Most people find crossword puzzles challenging and fun, but for a computer it merely comes down to time, processing speed, and the efficiency of one’s algorithm. For this assignment I created a crossword solving class called Crossword and an implementation of the Dictinterface called DLB. In conjunction with the provided DictTest, DictInterface, and MyDictionary classes, I was able to find solutions for several test crossword boards which ranged in size from 3x3 to 8x8.

The first part of this assignment was to find a single solution to a given crossword puzzle by creating a Crossword class. I broke this task into several pieces in order to find a solution. The first part was to create a searchable dictionary which I could then use to find words that fit into the puzzle board. The instructor provided an interface class, DictInterface, and a sorted array-based implementation of that interface, MyDictionary. The interface included an add() method to add new words to the dictionary, and an overloaded method called searchPrefix(), which searched for a given word in the array. Careful reading of those files along with the DictTest class showed how the 3 methods of the DictInterface were used and what they did. I then used the DictTest file as a template to start creating my Crossword class and finish the first step.

My next task was to simulate the crossword board. Valid boards are squares of cells between 3x3 and 8x8 cells in size. Each cell on the board consists of empty squares (‘+’), non-fillable white space (‘-‘), pre-filled in letters, or any combination of the three. To create my board data structure, I wrote a private inner class called Cell and created a two-dimensional array of Cell objects called board. Each Cell object consisted of a character field and several calculated fields which I used later on to improve the efficiency of my solving algorithm. The calculated fields describe the position of an individual cell within a substring and inform the search algorithm if that cell is at the beginning, middle, or end of a substring. Since these position properties would need to be assessed for every cell each time the algorithm visited the cell, I decided to preprocess the board when it was created in order to reduce the overhead of repeatedly making these checks later on. With the board and dictionary created, my final task for the first part of the assignment was to write the solving algorithm.

My solving algorithm used a recursive backtracking approach that incorporated two pruning optimizations in order to reduce the search space and improve efficiency. In order to efficiently add and remove letters from the board, I used two arrays of StringBuilder objects since they are mutable. One of the StringBuilder arrays was for the words read in the vertical direction (columns), and the other for the words read in the horizontal direction (rows). Each array had an index for each column or row respectively. The following algorithm describes the logic I used for solving the board:

1. Starting from the top left corner and proceeding from left to right, top to bottom, find the first empty cell (‘+’).
   1. If white space (‘-‘) or a prefilled in letter is encountered, append that character to the end of each StringBuilder object corresponding to that row and column on the board.
      1. If adding a prefilled in letter produces an invalid horizontal substring, return false and backtrack.
      2. If adding a prefilled in letter produces an invalid vertical substring, then one of the characters in the column above is the issue. Backtrack until the offending cell is located and try another character in that cell. This situation is similarly described on the assignment sheet with the “EQUX3” example. This optimization could result in one or more rows being removed and thus pruning the search space.
2. Once the first empty cell is found, try appending a single letter, starting with ‘a’, to the end of each StringBuilder object corresponding to that row and column on the board.
   1. Verify that the new substring is valid in both directions.
      1. If the append produces valid prefixes in both directions, then first verify if the cell is at the end of a substring in one or both directions.
         1. If either are true, check if there are more columns. If there are, go to the next column using a recursive call.
            1. If the cell is at the end of a row, verify that there are more rows. If there are, then go to the beginning of the next row using a recursive call.
            2. If there are no more columns or rows and the substring is also a valid word, then return true because the board is solved.
            3. If the cell is at the end of a substring but the substring is not a valid word, then backtrack.
         2. If the cell is somewhere in the middle of a board, but not at the end of a substring, make a recursive call and visit the cell in the next column.
      2. If the append doesn’t produce valid prefixes in both directions, then remove the character from both StringBuilders and try the next letter in the alphabet. Keep trying until condition (2.a.i) is met, or all twenty-six letters are used. If no appends produce valid substrings, do one of two things:
         1. First check that all twenty-six of the vertical substrings were invalid. If they were, then one of the characters in the column above is the issue. Backtrack until the offending cell is located and try another character in that cell. This situation is similarly described on the assignment sheet with the “EQUX3” example. This optimization could result in one or more rows being removed and thus pruning the search space.
         2. If only some, but not all of the vertical substring checks failed, then return false and backtrack one cell.
3. Continue appending and backtracking until a valid solution is found, or all possible solutions are tried, in which case the board has no solutions.

In order to further reduce the search space, I added one additional optimization that was similar to the “EQUX3” example from the assignment sheet. On boards with prefilled in letters, after the initial append and prefix checks in step 2a, the algorithm also checks if there is a prefilled in letter one row below (if it exists). If appending that prefilled in letter produces an invalid vertical substring, then the current letter is removed and the next letter in the alphabet is tried until a valid substring in both directions is found. This check can save from adding several more invalid characters to the board, and thus reduces the search space.

With my algorithm in place I set about running and verifying that my program was working using some of the provided test files. Some produced erroneous results at first, but most of that had to do with errors in my optimization logic which resulted in too much pruning. I ended up using print statements to print the board at various stages to see exactly where the extra pruning occurred. After several rewrites though, I was able to get the smaller boards working. At this point I proceeded to part 2 of the assignment and began implementing my own DictInterface class called DLB.

My DLB class created a de la Briandais (DLB) search tree using nodes consisting of three fields: a character field to store an individual letter of a word, and two nodes for a child and sibling list. In a DLB, nodes that are siblings of one another represent strings of the same length, that all have the same previous character in common. For example, consider a sibling list that starts with ‘B’, and consists of the characters ‘B’ → ‘C’ → ‘D’. If the parent node contains an ‘A’, then these four nodes represent the three strings “AB,” “AC,” and “AD.” The exception is the root list, which have no previous characters in common. To store a string in a DLB, each character is stored in a list whose level in the tree corresponds to the position of the character in the string. To indicate the end of a string, a non-alphabetical character is used, ‘^’. The termination character is stored as a child node of the last character in a given string. For example, the string “CLASS” has five characters, and would only take up six nodes. Five nodes for each character in the string, and one node for the terminator character. These six nodes would be connected from parent to child and create a tree with a height of six (or five if you count starting at zero). Adding another string, say “CLASSY”, to this DLB is done by linking the ‘^’ node to a new ‘Y’ node as a sibling (‘^’ → ‘Y’), and then making a second ‘^’ node the child of the ‘Y’ node at the bottom of the tree. Since words that have common starting prefixes are all represented as one single chain, DLBs are a dense data structure.

Searching a DLB for a given string is very similar to adding a new string. The search process starts by comparing the first character of the string to each node’s character in the root list until a match is found, or until the end of the root list is reached. If the first character is found, the next step is to search that node’s child list for the next character in the string in a similar manner. This continues until the last character of the string is found (followed by a terminating character), or if the list terminates (in which case the word is not found). DLBs exploit their density in order to make prefix and word searching very fast while not wasting storage space.

Unlike the DLB class, the MyDictionary (MD) class stores its data as individual String objects in a sorted ArrayList. While having the Strings indexed does allow for improved access time to individual Strings, this also means that the array length is equal to the number of words being stored in the dictionary. In comparison to the dense tree of nodes in the DLB, the MD requires vastly more storage space to construct since every word is stored in its own index. The next difference is the prefix searching methods in the MD. Unlike in the DLB, the MD’s searchPrefix() methods utilize sequential searching of the String objects for a given prefix. While on a surface level this may seem similar to what’s being done in the DLB, the biggest downside is that if a prefix isn’t found in the first String, it then has to search the String at the next index, once again starting at the beginning of that String until a match is found or until a difference between the search string and the String object is located.

Prefix searching in the MD in the worst case would compare each character of the search string to every String object - leading to k\*p comparisons, where k is the number of characters in the search string and p is the number of String objects in the ArrayList. MD prefix searching thus has a big O(kp). However, searching a DLB only takes a maximum of k\*S comparisons, where k is the number of characters in the search string and S is the size of the alphabet Σ. DLB searching has a big O(kS). Since Σ is usually twenty-six characters and most dictionaries contain more than 26 words, practically speaking, this gives the DLB a big advantage over the MD for finding both individual prefixes and full words. To put this in terms of numbers, there are 17, 271 words in the dict8.txt file, with an average word length of six characters. If one uses the MD class to search for a string of eight characters that isn’t in the dictionary, there could be somewhere between 51,813 and 138,168 comparisons, with an average somewhere near 103,626. The DLB class on the other hand would at most make about 216 (27\*8 = 216) comparisons, or between 0.2% and 0.42% of the number that the MD made in order to determine that the string was not contained in the dictionary.

With this knowledge of the expected runtimes, I began testing all of the provided crossword files with both the MD and DLB classes using my completed Crossword class. Table 1 shows the runtimes I achieved for the different files. Because of the large search space for the MD class, I had to terminate the majority of the boards with sizes over 5x5 (indicated by a T next to the runtime). The one notable exception was the 8c board which finished in only 26 seconds for the MD run. My second optimization in the Crossword solving algorithm and the fact that there were a lot of restrictions on the board to begin with, helped to give the MD a reasonable runtime compared with the DLB. The DLB was able to find all of the solutions for the puzzles with the exceptions of 6b and 8b, but according to the instructions, this was an expected result. Table 2 shows percent improvement in runtime of the DLB over the MD for all cases where both classes completed the search. On average, the DLB is 98.99% more efficient at searching. This was expected given that in the worst cases, the DLB only makes 0.2% the amount of comparisons that the MD search did.

Overall, this assignment showed the superiority of de la Briandais search trees as a data structure for storing and searching through large dictionaries of words. In all cases the DLB search outperformed the MD by a significant amount of time. The implication of this is that choosing the right data structure for a searching algorithm can have big performance impacts on a program. Possible improvement to my solve() method of the Crossword class could yield even faster runtimes and could be areas for improvement in the future.

Table 1. Runtime comparison of MyDictionary vs. DLB implementations of the DictInterface

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **MyDictionary** | **DLB 1st solution** | **DLB** | **# Solutions found** |
| test3a.txt | 0.048 sec | < 1 sec | 6.232 sec | 392,586 |
| test3b.txt | 0.356 sec | < 1 sec | 0.005 sec | 1 |
| test4a.txt | 7.104 sec | < 1 sec | 30.306 min | 1,621,736 |
| test4b.txt | 5.838 sec | < 1 sec | 2.050 hrs | 10,872,485 |
| test4c.txt | 0.040 sec | < 1 sec | 7.344 sec | 47,845 |
| test4d.txt | 0.103 sec | < 1 sec | 0.001 sec | 0 |
| test4e.txt | T 1.500 hrs | 7 sec | 7.305 sec | 0 |
| test4f.txt | 1.337 sec | < 1 sec | 0.010 sec | 1 |
| test5a.txt | 7.605 sec | < 1 sec | 7.15 0 hrs | 53,399 |
| test6a.txt | T 2.000 hrs | ~ 7 min | 6.03 0 hrs | 345 |
| test6b.txt | 22.800 min | < 1 sec | T 2.100 hrs | > 2,750,000 |
| test6c.txt | T 1.500 hrs | 67 sec | 67.029 sec | 0 |
| test7a.txt | T 2.000 hrs | 4.3 hrs | 4.283 hrs | 0 |
| test8a.txt | T 2.000 hrs | 2.2 hrs | 2.150 hrs | 0 |
| test8b.txt | T 2.000 hrs | 32 mins | T 2.266 hrs | >17 |
| test8c.txt | 26.002 sec | < 1 sec | 0.038 sec | 1 |

\* T before the runtime means that I terminated the process after running for so long

\*\* For tests 6b and 8b, since I terminated the process, more solutions could exist

Table 2. DLB improvement in runtime over MyDictionary for selected test puzzles

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **MD RT (sec)** | **DLB Average RT (sec)** | **% Improvement** |
| test3a.txt | 0.048000 | 0.000016 | 99.9669% |
| test3b.txt | 0.356000 | 0.005000 | 98.5955% |
| test4a.txt | 7.104000 | 0.001121 | 99.9842% |
| test4b.txt | 5.838000 | 0.000679 | 99.9884% |
| test4c.txt | 0.040000 | 0.000153 | 99.6163% |
| test4d.txt | 0.103000 | 0.001000 | 99.0291% |
| test4f.txt | 1.337000 | 0.010000 | 99.2521% |
| test5a.txt | 7.605000 | 0.482031 | 93.6617% |
| test6b.txt | 1368.000000 | 0.002749 | 99.9998% |
| test8c.txt | 26.002000 | 0.038000 | 99.8539% |

\*DLB average runtime calculated by taking total runtime divided by number of solutions found