Introduction to Algorithms

Lecture 9: Graph Search Algorithm

Yao-Chung Fan
yfan@nchu.edu.tw

Undirected Graphs

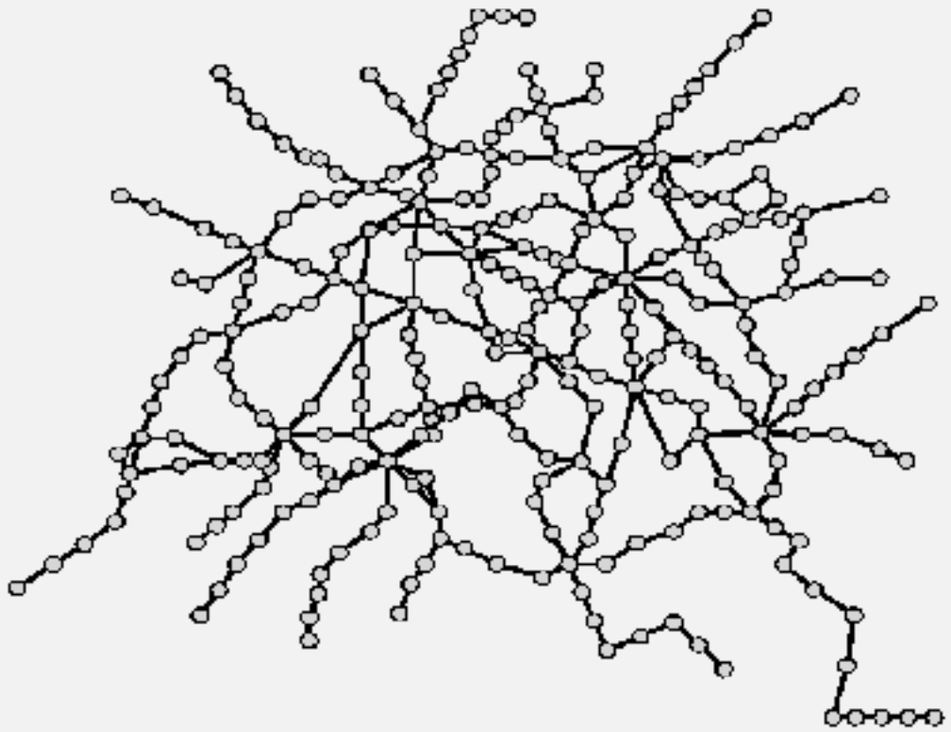
- *Introduction*
- *Graph API*
- *Depth-first Search*
- *Breadth-first Search*
- *Connected Components*

Undirected graphs

Graph. Set of vertices connected pairwise by edges.

Why study graph algorithms?

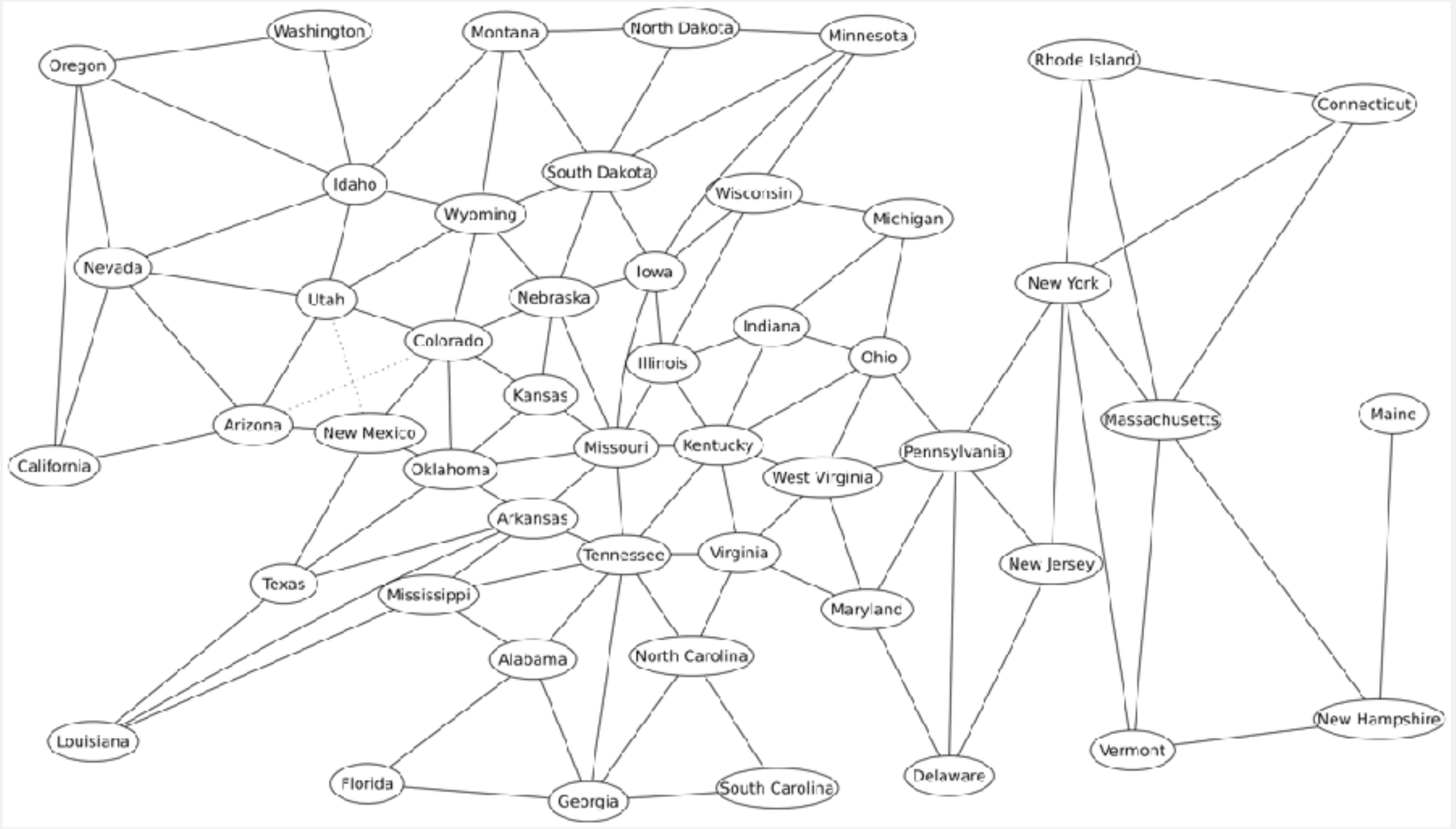
- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.



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	intersection	street
internet	class C network	connection
Software	Function	Function Call
social relationship	person	friendship
neural network	neuron	synapse
protein network	protein	protein-protein interaction
Web	Homepage	Hyperlink

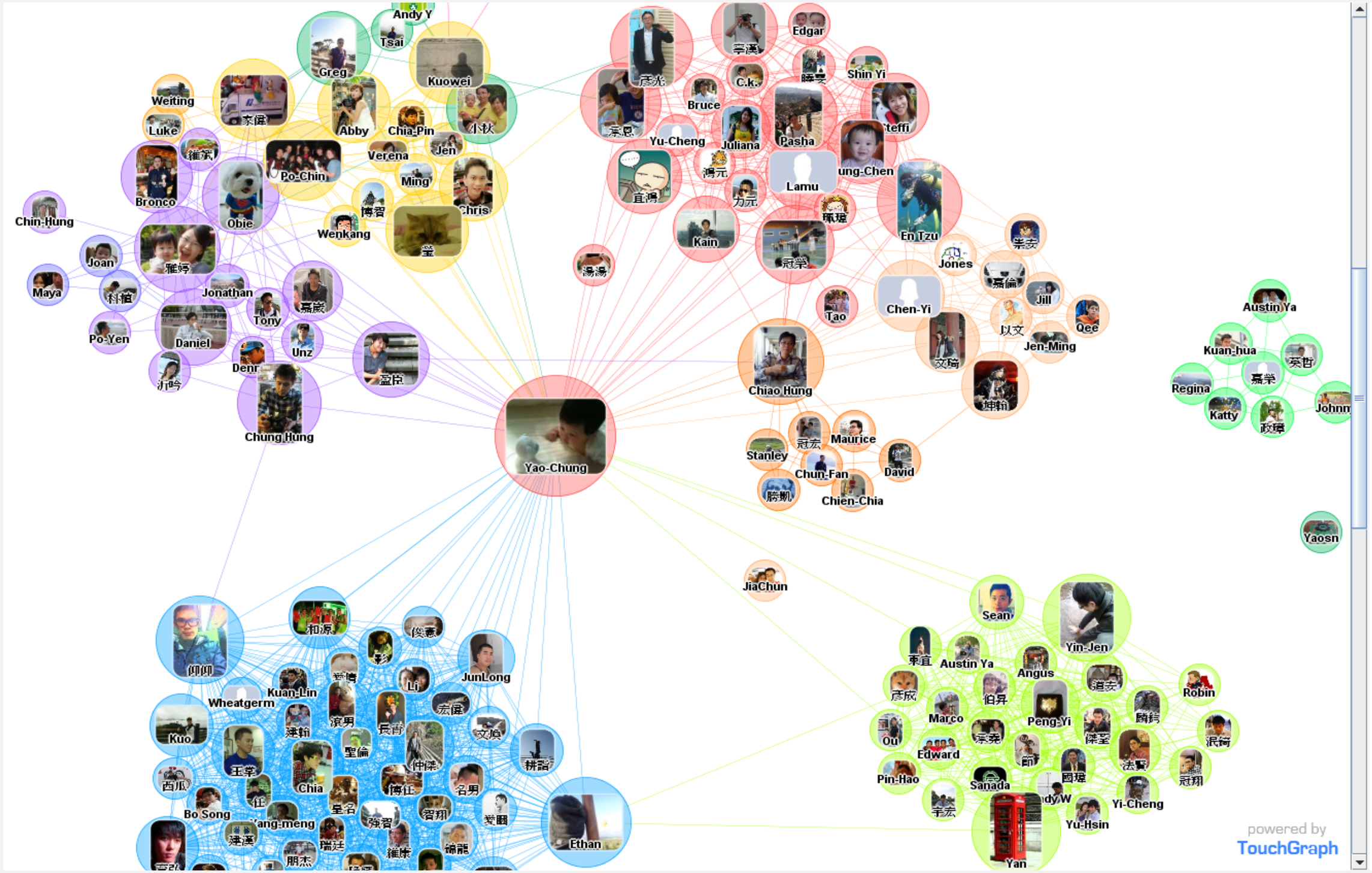
Border graph of 48 contiguous United States



10 million Facebook friends

"Visualizing Friendships" by Paul Butler

My facebook friends

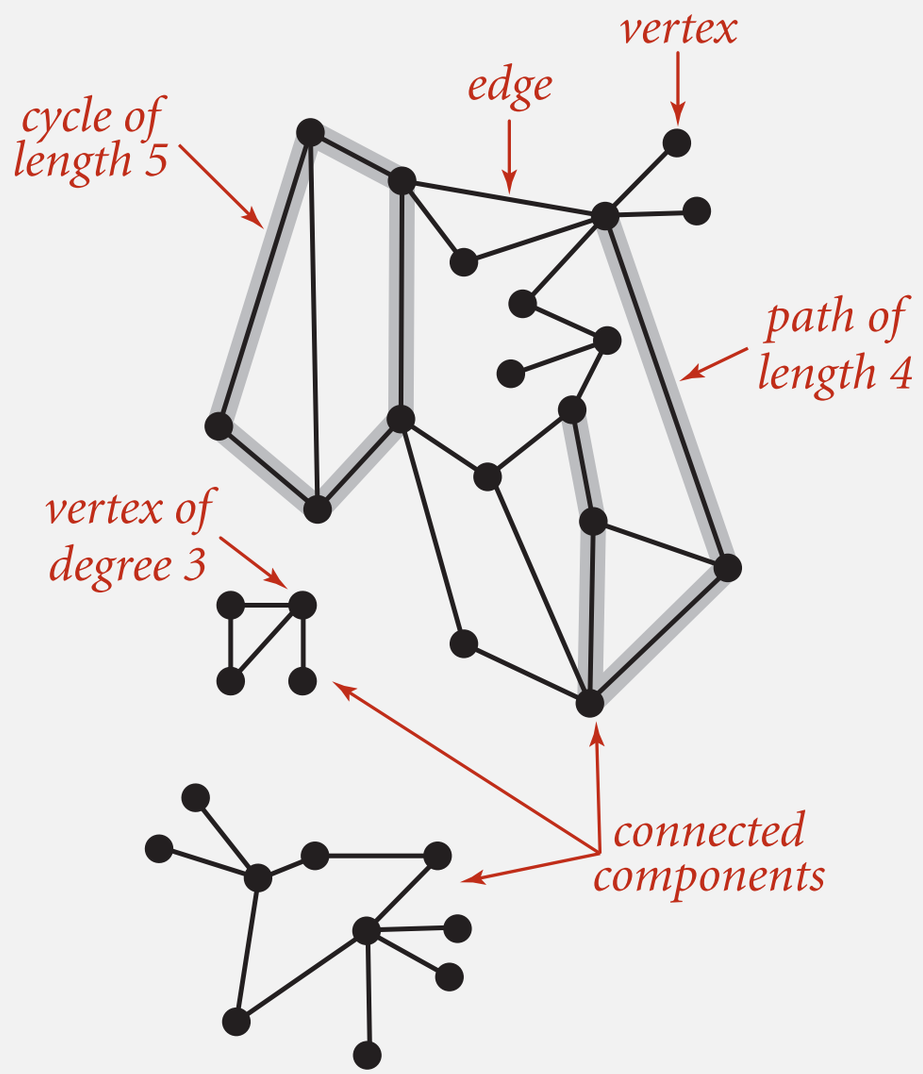


Graph terminology

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.



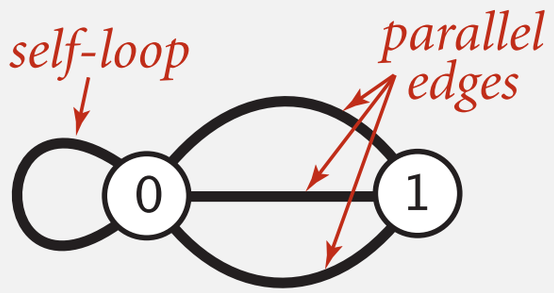
Undirected Graphs

- *Introduction*
- *Graph Representation*
- *Depth-first Search*
- *Breadth-first Search*
- *Connected Components*

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*

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



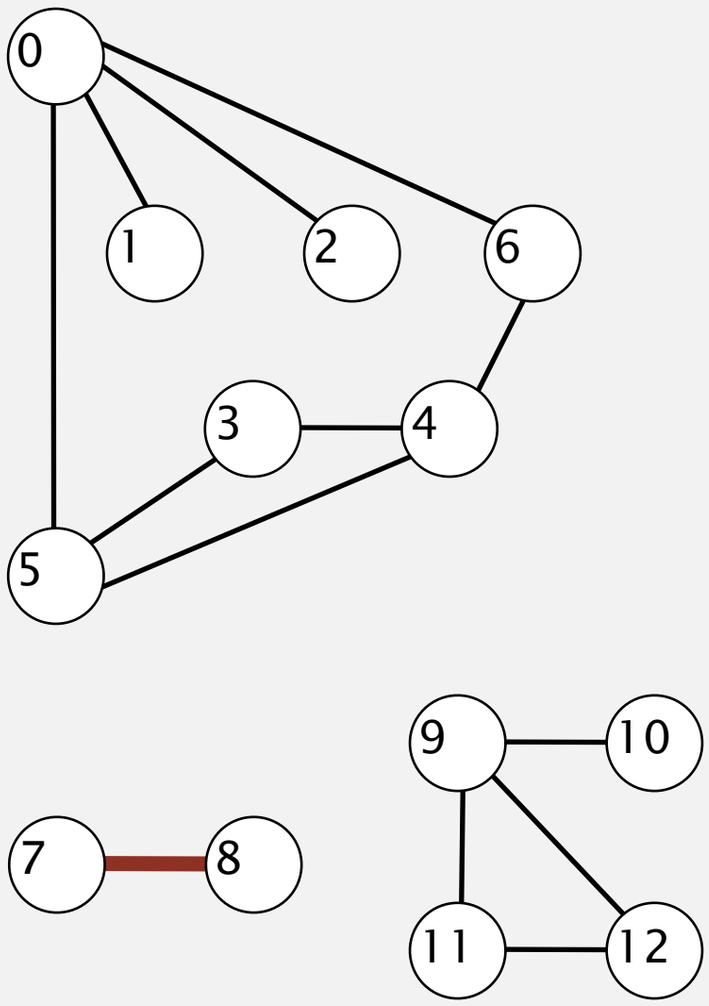
Graph Representation

We have the following options for the graph representation:

1. List of Edge
2. Adjacency-Matrix
3. Adjacency-List

Graph representation: list of edges (Edge Lists)

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



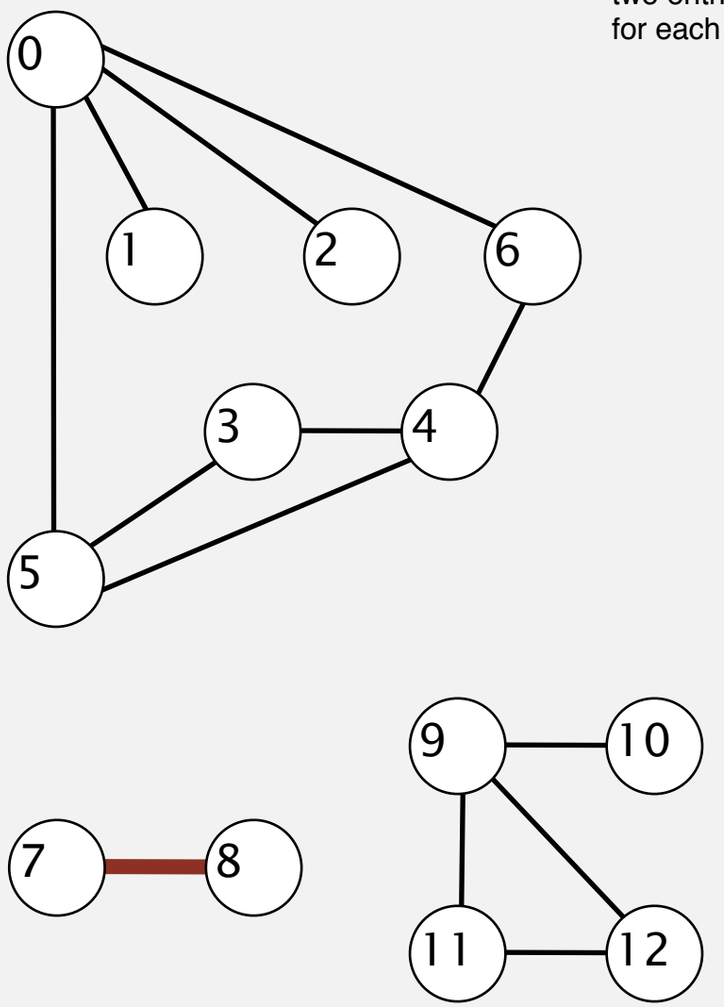

```
0  1
0  2
0  5
0  6
3  4
3  5
4  5
4  6
7  8
9 10
9 11
9 12
11 12
```

Q. How long to iterate over vertices adjacent to v (say 12)?

Graph representation: Adjacency Matrix

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

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



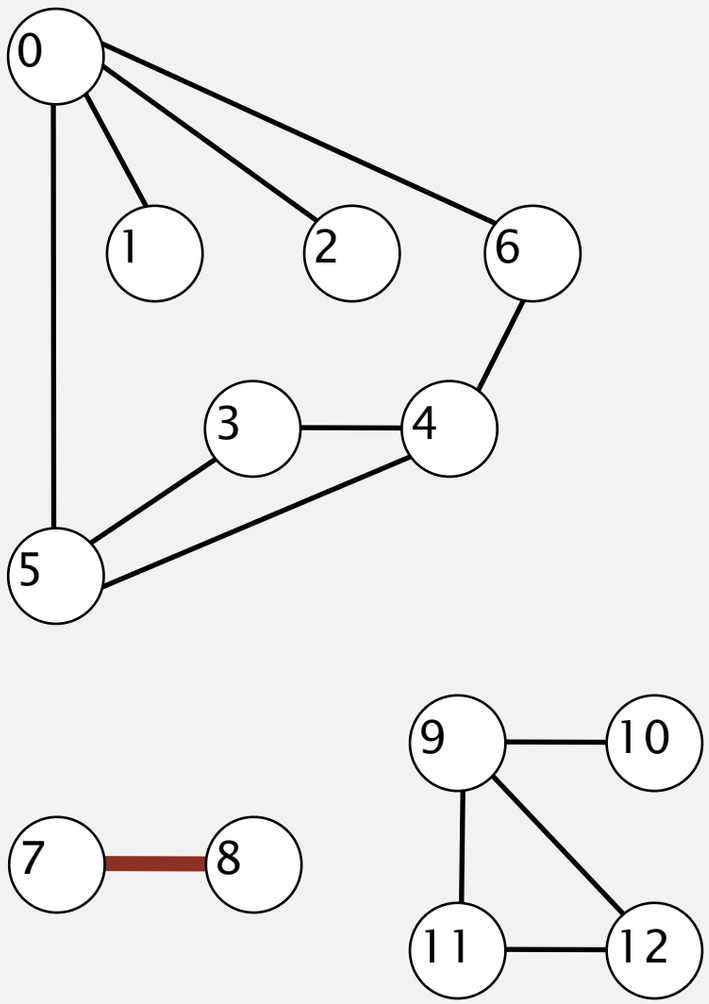
two entries
for each edge

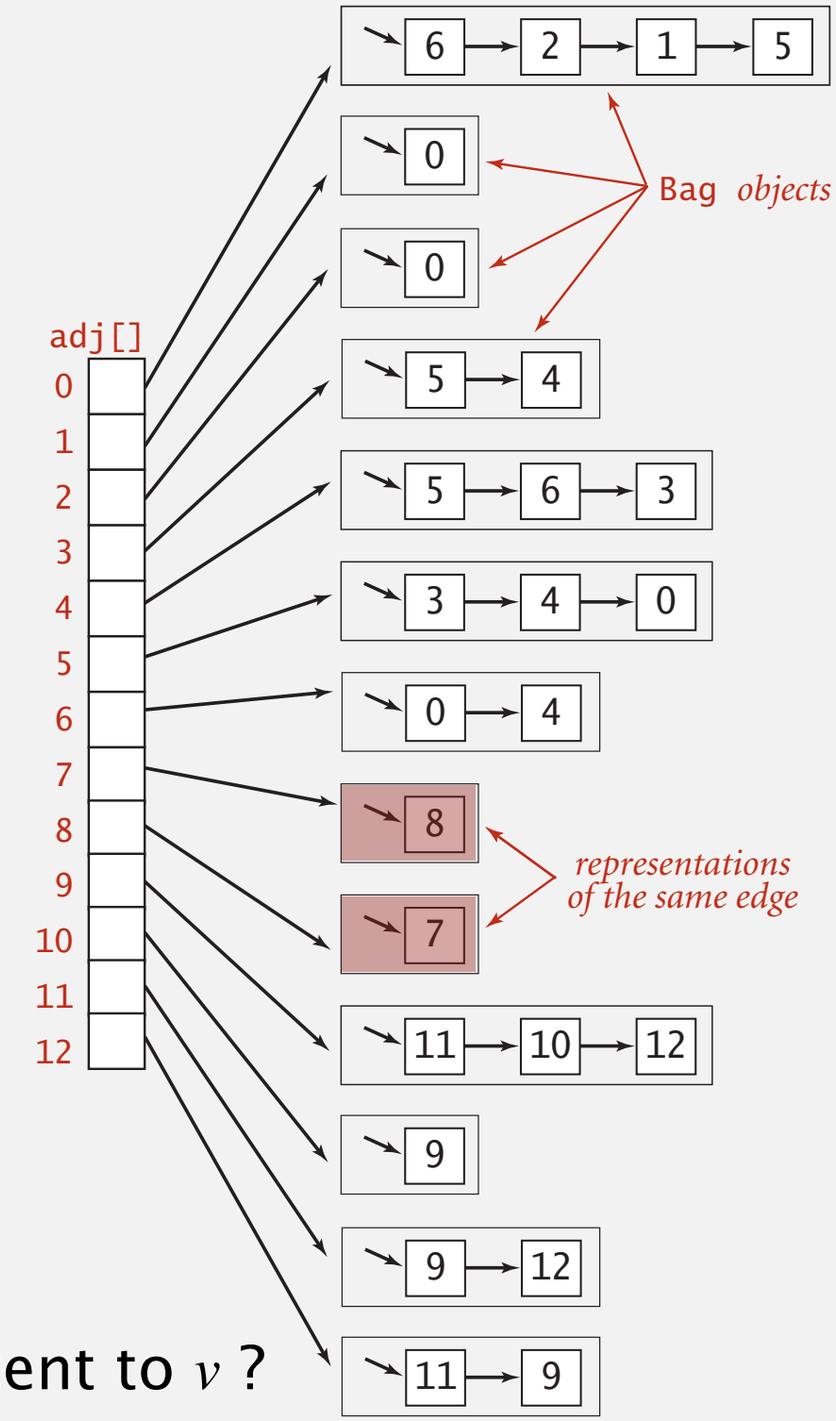
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0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

Q. How long to iterate over vertices adjacent to v ?

Graph representation: Adjacency Lists

Maintain vertex-indexed array of lists.



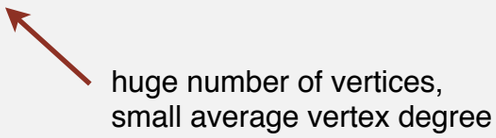


Q. How long to iterate over vertices adjacent to v ?

Graph representations

In practice. Use adjacency-lists representation.

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



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

Graph representations

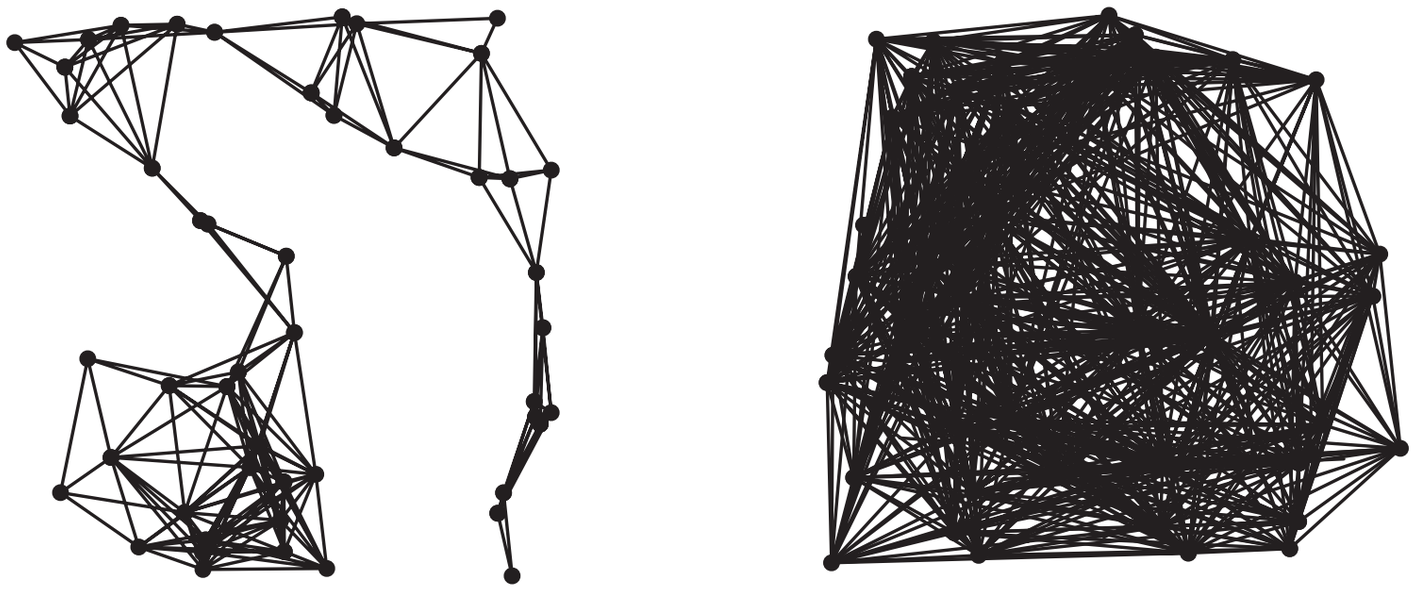
In practice. Use adjacency-lists representation.

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

huge number of vertices,
small average vertex degree

sparse (E = 200)

dense (E = 1000)



Two graphs (V = 50)

Adjacency-list graph representation: Java implementation

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

   public Graph(int V)
   {

   }

   public void addEdge(int v, int w)
   {

   }

   public Iterable<Integer> adj(int v)
   {                    }
}
```

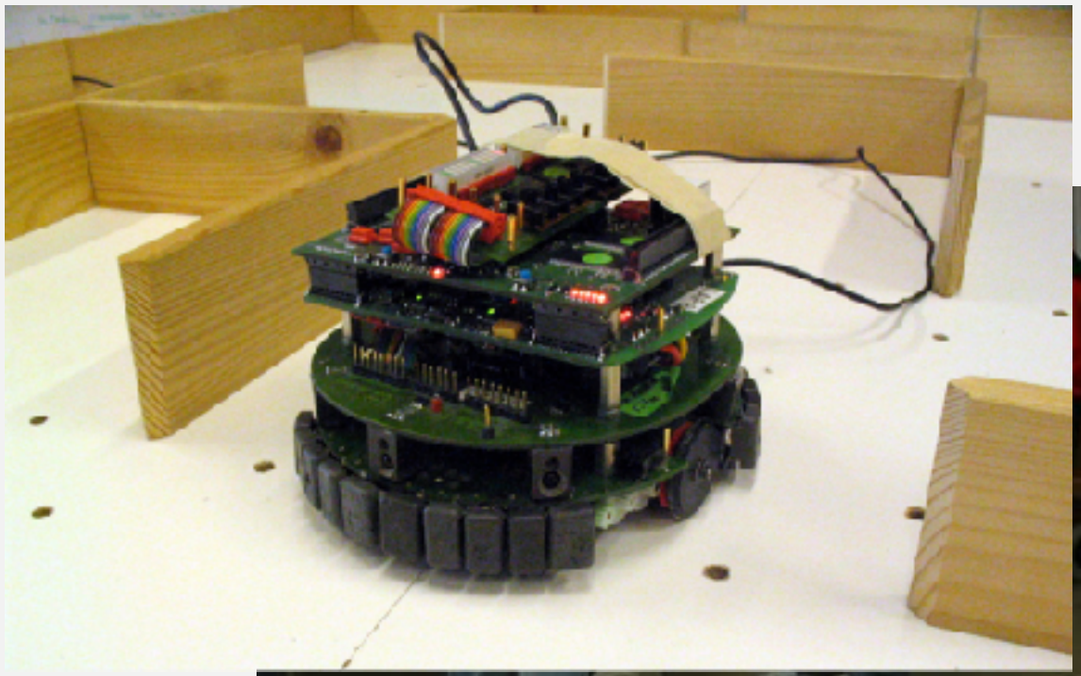
adjacency lists
(using Bag data type)

create empty graph
with V vertices

add edge v-w
(parallel edges and
self-loops allowed)

iterator for vertices adjacent to v

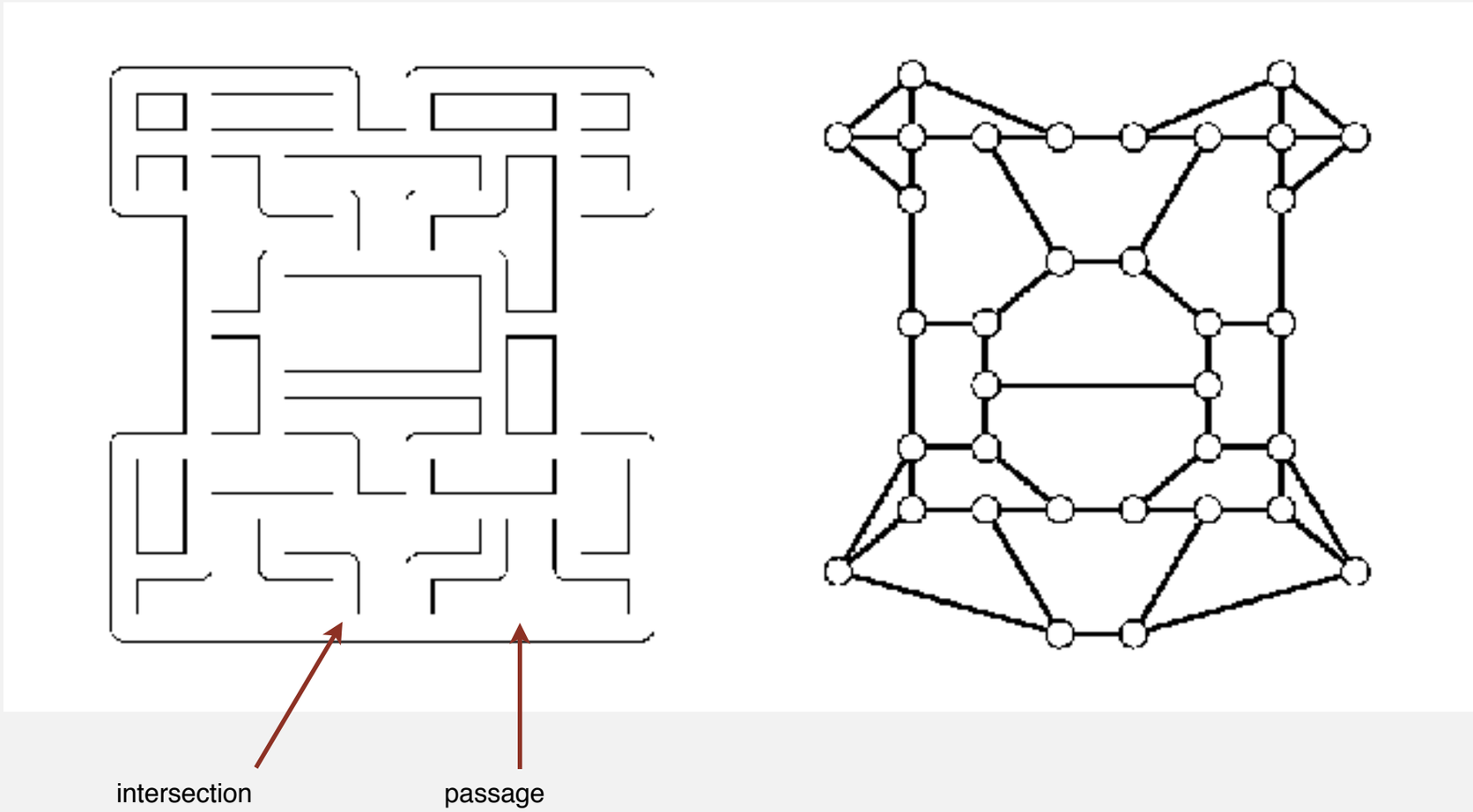
Robot Maze Competition



Maze exploration

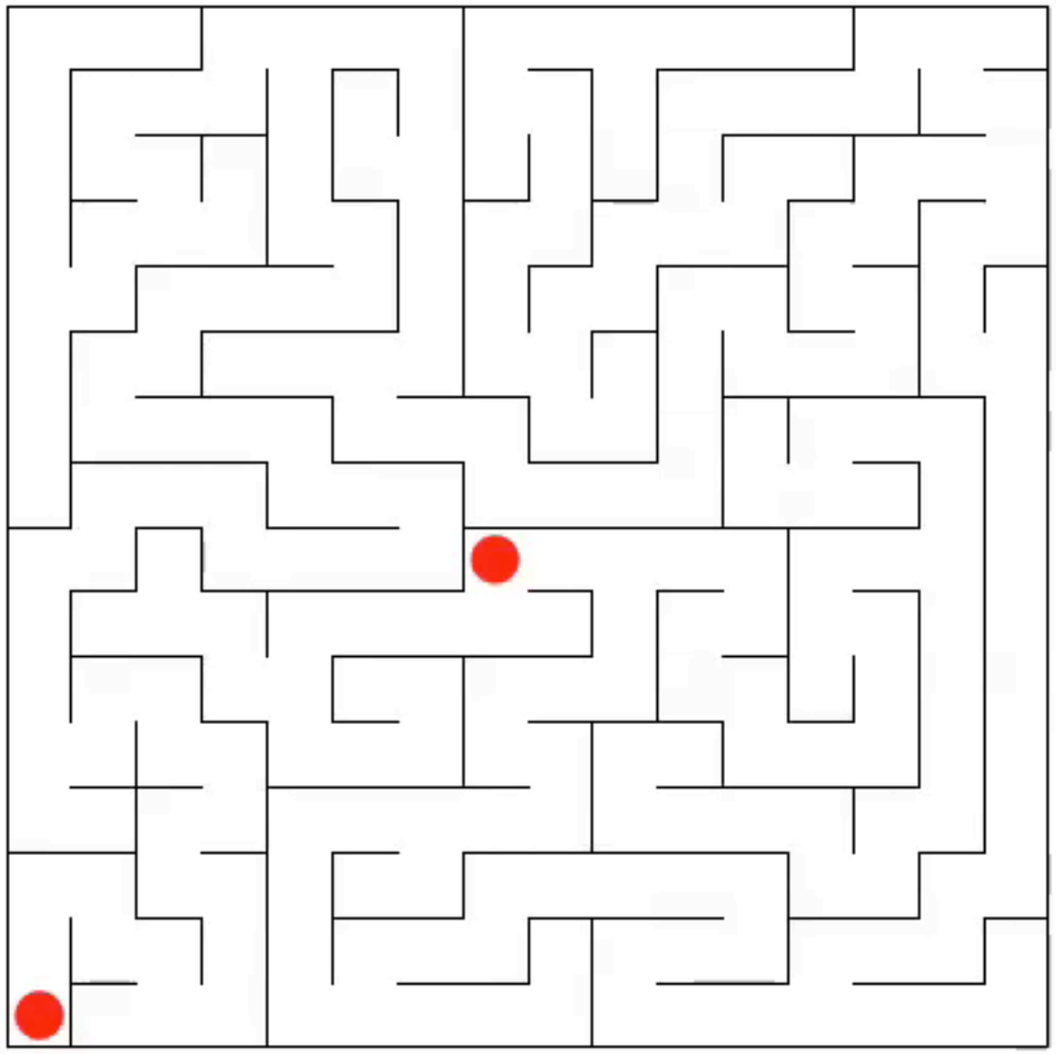
Maze graph.

- Vertex = intersection.
- Edge = passage.

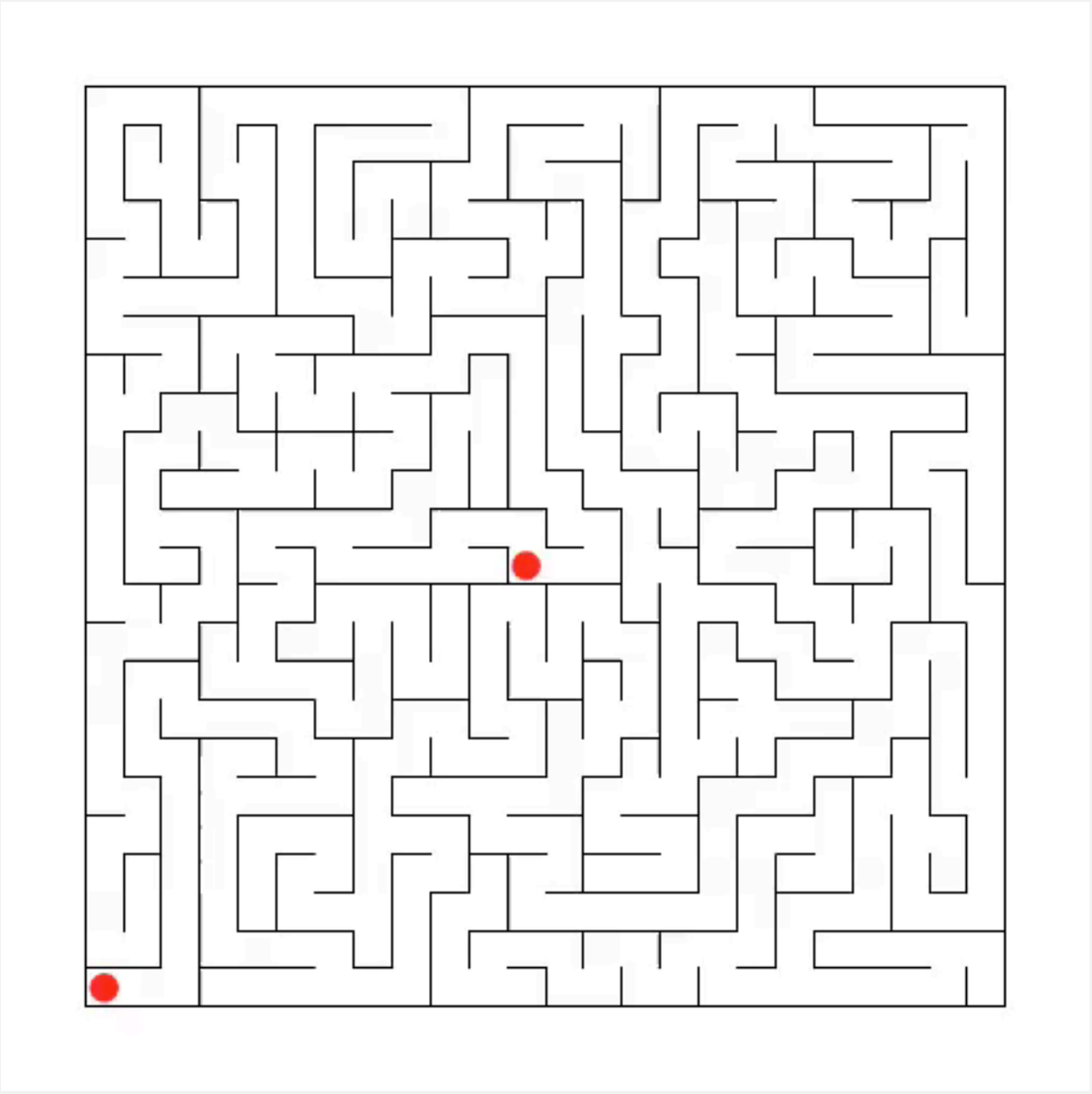


Goal. Explore every intersection in the maze.

Maze exploration: easy



Maze exploration: medium



Single Source Path Discovery Problem

Design pattern. Decouple graph data type from graph processing.

- Create a `Graph` object.
- Pass the `Graph` to a graph-processing routine.
- Query the graph-processing routine for information.

	`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*

UNDIRECTED GRAPHS

- *Introduction*
- *Graph API*
- *Depth-first Search*
- *Breadth-first Search*
- *Connected Components*

Depth-first search

Goal. Systematically traverse 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 challenge. How to implement?

Depth-first search: Java implementation

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

   public DepthFirstPaths(Graph G, int s)
   {



   }

   private void dfs(Graph G, int v)
   {



   }
```

marked[v] = true if v connected to s

edgeTo[v] = previous vertex on path from s to v

initialize data structures

find vertices connected to s

recursive DFS does the work

Depth-first search: data structures

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

Data structures.

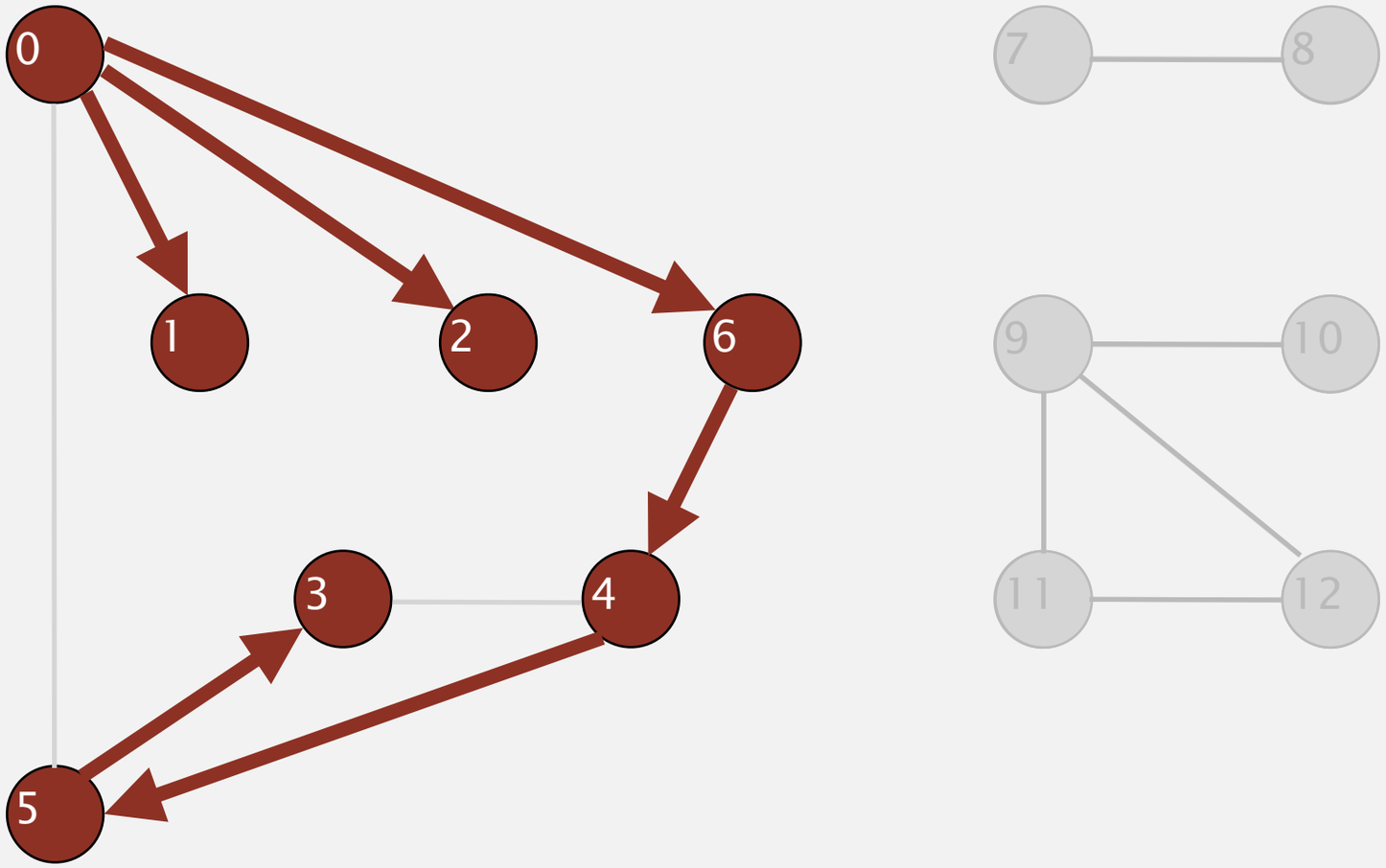
- Boolean array `marked[]` to mark visited vertices.
- Integer array `edgeTo[]` to keep track of paths.

  `(edgeTo[w] == v)` means that edge `v-w` taken to visit `w` for first time
- Function-call stack for recursion.

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

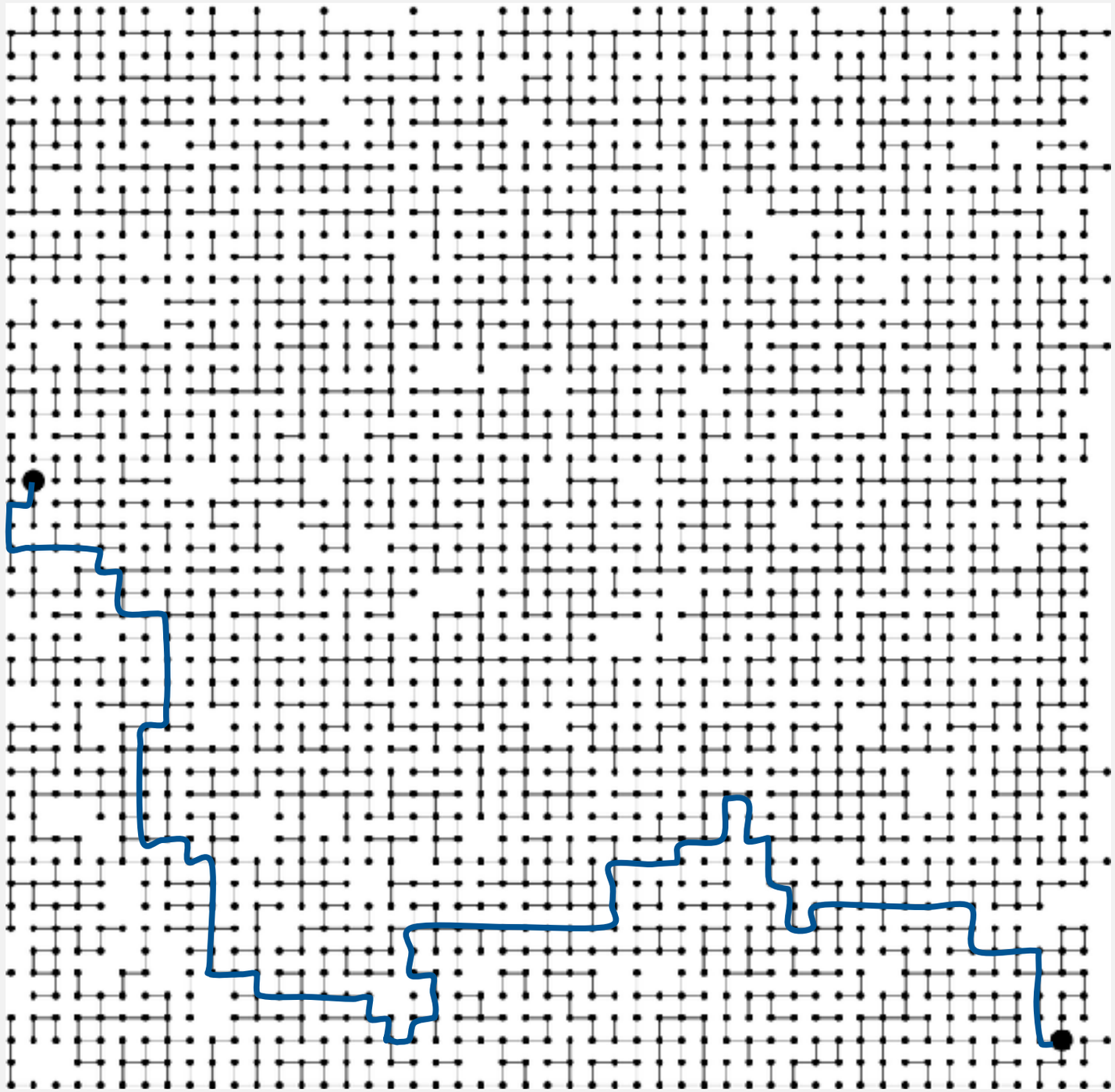


vertices reachable from 0

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

Pathfinding in graphs

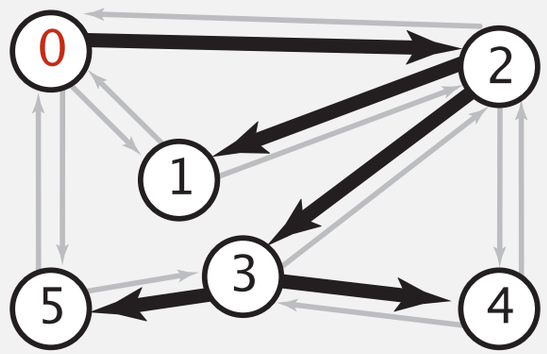
Goal. Does there exist a path from s to t? If yes, find any such path.



Report a path from s to a given v

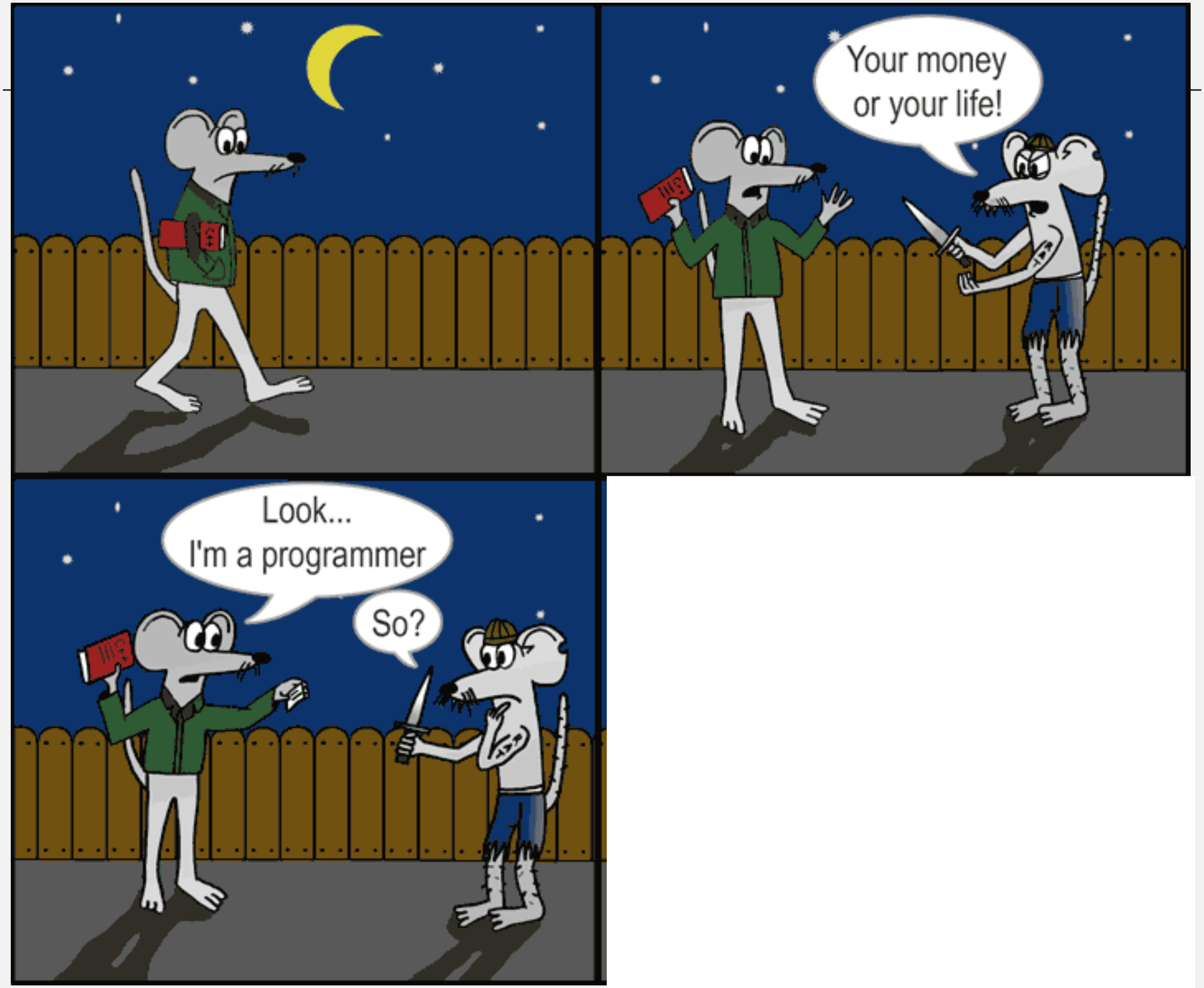
public class Paths		
	Paths(Graph G, int s)	*find paths in G from source s*
boolean	hasPathTo(int v)	*is there a path from s to v?*
Iterable<Integer>	pathTo(int v)	*path from s to v; null if no such path*

```
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;
}
```



edgeTo[]

0	
1	2
2	0
3	2
4	3
5	3


Your money or your life!
Look... I'm a programmer
So?

Undirected Graphs

- *Introduction*
- *Graph API*
- *Depth-first Search*
- *Breadth-first Search*
- *Connected Components*

Graph Problem

The most fundamental graph problem

1. is s and t connected ?
2. if yes, let me know a path from s to t
3. Furthermore, let me know the shortest path from s to t

Breadth-first search

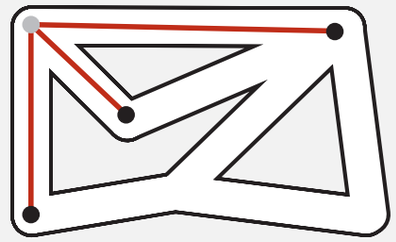
Repeat until queue is empty:

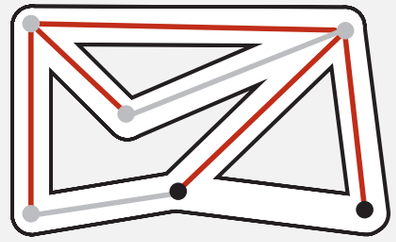
- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.

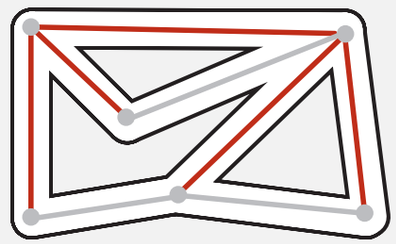
•

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.







Breadth-first search: Java implementation

```
public class BreadthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private int[] distTo;

   …

   private void bfs(Graph G, int s) {
      Queue<Integer> q = new Queue<Integer>();
      q.enqueue(s);
      marked[s] = true;
      distTo[s] = 0;

      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;
               distTo[w] = distTo[v] + 1;
            }
         }
      }
   }
}
```

initialize FIFO queue of vertices to explore

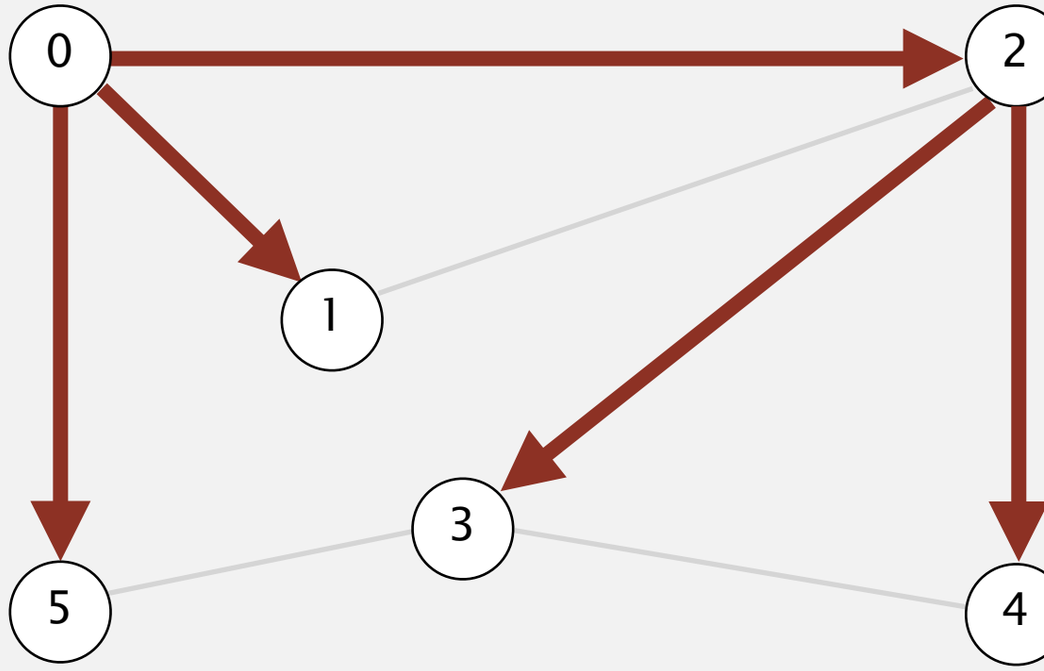
found new vertex w via edge v-w

Report a path from s to a given v

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*

```
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;
}
```

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

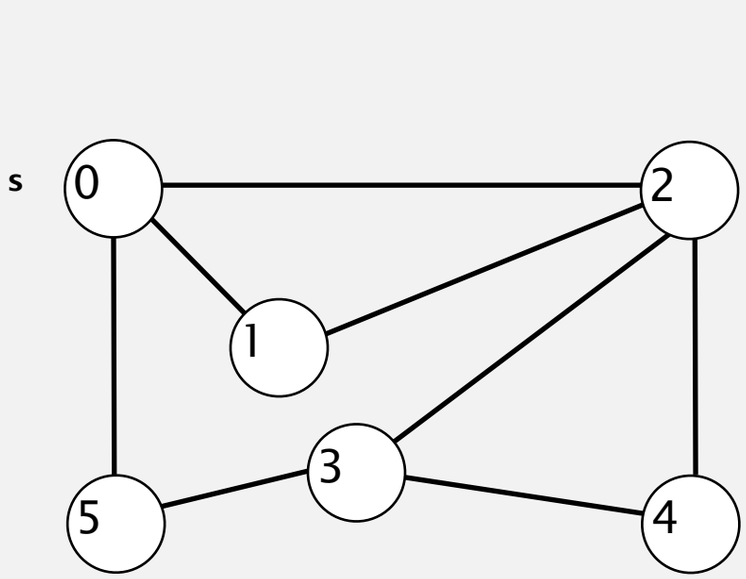


Breadth-first search properties

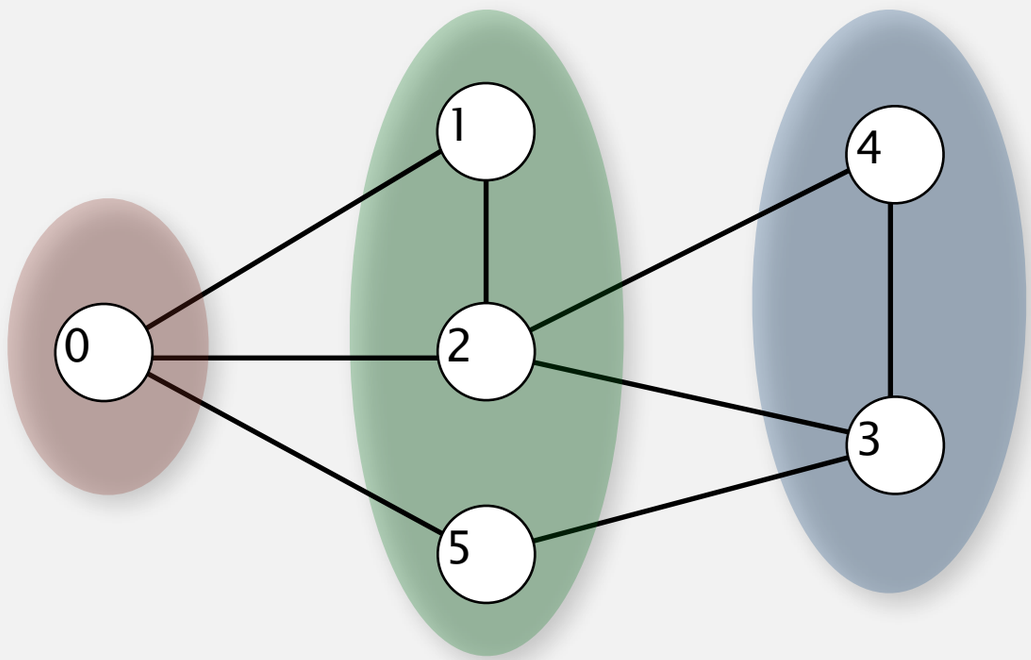
Q. In which order does BFS examine vertices?

A. Increasing distance (number of edges) from s.

Proposition. In any connected graph G, BFS computes shortest paths from s to all other vertices in time proportional to $E + V$.



graph G



dist = 0 dist = 1 dist = 2

Undirected Graphs

- *Introduction*
- *Graph API*
- *Depth-first Search*
- *Breadth-first Search*
- *Connected Components*

Connectivity queries

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

Goal. Preprocess graph to answer queries of the form *is v connected to w?* in constant time.

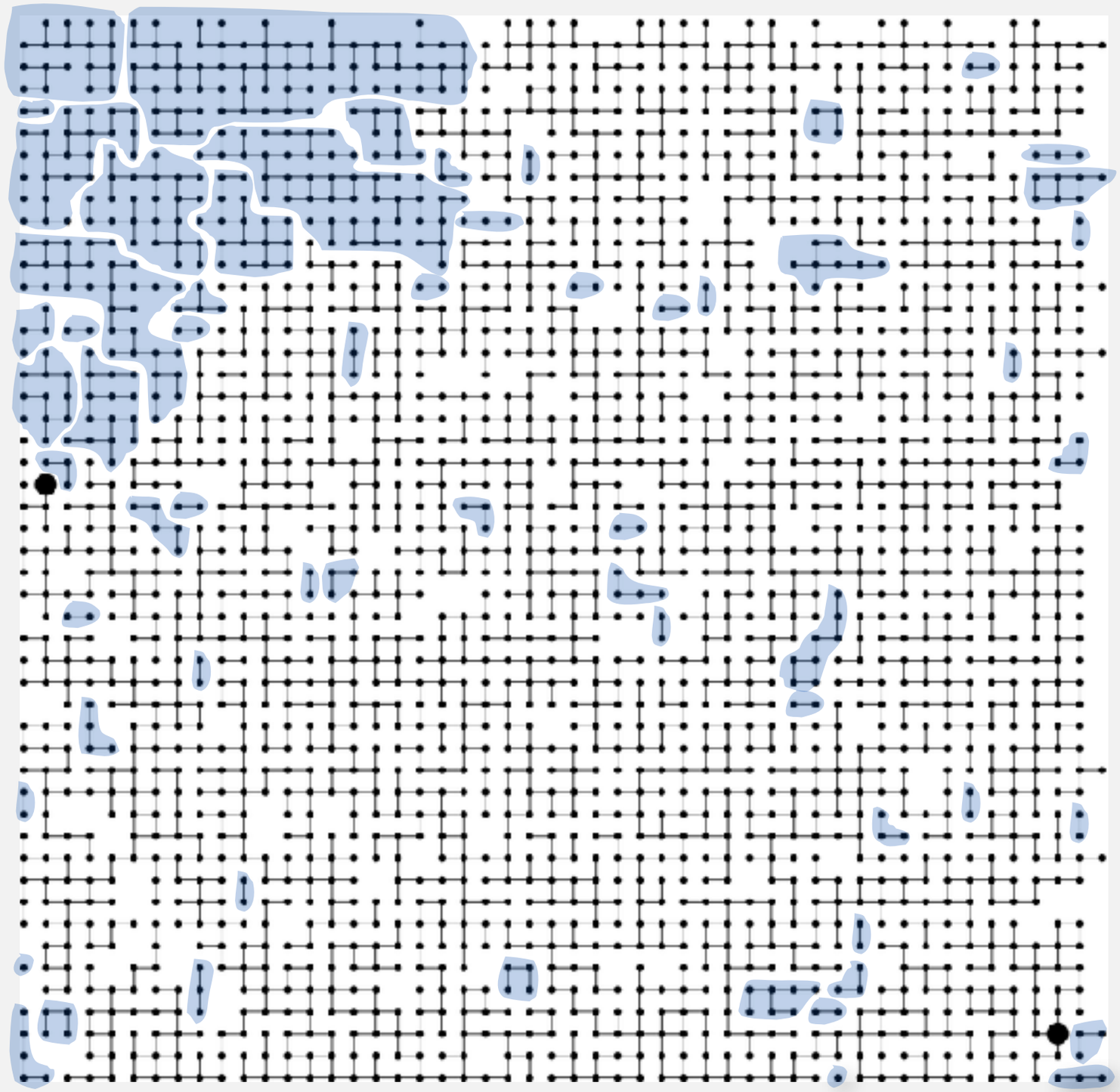
public class CC		
	CC(Graph G)	*find connected components in G*
boolean	connected(int v, int w)	*are v and w connected?*
int	count()	*number of connected components*
int	id(int v)	*component identifier for v (between* `0` *and* `count() - 1`*)*

Union-Find? Not quite.

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

Connected components

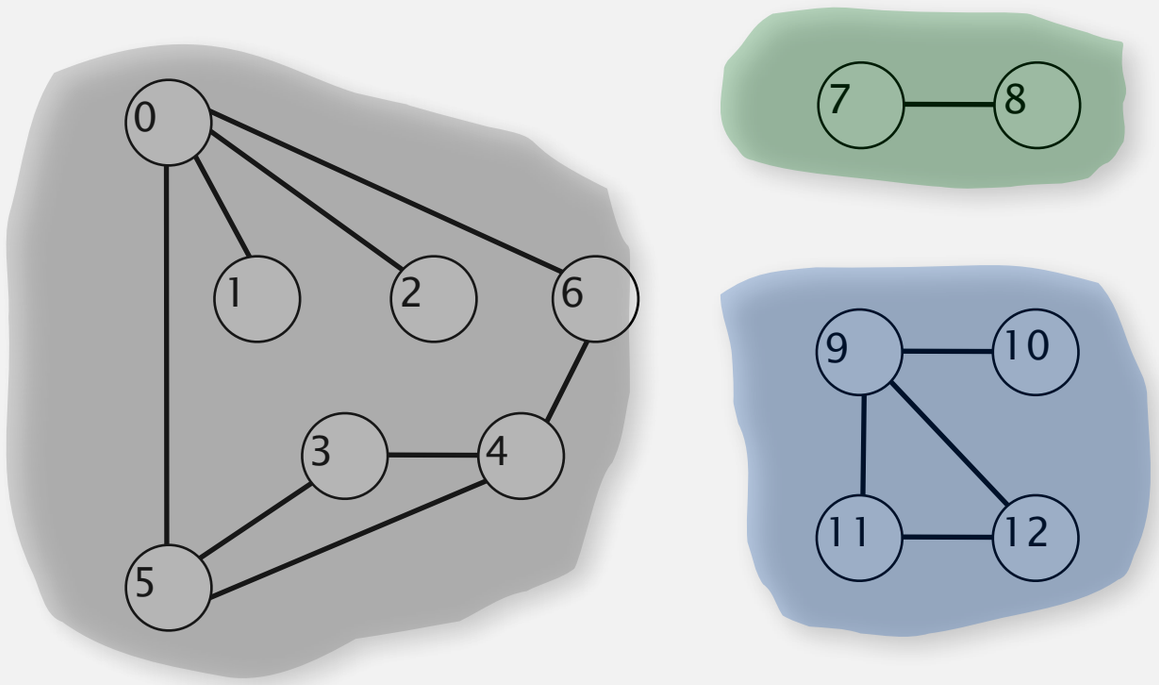
Def. A connected component is a maximal set of connected vertices.



63 connected components

Connected components

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



3 connected components

v	id[]
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.

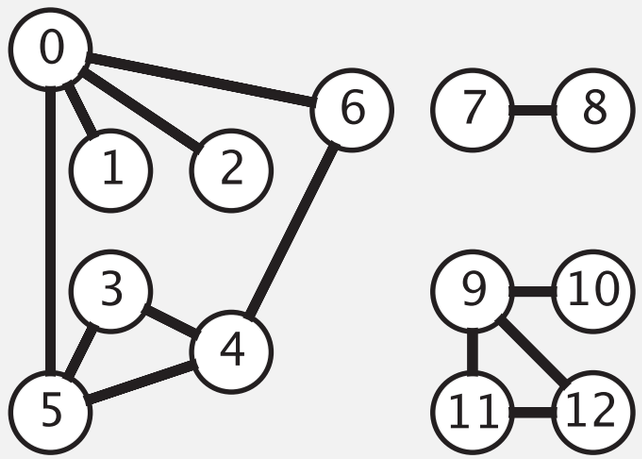
Connected components

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.



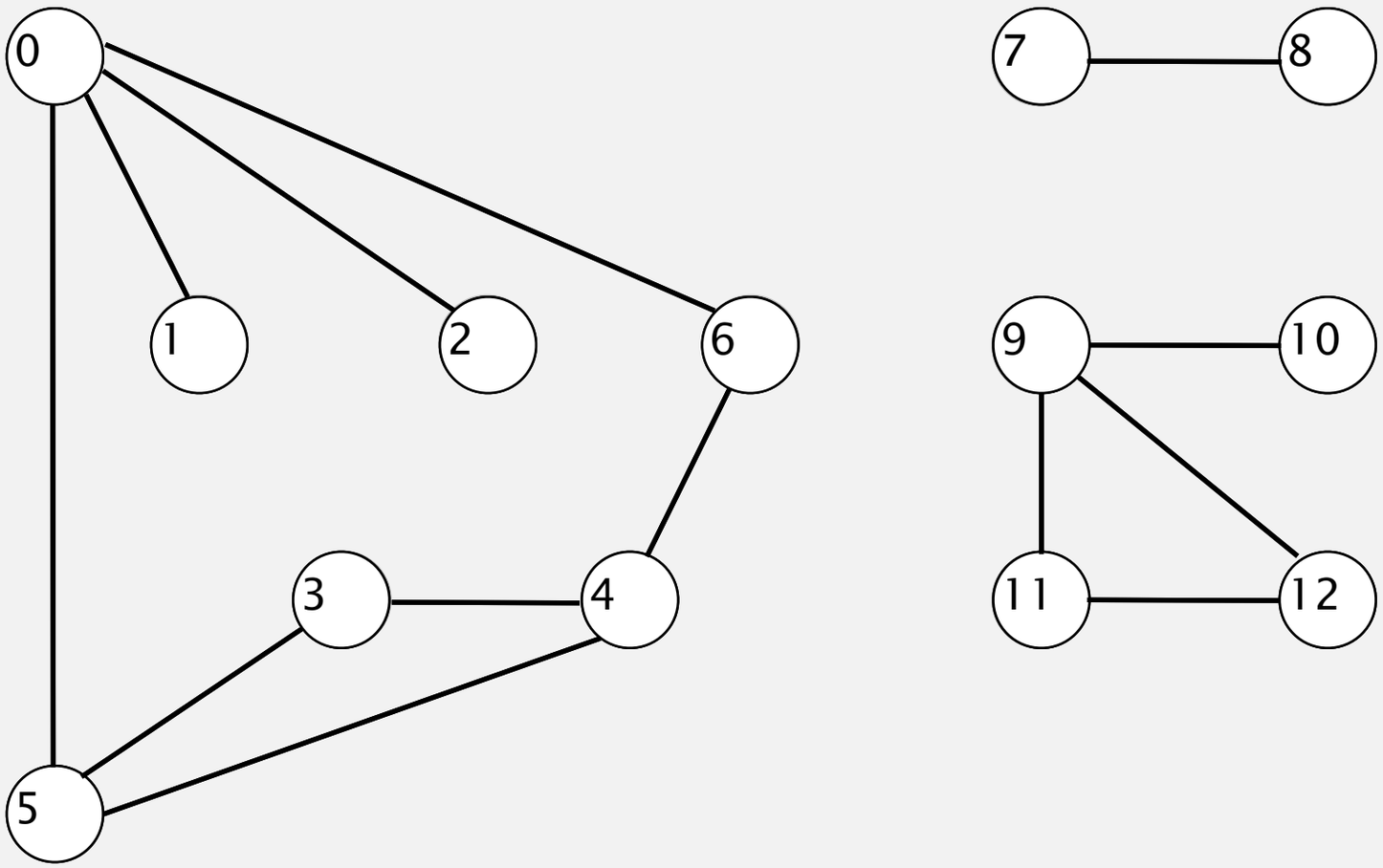
tinyG.txt

```
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
```

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



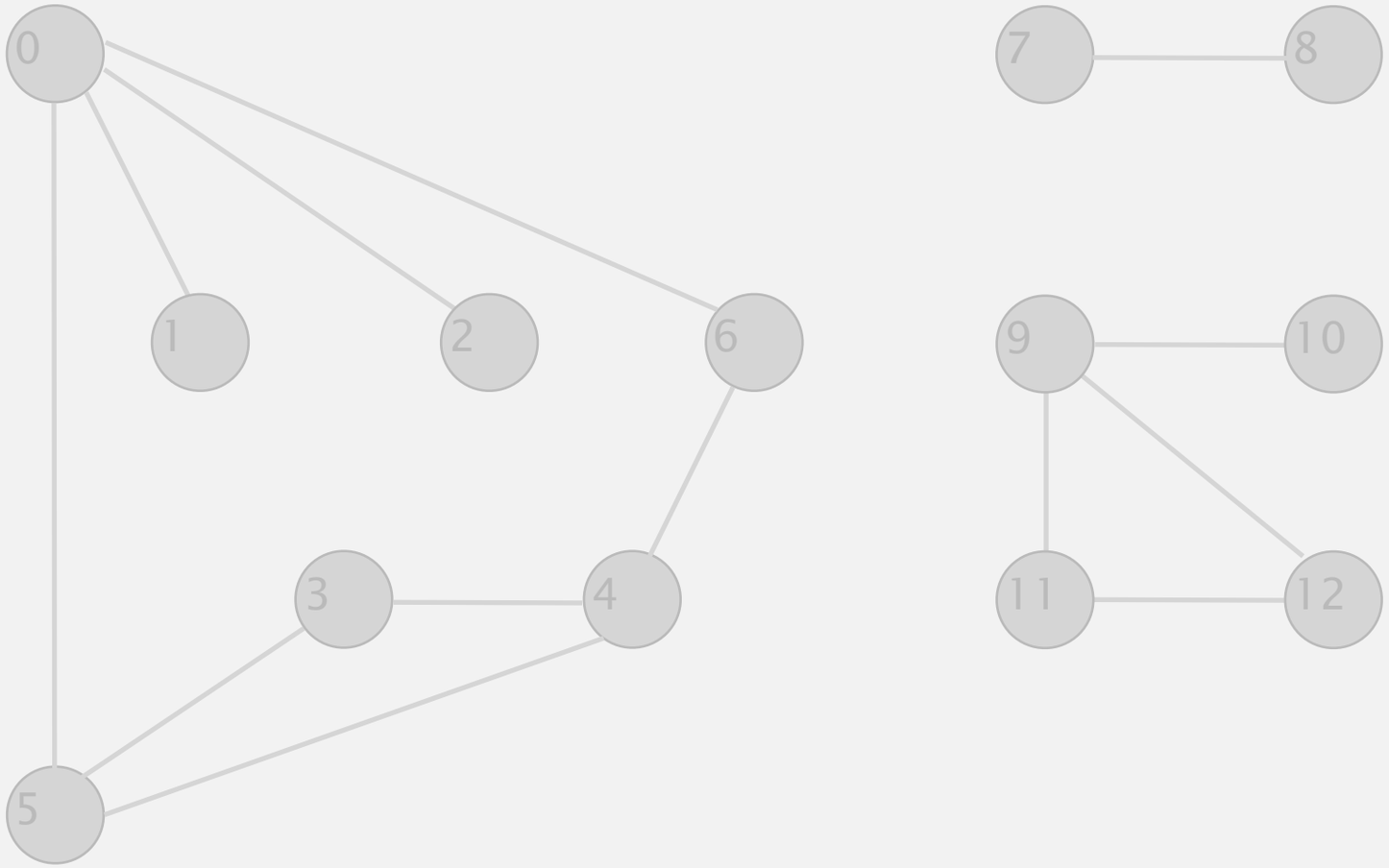
graph G

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

Connected components demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



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

done

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)
   public boolean connected(int v, int w)
   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];  }

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


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

v and w connected iff same id

all vertices discovered in same call of dfs have same id

Paths in graphs: union-find vs. DFS

Goal. Does there exist a path from s to t?

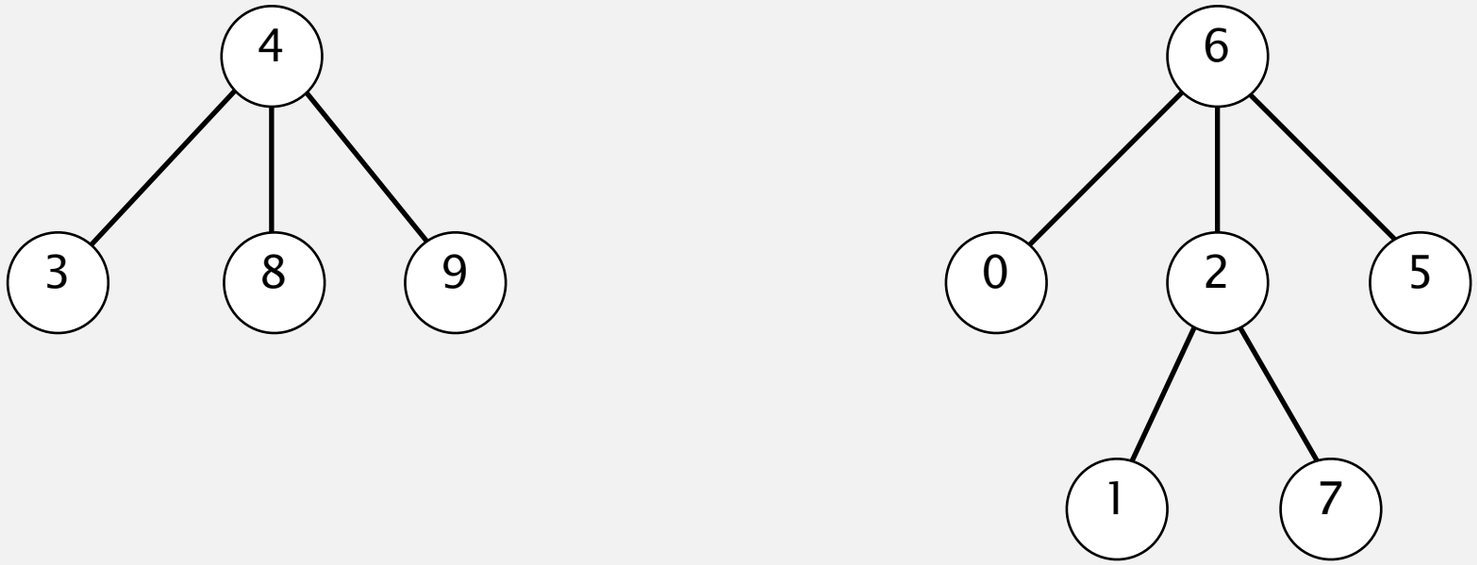
method	preprocessing time	query time	space
weighted union-find	E log* V	log* V	V
DFS	E	1	E + V

Union-find. Can intermix connected queries and edge insertions.

Depth-first search. Constant time per query.

Weighted quick-union demo (Just like to recall this)

union(7, 3)

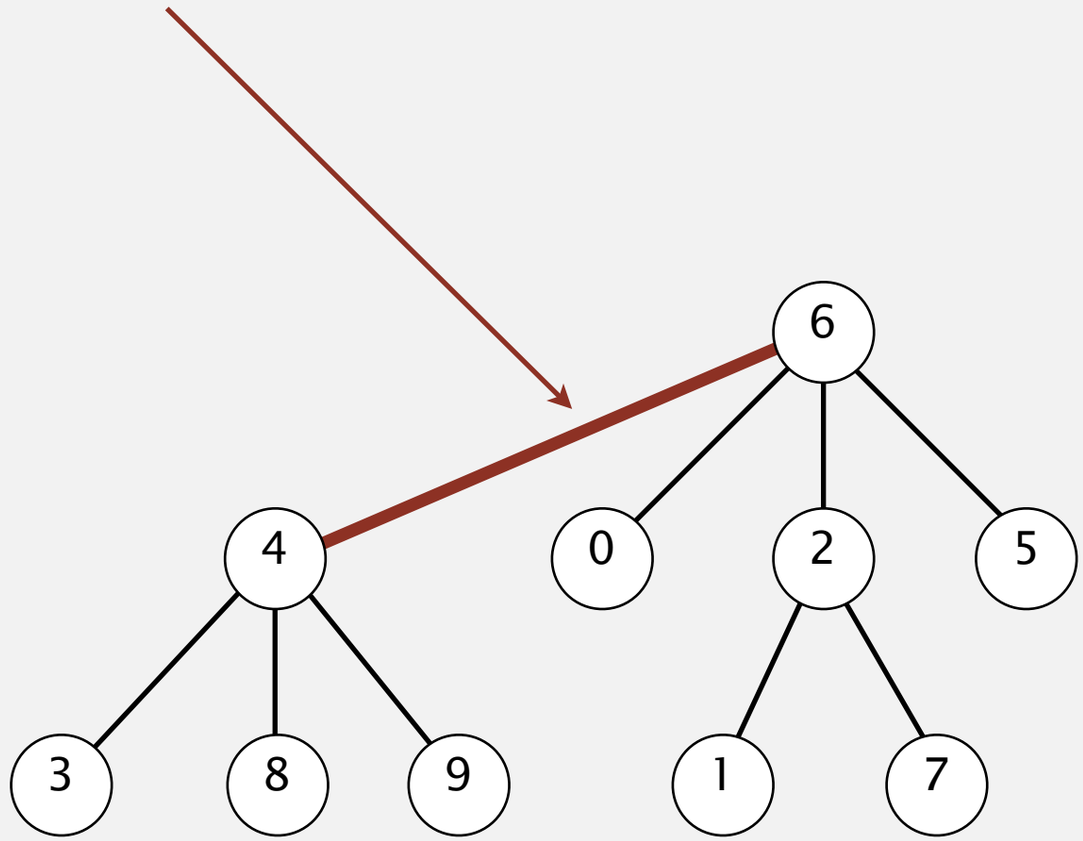


	0	1	2	3	4	5	6	7	8	9
id[]	6	2	6	4	4	6	6	2	4	4

Weighted quick-union demo (Just like to recall this)

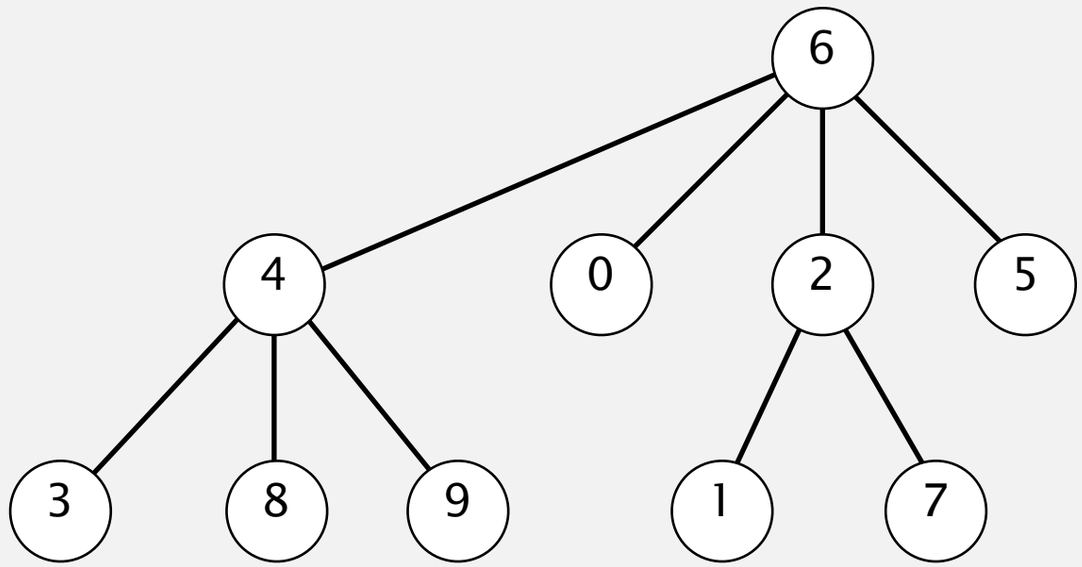
union(7, 3)

weighting: make 4 point to 6 (instead of 6 to 4)



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

Weighted quick-union demo (Just like to recall this)



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

Additional Problem?

Cycle detection. *Is a given graph acylic?*

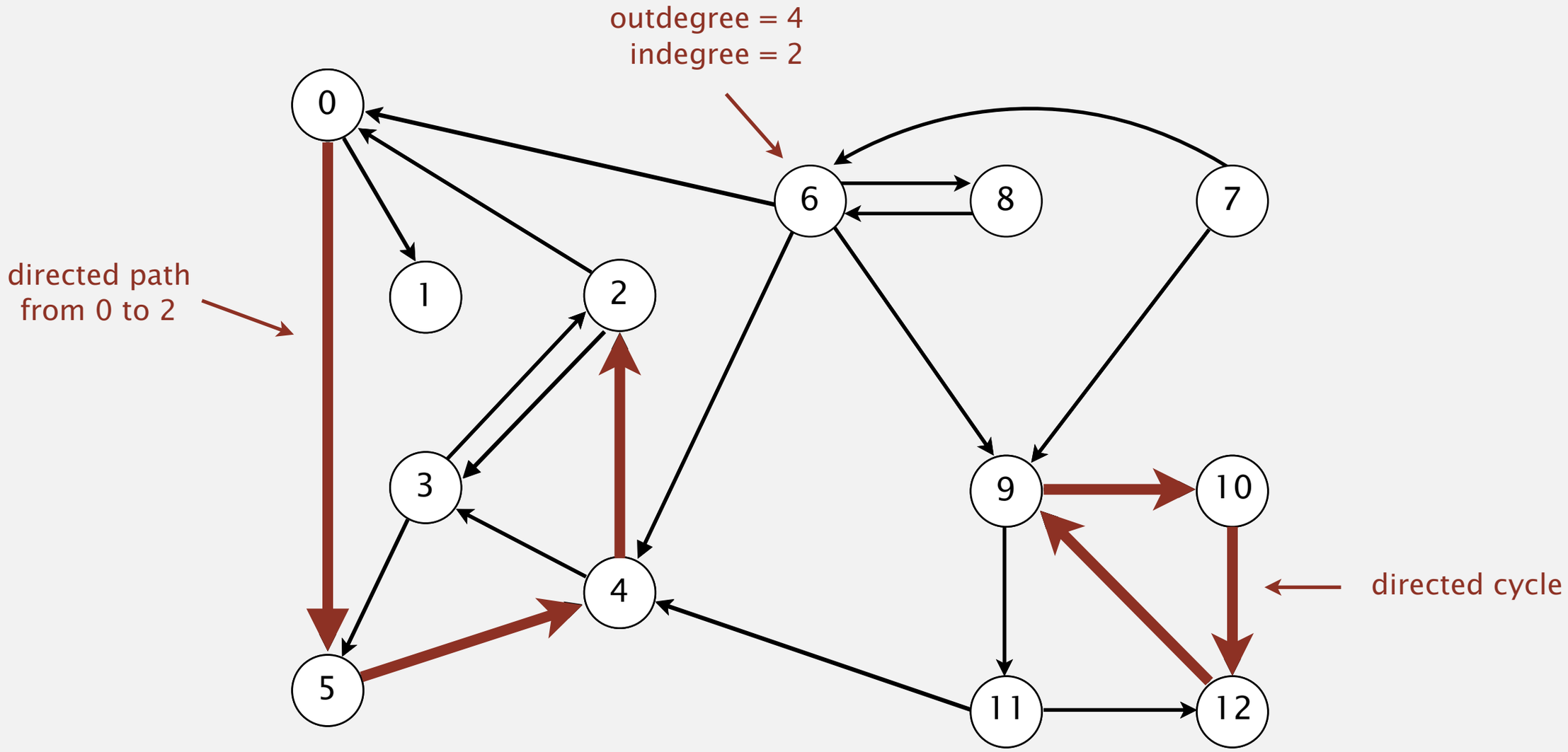
```
public class Cycle
{
   private boolean[] marked;
   private boolean hasCycle;

   public Cycle(Graph G)
   {
      marked = new boolean[G.V()];
      for (int s = 0; s < G.V(); s++)
         if (!marked[s])
            dfs(G, s, s);
   }

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

   public boolean hasCycle()
   {  return hasCycle;  }
}
```

Directed Graphs

- *Introduction*
- *Digraph API*
- *Digraph search*
- *Topological sort*
- *Strong components*

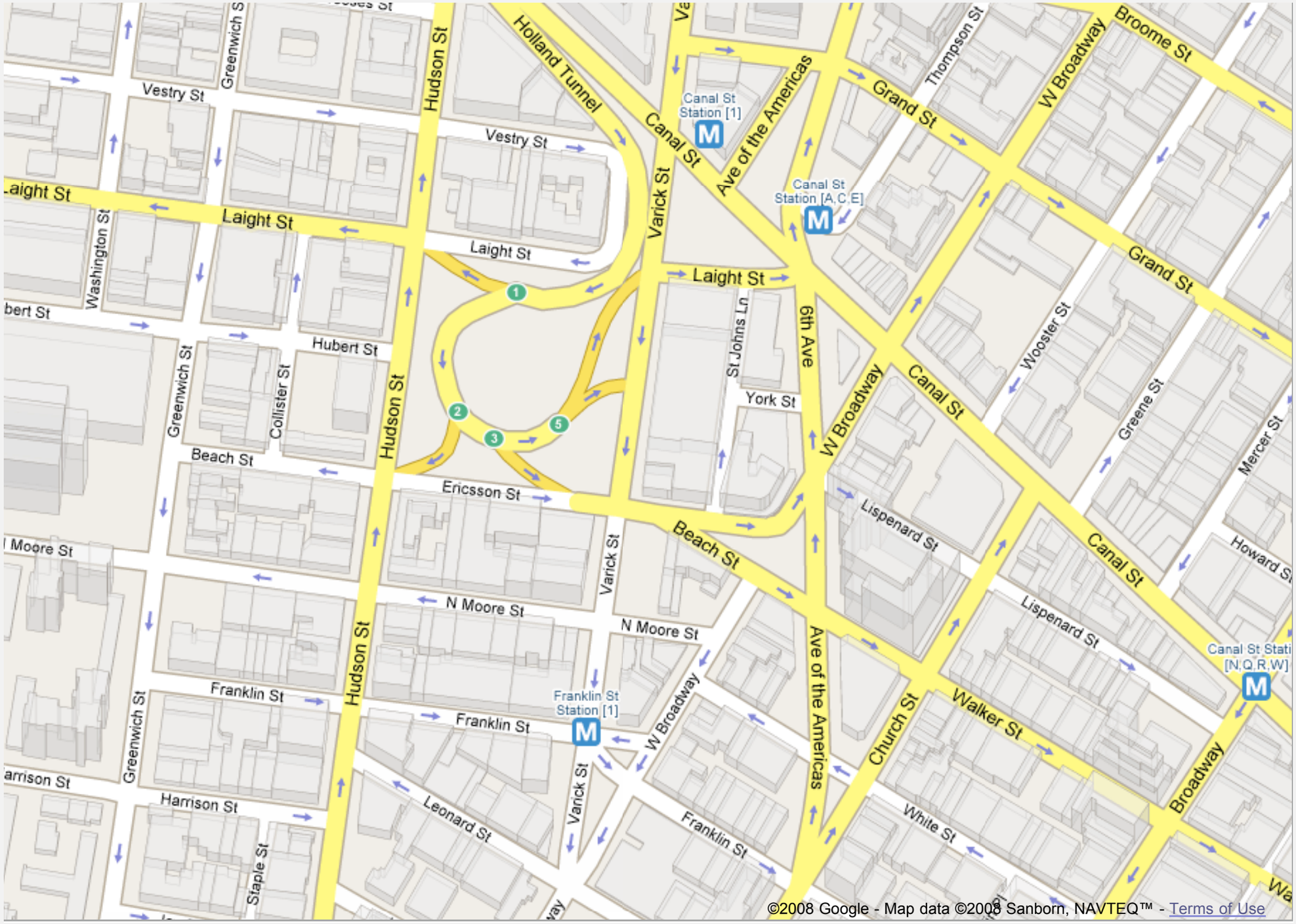
Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.



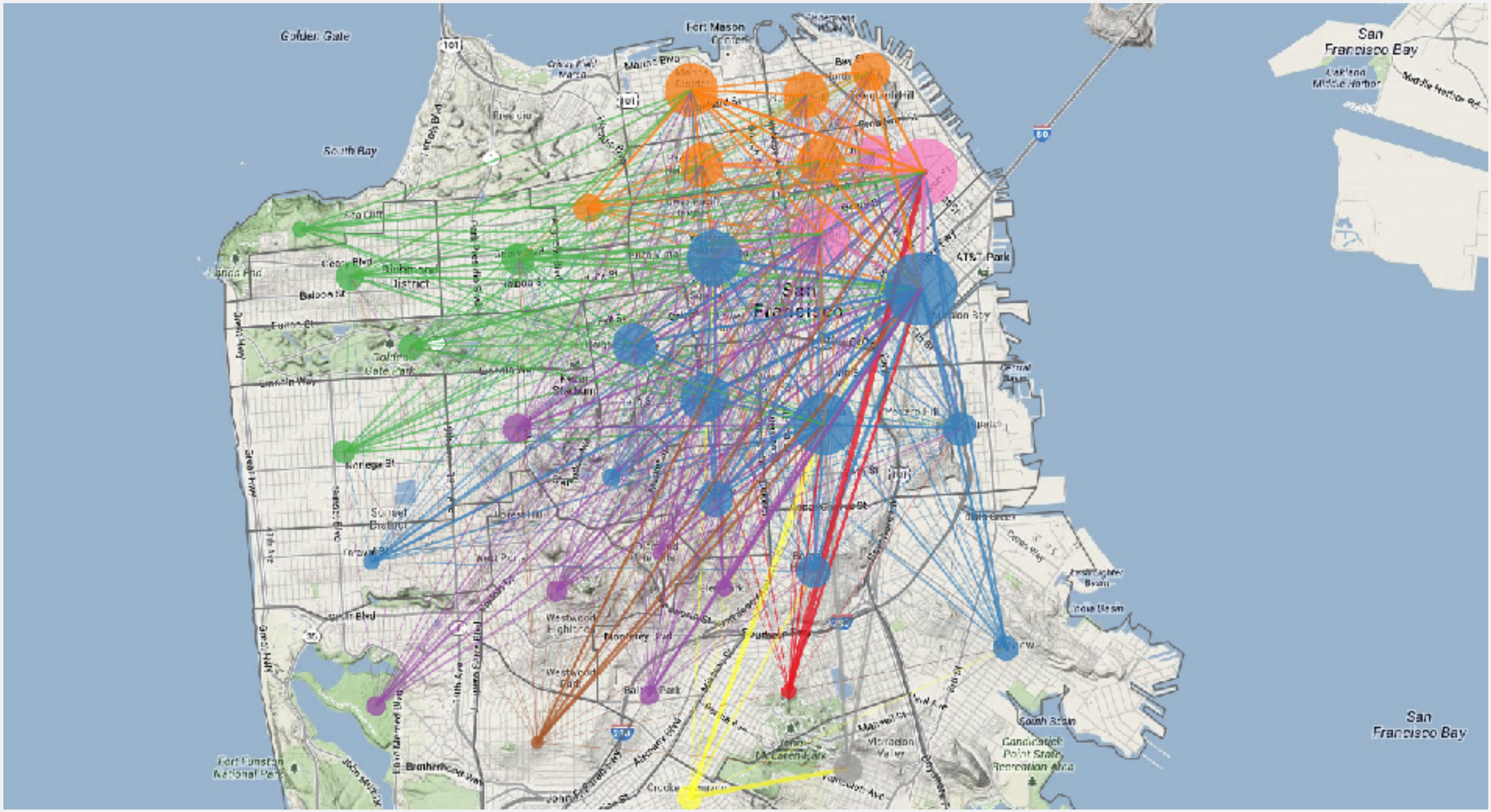
Road network

Vertex = intersection; edge = one-way street.



Uber taxi graph

Vertex = taxi pickup; edge = taxi ride.



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

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

problem	description
s→t path	*Is there a path from s to t ?*
shortest s→t path	*What is the shortest path from s to t ?*
directed cycle	*Is there a directed cycle in the graph ?*
topological sort	*Can the digraph be drawn so that all edges point upwards?*
strong connectivity	*Is there a directed path between all pairs of vertices ?*
PageRank	*What is the importance of a web page ?*

Directed Graphs

- *Introduction*
- *Digraph API*
- *Digraph Search*
- *Topological Sort*
- *Strong Components*

Digraph API

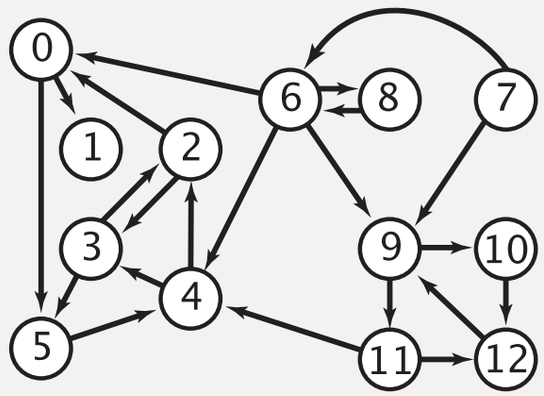
Almost identical to `Graph` API.

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*

Digraph API

tinyDG.txt

```
V → 13
E → 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
```

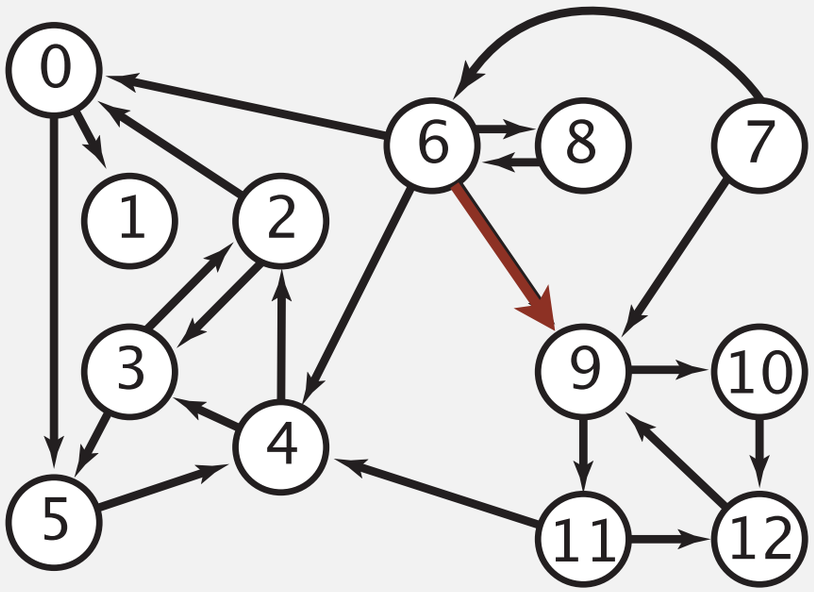
```
In in = new In(args[0]);
Digraph G = new Digraph(in);

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

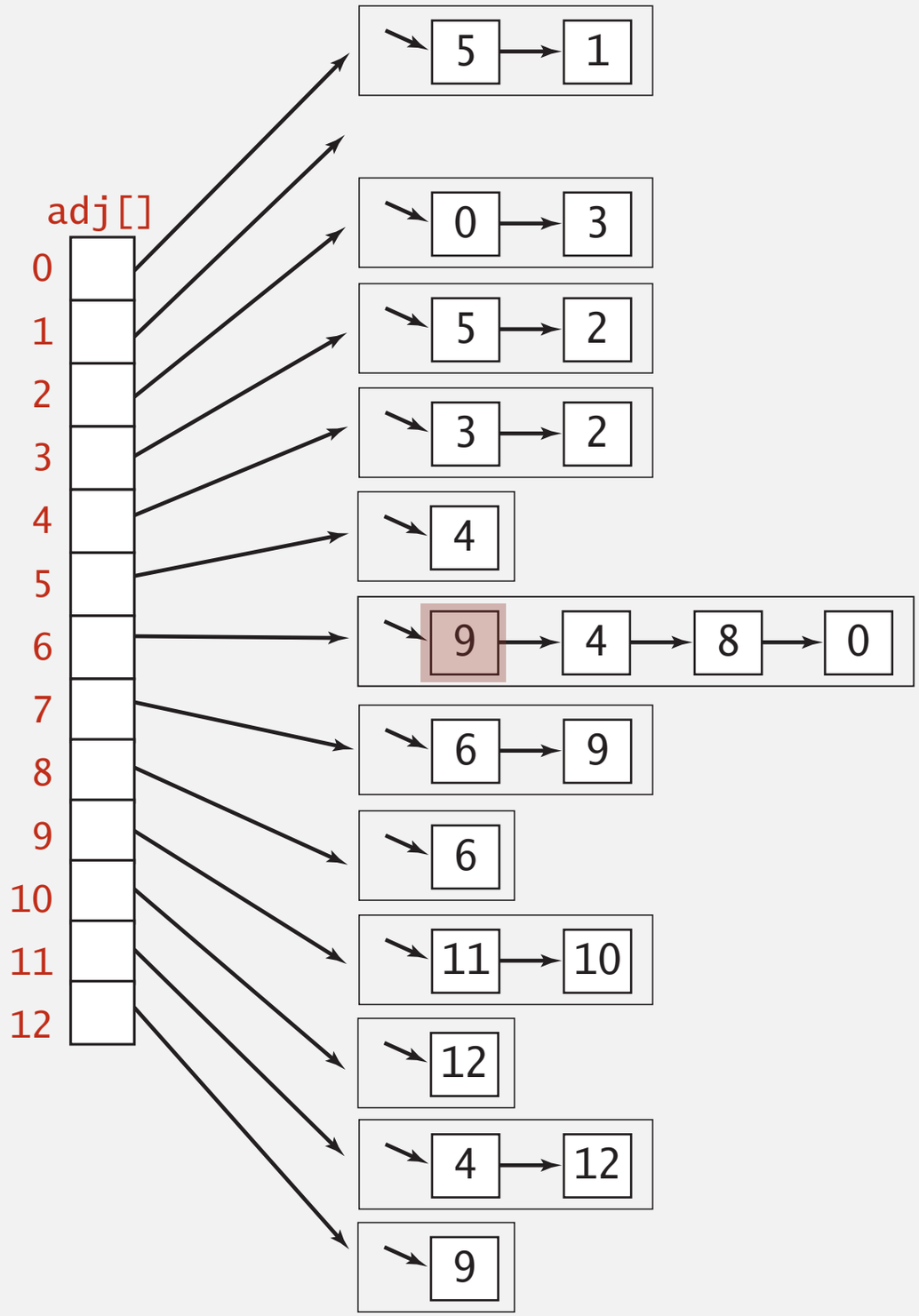
read digraph from input stream

print out each edge (once)

Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.





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?
adjacency matrix	V^2	$1^{\dagger}$	1	V
adjacency lists	$E + V$	1	$outdegree(v)$	$outdegree(v)$

† disallows parallel edges

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

Directed Graphs

- *Introduction*
- *Digraph API*
- *Cycle Detection*
- *Topological Sort*
- *Strong Components*

Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

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

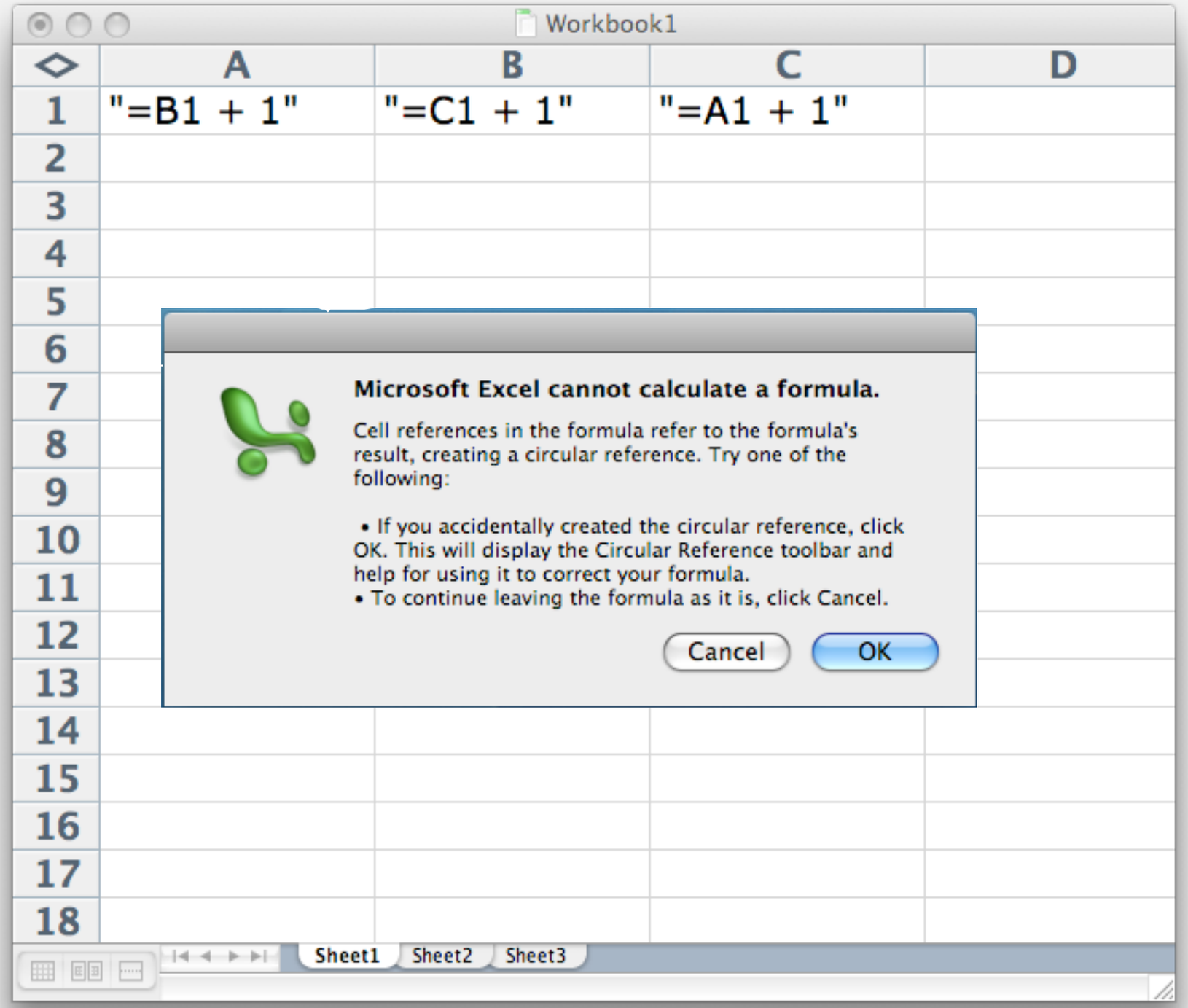
```
public class B extends C
{
   ...
}
```

```
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!)



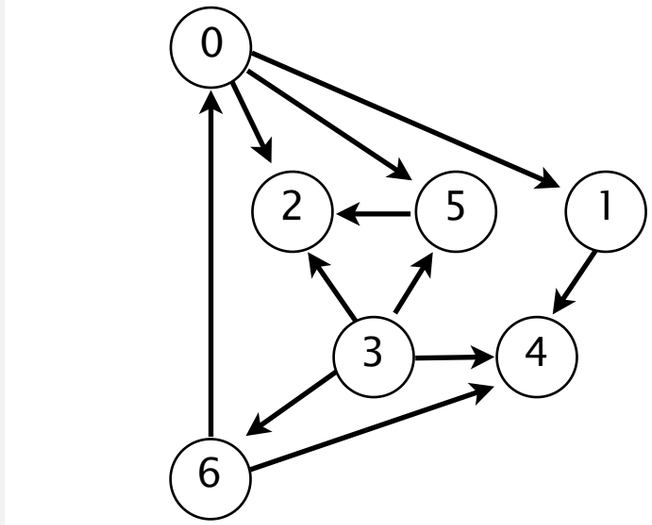
Directed cycle detection application: Precedence scheduling

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

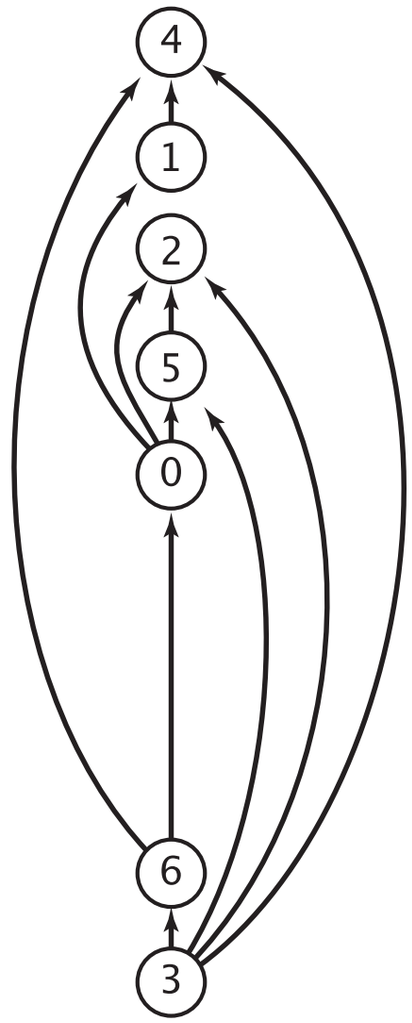
Digraph model. vertex = task; edge = precedence constraint.

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

tasks



precedence constraint graph



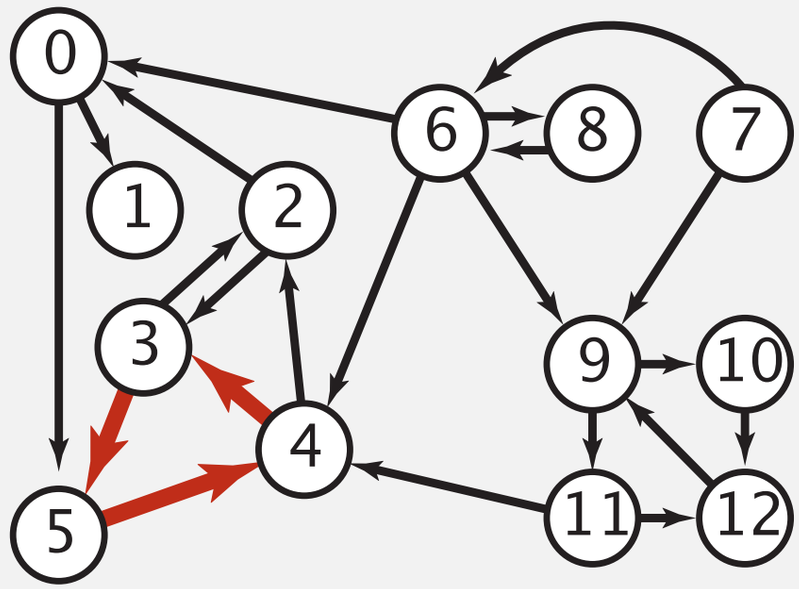
feasible schedule

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.



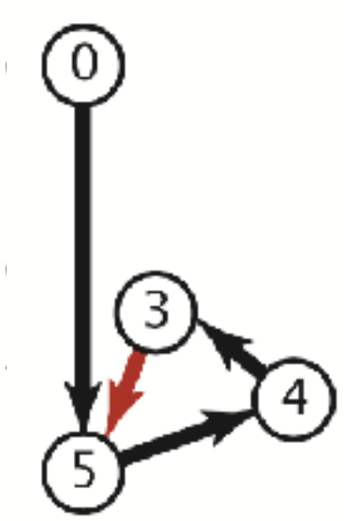
a digraph with a directed cycle

Goal. Given a digraph, find a directed cycle.

Solution. DFS. What else?

DirectedCycle Detection Algorithm

```
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;
   }

   public boolean hasCycle()
   {  return cycle != null;  }

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



edgeTo[]

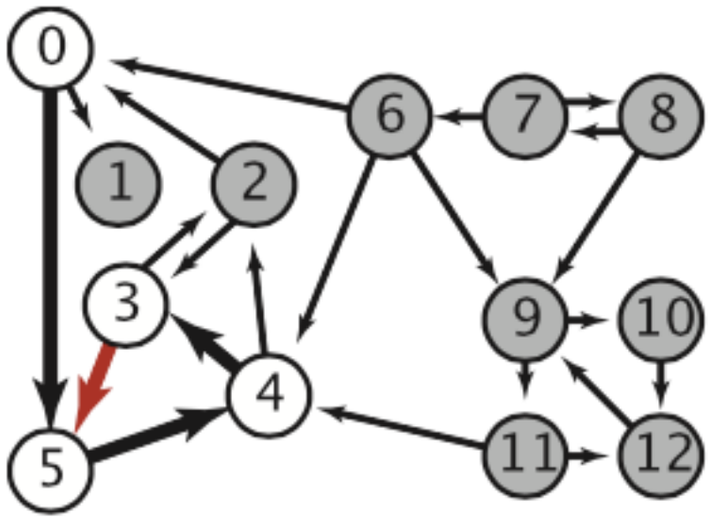
0	
1	
2	
3	4
4	5
5	0

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

DirectedCycle Detection Algorithm

```
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;
   }

   public boolean hasCycle()
   {  return cycle != null;  }

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



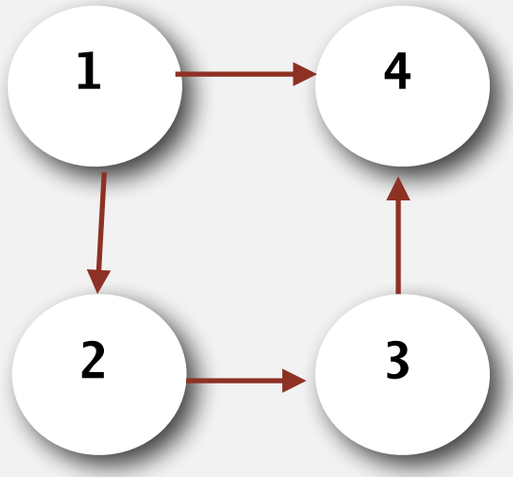

```
dfs(0)
  dfs(5)
    dfs(4)
      dfs(3)
        check 5
```

Trace of cycle computation

marked[]							edgeTo[]							onStack[]						
0	1	2	3	4	5	...	0	1	2	3	4	5	...	0	1	2	3	4	5	...
1	0	0	0	0	0		-	-	-	-	-	0		1	0	0	0	0	0	
1	0	0	0	0	1		-	-	-	-	5	0		1	0	0	0	0	1	
1	0	0	0	1	1		-	-	-	4	5	0		1	0	0	0	1	1	
1	0	0	1	1	1		-	-	-	4	5	0		1	0	0	1	1	1	

directed cycle in a digraph

Why onStack Data Structure ?



Marked[] is inadequate !

Directed cycle detection applications

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

DIRECTED GRAPHS

- *Introduction*
- *Digraph API*
- *Digraph search*
- *Cycle Detection*
- *Topological Sort*
- *Strong Components*

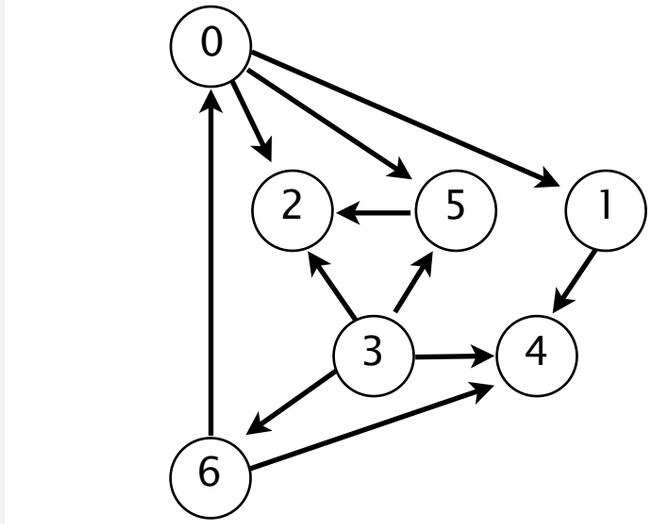
Precedence scheduling

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

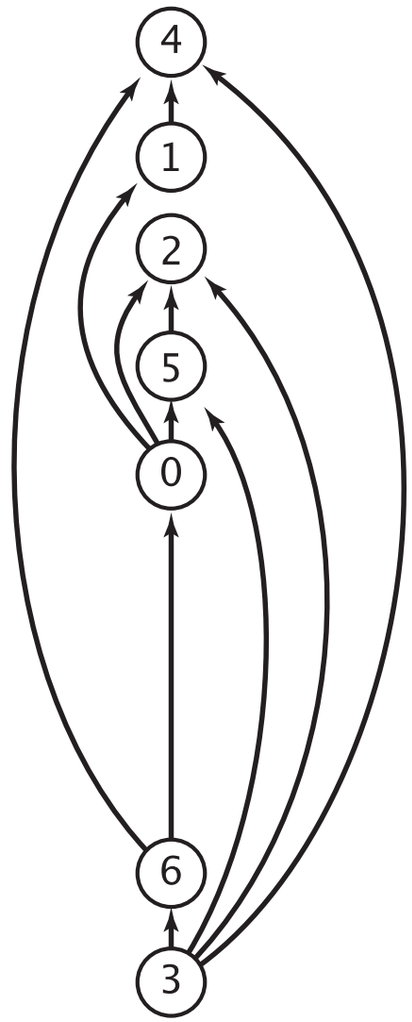
Digraph model. vertex = task; edge = precedence constraint.

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

tasks



precedence constraint graph

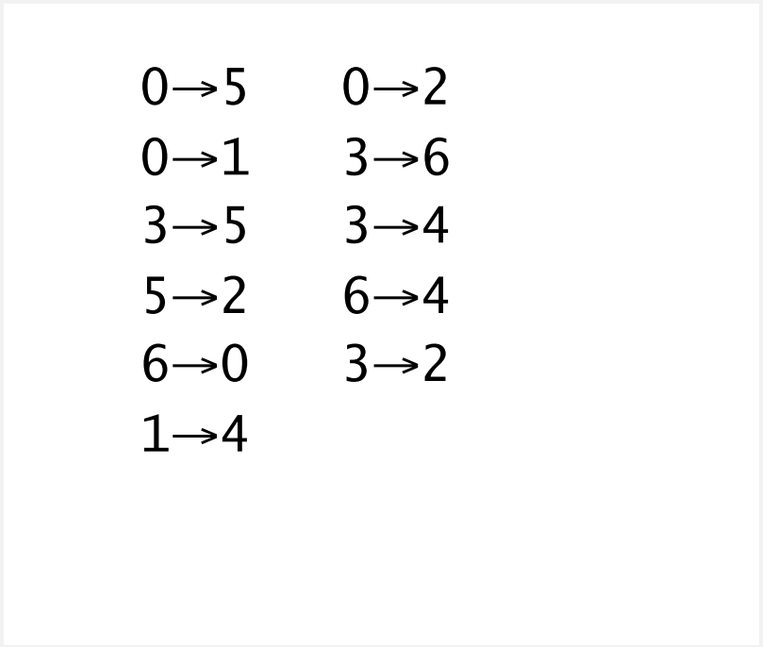


feasible schedule

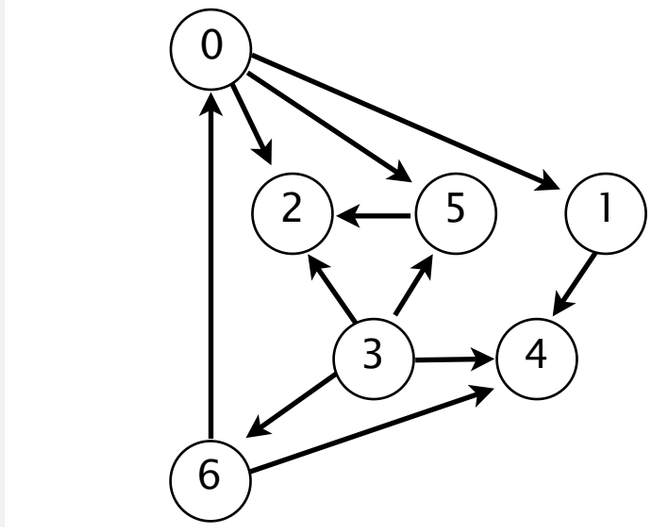
Topological sort

DAG. Directed acyclic graph.

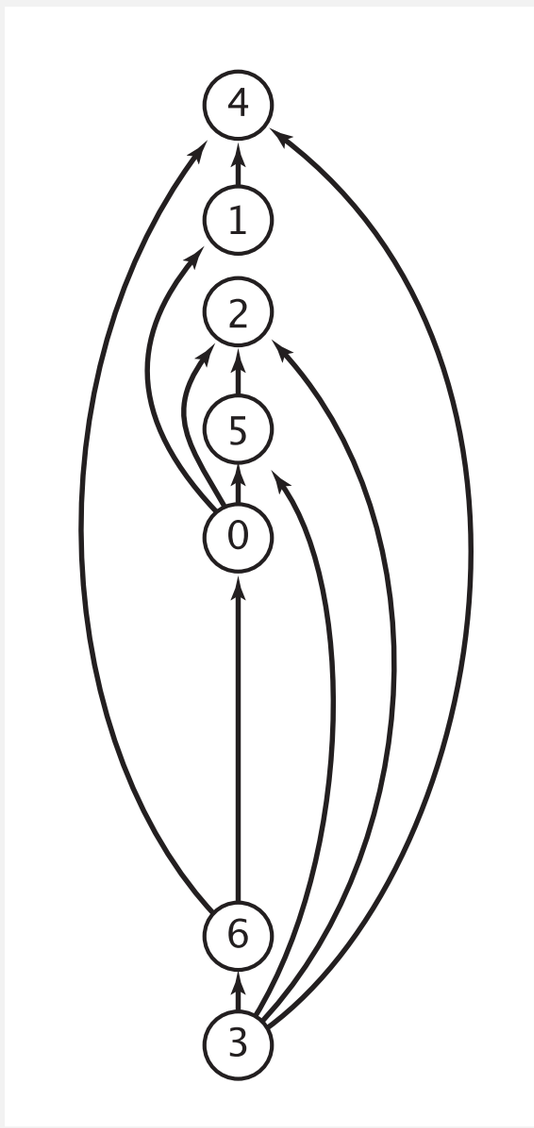
Topological sort. Redraw DAG so all edges point upwards.



directed edges



DAG



topological order

Solution. DFS. What else?

Depth-first search orders

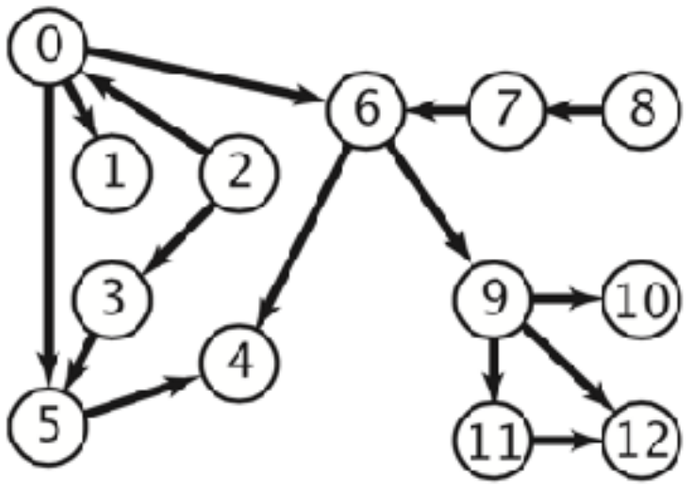
Observation. DFS visits each vertex exactly once. The order in which it does so can be important.

Orderings.

- Preorder: order in which `dfs()` is called.
- Postorder: order in which `dfs()` returns.
- Reverse postorder: reverse order in which `dfs()` returns.

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

Depth-first search orders




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

preorder is order of dfs() calls

postorder is order in which vertices are done

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

queue *queue* *stack*

reverse postorder

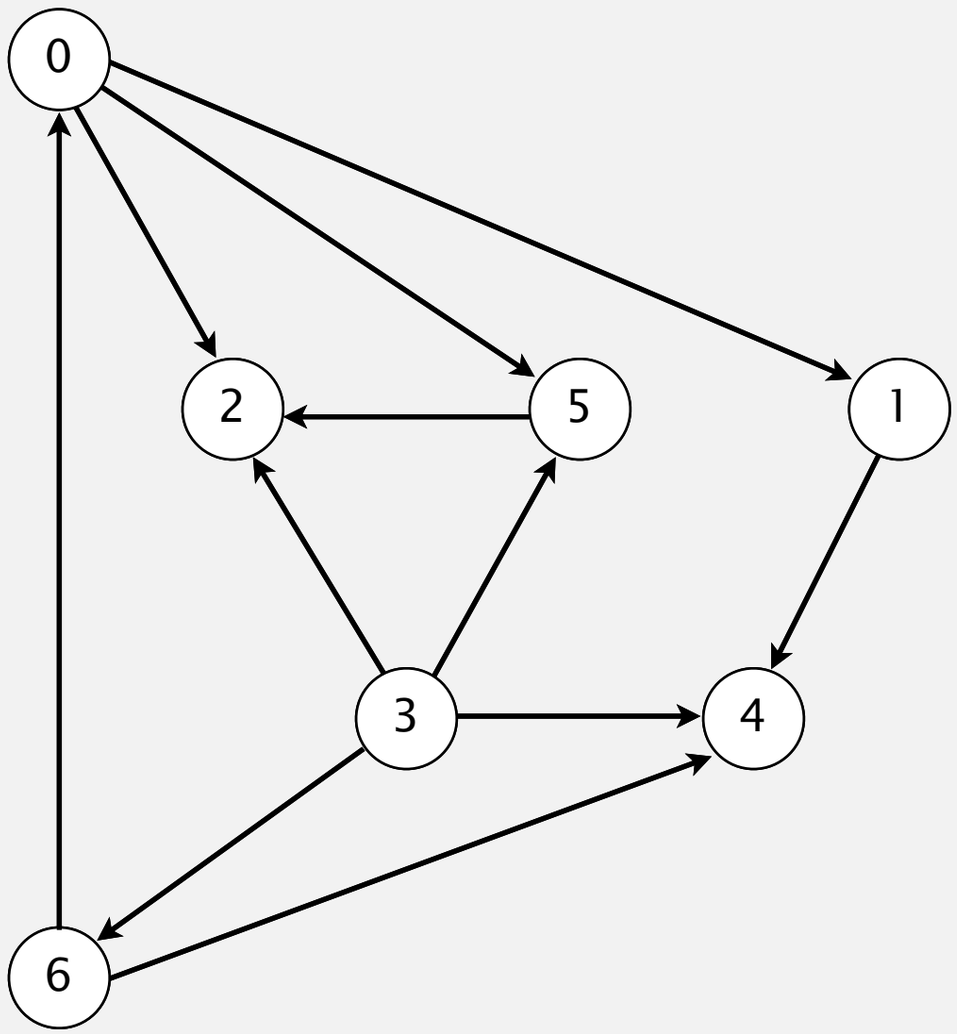
Topological sort algorithm

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

   public DepthFirstOrder(Digraph G)
   {
      reversePostorder = 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);
      reversePostorder.push(v);
   }

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

returns all vertices in "reverse DFS postorder"

Topological sort demo

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



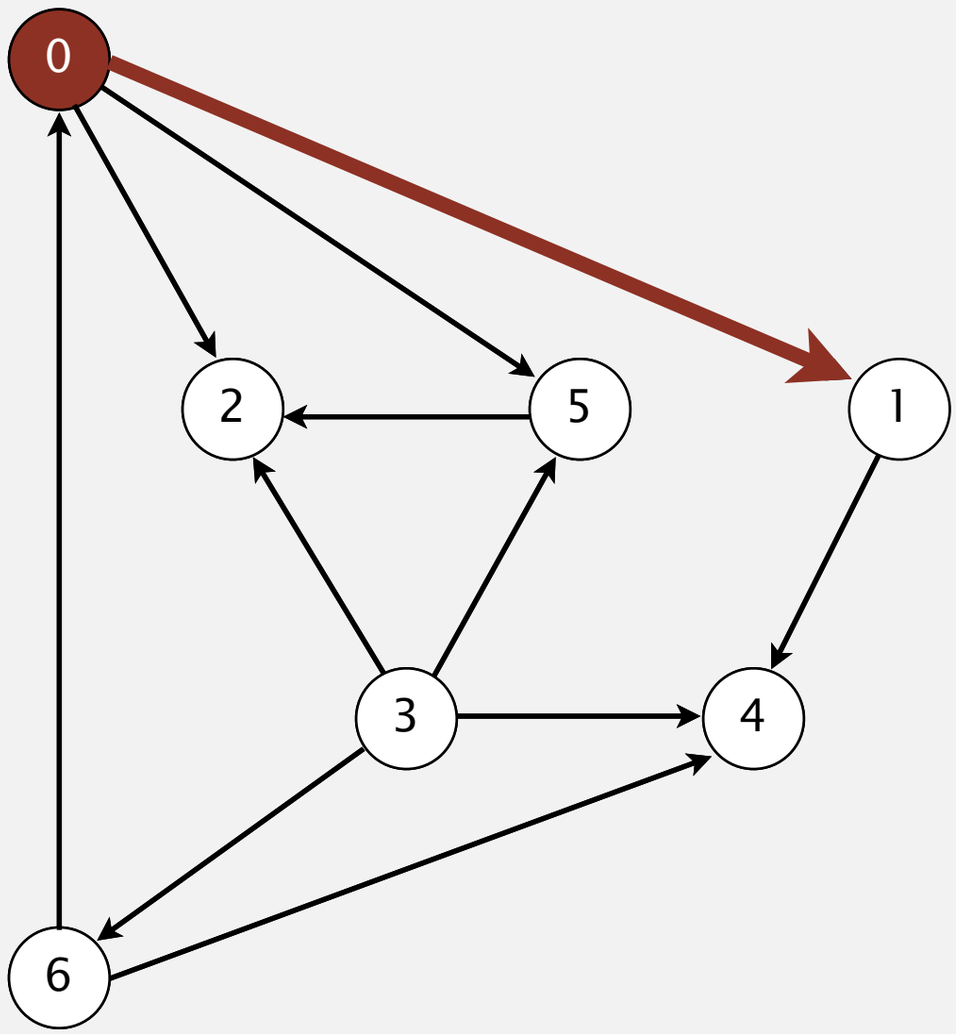
tinyDAG7.txt

```
7
11
 0  5
 0  2
 0  1
 3  6
 3  5
 3  4
 5  2
 6  4
 6  0
 3  2
```

a directed acyclic graph

Topological sort demo

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



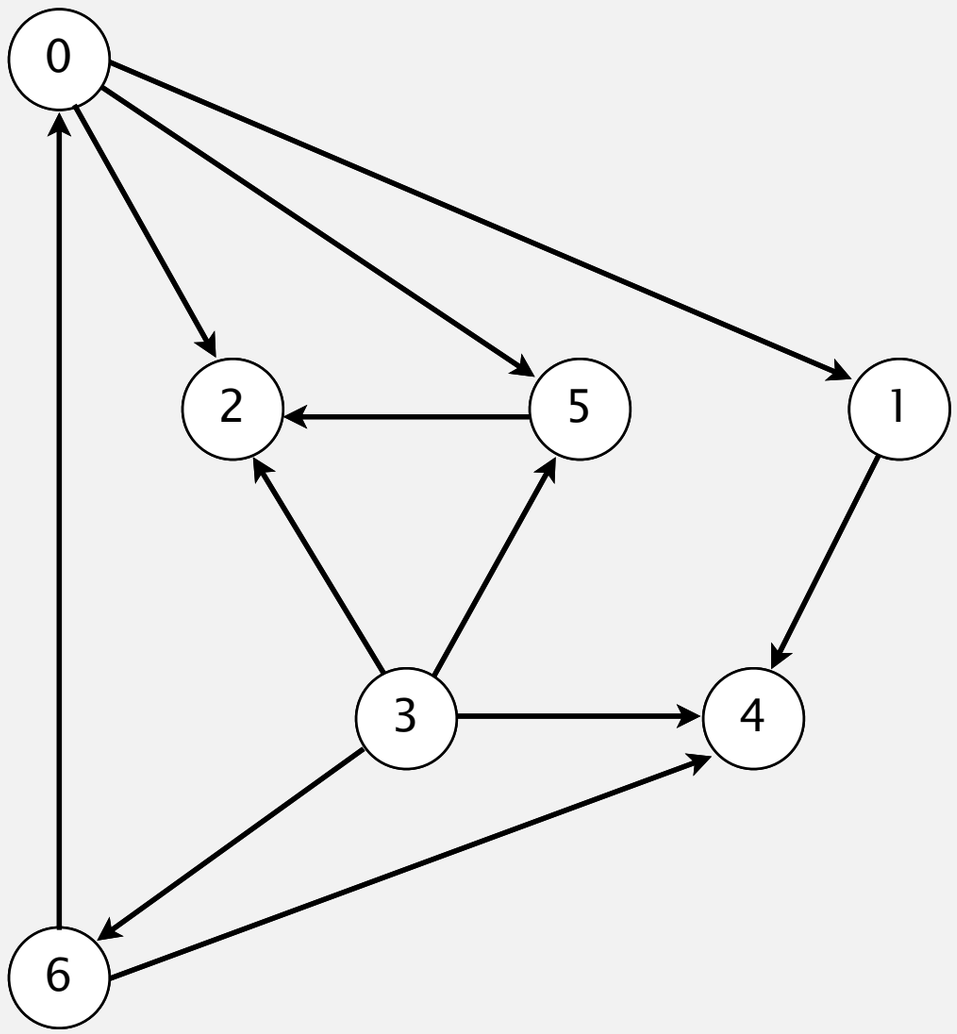
postorder

v	marked[]
0	T
1	F
2	F
3	F
4	F
5	F
6	F

visit 0: check 1, check 2, and check 5

Topological sort demo

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



postorder

4 1 2 5 0 6 3

topological order

3 6 0 5 2 1 4

done

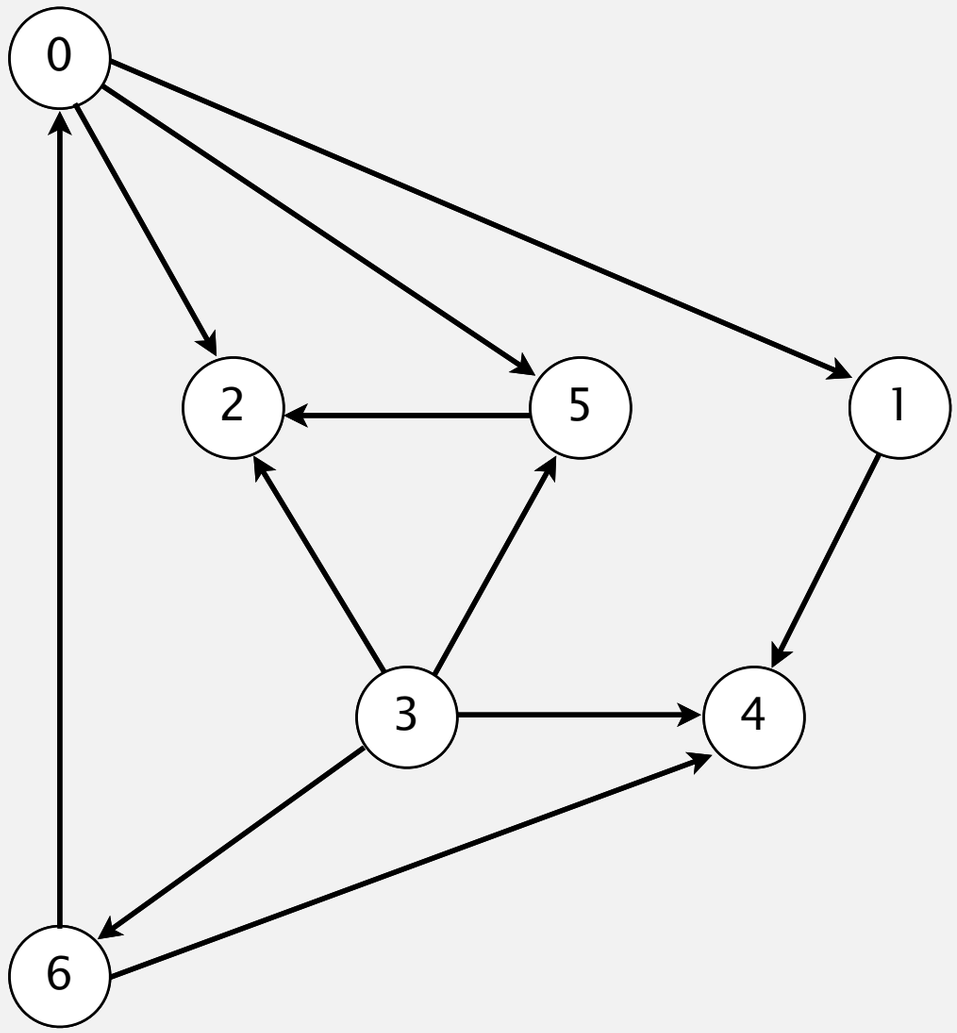
Topological sort algorithm

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

   public DepthFirstOrder(Digraph G)
   {
      reversePostorder = 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);
      reversePostorder.push(v);
   }

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

returns all vertices in "reverse DFS postorder"

Topological sort in a DAG: intuition (deeper node last !!)

```
dfs(0)
  dfs(1)
    dfs(4)
    4 done
  1 done
  dfs(2)
  2 done
  dfs(5)
    check 2
  5 done
0 done
check 1
check 2
dfs(3)
   check 2
   check 4
   check 5
   dfs(6)
      check 0
      check 4
   6 done
3 done
check 4
check 5
check 6
done
```



postorder

4 1 2 5 0 6 3

topological order

3 6 0 5 2 1 4

Topological sort in a DAG: correctness proof

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

Pf. Consider any edge v→w. When dfs(v) is called:

- Case 1: dfs(w) has already been called and returned.

  Thus, w was done before v.

- Case 2: dfs(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→w would complete a cycle.

```
dfs(0)
  dfs(1)
    dfs(4)
    4 done
  1 done
  dfs(2)
  2 done
  dfs(5)
    check 2
  5 done
  0 done
  check 1
  check 2
  dfs(3)
    check 2
    check 4
    check 5
    dfs(6)
      check 0
      check 4
    6 done
  3 done
  check 4
  check 5
  check 6
  done
```