08227 – Advanced Programming Report

# Introduction

I was asked to create a program which solves a word search when given a puzzle grid and a dictionary of words. We were tasked with storing each provided file in 2 different ways, one simple and one advanced, as such there was 4 combinations of solving using a different pair each time.

# Simple Puzzle

For my simple puzzle I have stored it using a 2d char array of pointers. To do this I first read in the characters in the puzzle grid file one at a time using an ifstream and add the character to a string ignoring any spaces between letters, as I do this I track the grid height incrementing it each time I reach and endline char, I do this so I don’t lose scope of the grid, after this I figure out my width by using the length of the constructed string and the grid height. Next I pass the created string to a function where it turns the string into a 2d array of pointers of size grid width and height.

When I get to solving the puzzle I use two for loops to loop through the x and the y and pass this into a method which returns the character at said position. I then check the returned character against the first letter of the current word in the dictionary if it matches I check around it and if any match the second letter of the current word I carry on in that direction matching one letter at a time until I have a matched word. If any of the characters don’t match at any point in the comparisons then I carry on in the dictionary until the end if no matches are found then I carry on in the grid until all words have been matched or the whole grid has been checked.

# Simple Dictionary

For my simple dictionary I use a vector of strings to store each word in the dictionary. To do this I use an ifstream to get each line in the dictionary and push\_back it to the vector one at a time creating a separate entry for each word.

When using the simple dictionary to solve I cycle through each entry in the vector using a vector iterator of type string. I then treat each word like an array and cycle through each character in the word checking against the grid for a match, if the whole word matches in the solving code then I output the words.

# Advanced Puzzle

For my advanced puzzle I create a vector of cells, each sell is a struct which contains the cells x and y position in the grid, the character at the certain position and a vector of strings which contains all eight strings stemming from each direction of the cell to the edge of the grid. To create the advanced puzzle I load the grid the same as the simple puzzle, using a string to hold all the characters in the file. I turn each character in the grid into a struct called cell this contains its character, its x and y positions and a vector of strings containing all of the strings stemming from said cell in all 8 directions creating strings from the characters in that direction until it reaches the edge of the grid.

When it comes to solving I cycle through the the x and y and pass this into a method along the current word being searched for, this method checks all the directional strings in the cell at the x and y position to see if they contain the word passed in. If the word is found then the method returns true and the word from the dictionary is outputted.

# Advanced Dictionary

My advanced dictionary is stored in a tree structure. To do this I use vector of base node pointers which are structs which contain a char, a bool whether it is and end char and a vector of node pointers to the children of the base node and each child contains the same set of data including a list of its children unless it is the an end char of a word with no children. To make it I take each separate line in the file one at a time and check if there is a base node with the same char as the first letter of the word if there is then the second letter is added as a child of the base node and its children is the next letter and so on. If a base node doesn’t already exist then it will add a new base node to the base node list and step through creating its children and children’s children.

When I use this for solving I start by getting the start node from the base node list where the character in the node matches the current letter being searched for, if it is found then I iterate through its children and recursively through its children checking each character against the grid. If a matching base node isn’t found then it will continue through the grid.

# Simple Puzzle + Simple Dictionary

|  |  |  |  |
| --- | --- | --- | --- |
| Data Set | Data set 1 (small) | Data set 2 (medium) | Data set 3 (large) |
| Time taken in secs | 0.0000168428 | 0.0000596432 | 0.0038661892 |
| No of matched words | 5 | 11 | 16 |
| No of gridcells visited | 124 | 249 | 8853 |
| No of dictionary entries visited | 405 | 1539 | 170100 |

# Simple Puzzle + Advanced Dictionary

|  |  |  |  |
| --- | --- | --- | --- |
| Data Set | Data set 1 (small) | Data set 2 (medium) | Data set 3 (large) |
| Time taken in secs | 0.0000120272 | 0.0000292805 | 0.0000461804 |
| No of matched words | 5 | 11 | 16 |
| No of gridcells visited | 276 | 554 | 1148 |
| No of dictionary entries visited | 276 | 554 | 1148 |

# Advanced Puzzle + Simple Dictionary

|  |  |  |  |
| --- | --- | --- | --- |
| Data Set | Data set 1 (small) | Data set 2 (medium) | Data set 3 (large) |
| Time taken in secs | 0.0000958966 | 0.0003238117 | 0.0322775531 |
| No of matched words | 5 | 11 | 16 |
| No of gridcells visited | 81 | 81 | 81 |
| No of dictionary entries visited | 405 | 1539 | 170100 |

# Advanced Puzzle + Advanced Dictionary

|  |  |  |  |
| --- | --- | --- | --- |
| Data Set | Data set 1 (small) | Data set 2 (medium) | Data set 3 (large) |
| Time taken in secs | 0.0000147695 | 0.0000311549 | 0.0000486206 |
| No of matched words | 5 | 11 | 16 |
| No of gridcells visited | 81 | 81 | 81 |
| No of dictionary entries visited | 235 | 490 | 980 |

# Comparison Conclusion

## Data Set 1

Looking at the timings above it shows that my slowest for this data set is the Advanced Puzzle + Simple Dictionary by almost five times that of the next slowest, with Simple Puzzle + Advanced dictionary as the quickest.

## Data Set 2

Looking at the timings above it shows that my slowest for this data set is still Advanced Puzzle + Simple Dictionary and the quickest is still Simple Puzzle + Advanced dictionary but the differences between them are greater than before

## Data Set 3

For the final set of data the quickest and slowest combinations are still the same but the difference between are even greater before

## Conclusion

Looking at all sets of data it all shows that Simple Puzzle + Advanced dictionary is always the quickest and that as you add more data to the dictionary not only does the times overall increase but so does the difference between the quickest and slowest.

By looking at my results I can see that the type of dictionary is the most limiting factor as the difference between Advanced Puzzle + Advanced Dictionary and Advanced Puzzle + Simple Dictionary is so much greater in both time and number of dictionary entries for all data sets than Simple Puzzle + Advanced Dictionary and Simple Puzzle + Simple Dictionary for which the same can be seen, this becomes more prominent as you add data. As such I can also see that the advanced puzzle is also more inefficient than the simple version however it can be seen that it is not as much of a limiting factor as the dictionary at larger data pools even though it is cycling through much less grid cells as each store eight strings and everyone is being searched rather than creating a string as the letters match and skipping the rest of the direction if a word isn’t matched halfway through.

# Efficiency Discussion

1. Select words from the dictionary and then search for them in the puzzle grid.
2. Visit each letter in the puzzle grid and attempt to match sequences from that position against the dictionary content.

It would be more efficient to use (b) by my reasoning because for each word in the dictionary you would be searching the whole grid each time so if there was five words in the dictionary and a 9x9 grid (81 cells) it means you would be searching 81 cells five times at least probably more depending on how you check the cell characters against the dictionary as you would have to combine all the characters into strings in all directions to be able to look for the word. However on the flip side using (a) when cycling through the cells you would only need to compare that character against the first character of each word in the dictionary so you can skip a lot of full word searches and behavior if the first letter doesn’t match but this would also work halfway through a match meaning you can cut out unnecessary comparisons across the board if the last letter of the sequence doesn’t match the current letter of the dictionary word.

Puzzle Grid

When using the simple puzzle I think it is better for (b), however for the advanced it would be better for (a) because you would be able to check if the cells directional string contain the word rather than checking each character in the string against each letter of the word, but for (b) it would be better to use the simple puzzle because you would need to step through each letter and compose the word as you go to match against the words in the dictionary and continuing to the next word or cell if it fails partway through.

Dictionary

When using the simple dictionary I think it would be better for (a) because you would already have

the full word when checking against the grid so you would not have to compose the word to be able

to check if it matches. For (b) advanced dictionary would be better to use because as you cycle

through the cells you would compare it against the character in the dictionary node iterating

through its children and so on as you match characters.

Conclusion

The optimum combination for scenario (a) in my opinion would be advanced puzzle + simple

dictionary but for (b) it would be simple puzzle + advanced dictionary. But overall (b) would be the

quickest which shows the same as my results.