Graphs

See Chapter 14 of the text.
So far this semester we have talked about many specific data structures -- lists, queues, stacks, binary trees, heaps, etc.

We will now look at graphs, which are much more general.

Many, many problems can be solved by creating a graph that represents the problem, processing the graph, and thereby creating a solution for the problem.
For example, suppose we are writing a scheduler for a collection of processes. Some processes get input from others. If process B gets input from A, we need to schedule A before we schedule B. Our task is to find an ordering of the processes so that we complete each before its output is needed by any other process.

Here is a way to solve this problem:

First, make a graph where each process is represented by a node of the graph. Add an edge in the graph from node X to node Y if X needs to be processed before Y.
Our graph might look like this:
To process this graph we will make a set of nodes we call our "WorkingSet". Initially our WorkingSet consists of all nodes that have no incoming edges.

For the graph above WorkingSet = \{B\}
Here is the algorithm we use to process the graph. On each step:

a) Remove any one node from the WorkingSet. Call this node X.

b) Remove every edge from node X to any other node Y.

c) If node Y has no other incoming edges, add node Y to the WorkingSet.

Continue these steps until the WorkingSet is empty.
We start with this graph:

On the first step we remove B from the WorkingSet, add it to the output, and we remove the edges from B to A, E, and C.
We removed the only incoming edge for node A, so we add it to the WorkingSet:

\[
\text{WorkingSet} = \{A\}
\]

\[
\text{Output} = [B]
\]

For the next step we remove A from the WorkingSet, add it to the output, and remove its outgoing edges.
C now has no incoming edges, so we add it to the WorkingSet:

WorkingSet = \{C\}

Output = [B, A ]

On the next step we remove C and its edges; this leaves E and F with no incoming edges.
On the next three steps the algorithm outputs E and F in either order, then D.
Our ordering is thus either [B, A, C, E, F, D] or [B, A, C, F, E, D]. If you compare these to the original graph you can see that no node appears in this list before any of its dependencies.
This idea of mapping a problem to a graph and processing the graph to solve the problem has many applications. To consider any of these we need some terminology and we need to look at some ways to represent graphs.
First, a *graph* is a set of nodes together with a set of edges. There are two big classes of graphs. A *directed graph* has directional edges; an edge goes from node X to node Y, and this is different from an edge that goes from node Y to node X. The example we started with is a directed graph:

![Directed Graph Diagram](image)

Directed graphs are sometimes called *digraphs* in honor of Diana, the late Princess of Wales.
In an *undirected graph* the edges are not directional; an edge from X to Y is the same as an edge from Y to X.

For both kinds of graphs an edge from X to Y is often written \((X, Y)\); you can think of \((X, Y)\) as being an ordered pair for a directed graph and being a set for an undirected graph.

In a directed graph, a *cycle* is a sequence of connected nodes that repeats itself: there might be an edge from A to B, from B to C, from C to D and then from D back to A.
Here is a graph with a cycle:

A graph with no cycles is said to be **acyclic**. The class of **directed acyclic graphs**, also known as DAGs, is very important because some algorithms only work on DAGs.
In some situations we attach numbers to the edges of a graph. These might represent costs, or weight, or distance according to the way we interpret the graph. For example, we might have a graph where the nodes are cities and each edge from one city to another has the cost of a plane ticket for traveling between those cities:

We might represent such a weighted edge as a triple:

(Denver, Chicago, 230)
In either a directed or an undirected graph a *path* is a sequence of nodes connected by edges. For example, in the graph

![Graph Diagram]

one path is A -> C -> E -> D
The length of a path is the number of edges it contains. If the edges are weighted, the weighted length of a path is the sum of the edge weights.

The set of edges of a graph is often represented by E, while the nodes (or vertices) is represented by V. If there is one edge from each node to each other node, the number of edges is $|E| = |V|^* (|V| - 1)$. If $|E| = \Theta(|V|^2)$ we say that the graph is dense. If $|E| = \Theta(|V|)$ the graph is sparse. These definitions are not universally used; some people use them informally in the sense that a graph is called dense if it has a relatively large number of edges and sparse if it has a relatively small number of edges.
Graph Representations

There are many ways to represent directed and undirected graphs. One simple scheme is to use an adjacency matrix. Let $n = |V|$ be the number of nodes of the graph. Number the nodes 0, 1, 2, ..., $n-1$. Create an $n\times n$ matrix where the $[i][j]$ entry is 1 if there is an edge from node $i$ to node $j$, and 0 if there is no such edge.
If we let A be node [0], B node [1] and so forth, the following graph

has adjacency matrix

<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>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
There are many variations on this idea. If there are edge weights, they can be stored in the adjacency matrix rather than markers 0 and 1. Missing edges might be represented by INFINITY.

Adjacency matrices are fine for small graphs but unless the graph is very dense the matrix representation is quite inefficient. There are a million entries in the adjacency matrix for a graph with one thousand nodes. Just initializing such a matrix takes a long time.
An alternative that is often more efficient is to store the graph in an *adjacency list*. We represent the matrix by an array of linked lists; each list represents node which the given node is adjacent to. This graph might be represented

```
A B C D E F
A C E
B A C E
C D E
D E F
E D
F D
```
In Section 14.1 Weiss has a Java implementation of a Graph class that can be used to represent graphs where each node has a String name (rather than a number or index). The primary data structure of this class is a HashMap<String, Vertex> called vertexMap. Vertex is a class that represents one node of the graph. The class variables for Vertex include

   String name;
   List<Edge> adjacent; //list of adjacent vertices

Here Edge is a nested class with class variables

   Vertex destination; // the vertex being connected to
double weight; // the weight or cost of the edge
When we come across a reference to a vertex name we can find its structure in the HashMap with

```java
Vertex getVertex(String name) {
    Vertex v = vertexMap.get(name);
    if (v == null) {
        v = new Vertex(name);
        vertexMap.put(name, v);
    }
    return v;
}
```
Adding an edge to the graph is easy:

```java
public void addEdge(String sourceName, String destName, double weight) {
    Vertex v = vertexMap.get(sourceName);
    Vertex w = vertexMap.get(destName);
    v.adjacent.add(new Edge(w, weight) );
}
```