4.1 Undirected Graphs

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

- Box and its subsets

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

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

**Challenge.** Which graph problems are easy? difficult? intractable?

### Graph representation

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

#### Caveat
Intuition can be misleading.

### Graph API: none available in JDK, so we’ll use this API

```java
class Graph {
  // methods for graph creation, modification, and traversal
}
```

**CS310 Notes:** No addVertex method, no hasEdge(i, j), unwelcome involvement with int, S&W specific (i.e. method “in”)

### Graph API: sample client

```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 input format.

```
1 2 3 4
2 4
5 1
3 5 4
0 1 2
5 4
1 3
```

```
3 1 4
```

The standard input form above is a space-separated sequence of vertices and adjacent vertices.

```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 output format.

```
0
1
2
3
4
```

```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 output format.

```
0
1
2
3
4
```
We can create a 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));
        System.out.println("File not found: " + filePath);
        return null;
    } 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);
createGraphFromFile(args[0]); // create G from given data
for (int v = 0; v < G.V(); v++) // go thru adjacency list of v
    System.out.println(v + " - " + G.adj(v));
```

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

```
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
    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
**Algorithms**

Set-of-edges graph representation

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

Adjacency-list graph representation

Maintain vertex-indexed array of lists.

### Graph representations

**In practice.** Use adjacency-lists representation.

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>add edge</th>
<th>edge between</th>
<th>iterate over vertices adjacent to v</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>$O(V+E)$</td>
<td>$O(1)$</td>
<td>$O(E)$</td>
<td>$O(V+E)$</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>$O(V^2)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>$O(V)$</td>
<td>$O(1)$</td>
<td>$O(deg(v))$</td>
<td>$O(deg(v))$</td>
</tr>
</tbody>
</table>

Adjacency-matrix graph representation

Maintain a two-dimensional $V$-by-$V$ boolean array; for each edge $v$-$w$ in graph: $adj[v][w] = adj[w][v] = true$.

Graph representations

In practice. Use adjacency-lists representation.

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

Adjacency-list graph representation: Java implementation

```java
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 i = 0; i < V; i++) {
            adj[i] = 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];
    }
}
```
4.1 UNDIRECTED GRAPHS

Maze exploration

Maze graph:
- Vertex
- Edge

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.

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
Goal. Systematically traverse a graph.

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

Design challenge. How to implement?

Depth-first search: data structures

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.
  (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 edgeTo[1] = 0

Second step from 0 to 2, so edgeTo[2] = 0

Then from 0 to 6, so edgeTo[6] = 0

Then 6 to 4, so edgeTo[4] = 6, and so on.

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.

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.

Depth-first search: Java implementation

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

    public DepthFirstPaths(Graph G, int s)
    {
        this.marked = new boolean[G.V()];
        this.edgeTo = new int[G.V()];
        this.s = s;
        this.dfs(G, s);
    }

    private int dfs(Graph G, int x)
    {
        if (marked[x]) return x;
        marked[x] = true;
        for (int w : G.adj(x))
            edgeTo[w] = x;
        dfs(G, w);
        edgeTo[x] = w;
        return x;
    }

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

    public Iterable<Integer> pathTo(int v)
    {
        Stack<Integer> path = new Stack<Integer> ();
        for (int x = v; x != s; x = edgeTo[x])
            path.push(x);
        return path;
    }
}
```

Depth-first search: properties

Proposition. After DFS, can check if vertex v is connected to s in constant time and can find s-v path (if one exists) in time proportional to its length.

Pf. 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.

Depth-first search application: preparing for a date

Solution. Build a grid graph (implicitly).

• Vertex: pixel.

• Edge: between two adjacent gray pixels.

• Blob: all pixels connected to given pixel.

Assumptions. Picture has millions to billions of pixels.