INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*



copyrighted content - Do not share

http://ds.cs.rutgers.edu

Some slides Adopted and modified from Sedgewick and Wayne

last modified on 2/11/08 8:38 AM

UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

http://ds.cs.rutgers.edu

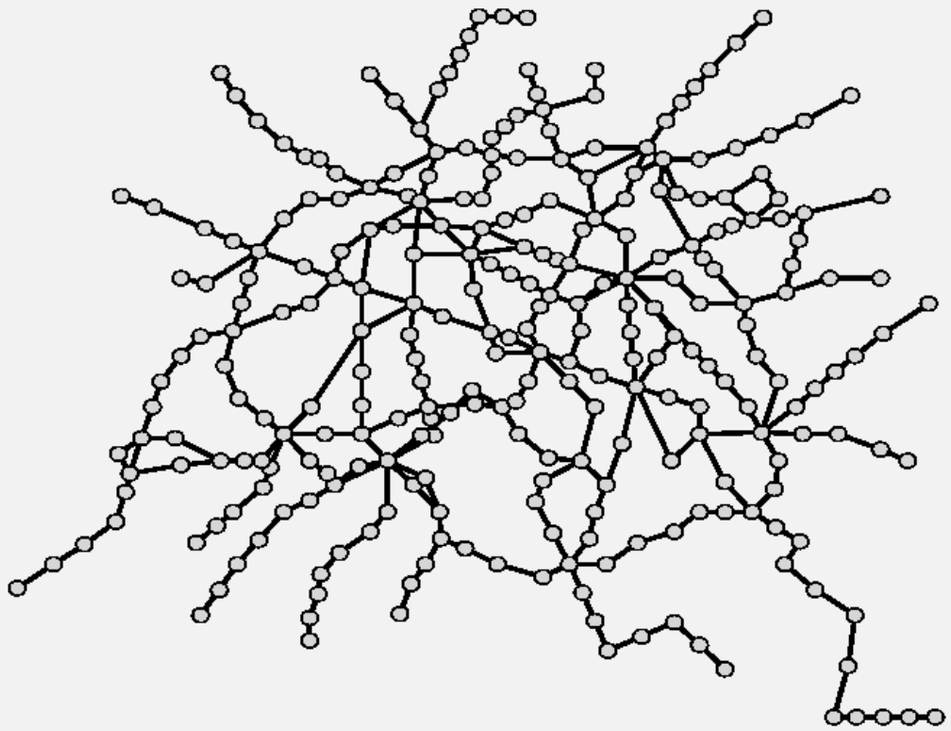
copyrighted content - Do not share

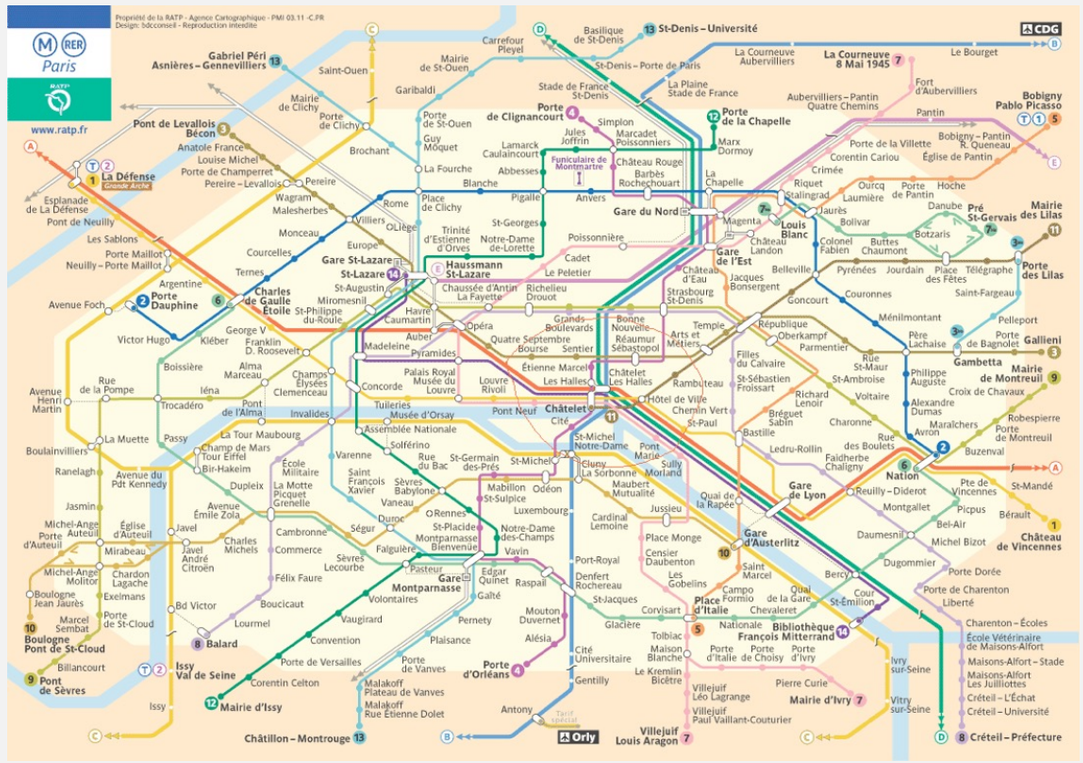
Undirected graphs

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.





copyrighted content – Do not share

Social networks

LO 8A.1, 8A.3

Vertex = person; edge = social relationship.

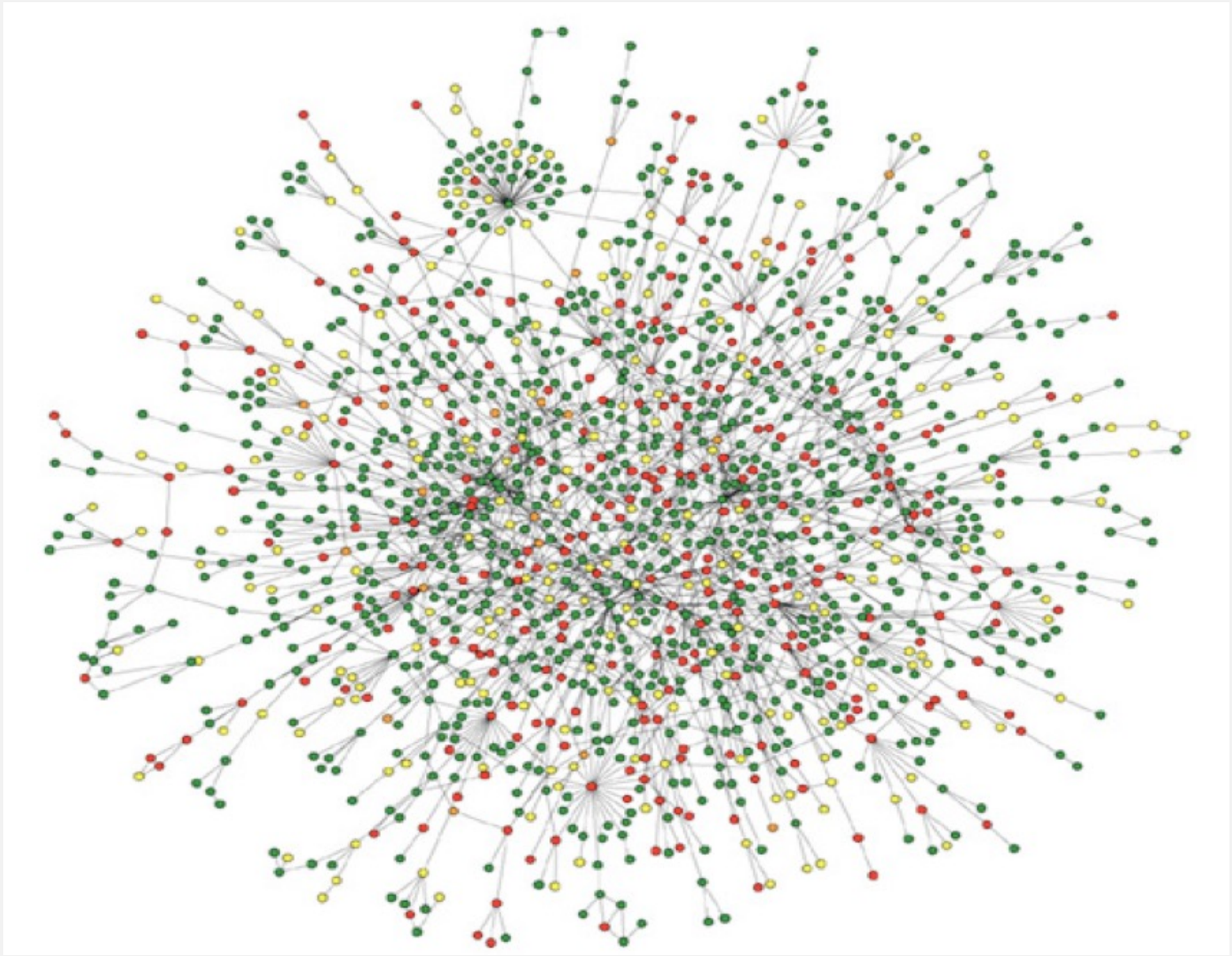


“Visualizing Friendships” by Paul Butler

copyrighted content – Do not share

Protein-protein interaction network

Vertex = protein; edge = interaction.



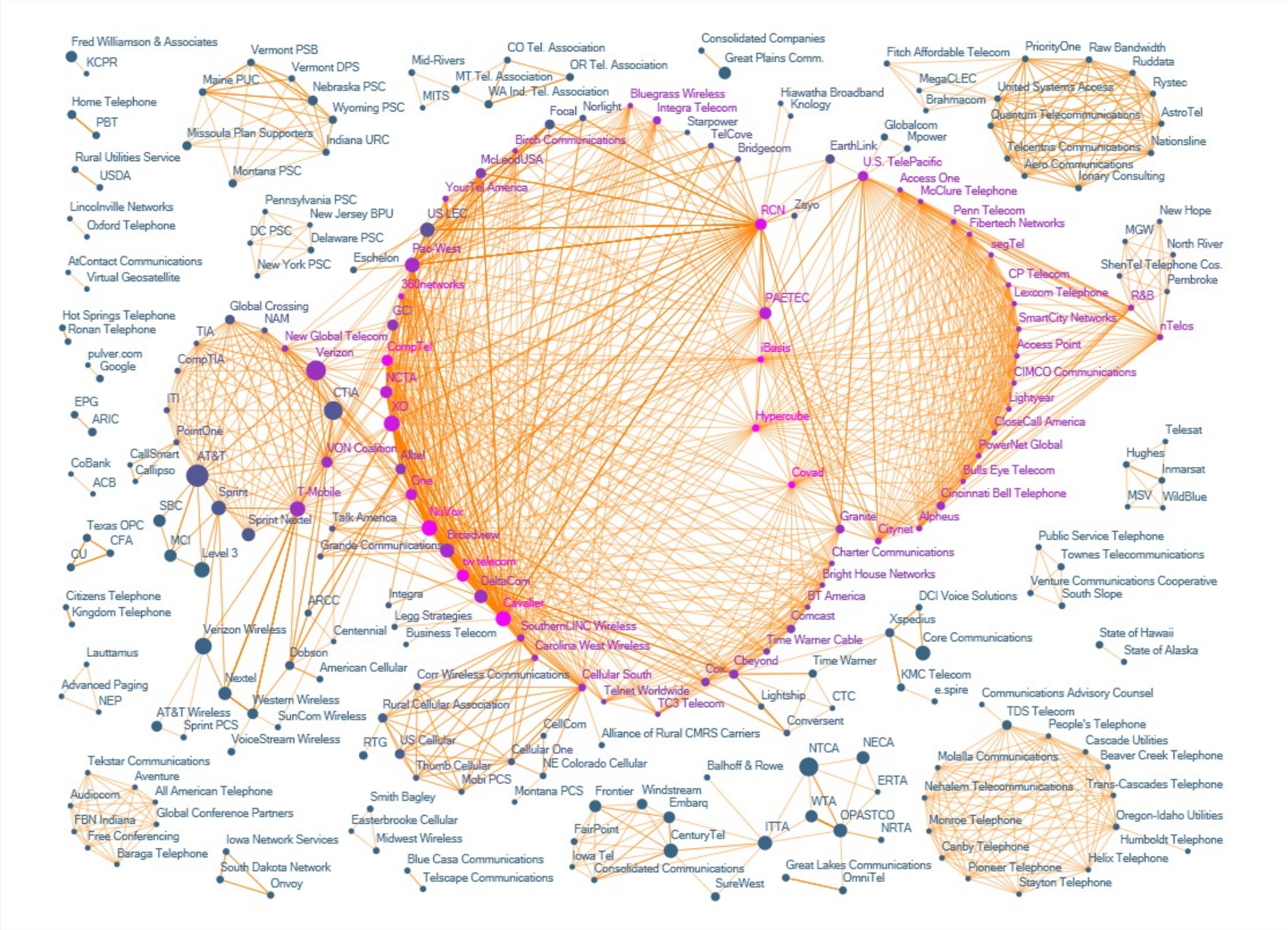
copyrighted content – Do not share

Reference: Jeong et al, Nature Review | Genetics

The evolution of FCC lobbying coalitions

LO 8A.1, 8A.3

Vertex = company; edge = lobbying partner.



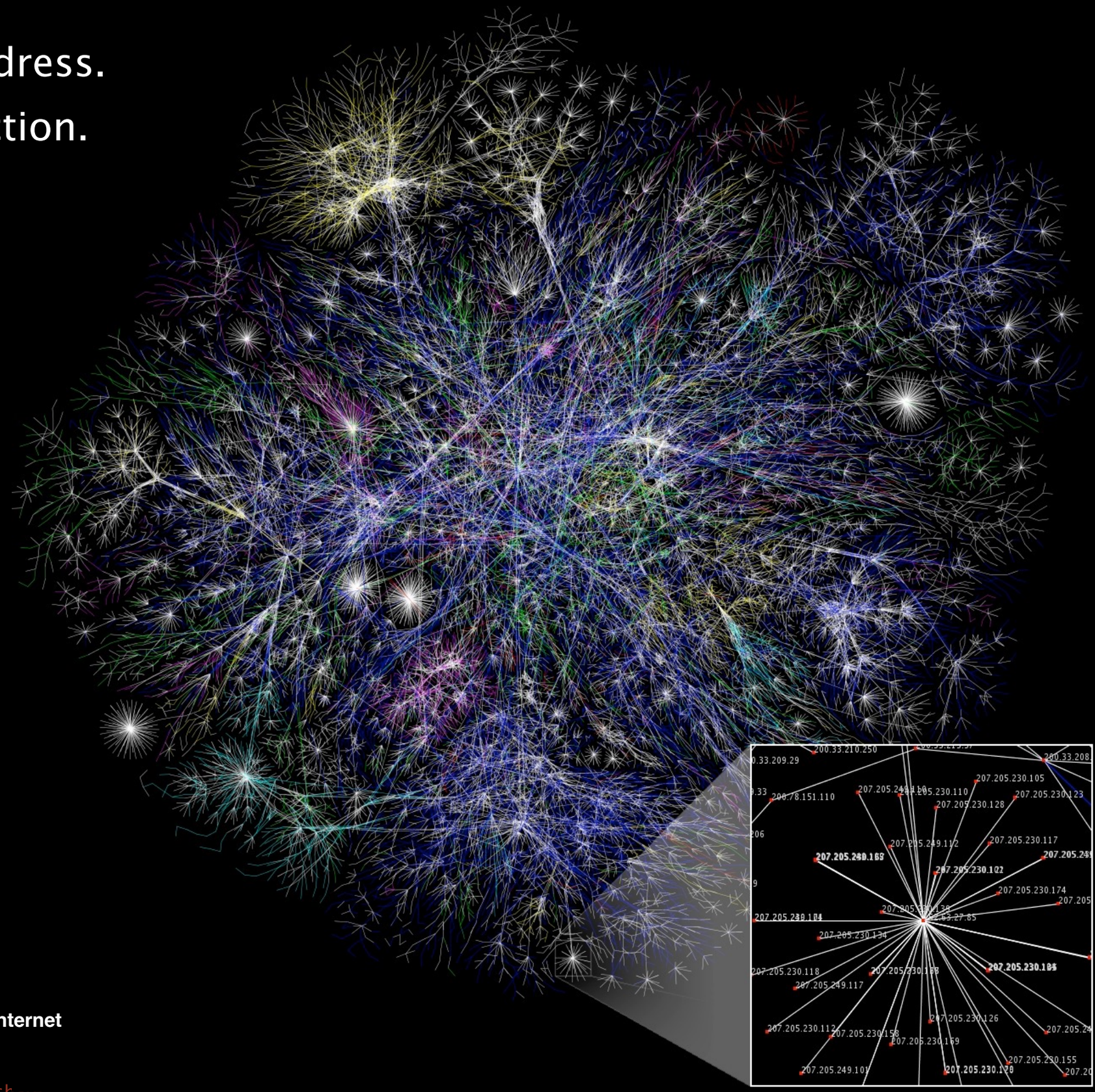
copyrighted content – Do not share

"The Evolution of FCC Lobbying Coalitions" by Pierre de Vries in JoSS Visualization Symposium 2010

The Internet as mapped by the Opte Project

Vertex = IP address.

Edge = connection.



http://en.wikipedia.org/wiki/Internet

copyrighted content – Do not share

Graph applications

LO 8A.3

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
game	board position	legal move
social relationship	person	friendship
neural network	neuron	synapse
protein network	protein	protein–protein interaction
molecule	atom	bond

copyrighted content – Do not share

Graph terminology



Graph. Set of vertices connected pairwise by edges. $G = (V, E)$

Path. Sequence of vertices connected by edges.

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

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

vertex (of degree 3)

edge (incident to vertices 6 and 8)

path between 0 to 2 (of length 3)

cycle (of length 4)

copyrighted content – Do not share

Some graph-processing problems

problem	description
s-t path	*Is there a path between s and t ?*
shortest s-t path	*What is the shortest path between s and t ?*
cycle	*Is there a cycle in the graph ?*
Euler cycle	*Is there a cycle that uses each edge exactly once ?*
Hamilton cycle	*Is there a cycle that uses each vertex exactly once ?*
connectivity	*Is there a path between every pair of vertices ?*
biconnectivity	*Is there a vertex whose removal disconnects the graph ?*
planarity	*Can the graph be drawn in the plane with no crossing edges ?*
graph isomorphism	*Are two graphs isomorphic?*

Challenge. Which graph problems are easy? difficult? intractable?

copyrighted content - Do not share

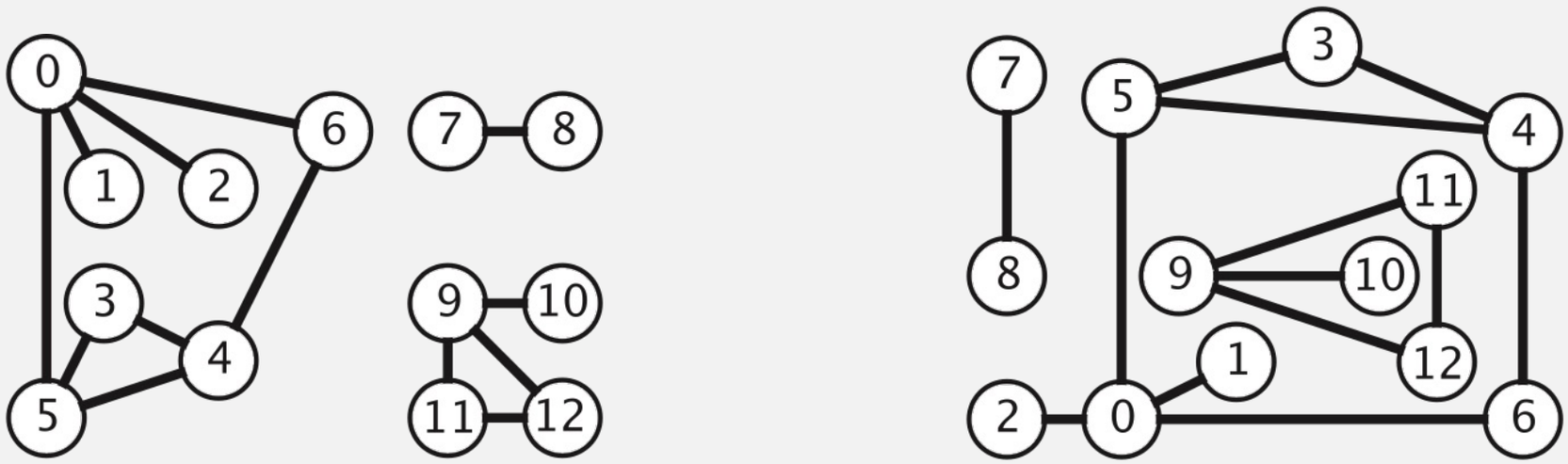
UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

copyrighted content - Do not share

Graph representation

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



two drawings of the same graph

Caveat. Intuition can be misleading.

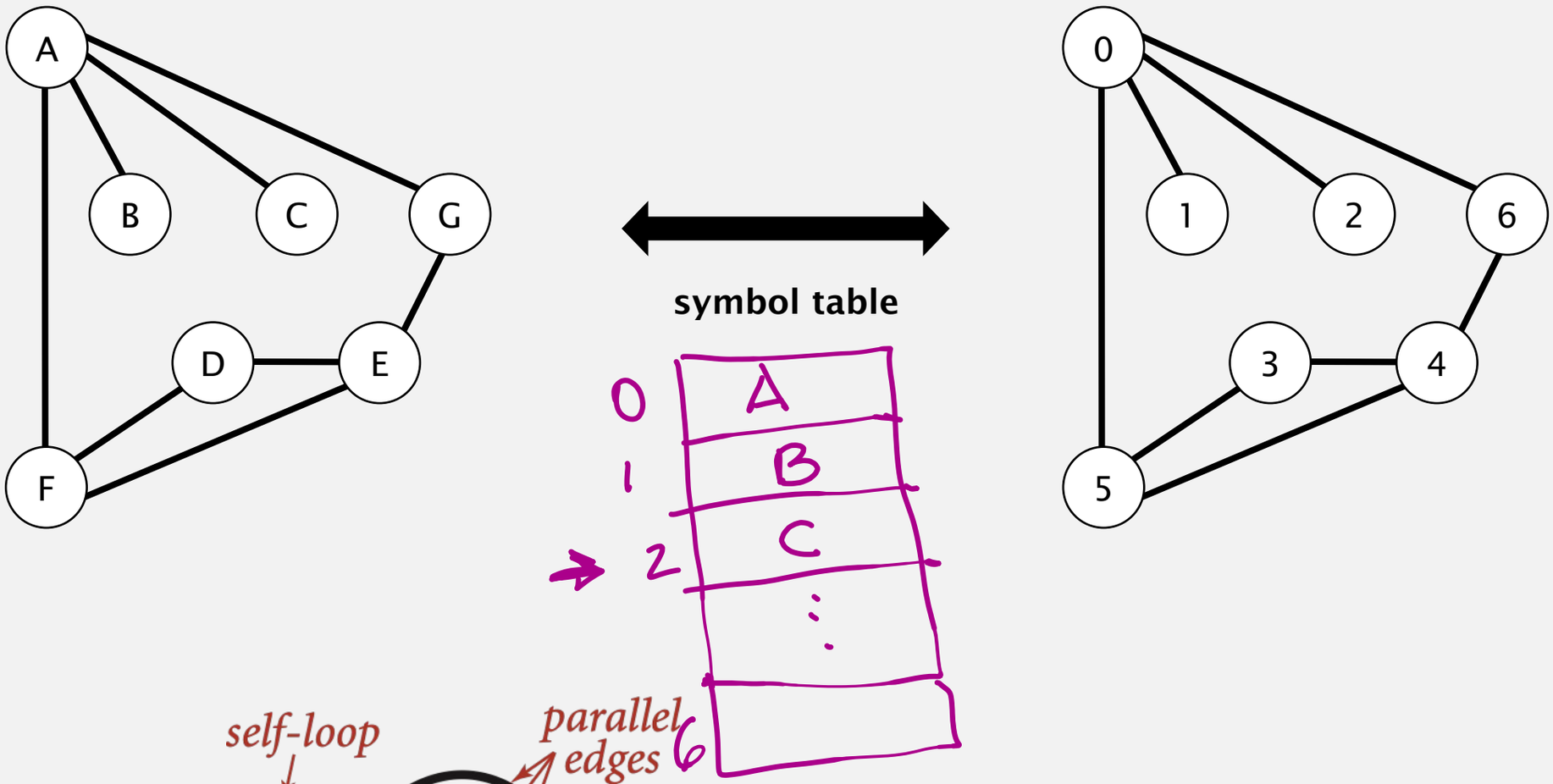
copyrighted content – Do not share

Graph representation

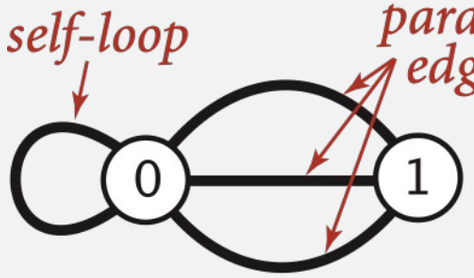
Vertex representation.

V vertices

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



Anomalies.



Does not happen in CS 112

copyrighted content – Do not share

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;
}
```

copyrighted content – Do not share

Graph representation: adjacency matrix

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

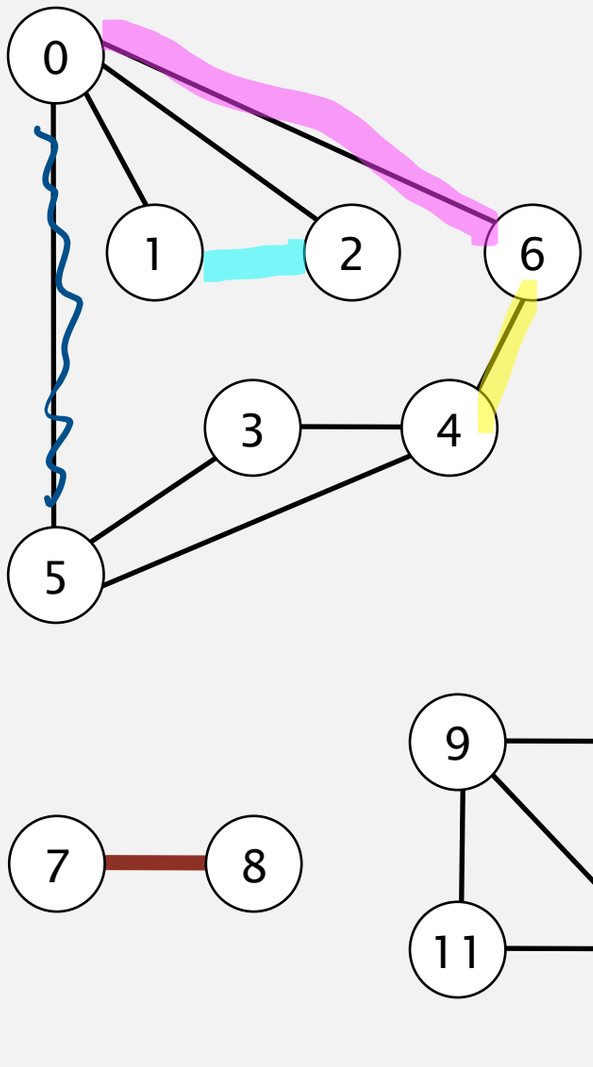
for each edge v–w in graph: adj[v][w] = adj[w][v] = true.

$G = (V, E)$

$G = (13, 13)$

TO

FROM



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

Space consumption: V^2

copyrighted content – Do not share

Graph representation: adjacency matrix

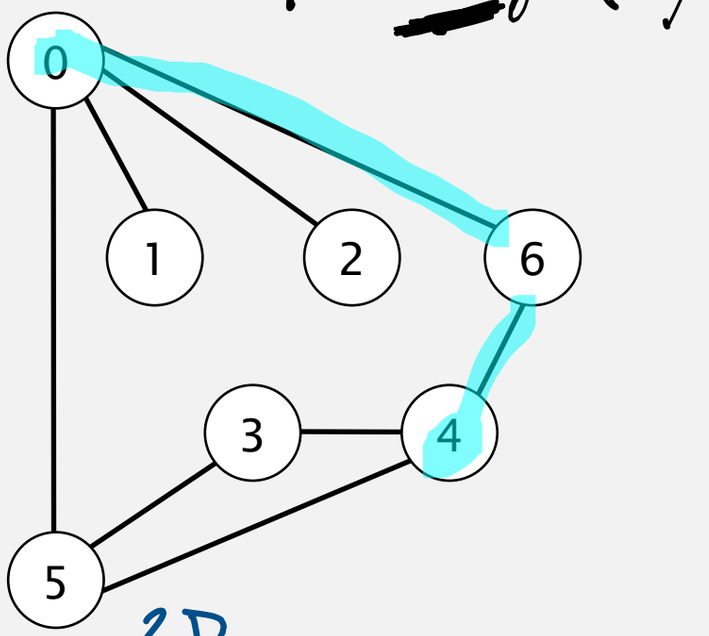
V checks

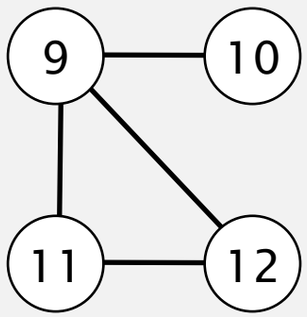
O(V)

In a Graph with V vertices and E edges.

What is the running time of G.adj(w) (finds the neighbors of vertex w)?

G.adj(6)





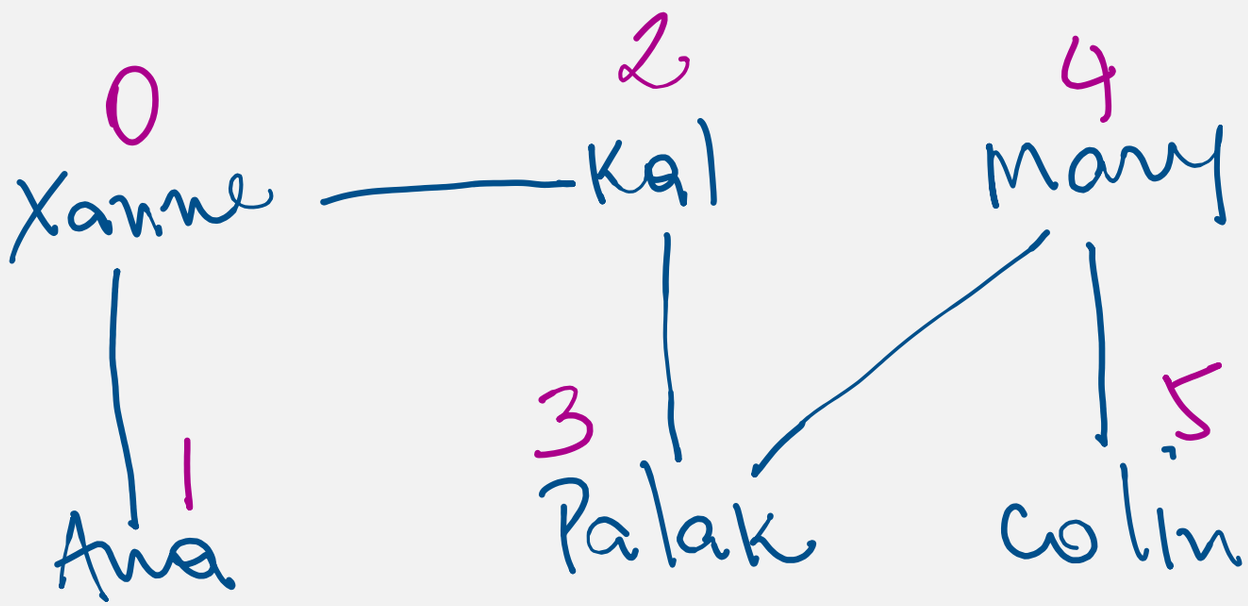
2D Ragged array could be used instead of 2D Complete array

	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

Space consumption: V^2

copyrighted content – Do not share

LinkedIn Example



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

TO

FROM



Unconnected

two islands of people

copyrighted content - Do not share

Undirected graphs: quiz 2

Which is order of growth of running time of the following code fragment if the graph uses the adjacency-matrix representation, where V is the number of vertices and E is the number of edges?

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

For every vertex v of G

For every neighbor of v

print each edge twice

A. V

B. $E + V$

C. V^2

D. $V E$

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

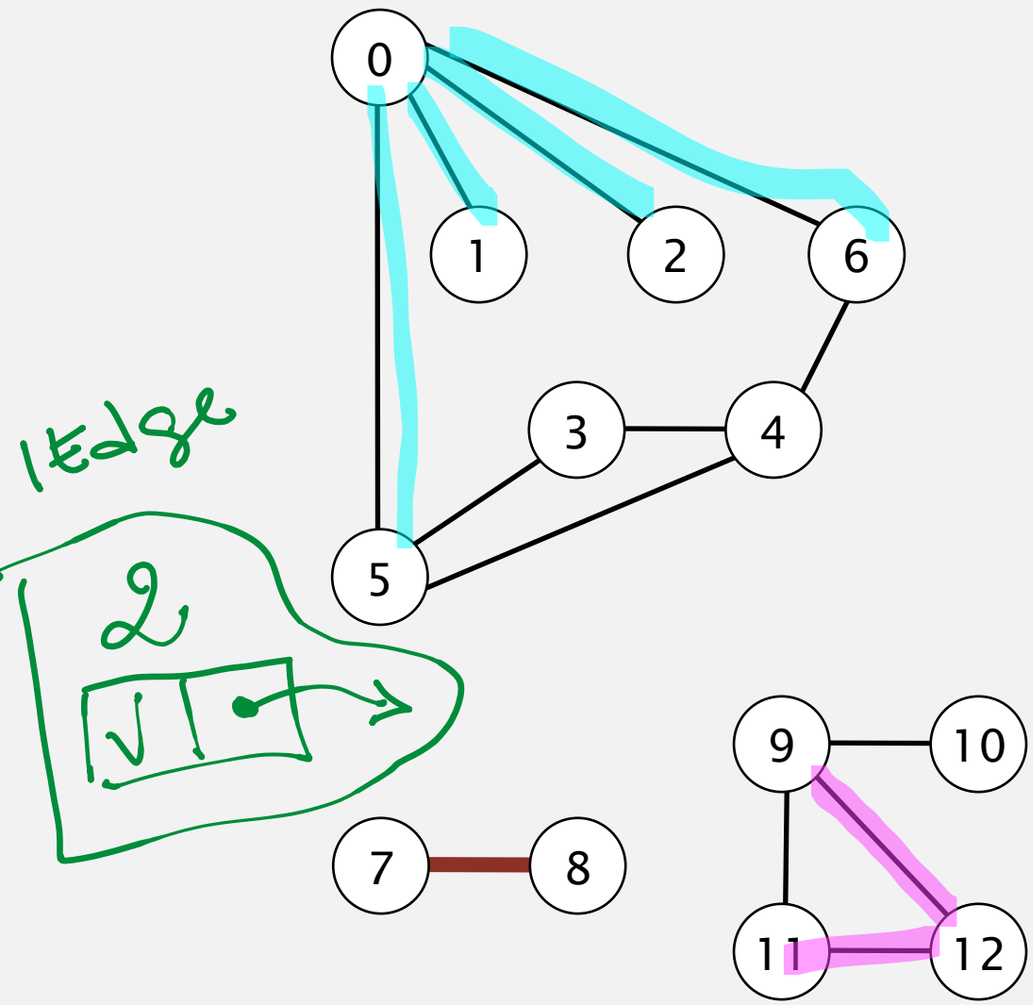
adjacency-matrix representation

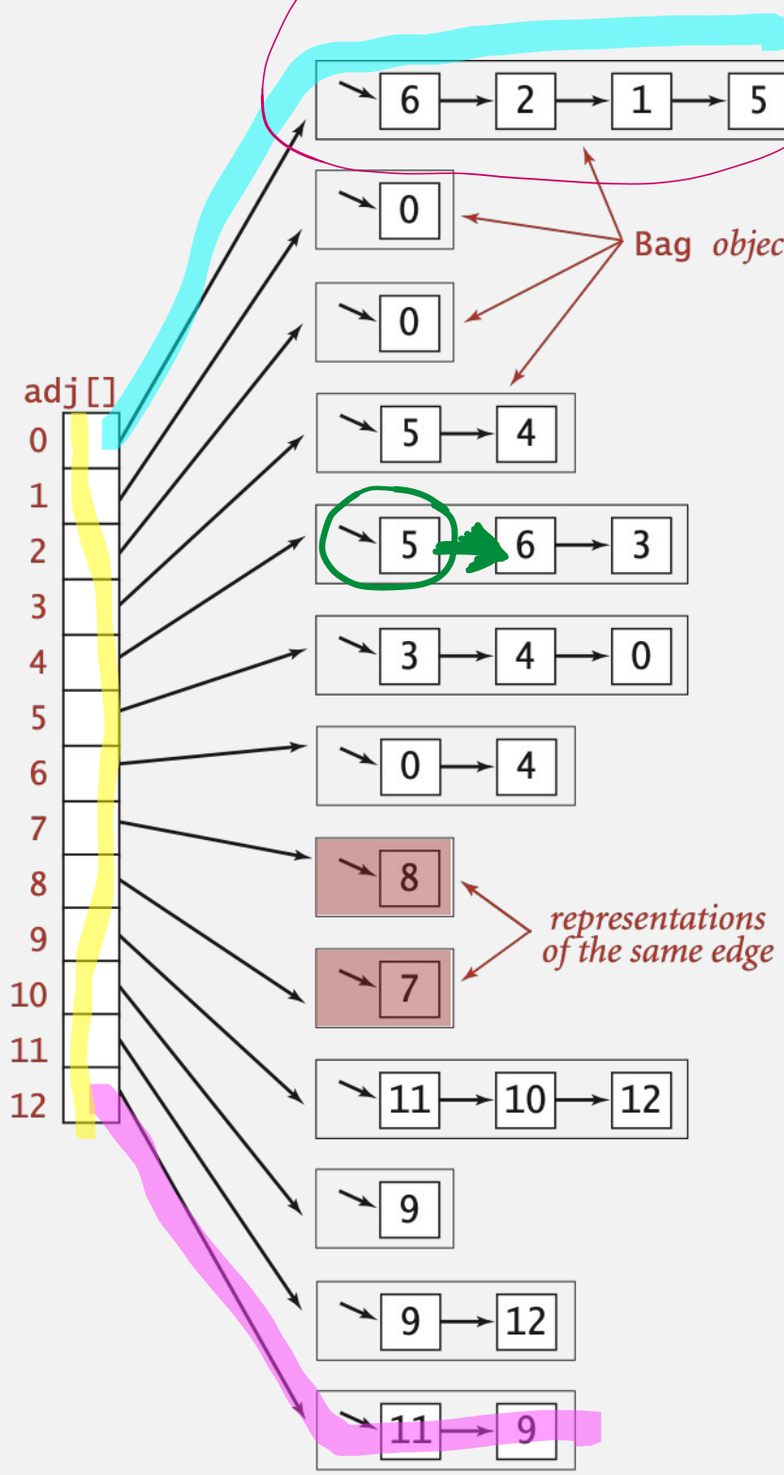
copyrighted content - Do not share

Graph representation: adjacency lists

all vertices are neighbors to ∅

Maintain vertex-indexed array of lists.





Assuming a linked list to store the adjacency list.

Space consumption: V + (2E) * 2 ← each edge is stored twice.

2 units per node: the vertex and the next pointer.

copyrighted content – Do not share

Graph representation: adjacency lists

In a Graph with V vertices and E edges.

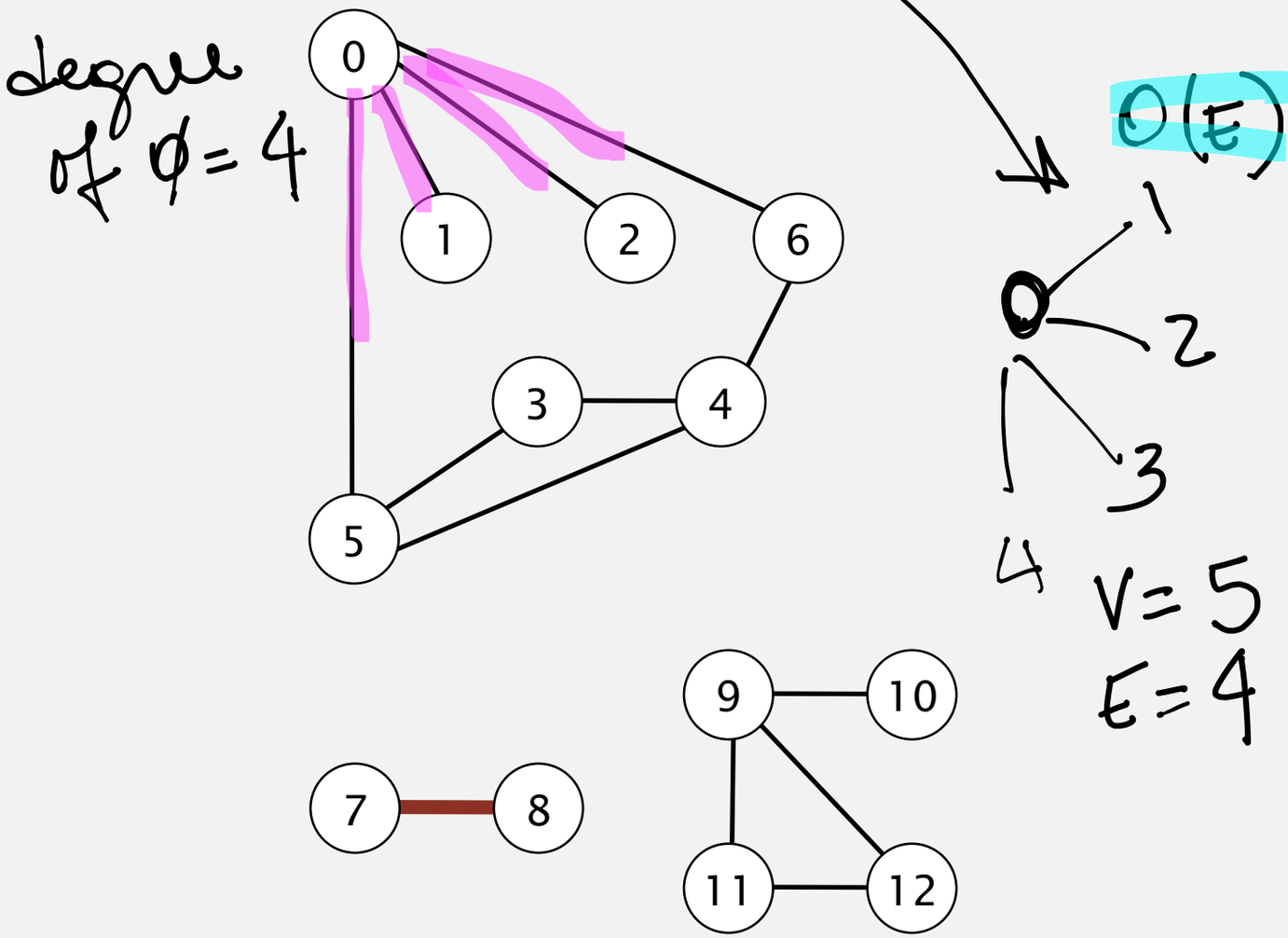
What is the worst-case running time of G.adj(w)

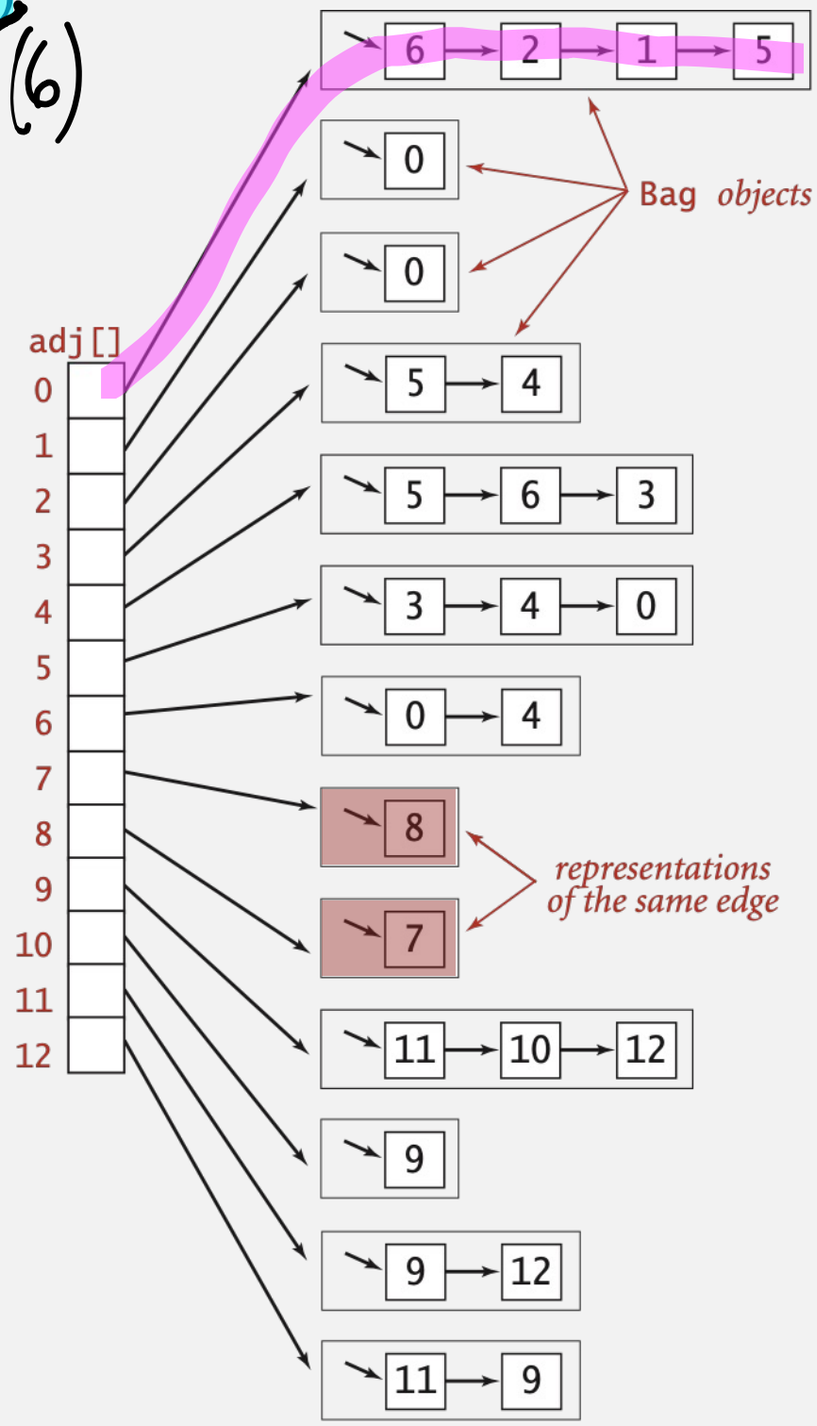
(finds the neighbors of vertex w)?

G.adj(6)

O(E)

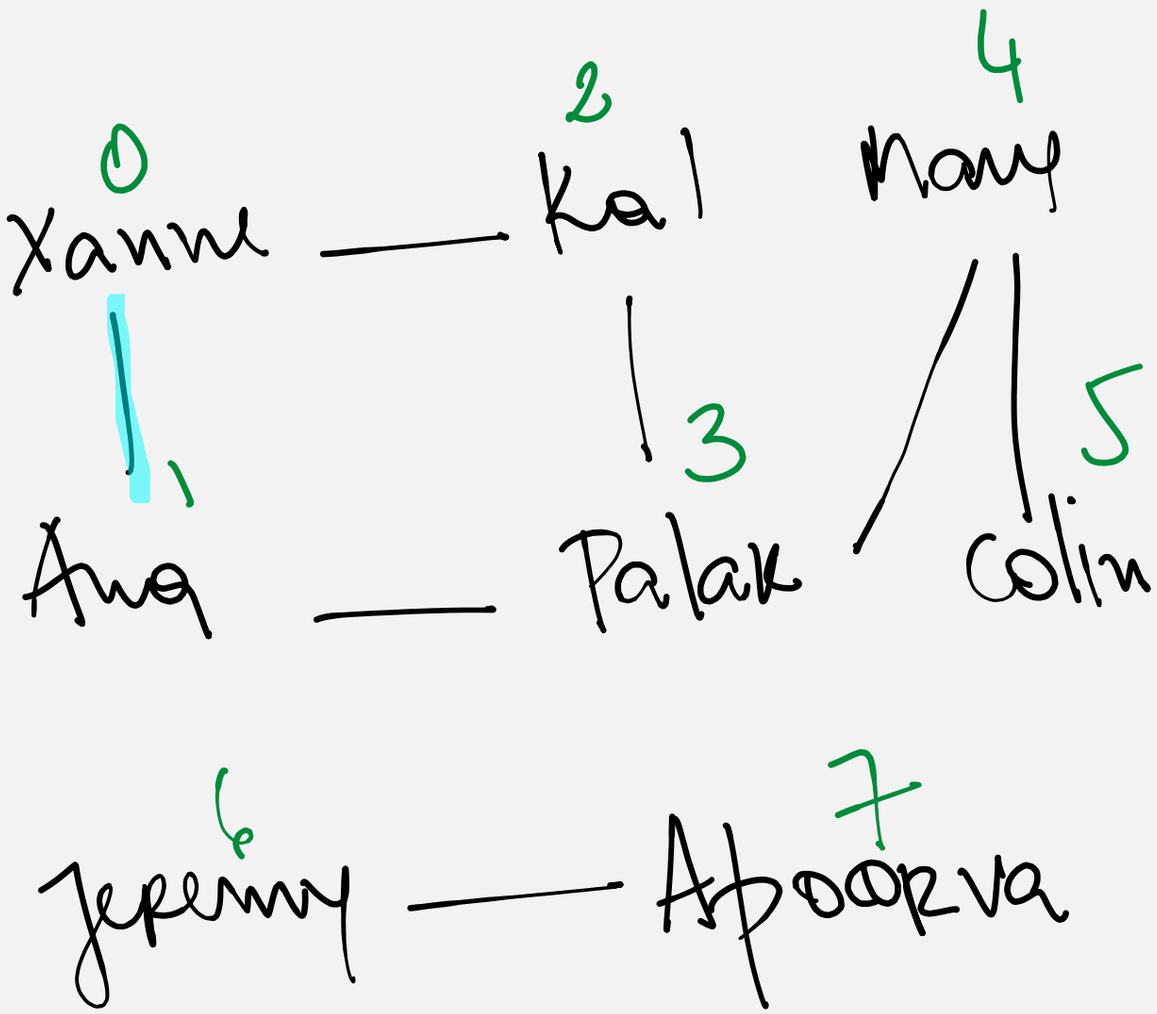
degree of 4 = 4





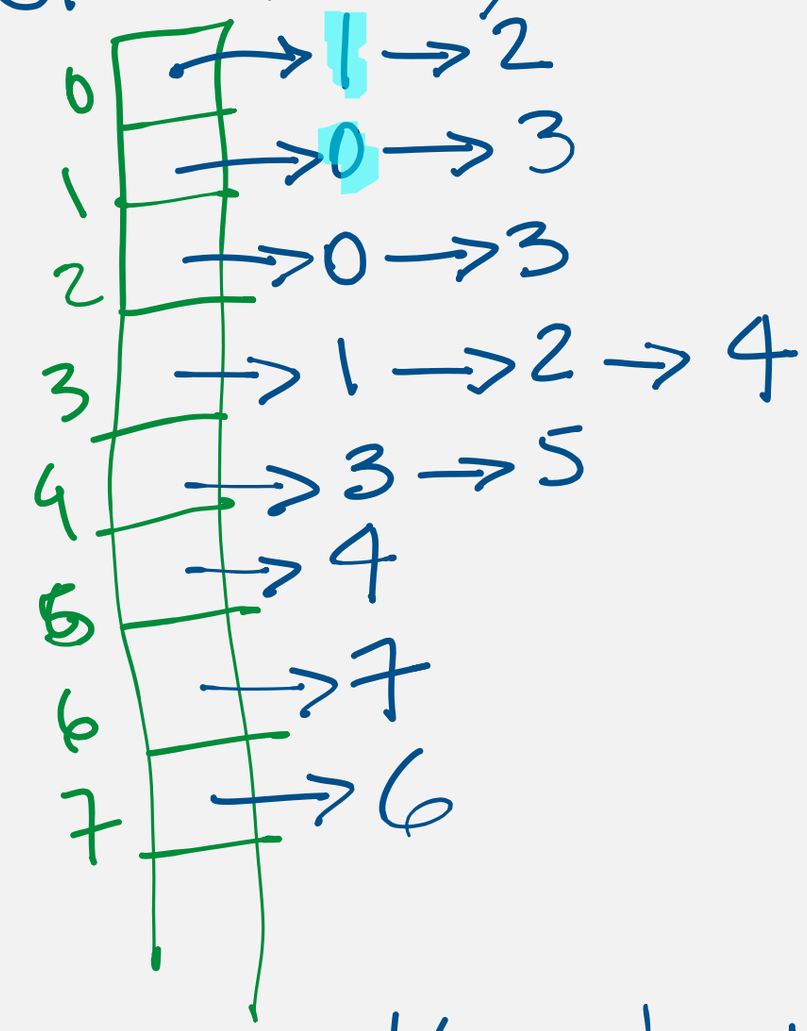
copyrighted content – Do not share

LinkedIn Example



$G = (V, E)$

$G = (8, 7)$



14 edges in the structure

copyrighted content – Do not share

Undirected graphs: quiz 3

Which is order of growth of running time of the following code fragment if the graph uses the adjacency-lists representation, where V is the number of vertices and E is the number of edges?

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

print each edge twice

For every vertex v
For every neighbor w of v

A. V

B. $E + V$

C. V^2

D. $V E$

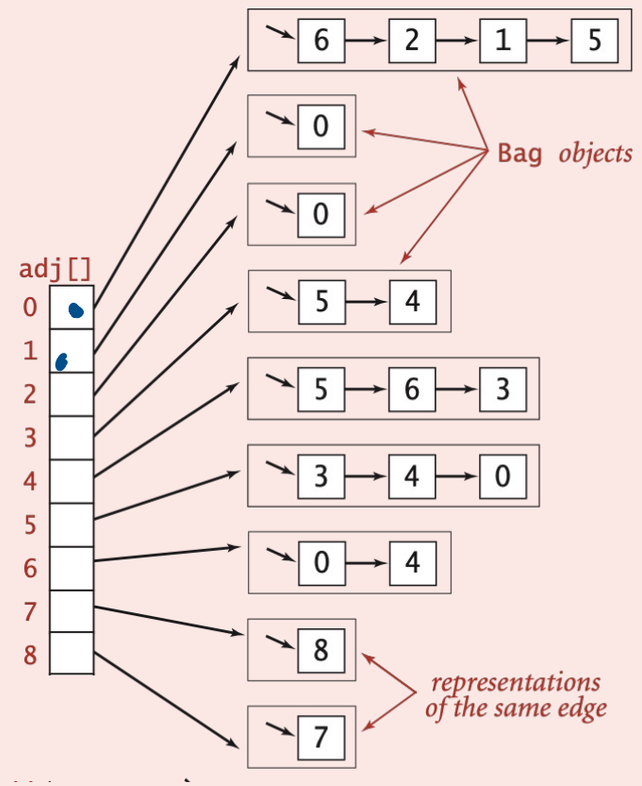
$O(V+E)$

V + $2E$

↑ array ↑ all lists

zero edge

$V+E \Rightarrow V+\emptyset \Rightarrow V$



copyrighted content – Do not share

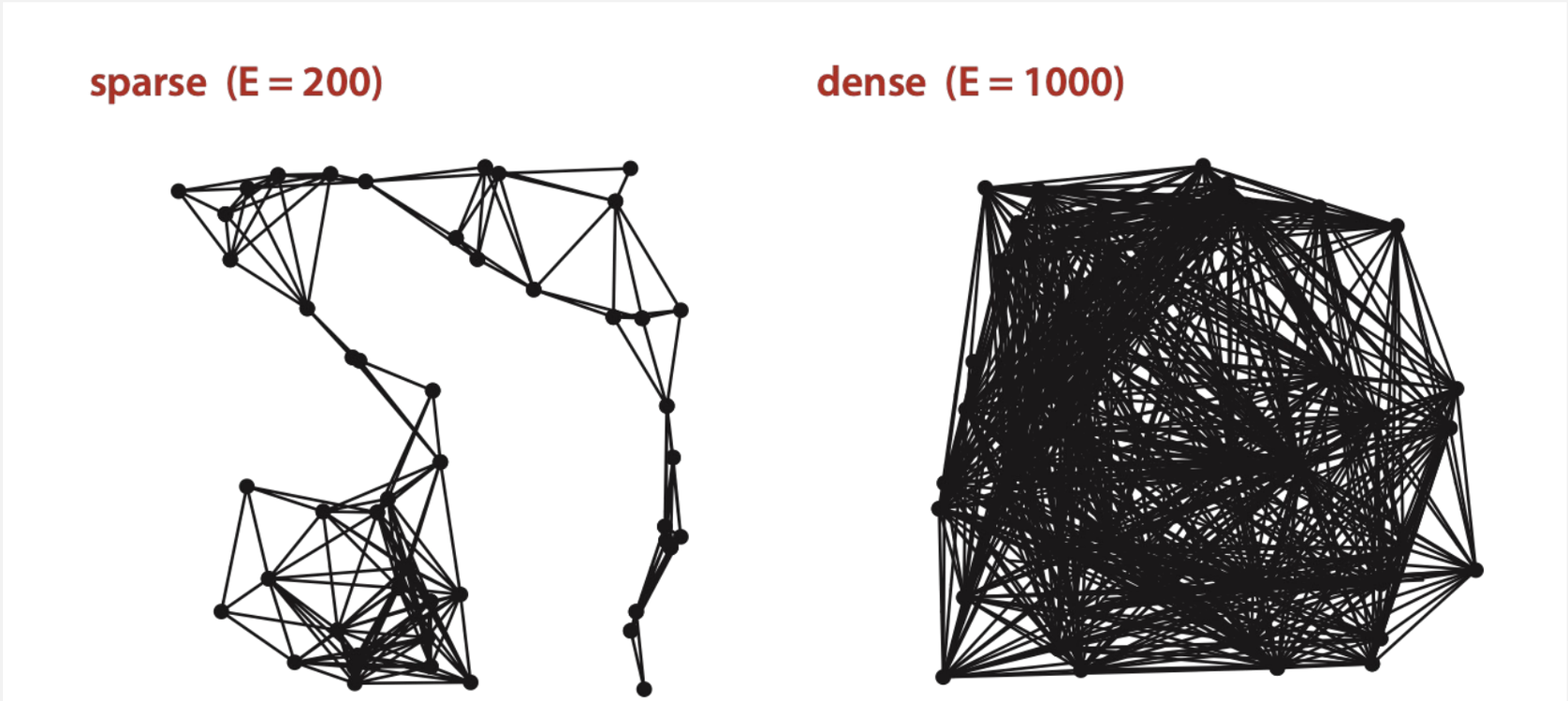
Graph representations

In practice. Use adjacency-lists representation.

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

proportional to V edges

proportional to V^2 edges



Two graphs (V = 50)

copyrighted content – Do not share

Graph representations

LO 8A.8

In practice. Use adjacency-lists representation.

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

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

copyrighted content – Do not share

Adjacency-list graph representation: Java implementation

LO 8A.7

```
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

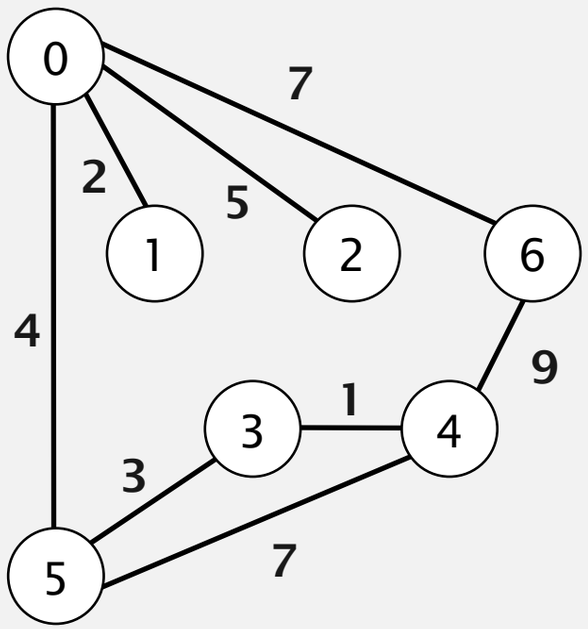
add edge v-w (parallel edges and self-loops allowed)

iterator for vertices adjacent to v

copyrigh

Weighted Undirected Graphs

Edges have a cost associated with them.



The cost of path 0, 6, 4 is 16

The cost of path 0, 5, 4 is 11

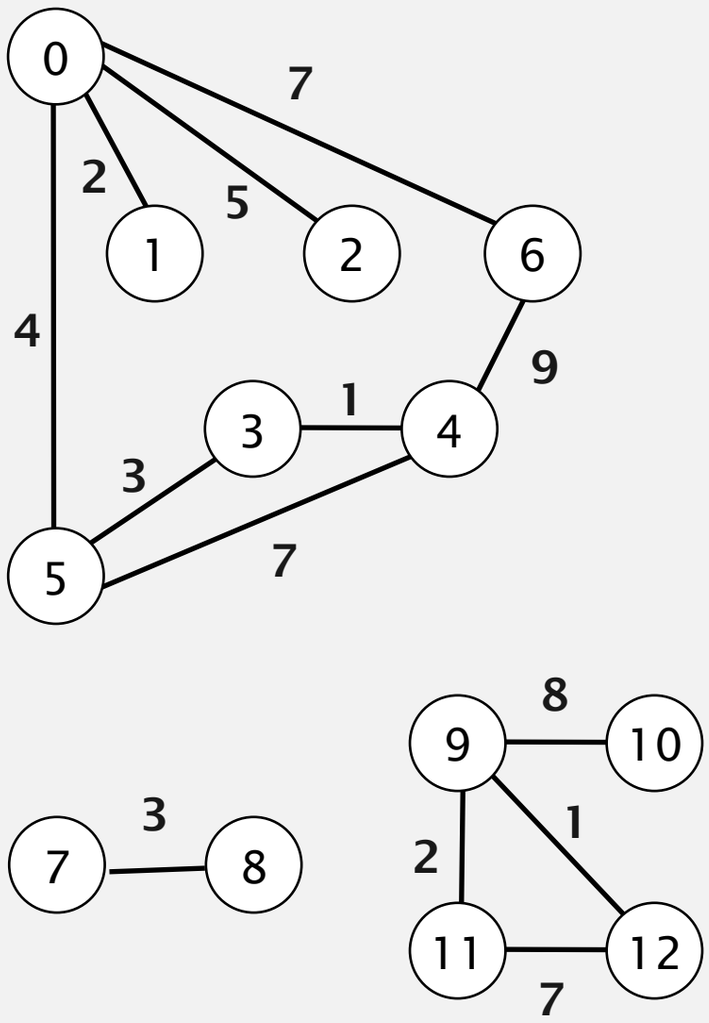
The cost of path 0, 5, 3, 4 is 8

copyrighted content – Do not share

Graph representation: adjacency matrix

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

for each edge v–w in graph: adj[v][w] = adj[w][v] = **edge cost or value**.



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

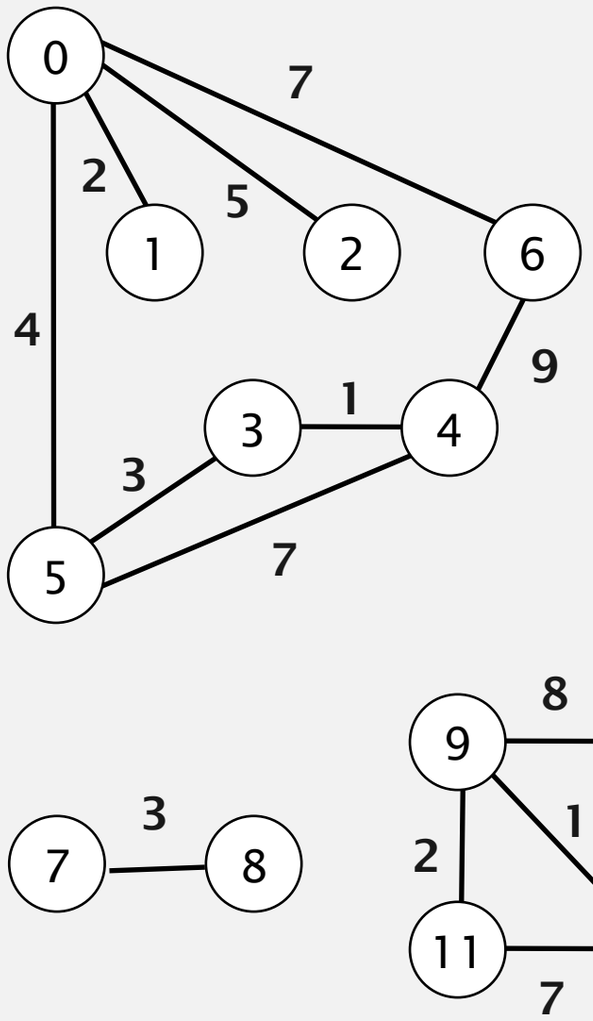
Space consumption: V^2

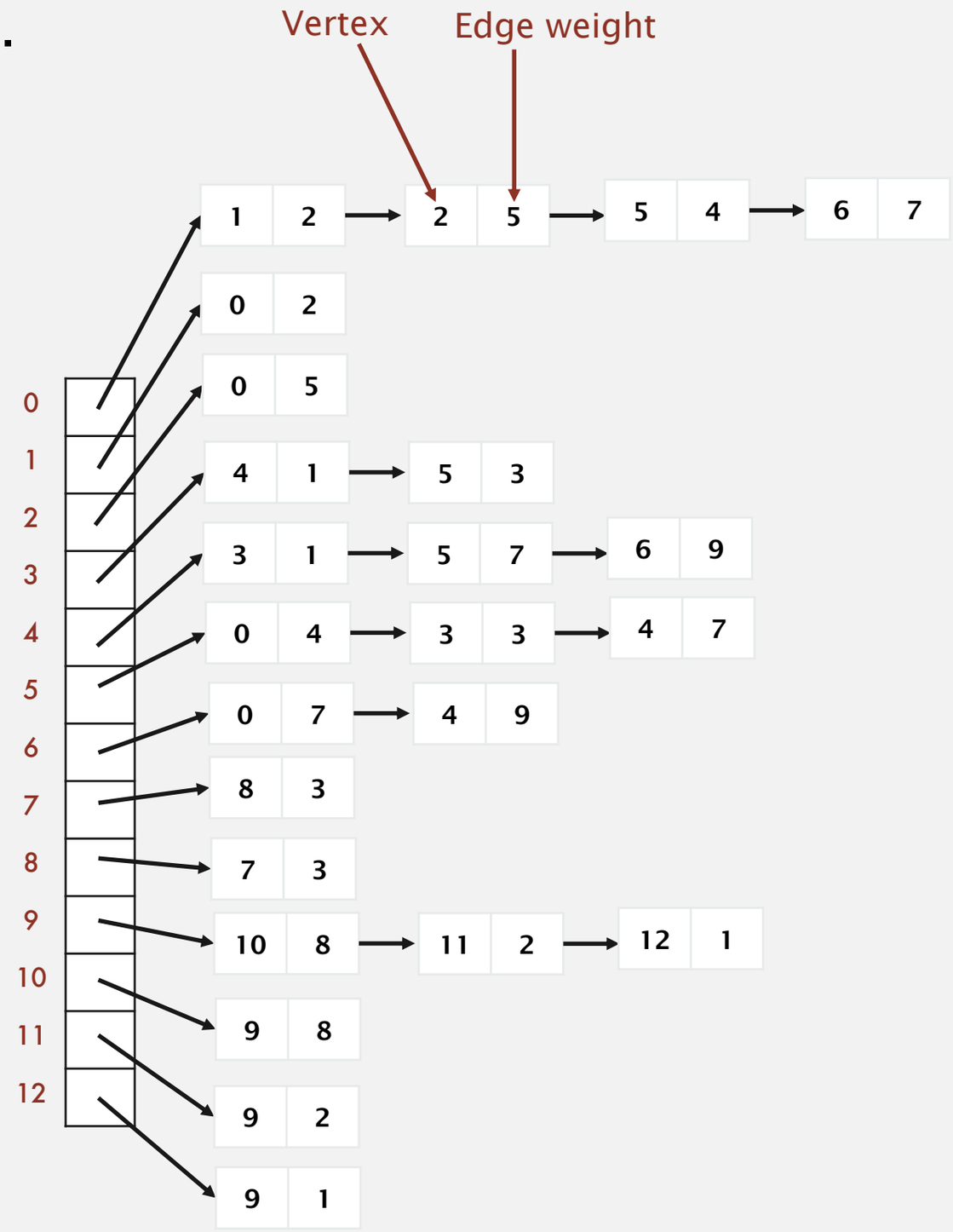
copyrighted content – Do not share

Graph representation: adjacency lists

Maintain vertex-indexed array of lists.

Add edge weight to node.





Assuming a linked list to store the adjacency list.

Space consumption: **V + (3E) * 2** ← each edge is stored twice.

3 units per node: the vertex, the edge weight, and the next pointer.

copyrighted content – Do not share

Undirected graphs: quiz 4

What is the maximum number of edges on a Graph of V vertices?

A. $V * (V - 1) / 2$

B. $E + V$

C. V^2

D. $V E$

E. *I don't know.*

copyrighted content – Do not share

UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

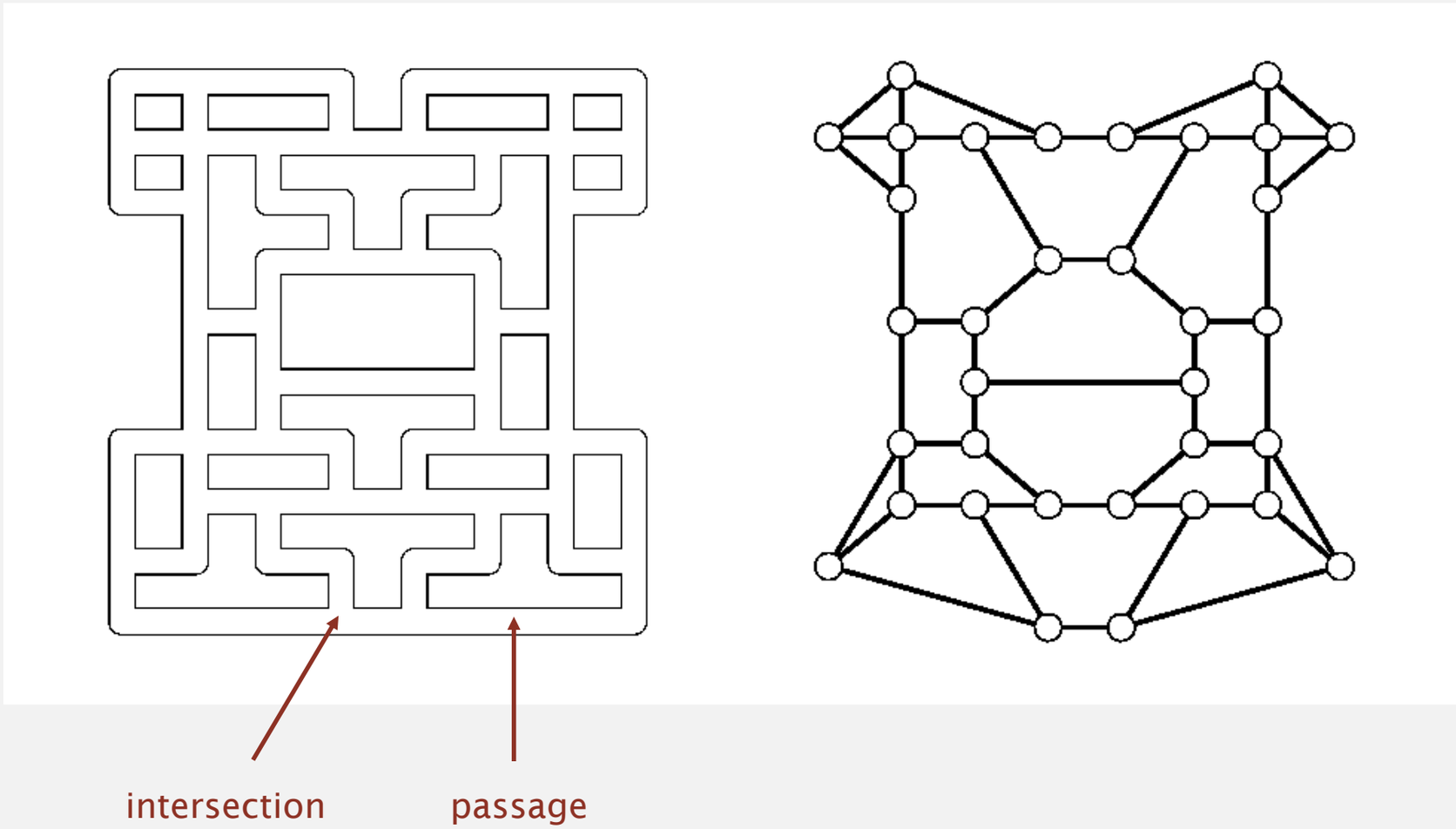
http://ds.cs.rutgers.edu

copyrighted content - Do not share

Maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.



Goal. Explore every intersection in the maze.

copyrighted content – Do not share

Maze exploration: National Building Museum



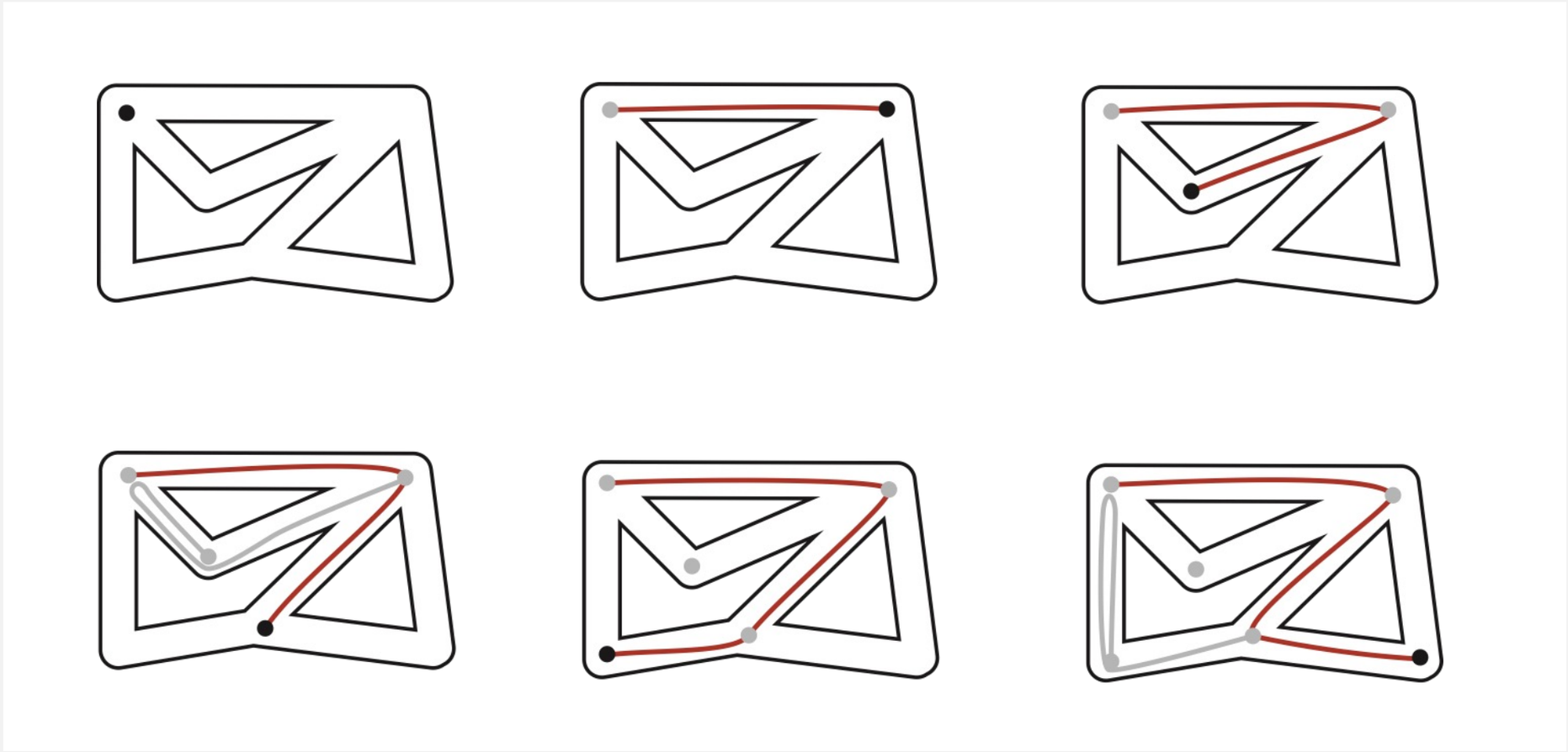
http://www.smithsonianmag.com/travel/winding-history-maze-180951998/?no-ist

copyrighted content – Do not share

Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each newly discovered intersection and passage.
- Retrace steps when no unmarked options.



copyrighted content – Do not share

Trémaux maze exploration

Algorithm.

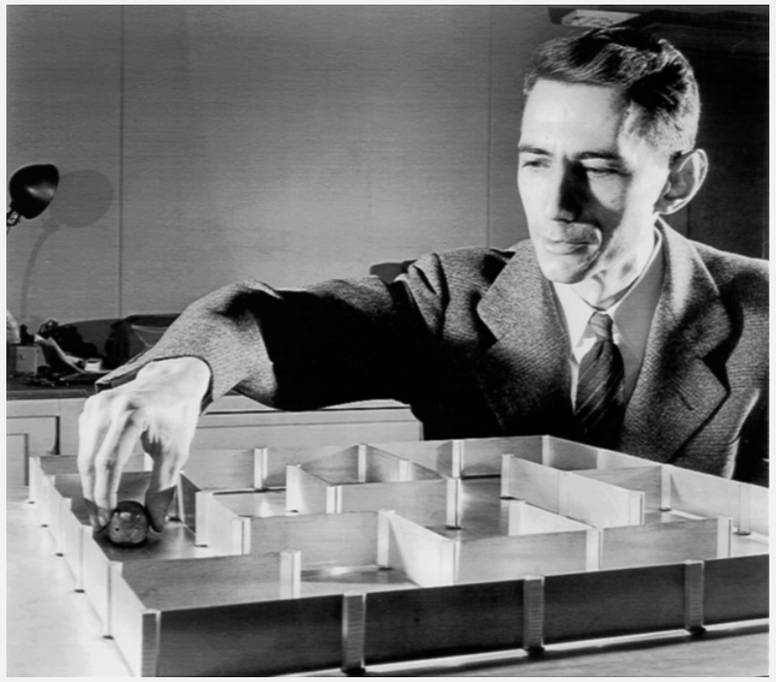
- Unroll a ball of string behind you.
- Mark each newly discovered intersection and passage.
- Retrace steps when no unmarked options.

First use? Theseus entered Labyrinth to kill the monstrous Minotaur; Ariadne instructed Theseus to use a ball of string to find his way back out.



The Cretan Labyrinth (with Minotaur)

http://commons.wikimedia.org/wiki/File:Minotaurus.gif

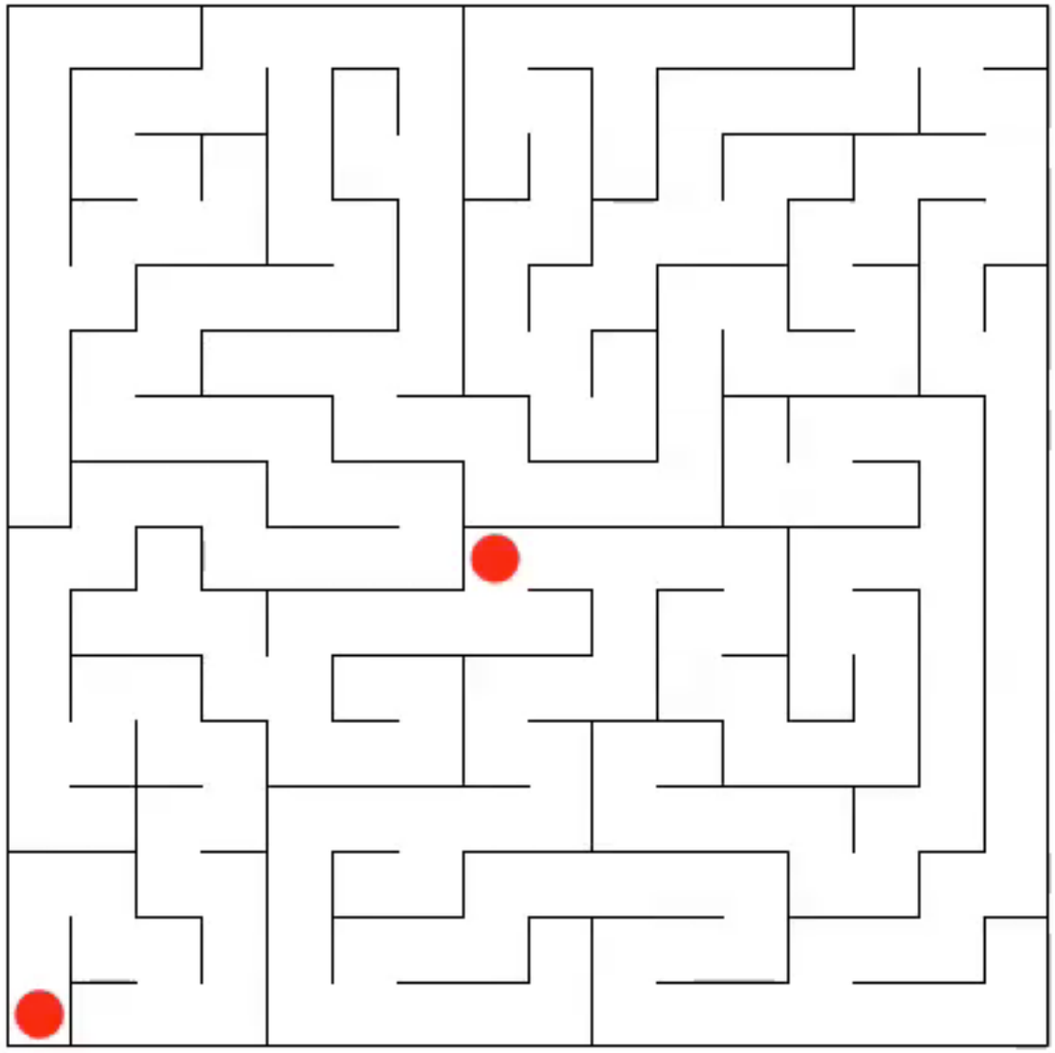


Claude Shannon (with electromechanical mouse)

http://www.corp.att.com/attlabs/reputation/timeline/16shannon.html

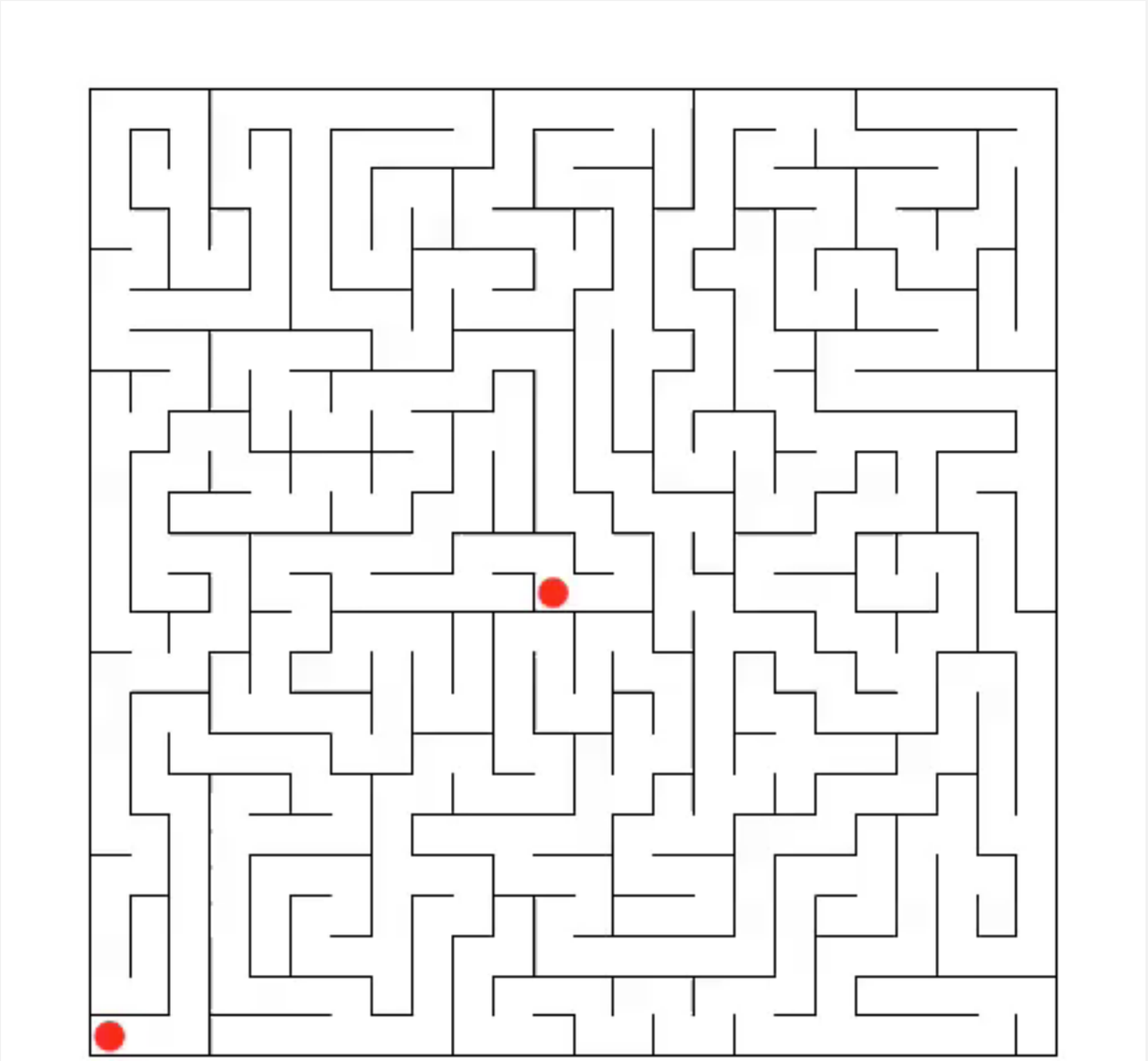
copyrighted content – Do not share

Maze exploration: easy



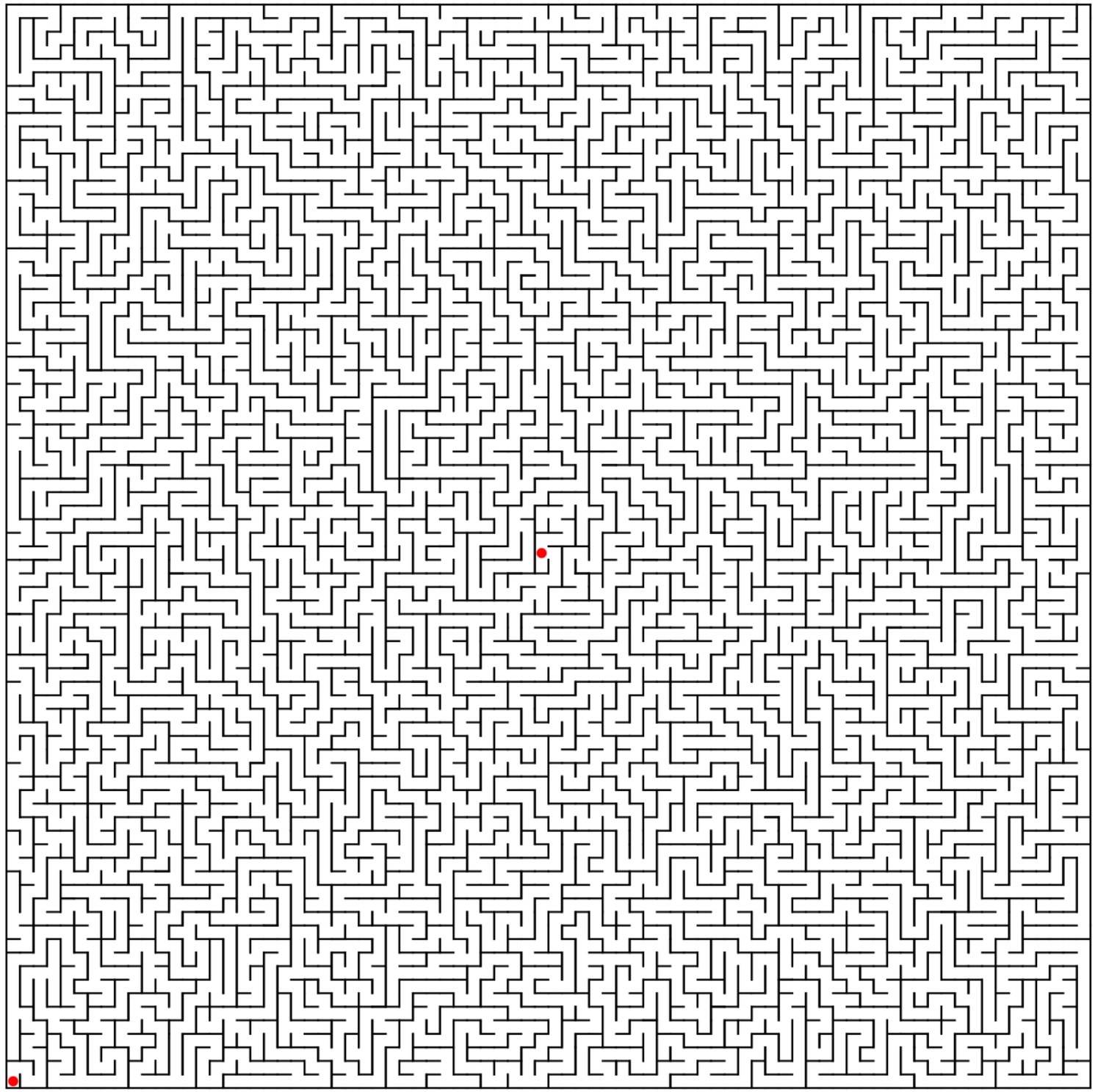
copyrighted content – Do not share

Maze exploration: medium



copyrighted content – Do not share

Maze exploration: challenge for the bored



copyrighted content – D

Depth-first search

Goal. Systematically traverse a graph.

Idea. Mimic maze exploration. ← function-call stack plays role of ball of string

DFS (to visit a vertex v)

Mark vertex v.

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?

copyrighted content – Do not share

Depth-first search: data structures

To visit a vertex v :

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

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
- Function-call stack for recursion.

copyrighted content – Do not share

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 v connected to s

edgeTo[v] = previous vertex

on path from s to v

initialize data structures

find vertices connected to s

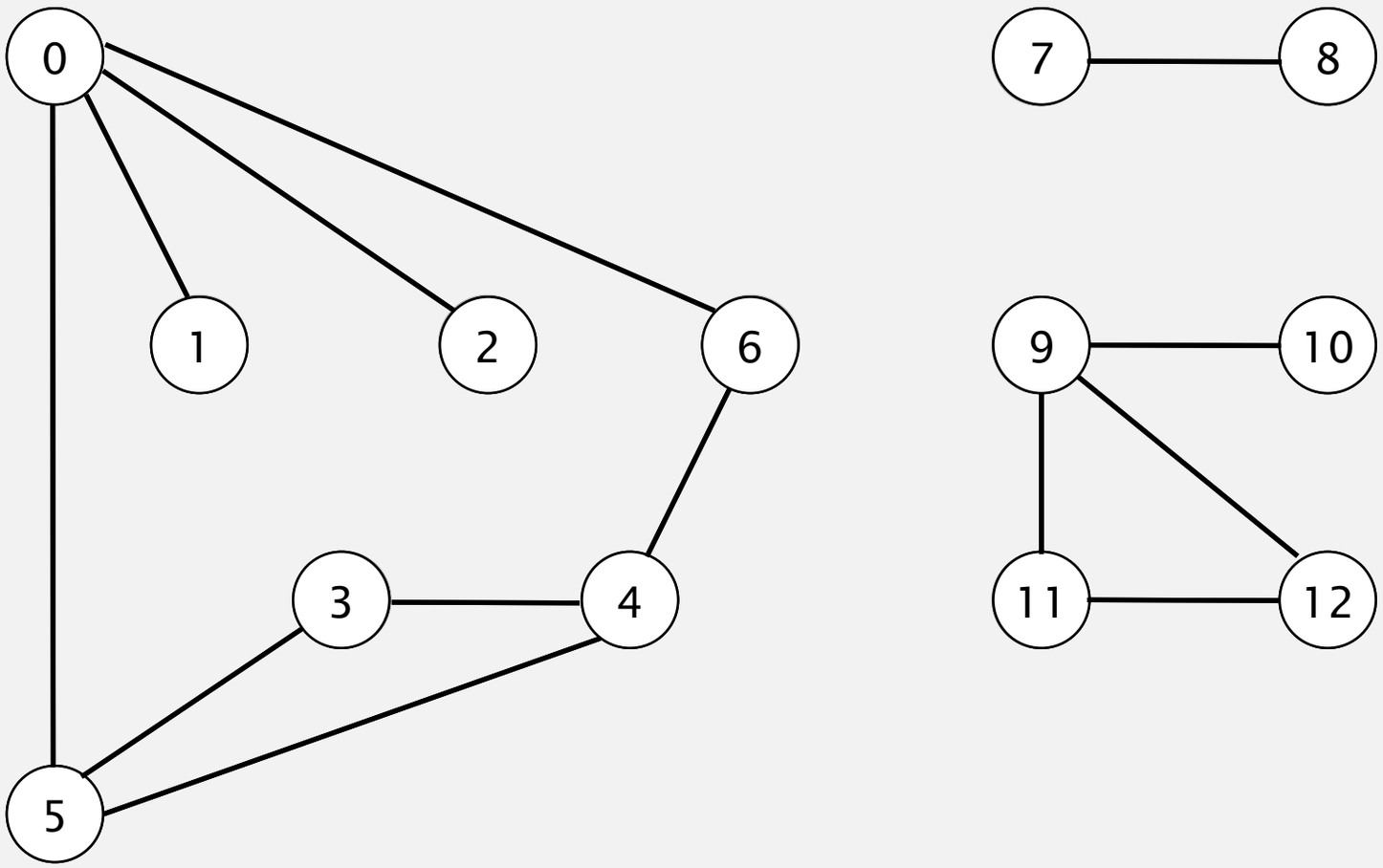
recursive DFS does the work

copyrig

Depth-first search demo

To visit a vertex v :

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



graph G

tinyG.txt

V → 13
E → 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

copyrighted content – Do not share

GRAPH TRAVERSE

Depth-first search demo

To visit a vertex v :

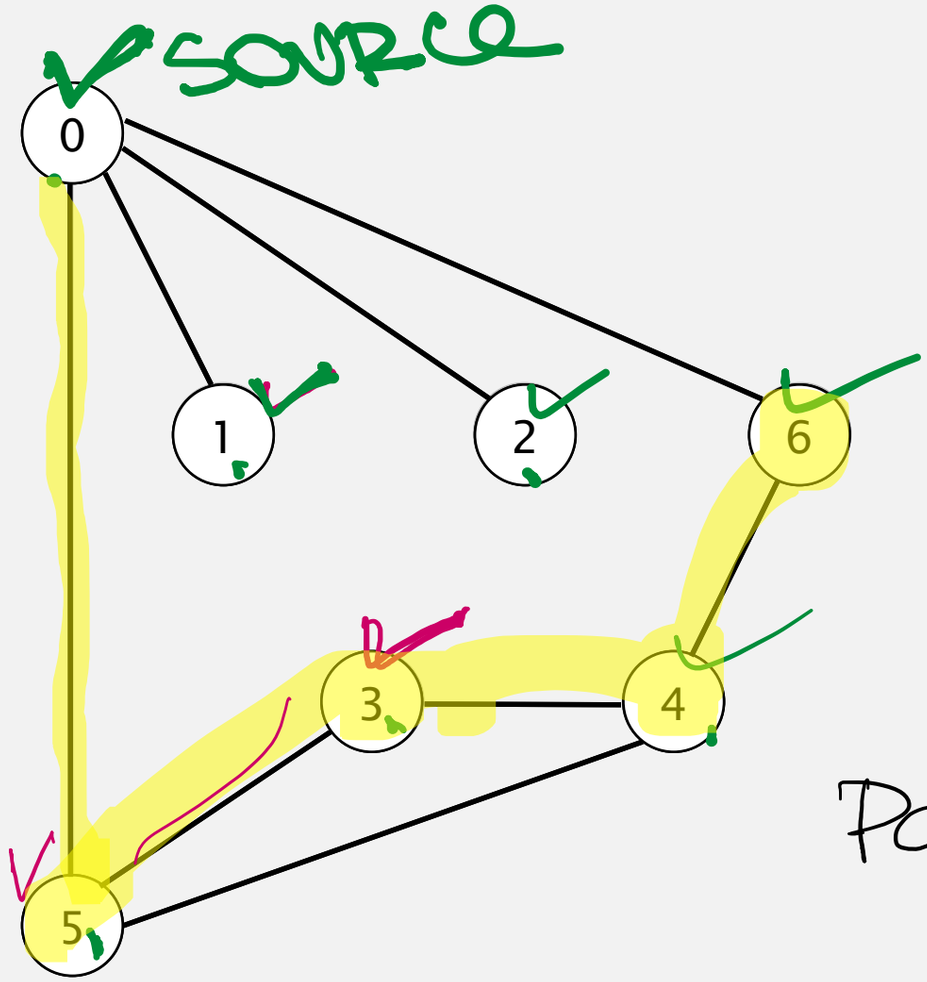
- Mark vertex v.
- Recursively visit all unmarked vertices adjacent to v.

```
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);
    }
}
```

V times

E times

O(V+E)





DFS(G,0)
DFS(G,1)
DFS(G,2)
DFS(G,5)
DFS(G,3)
DFS(G,4)
DFS(G,6)

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

Path ∅ → 6

S

0 5 3 4 6

0, 5, 3, 4, 6

Edges stored in ascending order

graph G

copyrighted content – Do not share

DRIVER to traverse the entire graph

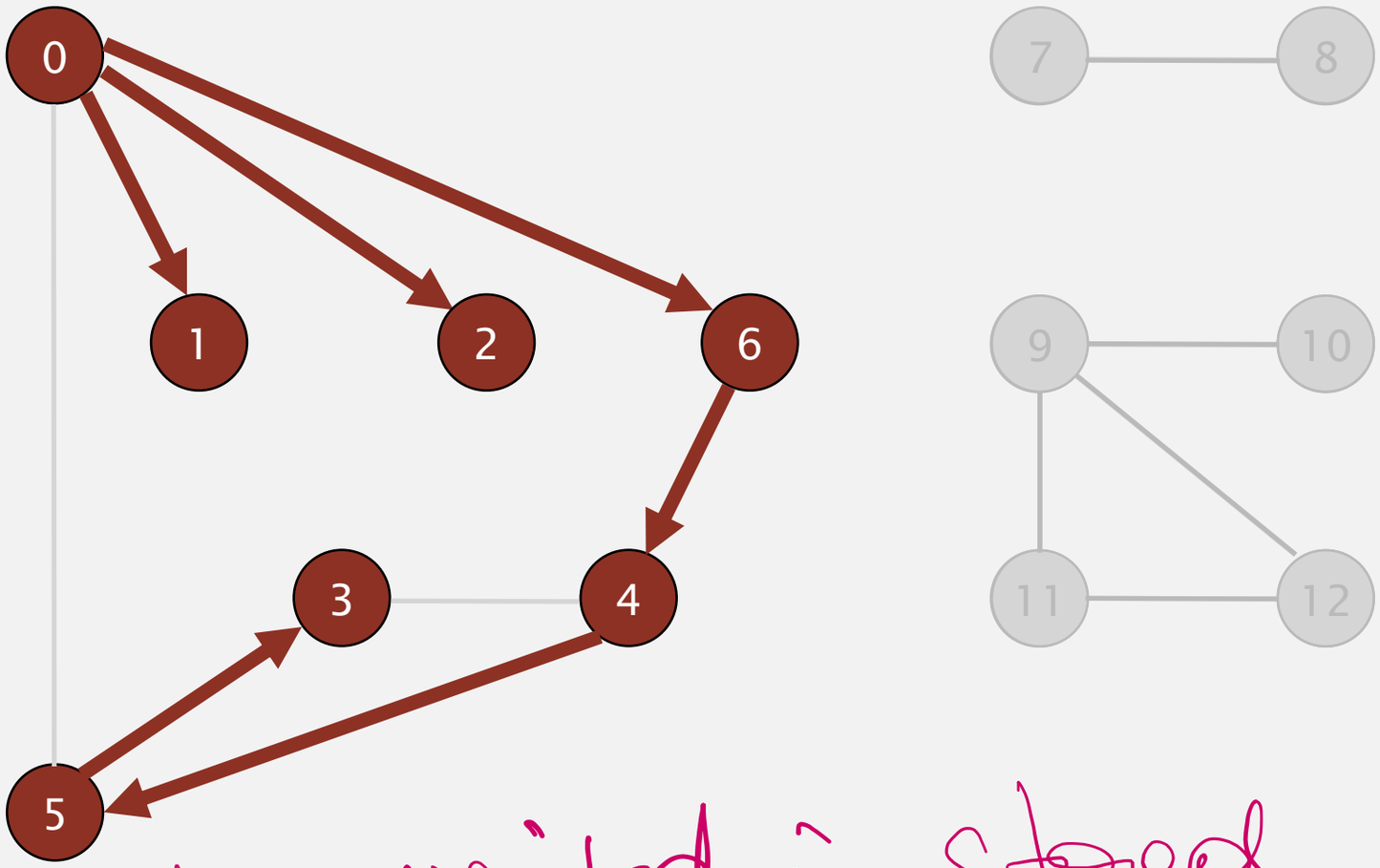
```
FOR every vertex v in G
    IF ( ! marked[v])
        DFS (G, v)
```

copyrighted content – Do not share

Depth-first search demo

To visit a vertex v :

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



edges visited in S-bred order

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	-

copyrighted content – Do not share

Design pattern for graph processing

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

copyrighted content – Do not share

Depth-first search: data structures

To visit a vertex v :

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

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
- Function-call stack for recursion.

copyrighted content – Do not share

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 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

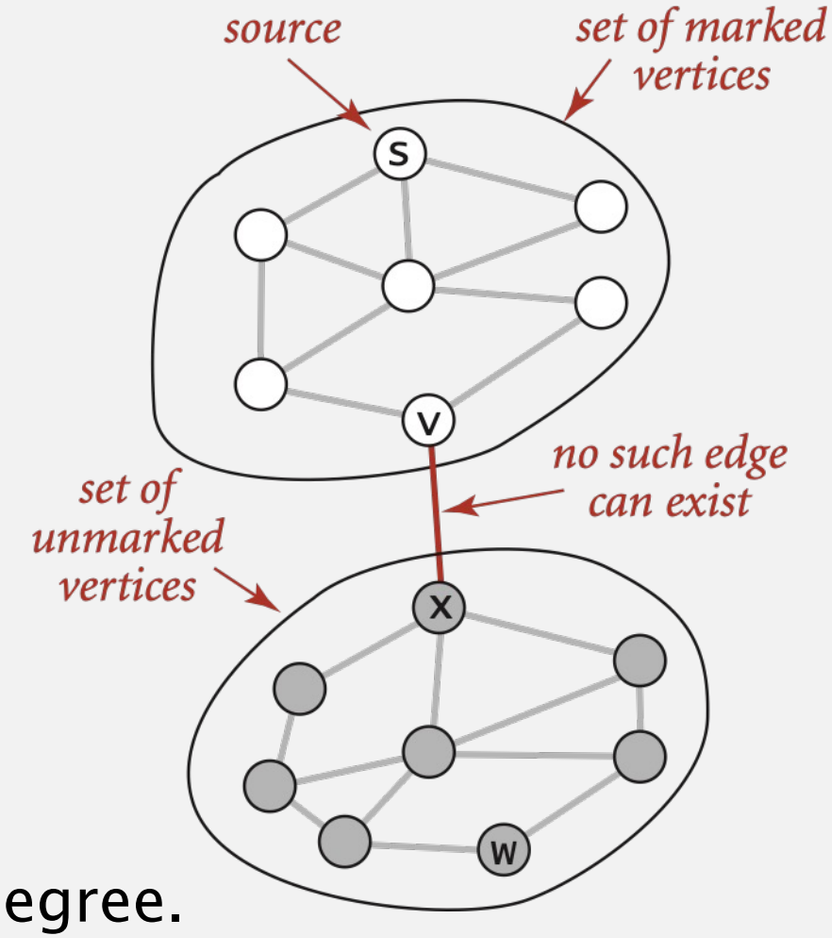
copyrig

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 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).



Pf. [running time]

- Each vertex connected to s is visited once.
- Visiting a vertex takes time proportional to its degree.

copyrighted content – Do not share

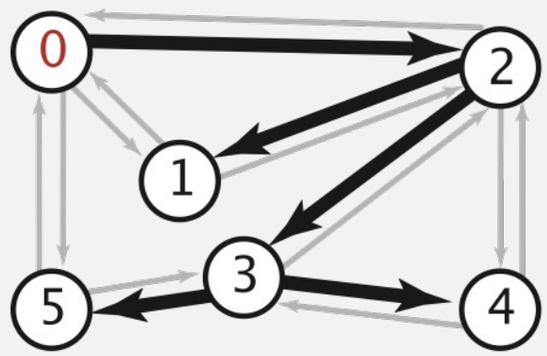
Depth-first search: properties

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

copyrighted content – Do not share

Flood Fill

Problem. Implement flood fill (Photoshop magic wand).





copyrighted content – Do not share

A5 is out – Graph assignment with very little starter code.

April 18th – in class photographer.

UNDIRECTED GRAPHS

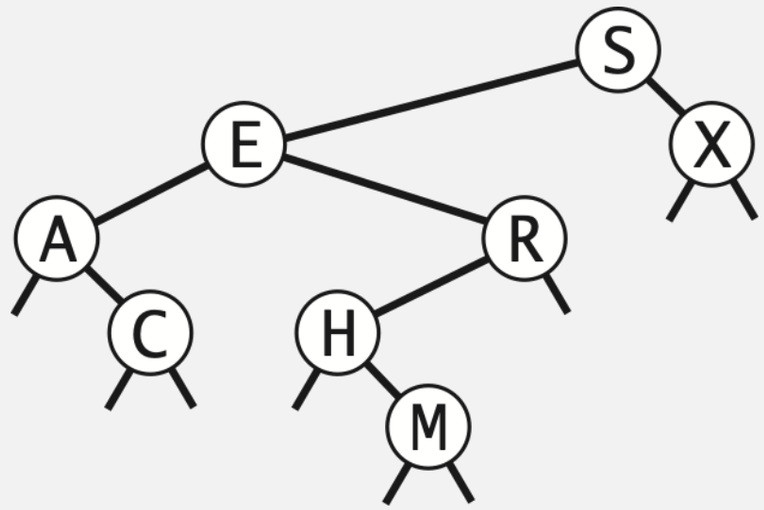
- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

copyrighted content – Do not share

Graph search

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).
- Postorder: vertices in order of returns from dfs(G, v).
- Level-order: vertices in increasing order of distance from s.

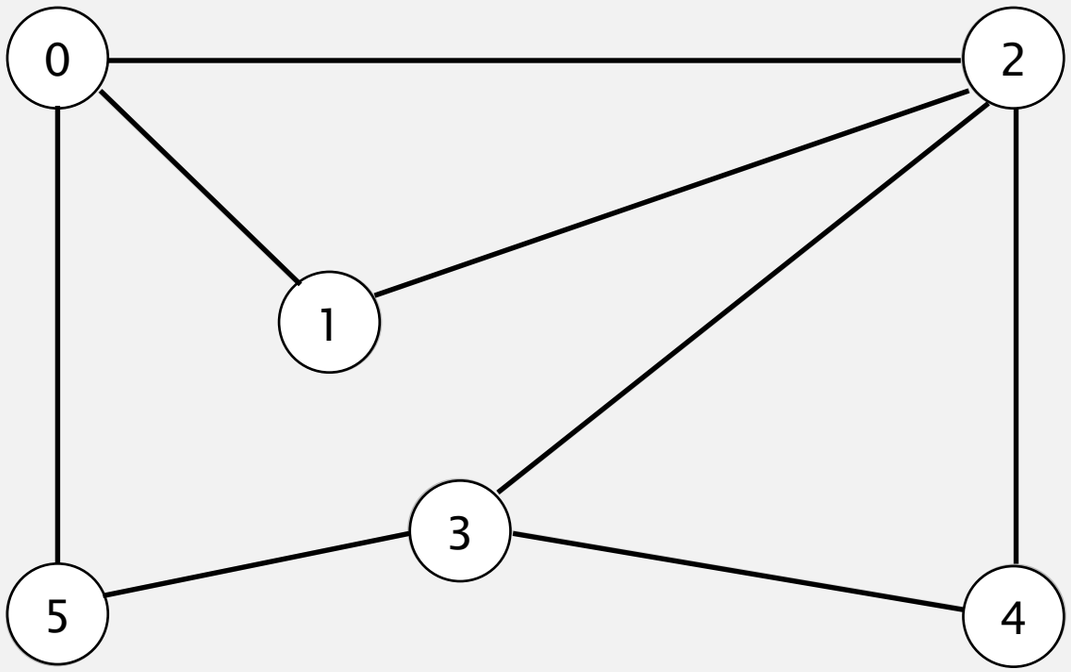
copyrighted content – Do not share

Breadth-first search

LO 8A.12

Repeat until queue is empty:

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



graph G

tinyCG.txt

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

copyrighted content – Do not share

ASending Order

Breadth-first search

Data structures.

- Boolean array marked[] to mark vertices.
- Integer array distTo[] to keep track of distance from source.

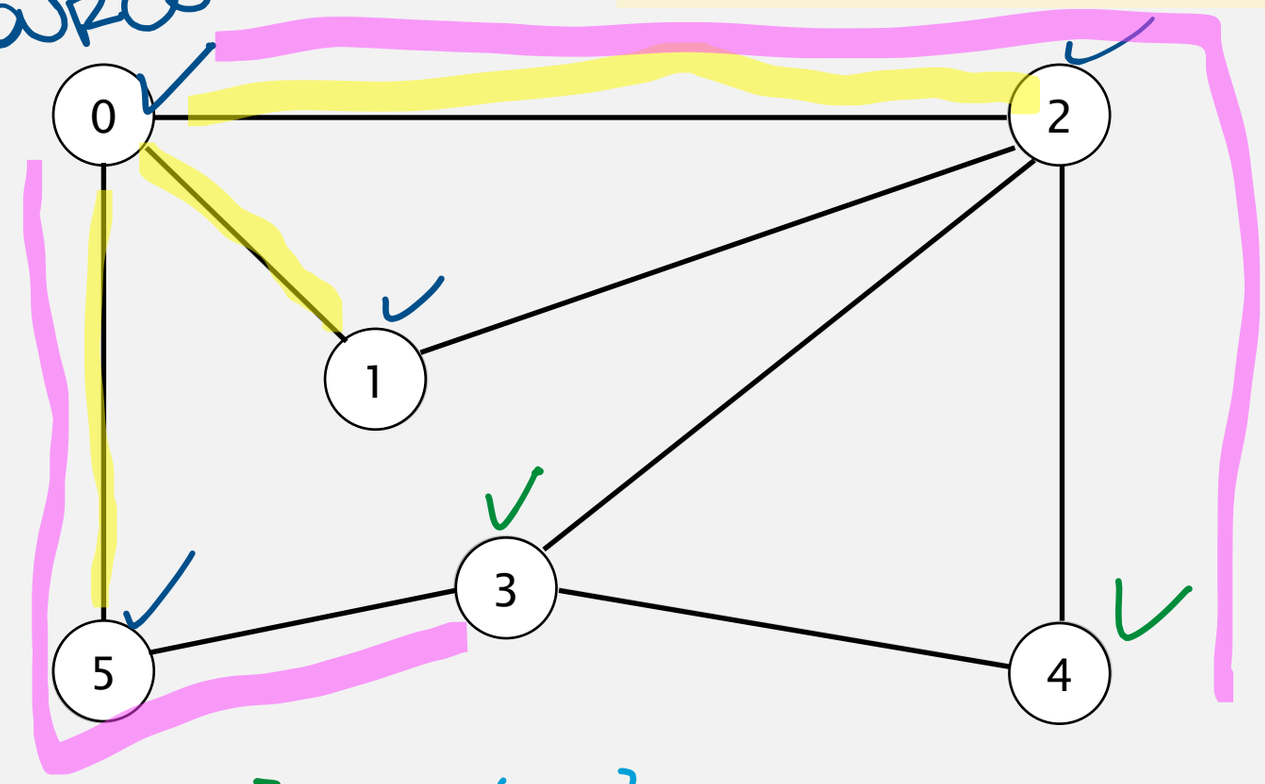
BFS (from source vertex s) BFS (∅)

1. **Put s onto a FIFO queue, and mark s as visited.**
2. **Repeat until the queue is empty:**
   - **v = dequeue vertex**
   - **add each of v's unmarked neighbors to the queue, and mark them.**

has the neighbor been visited ⇒ E times

marking a vertex visited ⇒ V times

source



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

Queue: 0, 1, 2, 5, 3, 4

Running time

V + E → # of times we check if a vertex has been visited

↳ # of times vertices are visited

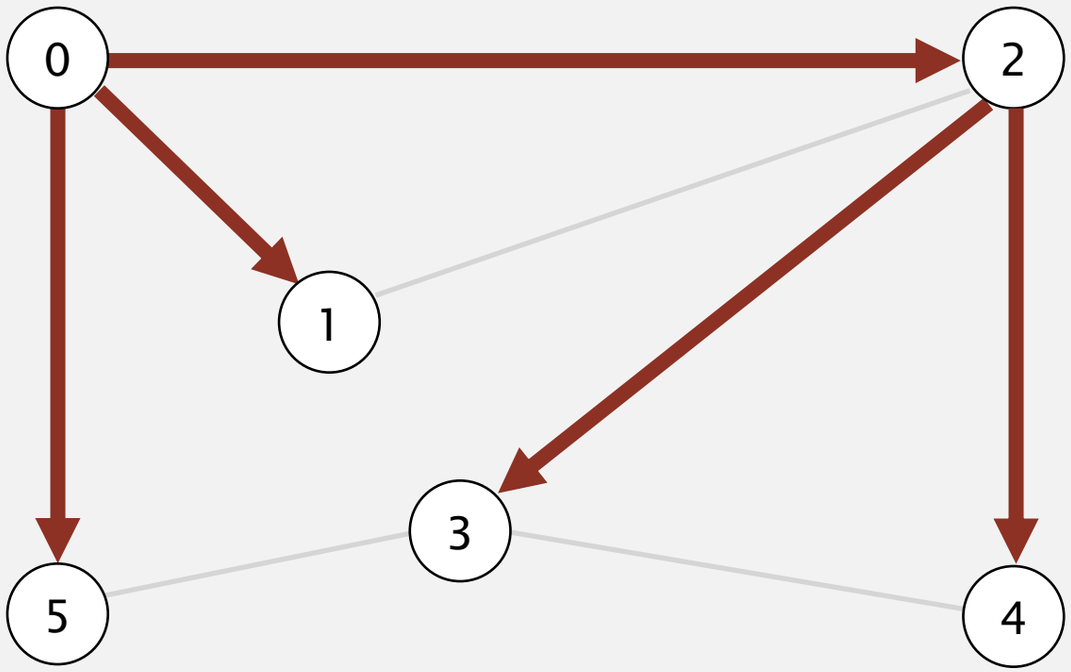
graph G

copyrighted content – Do not share

Breadth-first search demo

Repeat until queue is empty:

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



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

done

copyrighted content – Do not share

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.

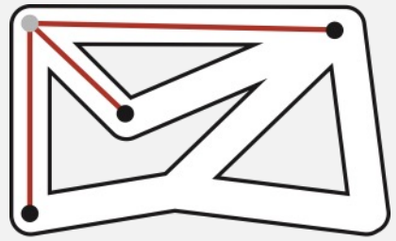
BFS (from source vertex s)

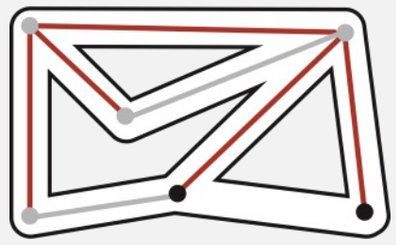
Put s onto a FIFO queue, and mark s as visited.

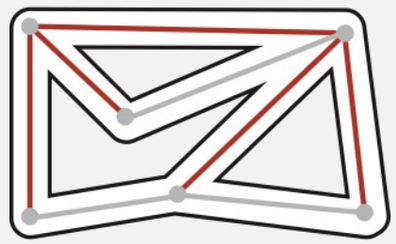
Repeat until the queue is empty:

- v = deque vertex

- add each of v's unmarked neighbors to the queue, and mark them.







copyrighted content – Do not share

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

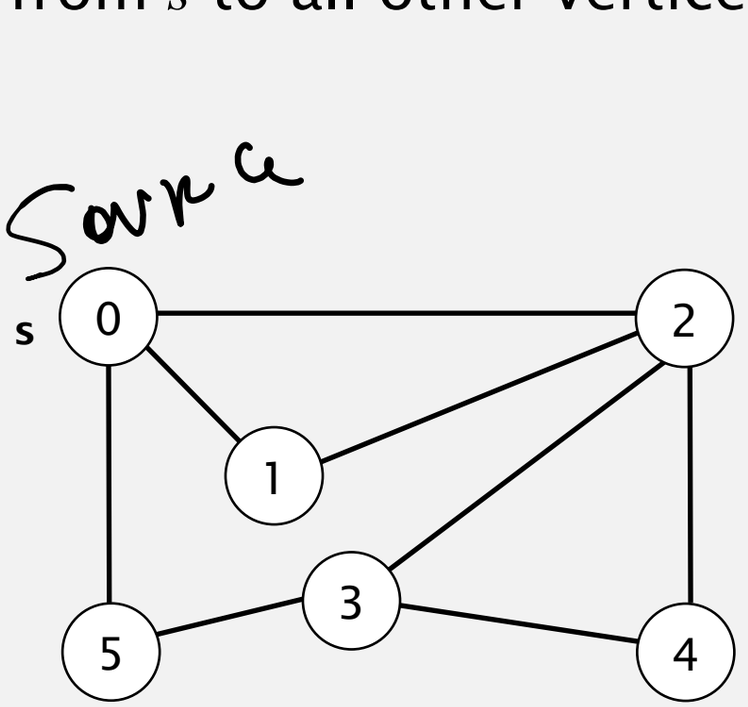
Breadth-first search properties

Q. In which order does BFS examine vertices?

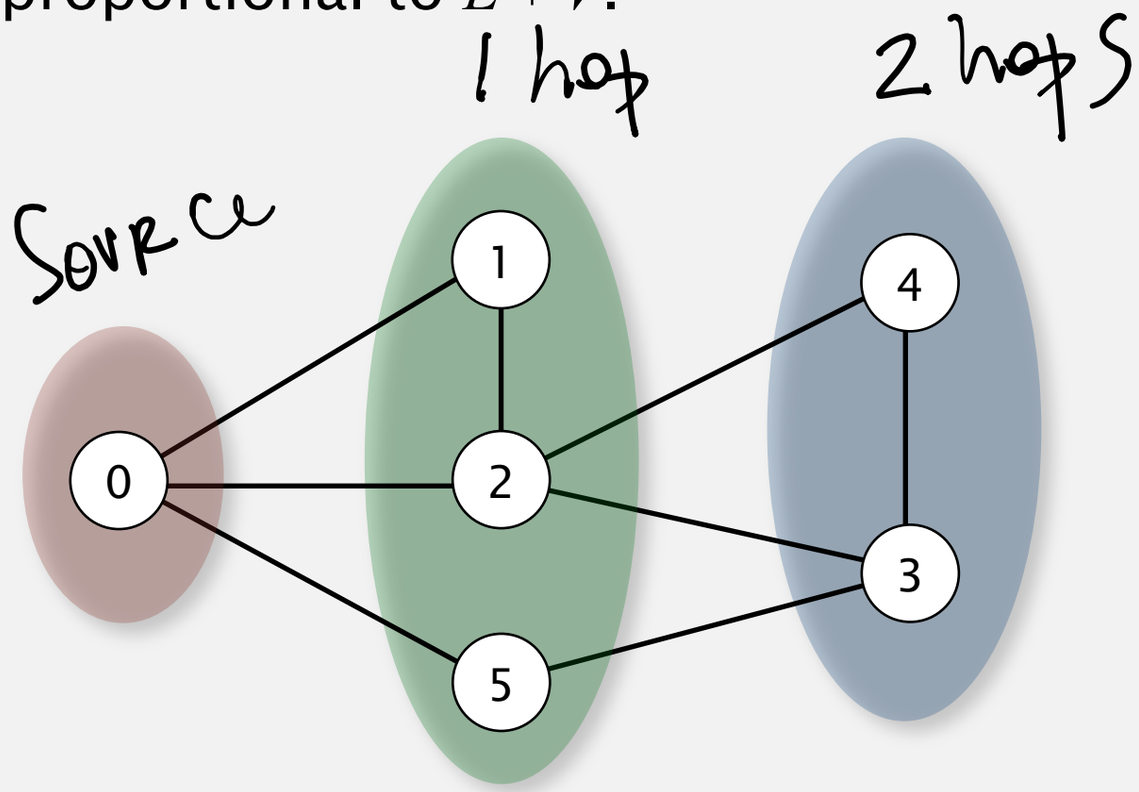
A. Increasing distance (number of edges) from s.

queue always consists of ≥ 0 vertices of distance k from s,
followed by ≥ 0 vertices of distance k+1

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



graph G

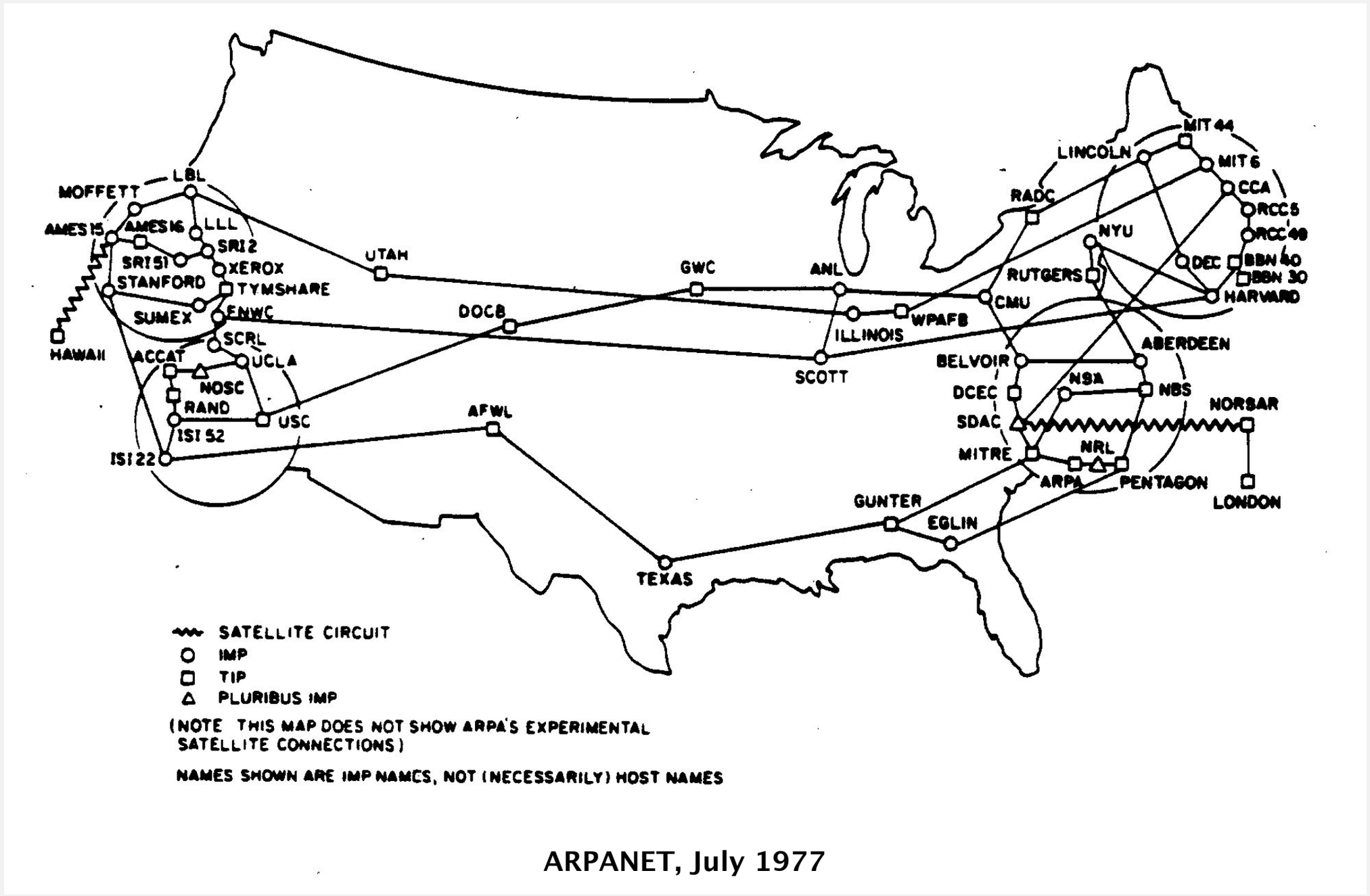


dist = 0 dist = 1 dist = 2

copyrighted content – Do not share

Breadth-first search application: routing

Fewest number of hops in a communication network.

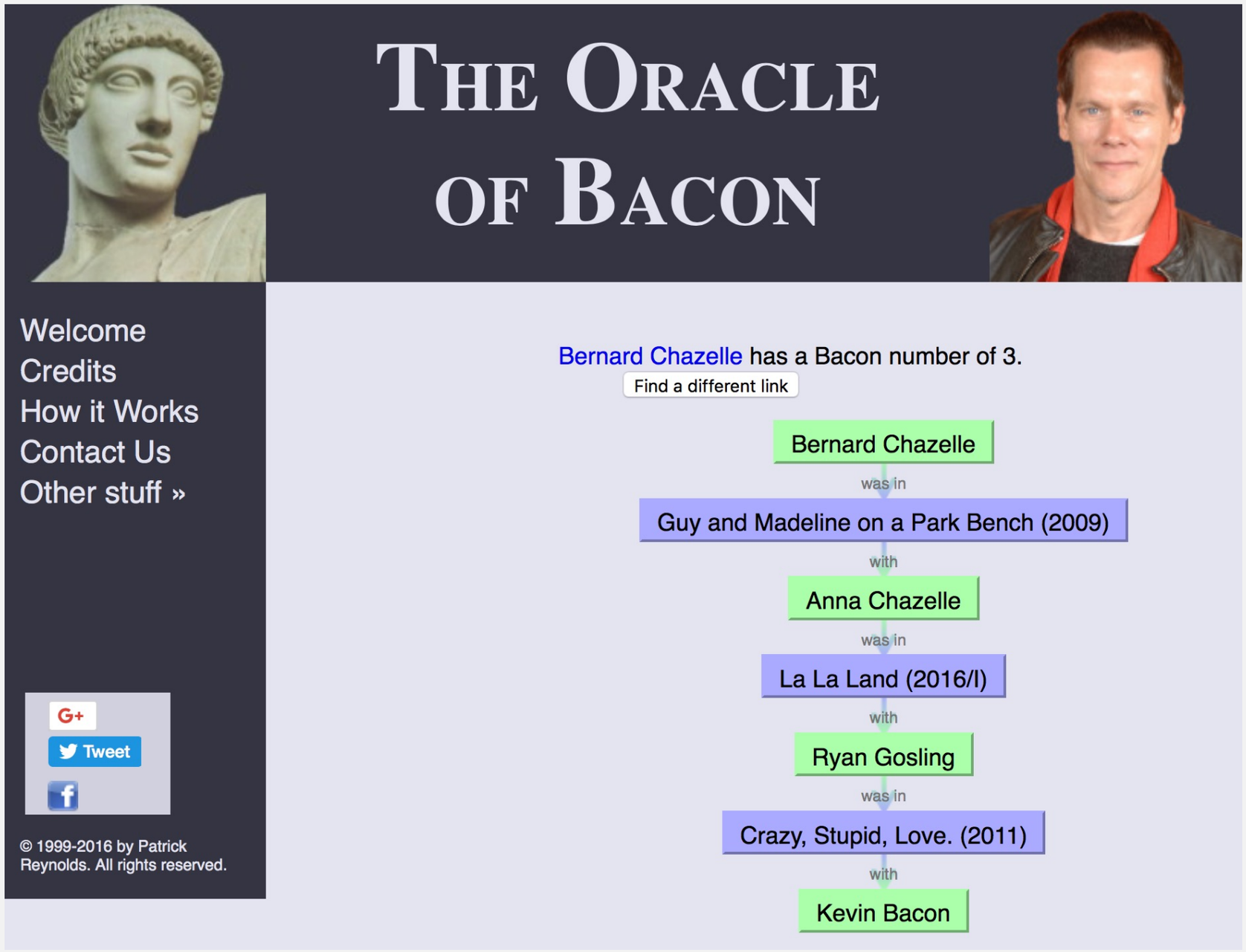


ARPANET, July 1977

copyrighted content – Do not share

Breadth-first search application: Kevin Bacon numbers

LO 8A.14



http://oracleofbacon.org



Endless Games board game

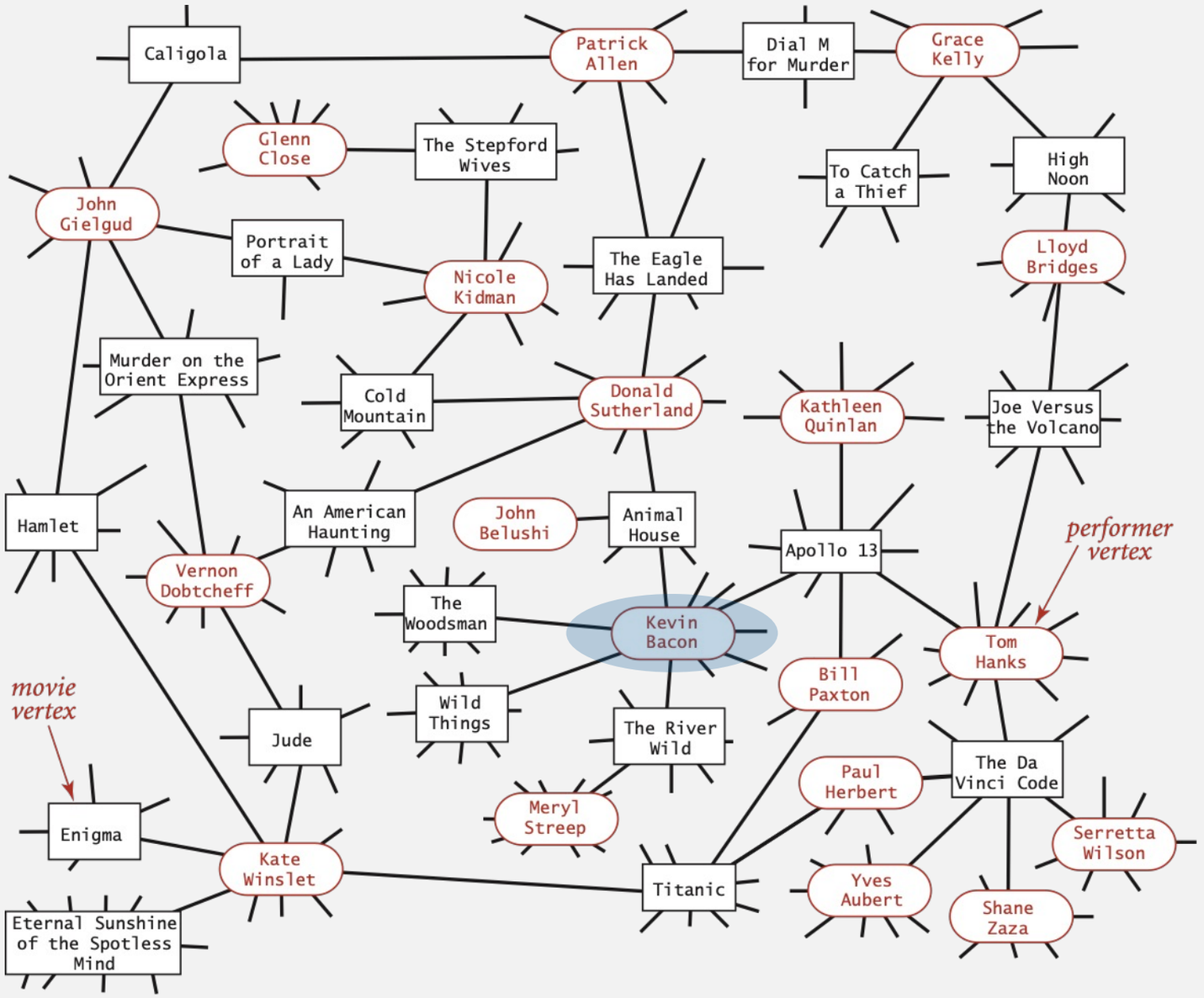


SixDegrees iPhone App

copyrighted content – Do not share

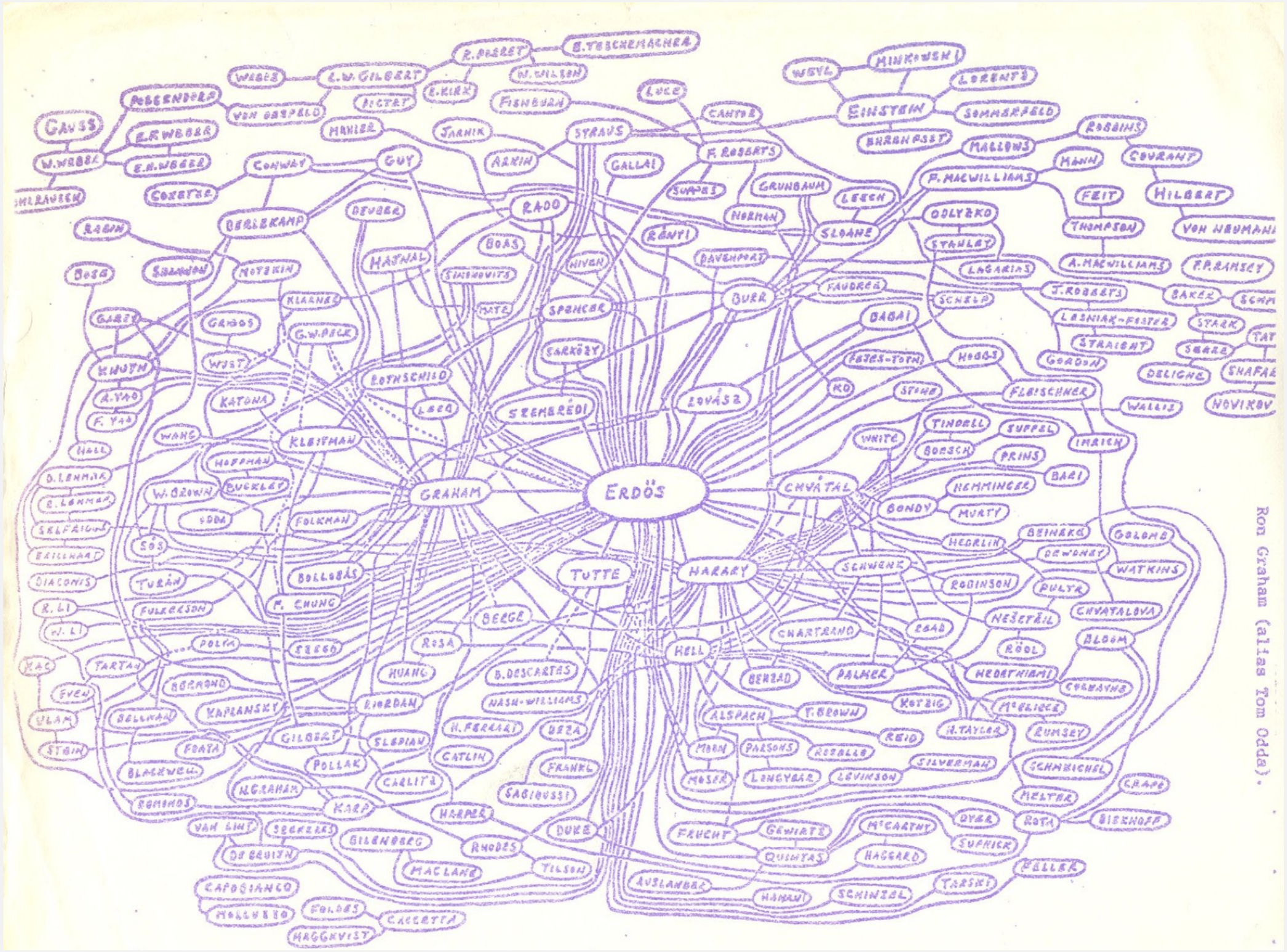
Kevin Bacon graph

- Include one vertex for each performer and one for each movie.
- Connect a movie to all performers that appear in that movie.
- Compute shortest path from s = Kevin Bacon.



copyrighted content – Do not share

Breadth-first search application: Erdös numbers



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

copyrighted content - Do not share

Undirected Graphs

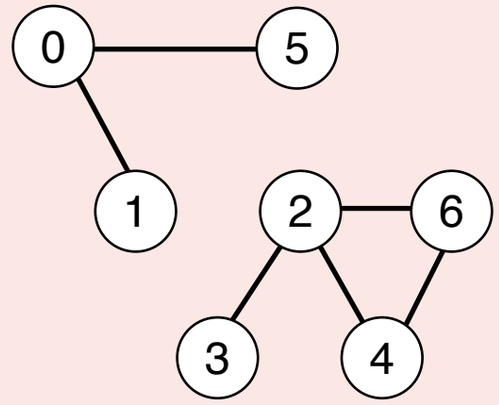
- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

http://ds.cs.rutgers.edu

copyrighted content - Do not share

Graph-processing challenge 1

Problem. Identify connected components.



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

How difficult?

A. Any programmer could do it.

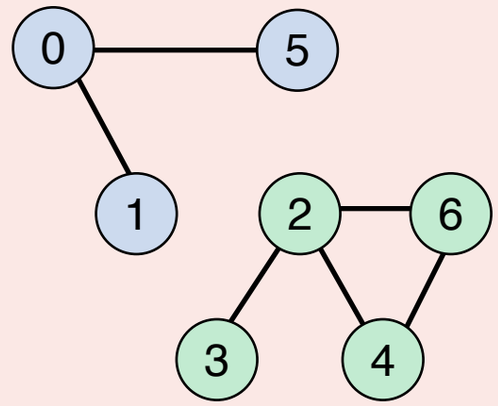
B. Diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

(see textbook)

E. No one knows.



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

copyrighted content – Do not share

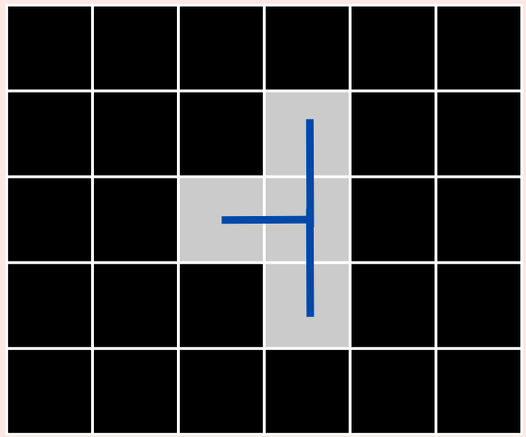
Graph-processing challenge 1

Problem. Identify connected components.

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.

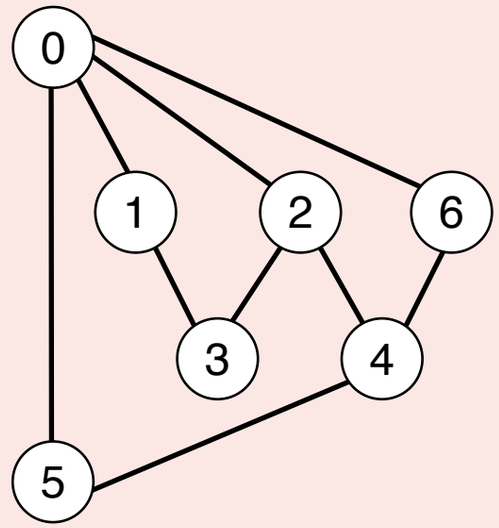




copyrighted content – Do not share

Graph-processing challenge 2

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?

A. Any programmer could do it.

B. Diligent algorithms student could do it.

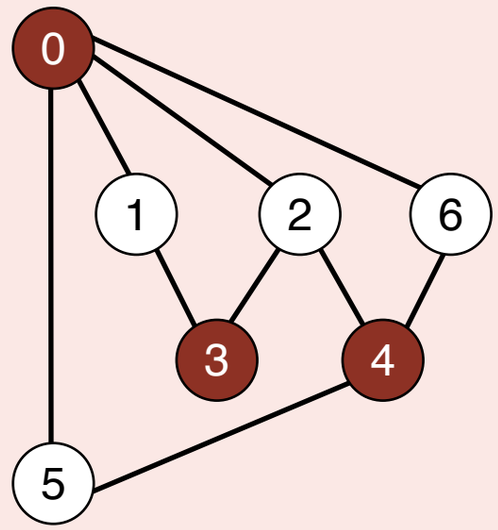
C. Hire an expert.

D. Intractable.

E. No one knows.

simple DFS-based solution

(see textbook)



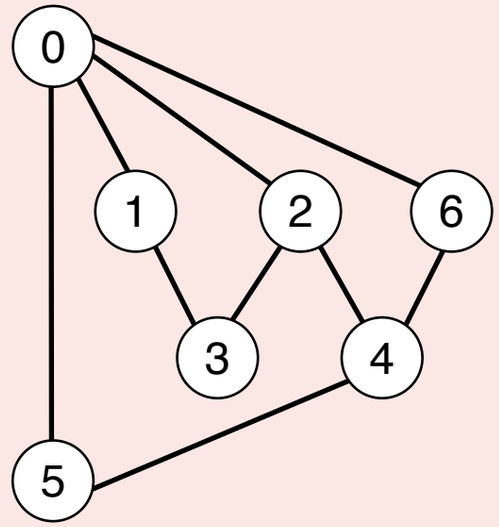
0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6

{ 0, 3, 4 }

copyrighted content – Do not share

Graph-processing challenge 3

Problem. Find a cycle in a graph (if one exists).



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

0-5-4-6-0

How difficult?

A. Any programmer could do it.

B. Diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows.

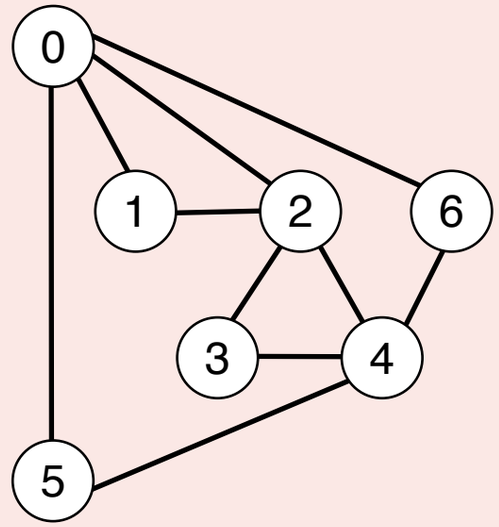
simple DFS-based solution

(see textbook)

copyrighted content – Do not share

Graph-processing challenge 4

Problem. Is there a (general) cycle that uses every 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?

A. Any programmer could do it.

B. Diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

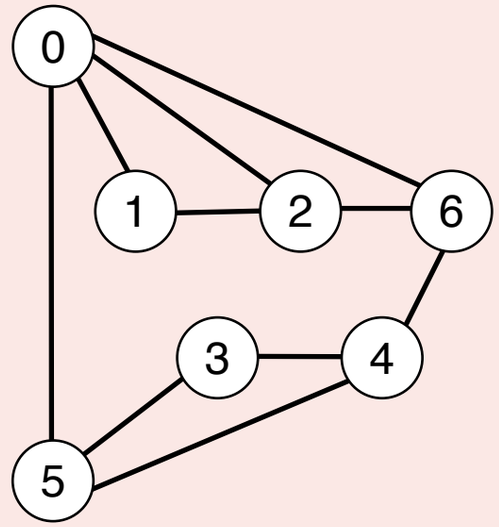
E. No one knows.

yes if and only if graph is connected
and every vertex has even degree
(Leonhard Euler 1786)

copyrighted content – Do not share

Graph-processing challenge 5

Problem. Is there a cycle that contains every 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?

A. Any programmer could do it.

B. Diligent algorithms student could do it.

C. Hire an expert.

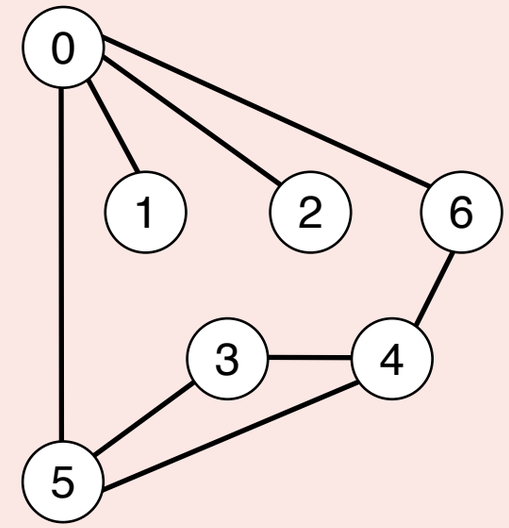
D. Intractable.

E. No one knows.

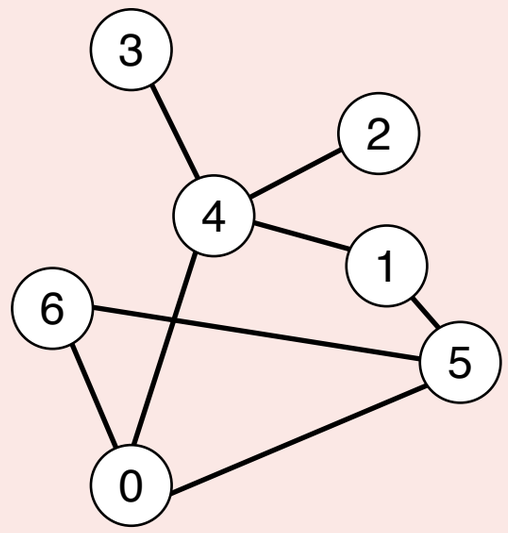
copyrighted content – Do not share

Graph-processing challenge 6

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



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

How difficult?

A. Any programmer could do it.

B. Diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

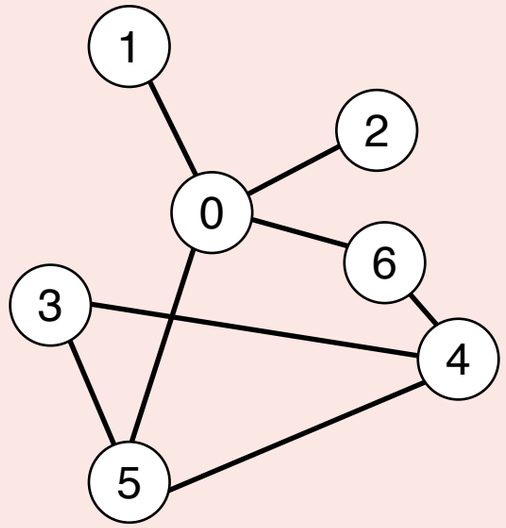
E. No one knows.

copyrighted content – Do not share

Graph-processing challenge 7

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

try it yourself at http://planarity.net



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

How difficult?

A. Any programmer could do it.

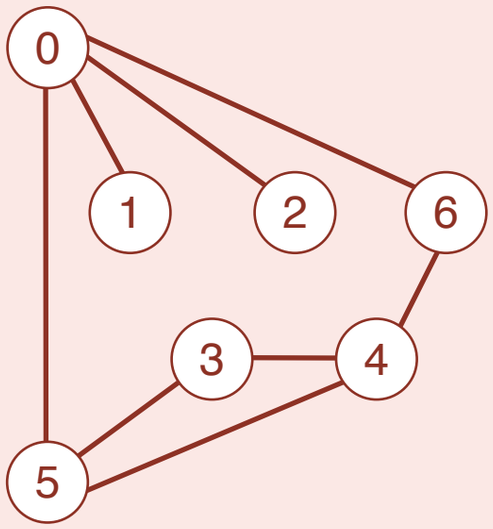
B. Diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows.

linear-time DFS-based planarity algorithm
discovered by Tarjan in 1970s
(too complicated for most practitioners)



copyrighted content – Do not share

Graph traversal summary

BFS and DFS enables efficient solution of many (but not all) graph problems.

graph problem	BFS	DFS	time
s–t path	✓	✓	$E + V$
shortest s–t path	✓		$E + V$
cycle	✓	✓	$E + V$
Euler cycle		✓	$E + V$
Hamilton cycle			$2^{1.657\,V}$
bipartiteness (odd cycle)	✓	✓	$E + V$
connected components	✓	✓	$E + V$
biconnected components		✓	$E + V$
planarity		✓	$E + V$
graph isomorphism			$2^{c \ln^3 V}$

copyrighted content – Do not share

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*



copyrighted content - Do not share

http://ds.cs.rutgers.edu

Some slides Adopted and modified from Sedgewick and Wayne

last updated: 2/11/08 8:38 AM