THE NEOTIA UNIVERSITY

SCHOOL OF SCIENCE & TECHNOLOGY

LAB MANUAL

ANALYSIS AND DESIGN OF ALGORITHMS LAB

(PC-CSEP601)

COMPUTER SCIENCE & ENGINEERING (AIML)

The Neotia University

School of Science & Technology, CSE (AIML)

Sixth Semester

Analysis and Design of Algorithms
(PC – CSEP601)

List of experiments/Lab Assignments

1. Implement iterative binary search.
2. Implement recursive binary search.
3. Implement divide and conquer to find the maximum and minimum.
4. Implement Quick sort
5. Implement Merge sort.
6. Implement Strassen's Matrix Multiplication.
7. Implement Floyd-Warshal algorithm.
8. Implement Kruskal's algorithm for minimum spanning trees
9. Implement Prim's algorithm for minimum spanning trees
10. Implement single sources shortest path algorithm.

Lab Assignment#1

Objective: To learn the concepts of iterative binary search.

Problem statement: Implement iterative binary search.

Underlying concepts: Binary search.

Binary search is an algorithm for locating the position of an element in a sorted list. It inspects the middle element of the sorted list: if equal to the sought value, then the position has been found; otherwise, the upper half or lower half is chosen for further searching based on whether the sought value is greater than or less than the middle element. The method reduces the number of elements needed to be checked by a factor of two each time, and finds the sought value if it exists in the list or if not determines "not present", in logarithmic time.

Code:

```
#include<iostream>
using namespace std;

int binsearch(int*, int, int, int);

int main()
{
	int a[10], x, loc, low, high;
	cout<<"Enter 10 elemets of the array:\n";
	for(int i=0; i<10;i++)
		cin>>a[i];
	cout<<"Enter the element to be searched : ";
	cin>>x;
	loc=binsearch(a, 0, 9, x);
	if(loc==-1)
		cout<<"Element not in the list\n";
	else
		cout<<"Element found at "<<loc<<endl;
	return 0;
}

int binsearch(int*a, int low, int high, int x)
{
	int mid;

	mid=(low+high)/2;

	while(a[mid]!=x)
	{
```

```
            if(a[mid]<x)
                low=mid+1;
            else
                high=mid-1;
            mid=(low+high)/2;
        }

        if(a[mid]==x)
            return mid;
        else
            return -1;
}
```

Output:

Lab Assignment#2

Objective: To learn the concepts of recursive binary search.

Problem statement: Implement recursive binary search.

Underlying concepts: Binary search, recursion.

Binary search is an algorithm for locating the position of an element in a sorted list. It inspects the middle element of the sorted list: if equal to the sought value, then the position has been found; otherwise, the upper half or lower half is chosen for further searching based on whether the sought value is greater than or less than the middle element. The method reduces the number of elements needed to be checked by a factor of two each time, and finds the sought value if it exists in the list or if not determines "not present", in logarithmic time.

Recursion is a process in which a function calls itself or calls a series of other functions that eventually calls the original function.

Code:

```
#include<iostream>
using namespace std;

int binsearch(int*, int, int, int);

int main()
{
	int a[10], x, loc, low, high;

	cout<<"Enter 10 elemets of the array:\n";
	for(int i=0; i<10;i++)
		cin>>a[i];

	cout<<"Enter the element to be searched : ";
	cin>>x;

	loc=binsearch(a, 0, 9, x);

	if(loc==-1)
		cout<<"Element not in the list\n";
	else
		cout<<"Element found at "<<loc<<endl;
	return 0;
}

int binsearch(int*a, int low, int high, int x)
```

```
{
        if(low==high)
        {
                if(x==a[low])
                        return low;
                else
                        return -1;
        }

        int mid;

        mid=(low+high)/2;

        if(a[mid]==x)
                return mid;
        else
                if(a[mid]<x)
                        return binsearch(a, mid+1, high, x);
                else
                        return binsearch(a, low, mid-1, x);
}
```

Output:

Lab Assignment#3

Objective: To learn the concepts of divide and conquer.

Problem statement: Implement divide and conquer strategy to find minimum and maximum elements.

Underlying concepts: Divide and conquer strategy, recursion.

In Divide and conquer strategy we divide a large problem into smaller sub problems so that we can solve them easily. If the divided sub problems are still large we further divide them.

Recursion is a process where a function calls itself or calls a series of functions that eventually calls the first one.

Code:

```
#include<iostream.h>
using namespace std;void

minmax(int*, int, int,int,int);
int main()
{
	int a[10], min, max, low, high;

	cout<<"Enter 10 elemets of the array:\n";
	for(int i=0; i<10;i++)
		cin>>a[i];

	minmax(a, 0, 9, min, max);

	cout<<"Maximum element: "<<max<<"Minimum element : "<<min<<endl;
	return 0;
}

void minmax(int* a, int low, int high, int min,int max)
{
	if(low==high)
		max=min=a[low];
	else
	{
		if(low==high-1)
		{
			if(a[low]<a[high])
```

```
                {
                        min=a[low]; max=a[high];
                }
                else
                {
                        min=a[high]; max=a[low];
                }
        }
        else
        {
                int mid=(low+high)/2;

                minmax(a,low, mid, min, max);

                int max1, min1;

                minmax(a, mid+1, high, max1, min1);

                if(max<max1)
                        max=max1;
                if(min>min1)
                        min=min1;
        }
    }
}
```

Output:

Lab Assignment#4

Objective: To learn the concepts of quick sort.

Problem statement: Implement quick sort method

Underlying concepts: Quick sort method.

Quick sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base cases of the recursion is lists of size zero or one, which are always sorted.

Code:

```
#include<iostream>
using namespace std;

void swap(int a, int b)
{
		int temp;
		temp = a;
		a = b;
		b = temp;
}

int partition(int*a,int low, int high);

void quicksort(int*a, int low, int high);

int main()
{
	int a[10], min, max, low, high;

	cout<<"Enter 10 elements of the array:\n";
```

```
	for(int i=0; i<10;i++)
		cin>>a[i];

	quicksort(a, 0, 9);

	cout<<"Array in sorted order:\n";

	for(i=0; i<10;i++)
		cout<<a[i]<<endl;
	return 0;
}

int partition(int*a,int low, int high)
{
	int left, right,temp;
	int pivot_item=a[low];
	left=low;
	right=high;
	while(left<right)
	{
		while(a[left]<=pivot_item)
			left++;
		while(a[right]>pivot_item)
			right--;
		if(left<right)
			swap(a[left], a[right]);
	}
	a[low]=a[right];
	a[right]=pivot_item;
	return right;
}

void quicksort(int* a, int low, int high)
{
		int pivot;

		if(high > low)
		{
			pivot=partition(a, low, high);
			quicksort(a, low, pivot-1);
			quicksort(a, pivot+1, high);
		}
}
```

Output:

Lab Assignment#5

Objective: To learn the concepts of merge sort.

Problem statement: Implement Merge sort method

Underlying concepts: Merge sort method.

Conceptually, a merge sort works as follows

1. If the list is of length 0 or 1, then it is already sorted. Otherwise:
2. Divide the unsorted list into two sub lists of about half the size.
3. Sort each sublist recursively by re-applying merge sort.
4. Merge the two sub lists back into one sorted list.

Merge sort incorporates two main ideas to improve its runtime:

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted (see the merge function below for an example implementation).

Code:

```
#include <iostream>

using namespace std;

int a[50];

void merge(int,int,int);

void merge_sort(int low,int high)
{
 int mid;
 if(low<high)
 {
  mid=(low+high)/2;
  merge_sort(low,mid);
  merge_sort(mid+1,high);
  merge(low,mid,high);
 }
}
void merge(int low,int mid,int high)
{
 int h,i,j,b[50],k;
 h=low;  i=low;  j=mid+1;
```

```
while((h<=mid)&&(j<=high))
{
 if(a[h]<=a[j])
 {
  b[i]=a[h];    h++;
 }
 else
 {
  b[i]=a[j];   j++;
 }
 i++;
}
if(h>mid)
{
 for(k=j;k<=high;k++)
 {
  b[i]=a[k];
  i++;
 }
}
else
{
 for(k=h;k<=mid;k++)
 {
  b[i]=a[k];
  i++;
 }
}
for(k=low;k<=high;k++) a[k]=b[k];
}
int main()
{
        int num,i;

        cout<<"Please Enter THE NUMBER OF ELEMENTS: "<<endl;
        cin>>num;
        cout<<endl;
        cout<<"Enter the ("<< num <<") numbers (ELEMENTS):" <<endl;
        for(i=1;i<=num;i++)
        {
          cin>>a[i] ;
        }
        merge_sort(1,num);
        cout<<endl;
        cout<<"So, the sorted list (using MERGE SORT) will be :"<<endl;
        cout<<endl<<endl;
```

```
        for(i=1;i<=num;i++)
                cout<<a[i]<<" ";
        return 0;
}
```

Output:

Lab Assignment#6

Objective: To learn the concepts of Strassen's Matrix Multiplication.
Problem statement: Implement Strassen's Matrix Multiplication

Underlying concepts: Merge sort method.

- **Given:** Two N by N matrices A and B.

  Problem: Compute $C = A \times B$

- **Brute Force**

```
for i:= 1 to N do
  for j:=1 to N do
     C[i,j] := 0;
     for k := 1 to N do
       C[i,j] := C[i,j] + A[i,k] * B[k,j]
```

- $O(N^3)$ multiplications
- **Divide and Conquer**

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ P_2 &= (A_{21} + A_{22})B_{11} \\ P_3 &= A_{11}(B_{12} - B_{22}) \\ P_4 &= A_{22}(B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{12})B_{22} \\ P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ C_{11} &= P_1 + P_4 - P_5 + P_7 \\ C_{12} &= P_3 + P_5 \\ C_{21} &= P_2 + P_4 \\ C_{22} &= P_1 + P_3 - P_2 + P_6 \end{aligned}$$

From

$$T(n) = \begin{cases} 7T(n/2) + cn & \text{if } n > 1 \\ c & \text{if } n = 1 \end{cases}$$

$T(n) = O(n^{\log 7}) = O(n^{2.81})$.

```
#include<iostream.h>
#include<conio.h>
const int s=4;
void matmul(int *A, int *B, int *R, int n);
void main()
{
clrscr();
int a[s][s], b[s][s], c[s][s], n=s, i,j;

for(i=0;i<n;i++)
        for(j=0;j<n;j++)
                a[i][j]=i+j;
for(i=0;i<n;i++)
        for(j=0;j<n;j++)
                b[i][j]=i-j;


matmul(*a,*b,*c,n);

for(i=0;i<n;i++)
{
        for(j=0;j<n;j++)
                cout<<a[i][j]<<"\t";
        cout<<endl;
}
cout<<endl<<endl;

for(i=0;i<n;i++)
{
        for(j=0;j<n;j++)
                cout<<b[i][j]<<"\t";
        cout<<endl;
}
cout<<endl<<endl;

for(i=0;i<n;i++)
{
        for(j=0;j<s;j++)
                cout<<c[i][j]<<"\t";
        cout<<endl;
}
}
void matmul(int *A, int *B, int *R, int n)
{
 if (n = = 1) {
        (*R) += (*A) * (*B);
```

```
 } else {
        matmul(A, B, R, n/4);
        matmul(A, B+(n/4), R+(n/4), n/4);
        matmul(A+2*(n/4), B, R+2*(n/4), n/4);
        matmul(A+2*(n/4), B+(n/4), R+3*(n/4), n/4);
        matmul(A+(n/4), B+2*(n/4), R, n/4);
        matmul(A+(n/4), B+3*(n/4), R+(n/4), n/4);
        matmul(A+3*(n/4), B+2*(n/4), R+2*(n/4), n/4);
        matmul(A+3*(n/4), B+3*(n/4), R+3*(n/4), n/4);
} }
```

Analysis:

Conclusion:

Lab Assignment#7

Objective: To learn the concepts of Floyd-Warshall algorithm using dynamic programming.

Problem statement: Implement Floyd-Warshall algorithm.

Underlying concepts: Floyd-Warshall algorithm, Dynamic programming.

The structure of a shortest path

In the Floyd-Warshall algorithm, we use a different characterization of the structure of a shortest path than we used in the matrix-multiplication-based all-pairs algorithms. The algorithm considers the "intermediate" vertices of a shortest path, where an **intermediate** vertex of a simple path $p = (v_1, v_2. \ . \ . \ v_k)$ is any vertex of p other than v_1 or v_k , that is, any vertex in the set $\{v_2, v_3, \ldots, v_{k-1}\}$.

The Floyd-Warshall algorithm is based on the following observation. Under our assumption that the vertices of G are $V = \{1, 2, \ldots, n\}$, let us consider a subset $\{1, 2, \ldots, k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them. (Path p is simple.) The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k - 1\}$. The relationship depends on whether or not k is an intermediate vertex of path p.

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1, 2, \ldots, k - 1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k - 1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.
- If k is an intermediate vertex of path p, then we break p down into $i \xrightarrow{p1} k \xrightarrow{p2} j$ as shown in Figure. p1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. Because vertex k is not an intermediate vertex of path p1, we see that p1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k - 1\}$. Similarly, p2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k - 1\}$.

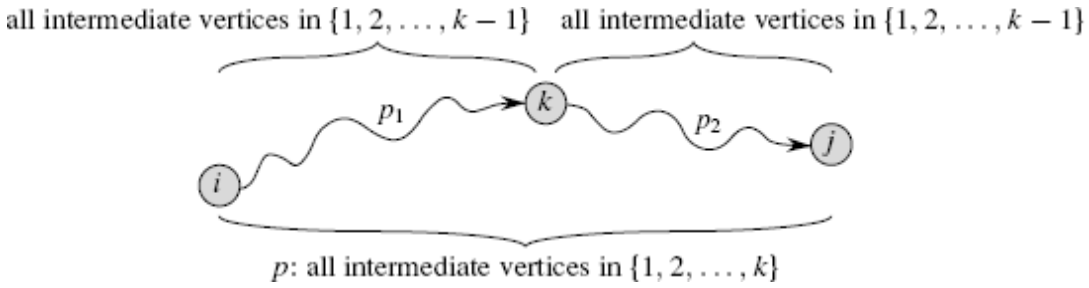


Figure Path *p* is a shortest path from vertex *i* to vertex *j*, and *k* is the highest-numbered intermediate vertex of *p*. Path *p*1, the portion of path *p* from vertex *i* to vertex *k*, has all

intermediate vertices in the set {1, 2, . . . , $k-1$}. The same holds for path $p2$ from vertex k to vertex j .

A recursive solution to the all-pairs shortest-paths problem

Based on the above observations, we define a recursive formulation of shortest-path estimates that is different from the one in Section 25.1. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, . . . , k}. When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. A recursive definition following the above discussion is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 . \end{cases}$$

Because for any path, all intermediate vertices are in the set {1, 2, . . . , n}, the matrix $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

Computing the shortest-path weights bottom up

Based on above recurrence, the following bottom-up procedure can be used to compute the values $d_{ij}^{(k)}$ in order of increasing values of k. Its input is an $n \times n$ matrix W defined as

$$w_{ij} = \begin{cases} 0 & \text{if } i = j , \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E , \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E . \end{cases}$$

The procedure returns the matrix $D^{(n)}$ of shortest-path weights.

FLOYD-WARSHALL*(W)*
1 $n \leftarrow rows[W]$
2 $D^{(0)} \leftarrow W$
3 **for** $k \leftarrow 1$ **to** n
4 **for** $i \leftarrow 1$ **to** n
5 **for** $j \leftarrow 1$ **to** n
6 $d_{ij}^{(k)} \leftarrow \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$
7 **return** *D(n)*

Code:

```
#include<iostream.h>
#include<conio.h>
void floyd(int m[10][10], int n);
int min(int, int);
void main()
{
        int i,j,n, w[10][10];
        cout<<"Enter the adjacency matrix for the graph:\n\n";
```

```
        cout<<"Enter the number of vertices: ";
        cin>>n;
        cout<<" Enter the weight of edges[Enter 999 for infinity]\n\n:";
        for(i=0; i<n; i++)
                for(j=0;j<n;j++)
                {
                        cout<<"Enter the cost of ("<<i<<" , "<<j<<" ):";
                        cin>>w[i][j];
                }
        floyd(w,n);
}
int min(int a, int b)
{
        if(a<b)
                return a;
        else
                return b;
}
void floyd(int w[10][10], int n)
{
        int d[5][10][10],i, j, k;
        for(i=0;i<n;i++)
                for(j=0; j<n; j++)
                        d[0][i][j]=w[i][j];
        for(k=1;k<n;k++)
                for(i=0;i<n;i++)
                        for(j=0; j<n; j++)
                                d[k][i][j]=min(d[k-1][i][j],
                                                        d[k-1][i][k]+d[k-
1][k][j]);
        for(k=0;k<n;k++)
        {
                for(i=0;i<n;i++)
                {
                        for(j=0; j<n; j++)
                                cout<<d[k][i][j]<<"\t";
                        cout<<endl;
                }
                cout<<endl;
        }
}
```

Output:

Enter the weight of edges[Enter 999 for infinity]:

```
Enter the cost of (0 , 0 ):0
Enter the cost of (0 , 1 ):6
Enter the cost of (0 , 2 ):5
Enter the cost of (1 , 0 ):2
Enter the cost of (1 , 1 ):0
Enter the cost of (1 , 2 ):7
Enter the cost of (2 , 0 ):3
Enter the cost of (2 , 1 ):1
Enter the cost of (2 , 2 ):0
0       6       5
2       0       7
3       1       0

0       6       5
2       0       7
3       1       0

0       6       5
2       0       7
3       1       0
```

Lab Assignment#8

Objective: To learn the concepts of Kruskal's algorithm

Problem statement: Implement Kruskal's algorithm for minimum spanning trees

Underlying concepts: Minimum spanning tree. Kruskal' algorithm.

Given a connected, undirected graph, a spanning tree of that graph is a sub graph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components.

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

It works as follows:

- create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
- create a set *S* containing all the edges in the graph
- while *S* is nonempty and F is not yet spanning
    - remove an edge with minimum weight from *S*
    - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
    - otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

```
1  function Kruskal(G)
 2   Define an elementary cluster C(v) ← {v}.
 3   Initialize a priority queue Q to contain all edges in G, using the weights as keys.
 4   Define a forest T ← Ø       //T will ultimately contain the edges of the MST
 5    // n is total number of vertices
 6   while T has fewer than n-1 edges do
 7     // edge u,v is the minimum weighted route from/to v
 8     (u,v) ← Q.removeMin()
```

9 // prevent cycles in T. add u,v only if T does not already contain a path between u and v.
10 // the vertices has been added to the tree.
11 Let $C(v)$ be the cluster containing v, and let $C(u)$ be the cluster containing u.
13 **if** $C(v) \neq C(u)$ **then**
14 Add edge (v,u) to T.
15 Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$.
16 **return** tree T

Code:

```
#include<iostream.h>
#define INFINITY 999
typedef struct Graph
{
 int v1;
 int v2;
 int cost;
}GR;
GR G[20];
int tot_edges,tot_nodes;
void create();
void spanning_tree();
int Minimum(int);

void main()
{
 cout<<"\n\t Graph Creation by adjacency matrix ";
 create();
 spanning_tree();
}
void create()
{

 int k;
 cout<<"\n Enter Total number of nodes: ";
 cin>>tot_nodes;
 cout<<"\n Enter Total number of edges: ";
 cin>>tot_edges;
 for(k=0;k<tot_edges;k++)
 {
          cout<<" Enter Edge in (V1 V2)form ";
          cin>>G[k].v1>>G[k].v2;
          cout<<2
          "\n Enter Corresponding Cost ";
          cin>>G[k].cost;
```

```
 }
}
void spanning_tree()
{
 int count,k,v1,v2,i,j,tree[10][10],pos,parent[10];
 int sum;
 int Find(int v2,int parent[]);
 void Union(int i,int j,int parent[]);
 count=0;
 k=0;
 sum=0;
 for(i=0;i<tot_nodes;i++)
                    parent[i]=i;
 while(count!=tot_nodes-1)
 {
                pos=Minimum(tot_edges);//finding the minimum cost edge
                if(pos==-1)//Perhaps no node in the graph
                        break;
                v1=G[pos].v1;
                v2=G[pos].v2;
                i=Find(v1,parent);
                j=Find(v2,parent);
                if(i!=j)
                {
                        tree[k][0]=v1;//storing the minimum edge in array tree[]
                        tree[k][1]=v2;
                        k++;
                        count++;
                        sum+=G[pos].cost;//accumulating the total cost of MST
                        Union(i,j,parent);
                }
                G[pos].cost=INFINITY;
        }
                if(count==tot_nodes-1)
                {
                 cout<<"\n Spanning tree is...";
                 cin>>"\n----------------\n";
                 for(i=0;i<tot_nodes-1;i++)
                 {
                        cout<<"["<<tree[i][0];
                        cout<<" - ";
                        cout<<"%d"<<tree[i][1];
                        cout<<"]";
                 }
                 cout<<"\n----------------";
                 cout<<"\nCost of Spanning Tree is = %d"<<sum;
```

```
                }
                else
                 {
                        cout<<"There is no Spanning Tree";
                 }
}
int Minimum(int n)
{
 int i,small,pos;
 small=INFINITY;
 pos=-1;
 for(i=0;i<n;i++)
 {
                if(G[i].cost<small)
                {
                        small=G[i].cost;
                        pos=i;
                }
 }
 return pos;
}
int Find(int v2,int parent[])
{
  while(parent[v2]!=v2)
{
                v2=parent[v2];
        }
        return v2;
}
void Union(int i,int j,int parent[])
{
 if(i<j)
                 parent[j]=i;
 else
                 parent[i]=j;
}
```

Lab Assignment#9

Objective: To learn the concepts of Prim's algorithm

Problem statement: Implement Prim's algorithm for minimum spanning trees

Underlying concepts: Minimum spanning tree. Prim's algorithm.

The minimum spanning tree of a planar graph. Each edge is labeled with its weight, which here is roughly proportional to its length.

Given a connected, undirected graph, a spanning tree of that graph is a sub graph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components.

Prim's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Prim's algorithm is an example of a greedy algorithm

The algorithm continuously increases the size of a tree starting with a single vertex until it spans all the vertices.

- Input: A connected weighted graph with vertices V and edges E.
- Initialize: $V_{new} = \{x\}$, where x is an arbitrary node (starting point) from V, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
    - Choose edge (u,v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, choose arbitrarily but consistently)
    - Add v to V_{new}, add (u, v) to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree

Code:

```
# include<iostream.h>
# include<conio.h>
```

```
# define SIZE 20
# define INFINITY 32767



void Prim(int G[][SIZE], int nodes)
{
        int select[SIZE], i, j, k;
        int min_dist, v1, v2,total=0;

        for (i=0 ; i<nodes ; i++)  // Initialize the selected vertices list
                        select[i] = 0;

        cout<<"\n\n The Minimal Spanning Tree Is :\n";
        select[0] = 1;
        for (k=1 ; k<nodes ; k++)
        {
                        min_dist = INFINITY;
                        for (i=0 ; i<nodes ; i++) // Select an edge such that one vertex is
                        {                       // selected and other is not and the edge
        for (j=0 ; j<nodes ; j++) // has the least weight.
        {
                if (G[i][j] && ((select[i] && !select[j]) || (!select[i] && select[j])))
                        {
                                if (G[i][j] < min_dist)//obtained edge with minimum wt
                                {
                                        min_dist = G[i][j];
                                        v1 = i;
                                        v2 = j;   //picking up those vertices
                                }
                        }
                }
        }
        cout<<"\n Edge "<v1<<v2<<" and weight = "<<min_dist;
        select[v1] = select[v2] = 1;
        total =total+min_dist;
        }
       cout<<"\n\n\t Total Path Length Is = "<<total;
}

void main()
{
        int G[SIZE][SIZE], nodes;
        int v1, v2, length, i, j, n;

        clrscr();
```

```
cout<<"\n\t Prim'S Algorithm\n"

cout<<"\n Enter Number of Nodes in The Graph  ";
cin>>nodes;
cout<<"\n Enter Number of Edges in The Graph  ";
cin>>n;

for (i=0 ; i<nodes ; i++)       // Initialize the graph
        for (j=0 ; j<nodes ; j++)
                G[i][j] = 0;
 //entering weighted graph
cout<<"\n Enter edges and weights \n";
for (i=0 ; i<n; i++)
{
                cout<<"\n Enter Edge by V1 and V2 :";
                cin>>v1>>v2;
                cout<<"\n Enter corresponding weight :";
                cin>>length;
                G[v1][v2] = G[v2][v1] = length;
}
getch();
cout<<"\n\t";
clrscr();
Prim(G,nodes);
getch();
                                        }
```

Lab Assignment#10

Objective: To learn the concept of Dijkstra's algorithm

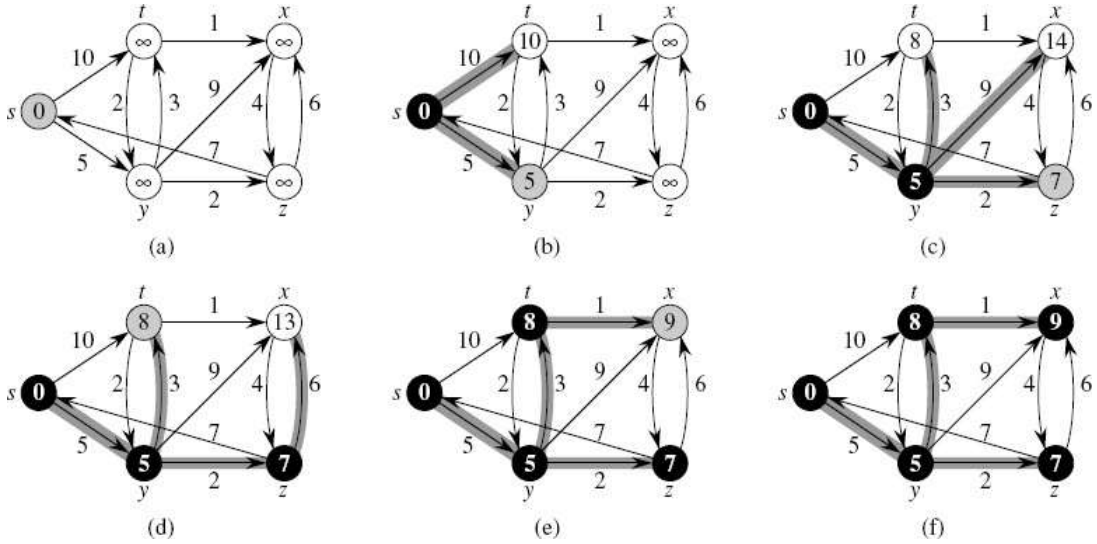
Problem statement: Implement single sources shortest path algorithm.

Underlying concepts: Dijkstra's algorithm.

Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. Therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.
Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.
DIJKSTRA*(G,w, s)*
1 INITIALIZE-SINGLE-SOURCE*(G, s)*
2 $S \leftarrow \emptyset$ 3 $Q \leftarrow V[G]$
4 **while** Q != $\emptyset$ **do** $u \leftarrow$ EXTRACT-MIN*(Q)*
6 $S \leftarrow S \cup \{u\}$ 7 **for** each vertex $v \in Adj[u]$
8 **do** RELAX*(u, v,w)*

Dijkstra's algorithm relaxes edges as shown



Code:

```
#include<iostream.h>
#include<conio.h>
#define infinity 999
int path[10];
void main()
{
 int tot_nodes,i,j,cost[10][10],dist[10],s[10];
 void create(int tot_nodes,int cost[][10]);
 void Dijkstra(int tot_nodes,int cost[][10],int i,int dist[10]);
 void display(int i,int j,int dist[10]);
 clrscr();
 cout<<"\n\t\t Creation of graph ";
 cout<<"\n Enter total number of nodes ";
 cin>>tot_nodes;
 create(tot_nodes,cost);
 for(i=0;i<tot_nodes;i++)
 {
                cout<<"\n\t\t\t Press any key to continue...";
                cout<<"\n\t\t When Source =”<<i<<endl;
                for(j=0;j<tot_nodes;j++)
                {
                 Dijkstra(tot_nodes,cost,i,dist);
                 if(dist[j]==infinity)
                        cout<<"\n There is no path to "<<j<<endl;
                 else
                 {

                        display(i,j,dist);
                 }
                }
        }
}
void create(int tot_nodes,int cost[][10])
{
 int i,j,val,tot_edges,count=0;
 for(i=0;i<tot_nodes;i++)
 {
                for(j=0;j<tot_nodes;j++)
                {
                 if(i==j)
                        cost[i][j]=0;//diagonal elements are 0
                 else
```

```
                    cost[i][j]=infinity;
                }
 }
 cout<<"\n Total number of edges ";
 cin>>&tot_edges;
 while(count<tot_edges)
                {
                 cout<<"\n Enter Vi and Vj";
                 cin>>i>>j;
                 cout<<"\n Enter the cost along this edge ";
                 cin>>val;
                 cost[j][i]=val;
                 cost[i][j]=val;
                 count++;
                }
}
```

Lab Assignment#10

Objective: To learn the concept of Dijkstra's algorithm

Problem statement: Implement single sources shortest path algorithm.

Underlying concepts: Dijkstra's algorithm.

Dijkstra's algorithm

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first

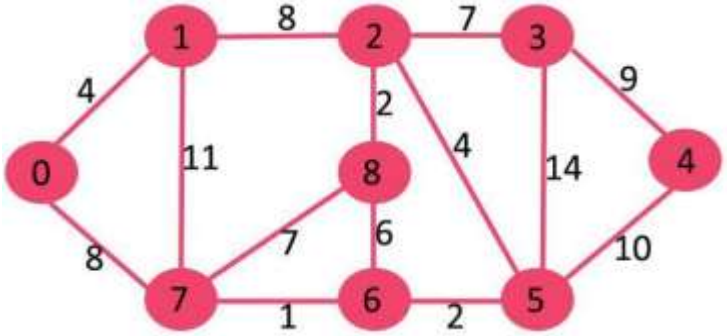
3) While *sptSet* doesn't include all vertices

a) Pick a vertex u which is not there in *sptSet*and has minimum distance value.

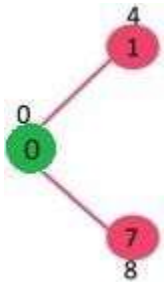
b) Include u to *sptSet*.

c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

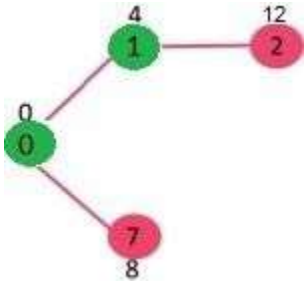
Let us understand with the following example:



The set *sptSet*is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.

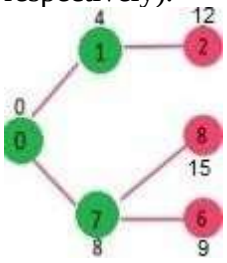


Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.

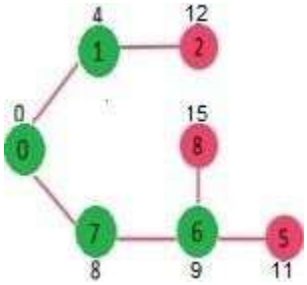


Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9

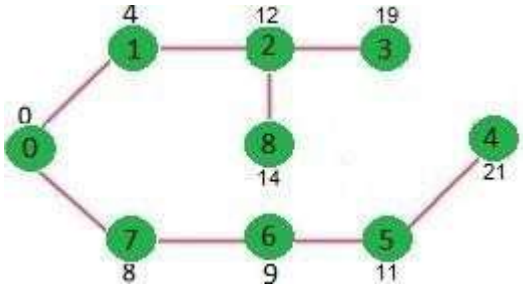
respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



How to implement the above algorithm?

We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store shortest distance values of all vertices.

// Program for Dijkstra's single source shortest path algorithm.The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
#define V 9

```
// A utility function to find the vertex with minimum distance value, //from the set of
vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
  // Initialize min value
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++)
    if (sptSet[v] == false && dist[v] < min)
       min = dist[v], min_index = v;

  return min_index;
}

// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
  printf("Vertex   Distance from Source\n");
  for (int i = 0; i < V; i++)
     printf("%d \t\t %d\n", i, dist[i]);
}

// Funtion that implements Dijkstra's single source shortest path //algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V];     // The output array.  dist[i] will hold the shortest
                     // distance from src to i

    bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest
                    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
       dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V-1; count++)
    {
      // Pick the minimum distance vertex from the set of vertices //not
      // yet processed. u is always equal to src in first iteration.
      int u = minDistance(dist, sptSet);
```

```
        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the picked //vertex.
        for (int v = 0; v < V; v++)

          // Update dist[v] only if is not in sptSet, there is an edge //from
          // u to v, and total weight of path from src to  v through u //is
          // smaller than current value of dist[v]
          if (!sptSet[v] && graph[u][v] && dist[u] + graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
      }

      // print the constructed distance array
      printSolution(dist, V);
}

// driver program to test above function
int main()
{
   /* Let us create the example graph discussed above */
   int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                      {0, 0, 8, 0, 0, 0, 0, 11, 0},
                      {0, 8, 0, 7, 0, 4, 0, 0, 2},
                      {0, 0, 7, 0, 9, 14, 0, 0, 0},
                      {0, 0, 0, 9, 0, 10, 0, 0, 0},
                      {0, 0, 4, 0, 10, 0, 2, 0, 0},
                      {0, 0, 0, 14, 0, 2, 0, 1, 6},
                      {8, 11, 0, 0, 0, 0, 1, 0, 7},
                      {0, 0, 2, 0, 0, 0, 6, 7, 0}
                     };

    dijkstra(graph, 0);

    return 0;
}
```

Output:

```
Vertex   Distance from Source
0              0
1              4
2              12
3              19
4              21
```

5	11
6	9
7	8
8	14

Implementation 2

```
#include<iostream>
#include<conio.h>
#include<stdio.h>
using namespace std;
int shortest(int ,int);
int cost[10][10],dist[20],i,j,n,k,m,S[20],v,totcost,path[20],p;
main()
{
        int c;
        cout <<"enter no of vertices";
        cin >> n;
        cout <<"enter no of edges";
        cin >>m;
        cout <<"\nenter\nEDGE Cost\n";
        for(k=1;k<=m;k++)
        {
          cin >> i >> j >>c;
          cost[i][j]=c;
        }
        for(i=1;i<=n;i++)
          for(j=1;j<=n;j++)
                  if(cost[i][j]==0)
                          cost[i][j]=31999;
        cout <<"enter initial vertex";
        cin >>v;
        cout << v<<"\n";
        shortest(v,n);
 }

int shortest(int v,int n)
{
        int min;
        for(i=1;i<=n;i++)
        {
          S[i]=0;
          dist[i]=cost[v][i];
        }
        path[++p]=v;
        S[v]=1;
        dist[v]=0;
        for(i=2;i<=n-1;i++)
        {
          k=-1;
          min=31999;
```

```
   for(j=1;j<=n;j++)
   {
                if(dist[j]<min && S[j]!=1)
                {
                        min=dist[j];
                        k=j;
                }
}
if(cost[v][k]<=dist[k])
   p=1;
path[++p]=k;
for(j=1;j<=p;j++)
   cout<<path[j];
   cout <<"\n";
   //cout <<k;
   S[k]=1;
for(j=1;j<=n;j++)
if(cost[k][j]!=31999 && dist[j]>=dist[k]+cost[k][j] && S[j]!=1)
 dist[j]=dist[k]+cost[k][j];
}
```