Example
Definition

A directed graph is a set of vertices (or “nodes”)

\[ V = \{ v_i : i \in 1, \ldots, n \} \]

and set of ordered pairs of these vertices called edges.

\[ E = \{ (v_i , v_j) : i, j \in 1, \ldots, n \} \]
Examples (Directed)

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>airports</td>
<td>flights</td>
</tr>
<tr>
<td>web pages</td>
<td>links (URLs)</td>
</tr>
<tr>
<td>Java objects</td>
<td>references</td>
</tr>
</tbody>
</table>
Definition

A *undirected graph* is a set of vertices

\[ V = \{v_i : i \in 1, \ldots, n \} \]

and set of unordered pairs, again called *edges*.

\[ E = \{ \{v_i, v_j\} : i, j \in 1, \ldots, n \} \]
Examples (Undirected)

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>people</td>
<td>friends</td>
</tr>
<tr>
<td>events</td>
<td>conflicts (edge if two events</td>
</tr>
<tr>
<td></td>
<td>cannot be at same time)</td>
</tr>
<tr>
<td>towns/cities</td>
<td>roads (two way)</td>
</tr>
</tbody>
</table>
We will mostly discuss directed graphs.
Terminology: “in degree”

<table>
<thead>
<tr>
<th>v</th>
<th>in degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>3</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>1</td>
</tr>
</tbody>
</table>
Terminology: “out degree”

```
<table>
<thead>
<tr>
<th>v</th>
<th>out degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
</tr>
</tbody>
</table>
```
In degree: How many web pages link to some web page (e.g. to f) ?

Out degree: How many web pages does some web page link to (e.g. from f) ?
Google “crawls” the web graph, retrieving and storing as many web pages as it can.

Google updates its web graph:
• the vertices $V$ are the web pages
• the edges $E$ are the hyperlink (references) within the web pages

ASIDE: Google also updates a hashmap:
• the keys $K$ are the URL’s, and the values are the web pages
ASIDE: Google PageRank

Google tries to find *important* web pages for your search term.

Q: How important is a web page $v$?

A:
- Which set of pages $\{w\}$ link to $v$ and how important is each page $w$ (recursive definition!)?
- How many other pages does each $w$ point to?
To define the “page rank” of $v$:

Let $w$ be a vertex such that $(w, v)$ is an edge.

Let $N_{out}(w)$ be the out-degree of $w$. 
To define the “page rank” of $v$:
Let $w$ be a vertex such that $(w, v)$ is an edge.

Let $N_{out}(w)$ be the out-degree of $w$.

Define the PageRank of $v$:

$$R(v) = \sum_{\text{incoming edges} (w,v) \text{ to } v} \frac{R(w)}{N_{out}(w)}$$

ASIDE: To calculate this, (1) we need a list of the incoming edges to each vertex, similar to an adjacency list but now we list the incoming instead of outgoing edges, and (2) we compute $R(v)$ for all $v$, and then plug the result back into the right side, and iterate. We initialize all $R(v)$ to 1.
Sergey Brin's Home Page

Ph.D. student in Computer Science at Stanford -
sergey@cs.stanford.edu

Research

Currently I am at Google.

In fall '98 I taught CS 349.

Data Mining

A major research interest is data mining and I run a meeting group here at Stanford. For more information take a look at the MIDAS home page or see the datamine mailing list archive. Here are some recent publications:
Terminology: path

A path is a sequence of edges such that the end vertex of one edge is the start vertex of the next edge. No vertex may be repeated except first and last.

Examples
- acfеб
- dac
- dcfеб
- .....
A cycle is a path such that the last vertex is the same as the first vertex.

Examples
- febf
- efe
- fbf
- ...
Directed Acyclic Graph
(directed graph with no cycles)

DAGs are used to capture dependencies. E.g.
• a *causes or implies* b
• a must happen *before* b can happen (temporally)
• .....
This will be a solid line next year.
Weighted Graph

![Weighted Graph Diagram]
ASIDE: Shortest path algorithms (COMP 251)

e.g. Given a graph, what is the shortest (weighted) path between two vertices?
ASIDE: “Travelling Salesman” (COMP 360)

Find the minimum weight cycle that visits all vertices once. (except first & last).

This is a hard problem (called “NP complete”).

With $n$ vertices and edges between each pair, time complexity is $O(n^2 2^n)$ which is very slow.
Graph ADT

• addVertex(), addEdge()
• removeVertex(), removeEdge()
• getVertex(), getEdge()

• containsVertex(), containsEdge()
• numVertices(), numEdges()
• ...


How to implement a Graph class?

• Graphs are a generalization of trees, but a graph does not have a root vertex.

• Outgoing edges from a vertex in a graph are like children of a vertex in a tree.

• Incoming edges are like parent(s).

There are two standard ways of representing edges (next few slides).
1. Adjacency List
   (generalization of children in trees)

Here each adjacency list is sorted, but that is not always possible/meaningful (or necessary).
How to implement a graph with adjacency lists in Java? (1)

class Graph<T> {
    :
    class Vertex<T> {                      // could have called it GNode
        ArrayList<Vertex> adjList;      // end vertex of edge (start is ‘this’ vertex)
        T element;
    }
    :
}

Q: What if you want the edges to have weights?
How to implement a graph with adjacency lists in Java? (2)

```java
class Graph<T> {
    // this would be a weighted graph

    class Vertex<T> {
        ArrayList<Edge> adjList; // end vertex of an edge (start is 'this' vertex)
        T element;
    }

    class Edge {
        Vertex endVertex;
        double weight;
    }
}

Q: How to access vertices?
```
Q: How to access vertices?

A: Use a HashMap. The key could be a string name for each vertex, e.g. “YUL” for Trudeau airport, “LAX” for Los Angeles, etc.

```java
class Graph<T> {
    HashMap<String, Vertex<T>> vertexMap;

    class Vertex<T> {
        ArrayList<Edge> adjList;
        T element;
    }

    class Edge {
        Vertex endVertex;
        double weight;
    }
}
```

HashMap’s have methods like getKeys(), getValues(), ...
2. Adjacency Matrix

Note:
- We need a hashmap from vertex names to indices 0, 1, ..., n-1.

```
boolean adjMatrix[6][6]
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
2. Adjacency Matrix

Note:
• Use the diagonal elements for loop edges.

boolean adjMatrix[6][6]

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```
2. Adjacency Matrix

Note:
- For a weighted graph, we could use weights in the matrix instead of booleans.
See Exercises for when you would use *adjacency list* versus *adjacency matrix*.

Hint: it depends on how many edges we have relative to number of vertices.
Coming up...

**Lectures**

Mon. March 28  
Graphs traversal

Wed & Fri, March 30 & April 1  
recurrences

Mon, Wed, Fri: April 4, 6, 8  
big O, ...

**Assessments**

Quiz 5 is in Mon. April 4

Assignment 4 due Wed. April 6.