

# DIRECTED GRAPHS

---

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *depth-first search*
- ▶ *breadth-first search*
- ▶ *topological sort*
- ▶ *strong components*

see videos



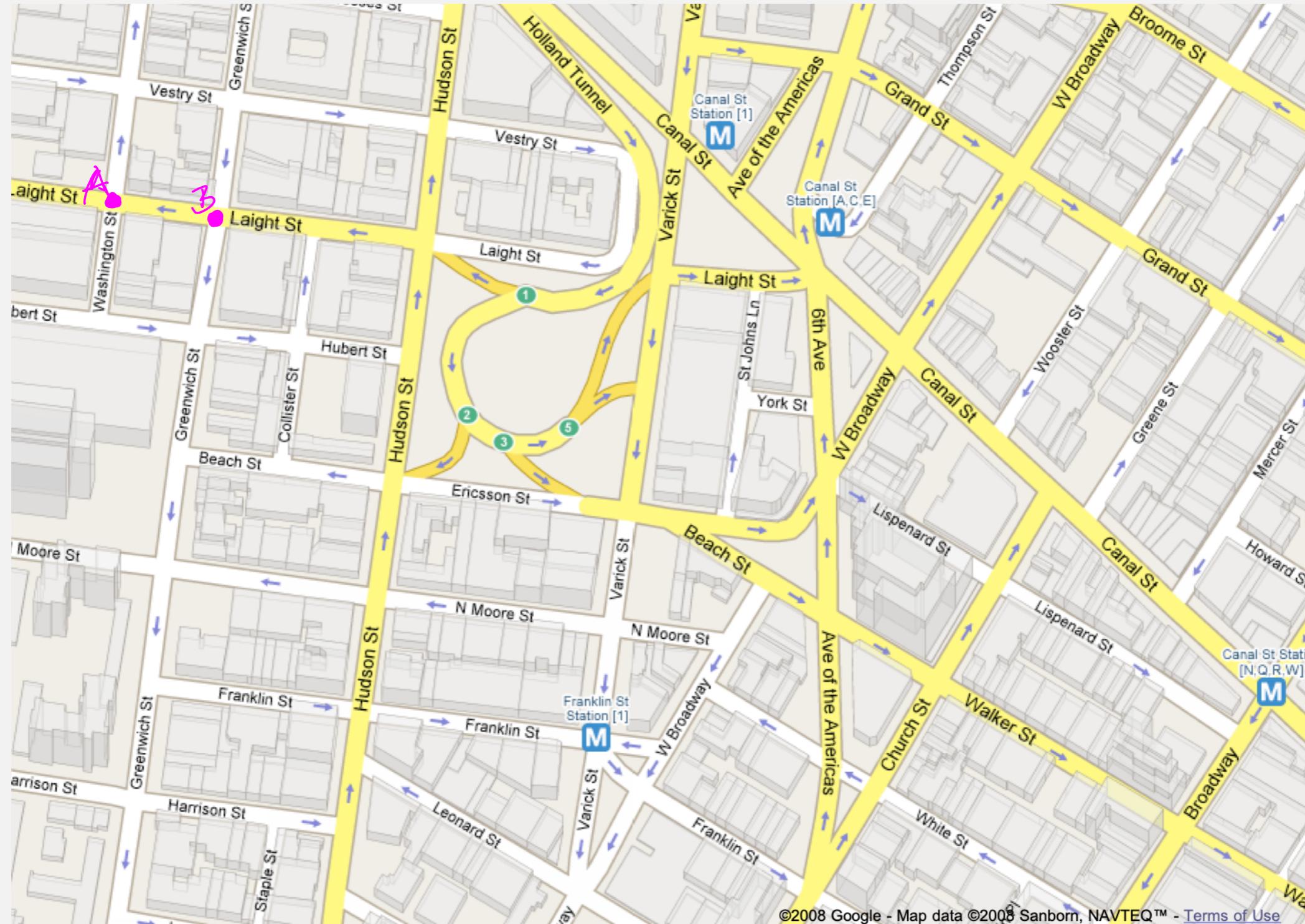
# DIRECTED GRAPHS

---

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *depth-first search*
- ▶ *breadth-first search*
- ▶ *topological sort*

# Road networks

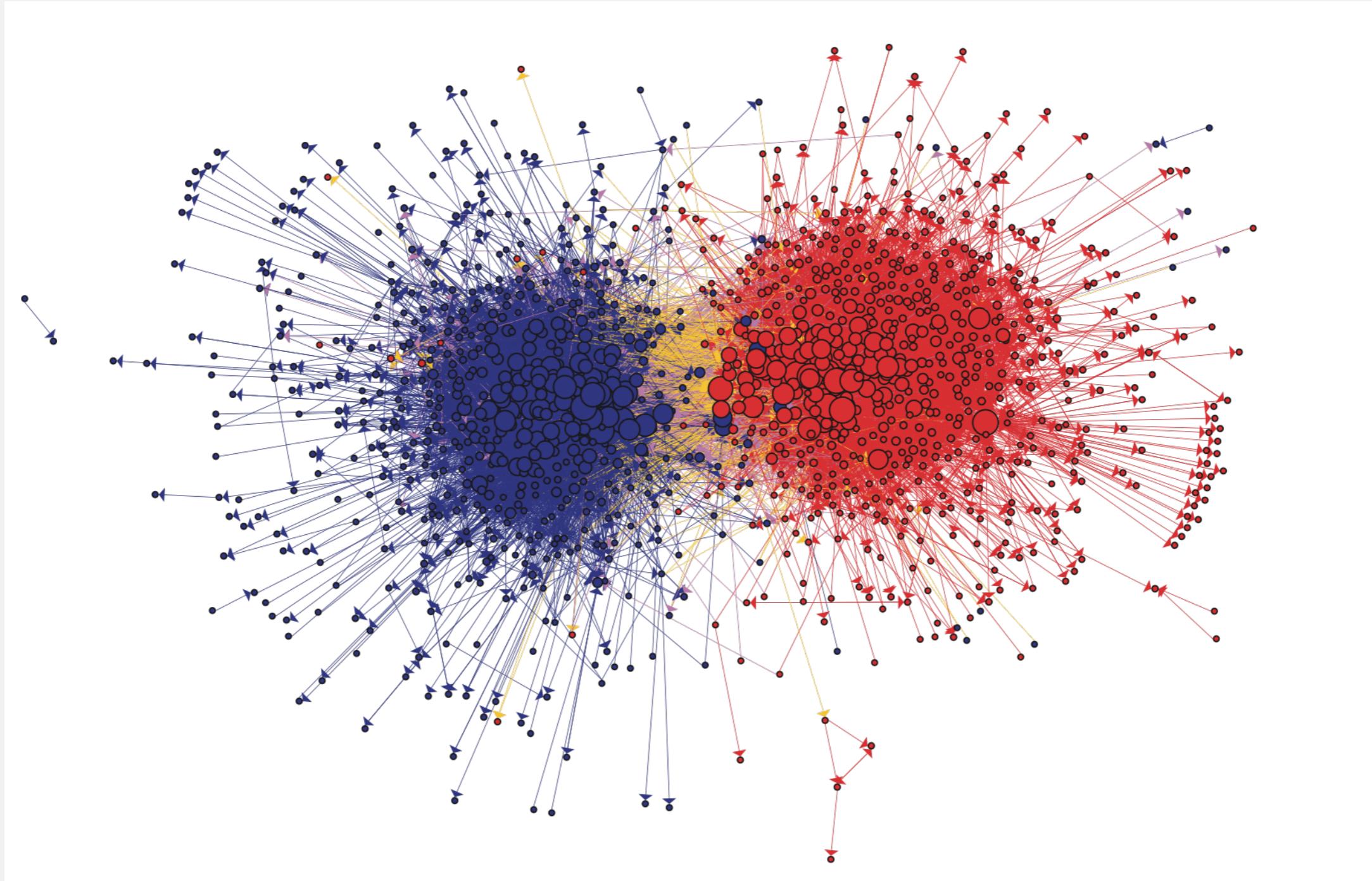
Vertex = intersection; edge = one-way street.



# Political blogosphere links

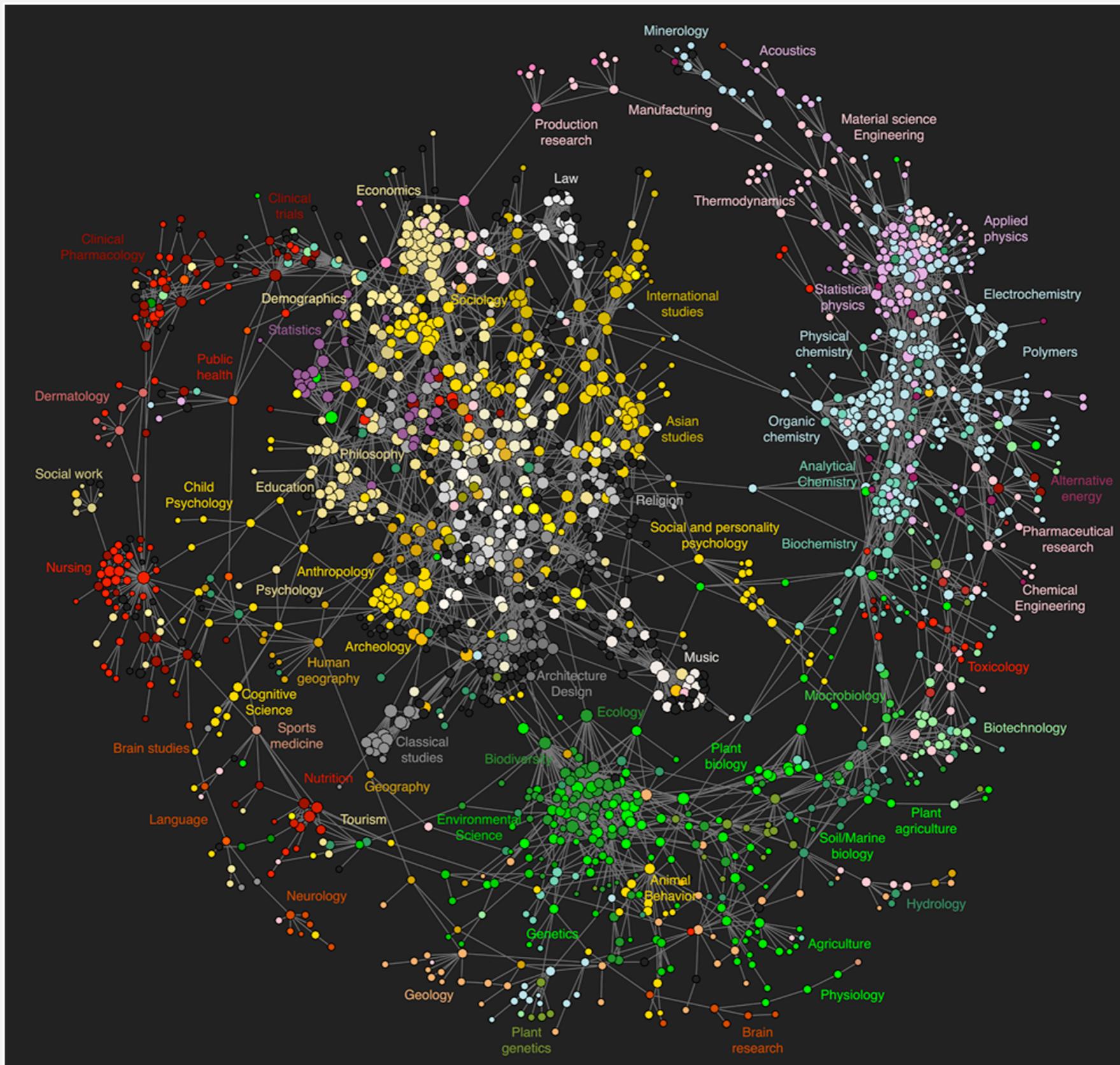
---

Vertex = political blog; edge = link.



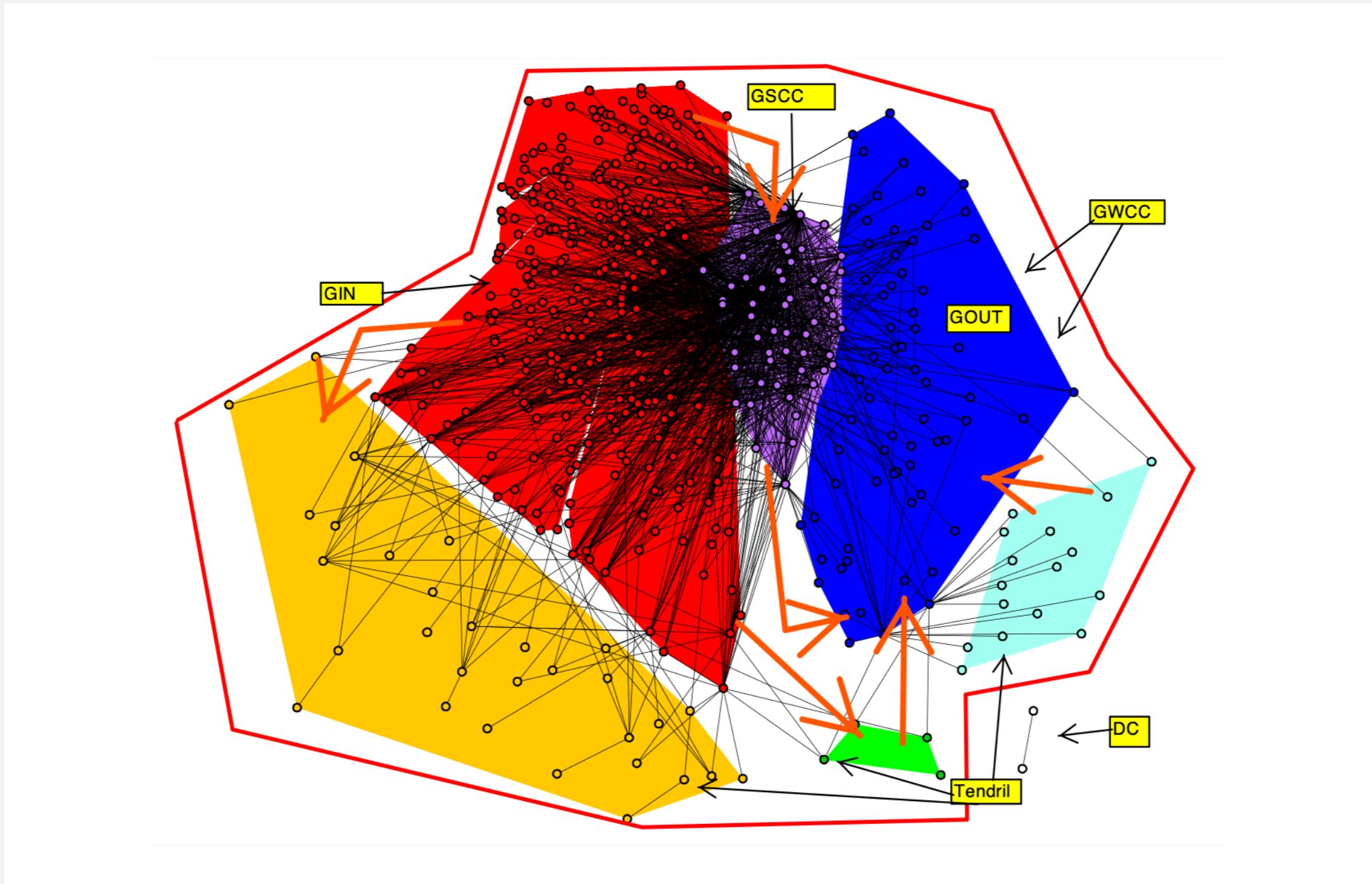
**The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, Adamic and Glance, 2005**

# Science clickstreams



# Overnight interbank loans

Vertex = bank; edge = overnight loan.

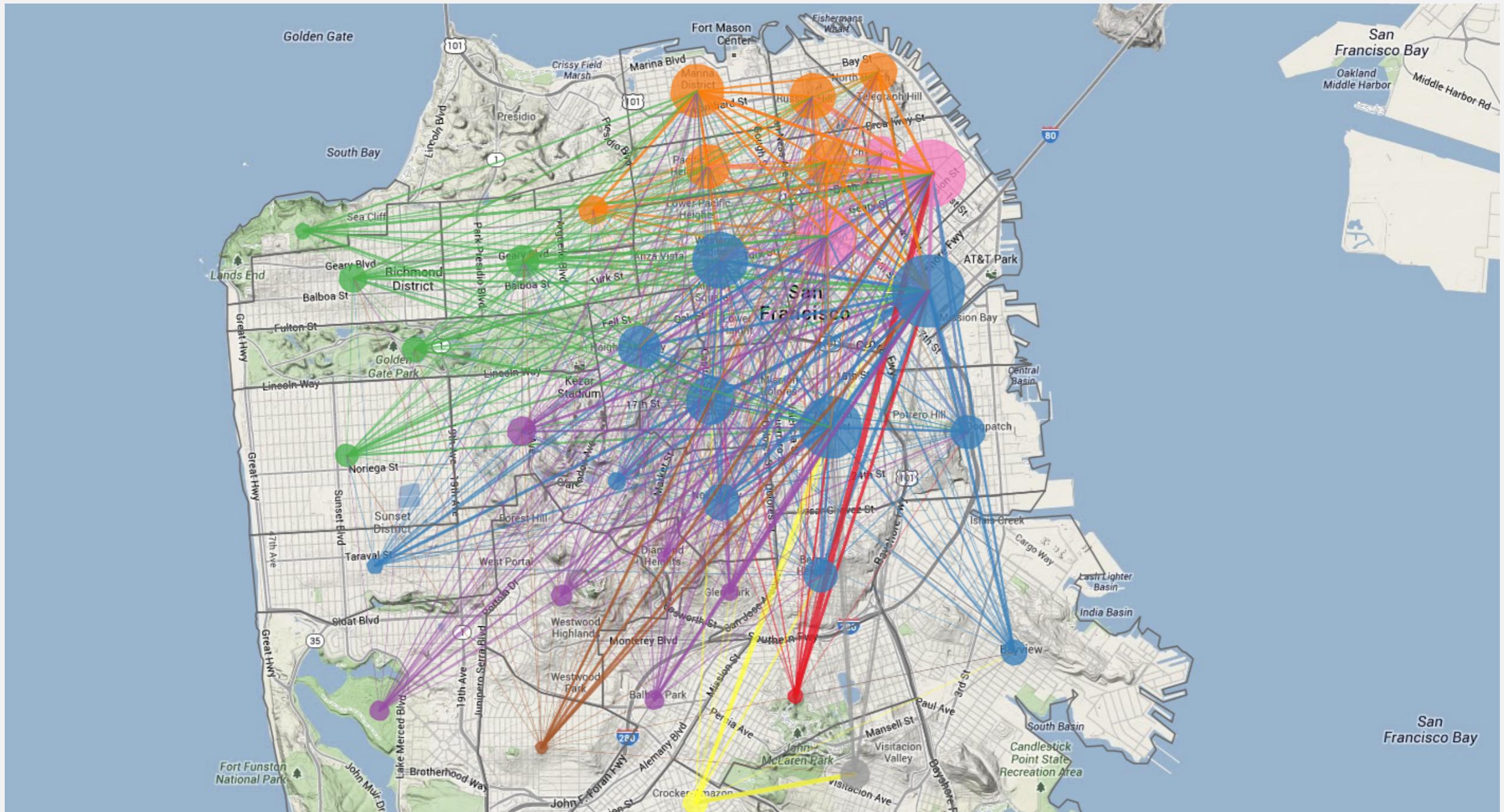


The Topology of the Federal Funds Market, Bech and Atalay, 2008

# Uber rides

LO 9B.4

Vertex = taxi pickup; edge = taxi ride.



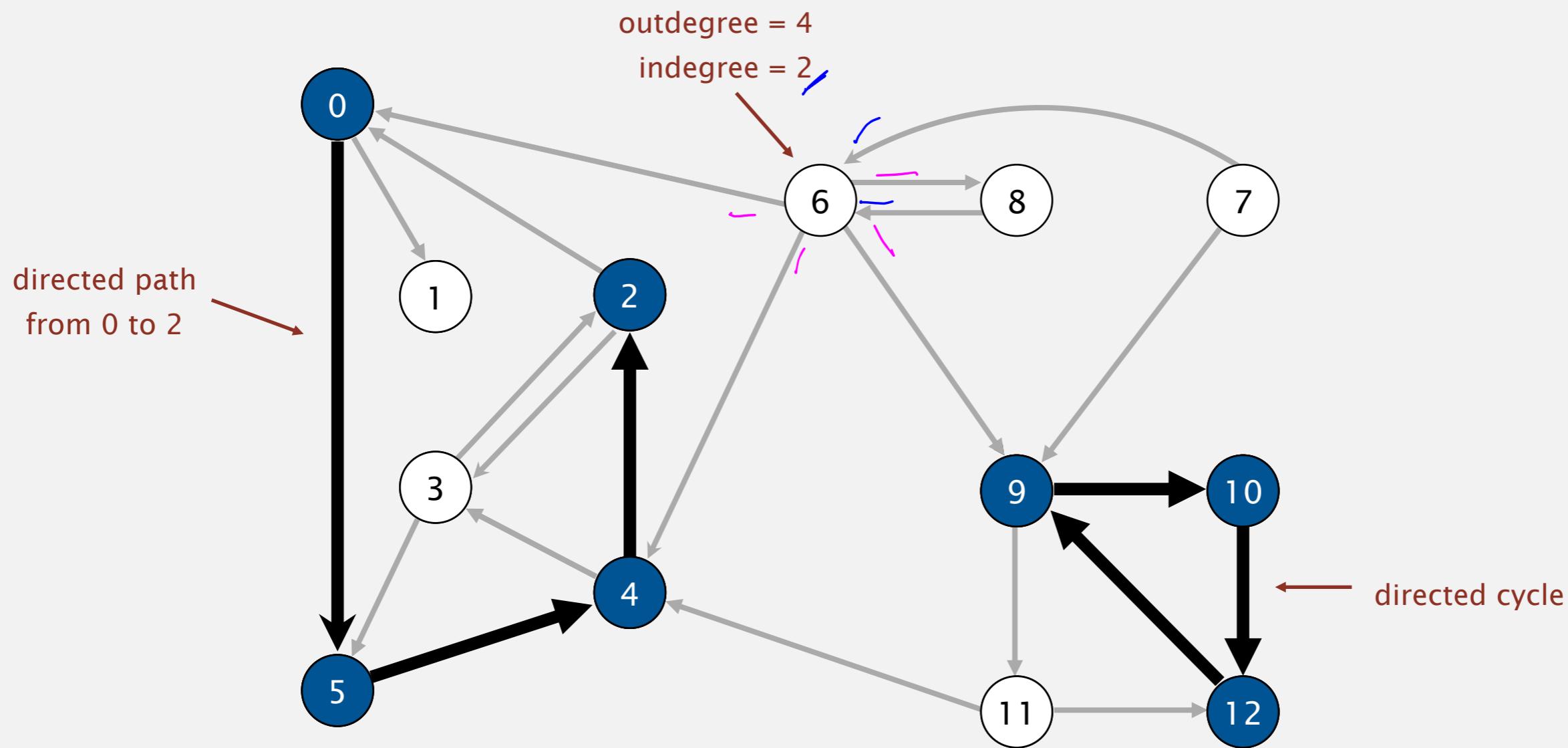
<http://blog.uber.com/2012/01/09/uberdata-san-francisocomics>

# Digraph applications

Directed graph

digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

**Digraph.** Set of vertices connected pairwise by directed edges.



# Some digraph problems

problem	description
<u>s-t path</u>	<i>Is there a path from s to t ?</i>
<u>shortest s-t path</u>	<i>What is the shortest path from s to t ?</i>
<b>directed cycle</b>	<i>Is there a <u>directed cycle</u> in the graph ?</i>
<b>topological sort</b>	<i>Can the digraph be drawn so that all edges point upwards?</i>
<b>strong connectivity</b>	<i>Is there a directed path between every pairs of vertices ?</i>
<b>transitive closure</b>	<i>For which vertices v and w is there a directed path from v to w ?</i>
<u>PageRank</u>	<i>What is the importance of a web page ?</i>

# DIRECTED GRAPHS

---

- ▶ *introduction*
- ▶ ***digraph API***
- ▶ *depth-first search*
- ▶ *breadth-first search*
- ▶ *topological sort*

# Digraph API

Almost identical to Graph API.

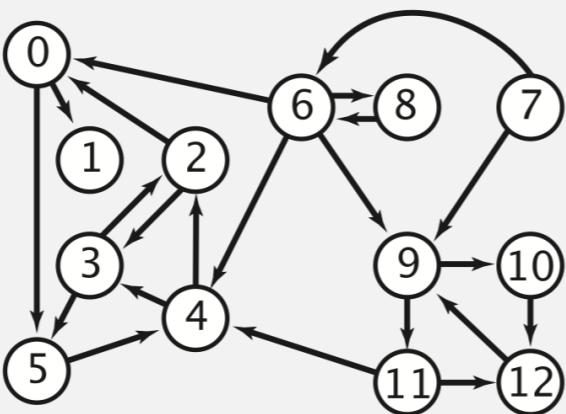
```
public class Digraph
```

	<u>Digraph(int V)</u>	<i>create an empty digraph with V vertices</i>
	<u>Digraph(<u>In</u> in)</u>	<i>create a digraph from input stream</i>
void	<u>addEdge(int v, int w)</u>	<i>add a directed edge <math>v \rightarrow w</math></i>
Iterable<Integer>	<u>adj(int v)</u>	<i>vertices adjacent from v</i>
int	<u>V()</u>	<i>number of vertices</i>
int	<u>E()</u>	<i>number of edges</i>
Digraph	<u>reverse()</u>	<i>reverse of this digraph</i>
String	<u>toString()</u>	<i>string representation</i>

# Digraph API

**tinyDG.txt**  
V → 13  
E ← 22

4 2  
2 3  
3 2  
6 0  
0 1  
2 0  
11 12  
12 9  
9 10  
9 11  
7 9  
10 12  
11 4  
4 3  
3 5  
6 8  
8 6  
⋮



% **java Digraph tinyDG.txt**

0->5  
0->1  
2->0  
2->3  
3->5  
3->2  
4->3  
4->2  
5->4  
⋮  
11->4  
11->12  
12->9

```
In in = new In(args[0]);
Digraph G = new Digraph(in);
```

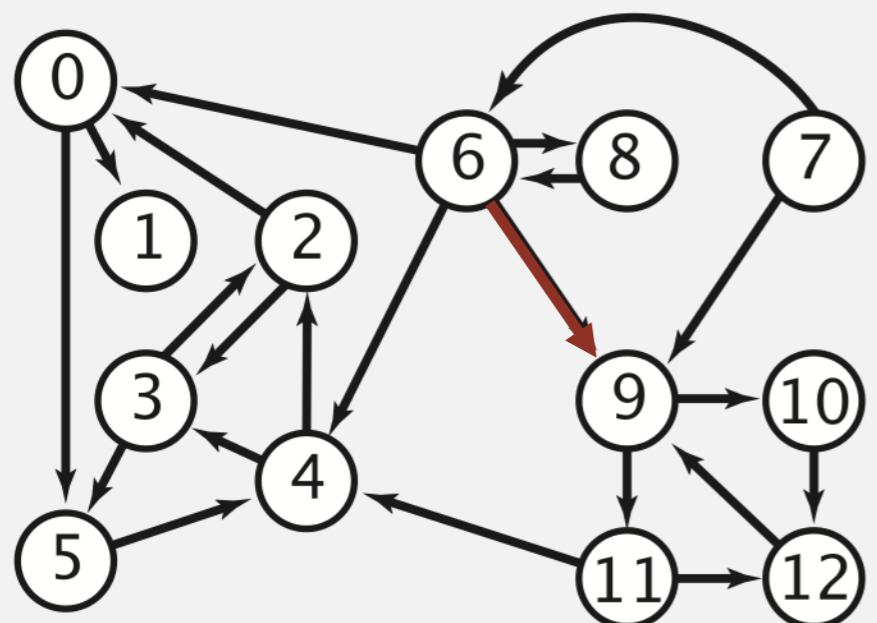
← read digraph from  
input stream

```
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

← print each edge (once)

# Digraph representation: set of edges

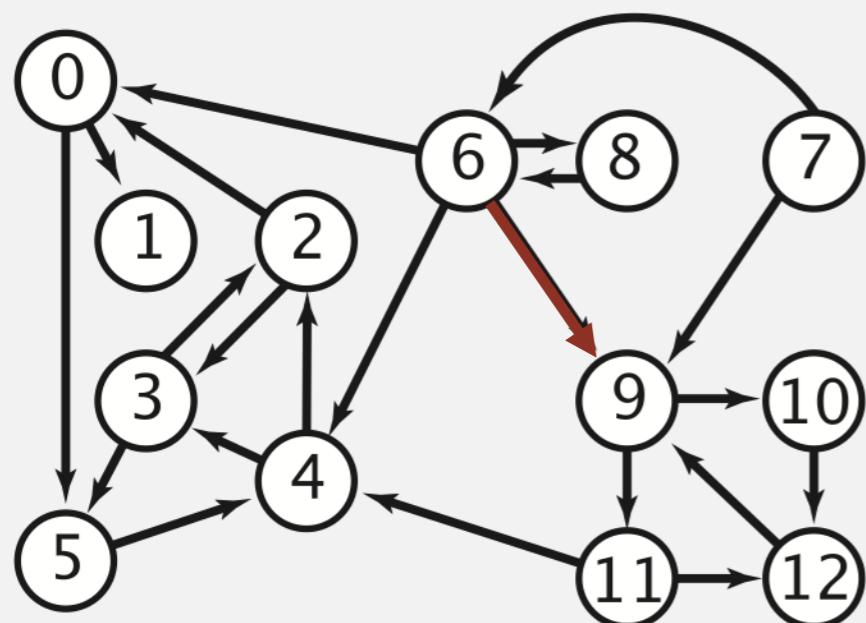
Store a list of the edges (linked list or array).



0	1
0	5
2	0
2	3
3	2
3	5
4	2
4	3
5	4
6	0
6	4
6	8
6	9
7	6
7	9
8	6
9	10
9	11
10	12
11	4
11	12
12	9

## Digraph representation: adjacency matrix

Maintain a two-dimensional  $v$ -by- $V$  boolean array;  
for each edge  $v \rightarrow w$  in the digraph:  $\text{adj}[v][w] = \text{true}$ .



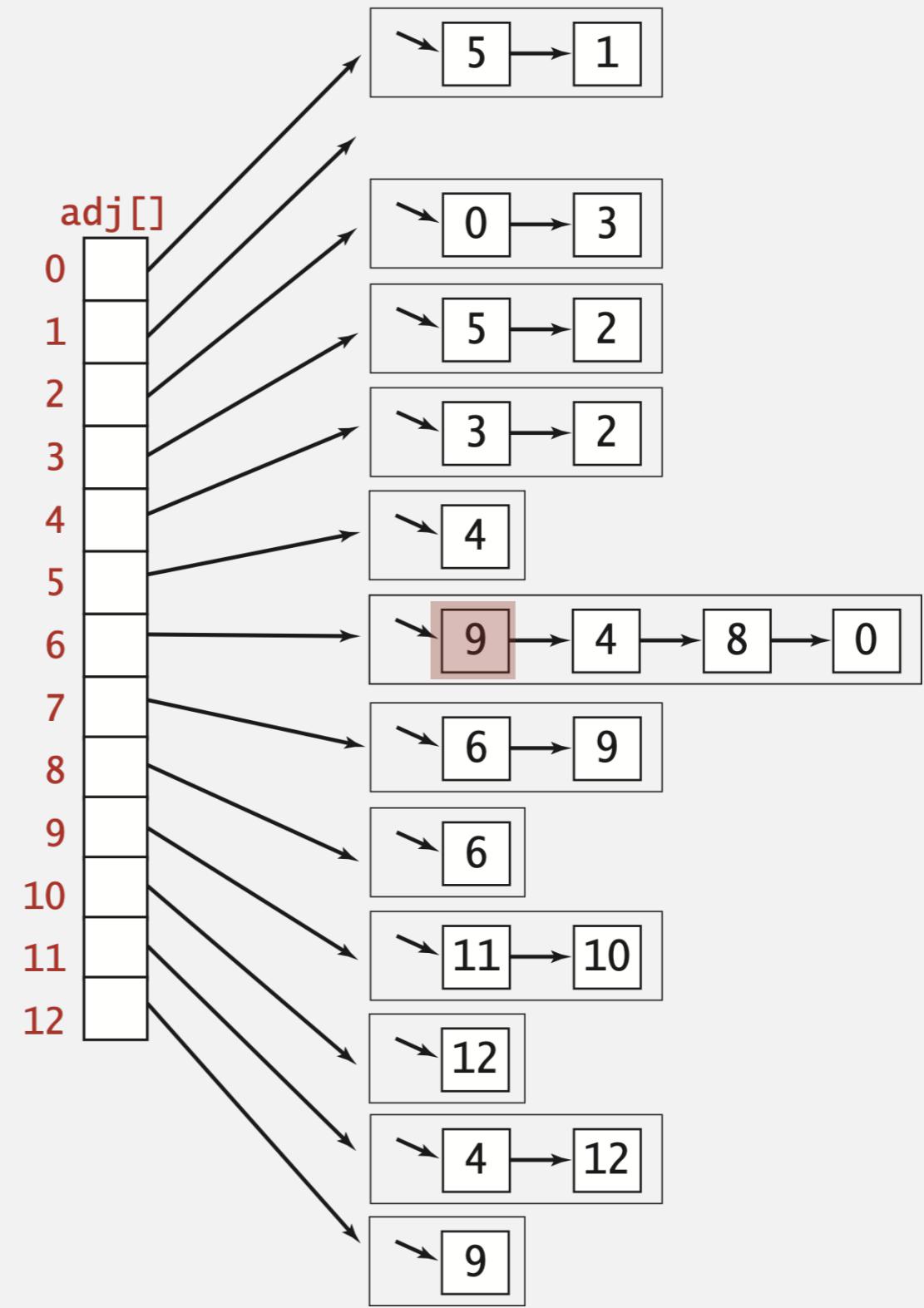
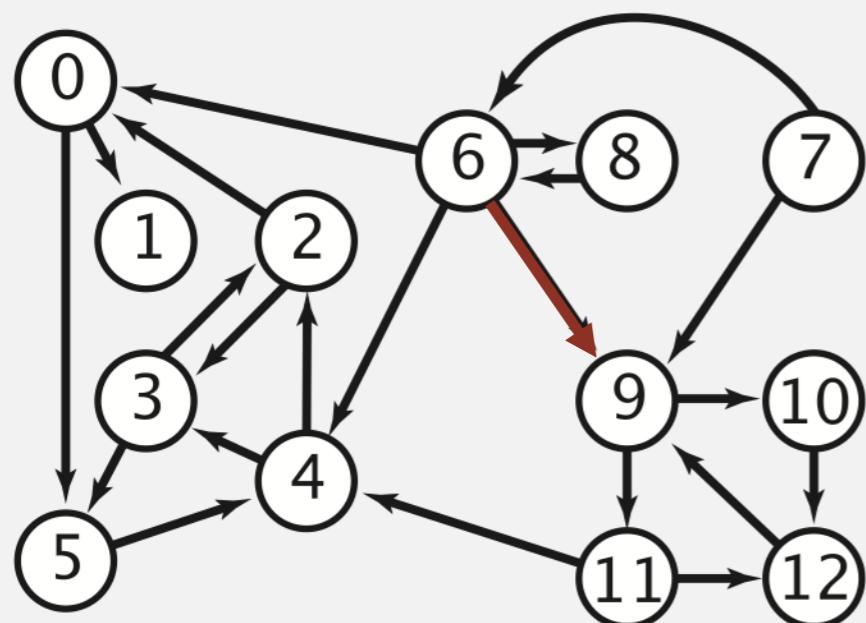
Note: parallel edges disallowed

from	to	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	1	0	0	0	0	0	0	0	0
4	0	0	1	1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	1	0	0	0	1	1	0	0	0
7	0	0	0	0	0	0	1	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	1	0	0	0	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	1	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	1	0	0

# Digraph representation: adjacency lists

LO 9B.6

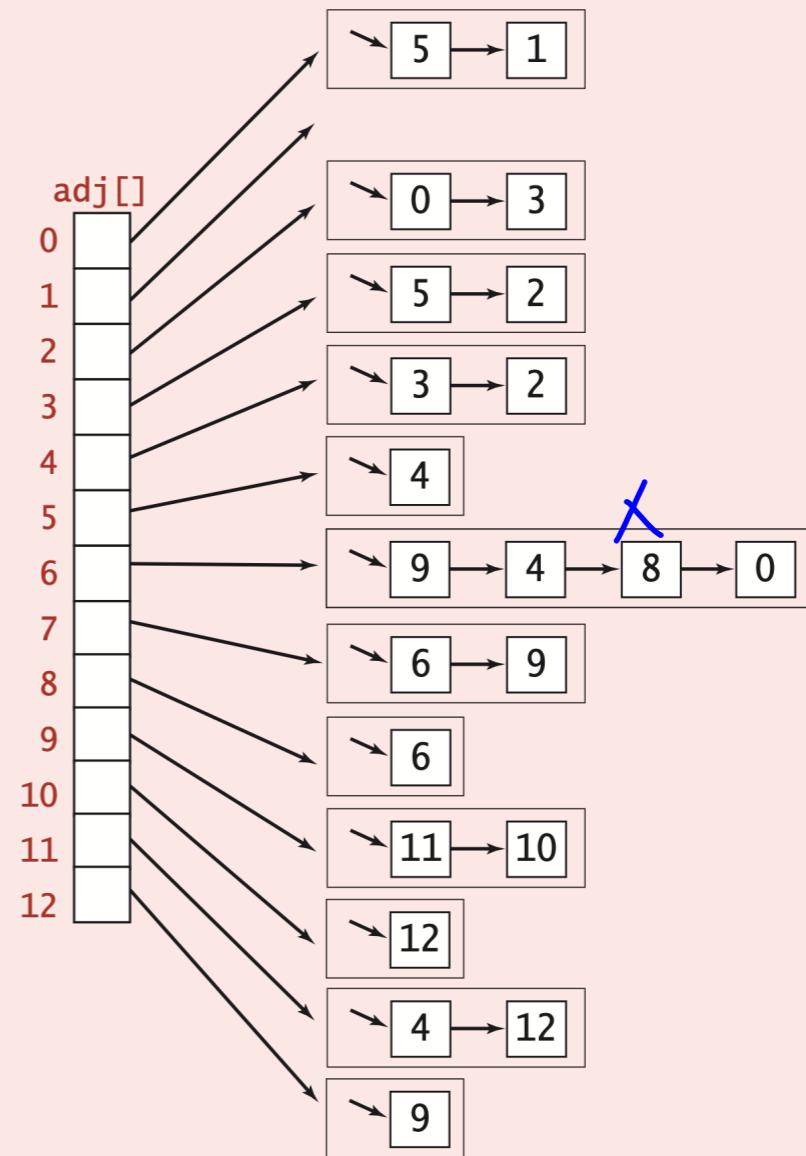
Maintain vertex-indexed array of lists.



Which is order of growth of running time of removing an edge  $v \rightarrow w$  from a digraph uses the adjacency-lists representation, where  $V$  is the number of vertices and  $E$  is the number of edges?

$6 \rightarrow 8$

- A. 1
- B.  $\text{outdegree}(v)$
- C.  $\text{indegree}(w)$
- D.  $\text{outdegree}(v) + \text{indegree}(w)$

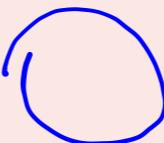


Which is order of growth of running time of the following code fragment if the digraph uses the adjacency-lists representation, where  $V$  is the number of vertices and  $E$  is the number of edges?

- A.  $V$
- B.  $\underline{E} + \underline{V}$
- C.  $V^2$
- D.  $VE$

```
for (int v = 0; v < G.V(); v++)  
    for (int w : G.adj(v))  
        StdOut.println(v + "->" + w);
```

prints each edge exactly once



# Digraph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent from  $v$ .
- Real-world graphs tend to be sparse (not dense).

↑  
proportional  
to  $V$  edges      ↑  
proportional  
to  $V^2$  edges

representation	space	insert edge from $v$ to $w$	edge from $v$ to $w$ ?	iterate over vertices adjacent from $v$ ?
list of edges	$E$	1	$E$	$E$
adjacency matrix	$V^2$	$1^\dagger$	1	$V$
adjacency lists	$E + V$	1	$outdegree(v)$	$outdegree(v)$

† disallows parallel edges

# Adjacency-lists graph representation (review): Java implementation

```
public class Graph
{
    private final int V;
    private Bag<Integer>[] adj;
```

← adjacency lists

```
public Graph(int V)
{
    this.V = V;
    adj = (Bag<Integer>[]) new Bag[V];
    for (int v = 0; v < V; v++)
        adj[v] = new Bag<Integer>();
}
```

← create empty graph  
with V vertices

```
public void addEdge(int v, int w)
{
    adj[v].add(w);
    adj[w].add(v);
}
```

← add edge v-w

```
public Iterable<Integer> adj(int v)
{ return adj[v]; }
```

← iterator for vertices  
adjacent to v

# Adjacency-lists digraph representation: Java implementation

```
public class Digraph
{
    private final int V;
    private Bag<Integer>[] adj;
```

← adjacency lists

```
public Digraph(int V)
{
    this.V = V;
    adj = (Bag<Integer>[]) new Bag[V];
    for (int v = 0; v < V; v++)
        adj[v] = new Bag<Integer>();
}
```

← create empty digraph  
with V vertices

```
public void addEdge(int v, int w)
{
    adj[v].add(w);
}
```

← add edge v→w

```
public Iterable<Integer> adj(int v)
{ return adj[v]; }
```

← iterator for vertices  
adjacent from v

# DIRECTED GRAPHS

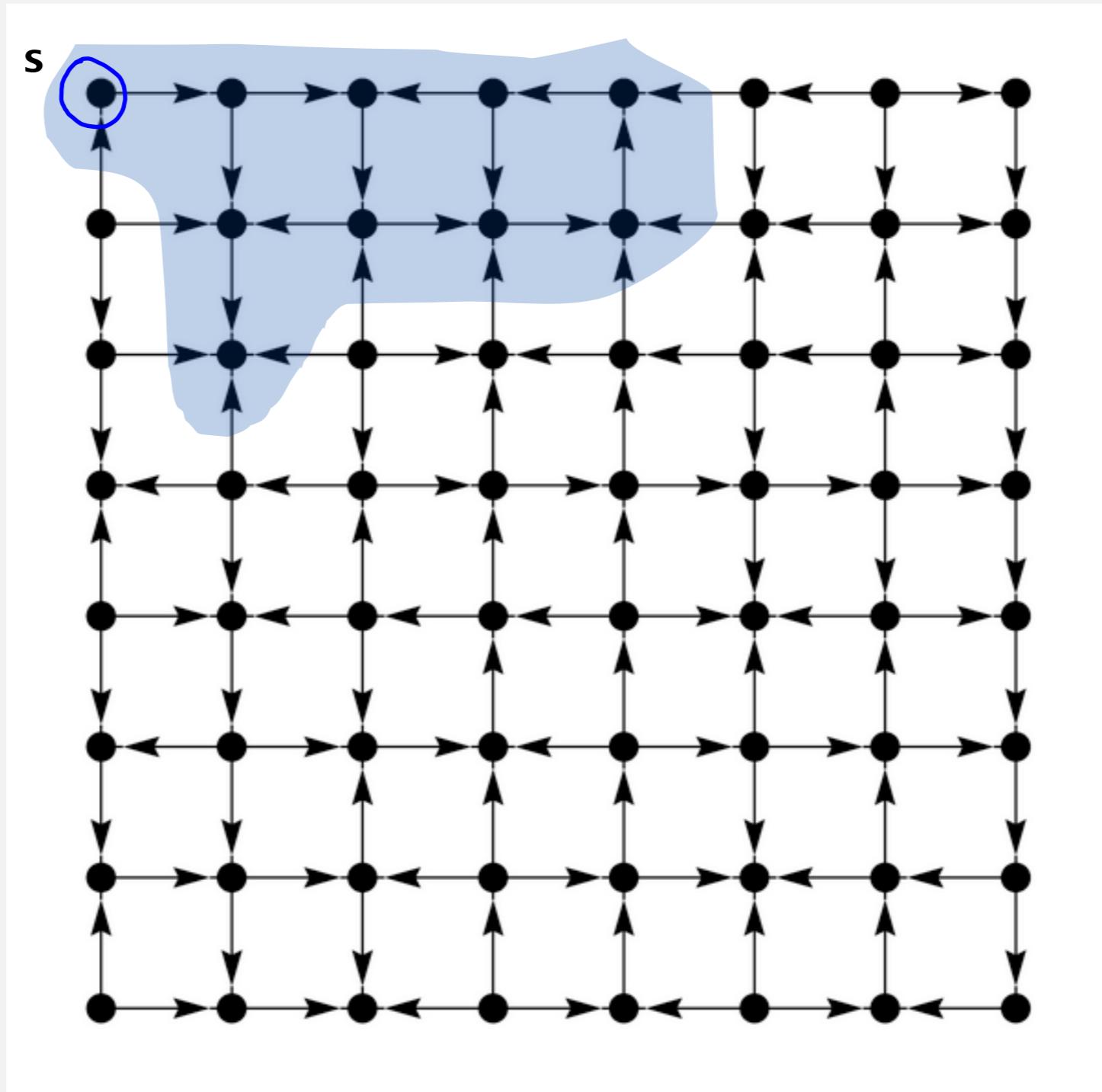
---

- ▶ *introduction*
- ▶ *digraph API*
- ▶ ***depth-first search***
- ▶ *breadth-first search*
- ▶ *topological sort*

# Reachability

---

Problem. Find all vertices reachable from  $s$  along a directed path.



# Depth-first search in digraphs

---

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a **digraph** algorithm.

**DFS (to visit a vertex v)**

**Mark vertex v.**

**Recursively visit all unmarked  
vertices w adjacent from v.**

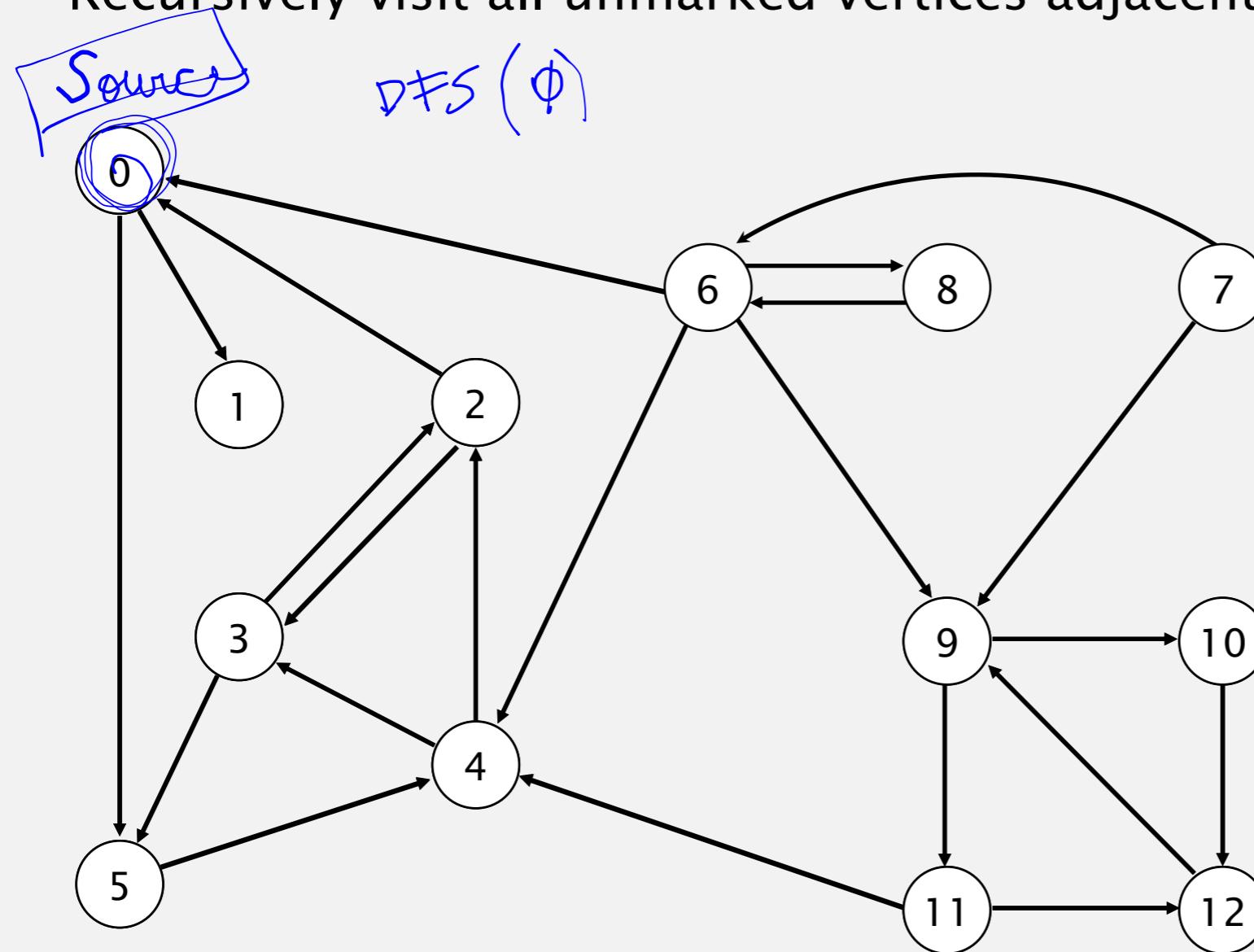
# Depth-first search demo

LO 9B.8

4→2

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent from  $v$ .



2→3

3→2

6→0

0→1

2→0

11→12

12→9

9→10

9→11

8→9

10→12

11→4

4→3

3→5

6→8

8→6

5→4

0→5

6→4

6→9

7→6

a directed graph

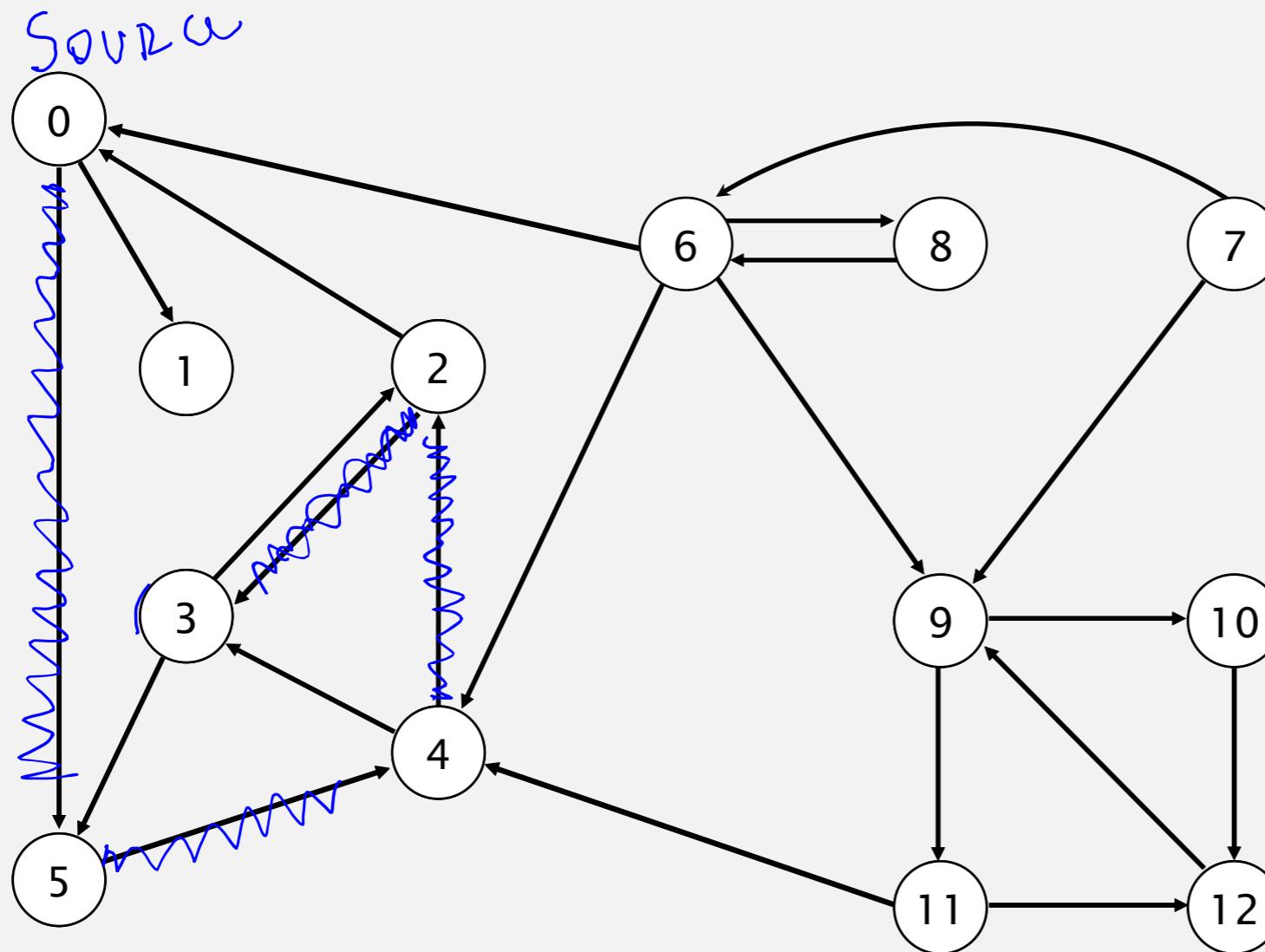
# Depth-first search demo

LO 9B.8

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent from  $v$ .

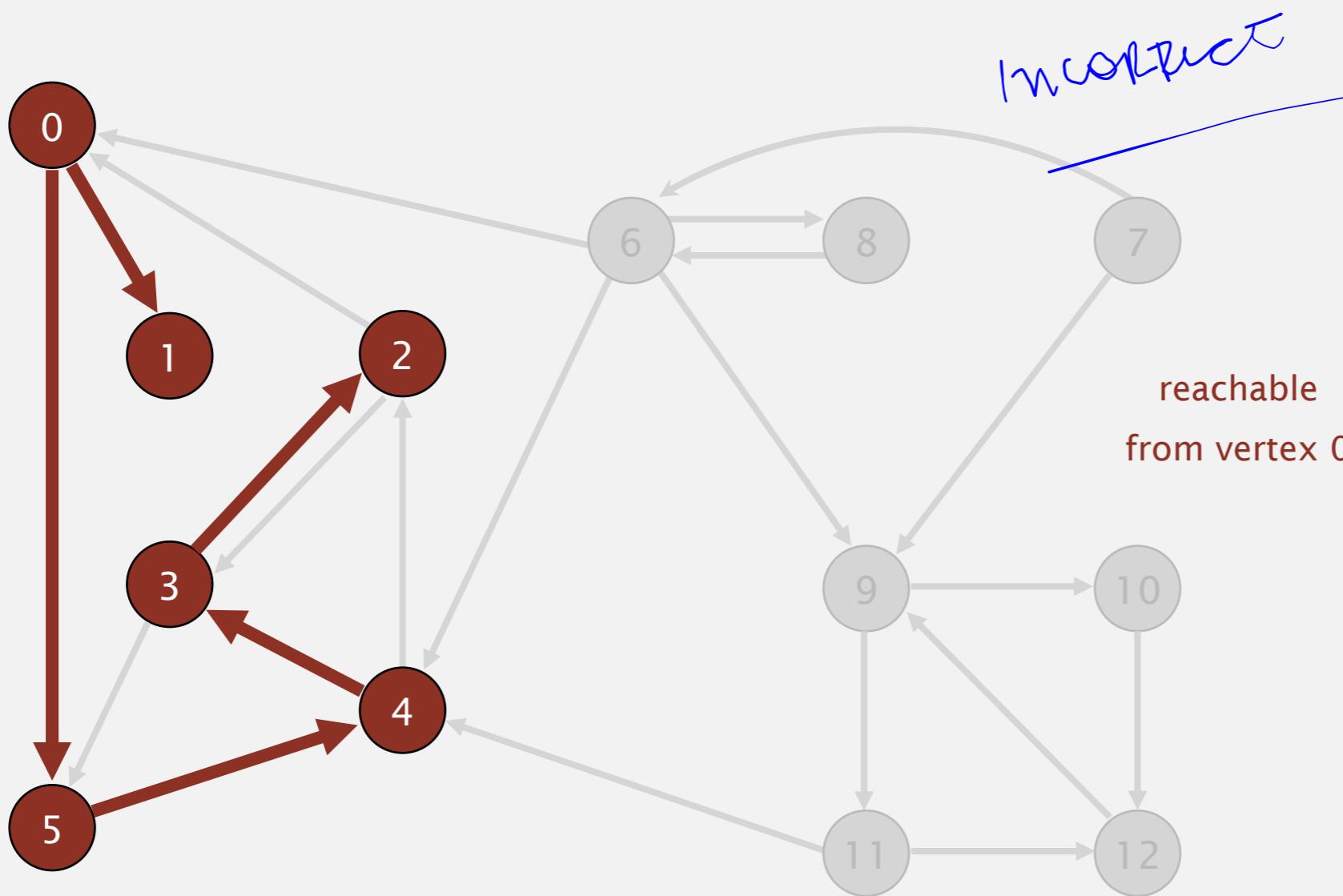
DFS(0)



$v$	marked[]	edgeTo[]
0	✓	
1	✓	0
2	✓	4
3	✓	2
4	✓	5
5	✓	0
6		
7		
8		
9		
10		
11		
12		

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent from  $v$ .



**reachable from 0**

$v$	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

# Depth-first search (in undirected graphs)

Recall code for undirected graphs.

```
public class DepthFirstSearch
{
    private boolean[] marked;

    public DepthFirstSearch(Graph G, int s)
    {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }

    public boolean visited(int v)
    {
        return marked[v];
    }
}
```

Annotations from right to left:

- marked[v] = true; → true if connected to s
- dfs(G, s); → constructor marks vertices connected to s
- dfs(G, w); → recursive DFS does the work
- return marked[v]; → client can ask whether any vertex is connected to s

# Depth-first search (in directed graphs)

Code for **directed** graphs identical to undirected one.

```
public class DirectedDFS
{
    private boolean[] marked;

    public DirectedDFS(Digraph G, int s)
    {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }

    public boolean visited(int v)
    { return marked[v]; }
}
```

Annotations from right to left:

- marked[v] = true; → true if connected to s
- dfs(G, s); → constructor marks vertices connected to s
- dfs(G, w); → recursive DFS does the work
- return marked[v]; → client can ask whether any vertex is connected to s

Every program is a digraph.

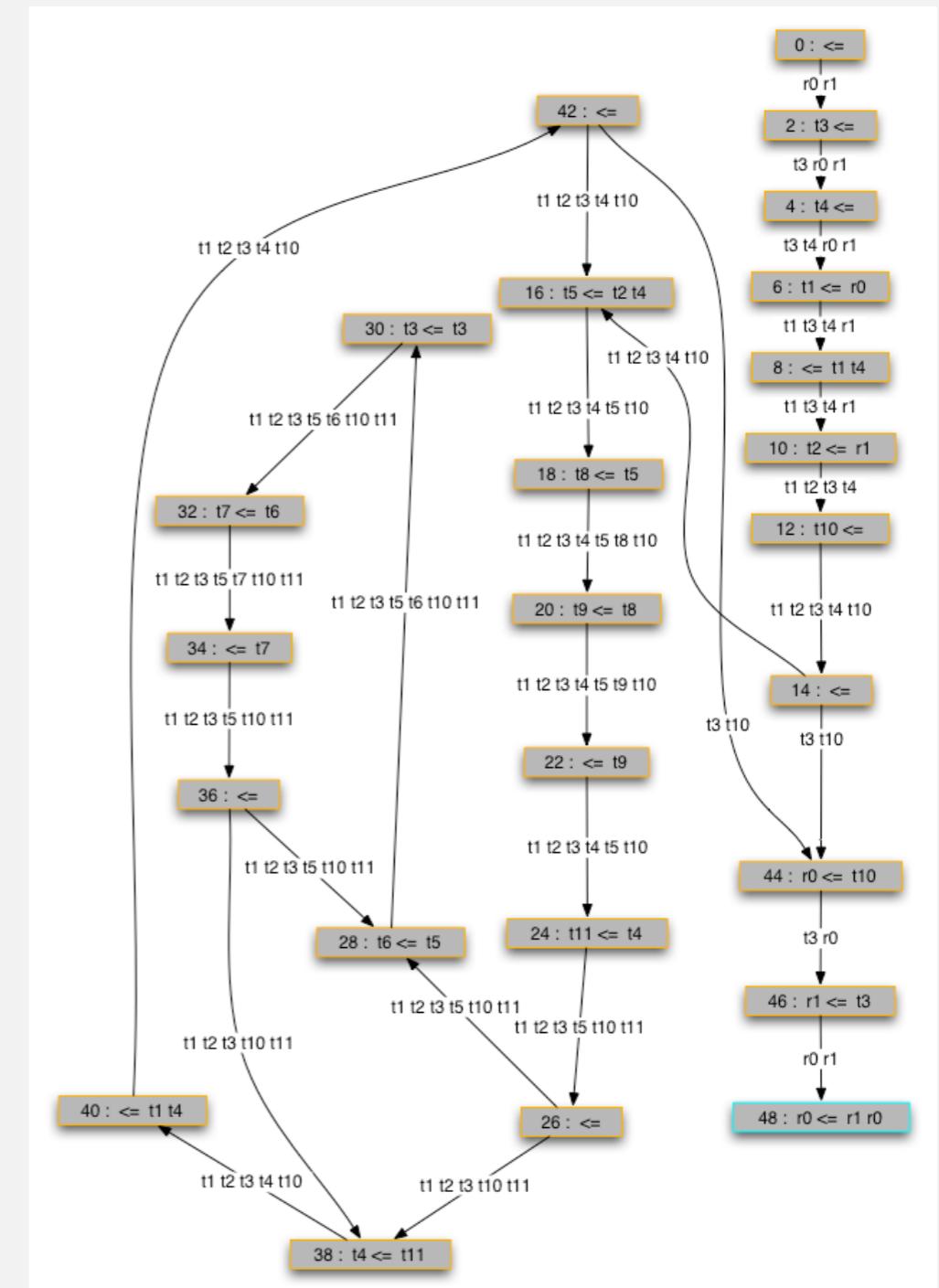
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.

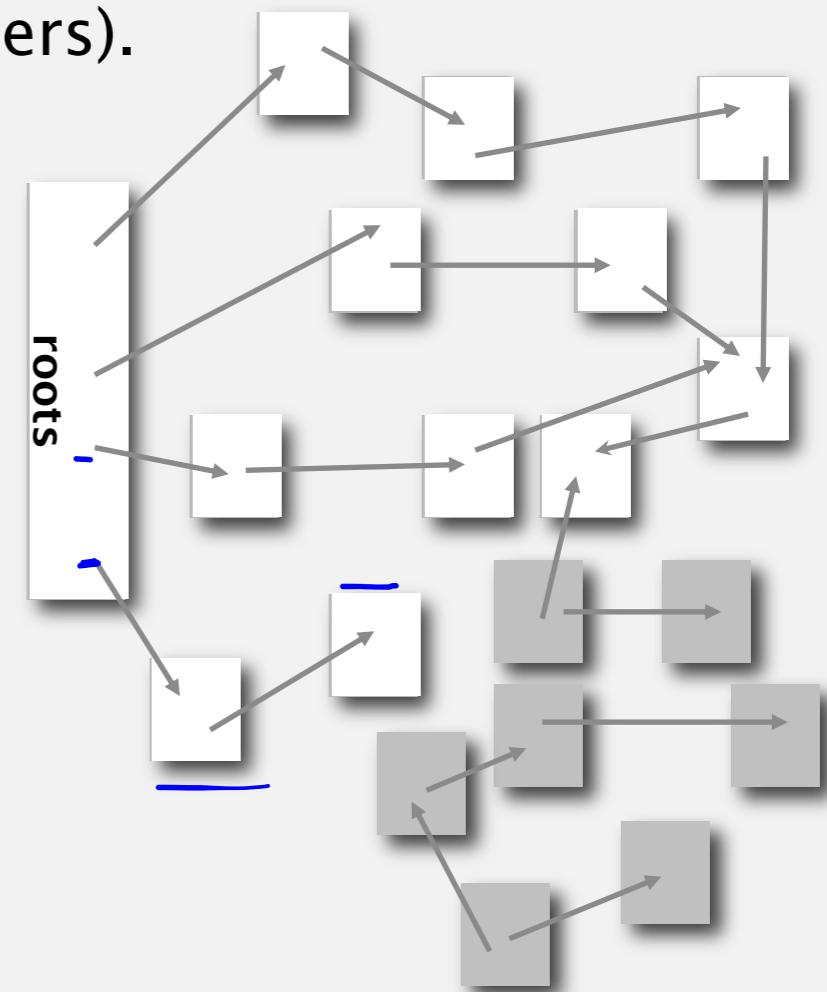


Every data structure is a digraph.

- Vertex = object.
- Edge = reference.

**Roots.** Objects known to be directly accessible by program (e.g., stack).

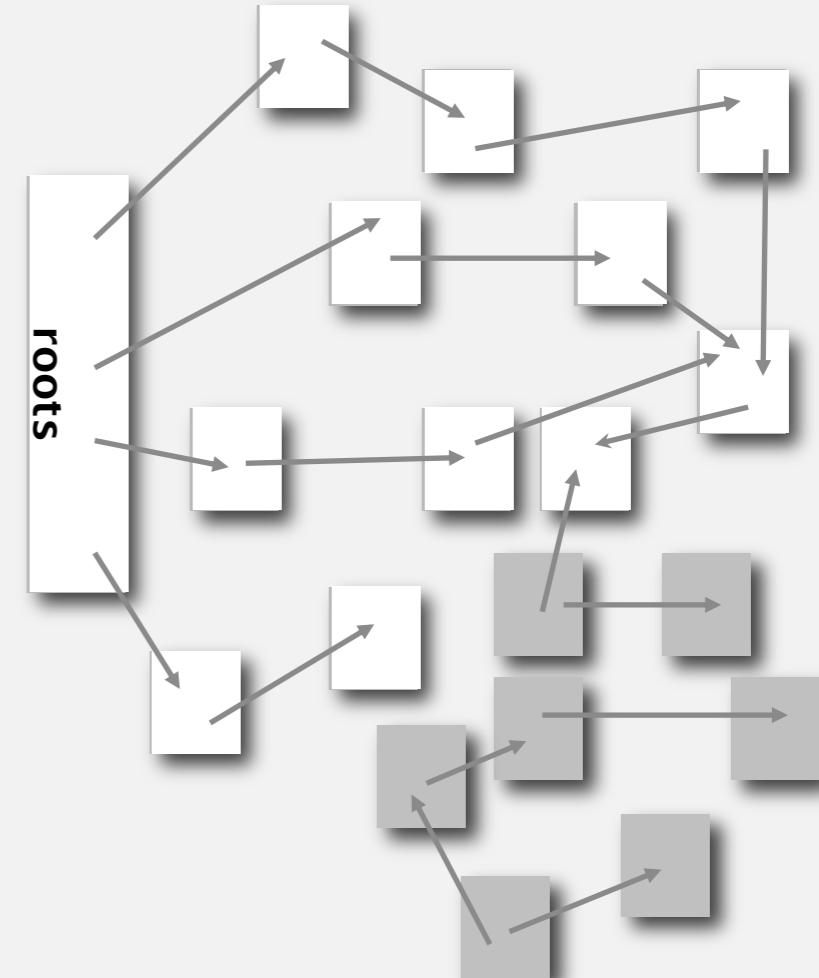
**Reachable objects.** Objects indirectly accessible by program  
(starting at a root and following a chain of pointers).



Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).



DFS enables direct solution of simple digraph problems.

- ✓ □ Reachability.
- Path finding.
- Topological sort.
- Directed cycle detection.

Basis for solving difficult digraph problems.

- 2-satisfiability.
- Directed Euler path.
- Strongly-connected components.

SIAM J. COMPUT.  
Vol. 1, No. 2, June 1972

## DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS\*

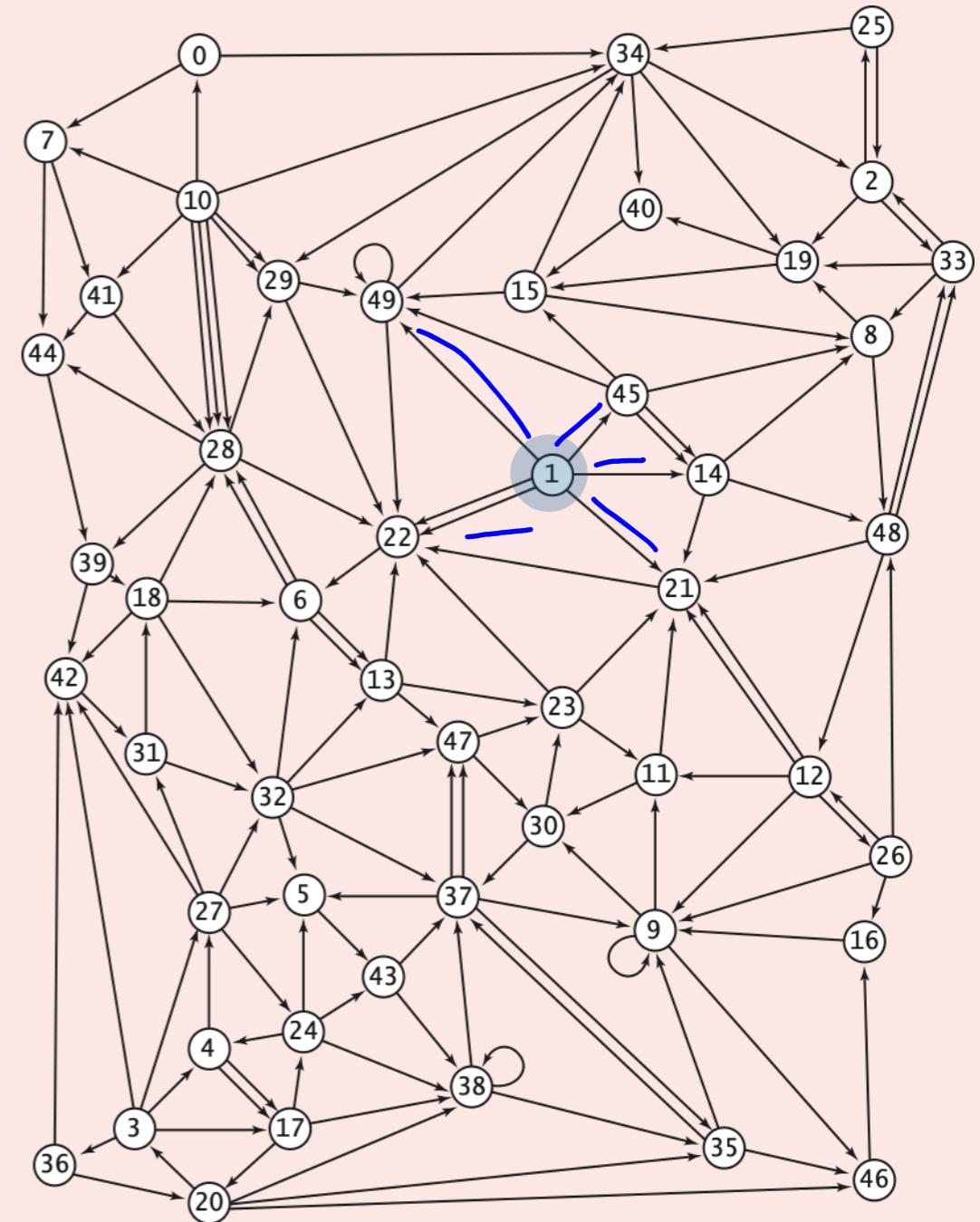
ROBERT TARJAN†

**Abstract.** The value of depth-first search or “backtracking” as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirected graph are presented. The space and time requirements of both algorithms are bounded by  $k_1V + k_2E + k_3$  for some constants  $k_1, k_2$ , and  $k_3$ , where  $V$  is the number of vertices and  $E$  is the number of edges of the graph being examined.

## Directed graphs: quiz 3

Suppose that you want to design a web crawler. Which graph search algorithm should you use?

- A. depth-first search
- B. breadth-first search
- C. either A or B
- D. neither A nor B



# Web crawler output (an example)

## BFS crawl

http://www.rutgers.edu  
http://www.w3.org  
http://ogp.me  
http://giving.Rutgers.edu  
http://www.princetonartmuseum.org  
http://www.goprinctontigers.com  
http://library.Rutgers.edu  
http://helpdesk.princeton.edu  
http://tigernet.Rutgers.edu  
http://alumni.princeton.edu  
http://gradschool.princeton.edu  
http://vimeo.com  
http://princetonusg.com  
http://artmuseum.princeton.edu  
http://jobs.princeton.edu  
http://odoc.princeton.edu  
http://blogs.princeton.edu  
http://www.facebook.com  
http://twitter.com  
http://www.youtube.com  
http://deimos.apple.com  
http://qeprize.org  
http://en.wikipedia.org

## DFS crawl

http://www.rutgers.edu  
http://deimos.apple.com  
http://www.youtube.com  
http://www.google.com  
http://news.google.com  
http://csi.gstatic.com  
http://googlenewsblog.blogspot.com  
http://labs.google.com  
http://groups.google.com  
http://img1.blogblog.com  
http://feeds.feedburner.com  
http://buttons.googlesyndication.com  
http://fusion.google.com  
http://insidesearch.blogspot.com  
http://agooleaday.com  
http://static.googleusercontent.com  
http://searchresearch1.blogspot.com  
http://feedburner.google.com  
http://www.dot.ca.gov  
http://www.TahoeRoads.com  
http://www.LakeTahoeTransit.com  
http://www.laketahoe.com  
http://ethel.tahoeguide.com

# DIRECTED GRAPHS

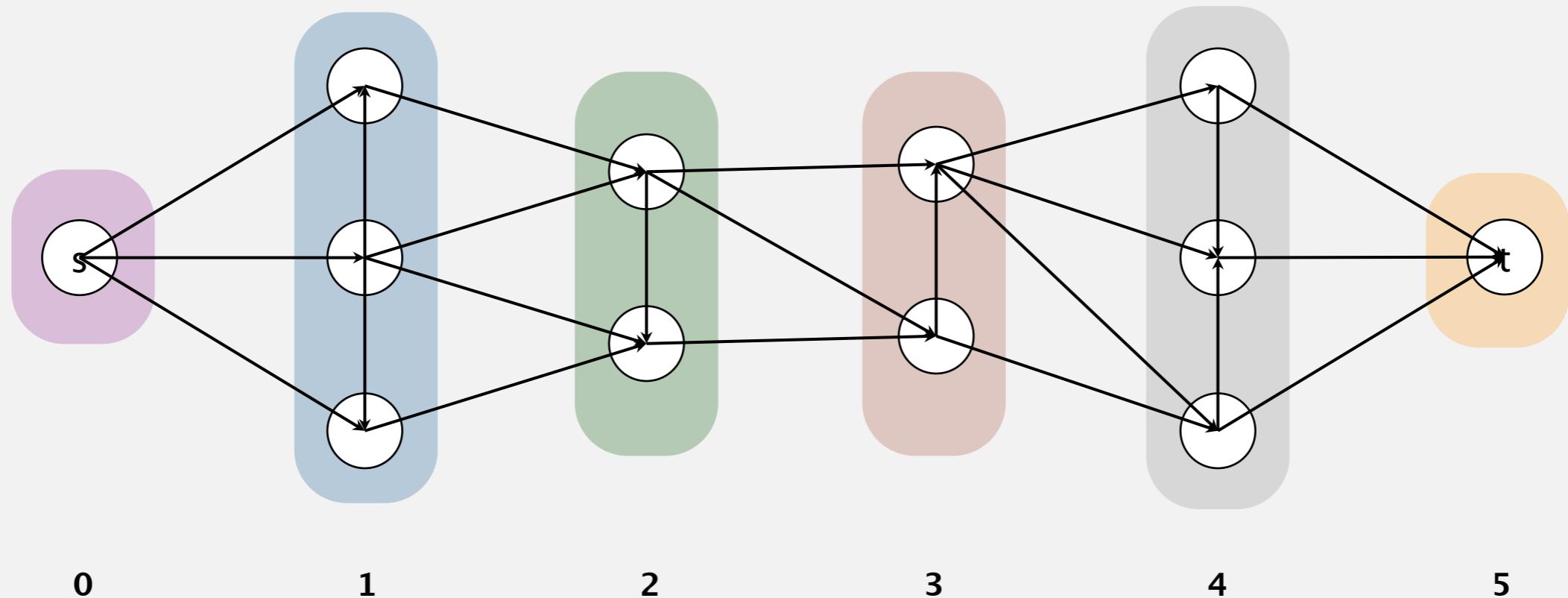
---

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *depth-first search*
- ▶ ***breadth-first search***
- ▶ *topological sort*

## Shortest directed paths

LO 9B.14

Problem. Find directed path from s to each vertex that uses fewest edges.



## Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a **digraph** algorithm.

### BFS (from source vertex $s$ )

Put  $\underline{s}$  onto a FIFO queue, and mark  $\underline{s}$  as visited.

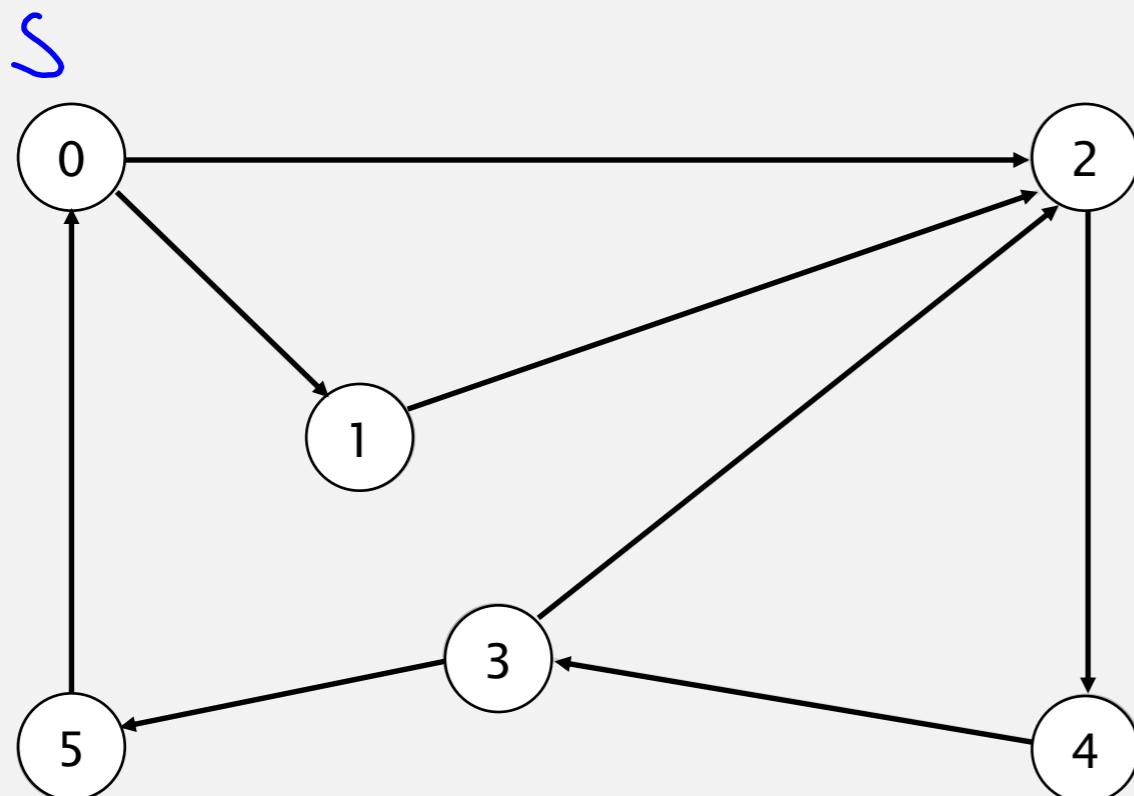
Repeat until the queue is empty:

- remove the least recently added vertex  $v$
- for each unmarked vertex adjacent from  $v$ :  
add to queue and mark as visited.

Proposition. BFS computes directed path with fewest edges from  $s$  to each vertex in time proportional to  $E + V$ .

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent from  $v$  and mark them.



*tinyDG2.txt*

<i>V</i>	6	<i>E</i>
	8	
	5 0	
	2 4	
	3 2	
	1 2	
	0 1	
	4 3	
	3 5	
	0 2	

**graph G**

# Directed breadth-first search demo

LO 9B.10

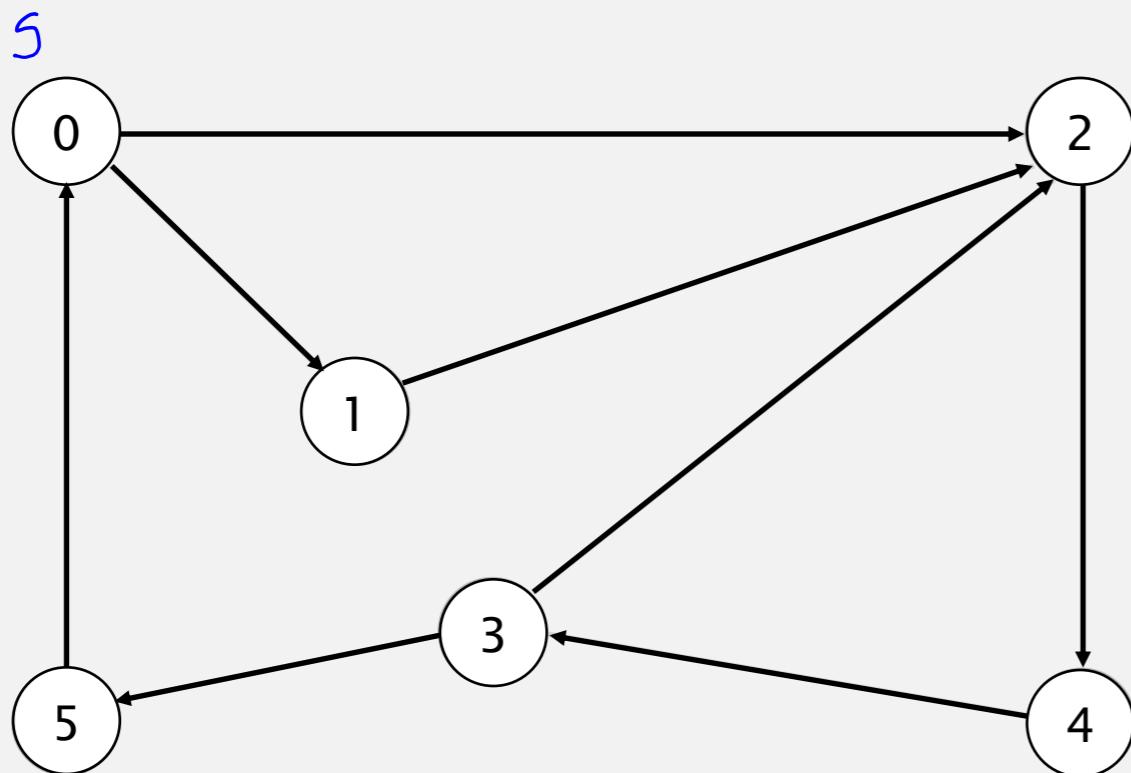
Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent from  $v$  and mark them.



BFS(0)

Queue:  $\emptyset \times \{A, B, C\}$



Marked	v	edgeTo[]	distTo[]
✓	0		0
✓	1	0	1
✓	2	0	1
✓	3	4	3
✓	4	2	2
✓	5	3	4

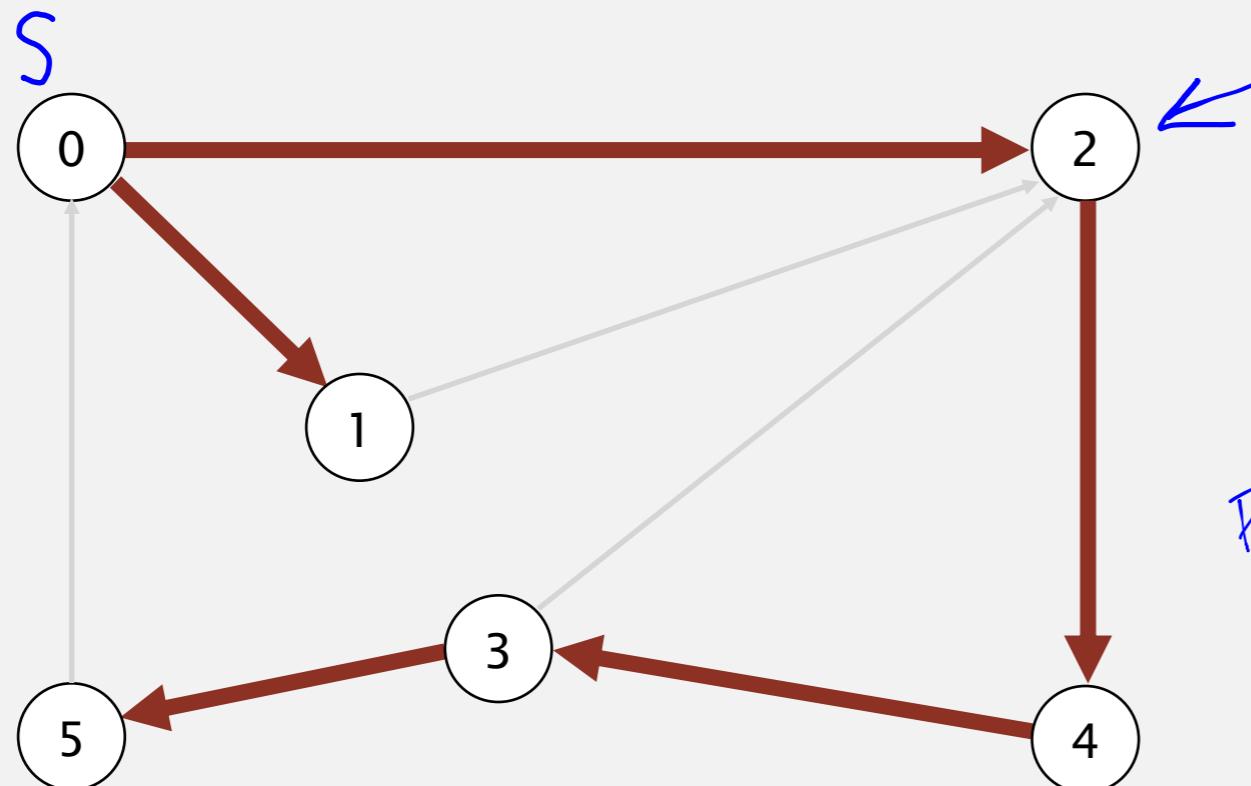
graph G

# Directed breadth-first search demo

LO 9B.10

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent from  $v$  and mark them.



$v$	edgeTo[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

Handwritten notes: Path (0,3) with an arrow pointing to edge 0-3. A stack-like diagram shows vertices 0, 2, 4, 3 in a vertical column, with an arrow pointing to the bottom vertex 3 and the text "0,2,4,3".

done

# MULTIPLE-SOURCE SHORTEST PATHS (OPTIONAL)

LO 9B.14

Given a digraph and a set of source vertices, find shortest path from any vertex in the set to every other vertex.

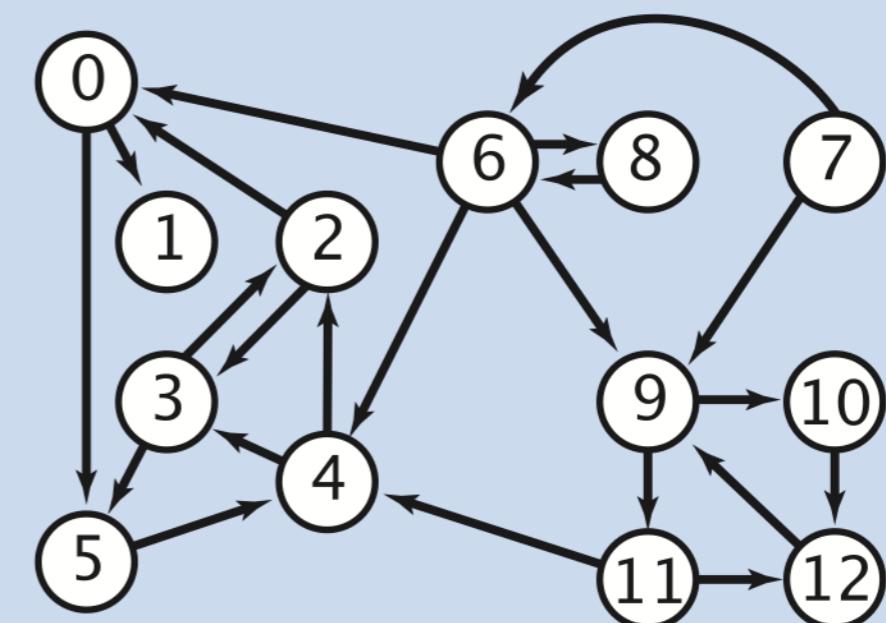
Ex.  $S = \{ 1, 7, 10 \}$ .

- Shortest path to 4 is  $7 \rightarrow 6 \rightarrow 4$ .
- Shortest path to 5 is  $7 \rightarrow 6 \rightarrow 0 \rightarrow 5$ .
- Shortest path to 12 is  $10 \rightarrow 12$ .

BFS(1)

BFS(7)

BFS(10)



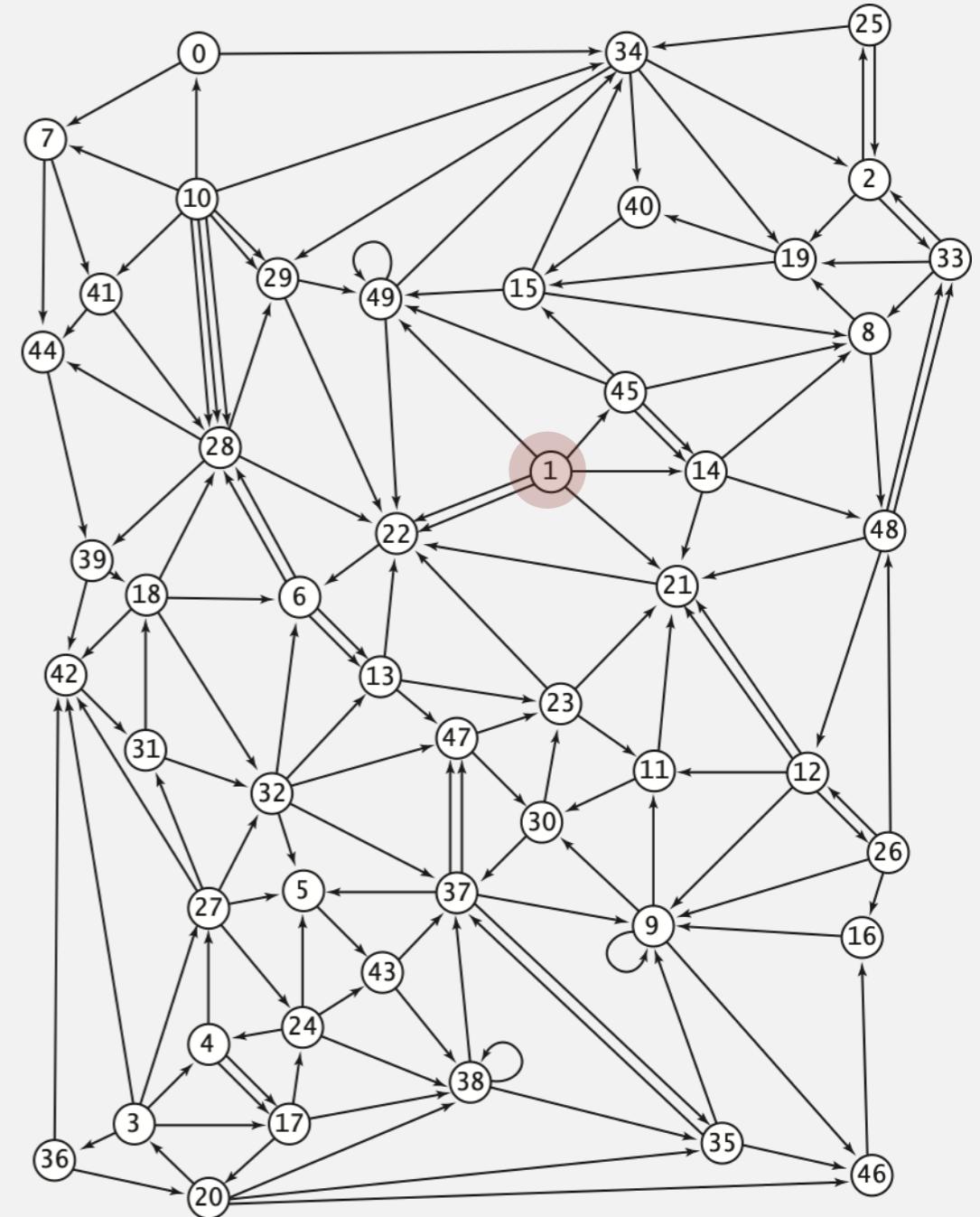
Q. How to implement multi-source shortest paths algorithm?

# Breadth-first search in digraphs application: web crawler

**Goal.** Crawl web, starting from some root web page, say [www.rutgers.edu](http://www.rutgers.edu).

**Solution.** [BFS with implicit digraph]

- Choose root web page as source  $s$ .
- Maintain a Queue of websites to explore.
- Maintain a SET of marked websites.
- Dequeue the next website and enqueue any unmarked websites to which it links.



# Bare-bones web crawler: Java implementation

```
Queue<String> queue = new Queue<String>();  
SET<String> marked = new SET<String>();
```

```
String root = "http://www.rutgers.edu";  
queue.enqueue(root);  
marked.add(root);  
while (!queue.isEmpty()) {
```

```
    String v = queue.dequeue();  
    StdOut.println(v);  
    In in = new In(v);  
    String input = in.readAll();
```

```
    String regexp = "http://(\w+\.\w+)+(\w+)";  
    Pattern pattern = Pattern.compile(regexp);  
    Matcher matcher = pattern.matcher(input);
```

```
    while (matcher.find()) {  
        String w = matcher.group();  
  
        if (!marked.contains(w)) {  
            marked.add(w);  
            queue.enqueue(w);  
        }  
    }  
}
```

queue of websites to crawl  
set of marked websites

start crawling from root website

read in raw html from next  
website in queue

use regular expression to find all URLs  
in website of form http://xxx.yyy.zzz  
[crude pattern misses relative URLs]

if unmarked, mark and enqueue

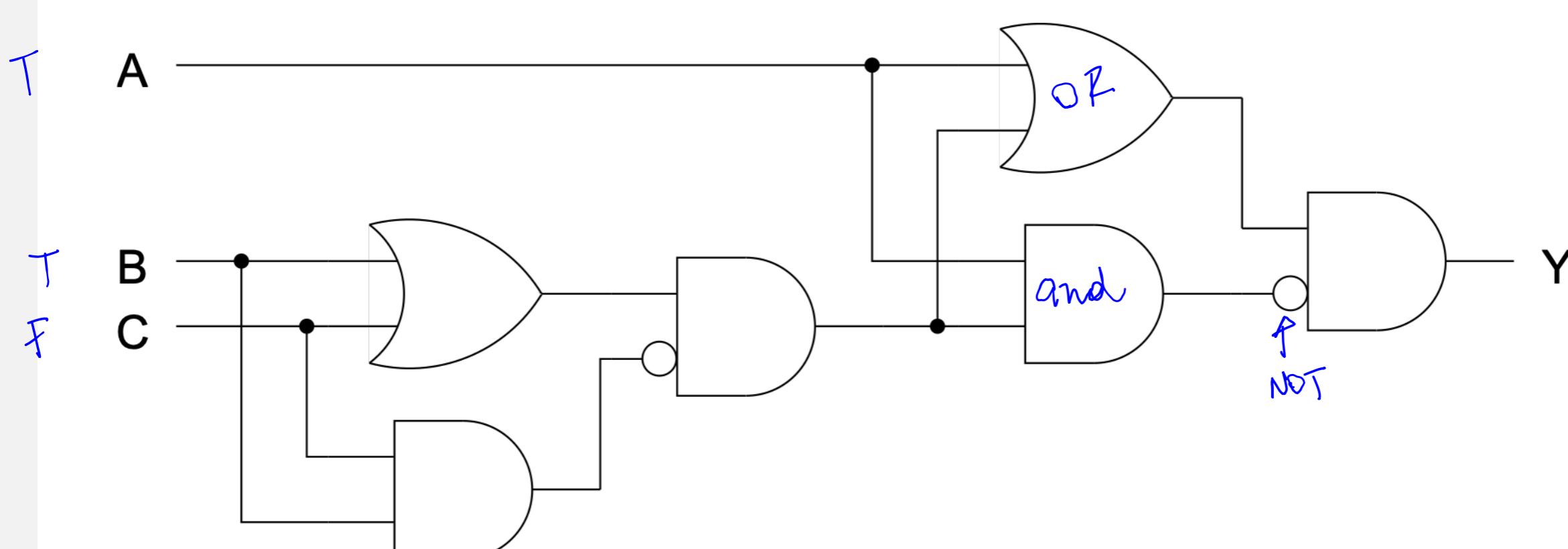
# DIRECTED GRAPHS

---

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *depth-first search*
- ▶ *breadth-first search*
- ▶ ***topological sort***

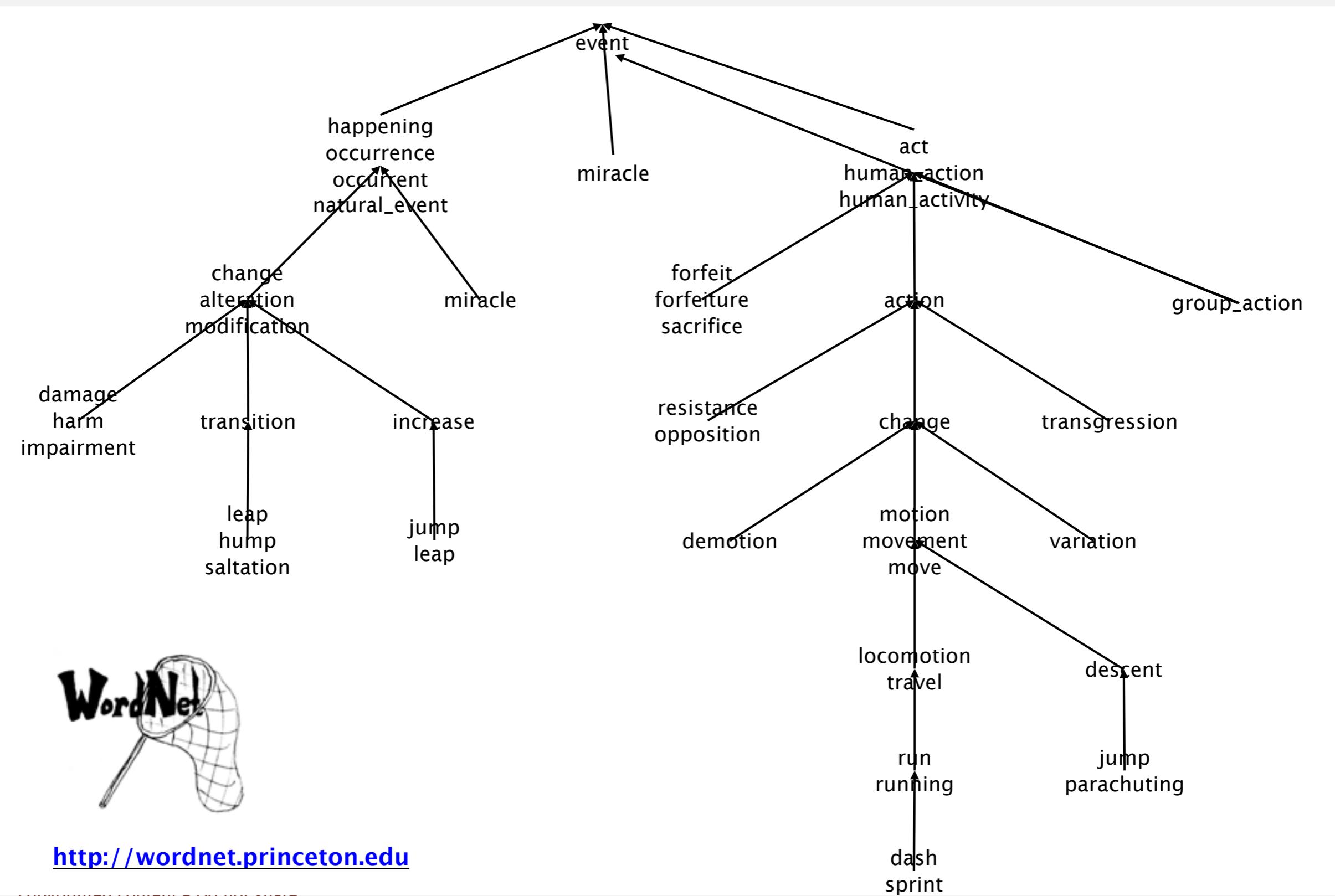
# Combinational circuit (optional)

Vertex = logical gate; edge = wire.



# WordNet digraph

Vertex = synset; edge = hypernym relationship.



# Git digraph



---

Vertex = revision of repository; edge = revision relationship.

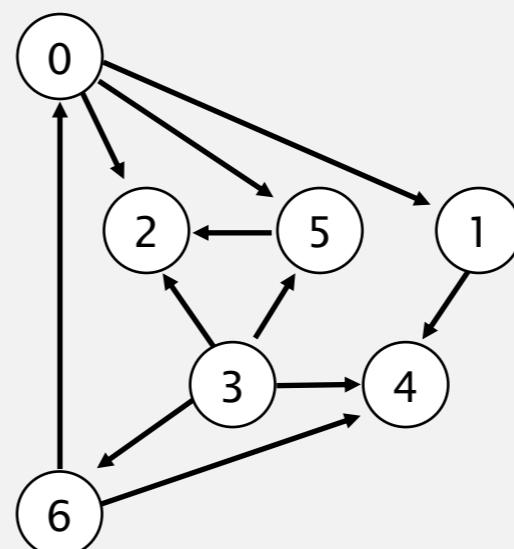
# Precedence scheduling

LO 9B.15

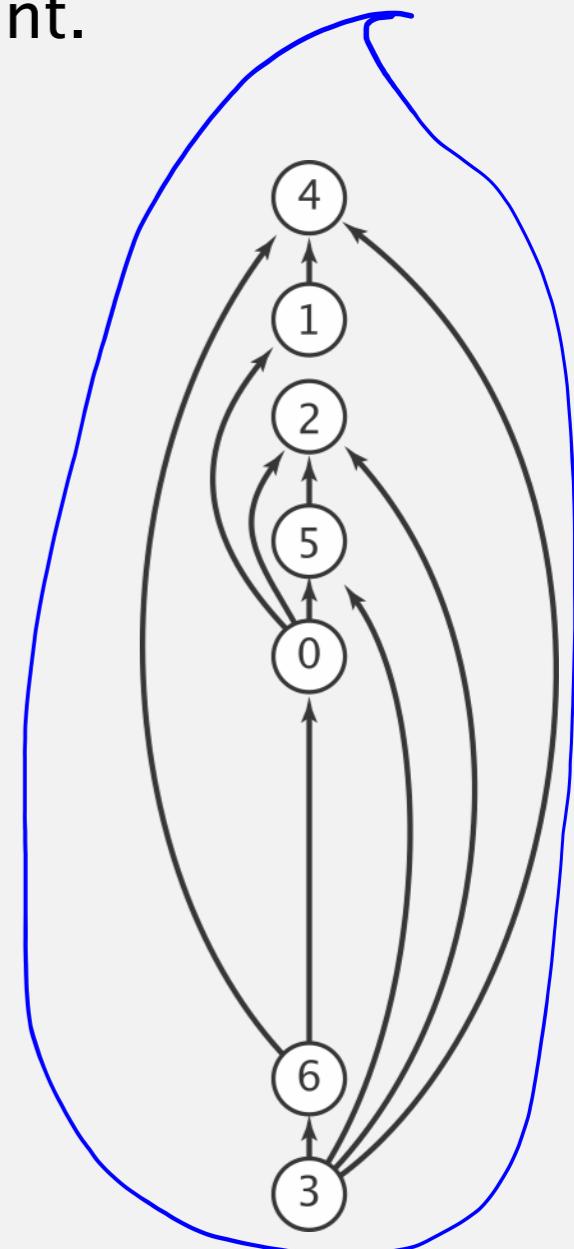
**Goal.** Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

Course      Pre Req  
Digraph model. vertex = task; edge = precedence constraint.

0. Algorithms
  1. Complexity Theory
  2. Artificial Intelligence
  3. Intro to CS
  4. Cryptography
  5. Scientific Computing
  6. Advanced Programming



## precedence constraint graph



## **feasible schedule**

# Topological sort

chicken  $\rightarrow$  egg

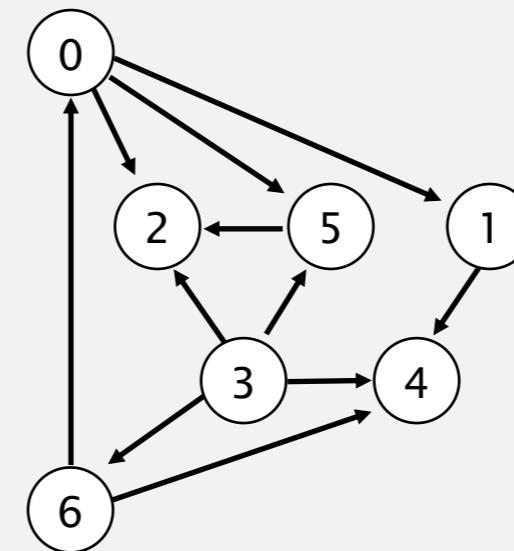
LO 9B.12, 9B.15  
NOT DAG

DAG. Directed acyclic graph.

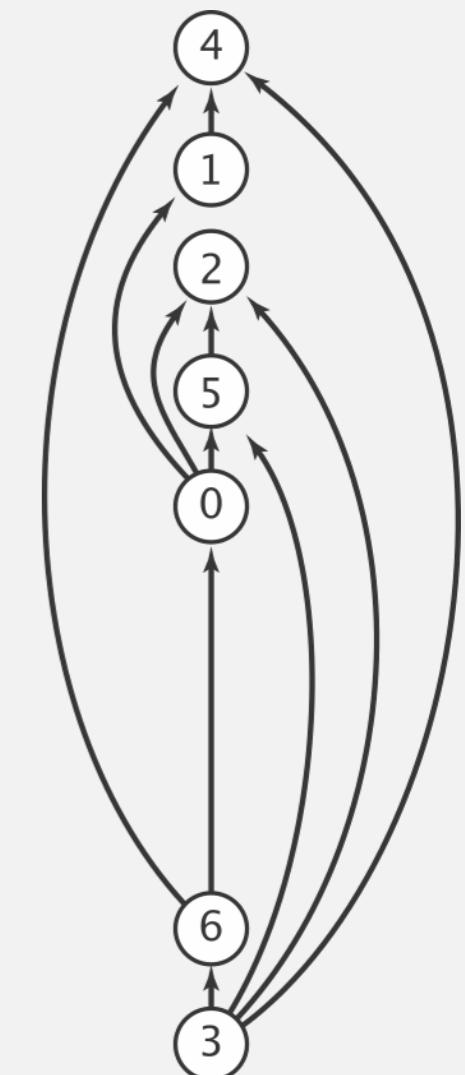
Topological sort. Redraw DAG so all edges point upwards.

$0 \rightarrow 5$      $0 \rightarrow 2$   
 $0 \rightarrow 1$      $3 \rightarrow 6$   
 $3 \rightarrow 5$      $3 \rightarrow 4$   
 $5 \rightarrow 2$      $6 \rightarrow 4$   
 $6 \rightarrow 0$      $3 \rightarrow 2$   
 $1 \rightarrow 4$

directed edges



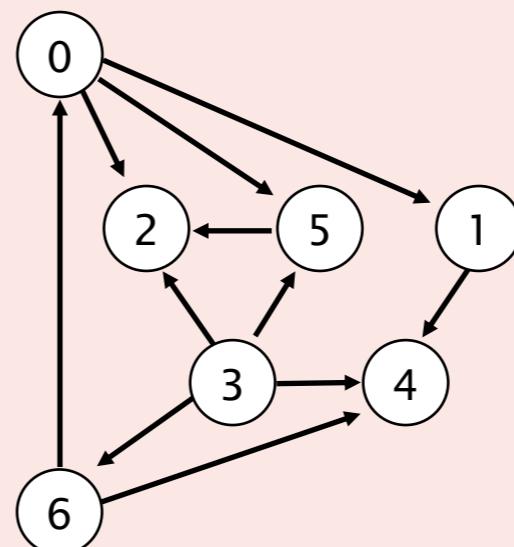
DAG



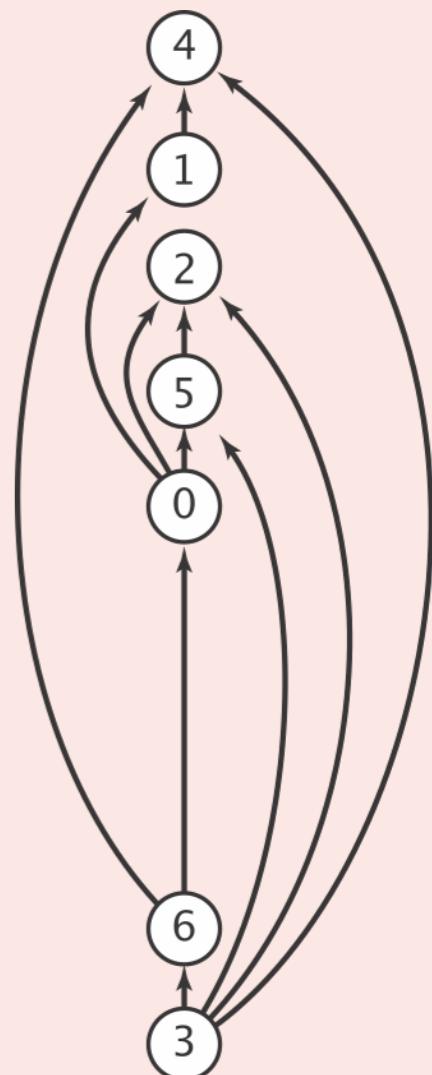
topological order

Suppose that you want to find a topological order of a DAG.  
Which graph search algorithm should you use?

- A. depth-first search
- B. breadth-first search
- C. either A or B
- D. neither A nor B



DAG

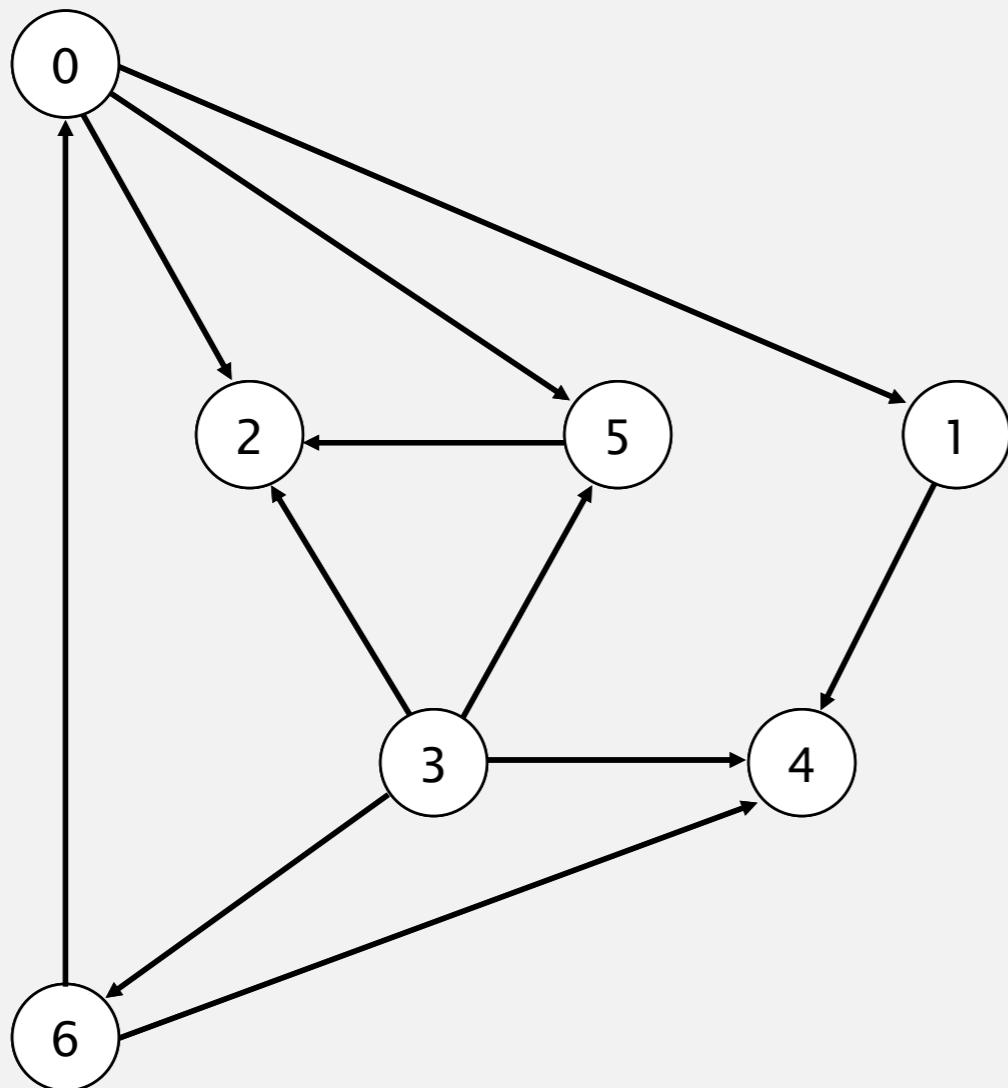


topological order

# Topological sort demo

LO 9B.15

- Run depth-first search.
- Return vertices in reverse postorder.



**tinyDAG7.txt**

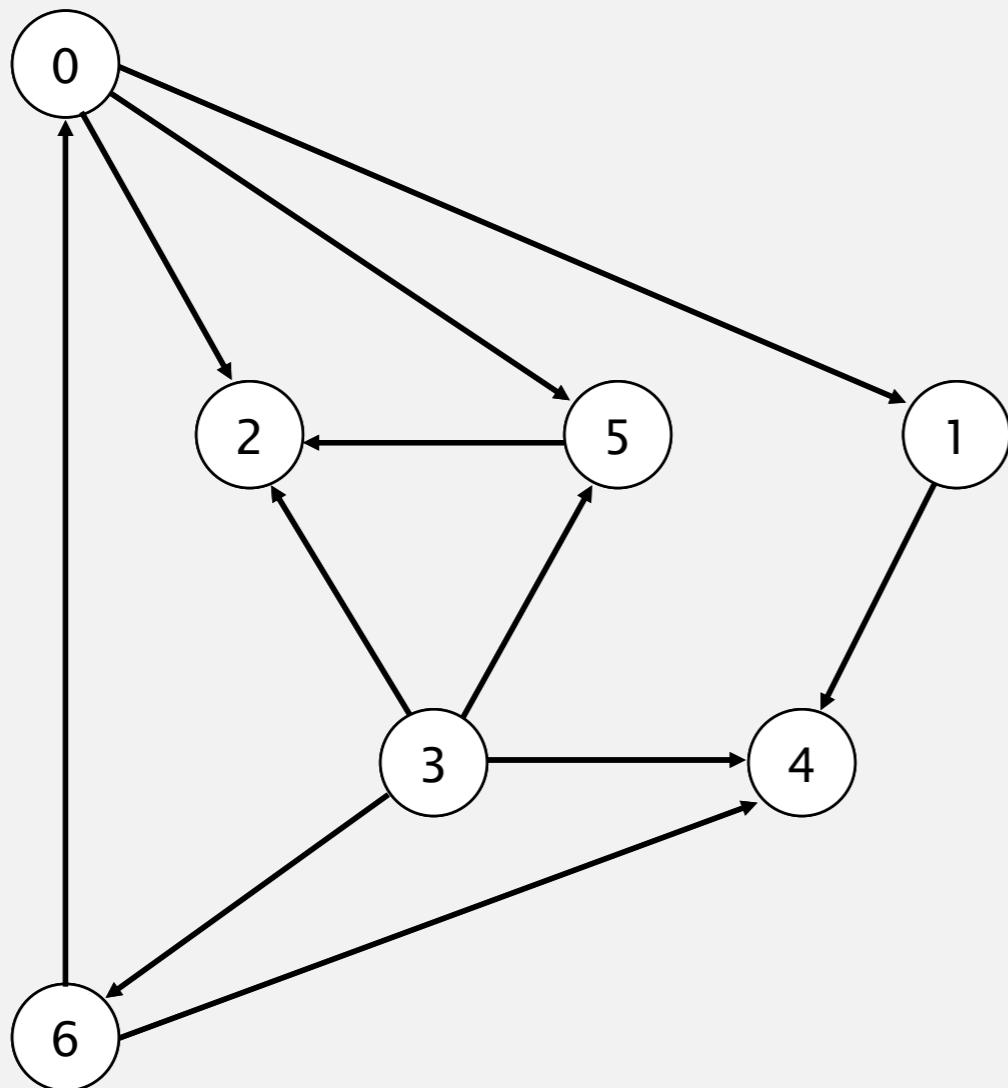
7  
11  
0 5  
0 2  
0 1  
1 4  
3 6  
3 5  
3 4  
5 2  
6 4  
6 0  
3 2

a directed acyclic graph

# Topological sort demo

LO 9B.15

- Run depth-first search.
- Return vertices in reverse postorder.



**postorder**

4 1 2 5 0 6 3

**topological order**

3 6 0 5 2 1 4

**done**

# Depth-first search order

LO 9B.9

```
public class DepthFirstOrder
{
    private boolean[] marked;
    private Stack<Integer> reversePostorder;

    public DepthFirstOrder(Digraph G)
    {
        reversePostorder = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
        reversePostorder.push(v); ←
    }

    public Iterable<Integer> reversePostorder()
    { return reversePostorder; }
}
```

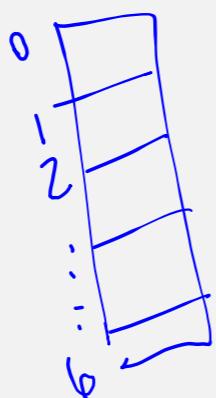
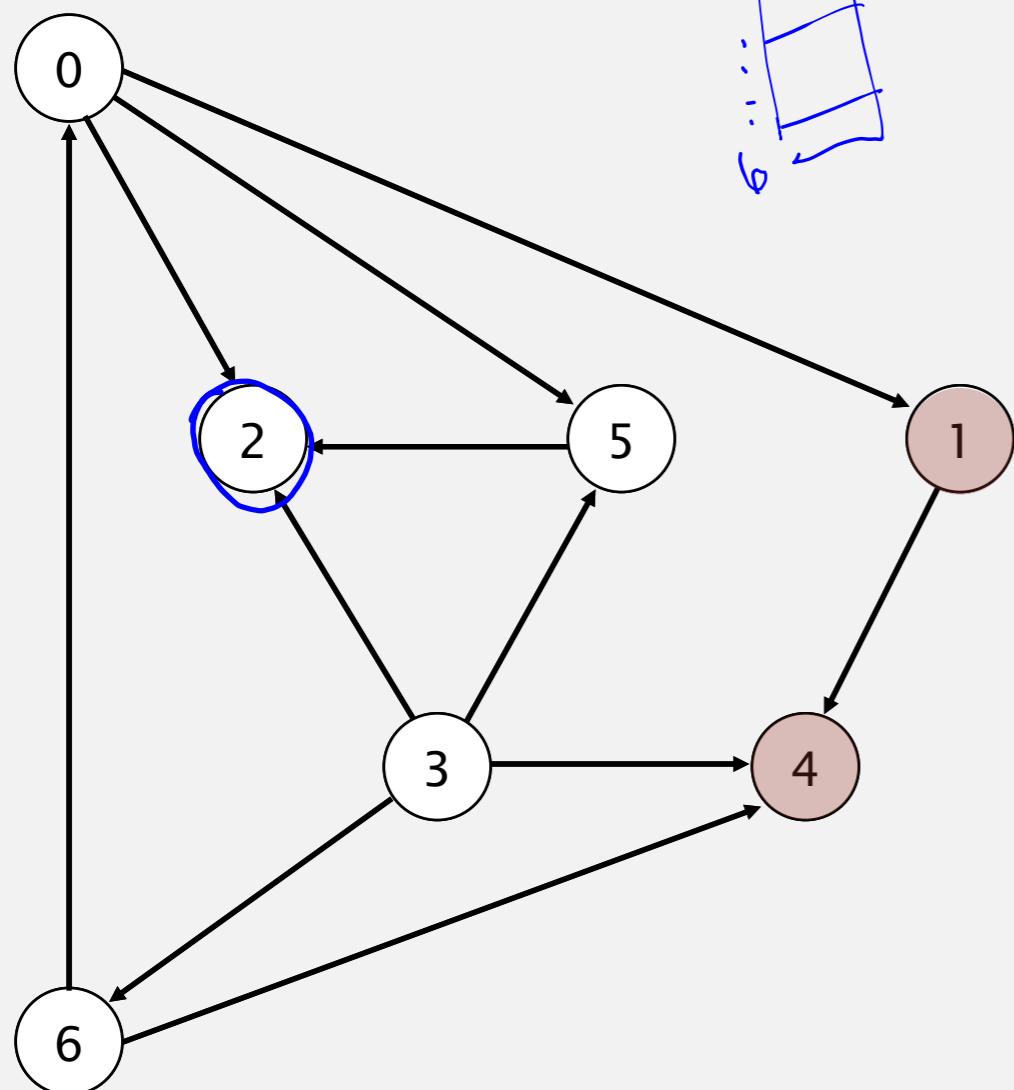
returns all vertices in  
“reverse DFS postorder”

# Topological sort in a DAG: intuition

## Why does topological sort algorithm work?

- First vertex in postorder has outdegree 0.
- Second-to-last vertex in postorder can only point to last vertex.
- ...

Source



For each vertex  $v$  in  $G$   
IF  $v$  has not been visited  
 $\text{DFS}(v)$

order of returning from  
**postorder**

4 1 2 5 0 6 3

**topological order**

3 6 0 5 2 1 4

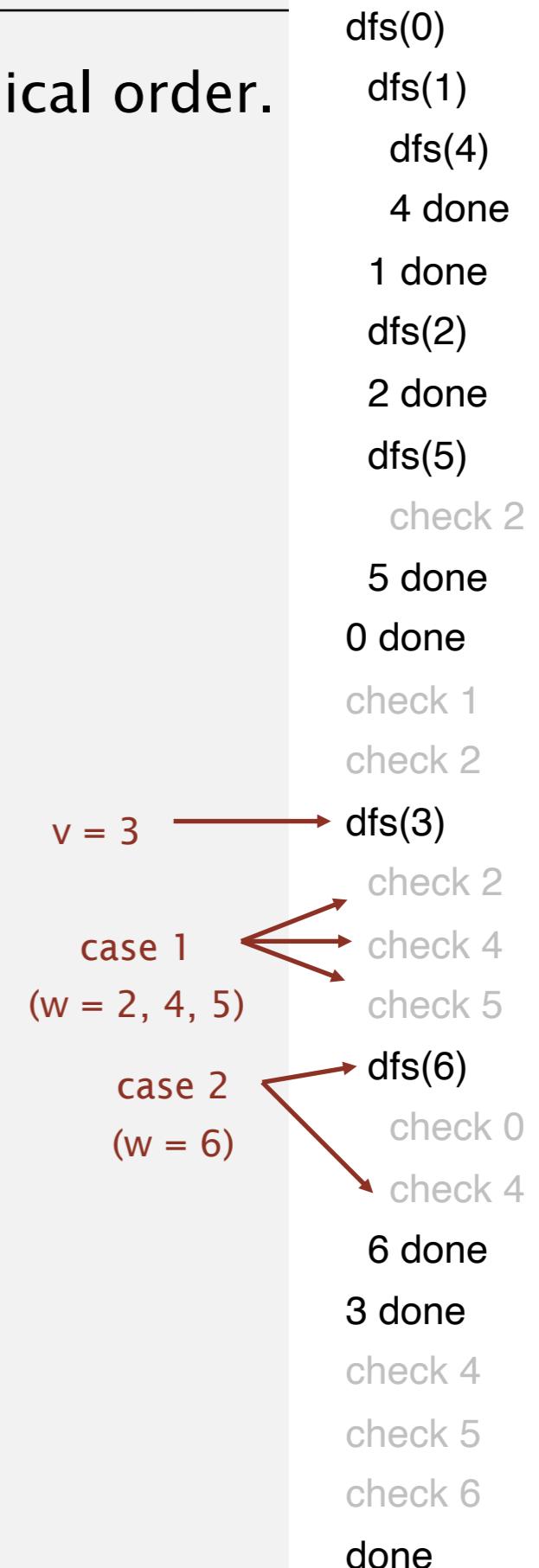
—dfs(0)(5)  
dfs(1)(2)  
dfs(4)(1)  
4 done  
1 done  
dfs(2)(3)  
2 done  
dfs(5)(4)  
check 2  
5 done  
0 done  
check 1  
check 2  
dfs(3)(7)  
check 2  
check 4  
check 5  
dfs(6)(6)  
check 0  
check 4  
6 done  
3 done  
check 4  
check 5  
check 6  
done

# Topological sort in a DAG: correctness proof

**Proposition.** Reverse DFS postorder of a DAG is a topological order.

Pf. Consider any edge  $v \rightarrow w$ . When  $\text{dfs}(v)$  is called:

- Case 1:  $\text{dfs}(w)$  has already been called and returned.
  - thus,  $w$  appears before  $v$  in postorder
- Case 2:  $\text{dfs}(w)$  has not yet been called.
  - $\text{dfs}(w)$  will get called directly or indirectly by  $\text{dfs}(v)$
  - so,  $\text{dfs}(w)$  will return before  $\text{dfs}(v)$
  - thus,  $w$  appears before  $v$  in postorder
- Case 3:  $\text{dfs}(w)$  has already been called, but has not yet returned.
  - function-call stack contains path from  $w$  to  $v$
  - edge  $v \rightarrow w$  would complete a directed cycle
  - contradiction (it's a DAG)

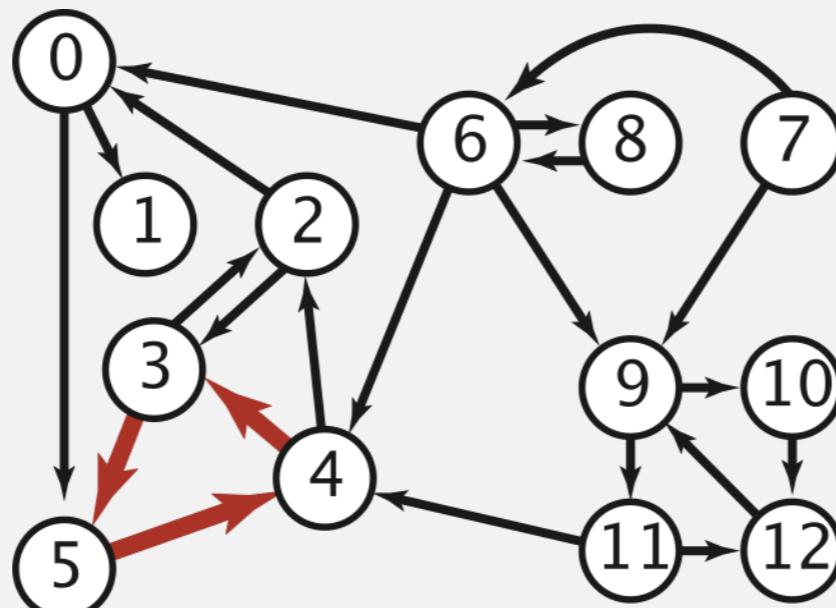


# Directed cycle detection

Proposition. A digraph has a topological order iff no directed cycle.

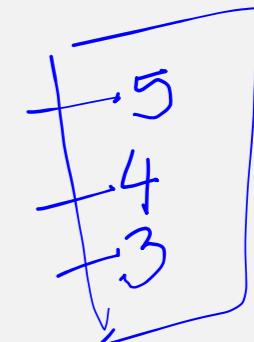
Pf.

- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.



Directed Graph  
is there a cycle?

DFS



Goal. Given a digraph, find a directed cycle.

Solution. DFS. What else? See textbook.

# Directed cycle detection application: precedence scheduling

**Scheduling.** Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

PAGE 3

DEPARTMENT	COURSE	DESCRIPTION	PREREQS
COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432

<http://xkcd.com/754>

**Remark.** A directed cycle implies scheduling problem is infeasible.

# Directed cycle detection application: cyclic inheritance

---

The Java compiler does cycle detection.

```
public class A extends B  
{  
    ...  
}
```

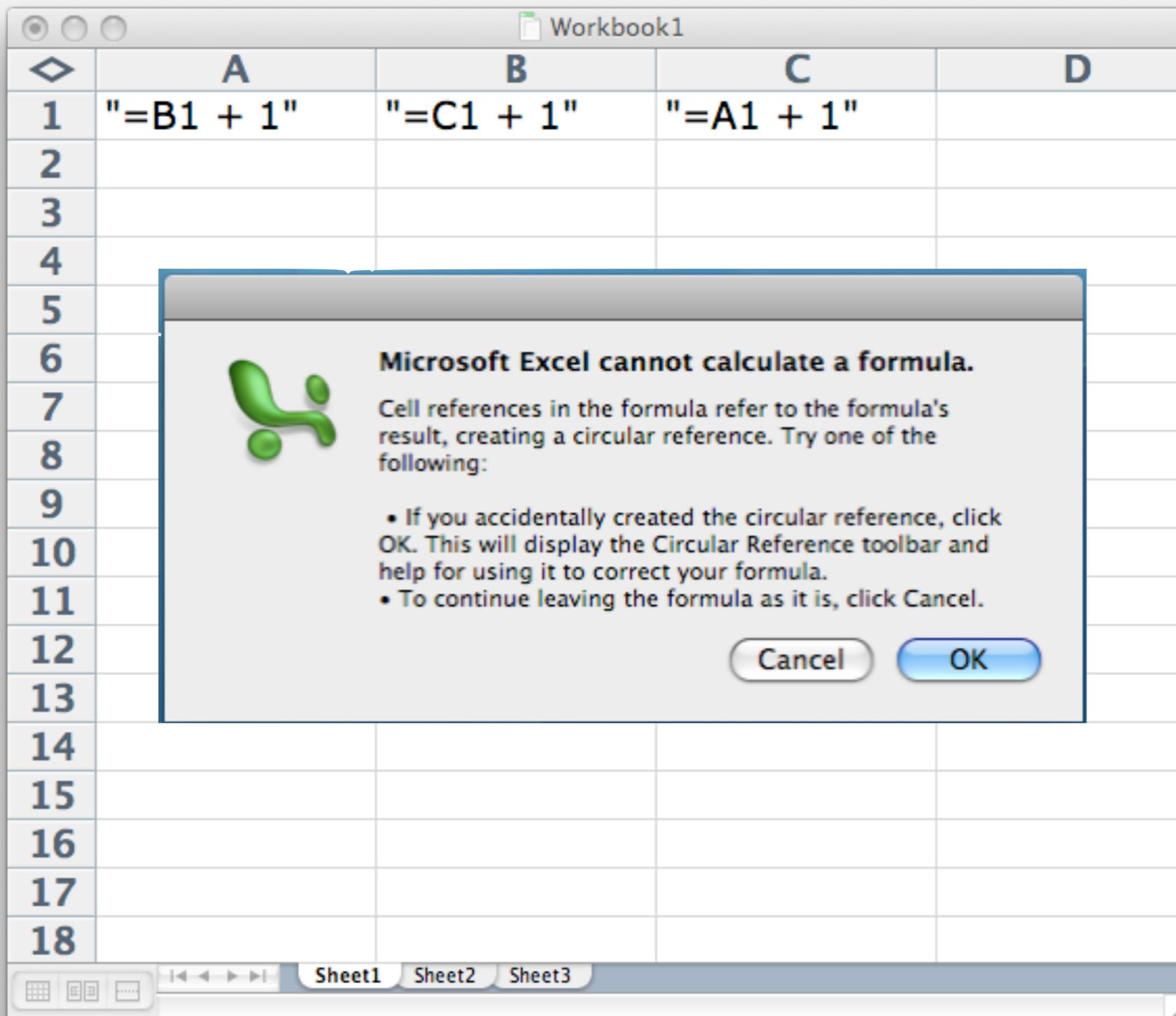
```
public class B extends C  
{  
    ...  
}
```

```
public class C extends A  
{  
    ...  
}
```

```
% javac A.java  
A.java:1: cyclic inheritance involving A  
public class A extends B { }  
          ^  
1 error
```

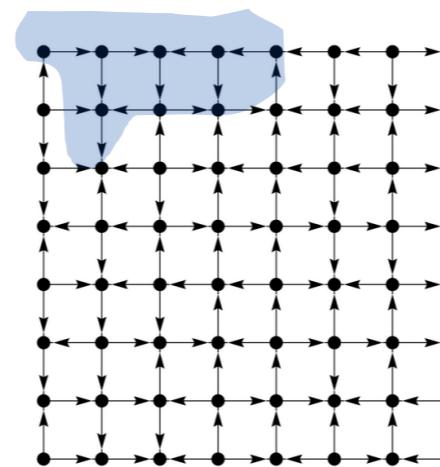
# Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection.



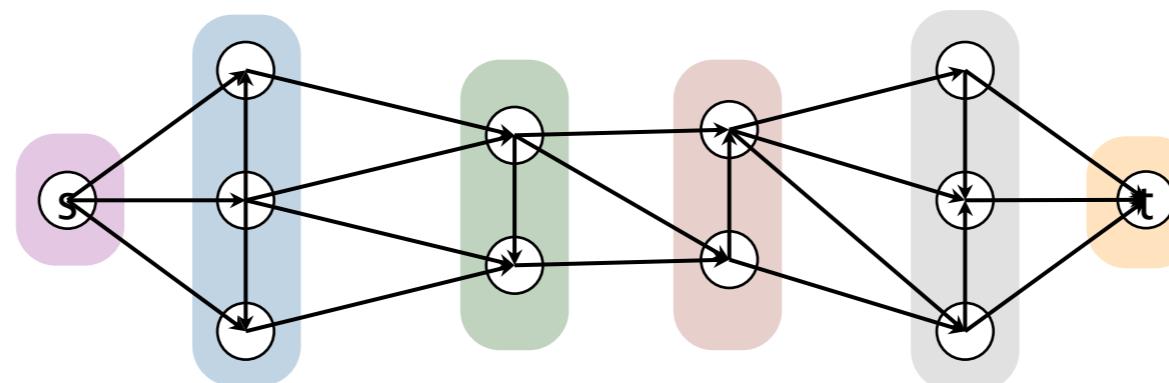
# Digraph-processing summary: algorithms of the day

**single-source  
reachability  
in a digraph**



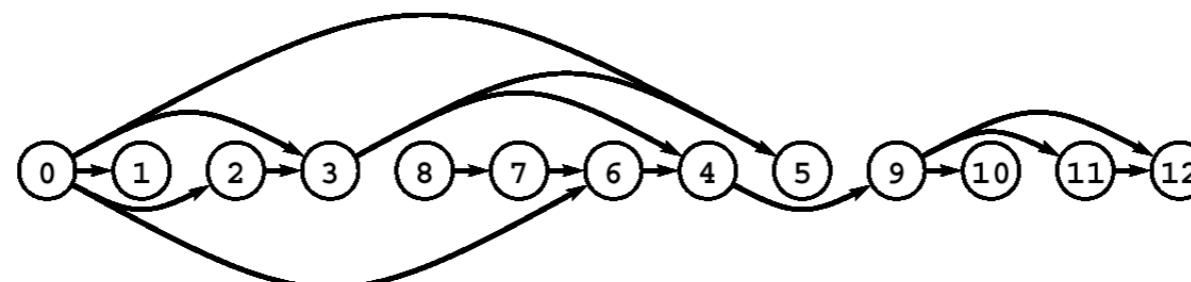
DFS/BFS

**shortest path  
in a digraph**



BFS

**topological sort  
in a DAG**



DFS

# DIRECTED GRAPHS

---

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *depth-first search*
- ▶ *breadth-first search*
- ▶ *topological sort*
- ▶ *strong components* ← see videos



<http://ds.cs.Rutgers.edu>

copyrighted content - Do not share

Some slides Adopted and modified from Sedgewick and Wayne

Last updated on 2/11/18 1:29 PM