Steven Montalbano

smm285@pitt.edu

Dr. Khattab

CS 1501 Algorithm Implementation

Project 1 – Write Up

October 4, 2019

Set Up and Data Structures

For task 1, where we had to implement the crossword logic using the MyDictionary class built on an ArrayList, the initial setup was rather straightforward and much of the game setup code could be based on the BoggleSolver class provided in Lab1. A 2-dimentional array of characters is created to contain the actual game board, as read in from the text file. Also, a second 2-dimentional array of Booleans was used in the case where the game board file contained a letter at a given space, meaning that it was final and could not be altered. When reading in the board, if a character exists in the file, then the Boolean at that same index is set to true and the char is appended to the board. After the initial board was initialized, the utility objects needed to generate the possible solutions had to be created. Two arrays of StringBuilders is initialized to be of length == to the size of the board, one to represent the characters for each row and column of the board. The English dictionary is read into a data structure of type DictInterface, allowing our same solve algorithm to work when that variable is instantiated with either the MyDictionary class built on top of the Java ArrayList class, or the De le Briandais Trie that was created for Task 2.

Debugging

The most difficult part of this project, for me, was figuring out how to debug the program effectively. To spare myself from typing ‘System.out.prinln()’ hundreds of times, I created a ‘debugPrint()’ method that accepts a string and depending on the value of a global variable DEBUG\_MODE, the print statements can be switched on and off. This made it very easy to debug my searchPrefix() and add() traversal algorithms. To debug my recursion and backtracking algorithms I could use the built-in Eclipse debugger, which allowed for my program to halt at certain breakpoints and allow me to inspect the value of all live objects and variables during the execution.

Game Logic Algorithm

To start generating possible solutions to the crossword board, two methods contained all the main logic. First, the ‘solve’ method was created which is the recursive method that iterates over the board and calls a validation method to check if every character in the alphabet are possible solutions. It accomplishes this by calling the ‘isValid’ helper method which performs a dictionary lookup on the contents of the StringBuilder that represents the current row and column checking to see if the last appended char makes the row and column hold a valid prefix, word, or a word and a prefix. The ‘isValid’ method is where most of the game logic takes place, handing the many corner cases that you can encounter when working through a complex puzzle that is a crossword.

Results

As seen in the table below, the DLB implementation of the DictInterface was significantly faster than the ArrayList implementation. In most cases, if the DLB finished within seconds, the MyDictionary finished in some minutes, where if the DLB finished in some minutes, then the MyDictionary would typically finish in some hours. This is due to the ArrayList having to perform a sequential search over the entire dictionary to search for prefixes and words. When using the DLB, the search algorithm is much more efficient being that it has a pruning heuristic built in. Speaking of pruning, to increase my performance regardless of which data structure held the dictionary, I implemented a simple if statement in the while loop that reads the dictionary text file into the given DictInterface being used. The if statement checks to see if the word on the current line of the Scanner is <= to the size of the crossword board. If true, the word is added to the dictionary, if not, then the word is not added because there is no case where it can be a possible solution. This made a significant improvement in performance of the MyDictionary class, and only a slight improvement in the DLB dictionary.

Asymptotic Analysis

For the analysis of the runtime, there are many factors that come into play, including the number of words in the dictionary (based on the dictionary pruning), the number of characters in a word, the size of the board, and the possible characters for the given location. For larger boards or words, the algorithm would require more backtracking to find a solution.

Outside of the game conditions, the state of the machine running the program could also factor into a reduced runtime. For example, I noticed in some cases that performance was slowed while running on battery power, or while other programs were running in the background.

The MyDictionary implementation’s searchPrefix() method runs in O(MN) time, where M is length of the key and N is the number of words in the pruned dictionary. The DLB implementation is much faster, a DLB traversal is run in O(M) time, with respect to the length of the word or prefix being searched for. The DLB’s structure saves an incredible amount of time verses the linear searching of the MyDictionary’s underlying ArrayList. The searchPrefix() methid is called from inside of the isValid() method, which gets called, at most, 26 times per space of the board. The amount of searchPrefix() calls being made to solve the puzzles scales exponentially once you factor in backtracking and revisiting spaces that were thought to be valid. This is why the DLB implementation is so much more efficient. Taking into account the varying sizes of the crossword boards, we can calculate a total run time. For the MyDictionary, I project an approximate runtime of O(26X \* NM+M), with X being the number of rows and columns in the board. Similarly, the approximate runtime of the DLB would be O(26X \* M+M), also with X being the number of rows and columns in the board.

Time for Finding All Solutions

|  |  |  |
| --- | --- | --- |
| Board File | MyDictionary (ArrayList) | DLB |
| Test3a.txt | 3:19 Minutes | 18.9 Seconds |
| Test3b.txt | 2 seconds | 232 Miliseconds |
| Test4a.txt | Did not finish, >3 hours est. | 23:08 Minutes |
| Test4c.txt | 5:58 Minutes | 10 Seconds |
| Test4f.txt | 778 Miliseconds | 266 Miliseconds |
| Test6a.txt | 45 Minutes | 3:26 Minutes |
| Test8c.txt | 23.517 minutes | 8.4 seconds |