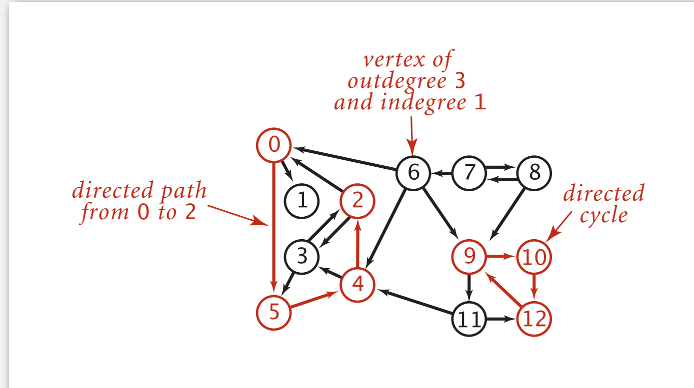
4.2 Directed Graphs

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

Algorithms, 4th Edition · *Robert Sedgewick and Kevin Wayne* · *Copyright © 2002–2010* · *November 8, 2010 8:43:22 PM*

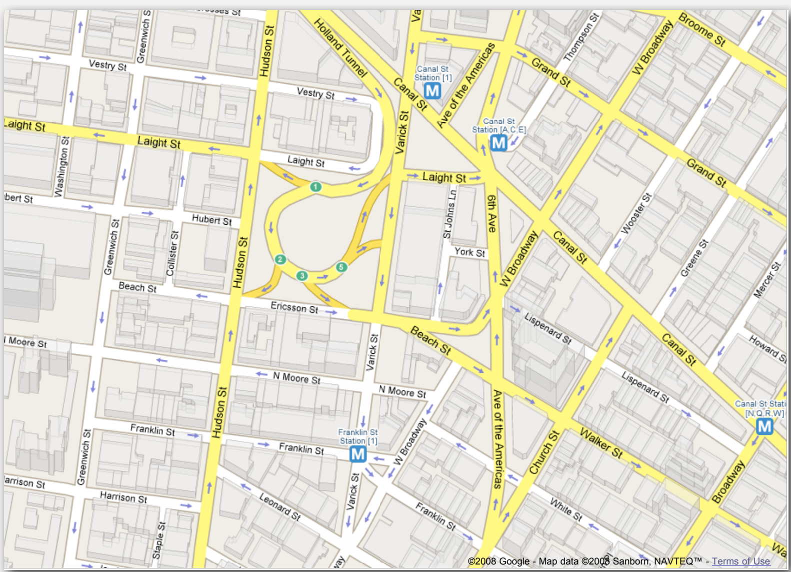
Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.



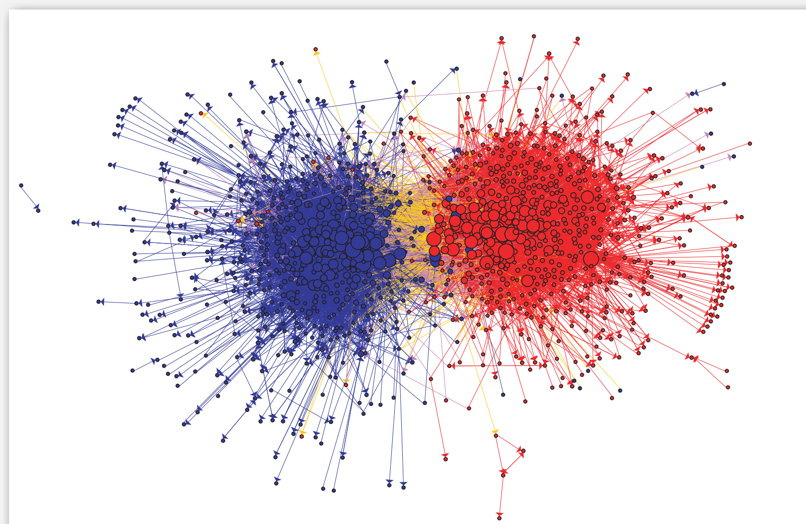
Road network

Vertex = intersection; edge = one-way street.



Political blogosphere graph

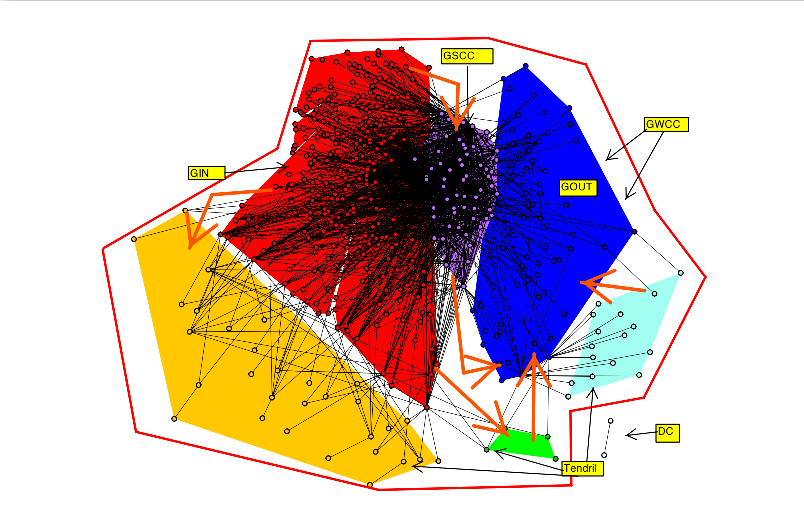
Vertex = political blog; edge = link.



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

Overnight interbank loan graph

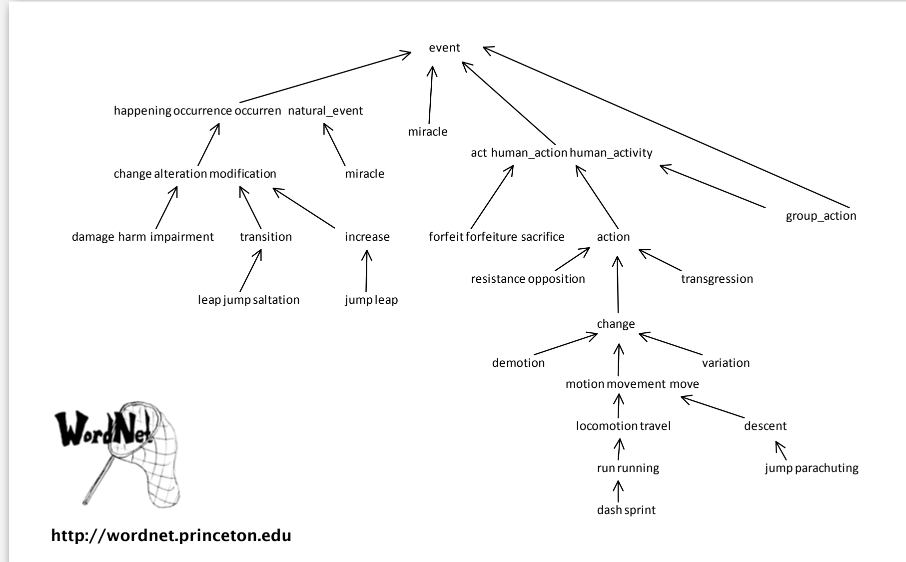
Vertex = bank; edge = overnight loan.



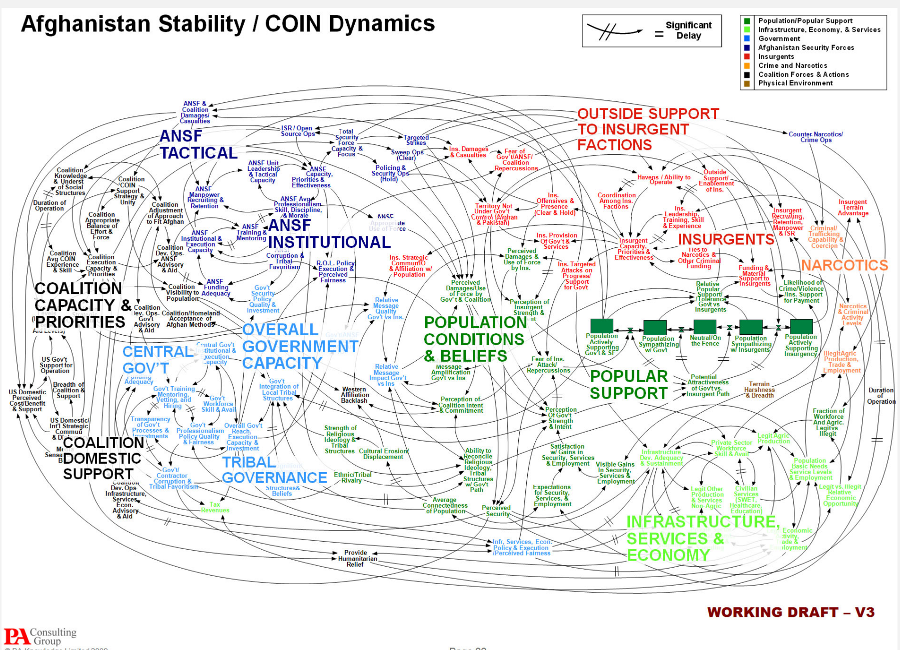
The Topology of the Federal Funds Market, Bech and Atalay, 2008

WordNet graph

Vertex = synset; edge = hypernym relationship.



The McChrystal Afghanistan PowerPoint slide



http://www.guardian.co.uk/news/datablog/2010/apr/29/mcchrystal-afghanistan-powerpoint-slide

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 the digraph so that all edges point down?

Strong connectivity. Are all vertices mutually reachable?

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?

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

Digraph 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 $v \to w$*
`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*

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

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

read digraph from input stream

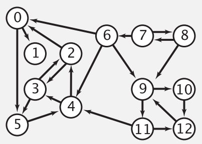
print out each edge (once)

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
 8  9
10 12
11  4
 4  3
 3  5
 7  8
 8  7
 5  4
 0  5
 6  4
 6  9
 7  6
```



```
% java TestDigraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
4->3
4->2
5->4
6->9
6->4
6->0
...
11->4
11->12
12-9
```

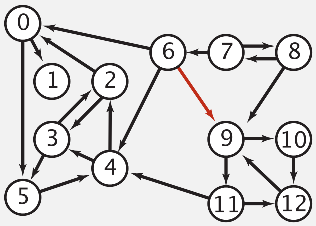
```
In in = new In(args[0]);
Digraph G = new Digraph(in);

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

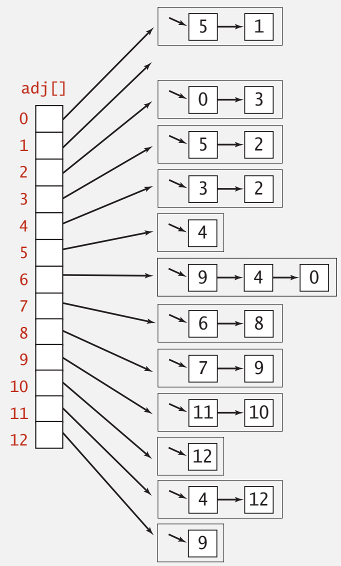
read digraph from input stream

print out each edge (once)

Adjacency-list digraph representation

Maintain vertex-indexed array of lists (use `Bag` abstraction).





Adjacency-lists digraph representation: Java implementation

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

```
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 graph with V vertices

add edge from v to w

iterator for vertices adjacent from v

Digraph representations

In practice. Use adjacency-list representation.

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

huge number of vertices, small average vertex degree

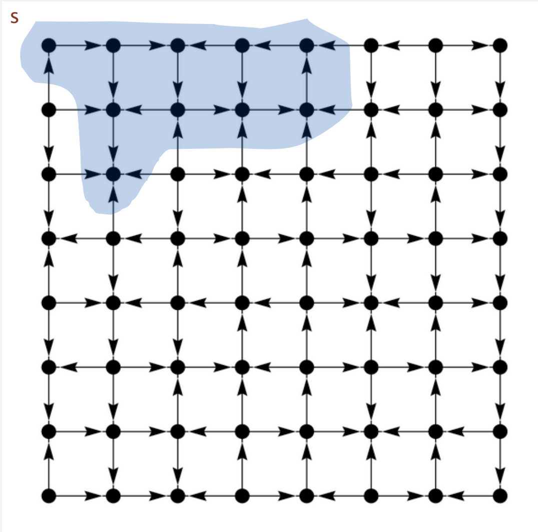
representation	space	insert edge from v to w	edge from v to w?	iterate over vertices adjacent from v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 †	1	V
adjacency list	E + V	1	outdegree(v)	outdegree(v)

† disallows parallel edges

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

Reachability

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



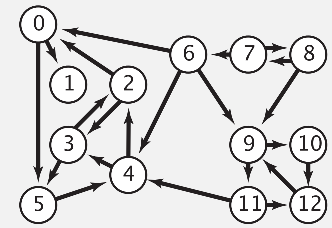
Depth-first search in digraphs

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 adjacent to v.



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 path 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)

Digraph version identical to undirected one (substitute `Digraph` for `Graph`).

```
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.

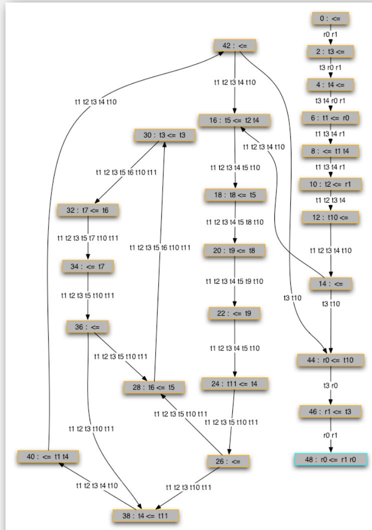
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



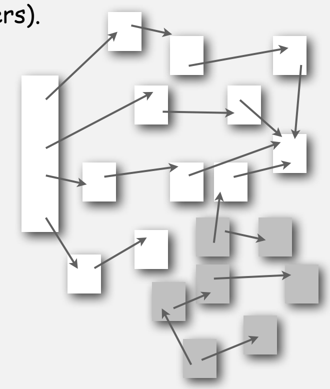
Reachability application: mark-sweep garbage collector

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).

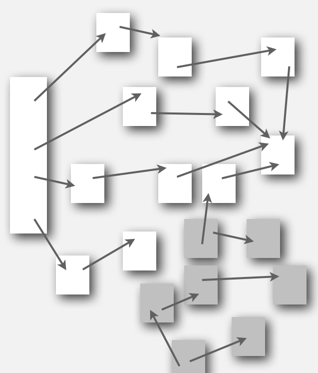


Reachability application: mark-sweep garbage collector

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.
- Transitive closure.

Basis for solving difficult digraph problems.

- Directed Euler path.
- Strongly-connected components.

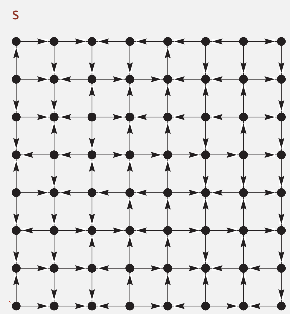
Breadth-first search in digraphs

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
- add each of v's unvisited neighbors to the queue, and mark them as visited.



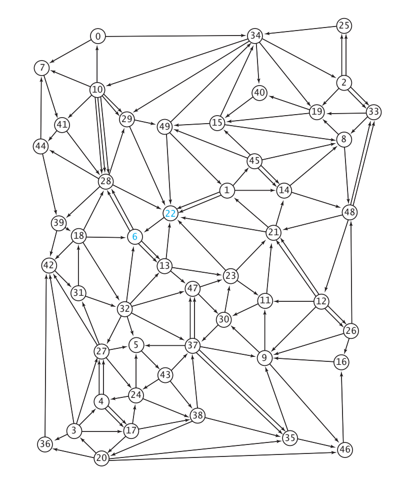
Proposition. BFS computes shortest paths (fewest number of edges).

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 graph.

BFS.

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

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

while (!q.isEmpty())
{
   String v = q.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 (!visited.contains(w))
      {
         visited.add(w);
         q.enqueue(w);
      }
   }
}
```

queue of websites to crawl
set of visited websites

start crawling from website s

read in raw html from next website in queue

use regular expression to find all URLs in website of form `http://xxx.yyy.zzz`

if unvisited, mark as visited and put on queue

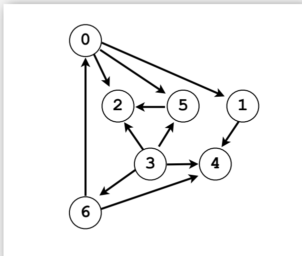
Precedence scheduling

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

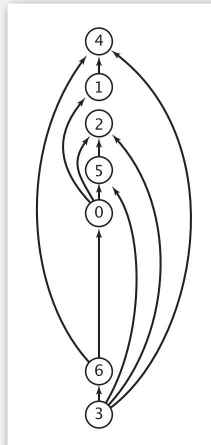
Graph model. vertex = task; edge = precedence constraint.

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

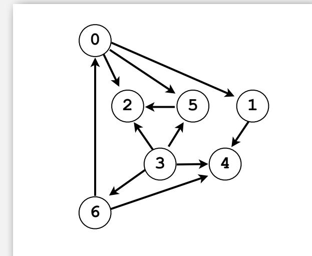
Topological sort

DAG. Directed acyclic graph.

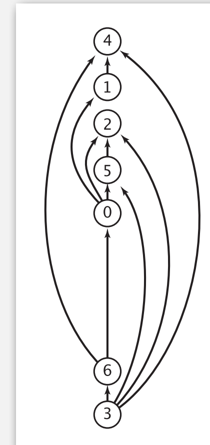
Topological sort. Redraw DAG so all edges point up.

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

directed edges



DAG



topological order

Solution. DFS. What else?

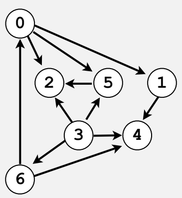
Topological sort demo

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;  }
}
```

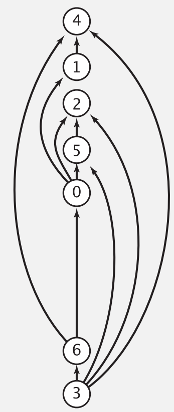
returns all vertices in "reverse DFS postorder"

Reverse DFS postorder in a DAG



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

	marked[]	reversePost
dfs(0)	1 0 0 0 0 0 0	–
dfs(1)	1 1 0 0 0 0 0	–
dfs(4)	1 1 0 0 1 0 0	–
4 done	1 1 0 0 1 0 0	4
1 done	1 1 0 0 1 0 0	4 1
dfs(2)	1 1 1 0 1 0 0	4 1
2 done	1 1 1 0 1 0 0	4 1 2
dfs(5)	1 1 1 0 1 1 0	4 1 2
check 2	1 1 1 0 1 1 0	4 1 2
5 done	1 1 1 0 1 1 0	4 1 2 5
0 done	1 1 1 0 1 1 0	4 1 2 5 0
check 1	1 1 1 0 1 1 0	4 1 2 5 0
check 2	1 1 1 0 1 1 0	4 1 2 5 0
dfs(3)	1 1 1 1 1 1 0	4 1 2 5 0
check 2	1 1 1 1 1 1 0	4 1 2 5 0
check 4	1 1 1 1 1 1 0	4 1 2 5 0
check 5	1 1 1 1 1 1 0	4 1 2 5 0
dfs(6)	1 1 1 1 1 1 1	4 1 2 5 0
6 done	1 1 1 1 1 1 1	4 1 2 5 0 6
3 done	1 1 1 1 1 1 1	4 1 2 5 0 6 3
check 4	1 1 1 1 1 1 0	4 1 2 5 0 6 3
check 5	1 1 1 1 1 1 0	4 1 2 5 0 6 3
check 6	1 1 1 1 1 1 0	4 1 2 5 0 6 3
done	1 1 1 1 1 1 1	4 1 2 5 0 6 3



reverse DFS postorder is a topological order!

Topological sort in a DAG: correctness proof

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

Pf. Consider any edge $v \to w$. When `dfs(G, v)` is called:

- Case 1: `dfs(G, w)` has already been called and returned. Thus, w was done before v.
- Case 2: `dfs(G, w)` has not yet been called. It will get called directly or indirectly by `dfs(G, v)` and will finish before `dfs(G, v)`. Thus, w will be done before v.
- Case 3: `dfs(G, w)` has already been called, but has not returned. Can't happen in a DAG: function call stack contains path from w to v, so $v \to w$ 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)
  6 done
3 done
check 4
check 5
check 6
done
```

Ex:

case 1

case 2

all vertices adjacent from 3 are done before 3 is done, so they all appear after 3

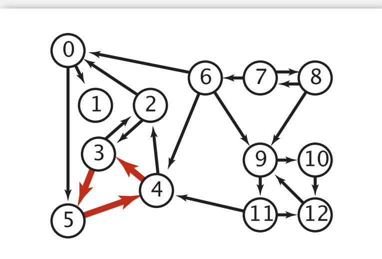
Directed cycle detection

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.

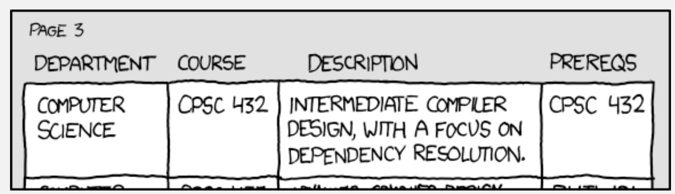
Goal. Given a digraph, find a directed cycle.



Solution. DFS. What else? See textbook for full details.

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
{
   ...
}
```

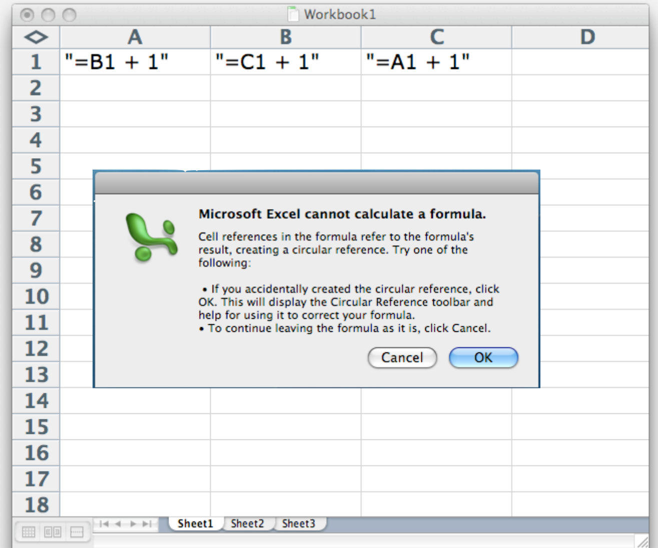
```
public class B extends C
{
   ...
}
```

```
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: 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
```


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

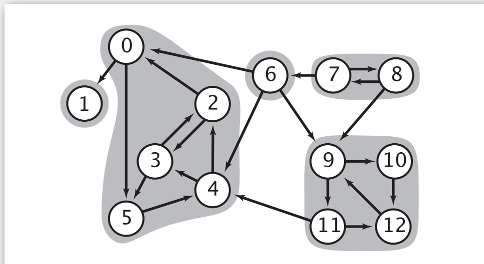
Strongly-connected components

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

Key property. Strong connectivity is an **equivalence relation**:

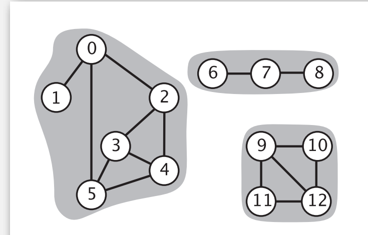
- v is strongly connected to v.
- If v is strongly connected to w, then w is strongly connected to v.
- If v is strongly connected to w and w to x, then v is strongly connected to x.

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



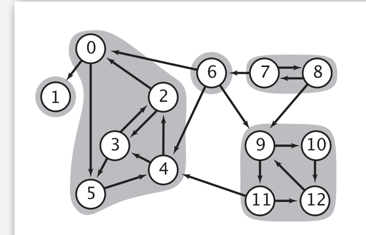
Connected components vs. strongly-connected components

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



3 connected components

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



5 strongly-connected components

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

strongly-connected component id (how to compute?)

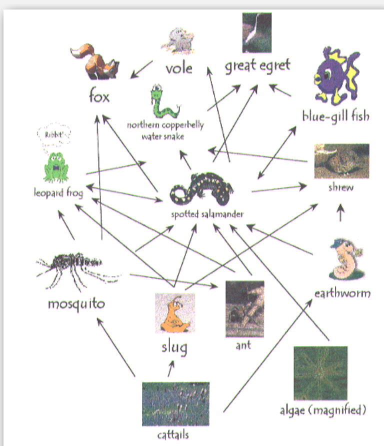
	0	1	2	3	4	5	6	7	8	9	10	11	12
scc	1	0	1	1	1	1	3	4	4	2	2	2	2

```
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.



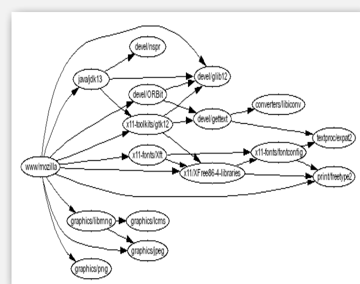
http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif

Strong component. Subset of species with common energy flow.

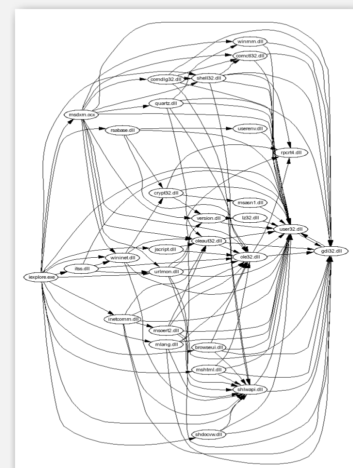
Strong component application: software modules

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

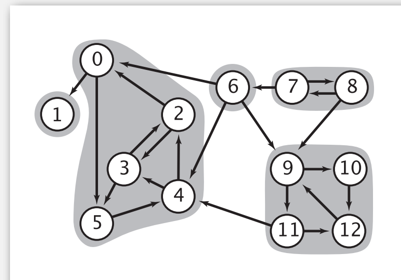
1990s: more easy linear-time algorithms (Gabow, Mehlhorn).
- Gabow: fixed old OR algorithm.
- Cheriyan-Mehlhorn: needed one-pass algorithm for LEDA.

Kosaraju's algorithm: intuition

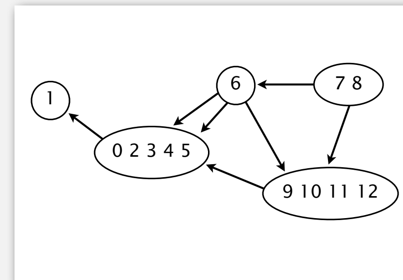
Reverse graph. Strong components in G are same as in G^R.

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

Idea. how to compute?
- Compute topological order in kernel DAG.
- Run DFS, considering vertices in reverse topological.



digraph G and its strong components



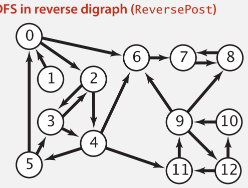
kernel DAG of G

Kosaraju's algorithm

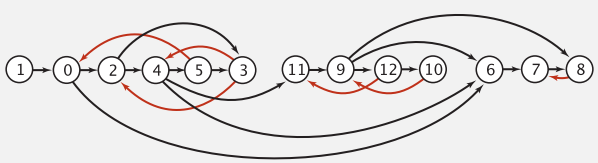
Simple (but mysterious) algorithm for computing strong components.
- Run DFS on G^R to compute reverse postorder.
- Run DFS on G, considering vertices in order given by first DFS.

DFS in reverse digraph (ReversePost)

G^R



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



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

reverse postorder

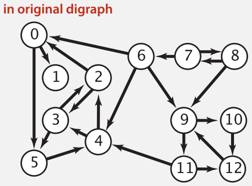
```
dfs(0)
  dfs(6)
    dfs(7)
      dfs(8)
        check 7
      8 done
    7 done
  6 done
  dfs(2)
    dfs(4)
      dfs(11)
        dfs(9)
          dfs(12)
            check 11
...
```


Kosaraju's algorithm

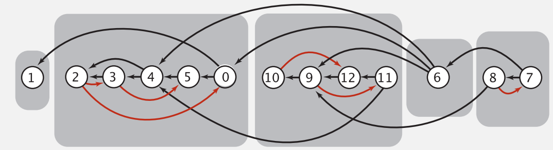
Simple (but mysterious) algorithm for computing strong components.
- Run DFS on G^R to compute reverse postorder.
- Run DFS on G, considering vertices in order given by first DFS.

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
  check 0
6 done
```

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

Proposition. Second DFS gives strong components. (!!)

Kosaraju proof of correctness

Proposition. Kosaraju's algorithm computes strong components.

Pf. We show that the vertices marked during the constructor call `dfs(G, s)` are the vertices strongly connected to s.

$\Leftarrow$ [If t is strongly connected to s, then t is marked during the call `dfs(G, s)`.]

- There is a path from s to t, so t will be marked during `dfs(G, s)` unless t was previously marked.
- There is a path from t to s, so if t were previously marked, then s would be marked before t finishes (so `dfs(G, s)` would not have been called in constructor).

```
dfs(G, s)
...
   dfs(G, t)
      ...
      check s
      ...
   t done
...
s done
```


Kosaraju proof of correctness (continued)

Proposition. Kosaraju's algorithm computes strong components.

$\Rightarrow$ [If t is marked during the call `dfs(G, s)`, then t is strongly connected to s.]

- Since t is marked during the call `dfs(G, s)`, there is a path from s to t in G (or equivalently, a path from t to s in G^R).
- Reverse postorder construction implies that t is done before s in dfs of G^R.
- The only possibility for dfs in G^R implies there is a path from s to t in G^R. (or equivalently, from t to s in G).

```
dfs(GR, s)
...
   dfs(GR, t)
      ...
      check s
      ...
   t done
...
s done          ✓
```

```
dfs(GR, t)
...
t done
...
dfs(GR, s)
...
s done          ✗
```

no path from t to s in G^R

```
dfs(GR, t)
...
dfs(GR, s)
...
t done
...
s done          ✗
```

DFS must nest

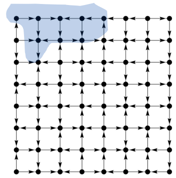
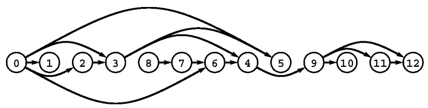
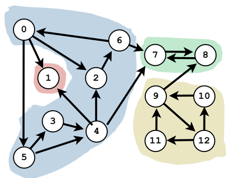
Connected components in an undirected graph (with DFS)

```
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 KosarajuSCC
{
   private boolean marked[];
   private int[] id;
   private int count;

   public KosarajuSCC(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

single-source reachability		DFS
topological sort (DAG)		DFS
strong components		Kosaraju DFS (twice)