Six Degrees of Kevin Bacon

This programming assignment focuses on implementation and usage of a graph data structure.

Background:

Kevin Bacon, a well-known actor, inspired a college movie game called Six Degrees of Kevin Bacon, which is centered on finding the Bacon number of an arbitrary actor or actress. The Bacon number of an actor or actress is determined by the following rules:

- Kevin Bacon himself has a Bacon number of zero.
- The Bacon number of any other actor is defined to be the minimum of the Bacon numbers of all others with whom the actor appeared in a movie produced by a major studio, plus one.

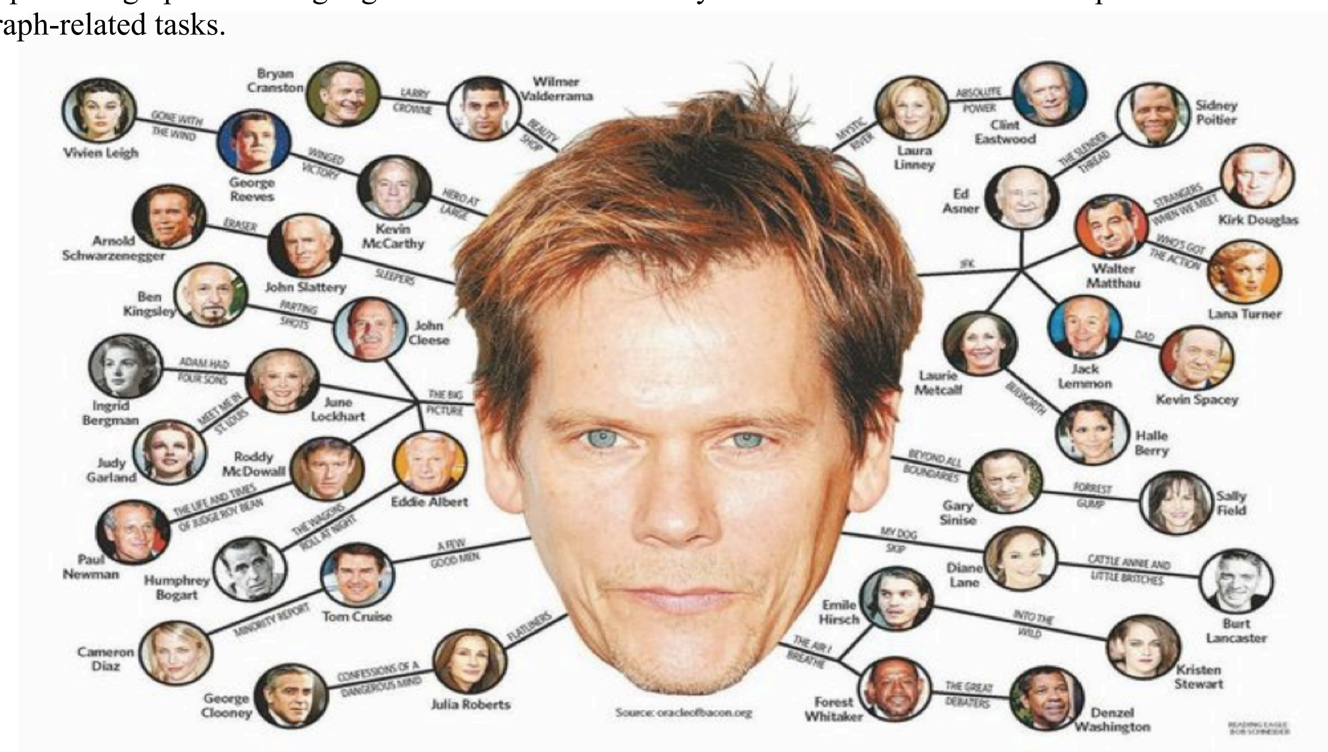
Almost every actor in Hollywood can be successfully linked to Kevin Bacon in 6 steps or fewer, hence the Six Degrees. In fact, the majority of actors have a Bacon number of 2 or 3. The higher the Bacon number of an actor, the less connected they are to other actors.

Notably, Bacon is not the most linkable actor. That honor currently goes to Dennis Hopper. The average Hopper number in the acting community is 2.743. By contrast, the average Bacon number is 2.951.

More information about the Six Degrees of Kevin Bacon is available on Wikipedia at http://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon. You can play an interactive web version of the game at http://oracleofbacon.org/.

Finding an actor's Bacon number and path to Kevin Bacon are tasks that can be solved by a computer. The data is a graph of actors, with edges connecting pairs of actors who appear in movies together. Common graph path searching algorithms such as breadth first search can discover an actor's Bacon number and path.

In this assignment, you will write the Kevin Bacon game using a provided graph object, and then you will implement graph searching algorithms that will enable you to solve the Kevin Bacon problem as well as other graph-related tasks.



Step 0 – Graph Implementation:

In this step of the assignment, you will complete a graph implementation. For this step, you are given supporting files `IGraph.java`, `AbstractGraph.java`, `VertexInfo.java`, and `EdgeInfo.java`. You will write a class named `Graph` (in file `Graph.java`) that extends the instructor-provided `AbstractGraph` class. `AbstractGraph` partially implements the `IGraph` interface. Your goal will be to add methods to the graph to complete the implementation of the `IGraph` interface found below. Documentation about the details of the behavior of each method you are to implement can be found in the "Methods to Implement" section below.

```
public interface IGraph<V, E> {
    // vertex-related methods
    public void addVertex(V v);
    public boolean containsVertex(V v);
    public Collection<V> neighbors(V v);
    public Collection<V> vertices();

    // edge-related methods
    public void addEdge(V v1, V v2, E e);
    public void addEdge(V v1, V v2, E e, int weight);
    public boolean containsEdge(V v1, V v2);
    public E edge(V v1, V v2);
    public Collection<E> edges();
    public int edgeWeight(V v1, V v2);
}
```

The graph representation that we will be using for our implementation is the "adjacency map". The adjacency map is a double mapping that connects pairs of vertices to their associated edges. This is represented by the data structure `adjacencyMap` of type `Map<V, Map<V, EdgeInfo<E>>>` in the `AbstractGraph` class. The benefit of this representation is that your graph will have constant (O(1)) expected runtime for common operations such as adding/retrieving vertices and edges, or getting collections of vertices and neighbors.

Two additional data structures can be found in the `AbstractGraph` class: `vertexInfo` (of type `Map<V, VertexInfo<V>>`) and `edgeList` (of type `List<E>`). `vertexInfo` contains a mapping from vertices to `VertexInfo` objects. The `VertexInfo` object keeps additional information about vertices that are helpful for different graph algorithms. `edgeList` is a collection of all edges in the graph. All of this data could be kept in the `adjacencyMap` data structure, but these additional data structures allow for code clarity, ease of use, and efficient support for common operations performed on the graph.

In addition to these data structures, the `AbstractGraph` class provides a no argument constructor that constructs an empty undirected graph by initializing the declared data structures, methods that can be used in your `Graph` class to check that the graph is in a valid state and parameters passed to methods are valid (i.e. `checkForNull`, `checkVertex`, `checkVertices`), and implementation of the following methods:

`public Collection<E> edges()`	returns a read-only collection of the graph's edges
`public String toString()`	returns a `String` representation of the graph
`public Collection<V> vertices()`	returns a read-only collection of the graph's vertices
`protected void clearVertexInfo()`	resets all distance/previous/visited data from all the `VertexInfo` objects in this graph (useful for Step 1)

Methods to Implement:

The methods you must implement to complete the `IGraph` interface are listed below in detail.

- `public void addVertex(V v)`
  In this method, you should add a vertex of generic type `V` to the graph. If there is already a vertex in your graph with this information, no change should be made to the graph. If the vertex passed is `null`, you should throw a `NullPointerException`.

- `public boolean containsVertex(V v)`
  You should implement this method to return `true` if there exists a vertex in your graph with the given information `v`; otherwise, return `false`.

- `public Collection<V> neighbors(V v)`
  Implement this method to return a collection containing all vertices that are connected to the given vertex `v` by an edge. If the vertex passed is `null`, you should throw a `NullPointerException`. If the vertex passed is not a part of the graph, you should throw an `IllegalArgumentException`.

- `public void addEdge(V v1, V v2, E e)`
  In this method, you should add an undirected edge to the graph between the two vertices `v1` and `v2`. `e` is of generic type `E` and represents the information to store in the edge. The edge should by given a default weight of 1. If an edge already exists between the vertices, it should be replaced with the given information. If any of the arguments are `null`, you should throw a `NullPointerException`. If either of the vertices passed is not a part of the graph, you should throw an `IllegalArgumentException`.

- `public void addEdge(V v1, V v2, E e, int weight)`
  Implement this method to add an undirected edge to the graph between the two vertices `v1` and `v2`. `e` is of generic type `E` and represents the information to store in the edge. The edge should have the given `weight`. If an edge already exists between the vertices, it should be replaced with the given information. If any of the arguments are `null`, you should throw a `NullPointerException`. If either of the vertices passed is not a part of the graph or if the edge weight is negative, you should throw an `IllegalArgumentException`.

- `public boolean containsEdge(V v1, V v2)`
  Implement this method to return `true` if there exists an edge between the two vertices `v1` and `v2`; return `false` otherwise.

- `public E edge(V v1, V v2);`
  Implement this method to return the edge that connects `v1` to `v2`. If `v1` and `v2` are legal vertices but there is no edge between them, you should return `null`. If either of the vertices passed is `null`, you should throw a `NullPointerException`. If either of the vertices passed is not a part of the graph, you should throw an `IllegalArgumentException`.

- `public int edgeWeight(V v1, V v2);`
  This method should return the weight of the edge that connects `v1` and `v2`. If `v1` and `v2` are legal vertices but there is no edge between them, you should return `-1`. If either of the vertices passed is `null`, you should throw a `NullPointerException`. If either of the vertices passed is not a part of the graph, you should throw an `IllegalArgumentException`.

Step 1 - Graph Search Implementation:

For this part of the assignment, you will write a class named `SearchableGraph` (in file `SearchableGraph.java`) that extends your `Graph` class from Step 0 and implements the `ISearchableGraph` interface. Your goal is to add path searching methods to the graph.

Methods to Implement:

The methods you must implement to complete the `ISearchableGraph` interface are listed below in detail. Each of these methods should not modify the state of the map's vertices or edges.

- `public boolean reachable(V v1, V v2)`
  Returns whether there is any path in this graph that leads from the given starting vertex `v1` to the given ending vertex `v2`. Any vertex can reach itself. This method should be O(V + E). If either of the vertices passed is `null`, you should throw a `NullPointerException`. If either of the vertices passed is not a part of the graph, you should throw an `IllegalArgumentException`.
- `public List<V> shortestPath(V v1, V v2)`
  Returns the path in this graph, with the least number of vertices, that leads from the given starting vertex `v1` to the given ending vertex `v2`. Use the breadth-first algorithm to find the path. The shortest path from a vertex `v1` to itself should be a one-element list containing only `v1`. This method should be O(V + E). If `v2` is not reachable from `v1`, the method returns `null`. If either of the vertices passed is `null`, you should throw a `NullPointerException`. If either of the vertices passed is not a part of the graph, you should throw an `IllegalArgumentException`.

Step 2 – Kevin Bacon Game:

In this step of the assignment, you will use your graph and search algorithm implementation to solve the Kevin Bacon problem. For this step, you will be given the supporting files `KevinBacon.java` and `movies.txt`. You will add your code to and turn in `KevinBacon.java`.

The given `KevinBacon.java` file builds a `SearchableGraph` from `movies.txt`, a file of actors and movies. You should add to this file by printing an introductory message to the user, and then prompt them for an actor's name. You should then search the graph for the shortest path between the actor and Kevin Bacon, and print the information about the path returned. Here's an example log of execution; your output should match <u>exactly</u>:

```
Welcome to the Six Degrees of Kevin Bacon.
If you tell me an actor's name, I'll connect them to Kevin Bacon through
the movies they've appeared in.  I bet your actor has a Kevin Bacon number
of less than six!

Actor's name (or ALL for everyone)? Brad Pitt

Path from Brad Pitt to Kevin Bacon:
Brad Pitt was in Ocean's Eleven (2001) with Julia Roberts
Julia Roberts was in Flatliners (1990) with Kevin Bacon
Brad Pitt's Bacon number is 2
```

When the user types "ALL", your program should print the paths between every actor and Kevin Bacon.

"There are two types of actors: those who say they want to be famous and those who are liars." -- *Kevin Bacon*