CS171 Introduction to Computer Science II

Graphs
Graphs

• Definitions
• Implementation/Representation of graphs

• Traversal and path finding
  – Depth-first search
    • Path finding
  – Breadth-first search
    • Path finding (shortest path)

• Additional applications
  – Connected component
  – Shortest path
Adjacency-list graph representation

Maintain vertex-indexed array of lists.
Traversing graphs

• Graph traversal: visit each vertex in the graph exactly once

• There are in general two ways to traverse a graph
  – Depth-first search (DFS): Uses a Stack or recursion
    • Begins at a node, explores as far as possible along each branch before backtracking
  – Breath-first search (BFS): uses a Queue
    • Begins at a node, explores all its neighboring nodes. Then for each of those nodes, explores their unexplored neighbor nodes, and so on
Maze exploration

Maze graphs.
- **Vertex** = intersection.
- **Edge** = passage.

**Goal.** Explore every intersection in the maze.
Depth-First Search (DFS) – Nonrecursive algorithm

• Push s onto a stack

• Repeat until the stack is empty:
  – remove the top vertex v, if not visited, mark as visited
  – add all v’s unvisited neighbors to the stack
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.
public class DepthFirstSearch
{
    private boolean[] marked;

    public DepthFirstSearch(Graph G, int s)
    {
        marked = new boolean[G.V()];
        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);
        }
    }

    public boolean marked(int v)
    { return marked[v]; } }

marked[v] = true if v connected to s
constructor marks vertices connected to s
recursive DFS does the work
client can ask whether vertex v is connected to s
Depth-first search

**Goal.** Find all vertices connected to s.

**Idea.** Mimic maze exploration.

**Algorithm.**
- Use recursion (ball of string).
- Mark each visited vertex.
- Return (retrace steps) when no unvisited options.

**Data structure.**
- boolean[] marked to mark visited vertices.
Graphs

• Definitions
• Implementation/Representation of graphs
• Traversal and path finding
  – Depth-first search
    • Path finding
  – Breadth-first search
    • Path finding (shortest path)
• Additional applications
  – Connected component
  – Shortest path
Maze exploration

Maze graphs.
- Vertex = intersection.
- Edge = passage.

Goal. Explore every intersection in the maze.
Pathfinding in graphs

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

```java
public class Paths {
    public 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 $s$.


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.
- $\text{boolean[]} \text{ marked}$ to mark visited vertices.
- $\text{int[]} \text{ edgeTo}$ to keep tree of paths.
- $(\text{edgeTo}[w] = v)$ 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 $s$.

![Graph](image.png)

<table>
<thead>
<tr>
<th>$x$</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3 5</td>
</tr>
<tr>
<td>2</td>
<td>2 3 5</td>
</tr>
<tr>
<td>0</td>
<td>0 2 3 5</td>
</tr>
</tbody>
</table>

```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;
}
```
Graphs

• Definitions
• Implementation/Representation of graphs

• Traversal and path finding
  – Depth-first search
    • Path finding
  – Breadth-first search
    • Path finding (shortest path)

• Additional applications
  – Connected component
  – Shortest path
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 $s$)**

<table>
<thead>
<tr>
<th>Put $s$ onto a FIFO queue, and mark $s$ as visited.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat until the queue is empty:</td>
</tr>
<tr>
<td>- remove the least recently added vertex $v$</td>
</tr>
<tr>
<td>- add each of $v$'s unvisited neighbors to the queue,</td>
</tr>
<tr>
<td>and mark them as visited.</td>
</tr>
</tbody>
</table>

---

**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.
Graphs

- Definitions
- Implementation/Representation of graphs
- Traversal and path finding
  - Depth-first search
    - Path finding
  - Breadth-first search
    - Path finding (shortest path)
- Additional applications
  - Shortest path
  - Connected component
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: Erdős numbers

Hand-drawing of part of the Erdős 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 → Index

Query: “computer” → Search.com

indexing → disks
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

![Diagram of Breadth-First Crawl]

- start pages
- distance-one pages
- distance-two pages
Graphs

• Definitions
• Implementation/Representation of graphs
• Traversal and path finding
  – Depth-first search
    • Path finding
  – Breadth-first search
    • Path finding (shortest path)
• Additional applications
  – Shortest path
  – Connected component
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.

<table>
<thead>
<tr>
<th>public class <strong>CC</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CC(Graph G)</strong></td>
</tr>
<tr>
<td>boolean <strong>connected</strong>(int $v$, int $w$)</td>
</tr>
<tr>
<td>int <strong>count</strong>()</td>
</tr>
<tr>
<td>int <strong>id</strong>(int $v$)</td>
</tr>
</tbody>
</table>
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

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

DFS Tracing:
- dfs(0)
  - check 0
- dfs(6)
  - check 0
  - dfs(4)
    - check 4
    - done
- dfs(5)
  - check 5
  - done
- dfs(3)
  - check 4
  - done
- dfs(2)
  - check 0
  - done
- dfs(1)
  - check 0
  - done
Finding connected components with DFS (trace)

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

The graph shown below illustrates the connected components found with DFS.
Connected components application: study spread of STDs

Relationship graph at "Jefferson High"