UNIT – IV

Graph Structure: Graph representation – Adjacency matrix, adjacency list, Warshall's algorithm, adjacency multilist representation. Orthogonal representation of graph. Graph traversals – bfs and dfs. Shortest path, all pairs of shortest paths, transitive closure, reflexive transitive closure.

UNIT – IV

Chapter ▸ 5

Graph Structure

5.1 Introduction

Graph is a abstract type of data structure which implement the mathematical concept of graphs. Graph (V,E) is a collection of finite set of vertices (V)and edges (E)which connect these vertices. A graph is often viewed as generalization of the tree structure ,where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.Graph are widely used to model any situation where entities or things are related to each other in pairs.

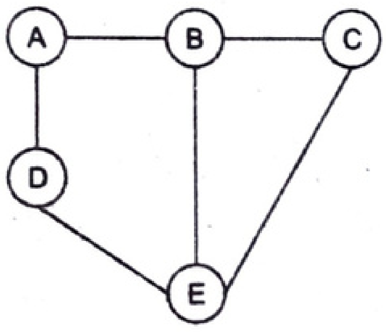


Figure 5.1 Undirected Graph

5.2 Graph Terminology

Adjacent node: For every edge e=(u,v) that connects the node u and v , the node u and v are end points of edge , we call u and v are adjacent to each other. u and v are neighbor of each other.

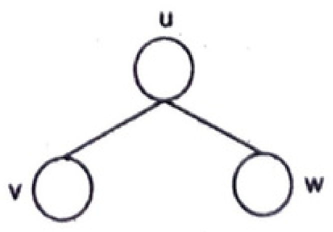


Figure 5.2 Adjacent Node

Here u and v are adjacent each other.

Degree of node: First clear that node is alternate name of vertex. Number of edges incoming or outgoing from vertex is called degree of vertex. Number of edges enter to a vertex is called it's incoming degree, number of edges leave to vertex is called outdegree of vertex.

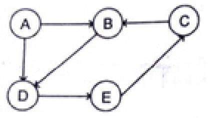


Figure 5.3 Degree of Node

Here the degree of A node is two. In-degree of A node is Zero and Out-degree is two.
Closed Path: A path p is known as closed path if the edge has the same end-points. That is, if $v_0=v_n$.
Simple Path: A path P is known as simple path if all the nodes in the path are distinct with an exception that v_0. If $v_0=v_n$ then the path is called as closed simple path.
Cycle: A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and the last vertices).
Connected graph: A graph is saide to be connected if for any two vertices (u,v) in G there is a path from u to v. It means there will be no isolated (isolated means a vertex whose degree is zero) vertex in a graph.A connected graph doesnot contain any cycle. Tree is a good example of connected graph (no cycle).

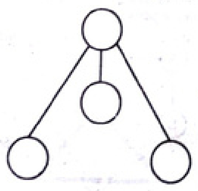


Figure 5.4 Connected Graph

Complete Graph: A graph in which each vertex connected to remaining all vertices is called a complete graph. It means if a graph has n vertices then each vertex is connected to n-1 vertices. A complete graph is always a connected graph but all connected graphs are not always complete graph.

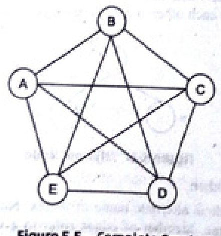


Figure 5.5 Complete Graph

***Note:* Complete graph is connected graph but every connected graph is not complete graph.**

Regular graph: A graph is said to be regular if each vertex has equal degree.For example , given graph is regular because each vertex has 2-degree. It is called 2-regular graph.

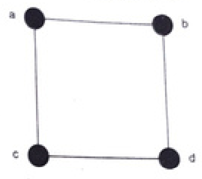


Figure 5.6 Regular Graph

n-regular graph: As we discussed above that in regular graph in each vertex has equal degree.

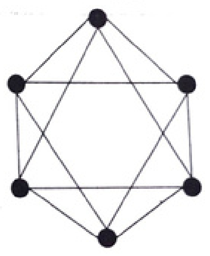


Figure 5.7 n-regular Graph

Multiple Edge: Multiple edge or parallel edge , are two or more edges that are incident on the same vertices. A simple graph doesnot have any multiple edges.

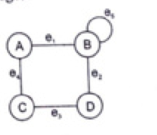


Figure 5.8 Multiple Edge

Size of graph: The size of graph is the total number of edges in it

5.3 Directed Graph

A directed graph , sometimes called digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u,v) of nodes in G. For an edge (u,v),

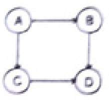


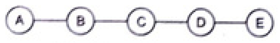
Figure 5.9 Directed Graph

- The edge begins at u and terminates at v.
- U is known as the origin or initial point of edge. Correspondingly,v is knowna as the destination or terminal point of edge.
- U is predecessor of v. Correspondingly,v is successor of u.
- Nodes u and v are adjacent to each other.

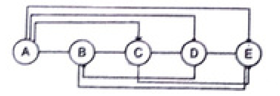
5.3.1 Transitive Closure of a Directed Graph

A transitive closure of a graph is for the answer of question of reachability. Reachability means , is there a path from a node A to node E in one or more hops(counts)? A binary relation indicates only whether the node A is connected to node B, whether node B is connected to node C, etc.But once transitive closure is shown as in fig. we can easily determine O(1) time whether the node E is reachable from node A or not.Transitive closure used to stored as a matrix T, so if T[1][5]=1. Then node 5 can be reached from node 1 in one hops.

Definition: For a directed graph G= (V,E), where Vis set of vertices and E is the set of edges, the transitive closure of G is a graph G*=(V,E*). In G*, for every vertex pair v,w in V there is an edge(v,w) in E* if and only if there is a valid path v to w in G.



(a) A Graph



(b) Transitive Closure

Figure 5.10 Transitive closure of a Directed Graph

Areas where Transitve closure needed

- Transitve closure is used to find the reachability analysis of transition networks representing distributed and parallel system.
- It is used in the construction of parsing automata in compiler.
- Recently, transitive closure computation is being used to evaluate recursive database queries

5.4 Representation of Graph

There are three common ways of storing graphs in computer's memory. They are:

- Sequential representation by using adjancey matrix.
- Linked representation by using an adjency list that stores the neighbours of a node using linked list.
- Adjancey multi-list which is an extension of linked representation.

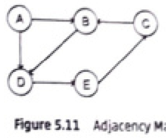
5.4.1 Adjacency Matrix Representation

An Adjancey matrix is used to represent which nodes are adjacent to one another. Two nodes are said to be **adjacent if** there is an edge connecting them.

In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That **is, if v is adjacent** to u, we can get from u to v by traversing one edge. For any graph G having n nodes , the **adjancey matrix** will have the dimension m×n.

In an adjacency matrix , the rows and columns are labeled by graph vertices. An entry a_{ij} in the adjacency **matrix will contain** 1, if vertices v_i and v_j are adjacent to each other.If the nodes are not adjacent to each other **a_{ij} will be** set to zero.

Since an adjacent matrix contains only 1s or 0s, it is called Bit matrix or a Boolean matrix.The entries in **the matrix depend** on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a **different adjacency** matrix.Fig 5.11 shows some graphs and their corresponding adjacency matrices.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 0 & 0 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

Figure 5.11 Adjacency Matrix Representation

Note: By observing the graphs,we come on conclusion that,

- For a simple graph(that has no loops) the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is $O(n^2)$, where n is the number of nodes in the graph.
- Numbers of 1s (or non- zero entries)in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

5.4.2 Adjacency List

An Adjacency List is another way in which graphs can be represented in the computer's memory.This structure contains a List of all nodes in G. Firthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

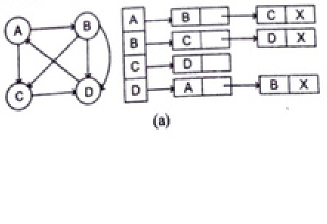
Benefits of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate numbers of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise an adjacency matrix is good choice.
- Adding a new nodes in G is easy and straightforward when G is represented using an adjacency list.

Fig 5.12(a,b,c) shows how the graph stored in computer's memory in form of adjacency list.

Fig 5.12(a) Graph G and its adjacency List (b) Adjacency List for an undirected graph

(c) Adjacency List fo a weighted graph



(a)

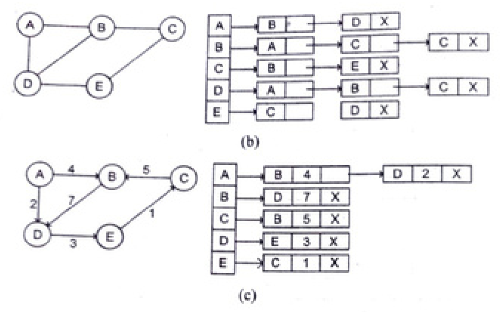


Figure 5.12 Adjacency List

5.4.3 Adjacency Multi-List Representation

An adjacency multi-list is a modified version of adjacency lists. Adjacency multi-list is an edge based rather than a vertex based representation of graphs. A multi-list representation basically consists of two parts-a directory of node's information and a set of linked lists storing information about the edges. While there is a single entry for each node in the node directory , every node, on the other hand, appears in two adjacency lists(one for the node at each end of the edge). For instance, the directory entry of node I points to the adjacency list for node i . It means the nodes are shared among several lists.

In a multi-list representation , the information about an edge (v_i,v_j) of an undirected graph can be stored using the following attributes:

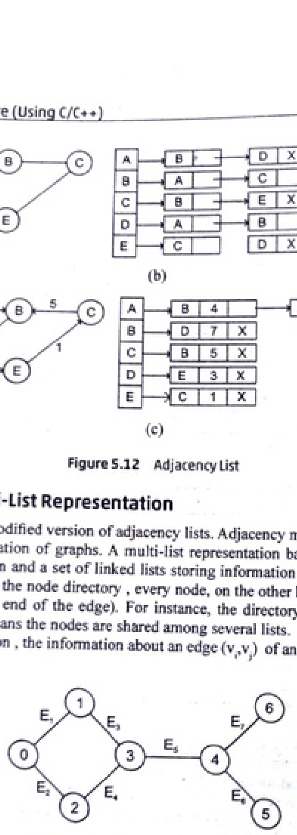


Figure 5.13 Adjacency Multi-List Representation

v_i: Avertex in the graph that is connected to vertex v_j by an edge.
V_j: A vertex in the graph that is connected to vertex v_i by an edge.
Link i for v_i: A link that points to another node that has an edge incident on V_j
Link j for v_j: A link that points to another node that has an edge incident on v_i.
Consider the fig 6.13 the adjacency multi-list for the graph can be given as:

Edge1

	0	1	Edge 2	Edge 3

Edge 2

	0	2	NULL	Edge 4

Edge 3

	1	3	NULL	Edge 4

Edge4

	2	3	NULL	Edge 5

Edge5

	3	4	NULL	Edge 6

Edge 6

	4	5	Edge 7	NULL

Edge 7

	4	6	NULL	NULL

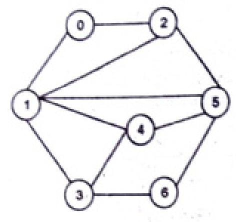
Using Adjacency multi-list given above, the adjacency list for vertices can be constructed as shown below:

VERTEX	LIST OF EDGES
0	Edge1, Edge2
1	Edge1, Edge3
2	Edge2, Edge4
3	Edge 3, Edge 4, Edge5
4	Edge5, Edge6, Edge7
5	Edge6
6	Edge7

5.5 Orthogonal Representation of Graph

Orthogonal representations of graphs were introduced by Lovisz (1979) in the study of the Shannon capacity of a graph. An orthogonal representation of G in Rd is an assignment f: V(G) ->Rd such that f(u) and f(v) are orthogonal for every pair of distinct nonadjacent nodes u and v. An orthonormal representation is an orthogonal representation such that $||f(u)|| = 1$ for every u ϵ V(G). We say that the orthogonal representation is in general position if every set of d representing vectors is linearly independent. If f is a general-position orthogonal representation, then $f(u) \neq 0$ and so $f(u)/||f(u)||$ is a general-position orthonormal representation in the same dimension. Another natural "nondegeneracy" property of an orthogonal representation is to be faithfull: this means that f(u) and f(v) are orthogonal if and only if u and v are nonadjacent. Another natural "nondegeneracy" property of an orthogonal representation is to be faithfull: this means that f(u) and f(v) are orthogonal if and only if u and v are nonadjacent. The assignment f = 0 is a trivial orthogonal representation for any graph. In dimension n = |V(G)|, every graph G has a general-position orthonormal representation (using n mutually orthogonal unit vectors). It is easy to give a faithful representation in this same dimension. It seems to be difficult to find the smallest dimension in which a given graph G has an orthonormal representation.

Example:



G3

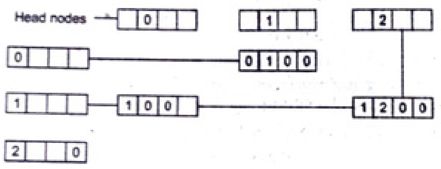


Figure 5.14 Orthogonal Representation for G3

5.6 Traversal Methods

Traversing of graph means examining the nodes and edges of the graph. There are two standard methods of graph traversal which we will going to discuss.

- Breadth first Search
- Depth first Search

5.6.1 Breadth First Search

Breadth first uses a **Queue** as an auxiliary data structure to store nodes for further processing, the Depth first search uses a **Stack**.

During execution of breadth first algorithm , each node N of G will be in one of three states, called the status of N, as follows;

STATUS =1; (Ready state) The initial state of the node N.
STATUS =2; (Waiting state) The node N is on the queue or stack , waiting to be processed.
STATUS=3; (Processed state) The Node N has been processed.

Algorithm

```
Step 1: Initialize all nodes to the ready state (STATUS=1)
Step 2: Put  the starting node A in QUEUE and change its status to the waiting
   state (STATUS=2)
Step 3: Repeat step 4 and 5 until QUEUE is empty
```

```
step 4: Remove the front node N of QUEUE . Process N and change the status of N
   To the processed  state (STATUS =3)
step 5: Add to the rear of QUEUE all the neighbours of N that are in the steady
   state(STATUS=1), and change their status to the waiting state (STATUS=2).
   [End of step 3 loop]
step 6: Exit
```

Example: Consider a graph G represents the daily flight between cities of some airline, and suppose want to fly from city A to city J with the minimum number of stops.

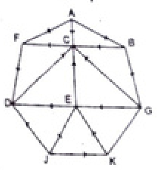


Figure 5.15 Breadth First Search

The minimum path P can be found by using a breadth-first search beginning at city A and ending when J is encountred . During the execution of the search , we will also keep track of the origin of each edge by using array ORIG together with the array QUEUE.The steps of our search follow.

(a) Initially, add A to Queue and add NULL to ORIG as follows:

FRONT=1	QUEUE: A
REAR =1	ORIG : Ø

(b) Remove the front element A from QUEUE by setting FRONT=FRONT+1, and add to QUEUE the neighbours of A as follows:

FRONT=2	QUEUE: A,F,C,B
REAR =4	ORIG : Ø,A,A,A

(c) Remove the front element f from QUEUE by setting FRONT=FRONT+1, and add to QUEUE the neighbous of f as follows:

FRONT=3	QUEUE: A,F,C,B,D
REAR =5	ORIG : Ø,A,A,A,F

(d) Remove the front element C from QUEUE , and add to QUEUE the neighbours of c (which are in the ready state) as follows:

FRONT=4	QUEUE: A,F,C,B,D
REAR =5	ORIG : Ø,A,A,A,F

(e) Remove the front element B from QUEUE , and add to QUEUE the neighbours of B as follows:

FRONT=5	QUEUE: A,F,C,B,D,G
REAR =6	ORIG : Ø.a,A,A,F,B

(f) Remove the front element D from QUEUE , and add to QUEUE the neighbours of D as follows:

FRONT=6	QUEUE: A,F,C,B,D,G
REAR =6	ORIG : Ø,A,A,A,F,B

(g) Remove the front element G from QUEUE and add to QUEUE the neighbours of G as follows:

FRONT=7 QUEUE: A,F,C,B,D,G,E

REAR =7 ORIG : Ø,A,A,A,F,B,G

(h) Remove the front element E from QUEUE and add to QUEUE the neighbours of E as follows

FRONT = 8 QUEUE : A,F,C,B,D,G,E,J

REAR =8 ORIG : Ø, A,A,A,F,B,G,E

We stop as soon as J is added to QUEUE , since J is our final destination . We now backtrack from J, using the array ORIG to find the Path. Thus

J<--E<--G<--F<--A

Space Complexity: All the nodes at a particular level must be saved until their child nodes in the next level has been generated. The space complexity is therefore to the number at the deepest level of graph.

Time complexity: In the worst case , breadth first search has to traverse through all paths to all possible node, thus the time complexity of this algorithm asymptotically approaches $O(b^{s})$.

However, the time complexity can also be expressed as $O(|E|+|V|)$,since every vertex and every edge will be explored in the worst case.

Completeness: Breadth-first search is said to be complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution ,it will diverge.

Optimality: Breadth-first search is optimal for a graph that has edges of equal length,since it always returns the result with the fewest edges between start and the goal node. But generally, in real-world applications , we have weighted graphs that have costs associated with each edge,so the goal next to the start doesnot have to be cheapest goal available.

5.6.2 Depth-First Search

The Depth-first algorithm processed by expanding the starting node of G and then going deeper and deeper until the goal is found,or until a node that has no children.When a dead-end is reached, the algorithm backtracks ,returning to the most recent node that has not been completely explored.

Algorithm

```
Step 1: Initialize all nodes to the ready state (STATUS=1)
Step 2: Push the starting node A onto STACK and change its status to the waiting
    state (STATUS=2)
Step 3: Repeat step 4 and 5 until STACK is empty
Step 4: POP the top node N of STACK . Process N and change the status of N to the
    processed state (STATUS =3)
Step 5: PUSH onto STACK all the neighbours of N that are still in the ready
    state(STATUS=1), and change their status to the waiting state (STATUS=2).
    [End of step 3 loop]
Step 6: Exit
```

Example: Consider the graph G in fig 5.15. suppose we want to find and print all the nodes reachable from the node J(including J itself). One way to do this is to use a depth first search of G starting the node J. The steps of our follows:

(a) Initially , push J onto the stack as follows:

STACK: J

(b) Pop and print the top element J, then push onto stack all the neighbors of J (those that are in the ready state) as follows:

Print J STACK: D,K

(c) Pop and print the top element K, then push onto stack all the neighbors of K (those that are in the ready state) as follows:

Print K STACK: D,E,G

(d) Pop and print the top element G, then push onto stack all the neighbors of G (those that are in the ready state) as follows:

Print G STACK:D,E,C

(e) Pop and print the top element C, then push onto stack all the neighbors of C (those that are in the ready state) as follows:

Print C STACK:D,E,F

(f) Pop and print the top element F, then push onto stack all the neighbors of F (those that are in the ready state) as follows:

Print F STACK:D,E

(g) Pop and print the top element E, then push onto stack all the neighbors of E (those that are in the ready state) as follows:

Print E STACK: D

(h) Pop and print the top element D, then push onto stack all the neighbors of D (those that are in the ready state) as follows:

Print D STACK:

The stack is empty , so the depth first search of G starting at J is now complete.Accordingly, the nodes which were printed,

J,K,G,C,F.E.D

Space Complexity: The space complexity of a depth first search is lower than that of a breadth first search
Time Complexity: The time complexity of depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed .The time complexity can be given as $O(|V|+|E|)$.
Completeness: Depth-first search is said to be complete algorithm .If there is a solution ,depth-first search will find it regardless of the kind of a graph. But in case of an infinite graph,where there is no possible solution, it will diverge.

5.7 Shortest Pah Algorithm

Here we will discussed three different algorithms to calculate the shortest path between the vertices of a graph G.

- Minimum spanning tree
- Dijkstra's algorithm
- Warshall's algorithm

The first two use the adjacency list to find the shortest path , Warshell's algorithm uses an adjacency matrix to do the same.

5.7.1 Minimum spanning tree

A spanning tree of any kind of graph is just a sub-graph G which is a tree that connects all the vertices together. A graph G have many different spanning tree

A minimum spanning tree (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. Means, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree is at a minimum.

Example: Consider the weighted graph G given below , this graph have many distinct spanning tree

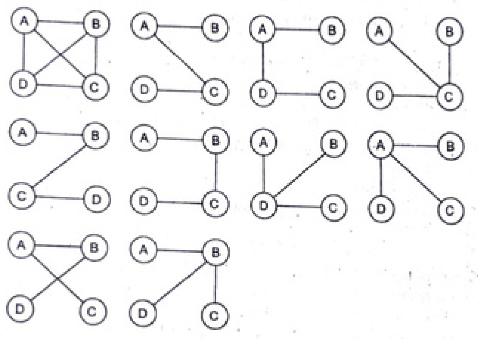


Figure 5.16

5.7.2 Prim's Algorithm

Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph.Means, build a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized. For this, algorithm maintsin three sets of vertices as follow:

Tree vertices: Vertices that are a part of the minimum spanning tree T.
Fringe vertices: Vertices that are currently not a part of T, but are adjacent to some tree vertex.
Unseen vertices: Vertices that are neither tree vertices nor fringe vertices fall under this category.

Algorithm

```
Step1: Select a starting vertex
Step2: Repeat step3 and 4 until there are fringe vertices
Step3: Select an edge e connecting the tree vertex and fringe vertex that has a
       minimum weight.
Step4: Add the selected edge and the vertex to the minimum spanning tree T
Step5: EXIT
```

The running time of Prim's algorithm is O(E log V) where E is the number of edges and V is the number of vertices in the graph.

Example: Consider a minimum spanning tree of the graph given in fig.

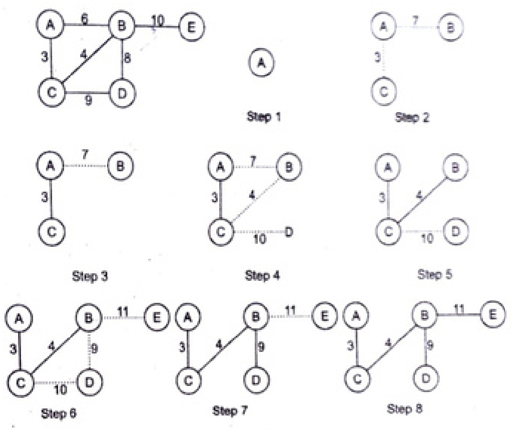


Figure 5.17

Step 1: Choose a starting vertex A.
Step 2: Add the Fringe vertices (that are adjacent to A) . The edges connecting the vertex and fringes vertices are shown are shown with dotted line.
Step 3: Select an edge connecting the tree vertex and the fringe vertex that has the minium weight and add the selected edge and the vertex to the minimum spanning tree T. since the edge connecting A and C has less weight , add C to the tree. Now C is not a fringe vertex but a tree vertex.
Step 4: Add the fringe vertices(that are adjacent to C)
Step 5: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting C and B has less weight , add B to the tree. Now B is not a fringe vertex but a tree vertex.
Step 6: Add the fringe vertices(that are adjacent to B)
Step 7: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting B and D has less weightr , add d to the tree. Now D is not a fringe vertex but a tree vertex.
Step 8: Note , now node E is not connected , so we will add it in the tree because a minimum spanning tree is one in which all the n nodes are connected with n-1 edges that have minimum weight. So, the minimum spanning tree can now be given as.

5.7.3 Kruskal's Algorithm

Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph.The aim of such algorithm is to find a subset of edges which includes every vertex of tree.The total weight of all edges in the tree is minimized.However, if graph is not connected , then it find the minimum spanning forest(forest is a cpllection of trees). A minimum spaaning forest is collection of minimum spanning trees

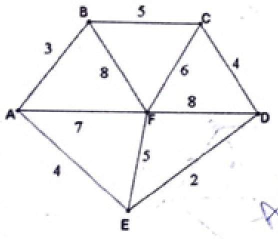
Kruskal's algorithm adopt the greedy approach ,as it makes locally optimal choice at each stage with the hope of finding the global optimum.

Algorithm

```
Step1: create a forest in such a way that each graph is separate tree.
Step2: Create a priority queue Q that contain all the edges of the graph.
Step3: Repeat steps 4 and 5 while Q is not empty
Step4: Remove an edges from Q
Step5: IF the edges obtained in step4 connects two different trees, then Add it
to the forest(for Combining two trees into one tree).
     Else
     Discard the edge
Step 6: END
```

In kruskal's algorithm we use a priority queue Q in which edges that have minimum weight takes a priority over any other edge in the graph.The running time of Kruskal;s algorithm is O(E log V), where E is the number of edges and V is the number of vertices in the graph.

Example: Let us solve the following by applying Kruskal's Algorithm.



AB → BC → CD
→ DE → C[illegible]
9

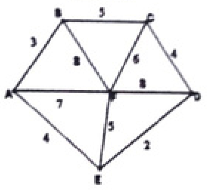
Solution:

The list of edges in order of their weights or sizes would be as follows:

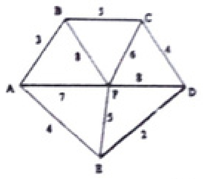
Edge	Weight	Edge	Weight
ED	2	EF	5
AB	3	CF	6
AE	4	AF	7
CD	4	BF	8
BC	5	CF	6

The various iterations would be as follows:

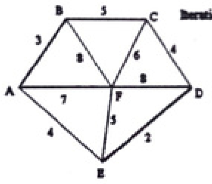
Iteration 0:



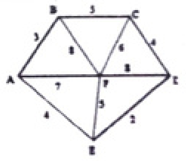
Iteration 1:



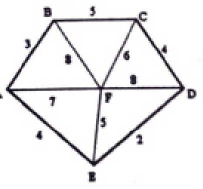
Iteration 2:



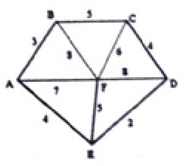
Iteration 3:



Iteration 4:



Iteration 5:



All the vertices have been connected now, hence, the last iteration number 5, gives us the optimal solution, and the minimum length would be the sum of all weights given to these edges, as 2 + 3 + 4 + 4 + 5 = 18 is the length of the shortest path by applying Kruskal's Algorithm.

That is, the solution is

ED 2
AB 3
CD 4
AE 4
EF 5,
with Total weight of tree equal to 18

5.7.4 Dijkstra 's Algorithm

Dijkstra's algorithm ,given by a Dutch scientist Edsger dijkstra in 1959, is used to find the shortest path tree.This algorithm is widely used in network routing protocols,mostly in OSPF(Open Shortest Path First). Dijkstra's algorithm Is used for finding the costs of the shortest paths(one having the lowest cost) from a source node to a destination node.

Dijkstra 's algorithm is used to find the length of an optimal path between two nodes in a graph. The term optimal means, anything,shortest,cheapest,or fastest. If we start the algorithm with an initial node , then the distance of a node Y can be given as the distance from the initial node to that node.

Algorithm

```
1. Select the source node also called the initial nodes.
2. Define an empty set N that will be used to hold nodes to which a shortest path has been found.
3. Label the initial node with 0, and insert it into N.
4. Repeat steps 5 to 7 until the destination node is in N or there are no more labeled nodes in N.
5. Consider each node that is not in N and is connected by an edge from the newly inserted node.
6. (a) If the node that is not in N has no label then SET the label of the node=the label of the newly
       inserted node .
   (b) ELSE if the node is not in N was already labeled ,then SET its new label= minimum (label of
       newly inserted vertex + length of edge , old label)
7. Pick a node not in N that has the smallest label assigned to it and add it to N.
```

Example: Consider the graph G is given in fig 6. . Taking d as initial node, execute tha Djkstra's algorithm on it.

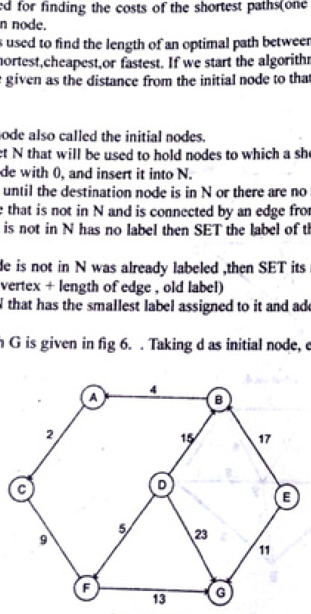


Figure 6.18

Step1: Set the label of D =0 and N={D}

Step2: Label of D=0, B=15 g=23 and F=5. Therefore N={D,F}

[illegible] be re-labelled 18 because minimum (5+13,23)=18 , C has been re-[illegible] 9. Therefore, N={D,F,C}

Step4: Label of D=0,B=15,G=18 .Therefore, N={D,F,C,B,}

Step5: Label of D=0,B=15,G=18 and A=19(15+4) .Therefore, N={D,F,C,B,G}

Step6: Label of D=0 and A=19. Therefore, N={D,f,C,B,G,A}

Note that we have no labels for node E; this means that E is not reachable from D. Only the nodes that *are in N are reachable from B.*

***NOTE:** The running time of Djkstra's algorithm is* O(|V|^2+|E|^2).

Warshell's Algorithm

In a graph is given as G=(E,V). where E is the set of edges and V is the set of vertices . the path matrix of G can be found as, P=A+A^2+A^3+..........+A^n. This is lengthy process. so Warshell has give an efficient algorithm to find the path matrix.

Warshall's algorithm defines matrices as $P_0, P_1, P_2, \ldots, P_n$ as given in fig.

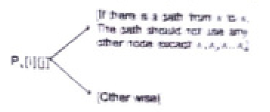

P$_k$[i][j]
[If there is a path from v$_i$ to v$_j$. The path should not use any other node except v$_1$,v$_2$,v$_3$...v$_k$]
[Other wise]

Fig 6.20

- If P_0[i][j]=1 then there exists an edge from node v_i to v_j
- If P_1[i][j]=1, then there exists an edge from v_i to v_j that doesnot use any other vertex except v_1.
- If P_2[i][j]=1, the there exists an edge from v_i to v_j that does not use any other vertex except v_1 and v_2.

Note that P_0 is equal to the adjancency matrix G . If G contains n nodes . then P_n=P which is the path matrix of the graph G.

It is concludes that, P_n[i][j] is equal to 1 only when either of the two following cases:

There is a path from v_i to v_j that does not use any other node except $v_1, v_2, v_3 \ldots v_{k-1}$. Therefore P_{k-1}[i][j]=1.

There is a path from v_i to v_k and a path v_k to v_j where all the nodes use $v_1, v_2, v_3, \ldots, v_{k-1}$. Therefore.

P_{k-1}[i][j]=1 AND P_{k-1}[k][j]=1

Hence, the path matrix P_n can be calculated with the formula given as:

P_k[i][j]=P_{k-1}[i][j] V (P_{k-1}[i][j] Λ P_{k-1}[i][j])

Where V indicates logical **OR** operation and Λ indicates logical **AND** operation.

Write a program to implement Warshell's algorithm to find the path matrix

```
# include <stdio.h>
# include <conio.h>
void read (int mat[5][5], int  n );
void display(int mat[5][5], int  n );
void mul (int mat[5][5], int  n );
void main()
{
  int adj[5][5], P[5][5],n,i,j,k;
  clrscr();
  printf(" \n Enter the number of nodes  in the graph: ");
  scanf(" %d", &n);
  printf(" \n Enter the adjacency matrix :  ");
 read(adj, n);
  clrscr();
  printf(" \n  The adjacency matrix is  :  ");
  display(adj,n)
  for(i=0;i<n;i++)
  {
    for(j=0;j<n, j++);
    {
      if(adj[i][j]==0)
```

```
            P[i][j]=0;
          else
            P[i][j]=1;
        }
      }
      for(k=0; k<n; k++)
      {
        for(i=0;i<n;i++)
        {


          for(j=0;j<n;j++)
          P[i][j]=P[i][j] | ( P[i][k] && P[k][j])

        }
      }
      printf(" \n The Path matrix :");
      display(P,n);
      getch();
      return 0;
  }
  Void read (int mat[5][5], int n)
  {
    int i, j;
    for(i=0;i<n;i++)
    {
      for(j=0;j<n; j++)
      {
        printf("\n mat[%d][%d]=", I,j);
        scanf("%d", &mat[i][j]);
      }
  }
  }
  Void display (int mat[5][5], int n)
  {
    int i,j;
    for(i=0;i<n;i++)
    printf("\n");
      for(j=0;j<n;j++)
        printf("%d\t", mat[i][j]);
      }
  }
```

Algorithm

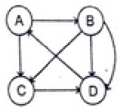
```
Step 1:   [Initialize the Path Matrix]  Repeat step2 for I=0 to n-1
          where n is the number of nodes in the graph.
Step 2:   Repeat step 3 for J=0 to n-1
Step 3:   IF A[I][J]=0, then SET P[I][J]=0
          ELSE P[I][J]=1
          [END OF LOOP]
          [END OF LOOP]
```

```
Step 4:   [Calculate the path matrix P] Repeat step 5 for k=0 to n-1
Step 5:   Repeat step 6 for I=0 to n-1
Step 6:   Repeat step 7 for J=0 to n-1
Step 7:   SET $P_k[I][J]=P_{k-1}[I][J] \vee (P_{k-1}[I][K] \wedge P_{k-1}[K][J])$
Step 8:   EXIT
```

Example: Consider the graph in Fig6. And its adjancency matrix A. we can straightway calculate the path matrix P using the Warshell's algorithm

The Path matrix P can be given in a single step as:

$$P = \begin{array}{c|cccc} & A & A & A & A \\ \hline A & 1 & 1 & 1 & 1 \\ B & 1 & 1 & 1 & 1 \\ C & 1 & 1 & 1 & 1 \\ D & 1 & 1 & 1 & 1 \end{array}$$



$$\begin{array}{c|cccc} & A & A & A & A \\ \hline A & 0 & 1 & 1 & 1 \\ B & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 \end{array}$$

Figure 5.19

Thus we see, that calculating A_1, A_2, A3.......A_5 to calculate P is very slow and inefficient technique as compared to warshell's technique.

■ ■ ■

Very Short Questions

1. Define Graph.
2. What do you understand by simple graph?
3. What is connected Graph?
4. What is regular graph?
5. Define weighted and unweighted graph
6. Define directed graph.
7. What is Adjacency List?
8. What do you understand by Traversal methods of graph?
9. Define shortest path algorithm and why they are use?
10. What do you understand by spanning tree?

Short Questions

1. How are graphs are presented inside a computer?
2. Draw a complete undirected graph having five nodes.
3. Consider the graph given below and find out the degree of each node.
4. Explain the graph traversal algorithms in detail with example.
5. Write a short notes on
   (a) Prim's algorithm   (b) Kruskal's algorithm

Long Questions

1. Explain all method of finding the shortest path with an example of each.
2. Explain the graph representation techniques used by computer.
3. Explain Orthogonal representation of graph
4. Discuss the running time complexity of all shortest find algorithms
5. Differentiatr between depth-first search and breadth-first search traversal of a graph.