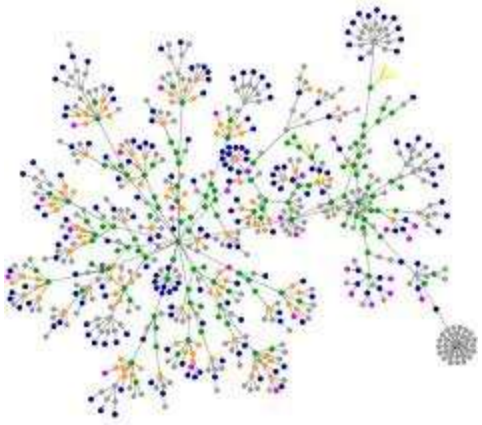


Graphs

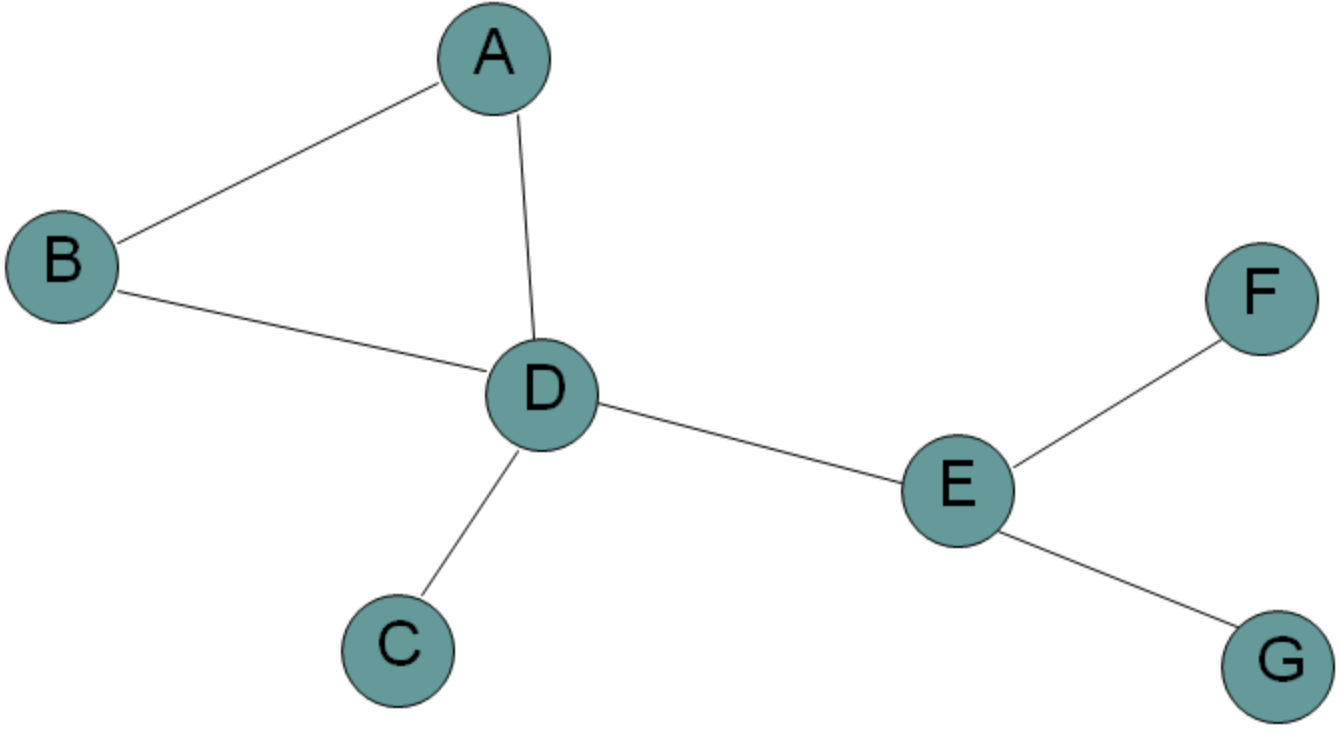
Deliverables

- Graph Terminology
- Types of Graphs
- Graph representations
- DFS and BFS

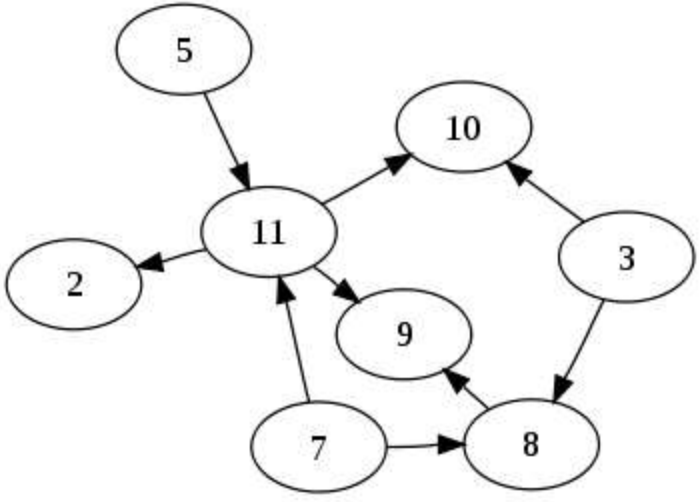
Graph-Basics

- Graph: pair(V,E) where V is set of nodes called vertices E is a collection of pair of vertices called edges. vertices and edges are positions and store elements
- Directed edge: ordered pair of vertices(u, v) where first vertex u is the origin and second vertex v is the destination
- Undirected edge: unordered pair of vertices for example distance between two cities
- In Directed graph all the edges are directed and in undirected graph all the edges are undirected

Undirected Graph

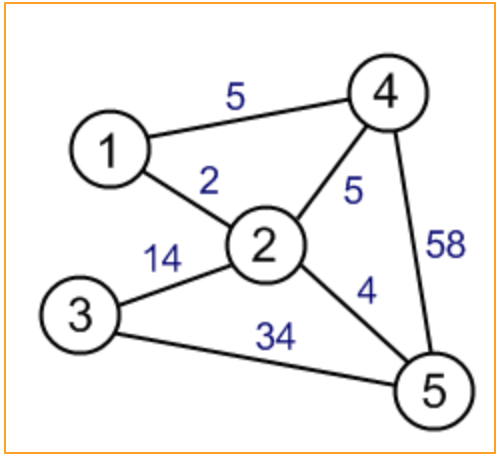


Directed Graph

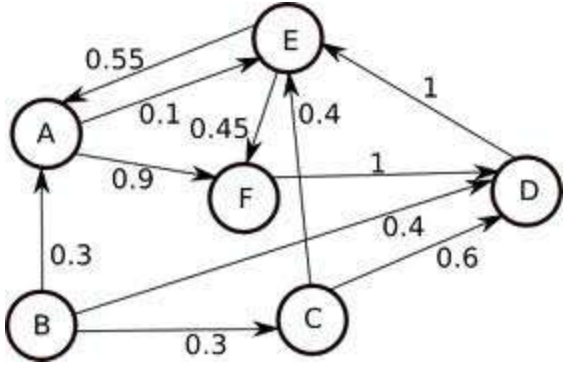


Weighted Graphs

Weighted undirected G

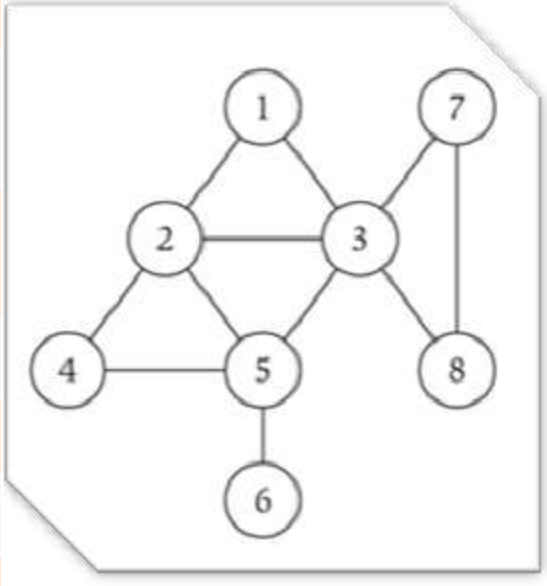


Weighted Directed G



Graph-Applications

Graph	Nodes	Edges
Electronic Circuit	Gates	Wires
Transportation	Crossings, Stations, Airports	Roads, Railway lines, Air Routes
Communication	Computers, Mobiles	Fiber cables, Wireless Links
Internet	Web Pages	Hyperlinks
Software Systems	Functions	Function calls
Social Networks	People	Relationships
ER Diagrams	Stores or Processes	Relation



Graph terminology

End vertices of an edge U and V are the endpoints of a

Edge incident on a vertex a, d, b are incident on v

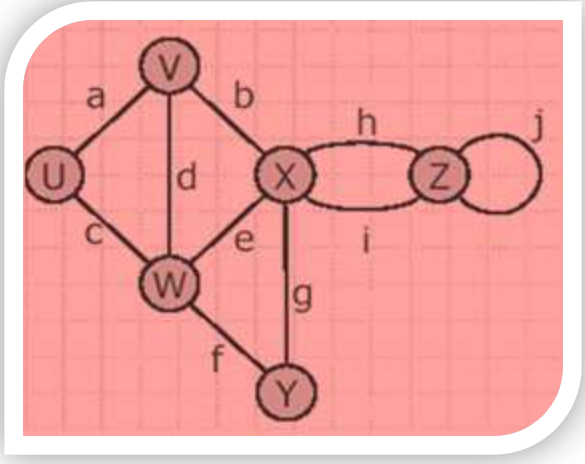
adjacent vertices u and v are adjacent

Degree of a vertex : x has degree 5

sum of degree of all vertices is double the no of edges of an undirected graph

parallel edges: h and i are parallel edges

Self loop : j is a self loop



Path

path-sequence of alternating vertices and edges

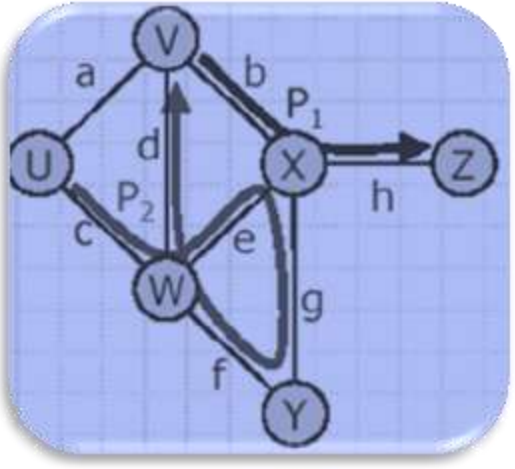
begins with a vertex and ends with a vertex

each edge is preceded and followed by its endpoints

Simple path- a path such that all edges and vertices are distinct

v, b, x, h, z is a simple path

u, c, w, e, x, g, y, f, w, d, v is not a simple path



Cycle

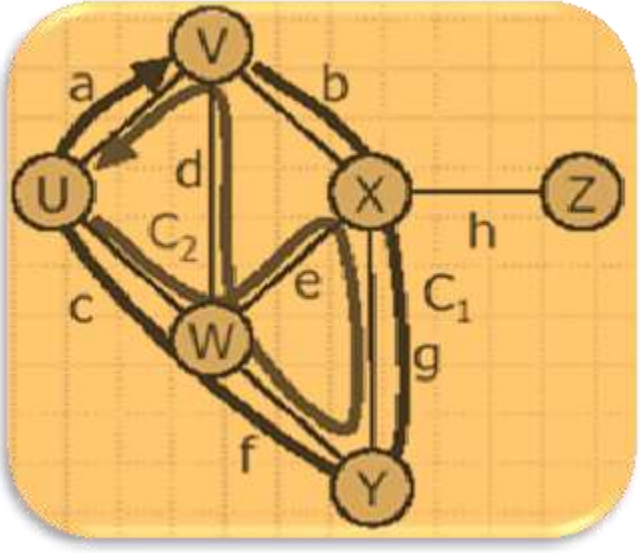
cycle- circular sequence of alternating vertices and edges

each edge is preceded and followed by its endpoints

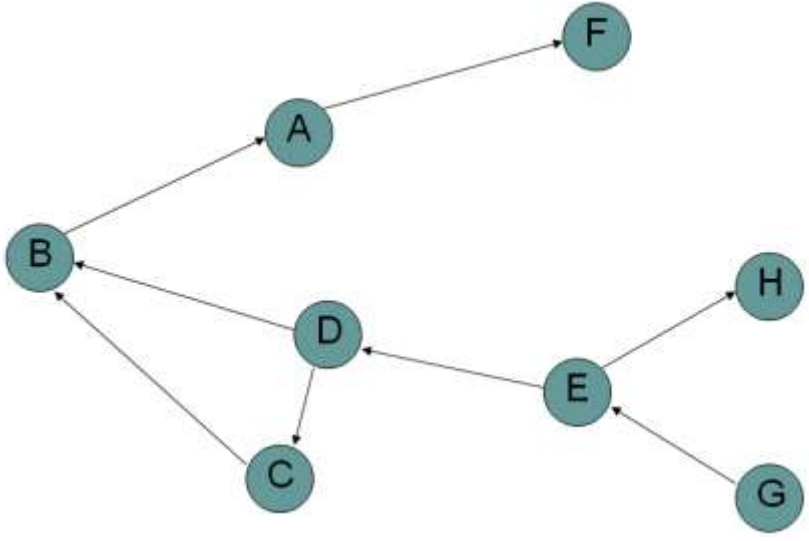
Simple cycle : such that all its vertices and edges are distinct

v, b, x, g, y, f, w, c, u, a, v is a simple cycle

u, c, w, e, x, g, y, f, w, d, v, a, u is not a simple cycle



DAG



Graph operations

incidentedges(v)	insertvertex(o)
endvertices(e)	insertedge(v,w,o)
isdirected(e)	insertdirectededge(v,w,o)
origin(e)	removevertex(v)
destination(e)	removeedge(e)
opposite(v,e)	numvertices()
areadjacent(v,w)	numedges()

Graph

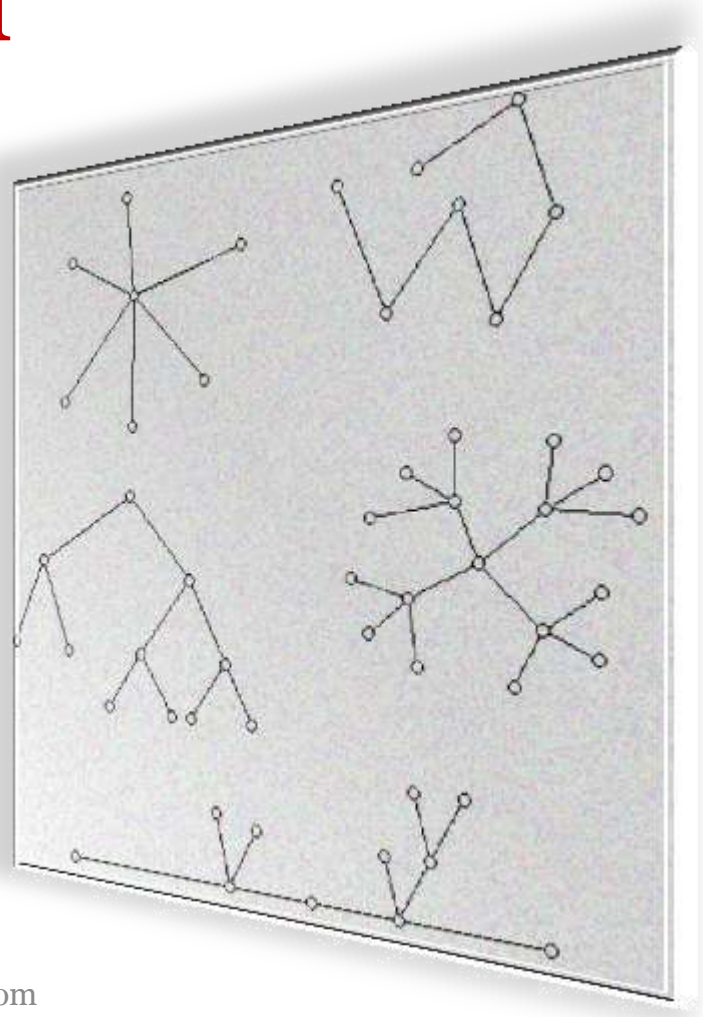
A connected graph without cycles is a tree

A Collection of trees is called a forest.

m=no. of edges n= no. of vertices

For a tree m=n-1

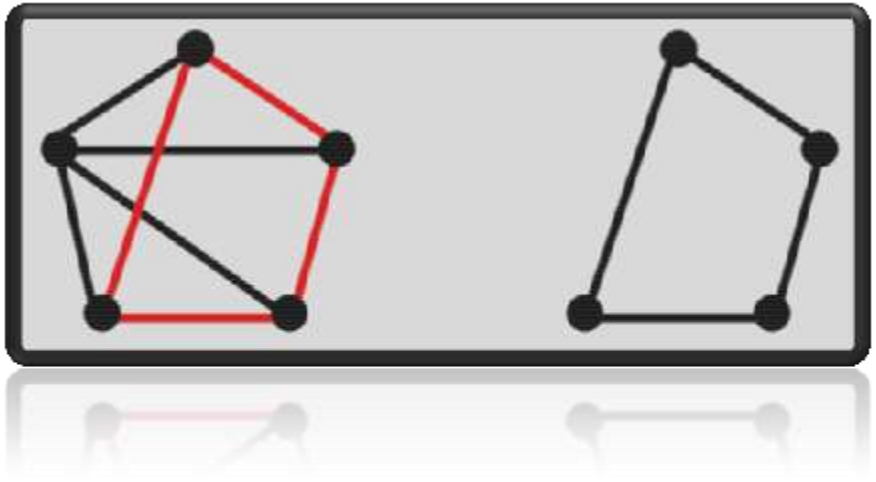
If m<n-1 Graph is not connected



Sub graph

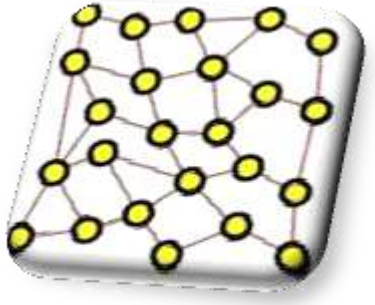
Sub Graph: A Graph consisting of Subset of edges and subset of vertices of another graph is a sub graph of that graph.

Super graph of a graph *G* is a graph of which *G* is a sub graph.



Connected Graph

If there is a path between any two pair of vertices it is called a connected graph

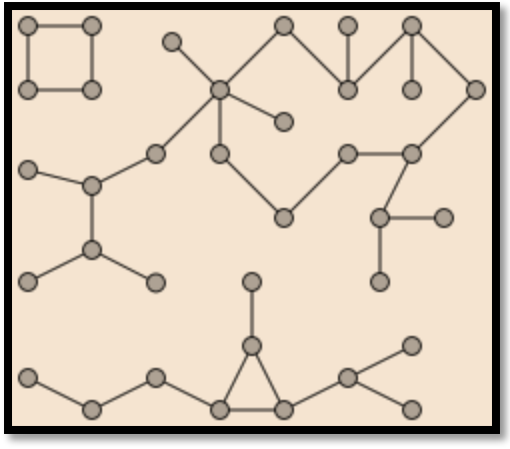


Connected Component of Graph

Connected components of graph : maximal connected sub graph

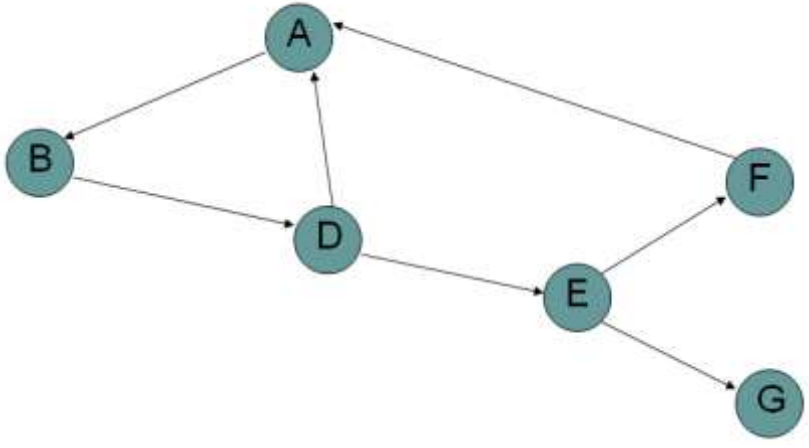
Given Graph is one graph containing three connected components

Connected component is that if we can not add more elements to the set and retain the connected property. Any sub graph of a connected component is not a connected component

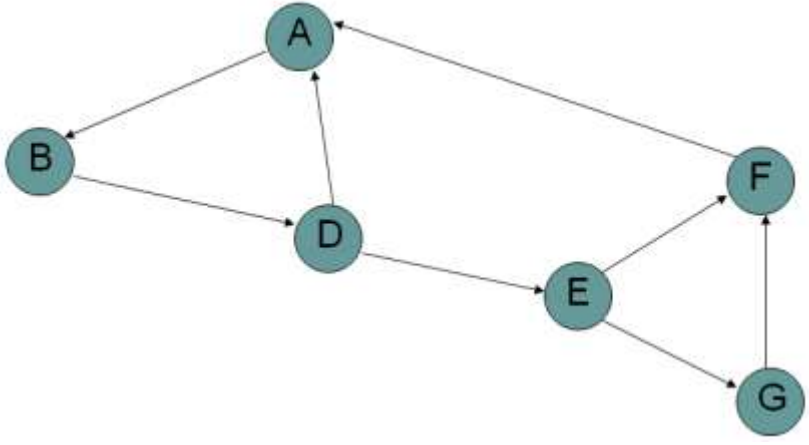


Strongly Connected

Not strongly Connected

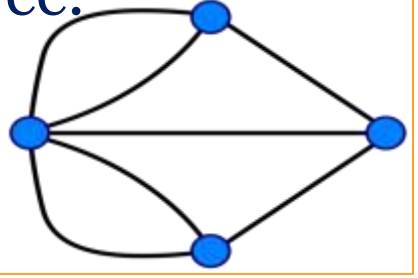


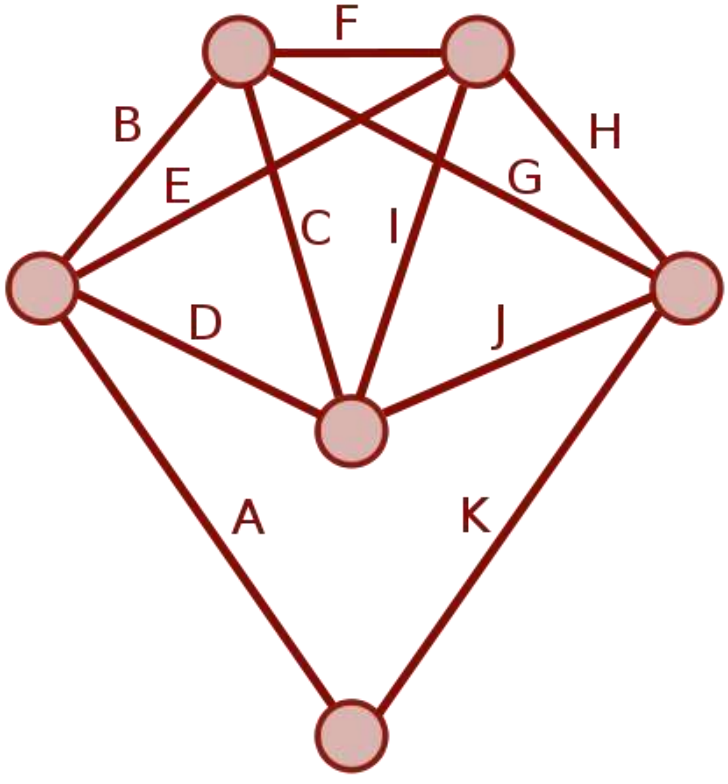
Strongly Connected



Euler Path

Eulerian path is a path in a graph which visits each edge exactly once and returns to the starting vertex. A Graph has a Eulerian Path if all vertices have even degree.

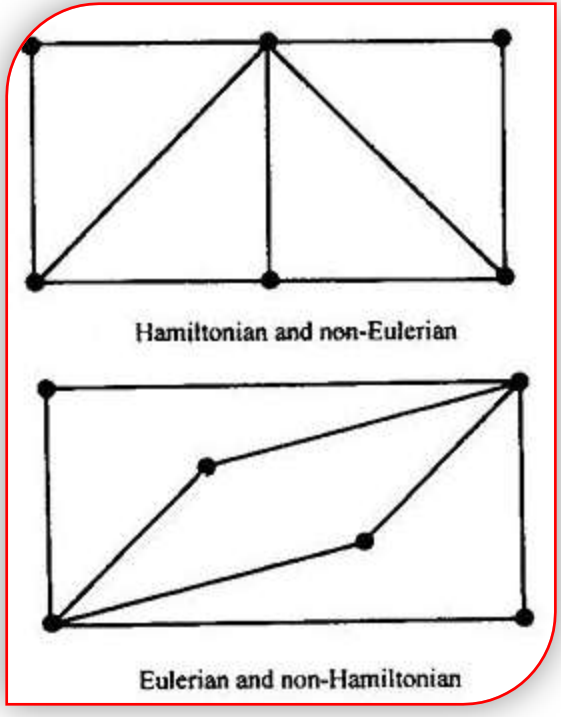




Hamiltonian Path

Hamiltonian path (or traceable path) is a path in an undirected graph which visits each vertex exactly once.

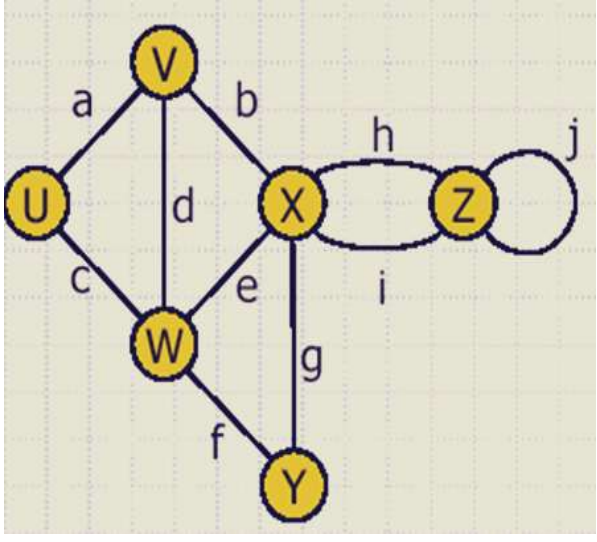
Hamiltonian cycle (or Hamiltonian circuit) is a cycle in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.



Independent Set

Independent set or stable set is a set of vertices in a graph, no two of which are adjacent. That is, it is a set *I* of vertices such that for every two vertices in *I*, there is no edge connecting the two.

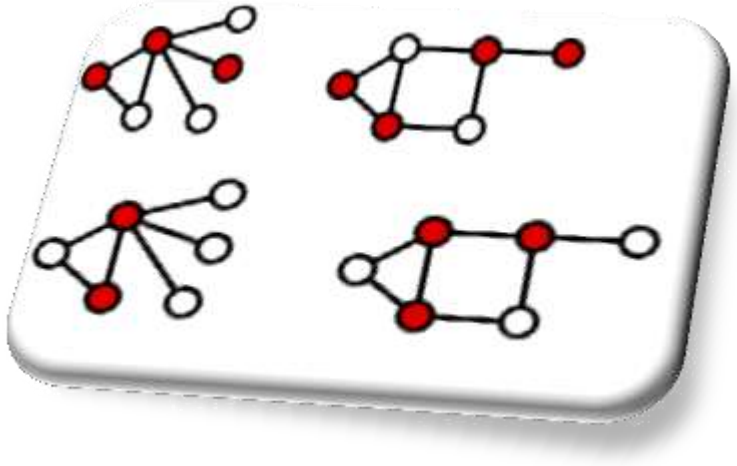
Maximum independent set is a largest independent set for a given graph *G*. Maximum Independent set is { U,Z,Y}



Vertex Cover

set of vertices such that each edge of the graph is incident to at least one vertex of the set. A vertex cover of G is a set C of vertices such that each edge of G is incident to at least one vertex in C. The set C is said to cover the edges of G.

Minimum vertex cover is vertex cover of smallest possible size.

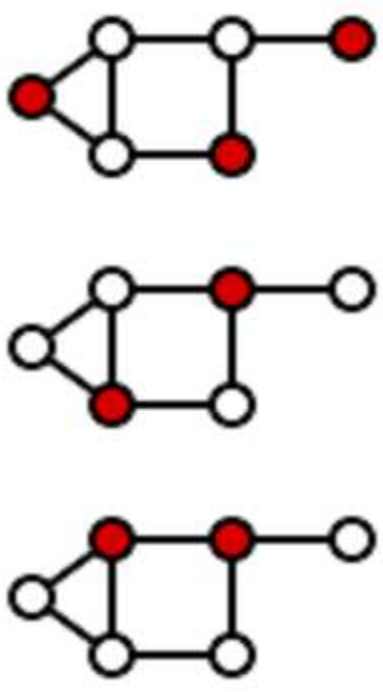


Dominating Set

Every vertex not in D is joined to at least one member of D by some edge

Domination number is no of vertices in smallest dominating set for G.

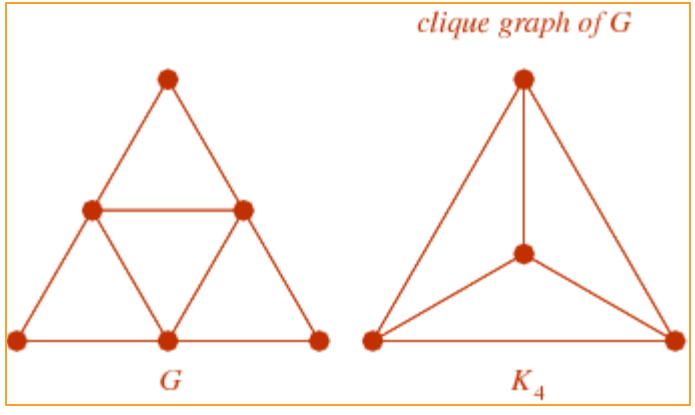
Nodes covering all nodes of graph



Clique

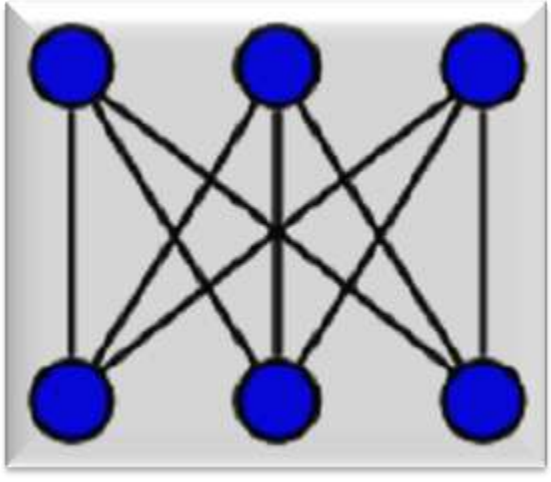
Clique in an undirected graph $G = (V, E)$ is a subset of the vertex set such that for every two vertices in the subset, there exists an edge connecting the two.

Maximum clique is a clique of the largest possible size in a given graph



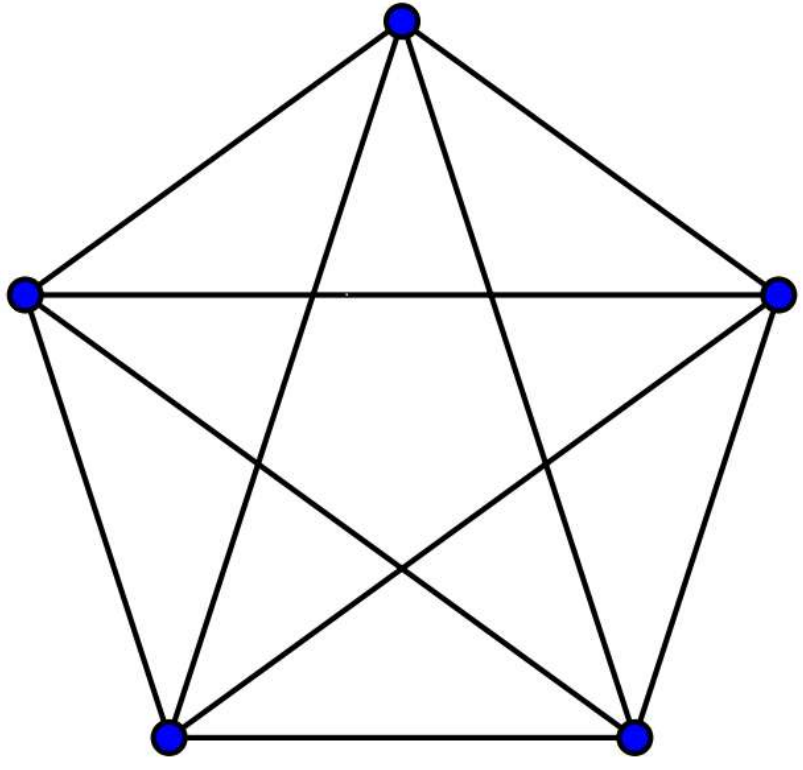
Bipartite K3,3

In a bipartite graph, the vertices can be divided into two sets, W and X, so that every edge has one vertex in each of the two sets.



Complete Graph

In a complete graph each pair of vertices is joined by an edge, that is, the graph contains all possible edges. K5 is given.

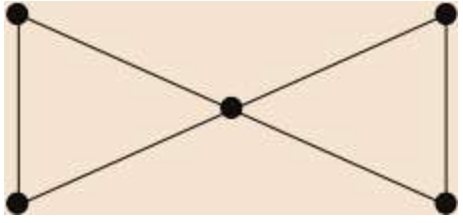


Planar Graph

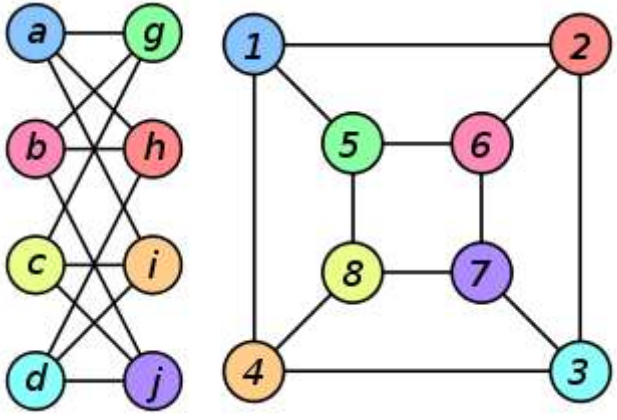
Graph F is planar if it can be drawn in such a way that no edges cross each other. K5 and K3,3 are not planar.

Kuratowski Theorem:

finite graph is planar iff it does not contain a sub graph of K5 or K3,3



Graph isomorphism



Graphs G and H are isomorphic if vertex sets of G and H any two vertices u and v of G are adjacent in G if and only if f(u) and f(v) are adjacent in *H*

f(a) = 1 f(b) = 6

f(c) = 8 f(d) = 3

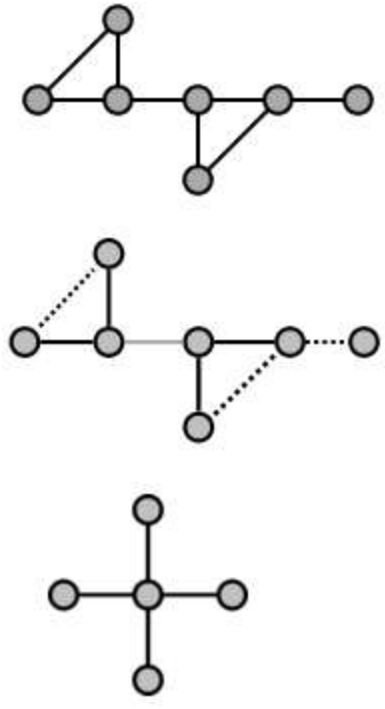
f(g) = 5 f(h) = 2

f(i) = 4 f(j) = 7

Graph Minor

Undirected graph H is called a minor of the graph G if H is isomorphic to a graph that can be obtained by zero or more edge contractions on a sub graph of G

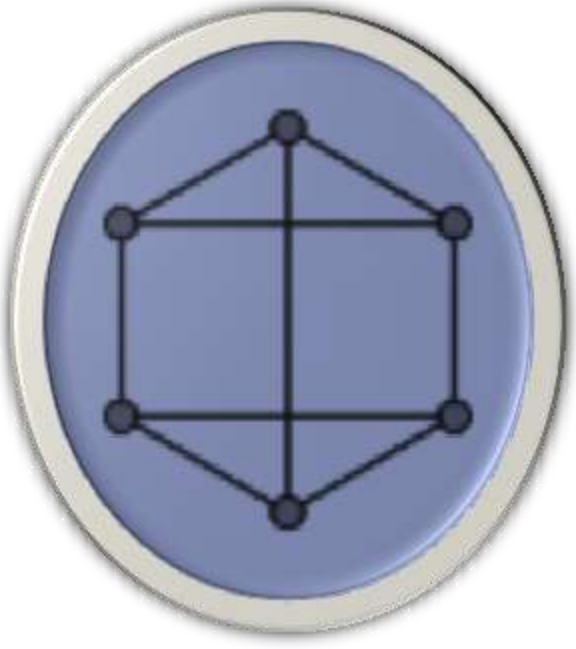
Edge contraction is an operation which removes an edge from a graph while simultaneously merging together the two vertices it used to connect.



Regular Graph

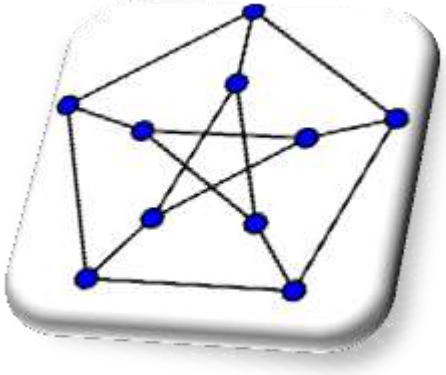
If all vertices have the same degree it is called the regular graph.

Given graph is 3-regular.



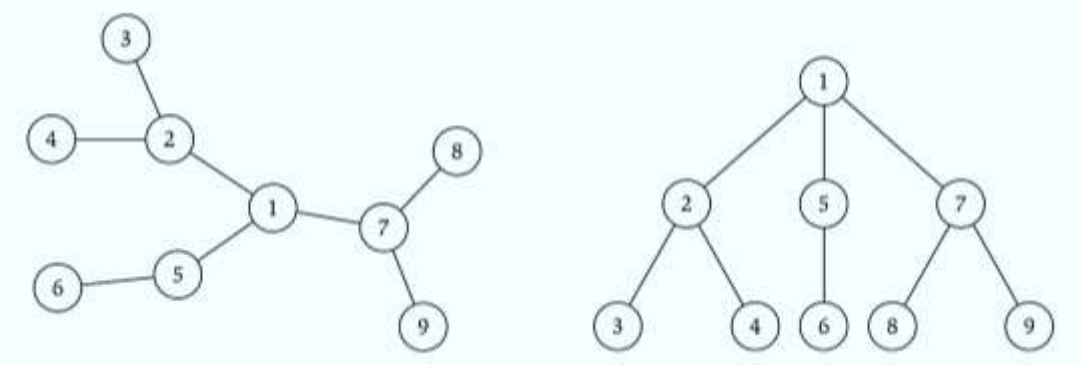
Peterson Graph

Undirected graph with 10 vertices and 15 edges. It is a small graph that serves as a useful example and counterexample for many problems.



Rooted tree graphs

Given a tree T, choose a root node r and orient each edge away from r.

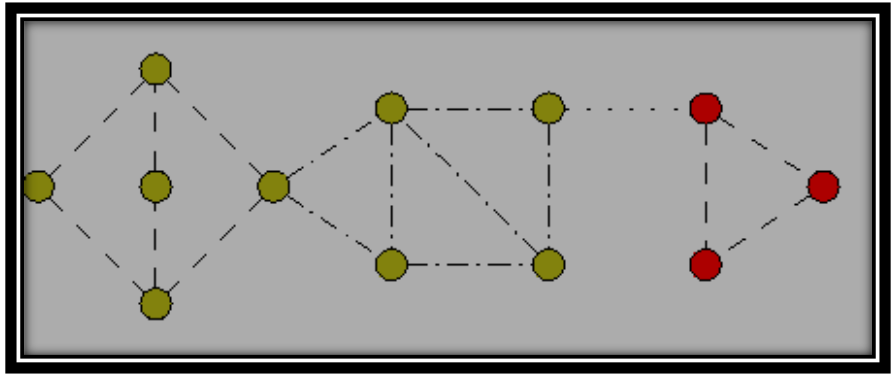


2-Edge Connectivity

A graph is 2-edge connected iff it remains connected after removal of any one edge.

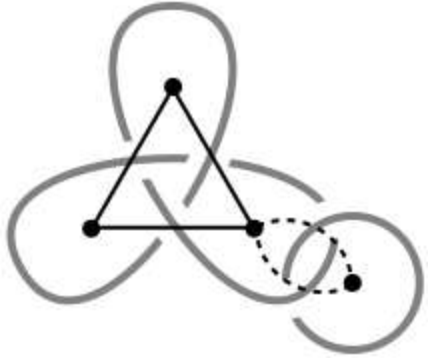
Means it should not have any bridge

Graph has a bridge and adding one edge parallel to the bridge will make it 2-edge connected



Knot in a graph

A knot in a directed graph is a collection of vertices and edges such that every vertex in the knot has outgoing edges, and all outgoing edges from vertices in the knot terminate at other vertices in the knot. Thus it is impossible to leave the knot while following directions of the edges



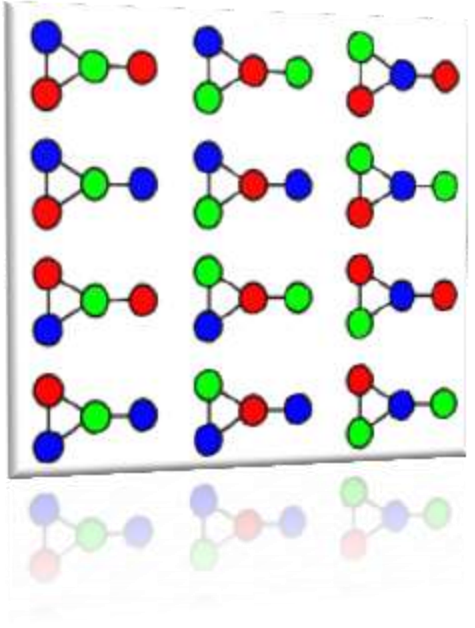
Graph Coloring

It is a way of colouring the vertices of a graph such that no two adjacent vertices share the same colour; this is called a vertex colouring.

Edge colouring assigns a colour to each edge so that no two adjacent edges share the same colour.

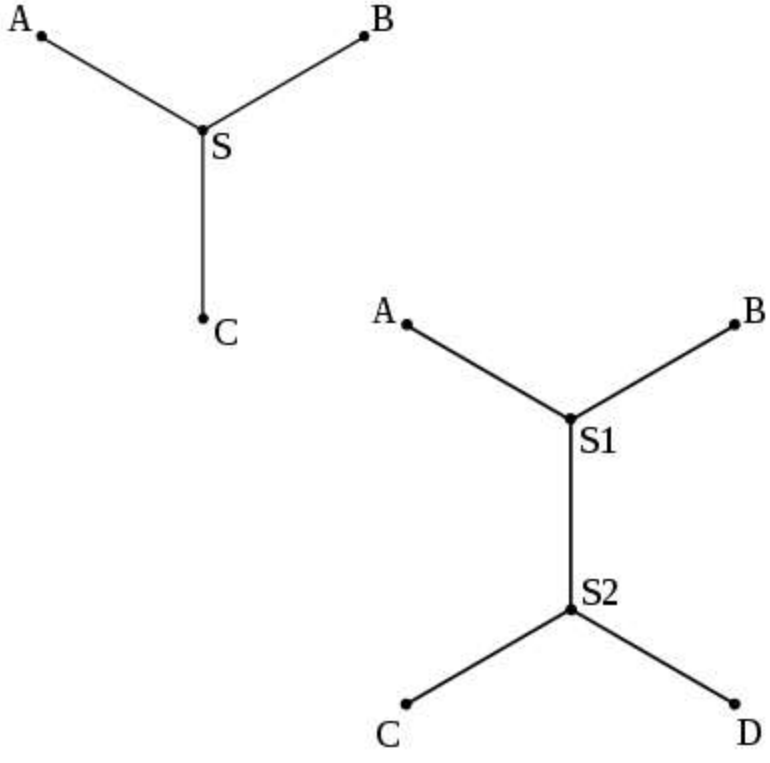
Minimum Coloring of a Graph is to use minimum number of different colors while holding the above conditions.

Any Planar Graph or any graph with clique of four or less can be colored in four colors.



Steiner Tree

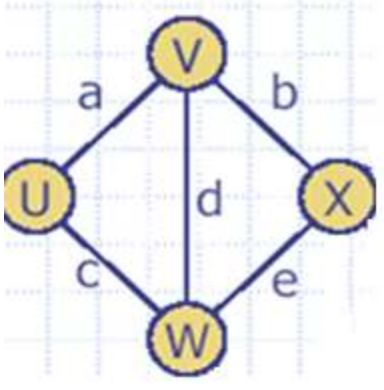
Given a set V of vertices, interconnect them by a network (graph) of shortest length, where length is sum of lengths of all edges. Difference between Steiner tree and MST is that, in Steiner , extra intermediate vertices and edges may be added to the graph to reduce length of spanning tree. These new vertices are known as Steiner points or Steiner vertices.



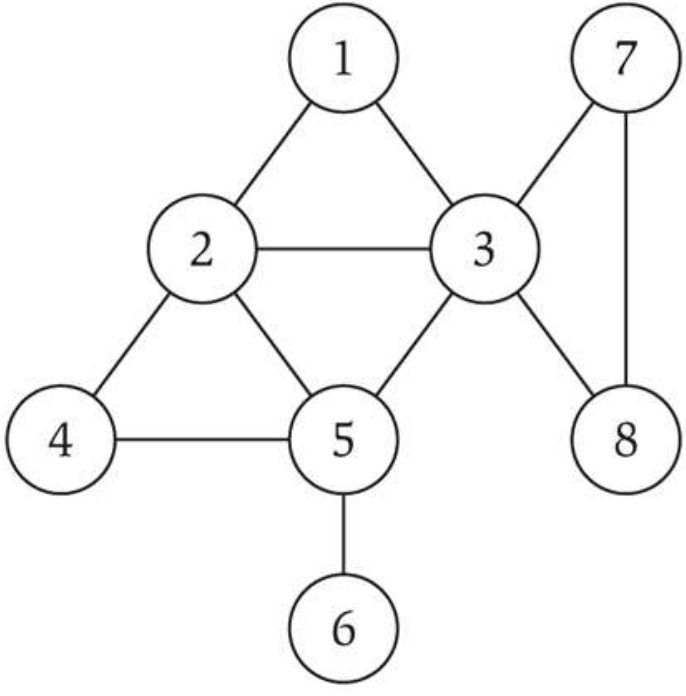
Graph representation-Incidence Matrix

The graph is represented by a matrix of size |V| (number of vertices) by |E| (number of edges) where the entry [vertex, edge] contains the edge's endpoint data (simplest case: 1 - incident, 0 - not incident).

	U	V	W	X
A	1	1	0	0
B	0	1	0	1
C	1	0	1	0
D	0	1	1	0
E	0	0	1	1



Adjacency Matrix



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Adjacency Matrix

nXn matrix A, where n is the number of vertices in the graph.

If there is an edge from a vertex x to a vertex y, then element $a_{x,y}$ is 1 , otherwise 0.

Every element in matrix can be represented by a single bit. However in case of sparse graphs it can be a waste of space and time.

In computing, this matrix makes it easy to find sub graphs, and to reverse a directed graph. Space $O(n^2)$

Adjacency Matrix

Checking if (u, v) is an edge takes O(1) time.

Identifying all edges takes (n^2) time.

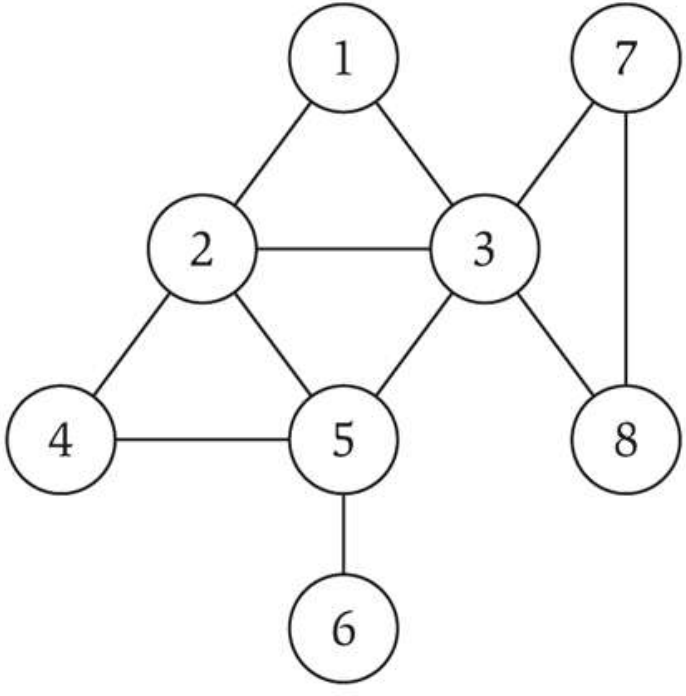
For undirected graphs it will be always symmetric matrix along the diagonal

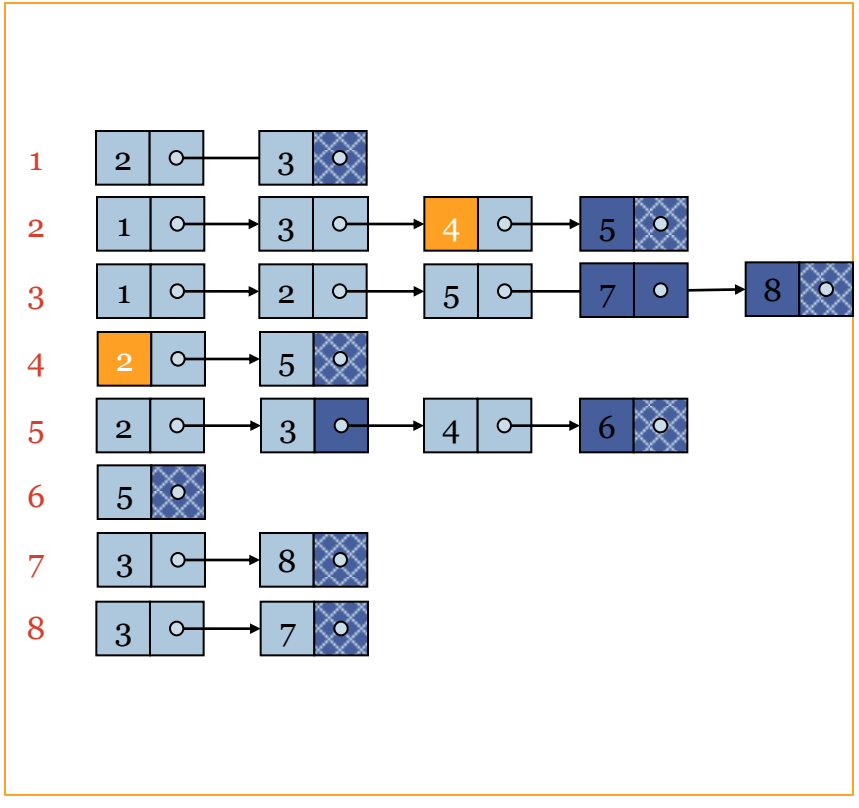
Good for dense graphs

For non-weighted graphs only a Boolean matrix will work

In weighted graphs actual weights will be written instead of 1

Adjacency List





Adjacency List

An array indexed by vertex numbers points to a singly-linked list of the neighbors of each vertex.

Space proportional to O(m + n) so it is Space efficient

Checking if (u, v) is an edge takes O(deg(u)) time.

Identifying all edges takes O(m + n) time.

Good for sparse graphs

Incidence List

Incidence list edges are represented by an array containing pairs (tuples if directed) of vertices (that the edge connects) and possibly weight and other data. Vertices connected by an edge are said to be adjacent.

(2,4)

(5,6)

(5,9)

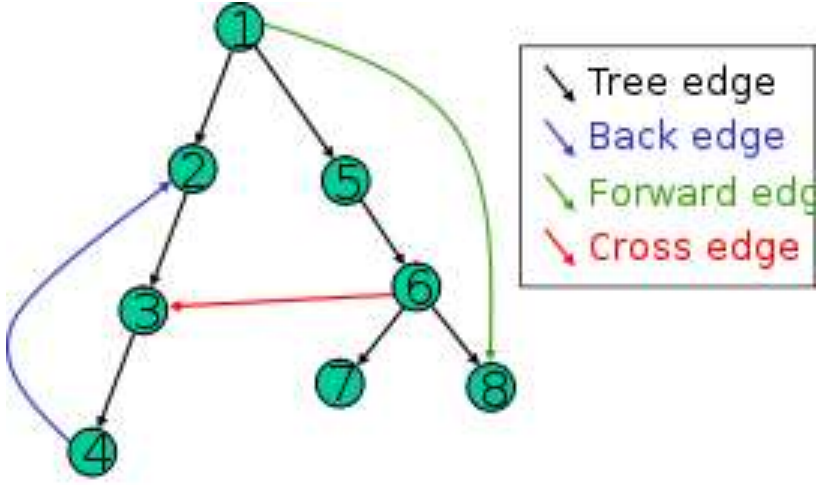
(1,2)

Classification of edges

Tree edges of a directed graph are edges which belong to the spanning tree.

An edge except tree edge in the forward direction of the tree is the forward edge. Back edges point from a node to one of its direct ancestors, and cross edges, which do neither.

If original graph is undirected then all of its edges are tree edges or back edges.



Graph traversal-DFS

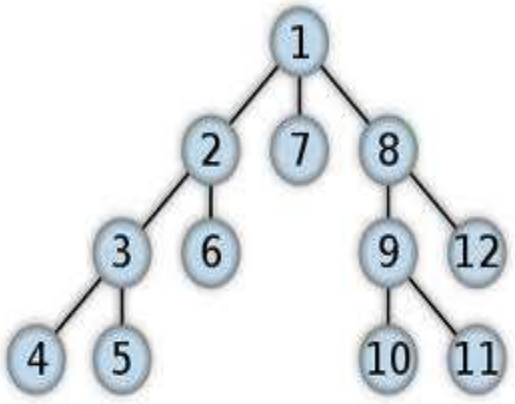
DFS is an uninformed search that progresses by expanding first child node of search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then it backtracks, returning to most recent node it hasn't finished exploring.

In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration. DFS on a graph with n vertices and m edges takes $O(n + m)$ time

DFS can be further extended to solve other graph problems. Like find and report a path between two given vertices.

DFS-example

			4								
		3	5	5							
	2	6	6	6	6				10		
	7	7	7	7	7	7		9	11	11	
1	8	8	8	8	8	8	8	12	12	12	12



DFS traversal for a Tree

```
DFSTree(T)
Push(Stack, Root(T))
While !Empty(Stack)
        v← Pop(Stack)
         visit(v)
         for all Children(v)
          Push(Stack, Children)
```

DFS Search Algorithm using stack

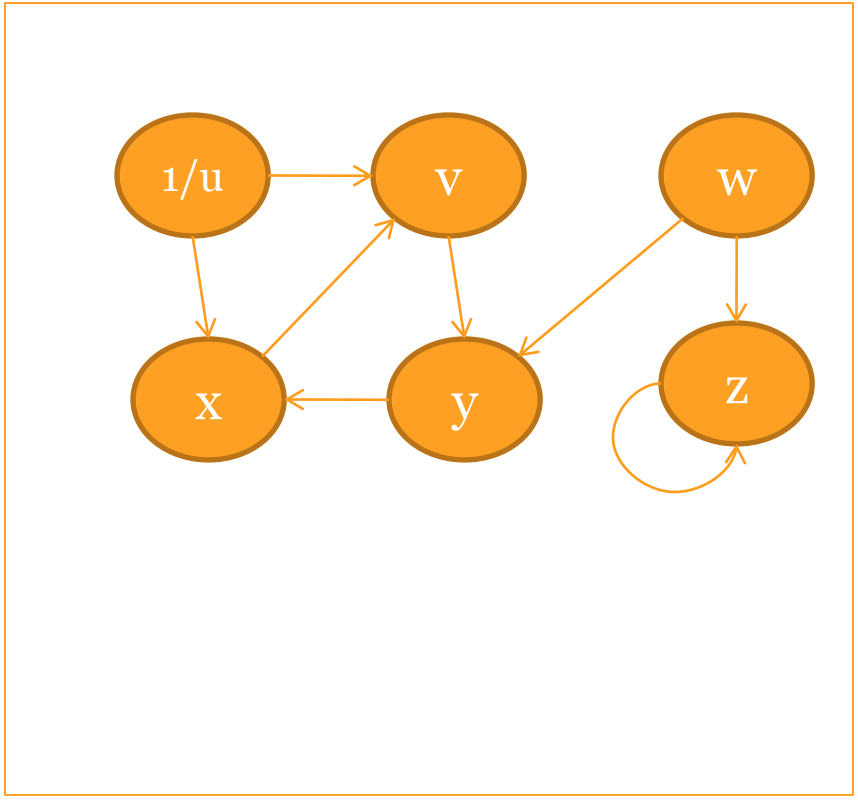
1. Push the root node onto a stack.
2. Pop a node from the stack and examine it.
3. If the element sought is found in this node, quit the search and return a result.
4. Otherwise push all its successors (child nodes) that have not yet been discovered onto the stack.
5. If the stack is empty, every node in the tree has been examined – quit the search and return "not found".
6. If the stack is not empty, repeat from Step 2

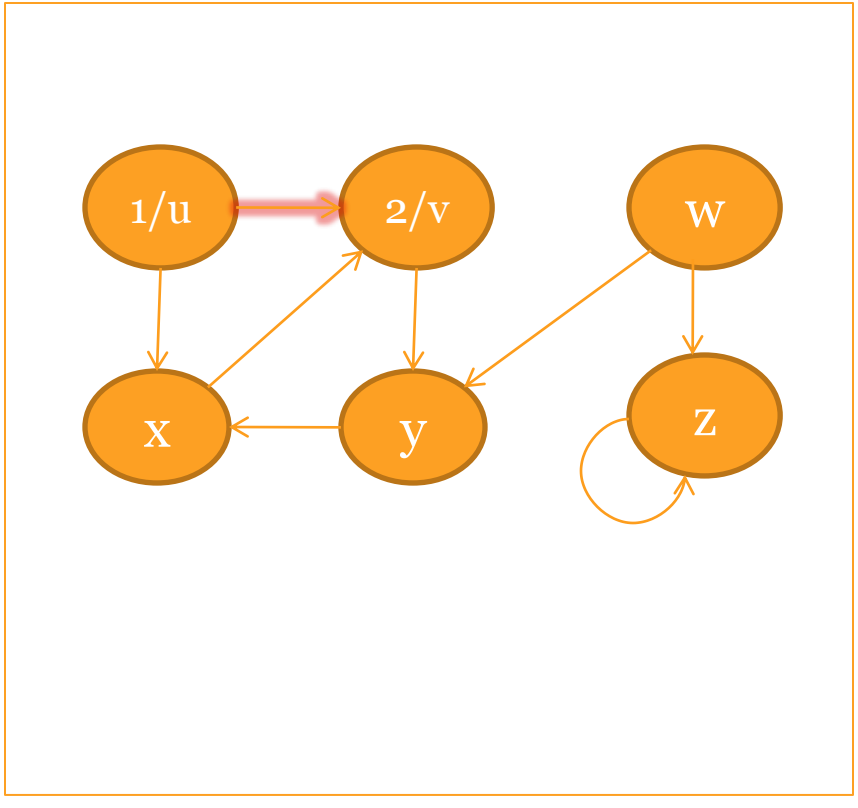
DFS Vs. BFS

```
DFSTree(T)
Push(Stack, Root(T))
While !Empty(Stack)
        v← Pop(Stack)
        visit(v)
        for all Children(v)
    Push(Stack, Children)
```

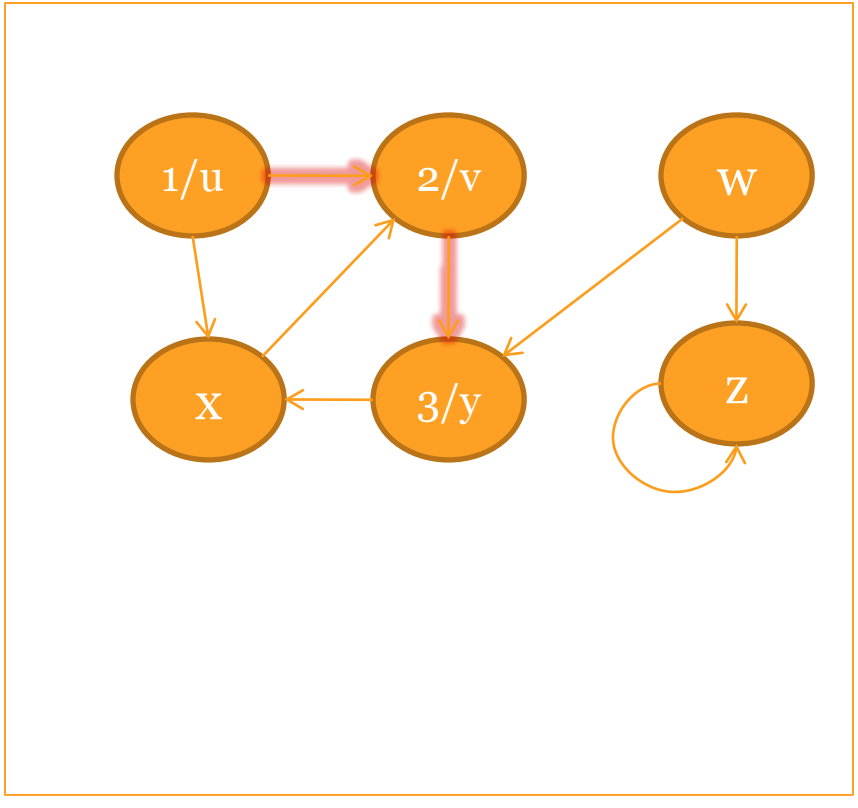
```
BFSTree(T)
Enqueue(Que, Root(T))
While !Empty(Que)
        v← Dequeue(Que)
        visit(v)
        for all Children(v)
        Push(Que, Children)
```

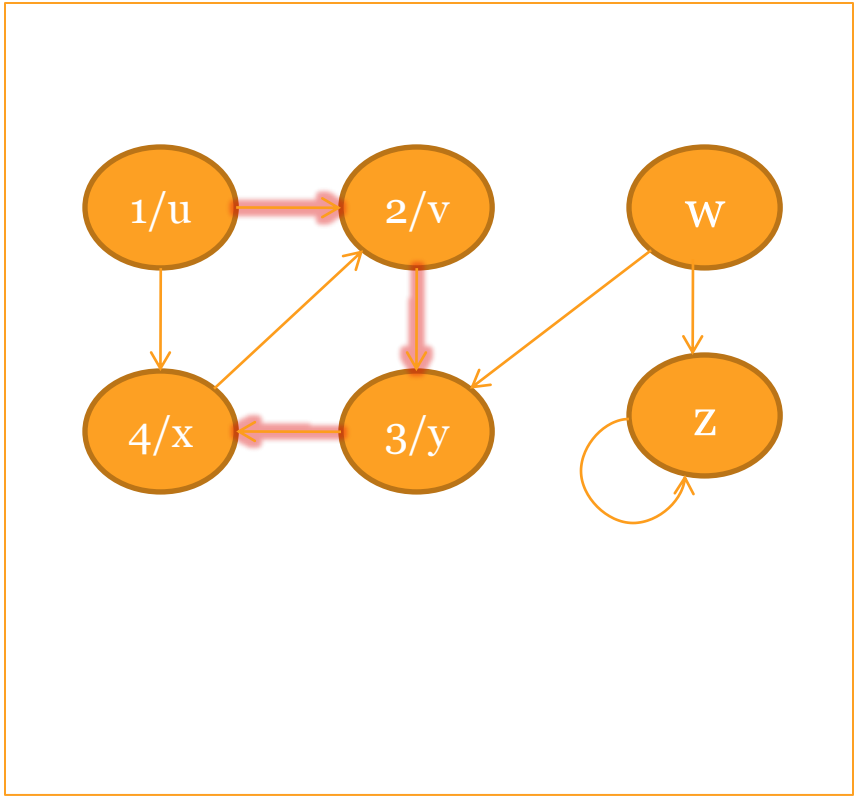
DFS example



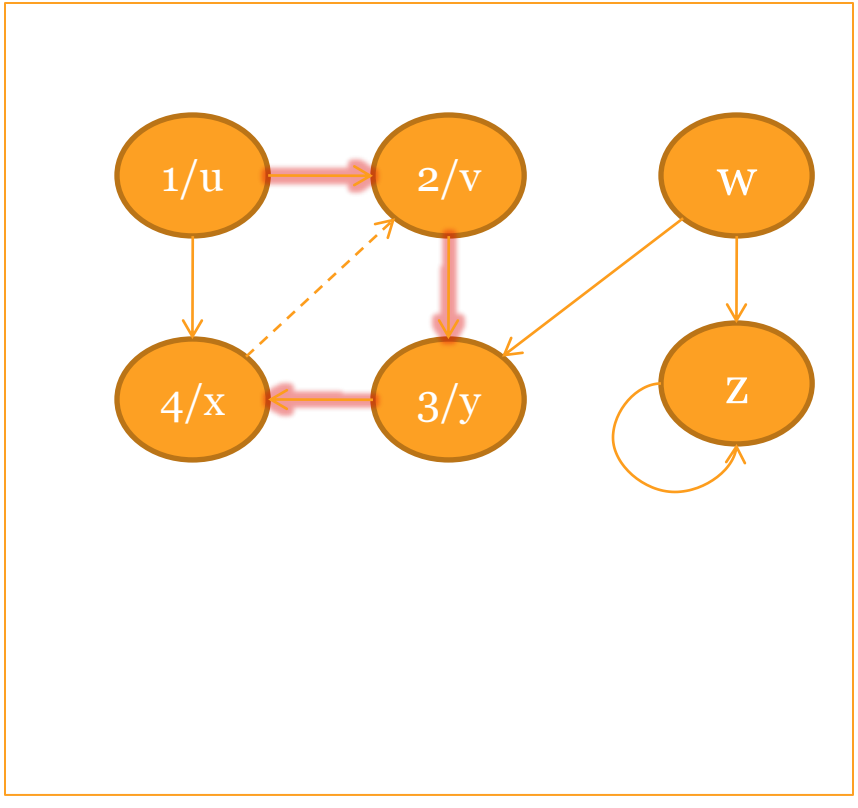


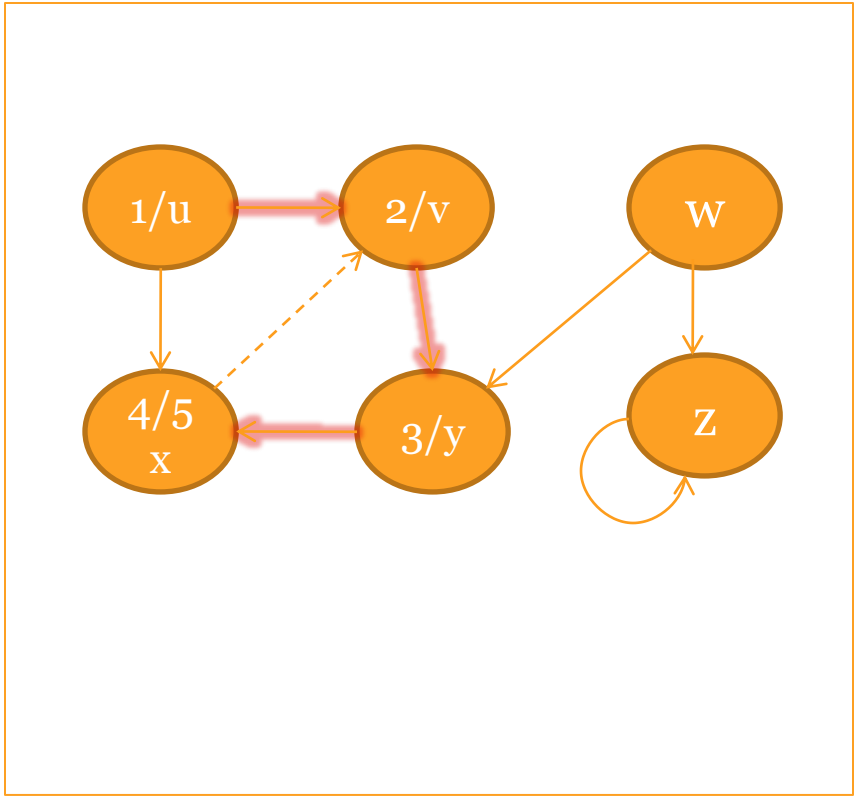
DFS example



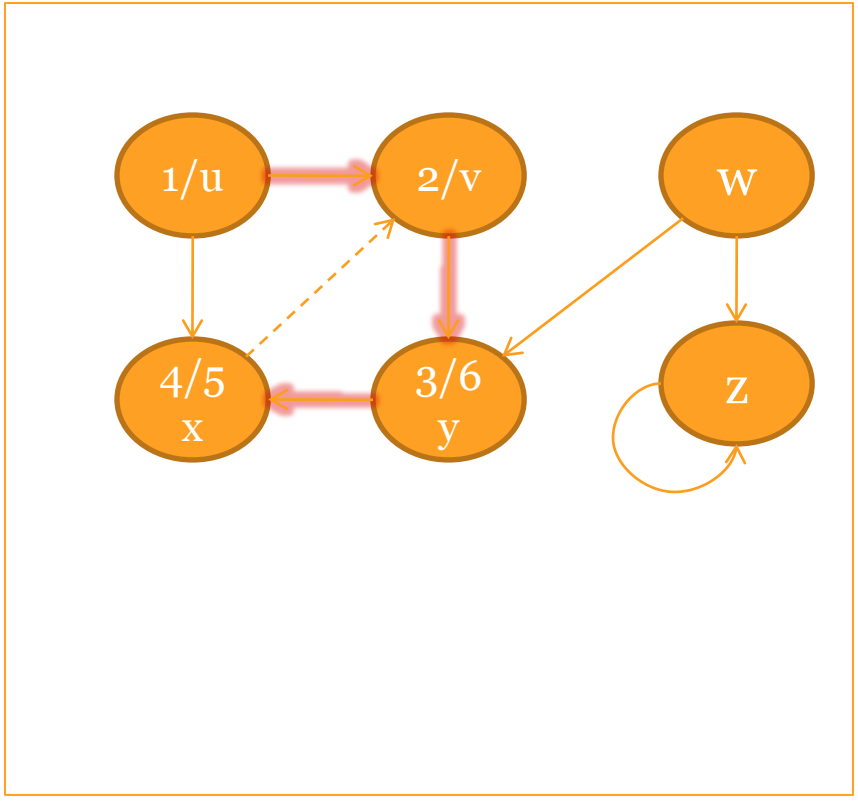


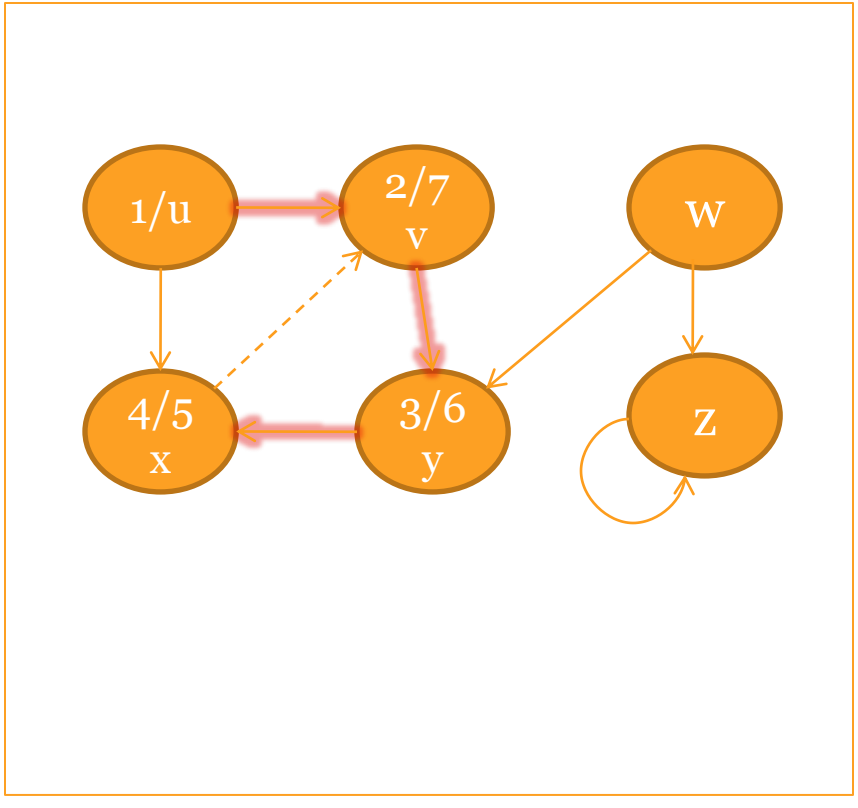
DFS example



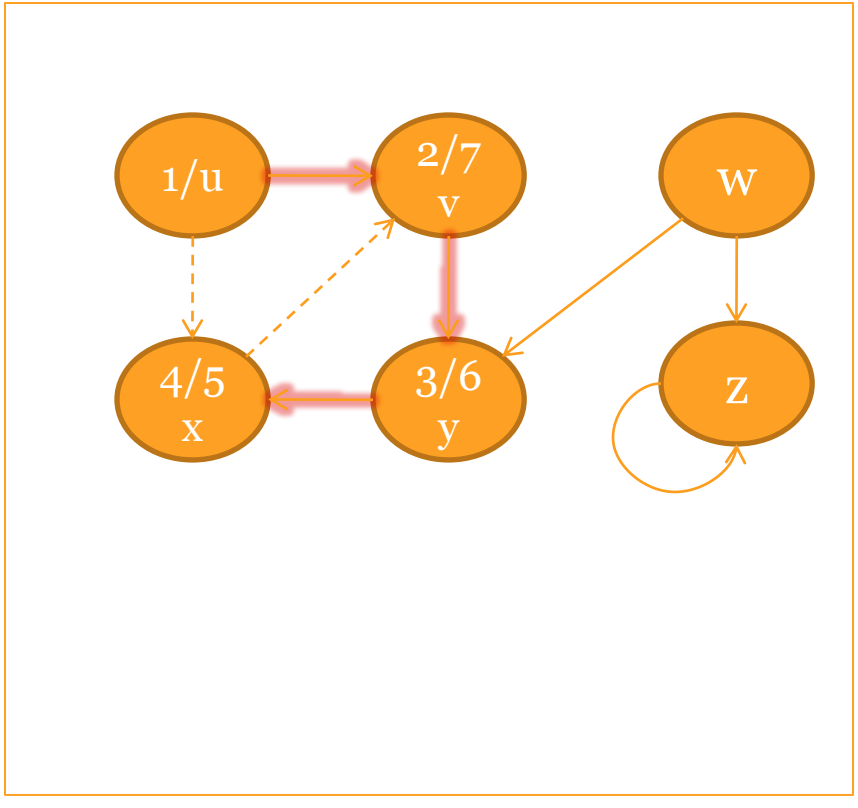


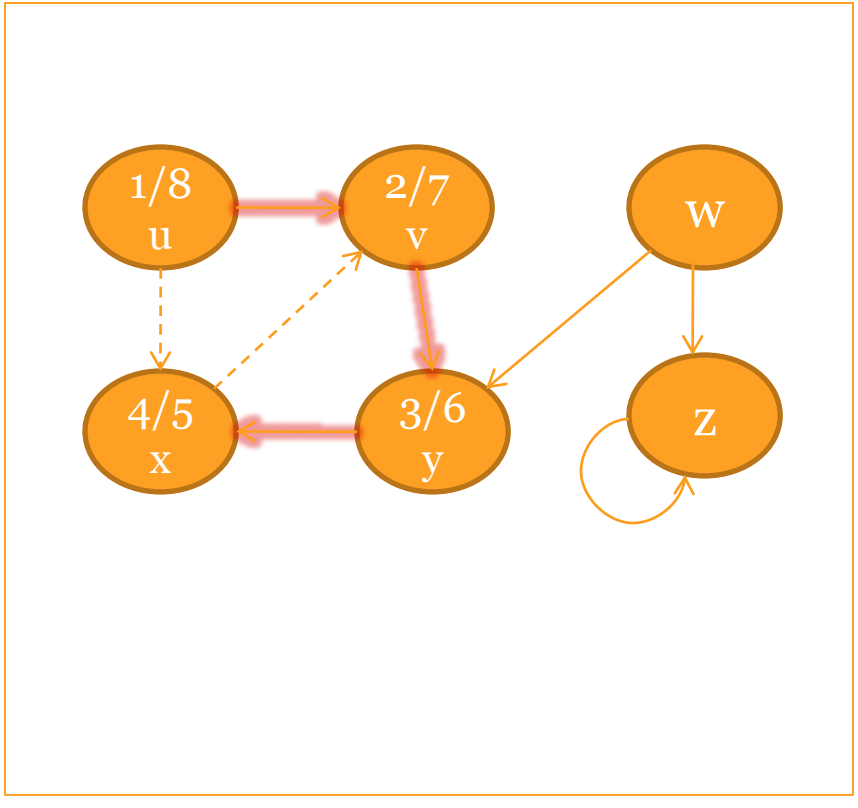
DFS example



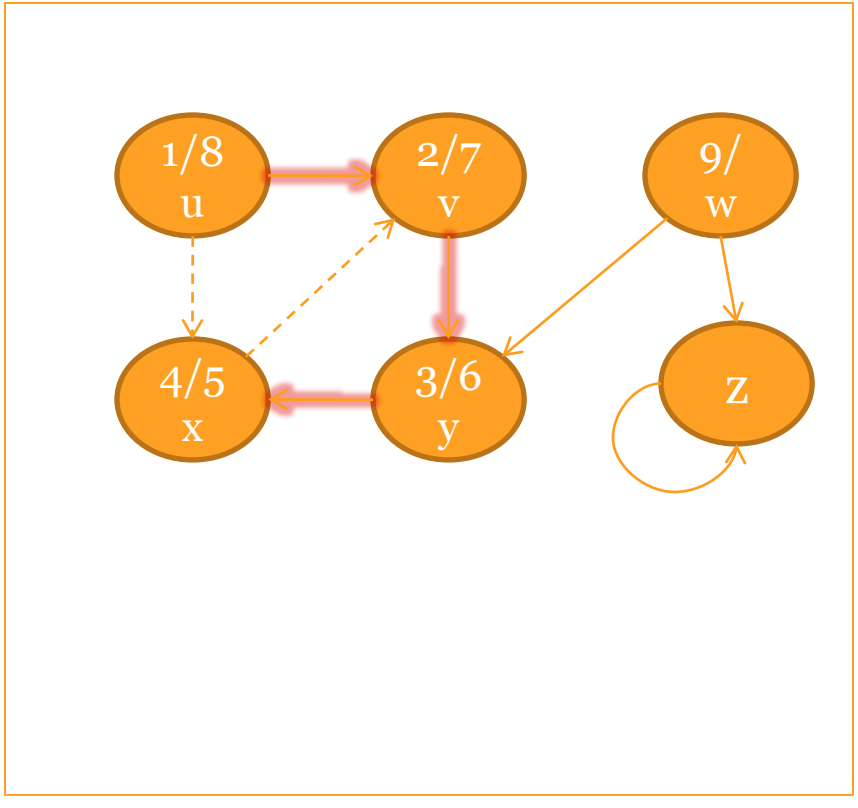


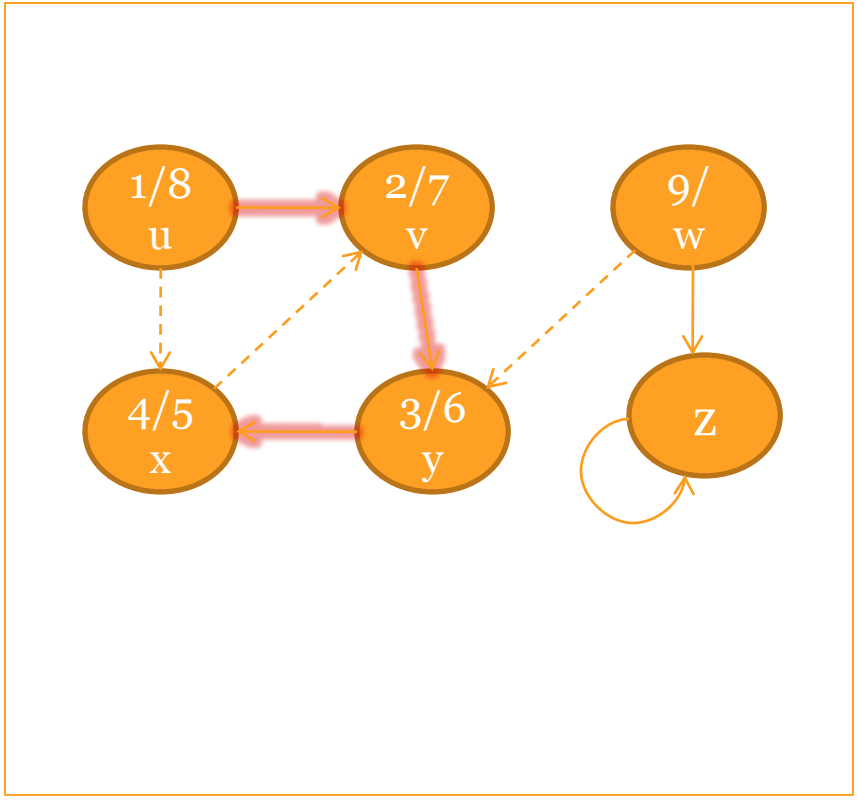
DFS example



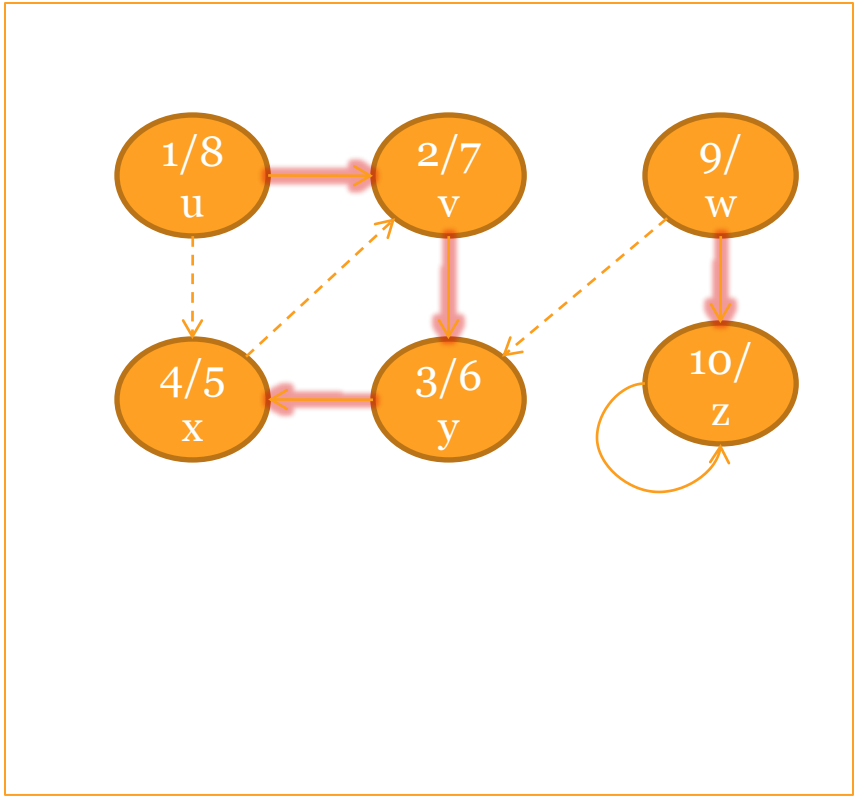


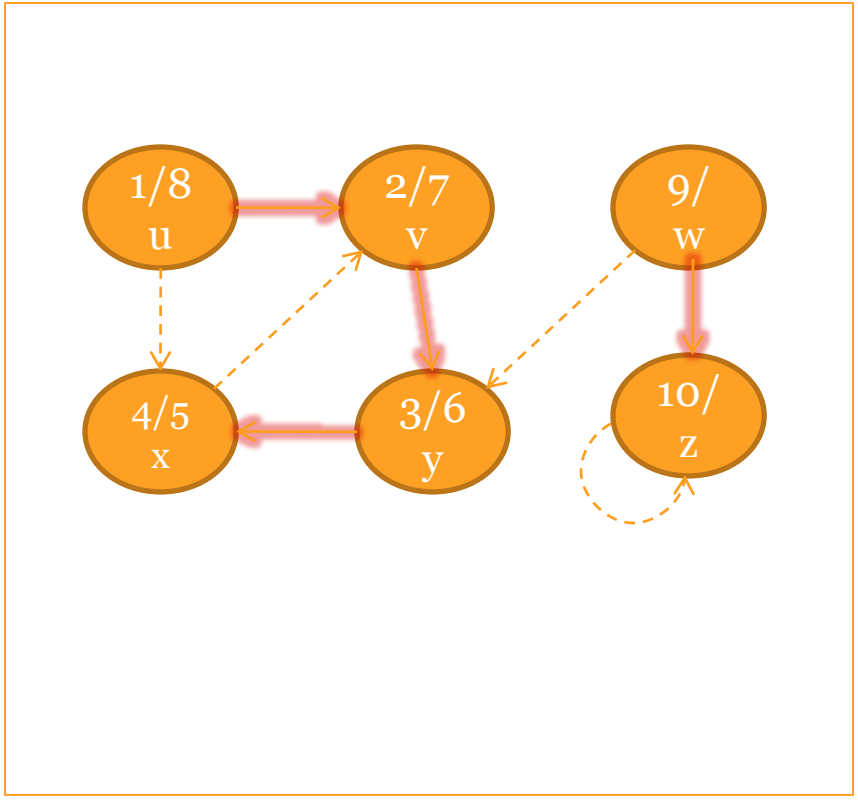
DFS example



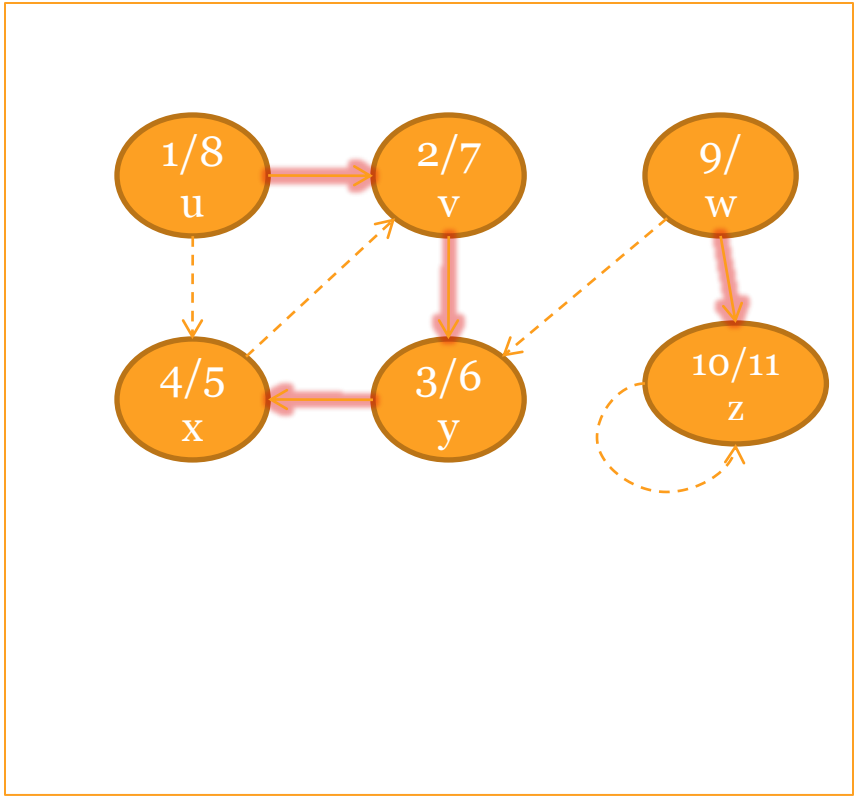


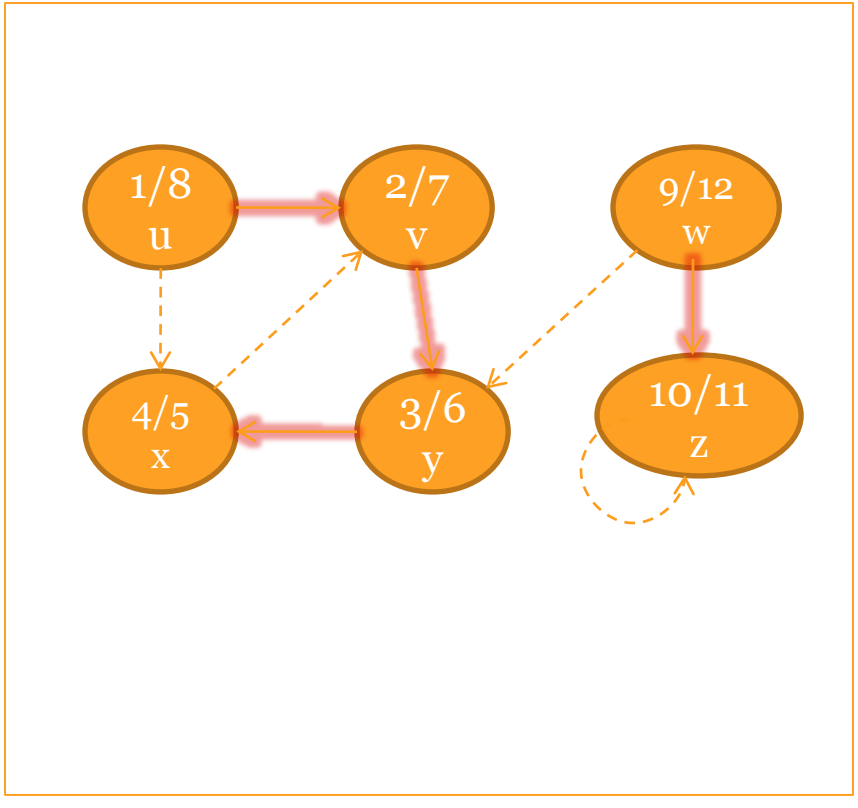
DFS example





DFS example





Recursive Algorithm

```
DFS(V, E)
for each u ∈ V
do visited[u] ← False
for each u ∈ V
    do if visited[u] = False
        then DFS-VISIT(u)
```

```
DFS-VISIT(u)
visited[u] ← True //discover u
for each v ∈ Adj[u] //explore (u, v)
    do if visited[v] = False
        Then DFS-VISIT(v)
visited[u] ← BLACK
```

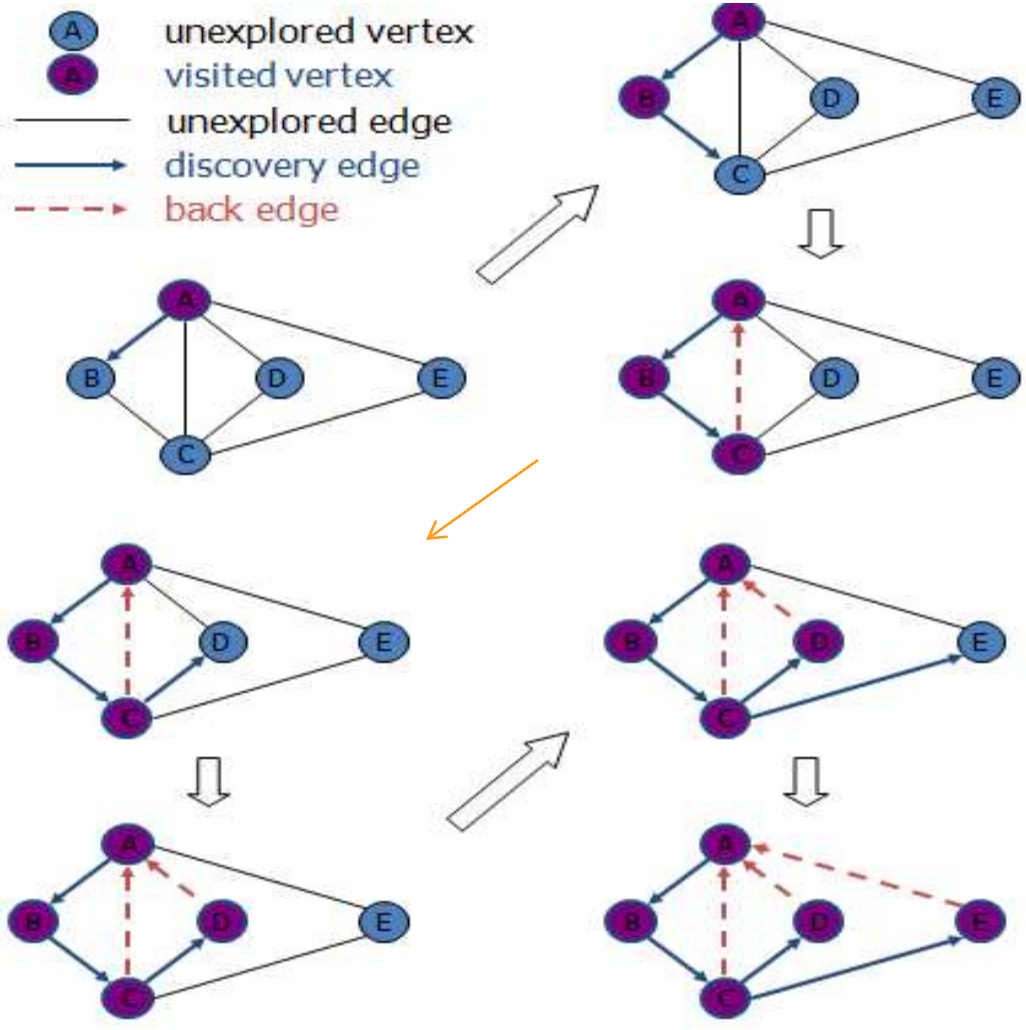
What we get free

Each call to DFS-visit gives us connected components of a graph.

DFS with labels

```
Algorithm DFS(G)
Input graph G
Output labelling of the edges of G as
    discovery edges and back edges
for all u ∈ G.vertices()
setLabel(u, Unexplored)
for all e ∈ G.edges()
setLabel(e, Unexplored)
for all v ∈ G.vertices()
if getLabel(v) = Unexplored
DFS(G, v)
```

```
Algorithm DFS(G, v)
Input start vertex v
setLabel(v, Visited)
for all e ∈ G.incidentEdges(v)
if getLabel(e) = Unexplored
w ← opposite(v,e)
if getLabel(w) = Unexplored
setLabel(e, Discovery)
DFS(G, w)
else
setLabel(e, Back)
```



Key points of label DFS

Property 1

DFS(*G*, *v*) visits all vertices and edges in connected component of *v*

Property 2

The discovery edges labeled by *DFS*(*G*, *v*) form a spanning tree of the connected component of *v*

- Each vertex is labeled twice
  - once as unexplored
  - once as Visited
- Each edge is labeled twice
  - once as unexplored
  - once as Discovery or Back
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

Path Finding

We can specialize the DFS to find path between two given vertices u and z

G is a connected Graph

Call DFS(G, u) with u as start vertex

Use a stack S to keep track of the path between the start vertex and the current vertex

As soon as destination vertex z is encountered, return the path as contents of the stack

```
Algorithm pathDFS(G, v, z)
      setLabel(v, Visited)
      S.push(v)
if  v = z
    return S.elements()
for all  e ∈ G.incidentEdges(v)
    if  getLabel(e) = Unexplored
        w ← opposite(v,e)
        if getLabel(w) = Unexplored
    setLabel(e, Discovery)
            S.push(e)
            pathDFS(G, w, z)
            S.pop(e)
        else
             setLabel(e, Back)
S.pop(v)
```

Cycle finding

We can specialize the DFS to find a simple cycle

Use stack S to keep track of path between start vertex and current vertex

As soon as a back edge (v, w) is encountered, return the cycle as portion of stack from top to vertex w

```
Algorithm cycleDFS(G, v, z)
    setLabel(v, Visited)
    S.push(v)
for all e ∈ G.incidentEdges(v)
  if getLabel(e) = Unexplored
    w ← opposite(v,e)
    S.push(e)
    if getLabel(w) = Unexplored
      setLabel(e, Discovery)
      pathDFS(G, w, z)
      S.pop(e)
    else
      T ← new empty stack
      repeat
        o ← S.pop()
        T.push(o)
      until o = w
      return T.elements()
S.pop(v)
```

DFS Complexity

Visits each node exactly once

Process each edge exactly twice (for an undirected graph)

Process each edge once (for an directed graph)

So complexity O(V+E)

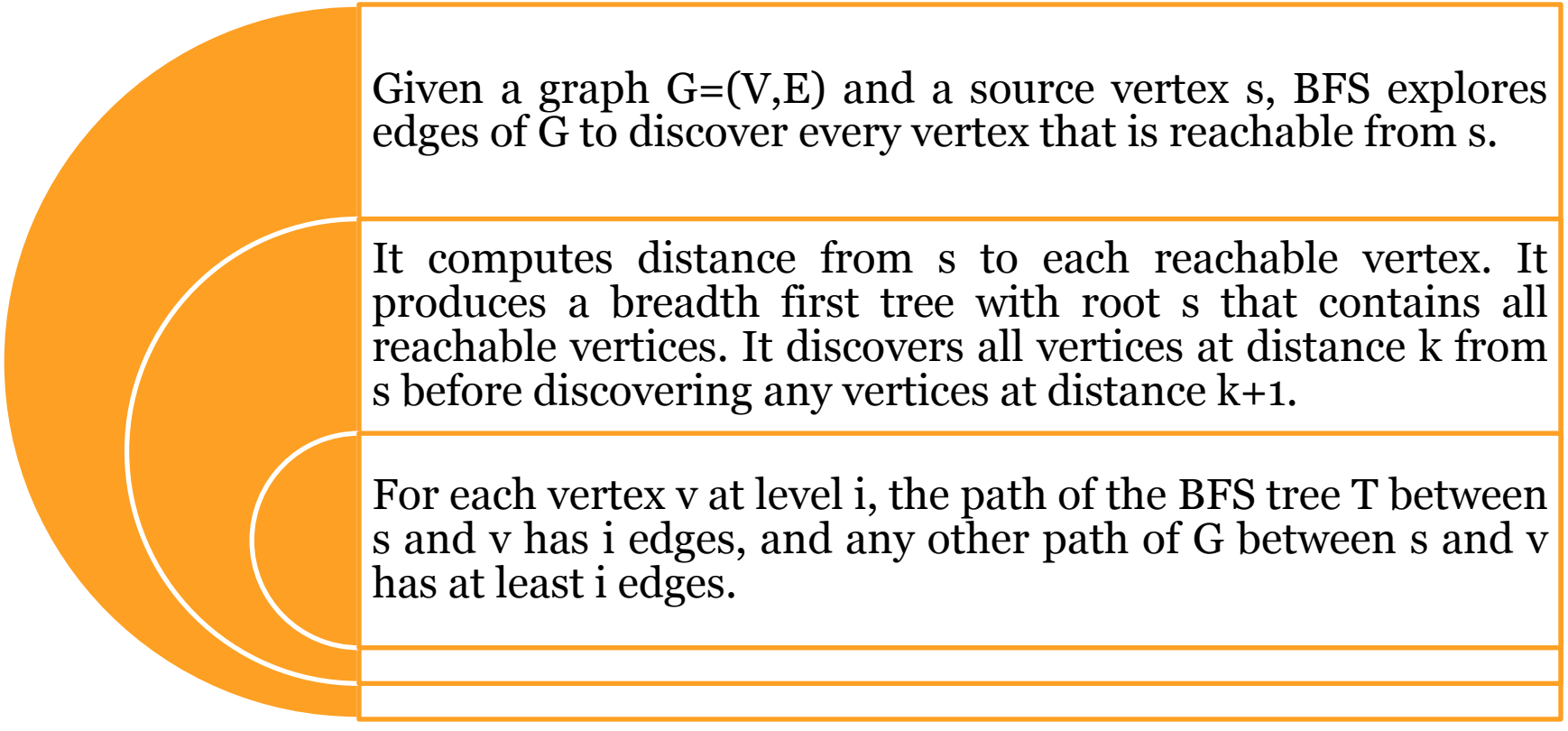
Adjacency matrix $O(V^2)$

Important Points

Post order traversal of DFS tree read in a reverse direction is topological sort in directed graphs.

DFS can do more things then BFS relatively because when it backs up it has lot of information then BFS.

Breadth First Search



BFS

BFS on a graph with n vertices and m edges takes $O(n + m)$ time

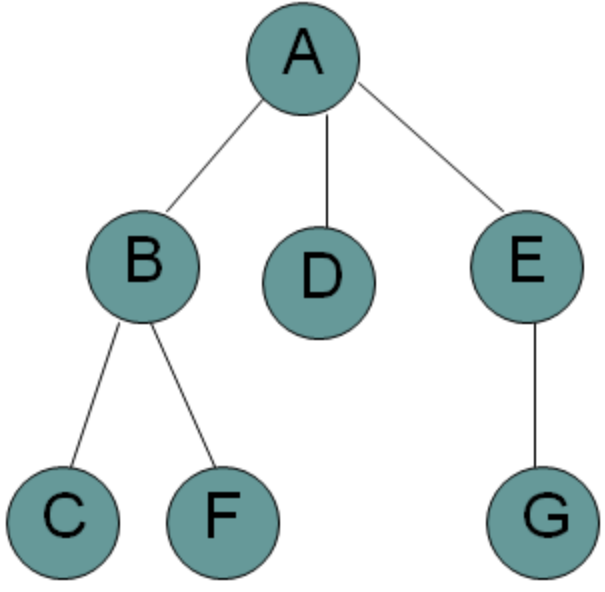
BFS can be further extended to solve other graph problems. Find and report a path with the minimum number of edges between two given vertices. To find the vertex with maximum number of edges from a vertex. Find a simple cycle, if there is one

BFS on trees

```
BFSTree(T)
Enqueue(Que, Root(T))
While !Empty(Que)
        v← Dequeue(Que)
        visit(v)
        for all Children(v)
         Push(Que, Children)
```

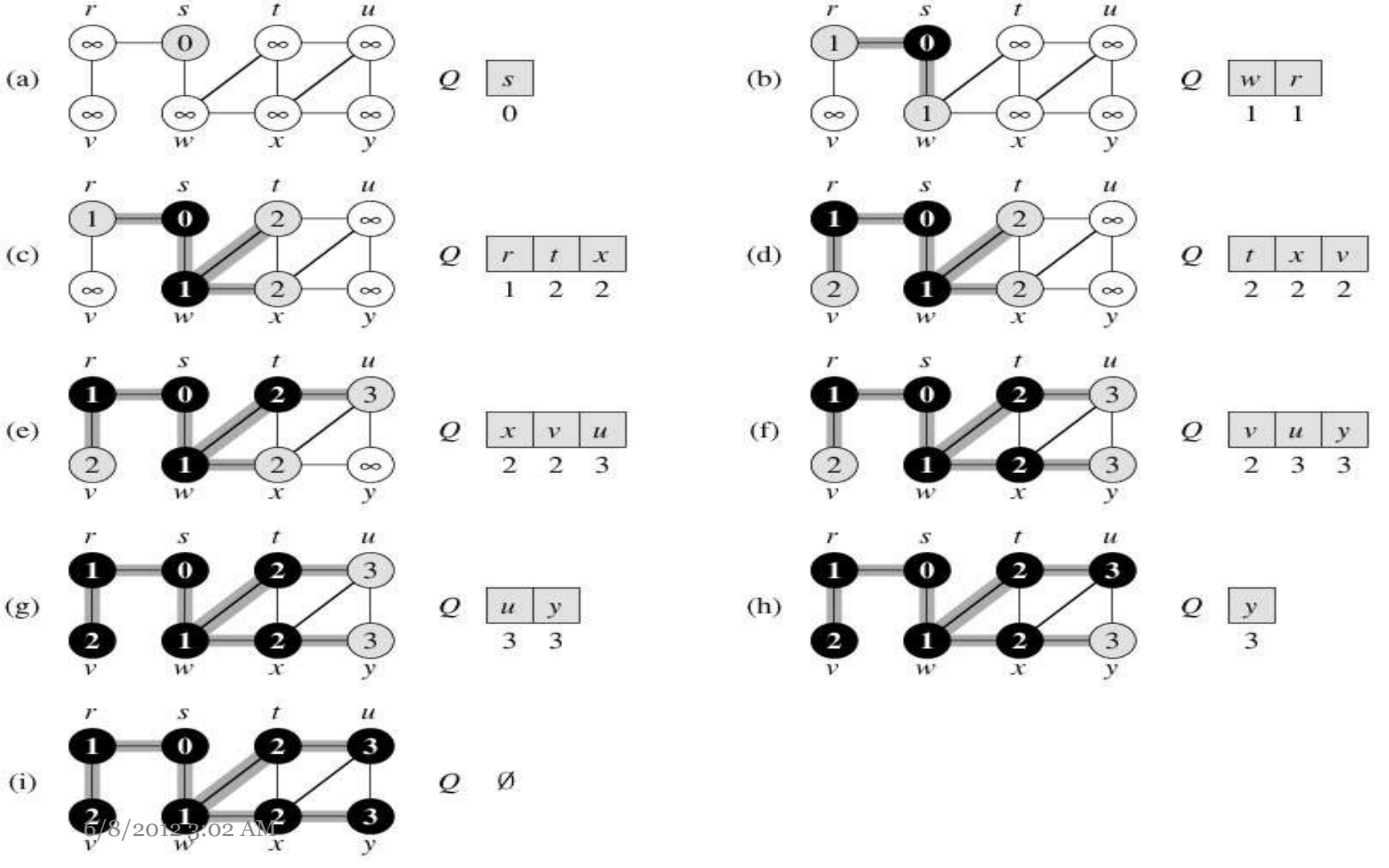
BFS tree example

ABDECFG is the order of BFS on tree



BFS search Algorithm

1. Enqueue the root node.
2. Dequeue a node and examine it.
   a) If the element sought is found in this node, quit the search and return a result.
   b) Otherwise Enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. If the queue is not empty, repeat from Step 2


(a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q
s
0
(b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q
w r
1 1
(c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q
r t x
1 2 2
(d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q
t x v
2 2 2
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q
x v u
2 2 3
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
∅

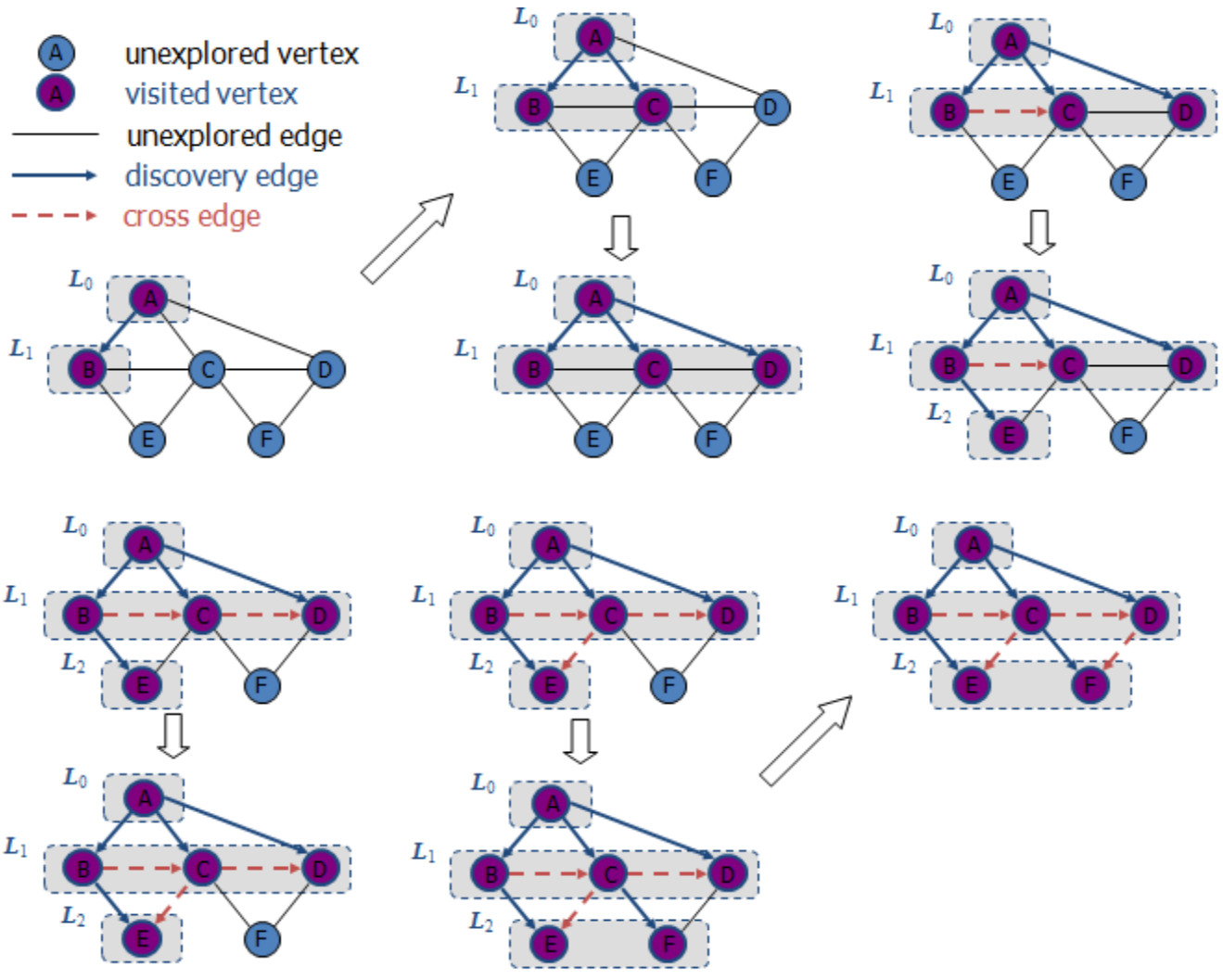
BFS algorithm

```
GraphBFS(G,s)
        For each Vertex in Graph
                dist[v] ← ∞
        dist[s] ← 0
Enqueue(Que, s)
While !Empty(Que)
         u← Dequeue(Que)
         visit(u)
         for each (u, v) in Edge list
                 if dist[v] ← ∞
                 Enqueue(Que, v)
                  dist[v] ← dist[u] + 1
```

BFS with labeling

```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
           and partition of the
           vertices of G
for all u ∈ G.vertices()
   setLabel(u, Unexplored)
for all e ∈ G.edges()
   setLabel(e, Unexplored)
for all v ∈ G.vertices()
   if getLabel(v) = Unexplored
     BFS(G, v)
```

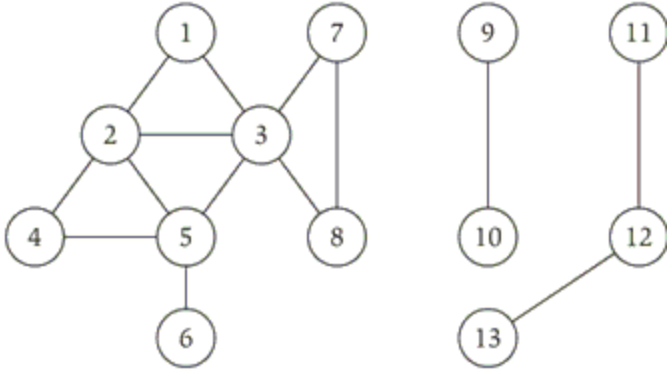
```
Algorithm BFS(G, s)
      L0 ← new empty sequence
L0.insertLast(s)
setLabel(s, Visited)
i ← 0
while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
          for all e ∈ G.incidentEdges(v)
          if getLabel(e) = Unexplored
              w ← opposite(v,e)
              if getLabel(w) = Unexplored
                  setLabel(e, Discovery)
                  setLabel(w, Visited)
                  Li+1.insertLast(w)
              else
                  setLabel(e, Cross)
                  i ← i +1
```



Connected Components

Find all nodes reachable from s.

BFS will find all the connected components



Label all nodes as 0, Do BFS and change visited components to 1, Again start BFS from new node with 0 label and so on

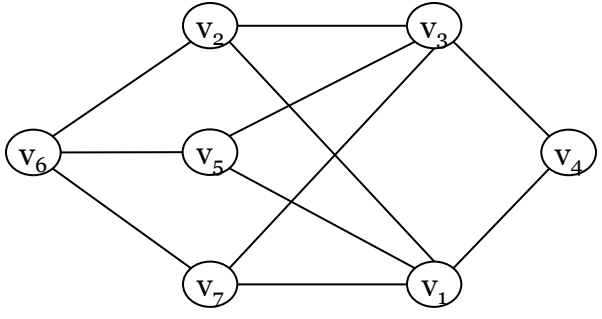
C1 {1,2,3,4,5,6,7,8} C2{9,10} C3{11,12,13}

Testing Bipartiteness

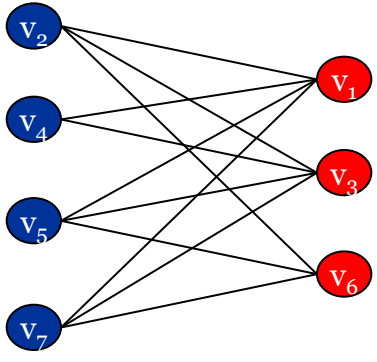
Given a graph G, is it bipartite?

Many graph problems become easier if underlying graph is bipartite

Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



a bipartite graph G

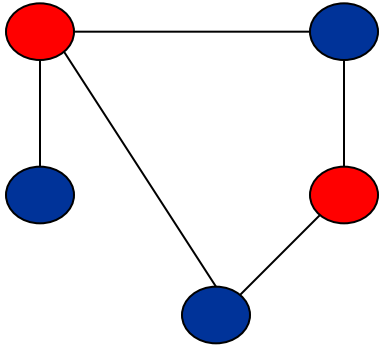


another drawing of G

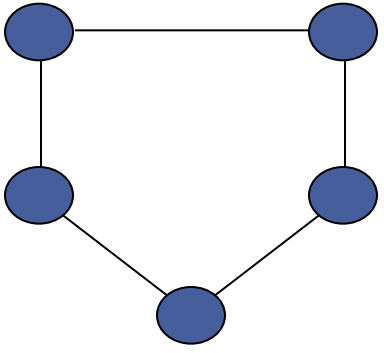
An Obstruction to Bipartiteness

If a graph G is bipartite, it cannot contain an odd length cycle.

Not possible to 2-color the odd cycle, let alone G.



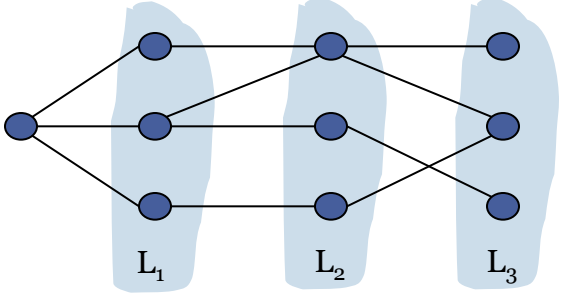
bipartite
(2-colorable)



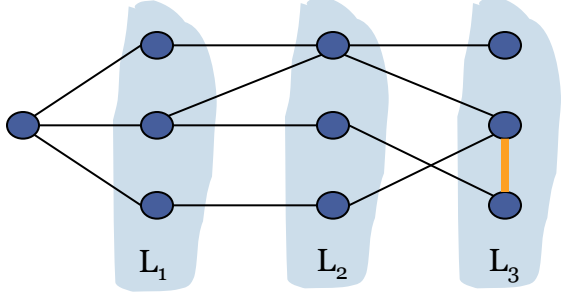
not bipartite
(not 2-colorable)

Bipartite Graphs

- Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.
  - (i) No edge of G joins two nodes of the same layer, and G is bipartite.
  - (ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



Case (i)

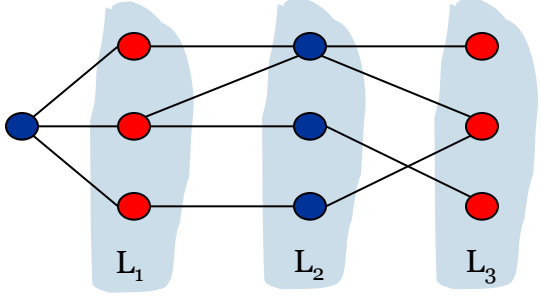


Case (ii)

Bipartite Graphs

Suppose no edge joins two nodes in the same layer, this implies all edges join nodes on different level.

Bipartition: red = nodes on odd levels, blue = nodes on even levels.



Bipartite Graphs

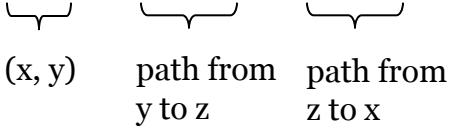
Suppose (x, y) is an edge with x, y in same level L_j.

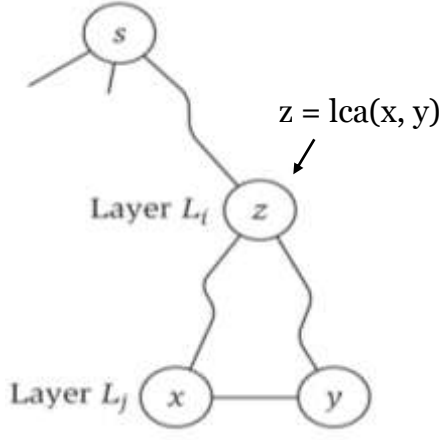
Let z = lca(x, y) = lowest common ancestor.

Let L_i be level containing z.

Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.

Its length is 1 + (j-i) + (j-i), which is odd. ▪

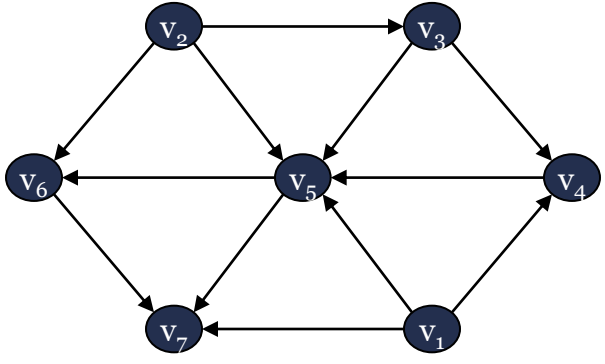




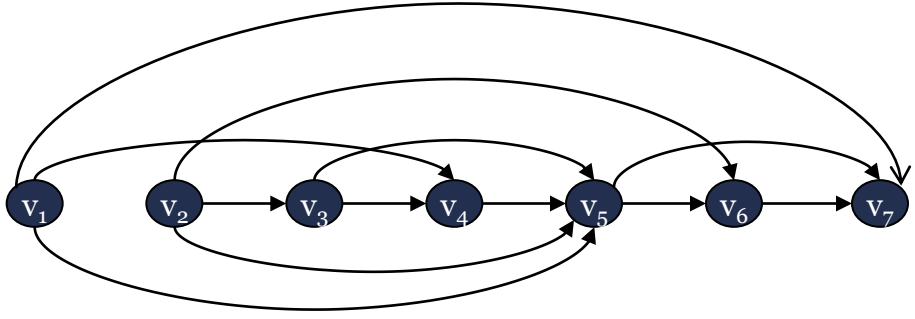
Directed Acyclic Graphs

A topological order of a directed graph G = (V, E) is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.

An ordering of the nodes that obeys the dependencies i.e. an activity can't happen until it's dependent activities have happened



a DAG



a topological ordering

Applications of DAG

Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.

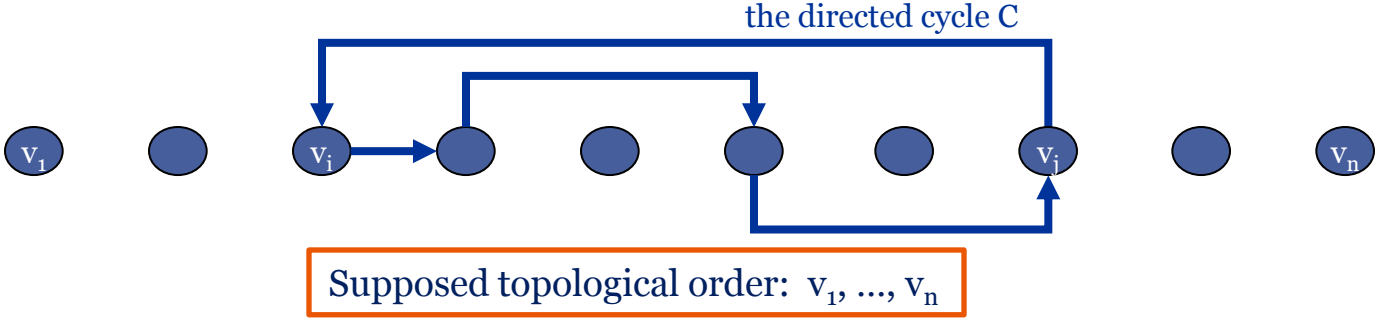
Course prerequisite graph: course v_i must be taken before v_j.

Compilation: module v_i must be compiled before v_j.

Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

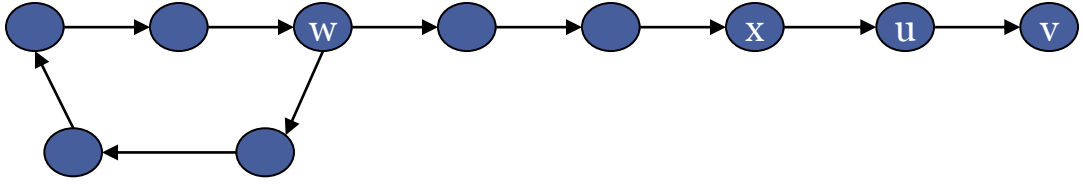
Directed Acyclic Graphs

If G has a topological order, then G is a DAG.
Pf. Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
By our choice of i, we have $i < j$.
On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction.



Directed Acyclic Graphs

If G is a DAG, then G has a node with no incoming edges.



Directed Acyclic Graphs

If G is a DAG, then G has a topological ordering.

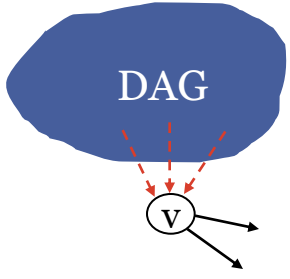
Base case: true if n = 1.

Given DAG on n > 1 nodes, find a node v with no incoming edges.

G - { v } is a DAG, since deleting v cannot create cycles.

By inductive hypothesis, G - { v } has a topological ordering.

Place v first in topological ordering; then append nodes of G - { v } in topological order. This is valid since v has no incoming edges. ▪



Topological ordering Algorithm

To compute a topological ordering of G:

Find a node v with no incoming edges and order it first

Delete v from G

Recursively compute a topological ordering of G-{v}

and append this order after v

Topological Sort

```
Topological-sort(G)
        for all edges (u, v) in E
                incoming[v]← incoming[v] +1
        for all v in V
                if incoming[v]=0
                        Enqueue(S, v)
        While S not empty
                u ← Dequeue(S)
                add u to linked list
                for each edge(u, v) in E
                        incoming[v] ← incoming[v]-1
                        if active[v]=0
                                Enqueue(S, v)
```

Topological Sorting Algorithm: Running Time

Algorithm finds a topological order in O(V + E) time.

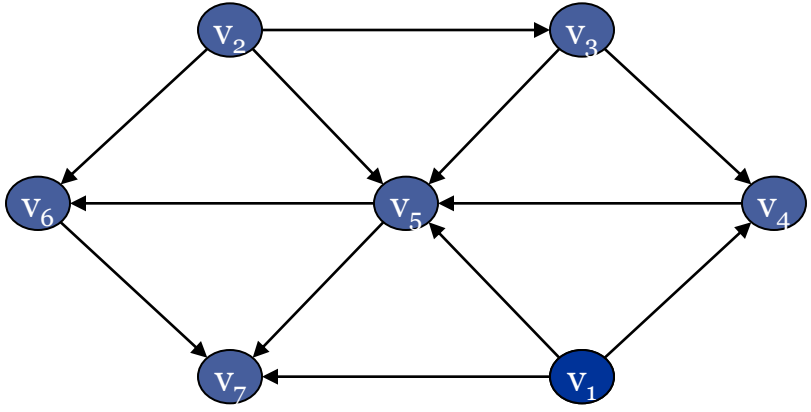
- `count[w]` = remaining number of incoming edges
- S = set of remaining nodes with no incoming edges

– Initialization: O(m + n) via single scan through graph.

– Update: to delete v

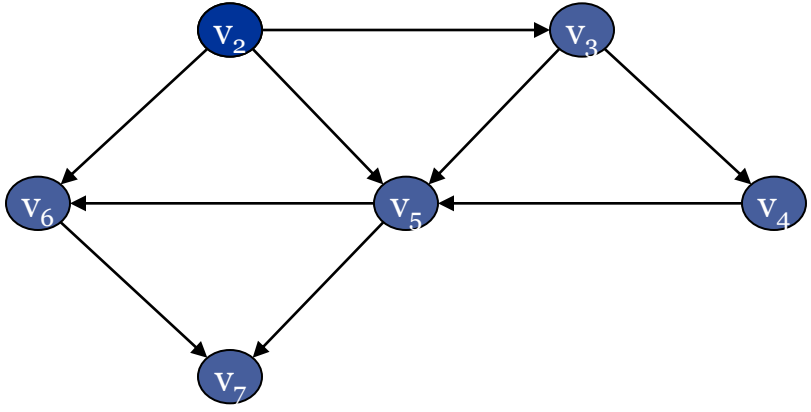
- remove v from S
- decrement `count[w]` for all edges from v to w, and add w to S if c `count[w]` hits 0
- this is O(1) per edge ▪

Topological Ordering Algorithm: Example



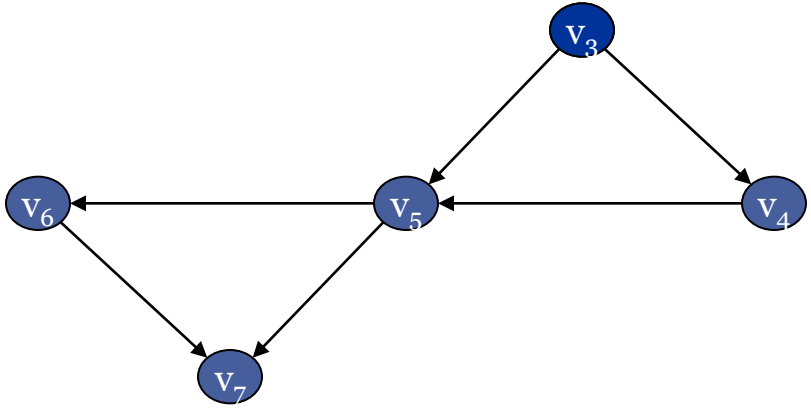
Topological order:

Topological Ordering Algorithm: Example



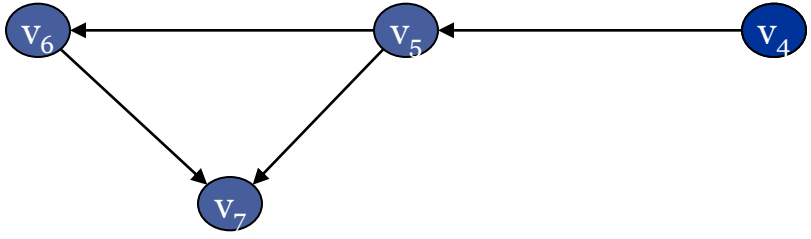
Topological order: v_1

Topological Ordering Algorithm: Example



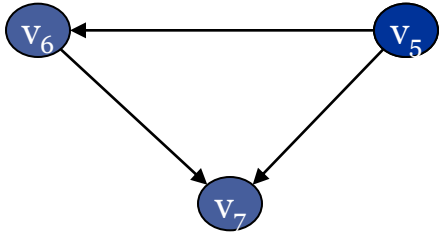
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



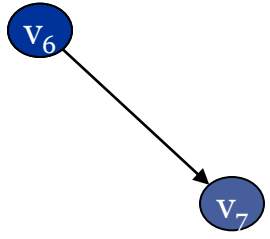
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

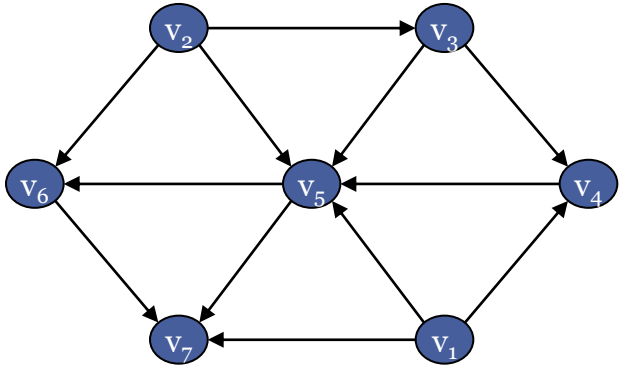


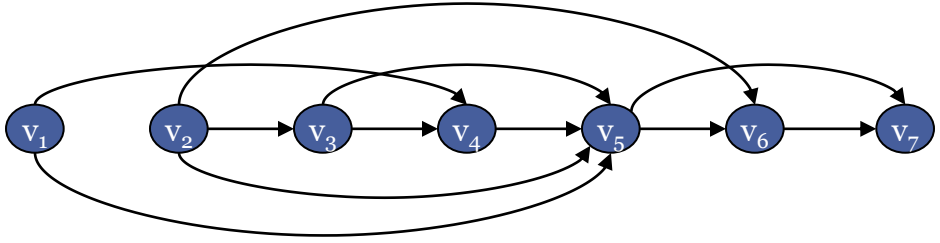
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

Topological Ordering Algorithm: Example

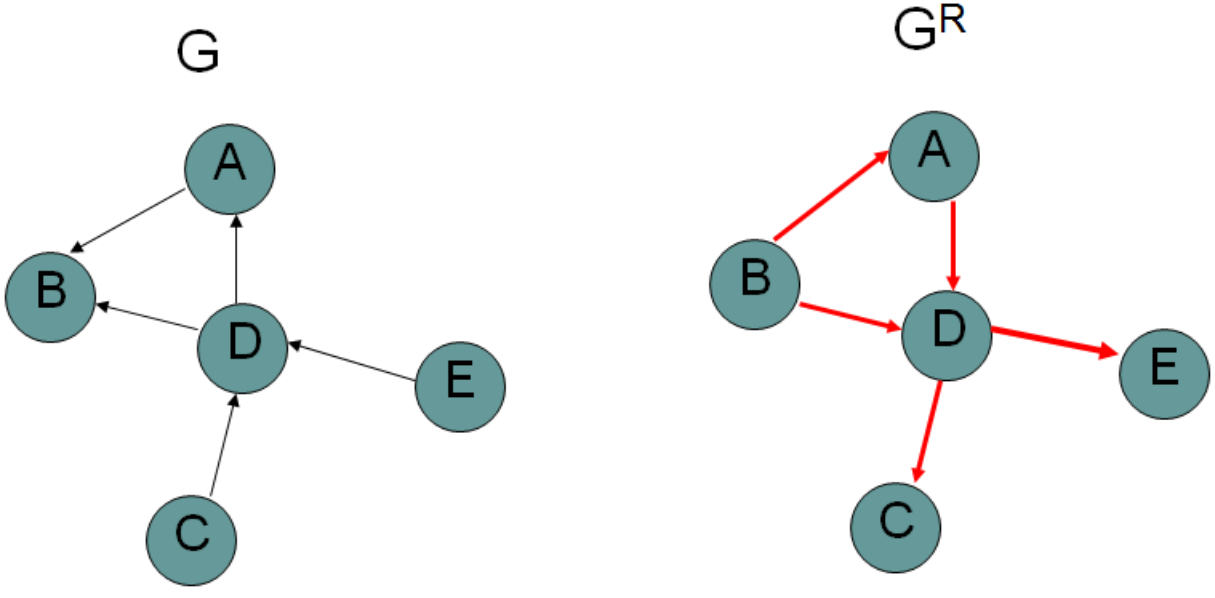




Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Transpose of a graph

Transpose of a graph is achieved by reversing all its edges.
Running time is O(V+E)



Strongly Connected

- StronglyConnectedcomponents(G)
- Run DFS from some node u and calculate the finishing time for each model
- Create Graph G^R by reversing all edge directions
- Run DFS on G^R in reverse order of finishing times

Cycle

Undirected graphs it is easy

If we see a node we had already seen during BFS or DFS, it is a cycle

In Directed Graph

Call Topological Sort

If the length of the list returned ≠ |V| then a cycle exists

Questions, Suggestions and Comments

Question 1

Which out of the following properly defines chromatic number of a graph.

A) If a graph has been colored with K colors then K is the chromatic number of the graph.

B) The smallest Number of colors to be used to color a graph such that no two adjacent vertices have the same color.

C) Maximum number of colors to be used to color a graph such that no two adjacent vertices have the same color.

D) Maximum number of colors to be used to color a graph such that all cliques in the graph use different colors.

Question 2

Consider a simple connected graph G with n vertices and n edges (n>2). Then which of the following statements are true?

A) G has no cycles

B) The graph obtained by removing any edges from G is not considered connected

C) G has at least one cycle

D) The graph is a tree

Question 3

if v is the number of vertices, e is the number of edges and f is the number of faces (regions bounded by edges, including the outer region) of a planar graph then as per the Euler's Formula

A) v+f = 2+e

B) v+e= 2+f

C) e+f = 2+ v

D) e+v = f-2