

# Directed Graphs

- ▶ digraph search
- ▶ transitive closure
- ▶ topological sort
- ▶ strong components

References:

Algorithms in Java, Chapter 19

<http://www.cs.princeton.edu/introalgsds/52directed>

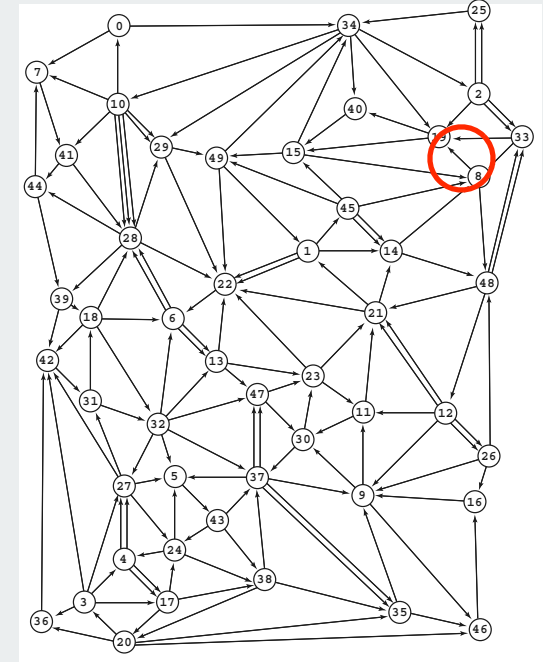
# Directed graphs (digraphs)

Set of objects with **oriented** pairwise connections.

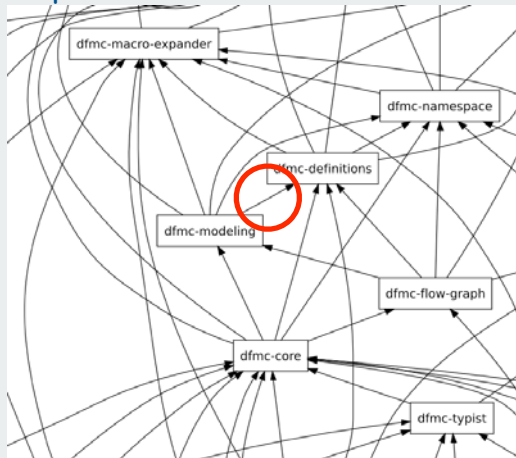
one-way streets in a map



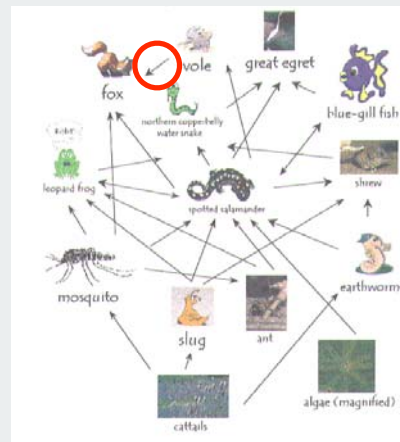
hyperlinks connecting web pages



dependencies in software modules



prey-predator relationships



## Digraph applications

digraph	vertex	edge
financial	stock, currency	transaction
transportation	street intersection, airport	highway, airway route
scheduling	task	precedence constraint
WordNet	synset	hypernym
Web	web page	hyperlink
game	board position	legal move
telephone	person	placed call
food web	species	predator-prey relation
infectious disease	person	infection
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

## Some digraph problems

### Transitive closure.

Is there a directed path from  $v$  to  $w$ ?

### Strong connectivity.

Are all vertices mutually reachable?

### Topological sort.

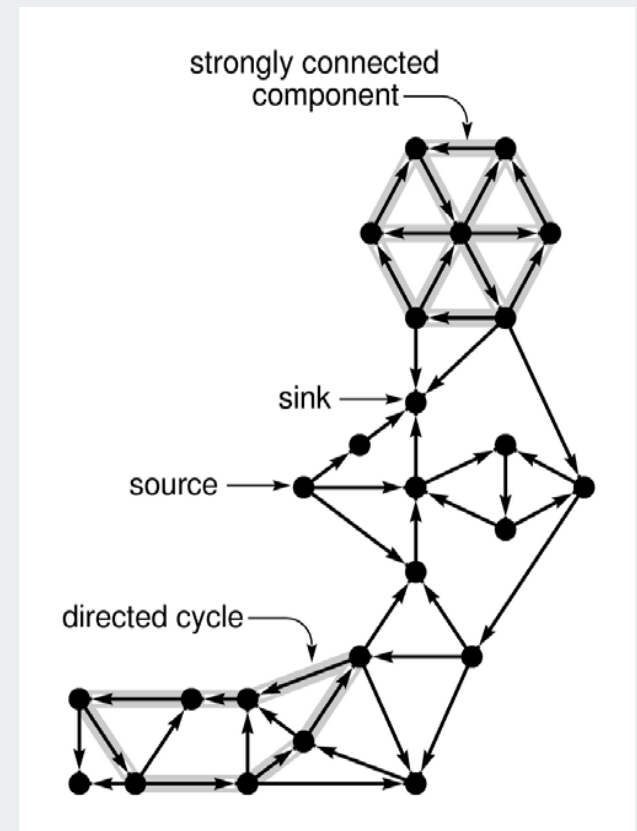
Can you draw the digraph so that all edges point from left to right?

### PERT/CPM.

Given a set of tasks with precedence constraints, how can we best complete them all?

**Shortest path.** Find best route from  $s$  to  $t$  in a weighted digraph

**PageRank.** What is the importance of a web page?



# Digraph representations

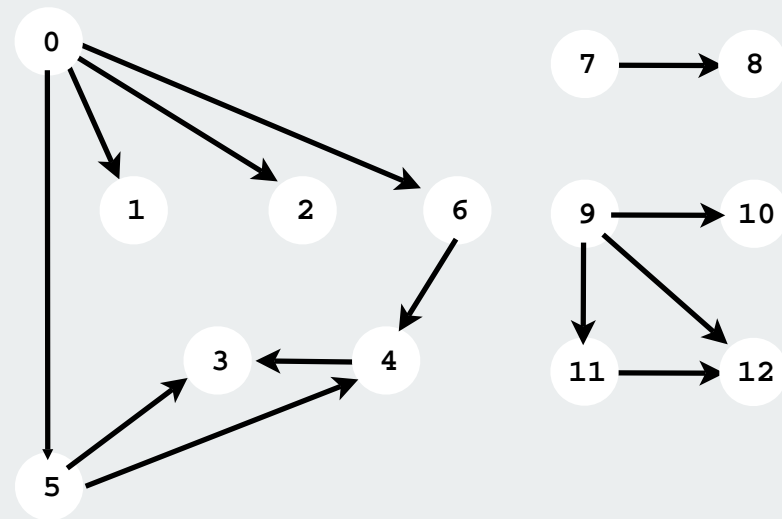
## Vertices

- this lecture: use integers between 0 and  $v-1$ .
- real world: convert between names and integers with symbol table.

## Edges: four easy options

- list of vertex pairs
- vertex-indexed adjacency arrays (adjacency matrix)
- vertex-indexed adjacency lists
- vertex-indexed adjacency SETs

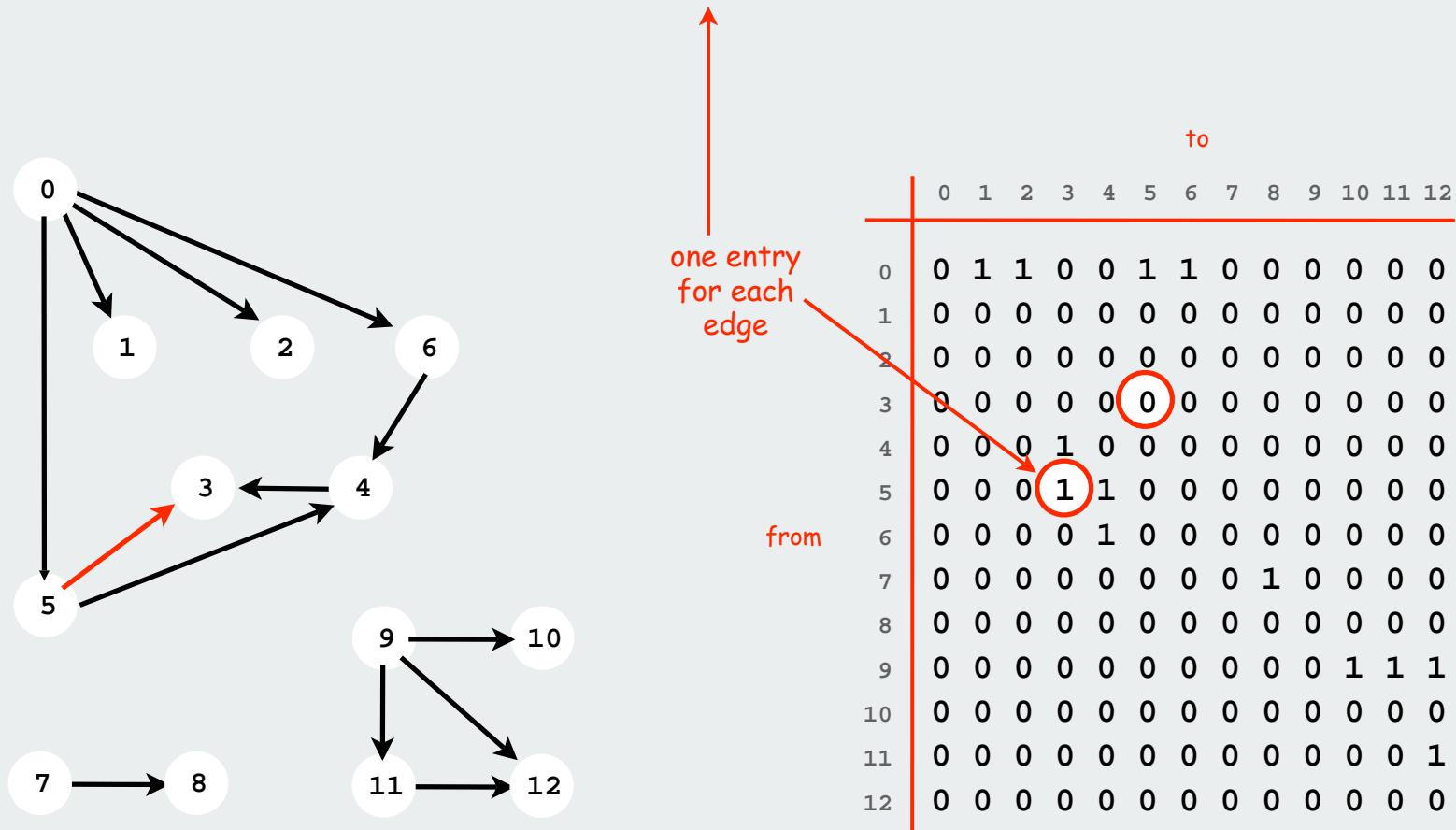
Same as undirected graph  
BUT  
orientation of edges is significant.



## Adjacency matrix digraph representation

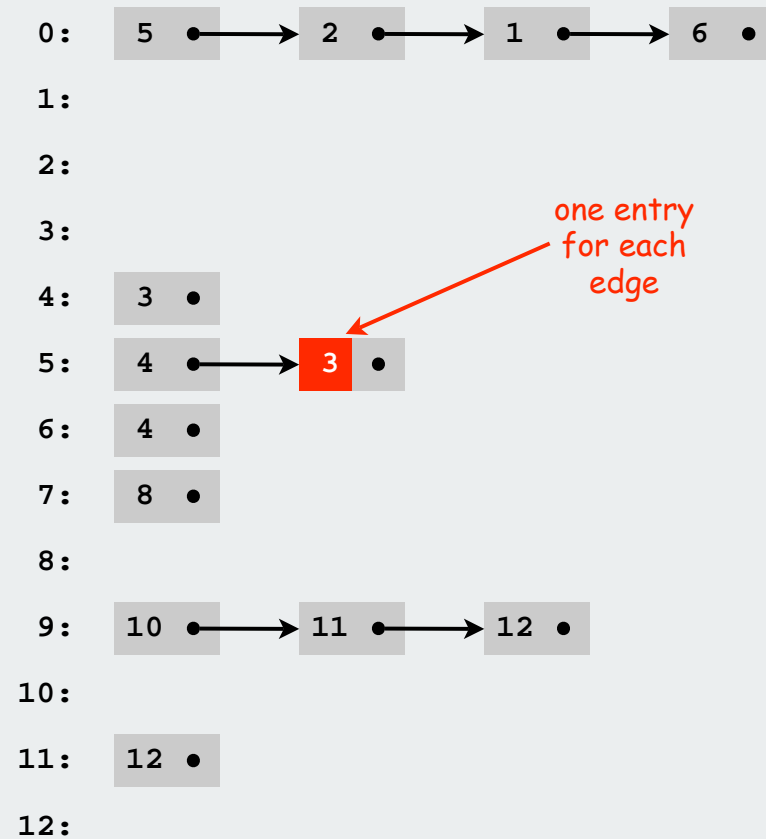
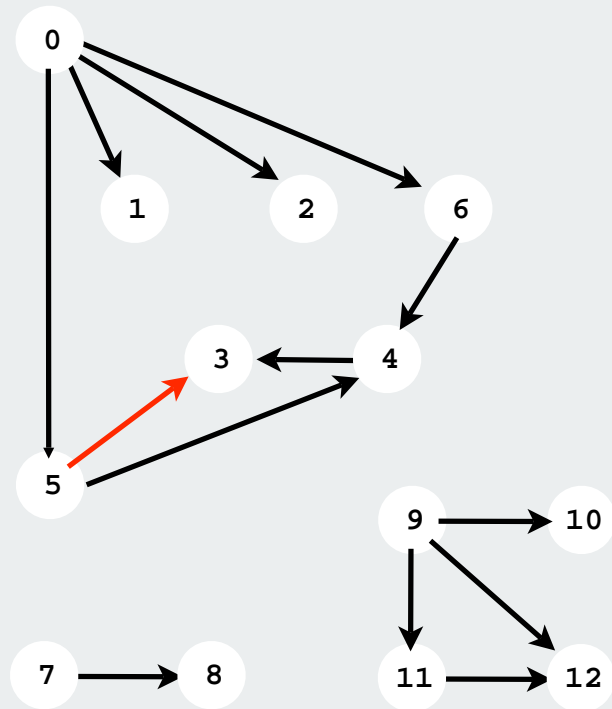
Maintain a two-dimensional  $v \times v$  boolean array.

For each edge  $v \rightarrow w$  in graph:  $\text{adj}[v][w] = \text{true}$ .



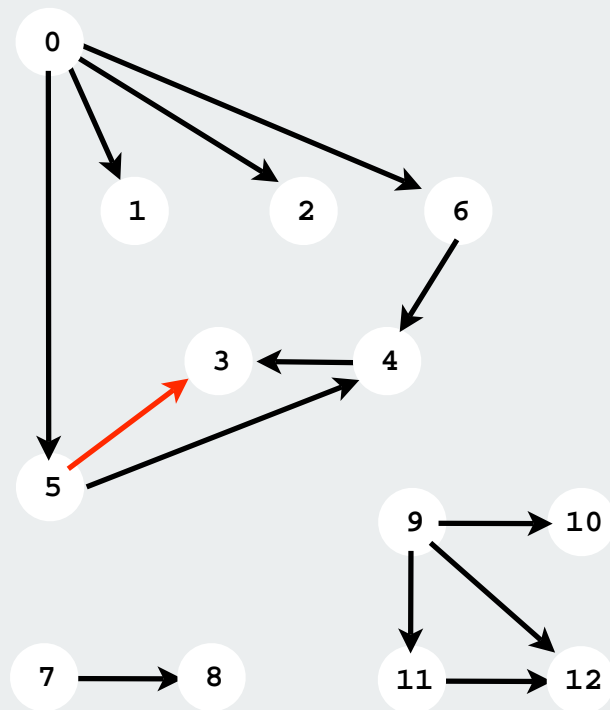
## Adjacency-list digraph representation

Maintain vertex-indexed array of lists.



## Adjacency-SET digraph representation

Maintain vertex-indexed array of SETs.



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

one entry  
for each  
edge



## Adjacency-SET digraph representation: Java implementation

Same as Graph, but only insert **one copy** of each edge.

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

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

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

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

← adjacency SETs

← create empty V-vertex graph

← add edge from v to w  
(Graph also has adj[w].add[v])

← iterable SET for v's neighbors

## Digraph representations

Digraphs are abstract mathematical objects, BUT

- ADT implementation requires specific representation.
- Efficiency depends on matching algorithms to representations.

representation	space	edge between $v$ and $w$ ?	iterate over edges incident to $v$ ?
list of edges	$E$	$E$	$E$
adjacency matrix	$V^2$	1	$V$
adjacency list	$E + V$	$\text{degree}(v)$	$\text{degree}(v)$
adjacency SET	$E + V$	$\log(\text{degree}(v))$	$\text{degree}(v)$

**In practice: Use adjacency SET representation**

- Take advantage of proven technology
- Real-world digraphs tend to be "sparse"  
[ huge number of vertices, small average vertex degree]
- Algs all based on iterating over edges incident to  $v$ .

## Typical digraph application: Google's PageRank algorithm

**Goal.** Determine which web pages on Internet are important.

**Solution.** Ignore keywords and content, focus on hyperlink structure.

**Random surfer model.**

- Start at random page.
- With probability 0.85, randomly select a **hyperlink** to visit next; with probability 0.15, randomly select **any** page.
- PageRank = proportion of time random surfer spends on each page.

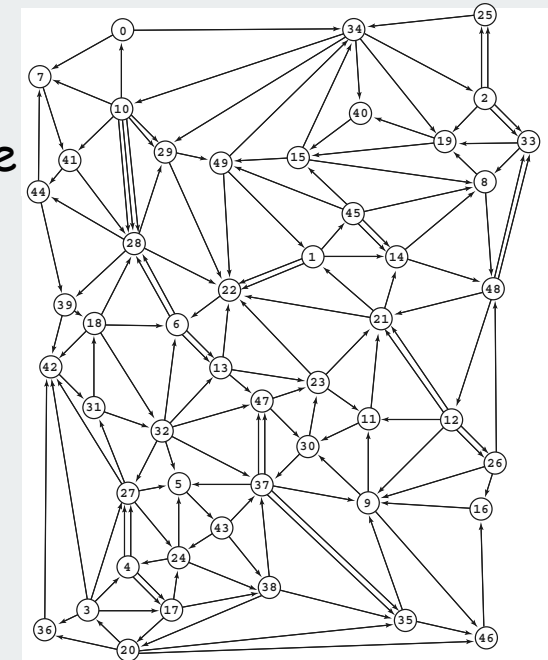
**Solution 1:** Simulate random surfer for a long time.

**Solution 2:** Compute ranks directly until they converge

**Solution 3:** Compute eigenvalues of adjacency matrix!

None feasible without sparse digraph representation

Every **square matrix** is a weighted digraph



## ▶ digraph search

- ▶ transitive closure
- ▶ topological sort
- ▶ strong components

## Digraph application: program control-flow analysis

Every **program** is a digraph (instructions connected to possible successors)

Dead code elimination.

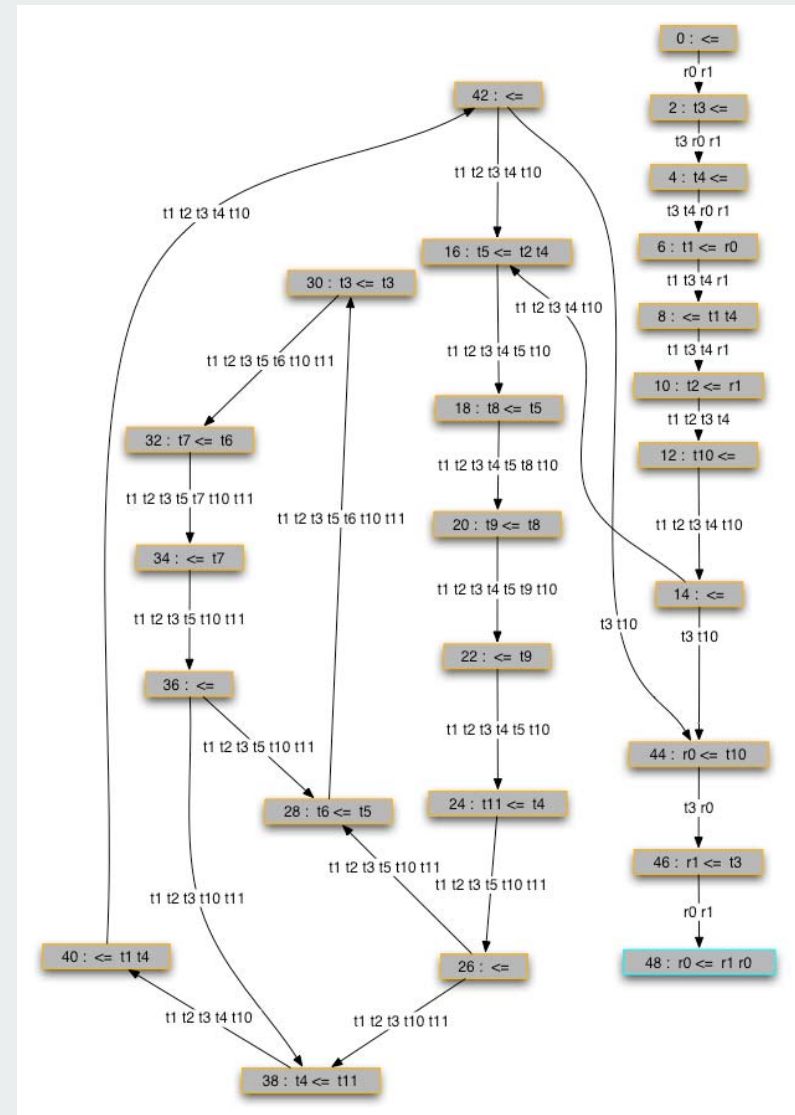
Find (and remove) **unreachable** code

can arise from compiler optimization (or bad code)

Infinite loop detection.

Determine whether exit is **unreachable**

can't detect all possible infinite loops (halting problem)



## Digraph application: mark-sweep garbage collector

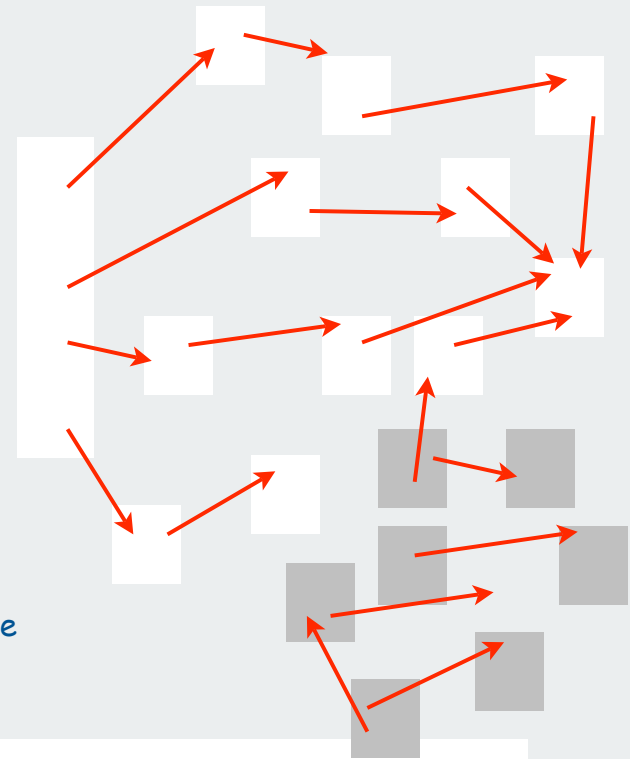
Every **data structure** is a digraph (objects connected by references)

**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).

easy to identify pointers in type-safe language



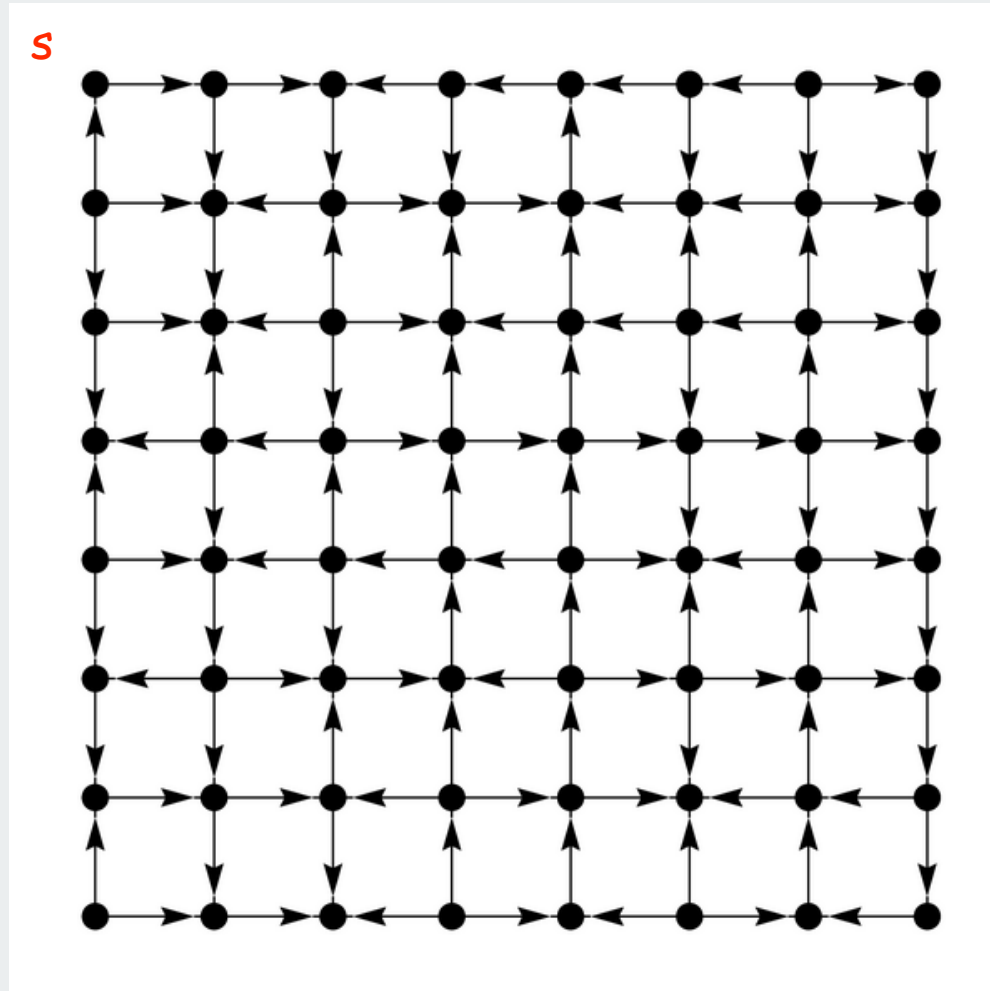
**Mark-sweep algorithm.** [McCarthy, 1960]

- 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.

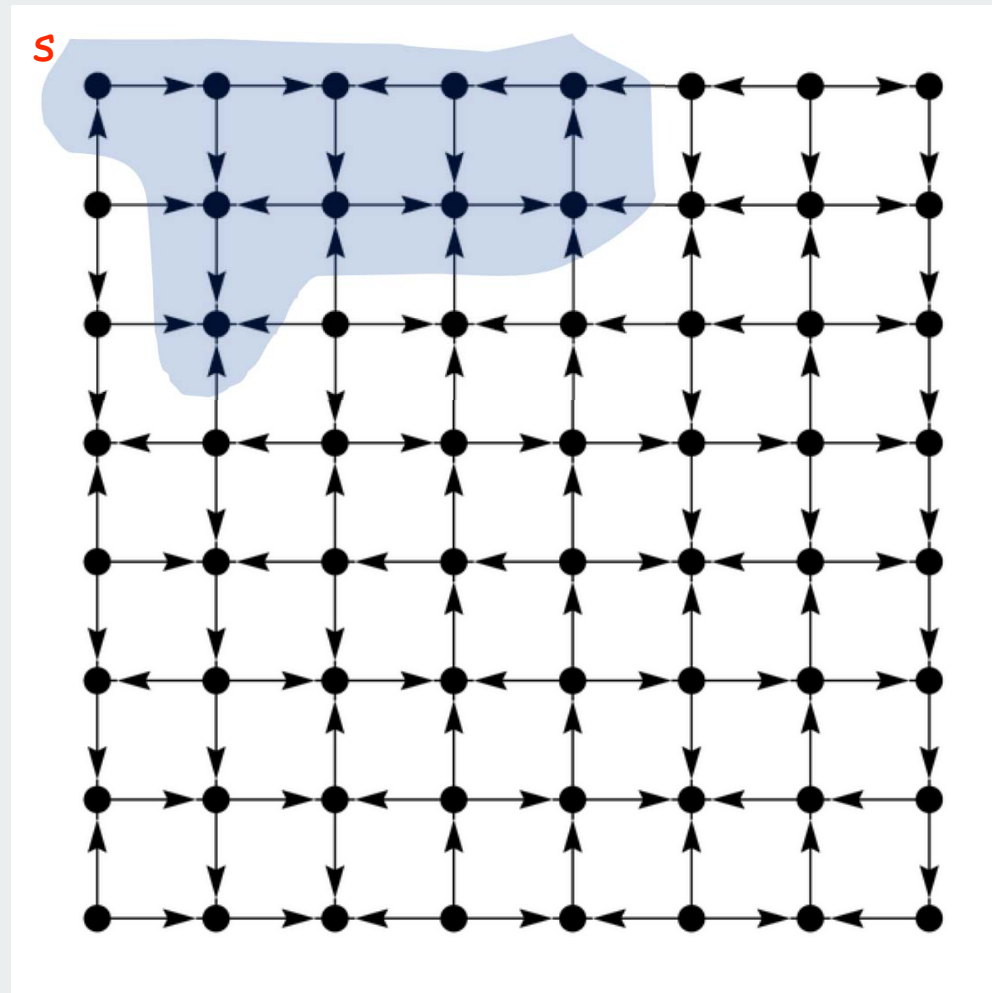
## Reachability

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



## Reachability

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



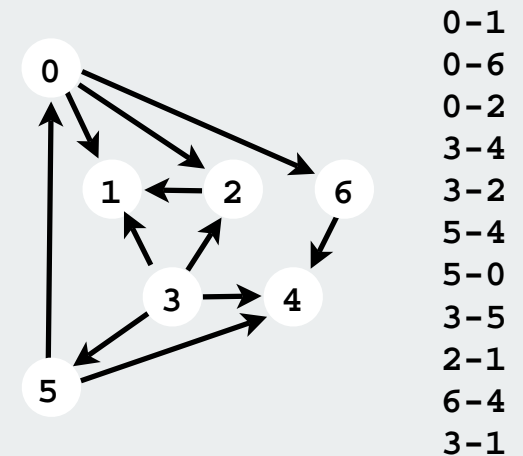


## Digraph-processing challenge 1:

**Problem:** Mark all vertices reachable from a given vertex.

How difficult?

- 1) any COS126 student could do it
- 2) need to be a typical diligent COS226 student
- 3) hire an expert
- 4) intractable
- 5) no one knows



## Depth-first search in digraphs

Same method as for undirected graphs

Every undirected graph *is* a digraph

- happens to have edges in both directions
- DFS is a *digraph* algorithm


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

---

Mark  $v$  as visited.

*Visit* all unmarked vertices  $w$  adjacent to  $v$ .

---

  
recursive

## Depth-first search (single-source reachability)

Identical to undirected version (substitute `Digraph` for `Graph`).

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

    public DFSearcher(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 isReachable(int v)
    {
        return marked[v];
    }
}
```

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is connected to s

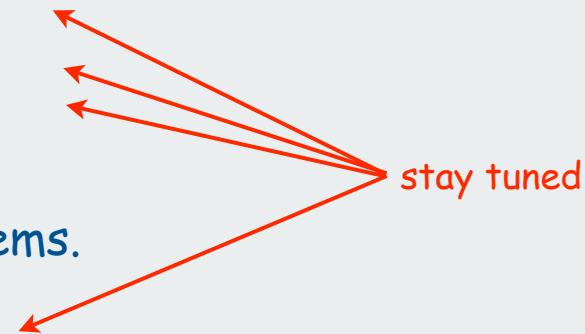
## Depth-first search (DFS)

DFS enables direct solution of simple digraph problems.

- ✓ • Reachability.
- Cycle detection
- Topological sort
- Transitive closure.
- Is there a path from  $s$  to  $t$  ?

Basis for solving difficult digraph problems.

- Directed Euler path.
- Strong connected components.



## Breadth-first search in digraphs

Same method as for undirected graphs

Every undirected graph *is* a digraph

- happens to have edges in both directions
- BFS is a *digraph* algorithm

BFS (from source vertex  $s$ )

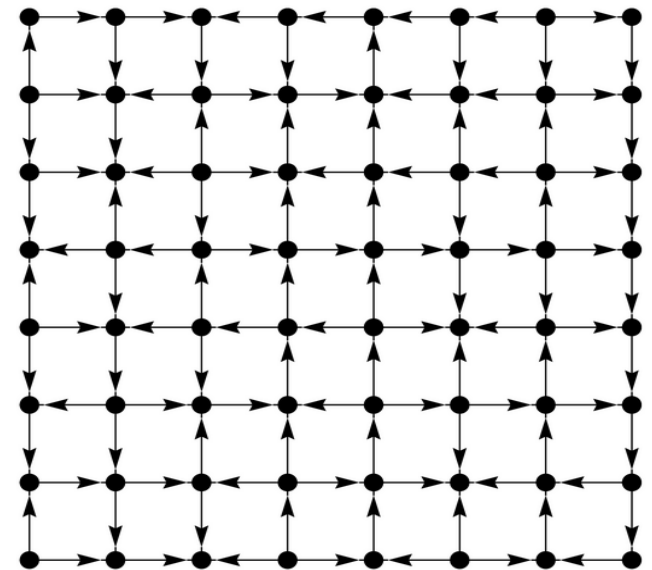
---

Put  $s$  onto a FIFO queue.

Repeat until the queue is empty:

- remove the least recently added vertex  $v$
  - add each of  $v$ 's unvisited neighbors to the queue and mark them as visited.
- 

Visits vertices in *increasing* distance from  $s$



## Digraph BFS application: Web Crawler

The **internet** is a digraph

**Goal.** Crawl Internet, starting from some root website.

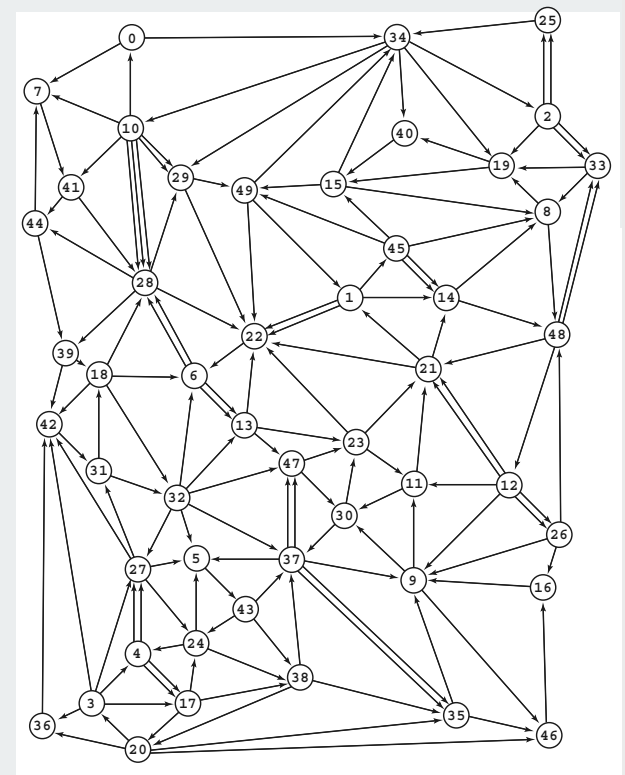
**Solution.** BFS with implicit graph.

**BFS.**

- Start at some root website  
( say <http://www.princeton.edu> ).
- Maintain a **Queue** of websites to explore.
- Maintain a **SET** of discovered websites.
- Dequeue the next website  
and enqueue websites to which it links  
(provided you haven't done so before).

**Q.** Why not use DFS?

**A.** Internet is not fixed (some pages generate new ones when visited)



← subtle point: think about it!

## Web crawler: BFS-based Java implementation

```
Queue<String> q = new Queue<String>();  
SET<String> visited = new SET<String>();
```

← queue of sites to crawl

← set of visited sites

```
String s = "http://www.princeton.edu";  
q.enqueue(s);  
visited.add(s);
```

← start crawling from s

```
while (!q.isEmpty())  
{
```

```
    String v = q.dequeue();  
    System.out.println(v);  
    In in = new In(v);  
    String input = in.readAll();
```

← read in raw html for next site in queue

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

← http://xxx.yyy.zzz

← use regular expression  
to find all URLs in site

```
    {  
        String w = matcher.group();  
        if (!visited.contains(w))  
        {
```

```
            visited.add(w);  
            q.enqueue(w);
```

← if unvisited, mark as visited  
and put on queue

```
        }  
    }  
}
```

- ▶ digraph search
- ▶ **transitive closure**
- ▶ topological sort
- ▶ strong components



## Graph-processing challenge (revisited)

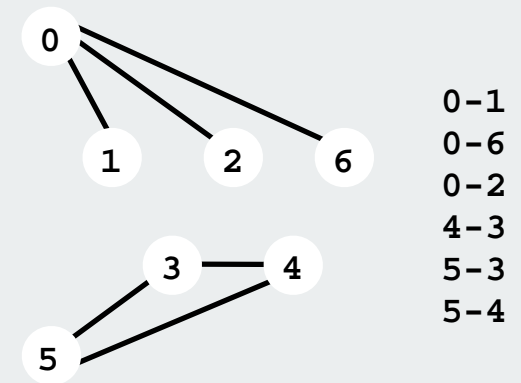
**Problem:** Is there a path from  $s$  to  $t$  ?

**Goals:** linear  $\sim(V + E)$  preprocessing time

constant query time

**How difficult?**

- 1) any COS126 student could do it
- 2) need to be a typical diligent COS226 student
- 3) hire an expert
- 4) intractable
- 5) no one knows
- 6) impossible



## Digraph-processing challenge 2

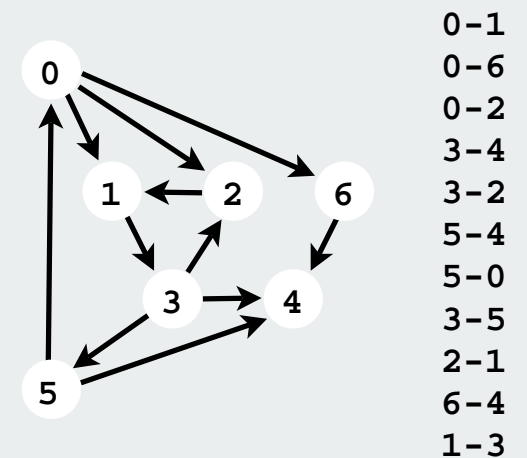
**Problem:** Is there a **directed** path from  $s$  to  $t$ ?

**Goals:** **linear**  $\sim(V + E)$  preprocessing time

**constant** query time

**How difficult?**

- 1) any COS126 student could do it
- 2) need to be a typical diligent COS226 student
- 3) hire an expert
- 4) intractable
- 5) no one knows
- 6) impossible

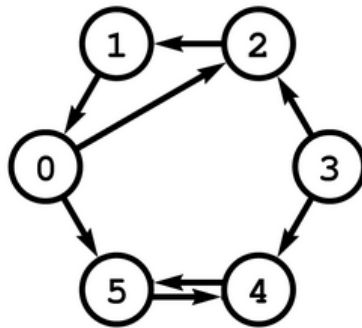


## Transitive Closure

The **transitive closure** of  $G$  has an directed **edge** from  $v$  to  $w$  if there is a directed **path** from  $v$  to  $w$  in  $G$

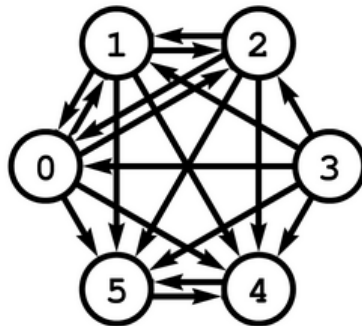
graph is usually sparse

$G$



	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1

Transitive closure  
of  $G$



	0	1	2	3	4	5
0	1	1	1	0	1	1
1	1	1	1	0	1	1
2	1	1	1	0	1	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1

TC is usually dense  
so adjacency matrix  
representation is OK

## Digraph-processing challenge 2 (revised)

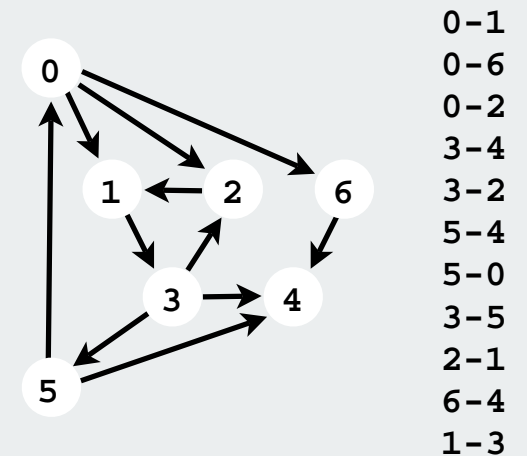
**Problem:** Is there a **directed** path from  $s$  to  $t$  ?

**Goals:**  $\sim V^2$  preprocessing time

**constant** query time

**How difficult?**

- 1) any COS126 student could do it
- 2) need to be a typical diligent COS226 student
- 3) hire an expert
- 4) intractable
- 5) no one knows
- 6) impossible



## Digraph-processing challenge 2 (revised again)

**Problem:** Is there a **directed** path from  $s$  to  $t$ ?

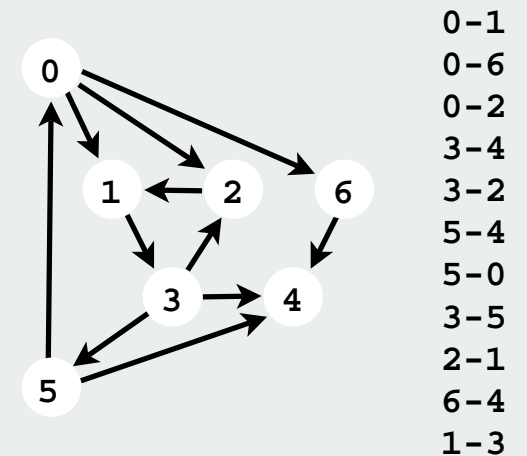
**Goals:**  $\sim VE$  preprocessing time ( $\sim V^3$  for dense digraphs)

$\sim V^2$  space

constant query time

How difficult?

- 1) any COS126 student could do it
- 2) need to be a typical diligent COS226 student
- 3) hire an expert
- 4) intractable
- 5) no one knows
- 6) impossible



## Transitive closure: Java implementation

Use an array of `DFSearcher` objects,  
one for each row of transitive closure

```
public class TransitiveClosure
{
    private DFSearcher[] tc;

    public TransitiveClosure(Digraph G)
    {
        tc = new DFSearcher[G.V()];
        for (int v = 0; v < G.V(); v++)
            tc[v] = new DFSearcher(G, v);
    }

    public boolean reachable(int v, int w)
    {
        return tc[v].isReachable(w);
    }
}
```

```
public class DFSearcher
{
    private boolean[] marked;
    public DFSearcher(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 isReachable(int v)
    {
        return marked[v];
    }
}
```

← is there a directed path from v to w ?

- ▶ digraph search
- ▶ transitive closure
- ▶ **topological sort**
- ▶ strong components

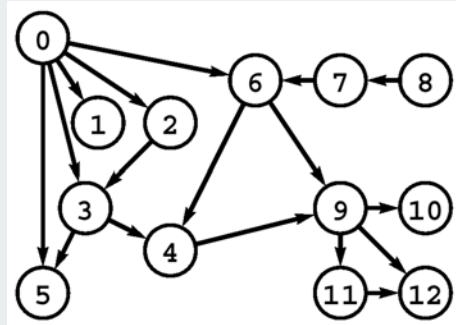
## Digraph application: Scheduling

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


### Graph model.

- Create a vertex  $v$  for each task.
- Create an edge  $v \rightarrow w$  if task  $v$  must precede task  $w$ .
- Schedule tasks in topological order.

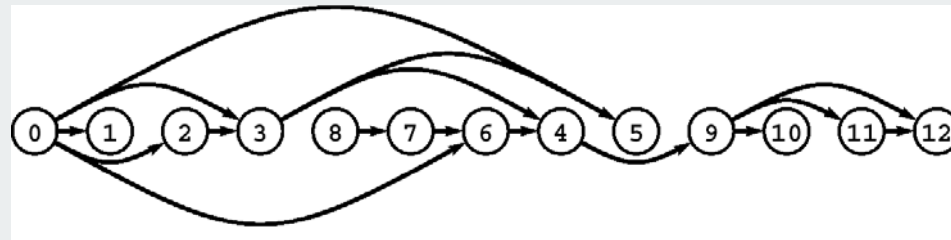
precedence  
constraints



tasks

- 
0. read programming assignment
  1. download files
  2. write code
  3. attend precept
  - ...
  12. sleep

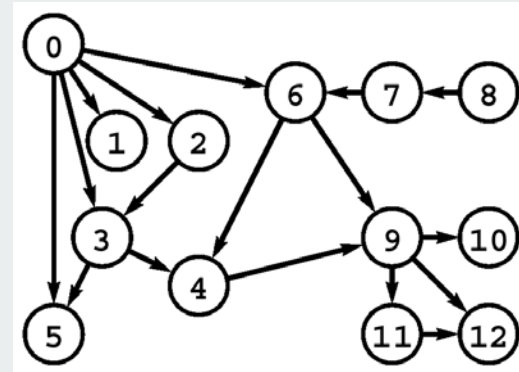
feasible  
schedule



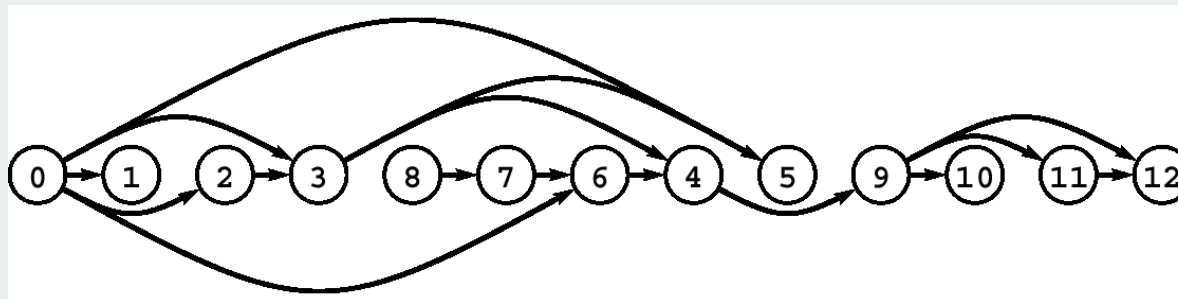


# Topological Sort

DAG. Directed **acyclic** graph.



Topological sort. Redraw DAG so all edges point left to right.



Observation. Not possible if graph has a directed cycle.

## Digraph-processing challenge 3

**Problem:** Check that the digraph is a DAG.

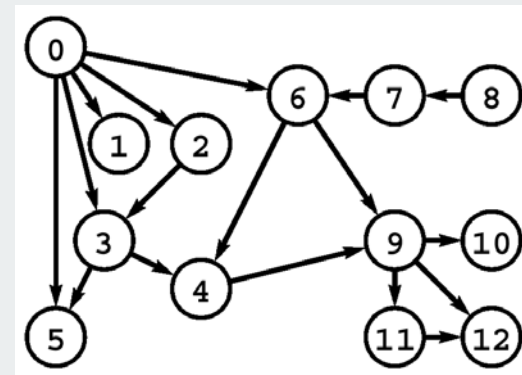
If it is a DAG, do a **topological sort**.

**Goals:** **linear**  $\sim(V + E)$  preprocessing time

provide client with vertex iterator for topological order

**How difficult?**

- 1) any CS126 student could do it
- 2) need to be a typical diligent CS226 student
- 3) hire an expert
- 4) intractable
- 5) no one knows
- 6) impossible



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

## Topological sort in a DAG: Java implementation

```
public class TopologicalSorter
{
    private int count;
    private boolean[] marked;
    private int[] ts;

    public TopologicalSorter(Digraph G)
    {
        marked = new boolean[G.V()];
        ts = new int[G.V()];
        count = G.V();
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) tsort(G, v);
    }

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

standard DFS  
with 5  
extra lines of code

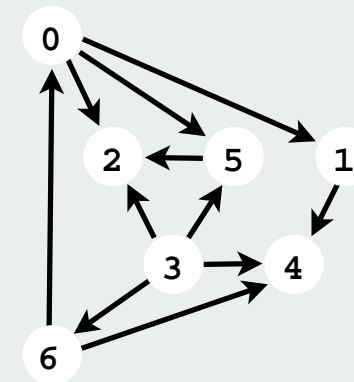
add iterator that returns  
ts[0], ts[1], ts[2]...

Seems easy? Missed by experts for a few decades

## Topological sort of a dag: trace

"visit" means "call `tsort()`" and "leave" means "return from `tsort()`"

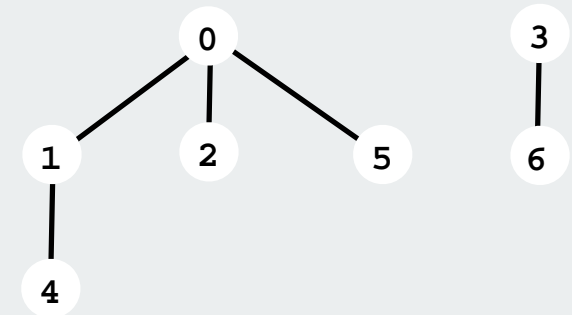
	marked[]	ts[]
visit 0:	1 0 0 0 0 0 0	0 0 0 0 0 0 0
visit 1:	1 1 0 0 0 0 0	0 0 0 0 0 0 0
visit 4:	1 1 0 0 1 0 0	0 0 0 0 0 0 0
leave 4:	1 1 0 0 1 0 0	0 0 0 0 0 0 4
leave 1:	1 1 0 0 1 0 0	0 0 0 0 0 1 4
visit 2:	1 1 1 0 1 0 0	0 0 0 0 0 1 4
leave 2:	1 1 1 0 1 0 0	0 0 0 0 2 1 4
visit 5:	1 1 1 0 1 1 0	0 0 0 0 2 1 4
check 2:	1 1 1 0 1 1 0	0 0 0 0 2 1 4
leave 5:	1 1 1 0 1 1 0	0 0 0 5 2 1 4
leave 0:	1 1 1 0 1 1 0	0 0 0 5 2 1 4
check 1:	1 1 1 0 1 1 0	0 0 0 5 2 1 4
check 2:	1 1 1 0 1 1 0	0 0 0 5 2 1 4
visit 3:	1 1 1 1 1 1 0	0 0 0 5 2 1 4
check 2:	1 1 1 1 1 1 0	0 0 0 5 2 1 4
check 4:	1 1 1 1 1 1 0	0 0 0 5 2 1 4
check 5:	1 1 1 1 1 1 0	0 0 0 5 2 1 4
visit 6:	1 1 1 1 1 1 1	0 0 0 5 2 1 4
leave 6:	1 1 1 1 1 1 1	0 6 0 5 2 1 4
leave 3:	1 1 1 1 1 1 1	3 6 0 5 2 1 4
check 4:	1 1 1 1 1 1 0	3 6 0 5 2 1 4
check 5:	1 1 1 1 1 1 0	3 6 0 5 2 1 4
check 6:	1 1 1 1 1 1 0	3 6 0 5 2 1 4



adj SETs

```

0: 1 2 5
1: 4
2:
3: 2 4 5 6
4:
5: 2
6: 0 4
  
```



## Topological sort in a DAG: correctness proof

Invariant:

`tsort(G, v)` visits all vertices  
reachable from `v` with a directed path

Proof by induction:

- `w` marked: vertices reachable from `w` are already visited
- `w` not marked: call `tsort(G, w)` to visit the vertices reachable from `w`

Therefore, algorithm is correct  
in placing `v` before all vertices visited  
during call to `tsort(G, v)` just before returning.

```
public class TopologicalSorter
{
    private int count;
    private boolean[] marked;
    private int[] ts;

    public TopologicalSorter(Digraph G)
    {
        marked = new boolean[G.V()];
        ts = new int[G.V()];
        count = G.V();
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) tsort(G, v);
    }

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

Q. How to tell whether the digraph has a cycle (is not a DAG)?

A. Use `TopologicalSorter` (exercise)

## Topological sort applications.

- Causalities.
- Compilation units.
- Class inheritance.
- Course prerequisites.
- Deadlocking detection.
- Temporal dependencies.
- Pipeline of computing jobs.
- Check for symbolic link loop.
- Evaluate formula in spreadsheet.
- Program Evaluation and Review Technique / Critical Path Method

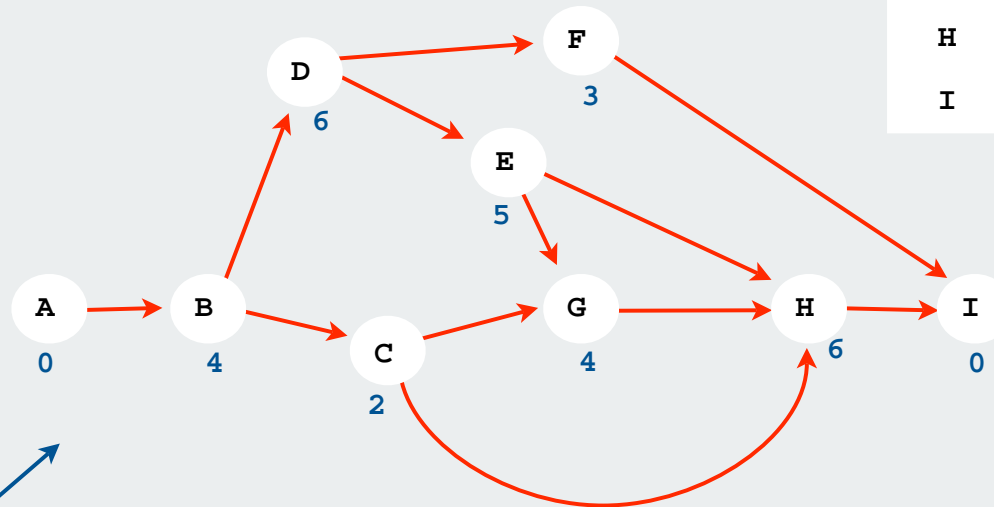
## Topological sort application (weighted DAG)

### Precedence scheduling

- Task  $v$  takes  $\text{time}[v]$  units of time.
- Can work on jobs in parallel.
- Precedence constraints:
- must finish task  $v$  before beginning task  $w$ .
- Goal: finish each task as soon as possible

index	task	time	prereq
A	begin	0	-
B	framing	4	A
C	roofing	2	B
D	siding	6	B
E	windows	5	D
F	plumbing	3	D
G	electricity	4	C, E
H	paint	6	C, E
I	finish	0	F, H

### Example:

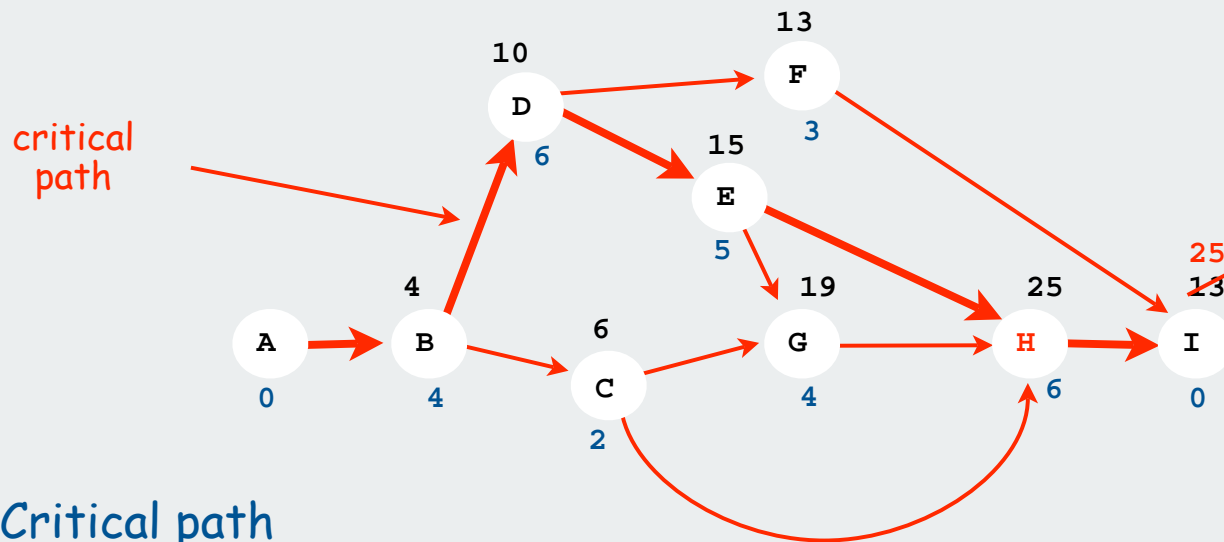


vertices labelled  
A-I in topological order

## Program Evaluation and Review Technique / Critical Path Method

### PERT/CPM algorithm.

- compute topological order of vertices.
- initialize  $\text{fin}[v] = 0$  for all vertices  $v$ .
- consider vertices  $v$  in topologically sorted order.  
for each edge  $v \rightarrow w$ , set  $\text{fin}[w] = \max(\text{fin}[w], \text{fin}[v] + \text{time}[w])$



### Critical path

- remember vertex that set value.
- work backwards from sink

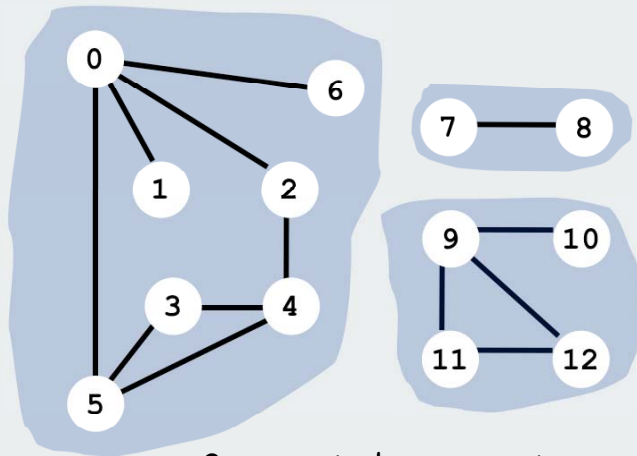


- ▶ digraph search
- ▶ transitive closure
- ▶ topological sort
- ▶ **strong components**

# Strong connectivity in digraphs

## Analog to connectivity in undirected graphs

In a **Graph**,  $u$  and  $v$  are connected when there is a path from  $u$  to  $v$



3 connected components  
(sets of mutually connected vertices)

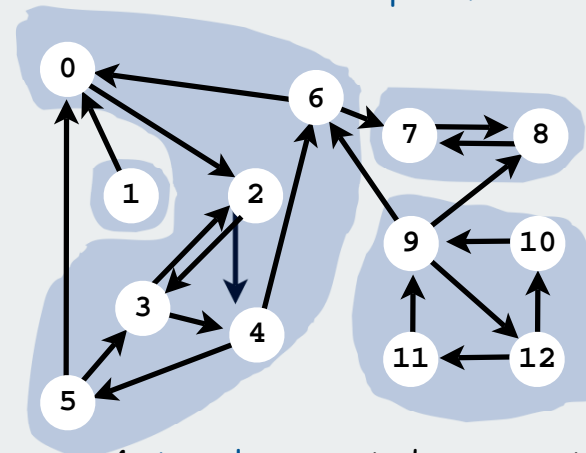
Connectivity table (easy to compute with DFS)

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

```
public int connected(int v, int w)
{ return cc[v] == cc[w]; }
```

constant-time client connectivity query

In a **Digraph**,  $u$  and  $v$  are **strongly connected** when there is a **directed** path from  $u$  to  $v$  and a directed path from  $v$  to  $u$



4 **strongly** connected components  
(sets of mutually **strongly** connected vertices)

Strong connectivity table (how to compute?)

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

```
public int connected(int v, int w)
{ return cc[v] == cc[w]; }
```

constant-time client strong connectivity query

## Digraph-processing challenge 4

**Problem:** Is there a **directed cycle** containing  $s$  and  $t$ ?

**Equivalent:** Are there **directed** paths from  $s$  to  $t$  **and** from  $t$  to  $s$ ?

**Equivalent:** Are  $s$  and  $t$  **strongly connected**?

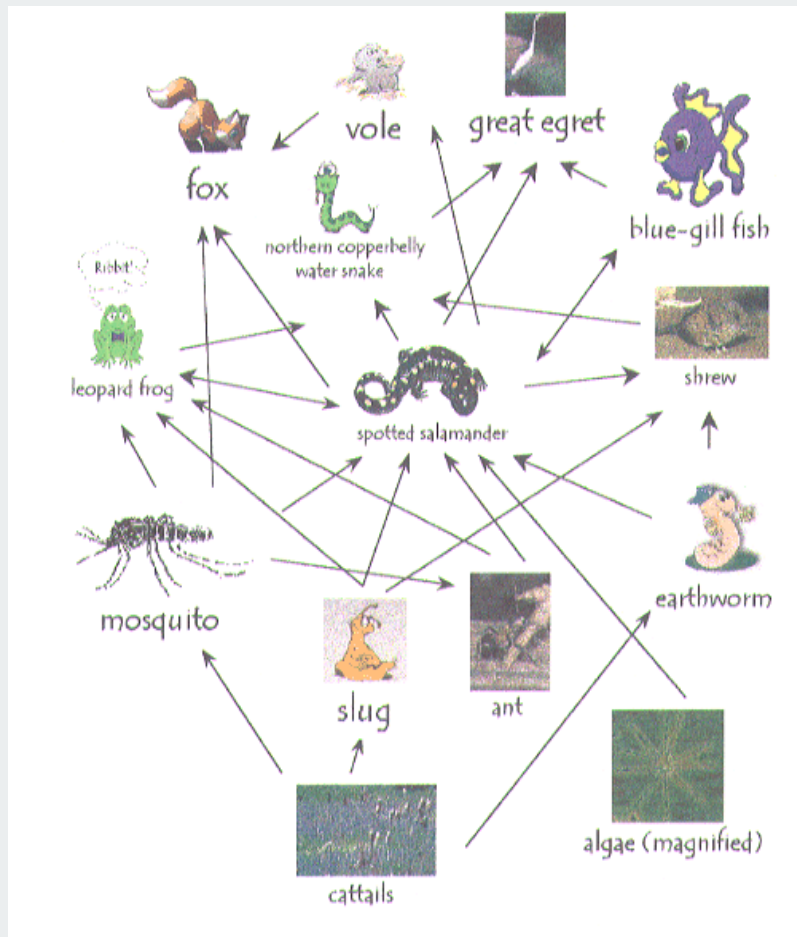
**Goals:** **linear** ( $V + E$ ) preprocessing time (like for undirected graphs)  
**constant** query time

**How difficult?**

- 1) any COS126 student could do it
- 2) need to be a typical diligent COS226 student
- 3) hire an expert
- 4) intractable
- 5) no one knows
- 6) impossible

## Typical strong components applications

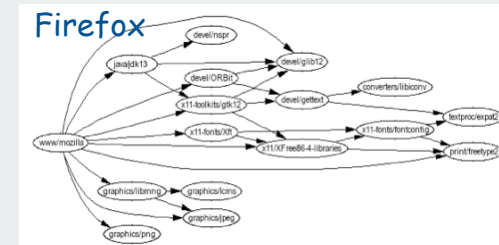
## Ecological food web



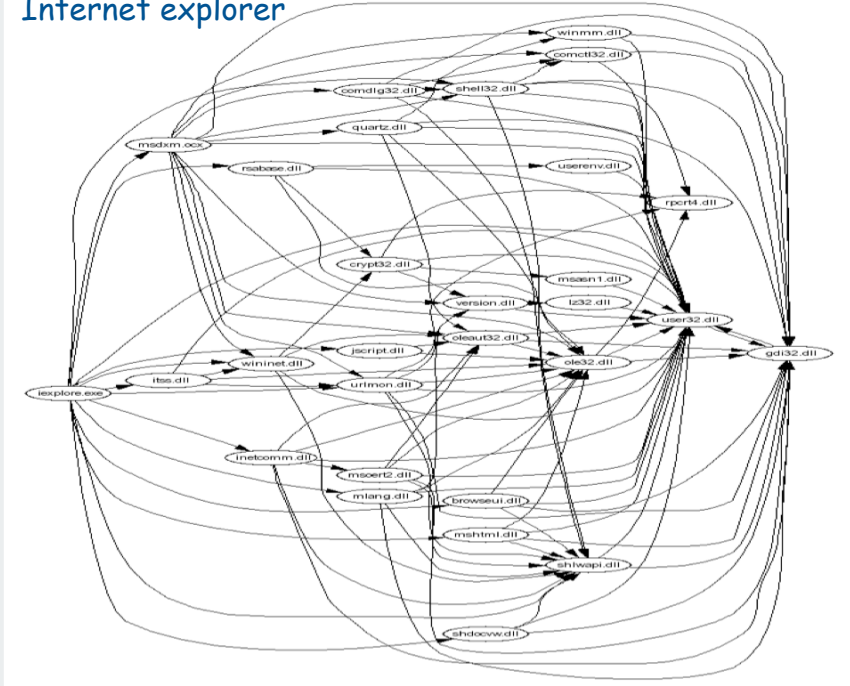
Strong component: subset with common energy flow

- source in kernel DAG: needs outside energy?
- sink in kernel DAG: heading for growth?

## Software module dependency digraphs



## Internet explorer



Strong component: subset of mutually interacting modules

- approach 1: package strong components together
- approach 2: use to improve design!

## Strong components algorithms: brief history

### 1960s: Core OR problem

- widely studied
- some practical algorithms
- complexity not understood

### 1972: Linear-time DFS algorithm (Tarjan)

- classic algorithm
- level of difficulty: CS226++
- demonstrated broad applicability and importance of DFS

### 1980s: Easy two-pass linear-time algorithm (Kosaraju)

- forgot notes for teaching algorithms class
- developed algorithm in order to teach it!
- later found in Russian scientific literature (1972)

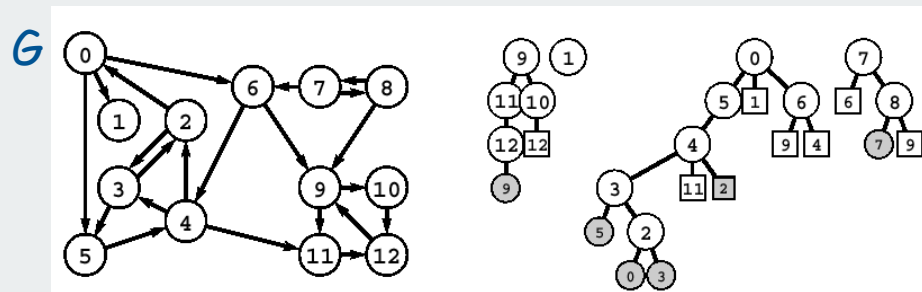
### 1990s: More easy linear-time algorithms (Gabow, Mehlhorn)

- Gabow: fixed old OR algorithm
- Mehlhorn: needed one-pass algorithm for LEDA

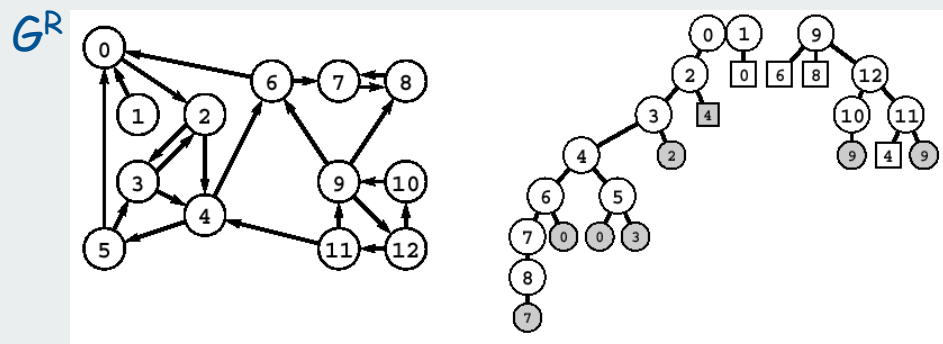
## Kosaraju's algorithm

## Simple (but mysterious) algorithm for computing strong components

- Run DFS on  $G^R$  and compute postorder.
- Run DFS on  $G$ , considering vertices in reverse postorder
- [has to be seen to be believed: follow example in book]



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



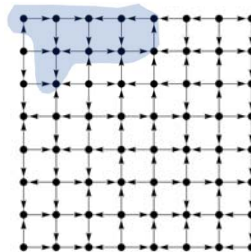
	0	1	2	3	4	5	6	7	8	9	10	11	12
sc	2	1	2	2	2	2	2	3	3	0	0	0	0

**Theorem.** Trees in second DFS are strong components. (!)

Proof. [stay tuned in COS 423]

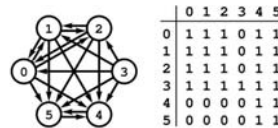
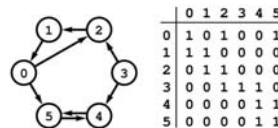
## Digraph-processing summary: Algorithms of the day

Single-source  
reachability



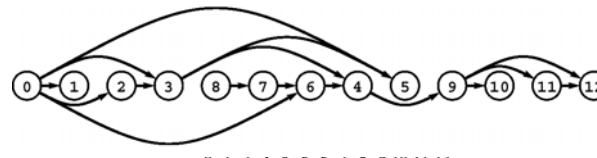
DFS

transitive closure



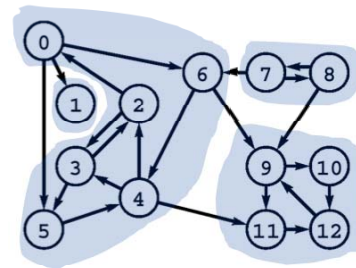
DFS from each vertex

topological sort  
(DAG)



DFS

strong components



Kosaraju  
DFS (twice)