A

Practical Training Report

on

Web Based Software for Computing Shortest Path Algorithm and Minimum Spanning Tree for A Weighted Graph

Submitted in partial fulfillment for the award of degree of

BACHELOR OF TECHNOLOGY

In

Computer Science & Engineering

Submitted to:
Ms. Pooja Sharma

Submitted by:
Abhilash Mathur
(17EGJCS003)

Department of Computer Science & Engineering
GLOBAL INSTITUTE OF TECHNOLOGY
JAIPUR (RAJASTHAN)-302022
SESSION: 2019-20

`

CERTIFICATE

DEFENCE LABORATORY, JODHPUR
(DRDO)

CERTIFICATE

This is to certify that Mr Abhilash Mathur student of Bachelor of Technology (CSE), IInd Year, Global Institute of Technology Jaipur, has undergone practical training on "Web Based Software for Computing Shortest Path Algorithm and Minimum Spanning Tree for a Weighted Graph" from 28th May 2019 to 12th July 2019 at this Laboratory.

DLJ/HRD/R/2044/V/2019/99
Dated : 22 July 2019

DKTripathi

(D K Tripathi, Sc-F)
Training Guide

DEFENCE LABORATORY MINISTRY OF DEFENCE JODHPUR

2/7/19

(Dr Prashant Vasistha, Sc-F)
Joint Director, HRD

ACKNOWLEDGEMENT

With great pleasure, respect and deep gratitude, I would like to express my sincere thanks to Sh. Ravindra Kumar, OS & Director, Defense Laboratory, Jodhpur for providing me the opportunity to accomplish training.

I am also extremely indebted to Sh. G.S. Gehlot, Training Officer for his obliging allowance and efforts in organizing the whole training program.

I am especially grateful to my program guide, Sh. D.K Tripathi, Scientist 'F' for his benevolent help and support throughout the training program, without whom things could not have proceeded as smoothly as they did.

Last but not the least, I would like to extend my thanks to the very cooperative **Nuclear Radiation Management & Application (NRMA)** Division, both technical and non-technical for their help during the training period.

I am also indebted to my parents for their valuable and constant encouragement.

`

ABSTRACT

A data structure is a special way of organizing and storing data in a computer so that it can be used efficiently. Array, linked list, Stack, Queue, Tree, Graph etc. are all data structures that stores the data in a special way so that we can access and use the data efficiently**.** Each of these mentioned data structures has a different special way of organizing data so we choose the data structure based on the requirement.

The shortest path problem is about finding a path between vertices in a graph such that the total sum of the edges weights is minimum.

In my project, I have implemented two algorithms, that is, Dijkstra's algorithm and Prim's algorithm.

Dijkstra's algorithm is used to tackle the shortest path problem while Prim's algorithm is used to find the minimum cost of a spanning tree.

It is hoped that this project will be used to calculate shortest path between two points and calculate the shortest path of traversing a minimum spanning tree.

`

LIST OF CONTENTS

`

LIST OF TABLE

`

LIST OF FIGURES

`

`

CHAPTER 1: INTRODUCTION

1.1 DATA STRUCTURE:

In computer science, a data structure is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by a pointer—a bit string, representing a memory address, that can be itself stored in memory and manipulated by the program. Thus, the array and record data structures are based on computing the addresses of data items with arithmetic operations, while the linked data structures are based on storing addresses of data items within the structure itself.

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure. The efficiency of a data structure cannot be analysed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).

Anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc. all are data structures. They are known as Primitive Data Structures.

1.2 SPECIFIC PRIMITIVE DATA TYPES:

1.2.1 Integer:

An integer data type that represents some range of mathematical integers. Integers may be either signed (allowing negative values) or unsigned (non-negative integers only). Common ranges are:

Literals for integers can be written as regular Arabic numerals, consisting of a sequence of digits and with negation indicated by a minus sign before the value. However, most

`

programming languages disallow use of commas for or spaces digit grouping. Examples of integer literals are:

- 42
- 10000
- -233000

There are several alternate methods for writing integer literals in many programming languages:

- Most programming languages, especially those influenced by C, prefix an integer literal with 0X or 0x to represent a hexadecimal value, e.g. 0xDEADBEEF. Other languages may use a different notation, e.g. some assembly languages append an H or h to the end of a hexadecimal value.
- Perl, Ruby, Java, Julia, D, Rust and Python (starting from version 3.6) allow embedded underscores for clarity, e.g. 10_000_000, and fixed-form Fortran ignores embedded spaces in integer literals.
- In C and C++, a leading zero indicates an octal value, e.g. 0755. This was primarily intended to be used with Unix modes; however, it has been criticized because normal integers may also lead with zero. As such, Python, Ruby, Haskell, and OCaml prefix octal values with 0O or 0o, following the layout used by hexadecimal values.
- Several languages, including Java, C#, Scala, Python, Ruby, and OCaml, can represent binary values by prefixing a number with 0B or 0b.

1.2.2 Boolean:

A boolean type, typically denoted "bool" or "boolean", is typically a *logical type* that can be either "true" or "false". Although only one bit is necessary to accommodate the value set "true" and "false", programming languages typically implement boolean types as one or more bytes.

Many languages (e.g. Java, Pascal and Ada) implement booleans adhering to the concept of *boolean* as a distinct logical type. Languages, though, may implicitly convert booleans to *numeric types* at times to give extended semantics to booleans and boolean expressions

`

or to achieve backwards compatibility with earlier versions of the language. For example, ANSI C and its former standards did not have a dedicated boolean type. Instead, numeric values of zero are interpreted as "false", and any other value is interpreted as "true". C99 adds a distinct boolean type that can be included with stdbool.h, and C++ supports bool as a built-in type and "true" and "false" as reserved words.

1.2.3 Floating point numbers:

A floating-point number represents a limited-precision rational number that may have a fractional part. These numbers are stored internally in a format equivalent to scientific notation, typically in binary but sometimes in decimal. Because floating-point numbers have limited precision, only a subset of real or rational numbers are exactly representable; other numbers can be represented only approximately.

Many languages have both a single precision (often called "float") and a double precision type.

Literals for floating point numbers include a decimal point, and typically use e or E to denote scientific notation. Examples of floating-point literals are:

- 20.0005
- 99.9
- -5000.12
- 6.02e23

Some languages (e.g., Fortran, Python, D) also have a complex number type comprising two floating-point numbers: a real part and an imaginary part.

1.2.4 Fixed point numbers:

A fixed-point number represents a limited-precision rational number that may have a fractional part. These numbers are stored internally in a scaled-integer form, typically in binary but sometimes in decimal. Because fixed-point numbers have limited precision, only a subset of real or rational numbers are exactly representable; other numbers can be represented only approximately. Fixed-point numbers also tend to have a more limited

`

range of values than floating point, and so the programmer must be careful to avoid overflow in intermediate calculations as well as the final result.

1.2.5 Characters and strings:

A character type (typically called "char") may contain a single letter, digit, punctuation mark, symbol, formatting code, control code, or some other specialized code (e.g., a byte order mark). In C, char is defined as the smallest addressable unit of memory. On most systems, this is 8 bits; Several standards, such as POSIX, require it to be this size. Some languages have two or more character types, for example a single-byte type for ASCII characters and a multi-byte type for Unicode characters. The term "character type" is normally used even for types whose values more precisely represent code units, for example a UTF-16 code unit as in Java and JavaScript.

Characters may be combined into strings. The string data can include numbers and other numerical symbols but will be treated as text.

Strings are implemented in various ways, depending on the programming language. The simplest way to implement strings is to create them as an array of characters, followed by a delimiting character used to signal the end of the string, usually NUL. These are referred to as null-terminated strings, and are usually found in languages with a low amount of hardware abstraction, such as C and Assembly. While easy to implement, null terminated strings have been criticized for causing buffer overflows. Most high-level scripting languages, such as Python, Ruby, and many dialects of BASIC, have no separate character type; strings with a length of one are normally used to represent single characters. Some languages, such as C++ and Java, have the capability to use null-terminated strings (usually for backwards-compatibility measures), but additionally provide their own class for string handling in the standard library.

There is also a difference on whether or not strings are mutable or immutable in a language. Mutable strings may be altered after their creation, whereas immutable strings maintain a constant size and content. In the latter, the only way to alter strings are to create new ones. There are both advantages and disadvantages to each approach: although immutable strings are much less flexible, they are simpler and completely thread-safe. Some examples of languages that use mutable strings include C++, Perl, and Ruby, whereas languages that do

`

not include JavaScript, Lua, Python and Go. A few languages, such as Objective-C, provide different types for mutable and immutable strings.

Literals for characters and strings are usually surrounded by quotation marks: sometimes, single quotes (') are used for characters and double quotes (") are used for strings. Python accepts either variant for its string notation.

Examples of character literals in C syntax are:

- 'A'
- '4'
- '$'
- '\t' (tab character)

Examples of string literals in C syntax are:

- "A"
- "Hello World"

1.2.6 Numeric data type ranges:

Each numeric data type has its maximum and minimum value known as the range. Attempting to store a number outside the range may lead to compiler/runtime errors, or to incorrect calculations (due to truncation) depending on the language being used.

The range of a variable is based on the number of bytes used to save the value, and an integer data type is usually able to store 2^n values (where n is the number of bits that contribute to the value). For other data types (e.g. floating-point values) the range is more complicated and will vary depending on the method used to store it. There are also some types that do not use entire bytes, e.g. a boolean that requires a single bit, and represents a binary value (although in practice a byte is often used, with the remaining 7 bits being redundant). Some programming languages (such as Ada and Pascal) also allow the opposite direction, that is, the programmer defines the range and precision needed to solve a given problem and the compiler chooses the most appropriate integer or floating-point type automatically.

`

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of Abstract Data Structure are:

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required.

1.3 COMPARISON OF DIFFERENT ABSTRACT DATA STRUCTURES:

Data Structure	Strengths	Weaknesses
arrays	Inserting and deleting elements, iterating through the collection	Sorting and searching, Inserting and deleting - especially if you are inserting and deleting at the beginning or the end of the array
linked list	direct indexing, Easy to create and use	direct access, searching and sorting
stack and queue	designed for LIFO / FIFO	direct access, searching and sorting
binary tree	speed of insertion and deletion, speed of access, maintaining sorted order	some overhead

Table 1.1 Comparison of Different Abstract Data Structures

`

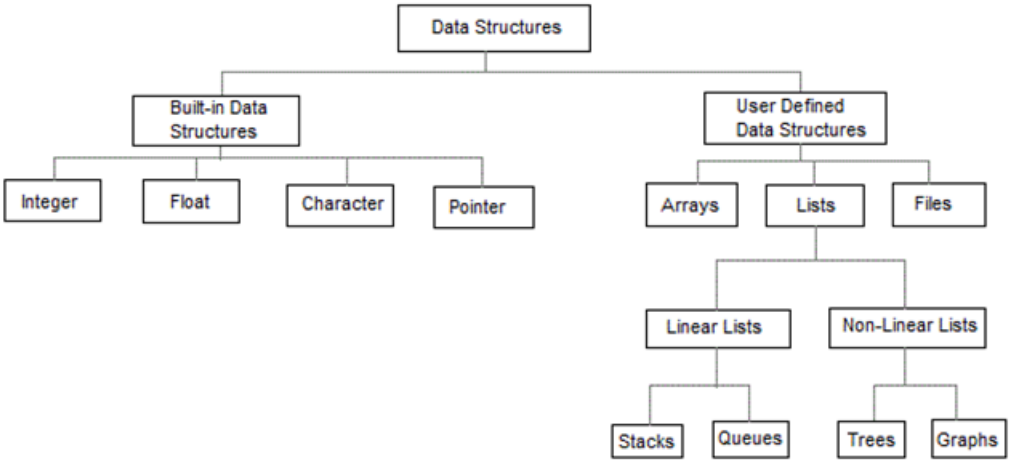


Fig 1.1 Introduction to Data Structures

1.4 GRAPHS:

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

A Graph consists of a finite set of vertices (or nodes) and set of edges which connect a pair of nodes. Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Each node is a structure and contains information like person id, name, gender, locale etc.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −

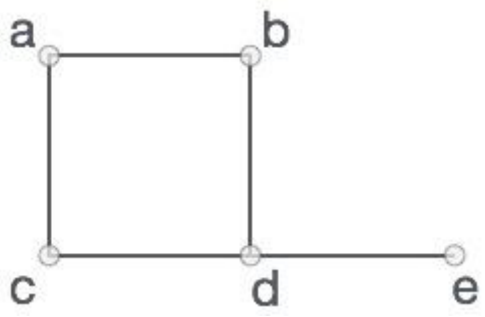


Fig 1.2 Introduction to Graph

In the above graph,

V = {a, b, c, d, e}

`

E = {ab, ac, bd, cd, de}

1.4.1 Types of Graph:

- **Finite Graphs:**

  A graph is said to be finite if it has finite number of vertices and finite number of edges.

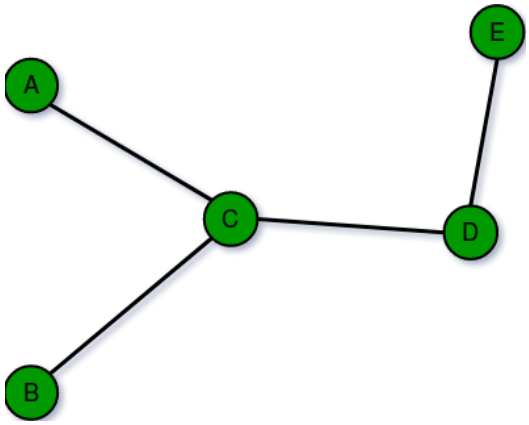


Fig 1.3 Finite Graph

- **Infinite Graph:**

  A graph is said to be infinite if it has infinite number of vertices as well as infinite number of edges.

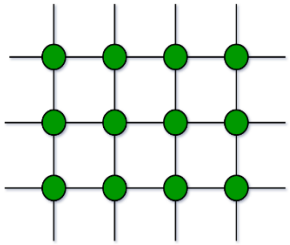


Fig 1.4 Infinite Graph

- **Trivial Graph:**

  A graph is said to be trivial if a finite graph contains only one vertex and no edge.

`

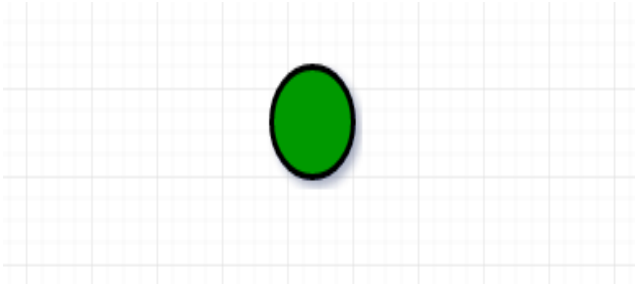


Fig 1.5 Trivial Graph

- **Simple Graph:**

A simple graph is a graph which does not contains more than one edge between the pair of vertices. A simple railway tracks connecting different cities is an example of simple graph.

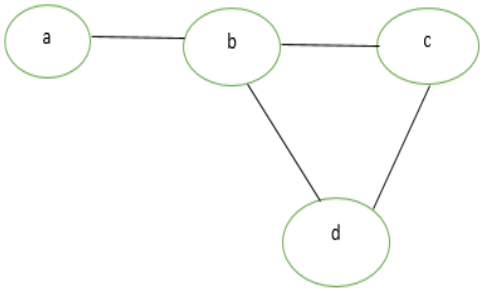


Fig 1.6 Simple Graph

- **Multi Graph:**

Any graph which contain some parallel edges but doesn't contain any self-loop is called multi graph. For example, A Road Map.

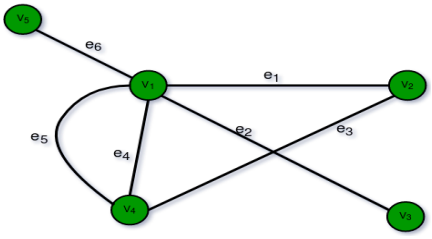


Fig 1.7 Multi Graph

- **Parallel Edges:** If two vertices are connected with more than one edge than such edges are called parallel edges that is many roots but one destination.

- **Loop:** An edge of a graph which join a vertex to itself is called loop or a self-loop.

- ## Null Graph:

A graph of order n and size zero that is a graph which contain n number of vertices but do not contain any edge.

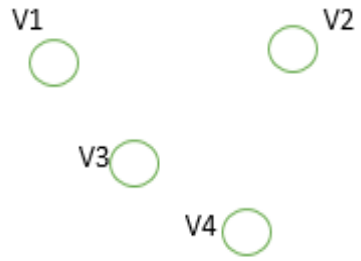


Fig 1.8 Null Graph

- ## Complete Graph:

A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attach with n-1 edges. A complete graph is also called Full Graph.

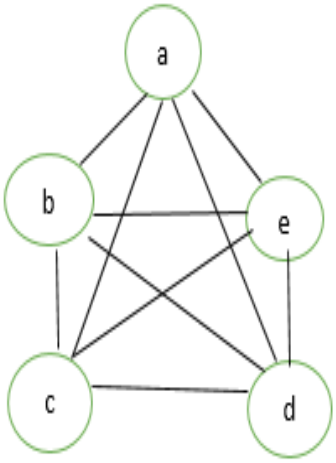


Fig 1.9 Complete Graph

`

- **Pseudo Graph:**

A graph G with a self-loop and some multiple edges is called pseudo graph.

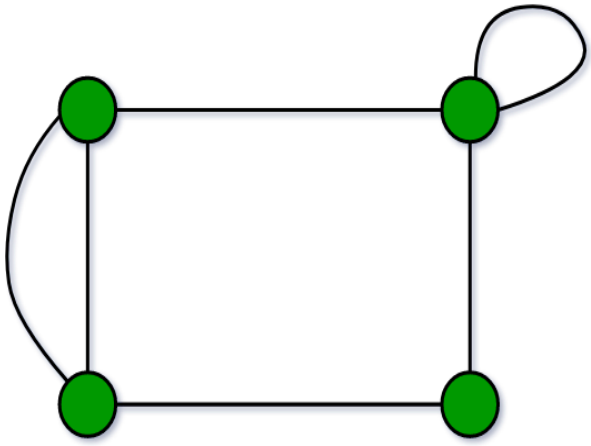


Fig 1.10 Pseudo Graph

- **Regular Graph:**

A simple graph is said to be regular if all vertices of a graph G are of equal degree. All complete graphs are regular but vice versa is not possible.

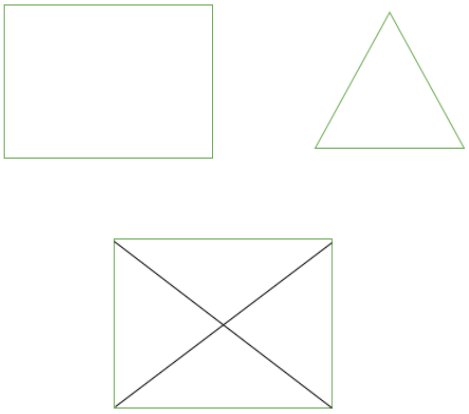


Fig 1.11 Regular Graph

- **Bipartite Graph:**

A graph G = (V, E) is said to be bipartite graph if its vertex set V(G) can be partitioned into two non-empty disjoint subsets. V1(G) and V2(G) in such a way

`

that each edge e of E(G) has its one end in V1(G) and other end in V2(G). The partition V1 U V2 = V is called Bipartite of G. Here in the figure:

V1(G)={V5, V4, V3}
V2(G)={V1, V2}

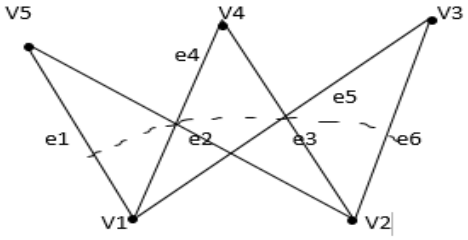


Fig 1.12 Bipartite Graph

- **Labelled Graph:**

If the vertices and edges of a graph are labelled with name, data or weight then it is called labelled graph. It is also called Weighted Graph.

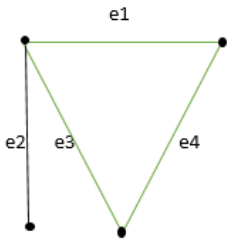


Fig 1.13 Labelled Graph

- **Digraph Graph:**

`

A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) is called Digraph. It is also called *Directed Graph*. Ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj. Here in the figure:

e1 = (V1, V2)

e2 = (V2, V3)

e4 = (V2, V4)

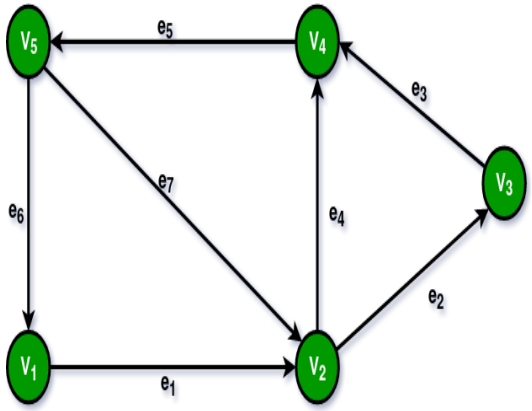


Fig 1.14 Digraph

1.5 TREES:

Tree is also a non-linear data structure which represents the nodes connected by edges.

A tree data structure can be defined recursively as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root. It is a nonlinear data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or one or more subtrees.

`

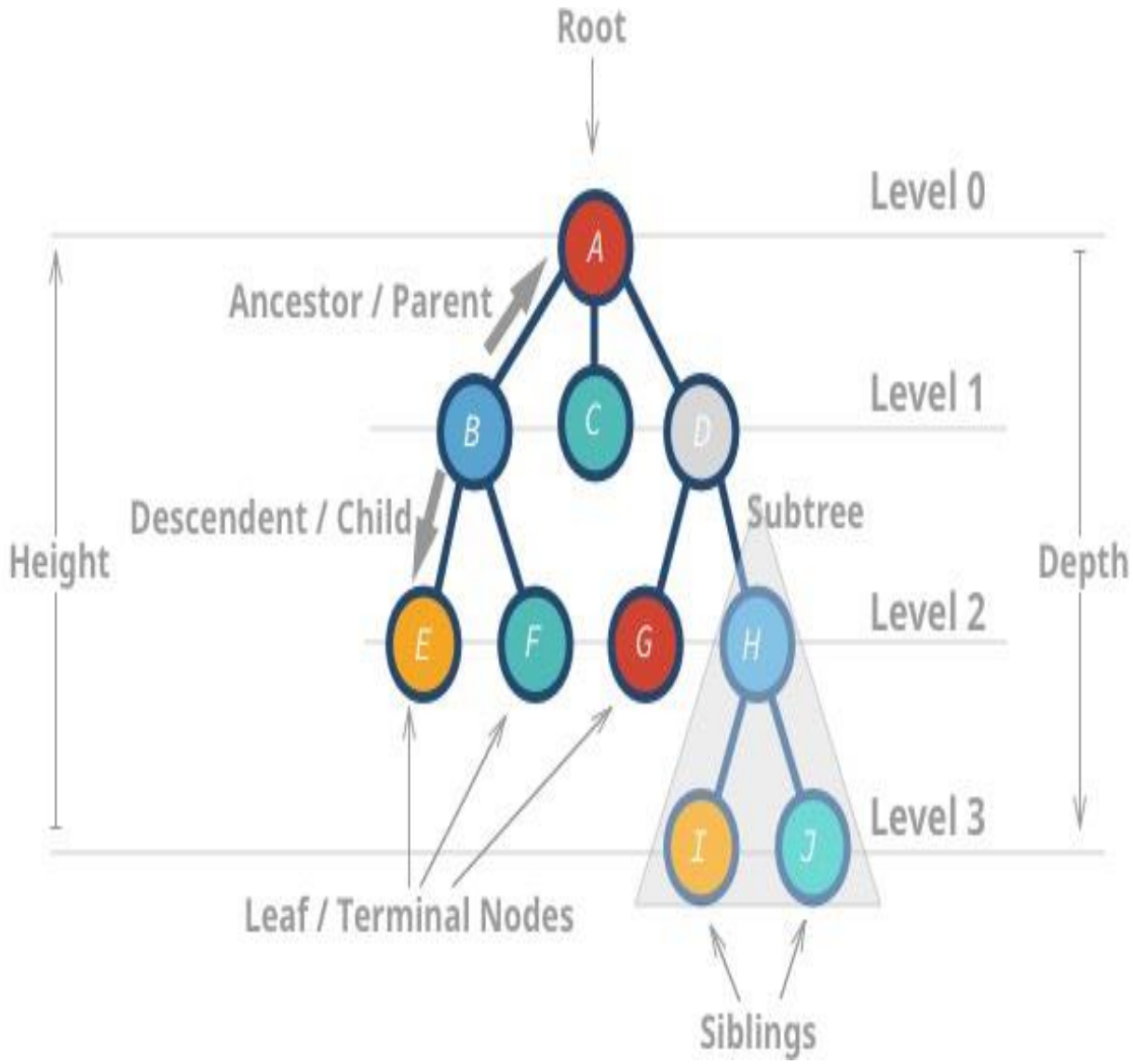


Fig 1.15 Tree Structure

1.5.1 Types of Trees:

Full Binary Tree:

A Binary Tree is full if every node has 0 or 2 children. Following are

```
        18
       /  \
      15   30
     / \   / \
    40  50 100 40
```


`

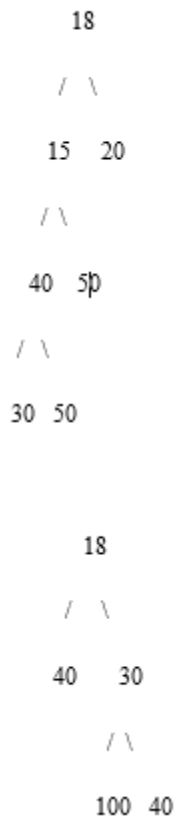


Fig 1.16 Full Binary Tree

In a Full Binary, number of leaf nodes is number of internal nodes plus 1

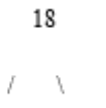
L = I + 1

Where L = Number of leaf nodes, I = Number of internal nodes

Complete Binary Tree:

A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.

Following are examples of Complete Binary Trees:



`

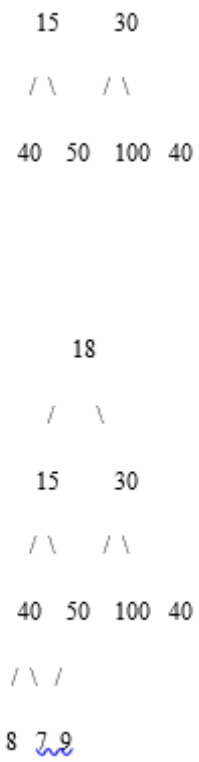


Fig 1.17 Complete Binary Tree

Practical example of Complete Binary Tree is Binary Heap.

Perfect Binary Tree:

A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.
Following are examples of Perfect Binary Trees.

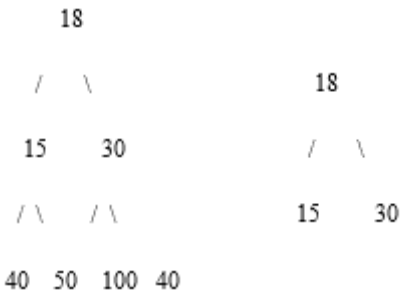


Fig 1.18 Perfect Binary Tree

A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h - 1$ node.

`

Example of a Perfect binary tree is ancestors in the family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree:

A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes. For Example, AVL tree maintains O (Log n) height by making sure that the difference between heights of left and right subtrees is at most 1. Red-Black trees maintain O (Log n) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O (log n) time for search, insert and delete.

AVL Tree:

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. Named after their inventors, **A**delson-**V**elskii and **L**andis, they were the first dynamically balanced trees to be proposed.

An Example Tree that is an AVL Tree:

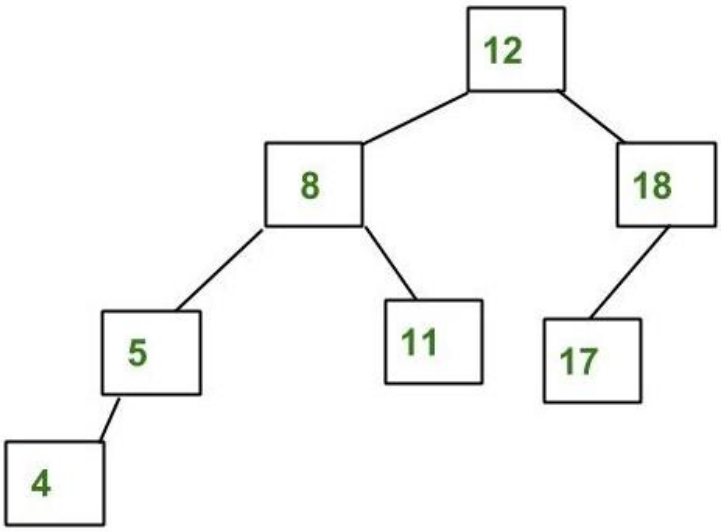


Fig 1.19 AVL Tree

`

M- way Tree:

The m-way search trees are multi-way trees which are generalised versions of binary trees where each node contains multiple elements. In an m-Way tree of order m, each node contains a maximum of m – 1 elements and m children.

The goal of m-Way search tree of height h calls for O(h) no. of accesses for an insert/delete/retrieval operation. Hence, it ensures that the height h is close to log (n + 1)**.**

The number of elements in an m-Way search tree of height h ranges from a minimum of h to a maximum of (m^h) -1.

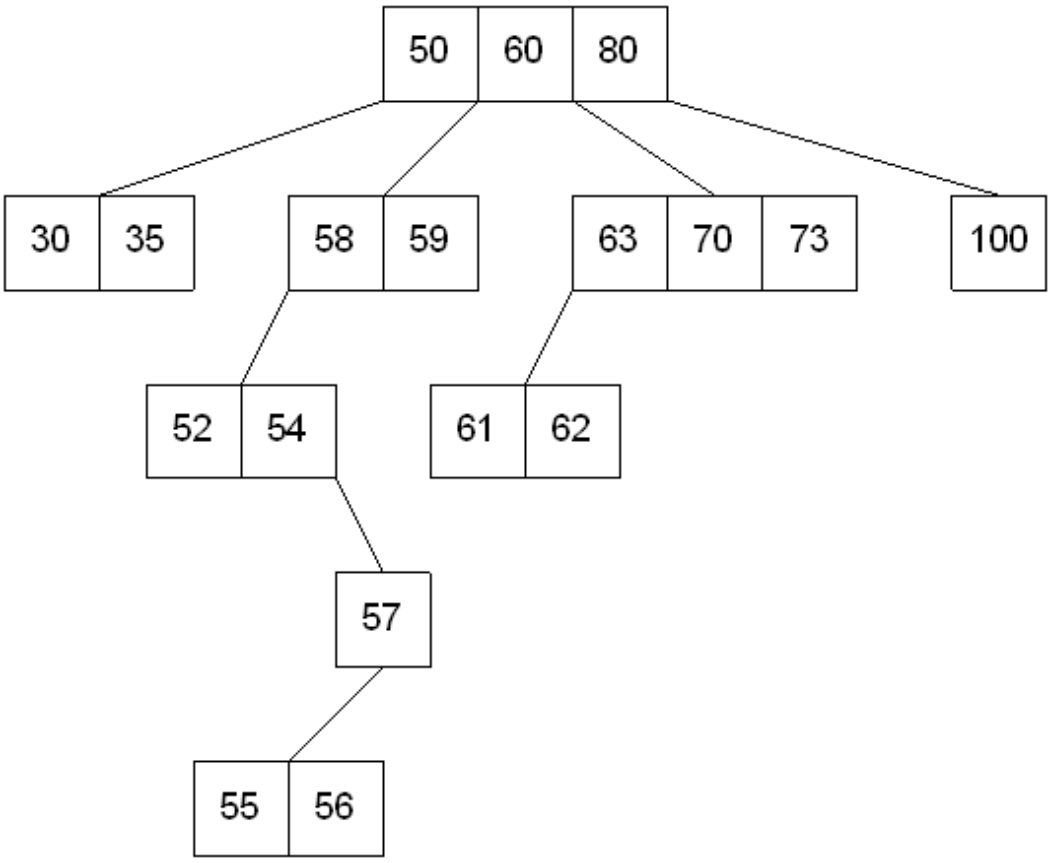


Fig 1.20 M-Way Tree

`

CHAPTER 2: SHORTEST PATH ALGORITHM

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. The shortest path problem can be defined for graphs whether undirected, directed, or mixed. It is defined here for undirected graphs; for directed graphs the definition of path requires that consecutive vertices be connected by an appropriate directed edge.

It can be used for the problem of finding the shortest path between two intersections on a road map may be modelled as a special case of the shortest path problem in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of the segment.

2.1 Applications:

Shortest path algorithms are applied to automatically find directions between physical locations, such as driving directions on web mapping websites like MapQuest or Google Maps. For this application fast specialized algorithms are available.

If one represents a nondeterministic abstract machine as a graph where vertices describe states and edges describe possible transitions, shortest path algorithms can be used to find an optimal sequence of choices to reach a certain goal state, or to establish lower bounds on the time needed to reach a given state. For example, if vertices represent the states of a puzzle like a Rubik's Cube and each directed edge corresponds to a single move or turn, shortest path algorithms can be used to find a solution that uses the minimum possible number of moves.

In a networking or telecommunications mind-set, this shortest path problem is sometimes called the min-delay path problem and usually tied with a widest path problem. For example, the algorithm may seek the shortest (min-delay) widest path, or widest shortest (min-delay) path.

A more light-hearted application is the games of "six degrees of separation" that try to find the shortest path in graphs like movie stars appearing in the same film.

`

Other applications, often studied in operations research, include plant and facility layout, robotics, transportation, and VLSI design.

2.2 DIJKSTRA'S ALGORITHM:

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

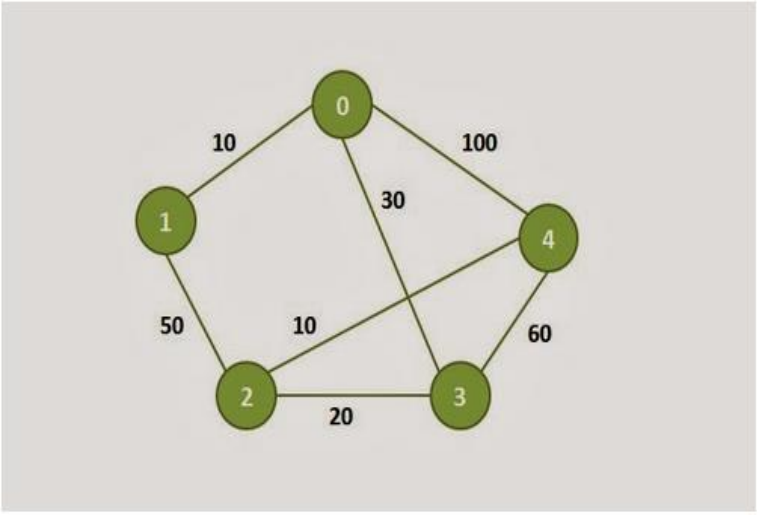
The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree. For a given source node in the graph, the algorithm finds the shortest path between that node and every other.

Dijkstra's algorithm is also called single source shortest path algorithm. It is based on greedy technique. The algorithm maintains a list visited[ ] of vertices, whose shortest distance from the source is already known.

- If visited[1], equals 1, then the shortest distance of vertex i is already known. Initially, visited[i] is marked as 0, for source vertex.

- At each step, we mark visited[v] as 1. Vertex v is a vertex at shortest distance from the source vertex. At each step of the algorithm, shortest distance of each vertex is stored in an array distance[ ].

Example:



`

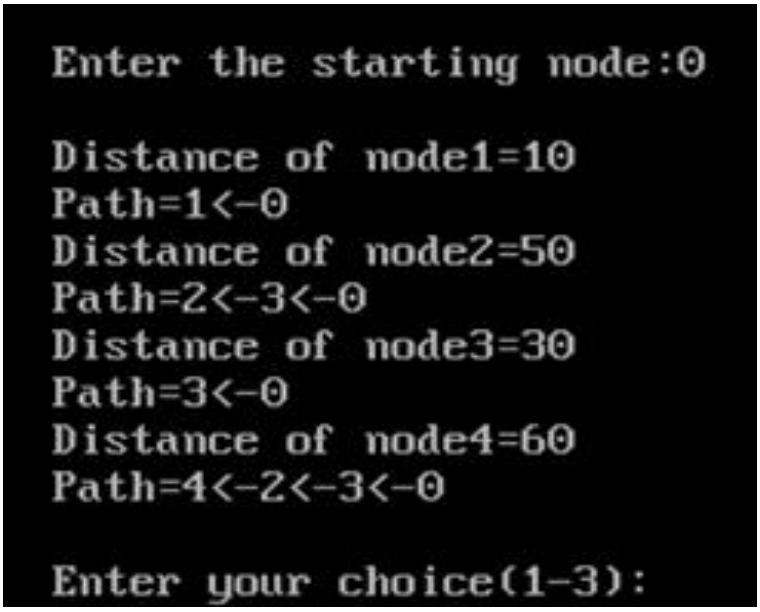


Fig 2.1 Dijkstra Example

`

CHAPTER 3: MINIMUM SPANNING TREE

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.

A minimum spanning tree is a special kind of tree that minimizes the lengths (or "weights") of the edges of the tree. An example is a cable company wanting to lay line to multiple neighbourhoods; by minimizing the amount of cable laid, the cable company will save money.

A **tree** has one path joins any two vertices. A spanning tree of a graph is a tree that:

- Contains all the original graph's vertices.
- Reaches out to (spans) all vertices.
- Is acyclic. In other words, the graph doesn't have any nodes which loop back to itself.

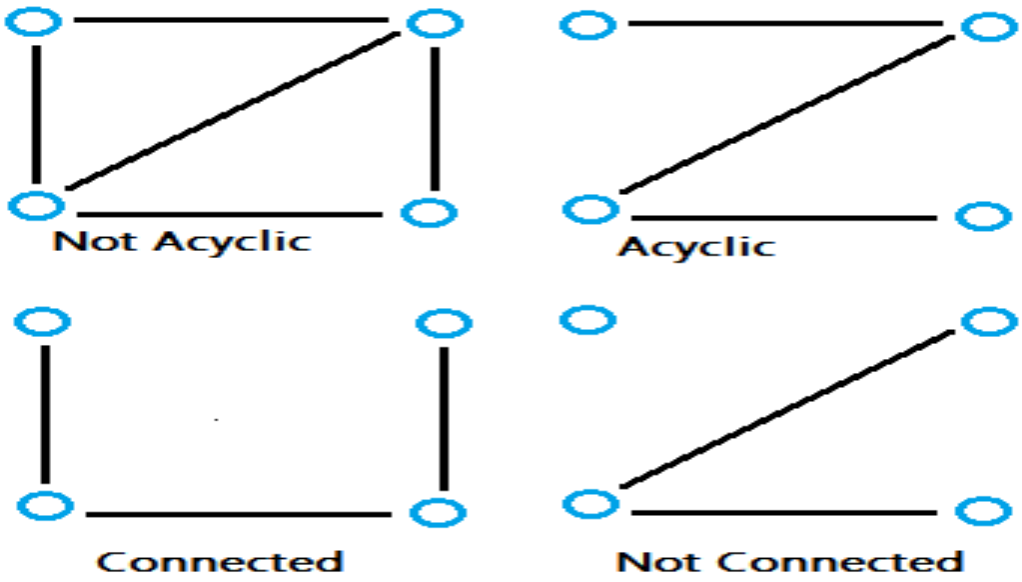


Fig 3.1 Cyclic-Connected Graphs

`

3.1 PRIM'S ALGORITHM:

Prim's (also known as Jarník's) algorithm is a greedy algorithm that finds a minimum spanning tree for a 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. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

Prim's Algorithm	**Kruskal's Algorithm**
In Prim's Algorithm, the tree that we are making or growing always remains connected.	In kruskal's Algorithm, the tree that we are making or growing usually remains disconnected.
Prim's Algorithm will grow a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm will grow a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.
Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

Table 3.1 Difference between Prim's and Kruskal's Algorithm

The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959. Therefore, it is also sometimes called the Jarník's algorithm, Prim–Jarník algorithm, Prim–Dijkstra algorithm-or the DJP algorithm.

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.
A group of edges that connects two set of vertices in a graph is called cut in graph

`

theory. So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Example:

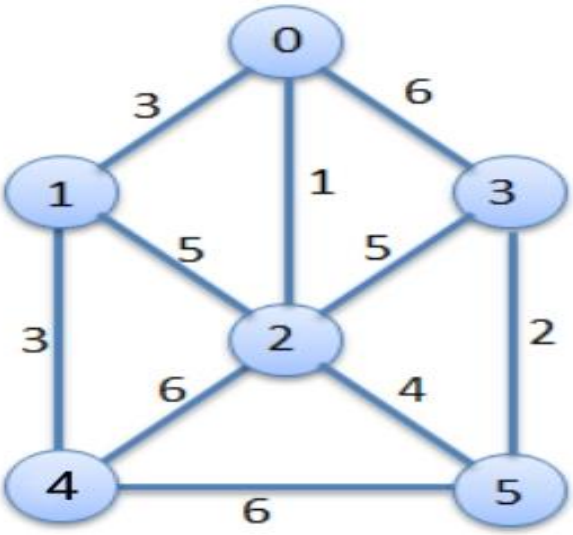


Fig 3.2 MST Example Problem

`

Procedure for finding Minimum Spanning Tree

Step1

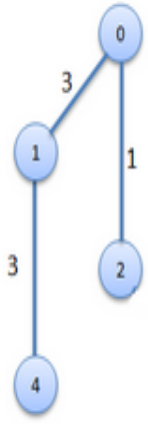
No. of Nodes	0	1	2	3	4	5
Distance	0	3	**1**	6	∞	∞
Distance From		0	0	0		

Step2

No. of Nodes	0	1	2	3	4	5
Distance	0	**3**	0	5	6	4
Distance From		0		2	2	2

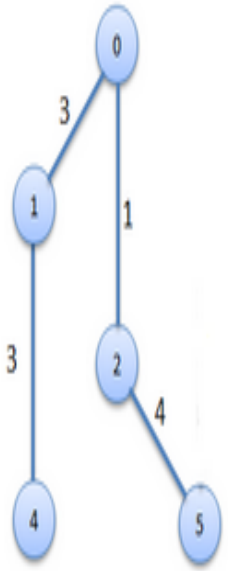
Step3

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	5	**3**	4
Distance From				2	1	2



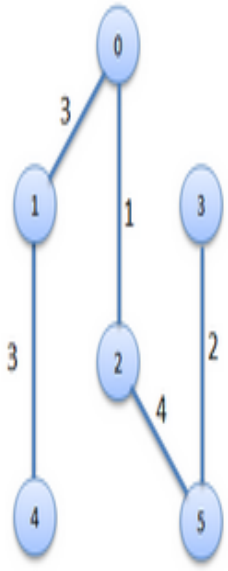
Step4

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	5	0	**4**
Distance From				2		2



Step5

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	**3**	0	0
Distance From				2		2



Minimum Cost = 1+2+3+3+4 = 13

`

CHAPTER 4: SOFTWARE DESIGN AND DEVELOPMENT

Operating System:

- Windows 7, Windows 8 or Windows 10

Hardware Requirement:

- Processor (CPU) with 2 gigahertz (GHz) frequency or above
- A minimum of 2 GB of RAM
- Monitor Resolution 1024 X 768 or higher
- A minimum of 20 GB of available space on the hard disk
- Internet Connection Broadband (high-speed) Internet connection with a speed of 4 Mbps or higher
- Keyboard and a Microsoft Mouse or some other compatible pointing device
- Sound card
- Speakers or headphones
- Strongly Recommended - Microphone and Webcam

Browsers Used:

- Chrome* 36+
- Edge* 20+
- Mozilla Firefox 31+
- Internet Explorer 11+ (Windows only)

`

Software Requirement:

Programing Language: C language, HTML, Notepad.

Compiler: Turbo C++, Notepad ++.

`

CHAPTER 5: SOFTWARE TESTING

5.1 OUTPUT OF C - PROGRAM:

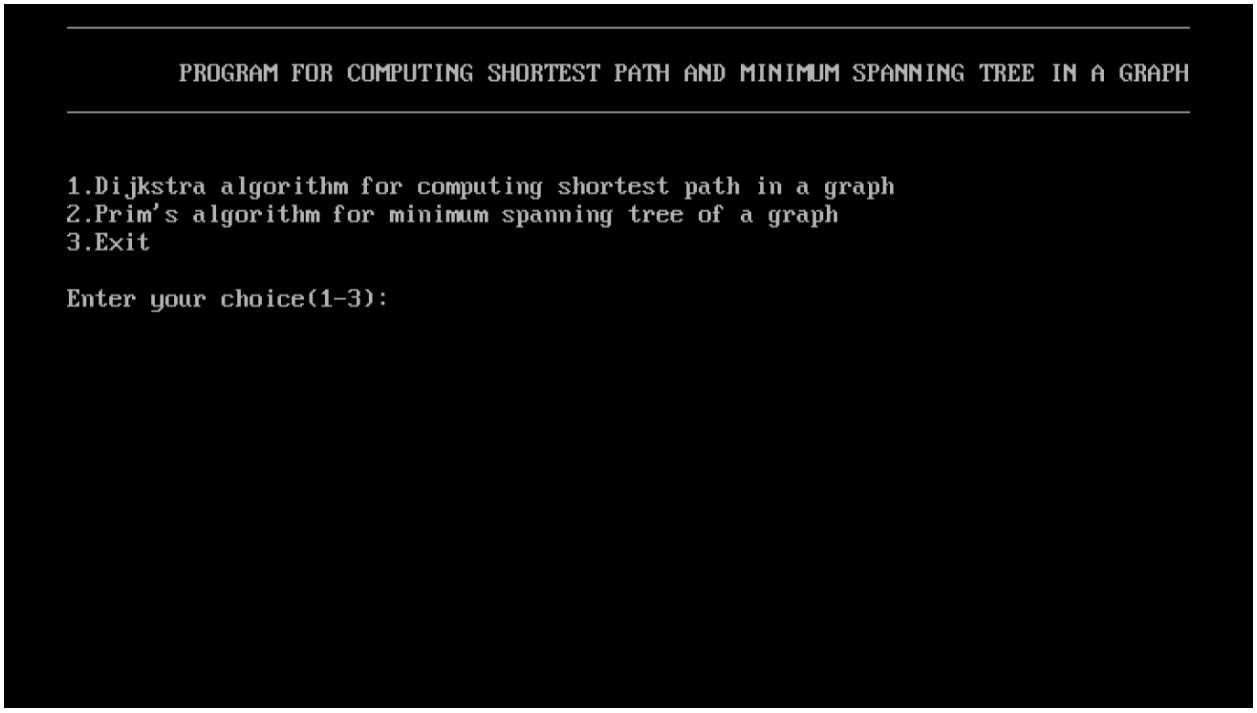


Fig 5.1 Program Output (a)

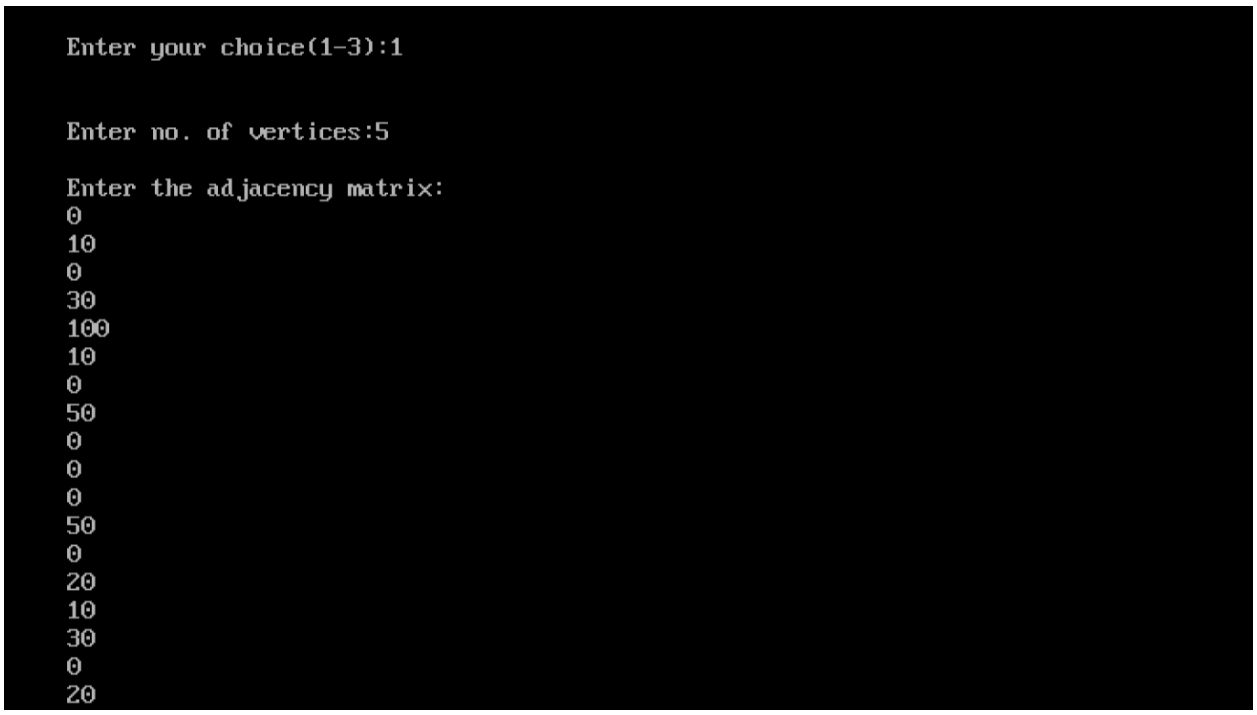


Fig 5.2 Program Output (b)

`

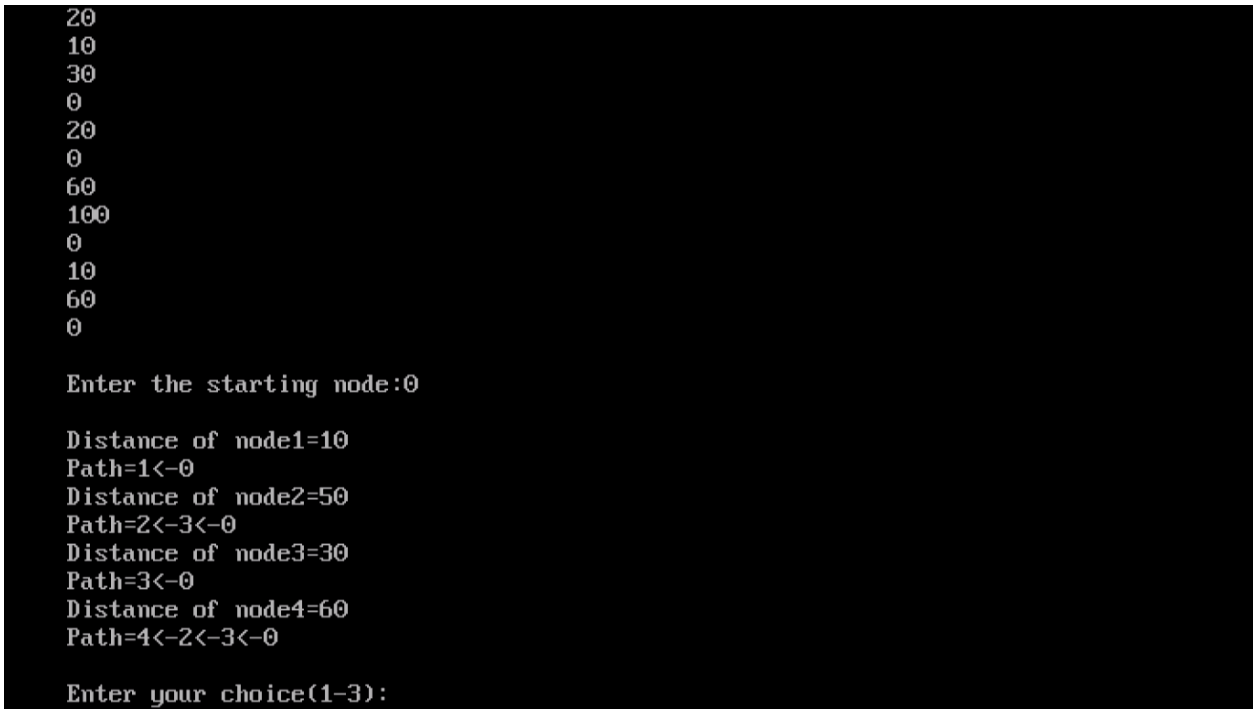


Fig 5.3 Program Output (c)

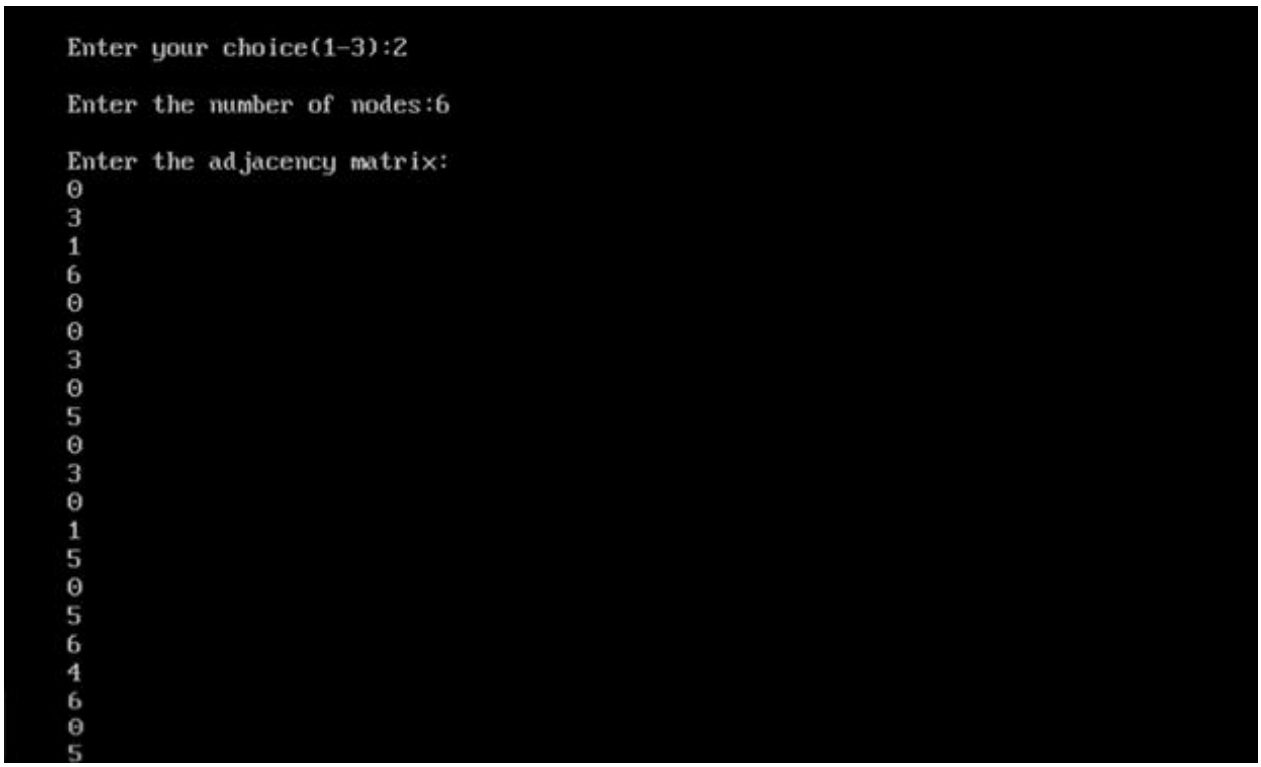


Fig 5.4 Program Output (d)

`

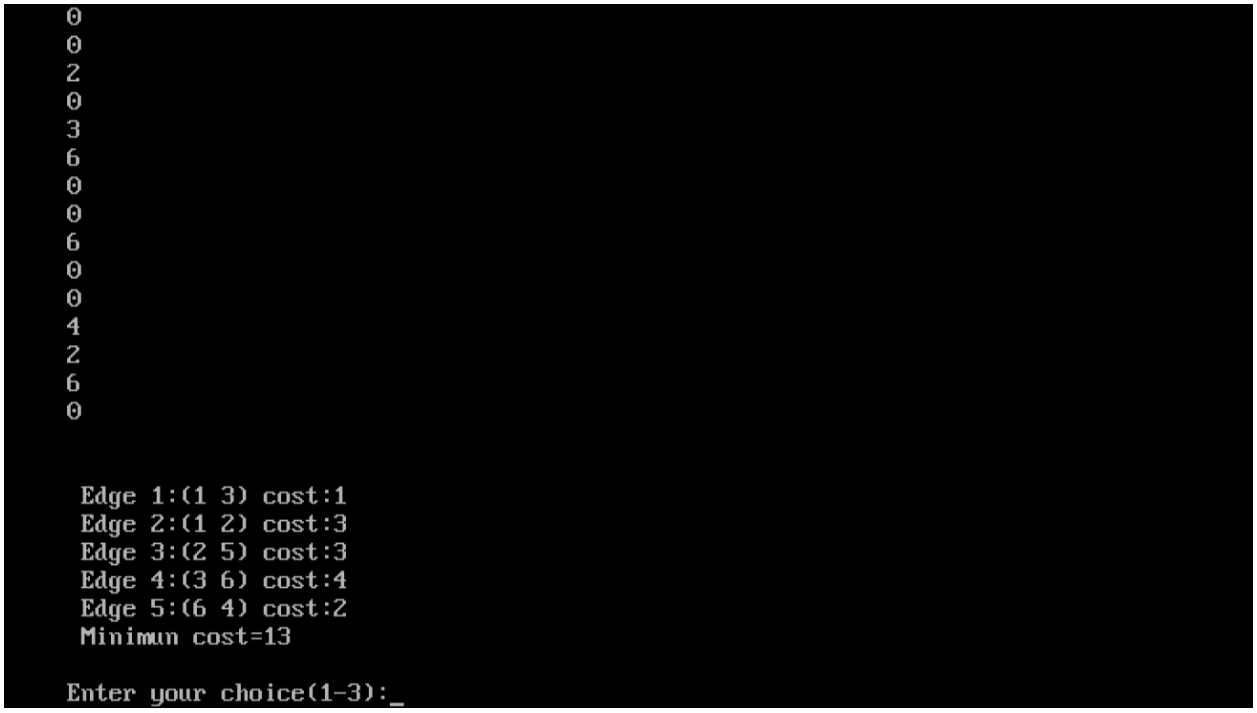


Fig 5.5 Program Output (e)

5.2 OUTPUT OF WEB PAGES:

a)

Fig 5.6 Web Pages Output (a)

`

b)

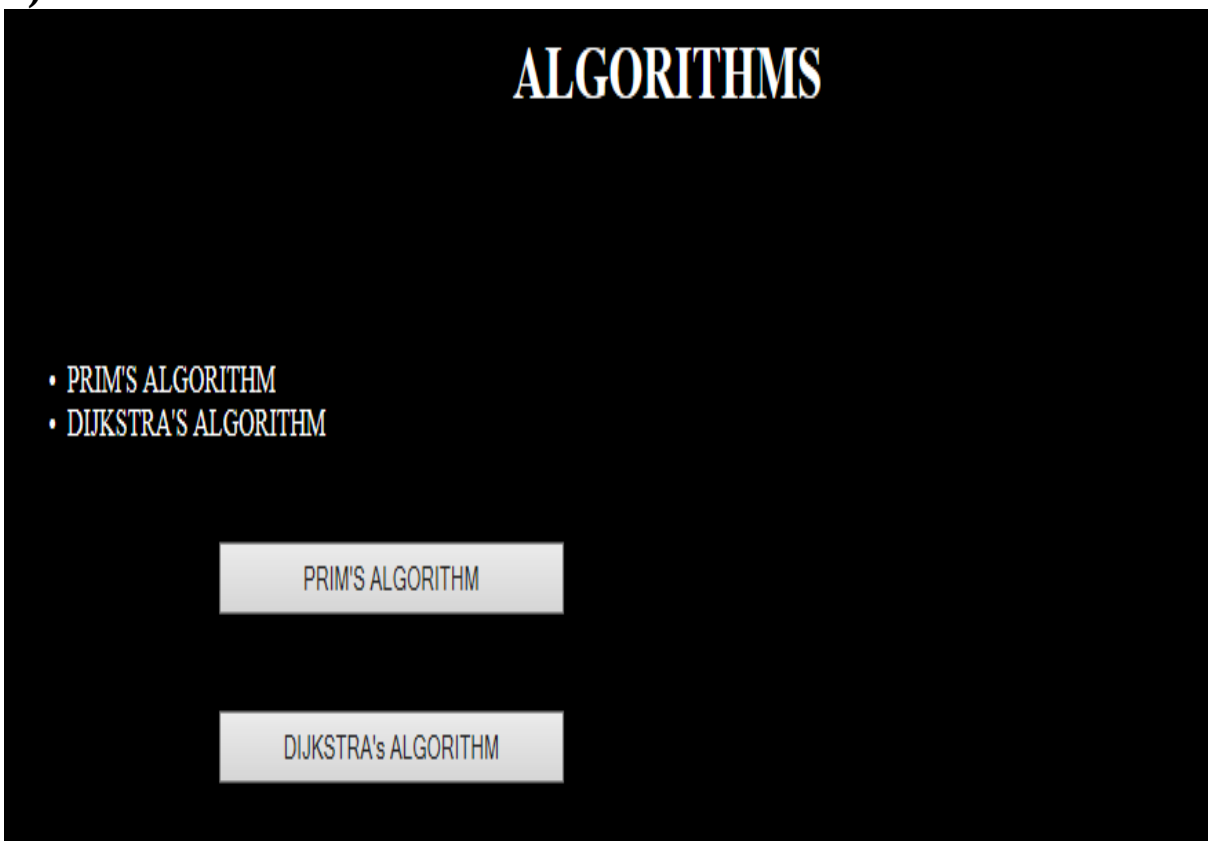


Fig 5.7 Web Pages Output (b)

c)

Fig 5.8 Web Pages Output (c)

`

CHAPTER 6: CONCLUSION

Shortest path and minimum spanning tree both are the parts of Data Structure and they play vital role in scientific and real life examples.

- Minimum cost required
- Optimizing routing of packets on the internet
- Protocols in computer science to avoid network cycles
- Cluster analysis

`

REFERENCES

1) E. Balaguruswamy (2011) *Programming in ANSI C*

2) Michael T. Goodrich (2003) *Data Structures and Algorithm*

3) Jean - Paul Tremblay and Paul G. Sorenson (2017) *An Introduction to Data Structures with Applications*

4) Jon Duckett (2011) *HTML Design*

`

APPENDIX

SOURCE CODE OF C PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
#include<time.h>
#define INFINITY 9999
#define MAX 20

void dijkstra(int G[MAX][MAX],int n,int startnode);
int a,b,u,v,n,i,j,ne=1;
int visited[10]={0},min,mincost=0,cost[10][10];
void main()
{
        clock_t starting_time,finishing_time;
        int i,j,total_cost;
        int G[MAX][MAX],n,u;
        int ch;
        starting_time=clock();
        finishing_time=clock();
        clrscr();
        printf("\nStarting time:\t%ld",starting_time);
printf("_____
_____");
        printf("\n\tPROGRAM FOR COMPUTING SHORTEST PATH AND
MINIMUM SPANNING TREE IN A GRAPH");
        printf("_____
_____");
        printf("\n\n1.Dijkstra algorithm for computing shortest path in a graph");
        printf("\n2.Prim's algorithm for minimum spanning tree of a graph");
        printf("\n3.Exit");
        do
```


`

```
{
printf("\n\nEnter your choice(1-3):");
scanf("%d",&ch);
switch(ch)
{
        case 1: //FOR DIJKSTRA
                printf("\n\nEnter no. of vertices:");
                scanf("%d",&n);
                printf("\nEnter the adjacency matrix:\n");
                for(i=0;i<n;i++)
                for(j=0;j<n;j++)
                        scanf("%d",&G[i][j]);
           printf("\nEnter the starting node:");
                scanf("%d",&u);
                dijkstra(G,n,u);
                break;

        case 2: //FOR PRIM'S
                printf("\nEnter the number of nodes:");
                scanf("%d",&n);
                printf("\nEnter the adjacency matrix:\n");
                for(i=1;i<=n;i++)
                for(j=1;j<=n;j++)
                {
                        scanf("%d",&cost[i][j]);
                        if(cost[i][j]==0)
                        cost[i][j]=999;
                }
                visited[1]=1;
                printf("\n");
                while(ne < n)
                {
                        for(i=1,min=999;i<=n;i++)
                        for(j=1;j<=n;j++)
```


`

```
                                if(cost[i][j]< min)
                                if(visited[i]!=0)
{
                                        min=cost[i][j];
                                        a=u=i;
                                        b=v=j;
                                }
                                if(visited[u]==0 || visited[v]==0)
                                {
                                        printf("\n Edge %d:(%d %d)
cost:%d",ne++,a,b,min);
                                        mincost+=min;
                                        visited[b]=1;
                                }
                                cost[a][b]=cost[b][a]=999;
                        }
                        printf("\n Minimun cost=%d",mincost);
                        break;

                case 3: exit(0);
                        break;

                default: printf("\nWrong choice!!");
                         break;
        }
        }while(ch!=0);
printf("\nEnding time:\t%ld",finishing_time);
        printf("\nExecution time:\t%ld\n",finishing_time-starting_time);
        getch();
}

void dijkstra(int G[MAX][MAX],int n,int startnode)
{
        int cost[MAX][MAX],distance[MAX],pred[MAX];
```


`

```
        int visited[MAX],count,mindistance,nextnode,i,j;

        //pred[] stores the predecessor of each node
        //count gives the number of nodes seen so far
        //create the cost matrix
        for(i=0;i<n;i++)
                for(j=0;j<n;j++)
                        if(G[i][j]==0)
                                cost[i][j]=INFINITY;
                        else
                                cost[i][j]=G[i][j];

        //initialize pred[],distance[] and visited[]
        for(i=0;i<n;i++)
{
                distance[i]=cost[startnode][i];
                pred[i]=startnode;
                visited[i]=0;
        }

        distance[startnode]=0;
        visited[startnode]=1;
        count=1;

        while(count<n-1)
        {
                mindistance=INFINITY;
                //nextnode gives the node at minimum distance
                for(i=0;i<n;i++)
                        if(distance[i]<mindistance&&!visited[i])
                        {
                                mindistance=distance[i];
                                nextnode=i;
                        }
```


`

```
            //check if a better path exists through nextnode
                visited[nextnode]=1;
                for(i=0;i<n;i++)
                    if(!visited[i])
                    if(mindistance+cost[nextnode][i]<distance[i])
                        {
                            distance[i]=mindistance+cost[nextnode][i];
                            pred[i]=nextnode;
                        }
            count++;
        }
        //print the path and distance of each node
        for(i=0;i<n;i++)
            if(i!=startnode)
            {
                printf("\nDistance of node%d=%d",i,distance[i]);
                printf("\nPath=%d",i);

                j=i;
                do
                {
                    j=pred[j];
                    printf("<-%d",j);
}
while(j!=startnode);
}}
```

SOURCE CODE OF WEB PAGES

a)

```
<!DOCTYPE html>
<html>
<head>
    <title>my file</title>
```


`

```
	<style>
		form{border: 1px solid; color:white; border-color: white; width:30%;
padding:5px;}
	</style>
</head>
<h1 style="color:white ;"> Web Based Software for computing shortest path and
minimum spanning tree in graphs</h1>

<body bgcolor=black>
	<div style="margin-left: 500px; margin-top: 100px;">
	<form action="1.html" style="font-size:30px;" >
			USERNAME <input type="text" name="username"
placeholder="username" required / style="margin-left: 40px; height:25px;"><BR>
	PASSWORD<input type="password" name="password" placeholder="password"
required / style="margin-left: 40px; height:25px;"><BR><br>
	<input type="submit" name="s" value="SUBMIT" style="width:90px;">
</form>
<form action= 2.html >
	<input type="submit" name="sub" value="SIGN UP" style="width: 90px;">
</form>
</div>

</body>
</html>

b)
<!DOCTYPE html>
<html>
<head>
	<title>NEWPAGE1</title>
</head>
<body bgcolor=black style="color:white;">
<h1 align="center">ALGORITHMS</h1>
	<div style="margin-left:200px; margin-top:100px;">
```


`

```
        <ul>
                <li>PRIM'S ALGORITHM</li>
                <li>DIJKSTRA'S ALGORITHM</li>
</ul>
</div>
        <form action="3.html" style="margin-left:350px; margin-top:40px;">
                <input type="submit" name="s3" value="PRIM'S ALGORITHM"
style="height:30px; width:250px;">

        </form>

        <form action="4.html" style="margin-left:350px; margin-top:40px;">
                <input type="submit" name="s4" value="DIJKSTRA's ALGORITHM"
style="height:30px; width:250px;">
        </form>

</body>
</html>
```

c)

```
<!DOCTYPE html>
<html>
<head>
        <title>signup</title>
        <style>
                form{color:white; border: 1px solid; padding:5px ;  width:30%;}
        </style>
</head>
<body bgcolor=black style="color:white;">
<div style="margin-left:400px; margin-top:100px;">
<form action="adity1.html" style="font-size:25px;">
        NAME<input type="txt" name="n1" placeholder="enter your name" required/
style="margin-left:90px; "><br>
```


`

NEW EMAIL<input type="email" name="n2" placeholder="enter your email" style="margin-left: 90px;">

NEW PASSWORD <input type="password" name="p1" placeholder="enter your password" required/ style="margin-left: 90px;">

<input type="submit" name="SIGN UP " style="height: 25px; width: 100px;">

</form>

</body>

</html>

`