Stacks API and finding strongly connected components

CS 146 - Spring 2017

Today

- DFS traversal
- Edge classification wrt DFS traversal
- Strongly connected components
- Tarjan’s algorithm

The stack API

- push()
- pop() <- **no parameter**

DFS on a tree is just a generalization of tree-traversal

```
preorder(v) {
    // do stuff on v
    if v has left child w
        preorder(w)
    if v has right child w
        preorder(w)
}
```

```
dfs(v) {
    // do stuff on v
    for every neighbor w of v
        dfs(w)
}
```

sometimes, it will also be useful to do stuff afterwards

DFS on a graph

```
Set visited = new Set();
dfs(v) {
    visited.add(v)
    // do stuff on v
    for every neighbor w of v
        if (!visited.contains(w))
            dfs(w)
}
```

observation: each node is “visited” (= input to a dfs call) once

Connection with textbook version

```
Set visited = new Set();
dfs(v) {
    visited.add(v)
    // do stuff on v
    for every neighbor w of v
        if (!visited.contains(w))
            dfs(w)
}
```

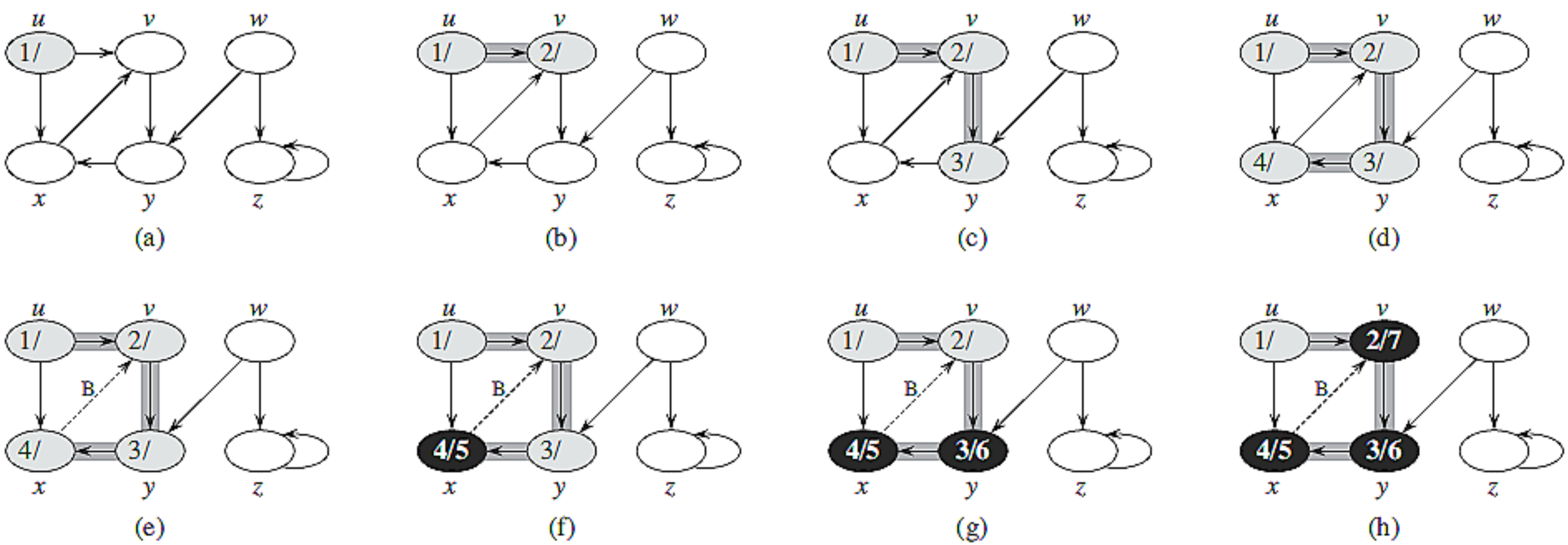
every vertex starts out **white (not visited)**

upon entering a recursive call, v turns **grey (visiting)**

upon exiting the recursive call, v turns **black (finished visiting)**

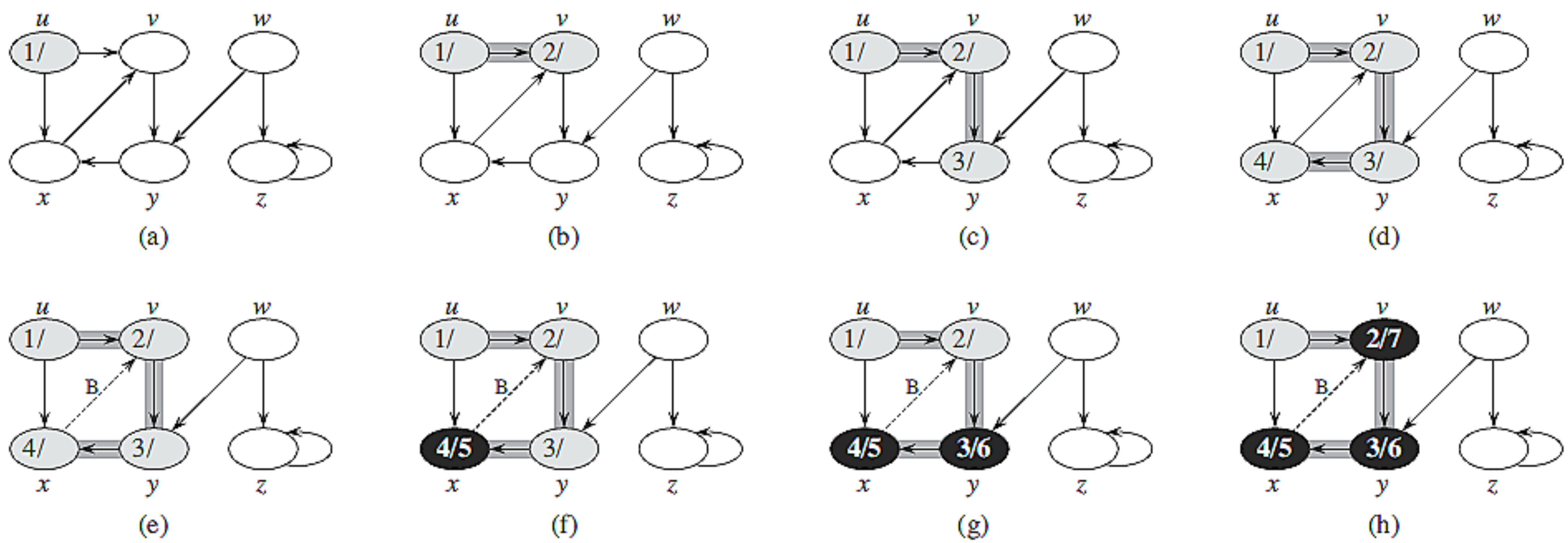
Note similarities with the lifecycle of a vertex in BFS

Example with timestamps

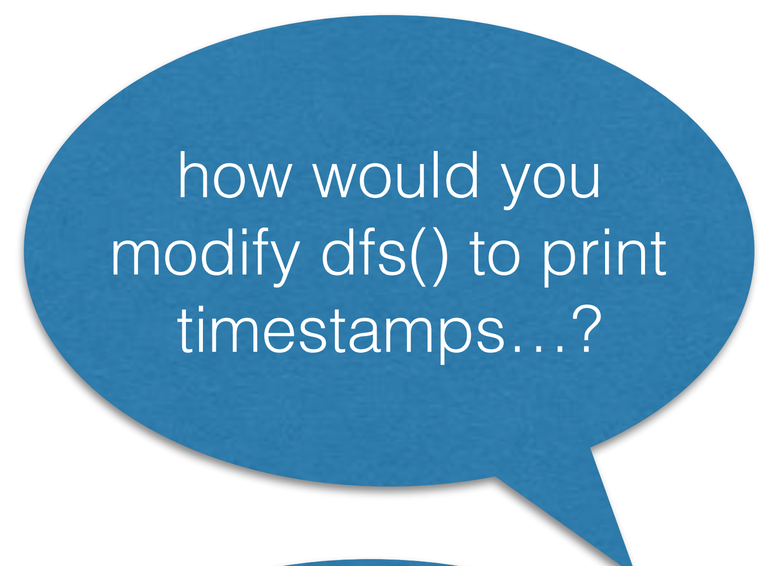


each vertex gets a **“visiting” timestamp** when first visited

each vertex gets a **“done visiting” timestamp** when finished




```
Set visited = new Set();
dfs(v) {
    visited.add(v)
    // do stuff on v
    for every neighbor w of v
        if (!visited.contains(w))
            dfs(w)
}
```



DFS for visit timestamp

```
Set visited = new Set();
time = 1


dfs(v) {
    visited.add(v)
    print("visiting v: " + time); time++
    for every neighbor w of v
        if (!visited.contains(w))
            dfs(w)
    print("done visiting v: " + time); time++
}
```

To ensure all vertices are visited (and get a timestamp)

```
Set visited = new Set();
time = 1

dfs(v) {
    visited.add(v)
    print("visiting v: " + time); time++
    for every neighbor w of v
        if (!visited.contains(w))
            dfs(w)
    print("done visiting v: " + time); time++
}
```

```
for each vertex v
    if (!visited.contains(v))
        dfs(v)
```

Time complexity of DFS

```
Set visited = new Set();
dfs(v) {
    visited.add(v)
    // do stuff on v
    for every neighbor w of v
        if (!visited.contains(w))
            dfs(w)
}
```

each vertex is input of at most 1 recursive call

each edge is explored at most once overall

as long as “do stuff” is O(1)
DFS is O(V + E)

Time complexity of DFS

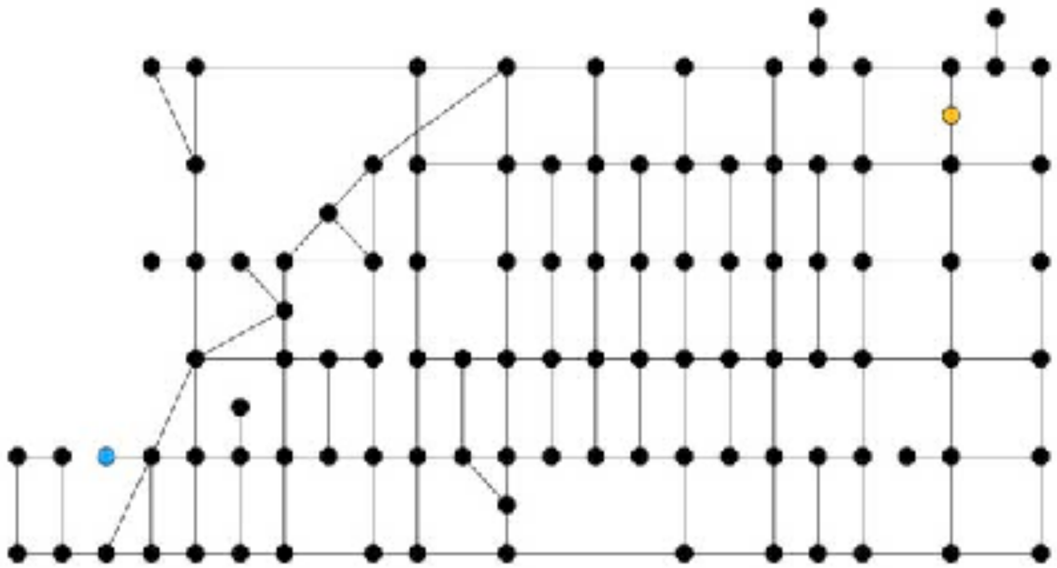
```
Set visited = new Set();

dfs(v) {

    visited.add(v)

    // do stuff on v

    for every neighbor w of v

        if (!visited.contains(w))

            dfs(w)

}
```

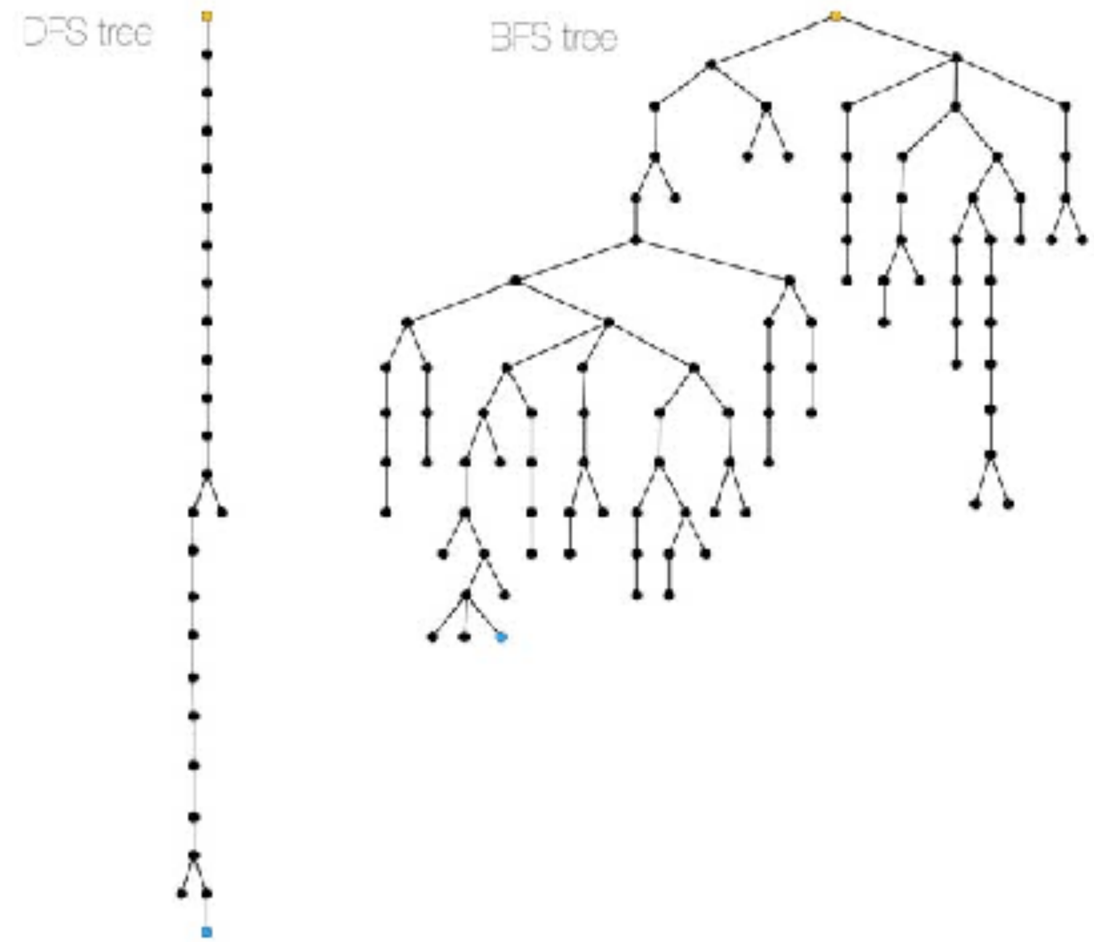
each vertex is input of at most 1 recursive call

each edge is explored at most once overall

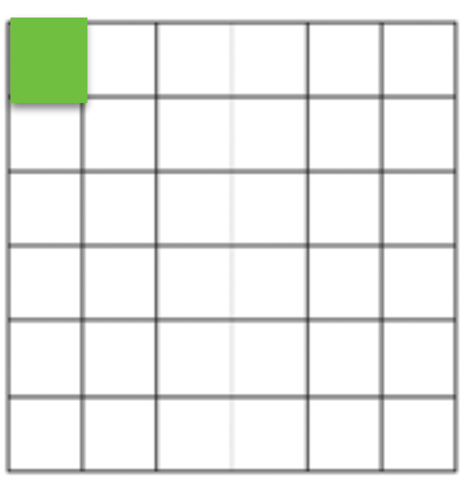
as long as “do stuff” is O(1) DFS is O(V + E)

Note similarities with BFS/Dijkstra analysis despite DFS being a recursive algorithm

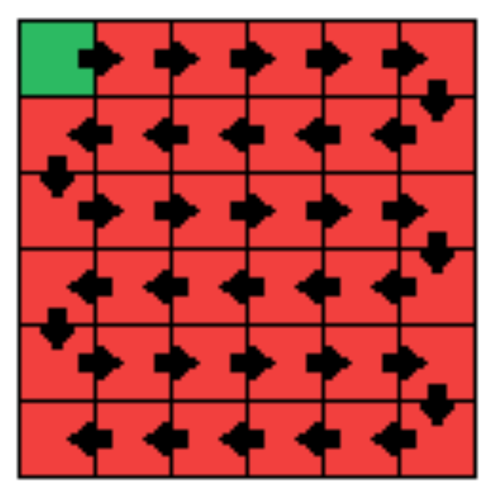




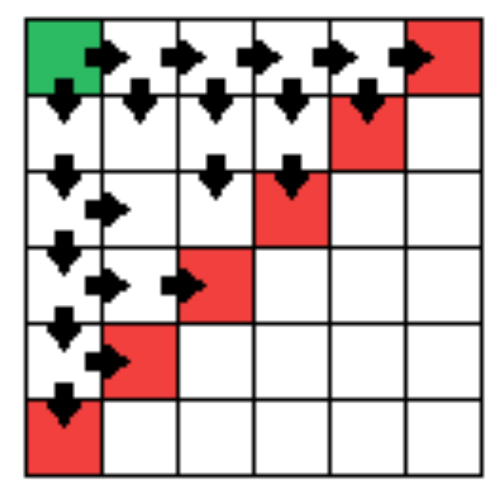
a graph to traverse



DFS



BFS



https://www.codefellows.org/blog/trees-as-graphs-vs-trees-as-data-structures/
http://stackoverflow.com/questions/20192445/which-procedure-we-can-use-for-maze-exploration-bfs-or-dfs

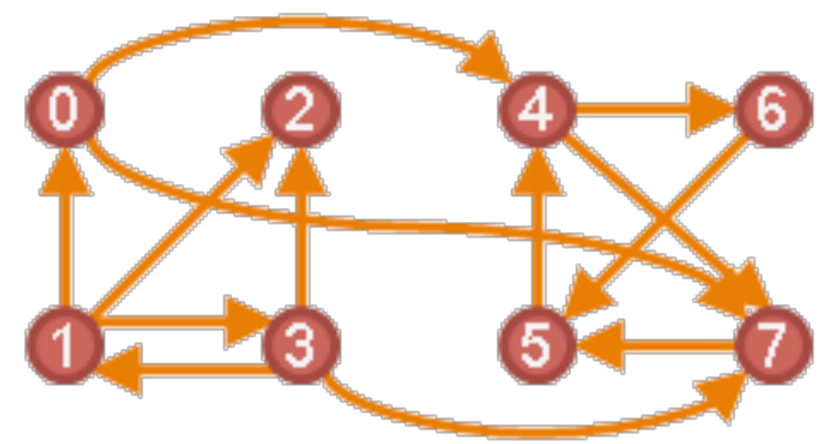
DFS forest

```
Set visited = new Set();
for each vertex v
    if (!visited.contains(v))
        dfs(v)
```

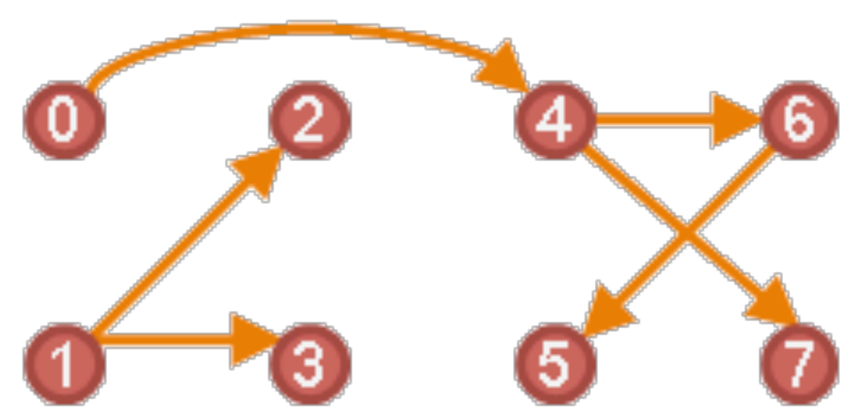
one DFS tree

```
dfs(v) {
    visited.add(v)
    for every neighbor w of v
        if (!visited.contains(w))
            dfs(w)
}
```

v→w is a tree edge
= edge of DFS tree

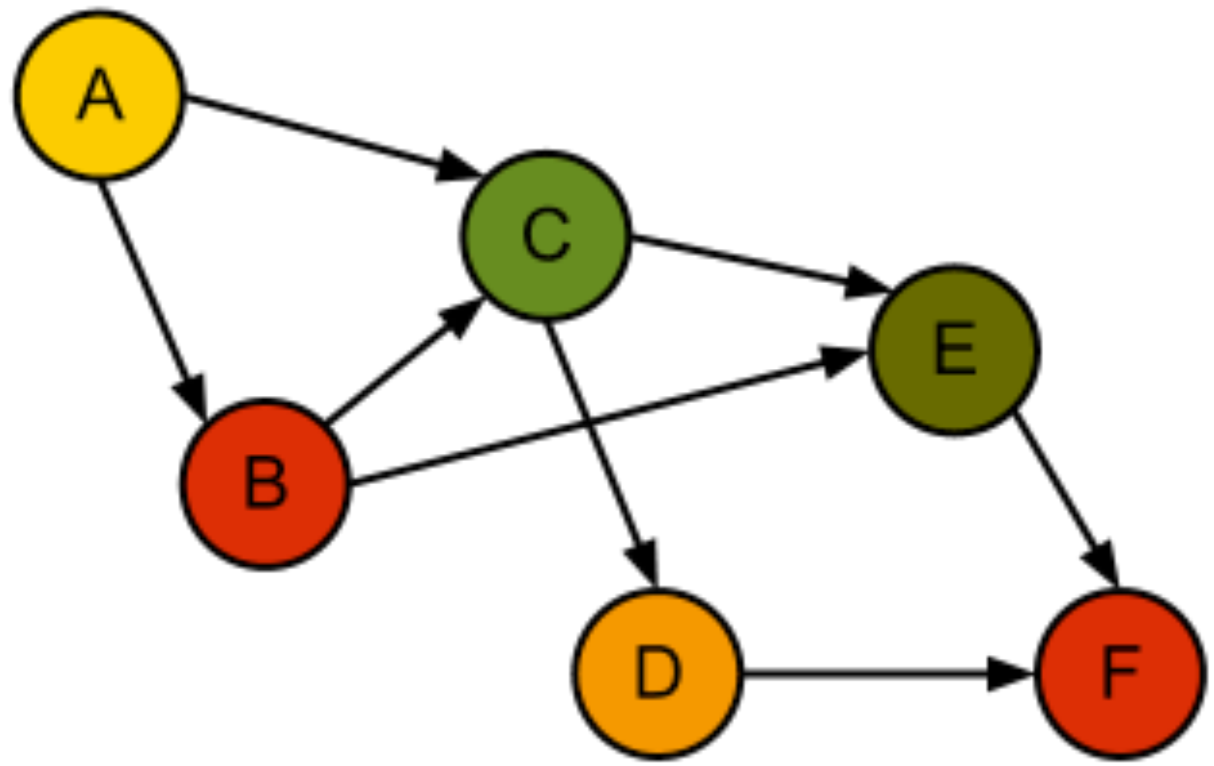


input graph



one possible
dfs forest made up of
two trees: dfs(0)
and dfs(1)

http://www.csie.ntnu.edu.tw/~u91029/Graph.html

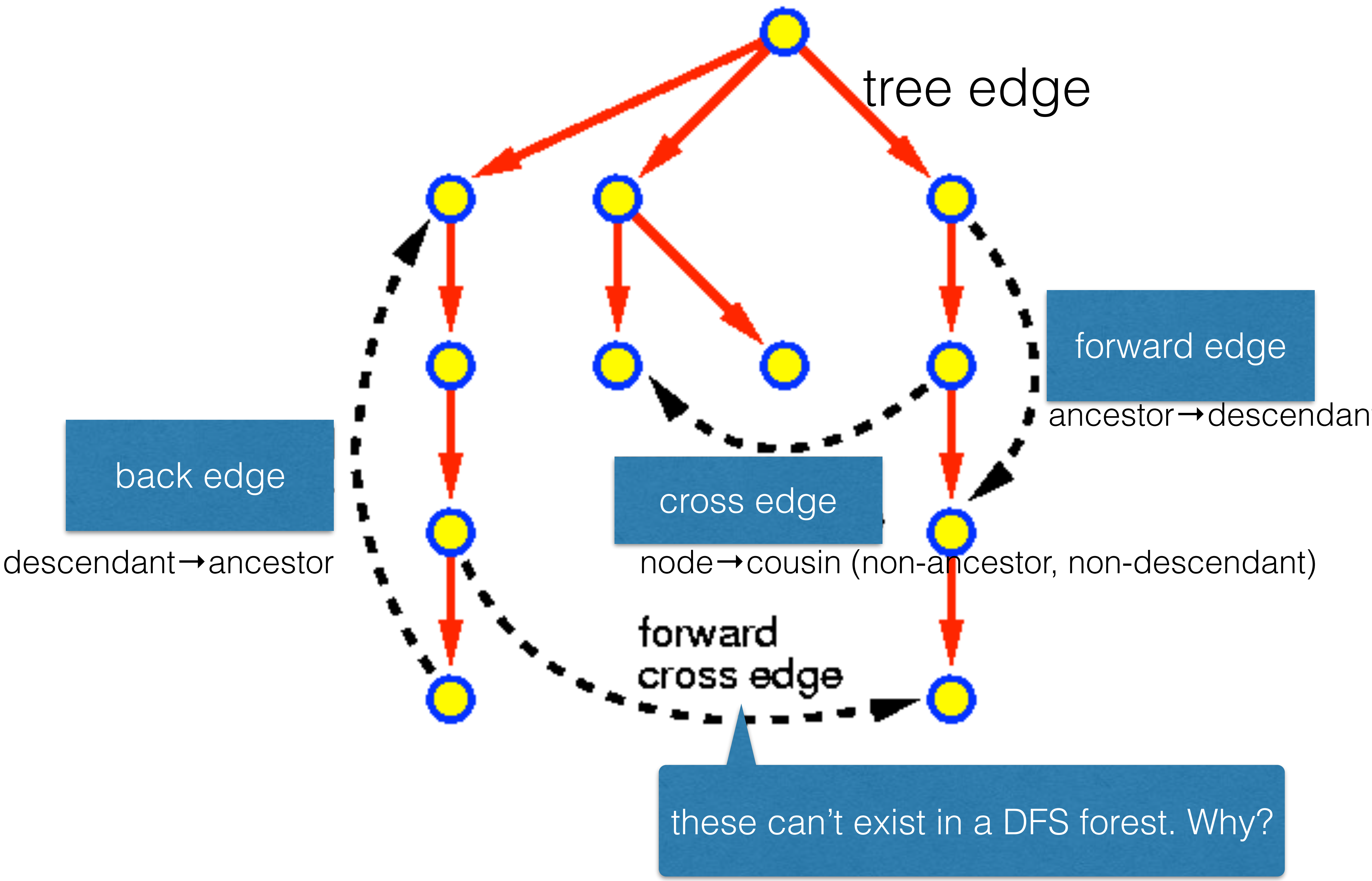


Find a DFS traversal that produces a forest with 1 tree.

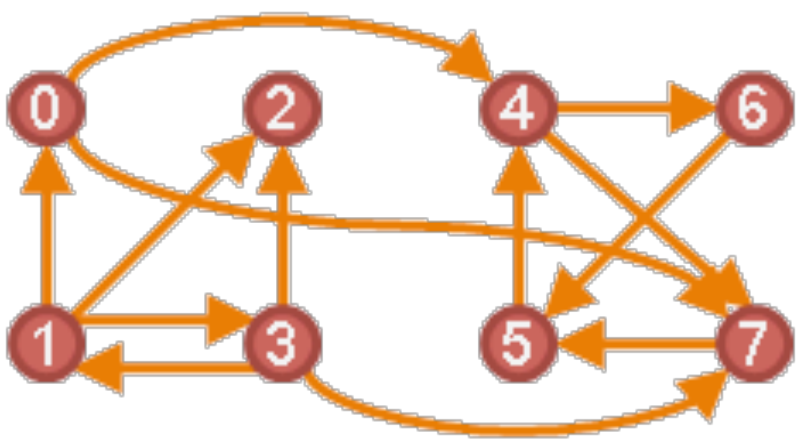
Find a DFS traversal that produces a forest with 3 trees.

http://www.stoimen.com/blog/2012/10/28/computer-algorithms-shortest-path-in-a-directed-acyclic-graph/

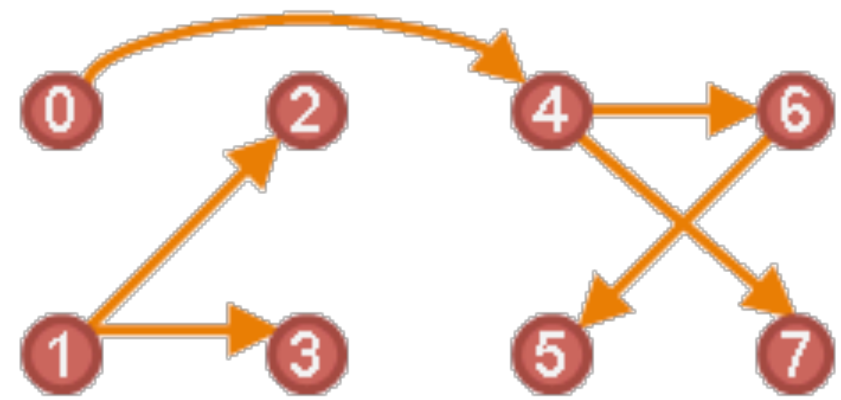
Edge classification



Can you classify all the other edges?

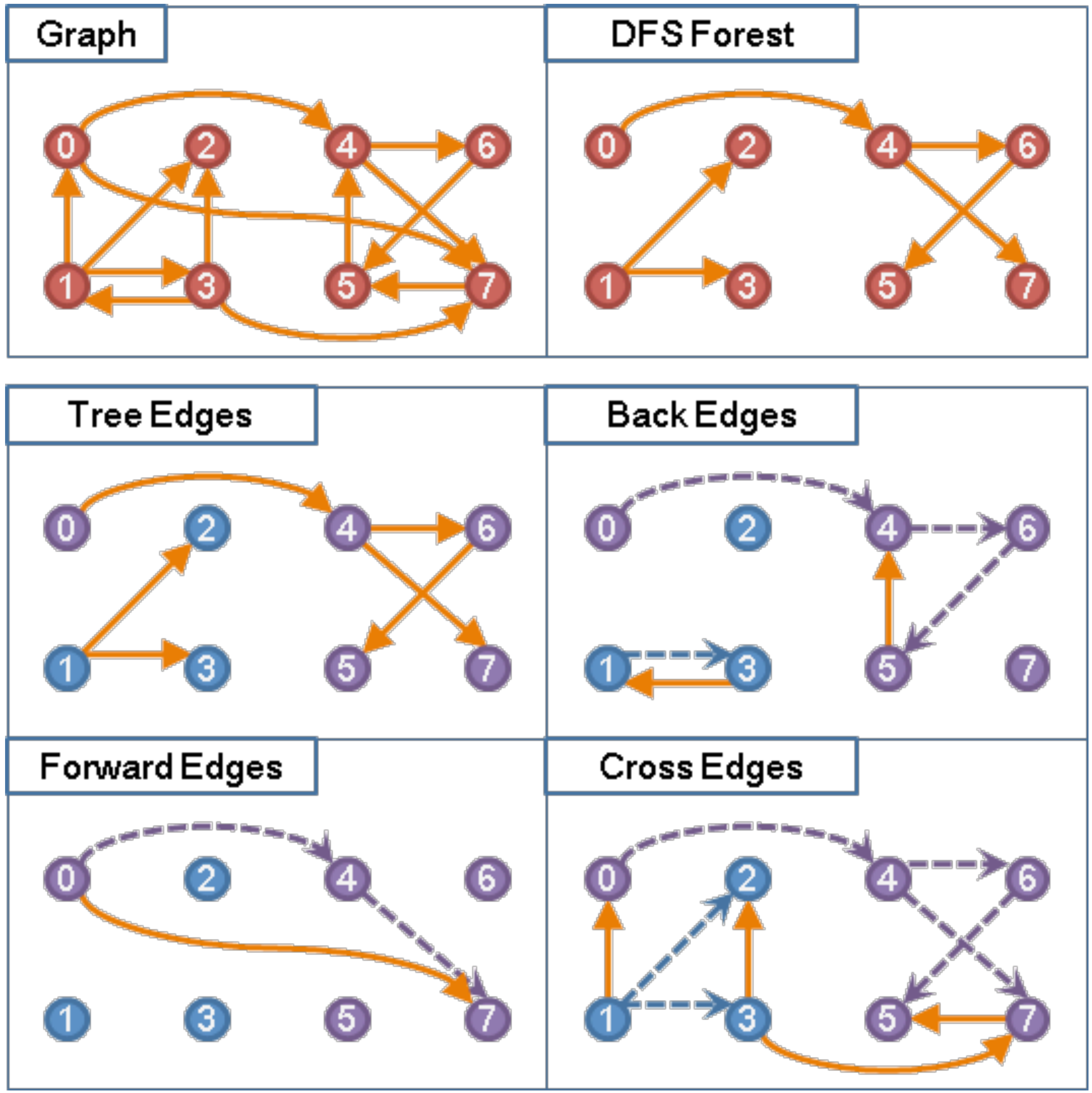


input graph



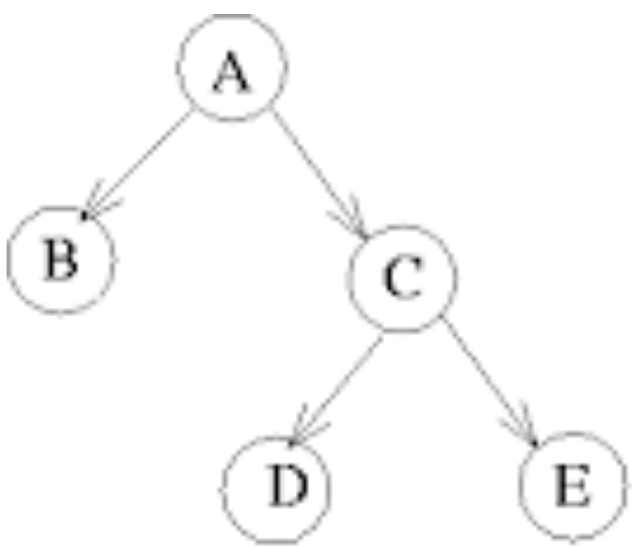
dfs forest made up of tree edges

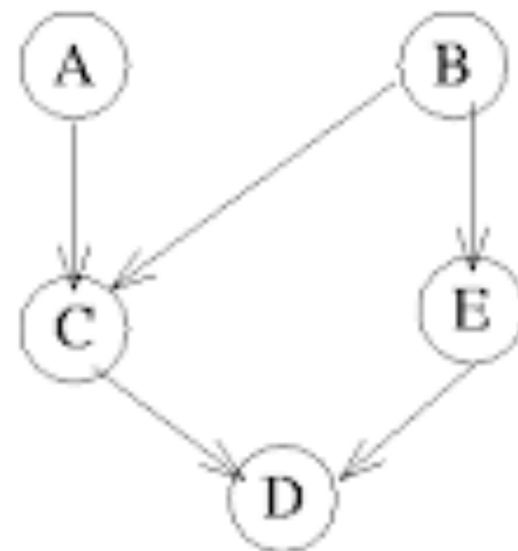
http://www.csie.ntnu.edu.tw/~u91029/Graph.html

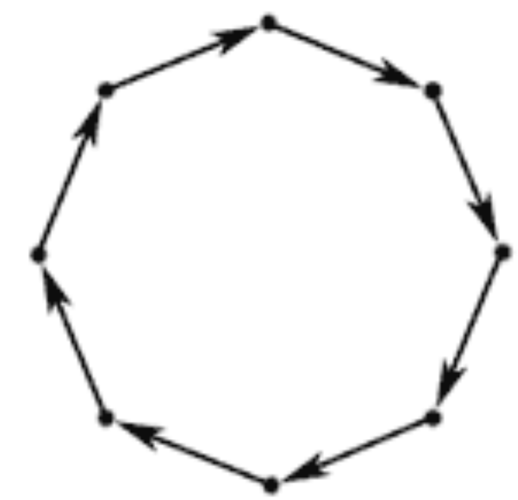


http://www.csie.ntnu.edu.tw/~u91029/Graph.html

What does it mean for a **directed graph** to be connected?

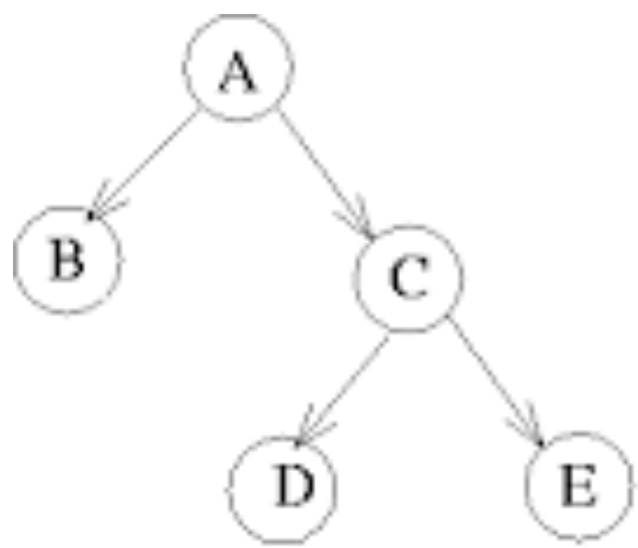




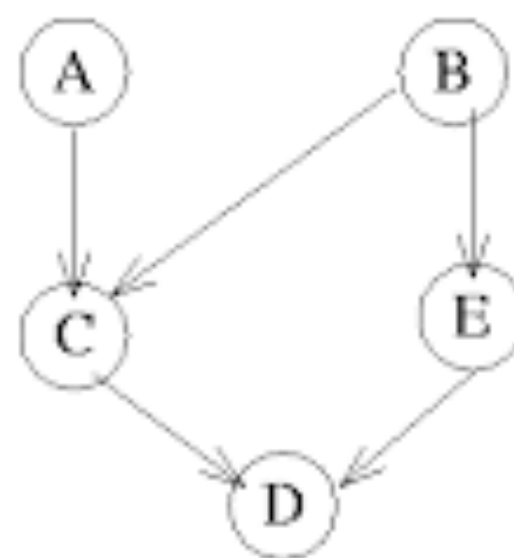


Which of these graphs would you consider to be "connected"? Why?

V can **reach** W if there is a directed path connecting V to W



A can reach every other vertex

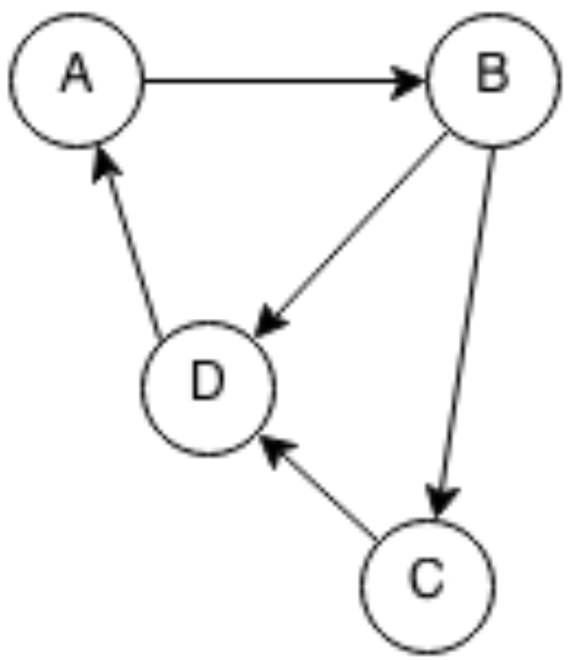


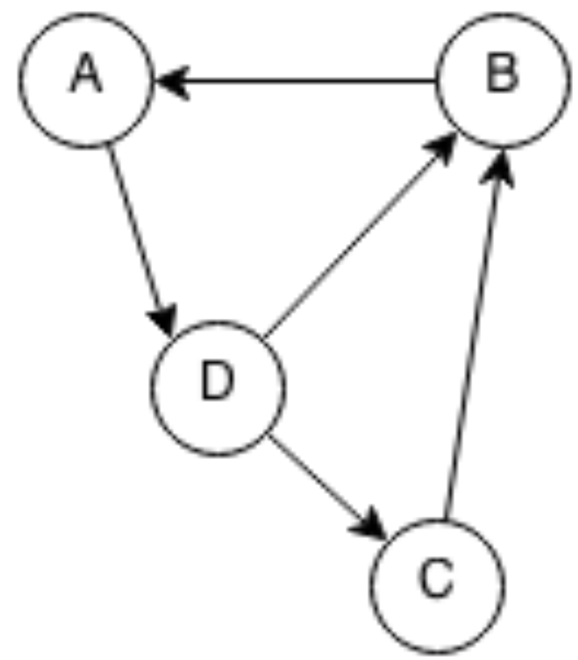
B can reach every other vertex except A

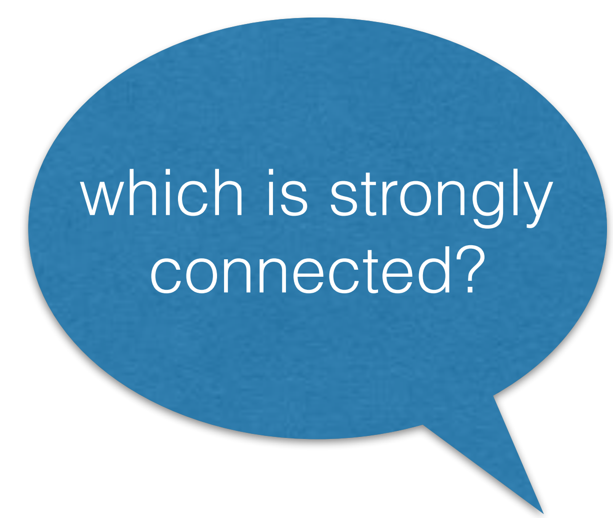
every vertex can reach every other vertex

A directed graph is said to be **strongly connected** if …

every vertex in the graph is reachable from
every other vertex in the graph





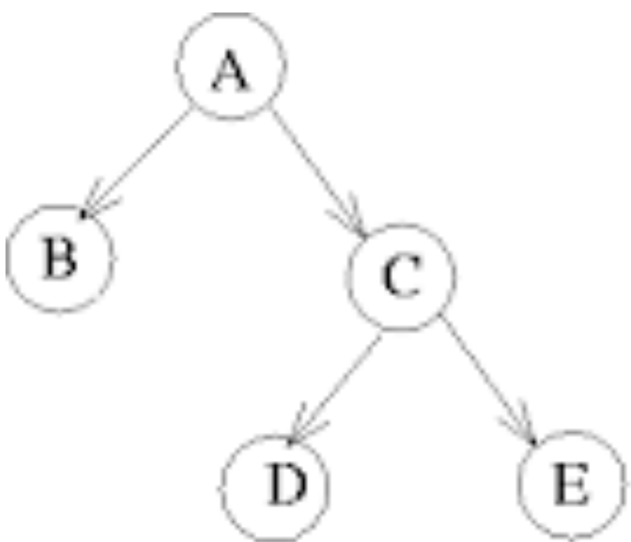


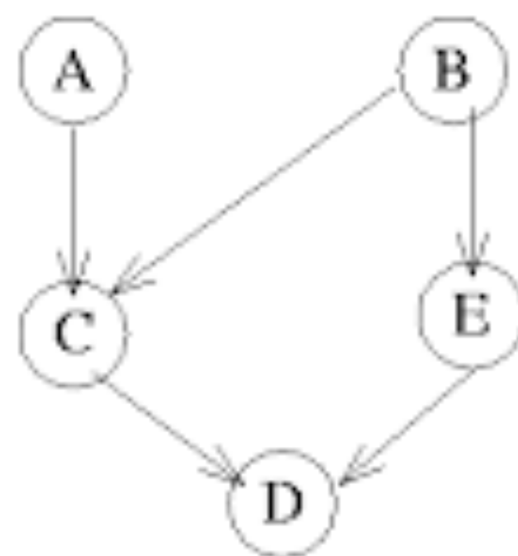
Every directed graph can be decomposed into a set of **strongly connected components (SCCs)**

A strongly connected component is a (largest possible) set of vertices that are mutually reachable.

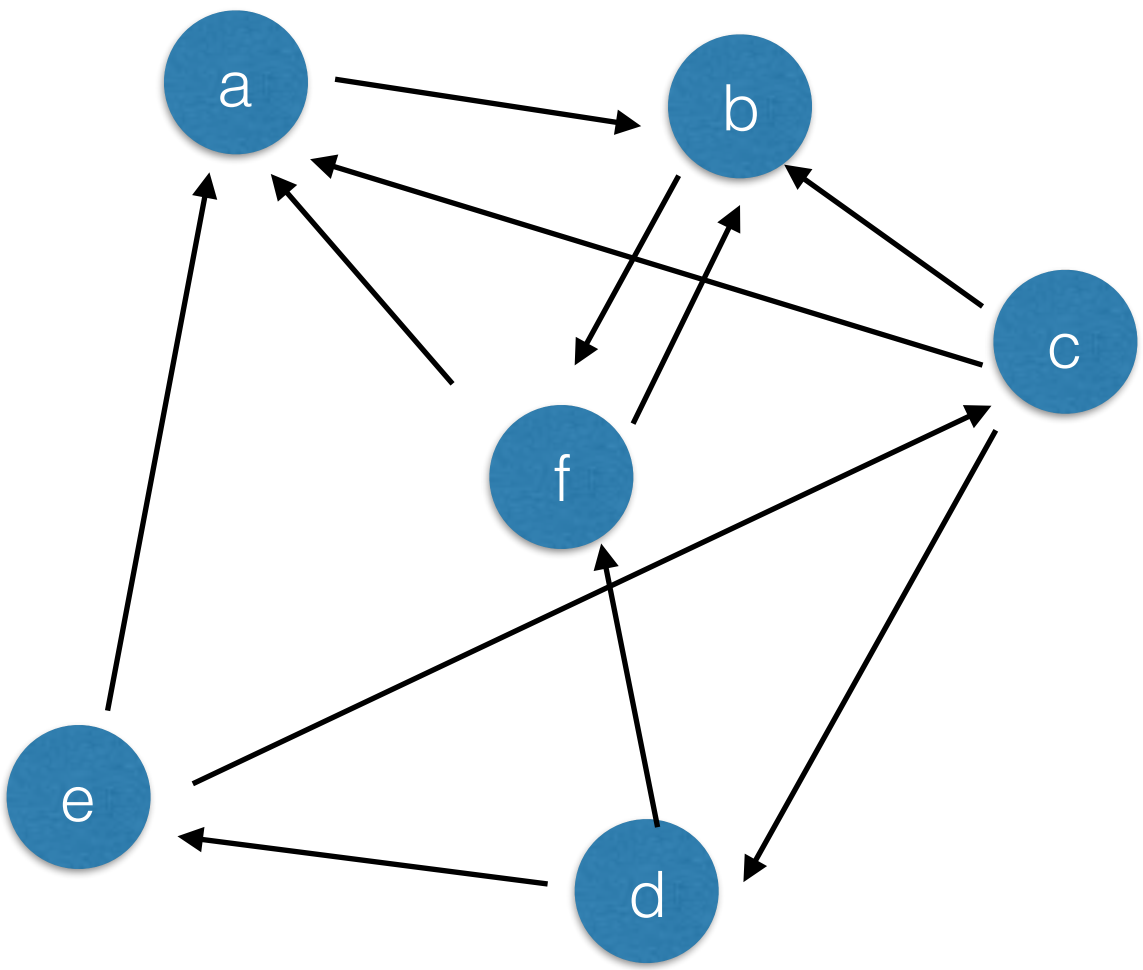
Every vertex is reachable from itself.

Find the strongly connected components of each of these graphs





Find the strongly connected components of this graph



What is an algorithm for finding
all the strongly connected components
of a directed graph?

```
Stack<Vertex> stack = new Stack();
Map<Vertex, Integer> index = new Map();
Map<Vertex, Integer> lowlink = new Map();
int nextIndex = 0;
for each vertex v
    if (!index.containsKey(v))
        scc(v)

scc(v) {
    // init v
    for every neighbor w of v {
        if (!index.containsKey(w)) { // tree edge
            scc(w)
            lowlink.put(v, min(lowlink.get(v), lowlink.get(w)))
        } else if (w on stack) { // non-tree edge
            lowlink.put(v, min(lowlink.get(v), index.get(w)))
        }
    }
    // pop current scc of v if done
}
```

Tarjan's algorithm

```
Set visited = new Set();
for each vertex v
    if (!visited.contains(v))
        dfs(v)

dfs(v) {
    visited.add(v)
    for every neighbor w of v
        if (!visited.contains(w
            dfs(w)
}
```

DFS

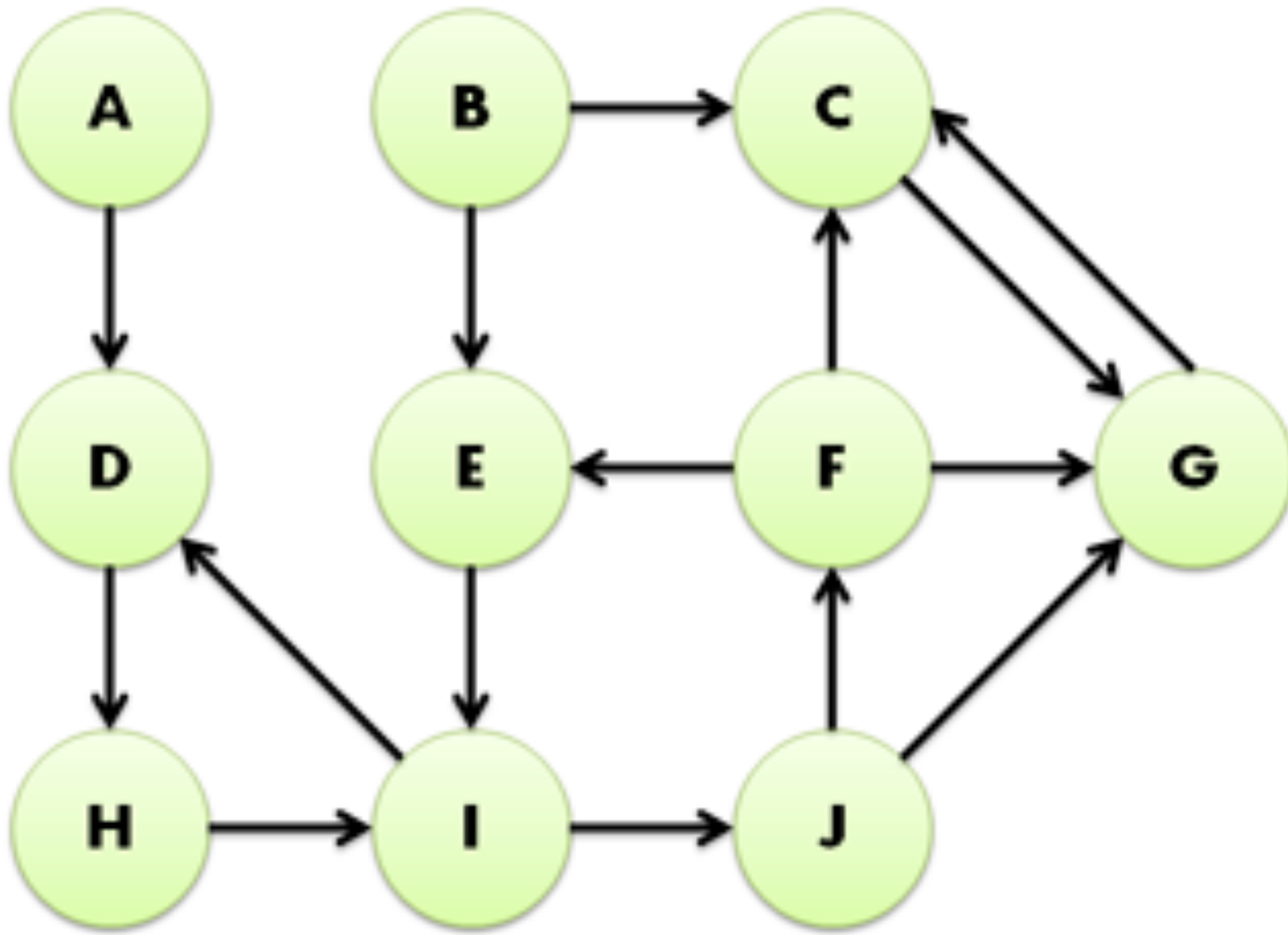
```
Stack<Vertex> stack = new Stack();
Map<Vertex, Integer> index = new Map();
Map<Vertex, Integer> lowlink = new Map();
int nextIndex = 0;
for each vertex v
    if (!index.containsKey(v))
        scc(v)

scc(v) {
    // init v
    for every neighbor w of v {
        if (!index.containsKey(w)) { // tree edge
            scc(w)
            lowlink.put(v, min(lowlink.get(v), low
        } else if (w on stack) { // non-tree edge
            lowlink.put(v, min(lowlink.get(v), ind
        }
    }
    // pop current scc of v if done
}
```

```
stack.push(v);
index.put(v, nextIndex);
lowlink.put(v, nextIndex);
nextIndex++;
```

```
if (index.get(v) == lowlink.get(v)) {
    List<Vertex> component = new List();
    while (stack.peek() != v)
        component.add(stack.pop());
    component.add(stack.pop());
    print component;
}
```

Tarjan’s algorithm

Tarjan’s algorithm: example



http://www.cs.rpi.edu/~goldsd/fall2013-csci2300-labs.php

```
Stack<Vertex> stack = new Stack();
Map<Vertex, Integer> index = new Map();
Map<Vertex, Integer> lowlink = new Map();
int nextIndex = 0;
for each vertex v
    if (!index.containsKey(v))
        dfs(v)

scc(v) {
    // init v
    for every neighbor w of v {
        if (!index.containsKey(w)) { // tree edge
            scc(w)
            lowlink.put(v, min(lowlink.get(v), low
        } else if (w on stack) { // non-tree edge
            lowlink.put(v, min(lowlink.get(v), ind
        }
    }
    // pop current scc of v if done
}
```

stack invariant:
a vertex stays on stack after it
is visited if it has an edge to a vertex
earlier on stack (and visited earlier)

```
stack.push(v);
index.put(v, nextIndex);
lowlink.put(v, nextIndex);
nextIndex++;
```

```
if (index.get(v) == lowlink.get(v)) {
    List<Vertex> component = new List();
    while (stack.peek() != v)
        component.add(stack.pop());
    component.add(stack.pop());
    print component;
}
```

Tarjan's algorithm

```
Stack<Vertex> stack = new Stack();
Map<Vertex, Integer> index = new Map();
Map<Vertex, Integer> lowlink = new Map();
int nextIndex = 0;
for each vertex v
    if (!index.containsKey(v))
        dfs(v)

scc(v) {
    // init v
    for every neighbor w of v {
        if (!index.containsKey(w)) { // tree edge
            scc(w)
            lowlink.put(v, min(lowlink.get(v), low
        } else if (w on stack) { // non-tree edge
            lowlink.put(v, min(lowlink.get(v), ind
        }
    }
    // pop current scc of v if done
}
```

time complexity:
DFS with additional bookkeeping
problem: “w on stack” not constant

```
stack.push(v);
index.put(v, nextIndex);
lowlink.put(v, nextIndex);
nextIndex++;
```

```
if (index.get(v) == lowlink.get(v)) {
    List<Vertex> component = new List();
    while (stack.peek() != v)
        component.add(stack.pop());
    component.add(stack.pop());
    print component;
}
```

Tarjan’s algorithm

Tarjan's algorithm

time complexity:
fix: use **set** to track what's on the stack
DFS with O(1) bookkeeping is **O(V+E)**

```
Stack<Vertex> stack = new Stack();
Set<Vertex> onStack = new Map();
Map<Vertex, Integer> index = new Map();
Map<Vertex, Integer> lowlink = new Map();
int nextIndex = 0;
for each vertex v
    if (!index.containsKey(v))
        dfs(v)

scc(v) {
    // init v
    for every neighbor w of v {
        if (!index.containsKey(w)) { // tree edge
            scc(w)
            lowlink.put(v, min(lowlink.get(v), low
        } else if (onStack.contains(w)) { // non-tr
            lowlink.put(v, min(lowlink.get(v), ind
        }
    }
    // pop current scc of v if done
}
```

```
stack.push(v); onStack.add(v);
index.put(v, nextIndex);
lowlink.put(v, nextIndex);
nextIndex++;
```

```
if (index.get(v) == lowlink.get(v)) {
    List<Vertex> component = new List();
    while (stack.peek() != v) {
        onStack.remove(stack.peek());
        component.add(stack.pop());
    }
    onStack.remove(stack.peek());
    component.add(stack.pop());
    print component;
}
```