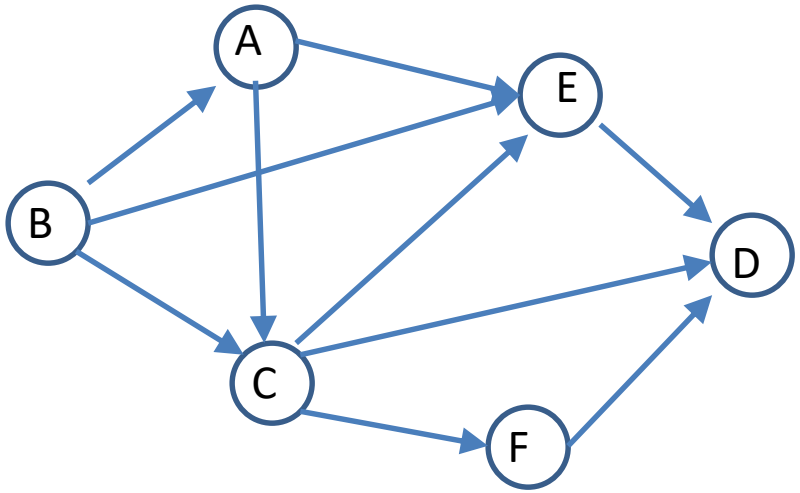
Graph Terminology and Implementation

See Chapter 14 of the text.

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:

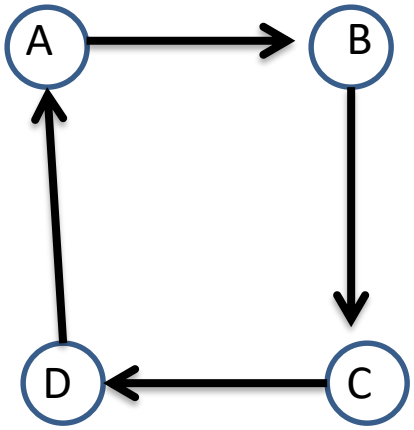


Directed graphs are sometimes called *digraphs* in honor of Diana, the late Princess of Wales.

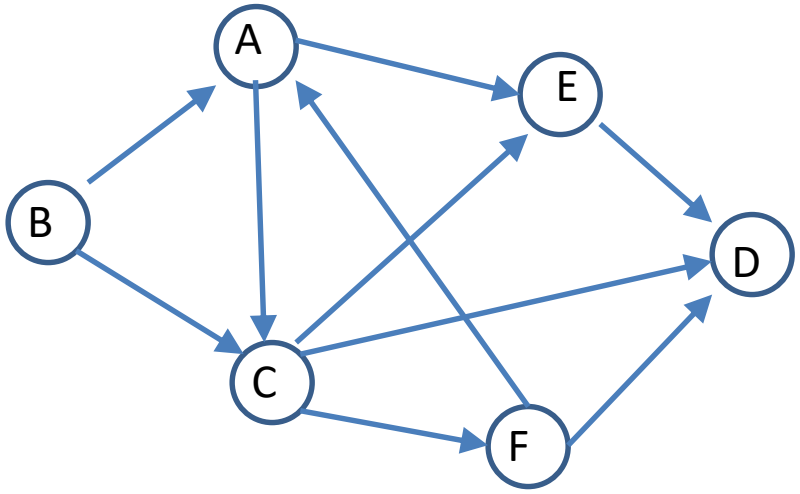
In an *undirected graph* the edges are not directional; an edge from X to Y is the same as an edge from Y to X.

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.



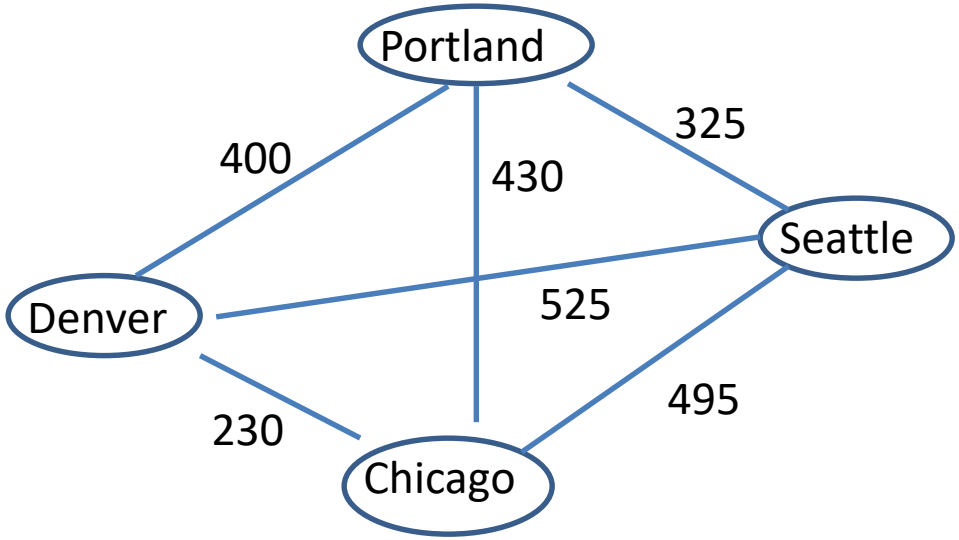
Here is a graph with a cycle:



The cycle is A -> C -> F -> A

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 (e.g., topological sorting) only work on DAGs.

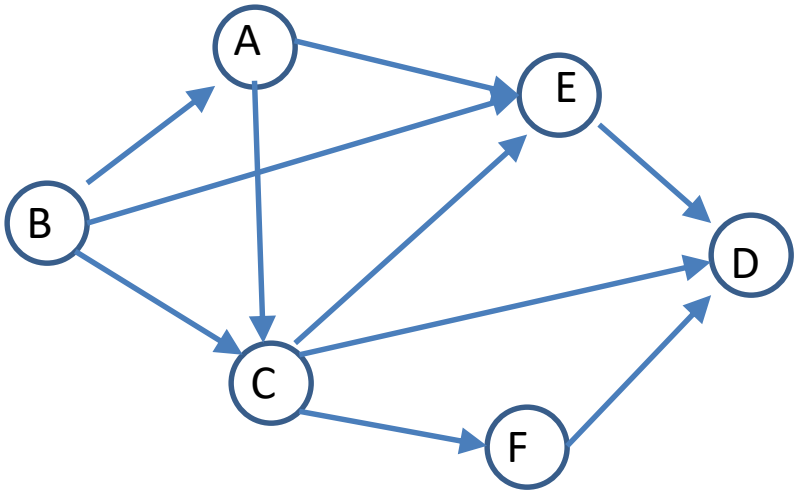
In some situations we attach numbers to the edges of a graph. These might represent costs, or weight, or distance according to the way we interpret the graph. For example, we might have a graph where the nodes are cities and each edge from one city to another has the cost of a plane ticket for traveling 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



one path from A to D is A -> C -> E -> D.
There is no path in this graph from C to A.

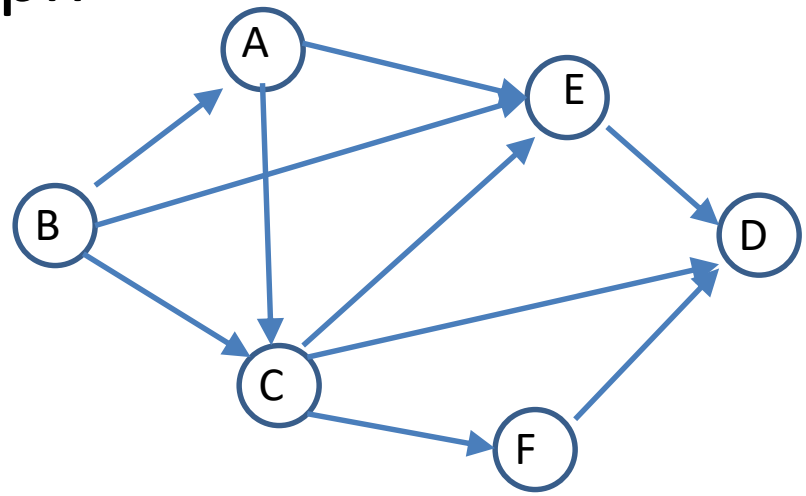
The *length* of a path is the number of edges it contains. If the edges are weighted, the *weighted length* of a path is the sum of the edge weights.

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 nxn 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

	A	B	C	D	E	F
A	0	0	1	0	1	0
B	1	0	1	0	1	0
C	0	0	0	1	1	1
D	0	0	0	0	0	0
E	0	0	0	1	0	0
F	0	0	0	1	0	0

Question: What is the shortest path from A to F?

source \ destination	A	B	C	D	E	F
A	0	1	0	1	1	0
B	0	0	1	1	1	0
C	0	0	0	0	0	1
D	0	0	1	0	0	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

A. A -> C -> D -> F
B. A -> B -> D -> C -> F
C. A -> B -> C -> F
D. None of these answers

The correct answer is D: neither A nor B nor C

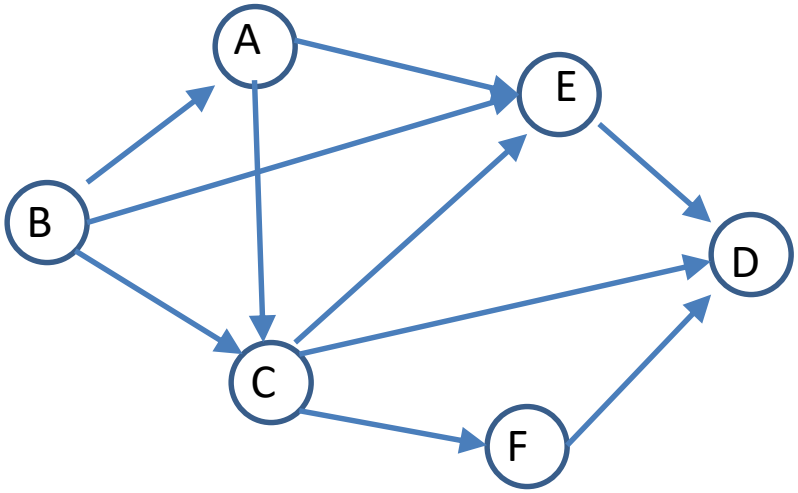
source \ destination	A	B	C	D	E	F
A	0	1	0	1	1	0
B	0	0	1	1	1	0
C	0	0	0	0	0	1
D	0	0	1	0	0	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

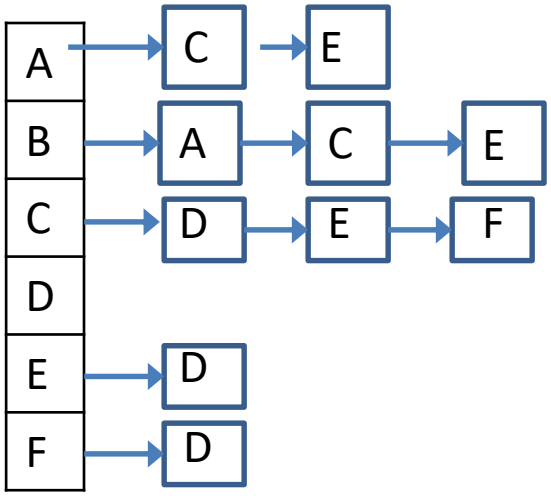
There is a path of length 2: A -> E-> F

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.

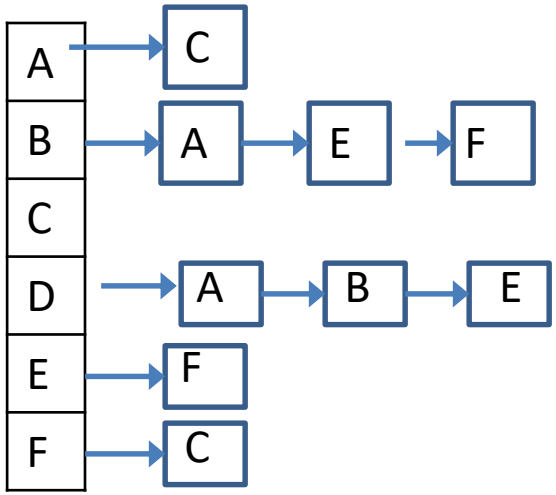
Adjacency matrices are fine for small graphs but unless the graph is very dense the matrix representation is quite inefficient. There are a million entries in the adjacency matrix for a graph with one thousand nodes. Just initializing such a matrix takes a long time.

An alternative that is often more efficient is to store the graph in an *adjacency list*. We represent the graph by an array of linked lists; each list represents nodes which the given node is adjacent to. This graph might be represented



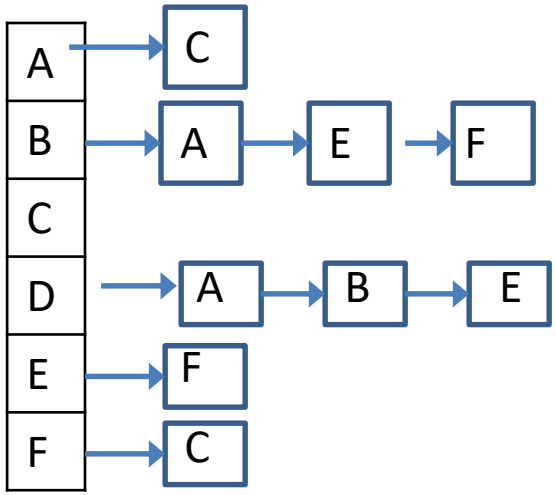


Question: In the following graph, what is a node with no incoming edges?



A. A
B. B
C. C
D. D

Answer: A node with no incoming edges is D



Here is a structure we will use for a number of graph algorithms, including Lab 9.

A. The graph is represented as a HashMap<String, Vertex>. If you give this map the name of a vertex it will give you back the structure for that vertex.

B. Vertex is a class that represents one node of the graph. The class variables for Vertex include

```
String name;
List<Edge> outgoing; //list of outgoing edges
```

(continued next slide)

C. Edge is a class with class variables

Vertex destination;
int weight; // the weight of the edge if there are weights

Let’s think about how well this new structure will implement the Topological Sorting algorithm. We need to start by finding the nodes that have no incoming edge. Give every Vertex a variable that holds its incoming edge count; initially all of those counts are 0. We run through all of the vertices (the keys of the Hashmap <String, Vertex> that represents the graph). Each vertex has an outgoing edge count; we go to the Vertex for the outgoing edge’s destination and increment its incoming edge count. After processing all of vertices we walk through the vertices again. Any edge with as its incoming edge count has no incoming edges.

After we remove a vertex from the WorkList we need to run through its outgoing edges. Our implementation makes that easy since every vertex has a list of its outgoing edges. The algorithm says to “delete” the vertex’s outgoing edges; we do this by going to the destination of the edge and decrementing its incoming edge count. If this count becomes 0 we can add the vertex to the WorkList.

So our new graph structure makes implementing the Topological Sorting algorithm easy.

We will read a graph in from a file as a list of weighted edges:

A B 2

means to create an edge from node A to node B with weight 2. This may be the first mention of either node A or node B.

When we come across a reference to a vertex name we can find or create its structure in the HashMap with

```
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 such as

A B 2

to the graph is easy:

```
public void addEdge(String sourceName, String destName, int weight) {
        Vertex source = getVertex(sourceName);
        Vertex dest = getVertex(destName);
        source.outgoing.add(new Edge(dest, weight) );
}
```