
    printf("Enter the starting vertex for BFS and DFS: ");
```

```
    scanf(" %c", &start_vertex);

    BFS(g, start_vertex);
    DFS(g, start_vertex);

    return 0;
}
```

Graph.c

```
#include <stdio.h>
#include <stdlib.h>
#include "queue.c"
#include "stack.c"

struct graph {
    int matrix[100][100];
    int size;
};

struct edgePair {
    char from;
    char to;
};

void create(struct graph *g, int v, int e, struct edgePair edges[]) {
    g->size = v;
    char from, to;
    for (int i = 0; i < e; i++) {
        from = edges[i].from;
        to = edges[i].to;
        g->matrix[from - 'a'][to - 'a'] = 1;
    }
}

void display(struct graph *g) {
    for (int i = 0; i < g->size; i++) {
        for (int j = 0; j < g->size; j++) {
            printf("%d\t", g->matrix[i][j]);
        }
        printf("\n");
    }
}

void BFS(struct graph *g, char start) {
    struct queue *q = (struct queue *)malloc(sizeof(struct queue));
    int visited[23] = {0};
```

```
    createQueue(q, g->size);
    visited[start - 'a'] = 1;
    enqueue(q, start - 'a');
    printf("BFS traversal:");

    while (!queueIsEmpty(q)) {
        int z = dequeue(q);
        printf("%c", z + 'a');
        for (int i = 0; i < g->size; i++) {
            if (g->matrix[z][i] && !(visited[i])) {
                visited[i] = 1;
                enqueue(q, i);
            }
        }
    }
    printf("\n");
}

void DFS(struct graph *g, char start) {
    struct stack *s = (struct stack *)malloc(sizeof(struct stack));
    init(s, g->size);
    int visited[100] = {0};
    push(s, start - 'a');
    printf("\nDFS traversal:");

    while (!isEmpty(s)) {
        int t = peek(s);
        if (!visited[t]) {
            printf("%c ", t + 'a');
            visited[t] = 1;
        } // Mark the current node as visited

        int found = 0; // Flag to check if a neighbor is found

        for (int i = 0; i < g->size; i++) {
            if (g->matrix[t][i] == 1 && !visited[i]) {
                push(s, i);
                found = 1;
                break; // Break after finding the first unvisited neighbor
            }
        }

        if (!found) {
            pop(s); // Pop only if no unvisited neighbor is found
        }
    }
```

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

Output:

```
C:\Users\SSN\OneDrive\Desktop>a
Enter the number of vertices: 6
Enter the number of edges: 7
Enter the edge pair: a b
Enter the edge pair: b c
Enter the edge pair: c a
Enter the edge pair: c f
Enter the edge pair: e f
Enter the edge pair: d e
Enter the edge pair: b d
13828288        1       454695192       522067228       589439264       656811300
0       0       1       1       0       0
1       0       0       0       0       1
0       0       0       0       1       0
0       0       0       0       0       1
0       0       0       0       0       0
Enter the starting vertex for BFS and DFS: a
BFS traversal:abcdef

DFS traversal:a b c f d e

C:\Users\SSN\OneDrive\Desktop>
```

2.
Code:

GraphADT.h

```
#include <stdio.h>
#include <stdlib.h>
#include "queue.c"
#include "stack.c"

struct graph {
    int matrix[26][26];
    int size;
};
```

```
struct edgePair {
    int from;
    int to;
};

void create(struct graph *g, int v, int e, struct edgePair edges[]) {
    g->size = v;
    int from, to;
    for (int i = 0; i < e; i++) {
        from = edges[i].from;
        to = edges[i].to;
        g->matrix[from][to] = 1;
        g->matrix[to][from] = 1; // Assuming undirected graph
    }
}

int hasPath(struct graph *g, int source, int destination) {
    struct queue *q = (struct queue *)malloc(sizeof(struct queue));
    int visited[26] = {0};
    createQueue(q, g->size);
    visited[source] = 1;
    enqueue(q, source);

    while (!queueIsEmpty(q)) {
        int current = dequeue(q);
        if (current == destination) {
            free(q);
            return 1; // Path found
        }

        for (int i = 0; i < g->size; i++) {
            if (g->matrix[current][i] && !visited[i]) {
                visited[i] = 1;
                enqueue(q, i);
            }
        }
    }


    return 0; // No path found
```

```
}

void DFSUtil(struct graph *g, int v, int visited[]) {
    visited[v] = 1;
    printf("%d ", v);

    for (int i = 0; i < g->size; i++) {
        if (g->matrix[v][i] && !visited[i]) {
            DFSUtil(g, i, visited);
        }
    }
}

void connectedComponents(struct graph *g) {
    int visited[26] = {0};

    printf("Connected Components:\n");
    for (int i = 0; i < g->size; i++) {
        if (!visited[i]) {
            DFSUtil(g, i, visited);
            printf("\n");
        }
    }
}
```

Main.h

```
int main() {
    struct graph *g;
    int vertices = 5;
    int edges = 4;
    struct edgePair edgeArray[] = {{0, 1}, {1, 4}, {2, 3}, {3, 4}};

    create(g, vertices, edges, edgeArray);

    int source = 0;
    int destination = 4;

    if (hasPath(g, source, destination)) {
        printf("There is a path from %d to %d.\n", source, destination);
```

```
    } else {
        printf("There is no path from %d to %d.\n", source, destination);
    }

    connectedComponents(g);

    return 0;
}
```

Output: