Mohit Jain

CS 1501

Project 1: Crossword Solver

In Project 1, I developed an algorithm that allowed a user to retrieve a solution of any n by n crossword puzzle. At first glance, this project seemed rather difficult and tedious. It seemed like an almost never-ending mind game, but the recursive solution turned out to be a lot easier than I thought.

To describe my solution, I thought of multiple different ways to approach this problem. One way I thought of was the direct approach and to label each box with an x and a y coordinate. This would be used to determine which location we are currently on. While I was thinking about this solution, I thought of an even better solution. After looking at a board of 3 by 3 squares for a long time, I tried out to do some division and modular arithmetic. I labeled each box of the 3 by 3 with a number in row by row order from 0 to 8. Then I found out that if we computed the current location of the square we were in and divided it by 3, in this example, we would get the number of the row based on the integer division that Java uses. For example, let’s say the location we were currently on was 5, which would correspond to row 2 in a 3 by 3 square, we would divide 5 by 3 using integer division and get 2. As for columns, if we took 5 and mod that number by 3, we would 2, which correctly represents the number for the column of this location. As such using this small algorithm, I was easily able to search through the square. Throughout, the program, while I would try to fit a word into a row or a column, I did a brute force on each square in the board trying all possible letters from A through Z. Then I would use MyDictionary.java to do a searchPrefix() on the current StringBuilder. Based on the output of this method, I would have to decide if I needed to backtrack on the current square and try a new letter in the square or backtrack more than one block and try a new letter in the previous square. I keep doing this until I find a possible solution.

As for the implementation of the DLB, I am currently unable to get it to work and am not sure of what the problem is. I am positive that my add and searchPrefix methods are implemented in a seemingly correct form. In the add method, I append an end of character symbol to each string that gets added. This is used in the search prefix method to determine if a given word is a prefix or if it is a word in the dictionary itself.

As such since my DLB is unable to function properly, it is difficult to get a clear gauge for which implementation would perform better (the MyDictionary ArrayList implementation or the De La Brandis Trie implementation). Currently, my implementation with MyDictionary takes very little time if almost no time at all to solve a crossword. To discuss a small analysis of best case and worst case run times in both cases, I believe that regardless the DLB will triumph over the ArrayList implementation. The reason for this is because unlike the ArrayList implementation, the DLB implementation does not require searching through all the words using a nested for loop. In fact, it only takes in one for loop, so runtimes will most definitely. I believe that in the worst case since the dictionary is such a huge list of words, if a word is located at the very end of the DLB, but is located at the very first entry of the arraylist implementation, then the arraylist implementation will be faster as it will take longer for the DLB to search for the specified word than it would for the ArrayList implementation, but this is an extremely rare case, and in most cases, the word to search for will be within the middle of list, so therefore, searching using a DLB would be a more appropriate method.

To delve into a more concise discussion of comparing the MyDictionary implementation to the DLB implementation, in MyDictionary, data is stored at a much faster speed than the DLB because all it does is that this algorithm simply takes an ArrayList, and individual adds each and every given string to the list. To make searches a bit faster, it then sorts the list in such a way that it makes such that all strings that were added first remain on the top of the list, and that the last string remain on the bottom of the list. However, in the DLB, data is stored using a trie, so it is not stored at a very fast pace as it requires to repeatedly iterate through each character of the string and determine at which position the string be added. For example, if we were to be given two strings: “Shells” and “Seat,” We would store the word “shells” from in a downward or vertical fashion. However, when we store the word “seat,” we would notice that the first character is the letter ‘S.’ We would then proceed down one step and right another step and insert the character ‘e,’ according to the structure of the DLB trie. As you can probably, this is a lot more memory efficient because instead of storing each word into a list, we are using previous characters of other words that already have been stored to store new words. This is ingenious because it saves a lot of space in memory, and we can avoid repeated prefixes of words as well as determine a number of possible solutions of words based on their prefixes. With the ArrayList implementation we cannot do this as we are looking at each word individually. The DLB combines all the words in the dictionary into a huge, but compressed block making it faster to search for words. As I discussed earlier, the ArrayList implementation uses a nested for loop to search for words. As such the runtime for this search would be around O(n^2), whereas the DLB uses only for loop to search through the structure. As such the runtime for this search would be around O(n). Therefore this proves that overall, the DLB will have much faster search times than the ArrayList implementation. Even though it may take longer for the DLB to add a word into its structure, I believe that improving the cost of searching is far more important than improving the cost of adding words to a structure.