Lecture 15: Basic Graph Algorithms

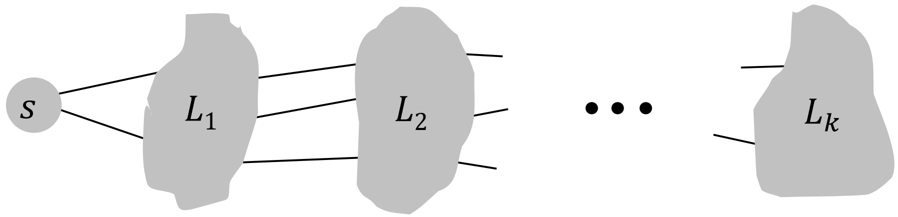
Version of March 10, 2019

Processing Graphs

- Graphs model many scenarios
  - Many problems are presented as graph problems
  - Can then use known general graph algorithms to solve those problems
- Data is inputted as adjacency matrix or, more commonly, an adjacency lists
- To start processing the data, we often need some way to derive structure from this input
- Breadth First Search and Depth First Search are the most common simple ways of imposing structure.

Breadth First Search

BFS idea. Explore outward from s in all possible directions, adding nodes one "layer" at a time.



BFS.

- $L_0 = \{s\}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

Def: The distance from u to v is the number of edges on the shortest path from u to v.

Theorem. For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.

BFS Algorithm

BFS(G, s):
for each vertex $u \in V - \{s\}$
$u.color \leftarrow white$
$u.d \leftarrow \infty$
$u.p \leftarrow nil$
$s.color \leftarrow gray$
$s.d \leftarrow 0$
initialize an empty queue Q
Enqueue(Q, s**)**
……………

Color: indicates status

- white: (initial value) undiscovered
- gray: discovered, but neighbors not fully processed
- black: discovered and neighbors fully processed

Every node stores a color, a distance and a parent

Distance (d): the length of shortest path from s to u

Parent (p): u's predecessor on the shortest path from s to u

BFS Algorithm

BFS(G,s):
for each vertex $u \in V - \{s\}$
    $u.color \leftarrow white$
    $u.d \leftarrow \infty$
    $u.p \leftarrow nil$
$s.color \leftarrow gray$
$s.d \leftarrow 0$
1. **initialize an empty queue** Q
2. **Enqueue(**Q,s**)**
3. **while** $Q \neq \emptyset$ **do**
4. $u \leftarrow$ **Dequeue(**Q**)**
5.   **for each** $v \in Adj[u]$
6.   **if** $v.color = white$ **then**
7. $v.color \leftarrow gray$
8. $v.d \leftarrow u.d + 1$
9. $v.p \leftarrow u$
10.   **Enqueue(**Q,v**)**
11. $u.color \leftarrow black$

- Algorithm keeps current active nodes in a Queue Q (First In First Out)
- Starts by inserting s in Q (2)
- At each step takes node u off Q (4)
  - Checks all neighbors v of u (5)
  - If v has not been seen yet (6)
    - Marks v as seen (7)
    - Says that distance from s to v is 1 + dist to u (8)
    - Makes u the parent of v (9)
    - inserts v in queue (10)
  - Marks u as being fully processed (11)

Note: Nodes in Queue
- Are ones that have been seen but are unprocessed (gray)

BFS Algorithm

```
BFS(G, s):
for each vertex u ∈ V − {s}
    u.color ← white
    u.d ← ∞
    u.p ← nil
s.color ← gray
s.d ← 0
initialize an empty queue Q
Enqueue(Q, s)
while Q ≠ Ø do
    u ← Dequeue(Q)
    for each v ∈ Adj[u]
        if v.color = white then
            v.color ← gray
            v.d ← u.d + 1
            v.p ← u
            Enqueue(Q, v)
    u.color ← black
```

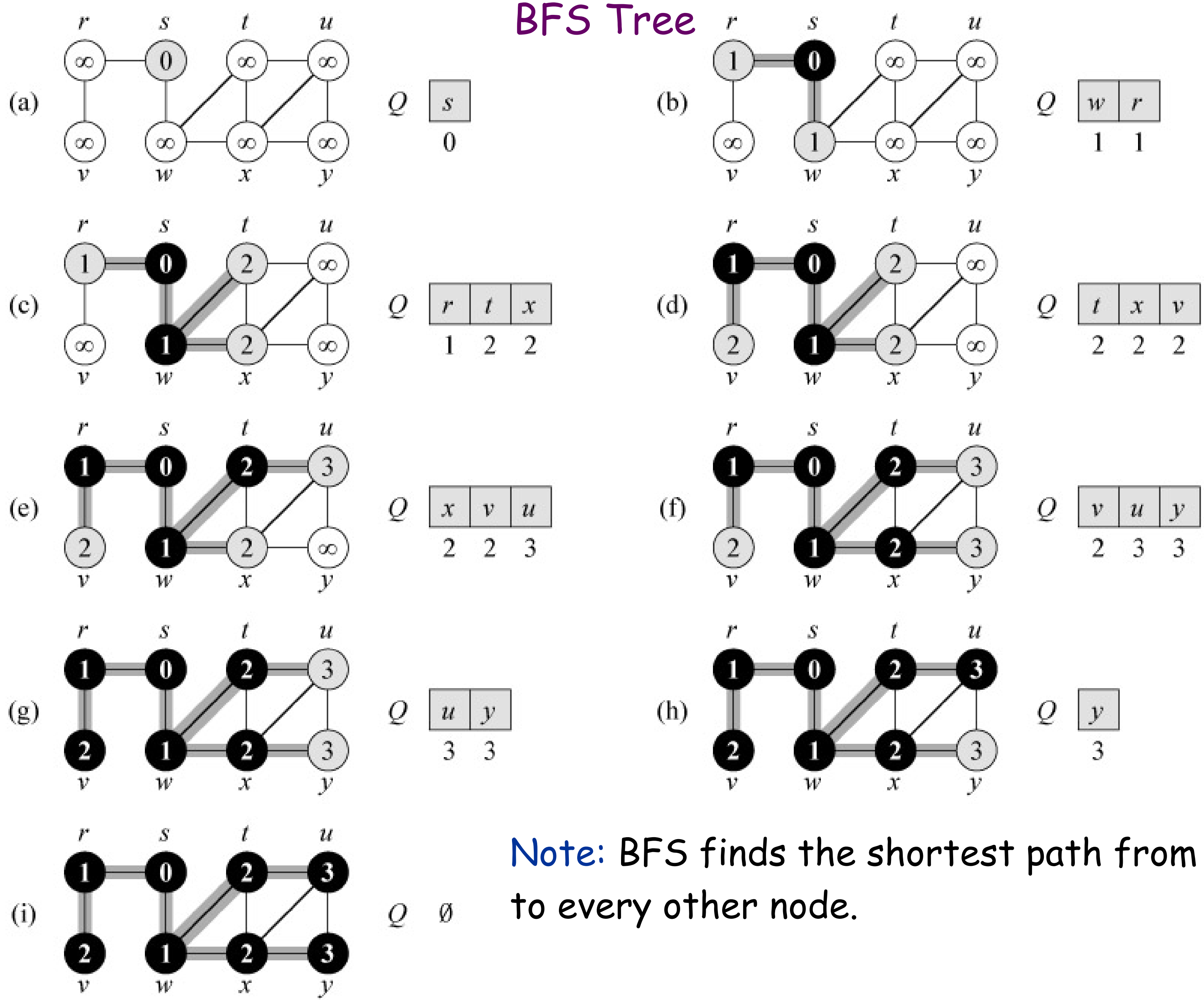
Parent pointers:

- Pointing to the node that leads to its discovery
- Parent must be in L_{i-1}
- Can follow parent pointers to find the actual shortest path
- The pointers form a tree, rooted at s

Running time:

$\sum_u (1 + \deg(u)) = \Theta(V + E)$, which is $\Theta(E)$ if the graph is connected.

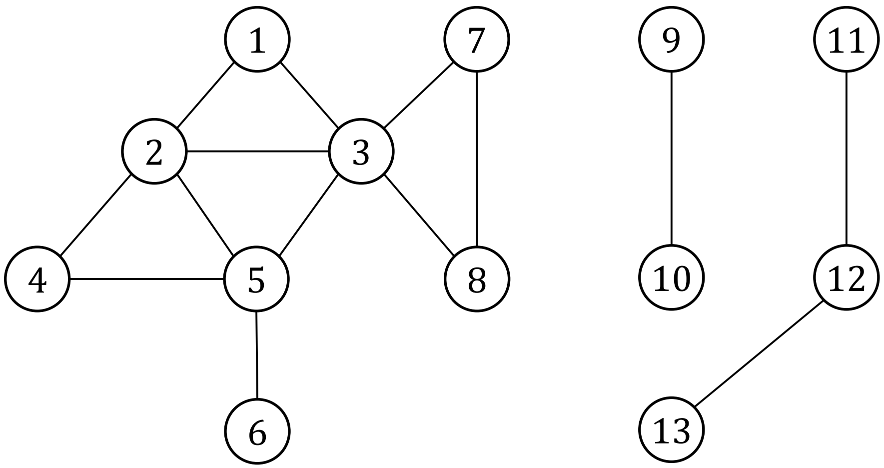
BFS Tree



Note: BFS finds the shortest path from s to every other node.

Connected Components

Connected component containing s. All nodes reachable from s.



Connected component containing node $1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

BFS starting from s finds the connected component containing s.

Repeatedly running BFS from an undiscovered node finds all the connected components.

Modification for Finding Connected Components

```
BFS(G):
for each vertex u ∈ V do
    u.color ← white
    u.d ← ∞
    u.p ← nil
for each vertex u ∈ V do
    if u.color = white then
        BFS-Visit(u)
```

```
BFS-Visit(G,s):
/*Assumes s is white*/

s.color ← gray
s.d ← 0
initialize an empty queue Q
Enqueue(Q,s)
while Q ≠ Ø do
    u ← Dequeue(Q)
    for each v ∈ Adj[u]
        if v.color = white then
            v.color ← gray
            v.d ← u.d + 1
            v.p ← u
            Enqueue(Q,v)
    u.color ← black
```

The old BFS(G,s) algorithm is renamed BFS-Visit(G,s).

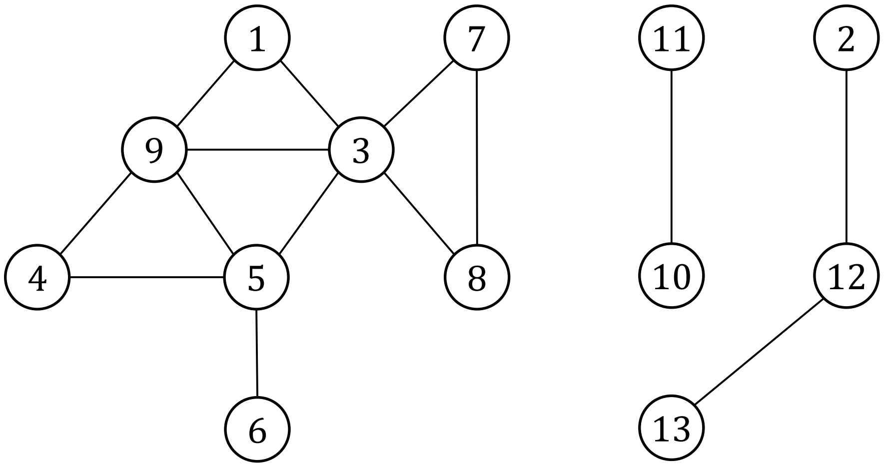
A new upper-level BFS(G) is created.

BFS(G) initializes all vertices to white (unvisited)

It then calls all vertices s, passing them to BFS-visit(s), **if s was not already seen while traversing a previously visited connected component.**

Connected Components

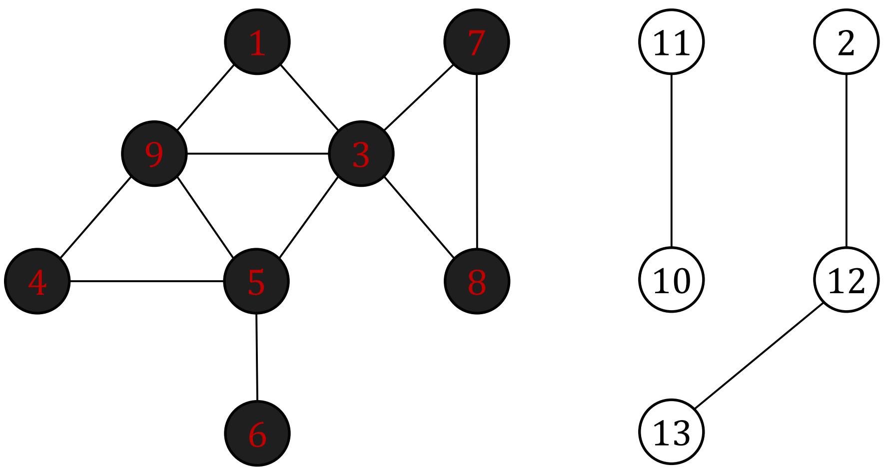
Connected component containing s. All nodes reachable from s.



BFS-Visit(1) would turn all nodes in leftmost component black

Connected Components

Connected component containing s. All nodes reachable from s.

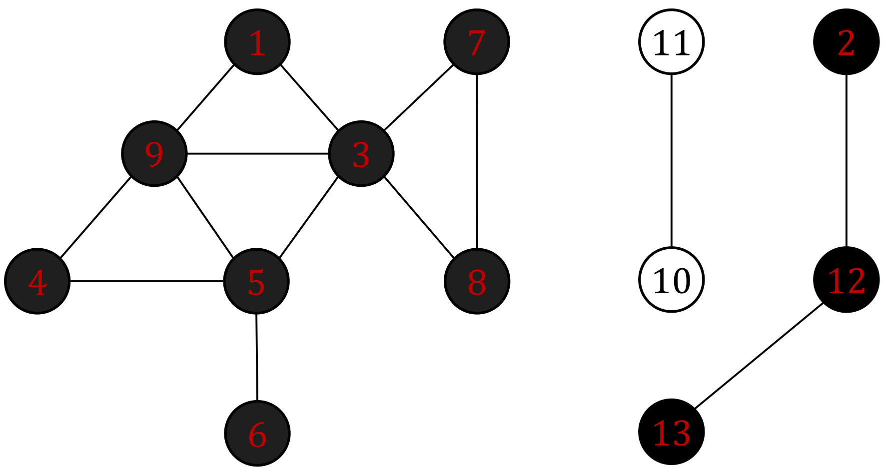


BFS-Visit(1) would turn all nodes in leftmost component black

BFS-Visit(2) would turn all nodes in rightmost component black

Connected Components

Connected component containing s. All nodes reachable from s.

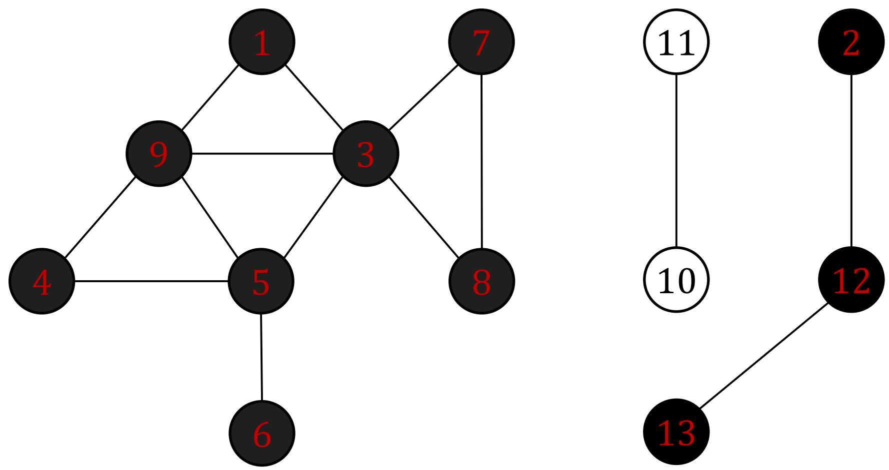


BFS-Visit(1) would turn all nodes in leftmost component black

BFS-Visit(2) would turn all nodes in rightmost component black

Connected Components

Connected component containing s. All nodes reachable from s.



BFS-Visit(1) would turn all nodes in leftmost component black

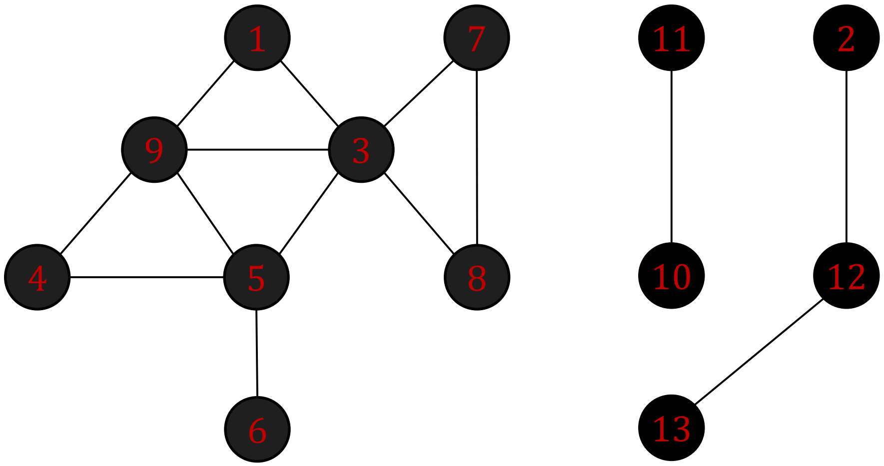
BFS-Visit(2) would turn all nodes in rightmost component black

BFS-Visit(i) for $3 \le i \le 9$ would do nothing.

BFS-Visit(10) would then turn all nodes in middle component black

Connected Components

Connected component containing s. All nodes reachable from s.



BFS-Visit(1) would turn all nodes in leftmost component black

BFS-Visit(2) would turn all nodes in rightmost component black

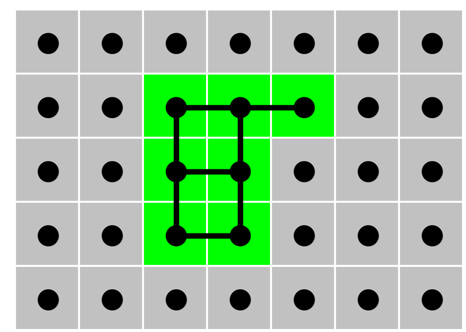
BFS-Visit(i) for $3 \leq i \leq 9$ would do nothing.
BFS-Visit(10) would then turn all nodes in middle component black

Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

recolor lime green blob to blue

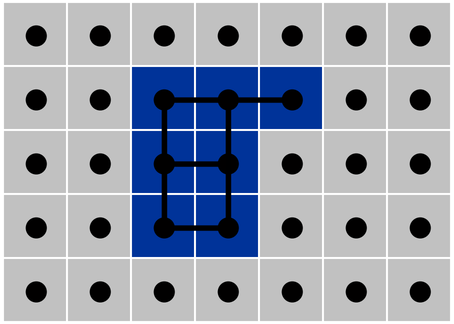


Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

recolor lime green blob to blue



s-t connectivity and shortest path in directed graphs

s-t connectivity (often called reachability for directed graphs). Given two nodes s and t, is there a path from s to t?

- Undirected graph: s can reach $t \Leftrightarrow t$ can reach s
- Directed graph: Not necessarily true

s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

- Undirected graph: p is the shortest path from s to $t \Leftrightarrow p$ is the shortest path from t to s
  - See *Six Degrees of Kevin Bacon*
- Directed graph: Not necessarily true

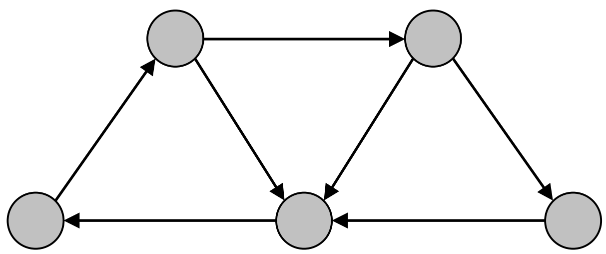
BFS on a directed graph. Same as in undirected case

- Ex: Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.

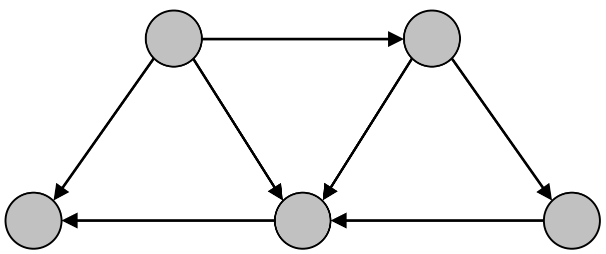
Strong Connectivity in Directed Graphs

Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.



strongly connected



not strongly connected

Definition: vertex s is "strong" in Graph G if,
for every vertex t, there is a path from s to t and from t to s.

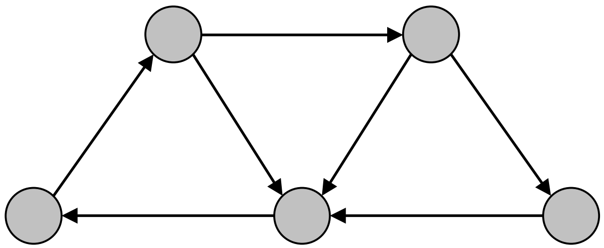
Observation 1: If graph G has a strong vertex s then
EVERY vertex in G is strong

Observation 2: A graph G is strongly connected
if and only if every vertex in G is strong

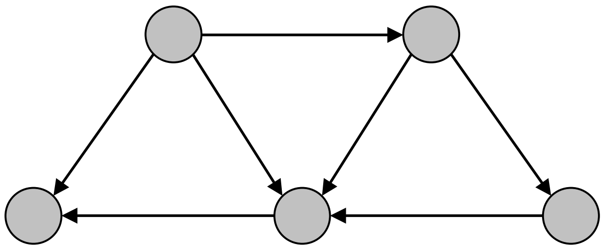
Strong Connectivity in Directed Graphs

Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.



strongly connected

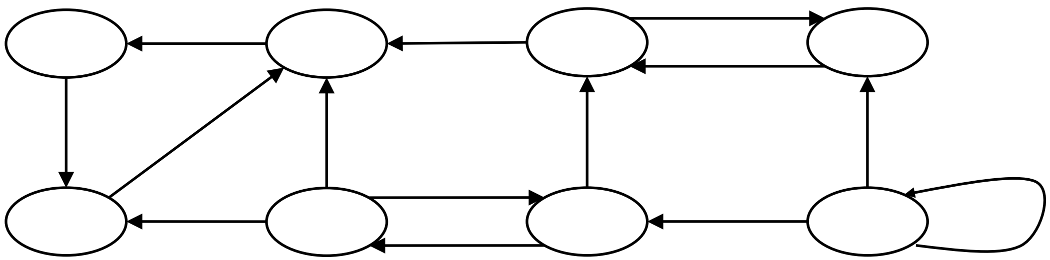


not strongly connected

Algorithm for checking strong connectivity

- Pick any node s.
- Run BFS from s in G.
- Reverse all edges in G, and run BFS from s.
- Return true iff all nodes reached in both BFS executions.

Strongly Connected Components

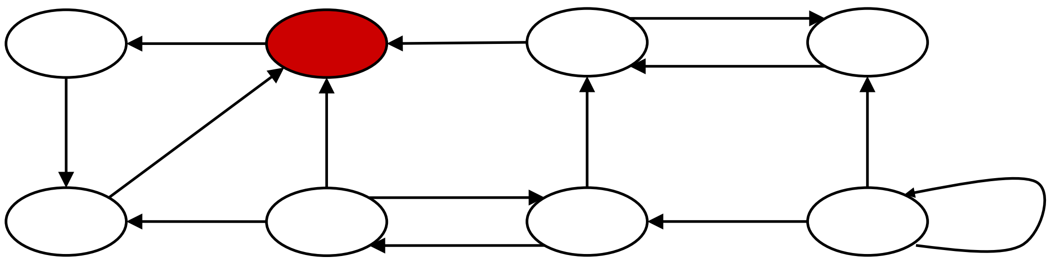


```
Strongly-Connected-Components(G):
create G^rev which is G with all edges reversed
while there are nodes left do
    u ← any node
    run BFS in G starting from u
    run BFS in G^rev starting from u
    C ← {nodes reached in both BFSs}
    output C as a strongly connected component
    remove C and its edges from G and G^rev
```

Running time: $O(VE)$

See text book for a $\Theta(V+E)$ algorithm (not required)

Strongly Connected Components

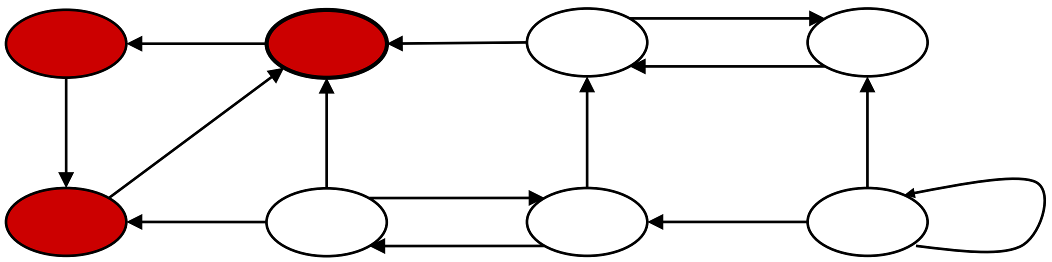


```
Strongly-Connected-Components(G):
create G^rev which is G with all edges reversed
while there are nodes left do
    u ← any node
    run BFS in G starting from u
    run BFS in G^rev starting from u
    C ← {nodes reached in both BFSs}
    output C as a strongly connected component
    remove C and its edges from G and G^rev
```

Running time: $O(VE)$

See text book for a $\Theta(V+E)$ algorithm (not required)

Strongly Connected Components

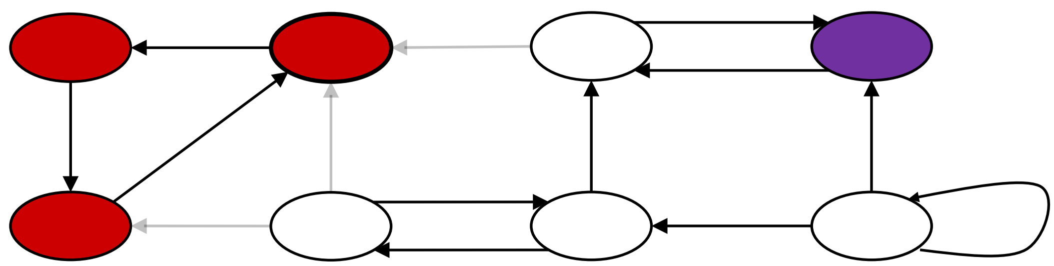


```
Strongly-Connected-Components(G):
create G^rev which is G with all edges reversed
while there are nodes left do
    u ← any node
    run BFS in G starting from u
    run BFS in G^rev starting from u
    C ← {nodes reached in both BFSs}
    output C as a strongly connected component
    remove C and its edges from G and G^rev
```

Running time: $O(VE)$

See text book for a $\Theta(V+E)$ algorithm (not required)

Strongly Connected Components

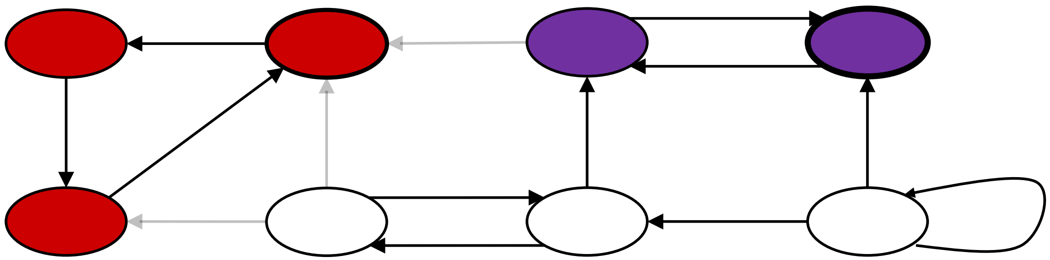


```
Strongly-Connected-Components(G):
create G^rev which is G with all edges reversed
while there are nodes left do
    u ← any node
    run BFS in G starting from u
    run BFS in G^rev starting from u
    C ← {nodes reached in both BFSs}
    output C as a strongly connected component
    remove C and its edges from G and G^rev
```

Running time: $O(VE)$

See text book for a $\Theta(V+E)$ algorithm (not required)

Strongly Connected Components

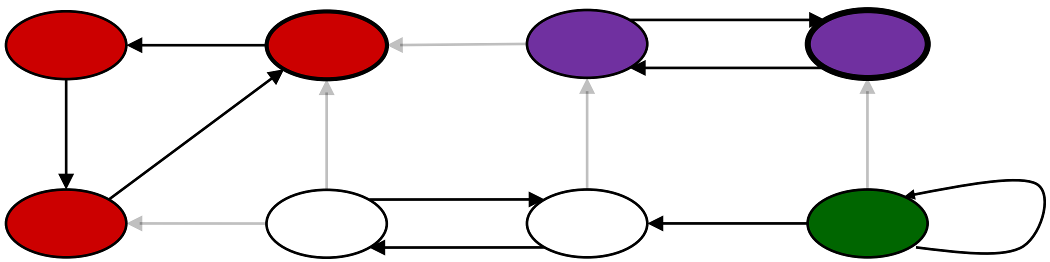


```
Strongly-Connected-Components(G):
create G^rev which is G with all edges reversed
while there are nodes left do
    u ← any node
    run BFS in G starting from u
    run BFS in G^rev starting from u
    C ← {nodes reached in both BFSs}
    output C as a strongly connected component
    remove C and its edges from G and G^rev
```

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)

Strongly Connected Components

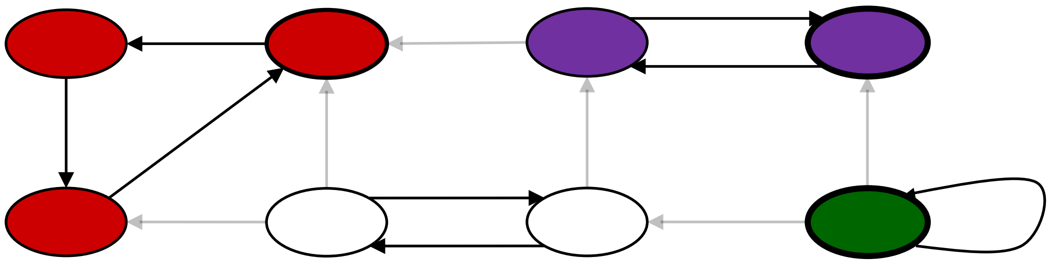


```
Strongly-Connected-Components(G):
create G^rev which is G with all edges reversed
while there are nodes left do
    u ← any node
    run BFS in G starting from u
    run BFS in G^rev starting from u
    C ← {nodes reached in both BFSs}
    output C as a strongly connected component
    remove C and its edges from G and G^rev
```

Running time: $O(VE)$

See text book for a $\Theta(V+E)$ algorithm (not required)

Strongly Connected Components

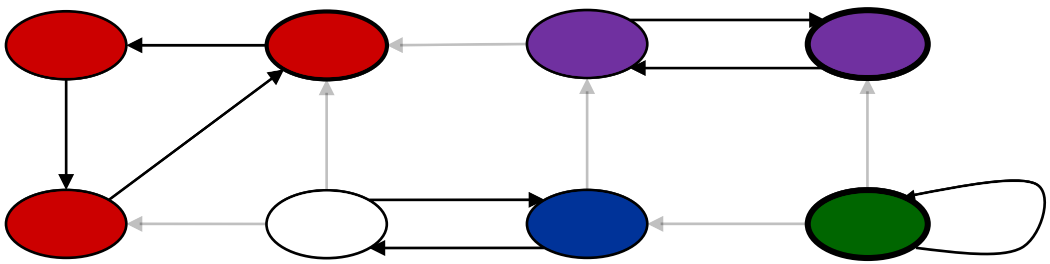


```
Strongly-Connected-Components(G):
create G^rev which is G with all edges reversed
while there are nodes left do
    u ← any node
    run BFS in G starting from u
    run BFS in G^rev starting from u
    C ← {nodes reached in both BFSs}
    output C as a strongly connected component
    remove C and its edges from G and G^rev
```

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)

Strongly Connected Components

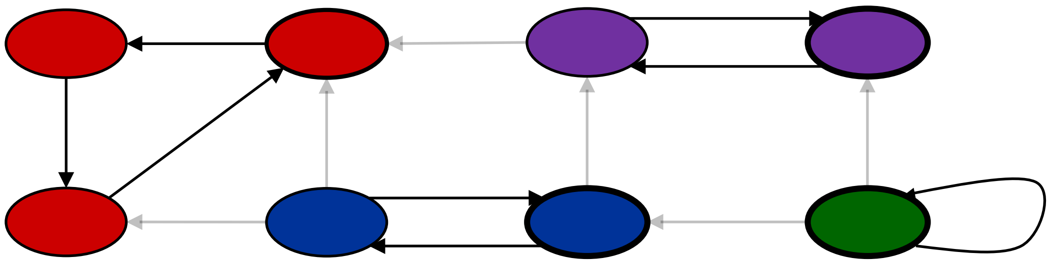


```
Strongly-Connected-Components(G):
create G^rev which is G with all edges reversed
while there are nodes left do
    u ← any node
    run BFS in G starting from u
    run BFS in G^rev starting from u
    C ← {nodes reached in both BFSs}
    output C as a strongly connected component
    remove C and its edges from G and G^rev
```

Running time: $O(VE)$

See text book for a $\Theta(V+E)$ algorithm (not required)

Strongly Connected Components

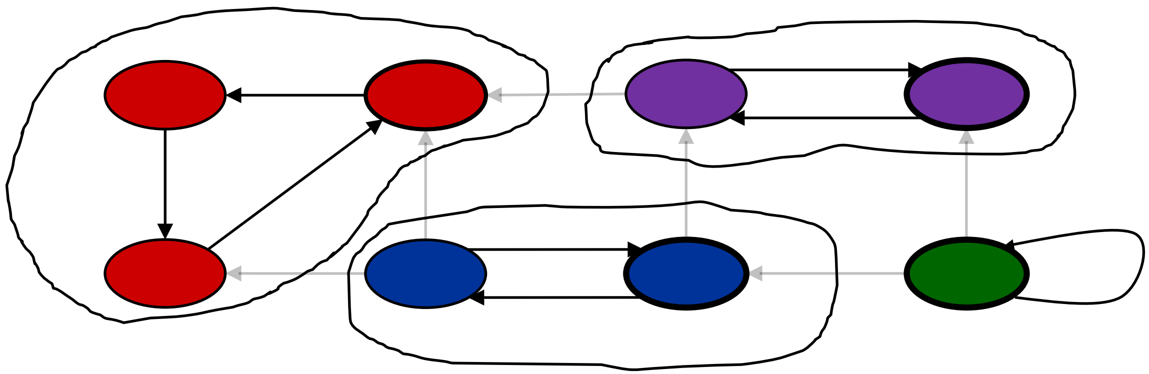


```
Strongly-Connected-Components(G):
create G^rev which is G with all edges reversed
while there are nodes left do
    u ← any node
    run BFS in G starting from u
    run BFS in G^rev starting from u
    C ← {nodes reached in both BFSs}
    output C as a strongly connected component
    remove C and its edges from G and G^rev
```

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)

Strongly Connected Components



<u>Strongly-Connected-Components(G):</u>
create G^{rev} **which is** G **with all edges reversed**
while there are nodes left do
    $u \leftarrow$ **any node**
    run BFS in G **starting from** u
    run BFS in G^{rev} **starting from** u
    $C \leftarrow \{$**nodes reached in both BFSs**$\}$
    output C **as a strongly connected component**
    remove C **and its edges from** G **and** G^{rev}

Running time: $O(VE)$

See text book for a $\Theta(V+E)$ algorithm (not required)

Depth First Search and DFS Tree

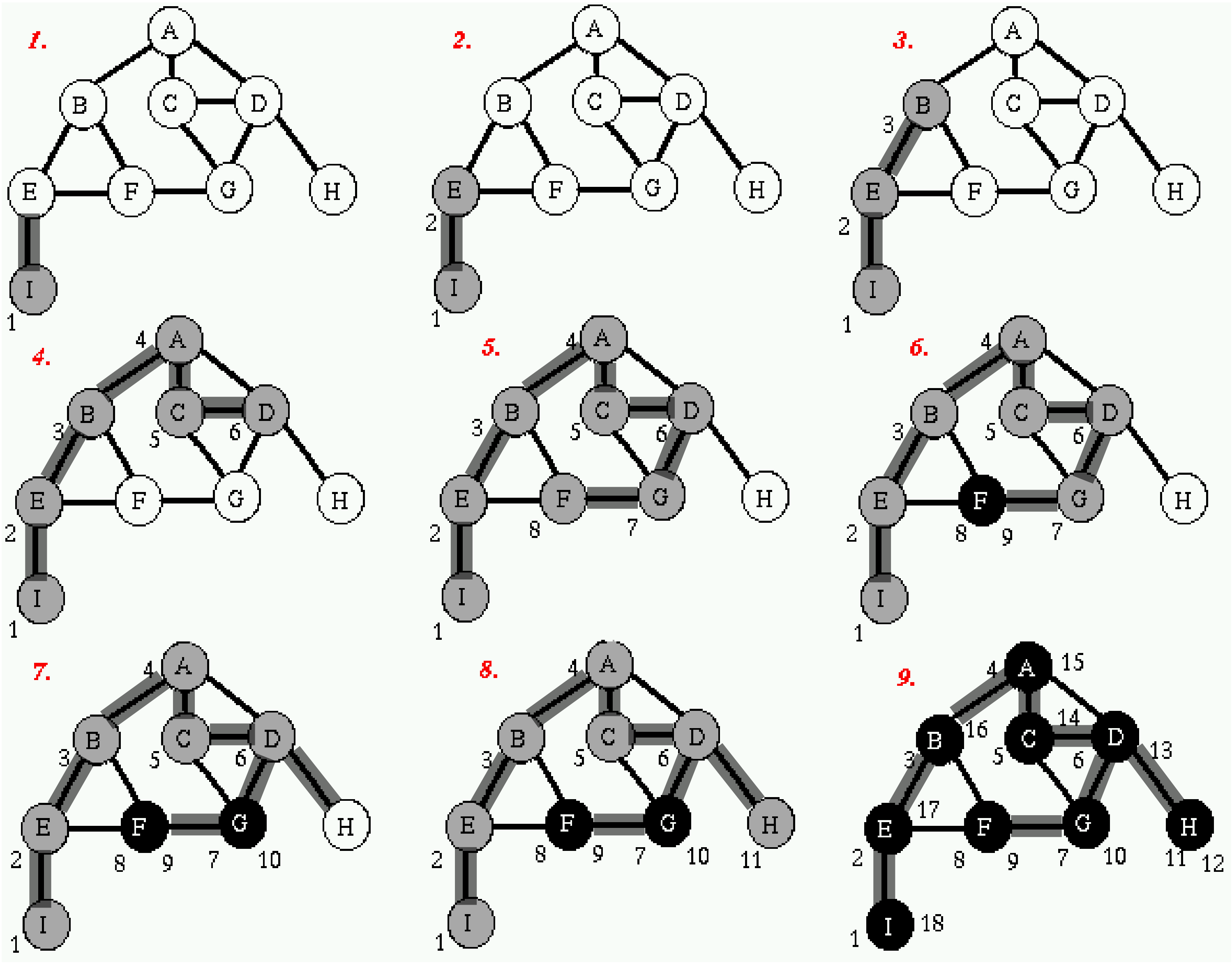
- Breadth first search is "Broad".
  - It builds a wide tree, connecting a node to ALL of the neighbors that have not yet been processed.
  - Once a node starts being processed, it sees ALL of its neighbors before any other node is processed

- There is another procedure, called DEPTH first search.
  - Instead of going broad, it goes DEEP
  - It recursively searches deep into the tree

  - When a node *u* is processed, it looks at each of its neighbors in order
    - At the time u checks a neighbor *v*, DFS starts processing *v* (which starts processing it's children, which start processing their children, etc.).

    - Only after all of *v*'s descendants have been processed does *u* go on to process its next neighbor

Depth First Search and DFS Tree



DFS Algorithm

```
DFS(G):
for each vertex u ∈ V do
      u.color ← white
      u.p ← nil
for each vertex u ∈ V do
      if u.color = white then
            DFS-Visit(u)
………….
```

Colors:

- **White**: undiscovered
- **Gray**: discovered, but neighbors not fully explored (on recursion stack)
- **Black**: discovered and neighbors fully explored

Parent pointers:

- Pointing to the node that leads to its discovery
- The pointers form a tree, rooted at s

- DFS(G) calls the DFS-visit search on each vertex u
- Before DFS-Visit(u) returns, all nodes in the connected component containing u are turned black (will see later)
- So DFS-Visit will only be called once for each connected component in G

DFS Algorithm

```
DFS(G):
for each vertex u ∈ V do
    u.color ← white
    u.p ← nil
for each vertex u ∈ V do
    if u.color = white then
        DFS-Visit(u)

DFS-Visit(u):
u.color ← gray
for each v ∈ Adj[u] do
    if v.color = white then
        v.p ← u
        DFS-Visit(v)
u.color ← black
```

Running time: $\Theta(V+E)$

Colors:

- **White:** undiscovered
- **Gray:** discovered, but neighbors not fully explored (on recursion stack)
- **Black:** discovered and neighbors fully explored

Parent pointers:

- Pointing to the node that leads to its discovery
- The pointers form a tree, rooted at s

Application: Cycle Detection

Problem: Given an undirected graph $G = (V, E)$, check if it contains a cycle.

Idea:

- A tree (connected and acyclic) contains **exactly** $V - 1$ edges.
- If it has fewer edges, it cannot be connected.
- If it has more edges, it must contain a cycle.

Algorithm:

- Run BFS/DFS to find all the connected components of G.
- For each connected component, count the number of edges.
- If # edges $\geq$ # vertices, return "cycle detected".

Running time: $\Theta(V + E)$

Q: What if we also want to **find** a cycle (any is OK) if it exists?

Tree edges, back edges, and cross edges

After we have run BFS or DFS on an undirected graph, the edges can be classified into 3 types:

- Tree edges: traversed by the BFS/DFS.
- Back edges: connecting a node with one of its ancestors in the BFS/DFS-tree (other than its parent).
- Cross edges: connecting two nodes with no ancestor/descendent relationship.

Theorem: In a DFS on an undirected graph, there are no cross edges.

Pf: Consider any edge (u, v) in G.

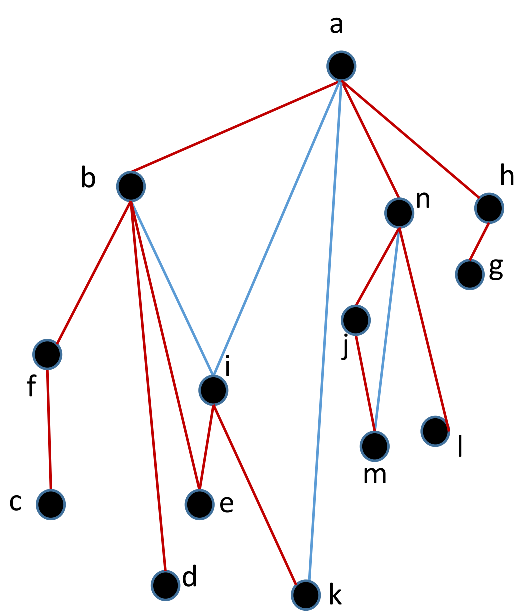
- Without loss of generality, assume u is discovered before v.
- Then v is discovered while u is gray (why?).
- Hence v is in the DFS subtree rooted at u.
  - If $v.p = u$, then (u, v) is a tree edge.
  - If $v.p \neq u$, then (u, v) is a back edge.

Theorem: In a BFS on an undirected graph, there are no back edges.
(Not proven)

DFS for cycle detection

Idea: Run DFS on each connected component of G.

- If (u, v) is a back edge.
  - => v is an ancestor (but not parent) of u in the DFS trees.
    =>There is thus a path from v to u in the DFS-tree and
  - => v to u plus back edge (u, v) creates a cycle.
- If no back edge exists then only contains (DFS) tree edges
  - => the graph is a forest, and hence is acyclic.



- In DFS starting at a,
  (i,b) was first back edge found
- => b was ancestor (not parent) of i in tree
- => tree contains path (b->e->i) from b to i
- => this path plus edge (i ,b) is the cycle
  b->e->i->b

DFS for cycle detection

```
CycleDetection(G):
for each vertex u ∈ V do
      u.color ← white
      u.p ← nil
for each vertex u ∈ V do
      if u.color = white then DFS-Visit(u)
return "No cycle"

DFS-Visit(u):
u.color ← gray
for each v ∈ Adj[u] do
      if v.color = white then
            v.p ← u
            DFS-Visit(v)
      else if v ≠ u.p then
            output "Cycle found:"
            while u ≠ v do
                  output u
                  u ← u.p
            output v
            return
u.color ← black
```

Running time: $\Theta(V)$

- Only traverse DFS-tree edges, until the first non-tree edge is found
- At most $V - 1$ tree edges