P[CST8130: Data Structures

Assignment 4: Searching Postal Codes by Distance

Overview

In Assignment 1, you loaded and organized a LARGE (58 MB) data file consisting of ALL postal codes across Canada (about ¾ million records). Each postal code entry identified the city, province, latitude and longitude.

You will rework Assignment 1 to provide enhanced search capabilities. Given a specific *latitude* and *longitude*, allow the user to find the closest *PostalCode*. You will implement a **BRUTE FORCE** algorithm that has O(*n*) characteristics for a single lookup. It will work, in fact, it will be guaranteed to find the closest match, but the performance will be pathetic in a BIG enterprise system. There are a huge number of entries: about ¾ million entries. Searching *n* lookup requests amongst the *n* entries would result in O(n^2) performance (which is nasty). Nevertheless, you do need this **BRUTE FORCE** algorithm because it does guarantee finding a correct search result. It will be an important tool in testing the behaviour of your improved search algorithm. Your improved algorithm should find the correct answer, and the **BRUTE FORCE** algorithm will be used to verify your results.

Following a lookup, display the search result: console for *jUnit*; and JavaFX GUI for single GUI lookups. There are several different ways you can display the result in the GUI. I'll let you use your creativity in crafting the most elegant JavaFX technique.

Finding the Distance

Perform an internet-based search for suitable code to calculate the distance between 2 points on the Earth's surface (and the 2 points will be defined by their *latitude*/*longitude*).

Of course, you must provide attribution for the source code you found to calculate the distance (to avoid accusations of plagiarism).

The Brute Force Algorithm

Here's a pseudo-code description of the **BRUTE FORCE** algorithm. When the method finishes, it can return one of two things, depending on how you want to display the lookup result (look to **Displaying Search Results in JavaFX**). If you intend to display the actual *PostalCode* object (using *toString()*) you can return the reference-to the *PostalCode* object; if you want to scroll the target into view in the *ListView* and then highlight it, you can return the index value.

- Method receives the target *latitude*/*longitude* values.
- Create a *double* variable to track the closest distance. Initialize it with *Double.MAX_VALUE*. Later, when calculating the distance between the target *latitude*/*longitude* values and the first *PostalCode*, the calculated distance will be guaranteed to be smaller than *Double.MAX_VALUE*, and *Double.MAX_VALUE* will be overwritten by the new and first calculated distance.
- Create a variable to track the closest *PostalCode* object so far. This could be an *int* if you intend returning the index position, or it could be a reference-to *PostalCode*.
- Iterate through the entire collection of ¾ million *PostalCode* objects.
    - Calculate the distance to this *PostalCode* object.
    - Compare the newly calculated distance to the current known closest distance. If this one is closer, reset both the current known closest distance, and the variable tracking the closest *PostalCode* object (either the *int* or the reference-to).
- After exiting the iterating loop, you will have identified the closest *PostalCode* object.

Improving Performance with a *QuadTree*

Tree structures provide excellent performance improvements over linear structures. Clearly, a *TreeSet* will perform lookups MUCH faster than a *LinkedList*. But you CANNOT use a *TreeSet* for to manage *PostalCode* objects for the purpose of searching distances. Distance requires manipulating spatial data (that is *latitude* and *longitude* values), and you will need to create a *QuadTree* to organize your *PostalCode* objects. A *TreeSet* has 2 paths flowing from each node: *left* and *right*. Your *QuadTree* will have 4 paths: *north-east*, *south-east*, *south-west* and *north-west*.

I provided working code for a simplified *BinarySearchTree* class. Use that as a starting point to create a *QuadTree* class.

I will publish selected snippets of Java code to assist in building your *QuadTree*.

jUnit Tests to Assess Performance

Implement the **BRUTE FORCE** algorithm, and capture the elapsed time of the **BRUTE FORCE** lookup. The elapse time can be compared to your improved *QuadTree* algorithm.

Implement a mechanism to generate many random *latitude:longitude* pairs of numbers so that you stress test your solution as if it were part of a production system (imagine that you're now working for *Google Maps* and have to build a high-performance, planetary software tool). Because the **BRUTE FORCE** lookup is O(*n*) for a single *latitude:longitude* pair, looking up *n* items would result in an O(n^2) algorithm . . . and that's **BAD** . . . how bad? . . . track the elapse time for growing numbers of lookups.

There are two overriding components to testing. First, verify that your improved search algorithm does in fact find the closest matching *PostalCode* object (and the **BRUTE FORCE** algorithm will support this testing). Second, craft a systematic approach to measuring performance by measuring the *elapse time*.

The first phase of testing will involve single search actions. A later phase of testing will *stress test* your application. The

stress testing will not involve the use of a GUI. You will want to evaluate the performance of the search algorithm with a massive number of lookup operations, but without encountering the performance bottleneck of GUI display.[1]

Displaying Search Results in JavaFX

For single lookups by a user, make it beautiful with JavaFX.

You will need at least two *TextField* objects: one to capture the user's *latitude* entry; the other, the *longitude* entry. You'll probably also need a *Button* object to trigger the retrieval of the *TextField* values and then launch the search process.

You can display a single postal code lookup result in one of several different ways. Here are two:

- Call the *toString()* method for the matching *PostalCode* object and display it using a *Text* object that has been added to the *scenegraph*.
- Highlight the target *PostalCode* entry directly in the *ListView* control and scroll it into view using code something like this. Of course, this presumes that you have identified the index position of the matching *PostalCode* record:
  listView.getSelectionModel().clearAndSelect(indexClosest); listViewCodeOrder.scrollTo(indexClosest);

The performance statistics can be output as text using: **System.out.printf(. . . .);** output since this information is diagnostic / testing information, not a requisite part of the GUI. On the other hand, you could also display it on the GUI.

The *stress testing* in your *jUnit* code, will not generate any GUI output. Any *jUnit* results will be displayed solely on the console.

Submission Requirements

The following need to be part of your submission.

- *Title Page*: It must identify you, the course, section number, course professor, assignment name and number, date of submission.
- *Memory maps*: Your *memory map* will communicate the kind of data structures you are using to implement the postal code analyzer. You can ignore *Collection* and *Map* fields that do not contribute to the clarification of your data structure choices.
- *Source and Executable code*: I will evaluate on your computer.
- *Problem Statement*: In your own words, write a paragraph or two that describes what you are attempting to achieve with this assignment. You must justify the *wise choices* that you make for your implementation with references to Big-O issues.

The *Problem Statement* is weighted more heavily since you'll be justifying your implementation choices with reference to Big-O issues.

CST8130: Data Structures

Lab Assignment 4: Searching Postal Codes by Distance

Your lab assignment has been evaluated using the following criteria:

- ☐ Problem Statement including justification of *Wise Choices* (2 marks)
- ☐ Test Plan (1 mark)
- ☐ Memory Maps (2 marks)
- ☐ Source Code: Indentation / Naming / Commenting / Attribution for other code (1 mark)
- ☐ Successful Execution (3 marks)
- ☐ Format / Organization (1 mark)

Subtotal /10

- ☐ Penalties (unlimited).
- ☐ Basic JavaFX functionality is expected. Enhanced GUI design deserves bonuses (up to 3 marks added)
- ☐ Other Bonuses (up to 3 marks added).

Total /10

[1] Remember, under regular use, the GUI will be exercised at human speed, perhaps one lookup every few seconds. If this algorithm were to be embedded in a server-side application, it could be hammered tens of thousands of times per second. But the server response would not involve detailed GUI activity at the server side. The GUI activity would reside on the client side where one of the tens of thousands of individual users might be making a request. So, stress testing needs to be done without GUI activity.