Joanne Chen

Professor John C. Ramirez

CS1501

5th June 201Crossword puzzles are challenging games that usually takes the form of a square or a rectangular grid of white-and black-shaded squares. The game's goal is to fill the white squares with letters, forming words or phrases, by solving clues, which lead to the answers. Every word is read left-to-right. The answer words and phrases both across and down must be legal. The choice of a word in one direction would restrict the possibilities of words in the other direction.  The shaded squares are used to separate the words or phrases.

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, it 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 considered empty and this call has 26 possible letters (a~z) to put into the cell. This call will pick the ﬁrst letter that creating either 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 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. Take figure 1 as an example, we are in first row and last column. We first try letter “A”, which does not form a word with previous letters. Then we try “B”, which also does not form a word with previous letters. We would keep doing this. 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. In our example, now figure 1 transforms ro figure 2 as we find out that the next possible letter in row 1 and column 2 is “B”.

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. For example, figure 3 has fixed letter B at (1,2). We have to check if the letter we filled out in (1,1) and “B” forms a valid prefix. If the fixed letter does, simply make a call on the next cell Otherwise return false to previous cell without editing this cell. In our example, we have to find out next possible letters at (1,1) when the letter we filled in does not form a prefix with “B”.

If the character at (r, c) is a minus sign, the cell is a block. There are some edge cases for a block cell. I came up most of them when I debugged my code. . When 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. When the block is on the first row, we need to check if the left neighbor cell is a block. If it is, make next call since the left neighbor has checked letters in previous cells and we don’t have to check second times. If it isn’t, check if previous letters until this cell in this row form a word. For example, we need to check if the letters in (1,1) and (1,2) form a valid word when we reach (1,3) in figure 4. If they do, call the next one, which is cell (2,1). Otherwise return false to previous cell, which is cell (1,2). If the block is on the first column, we need to check if the top neighbor of this cell is a block. And the algorithm follows similarly as block on the first row. In figure 4, we have to check if letters in (1,1) and (2,1) form a valid word when we reach a block at (3,1). For block that is none of the above three cases, like the situation of figure 5, we need to check both its left neighbor cell and top neighbor cell. If they both are blocks, simply make next call. If one of them is block, check if previous letters form a word. If neither of them is block, check if previous letters from directions of row r and column c form a word. So in figure 5, we need to check letters in (1,2) and (2,1) separately. We would handle solution print and runtime record once the last cell on the board is solved.

|  |  |  |
| --- | --- | --- |
| + | + | + |
| + |  | + |
| + | + | + |

Figure 5

|  |  |  |
| --- | --- | --- |
| + | + |  |
| + | + | + |
|  | + | + |

Figure 4

|  |  |  |
| --- | --- | --- |
| + | B | + |
| + | + | + |
| + | + | + |

Figure 3

|  |  |  |
| --- | --- | --- |
| A | B | + |
| + | + | + |
| + | + | + |

Figure 2

|  |  |  |
| --- | --- | --- |
| A | A | + |
| + | + | + |
| + | + | + |

Figure 1

The table in the following page shows runtimes for DLB and MyDictionary to figure out first solution toward various test files. To investigate these data closely, I create two charts. From Chart A, we can see that runtime for both DLB and MyDictionary increase as board size grows . 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. That is because we have to iterate through the sorted arrayList until the end or until the key is passed up. Thus, number of words in MyDictionary is proportional to the runtime. 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. Since DLB trees allow a string to be tested as a word or as a prefix in the dictionary in time proportional to the length of the string. I consider the reason of this phenomenon is there are many words of length 8 than length 7 and length 9 in the dictionary.

On Chart B I fix the board size unchanged and try to find the relationship between runtime and 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 information with runtime, we conclude that runtime gets longer as there are more plus signs in the board for us to fill in. By comparing MyDictionary’s runtime of test4c and test4f, which have 2 blocks in the same locations while test4f has 7 fixed letters and test4c has just 4 fixed letters, we can see that runtime gets faster as more fixed words in the board. Another thing that catches my attention 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 with test4d, which also has a fixed letter “x” but locates in a different cell , we would find out that the runtime of DLB and MyDictionary for test4e is slower than runtime for test4d. That implies the location of fixed letter has an influence on the runtime.

Generally, the DLB implementation of DictInterface achieves a great improvement in runtime over MyDictionary. And such achievement is significant as the board size gets large. MyDictionary uses 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) implementations. However, DLB applies linked list. In our example, we have 256 ASCII characters possible in our "alphabet" and our key has length varied from 1 to N, which is the board size. In worst case we can have up to Θ(N) character comparisons required for a search and up to S comparisons to find the character on each level. .

|  |  |  |
| --- | --- | --- |
|  | 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).