4.1 UNDIRECTED GRAPHS

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges

With added notes and slides by Betty O’Neil for cs310
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- 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.
Border graph of 48 contiguous United States
Protein-protein interaction network

Reference: Jeong et al, Nature Review Genetics
Map of science clickstreams

http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803
10 million Facebook friends

"Visualizing Friendships" by Paul Butler
The evolution of FCC lobbying coalitions
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, $\geq 30$) 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 Internet as mapped by the Opte Project

http://en.wikipedia.org/wiki/Internet
## Graph applications

<table>
<thead>
<tr>
<th>graph</th>
<th>vertex</th>
<th>edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>telephone, computer</td>
<td>fiber optic cable</td>
</tr>
<tr>
<td>circuit</td>
<td>gate, register, processor</td>
<td>wire</td>
</tr>
<tr>
<td>mechanical</td>
<td>joint</td>
<td>rod, beam, spring</td>
</tr>
<tr>
<td>financial</td>
<td>stock, currency</td>
<td>transactions</td>
</tr>
<tr>
<td>transportation</td>
<td>intersection</td>
<td>street</td>
</tr>
<tr>
<td>internet</td>
<td>class C network</td>
<td>connection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>social relationship</td>
<td>person</td>
<td>friendship</td>
</tr>
<tr>
<td>neural network</td>
<td>neuron</td>
<td>synapse</td>
</tr>
<tr>
<td>protein network</td>
<td>protein</td>
<td>protein-protein interaction</td>
</tr>
<tr>
<td>molecule</td>
<td>atom</td>
<td>bond</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>problem</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s-t path</strong></td>
<td><em>Is there a path between s and t?</em></td>
</tr>
<tr>
<td><strong>shortest s-t path</strong></td>
<td><em>What is the shortest path between s and t?</em></td>
</tr>
<tr>
<td><strong>cycle</strong></td>
<td><em>Is there a cycle in the graph?</em></td>
</tr>
<tr>
<td><strong>Euler cycle</strong></td>
<td><em>Is there a cycle that uses each edge exactly once?</em></td>
</tr>
<tr>
<td><strong>Hamilton cycle</strong></td>
<td>*Is there a cycle that uses each vertex exactly once?</td>
</tr>
<tr>
<td><strong>connectivity</strong></td>
<td><em>Is there a way to connect all of the vertices?</em></td>
</tr>
<tr>
<td><strong>biconnectivity</strong></td>
<td><em>Is there a vertex whose removal disconnects the graph?</em></td>
</tr>
<tr>
<td><strong>planarity</strong></td>
<td><em>Can the graph be drawn in the plane with no crossing edges?</em></td>
</tr>
<tr>
<td><strong>graph isomorphism</strong></td>
<td><em>Do two adjacency lists represent the same graph?</em></td>
</tr>
</tbody>
</table>

**Challenge.** Which graph problems are easy? difficult? intractable?
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
Graph representation

**Graph drawing.** Provides intuition about the structure of the 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: none available in JDK, so we’ll use S&W graph classes

```java
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
}
```

**CS310 Notes:** This API has no addVertex method, no hasEdge(v, w), unwelcome involvement with S&W specific i/o class “In”

```java
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);
```
Graph API: sample client

Graph input format.

```
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);
```

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

read graph from input stream
print out each edge (twice)
We can create a S&W Graph from a file using JDK file i/o:

```java
public static Graph createGraphFromFile(String filePath) {
    Scanner in = null;
    try {
        in = new Scanner(new File(filePath));
    } catch (FileNotFoundException e) {
        System.out.println("File not found: "+ filePath);
        return null;
    }
    int nV = Integer.parseInt(in.nextLine());
    int nE = Integer.parseInt(in.nextLine());
    Graph G = new Graph(nV);
    while (in.hasNextLine()) {
        String line1 = in.nextLine();
        String[] tokens = line1.split(" ");
        G.addEdge(Integer.parseInt(tokens[0]),
                   Integer.parseInt(tokens[1]));
    }
    in.close();
    return G;
}
```
Then the little test program looks like this, where use of `Graph(In in)` is replaced by `createGraphFromFile`, specifically:

```java
In in = new In(args[0]);
Graph G = new Graph(in); // Let Graph use In to fill itself up
```

is replaced by

```java
Graph G = createGraphFromFile(args[0]); // use JDK file i/o to read file, call Graph API to add data to Graph.
```

```java
Graph G = createGraphFromFile(args[0]); // create G from given data
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v)) // go thru adjacency list of v
        System.out.println(v + "-" + w);
}
```

% java Test tinyG.txt
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
Same edge seen from each end
You can download this code: see TestGraph(zip) on our home page under today’s class

After unzip, see directories src and lib
Base directory: has README and tinyG.txt
src: has TestGraph.java, with import edu.Princeton.cs.algs4.*;
lib: has algs4.jar, library of S&W code (a compressed directory tree of .class files)
README says:

To build TestGraph:
in src: javac -cp ../lib/algs4.jar TestGraph.java (Windows/Mac/Linux)

To run TestGraph:
in src: java -cp ../lib/algs4.jar:. TestGraph ../tinyG.txt (Windows)
java -cp ../lib/algs4.jar:. TestGraph ../tinyG.txt (Mac/Linux)
^different: semi-colon vs. colon
Other ways to use the algs4.jar library

• It is possible to put the library on the CLASSPATH environment variable as S&W suggest on their download page, or put the library in some special place known to Java, also specified there.

• But the approach of the last slide shows what’s happening better in my opinion: we are providing Java access to all these classes in this build.

• If you are using eclipse, just use “Open Projects from File System”, then Directory, and specify the TestGraph directory and confirm. Eclipse will take the hint that algs4.jar is a project library because it is in directory “lib”. So it will build the program immediately.
## Typical graph-processing code

```java
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;
}
```
This graph API and implementation is based on adjacency lists, but that is not the only approach.

Let’s look at three possible approaches to representing graphs:

1. Set of Edges
2. Adjacency Matrix
3. Adjacency Lists
Set-of-edges graph representation

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

Q. How long to iterate over vertices adjacent to $v$?
Maintain a two-dimensional $V$-by-$V$ boolean array; for each edge $v$–$w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$.

Q. How long to iterate over vertices adjacent to $v$?
Adjacency-list graph representation

Maintain vertex-indexed array of lists.

Q. How long to iterate over vertices adjacent to \( v \)?
Graph representations

**In practice.** Use adjacency-lists representation.
- Algorithms based on iterating over vertices adjacent to \( v \).
- Real-world graphs tend to be sparse.

Two graphs (\( V = 50 \))

- **Sparse** (\( E = 200 \))
- **Dense** (\( E = 1000 \))

huge number of vertices, small average vertex degree
Graph representations

**In practice.** Use adjacency-lists representation. Algorithms based on iterating over vertices adjacent to \( v \).

Real-world graphs tend to be **sparse**.

---

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>add edge</th>
<th>edge between ( v ) and ( w )?</th>
<th>iterate over vertices adjacent to ( v )?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>( E )</td>
<td>1</td>
<td>( E )</td>
<td>( E )</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>( V^2 )</td>
<td>1 *</td>
<td>1</td>
<td>( V )</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>( E + V )</td>
<td>1</td>
<td>( \text{degree}(v) )</td>
<td>( \text{degree}(v) )</td>
</tr>
</tbody>
</table>

* disallows parallel edges
Adjacency-list graph representation: Java implementation

```java
class Graph {
    private final int V;
    private Bag<Integer>[] adj;

    Graph(int V) {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    void addEdge(int v, int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    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`
4.1 Undirected Graphs

- Introduction
- Graph API
- Depth-first search
- Breadth-first search
- Connected components
- Challenges
Maze exploration

Maze graph.

- **Vertex** = intersection
- **Edge** = passage

**Goal.** Explore every intersection in the maze.
Trémaux maze exploration

Algorithm.
- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when return to a marked intersection or no unvisited options.
Trémaux maze exploration

Algorithm.
• Unroll a ball of string behind you.
• Mark each visited intersection and each visited passage.
• Retrace steps when return to a marked intersection or no unvisited 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.
Maze exploration: easy
Maze exploration: medium
Depth-first search

**Goal.** Systematically traverse a graph.

**Idea.** Mimic maze exploration.  

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

**Design challenge.** How to implement?

---

**DFS (to visit a vertex \( v \))**

- Mark \( v \) as visited.
- Recursively visit all unmarked vertices \( w \) adjacent to \( v \).

function-call stack acts as ball of string
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

\[
V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}
E = \{(0, 1), (0, 5), (0, 6), (1, 2), (2, 6), (2, 3), (3, 4), (4, 5), (5, 6), (5, 11), (11, 12), (12, 8), (8, 7), (7, 9), (9, 10), (10, 12), (9, 13), (13, 11), (11, 10), (10, 13), (10, 0), (11, 0), (13, 5), (5, 3), (3, 4), (4, 5)\}

**tinyG.txt**

V 13
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
To visit a vertex $v$:
Mark vertex $v$ as visited.
Recursively visit all unmarked vertices adjacent to $v$. 

```
<table>
<thead>
<tr>
<th>v</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>
```

vertices reachable from 0
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.

```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 */

      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
```
To visit a vertex \( v \):
- Mark vertex \( v \) as visited.
- Recursively visit all unmarked vertices adjacent to \( v \).

Data structures.
- Boolean array `marked[]` to mark visited vertices.
- Integer array `edgeTo[]` to keep track of paths.
  \((\text{edgeTo}[w] == v)\) means that edge \( v-w \) taken to visit \( w \) for first time
- Function-call stack for recursion.
Try this `edgeTo` array out...

`(edgeTo[w] == v)` means that edge $v$-$w$ taken to visit $w$ for first time, so here $v$ is the from-vertex and $w$ is the to-vertex.

We see an edge from 0 to 1 taken as first step from 0, so put $\text{edgeTo}[1] = 0$

Second step from 0 to 2, so $\text{edgeTo}[2] = 0$

Then from 0 to 6, so $\text{edgeTo}[6] = 0$

Then 6 to 4, so $\text{edgeTo}[4] = 6$, and so on.
Depth-first search: Java implementation

```java
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]) {
                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
recursive DFS does the work

We see the great advantage in having Integer/int ids for vertices: can use arrays for lookup/storage by vertex.
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, in the sense of being marked and trying to explore its unmarked neighbors. Once marked, it never again is “visited” because of its marking.
**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.** $\text{edgeTo[]}[]$ is parent-link representation of a tree rooted at vertex $s$, so we can use it to step from $s$ to parent to parent’s parent, etc. all the way back to $v$.

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

global 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;
}
```
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** (implicitly).

- **Vertex:** pixel.
- **Edge:** between two adjacent gray pixels.
- **Blob:** all pixels connected to given pixel.
Recall our earlier Image Processing Example doing this: same basic algorithm

private void regionSizeHelper(Set foundSet, int theColor, int x, int y)
    Check if (x,y) is inside the image, return if not
    If the pixel at (x,y) has color theColor:
        Create a Pixel for it.
        Check if that pixel is already in the foundSet
        If not, add it, call the helper on its four adjacent pixels

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;
        }
}
Depth-first search application: preparing for a date

http://xkcd.com/761/