



<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

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



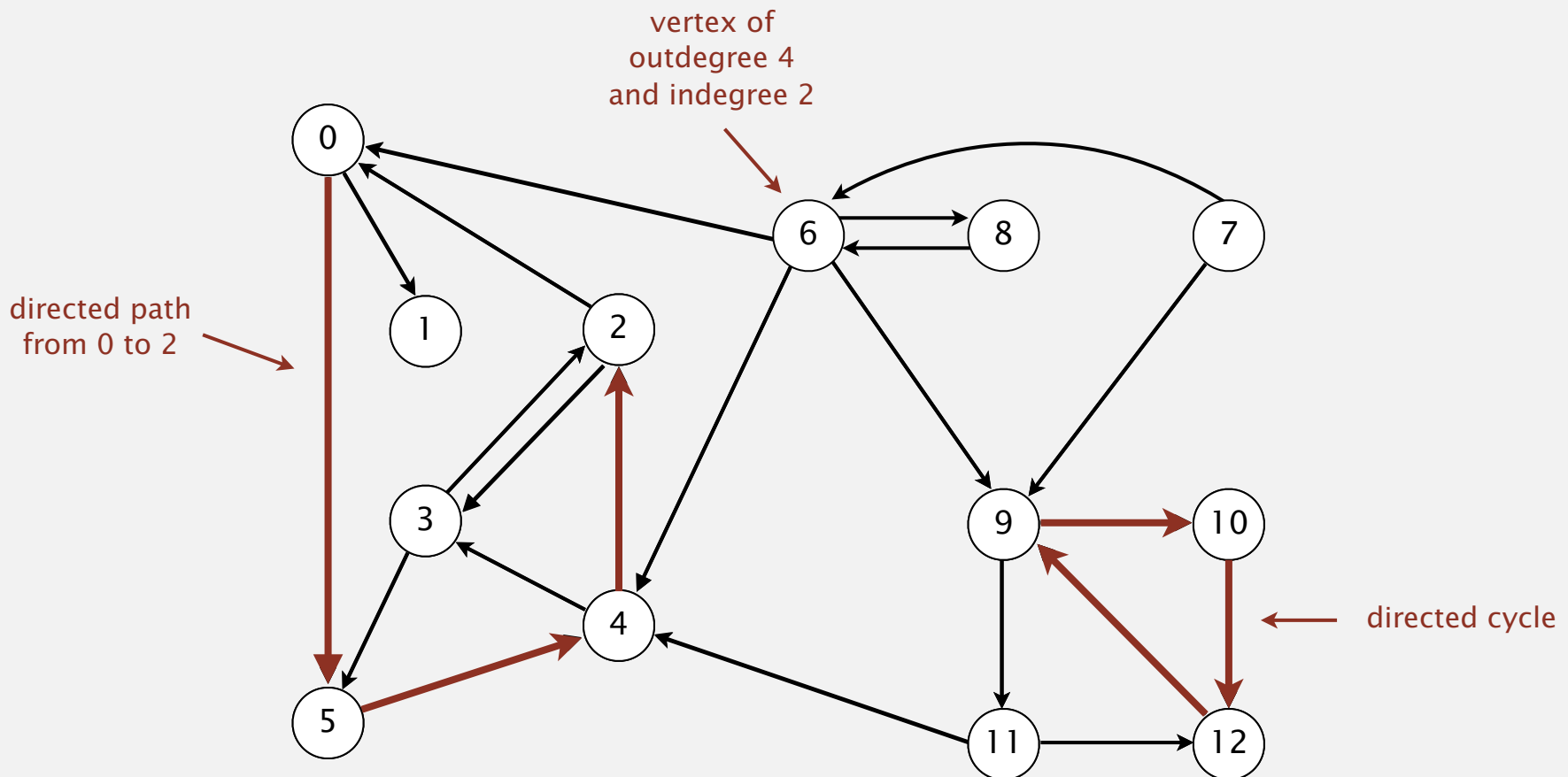
<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

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

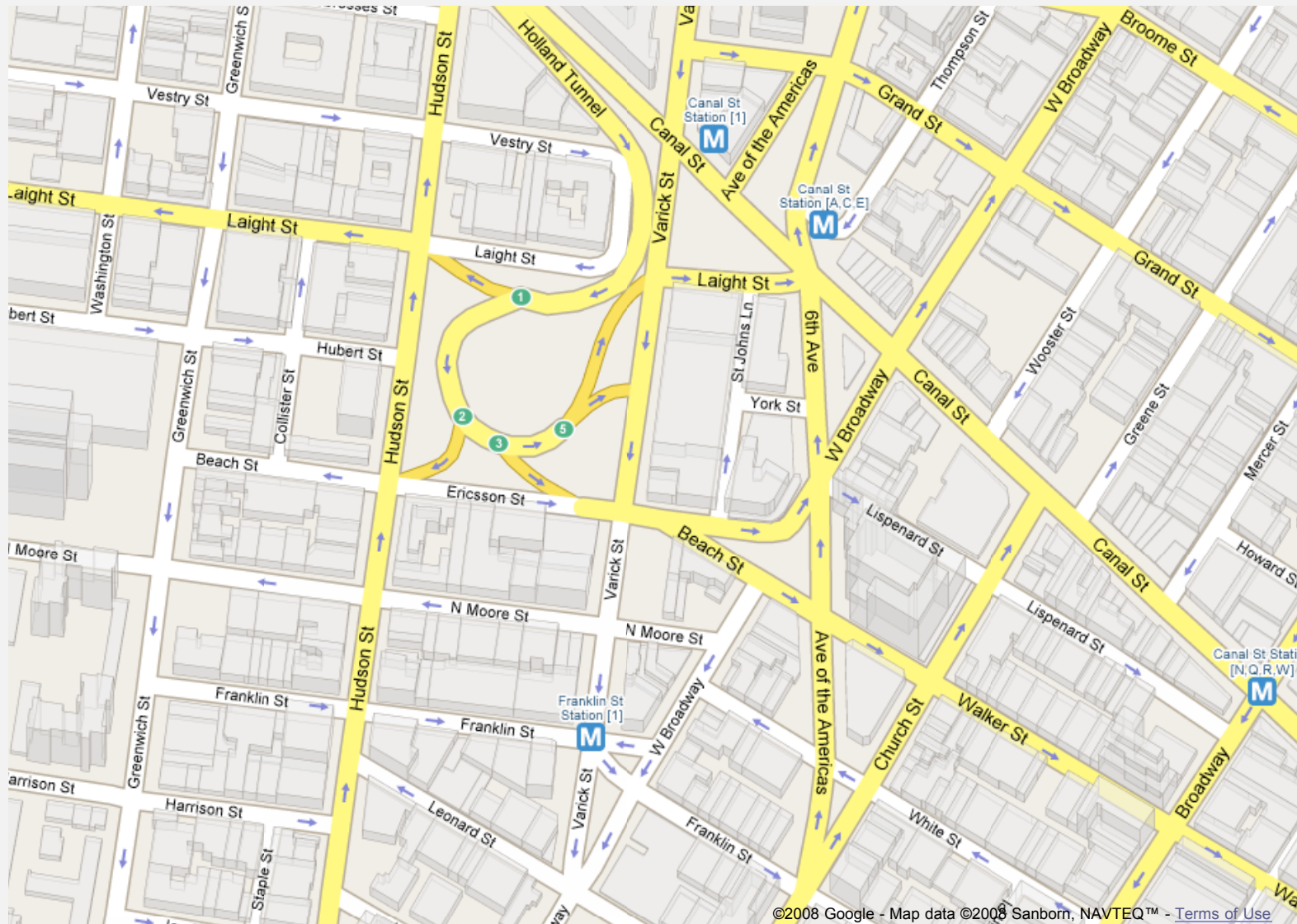
Directed graphs

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



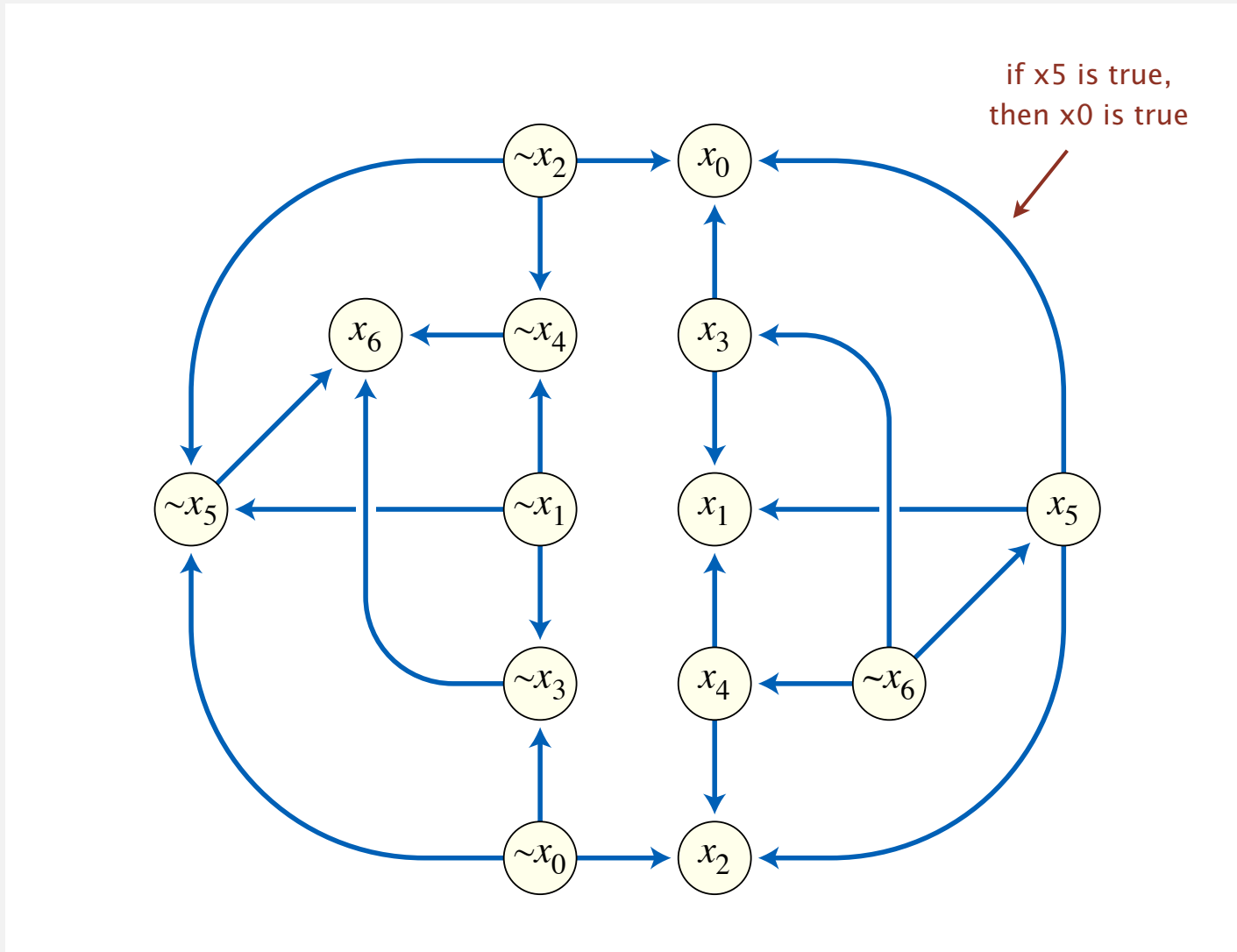
Road network

Vertex = intersection; edge = one-way street.



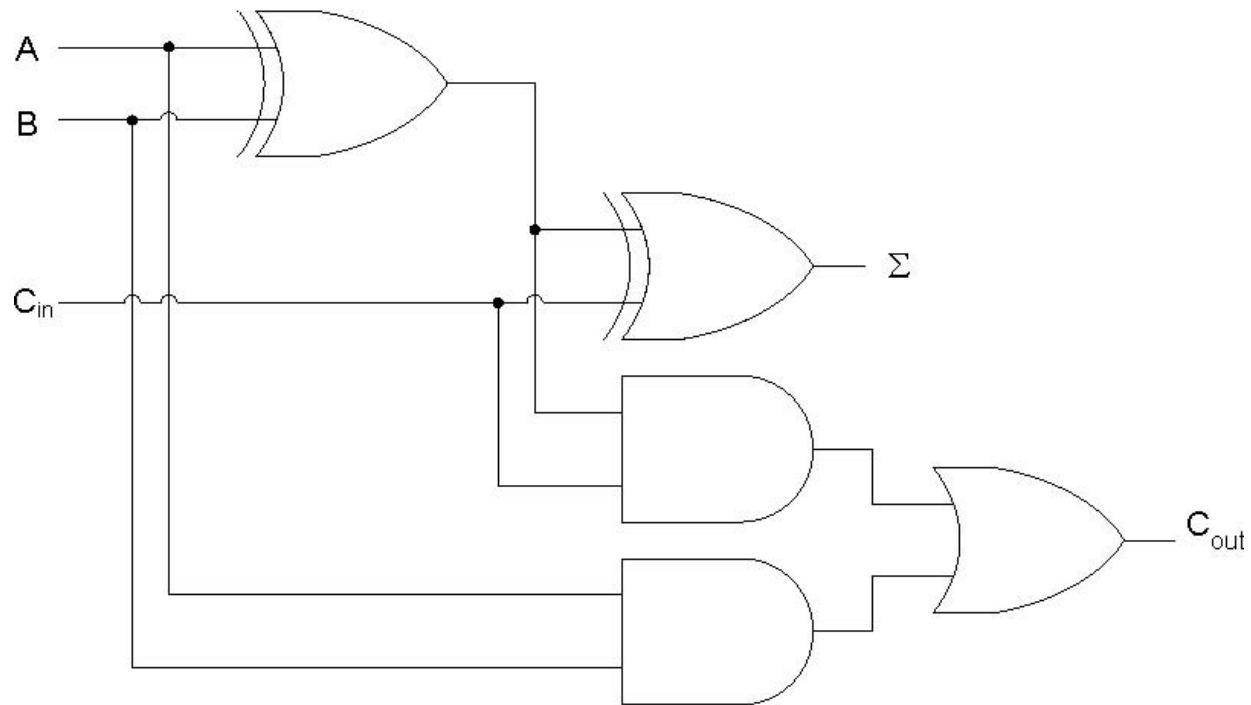
Implication graph

Vertex = variable; edge = logical implication.



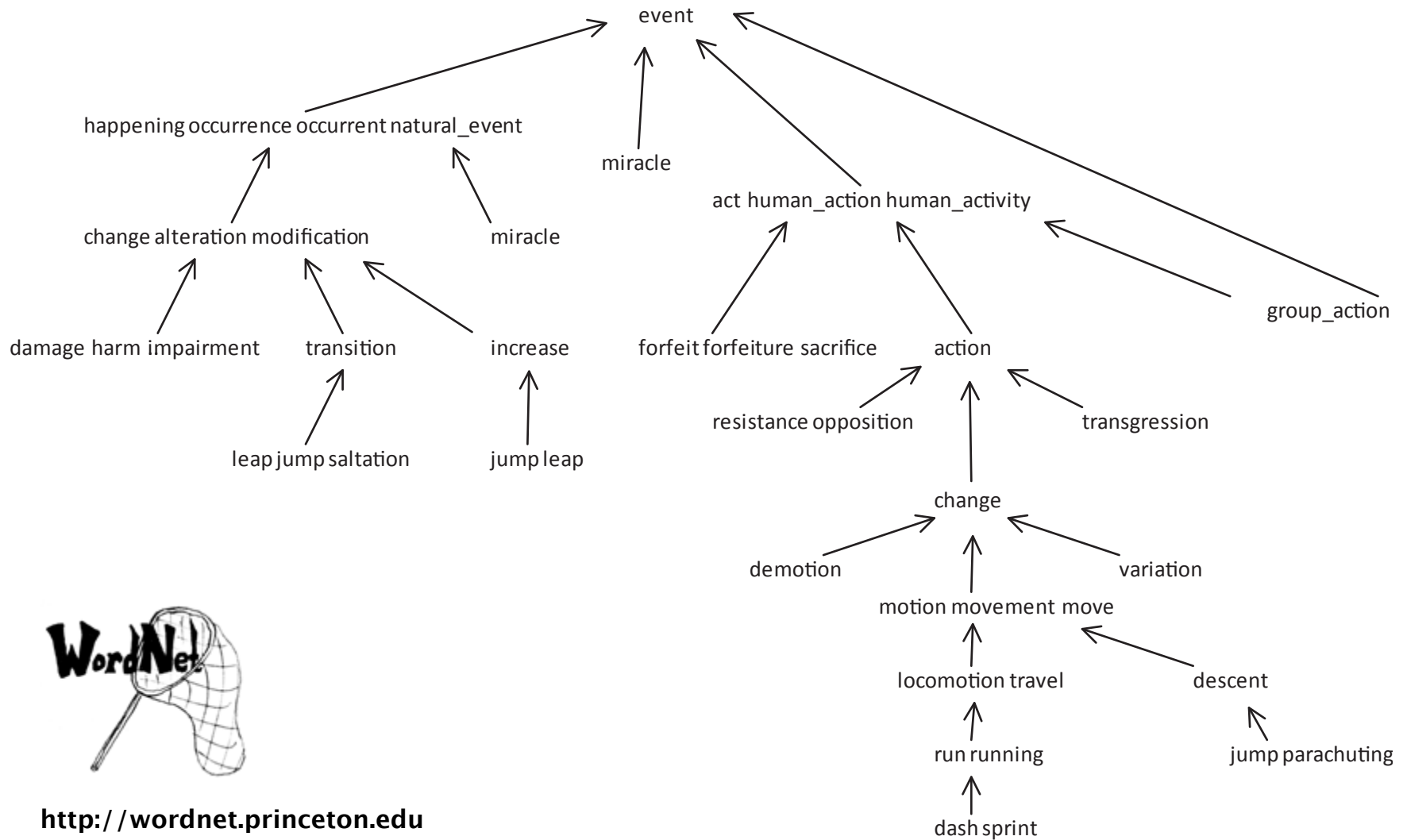
Combinational circuit

Vertex = logical gate; edge = wire.



WordNet graph

Vertex = synset; edge = hypernym relationship.



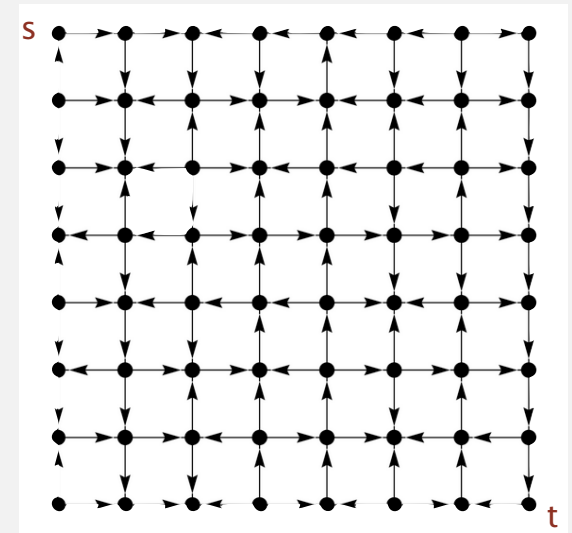
<http://wordnet.princeton.edu>

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

Path. Is there a directed path from s to t ?



Shortest path. What is the shortest directed path from s to t ?

Topological sort. Can you draw a digraph 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 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*



<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

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

Digraph API

```
public class Digraph
```

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

```
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);
```

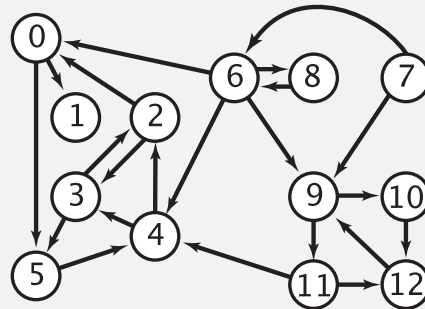
← read digraph from
input stream

← print out each
edge (once)

Digraph API

tinyDG.txt

```
V → 13  
← E 22  
4 2  
2 3  
3 2  
6 0  
0 1  
2 0  
11 12  
12 9  
9 10  
9 11  
7 9  
10 12  
11 4  
4 3  
3 5  
6 8  
8 6  
⋮
```



```
% java Digraph tinyDG.txt  
0->5  
0->1  
2->0  
2->3  
3->5  
3->2  
4->3  
4->2  
5->4  
:  
11->4  
11->12  
12-9
```

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

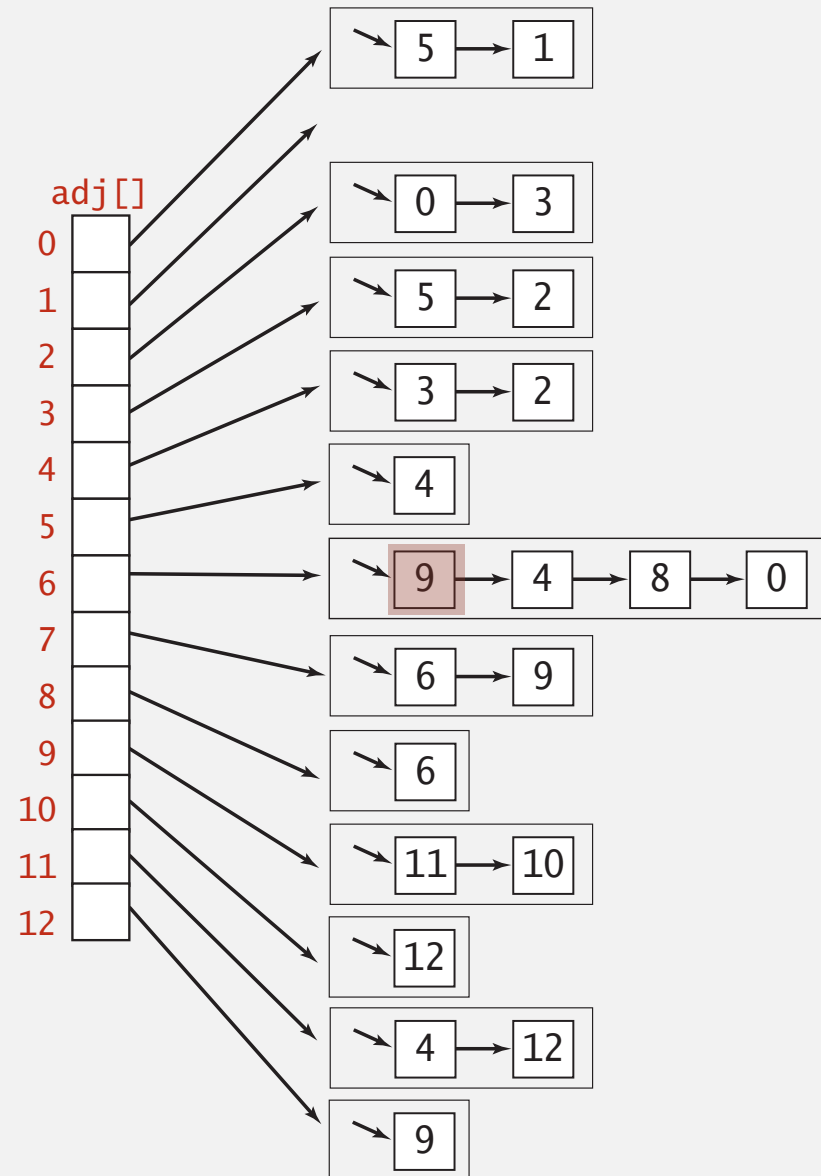
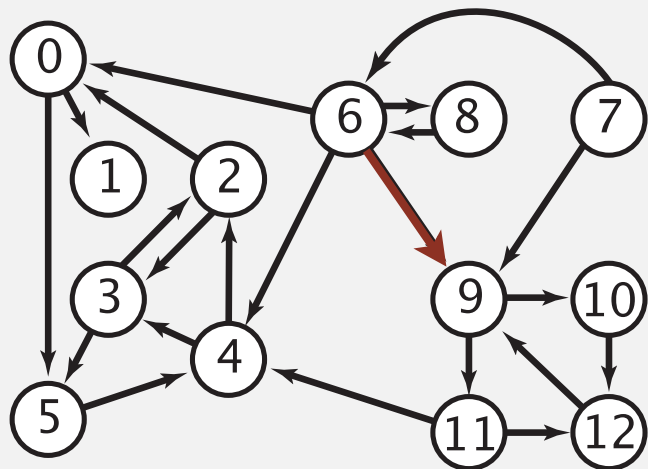
← read digraph from
input stream

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

← print out each
edge (once)

Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.



Adjacency-lists graph representation (review): Java implementation

```
public class Graph  
{
```

```
    private final int V;  
    private final Bag<Integer>[] adj;
```

← adjacency lists

```
    public Graph(int V)
```

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

← create empty graph
with V vertices

```
    public void addEdge(int v, int w)
```

```
    {  
        adj[v].add(w);  
        adj[w].add(v);  
    }
```

← add edge v-w

```
    public Iterable<Integer> adj(int v)
```

```
    { return adj[v]; }
```

← iterator for vertices
adjacent to v

```
}
```

Adjacency-lists digraph representation: Java implementation

```
public class Digraph  
{
```

```
    private final int V;  
    private final Bag<Integer>[] adj;
```

← adjacency lists

```
    public Digraph(int V)
```

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

← create empty digraph
with V vertices

```
    public void addEdge(int v, int w)
```

```
    {  
        adj[v].add(w);  
    }
```

← add edge v→w

```
    public Iterable<Integer> adj(int v)
```

```
    { return adj[v]; }
```

← iterator for vertices
pointing from v

```
}
```

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



<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

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



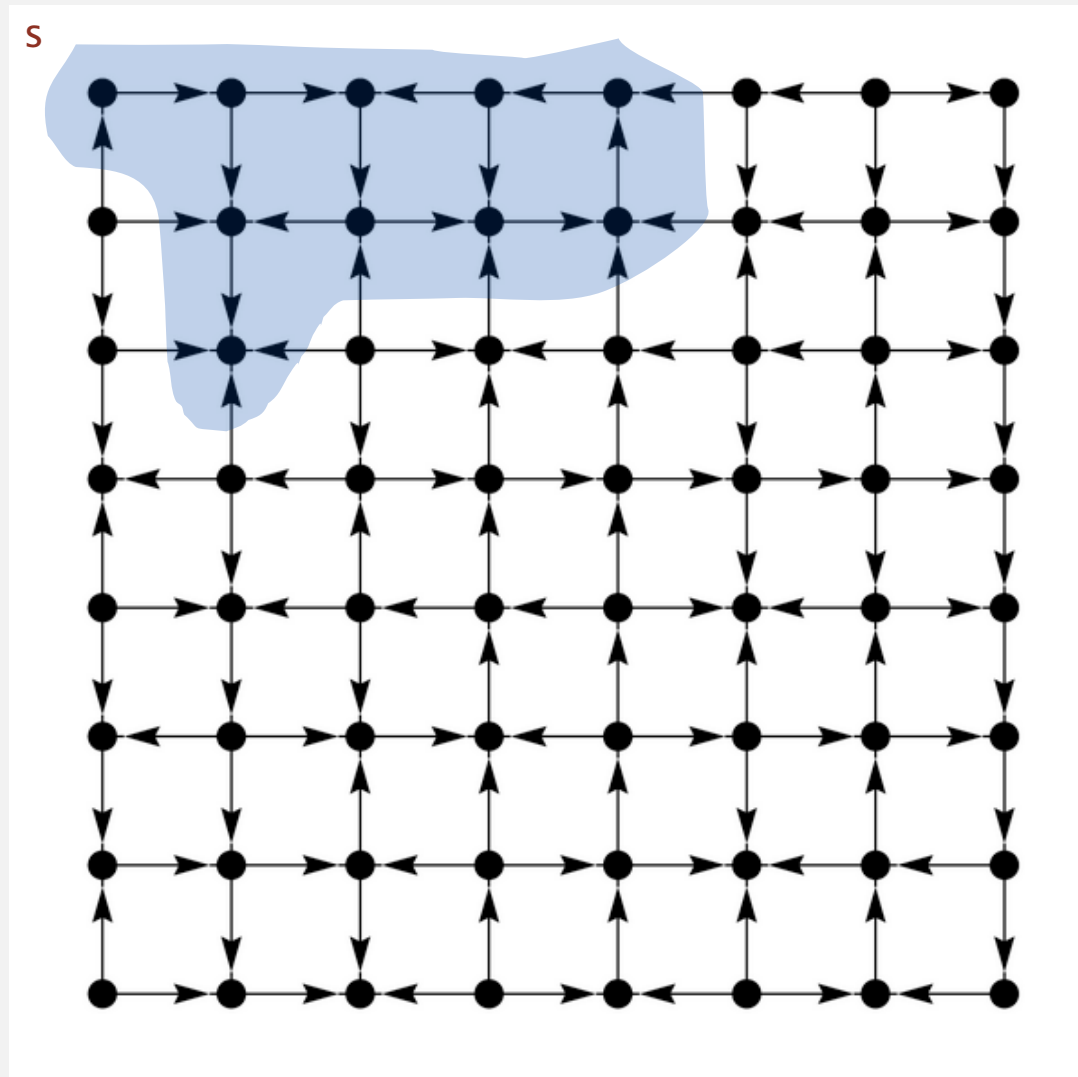
<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

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

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.

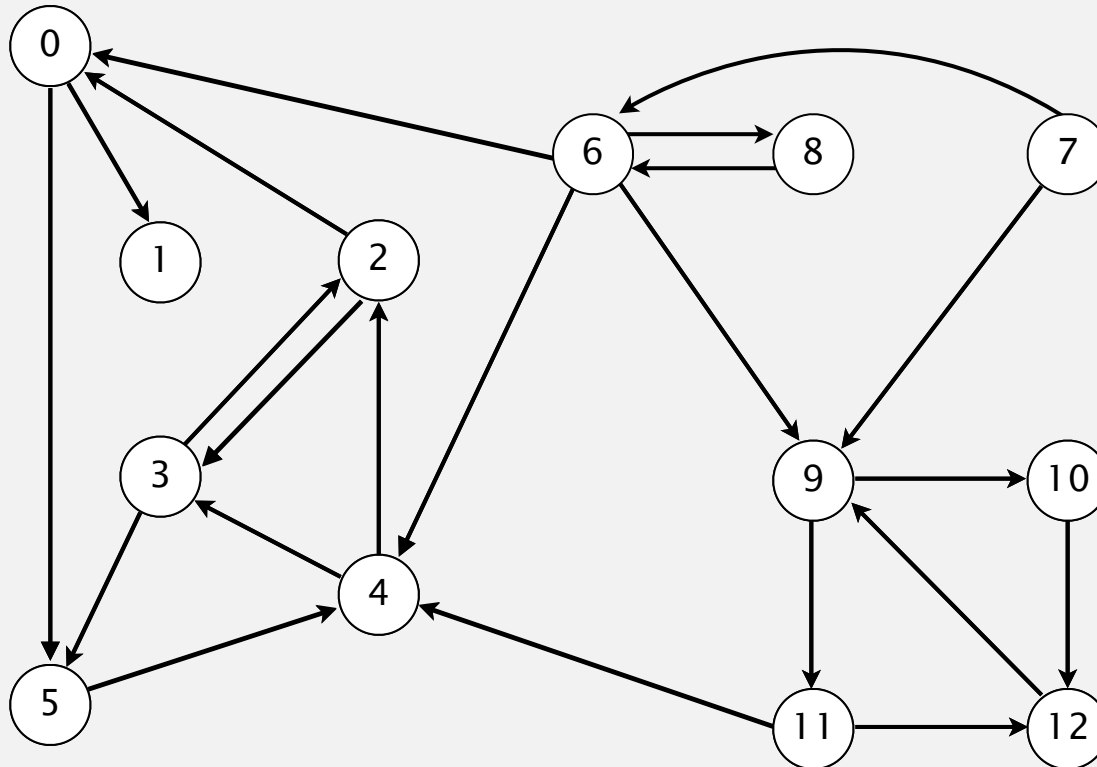
Recursively visit all unmarked
vertices w pointing from v .

Depth-first search demo

To visit a vertex v :



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



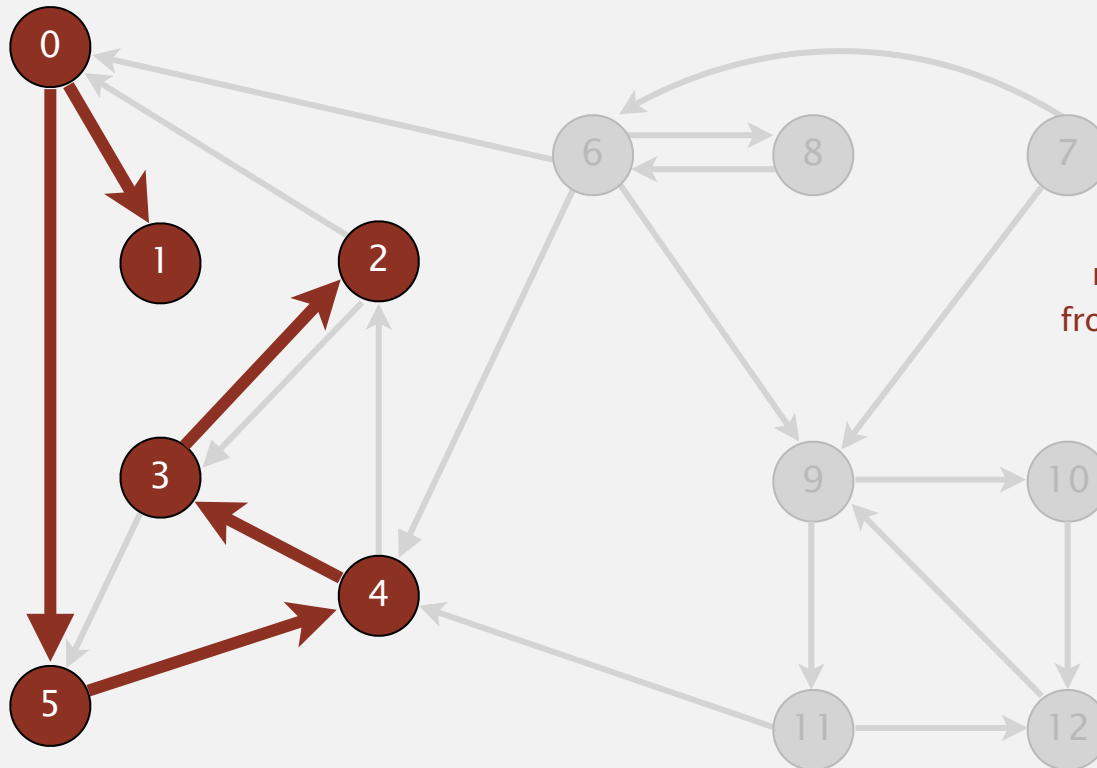
a directed graph

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 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;
```

← true if path to s

```
    public DepthFirstSearch(Graph G, int s)
```

```
    {
```

```
        marked = new boolean[G.V()];
```

← constructor marks
vertices connected to s

```
        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);
```

← recursive DFS does the work

```
    }
```

```
    public boolean visited(int v)
```

```
    { return marked[v]; }
```

← client can ask whether any
vertex is connected to s

```
}
```

Depth-first search (in directed graphs)

Code for **directed** graphs identical to undirected one.
[substitute Digraph for Graph]

```
public class DirectedDFS
```

```
{
```

```
    private boolean[] marked;
```

← true if path from s

```
    public DirectedDFS(Digraph G, int s)
```

```
    {
```

```
        marked = new boolean[G.V()];
```

```
        dfs(G, s);
```

```
    }
```

← constructor marks
vertices reachable from s

```
    private void dfs(Digraph G, int v)
```

```
    {
```

```
        marked[v] = true;
```

```
        for (int w : G.adj(v))
```

```
            if (!marked[w]) dfs(G, w);
```

```
    }
```

← recursive DFS does the work

```
    public boolean visited(int v)
```

```
    { return marked[v]; }
```

```
}
```

← client can ask whether any
vertex is reachable from s

Reachability application: program control-flow analysis

Every program is a digraph.

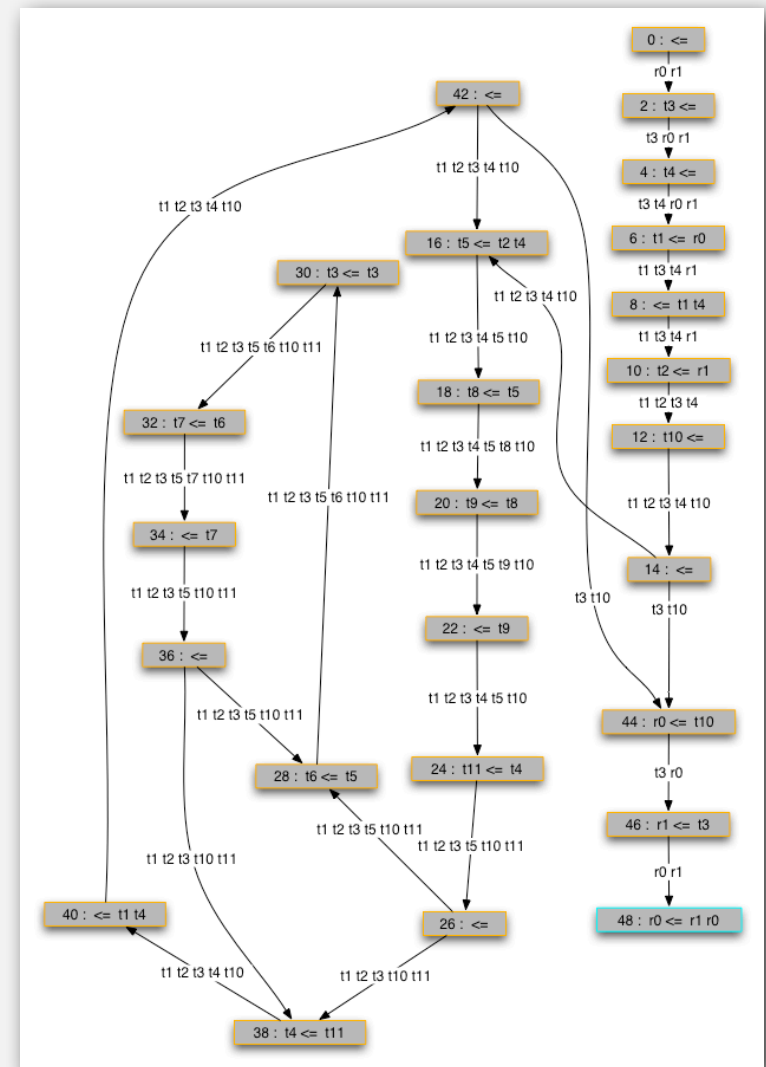
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



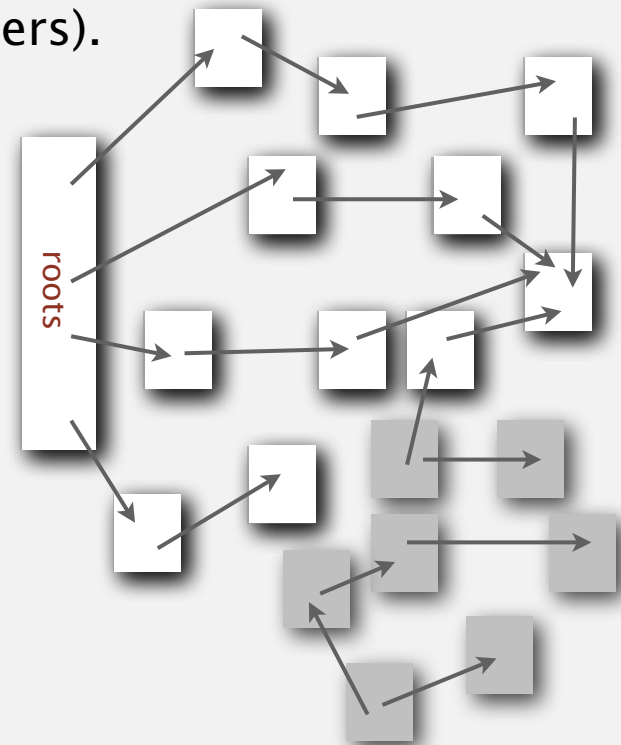
Reachability application: mark-sweep garbage collector

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).

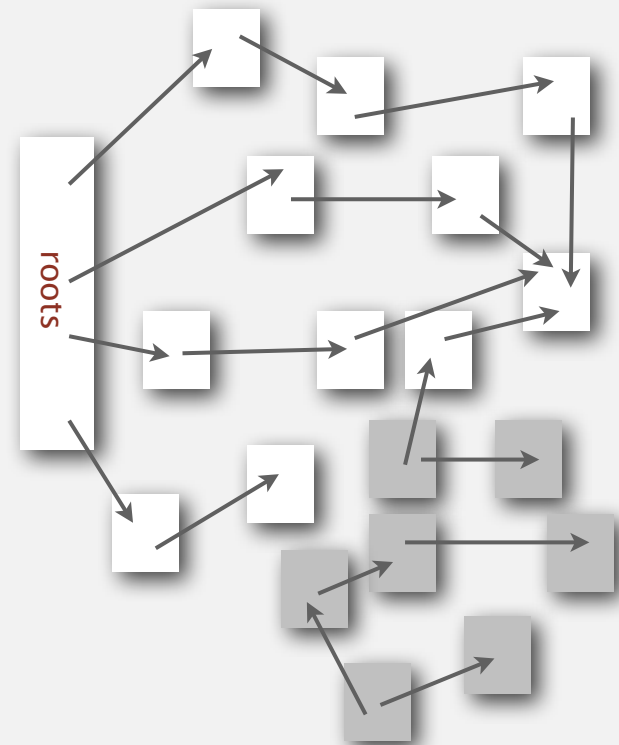


Reachability application: mark-sweep garbage collector

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.

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.
-

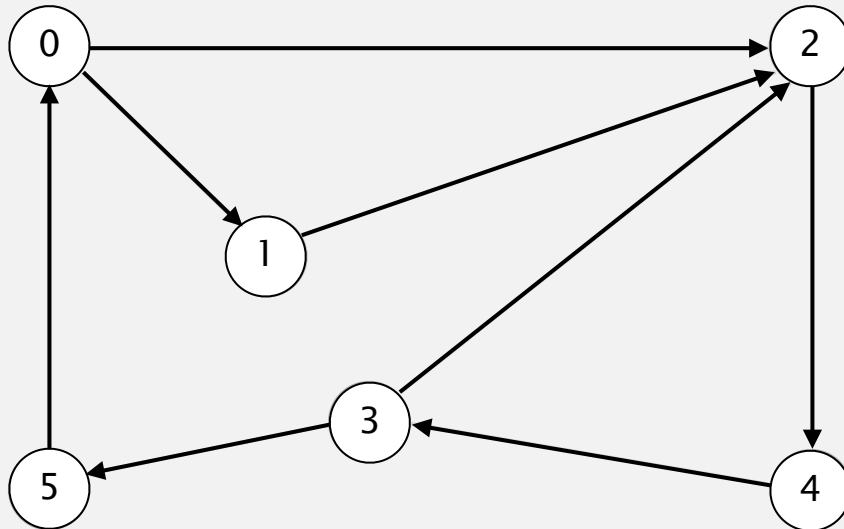
Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E + V$.

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.



tinyDG2.txt

V → 6
8 ← E

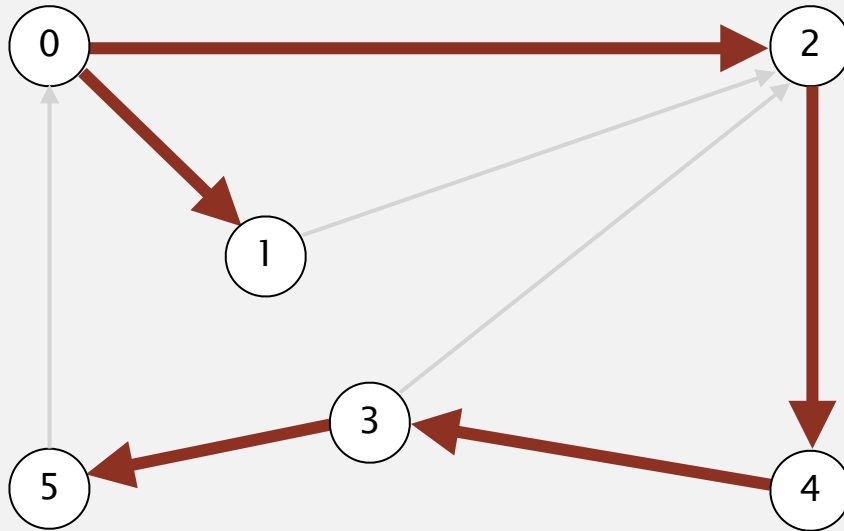
5 0
2 4
3 2
1 2
0 1
4 3
3 5
0 2

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

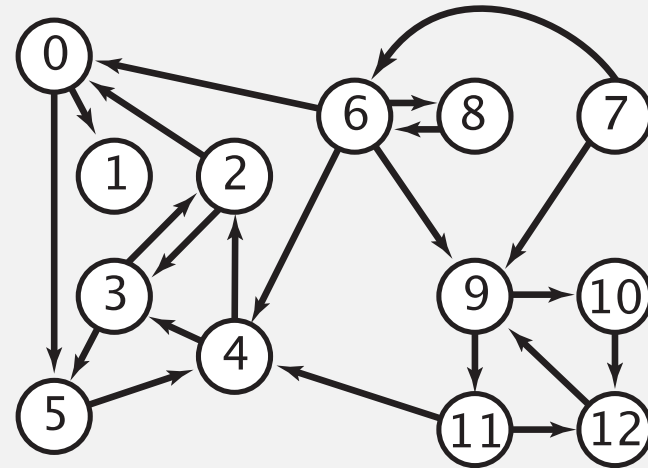
done

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 $7 \rightarrow 6 \rightarrow 4$.
- 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?

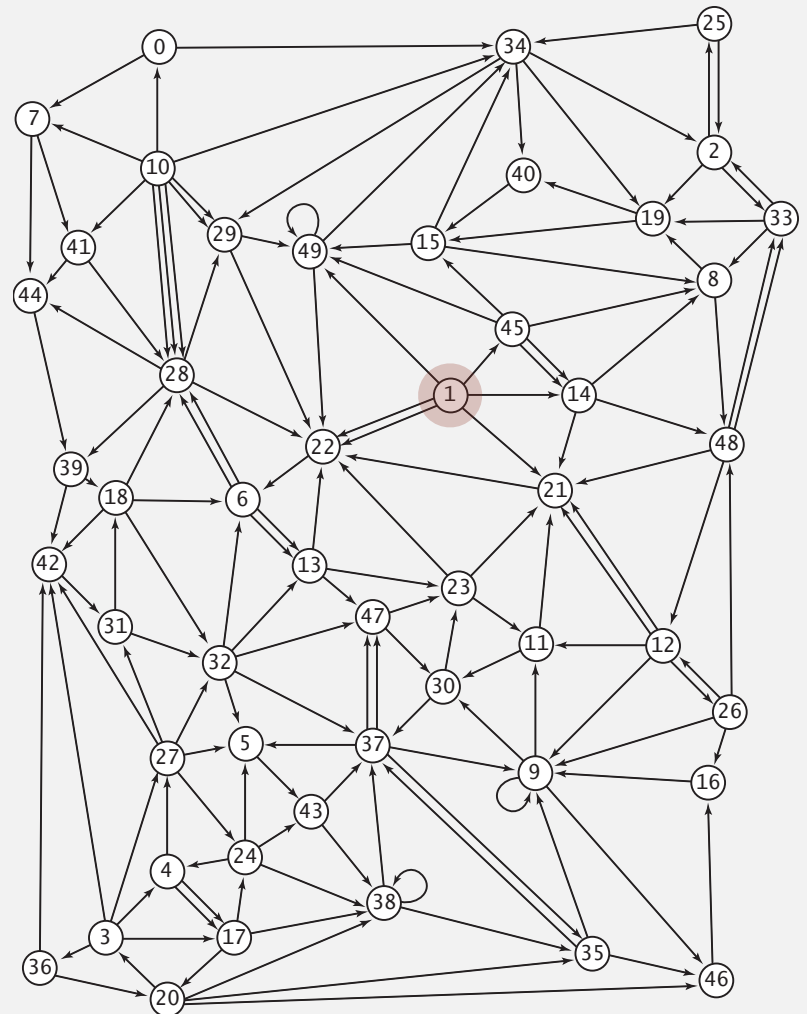
A. Use BFS, but initialize by enqueueing all source vertices.

Breadth-first search in digraphs application: web crawler

Goal. Crawl web, starting from some root web page, say `www.isical.ac.in`.

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>();
```

← queue of websites to crawl
← set of marked websites

```
String root = "http://www.isical.ac.in";  
queue.enqueue(root);  
marked.add(root);
```

← start crawling from root website

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

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

← read in raw html from next
website in queue

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

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

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

← if unmarked, mark it and put
on the queue

```
    }  
}
```



<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*

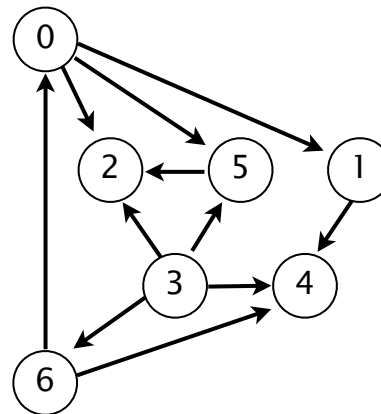
Precedence scheduling

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

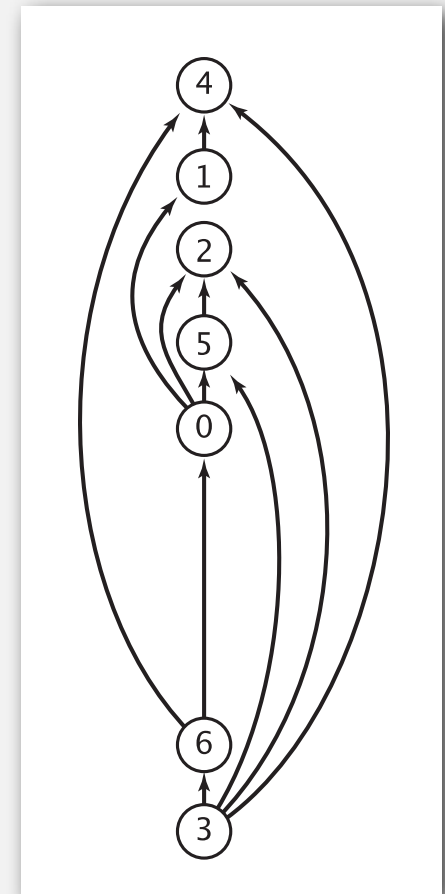
Digraph model. vertex = task; edge = precedence constraint.

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

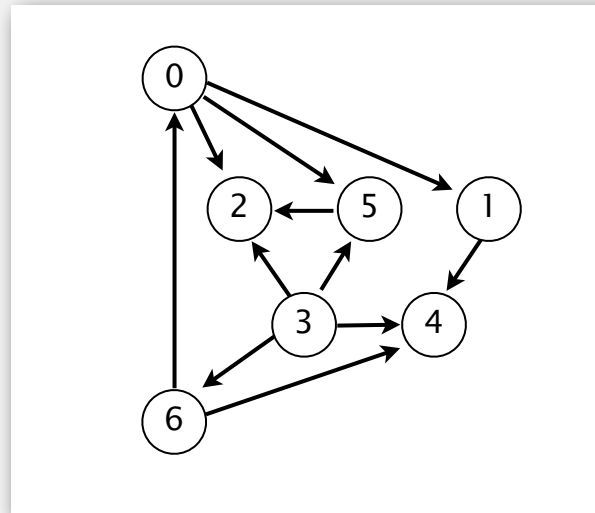
Topological sort

DAG. Directed **acyclic** graph.

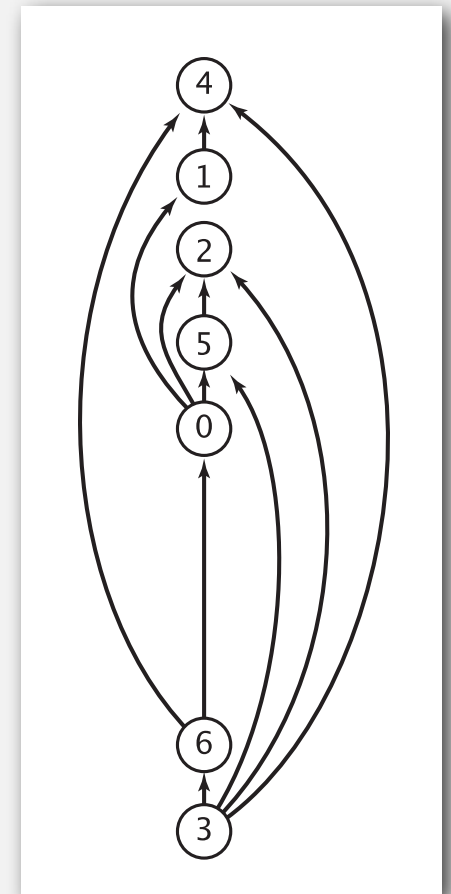
Topological sort. Redraw DAG so all edges point upwards.

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

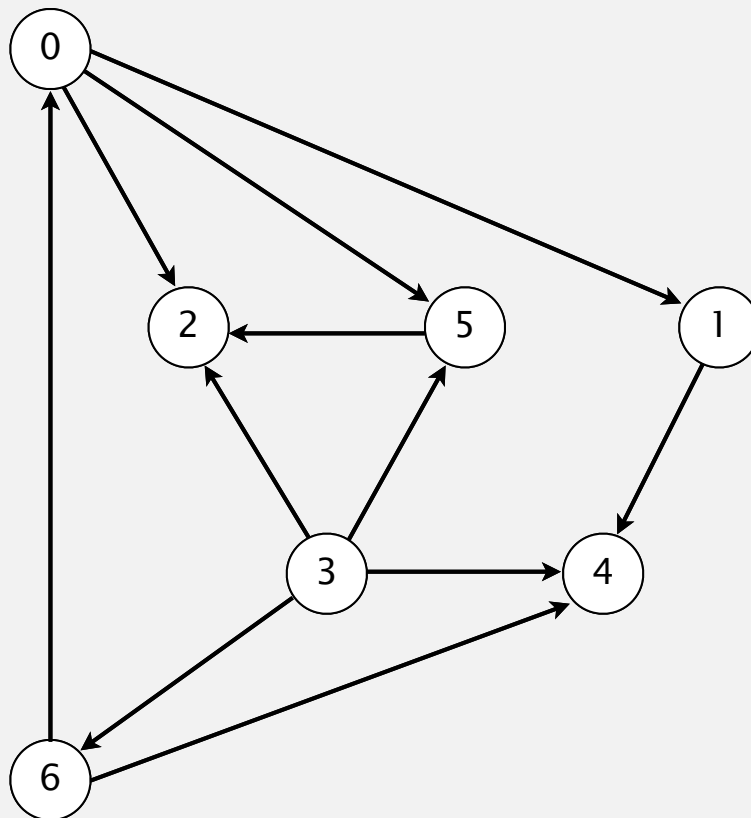


topological order

Solution. DFS. What else?

Topological sort demo

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

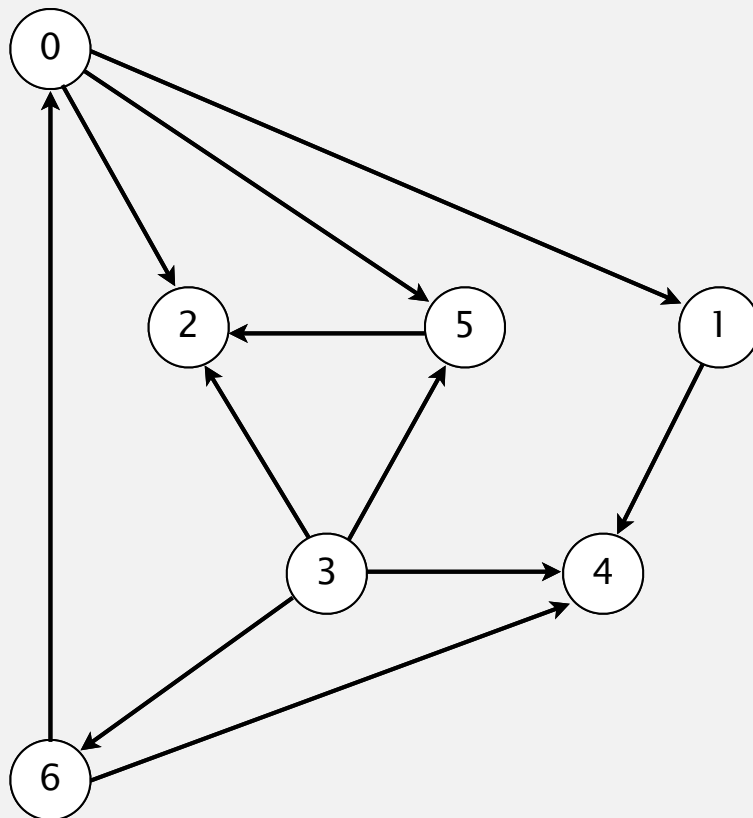


0→5
0→2
0→1
3→6
3→5
3→4
5→2
6→4
6→0
3→2
1→4

a directed acyclic graph

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> reversePost;

    public DepthFirstOrder(Digraph G)
    {
        reversePost = 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);
        reversePost.push(v);
    }

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

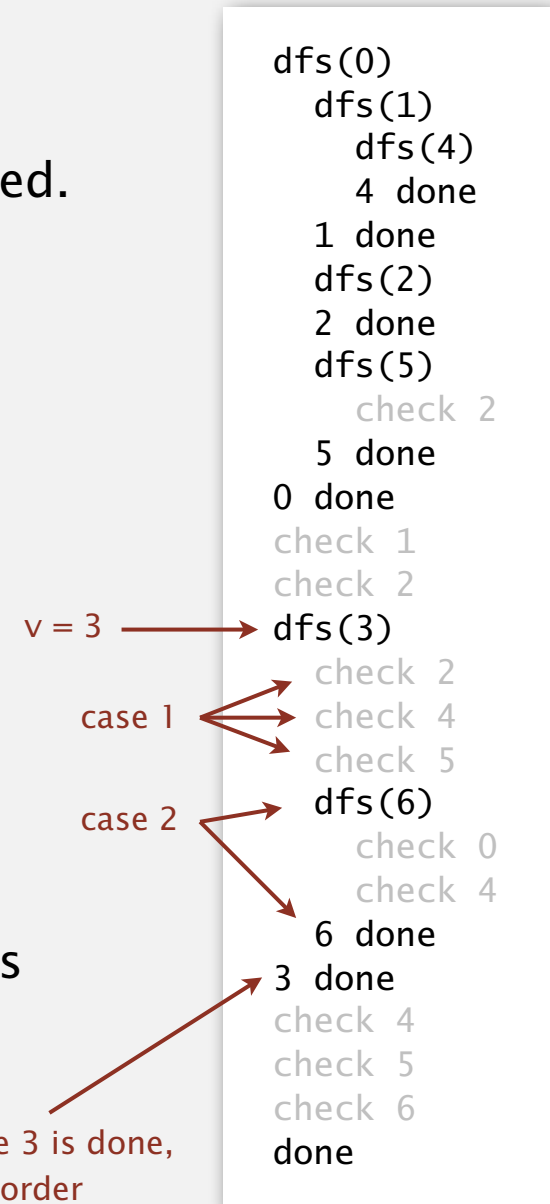
← returns all vertices in
“reverse DFS postorder”

Topological sort in a DAG: correctness proof

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

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

- Case 1: $\text{dfs}(w)$ has already been called and returned.
Thus, w was done before v .
- Case 2: $\text{dfs}(w)$ has not yet been called.
 $\text{dfs}(w)$ will get called directly or indirectly
by $\text{dfs}(v)$ and will finish before $\text{dfs}(v)$.
Thus, w will be done before v .
- Case 3: $\text{dfs}(w)$ 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 $v \rightarrow w$ would complete a cycle.

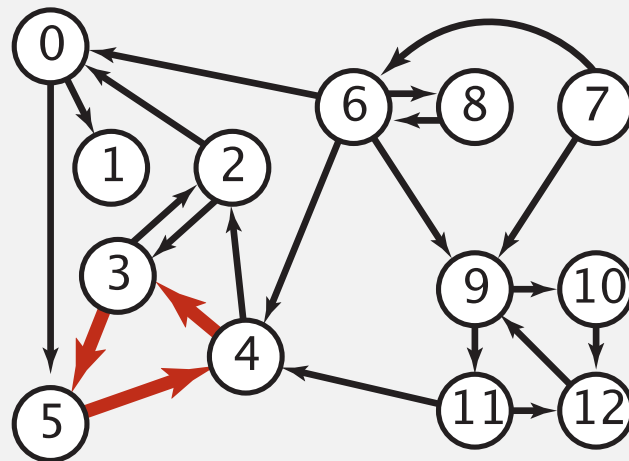


Directed cycle detection

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?

PAGE 3

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

<http://xkcd.com/754>

Remark. A directed cycle implies scheduling problem is infeasible.

Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

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

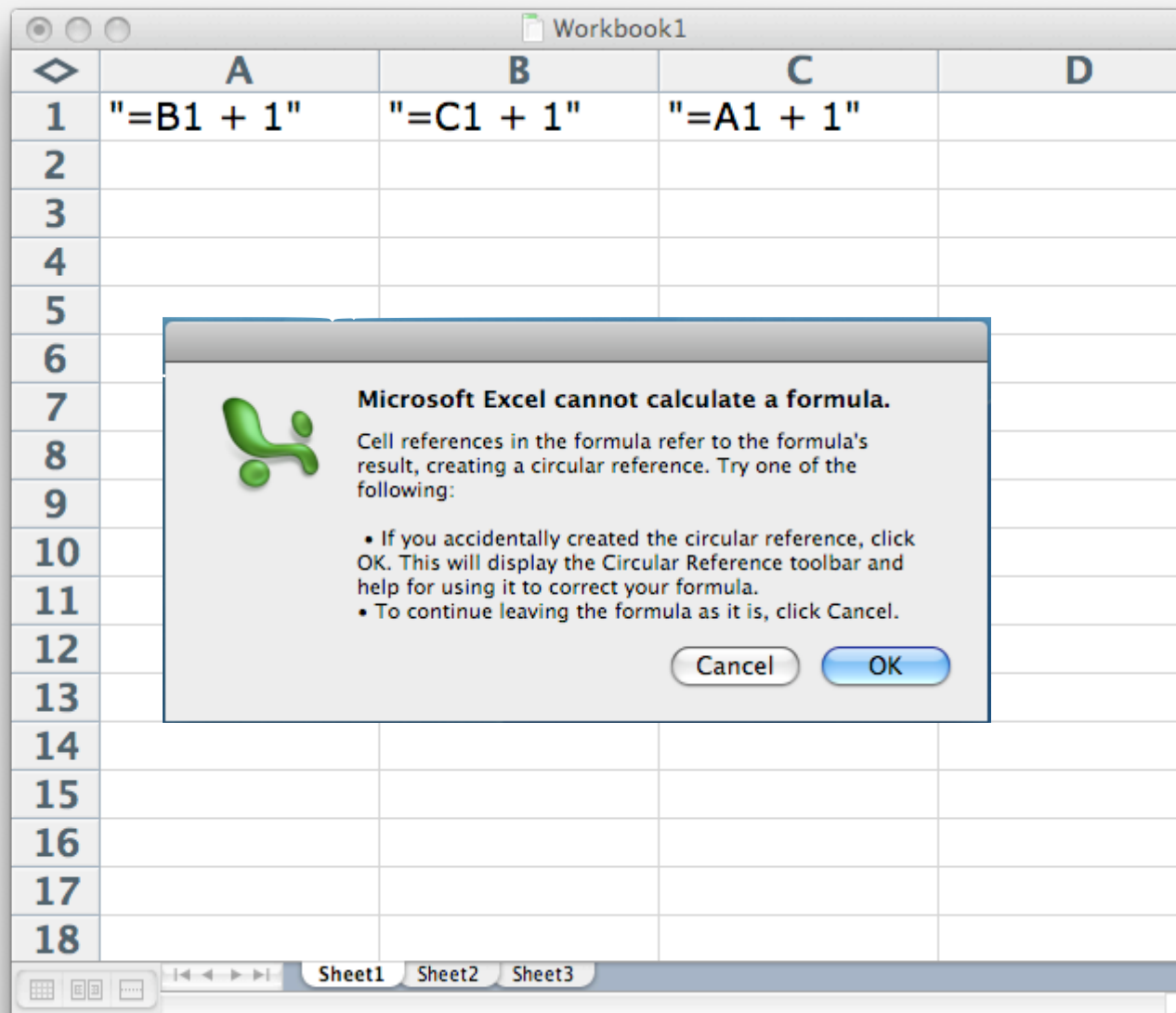
```
public class B extends C
{
    ...
}
```

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

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

Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection (and has a circular reference toolbar!)





<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***

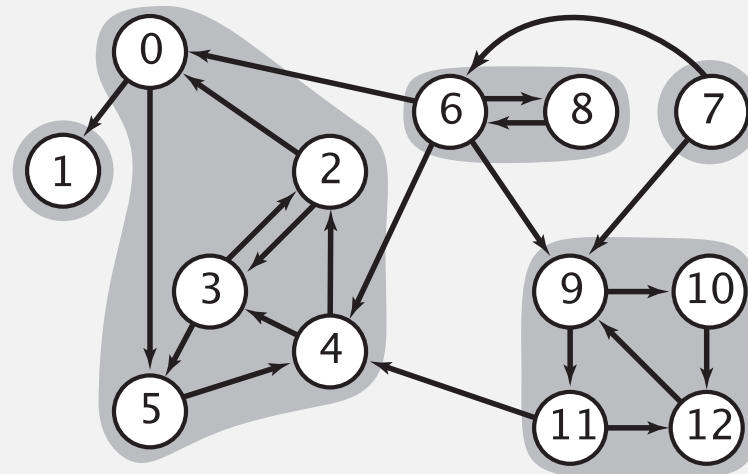
Strongly-connected components

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**:

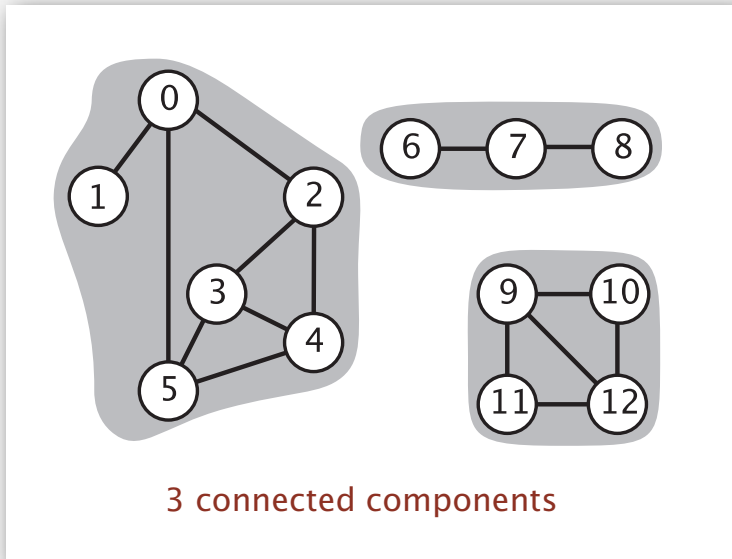
- v is strongly connected to v .
- If v is strongly connected to w , then w is strongly connected to v .
- If v is strongly connected to w and w to x , then v is strongly connected to x .

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



Connected components vs. strongly-connected components

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



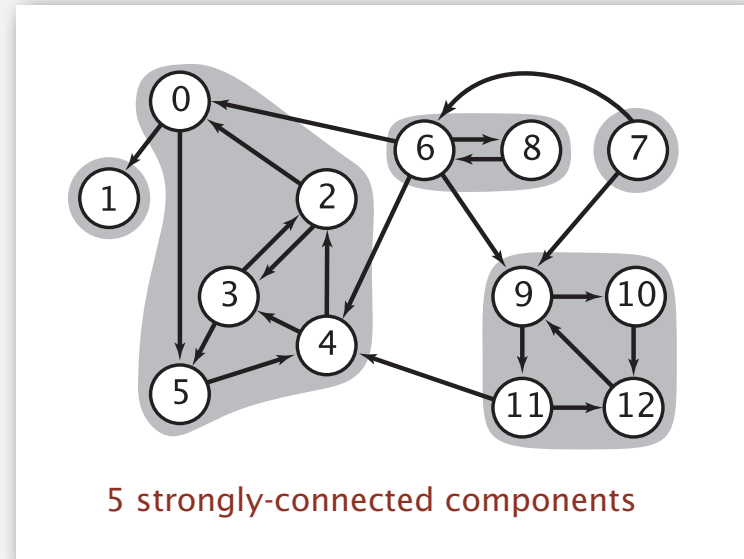
connected component id (easy to compute with DFS)

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

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



strongly-connected component id (how to compute?)

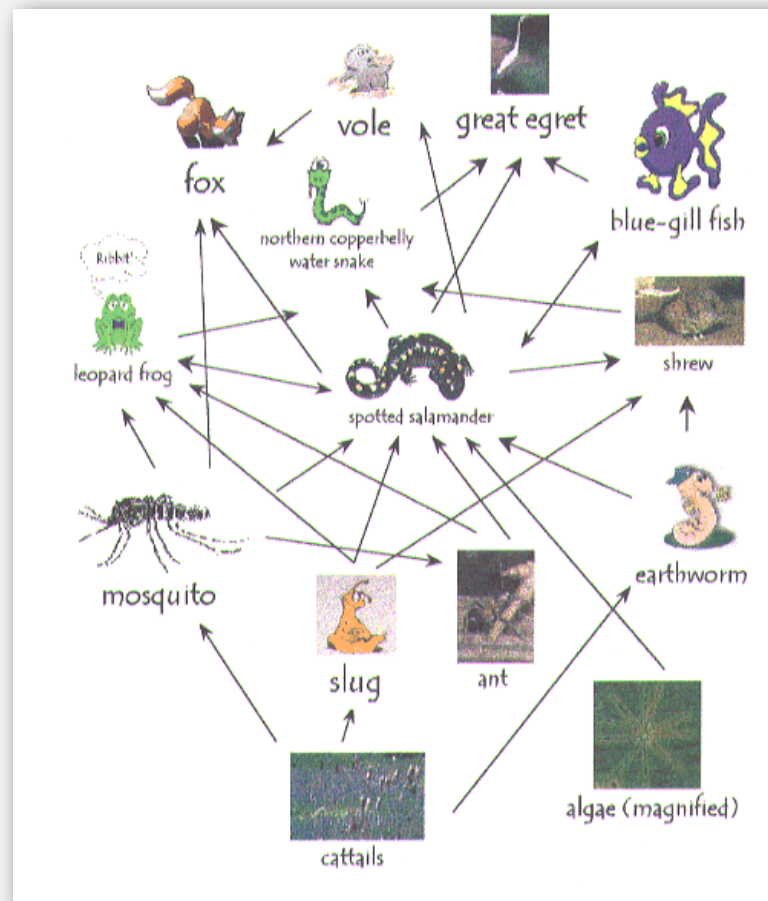
	0	1	2	3	4	5	6	7	8	9	10	11	12
scc[]	1	0	1	1	1	1	3	4	3	2	2	2	2

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

constant-time client strong-connectivity query

Strong component application: ecological food webs

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



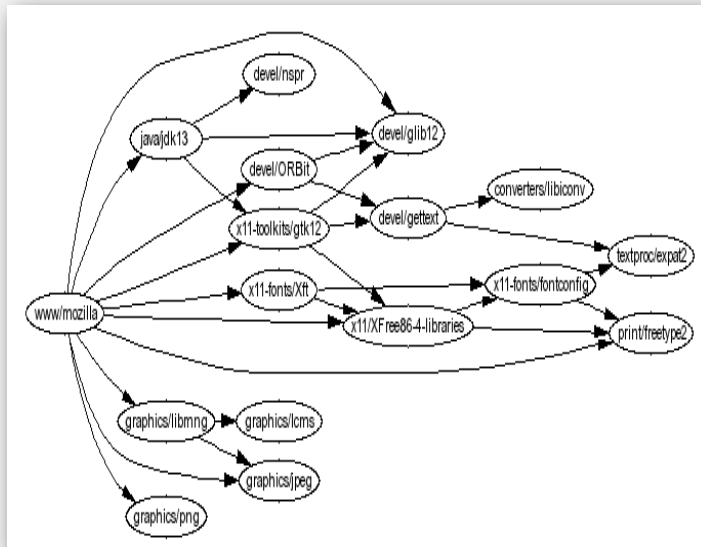
<http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif>

Strong component. Subset of species with common energy flow.

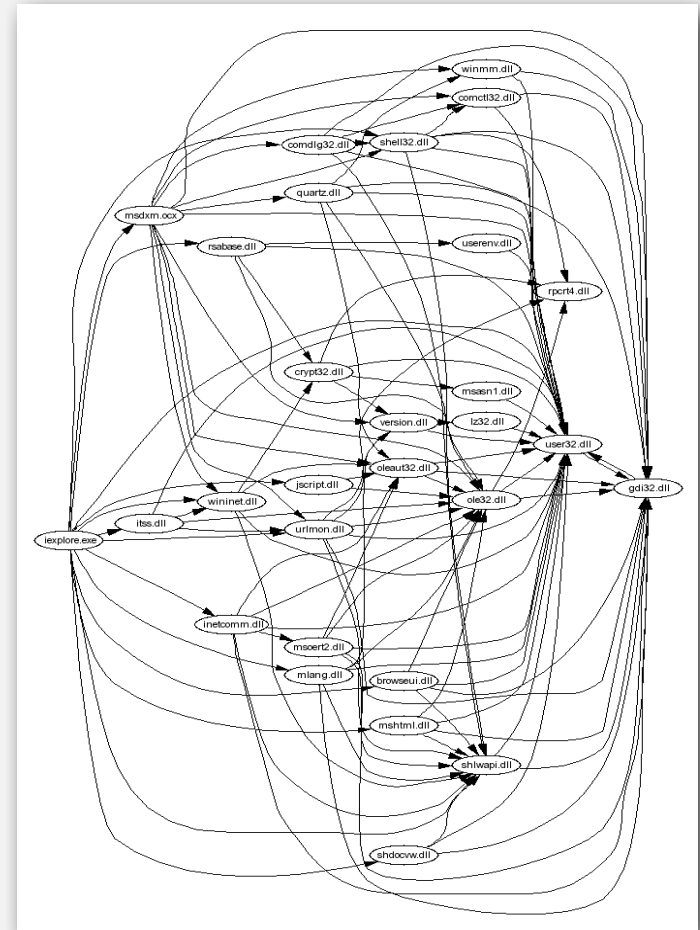
Strong component application: software modules

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

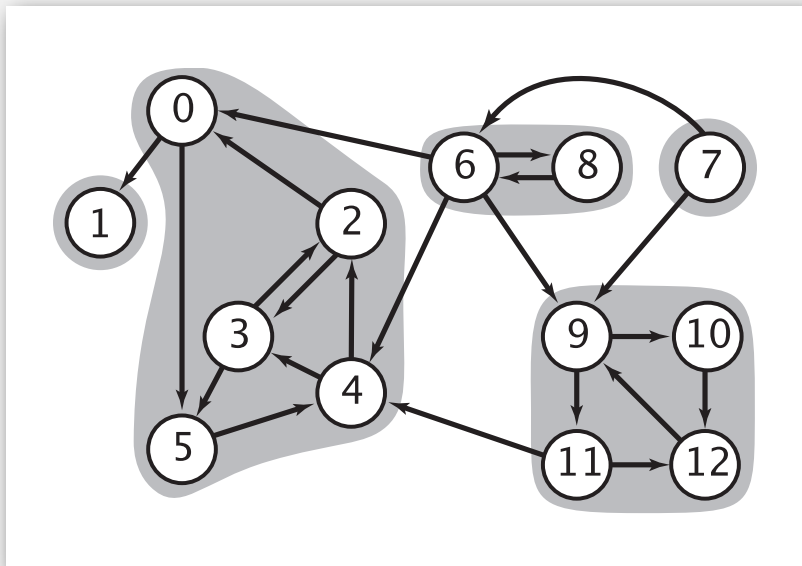
Reverse graph. Strong components in G are same as in G^R .

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

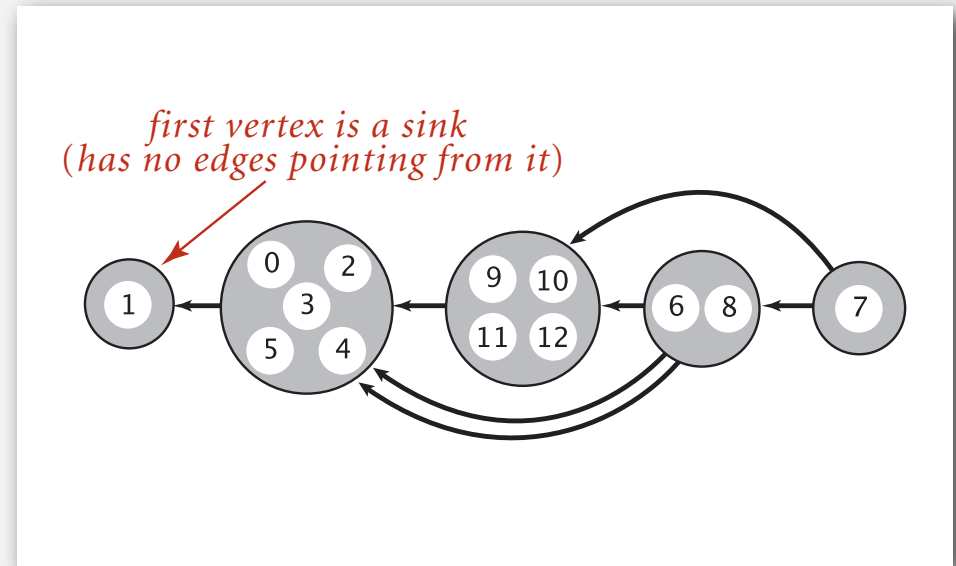
Idea.

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

how to compute?
←



digraph G and its strong components

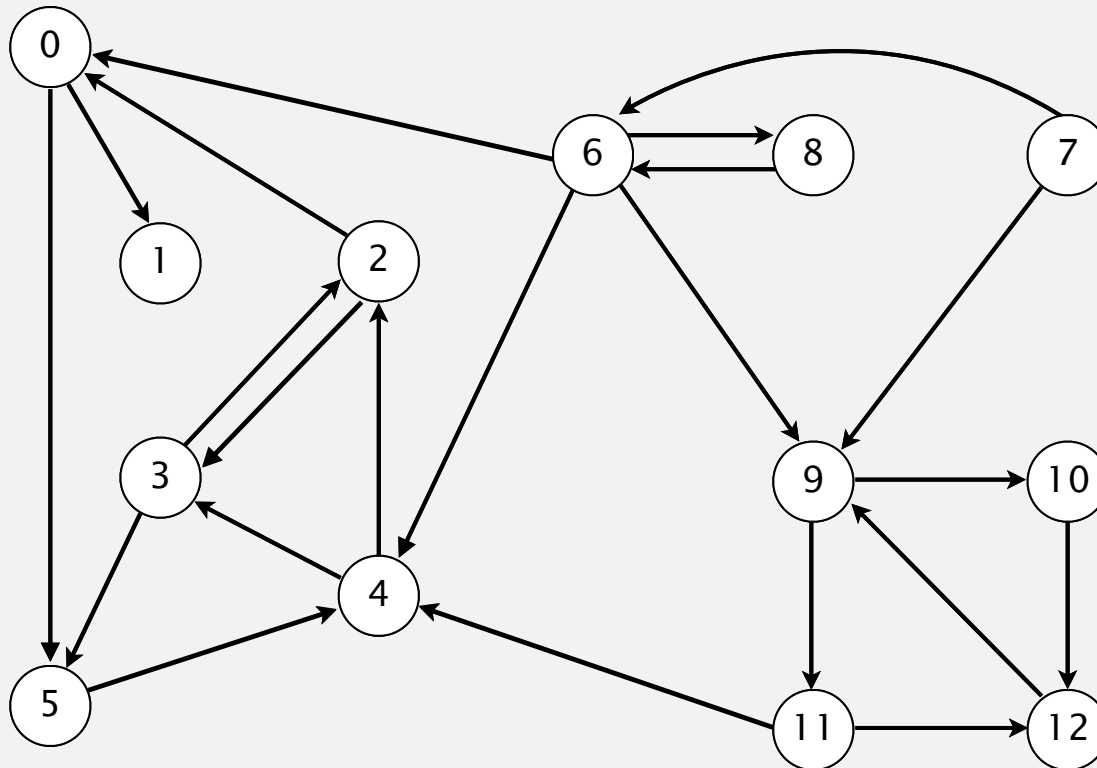


kernel DAG of G (in reverse topological order)

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 .

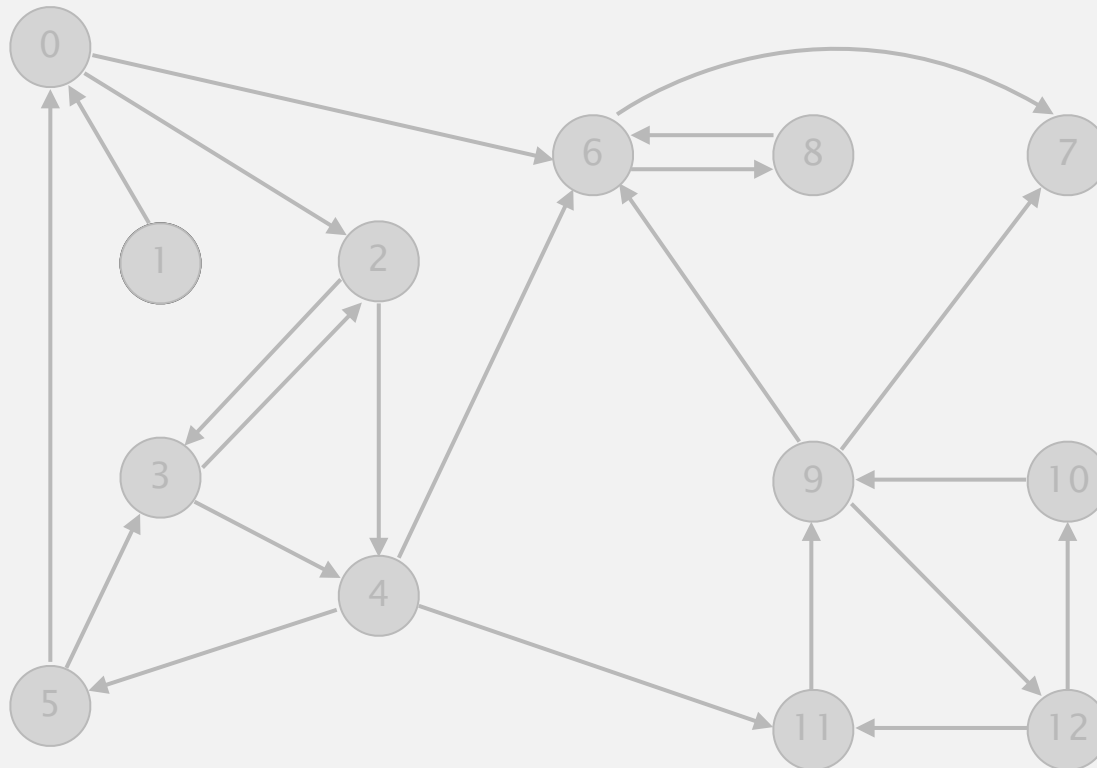


digraph G

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

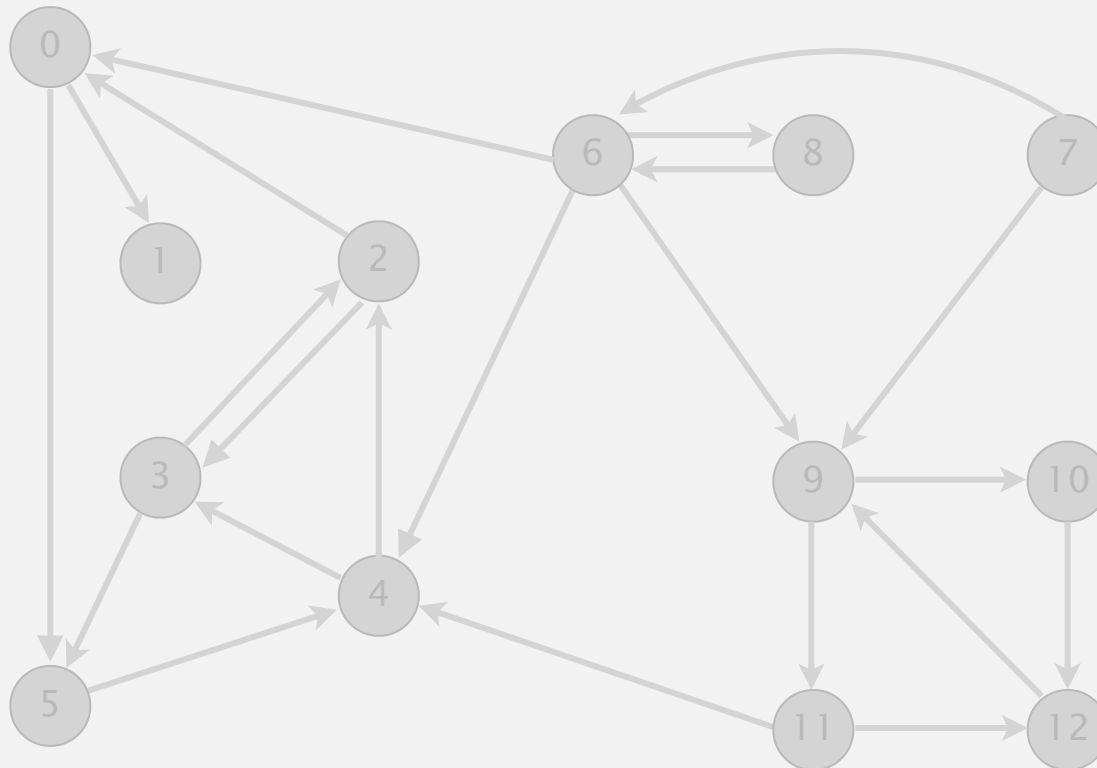


reverse digraph G^R

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	scc[]
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

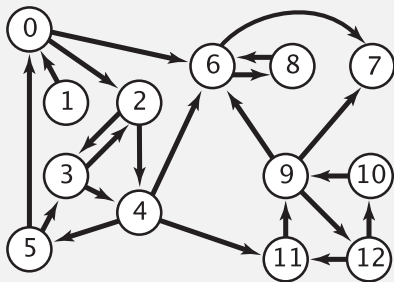
done

Kosaraju-Sharir algorithm

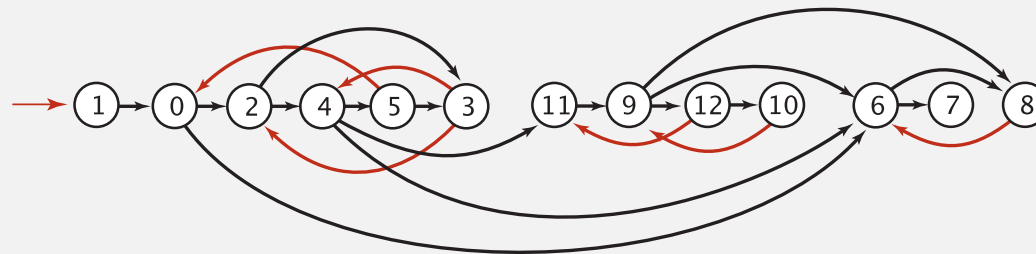
Simple (but mysterious) algorithm for computing strong components.

- 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

```

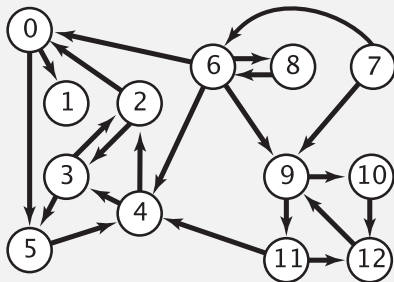
dfs(0)
  dfs(6)
    dfs(8)
      | check 6
      8 done
    dfs(7)
      7 done
    6 done
  dfs(2)
    dfs(4)
      dfs(11)
        dfs(9)
          dfs(12)
            | check 11
            dfs(10)
              | check 9
              10 done
            12 done
          check 7
        check 6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

Kosaraju-Sharir algorithm

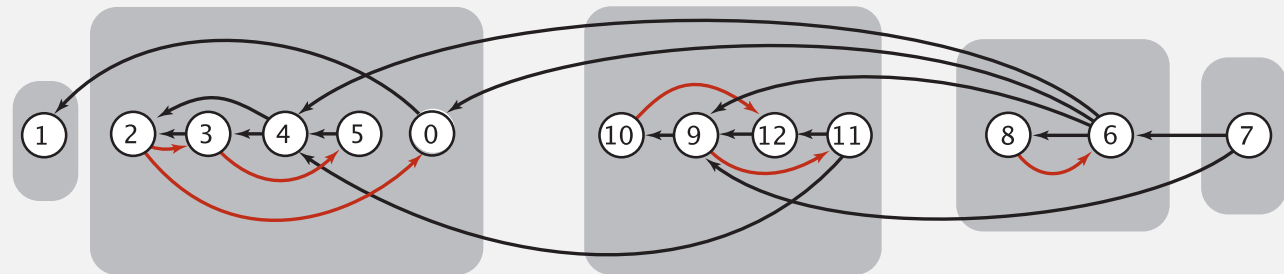
Simple (but mysterious) algorithm for computing strong components.

- 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)

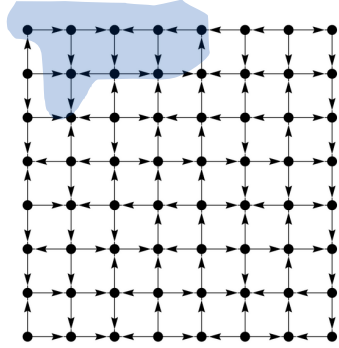
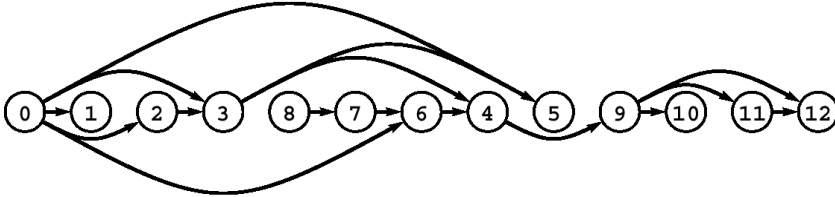
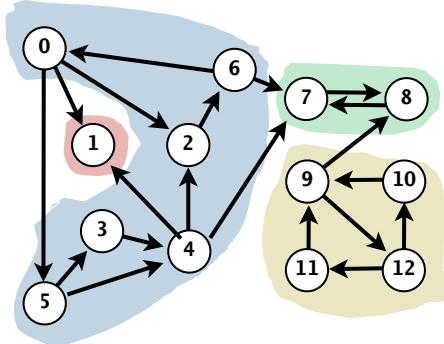
```
public class KosarajuSharirSCC
{
    private boolean marked[];
    private int[] id;
    private int count;

    public KosarajuSharirSCC(Digraph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
        for (int v : dfs.reversePost())
        {
            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

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



<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*