Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

CET2001B Advanced data Structure

S. Y. B. Tech CSE

Semester - IV

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

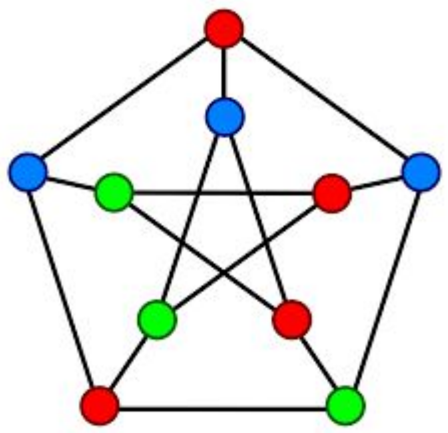
Graph

Graph- Basic Terminology, memory representation**:** Adjacency matrix, Adjacency list, Creation of Graph and Traversals,

Minimum spanning Tree- Prim’s and Kruskal’s Algorithms, Dikjtra's Single source shortest path, Topological sorting

Graph

- Basic Terminology
- Memory representation
- Creation of graph and traversals
- Minimum spanning tree
- Topological sorting

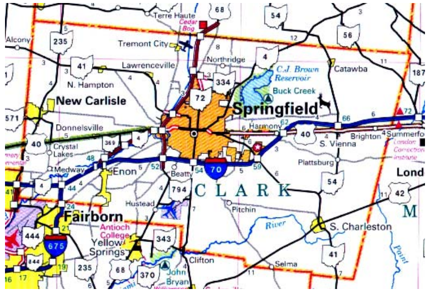


Basic Terminology

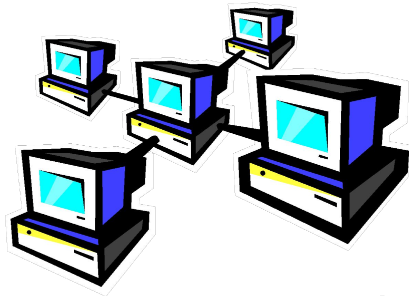
- Definition
- Complete Graph
- Adjacent and Incident
- Subgraph and path
- Simple path and cycle
- Connected components
- Degree

Graph Applications

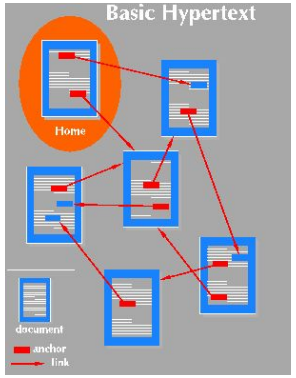
Social Network



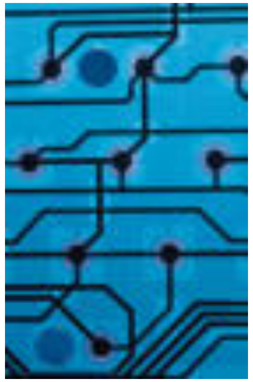
Maps



Computer Network



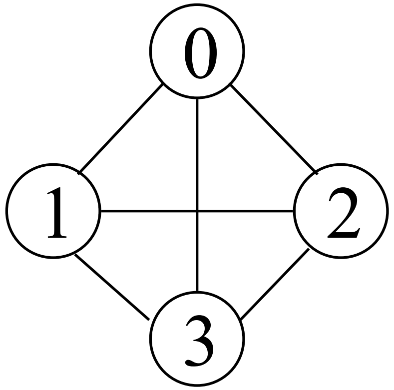
Hypertext



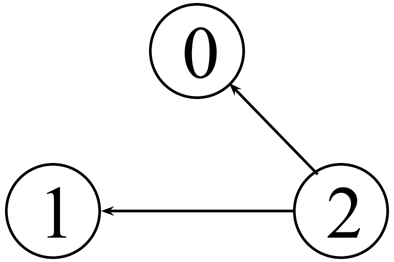
Circuits

Definition

- A graph G consists of two sets
  - a finite, nonempty set of vertices V(G)
  - a finite, possible empty set of edges E(G)
  - G(V,E) represents a graph



Graph G_1



Graph G2

Vertex Set: V(G1)={0,1,2,3}
Edge Set: E(G1)={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}

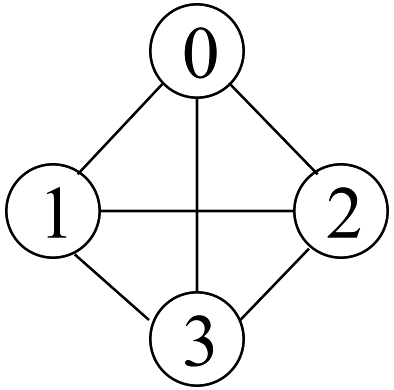
Vertex Set: V(G2)={0,1,2}
Edge Set: E(G2)={(2,0),(2,1)}

Directed and Undirected Graph

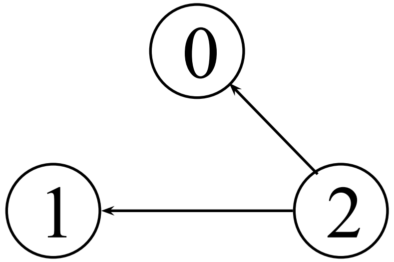
- An undirected graph is one in which the pair of vertices in an edge is unordered,
  (v0, v1) = (v1,v0)

- A directed graph is one in which each edge is a directed pair of vertices,

  <v0, v1> != <v1,v0>



Undirected Graph



Directed Graph

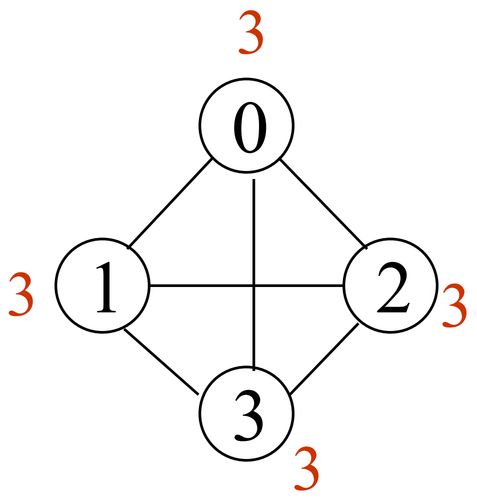
Degree

- 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

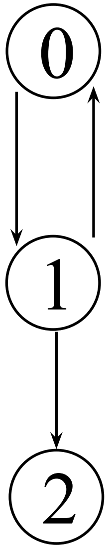
  - If d_i is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges (of undirected graph) are :-

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

Examples for Degree



Undirected Graph : G_1



in:1, out: 1

in: 1, out: 2

in: 1, out: 0

Directed Graph: G_3

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

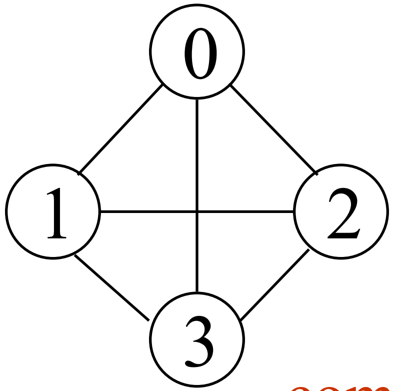
Complete graph

- 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 $n(n-1)/2$

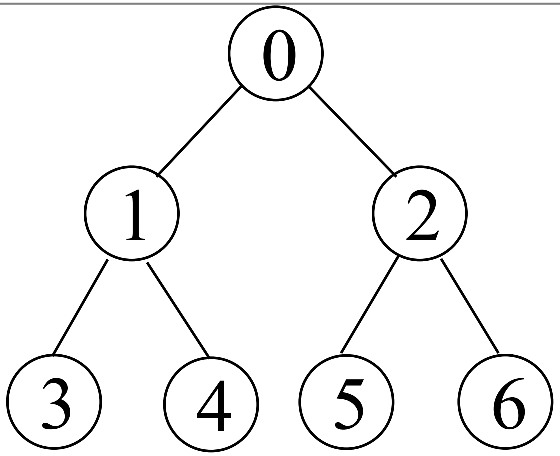
 for directed graph with n vertices, the maximum number of edges is $n(n-1)$

Examples for Graph

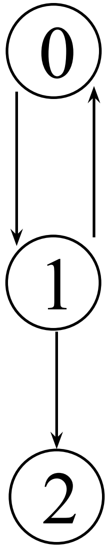


complete graph

G_1



G_2 incomplete graph



G_3

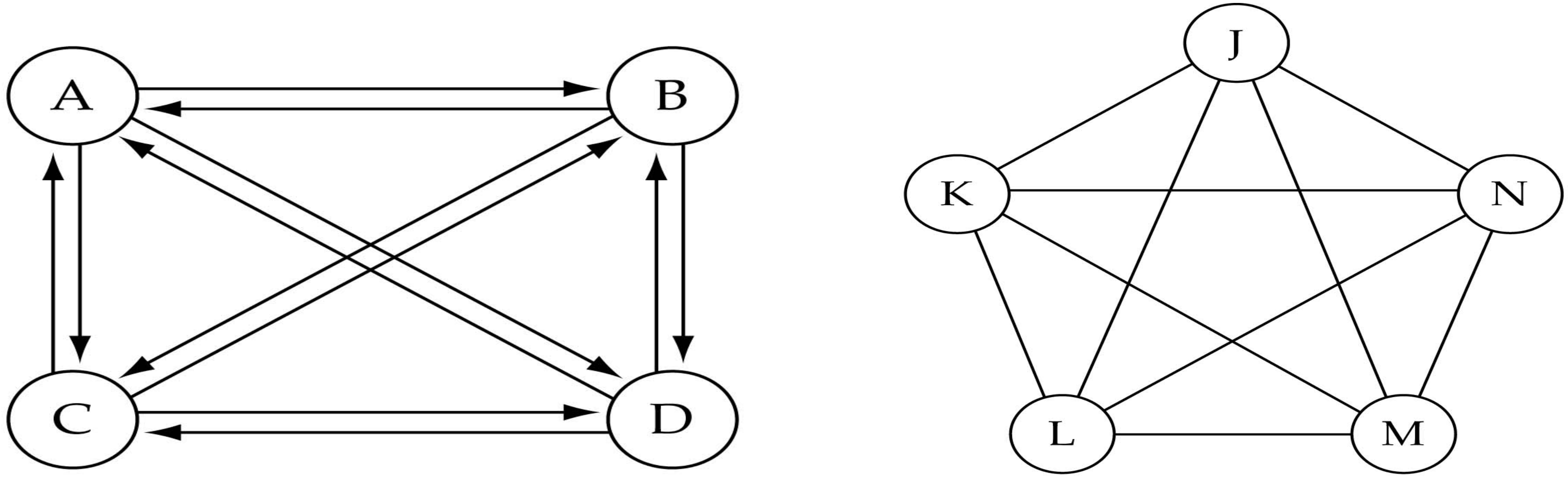
Directed graph

$V(G_1)=\{0,1,2,3\}$ $E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

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

$V(G_3)=\{0,1,2\}$ $E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

Complete Graph



(a) Complete directed graph.

(b) Complete undirected graph.

No. of edges (complete undirected graph) : n(n-1)/2
No. of edges (complete directed graph): n(n-1)

MIT-WPU

॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

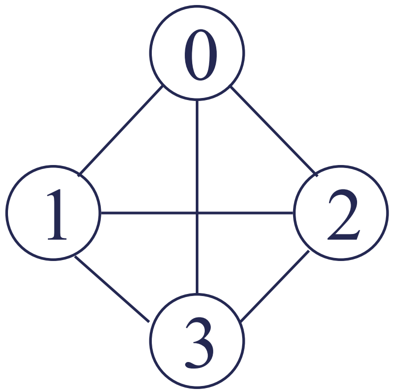
Subgraph and Path

- 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)

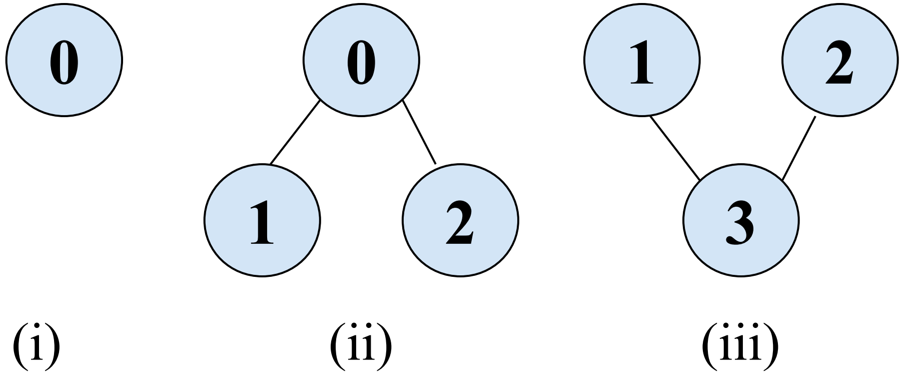
- A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) are edges in an undirected graph

- The length of a path is the number of edges on it

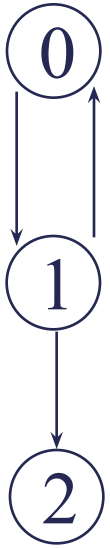
Example for Subgraph



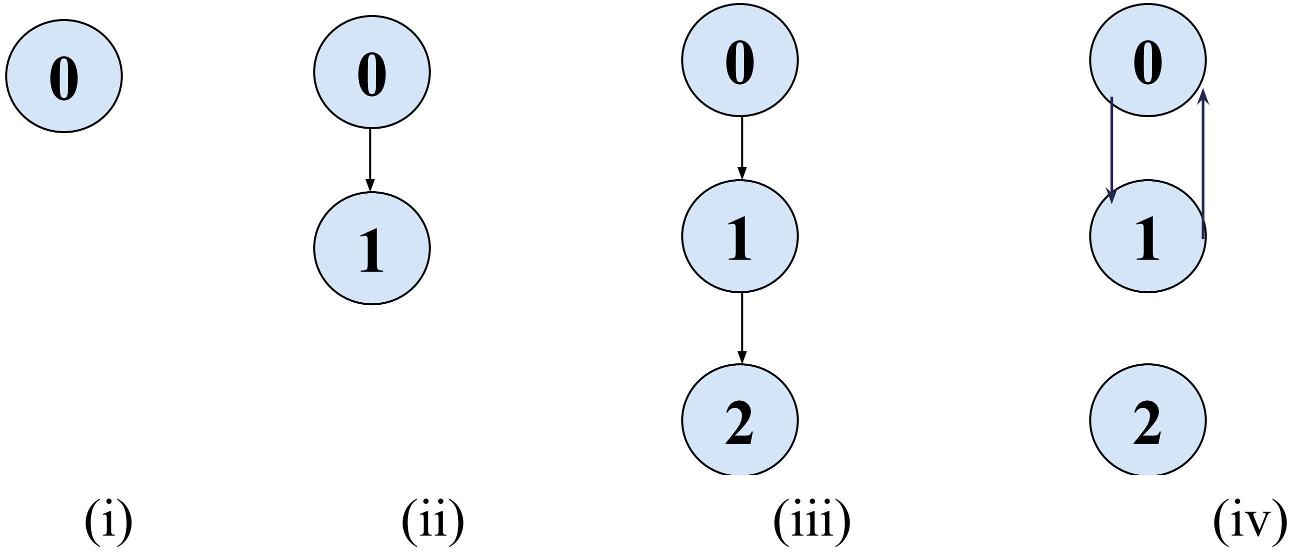
G_1



(a) Some of the subgraph of G_1



G_3

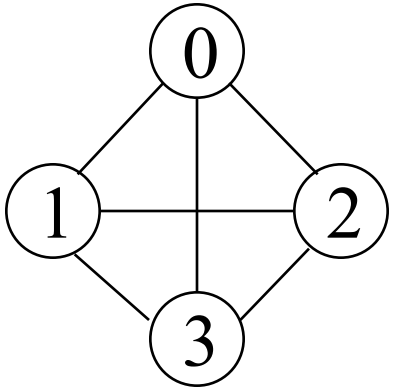


(b) Some of the subgraph of G_3

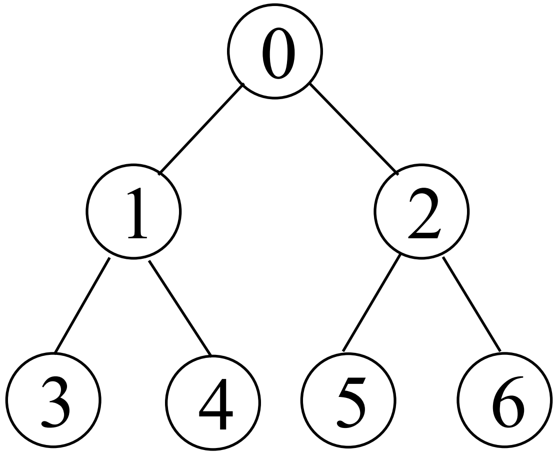
Simple path and cycle

- A simple path is a path in which all the vertices, are distinct.
- A cycle is a path, in which the first and the last vertices are 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.

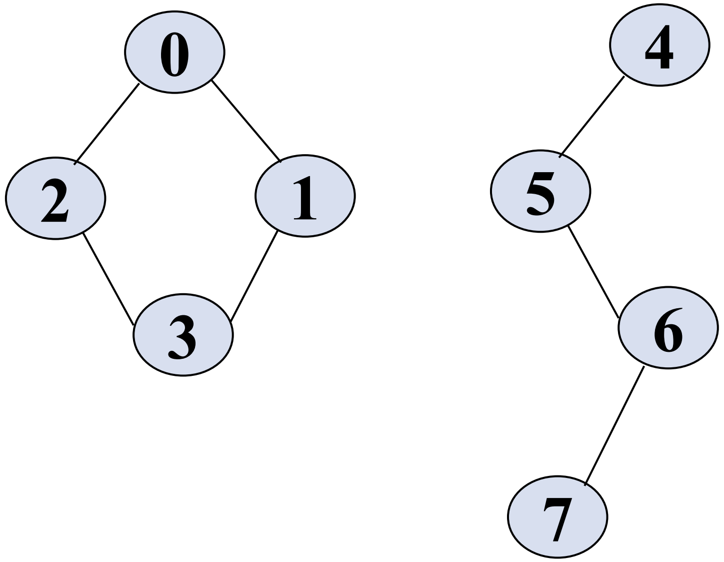
Examples for Graph



G1



G2



G3

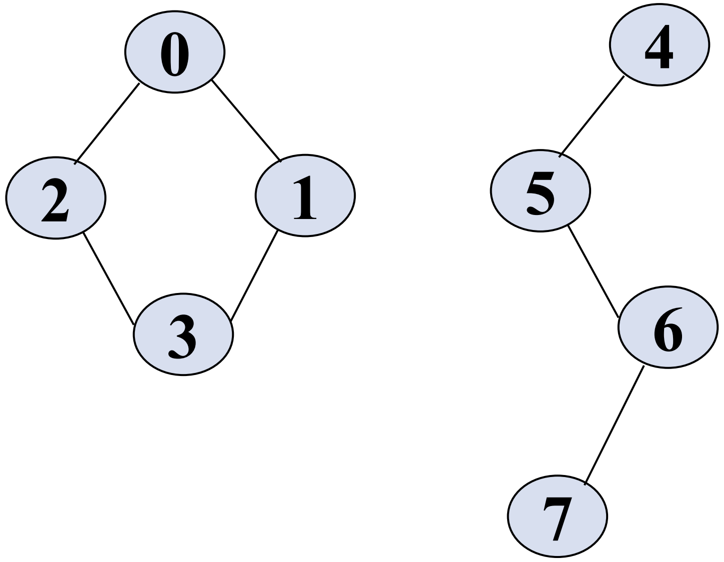
Connected Graphs: G_1 , G_2

Graph G_3 : (not connected)

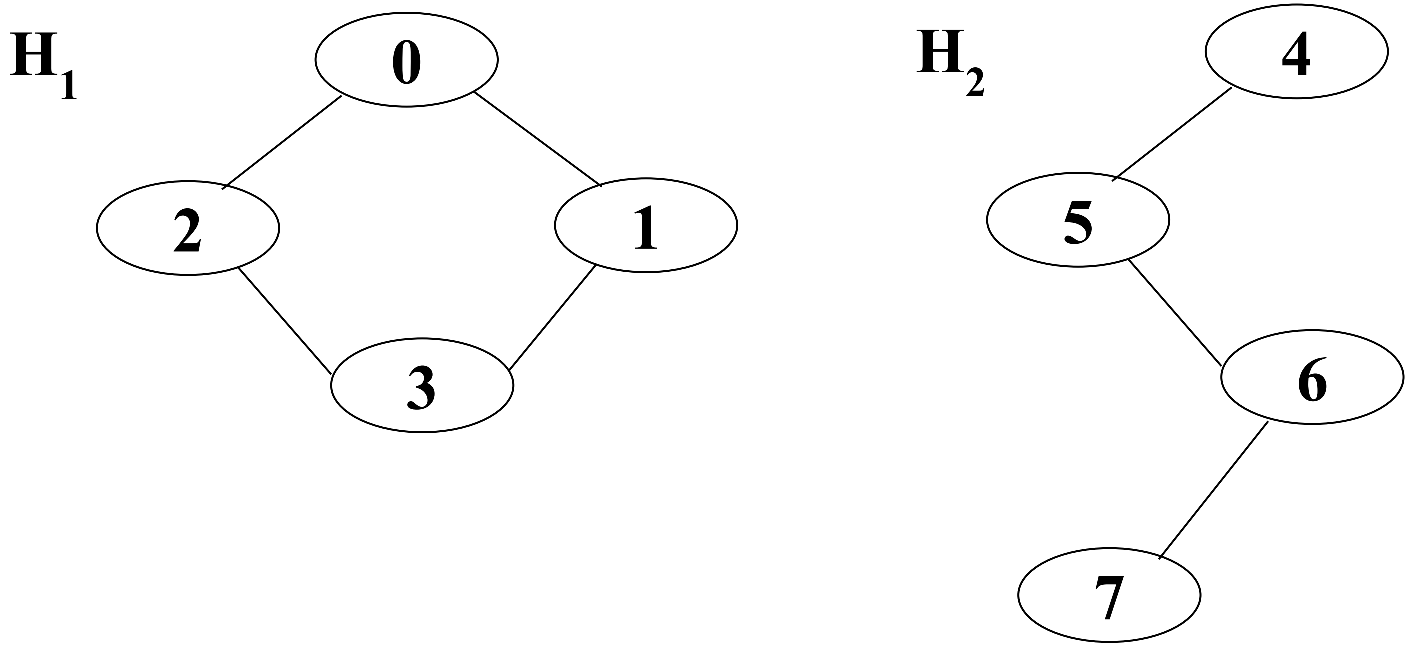
Connected Component

- A connected component of an undirected graph is a maximal connected subgraph.
- A tree is a graph that is connected and acyclic.
- A directed graph is strongly connected if there is a directed path from vi to vj and also from vj to vi.
- A strongly connected component is a maximal subgraph that is strongly connected.

Examples for Connected Component

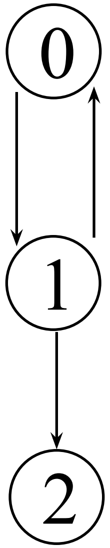


Graph G_4

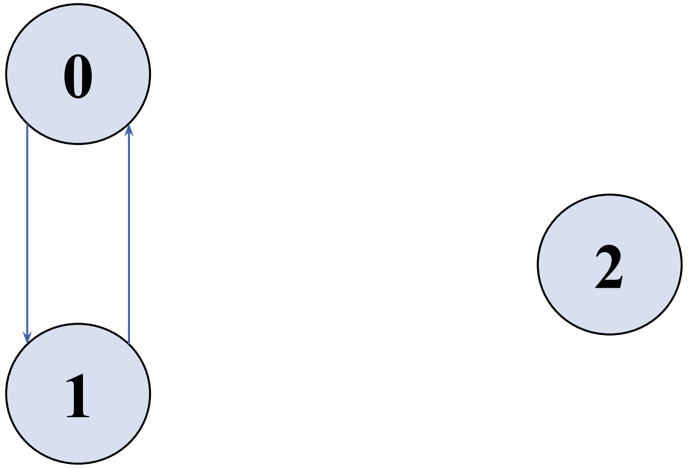


Two Connected Components for Graphs G_4: H_1 and H_2

Examples for Strongly Connected Component



G3 (Not strongly connected)



Strongly connected components of G_3

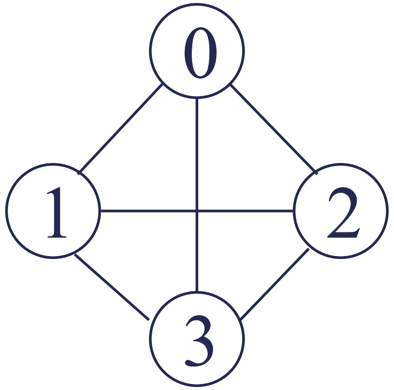
Graph Representation

- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n x n 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

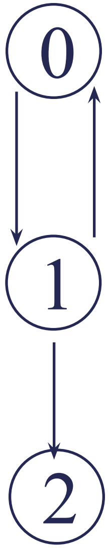
Adjacency Matrix



Graph G_1

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

Adjacency Matrix for Graph G_1



Graph G_2

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

Adjacency Matrix for Graph G_2

Merits: Adjacency Matrix

- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph, the row sum is the out_degree, while the column sum is the in_degree

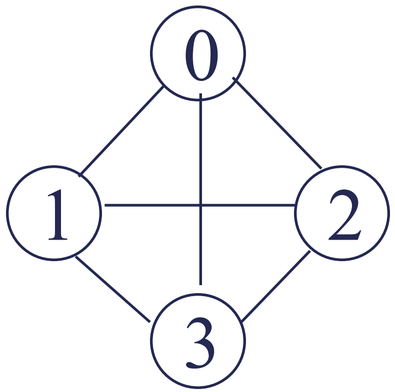
$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

Adjacency List: Interesting Operations

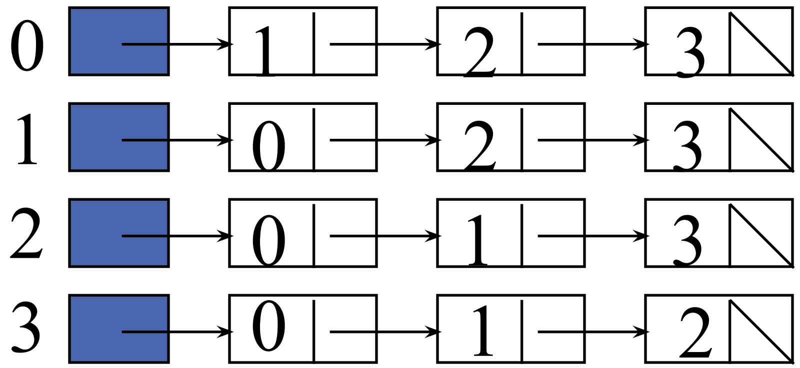
- The degree of any vertex in an undirected graph is determined by counting the no. of nodes in its adjacency list.
- No. of edges in a graph is determined in O(n+e)
- out-degree of a any vertex in a directed graph is determined by counting No. of nodes in its adjacency list.

Adjacency Lists

```
class Gnode
{   int vertex;
    node *next;
    friend class Graph;
};
class Graph
{
private:
    Gnode *Head[20];
    int n;
public:
    Graph()
    {           create head nodes for n vertices
    }
};
```



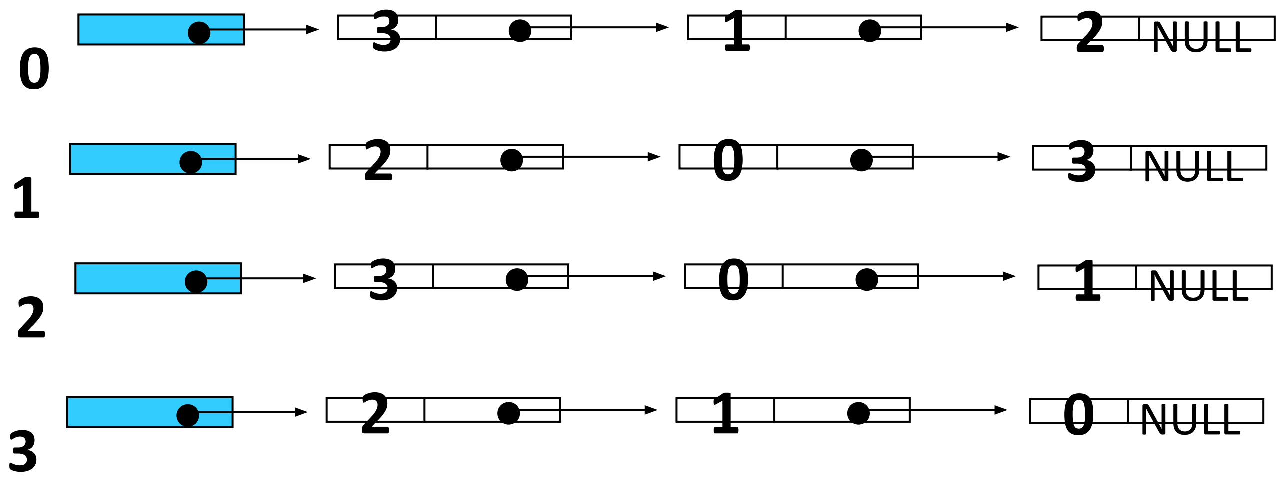
Graph G_1



Adjacency List for Graph G_1

```
graph()
{
  Accept no of vertices;
  for i=0 to n-1
     {Allocate a memory for head[i] node (array)
     head[i]->vertex=i; }
  }

create()
{
  for i=0 to n-1
  {
    temp=head[i];
    do
    {
      Accept adjacent vertex v;
     if(v==i)
        Print Self loop are not allowed;
     else
     {
       Allocate  memory for curr node;
        curr->vertex=v;
        temp->next=curr;
        temp=temp->next;
      }
      accept the choice ;
    }while(ans=='y'||ans=='Y');
  }
}
```



Graph Traversal

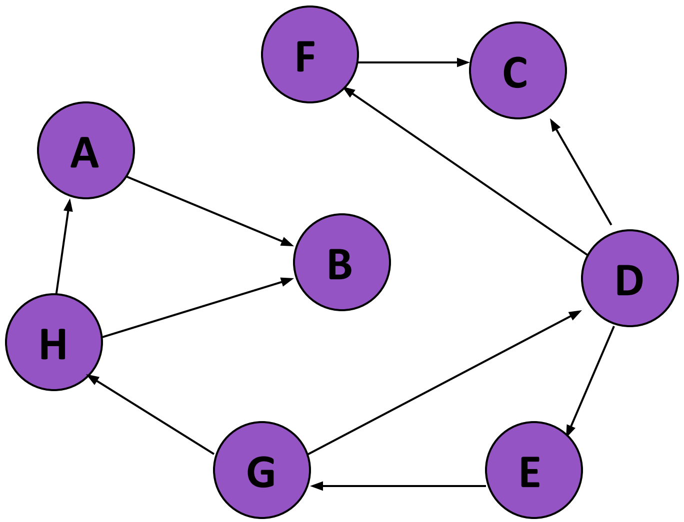
- Depth First Traversal
- Breadth First Traversal

Depth First Traversal (Recursive)

```
Algorithm DFS()
{
  //initially no vertex will be visited
   for( int i=0 ; i<n; i++)
        visited[i]=0;
 //start search at vertex v
    accept starting vertex v
   DFS(v);
}
```

```
Algorithm DFS(int v)
{          print v;
     visited[v]=1;
     for(each vertex w adjacent to v)
          if(!visited[w])
DFS(w);
}
```

Depth First Search Traversal

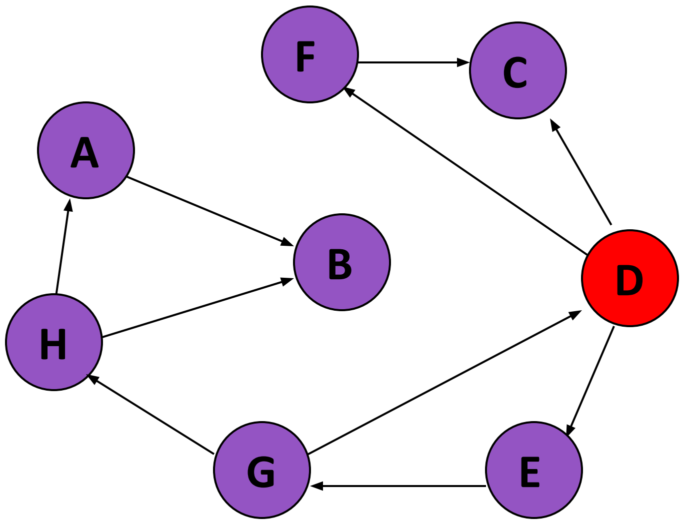


Visited Array

A	
B	
C	
D	
E	
F	
G	
H	

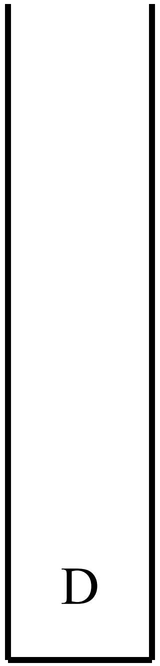
Task: Conduct a depth-first search of the graph starting with node D

Depth First SearchTraversal



Visited Array

A	
B	
C	
D	1
E	
F	
G	
H	

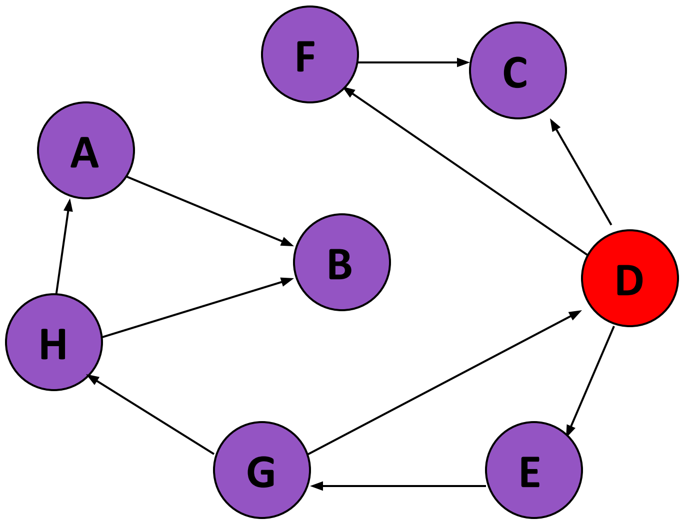


The DFT of nodes in graph :

D

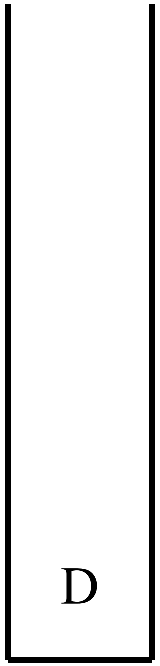
Visit D

Depth First Traversal



Visited Array

A	
B	
C	
D	1
E	
F	
G	
H	

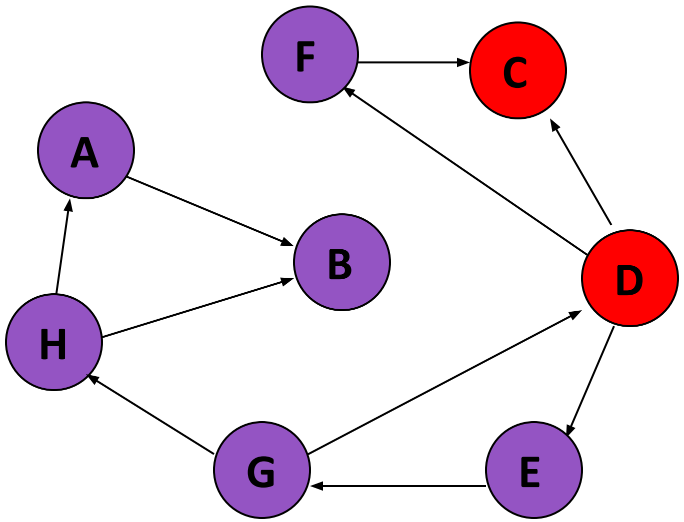


The DFT of nodes in graph :

D

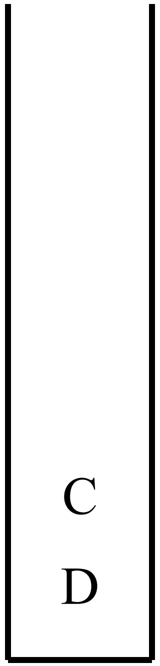
Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order or in order of the adjacency list)

Depth First Traversal



Visited Array

A	
B	
C	1
D	1
E	
F	
G	
H	

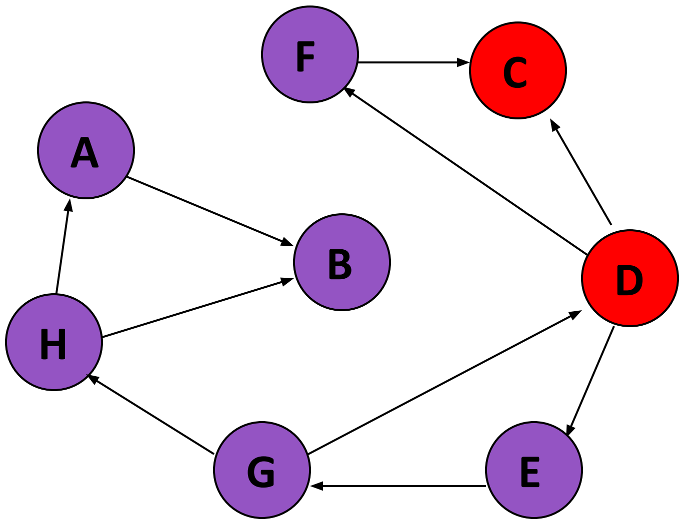


Visit C

The DFT of nodes in graph :

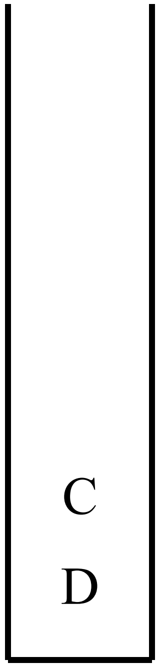
D, C

Depth First Traversal



Visited Array

A	
B	
C	1
D	1
E	
F	
G	
H	

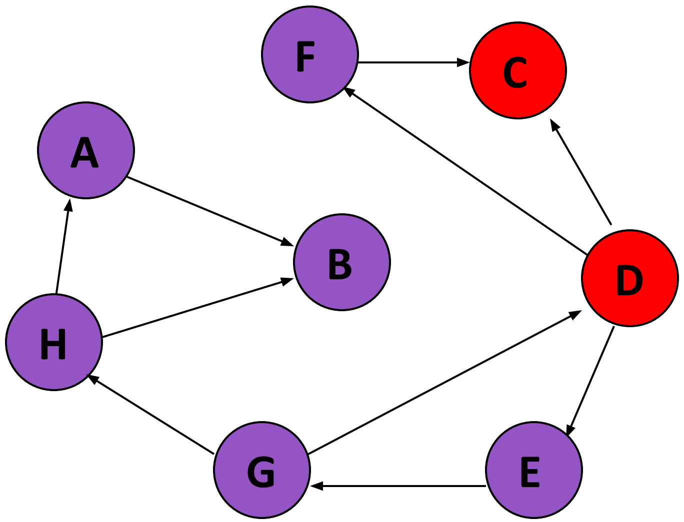


The DFT of nodes in graph :

D, C

No nodes adjacent to C; cannot continue *backtrack*, i.e., pop stack and restore previous state

Depth First Traversal



Visited Array

A	
B	
C	1
D	1
E	
F	
G	
H	

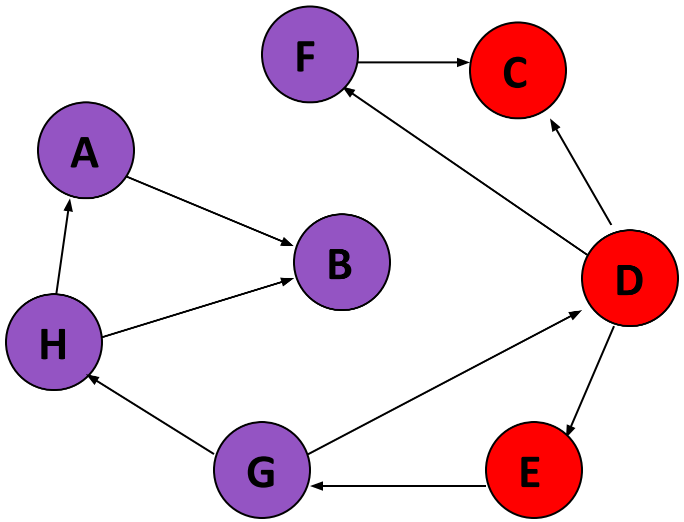
D

The DFT of nodes in graph :

D, C

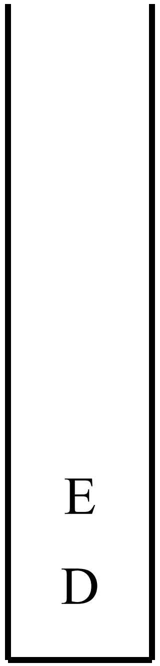
Back to D – C has been visited, decide to visit E next

Depth First Traversal



Visited Array

A	
B	
C	1
D	1
E	1
F	
G	
H	

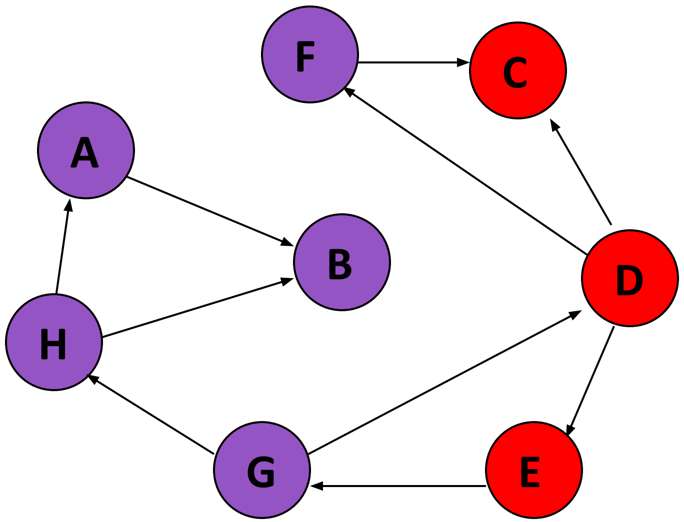


The DFT of nodes in graph :

D, C, E

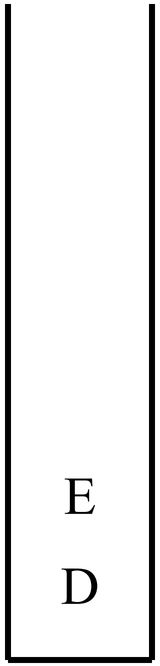
Back to D – C has been visited, decide to visit E next

Depth First Traversal



Visited Array

A	
B	
C	1
D	1
E	1
F	
G	
H	

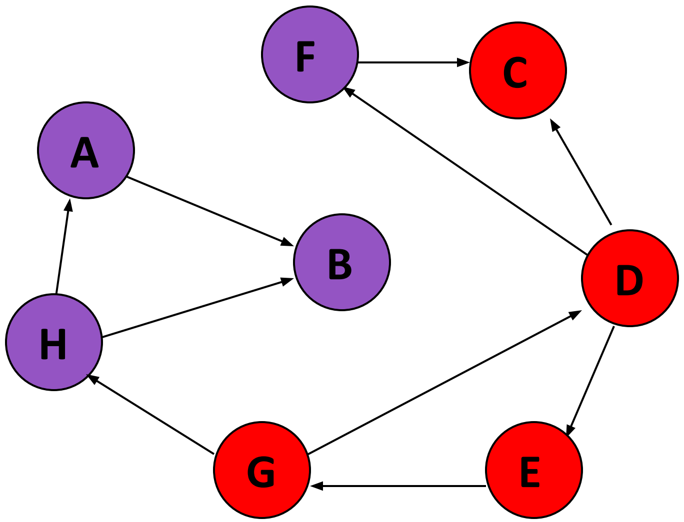


The DFT of nodes in graph :

D, C, E

Only G is adjacent to E

Depth First Traversal



Visited Array

A	
B	
C	1
D	1
E	1
F	
G	1
H	

G

E

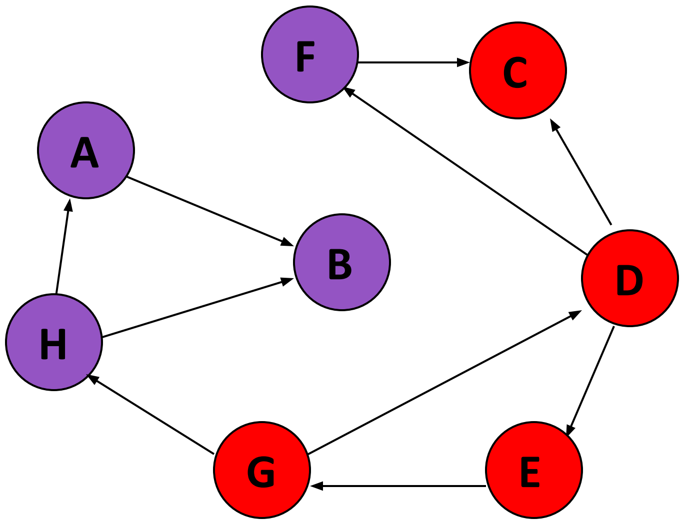
D

Visit G

The DFT of nodes in graph :

D, C, E, G

Depth First Traversal



Visited Array

A	
B	
C	**1**
D	**1**
E	**1**
F	
G	1
H	

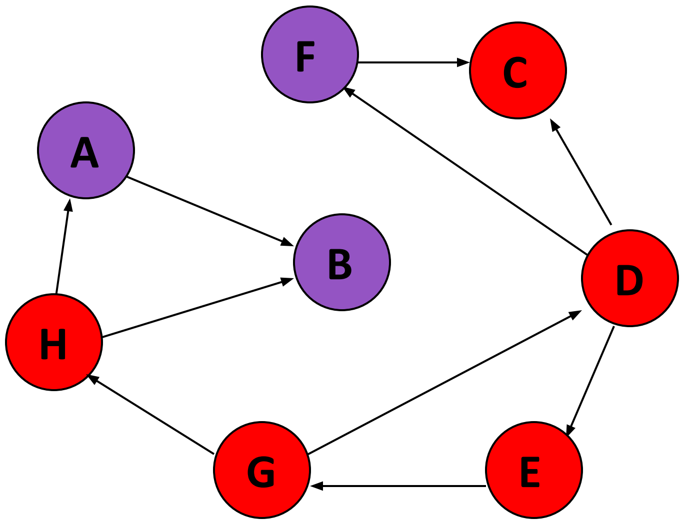
G
E
D

The DFT of nodes in graph :

D, C, E, G

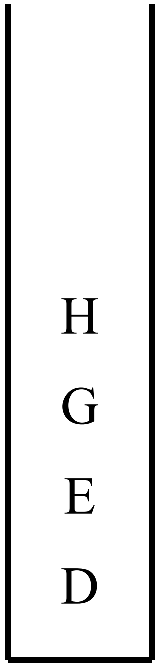
Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.

Depth First Traversal



Visited Array

A	
B	
C	**1**
D	**1**
E	**1**
F	
G	1
H	**1**

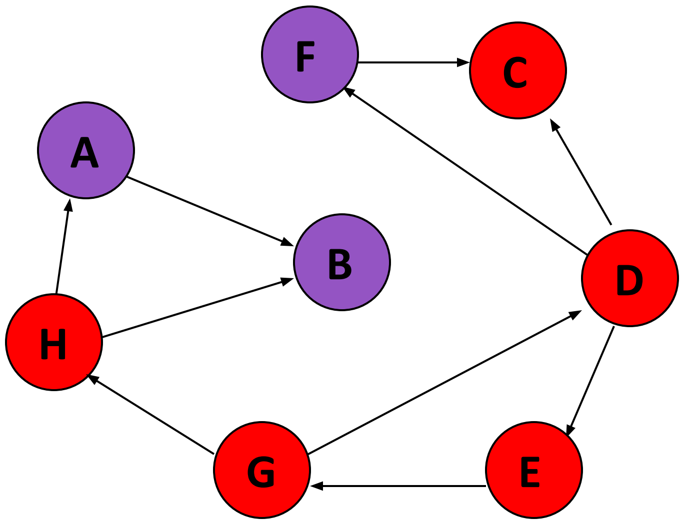


Visit H

The DFT of nodes in graph :

D, C, E, G, H

Depth First Traversal



Visited Array

A	
B	
C	**1**
D	**1**
E	**1**
F	
G	1
H	**1**

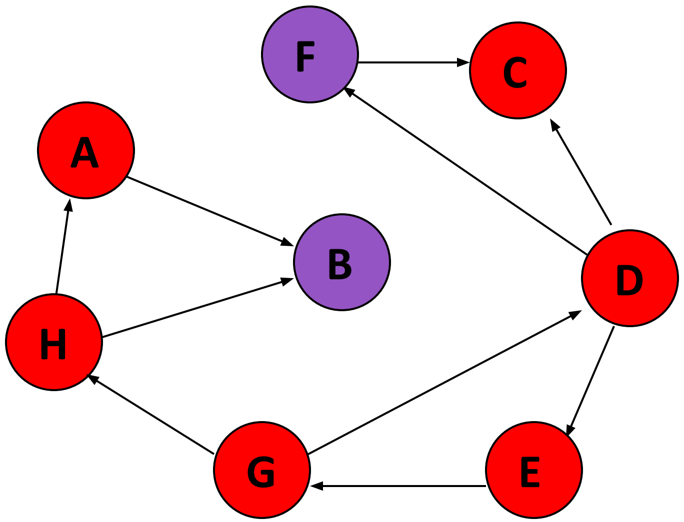
H
G
E
D

The DFT of nodes in graph :

D, C, E, G, H

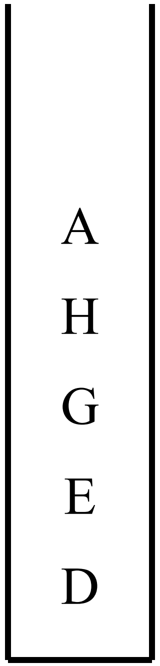
Nodes A and B are adjacent to F. Decide to visit A next.

Depth First Traversal



Visited Array

A	1
B	
C	1
D	1
E	1
F	
G	1
H	1

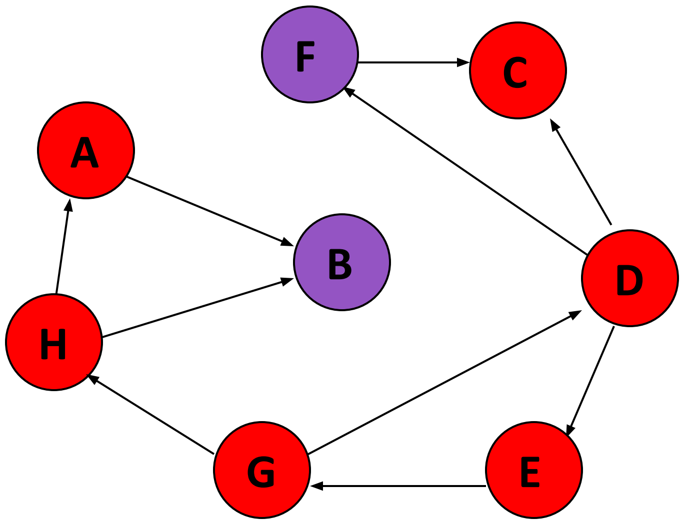


Visit A

The DFT of nodes in graph :

D, C, E, G, H, A

Depth First Traversal



Visited Array

A	1
B	
C	1
D	1
E	1
F	
G	1
H	1

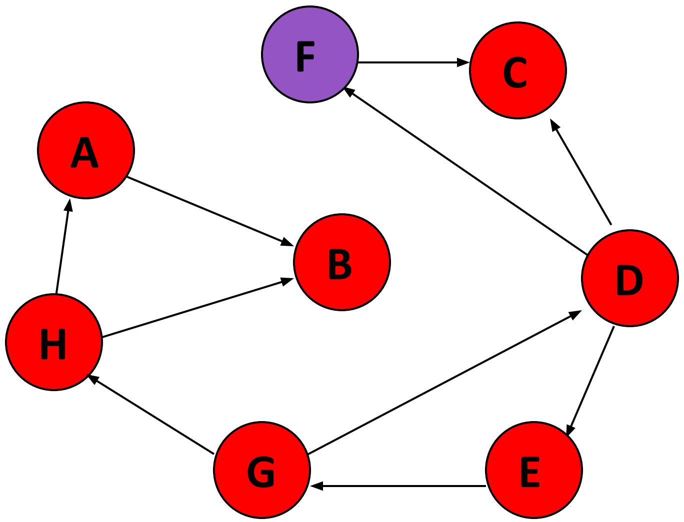
A
H
G
E
D

The DFT of nodes in graph :

D, C, E, G, H, A

Only Node B is adjacent to A.
Decide to visit B next.

Depth First Traversal



Visited Array

A	1
B	1
C	1
D	1
E	1
F	
G	1
H	1

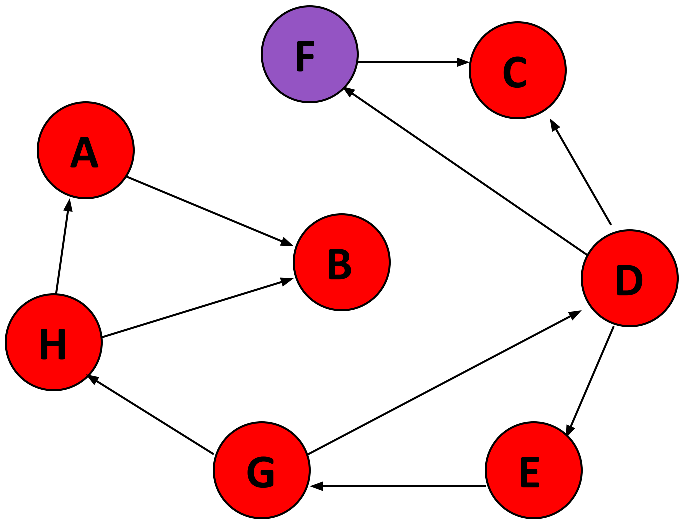
B
A
H
G
E
D

Visit B

The DFT of nodes in graph :

D, C, E, G, H, A, B

Depth First Traversal



Visited Array

A	1
B	1
C	1
D	1
E	1
F	
G	1
H	1

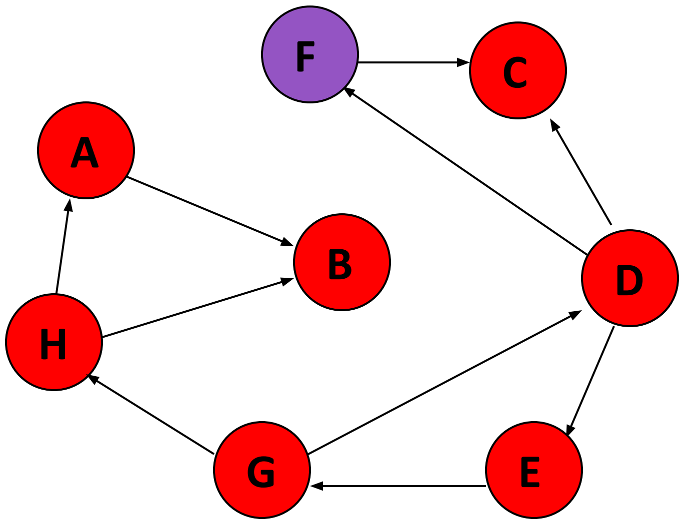
A
H
G
E
D

The DFT of nodes in graph :

D, C, E, G, H, A, B

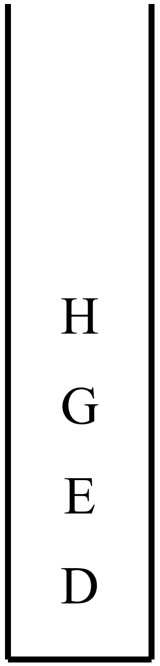
No unvisited nodes adjacent to B. Backtrack (pop the stack).

Depth First Traversal



Visited Array

A	1
B	1
C	1
D	1
E	1
F	
G	1
H	1

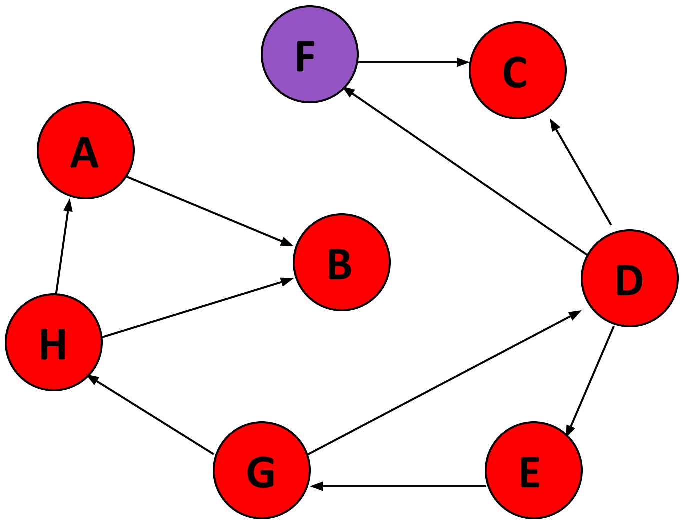


The DFT of nodes in graph :

D, C, E, G, H, A, B

No unvisited nodes adjacent to A. Backtrack (pop the stack).

Depth First Traversal



Visited Array

A	1
B	1
C	1
D	1
E	1
F	
G	1
H	1

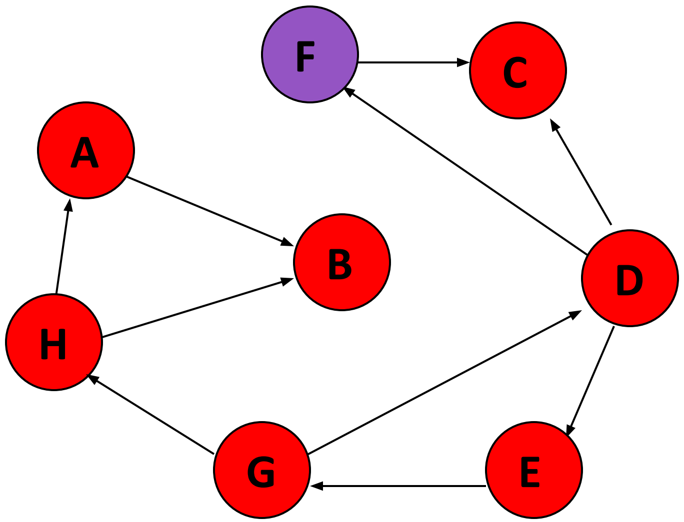
G
E
D

The DFT of nodes in graph :

D, C, E, G, H, A, B

No unvisited nodes adjacent to H. Backtrack (pop the stack).

Depth First Traversal



Visited Array

A	1
B	1
C	1
D	1
E	1
F	
G	1
H	1

E

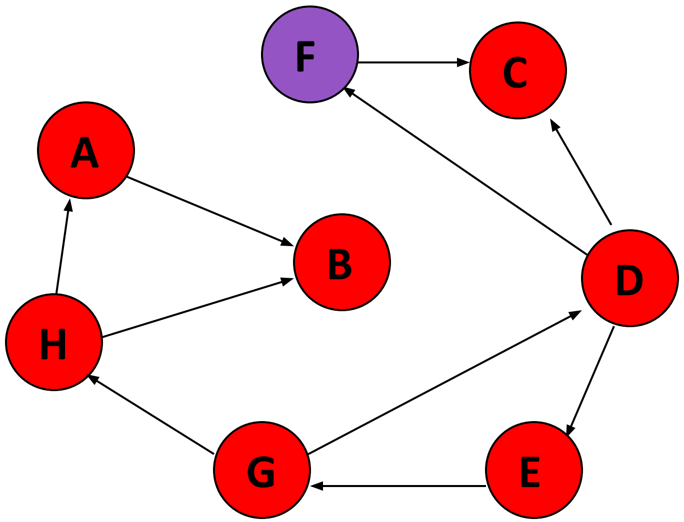
D

The DFT of nodes in graph :

D, C, E, G, H, A, B

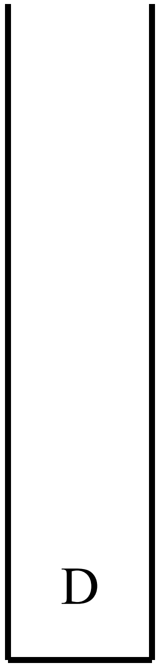
No unvisited nodes adjacent to G.
Backtrack (pop the stack).

Depth First Traversal



Visited Array

A	1
B	1
C	1
D	1
E	1
F	
G	1
H	1

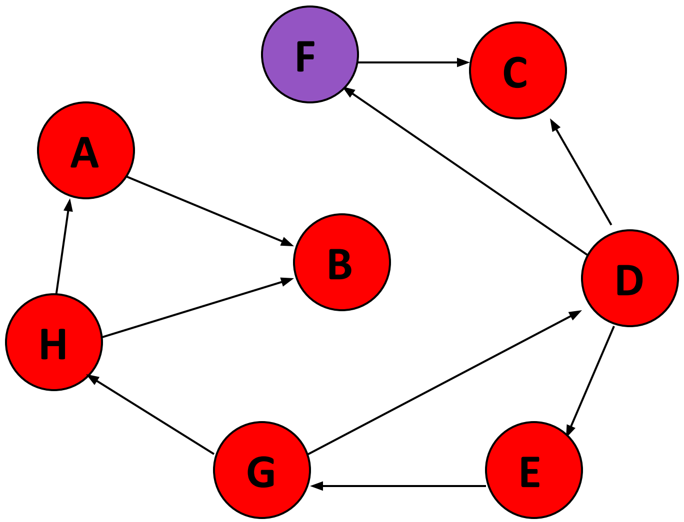


The DFT of nodes in graph :

D, C, E, G, H, A, B

No unvisited nodes adjacent to E. Backtrack (pop the stack).

Depth First Traversal



Visited Array

A	1
B	1
C	1
D	1
E	1
F	
G	1
H	1

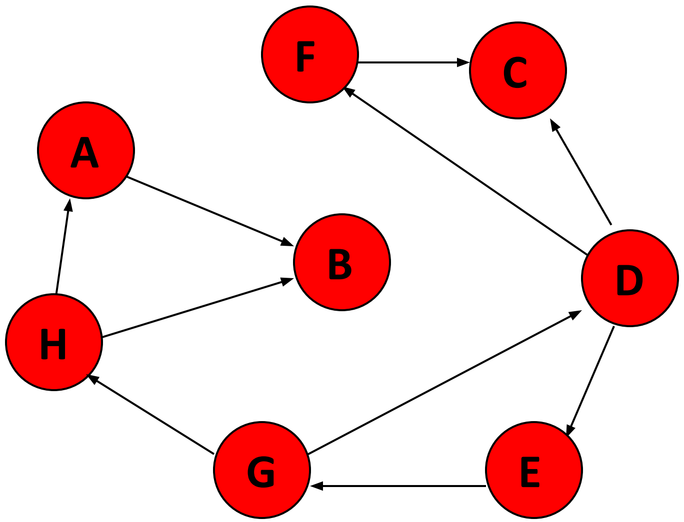
D

The DFT of nodes in graph :

D, C, E, G, H, A, B

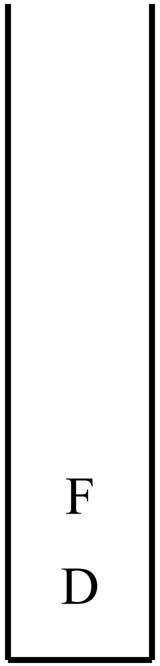
F is unvisited and is adjacent to D. Decide to visit F next.

Depth First Traversal



Visited Array

A	**1**
B	**1**
C	**1**
D	**1**
E	**1**
F	1
G	1
H	**1**

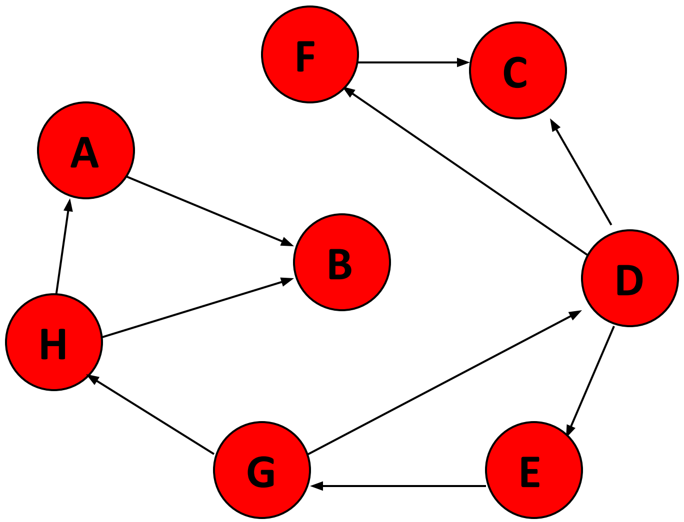


Visit F

The DFT of nodes in graph :

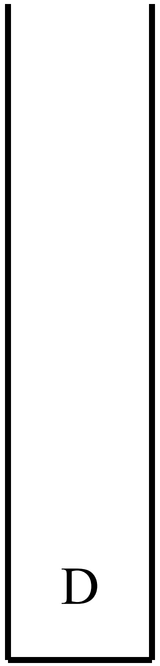
D, C, E, G, H, A, B, F

Depth First Traversal



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

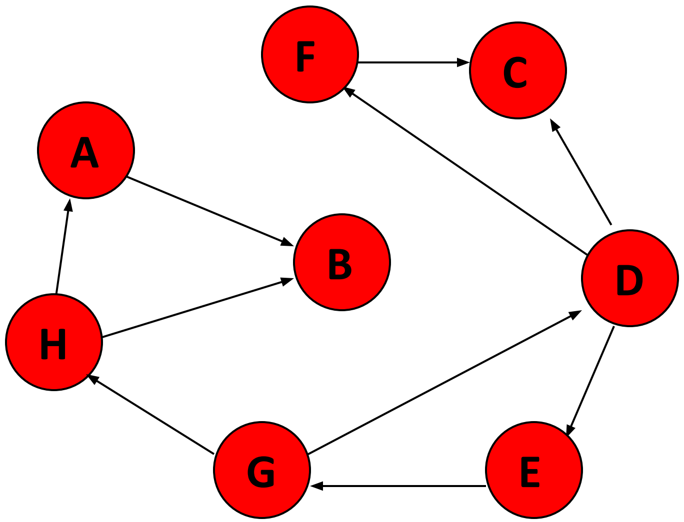


The DFT of nodes in graph :

D, C, E, G, H, A, B, F

No unvisited nodes adjacent to F. Backtrack.

Depth First Traversal



Visited Array

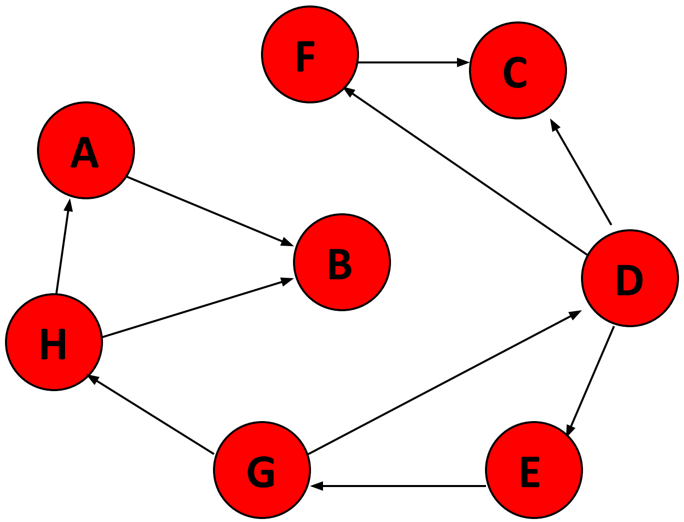
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

The order nodes are visited:

D, C, E, G, H, A, B, F

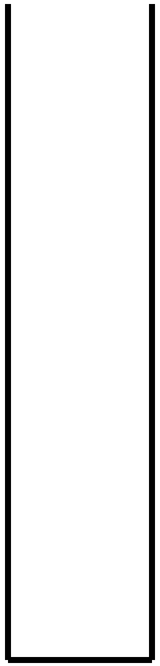
No unvisited nodes adjacent to D. Backtrack.

Depth First Traversal



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√



The DFT of nodes in graph :

D, C, E, G, H, A, B, F

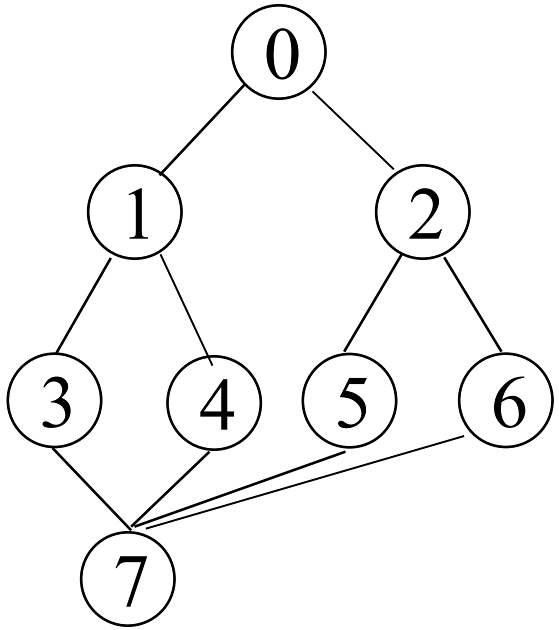
Stack is empty. Depth-first traversal is done.

Depth First Traversal (Non-recursive)

```
Algorithm DFS(int v)
{
  for all vertices of graph
         visited[i]=0;
   push(v);
   visited[v]=1;
   do
    {
     v=pop();
     print(v);
      for(each vertex w adjacent to v)
 {
     if(!visited[w])
               {  push(w); visited[w]=1;}
 } //end for
 } while(stack not empty)
 } //end dfs
```

Depth First Traversal

Find DFT for given graph G1 starting at vertex 0

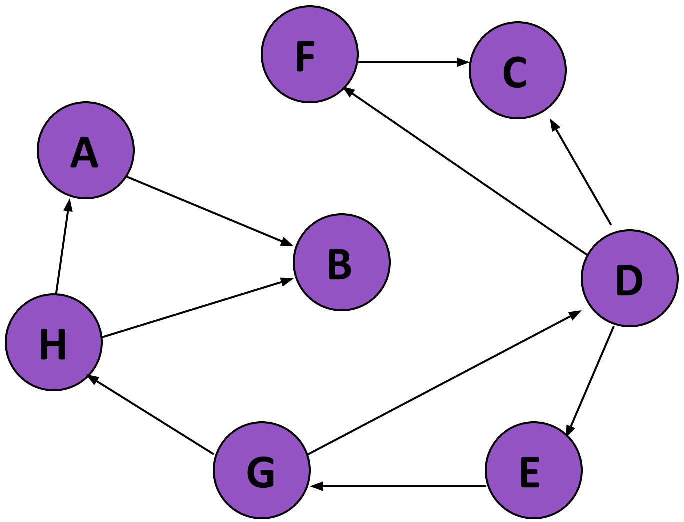


Graph G1

Breadth First Traversal

```
Algorithm BFS(int v) {
 for(int i=0;i<n;i++)
       visited[i]=0;
   Queue  q;
   q.insert(v);
   while(!q.IsEmpty())
   {
          v=q.Delete();
          for(all vertices w adjacent to v)
          if(!visited[w])
          {
                 q.insert(w);
                 visited[w]=1;
          }
   }
}
```

Breadth First Traversal



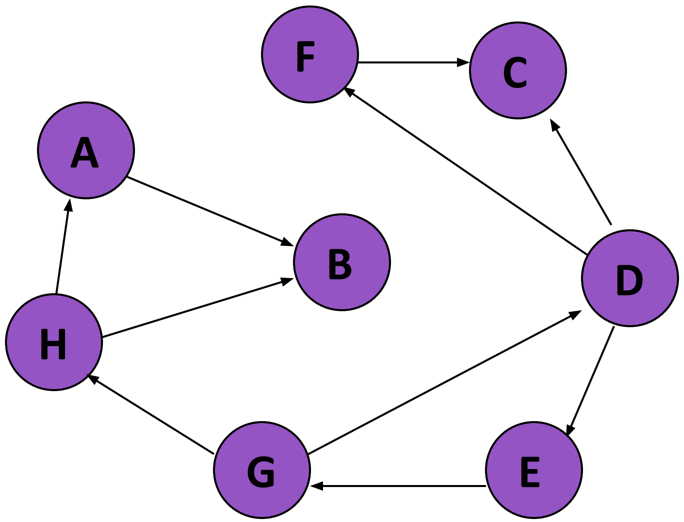
Enqueued Array

A	
B	
C	
D	
E	
F	
G	
H	

Q:

How is this accomplished? Simply replace the stack with a queue! Rules: (1) Maintain an *enqueued* array. (2) Visit node when *dequeued*.

Breadth First Traversal



Enqueued Array

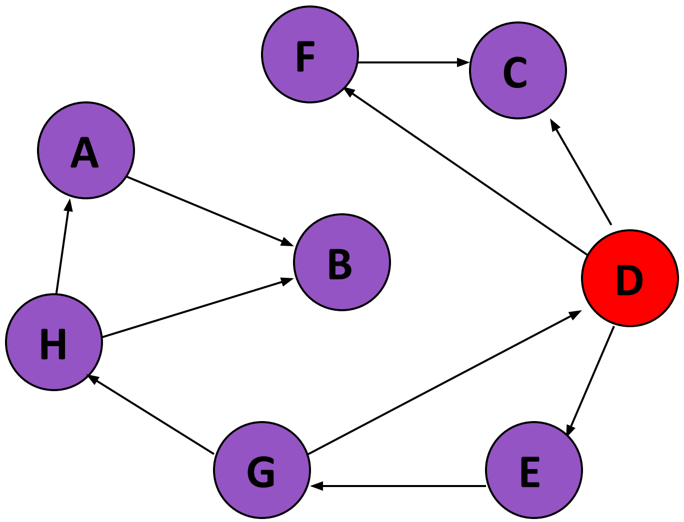
A	
B	
C	
D	√
E	
F	
G	
H	

Q :D

Nodes visited:

Enqueue D. Notice, D not yet visited.

Breadth First Traversal



Enqueued Array

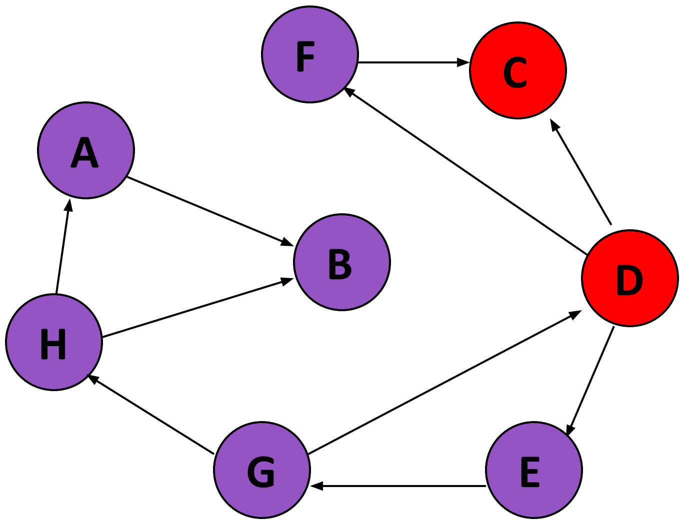
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q : C , E, F

Nodes visited: D

Dequeue D. Visit D. Enqueue unenqueued nodes adjacent to D.

Breadth First Traversal



Enqueued Array

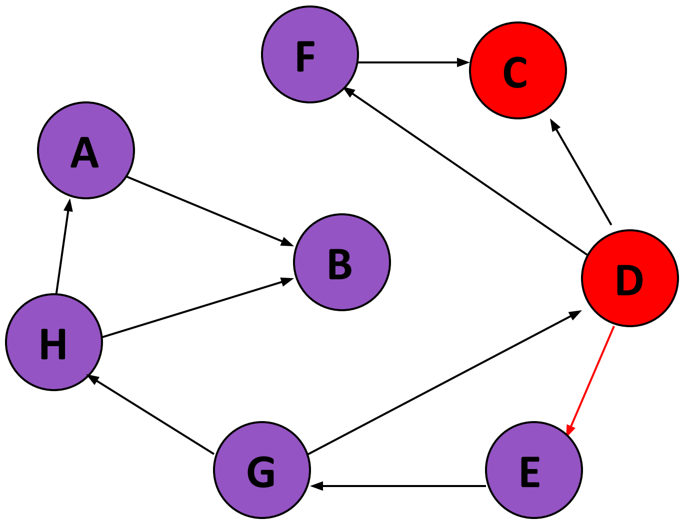
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q : E , F

Nodes visited: D, C

Dequeue C. Visit C. Enqueue unenqueued nodes adjacent to C.

Breadth First Traversal



Enqueued Array

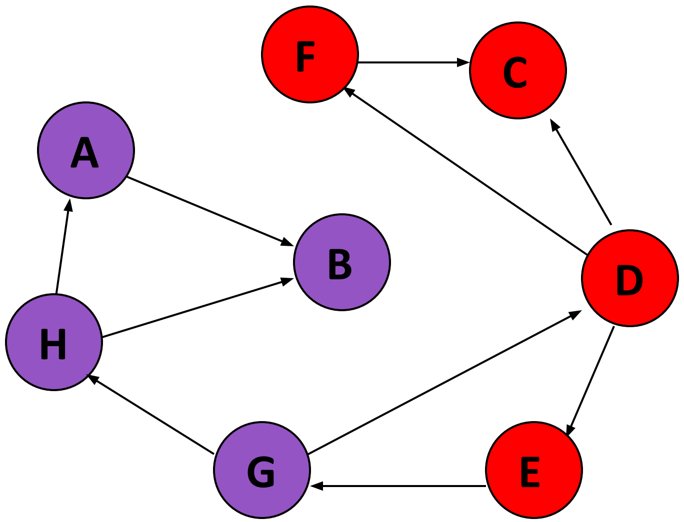
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q : F , G

Nodes visited: D, C, E

Dequeue E. Visit E. Enqueue unenqueued nodes adjacent to E.

Breadth First Traversal



Enqueued Array

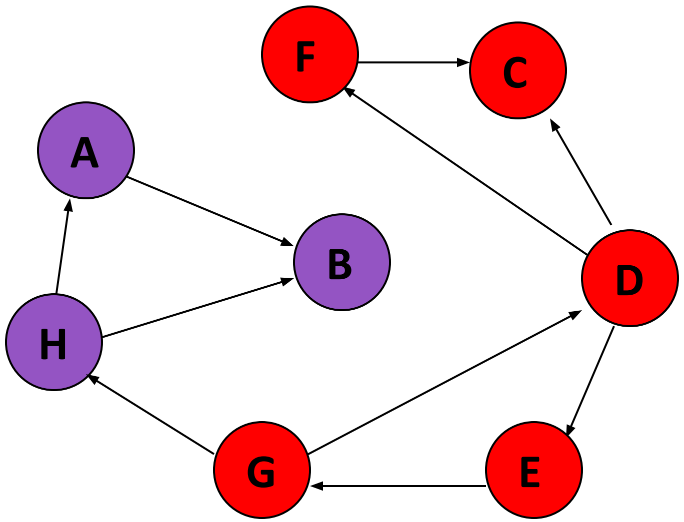
A	
B	
C	√
D	√
E	√
F	√
G	√
H	

Q : G

Nodes visited: D, C, E, F

Dequeue F. Visit F. Enqueue unenqueued nodes adjacent to F.

Breadth First Traversal



Enqueued Array

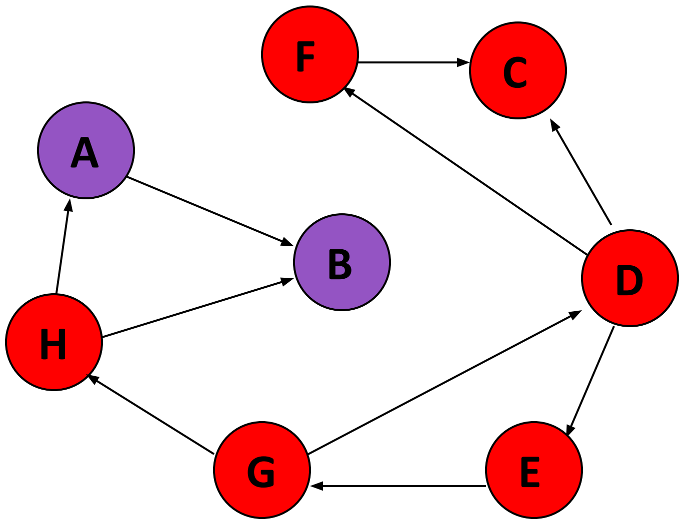
A	
B	
C	√
D	√
E	√
F	√
G	√
H	√

Q : H

Nodes visited: D, C, E, F, G

Dequeue G. Visit G. Enqueue unenqueued nodes adjacent to G.

Breadth First Traversal



Enqueued Array

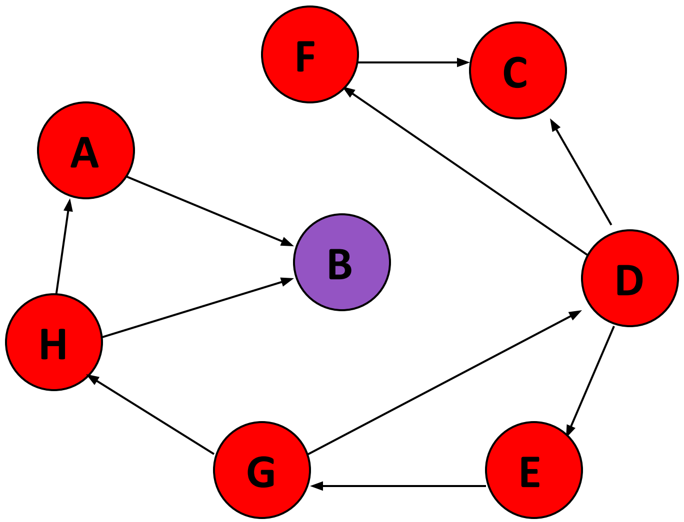
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

Q : A , B

Nodes visited: D, C, E, F, G, H

Dequeue H. Visit H. Enqueue unenqueued nodes adjacent to H.

Breadth First Traversal



Enqueued Array

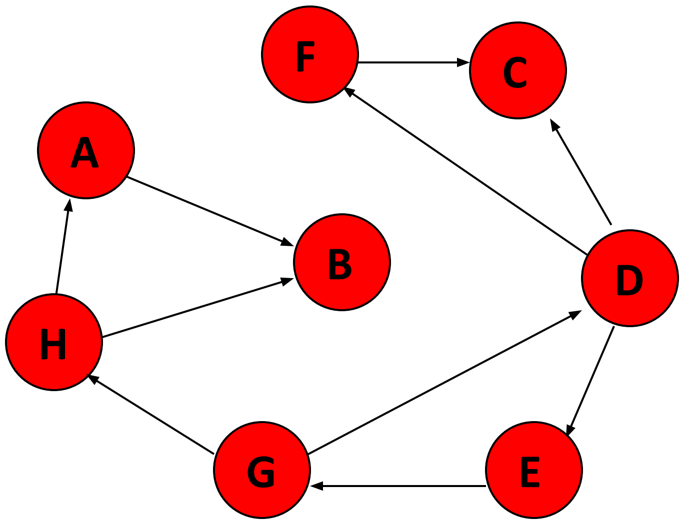
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

Q : B

Nodes visited: D, C, E, F, G, H, A

Dequeue A. Visit A. Enqueue unenqueued nodes adjacent to A.

Breadth First Traversal



Enqueued Array

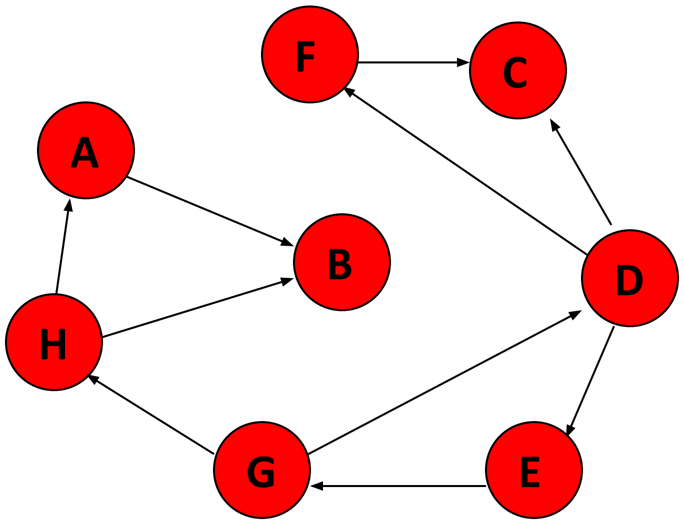
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

Q empty

Nodes visited: D, C, E, F, G, H, A, B

Dequeue B. Visit B. Enqueue unenqueued nodes adjacent to B.

Breadth First Traversal



Enqueued Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

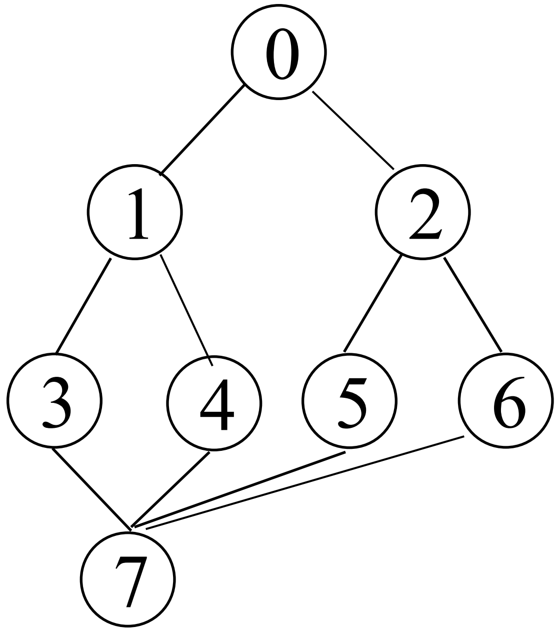
Q empty

Nodes visited: D, C, E, F, G, H, A, B

Q empty. Algorithm done.

Breadth First Traversal

Find BFT for given graph G1 starting at vertex 0

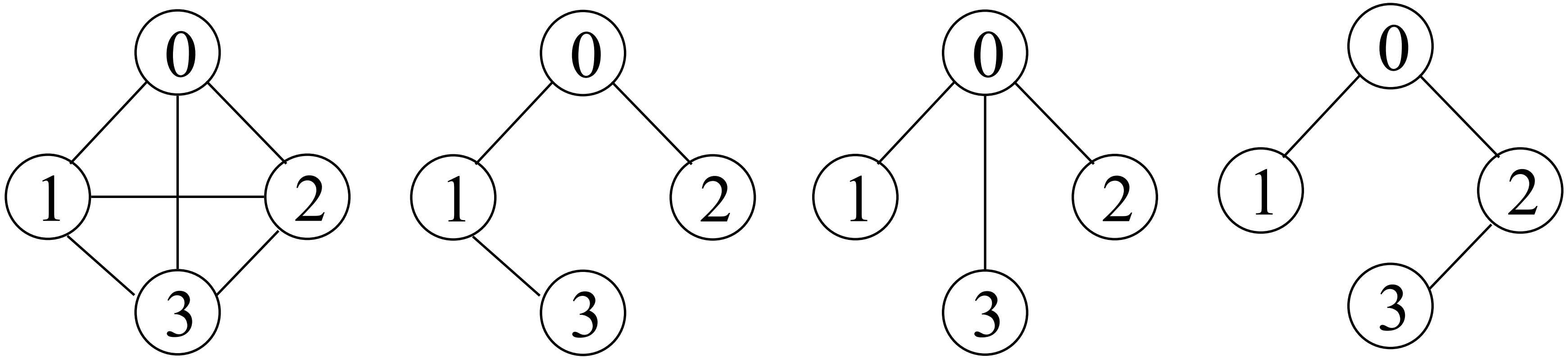


Graph G1

Spanning Trees

- A spanning tree is any tree that consists solely of edges in G and that includes all the vertices
- A spanning tree is a minimal subgraph, G’, of G such that V(G’)=V(G) and G’ is connected.
- 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

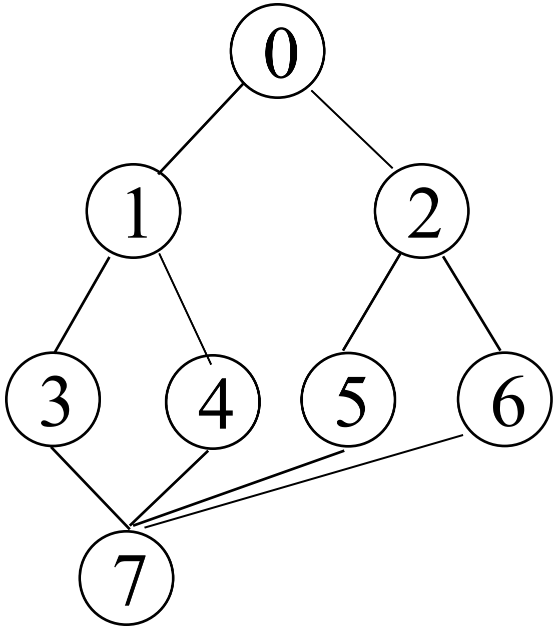
Examples of Spanning Trees



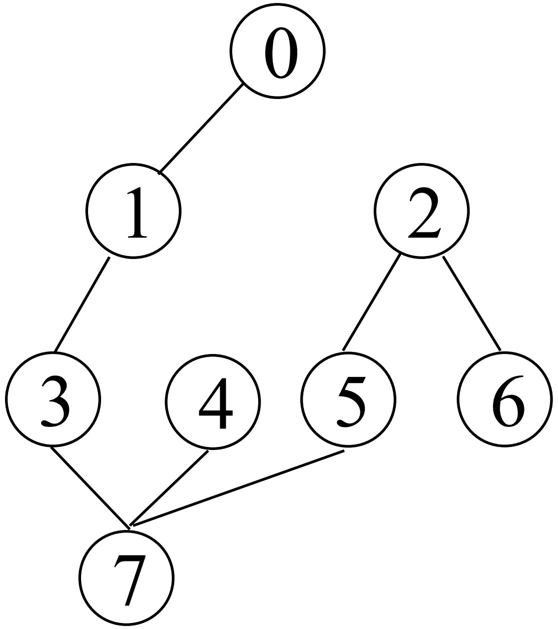
Graph G_1

Possible spanning trees

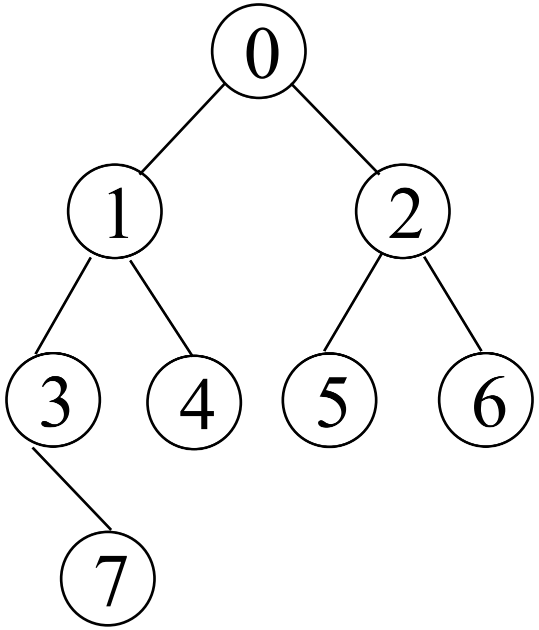
DFS VS BFS Spanning Trees



Graph



DFS Spanning Tree



BFS Spanning Tree

Minimum 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

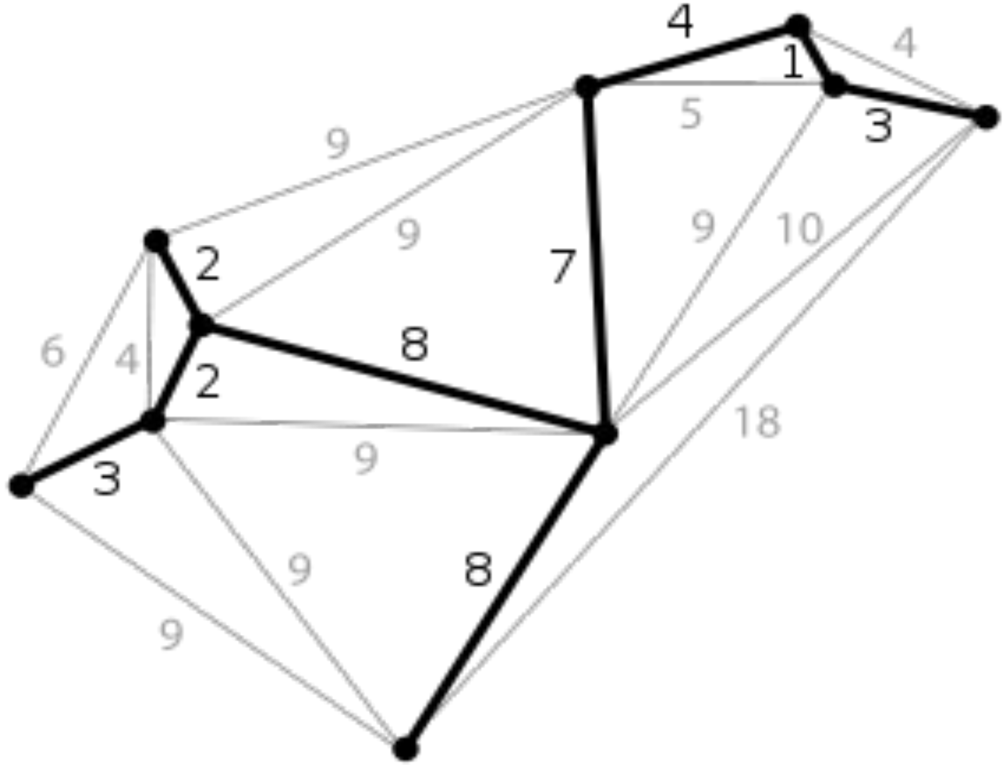
- A minimum cost spanning tree is a spanning tree of least cost

- n-1 edges from a weighted graph of n vertices with minimum cost.

- Two different algorithms can be used
  - Kruskal
  - Prim

Minimum Spanning Tree

- Applications of MST in Network design
  - Telephone
  - Electrical
  - TV cable
  - Computer
  - road



Greedy Strategy

- An optimal solution is constructed in stages
- At each stage, the best decision is made at this time
- Since this decision cannot be changed later, we make sure that the decision will result in a feasible solution
- Typically, the selection of an item at each stage is based on a least cost or a highest profit criterion

Kruskal's Algorithm

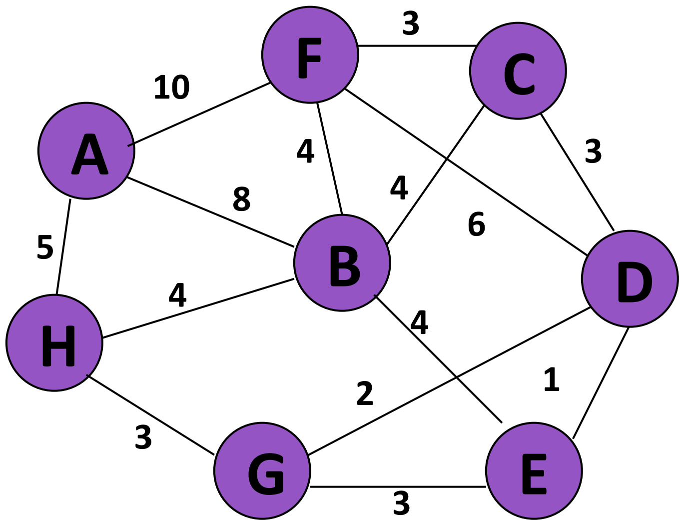
- Build a minimum cost spanning tree T by adding edges to T one at a time
- Select the edges for inclusion in T in nondecreasing order of the cost
- An edge is added to T if it does not form a cycle
- Since G is connected and has $n > 0$ vertices, exactly $n-1$ edges will be selected

Kruskal’s Algorithm

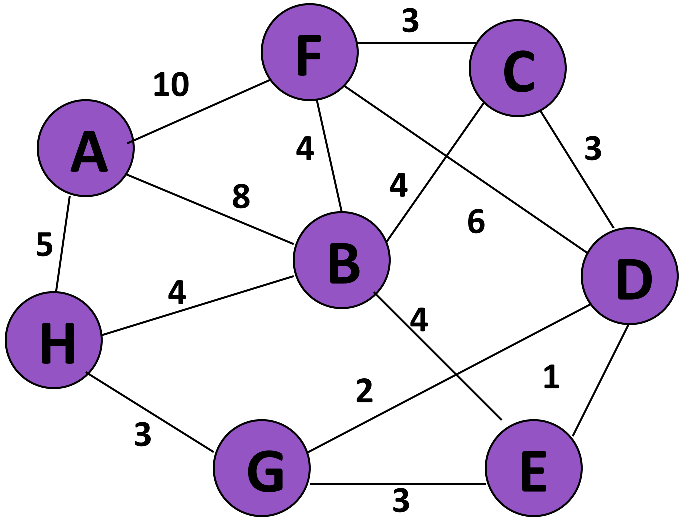
```
T= {};
while (T contains less than n-1 edges  && E is not empty)
{
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
    add (v,w) to T
 else
      discard (v,w);
}
if (T contains fewer than n-1 edges)
  printf("No spanning tree\n");
```

Kruskal’s Algorithm

Consider an undirected, weight graph



Kruskal's Algorithm

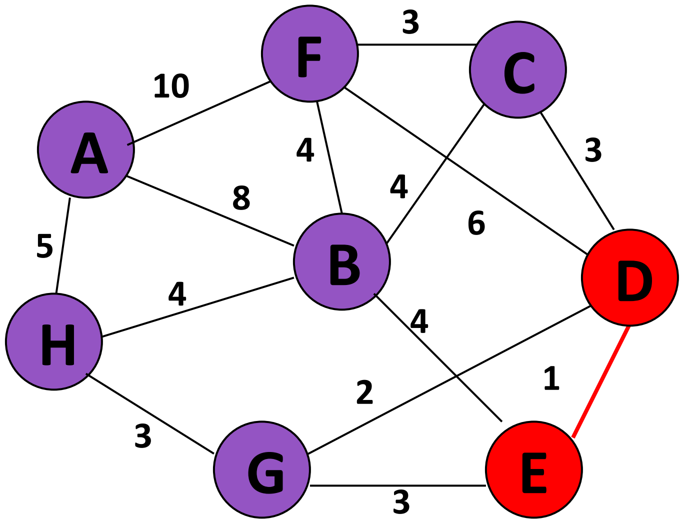


Sort the edges by increasing edge weight

edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's Algorithm

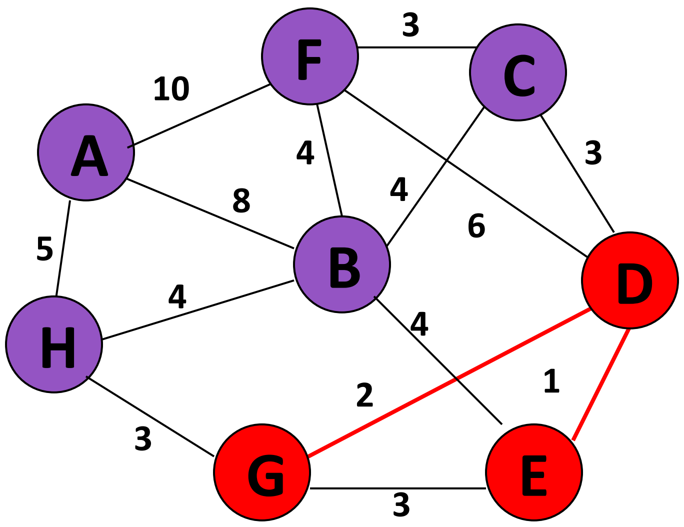


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's Algorithm

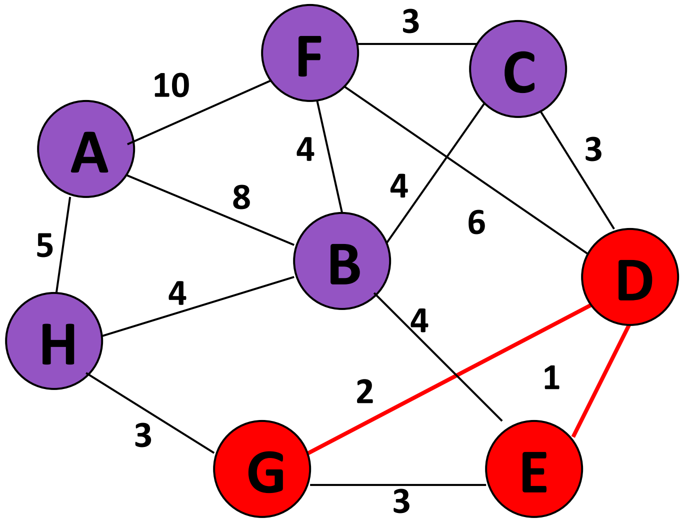


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's Algorithm



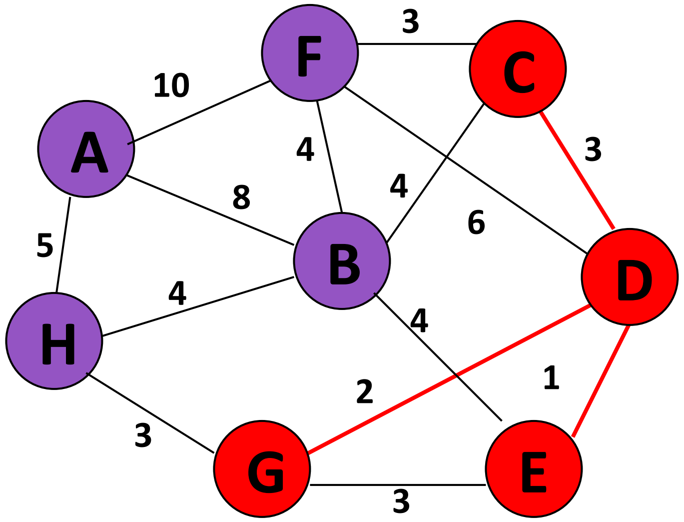
Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Accepting edge (E,G) would create a cycle

Kruskal's Algorithm

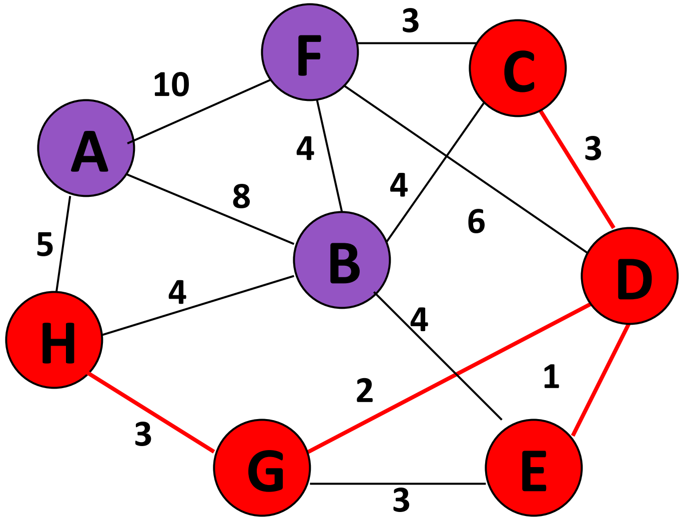


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal’s Algorithm

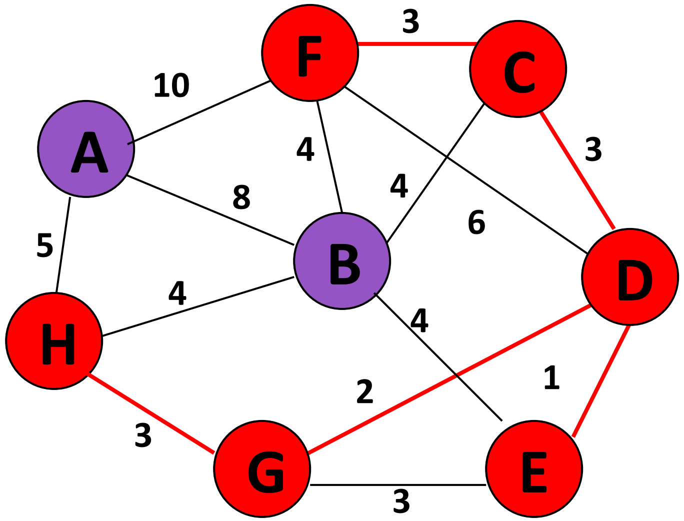


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's Algorithm



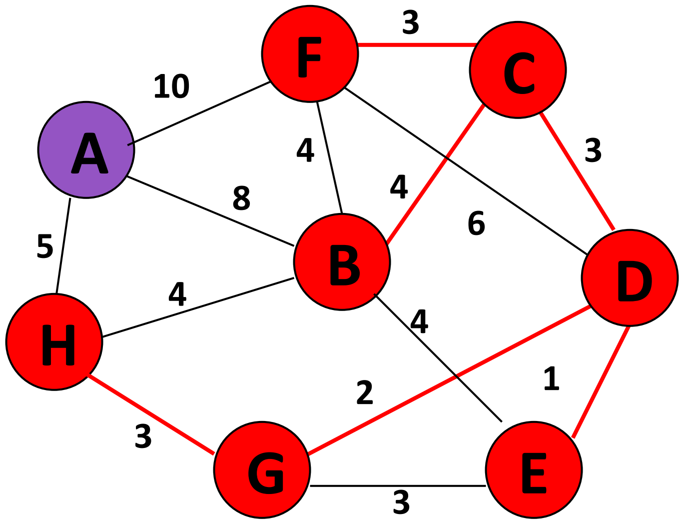
Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal’s Algorithm

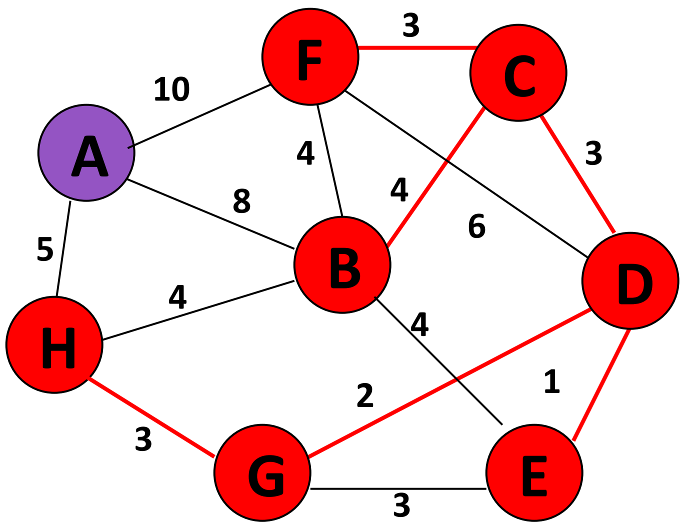
Select first |V|–1 edges which do not generate a cycle



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's Algorithm

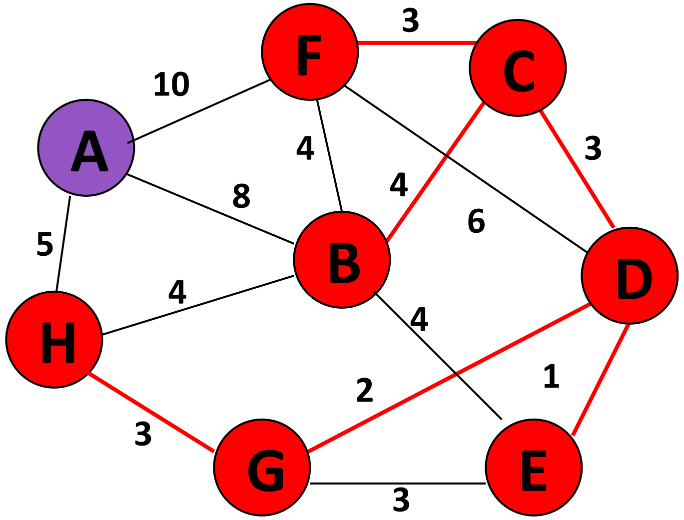


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's Algorithm

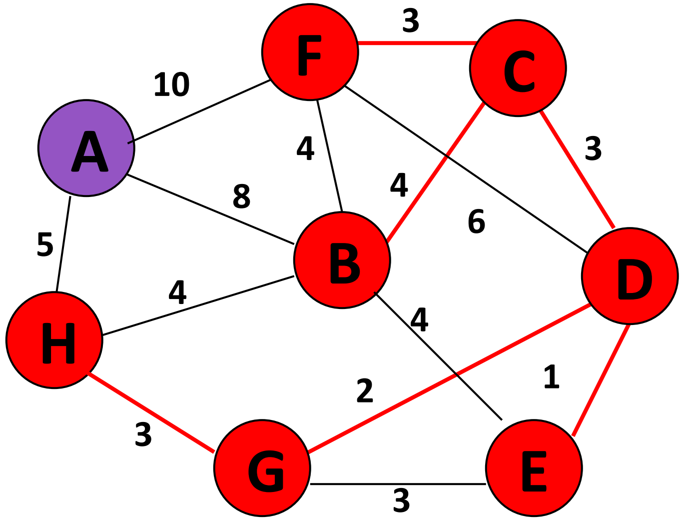


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's Algorithm

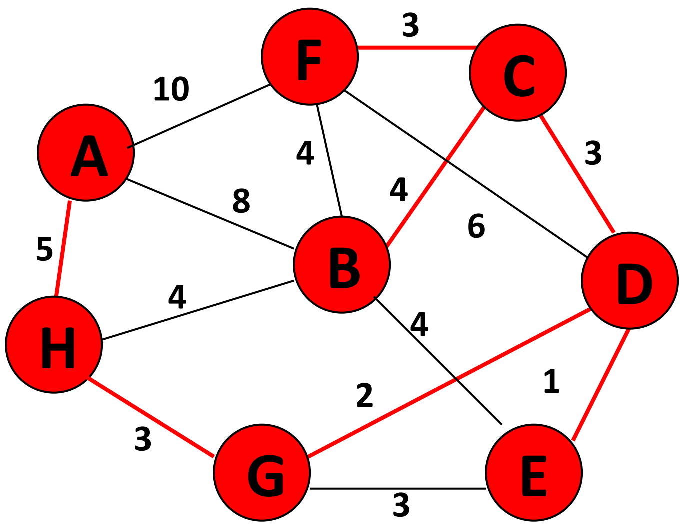


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's Algorithm



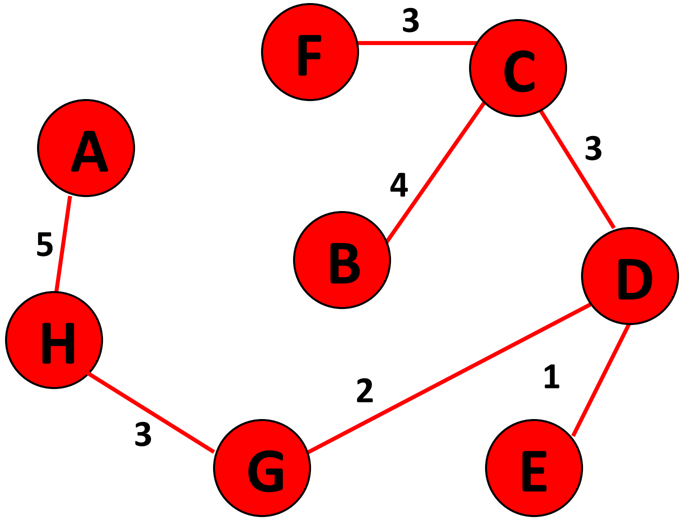
Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

Kruskal's Algorithm

Select first |V|–1 edges which do not generate a cycle



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

} not considered

Done

Total Cost = $\Sigma\, d_v = 21$

Prim’s Algorithm

//Assume G has at least one vertex
TV={0}; *//start with vertex 0 and no edges*

for (T=Ø; T contains less than n-1 edges; add(u,v) to T)
{
  let (u,v) be a least cost edge such that u $\in$ TV and v $\notin$ TV;
  if (there is no such edge) break;
  add v to TV;
}

if (T contains fewer than n-1 edges)
cout<<“No spanning tree\n”;

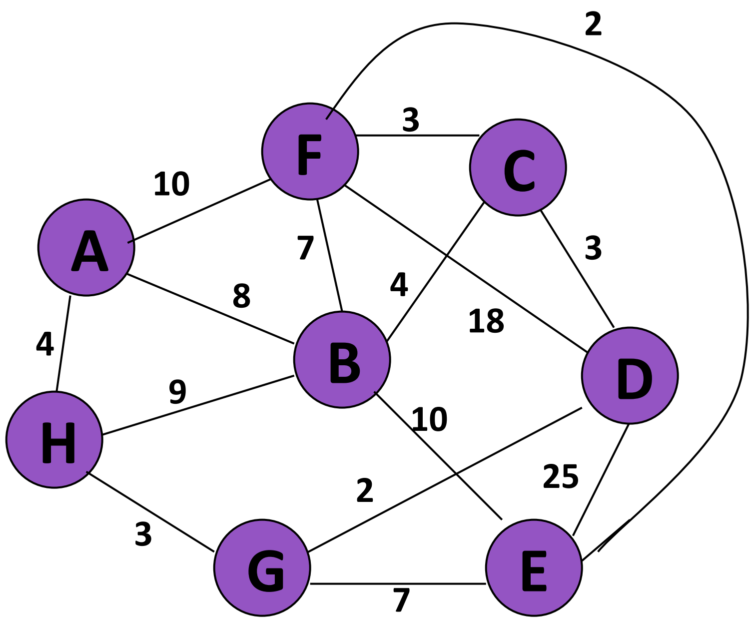
Prim’s Algorithm

```
Algorithm prims(start_v){
 //cost[i][j] is either +ve or infinity.
//A MST is  computed & stored as a set of edges in the
//array  t[n][1].  t[i][0], t[i][1]) is an edge in the MST
//where 0<i<n.
// start_v be the starting vertex
{
//Initialize nearest
 nearest [start_v] =-1;
 for i=0 to n-1 do
 {   if(i!=start_v)
                    nearest[i]= start_v;
 }
 r=0;

for  i=1 to n-1 do
{  //find n-1 additional edges for t
    min= ∞
   for k=0 to n-1
   {  // find  j :  vertex such that;
      if (nearest[k]!= -1 and cost[k, nearest[k]] <min)
             {  j=k ; min= cost[k, nearest[k]];}
    }
//update tree and total cost
   t[ r][0]=j , t[r][1]=nearest[j];  r=r+1;
   mincost = mincost +cost[j, nearest[j]);
   nearest[j]=-1;

//update nearest for remaining vertices
   for k=0 to n-1
   {            if(nearest[k]!= -1 and (cost[k, nearest[k])> cost[k, j]
                  nearest[k]=j;
   }
    return mincost;
  }  //end for i=1 to n-1
}
```

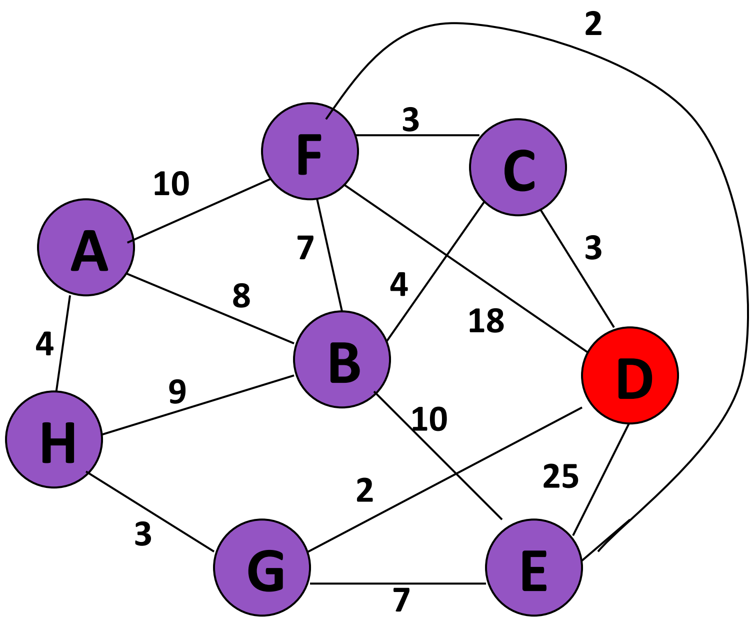
Prim's Algorithm



Initialize array

	K	d_v	p_v
A	F	∞	–
B	F	∞	–
C	F	∞	–
D	F	∞	–
E	F	∞	–
F	F	∞	–
G	F	∞	–
H	F	∞	–

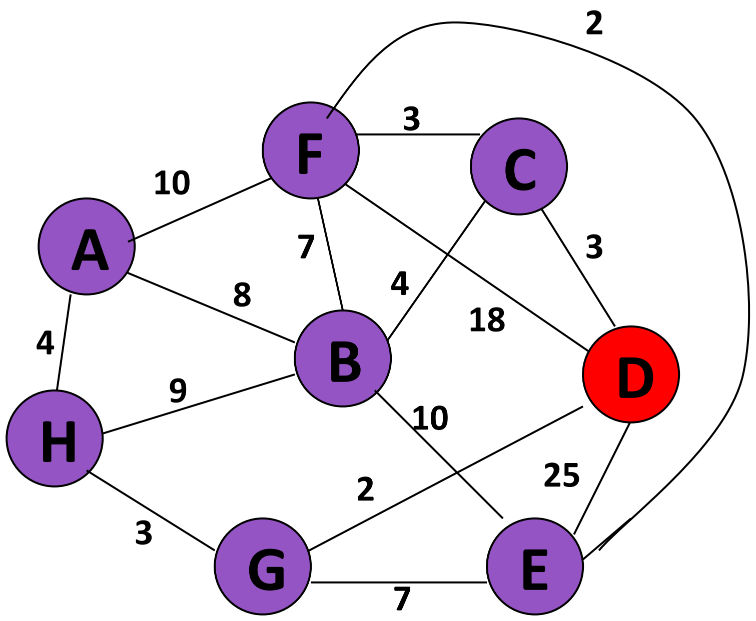
Prim’s Algorithm



Start with any node, say D

	K	d_v	p_v
A			
B			
C			
D	T	0	–
E			
F			
G			
H			

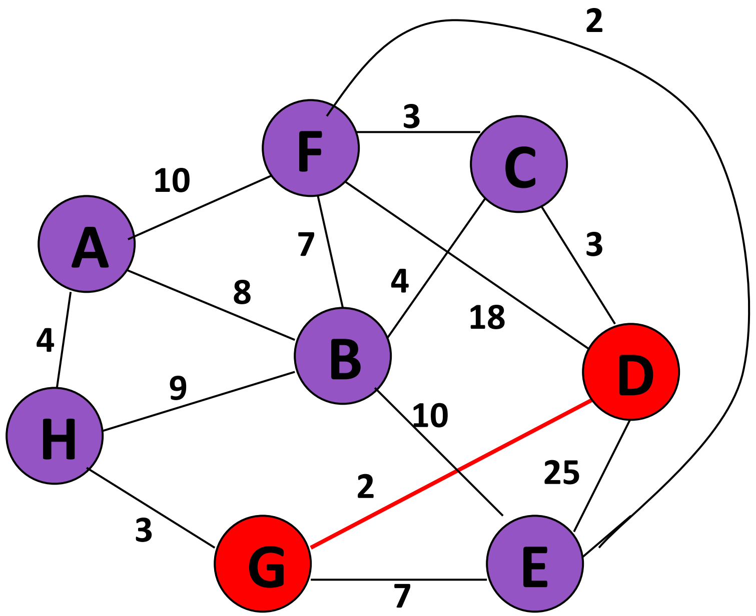
Prim’s Algorithm



Update distances of
adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G		2	D
H			

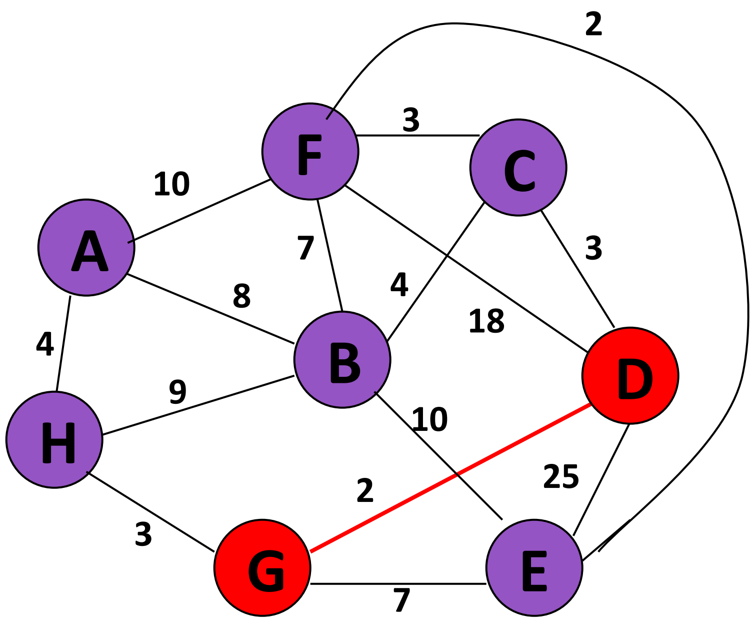
Prim’s Algorithm



Select node with minimum distance

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G	T	2	D
H			

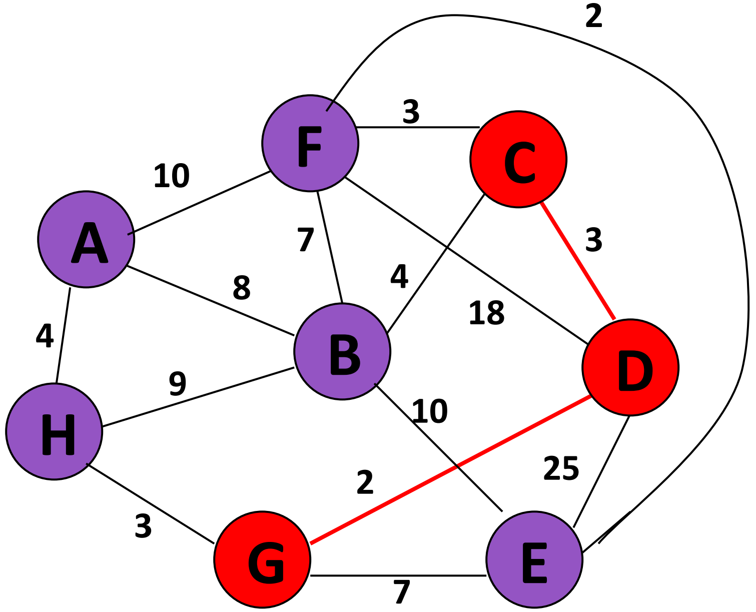
Prim's Algorithm



Update distances of adjacent, unselected nodes

	K	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G

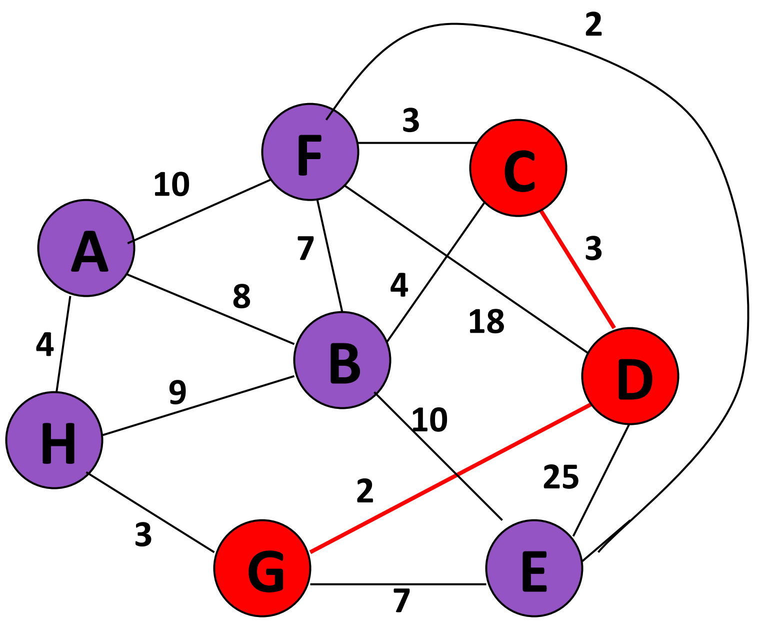
Prim's Algorithm



Select node with minimum distance

	K	d_v	p_v
A			
B			
C	T	3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G

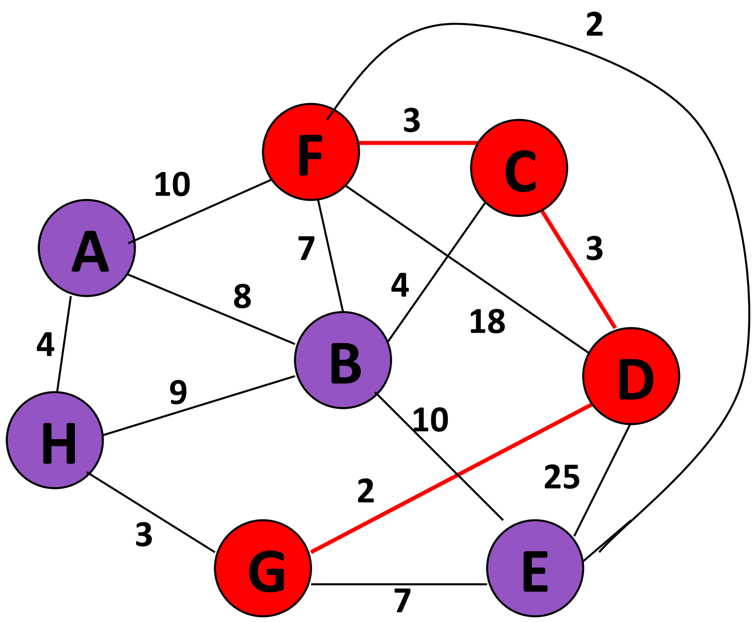
Prim’s Algorithm



Update distances of
adjacent, unselected nodes

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F		3	C
G	T	2	D
H		3	G

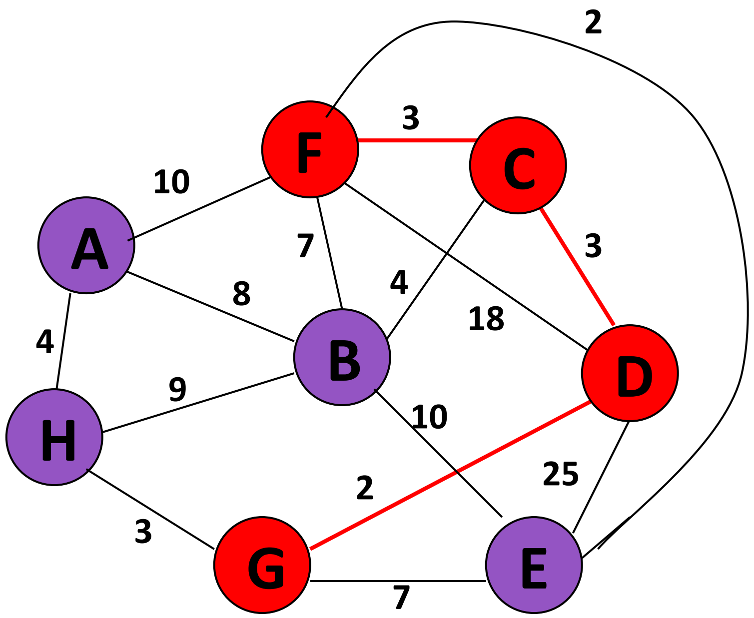
Prim's Algorithm



Select node with minimum distance

	K	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F	T	3	C
G	T	2	D
H		3	G

Prim’s Algorithm

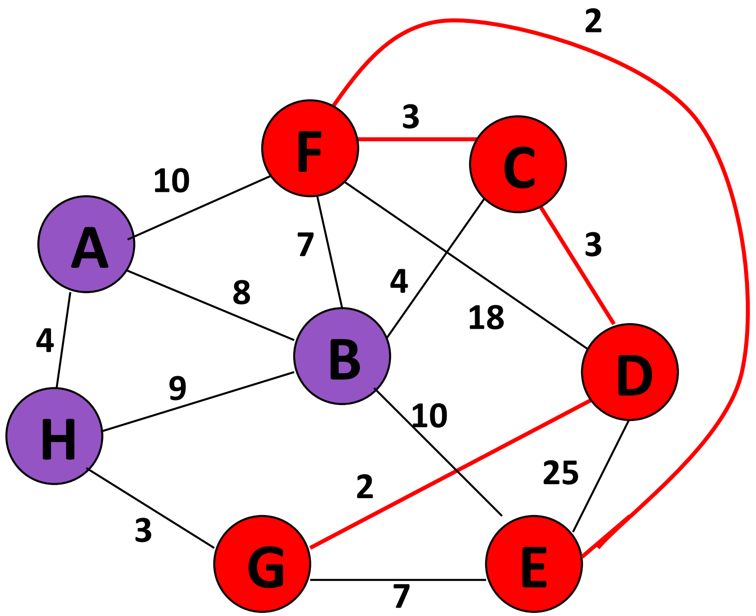


Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E		2	F
F	T	3	C
G	T	2	D
H		3	G

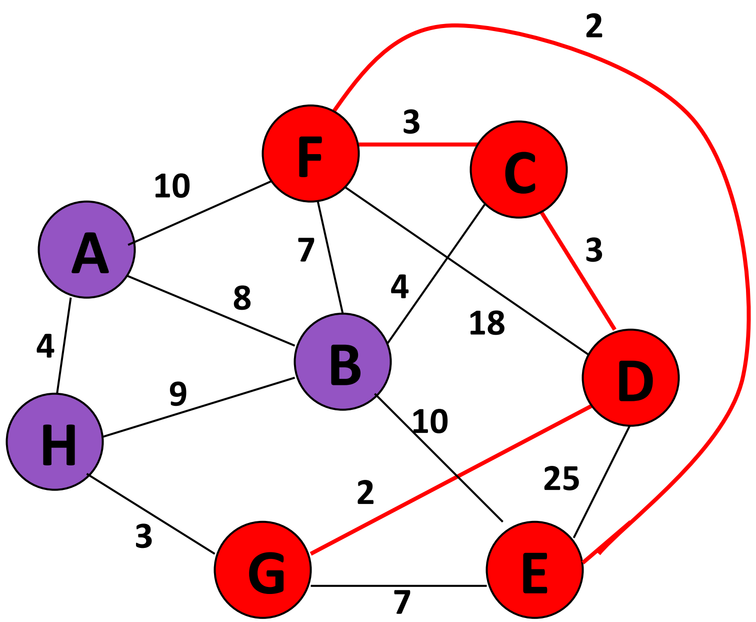
Prim’s Algorithm

Select node with minimum distance



	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Prim’s Algorithm

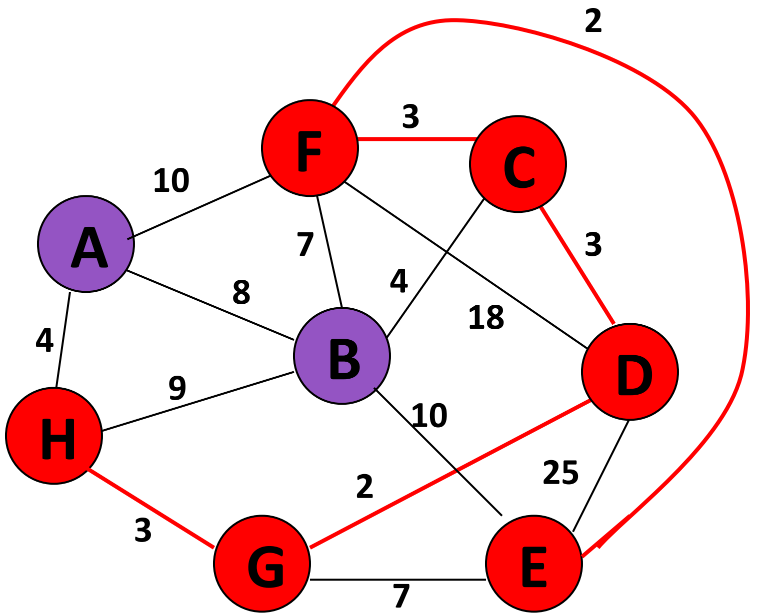


Update distances of
adjacent, unselected nodes

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Table entries unchanged

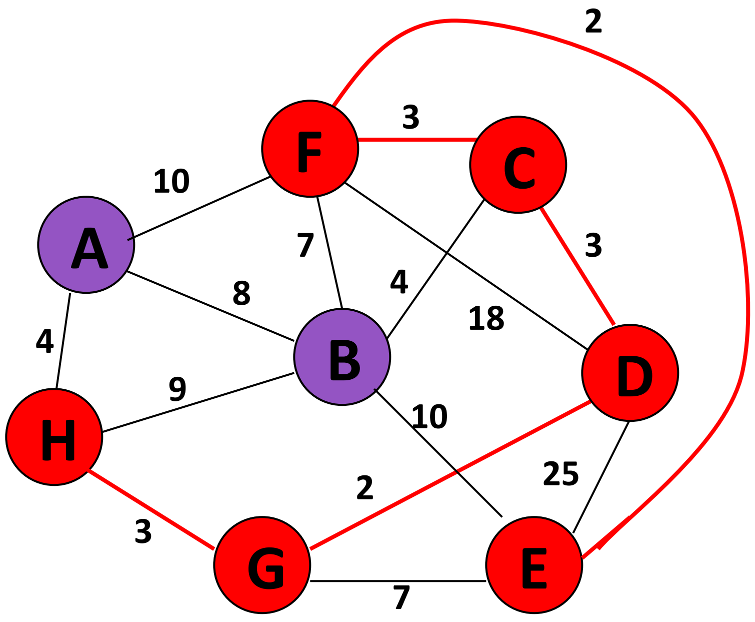
Prim’s Algorithm



Select node with minimum distance

	K	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

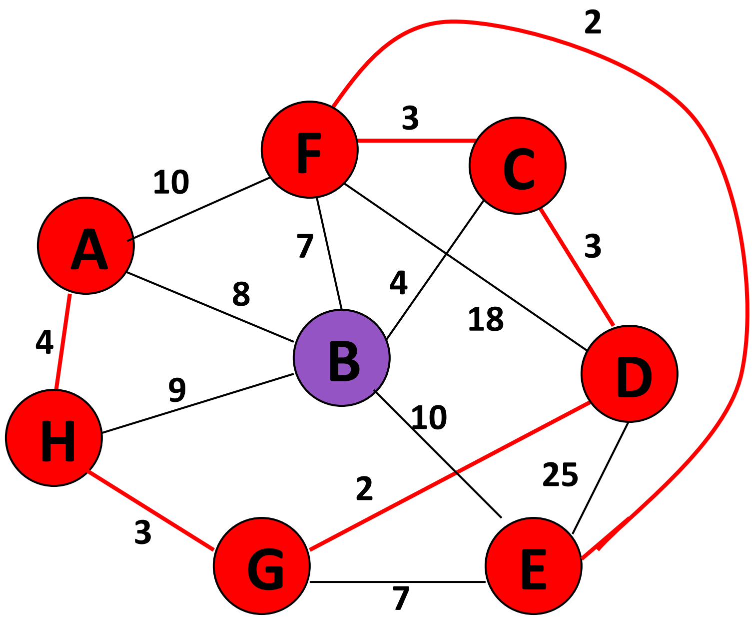
Prim's Algorithm



Update distances of adjacent, unselected nodes

	K	d_v	p_v
A		4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

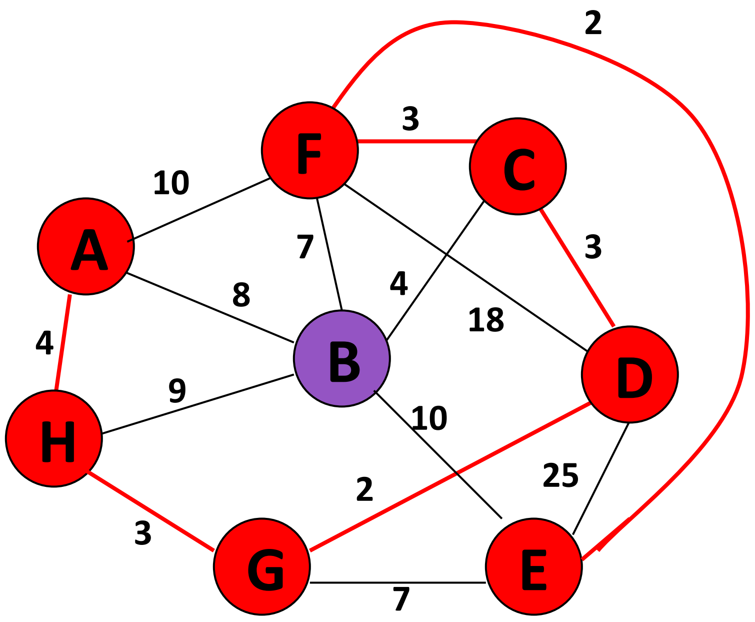
Prim’s Algorithm



Select node with minimum distance

	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Prim’s Algorithm

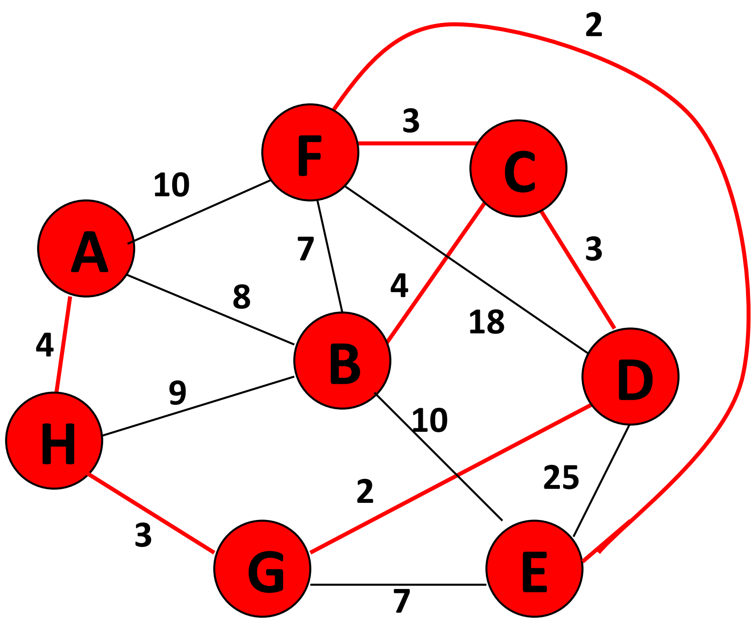


Update distances of adjacent, unselected nodes

	K	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Table entries unchanged

Prim’s Algorithm



Select node with minimum distance

	K	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G