Dijkstra on sparse graphs

Aim: To find the shortest distance between one source node and all other nodes of a graph(where nodes are cities and edges represent the path between various cities) using Dijkstra algorithm in O(n.logn) .

Description:

Algorithm:-

We recall in the derivation of the complexity of Dijkstra's algorithm we used two factors: the time of finding the unmarked vertex with the smallest distance d[v]d[v], and the time of the relaxation, i.e. the time of changing the values d[to]d[to].

In the simplest implementation these operations require O(n)O(n) and O(1)O(1) time. Therefore, since we perform the first operation O(n)O(n)times, and the second one O(m)O(m) times, we obtained the complexity O(n2+m)O(n2+m).

It is clear, that this complexity is optimal for a dense graph, i.e. when m≈n2m≈n2. However in sparse graphs, when mm is much smaller than the maximal number of edges n2n2, the complexity gets less optimal because of the first term. Thus it is necessary to improve the execution time of the first operation (and of course without greatly affecting the second operation by much).

To accomplish that we can use a variation of multiple auxiliary data structures. The most efficient is the **Fibonacci heap**, which allows the first operation to run in O(logn), and the second operation in O(1). Therefore we will get the complexity O(nlogn+m) for Dijkstra's algorithm, which is also the theoretical minimum for the shortest path search problem. Therefore this algorithm works optimal, and Fibonacci heaps are the optimal data structure. There doesn't exist any data structure, that can perform both operations in O(1), because this would also allow to sort a list of random numbers in linear time, which is impossible. Interestingly there exists an algorithm by Thorup that finds the shortest path in O(m) time, however only works for integer weights, and uses a completely different idea. So this doesn't lead to any contradictions. Fibonacci heaps provide the optimal complexity for this task. However they are quite complex to implement, and also have a quite large hidden constant.

As a compromise you can use data structures, that perform both types of operations (extracting a minimum and updating an item) in O(logn). Then the complexity of Dijkstra's algorithm is O(nlogm+mlogn)=O(mlogn).

Therefore `priority_queue` has a smaller hidden constant, but also has a drawback: it doesn't support the operation of removing an element. Because of this we need to do a "workaround", that actually leads to a slightly worse factor logm instead of logn (although in terms of complexity they are identical).

Priority queue:-
The main difference to the implementation with **`set`** is that we cannot remove elements from the **`priority_queue`** (although heaps can support that operation in theory). Therefore we have to cheat a little bit. We simply don't delete the old pair from the queue. As a result a vertex can appear multiple times with different distance in the queue at the same time. Among these pairs we are only interested in the pairs where the first element is equal to the corresponding value in $d[]$, all the other pairs are old. Therefore we need to make a small modification: at the beginning of each iteration, after extracting the next pair, we check if it is an important pair or if it is already an old and handled pair. This check is important, otherwise the complexity can increase up to $O(nm)$.

By default a **`priority_queue`** sorts elements in descending order. To make it sort the elements in ascending order, we can either store the negated distances in it, or pass it a different sorting function. We will do the second option.

Data Structures:

1.Graphs:

To Store the City details and its connections in form of a adjacency list.

2.Linked List:

The element of the graph structure is a linked list. The linked list is represented by a array of pointers to nodes where each node has city details and distance from the I th city.

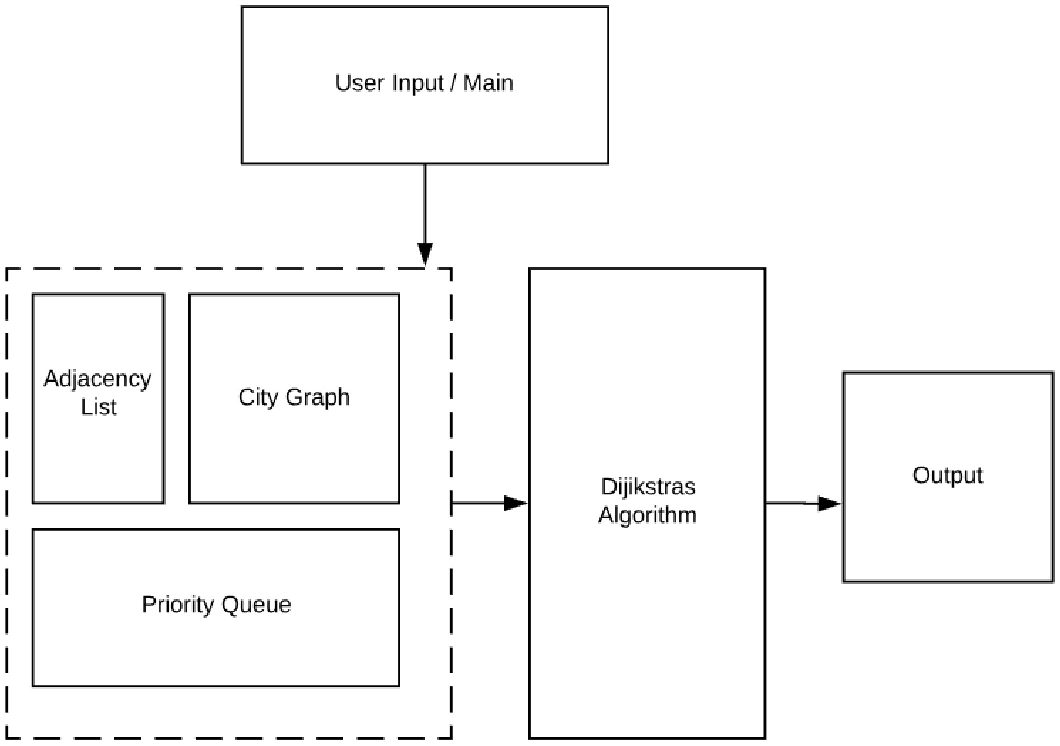
3.Priority Queue:

Priority queue is built using binary min heap using a array.

4.Arrays

Used to store city details and other distance details. Arrays are also used in dijikstra implementation.

Project layout:



Learning Experience:

1.I have learnt to arrange the data using arrays, linked list, graphs and priority queue.

2.The type of data used in graph is adjacency list and size of the graph.

3.The type of data stored in each node of adjacency list is city number and distance.

4.The type of data stored in the priority queue is node pointer ,size and capacity.

5.The operation performed in priority queue is insert and delete.

6.The operation performed in graphs is insertion of a city, deletion of a city ,display city connection details and find shortest distance using dijikstra algorithm.

7.Learnt to create ADT with data members and member functions.

8.Learnt to develop an application using graphs.

Code :

Interface File:

```
const int INF = 1e+9;
const int MAXN= 1e+5;
int d[100000];

typedef struct {
	int city_no;
	int distance;
}node;

typedef struct {
	int capacity;
	int size;
	node *e;
}pq;

struct listnode{
	node element;
	struct listnode *next;
};
typedef struct listnode listnode;

typedef struct{
	int size;
	listnode *a[1000];
}citygraph;

void _initialize(pq *q,int max);
void insert_in_pq(pq *q,node e);
node del(pq *q);
citygraph *initgraph();
void insert(citygraph *t);
void delete(citygraph *t,int cityno);
void display(citygraph *t);
```

implementation:

```
citygraph *initgraph()
{
	citygraph *t=(citygraph *)malloc(sizeof(citygraph));
	t->size=0;
	int i;
	for(i=0;i<1000;i++)
	{
```

```
                t->a[i]=(listnode *)malloc(sizeof(listnode));
        }
        return t;
}

int findcity(citygraph *t,int city)
{

                if((t->a[city])->element.city_no==city)
                        {
                                return city;
                        }

                return -1;
}

void insert(citygraph *t)
{
        int fcity,tcity,dist,ch;
        printf("Enter city no of city to be inserted:");
        scanf("%d",&fcity);
        int k=findcity(t,fcity);
        if(k==-1)
        {
                (t->a[fcity])->element.city_no=fcity;
                (t->a[fcity])->element.distance=0;
                (t->a[fcity])->next=NULL;
                t->size++;
        }
        printf("\nDo you want to make any connections(1 if yes):");
        scanf("%d",&ch);
        while(ch==1)
        {
        printf("\nEnter the city to which edge has to be made:");
        scanf("%d",&tcity);
        printf("Enter distance between them : ");
        scanf("%d",&dist);
        int m=findcity(t,tcity);
        if(m==-1)
        {
                (t->a[tcity])->element.city_no=tcity;
                (t->a[tcity])->element.distance=0;
                (t->a[tcity])->next=NULL;
                t->size++;
        }
        k=findcity(t,fcity);
        listnode *p=(listnode *)malloc(sizeof(listnode));
```

```
	p->element.city_no=tcity;
	p->element.distance=dist;
	p->next=(t->a[k])->next;
	(t->a[k])->next=p;
	printf("\nDo you want to make any connections(1 if yes):");
	scanf("%d",&ch);
	}
}

void delete(citygraph *t,int cityno)
{
	int i,j;
	int k=findcity(t,cityno);
	(t->a[k])->element.city_no=-1;
	(t->a[k])->next=0;

	for(i=0;i<1000;i++)
	{
		int m=findcity(t,i);
		if(m==i)
		{
		listnode *temp=(t->a[i])->next;
		listnode *prev=t->a[i];
		while(temp!=NULL)
		{
			if(temp->element.city_no==cityno)
			{
				prev->next=temp->next;
				break;
			}
			prev=temp;
			temp=temp->next;
		}
		}
	}

	t->size--;
}

void display(citygraph *t){
	int i;
	for(i=0;i<1000;i++){
		if(i==findcity(t,i)){
			if(((t->a[i])->next)!=NULL){
				printf("\ndistance from %d :",(t->a[i])->element.city_no);
				listnode *temp=(t->a[i])->next;
				while(temp!=NULL){
```

```
                                                            printf("(%2d,%3d)",temp-
>element.city_no,temp->element.distance);
                                                            temp=temp->next;
                                        }
                              }
                    }
          }
}

void _initialize(pq *q,int max){
          q->e=malloc(sizeof(node)*max);
          q->capacity=max;
          q->size=0;
          q->e[0].distance=-1;
}

void insert_in_queue(pq *q,node x){
          int i;
          if(q->size==q->capacity){
                    printf("MEMORY IS FULL!!!\n");
                    return ;
          }
          for(i=++q->size;q->e[i/2].distance>x.distance;i/=2)
                    q->e[i]=q->e[i/2];
          q->e[i]=x;
}


node del(pq *q){
          if(q->size==0){
                    node y;
                    y.distance=INF;
                    printf("MEMORY IS EMPTY!!!\n");
                    return y;
          }
          int i, child;
          node minelement, lastelement;
          minelement=q->e[1];
          lastelement=q->e[q->size--];
          for(i=1;(i*2)<=q->size;i=child){
                    child=i*2;
                    if(q->e[child+1].distance < q->e[child].distance)
                              child++;
                    if(lastelement.distance > q->e[child].distance)
                              q->e[i]=q->e[child];
                    else
                              break;
```

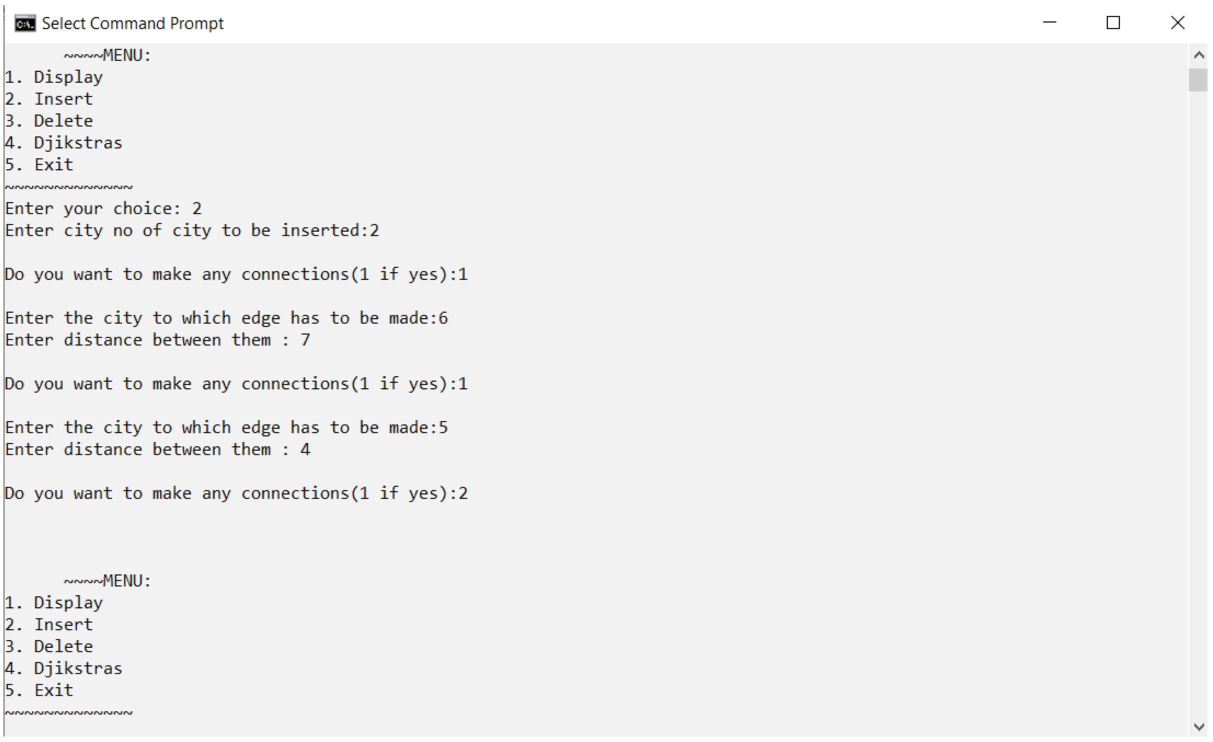
```
        }
        q->e[i]=lastelement;
        return minelement;
}

void Djik(citygraph *t, int s){
        int i=0;
        for(;i<=MAXN; i++)d[i]=INF;
        d[s] = 0;
        pq q;_initialize(&q, 100);
        node x; x.city_no=s; x.distance=0;
        insert_in_queue(&q,x);
        while ((q.size)) {
                x=del(&q);
                int v=x.city_no;
                int d_v=x.distance;

                if (d_v != d[v])
                   continue;

                listnode *temp=(t->a[v])->next;
                while(temp!=NULL){
                        int to=temp->element.city_no;
                        int len=temp->element.distance;
                        if (d[v] + len < d[to])
                        {
                                d[to] = d[v] + len;
                                x.city_no=to; x.distance=d[to];
                                insert_in_queue(&q,x);
                        }
                        temp=temp->next;
                }

        }

}
```

Application:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<math.h>
#include "pif2.h"
#include "pimpl2.h"
```

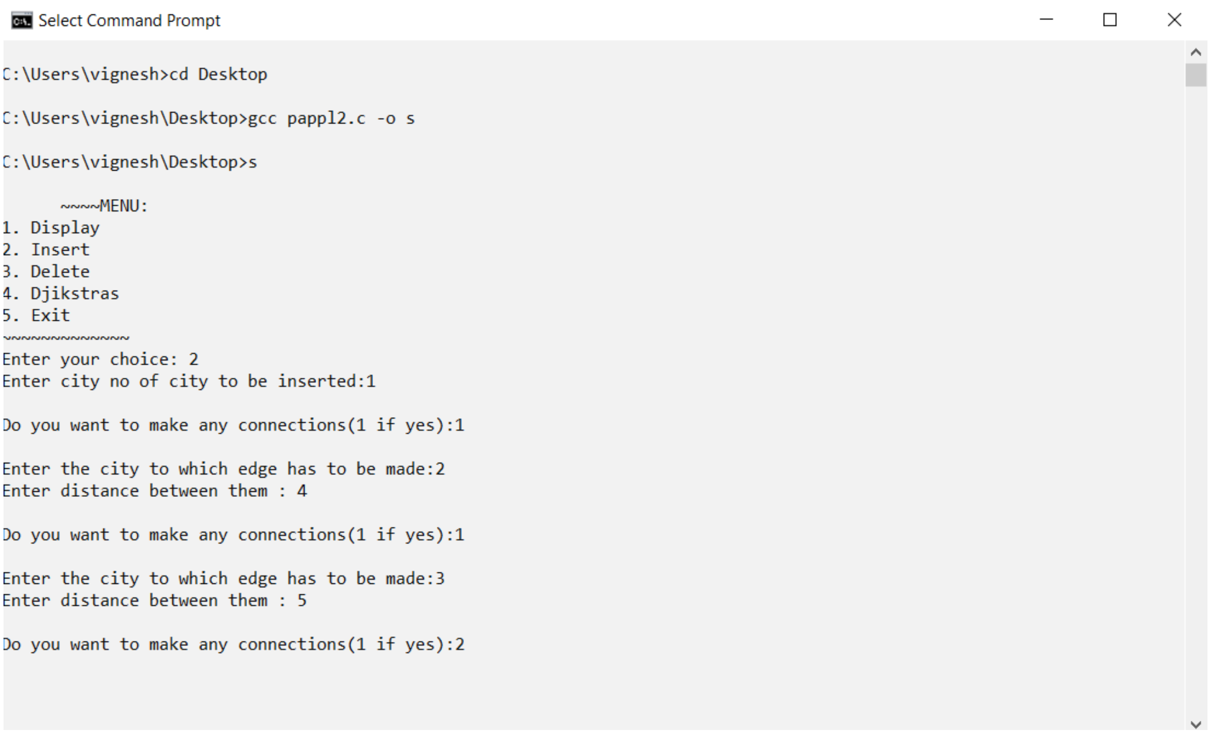
```
void main()
{
	citygraph *t=initgraph();
	int ch;
	do
	{
		printf("\n      ~~~~MENU:\n1. Display\n2. Insert\n3. Delete\n4. Djikstras\n5.
Exit\n~~~~~~~~~~~~~~\nEnter your choice: ");

		scanf("%d",&ch);
		if(ch==2){
			insert(t);
		}
		if(ch==1){
			display(t);
		}
		if(ch==3){
			int cityno;
			printf("Enter city no to be deleted");
			scanf("%d",&cityno);
			delete(t,cityno);
		}
		if(ch==4){
			int i;
			int s;
			printf("Enter the source vertex: ");
			scanf("%d", &s);
			Djik(t, s);
			printf("\nSOURCE->VERTEX\tDISTANCE\n");
			for(i=1; i<=MAXN; i++){
				if(d[i]!=INF)
					printf("   %d->%d\t\t %d\n", s,i,d[i]);
			}

		}
		printf("\n\n");

	}while(ch!=5);
}
```

Output:



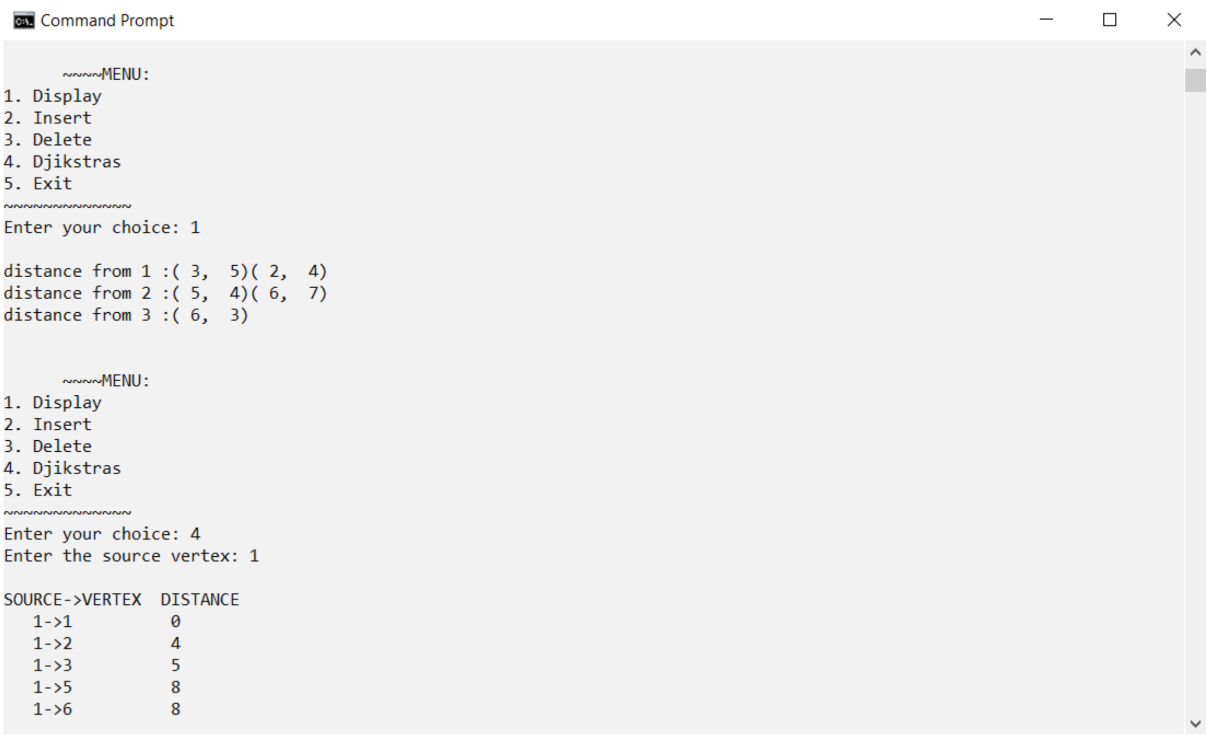
Select Command Prompt

```
      ~~~~MENU:
1. Display
2. Insert
3. Delete
4. Djikstras
5. Exit
~~~~~~~~~~~~~
Enter your choice: 2
Enter city no of city to be inserted:2

Do you want to make any connections(1 if yes):1

Enter the city to which edge has to be made:6
Enter distance between them : 7

Do you want to make any connections(1 if yes):1

Enter the city to which edge has to be made:5
Enter distance between them : 4

Do you want to make any connections(1 if yes):2



      ~~~~MENU:
1. Display
2. Insert
3. Delete
4. Djikstras
5. Exit
~~~~~~~~~~~~~
```

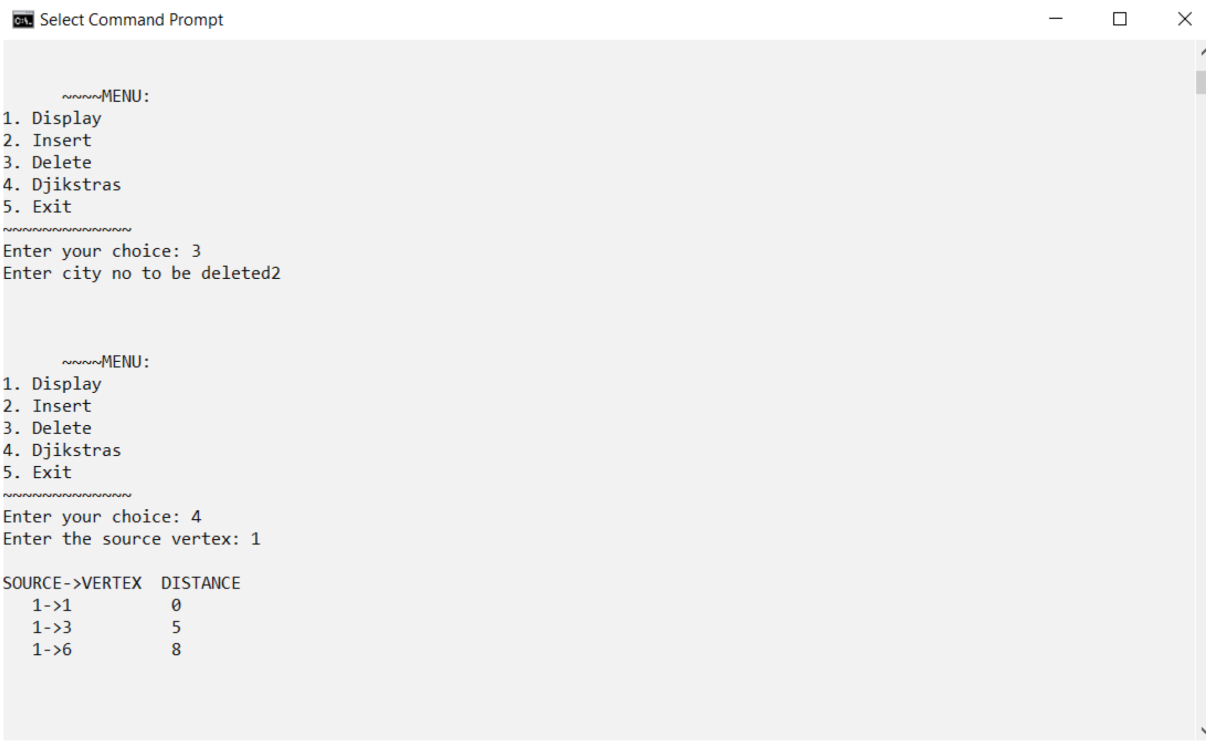


Select Command Prompt

```
C:\Users\vignesh>cd Desktop

C:\Users\vignesh\Desktop>gcc pappl2.c -o s

C:\Users\vignesh\Desktop>s

      ~~~~MENU:
1. Display
2. Insert
3. Delete
4. Djikstras
5. Exit
~~~~~~~~~~~~~
Enter your choice: 2
Enter city no of city to be inserted:1

Do you want to make any connections(1 if yes):1

Enter the city to which edge has to be made:2
Enter distance between them : 4

Do you want to make any connections(1 if yes):1

Enter the city to which edge has to be made:3
Enter distance between them : 5

Do you want to make any connections(1 if yes):2
```



```
      ~~~~MENU:
1. Display
2. Insert
3. Delete
4. Djikstras
5. Exit
~~~~~~~~~~~~~
Enter your choice: 1

distance from 1 :( 3,  5)( 2,  4)
distance from 2 :( 5,  4)( 6,  7)
distance from 3 :( 6,  3)


      ~~~~MENU:
1. Display
2. Insert
3. Delete
4. Djikstras
5. Exit
~~~~~~~~~~~~~
Enter your choice: 4
Enter the source vertex: 1

SOURCE->VERTEX  DISTANCE
   1->1          0
   1->2          4
   1->3          5
   1->5          8
   1->6          8
```



```
      ~~~~MENU:
1. Display
2. Insert
3. Delete
4. Djikstras
5. Exit
~~~~~~~~~~~~~
Enter your choice: 3
Enter city no to be deleted2



      ~~~~MENU:
1. Display
2. Insert
3. Delete
4. Djikstras
5. Exit
~~~~~~~~~~~~~
Enter your choice: 4
Enter the source vertex: 1

SOURCE->VERTEX  DISTANCE
   1->1          0
   1->3          5
   1->6          8
```