
http://algs4.cs.princeton.edu

# 4.2 DIRECTED GRAPHS

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

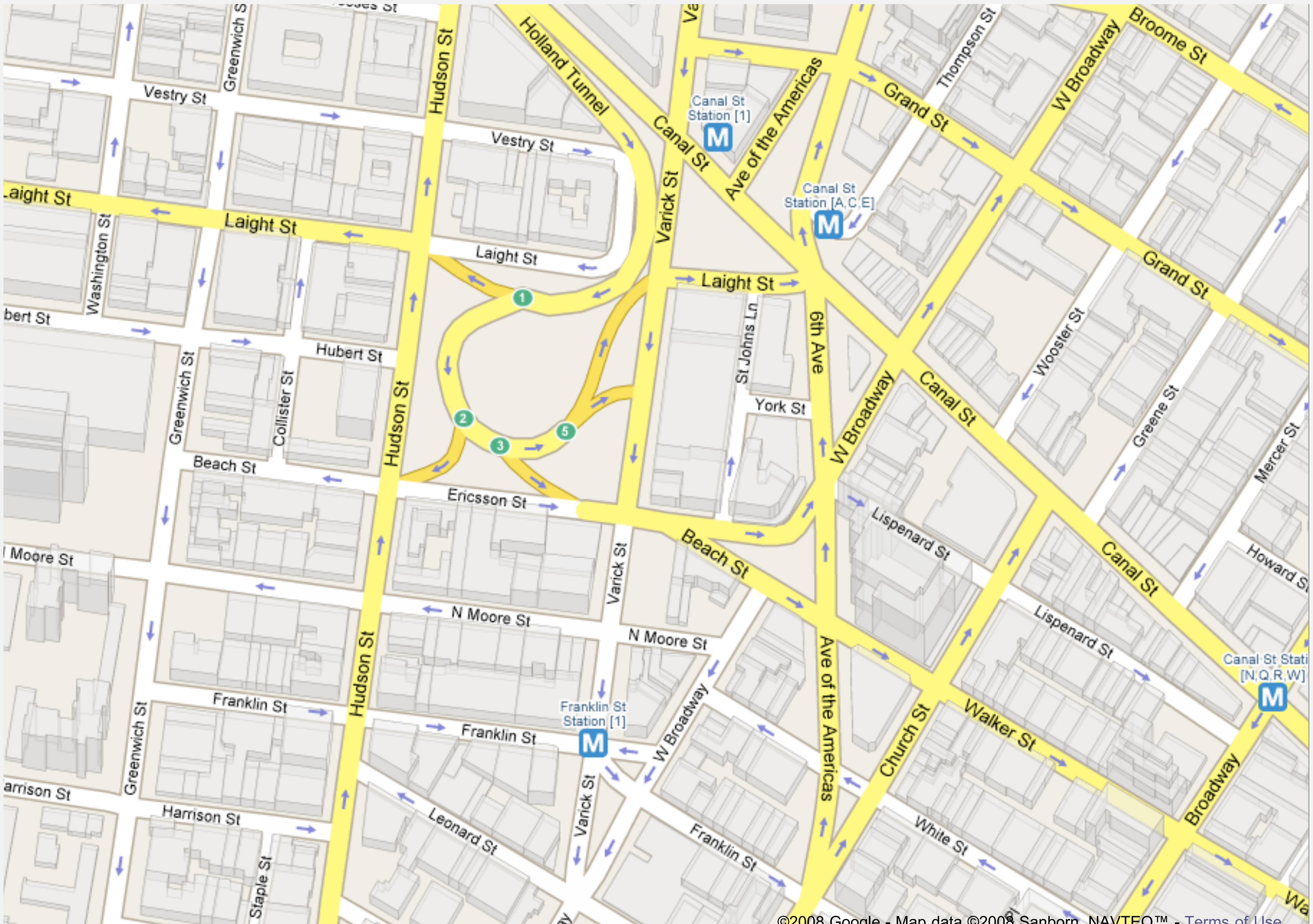

http://algs4.cs.princeton.edu

# 4.2 Directed Graphs

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

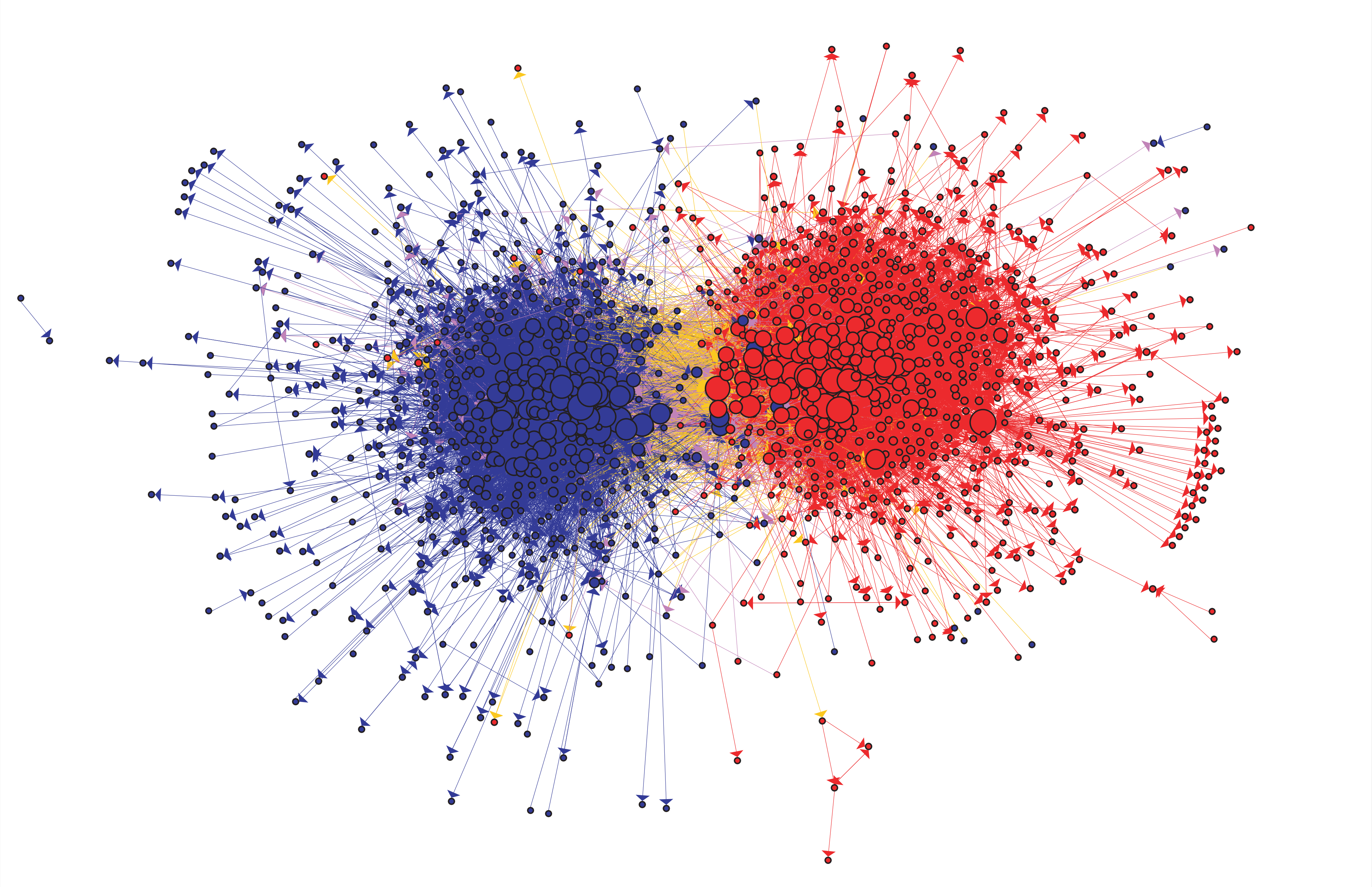
# Road networks

Vertex = intersection; edge = one-way street.



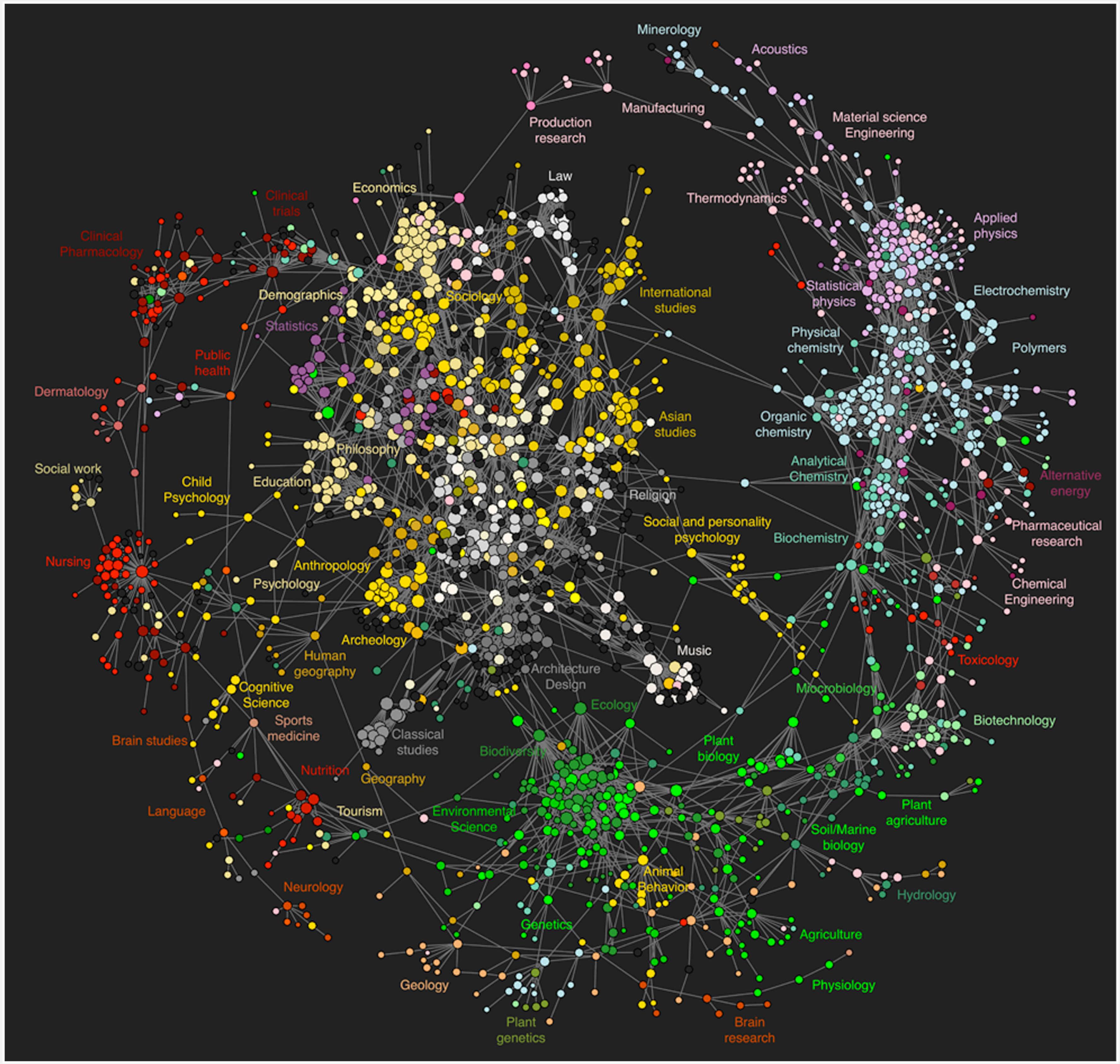
# Political blogosphere links

Vertex = political blog; edge = link.



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

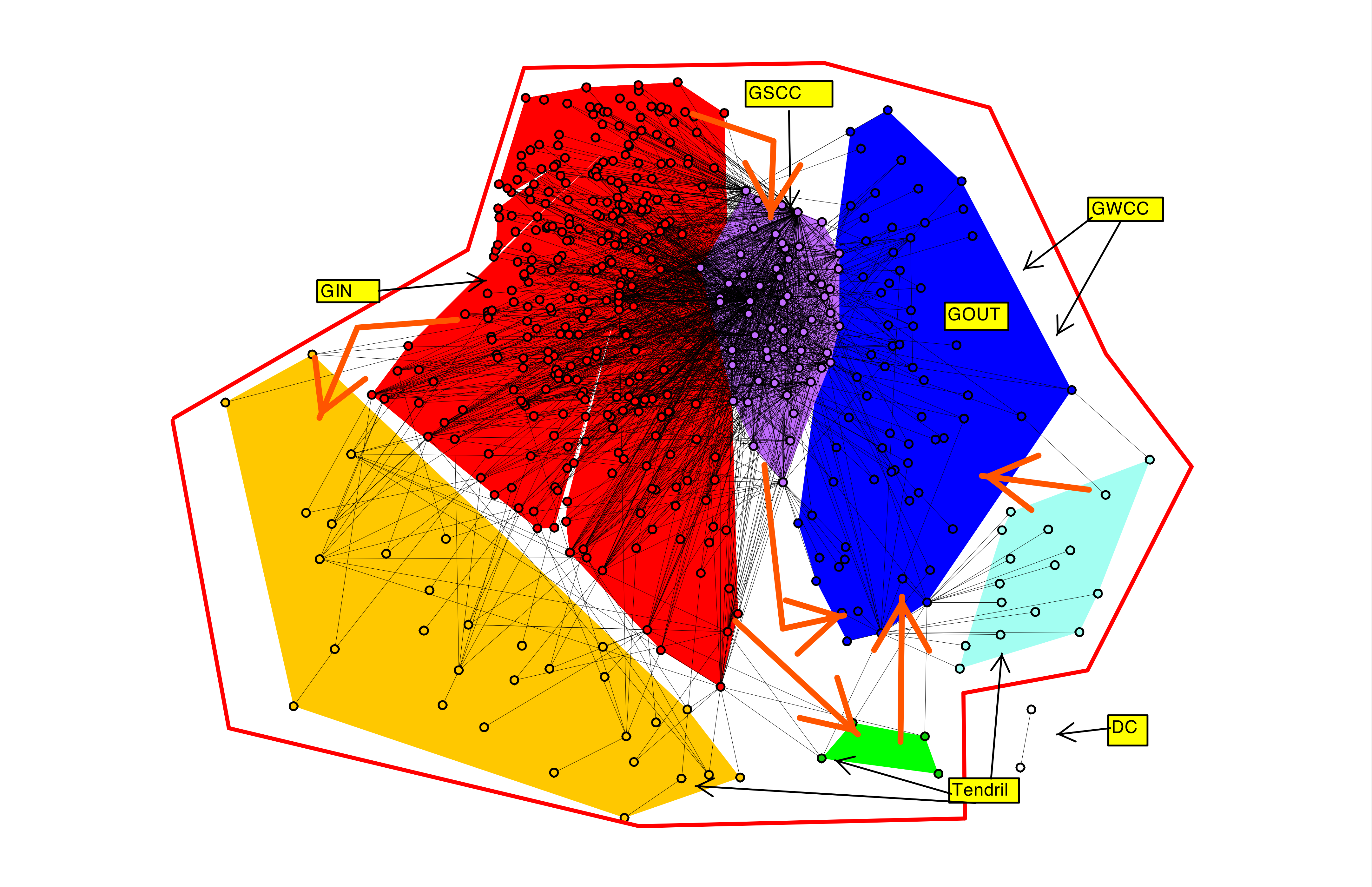
# Science clickstreams




http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803

# Overnight interbank loans

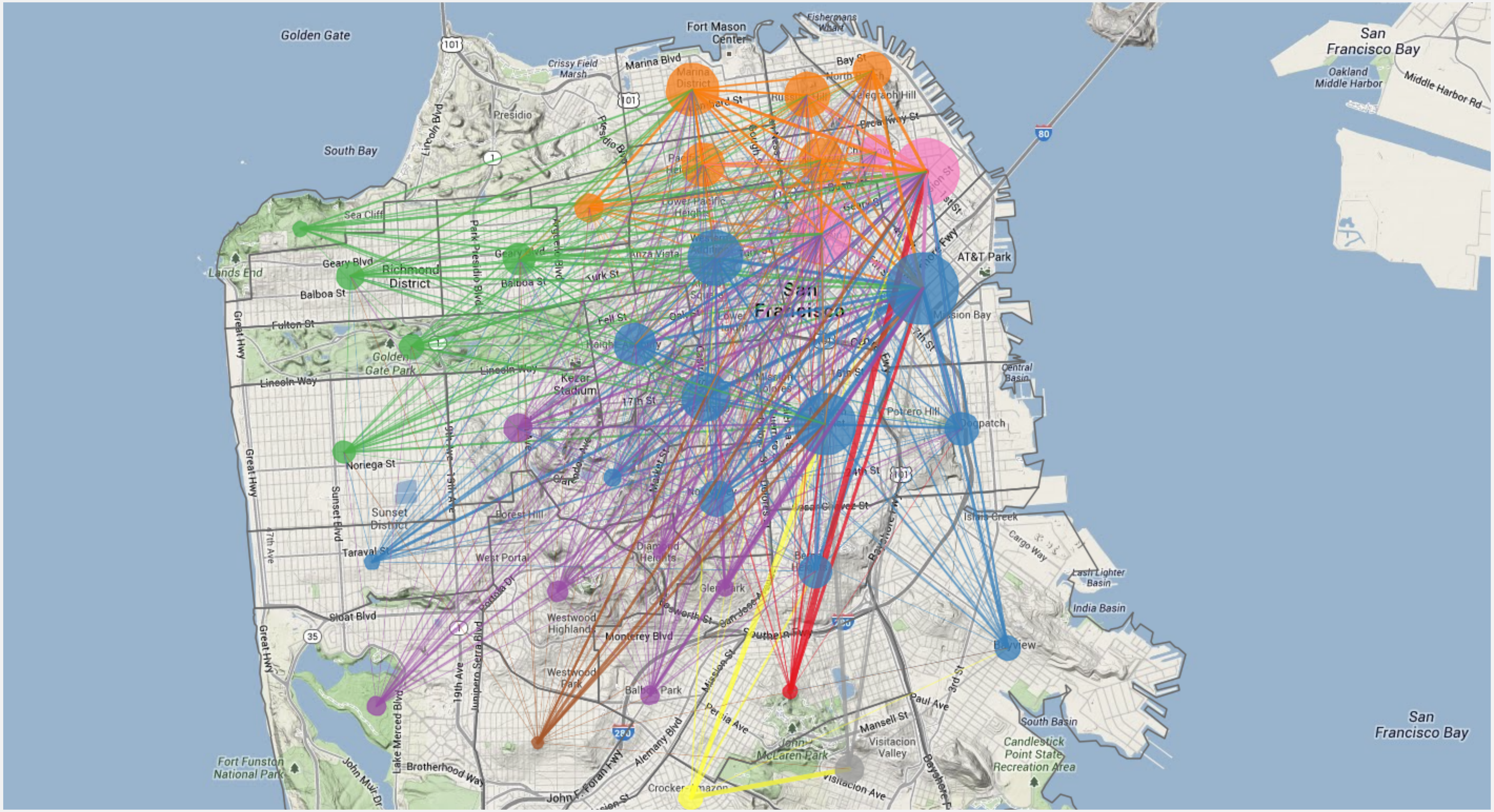
Vertex = bank; edge = overnight loan.




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

# Uber rides

Vertex = taxi pickup; edge = taxi ride.



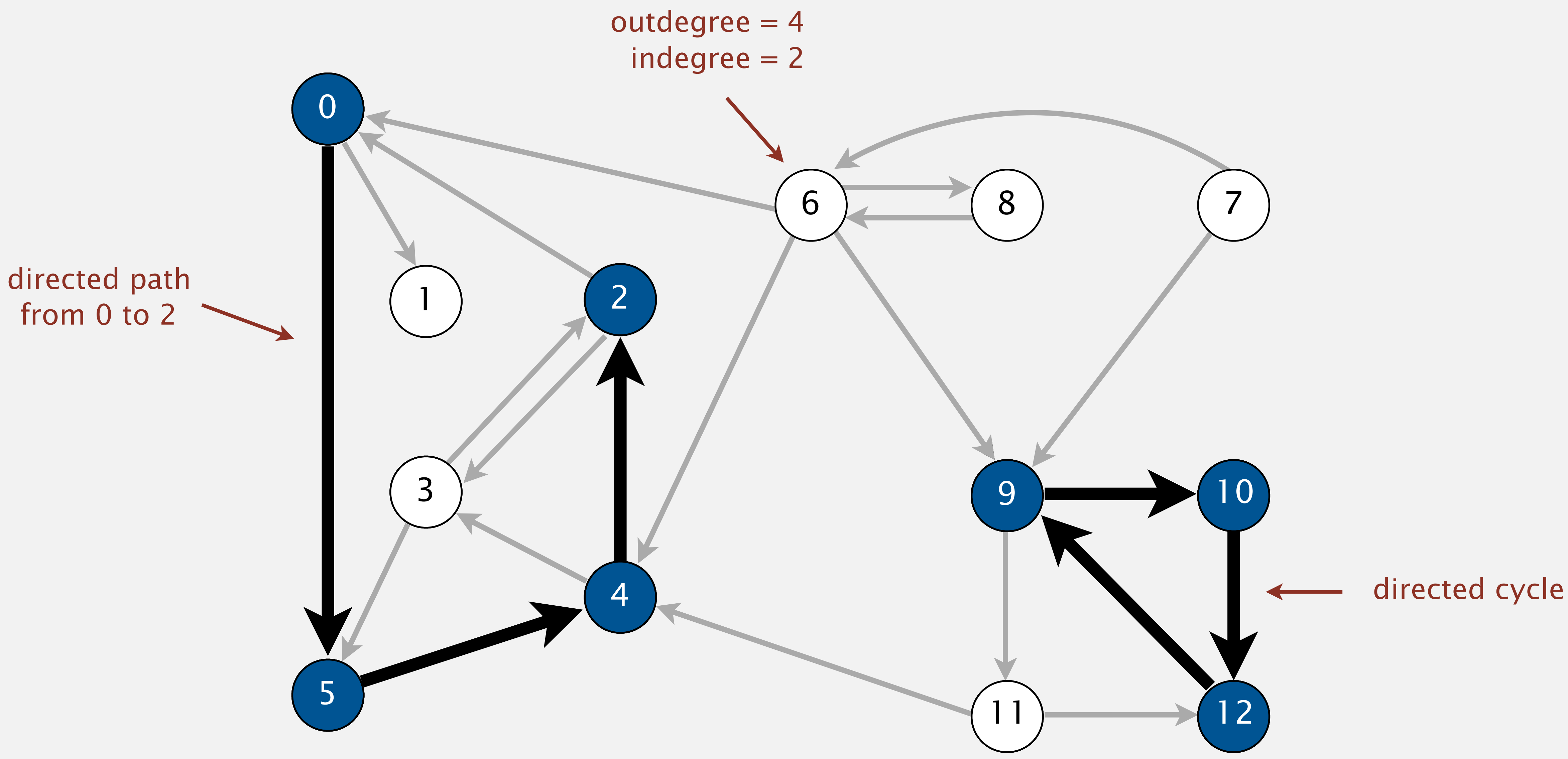
http://blog.uber.com/2012/01/09/uberdata-san-franciscomics

# Digraph applications

| 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 |

# Directed graph terminology

Digraph. Set of vertices connected pairwise by directed edges.



# Some digraph problems

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


http://algs4.cs.princeton.edu

# 4.2 Directed Graphs

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

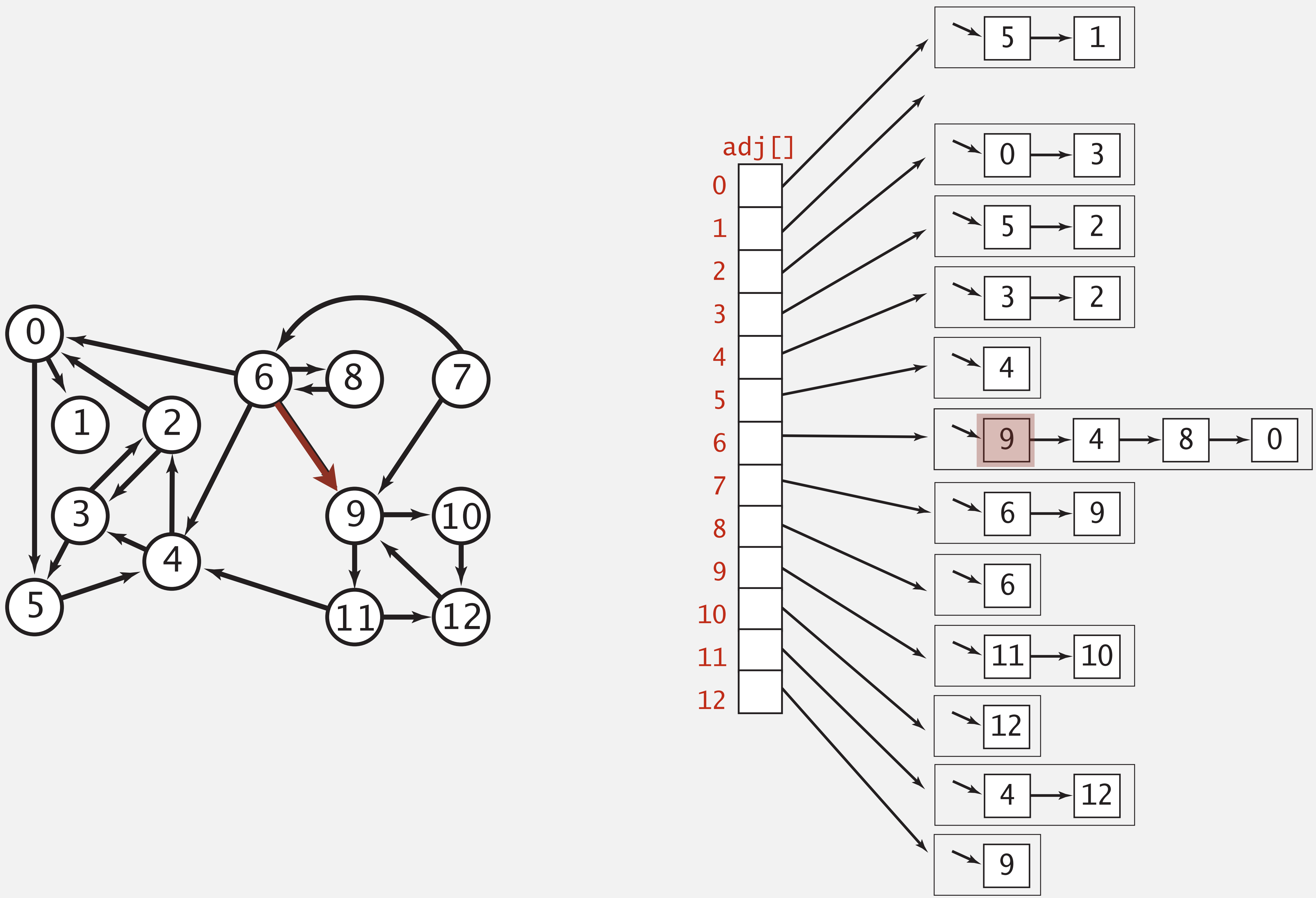
# Digraph API

Almost identical to `Graph` API.

| public class Digraph | | |
|---|---|---|
| | `Digraph(int V)` | *create an empty digraph with V vertices* |
| | `Digraph(In in)` | *create a digraph from input stream* |
| `void` | `addEdge(int v, int w)` | *add a directed edge v→w* |
| `Iterable<Integer>` | `adj(int v)` | *vertices adjacent from v* |
| `int` | `V()` | *number of vertices* |
| `int` | `E()` | *number of edges* |
| `Digraph` | `reverse()` | *reverse of this digraph* |
| `String` | `toString()` | *string representation* |

# Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.



# Directed graphs: quiz 1

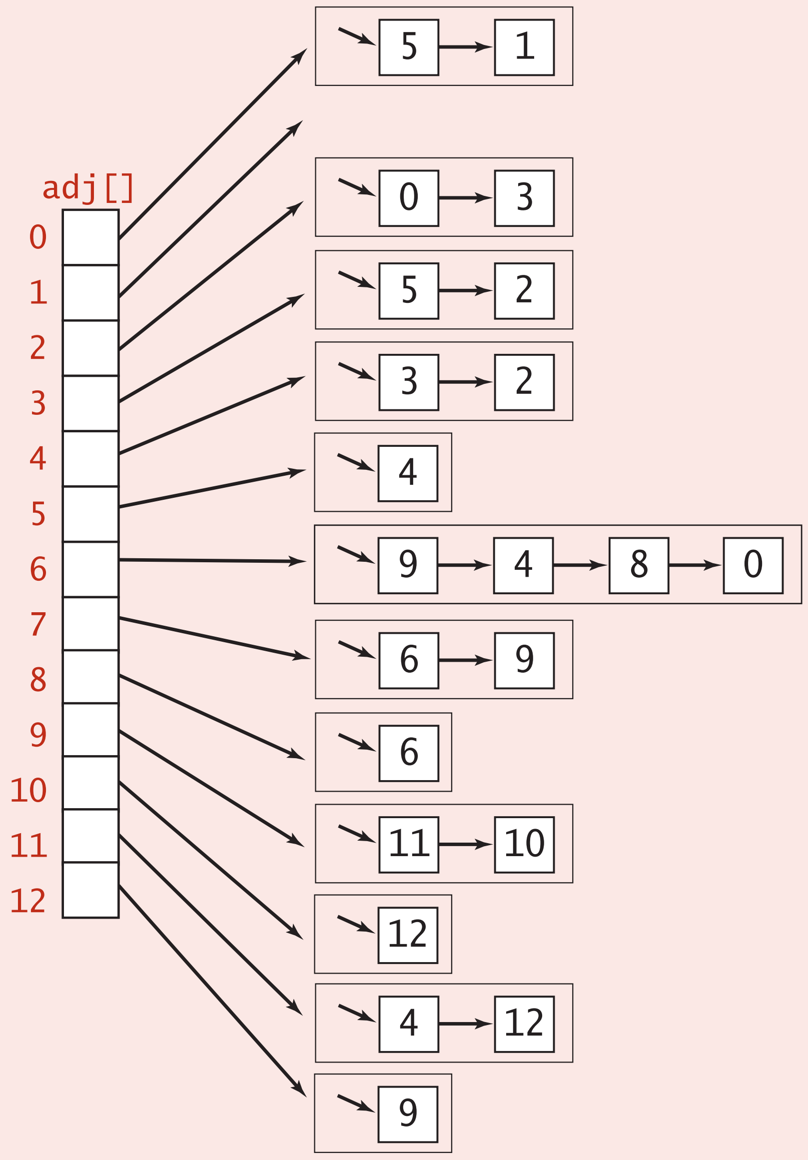
**Which is order of growth of running time of removing an edge v→w from a digraph uses the adjacency-lists representation, where $V$ is the number of vertices and $E$ is the number of edges?**

**A.** $1$

**B.** $outdegree(v)$

**C.** $indegree(w)$

**D.** $outdegree(v) + indegree(w)$



# Directed graphs: quiz 2

**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.** $E + V$

**C.** $V^2$

**D.** $V E$

```
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).

sparse: proportional to $V$ edges

dense: 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 † | 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;

   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>();
   }

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

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

adjacency lists

create empty graph with V vertices

add edge v–w

iterator for vertices adjacent to v

# Adjacency-lists digraph representation: Java implementation

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

   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>();
   }

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

   }

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

adjacency lists

create empty digraph with V vertices

add edge v→w

iterator for vertices adjacent from v

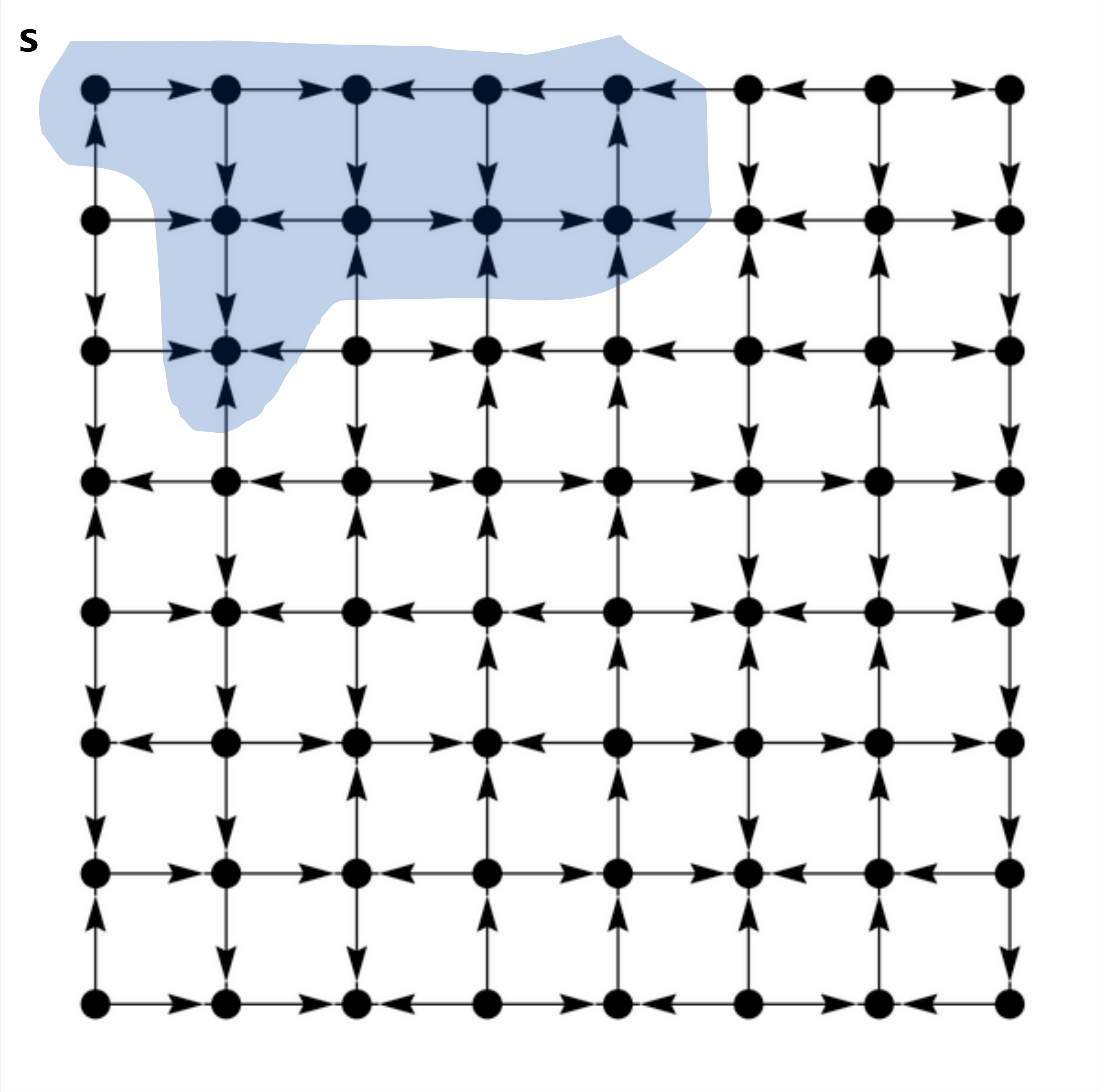

http://algs4.cs.princeton.edu

# 4.2 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)**

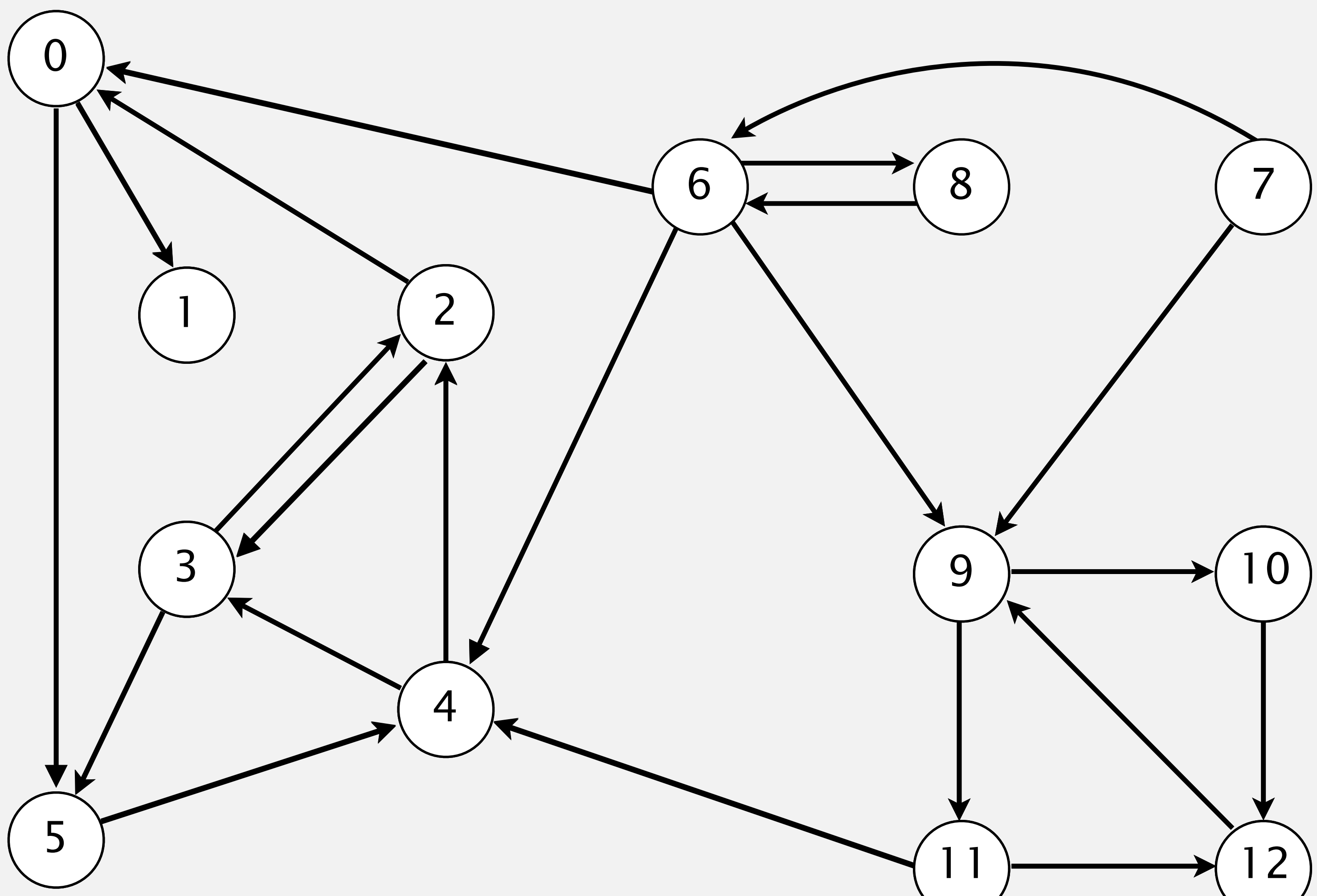
**Mark vertex v.**

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

# Depth-first search demo

To visit a vertex $v$ :

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



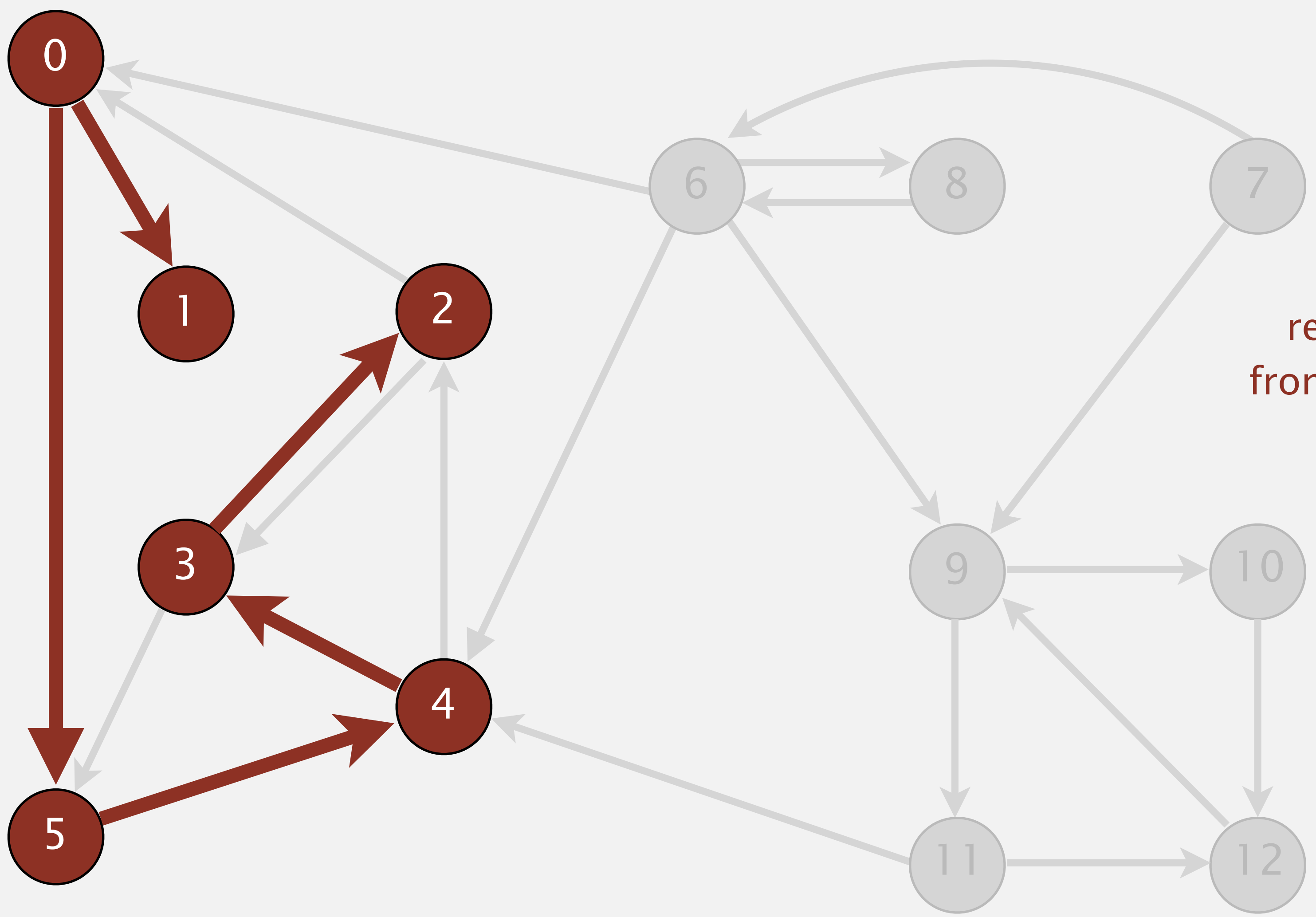

a directed graph

4→2
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

# Depth-first search demo

To visit a vertex $v$ :

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




reachable from vertex 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 | – |

**reachable from 0**

# 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];  }
}
```

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 (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];  }
}
```

- `private boolean[] marked;` ← true if connected to s
- constructor ← constructor marks vertices connected to s
- `dfs()` ← recursive DFS does the work
- `visited()` ← client can ask whether any vertex is connected to s

# Reachability application: program control-flow analysis

## Every program is a digraph.

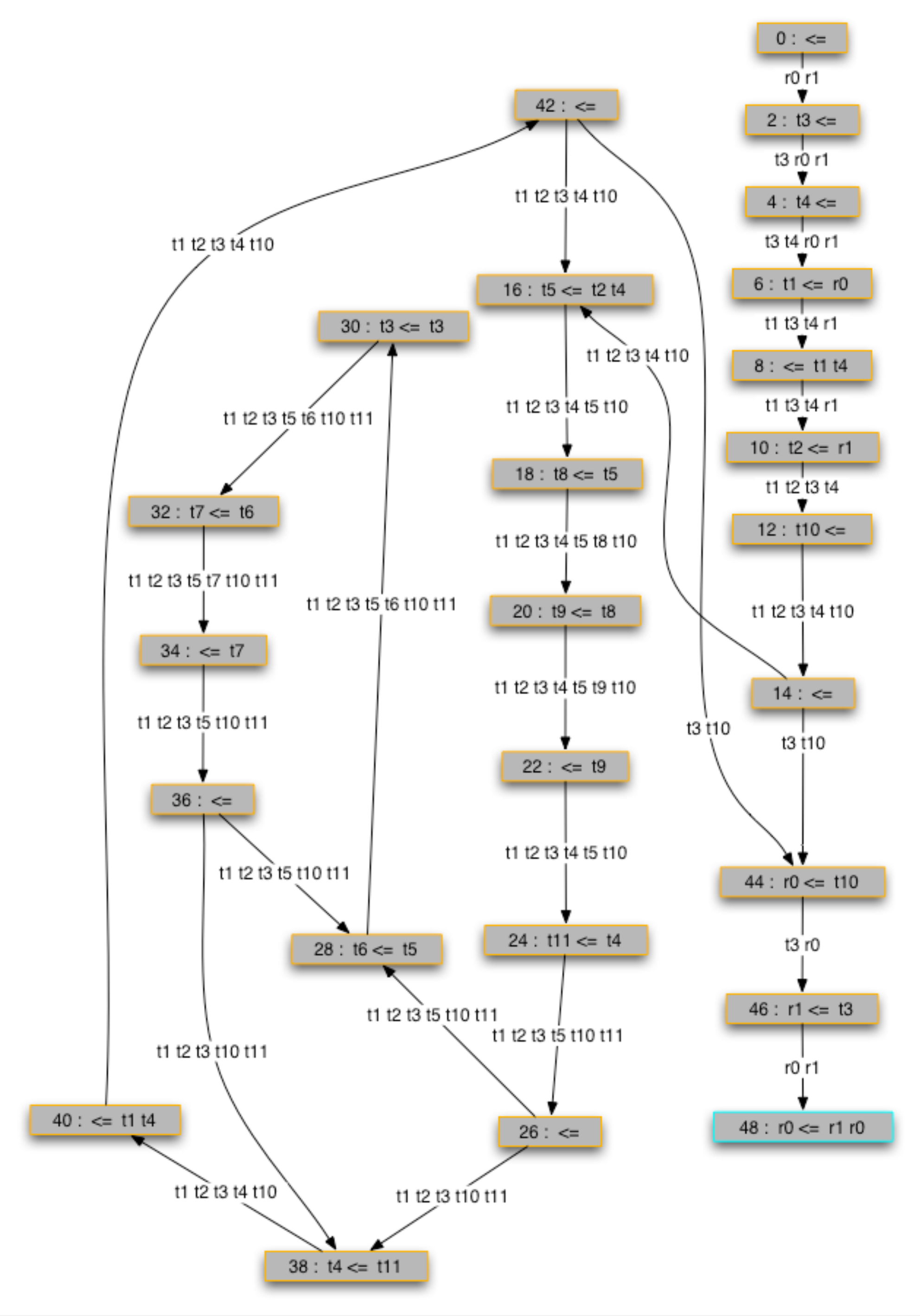
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

## Dead-code elimination.

Find (and remove) unreachable code.

## Infinite-loop detection.

Determine whether exit is unreachable.



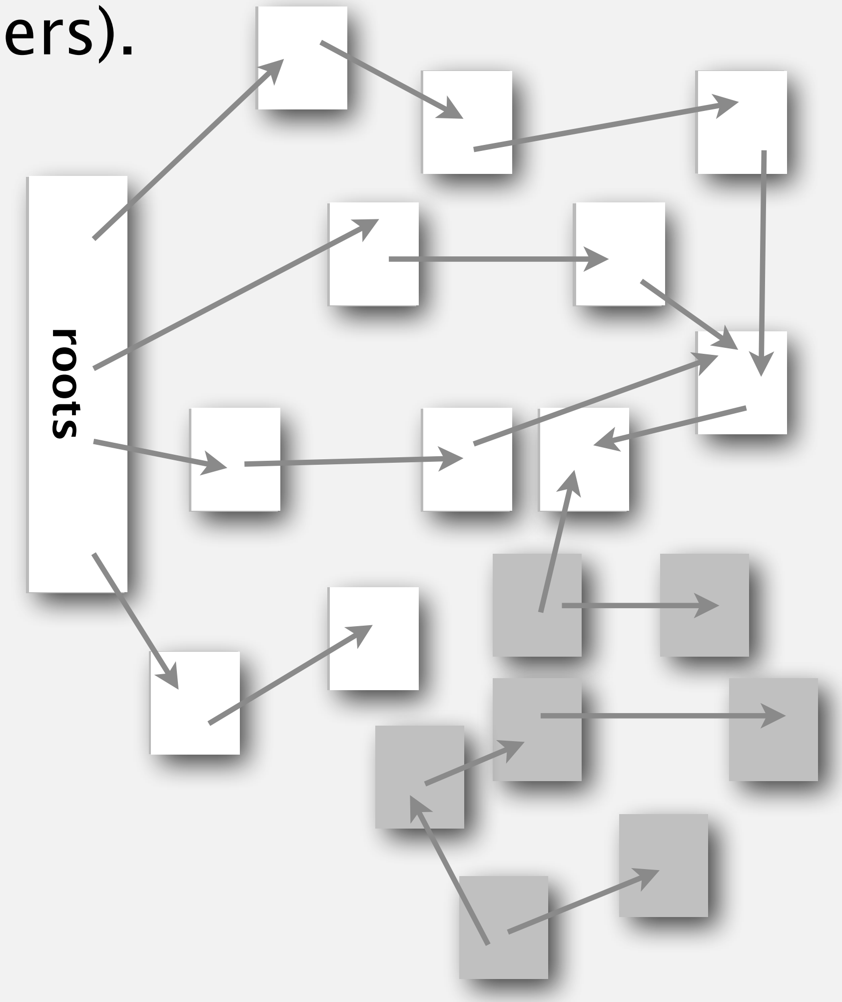
# Reachability application: mark-sweep garbage collector

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).

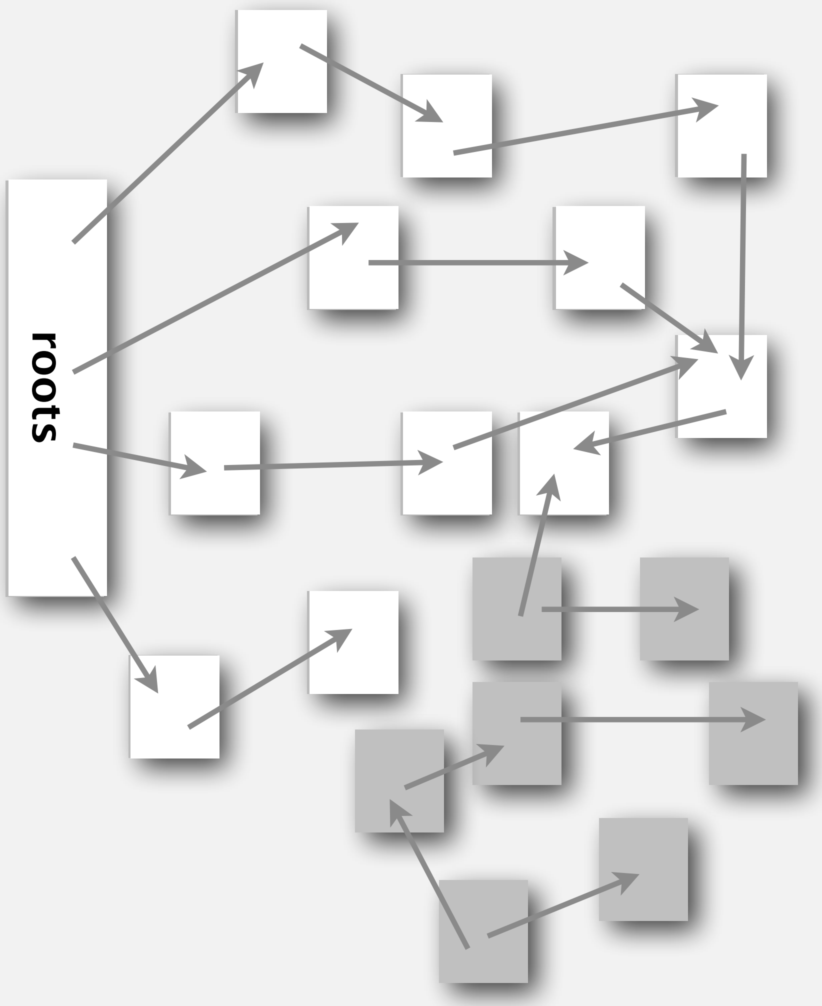


# Reachability application: mark-sweep garbage collector

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).



# Depth-first search in digraphs summary

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 undirect 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

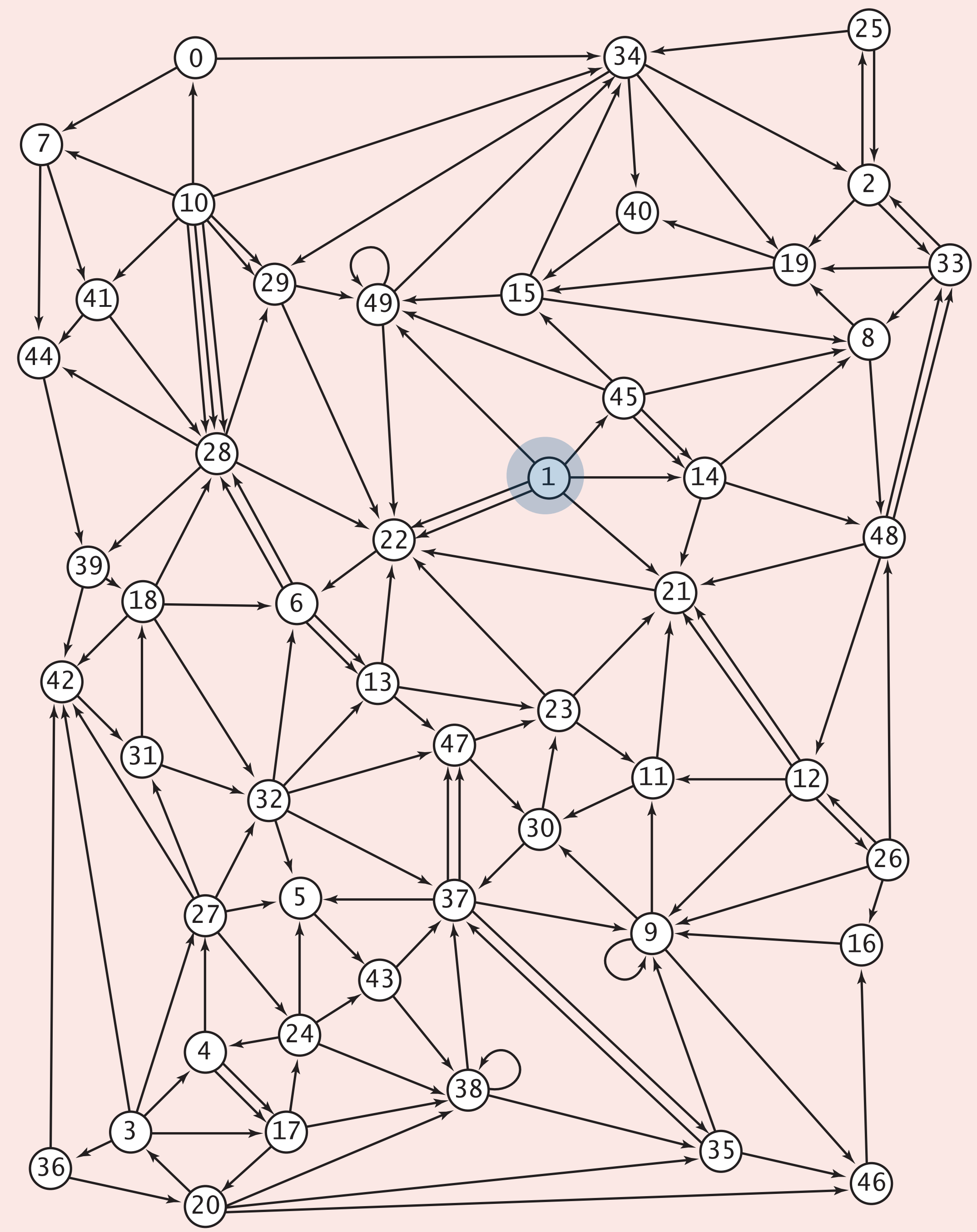
**Suppose that you want to design a web crawler. Which graph search algorithm should you use?**

**A.** depth-first search

**B.** breadth-first search

**C.** either A or B

**D.** neither A nor B

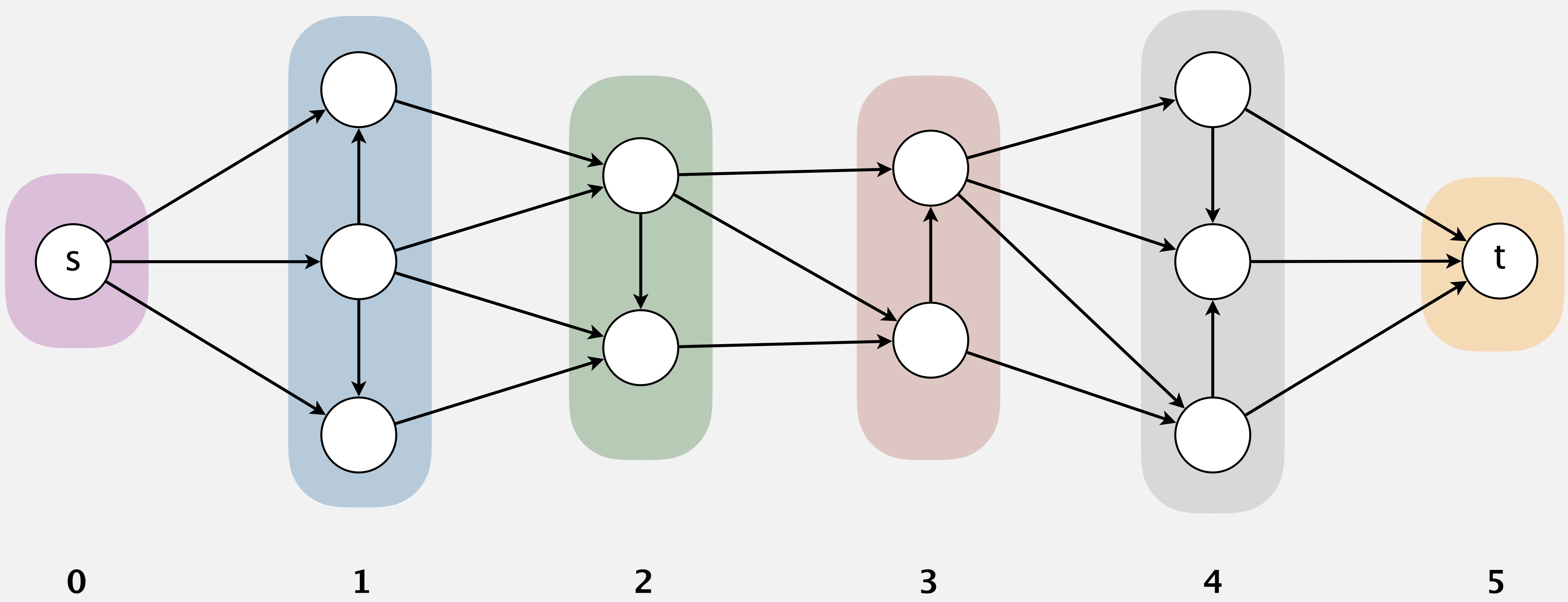



http://algs4.cs.princeton.edu

# 4.2 Directed Graphs

# Shortest directed paths

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 s onto a FIFO queue, and mark s as visited.**

**Repeat until the queue is empty:**

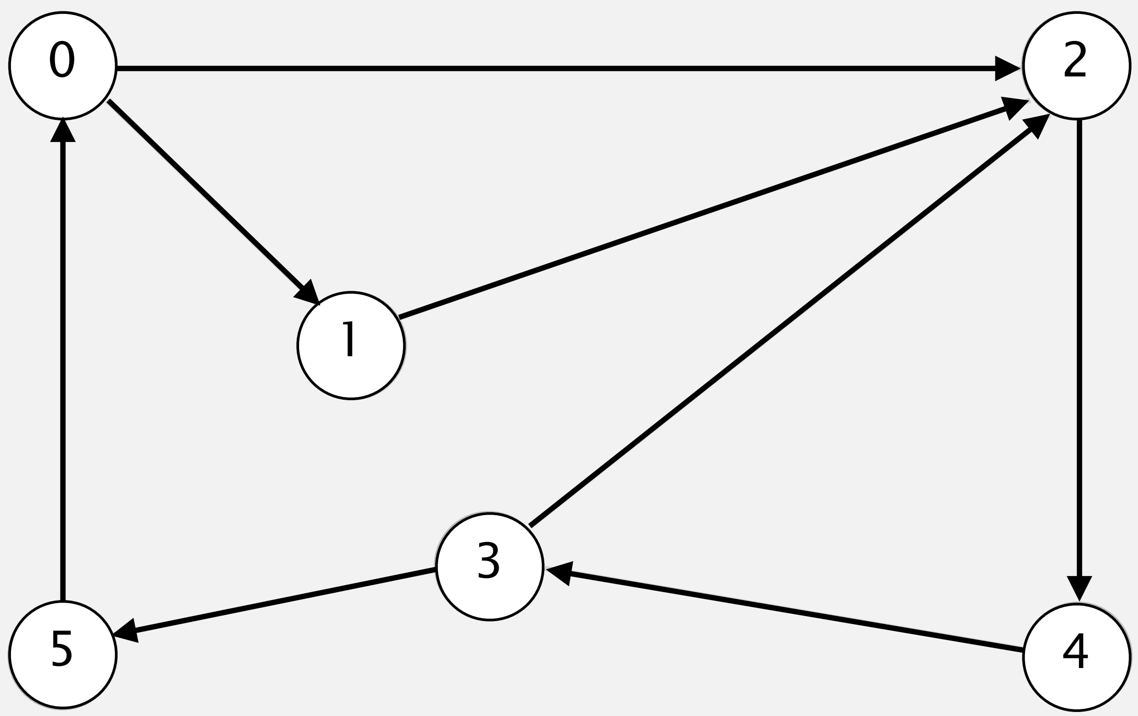
- **remove the least recently added vertex v**
- **for each unmarked vertex adjacent from v: add to queue and mark as visited.**

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

# Directed breadth-first search demo

Repeat until queue is empty:

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




graph G

**tinyDG2.txt**

V → 6

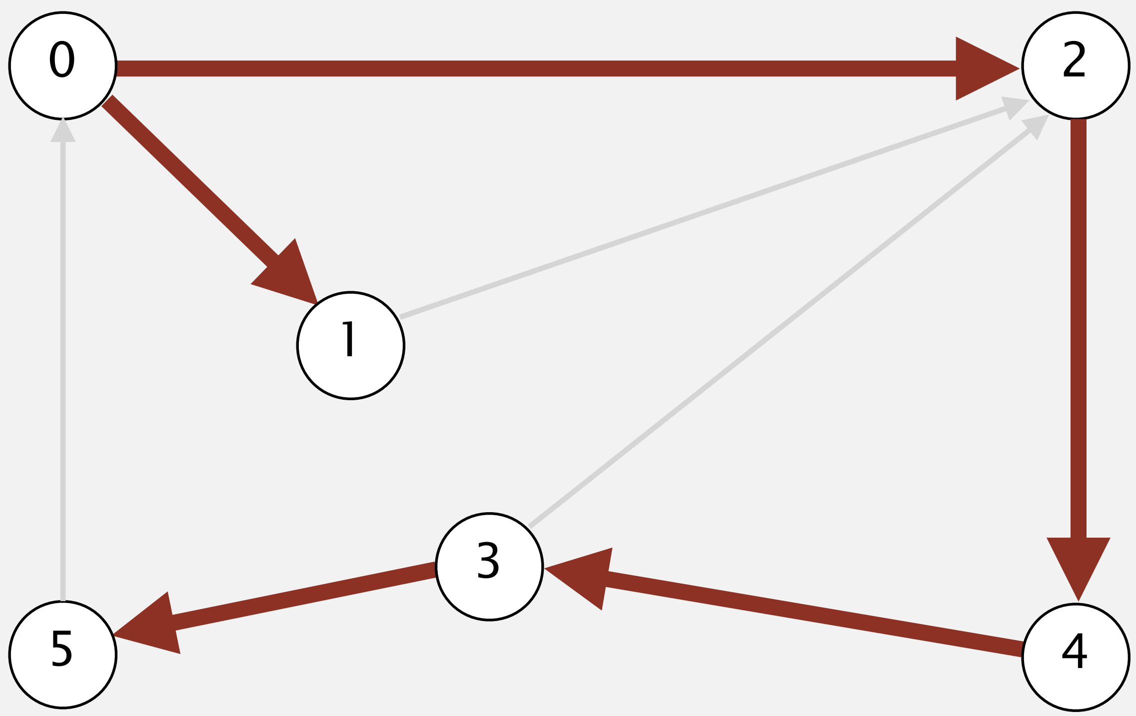
E → 8

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

# Directed breadth-first search demo

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 |

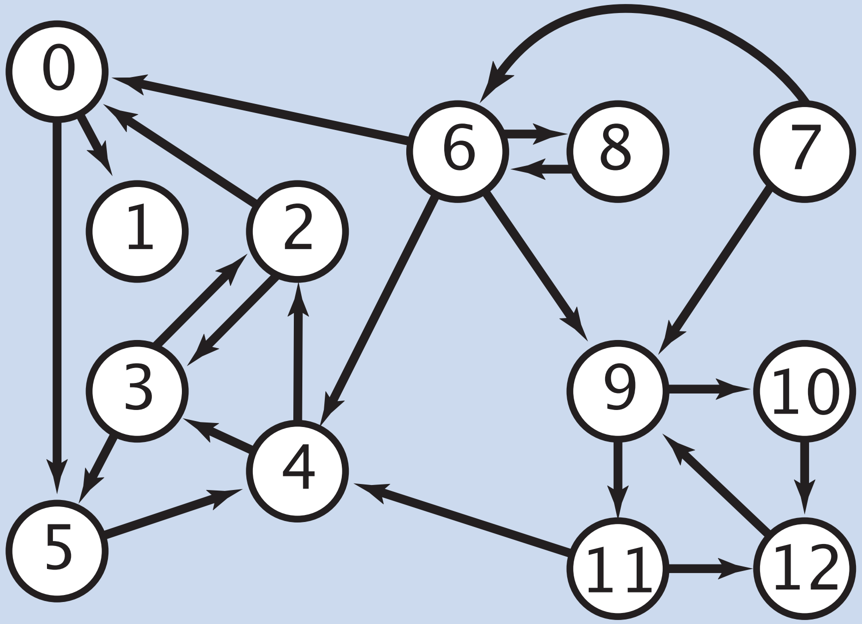
**done**

# Multiple-source shortest paths

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 $10 \rightarrow 12$.



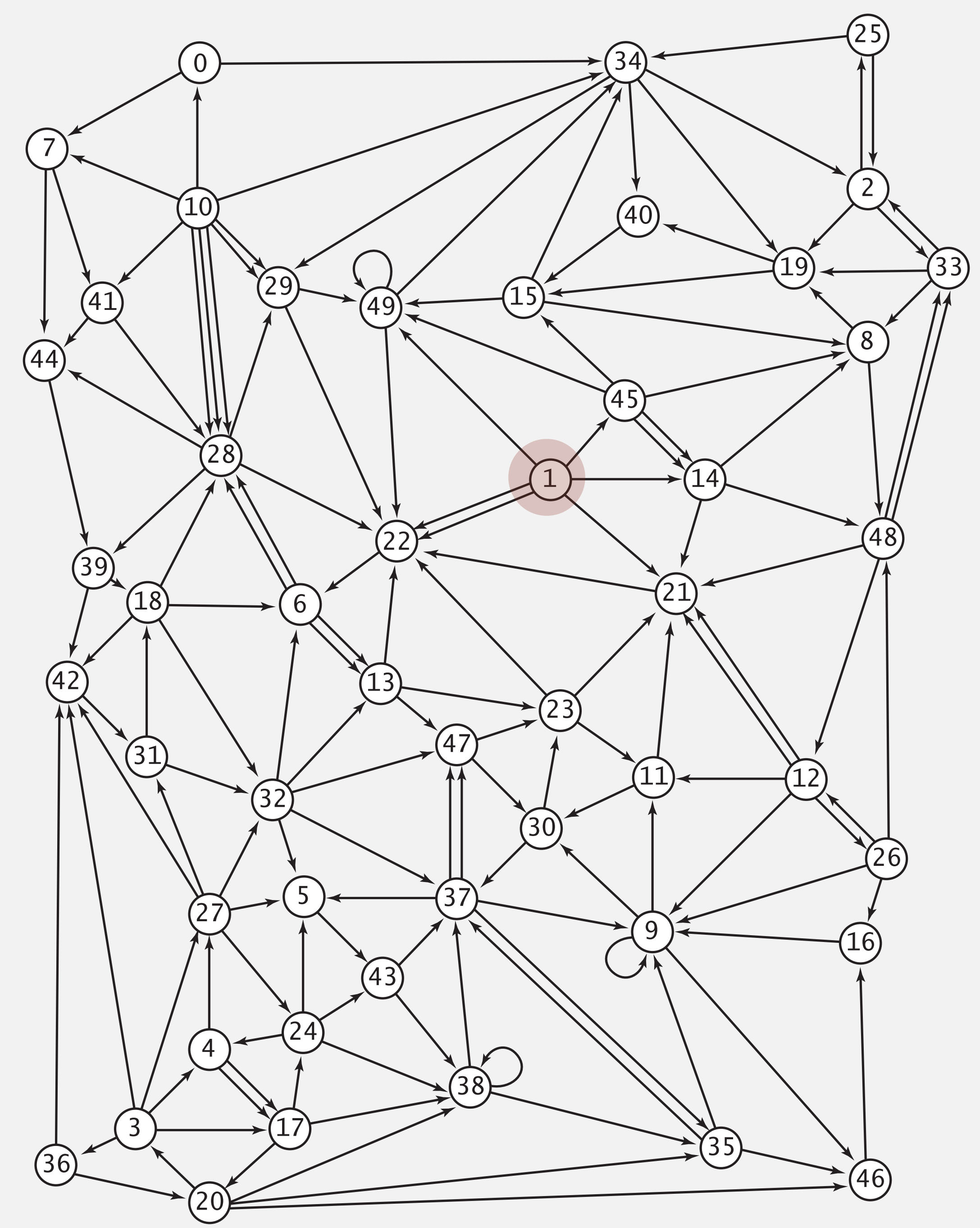

Q. How to implement multi-source shortest paths algorithm?

# Breadth-first search in digraphs application: web crawler

Goal. Crawl web, starting from some root web page, say `www.princeton.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.princeton.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+)";
    Pattern pattern = Pattern.compile(regexp);
    Matcher matcher = pattern.matcher(input);

    while (matcher.find())
    {
        String w = matcher.group();
        if (!marked.contains(w))
        {
            marked.add(w);
            q.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

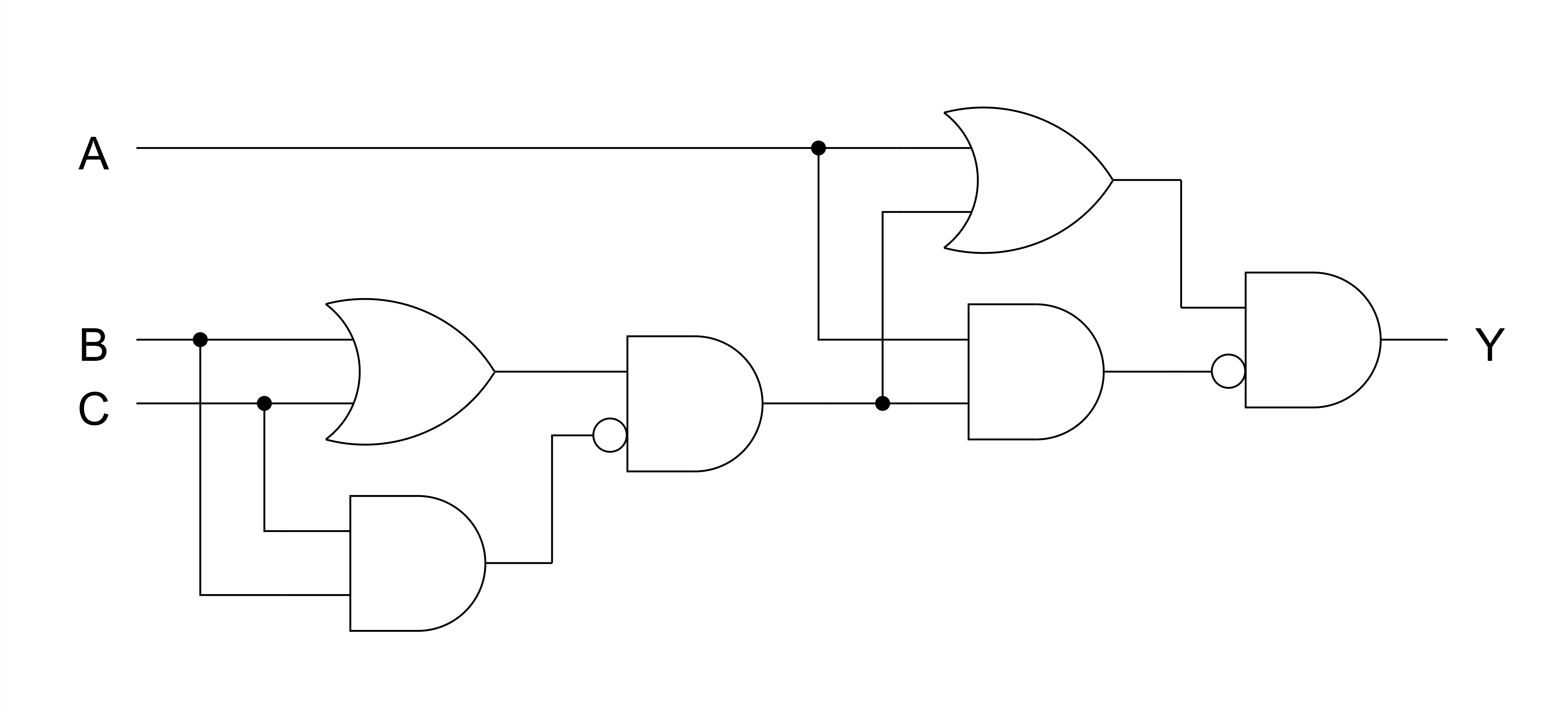

http://algs4.cs.princeton.edu

# 4.2 Directed Graphs

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

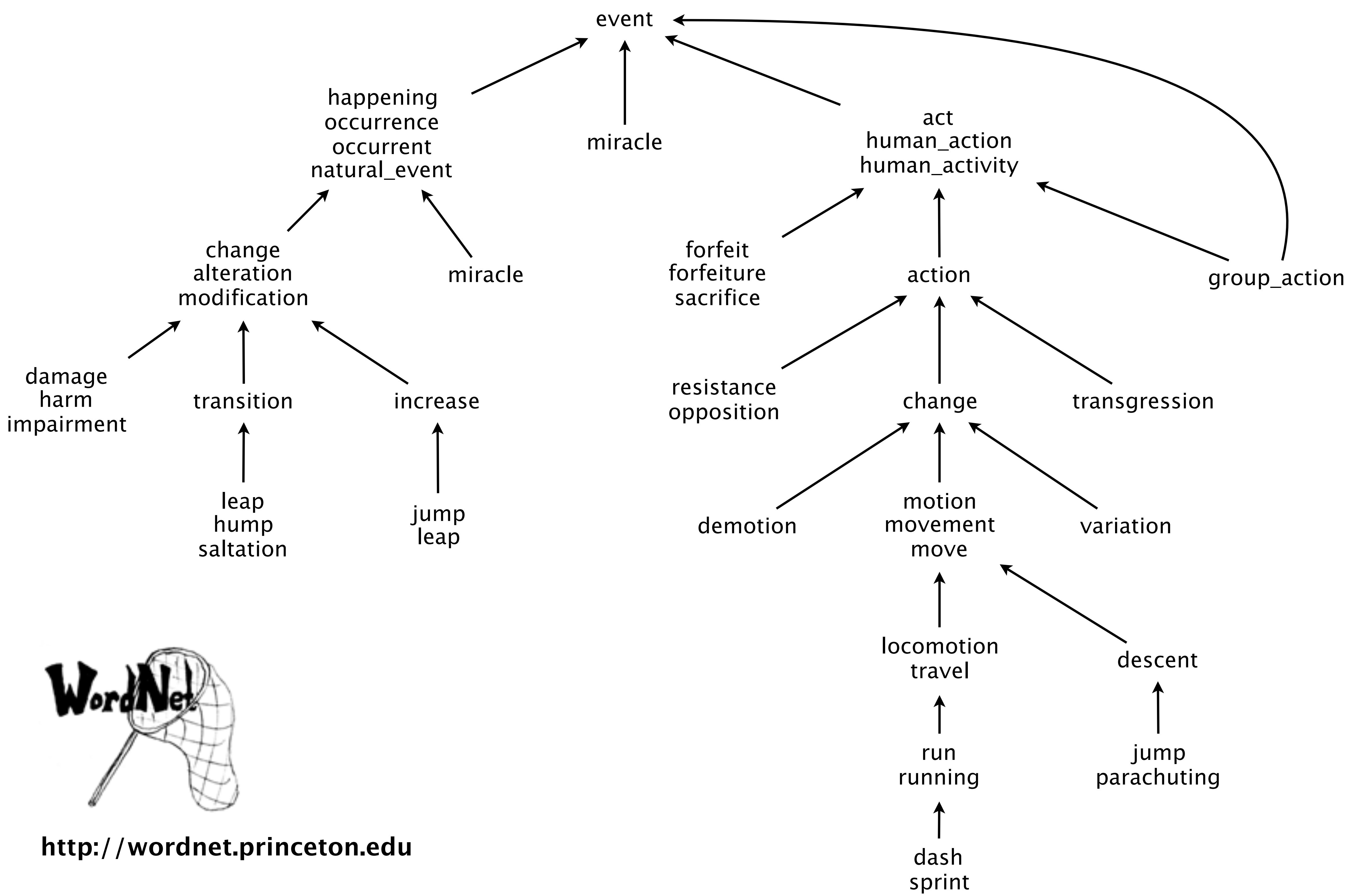
# Combinational circuit

Vertex = logical gate; edge = wire.



# WordNet digraph

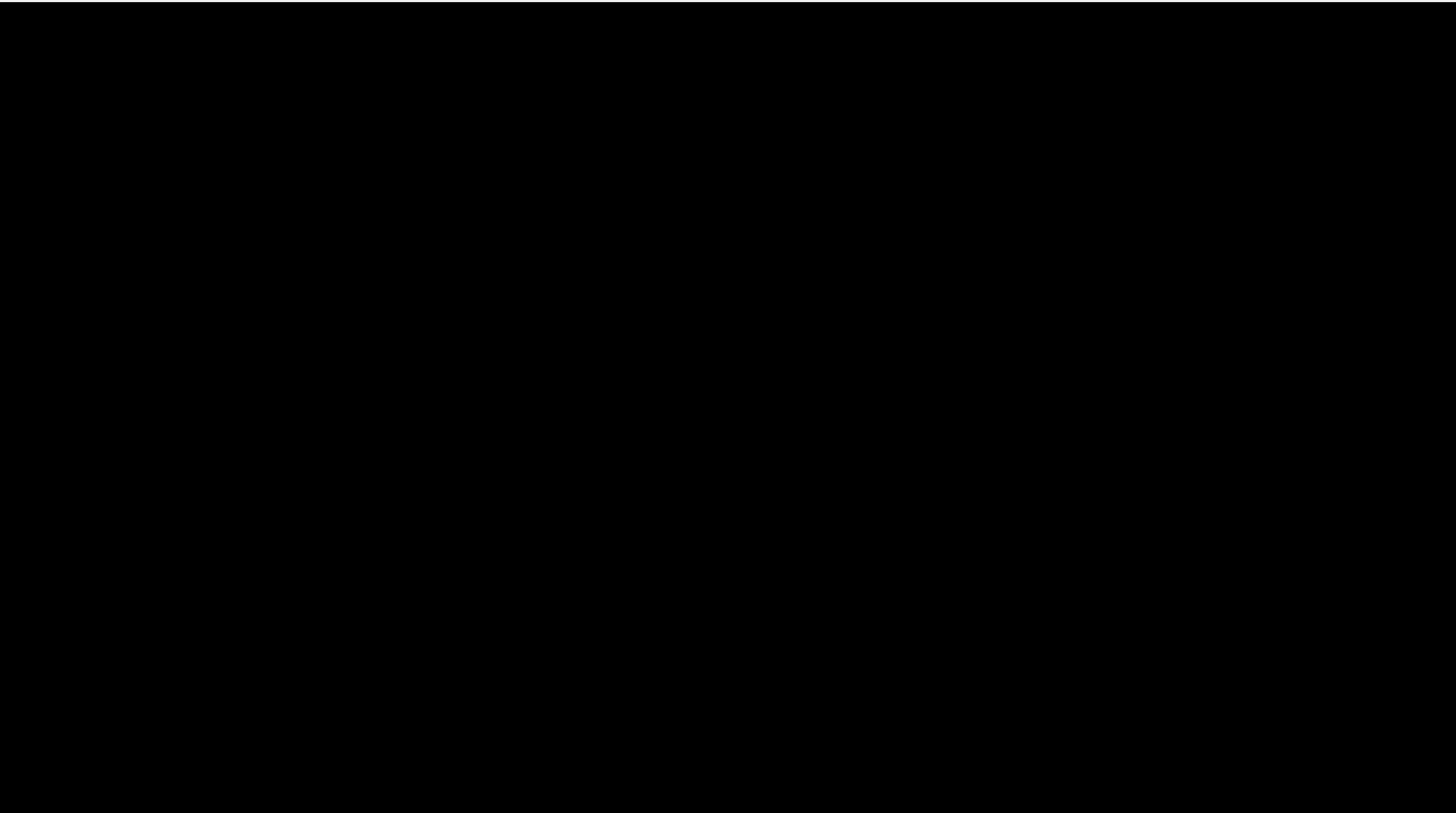
Vertex = synset; edge = hypernym relationship.




http://wordnet.princeton.edu

# Git digraph

Vertex = revision of repository; edge = revision relationship.



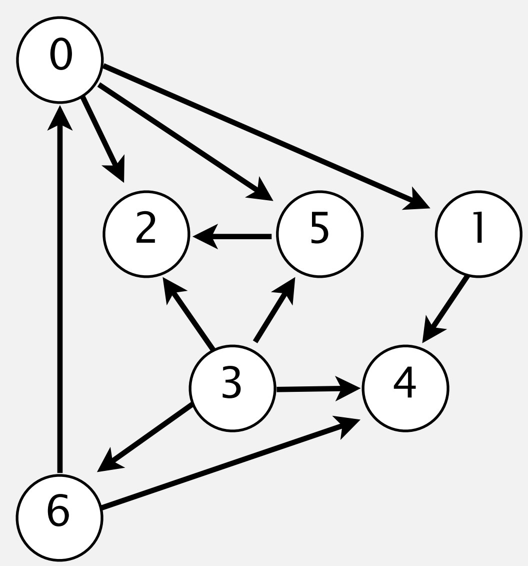
# Precedence scheduling

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

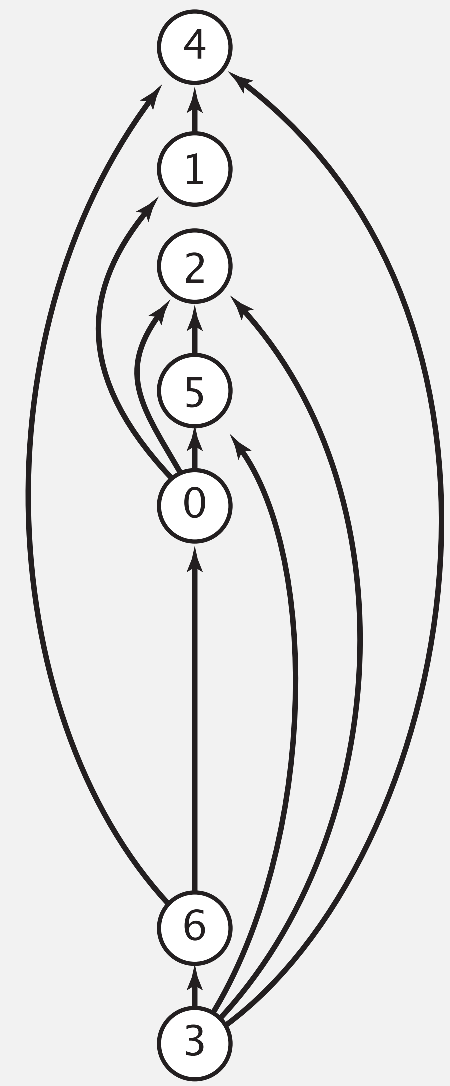
Digraph model. vertex = task; edge = precedence constraint.

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

**tasks**




**precedence constraint graph**




**feasible schedule**

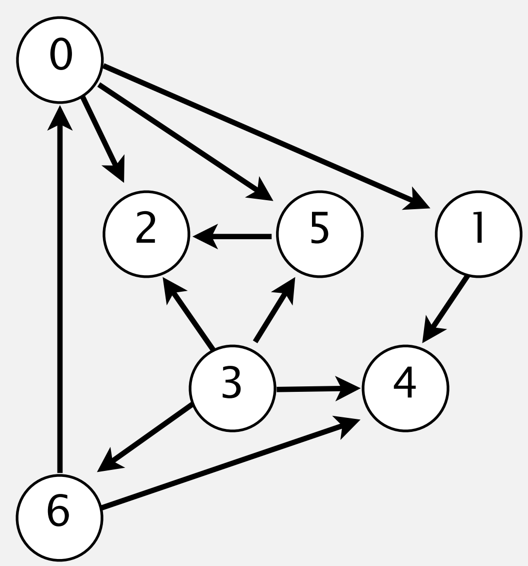
# Topological sort

DAG. Directed acyclic graph.

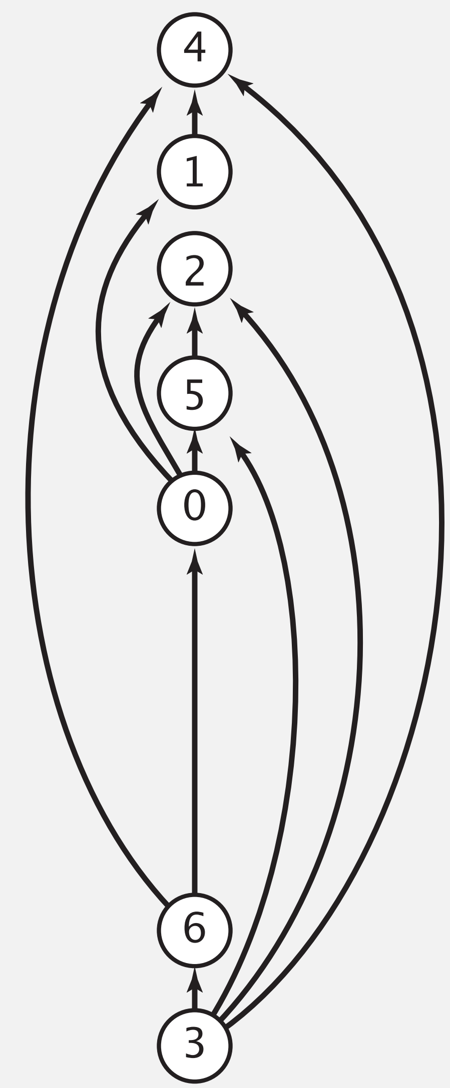
Topological sort. Redraw DAG so all edges point upwards.

| | |
|---|---|
| 0→5 | 0→2 |
| 0→1 | 3→6 |
| 3→5 | 3→4 |
| 5→2 | 6→4 |
| 6→0 | 3→2 |
| 1→4 | |

**directed edges**




**DAG**

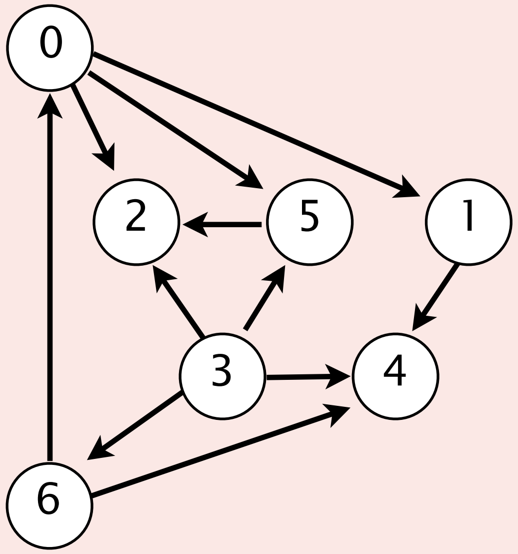



**topological order**

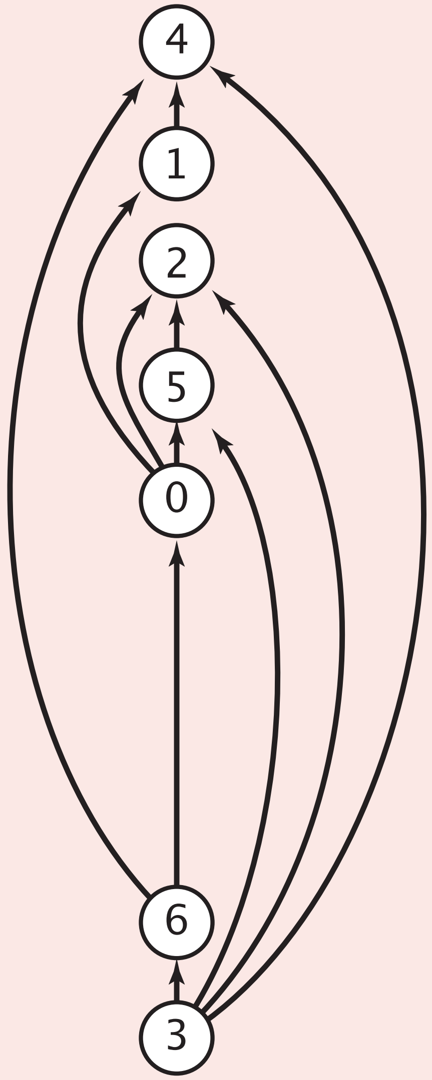
# Directed graphs: quiz 3

**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



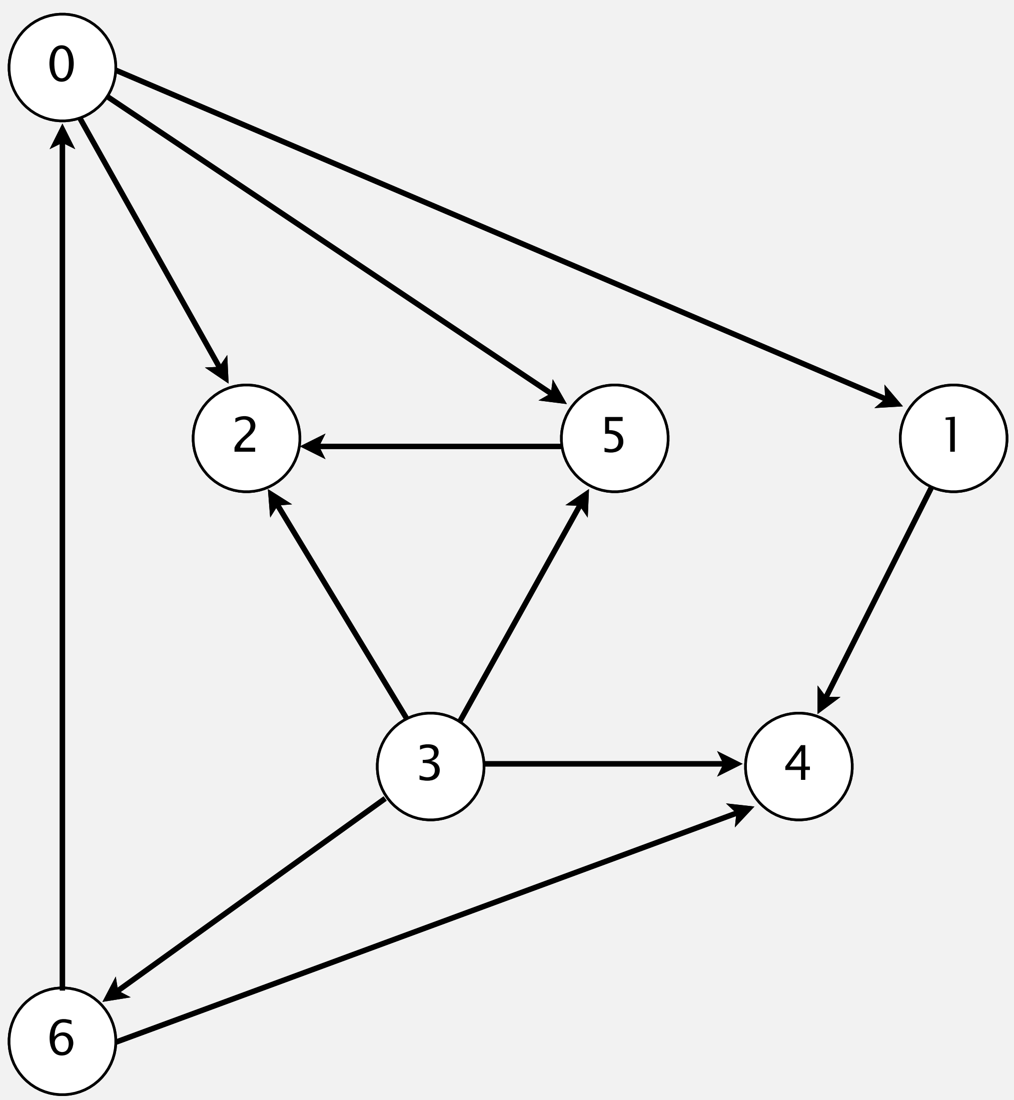

DAG




topological order

# Topological sort demo

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



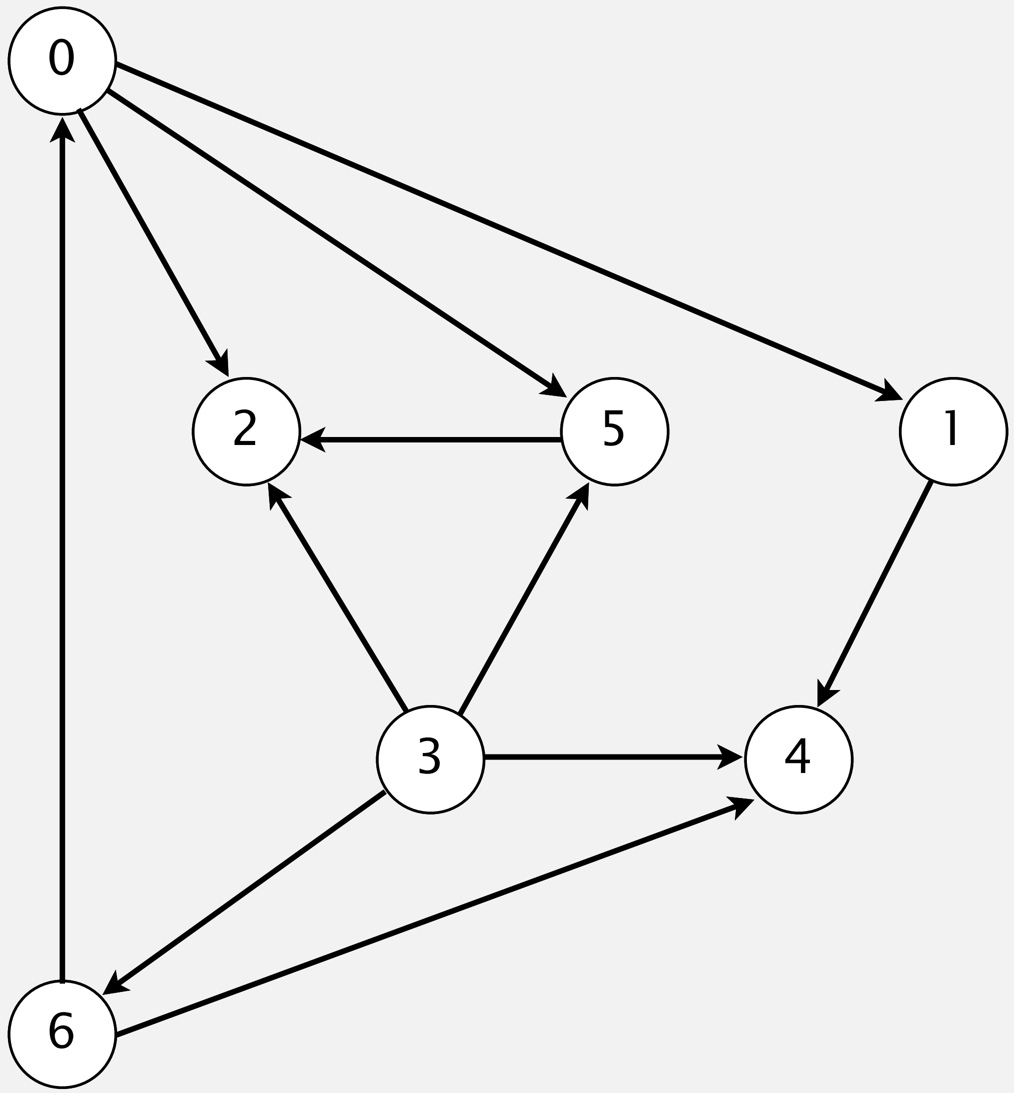

a directed acyclic graph

**tinyDAG7.txt**

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

# Topological sort demo

- 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

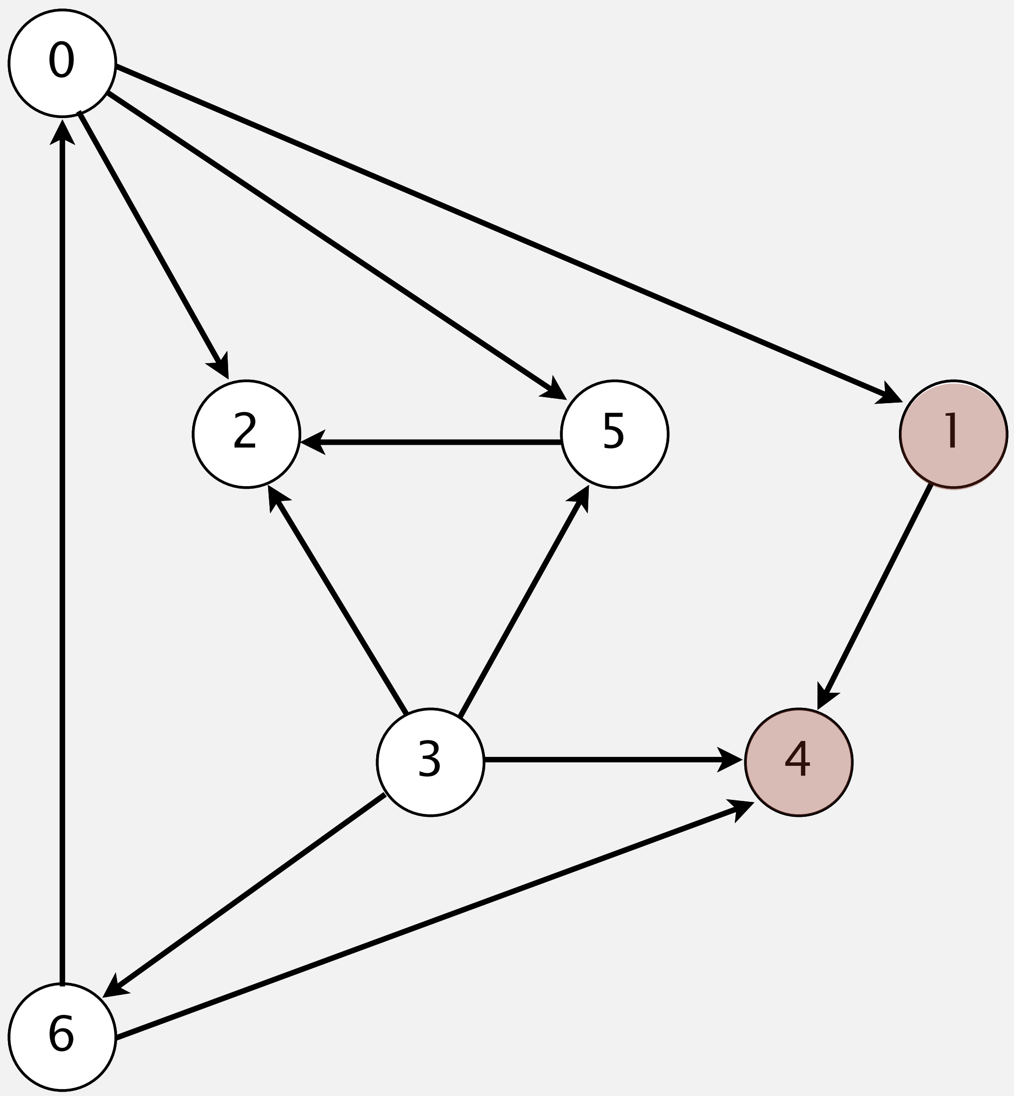
```
public class DepthFirstOrder
{
   private boolean[] marked;
   private Stack<Integer> reversePostorder;

   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.
- ...




**postorder**

4 1 2 5 0 6 3

**topological order**

3 6 0 5 2 1 4

# Topological sort in a DAG: correctness proof

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

**Pf.** Consider any edge `v→w`. When `dfs(v)` is called:

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

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

v = 3 → `dfs(3)`

case 1 (w = 2, 4, 5) → `check 2`, `check 4`, `check 5`

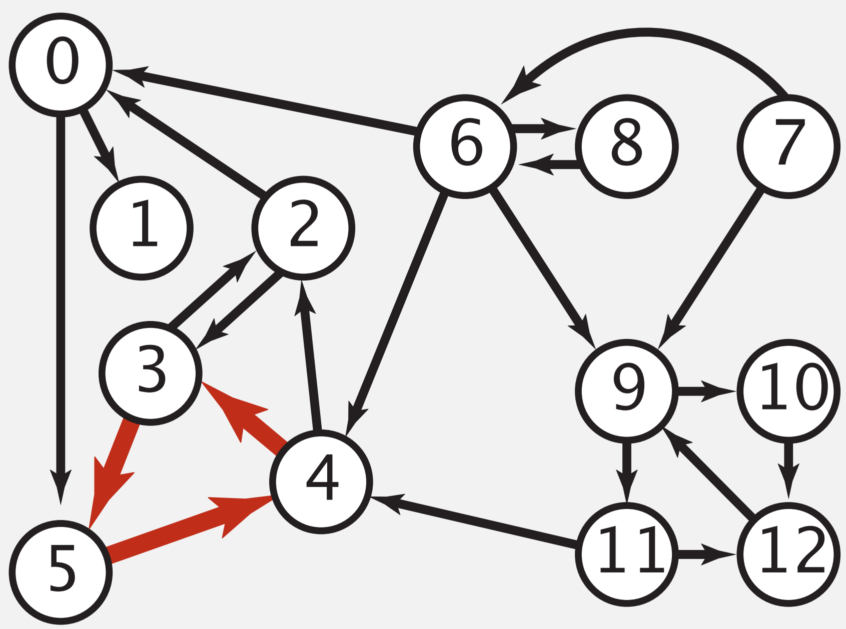
case 2 (w = 6) → `dfs(6)`, `6 done`

# 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.



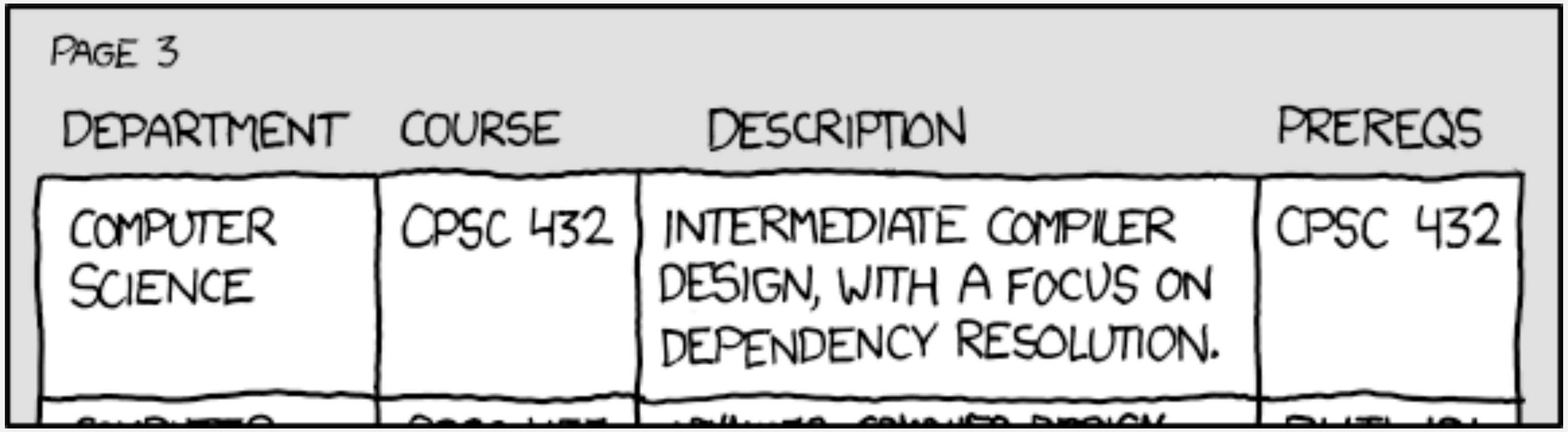

a digraph with a directed cycle

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?




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
{
   ...
}
```

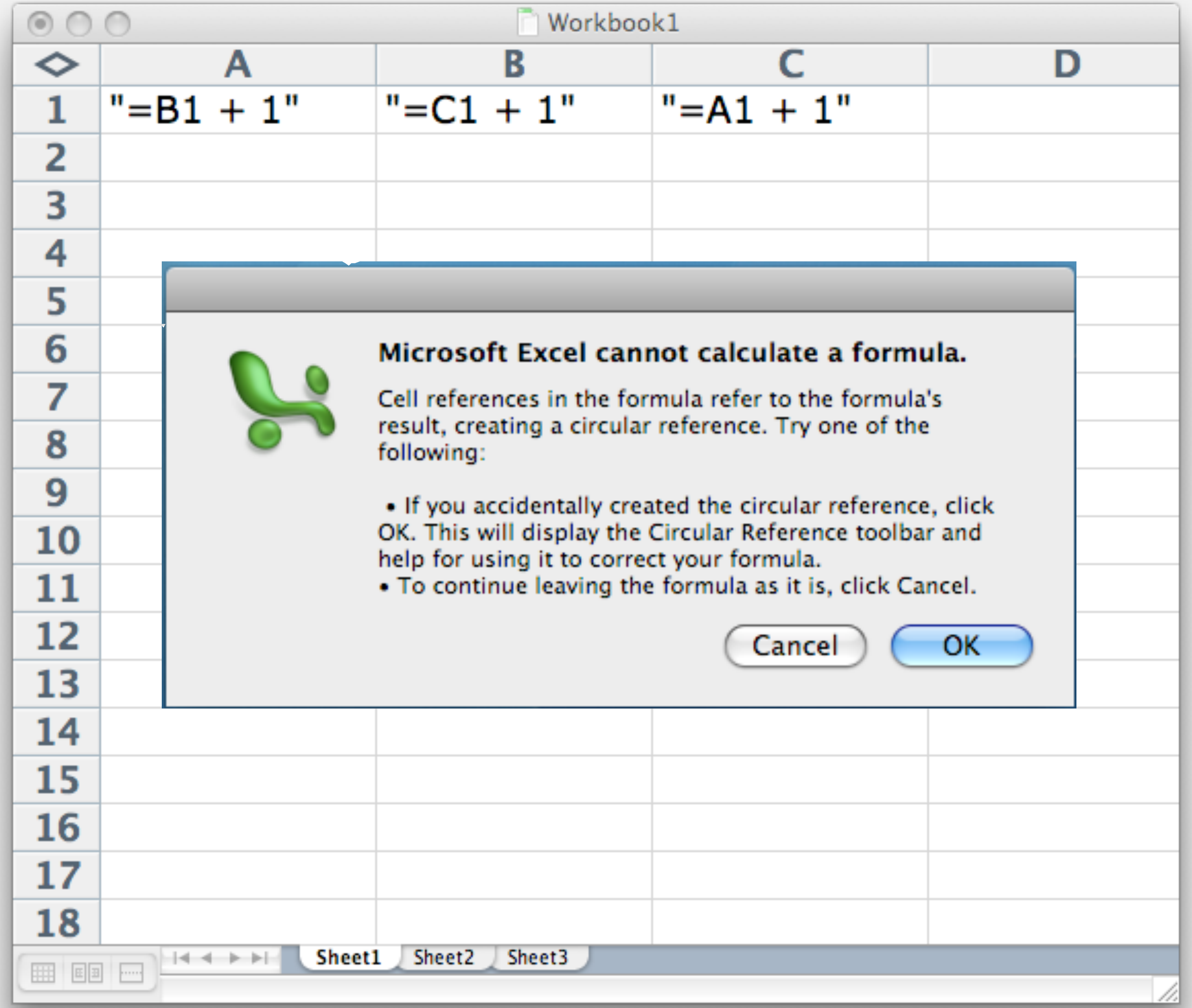
```
public class B extends C
{
   ...
}
```

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

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

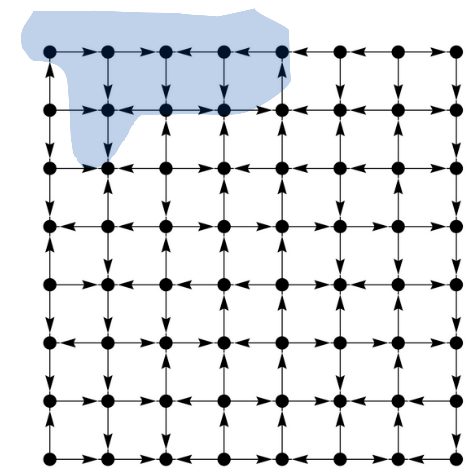
# Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection.



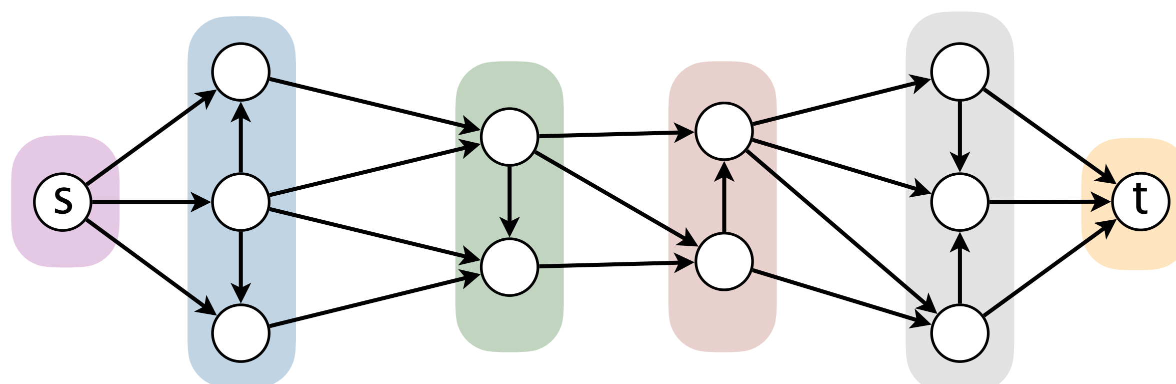
# Digraph-processing summary: algorithms of the day

**single-source reachability in a digraph**



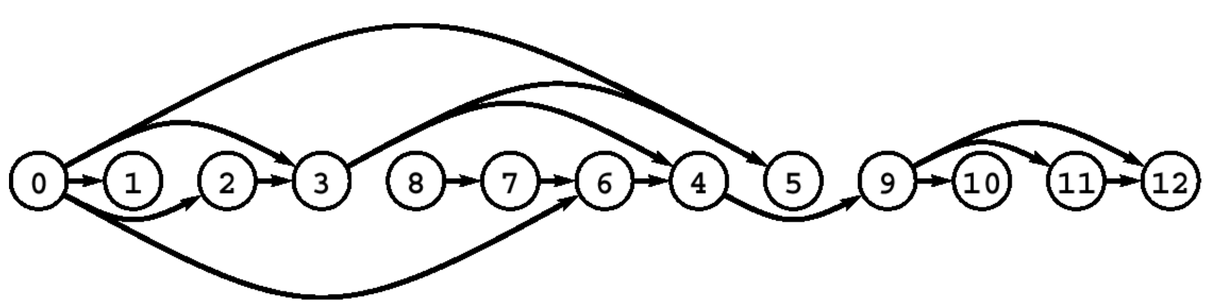
DFS/BFS

**shortest path in a digraph**




BFS

**topological sort in a DAG**




DFS