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.

```java
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.
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)
Depth-first search (pathfinding iterator)

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

```java
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.

```java
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.

![Diagram of connected components]

3 connected components

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

```java
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)

```java
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)

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>marked[]</th>
<th>id[]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12</td>
</tr>
<tr>
<td>dfs(0)</td>
<td>0</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>dfs(6)</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>check 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfs(4)</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>dfs(5)</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>dfs(3)</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>check 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 done</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 done</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 done</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfs(2)</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>check 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 done</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfs(1)</td>
<td>0</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>check 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 done</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 done</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finding connected components with DFS (trace)

<table>
<thead>
<tr>
<th>count</th>
<th>marked[]</th>
<th>id[]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12</td>
</tr>
<tr>
<td></td>
<td>1 T T T T T T T T T T</td>
<td>0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>dfs(7)</td>
<td></td>
<td>1 T T T T T T T T T T</td>
</tr>
<tr>
<td></td>
<td>2 T T T T T T T T T T T T</td>
<td>0 0 0 0 0 0 0 1 1 2</td>
</tr>
<tr>
<td>dfs(8)</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>check 7</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>8 done</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>dfs(9)</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>dfs(11)</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>check 9</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>dfs(12)</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>check 11</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>check 9</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>12 done</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>11 done</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>dfs(10)</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>check 9</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>10 done</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>check 12</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
<tr>
<td>9 done</td>
<td></td>
<td>2 T T T T T T T T T T T</td>
</tr>
</tbody>
</table>

Graph:

```
0 ——— 1 ——— 2
  |       |
  v       v
  3 ——— 4

6 ——— 7 ——— 8

9 ——— 10 ——— 11 ——— 12
```
Connected components application: study spread of STDs

Relationship graph at "Jefferson High"

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 $s$. 
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.
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
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

Indexing

Index

Disk

Query: “computer”

Search.com

Look up
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