Date: 18.11.2023 Name: Niranjan.B
Ex. No: 10 Reg. No.: 3122225001082

**UCS 2312 Data Structures Lab**

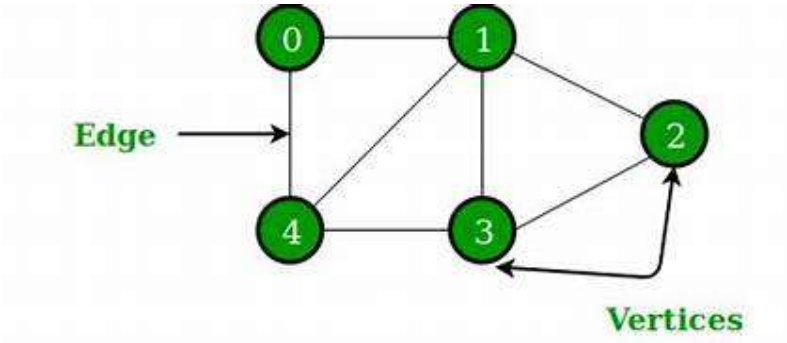
**Assignment 10: Implementation of Shortest Path Finding algorithm**

**Date of Assignment: 18.11.2023**

The cityADT contains the number of cities and the connectivity information between the cities (adjacency matrix). Write the following methods. [CO2, K3]

- void create(cityADT *C) – will represent the graph using adjacency matrix
- void disp(cityADT *C) – Display the graph
- void Dijkstra(cityADT *C) – Displays the intermediate and final tables
- char * displayPath(cityADT *C, source, destination) – Find the path of the intermediate cities between the source and destination cities along with the cost

**Data Structure – Graph:**




```
struct table
{
    int v,k,d,p;
};
struct graph
{
    int adj[100][100];
    int v;
    struct table t[100];
};

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

**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: Displays the intermediate and final tables**

Input – Pointer to Graph, starting vertex x
Output – void

1. while(there is an unknown vertex)
   print table
   vertex v=smallest dist unknown vertex
   v.known=true
   for each vertex w adjacent to v
   if(!w.known)
   c=cost of edge from v to w
   if(distance v + c < distance w)
   distance w=distance v + c
   path w=v

**Algorithm: Find the path of the intermediate cities between the source and destination cities along with the cost**

Input – Pointer to Graph, destination vertex v
Output – void

1. if(path v != -1)
   path(G,path v)
2. print v

**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 table
{
      int v,k,d,p;
};
struct graph
{
      int adj[100][100];
      int v;
```

```
	struct table t[100];
};

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

void create(struct graph *G, int v, int e, struct pair pairs[])
{
	G->v=v;
	for(int i=0;i<e;i++)
	{
		G->adj[pairs[i].first][pairs[i].second]=pairs[i].weight;
	}
}

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("%d ",x);
}

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 visit[G->v+1];
    visit[x]=1;
    printf("%d ",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 && visit[i]!=1)
            {
                visit[i]=1;
                push(S,i);
                printf("%d ",i);
            }
        }
        pop(S);
    }
}
```

**dijkstra.h code:**

```
#include "graph.h"

void init(struct graph *G)
{
    for(int i=1;i<=G->v;i++)
    {
        G->t[i].k=0;
        G->t[i].d=999;
        G->t[i].p=-1;
        G->t[i].v=i;
    }
}

void printTable(struct graph *G)
{
    printf("\n\nV K D P");
    for(int i=1;i<=G->v;i++)
    {
        printf("\n%d %d %d %d",G->t[i].v,G->t[i].k,G->t[i].d,G-
>t[i].p);
```

```
	}
}

int check(struct graph *G)
{
	for(int i=1;i<=G->v;i++)
	{
		if(G->t[i].k==0)
			return 1;
	}
	return 0;
}

int minimum(struct graph *G)
{
	int min=999;
	int v=-1;
	for(int i=1;i<=G->v;i++)
	{
		if(G->t[i].d<min && G->t[i].k==0)
		{
			min=G->t[i].d;
			v=i;
		}
	}
	return v;
}

void dijkstra(struct graph *G, int s)
{
	init(G);
	int v;
	G->t[s].d=0;
	while(check(G))
	{
		printTable(G);
		v=minimum(G);
		G->t[v].k=1;
		for(int i=1;i<=G->v;i++)
		{
			if(G->adj[v][i]!=0 && G->t[i].k==0)
			{
				if(G->t[i].d>(G->t[v].d + G->adj[v][i]))
				{
					G->t[i].d=G->t[v].d + G->adj[v][i];
					G->t[i].p=v;
				}
			}
		}
	}
	printTable(G);
}

void path(struct graph *G,int v)
```

```
{
      if(G->t[v].p!=-1)
      {
            path(G,G->t[v].p);
            printf("->");
      }
      printf("%c",v+64);
}
```

**dijkstra.c code:**

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

void main()
{
      int choice=1;
      int v,e;
      char c;
      char first,second;
      printf("Vertices = ");
      scanf("%d",&v);
      printf("Edges = ");
      scanf("%d",&e);
      printf("Edge pairs:\n");
      struct pair pairs[e];
      for(int i=0;i<e;i++)
      {
            printf("First, Second Point and distance= ");
            while ((c = getchar()) != '\n' && c != EOF) {}
            scanf("%c %c %d",&first,&second,&pairs[i].weight);
            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);
      display(G);
      char x;
      printf("Staring point = ");
      while ((c = getchar()) != '\n' && c != EOF) {}
      scanf("%c",&x);
      printf("\n");
      dijkstra(G,(int)x-64);
      printf("\n");
      printf("Destination = ");
      while ((c = getchar()) != '\n' && c != EOF) {}
      scanf("%c",&x);
      path(G,(int)x-64);
      printf("\n");
}
```

Name: Niranjan.B
Reg. No.: 3122225001082

**Output Screen:**

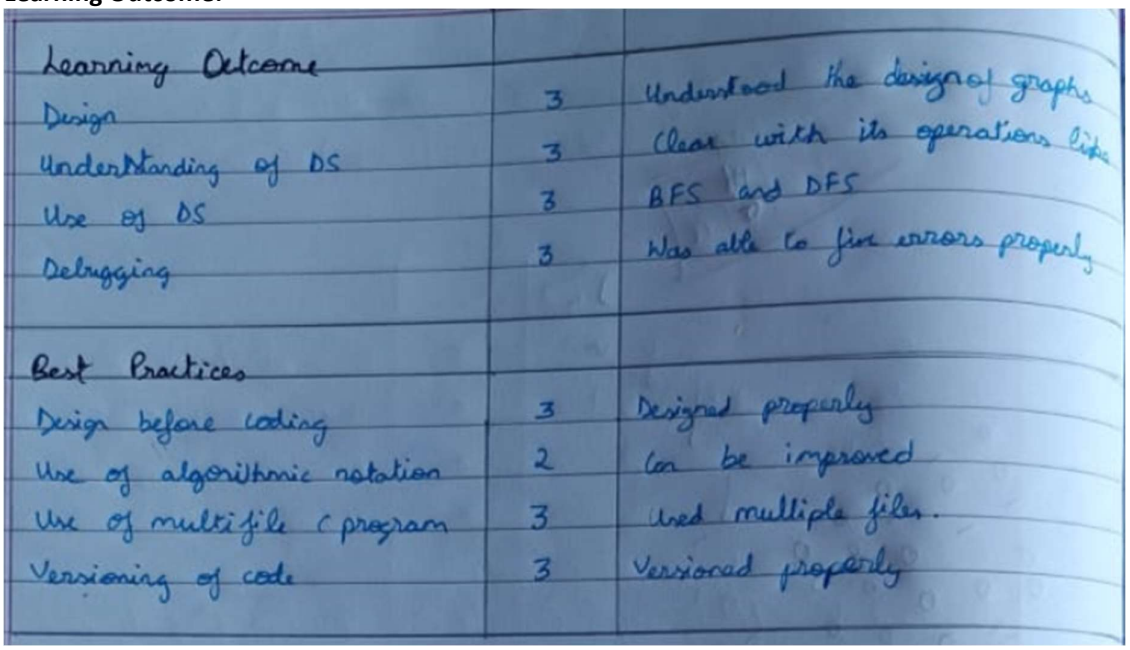
```
PS D:\College\Sem 3\Data Structures\Dijkstra> gcc dijkstra.c
PS D:\College\Sem 3\Data Structures\Dijkstra> ./a.exe
Vertices = 5
Edges = 8
Edge pairs:
First, Second Point and distance= A B 5
First, Second Point and distance= A E 1
First, Second Point and distance= B C 6
First, Second Point and distance= C A 2
First, Second Point and distance= C D 1
First, Second Point and distance= C E 4
First, Second Point and distance= E C 2
First, Second Point and distance= E D 6
  A B C D E
A 0 5 0 0 1
B 0 0 6 0 0
C 2 0 0 1 4
D 0 0 0 0 0
E 0 0 2 6 0
Staring point = A



V K D P
1 0 0 -1
2 0 999 -1
3 0 999 -1
4 0 999 -1
5 0 999 -1

V K D P
1 1 0 -1
2 0 5 1
3 0 999 -1
4 0 999 -1
5 0 1 1

V K D P
1 1 0 -1
2 0 5 1
3 0 3 5
4 0 7 5
5 1 1 1
```

```
V K D P
1 1 0 -1
2 0 5 1
3 1 3 5
4 0 4 3
5 1 1 1

V K D P
1 1 0 -1
2 0 5 1
3 1 3 5
4 1 4 3
5 1 1 1

V K D P
1 1 0 -1
2 1 5 1
3 1 3 5
4 1 4 3
5 1 1 1
Destination = D
A->E->C->D
```

Date: 18.11.2023
Ex. No: 10

Name: Niranjan.B
Reg. No.: 3122225001082

**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 |