COMP S265F(W) & 2650SEF & 8650SEF Unit 4: Graph Algorithms (BFS, DFS, Topological Sort)

Dr. WANG DAN, Debby
dwang@hkmu.edu.hk
School of Science and Technology
Hong Kong Metropolitan University

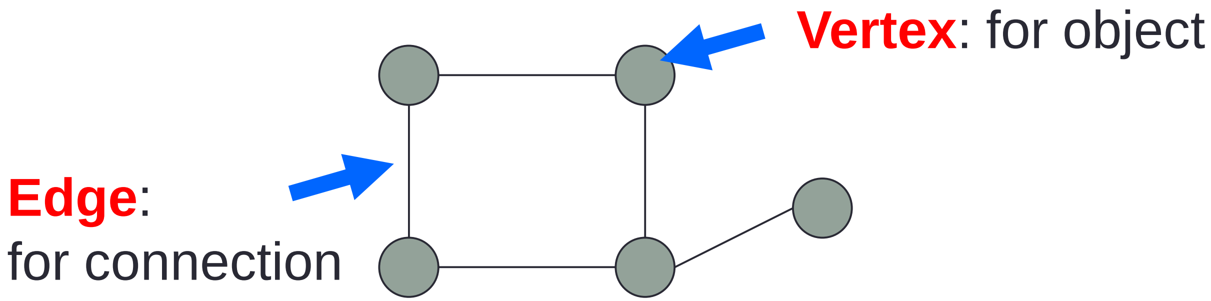
Many thanks to Dr. Keith Lee for providing the original course materials.

Overview

- **Graph representation:** Adjacency Matrix, Adjacency List
- **Breadth-First Search (BFS)**
  - Breadth-First Tree
  - BFS algorithm design and analysis
- **Depth-First Search (DFS)**
  - Directed Graphs
  - Depth-First Tree
  - DFS algorithm
  - Timestamps, Parenthesis Theorem, White-Path Theorem
- **Topological Sort**
  - Directed Acyclic Graphs (DAG)
  - Algorithm design (Simple application of DFS) and analysis

Graph algorithms

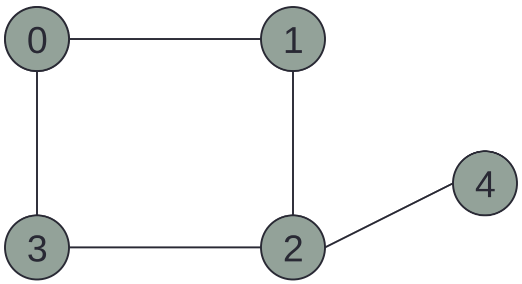
- **Graphs**: A set **V** of OBJECTS (**vertices**) with a set **E** of pairwise CONNECTIONS (**edges**).
- Design and analysis of graph algorithms is a **challenging branch** of computer science.



- **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**

Index *j*

Index *i*		*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**

Generation: For any pair of vertices **i, j**, **a[i,j] = 1** if edge (i, j) exists; otherwise **a[i,j] = 0**.

Symmetry: (i, j) is an edge ⇔ (j, i) is an edge, so **a[i,j]=a[j,i]**.

Adjacency Matrix: Python code

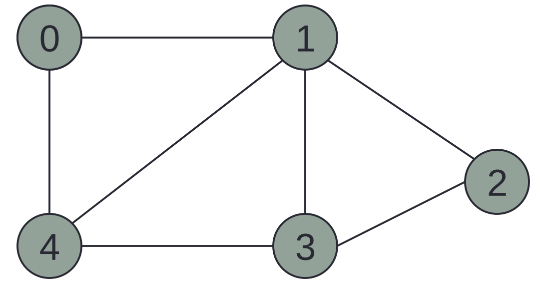
```
class GraphAM:

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

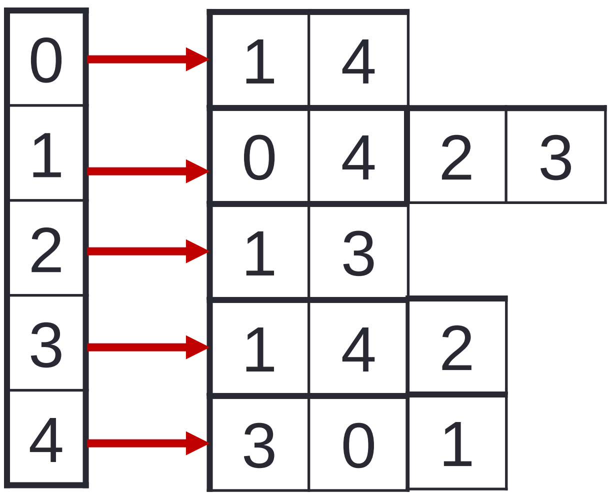
- O(**|V|^2**) space, even if *|E| is very small*.
- **O(1) time** to decide if an edge (i, j) exists.

- To run a Jupyter notebook (with file extension .ipynb), install the ***notebook*** package by `conda install notebook`

Adjacency List

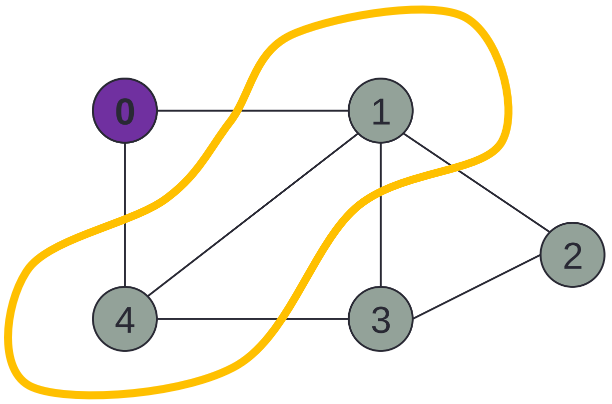
- Every vertex u is associated with a linked list **Adj[u]**, which contains all the vertices adjacent to u.

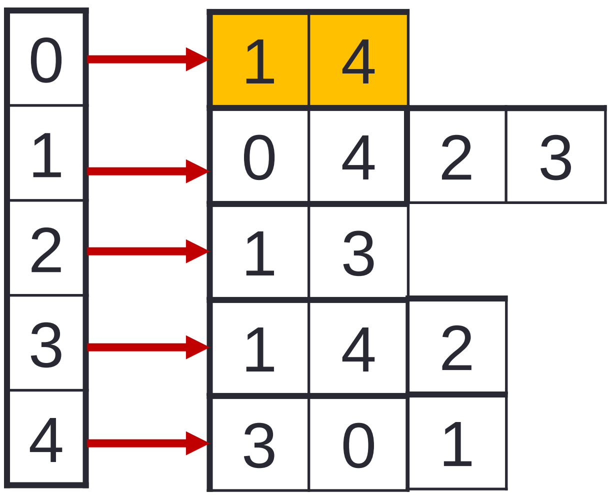




Adjacency List

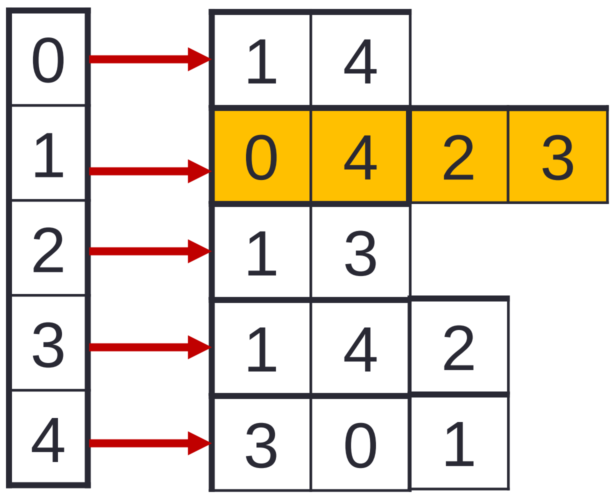
- Every vertex u is associated with a linked list **Adj[u]**, which contains all the vertices adjacent to u.

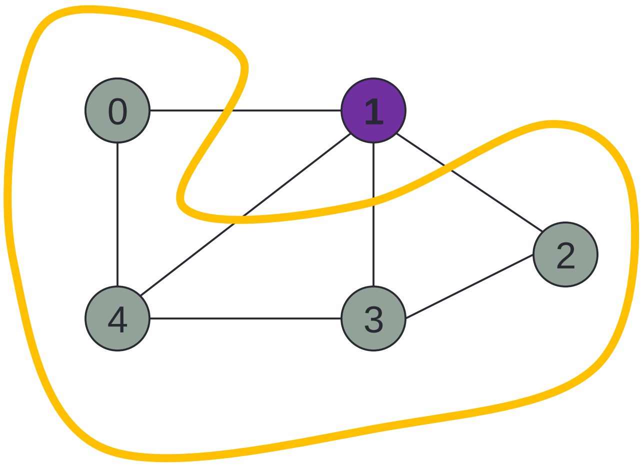




Adjacency List

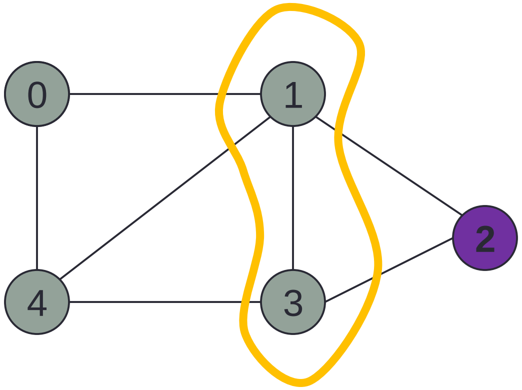
- Every vertex u is associated with a linked list **Adj[u]**, which contains all the vertices adjacent to u.

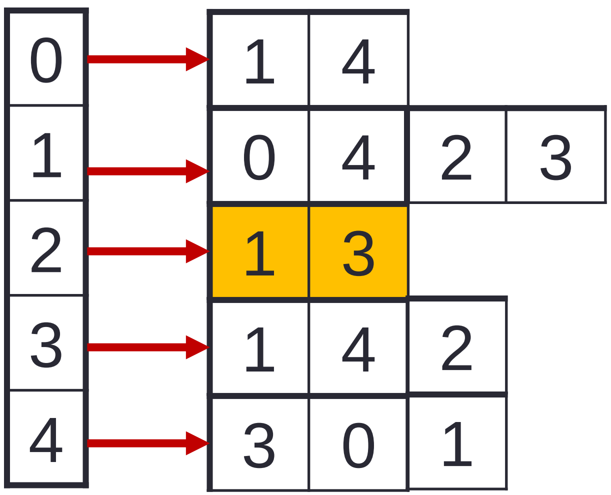




Adjacency List

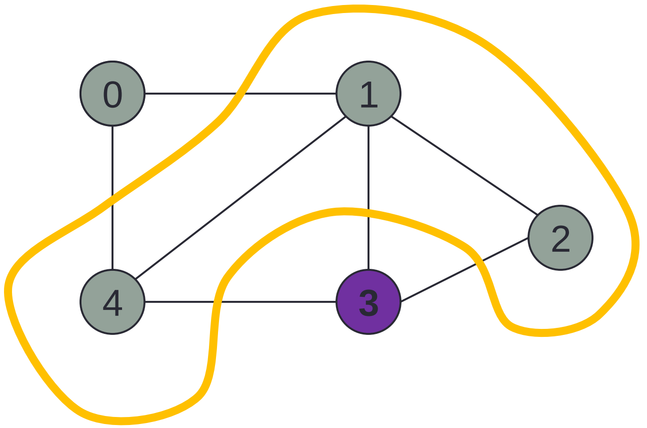
- Every vertex u is associated with a linked list **Adj[u]**, which contains all the vertices adjacent to u.

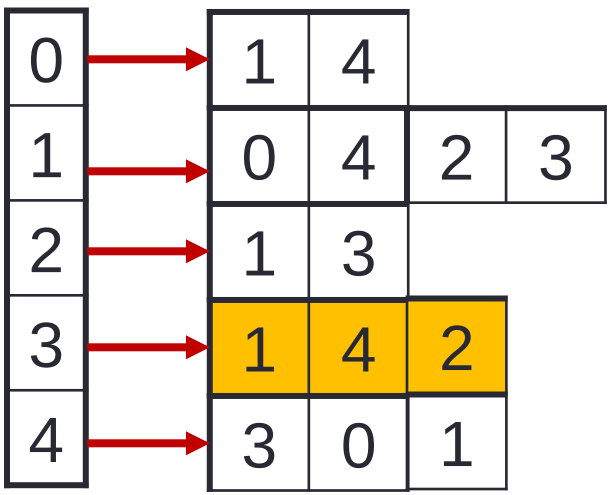




Adjacency List

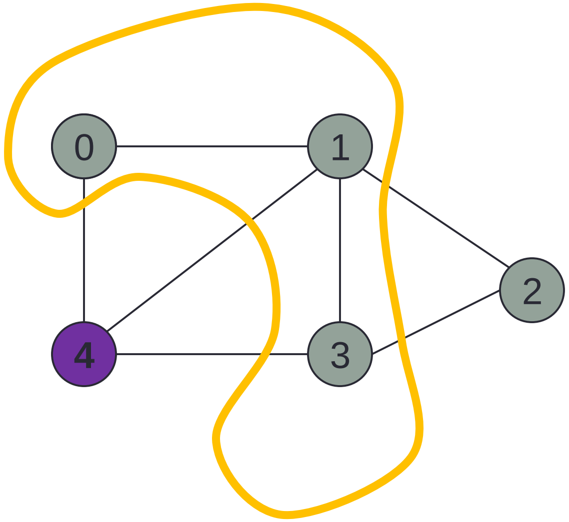
- Every vertex u is associated with a linked list **Adj[u]**, which contains all the vertices adjacent to u.

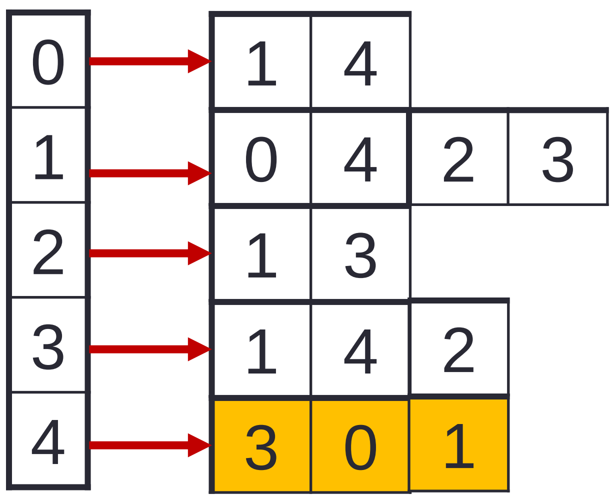




Adjacency List

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





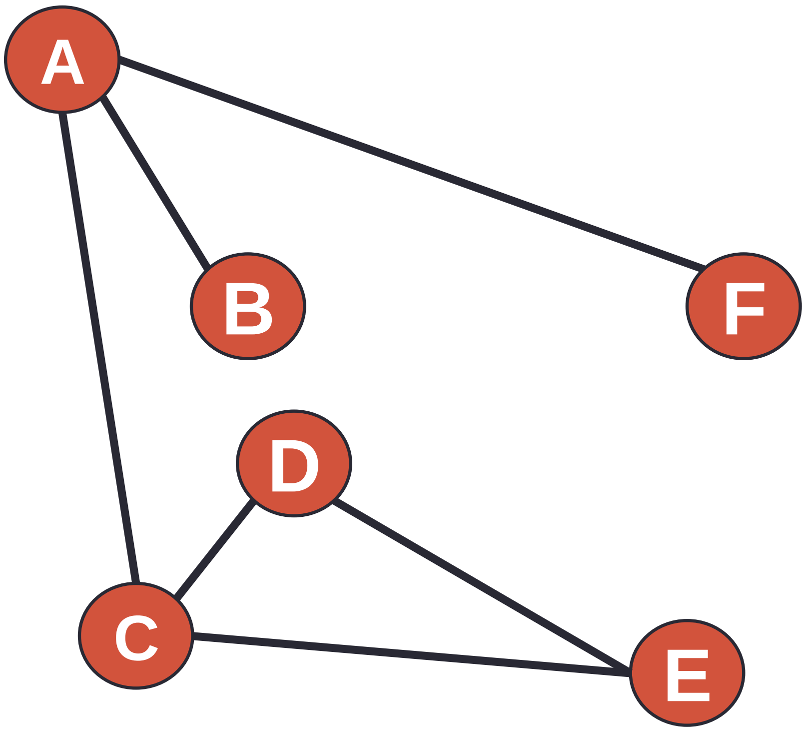
Adjacency List: Python code

```
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] and 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.

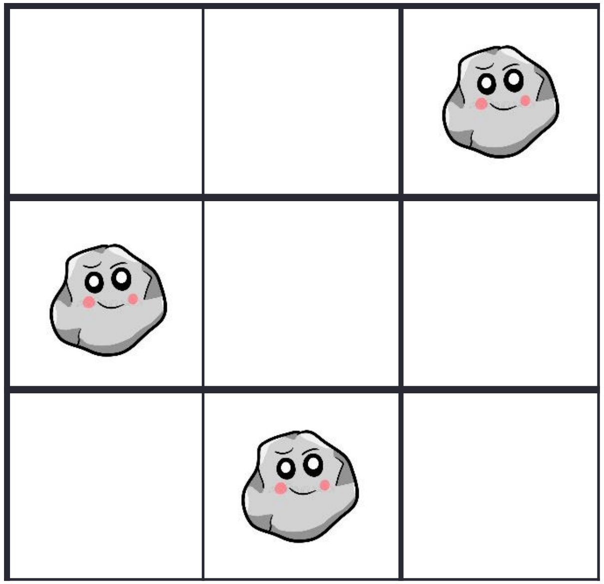
Quick exercise

1. Create the adjacency matrix and adjacency list for the following graph.



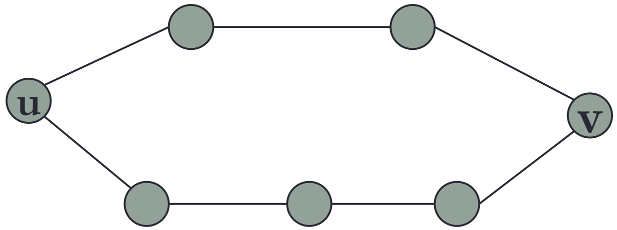
Quick exercise

2. Create the adjacency list for the vacant cells in the following grid.



Breadth-First Search (BFS)

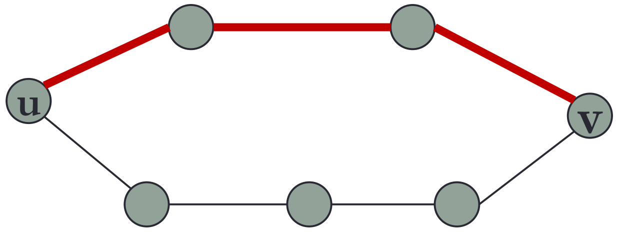
- **Breadth-first search (BFS)** is a simple algorithm for **searching** a graph.
- Given G=(**V**, **E**), and a distinguished source vertex **s**, BFS systematically explores the edges of G to
  - discover every vertex that is reachable from **s**,
  - compute nearest distance from **s** to each reachable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



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.

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 nearest distance from **s** to each reachable vertex,
  - produce a "breadth-first" tree with root **s** that contains all reachable vertices.

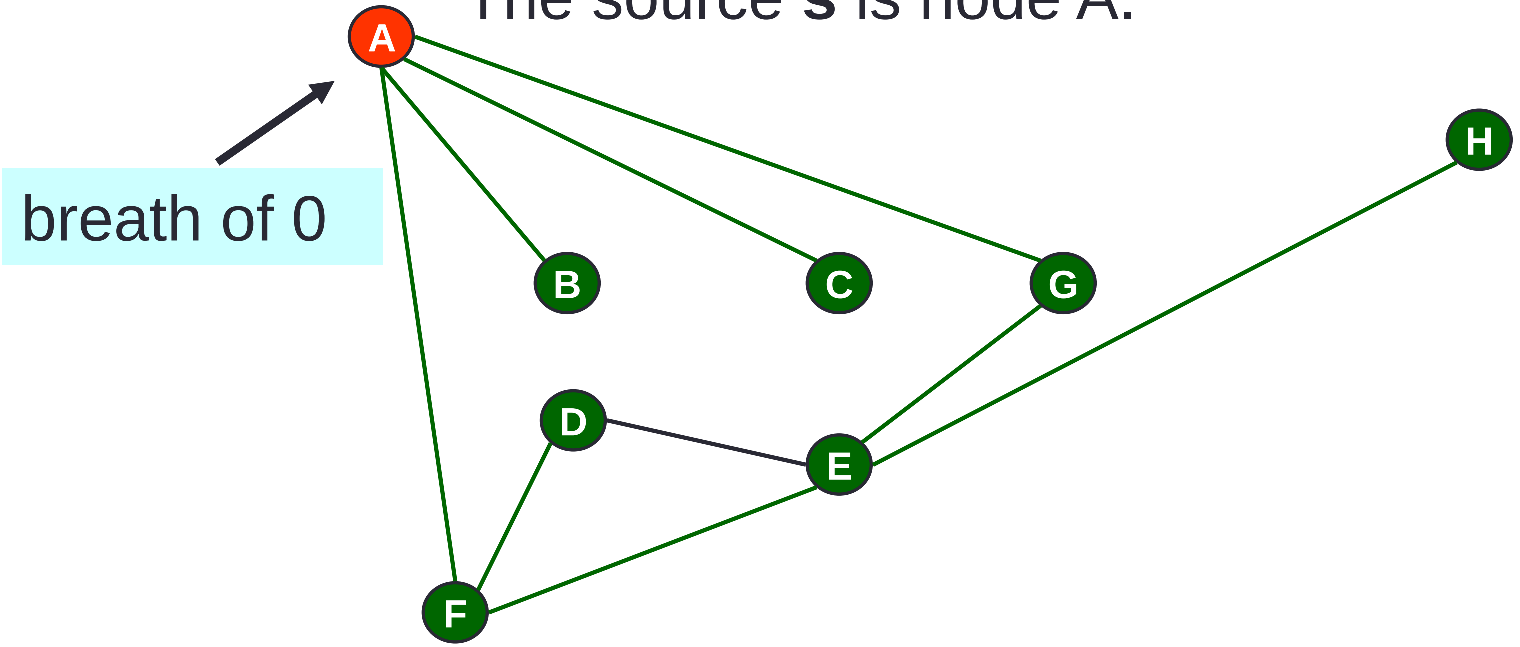


v is reachable from **u** because there is a sequence of consecutive edges from **u** to **v**.
The distance from **u** to **v** is 3.

BFS: Example (Step 1)

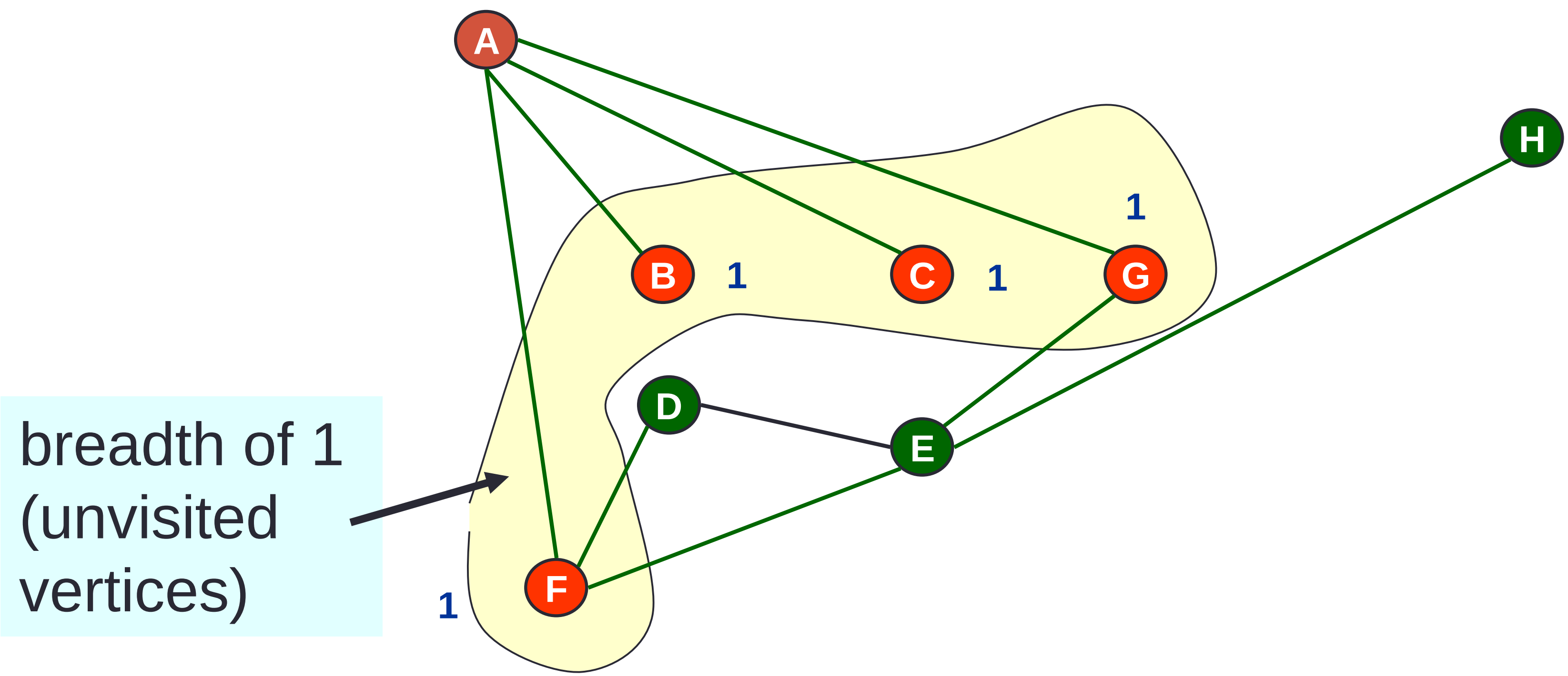
- From a source **s**, breadth-first search means:
  - ➢In each stage, we reach the vertices at the same distance k from **s** (k is a specific breadth).
  - ➢The algorithm reaches all vertices at distance k from **s** before reaching any vertices at distance k+1 (breadth: k -> k+1).

The source **s** is node A.



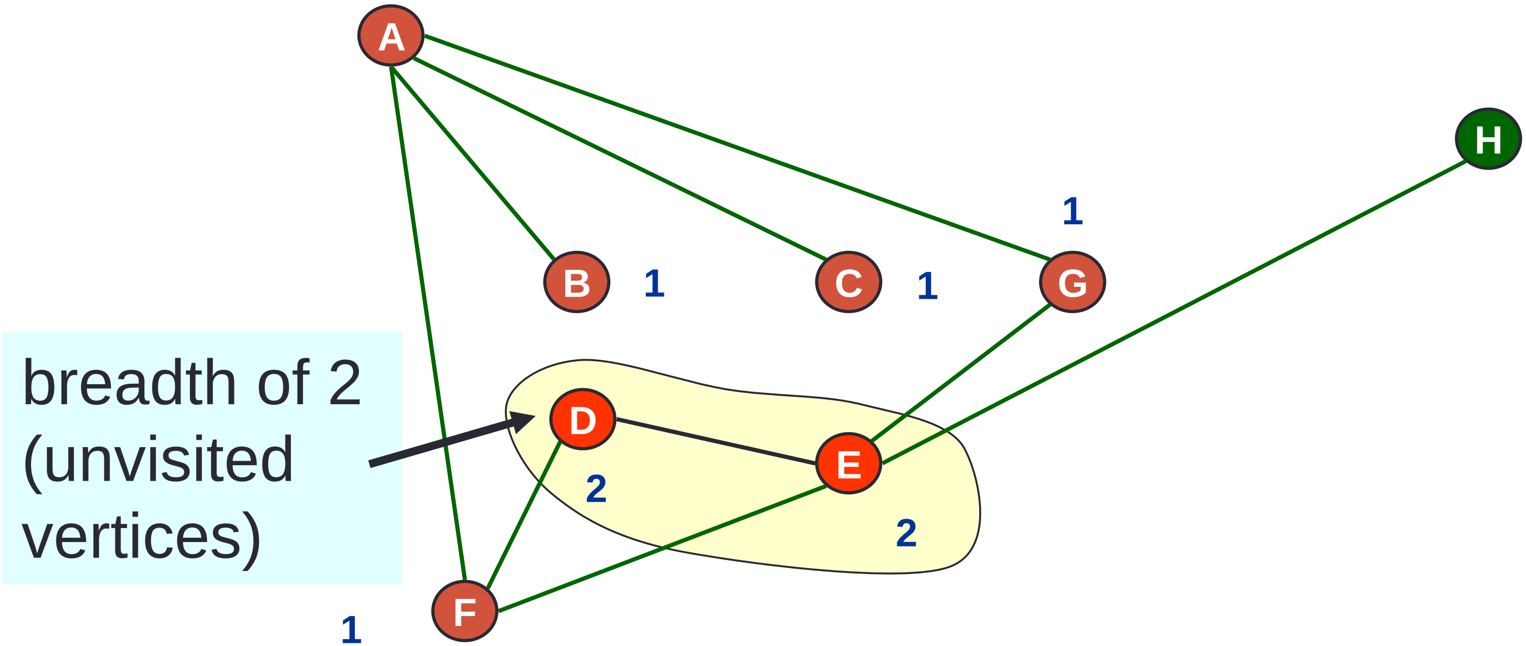
BFS: Example (Step 2)

- From a source **s**, breadth-first search means:
  - In each stage, we reach the vertices at the same distance k from **s** (k is a specific breadth).
  - The algorithm reaches all vertices at distance k from **s** before reaching any vertices at distance k+1 (breadth: k -> k+1).



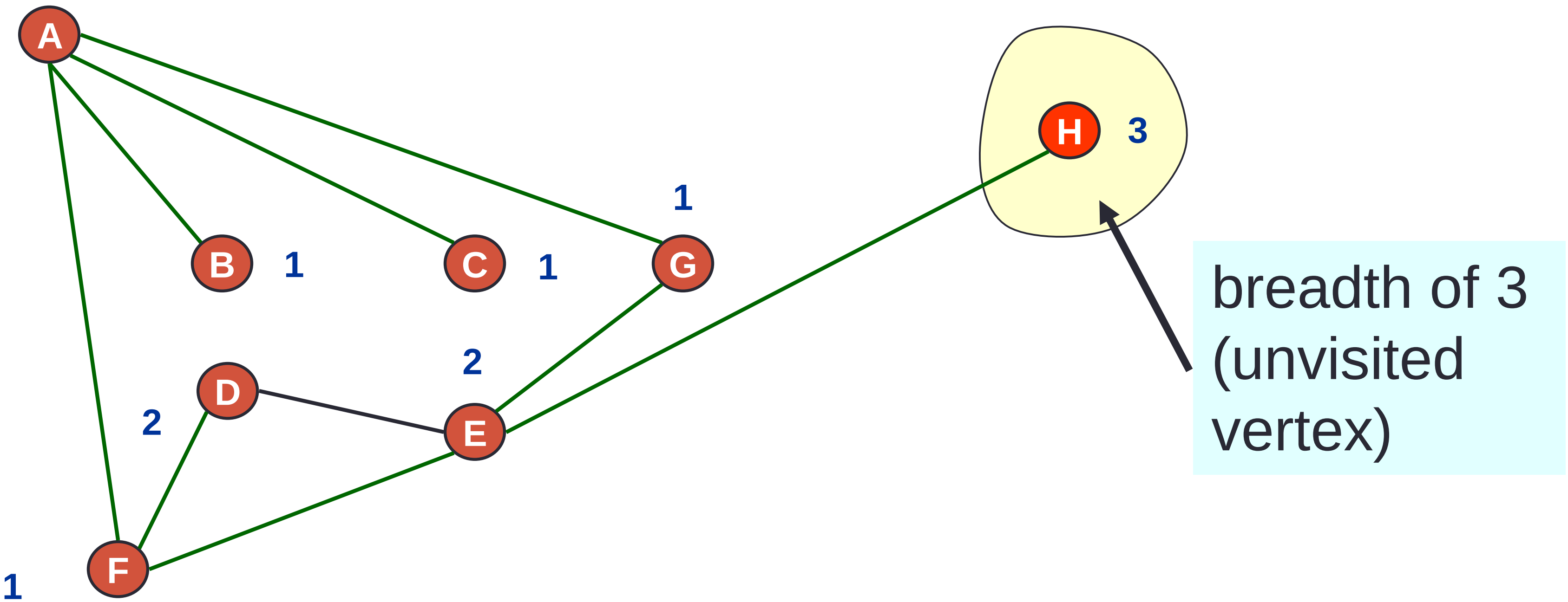
BFS: Example (Step 3)

- From a source **s**, breadth-first search means:
  - In each stage, we reach the vertices at the same distance k from **s** (k is a specific breadth).
  - The algorithm reaches all vertices at distance k from **s** before reaching any vertices at distance k+1 (breadth: k -> k+1).



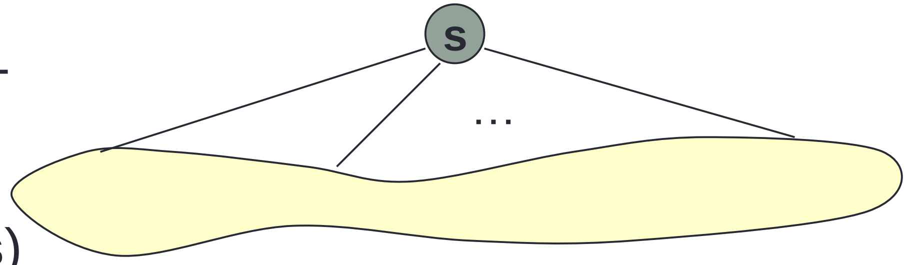
BFS: Example (Step 4)

- From a source **s**, breadth-first search means:
  - In each stage, we reach the vertices at the same distance k from **s** (k is a specific breadth).
  - The algorithm reaches all vertices at distance k from **s** before reaching any vertices at distance k+1 (breadth: k -> k+1).



Breadth-First Tree

Visit breadth-1 layer (explore the direct neighbors of **s**)

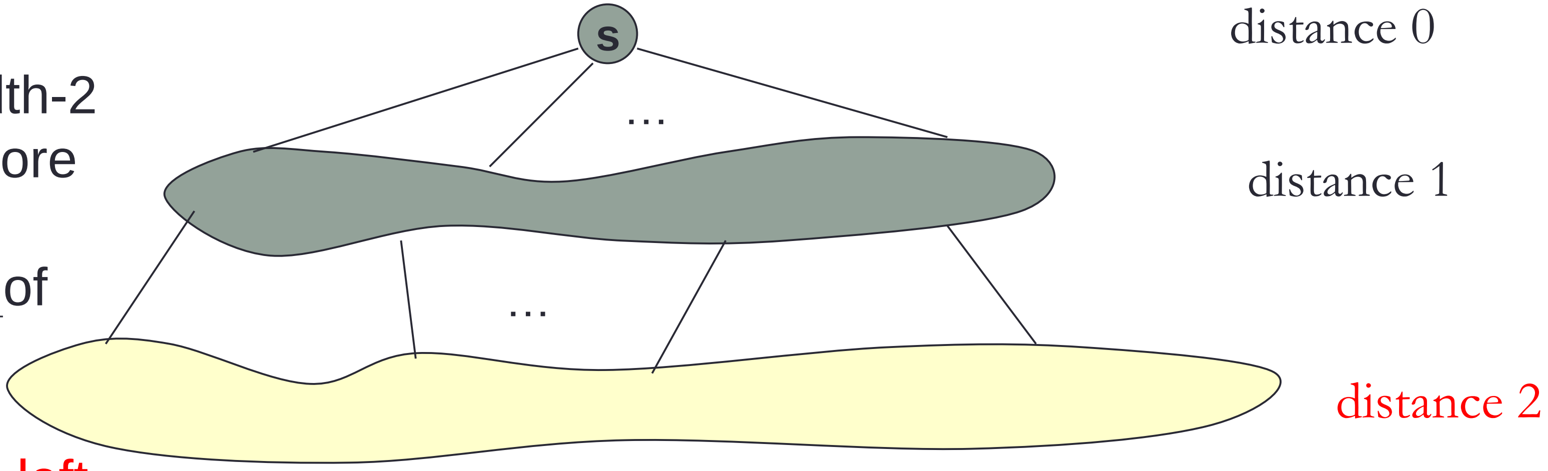


distance 0

distance 1

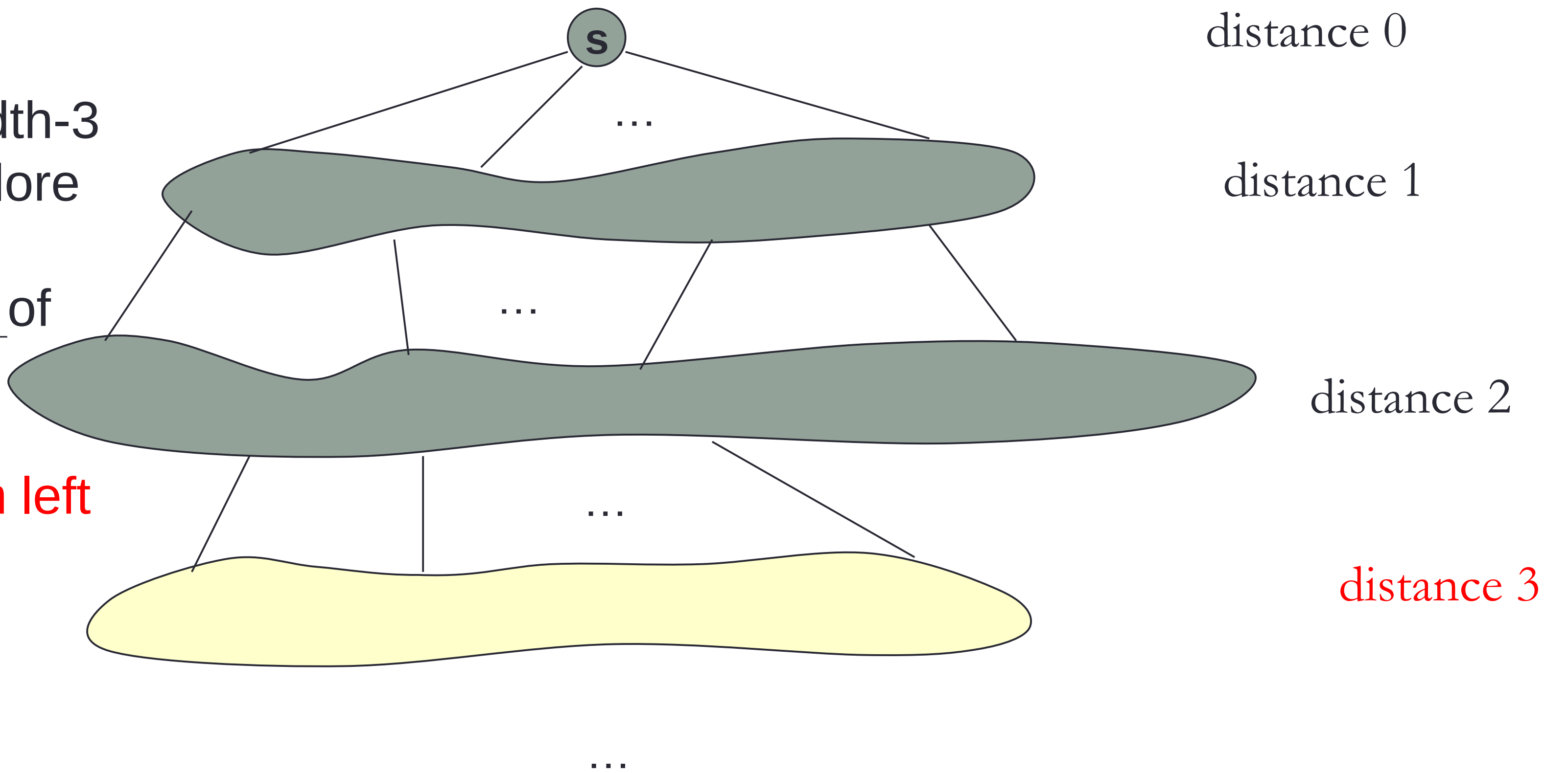
Breadth-First Tree

Visit breadth-2 layer (explore the direct neighbors of vertices in breadth-1 layer, from left to right)

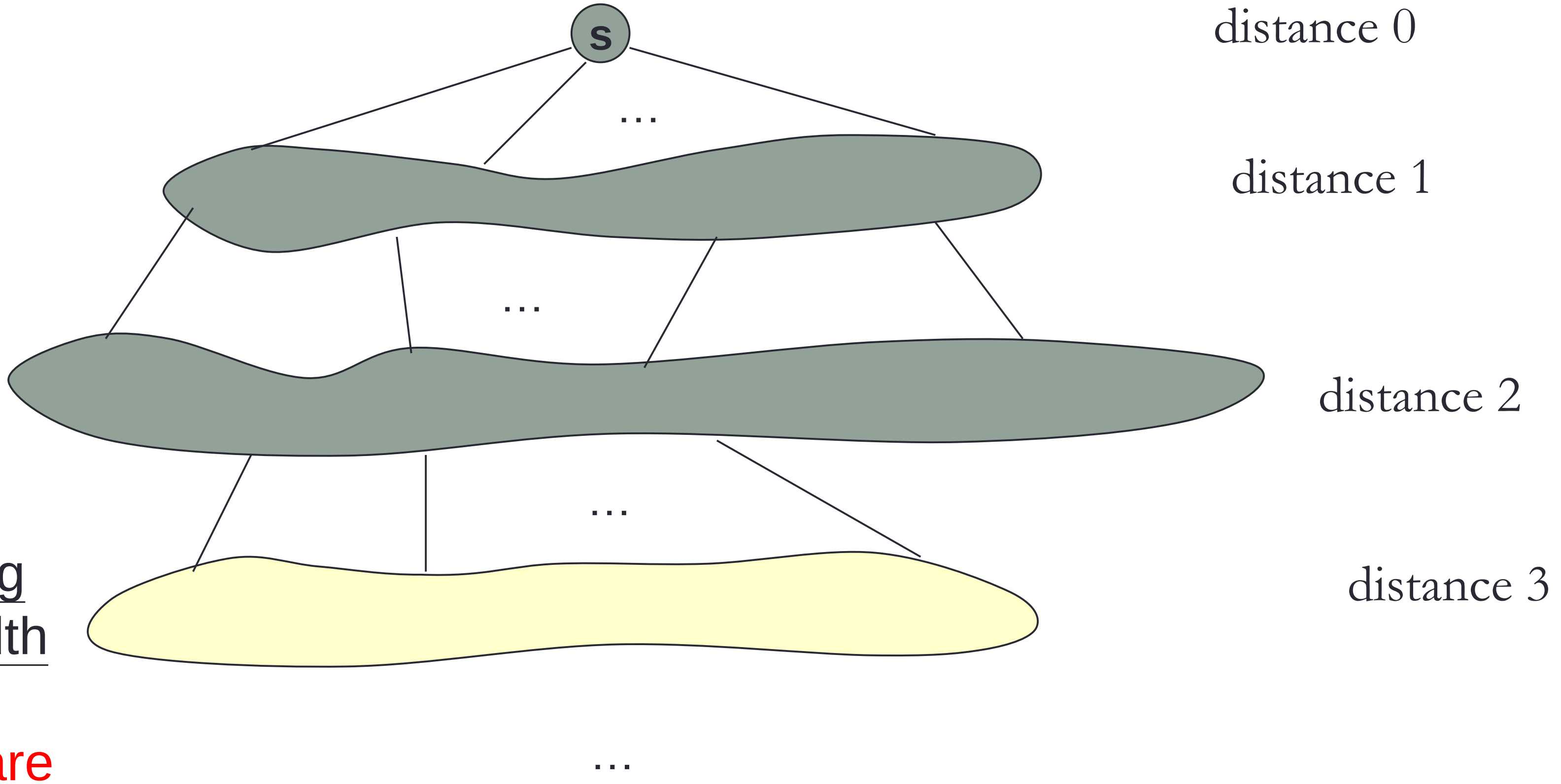


Breadth-First Tree

Visit breadth-3 layer (explore the direct neighbors of vertices in breadth-2 layer, from left to right)

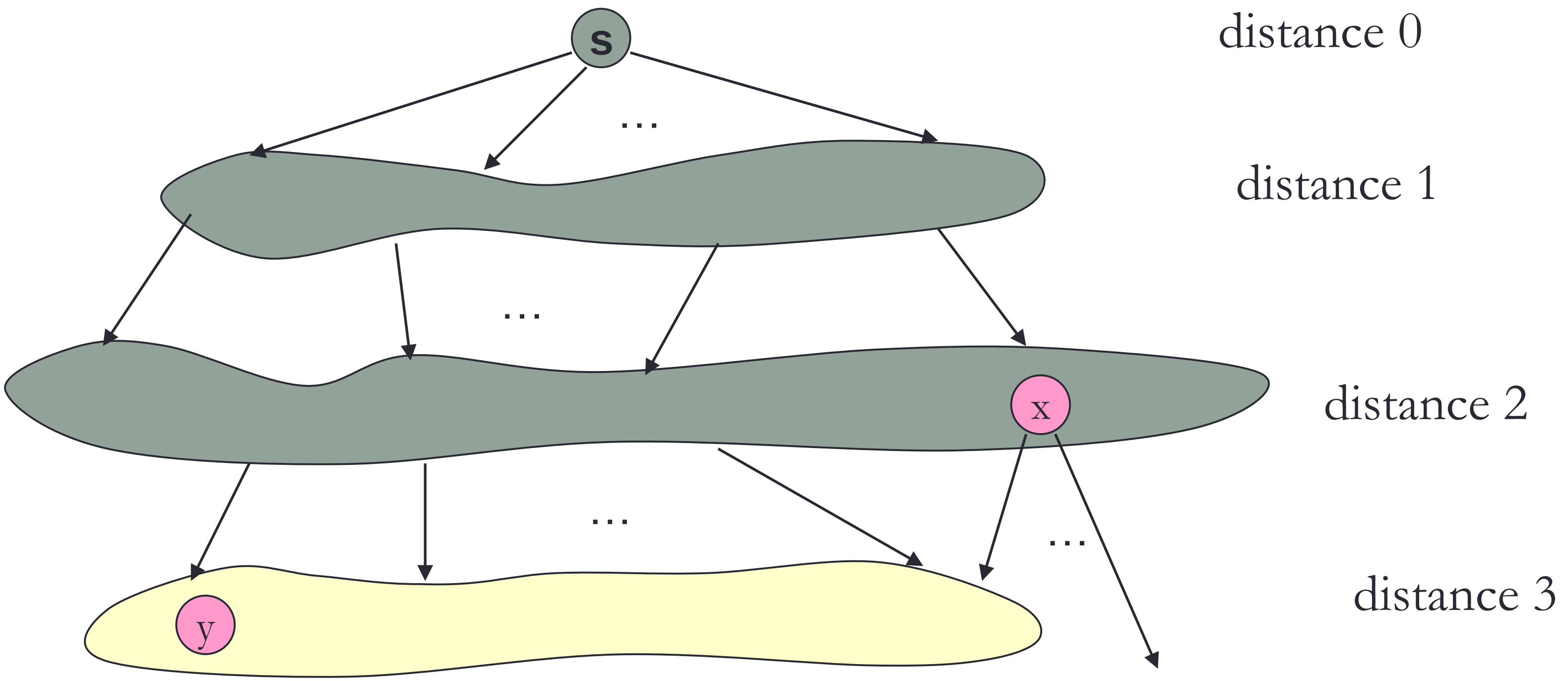


Breadth-First Tree



Keep expanding the breadth until all vertices are visited

Breadth-First Tree



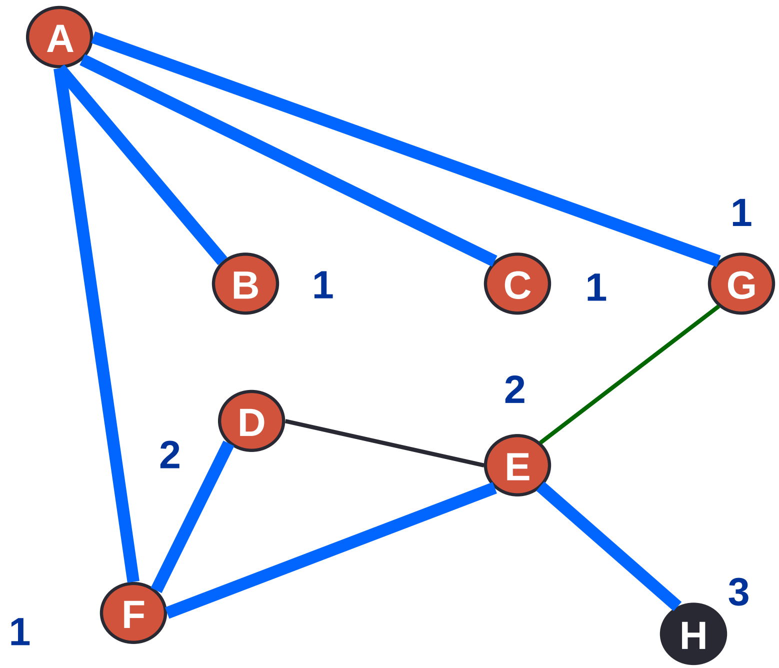
Note that because of the Breadth-first manner, if we visit vertex **x** before **y**, we explore **x**'s neighbors before **y**'s neighbors . Namely, if a vertex is first visited, then it is first served.

Linear Data structures

- **Array** – access an element with an index
- **Linked list** – elements are connected through a series of nodes
- **Stack** – last in first out
- **Queue** – first in first out

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 :



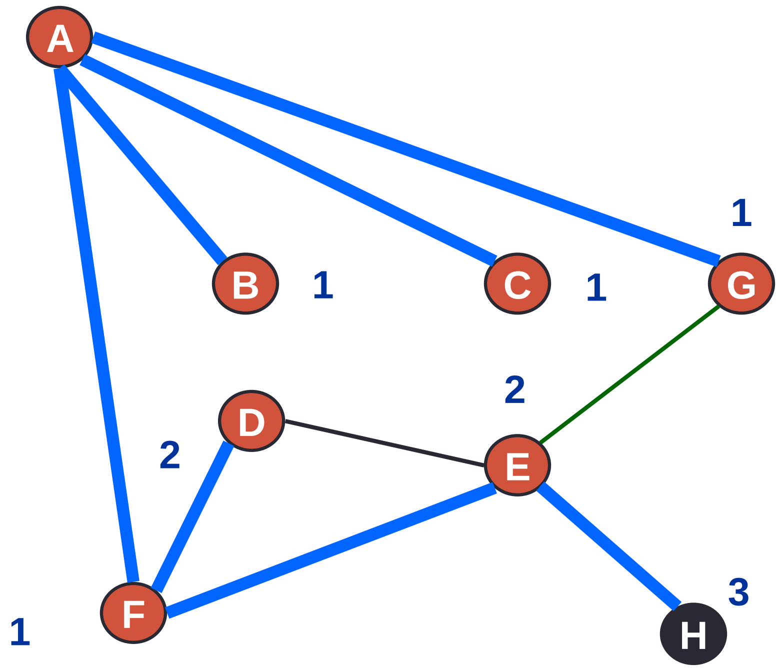
The **BF tree** produced.

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

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 :

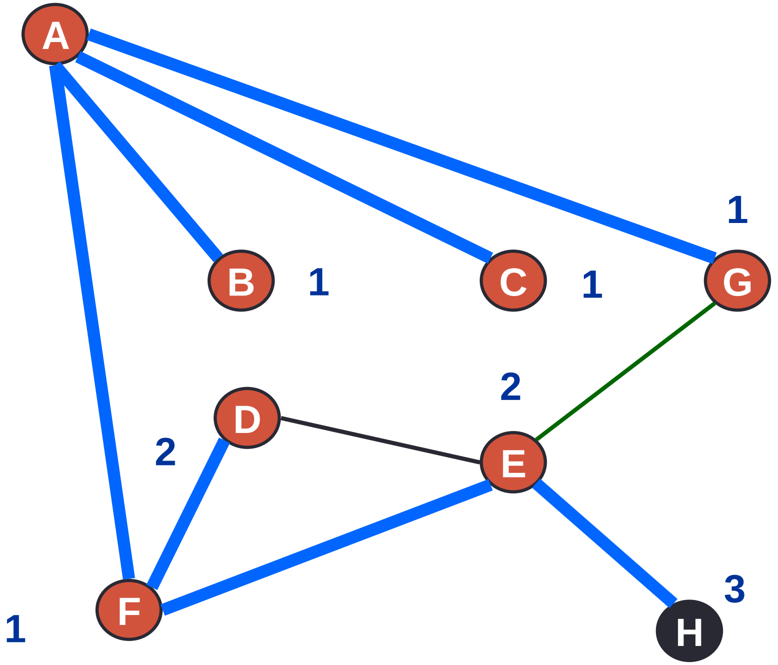
The **Queue** for storing unvisited vertices.



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 **Queue** for storing unvisited vertices.

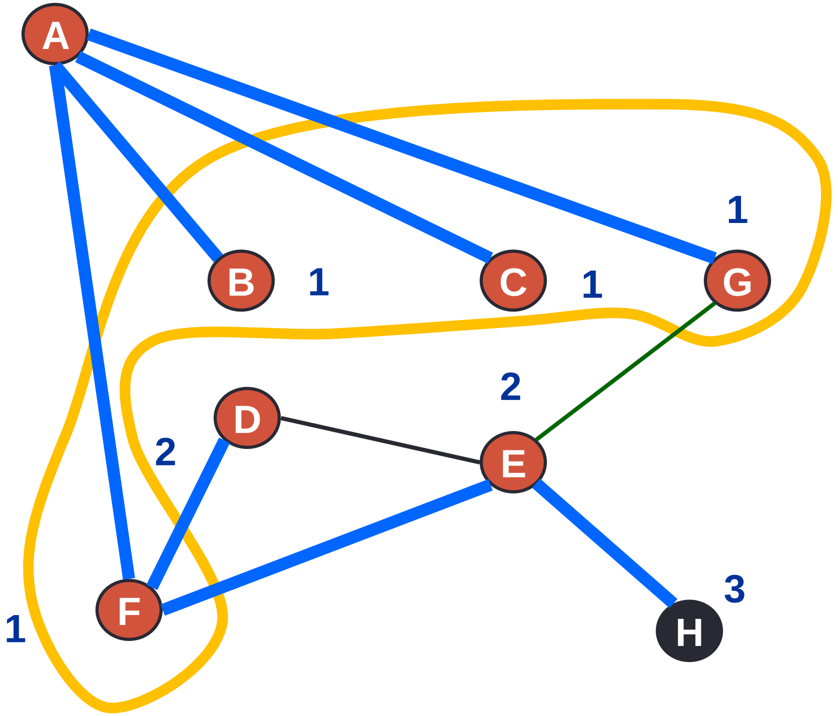


visited[A] = 1

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 **Queue** for storing unvisited vertices.



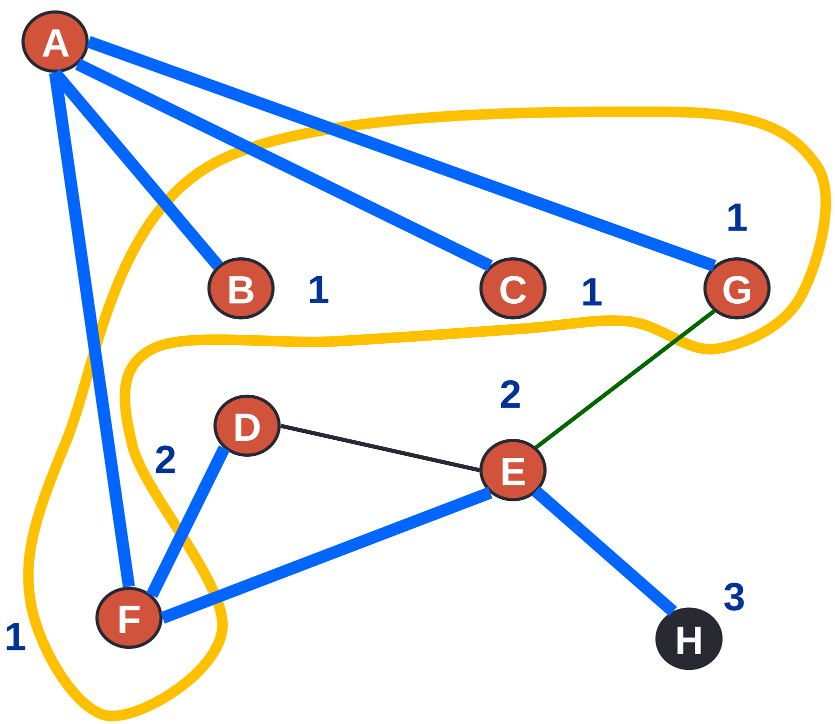
visited[A] = 1

A

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 **Queue** for storing unvisited vertices.

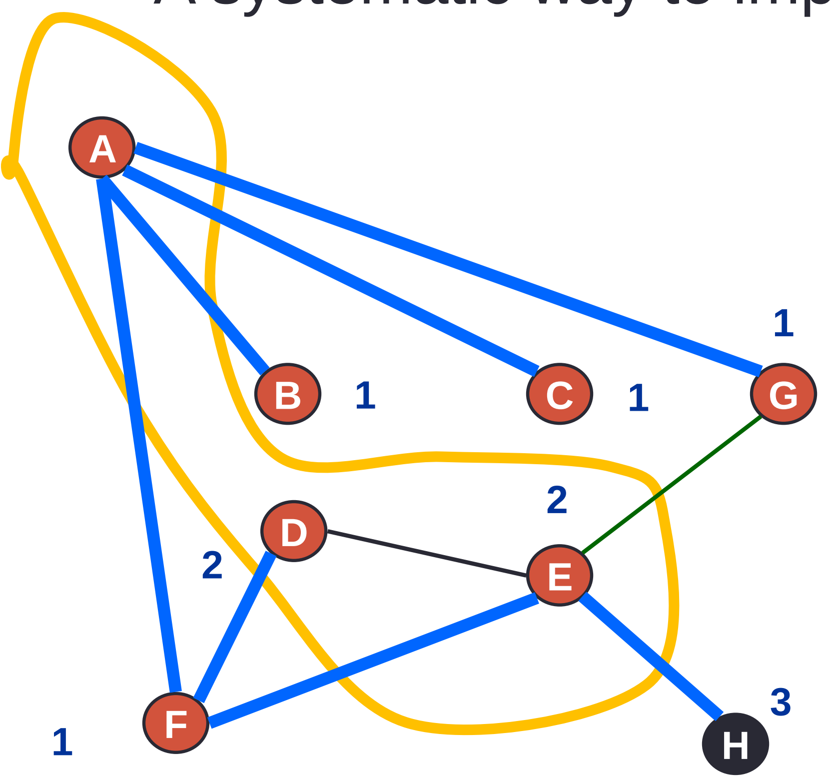


visited[G] = 1	G
visited[C] = 1	C
visited[B] = 1	B
visited[F] = 1	F
visited[A] = 1	A

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 **Queue** for storing unvisited vertices.

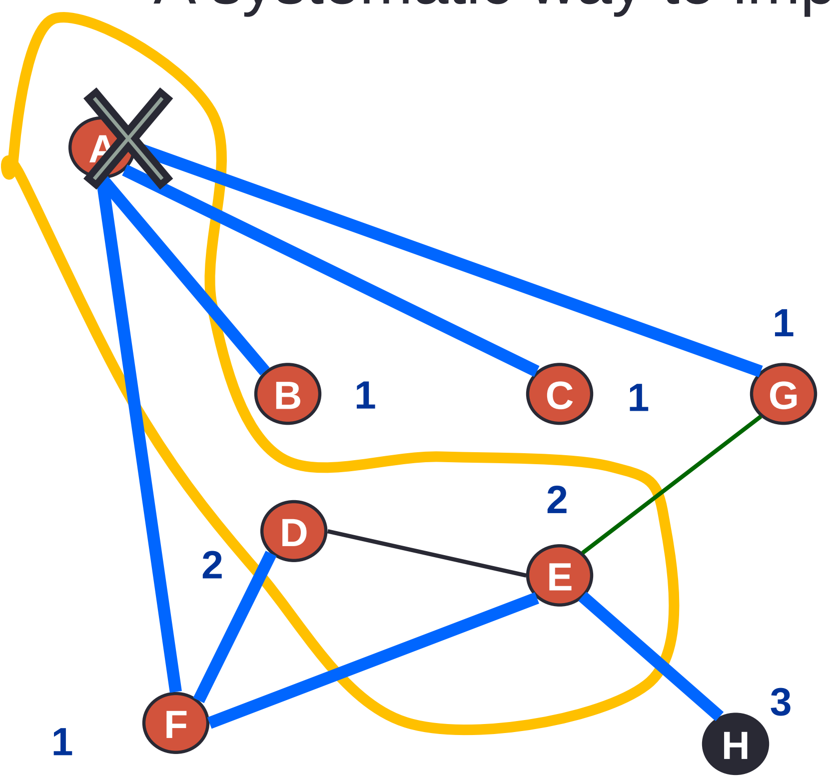


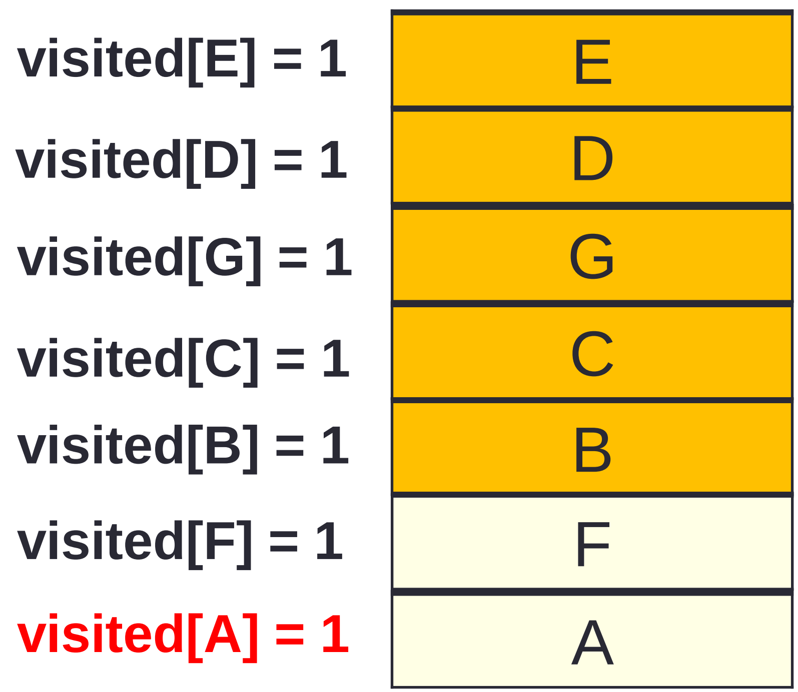
visited[G] = 1	G
visited[C] = 1	C
visited[B] = 1	B
visited[F] = 1	F
visited[A] = 1	A

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 **Queue** for storing unvisited vertices.

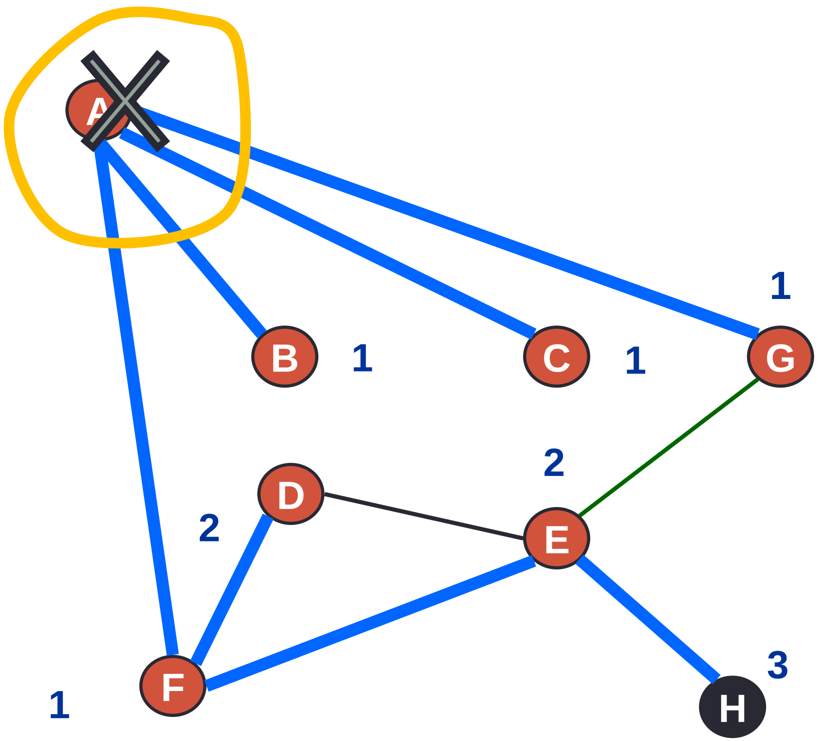




Breadth-First Tree: Example

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

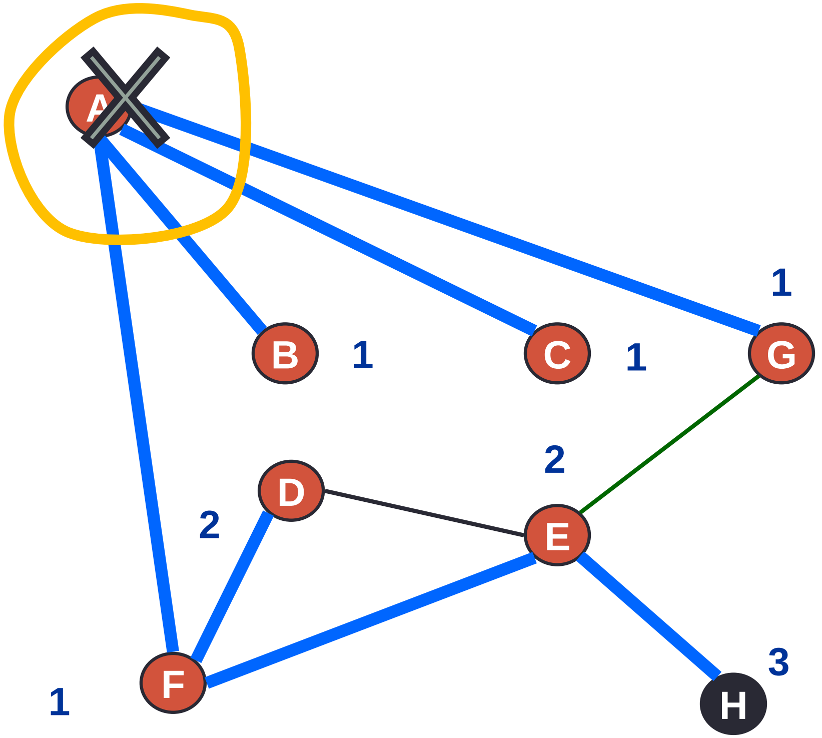
The **Queue** for storing unvisited vertices.



visited[E] = 1	E
visited[D] = 1	D
visited[G] = 1	G
visited[C] = 1	C
visited[B] = 1	B
visited[F] = 1	F
visited[A] = 1	A

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 :



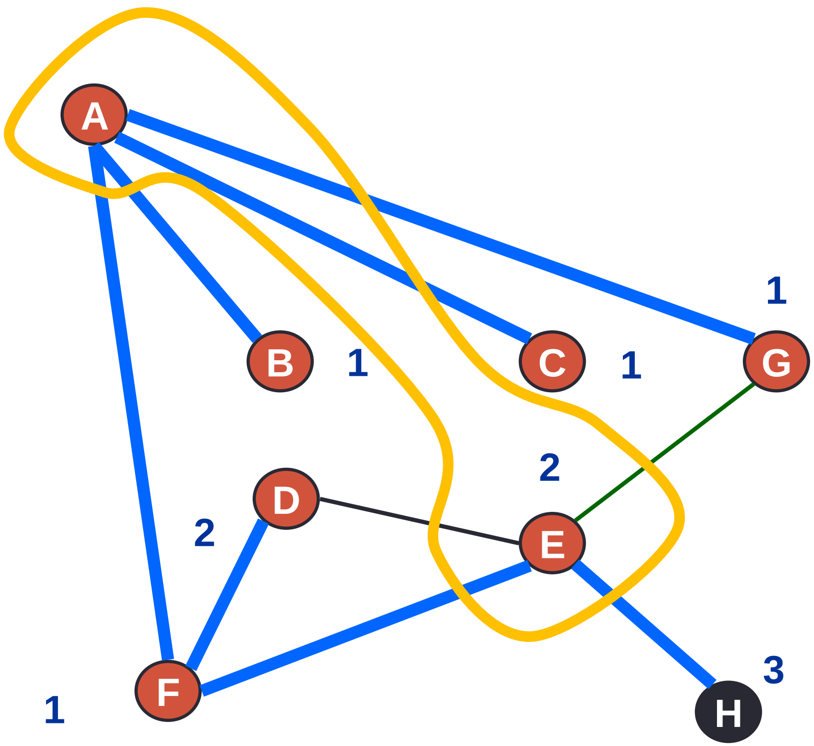
The **Queue** for storing unvisited vertices.

visited[E] = 1	E
visited[D] = 1	D
visited[G] = 1	G
visited[C] = 1	C
visited[B] = 1	B
visited[F] = 1	F
visited[A] = 1	A

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 **Queue** for storing unvisited vertices.

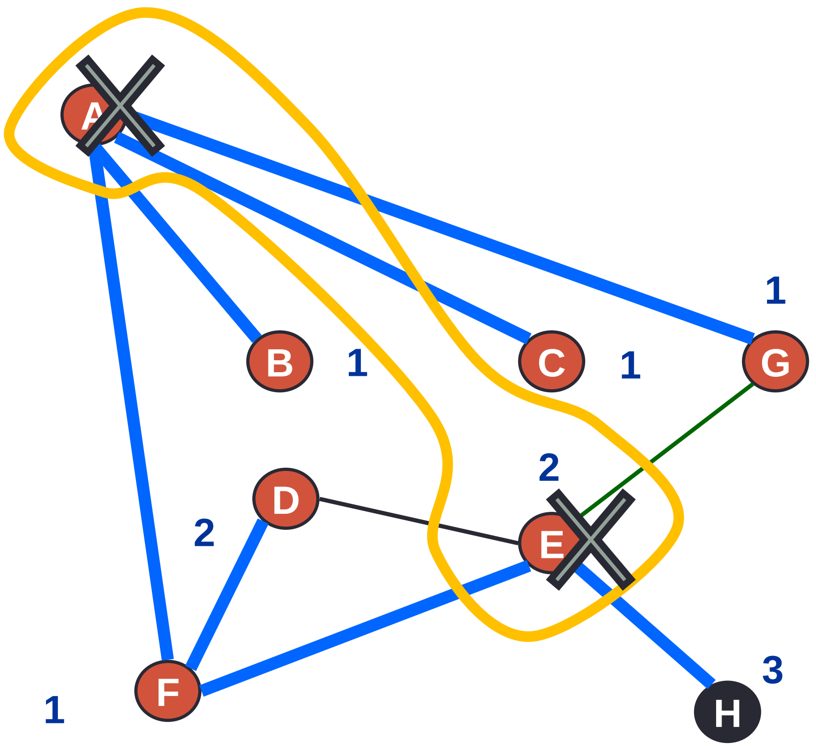


visited[E] = 1	E
visited[D] = 1	D
visited[G] = 1	G
visited[C] = 1	C
visited[B] = 1	B
visited[F] = 1	F
visited[A] = 1	A

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 **Queue** for storing unvisited vertices.

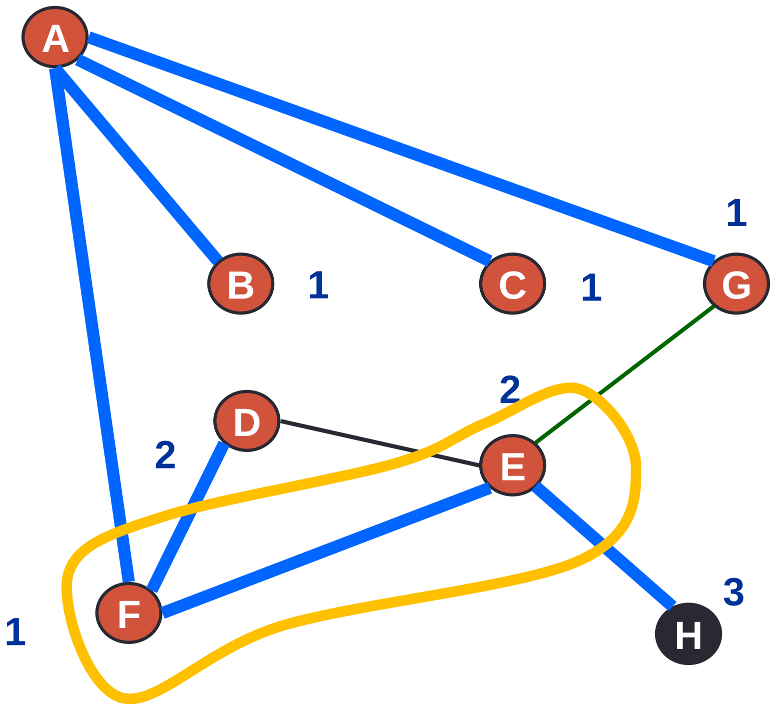


visited[E] = 1	E
visited[D] = 1	D
visited[G] = 1	G
visited[C] = 1	C
visited[B] = 1	B
visited[F] = 1	F
visited[A] = 1	A

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 **Queue** for storing unvisited vertices.



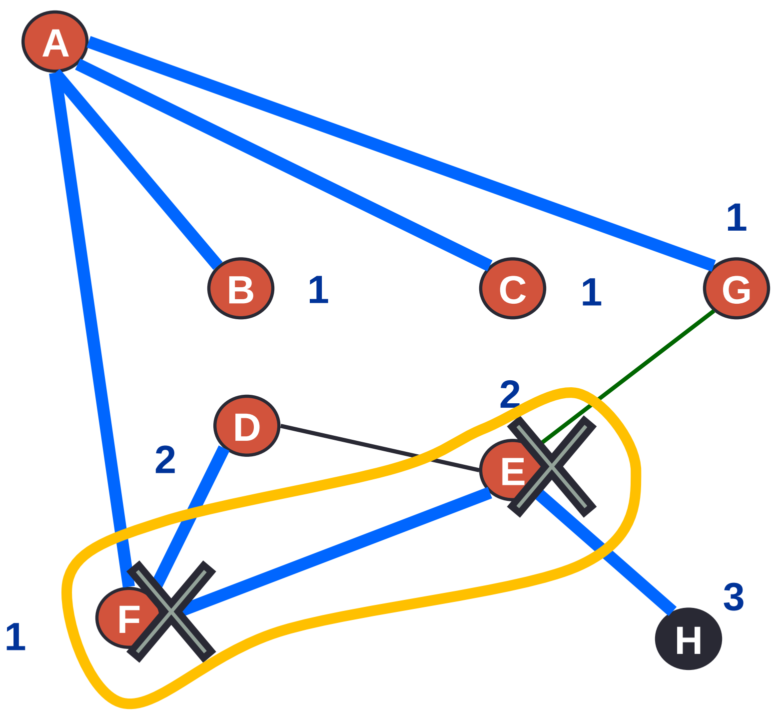
visited[E] = 1
visited[D] = 1
visited[G] = 1
visited[C] = 1
visited[B] = 1
visited[F] = 1
visited[A] = 1

E
D
G
C
B
F
A

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 **Queue** for storing unvisited vertices.

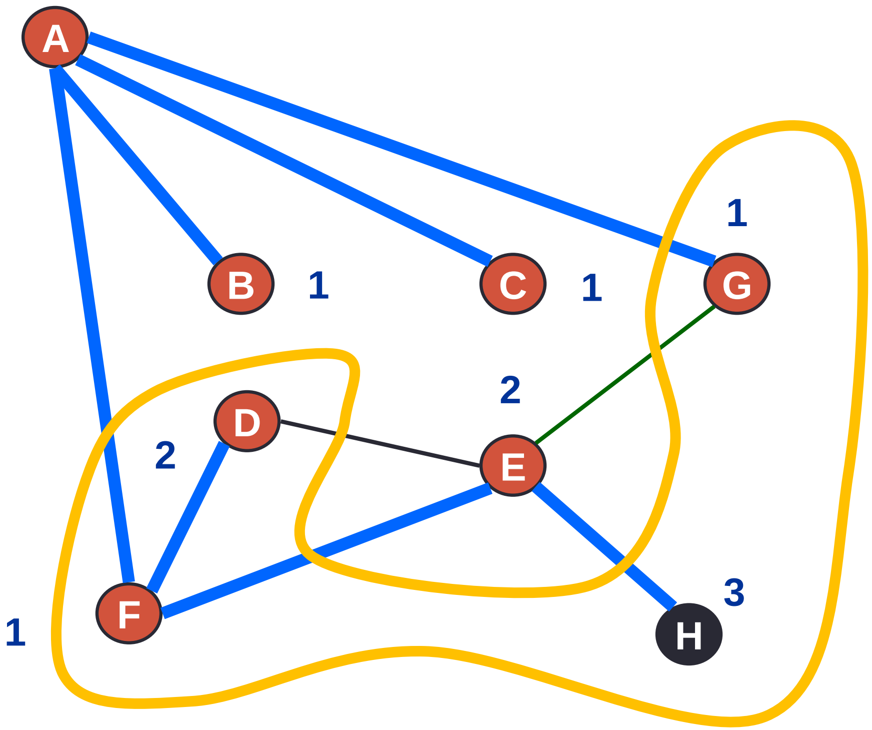


visited[E] = 1	E
visited[D] = 1	D
visited[G] = 1	G
visited[C] = 1	C
visited[B] = 1	B
visited[F] = 1	F
visited[A] = 1	A

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 **Queue** for storing unvisited vertices.

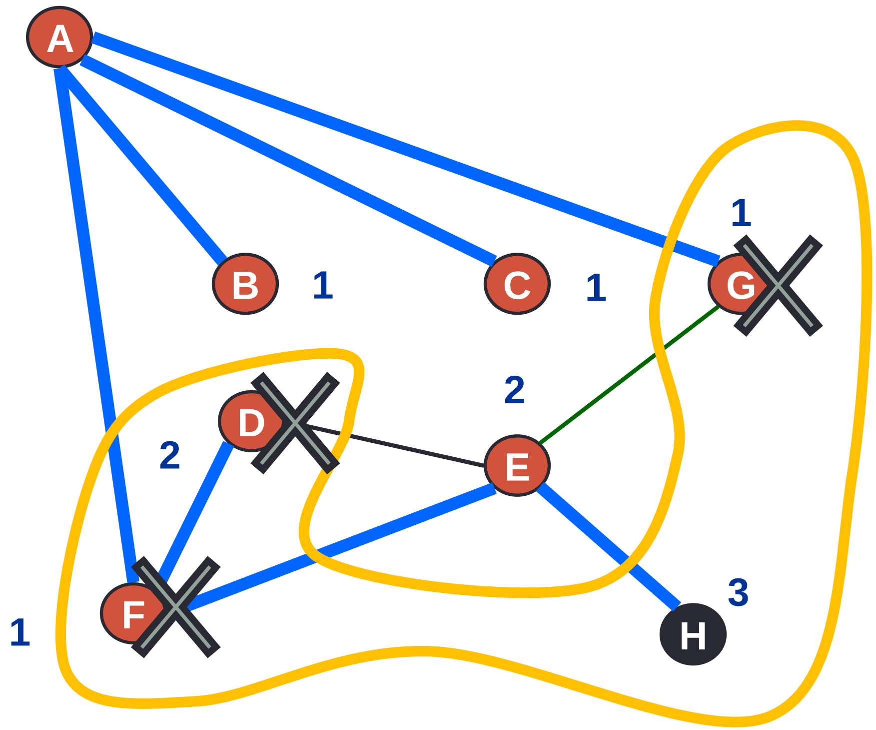


visited[E] = 1	E
visited[D] = 1	D
visited[G] = 1	G
visited[C] = 1	C
visited[B] = 1	B
visited[F] = 1	F
visited[A] = 1	A

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 **Queue** for storing unvisited vertices.

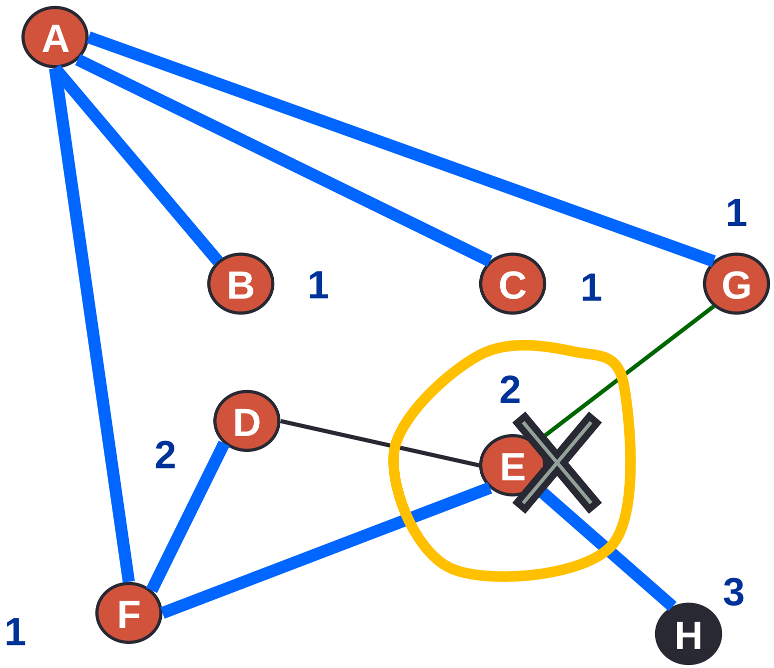


visited[H] = 1	H
visited[E] = 1	E
visited[D] = 1	D
visited[G] = 1	G
visited[C] = 1	C
visited[B] = 1	B
visited[F] = 1	F
visited[A] = 1	A

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 **Queue** for storing unvisited vertices.



visited[H] = 1	H
visited[E] = 1	E
visited[D] = 1	D
visited[G] = 1	G
visited[C] = 1	C
visited[B] = 1	B
visited[F] = 1	F
visited[A] = 1	A

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**

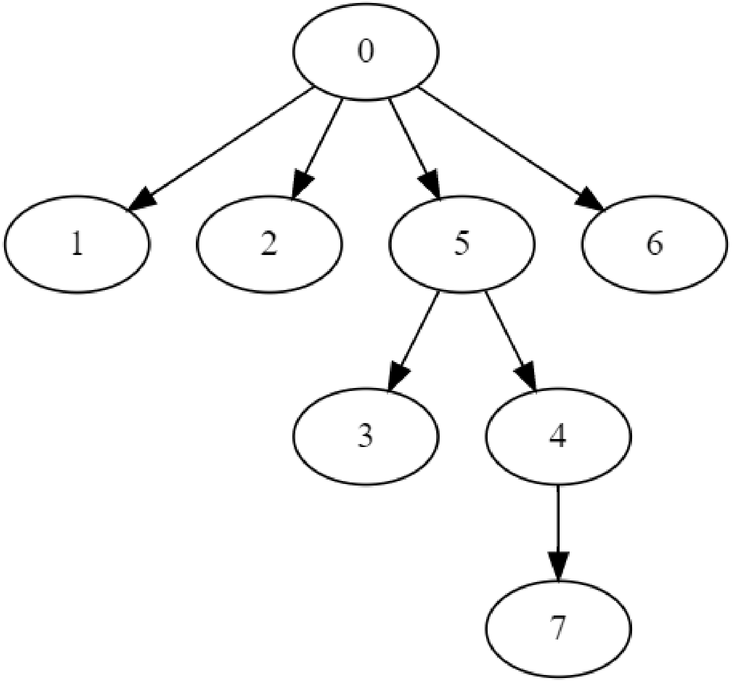
BFS on Adjacency List: Python code

- 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|)]

➢ Time complexity = **O(V+E)**

- **Correctness:** The number we find for vertex **u** is indeed the shortest 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 = **$O(V+V^2)$** = **$O(V^2)$**

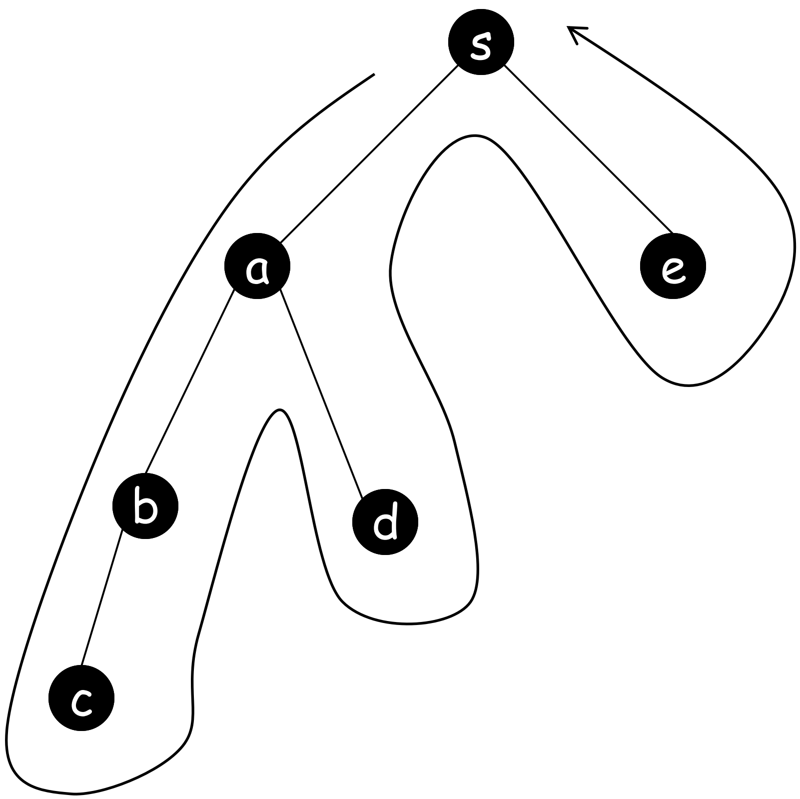
Depth-First Search (DFS)

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

- Keep exploring **v**'s descendants until we meet a leaf (no deeper to go);
- When all of **v**'s edges have been explored, the search "backtracks" to **v**'s parent to explore its other edges.

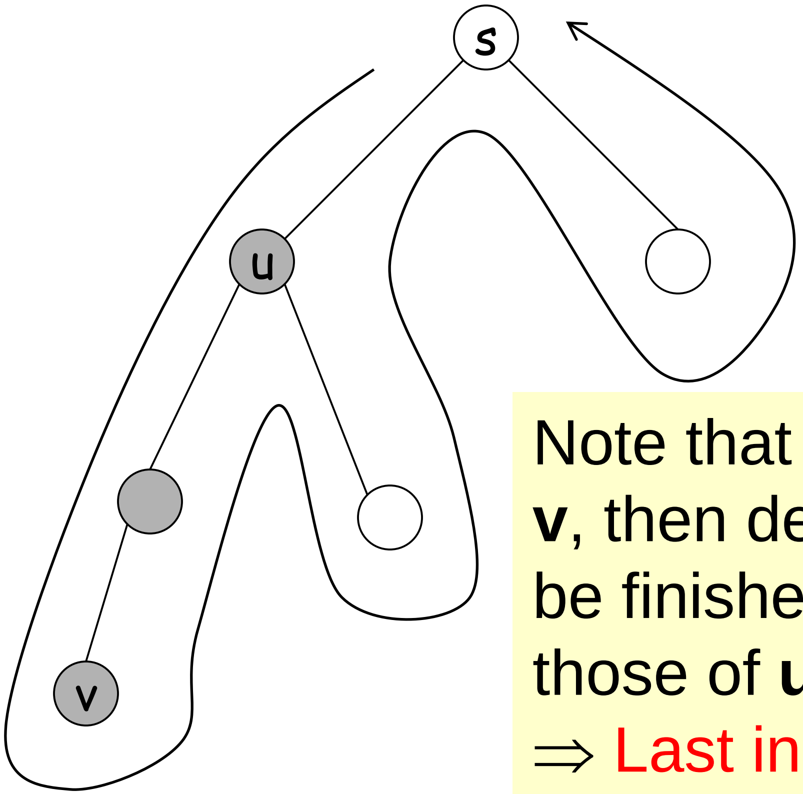
- **Note:** BFS visits all neighbors first; while DFS keeps going deeper until nowhere to go, and **backtrack**.

DFS of a Tree



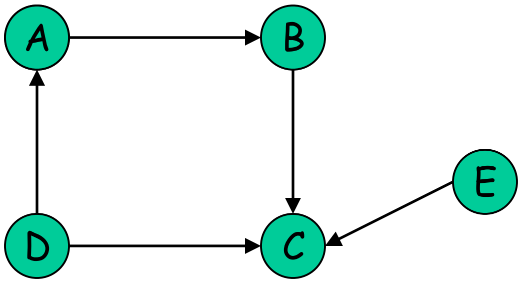
DFS of a Tree

- **Array** – access an element with an index
- **Linked list** – elements are connected through a series of nodes
- **Stack** – last in first out
- **Queue** – first in first out



Note that if **u** is visited before **v**, then descendants of **v** will be finished explored before those of **u**.
$\Rightarrow$ Last in first out.
Thus, we can use a **stack** to maintain the set of **visited-but-not-finished** vertices.

Directed Graph: Adjacency Matrix



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

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

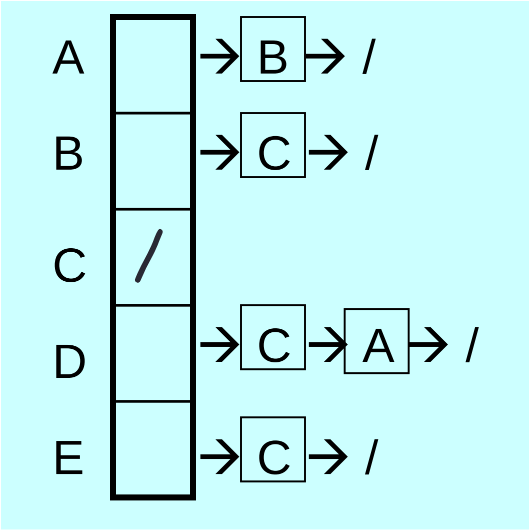
  E = {(A,B), (B,C), (E,C), (D,C), (D,A)}
- (A,B) means the edge is from vertex A to vertex B.
- **The Adjacency matrix**

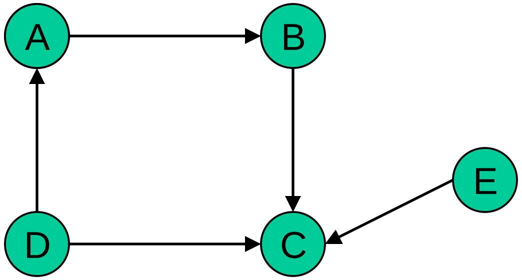
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	0	0
C	0	0	0	0	0
D	1	0	1	0	0
E	0	0	1	0	0

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

Directed Graph: Adjacency List

The Adjacency list

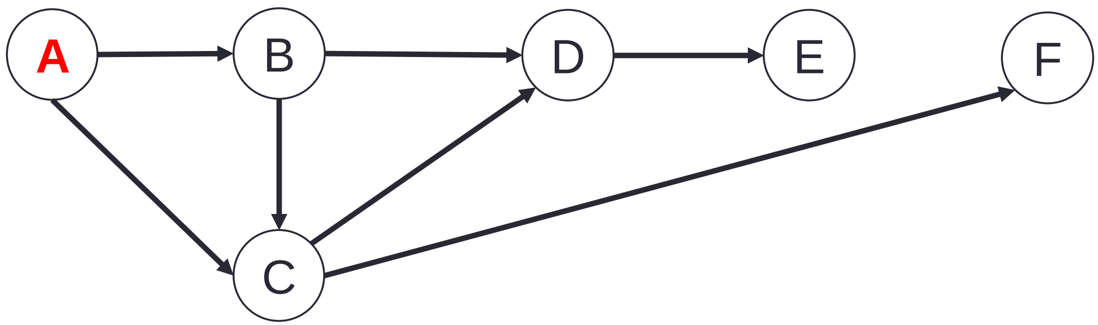




BFS on **<u>directed graph</u>** is very similar to BFS on **<u>undirected graph</u>**, except that we must follow the direction of an edge in order to traverse it.

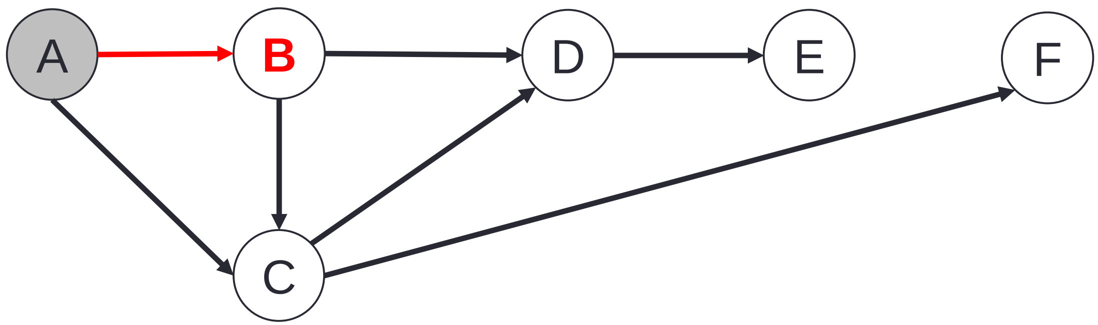
DFS on Directed Graph: Example (Step 1)

- Suppose we start DFS from vertex A.



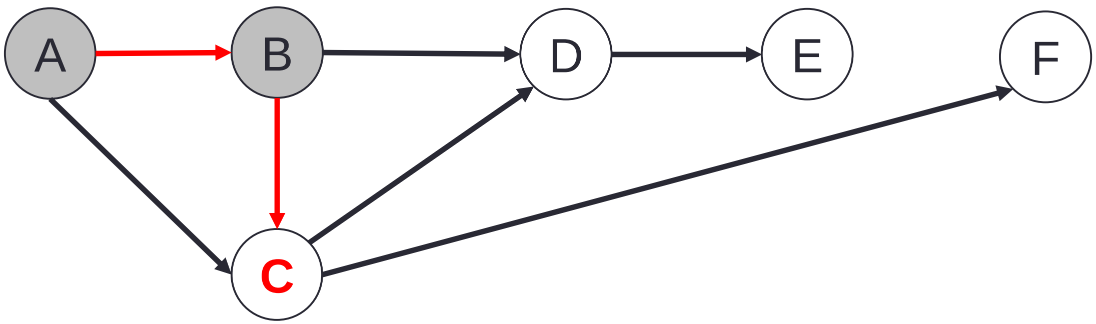
DFS on Directed Graph: Example (Step 2)

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



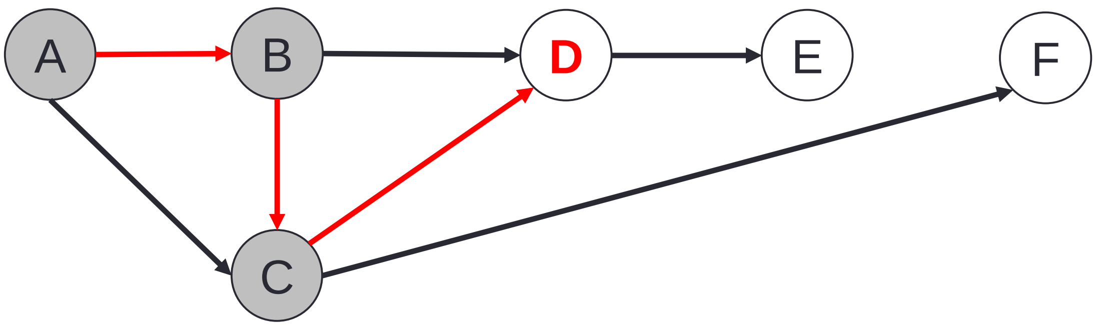
DFS on Directed Graph: Example (Step 3)

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



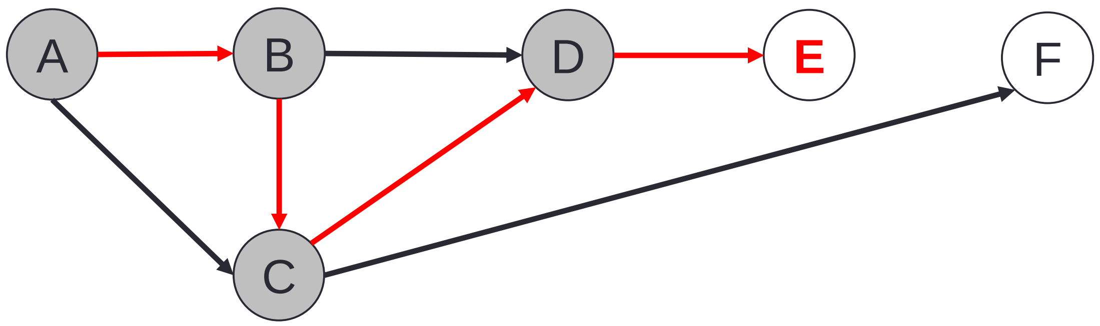
DFS on Directed Graph: Example (Step 4)

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



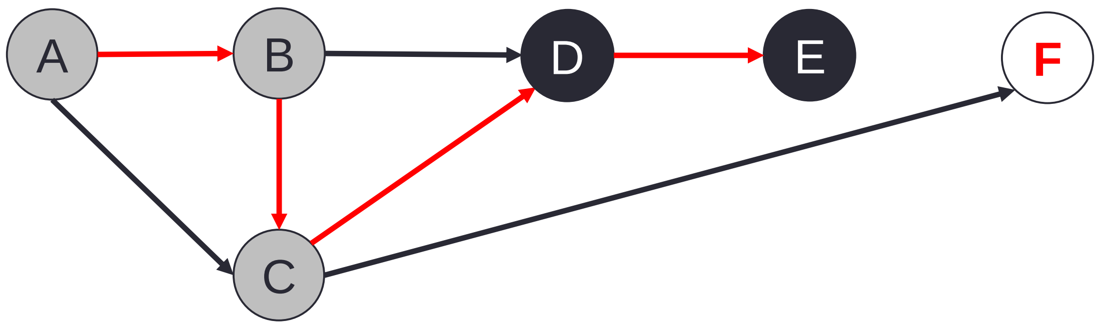
DFS on Directed Graph: Example (Step 5)

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



DFS on Directed Graph: Example (Step 6)

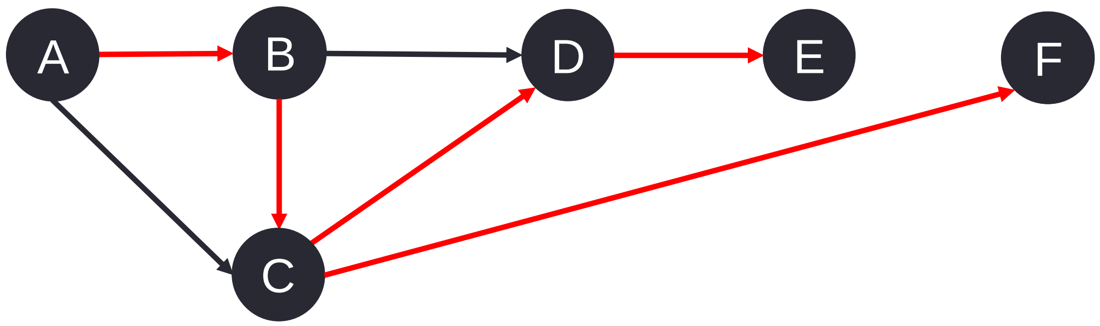
- We mark E as visited.
- E does not have any unvisited neighbor, so it becomes finished.



- 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 on Directed Graph: 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):
    mark x as visited
    print vertex x

    for each neighbor y of x:
        if vertex y is not visited:
            DFS(y)
```

- **Time complexity**

➢ Each vertex is visited once $\Rightarrow$ **O(V)**

➢ All edges are explored at most twice (discover/backtrack) $\Rightarrow$ **O(E)**

➢ Time complexity = **O(V+E)**

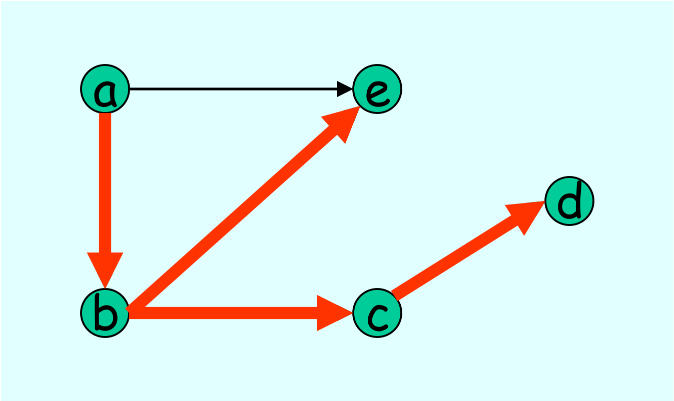
DFS on Adjacency List: Python code

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

    for y in self.graph[x]:
       if visited[y] == False: #explore unvisited neighbors
         self.rdfs(y, visited)
```

Depth-First Tree

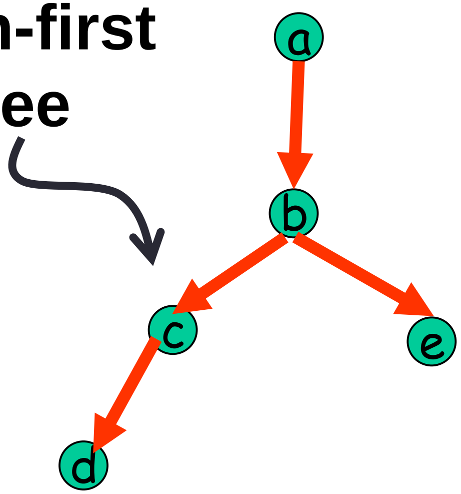
- What do we get after a depth-first search on a directed graph?
- **Answer:** A depth-first tree.



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

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

Depth-first Tree



- Then the tree $G_\pi(V, E_\pi)$ is

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

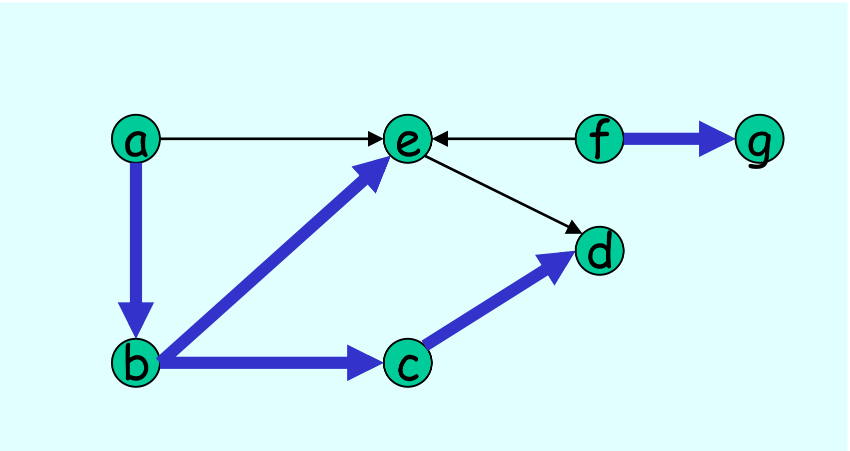
$E_\pi = \{(\pi(b),b), (\pi(c), c), (\pi(d),d), (\pi(e),e)\}$

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

Depth-First Tree

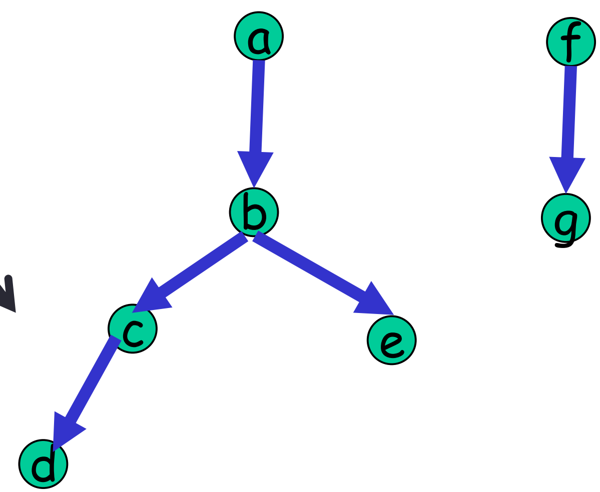
- In general, we get a depth-first forest

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)

Depth-first Forest



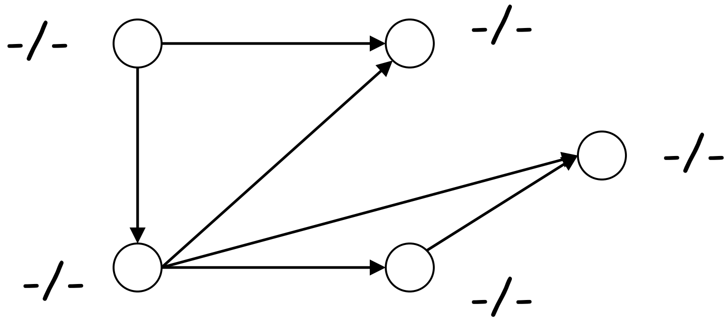
Timestamps

- 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 exploring **v**'s adjacency list, i.e., when **v**'s color is changed from **gray to black**.

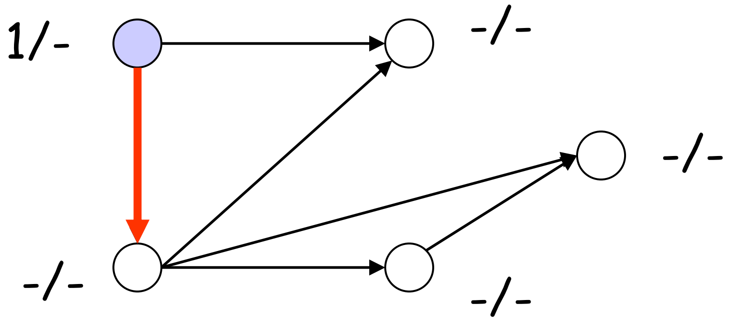
Timestamps: Example

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

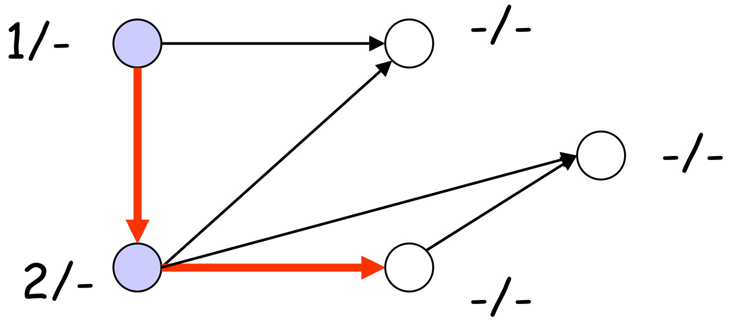


Timestamps: Example (Time 1)

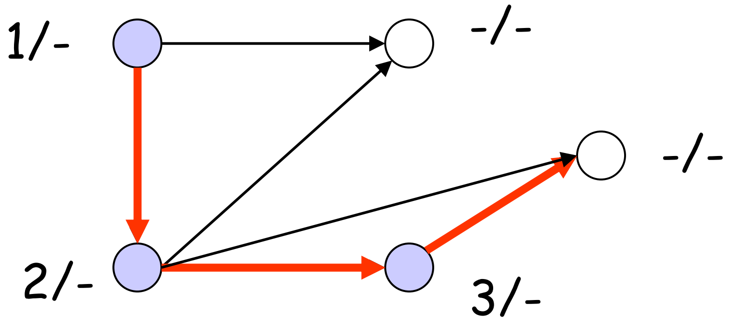
- At time 1, the source is discovered, and its color is changed to gray.



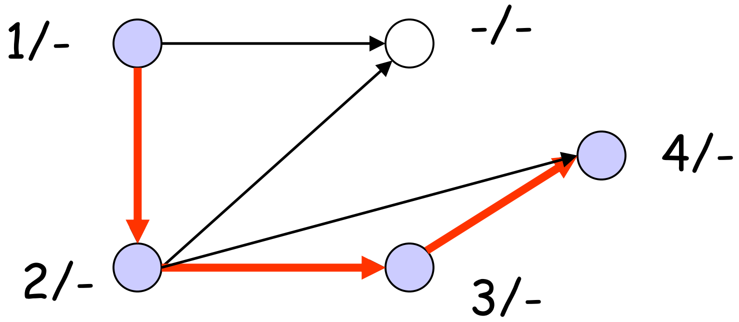
Timestamps: Example (Time 2)



Timestamps: Example (Time 3)

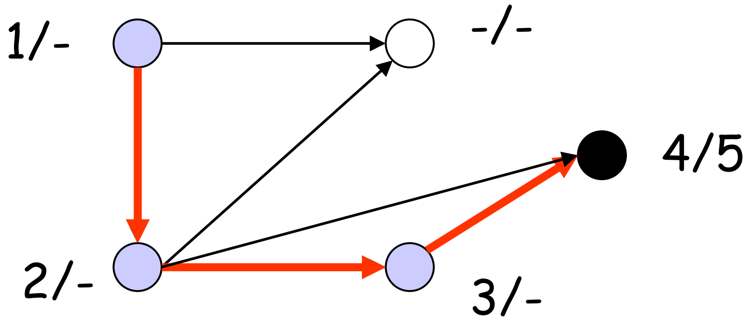


Timestamps: Example (Time 4)

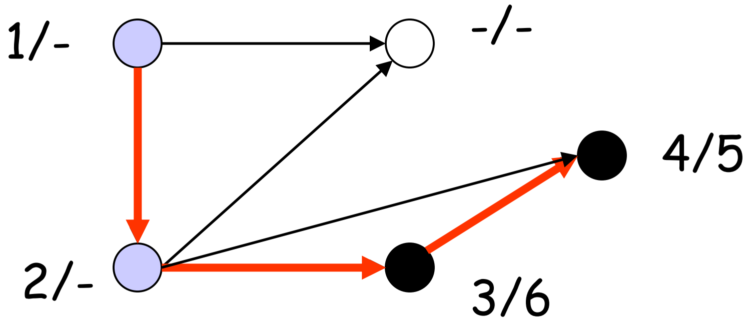


Timestamps: Example (Time 5)

- 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.

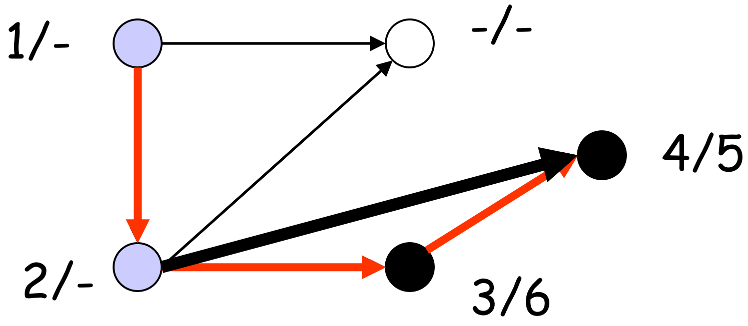


Timestamps: Example (Time 6)

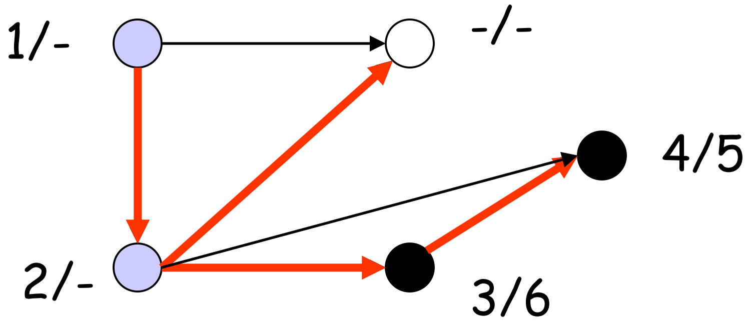


Timestamps: Example (Time 7)

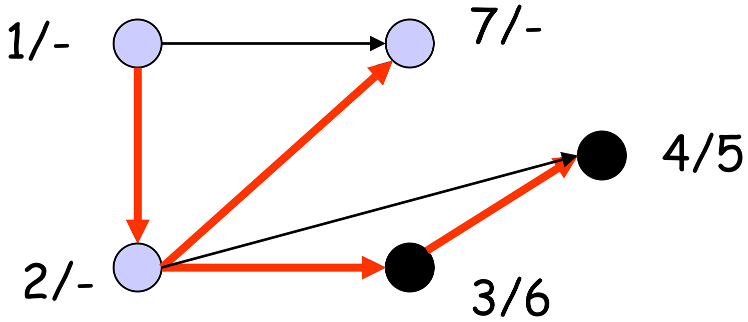
- A gray vertex **u** finds **a black vertex** **v**; the corresponding edge cannot be a tree edge because the black vertex has **already had a predecessor**, i.e., the π(**v**) is found.



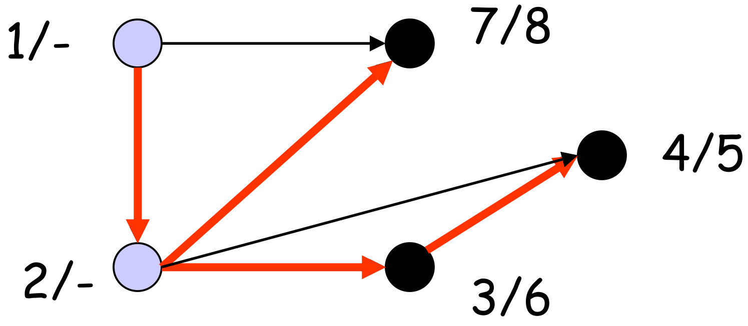
Timestamps: Example (Time 7)



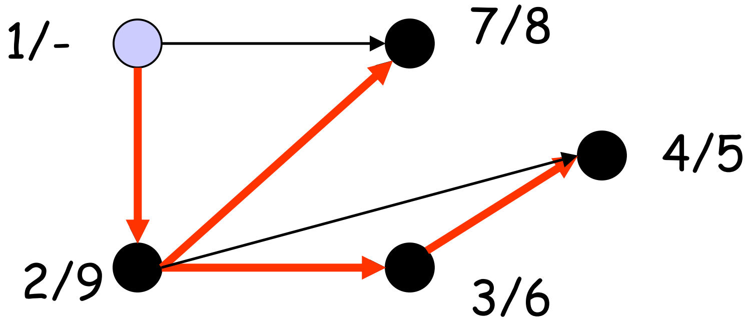
Timestamps: Example (Time 7)



Timestamps: Example (Time 8)

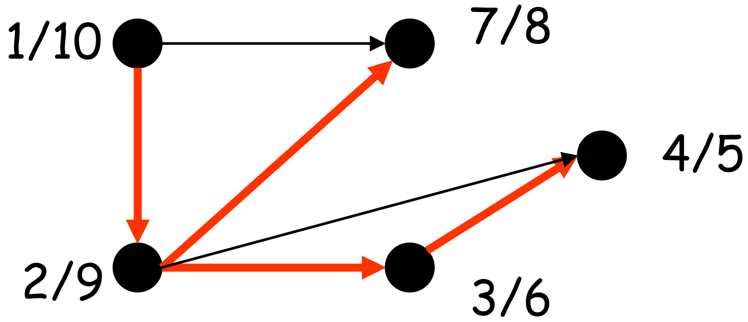


Timestamps: Example (Time 9)

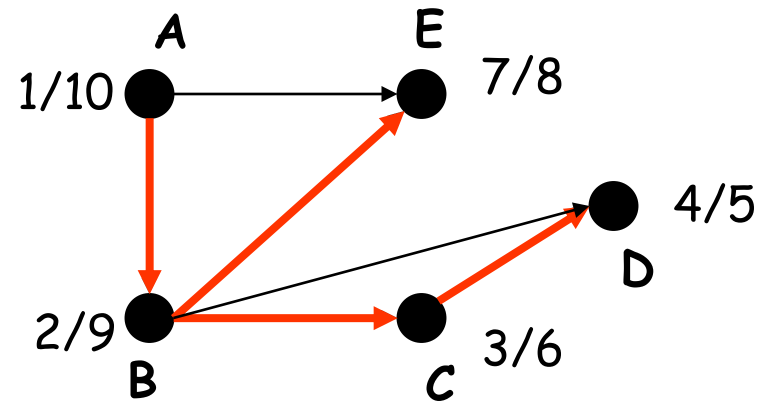


Timestamps: Example (Time 10)

- 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.

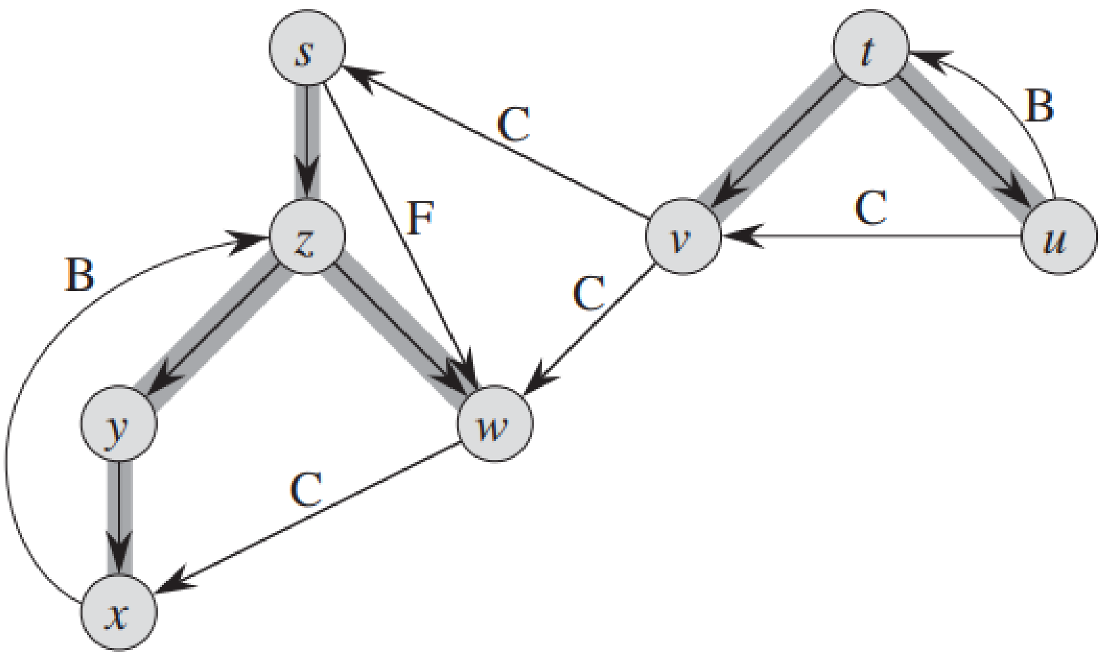
1	2	3	4	5	6	7	8	9	10

A (B(C((D)) (E)))

Classification of edges

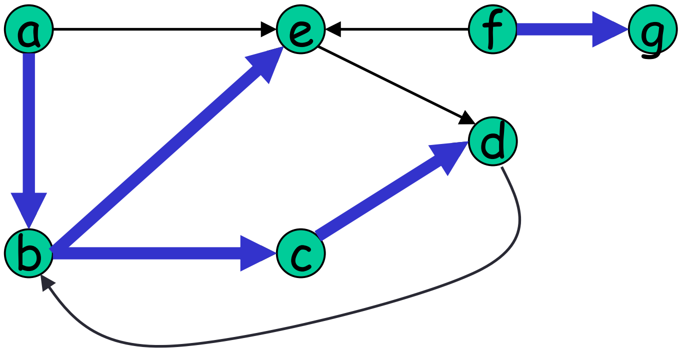
- Tree edges: (π(**v**),**v**) for every **v**.
- **B**ack edges: those edges from a vertex to its ancestor.
- **F**orward edges: those non-tree edges from some vertex **u** to one of its descendant **v**.
- **C**ross edges: go from **one branch to another** or from **one tree to another**.

Example:



Quick exercise

1. For the following graph, a DFS gives the blue edges as DF-tree edges. Classify the remaining edges (a, e), (e, d), (d, b) and (f, e).



Topological Sort: Directed Acyclic Graphs

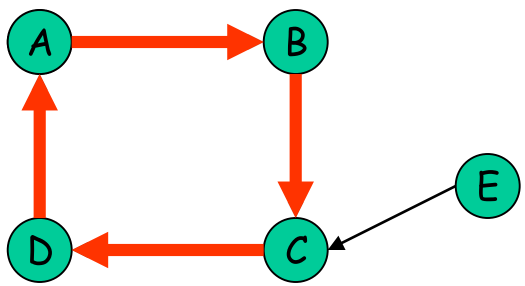
- Topological Sort is a simple application of DFS.
- We first introduce the notion of Directed Acyclic Graphs (DAG).

- 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 DAG is a directed graph that **<u>does not</u>** contain any cycle.

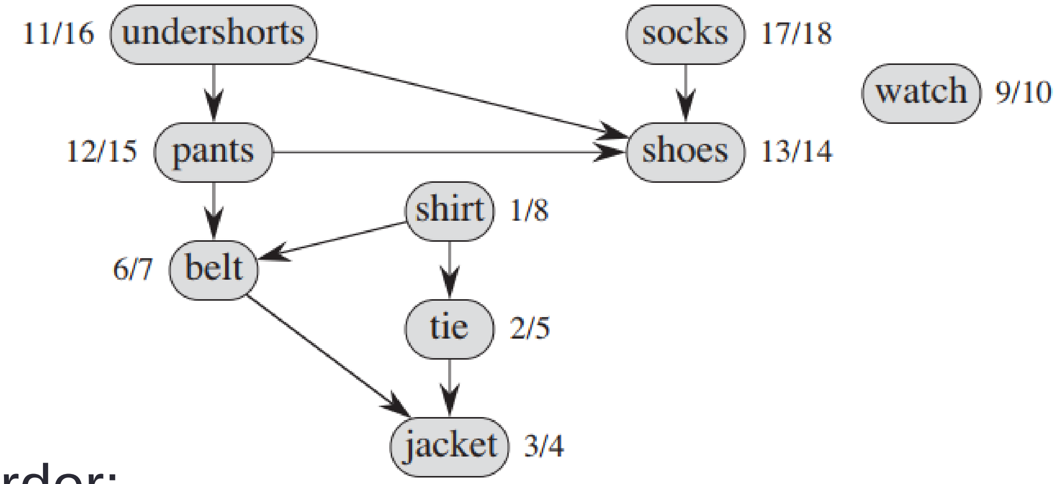


(A,B), (B,C), (C,D), (D,A)

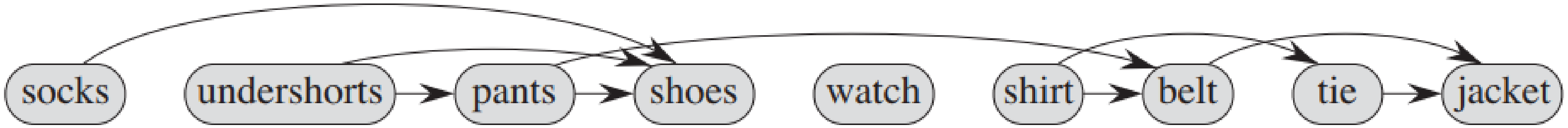
Topological Sort on a DAG

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:

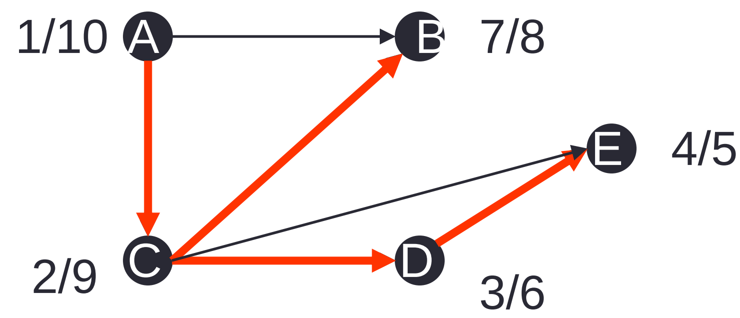


- Topological sort order:



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. Repeatedly 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

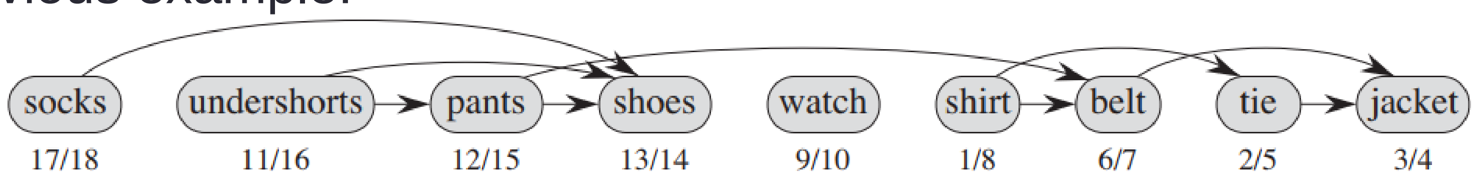
D

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):
    visited[x] = True # If x is not visited, mark it visited
    for y in self.graph[x]:
       if visited[y] == False:
          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.

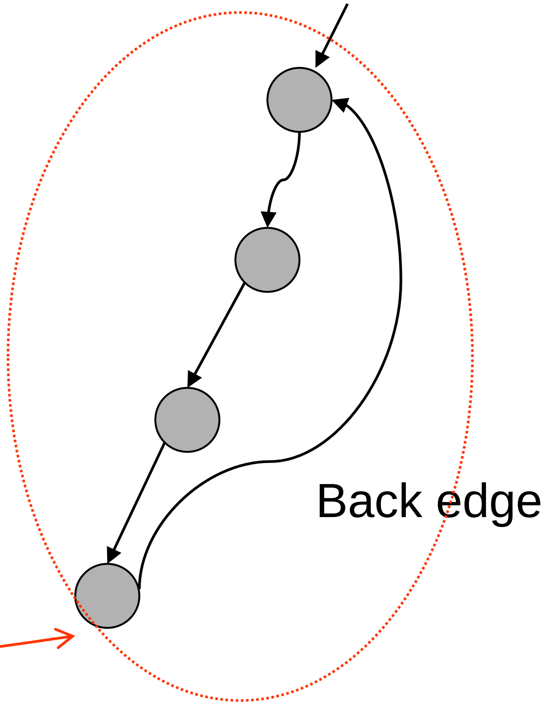
Proof.

$\Rightarrow$ 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

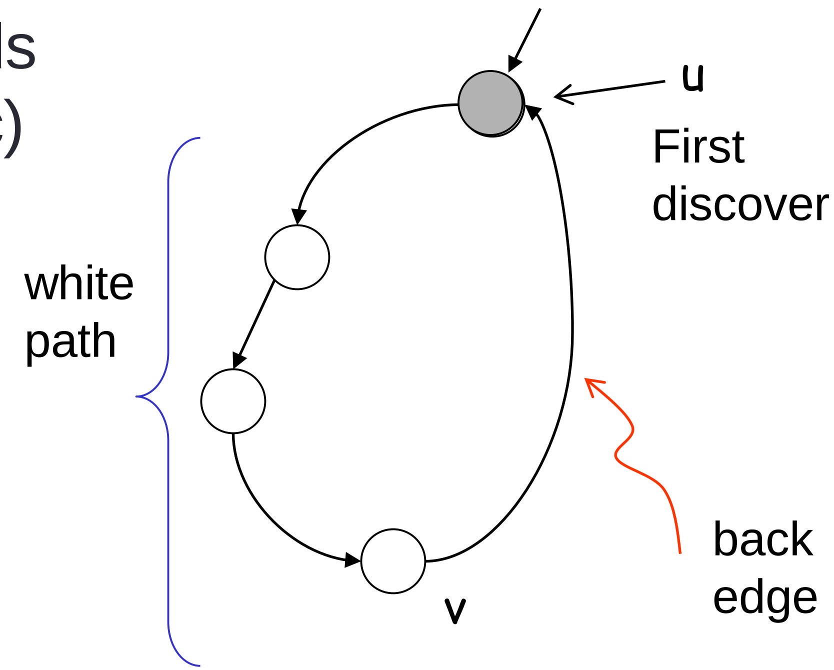
Lemma 1: A directed graph G is acyclic if and only if DFS(G) yields no back edges.

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