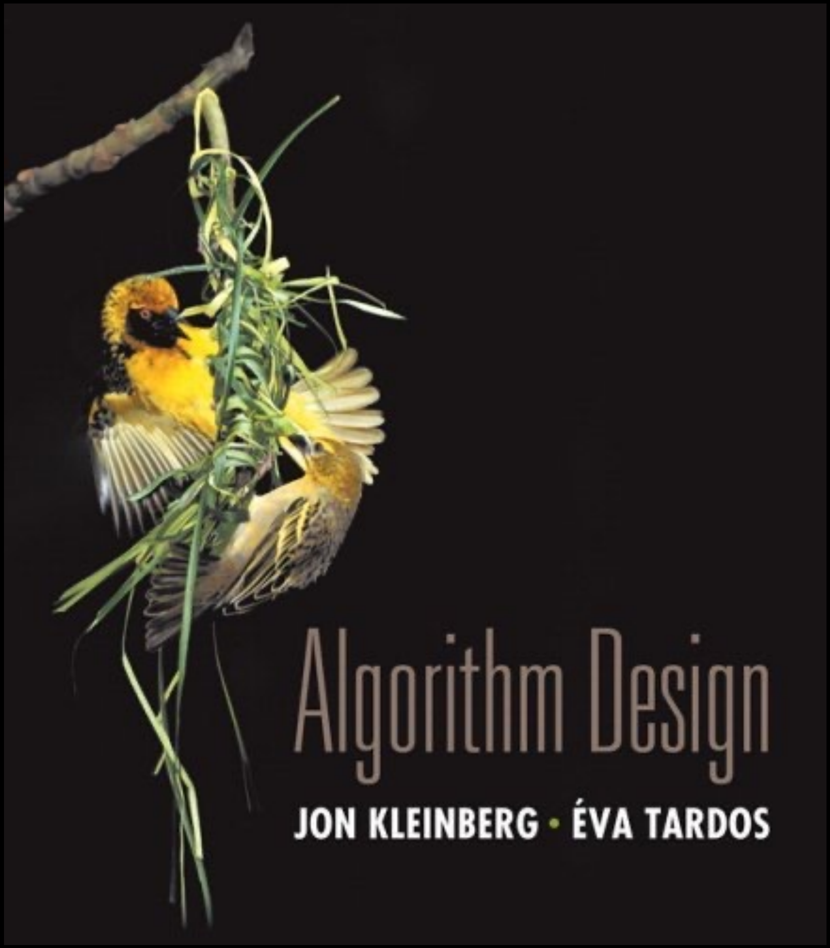


Chapter 3

Graphs

Slides by Kevin Wayne.

Overview

- Basic Definitions and Applications
- Graph Traversal
  - BFS and DFS
- Applications
  - Connected component
  - Testing bipartite

- Connectivity in directed graph
- DAGs and topological ordering

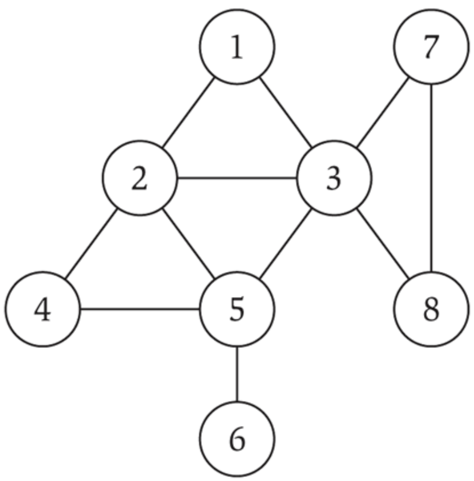
3.1 Basic Definitions and Applications

Annotated and slightly changed by Yang Xu (徐炀)
Contact: xuyang@sustech.edu.cn

Undirected Graphs

Undirected graph. G = (V, E)

- V = nodes. (Vertex, vertices)
- E = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: n = |V|, m = |E|.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

n = 8

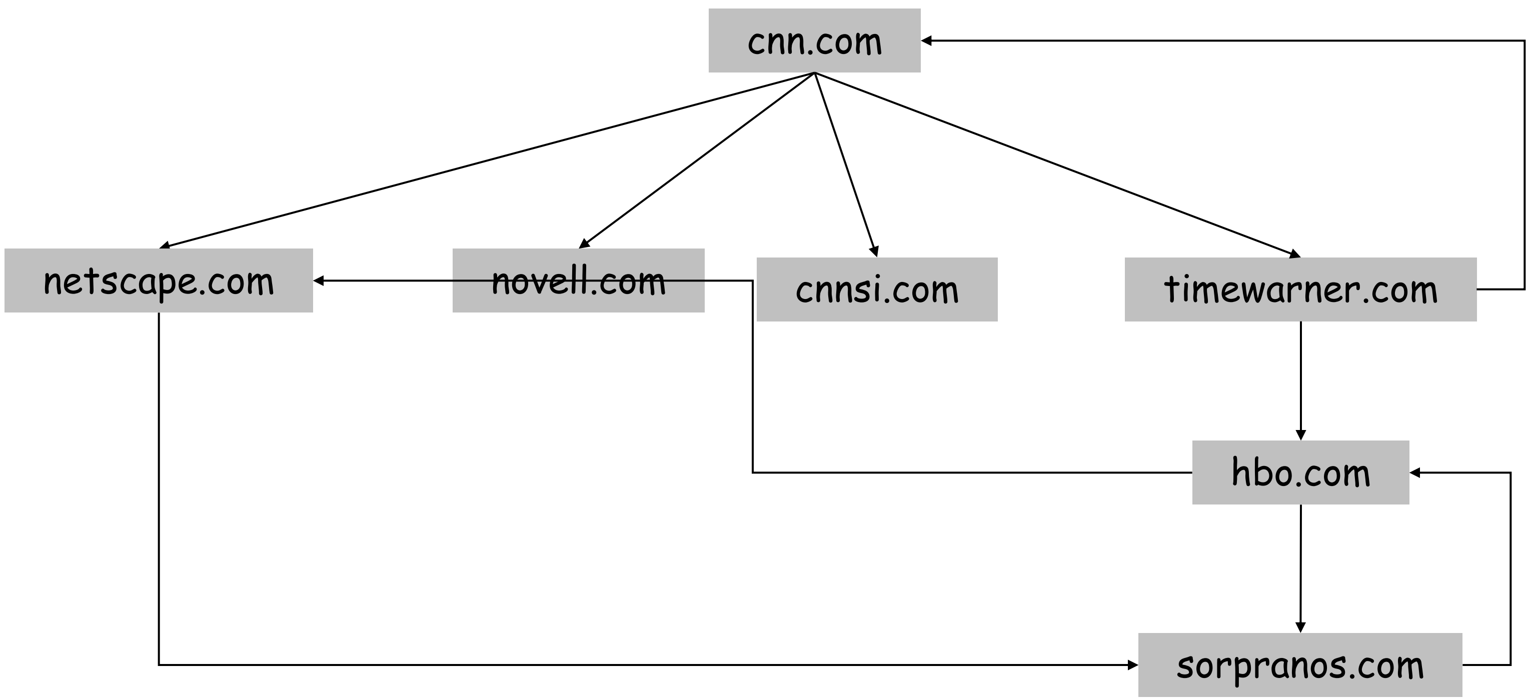
m = 11

Some Graph Applications

Graph	*Nodes*	*Edges*
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires
Computational graph	Tensor, matrix, vector, .	Function arguments

World Wide Web

Web graph.
- Node: web page.
- Edge: hyperlink from one page to another.



9-11 Terrorist Network

Social network graph.

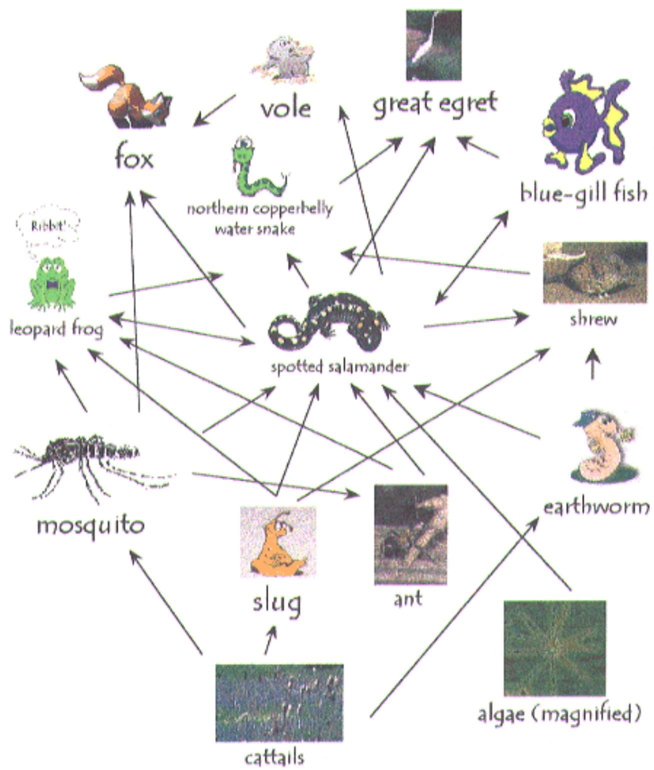
- Node: people.
- Edge: relationship between two people.

Reference: Valdis Krebs, http://www.firstmonday.org/issues/issue7_4/krebs

Ecological Food Web

Food web graph.

- Node = species.
- Edge = from prey to predator.

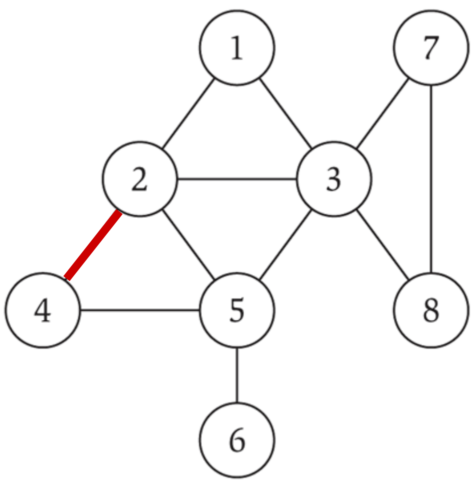


Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.

- Two representations of each edge.
- Space proportional to n^2.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.



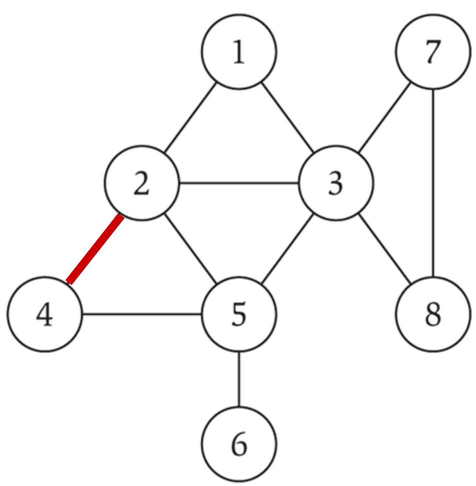
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

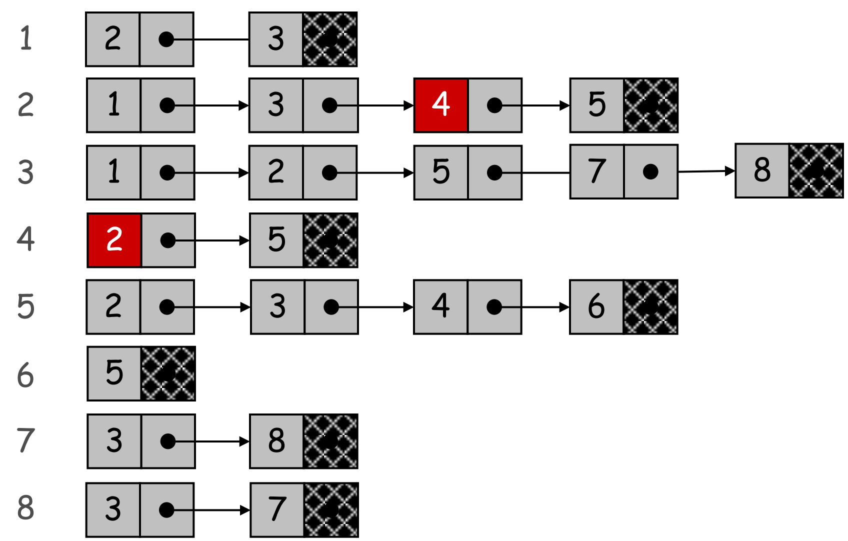
Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- Two representations of each edge.
- Space proportional to m + n.
- Checking if (u, v) is an edge takes O(deg(u)) time.
- Identifying all edges takes Θ(m + n) time.

degree = number of neighbors of u

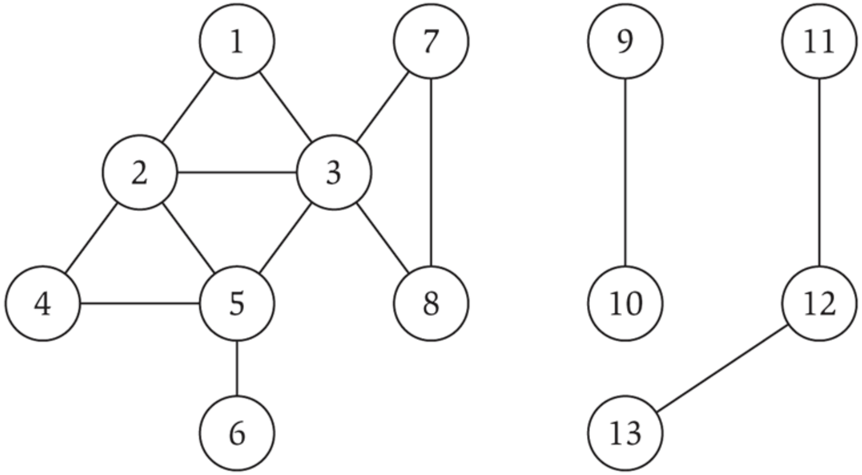




Def. A path in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.

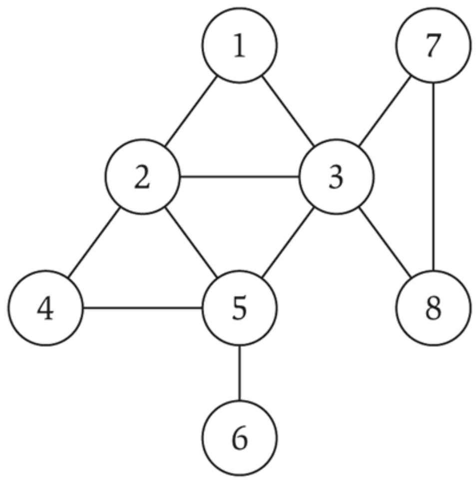
Def. A path is simple if all nodes are distinct.

Def. An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v.



Cycles

Def. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first k-1 nodes are all distinct.



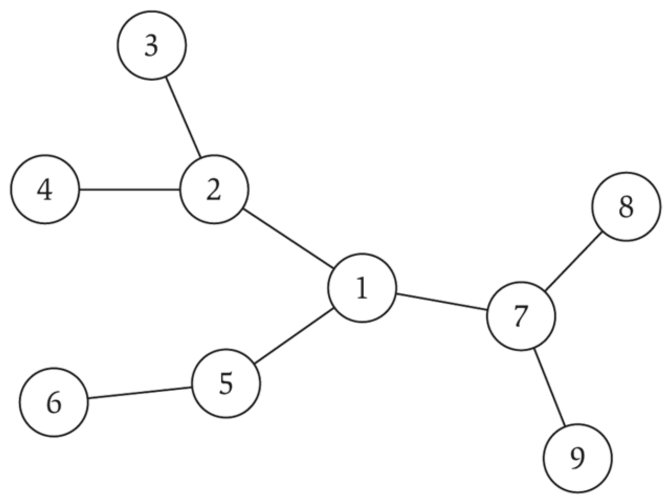
cycle C = 1-2-4-5-3-1

Trees

Def. An undirected graph is a tree if it is connected and does not contain a cycle.

Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.

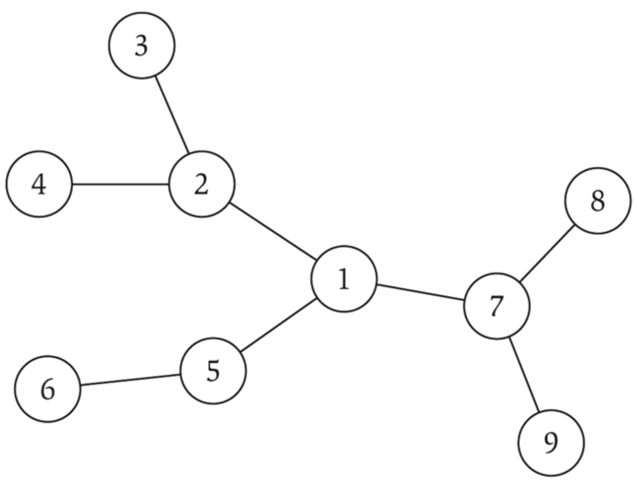
- G is connected.
- G does not contain a cycle.
- G has n-1 edges.



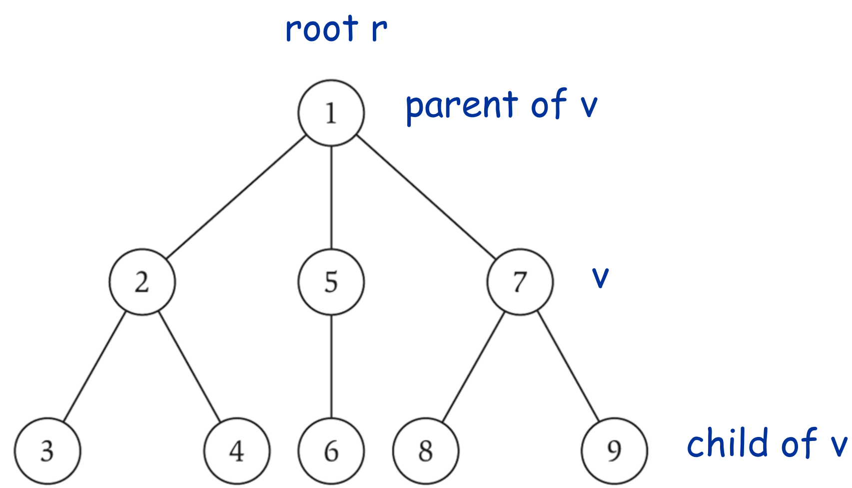
Rooted Trees

Rooted tree. Given a tree T, choose a root node r and orient each edge away from r.

Importance. Models hierarchical structure.



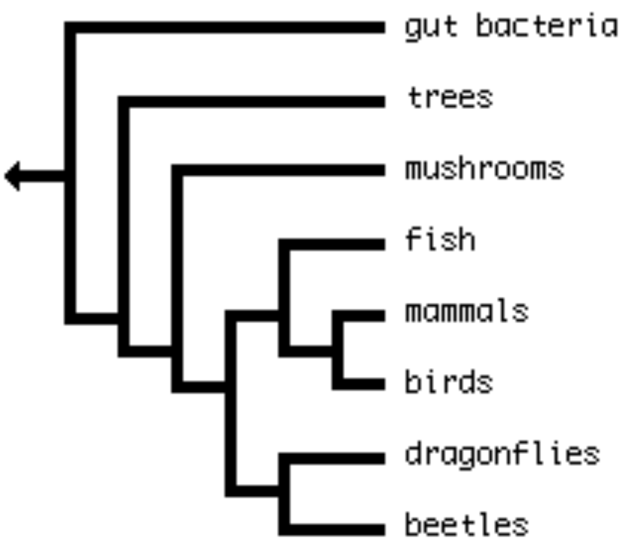
a tree



the same tree, rooted at 1

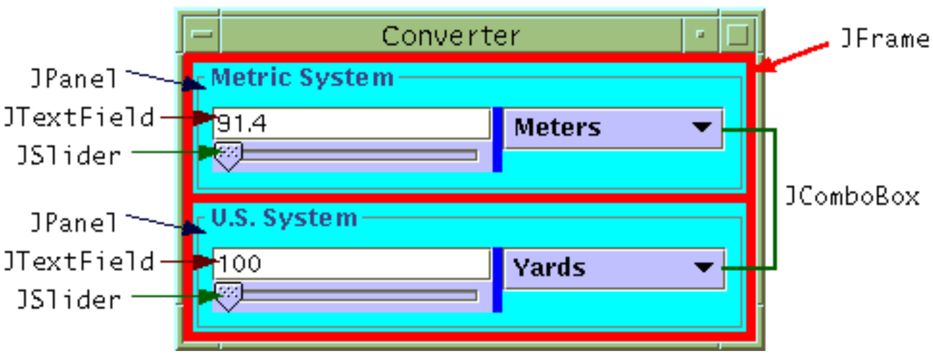
Phylogeny Trees

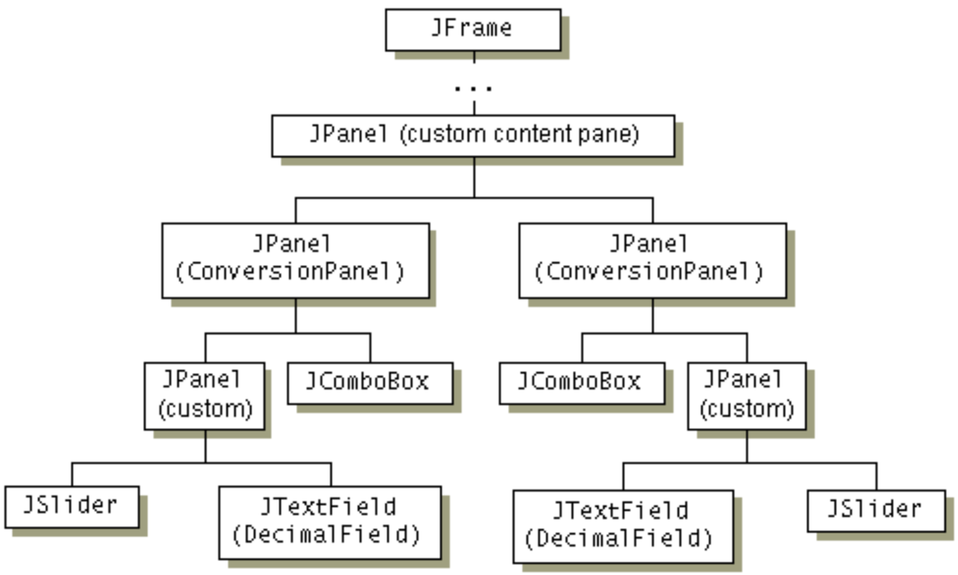
Phylogeny trees. Describe evolutionary history of species.



GUI Containment Hierarchy

GUI containment hierarchy. Describe organization of GUI widgets.





Reference: http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html

3.2 Graph Traversal

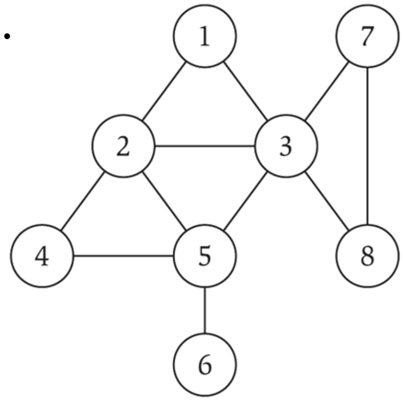
Connectivity

s-t connectivity problem. Given two node s and t, is there a path between s and t?

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

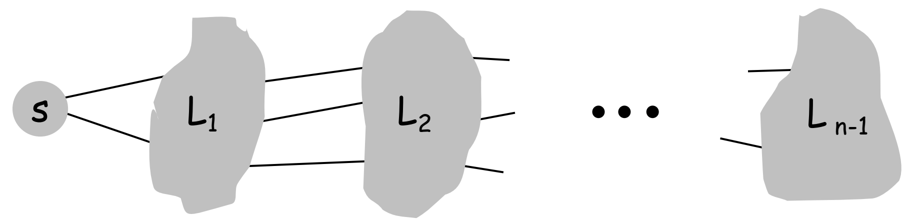
Applications.

- Friendster. (a social networks website)
- Maze traversal.
- Kevin Bacon number.
- Fewest number of hops in a communication network.



Breadth First Search

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



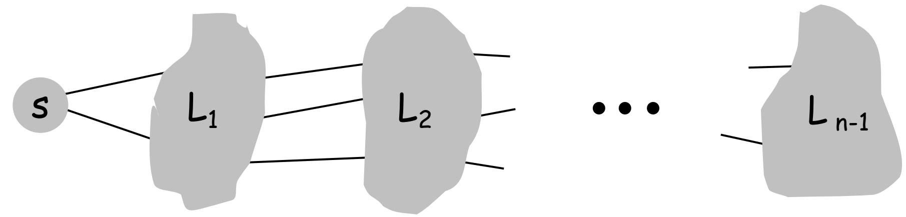
BFS algorithm.

- $L_0 = \{ s \}$.
- 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.

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 Implementation

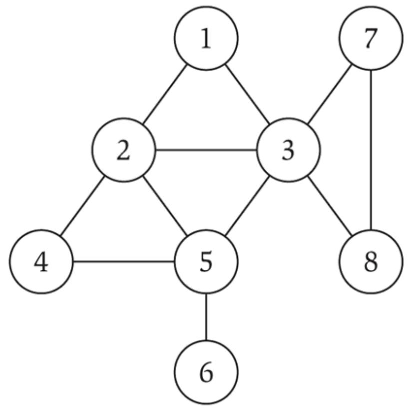
Textbook version, page 90

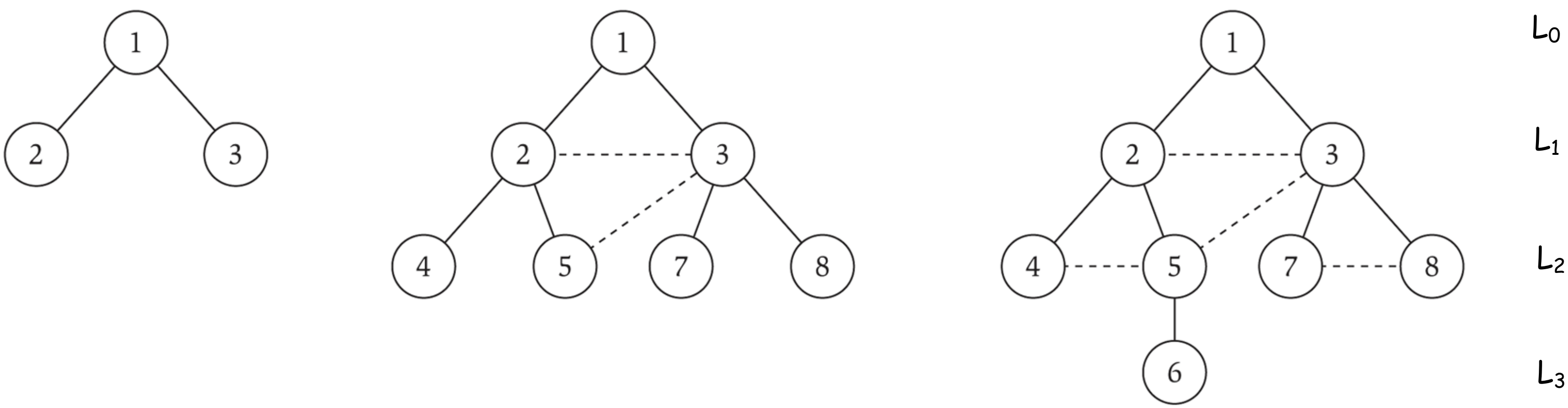



```
BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v
  Initialize L[0] to consist of the single element s
  Set the layer counter i = 0
  Set the current BFS tree T = Ø
  While L[i] is not empty
    Initialize an empty list L[i + 1]
    For each node u ∈ L[i]
      Consider each edge (u, v) incident to u
      If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Add v to the list L[i + 1]
      Endif
    Endfor
    Increment the layer counter i by one
  Endwhile
```

Breadth First Search

Property. Let T be a BFS tree of G = (V, E), and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.





(a) (b) (c)

A Clearer Implementation

MIT book, page 594

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

All vertices start out *white*.

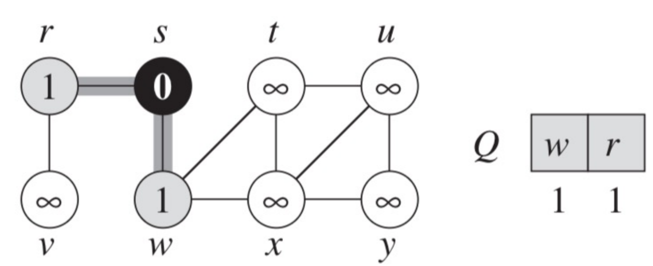
A vertex becomes non-white the first time it is discovered.

Gray and ***black*** vertices are already discovered:

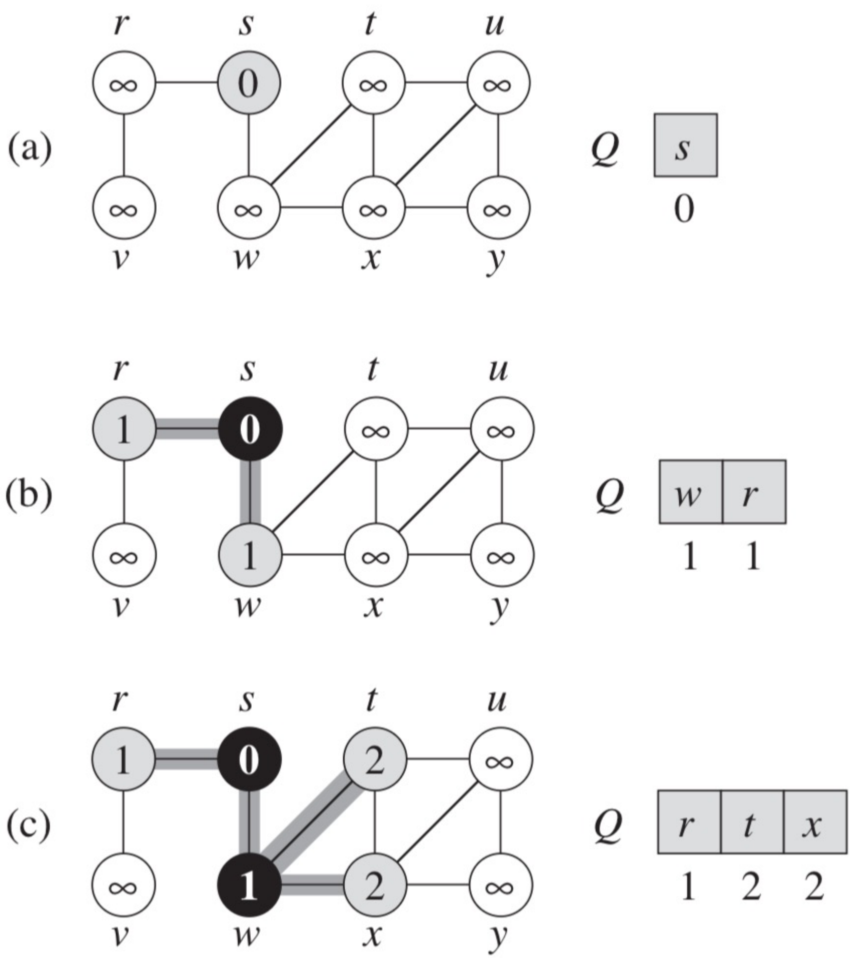
- ***Gray*** vertices may have adjacent white vertices.
- All vertices adjacent to ***black*** vertices are discovered.

Gray vertices represent the *frontier* between <u>discovered</u> and <u>undiscovered</u> vertices.

(b)

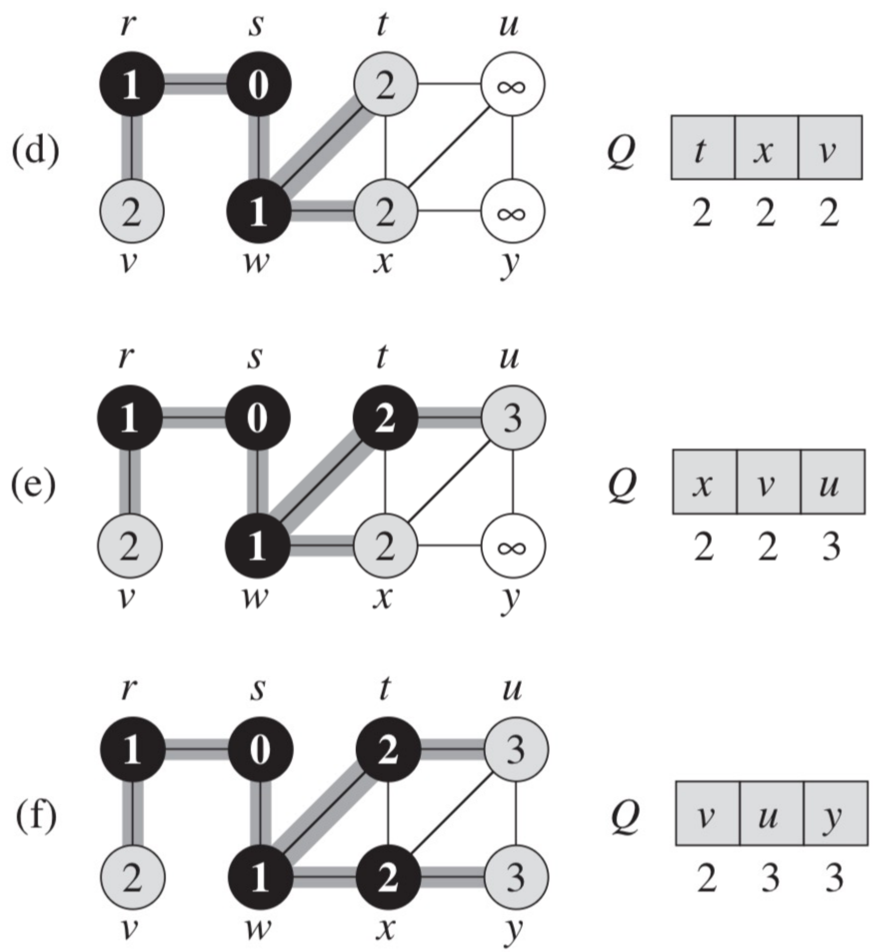


BFS Example



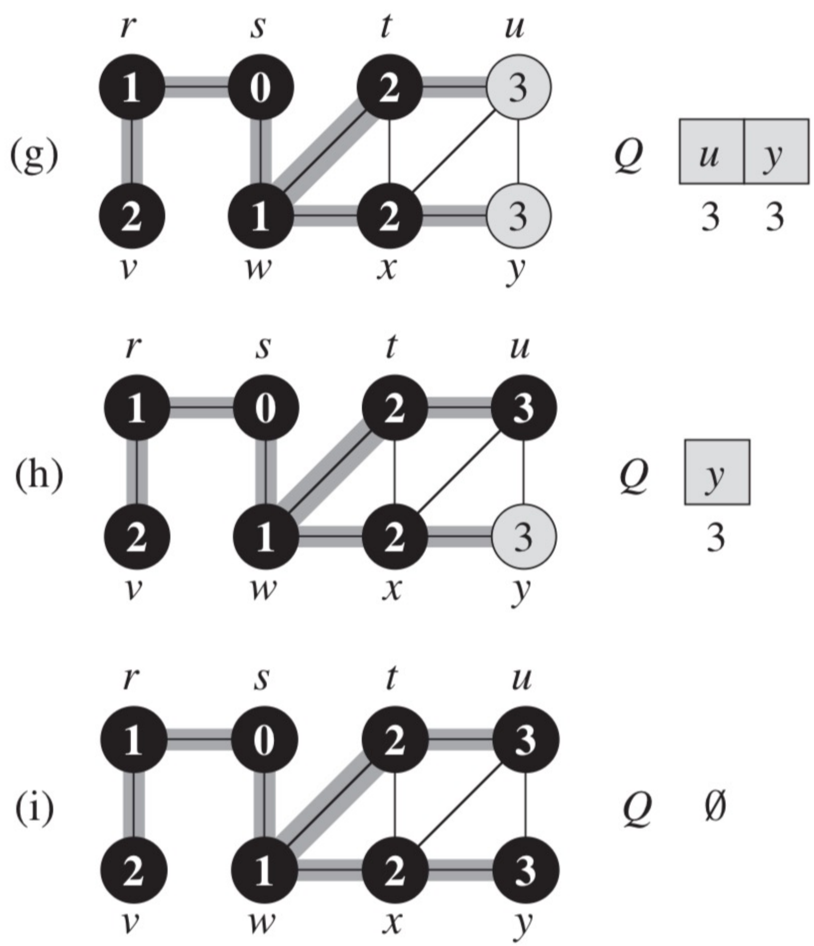

```
while Q ≠ Ø
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

BFS Example




```
while Q ≠ Ø
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

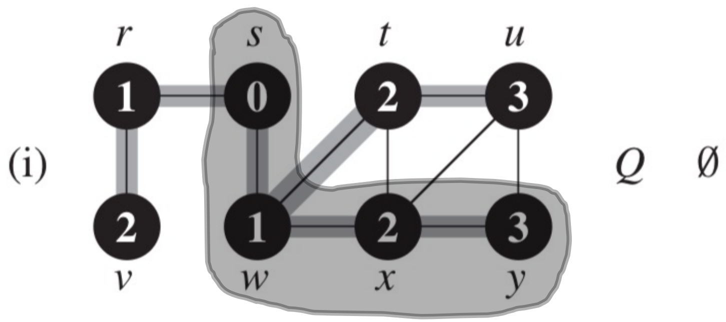
BFS Example

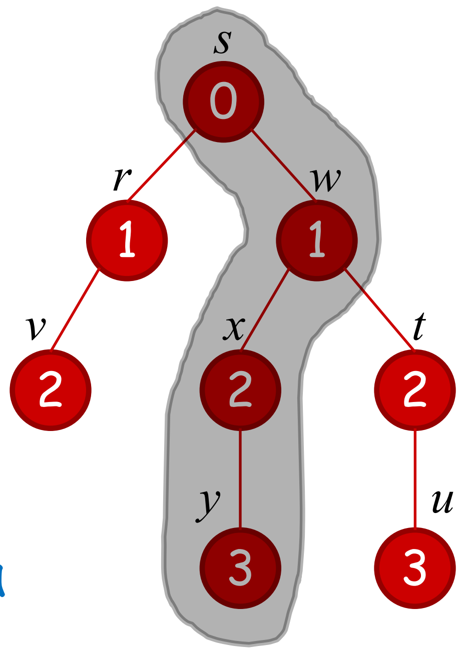



```
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Output of BFS

BFS builds a tree as it searches the graph.





A simple path from s to each vertex is a ***shortest path*** in G.

Breadth First Search: Analysis

Theorem. BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Pf.

- Easy to prove $O(n^2)$ running time:
  - at most n lists L[i]
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node u, there are $\leq n$ incident edges (u, v), and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
  - when we consider node u, there are deg(u) incident edges (u, v)
  - total time processing edges is $\sum_{u \in V} \deg(u) = 2m$ ■

each edge (u, v) is counted exactly twice
in sum: once in deg(u) and once in deg(v)

Breadth First Search: Analysis

Each vertex is enqueued at most once,
and hence dequeued at most once.
=> Total time for queue ops is $O(n)$

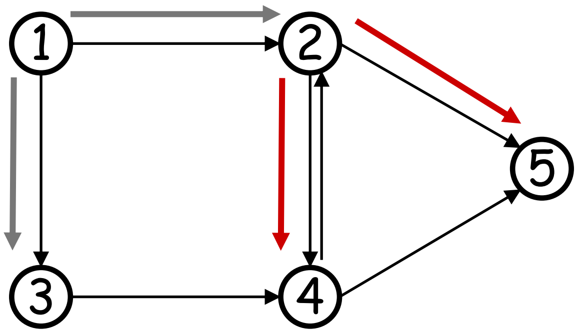
The for loop (line 12) scans the adjacency list of each vertex only when it is dequeued, each adjacency list is scanned only once
=> Total time for scanning is $O(E)$

```
BFS(G, s)
1   for each vertex u ∈ G.V − {s}
2       u.color = WHITE
3       u.d = ∞
4       u.π = NIL
5   s.color = GRAY
6   s.d = 0
7   s.π = NIL
8   Q = ∅
9   ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Depth First Search

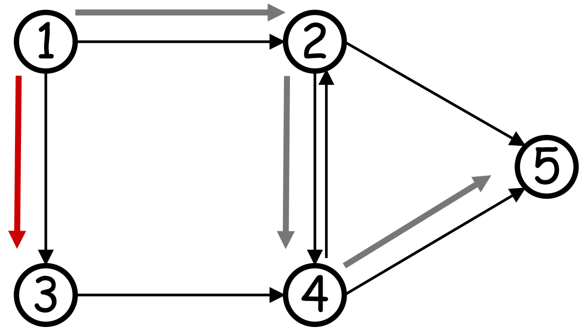
Difference from BFS?

BFS



Sequence of being discovered:
1, 2, 3, 4, 5

DFS



Sequence of being discovered: 1, 2, 4, 5, 3

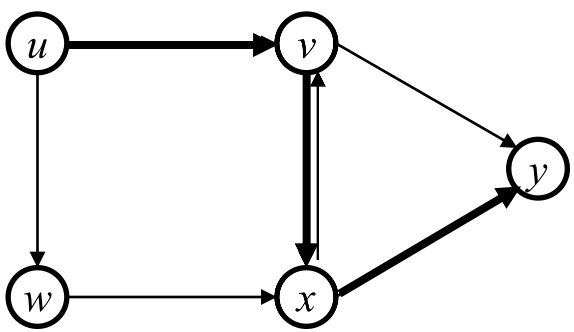
```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

Textbook page 84

DFS Implementations

A simple recursive version

```
DFS-RECURSIVE(G, s)
    s.visited = true
    for v in G.adj[s]:
        if v.visited == false
            DFS-RECURSIVE(G, v)
```



DFS-RECURSIVE(G, u)

$u.visited = true$

DFS-RECURSIVE(G, v)

$v.visited = true$

DFS-RECURSIVE(G, x)

$x.visited = true$

DFS-RECURSIVE(G, y)

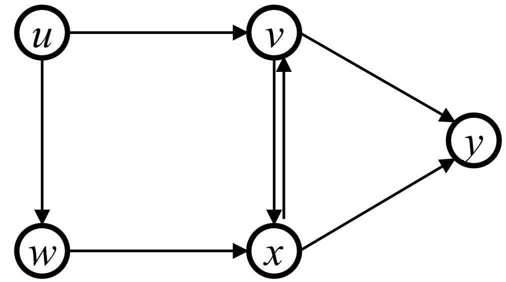
$y.visited = true$

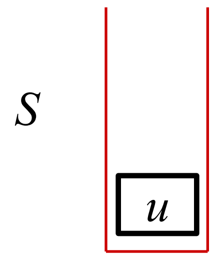
DFS Implementations

A simple non-recursive version

DFS-ITERATIVE(G, s)

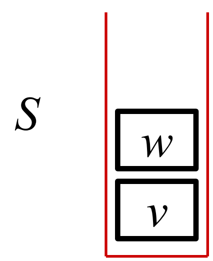
```
let S be a stack
S.push(s)
while S ≠ Ø
        u = S.pop()
        if u.visited == false:
                u.visited = true
                for v in G.adj[u]:
                        S.push(v)
```



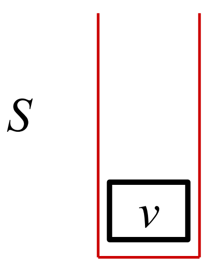


$u = S.\text{pop}()$

$u.visited = \text{true}$



$S.\text{push}(v)$, $S.\text{push}(w)$

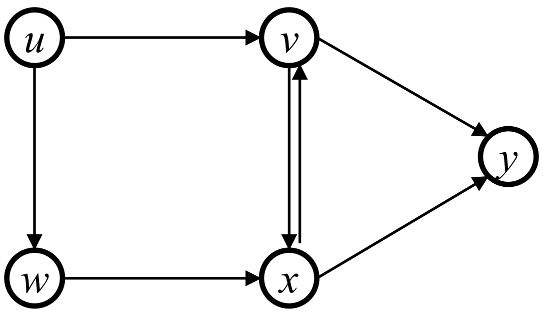


$w = S.\text{pop}()$

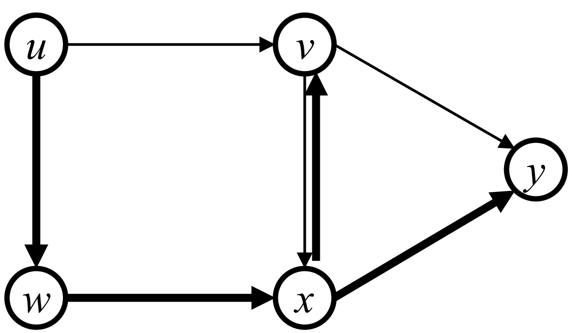
$w.visited = \text{true}$

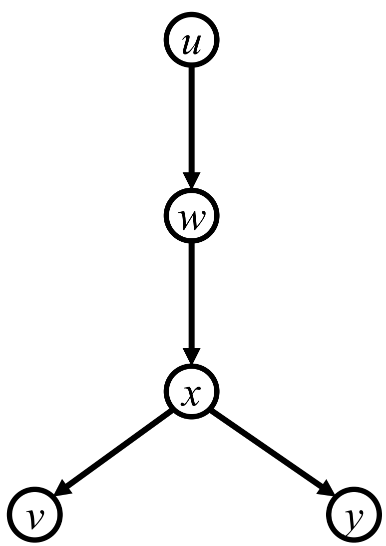
Output of DFS

A DFS tree (or forest)

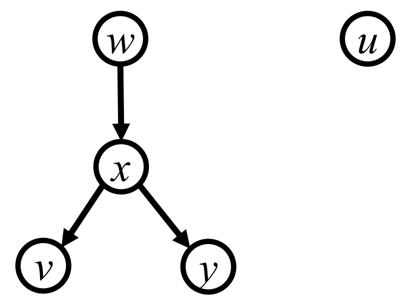


Assume that in the call of DFS(G), DFS-RECURSIVE(G, ***w***) is first called.





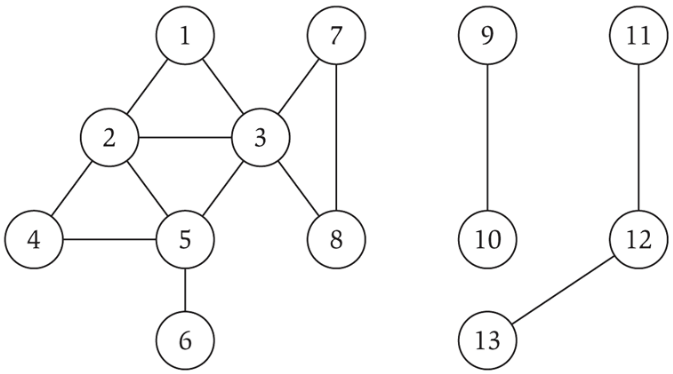
Resulting forest:



3.3 Application 1: Connected Component

Connected component. Find all nodes reachable from s.

Textbook page 82, Exploring a Connected Component

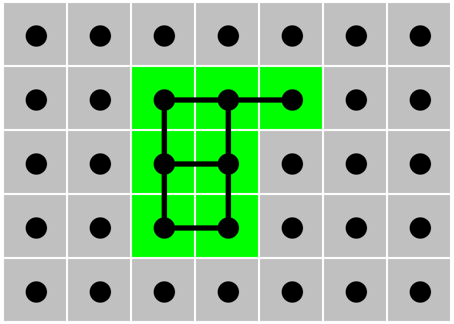


Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }.

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

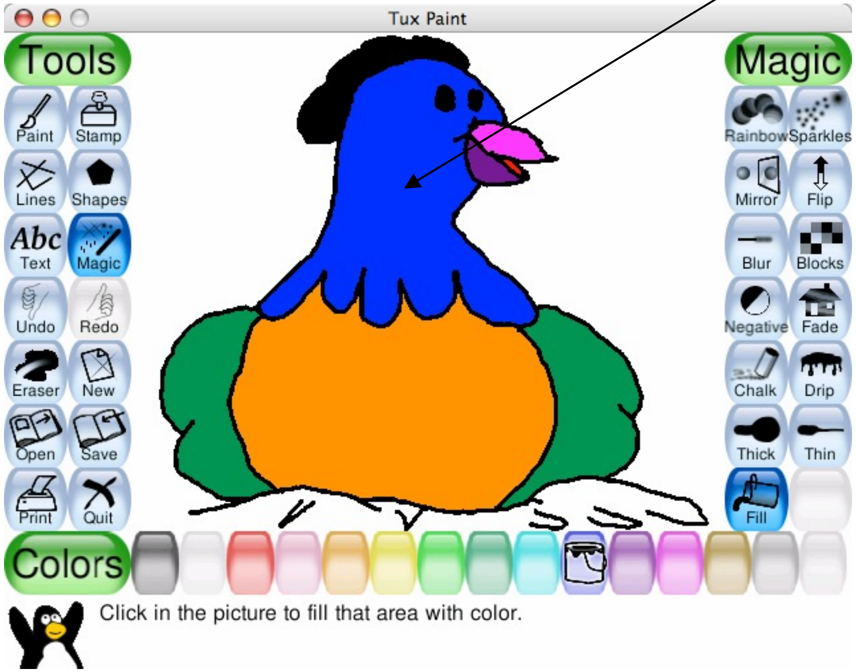


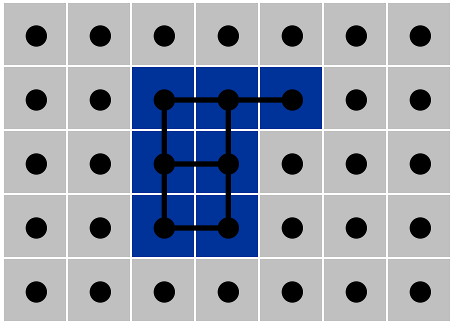
Flood Fill

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



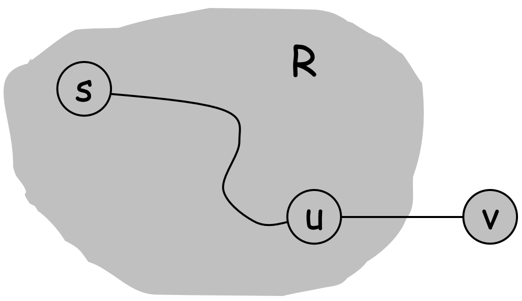


Connected Component

Connected component. Find all nodes reachable from s.

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R
Endwhile

Textbook page 82



it's safe to add v

Theorem. Upon termination, R is the connected component containing s.

- BFS = explore in order of distance from s.
- DFS = explore in a different way.

Both BFS and DFS can find the connected component

3.4 Application 2: Testing Bipartiteness

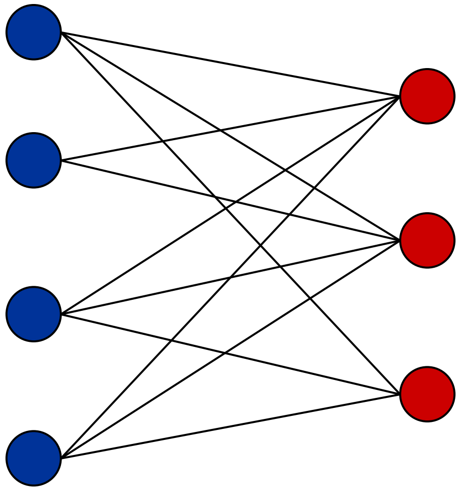
（二分性）

Bipartite Graphs

Def. An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications.

- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.

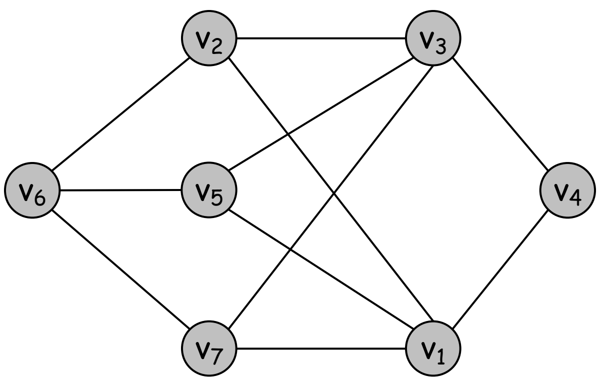


a bipartite graph

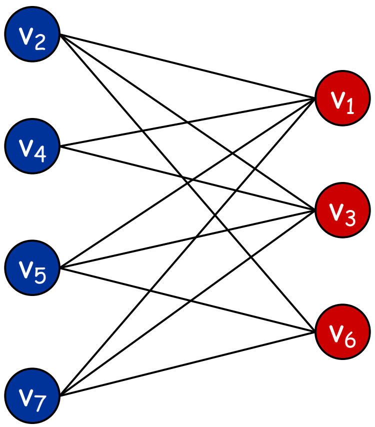
Testing Bipartiteness

Testing bipartiteness. Given a graph G, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



a bipartite graph G



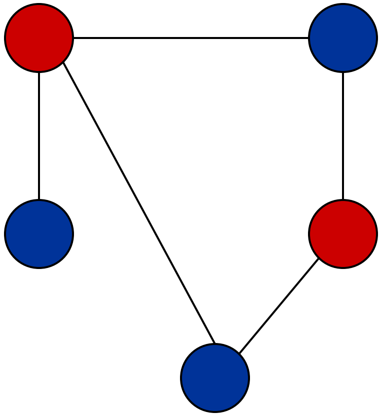
another drawing of G

An Obstruction to Bipartiteness

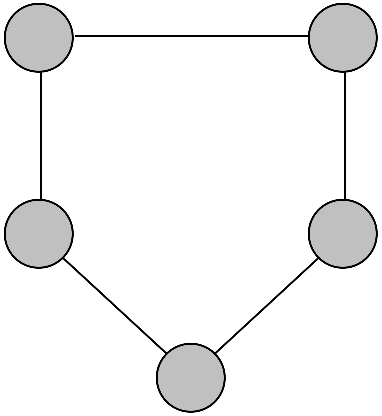
Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

Pf. Not possible to 2-color the odd cycle, let alone G.

# of edges in
this cycle is odd



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

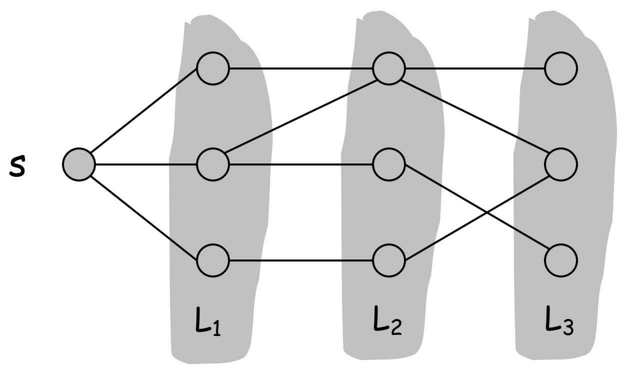
Bipartite Graphs

Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

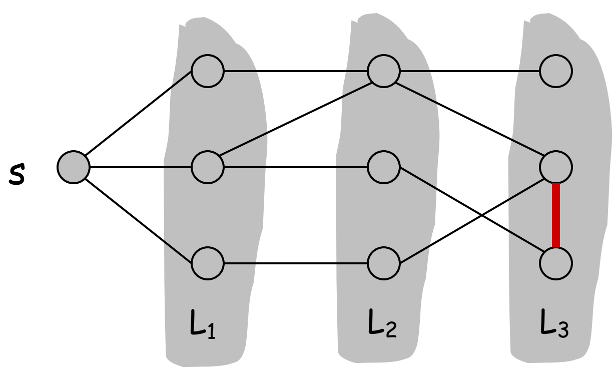
(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

(we can find an odd-length cycle whenever it is not bipartite)



Case (i)



Case (ii)

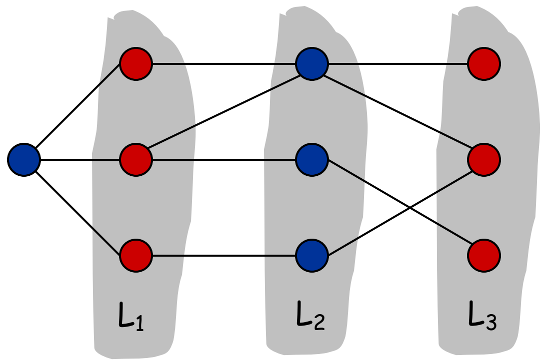
Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (i) Textbook page 96

- Suppose no edge joins two nodes in the same layer.
- By previous lemma, this implies all edges join nodes on adjacent layers.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.



Case (i)

Bipartite Graphs

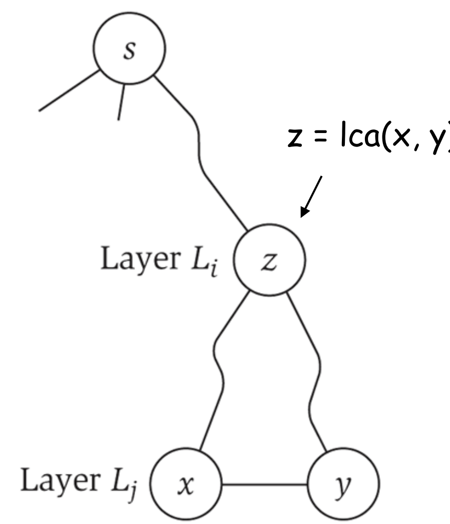
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

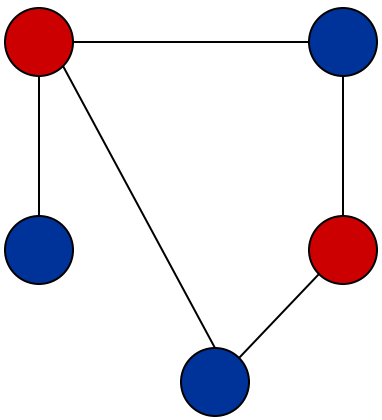
Pf. (ii)

- Suppose (x, y) is an edge with x, y in same level L_j.
- Let z = lca(x, y) = lowest common ancestor.
- Let L_i be level containing z.
- Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.
- Its length is $\underbrace{1}_{(x,\,y)} + \underbrace{(j-i)}_{\text{path from y to z}} + \underbrace{(j-i)}_{\text{path from z to x}}$, which is odd. ■

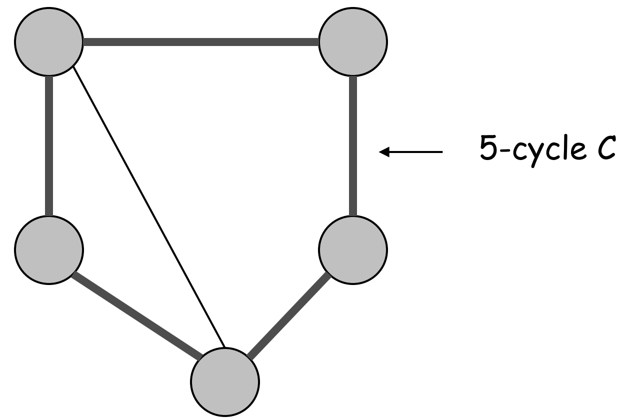


Obstruction to Bipartiteness

Corollary. A graph G is bipartite iff it contain no odd length cycle.



bipartite
(2-colorable)



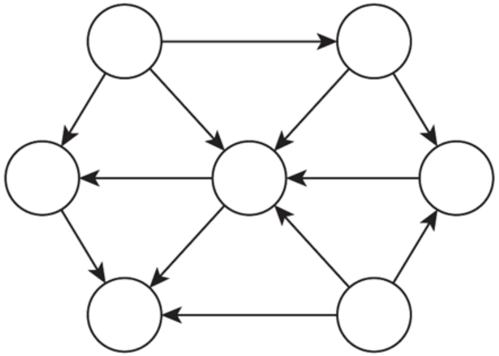
not bipartite
(not 2-colorable)

3.5 Connectivity in Directed Graphs

Directed Graphs

Directed graph. $G = (V, E)$

- Edge (u, v) goes from node u to node v.



Ex. Web graph - hyperlink points from one web page to another.

- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.

Graph Search

Directed reachability. Given a node s, find all nodes reachable from s.

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

Graph search. BFS extends naturally to directed graphs.

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

Strong Connectivity (强连通性)

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.

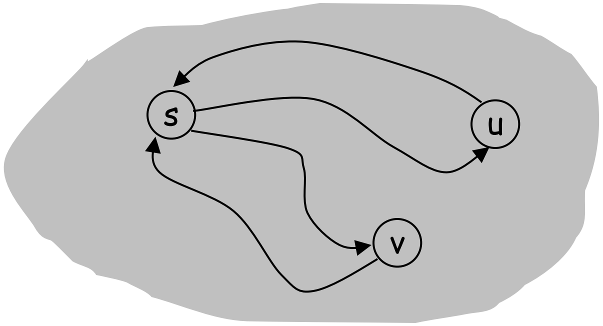
Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Pf. $\Rightarrow$ Follows from definition.

Pf. $\Leftarrow$ Path from u to v: concatenate u-s path with s-v path.

Path from v to u: concatenate v-s path with s-u path. ▪

ok if paths overlap



Strong Connectivity: Algorithm

Theorem. Can determine if G is strongly connected in $O(m + n)$ time.

Pf.

- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in G^{rev}.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. ■

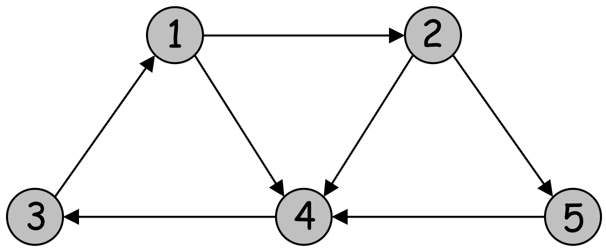
reverse orientation of every edge in G

Some books call it transpose G^T

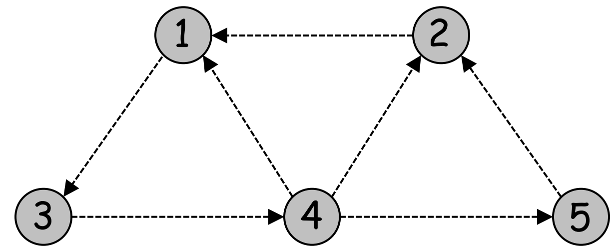
strongly connected

not strongly connected

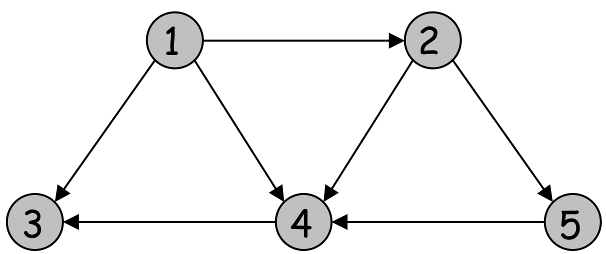
Strong Connectivity: Algorithm



G_1



G_1^{rev}

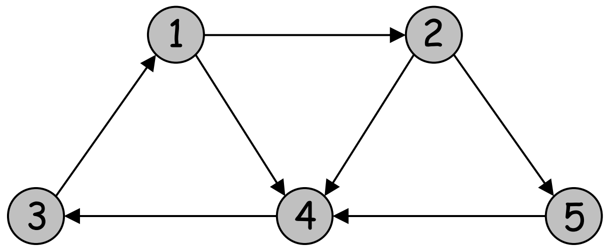


G_2

What's the result of running DFS on G_2?

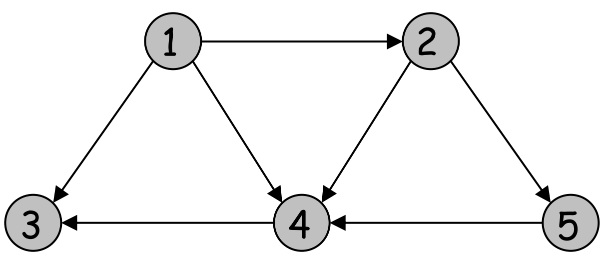
Strong Connected Components （强连通分量）

If G is strongly connected, how many trees will be returned from BFS/DFS?



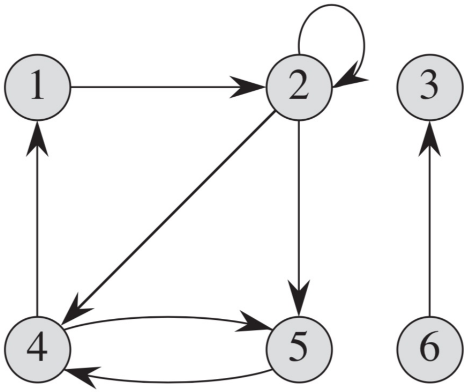
G_1

How many strongly connected components?



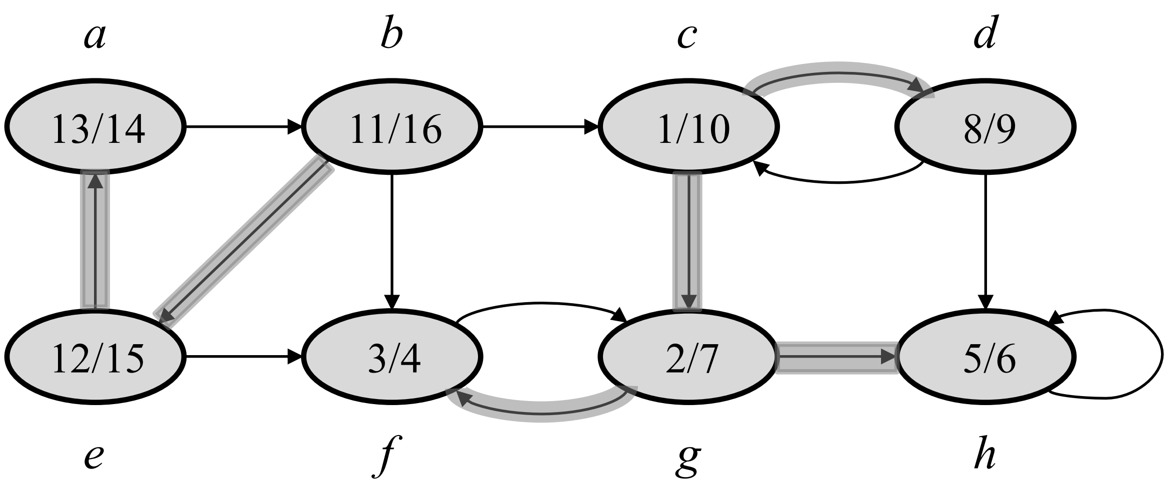
G_2

G_3

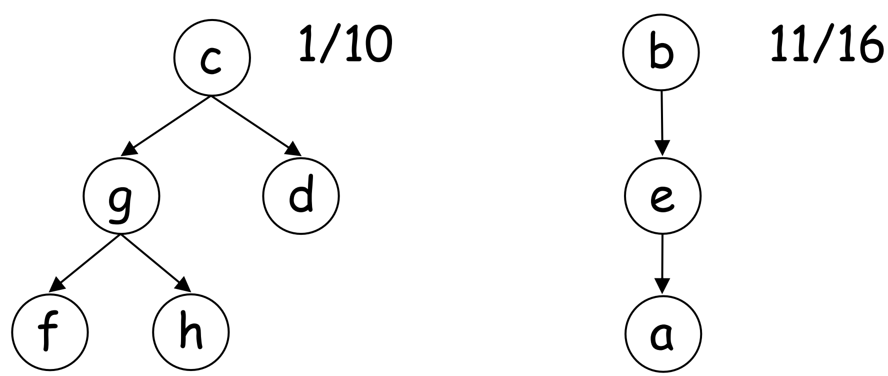


Algorithm for finding strongly connected components

Observe the visiting order nodes in DFS



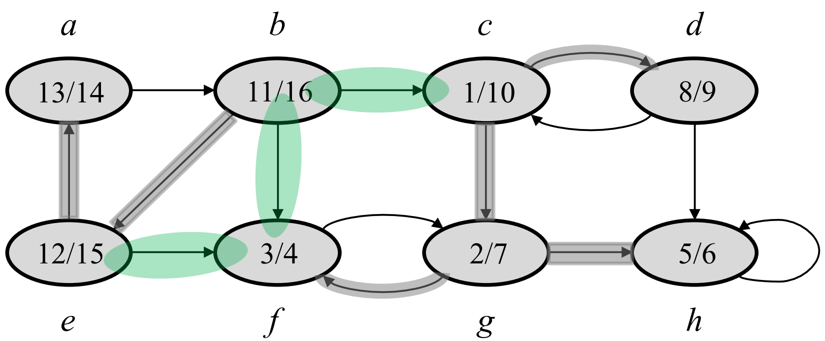
Two resulting trees:



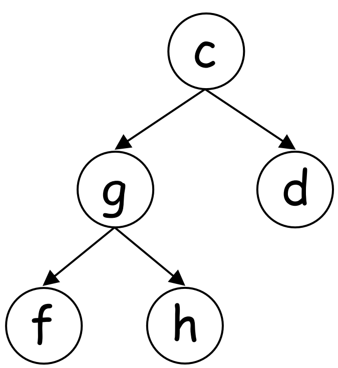
Algorithm for finding strongly connected components

Lemma 3.6 (Textbook page 85)

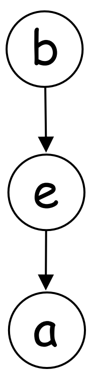
For a given recursive call DFS(u), all nodes that are visited between the invocation and end of this recursive call are descendants of u in T.



DFS(c) => T_A:



DFS(b) => T_B:

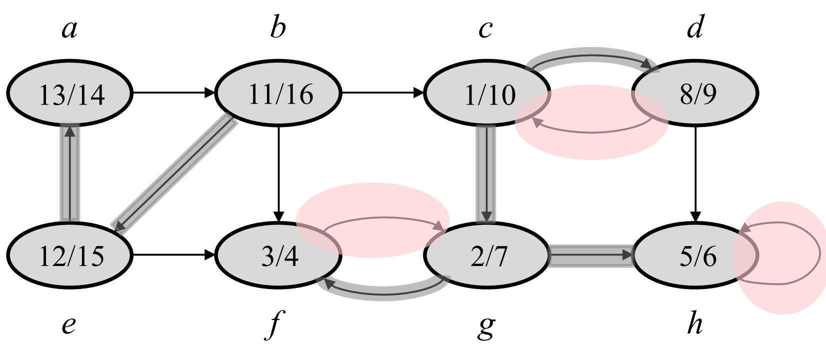


Q: if we run DFS on any node in T_A, can we reach T_B?

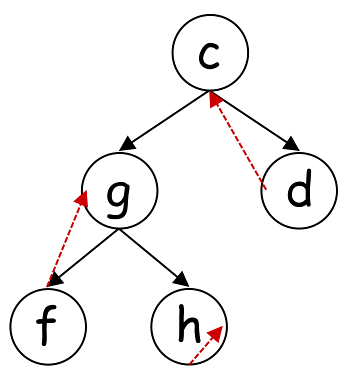
Algorithm for finding strongly connected components

Lemma 3.7 (Textbook page 85)

Let T be a depth-first search tree, let x and y be nodes in T, and let (x, y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other.



DFS(c) => T_A:



Algorithm for finding strongly connected components

STRONGLY-CONNECTED-COMPONENT(G)

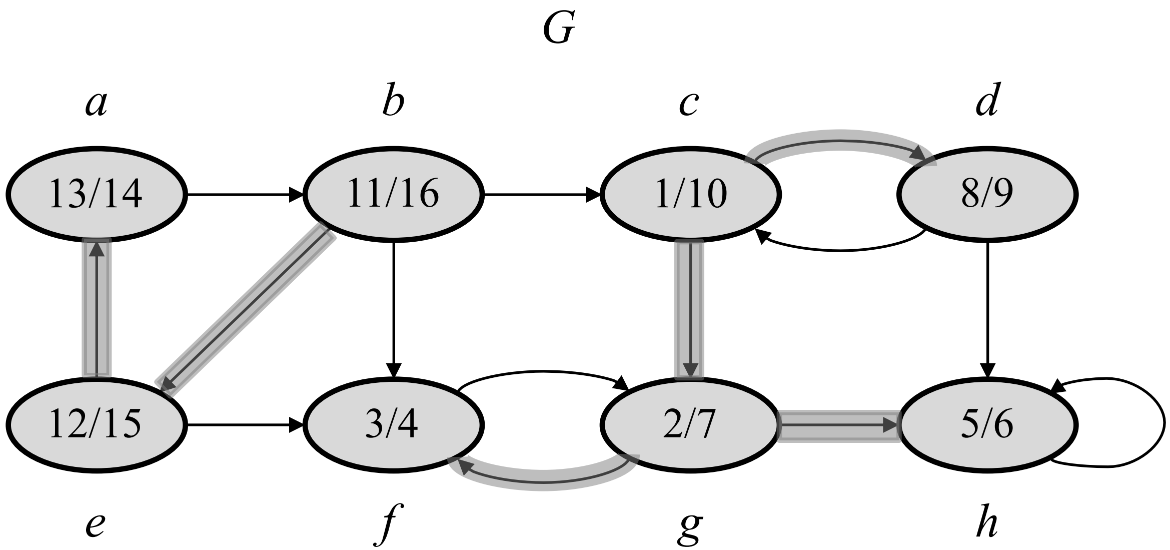
1. call DFS(G) to compute the finishing times $u.f$ for each vertex u
2. compute G^T ($G^T = G^{rev}$ => every edge in reversed direction)
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate SCC

MIT Book, page 604

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

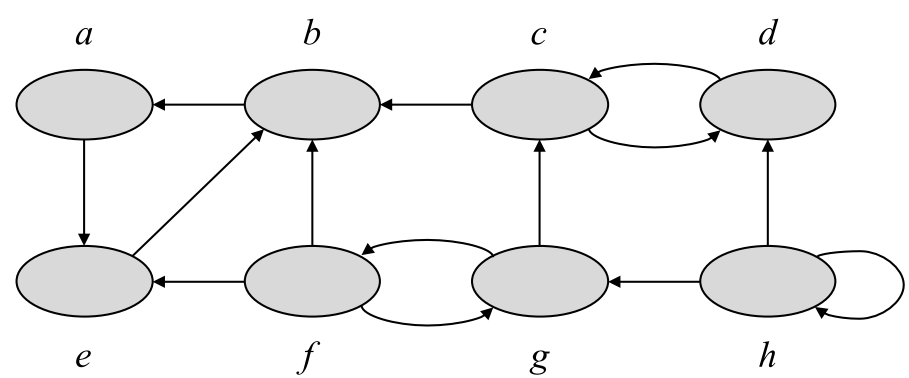
```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

Strongly connected components - demo



Call DFS(G)

Get reversed graph G^{rev}:

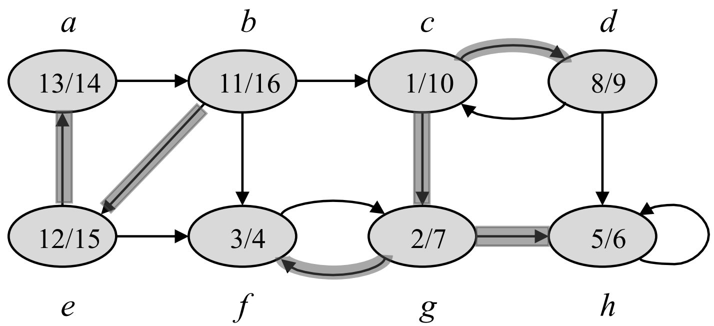


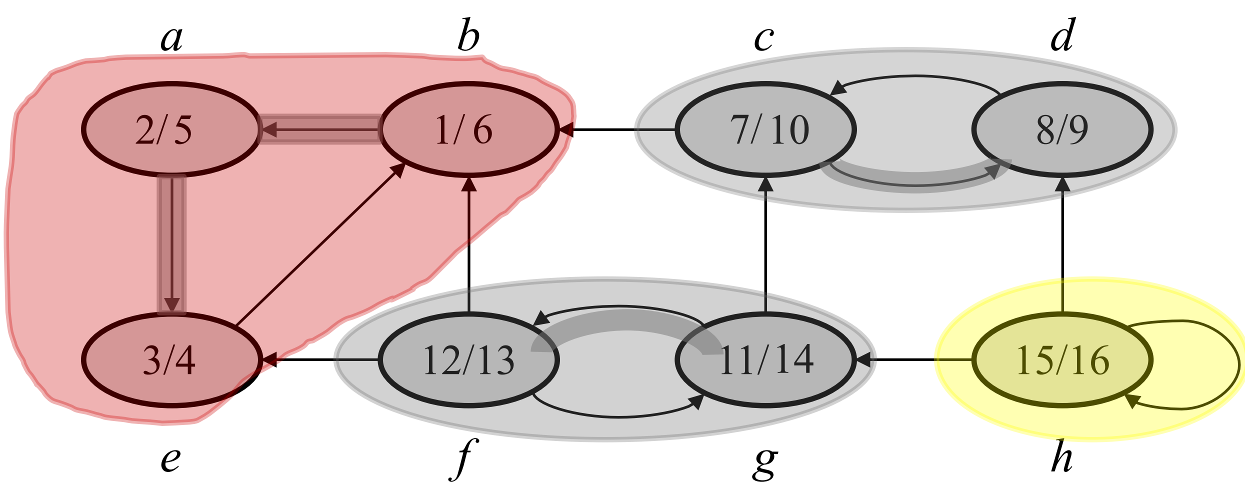
Call DFS(G^{rev})

consider the vertices in order of decreasing $u.f$ (finishing times from previous DFS)

Strongly connected components - demo

Old finishing times from previous DFS(G)

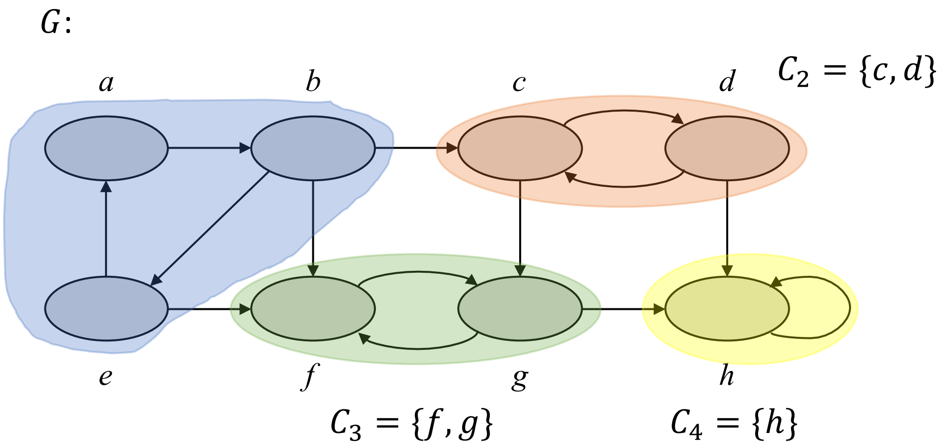




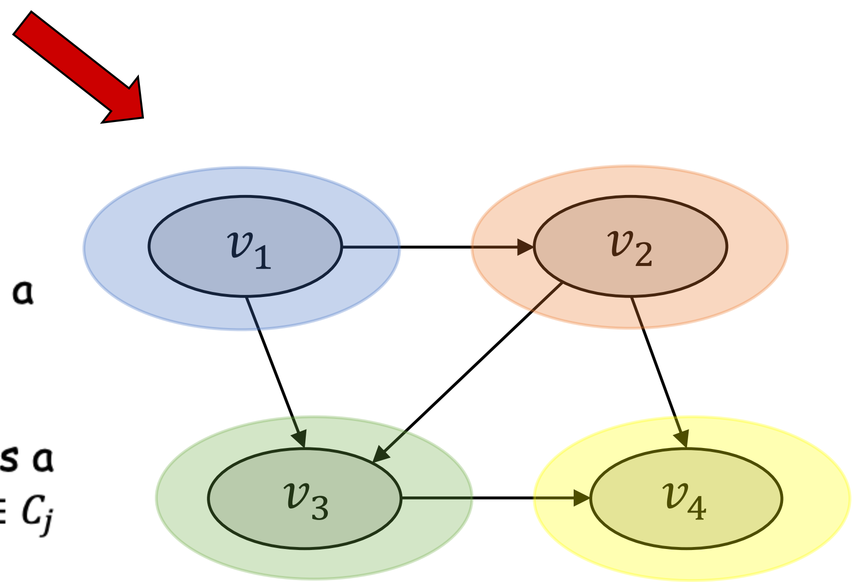
DFS(G^{rev}) produces 4 DFS trees

That is, 4 strongly connected components (of G and G^{rev})

Strongly connected components - Result



Q: Is there any cycles in G^{SCC}?



Suppose that G has SCCs, $C_1, C_2, \ldots, C_k$

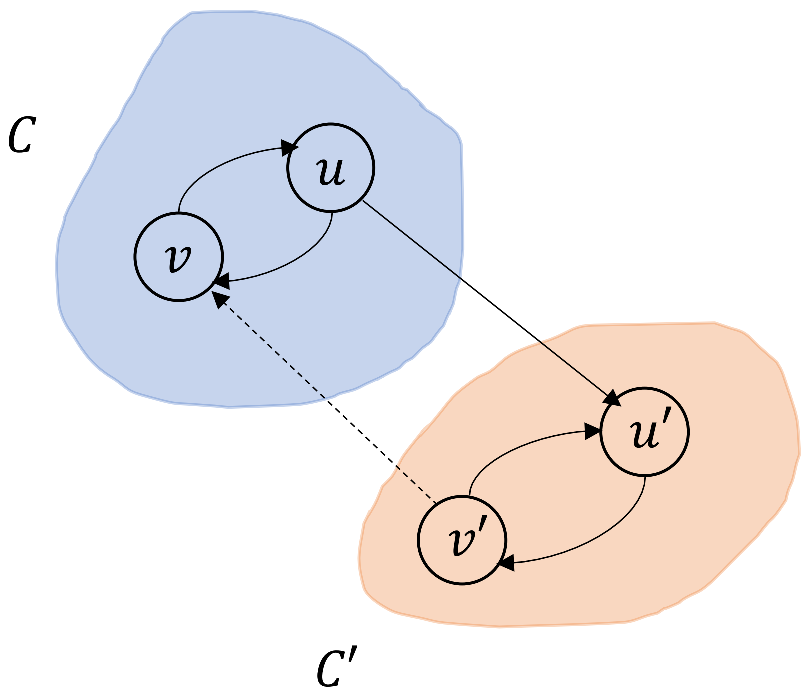
Define $V^{SCC} = \{v_1, v_2, \ldots, v_k\}$, which contains a vertex v_i for each SCC C_i of G.

There is an edge $(v_i, v_j) \in E^{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and $y \in C_j$

Def. Component graph $G^{SCC} = (V^{SCC}, E^{SCC})$

Correctness of strongly connected components algorithm

Lemma 1: Let C and C' be distinct strongly connected components in directed graph $G=(V,E)$, let $u,v \in C$, let $u',v' \in C'$, and suppose that G contains a path $u \rightsquigarrow u'$. Then G cannot also contain a path $v' \rightsquigarrow v$.
(Lemma 22.13 in MIT book, page 617)



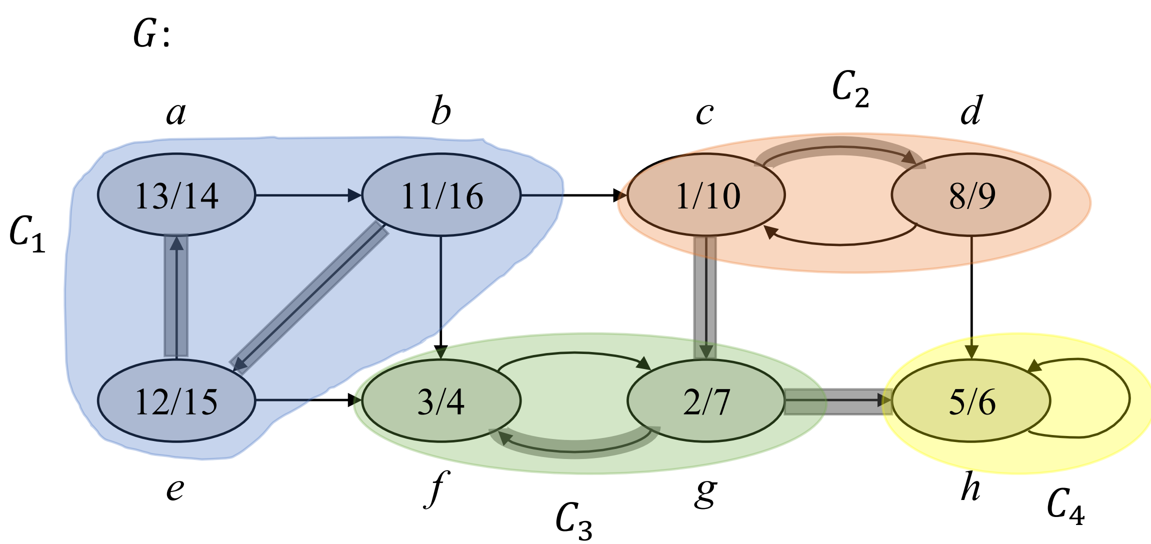
If G contains a path $v' \rightsquigarrow v$
Then $u' \rightsquigarrow v' \rightsquigarrow v \rightsquigarrow u$

Thus, u and u' are mutually reachable

contradicting the assumption that C and C' are distinct SCCs

Correctness of strongly connected components algorithm

Lemma 2: Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$. (Lemma 22.14 in MIT book, page 618)



$f(C_1) = 16$

$f(C_2) = 10$

$f(C_3) = 7$

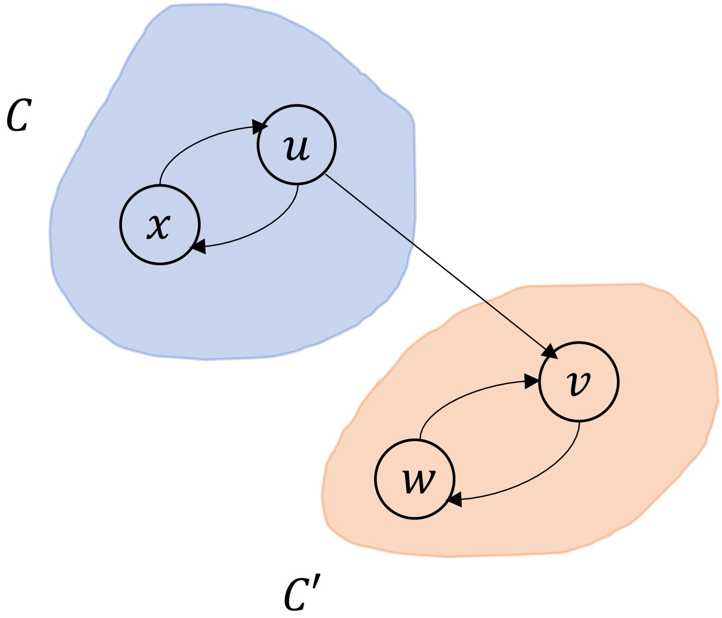
$f(C_4) = 6$

Correctness of strongly connected components algorithm

Proof of Lemma 2: Assuming two SCCs, C and C', and consider two cases

Case 1: C is first discovered during the first DFS

Case 2: C' is first discovered …



Case 1: C is first discovered => $d(C) < d(C')$

Assuming x is the first discovered node in C

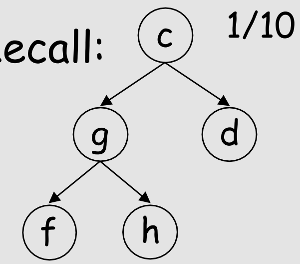
=>

All nodes in C and C' are descendants of x

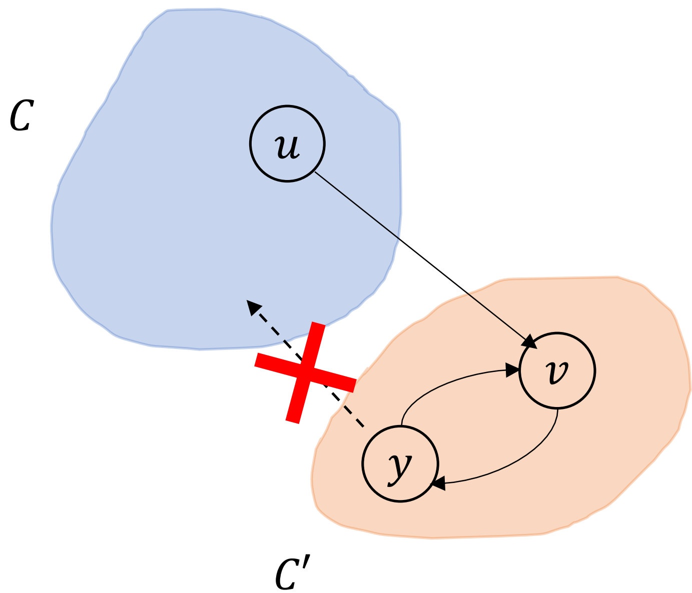
=>

$$x.f = f(C) > f(C')$$

Recall:



Correctness of strongly connected components algorithm



Case 2: C' is first discovered => $d(C) > d(C')$,
let y be the first vertex discovered in C'.
=> $y.f = f(C')$

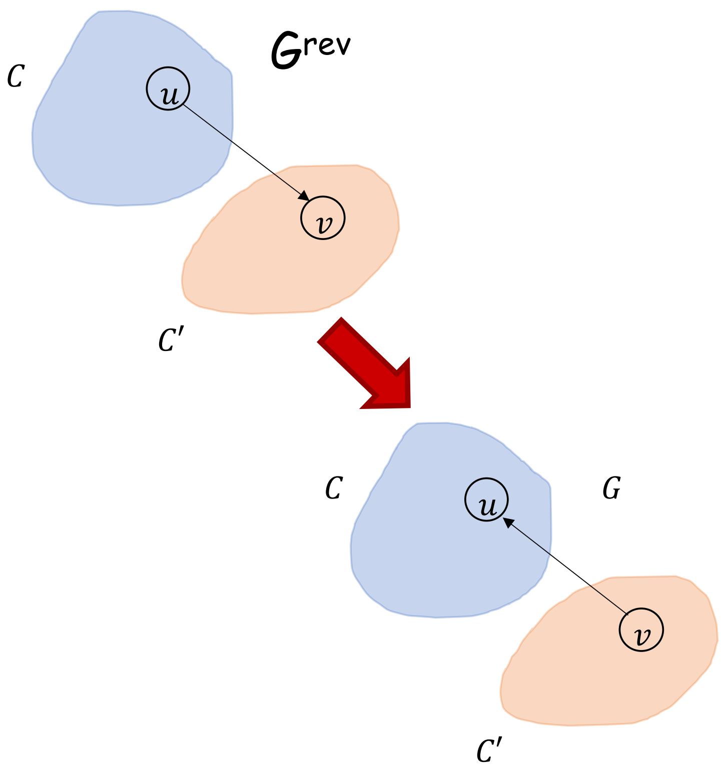
By Lemma 1, there cannot be a path $C' \rightsquigarrow C$
=> no nodes in C is reachable from y

Thus, for any node $w \in C$, $w.f > y.f$
That is, $f(C) > f(C')$. Proof done.

Correctness of strongly connected components algorithm

（推论）

Corollary of lemma 2: Let C and C' be distinct strongly connected components in directed graph G=(V,E). Suppose that there is an edge (u,v) ∈ G^{rev}, where u∈C and v∈C'. Then $f(C) < f(C')$.

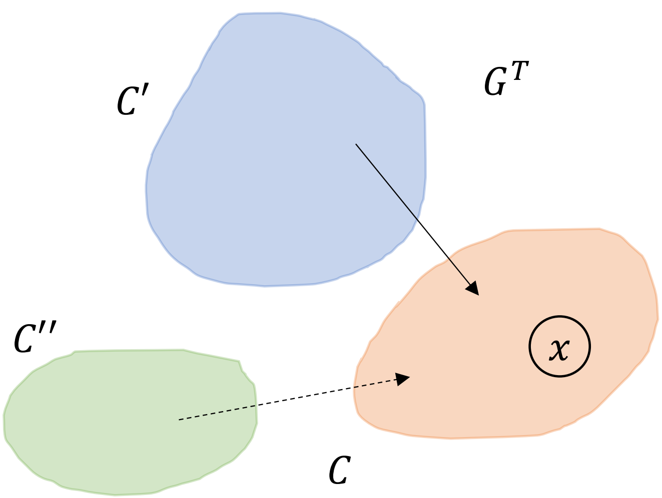


According to lemma 2, $f(C) < f(C')$

Correctness of strongly connected components algorithm

Now, consider the second round of DFS in the algorithm

STRONGLY-CONNECTED-COMPONENT(G)
1. call DFS(G) to compute the finishing times $u.f$ for each vertex u
2. compute G^T
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate SCC



We start from C whose finishing time $f(C)$ is max

Because $f(C)$ is maximum, there can only be edges from other SCCs to C

Thus, the DFS tree rooted at x contains exactly the nodes of C.
(no nodes in C′ or C″ will be visited)

3.6 DAGs and Topological Ordering

（DAG：有向无环图） （拓扑排序）

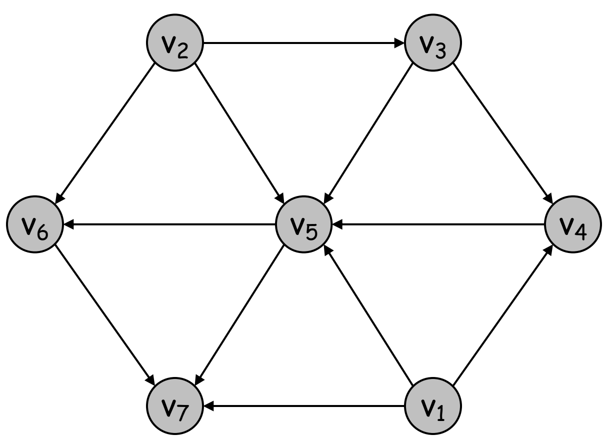
Directed Acyclic Graphs （DAG：有向无环图）

Def. An DAG is a directed graph that contains no directed cycles.

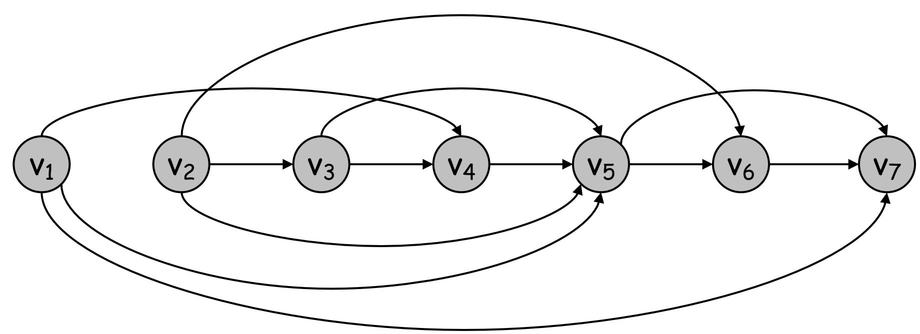
（优先序）

Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.

Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.

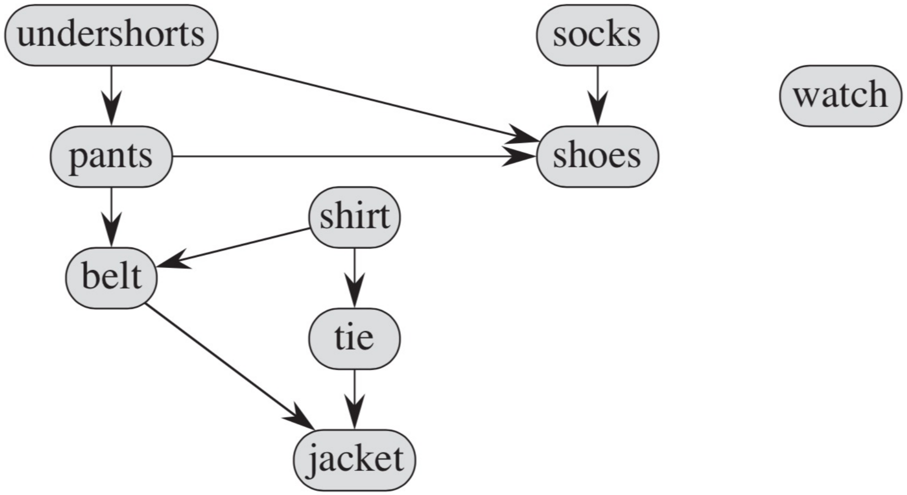


a DAG



a topological ordering

DAG example



MIT Book, page 612

DAG describes precedence relations or dependencies

Precedence Constraints

Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.

Applications.

- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j. Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

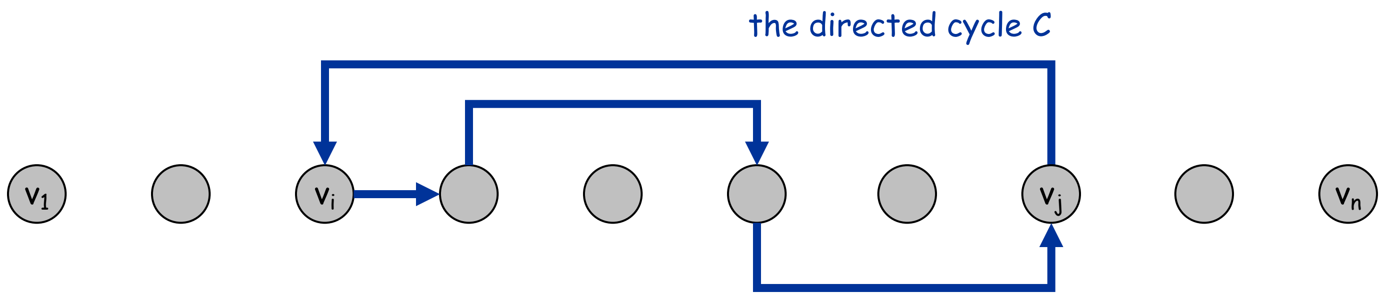
Directed Acyclic Graphs

(Lemma 3.18 textbook page 101)

Lemma. If G has a topological order, then G is a DAG.

Pf. (by contradiction)

- Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C. Let's see what happens.
- Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
- By our choice of i, we have i < j.
- On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have j < i, a contradiction. ■



the supposed topological order: $v_1, \ldots, v_n$

Lemma. If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?

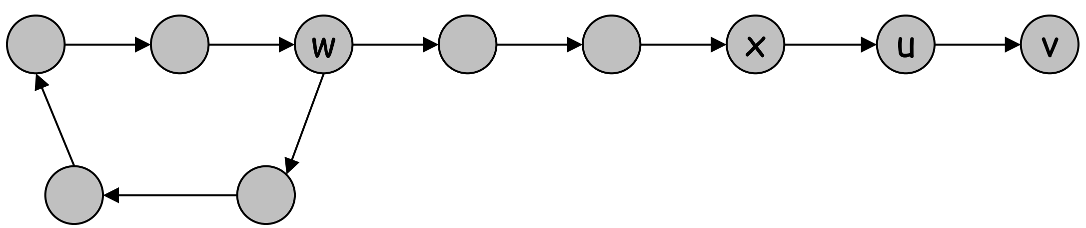
Directed Acyclic Graphs

Textbook, lemma 3.19, page 102

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
- Then, since u has at least one incoming edge (x, u), we can walk backward to x.
- Repeat (after n+1 steps) until we visit a node, say w, twice.
- Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. ▪



Directed Acyclic Graphs

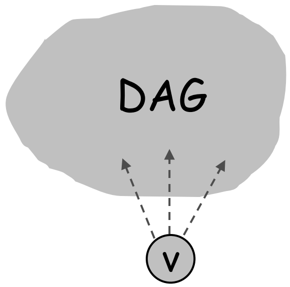
Textbook, lemma 3.20, page 102

Lemma. If G is a DAG, then G has a topological ordering.

Pf. (by induction on n) 数学归纳法

- Base case: true if n = 1.
- Given DAG on n > 1 nodes, find a node v with no incoming edges.
- G - { v } is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, G - { v } has a topological ordering.
- Place v first in topological ordering; then append nodes of G - { v }
- in topological order. This is valid since v has no incoming edges. ▪

To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of $G-\{v\}$
  and append this order after v



Topological Sorting Algorithm: Running Time

Theorem. Algorithm finds a topological order in $O(m + n)$ time.

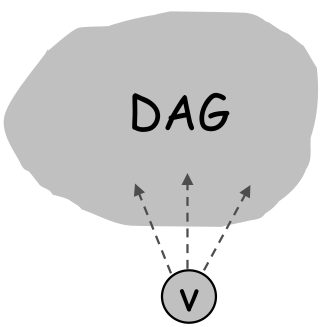
Pf.

- Maintain the following information:
  - `count[w]` = remaining number of incoming edges
  - S = set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete v
  - remove v from S
  - decrement `count[w]` for all edges from v to w, and add w to S if c `count[w]` hits 0
  - this is $O(1)$ per edge ▪

Topological Sorting Algorithm: Another View

Alternative:
Order nodes in reverse order that DFS finishes visiting them

Q: Which node has the biggest finish time?

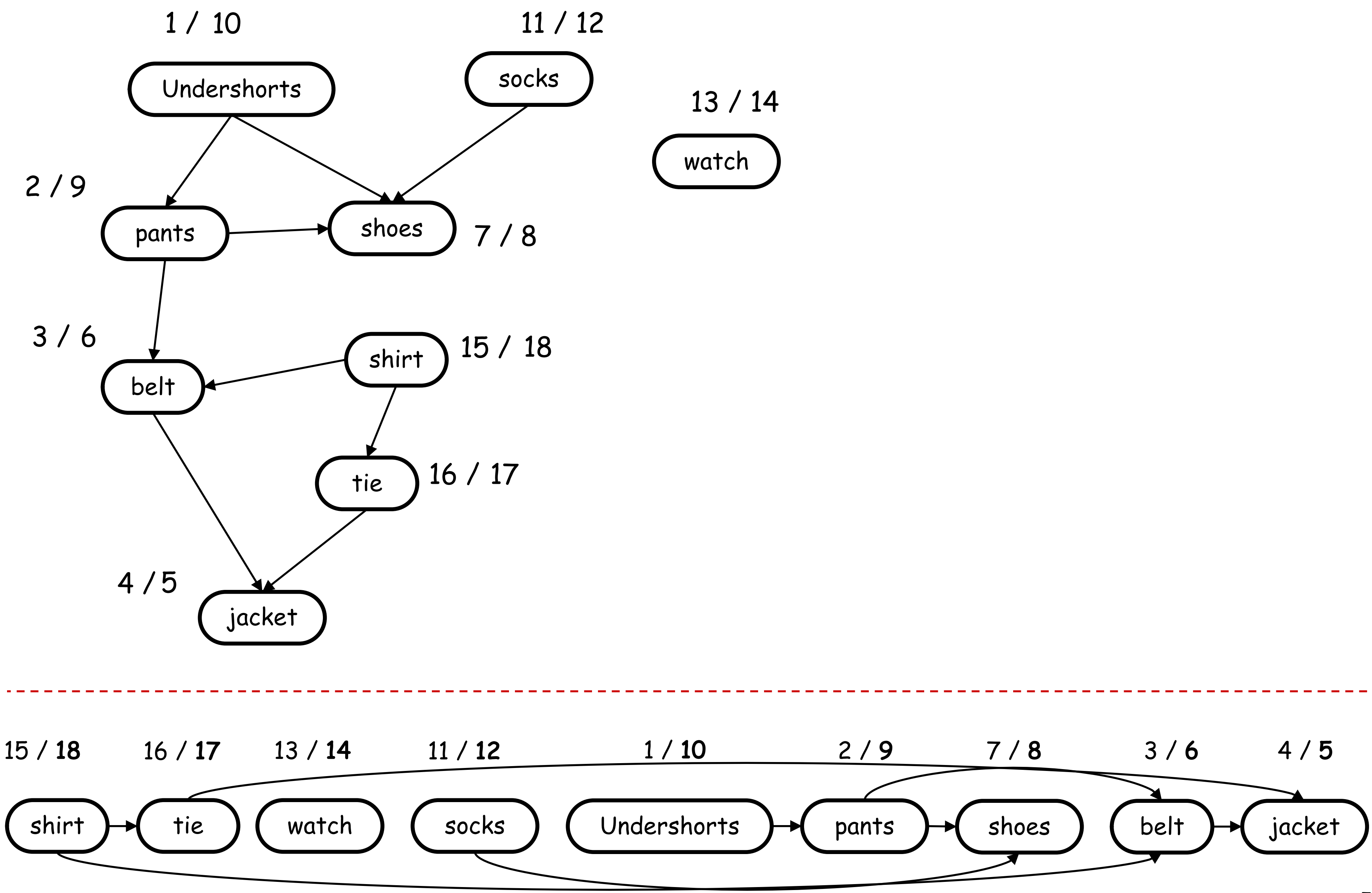


Consider using this DFS implementation

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

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

Topological Sorting Algorithm: Using finishing time in DFS



Topological Sorting Algorithm: Using finishing time in DFS

MIT Book, page 612

TOPOLOGICAL-SORT(G)

1. call DFS(G) to compute the finishing times $v.f$ for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. **return** the linked list of vertices

Theorem: TOPOLOGICAL-SORT produces a topological sort of the input DAG.

Proof:
It suffices to show that for any pair of nodes u, v, if G contains an edge (u, v), then $v.f < u.f$.

Case 1: when going from u to v, v is finished (black)
Case 2: when going from u to v, v is not finished (white)