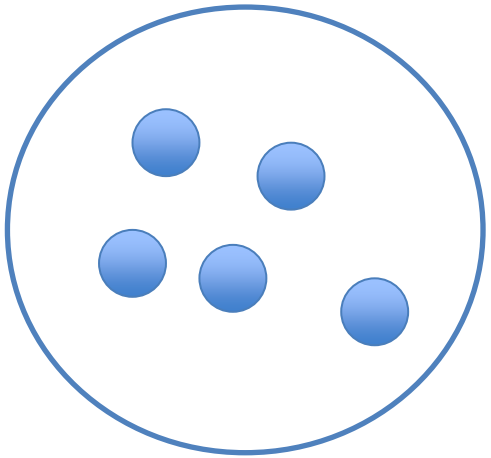
Lecture 10
Basic Graph Algorithms

Jianchen Shan
Department of Computer Science
Hofstra University

Lecture Goals

- Compare the Graph ADT with other ADTs
- Define basic notions associated with graphs
- Write classes in Java to implement graphs
- Implement graphs in Java using an adjacency matrix representation and an adjacency list representation
- Implement a method to find the neighbors of a vertex in two ways.
- We introduce two classic algorithms for searching a graph—depth-first search and breadth-first search.
- We also consider the problem of computing connected components and conclude with related problems and applications.
- we introduce a depth-first search based algorithm for computing the topological order of an acyclic digraph.

ADT of Graph



Unstructured structures

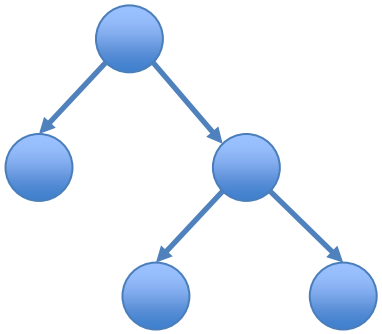
Sets

Sequential, linear structures

Arrays, linked lists

Useful for

- iterating over all elements,
- accessing via index



Hierarchical structures

Trees

Can indicate common structure in key

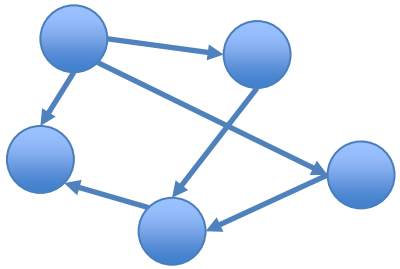
- for example, the prefix in tire

Principle: Basic objects & Relationships between them

Graph is a generalization of this principle

Basic objects: vertices, nodes

Relationships between them: edges, arcs, links



Examples of Graphs

Basic objects: we

Relationships bet

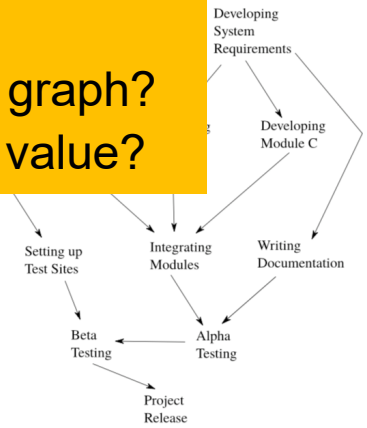
hem: nonstop flights

Some general questions related to graphs:

- How to create a graph?
- Are two vertices adjacent?
- Is the graph dense? sparse?
- How far are two vertices in the graph?
- How many components are there in the graph?
- Can we find a vertex with particular key value?

Basic objects: people

Relationships between them: friends



Basic objects: tasks

Relationships between them: dependencies

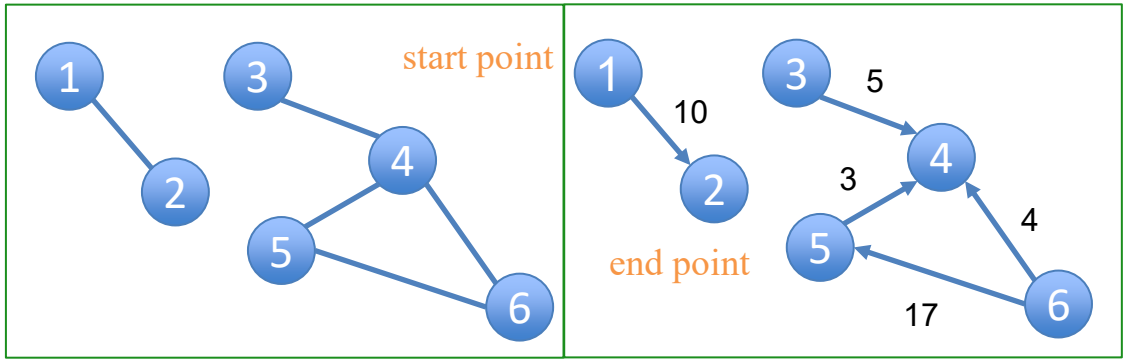
Graph Definitions

Basic objects: vertices, nodes — V

Relationships between them: edges, arcs, links — E

have a huge impact on the performance

Size of graph: |V| + |E|



Undirected

edges are symmetric

Directed

Weighted

cost

Neighbor: u is a neighbor of v if:
there is an edge from u to v
OR
there is an edge from v to u

What are the neighbors of the vertex 4?

A. 3,4,5,6
B. 3,5,6
C. 3,6
D. 5

Path: sequence of vertices and edges that depicts hopping along graph

For which pair of vertices is there a path in the graph starting at the first and ending at the second?

A. vertex 1 and vertex 3
B. vertex 4 and vertex 6
C. vertex 6 and vertex 5

What's the maximum number of edges in a directed and undirected graph with n vertices?

- n*(n-1) — e th — n*(n-1)/2 — elf-loops (i.e. edges from a node back to itself).
- Assume there there is at most one edge from a given start vertex to a given end vertex.

Implementing Graphs in Java

Basic objects: vertices, nodes ← Label by integers

Relationships between them: edges, arcs, links

```
public abstract class Graph {
        private int numVertices;
        private int numEdges;

        public Graph() {
                numVertices = numEdges = 0;
        }

        public int getNumVertices() {
                return numVertices;
        }

        public int getNumEdges() {
                return numEdges;
        }

        public void addVertex() {
                implementAddVertex();
                numVertices++;
        }

        public abstract void implementAddVertex();

        public abstract List<Integer> getNeighbors(int v);
}
```

size of a graph

data associated with any graph

methods that ought to be available with any graph.

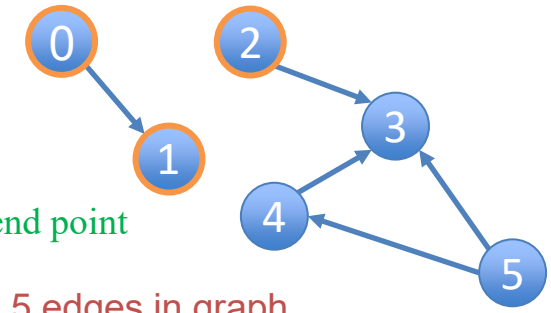
leave implementation of key functionalities to subclasses

For example, which cities we can reach with nonstop flight?

Graph Representation: Adjacency Matrix

relabel vertices starting at 0 to use as index

start point



end point

$V = \{0, 1, 2, 3, 4, 5\}$

5 edges in graph

→ 5 nonzero entries in adjacency matrix

Column index

row index

Symmetric entry

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

array entry > 1:
- multiple edges,
- or weighted edges

The grid (2-d array) is indexed by the vertices labels and stores information in a particular location based on whether these two vertices have an edge between them or not

```
public class GraphAdjMatrix extends Graph {
  private int[][] adjMatrix;

  public void implementAddEdge(int v, int w) {
    adjMatrix[v][w] = 1;
  }
  public void implementAddVertex() {
    int v = getNumVertices();
    if (v >= adjMatrix.length) {
          int[][] newAdjMatrix = new int[v * 2][v * 2];
      for (int i = 0; i < adjMatrix.length; i++) {
          for (int j = 0; j < adjMatrix.length; j++) {
              newAdjMatrix[i][j] = adjMatrix[i][j];
            }
        }
        adjMatrix = newAdjMatrix;
    }
  } // constructor, methods update the adjMatrix
}
```

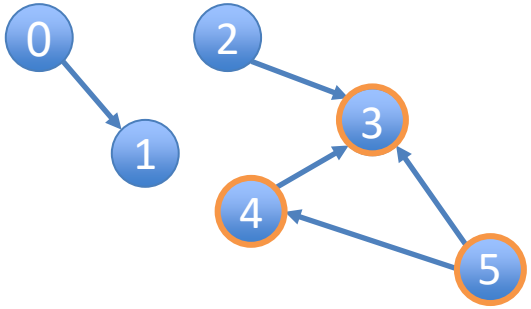
expand the 2-d array

How long does it take to test whether there is an edge between vertex v and vertex w in the graph?

O(1)

- Algebraic representation of graph structure.
- Fast to test for edges.
- Fast to add/remove edges.
- Slow to add/remove vertices.
- Requires a lot of memory. sparse

Graph Representation: Adjacency List



Motivation for new representation:
- want to avoid storing information on edges that aren't in the graph
- Edges connect a vertex to its neighbors

Neighbour can be reached by one hop

0 → {1}

1 → null

2 → {3}

3 → null

4 → {3}

5 → {3, 4}

- Easy to add vertices.
- Easy to add/remove edges.
- May use a lot less memory than adjacency matrices.

- Sparse graph: O(1) edges for each vertex
- most applications use sparse graphs

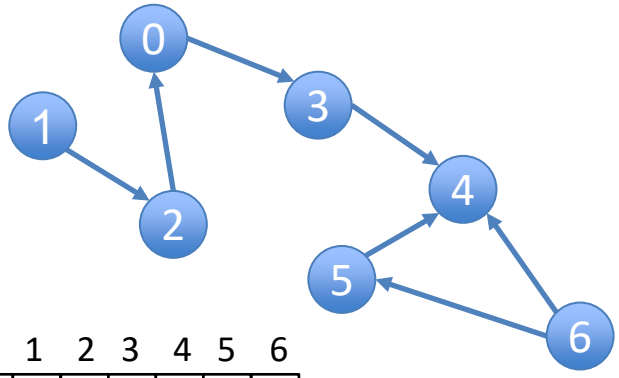
Is it also fast?

```
public class GraphAdjList extends Graph {
        private Map<Integer, ArrayList<Integer>> adjListsMap;

                                vertex → {neighbors}

        public void implementAddVertex() {
                int v = getNumVertices();
                ArrayList<Integer> neighbors = new ArrayList<Integ
                adjListsMap.put(v, neighbors);
        }

        public void implementAddEdge(int v, int w) {
                (adjListsMap.get(v)).add(w);
        }
}
```

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional
list operations, and permits all elements, including null. In addition to
implementing the List interface, this class provides methods to manipulate
the size of the array that is used internally to store the list. (This class is
roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in
constant time. The add operation runs in amortized constant time, that is,
adding n elements requires O(n) time. All of the other operations run in linear
time (roughly speaking). The constant factor is low compared to that for the
```

Yes. Operations are all O(1)

Some Practices



How much storage is required to represent a graph as a **matrix**? (Big-O, Tightest Bound)

A. |V| B. |E|
C. |V|+|E| D. $|V|^2$
E. $|E|^2$

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

What would change if undirected?

Half is redundant, but still $O(|V|^2)$

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

Symmetric matrix

For dense graphs with lots of edges, |E| will be as large as $|V|^2$

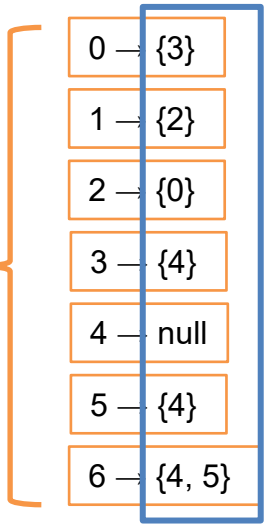
How much storage is required to represent a graph as **an adiancency list**? (Big-O, Tightest Bound)

A. |V| B. |E|
C. |V|+|E| D. $|V|^2$
E. $|E|^2$

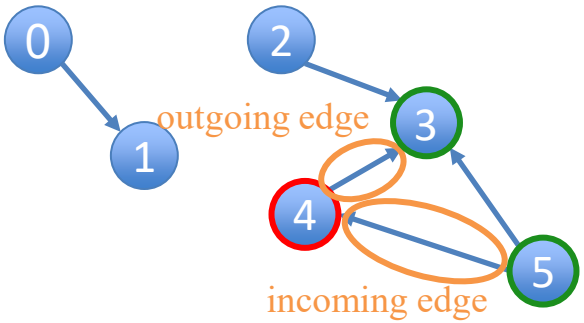
Much more efficient for sparse graphs!

O(|V|)

O(|E|)



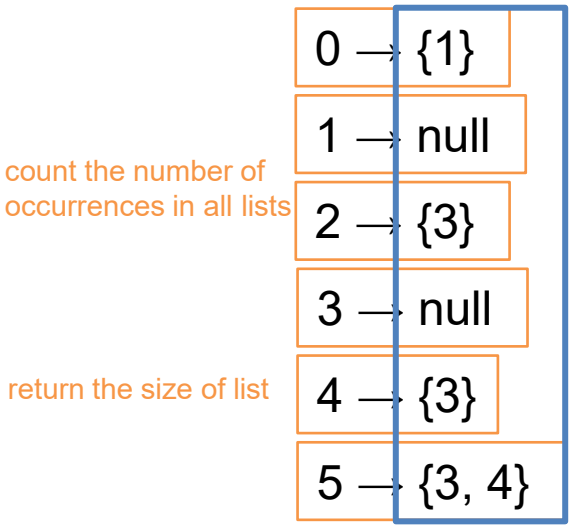
Find the Neighbors



Neighbors: vertices that are adjacent. there is edge in between

Out degree: number of outgoing edges.

In degree: number of incoming edges.



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

count the number of nonzero slots

Which implementation makes finding the out degree more efficient?

Matrix: O(|V|) List: O(1)

Which implementation makes finding the in degree more efficient?

Matrix: O(|V|) List: O(|E| + |V|)

For dense graphs without multiple edges between pairs of vertices, |E| is $O(|V|^2)$. so the adjacency matrix representation is faster. For sparse graphs, |E| = O(|V|) so both representations have the same performance.

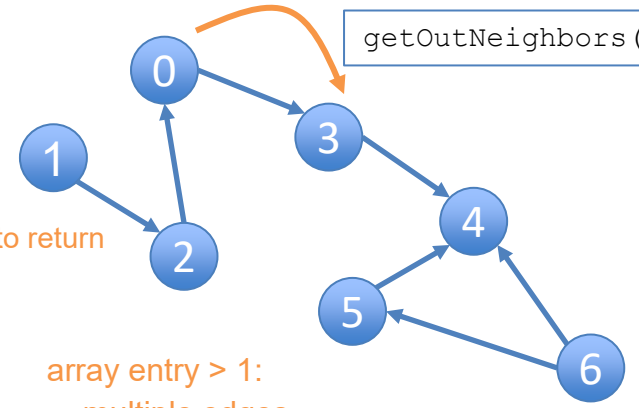
Coding getOutNeighbors (outgoing)

```
public class GraphAdjMatrix extends Graph {

        private int[][] adjMatrix;

        public List<Integer> getOutNeighbors(int v) {
                List<Integer> neighbors = new ArrayList<Integer>();
                for (int i = 0; i < getNumVertices(); i++) {
                        for (int j=0; j< adjMatrix[v][i]; j++)
                        if (adjMatrix[v][i] > 0)
                                neighbors.add(i);
                }
                return neighbors;
        }
}
```

// list to return

```
public class GraphAdjList extends Graph {

        private Map<Integer,ArrayList<Integer>> adjListsMap;

        public List<Integer> getOutNeighbors(int v) {
                return adjListsMap.get(v);                          // return v's list
                return new ArrayList<Integer>(adjListsMap.get(v));

        }                                                           // return a COPY of v's list
}
```

From a software development point of view, is this a good implementation?



array entry > 1:
- multiple edges

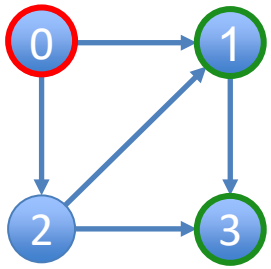
	0	1	2	3	4	5	6
0	0	0	0	2	0	0	0
1	0	0	1	0	0	0	0
2	1	0	0	0	0	0	0
3	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0
6	0	0	0	0	1	1	0

0 → {3}
1 → {2}
2 → {0}
3 → {4}
4 → null
5 → {4}
6 → {4, 5}

What does this change do?

A. It's a change in the code but will not materially affect the output.
B. It will take multiple edges into account.
C. It will have some other effect on the code behavior.

Coding 2-Hop Neighbors (outgoing)



0 → {1, 2}

1 → {3}

2 → {1, 3}

3 → null

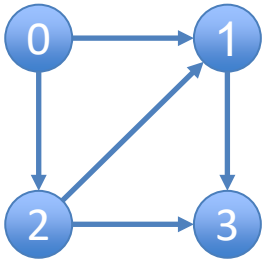
	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	1	0	1
3	0	0	0	0

Find all two-hop neighbors from given vertex

```
public class GraphAdjList extends Graph {

        private Map<Integer,ArrayList<Integer>> adjListsMap;

        public List<Integer> getDistance2 (int v) {
                List<Integer> distance2 = new ArrayList<>();

                // Loop through oneHop and get the neighbors of each
                for(int u : getOutNeighbors(v)){
                        distance2.addAll(getOutNeighbors(u));
                }
                return distance2;
        }
}
```

```
public class GraphAdjMatrix extends Graph {

        private int[][] adjMatrix;

        public List<Integer> getDistance2 (int v) {
                List<Integer> distance2 = new ArrayList<Integer>();

                // Loop through oneHop and get the neighbors of each
                for(int u : getOutNeighbors(v)){
                        distance2.addAll(getOutNeighbors(u));
                }
                return distance2;
        }
}
```

Coding 2-Hop Neighbors (Matrix Multiplication)



Matrix multiplication for finding two-hop neighbors

For all the vertices in the graph

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}^2 = \text{matrix whose entries are two-hop neighbors!}$$

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	1	0	1
3	0	0	0	0

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} & & & \\ & & & \\ & & & \\ & & & \end{pmatrix}$$

Dot product

$0*0 + 1*0 + 1*0 + 0*0 = 0$

$0*1 + 1*0 + 1*1 + 0*0 = 1$

$0*1 + 1*0 + 1*0 + 0*0 = 0$

$0*0 + 1*1 + 1*1 + 0*0 = 2$

$0*0 + 0*0 + 0*0 + 1*0 = 0$

	0	1	2	3
0	0	1	0	2
1	0	0	0	0
2	0	0	0	1
3	0	0	0	0

Matrix multiplication is well studied and optimized in software and hardware, and can be done very fast

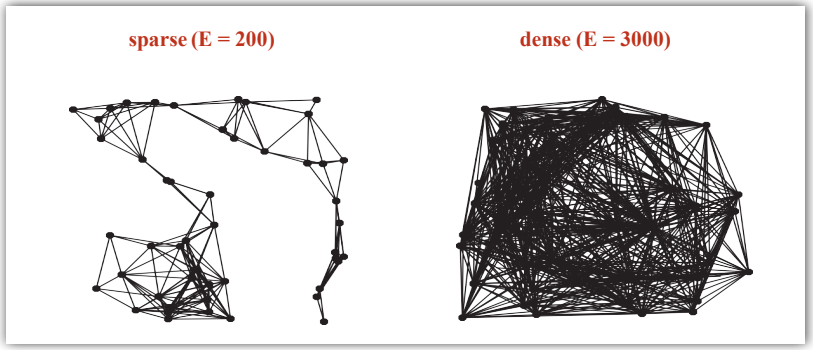
Summary of Digraph Representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent from v.
- Real-world graphs tend to be sparse (not dense).

proportional to V

proportional to V^2



Two graphs (V = 50)

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices adjacent from v?
adjacency matrix	V^2	1 †	1	V
adjacency lists	$E + V$	1	*outdegree(v)*	*outdegree(v)*

† disallows parallel edges

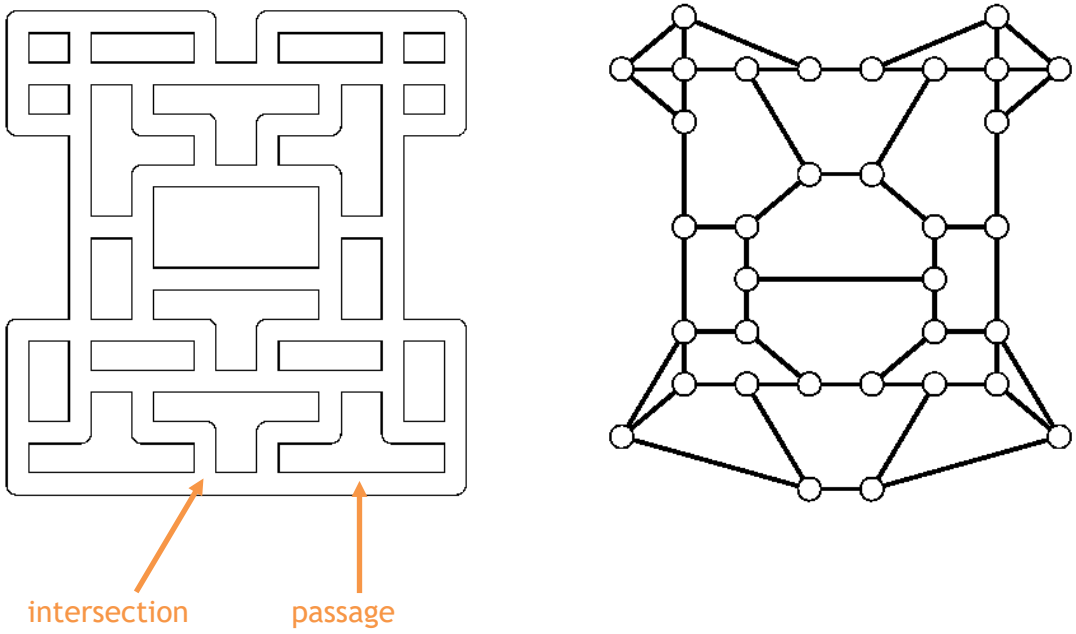
Lecture Goals

- Compare the Graph ADT with other ADTs
- Define basic notions associated with graphs
- Write classes in Java to implement graphs
- Implement graphs in Java using an adjacency matrix representation and an adjacency list representation
- Implement a method to find the neighbors of a vertex in two ways.
- We introduce two classic algorithms for searching a graph—depth-first search and breadth-first search.
- We also consider the problem of computing connected components and conclude with related problems and applications.
- we introduce a depth-first search based algorithm for computing the topological order of an acyclic digraph.

Represent Problems as Graphs: Maze Exploration

Goal. Explore every intersection in the maze.

Maze graph. Vertex = intersection. Edge = passage.



Depth-First Search (DFS)

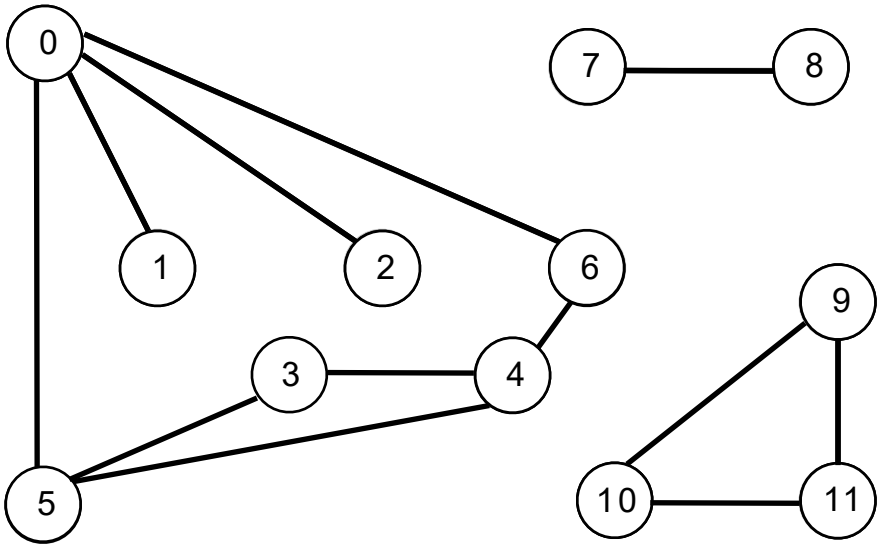
Goal. Systematically traverse a graph.

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

DFS (to visit a vertex v)

Mark vertex v.

Recursively visit all unmarked vertices w adjacent to v.



Java execution stack is used to keep track of where to search next

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	–	

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

Data structures.
- Boolean array marked[] to mark vertices.
- Integer array edgeTo[] to keep track of paths.
  - (edgeTo[w] == v) means that edge v-w taken to discover vertex w

Class 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
```

```
Paths paths = new Paths(G, s);
        for (int v = 0; v < G.V(); v++)
                if (paths.hasPathTo(v))
                        StdOut.println(v);
```

← print all vertices connected to s

Depth-First Search: Java Implementation

```
public class DepthFirstPaths {

  private boolean[] marked;
  private int[] edgeTo;
  private int s;

  public DepthFirstPaths(Graph G, int 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);
        }
  }
}
```

marked[v] = true if vconnected to s

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

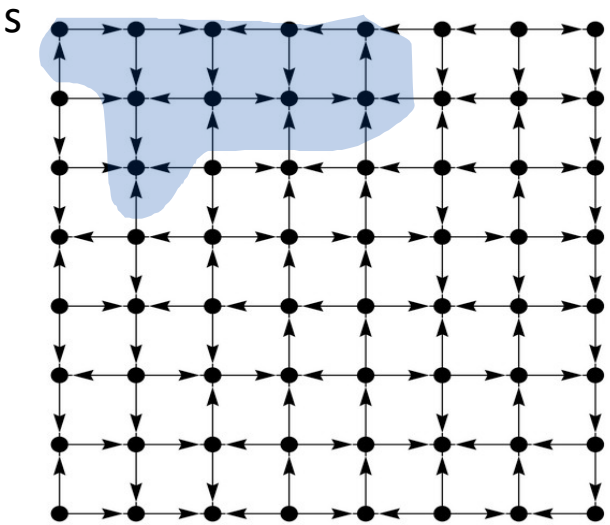
initialize data structures

find vertices connected to s

recursive DFS does the work

Depth-First Search For Directed Graph

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



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

- Same method as for undirected graphs.
- Code for directed graphs identical to undirected one.

```
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 connected to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is connected to s

Depth-First Search: Properties

Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees (plus time to initialize the `marked[ ]` array).

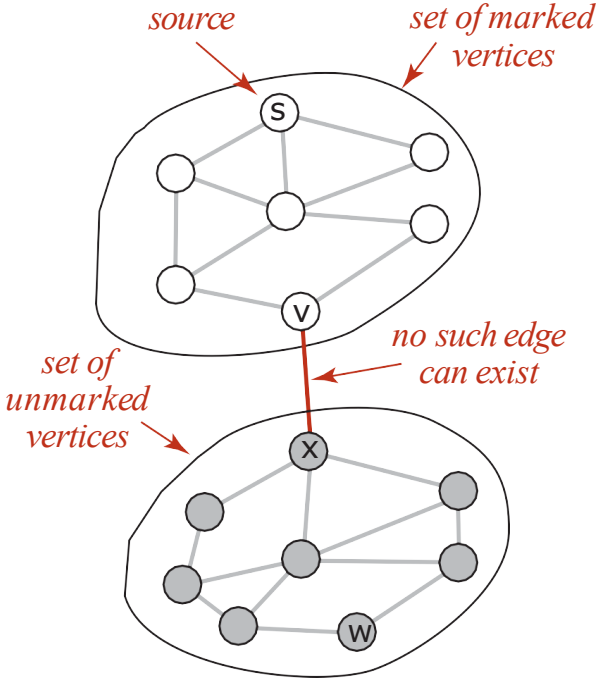
Pf. [correctness]
If w connected to s, then w marked. (if w unmarked, then consider last edge on a path from s to w that goes from a marked vertex to an unmarked one).

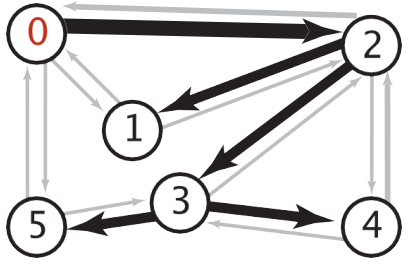
Pf. [running time]
Each vertex connected to s is visited once.

Proposition. After DFS, can check if vertex v is connected to s in constant time and can find v–s path (if one exists) in time proportional to its length.

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

```
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[]

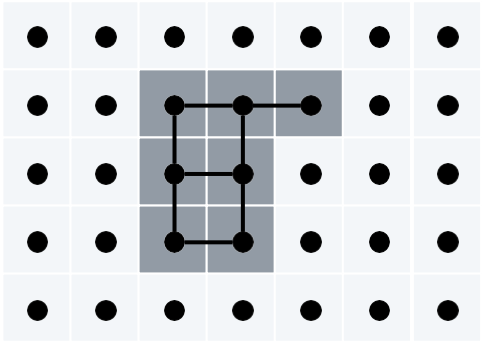
0	
1	2
2	0
3	2
4	3
5	3

Depth-First Search Application: Flood Fill

Problem. Flood fill (Photoshop magic wand).
Assumptions. Picture has millions to billions of pixels.

Solution.

- Build a grid graph.
- Vertex: pixel.
- Edge: between two adjacent gray pixels.
- Blob: all pixels connected to given pixel.



Reachability Application: Mark–Sweep Garbage Collector

Every data structure is a digraph.

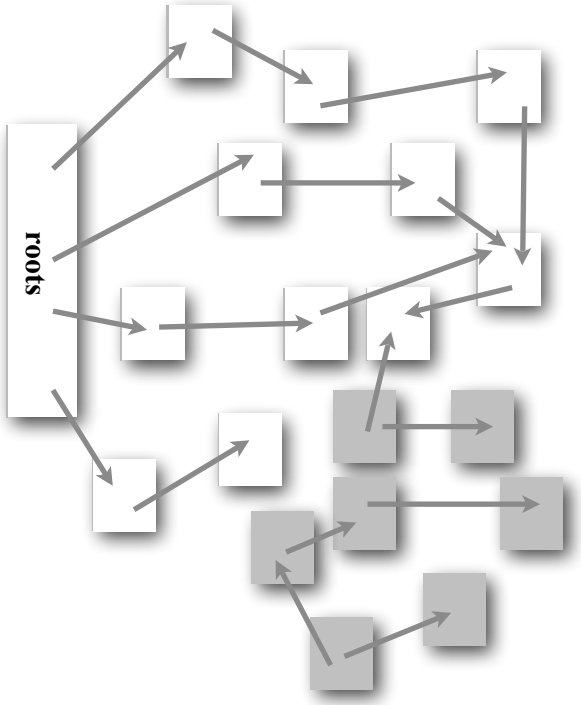
- Vertex = object.
- Edge = reference.
- Roots: Objects known to be directly accessible by program (e.g., stack).
- Reachable objects: Objects indirectly accessible by program (starting at a root and following a chain of pointers).

Mark–sweep algorithm. [McCarthy, 1960]

Mark: mark all reachable objects.

Sweep: if object is unmarked, it is garbage (so add to free list).

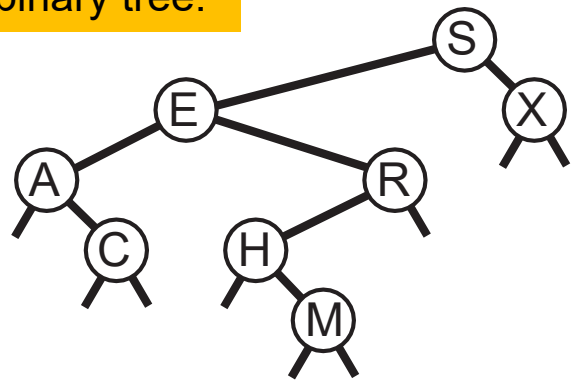
Memory cost. Uses 1 extra mark bit per object (plus DFS stack).



Graph Traversals

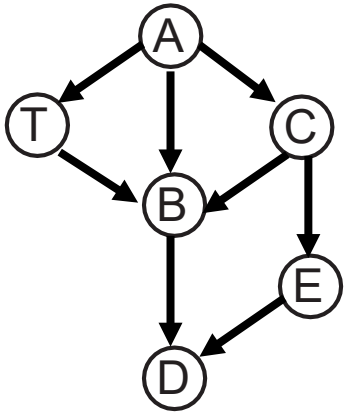
Tree traversal. Many ways to explore every vertex in a binary tree.

Inorder:	A C E H M R S X
Preorder:	S E A C R H M X
Postorder:	C A M H R E X S
Level-order:	S E X A R C H M



Graph search. Many ways to explore every vertex in a graph.

- Preorder: vertices in order of calls to dfs(G, v).
  - A, C, B, D, E, T
- Postorder: vertices in order of returns from dfs(G, v) (when all its outgoing edges have been visited.).
  - D, B, E, C, T, A
  - In DAG (directed acyclic graph), it ensures that for two nodes V and W, if there is a path from W to V in the graph, then V comes before W in the list
- Level-order: vertices in increasing order of distance from s.
  - A, C, B, T, E, D
  - vertices in order of calls to bfs(G, v) (Breath First Search)
- Reverse Postorder: the reverse of the list created by post-order traversal
  - A, T, C, E, B, D
  - the RPO of a DAG has another name - topological sort.



Breadth-First Search (BFS)

BFS (from source vertex s)

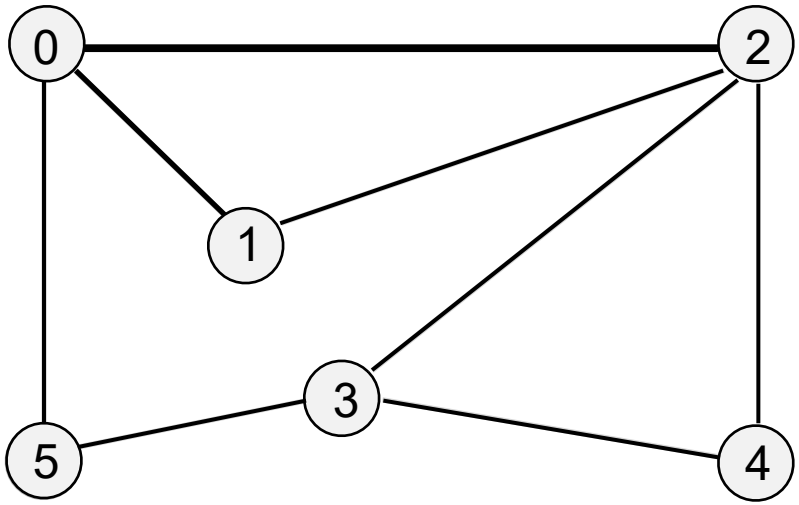
Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

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

Queue

4
3
5
1
2
0



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

distTo[v] = distTo[edgeTo[v]] + 1;

s.distTo[v] stores the distance from s to v

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;
            }
         }
      }
   }
}
```

- DFS. Put unvisited vertices on a stack.
- BFS. Put unvisited vertices on a queue.

initialize FIFO queue of vertices to explore

found new vertex w via edge v–w

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

- For directed graph, same method as for undirected graphs.
- Code for directed graphs identical to undirected one.

Breadth-First Search Properties

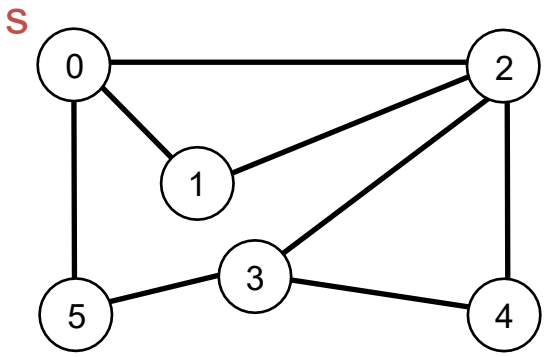
level-order

Proposition. BFS examines vertices in increasing distance (number of edges) from s.

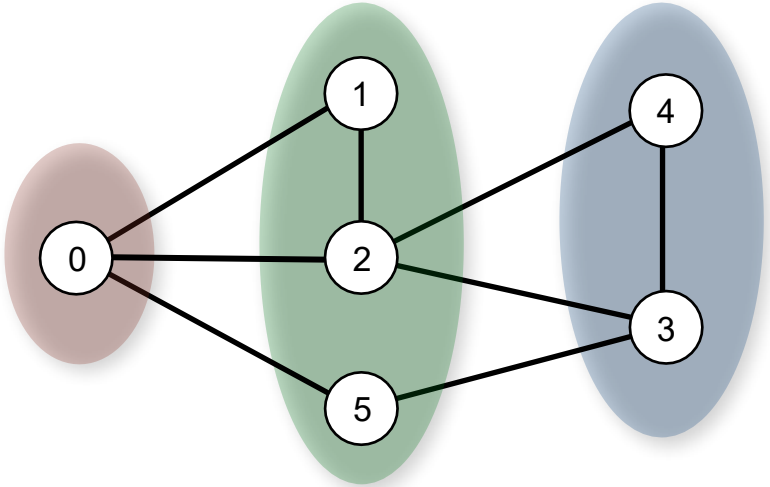
Proposition. In any connected graph, BFS computes shortest paths (fewest number of edges) from s to all other vertices in time proportional to E + V.

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.

Pf. [running time] Each vertex connected to s is visited once, and all its edges are checked.

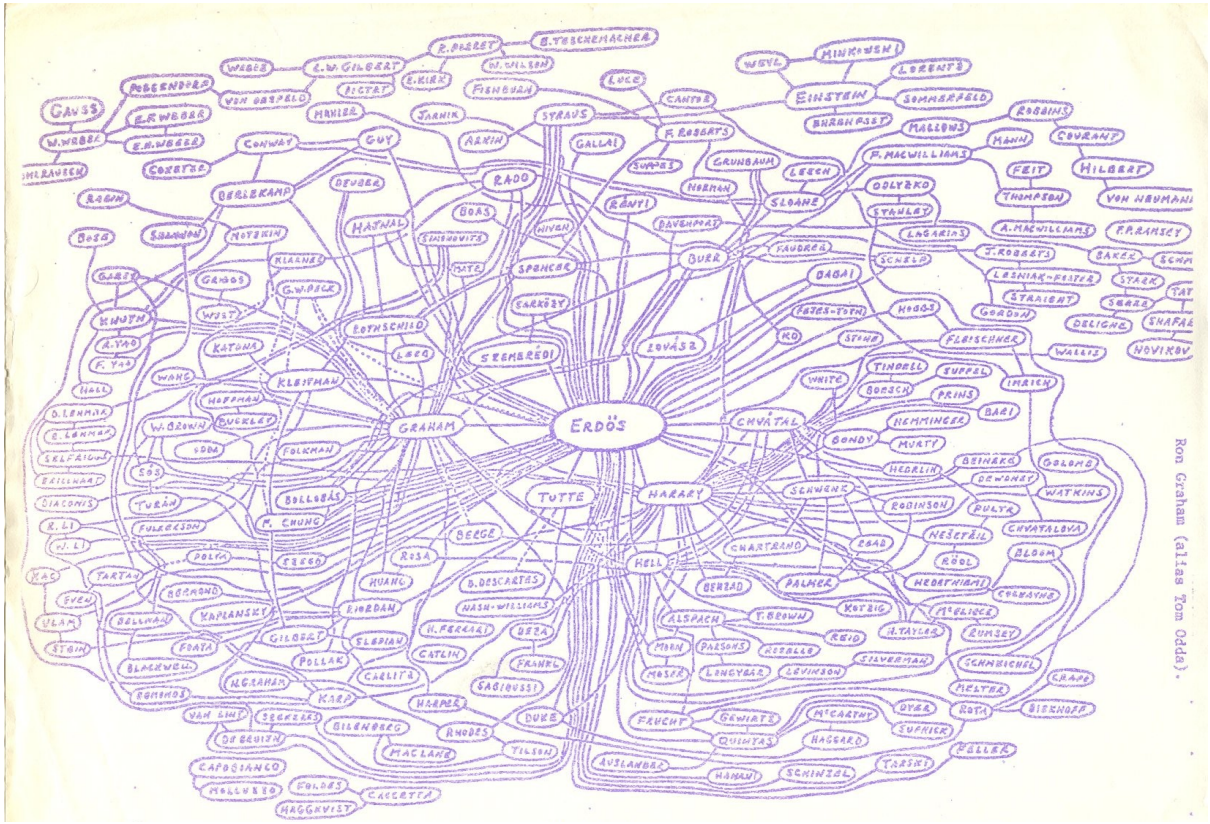


graph G



dist = 0 dist = 1 dist = 2

Breadth-First Search Application: Erdös numbers



My Erdös Number is: 4
(Jianchen Shan → Reza Curtmola → Baruch Awerbuch → Noga Alon → Paul Erdős)

The Erdős number is the number of "hops" needed to connect the author of a paper with the prolific late mathematician Paul Erdős. An author's Erdős number is 1 if he has co-authored a paper with Erdős, 2 if he has co-authored a paper with someone who has co-authored a paper with Erdős, etc. Erdős wrote papers with a total of 509 coauthors, meaning 509 people have Erdős number 1.

Breadth-First Search Application: Web Crawler

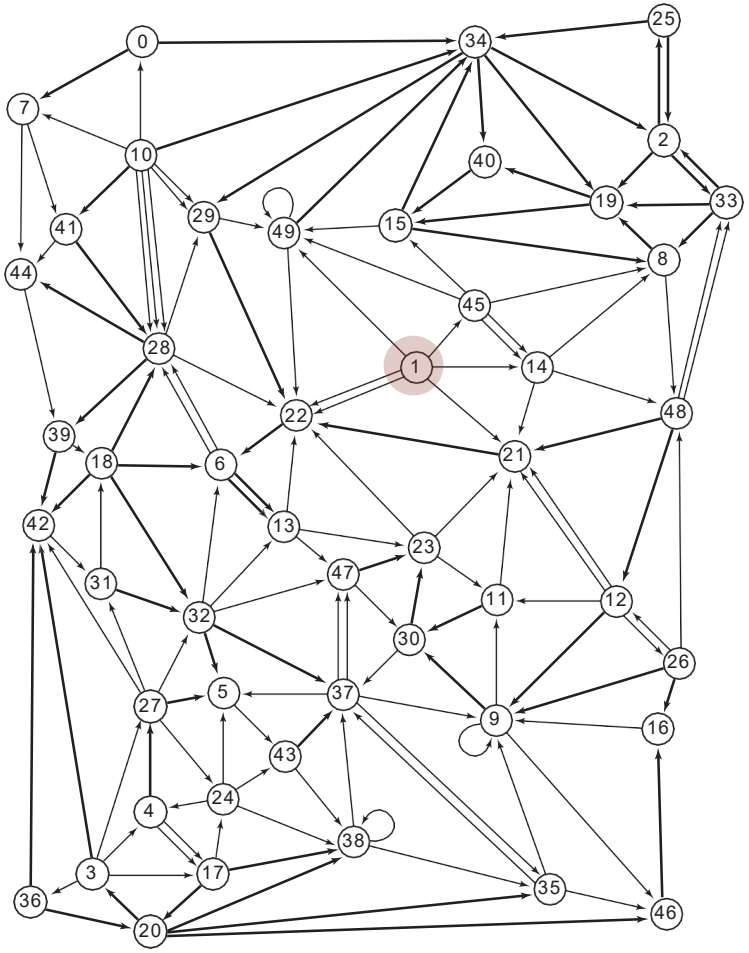
Goal. Crawl web, starting from some root web page, say www.hofstra.edu.

Solution. [BFS with implicit digraph]

- Choose root web page as source s.
- Maintain a Queue of websites to explore.
- Maintain a SET of marked websites.
- Dequeue the next website and enqueue any unmarked websites to which it links.

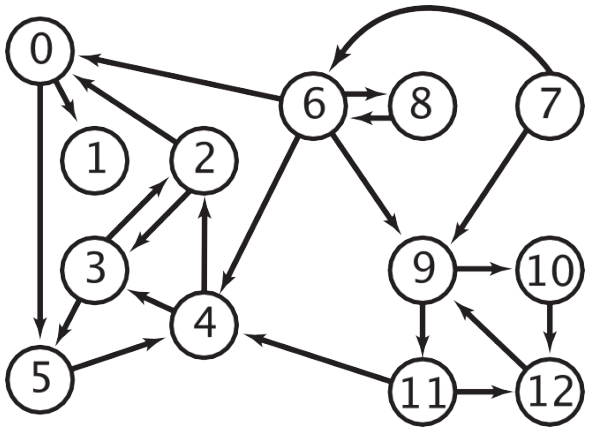
Why not use DFS?

Some web pages would **trap** the DFS search by creating new web pages and make links to them the first time that you visit them. DFS would always go to a new web page like that and it'd keep creating new ones and you wouldn't get very far.



Multiple-Source Shortest Paths Problem

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

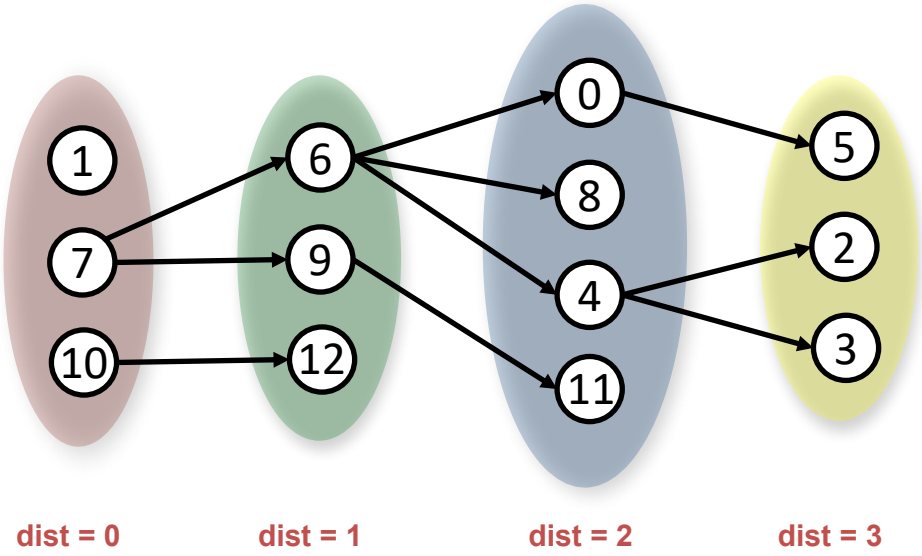


Ex. S = { 1, 7, 10 }.

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

How to implement multi-source shortest paths algorithm?

Use BFS, but initialize by enqueuing all source vertices.



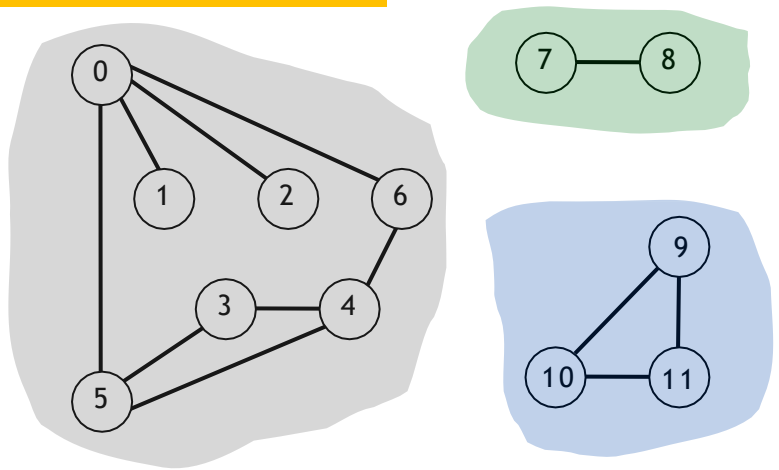
Connectivity Queries Problem

- Vertices v and w are connected if there is a path between them.
- In undirected graph, 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.
- Goal. Preprocess undirected graph to answer queries of the form *is v connected to w?* in constant time while using adjacency list.
- A connected component is a maximal set of connected vertices.
- Given connected components, can answer queries in constant time.

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

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*

DFS is used



3 connected components

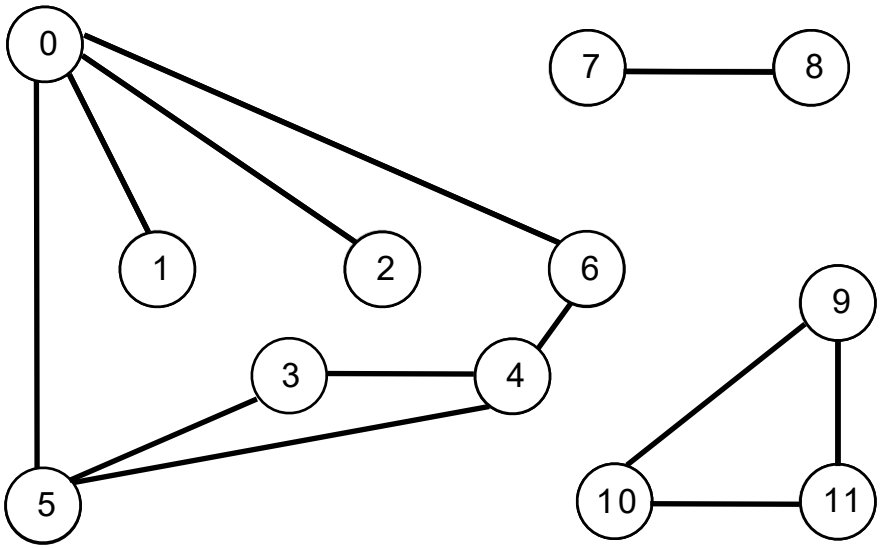
Finding Connected Components with DFS

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.



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	←

Java execution stack

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

Finding CCs with DFS: Java Implementation

```
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() { 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);
}
```

id[v] = id of component containing v

number of components

run DFS from one vertex in each component

number of components

id of component containing v

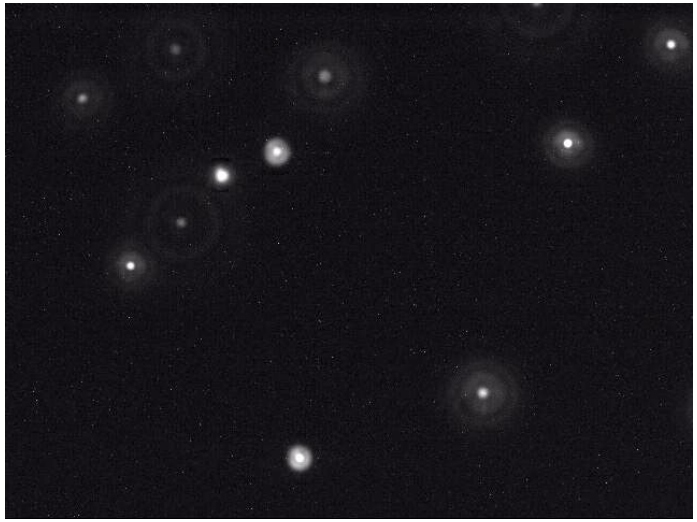
all vertices discovered in same call of dfs have same id

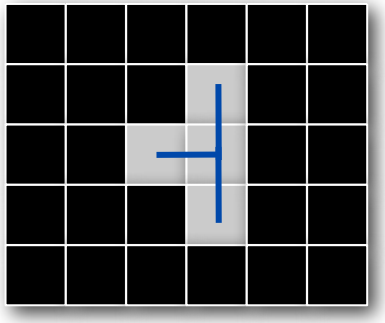
Connected Components Application: Particle Detection

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.

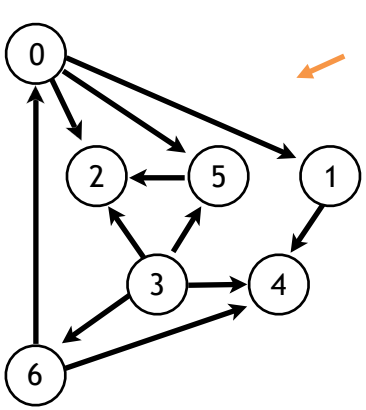
Precedence Scheduling Problem

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

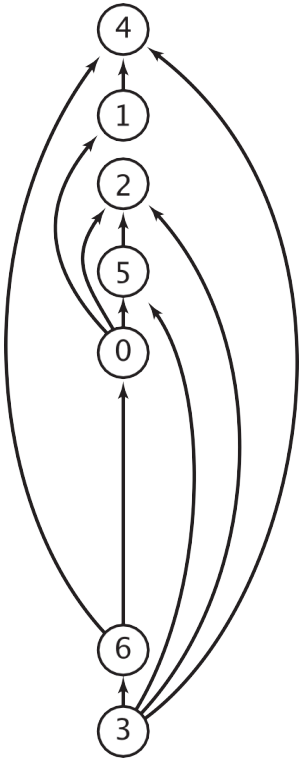
Digraph model. vertex = task; edge = precedence constraint.

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

tasks



precedence constraint graph



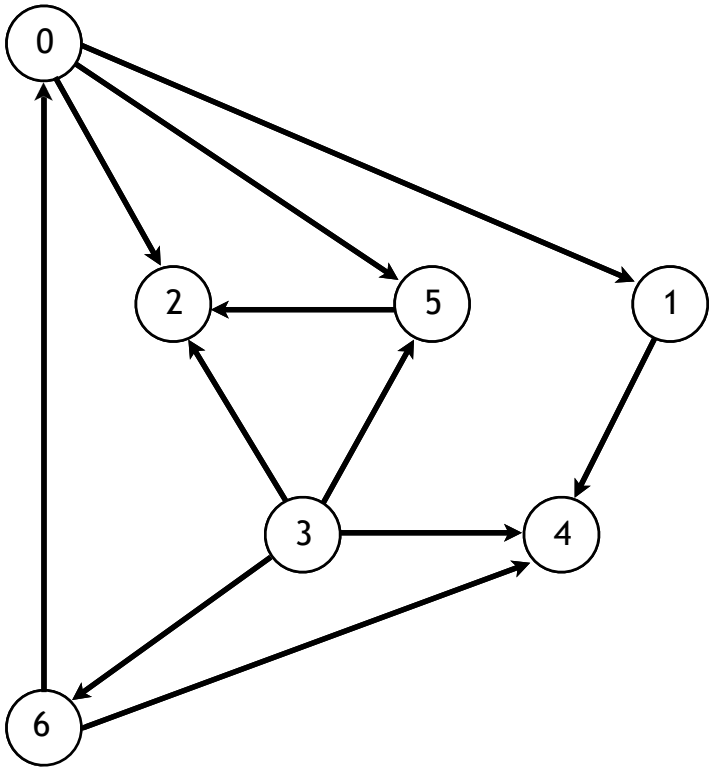
feasible schedule

Topological sort. Redraw DAG(Directed acyclic graph) so all edges point upwards.

Topological Sort

- Run depth-first search from every unmarked vertex
- Return vertices in reverse postorder.

not a reachability problem



stack top

Postorder

4	1	2	5	0	6	3

Topological order

3	6	0	5	2	1	4

pop from the stack → reversed postorder

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

Java execution stack

```
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
```

Topological Sort: Java Implementation

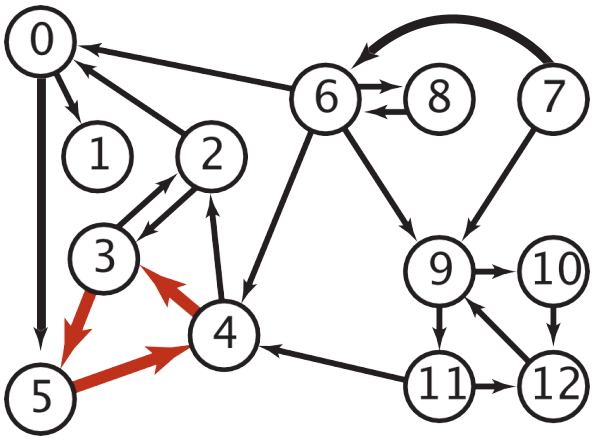
```
public class DepthFirstOrder {
   private boolean[] marked;
   private Stack<Integer> reversePostorder;

   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"

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

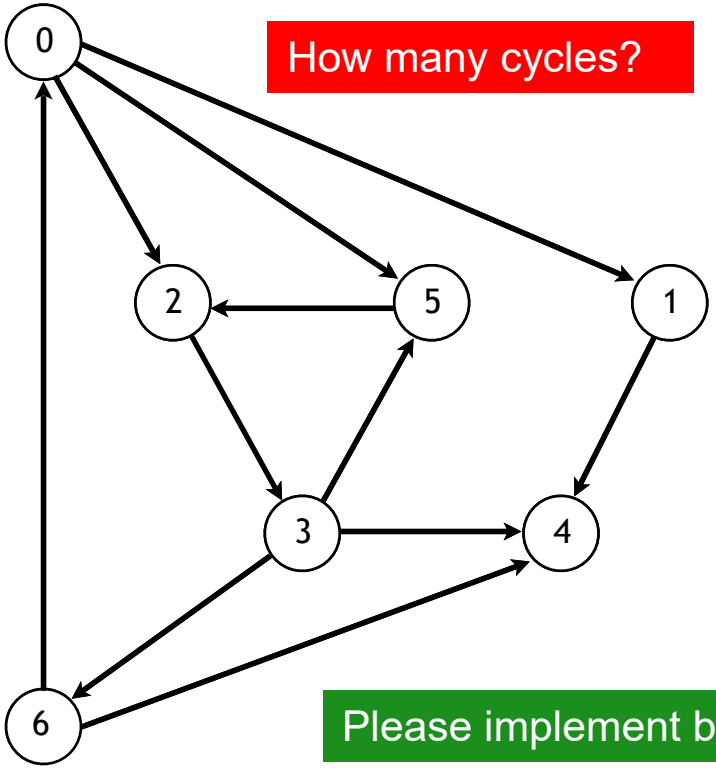
Goal. Given a digraph, find a directed cycle.

Solution. DFS. See next slide.

Directed Cycle Detection

- Run depth-first search from every unmarked vertex.
- Keep track of vertices currently in recursion stack of function for DFS traversal with onStack[ ] array.
- If we reach a vertex that is already in the recursion stack, then we found a cycle in the tree, and we're done
- Retrive the cycle using edgeTo[ ] array.

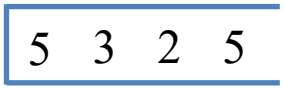
- set onStack[v] to T when dfs(v) is called
- set onStack[v] to F when dfs(v) returns



How many cycles?

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

stack top



Java execution stack

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

- vertex is marked and onStack
- Found the cycle
- Save the cycle using edgeTo[ ] to a stack

Please implement by yourself

Directed Cycle Detection Application: Cyclic Inheritance

The Java compiler does cycle detection.

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

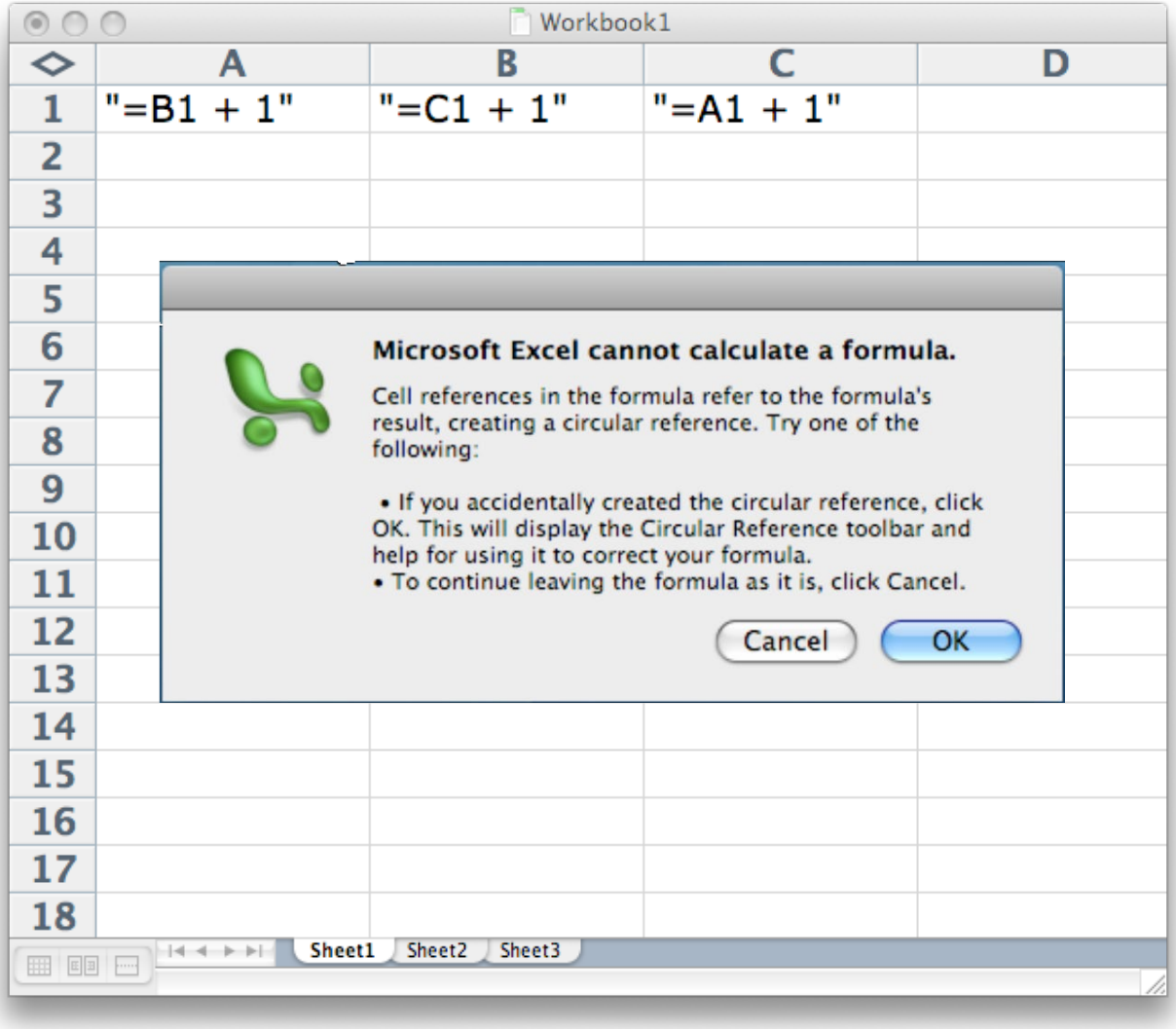
```
public class B extends C
{
   ...
}
```

```
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.



Topological Sort in a DAG: Correctness Proof

Consider an execution of DFS on a DAG, G which contains the edge $v \rightarrow w$. Which one of the following is impossible at the time dfs(v) is called?

A. dfs(w) has not yet been called.

B. dfs(w) will be the next recursive call after dfs(v).

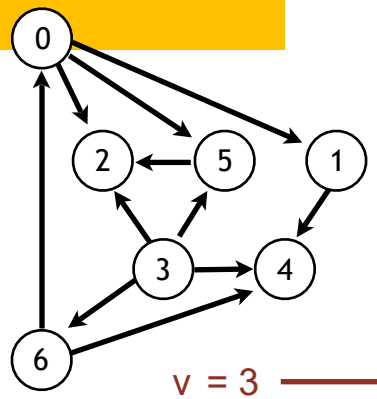
C. dfs(w) has already been called but not yet returned.

- 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 appears before v in postorder

Case 2: dfs(w) has not yet been called.
- dfs(w) will get called directly or indirectly by dfs(v)
- so, dfs(w) will return before dfs(v)
- thus, w appears before v in postorder

Case 3: dfs(w) has already been called, but has not yet returned.
- function-call stack contains path from w to v
- edge v → w would complete a directed cycle
- contradiction (it's a DAG)



v = 3

case 1
(w = 2, 4, 5)

case 2
(w = 6)

Java execution stack

```
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
…
```

Additional Resources

- Graphs, generally
  - http://www.tutorialspoint.com/data_structures_algorithms/graph_data_structure.htm -- what is graph, verteces, and edges
- Adjacency matrix and list
  - https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs -- explanation with examples
- DFS and BFS
  - https://www.youtube.com/watch?v=bIA8HEEUxZI -- video of BFS and DFS with example
  - http://www.programmerinterview.com/index.php/data-structures/dfs-vs-bfs/ -- explanation for both as well as their differences