Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Febin.A.Vahab

Depth-First Search

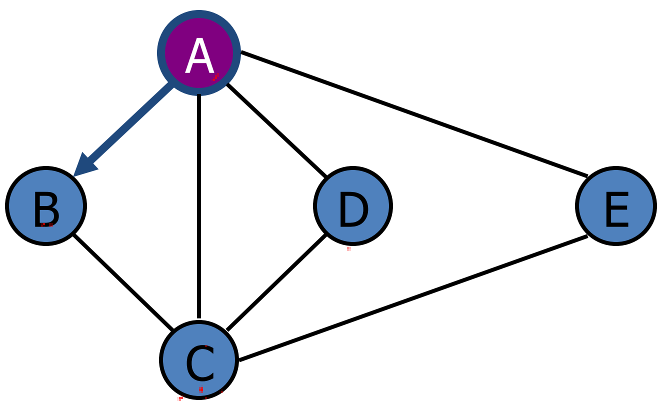
- Definitions
  - Subgraph
  - Connectivity
  - Spanning trees and forests
- Depth-first search
  - Algorithm
  - Example
  - Properties
  - Analysis
- Applications of DFS
  - Cycle finding
  - Path finding

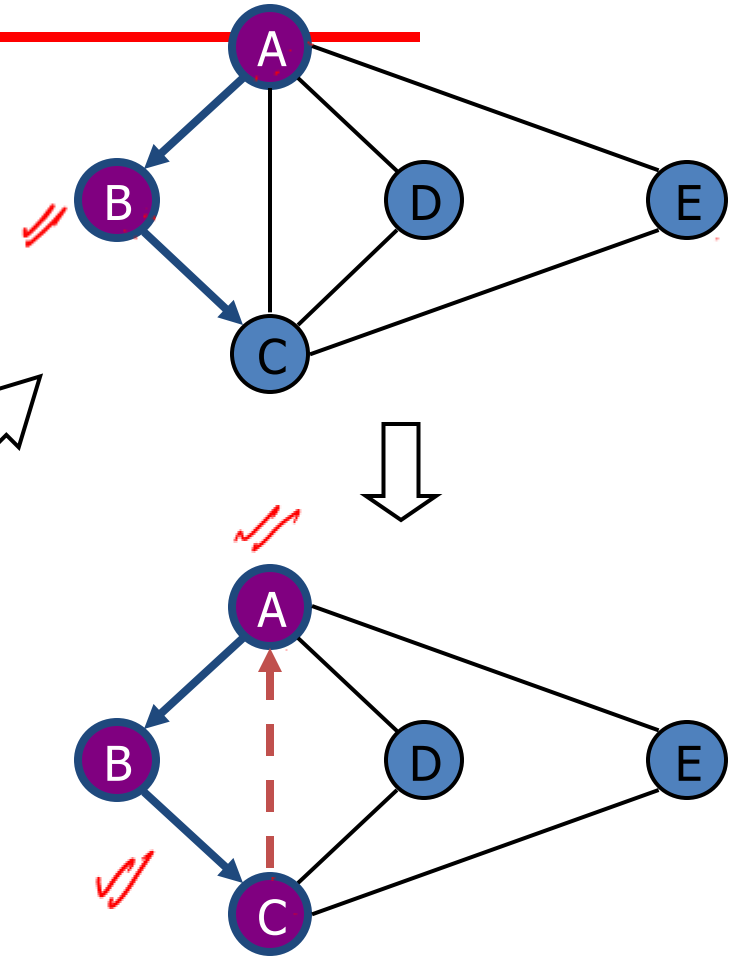
Depth-First Search

- 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

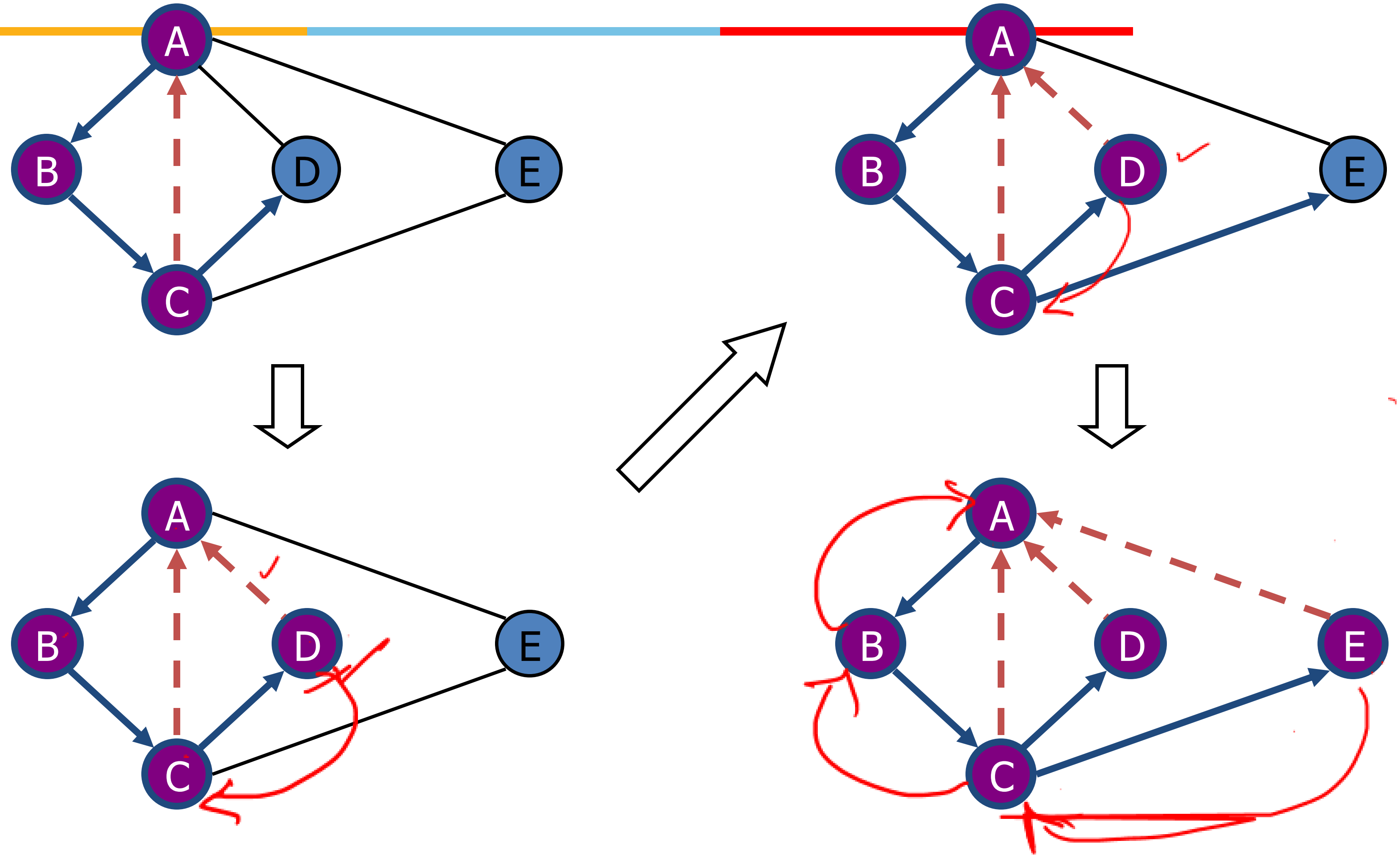
Depth-First Search

- A unexplored vertex
- A visited vertex
- unexplored edge
- discovery edge
- back edge





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)

innovate achieve lead

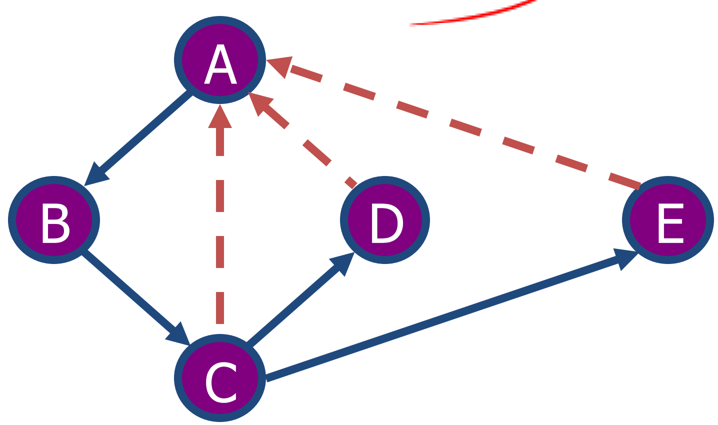
Depth-First Search-Properties

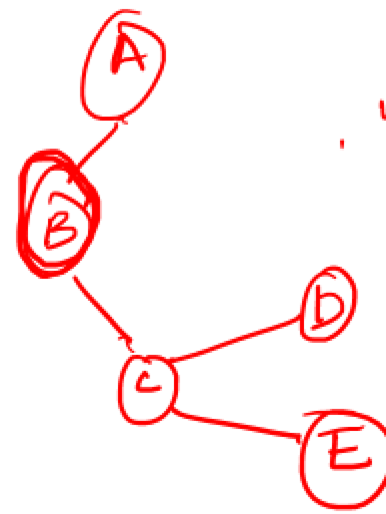
Property 1

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***)

A B C D E

unexp

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*** ∈ ***G.incidentEdges*(*v*)**

  if ***getLabel*(*e*) = *UNEXPLORED***

    w ← ***G.opposite*(*v,e*)**

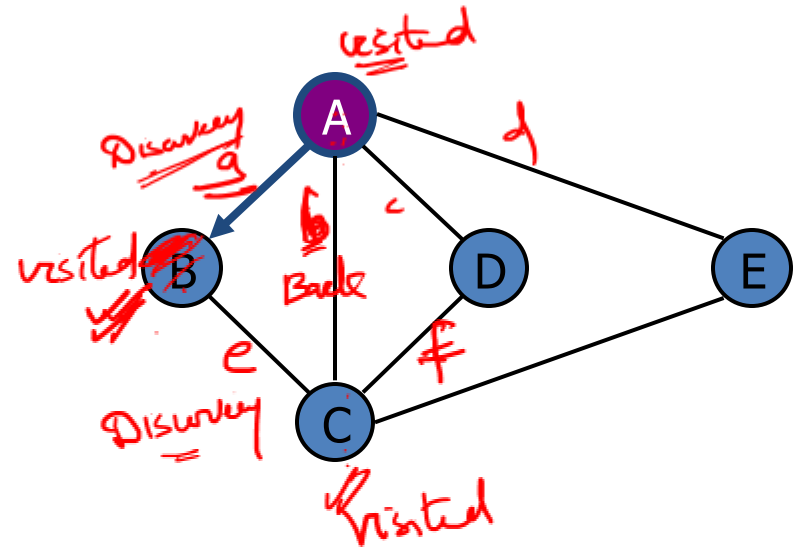
    if ***getLabel*(*w*) = *UNEXPLORED***

      ***setLabel*(*e, DISCOVERY*)**

      ***DFS*(*G, w*)**

    else

      ***setLabel*(*e, BACK*)**



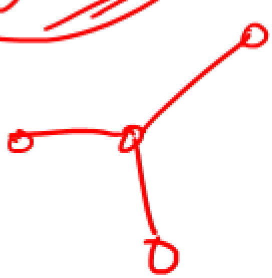
Analysis of DFS

- Setting/getting a vertex/edge label takes $\boldsymbol{O}(1)$ time $O(n)$
- Each vertex is labeled twice $O(m)$
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK $O(\cancel{m})$ $O(n)$
- Method incidentEdges is called once for each vertex
- DFS runs in $\boxed{\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{m})}$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(\boldsymbol{v}) = 2\boldsymbol{m}$ $O(E)$

Depth-First Search

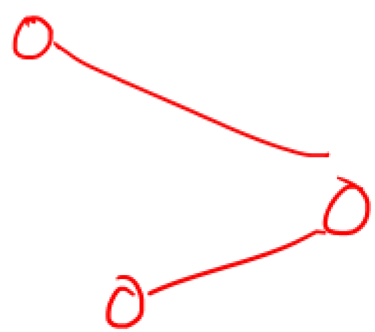
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning tree of G
  - Computing a cycle in G, or reporting that G has no cycles
  - Find and report a path between two given vertices

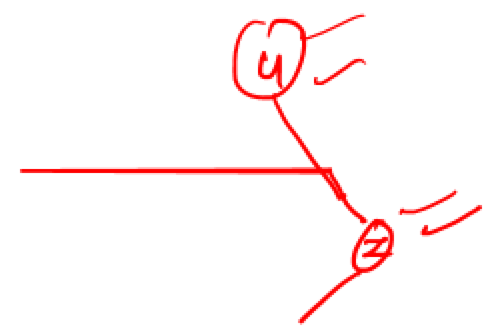
clusters



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



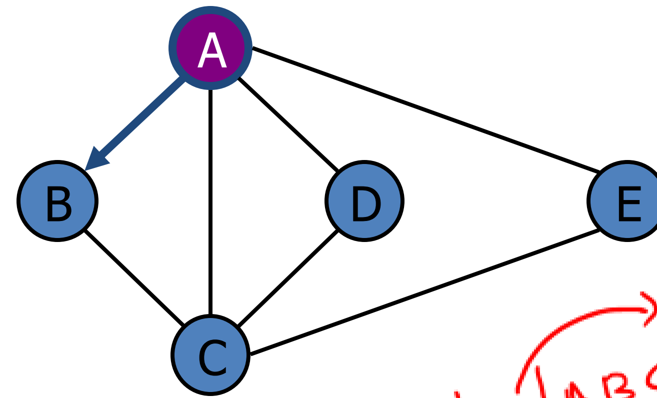


innovate achieve lead

Path Finding

```
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 gets popped }
            else
                    setLabel(e, BACK)
     S.pop()                                   { v gets popped }
```

A   D



D | ABCD

C
B
A

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()
```

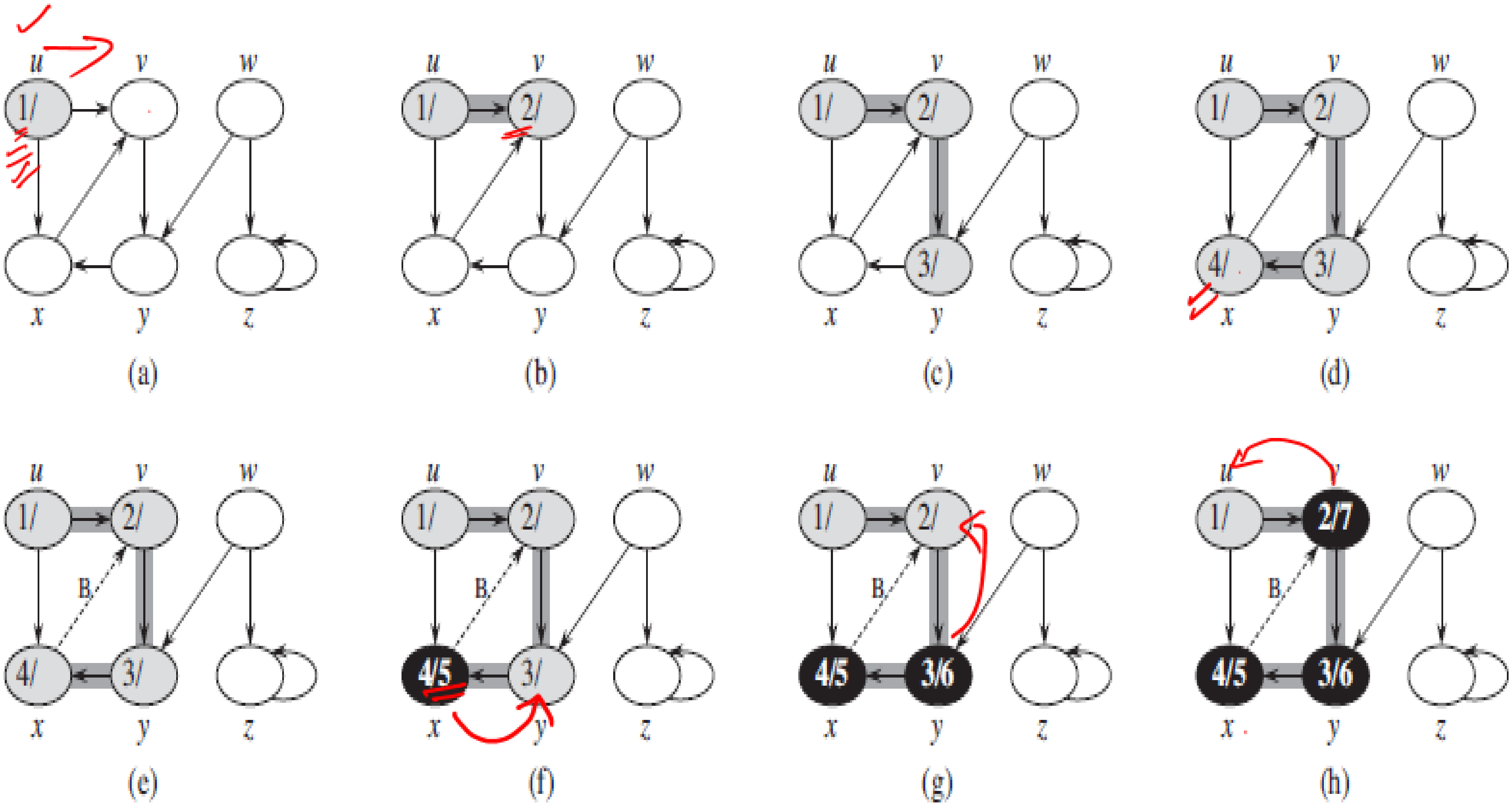
DFS:R2-Chapter 22

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1              // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]        // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK              // blacken u; it is finished
time = time + 1
u.f = time
```

DFS:R2-Chapter 22

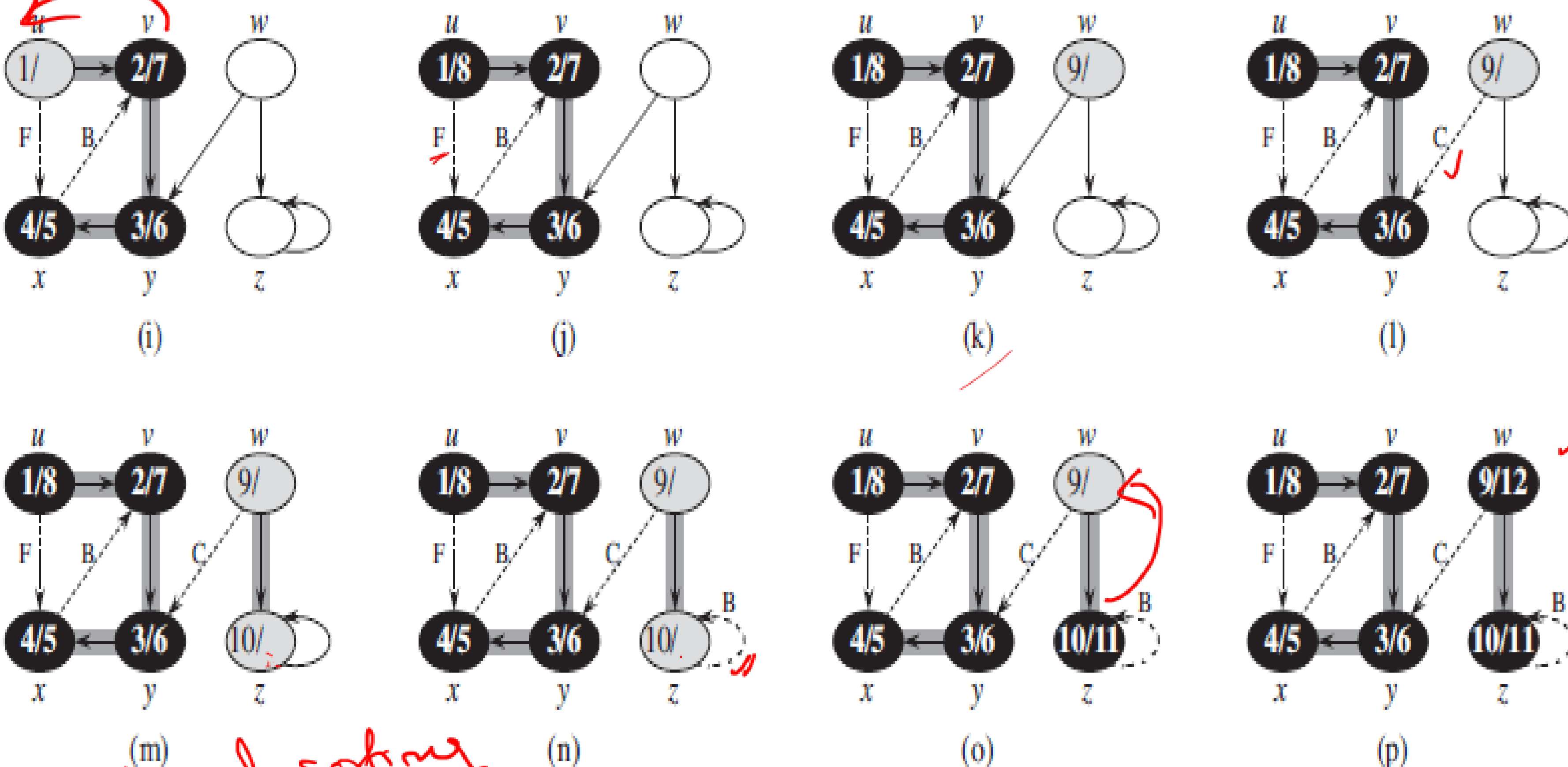
x/y



DFS:R2-Chapter 22

x/y

(u,v)



Topological sorting

w z u v y x

Connected components

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

Can you implement it????

What will be the time complexity?

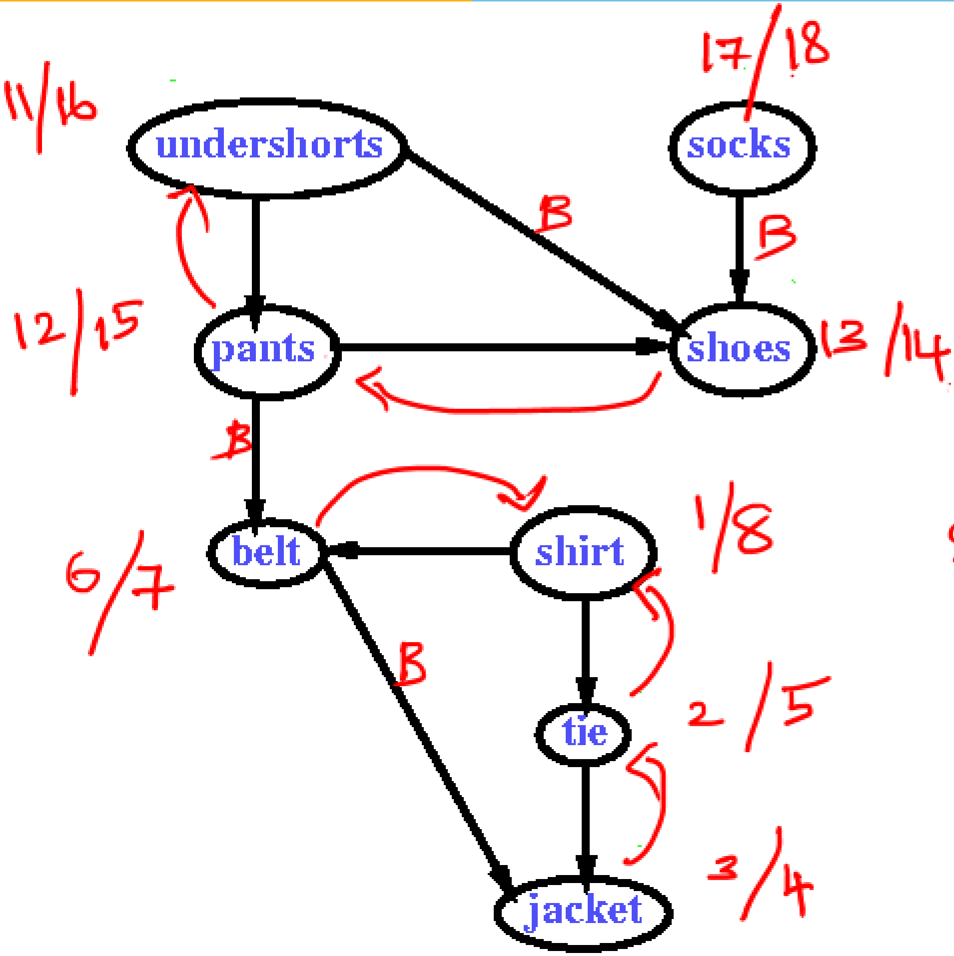
Connected components

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 Toplogical Sort



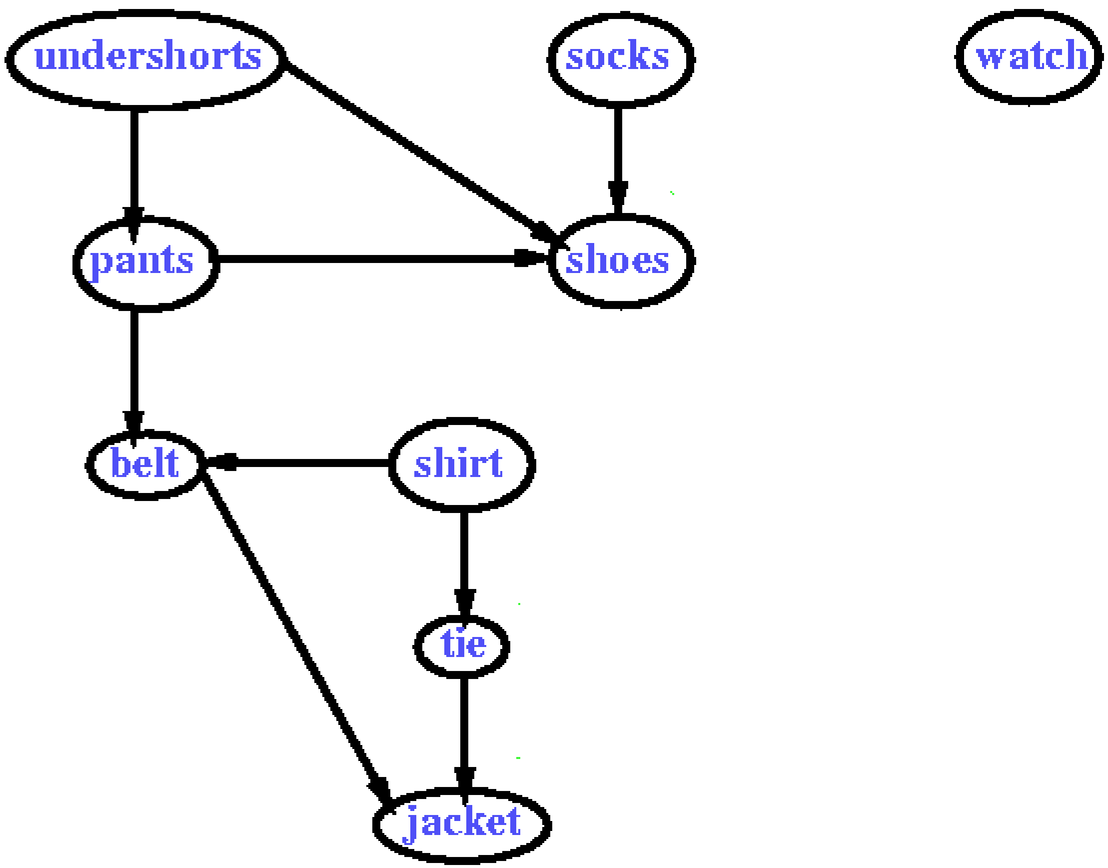
decreasing order of finishing times

socks undershorts → pants → shoes

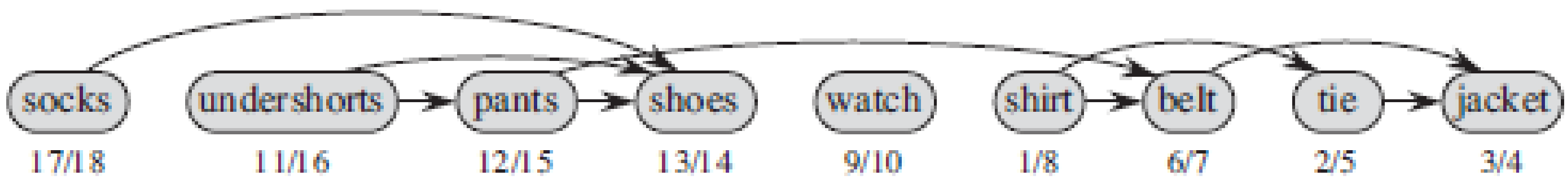
watch shirt → belt

tie → jacket

DFS for Toplogical Sort



DFS for Toplogical Sort-Result

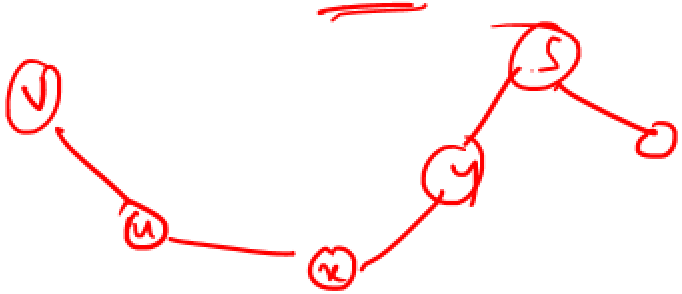


Breadth-first search

- Algorithm
- Example
- Properties
- Analysis
- Applications

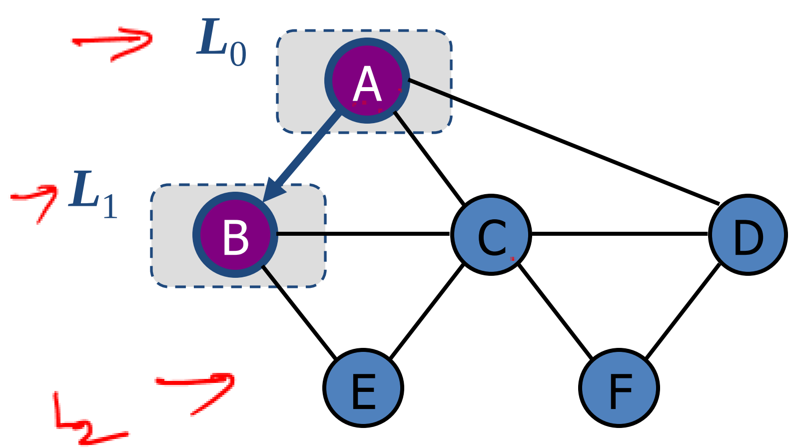
Breadth-first search

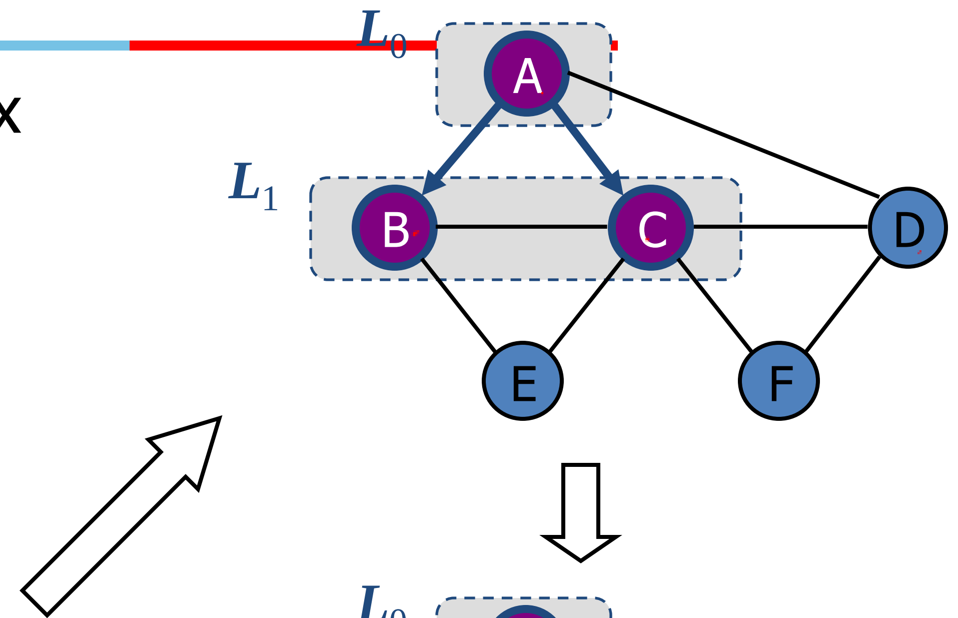
- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - discovers all vertices at distance k from s before discovering any vertices at distance k + 1.
- For any vertex v reachable from vertex s, the simple path in the breadth-first tree from s to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

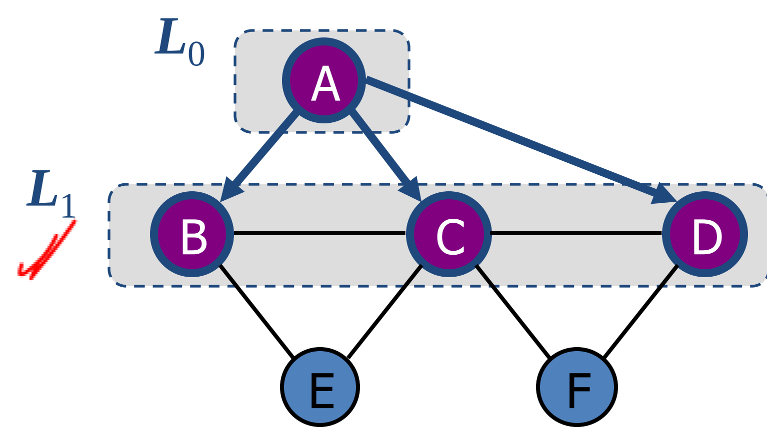


Breadth-first search

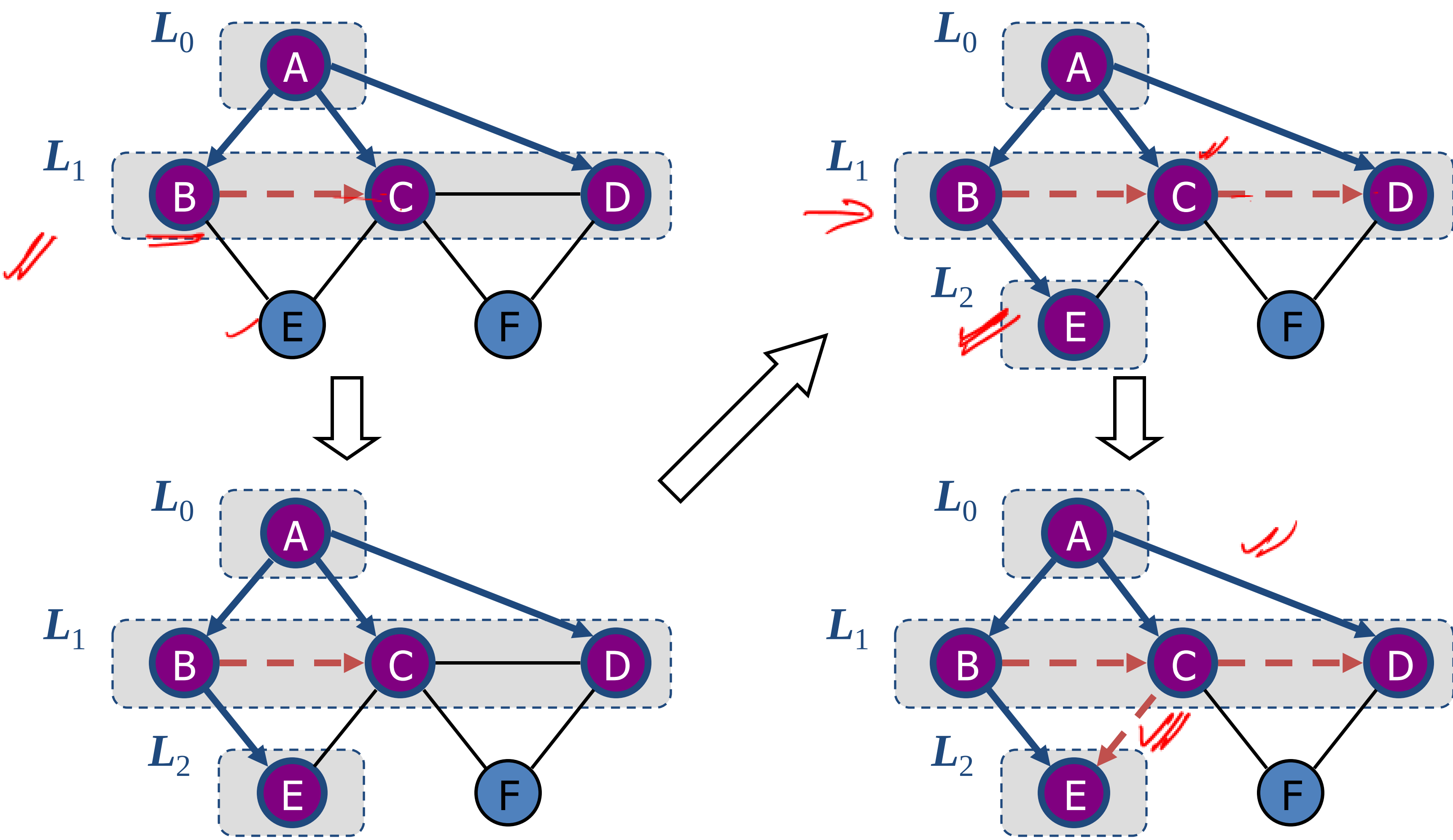
- unexplored vertex
- visited vertex
- unexplored edge
- discovery edge
- cross edge



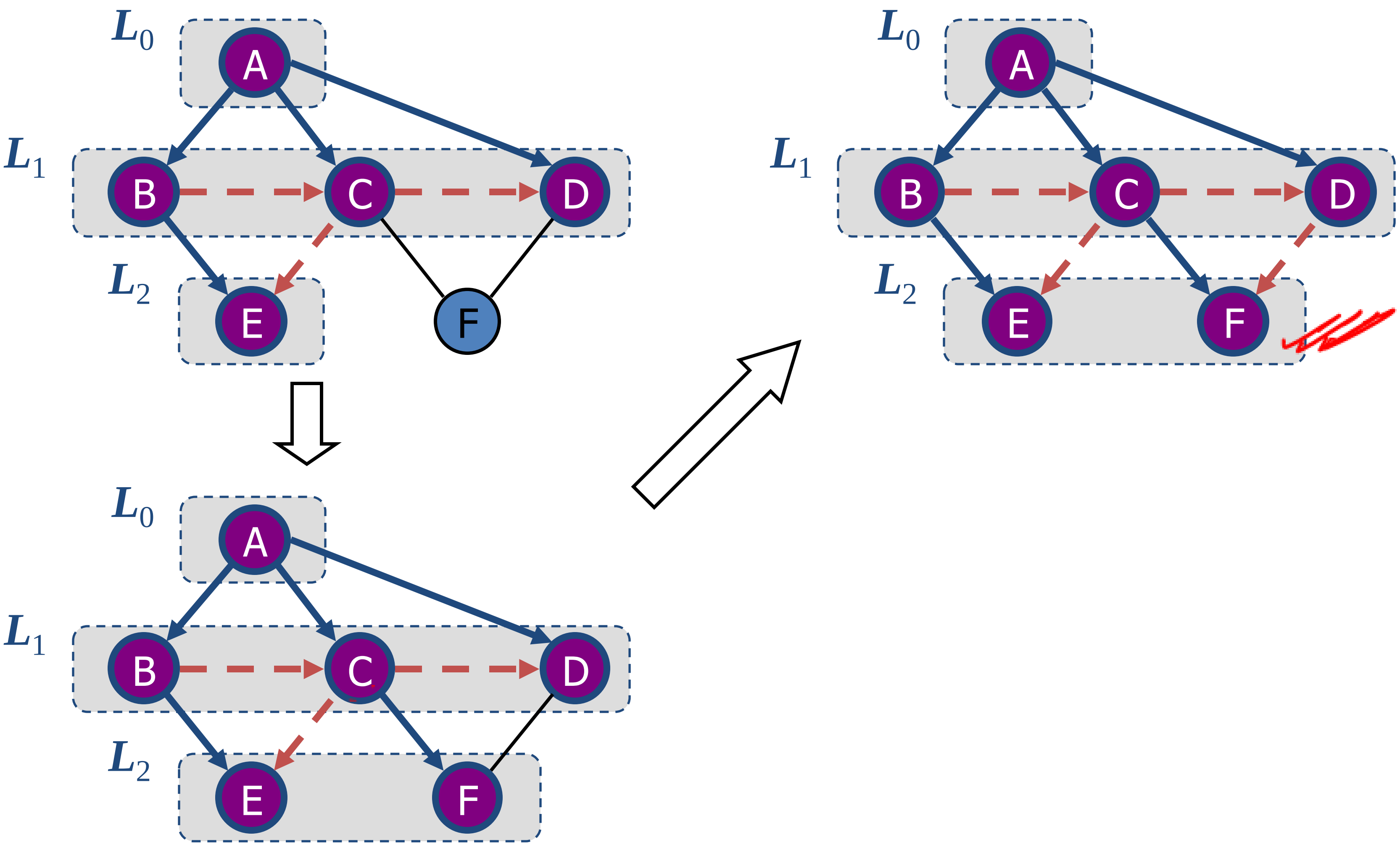




Breadth-first search



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***

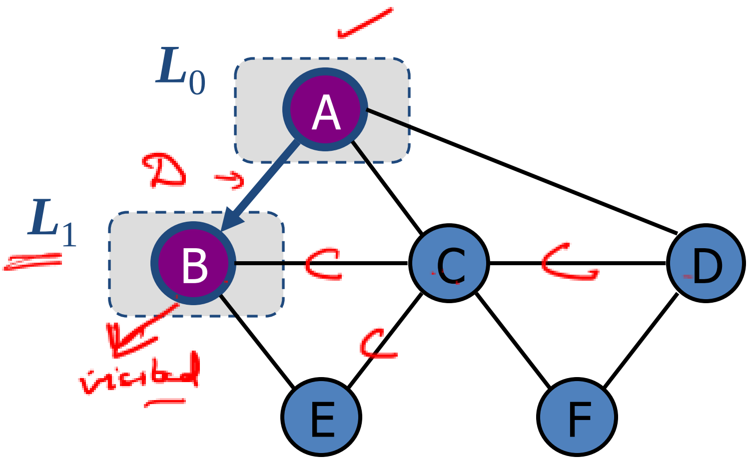
BFS(***G***, ***v***)

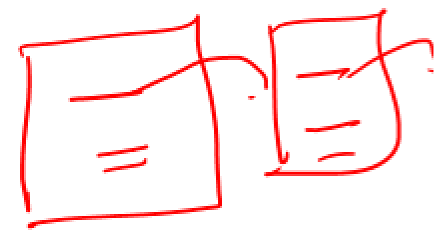
BFS (G, A)

Breadth-first search – **Algorithm *BFS*(*G*, *s*)**

A

```
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
```





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 L0, L 1 , L2, and so on, store the nodes that are in level 0, level 1 , level 2, and so on.

Properties

Notation

$\boldsymbol{G_s}$: connected component of $\boldsymbol{s}$

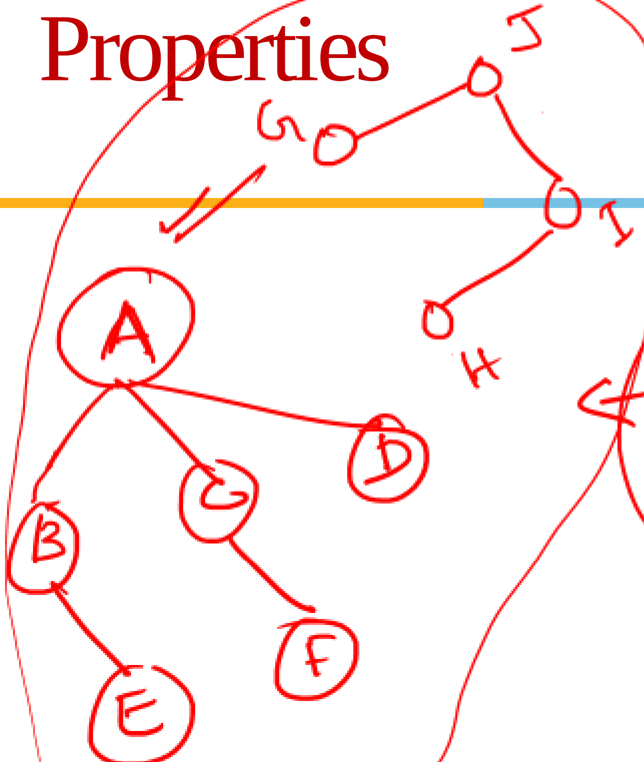
Property 1

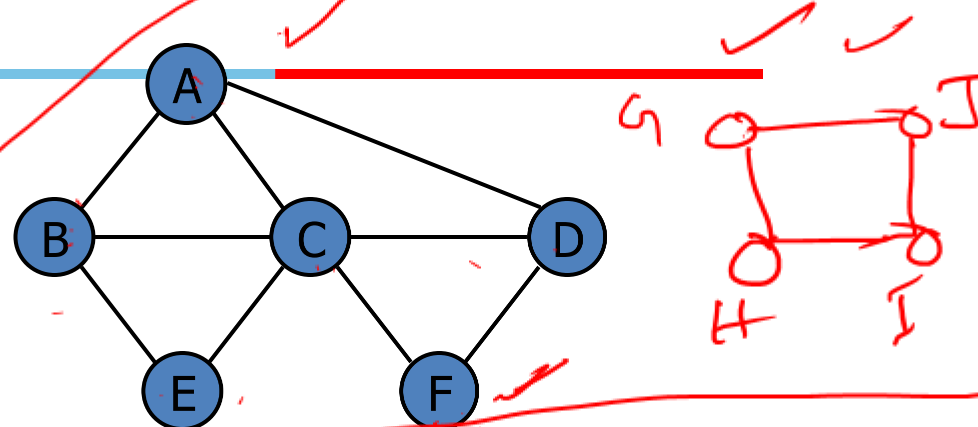
$\boldsymbol{BFS}(\boldsymbol{G}, \boldsymbol{s})$ visits all the vertices and edges of $\boldsymbol{G_s}$

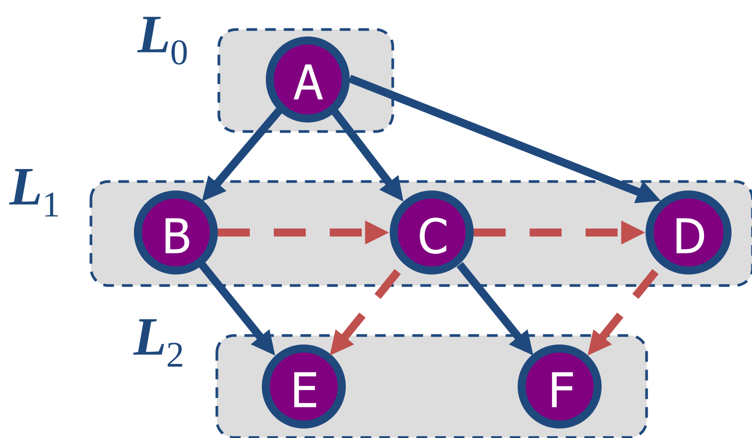
Property 2

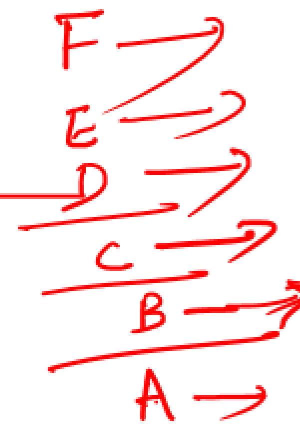
The discovery edges of a connected component labeled by $\boldsymbol{BFS}(\boldsymbol{G}, \boldsymbol{s})$ form a spanning tree $\boldsymbol{T_s}$ of $\boldsymbol{G_s}$

Properties







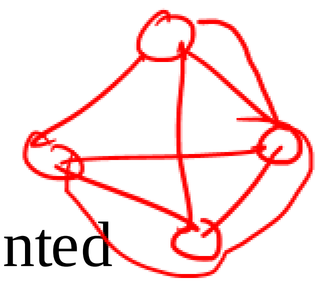


Analysis

- 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 $\boxed{\Sigma_v \deg(v) = 2m}$

$O(n)$

$O(m)$



Applications

- 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

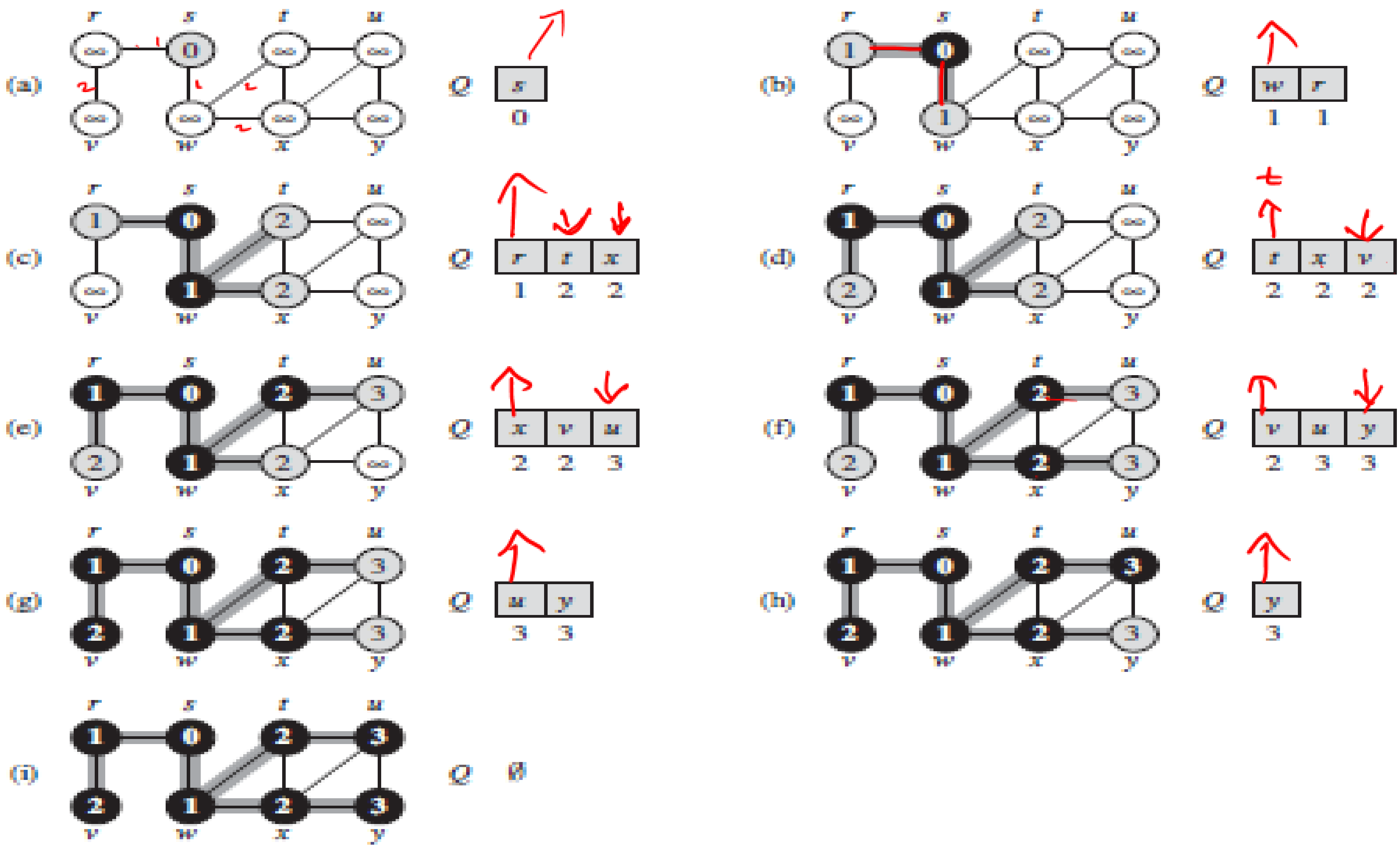
BFS-CLRS

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

D
C
B
A

~~A~~

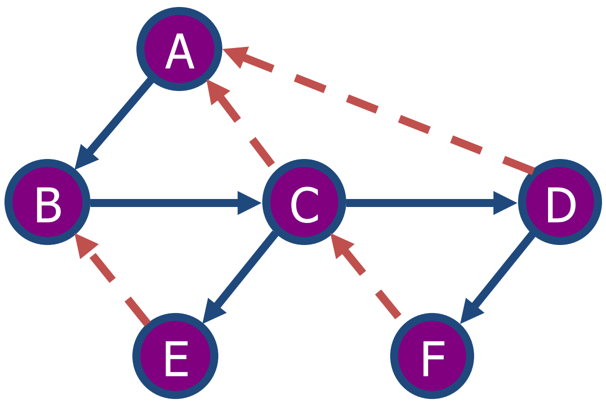
BFS-CLRS

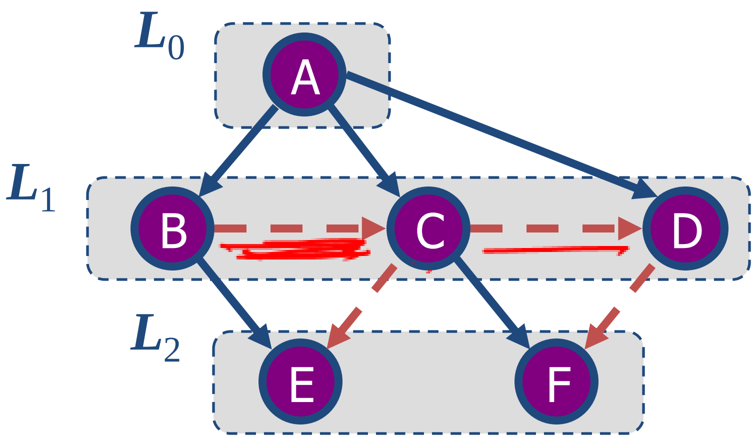


DFS vs. BFS

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

DFS vs. BFS





DFS vs. BFS

innovate achieve lead

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