Joanne Chen

Professor John C. Ramirez

CS1501

5th June 2018

My idea to solve the crossword-filling problem is to associate a recursive call with a cell in crossword board by my solveWord(r, c, D) function. When solveWord(r, c, D) is called, this call is first responsible for checking what character is initially on the cell (r, c) of the crossword board from the test file. Consider a recursive call solveWord(r, c, D). If the character is a plus sign “+”, the cell is empty and this call has 26 possible letters (a~z) to put into the cell. This call will pick the ﬁrst letter that either composing a valid prefix or word with letters in previous row and column and put it into the cell. This letter should form a word if it is on the last cell of the row and/or column. Note that it may or may not lead to a final correct solution. Once it fills a possible letter, it would make a recursive call to solveWord(r, c+1, D) so that the next call will take care of cell at row r, column c+1. If the select letter does not lead to a solution, this call would pick the next possible letter. If this call runs out of a choice of letters, it means the choice picked by previous call does not work. Then, this call restores the plus sign and return false to previous call to let it pick a new possible letter.

Some cells may already have letters in the test file. If that is the case, the recursive call would check if this fixed letter forms a valid prefix or word with letters in previous row and column, depending on this cell’s location. If it does, simply call the next one. Otherwise return false to previous cell without editing this cell.

If the character at (r, c) is a minus sign, the cell is a block. There are some edge cases for a block cell. Most of them come up during my struggle along debugging. If the block cell is the first cell in the board (r = 0, c = 0), simply call the next cell as we have no previous letter to check. If the block is on the first row, we need to check if the left neighbor cell is a block. If it is, make a next call as the neighbor block has checked letters in previous cells and we don’t have to check second times. If it isn’t, check if previous letters in this row form a word. If they do, call the next one. Otherwise return false to previous cell. If the block is on the first column, we would only check if the top neighbor cell is a block. And the algorithm follows similarly as block on the first row. For block that is none of the above three cases, we would check if its left neighbor cell and top neighbor cell are block. If they both are blocks, simply call next. If one of them is block, check if the previous letters form a word. If neither of them is block, check if previous letters in both row r and column c form a word. We would handle solution print and runtime record once the last cell on the board is solved.

The table in the following page shows runtimes for DLB and MyDictionary finding out first solution toward various test files. To investigate these data closely, I create several charts. In Chart A, the runtime increases as board size grows for both DLB and MyDictionary. First, we fix the content of all test files unchanged (They are all plus signs). There is a significant growth in runtime for MyDictionary when the board size increases from 5 to 6. And after that, the runtime for MyDictionary to generate a solution becomes extremely high. The reason is that the path of searching grows longer and longer and we have to go through more levels to try more possibilities to find a letter satisfies the conditions. Another thing to notice on this chart is the rapid change in runtime for DLB as board size grows to 7. The runtime rapidly grows up and then slows down as the board size reaches 7. I consider the reason of that phenomenon is there are less words of length 7 than words of length 8 in the dictionary. Thus, number of words in dictionary is inversely proportional to the runtime.

On Chart B I fix the board size unchanged and look at these fixed letters’ and blocks’ locations. First notice that test4a are all plus signs, test4b contains 1 block, test4c has 4 fixed letters and 2 blocks, tets4d contains 1 fixed letter, test4e has 1 fixed letter and test4f has 7 fixed letters and 2 blocks. Comparing these data with runtime, we can see the runtime gets longer as more plus signs in the board for us to fill in. To look at these data more closely, if we compare MyDictionary’s runtime of test4c and test4f, which has 2 blocks in the same locations, we can see that runtime gets faster as more fixed words in the board. Another thing that catches my notice in this chart is both DLB and MyDictionary increase in runtime on test4e, which has fixed letter “x” at first row and forth column of the board. But if we compare test4e and test4d, we would find out that test4d also has the fixed letter “x”, while locates differently at the first cell of the board. The runtime of DLB and MyDictionary for test4d is very quick while for test4e is slower. That implies the location of fixed letter has influence on the runtime.

Generally, the DLB implementation of DictInterface achieves great improvement in runtime over MyDictionary. And such achievement is significant as board size gets large. MyDictionary implements ArrayList. Its searchPrefix() iterates through the list until the end or until the key is passed up while each time iterating the current String through the list character by character. That creates two for loops, which is O(n^2) in implementations. However, DLB applies linked list, which allows for linear-time O(n) search. Each node in DLB is uniform with two references, one for sibling and one for a single child. We walk the list down us to follow a child pointer if character matches and we walk right to a sibling if mismatch. For a key of length n, we have n levels to get to the end of the key and up to constant comparisons to find the character on each level. So in worst case we can end up to O(n) comparisons required for a search.

|  |  |  |
| --- | --- | --- |
|  | MyDictionary | DLB |
| test3a | 0 minutes 4 seconds | 0 minutes 5 seconds |
| test3b | 0 minutes 7 seconds | 0 minutes 4 seconds |
| test4a | 0 minutes 46 seconds | 0 minutes 4 seconds |
| tast4b | 0 minutes 24 seconds | 0 minutes 4 seconds |
| test4c | 0 minutes 37 seconds | 0 minutes 4 seconds |
| test4d | 0 minutes 5 seconds | 0 minutes 5 seconds |
| test4e | 59 minutes 50 seconds | 0 minutes 10 seconds |
| test4f | 0 minutes 13 seconds | 0 minutes 4 seconds |
| test5a | 0 minutes 35 seconds | 0 minutes 6 seconds |
| test6a | 3 hours without terminating | 3 minutes 55 seconds |
| test6b | 257 minutes 1 seconds | 0 minutes 28 seconds |
| test6c | 392 minutes 51 seconds | 0 minutes 35 seconds |
| test7a | Extremely Long | 200 minutes 7 seconds |
| test8a | Extremely Long | 100 minutes 8 seconds |
| test8b |  | 123 minutes 39 seconds |
| test8c | 16 minutes 22 seconds | 0 minutes 5 seconds |

\*Table: the time used for two searching classes (first row) to generate first solution (or no solution) matched with each test file.

\*Chart A: test files that contain only plus signs (x-axis) over runtime in second (y-axis).

\*Chart B: test files of same size containing different amount of fixed letters and blocks (x-axis) over runtime in second (y-axis).