CSC 207 **Assignment 7: Doublet Priority** **Spring 2011**

Assigned: Monday 4 April 2011

Due: Tuesday 12 April 2011

Objectives:

- Increase familiarity with **polymorphism**
- Continue practice with the Java **Collections** Framework
- Use **priority queues** in a realistic application
- Gain experience developing and using **inner classes**
- Learn how to **unit test** more complex objects
- Practice empirical **analysis**
- Refine **technical writing** skills

Collaboration: This homework assignment may be completed individually or in pairs.

Submission: Follow the instructions for submitting programs via P-Web and handing in a printed copy. Be sure to generate unit tests and a statement justifying their sufficiency.

In this assignment, you'll continue working with the support structures from the previous two assignments to implement another management strategy in the search for a chain of links between doublets.

You can find an implementation binary for this and the previous assignment on the MathLAN in `~weinman/courses/CSC207/code/doublets`. The file `doubletspq.jar` contains `Doublets`, `Links`, `Chain`, `ChainManager`, `StackChainManager`, `QueueChainManager`, and `PriorityQueueChainManager` (see below). Documentation for these may be found in the subdirectory `docpq/`.

Background

In your prior experiments, you likely found that using a stack to manage the search is akin to following a chain all the way to completion first, no matter how long it may be. Conversely, using a queue to manage the search is like trying (and remembering!) every possible link at every possible stage. In this way, a queue unfolds the search into a large tree of possibilities that expands layer by layer until you find a solution or no possibilities remain.

The stack manager tends to find chains that are unnecessarily long. While, the queue manager is guaranteed to find the shortest solution, it can take gobs of memory. Both of these managers somewhat unintelligently choose which candidate to explore next. Can we perhaps do better?

When new chains are added to a queue, they are necessarily at the end. Thus the queue manager returns the candidate with the shortest chain for examination. While this is an advantage, when several chains have the same length, we have no way of distinguishing which may be better. For instance, which of the following two chains would you prefer to be working from for the doublet *head/tail*?

head-held-hell-hall-hail
head-heat-feat-fear-pear

If you've played the game before, you'd probably pick the first. Why? After five moves, the end of the first chain differs from the goal word by only a single character. By contrast, the end of the second chain differs from the goal word in all four characters. We have an intuitive sense that the first is better. Can we formalize this somehow?

Instead of returning the shortest chain for examination, we would like to return the chain with the shortest *total estimated solution length*. That is, when a particular incomplete chain is extended to yield a solution, what will the solution's length be? We already know how long the candidate chain is, but how can we estimate how much longer the chain must be to reach the solution? One optimistic estimate is to

count the number of characters that differ between the chain's end and the goal word. Because you can only change one character at a time, the remaining chain must have at least this many words in it (though perhaps more).

The value of a chain (for a particular doublet) with a particular "Length" is thus given by

$$\text{Value} = \text{Length} + \text{Differences},$$

where "Differences" means the the number of characters that differ between the goal word and the end of the chain.

In this assignment, you'll create a new chain manager that efficiently returns the most promising chain for consideration in the search; that is, the one with the lowest value. Will this strategy improve our situation? We will find out!

Priority Queue Chain Manager

If you hadn't already guessed, one of the most efficient ways to keep a collection of things and quickly access the "best" is a priority queue. In this part, we will build up a `PriorityQueueChainManager` that implements the `ChainManager` interface. You can begin by creating a skeletal implementation.

Java's `PriorityQueue` uses the elements' so-called "natural ordering", which is revealed by the `compareTo` method. That means the items it stores must be `Comparable`. To this point, you've likely been storing items inside the stack and queue as `Chain`s. However, these are not comparable. While it might make sense to compare chains by their length, our new manager will require that we compare candidate chains by their total estimated length.

One might argue that a chain is independent of a solution or even the doublets or the search strategy; it is just a sequence of words. Your prior managers knew nothing about the doublets, chain length (explicitly), or even the search they played a part in. Because we might use estimation methods other than the one outlined above, the estimated total length is something that should be the chain manager's responsibility for calculating. In addition, it is often true in real-world practice that you are handed classes and objects, and you cannot change them. Consider that to be the case here: the `Chain` class simply cannot be modified.

Constructor Add a constructor to your `PriorityQueueChainManager` class that takes the ending word as a parameter (which should then be a member of your class).

Distance Estimate Add a private method to your `PriorityQueueChainManager` class called `targetDistance` that takes a string and returns the number of characters that differ from the target string. It may take as a precondition that the strings are the same length.

Entry Class Because chains are not directly comparable using their length, we must create a new composite type to store in the priority queue. This new type will use the "Value," as defined above, as the basis of its natural ordering.

Create an inner class of `PriorityQueueChainManager` called `Entry`. Its constructor should take a `Chain` as a parameter, calculate and store the total value of that chain using its length and your distance estimate, as outlined above.

The class should implement `Comparable` using the total value as the basis for comparison. (Remember that `Comparable` is a parameterized interface!)

Completion You should now be able to complete the implementation of your priority queue-based chain manager. A call to `next` should always return the chain that has the lowest value (length plus differences).

You can now augment your `Doublets` program so that it can use the new search strategy as an option. Alternatively, the version of `Doublets` in `doubletspq.jar` works as in the previous assignment, but now with a priority queue-based search option.

Unit Testing

Note that you should be testing every `public` method of every (`public`) class you write. Private members influence the behavior of the class and are private because they are implementation details that are likely rightly hidden. Your job as a tester is to come up with cases that appropriately test these inner parts implicitly.

Analysis

Write a short paper examining and explaining the behavior of your program on this problem. The audience for your writing is your peers in this class. While your essay should include all the hallmarks of good writing (e.g., paragraph cohesion and unity, transitions, clear language), you should be sure to incorporate answers to the following questions along the way:

- For a given doublet, how does the solution quality of the priority queue compare to the standard queue?
- Can you empirically characterize/compare the time complexity? That is, how do the number of candidates examined compare between the priority queue-based and queue-based searches? Report examples.
- What about space complexity? How does the (maximum) size compare between the priority queue-based and queue-based searches? Report examples.