Assignment 4 README

Here is a description of the fourth programming assignment for CSE 100, Fall 2013. In this assignment, you will implement the back-end logic for a graphical version of the popular board game Boggle®. Specifically, you will design and implement the data structures and algorithms necessary for a computer player to play against a human opponent. If you get everything right, the computer player should almost always "win" the game by a large margin, but it will be fun to play anyway.

The main focus of this assignment is designing and implementing the data structures and operations on them that will efficiently find words that appear in a Boggle board. Because your focus will be on the back-end logic, we will provide the graphical display shell. This shell knows how to generate and render a random Boggle board, interact with a human player, and display results of the game. It will use your code to verify input from the human player, and to obtain the computer player's results. Note, however, that the full specification of the requirements given below are *not* fully tested by this GUI application and for you to fully test your program, you may need to write other code that does not use the GUI. The steps required to create the GUI application are described below.

>>> **Due Fri Dec 6 8:00 PM**

>>> Checkpoint submission Tue Dec 3 8:00 PM

>>> **Required solution files: boggleplayer.h, boggleplayer.cpp, boggleutil.h, boggleutil.cpp**

The files that contain your solution to the assignment must be turned in by the due deadline. The turn in procedure is described below in the section "Turning in your assignment".

It's a good idea to plan to do the assignment well before the deadlines. Although the GUI component has been completed for you, there is still plenty left for you to do, so start early. As always, terminals and tutors can get very busy at the last minute.

In doing this assignment, it is your responsibility to understand the course rules for integrity of scholarship.

Please read this whole document before you get started. In particular, note the grading section at the end which gives you some requirements (about both speed and accuracy) against which you will be graded.

Getting Started

A little background on Boggle®

The official game of Boggle® uses a board which is a 4-by-4 grid on which you randomly shake and distribute 16 dice. The 6-sided dice have letters rather than dots on the faces, so this creates a 4-by-4 array of letters in which you can find words. In the original version of the game, any number of players can play. The players all start together and write down all of the words they can spell out by tracing paths through neighboring dice faces on the board. (Two dice are neighbors if they are next to each other horizontally, vertically, or diagonally. So there are up to eight neighbors of a die; dice on the edges of the grid have fewer neighbors.) A die can only be used once in a word, and a word must be at least 3 letters long. At the end of a three minute session, all players stop recording words. Non-unique words -- that is, words found by more than one player -- are then removed from all players' lists, and the players receive points equal to the number of letters beyond the minimum word length found in their remaining words.

The version of Boggle in this assignment is different from the official one in some respects, as explained in the next section.

How the Boggle program works

In the computer version of the game that you will implement in this assignment, the human player gets to go first. (Don't worry, the computer player will still trounce you.) The human player has to enter in the text area at the bottom of the GUI window, one at a time, each word that she finds in the game board; there is no time limit for the human player. A valid word must meet the following three requirements.

1. The word must not already have been entered by the player.
2. The word can be formed from the letters showing on the board following, in order, a path through neighboring dice without using the same die twice (i.e. by following an acyclic simple path).
3. The word must exist in the lexicon of words.

When a valid word is entered by the player, it is highlighted graphically on the board display. The computer player maintains the lexicon and so determines the validity of the words, but we assume it can be trusted to do so fairly.

Whenever the human player decides she is done entering words, clicking the "Computer play!" button lets the computer get its turn. The computer searches through the board looking for all the valid words it can find. The computer typically beats the human player mercilessly, but the player is free to try again by starting a new game. Nonunique words are not removed from the players' lists (if they were, the human player would never have any words left!). Players' scores are just the number of words on their lists.

What you need to implement

`BaseBogglePlayer` is a class which declares several pure virtual member functions to define the interface that all BogglePlayer objects must provide. You will declare and define a class named `BogglePlayer` as a subclass of `BaseBogglePlayer`, and provide definitions for those functions (and any other functions you may find useful).

The member functions you must define have these signatures:

```
using std::vector;
using std::set;
using std::string;

class BaseBogglePlayer {
public:
  virtual void buildLexicon(const set<string>& word_list) = 0;
  virtual void setBoard(unsigned int rows, unsigned int cols, string** diceArray) = 0;
  virtual bool getAllValidWords(unsigned int minimum_word_length, set<string>* words) = 0;
  virtual bool isInLexicon(const string& word_to_check) = 0;
  virtual vector<int> isOnBoard(const string& word_to_check) = 0;
  virtual void getCustomBoard(string** &new_board, unsigned int *rows, unsigned int *cols) = 0
  virtual ~BaseBogglePlayer() {}
};
```

In addition, your class must provide a public default constructor and a public destructor. The source code definition of this interface is available as `baseboggleplayer.h` in your Github repository.

The semantics of these methods are stipulated as follows:

- `buildLexicon`: this function takes as argument a set containing the words specifying the official lexicon to be used for the game. Each word in the set will be a string consisting of lowercase letters a-z only. This function must load the words into an efficient data structure that will be used internally as needed by the `BogglePlayer`.

- `setBoard`: this function takes as arguments the number of rows and columns in the board, and an array of arrays of strings representing what is shown on the face of dice on a Boggle board. A Boggle board is a rectangular grid of dice; the height (number of rows) is given by the first argument, the width of this grid (i.e. number of columns) is given by the second argument. (In the original official standard Boggle® the board grid has 4 rows and 4 columns; but your `BogglePlayer` should work for any rectangular board.) The elements of the vector specify the contents of the board in row major order. That is, consider the board as a two-dimensional array, with columns indexed 0 to (width - 1), and rows indexed 0 to (height - 1). The die at index R and column C has on its face the string in the position diceArray[R][C]. Each string may be upper or lower case, and may contain one or more letters; for purposes of finding words on the board, they should all be considered lowercase. (In official Boggle, these strings would contain one character only, except for the double letter die face "Qu"; but your `BogglePlayer` should work correctly for strings of any length as we have modified the game so that a die can show multi-character strings, e.g. "ion") This function will use the information passed in to construct a data structure representing the Boggle board in a way that lends itself well to your search algorithms.

- `getAllValidWords`: this function takes two arguments: an integer specifying a minimum word length, and a pointer to a set of strings. It returns `false` if either `setBoard()` or `buildLexicon()` have not yet been called for this `BogglePlayer`. If they have both been called, it will return `true`, after filling the set with all the words that (1) are of at least the given minimum length, (2) are in the lexicon specified by the most recent call to `buildLexicon()`, and (3) can be found by following an acyclic simple path on the board specified by the most recent call to `setBoard()`.

- `isInLexicon`: this function takes as argument a `const string` passed by reference, and determines whether it be found in the lexicon specified by the most recent call to `buildLexicon()`. The function returns `true` if the word is in the lexicon, and returns `false` if it is not in the lexicon or if `buildLexicon()` has not yet been called.

- `isOnBoard`: this function takes as argument a string passed by reference. It determines whether the string can be found by following an acyclic simple path on the board specified by the most recent call to `setBoard()`. If it is possible to find the word in the current board, the function returns vector with integers specifying the locations of dice that can be used to form the word, in the order that spells the word. The integers used to specify the locations are row-major-order indices, that is, if a letter is in row R and column C, the location will be R*Width+C. If is is NOT possible to form the word, or if `setBoard()` has not yet been called the function returns an empty vector.

- `getCustomBoard`: this is a method that will allow you to implement extra functionality, should you choose to (you may find it very useful for testing). This function will be called by the GUI application when a choice of "custom", non-randomly selected board layout is requested. This function should populate the array of arrays referenced by the first argument, and the integers pointed to by the second and third argument, with a Boggle board specification suitable for passing to `setBoard()`. Note, the first argument, which will contain the array of arrays that represents the board, will not be allocated for you; you must allocate the memory for it yourself.

**** CHECKPOINT Submission (worth 5 of the 25 total points on this assignment)****

In order to get you started early, we require that you complete the `isOnBoard()` method by **Tuesday, December 3 at 8:00 PM**. This means that you will need to commit this solution to your repository before the deadline with the commit message `CHECKPOINT SUBMISSION`. You also probably want to push your repo to github at this point, though it's not strictly required. We will grade these submissions for correctness of the `isOnBoard()` method.

However, keep in mind that you will only have 3 days to complete the code for the final submission. The checkpoint is less than half way and represents the BARE MINIMUM that you should have completed by Tuesday. You should strive to be well beyond this point before the checkpoint deadline (i.e., strive to submit your checkpoint early). We will only grade up to this point while awarding the checkpoint points, but if you're done early with the part that you need to submit before the checkpoint deadline, you should move on and try to complete the rest of the project as soon as possible.

Finally, make sure that if you do go beyond the basic checkpoint requirement by the checkpoint deadline, that your checkpoint submission compiles and runs the `isOnBoard()` function correctly. That is, when you tag your push with the commit message `CHECKPOINT SUBMISSION` it should not include any code that might break the `isOnBoard()` implementation or cause the code not to run or compile. We will be testing your checkpoint submission directly.

Hints & Suggestions

In a project of this complexity, it is important that you get an early start and work consistently toward your goal. To be sure that you are making progress, it also helps to divide up the work into manageable pieces, each of which has identifiable milestones. Here is a suggested plan of attack that breaks the problem down into phases:

1. Implement a Boggle board representation: The GUI application will call your `setBoard()` method with an array of arrays of strings specifying a random Boggle board; you should also write your own test driver to pass it a known board. In any case, in order to simplify your logic (and your life), you should design a class (or classes) that you can use to represent the layout of the letters on the Boggle board. One good idea is to design a representation that basically treats the board as a graph, and then think of the search for all possible words in the graph as a search for all acyclic simple paths. Since a Boggle board is a graph with a quite specific topology, however, a completely general graph representation may not be the best approach. If you represent the board as a 2D array, you can follow the links of the graph by adding to and subtracting from the current position, i.e. `A[i+1][j-1]`. All supporting classes need to be declared in your `boggleutil.h` file, and defined in your `boggleutil.cpp` file.

2. Implement `isOnBoard()`: Now you will test your talent for writing and debugging code to verify that the user's words can actually be formed from dice on the board. Remember that a valid word must correspond to an acyclic simple path according to the neighborhood relation on the dice. You should search the board recursively, trying to find a legal formation of the user's word. This recursion can be made what you might call a "fail-fast" recursion: as soon as you realize you can't form the word along a path, you can immediately backtrack or move on to the next untried path start point. Reject any word that cannot be formed from the dice currently on the board.

3. Design a Lexicon data structure: The `buildLexicon()` method will be passed a vector of strings to populate your player's lexicon. Your lexicon should be implemented using a data structure that will allow you quickly to determine if a string is in the lexicon or not. In addition, you may find it useful for the lexicon to return additional information in case of a failed lookup; or you may find it useful for the lexicon itself to direct search of the game board, instead of the game board directing search of the lexicon. Your knowledge of tries and efficient sorted structures should help immensely here.

4. Implement `getAllValidWords()`: Now it's time to implement the world-class computer Boggle player. The computer should easily trounce a human player; it can very quickly learn more words than any human knows, and it can very quickly and perfectly check them against the board. The trick is to program the machine to do that. Part of the problem is to have a good word list to start with (the file "boglex.txt" contains such a list and is provided in your Github repository, though you may want to test on a shorter list to start). But the challenging part is implementing data structures and algorithms to make the computer's search efficient. It is easy to get entangled in the recursive decomposition and you should

think carefully about how to proceed. Bear in mind that you can significantly speed up this algorithm using "pruning" by recognizing when you are going down a dead end, abandoning it, and backtracking to where there are still live alternatives. For example, if you have built a path starting with "zx", you could use an `isPrefix()` function implemented on your lexicon to determine that there are no words in English beginning with that prefix. If you miss this optimization, you may find yourself taking long coffee breaks while the computer is busy checking out non-existent words like "zxgub" and "zxaep", etc. Or, as another strategy, if you have traversed your lexicon starting with "ab" and your game board graph is able to tell you there are no paths on the board starting with those letters, you know you don't have to even consider any lexicon words that have "ab" as prefix.

Keep in mind that in order to implement the `getAllValidWords()` and `isOnBoard()` functions, there are two types of search necessary. For the former, you are searching for all words that are on the board and in the lexicon and of the minimum length, while for the latter, you search for a specific word on the board and stop as soon as you find an occurrence of it. It is possible to implement the former in part by using multiple calls to the latter, though it is not the only, or necessarily the best, approach.

The files in your Github repository define a GUI application that you can use as a nice interface when testing some things about your BogglePlayer. This GUI application uses the Qt libraries as installed on ieng6, but they are available for many platforms; see http://qt.digia.com.

Work plan

We suggest that you tackle the board representation first, and verify the correctness of `isOnBoard()` using test cases that you can verify by hand. At this point you'll populate the board with symbols of your own choosing. Once this is done, you can then test `isOnbBoard()`; choose board configuration that you can verify by hand. Next, turn the lights on `buildLexicon()` and then test `getAllValidWords()`. At any time you may add the code needed to interface with the GUI so that you verify the above capabilities using a pleasing visual display. However, you should always retain the ability to work without the display, as it can be convenient when testing for bugs.

There are some topics that will be discussed this coming week, including ternary tries, but there is a lot you can do before reaching that point. We suggest you set milestones for when you'll complete each item, so that you can chart your progress and make adjustments as needed. This assignment will take time and you should not wait until the 2nd week to begin!

A note about using the GUI when working remotely

To work remotely, you'll need to use the "tunneling" capabilty of `ssh`. If you are connecting to `ieng6` from a Linux platform using `ssh`, use the `-X` option:

```
$ ssh -X yourlogin@ieng6.ucsd.edu
```

If you're working remotely on any other system, you can always use VNCgnome to enable X11 forwarding, instructions for which can be found here.

Building the Code

After you have created your `boggleplayer.h, boggleplayer.cpp, boggleutil.h, boggleutil.cpp` files, run the following commands, which set you up to use the `Qt` GUI capability.

1. On the ieng6 machine, type the following command in order to change your working environment to the one configured for CSE 100.
   `cs100e`
   This step should relieve you of errors of the form "cannot open file :“QtWidgets/QApplication”: No such file or directory".

2. Create a header file `ui_mainwindow.h`:

   ```
   uic mainwindow.ui -o ui_mainwindow.h
   ```

3. Create a Qt project file `P4.pro` (or in general, <your-working-directory>.pro):

   ```
   qmake -project
   ```

4. Edit `P4.pro` (or <your-woking-directory>.pro), adding these two lines at the beginning:

   ```
   QMAKE_CXXFLAGS += -std=c++11
   QT += core gui widgets
   ```

   and also remove the reference to `bogtest.cpp` from the SOURCES list.

5. Run `qmake` to create a Makefile with appropriate targets.

From this point on, you can just run

```
make
```

to build or re-build the GUI application as an executable named after your current working directory. Note however that your BogglePlayer class must have all the required functionality *without* the GUI code present. Also keep in mind that the GUI application by default only deals with square Boggle boards, and official Boggle dice, while your BogglePlayer needs to be able to handle any dimension board, and any strings on the die faces.

The executable `boggleref` in your Github repository uses a basically correct, but not very efficient, solution to the assignment. You can run the executable by typing

```
./boggleref <lexicon-file-name>
```

or just `./boggleref` in which case it will pick up boglex.txt as the default lexicon file.

Turning in your assignment

When you have completed the checkpoint requirements or the whole assignment, you must submit it. You will do this by pushing your changes to Github. Follow these steps (which are more or less the same as the steps from previous assignments).

1. Create the files that are required to be submitted in your working directory.
2. When you want to make a submission, run a '`git status`' to see what files have been changed.
3. Use '`git add <files>`' to stage certain files for a commit, use '`git add .`' to add all files in the working directory to the commit (hopefully you will not be doing this at this late stage--all files should already be in your repository!)
4. Use `git commit -m "FINAL SUBMISSION"` for your final submission (or `git commit -m "CHECKPOINT SUBMISSION"` for checkpoint submission) to commit the changes to your repository and indicate that you are done and this is your final submission.
5. To push your changes onto your repository on Github run '`git push origin master`' Don't forget this push! You must do this to get credit.
6. **As an added safety check, you should log on to Github on the web and verify that your push went through.**

A couple of notes about this process:

- **WATCH OUT FOR (AND RESOLVE) MERGE CONFLICTS AT YOUR FINAL PUSH!!** If you have been developing on multiple computers and NOT syncing with github between switching computers, you will likely get merge conflicts. This happens because your code from one repo is out of sync with your code from another and github doesn't know how to merge the files. You will get these conflicts "at the last minute," because git must sync the code in the repo you're trying to push with the code in the repo you're trying to push to. MERGE CONFLICTS WILL CAUSE YOUR CODE NOT TO COMPILE. So even if your code was working perfectly just before you attempted your final push, these conflicts will BREAK YOUR CODE. It is your responsibility to recognize when you have them (it's not hard--you get a big warning), and to FIX THEM. Please leave yourself enough time to do this.

- If you are submitting with a partner, just push to the joint repository on behalf of both partners. That's your joint submission.
- Use the commit message `FINAL SUBMISSION` for your final submission (or `CHECKPOINT SUBMISSION` for checkpoint submission) in ONLY ONE of your individual and partner repositories. You shouldn't be using both anyway, but if you happen to have pushed changes to both, just make sure only one contains the `FINAL SUBMISSION` (or `CHECKPOINT SUBMISSION`) commit message.
- If you decide you want to resubmit your assignment before the deadline, just do another commit with the message `FINAL SUBMISSION` (or `CHECKPOINT SUBMISSION`) (**and don't forget to push to Github!!**). We will grade the last commit before the deadline (or the slip day deadline if you use slip day(s)) that is tagged as the `FINAL SUBMISSION` (or `CHECKPOINT SUBMISSION`).
- If you submit the assignment within 24 hours after the deadline you will be charged a slip day. If it is more than 24 but within 48 hours, you will be charged 2 slip days. If you are out of slip days, or after 48 hours, we'll roll back to the last commit tagged as final that was submitted before the deadline.

Grading

There are 25 possible points on the assignment. If your solution files do not compile and link error-free, you will receive 0 points for this assignment. We will compile your code by unbundling your files, copying in the `baseboggleplayer.h` interface definition file, and running the commands

```
g++ -std=c++11 -O2 -c boggleutil.cpp

g++ -std=c++11 -O2 -c boggleplayer.cpp
```

and linking them with a test driver that expects your BogglePlayer class to satisfy the interface specifications in this README, using the environment on ieng6. We will test your code for conformance to the specification given above (i.e. not necessarily, and possibly not at all, in conjunction with the Boggle GUI application.) To get full credit, design, style and comments count.

You will get 0 points for functionality if your code does not compile. THIS INCLUDES CASES OF NON-COMPILATION DUE TO MERGE CONFLICTS. You MUST ensure that you resolve all of your merge conflicts and that your code compiles and runs correctly IMMEDIATELY BEFORE you push to github.

The grading criteria are as follows.

- 5 points for the implementation of `isOnBoard()` submitted by the checkpoint deadline (accuracy).
- 3 points for the implementation of `isOnBoard()` submitted by the final deadline (accuracy).
- 10 points for the implementation of `getAllValidWords()` (speed and accuracy).
  You've been provided with a 50,000 word lexicon in `lex.txt` in your repo and board configuration specified in `brd.txt` (the format is explained in `README_brd`). There are 5,450 words of length 2 or more with this lexicon-board pair. To get any credit for speed, your `getAllValidWords()` function should be able to find those 5450 words in less than 10 seconds (this is the amount taken by the reference solution), running on ieng6-201 or ieng6-202. The points for speed will be awarded based on the margin by which your implementation can beat the reference solution (more than twice as fast = 3 points / only a little faster but not twice as fast = 2 points / about as fast = 1 point / slower than the reference solution = 0 points). Also, note that we may test your program for efficiency against other lexicon and board files,

which is why you should make sure that your solution beats the reference solution for at least the above mentioned files by a good margin.

NOTE: brd.txt is just one example of how a boggle board can be represented. Your program should be independent of the format of the file used to represent the boggle board, i.e., your functions should not be dealing with files. The board and lexicon files will be processed by the calling `main()` method.

- 3 points for your implementation of the remaining methods.
- 4 points for overall design and style.

Note that the reference solution contains a "slow" implementation of the lexicon which simply uses a C++ set. You can use this solution to see the correct behavior of the method, and to run a solution which does not meet the speed criteria.

Comments

You do not have permission to add comments.