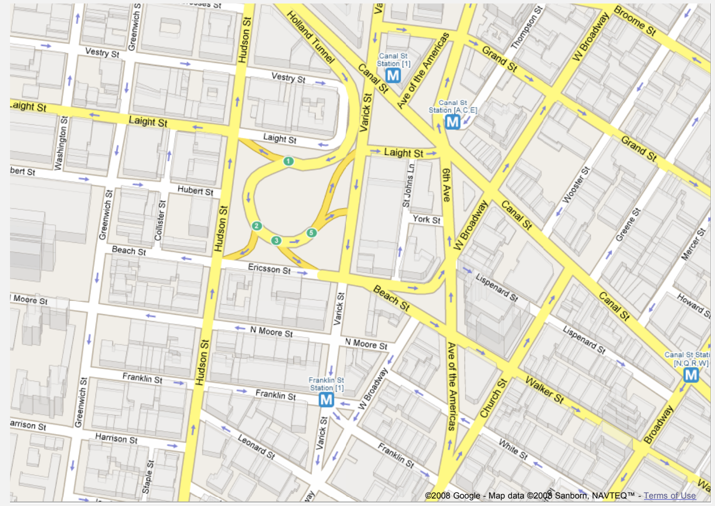
5.2 Directed Graphs

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

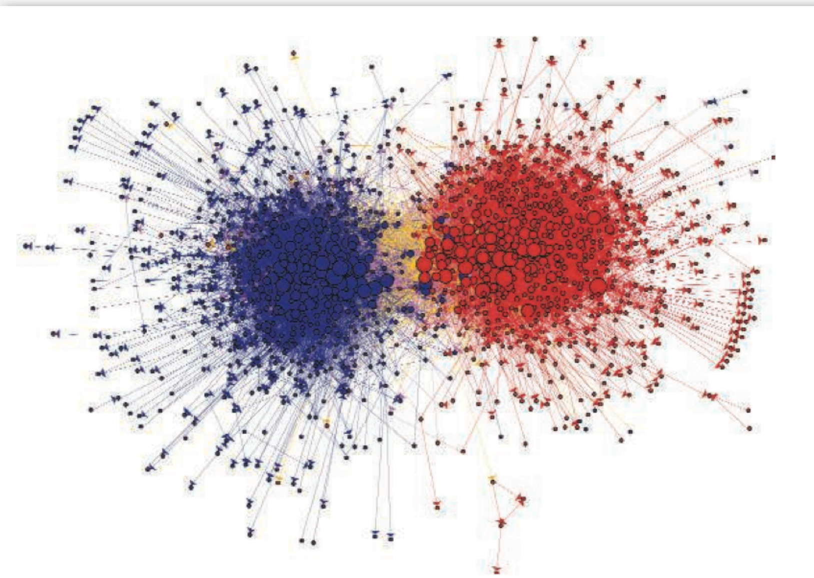
References: Algorithms in Java, 3rd edition, Chapter 19

Directed graphs

Digraph. Set of vertices connected pairwise by oriented edges.



Link structure of political blogs

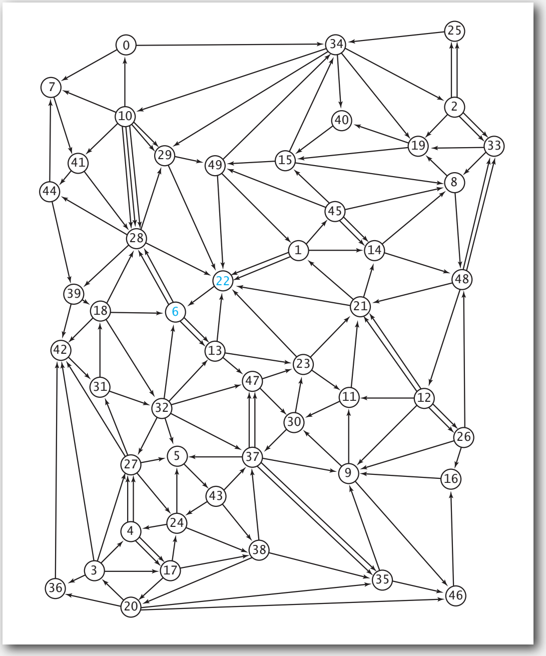


Data from the blogosphere. Shown is a link structure within a community of political blogs (from 2004), where red nodes indicate conservative blogs, and blue liberal. Orange links go from liberal to conservative, and purple ones from conservative to liberal. The size of each blog reflects the number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

Web graph

Vertex = web page.

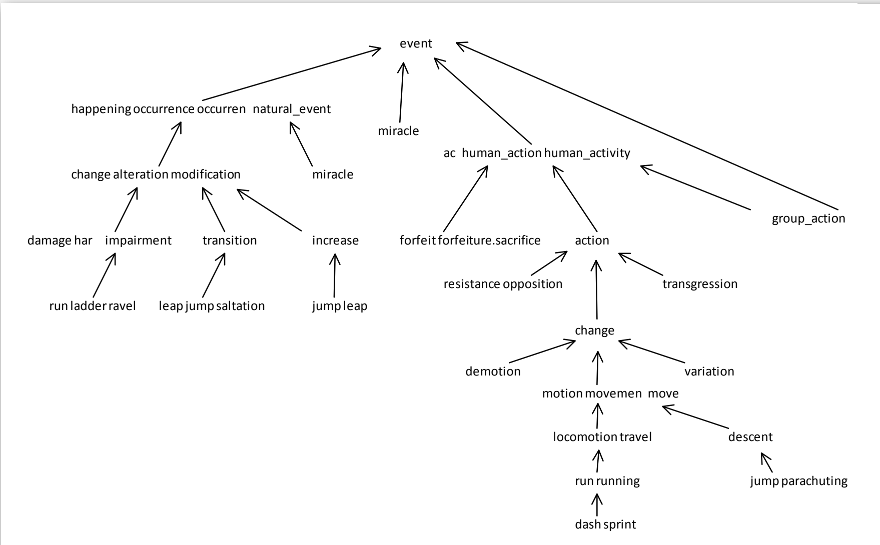
Edge = hyperlink.



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

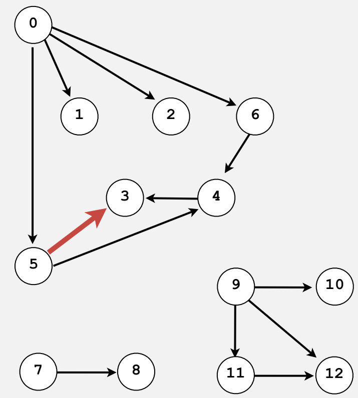
public class Digraph		*digraph data type*
	Digraph(int V)	*create an empty digraph with V vertices*
	Digraph(In in)	*create a digraph from input stream*
void	addEdge(int v, int w)	*add an edge from v to w*
Iterable<Integer>	adj(int v)	*return an iterator over the neighbors of v*
int	V()	*return number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

```
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 */
```


Set of edges representation

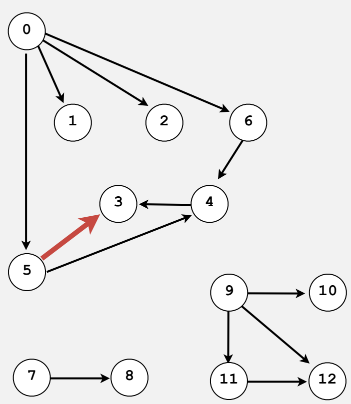
Store a list of the edges (linked list or array).



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`.

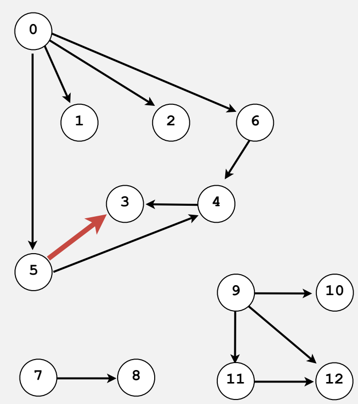


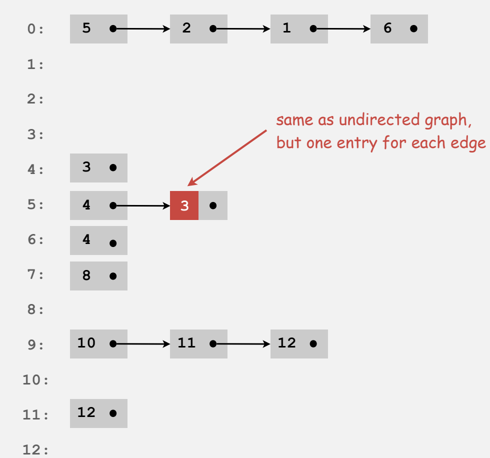
to

from	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

Adjacency-list representation

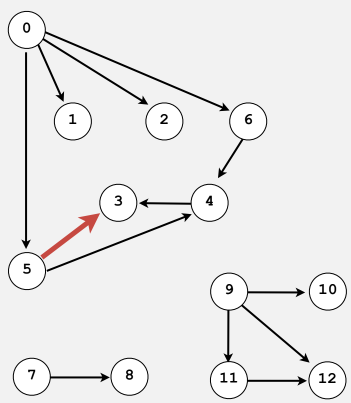
Maintain vertex-indexed array of lists.





Adjacency-set representation

Maintain vertex-indexed array of sets.



0:	{ 1 2 5 6 }
1:	{ }
2:	{ }
3:	{ }
4:	{ 3 }
5:	{ 3, 4 }
6:	{ 4 }
7:	{ 8 }
8:	{ }
9:	{ 10, 11, 12 }
10:	{ }
11:	{ 12 }
12:	{ }

same as undirected graph, but one entry for each edge

Adjacency-set representation: Java implementation

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

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

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

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

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

adjacency sets

create empty graph with V vertices

add edge from v to w (no parallel edges)

iterator for v's neighbors

Digraph representations

In practice. Use adjacency-set (or adjacency-list) representation.

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

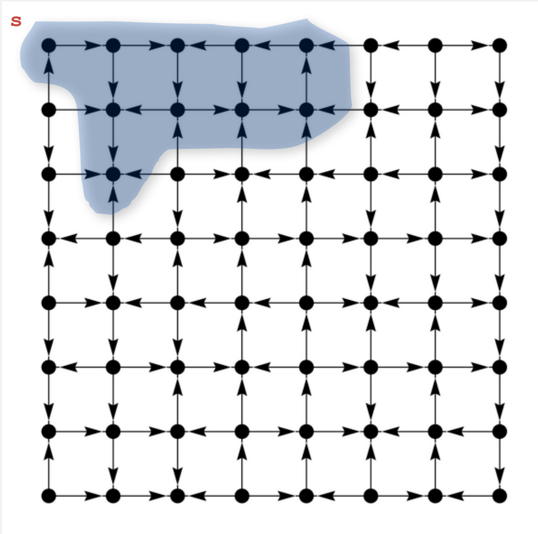
huge number of vertices, small average vertex degree

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

- digraph API
- **digraph search**
- transitive closure
- 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.

- Happens to have 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 (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 isReachable(int v)
   {  return marked[v];  }
}
```

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is reachable from s

Reachability application: program control-flow analysis

Every program is a digraph.

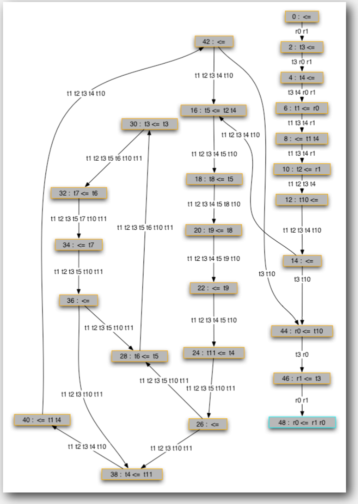
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead code elimination.

Find (and remove) unreachable code.

Infinite loop detection.

Determine whether exit is unreachable.



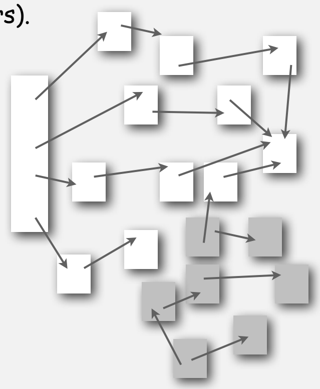
Reachability application: mark-sweep garbage collector

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).

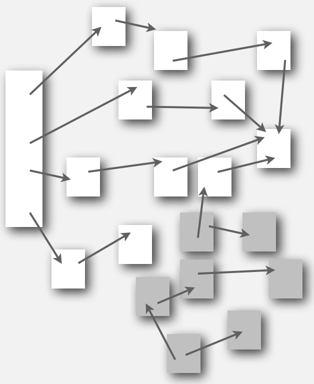


Reachability application: mark-sweep garbage collector

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 (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

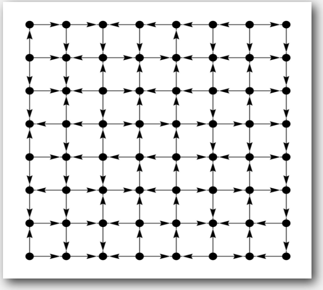
Every undirected graph is a digraph.

- 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.*



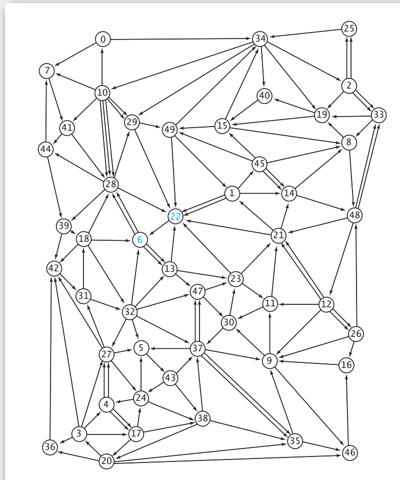
Property. Visits vertices in increasing distance from s.

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>();            // queue of websites to crawl
SET<String> visited = new SET<String>();          // set of visited websites

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

while (!q.isEmpty())
{
   String v = q.dequeue();                        // read in raw html for next website in queue
   StdOut.println(v);
   In in = new In(v);
   String input = in.readAll();

   String regexp = "http://(\\w+\\.)*(\\w+)";
   Pattern pattern = Pattern.compile(regexp);     // use regular expression to find all URLs
   Matcher matcher = pattern.matcher(input);      // in website of form http://xxx.yyy.zzz
   while (matcher.find())
   {
      String w = matcher.group();
      if (!visited.contains(w))
      {
         visited.add(w);                          // if unvisited, mark as visited
         q.enqueue(w);                            // and put on queue
      }
   }
}
```

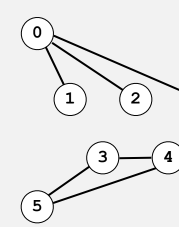

- digraph API
- digraph search
- **transitive closure**
- topological sort
- strong components

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.




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

Digraph-processing challenge 1

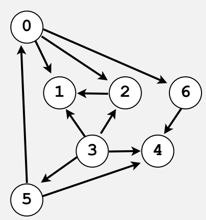
Problem. Is there a directed path from `v` to `w` ?

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)

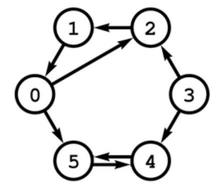


0→1
0→6
0→2
3→4
3→2
5→4
5→0
3→5
2→1
6→4
3→1

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.

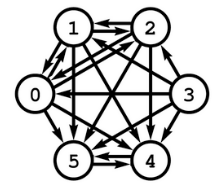
digraph G



	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

digraph G is usually sparse

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

TC(G) is usually dense

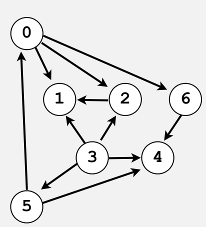
Digraph-processing challenge 1 (revised)

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.



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

Digraph-processing challenge 1 (revised again)

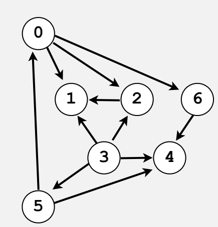
Problem. Is there a directed path from `v` to `w` ?

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



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

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].isReachable(w);  }
}
```

array of `DFSearcher` objects

initialize array

is there a directed path from v to w ?

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

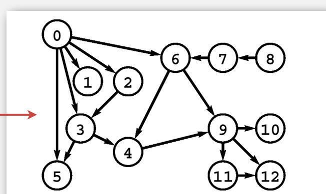
Digraph application: scheduling

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`.

precedence constraint graph



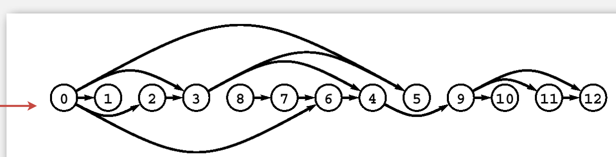
tasks

0. read programming assignment
1. download files
2. write code
3. attend precept

...

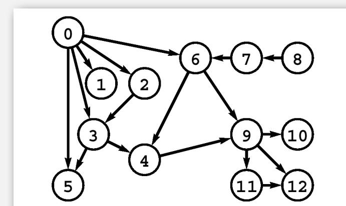
12. sleep

feasible schedule

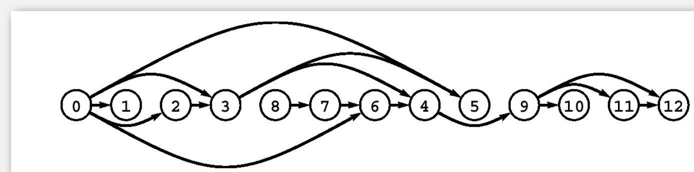


Topological sort

DAG. Directed acyclic graph.



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



Fact. Digraph is a DAG iff no directed cycle.

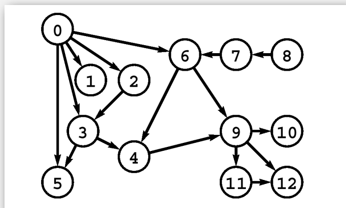
Digraph-processing challenge 2a

Problem. Check that a digraph is a DAG; if so, find a topological order.
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



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

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

Topological sort in a DAG: Java implementation

```
public class TopologicalSorter
{
   private boolean[] marked;
   private Stack<Integer> sorted;

   public TopologicalSorter(Digraph G)
   {
      marked = new boolean[G.V()];
      sorted = new Stack<Integer>();
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) tsort(G, v);
   }

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

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

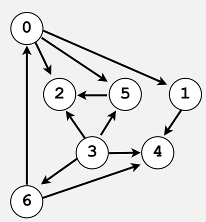
vertices in topological order

reverse DFS postorder

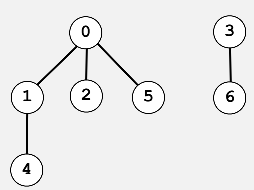
Topological sort in a DAG: trace

Visit means *call* `tsort()` and *leave* means *return from* `tsort()`.

	marked[]	sorted
visit 0:	1 0 0 0 0 0 0	-
visit 1:	1 1 0 0 0 0 0	-
visit 4:	1 1 0 0 1 0 0	-
leave 4:	1 1 0 0 1 0 0	4
leave 1:	1 1 0 0 1 0 0	4 1
visit 2:	1 1 1 0 1 0 0	4 1
leave 2:	1 1 1 0 1 0 0	4 1 2
visit 5:	1 1 1 0 1 1 0	4 1 2
check 2:	1 1 1 0 1 1 0	4 1 2
leave 5:	1 1 1 0 1 1 0	4 1 2 5
leave 0:	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
visit 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
visit 6:	1 1 1 1 1 1 1	4 1 2 5 0
leave 6:	1 1 1 1 1 1 1	4 1 2 5 0 6
leave 3:	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



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



3 6 0 5 2 1 4

Topological sort in a DAG: correctness proof

Proposition. If digraph is a DAG, algorithm yields a topological order.

Pf.

- Algorithm terminates in O(E + V) time since it's just a version of DFS.
- Consider any edge `v→w`. When `tsort(G, v)` is called,
  - Case 1: `tsort(G, w)` has already been called and returned. Thus, `w` will appear after `v` in topological order.
  - Case 2: `tsort(G, w)` has not yet been called, so it will get called directly or indirectly by `tsort(G, v)` and it will finish before `tsort(G, v)`. Thus, `w` will appear after `v` in topological order.
  - Case 3: `tsort(G, w)` has already been called, but not returned. Then the function call stack contains a directed path from `w` to `v`. Combining this path with the edge `v→w` yields a directed cycle, contradicting DAG.

Digraph-processing challenge 2b

Problem. Given a digraph, is there a directed cycle?

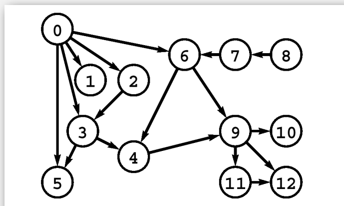
Goal. Linear time.

run DFS-based topological sort algorithm;
if it yields a topological sort, no directed cycle
(can modify code to find cycle)

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.

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



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

Topological sort and 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.

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

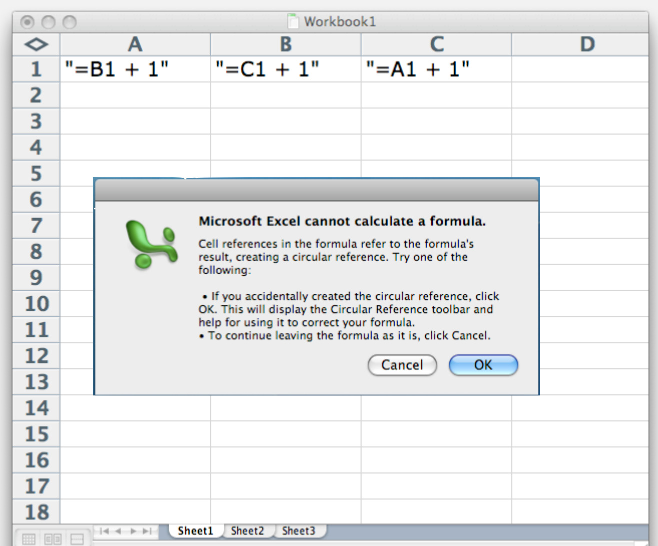
```
public class B extends C
{
   ...
}
```

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

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


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
```

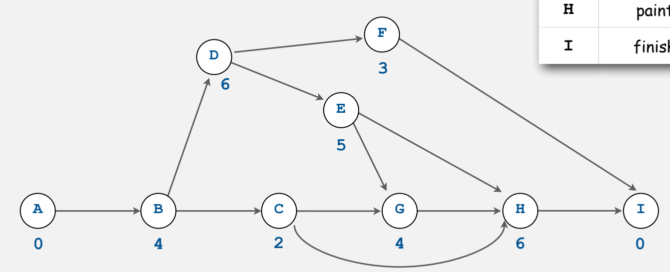

Topological sort application: precedence scheduling

Precedence scheduling.

- Task `v` takes `time[v]` units of time.
- Can work on jobs in parallel.
- Precedence constraints: must finish task `v` before beginning task `w`.
- Goal: finish each task as soon as possible.

index	task	time	prereqs
A	begin	0	-
B	framing	4	A
C	roofing	2	B
D	siding	6	B
E	windows	5	D
F	plumbing	3	D
G	electricity	4	C, E
H	paint	6	C, E
I	finish	0	F, H

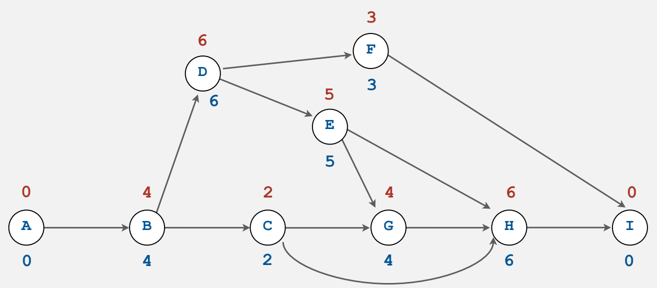
Ex.



Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.

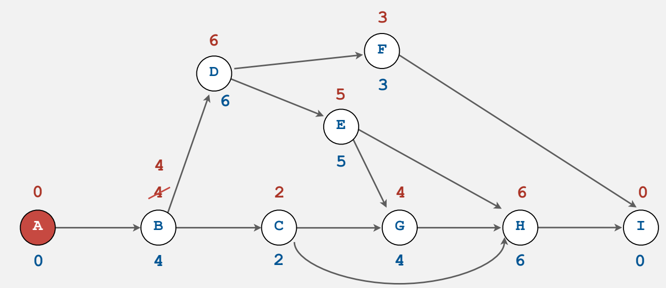
- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



Program Evaluation and Review Technique / Critical Path Method

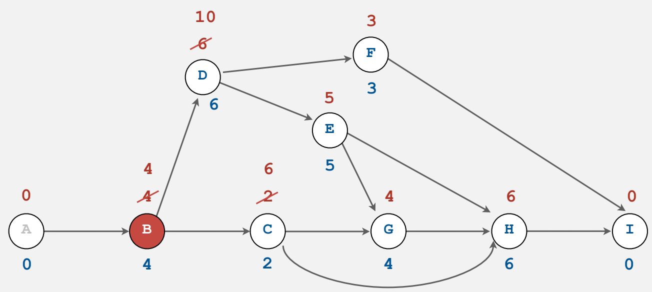
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



PERT/CPM algorithm.

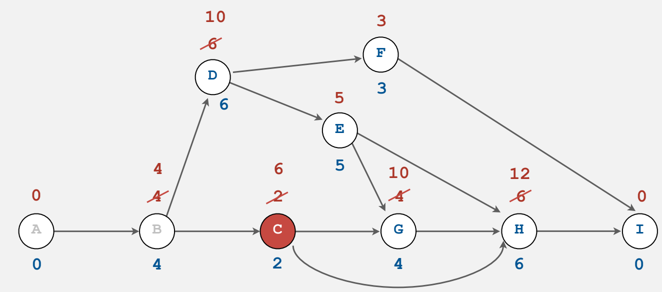
- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.

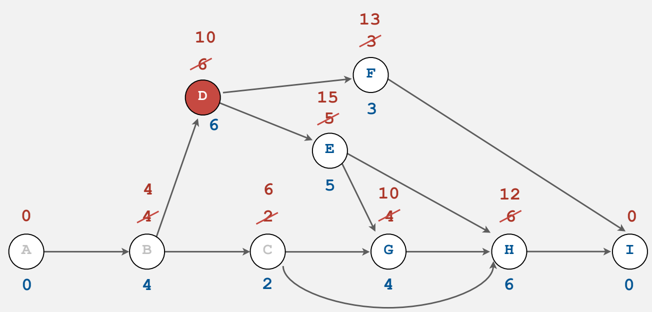
- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.

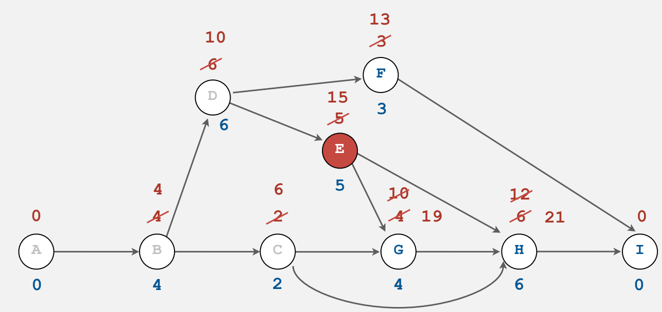
- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



Program Evaluation and Review Technique / Critical Path Method

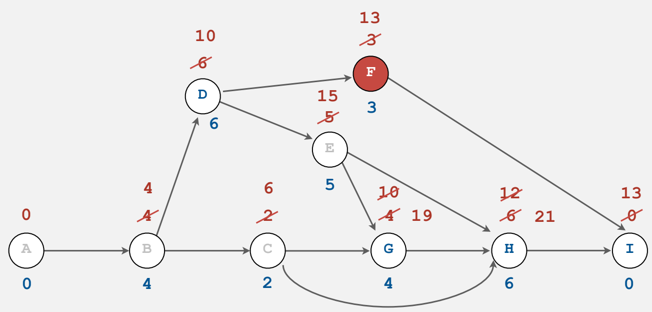
PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



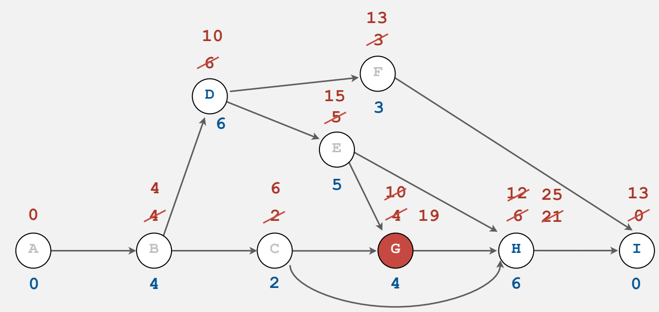
Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.
- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



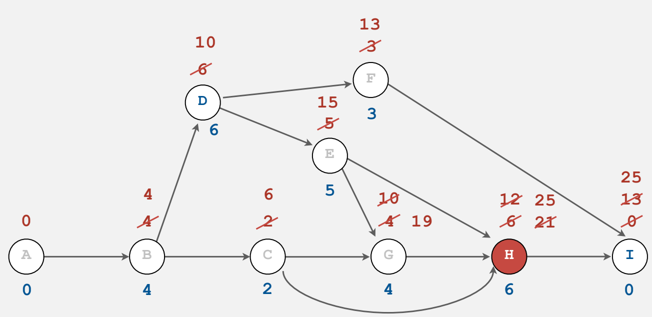
Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.
- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



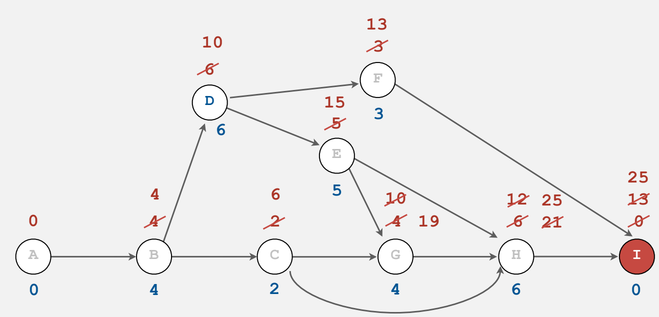
Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.
- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



Program Evaluation and Review Technique / Critical Path Method

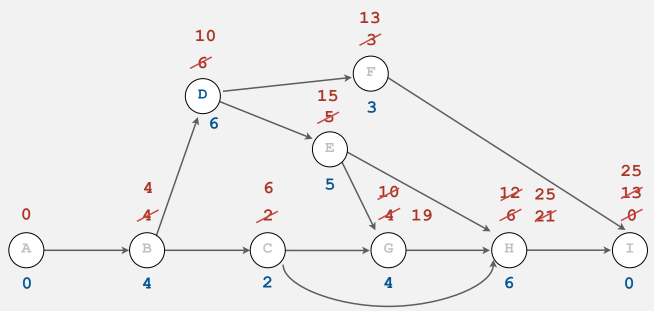
PERT/CPM algorithm.
- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



Program Evaluation and Review Technique / Critical Path Method

PERT/CPM algorithm.

- Compute topological order of vertices.
- Initialize `fin[v] = time[v]` for all vertices `v`.
- Consider vertices `v` in topologically sorted order.
  - for each edge `v→w`, set `fin[w] = max(fin[w], fin[v] + time[w])`



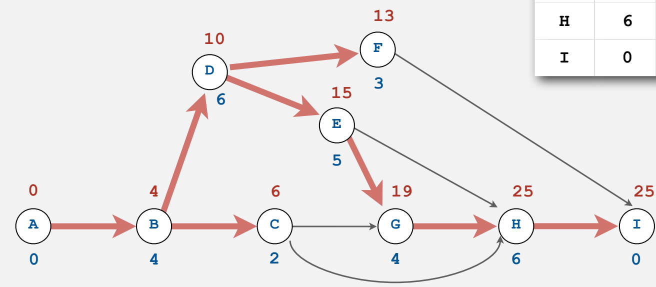
Program Evaluation and Review Technique / Critical Path Method

Critical path. Longest path from source to sink.

To compute:

- Remember vertex that set value (parent-link).
- Work backwards from sink.

index	time	prereqs	finish
A	0	–	0
B	4	A	4
C	2	B	6
D	6	B	10
E	5	D	15
F	3	D	13
G	4	C, E	19
H	6	C, E	25
I	0	F, H	25



PERT/CPM: Java implementation

G = DAG of precedence constraints

```
double[] fin = new double[G.V()];
for (int v = 0; v < G.V(); v++)
   fin[v] = time[v];
```

`fin[v]` = finishing time of task v

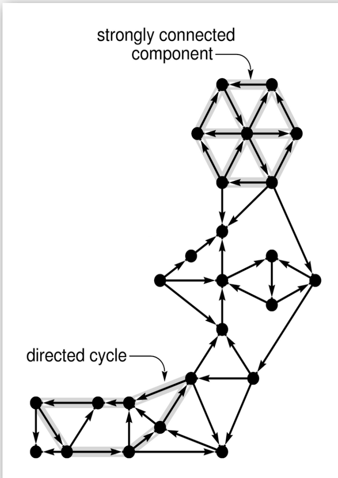
```
TopologicalSorter ts = new TopologicalSorter(G);
for (int v : ts.order())
   for (int w : G.adj(v))
      fin[w] = Math.max(fin[w], fin[v] + time[w]);
```

apply updates to vertices in topological order

Strongly connected components

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

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

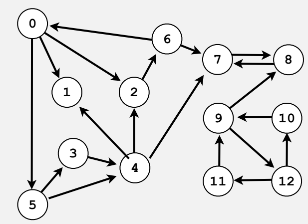


Digraph-processing challenge 3

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.
- Intractable.
- No one knows.
- Impossible.



Digraph-processing challenge 3

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

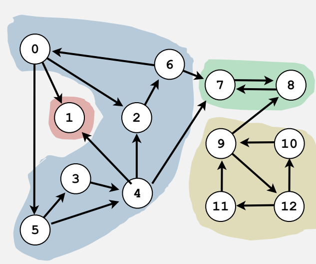
implementation: use DFS twice to find strong components (see textbook)

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.

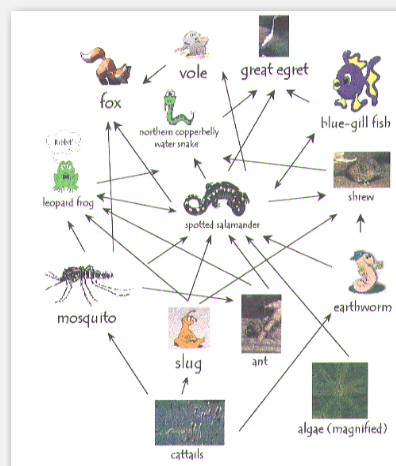
correctness proof

5 strong components



Ecological food web graph

Vertex = species.
Edge: from producer to consumer.

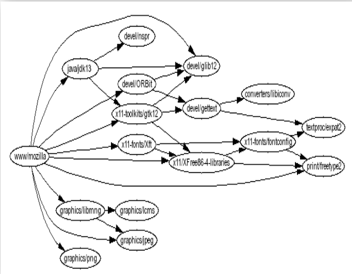


Strong component. Subset of species with common energy flow.

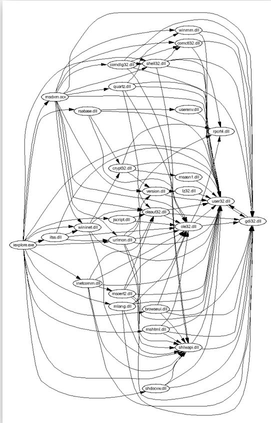
Software module dependency graph

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.

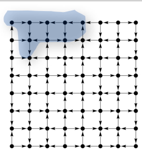
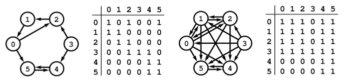
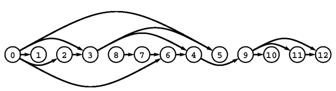
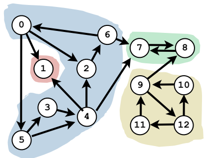
1980s: easy two-pass linear-time algorithm (Kosaraju).

- 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.
- Mehlhorn: needed one-pass algorithm for LEDA.

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)