Graph Search in Java

Synopsis: Your task is to implement a graph class in Java, patterned after the Java Collections API; instances are collections of nodes, edges define relations for the purpose of traversal. The class will provide iterators for depth-first, breadth-first, and best-first search. To demonstrate, build a program to solve a problem (or a few) that can be cast as graph search, such as those presented in class or in the text. A few suggestions are given below. **NOTE: there is additional material at the end of this documenet for those of you that choose to do this project in C++**.

Details: It is not required that your code adhere strictly to the following guidelines, especially if you encounter an error or flaw in the design presented. These are merely suggestions that illustrate and use modularity features provided by Java. Plus, they make grading easier.

- The graph class should be a collection, i.e., it should (at least partially) implement Java's `Collection` interface. It should completely implement the `GraphADT` interface, defined below.
- `Collection` defines 15 methods: *you do not have to implement all of them*. In particular, the methods for object removal/retention and conversion to an array are beyond the objectives of this project. The `iterator()` method needs not be implemented.
- Any method not implemented should throw `UnsupportedOperationException`.
- Your graph class can inherit from whatever you want; you may find it useful to use the `AbstractCollection` class.
- You may use any graph representation (adjacency list/matrix, node and edge sets, . . .), as long as your class provides the desired behavior. Your representation should not assume any problem-specific restrictions.
- Objects (nodes) contained in graph instances can be of any type, but should implement the `Node` interface (see below) so that they can be evaluated for heuristic merit during a best-first traversal.
- All nodes should contain a fixed heuristic value, which can be set during construction or by a mutator method, but should never be changed by the graph or its iterators. Nodes should also act as a container for the *evaluated* value, which should be set and modified during traversal.
- For traversing a graph, write three simple classes that implement the `NodeBag` interface (below): One stack, one queue, and one priority queue. The priority queue should sort based on a node's evaluated value. Name these classes whatever you want, they may inherit from anything, and the internal representation is up to you. Note that Java provides the `Stack` class for you.

- Your graph class will have at least three methods that return iterators, as required by the `GraphADT` interface. These methods will take one parameter: the object before which traversal begins. These methods should verify that the graph contains this object, or otherwise throw an exception that you define (it might make sense to extend `RuntimeException` here).

- You will need to write only one class for your iterators. This class should implement the `GraphIterator` interface, which, in turn, extends Java's `Iterator` interface. `remove()` is optional. The constructor should take:

  – a starting node;

  – a `Class` object, representing a class that implements the `NodeBag` interface; and,

  – anything necessary for the iterator to traverse the graph (e.g., an adjacency matrix).

  Iterators behave like they are "between" nodes. Once an iterator has been instantiated, the first call to `next()` should return the start node. The `getPath()` method should return an array of nodes, from start node to the last node returned by `next()`.

Interfaces. Below are the interface definitions that your classes should implement:

- Java's `Collection` interface, with commentary on method usage in this context:

```
public interface Collection {
  boolean add(Object obj); // Add node; true if node is new to graph
  boolean contains(Object obj); // Check for a node
  boolean isEmpty(); // True if graph has no nodes
  int size(); // Number of nodes
  int hashCode(); // hash for this instance
  // Optional methods
  boolean addAll(Collection coll); // Add collection of nodes
  boolean containsAll(Collection coll); // Check for given nodes
  void clear(); // Remove all nodes
  boolean remove(Object obj); // Remove a node; true if node existed
  boolean removeAll(Collection coll); // Remove given nodes
  boolean retainAll(Collection coll); // Remove all but given nodes
  boolean equals(Object obj); // Are two graphs equal?
  // Optional methods with undefined semantics
  Object[] toArray();
  Object[] toArray(Object[]);
  Iterator iterator();
}
```

- Our `Node` interface:

```
public interface Node {
  float getHeuristic(); // Return the (fixed) HEURISTIC value of this node
  void setValue(float val); // Set the EVALUATED value of this node
  float getValue(); // Return the EVALUATED value of this node
}
```

- Java's `Iterator` and our `GraphIterator` interfaces:

```
public interface Iterator {
  boolean hasNext();
  Object  next();
  void    remove(); // optional
}
public interface GraphIterator extends Iterator {
  Node[] getPath(); // Array of nodes from path to last node from next()
}
```

- Our `GraphADT` interface (all methods required):

```
public interface GraphADT {
  boolean addDirectedEdge(Node from, Node to, float weight);
  boolean addUndirectedEdge(Node from, Node to, float weight);
  boolean removeDirectedEdge(Node from, Node to);
  boolean removeUndirectedEdge(Node from, Node to);
  boolean containsEdge(Node from, Node to);
  GraphIterator breadthFirstIterator(Node start);
  GraphIterator depthFirstIterator(Node start);
  GraphIterator bestFirstIterator(Node start);
}
```

- Finally, the `NodeBag` interface:

```
public interface NodeBag {
  void insertNode(Node n); // Put a node into the bag
  Node removeNext(); // Pull a node from the bag
  boolean isEmpty(); // True if bag is empty
}
```

Suggested Problems. 8-puzzle (pg 89), Farmer/Wolf/Goat/Cabbage (pg 620), Sliding Tiles (pg 156), Missionaries and cannibals (pg 676).
All page numbers refer to the 4th edition.

C++ Addendum If you wish to do this project in C++, then the following considerations might be useful:

- All objects in Java inherit (at some point) from the class named `Object`. Thus, in the methods defined above, method parameters of type `Object` could be of any class.
- There are no pointers in Java. Furthermore, objects are not actually passed to functions. Instead, *references to objects are passed by value*. You will probably want to take objects by references in many of the above methods.
- Unlike a parent class, a Java interface does not implement any methods, but defines methods to be defined by classes that implement it. Thus, Java interfaces are roughly equivalent to abstract base classes (ABC's) in C++.
- ABC's in C++ define at least one pure virtual member function, i.e., a member function with no implementation. In this case, all of the methods in the ABC should be pure virtual. The syntax would be:

```
class GraphADT {
  virtual bool addDirectedEdge (Node &from, Node &to, float weight) = 0;
  virtual bool addUndirectedEdge (Node &from, Node &to, float weight) = 0;
  ...and so on.
}
```

- `Collection` is not defined by the standard library in C++, so it would probably make sense to combine the methods declared in `Collection` and `GraphADT` into a single ABC. You could also change the parameters types from `Object` to `Node` in the `Collection` interface.
- C++ has no class objects. For your iterator constructors, it might make sense to simply take an object of type `NodeBag`, which can be instantiated before the `Iterator` constructor is called.
- Memory management is handled by a garbage collector in Java, so there is no need for destructors. You might consider implementing destructors in your classes.