Data Structure Analysis

Introduction

The objective of this ACW was to investigate the efficiency and performance of alternative data structures. These data structures are required to be created and developed to represent two 'WordSearch' type puzzles and two 'WordSearch' Dictionaries for the puzzles. The 'WordSearch' that will be used for this investigation would be a 9x9, typical square grid of characters and include a dictionary of the words within the grid. The grid will consist of some if not all words provided in the dictionaries; these words will be hidden within rows, columns and diagonals, both forward and backwards. The words will also cross over, so one character in a grid may contribute to multiple other words. The initial objective being to develop and implement the relevant data structures to represent the grid and dictionary for each puzzle described, to then be able to evaluate the performance and efficiency of the data structures in operation of the program that solves each puzzle. Two alternative structures for each aspect of the investigation; a simple method and an advanced method. Will be put under the same scenario of Puzzles to investigate the data they produce. In my investigation I can only provide data from the simple method; this data will be achieved from the program as a diagnostic report on the behavior and efficiency of the data structures.

Architecture & Implementation

The architecture of the program is pretty simple, the program starts by reading in the different data sets: ReadSimplePuzzle(), ReadSimpleDictionary(), ReadAdvancedPuzzle and ReadAdvancedDictionary(). Using the data sets solves the puzzle with different scenarios: SolveSimplePuzzleWithSimpleDictionary(), SolveSimplePuzzleWithAdvancedDictionary(), SolveAdvancedPuzzleWithSimpleDictionary() and SolveAdvancedPuzzleWithAdvancedDictionary(). All four scenarios will output a diagnostic type report that can be used to compare the different data sets and the methods used to process them.

My program only produced one set of results, thus starts by getting the data sets from a file to store on memory. ReadSimplePuzzle() method gets the data from the wordsearch_grid.txt file and stores it as a 2D character array, this is followed up by ReadSimpleDictionary() method which similarly gets the data from the dictionary.txt file and stores in a vector list. From this point the program will go one scenario and produce results on the data sets; SolveSimplePuzzleWithSimpleDictionary(). My Solving method goes through each direction individually; I have a temporary string variable that gets the 'strip' of characters in the direction it is going and checks the strip against any words given in the dictionary. Should the check fail the temporary string gets reset and the next 'strip' of characters gets check, should the check pass a word has been found in that 'strip'; using other variables planted in the code that counts the strips the position of the word can be found and also printed. The different directions require different algorithms to get the desired strip of characters from the grid that go in the string to be checked. The 'Right' Horizontal check; will consist of a 'for' loop in a 'for' loop, with the parameters set for the size of the grid. The initial for loop will for through each column of the grid and for every column another for loop will go through every row. Each column would be stored and check for a match before moving to the next column to repeat the process. This check is repeated with different parameters to go the opposite way; the 'Left' Horizontal check, instead of started on the right hand side of the grid it will start and the left and go the opposite way. The Diagonal checks where the algorithm got a lot more tricky as the length of the string varied; the diagonal check starts in the corner where this is only 1 character that makes a 'strip' for the string the next string will consist of 2, then 3… etc until it reaches the middle of the grid at its highest length 'strip' in this case being 9, after which it will start going back down to 1. Using an algorithm in the starting 'for' loop I was able to achieve the desired length values to get the correct length and not carry on to the next line. Inside this for loop resides a 'while' loop, which loops and gets a character based on the amount of characters that are needed which was determined by the initial for loop. Like described above, at the beginning of the diagonal search there is only one character in the corner of the grid so the 'while' loop can only loop once, then twice… etc. The method within the while loop was simple add/take from the x-axis, add/take from y-axis depending on which diagonal direction. The algorithm parameters is modified for each diagonal direction.

Once the program goes through all 8 directions, all possibilities have been checked and any words found are the only words present in the grid. So different variables placed through the process that count the amount of times the program did a function are then saved onto an output text file which is named in correspondence with the method that was used to get the results, in my case: results_simple_puzzle_simple_dictionary.txt. What my program does is essentially loops through all available words provided in the dictionary and attempts to match them with the 'strip' of text achieved from a direction on the grid, the ultimate goal of this investigation is to compare the results of that strategy against that of one that visits each letter in the puzzle grid and attempts to match the sequences from the position against the dictionary.

Results

I ran my program using 3 different Puzzle grids and Dictionaries on my PC*; all 3 grids consisted of a 9x9 square gird of characters and varying lengths of Dictionary. The first Dictionary consisted all words being located within the grid, the second most words being located within the grid and finally the last one only a few words could be located. Each Dictionary drastically increased in number of words it held.

*Need to note that my PC, is very high end and is capable of high processing speeds with an i7 Intel chip. However this should not impact the general correlation. The program was also ran in 'Release' mode and 'Run without Debug'.

Data Set	***1***	***2***	***3***
# of Words Matched	5	11	16
# of Grid Cells Visited	657	657	657
# of Dictionary Entries Visited	520	1976	218400
Time to Populate Grid Structure	0.000028	0.000035	0.000031
Time to Solve Puzzle	0.000033	0.000043	0.002192

1.1 Data Set 1
This Dictionary consisted of 5 words, all of which were located on the grid and could be found. As the table data states my method was able to pick up on all 5 words. The results also show that it took 657 cells to be visited along with 520 Dictionary entries visited to achieve all 5 words to be found. Time to populate the Grid and time to solve the puzzle was done incredibly fast

1.2 Data Set 2
This Dictionary consisted of 19 words, of which only 11 were located on the grid. The results show that my method found all 11 possible words in the grid and it took 657 cells to be visited with 1976 Dictionary entries visited to do so. Both the time to populate grid structure and time to solve puzzle increased very slightly.

1.3 Data Set 3
This Dictionary consisted of 2100 words, only 16 of which were located on the grid. My method again managed to find all 16 available words within the grid with the same amount of grid cells visited at 657, but with an incredibly increase to Dictionary Entries Visited at 218400, to find all 16 words. The time to solve the puzzle increased with the time to populate slightly decreasing.

Discussion of Results
My very first expectation before running the results was that the more words in the dictionary, would result in increased solve times. Which was proven correct, as the first Data Set which the Dictionary consisted of 5 words was the fastest out of all 3, as it had the least amount of words to process and thus match, allowing the program to go through the whole method the fastest. Data Set 2 had a little increase in time taken to solve the puzzle as the dictionary held 19 entries. Finally Data Set 3 was the slowest out of all with a big increase in duration to solve the puzzle in comparison to the other 2, as all 3 where able to solve the puzzle incredibly fast; even Data Set 3 was the slowest as it consisted of 2100 Dictionary Entries only took 0.002 seconds to complete.

The time it took to populate the grid structure was done very fast throughout all 3 data sets but was expected as all 3 had the same amount of characters laid out on a 9x9 square grid. The incredibly slight difference in time might be due to a number of things related to the running of the machine and the process in the background, it is safe to say that the time it took to populate the grid structure is the same across the board.

The number of Dictionary Entries Visited shows a very steep incline from Data set 1 to 3. The functionality behind these results would come down to the number of Dictionary Entries; every 'strip' of characters my method takes would be checked against the entire contents of the dictionary, so the more entries the more times these entries are visited. For example, given Data Set 1 with 5 words, given the Horizontal search algorithm and the grid being 9x9 resulting in 9 strips of characters would mean that the 5 Dictionary entries are tested 9 times (for every strip) for one horizontal method, the rest would follow and add to the count.

The number of Grid Cells visited stayed the same throughout the 3 Data Sets. As mentioned before my solve method goes through every direction and check every strip in that direction; this results in the solve method being completed after one take of all 8 directions. 657 cells visited is both the least and most cells needed to check through the entire grid on all possible direction for the words in the dictionary meaning that it will be the same no matter what data set is given as long as it fits the 9x9 cell grid parameter.

Finally numbers of words matched as the name suggests is the amount of words found in the data sets. This number is not always equal to the amount of data entries in the dictionary suggesting not all words were found; however not all words that were given are in the grid just like in a real life scenario where the WordSearch puzzle does not provide the words within the puzzle and the user uses a dictionary to go through a number of words that are not within the grid. The results do show that all possible words that could be found within the grid have been found.

Theory & Conclusion

The solving method I used was one of the two that was specified, the other solving strategy is to visit each letter in the puzzle grid and attempt to match the sequences from that position against the dictionary content. If I was to implement this strategy and achieve the results I would expect that this strategy would be more efficient in dealing with small amount of entries, especially the ones that consist of all the words being located in the grid. As there would be no need to go through all the possibly directions and test against the dictionary entries. Once a word has been found move on to the next one and once all words have been found just end the search. However when it comes to a higher amount of dictionary entries, especially ones that consist of entries that are not located in the grid it would greatly affect the efficiency of that strategy. Under those circumstances that is where my strategy would give and continue giving higher performance and efficiency with higher number of dictionary entries. I believe this will be true due to the fact my solve method only needs to loop once and all possibilities have been checked and if they are there, they have been found. It would struggle for efficiency against the other strategy with low entry dictionaries as it would still go through all possibilities even when the puzzle has been solved and/or check for words that have already been found. However with dictionaries with a higher entry count would be done with only one loop and any entries that do not exist within the grid would not impact the performance and efficiency as it would just keep checking the current 'strip' against the next entry until it is out of entries, sometimes resulting in multiple word matches in one 'strip' and one loop of the dictionary. Whereas the other strategy would have to check single cells versus 'strips' which hold multiple cells, and it would then have to test against all directions per cell. If it is trying to find a word that is not located in the grid it would have to do a whole loop of the grid before moving to the next word, which becomes a bigger problem and less efficient with every words that isn't in the grid.

In conclusion there is only a small window in which the second strategy is superior to mine, and after a certain small number of dictionary entries it would start to fall further and further behind in performance and efficiency. My solve method does is not efficient when it comes down to small amount of entries but shines after that same small number of dictionary entries and continues to get better the bigger that number gets.