



# Data Structures and Algorithms Design

**BITS Pilani**  
Hyderabad Campus

Febin.A.Vahab

# CONTACT SESSION 6 -PLAN



Contact Sessions(#)	List of Topic Title	Text/Ref Book/external resource
6	Graphs - Terms and Definitions, Properties, Representations (Edge List, Adjacency list, Adjacency Matrix), Graph Traversals (Depth First and Breadth First Search )	T1: 6.1, 6.2, 6.3

# Depth-First Search



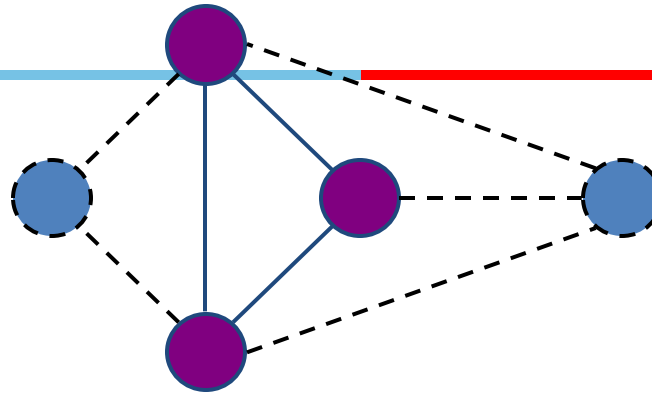
- Definitions
  - Subgraph
  - Connectivity
  - Spanning trees and forests
- Depth-first search
  - Algorithm
  - Example
  - Properties
  - Analysis
- Applications of DFS
  - Cycle finding
  - Path finding

# SUBGRAPHS

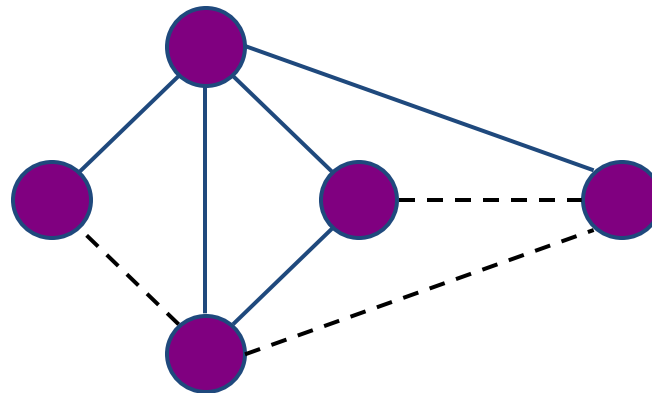


- Subgraphs
- A subgraph  $S$  of a graph  $G$  is a graph such that
  - The vertices of  $S$  are a subset of the vertices of  $G$
  - The edges of  $S$  are a subset of the edges of  $G$
- A spanning subgraph of  $G$  is a subgraph that contains all the vertices of  $G$

# SUBGRAPHS



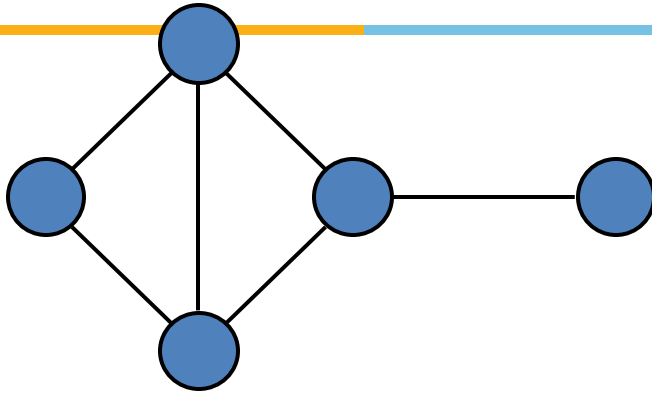
Subgraph



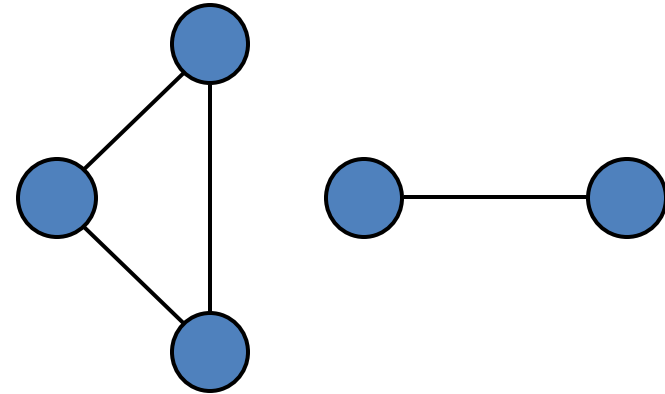
Spanning subgraph

- A graph is **connected** if there is a path between every pair of vertices
- A connected component of a graph  $G$  is a maximal connected subgraph of  $G$
- A directed graph  $G$  is **strongly connected** if:
  - For any two vertices  $u$  and  $v$ :
  - There is a directed path  $u \rightarrow v$ , and
  - There is a directed path  $v \rightarrow u$

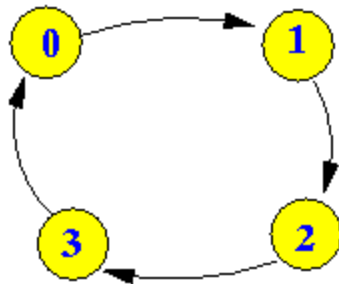
# Connected graph



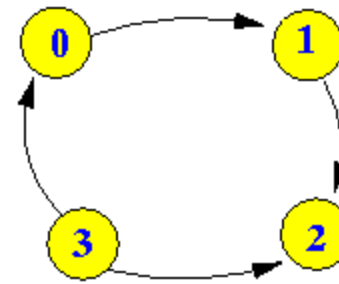
Connected graph



Non connected graph with two connected components



Strongly Connected

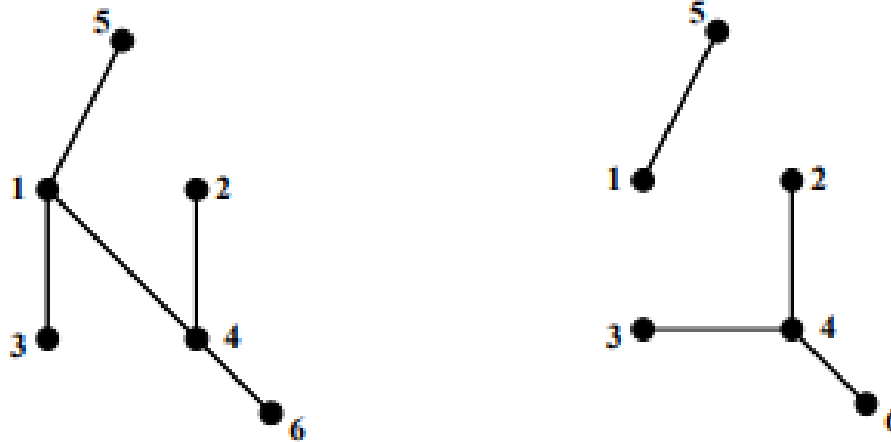


Not Strongly Connected

# Trees and Forests

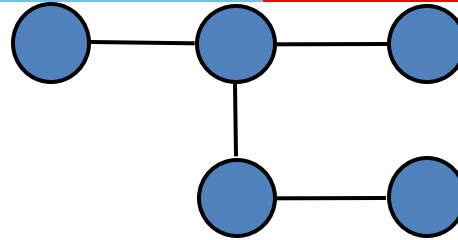


- A tree is a connected graph with no cycles.
- A forest is a graph with each connected component a tree

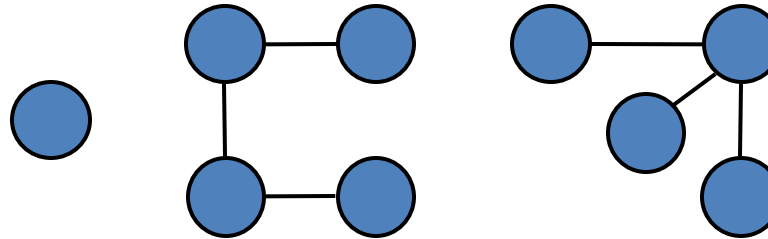




# Trees and Forests



Tree



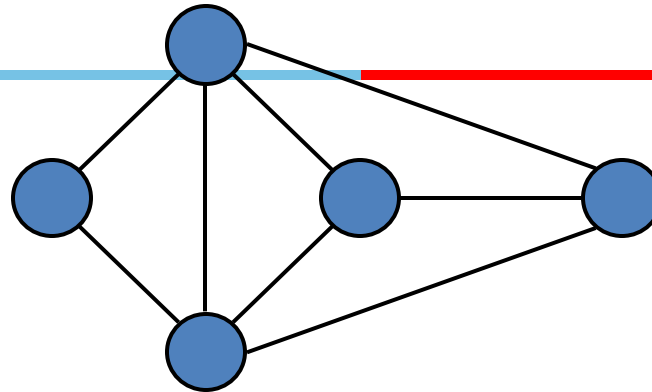
Forest

# Spanning Trees

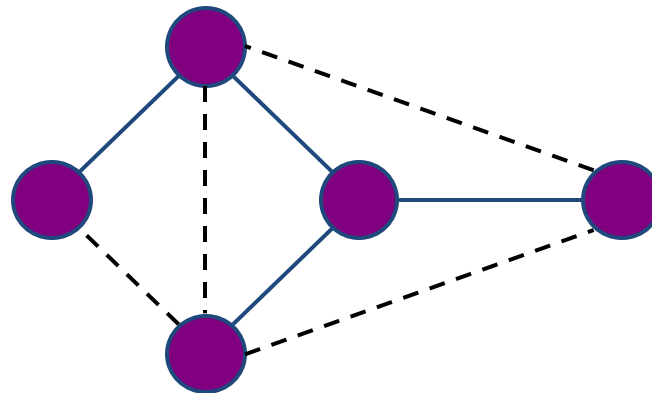


- A spanning tree of a connected graph is a spanning subgraph that is a tree:
- which includes all of the vertices of  $G$ , with minimum possible number of edges

# Spanning Tree



Graph



Spanning tree

# Subgraphs, trees-Example

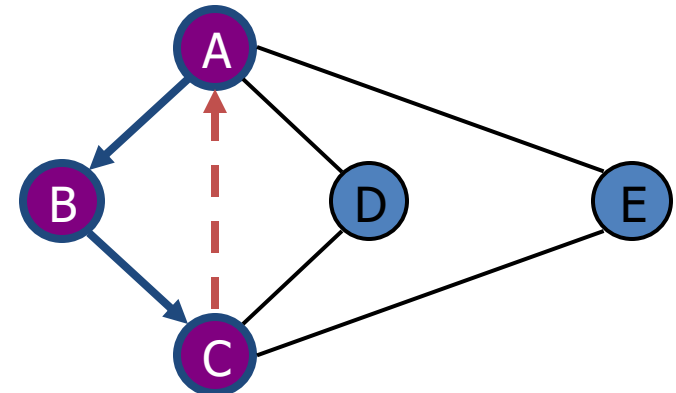
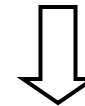
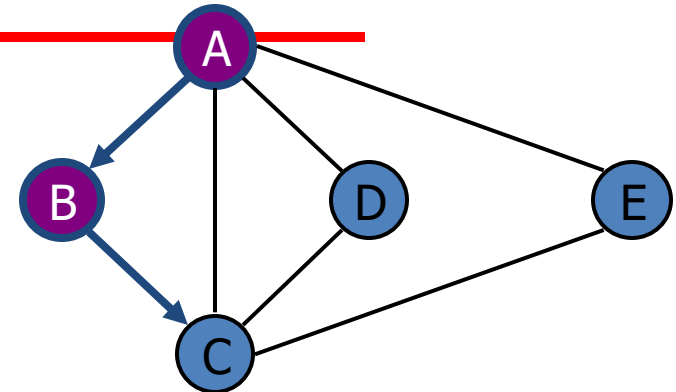
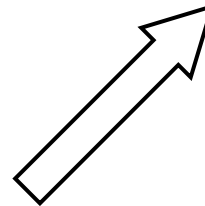
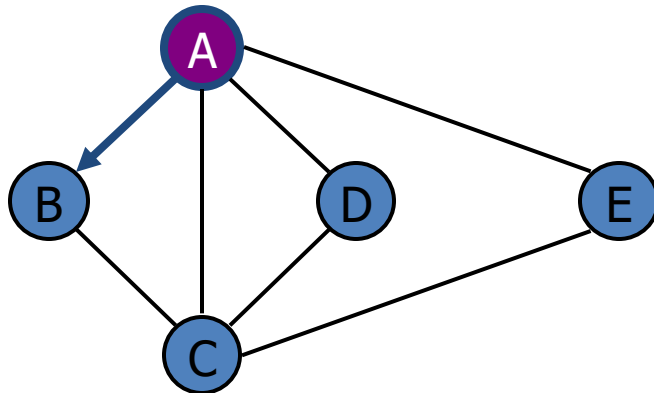
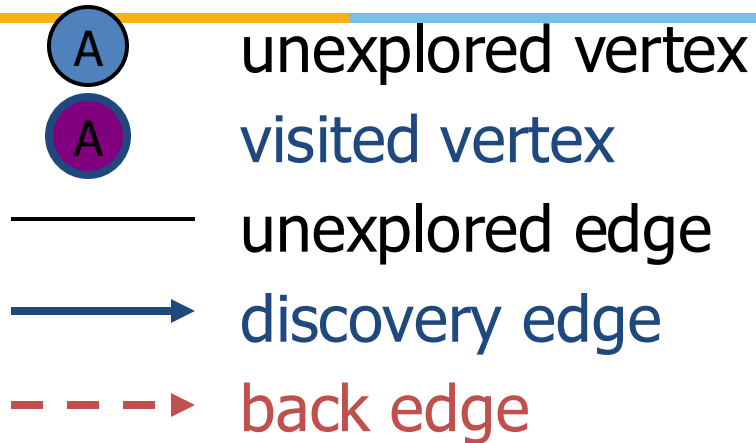
- Perhaps the most talked about graph today is the Internet, which can be viewed as a graph whose vertices are computers and whose (undirected) edges are communication connections between pairs of computers on the Internet.
- The computers and the connections between them in a single domain, like <http://www.bits-pilani.ac.in/> , form a subgraph of the Internet. If this subgraph is connected, then two users on computers in this domain can send e-mail to one another without having their information packets ever leave their domain.
- Suppose the edges of this subgraph form a spanning tree. This implies that, even if a single connection goes down (for example, because someone pulls a communication cable out of the back of a computer in this domain), then this subgraph will no longer be connected.

# Depth-First Search

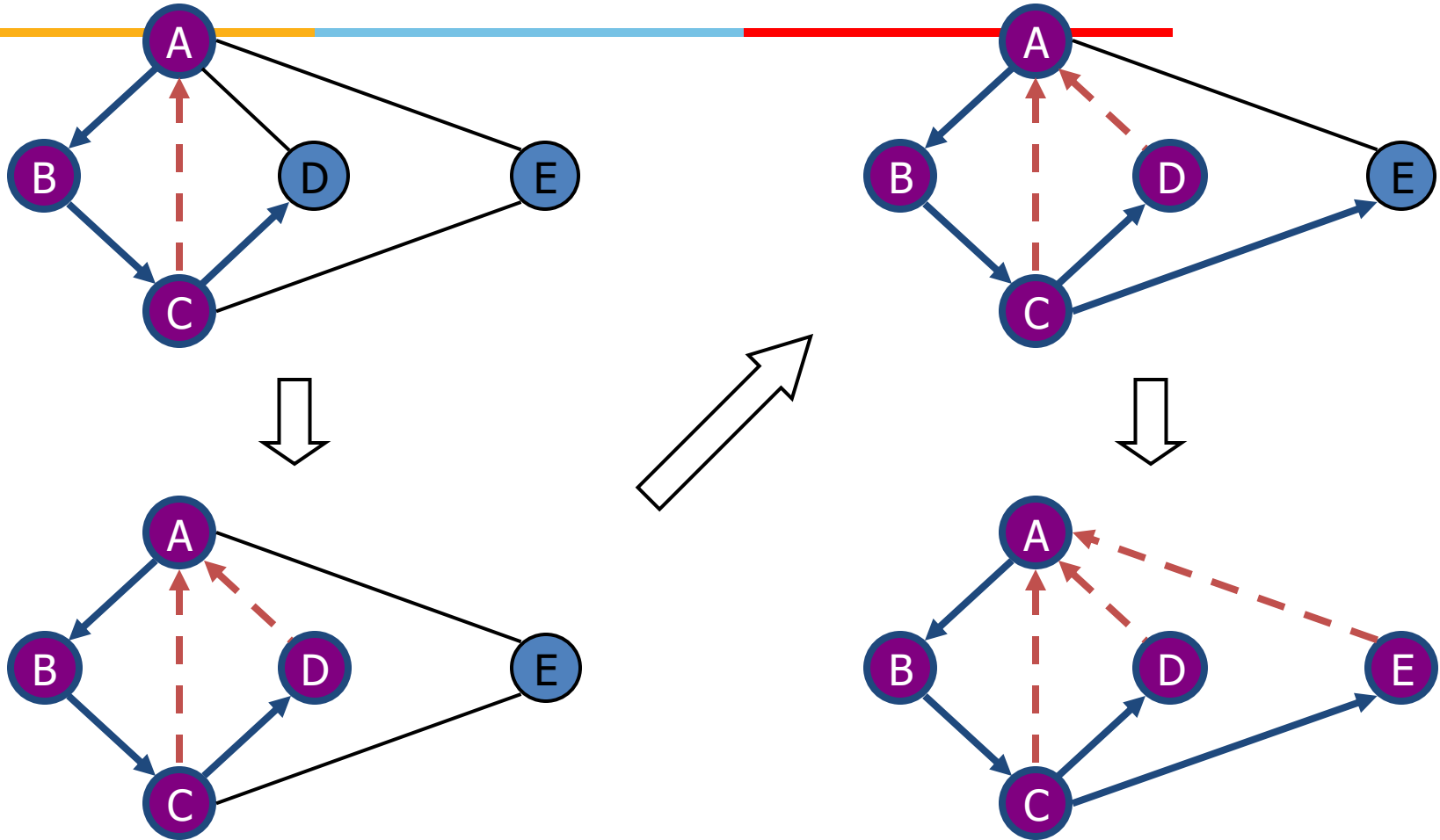


- Depth-first search (DFS) is a general technique for traversing a graph
- Search “deeper” in the graph whenever possible
- Explores edges out of the most recently discovered vertex that still has unexplored edges leaving it.
- Once all of  $v$ 's edges have been explored, the search “backtracks” to explore edges leaving the vertex from which  $v$  was discovered.
- This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source.
- The algorithm repeats this entire process until it has discovered every vertex

# Depth-First Search



# Depth-First Search



# Depth-First Search



- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge ) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



# Depth-First Search-Properties

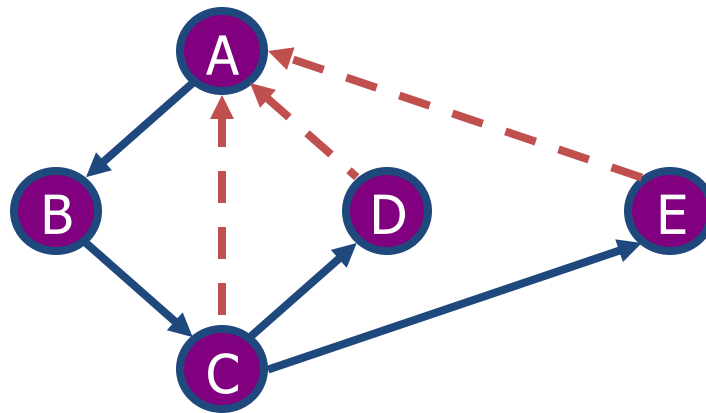


## Property 1

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

## Property 2

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



# Depth-First Search



- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

**Algorithm** *DFS*(*G*)

**Input** graph *G*

**Output** labeling of the edges of *G* as discovery edges and back edges

**for all** *u*  $\in$  *G.vertices*()

*setLabel*(*u*, *UNEXPLORED*)

**for all** *e*  $\in$  *G.edges*()

*setLabel*(*e*, *UNEXPLORED*)

**for all** *v*  $\in$  *G.vertices*()

**if** *getLabel*(*v*) = *UNEXPLORED*

*DFS*(*G*, *v*)

# Depth-First Search



## Algorithm $DFS(G, v)$

**Input** graph  $G$  and a start vertex  $v$  of  $G$

**Output** labeling of the edges of  $G$  in the connected component of  $v$  as discovery edges and back edges

$setLabel(v, VISITED)$

for all  $e \in G.incidentEdges(v)$

if  $getLabel(e) = UNEXPLORED$

$w \leftarrow G.opposite(v, e)$

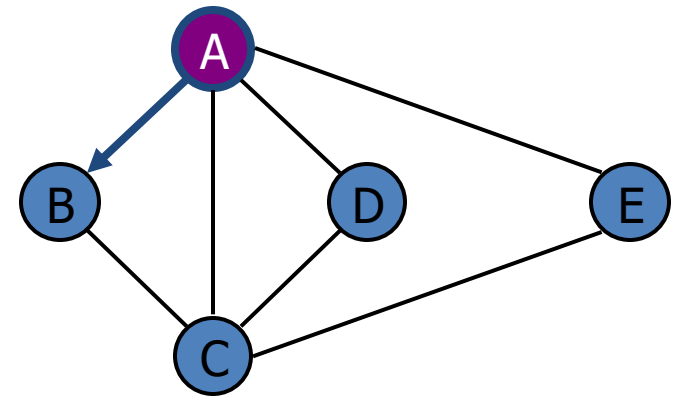
if  $getLabel(w) = UNEXPLORED$

$setLabel(e, DISCOVERY)$

$DFS(G, w)$

else

$setLabel(e, BACK)$



# Analysis of DFS



- Setting/getting a vertex/edge label takes  $O(1)$  time
- 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
  - Recall that  $\sum_v \deg(v) = 2m$

# Depth-First Search



- A DFS traversal of a graph  $G$ 
  - Visits all the vertices and edges of  $G$
  - Determines whether  $G$  is connected
  - Computes the connected components of  $G$
  - Computes a spanning tree of  $G$
  - Computing a cycle in  $G$ , or reporting that  $G$  has no cycles
  - Find and report a path between two given vertices

# Path Finding



- We can specialize the DFS algorithm to find a path between two given vertices  $u$  and  $z$  using the template method pattern
- We call  $DFS(G, u)$  with  $u$  as the start vertex
- We 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, we return the path as the contents of the stack

# Path Finding



```
Algorithm pathDFS( $G, v, z$ )
  setLabel( $v, VISITED$ )
  S.push( $v$ )
  if  $v = z$ 
    return S.elements()
  for all  $e \in G.incidentEdges(v)$ 
    if getLabel( $e$ ) = UNEXPLORED
       $w \leftarrow opposite(v, e)$ 
      if getLabel( $w$ ) = UNEXPLORED
        setLabel( $e, DISCOVERY$ )
        S.push( $e$ )
        pathDFS( $G, w, z$ )
        S.pop()                                {  $e$  gets popped }
      else
        setLabel( $e, BACK$ )
  S.pop()                                    {  $v$  gets popped }
```

# Cycle Finding



- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack  $S$  to keep track of the path between the start vertex and the current vertex
- As soon as a back edge  $(v, w)$  is encountered, we return the cycle as the portion of the stack from the top to vertex  $w$



# Cycle Finding



```
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()
      else
        C ← new empty stack
        repeat
          o ← S.pop()
          C.push(o)
        until o = w
        return C.elements()
  S.pop()
```

# DFS:R2-Chapter 22

DFS( $G$ )

```

1  for each vertex  $u \in G.V$ 
2       $u.color = WHITE$ 
3       $u.\pi = NIL$ 
4   $time = 0$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.color == WHITE$ 
7          DFS-VISIT( $G, u$ )

```

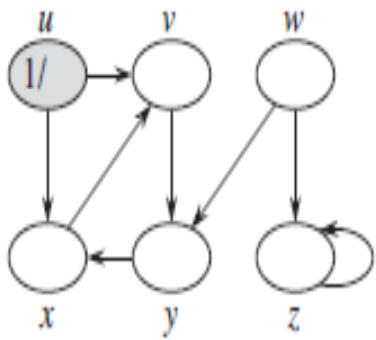
DFS-VISIT( $G, u$ )

```

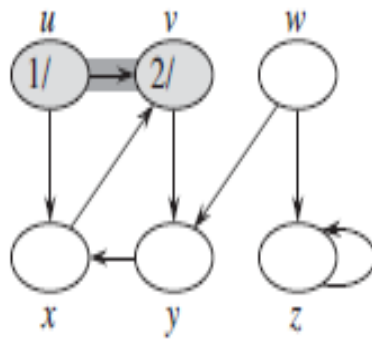
1   $time = time + 1$                                 // white vertex  $u$  has just been discovered
2   $u.d = time$ 
3   $u.color = GRAY$ 
4  for each  $v \in G.Adj[u]$                             // explore edge  $(u, v)$ 
5      if  $v.color == WHITE$ 
6           $v.\pi = u$ 
7          DFS-VISIT( $G, v$ )
8   $u.color = BLACK$                                 // blacken  $u$ ; it is finished
9   $time = time + 1$ 
10  $u.f = time$ 

```

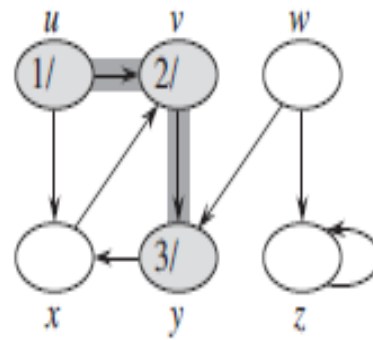
# DFS:R2-Chapter 22



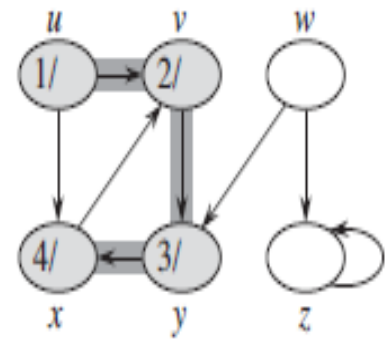
(a)



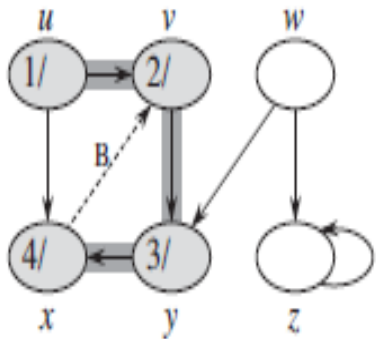
(b)



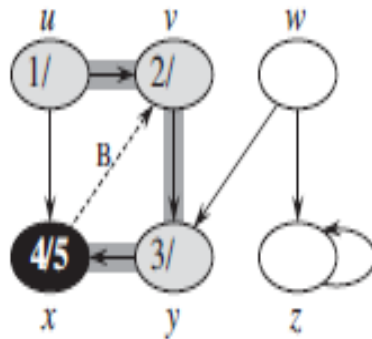
(c)



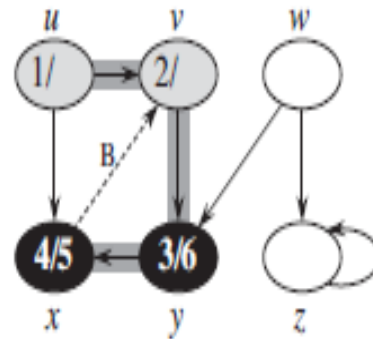
(d)



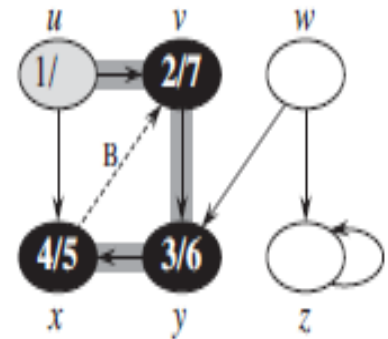
(e)



(f)

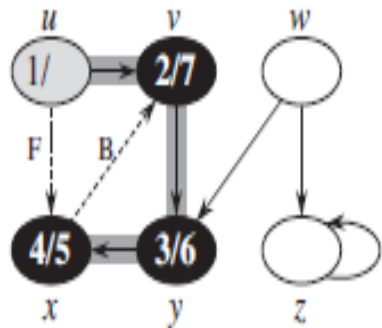


(g)

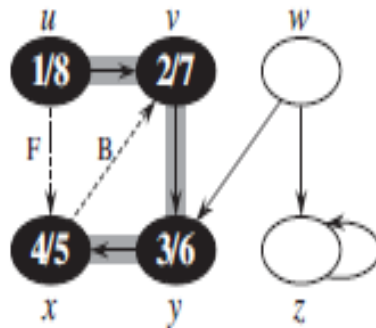


(h)

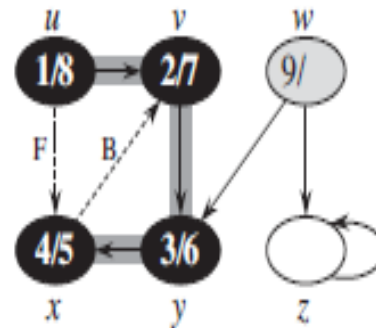
# DFS:R2-Chapter 22



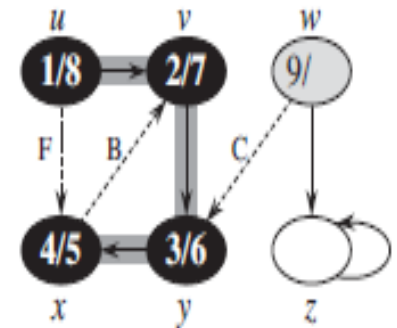
(i)



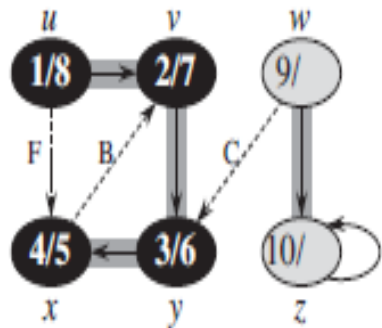
(j)



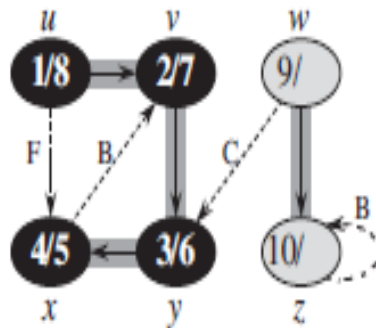
(k)



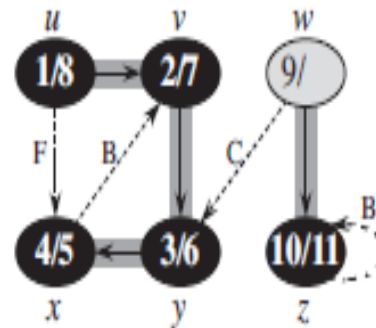
(l)



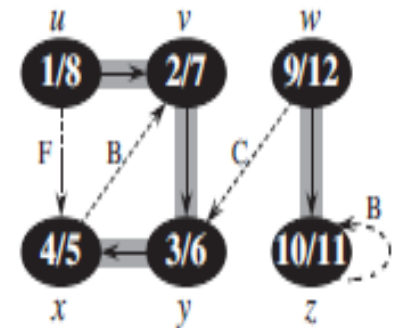
(m)



(n)



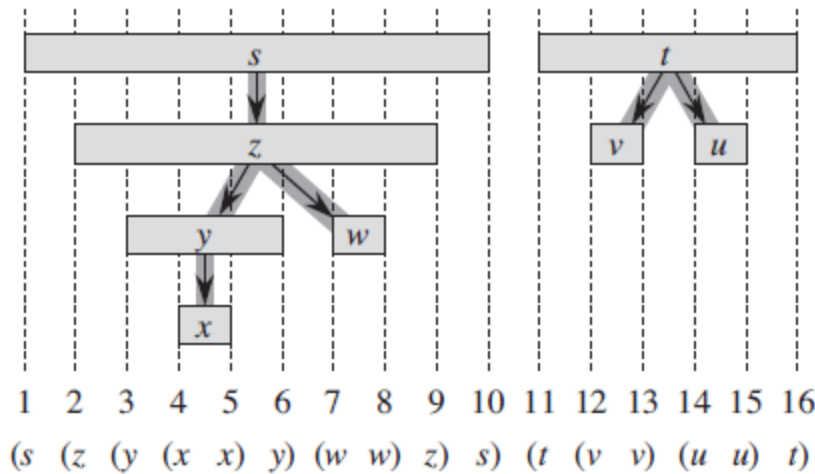
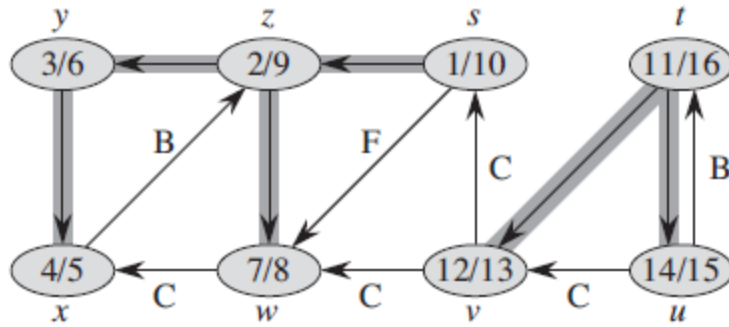
(o)



(p)

# DFS:R2-Chapter 22

## Paranthesis Structure



**How can DFS be used to find the connected components of a graph!**

**Can you implement it????  
What will be the time complexity?**

# Connected components

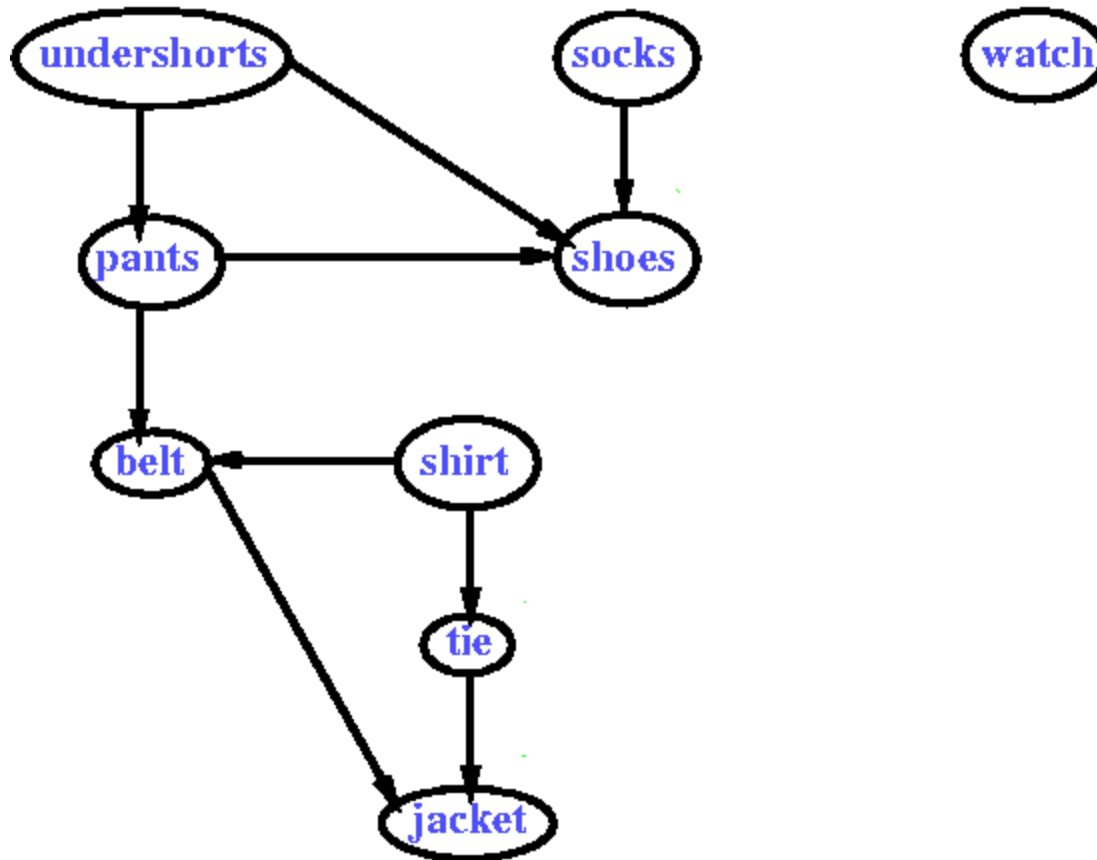
---



**How can DFS be used to check whether a graph is connected or not?**

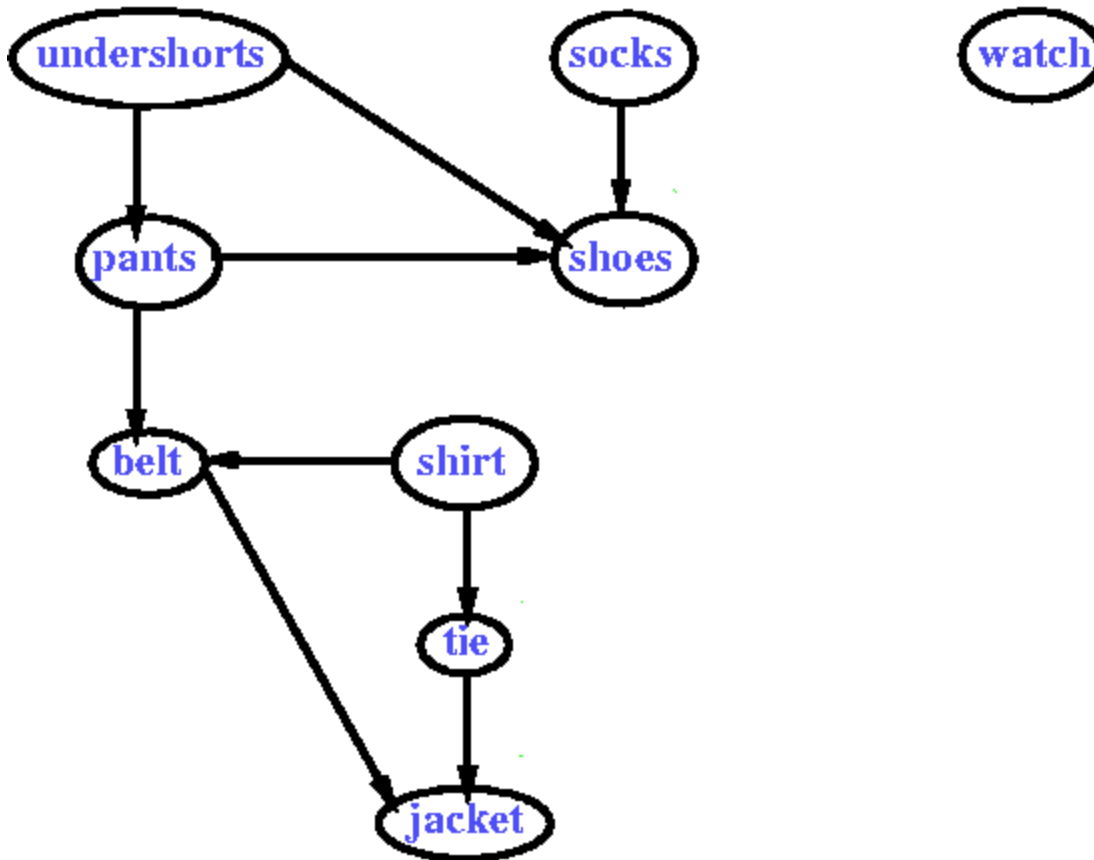
**Can you implement it????  
What will be the time complexity?**

# DFS for Topological Sort

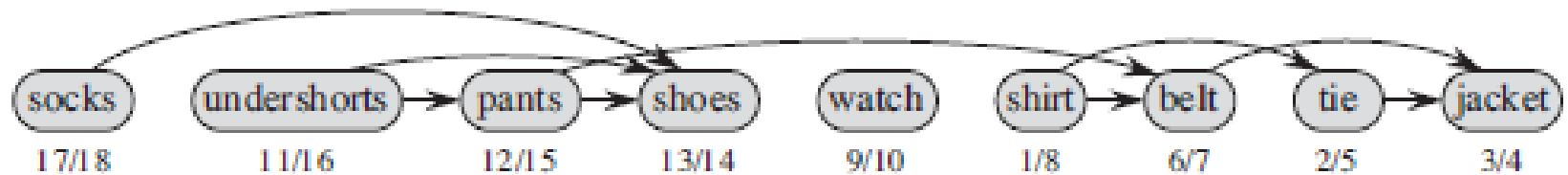




# DFS for Topological Sort



# DFS for Topological Sort-Result



# Breadth-first search

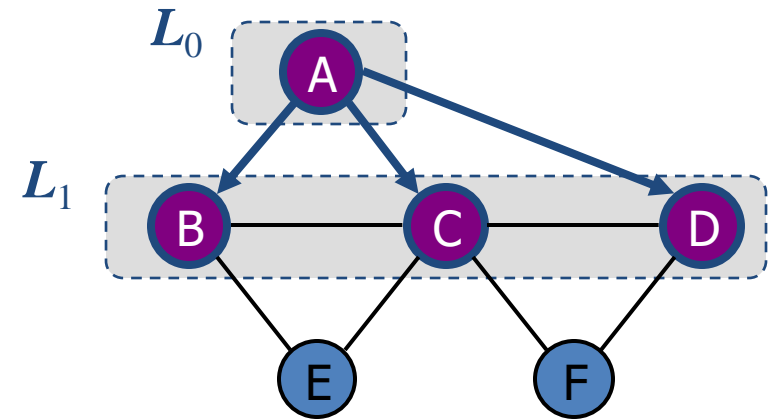
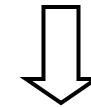
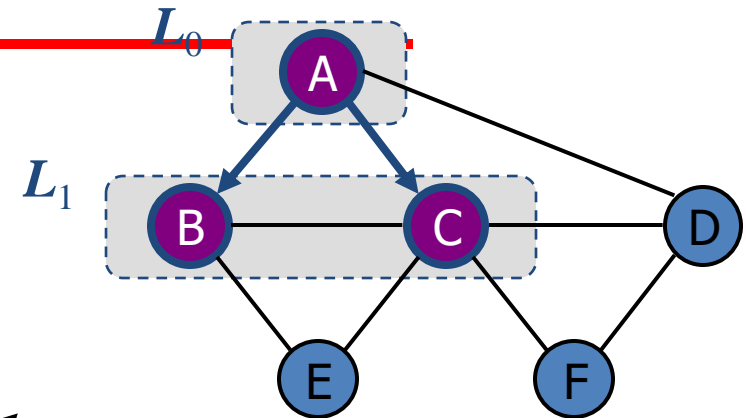
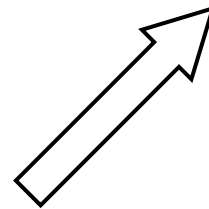
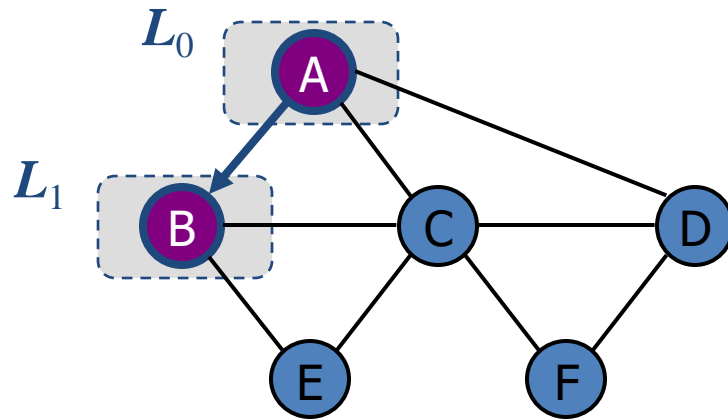
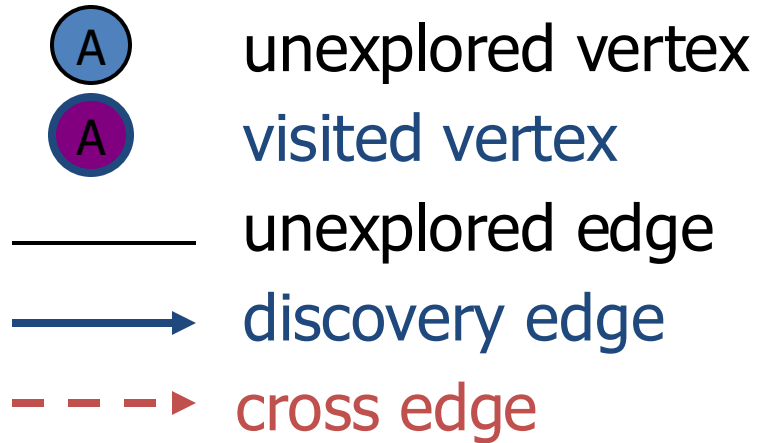


- Algorithm
- Example
- Properties
- Analysis
- Applications

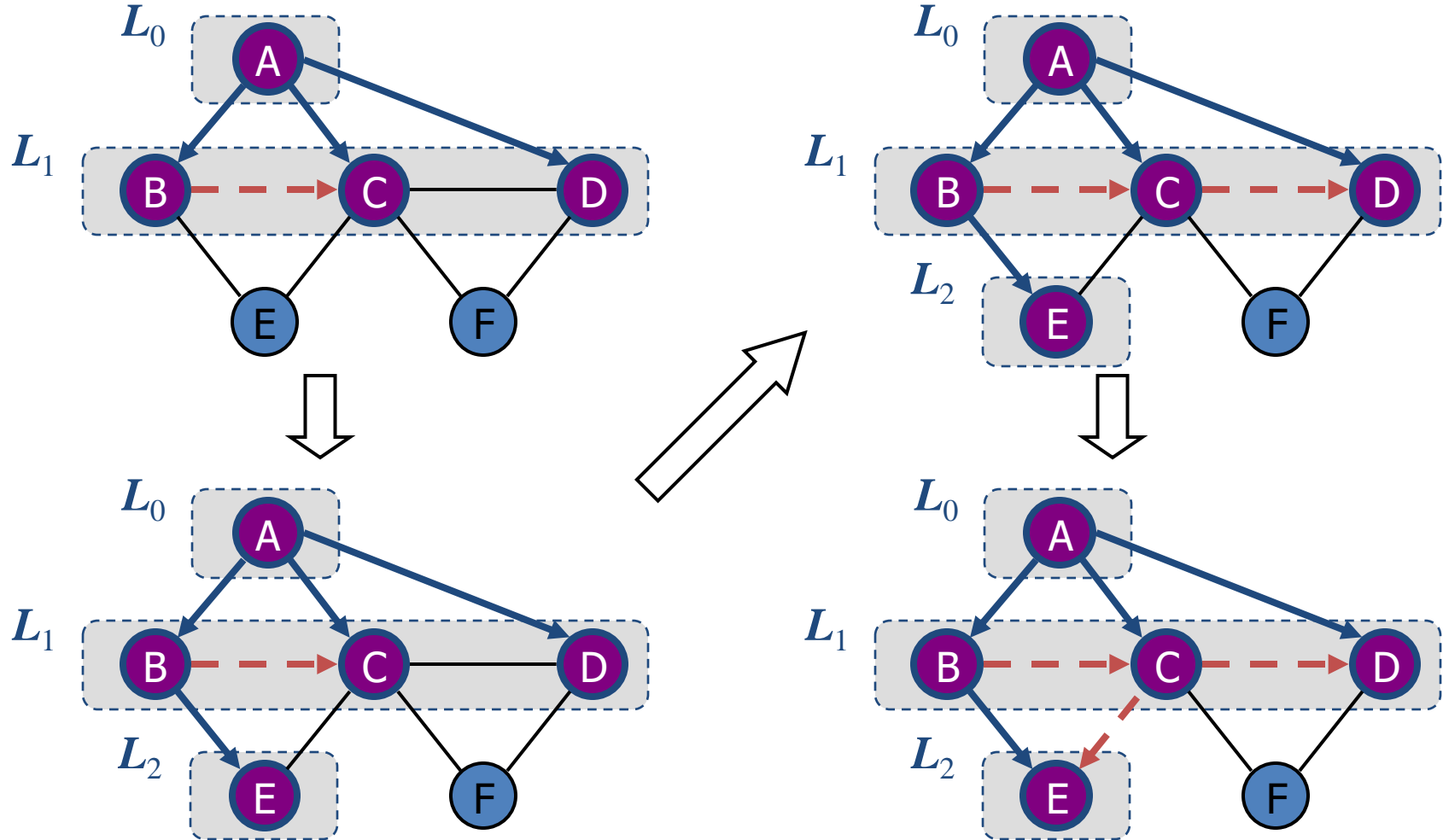
# Breadth-first search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph  $G$ 
  - discovers all vertices at distance  $k$  from  $s$  before discovering any vertices at distance  $k + 1$ .
- For any vertex  $v$  reachable from vertex  $s$ , the simple path in the breadth-first tree from  $s$  to  $v$  corresponds to a “**shortest path**” from  $s$  to  $v$  in  $G$ , that is, a path containing the smallest number of edges.

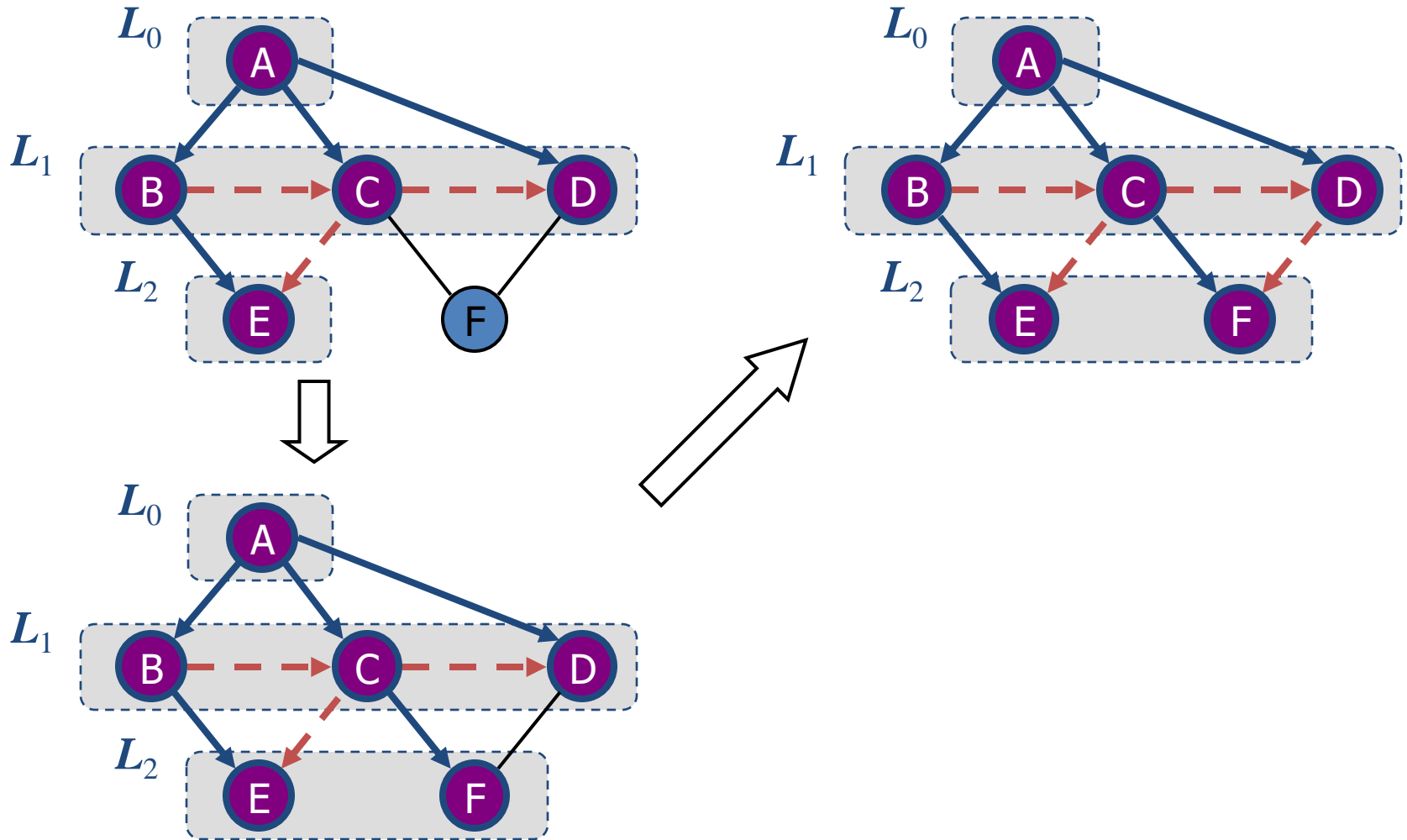
# Breadth-first search



# Breadth-first search



# Breadth-first search



# Breadth-first search



- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

**Algorithm** *BFS*(*G*)

**Input** graph *G*

**Output** labeling of the edges and partition of the vertices of *G*

for all  $u \in G.vertices()$

$setLabel(u, UNEXPLORED)$

for all  $e \in G.edges()$

$setLabel(e, UNEXPLORED)$

for all  $v \in G.vertices()$

    if  $getLabel(v) = UNEXPLORED$

$BFS(G, v)$



# Breadth-first search –

## Algorithm $BFS(G, s)$



$L_0 \leftarrow$  new empty sequence

$L_0.insertLast(s)$

$setLabel(s, VISITED)$

$i \leftarrow 0$

while  $\neg L_i.isEmpty()$

$L_{i+1} \leftarrow$  new empty sequence

    for all  $v \in L_i.elements()$

        for all  $e \in G.incidentEdges(v)$

            if  $getLabel(e) = UNEXPLORED$

$w \leftarrow opposite(v, e)$

                if  $getLabel(w) = UNEXPLORED$

$setLabel(e, DISCOVERY)$

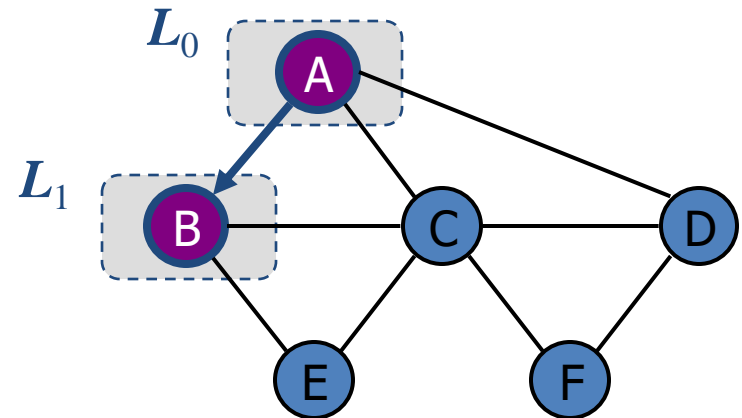
$setLabel(w, VISITED)$

$L_{i+1}.insertLast(w)$

        else

$setLabel(e, CROSS)$

$i \leftarrow i + 1$



# Breadth-first search –

## Algorithm $BFS(G, s)$



- We use auxiliary space to label edges, mark visited vertices, and store containers associated with levels.
- That is, the containers  $L_0$ ,  $L_1$ ,  $L_2$ , and so on, store the nodes that are in level 0, level 1, level 2, and so on.

## Notation

$G_s$ : connected component of  $s$

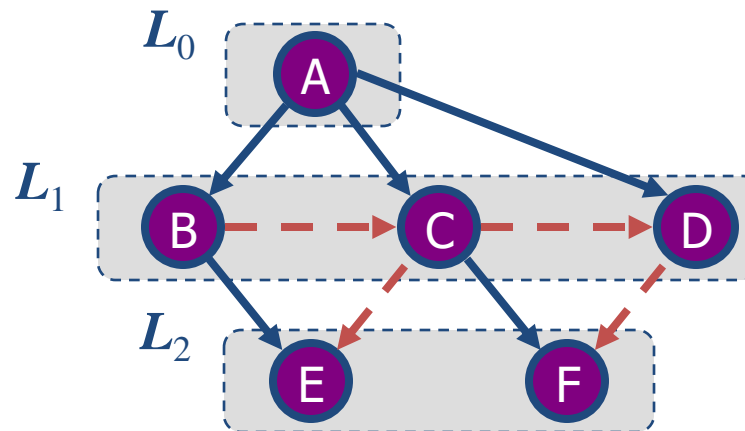
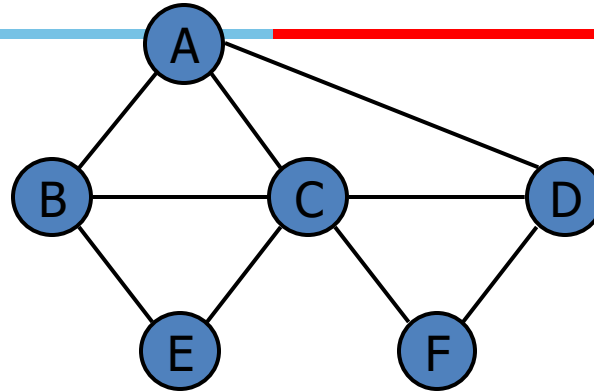
## Property 1

$BFS(G, s)$  visits all the vertices and edges of  $G_s$

## Property 2

The discovery edges of a connected component labeled by  $BFS(G, s)$  form a spanning tree  $T_s$  of  $G_s$

# Properties



- Setting/getting a vertex/edge label takes  $O(1)$  time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence  $L_i$
- Method incidentEdges is called once for each vertex
- BFS runs in  $O(n + m)$  time provided the graph is represented by the adjacency list structure
  - Recall that  $\sum_v \deg(v) = 2m$

- We can specialize the BFS traversal of a graph  $G$  to solve the following problems in  $O(n + m)$  time
  - Compute the connected components of  $G$
  - Compute a spanning forest of  $G$
  - Find a simple cycle in  $G$ , or report that  $G$  is a forest
  - Given two vertices of  $G$ , find a path in  $G$  between them with the minimum number of edges, or report that no such path exists

# Facebook as Graph



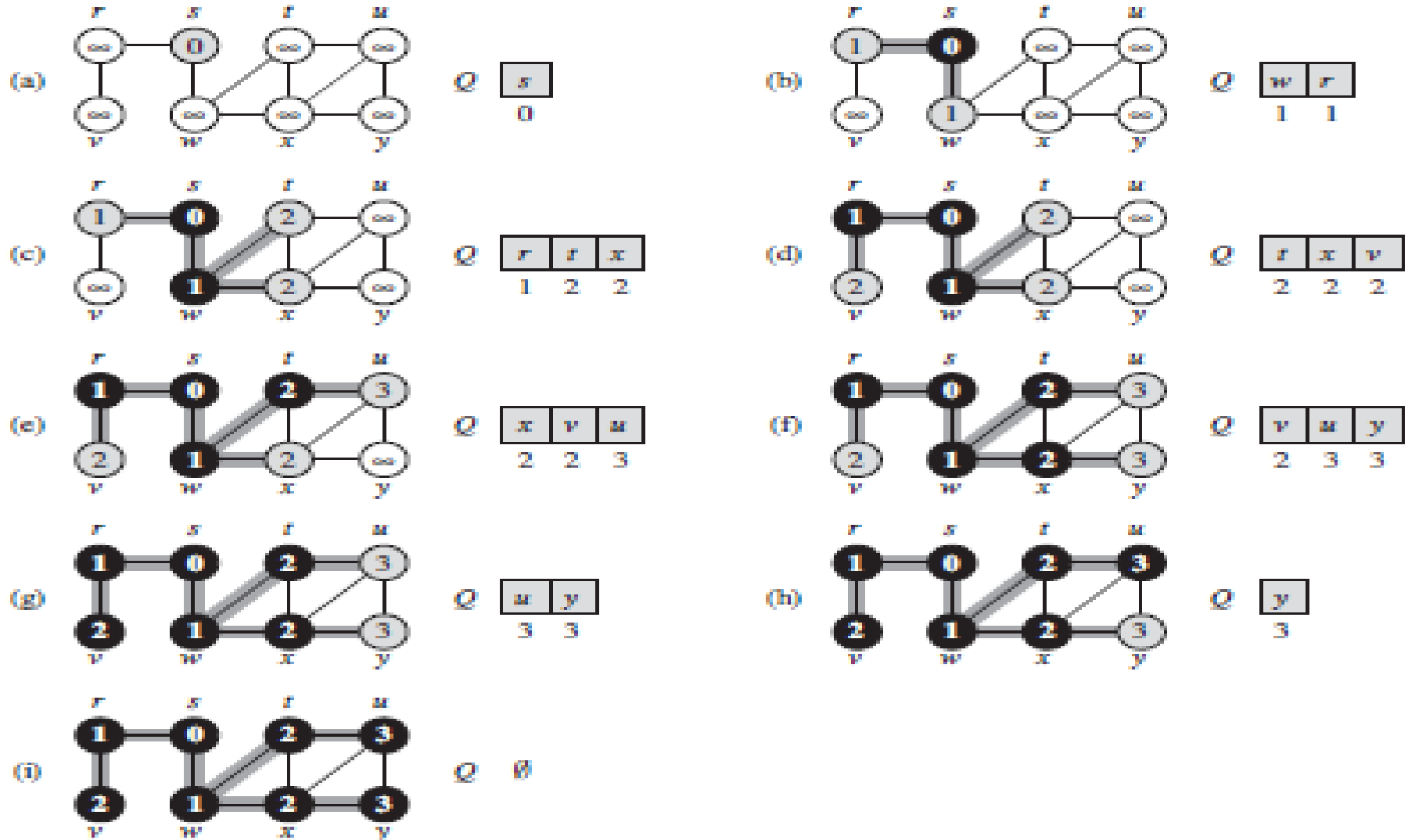
- Traversal: go to 'Friends' to display all your friends (like G.Neighbors)
- BFS: the tabs are a queue - open all friends profiles in new tabs, then close current tab and go to the next one
- DFS: the history is a stack - open the first hot friend profile in the same window; when hitting a dead end, use back button

BFS( $G, s$ )

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```



# BFS-CLRS

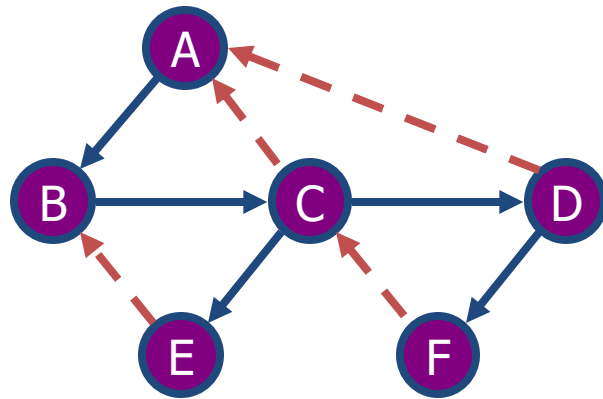


# DFS vs. BFS

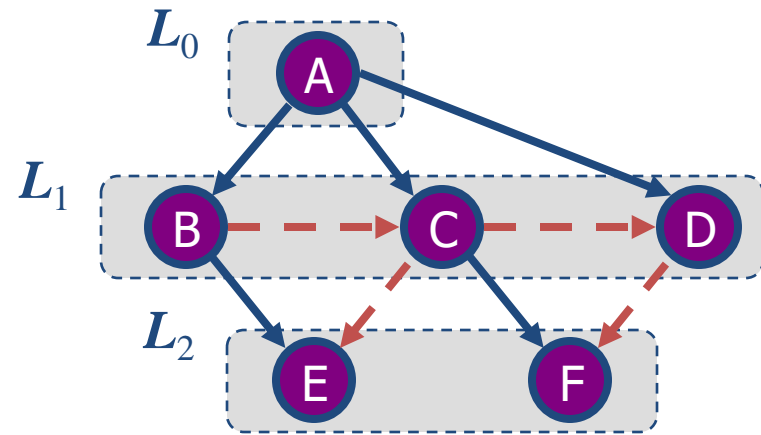


Application	DFS	BFS
Spanning forest, connected components, paths, cycles	Y	Y
Shortest Paths		Y

# DFS vs. BFS



DFS



BFS

## Back edge ( $v, w$ )

- $w$  is an ancestor of  $v$  in the tree of discovery edges

## Cross edge ( $v, w$ )

- $w$  is in the same level as  $v$  or in the next level in the tree of discovery edges



# THANK YOU!

**BITS Pilani**  
Hyderabad Campus

