Algorithms Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.2 Directed Graphs

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

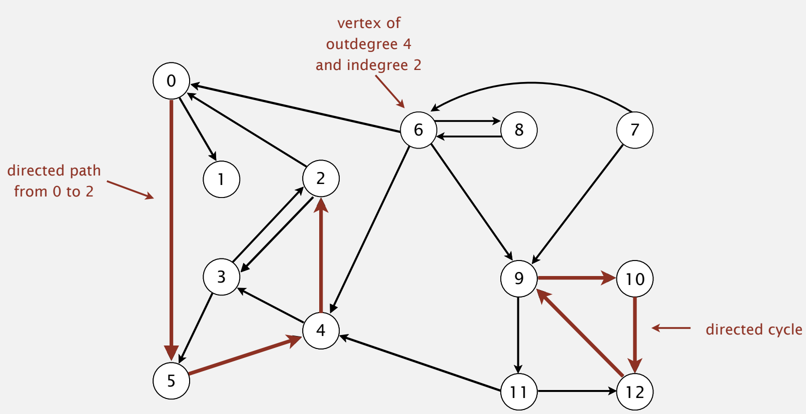
http://algs4.cs.princeton.edu

4.2 Directed Graphs

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

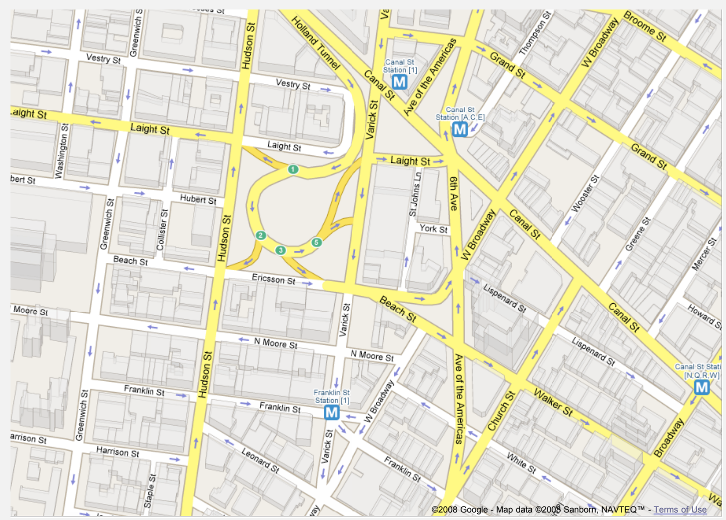
Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.



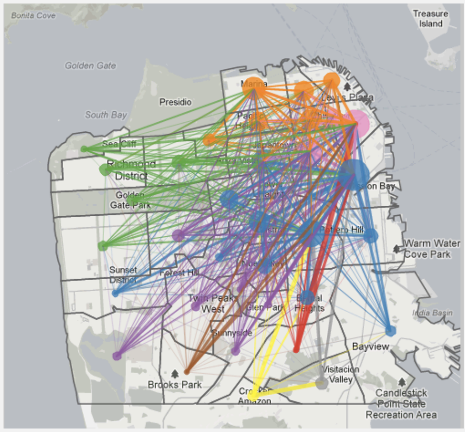
Road network

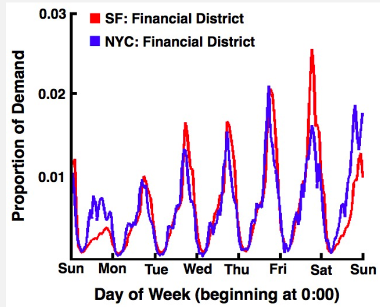
Vertex = intersection; edge = one-way street.



Taxi flow patterns (Uber)

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



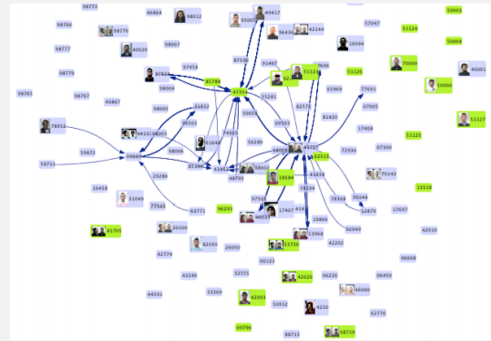


Uber cab service

- Left Digraph: Color is the source neighborhood (no arrows).
- Right Plot: Digraph analysis shows financial districts have similar demand.

Reverse engineering criminal organizations (LogAnalysis)

"The analysis of reports supplied by mobile phone service providers makes it possible to reconstruct the network of relationships among individuals, such as in the context of criminal organizations. It is possible, in other terms, to unveil the existence of criminal networks, sometimes called rings, identifying actors within the network together with their roles" — Cantanese et. al



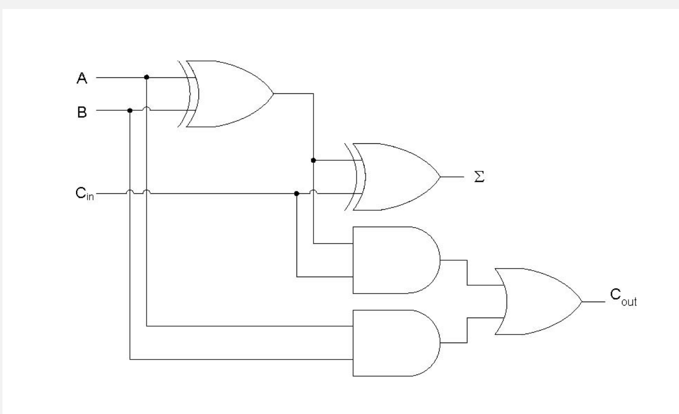
Field	Description
IMEI	IMEI code MS
called	called user
calling	calling user
date/time start	date/time start calling (GMT)
date/time end	date/time end calling (GMT)
type	sms, mms, voice, data etc.
IMSI	calling or called SIM card
CGI	Lat. long. BTS company

Table 1 An example of the structure of a log file.

Forensic Analysis of Phone Call Networks, Salvatore Cantanese, http://arxiv.org/abs/1303.1827

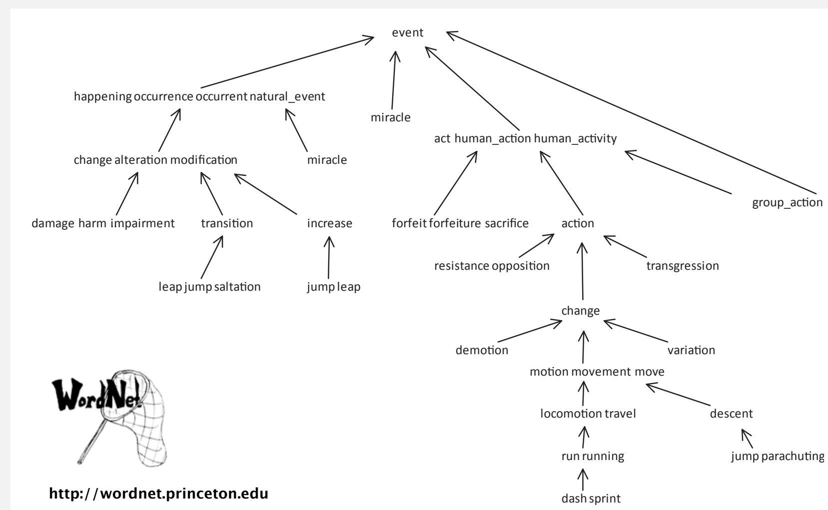
Combinational circuit

Vertex = logical gate; edge = wire.

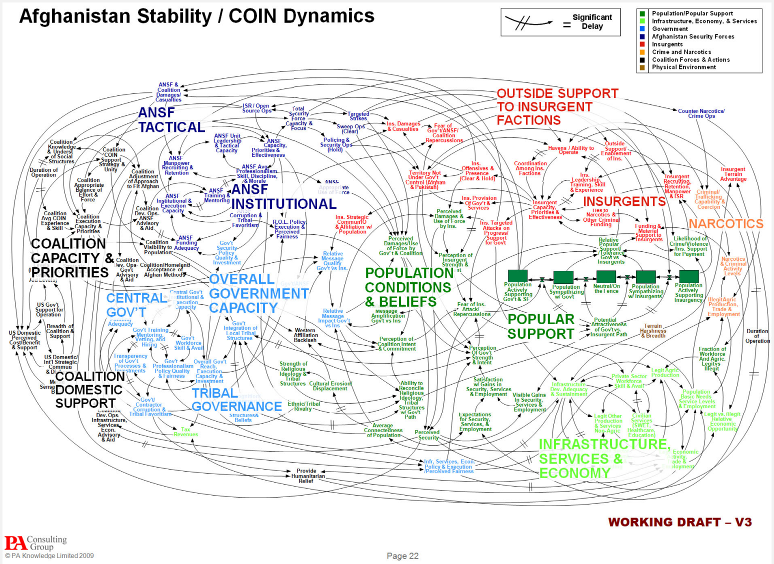


WordNet graph

Vertex = synset; edge = hypernym relationship.



The McChrystal Afghanistan PowerPoint slide



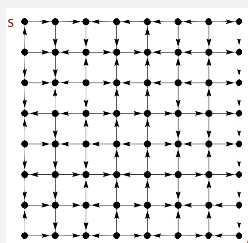
http://www.guardian.co.uk/news/datablog/2010/apr/29/mcchrystal-afghanistan-powerpoint-slide

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 a digraph so that all edges point upwards?

Strong connectivity. Is there a directed path between all pairs of vertices?

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

PageRank. What is the importance of a web page?

http://algs4.cs.princeton.edu

4.2 Directed Graphs

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

Digraph API

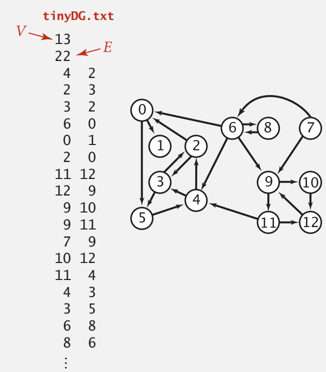
	public class Digraph	
	Digraph(int V)	*create an empty digraph with V vertices*
	Digraph(In in)	*create a digraph from input stream*
void	addEdge(int v, int w)	*add a directed edge v→w*
Iterable<Integer>	adj(int v)	*vertices pointing from v*
int	V()	*number of vertices*
int	E()	*number of edges*
Digraph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

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

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

read digraph from input stream

print out each edge (once)

Digraph API




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

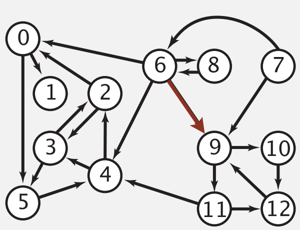
```
In in = new In(args[0]);
Digraph G = new Digraph(in);

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

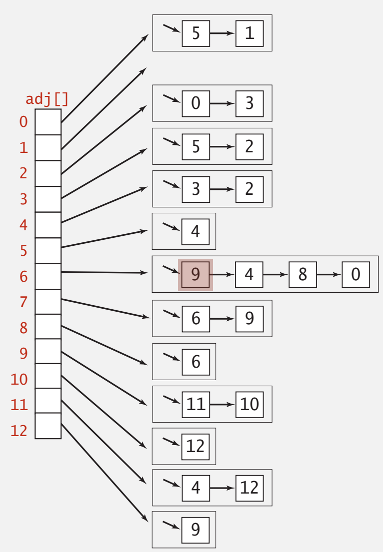
read digraph from input stream

print out each edge (once)

Adjacency-lists digraph representation

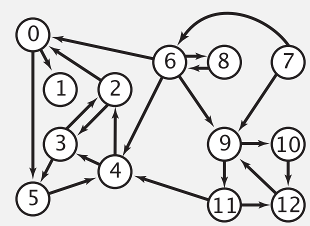
Maintain vertex-indexed array of lists.

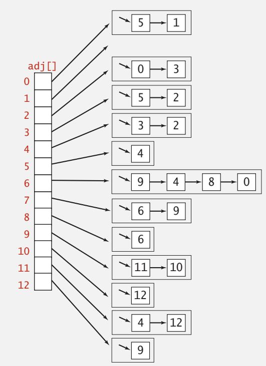




Do you slumber?

Suppose we are given an arbitrary Digraph G and a path of length V given by int[] P.





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

pollEv.com/jhug text to **37607**

Q: What is the worst case run time to check validity of a path P for a general graph with E edges and V vertices?

A. E [41138]

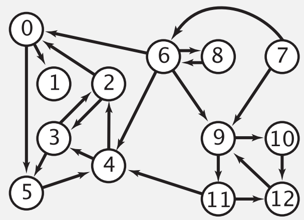
B. V [41142]

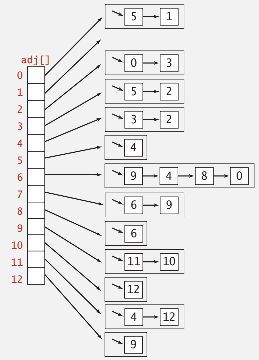
C. EV [41146]

D. E+V [41182]

Do you slumber?

Suppose we are given an arbitrary `Digraph` G and a path of length V given by `int[]` P.





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

pollEv.com/jhug   text to **37607**

Q: What is the worst case run time to check validity of a path P for a general graph with V vertices?

A. 1
B. V
C. V^2

Adjacency-lists graph representation (review): Java implementation

```
public class Graph
{
   private final int V;
   private final Bag<Integer>[] adj;

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

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

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

adjacency lists

create empty graph with V vertices

add edge v–w

iterator for vertices adjacent to v

Adjacency-lists digraph representation: Java implementation

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

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

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

   }

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

adjacency lists

create empty digraph with V vertices

add edge v→w

iterator for vertices pointing from v

Digraph representations

In practice. Use adjacency-lists representation.

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

huge number of vertices, small average vertex degree

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

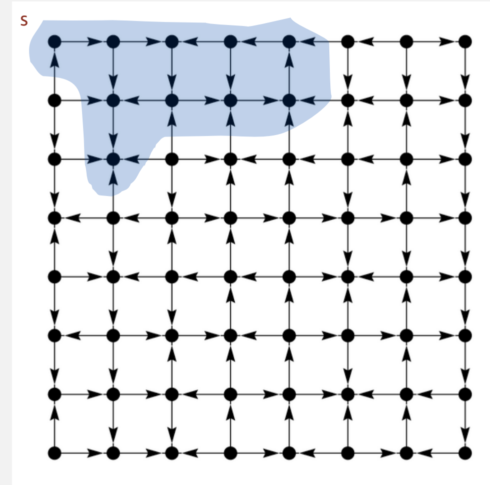
† disallows parallel edges

4.2 Directed Graphs

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

Reachability

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



Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

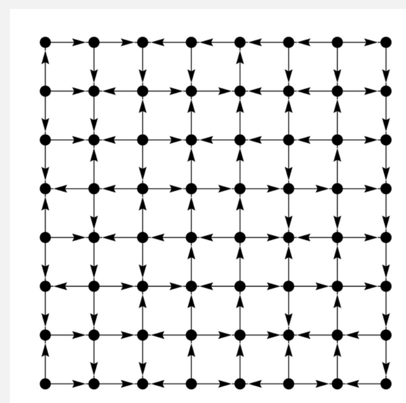
DFS (to visit a vertex v)

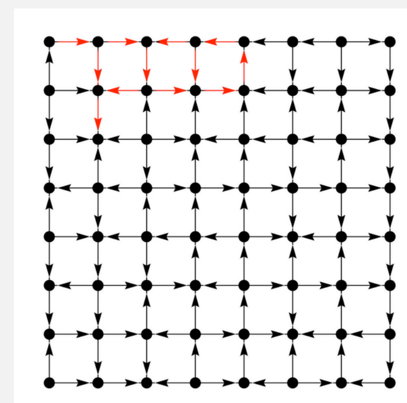
Mark v as visited.
Recursively visit all unmarked vertices w pointing from v.

Difficulty level.

- Exactly the same problem for computers.
- Harder for humans than undirected graphs.
  - Edge interpretation is context dependent!

The man-machine





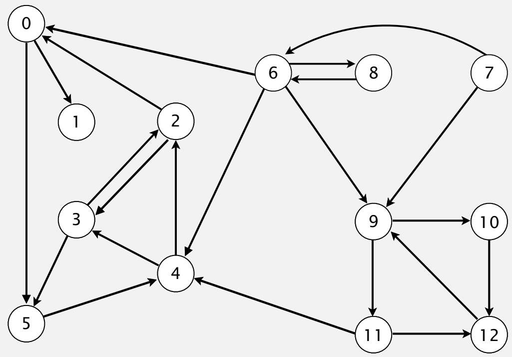
Difficulty level.

- Exactly the same problem for computers.
- Harder for humans than undirected graphs.
  - Edge interpretation is context dependent!

Depth-first search demo

To visit a vertex v :

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



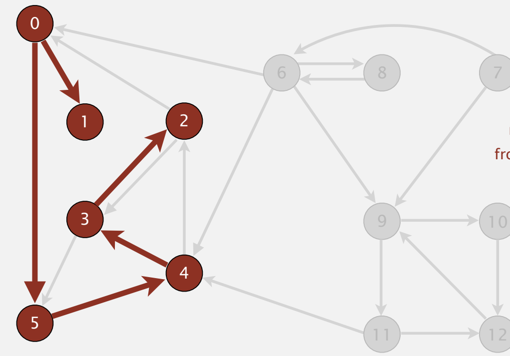
a directed graph

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

Depth-first search demo

To visit a vertex v :

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



reachable from 0

reachable from vertex 0

v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Depth-first search (in undirected graphs)

Recall code for undirected graphs.

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

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

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

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

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is connected to s

Depth-first search (in directed graphs)

Code for directed graphs identical to undirected one.
[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.

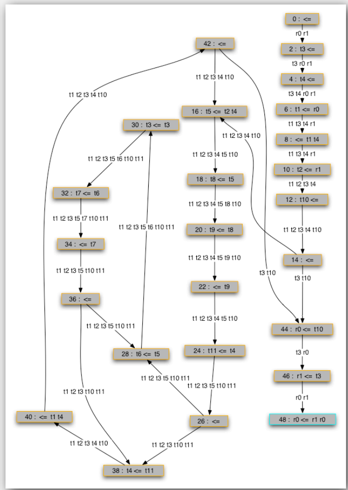
Find (and remove) unreachable code.

- `Cow.java:5: unreachable statement`

Infinite-loop detection.

Determine whether exit is unreachable.

- Trivial?
- Doable by student?
- Doable by expert?
- Intractable?
- Unknown?
- Impossible?



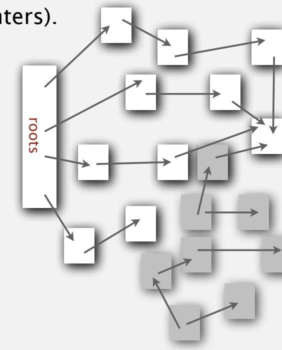
Reachability application: mark-sweep garbage collector

Every data structure is a digraph.

- Vertex = object.
- Edge = reference.

Roots. Objects known to be directly accessible by program (e.g., stack).

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

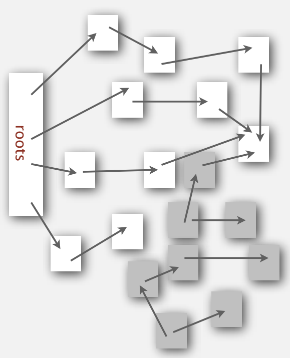


Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).



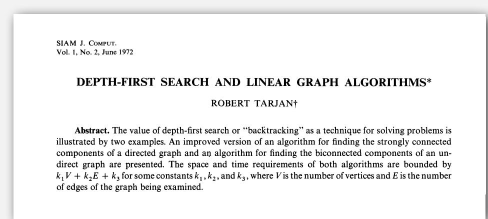
Depth-first search in digraphs summary

DFS enables direct solution of simple digraph problems.

- ✓ Reachability.
- Path finding.
- Topological sort.
- Directed cycle detection.

Basis for solving difficult digraph problems.

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


SIAM J. Comput.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

Abstract. The value of depth-first search or "backtracking" as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirected graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2, and k_3, where V is the number of vertices and E is the number of edges of the graph being examined.

Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

BFS (from source vertex s)

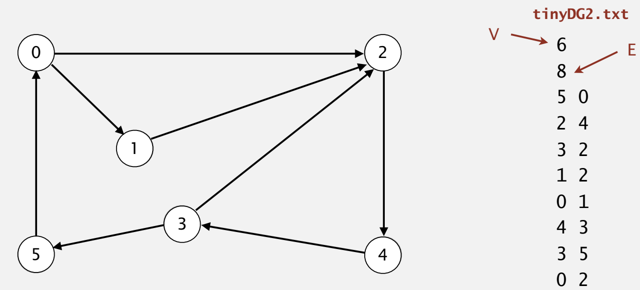
Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
- **remove the least recently added vertex v**
- **for each unmarked vertex pointing from v:**
  add to queue and mark as visited.

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

Directed breadth-first search demo

Repeat until queue is empty:

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

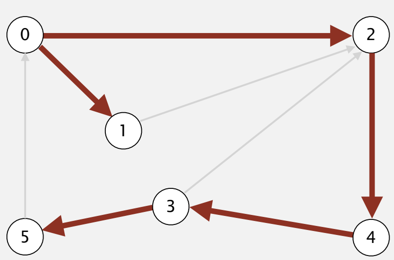


graph G

Directed breadth-first search demo

Repeat until queue is empty:

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



v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

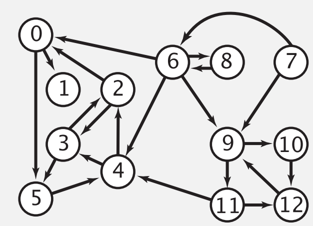
done

Multiple-source shortest paths

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

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

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



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

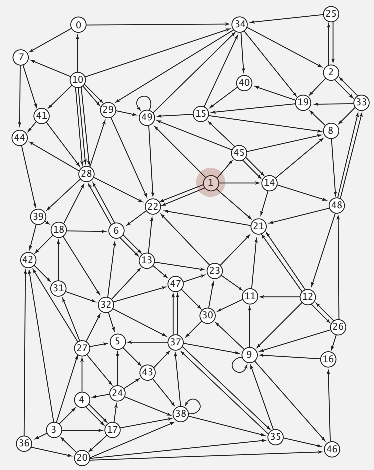
A. Use BFS, but initialize by enqueuing all source vertices.

Breadth-first search in digraphs application: web crawler

Goal. Crawl web, starting from some root web page, say `www.princeton.edu`.

Solution. [BFS with implicit digraph]

- Choose root web page as source s.
- Maintain a `Queue` of websites to explore.
- Maintain a SET of discovered websites.
- Dequeue the next website and enqueue websites to which it links (provided you haven't done so before).



Q. Why not use DFS?

Bare-bones web crawler: Java implementation

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

String root = "http://www.princeton.edu";
queue.enqueue(root);                              // start crawling from root website
marked.add(root);

while (!queue.isEmpty())
{
   String v = queue.dequeue();
   StdOut.println(v);
   In in = new In(v);                             // read in raw html from next website in queue
   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())                         // [crude pattern misses relative URLs]
   {
      String w = matcher.group();
      if (!marked.contains(w))
      {
         marked.add(w);
         queue.enqueue(w);                        // if unmarked, mark it and put on the queue
      }
   }
}
```


BFS Webcrawler Output

http://www.princeton.edu
http://www.w3.org
http://ogp.me
http://giving.princeton.edu
http://www.princetonartmuseum.org
http://www.goprincetontigers.com
http://library.princeton.edu
http://helpdesk.princeton.edu
http://tigernet.princeton.edu
http://alumni.princeton.edu
http://gradschool.princeton.edu
http://vimeo.com
http://princetonusg.com
http://artmuseum.princeton.edu
http://jobs.princeton.edu
http://odoc.princeton.edu
http://blogs.princeton.edu
http://www.facebook.com
http://twitter.com
http://www.youtube.com
http://deimos.apple.com
http://qeprize.org
http://en.wikipedia.org
...

DFS Webcrawler Output

http://www.princeton.edu
http://deimos.apple.com [dead end]
http://www.youtube.com
http://www.google.com
http://news.google.com
http://csi.gstatic.com
http://googlenewsblog.blogspot.com
http://labs.google.com
http://groups.google.com
http://img1.blogblog.com
http://feeds.feedburner.com
http://buttons.googlesyndication.com
http://fusion.google.com
http://insidesearch.blogspot.com
http://agoogleaday.com
http://static.googleusercontent.com
http://searchresearch1.blogspot.com
http://feedburner.google.com
http://www.dot.ca.gov
http://www.getacross80.com
http://www.TahoeRoads.com
http://www.LakeTahoeTransit.com
http://www.laketahoe.com
http://ethel.tahoeguide.com
...

http://algs4.cs.princeton.edu

4.2 Directed Graphs

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

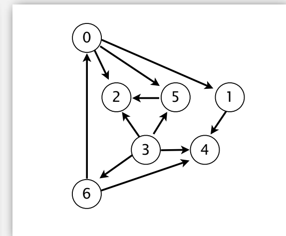
Precedence scheduling

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

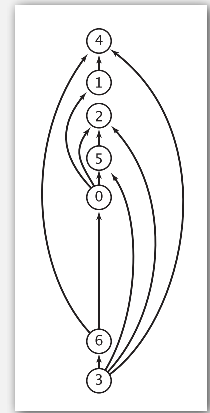
Digraph model. vertex = task; edge = precedence constraint.

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

tasks



precedence constraint graph



feasible schedule

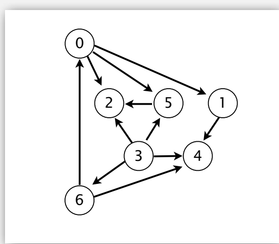
Topological sort

DAG. Directed acyclic graph.

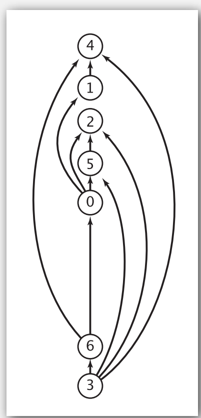
Topological sort. Redraw DAG so all edges point upwards.

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

directed edges



DAG

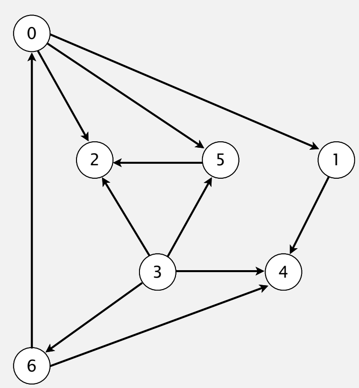


topological order

Solution. DFS. What else?

Topological sort demo

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

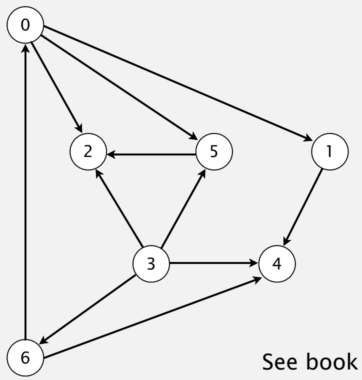


0→5
0→2
0→1
3→6
3→5
3→4
5→4
6→4
6→0
3→2
1→4

a directed acyclic graph

Topological sort intuitive proof

- Run depth-first search.
- Return vertices in reverse postorder.
- Why does it work?
  - Last item in postorder has indegree 0. Good starting point.
  - Second to last can only be pointed to by last item. Good follow-up.
  - ...



postorder

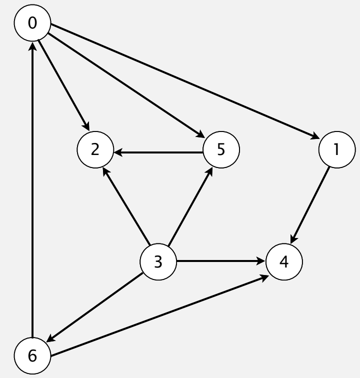
4 1 2 5 0 6 3

topological order

3 6 0 5 2 1 4

See book / online slides for foolproof full proof.

Topological sort demo

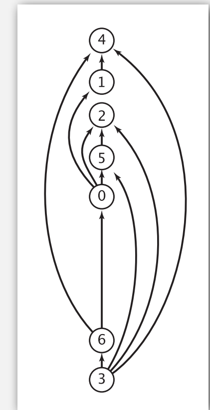


postorder

4 1 2 5 0 6 3

topological order

3 6 0 5 2 1 4

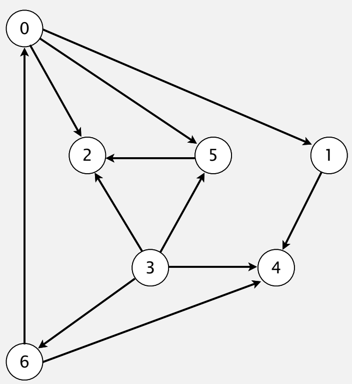


topological order

pollEv.com/jhug text to **37607**

Q: Is the reverse postorder the only valid topological order for this graph?
A. No [452392]
B. Yes [452393]

Topological sort demo

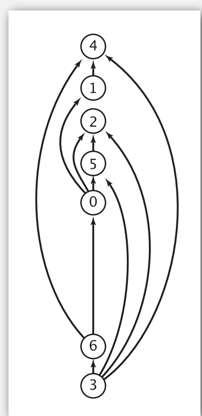


postorder

4 1 2 5 0 6 3

topological order

3 6 0 5 2 1 4



topological order

pollEv.com/jhug text to **37607**

Q: Is the reverse postorder the only valid topological order for this graph?
A. No [452392]

Example: Could move 1 down one step. 0 → 1 still points up.

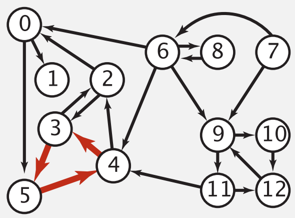
Depth-first search order

```
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"

Directed cycle detection

Proposition. A digraph has a topological order iff no directed cycle.
Pf.
- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.



a digraph with a directed cycle

Goal. Given a digraph, find a directed cycle.
Solution. DFS. What else? See textbook.

Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

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

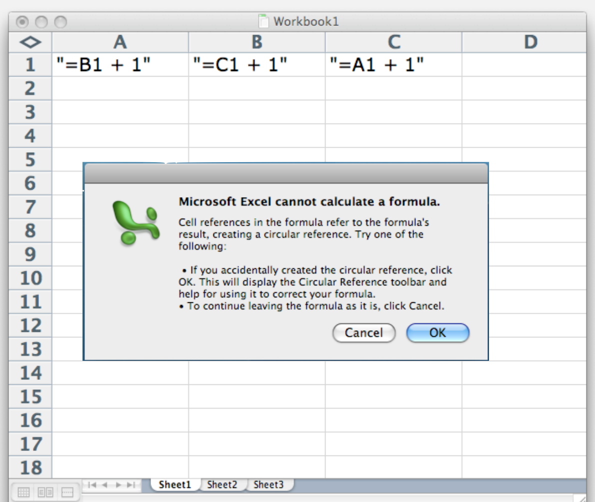
```
public class B extends C
{
   ...
}
```

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

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


Directed cycle detection application: spreadsheet recalculation

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



http://algs4.cs.princeton.edu

4.2 Directed Graphs

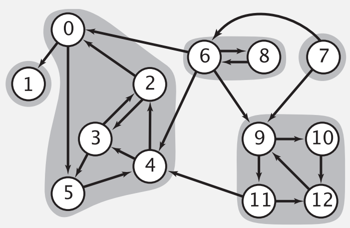
- introduction
- digraph API
- digraph search
- topological sort
- **strong components**

Strongly-connected components

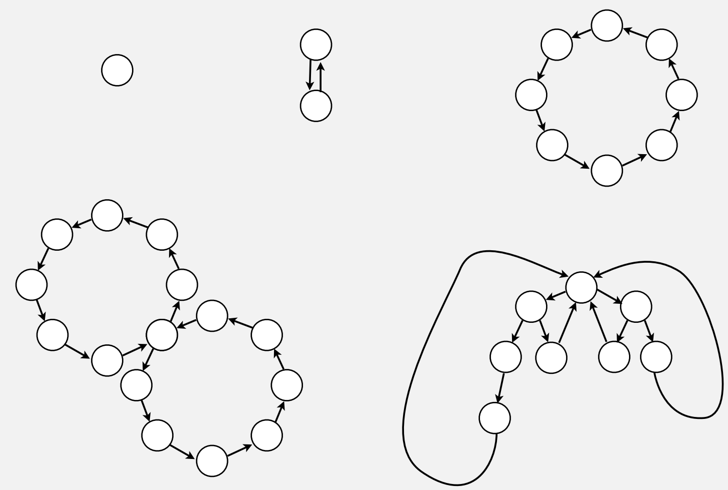
Def. Vertices v and w are strongly connected if there is both a directed path from v to w and a directed path from w to v. Every node is strongly connected to itself.

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.

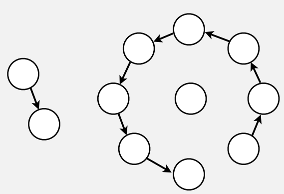


Examples of strongly-connected digraphs: 1 strong component



Strongly-connected components

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



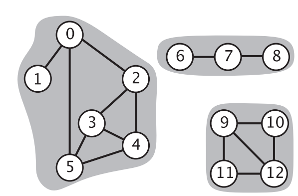
pollEv.com/jhug text to **37607**

Q: How many strong components does a DAG on V vertices and E edges have?

A. 0 [452453]
B. 1 [452459]
C. E [452460]
D. V [452461]

Connected components vs. strongly-connected components

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



3 connected components

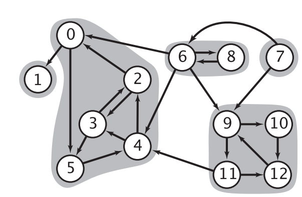
connected component id (easy to compute with DFS)

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

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

constant-time client connectivity query

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



5 strongly-connected components

strongly-connected component id (how to compute?)

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

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

constant-time client strong-connectivity query

Strongly connected components

Analysis of Yahoo Answers

- Edge is from asker to answerer.
- "A large SCC indicates the presence of a community where many users interact, directly or indirectly."

Table 1: Summary statistics for selected QA networks

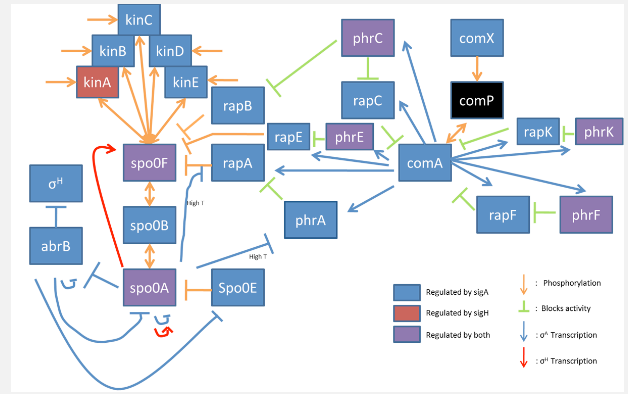
Category	Nodes	Edges	Avg. deg.	Mutual edges	SCC
Wrestling	9,959	56,859	7.02	1,898	13.5%
Program.	12,538	18,311	1.48	0	0.01%
Marriage	45,090	164,887	3.37	179	4.73%

Knowledge sharing and yahoo answers: everyone knows something, Adamic et al (2008)

Strongly connected components

Understanding biological control systems

- *Bacillus subtilis* spore formation control network.
- SCC constitutes a functional module.



Josh Hug: Qualifying exam talk (2008)

Strong components algorithms: brief history

1960s: Core OR problem.

- Widely studied; some practical algorithms.
- Complexity not understood.

1972: linear-time one-pass DFS algorithm (Tarjan).

- Classic algorithm.
- Level of difficulty: Algs4++.
- Demonstrated broad applicability and importance of DFS.

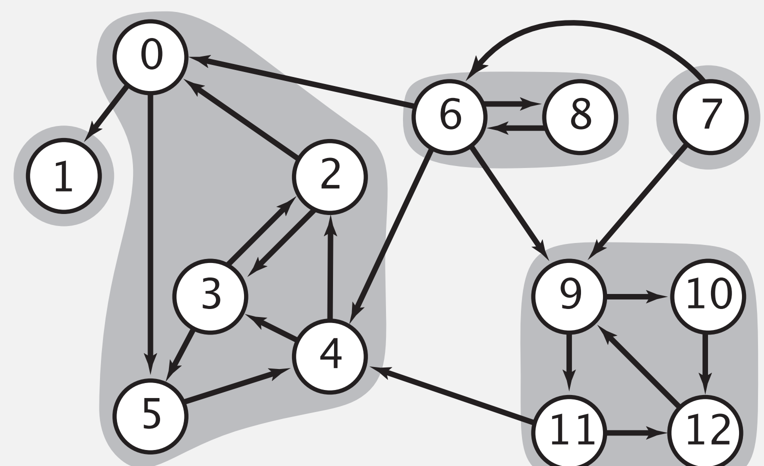
1980s: easy two-pass linear-time algorithm (Kosaraju-Sharir).

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

1990s: easier one-pass linear-time algorithms.

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

Intuitive solution to finding strongly connected components.



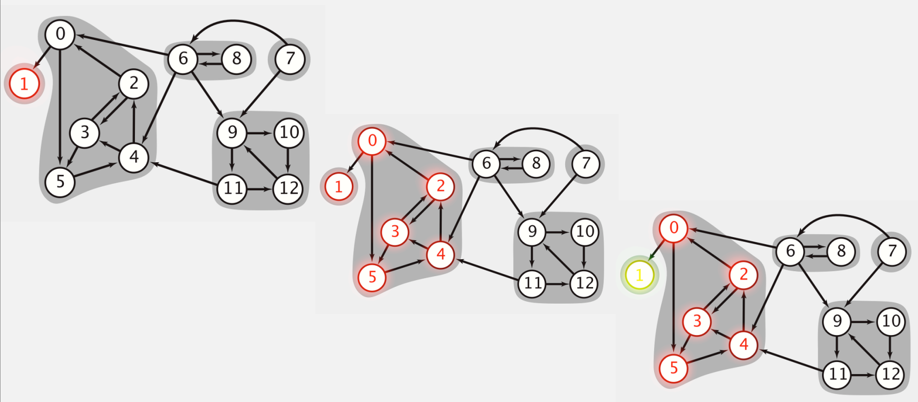
Intuitive solution to finding strongly connected components.

Example

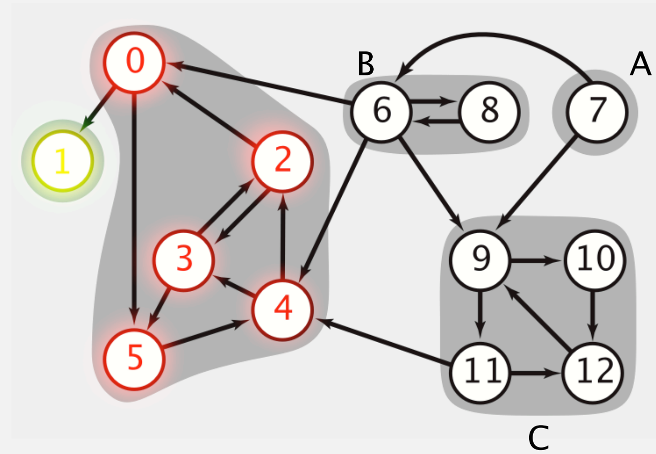
Run DFS(1), get the SCC: {1}.

Run DFS(0), get {0, 1, 2, 3, 4, 5} - not an SCC.

Run DFS(1), then DFS(0), get SCC {1} and SCC {0, 2, 3, 4, 5}.



Intuitive solution to finding strongly connected components.



pollEv.com/jhug text to **37607**

Q: Which DFS call should come next?

```
A. DFS(7)                                   [397963]
B. DFS(6) or DFS(8)                         [398061]
C. DFS(9), DFS(10), DFS(11), or DFS(12)     [398062]
```

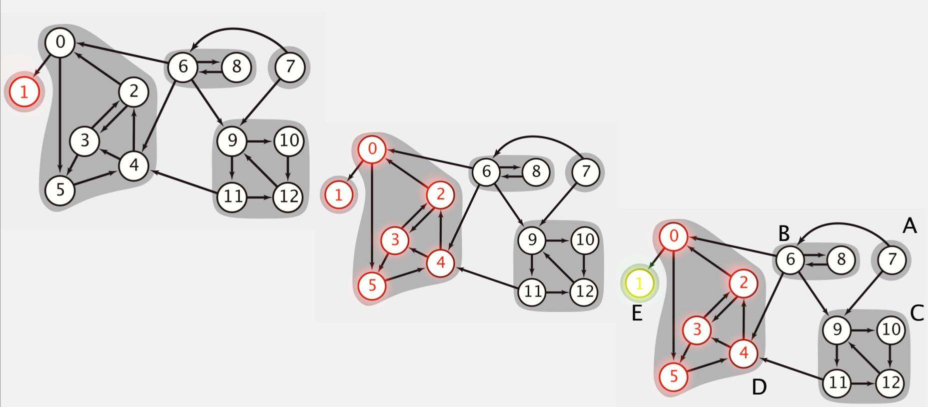

Intuitive solution to finding strongly connected components.

Example

Run DFS(1), get the SCC: {1}.

Run DFS(0), get {0, 1, 2, 3, 4, 5} - not an SCC.

Run DFS(1), then DFS(0), get SCC {1} and SCC {0, 2, 3, 4, 5}.



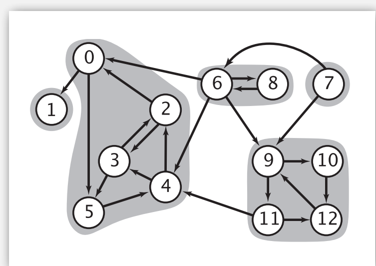
Punchline. A Magic Sequence of DFS calls yields SCC (MSDFSSCC)

Intuitive solution to finding strongly connected components.

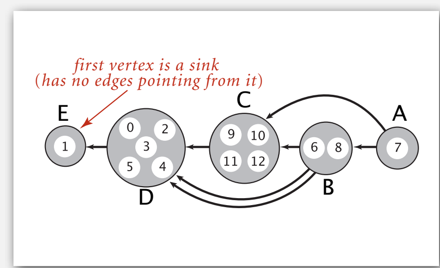
DFS. Calling DFS wantonly is a problem. Never want to leave your SCC.

Starting SCCs. There's always some set of SCCs with outdegree 0, e.g. {1}. Calling DFS on any node in these SCCs finds the SCC.

DFS Order. After calling DFS on all starting SCCs, there's at least one SCC that only points at the starting SCCs.



digraph G and its strong components



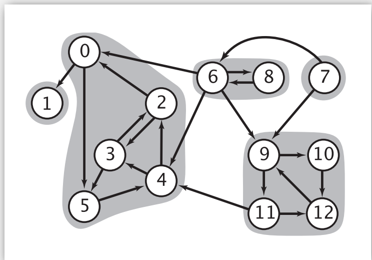
Treat SCCs as one big node. Kernel DAG.
Arrows only connect SCCs. Graph is acyclic.

Kosaraju-Sharir algorithm: intuitive example

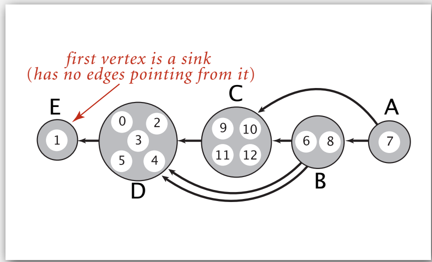
Kernel DAG. Topological sort of kernelDAG(G) is A, B, C, D, E.

MSDFSSCC. Call DFS on element from E, D, C, B, A. Valid MSDFSSCC.
For example, DFS(1), DFS(2), DFS(9), DFS(6), DFS(7).

Summary.
- The MSDFSSCC is given by **reverse of the topological sort** of kernelDAG(G).



digraph G and its strong components



kernel DAG of G. Topological order: A, B, C, D, E.

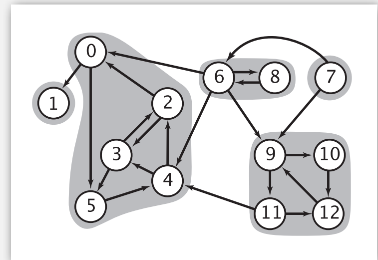
Kosaraju-Sharir algorithm: intuition (general)

???

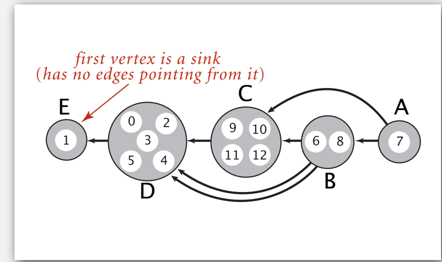
Kernel DAG. MSDFSSCC is given by **reverse of topological sort** of kernelDAG(G).

Reverse Graph Lemma. **Reverse of topological sort** of kernalDAG(G) is given by reverse postorder of G^R (see book), where G^R is G with all arrows flipped around.

Punchline.
- MSDFSSCC: The reverse postorder of G^R.



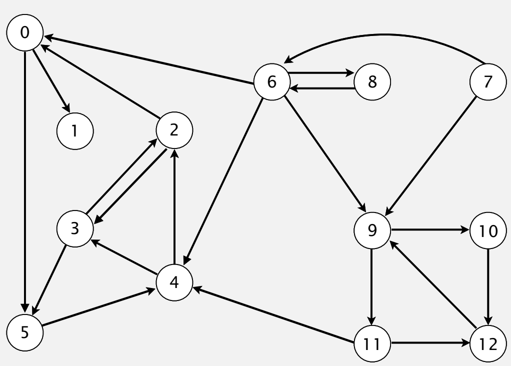
digraph G and its strong components



kernel DAG of G (in reverse topological order)

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.
Phase 2. Run DFS in G, visiting unmarked vertices in reverse postorder of G^R.

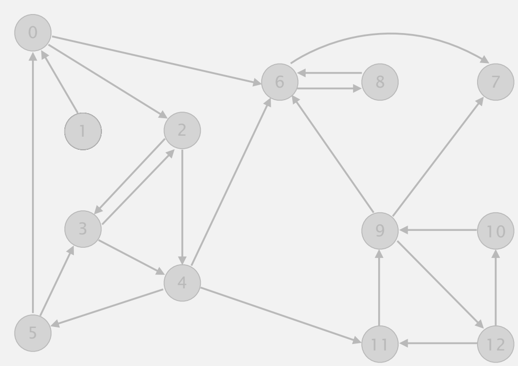


digraph G

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

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

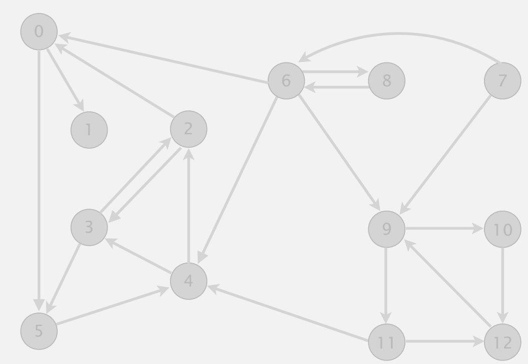


reverse digraph G^R

Kosaraju-Sharir algorithm demo

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

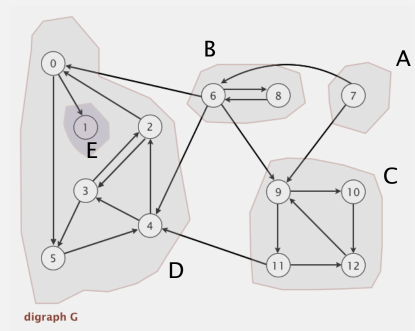
1 0 2 4 5 3 11 9 12 10 6 7 8

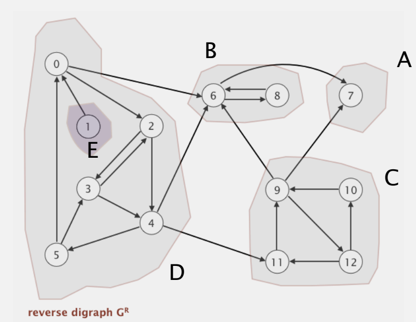


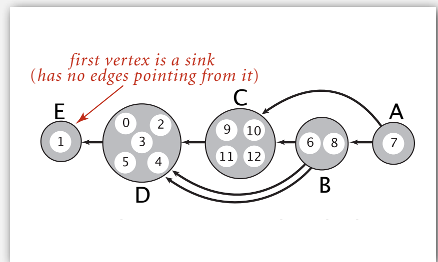
v	id[]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

done

Kosaraju-Sharir algorithm: intuition







E D C B A

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

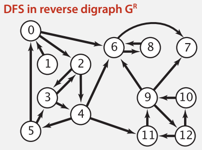
kernel DAG of G (in reverse topological order)

Kosaraju-Sharir algorithm

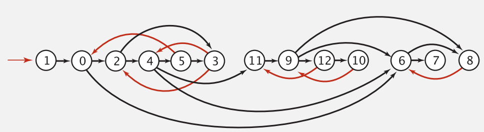
Simple (but mysterious) algorithm for computing strong components.

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

DFS in reverse digraph G^R



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



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

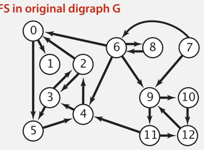
```
dfs(0)
| dfs(6)
| | dfs(8)
| | | check 6
| | 8 done
| | dfs(7)
| | 7 done
| 6 done
| dfs(2)
| | dfs(4)
| | | dfs(11)
| | | | dfs(9)
| | | | | dfs(12)
| | | | | | check 11
| | | | | | dfs(10)
| | | | | | | check 9
| | | | | | 10 done
| | | | | 12 done
| | | | | check 7
| | | | | check 6
...
```


Kosaraju-Sharir algorithm

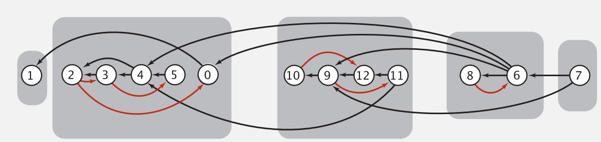
Simple (but mysterious) algorithm for computing strong components.

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

DFS in original digraph G



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



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

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

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

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

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

idarray

Kosaraju-Sharir algorithm

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

Pf.

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

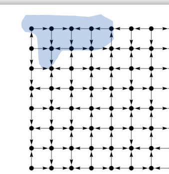
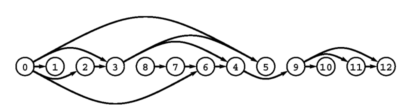
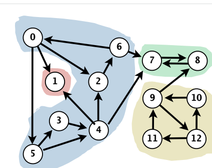
Connected components in an undirected graph (with DFS)

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

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

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

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

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


Strong components in a digraph (with two DFSs)

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

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

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

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


Digraph-processing summary: algorithms of the day

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