

Assignment #5

Purpose

The purpose of this assignment was to code a simplified version of the maybe-popular game, Boggle™. To do this, we had to write a generic dictionary class, a representation of a Boggle™ game, and a G.U.I., with which users can interface with and therefore possibly enjoy the game of Boggle™.

The meta-purpose of writing Boggle™ was to make us conscious of the effects our choice of algorithms can have on the asymptotic runtime and memory usage of programs.

Implementation

GameDictionary

The GameDictionary class is simply a representation of a normal dictionary, sans definitions, pronunciations, and word origins; effectively, it is not much of a dictionary at all. Despite these shortcomings, GameDictionary sports the ability to search for both words and prefixes and is expected to do so both quickly and without significant memory usage.

Initially, we implemented GameDictionary with a sorted ArrayList, allowing us to perform a binary search on the list when trying to find a given word or prefix. This gave us a worst case runtime of $O(P \log(n))$, where n is the number of words in the dictionary and P is the length of the word or prefix being found. After being given the requirement that the isPrefix method run in $O(P)$ time or better, we performed a significant overhaul of the GameDictionary class.

In a word, letters are followed and/or preceded by other letters unless the word only has one letter. In a linked list form, this structure would be represented by the first letter pointing to the second letter, the second letter pointing to the third letter, and so on. Thus, a single word can be exactly modeled by a linked list.

However, when dealing with multiple words, a single linked list is insufficient. Consider the words “ask” and “asp.” Because they share a two letter prefix, these words could be modeled by mapping the prefix to a linked list of the suffixes. Thus, the prefix “as” could be mapped to the linked list [$k \rightarrow p$]. Appending an element from the suffix list to the prefix results in the creation of valid words.

If the word “ad” were also included in this representation, the prefix “as” as a whole could not be used. However, mapping the prefix ‘a’ to the linked list [$d \rightarrow s$] and mapping ‘s’ to [$k \rightarrow p$] would satisfactorily represent this collection of words. This is because a path taken from the root letter ‘a’ to any letter not mapped to a list (in this example, the letters ‘d’, ‘k’, and ‘p’) results in the creation of a valid word.

Generalizing this allows for the satisfactory storage of the words in a dictionary. We implemented this with a pseudo - binary tree. For each node, the left children are those letters in the same linked list as the node, pointed at by a shared prefix. Thus, the prefix can be found by traversing from the root, inclusive, to the current linked list, exclusive. The right child is the head of a linked list of suffixes for the current node.

Consider some new words, “ask”, “asp”, “bee”, and “hellojeffdiamond”. [$a \rightarrow b \rightarrow h$] is the first linked list. In tree form, ‘a’ would be the root, its left child would be ‘b’,

and 'h' would be the left child of 'b'. The right child of 'a' is 's', and 's' has the right child ['k' → 'p'], but no siblings. This is because no other letter follows 'a'. Note that, in tree form, ['k' → 'p'] would become a subtree with 'k' as the root, null as its right child, and 'p' as its left. "bee" and "hellojeffdiamond" would just be linked lists.

Now that you are half-asleep you will be amazed at the glory of our dictionary implementation and give us a high grade. Our implementation allows for low memory usage and speedy operations. The isPrefix() method functions as follows: starting with the prefix's first letter, iterate through the linked list of suffixes which follow. Repeat this process for every letter of the suffix, that is, until the letter is the last character of the suffix. If the end is reached then the prefix exists. The contains() method acts in a similar manner, however, upon reaching the final character, a check must be done to see if that letter is the end of a word. This is done with the isEnd() method.

Sorry, Microsoft Word is putting weird spaces everywhere. That's what we get for using M\$ products.

The Big-Oh runtime for isPrefix() and contains() are both $O(P)$, where P is the length of the input String. This can be seen because when searching for a word, the algorithm can be analyzed as such: first, try to find the first character. This involves finding the character in a linked list, which at worst takes 26 operations, one for each letter of a full list. If the character is found, then this process is repeated for every letter of the input String. Therefore, at worst case, $26 * P$ comparisons are made, resulting in $O(P)$. As you can see, though, this only occurs when your word is "zzzz...", and such a word is just plain stupid ☺. Like Microsoft.

Why is this double spacing?

Concerning the memory usage of our algorithm, the first test was obvious – ensure we weren't using more memory than a simple list of strings. An ArrayList implementation used more than our tree implementation with the given dictionary. This may vary by word sets, but for large sets such as the one provided our algorithm should achieve relative success.

The methods resetIterator() and nextWord() function as specified. When resetIterator() is called, the iterator references the first character of the first, or topmost, linked list. Calls to nextWord() return the word currently formed and then advance the iterator. Its first attempt will be to move to the first child of the current character. If this can be done, that child will be added to the return String. Next, it must be checked if this character is the end of a word. If it is, the next word has been found; otherwise, the iterator continues to reference more children until the end of a word is found. Note that this will always find an actual word, as all leaves are the ends of words.

If a child cannot be found, then the current character is removed from the string and an attempt is made to replace it with the next character in the linked list, if one exists. That is, if the current character has a sibling, another character that is a suffix to the current character's prefix, it is selected and added to the String. If that character does not represent the end of a word, the child selection process continues.

Now that you are fully asleep, you will be even more amazed or something like that. In the event that your current character cannot advance to another sibling and cannot go to a child, it must recurse backwards through the tree structure until it reaches an ancestor that

has another child to choose or a sibling to choose. If there is no child or sibling to choose while recursing through ancestors, then you have reached the true root of the tree, which symbolizes the end of the dictionary.

This entire process is a convoluted inorder traversal which takes one step at a time. In order to keep track of the current String, our CharStack class is used. When the iterator references a child of its previous node, the child's character value is pushed on the stack. When a parent or sibling is selected, the previous node's character value is popped, as it is no longer in the String found from traversing from the root to the current node.

GameManager

GameManager simulates a Boggle™ game. A positive integral number of players can participate in the game. As per Boggle™ specification, these players find words on the board and are awarded points for correct guesses. A single player cannot receive credit for the same word more than once, but the implementation says nothing about multiple players guessing the same word. As a result, we stored each player's words in a Set and awarded points for guesses if the word was not already in their Set.

We implemented the getLocation method, which tests whether a given word was on the board. To do this, we searched the board for the first character of the word. From there, the program searches the immediate 3x3 square for the second letter. The same process is repeated until either the word is found or the fact that the word cannot be found is determined.

Since the same character from the board cannot be used multiple times for the same word (although duplicate characters in different spots can be used), we created a boolean array with the same dimensions as the board. Each index of the array corresponds to the index of the board with the same coordinates and is set to true if the character has been used and is false otherwise.

Another method required by the BoggleGame interface is the getAllWords() method. This method finds every word from the board contained in the BoggleDictionary. There are two approaches that require implementation: the dictionary-driven search, in which every word in the dictionary is tested to see whether it is located on the board, and the board-driven search, in which every valid combination of characters from the board is tested to see whether it is a part of the dictionary. These determine the same information, but have tradeoffs depending primarily on the board and dictionary size.

The dictionary-driven search was relatively easy to implement. Since the BoggleDictionary interface allows the words of the dictionary to be iterated through, we were able to iterate through every word and use the getLocation method to test whether it could be found on the board. The runtime of this method, falsely assuming that the getLocation is of constant runtime, is $O(n)$, where n is the number of words in the dictionary.

The board-driven search, while complex, bears a spiritual brotherhood to the getLocation method. For each location on the board, it iterates through the surrounding 3x3 square of characters. It then tests whether the combination of characters is a prefix in the dictionary. If it is a prefix, it tests whether the combination of characters is a word in the dictionary. Then, it recursively iterates through this process, using as the new center the previously chosen character. If it is not a prefix, however, it cannot be a word in the dictionary, and the iteration progresses to the next 3x3 character. Again, since the same

character from the board cannot be used repeatedly, the same boolean array process from the getLocation method was used.

The trivial best case runtime of the board driven-search is $O(s^2)$, where s is the size of the board since it must at look at every location on the board at least once. The tradeoff between the two procedures is apparent: for large dictionary sizes, the dictionary-driven search is far too slow because it grows linearly based on the number of words in the dictionary. The board-driven search, on the other hand, is better for these large dictionary files because its running time is independent of the number of String in the dictionary. Additionally, for small boards, very few possibilities are generated in relation to the dictionary size, resulting in fewer words being compared.

Boggle

I, John Scott Wright, am ashamed to admit that the Boggle.java GUI was coded by myself. It is a simple one-player representation of Boggle (because the specification in the handout only refers to a single person) and was coded in haste.

Testing

Our testing was extreme and you should know about it, Jeff Diamond. We tested numerous times laxly by running Boggle and actually playing the game. Whenever we entered a valid word, it was found. Whenever we entered an invalid word, it was not found. In addition, every time we ran Boggle, we had the full contents of the getAllWords method printed twice: one for each search type. Since the command prompt's font is truetype, we were able to check to see that the last words were aligned, which indicated that there were most likely an equal number words with the same average length found in both searches, immediately giving support to the validity of our code.

In addition to this, we created a simple test suite called BoggleTestSuite.java (the name comes from the fact that it is a test suite for the Boggle™ implementation and was coded in the Java Programming Language). This tested for an invalid number of players, an invalid board size, a dictionary file with repetitions of the same word, and a dictionary file with words such as “a”, “aa”, “aaa”, “aaaa”, “aaaaa”, “aaaaaa”, “aaaaaaa”, “aaaaaaaa”, “aaaaaaaaa”, “aaaaaaaaaa”, and “aaaaaaaaaaa”. In addition, the cube file had one – sided cubes to validate that case. Finally, the test suite compared the results from getAllWords using the board-driven search and the dictionary-driven search. Every test was passed with flying colors.

Additionally, some kind, nameless individual posted a test case on the wiki which our program worked well with without errors.

Pair Programming Log

Wednesday, 25 October, 2006

- We break down and actually work on the design of the project for about an hour, as opposed to diving headfirst into the work and sorting out details along the way. This period has been renamed from “waste of time” phase to “time saving” phase.

Friday, 27 October, 2006

- Roughly three hours spent coding and thinking, with approximately equal time spent on the computer. During this time, GameManager was written and the original

GameDictionary was written. We also spent over an hour trying to find a way to make `isPrefix()` $O(1)$, i.e., to do the impossible. ☹

Tuesday, 26 October, 2006

- Six eye-blurring hours spent making trees and stacks and arguing over the correct syntax for this report (I know, you think we wasted that time). All previous code was overhauled for this change.

Wednesday, 27 October, 2006

- It's not procrastination, it's quality control! And we were working on other stuff. We tested the code, John finished the GUI (Colin doesn't know what that means), and everything worked, like the Ewok song in Star Wars: Episode VI. Estimated time until this project is late: 1 hour, 15 minutes.

Solo Time

- John coded most of the GUI on his own, which was good. Colin tried to derive the Big-Oh worst case runtime for the `getLocation()` method, but can't do math. I hate my partner.
 - Anonymous