COMP S265F Unit 4: Graph Algorithms: BFS, DFS, Topological Sort

Dr. Keith Lee

School of Science and Technology

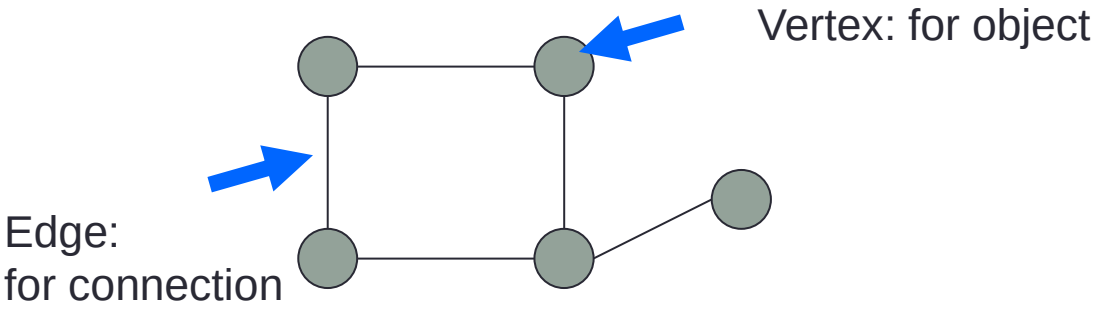
The Open University of Hong Kong

Overview

- **Graph representation:** Adjacency Matrix, Adjacency List
- **Breadth-First Search (BFS)**
  - Breadth-First Tree
  - BFS algorithm
  - Time complexity on adjacency list & adjacency matrix
- **Depth-First Search (DFS)**
  - Directed Graphs
  - DFS algorithm
  - Depth-First Tree
  - Timestamps, Parenthesis Theorem, White-Path Theorem
- **Topological Sort**
  - Directed Acyclic Graphs (DAG)
  - Algorithm (Simple application of DFS)
  - Proof of Correctness

Graph algorithms

- **Graphs**: A set **V** of OBJECTS (**vertices**; singular: vertex) with a set **E** of pairwise CONNECTIONS (**edges**).
- Design and analysis of graph algorithms is a challenging branch of computer science.
- There are hundreds of graph algorithms known, and thousands of practical applications.



- **Note:** We have $0 \le |E| \le |V|(|V|-1)/2$ because there are $C_2^{|V|} = \frac{|V|(|V|-1)}{2}$ possible pairs of vertices.

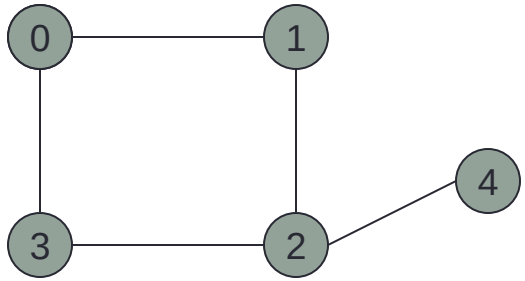
Graph representation: Adjacency Matrix

- **Mathematical representation**

  G=(V, E) where

  V={0, 1, 2, 3, 4} and

  E={(0,1), (1,2), (2,3), (3,0),(2,4)}



- **Adjacency matrix**

	0	1	2	3	4
0	0	1	0	1	0
1	1	0	1	0	0
2	0	1	0	1	1
3	1	0	1	0	0
4	0	0	1	0	0

For any pair of vertices **i, j**,
a[i,j] = 1 if there is an edge (i, j);
otherwise **a[i,j] = 0**.
Note that (i, j) is an edge
⇔ (j, i) is an edge, so a[i,j]=a[j,i].

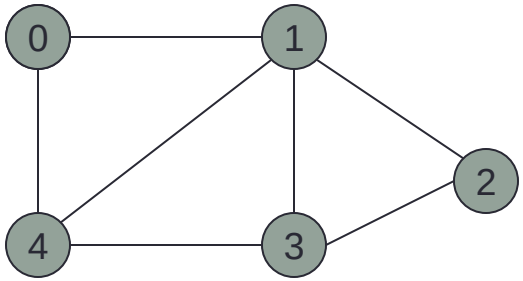
Adjacency Matrix: Python code

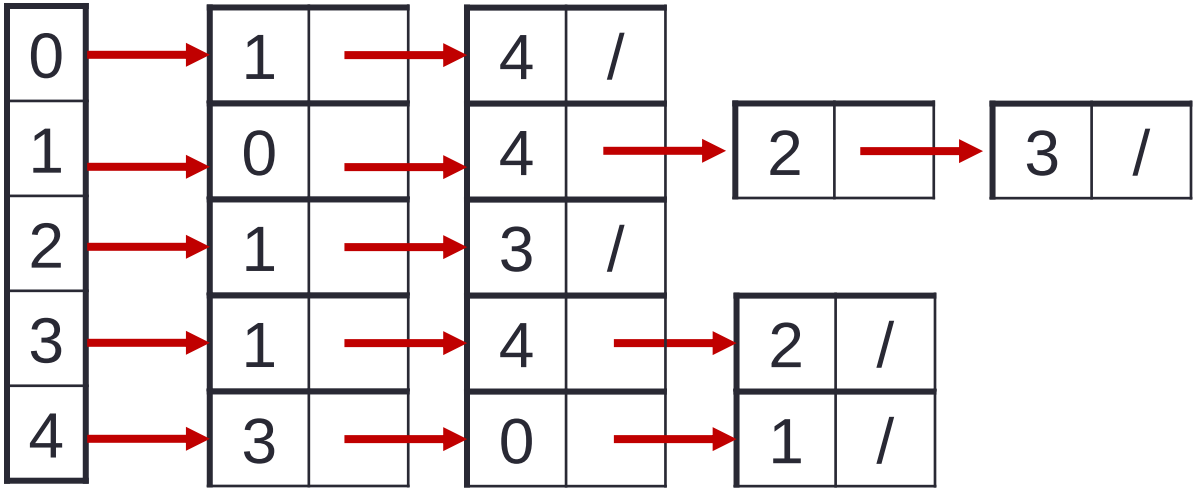
```
class GraphAM:

    # Constructor
    def __init__(self, numNodes):
        self.graph = [] # 2D list
        for i in range(numNodes):
            self.graph.append([0 for j in range(numNodes)])
        self.numNodes = numNodes

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u][v] = 1
        self.graph[v][u] = 1
```

- O(**|V|²**) space, even if |*E*| *is very small*.
- **O(1) time** to decide if an edge (i, j) exists.

Adjacency List

- Every vertex u is associated with a linked list **Adj[u]**, which contains all the vertices adjacent to u.





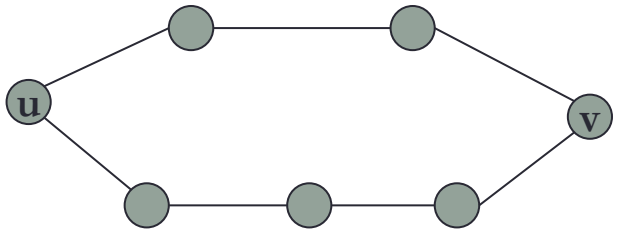
Adjacency List: Python code

```
from collections import defaultdict

class GraphAL:
    # Constructor
    def __init__(self, numNodes):
        # default dictionary to store graph
        self.graph = defaultdict(list)
        self.numNodes = numNodes

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
        self.graph[v].append(u)
```

- O(2**|E|**) = O(**|E|**) space
- **O(|V|) time** to scan Adj[u] decide if an edge (u, v) exists.
- **Adjacency matrix** is good for **dense** graph (with many edges); while **adjacency list** is good for **sparse** graph.

Breadth-First Search (BFS)

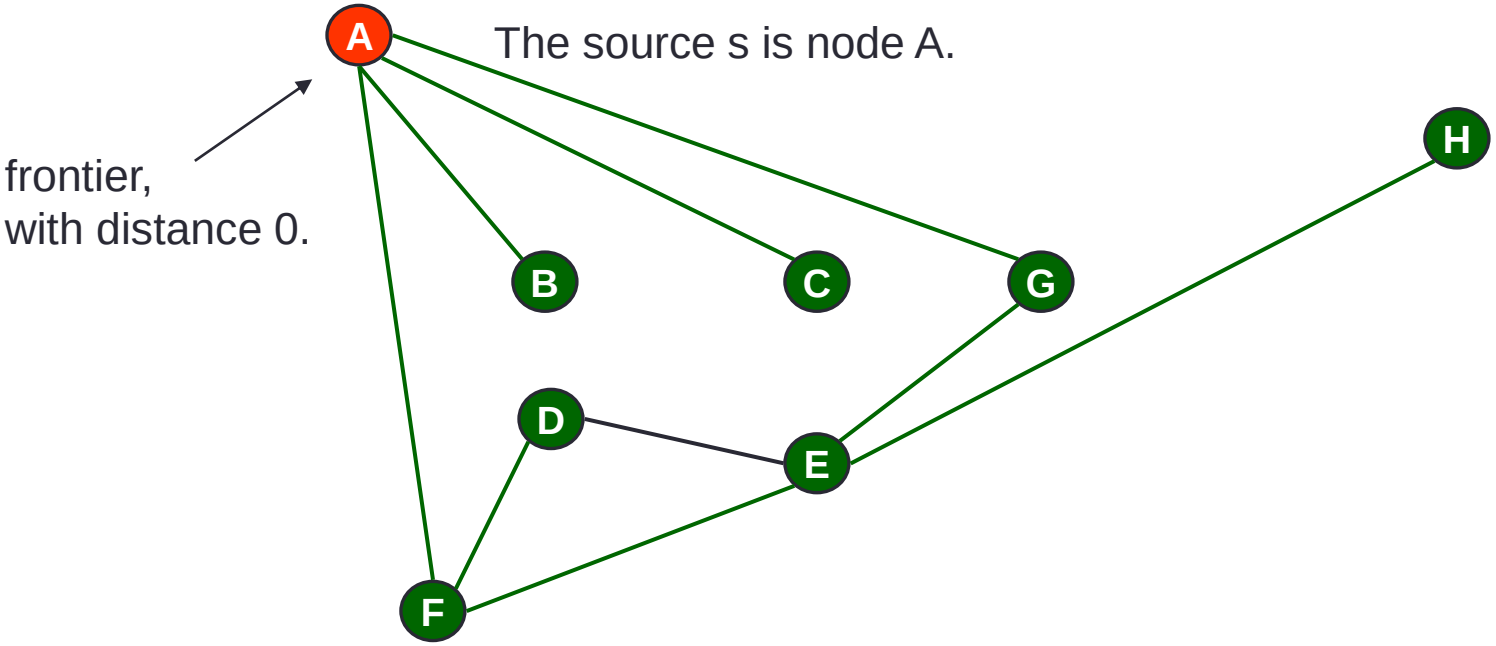
- Breadth-first search (BFS) is a simple algorithm for searching a graph.
- Given G=(V, E), and a distinguished source vertex **s**, BFS systematically explores the edges of G to
  - discover every vertex that is reachable from **s**,
  - compute distance (i.e., smallest number of edges) from **s** to each reachable vertex, and
  - produce a "breadth-first" tree with root **s** that contains all reachable vertices.



Vertex v is reachable from u because there is a sequence of consecutive edges from u to v.
The distance from u to v is 3.

BFS: Example (Step 1)

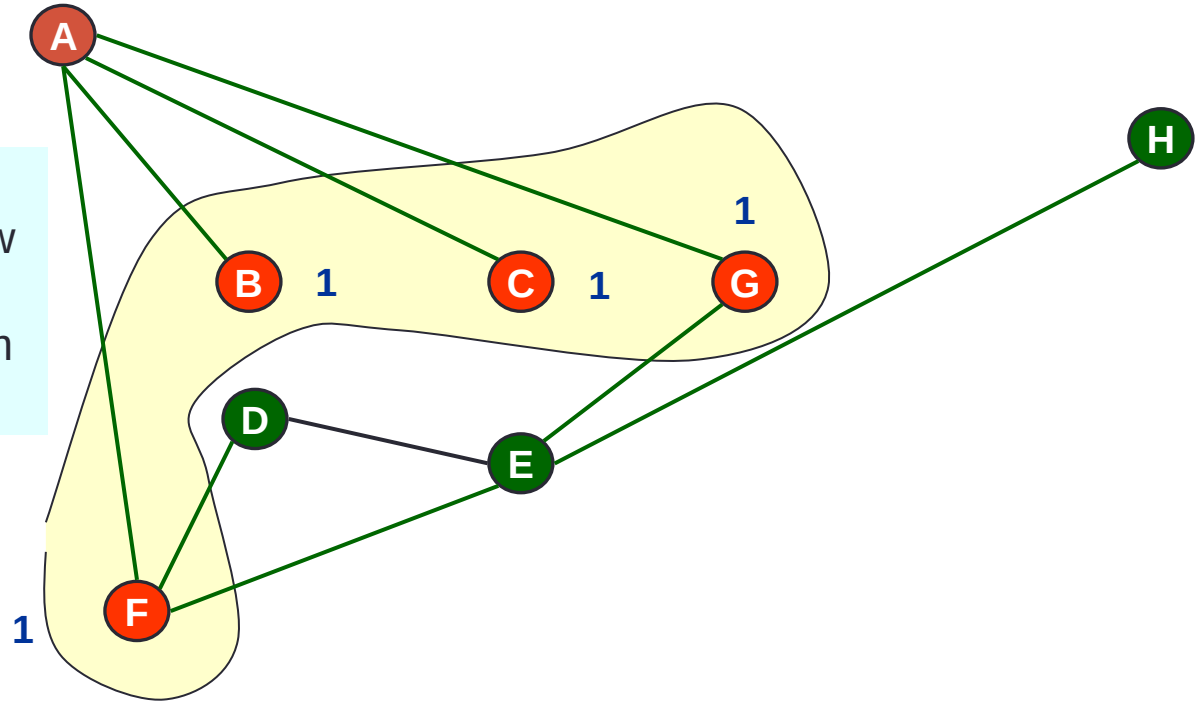
- Breadth-first means to expand the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance **k** from **s** before discovering any vertices at distance **k+1**.



BFS: Example (Step 2)

- Breadth-first means to expand the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance **k** from **s** before discovering any vertices at distance **k+1**.

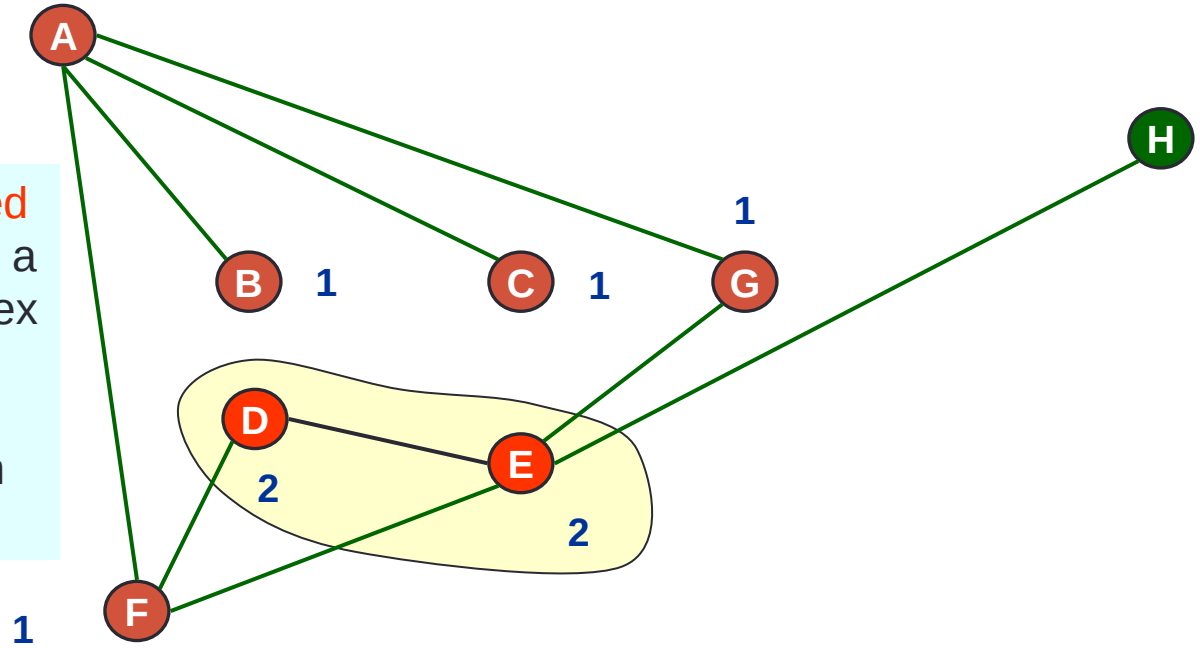
All nodes next to A form a new frontier with distance 1 from A.



BFS: Example (Step 3)

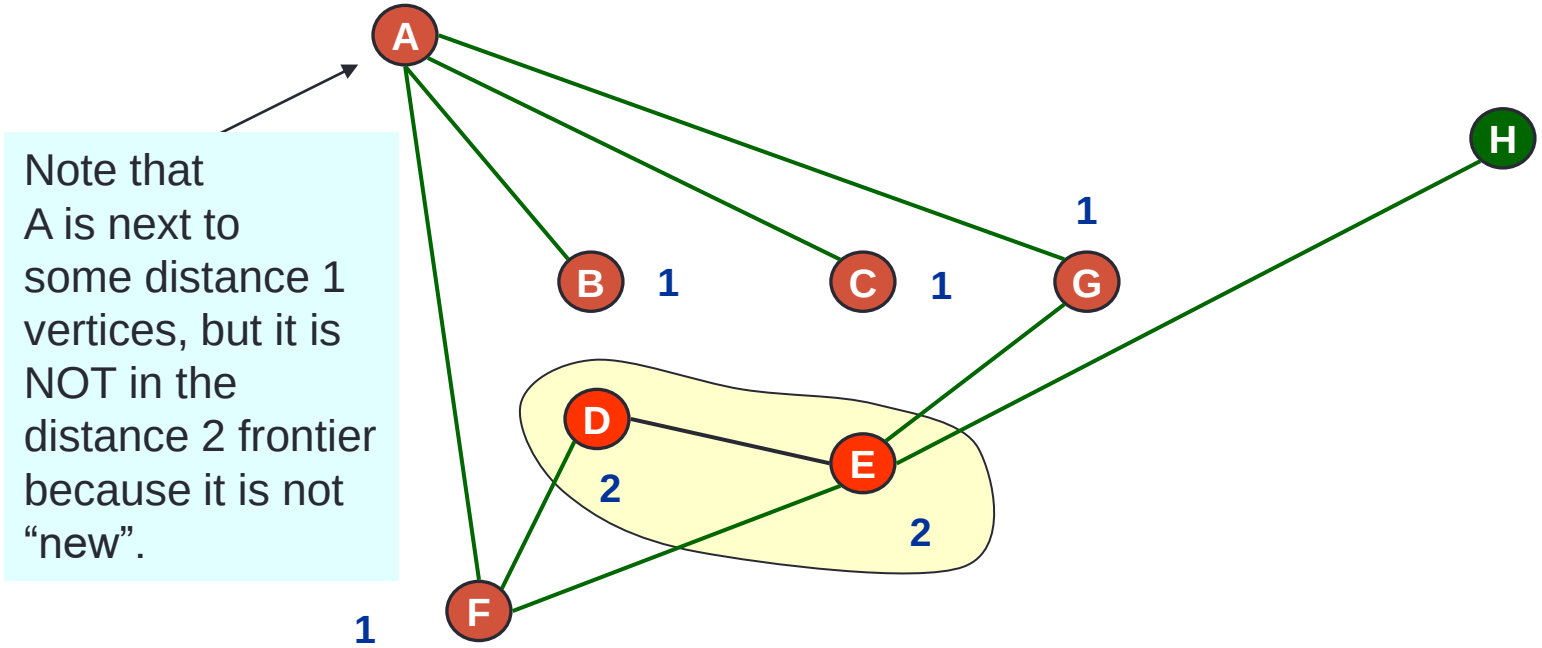
- Breadth-first means to expand the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance **k** from **s** before discovering any vertices at distance **k+1**.

All undiscovered vertices next to a distance 1 vertex form a new frontier with distance 2 from A.



BFS: Example (Step 3, cont')

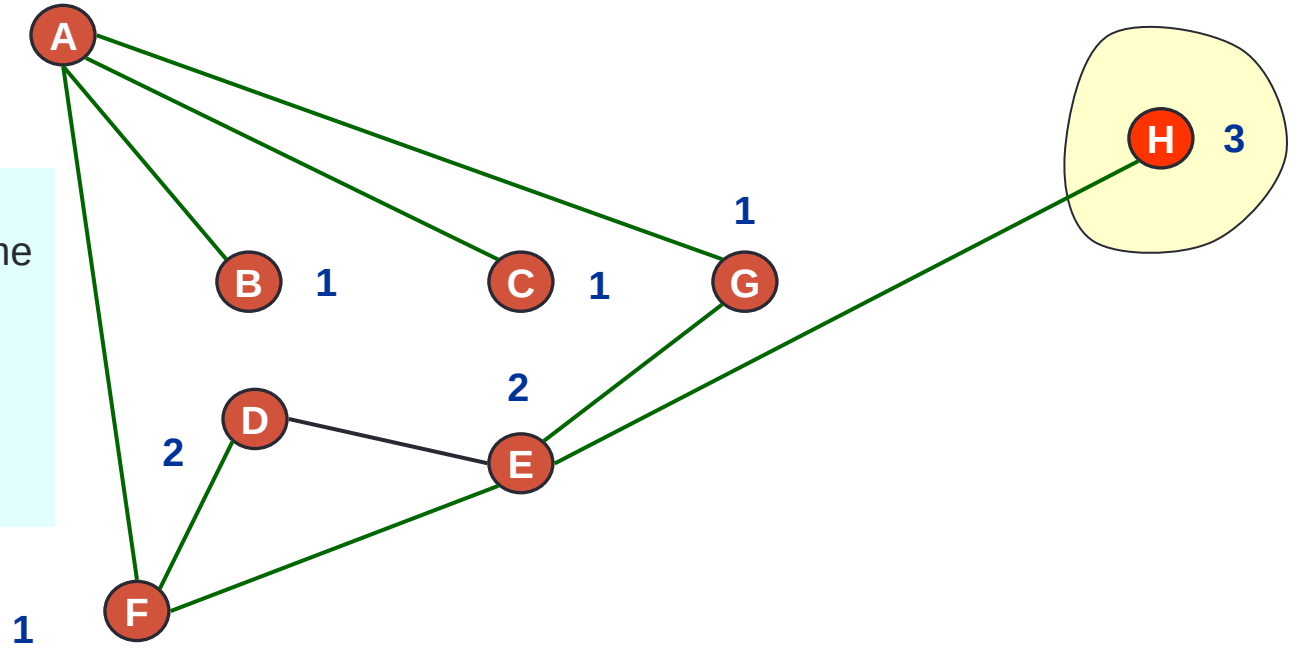
- Breadth-first means to expand the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance **k** from **s** before discovering any vertices at distance **k+1**.



BFS: Example (Step 4)

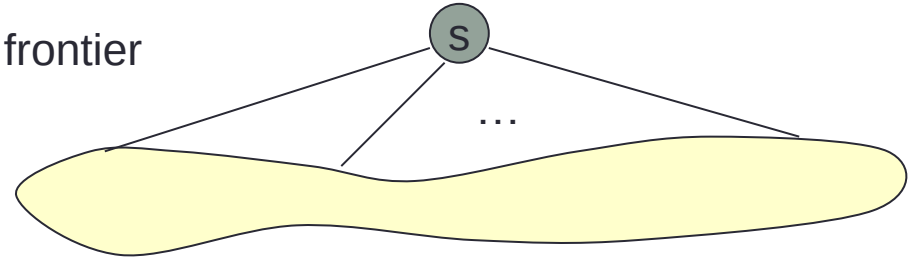
- Breadth-first means to expand the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance **k** from **s** before discovering any vertices at distance **k+1**.

All undiscovered nodes next to some distance 2 vertex form a new frontier with distance 3 from A.



Breadth-First Tree

Explore dist 0 frontier

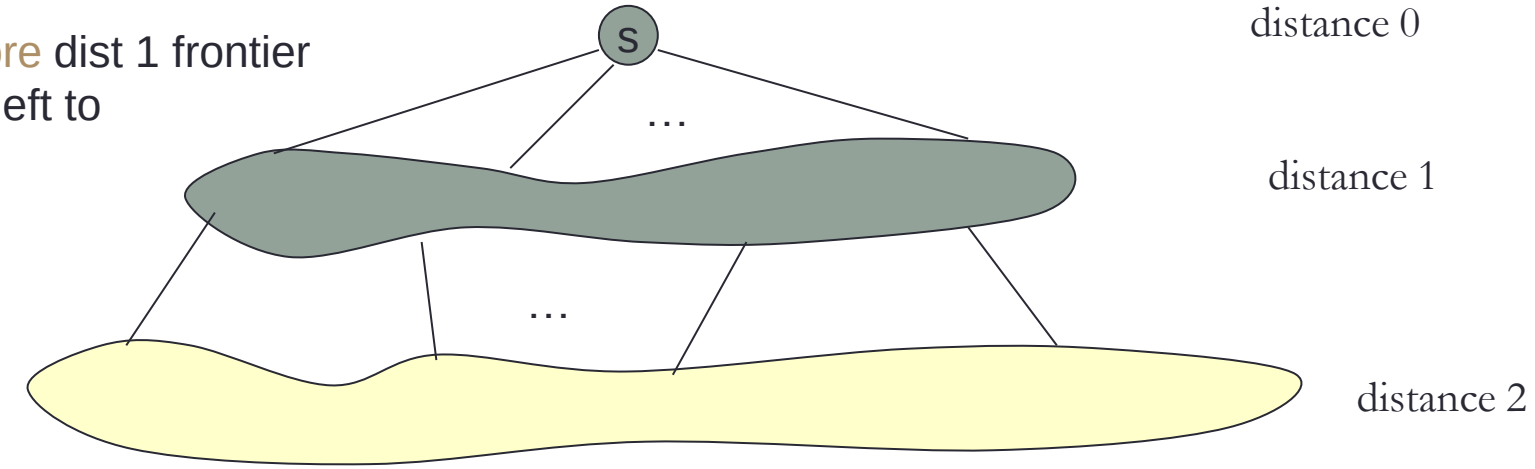


distance 0

distance 1

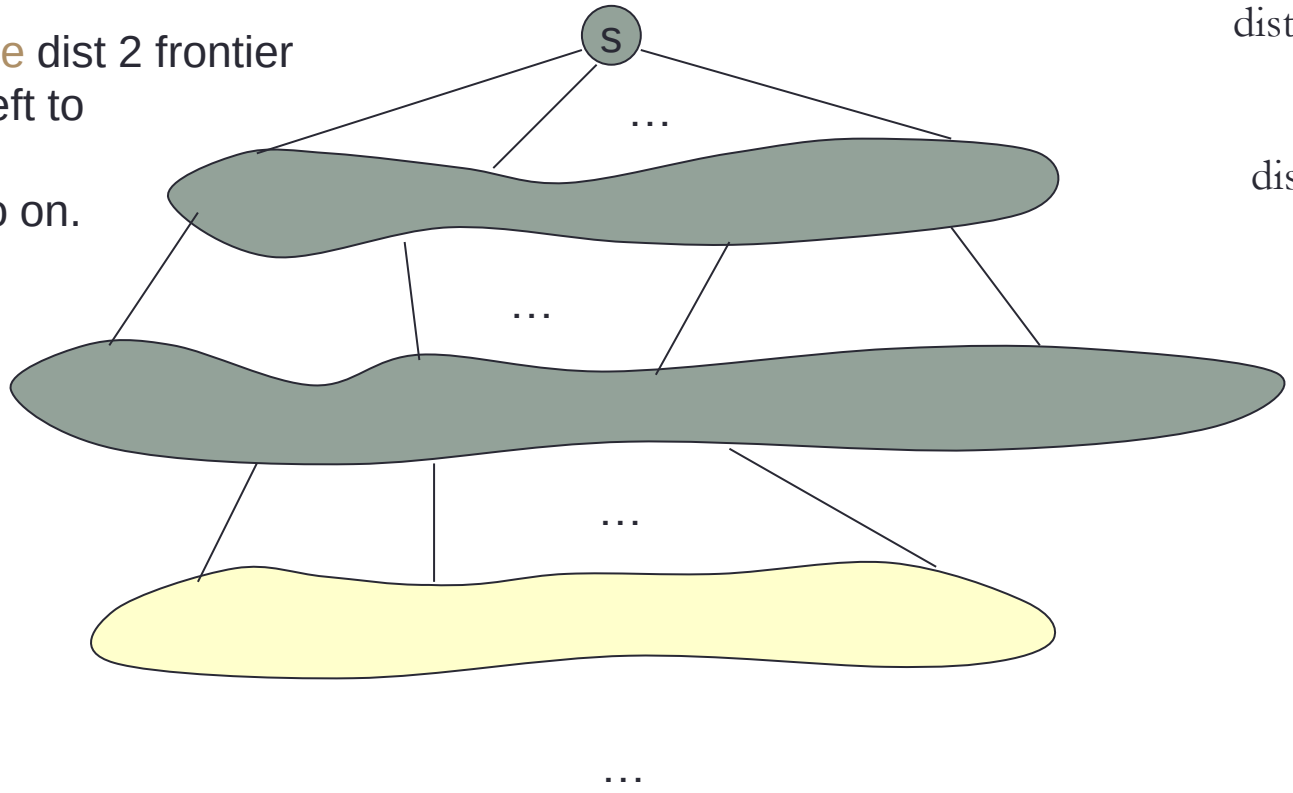
Breadth-First Tree

Explore dist 1 frontier from left to right



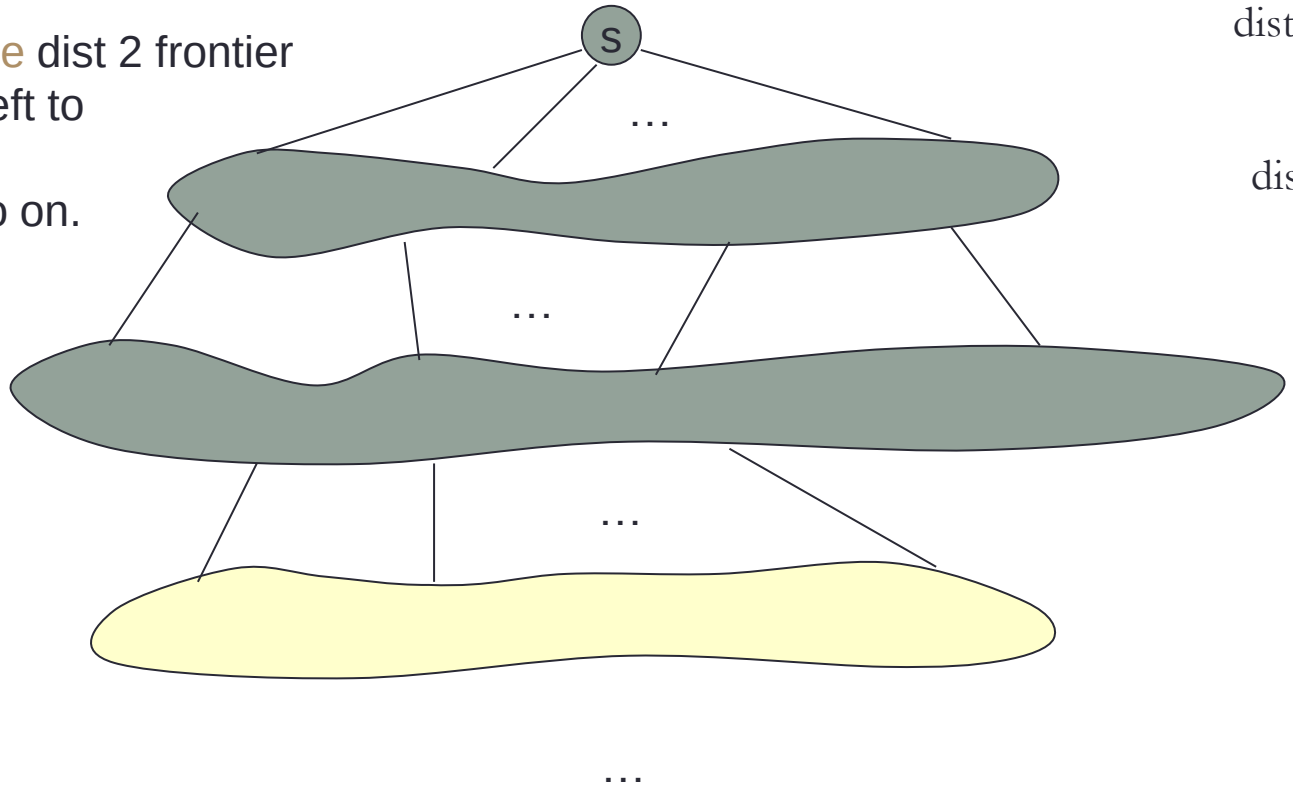
Breadth-First Tree

Explore dist 2 frontier from left to right, and so on.

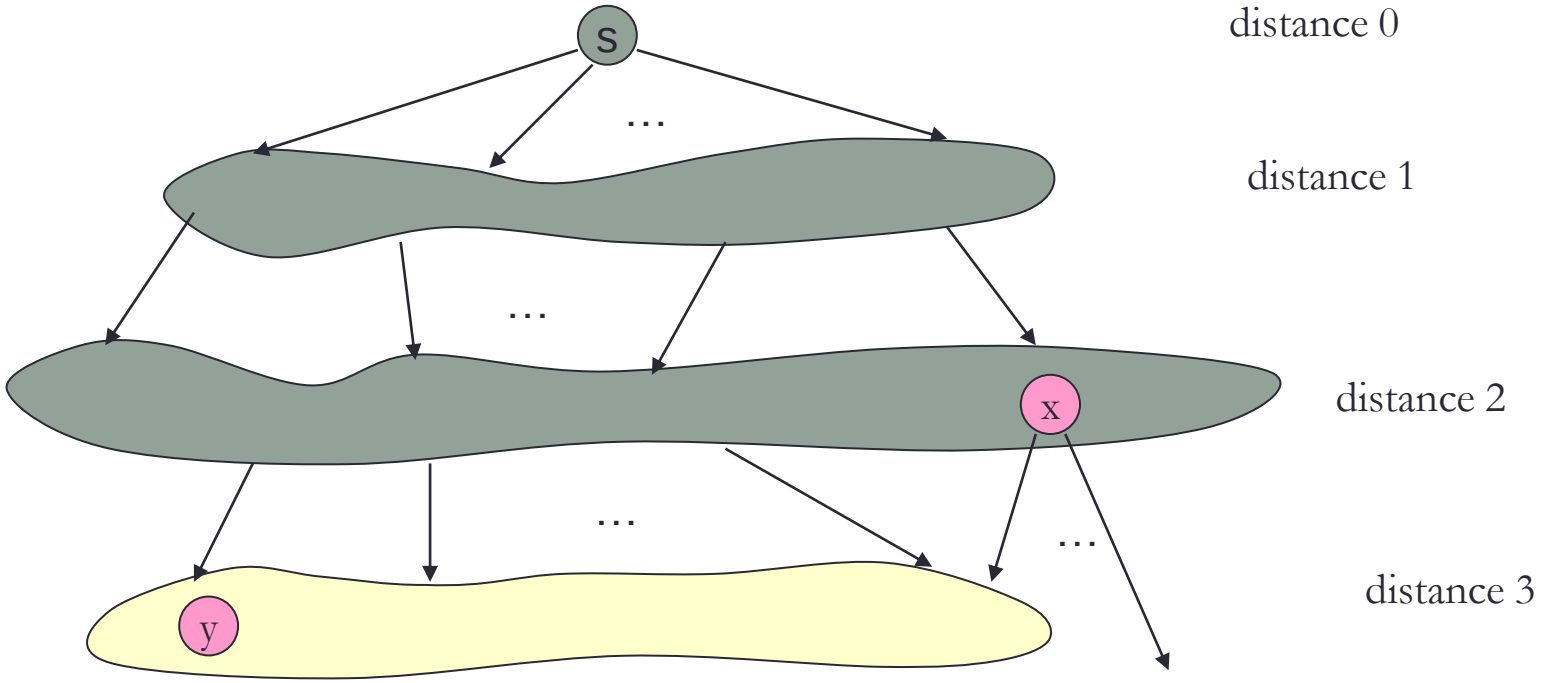


Breadth-First Tree

Explore dist 2 frontier from left to right, and so on.



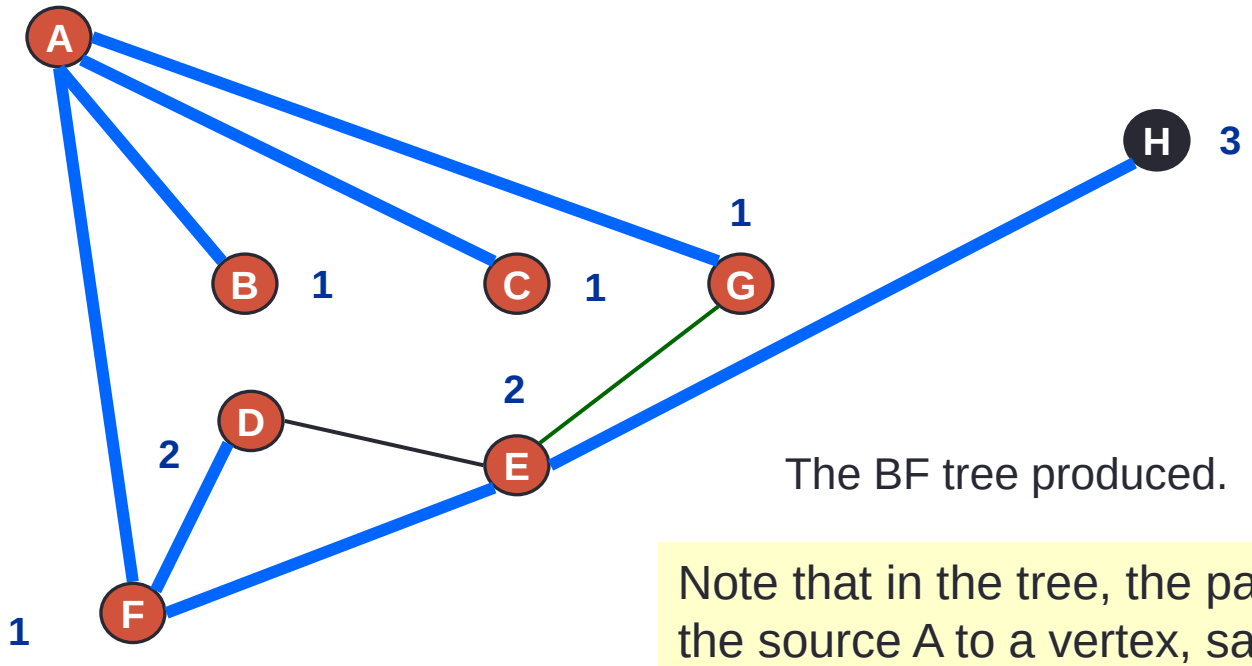
Breadth-First Tree



Note that because of the Breadth-first manner, if we visit vertex **x** before **y**, we explore **x** before **y**. In other words, if a vertex is first visited, it is first served.

Breadth-First Tree: Example

- Thus, we will use a **queue** to maintain the list of visited-but-not-explored vertices.
- A systematic way to implement BFS :



The BF tree produced.

Note that in the tree, the path from the source A to a vertex, say H, has the minimum number of edges.

Breadth-First Search (BFS) Algorithm

- Print the vertices in the visited order

```
BFS(s): // s is the source vertex
    Mark all vertices u as not visited
    Create a queue Q
    Mark s as visited and enqueue (i.e., add) s to Q

    while Q is not empty:
          dequeue (i.e., remove) a vertex u from Q
          print vertex u

          for each neighbor i of u:
              if i is not visited:
                  mark i as visited
                  enqueue i to Q
```

BFS on Adjacency List: Python code

- Print the vertices in the visited order

```
class GraphAL:
    # ...
    def BFS(self, s):
        visited = [False] * self.numNodes

        queue = []
        queue.append(s)
        visited[s] = True

        while queue:
            u = queue.pop(0)
            print (u, end = " ")
            for i in self.graph[u]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
```

All vertices are not visited yet.

Put source vertex **s** to the queue, and mark **s** as visited.

Loop until queue is empty:

Dequeue a visited vertex **u**, and print **u**

for each neighbor **i** of vertex **u**, if **i** is not visited yet, enqueue **i**

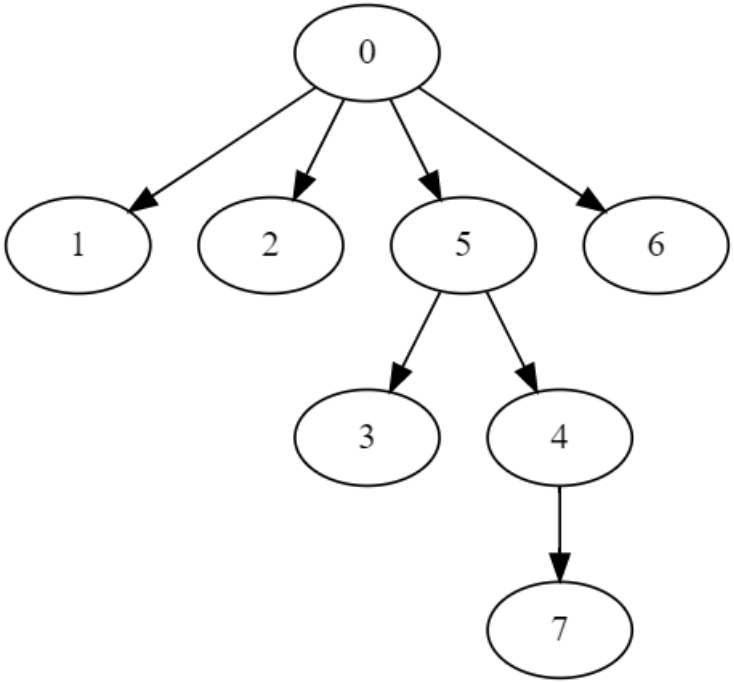
BFS on Adjacency List: Python code

- Print the vertices **and their distance from s**, and the **BF tree**.

```
from graphviz import Digraph
def BFS2(self, s):
        dist = [None] * self.numNodes # distance from s

        bf_tree = Digraph() # breadth-first tree
        queue = []
        queue.append(s)
        dist[s] = 0

        while queue:
            u = queue.pop(0)
            print ("(node %d, dist %d)" % (u, dist[u]), end = " ")

            for i in self.graph[u]:
                if dist[i] == None:
                    bf_tree.addEdge(str(u),str(i))
                    queue.append(i)
                    dist[i] = dist[u]+1
        print()
        return bf_tree
```

BFS on Adjacency List: Python code

- Print the vertices **and their distance from s**, and the **BF tree**.
- **Output:**
  (node 0, dist 0) (node 1, dist 1) (node 2, dist 1) (node 5, dist 1)
  (node 6, dist 1) (node 3, dist 2) (node 4, dist 2) (node 7, dist 3)



BFS on Adjacency List: Time complexity & Correctness

Time Complexity:

- Every vertex will be put in the queue once and take out from the queue once $\Rightarrow$ **O(V)** [i.e., O(|V|)]
- When we explore a vertex, we explore all its adjacency edges once $\Rightarrow$ **O(E)** [i.e., O(|E|)]

$\Rightarrow$ Time complexity = **O(V+E)**

Correctness: The number we find for vertex u is indeed the distance between the source s and vertex u.

- **Idea:** By Mathematical Induction

BFS on Adjacency Matrix: Python code

- Print the vertices in the visited order

```
class GraphAM:
    # ...
    def BFS(self, s):
        visited = [False] * self.numNodes

        queue = []
        queue.append(s)
        visited[s] = True

        while queue:
            u = queue.pop(0)
            print (u, end = " ")
            for i in range(self.numNodes):
                if self.graph[u][i] == 1
                   and visited[i] == False:
                    queue.append(i)
                    visited[i] = True
```

All vertices are not visited yet.

Put source vertex **s** to the queue, and mark **s** as visited.

Loop until queue is empty:

Dequeue **u**, and print **u**

for each neighbor **i** of vertex **u**, if **i** is not visited yet, enqueue **i**

BFS on Adjacency Matrix: Time complexity

- Every vertex will be put in the queue once and take out from the queue once
  $\Rightarrow$ **O(V)**

- When we explore a vertex u, we explore all vertices i once and check if (u, i) is an edge
  $\Rightarrow$ **$O(V) \times O(V) = O(V^2)$**

$\Rightarrow$ Time complexity = **$O(V+ V^2)$** = **$O(V^2)$**

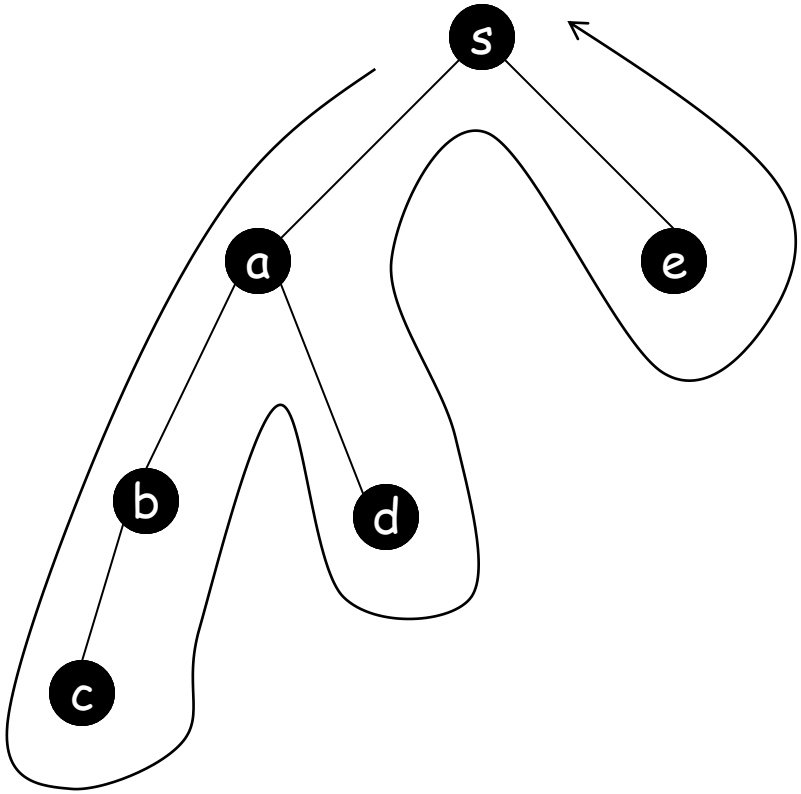
Depth-First Search (DFS)

Depth-first search is another strategy for exploring a graph; it searches "deeper" in the graph whenever possible.

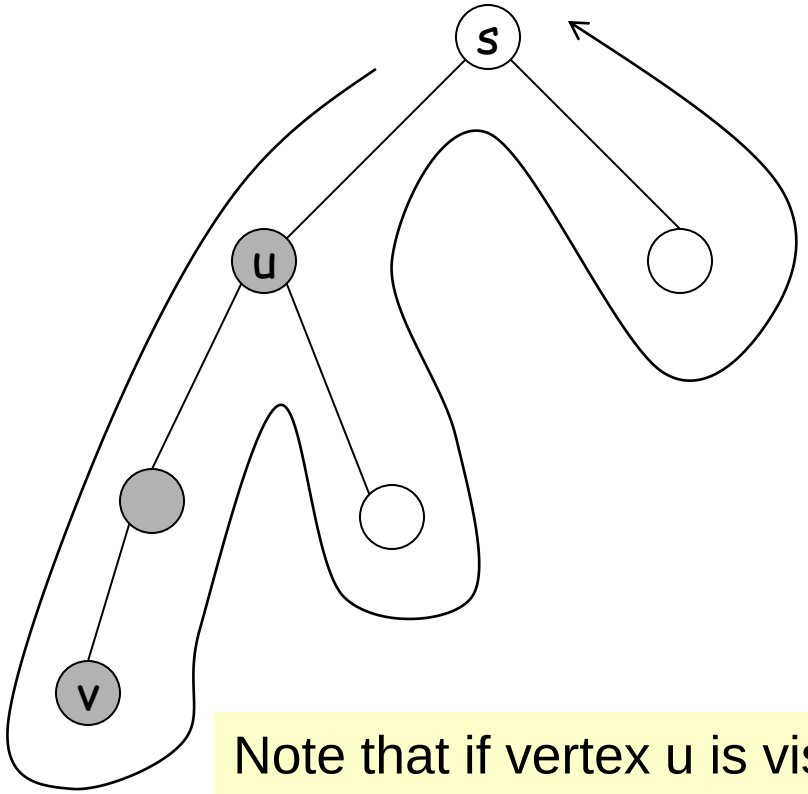
- Edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it.
- When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered.

- **Note:** BFS visits all neighbors first; while DFS keeps going deeper first until nowhere to go, and backtrack.

DFS of a Tree



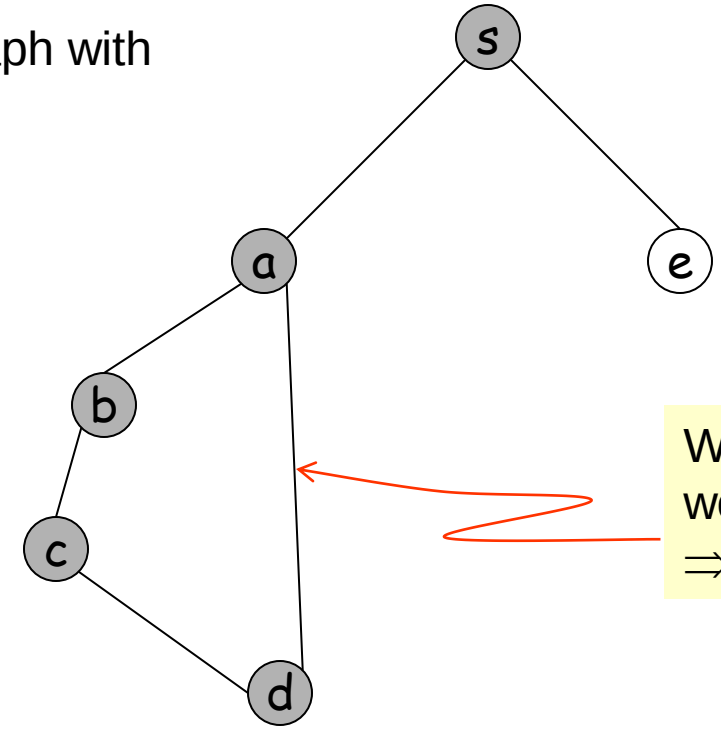
DFS of a Tree



Note that if vertex u is visited (gray) before vertex v, then vertex v will be finished processed (black) before u. ⇒ Last in first out.
Thus, we can use a **stack** to maintain the set of visited-but-not-finished vertices.

DFS of a General Graph

That is, DFS a graph with cycle.



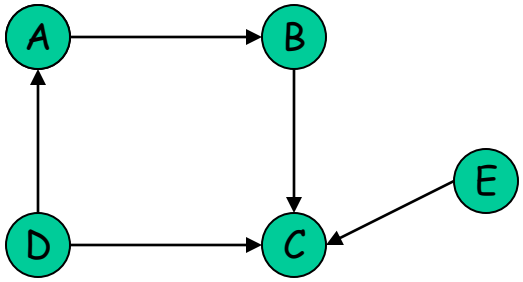
When exploring a gray vertex, we find a gray vertex
$\Rightarrow$ simply ignore this vertex.

Directed Graph: Adjacency Matrix

- The edges have direction.
- The directed graph G=(V, E) where

  V = { A, B, C, D, E} and

  E = {(A,B), (B,C), (E,C), (D,C), (D,A)}



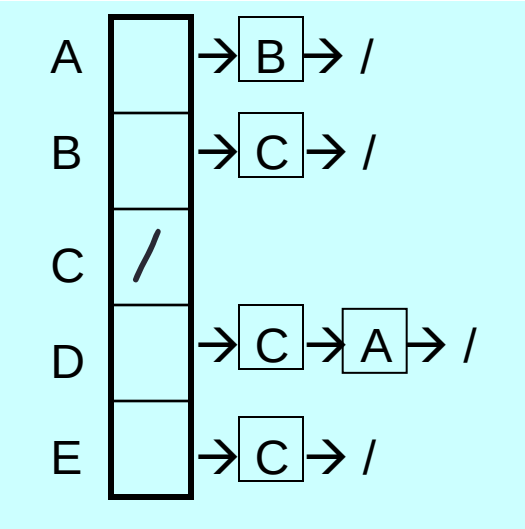
- (A,B) means the edge is from vertex A to vertex B.
- **The Adjacency matrix**

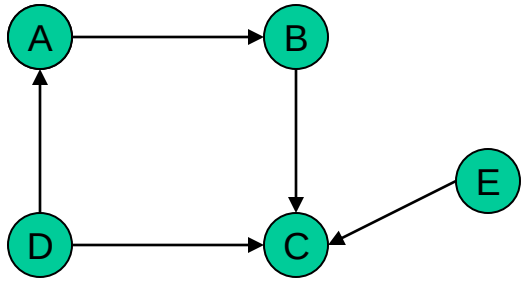
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	0	0
C	0	0	0	0	0
D	1	0	1	0	0
E	0	0	1	0	0

Note that the matrix is not necessarily symmetric, i.e., the following property not necessary hold: a[i, j] = 1 $\Leftrightarrow$ a[j, i] = 1.

Directed Graph: Adjacency List

The Adjacency list

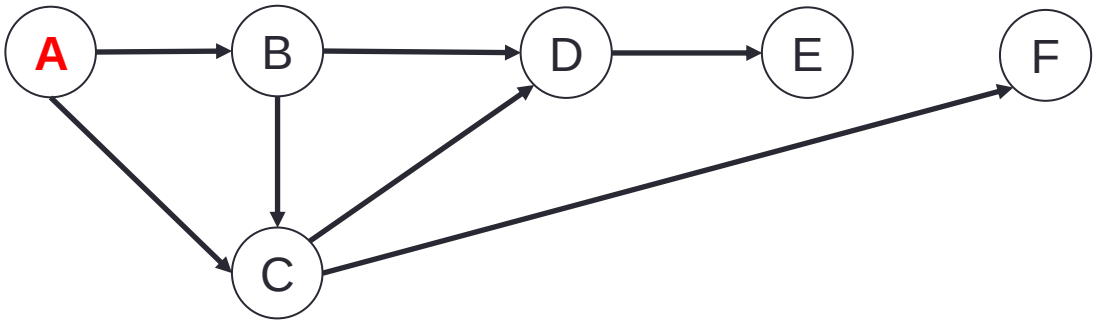




Depth-first search (Breadth-first search) on directed graph is very similar to DFS (BFS) on undirected graph, except that we must follow the direction of an edge in order to traverse it.

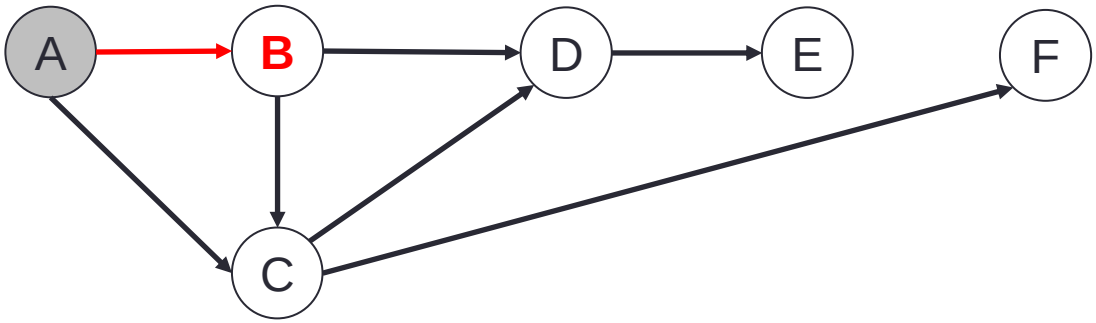
DFS: Example (Step 1)

- Suppose we start DFS from vertex A.



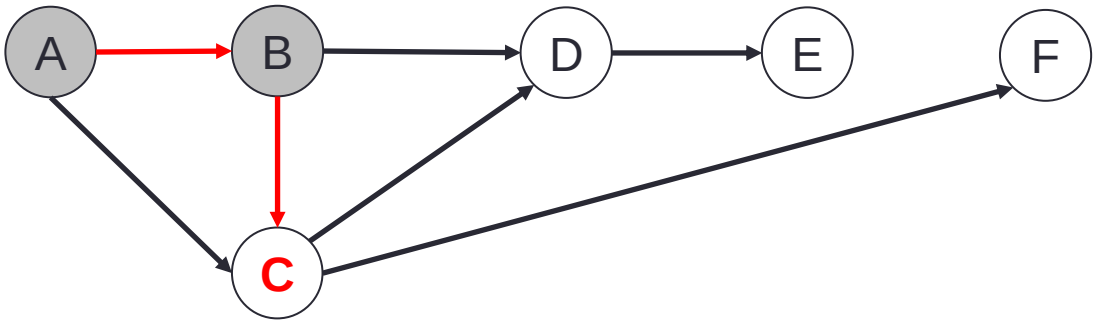
DFS: Example (Step 2)

- We mark A as visited.
- We visit B, which is an unvisited neighbor of A.



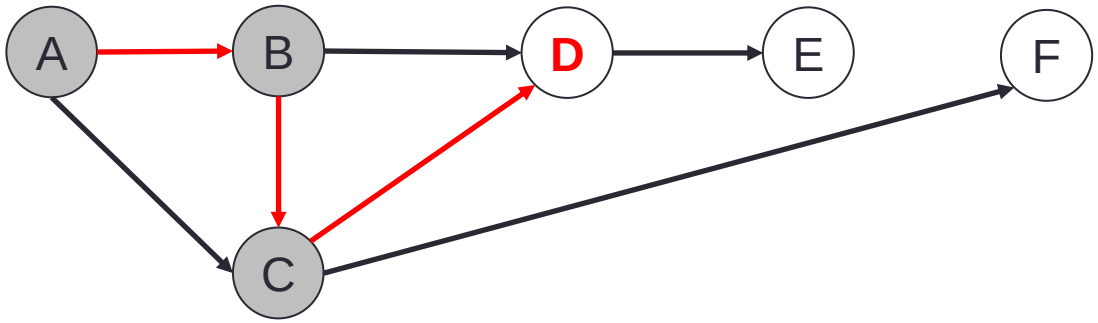
DFS: Example (Step 3)

- We mark B as visited.
- We visit C, which is an unvisited neighbor of B.



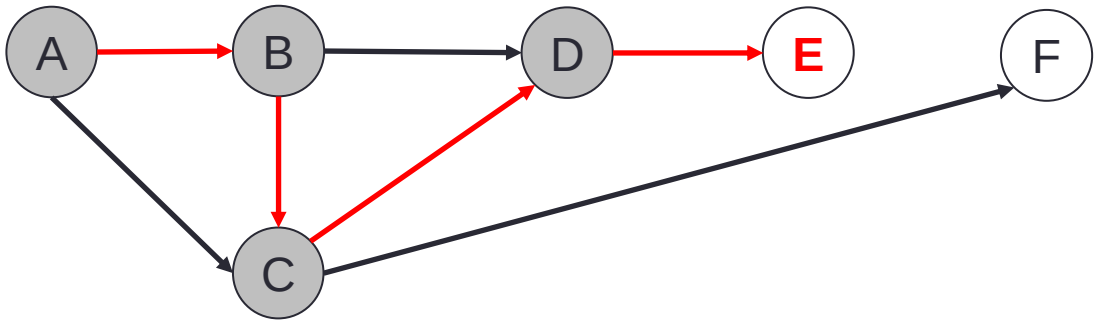
DFS: Example (Step 4)

- We mark C as visited.
- We visit D, which is an unvisited neighbor of C.



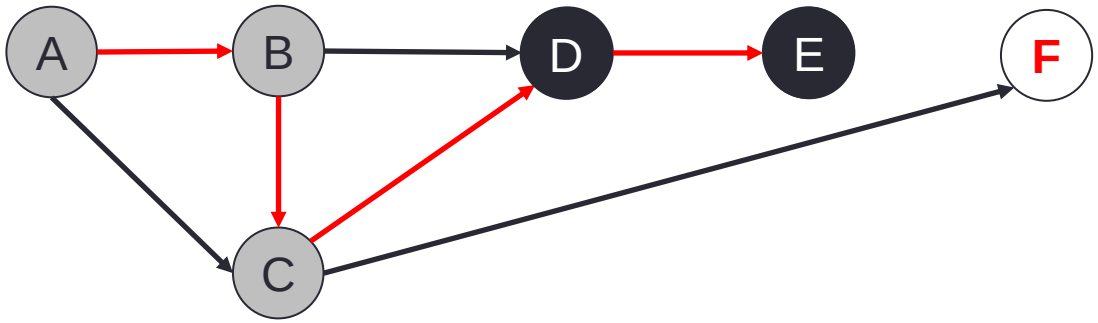
DFS: Example (Step 5)

- We mark D as visited.
- We visit E, which is an unvisited neighbor of D.



DFS: Example (Step 6)

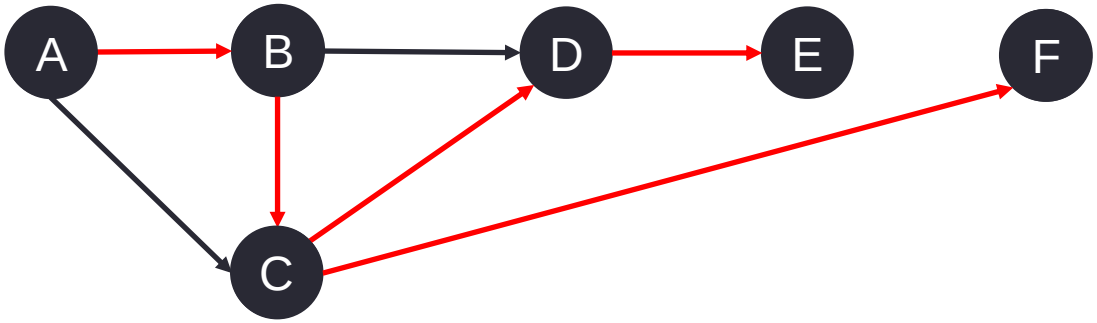
- We mark E as visited.
- E does not have any unvisited neighbor, so it becomes finished.



- We backtrack to D and try to visit another unvisited neighbor of D.
- D does not have any unvisited neighbor, so it becomes finished.
- We backtrack to C.
- We visit F, which is an unvisited neighbor of C.

DFS: Example (Step 7)

- We mark F as visited.
- F does not have any unvisited neighbor, so it is finished.
- We backtrack to C, which has no unvisited neighbor & is finished.



- We backtrack to B, which has no unvisited neighbor & is finished.
- We backtrack to A, which has no unvisited neighbor & is finished.
- We cannot backtrack anymore and the DFS stops.

Depth-First Search (DFS) Algorithm

- Print the vertices in the visited order

```
Mark all vertices u as not visited
DFS(s)

DFS(x):
    if vertex x is visited:
        return

    mark x as visited
    print vertex x

    for each neighbor y of x:
        DFS(y)
```

Time complexity

- Each vertex is visited once ⇒ **O(V)**
- All edges are explored at most twice (discover/backtrack) ⇒ **O(E)**

⇒ Time complexity = **O(V+E)**

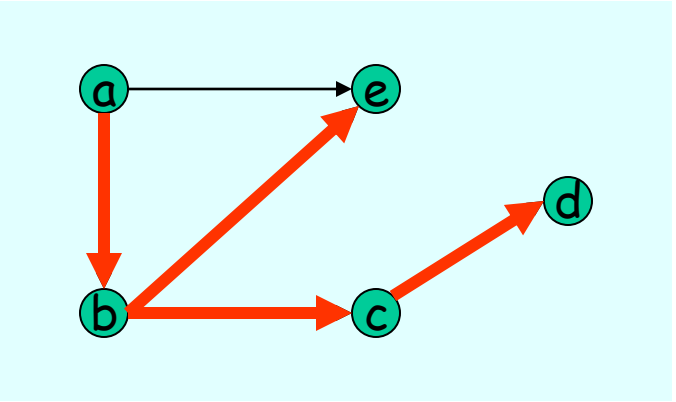
DFS on Adjacency List: Python code

- Print the vertices in the visited order

```
def DFS(self, s):
    visited = [False] * self.numNodes
    self.rdfs(s, visited) # Call the recursive function

def rdfs(self, x, visited):
    if visited[x] == True: # Skip visited vertex u
        return

    visited[x] = True
    print(x, end = ' ')

    for y in self.graph[x]:
        self.rdfs(y, visited)
```

Depth-First Tree

- What do we get after a depth-first search on a directed graph?
- **Answer:** A depth-first tree.



- To be more precise, define the predecessor function π:

$\pi(v) = u$ if v is first discovered when we are exploring u.

- Then, for the example in this slide, the tree $G_\pi(V, E_\pi)$ is

$V = \{a, b, c, d, e\}$

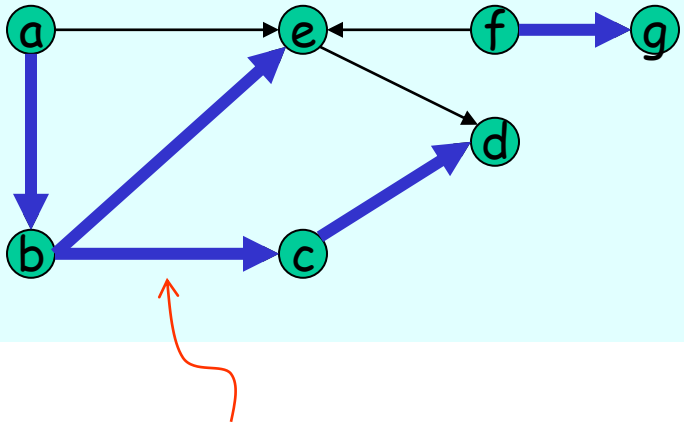
$E_\pi = \{(\pi(b),b), (\pi(c), c), (\pi(d),d), (\pi(e),e)\}$

$= \{(a, b), (b, c), (c, d), (b, e)\}$

Depth-First Tree (cont')

- In general, we get a depth-first forest

The **forest** $G_\pi(V, E_\pi)$ where
$V = \{a, b, c, d, e, f, g\}$
$E_\pi = \{(\pi(b),b), (\pi(c), c), (\pi(d),d), (\pi(e),e), (\pi(g),g)\}$
$= \{(a, b), (b, c), (c, d), (b, e), (f, g)\}$



Time complexity: O(V+E)

Tree edges

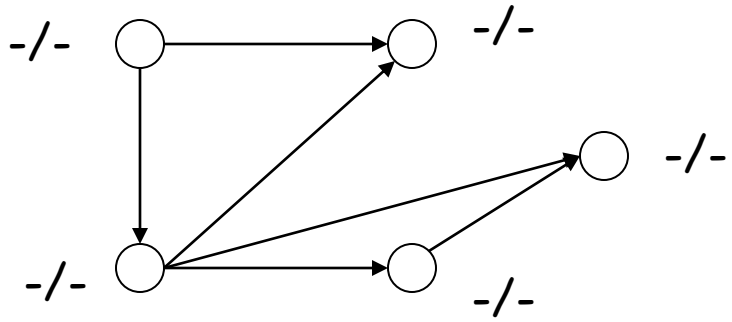
Timestamps

- Timestamp is a simple, but important notation.

- During the execution of a DFS, every node v will be assigned two timestamps:
  - **d[v]**: discovery time of v, which records when v is first discovered, i.e., when v's color is changed from white to gray.
  - **f[v]**: finish time of v, which records when the search finishes examining v's adjacency list, i.e., when v's color is changed from gray to black.

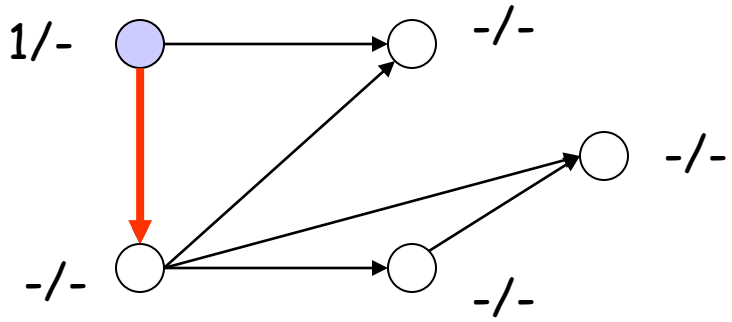
Timestamps: Example

- Initially, all vertices are un-discovered, and all have color white.

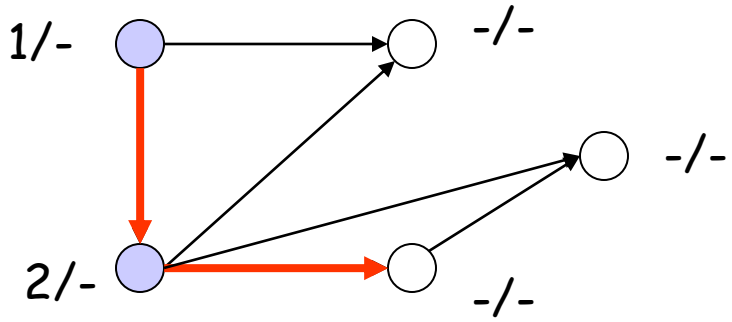


Timestamps: Example (Time 1)

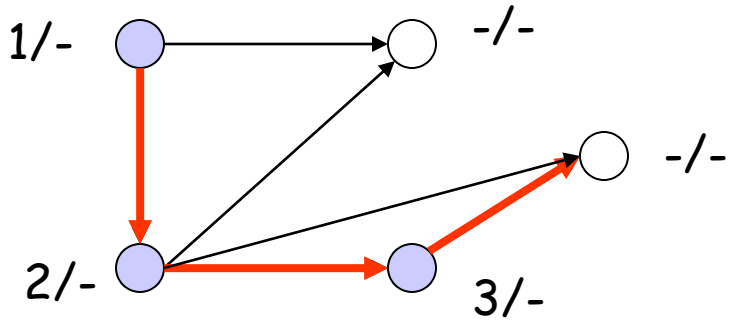
- At time 1, the source is discovered, and its color is changed to gray.



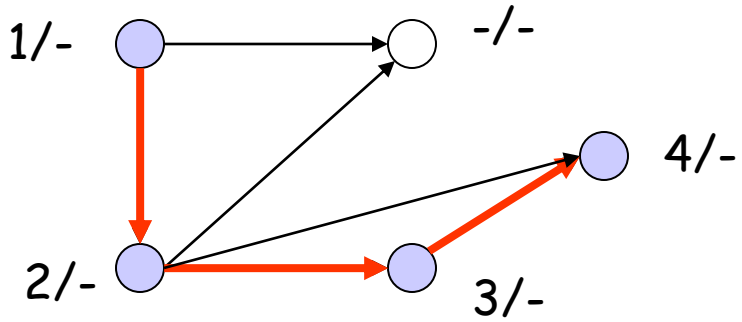
Timestamps: Example (Time 2)



Timestamps: Example (Time 3)

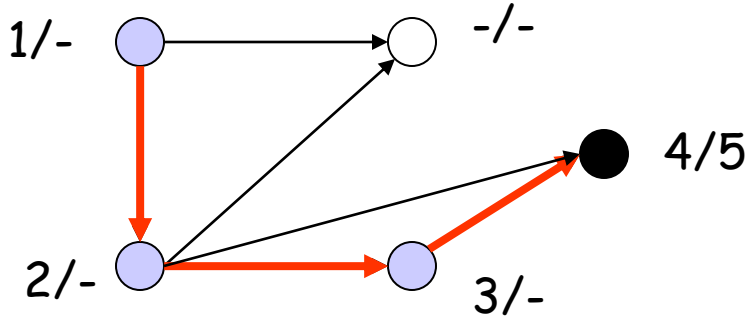


Timestamps: Example (Time 4)

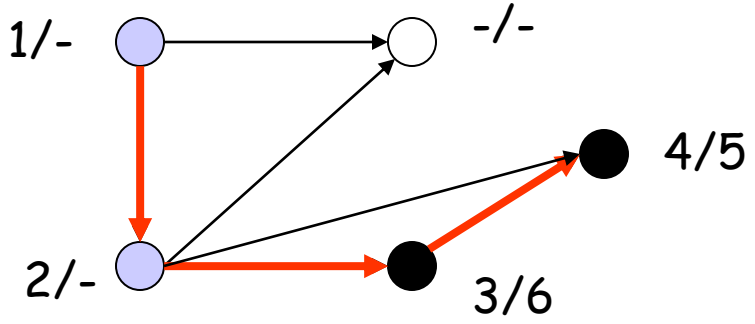


Timestamps: Example (Time 5)

- At time 5, the vertex has explored all its outgoing adjacent edges (indeed, it has none).
- Its color is changed to black, and its finish time is set to 5.

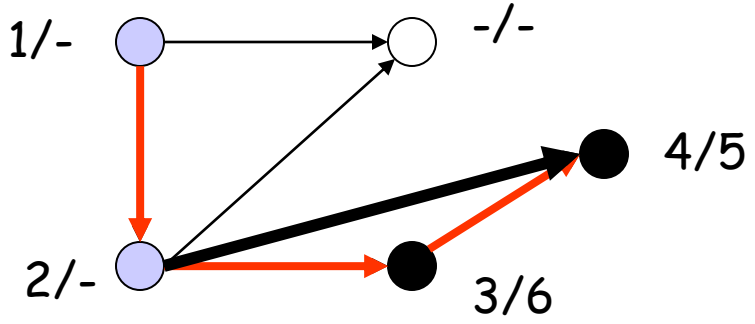


Timestamps: Example (Time 6)

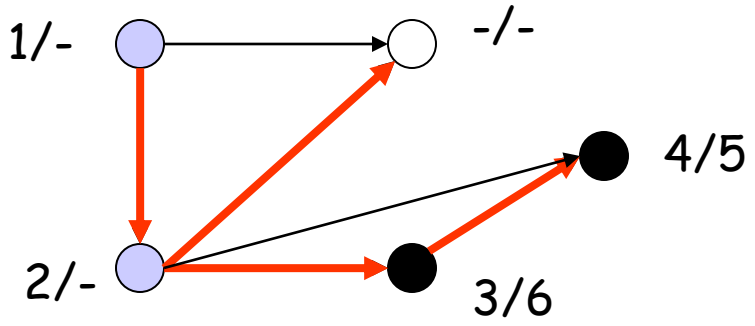


Timestamps: Example (Time 7)

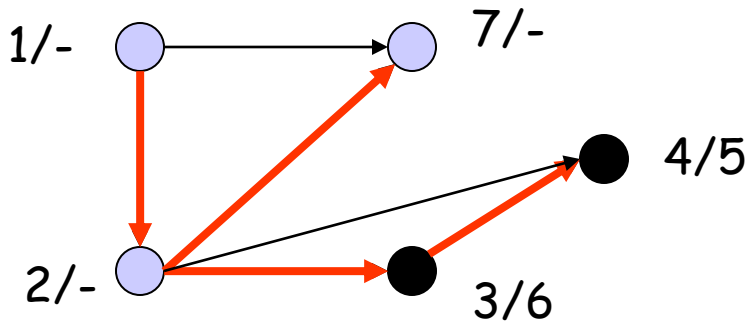
- A gray vertex u finds a black vertex v; the corresponding edge cannot be a tree edge because the black vertex has already had a predecessor, i.e., the $\pi(v)$ is found.



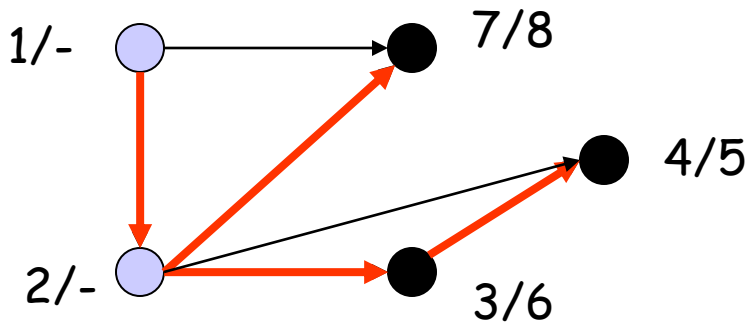
Timestamps: Example (Time 7)



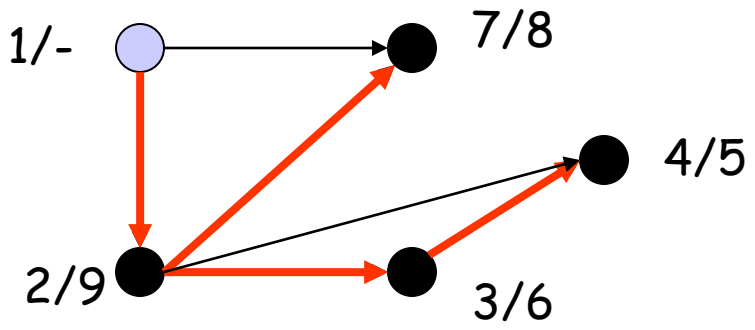
Timestamps: Example (Time 7)



Timestamps: Example (Time 8)

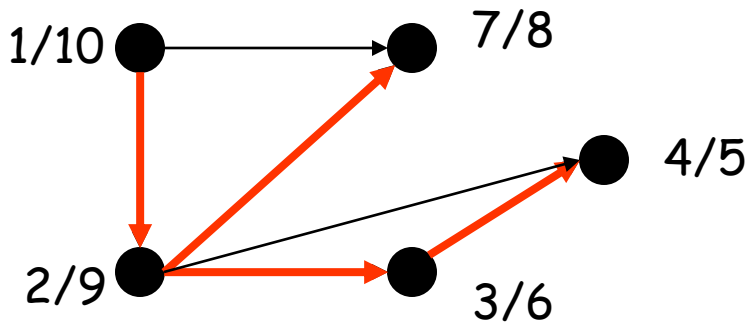


Timestamps: Example (Time 9)



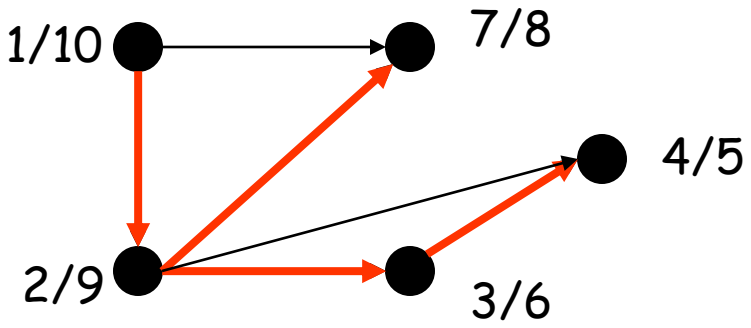
Timestamps: Example (Time 10)

- Note that we increase the time stamp only when some vertex changes color.



Parenthesis Theorem

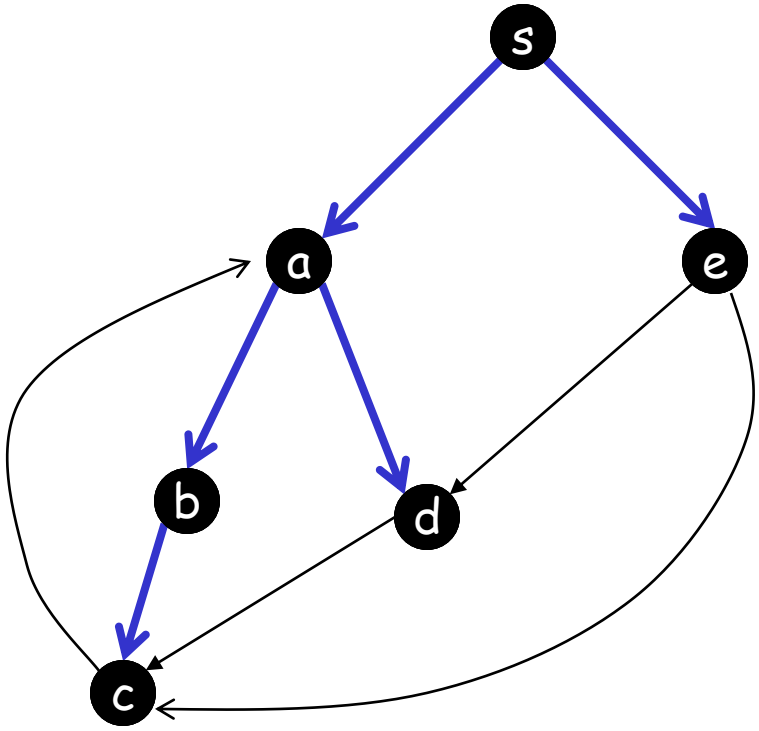
Theorem (Parenthesis theorem)
Given any two intervals
[d(u), f(u)] and [d(v), f(v)], either
(1) they are entirely disjoint, or
(2) one of them is contained totally within another.



1	2	3	4	5	6	7	8	9	10
(	(	(	(	)	)	(	)	)	)

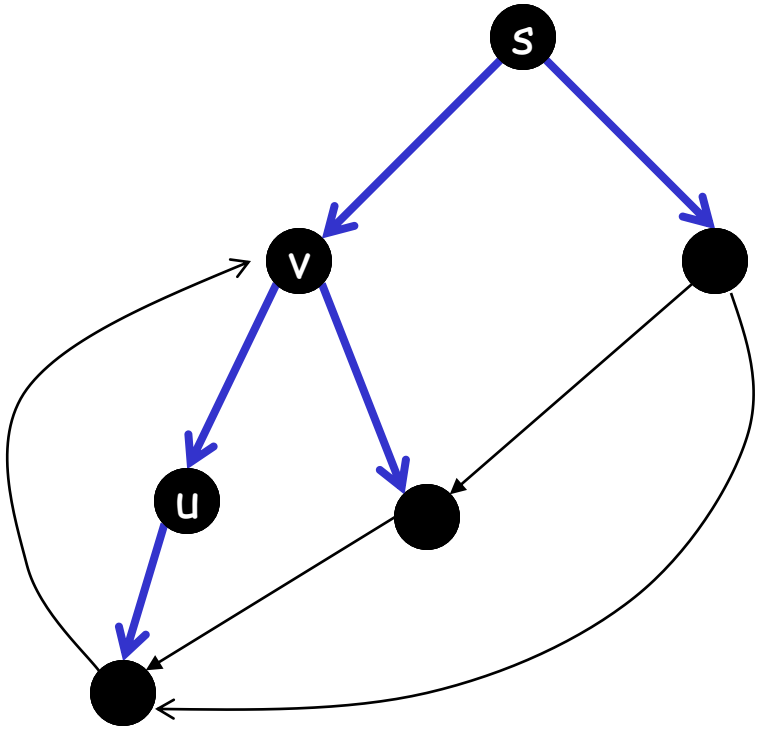
Proof (Sketch)

- Consider any two vertices u and v. Look at the DF tree.

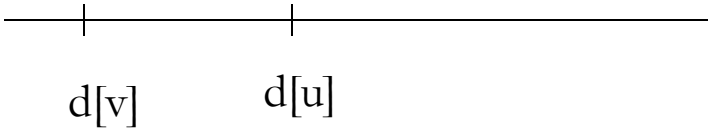


Proof (Sketch): Case 1

- Consider any two vertices u and v. Look at the DF tree.

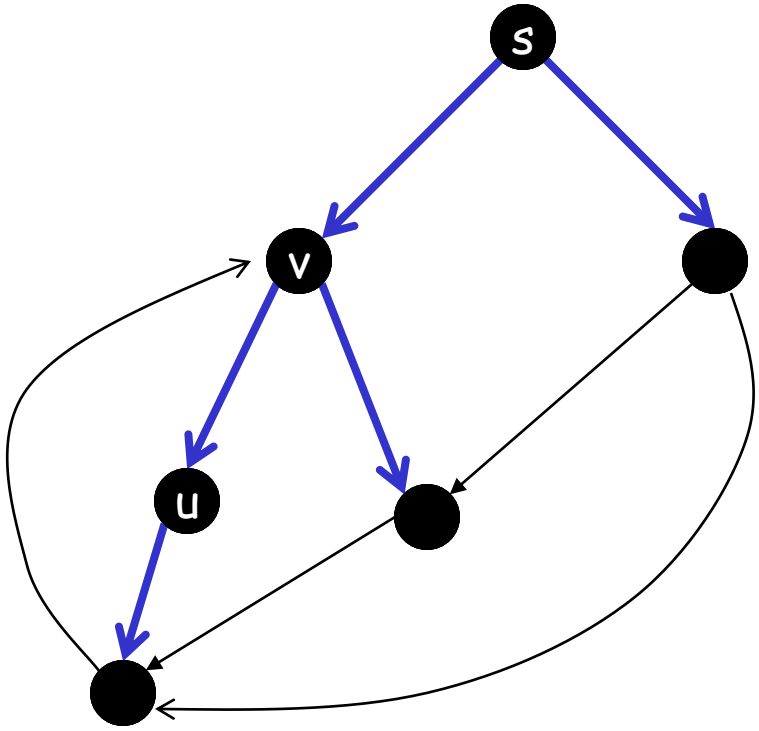


Suppose v is a parent of u, i.e., $\pi(u) = v$. Then, we discover u from v $\Rightarrow$ d[v] < d[u].

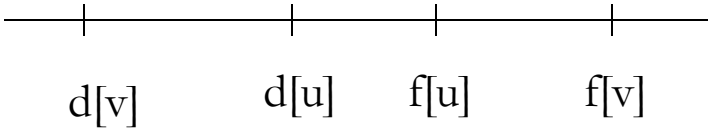


Proof (Sketch): Case 1

- Consider any two vertices u and v. Look at the DF tree.



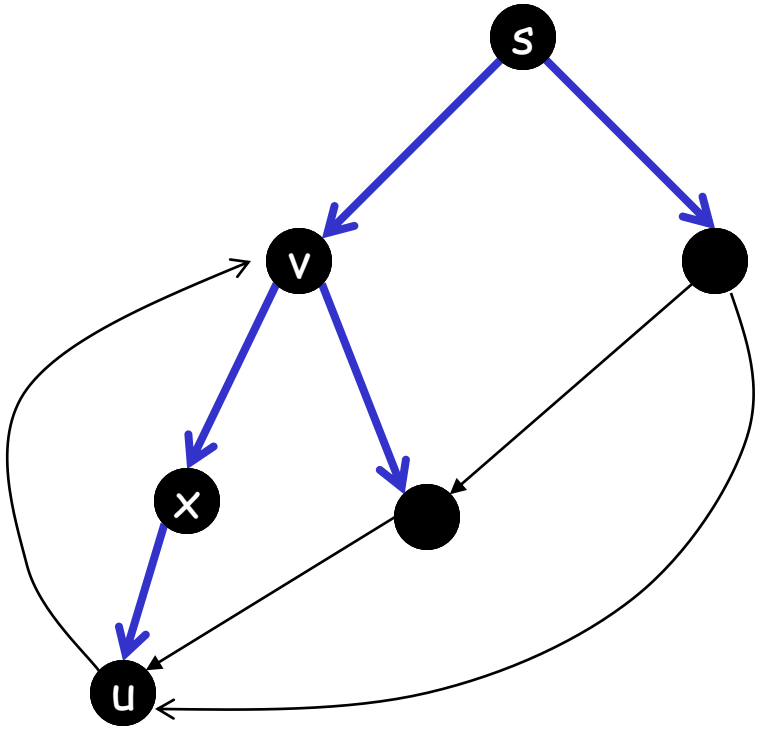
Suppose v is a parent of u. Then, we finish exploring u before we return to v
$\Rightarrow$ f[u] < f[v].



Thus, the interval [d[v],f[v]] includes [d[u],f[u]].

Proof (Sketch): Case 1

- Consider any two vertices u and v. Look at the DF tree.



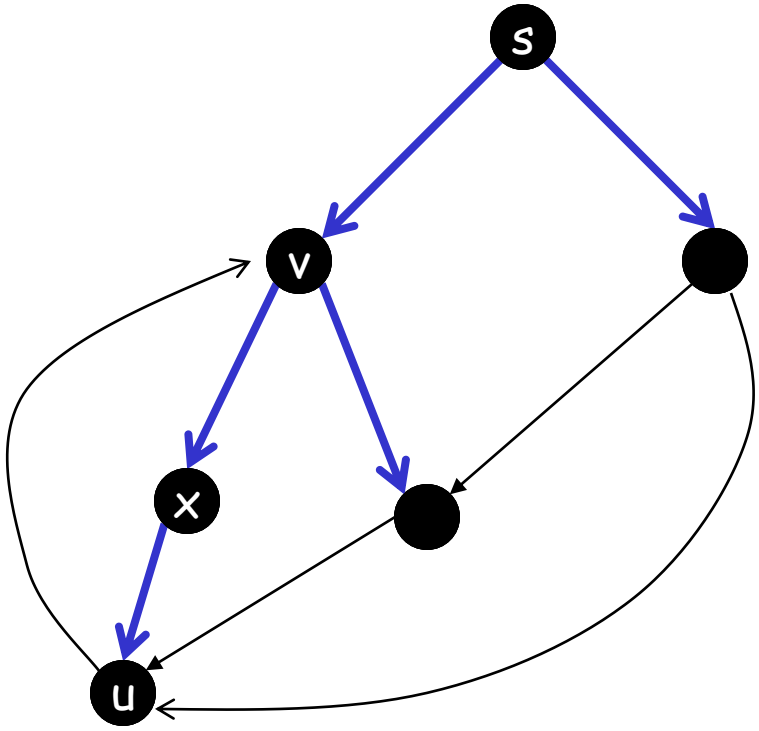
In general, suppose v is an ancestor of u. i.e., v is on the path from the source s to u.
Suppose there is a vertex x between u and v on the path (the case when there are more than one vertex can be handled identically). Then

- [d[v], f[v]] includes [d[x],f[x]]
- [d[x],f[x]] includes [d[u],f[u]]

$\Rightarrow$ [d[v],f[v]] includes [d[u],f[u]].

Proof (Sketch): Case 1

- Consider any two vertices u and v. Look at the DF tree.

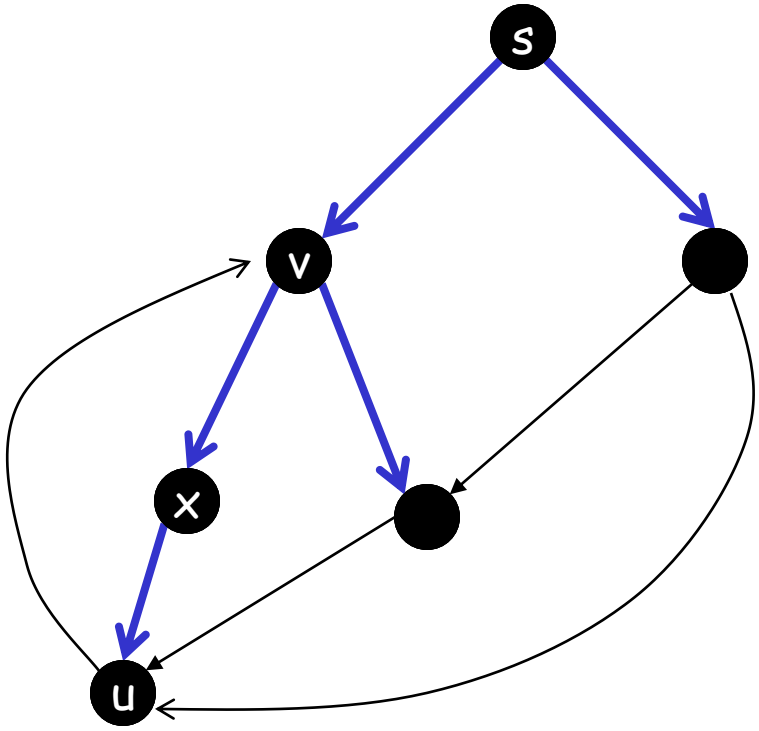


Conclusion:
if v is an ancestor of u, then
[d[v],f[v]] includes [d[u],f[u]].

Symmetrically,
if u is an ancestor of v, then
[d[u],f[u]] includes [d[v], f[v]]

Proof (Sketch): Case 2

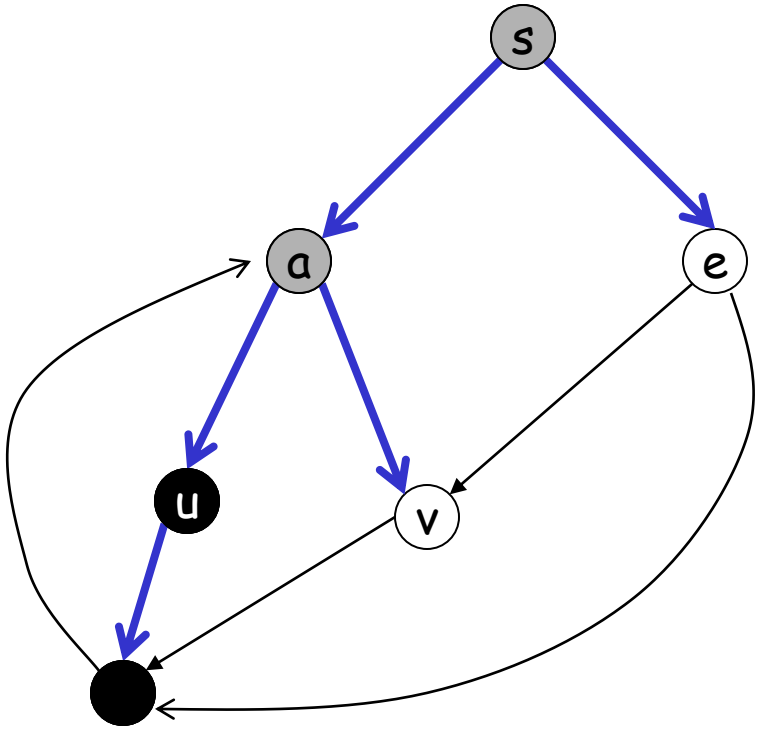
- Consider any two vertices u and v. Look at the DF tree.



Now, consider the case when u and v does not have any ancestor-descendant relation.

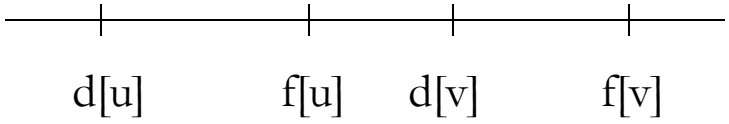
Proof (Sketch): Case 2

- Consider any two vertices u and v. Look at the DF tree.



Now, consider the case when u and v does not have any ancestor-descendant relation.

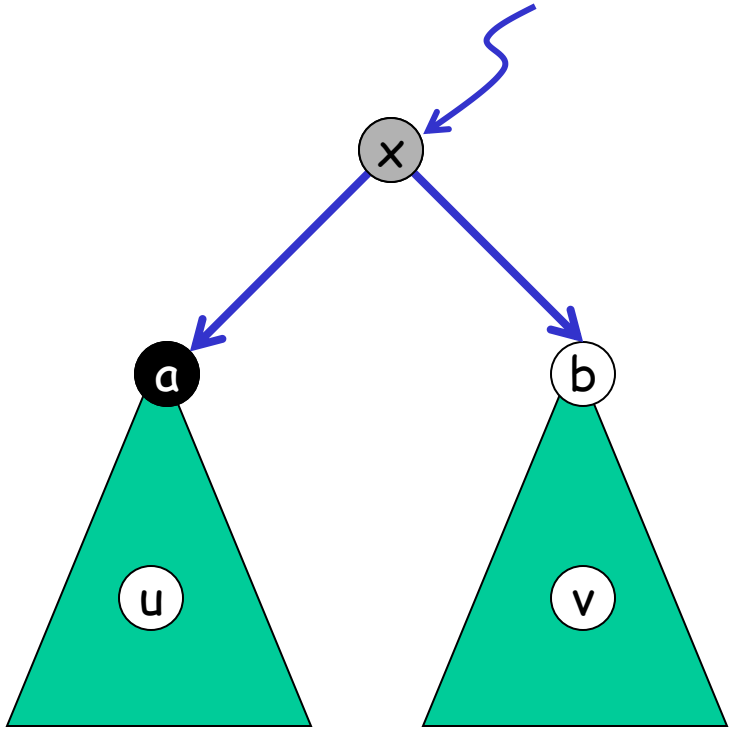
The simplest case: u and v are siblings.
Suppose a explores u before v.
Then, f[u] < d[v]



Thus, [d[u],f[u]] and [d[v],f[v]] are disjoint.

Proof (Sketch): Case 2

- Consider any two vertices u and v. Look at the DF tree.



In general, let x be the least common ancestor of u and v.

- Since a is an ancestor of u
  $\Rightarrow$ [d[a],f[a]] includes [d[u],f[u]].
- Since b is an ancestor of v
  $\Rightarrow$ [d[b],f[b]] includes [d[v],f[v]].

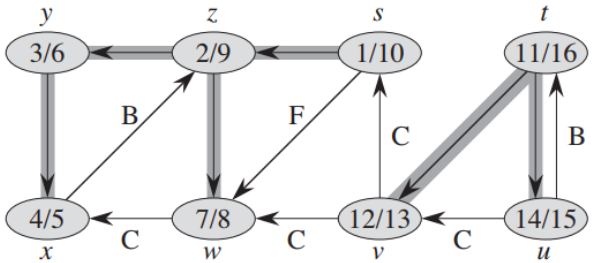
Furthermore, a and b are siblings
$\Rightarrow$ [d[a],f[a]] and [d[b],f[b]] are disjoint.

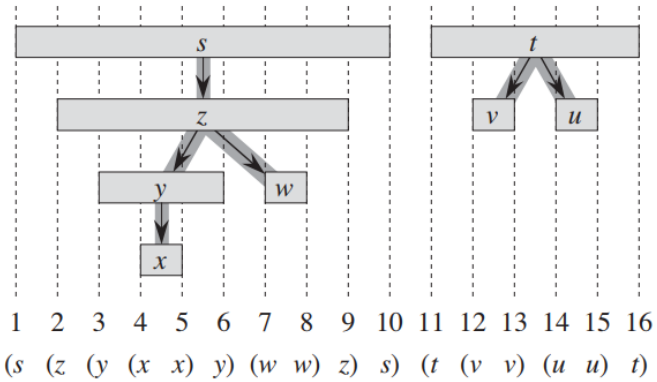
Thus, [d[u],f[u]] and [d[v],f[v]] are disjoint.

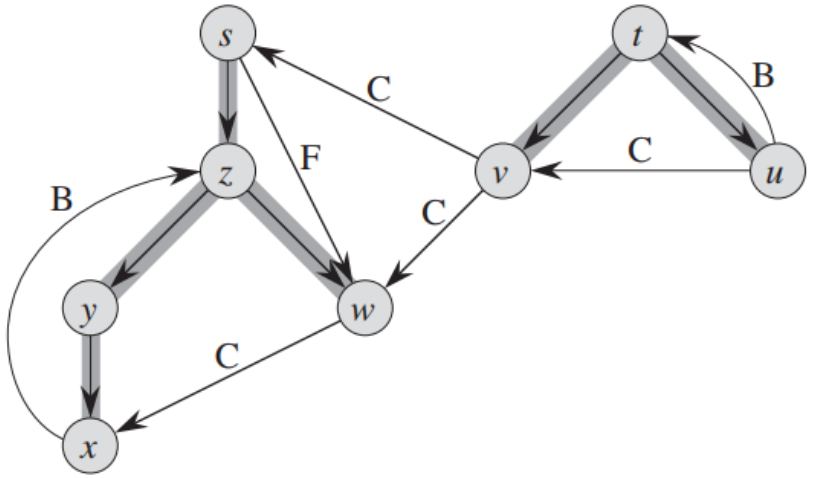
Classification of edges

- Tree edges: (π(v),v) for every v.
- **B**ack edges: those edges from a vertex to its ancestor.
- **F**orward edges: those non-tree edges from some vertex u to one of its descendant v.
- **C**ross edges: go from one tree to another tree.

Example:



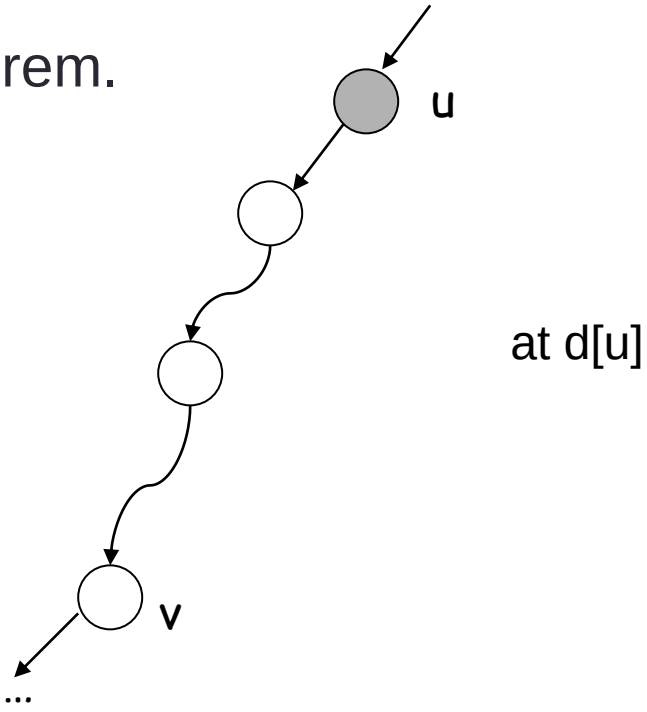




White-Path Theorem

In the depth-first forest, vertex v is a descendant of u if and only if at time d(u) that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

- It can be proved using Parenthesis Theorem.
- The details are omitted from the notes.



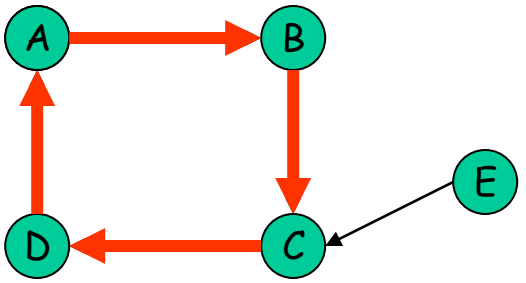
Topological Sort: Directed Acyclic Graphs

- Topological Sort is a simple application of DFS.
- We first introduce the notion of Directed Acyclic Graphs (DAG).

- A directed graph may have a cycle, which is a sequence of edges:

(a,b), (b,c), (c,d),…, (x,a) ← End point of last edge is the starting point of first edge

The end point of one edge is the starting point of the following edge.



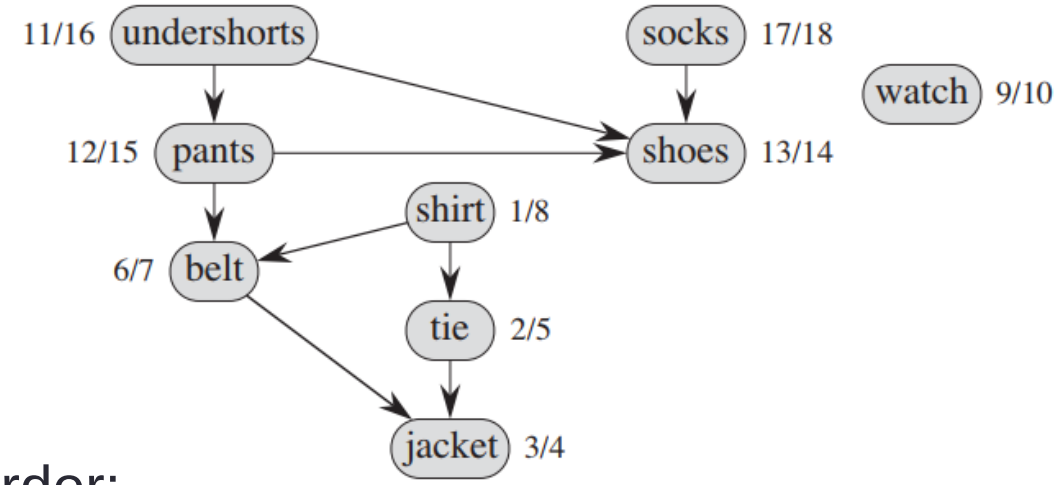
(A,B), (B,C), (C,D), (D,A)

- A DAG is a directed graph that does not contain any cycle.

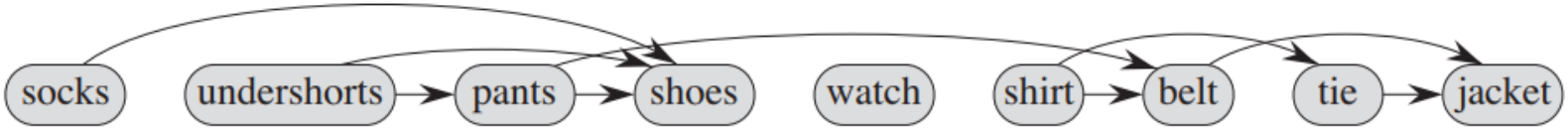
Topological Sort on a DAG

Find a linear order of the vertices such that for any edge **(u,v)** in the DAG,
u appears before v in the order.

- DAG example:

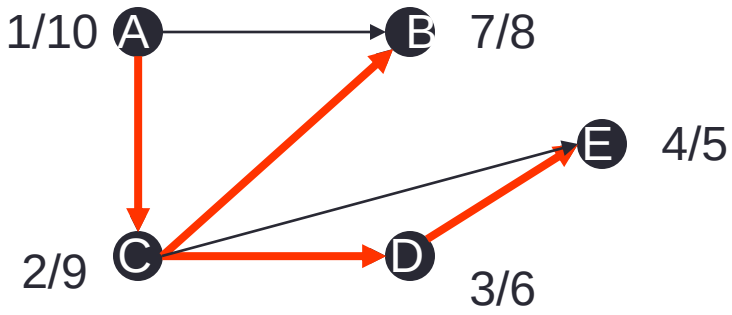


- Topological sort order:



Topological Sort Algorithm

```
TopologicalSort():
1. Call DFS(s) to compute the
   finish time of every
   unvisited vertex s.
2. Push vertex v onto a stack as
   soon as f[v] is decided.
3. Repeatly pop and output the
   vertex until stack is empty.
   (This step basically lists
   all vertices in descending
   order of the finishing time.)
```



A

C

B

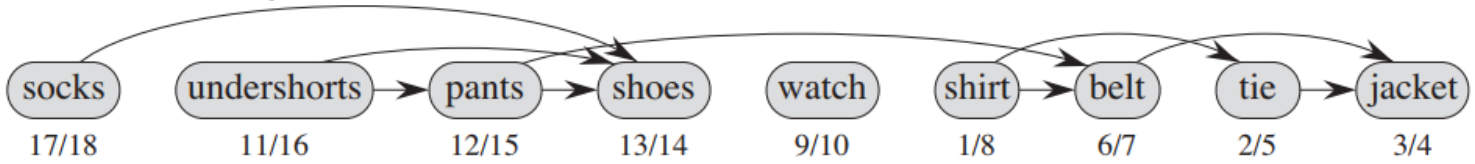
D

E

A C B D E

- Time complexity: O(V+E)

- Previous example:



Topological Sort: Python code on AL

- Print the topologically sorted vertices

```
def TopologicalSort(self):
    visited = [False] * self.numNodes
    stack = []
    for s in range(self.numNodes):
        if visited[s] == False:
            self.rdfs3(s, visited, stack)

    while stack: # print the topologically sorted vertices
        print(stack.pop(), end = ' ')
    print()

def rdfs3(self, x, visited, stack):
    if visited[x] == True: # Skip visited vertex x
        return

    visited[x] = True # If x is not visited, mark it visited
    for y in self.graph[x]:
        self.rdfs3(y, visited, stack)
    stack.append(x)
```

Lemma 1

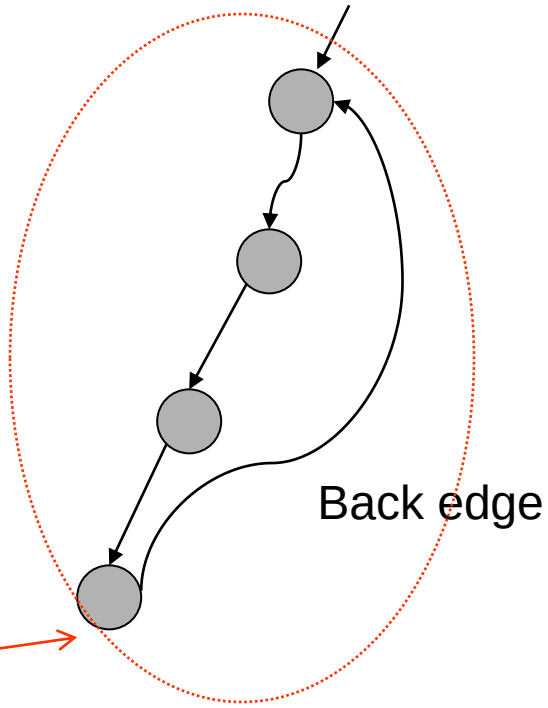
Lemma 1: A directed graph G is acyclic if and only if DFS(G) yields no back edges.

Proof.

$\Rightarrow$ direction:
(i.e., If G is acyclic,
then DFS(G) does not have back edge)

It is equivalent to prove:

If DFS(G) has back edge,
then G is not acyclic.



this is the cycle

Lemma 1 (cont')

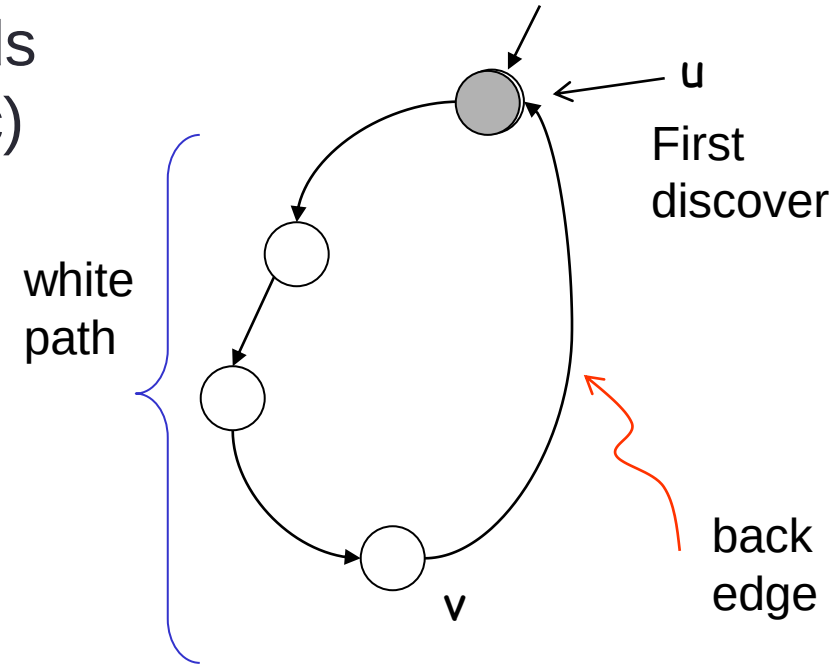
Lemma 1: A directed graph G is acyclic if and only if DFS(G) yields no back edges.

Proof (idea).

$\Leftarrow$ direction: (i.e., if DFS(G) yields no back edges, then G is acyclic)

It is equivalent to prove:

If G has cycle, then DFS(G) has a back edges.



u is an ancestor of v

Proof of Correctness

- To prove `TopologicalSort()` is correct, it suffices to prove that for any edge (u,v) in G, f[u] > f[v].

- When DFS(G) explores (u,v), v cannot be gray;
  otherwise (u,v) is a back edge, and by Lemma 1, G is not acyclic.

Thus, we have two remaining cases:

- v is white: then v is a descendant of u $\Rightarrow$ f[u] > f[v].
- v is black: then v is finished, but u is not finished (it's still exploring (u,v))
  $\Rightarrow$ f[u] > f[v].

Visualization of Algorithms

- Check VisuAlgo for more visualizations: https://visualgo.net/en/dfsbfs