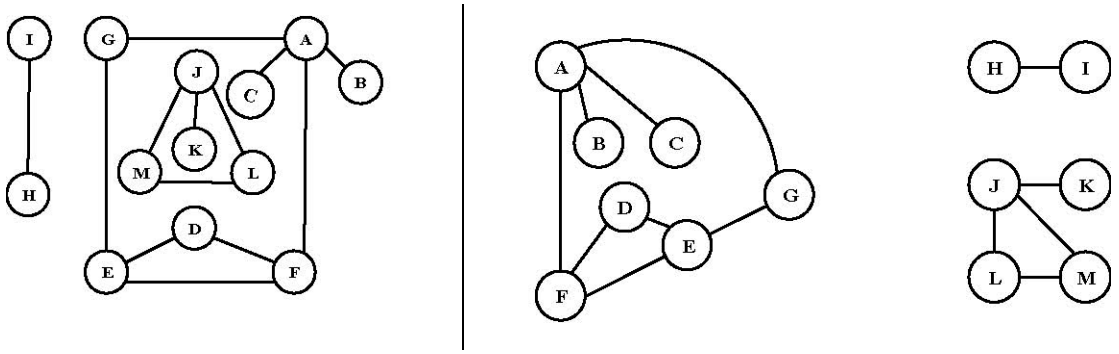
Graphs

Many problems are naturally formulated in terms of points and connections between them. For example, an electric circuit has gates connected by wires, an airline map has cities connected by routes, and a program flowchart has boxes connected by arrows. A graph is a mathematical object which models such situations.

A *graph* is a collection of vertices and edges. An *edge* is a connection between two *vertices* (or *nodes*). One can draw a graph by marking points for the vertices and drawing lines connecting them for the edges, but it must be borne in mind that the graph is defined independently of the representation. For example, the following two drawings represent the same graph:



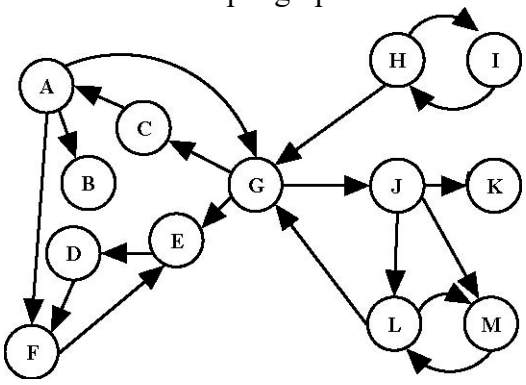
The precise way to represent this graph is to say that it consists of the set of vertices {*A, B, C, D, E, F, G, H, I, J, K, L, M*}, and the set of edges between these vertices {*AG, AB, AC, LM, JM, JL, JK, ED, FD, HI, FE, AF, GE*}.

A *path* from vertex *x* to *y* in a graph is a list of vertices, in which successive vertices are connected by edges in the graph. For example, *BAFEG* is path from *B* to *G* in the graph above. A *simple path* is a path with no vertex repeated. For example, *BAFEGAC* is not a simple path.

A graph is *connected* if there is a path from every vertex to every other vertex in the graph. Intuitively, if the vertices were physical objects and the edges were strings connecting them, a connected graph would stay in one piece if picked up by any vertex. A graph which is not connected is made up of *connected components*. For example, the graph above has three connected components: {*I, H*}, {*J, K, L, M*} and {*A, B, C, D, E, F, G*}.

A *cycle* is a path, which is simple except that the first and last vertex are the same (a path from a point back to itself). For example, the path *AFEGA* is a cycle in our example. Vertices must be listed in the order that they are traveled to make the path; any of the vertices may be listed first. Thus, *FEGAF* and *GAFEG* are different ways to identify the same cycle. For clarity, we list the start / end vertex twice: once at the start of the cycle and once at the end. A graph with no cycles is called a *tree*. There is only one path between any two nodes in a tree. A tree on N vertices contains exactly N-1 edges. A *spanning tree* of a graph is a subgraph that contains all the vertices and forms a tree. A group of disconnected trees is called a *forest*.

Directed graphs are graphs which have a direction associated with each edge. An edge *xy* in a directed graph can be used in a path that goes from *x* to *y* but not necessarily from *y* to *x*. For example, a directed graph similar to our example graph is drawn below.



There is only one directed path from *D* to *F*. Note that there are two edges between *H* and *I*, one each direction, which essentially makes an undirected edge. An undirected graph can be thought of as a directed graph with all edges occurring in pairs in this way. A *dag* (*directed acyclic graph*) is a directed graph with no cycles.

We'll denote the number of vertices in a given graph by *V*, the number of edges by *E*. Note that *E* can range anywhere from *V* to V^2 (or 1/2 *V*(*V*-1) in an undirected graph). Graphs will all edges present are called *complete* graphs; graphs with relatively few edges present (say less than *V* log(*V*)) are called *sparse*; graphs with relatively few edges missing are called *dense*.

It is frequently convenient to represent a graph by a matrix, as shown in the second sample problem below. If we consider vertex *A* as 1, *B* as 2, etc., then a "one" in *M* at row *i* and column *j* indicates that there is a path from vertex *i* to *j*. If we raise *M* to the *p*th power, the resulting matrix indicates which paths of length *p* exist in the graph. In fact, the quantity $M^p(i,j)$ is the number of paths.

The graph data structure has many applications in computer science. We use graphs to represent networks, solve a variety of manufacturing and transport problems, play games, describe 3-D virtual worlds, simulate the dynamic behavior of architectural structures, and so on.

The structure of a graph in a computer program is defined by the physical problem being solved. For example, nodes in a graph may represent cities, while edges may represent airline flight routes between the cities. The shape of the graph arises from the specific real world situation.

A graph structure can be extended by assigning a weight to each edge of the graph. Graphs with weights can be used to represent many different concepts; for example if the graph represents a road network, the weights could represent the length of each road. A directed graph with weighted edges is called a network.

Drawing Graphs

A graph drawing should not be confused with the graph itself (the abstract, non-graphical structure) as there are several ways to structure the graph drawing. All that matters is which vertices are connected to which others by how many edges and not the exact layout. In practice it is often difficult to decide if two drawings represent the same graph. Depending on the problem domain some layouts may be better suited and easier to understand than others.

Data Structures

There are different ways to store graphs in a computer system. The data structure used depends on both the graph structure and the algorithm used for manipulating the graph. Theoretically one can distinguish between list and matrix structures but in concrete applications the best structure is often a combination of both. List structures are often preferred for sparse graphs as they have smaller memory requirements. Matrix structures on the other hand provide faster access but can consume huge amounts of memory if the graph is very large.

Adjacency matrix

An adjacency matrix is an N by N matrix, where N is the number of vertices in the graph. If there is an edge from some vertex x to some vertex y, then the element M(x,y) is 1, otherwise it is 0. If the graph is undirected, the adjacency matrix is *symmetric*.

Matrix Multiplication

Matrix multiplication is a binary operation that takes a pair of matrices and produces another matrix. The elements of the rows in the first matrix are multiplied with corresponding columns in the second matrix.

$$\begin{Bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{Bmatrix} \begin{Bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{Bmatrix} = \begin{Bmatrix} 2 & & \\ & & \\ & & \end{Bmatrix}$$

$$0 \times 0 + 1 * 1 + 1 * 1 = 2$$

Lab 08A - 60 points – Airline Route Simulator

For this exercise we will use a graph with ten nodes. Each node represents a city and the edges connecting the nodes indicate whether there is a connecting flight from any two cities. The graph will be implemented using a matrix of **boolean** values. **true** means a connection exists, **false** means there is no connection.

We will need a class to hold the edges for the AirlineRoute simulator. We could create our own ADT called Edge that holds an X and a Y position (since that is all an edge is defined as) or we can simply use one of the build in Java classes – Point. The point class has two public variables int x and int y. Since they are public they can be accessed directly. The Point class has on associated methods. We will use the Point class to represent the edges in our simulator.

Now that we have a class to hold the edges, write a class that implements the **AirlineGraph** interface. The AirlineGraph interface is defined as:

```
interface AirlineGraph
{
    static final int SIZE = 10;

    static final String airportCode[] = { "BOS", "CHI", "DFW", "DEN",  "HNL",
                                          "IAH", "MIA", "JFK", "PHX", "SFO" };

    static final String city[] = {  "Boston, MA", "Chicago, IL", "Dallas-Ft Worth, TX",
                                    "Denver, CO", "Honolulu, HI", "Houston, TX",
                                    "Miami, FL", "New York, NY", "Phoenix, AX",
                                    "San Francisco, CA" };

    abstract String findRoute(int length, String start, String end);
}
```

In addition to the **public String findRoute(int length, String start, String end)** method required by the **AirlineGraph interface** you will need a **public String toString()** method. The **toString()** method should return a String representation of the matrix. If a Graph object were printed it would look like this:

	BOS	CHI	DFW	DEN	HNL	IAH	MIA	JFK	PHX	SFO
BOS	-	-	-	-	-	-	-	*	-	-
CHI	-	-	-	*	-	*	-	*	-	-
DFW	-	-	-	-	*	*	-	-	-	*
DEN	-	*	-	-	-	-	-	-	-	*
HNL	-	-	*	-	-	*	-	-	-	*
IAH	-	*	*	-	*	-	*	-	*	-
MIA	-	-	-	-	-	*	-	*	-	-
JFK	*	*	-	-	-	-	*	-	-	-
PHX	-	-	-	-	-	*	-	-	-	-
SFO	-	-	*	*	*	-	-	-	-	-

The Graph class will need to implement a two-dimensional matrix and a Stack.

Field Summary

private static boolean[][]	**graph** A two dimensional *adjacency matrix*.
private static Stack<Integer>	**stack** A last in first out data structure.

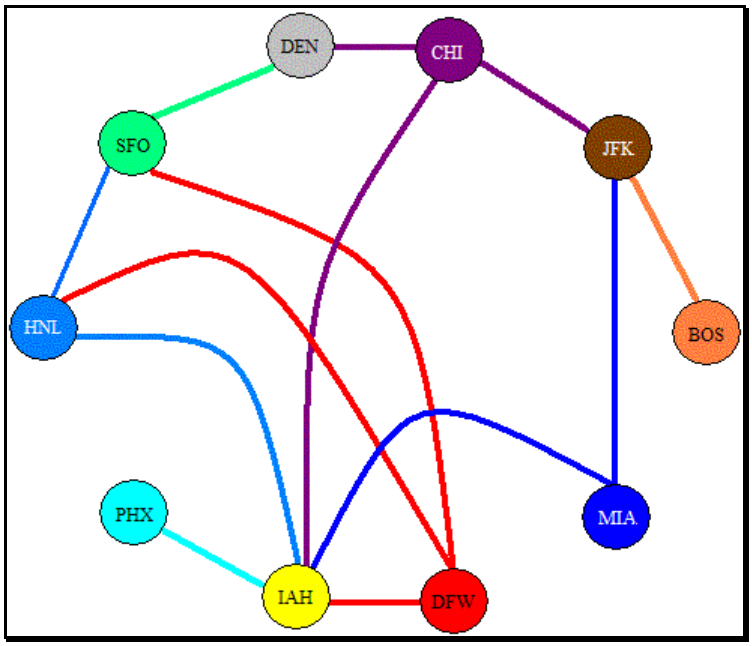
Constructor Summary

Graph()
Initializes a newly created Graph object. Initializes the *adjacency matrix* to be a **10 X 10** grid and initializes the grid by setting the points contained in the data file "connections.dat" to be **true**.

Method Summary

public String	**findRoute(int length, String start, String end)** Returns a String containing all the cities visited from <*start*> to <*end*> including <*start*> and <*end*> in the order visited.
public String	**toString()** Returns a String representation of this Graph object.
private int	**findAirportCode(String airportCode)** Returns the index position of the specified airportCode.
private boolean	**adjacent(Point p)** Returns true if Point p is connected, false otherwise.

private boolean	**findPath(int length, Point p)** Returns true if a path of length <length> exists between the two points in Point <p>. If a path exists, the index values for each airport visited is pushed onto the stack. **Algorithm:** **if length equals 1** **if adjacent(edge)** **push the current city onto the stack** **return true** **else** **for every *node* in the graph** **if adjacent(Point(p.y, i) && findPath(length – 1, Point(*node*, p.x))** **push the current city onto the stack** **return true** **return false**



Airline Graph

	BOS	CHI	DFW	DEN	HNL	IAH	MIA	JFK	PHX	SFO
BOS								*		
CHI				*		*		*		
DFW					*	*				*
DEN		*								*
HNL			*			*				*
IAH		*	*		*		*		*	
MIA						*		*		
JFK	*	*					*			
PHX						*				
SFO			*	*	*					

A Matrix representation of the Airline Graph

Run **AirlineRoutes** to test your class.

OUTPUT

	BOS	CHI	DFW	DEN	HNL	IAH	MIA	JFK	PHX	SFO
BOS	-	-	-	-	-	-	-	*	-	-
CHI	-	-	-	*	-	*	-	*	-	-
DFW	-	-	-	-	*	*	-	-	-	*
DEN	-	*	-	-	-	-	-	-	-	*
HNL	-	-	*	-	-	*	-	-	-	*
IAH	-	*	*	-	*	-	*	-	*	-
MIA	-	-	-	-	-	*	-	*	-	-
JFK	*	*	-	-	-	-	*	-	-	-
PHX	-	-	-	-	-	*	-	-	-	-
SFO	-	-	*	*	*	-	-	-	-	-

There is no direct connection!
Phoenix, AX -> Houston, TX -> Miami, FL
San Francisco, CA -> Denver, CO -> Chicago, IL -> New York, NY
Honolulu, HI -> Houston, TX -> Chicago, IL -> New York, NY -> Boston, MA