CS1210 Computer Science I: Fundamentals
Homework 3: Boggle!
Due Friday, December 8 at 11:59PM

Introduction

This is the first part of the last project for CS1210. I will release an updated version of this document containing the second part sometime over the holiday week; please don't wait to start on this project, or I can guarantee you won't finish it. So, just to review:

(1) This is a challenging project, and you have been given two weeks to work on it. **If you wait to begin, you will almost surely fail to complete it.** The best strategy for success is to work on the project a little bit every day.

(2) **The work you hand in should be only your own; you are not to work with or discuss your work with any other student.** Sharing your code or referring to code produced by others is a violation of the student honor code and will be dealt with accordingly.

(3) **Help is always available from the TAs or the instructor during their posted office hours.** You may also post general questions on the discussion board (although you should never post your Python code). **I have opened a discussion board topic specifically for HW2.**

Background

In this assignment, we will be writing the skeleton for a simplified version of the game Boggle. Boggle is a popular party game; for more information on Boggle, visit:

https://www.wikihow.com/Play-Boggle

We will be building a version of Boggle starting from the Othello platform we studied in class. Our version of Boggle will use exclusively 5-letter words from the Word Ladder problem we (also) studied in class. You will notice that the template file for this homework is significantly less complete than those for previous labs and homeworks. That's because each homework assumes you will do more and more of the work yourselves. Both the Othello code and the Word Ladder dictionary are available on ICON; feel free to use these as models.

The rest of these instructions outline the methods that you should implement, describing their input/output behaviors. As you work on each function, test your work on the document provided to make sure your code functions as expected. Feel free to upload versions of your code as you go; we only grade the last version uploaded, so this practice allows you to "lock in" working partial solutions prior to the deadline. Finally, some general guidance.

(1) Start with the template file and **immediately complete** the *hawkid*() function so that we may properly credit you for your work. **Test** *hawkid*() **to ensure it in fact returns your own hawkid as the only element in a single element tuple.** Also be sure to add your name and section to the comments at the top. Ignore this instruction at your own risk: some of you have become serial offenders and I have had to manually regrade your code too many times!

(2) As with HW1 and HW2, you will be graded on both the **correctness** and the **quality** of your code, including the quality of your **comments**!

(3) Be careful with iteration; **always choose the most appropriate form of iteration (comprehension, while, or for)** as the function mandates. Poorly selected iterative forms may be graded down, even if they work!

class Boggle()

Our version of Boggle is always played on a 5x5 board, which is populated with letters according to the distribution implicit in the file *words.dat*. The __init__() method for class Boggle should create a representation of the Boggle board, and pop up a TKinter version of the board, showing letters randomly allocated to each of the 25 locations. *Hint:* the random allocation of letters should be weighted using the random.choices() function from Python's random module and based on the data obtained from the readData() method described next.

Boggle.readData(self, file)

The readData() method[1] will be called from the __init__() method. It should open the specified file and read in the list of words it contains. It should construct two data structures, both variables in the class Boggle.

The first data structure (let's call it F, for frequency) should be a dictionary indexed by the letters in the words read in. As in HW2, the construction of F should involve counting the number of letters of the type specified by the key k in F[k]. Once you have read in all the words, each F[k] should be recast to represent the probability that a letter, drawn at random from the collection of letters contained in the words of the file you have just finished reading, is, in fact, the letter k. You will use this value to create the population and the weights parameters for the random.choices() function.[2]

The second data structure (let's call it T, for trie, the technical name for this kind of data structure), is a deeply nested set of dictionaries, each using letters as keys. At the outermost level, the value of key k will itself be a dictionary, also using letters as keys. At the innermost level, the key will still be a letter, but the value will be a word from the file you just read in. An example will help make this clear.

Assume your file contains only the word 'paste'. The value for F and T would then be:

```
F = { 'p':0.2, 'a':0.2, 's':0.2, 't':0.2, 'e':0.2 }
T = { 'p':{ 'a':{ 's':{ 't':{ 'e':'paste' }}}}}
```

Notice how F represents the frequency estimates for the letters in the (lone) word from the file, while the trie indexes each letter of the word, in order as you descend the nested dictionaries, until you get to the original word. If the file contained instead the words 'paste' and 'pasta', the value of F and T would instead look like:

```
F = { 'p':0.2, 'a':0.3, 's':0.2, 't':0.2, 'e':0.1 }
T = { 'p':{ 'a':{ 's':{ 't':{ 'e':'paste', 'a':'pasta' }}}}}
```

If it also included 'pasha', T would look like:

```
T = { 'p':{ 'a':{ 's':{ 't':{ 'e':'paste', 'a':'pasta' }, 'h':{ 'a':'pasha' }}}}}
```

We will use this second data structure in the second half of the assignment.[3]

---

[1] The original handout omitted the *self* argument, which of course is necessary for every method of class Boggle().

[2] You should also feel free to use cumulative counts and the cum_weights parameter instead of frequencies and the weights parameter.

[3] There was an extra curly bracket in this expression which has since been corrected.

Boggle.ckSoln(self, soln)

This function should check that the solution specified (a list of tuples, with each tuple corresponding to a (row, col) location on the board) is a viable solution. To be viable, a solution should consist of board locations that differ by at most 1 in either row or col (that is, we only allow solutions to consist of successive tiles that are to the left, right, above or below the previous tile in the solution). In addition, a solution must correspond to a valid traveral of the trie, meaning that the letters in its constituent locations could, when properly extended, lead to a word in the dictionary. The function should work for solutions of any length, up to the maximum "depth" of the trie. It should return False if the solution is not viable, and should return the value of the dictionary in the trie indexed by the solution. If the solution is of the correct length, this value will be the word from the dictionary; otherwise it will consist of one of the embedded dicionarites in the trie.

An example should help make this clear. Assuming the value of T given in the previous example, and assuming the word 'pasta' starts in the left-top corner of the board and drops down the second column, we would have:

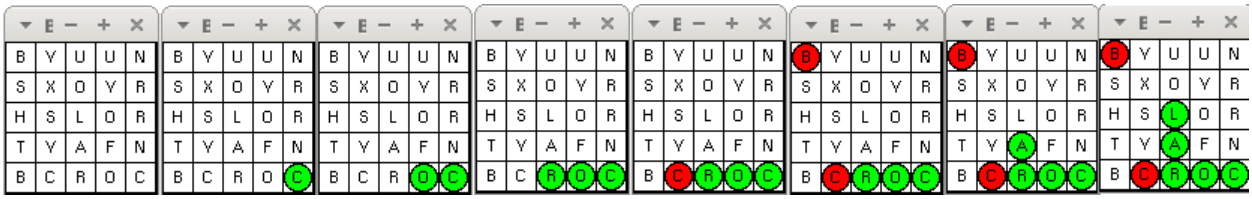
```
>>> Boggle.ckSoln([(0, 0), (0, 1), (1, 1), (1, 2), (1, 3)])
'pasta'
>>> Boggle.ckSoln([(0, 0), (0, 1), (1, 1)])
{ 't':{ 'e':'paste', 'a':'pasta' }, 'h':{ 'a':'pasha' }}
>>> Boggle.ckSoln([(0, 0), (0, 1), (0, 2)])
False
```

The last example is what would happen if the solution were to correspond, for example, to the letters 'paq'.

Boggle.extend(self, event)

This method should be bound to the left mouse button. It is used to extend the user's solution, which we sill store in a variable self.soln. Each time I click on a tile, I should check that the solution is still viable (that is, is legally extends the current solution in self.soln and corresponds to a legal traversal of the trie and thus can still possibly lead to a word in the dictionary). If the solution is viable, I should color the new tile green; else I should color the new tile red. When the solution in self.soln corresponds to a complete word in the trie, you should print out a message indicating the word that was found by the user and then reset the puzzle.

Here is a progression of extent events. The puzzle contains the word 'coral'. Note how each successive letter found turns green, but when the user picks 'corc' — which does not correspond to any word in the dictionary — or when the user picks a letter that is not contiguous to the partial solution 'cor', the selection turns red. The user is always free to complete the partial solution indicated by the green letters.



Boggle.new(self, event) and Boggle.reset(self, event)

These two methods should be bound to the middle and right mouse buttons, respectively. The new() method should create a new Boggle puzzle in place, overwriting the old puzzle and setting self.soln to [], while the reset() method should redisplay the board and reset the value of self.soln. In this way, it can be

used both to present a new puzzle as well as to allow the user to find the next word (by clearing any colored tiles and resetting self.soln to []).

Boggle.solve(self)

This method should return a list of all of the words found in the current puzzle. The general strategy should be to conduct independent searches starting from each square of the puzzle and then merge all the solutions together. If a word appears in more than one fashion within the same puzzle, the word should appear that many times in the value returned. A puzzle with no solutions should return [] (the puzzle shown above would return ['coral'] as that is the only word it contains).

Closing Words

You should feel free to add other methods, especially if they can be shared across different parts of your solution. For example, you might create a separate ckPath(self, soln) method that checks to see that a partial solution represents contiguous tiles in the puzzle.