CS171 Introduction to Computer Science II

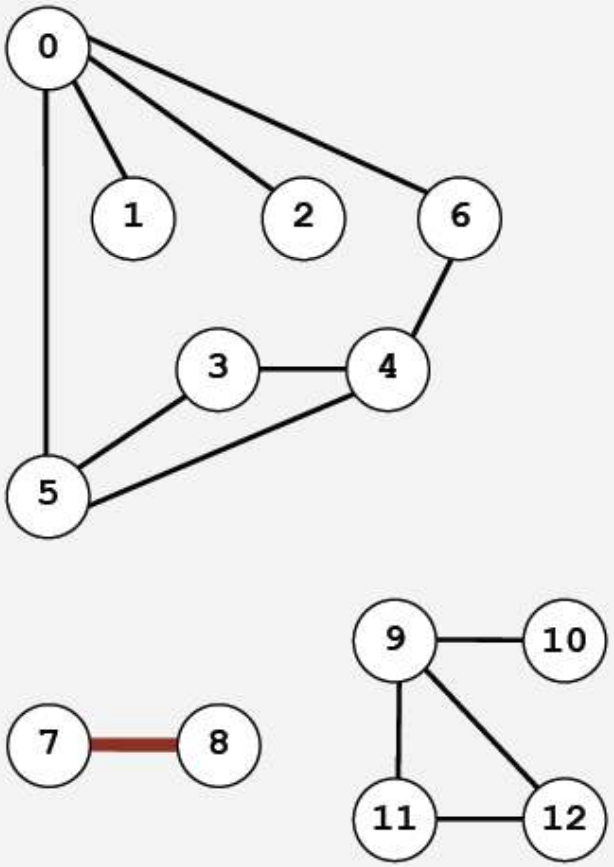
Graphs

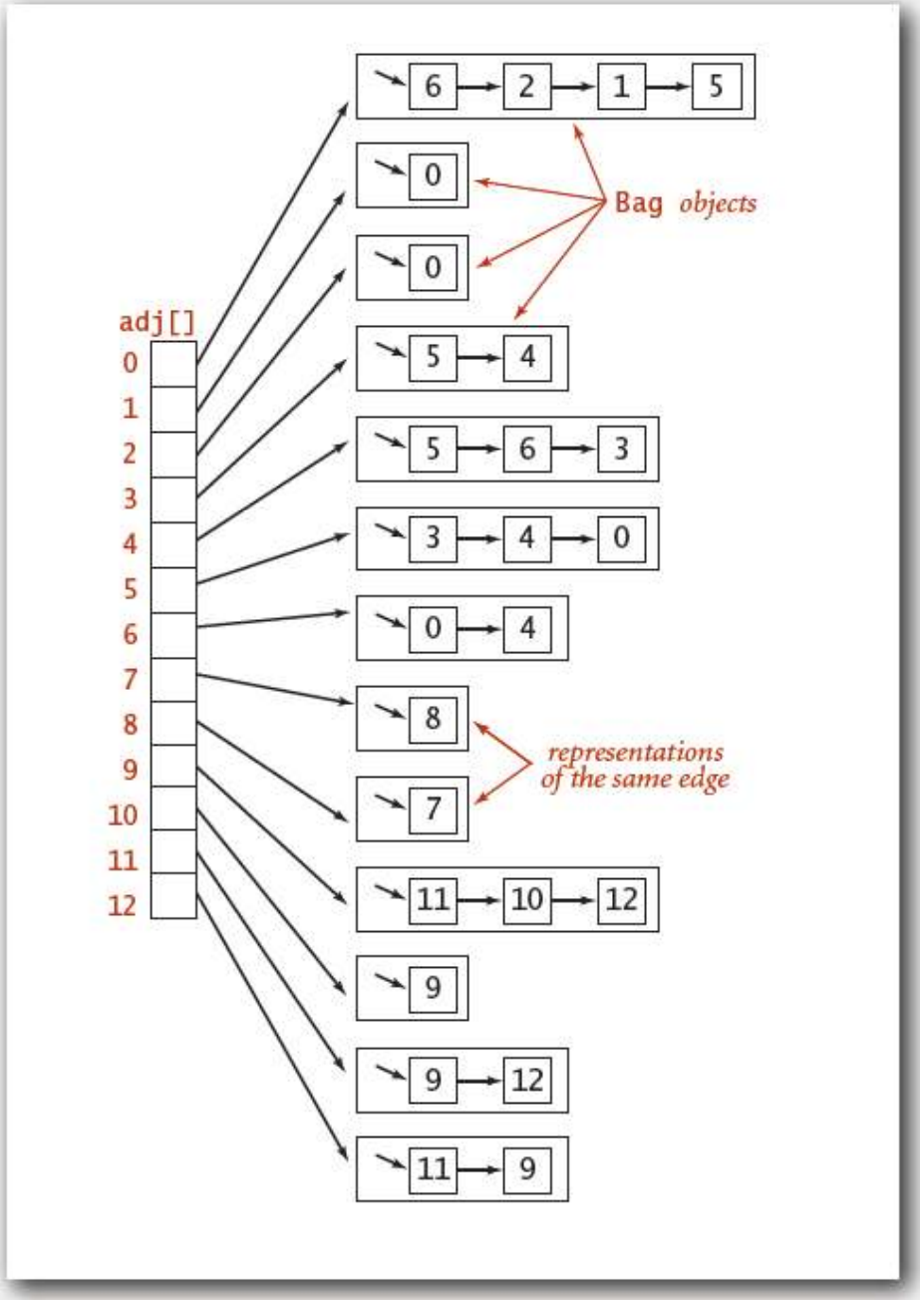
Graphs

- Definitions
- Implementation/Representation of graphs
- Search
  - Depth-first search
  - Breadth-first search
- Applications
  - Find a path
  - Connected component
  - Shortest path

Adjacency-list graph representation

Maintain vertex-indexed array of lists.





Depth-first search

Goal. Systematically search through a graph.

Idea. Mimic maze exploration.

DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked vertices w adjacent to v.

Typical applications. [ahead]

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

Pathfinding in graphs

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

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

Depth-first search (pathfinding)

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

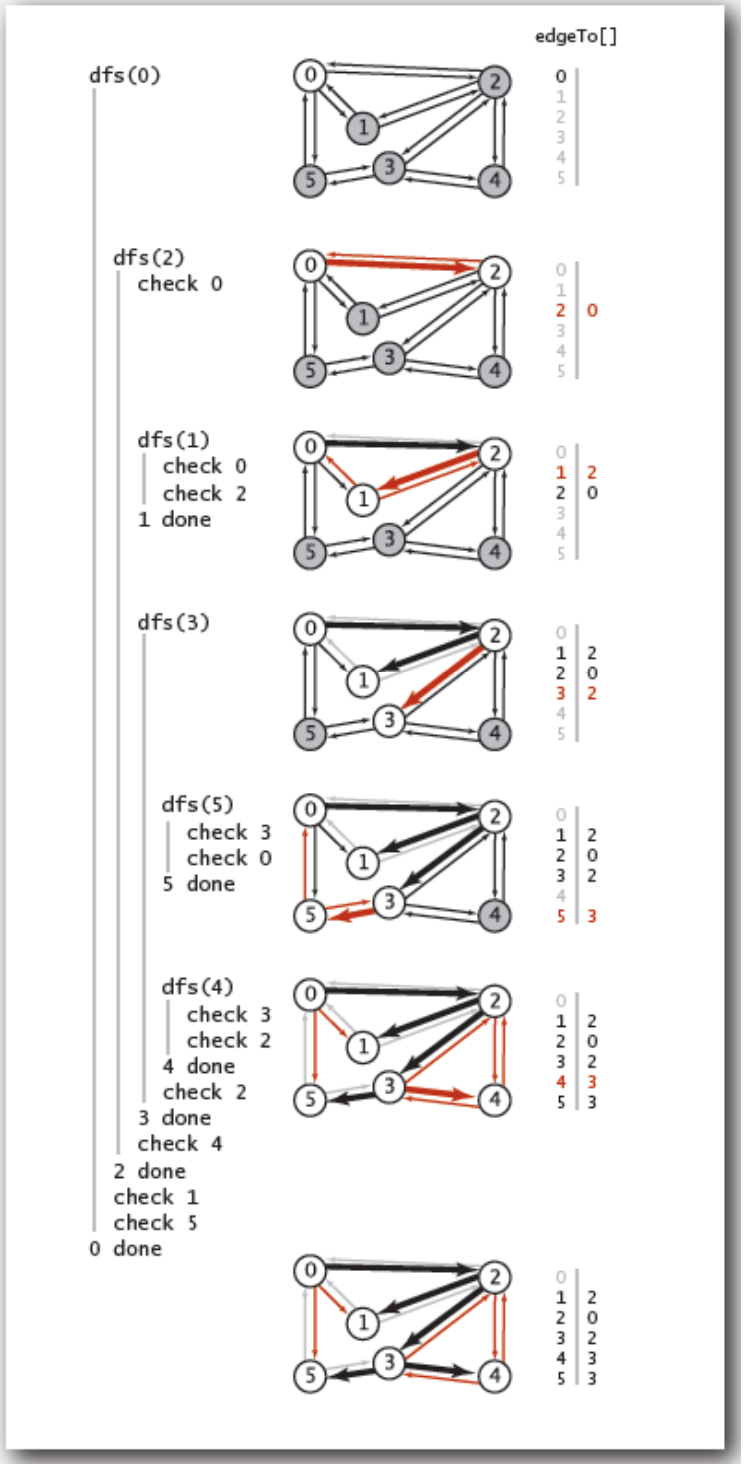
Idea. Mimic maze exploration.

Algorithm.

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

Data structures.

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



Depth-first search (pathfinding)

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

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

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

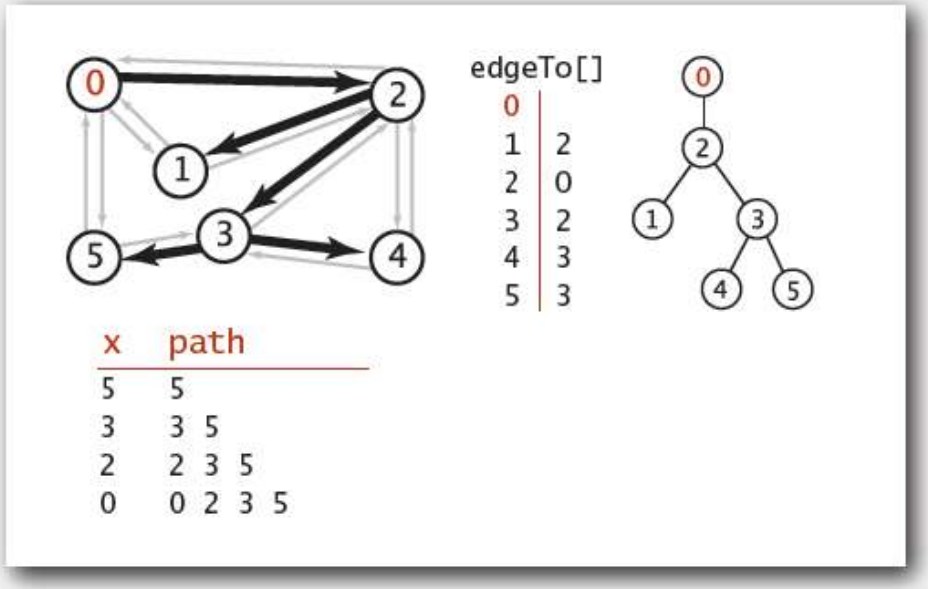
parent-link representation of DFS tree

set parent link

ahead

Depth-first search (pathfinding iterator)

`edgeTo[]` is a 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;
}
```

Connectivity queries

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

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

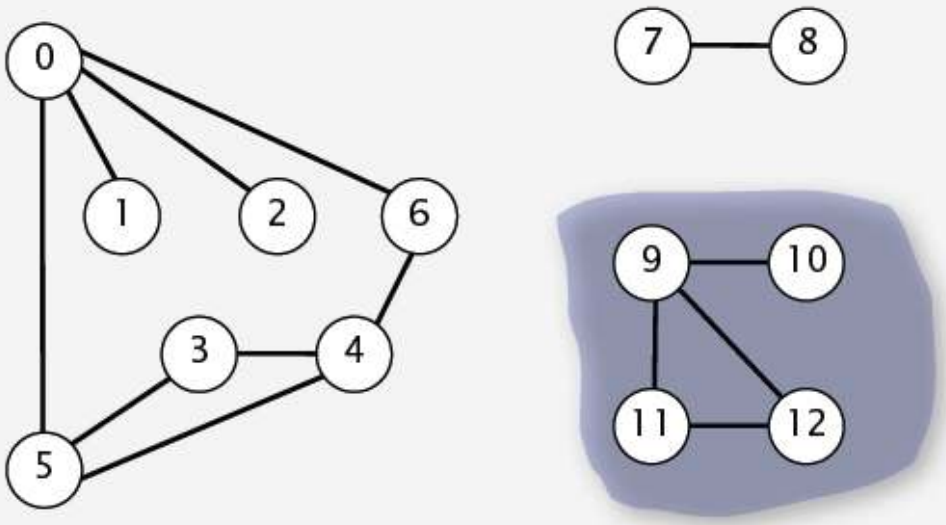
public class CC	
CC(Graph G)	*find connected components in G*
boolean connected(int v, int w)	*are v and w connected?*
int count()	*number of connected components*
int id(int v)	*component identifier for v*

Connected components

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

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

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



3 connected components

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

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

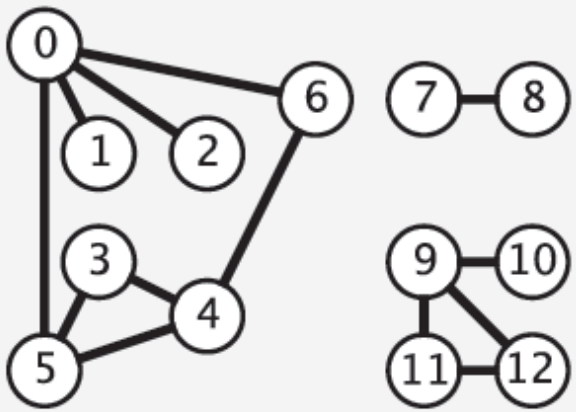
Connected components

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.



Finding connected components with DFS

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

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

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

}
```

id[v] = id of component containing v

number of components

run DFS from one vertex in each component

see next slide

Finding connected components with DFS (continued)

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

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

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

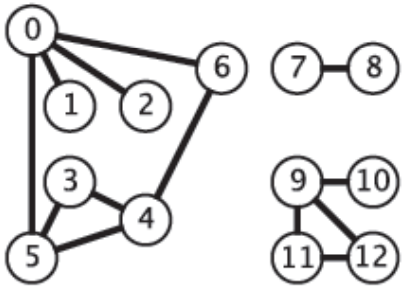
number of components

id of component containing v

all vertices discovered in same call of dfs have same id

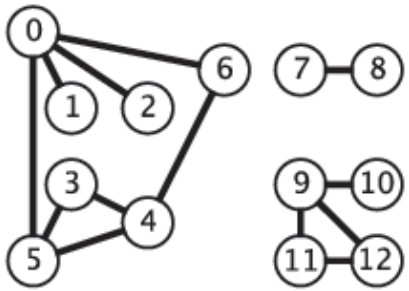
Finding connected components with DFS (trace)

	count	marked[]	id[]
		0 1 2 3 4 5 6 7 8 9 10 11 12	0 1 2 3 4 5 6 7 8 9 10 11 12
dfs(0)	0	T	0
dfs(6)	0	T T	0 0
check 0			
dfs(4)	0	T T T	0 0 0
dfs(5)	0	T T T T	0 0 0 0
dfs(3)	0	T T T T T	0 0 0 0 0
check 5			
check 4			
3 done			
check 4			
check 0			
5 done			
check 6			
check 3			
4 done			
6 done			
dfs(2)	0	T T T T T T	0 0 0 0 0 0
check 0			
2 done			
dfs(1)	0	T T T T T T T	0 0 0 0 0 0 0
check 0			
1 done			
check 5			
0 done			

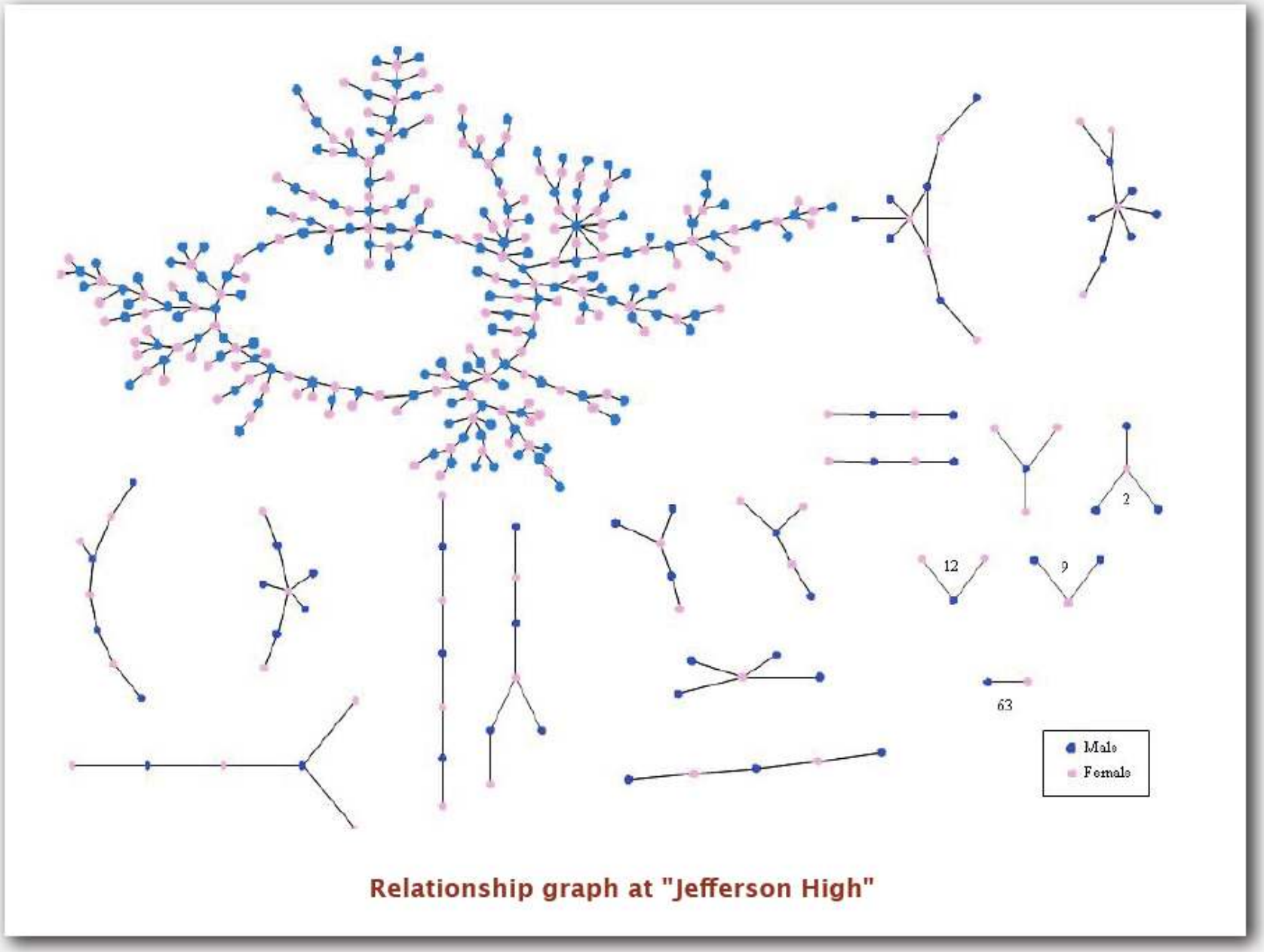


Finding connected components with DFS (trace)

	count	marked[]	id[]
		0 1 2 3 4 5 6 7 8 9 10 11 12	0 1 2 3 4 5 6 7 8 9 10 11 12
0 done			
dfs(7)	1	T T T T T T T T	0 0 0 0 0 0 0 1
| dfs(8)	1	T T T T T T T T T	0 0 0 0 0 0 0 1 1
| | check 7			
| 8 done			
7 done			
dfs(9)	2	T T T T T T T T T T	0 0 0 0 0 0 0 1 1 2
| dfs(11)	2	T T T T T T T T T T T	0 0 0 0 0 0 0 1 1 2 2
| | check 9			
| | dfs(12)	2	T T T T T T T T T T T T	0 0 0 0 0 0 0 1 1 2 2 2
| | | check 11			
| | | check 9			
| | 12 done			
| 11 done			
| dfs(10)	2	T T T T T T T T T T T T T	0 0 0 0 0 0 0 1 1 2 2 2 2
| | check 9			
| 10 done			
| check 12			
9 done			



Connected components application: study spread of STDs



Relationship graph at "Jefferson High"

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

Graph Search

- Depth-first search
  - Finding a path
  - Connected components
- Breadth-first search

Breadth-first search

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

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

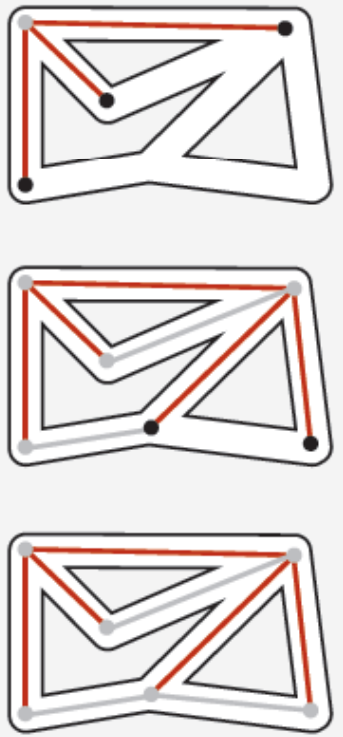
Shortest path. Find path from s to t that uses fewest number of edges.

BFS (from source vertex s)

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

Repeat until the queue is empty:

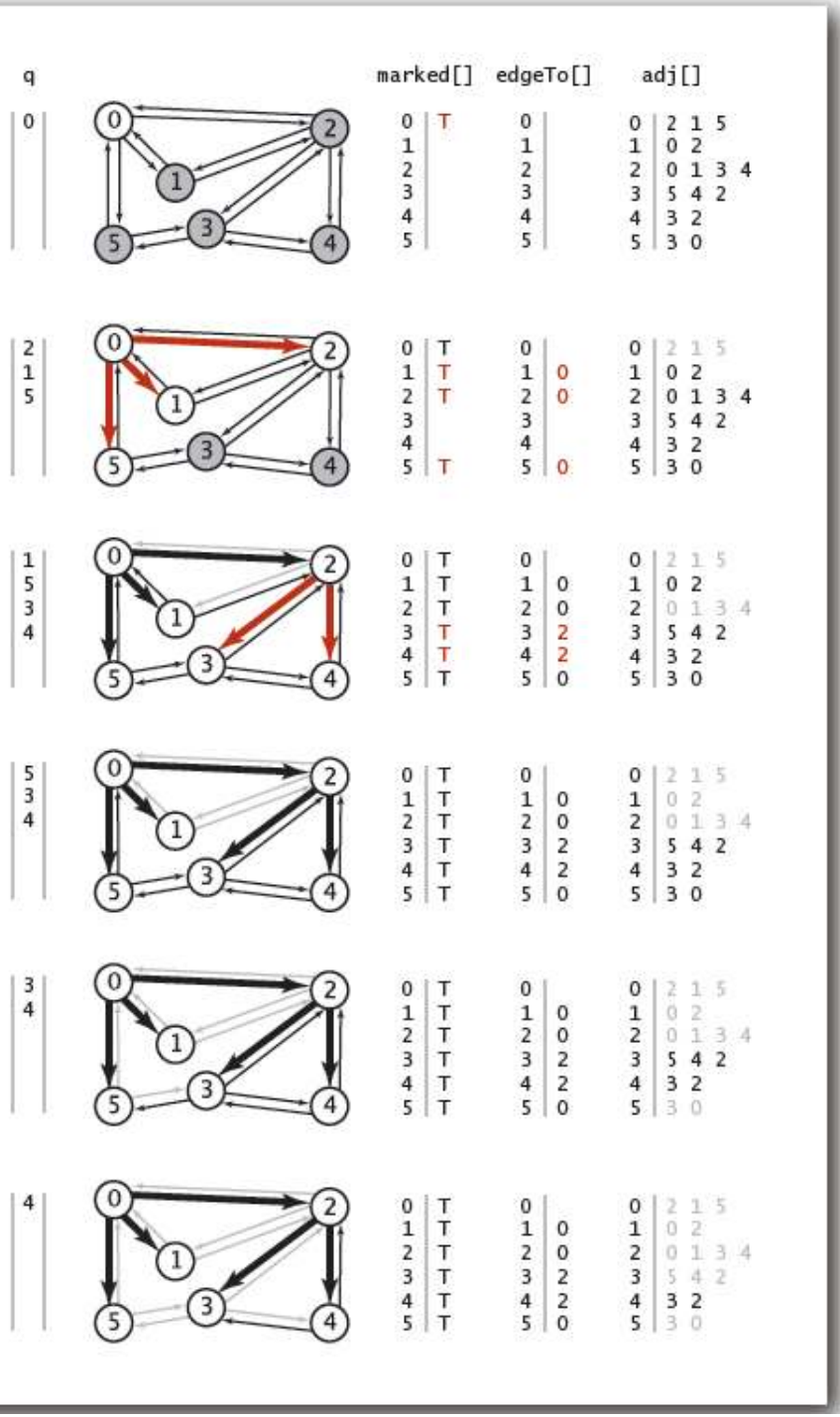
- remove the least recently added vertex v
- add each of v's unvisited neighbors to the queue, and mark them as visited.



Intuition. BFS examines vertices in increasing distance from s.

Breadth-first search (pathfinding)

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



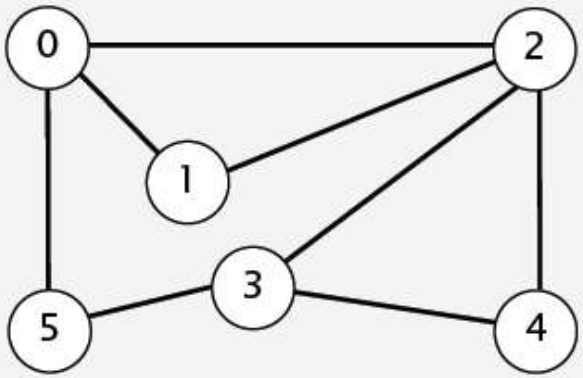
Breadth-first search properties

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

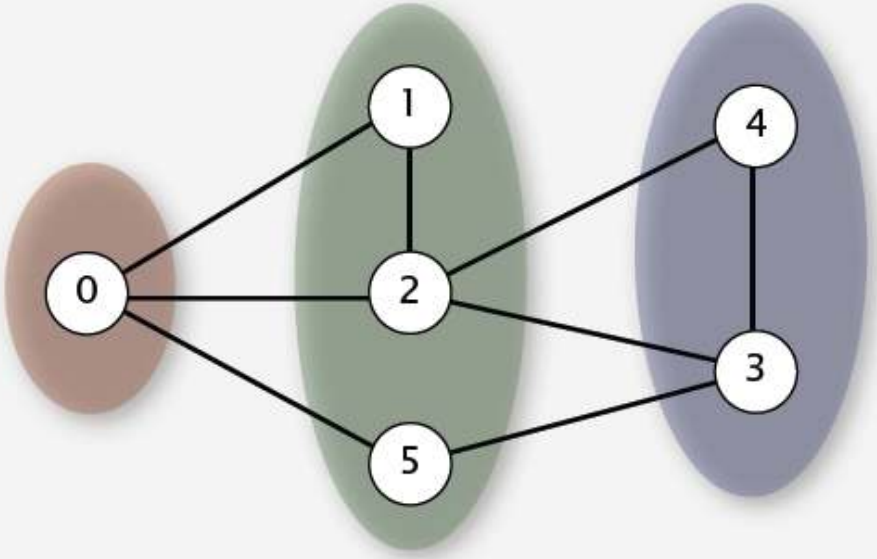
Pf.

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

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



standard drawing

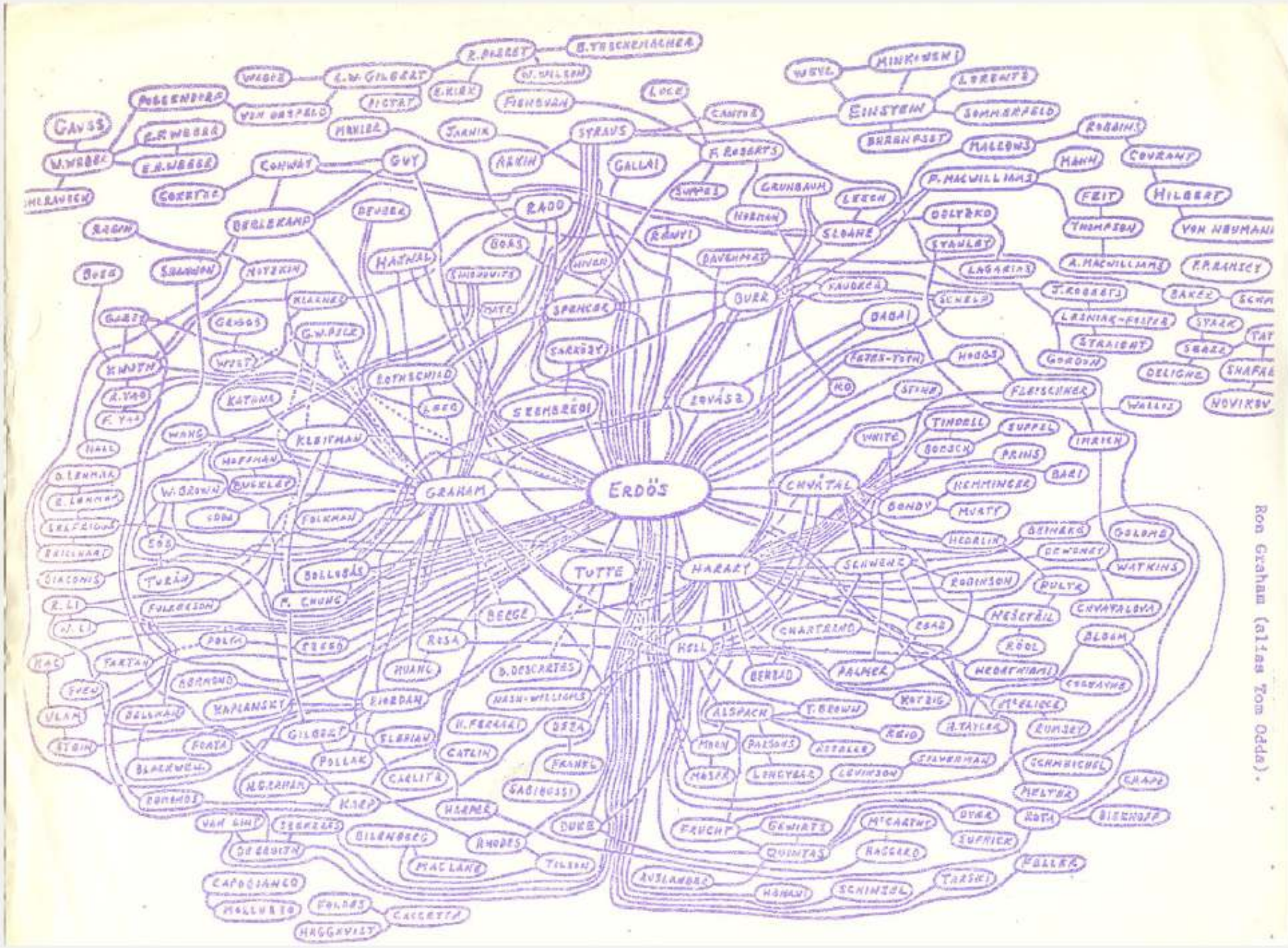


dist = 0 dist = 1 dist = 2

Six degrees of separation

- Everyone is on average approximately six steps away, by way of introduction, from any other person on Earth
- Online social networks
  - Facebook: average distance is 4.74 (Nov 2011)
  - Twitter: average distance is 4.67
- Erdos number
- Bacon number

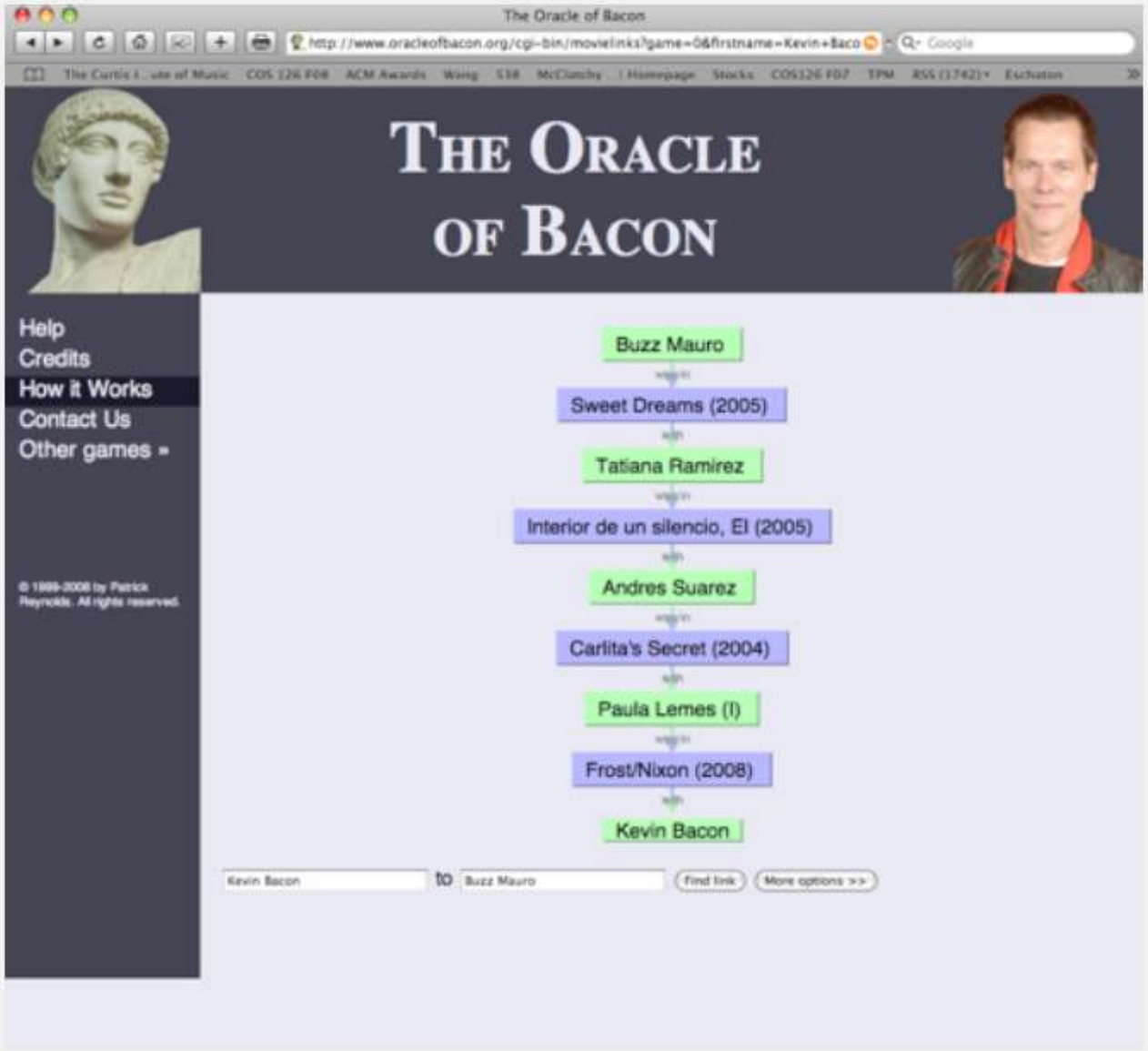
Breadth-first search application: Erdös numbers



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

Breadth-first search application: Kevin Bacon numbers

Kevin Bacon numbers.



http://oracleofbacon.org

Endless Games board game

SixDegrees iPhone App

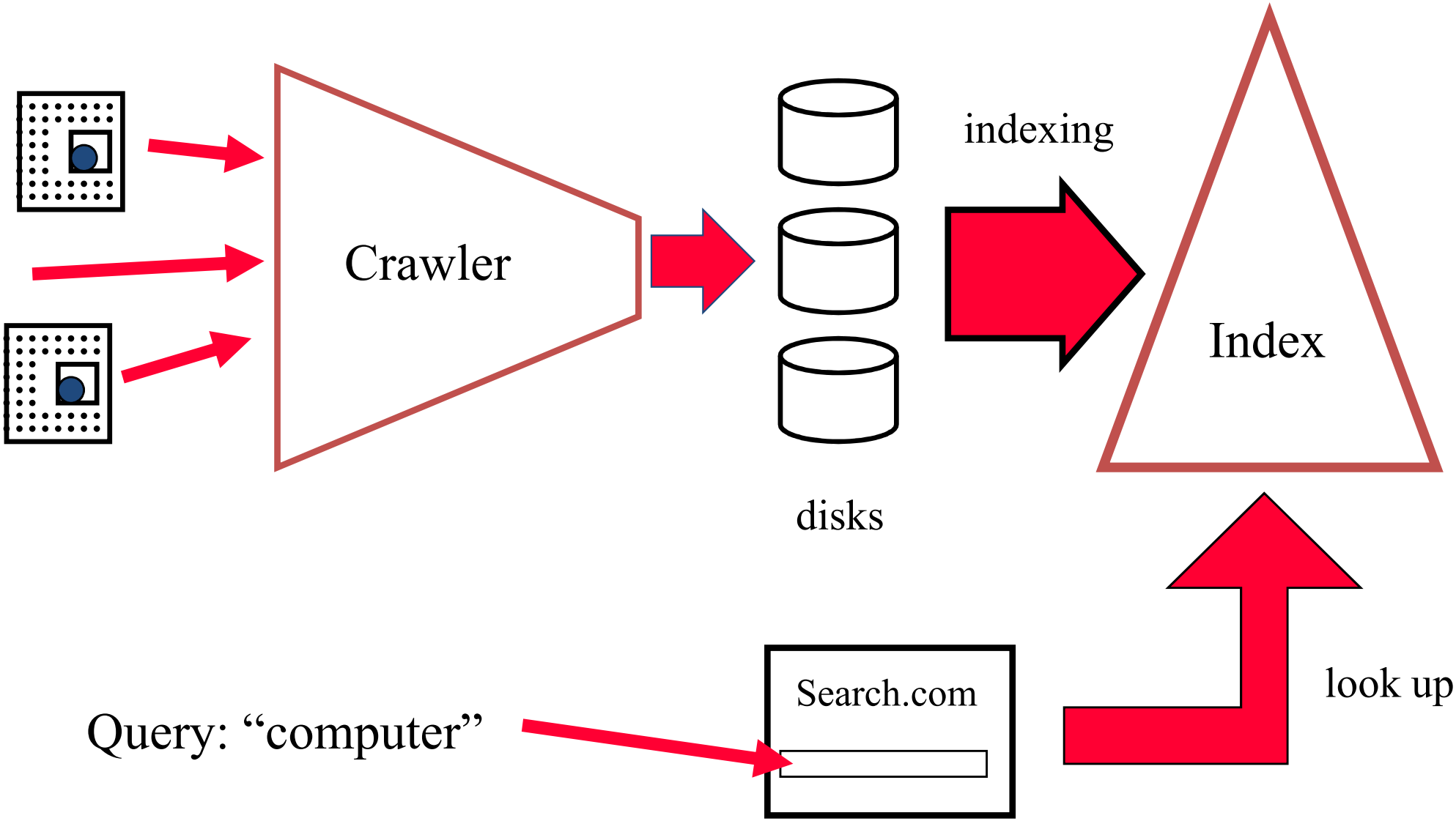
Map Routing (Shortest Path)

Application: Web Search Engines

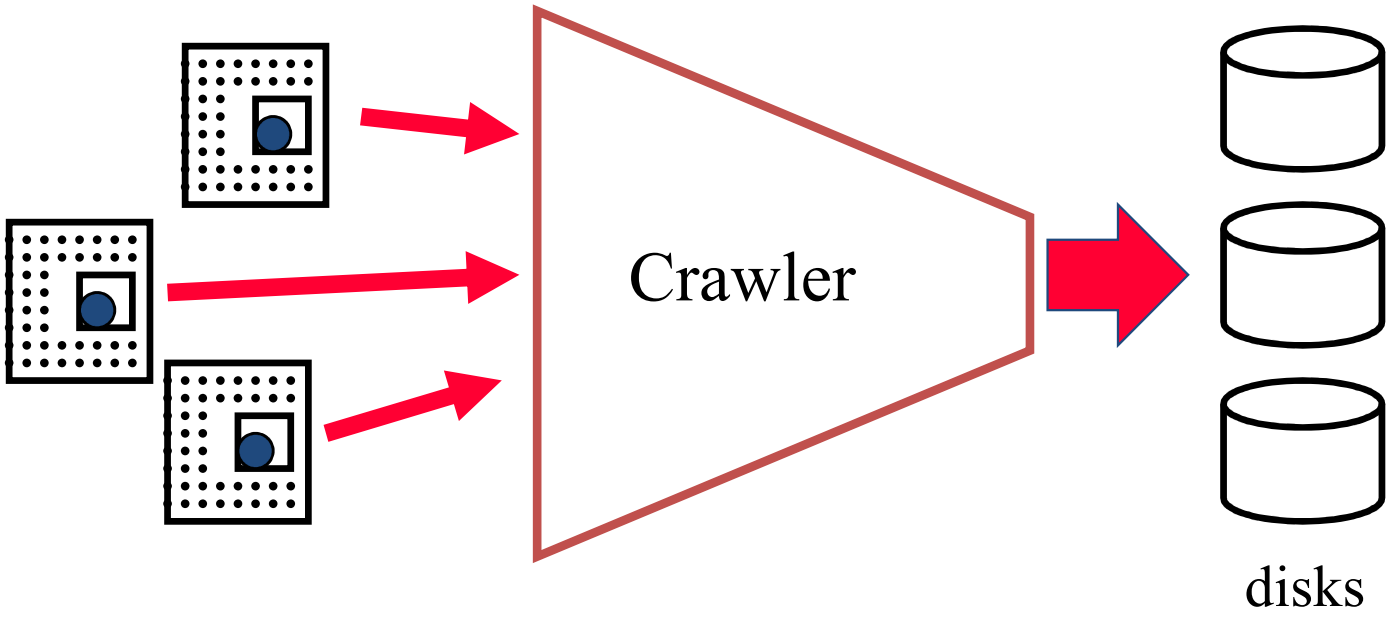
A Search Engine does three main things:

i. Gather the contents of all web pages (using a program called a **crawler** or **spider**)
ii. Organize the contents of the pages in a way that allows efficient retrieval (**indexing**)
iii. Take in a query, determine which pages match, and show the results (**ranking** and **display** of results)

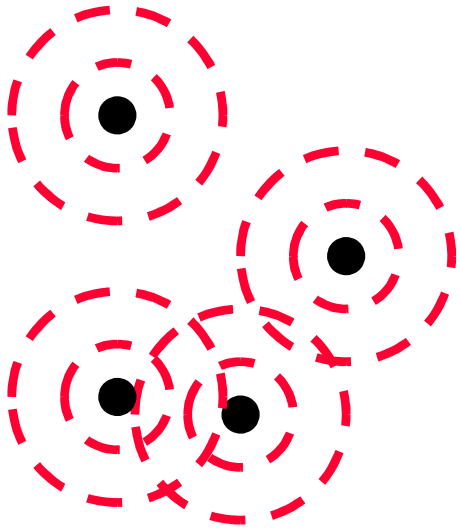
Basic structure of a search engine:



Crawler



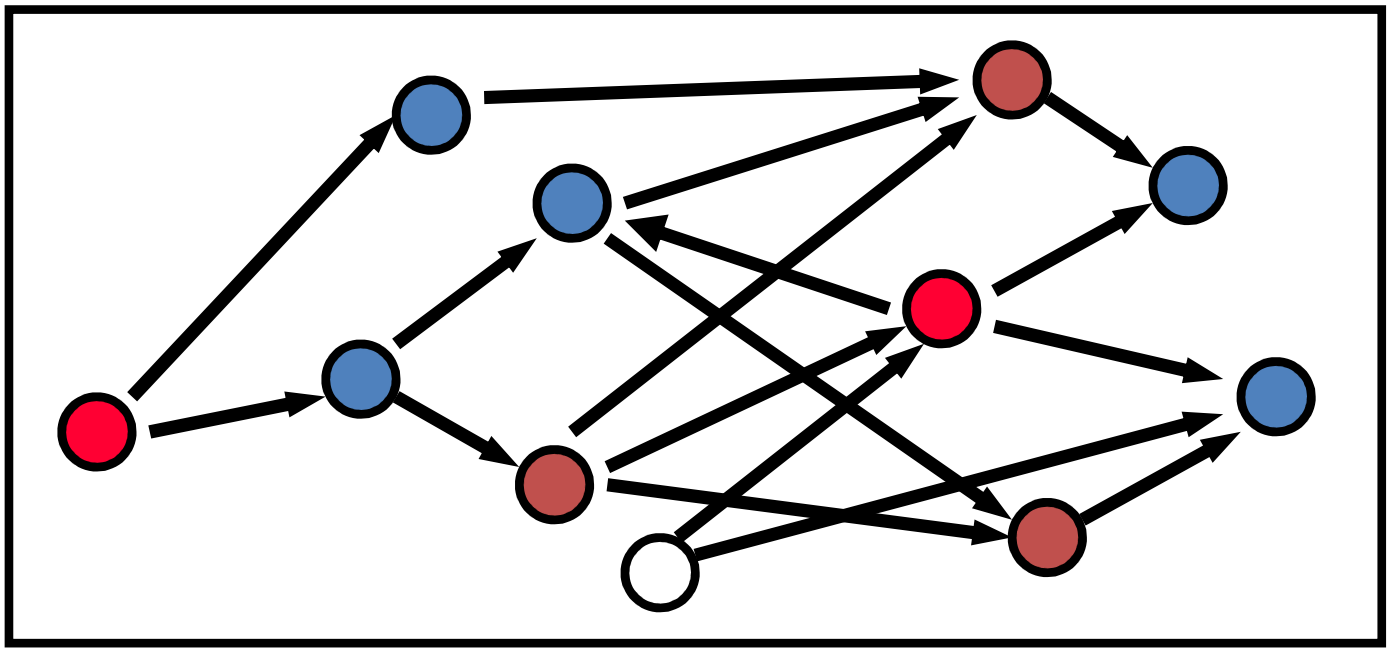
- fetches pages from the web
- starts at set of "seed pages"
- parses fetched pages for hyperlinks
- then follows those links
- variations:
  - recrawling
  - focused crawling
  - random walks



Breadth-First Crawl:

- Basic idea:
  - start at a set of known URLs
  - explore in “concentric circles” around these URLs

start pages
distance-one pages
distance-two pages



Project

- Option 1: The Emory MapQuest Project (TEMP)
- Option 2: FaceSpace
- Option 3: Oracle of Bacon (?)

- Project workshop: May 1, 2012

Midterm Exam

- Maximum: 101
- Mean: 86.57
- Median: 90.5