4.1 Undirected Graphs

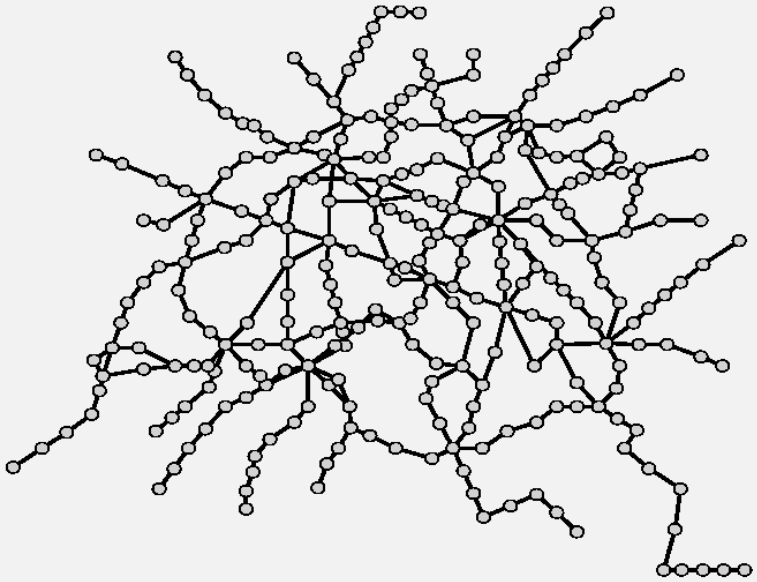
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges

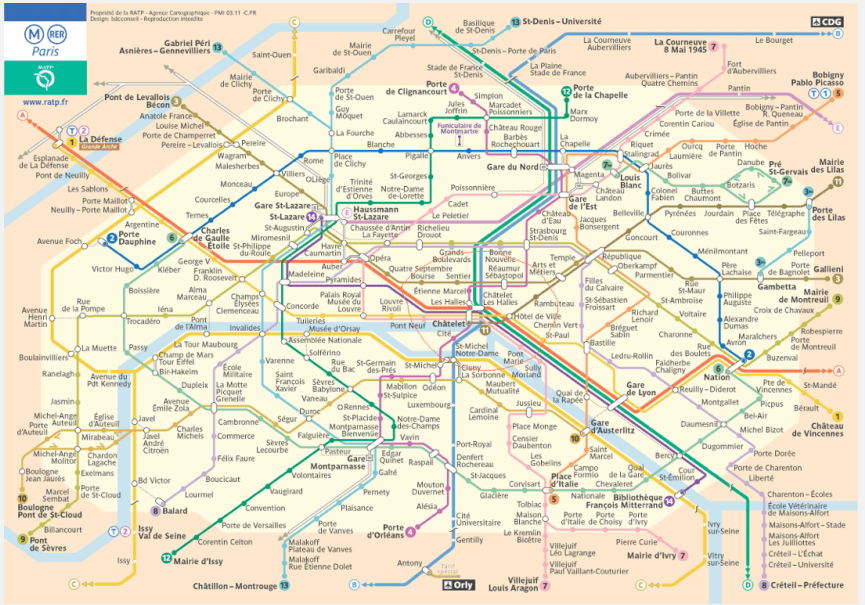
Undirected graphs

Graph. Set of vertices connected pairwise by edges.

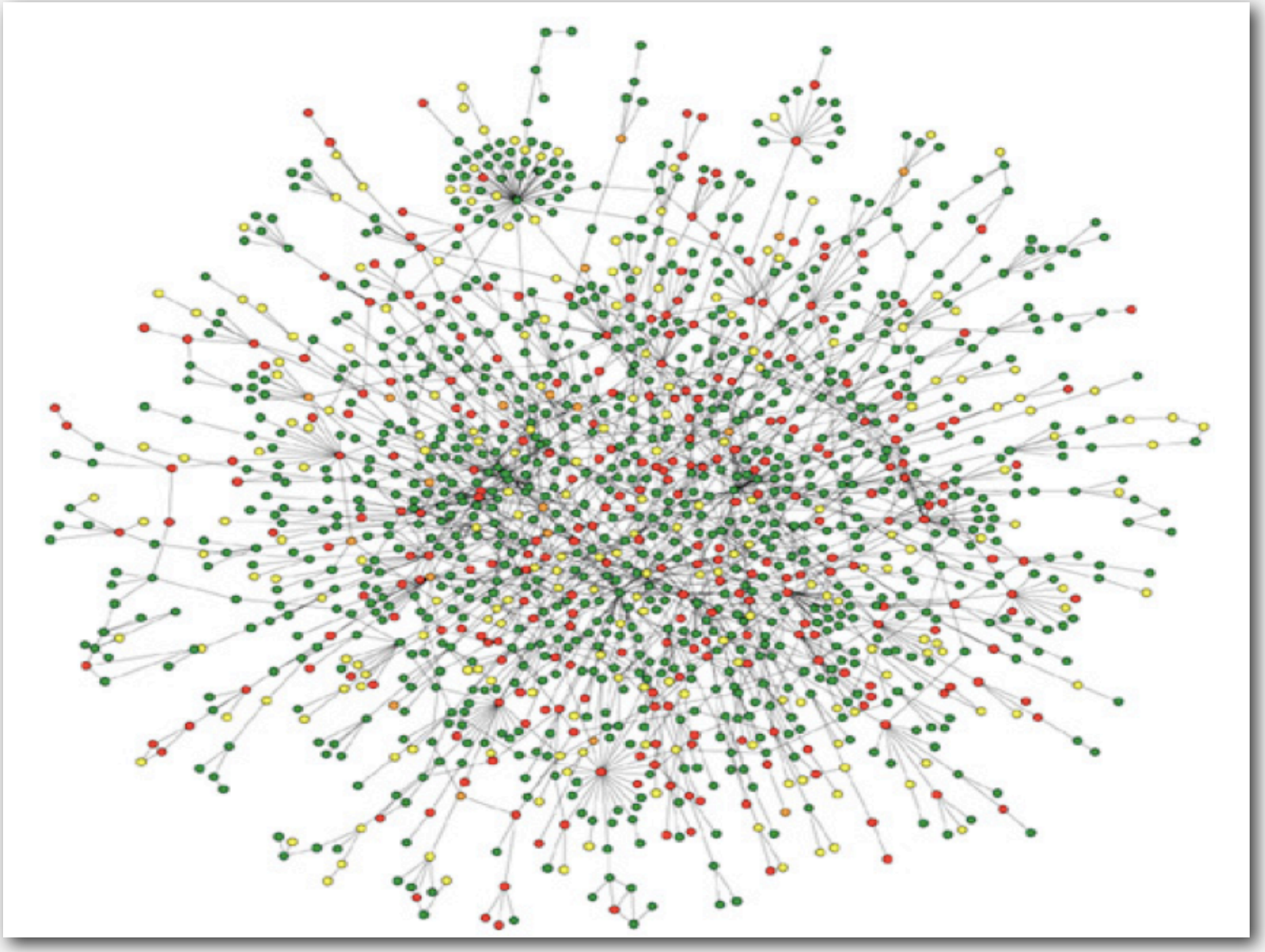
Why study graph algorithms?

- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.





Protein-protein interaction network



Reference: Jeong et al, Nature Review | Genetics

10 million Facebook friends

"Visualizing Friendships" by Paul Butler

Graph applications

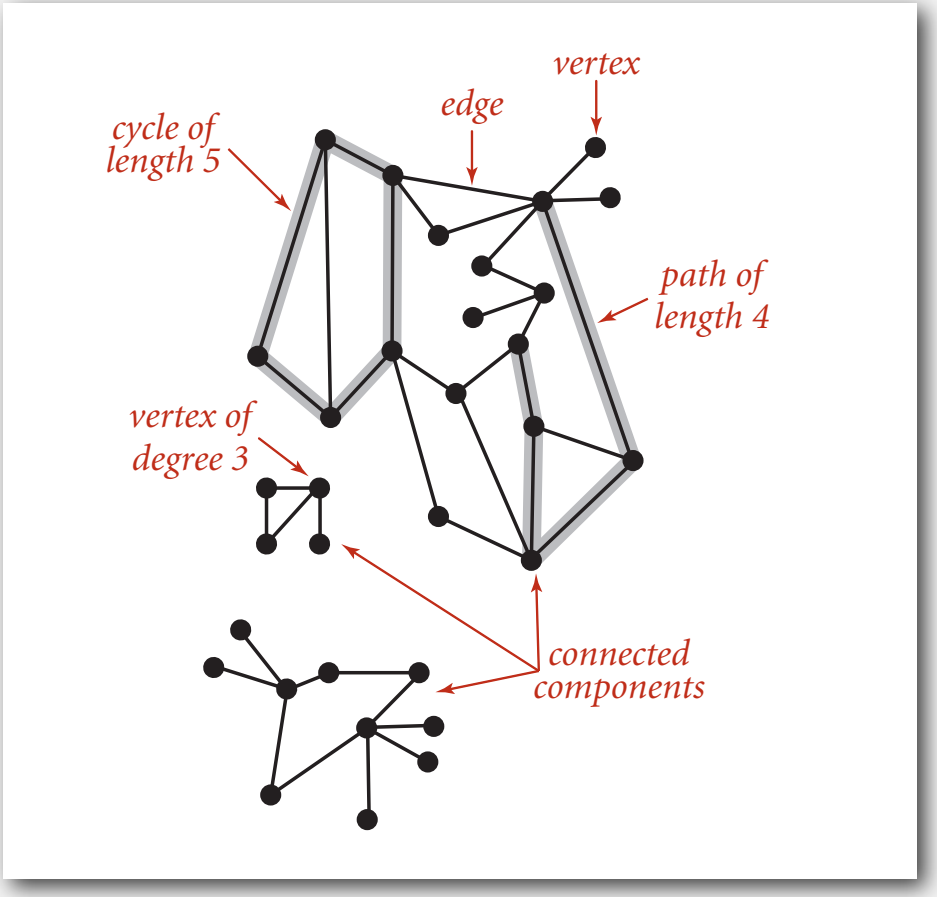
graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
chemical compound	molecule	bond

Graph terminology

Path. Sequence of vertices connected by edges.

Cycle. Path whose first and last vertices are the same.

Two vertices are connected if there is a path between them.



Some graph-processing problems

Path. Is there a path between s and t ?

Shortest path. What is the shortest path between s and t ?

Cycle. Is there a cycle in the graph?

Euler tour. Is there a cycle that uses each edge exactly once?

Hamilton tour. Is there a cycle that uses each vertex exactly once?

Connectivity. Is there a way to connect all of the vertices?

MST. What is the best way to connect all of the vertices?

Biconnectivity. Is there a vertex whose removal disconnects the graph?

Planarity. Can you draw the graph in the plane with no crossing edges?

Graph isomorphism. Do two adjacency lists represent the same graph?

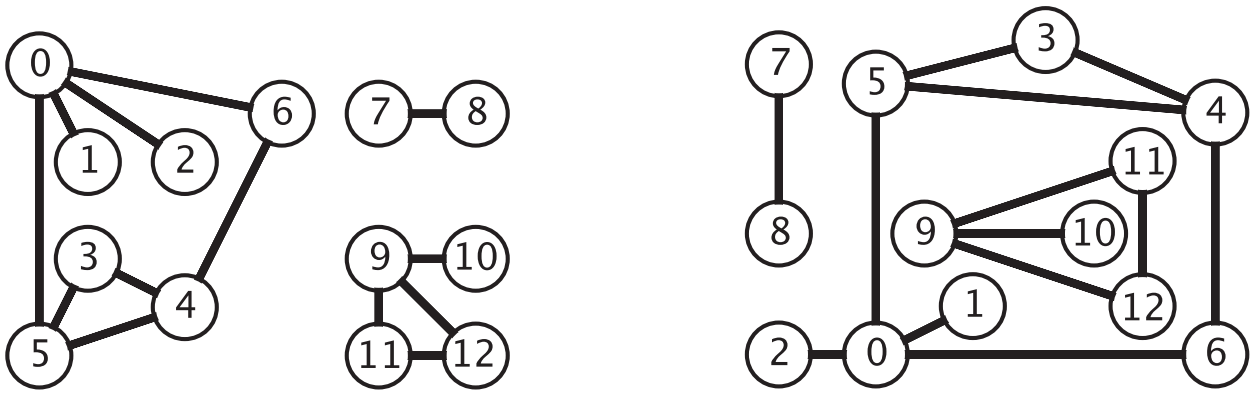
Challenge. Which of these problems are easy? difficult? intractable?

- graph API
- depth-first search
- breadth-first search
- connected components
- challenges

Graph representation

Graph drawing. Provides intuition about the structure of the graph.

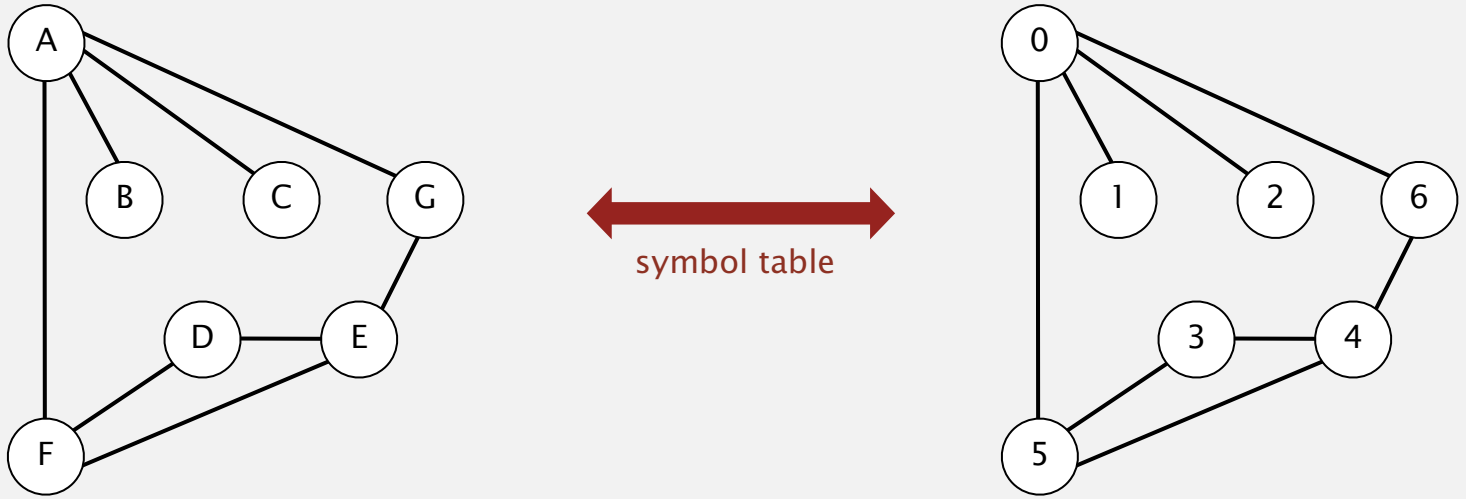
Caveat. Intuition can be misleading.



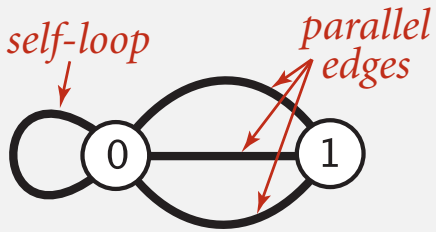
two drawings of the same graph

Vertex representation.

- This lecture: use integers between 0 and $V - 1$.
- Applications: convert between names and integers with symbol table.



Anomalies.



Graph API

public class Graph	
Graph(int V)	*create an empty graph with V vertices*
Graph(In in)	*create a graph from input stream*
void addEdge(int v, int w)	*add an edge v-w*
Iterable<Integer> adj(int v)	*vertices adjacent to v*
int V()	*number of vertices*
int E()	*number of edges*
String toString()	*string representation*

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

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

read graph from input stream

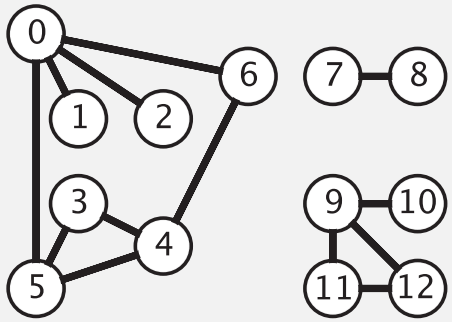
print out each edge (twice)

Graph API: sample client

Graph input format.

tinyG.txt

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




```
% java Test tinyG.txt
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
...
12-11
12-9
```

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

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

read graph from input stream

print out each edge (twice)

Typical graph-processing code

compute the degree of v

```
public static int degree(Graph G, int v)
{
   int degree = 0;
   for (int w : G.adj(v)) degree++;
   return degree;
}
```

compute maximum degree

```
public static int maxDegree(Graph G)
{
   int max = 0;
   for (int v = 0; v < G.V(); v++)
      if (degree(G, v) > max)
         max = degree(G, v);
   return max;
}
```

compute average degree

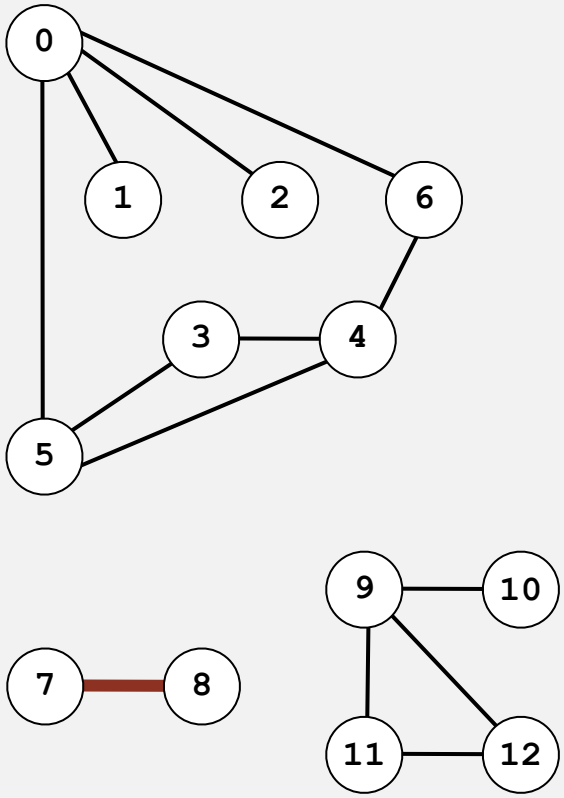
```
public static int avgDegree(Graph G)
{
   return 2 * G.E() / G.V();
}
```

count self-loops

```
public static int numberOfSelfLoops(Graph G)
{
   int count = 0;
   for (int v = 0; v < G.V(); v++)
      for (int w : G.adj(v))
         if (v == w) count++;
   return count/2;
}
```

Set-of-edges graph representation

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

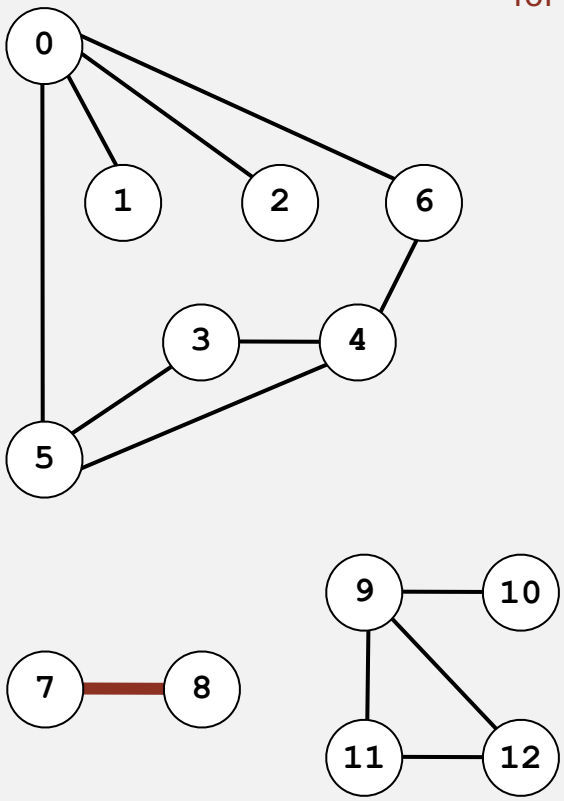


0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Adjacency-matrix graph representation

Maintain a two-dimensional V-by-V boolean array;

for each edge v–w in graph: `adj[v][w] = adj[w][v] = true.`



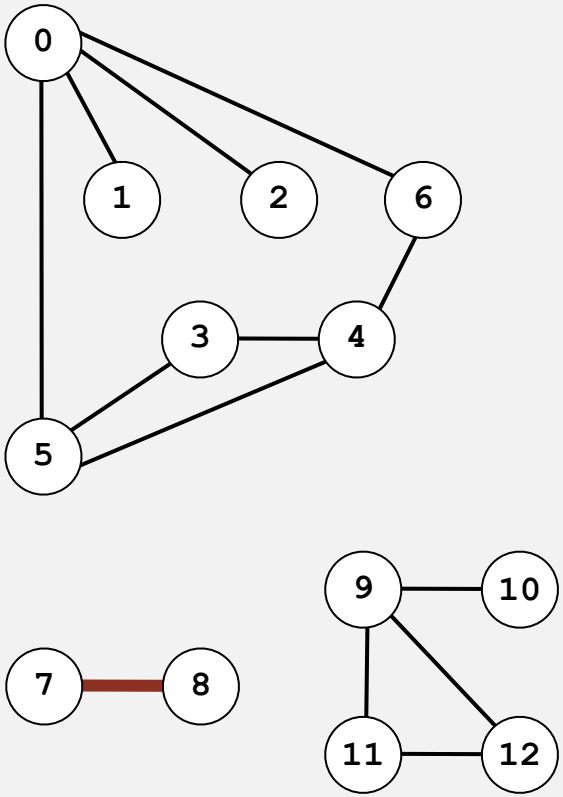
two entries
for each edge

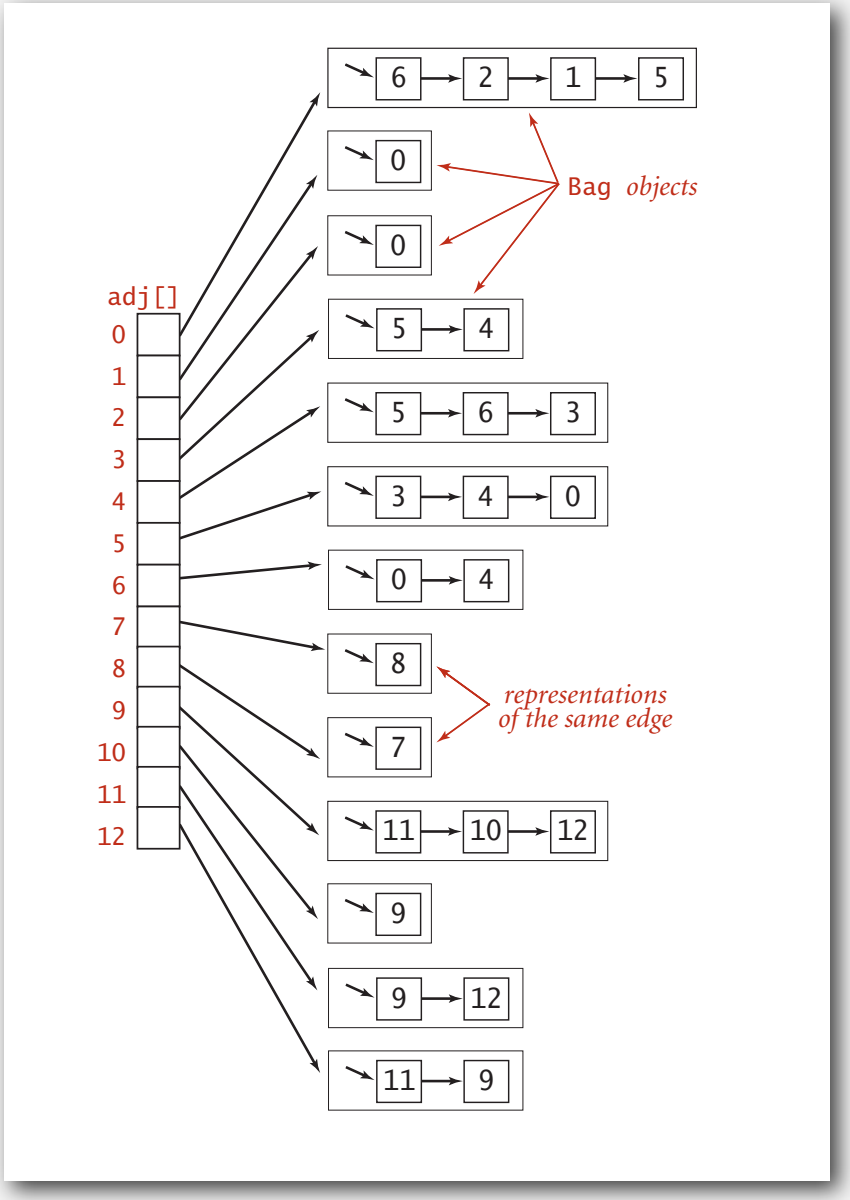
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0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

Adjacency-list graph representation

Maintain vertex-indexed array of lists.

(use `Bag` abstraction)





Adjacency-list graph representation: Java implementation

```
public class Graph
{
   private final int V;
   private 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
(using **Bag** data type)

create empty graph
with **V** vertices

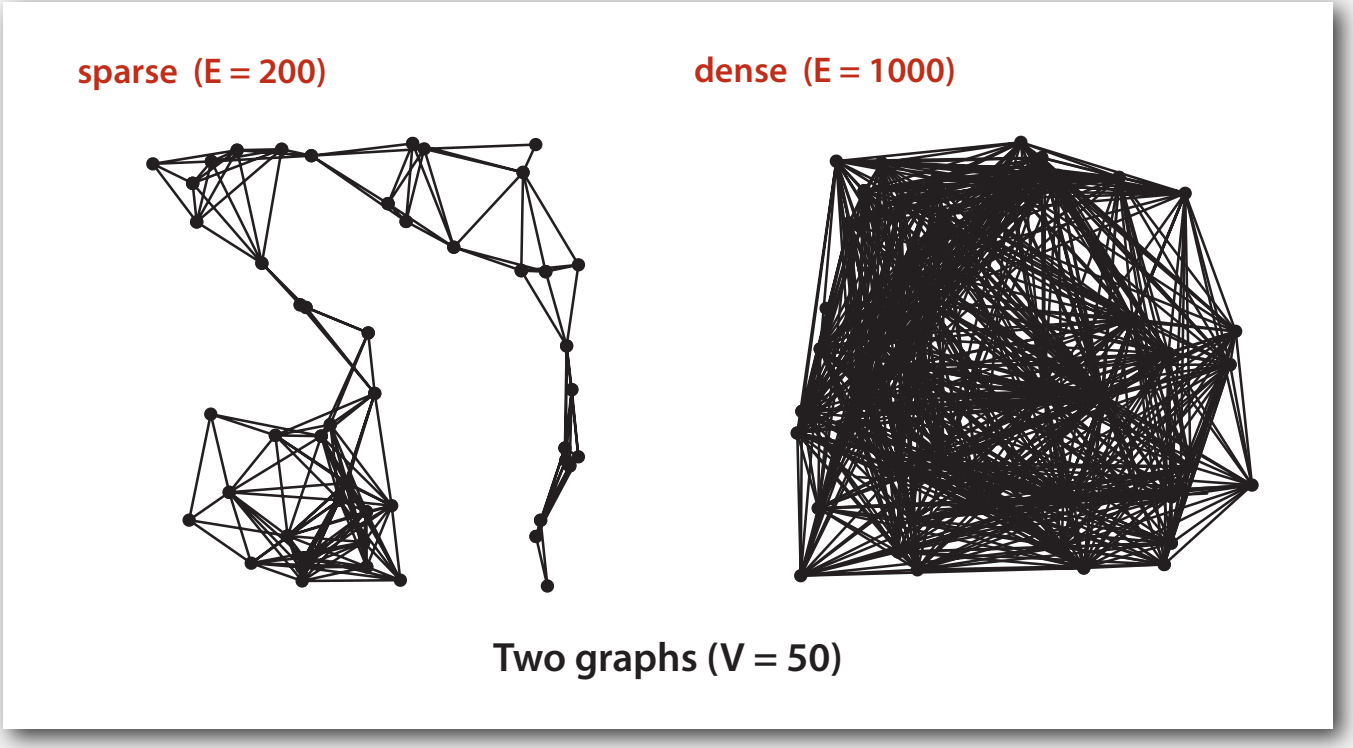
add edge **v-w**
(parallel edges allowed)

iterator for vertices adjacent to **v**

In practice. Use adjacency-lists representation.

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

huge number of vertices,
small average vertex degree



Graph representations

In practice. Use adjacency-lists representation.

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

huge number of vertices,
small average vertex degree

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	E + V	1	degree(v)	degree(v)

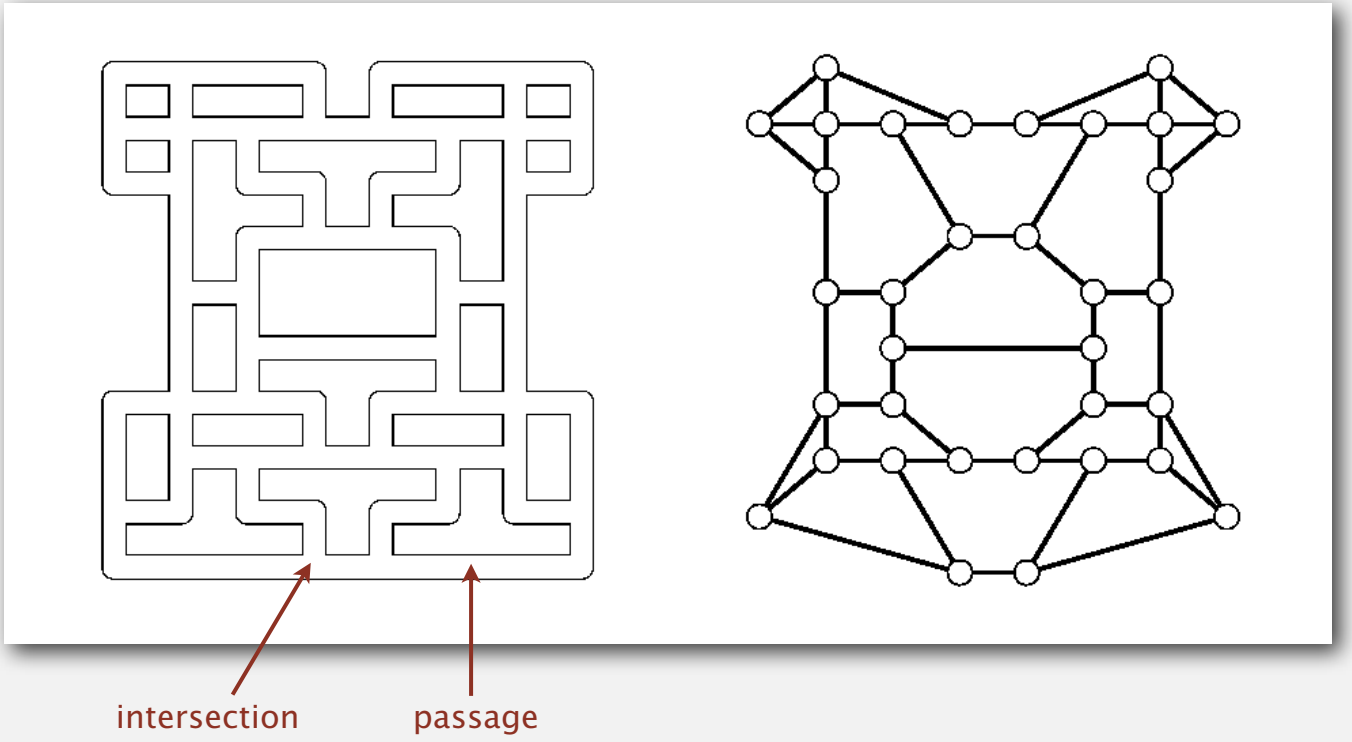
* disallows parallel edges

- graph API
- **depth-first search**
- breadth-first search
- connected components
- challenges

Maze exploration

Maze graphs.

- Vertex = intersection.
- Edge = passage.

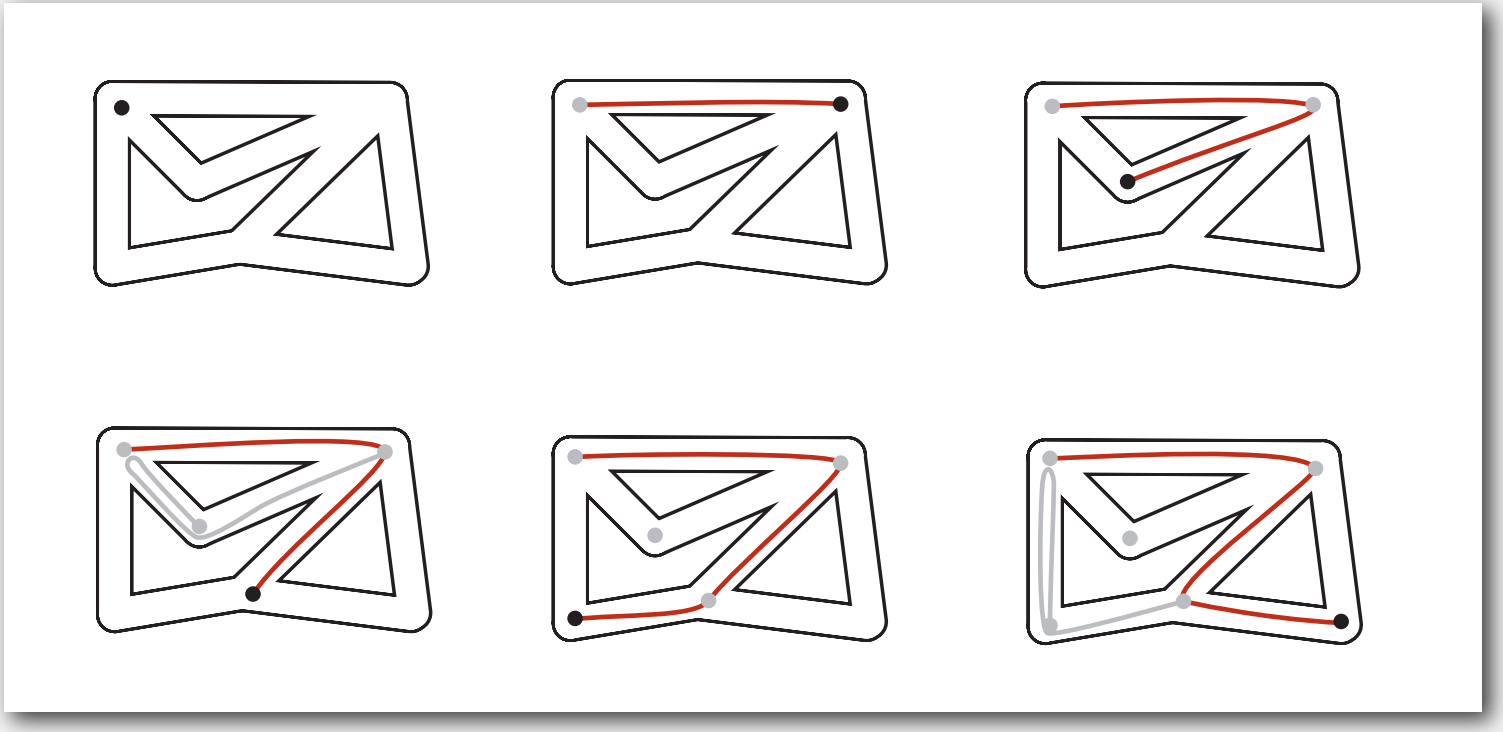


Goal. Explore every intersection in the maze.

Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.

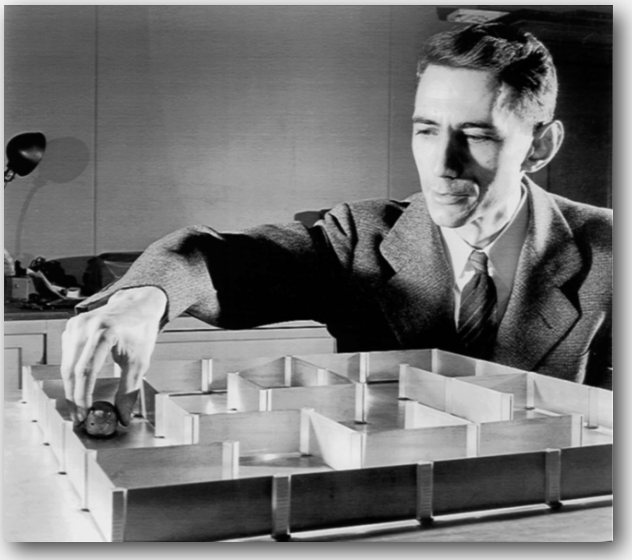


Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.

First use? Theseus entered labyrinth to kill the monstrous Minotaur; Ariadne held ball of string.



Claude Shannon (with Theseus mouse)

Depth-first search

Goal. Systematically search through a graph.

Idea. Mimic maze exploration.

DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked vertices w adjacent to v.

Typical applications.

- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Design pattern for graph processing

Design pattern. Decouple graph data type from graph processing.

public class Search	
Search(Graph G, int s)	*find vertices connected to s*
boolean marked(int v)	*is vertex v connected to s?*
int count()	*how many vertices connected to s?*

Typical client program.

- Create a `Graph`.
- Pass the `Graph` to a graph-processing routine, e.g., `Search`.
- Query the graph-processing routine for information.

```
Search search = new Search(G, s);
for (int v = 0; v < G.V(); v++)
   if (search.marked(v))
      StdOut.println(v);
```

print all vertices connected to s

Depth-first search (warmup)

Goal. Find all vertices connected to s.

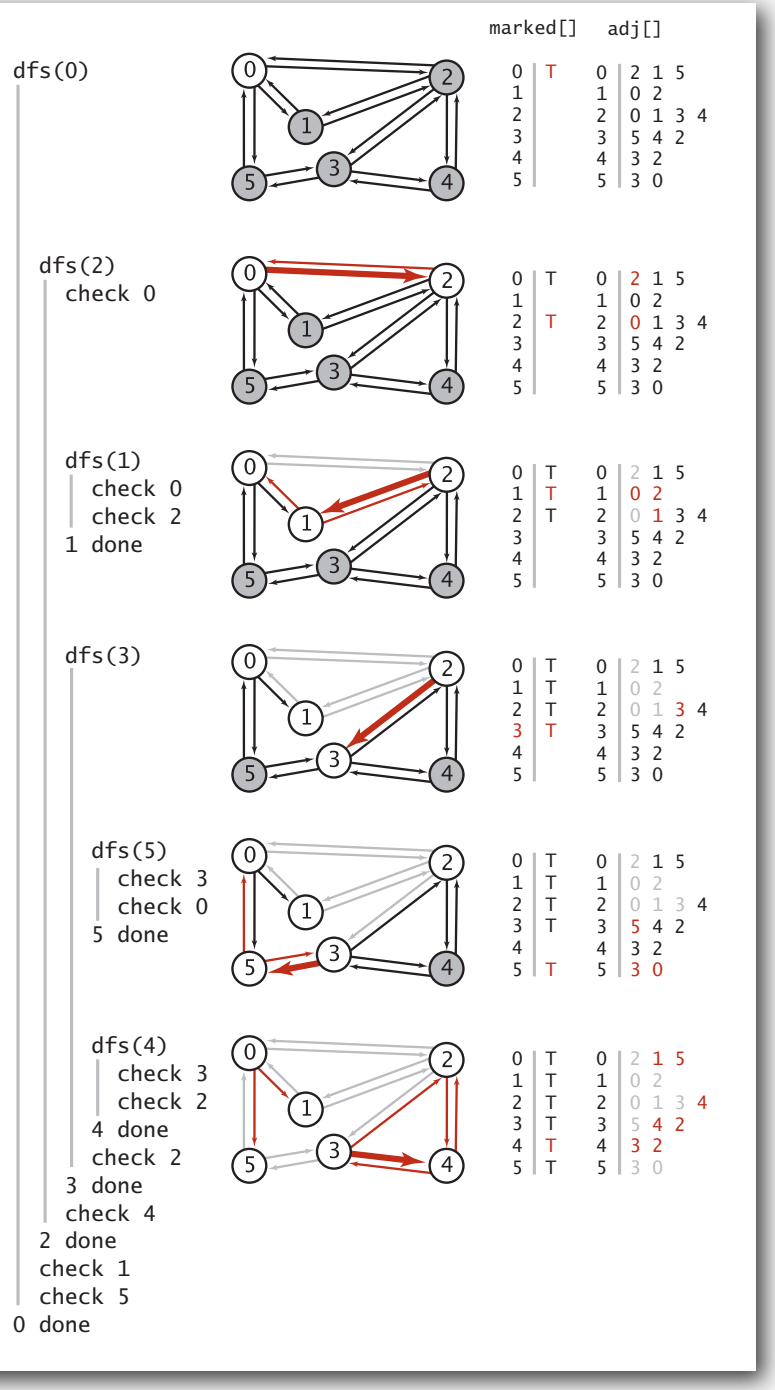
Idea. Mimic maze exploration.

Algorithm.

- Use recursion (ball of string).
- Mark each visited vertex.
- Return (retrace steps) when no unvisited options.

Data structure.

- `boolean[] marked` to mark visited vertices.



Depth-first search (warmup)

```
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 marked(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 vertex v is connected to s

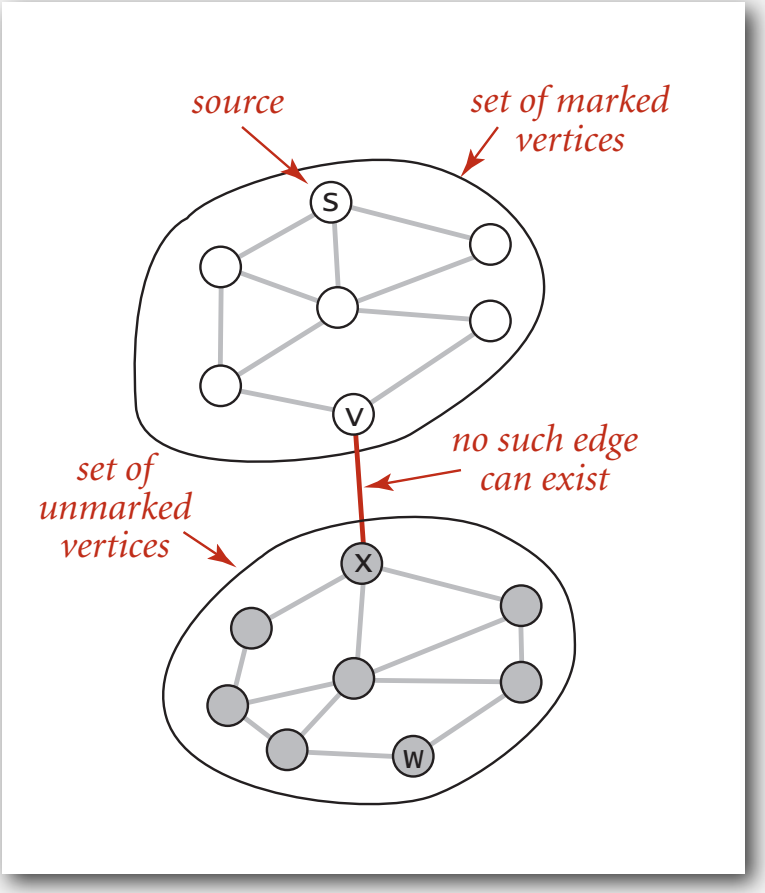
Depth-first search properties

Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees.

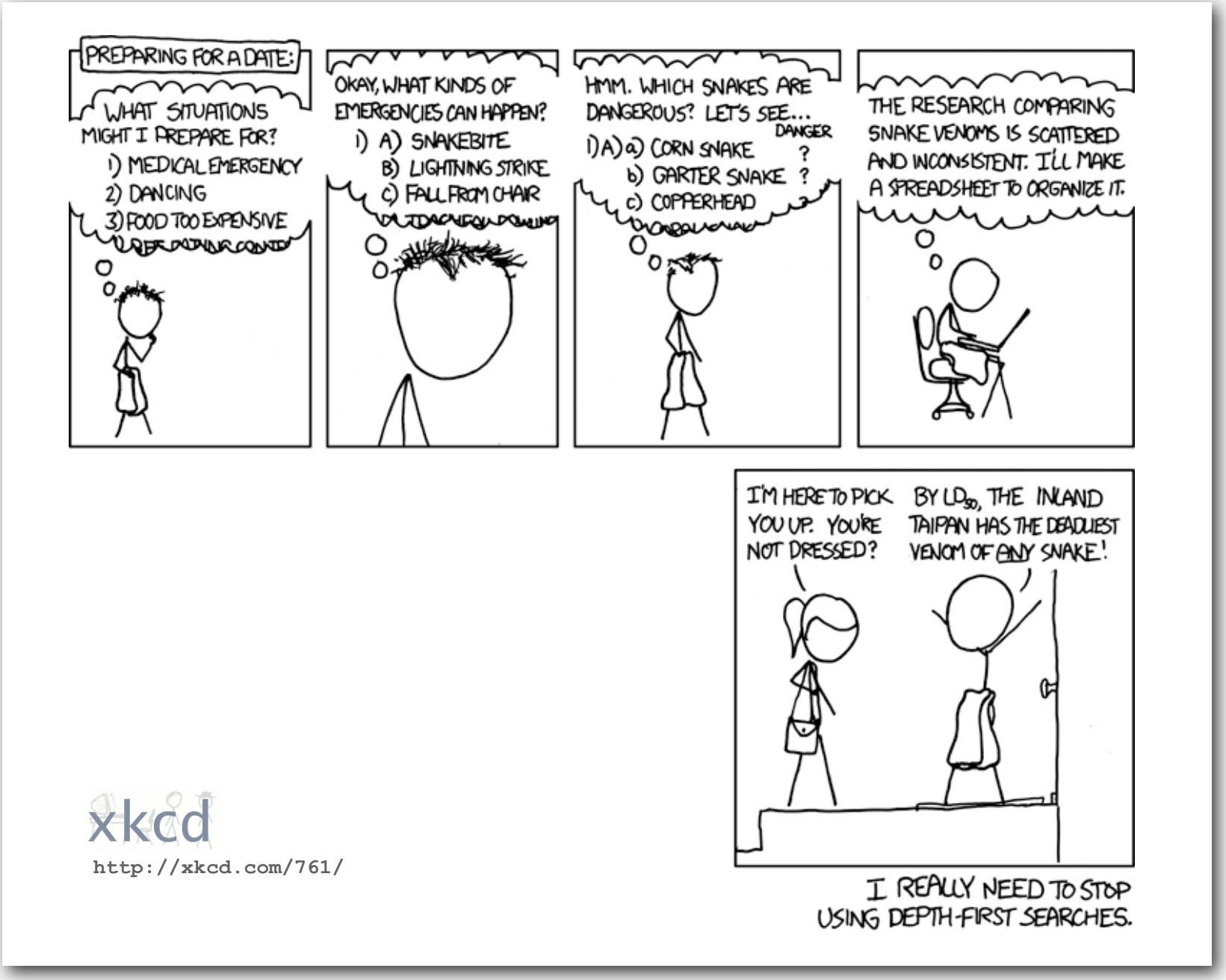
Pf.

- Correctness:
  - if w marked, then w connected to s (why?)
  - if w connected to s, then w marked
    (if w unmarked, then consider last edge on a path from s to w that goes from a marked vertex to an unmarked one)

- Running time: each vertex connected to s is visited once.



Depth-first search application: preparing for a date



xkcd

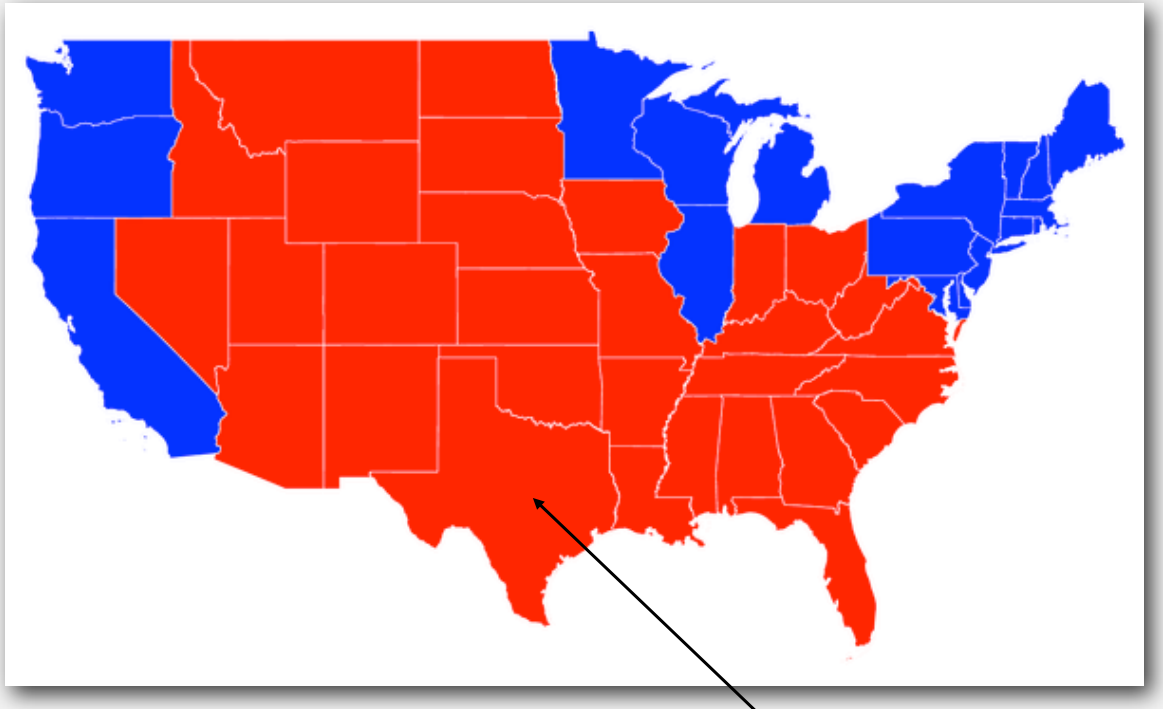
http://xkcd.com/761/

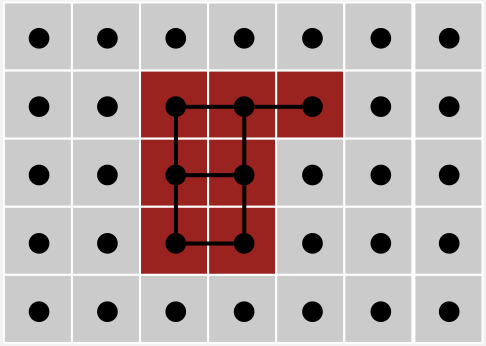
Depth-first search application: flood fill

Change color of entire blob of neighboring red pixels to blue.

Build a grid graph.

- Vertex: pixel.
- Edge: between two adjacent red pixels.
- Blob: all pixels connected to given pixel.



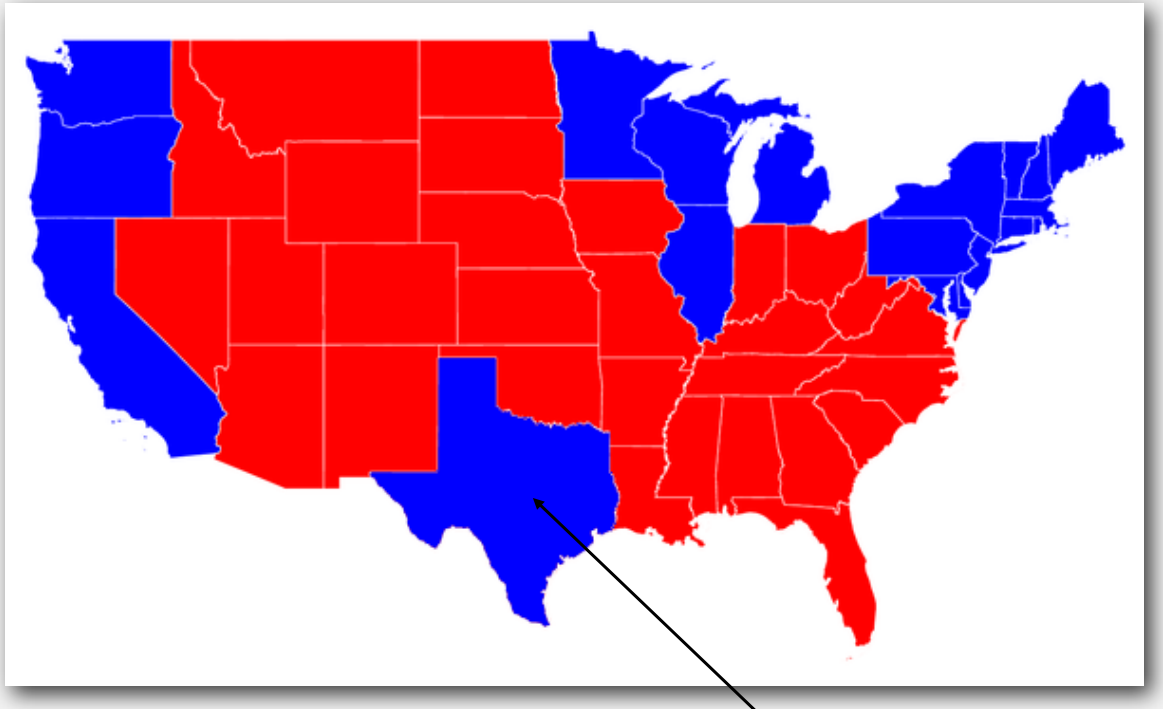


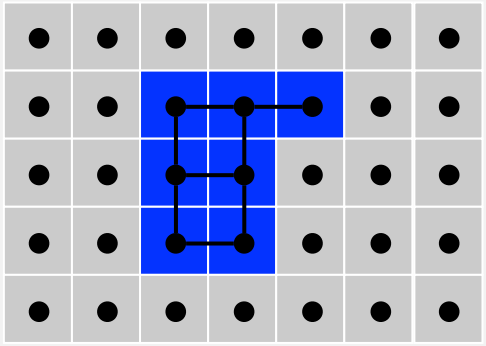
Depth-first search application: flood fill

Change color of entire blob of neighboring red pixels to blue.

Build a grid graph.

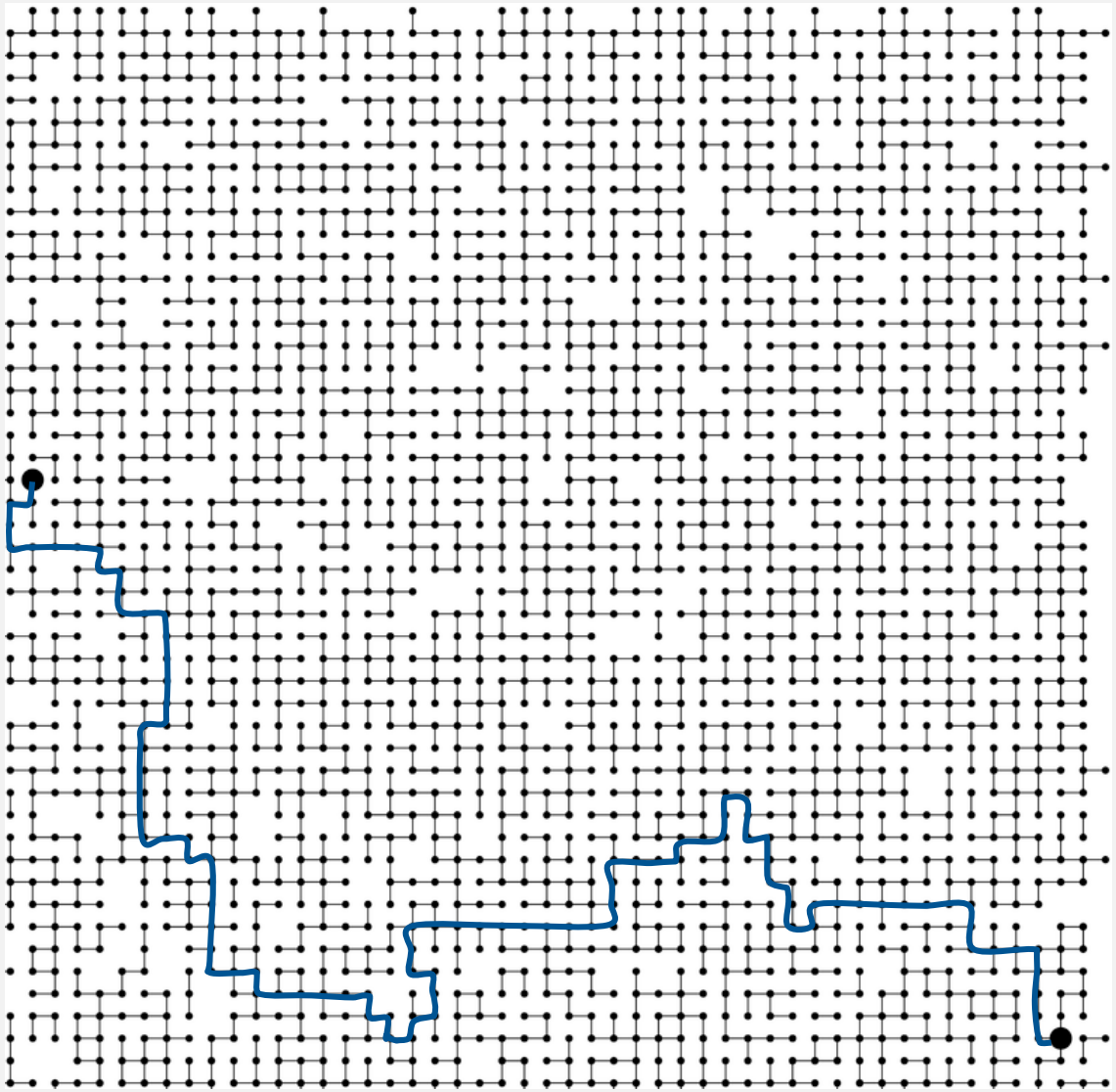
- Vertex: pixel.
- Edge: between two adjacent red pixels.
- Blob: all pixels connected to given pixel.





Pathfinding in graphs

Goal. Does there exist a path from s to t? If yes, find any such path.



Pathfinding in graphs

Goal. Does there exist a path from s to t? If yes, find any such path.

public class Paths		
	`Paths(Graph G, int s)`	*find paths in G from source s*
`boolean`	`hasPathTo(int v)`	*is there a path from s to v?*
`Iterable<Integer>`	`pathTo(int v)`	*path from s to v; null if no such path*

Union-find. Not much help.

Depth-first search. After linear-time preprocessing, can recover path itself in time proportional to its length.

easy modification
(stay tuned)

Depth-first search (pathfinding)

Goal. Find paths to all vertices connected to a given source s.

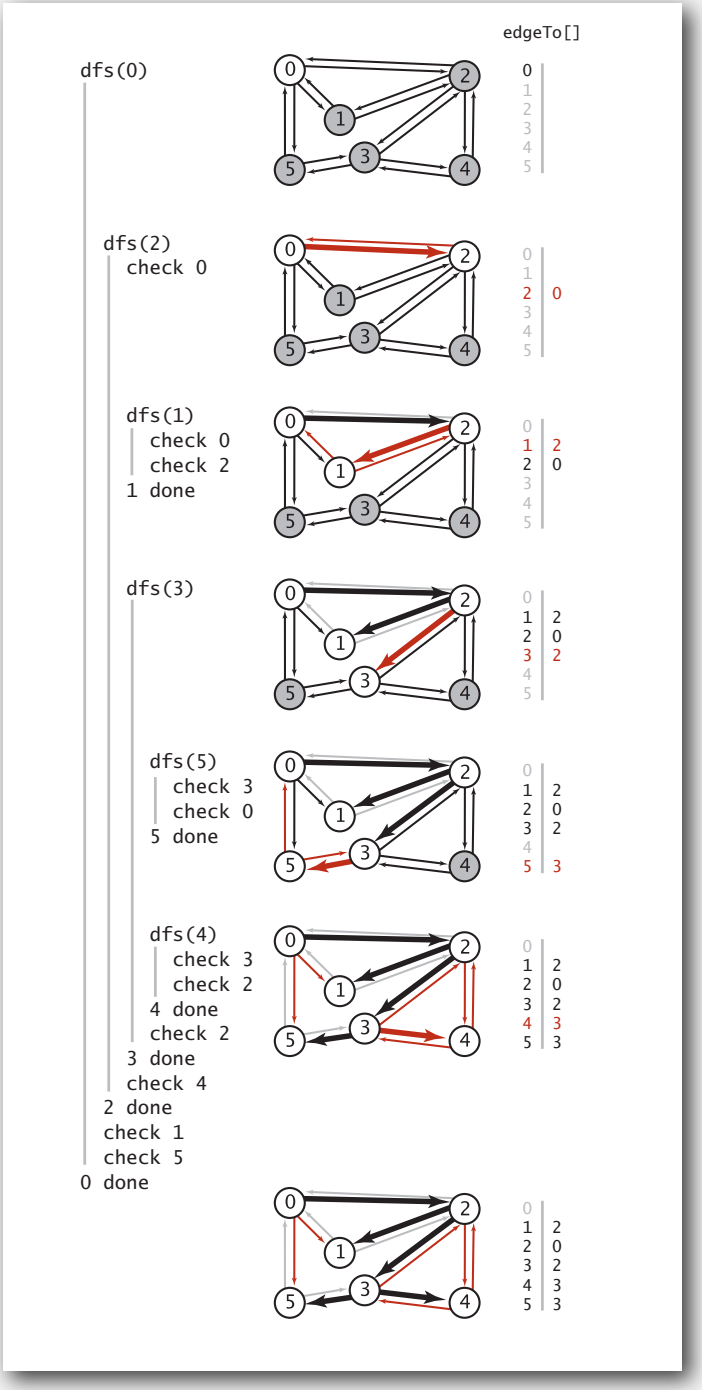
Idea. Mimic maze exploration.

Algorithm.

- Use recursion (ball of string).
- Mark each visited vertex by keeping
- track of edge taken to visit it.
- Return (retrace steps) when no unvisited options.

Data structures.

- `boolean[] marked` to mark visited vertices.
- `int[] edgeTo` to keep tree of paths.
- `(edgeTo[w] == v)` means that edge `v-w` was taken to visit `w` the first time



Depth-first search (pathfinding)

```
public class DepthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private final int s;

   public DepthFirstPaths(Graph G, int s)
   {
      marked = new boolean[G.V()];
      edgeTo = new int[G.V()];
      this.s = s;
      dfs(G, s);
   }
   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w])
         {
            edgeTo[w] = v;
            dfs(G, w);
         }
   }

   public boolean hasPathTo(int v)
   public Iterable<Integer> pathTo(int v)
}
```

parent-link representation of DFS tree

set parent link

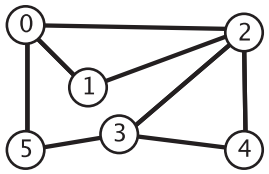
ahead

Depth-first search (pathfinding trace)

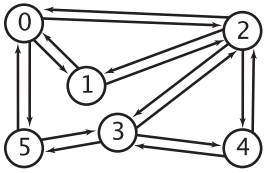
tinyCG.txt

V → 6
E → 8
0 5
2 4
2 3
1 2
0 1
3 4
3 5
0 2

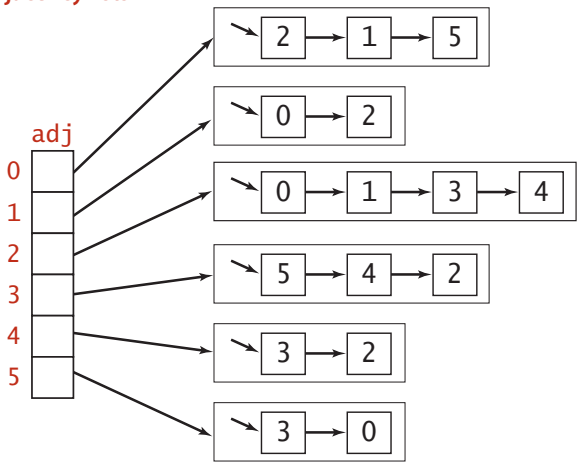
standard drawing

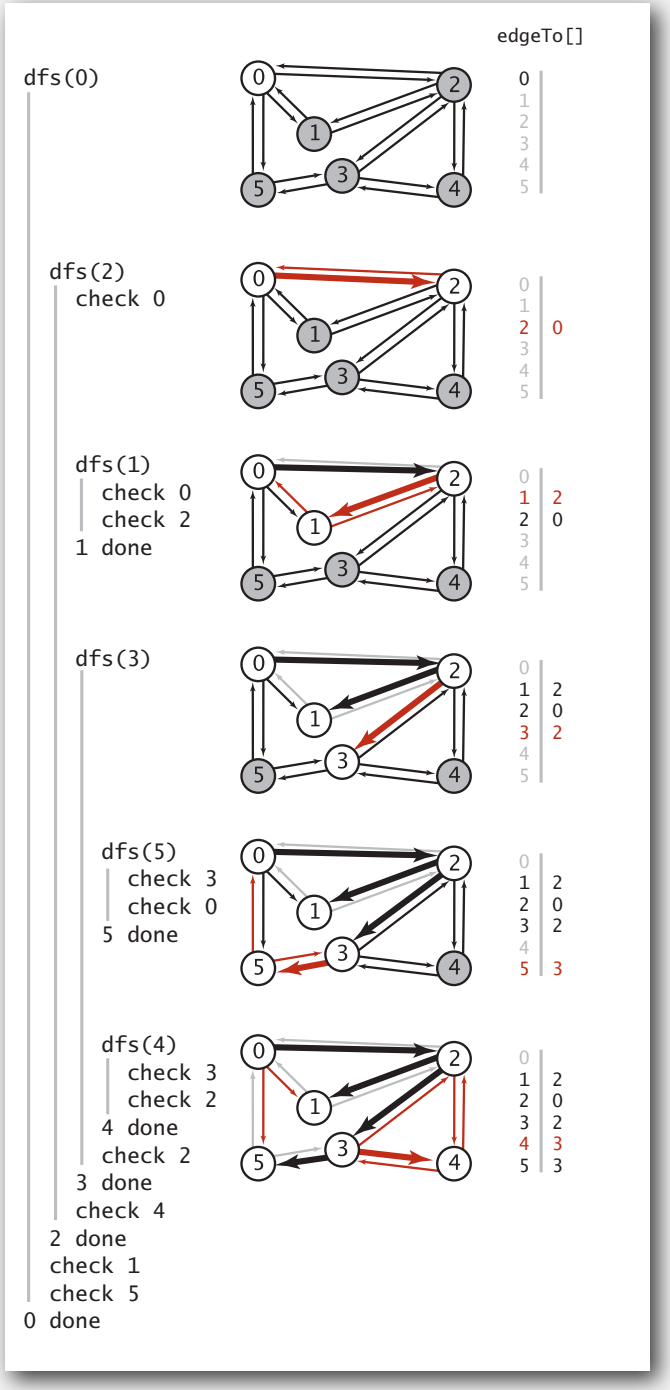


drawing with both edges



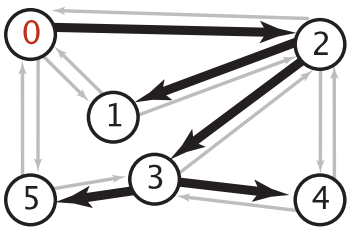
adjacency lists





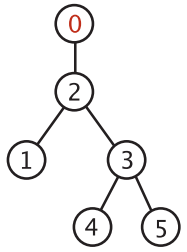
Depth-first search (pathfinding iterator)

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



edgeTo[]

0	
1	2
2	0
3	2
4	3
5	3



x	path
5	5
3	3 5
2	2 3 5
0	0 2 3 5

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

public Iterable<Integer> pathTo(int v)
{
   if (!hasPathTo(v)) return null;
   Stack<Integer> path = new Stack<Integer>();
   for (int x = v; x != s; x = edgeTo[x])
      path.push(x);
   path.push(s);
   return path;
}
```

- graph API
- depth-first search
- **breadth-first search**
- connected components
- challenges

Breadth-first search

Depth-first search. Put unvisited vertices on a stack.

Breadth-first search. Put unvisited vertices on a queue.

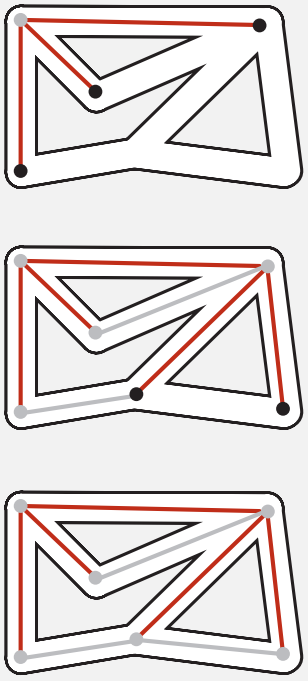
Shortest path. Find path from s to t that uses fewest number of edges.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

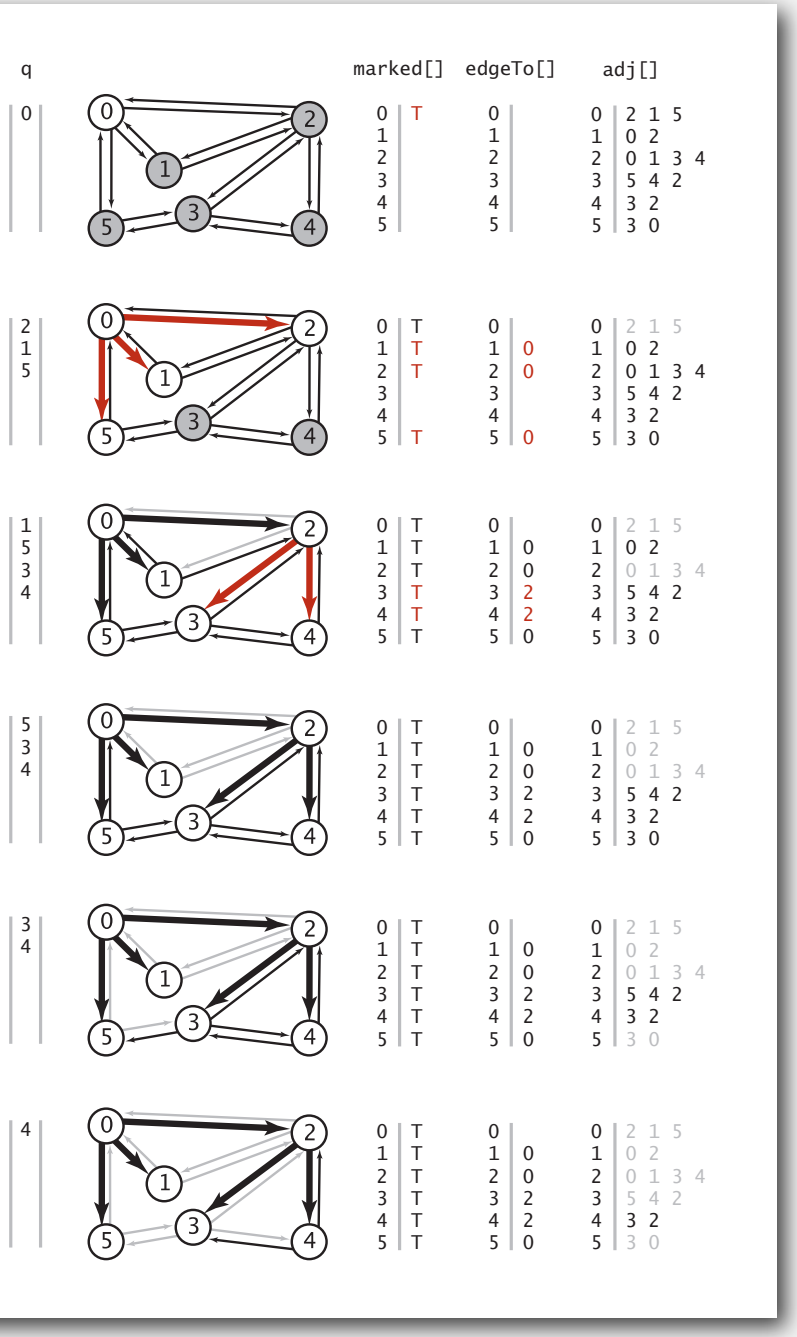
- **remove the least recently added vertex v**
- **add each of v's unvisited neighbors to the queue, and mark them as visited.**



Intuition. BFS examines vertices in increasing distance from s.

Breadth-first search (pathfinding)

```
private void bfs(Graph G, int s)
{
   Queue<Integer> q = new Queue<Integer>();
   q.enqueue(s);
   marked[s] = true;
   while (!q.isEmpty())
   {
      int v = q.dequeue();
      for (int w : G.adj(v))
         if (!marked[w])
         {
            q.enqueue(w);
            marked[w] = true;
            edgeTo[w] = v;
         }
   }
}
```



Breadth-first search

```
public class BreadthFirstPaths

{
   private boolean[] marked; // Is a shortest path to this vertex known?
   private int[] edgeTo;     // last vertex on known path to this vertex
   private final int s;      // source

   public BreadthFirstPaths(Graph G, int s)
   {
      marked = new boolean[G.V()];
      edgeTo = new int[G.V()];
      this.s = s;
      bfs(G, s);
   }

   private void bfs(Graph G, int s)
   {
      Queue<Integer> queue = new Queue<Integer>();
      marked[s] = true;           // Mark the source
      queue.enqueue(s);           //   and put it on the queue.
      while (!q.isEmpty())
      {
         int v = queue.dequeue(); // Remove next vertex from the queue.
         for (int w : G.adj(v))
            if (!marked[w])       // For every unmarked adjacent vertex,
            {
               edgeTo[w] = v;     //   save last edge on a shortest path,
               marked[w] = true;  //   mark it because path is known,
               queue.enqueue(w);  //   and add it to the queue.
            }
      }
   }
```

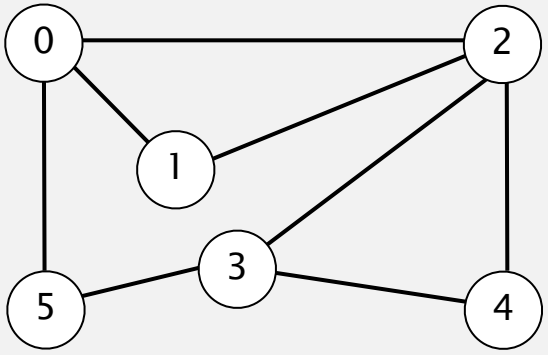
Breadth-first search properties

Proposition. BFS computes shortest path (number of edges) from s in a connected graph in time proportional to $E + V$.

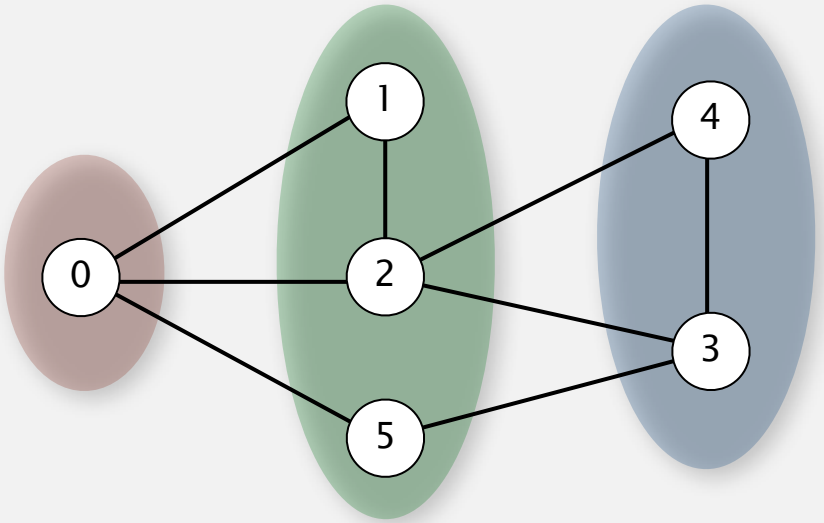
Pf.

- Correctness: queue always consists of zero or more vertices of distance k from s, followed by zero or more vertices of distance $k + 1$.

- Running time: each vertex connected to s is visited once.

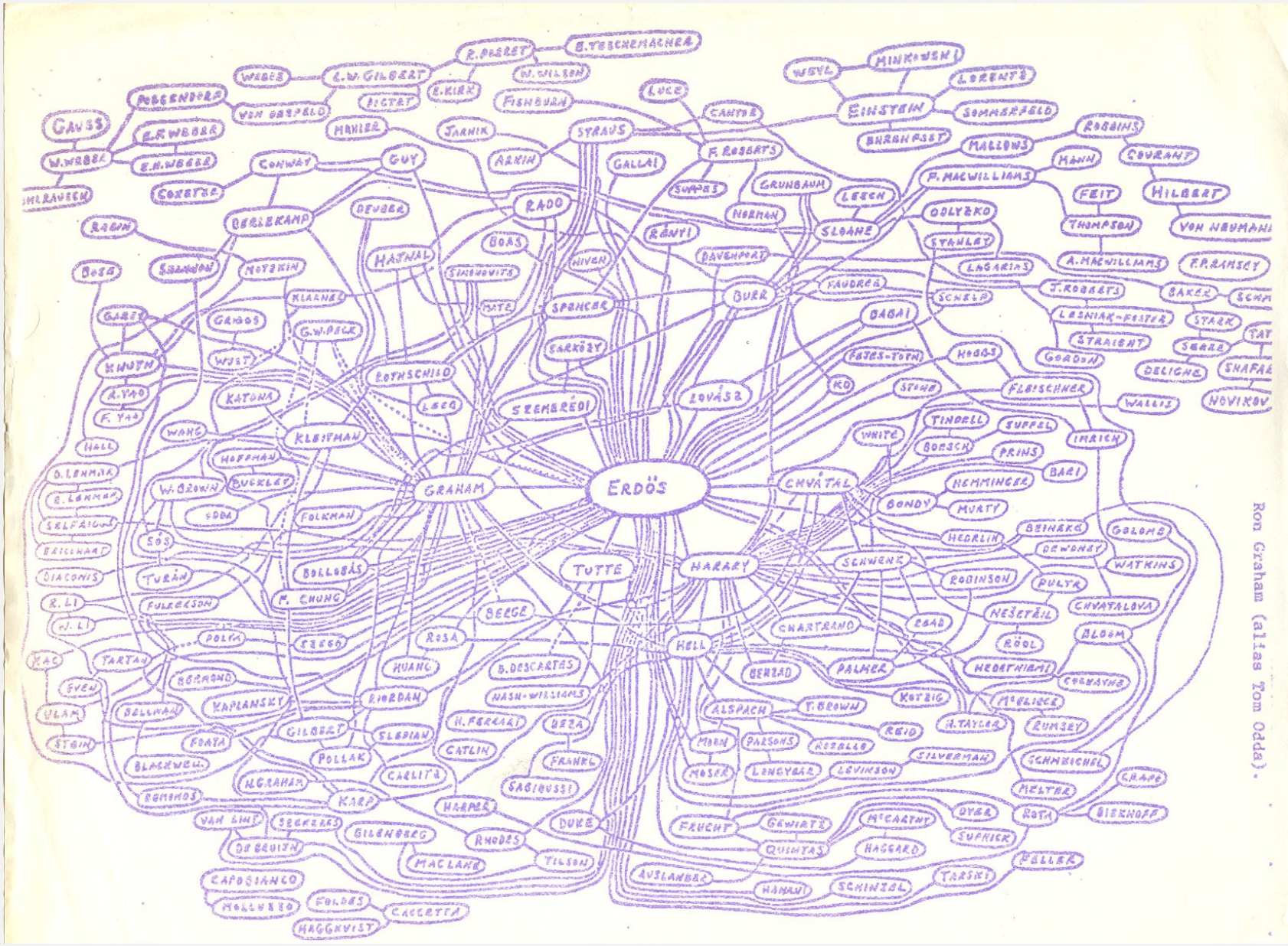


standard drawing



dist = 0 dist = 1 dist = 2

Breadth-first search application: Erdös numbers



hand-drawing of part of the Erdös graph by Ron Graham

- graph API
- depth-first search
- breadth-first search
- **connected components**
- challenges

Connectivity queries

Def. Vertices v and w are connected if there is a path between them.

Goal. Preprocess graph to answer queries: is v connected to w ?
in constant time.

public class CC	
CC(Graph G)	*find connected components in G*
boolean connected(int v, int w)	*are v and w connected?*
int count()	*number of connected components*
int id(int v)	*component identifier for v*

Union-Find? Not quite.

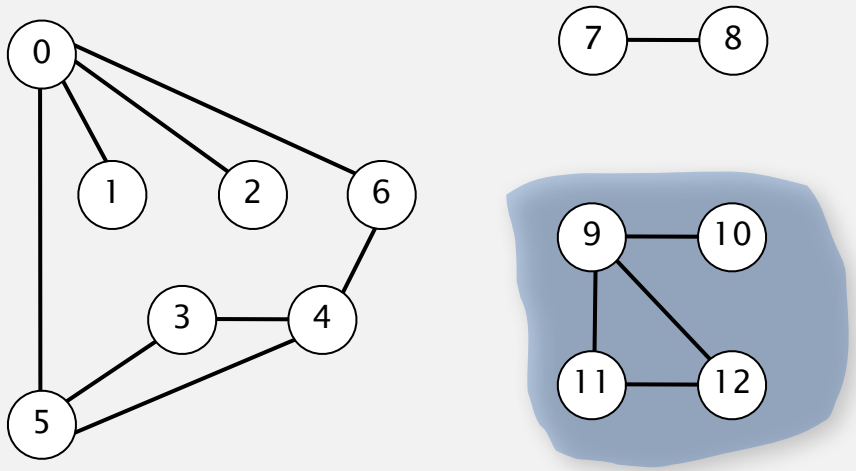
Depth-first search. Yes. [next few slides]

Connected components

The relation "is connected to" is an equivalence relation:

- Reflexive: v is connected to v.
- Symmetric: if v is connected to w, then w is connected to v.
- Transitive: if v connected to w and w connected to x, then v connected to x.

Def. A connected component is a maximal set of connected vertices.



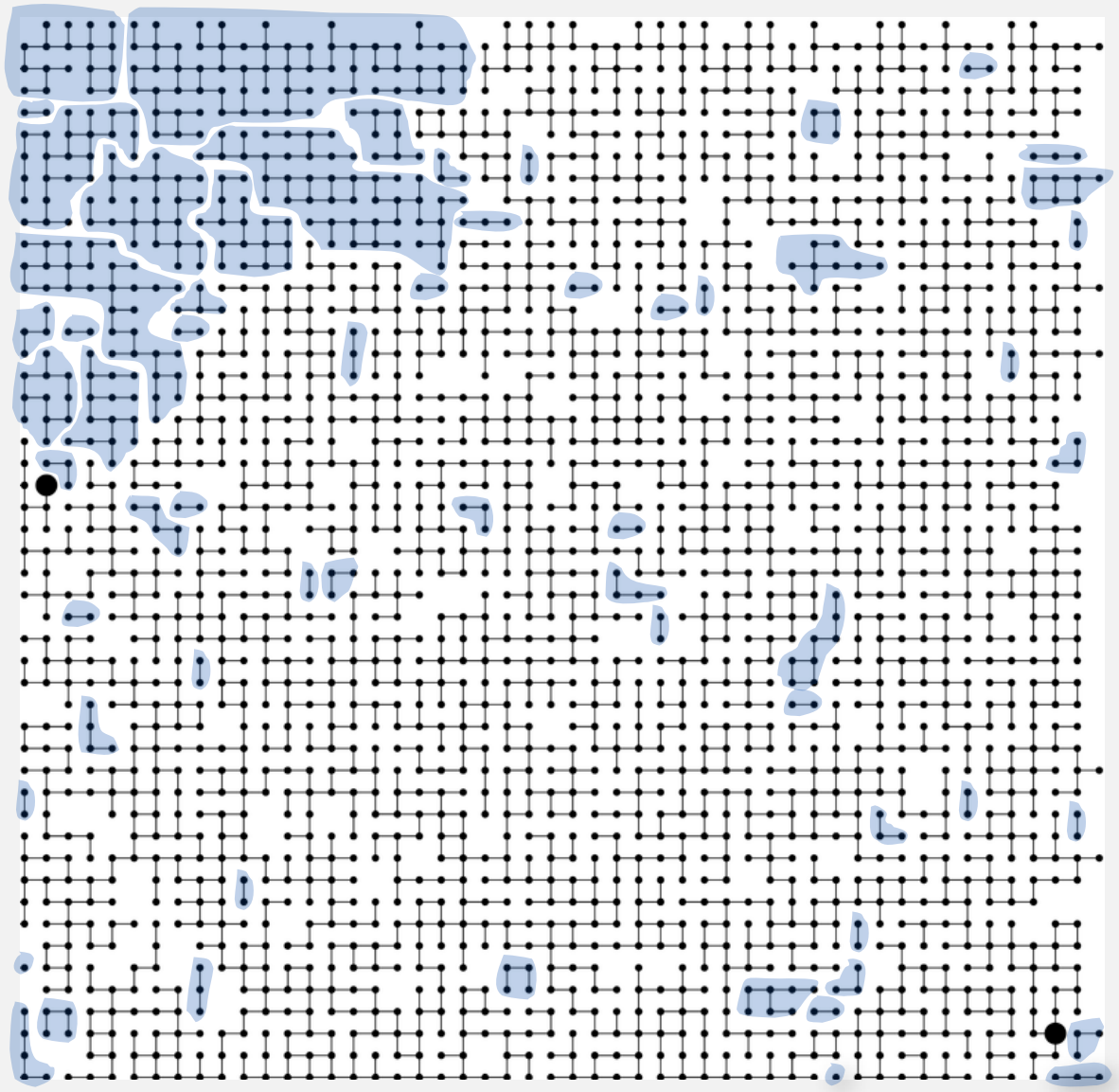
3 connected components

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

Remark. Given connected components, can answer queries in constant time.

Connected components

Def. A connected component is a maximal set of connected vertices.



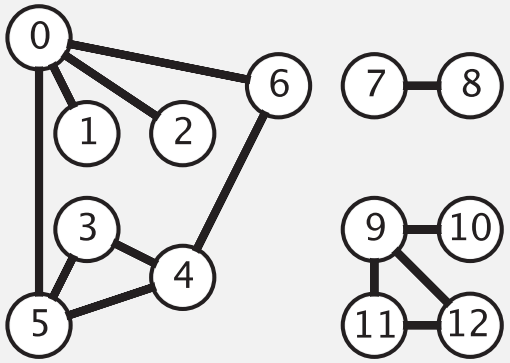
63 connected components

Goal. Partition vertices into connected components.

Connected components

Initialize all vertices v as unmarked.

For each unmarked vertex v, run DFS to identify all vertices discovered as part of the same component.



tinyG.txt

```
13
13
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

V E

Finding connected components 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++;
         }
      }
   }

   public int count()
   public int id(int v)
   private void dfs(Graph G, int v)

}
```

id[v] = id of component containing v

number of components

run DFS from one vertex in each component

see next slide

Finding connected components with DFS (continued)

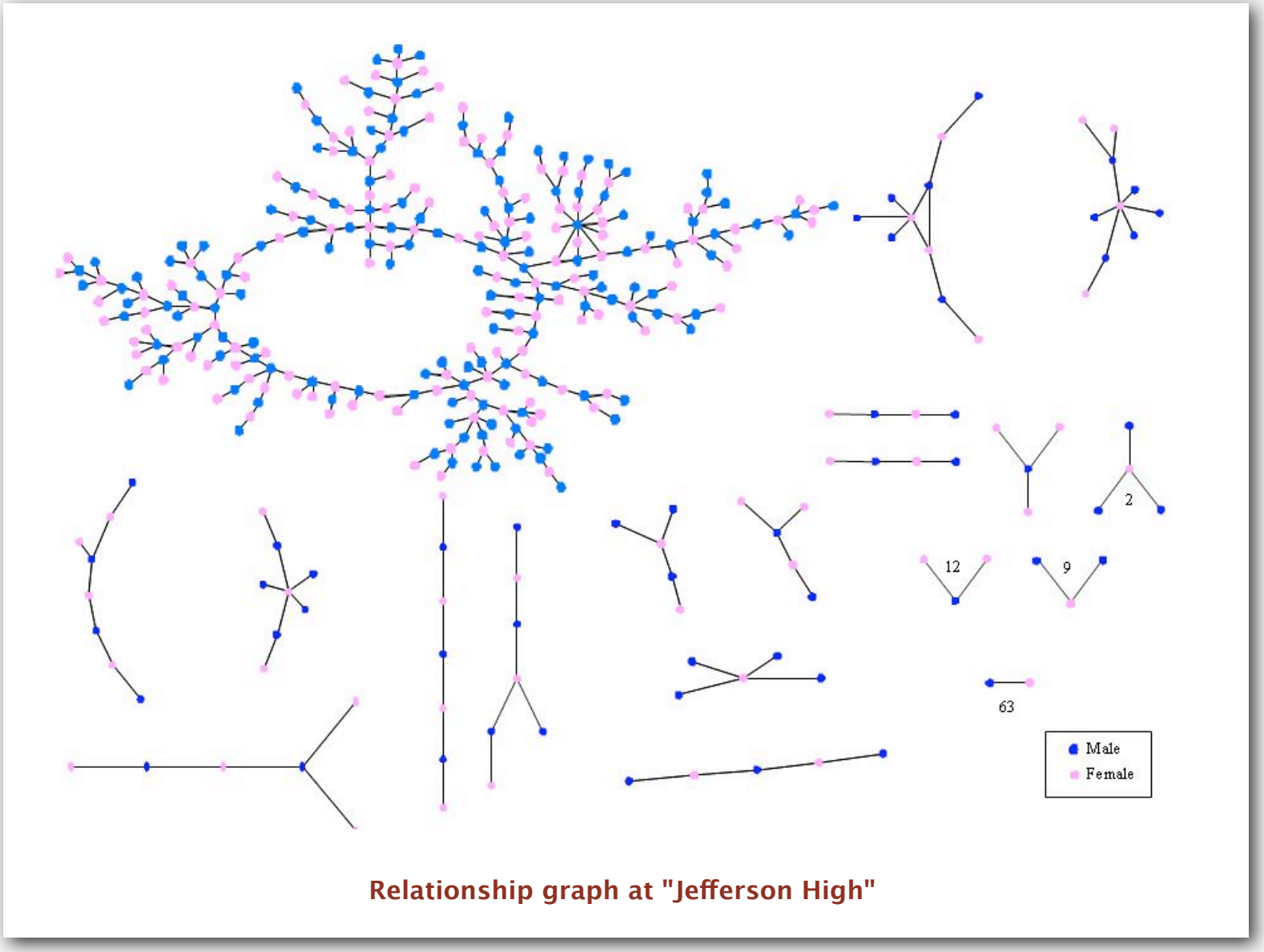
```
public int count()
{  return count;  }

public int id(int v)
{  return id[v];  }

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);
}
```

number of components

id of component containing v

all vertices discovered in same call of dfs have same id

Connected components application: study spread of STDs



Relationship graph at "Jefferson High"

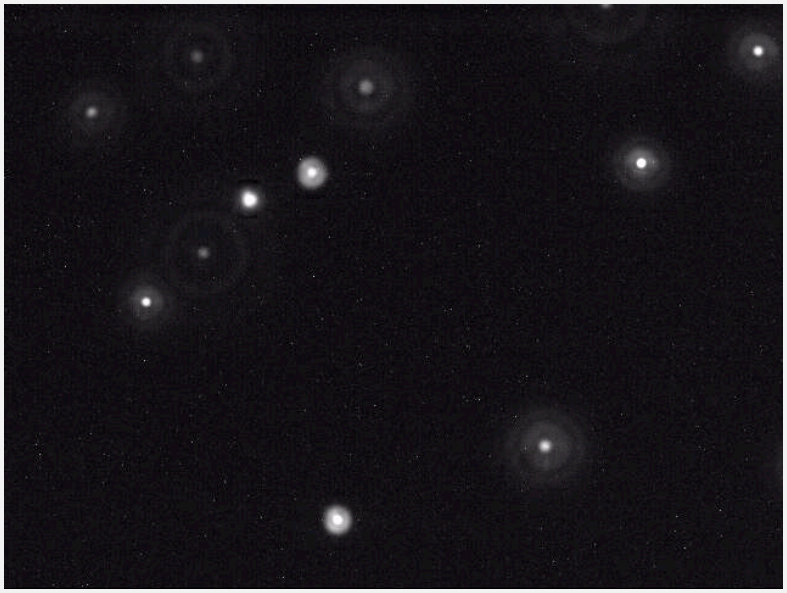
Peter Bearman, James Moody, and Katherine Stovel. Chains of affection: The structure of adolescent romantic and sexual networks. American Journal of Sociology, 110(1): 44–99, 2004.

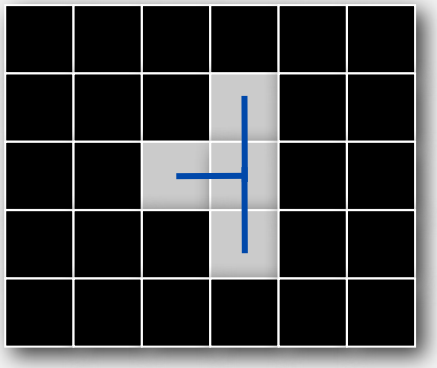
Connected components application: particle detection

Particle detection. Given grayscale image of particles, identify "blobs."

- Vertex: pixel.
- Edge: between two adjacent pixels with grayscale value ≥ 70.
- Blob: connected component of 20-30 pixels.

black = 0
white = 255



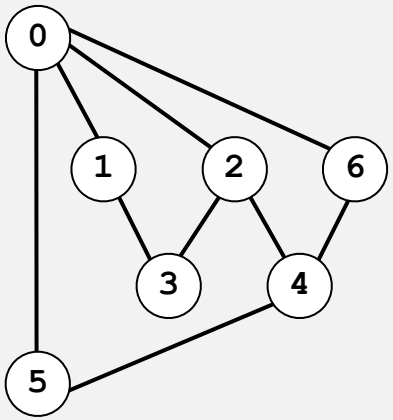


Particle tracking. Track moving particles over time.

- graph API
- depth-first search
- breadth-first search
- connected components
- **challenges**

Graph-processing challenge 1

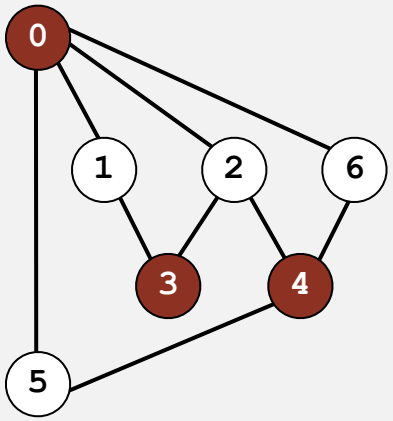
Problem. Is a graph bipartite?



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

How difficult?

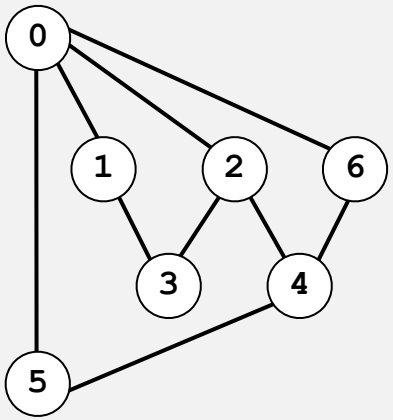
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



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

Graph-processing challenge 2

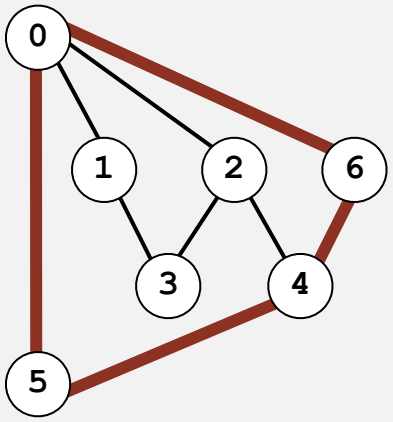
Problem. Find a cycle.



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

How difficult?

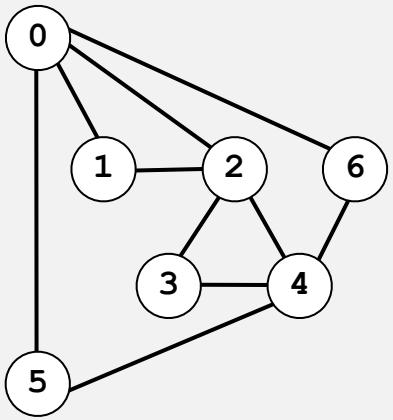
- Any COS 126 student could do it.
- Need to be a typical diligent COS 226 student.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



Graph-processing challenge 3

Problem. Find a cycle that uses every edge.

Assumption. Need to use each edge exactly once.



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

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

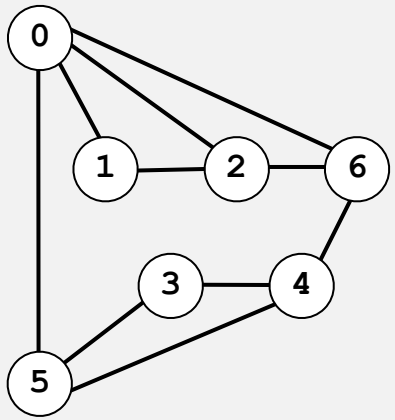
How difficult?

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

Graph-processing challenge 4

Problem. Find a cycle that visits every vertex.

Assumption. Need to visit each vertex exactly once.



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

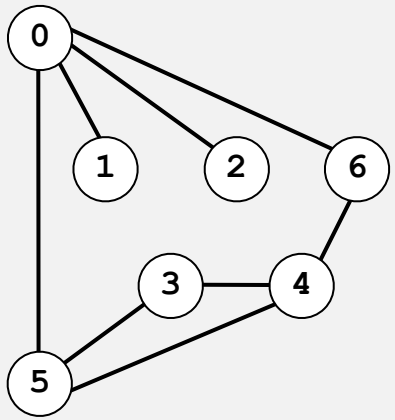
0-5-3-4-6-2-1-0

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.

Graph-processing challenge 5

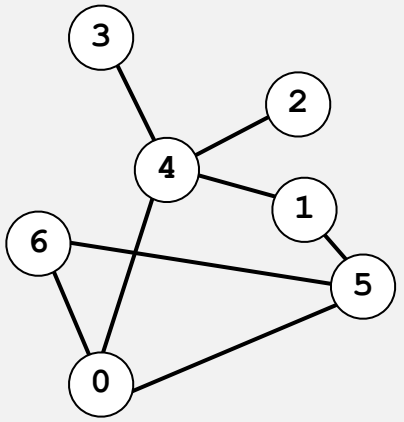
Problem. Are two graphs identical except for vertex names?



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

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.

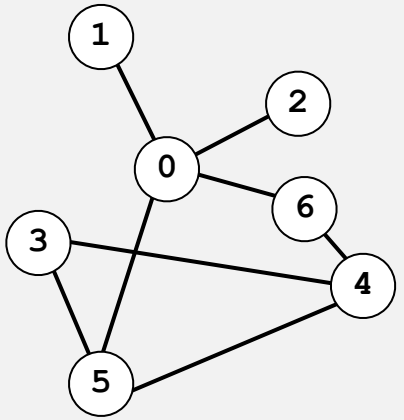


0-4
0-5
0-6
1-4
1-5
2-4
3-4
5-6

0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

Graph-processing challenge 6

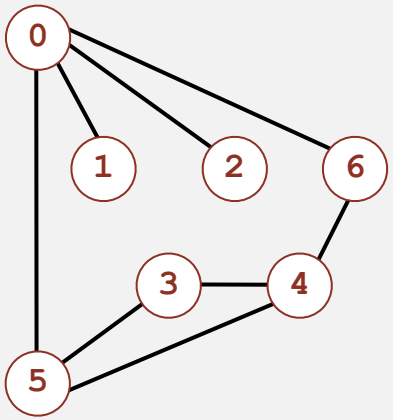
Problem. Lay out a graph in the plane without crossing edges?



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

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.

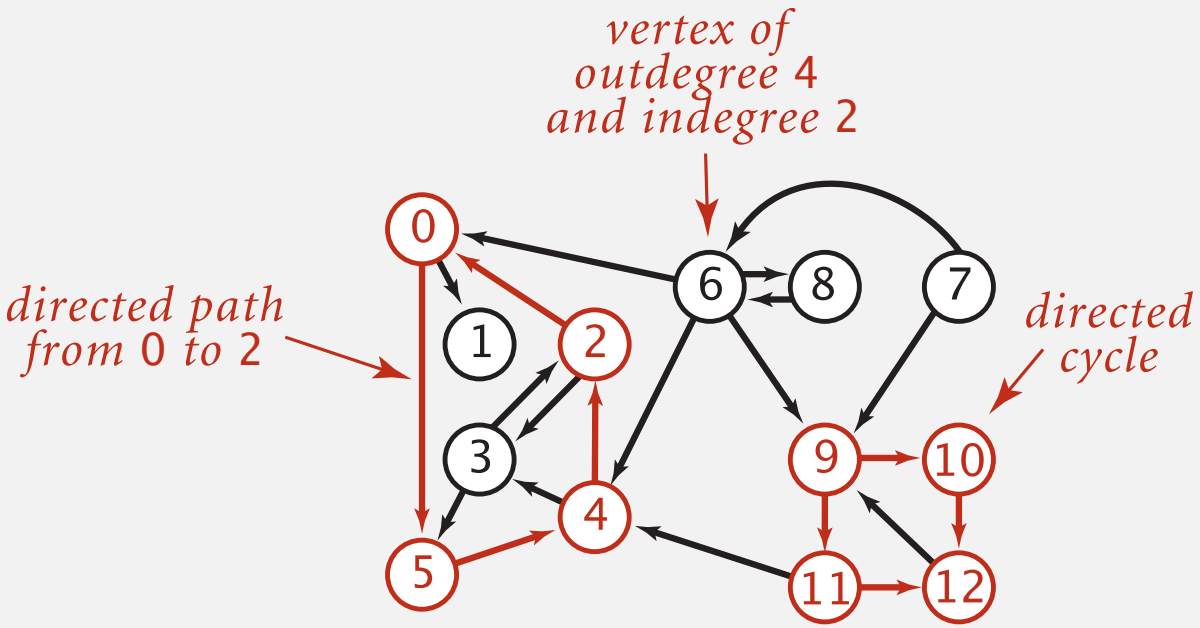


4.2 Directed Graphs

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

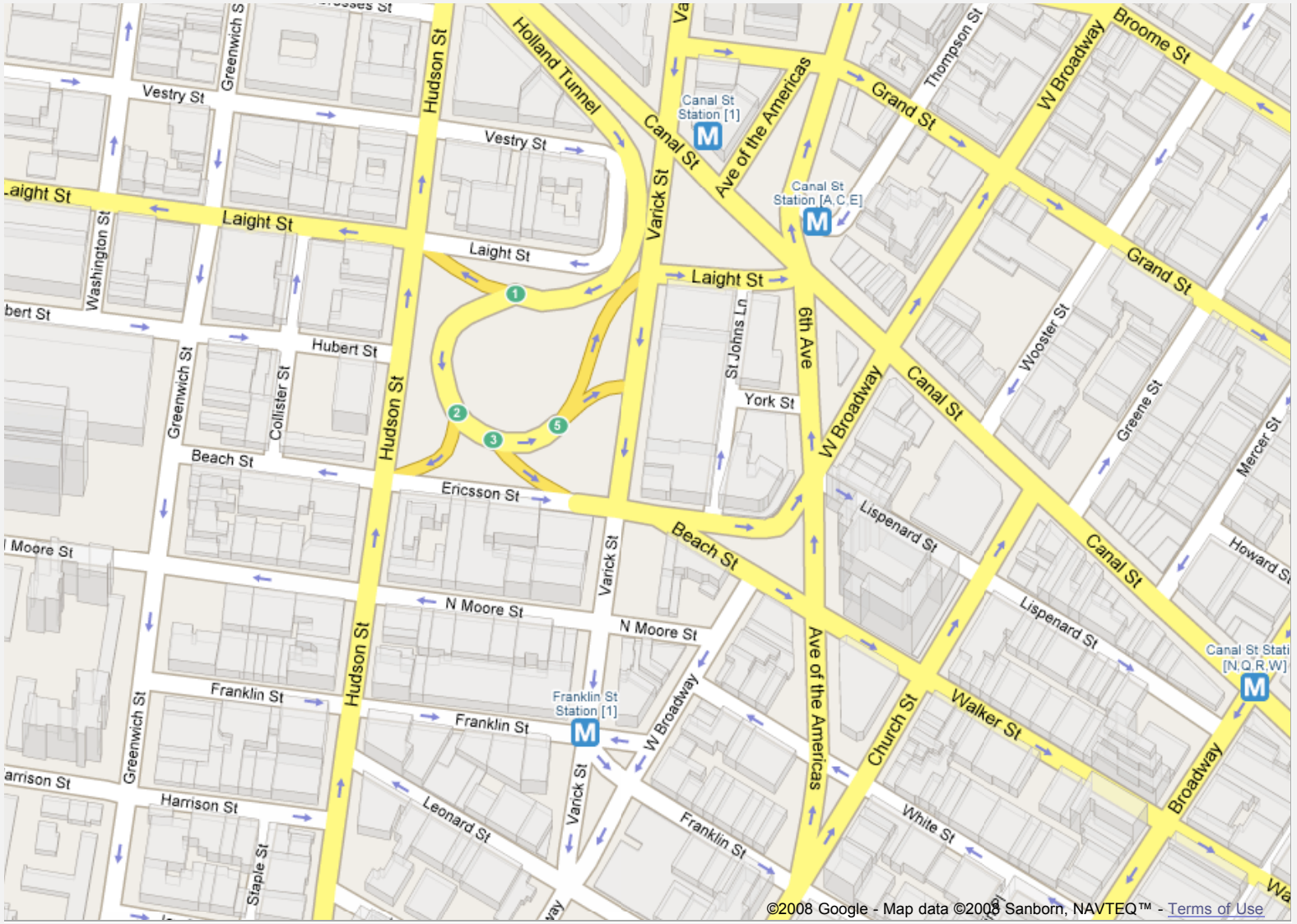
Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.



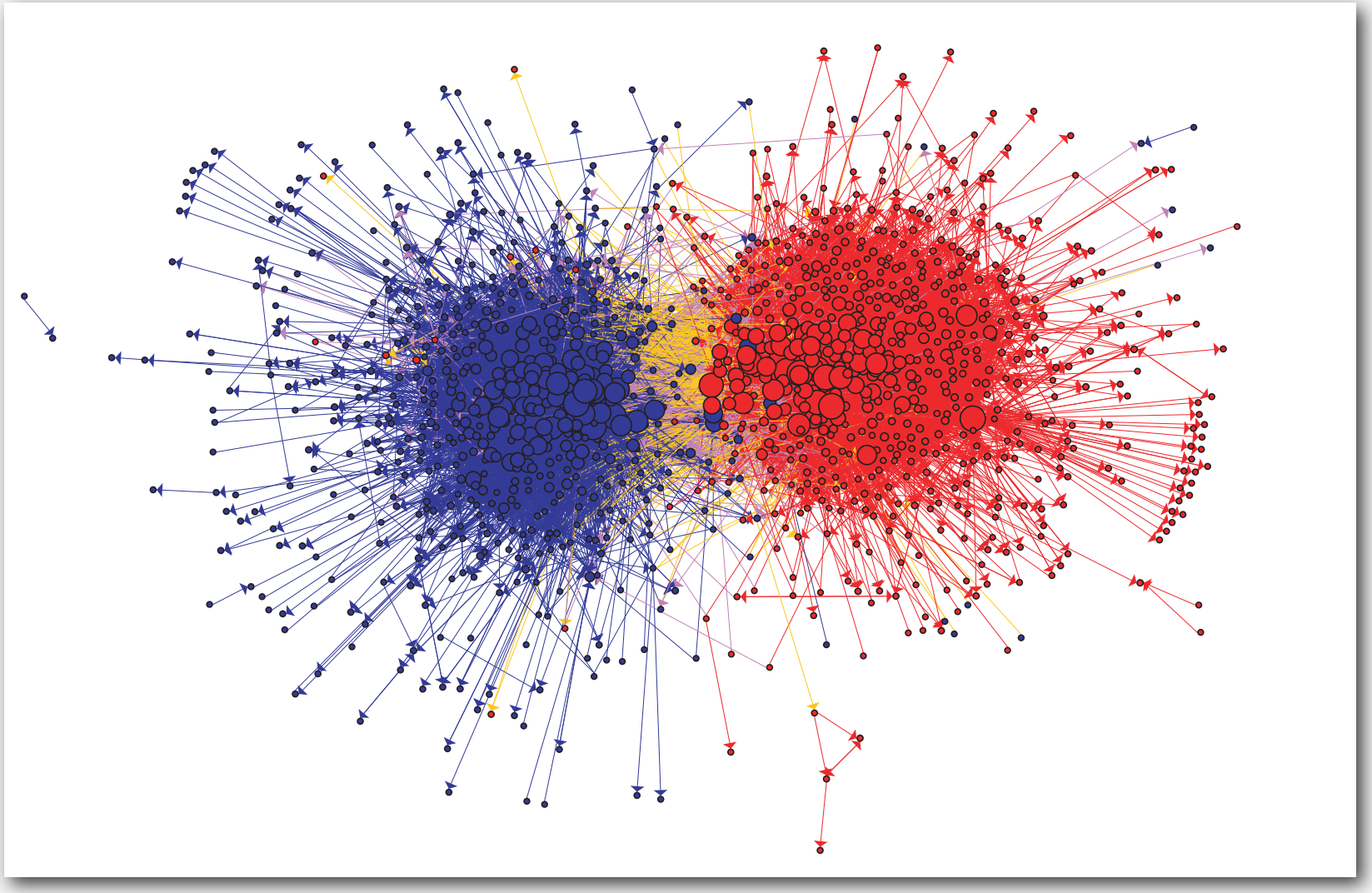
Road network

Vertex = intersection; edge = one-way street.



Political blogosphere graph

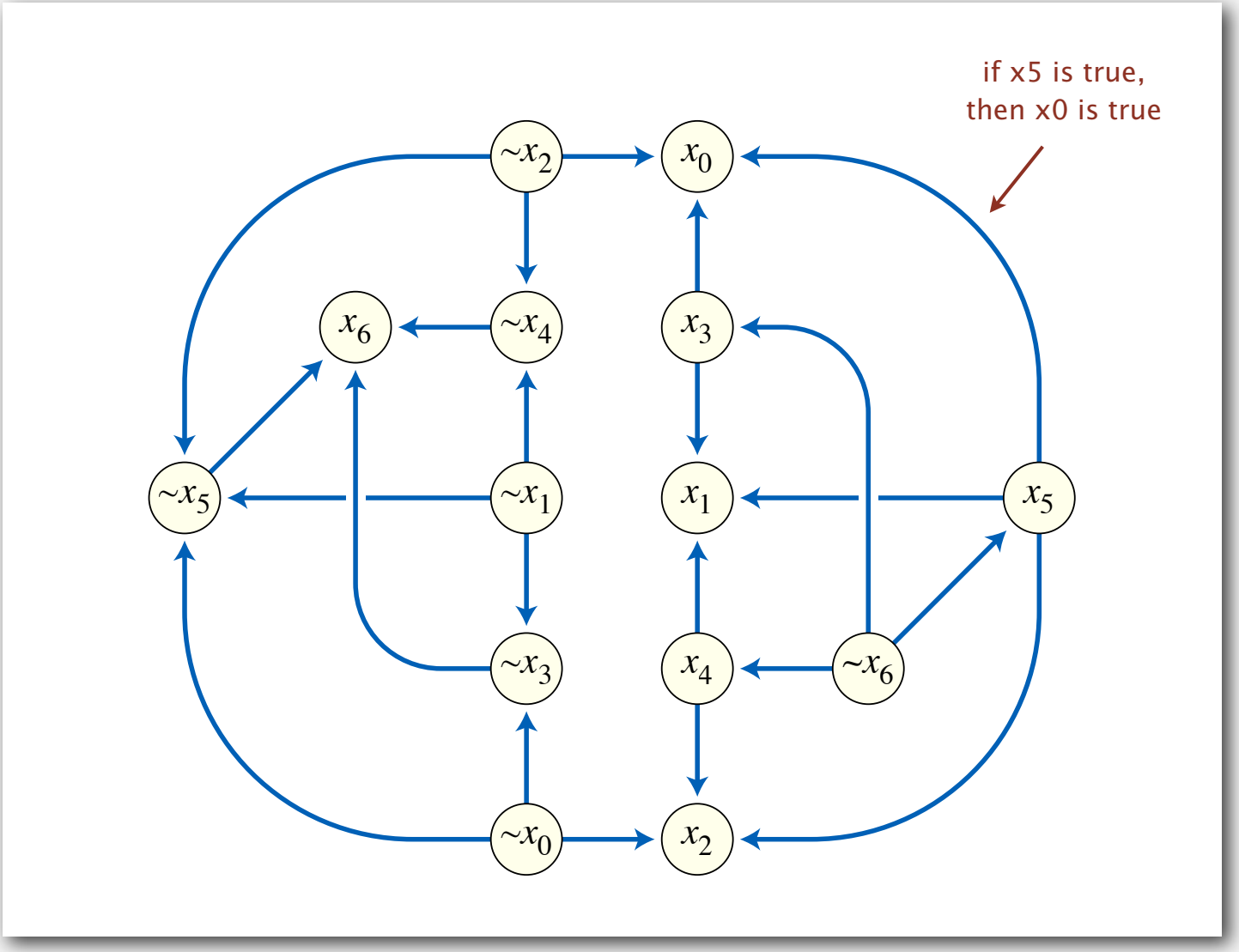
Vertex = political blog; edge = link.



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

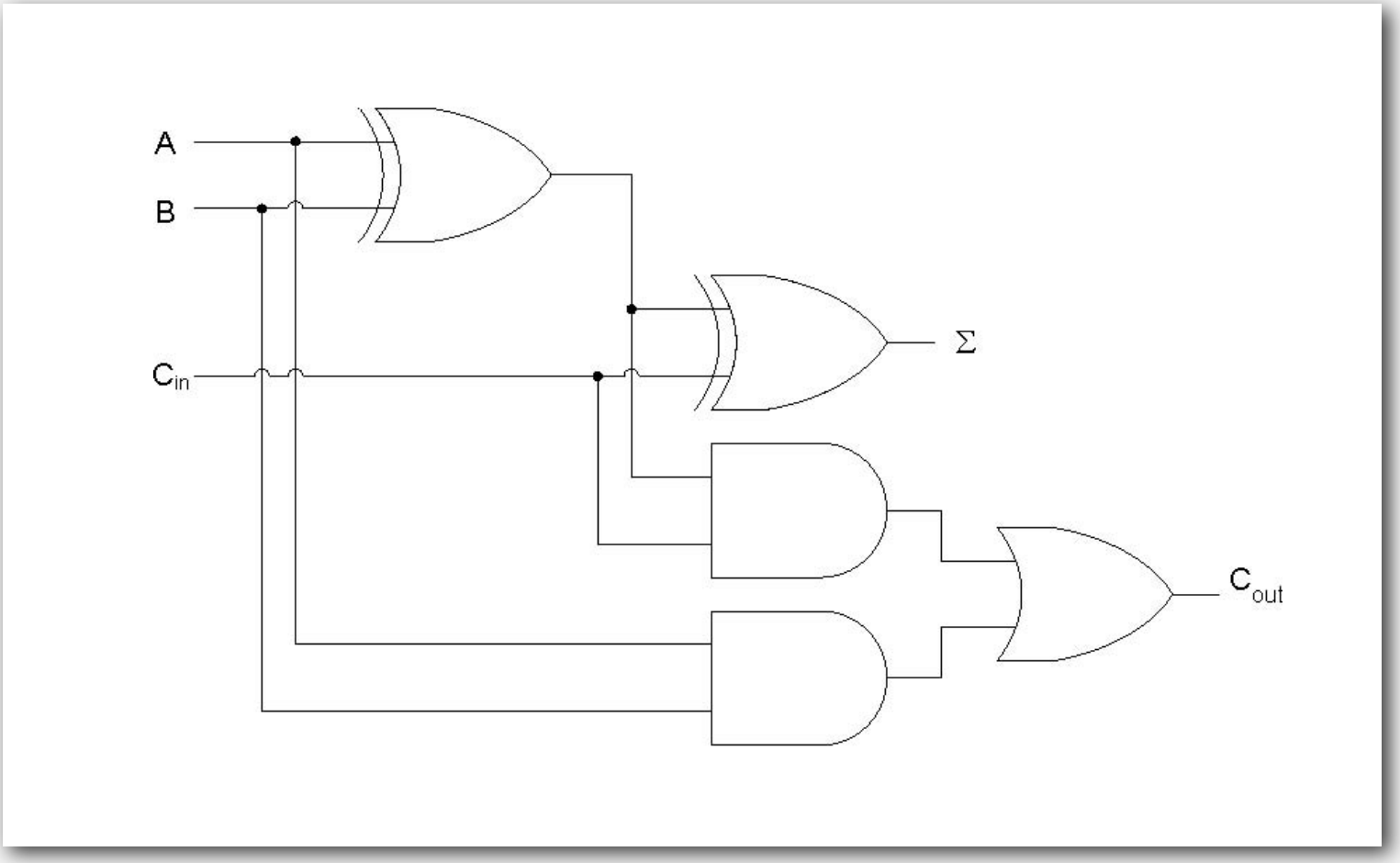
Implication graph

Vertex = variable; edge = logical implication.



Combinational circuit

Vertex = logical gate; edge = wire.



Digraph applications

digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Some digraph problems

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

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

Topological sort. Can you draw the digraph so that all edges point 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?

- 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);
```

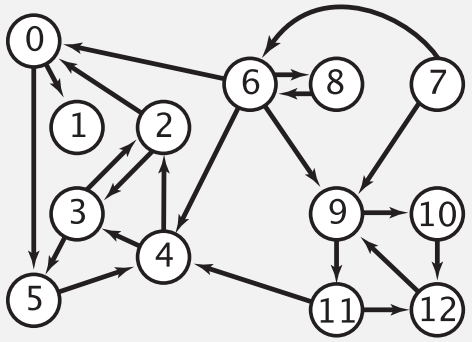
read digraph from input stream

print out each edge (once)

Digraph API

tinyDG.txt

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




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

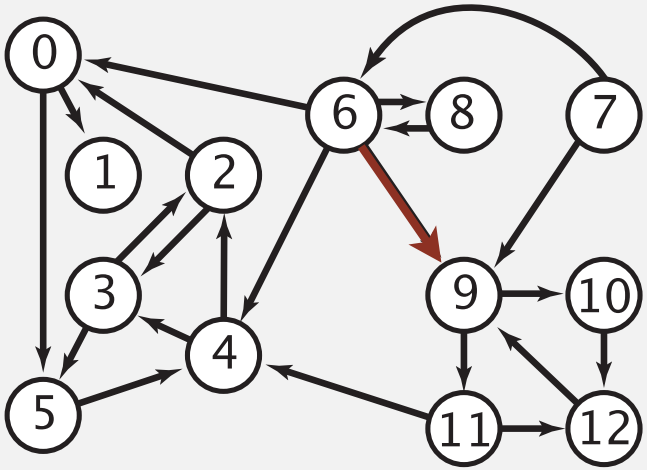
```
In in = new In(args[0]);
Digraph G = new Digraph(in);

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

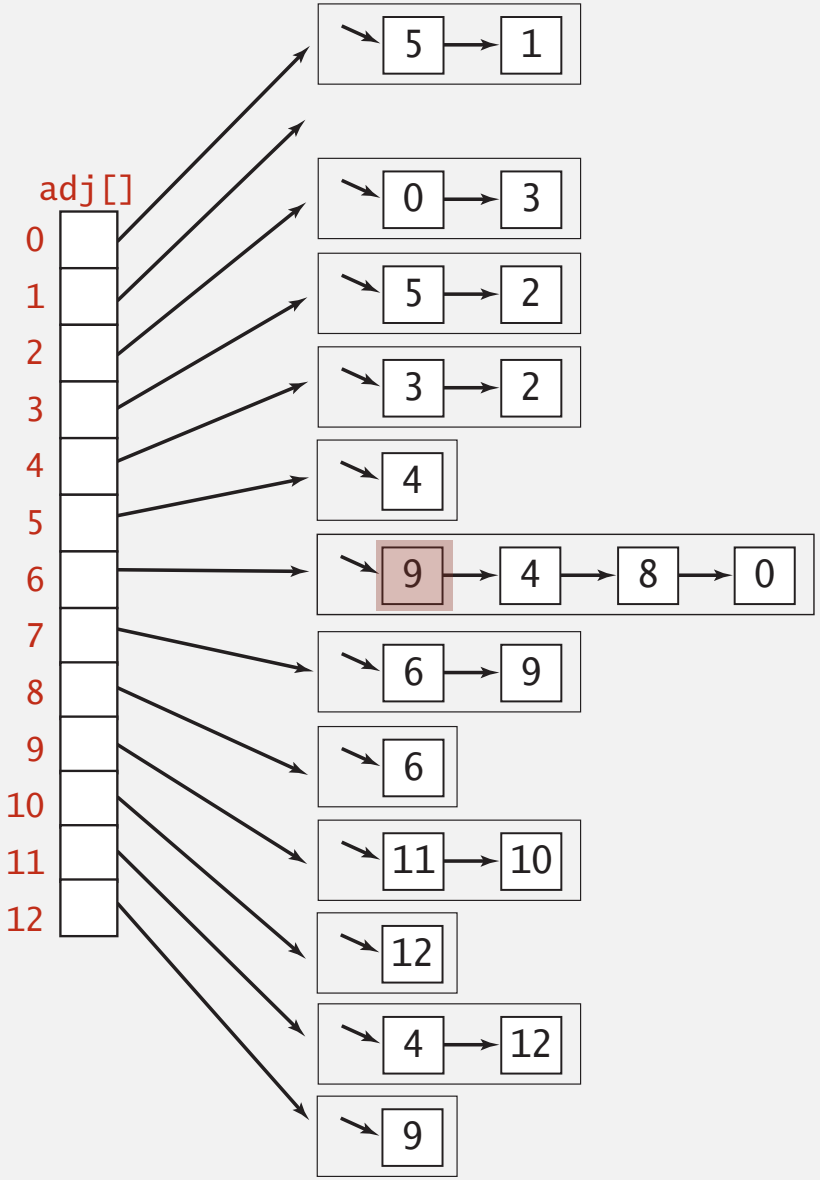
read digraph from input stream

print out each edge (once)

Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.





Adjacency-lists graph representation: 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

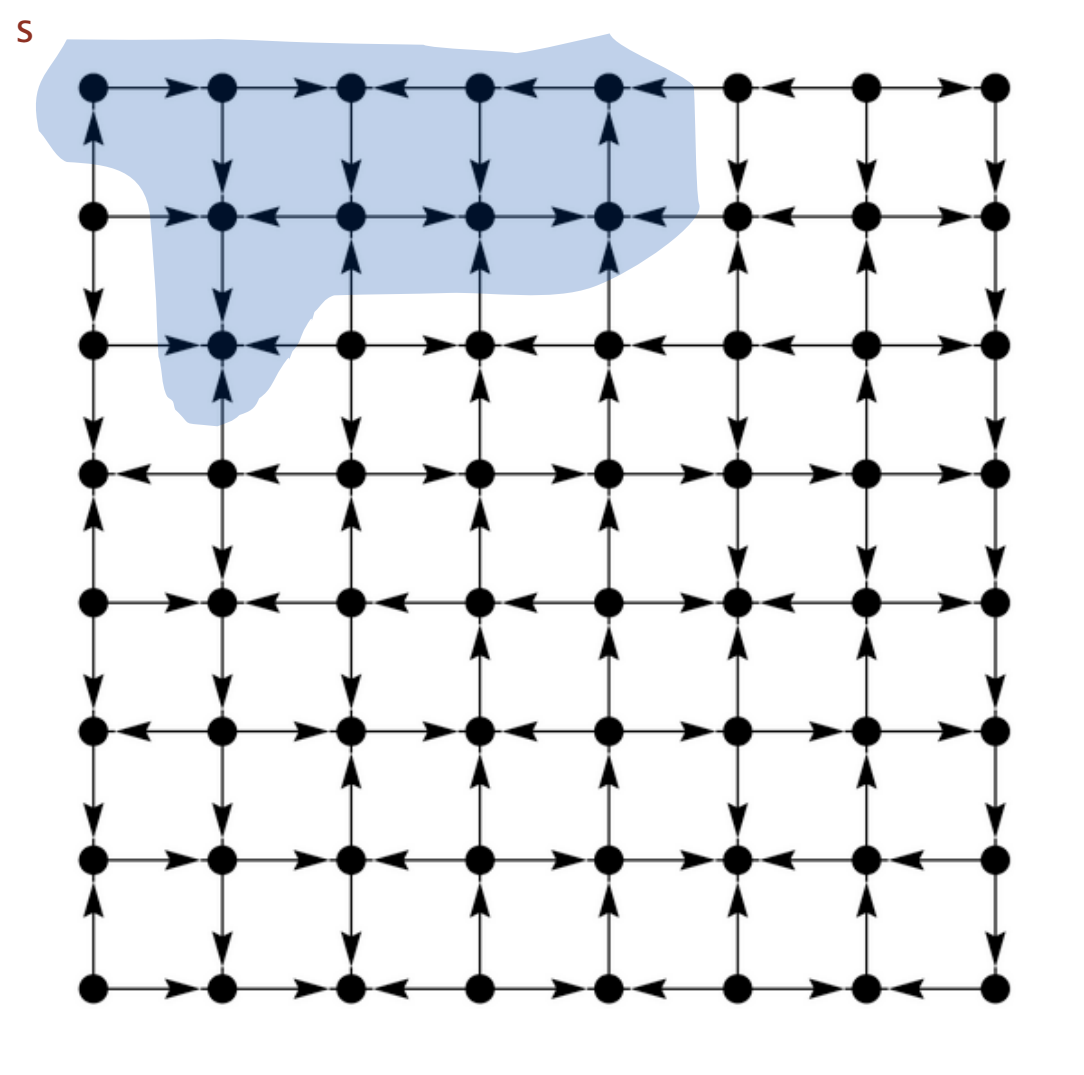
representation	space	insert edge from v to w	edge from v to w?	iterate over vertices pointing from v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 †	1	V
adjacency lists	E + V	1	outdegree(v)	outdegree(v)

† disallows parallel edges

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

Reachability

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



Depth-first search in digraphs

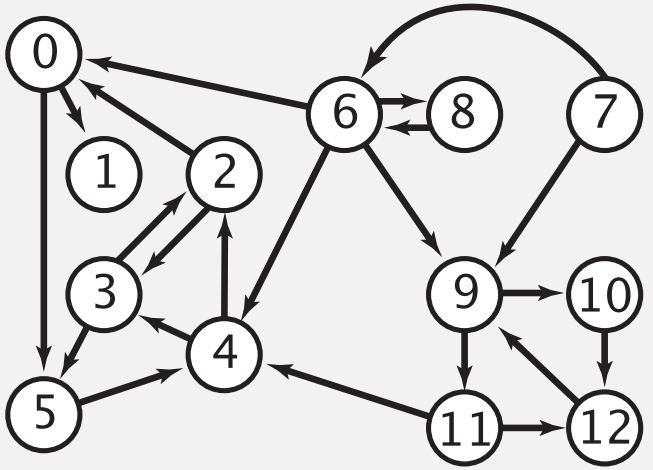
Same method as for undirected graphs.

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

DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked vertices w pointing from v.



Depth-first search (in undirected graphs)

Recall code for undirected graphs.

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

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

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

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

true if path to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is connected to s

Depth-first search (in directed graphs)

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];  }
}
```

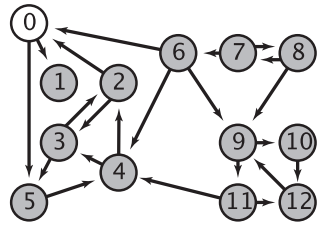
true if path from s

constructor marks vertices reachable from s

recursive DFS does the work

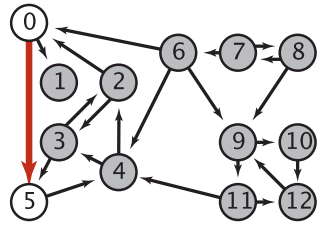
client can ask whether any vertex is reachable from s

```
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

    dfs(1)
    1 done
0 done
```



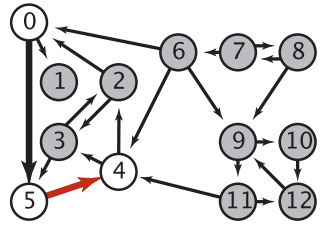
marked[]	
0	T
1	
2	
3	
4	
5	

adj[]		
0	5	1
1		
2	0	3
3	5	2
4	3	2
5	4	



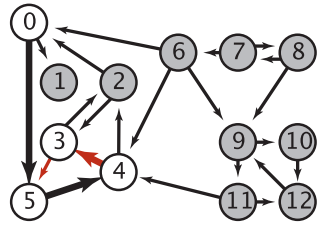
marked[]	
0	T
1	
2	
3	
4	
5	T

adj[]		
0	5	1
1		
2	0	3
3	5	2
4	3	2
5	4	



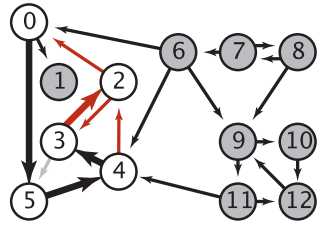
marked[]	
0	T
1	
2	
3	
4	T
5	T

adj[]		
0	5	1
1		
2	0	3
3	5	2
4	3	2
5	4	



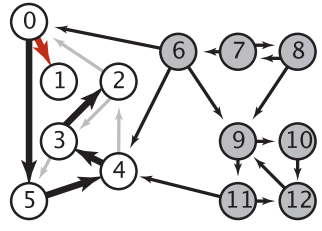
marked[]	
0	T
1	
2	
3	T
4	T
5	T

adj[]		
0	5	1
1		
2	0	3
3	5	2
4	3	2
5	4	



marked[]	
0	T
1	
2	T
3	T
4	T
5	T

adj[]		
0	5	1
1		
2	0	3
3	5	2
4	3	2
5	4	



marked[]	
0	T
1	T
2	T
3	T
4	T
5	T

adj[]		
0	5	1
1		
2	0	3
3	5	2
4	3	
5	4	

Reachability application: program control-flow analysis

Every program is a digraph.

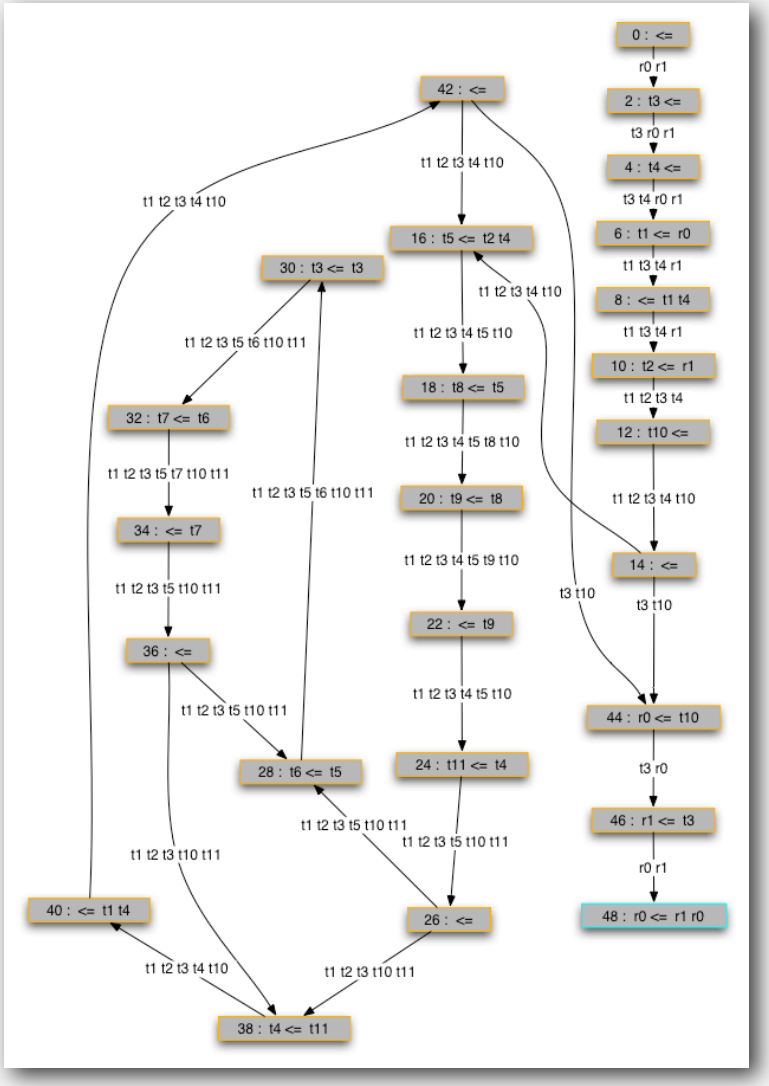
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



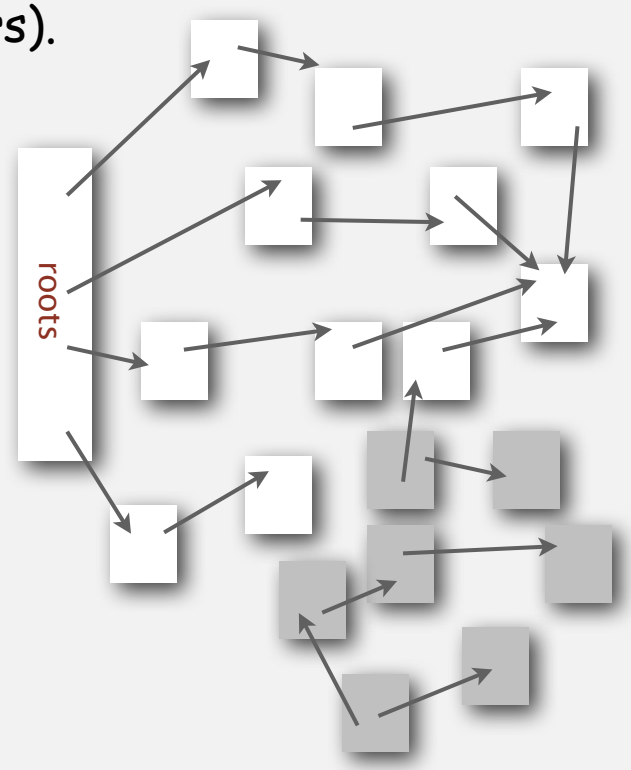
Reachability application: mark-sweep garbage collector

Every data structure is a digraph.

- Vertex = object.
- Edge = reference.

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

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

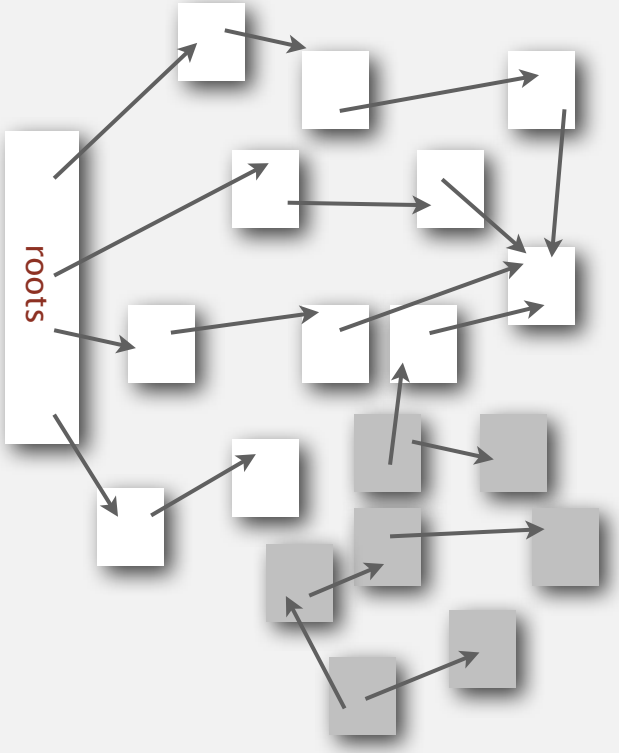


Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

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

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



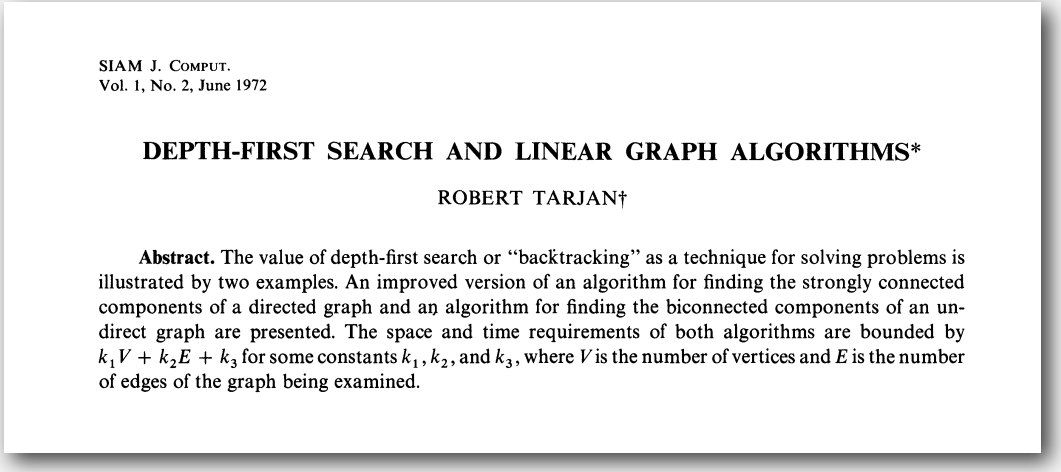
Depth-first search in digraphs summary

DFS enables direct solution of simple digraph problems.

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

Basis for solving difficult digraph problems.

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



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

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

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

Breadth-first search in digraphs

Same method as for undirected graphs.

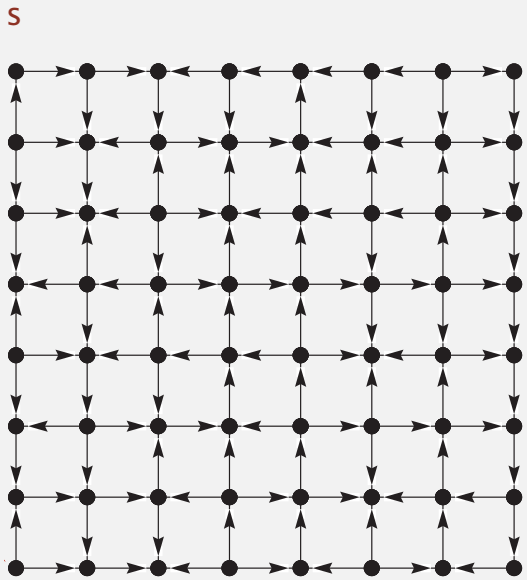
- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- **remove the least recently added vertex v**
- **for each unmarked vertex pointing from v:**
  add to queue and mark as visited.

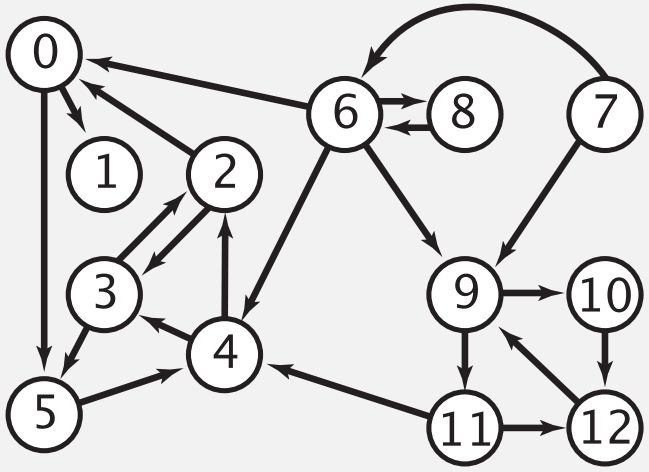


Proposition. BFS computes shortest paths (fewest number of edges).

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. Shortest path from $\{ 1, 7, 10 \}$ to 5 is $7 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 5$.



Q. How to implement multi-source constructor?

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

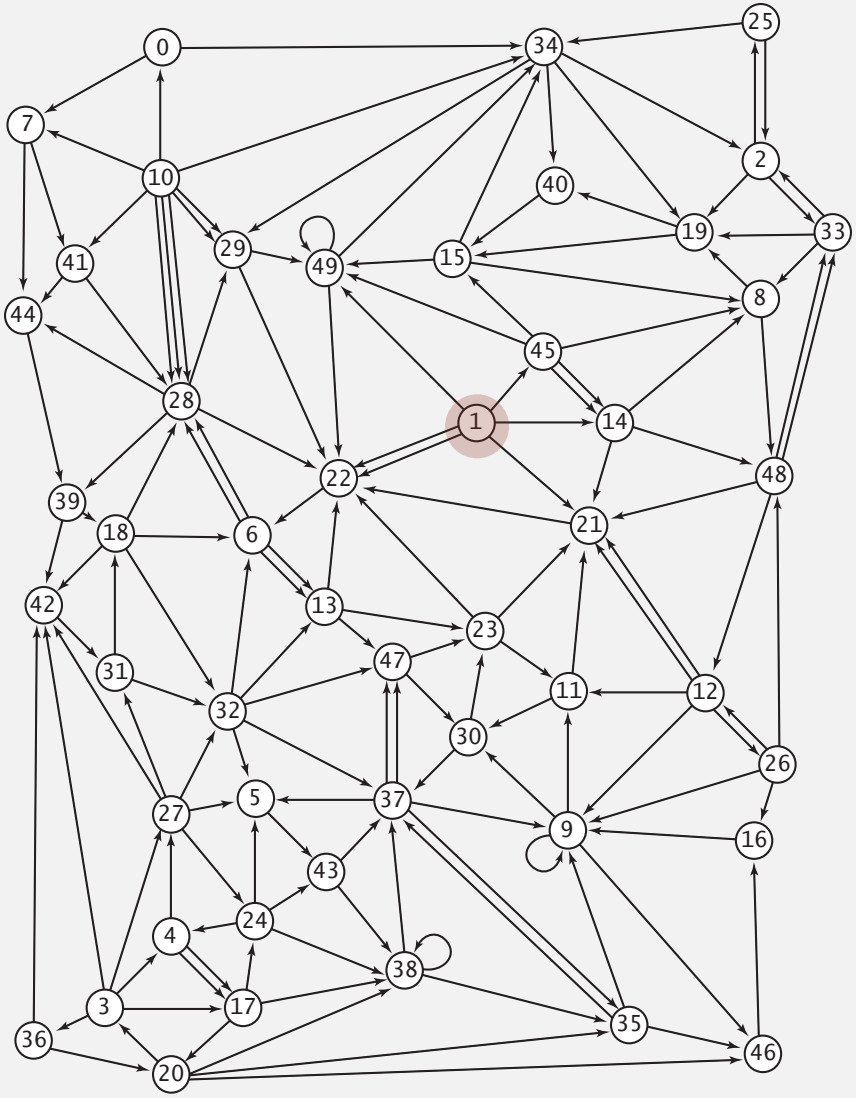
Breadth-first search in digraphs application: web crawler

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

Solution. BFS with implicit graph.

BFS.

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



Q. Why not use DFS?

Bare-bones web crawler: Java implementation

```
Queue<String> queue = new Queue<String>();
SET<String> discovered = new SET<String>();

String root = "http://www.princeton.edu";
queue.enqueue(root);
discovered.add(root);

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

queue of websites to crawl

set of discovered websites

start crawling from root website

read in raw html from next website in queue

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

if undiscovered, mark it as discovered and put on queue

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

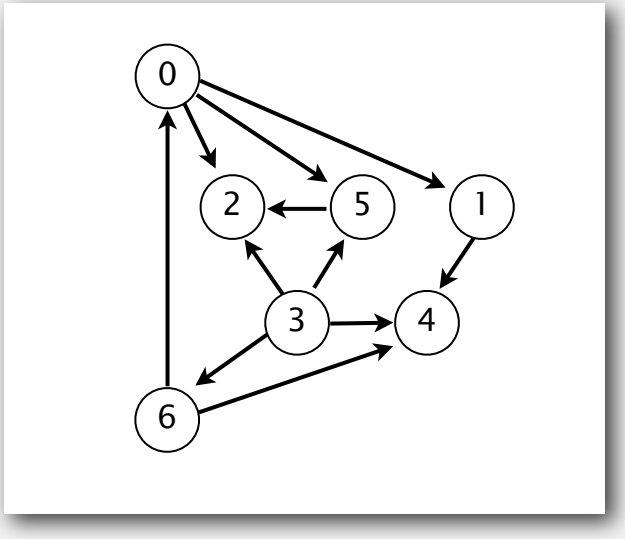
Precedence scheduling

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

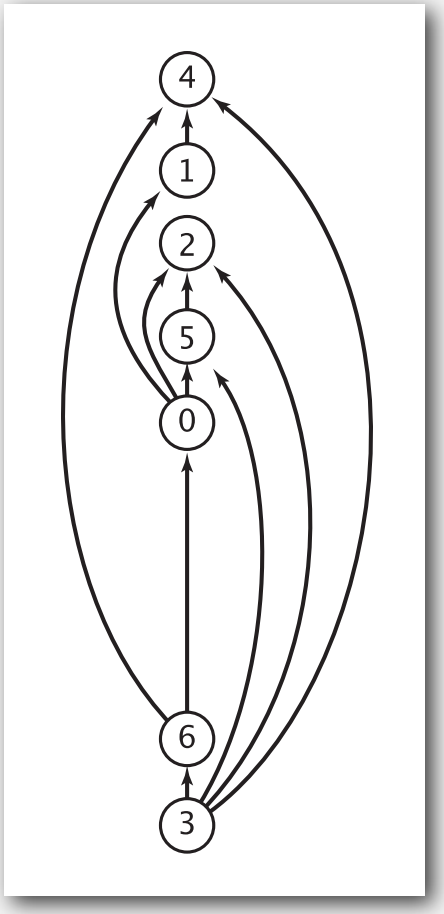
Digraph model. vertex = task; edge = precedence constraint.

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

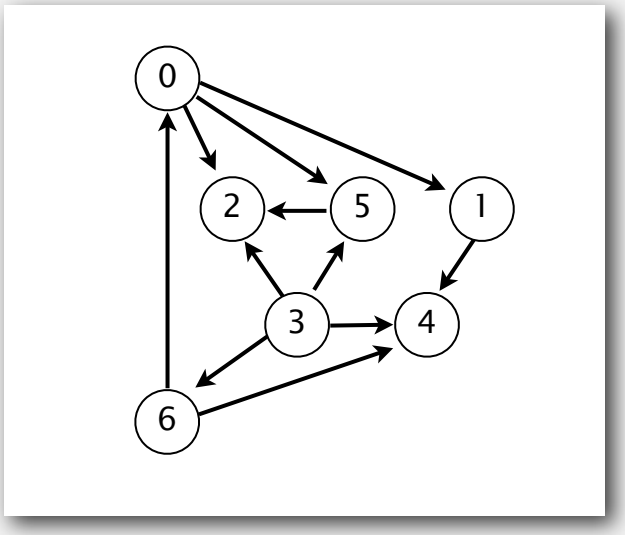
Topological sort

DAG. Directed acyclic graph.

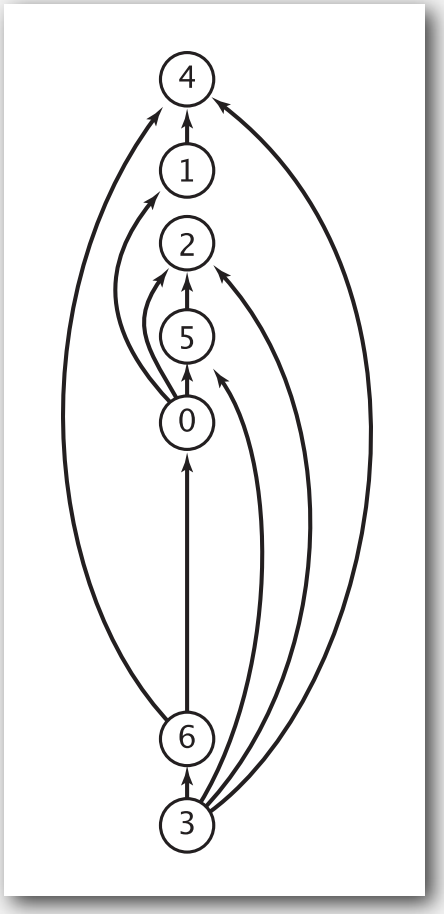
Topological sort. Redraw DAG so all edges point upwards.

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

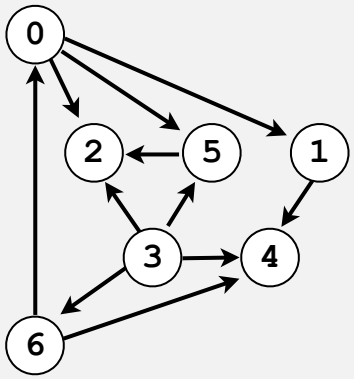
Solution. DFS. What else?

Depth-first search order

```
public class DepthFirstOrder
{
   private boolean[] marked;
   private Stack<Integer> reversePost;

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

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

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

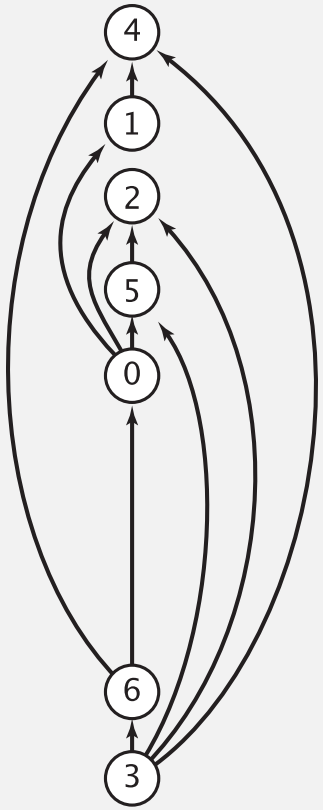
returns all vertices in "reverse DFS postorder"

Reverse DFS postorder in a DAG



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

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



reverse DFS postorder is a topological order!

Topological sort in a DAG: correctness proof

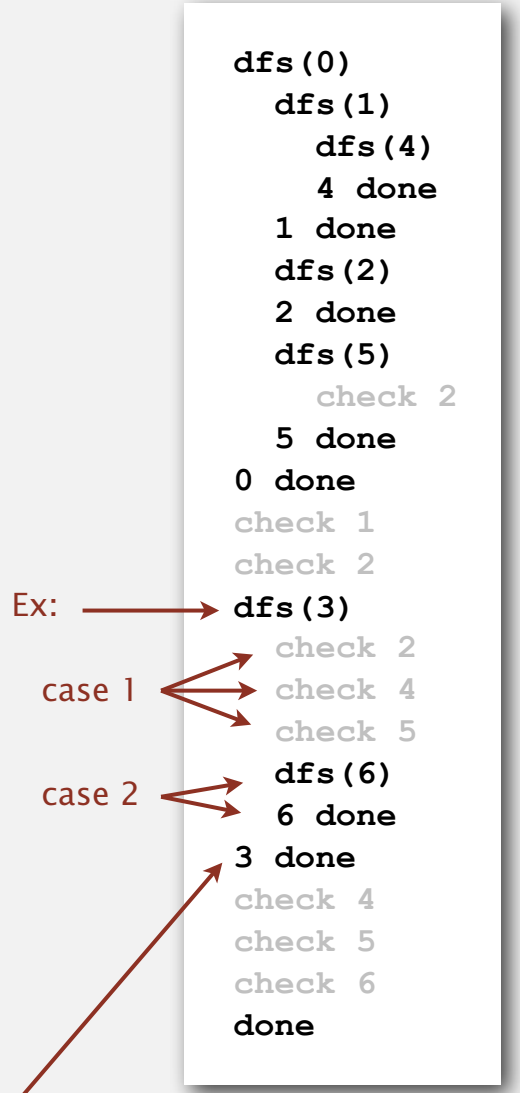
Proposition. Reverse DFS postorder of a DAG is a topological order.

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

- Case 1: `dfs(w)` has already been called and returned.
  Thus, w was done before v.

- Case 2: `dfs(w)` has not yet been called.
  `dfs(w)` will get called directly or indirectly
  by `dfs(v)` and will finish before `dfs(v)`.
  Thus, w will be done before v.

- Case 3: `dfs(w)` has already been called,
  but has not yet returned.
  Can't happen in a DAG: function call stack contains
  path from w to v, so $v \rightarrow w$ would complete a cycle.

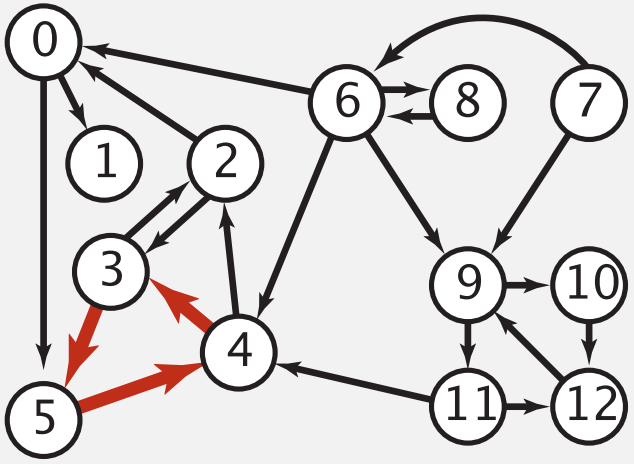


Directed cycle detection

Proposition. A digraph has a topological order iff no directed cycle.

Pf.

- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.



a digraph with a directed cycle

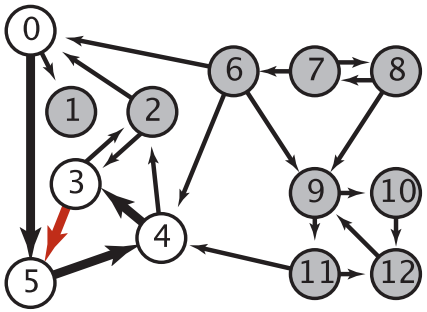
Goal. Given a digraph, find a directed cycle.

Solution. DFS. What else? See textbook.

Finding a directed cycle

public class	DirectedCycle	
	DirectedCycle(Digraph G)	*cycle-finding constructor*
boolean	hasCycle()	*does* G *have a directed cycle?*
Iterable<Integer>	cycle()	*vertices on a cycle (if one exists)*

API for reachability in digraphs

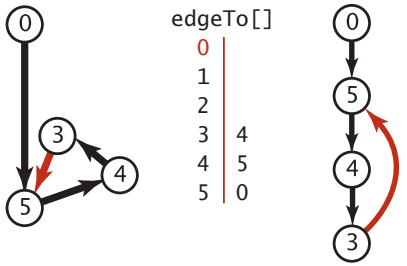


	marked[] 0 1 2 3 4 5 ...	edgeTo[] 0 1 2 3 4 5 ...	onStack[] 0 1 2 3 4 5 ...
dfs(0)			
dfs(5)	1 0 0 0 0 0	- - - - - 0	1 0 0 0 0 0
dfs(4)	1 0 0 0 0 1	- - - - 5 0	1 0 0 0 0 1
dfs(3)	1 0 0 0 1 1	- - - 4 5 0	1 0 0 0 1 1
check 5	1 0 0 1 1 1	- - - 4 5 0	1 0 0 1 1 1

Finding a directed cycle in a digraph

Finding a directed cycle

```
public class DirectedCycle
{
   private boolean[] marked;
   private int[] edgeTo;
   private Stack<Integer> cycle;   // vertices on a cycle (if one exists)
   private boolean[] onStack;      // vertices on recursive call stack

   public DirectedCycle(Digraph G)
   {
      onStack = new boolean[G.V()];
      edgeTo  = new int[G.V()];
      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)
   {
      onStack[v] = true;
      marked[v] = true;
      for (int w : G.adj(v))
         if (this.hasCycle()) return;
         else if (!marked[w])
         {  edgeTo[w] = v; dfs(G, w);  }
         else if (onStack[w])
         {
            cycle = new Stack<Integer>();
            for (int x = v; x != w; x = edgeTo[x])
               cycle.push(x);
            cycle.push(w);
            cycle.push(v);
         }
      onStack[v] = false;
   }
```

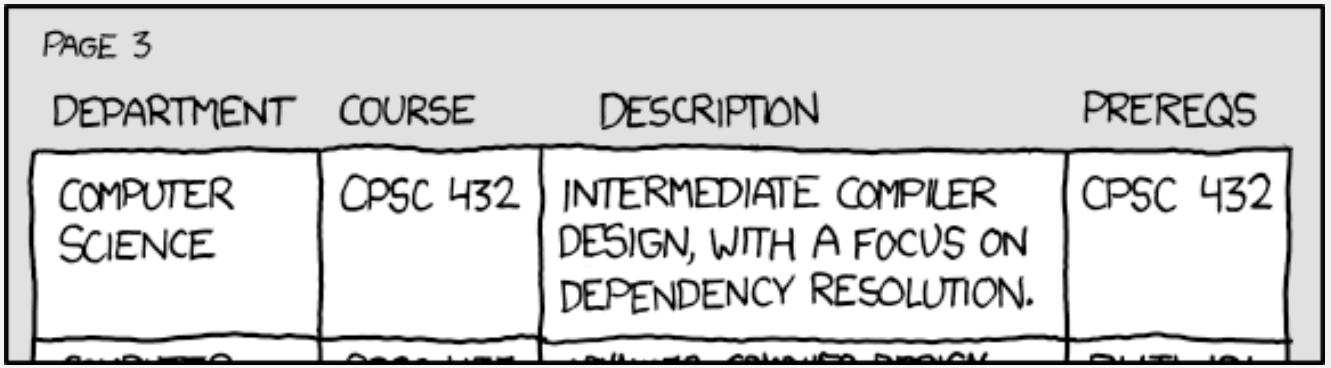


v	w	x	cycle
3	5	3	3
3	5	4	4 3
3	5	4	5 4 3
3	5	4	3 5 4 3

Trace of cycle computation

Directed cycle detection application: precedence scheduling

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



http://xkcd.com/754

Remark. A directed cycle implies scheduling problem is infeasible.

Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

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

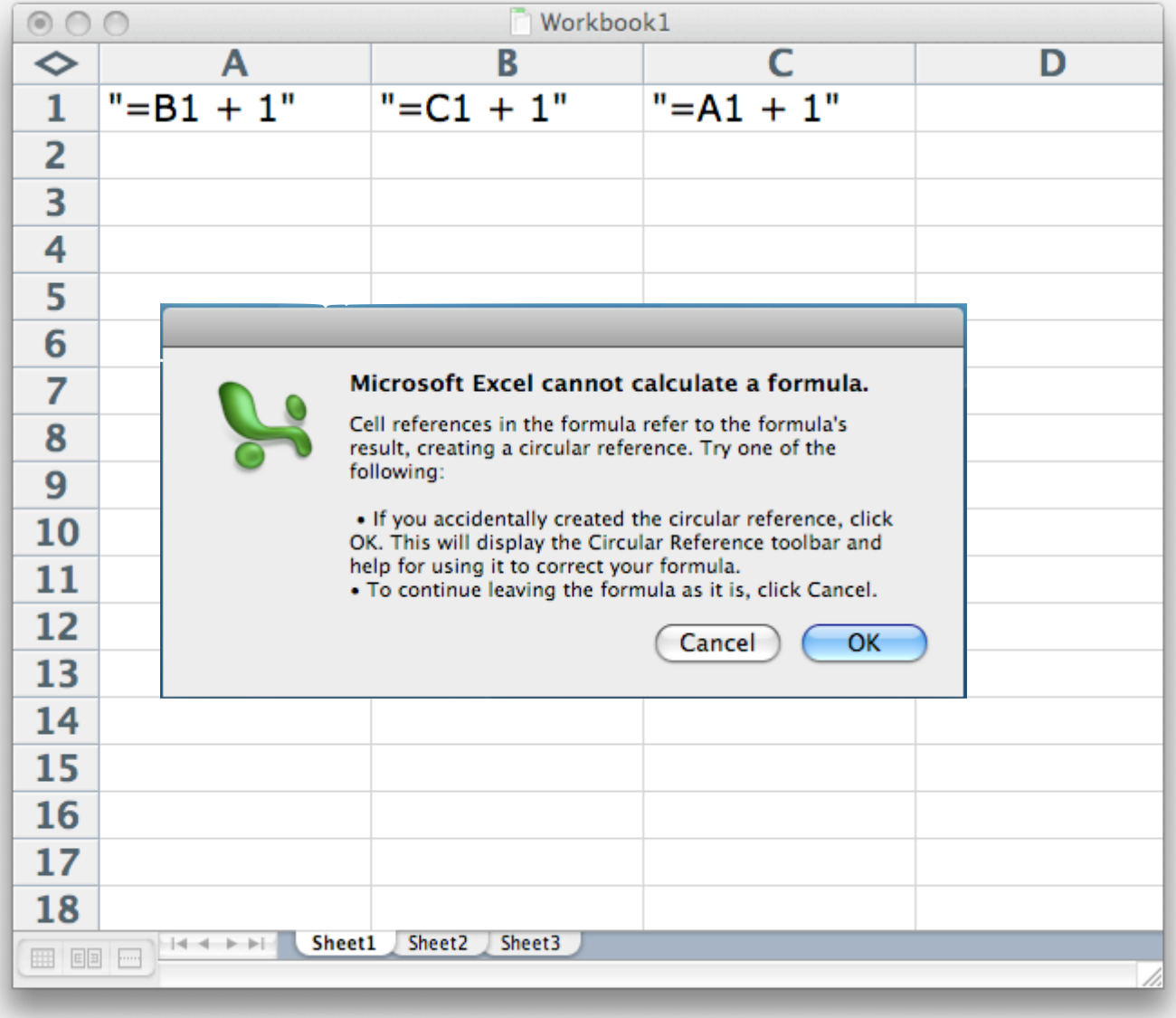
```
public class B extends C
{
   ...
}
```

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

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

Directed cycle detection application: spreadsheet recalculation

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



Directed cycle detection application: symbolic links

The Linux file system does not do cycle detection.

```
% ln -s a.txt b.txt
% ln -s b.txt c.txt
% ln -s c.txt a.txt

% more a.txt
a.txt: Too many levels of symbolic links
```

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

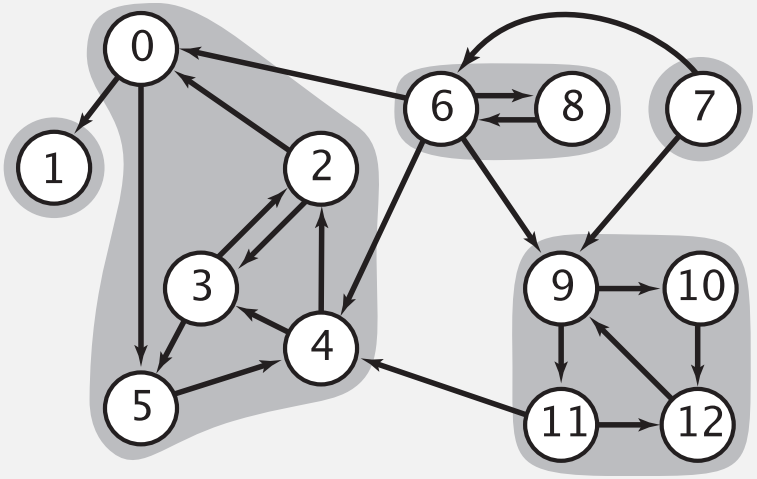
Strongly-connected components

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

Key property. Strong connectivity is an equivalence relation:

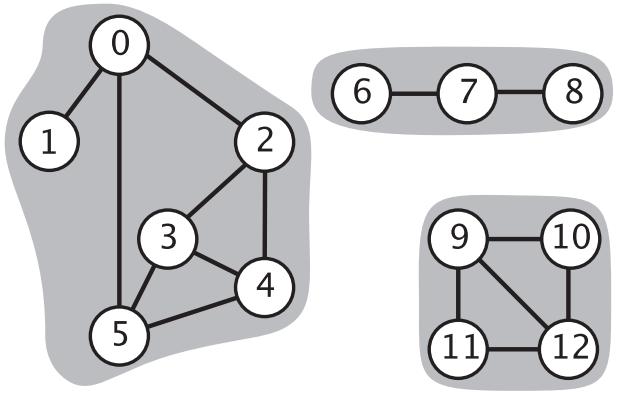
- v is strongly connected to v.
- If v is strongly connected to w, then w is strongly connected to v.
- If v is strongly connected to w and w to x, then v is strongly connected to x.

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



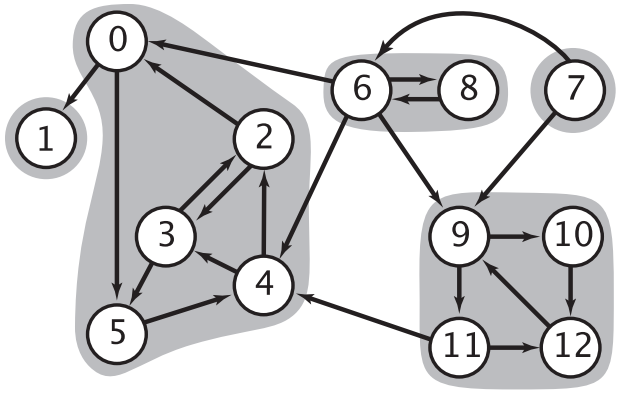
Connected components vs. strongly-connected components

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



3 connected components

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



5 strongly-connected components

connected component id (easy to compute with DFS)

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

strongly-connected component id (how to compute?)

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

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

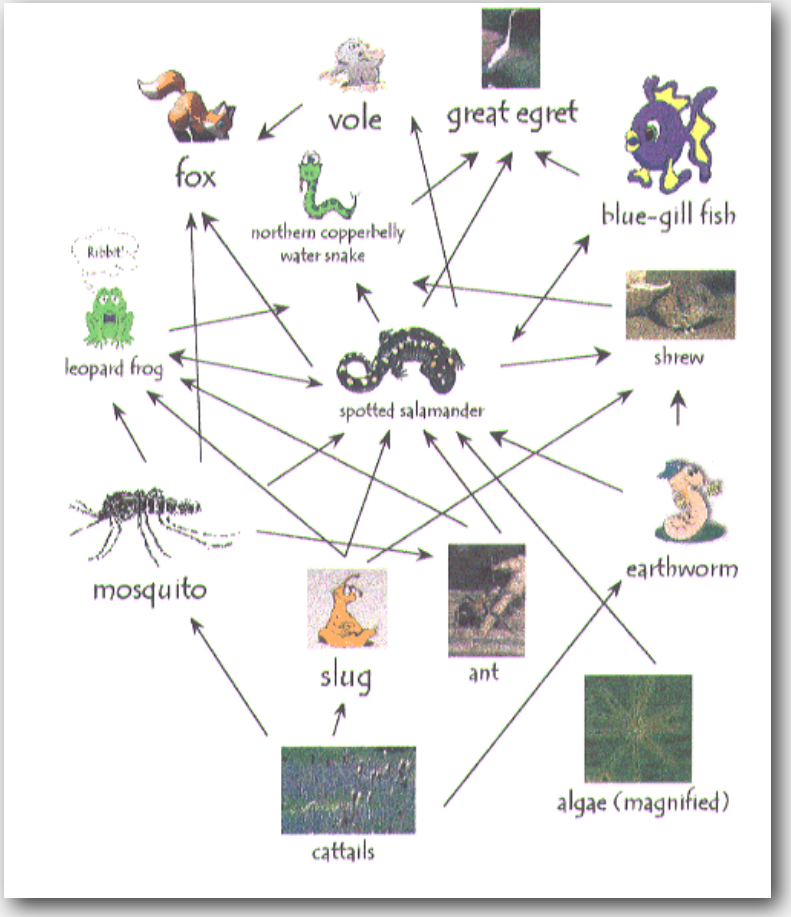
constant-time client connectivity query

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

constant-time client strong-connectivity query

Strong component application: ecological food webs

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



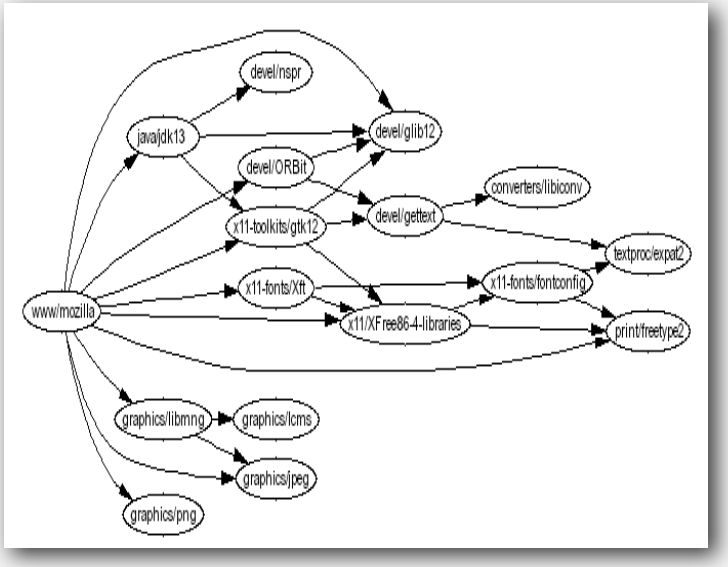
http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif

Strong component. Subset of species with common energy flow.

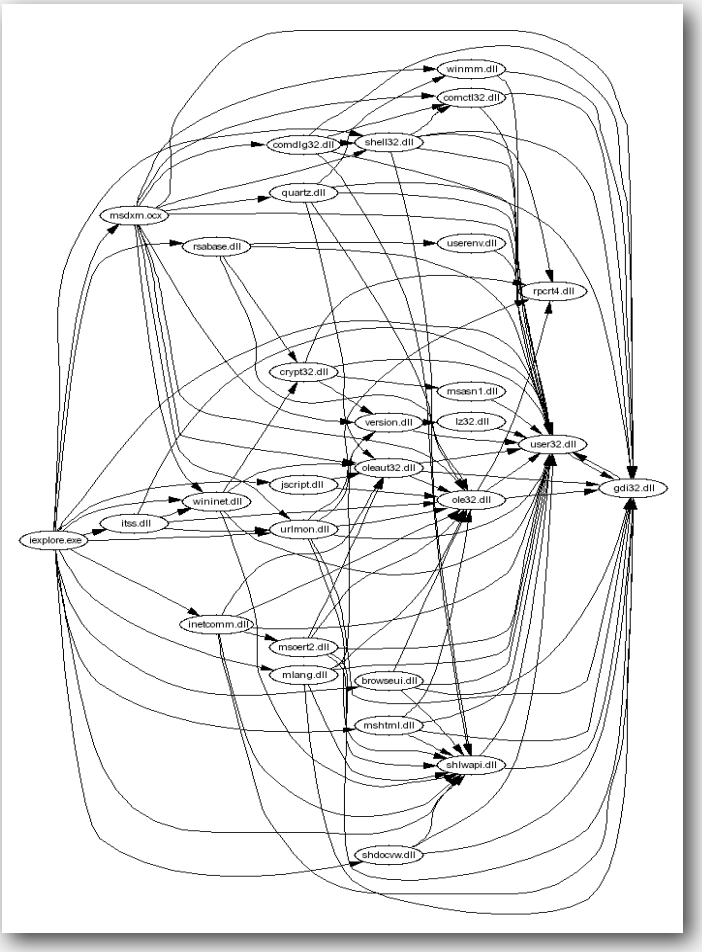
Strong component application: software modules

Software module dependency graph.

- Vertex = software module.
- Edge: from module to dependency.



Firefox



Internet Explorer

Strong component. Subset of mutually interacting modules.

Approach 1. Package strong components together.

Approach 2. Use to improve design!

Kosaraju's algorithm: intuition

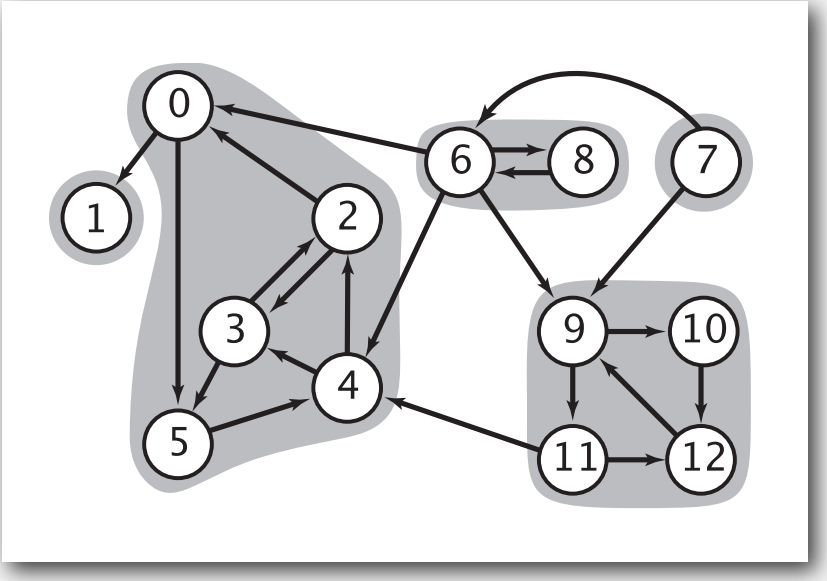
Reverse graph. Strong components in G are same as in G^R.

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

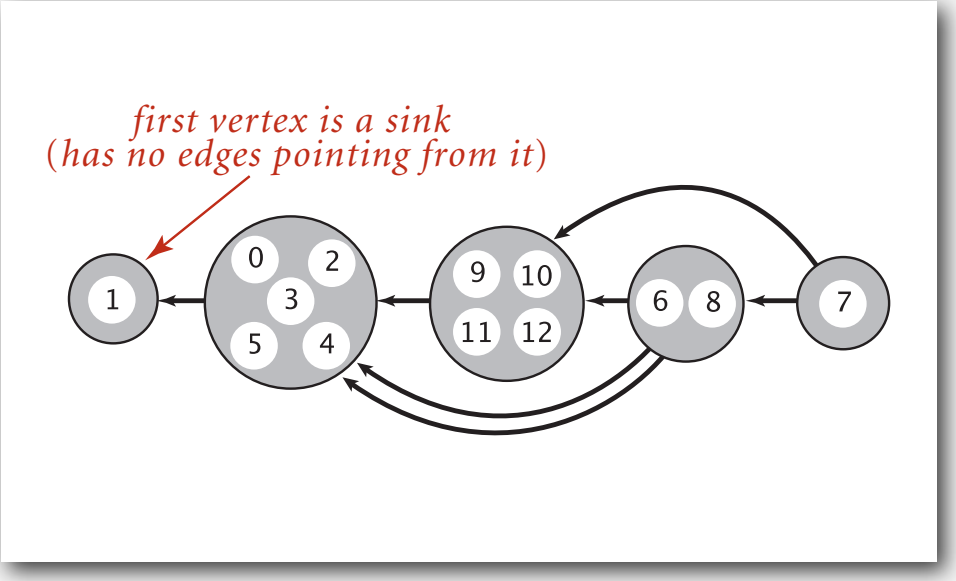
Idea.

how to compute?

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



digraph G and its strong components



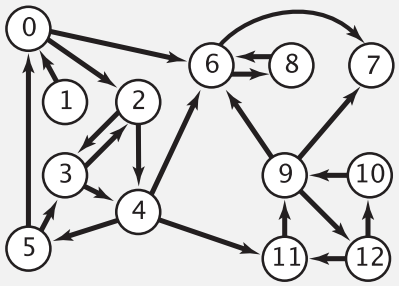
kernel DAG of G (in reverse topological order)

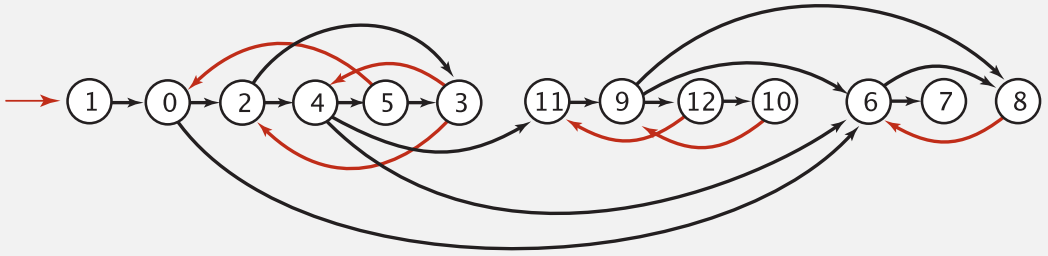
Kosaraju's algorithm

Simple (but mysterious) algorithm for computing strong components.

- Run DFS on G^R to compute reverse postorder.
- Run DFS on G, considering vertices in order given by first DFS.

DFS in reverse digraph 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

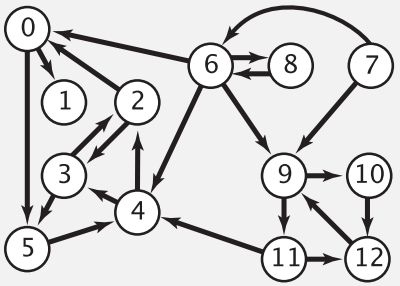
```
dfs(0)
  dfs(6)
    dfs(8)
      check 6
    8 done
    dfs(7)
    7 done
  6 done
  dfs(2)
    dfs(4)
      dfs(11)
        dfs(9)
          dfs(12)
            check 11
            dfs(10)
              check 9
            10 done
          12 done
          check 7
          check 6
. . .
```

Kosaraju's algorithm

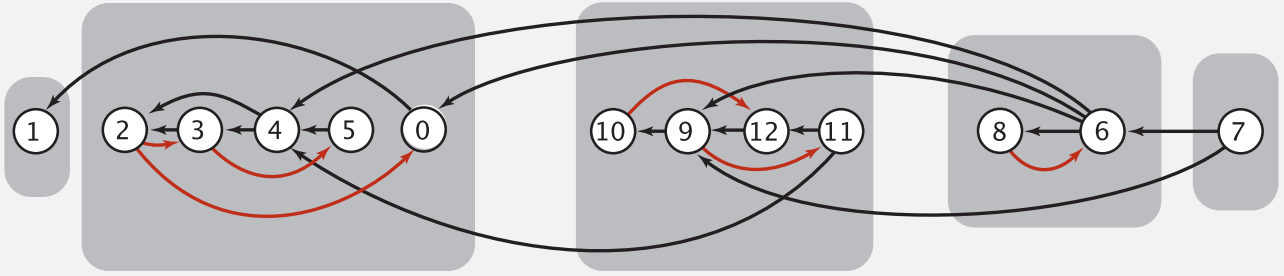
Simple (but mysterious) algorithm for computing strong components.

- Run DFS on G^R to compute reverse postorder.
- Run DFS on G, considering vertices in order given by first DFS.

DFS in original digraph G



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




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

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

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

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

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

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

Kosaraju proof of correctness

Proposition. Kosaraju's algorithm computes strong components.

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

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

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

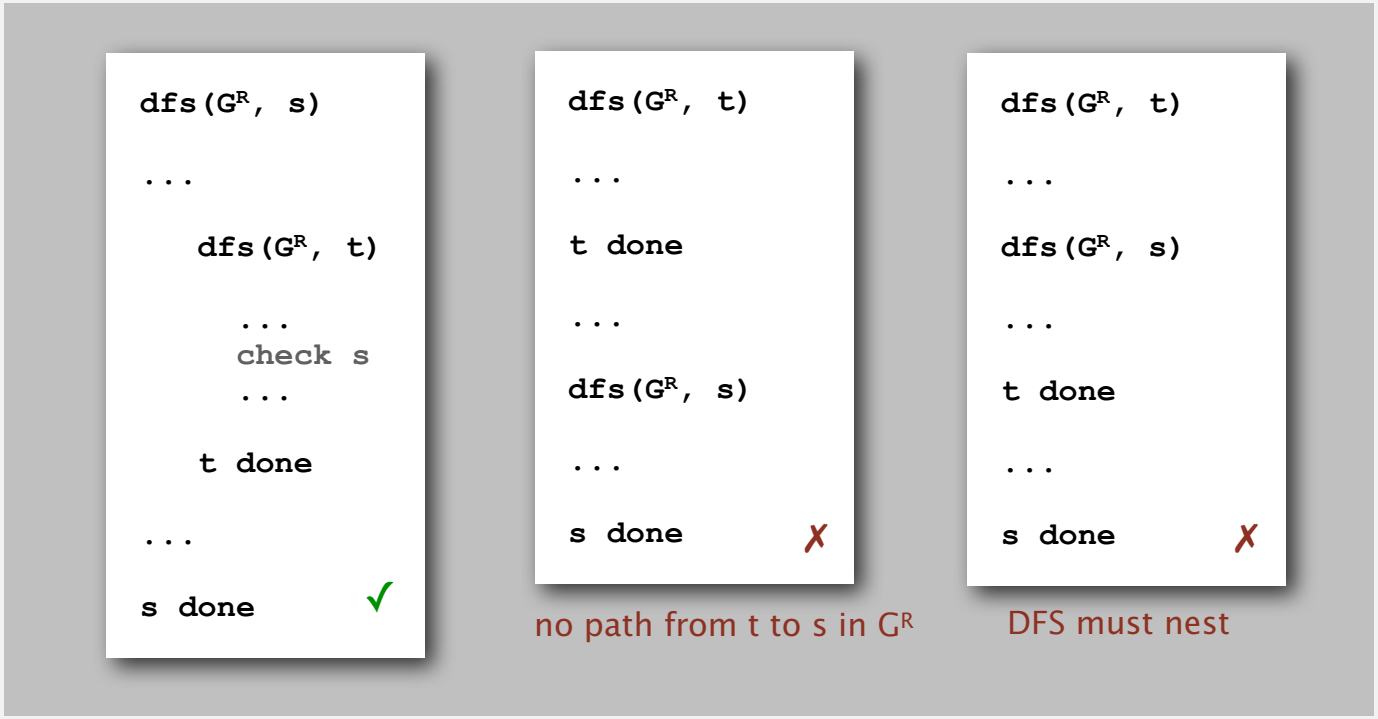
```
dfs(G, s)
...
   dfs(G, t)
      ...
      check s
      ...
   t done
...
s done
```

Kosaraju proof of correctness (continued)

Proposition. Kosaraju's algorithm computes strong components.

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

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



Connected components in an undirected graph (with DFS)

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

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

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

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

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

Strong components in a digraph (with two DFSs)

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

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

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

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