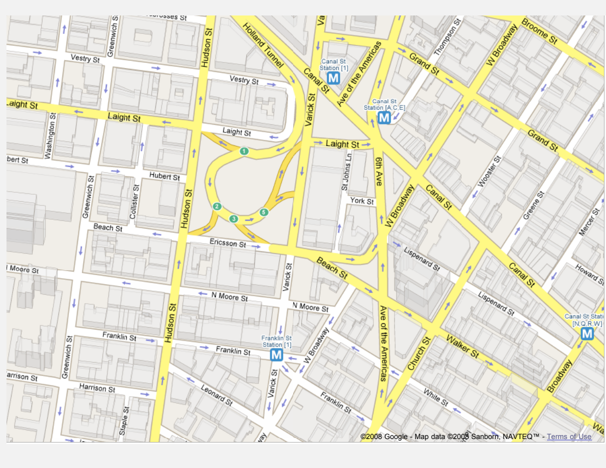
4.2 Directed Graphs

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

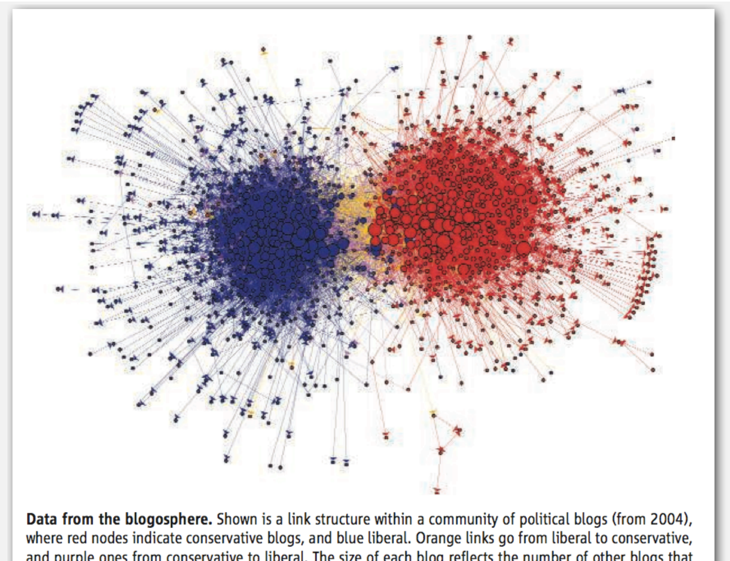
References: Algorithms in Java, 3rd edition, Chapter 19

Directed graphs

Digraph. Set of vertices connected pairwise by oriented edges.



Link structure of political blogs

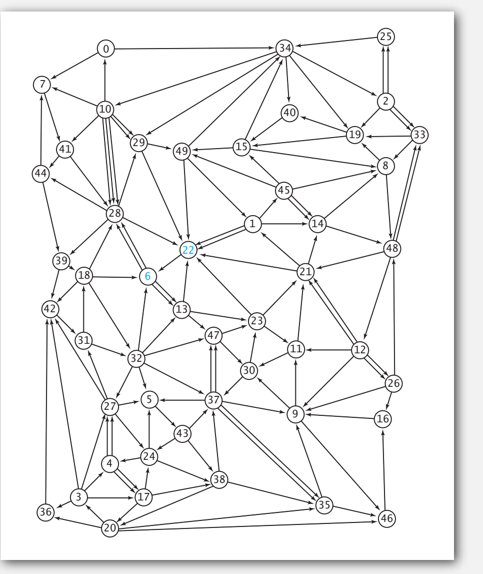


Data from the blogosphere. Shown is a link structure within a community of political blogs (from 2004), where red nodes indicate conservative blogs, and blue liberal. Orange links go from liberal to conservative, and purple ones from conservative to liberal. The size of each blog reflects the number of other blogs that link to it. [Reproduced from (8) with permission from the Association for Computing Machinery]

Web graph

Vertex = web page.

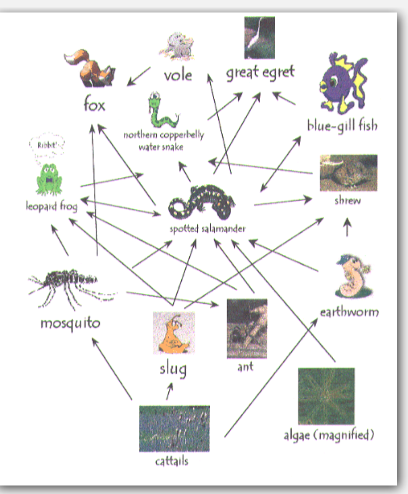
Edge = hyperlink.



Ecological food web graph

Vertex = species.

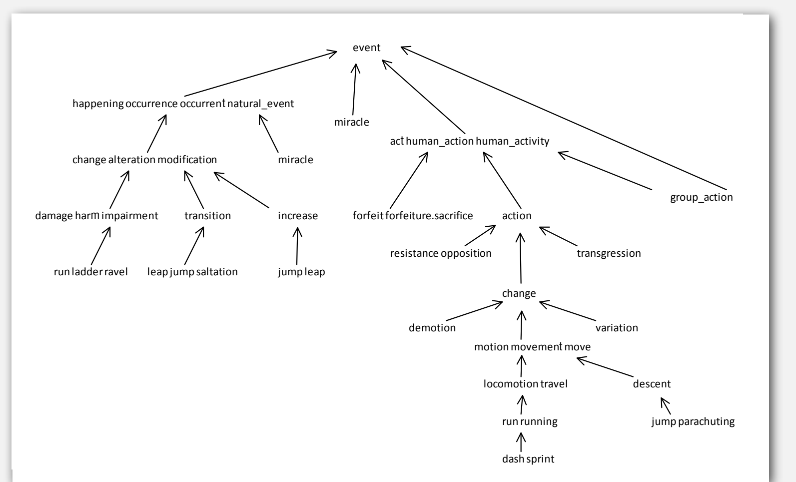
Edge: from producer to consumer.



WordNet graph

Vertex = synset.

Edge = hypernym relationship.



Digraph applications

graph	vertex	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	stock, currency	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 and t?

Strong connectivity. Are all vertices mutually reachable?

Transitive closure. For which vertices v and w is there a path from v to w?

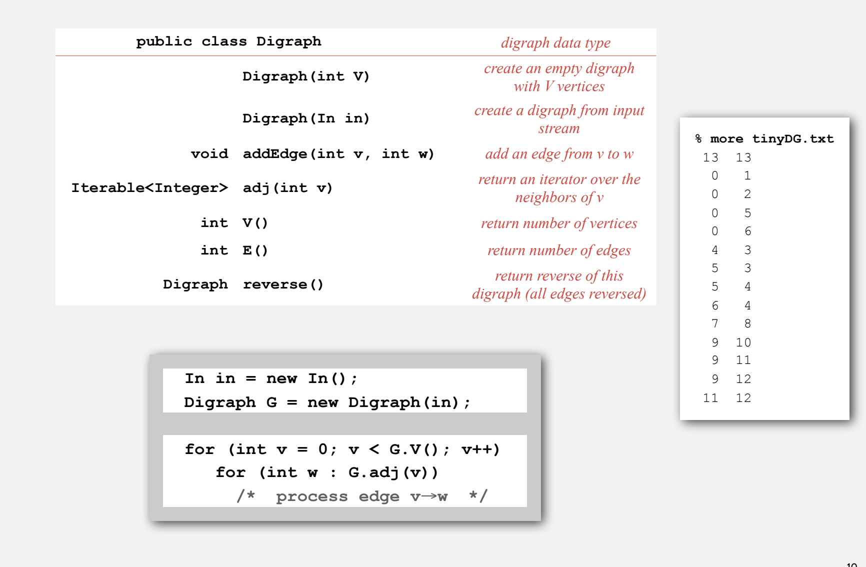
Topological sort. Can you draw the digraph so that all edges point from left to right?

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

PageRank. What is the importance of a web page?

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

Digraph API



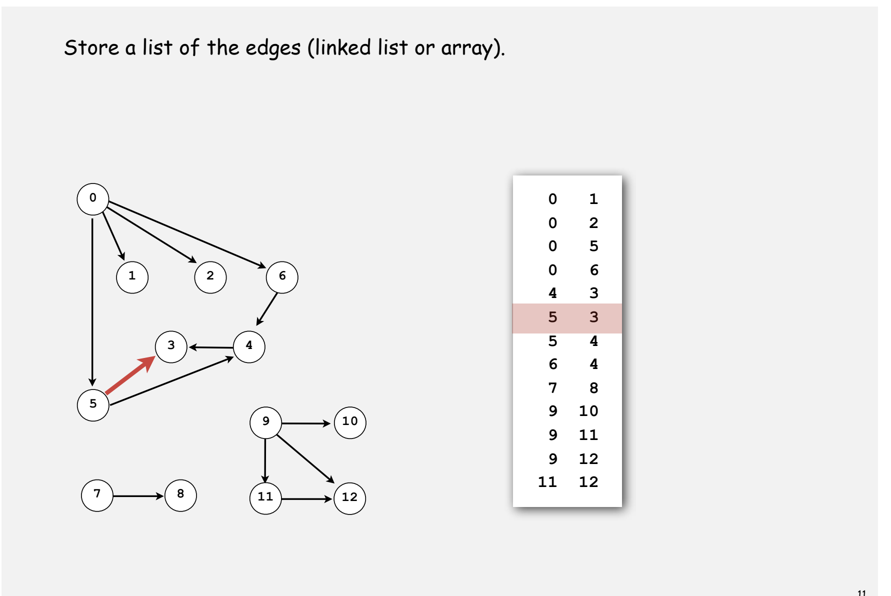
public class Digraph		*digraph data type*
	`Digraph(int V)`	*create an empty digraph with V vertices*
	`Digraph(In in)`	*create a digraph from input stream*
`void`	`addEdge(int v, int w)`	*add an edge from v to w*
`Iterable<Integer>`	`adj(int v)`	*return an iterator over the neighbors of v*
`int`	`V()`	*return number of vertices*
`int`	`E()`	*return number of edges*
`Digraph`	`reverse()`	*return reverse of this digraph (all edges reversed)*

```
In in = new In();
Digraph G = new Digraph(in);
```

```
for (int v = 0; v < G.V(); v++)
   for (int w : G.adj(v))
      /* process edge v→w */
```

```
% more tinyDG.txt
13  13
 0   1
 0   2
 0   5
 0   6
 4   3
 5   3
 5   4
 6   4
 7   8
 9  10
 9  11
 9  12
11  12
```

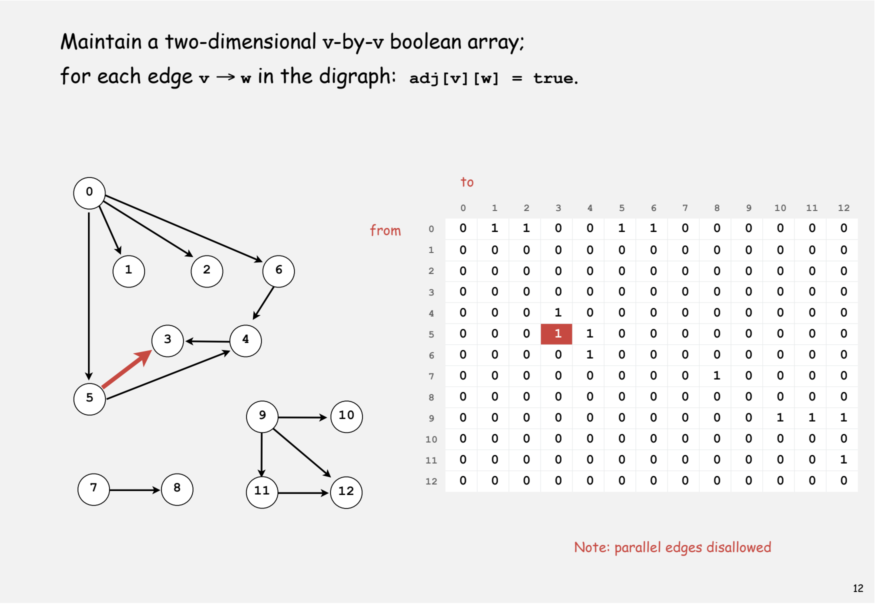

Set of edges representation



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

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

Adjacency-matrix representation



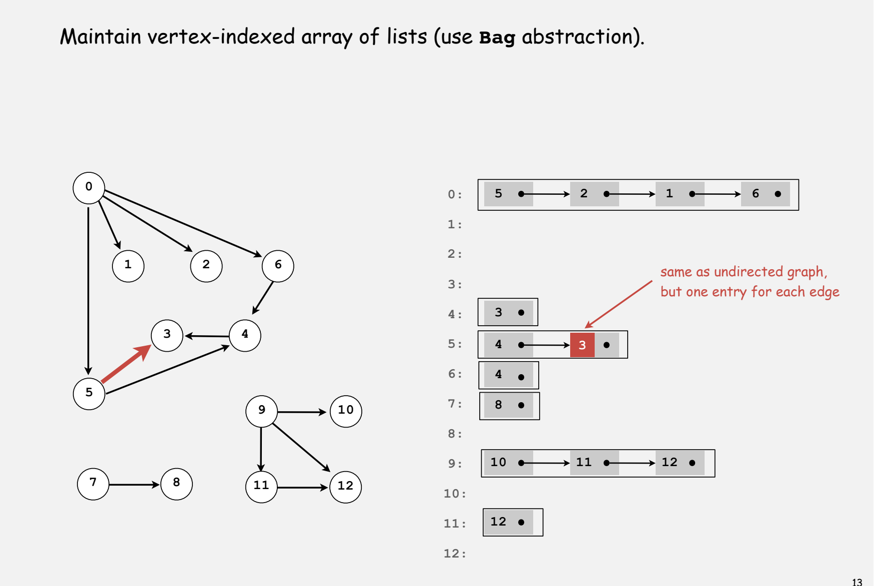
Maintain a two-dimensional v-by-v boolean array;
for each edge v → w in the digraph: `adj[v][w] = true`.

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

Note: parallel edges disallowed

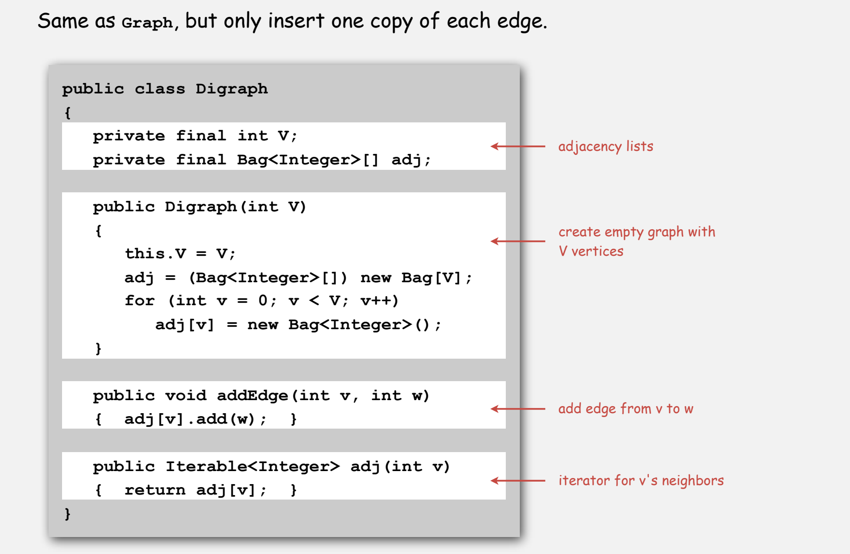
Adjacency-list representation

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



Adjacency-lists 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];  }
}
```


Digraph representations

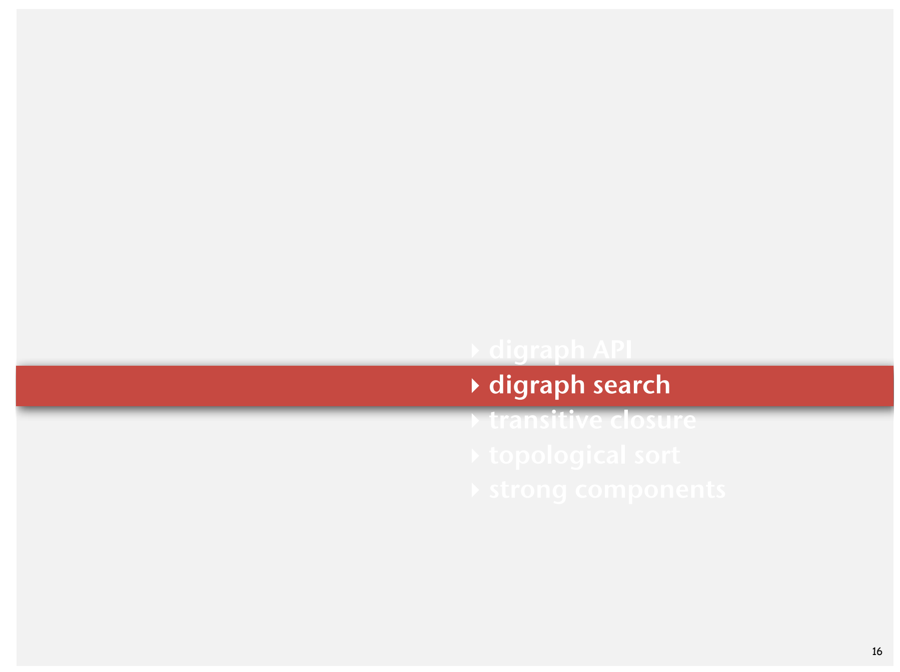
In practice. Use adjacency-list representation.

- Algorithms all based on iterating over edges incident to v.
- Real-world digraphs tend to be sparse.

huge number of vertices, small average vertex degree

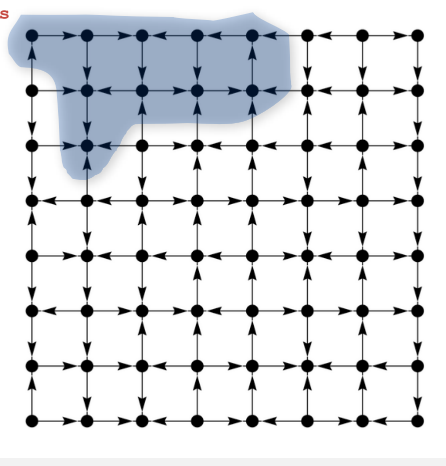
representation	space	insert edge from v to w	edge from v to w?	iterate over edges leaving v?
list of edges	E	1 *	E	E
adjacency matrix	V^2	1	1	V
adjacency list	E + V	1 *	outdegree(v)	outdegree(v)
adjacency set	E + V	log (outdegree(v))	log (outdegree(v))	outdegree(v)

* only if parallel edges allowed



Reachability

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



Depth-first search in digraphs

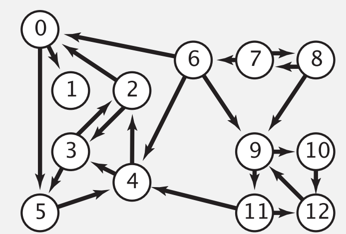
Same method as for undirected graphs.

Every undirected graph is a digraph.

- Happens to have edges in both directions.
- DFS is a digraph algorithm.

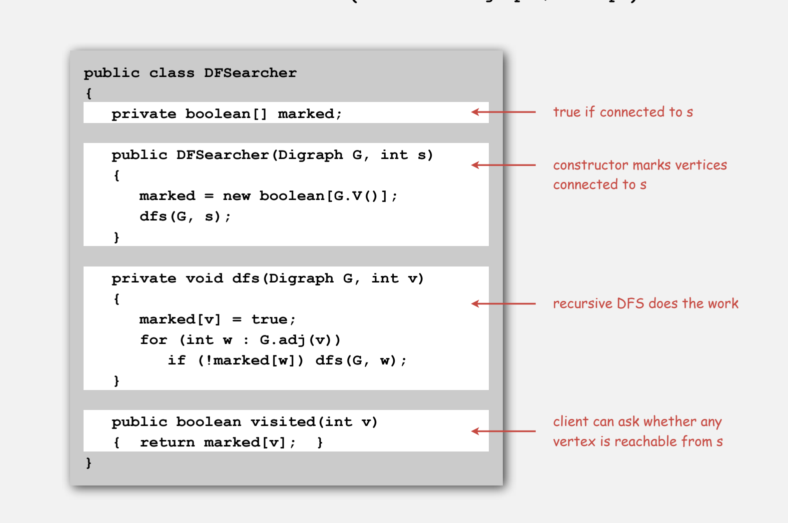
DFS (to visit a vertex s)

Mark s as visited.
Recursively visit all unmarked vertices w adjacent to s.



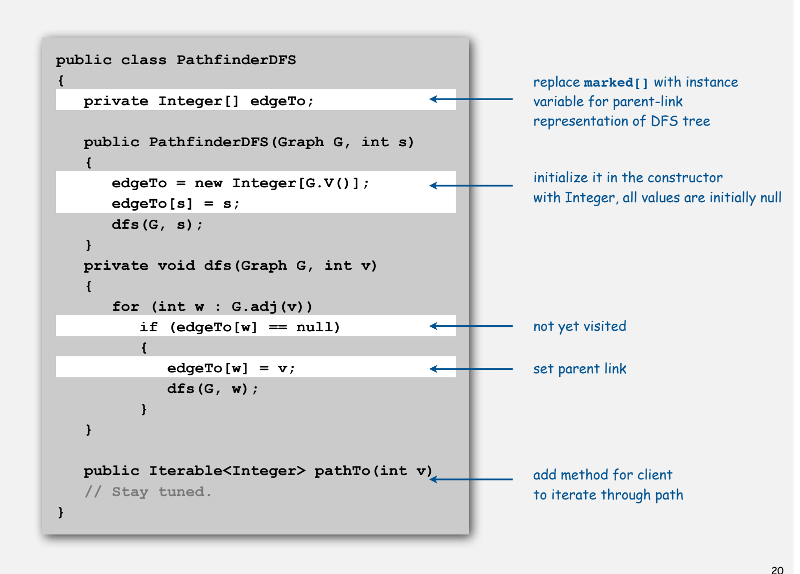
Depth-first search (single-source reachability)

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



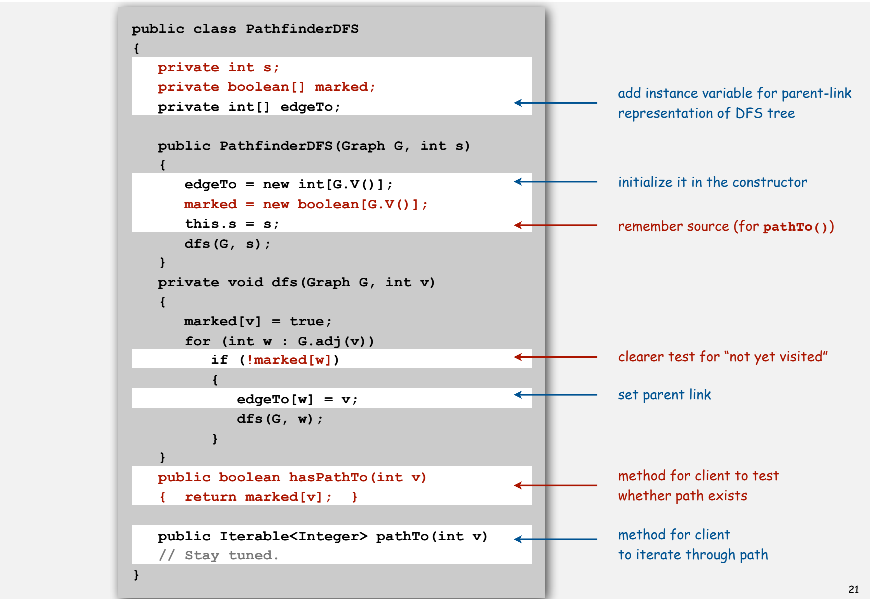
```
public class DFSearcher
{
   private boolean[] marked;

   public DFSearcher(Digraph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

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

   public boolean visited(int v)
   {  return marked[v];  }
}
```


Depth-first-search (pathfinding) for undirected graphs [from Lecture 12]

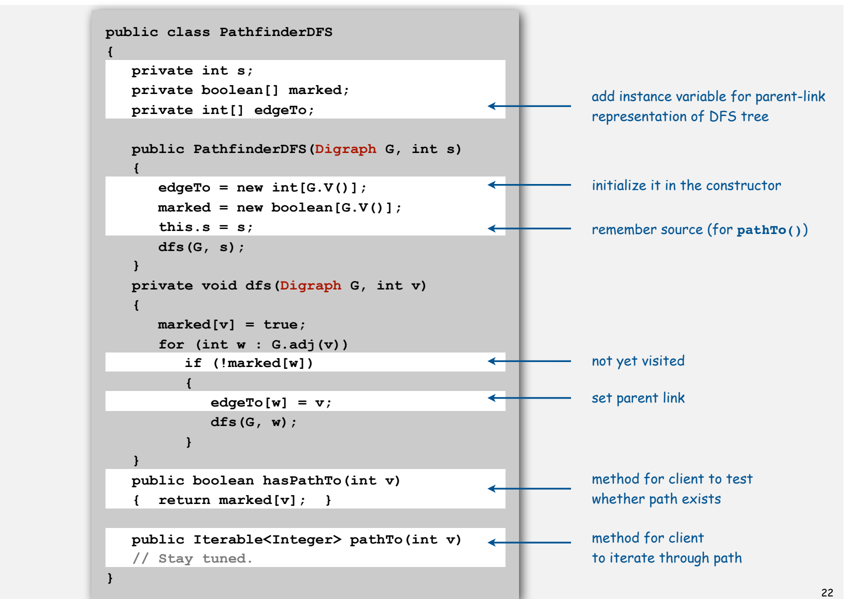


```
public class PathfinderDFS
{
   private Integer[] edgeTo;

   public PathfinderDFS(Graph G, int s)
   {
      edgeTo = new Integer[G.V()];
      edgeTo[s] = s;
      dfs(G, s);
   }
   private void dfs(Graph G, int v)
   {
      for (int w : G.adj(v))
         if (edgeTo[w] == null)
         {
            edgeTo[w] = v;
            dfs(G, w);
         }
   }

   public Iterable<Integer> pathTo(int v)
   // Stay tuned.
}
```

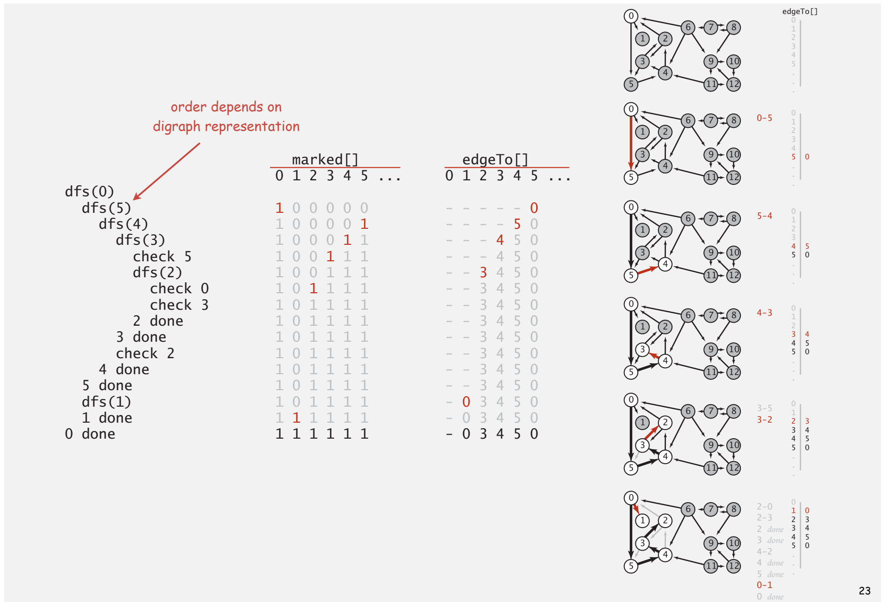
Depth-first-search (pathfinding) for undirected graphs [slightly different version]



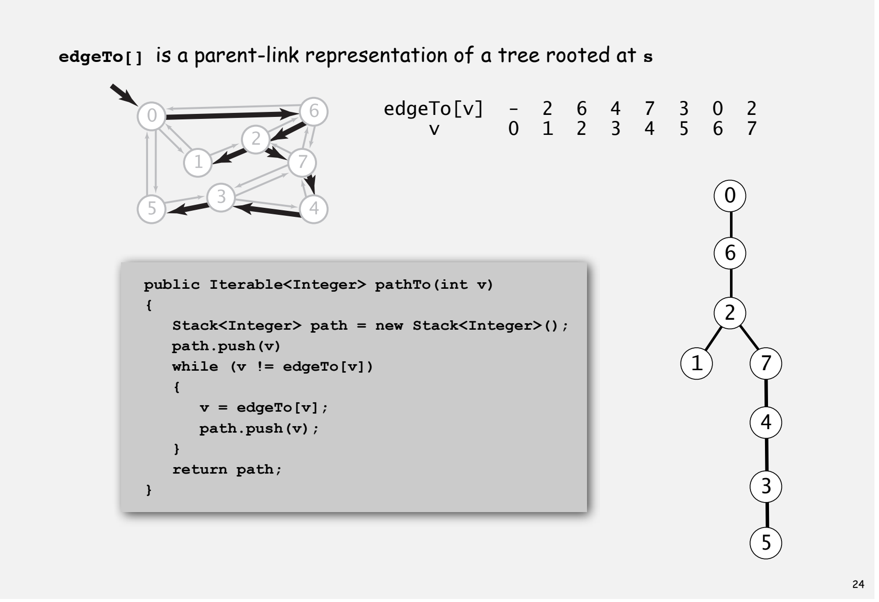
Depth-first-search (pathfinding) for digraphs



DFS pathfinding trace in a digraph

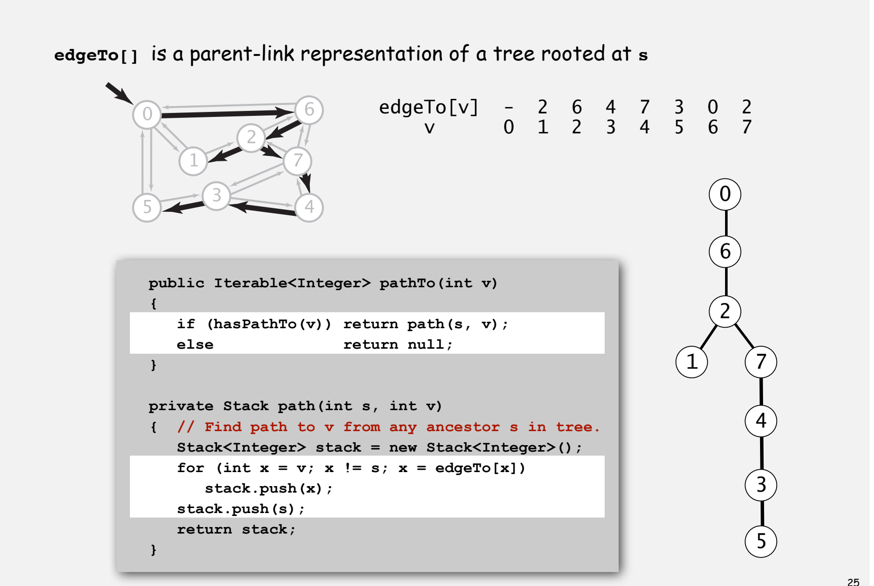


Depth-first-search (pathfinding iterator) [from lecture 12]



Depth-first-search (pathfinding iterator) [slightly different version]

`edgeTo[]` is a parent-link representation of a tree rooted at `s`

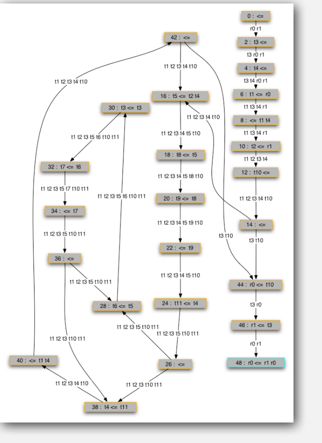



```
public Iterable<Integer> pathTo(int v)
{
   if (hasPathTo(v)) return path(s, v);
   else              return null;
}

private Stack path(int s, int v)
{  // Find path to v from any ancestor s in tree.
   Stack<Integer> stack = new Stack<Integer>();
   for (int x = v; x != s; x = edgeTo[x])
      stack.push(x);
   stack.push(s);
   return stack;
}
```


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).

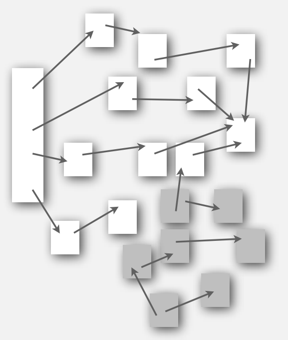
Reachable objects. Objects indirectly accessible by program (starting at a root and following a chain of pointers).

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 (DFS)

DFS enables direct solution of simple digraph problems.

- ✓ Reachability.
- Cycle detection.
- Topological sort.
- Transitive closure.

Basis for solving difficult digraph problems.

- Directed Euler path.
- Strong connected components.

Breadth-first search in digraphs

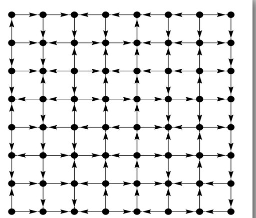
Every undirected graph is a digraph.

- Happens to have edges in both directions.
- BFS is a digraph algorithm.

BFS (from source vertex s)

Put s onto a FIFO queue.
Repeat until the queue is empty:

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



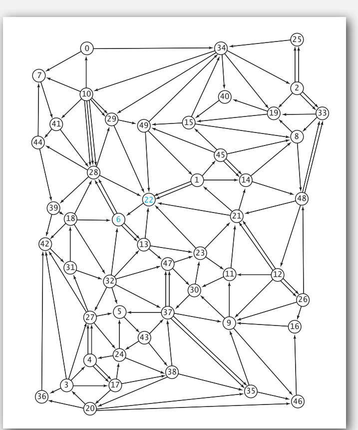
Property. Visits vertices in increasing distance from s.

Digraph BFS application: web crawler

Goal. Crawl web, starting from some root web page, say `www.princeton.edu`.
Solution. BFS with implicit graph.

BFS.

- Start at some root web page.
- 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?

Web crawler: BFS-based 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 for 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

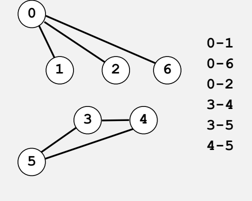
‣ digraph API
‣ digraph search
‣ transitive closure
‣ topological sort
‣ strong components

Graph-processing challenge (revisited)

Problem. Is there an undirected path between v and w ?
Goals. Linear preprocessing time, constant query time.

How difficult?

- Any COS 126 student could do it.
- ✓ Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



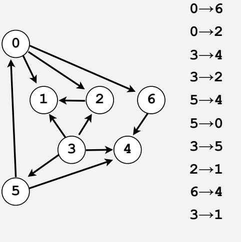
Digraph-processing challenge 1

Problem. Is there a directed path from v to w ?
Goals. Linear preprocessing time, constant query time.

How difficult?

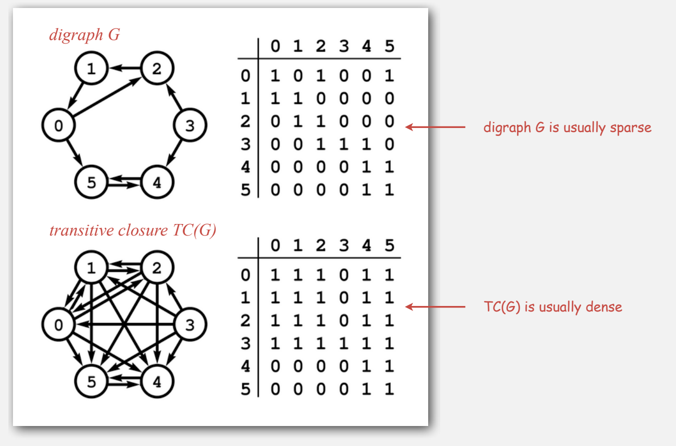
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- ✓ Impossible.

can't do better than V^2
(reduction from boolean matrix multiplication)



Transitive closure

Def. The transitive closure of a digraph G is another digraph with a directed edge from v to w if there is a directed path from v to w in G.



digraph G

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

transitive closure TC(G)

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

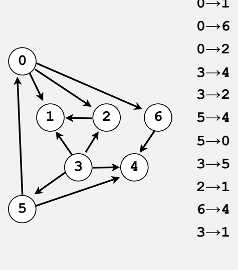
Digraph-processing challenge 1 (revised)

Problem. Is there a directed path from `v` to `w` ?

Goals. ~ V^2 preprocessing time, constant query time.

How difficult?

- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- ✓ No one knows. ← open research problem
- Impossible.



Digraph-processing challenge 1 (revised again)

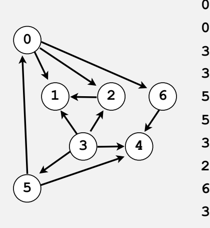
Problem. Is there a directed path from `v` to `w` ?

Goals. ~ V E preprocessing time, ~ V^2 space, constant query time.

How difficult?

- Any COS 126 student could do it.
- ✓ Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Use DFS once for each vertex to compute rows of transitive closure



Transitive closure: Java implementation

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

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

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

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

- array of `DFSearcher` objects
- initialize array
- is there a directed path from v to w ?

Similar approach (array of `PathFinderDFS` objects) can provide paths.

Warning: Not for use with huge graphs (~V^2 space)

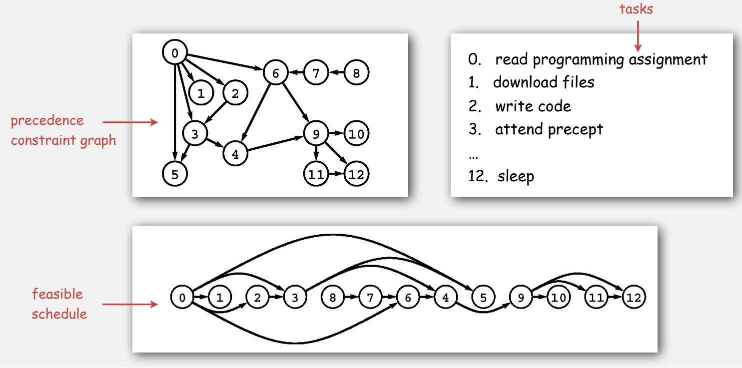
- digraph API
- digraph search
- transitive closure
- **DAGs**
- strong components

Digraph application: scheduling

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

Graph model.

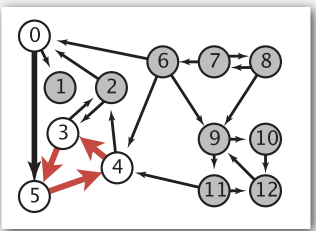
- Create a vertex v for each task.
- Create an edge v→w if task v must precede task w.



Digraph application: scheduling

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

No solution iff digraph has a directed cycle



First problem. Make sure digraph has no cycles.

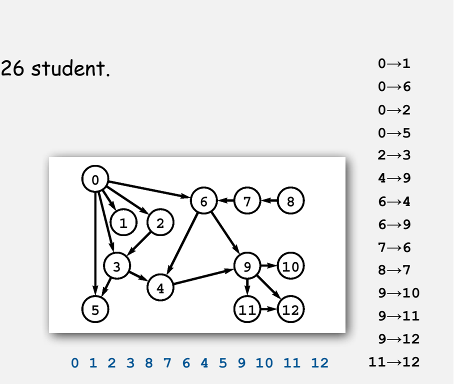
Digraph-processing challenge 2a

Problem. Check that a digraph is a DAG.
Goal. Linear time.

How difficult?

- Any COS 126 student could do it.
- ✓ Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

use DFS

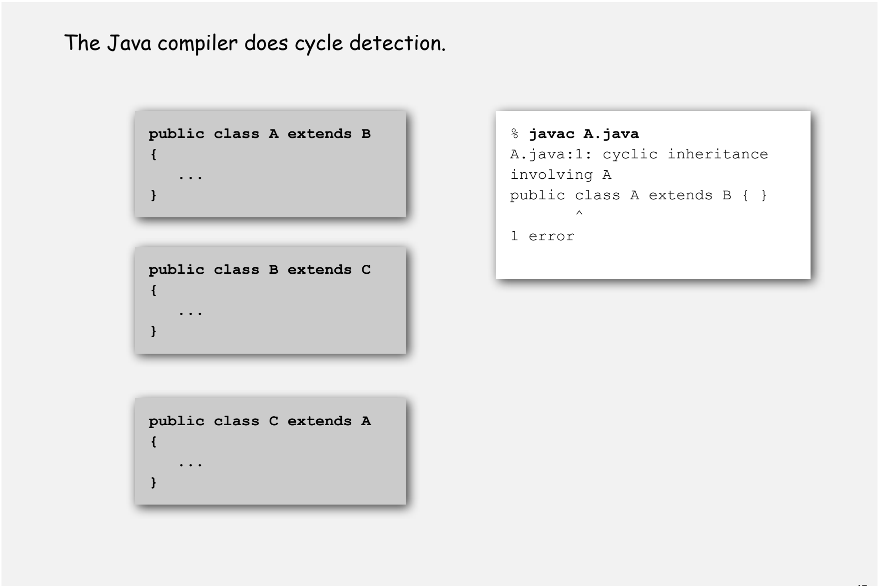


Cycle detection applications

- Causalities.
- Email loops.
- Compilation units.
- Class inheritance.
- Course prerequisites.
- Deadlocking detection.
- Precedence scheduling.
- Temporal dependencies.
- Pipeline of computing jobs.
- Check for symbolic link loop.
- Evaluate formula in spreadsheet.

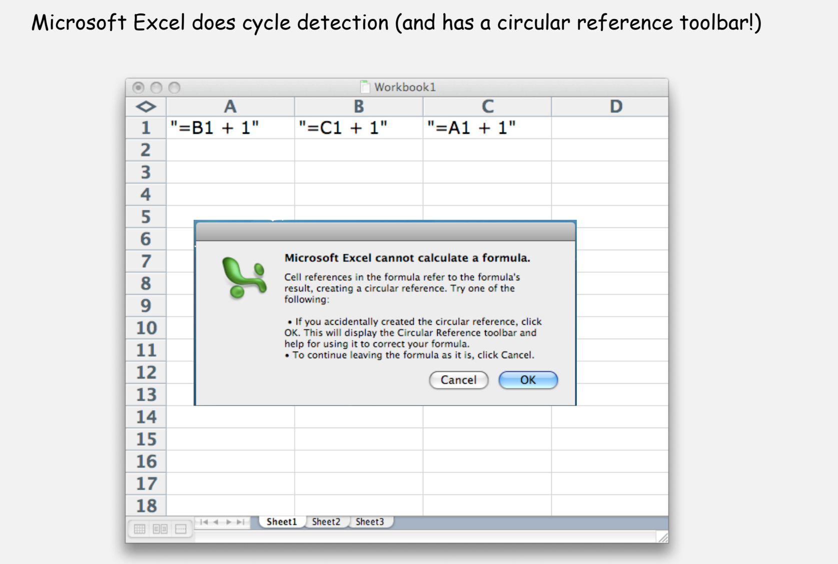
Cycle detection application: cyclic inheritance

The Java compiler does cycle detection.



Cycle detection application: spreadsheet recalculation

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

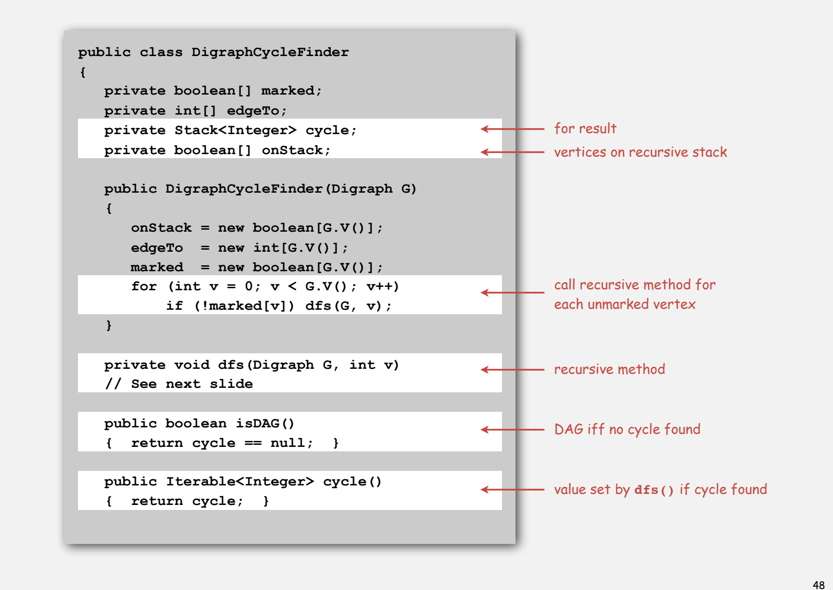


Cycle detection application: symbolic links

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
```


Finding a cycle in a digraph: Java implementation



for result

vertices on recursive stack

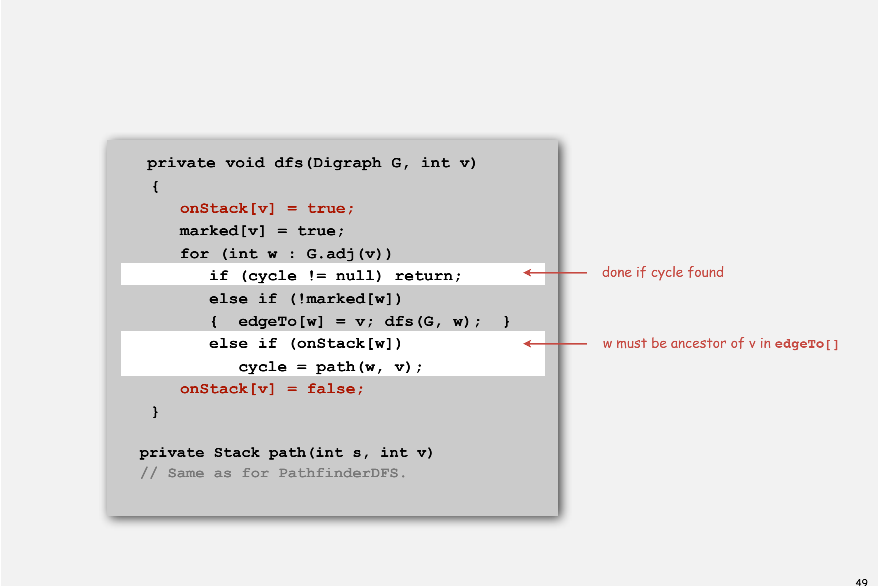
call recursive method for each unmarked vertex

recursive method

DAG iff no cycle found

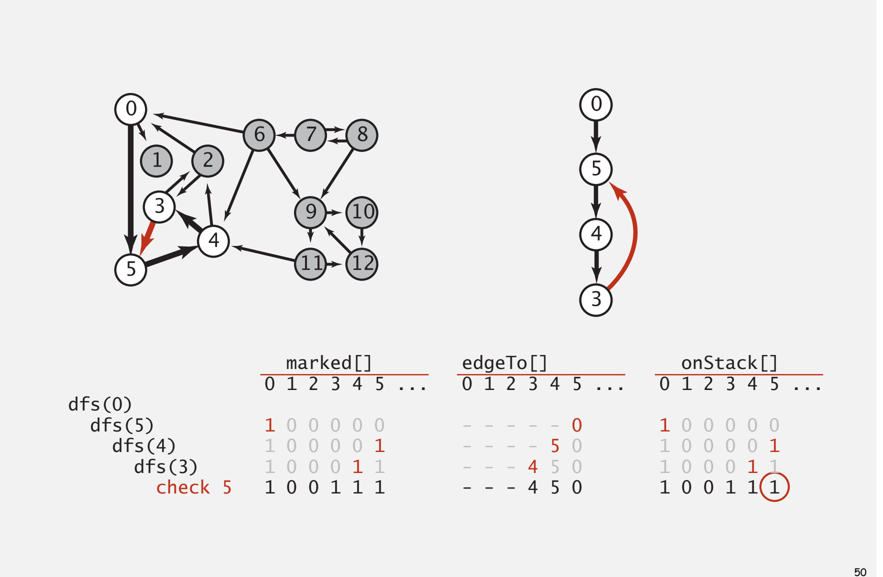
value set by `dfs()` if cycle found

Finding a cycle in a digraph: Java implementation (continued)



```
private void dfs(Digraph G, int v)
{
   onStack[v] = true;
   marked[v] = true;
   for (int w : G.adj(v))
      if (cycle != null) return;
      else if (!marked[w])
      {  edgeTo[w] = v; dfs(G, w);  }
      else if (onStack[w])
         cycle = path(w, v);
   onStack[v] = false;
}

private Stack path(int s, int v)
// Same as for PathfinderDFS.
```

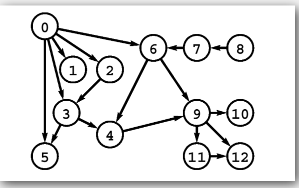

Finding a cycle in a digraph



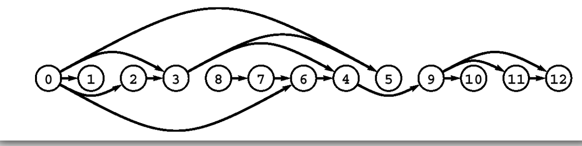
	marked[] 0 1 2 3 4 5 ...	edgeTo[] 0 1 2 3 4 5 ...	onStack[] 0 1 2 3 4 5 ...
dfs(0)			
dfs(5)	1 0 0 0 0 0	- - - - - 0	1 0 0 0 0 0
dfs(4)	1 0 0 0 0 1	- - - - 5 0	1 0 0 0 0 1
dfs(3)	1 0 0 0 1 1	- - - 4 5 0	1 0 0 0 1 1
check 5	1 0 0 1 1 1	- - - 4 5 0	1 0 0 1 1 1

Topological sort

DAG. Directed acyclic graph.



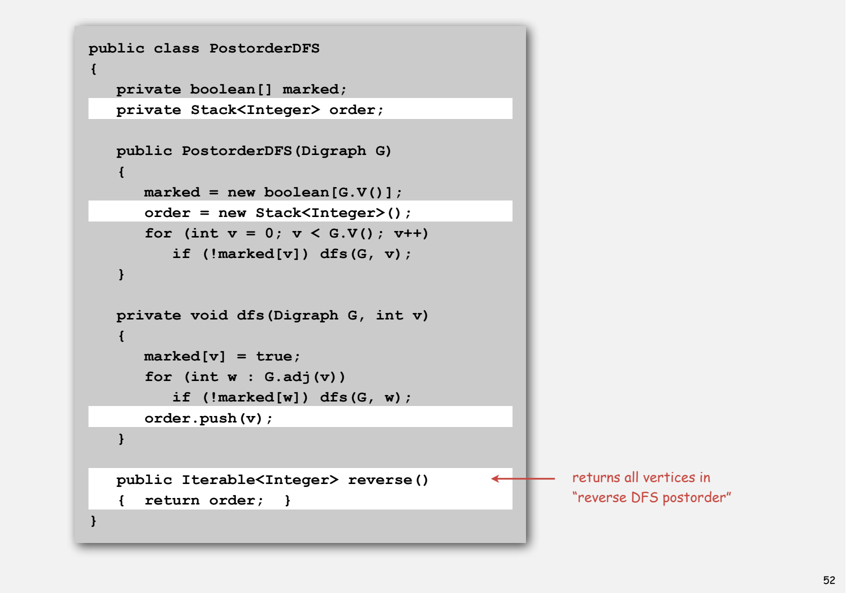
Topological sort. Redraw DAG so all edges point left to right.



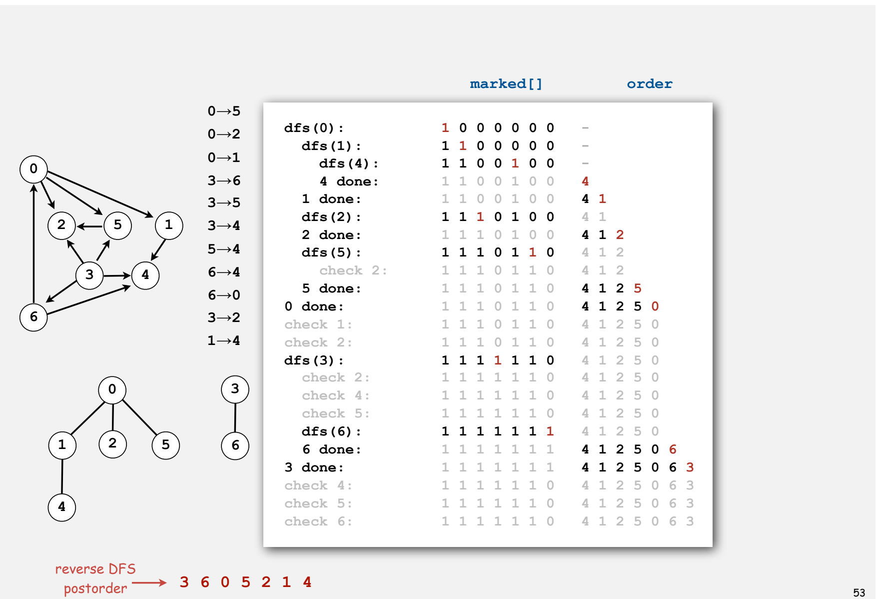
Application. Scheduling.

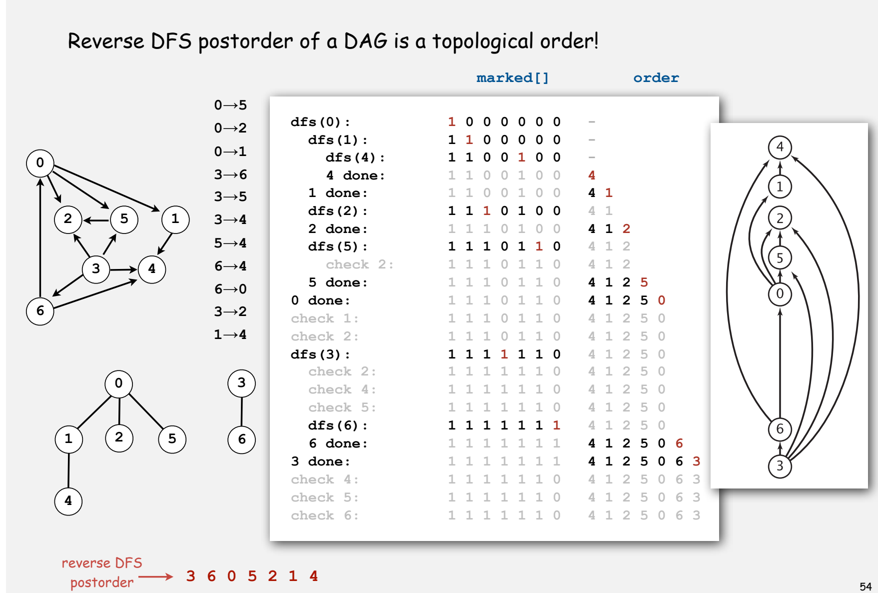
Solution. DFS (what else!).

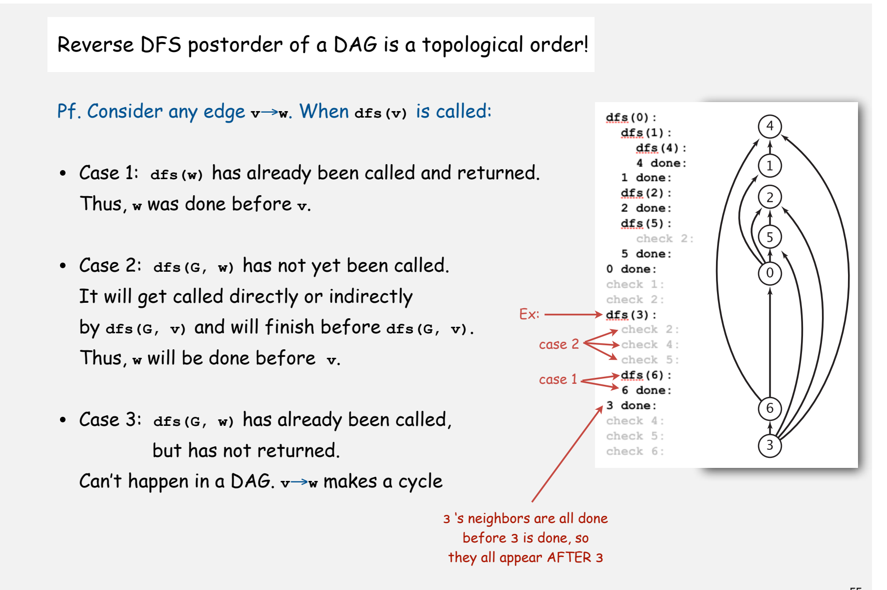
Reverse DFS postorder in a digraph: Java implementation

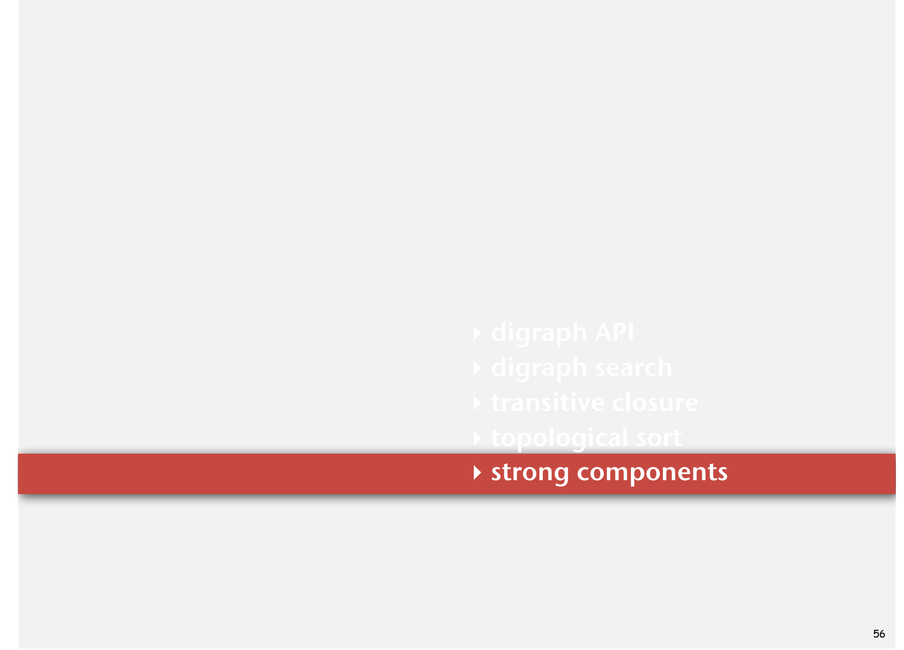


```
public class PostorderDFS
{
   private boolean[] marked;
   private Stack<Integer> order;

   public PostorderDFS(Digraph G)
   {
      marked = new boolean[G.V()];
      order = new Stack<Integer>();
      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);
      order.push(v);
   }

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


Reverse DFS postorder in a DAG
marked[]
order
0→5
0→2
0→1
3→6
3→5
3→4
5→4
6→4
6→0
3→2
1→4
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
reverse DFS postorder → 3 6 0 5 2 1 4
53


Reverse DFS postorder in a DAG: an amazing fact
Reverse DFS postorder of a DAG is a topological order!
marked[]
order
0→5
0→2
0→1
3→6
3→5
3→4
5→4
6→4
6→0
3→2
1→4
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
reverse DFS postorder → 3 6 0 5 2 1 4
54


Topological sort in a DAG: correctness proof
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(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. v→w makes 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:
Ex:
case 2
case 1
3 's neighbors are all done before 3 is done, so they all appear AFTER 3
55

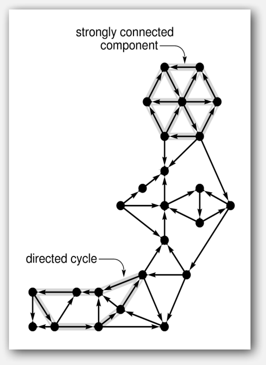

digraph API
digraph search
transitive closure
topological sort
strong components
56

Strongly connected components

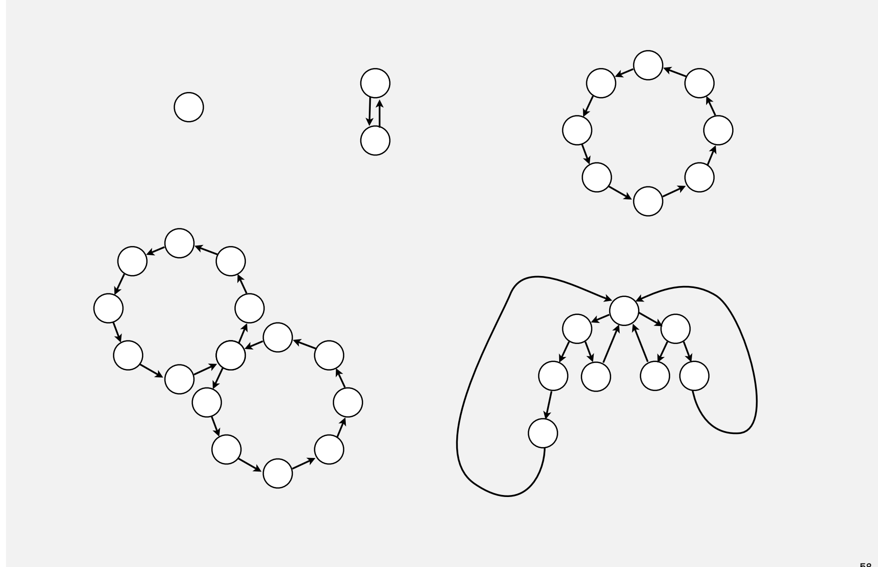
Def. Vertices v and w are strongly connected if
there is a directed path from v to w
and a directed path from w to v.

(Equivalent) Vertices v and w are strongly connected if
there is a directed cycle containing v and w

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



Examples of strongly connected digraphs



Digraph-processing challenge 3

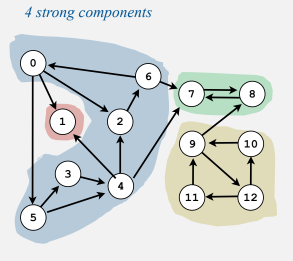
Problem. Are v and w strongly connected?
Goal. Linear preprocessing time, constant query time.

How difficult?

- Any COS 126 student could do it.
- ✓ Need to be a typical diligent COS 226 student.
- Hire an expert (or a COS 423 student).
- Intractable.
- No one knows.
- Impossible.

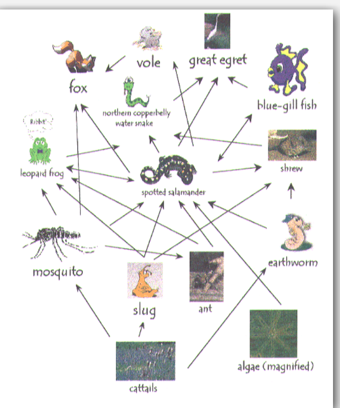
use DFS twice to find strong components (stay tuned)

4 strong components



Ecological food web graph

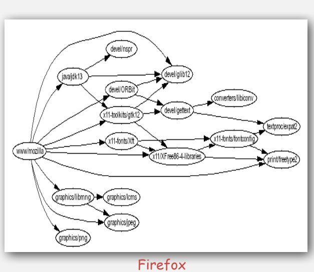
Vertex = species.
Edge: from producer to consumer.



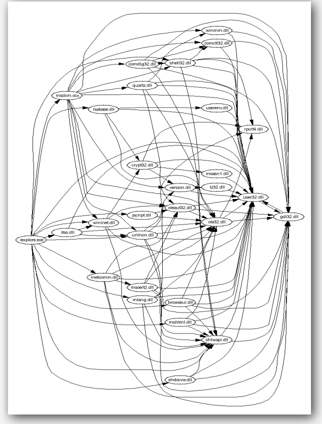
Strong component. Subset of species with common energy flow.

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.

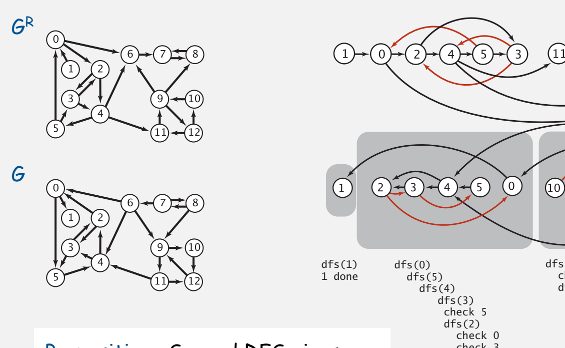
1972: linear-time DFS algorithm (Tarjan).
- Classic algorithm.
- Level of difficulty: CS226++.
- Demonstrated broad applicability and importance of DFS.

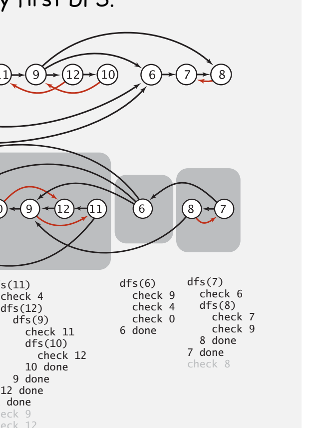
1980s: easy two-pass linear-time algorithm (Kosaraju).
- Forgot notes for teaching algorithms class; developed alg in order to teach it!
- Later found in Russian scientific literature (1972).

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

Kosaraju's algorithm

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(1)      dfs(0)                dfs(11)              dfs(6)       dfs(7)
1 done        dfs(5)                check 4              check 9      check 6
                dfs(4)              dfs(12)              check 4      dfs(8)
                   dfs(3)             dfs(9)             check 0        check 7
                    check 5             check 11         6 done         check 9
                    dfs(2)              dfs(10)                       8 done
                      check 0             check 12                  7 done
                      check 3           10 done                     check 8
                    2 done            9 done
                  3 done            12 done
                  check 2         11 done
                4 done            check 9
              5 done              check 12
              check 1             check 10
            0 done
            check 2
            check 4
            check 5
```

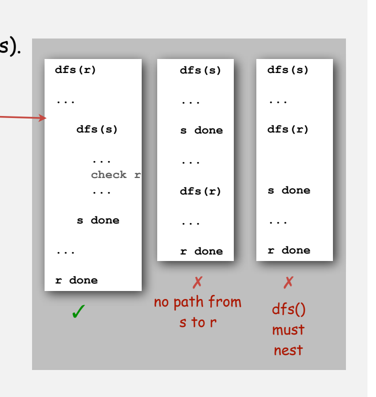
Proposition. Second DFS gives strong components. (!!)

Kosaraju proof sketch

Proposition: Kosaraju's algorithm computes strong components

Lemma. Every vertex **s** visited by **dfs(r)** is strongly connected to **r**.

Proof of Lemma:
- Path from **r** to **s** exists (**dfs()** follows edges). Thus, path from **s** to **r** exists in G^R
- The only possibility for the **dfs()** in G^R implies path from **r** to **s** exists in G^R.
- Thus, **s** and **r** are strongly connected in G^R.
- Thus, **s** and **r** are strongly connected in G.



Proof of Proposition:
All vertices visited by **dfs(r)** are strongly connected because they are all strongly connected to **r**.

Finding connected components in an undirected graph with DFS (from Section 4.1)

```
public class CCfinder
{
   private boolean[] marked;
   private Bag<Integer>[] connectedTo;
   private Bag<Integer> representatives;

   public CCfinder(Graph G)
   {
      marked = new boolean[G.V()];
      representatives = new Bag<Integer>();
      connectedTo = (Bag<Integer>[]) new Bag[G.V()];

      for (int s = 0; s < G.V(); s++)
         if (!marked[s])
         {
            Bag<Integer> bag = new Bag<Integer>();
            representatives.add(s);
            dfs(G, s, bag);
         }
   }

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

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

   public Iterable<Integer> representatives()
   {  return representatives;  }

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


Finding strongly connected components in a digraph with DFS (Kosaraju)

```
public class KosarajuSCC
{
   private boolean[] marked;
   private Bag<Integer>[] connectedTo;
   private Bag<Integer> representatives;

   public KosarajuSCC(Digraph G)
   {
      marked = new boolean[G.V()];
      representatives = new Bag<Integer>();
      connectedTo = (Bag<Integer>[]) new Bag[G.V()];
      PostorderDFS postorder = new PostorderDFS(G.reverse());
      for (int s : postorder.reverse())
         if (!marked[s])
         {
            Bag<Integer> bag = new Bag<Integer>();
            representatives.add(s);
            dfs(G, s, bag);
         }
   }

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

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

   public Iterable<Integer> representatives()
   {  return representatives;  }

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


Digraph-processing summary: algorithms of the day

single-source reachability		DFS
transitive closure		DFS (from each vertex)
topological sort (DAG)		DFS
strong components		Kosaraju DFS (twice)