Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

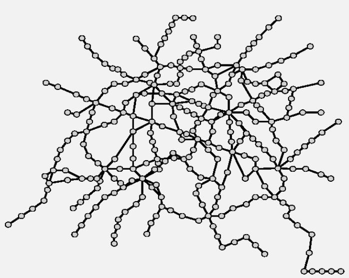
- *introduction*
- *graph API*
- *graph search*
- *depth-first search*
- *breadth-first search*
- *challenges*

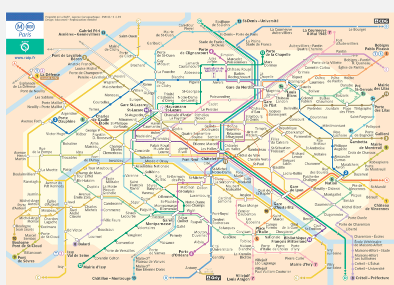
Undirected graphs

Graph. Set of vertices connected pairwise by edges.

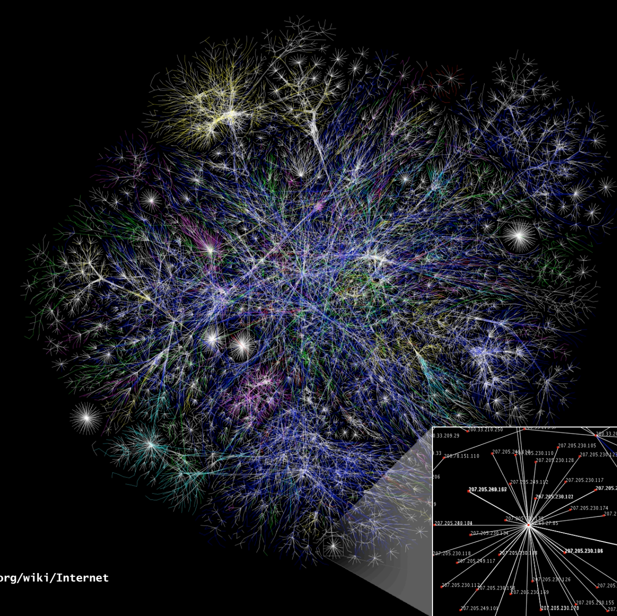
Why study graph algorithms?

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



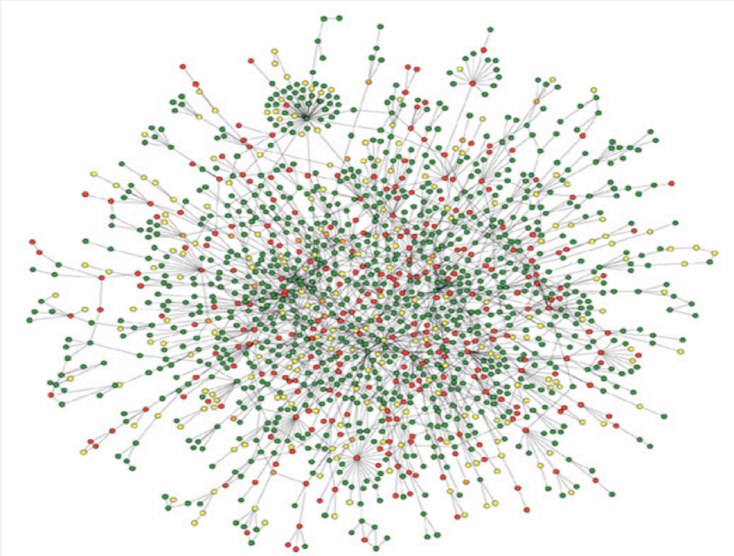


The Internet as mapped by the Opte Project



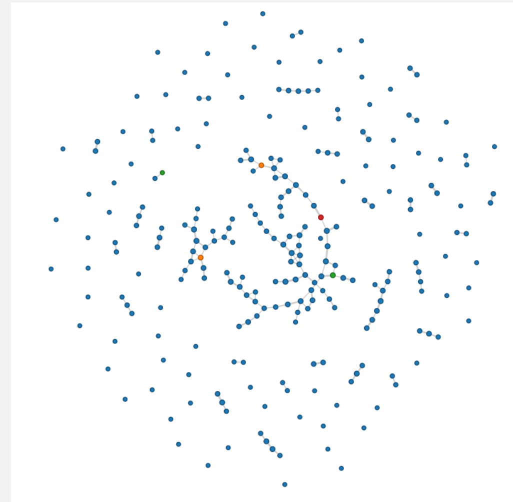
http://en.wikipedia.org/wiki/Internet

Protein-protein interaction network

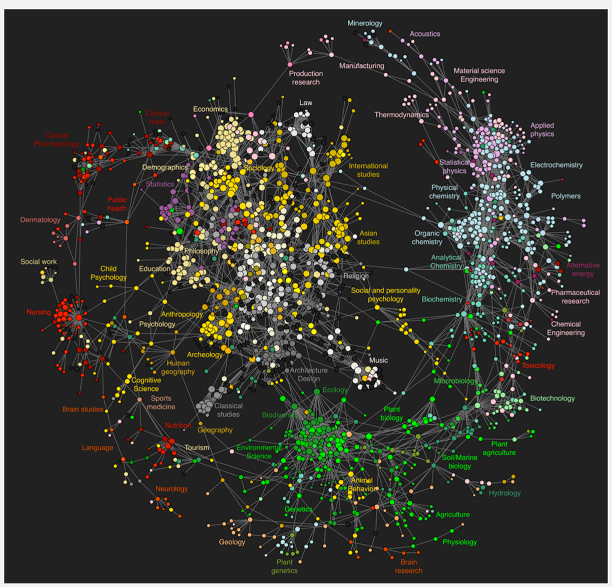


Reference: Jeong et al, Nature Review | Genetics

Partners for COS 226 Spring 2013

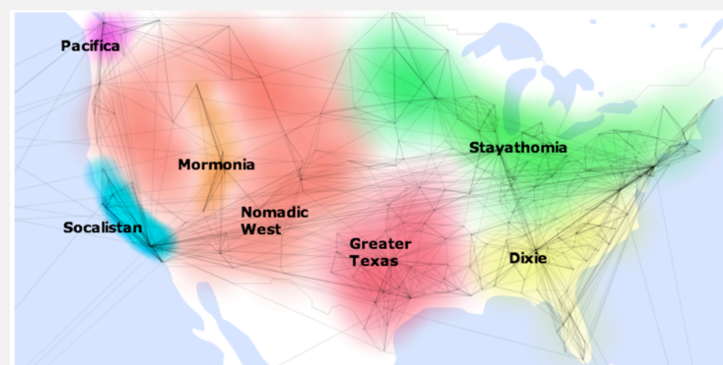


Map of science clickstreams



http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803

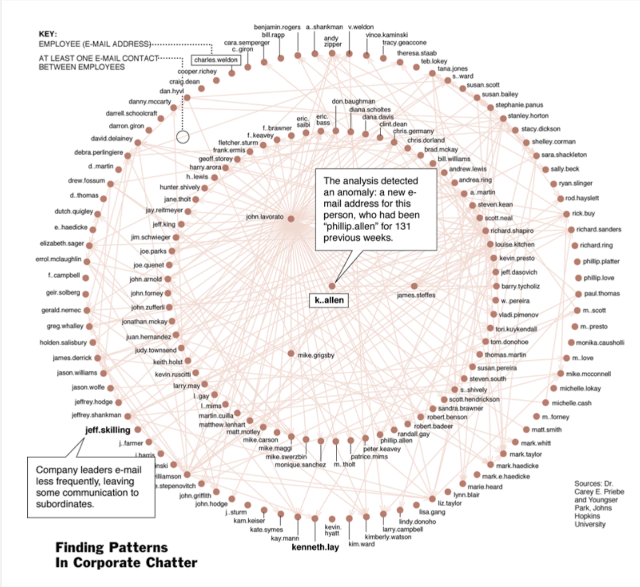
America according to the Facebook graph



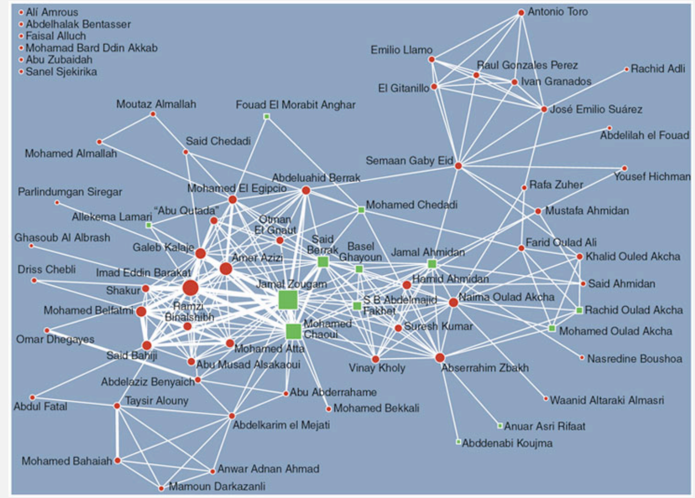
"How to split up the US" by Pete Warden

http://petewarden.com/2010/02/06/how-to-split-up-the-us/

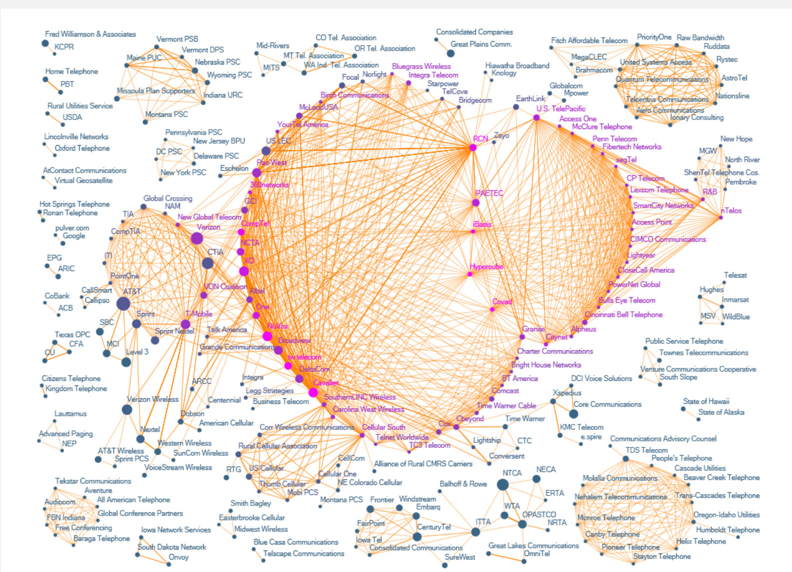
One week of Enron emails



Terrorist network



The evolution of FCC lobbying coalitions



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

Framingham heart study

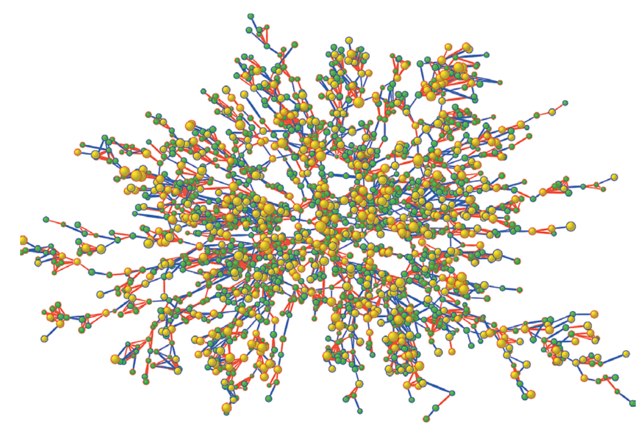
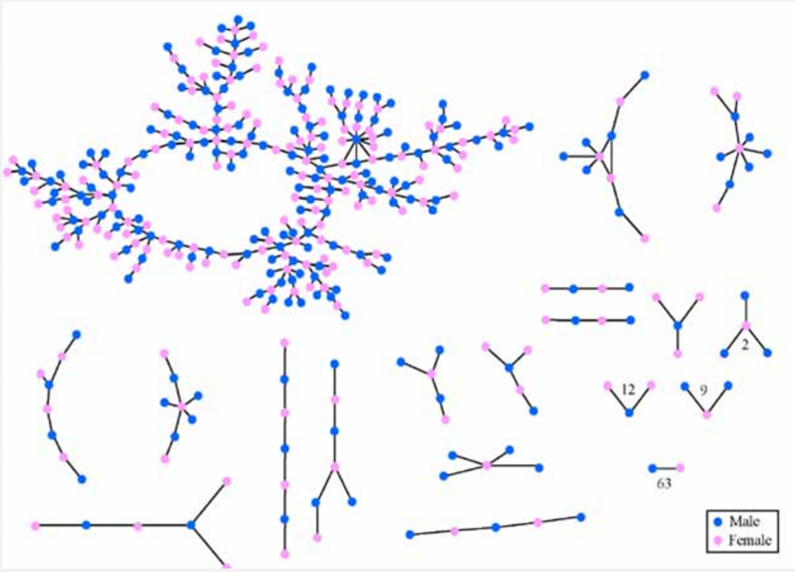


Figure 1. Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000.
Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ≥30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.

"The Spread of Obesity in a Large Social Network over 32 Years" by Christakis and Fowler in New England Journal of Medicine, 2007

Sexual/romantic network of a high school



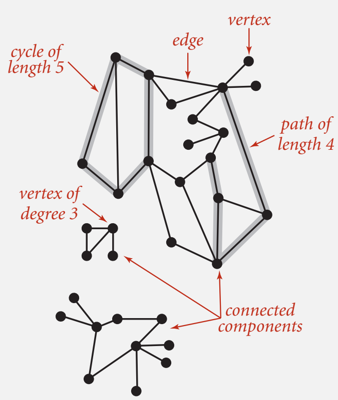
Graph applications

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

Graph terminology

Path. Sequence of vertices connected by edges.
Cycle. Path whose first and last vertices are the same.

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



Some graph-processing problems

Path. Is there a path between s and t?
Shortest path. What is the shortest path between s and t?

Cycle. Is there a cycle in the graph?
Euler tour. Is there a cycle that uses each edge exactly once?
Hamilton tour. Is there a cycle that uses each vertex exactly once?

Connectivity. Is there a way to connect all of the vertices?
MST. What is the best way to connect all of the vertices?
Biconnectivity. Is there a vertex whose removal disconnects the graph?

Planarity. Can you draw the graph in the plane with no crossing edges
Graph isomorphism. Do two adjacency lists represent the same graph?

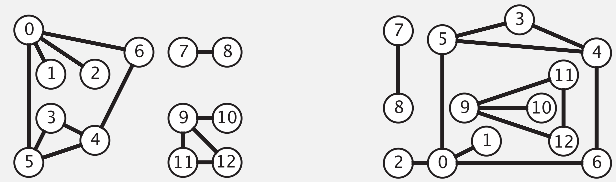
Challenge. Which of these problems are easy? difficult? intractable?

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

Graph representation

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



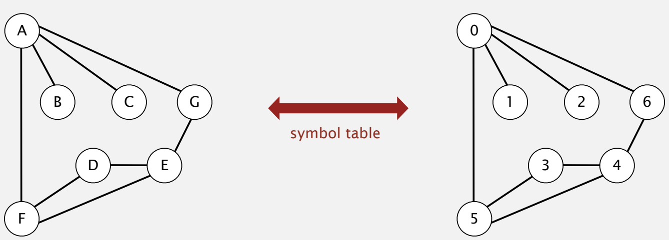
two drawings of the same graph

Caveat. Intuition can be misleading.

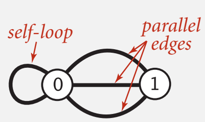
Graph representation

Vertex representation.

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



Anomalies.



Graph API

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

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

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

read graph from input stream

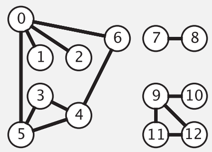
print out each edge (twice)

Graph API: sample client

Graph input format.

tinyG.txt

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




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

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

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

read graph from input stream

print out each edge (twice)

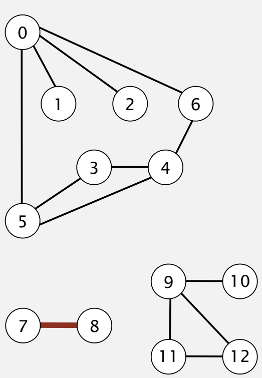
Typical graph-processing code

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;
}
```


Set-of-edges graph representation

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



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

```
// 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;
}
```

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

Given exactly this data structure, what is the best possible run time we can achieve for degree?

A. Θ(E) [90833]

B. Θ(V) [90846]

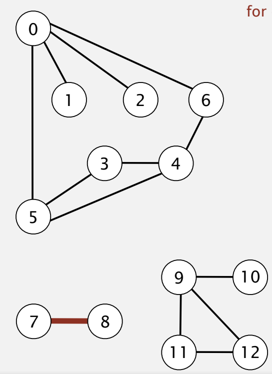
C. Θ(degree(v)) [90901]

Adjacency-matrix graph representation

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

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

two entries for each edge



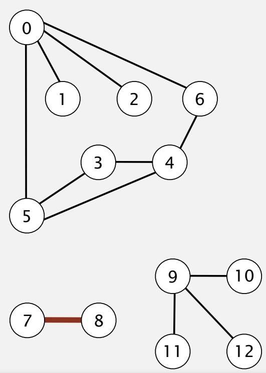
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0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

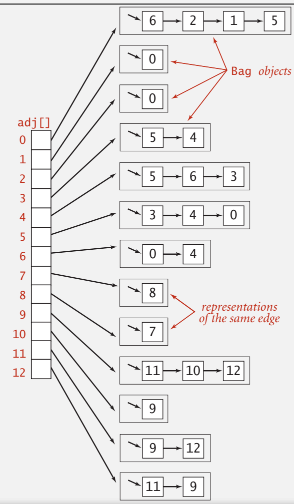
Same question. pollEv.com/jhug text to **37607**

Θ(E) - [93859] Θ(V) - [93893] Θ(degree(v)) - [93898]

Adjacency-list graph representation

Maintain vertex-indexed array of lists.





Same question. pollEv.com/jhug text to 37607

Θ(E) - [12833] Θ(V) - [96709] Θ(degree(v)) - [96925]

Graph representations

In practice. Use adjacency-lists representation.

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

huge number of vertices, small average vertex degree

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

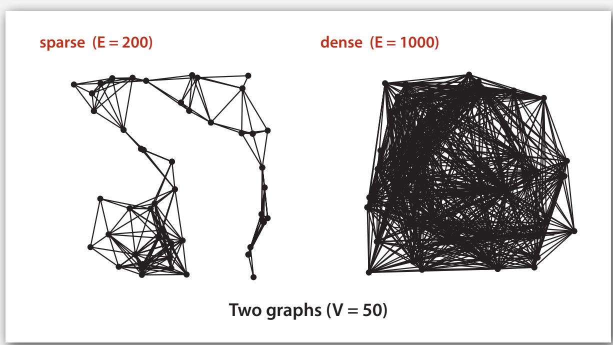
* disallows parallel edges

Graph 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



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)
   {
      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

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

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

Maze exploration

Graph search

- Traverse entire graph from starting region.
  - For some objectives, quit early when objective is achieved.
- Never go any place more than once.

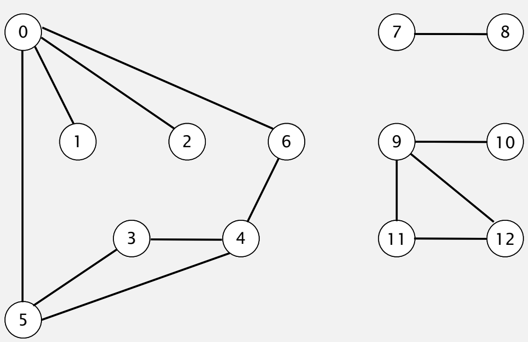
Examples of problems solvable using graph search.

- Finding all vertices reachable from A.
  - What areas are in danger of fire?
- Testing connectivity of A and B.
  - Could a fire raging in *my hair* reach *your computer*?
- Finding the shortest path from A to B.
  - Kevin Bacon number.
- Finding the connected components in a graph.
  - Reverse engineering of biological systems.

Basic graph search demo

Algorithm

- Two regions: Explored (marked in red) and unexplored.
- Given explored region:
  - Select any unexplored vertex adjacent to the explored region.
  - Mark that vertex as explored.
- Repeat until no more vertices can be selected.



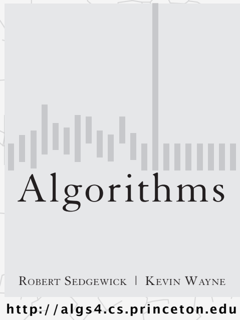
Basic graph search

Graph search for problem solving

- So far:
  - Connectivity to a particular region (using marked array).
  - Finding paths from a particular region (using edgeTo array).
- Coming up:
  - Shortest paths.
  - Connected components.
  - And more!

Algorithmic specifics

- Vertex selection strategy.
  - Must select some order in which to add vertices.
- Data structure selection.
  - Based on vertex selection strategy.
  - Based on problem we'd like to solve.

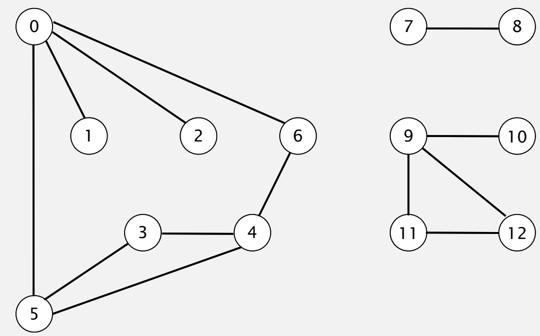


4.1 Undirected Graphs

Depth-first search

Selection strategy

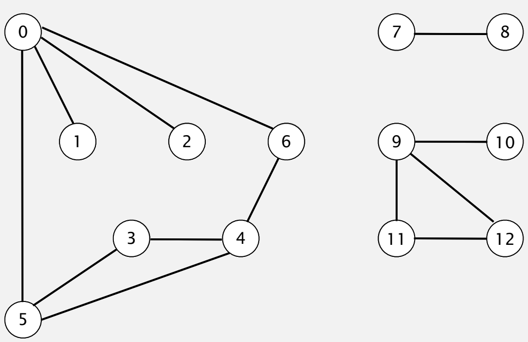
- Visiting a vertex consists of:
  - Marking that vertex as visited.
  - Visiting all of its unvisited neighbors. ← recursive!



Depth-first search demo

To visit a vertex v :

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



graph G

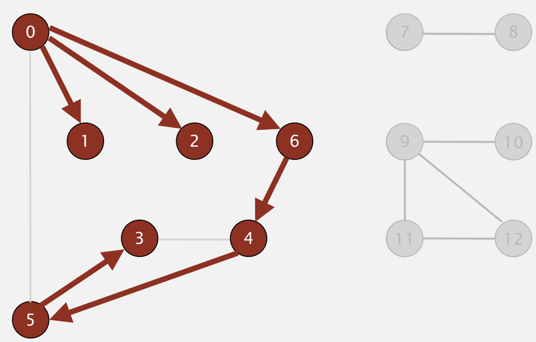
tinyG.txt

```
13      ← V
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
```


Depth-first search demo

To visit a vertex v :

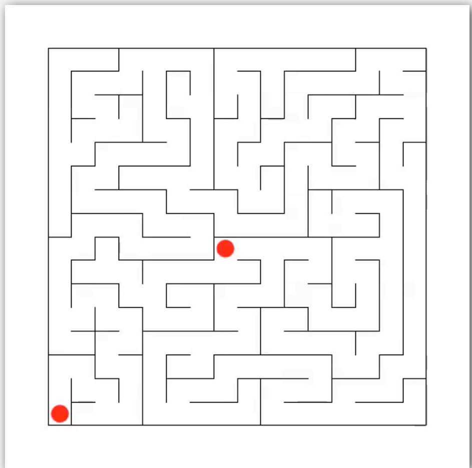
- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



vertices reachable from 0

v	marked[]	edgeTo[v]
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	–

Maze exploration



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

Depth-first search

Goal. Find all vertices connected to *s* (and a corresponding path).

Idea. Fully explore one branch before going to another.

Algorithm.

- Use recursion to track where you've been.
  - Hit a dead end? Go back to the last time you made a choice.
- Mark each visited vertex (and maybe keep track of edge taken to visit it).

Data structures.

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

Depth-first search

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

   public DepthFirstSearch(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])
         {
            dfs(G, w);
            edgeTo[w] = 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

starting region consists of 1 vertex. easy to generalize!

recursive DFS does the work

Depth-first search properties

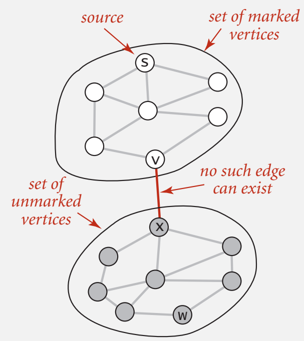
Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees.

Pf. [correctness]

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

Pf. [running time]

Each vertex connected to s is visited once.

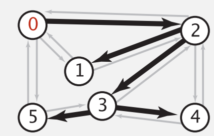


Depth-first search properties

Proposition. After DFS, can find vertices connected to s in constant time and can find a path to s (if one exists) in time proportional to its length.

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

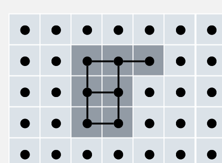
Depth-first search application: flood fill

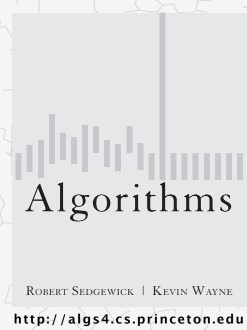
Challenge. Flood fill (Photoshop magic wand).

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.



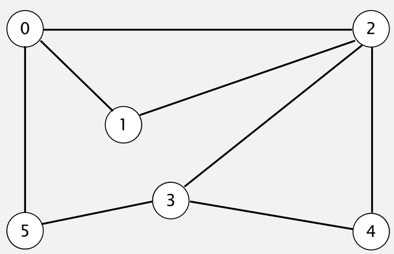


4.1 Undirected Graphs

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.



tinyCG.txt

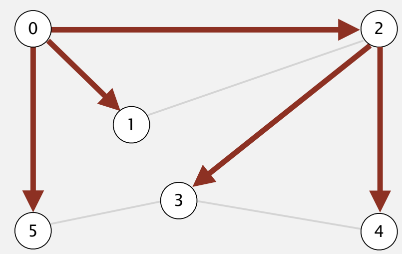
```
V → 6
    8 ← E
    0 5
    2 4
    2 3
    1 2
    0 1
    3 4
    3 5
    0 2
```

graph G

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

Breadth-first search

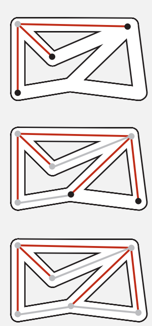
Depth-first search. Put unvisited vertices on a stack. ← Q: bu.. bu.. we did recursion? A: That's just a stack!

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

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

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
- **remove the least recently added vertex v**
- **add each of v's unvisited neighbors to the queue, and mark them as visited.**



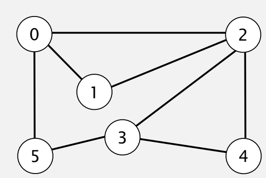
Intuition. BFS examines vertices in increasing distance from s.

Breadth-first search properties

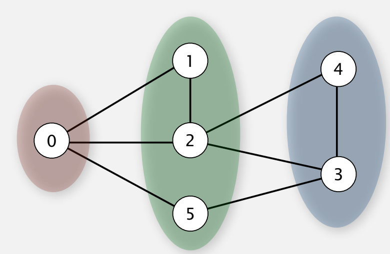
Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a graph in time proportional to $E + V$.

Pf. [correctness] Queue always consists of zero or more vertices of distance k from s, followed by zero or more vertices of distance $k + 1$.

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



graph



dist = 0 dist = 1 dist = 2

Breadth-first search

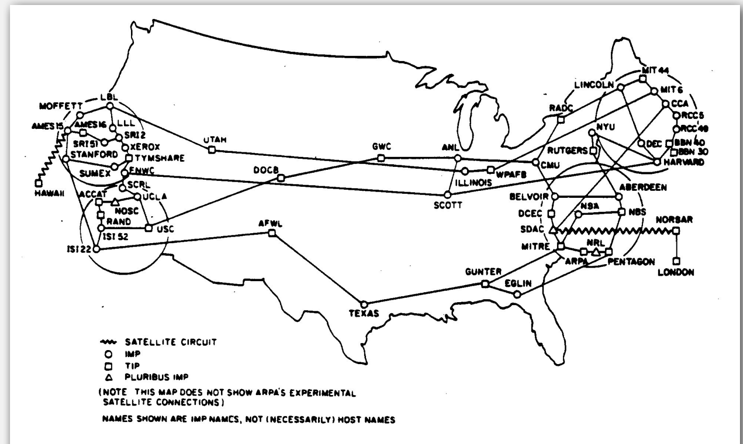
```
public class BreadthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private int[] distTo;

   …

   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 application: routing

Fewest number of hops in a communication network.



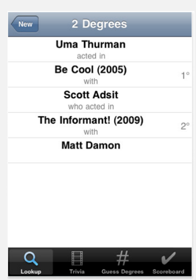
ARPANET, July 1977

Breadth-first search application: Kevin Bacon numbers

Kevin Bacon numbers.

http://oracleofbacon.org

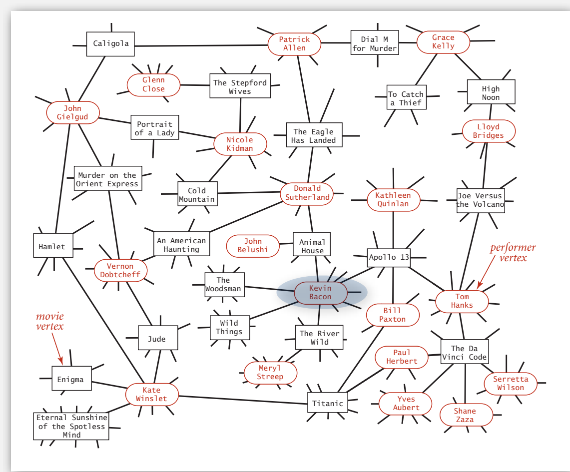
Endless Games board game



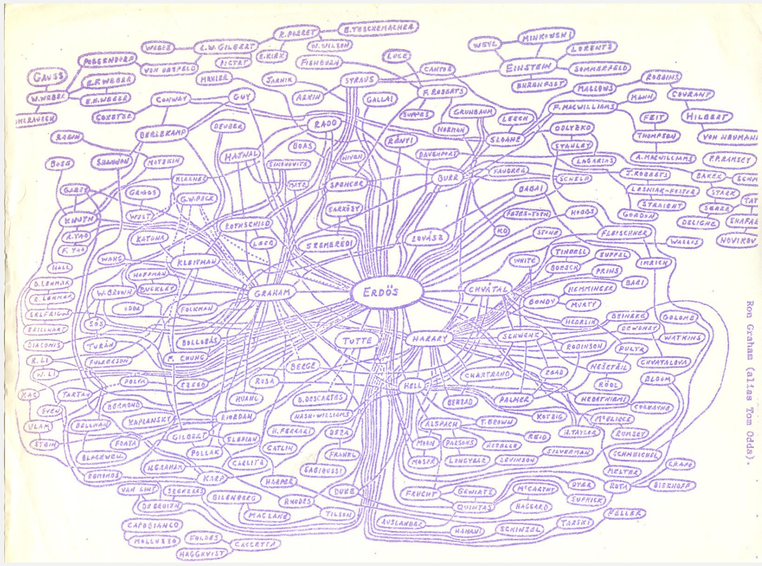
SixDegrees iPhone App

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.



Breadth-first search application: Erdös numbers



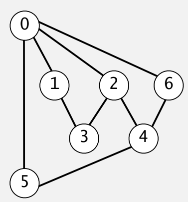
hand-drawing of part of the Erdös graph by Ron Graham

http://algs4.cs.princeton.edu

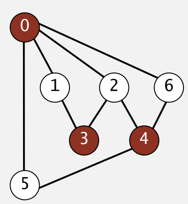
4.1 Undirected Graphs

Graph-processing challenge 1

Problem. Is a graph bipartite?



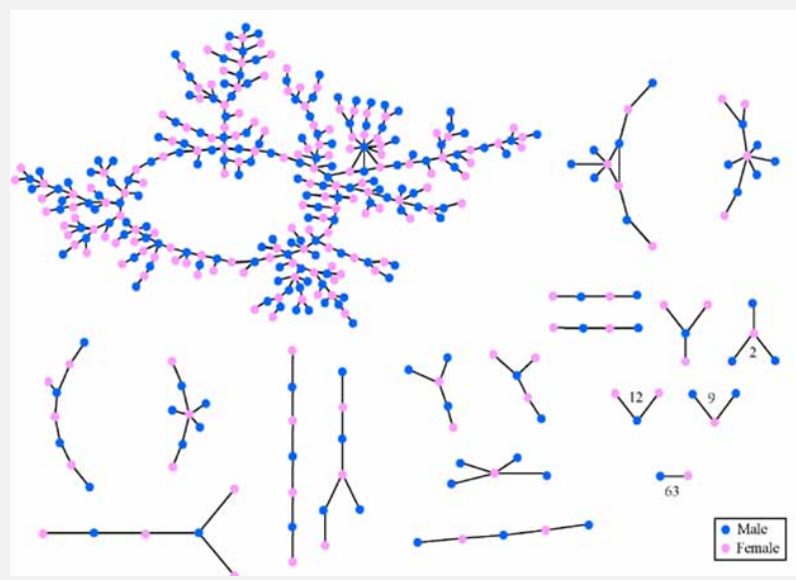
0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6



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

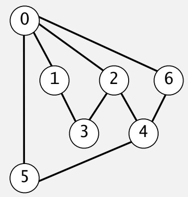
{ 0, 3, 4 }

Intuition: think about it as a dating graph



Graph-processing challenge 1

Problem. Is a graph bipartite?

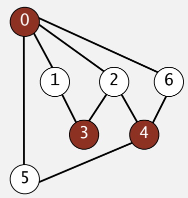


0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6

How difficult?

- Any programmer could do it.
- ✓ Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS-based solution
(see textbook)

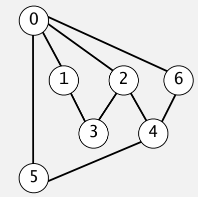


0-1
0-2
0-5
0-6
1-3
2-3
2-4
4-5
4-6

{ 0, 3, 4 }

Graph-processing challenge 2

Problem. Find a cycle.



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

0-5-4-6-0

How difficult?

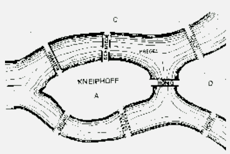
- Any programmer could do it.
- ✓ Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

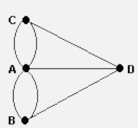
simple DFS-based solution
(see textbook)

Bridges of Königsberg

The Seven Bridges of Königsberg. [Leonhard Euler 1736]

> *" … in Königsberg in Prussia, there is an island A, called the Kneiphof; the river which surrounds it is divided into two branches … and these branches are crossed by seven bridges. Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he could cross each bridge once and only once."*



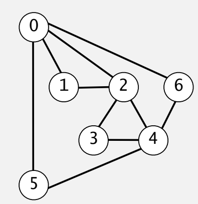


Euler tour. Is there a (general) cycle that uses each edge exactly once?

Answer. Yes iff connected and all vertices have even degree.

Graph-processing challenge 3

Problem. Find 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

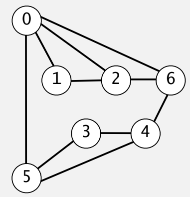
How difficult?

- Any programmer could do it.
- ✓ Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Eulerian tour
(classic graph-processing problem)

Graph-processing challenge 4

Problem. Find a cycle that visits every vertex exactly once.



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

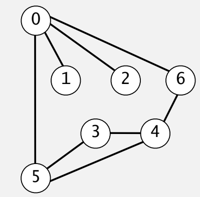
0–5–3–4–6–2–1–0

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- ✓ Intractable. ← Hamiltonian cycle (classical NP-complete problem)
- No one knows.
- Impossible.

Graph-processing challenge 5

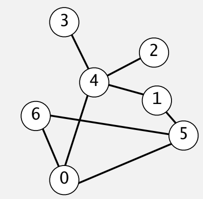
Problem. Are two graphs identical except for vertex names?



0–1
0–2
0–5
0–6
3–4
3–5
4–5
4–6

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- ✓ No one knows. ← graph isomorphism is longstanding open problem
- Impossible.

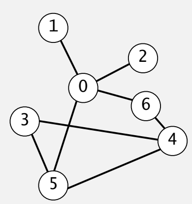


0–4
0–5
0–6
1–4
1–5
2–4
3–4
5–6

0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

Graph-processing challenge 6

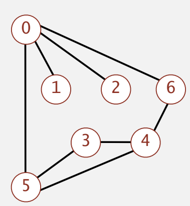
Problem. Lay out a graph in the plane without crossing edges?



0–1
0–2
0–5
0–6
3–4
3–5
4–5
4–6

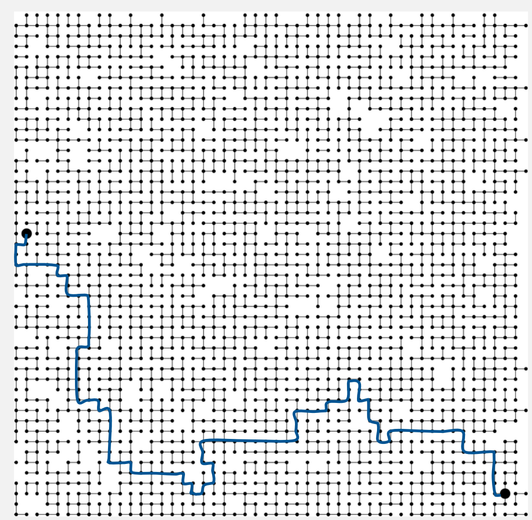
How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- ✓ Hire an expert. ← linear-time DFS-based planarity algorithm discovered by Tarjan in 1970s (too complicated for most practitioners)
- Intractable.
- No one knows.
- Impossible.



Graph-processing challenge 7

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

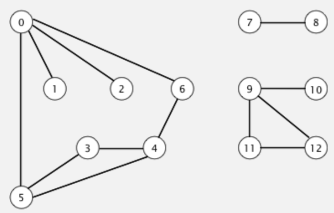


Graph-processing challenge 7

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

How difficult?

- ✓ Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



Paths in graphs: union-find vs. DFS

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

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

Effectively constant with path compression.

DFS preprocessing time. Use connected component algorithm. E+V time.
DFS query time. Simply look up in id[] array.

Union-find. Can intermix connected queries and edge insertions.
Depth-first search. edgeTo[] provides an actual path.