Garrett Hagen

COE 1501

Assignment 1 Reflection and Analysis

*Writing the Program:* When writing my crossword-solving program, I approached the problem by dissecting it into smaller, modular tasks. I began by implementing the basic functionality such as argument parsing, creating the dictionary, reading in the file, board initialization, and printing solutions. For a puzzle of size NxN, I used the recommended approach of making StringBuilder arrays for the row and column words of size N and a 2D char array of size NxN for the game board. Following this, I was able to begin working on the buildCrossword(i,j) method which I will break down in the following section:

1. **Backtracking Approach**: The parameters for my recursive buildCrossword(int i, int j) method were the current row position (i) and the column position (j) of the puzzle with the initial call being buildCrossword(0,0)
   1. Calculate the parameters of the next recursive call: The puzzle will iterate along the row (increment j) until the end of the row and then increment i and reset j to 0. If i was greater than the number of rows in the puzzle that indicates we have found a solution. If we are using the DLB, backtrack and continue solving, otherwise, print the solution and kill the program.
   2. Append a character to rowStr[i] and colStr[j]: If the character in gameBoard[i][j] is a ‘-’ or a letter we need to append it immediately, check it, and either recurse or remove it and backtrack depending on if the safe(i,j) method determines it as a valid placement. Otherwise, we will have to pick a character ourselves to add which is accomplished by iterating through the alphabet and appending a character and checking it. If it is valid, recurse, otherwise remove it and keep iterating through the alphabet. If the end of the alphabet is reached and we do not have a valid placement, backtrack.
2. **Placement Check:** The safe(i,j) method checks valid character placement in both the row and column directions and will return true if both the placement is valid in both directions. The rowCheck and colCheck methods are identical, but check conditions in horizontal and vertical conditions respectively. The methods work as follows:
   * 1. Check to see if the current character is a ‘-’.
        1. The ‘-’ is at the 0th index and nothing precedes it so return true.
        2. The character preceding it is also ‘-’ so return true.
        3. There is a string between the ‘-’ and the 2nd most recent ‘-’ (or 0th index if this is the first ‘-’ in the string) and it must be a word.
     2. In the latter case from Part i. shown above, 1 of 2 scenarios will happen:
        1. We are at an end index (i.e. j==endIndex, i==endIndex or the current character is ‘-’) so the string in (rowStr[i],colStr[j]) respectively must be a word.
        2. (j<endIndex, i<endIndex and the current character is not a ‘-’) so rowStr[i] and colStr[j] respectively must be a prefix or a prefix+word because we still need to append more letters.
3. **Issues/Debugging**: Currently, my program runs with the exception of a small bug that prevents it from finding a solution for test8c and drops solutions on test4c.
   1. Backtracking Algorithm: Getting the backtracking algorithm to run was not terribly difficult, but I initially had some trouble with iterating through the gameBoard and my first implementation was very inefficient. My first implementation had one recursive call and would check the character type inside the alphabet loop, thus, if the board character was not a ‘+’ it would add and delete an immutable character 26 times before finally backtracking. I fixed this by splitting the method into the two sections described in part b of the Backtracking Approach section above.
   2. Safe Method: I started by just solving for a blank puzzle, which was relatively straightforward thanks to the hints on the Assignment 1 pdf. The ‘-’ character is what caused the most problems, as there were special cases (i.e index 0, back to back ‘-’ and having multiple ‘-’ in the same string). I was able to work through this by examining all of these possible scenarios on a whiteboard and then implementing the logic to handle each case.

*Runtime Analysis:* The DLB implementation of DictInterface provides a drastic increase in run-time. I included timing in the program and terminated execution for test cases that did not find a solution within 2.5 hours (9000 sec). The asymptotic analysis of worst-case for NxN crossword of blank squares is explained below:

1. MyDictionary:
   1. Build Cost: For an ArrayList dictionary of size M \* log2M (MergeSort) = M\*logM = O(Mlog2M)
   2. Search Cost: For a puzzle of size NxNsquares, with 26letter options per square, so 26NxN letter combinations multiplied a lookup time of dictionary length M \* word length N so worst case is approximated as O(Lookup\*Combinations) = O(M\*N\*26NxN)
   3. Total = Build + Search → O(Mlog2N) + O(MN\*26NxN) ~ O(MN\*26NxN)
2. DLB:
   1. Build Cost: For a dictionary of size M \* word length N = O(MN)
   2. Search Cost: Backtracking run-time same as MyDictionary but now word search time is linear so we don’t have to iterate over a dictionary of size M and run-time O(N\*26NxN)
   3. Total Total = Build + Search → O(MN) + O(N\*26NxN) ~ O(N\*26NxN)
3. Graphical Analysis:

MyDictionary vs. DLB Run-Times (seconds)

|  |  |  |
| --- | --- | --- |
| **Test** | **MyDictionary** | **DLB** |
| test3a | 0.326 | 0.002 |
| test3b | 5.331 | 0.017 |
| test4a | 15.911 | 0.026 |
| test4b | 10.781 | 0.076 |
| test4c | 1.06 | 0.007 |
| test4d | 0.259 | 0.001 |
| test4e | 9000 | 6.692 |
| test4f | 20.727 | 0.055 |
| test5a | 15.448 | 0.056 |
| test6a | 9000 | 236.714 |
| test6b | 9000 | 31.918 |
| test6c | 9000 | 22.354 |
| test7a | 9000 | 9000 |
| test8a | 9000 | 9000 |
| test8b | 9000 | 9000 |
| test8c | 20.867 | 0.035 |

