Date: 14.11.2023 Name: Niranjan.B
Ex. No: 9 Reg. No.: 3122225001082

UCS 2312 Data Structures Lab

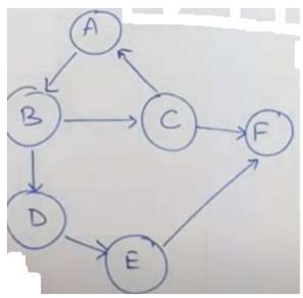
Assignment 9: Graph Traversal and its Applications

Date of Assignment: 14.11.2023

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.

- 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 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

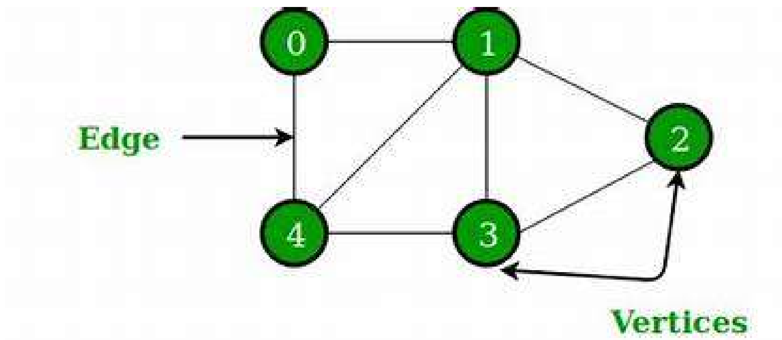
BFS Output: ABCDFE for Start vertex A

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

Data Structure – Graph:




```
struct graph
{
    int adj[100][100];
    int v;
};

struct pair
{
    int first;
    int second;
};
```

Algorithm –
Algorithm: will create the graph using adjacency matrix
Input – Pointer to Graph, no. of vertices, no. of edges, array of pairs
Output – void

1. G->v=v
2. for(i=0;i<e;i++)
   if directed graph
   G->adj[pairs[i].first][pairs[i].second]=1
   else
   G->adj[pairs[i].first][pairs[i].second]=1
   G->adj[pairs[i].second][pairs[i].first]=1

Algorithm: display the adjacency matrix
Input – Pointer to Graph
Output – void

```
1.  i=1 and j=1
2.  while(i<=G->v)
        while(j<=G->v)
                print G->adj[i][j]
        print a new line
```

Algorithm: provides the output of visiting the cities following breadth first
Input – Pointer to Graph, starting vertex x
Output – void

```
1.  create a queue Q
2.  visit x
3.  enqueue x
4.  while(Q is not empty)
        z=dequeue Q
        i=1
        while(i<=G->v)
                if(G->adj[z][i]==1 && vis[i]!=1)
                        visit i
                        enqueue i
                i++
```

Algorithm: provides the output of visiting the cities by following depth first
Input – Pointer to Graph, starting vertex x
Output – void

```
1.  create a stack S
2.  visit x
3.  push x
4.  while(S is not empty)
        i=1
        while(i<=G->v)
                t=peek of S
                if(G->adj[z][i]==1 && vis[i]!=1)
                        visit i
                        push i
                i++
        pop S
```

Algorithm: finds whether path exists or not
Input – Pointer to Graph, source, destination
Output – int

```
1.  create a stack S
2.  if(source==destination)
        return 1
3.  visit source
4.  push source
5.  while(S is not empty)
        i=1
        while(i<=G->v)
                t=peek of S
```

```
        if(G->adj[z][i]==1 && vis[i]!=1)
            if(destination==i)
                return i
            visit i
            push i
        i++
    pop S
```

Algorithm: find the connected components

Input – Pointer to Graph

Output – void

1. visited[G->v+1]
2. i=1
3. while(i<=G->v)

```
    if(visited[i]!=1)
        DFS(G, i, visted)
        print new line
    i++
```

queue.h code:

```
struct queue{
	int arr[100];
	int size;
	int front, rear;
};

void createQueue(struct queue* q, int size){
	q->size = size;
	q->front = q->rear = -1;
}

int isQueueFull(struct queue* q){
	if(q->rear + 1 >= q->size) return 1;
	else return 0;
}

int isQueueEmpty(struct queue* q){
	if(q->rear == -1 && q->front == -1) return 1;
	else if(q->front > q->rear){
		q->front = q->rear = -1;
		return 1;
	}
	else return 0;
}

void enqueue(struct queue* q, int data){
	if(isQueueFull(q)){
		printf("\nQueue is full");
	}
	else{
		if(q->rear == -1){
			q->front++;
```

```
            }
            q->rear++;
            q->arr[q->rear] = data;
      }
}

int dequeue(struct queue* q){
      if(isQueueEmpty(q)){
            printf("\nQueue is empty");
            return -1;
      }
      else{
            int data = q->arr[q->front];
            q->front++;
            return data;
      }
}
```

stack.h code:

```
struct stack{
      int arr[100];
      int size;
      int top;
};

void createStack(struct stack *s, int size){
      s->size = size;
      s->top = -1;
}

int isStackEmpty(struct stack *s){
      if(s->top == -1) return 1;
      else return 0;
}

int isStackFull(struct stack *s){
      if(s->top + 1 >= s->size) return 1;
      else return 0;
}

void push(struct stack *s, int data){
      if(isStackFull(s)){
            printf("\nStack is Full");
      }
      else{
            s->top += 1;
            s->arr[s->top] = data;
      }
}

int pop(struct stack *s){
      if(isStackEmpty(s)){
            return -1;
      }
```

```
	else{
		int val = s->arr[s->top];
		s->top -= 1;
		return val;
	}
}

int peek(struct stack *s){
	if(isStackEmpty(s)){
		return -1;
	}
	else{
		return s->arr[s->top];
	}
}
```

graph.h code:

```
#include "stack.h"
#include "queue.h"

struct graph
{
	int adj[100][100];
	int v;
};

struct pair
{
	int first;
	int second;
};

void create(struct graph *G, int v, int e, struct pair pairs[], char c)
{
	G->v=v;
	for(int i=0;i<e;i++)
	{
		if(c=='n' || c=='N')
		{
			G->adj[pairs[i].first][pairs[i].second]=1;
			G->adj[pairs[i].second][pairs[i].first]=1;
		}
		else if(c=='y' || c=='Y')
		{
			G->adj[pairs[i].first][pairs[i].second]=1;
		}
	}
}

void display(struct graph *G)
{
	printf("  ");
	for(int i=1;i<=G->v;i++)
		printf("%c ",(char)(i+64));
```

```
	printf("\n");
	for(int i=1;i<=G->v;i++)
	{
		printf("%c ",(char)(i+64));
		for(int j=1;j<=G->v;j++)
		{
			printf("%d ",G->adj[i][j]);
		}
		printf("\n");
	}
}

void visit(int vis[], int x)
{
	vis[x]=1;
	printf("%c ",(char)(x+64));
}

void BFS(struct graph *G, int x)
{
	struct queue *Q=(struct queue*)malloc(sizeof(struct queue));
	createQueue(Q,G->v);
	int vis[G->v+1];
	visit(vis,x);
	enqueue(Q,x);
	while(!isQueueEmpty(Q))
	{
		int z=dequeue(Q);
		for(int i=1;i<=G->v;i++)
		{
			if(G->adj[z][i] == 1 && vis[i]!=1)
			{
				visit(vis,i);
				enqueue(Q,i);
			}
		}
	}
}

void DFS(struct graph *G, int x)
{
	struct stack *S=(struct stack*)malloc(sizeof(struct stack));
	createStack(S,G->v);
	int vis[G->v+1];
	visit(vis,x);
	push(S,x);
	while(!isStackEmpty(S))
	{
		for(int i=1;i<=G->v;i++)
		{
			int t=peek(S);
			if(G->adj[t][i] == 1 && vis[i]!=1)
			{
				visit(vis,i);
```

```
                        push(S,i);
                    }
                }
                pop(S);
        }
}

int path(struct graph *G, int source, int destination)
{
        struct stack *S=(struct stack*)malloc(sizeof(struct stack));
        createStack(S,G->v);
        if(source==destination)
                return 1;
        int vis[G->v+1];
        vis[source]=1;
        push(S,source);
        while(!isStackEmpty(S))
        {
                for(int i=1;i<=G->v;i++)
                {
                        int t=peek(S);
                        if(G->adj[t][i] == 1 && vis[i]!=1)
                        {
                                if(destination==i)
                                        return 1;
                                vis[i]=1;
                                push(S,i);
                        }
                }
                pop(S);
        }
        return 0;
}

void DFS1(struct graph *G, int x, int vis[])
{
        struct stack *S=(struct stack*)malloc(sizeof(struct stack));
        createStack(S,G->v);
        visit(vis,x);
        push(S,x);
        while(!isStackEmpty(S))
        {
                for(int i=1;i<=G->v;i++)
                {
                        int t=peek(S);
                        if(G->adj[t][i] == 1 && vis[i]!=1)
                        {
                                visit(vis,i);
                                push(S,i);
                        }
                }
                pop(S);
        }
}
```

```
void connectedComponents(struct graph* G)
{
    int visited[G->v+1];
    printf("Connected Components:\n");
    for(int i=1;i<=G->v;i++)
      {
        if (visited[i]!=1)
           {
            DFS1(G, i, visited);
            printf("\n");
        }
    }
}
```

graph.c code:

```
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"

void main()
{
	int choice=1;
	int v,e;
	char first,second;
	char source,destination;
	printf("Vertices = ");
	scanf("%d",&v);
	printf("Edges = ");
	scanf("%d",&e);
	printf("Directed (y|n) = ");
	while ((getchar()) != '\n');
	char c=getchar();
	printf("Edge pairs:\n");
	struct pair pairs[e];
	for(int i=0;i<e;i++)
	{
		printf("First and Second Point = ");
		while ((getchar()) != '\n');
		scanf("%c %c",&first,&second);
		pairs[i].first=(int)first-64;
		pairs[i].second=(int)second-64;
	}
	struct graph *G=(struct graph*)malloc(sizeof(struct graph));
	create(G, v, e, pairs, c);
	while(choice)
	{
		printf("\n\n1.Display\n2.BFS AND DFS\n3.Find Path\n4.Connected 
Components\n5.Exit\nChoice : ");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1:
				display(G);
```

```
                    break;
                case 2:
                {
                    printf("Staring point = ");
                    while ((getchar()) != '\n');
                    char x=getchar();
                    printf("BFS = ");
                    BFS(G,((int)x-64));
                    printf("\nDFS = ");
                    DFS(G,((int)x-64));
                    break;
                }
                case 3:
                {
                    printf("Source = ");
                    while ((getchar()) != '\n');
                    char source=getchar();
                    printf("Destination = ");
                    while ((getchar()) != '\n');
                    char destination=getchar();
                    if(path(G, ((int)source-64), ((int)destination-
64)))
                        printf("Path exists");
                    else
                        printf("Path not exists");
                    break;
                }
                case 4:
                {
                    connectedComponents(G);
                    break;
                }
                case 5:
                choice=0;
                break;
                default:
                printf("Invalid Choice");
            }
        }
}
```

Output Screen:

```
PS D:\College\Sem 3\Data Structures\Graph> gcc graph.c
PS D:\College\Sem 3\Data Structures\Graph> ./a.exe
Vertices = 6
Edges = 7
Directed (y|n) = y
Edge pairs:
First and Second Point = A B
First and Second Point = B C
First and Second Point = B D
First and Second Point = C A
First and Second Point = C F
First and Second Point = D E
First and Second Point = E F


1.Display
2.BFS AND DFS
3.Find Path
4.Connected Components
5.Exit
Choice : 1
  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


1.Display
2.BFS AND DFS
3.Find Path
4.Connected Components
5.Exit
Choice : 2
Staring point = A
BFS = A B C D F E
DFS = A B C F D E
1.Display
2.BFS AND DFS
3.Find Path
4.Connected Components
5.Exit
Choice : 3
Source = D
Destination = F
Path exists

1.Display
2.BFS AND DFS
3.Find Path
4.Connected Components
5.Exit
Choice : 3
Source = F
Destination = B
Path not exists

1.Display
2.BFS AND DFS
3.Find Path
4.Connected Components
5.Exit
Choice : 4
Connected Components:
A B C F D E
```

Learning Outcome:

Learning Outcome		
Design	3	Understood the design of graphs
Understanding of DS	3	Clear with its operations like
Use of DS	3	BFS and DFS
Debugging	3	Was able to fix errors properly
Best Practices		
Design before coding	3	Designed properly
Use of algorithmic notation	2	Can be improved
Use of multifile C program	3	Used multiple files.
Versioning of code	3	Versioned properly