CSE 373, Spring 2012
Homework #7: Six Degrees of Kevin Bacon (75 points)
Step 0: Due Wednesday, May 23, 2012, 2:30 PM
Steps 1 and 2: Due Wednesday, May 30, 2012, 2:30 PM

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 2.802. By contrast, the average Bacon number is 2.981.

Read more at http://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon. You can also 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 homework assignment, you will implement a graph representation 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. 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 you will be using for your 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. Each key in the `adjacencyMap` data structure is a vertex v_0 in the graph. The value that v_0 is mapped to in `adjacencyMap` is a second `Map` that has as its keys all vertices that v_0 is connected to in the graph. In this second `Map`, each vertex v_m (that v_0 is connected to) maps to information about the edge that is connecting v_0 to v_m. So, to find out if there is an edge between v_1 and v_2 we could call `adjacencyMap.get(v1).containsKey(v2)` and to get the information about the edge between v_1 and v_2 we could call `adjacencyMap.get(v1).get(v2)`. 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 (`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)`
  Adds a vertex of generic type `V` to the graph. If there is already a vertex in the 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)`
  Returns `true` if there exists a vertex in the graph with the given vertex `v`; otherwise, return `false`.
- `public Collection<V> neighbors(V v)`
  Returns 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)`
  Adds 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 is not a part of the graph, you should throw an `IllegalArgumentException`.
- `public void addEdge(V v1, V v2, E e, int weight)`
  Adds 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 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)`
  Returns `true` if there exists an edge between the two vertices `v1` and `v2`; return `false` otherwise.
- `public E edge(V v1, V v2);`
  Returns 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);`
  Returns the weight of the edge that connects `v1` and `v2`. If either of the vertices is `null`, you should throw a `NullPointerException`. If `v1` and `v2` are legal vertices but there is no edge between them or if either of the vertices is not a part of the graph, you should throw an `IllegalArgumentException`.

Step 1 - Graph Search Implementation:

For this step, 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 List<V> minimumWeightPath(V v1, V v2)`
  Returns the path in this graph, with the lowest total path weight, that leads from the given starting vertex `v1` to the given ending vertex `v2`. Use Dijkstra's algorithm to find the path. The minimum weight path from a vertex `v1` to itself should be a one-element list containing only `v1`. This method should be $O(|V|^2)$. 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 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 is not a part of the graph, you should throw an `IllegalArgumentException`.
- `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`.

Graph Searching Algorithms:

Use the following pseudo-code for breadth-first search and Dijkstra's algorithm to help you implement the graph search algorithms.

```
BFS(v1, v2):
  for each vertex v:
    v's visited := false.
    v's previous := none.

  List := {v1}.
  mark v1 as visited.
  while List is not empty:
    v := List.removeFirst().
    if v is v2:
      path is found.
      reconstruct path from v2 back to v1,
        following previous pointers.
    else, for each unvisited neighbor n of v:
      mark n as visited.
      n's previous := v.
      List.addLast(n).
  path is not found.
```

```
Dijkstra(v1, v2):
  for each vertex v:
    v's distance := infinity.
    v's previous := none.

  v1's distance := 0.

  List := {all vertices}.
  while List is not empty:
    v := remove List vertex with minimum distance.
    mark v as visited.
    for each unvisited neighbor n of v:
      dist := v's distance + edge (v, n)'s weight.
      if dist is smaller than n's distance:
        n's distance := dist.
        n's previous := v.

  if path is found
    reconstruct path from v2 back to v1, following
      previous pointers.
```

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`. If you are using Eclipse, `movies.txt` should be saved in your main project directory.

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 exactly:

```
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. If the user types the name of an actor that is not found in the graph, you program should print "No such actor."

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

Hints and Suggestions:

For Step 0, before beginning to write any code make sure you are well acquainted with all of the provided files. Most of the methods can be easily implemented by using methods that are available to you using the Java API. Our solution for `Graph.java` is 139 lines long (44 lines if you ignore blank, closing braces, and commented lines).

For Step 1, implement your shortest path search first, using a breadth-first search. Write the minimum weight path search second using Dijkstra's algorithm. The `reachable` method can be based upon the other two. These search algorithms may require you to store information about each vertex, such as whether it is visited or its best weight path seen so far. Consider storing this information in the `VertexInfo` objects associated with each vertex. Don't forget to clear out this information between multiple path searches. Our solution for `SearchableGraph.java` is 135 lines long (70 lines if you ignore blank, closing braces, and commented lines).

For Step 2, even though the Kevin Bacon Game is a fun application in which to use graphs, the `KevinBacon.java` file will not thoroughly be testing your graph and graph search implementations so you are encouraged to write additional test code. For each of your search algorithms, you may want to test them with edge cases such as the result of finding a path from a node to itself, asking for a path when none exists, and testing with graphs that contain disconnected vertices (vertices with no edges). Our solution for `KevinBacon.java` is 90 lines long (58 lines if you ignore blank, closing braces, and commented lines).

Submission and Grading:

Submit this assignment online, as with all programming assignments, via the link on the course web site. Since this assignment is due in 2 parts, each part has a separate turnin area on the web site. Turn in your `Graph.java`, `SearchableGraph.java`, and `KevinBacon.java` only. Your `Graph` should extend the `AbstractGraph` unmodified and your `SearchableGraph` should extend the `Graph` class and implement the `ISearchableGraph` interface.

The external correctness of your program will be graded on matching the expected behavior and output of the methods/classes to be implemented. Expected output is provided for the Kevin Bacon game; you should match this output exactly. You may want to use a diff tool (http://diffchecker.com/) to ensure your output for `KevinBacon.java` matches the given expected output file.

In addition to external correctness, you will be graded on whether you follow the program specification above, whether you implement the searching algorithms as specified, whether you have obeyed the Big-Oh runtime requested for each method.

You should follow good general style guidelines such as: making fields `private` and avoiding unnecessary fields; appropriately using control structures like loops and `if/else`; properly using indentation, good variable names and types. Redundancy is another major grading focus; some methods are similar in behavior or based off of each other's behavior. You should avoid repeated logic as much as possible. Your class may have other methods besides those specified, but any other methods you add should be `private`.

Comment your code descriptively in your own words at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, and exceptions. The files provided to you use the "doc comments" format used by Javadoc, but you do not have to do this.