Due: Wednesday 6 May 2009

Topics: Recursion, trees, search.

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 test cases and a statement justifying their sufficiency. One only one submission (paper and digital) per group is necessary.

1 Introduction

Boggle® is a (fun!) multiplayer game involving a 4x4 square board of letters. The letters appearing on the board are the faces of sixteen 6-sided die. Thus, a board is randomly generated (with some constraints). The game is essentially an amped-up word search. The goal to generate words that may be found on the board by tracing paths through adjacent squares without re-using any squares. Adjacency can be vertical, horizontal, or diagonal. Words that any two players find in common are eliminated, and any remaining words are scored based on length.

Your task in this assignment is to write the most (asymptotically) efficient computer Boggle® player possible. You will square off against programs written by your peers for the first (annual?) CSC207 Boggle® Championship. We will use the official game rules as described here: `http://www.centralconnector.com/GAMES/boggle.html`, with the exception of the time. Since computers are much faster than people, your program will be given limited time (with as much granularity as standard Java allows) to search the board for words.

2 Set-Up

Two interfaces have been defined for coordinating game play, `BoggleMaster` and `BogglePlayer`. You will be provided bytecode for a `BoggleMaster` to test your implementation, but you will create a `BogglePlayer` implementation. Just four methods connect these, making the game play relatively simple.

After instantiating a `BogglePlayer` object, the tournament master will call the player's `setWords` method to communicate the list of valid words. After this, the master calls the player's `play` method, passing a reference to the master. This gives the player the ability to begin searching for words The player may use the `getBoard` method of the master to determine the board to search, and as it finds words (since time is limited), it may call the master's `add` method to add a discovered word to the list. The only other requirement is that your `BogglePlayer` have a public constructor that takes no arguments.

3 Implementation

We can write a range of solutions. The following pseudo-code algorithm will work, but it is not very efficient. B represents the set of board squares so that $s \in B$ is a particular location (square) on the board. L represents the "lexicon" of possible words, and V represents the set of squares that have been visited along the path thus far. The symbol ϵ represents an "empty" string, and $\emptyset$ represents an empty set. Thus, $\cup$ is the usual set union operator and $+$ is the string concatenation operator. We have compactly noted that the same operation holds for each adjacent square by indicating a procedure (Adjacent-Squares) that returns a set of neighboring/adjacent locations on the board. Since the board is a square, some locations will have more neighbors than others. In summary, keep in mind that the pseudo-code uses mathematical abstractions. You will need to figure out what code representation is most practical and/or efficient for your program.

3.1 Take One

Algorithm 1 Recursive word search for a Boggle® board.

```
BOGGLE-SEARCH
    for each s ∈ B
        SIMPLE-WORD-SEARCH(s,CHAR-AT(s),{s})

SIMPLE-WORD-SEARCH(s,p,V)
    if p ∈ L
        ADD-WORD(p)
    for each n ∈ ADJACENT-SQUARES(s)
        if n ∉ V
            SIMPLE-WORD-SEARCH(n,p + CHAR-AT(n),V ∪ {n})
```

Note that this is a recursive algorithm. Thus, the first thing we should be doing is looking for a base case. When does recursion terminate? It is only when we try to re-visit a square that has already been visited. The result is that this algorithm searches every possible path, starting from every possible board location, asking along the way whether the characters from the beginning to the current point along the path form a word. If every path and subpath formed a word, this would be a great algorithm. The chances of that happening are pretty slim, though. A functional (but non-efficient implementation) of this algorithm would probably garner a C.

3.2 Take Two

What we'd like to do is stop the search even earlier if the path being traced out can not form a word. Fortunately, you have a data structure at your disposal from the last assignment that allows you to ask quite easily and efficiently whether a string p is a prefix of any word in the lexicon. How should you use this information? Well, if the current prefix is not in the trie, there's no sense continuing to search along the path, since the addition of more characters cannot possibly transform the prefix into a word.

This is a better algorithm, but how efficient is it? We know that we will stop exploring paths that can't contain words, so that's an improvement. How costly is it to ask whether something is a prefix? If your data structure is written correctly, it's $O(|p|)$, where $|p|$ is the length of the prefix being checked. Well that seems not too bad. A functional implementation of this better algorithm would likely garner a B.

3.3 Take Three

Assuming this prefix check is made about the same place we ask if $p \in L$ (also a $O(|p|)$ operation), then what is the difference between the check for p and the check for $p + \text{CHAR-AT}(n)$ made in the recursive call? Not that much. In fact, you'd be repeating the same $|p|$ operations as before, with one additional operation for the extra character. Getting rid of this redundancy will make your algorithm *much* more efficient. How can you do that? Well, you will need to pass around more information with the search that keeps track of the computation done so far. If you have implemented the `WordTrie` interface from the previous assignment, then you have all the data structure functionality you need, and it is simply a matter of putting it to use. Implementing this would likely garner an A.

3.4 Final Concerns

The final question is how to maximize your efficiency (remember, there is a tournament at stake here!) You will need to think carefully about how you track your visited squares on the board, represent the current prefix, and reuse your prefix check computations. With the overarching advice of "make it *work* FIRST, make it work *fast* SECOND," I offer the following considerations:

- Remember that you are (or probably should be) writing object methods. Objects are methods plus state (variables). While the algorithm outline given above is part procedural (ADD-WORD) but mostly

functional (SIMPLE-WORD-SEARCH), you can capture some search state information in member variables, rather than parameters.

- `String` objects are immutable. `StringBuilder` objects are not. (They are backed by a `char` array, and so act more like C's `char*` or a specialized `ArrayList<Character>`.)
- Depending on how you take the above suggestions, you should make your code easier to read and understand by using small (perhaps co-recursive) methods that only do one or two things. This will reduce the chances for error and make debugging easier. The overhead of function calls will really be minimal here and is rarely worth the benefit to eliminate them at the cost of clarity.

(If you knew ahead of time what the letter cubes were, you could also possibly do even better.) The top-performing team(s), who also keep their code well-documented and understandable will likely garner an A+.

My implementation tops out at about 100 lines (not including import statements). It's not tremendously complicated, but the recursion and special cases make it very error prone, so think carefully!

4 Practica

4.1 Getting Started

You may find bytecode for some implementations and the Java source for the interfaces on the MathLAN:

`~weinman/courses/CSC207/code/boggle`

Copy the entire directory. Note that the interfaces belong to the package `boggle`. That means you will have to keep them in a directory called `boggle`.

In addition, your implementation must be called `boggle.`*`username`*`.MyBogglePlayer`. This means you must at least submit a file `boggle/`*`username`*`/MyBogglePlayer.java` and it will belong to `package boggle.`*`username`*. Other files may be submitted, but they must also be a part of that package, because everything will be placed under your *`username`* directory.

4.2 Testing

If you wish to have the `BoggleMaster` run your code on a board and score it, you may use the `ConcreteBoggleMaster` program, as follows (be sure your present directory is just above the `boggle` directory).

`java boggle.ConcreteBoggleMaster` *`wordFile [seed] username`*

The `seed` argument is optional; it should be an integer that seeds the random number generator used to create the board. You may find it necessary to increase the heap allowed by Java by using the `-XmxNNNNm` switch. Here are two examples:

```
$ cd ~weinman/courses/CSC207/code
$ java -Xmx200m boggle.ConcreteBoggleMaster boggle/words.txt weinman
ESTJ
PLTW
SYER
TLHN
(Possible words: 104)
        Score for 104 words: 275
$ java -Xmx200m boggle.ConcreteBoggleMaster boggle/words.txt 27 weinman
NEEA
BART
TIEN
TREF
(Possible words: 130)
        Score for 130 words: 379
```

The board used is shown first, and the number of words found in that board from the word list is then given, along with the score for those words.

4.3 Submission

Submit your code as a tar file including the entire hierarchy, with only your Java files. You may do this from one directory above the `boggle` directory with the following command:

```
tar cf username.tar boggle
```