Jack Anderson

Sherif Khattab

CS 1501

26 November, 2018

CS 1501 Post-Assignment 1 Analysis (Assignment 4 Re-write)

For Project 1, we were tasked with implementing a recursive backtracking algorithm in order to solve variable size crossword puzzles of up to 8x8 using a dictionary containing roughly 17000 words of up to length 8. This project highlights and tests our knowledge an ability to implement a De La Briandais Trie data structure, as well as pushing our understanding of backtracking recursion even further. In solving this problem, we were provided with a “MyDictionary” class which served essentially as a sorted ArrayList of the dictionary input that would be searched linearly to determine whether or a not a provided input was a prefix, a word, a word and a prefix, or non-existent in the dictionary. Initially we had to only find one solution for each provided crossword file. To tackle this problem, I thought back to previous backtracking algorithms I had worked with or written previously, including the 8 Queens and Semi-Magic Squares projects from CS 445, as well as a Boggle solving program from CS 401.

Beginning my implementation I started writing a method called “solve”, accepting 2 StringBuilder arrays “colStr” and “rowStr” which would hold each row and columns respective partial solution as the algorithm ran, two integers “r” and “c” which held the current position on the board the algorithm was working on, and finally the DictInterface “dict” which was either the MyDictionary or DLB depending on the type specified on the command line. The method itself would first check for the base case where the current row was the last index in the rowStr array and the current column was one past the last index in the colStr array. If not, the method then checked if the current column was past the valid range, and if so, it would reset c to 0 and increment the row by 1. These two if-checks are possible because of the way in which my recursion algorithm works. It progresses through the board beginning at rowStr[0] and colStr[0] and moving over 1 column after every successful placement of a letter or filled in space. Once the column index becomes invalid (past the last possible column) the algorithm realizes that the row must be incremented unless we have reached the end of the crossword puzzle, in which case we have a successful solution.

The actual algorithm itself for filling the board past the base case is rather complex. First, it checks what position we are at in the crossword board input. Spaces were either marked by a “+” indicating an open space, a letter, or a “-” indicating a filled in space. If the current space is not open, it checks whether it is a letter or “-”. If a letter, the character is appended to both StringBuilders and the algorithm checks whether we are at the end of a row, end of a column, the end of the board, or somewhere in the middle. If any of the first three are true, using the searchPrefix method of the respective DictInterface used, the StringBuilders must hold either a valid word or a valid word and prefix. For the last case, we make sure that the input is at least a valid prefix, however, if the next char in the board is a “-”, we allow the letter to be placed anyways if it makes a valid “word only”. The same algorithm is used in order to place “-” characters in the StringBuilders, however there is one extra consideration—two or more dashes placed back to back. To find this case, I loop through the current row or column and determine whether the last placed character was a dash. If so, a special case is entered which checks to make sure that the column or row not featuring the double dash is a word or word/prefix. This sort-of edge case was one of the last ones implemented and solved in my algorithm.

If the current board position was an open space, the algorithm is run much the same with a few key differences that allow for backtracking. To start, the remainder of the algorithm is encased in a for loop that sets each respective letter in the alphabet to the next letter to try (alphabet is built in main). From there the algorithm does essentially the same aforementioned searchPrefix checks, although that algorithm itself should be of note. To make sure that the correct part of a row or column was tested, I created a loop that goes through the current row or column. When a “-” is found, a “startIndex” integer initially set to 0 is set to the current position + 1. This means that we can then call searchPrefix with the extra parameters of a start index set to whatever the loop found, and an end index of the end of the respective StringBuilder. Characters before a dash in this case can be ignored as they would have already been tested and accepted by my algorithm previously. Under each recursive call is a call to delete the last-placed char on the current row and column in order to backtrack correctly. When all possible solutions have been exhausted, the algorithm backtracks itself, slowly clearing the StringBuilders until empty and back out to the original call in main. Then at the end, the total amount of solutions found is printed.

The DLB itself was implemented rather seamlessly and quickly, and I found it rightfully simpler than writing the main backtracking algorithm. The DLB itself is made of a Node “root” and a Boolean “isEmpty”. The latter is only set to true during initialization and used to set the root node when first adding to the DLB. The Node is itself a separate class within the DLB that has a sibling node, a child node, a char value, and a Boolean “isWord” which is set to true if the current node is in fact a word. Adding to the DLB utilizes a StringCharacterIterator that loops through the string and checks first for a sibling node that equals the current char before moving on to that sibling’s child node. Nodes are added if a respective sibling or child is not found, and the final node’s isWord is set to true. For searching, much the same is done in terms of traversing through the DLB, though the integer returned is the important difference. If the currentNode is ever null, 0 is returned as the searched StringBuilder (SB) is not contained within the DLB. If the full SB is found in the DLB, 3 checks are made. If the currentNode has a child and is not a word, it is a prefix and 1 is returned. If it does not have a child and is a word, then it is just a word and 2 is returned. Finally, if it has a child and is also a word, then it is a prefix and a word and 3 is returned.

In terms of debugging, I often utilized numerous print statements as a default for finding if the algorithm was making it to certain points or for showing me what a certain row or column’s current status was. This worked particularly well in many cases, especially for solving the double dash edge case. Ultimately though, I find that tracing through my code myself, writing things down and creating test cases in my head works the best for finding various issues in implementation as well as allowing me to fully visualize what I had written.

For performance comparison between the MyDictionary and DLB implementations of DictInterface, I ran many of the test crosswords and approximately recorded the time required to find the first possible solution in each case, placing them in the following table:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3A | 3B | 4A | 4B | 4C | 4D | 4E |
| MyDict | ~<1s | ~5s | ~10s | ~8s | ~1s | ~<1s | N/A |
| DLB | ~<1s | ~<1s | ~<1s | ~1s | ~<1s | ~<1s | ~7s |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 4F | 5A | 6A | 6B | 6C | 8B | 8C |
| MyDict | ~12s | ~5s | N/A | ~20-30m | N/A | N/A | ~50-60m |
| DLB | ~<1s | ~<1s | ~3m20s | ~4s | ~22s | ~1h30m | ~5s |

For cells marked N/A, I either determined that the MyDictionary would take too long or terminated them early because of the length that they ran without finding a solution yet. 7A and 8A are not included for similar reasons as they both would take extraordinary amounts of time to complete a solution for MyDictionary, and their comparisons do not add anything that was not already present and obvious in the table. Clearly from the results the DLB implementation is noticeably faster and superior to MyDictionary. In cases where the DLB took more than a few seconds, the MyDictionary implementation would take exponentially longer, even up to an entire hour in the case of test 8C. For 8B, which took the DLB 90 minutes to complete, one can only imagine the amount of time it would take for a MyDictionary solution to be found. In terms of asymptotic analysis, for a DLB if we have S characters in our alphabet (26 here) and our key contains K characters, in an average case we will most likely see a run time of Θ(K) as most levels in the DLB will have fewer and fewer siblings the farther we get from the top. Absolute worst case we may see Θ(KS) comparisons. Again though, this would only occur if our key was something similar to “zzzzzzzz” where every level had S number of siblings to traverse through. Regarding the MyDictionary implementation, we are looking at W words in the list/dictionary (over 17000 in our dictionary used), and a key with K characters. In the absolute worst case we may see a run time of Θ(KW), significantly greater than the worst case provided in the DLB. MyDictionary’s implementation of looking through every word in the dictionary before finding a match is incredibly archaic and naive and results in the disappointing and long run times featured in the table above. Of course, for both, run-time gets increasingly worse as crossword size increases and with fewer restrictions placed in the board (i.e. 1 solution of 8B with the DLB took roughly 90 minutes)

The compositions of each implementation, as briefly discussed previously, are quite different from each other in drastically important ways to the overall efficiency and run time of each. MyDictionary features an ArrayList to store each new input which is routinely sorted if a string is added out of order. The constant sorted-nature of this list helps to increase performance times and a key will repeatedly reject a word in the list until its characters start to match, allowing for faster traversal through the list comparatively. To search, MyDictionary linearly traverses through its ArrayList comparing the key to each word. As a pruning case for this, if the current character in the key is less than the current in the word then the search can be stopped prematurely because of its sorted nature. This implementation of comparing the key to every word until finding an identical match of a word or at least a prefix match is incredibly time consuming and inefficient. The DLB implementation on the other hand reads in the dictionary traversing through the previously entered nodes as if it were a LinkedList with each sibling having a child node. The Node class allows for a char to be stored as well as references to a sibling and child node. The way that additions are handled is almost identical to the way that searches are done in terms of traversing the data structure, with the only main difference being the parsing of the string input for add with an iterator versus using the more robust and helpful methods of StringBuilders when performing a search. Each level we loop through the siblings of the current node until finding a character match, and then move to that node’s child which is essentially another linked list of nodes until the input is found. If the current node were to “fall off” and end up null, or we are unable to find a sibling node at the current level that matches, then the search quits and returns 0. The general structure of a DLB features smaller LinkedLists the further down the DLB a search takes it, allowing for similarly quick times for longer word searches. As previously mentioned, we will usually see only a run time of Θ(K), with K being the length of the key in the average case for a DLB search, making the DLB a vastly superior implementation of the DictInterface, and a vastly superior data structure for solving the crossword problem presented in Project 1.