4.1 **Undirected Graphs**

- introduction
- graph API
- depth-first search
- breadth-first search
- connected components
- challenges
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.
The Internet as mapped by the Opte Project

http://en.wikipedia.org/wiki/Internet
10 million Facebook friends

"Visualizing Friendships" by Paul Butler
## 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>street intersection, airport</td>
<td>highway, airway route</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, actor</td>
<td>friendship, movie cast</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>chemical compound</td>
<td>molecule</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

**Path.**  Is there a path between $s$ and $t$?

**Shortest path.**  What is the shortest path between $s$ and $t$?

**Cycle.**  Is there a cycle in the graph?

**Euler tour.**  Is there a cycle that uses each edge exactly once?

**Hamilton tour.**  Is there a cycle that uses each vertex exactly once.

**Connectivity.**  Is there a way to connect all of the vertices?

**MST.**  What is the best way to connect all of the vertices?

**Biconnectivity.**  Is there a vertex whose removal disconnects the graph?

**Planarity.**  Can you draw the graph in the plane with no crossing edges

**Graph isomorphism.**  Do two adjacency lists represent the same graph?

**Challenge.**  Which of these problems are easy? difficult? intractable?
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

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

    String toString()  // string representation

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);

read graph from input stream
print out each edge (twice)
Typical graph-processing code

**compute the degree of v**

```java
public static int degree(Graph G, int v) {
    int degree = 0;
    for (int w : G.adj(v)) degree++;
    return degree;
}
```

**compute maximum degree**

```java
public static int maxDegree(Graph G) {
    int max = 0;
    for (int v = 0; v < G.V(); v++)
        if (degree(G, v) > max) max = degree(G, v);
    return max;
}
```

**compute average degree**

```java
public static double averageDegree(Graph G) {
    return 2.0 * G.E() / G.V();
}
```

**count self-loops**

```java
public static int numberOfSelfLoops(Graph G) {
    int count = 0;
    for (int v = 0; v < G.V(); v++)
        for (int w : G.adj(v))
            if (v == w) count++;
    return count/2; // each edge counted twice
}
```
Maintain a two-dimensional $V$-by-$V$ boolean array; for each edge $v \rightarrow w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$. 

Adjacency-matrix graph representation
Adjacency-list graph representation

Maintain vertex-indexed array of lists.

Adjacency-lists representation (undirected graph)
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 v = 0; v < V; v++)
            adj[v] = 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];
    }
}
```
**Graph representations**

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

Huge number of vertices, small average vertex degree

Two graphs (\( V = 50 \))

- **Sparse** (\( E = 200 \))
- **Dense** (\( E = 1000 \)
In practice. Use adjacency-lists representation.

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

### Graph representations

<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

huge number of vertices, small average vertex degree
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 no unvisited options.
Maze exploration
Depth-first search

Goal. Systematically search through a graph.

DFS (to visit a vertex v)

Mark v as visited.
Recursively visit all unmarked vertices w adjacent to v.

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

Design challenge. How to implement?
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
}
```

```java
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
Depth-first search demo

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

To visit a vertex \( v \):

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

vertices reachable from 0

\[
\begin{array}{c|c|c}
  v & \text{marked[]} & \text{edgeTo}[v] \\
  \hline
  0 & T & - \\
  1 & T & 0 \\
  2 & T & 0 \\
  3 & T & 5 \\
  4 & T & 6 \\
  5 & T & 4 \\
  6 & T & 0 \\
  7 & F & - \\
  8 & F & - \\
  9 & F & - \\
  10 & F & - \\
  11 & F & - \\
  12 & F & - \\
\end{array}
\]
Depth-first search

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

    public DepthFirstSearch(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
Depth-first search properties

**Proposition.** After DFS, can find vertices connected to \( s \) in constant time and can find a path to \( s \) (if one exists) in time proportional to its length.

**Pf.** \texttt{edgeTo[]} is 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;
}
```

Trace of \texttt{pathTo()} computation:

<table>
<thead>
<tr>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1 2</td>
</tr>
<tr>
<td>2 0</td>
</tr>
<tr>
<td>3 2</td>
</tr>
<tr>
<td>4 3</td>
</tr>
<tr>
<td>5 3</td>
</tr>
</tbody>
</table>
4.1 Undirected Graphs

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

Repeat until queue is empty:

- Remove vertex \( v \) from queue.
- Add to queue all unmarked vertices adjacent to \( v \) and mark them.

graph \( G \)
Breadth-first search demo

Repeat until queue is empty:

- Remove vertex $v$ from queue.
- Add to queue all unmarked vertices adjacent to $v$ and mark them.

<table>
<thead>
<tr>
<th>$v$</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

done
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

```java
public class BreadthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;
    ...

    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 application: routing

Fewest number of hops in a communication network.

ARPANET, July 1977
Graph-processing challenge 2

**Problem.** Find a cycle.

**How difficult?**
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

![Diagram of a graph with vertices labeled 0 to 6 and edges connecting them. The cycle is highlighted with an arrow showing the path 0→5→4→6→0. There is a note indicating a simple DFS-based solution (see textbook).]
Graph-processing challenge 3

Problem. Find a (general) cycle that uses every edge exactly once.

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Eulerian tour
(classic graph-processing problem)
Graph-processing challenge 4

Problem. Find a cycle that visits every vertex exactly once.

How difficult?
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
✓ Intractable.
- No one knows.
- Impossible.

**Hamiltonian cycle**
(classical NP-complete problem)
Graph-processing challenge 5

**Problem.** Are two graphs identical except for vertex names?

**How difficult?**
- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

*graph isomorphism is longstanding open problem*
Graph-processing challenge 6

**Problem.** Lay out a graph in the plane without crossing edges?

**How difficult?**

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

linear-time DFS-based planarity algorithm discovered by Tarjan in 1970s (too complicated for most practitioners)