Joshua Spisak

Assignment 1

CS 1501

09/27/2019

1. Discuss how I solved the crossword filling problem.
   1. Data Structures

I created a CrosswordState class which maintained a 2 by 2 array of chars to reflect the state a crossword puzzle. Within that board ‘+’ and ‘-‘ represented the same thing as in crossword text file, a lowercase character reflected a character that was set by the search algorithm, an upper-case character reflected one that was set in the crossword text file and could not be changed by the search algorithm.

Whenever the text file was inputted, the dimension would be stored in the class and checked whenever the rows and columns were read in. All characters read in from the text file would be converted to uppercase. Whenever the crossword would be printed out all characters would be converted to lower-case.

Functions were provided to tell whether a cell was part of a word, or were a filled in black; and also to tell if a cell could be modified. Functions were also provided to get a string builder for the word leading up to a certain cell in both the vertical and horizontal directions. So if you requested the horizontal word for the cell at row 0 column 2 and there were no filled in cells you would get the contents of (0,0), (0,1), and (0,2). This operates with the assumption that the cells will be filled in from the top left, to the end of that row... then down to column 0 of the next row to the end of that row… and so forth until it reaches the bottom right.

A few other accessory functions were added to set and unset cells, get the board dimensions, print it, get a copy of the char array, among other things.

* 1. How the algorithm Proceeded

The algorithm starts at the top left corner of the puzzle. It will then proceed along that row until until it reaches the end. It will then go to the first column of the next row and on from there until it reaches the bottom right. For each cell it is going to see if the cell is part of a word. If it can it will try each letter a-z, if not it will only try the existing letter. If/when it gets a successfully match for both the vertical and horizontal words with a particular letter it will proceed recursively to the next and so forth. If it does not match and it was a letter it can change it will remove it and backtrack otherwise it will leave it with the existing letter.

If the cell is not part of a word then it will proceed to the next cell automatically, and backtrack if/once the next cell returns. If the algorithm reaches the bottom right cell and it either contains a letter that it successfully sets or it cannot set it at all, that means all the previous cells were successful as well and the current crossword state is valid so it prints and increments the number of solutions. For the first task it will return all the way down the stack, otherwise it will either try the next letter for that cell or backtrack to try other permutations of the board for more solutions.

* 1. Coding/Debugging Issues

When I had started implementing my logic was a little scattered and there was a lot of repeat logic. Since I was having some issues I added a verbosity argument to my program that added debug printouts during runtime. This quickly let me find the errors in my logic, streamline the repeat logic into functions and proceed to the DLB implementation.

With the DLB implementation I had little trouble but eventually implemented a print\_nodes function that showed the tree structure/contents to make it easier to see what was going on while implementing a delete() function.

1. Run-Time Differences between MyDictionary and DLB (to find all solutions).

|  |  |  |
| --- | --- | --- |
| File | DLB Run-time | MyDictionary Run-time |
| Test3a.txt | 71 seconds | > 2 hours (est 200 hrs) |
| Test3b.txt | 1 second | 49 seconds |
| Test4a.txt | > 2 hours (est 1.58 x hrs) | > 2 hours (est 5.58 x hrs) |
| Test4b.txt | > 2 hours (est 6092505 hrs) | > 2 hours (est 4.58 x hrs) |
| Test4c.txt | 99 seconds | > 2 hours (est 300 hrs) |
| Test4d.txt | 1 second | 4 seconds |
| Test4e.txt | 6950 seconds | > 2 hours (est 22000 hrs) |
| Test4f.txt | 1.29 seconds | 134 seconds |
| Test5a.txt | > 2 hours (est 8.6 x hrs) | > 2 hours (est 1.4 x hrs) |
| Test6a.txt | > 2 hours (est 2.1 x hrs) | > 2 hours (est 2.1 x hrs) |
| Test6b.txt | > 2 hours (est 1.1 x hrs) | > 2 hours (est 1.0 x hrs) |
| Test6c.txt | 72 seconds | > 2 hours |
| Test7a.txt | > 2 hours (est 6.0 x hrs) | > 2 hours (est 9.1 x hrs) |
| Test8a.txt | > 2 hours (est 1.3 x hrs) | > 2 hours (est 2.2 x hrs) |
| Text8b.txt | > 2 hours (est 4.2 x hrs) | > 2 hours (est 8.7 x hrs) |
| Test8c.txt | 18 seconds | > 2 hours (est 200,000 hrs) |

(estimates were made from estimating the factors of the worst case equations.. then taking it down a bit)

1. Asymptotic Analysis of Worst Case Run Time for each Version

|  |  |
| --- | --- |
| Implementation | Worst Case Search Time |
| MyDictionary |  |
| DLB |  |