http://algs4.cs.princeton.edu

4.2 DIRECTED GRAPHS

- *introduction*
- *digraph API*
- *digraph search*
- *topological sort*
- *strong components*

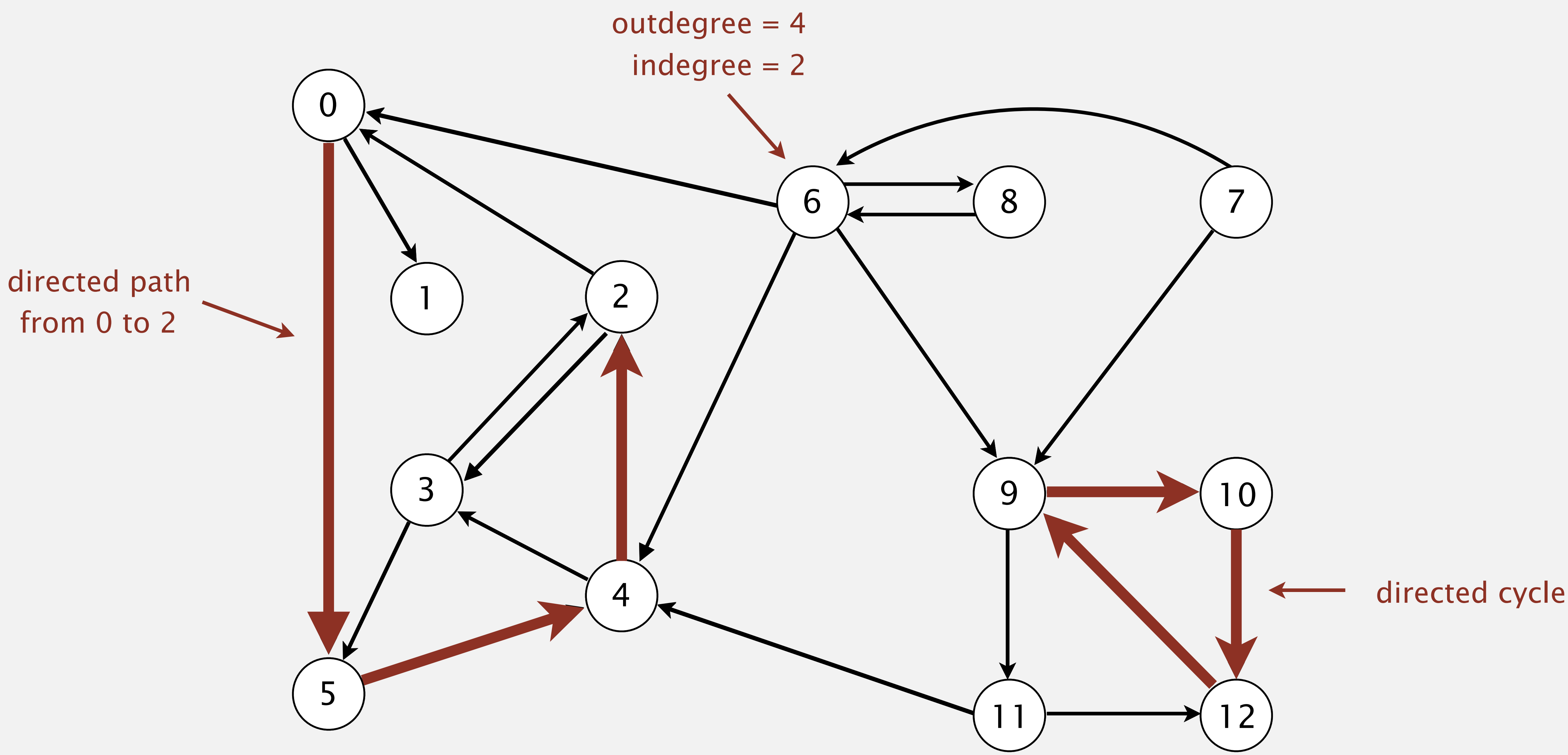
http://algs4.cs.princeton.edu

4.2 Directed Graphs

- *introduction*
- *digraph API*
- *digraph search*
- *topological sort*
- *strong components*

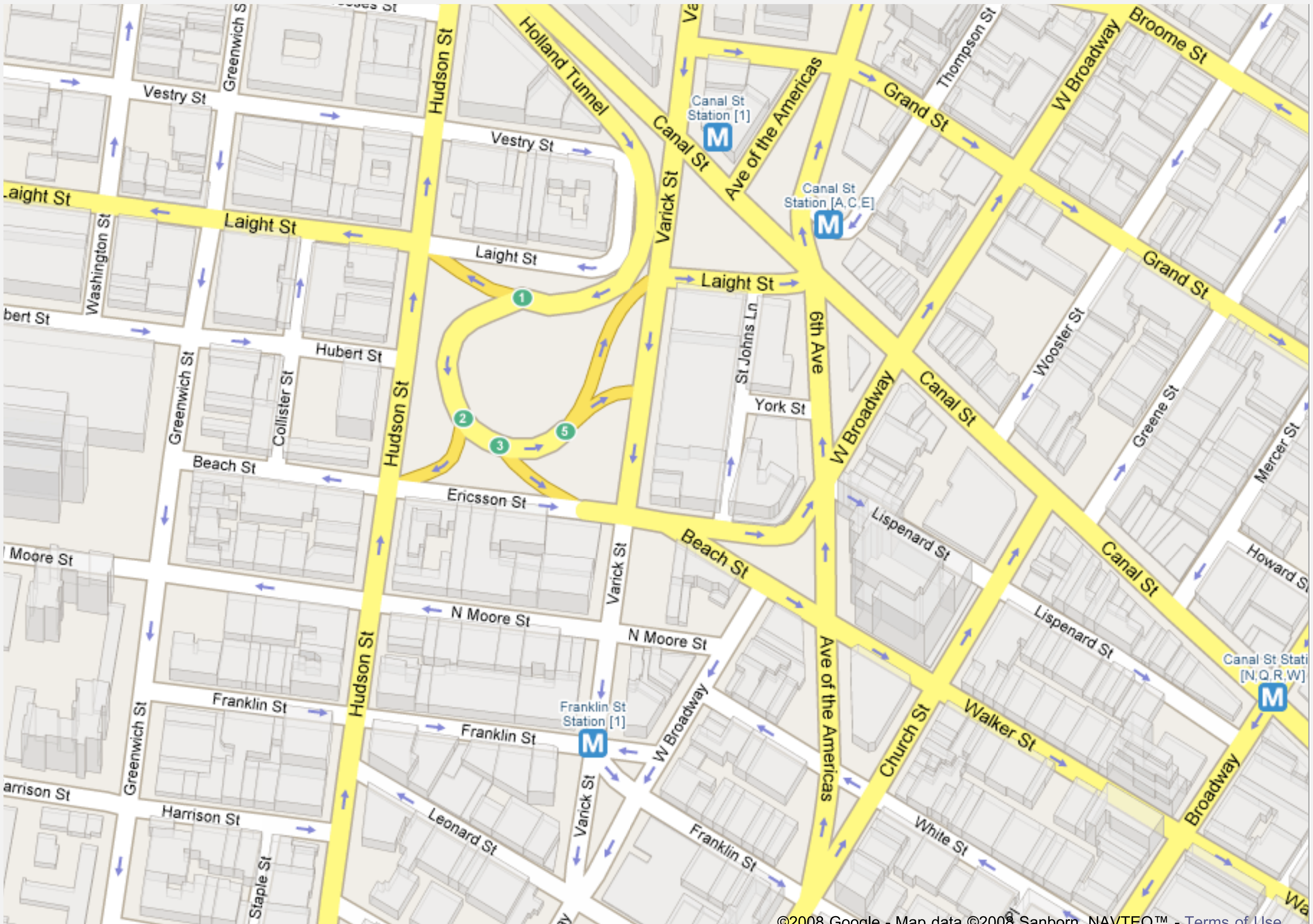
Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.



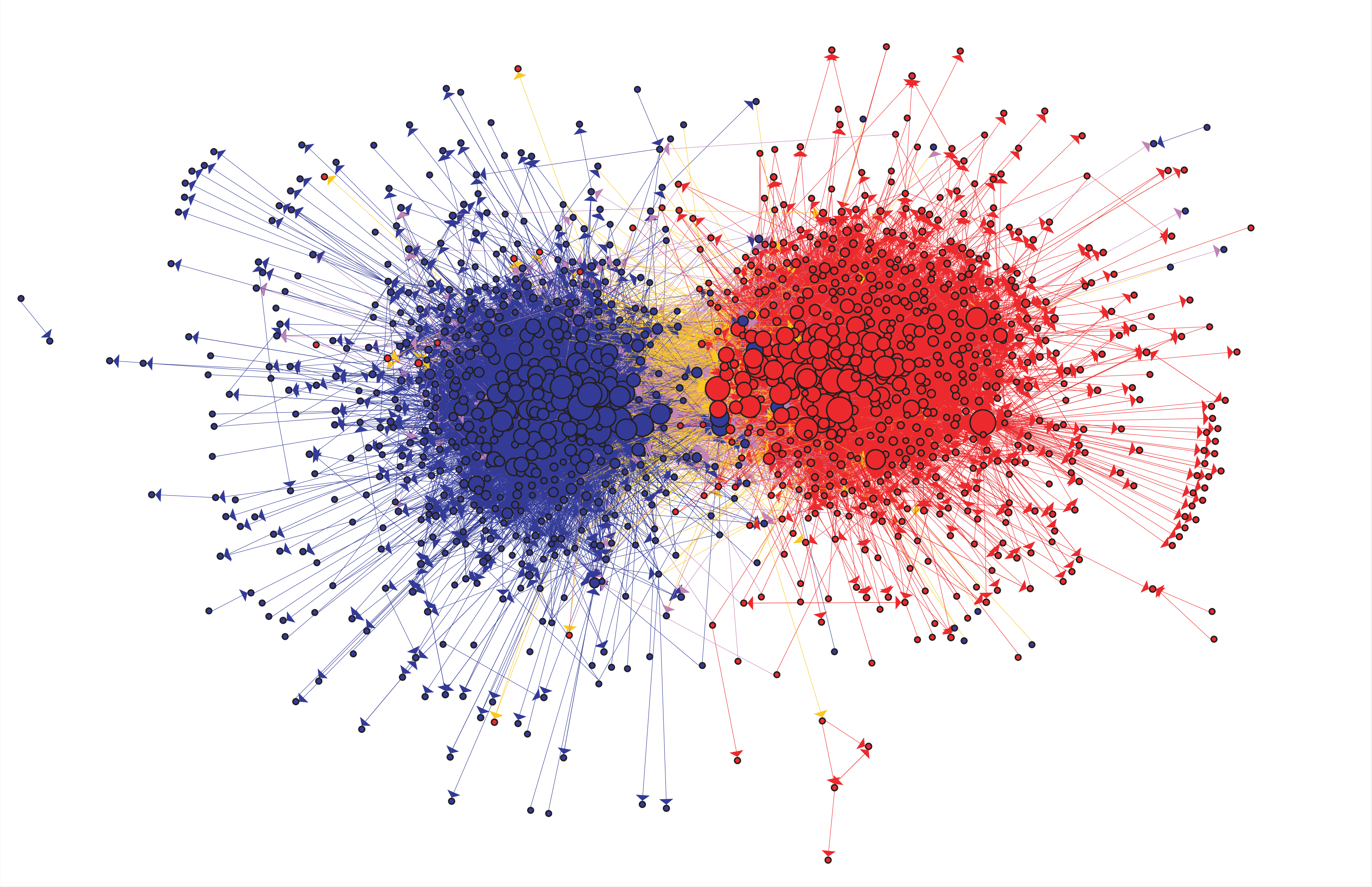
Road network

Vertex = intersection; edge = one-way street.



Political blogosphere graph

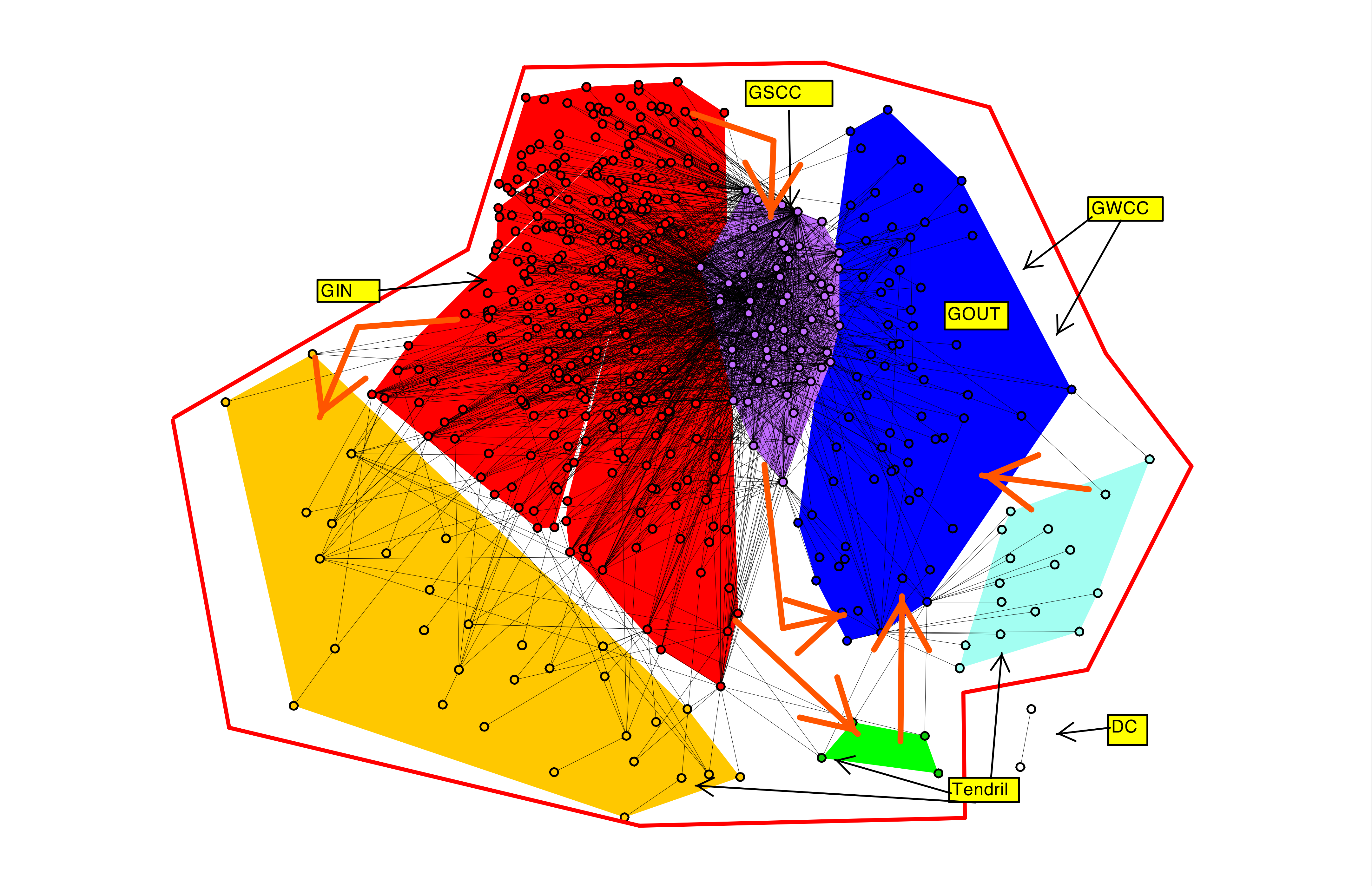
Vertex = political blog; edge = link.



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

Overnight interbank loan graph

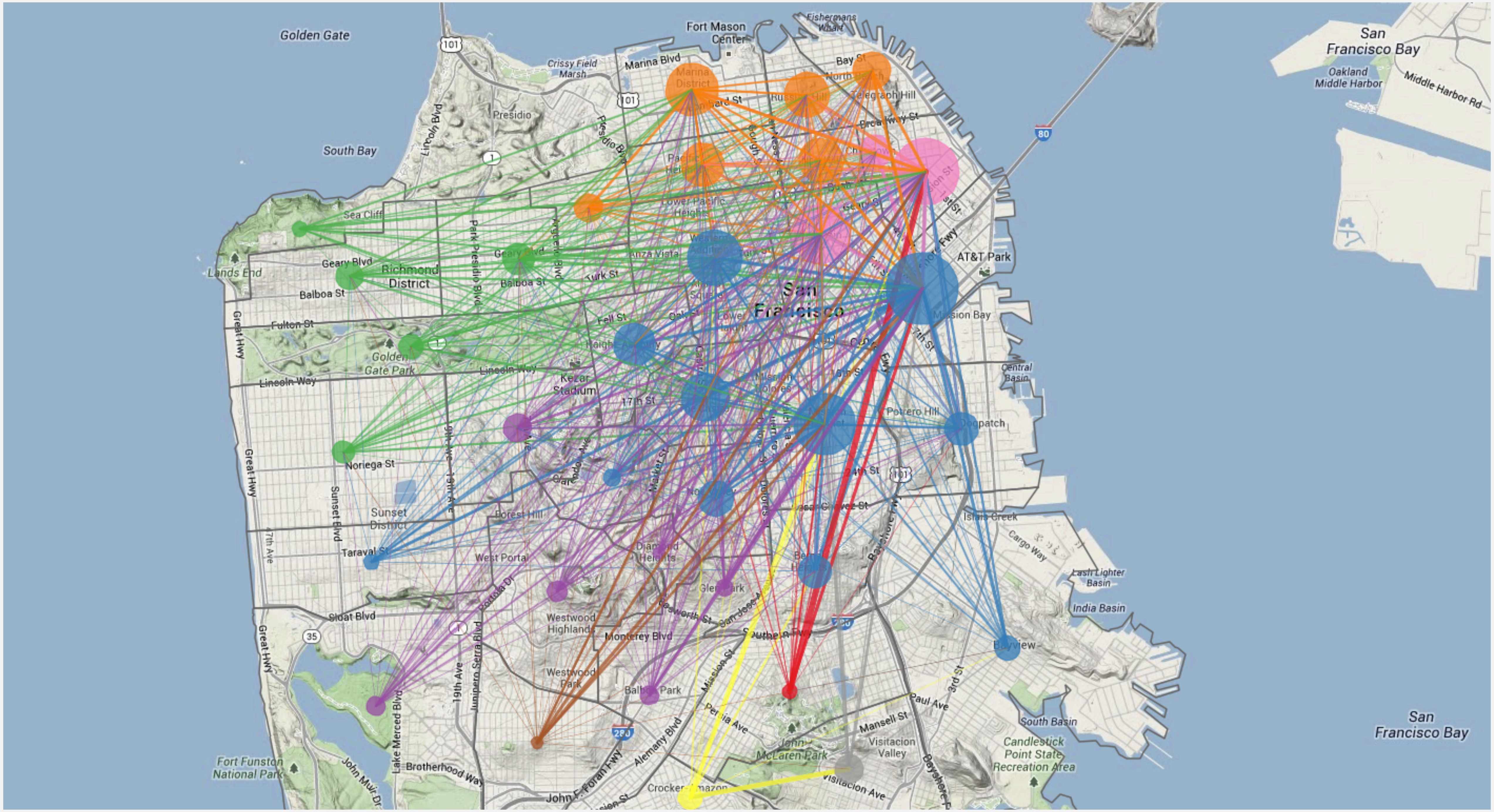
Vertex = bank; edge = overnight loan.



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

Uber taxi graph

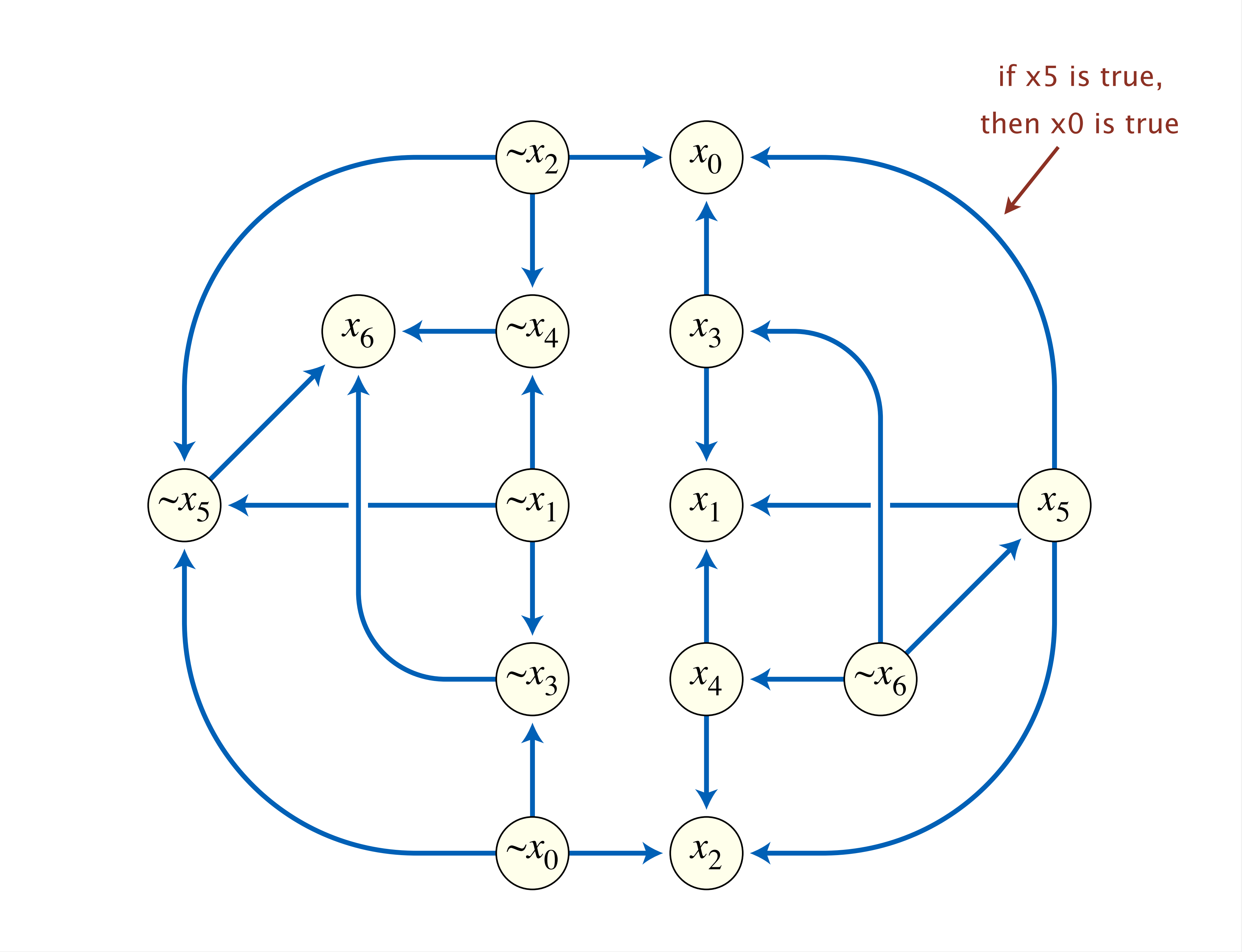
Vertex = taxi pickup; edge = taxi ride.



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

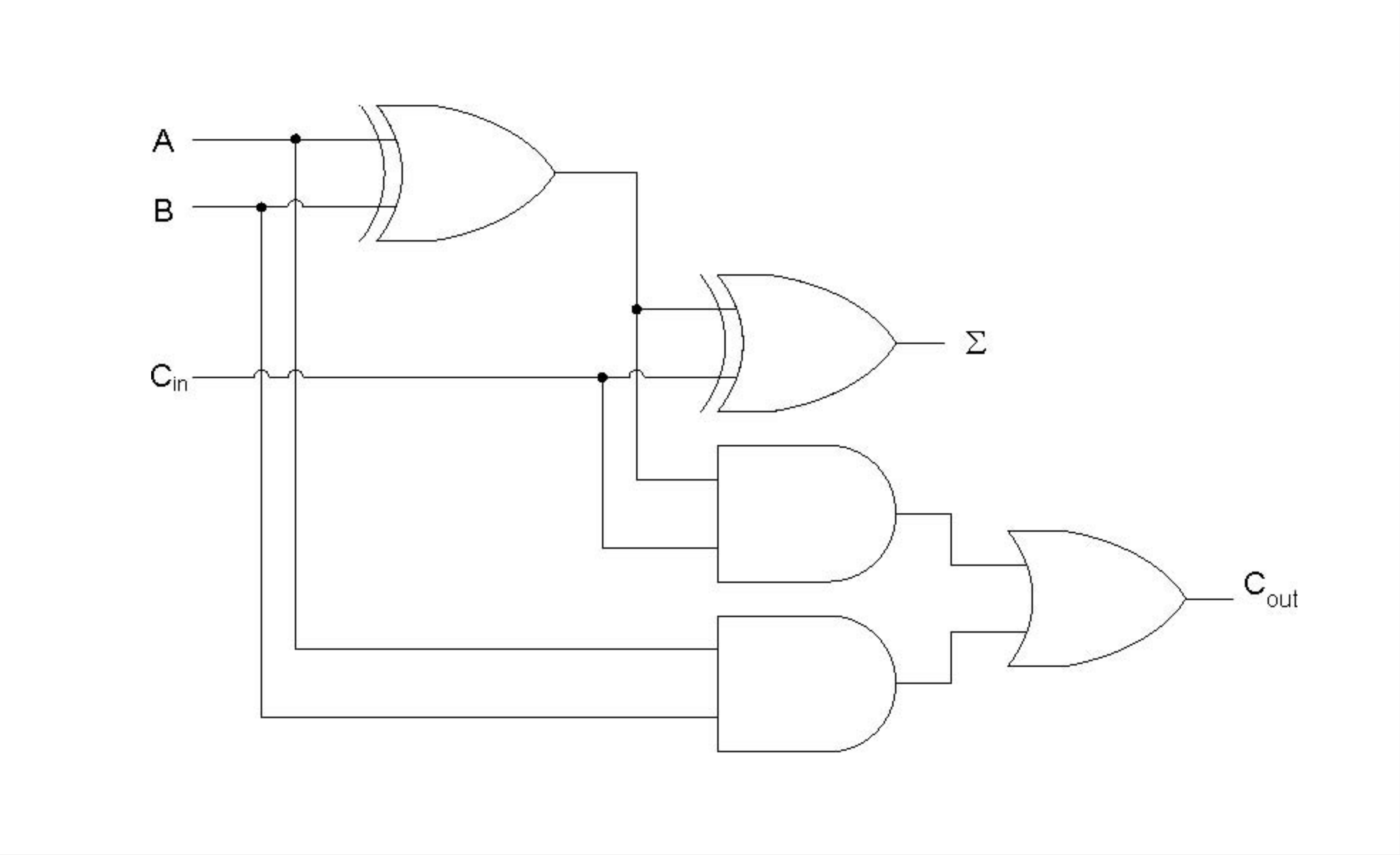
Implication graph

Vertex = variable; edge = logical implication.



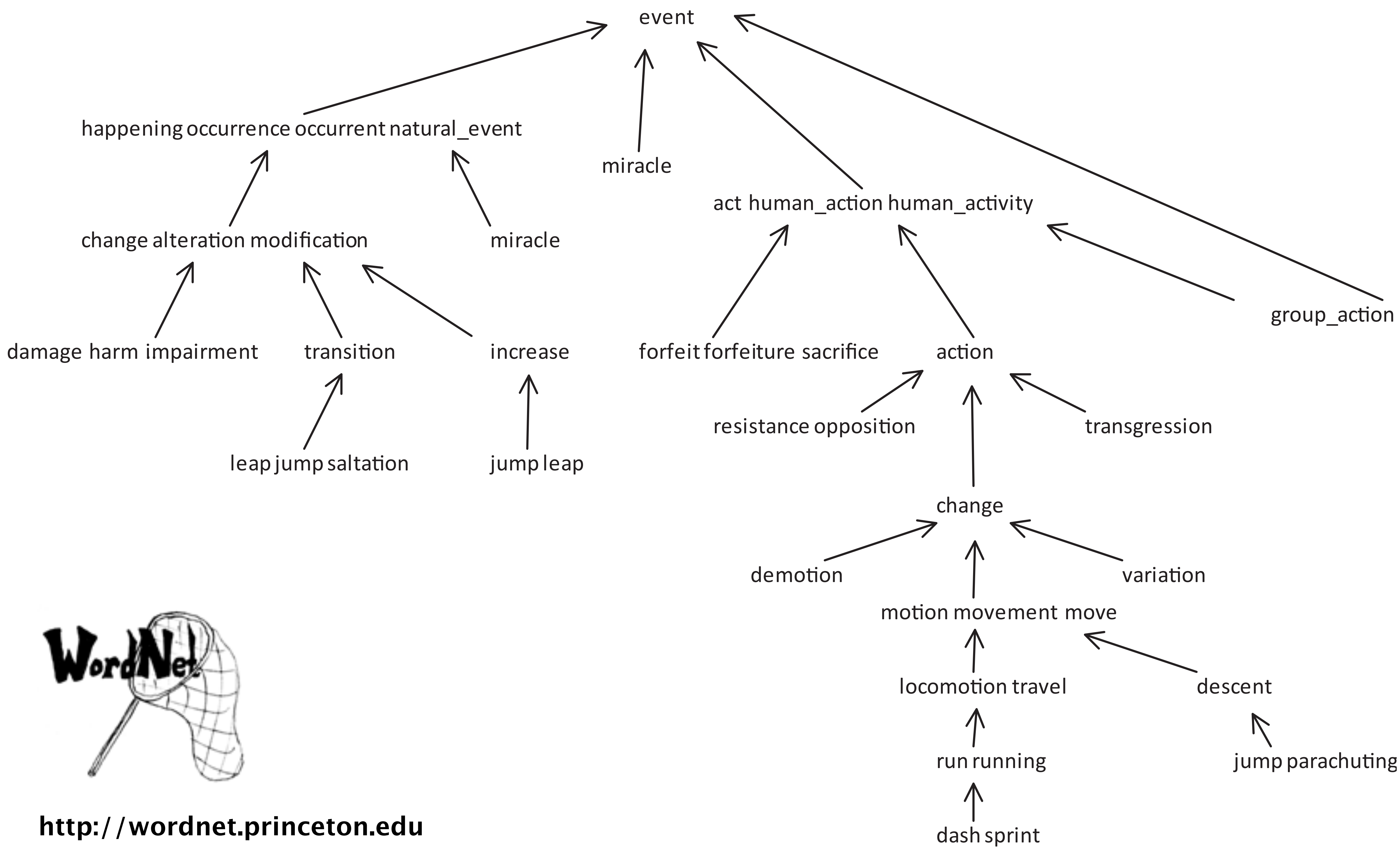
Combinational circuit

Vertex = logical gate; edge = wire.

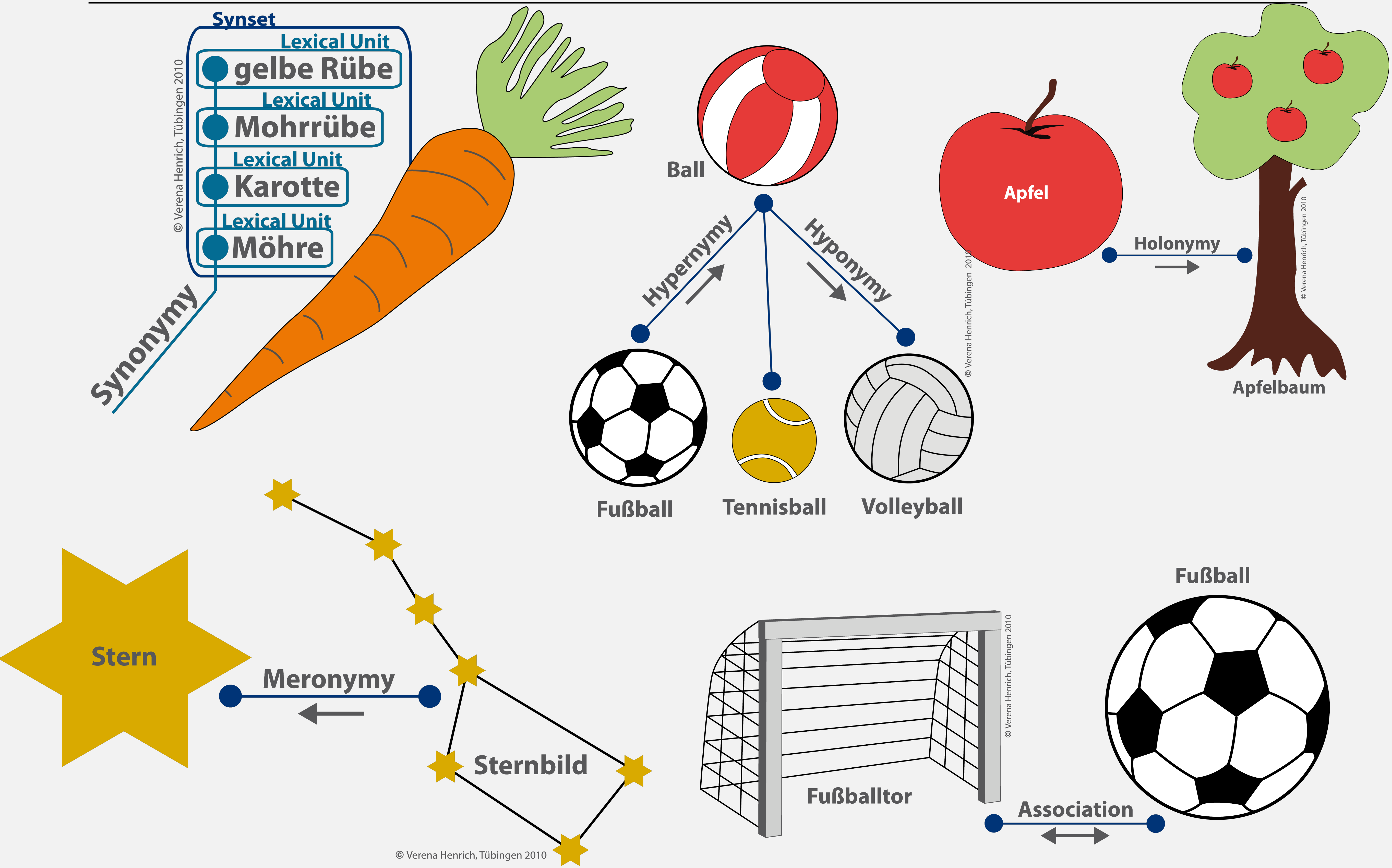


WordNet graph

Vertex = synset; edge = hypernym relationship.



http://wordnet.princeton.edu



Synset
Lexical Unit
gelbe Rübe
Lexical Unit
Mohrrübe
Lexical Unit
Karotte
Lexical Unit
Möhre
Synonymy
© Verena Henrich, Tübingen 2010
Ball
Hypernymy
Hyponymy
Fußball
Tennisball
Volleyball
© Verena Henrich, Tübingen 2010
Apfel
Holonymy
Apfelbaum
© Verena Henrich, Tübingen 2010
Stern
Meronymy
Sternbild
© Verena Henrich, Tübingen 2010
Fußballtor
© Verena Henrich, Tübingen 2010
Association
Fußball

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

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 all 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*
- *digraph search*
- *topological sort*
- *strong components*

Digraph API

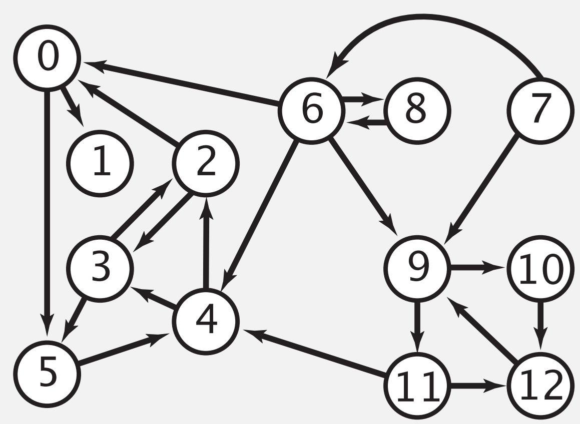
Almost identical to `Graph` API.

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 pointing from v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*
`Digraph`	`reverse()`	*reverse of this digraph*
`String`	`toString()`	*string representation*

Digraph API

tinyDG.txt

```
V -> 13
     22 <- E
  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
```

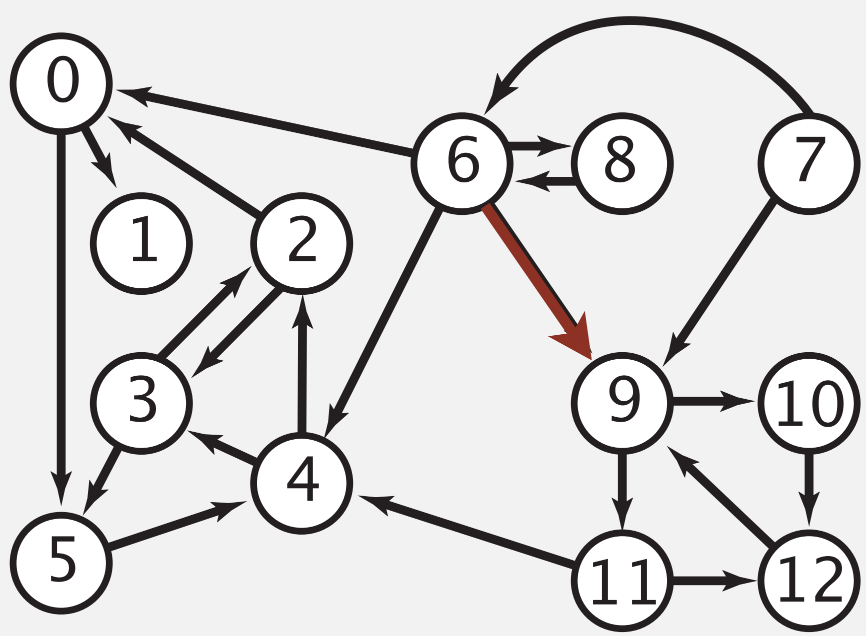
```
In in = new In(args[0]);
Digraph G = new Digraph(in);

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

read digraph from input stream

print out each edge (once)

Digraph representation: set of edges

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

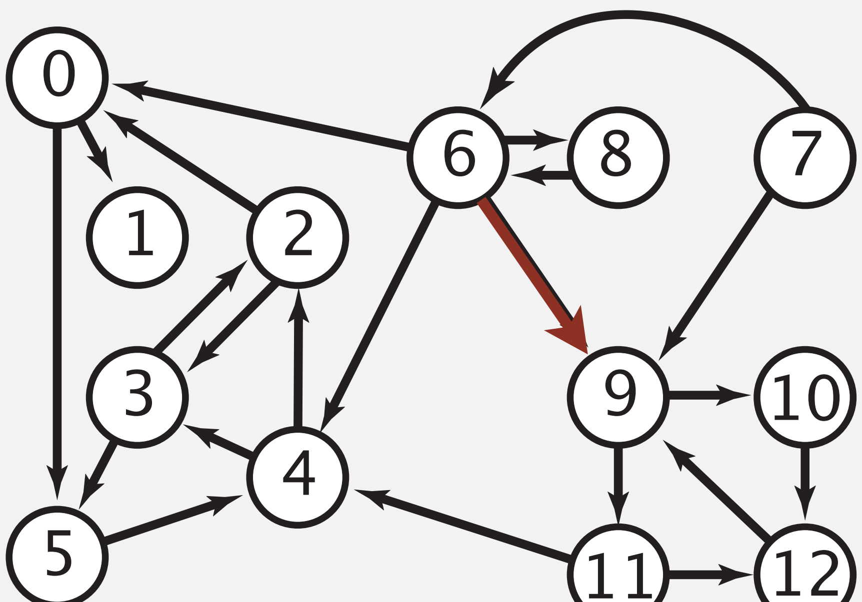


0	1
0	5
2	0
2	3
3	2
3	5
4	2
4	3
5	4
6	0
6	4
6	8
6	9
7	6
7	9
8	6
9	10
9	11
10	12
11	4
11	12
12	9

Digraph representation: adjacency matrix

Maintain a two-dimensional V-by-V boolean array;

for each edge $v \to w$ in the digraph: `adj[v][w] = true`.

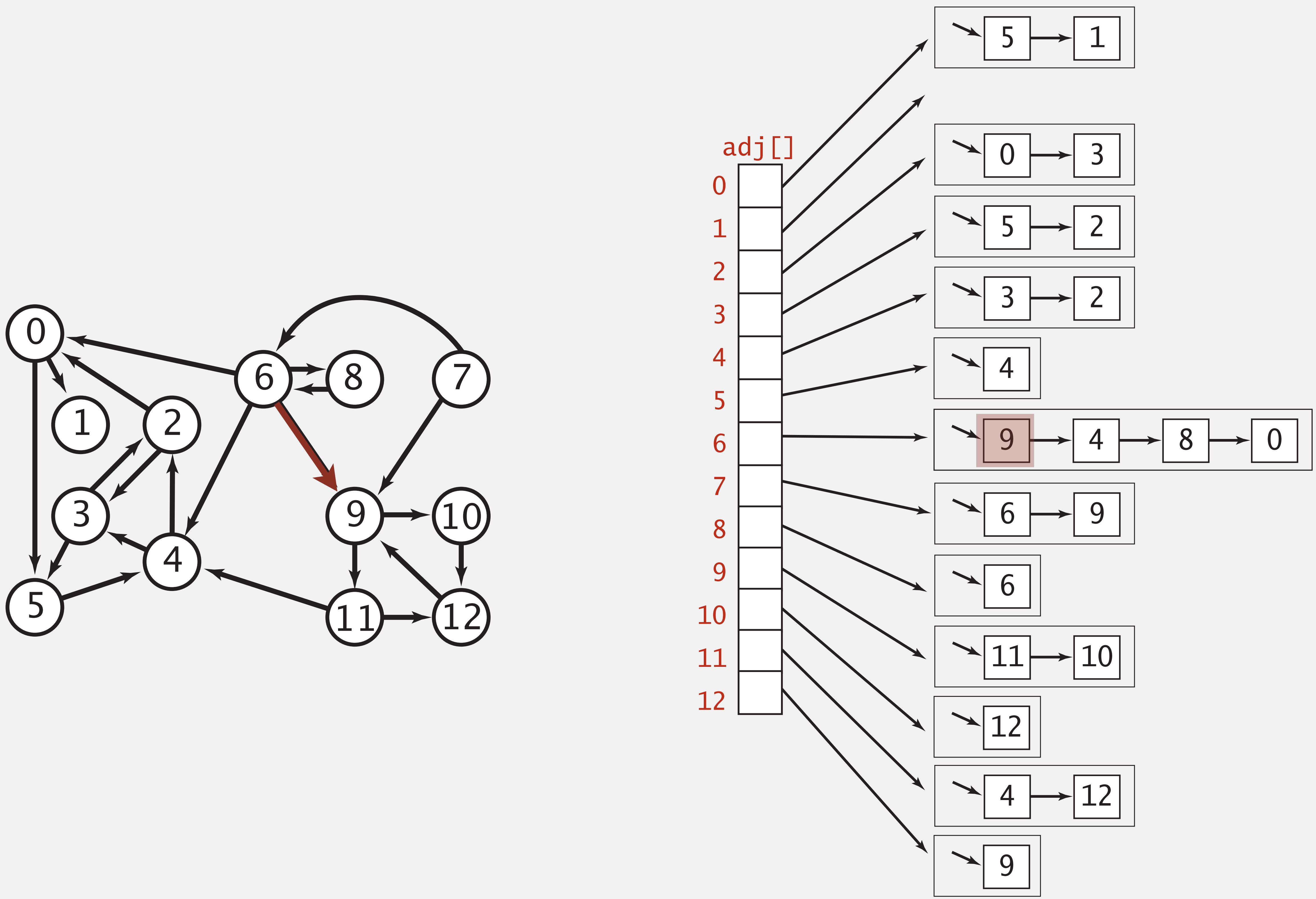


from \ to	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	1	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	1	0	0	0	0	0	0	0
4	0	0	1	1	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	1	1	0	0	0
7	0	0	0	0	0	0	1	0	0	1	0	0	0
8	0	0	0	0	0	0	1	0	0	0	0	0	0
9	0	0	0	0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

Note: parallel edges disallowed

Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.



Digraph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices pointing from v.
- Real-world digraphs tend to be sparse.

huge number of vertices,
small average vertex degree

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices pointing 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 final 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 final 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 pointing from v

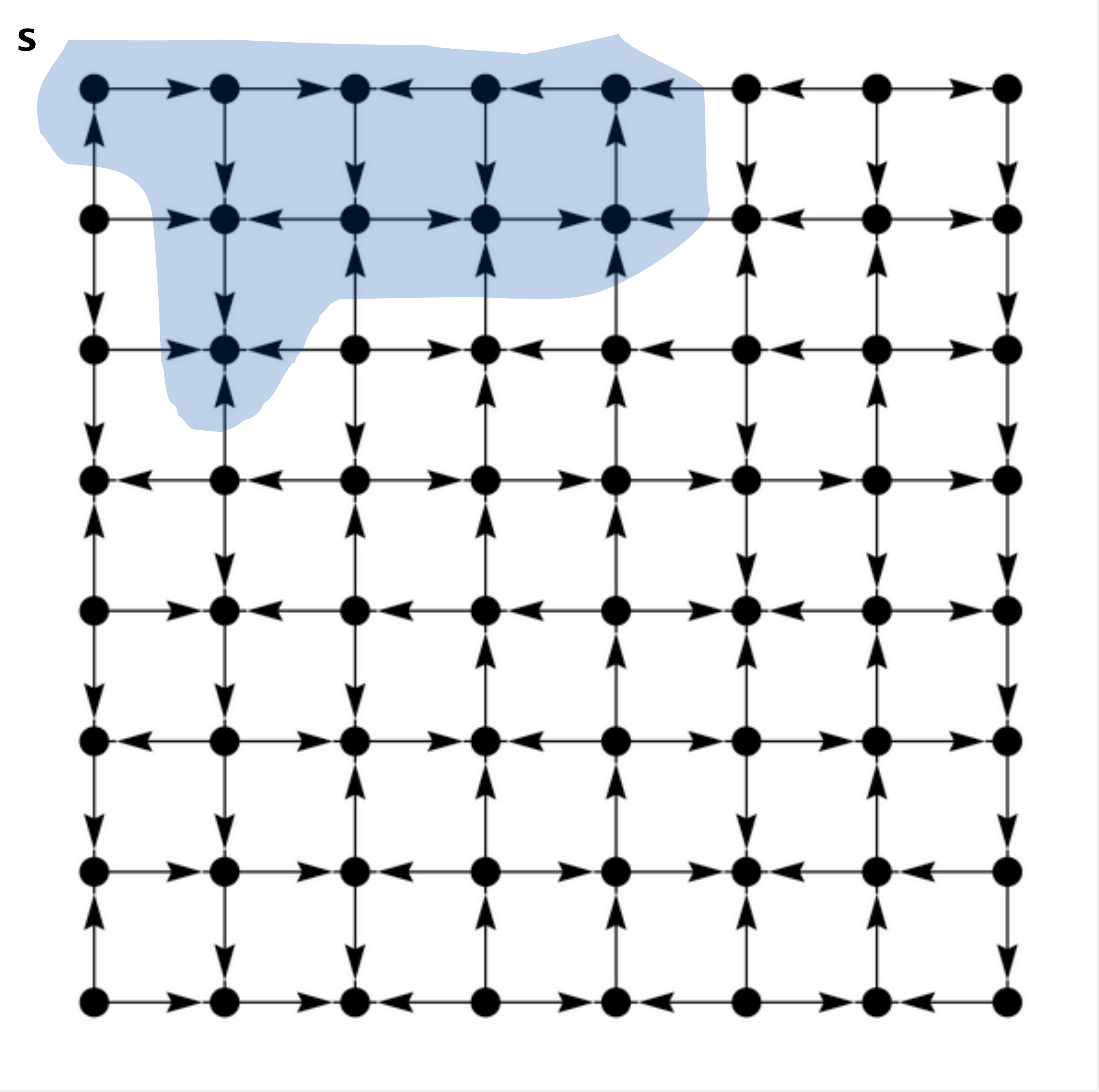
http://algs4.cs.princeton.edu

4.2 Directed Graphs

- *introduction*
- *digraph API*
- *digraph search*
- *topological sort*
- *strong components*

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)

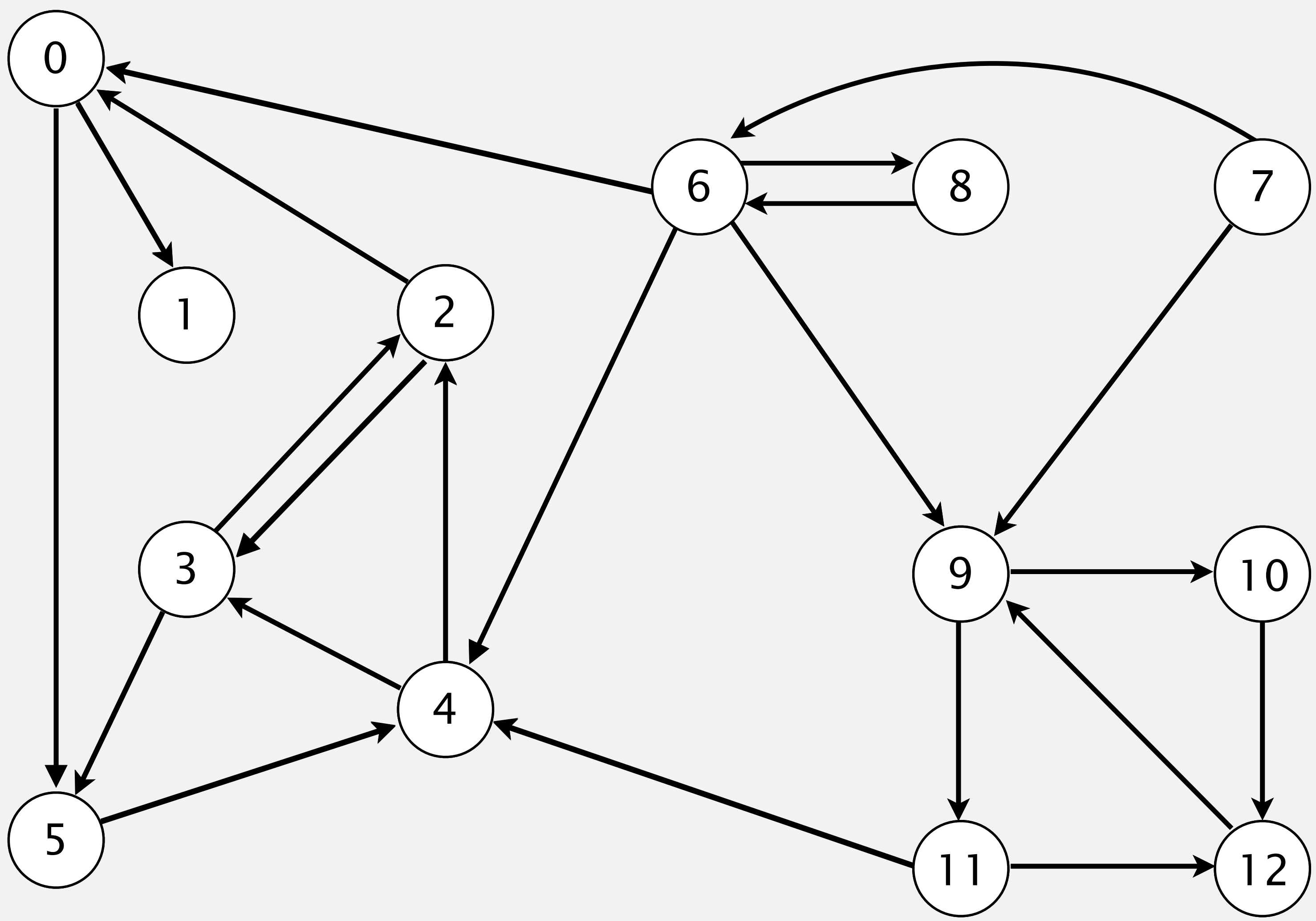
Mark v as visited.

Recursively visit all unmarked vertices w pointing from v.

Depth-first search demo

To visit a vertex v :

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



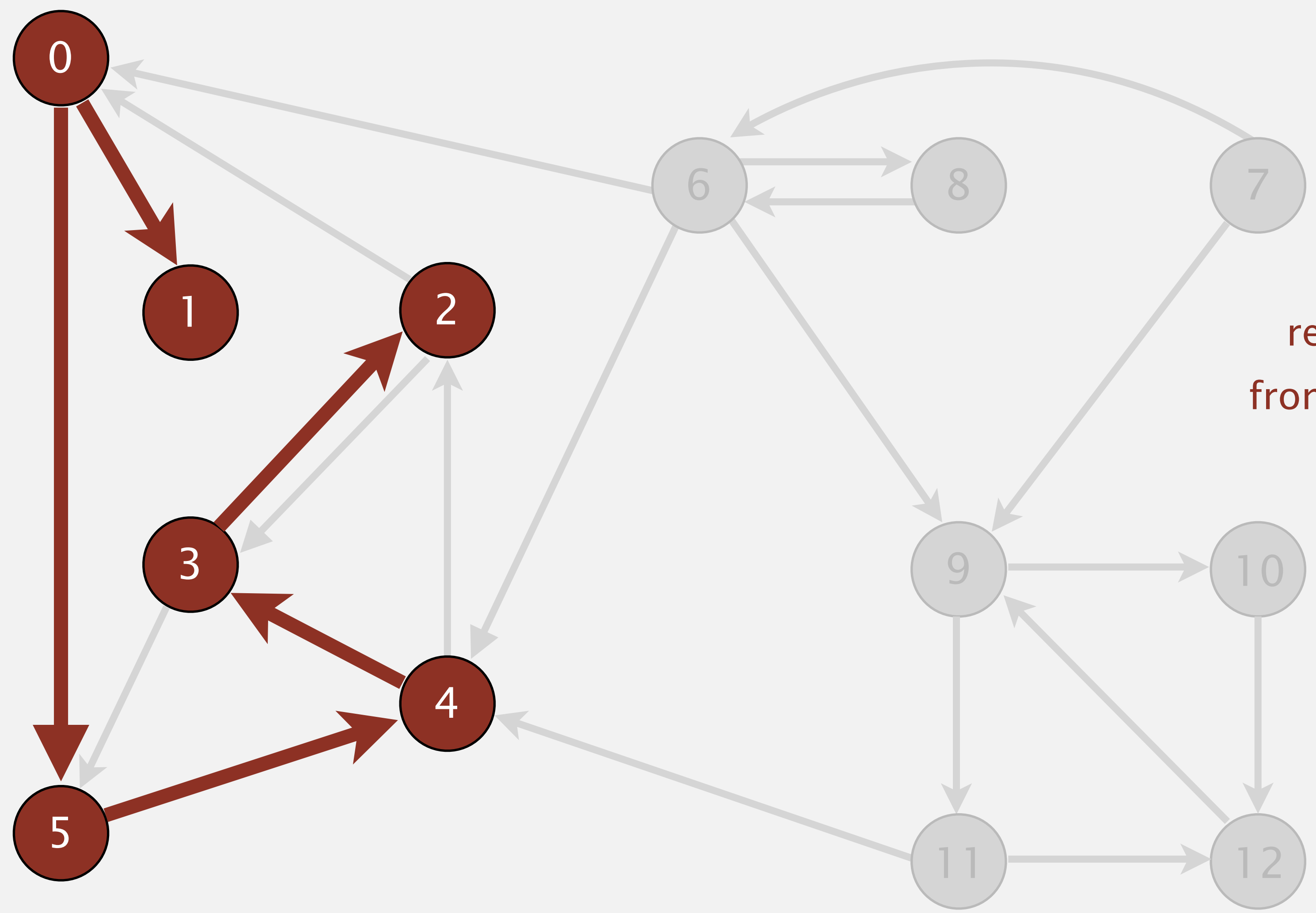
a directed graph

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 pointing from v.



reachable from 0

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	–

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];  }
}
```

true if path from s

constructor marks vertices reachable from s

recursive DFS does the work

client can ask whether any vertex is reachable from s

Reachability application: program control-flow analysis

Every program is a digraph.

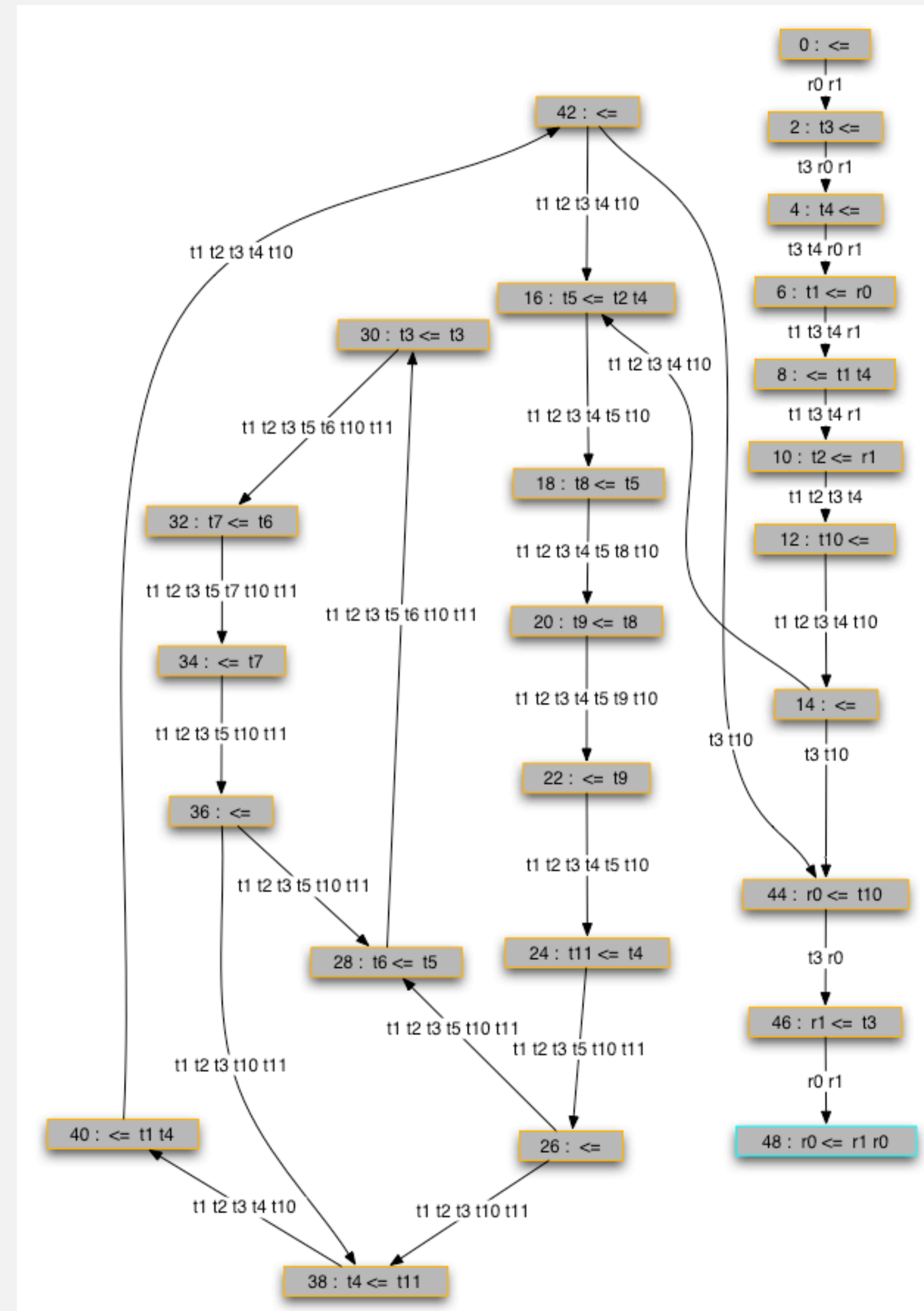
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



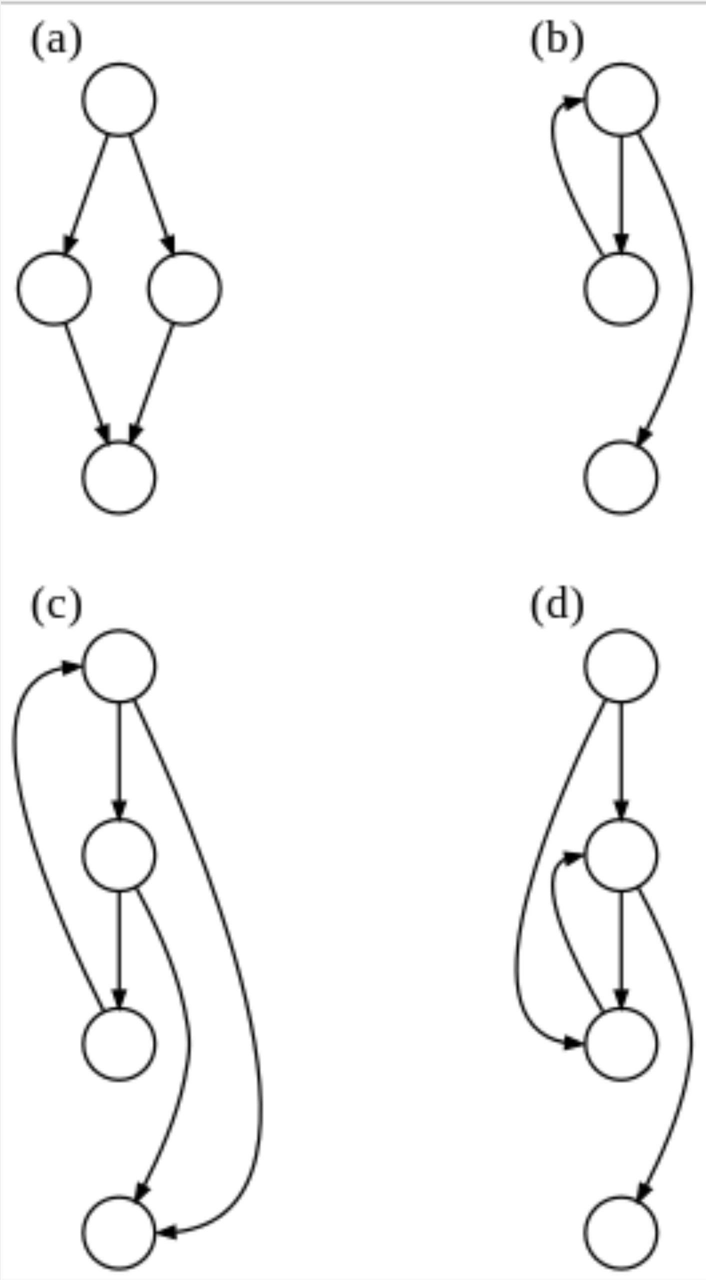
Control flow graphs (examples)

(a) an if-then-else

(b) a while loop

(c) a natural loop with two exits,
e.g. while with an if...break in the middle;

(d) an irreducible CFG:
a loop with two entry points,
e.g. goto into a while or for loop



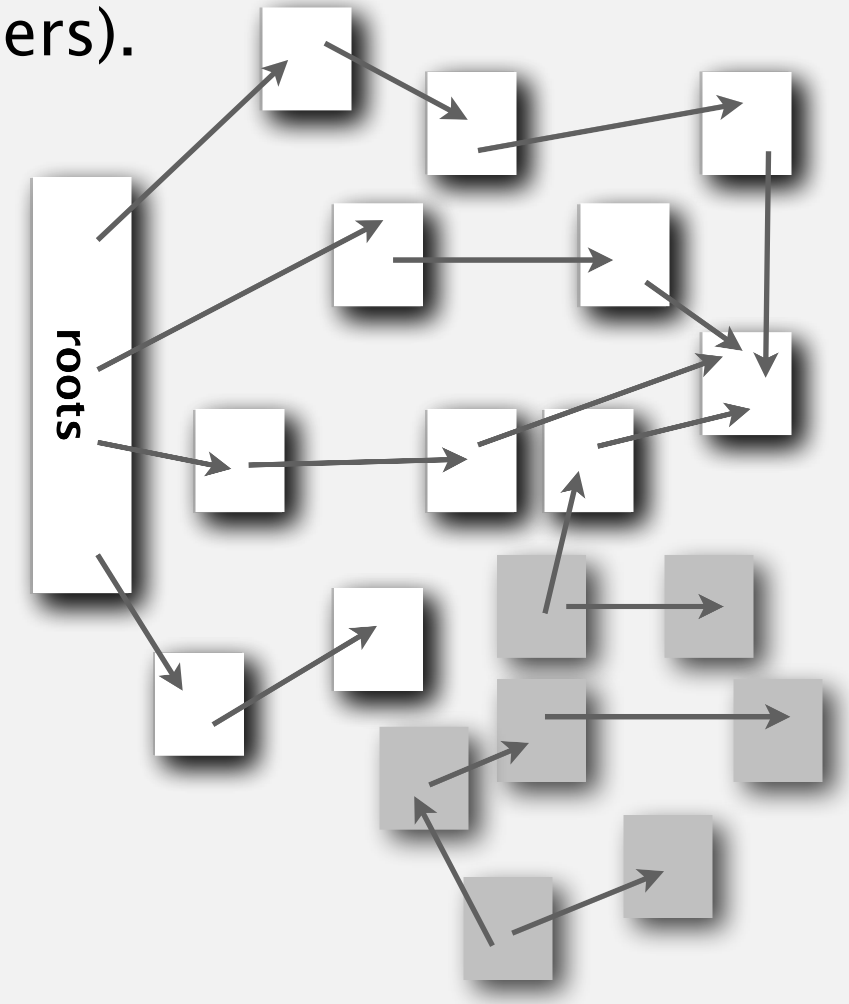
Reachability application: mark-sweep garbage collector

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).

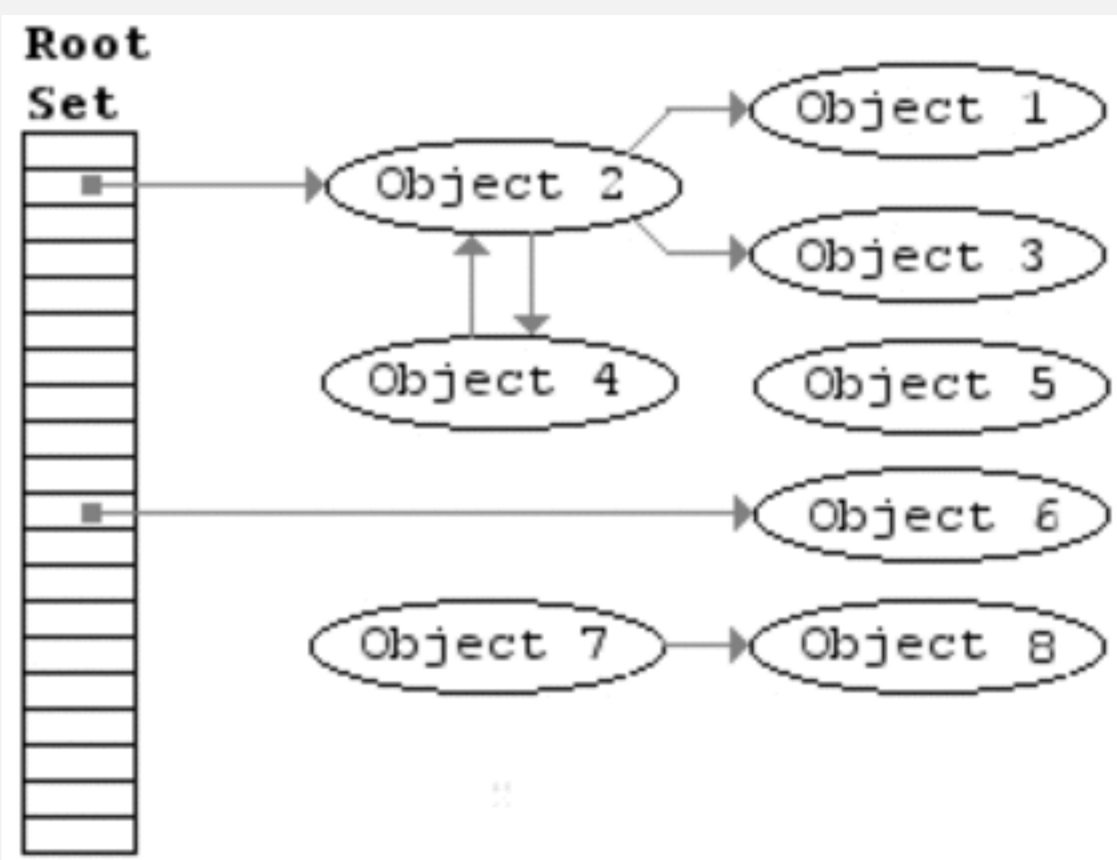


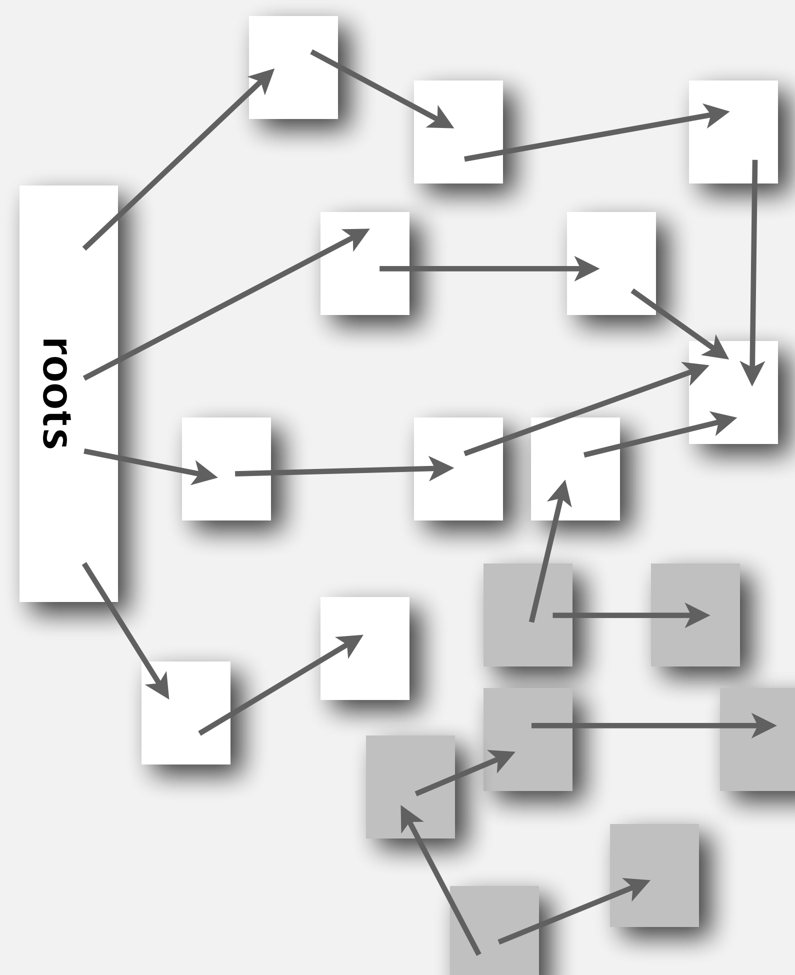
Reachability application: mark-sweep garbage collector

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.

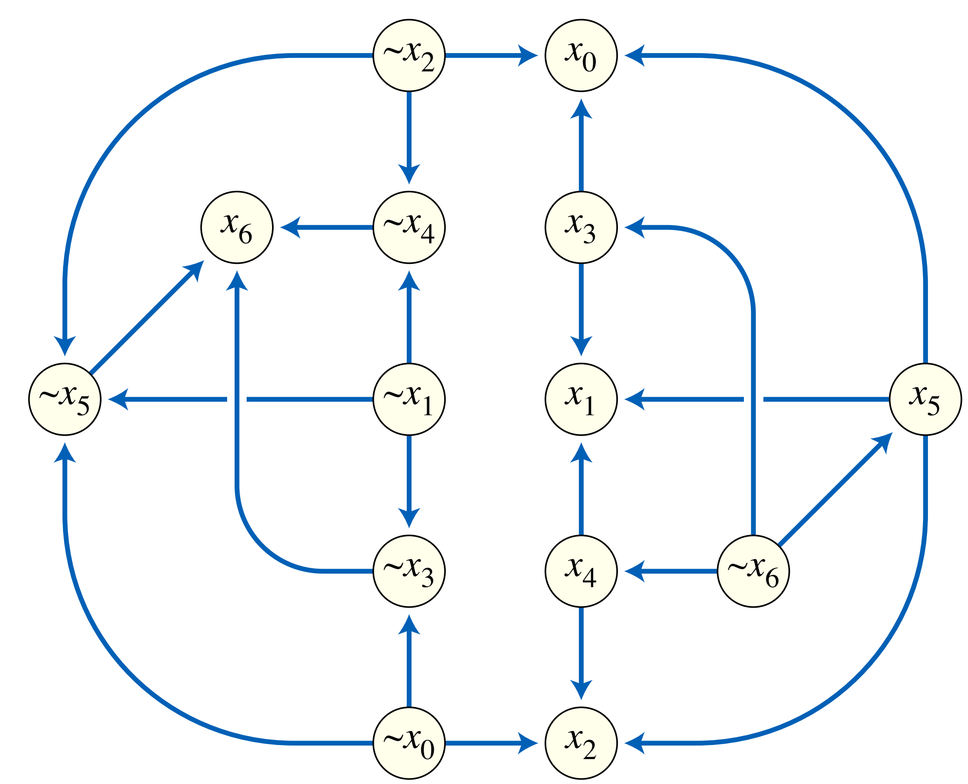
- 2-satisfiability.
- Directed Euler path.
- Strongly-connected components.

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.



$$(x_0 \vee x_2) \wedge (x_0 \vee \neg x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge (x_2 \vee \neg x_4) \wedge (x_0 \vee \neg x_5) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_5) \wedge (x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6).$$

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:

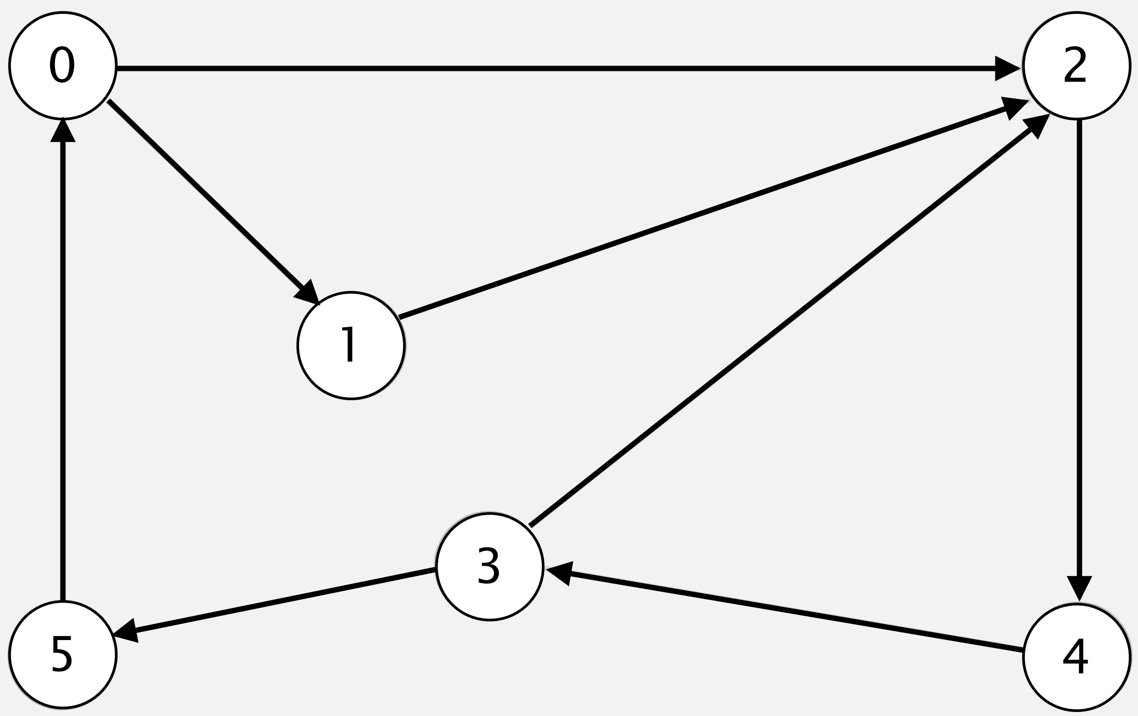
- **remove the least recently added vertex v**
- **for each unmarked vertex pointing from v:**
  add to queue and mark as visited.

Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E + V$.

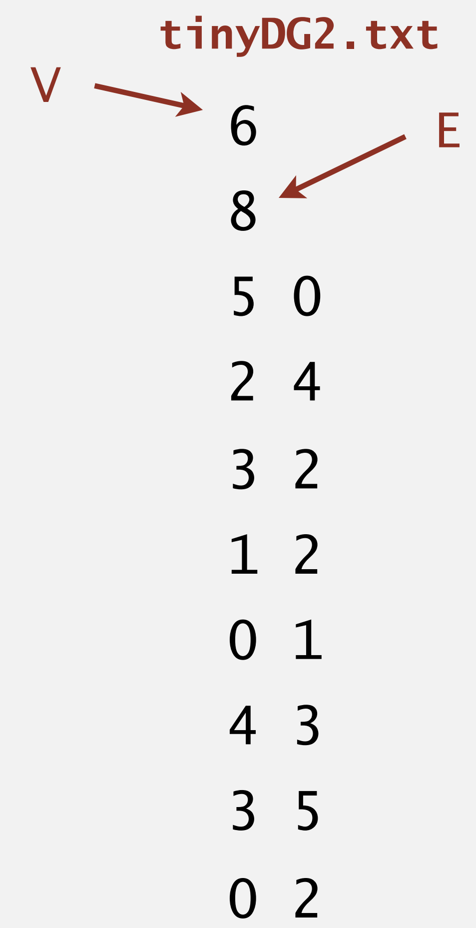
Directed breadth-first search demo

Repeat until queue is empty:

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



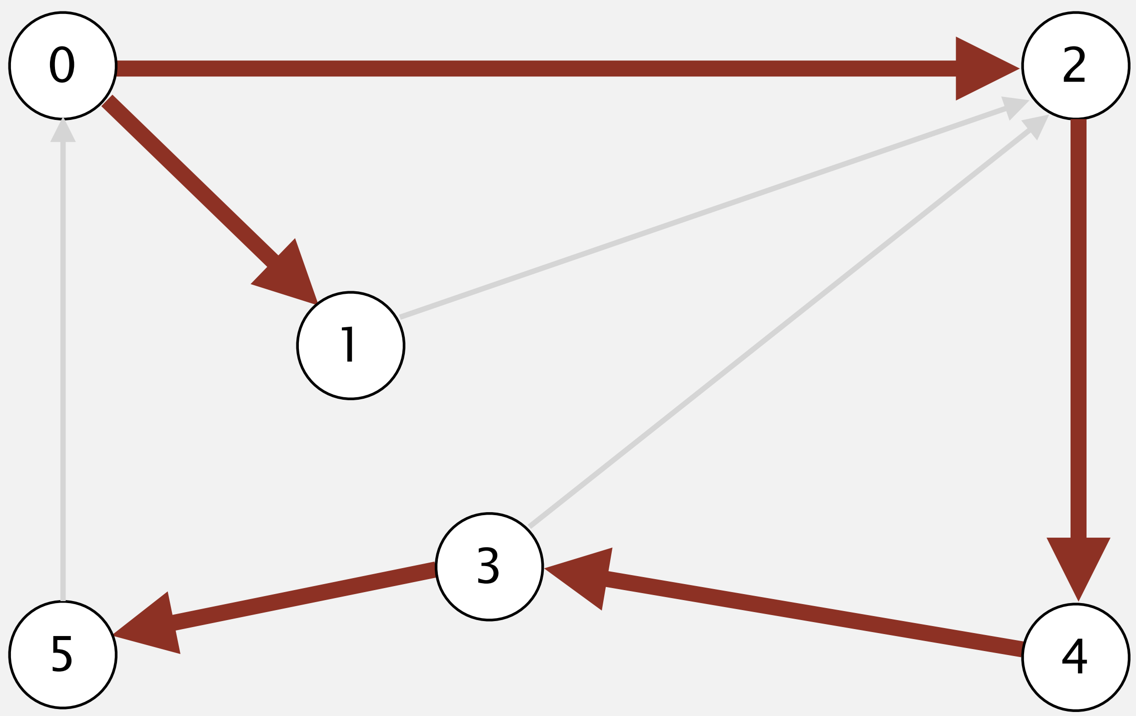
graph G



Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing 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

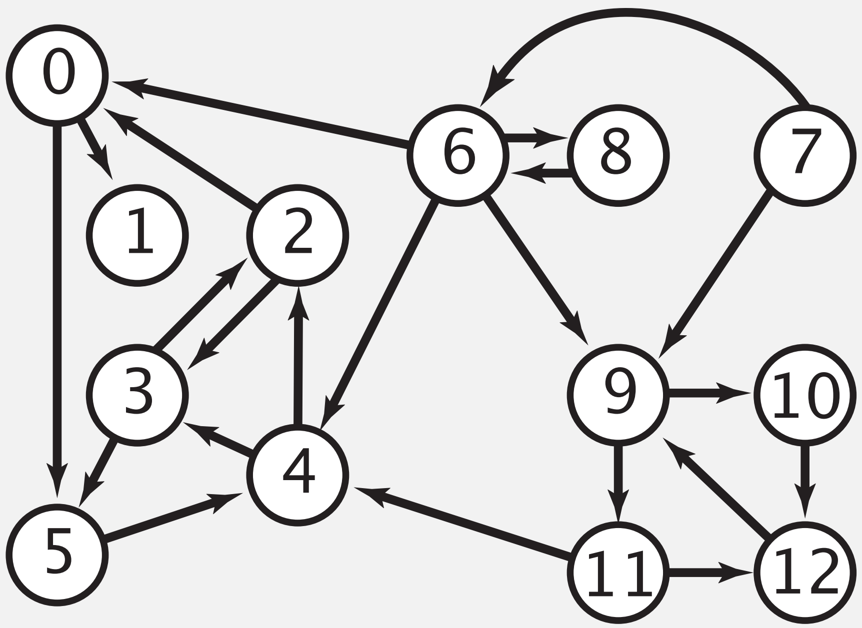
done

Multiple-source shortest paths

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

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

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



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

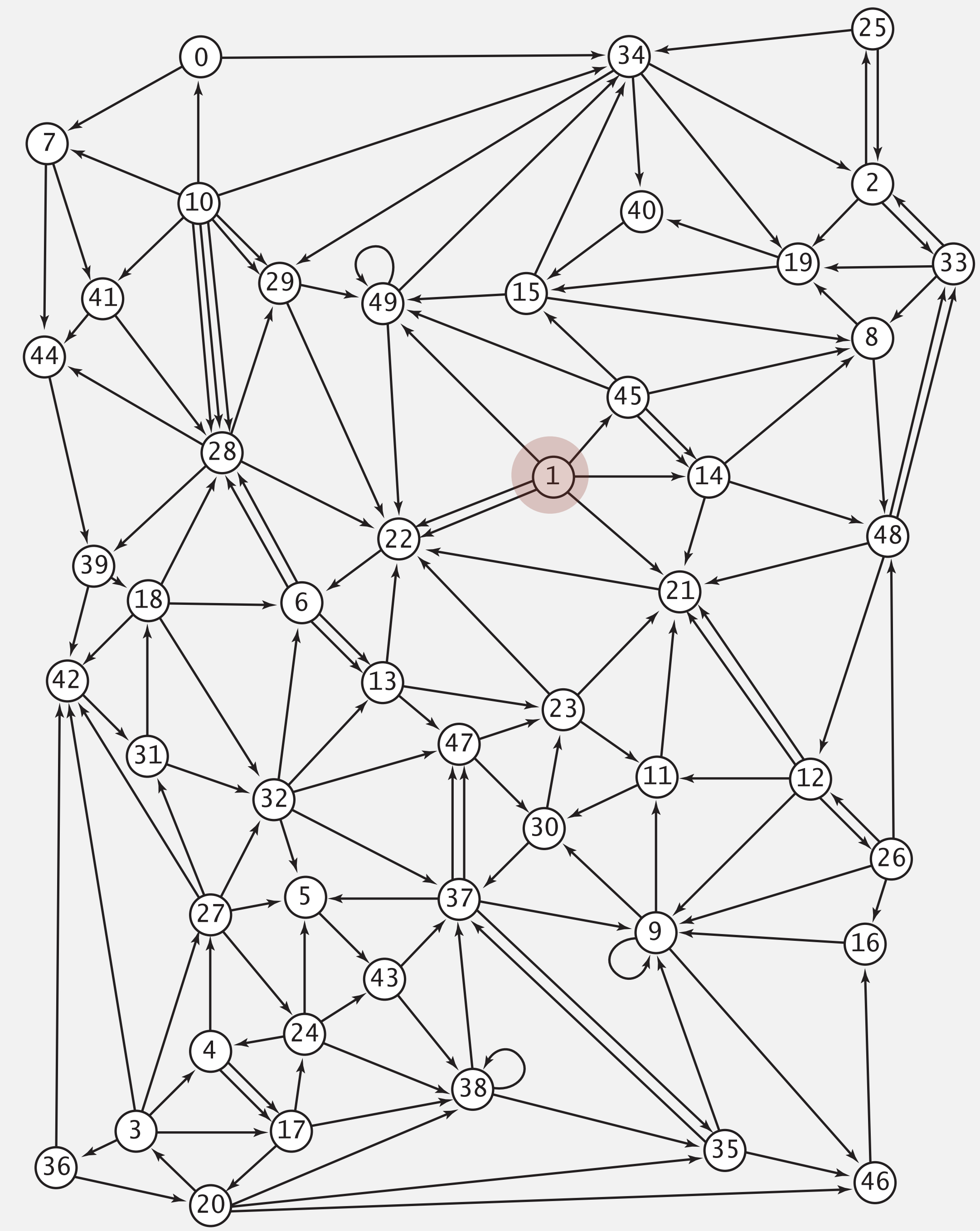
A. Use BFS, but initialize by enqueuing all source vertices.

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 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?

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);
         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 it and put on the queue

Web crawler output

BFS crawl

```
http://www.princeton.edu
http://www.w3.org
http://ogp.me
http://giving.princeton.edu
http://www.princetonartmuseum.org
http://www.goprincetontigers.com
http://library.princeton.edu
http://helpdesk.princeton.edu
http://tigernet.princeton.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.princeton.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://agoogleaday.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
...
```

http://algs4.cs.princeton.edu

4.2 Directed Graphs

- *introduction*
- *digraph API*
- *digraph search*
- *topological sort*
- *strong components*

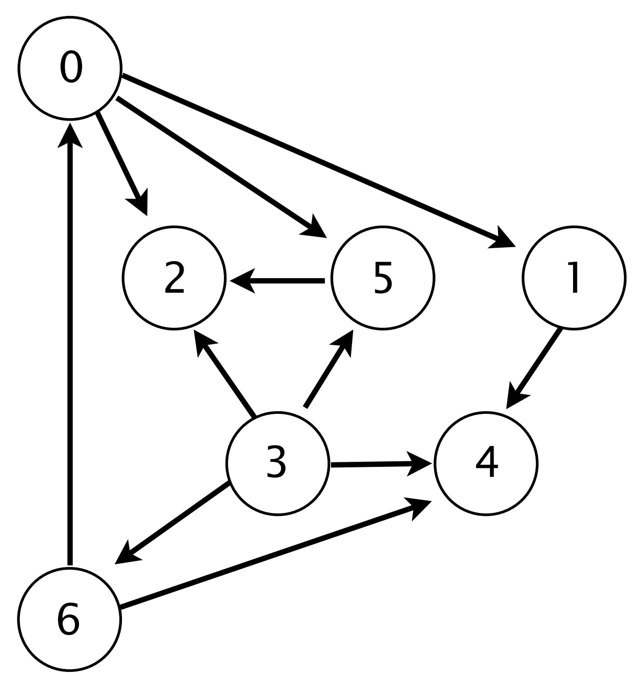
Precedence scheduling

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

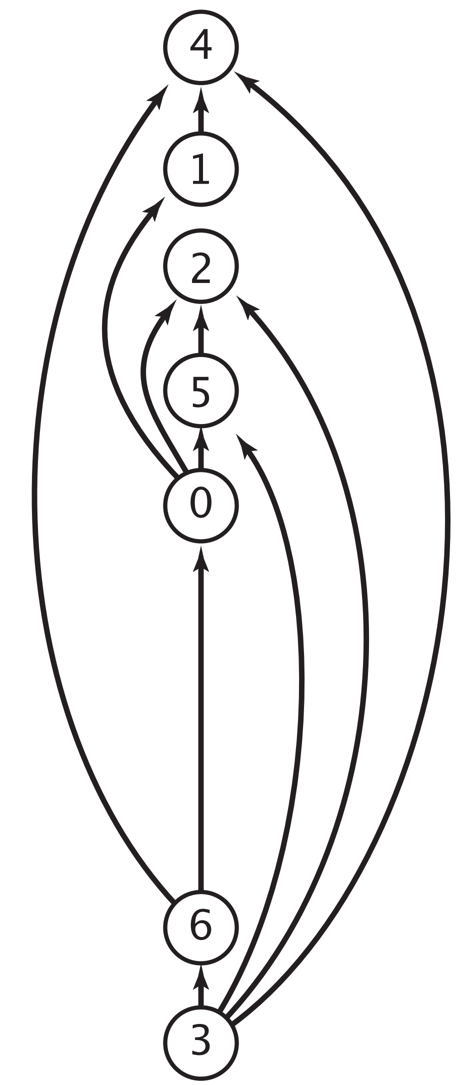
Digraph model. vertex = task; edge = precedence constraint.

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

tasks



precedence constraint graph



feasible schedule

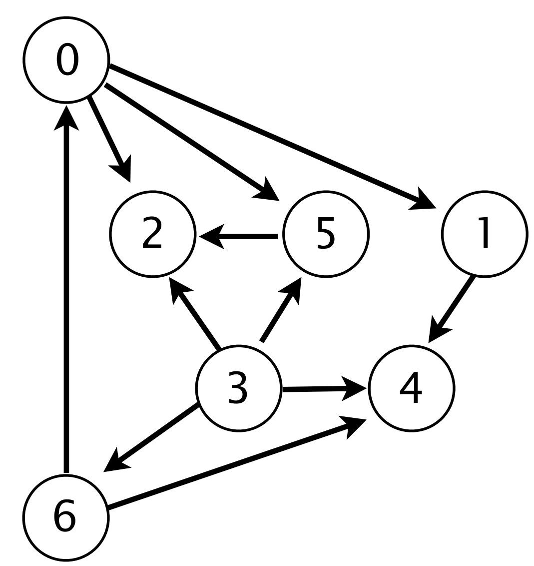
Topological sort

DAG. Directed acyclic graph.

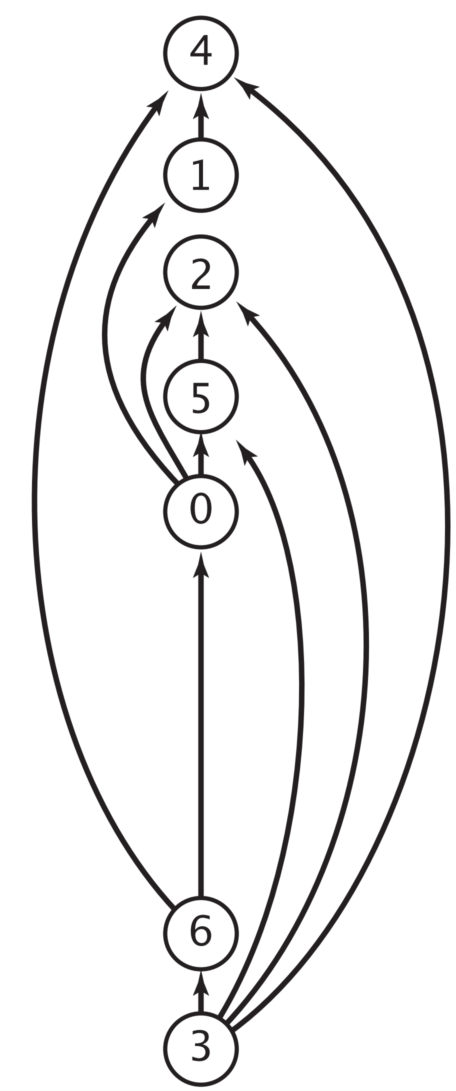
Topological sort. Redraw DAG so all edges point upwards.

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

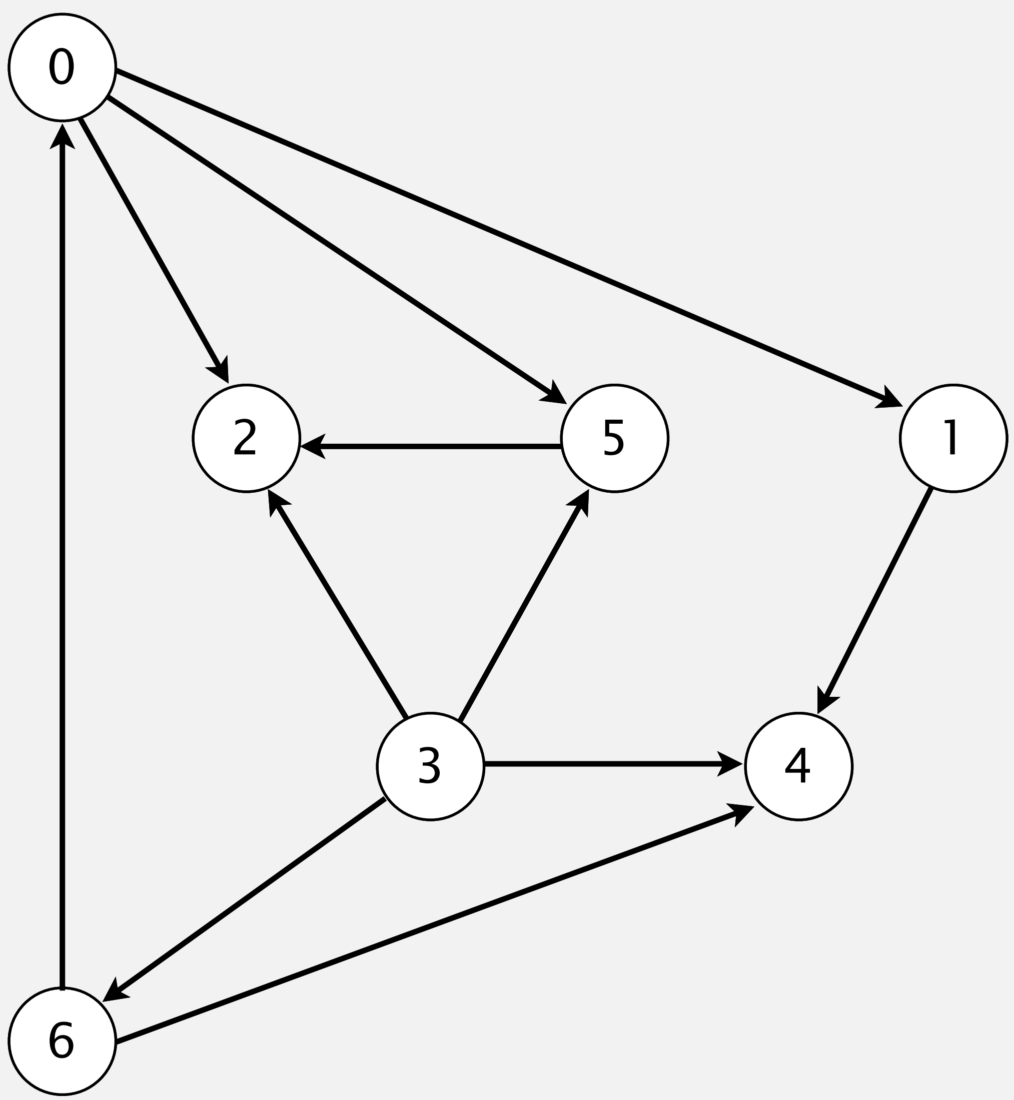


topological order

Solution. DFS. What else?

Topological sort demo

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



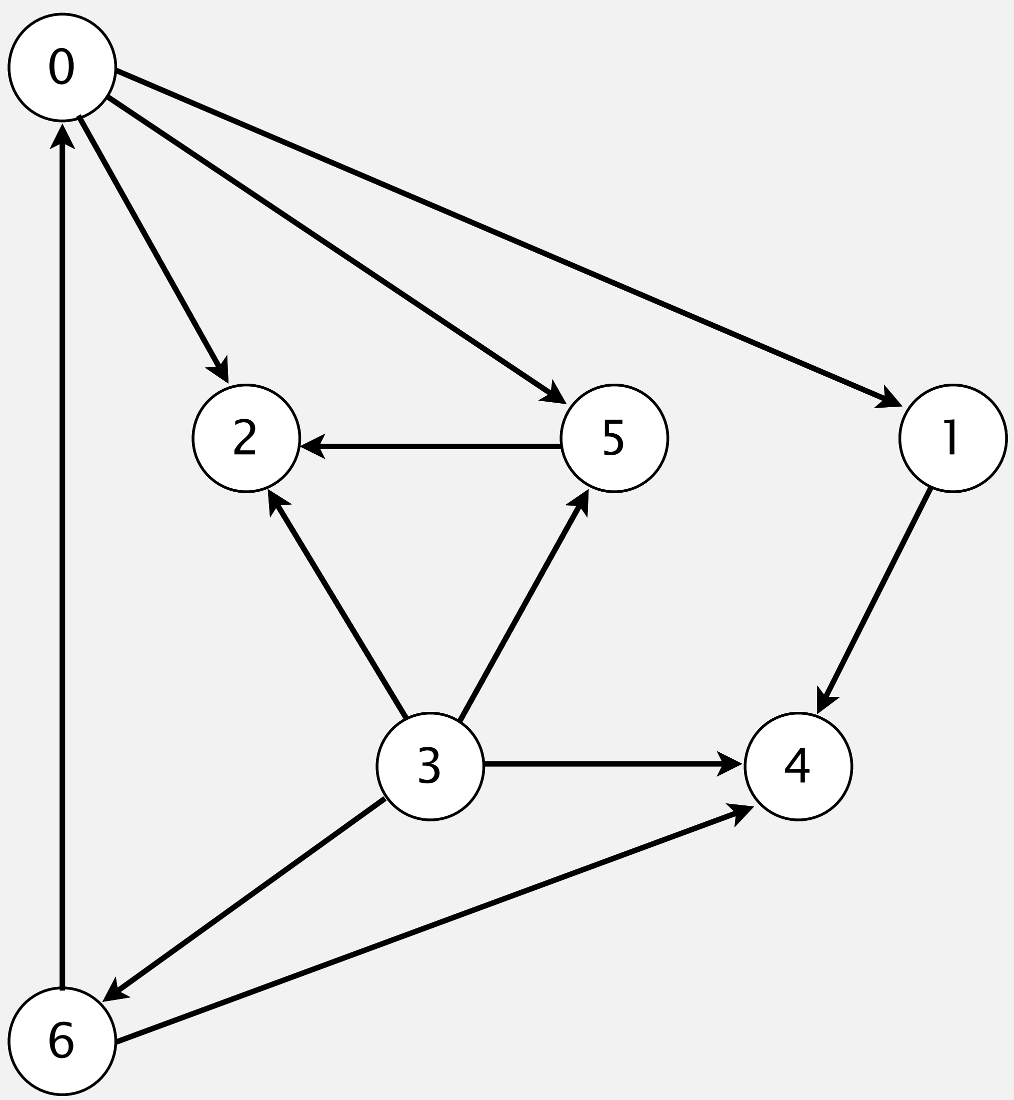
a directed acyclic graph

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

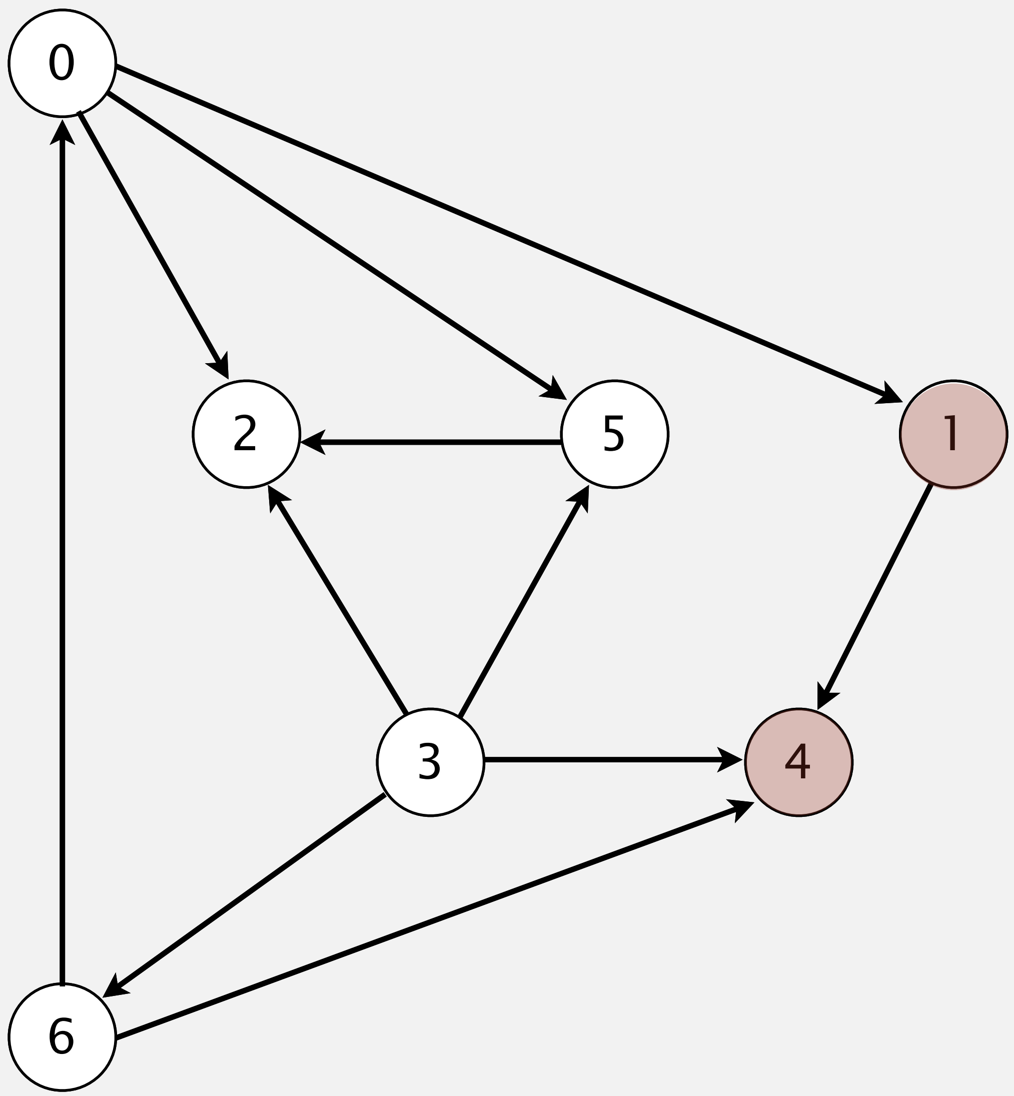
```
public class DepthFirstOrder
{
   private boolean[] marked;
   private Stack<Integer> reversePostorder;

   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 was done before v.

- Case 2: dfs(w) has not yet been called.
  dfs(w) will get called directly or indirectly
  by dfs(v) and will finish before dfs(v).
  Thus, w will be done before v.

- Case 3: dfs(w) has already been called,
  but has not yet returned.
  Can't happen in a DAG: function call stack contains
  path from w to v, so v→w would complete a cycle.

```
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

case 1

case 2

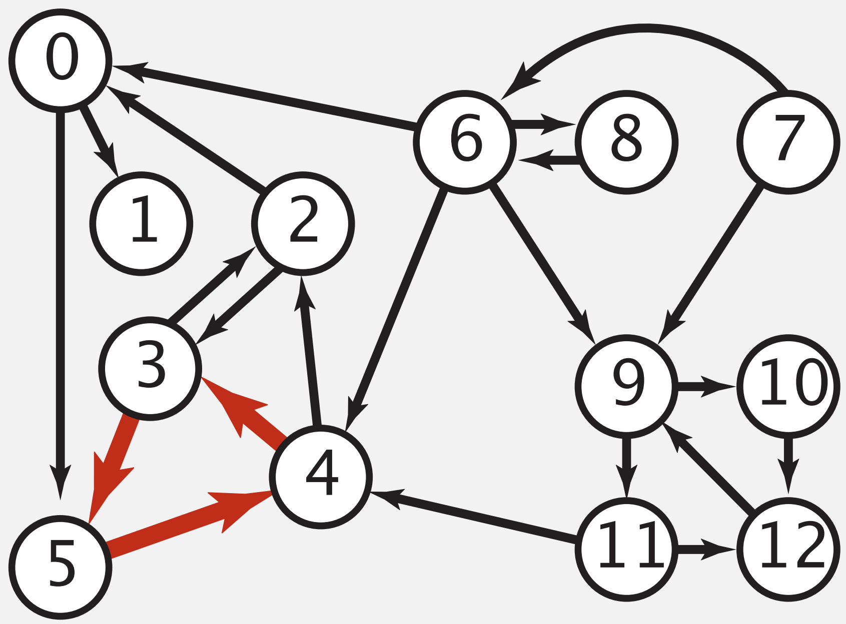
all vertices pointing from 3 are done before 3 is done,
so they appear after 3 in topological order

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.



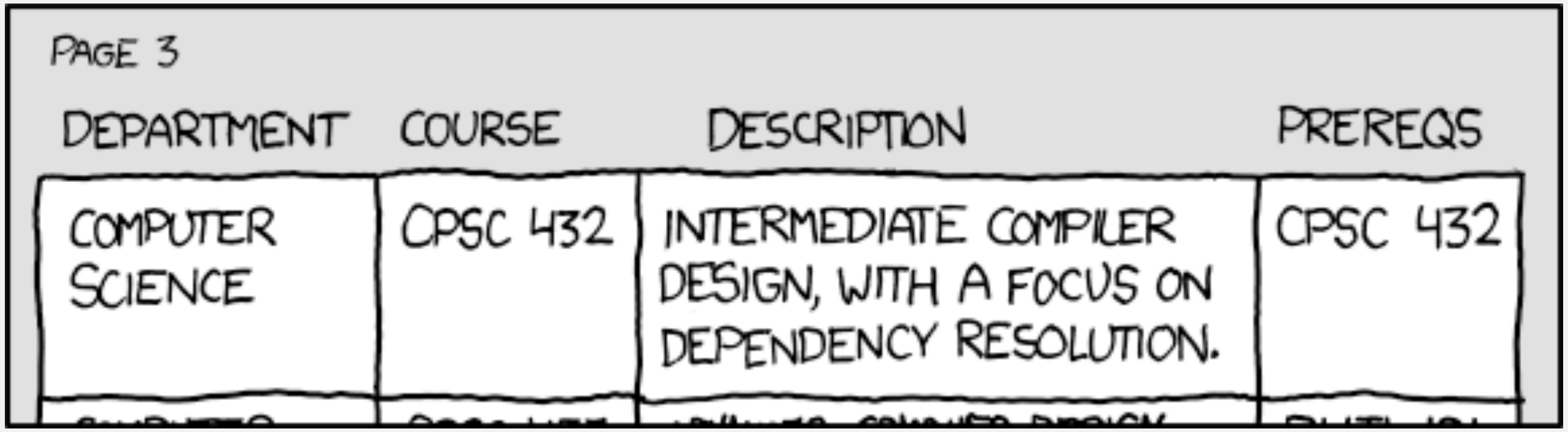
a digraph with a directed cycle

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
{
   ...
}
```

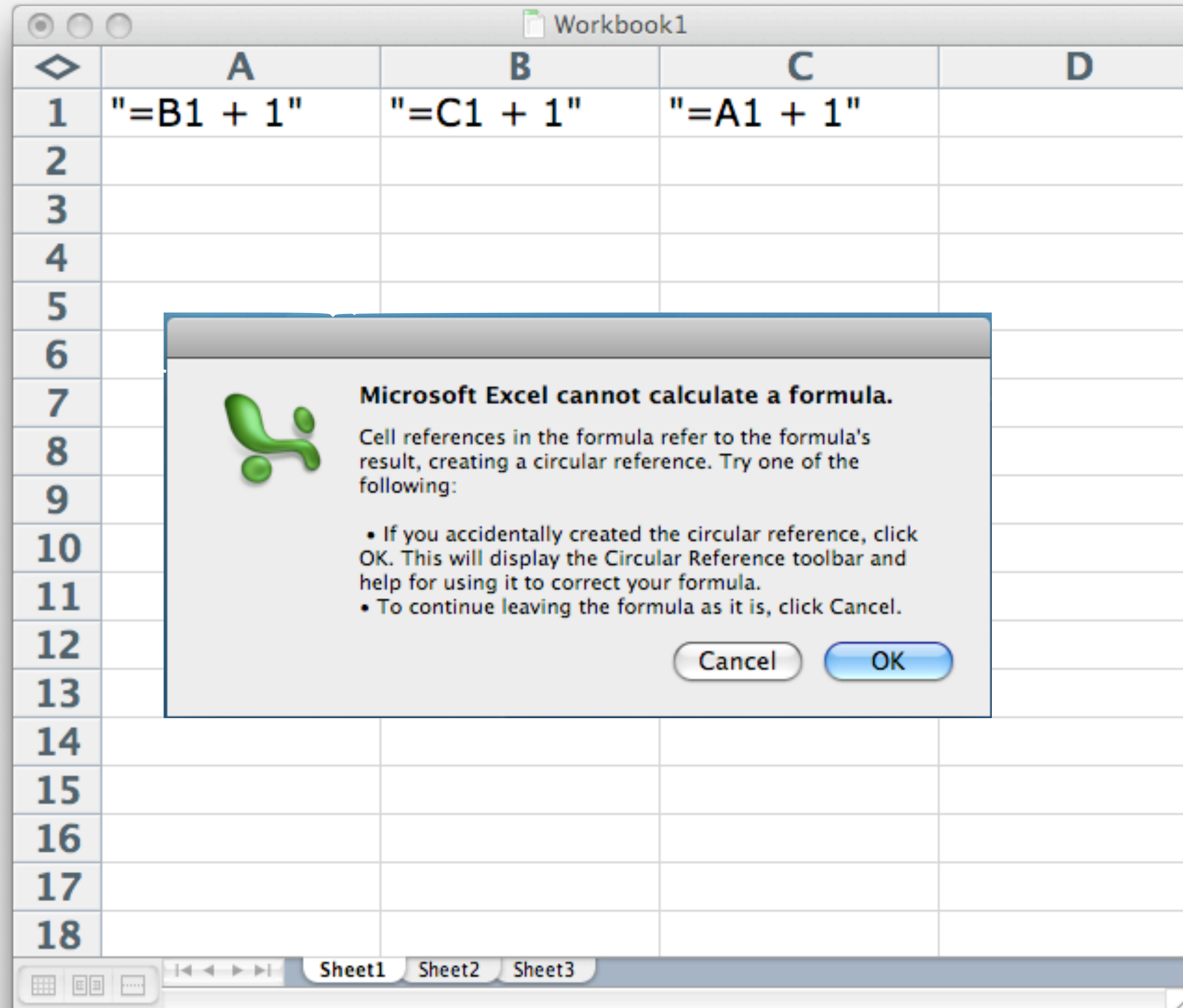
```
public class B extends C
{
   ...
}
```

```
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 (and has a circular reference toolbar!)



Directed cycle detection application: WordNet

The WordNet database (occasionally) needs directed cycle detection.

WordNet Search - 3.0 - WordNet home page - Glossary - Help

Word to search for: dampen [Search WordNet]

Display Options: (Select option to change) [Change]

Key: "S:" = Show Synset (semantic) relations, "W:" = Show Word (lexical) relations

Verb

- S: (v) stifle, **dampen** (smother or suppress) *"Stifle your curiosity"*
  - *direct troponym* / *full troponym*
  - *direct hypernym* / *inherited hypernym* / *sister term*
    - S: (v) suppress, stamp down, inhibit, subdue, conquer, curb (to put down by force or authority) *"suppress a nascent uprising"; "stamp down on littering"; "conquer one's desires"*
      - *direct troponym* / *full troponym*
      - *direct hypernym* / *inherited hypernym* / *sister term*
        - S: (v) control, hold in, hold, contain, check, curb, moderate (lessen the intensity of; temper; hold in restraint; hold or keep within limits) *"moderate your alcohol intake"; "hold your tongue"; "hold your temper"; "control your anger"*
          - *direct troponym* / *full troponym*
          - *direct hypernym* / *inherited hypernym* / *sister term*
            - S: (v) restrain, keep, keep back, hold back (keep under control; keep in check) *"suppress a smile"; "Keep your temper"; "keep your cool"*
              - *direct troponym* / *full troponym*
              - *direct hypernym* / *inherited hypernym* / *sister term*
                - S: (v) inhibit, bottle up, suppress (control and refrain from showing; of emotions, desires, impulses, or behavior)
                  - *direct troponym* / *full troponym*
                  - *direct hypernym* / *inherited hypernym* / *sister term*
                    - S: (v) restrain, keep, keep back, hold back (keep under control; keep in check) *"suppress a smile"; "Keep your temper"; "keep your cool"*
                      - *direct troponym* / *full troponym*
                      - ***direct hypernym*** / *inherited hypernym* / *sister term*
                        - S: (v) inhibit, bottle up, suppress (control and refrain from showing; of emotions, desires, impulses, or behavior)
                      - *derivationally related form*
                      - *sentence frame*
                  - *derivationally related form*

Directed cycle detection application: symbolic links

The Linux file system does not do cycle detection.

```
% ln -s a.txt b.txt
% ln -s b.txt c.txt
% ln -s c.txt a.txt

% more a.txt
a.txt: Too many levels of symbolic links
```

Depth-first search orders

Observation. DFS visits each vertex exactly once. The order in which it does so can be important.

Orderings.

- Preorder: order in which `dfs()` is called.
- Postorder: order in which `dfs()` returns.
- Reverse postorder: reverse order in which `dfs()` returns.

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

http://algs4.cs.princeton.edu

4.2 Directed Graphs

- *introduction*
- *digraph API*
- *digraph search*
- *topological sort*
- *strong components*

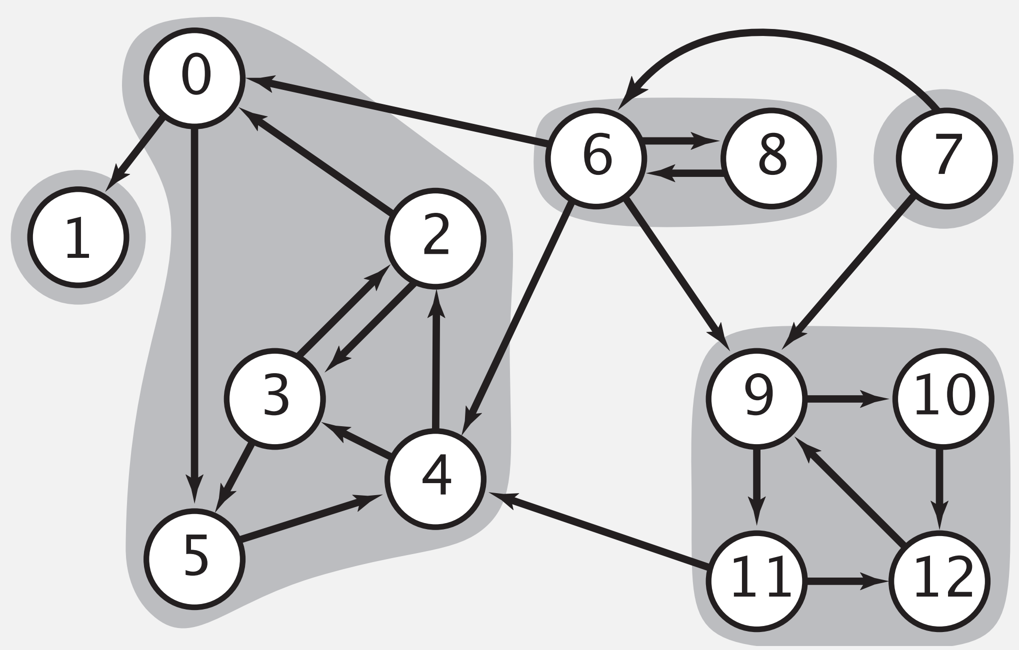
Strongly-connected components

Def. Vertices v and w are strongly connected if there is both a directed path from v to w and a directed path from w to v.

Key property. Strong connectivity is an equivalence relation:

- v is strongly connected to v.
- If v is strongly connected to w, then w is strongly connected to v.
- If v is strongly connected to w and w to x, then v is strongly connected to x.

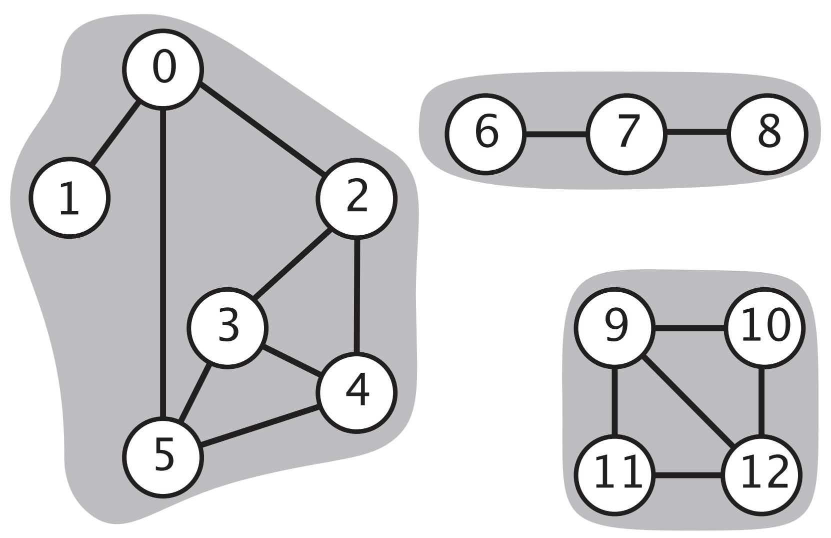
Def. A strong component is a maximal subset of strongly-connected vertices.



5 strongly-connected components

Connected components vs. strongly-connected components

v and w are **connected** if there is a path between v and w



3 connected components

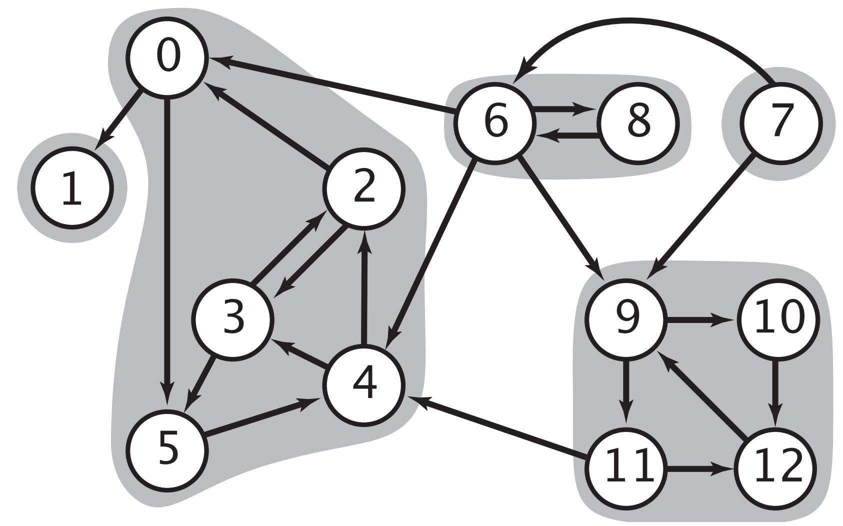
connected component id (easy to compute with DFS)

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

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

constant-time client connectivity query

v and w are **strongly connected** if there is both a directed path from v to w and a directed path from w to v



5 strongly-connected components

strongly-connected component id (how to compute?)

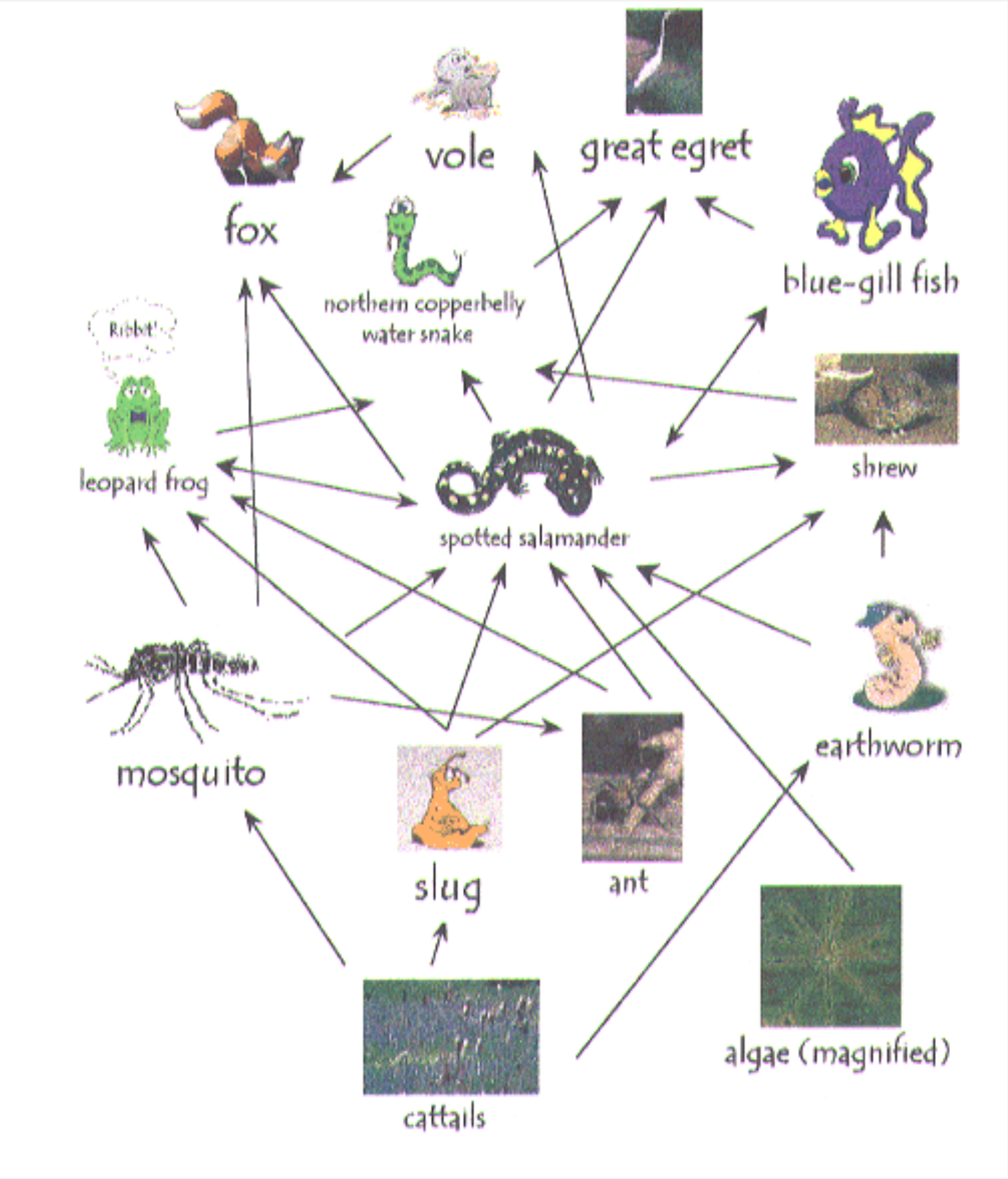
	0	1	2	3	4	5	6	7	8	9	10	11	12
id[]	1	0	1	1	1	1	3	4	3	2	2	2	2

```
public boolean stronglyConnected(int v, int w)
{  return id[v] == id[w];  }
```

constant-time client strong-connectivity query

Strong component application: ecological food webs

Food web graph. Vertex = species; edge = from producer to consumer.



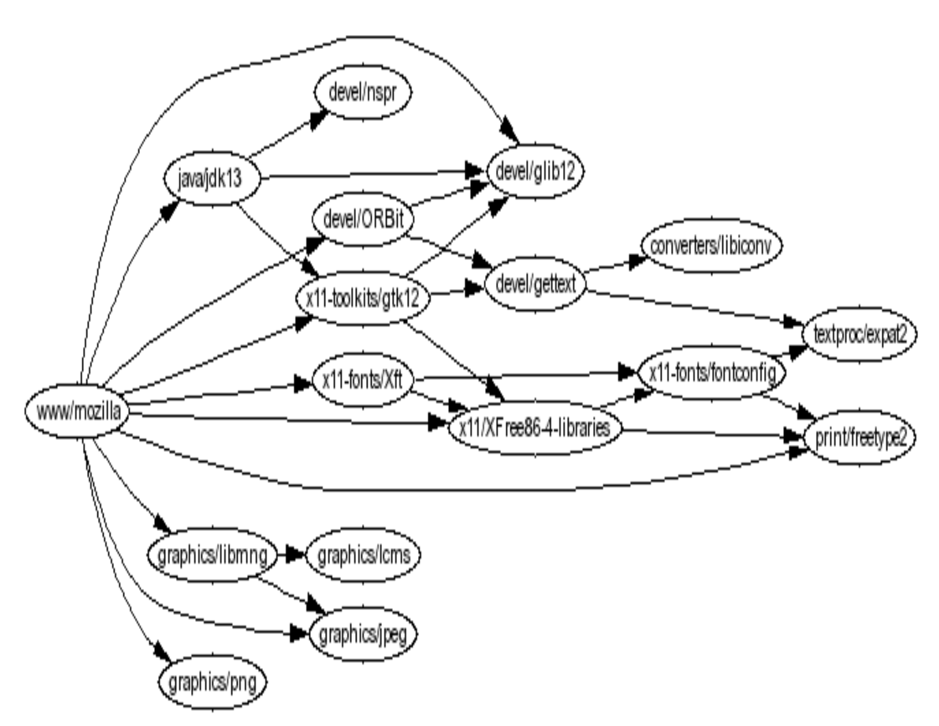
http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif

Strong component. Subset of species with common energy flow.

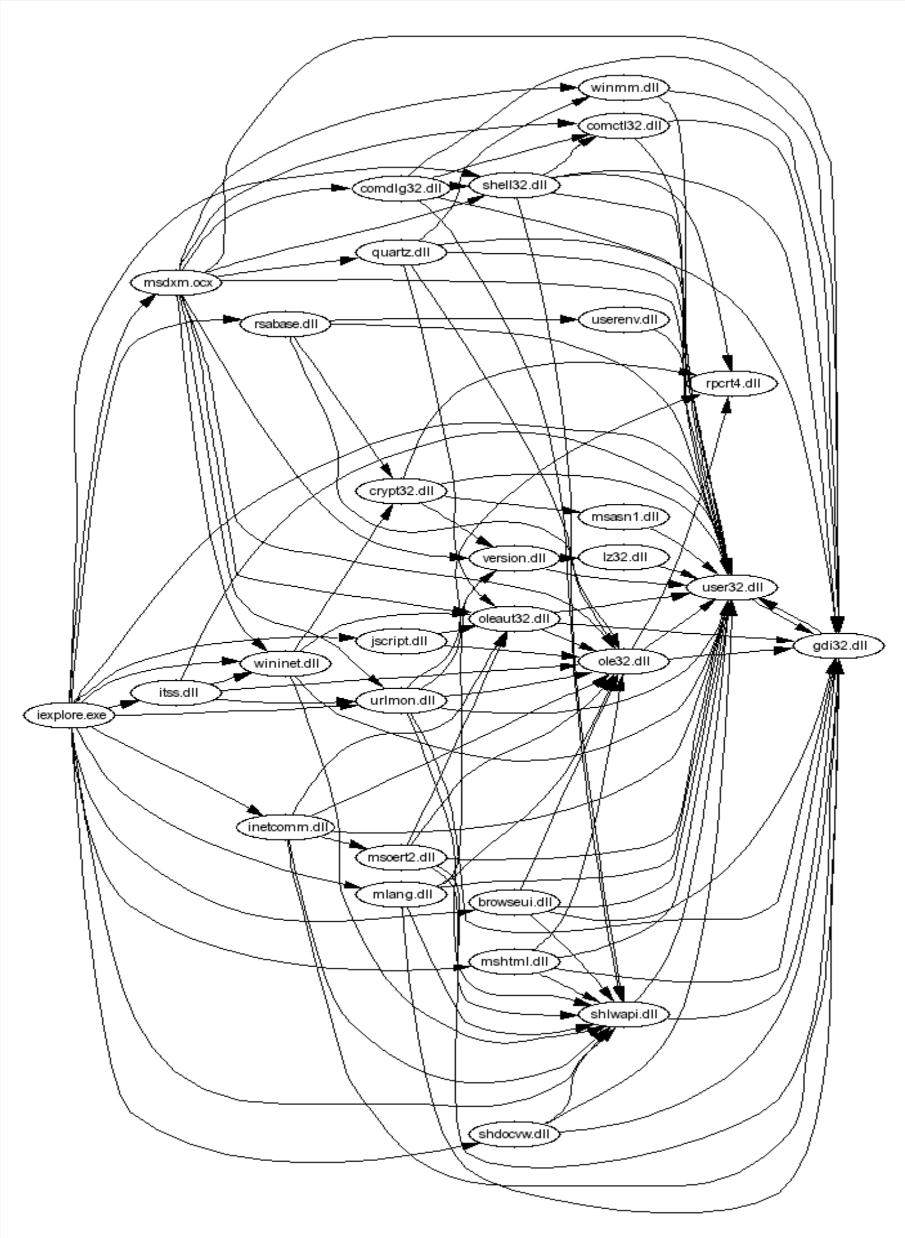
Strong component application: software modules

Software module dependency graph.

- Vertex = software module.
- Edge: from module to dependency.



Firefox



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: Algs4++.
- Demonstrated broad applicability and importance of DFS.

1980s: easy two-pass linear-time algorithm (Kosaraju-Sharir).

- Forgot notes for lecture; developed algorithm in order to teach it!
- Later found in Russian scientific literature (1972).

1990s: more easy linear-time algorithms.

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

Kosaraju-Sharir algorithm: intuition

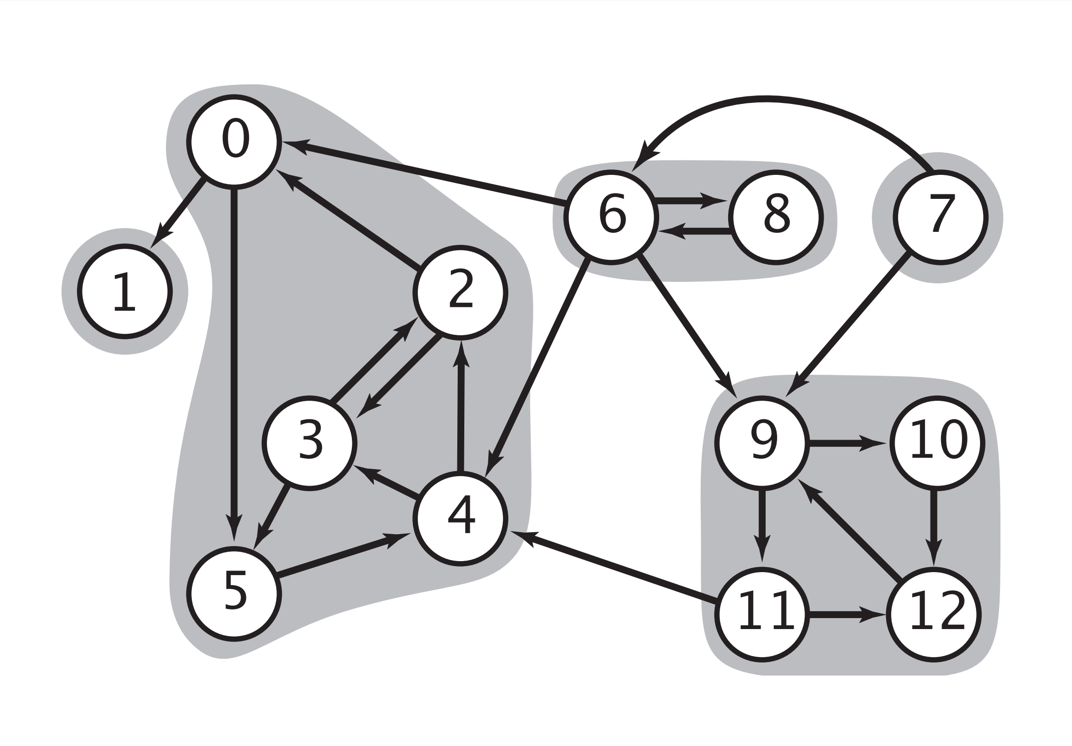
Reverse graph. Strong components in G are same as in G^R.

Kernel DAG. Contract each strong component into a single vertex.

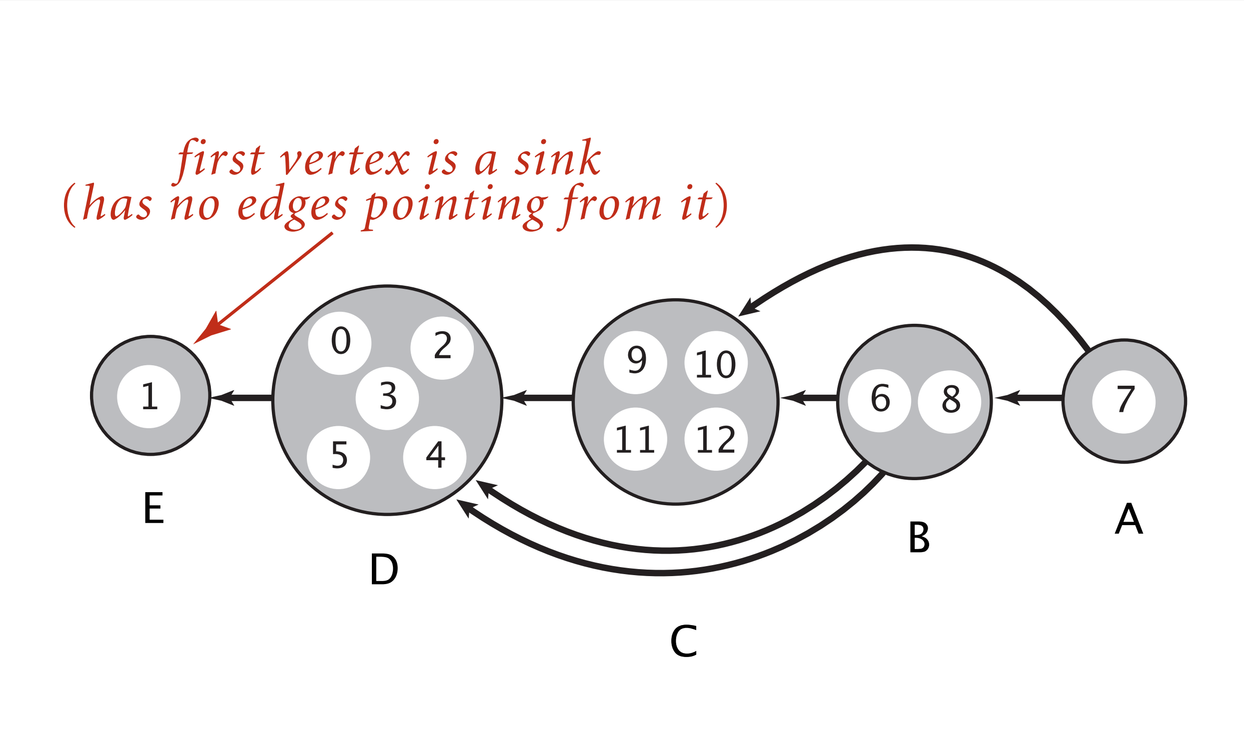
Idea.

how to compute?

- Compute topological order (reverse postorder) in kernel DAG.
- Run DFS, considering vertices in reverse topological order.



digraph G and its strong components

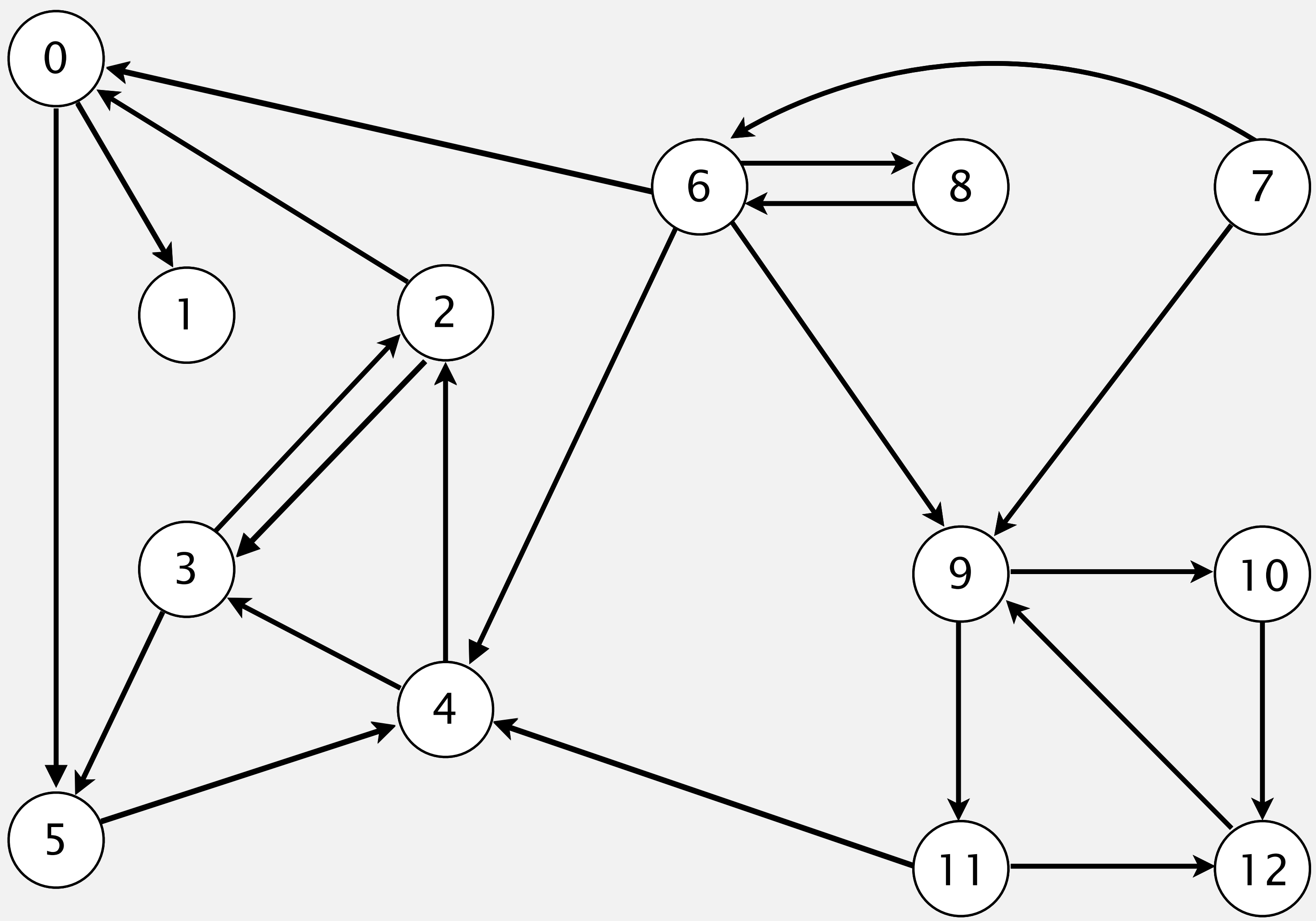


kernel DAG of G (topological order: A B C D E)

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

Phase 2. Run DFS in G, visiting unmarked vertices in reverse postorder of G^R.

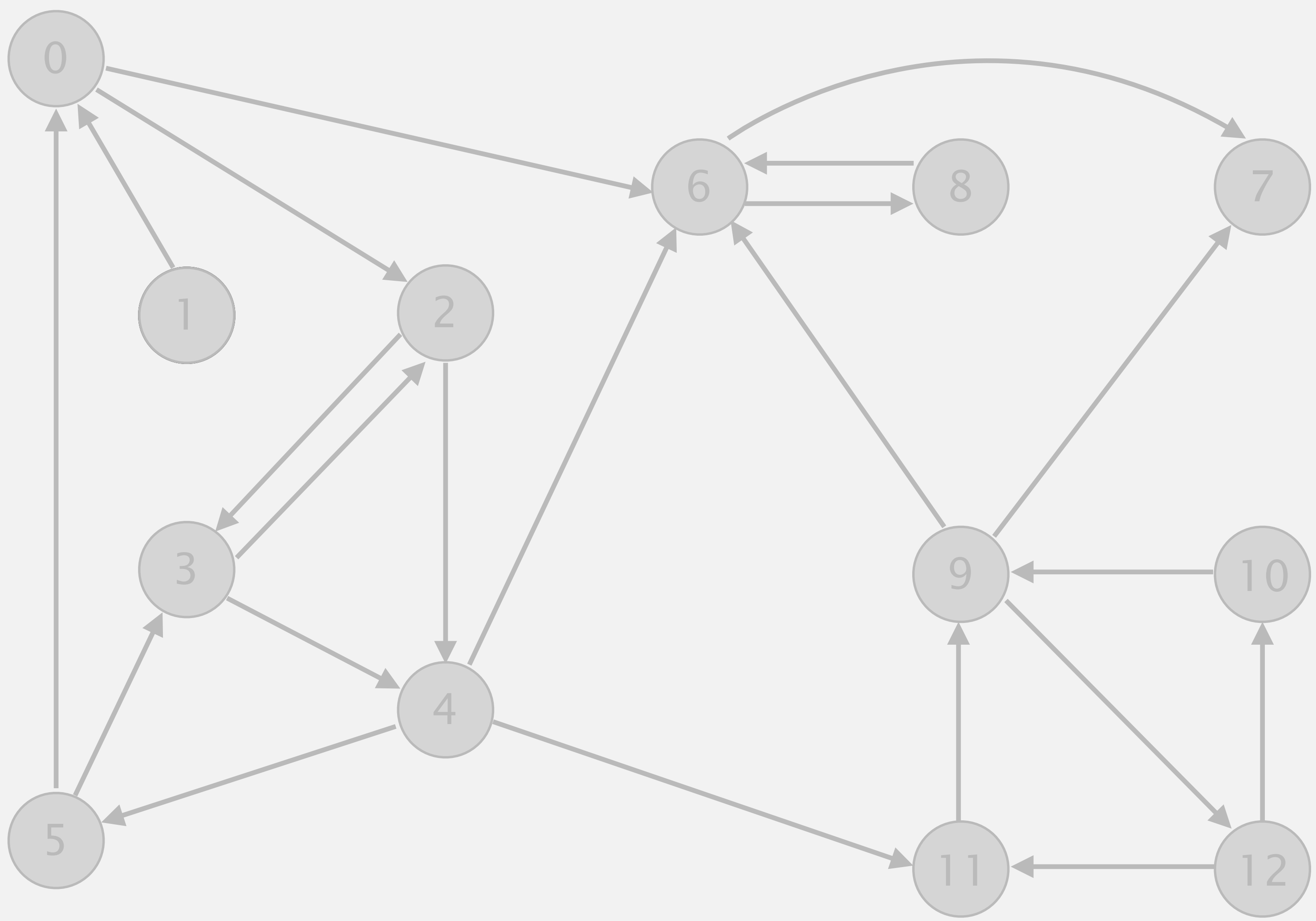


digraph G

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

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

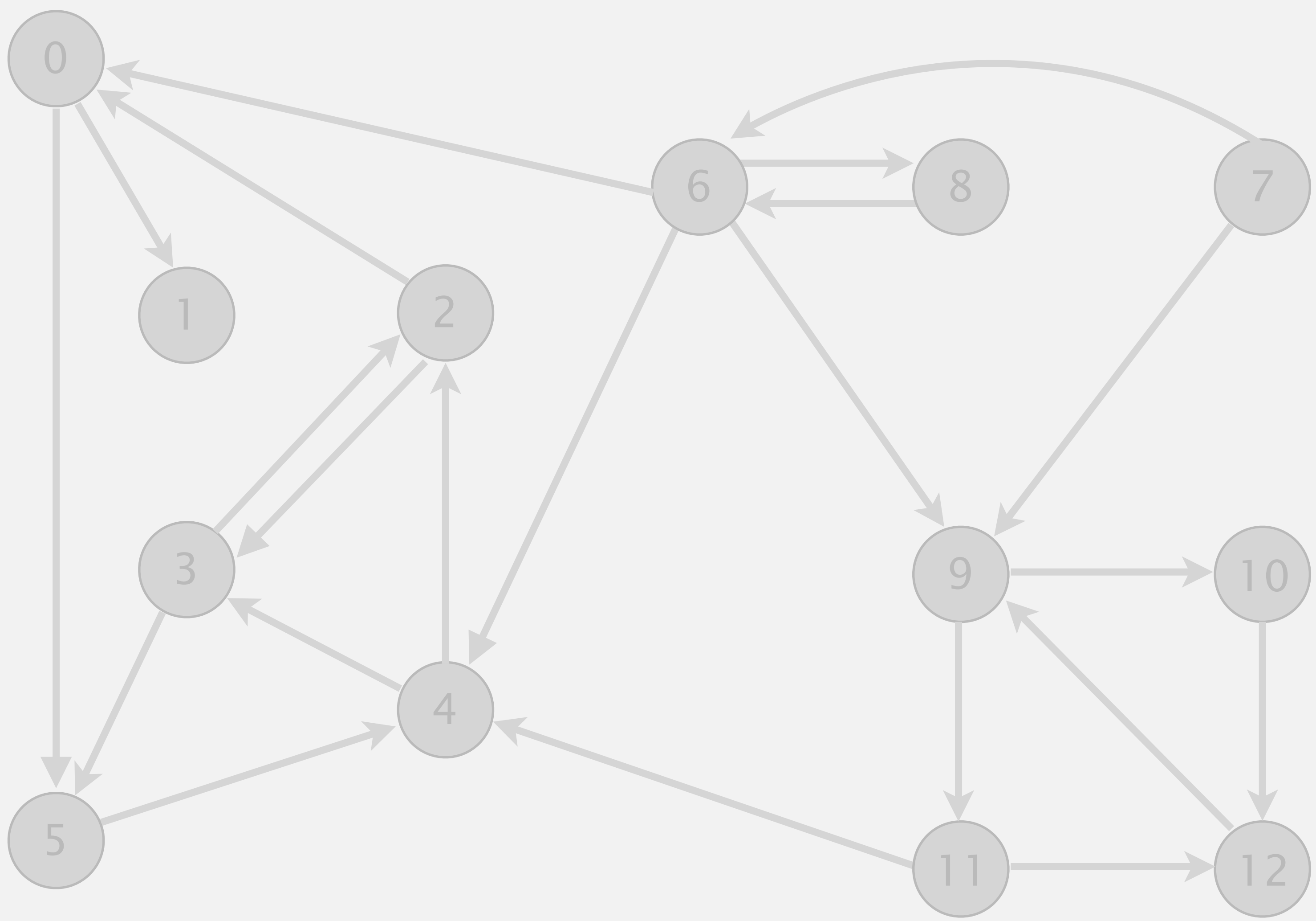


reverse digraph G^R

Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in G, visiting unmarked vertices in reverse postorder of G^R.

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



v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

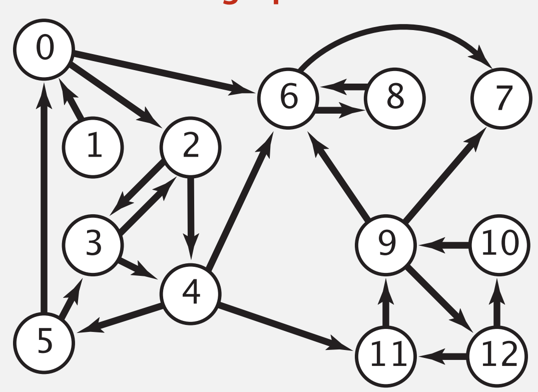
done

Kosaraju-Sharir algorithm

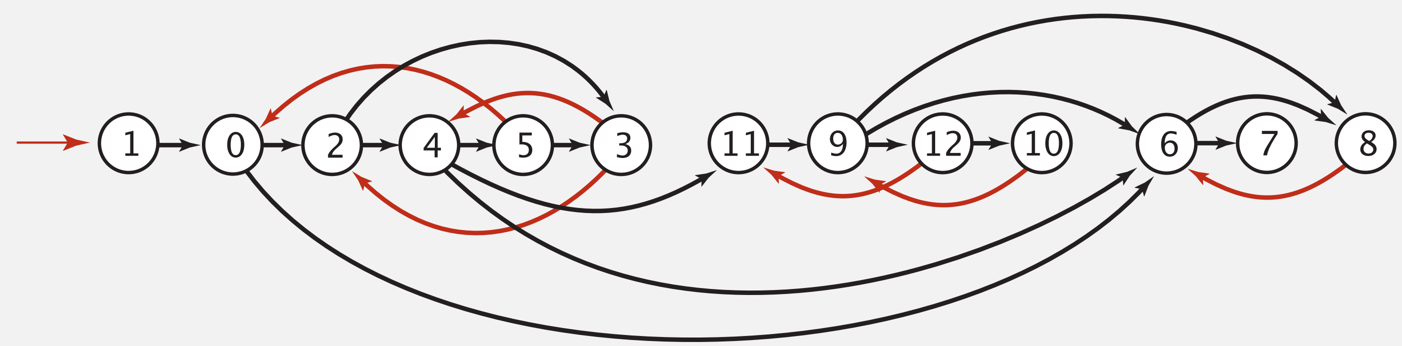
Simple (but mysterious) algorithm for computing strong components.

- Phase 1: run DFS on G^R to compute reverse postorder.
- Phase 2: run DFS on G, considering vertices in order given by first DFS.

DFS in reverse digraph G^R



check unmarked vertices in the order
0 1 2 3 4 5 6 7 8 9 10 11 12



reverse postorder for use in second `dfs()`
1 0 2 4 5 3 11 9 12 10 6 7 8

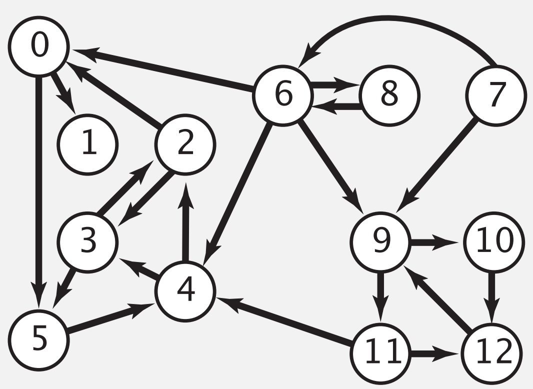
```
dfs(0)
  dfs(6)
    dfs(8)
      check 6
    8 done
    dfs(7)
    7 done
  6 done
  dfs(2)
    dfs(4)
      dfs(11)
        dfs(9)
          dfs(12)
            check 11
            dfs(10)
              check 9
            10 done
          12 done
          check 7
          check 6
...
```

Kosaraju-Sharir algorithm

Simple (but mysterious) algorithm for computing strong components.

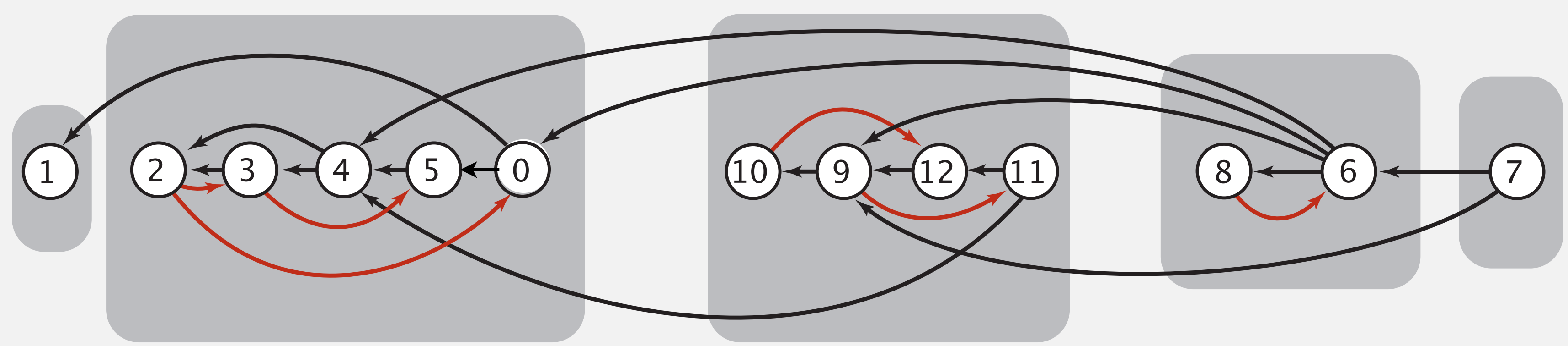
- Phase 1: run DFS on G^R to compute reverse postorder.
- Phase 2: run DFS on G, considering vertices in order given by first DFS.

DFS in original digraph G



check unmarked vertices in the order

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



```
dfs(1)
1 done
```

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

```
dfs(11)
| check 4
| dfs(12)
| | dfs(9)
| | | check 11
| | | dfs(10)
| | | | check 12
| | | 10 done
| | 9 done
| 12 done
11 done
check 9
check 12
check 10
```

```
dfs(6)
| check 9
| check 4
| dfs(8)
| | check 6
| 8 done
| check 0
6 done
```

```
dfs(7)
| check 6
| check 9
7 done
check 8
```

Kosaraju-Sharir algorithm

Proposition. Kosaraju-Sharir algorithm computes the strong components of a digraph in time proportional to $E + V$.

Pf.

- Running time: bottleneck is running DFS twice (and computing G^R).
- Correctness: tricky, see textbook (2nd printing).
- Implementation: easy!

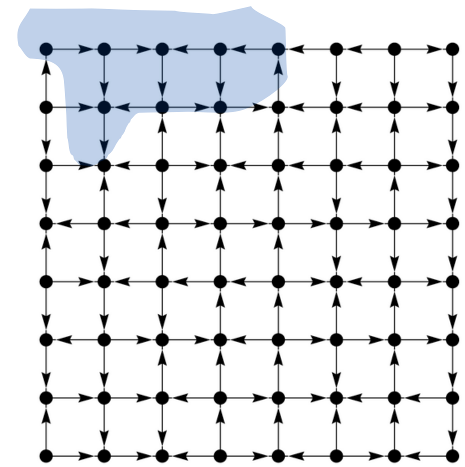
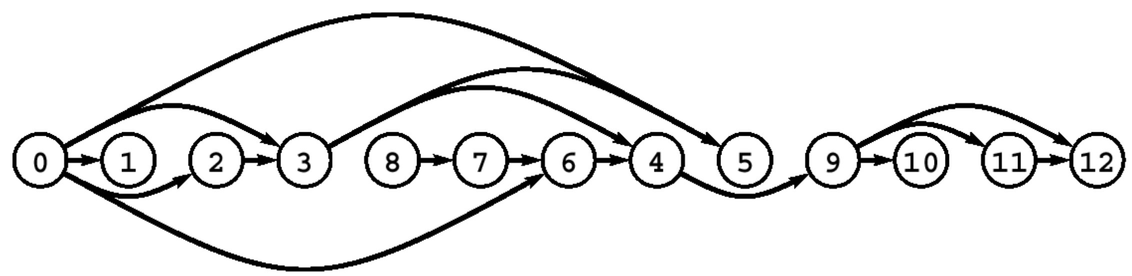
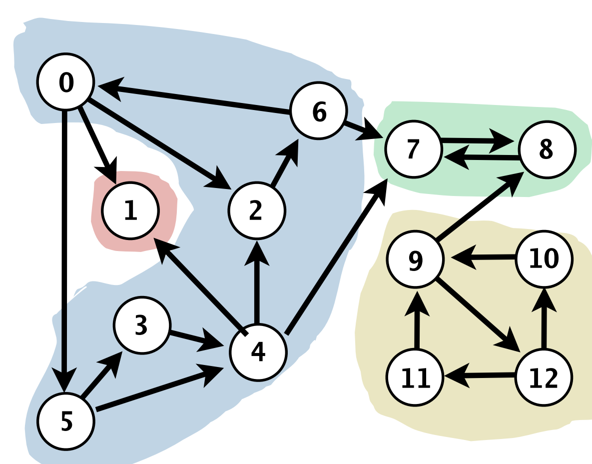
Connected components in an undirected graph (with DFS)

```
public class CC
{
   private boolean marked[];
   private int[] id;
   private int count;

   public CC(Graph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];

      for (int v = 0; v < G.V(); v++)
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }

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

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

Strong components in a digraph (with two DFSs)

```
public class KosarajuSharirSCC
{
   private boolean marked[];
   private int[] id;
   private int count;

   public KosarajuSharirSCC(Digraph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];
      DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
      for (int v : dfs.reversePostorder())
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }

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

   public boolean stronglyConnected(int v, int w)
   {  return id[v] == id[w];  }
}
```

Digraph-processing summary: algorithms of the day

single-source reachability in a digraph		DFS
topological sort in a DAG		DFS
strong components in a digraph		Kosaraju-Sharir DFS (twice)