UNIT-V

GRAPHS

1. BASIC CONCEPTS

INTRODUCTION

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a 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.

WHY GRAPHS ARE USEFUL

Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:

- *Family trees:* in which the member nodes have an edge from parent to each of their children.
- *Transportation networks :* in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

DEFINATION:

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.

We have two types of Graphs. Basically:

1. UNDIRECTED GRAPH
2. DIRECTED GRAPH

UNDIRECTED GRAPH**:**

Shows a graph with V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D,E), (C, E)}. Note that there are five vertices or nodes and six edges in the graph.

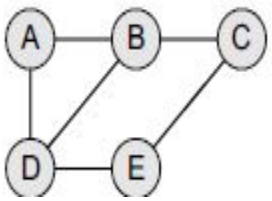


FIGURE 5.1

A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. Figure 5.1 shows an undirected graph because it does not give any information about the direction of the edges.

DIRECTED GRAPH:

A directed graph G, also known as a *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),

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

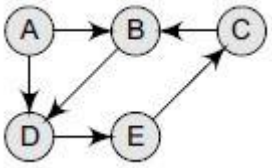


FIGURE 5.2

Which shows a directed graph. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

2. REPRESENTATION OF GRAPHS

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

- ***Sequential representation*** by using an adjacency matrix.
- ***Linked representation*** by using an adjacency list that stores the neighbours of a node using a linked list.

2.1 ADJACENCY MATRIX REPRESENTATION

An adjacency matrix is used to represent which nodes are adjacent to one another.

By definition: 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 adjacency matrix will have the dimension of n X n.

In an adjacency matrix, the rows and columns are labelled 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.
- However, if the nodes are not adjacent, a_{ij} will be set to zero.

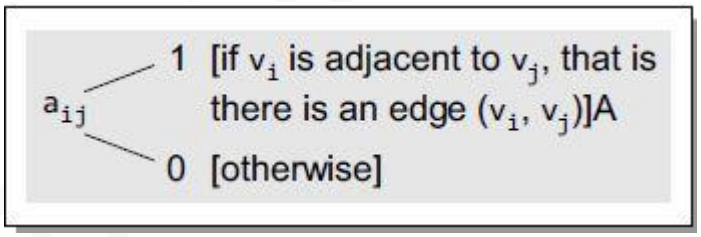


FIGURE 5.3 Adjacency Matrix Entry

Since an adjacency matrix contains only 0s and 1s, it is called a *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.

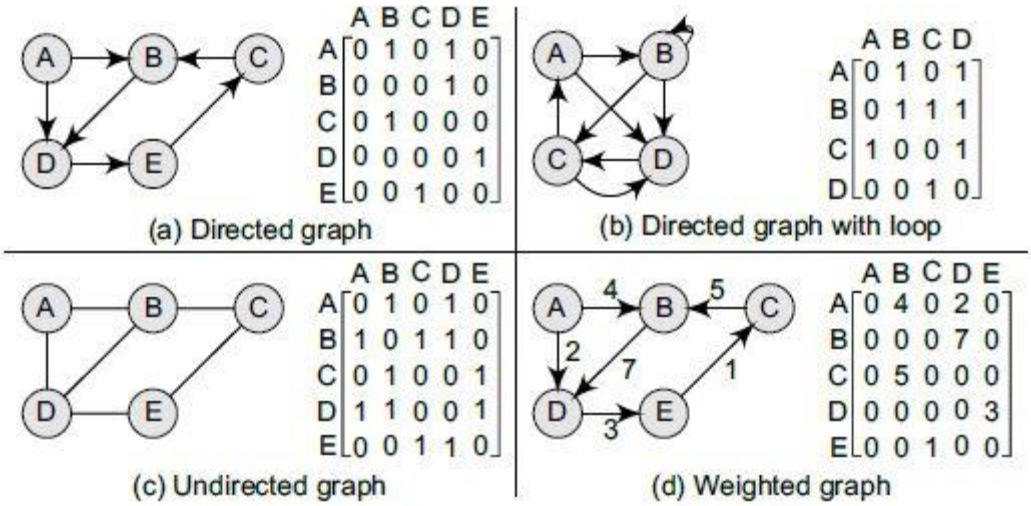


Figure 5.4 shows some graphs and their corresponding adjacency matrices.

From the above examples, we can draw the following conclusions:

- 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(n2), where n is the number of nodes in the graph.
- Number 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.

Now let us discuss the powers of an adjacency matrix:

From adjacency matrix A1, we can conclude that an entry 1 in the ith row and jth column means that there exists a path of length 1 from v_i to v_j. Now consider, A2, A3, and A4.

Any entry $a_{ij} = 1$ if $a_{ik} = a_{kj} = 1$. That is, if there is an edge (v_i, v_k) and (v_k, v_j), then there is a path from vi to vj of length 2.

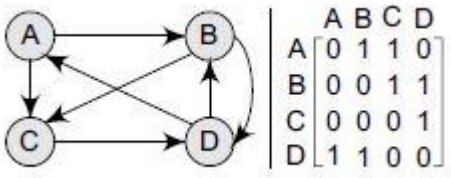


FIGURE 5.5 Directed graph with its adjacency matrix

$$A^2 = A^1 \times A^1$$

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$

Now, based on the above calculations, we define matrix B as:

$B_r = A_1 + A_2 + A_3 + \ldots + A_r$

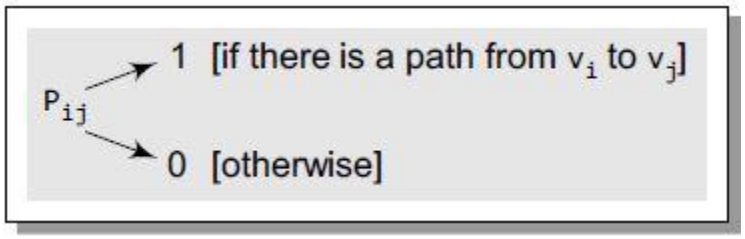


FIGURE 5.6 Path Matrix Entry

The main goal to define matrix B is to obtain the path matrix P. The path matrix P can be calculated from B by setting an entry Pij = 1, if Bij is non-zero and Pij = 0, if otherwise. The path matrix is used to show whether there exists a simple path from node vi to vj or not.

Let us now calculate matrix B and matrix P using the above discussion.

$$\begin{bmatrix}0&1&1&0\\0&0&1&1\\0&0&0&1\\1&1&0&0\end{bmatrix}+\begin{bmatrix}0&0&1&2\\1&1&0&1\\1&1&0&0\\1&1&2&1\end{bmatrix}+\begin{bmatrix}2&2&0&1\\1&2&2&1\\0&1&2&1\\1&2&2&3\end{bmatrix}+\begin{bmatrix}1&3&4&2\\1&2&3&4\\1&1&1&3\\3&4&3&4\end{bmatrix}=\begin{bmatrix}3&6&6&5\\3&5&6&7\\2&3&3&5\\6&8&7&8\end{bmatrix}$$

B

Now the path matrix P can be given as:

$$P=\begin{bmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{bmatrix}$$

2.2 ADJACENCY LINKED LIST REPRESEENTATION

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

The key advantages 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 number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

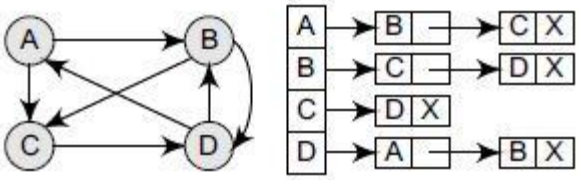


FIGURE 5.7 Graph G and its adjacency list

- For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G.
- However, for an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u.
- Adjacency lists can also be modified to store weighted graphs.

Let us now see an adjacency list for an undirected graph as well as a weighted graph.

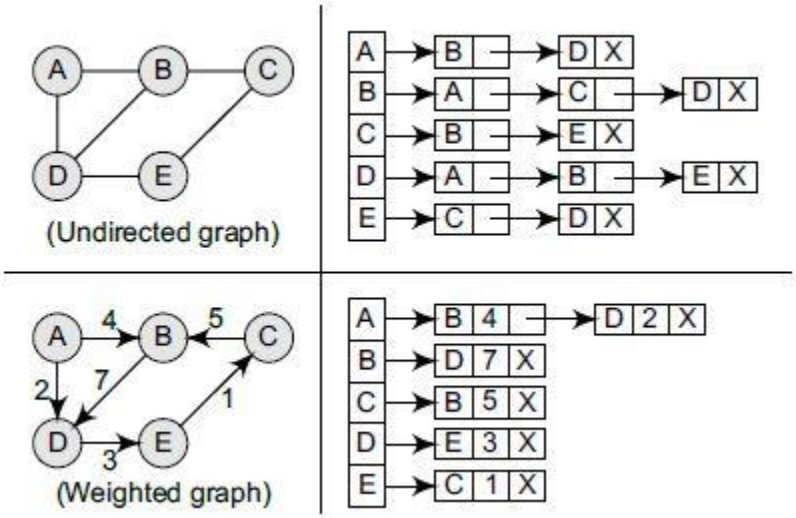


FIGURE 5.8 Adjacency list for an undirected graph and a weighted graph

PROGRAMMING EXAMPLE

1. Write a program to create a graph of *n* vertices using an adjacency list. Also write the code to read and print its information and finally to delete the graph.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct node
{
char vertex;
struct node *next;
};
struct node *gnode;
void displayGraph(struct node *adj[], int no_of_nodes);
void deleteGraph(struct node *adj[], int no_of_nodes);
void createGraph(struct node *adj[], int no_of_nodes);
int main()
{
struct node *Adj[10];
int i, no_of_nodes;
clrscr();
printf("\n Enter the number of nodes in G: ");
scanf("%d", &no_of_nodes);
for(i = 0; i < no_of_nodes; i++)
Adj[i] = NULL;
createGraph(Adj, no_of_nodes);
printf("\n The graph is: ");
displayGraph(Adj, no_of_nodes);
deleteGraph(Adj, no_of_nodes);
getch();
return 0;
}
```

```
void createGraph(struct node *Adj[], int no_of_nodes)
{
struct node *new_node, *last;
int i, j, n, val;
for(i = 0; i < no_of_nodes; i++)
{
last = NULL;
printf("\n Enter the number of neighbours of %d: ", i);
scanf("%d", &n);
for(j = 1; j <= n; j++)
{
printf("\n Enter the neighbour %d of %d: ", j, i);
scanf("%d", &val);
new_node = (struct node *) malloc(sizeof(struct node));
new_node –> vertex = val;
new_node –> next = NULL;
if (Adj[i] == NULL)
Adj[i] = new_node;
else
last –> next = new_node;
last = new_node
}
}
}
void displayGraph (struct node *Adj[], int no_of_nodes)
```



```
{
struct node *ptr;
int i;
for(i = 0; i < no_of_nodes; i++)
{
```

```
ptr = Adj[i];
printf("\n The neighbours of node %d are:", i);
while(ptr != NULL)
{
printf("\t%d", ptr –> vertex);
ptr = ptr –> next;
}
}
}
void deleteGraph (struct node *Adj[], int no_of_nodes)
{
int i;
struct node *temp, *ptr;
for(i = 0; i <= no_of_nodes; i++)
{
ptr = Adj[i];
while(ptr ! = NULL)
{
temp = ptr;
ptr = ptr –> next;
free(temp);
}
Adj[i] = NULL;
}
}
```

Output

```
Enter the number of nodes in G: 3
Enter the number of neighbours of 0: 1
Enter the neighbour 1 of 0: 2
Enter the number of neighbours of 1: 2
Enter the neighbour 1 of 1: 0
```

Enter the neighbour 2 of 1: 2

Enter the number of neighbours of 2: 1

Enter the neighbour 1 of 2: 1

The neighbours of node 0 are: 1

The neighbours of node 1 are: 0 2

The neighbours of node 2 are: 0

Note If the graph in the above program had been a weighted graph, then the structure of the node would have been:

```
typedef struct node
{
int vertex;
int weight;
struct node *next;
};
```

3.GRAPH TRAVERSALS

There are two standard methods of graph traversal. These two methods are:

1. Breadth-first search
2. Depth-first search

1.Breadth-First Search Algorithm

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.

ALGORITHM

Step 1: SET STATUS = 1 (ready state)

for each node in G

Step 2: Enqueue the starting node A

and set its STATUS = 2

(waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it
and set its STATUS = 3
(processed state).
Step 5: Enqueue all the neighbours of
N that are in the ready state
(whose STATUS = 1) and set
their STATUS = 2
(waiting state)
[END OF LOOP]
Step 6: EXIT

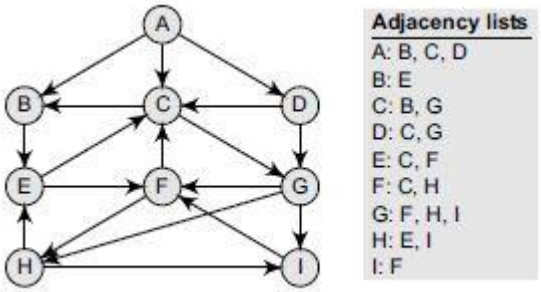


FIGURE 5.9 Graph G And Its Adjacnecy List

EXAMPLE

Consider the graph G given in Fig. 5.9.The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1**.**

SOLUTION:

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays:

1. QUEUE
2. ORIG

- While QUEUE is used to hold the nodes that have to be processed,
- ORIG is used to keep track of the origin of each edge.
- Initially, FRONT = REAR = –1.

The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0 QUEUE = A

REAR = 0 ORIG = \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1 QUEUE = A B C D

REAR = 3 ORIG = \0 A A A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2 QUEUE = A B C D E

REAR = 4 ORIG = \0 A A A B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3 QUEUE = A B C D E G

REAR = 5 ORIG = \0 A A A B C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4 QUEUE = A B C D E G

REAR = 5 ORIG = \0 A A A B C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5 QUEUE = A B C D E G F

REAR = 6 ORIG = \0 A A A B C E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6 QUEUE = A B C D E G F H I

REAR = 9 ORIG = \0 A A A B C E G G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have
P as A -> C -> G -> I.

Features of Breadth-First Search Algorithm

Space complexity:

The space complexity is therefore proportional to the number of nodes at the deepest level of the graph.

Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level $O(^{bd})$.

The space complexity can also be expressed as O (| E | + | V |), where | E | is the total number of edges in G and | V | is the number of nodes or vertices.

Time Complexity:

In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches $O(^{bd})$.

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 a 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 the start node and the goal node.

we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

Programming Example

2. Write a program to implement the breadth-first search algorithm.

```
#include <stdio.h>

#define MAX 10

void breadth_first_search(int adj[][MAX],int visited[],int start)

{

int queue[MAX],rear = –1,front =– 1, i;

queue[++rear] = start;

visited[start] = 1;

while(rear != front)

{

start = queue[++front];

if(start == 4)

printf("5\t");

else

printf("%c \t",start + 65);

for(i = 0; i < MAX; i++)

{
```

```
if(adj[start][i] == 1 && visited[i] == 0)
{
queue[++rear] = i;
visited[i] = 1;
}
}
}
}
int main()
{
int visited[MAX] = {0};
int adj[MAX][MAX], i, j;
printf("\n Enter the adjacency matrix: ");
for(i = 0; i < MAX; i++)
for(j = 0; j < MAX; j++)
scanf("%d", &adj[i][j]);
breadth_first_search(adj,visited,0);
return 0;
}
```

Output

```
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
A B D C E
```

2. Depth First Algorithm

The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.

When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set
its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its
STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that
are in the ready state (whose STATUS = 1) and
set their STATUS = 2 (waiting state)
[END OF LOOP]

Step 6: EXIT

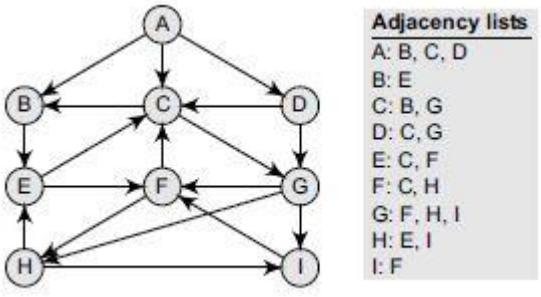


FIGURE 5.10 Graph G And Its Adjacency List

Example:

Consider the graph G given in. The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H. The procedure can be explained here.

Solution:

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B STACK: E

h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

Features of Depth-First Search Algorithm

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 a 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 a complete algorithm. If there is a solution, depthfirst 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.

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

Programming Example:

3. Write a program to implement the depth-first search algorithm.

```
#include <stdio.h>
#define MAX 5
void depth_first_search(int adj[][MAX],int visited[],int start)
{
int stack[MAX];
int top = –1, i;
printf("%c–",start + 65);
visited[start] = 1;
stack[++top] = start;
while(top ! = –1)
{
start = stack[top];
for(i = 0; i < MAX; i++)
{
if(adj[start][i] && visited[i] == 0)
{
stack[++top] = i;
printf("%c–", i + 65);
visited[i] = 1;
break;
}
}
if(i == MAX)
top—;
}
}
int main()
{
int adj[MAX][MAX];
```

```
int visited[MAX] = {0}, i, j;
```



```
printf("\n Enter the adjacency matrix: ");
for(i = 0; i < MAX; i++)
for(j = 0; j < MAX; j++)
scanf("%d", &adj[i][j]);
printf("DFS Traversal: ");
depth_first_search(adj,visited,0);
printf("\n");
return 0;
}
```

Output

```
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
DFS Traversal: A –> C –> E –>
```

APPLICATIONS

MINIMUM SPANNING TREES:

- A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together
- A graph G can have many different spanning trees.
- We can assign *weights* to each edge (which is a number that represents how unfavourable the edge is), and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that 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. In other words, a minimum spanning

tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

Example: Consider an unweighted graph G given below (Fig. 5.11). From G, we can draw many distinct spanning trees. Eight of them are given here. For an unweighted graph, every spanning tree is a minimum spanning tree.

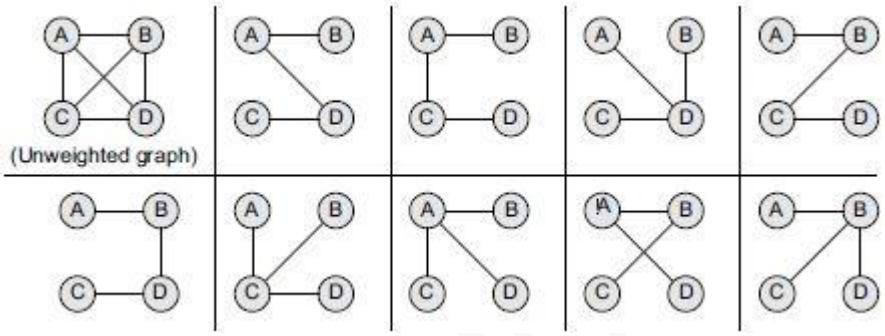


FIGURE 5.11 Unweighted Graph And Its Spanning Trees

EXAMPLE: Consider a weighted graph G shown in Fig. 5.12. From G, we can draw three distinct spanning trees. But only a single minimum spanning tree can be obtained, that is, the one that has the minimum weight (cost) associated with it. Of all the spanning trees given in Fig. 5.12, the one that is highlighted is called the minimum spanning tree, as it has the lowest cost associated with it.

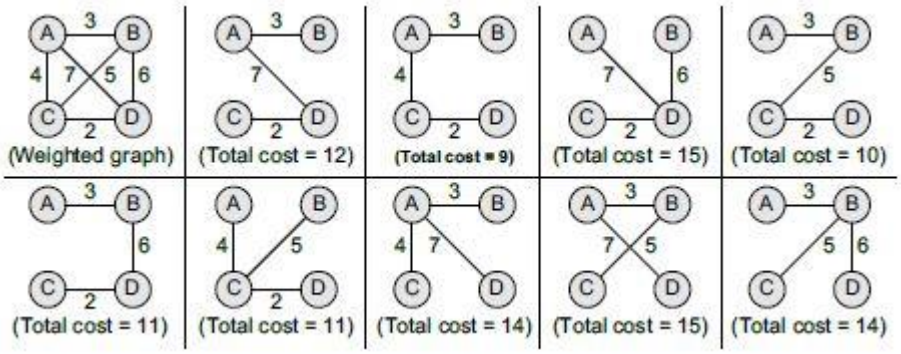


FIGURE 5.12 Weighted Graph And Its Spanning Trees.

APPLICATIONS FOR MINIMUM SPANNING TREES:

- **MST'S** is widely used for designing networks.
- **MST'S** are used to find airlane routes.
- **MST'S** are also used to find the cheapest way to connect terminals, such as cities, electronic components or computers via roads, airlines, railways, wires or telephone lines.
- **MST'S** are applied in routing algorithms for finding the most efficient path.

We have two types of ALGORITHMS in Minimum Spanning Trees. They are:

1. PRIM'S ALGORITHM
2. KRUSKAL'S ALGORITHM

1.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.
- In other words, the algorithm builds 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, the algorithm maintains three sets of vertices which can be given as below:

- **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

Step 1: Select a starting vertex

Step 2: Repeat Steps 3 and 4 until there are fringe vertices

Step 3: Select an edge e connecting the tree vertex and
fringe vertex that has minimum weight

Step 4: Add the selected edge and the vertex to the
minimum spanning tree T
[END OF LOOP]

Step 5: EXIT

EXAMPLE: Construct a minimum spanning tree of the graph given in Fig. 5.13

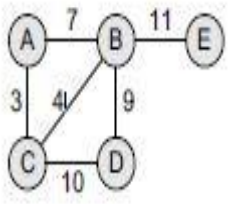


FIGURE 5.13

Step 1: Choose a starting vertex A.

Step 2: Add the fringe vertices (that are adjacent to A). The edges connecting the vertex and fringe vertices are shown with dotted lines.

Step 3: Select an edge connecting the tree vertex and the fringe vertex that has the minimum 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.

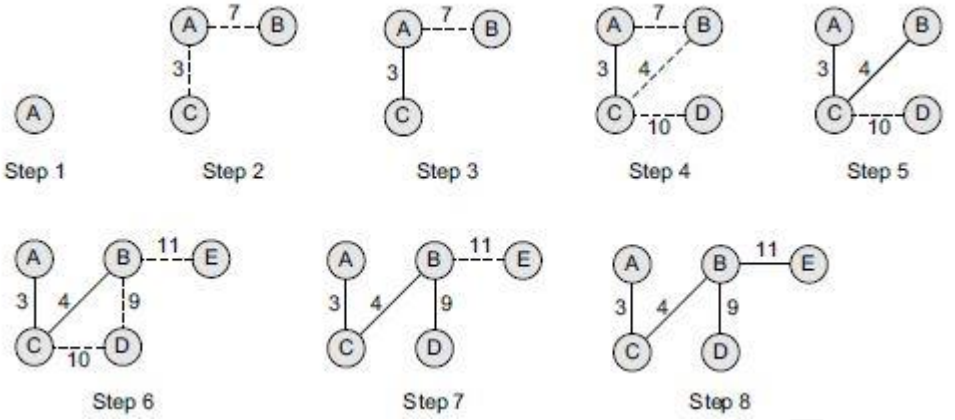
Step 4: Add the fringe vertices (that are adjacent to C).

Step 5: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add 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 add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting B and D has less weight, 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,



2.KRUSKAL'S ALGORITHM

- Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph.
- The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized.
- However, if the graph is not connected, then it finds a *minimum spanning forest*. Note that a forest is a collection of trees. Similarly, a minimum spanning forest is a collection of minimum spanning trees.
- Kruskal's algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum.

ALGORITHM

Step 1: Create a forest in such a way that each graph is a separate tree.

Step 2: Create a priority queue Q that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY

Step 4: Remove an edge from Q

Step 5: IF the edge obtained in Step 4 connects two different trees,
then Add it to the forest (for combining two trees into one tree).
ELSE
Discard the edge

Step 6: END

EXAMPLE: Apply Kruskal's algorithm on the graph given in Fig. 5.14.

Initially, we have F = {{A}, {B}, {C}, {D}, {E}, {F}}

MST = {}

Q = {(A, D), (E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

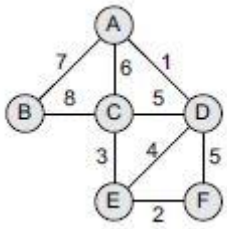
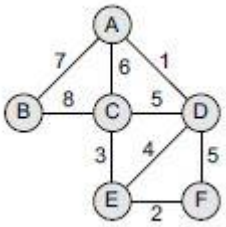


FIGURE 5.14

Step 1: Remove the edge (A, D) from Q and make the following changes:

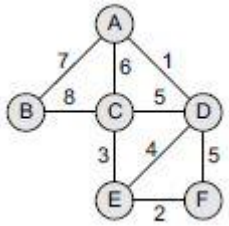


F = {{A, D}, {B}, {C}, {E}, {F}}

MST = {A, D}

Q = {(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 2: Remove the edge (E, F) from Q and make the following changes:

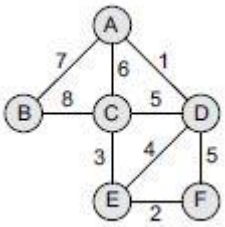


F = {{A, D}, {B}, {C}, {E, F}}

MST = {(A, D), (E, F)}

Q = {(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 3*:* Remove the edge (C, E) from Q and make the following changes:

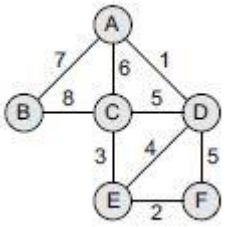


F = {{A, D}, {B}, {C, E, F}}

MST = {(A, D), (C, E), (E, F)}

Q = {(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

Step 4: Remove the edge (E, D) from Q and make the following changes:



F = {{A, C, D, E, F}, {B}}

MST = {(A, D), (C, E), (E, F), (E, D)}

Q= {(C, D), (D, F), (A, C), (A, B), (B, C)}

Step 5: Remove the edge (C, D) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST. Therefore,

F = {{A, C, D, E, F}, {B}}

MST = {(A, D), (C, E), (E, F), (E, D)}

Q = {(D, F), (A, C), (A, B), (B, C)}

Step 6: Remove the edge (D, F) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

F = {{A, C, D, E, F}, {B}}

MST = {(A, D), (C, E), (E, F), (E, D)}

Q = {(A, C), (A, B), (B, C)}

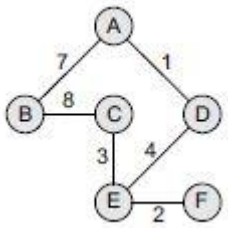
Step 7: Remove the edge (A, C) from Q. Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

F = {{A, C, D, E, F}, {B}}

MST = {(A, D), (C, E), (E, F), (E, D)}

Q = {(A, B), (B, C)}

Step 8: Remove the edge (A, B) from Q and make the following changes:

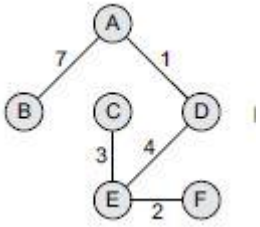


F = {A, B, C, D, E, F}

MST = {(A, D), (C, E), (E, F), (E,D), (A, B)}

Q = {(B, C)}

Step 9: The algorithm continues until Q is empty. Since the entire forest has become one tree, all the remaining edges will simply be discarded. The resultant MS can be given as shown below



F = {A, B, C, D, E, F}

MST = {(A, D), (C, E), (E, F), (E,D), (A, B)}

Q = {}

PROGRAMMING EXAMPLE:

5. Write a program which finds the cost of a minimum spanning tree.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
int adj[MAX][MAX], tree[MAX][MAX], n;
void readmatrix()
{
int i, j;
printf("\n Enter the number of nodes in the Graph : ");
scanf("%d", &n);
printf("\n Enter the adjacency matrix of the Graph");
for (i = 1; i <= n; i++)
for (j = 1; j <= n; j++)
scanf("%d", &adj[i][j]);
}
int spanningtree(int src)
{
```

```
int visited[MAX], d[MAX], parent[MAX];
int i, j, k, min, u, v, cost;
for (i = 1; i <= n; i++)
{
d[i] = adj[src][i];
visited[i] = 0;
parent[i] = src;
}
visited[src] = 1;
cost = 0;
k = 1;
for (i = 1; i < n; i++)
{
min = 9999;
for (j = 1; j <= n; j++)
{
if (visited[j]==0 && d[j] < min)
{
min = d[j];
u = j;
cost += d[u];
}
}
visited[u] = 1;
//cost = cost + d[u];
tree[k][1] = parent[u];
tree[k++][2] = u;
for (v = 1; v <= n; v++)
if (visited[v]==0 && (adj[u][v] < d[v]))
{
d[v] = adj[u][v];
```

```
parent[v] = u;
}
}
return cost;
}
void display(int cost)
{
int i;
printf("\n The Edges of the Mininum Spanning Tree are");
for (i = 1; i < n; i++)
printf(" %d %d \n", tree[i][1], tree[i][2]);
printf("\n The Total cost of the Minimum Spanning Tree is : %d", cost);
}
main()
{
int source, treecost;
readmatrix();
printf("\n Enter the Source : ");
scanf("%d", &source);
treecost = spanningtree(source);
display(treecost);
return 0;
}
```

Output

Enter the number of nodes in the Graph : 4

Enter the adjacency matrix : 0 1 1 0

0 0 0 1

0 1 0 0

1 0 1 0

Enter the source : 1

The edges of the Minimum Spanning Tree are 1 4

4 2

2 3

The total cost of the Minimum Spanning Tree is : 1

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, most notably IS-IS and OSPF (Open Shortest Path First).

Given a graph G and a source node A, the algorithm is used to find the shortest path (one having the lowest cost) between A (source node) and every other node. Moreover, Dijkstra's algorithm is also used for finding the costs of the shortest paths from a source node to a destination node.

For example, if we draw a graph in which nodes represent the cities and weighted edges represent the driving distances between pairs of cities connected by a direct road, then Dijkstra's algorithm when applied gives the shortest route between one city and all other cities.

ALGORITHM

- Dijkstra's algorithm is used to find the length of an *optimal* path between two nodes in a graph.
- The term *optimal* can mean 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.

1. Select the source node also called the initial node
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 , and insert it into N.
4. Repeat Steps 5 to 7 until the destination node is in N or there are no more labelled 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 + the length of the edge.
(b) Else if the node that is not in N was already labelled, 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.

Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node.
There are two kinds of labels: **temporary** and ***permanent*.**
Temporary labels are assigned to nodes that have not been reached, while permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known. A node must be a permanent label or a temporary label, but not both.
The execution of this algorithm will produce either of the following two results:
1. If the destination node is labelled, then the label will in turn represent the distance from the source node to the destination node.
2. If the destination node is not labelled, then there is no path from the source to the destination node.

EXAMPLE:
Consider the graph G given in Fig. 5.14. Taking D as the initial node, execute the Dijkstra's algorithm on it.

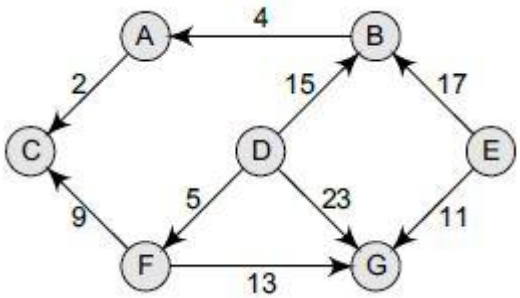


FIGURE 5.14

Step 1: Set the label of D = 0 and N = {D}.

Step 2: Label of D = 0, B = 15, G = 23, and F = 5. Therefore, N = {D, F}.

Step 3: Label of D = 0, B = 15, G has been re-labelled 18 because minimum (5 + 13, 23) = 18, C has been re-labelled 14 (5 + 9). Therefore, N = {D, F, C}.

Step 4: Label of D = 0, B = 15, G = 18. Therefore, N = {D, F, C, B}.

Step 5: Label of D = 0, B = 15, G = 18 and A = 19 (15 + 4). Therefore, N = {D, F, C, B, G}.

Step 6: Label of D = 0 and A = 19. Therefore, N = {D, F, C, B, G, A}

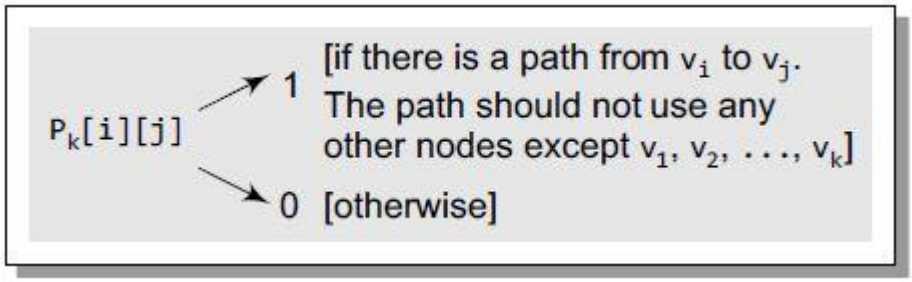
Note that we have no labels for node E; this means that E is not reachable from D. Only the nodes that are in N are reachable from B.

The running time of Dijkstra's algorithm can be given as O(|V|2+|E|)=O(|V|2) where V is the set of vertices and E in the graph.

Warshall's Algorithm

If a graph G is given as G=(V, E), where V is the set of vertices and E is the set of edges, the path matrix of G can be found as, P = A + A2 + A3 + ... + An.

This is a lengthy process, so Warshall has given a very efficient algorithm to calculate the path matrix. Warshall's algorithm defines matrices P0, P1, P2, º, Pn.



Path Matrix Entry

- This means that if P0[i][j] = 1, then there exists an edge from node vi to vj.
- If P1[i][j] = 1, then there exists an edge from vi to vj that does not use any other vertex except v1.

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

  Pk[i][j] = Pk–1[i][j] V (Pk–1[i][k] ^ Pk–1[k][j])

  where **V** indicates logical OR operation and ^ indicates logical AND operation.

ALGORITHM

Step 1: [the Path Matrix] Repeat Step 2 for I = to n-1,

where n is the number of nodes in the graph

Step 2: Repeat Step 3 for J = to n-1

Step 3: IF A[I][J] = , then SET P[I][J] =

ELSE P[I][J] = 1

[END OF LOOP]

[END OF LOOP]

Step 4: [Calculate the path matrix P] Repeat Step 5 for K = to n-1

Step 5: Repeat Step 6 for I = to n-1

Step 6: Repeat Step 7 for J= to n-1

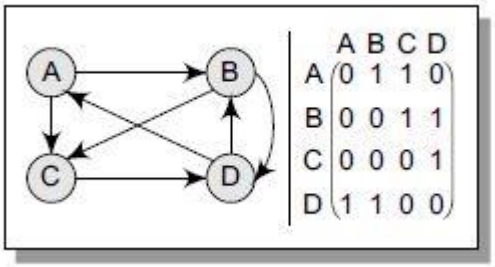
Step 7: SET P [I][J] = P [I][J] V (P [I][K]

P [K][J])

Step 8: EXIT

EXAMPLE:

Consider the graph in Fig. 13.39 and its adjacency matrix A. We can straightaway calculate the path matrix P using the Warshall's algorithm. The path matrix P can be given in a single step as:



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

GRAPH G AND ITS PATH MATRIX

PROGRAMMING EXAMPLE

6. Write a program to implement Warshall'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);
int 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 is :");
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]);
}
}
```

Output

The adjacency matrix is

0 1 1 0

0 0 1 1

0 0 0 1

1 1 0 0

The Path Matrix is

1 1 1 1

1 1 1 1

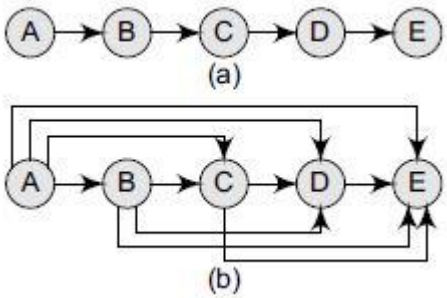
1 1 1 1

1 1 1 1

Transitive Closure of a Directed Graph

Definition

For a directed graph G = (V,E), where V is the 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 from v to w in G.



(a) A graph G and its
(b) transitive closure G*

Where and Why is it Needed?

Finding the transitive closure of a directed graph is an important problem in the following computational tasks:

- Transitive closure is used to find the reachability analysis of transition networks representing distributed and parallel systems.I
- It is used in the construction of parsing automata in compiler construction.
- Recently, transitive closure computation is being used to evaluate recursive database queries (because almost all practical recursive queries are transitive in nature).

ALGORITHM

Transitive_Closure(A, t, n)

Step 1: SET i=1, j=1, k=1

Step 2: Repeat Steps 3 and 4 while i<=n

Step 3: Repeat Step 4 while j<=n

Step 4: IF (A[i][j] = 1)

SET t[i][j] = 1

ELSE

SET t[i][j] =

INCREMENT j

[END OF LOOP]

INCREMENT i

[END OF LOOP]

Step 5: Repeat Steps 6 to 11 while k<=n

Step 6: Repeat Steps 7 to 1 while i<=n

Step 7: Repeat Steps 8 and 9 while j<=n

Step 8: SET t[i,j] = t[i][j] V (t[i][k] t[k][j])

Step 9: INCREMENT j

[END OF LOOP]

Step 10 : INCREMENT i

[END OF LOOP]

Step 11: INCREMENT k

[END OF LOOP]

Step 12: END