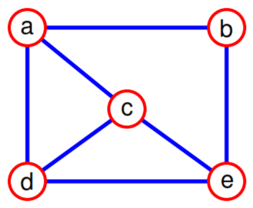
UNIT-V
GRAPHS

Graph: A **graph G** = (**V**,**E**) is composed of:
V: set of *vertices*
E: set of *edges* connecting the *vertices* in **V**
• An **edge e** = (u,v) is a pair of vertices

Example:



V= {a,b,c,d,e}
E={(a,b),(a,c),(a,d),(b,e),(c,d),(c,e),(d,e)}

Graph Terminology

Undirected Graph:
An undirected graph is one in which the pair of vertices in a edge is unordered, (v0, v1) = (v1,v0)

Directed Graph:
A directed graph is one in which each edge is a directed pair of vertices, <v0, v1> != <v1,v0>

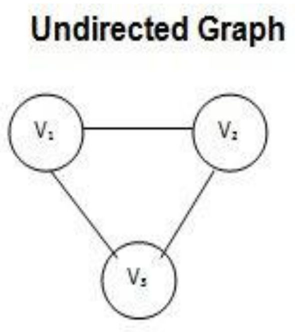


Figure 1: An Undirected Graph

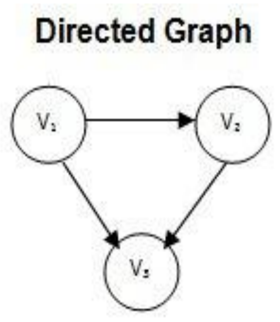
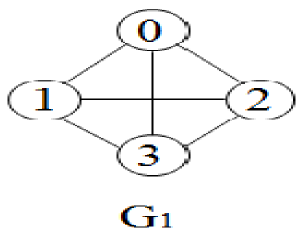


Figure 2: A Directed Graph

Complete Graph:
A complete graph is a graph that has the maximum number of edges for undirected graph with n vertices, the maximum number of edges is n(n-1)/2 for directed graph with n vertices, the maximum number of edges is n(n-1)

example: G1 is a complete graph



complete graph

$V(G_1)=\{0,1,2,3\}$
$V(G_2)=\{0,1,2,3,4,5,6\}$
$V(G_3)=\{0,1,2\}$

Adjacent and Incident:

If (v0, v1) is an edge in an undirected graph,

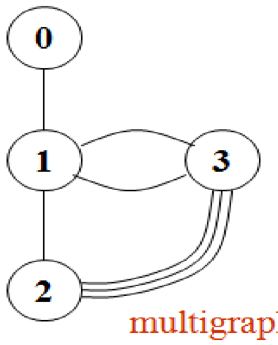
- v0 and v1 are adjacent
- The edge (v0, v1) is incident on vertices v0 and v1

If <v0, v1> is an edge in a directed graph

- v0 is adjacent to v1, and v1 is adjacent from v0
- The edge <v0, v1> is incident on v0 and v1

Multigraph:

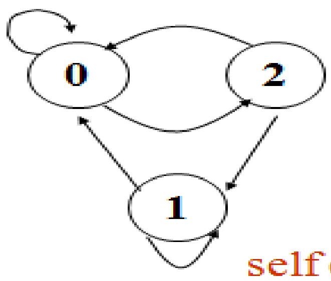
In a multigraph, there can be more than one edge from vertex P to vertex Q. In a simple graph there is at most one.



multigraph:
multiple occurrences
of the same edge

Graph with self edge or graph with feedback loops:

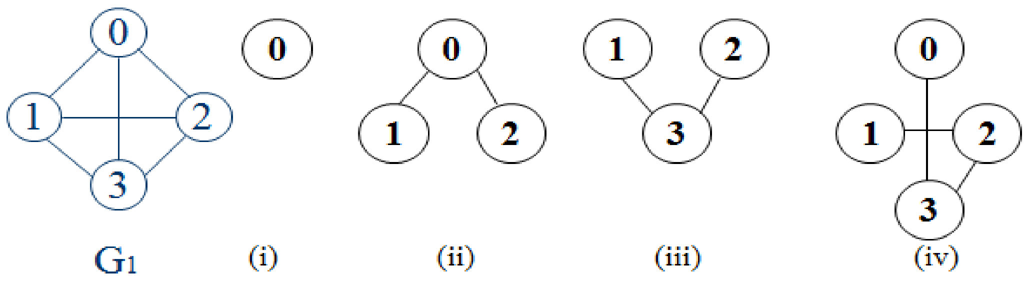
A self loop is an edge that connects a vertex to itself. In some graph it makes sense to allow self-loops; in some it doesn't.



self edge

Subgraph:

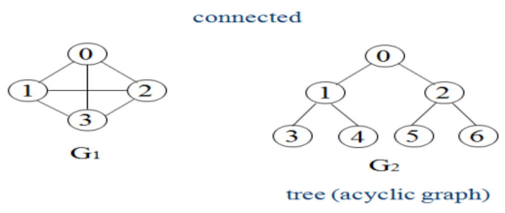
A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G)



Some of the subgraph of G_1

Path:
A path from vertex vp to vertex vq in a graph G, is a sequence of vertices, vp, vi1, vi2, ..., vin, vq, such that (vp, vi1), (vi1, vi2), ..., (vin, vq) are edges in an undirected graph
The length of a path is the number of edges on it.

Simple Path and Style:
A simple path is a path in which all vertices, except possibly the first and the last, are distinct.
A cycle is a simple path in which the first and the last vertices are the same
In an undirected graph G, two vertices, v0 and v1, are connected if there is a path in G from v0 to v1.
An undirected graph is connected if, for every pair of distinct vertices vi, vj, there is a path from vi to vj

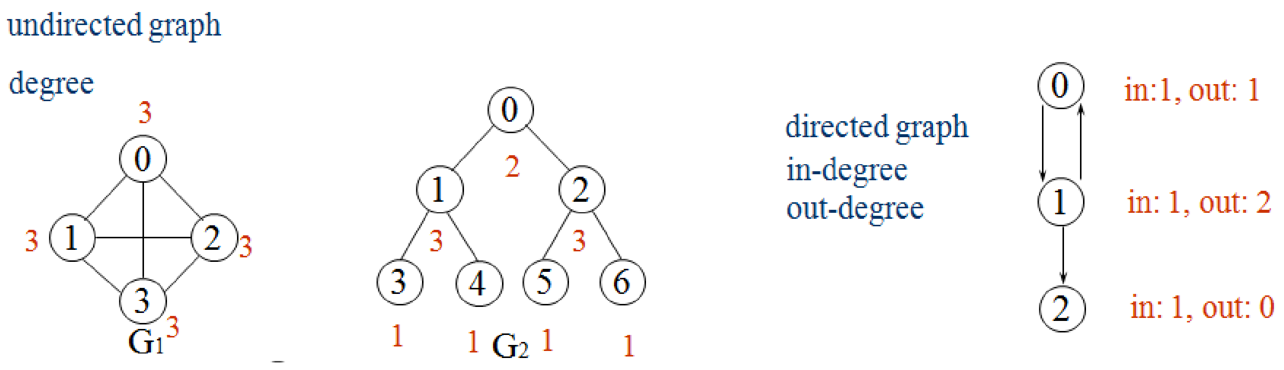


Degree
The degree of a vertex is the number of edges incident to that vertex

For directed graph,
- the ***in-degree*** of a vertex *v* is the number of edges that have *v* as the head
- the ***out-degree*** of a vertex *v* is the number of edges that have *v* as the tail
- if *di* is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

Example:



ADT for Graph

Graph ADT is

Data structures: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

Functions: for all *graph* ∈ *Graph*, *v*, v_1 and v_2 ∈ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *V* has no incident edge.

Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

Graph Representations

Graph can be represented in the following ways:

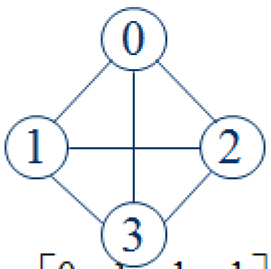
a) Adjacency Matrix
b) Adjacency Lists
c) Adjacency Multilists

a) Adjacency Matrix

Let G=(V,E) be a graph with n vertices.
The adjacency matrix of G is a two-dimensional by array, say adj_mat.
If the edge (vi, vj) is in E(G), adj_mat[i][j]=1
If there is no such edge in E(G), adj_mat[i][j]=0

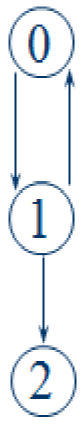
The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix:



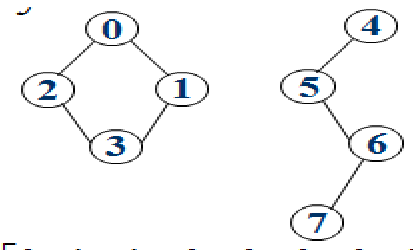
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2



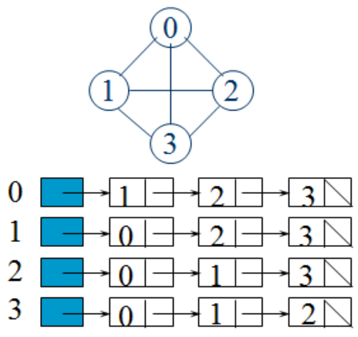
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

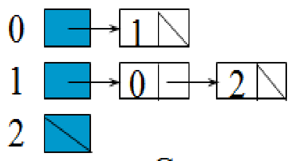
Merits of Adjacency Matrix
From the adjacency matrix, to determine the connection of vertices is easy
The degree of a vertex is
For a digraph, the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

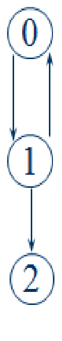
b) Adjacency Lists
Each row in adjacency matrix is represented as an adjacency list.

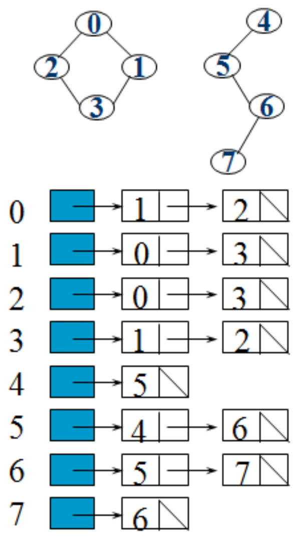


G_1



G_3



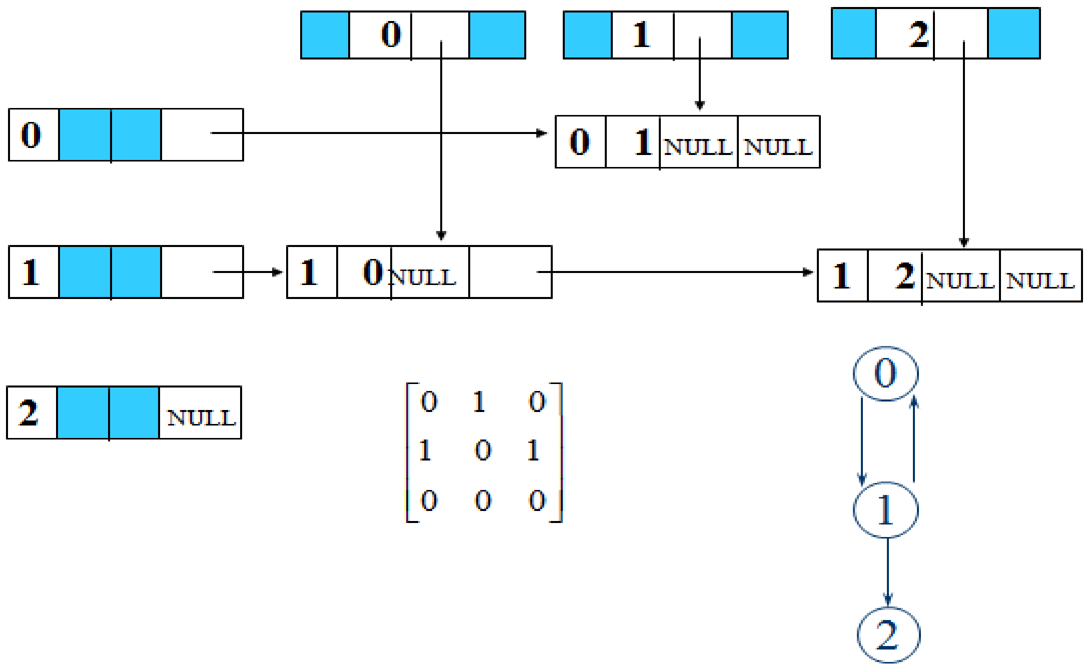


G_4

Interesting Operations

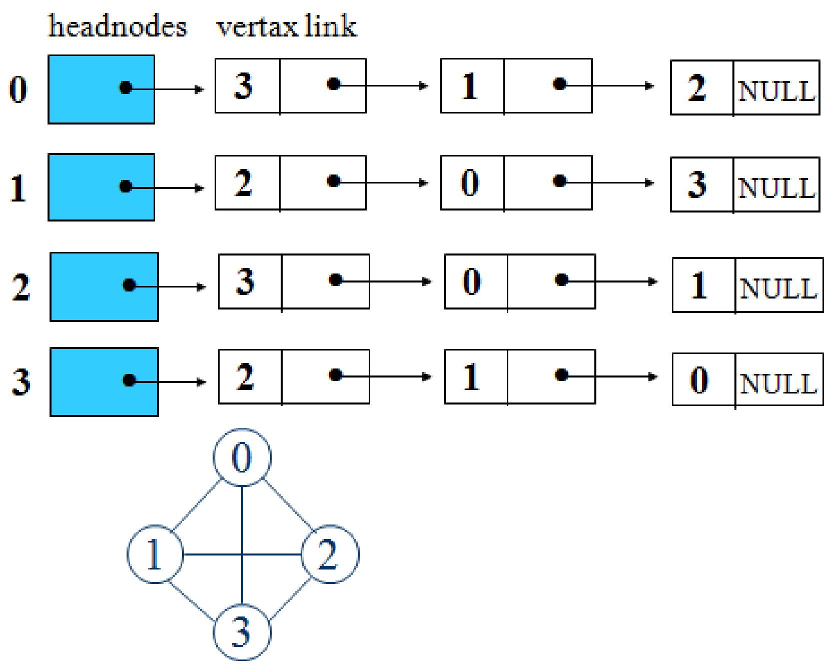
- degree of a vertex in an undirected graph
  # of nodes in adjacency list
- # of edges in a graph
  determined in O(n+e)
- out-degree of a vertex in a directed graph
  # of nodes in its adjacency list
- in-degree of a vertex in a directed graph
  traverse the whole data structure

Orthogonal representation for graph G$_3$



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Order is of no significance.



c) Adjacency Multilists

An edge in an undirected graph is represented by two nodes in adjacency list representation.

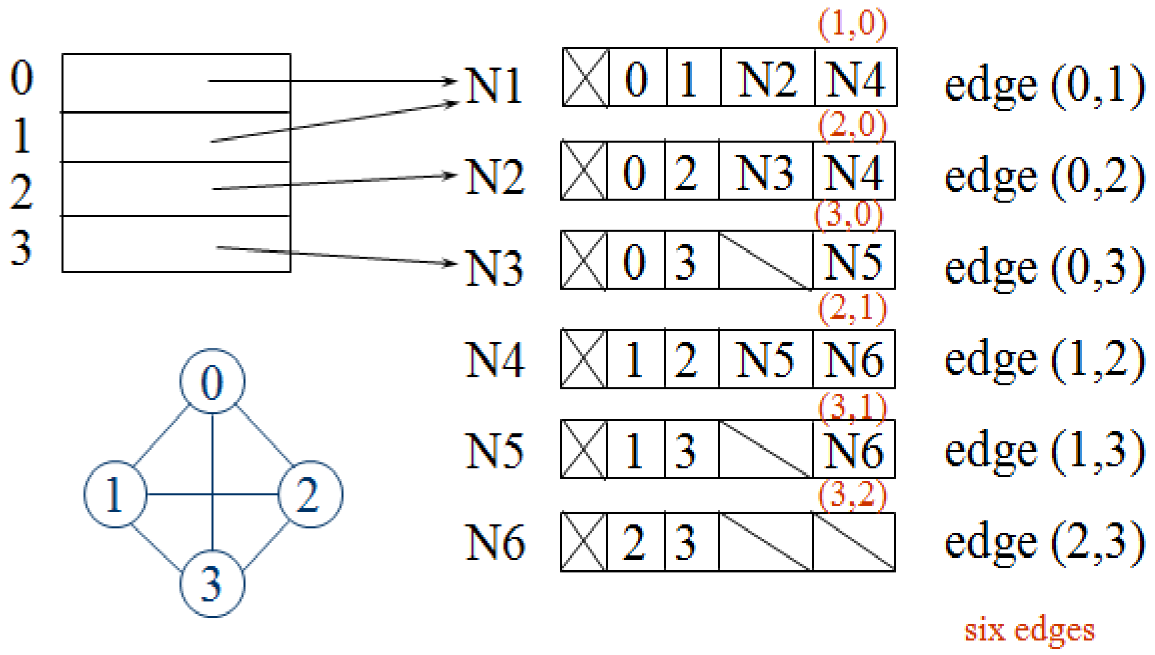
Adjacency Multilists

- lists in which nodes may be shared among several lists. (an edge is shared by two different paths)

marked	vertex1	vertex2	path1	path2

Example for Adjacency Multlists

Lists: vertex 0: M1->M2->M3, vertex 1: M1->M4->M5
vertex 2: M2->M4->M6, vertex 3: M3->M5->M6



Some Graph Operations

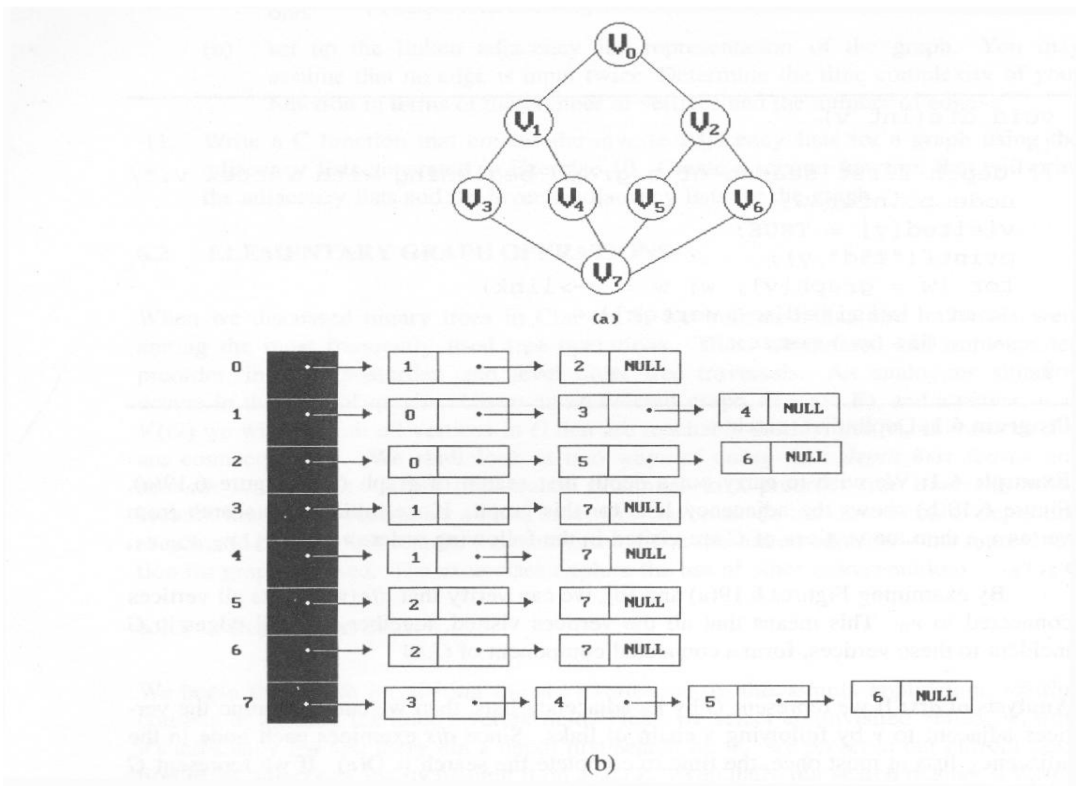
The following are some graph operations:

a) Traversal
   Given G=(V,E) and vertex v, find all w∊V, such that w connects v.
   - Depth First Search (DFS)
     preorder tree traversal
   - Breadth First Search (BFS)
     level order tree traversal

b) Spanning Trees

c) Connected Components

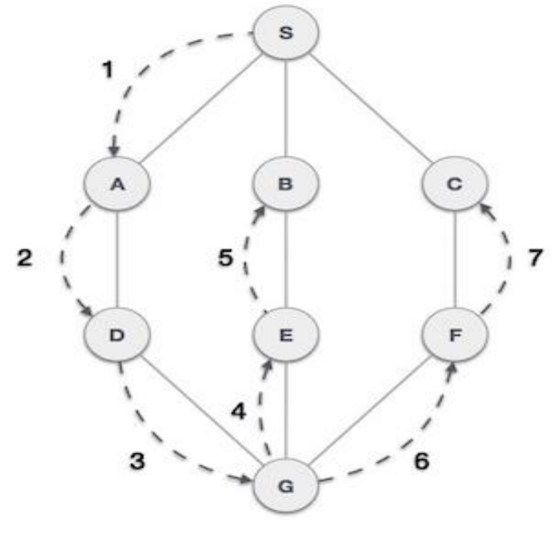
Graph *G* and its adjacency lists



depth first search: v0, v1, v3, v7, v4, v5, v2, v6
breadth first search: v0, v1, v2, v3, v4, v5, v6, v7

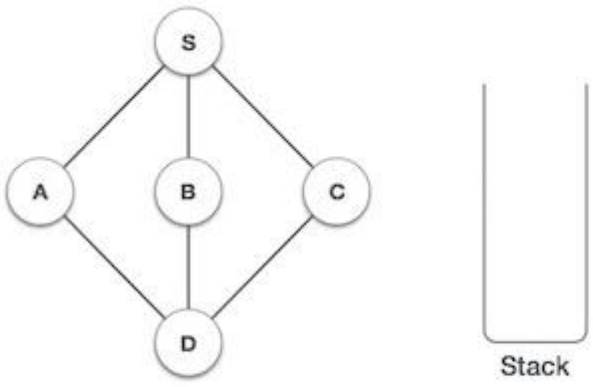
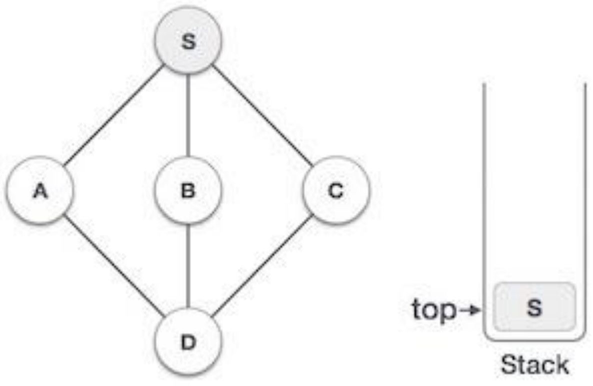
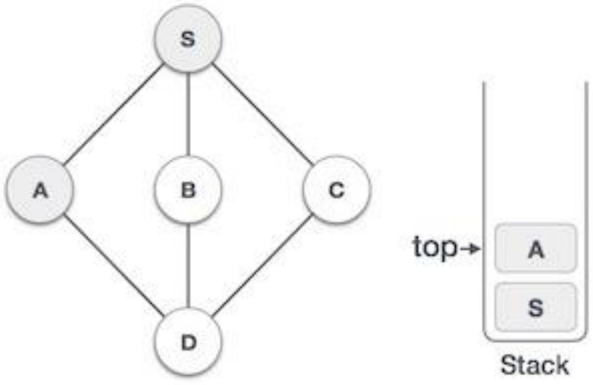
Depth First Search

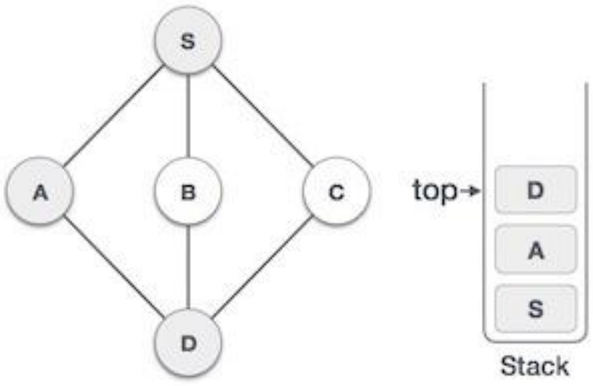
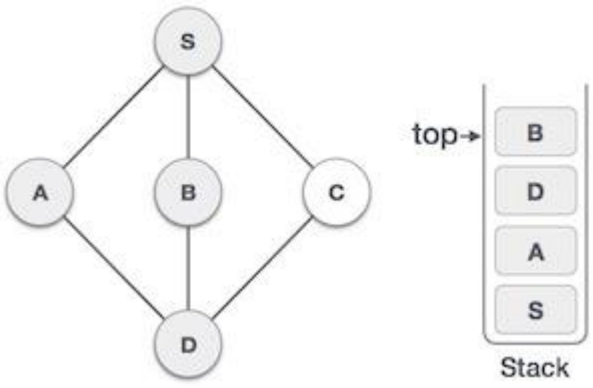
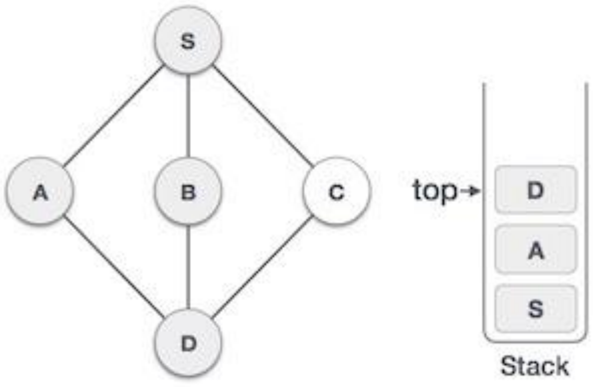
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

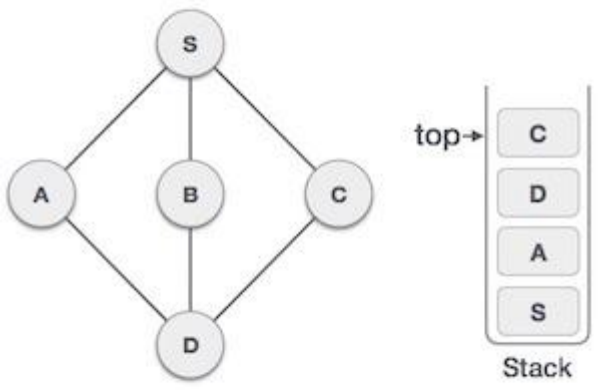


As in the example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1.		Initialize the stack.
2.		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4.		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5.		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6.		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7.		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

- As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

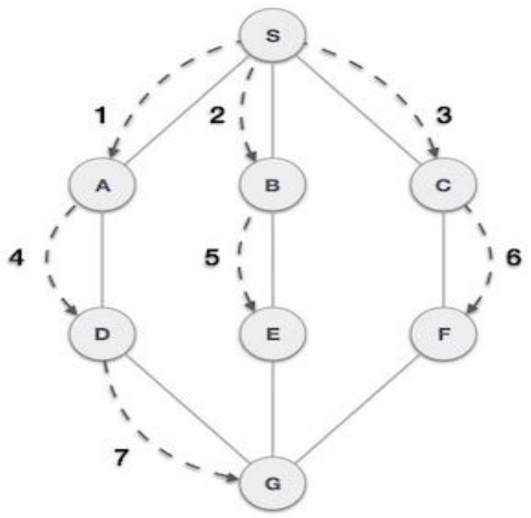
Psuedocode for DFS

```
DFS-iterative (G, s):    //Where G is graph and s is source vertex
    let S be stack
    S.push( s )            //Inserting s in stack
    mark s as visited.
    while ( S is not empty):
        //Pop a vertex from stack to visit next
        v  =  S.top( )
       S.pop( )
       //Push all the neighbours of v in stack that are not visited
      for all neighbours w of v in Graph G:
          if w is not visited :
                   S.push( w )
                  mark w as visited
---------------------------------------------------------------------------------------------------------

  DFS-recursive(G, s):
      mark s as visited
      for all neighbours w of s in Graph G:
          if w is not visited:
              DFS-recursive(G, w)
```

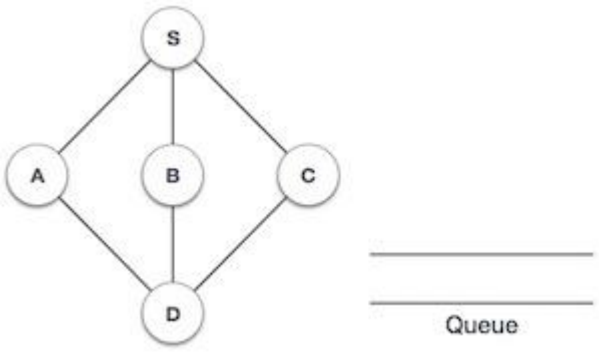
Breadth First Search

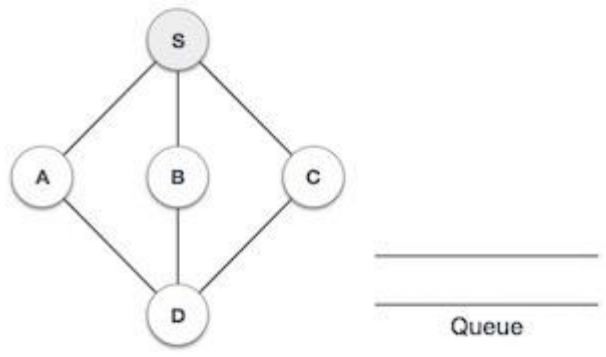
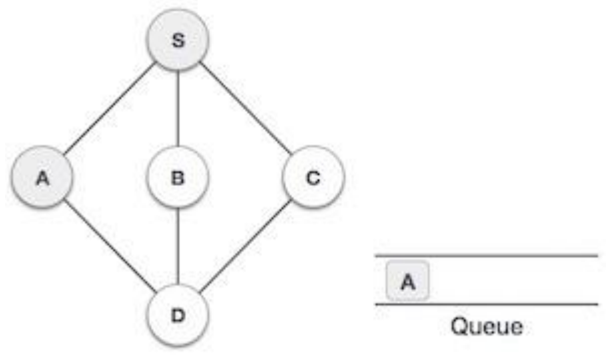
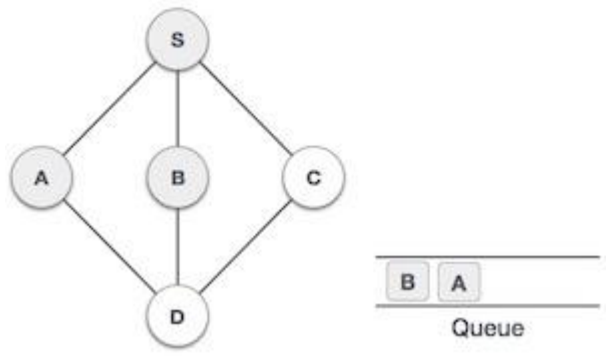
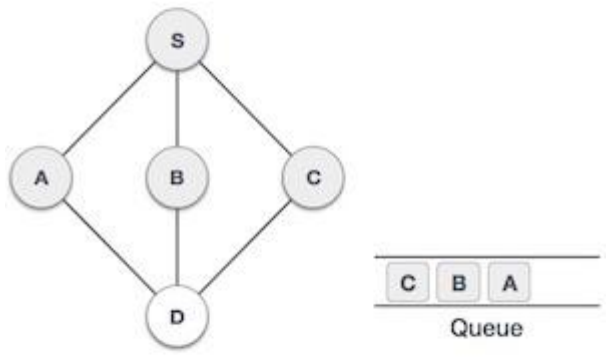
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

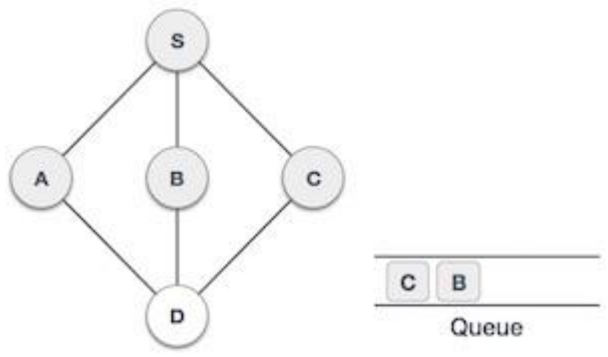
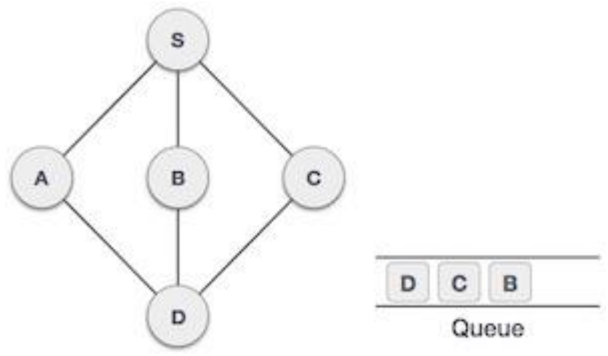


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1.		Initialize the queue.

2.		We start from visiting **S** (starting node), and mark it as visited.
3.		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4.		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5.		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

6.		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7.		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

- At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Psuedocode for BFS

```
BFS (G, s)                   //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s )
      mark s as visited.
      while ( Q is not empty)
           v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                        Q.enqueue( w )
                        mark w as visited.
```

Spanning Trees

When graph G is connected, a depth first or breadth first search starting at any vertex will visit all vertices in G

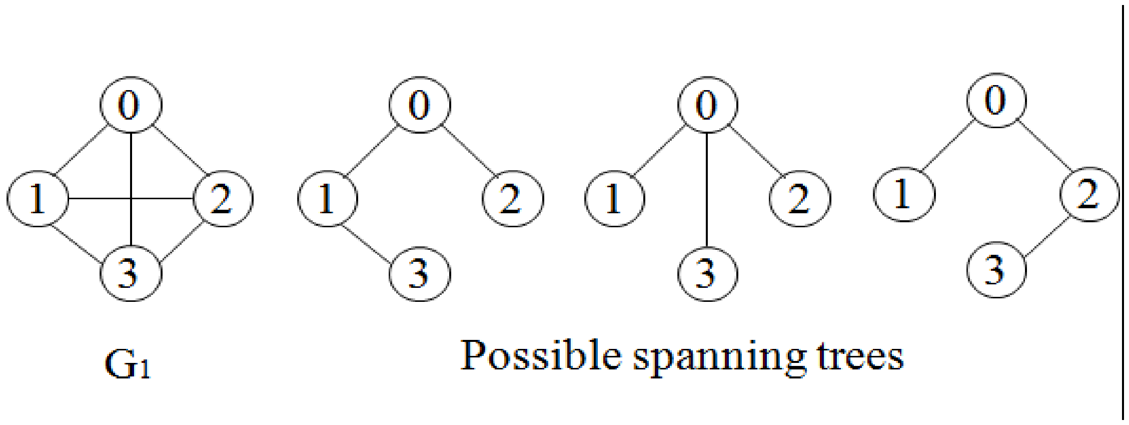
A spanning tree is any tree that consists solely of edges in G and that includes all the vertices

E(G): T (tree edges) + N (nontree edges)

where T: set of edges used during search

N: set of remaining edges

Examples of Spanning Tree

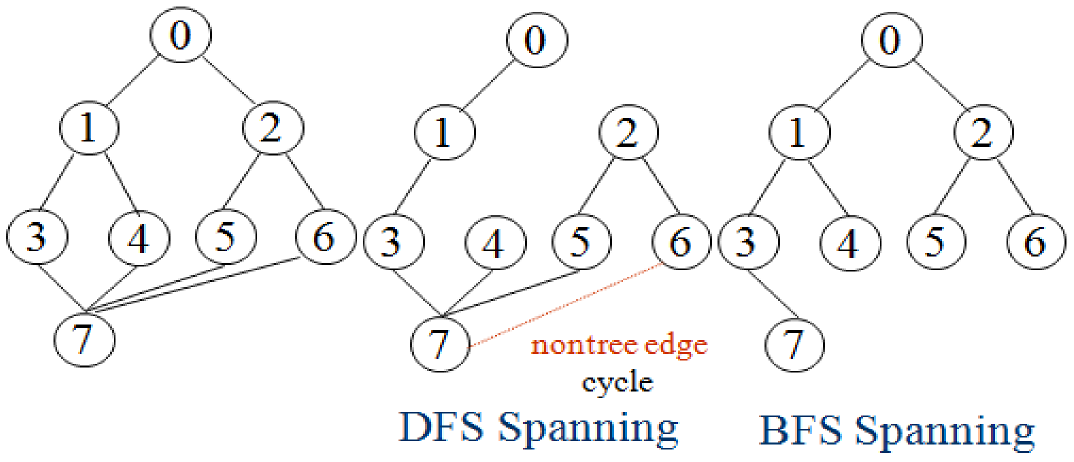


Either dfs or bfs can be used to create a spanning tree

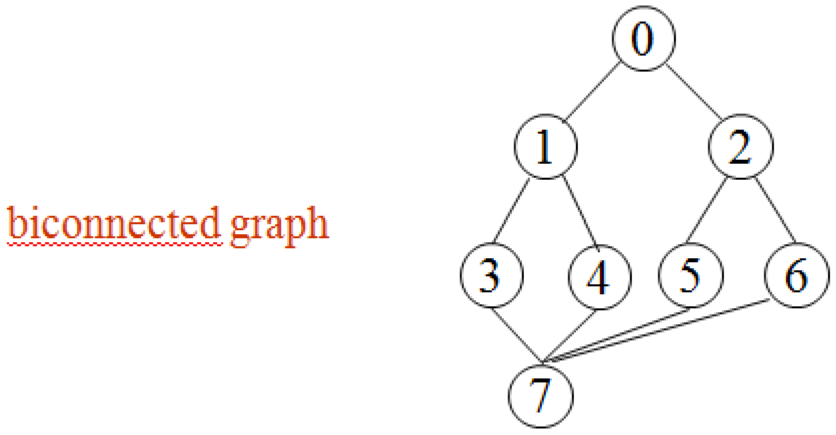
- When dfs is used, the resulting spanning tree is known as a depth first spanning tree
- When bfs is used, the resulting spanning tree is known as a breadth first spanning tree

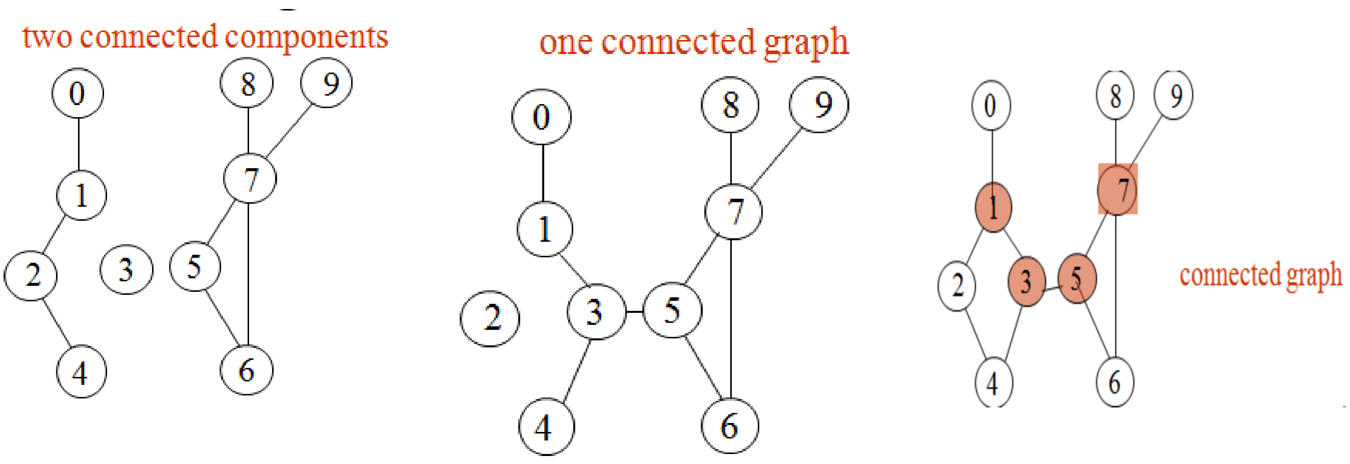
While adding a nontree edge into any spanning tree, this will create a cycle

DFS VS BFS Spanning Tree

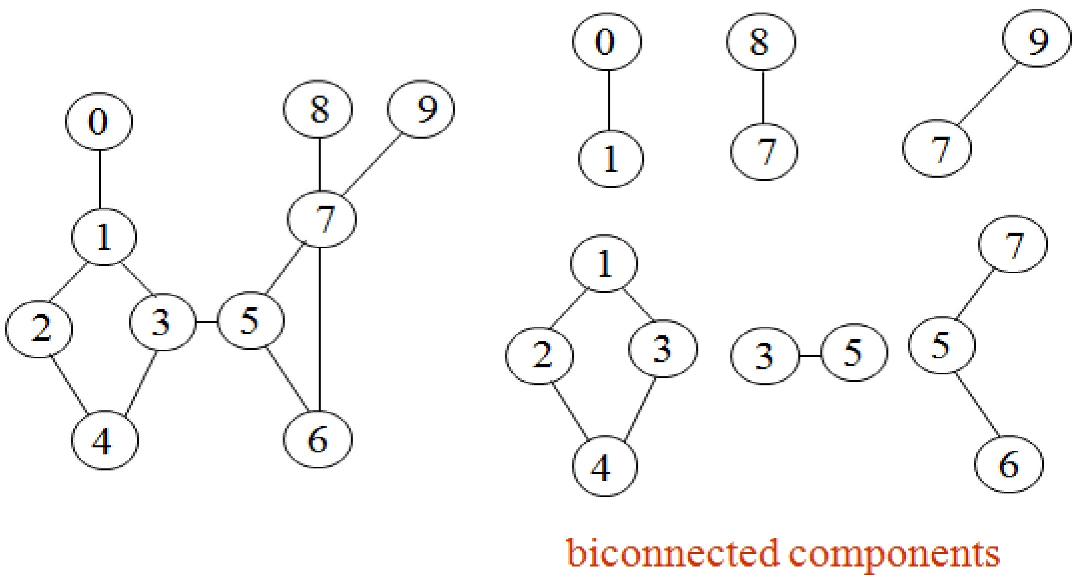


A spanning tree is a minimal subgraph, G', of G such that V(G')=V(G) and G' is connected.
Any connected graph with n vertices must have at least n-1 edges.
A biconnected graph is a connected graph that hasno articulation points.





biconnected component: a maximal connected subgraph H (no subgraph that is both biconnected and properly contains H).



Minimum Cost Spanning Tree

- The cost of a spanning tree of a weighted undirected graph is the sum of the costs of the edges in the spanning tree
- A minimum cost spanning tree is a spanning tree of least cost
- Three different algorithms can be used
  - Kruskal
  - Prim
  - Sollin

Kruskal's Algorithm

Build a minimum cost spanning tree T by adding edges to T one at a time
Select the edges for inclusion in T in nondecreasing order of the cost
An edge is added to T if it does not form a cycle
Since G is connected and has n > 0 vertices, exactly n-1 edges will be selected

Kruskal's algorithm

```
1. Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

3. Repeat step#2 until there are (V-1) edges in the spanning tree.
```

Psuedocode for Kruskal's Algorithm

```
Kruskal(G, V, E)
{
        T= {};
        while(T contains less than n-1 edges && E is not empty)
        {
            choose a least cost edge (v,w) from E;
            delete (v,w) from E;
            if ((v,w) does not create a cycle in T)
                 add (v,w) to T
            else
                 discard (v,w);
        }
        if (T contains fewer than n-1 edges)
                printf("No spanning tree\n");

}
```

Examples for Kruskal's Algorithm

$0 \overset{10}{\text{———}} 5$

$2 \overset{12}{\text{———}} 3$

$1 \overset{14}{\text{———}} 6$

$1 \overset{16}{\text{———}} 2$

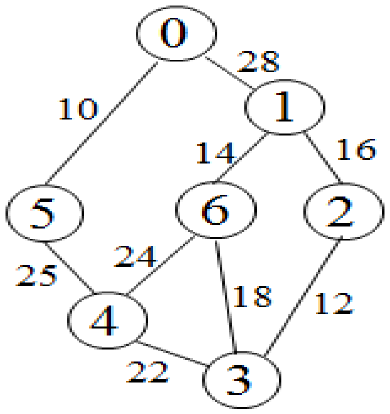
$3 \overset{18}{\text{———}} 6$

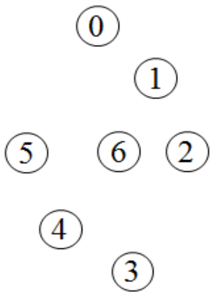
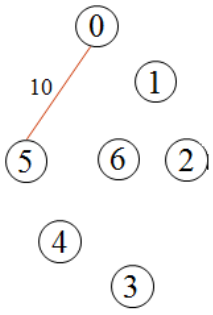
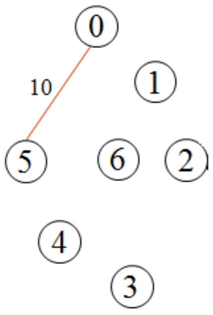
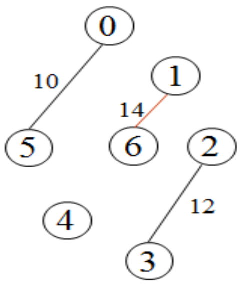
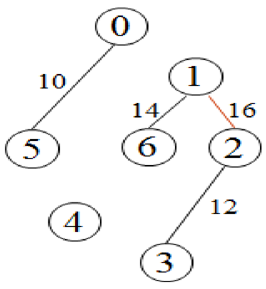
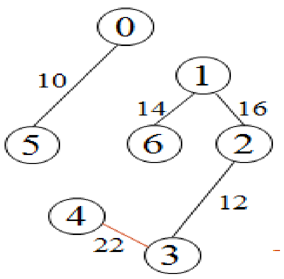
$3 \overset{22}{\text{———}} 4$

$4 \overset{24}{\text{———}} 6$

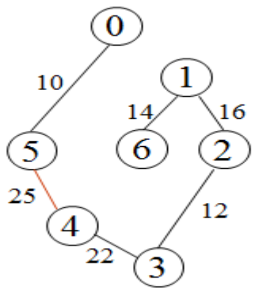
$4 \overset{25}{\text{———}} 5$

$0 \overset{28}{\text{———}} 1$



	Iteration 1 	Iteration 2 
Iteration 3 	Iteration 4 	Iteration 5 

Iteration 6



Prim’s Algorithm

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –

Steps of Prim's Algorithm:

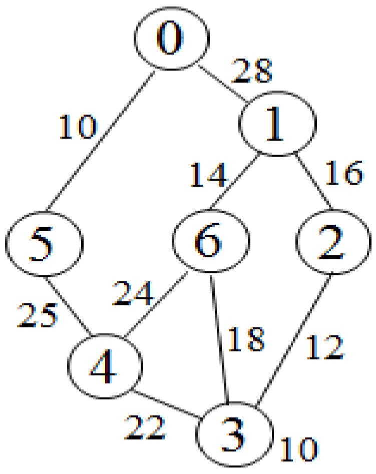
The following are the main 3 steps of the Prim's Algorithm:

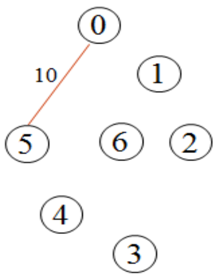
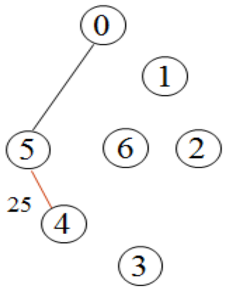
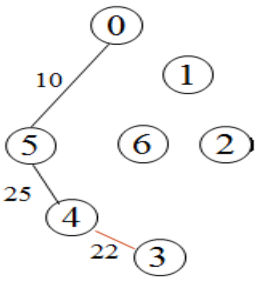
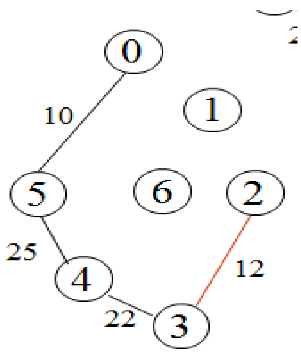
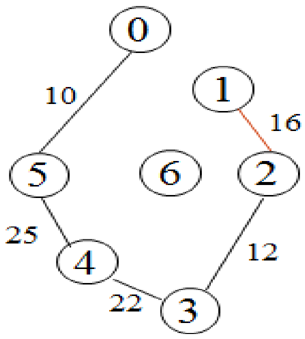
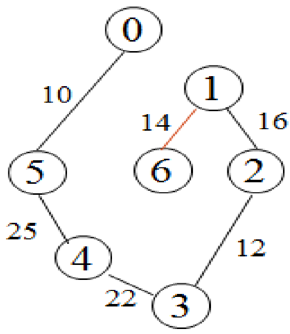
1. Begin with any vertex which you think would be suitable and add it to the tree.
2. Find an edge that connects any vertex in the tree to any vertex that is not in the tree. Note that, we don't have to form cycles.
3. Stop when n - 1 edges have been added to the tree.

Psuedocode of Prim’s algorithm

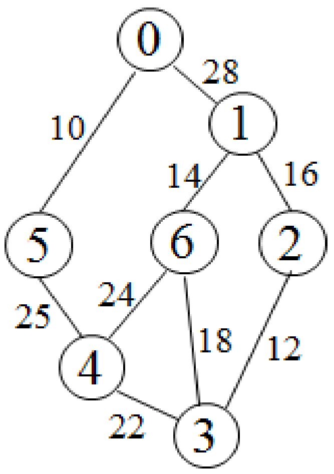
```
Prims(G,V,E)
{
        T={};
        TV={0};
        while (T contains fewer than n-1 edges)
        {
                let (u,v) be a least cost edge such that and if (there is no
                such edge ) break;
                add v to TV;
                add (u,v) to T;
        }
        if (T contains fewer than n-1 edges)
                printf(“No spanning tree\n”);
}
```

Examples for Prim’s Algorithm

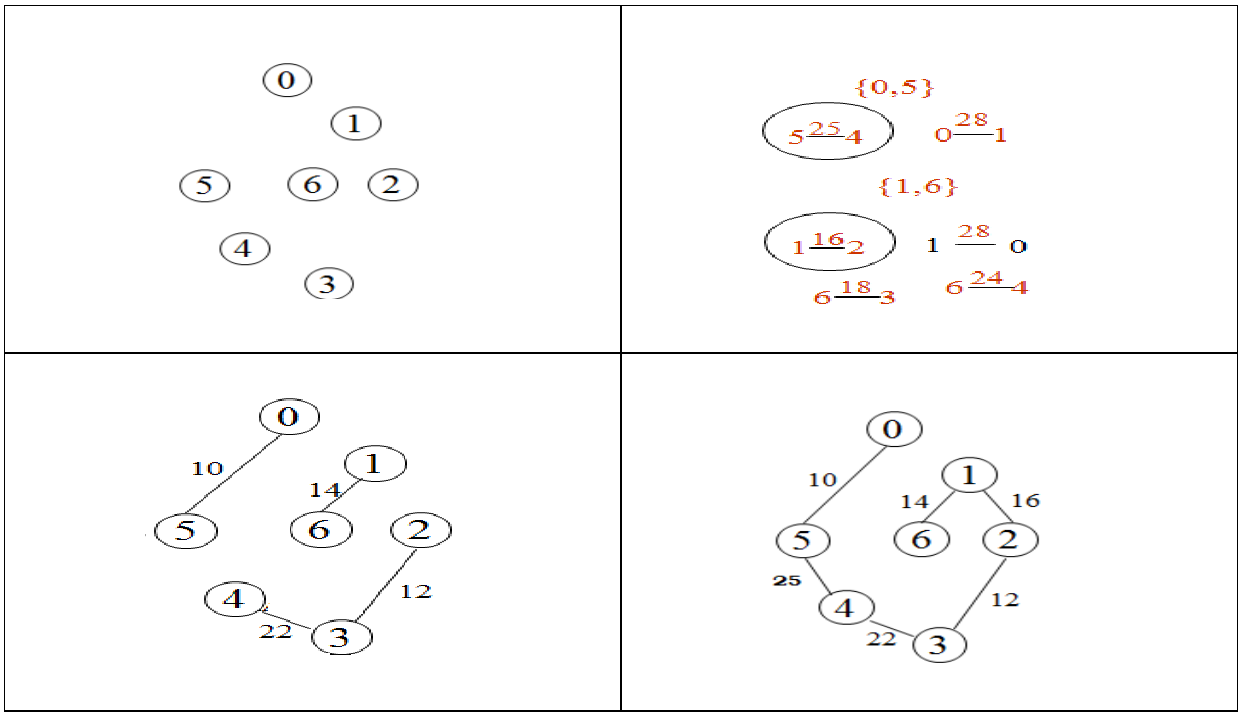


Sollin's Algorithm

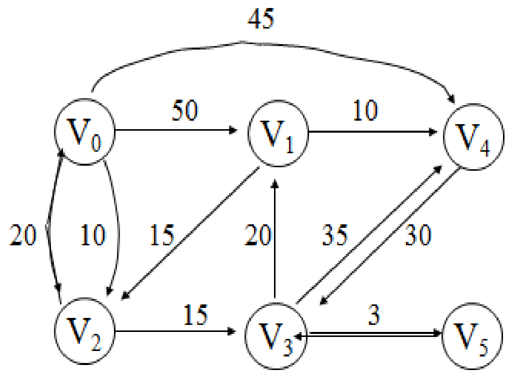


vertex	edge
0	0 -- 10 --> 5, 0 -- 28 --> 1
1	1 -- 14 --> 6, 1-- 16 --> 2, 1 -- 28 --> 0
2	2 -- 12 --> 3, 2 -- 16 --> 1
3	3 -- 12 --> 2, 3 -- 18 --> 6, 3 -- 22 --> 4
4	4 -- 22 --> 3, 4 -- 24 --> 6, 5 -- 25 --> 5
5	5 -- 10 --> 0, 5 -- 25 --> 4
6	6 -- 14 --> 1, 6 -- 18 --> 3, 6 -- 24 --> 4



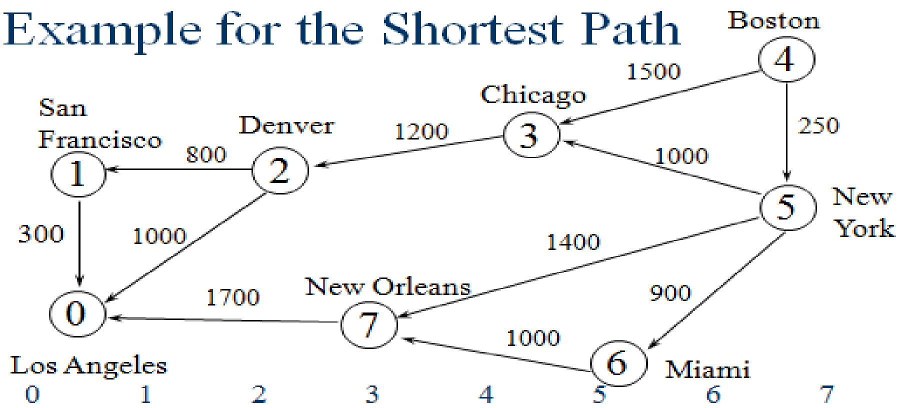
Single Source All Destinations
<u>Graph and shortest paths from v_0</u>

Dijkstra's Algorithm



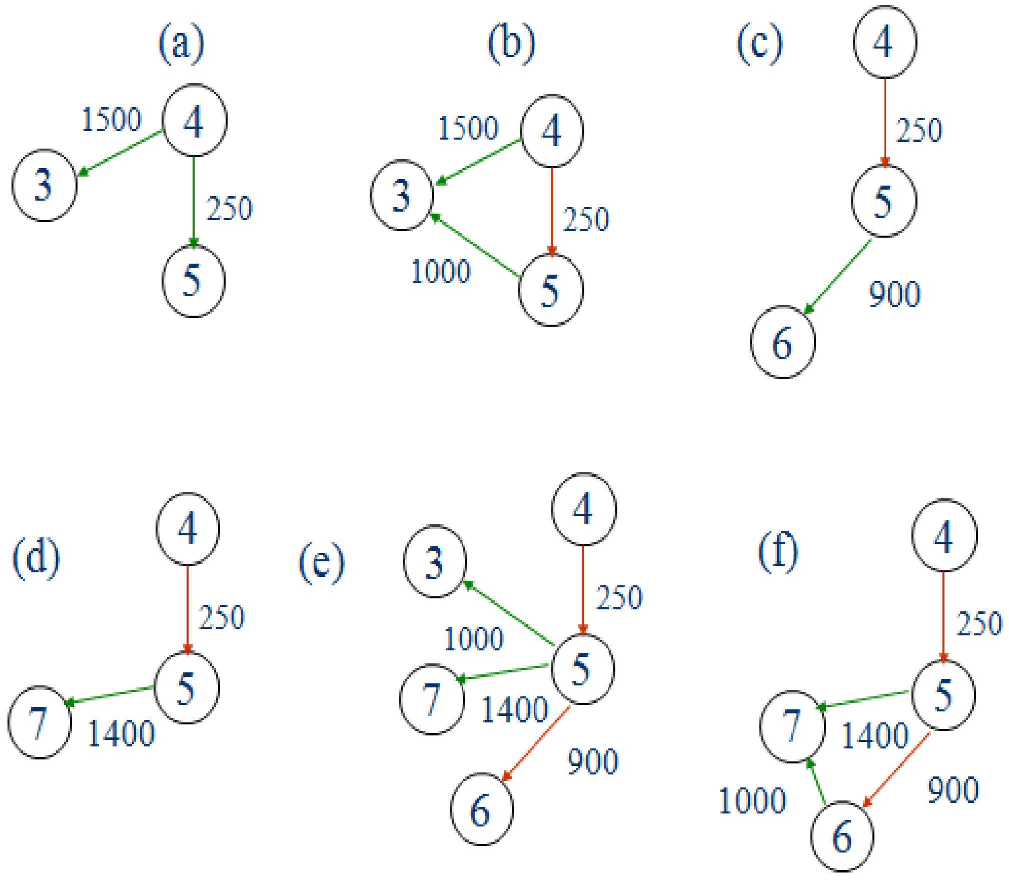
path	length
1) v0 v2	10
2) v0 v2 v3	25
3) v0 v2 v3 v1	45
4) v0 v4	45

Example for the Shortest Path

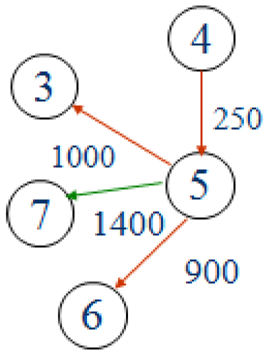


	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

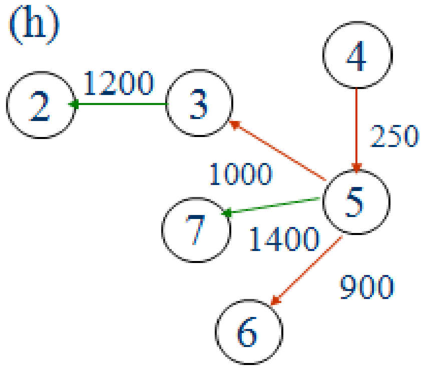
Cost adjacency matrix



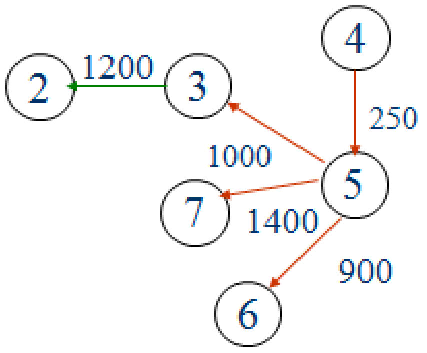
(g)



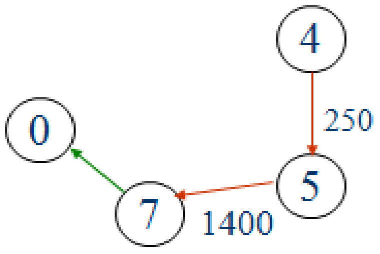
(h)



(i)



(j)



Iteration	S	Vertex Selected	LA [0]	SF [1]	DEN [2]	CHI [3]	BO [4]	NY [5]	MIA [6]	NO
Initial	--	----	+∞	+∞	+∞ (b)	1500	0	250 (c)	+∞ (d)	+∞
1	{4} (a)	5	+∞	+∞	+∞	1250	0	250	1150	1650
2	{4,5} (e)	6	+∞	+∞	+∞ (h)	1250	0	250	1150 (f)	1650
3	{4,5,6} (g)	3	+∞	+∞	2450	1250	0	250	1150	1650
4	{4,5,6,3} (i)	7	(j) 3350	+∞	2450	1250	0	250	1150	1650
5	{4,5,6,3,7}	2	3350	3250	2450	1250	0	250	1150	1650
6	{4,5,6,3,7,2}	1	3350	3250	2450	1250	0	250	1150	1650
7	{4,5,6,3,7,2,1}									

```
#define MAX_VERTICES 6
int cost[][MAX_VERTICES]=
   {{   0,    50,    10, 1000,    45, 1000},
    {1000,     0,    15, 1000,    10, 1000},
    {  20, 1000,     0,    15, 1000, 1000},
    {1000,    20, 1000,     0,    35, 1000},
    {1000, 1000,    30, 1000,     0, 1000},
    {1000, 1000, 1000,     3, 1000,     0}};
int distance[MAX_VERTICES];
short int found{MAX_VERTICES];
int n = MAX_VERTICES;
```

void shortestpath(int v, int cost[][MAX_ERXTICES], int distance[], int n, short int found[])

```
{
    int i, u, w;
    for (i=0; i<n; i++)
    {
        found[i] = FALSE;
        distance[i] = cost[v][i];
     }
    found[v] = TRUE;
    distance[v] = 0;
    for (i=0; i<n-2; i++)
        {
            determine n-1 paths from v
            u = choose(distance, n, found);
            found[u] = TRUE;
            for (w=0; w<n; w++)
            if (!found[w])
                if (distance[u]+cost[u][w]<distance[w])
                        distance[w] = distance[u]+cost[u][w];
          }
}
```

Floyd Warshall Algorithm

All Pairs Shortest Paths

All pairs shortest path algorithm finds the shortest paths between all pairs of vertices.

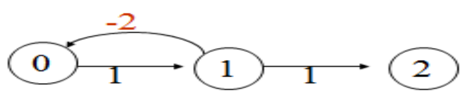
Solution 1

- Apply shortest path n times with each vertex as source. O(n^3)

Solution 2

- Represent the graph G by its cost adjacency matrix with cost[i][j]
- If the edge <i,j> is not in G, the cost[i][j] is set to some sufficiently large number
- A[i][j] is the cost of the shortest path form i to j, using only those intermediate vertices with an index <= k
- The cost of the shortest path from i to j is A [i][j], as no vertex in G has an index greater than n-1
- A [i][j]=cost[i][j]
- Calculate the A, A, A, ..., A from A iteratively
- A [i][j]=min{A [i][j], A [i][k]+A [k][j]}, k>=0

Graph with negative cycle



(a) Directed graph

$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

(b) A^{-1}

The length of the shortest path from vertex 0 to vertex 2 is -∞.

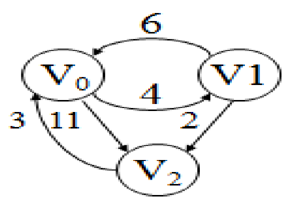
0, 1, 0, 1,0, 1, ..., 0, 1, 2

Algorithm for All Pairs Shortest Paths

```
void allcosts(int cost[][MAX_VERTICES], int distance[][MAX_VERTICES], int n)
{
  int i, j, k;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
       distance[i][j] = cost[i][j];
  for (k=0; k<n; k++)
    for (i=0; i<n; i++)
      for (j=0; j<n; j++)
        if (distance[i][k]+distance[k][j] < distance[i][j])
           distance[i][j]= distance[i][k]+distance[k][j];
}
```

Example

<u>Directed graph and its cost matrix</u>



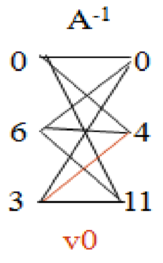
(a)Digraph G

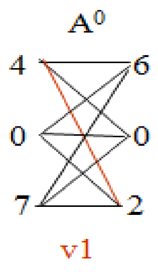
	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(b)Cost adjacency matrix for G

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

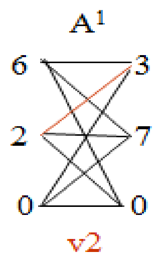
A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0





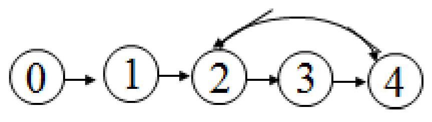
A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0



Transitive Closure

Goal: given a graph with unweighted edges, determine if there is a path from i to j for all i and j.
(1) Require positive path (> 0) lengths. transitive closure matrix
(2) Require nonnegative path (≥0) lengths. reflexive transitive closure matrix



(a) Digraph G

$$\begin{array}{c|ccccc} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 & 0 \end{array}$$

(b) Adjacency matrix A for G

$$\begin{array}{c|ccccc} 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 & 1 & 1 \\ 3 & 0 & 0 & 1 & 1 & 1 \\ 4 & 0 & 0 & 1 & 1 & 1 \end{array}$$

cycle

(c) transitive closure matrix A^{+}

There is a path of length > 0

$$\begin{array}{c|ccccc} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 & 1 & 1 \\ 3 & 0 & 0 & 1 & 1 & 1 \\ 4 & 0 & 0 & 1 & 1 & 1 \end{array}$$

reflexive

(d) reflexive transitive closure matrix A^{*}

There is a path of length ≥0

****