Lab 7: Boggle[1]

Due: 9 November 2016

In this lab, you are going to write a program to play Boggle. A Boggle board is a 4x4 grid of 16 dice, randomly distributed. The 6-sided dice have letters on their faces, creating a 4x4 grid of letters. In the human version, the players write down all of the words they can find in three minutes by tracing paths through neighboring dice faces. Two dice are neighbors if they are next to each other horizontally, vertically, or diagonally. So a die has up to eight neighbors; dice on the edges of the grid have fewer neighbors. A die/letter can only be used once in a word and a word must be at least 3 letters long. Play stops at the end of the three minute session. Words found by more than one player are removed from all players' lists and the players receive points for the remaining words: 1 point for 3-4-letter words, 2 points for 5-letter words, 3 points for 6-letter words, 5 points for 7-letter words, and 11 points for words with 8 or more letters). Highest score wins.

The Assignment

I have provided a **PlayBoggle** class that creates a **Boggle** object and calls the play method. You must write the **Boggle** class. Write the class so that the user goes first and enters as many words as they can find, with no time limit. Your program needs to check each word for legality (long enough, on the board, and in the dictionary) and save the legal ones, because we are going to give the poor human a little help by not allowing the computer to use any of the words the human finds. So, you will need to disallow any words the computer finds that the human has already found. Both scores should be computed and displayed, with appropriate words of sympathy for the human. Your program should:

1. Get the dictionary file name and create the dictionary.

2. Get the Boggle board file name, read the board, and print the board, in upper case.

3. Ask the user if they 1) want to play, or 2) just have the computer solve the puzzle.

4. If they want to play:
    1. Tell them what symbol(s) they need to enter to signify that they are done, e.g. “!”.
    2. Allow them to enter words.
    3. If a word is in the dictionary, long enough, on the board, and they haven’t already used it, save the word and let them know that it counts; if it doesn’t count, ***let them know why***, in the following order:
        1. It’s not in the dictionary.
        2. It’s in the dicitonary, but it’s not long enough.
        3. It’s in the dictionary and long enough, but not on the board.
        4. It’s in the dictionary, long enough, and on the board, but they’ve already used it.
    4. When the user is done, display the words they found in alphabetical order (***with no repeated words***),

5. Whether the user plays or not, have the computer find *all* the legal words on the board (in

---

[1] Adapted from a Data Structures lab by Eric Chown.

the dictionary and long enough),

6. Display the words found by the computer in alphabetical order (***with no repeated words***), indicating which ones were found by the human (if the human played); note that the human words will be a subset of this list of words.

7. Compute and display the score for the human (if they played) and the computer, remembering not to count words for the computer that the human found.

NOTE 1: In Boggle, Q and U are not on separate die faces. Since a Q is virtually worthless without a U, they always appear together on a die face, i.e. QU. To simulate this, whenever you see a Q, treat it as if it is a QU, both for word finding purposes and word scoring purposes, i.e. it counts for 2 letters, even though it's on one die face.

NOTE 2: There is a method in the **Collections** class that you will find useful. If **humanWords** is an **ArrayList** of **String**s, the following will sort those **String**s in alphabetical order:

Collections.sort(humanWords);

To use this, you will need to import the **Collections** class:

import java.util.Collections;

NOTE 3: Write your program so that the computer finds all the words first, but does not print them out or score them. Then, if the human plays, it is easy to check if a word they try is actually in in the board. Regardless of whether the human plays, you can then print out and score the computer list of words (remembering to disallow any words found by the human).

NOTE 4: One way to solve a Boggle board is to run through the dictionary and repeatedly check if each word is on the board. **Do not use this approach — if you do, you will get 0 credit.** Your solution should recursively assemble character sequences from the puzzle by following the Boggle rules, and check if they are in the dictionary. **Your program must find words recursively — if it does not, you will get 0 credit.**

Dictionary

An object created from the **Dictionary** class will hold the list of all legal words. You should use an **ArrayList** of **String**s to hold the words. I have provided a list of words in the file **d.txt**. Your **Dictionary** class should do two basic things:

1. read the dictionary into an **ArrayList** of **String**s, and
2. provide a **binarySearch** method to check if a given string is in the dictionary.

The **readPuzzle** code I gave you for the **Sudoku** lab should be very helpful here, since you need to get a file name from the user, open the file, and crate a **Scanner** object to read data from the file. Recall that if **scan** is a **Scanner** reading from a file, **scan.next()** will return the next line as a **String**. Since **d.txt**, the dictionary file, contains one word per line, this should be useful. In the Sudoku lab, puzzle files and solutions were in a **puzzles** folder; here, the

dictionary file, and four board and solution files, will be in a folder called **files**.

Keep in mind that you need to search for words recursively. And remember that you can change direction when looking for a word. For example, a word can be assembled by starting at a letter, going right, then up, then left, then “northeast,” etc. So, at any given point in assembling a character sequence, there are eight directions you can go in to continue assembling the sequence (fewer if you are on an edge or in a corner). The only restriction is that a letter on the board cannot be used more than once in the same word.The ideas behind counting blobs should be useful here.

Do your testing with small files, e.g. a dictionary with a small number of words, only some of which are on a board that you create. Also, we strongly recommend that you get the computer player part working first before you add the human player part. We have provided four Boggle boards with solutions that you can test your program on. The first one also illustrates the required interaction with the user. You do not, however, need to copy the (attempt at) humor. In fact, you are encouraged to come up with your own. The other three solutions come from a version of the program where the human is not allowed to play; you should use the solution to the first test board as a guideline for what your output should look like. We have also supplied an example **main.cpp**. You may, but do not need to, use it (without loss of credit).

Submitting Your Program

Please submit your program in the usual way on BlackBoard and turn in a hardcopy.