Project 5 Report

Chris Getz
Danny Chen
11/5/13

1 Overview

This project breaks down the game of Boggle into 3 distinct parts and provides the BoggleGame and BoggleDictionary interfaces. The key focus of this assignment is on proper data storage and searching, so as to minimize memory usage and asymptotic runtime.

1.1 GameDictionary

The GameDictionary is required to read in and store large amounts of words in an iterable data structure, and implements the BoggleDictionary interface. It creates a dictionary from a file of lexicographically ordered words, and must store them in structure that can easily be searched for words (at most O(log(n)) lookup time). It must also return if a string is a prefix of a word, and if a string is a word contained in the dictionary.

1.2 GameManager

The GameManager holds the information about the Boggle board, and provides the mechanics to play Boggle. It can create and reset a Boggle game, as well as load a certain board. It allows players to choose words, and determines if they are in both the current board and the dictionary, and determines the score if they are. Finally it can search through the entire board and find all the words they players missed, and can do so by checking all the combinations of characters on the board or by checking to see if any the words in the dictionary are on the board.

1.3 User Interface

The UI for Boggle was left almost completely to the students discretion. It only needs to call the GameManger to make a new game, display the board, get player input, and be able to start the game over. This may be implemented in any way and any interface, graphical or text, that the students thought suitable.

2 Solution

2.1 GameDictionary

Our implementation of GameDictionary uses each instance of GameDictionary as a node on a Trie that stores a character. Each node has an array to store up to 26 children, a parent, the character it stores, and if it is a word or not.

2.1.1 Searching

This implementation allows our search to be O(L) where L is the length of the longest word, which is normally much more efficient than anything based off of the number of words in

the dictionary. This is because the number of words grows much more rapidly in most cases than the length of the word. The tradeoff is that if the dictionary is a small number of very long words, our dictionary is much slower than a search based off the number of words.

2.1.2 LoadDictionary

Our loadDictionary method reads in each line and goes through character by character adding in any nodes that do not already exist, and when it comes to the end of the line it sets that node to be a word. We check that the line is all alphabetical letters because other characters will give an out of bounds exception. Because each node has an array of 26 pointers we will always have a place to add our character, and can create all possible words. In the original GameDictionary we also keep a pointer to the lastNode, which is updated after each line. This works because the dictionary file is in lexicographical order.

2.1.3 Prefix and Word

Each node has a boolean which stores if that node represents the end of a word or not. To determine if a string is a word, we simply traverse down the trie using the characters as indices and return if the node we end at is a word. If the next node does not exist at any time we know that the branch has never been loaded, and is not a word. Similarly we only need to check if the ending node exists to check if it is a prefix. If it has been created then it or a word that follows it's path exists in the dictionary, and that string is a prefix.

2.1.4 Iterator

Traversal of a tree is normally implemented recursively. However, our traversal is dispersed among multiple invocations of next(), so instead we used a stack to simulate recursion. We saved the current node in the iterator, and pushed nodes onto the stack as we went down the tree, and popped them back off as we went back up. We traversed the tree in order, which returns all the words in lexicographic order.

2.2 GameManager

Our GameManager stores the data in standard java data structures, and implements the BoggleGame interface to access the data. We use arrays for the board and player scores, and use a hashset to store the guessed words.

2.2.1 New Game & Set Game

After checking that the arguments are of suitable values we read in cubes and randomly place them, as well as initialize the arrays storing player and game data. When reading in cubes we ignore lines with less than 6 characters (as these are not cubes) and only choose from the first 6 characters on the line. We also check that the characters are alphabetical, and ignore lines that are not. Then we check that we have enough cubes to fill the board, and if we do then we randomly choose a cube and randomly choose a face of the cube, filling the board from left to right, top to bottom.

In Set Game we simply reset all of the data structures and set our board to be equal to

the one passed in to the method.

2.2.2 Add Word & Contains

Our addWord checks initial conditions that are quick, then checks that it is actually a word, then finally calls the recursive method Contains which checks that the board has the word on it. We check in this order so that we can avoid running the costly recursive search on the board all times that we can. Checking that it is a word only takes O(L) where L is the length of the string (see 2.1.1). The recursive method Contains returns the list of points that describes the location of the word, or null if it is not on the board.

Contains is overloaded so that we can have an initial check where we start from every square in the board where the character is equal to the first character in the string. We then start the recursion, setting the starting letter to a capital to denote that it has been used already.

We immediately check for the base case of a string of length 1, which means that we have found the word. We start a new list and add in the point, and return the list.

If it is not the base case, we set the current square to uppercase. we check every square around the current one, and after checking that it is valid (in bounds, not used, and == next char) we recurse down, and store the result in a temporary list. If it is null, we know that that path has failed. If all paths fail, then we return the character to lowercase and we return null. If any path is not null, we know that a path has been found to contain the word, and so we change the current character back to lowercase, add the current point to the list, and return it.

2.2.3 GetAllWords

Our get all words uses the dictSearch and boardSearch helper methods to maintain readability.

DictSearch creates a new iterator that goes through the dictionary and, after checking that the word has not been used, and is length >4, calls our GameManager.contains (2.2.2) to check if that word is on the board. It does this until there are no more words in the iterator.

BoardSearch utilizes a recursive method that is very similar to our contains method to do the work. This method is called crawler. It calls crawler starting on every square in the board.

At the start of the board it sets the character to uppercase to show it has been used already. Crawler keeps a path of the characters that it has traveled to get to the current point. If the current path is > 4 and a word and has not been found already it is added to the found words. It then checks each square around it, checking if the path plus that character is a prefix and is valid (in bounds, not used). If it is, then it calls itself with the updated path and location. After all paths are traveled the method simply ends, and BoardSearch calls crawler on the next space.

2.3 User Interface

Our user interface utilizes unix-based terminal for our I/O system, and uses ANSI escape sequences to format output. Examples of this are clearing the terminal, moving the cursor to a specific location, and coloring output.

Our interface has player 1 guess all of the words they can, then player 2 guess all of the words they can, and so on until all players end their turns. Then it calls the getAllWords to print out the words that were not found. It prints out the board, the players scores next to it, and then takes user input underneath the board. We used /end as the string to end a turn.

2.4 Assumptions

We assumed that people do not want boards less than 2x2. We assume that there are only 26 characters case insensitive. We ignore words/cubes with non-alphabetical characters in the line. We assume that cubes are supposed to be 6 sided (and ignore those that are smaller. We simply treat larger ones as if they only had the first 6 characters).

We are assuming that the client's terminal supports ANSI escape sequences. The Win32 console does not do this. The UI also makes the assumption that there are not enough players to make the scores wrap to the next line and clutter stdout. For size considerations, we do not expect the board to ever exceed a size of 36.

3 Testing

3.1 Peer Review

The peer review process was very useful in exposing misunderstandings with the specs for the project, as well as revealing several bugs due to corner cases that we did not think of. While reviewing others code we discovered things that were wrong with their code and while trying to figure out what the problem was, discovered issues with our own code. Overall testing code without having access to the class files forced us to really use the interface and automated testing instead of internally altering the file.

3.1.1 Ranking

Best- Group 4:

Their tests were the most extensive, but were also well organized and clearly all useful. They tested many of the edge cases that we did not think of, and all of the results were short and descriptive.

Middle- Group 1 & Group 2

Both groups provided fairly thorough tests which covered important cases. The results were well described, but several contradicted those of other groups and our own tests. Despite this they both proved useful in identifying errors and bugs in the code.

Worst- Group 3

Despite having a large range of tests which were explained, the only thing in the results section was that there was various errors regarding tests with addWord, and they thought it had to do with using lowercase letters. Not helpful in pinpointing anything that could be wrong.

3.1.2 Our Own Tests

Edge Cases - these usually involved file i/o and user input. We made sure that our solution could handle any possible user input, even if they were not English alphabetical characters. Other strange corner cases included newlines in the dictionary/cube files, improperly formatted text, or non-existent files.

Program Correctness - We had to be sure that our dictionary would not be modified after it loaded from a file. We basically created multiple iterators that would simultaneously iterate through the dictionary and check that every complete iteration matched each other. It was difficult to verify if getAllWords() in GameManager was correct, but verifying addWord(), getScore(), getLastAddedWord() were tested through the GUI. We predominantly used blackbox testing because inspecting the dictionary directly is unfeasible, and the GameManager could be inspected by the methods defined by the interface.

3.2 Revised solution

Our revised solution simply checked for more initial conditions when calling many of our methods. The overall structure and implementation of the solution was unchanged.

4 Paired programming

Once again paired programming helped to reduce bugs and take the task of coding the solutions off the shoulders of just one person. We are continuing to work well together and be productive.

4.1 Programming Log

10/24/13	Chris -	50 minutes driving
	Danny -	60 minutes driving
	Chris -	30 minutes driving
10/29/13	Danny -	45 minutes driving
	Chris -	30 minutes driving
	Danny -	45 minutes driving
	Chris -	45 minutes driving
10/30/13	Chris -	45 minutes driving
	Danny -	55 minutes driving
	Chris -	30 minutes driving
	Danny -	40 minutes driving
10/1/13	Chris -	120 minutes individual testing
	Danny -	120 minutes individual testing
10/2/13	Chris & Danny-	165 minutes testing and reviewing

10/4/13	Chris & Danny - 30 minutes bug fixing
	Chris & Danny - 150 minutes writing report