CS/COE 1501 – Algorithm Implementation – Assignment#1[1]

Due: Wednesday October 2nd @ 11:59pm

Late submission deadline: Friday October 4th @11:59pm with 10% penalty per late day

OVERVIEW

Purpose: To implement backtracking algorithms and search trees.

Task 1: The first task of the assignment is to create a backtracking algorithm that finds one legal filling of the squares of a given crossword puzzle (if a legal filling exists), as specified in detail below.

Task 2: The second task is to implement a De La Briandais (DLB) trie and use it to improve the search efficiency in Task 1 and hence to be able to produce all possible legal fillings of a given crossword puzzle.

BACKGROUND

Crossword puzzles are challenging games that test both our vocabularies and our reasoning skills. However, creating a legal crossword puzzle is not a trivial task. This is because the words both across and down must be legal, and the choice of a word in one direction restricts the possibilities of words in the other direction. This restriction progresses recursively, so that some word choices "early" in the board could make it impossible to complete the board successfully. For example, look at the simple crossword puzzle below (note: no Xs appear in the actual puzzle shown below – in this example X is always used as a variable):

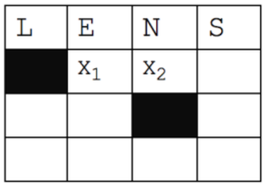

Figure 1

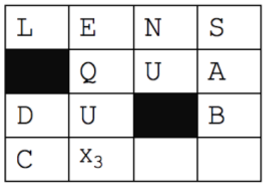

Figure 2

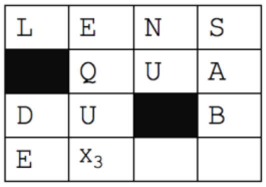

Figure 3

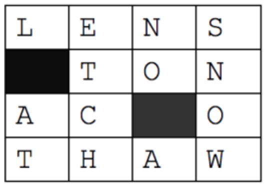

Figure 4

Assume that the word LENS has been selected for row 0 of the puzzle, as shown in Figure 1 above. Now, the word in column 1 (the second column) must begin with an E, the word in column 2 must being with an N and the word in column 3 must begin with an S. All single characters are valid words in our dictionary so the L in column 0 is a valid word but is also irrelevant to the rest of the puzzle, since its progress is blocked by a filled-in square. There are many ways to proceed from this point, and finding a good way is part of the assignment. However, if we are proceeding character by character in a row-wise

[1] Assignment adapted from Dr. John Ramirez's CS 1501 class.

fashion, we now need a letter X_1 such that EX_1 is a *valid prefix* to a word. Several letters will meet this criterion (EA, EB and EC are all valid prefixes, just to pick the first three letters of the alphabet). Once a possibility is selected, there are now two restrictions on the next character X_2: NX_2 must be a valid word and X_1X_2 must be a valid prefix to a word (see Figure 1). Assume that we choose Q for X_1 (since EQ is a valid prefix). We can then choose U for X_2, (see Figure 2 (NU is a valid word in our dictionary)). Continuing in the same fashion, we can choose the other letters shown in Figure 2 (in our dictionary QUA, DU and DC are all legal words).

Unfortunately, in row 3, column 1 we run into a problem. There is no word in our dictionary $EQUX_3$ for any letter X_3 (note that since we are at a terminating block, we are no longer just looking for a prefix) so we are stuck. At this point we need to undo some of our previous choices (i.e., backtrack) in order to move forward again toward a solution. If our algorithm were very intelligent, it would know that the problem that we need to fix is the prefix EQU in the second column. However, based on the way we progressed in this example, we would simply go back to the previous square (row 3, column 0), try the next legal letter there, and move forward again. This would again fail at row 3, column 1, as shown in Figure 3. Note that the backtracking could occur many times for a given board, possibly going all the way back to the first word on more than one occasion. In fact, the general run-time complexity for this problem is exponential. However, if the board sizes are not too large, we can likely solve the problem (or determine that no solution exists) in a reasonable amount of time. One solution (but not the only one) to the puzzle above is shown in Figure 4.

TASK 1 – FINDING A SINGLE SOLUTION

The first task of your assignment is to create a legal crossword puzzle (if it exists) in the following way:
1) Read a dictionary of words in from a file and form a MyDictionary of these words. The name of the file should be specified as a command-line argument. The interface DictInterface (in DictInterface.java) and the class MyDictionary (in MyDictionary.java) are provided for you on CourseWeb, and you **must** use them in this assignment. Read over the code and comments carefully so that you understand what they do and how. The interface DictInterface will also be important for the second task of this assignment. The file used to initialize the MyDictionary will contain ASCII strings, one word per line. Use the file dict8.txt on CourseWeb. If you are unsure of how to use DictInterface and MyDictionary correctly, see the DictTest.java example program (and read the comments). Lab 1 solution can be used for reference as well.

2) Read a crossword board in from a file. The name of the file should be specified as a command-line argument. The crossword board will be formatted in the following way:

a) The first line contains a single integer, N. This represents the number of rows and columns that will be in the board. Since the dictionary will contain up to 8-letter words, your program should handle crosswords up to 8x8 in size.

b) The next N lines will each have N characters, representing the NxN total locations on the board. Each character will be either

i. + (plus) which means that any letter can go in this square
ii. – (minus) which means that the square is solid (filled-in) and no letter can go in here
iii. A..Z (a letter from A to Z) which means that the specified letter must be in this square (i.e., the square can be used in the puzzle, but only for the letter indicated)

For the board shown above, the sample.txt file would be as follows:

```
4
++++
-+++
++-+
++++
```

Some test boards have been put onto the assignment page on CourseWeb.

3) Create a legal crossword puzzle for the given board and print it out to standard output. Many of the test files may have many solutions, but for this part of the project you only need to find **one solution**. See the second task of this assignment for information about finding multiple solutions. For example, one output to the crossword shown above in Figure 4 would be

```
> java Crossword dict8.txt sample.txt
LENS
-TON
AC-O
THAW
>
```

Depending upon your algorithm, the single solution that you find may differ from that of my program or your classmates' programs. This is fine as long as all of the solutions are legal. Note that because of the severe performance limitations of the MyDictionary class, some of the run-times for the test files will be very long. See more details on this in testFiles.html available on CourseWeb.

Important Notes on Task 1:

– To help you to get started,
    - first think of boards with all squares open (you can consider filled in squares later). In this case a solution for a KxK board will consist of K words of length K in the columns of the board and K words of length K in the rows of the board.
    - Your program has to make one **decision** for each square of the board. How many options do you have for each decision? What are these options? You will have to choose from the 26 letters.
    - Construct an array of K StringBuilders for the columns (call it colStr) and an array of K StringBuilders for the rows (call it rowStr), each initially empty. Now consider a single recursive call at square (i,j) on the board. For an option (i.e., a character) at position (i,j) to be **valid**, the following must be true:

        - If j is not an end index, then rowStr[i] + the char must be a valid prefix in the dictionary
        - If j is an end index, then rowStr[i] + the char must be a valid word in the dictionary
        - If i is not an end index, then colStr[j] + the char must be a valid prefix in the dictionary
        - If i is an end index, then colStr[j] + the char must be a valid word in the dictionary
    - If the character is valid you append it to both corresponding StringBuilders and recurse to the next square (unless you are on the last square of the board, in which case you have a solution!). If it is not valid you try the next character at that square or, if all have been tried, you backtrack.
- For consistency, call your main program Crossword.java.
- **Search algorithm details:** Carefully consider the algorithm to fill the words into the board. Make sure it potentially considers all possibilities yet does not waste time rechecking prefixes that have already been checked. Although you are not required to use the exact algorithm described above, your algorithm must be a recursive backtracking algorithm that uses pruning. The algorithm you use can vary greatly in its efficiency. If your algorithm is very inefficient or otherwise poorly implemented, you will lose some style points. This algorithm is a significant part of the overall assignment, so put a good amount of effort into doing it correctly. For guidance on your board-filling algorithm, it is strongly recommended that you revise the Boggle game lab code.
- The MyDictionary implementation of the DictInterface that is provided to you should work correctly, but it is not very efficient. Note that it is doing a linear search of an ArrayList to determine if the argument is a prefix or word in the dictionary. In the second task of this assignment you will write a more efficient implementation of the DictInterface.
- Be sure to thoroughly document your code, especially the code that fills the board.

TASK 2 – FINDING ALL SOLUTIONS USING A DE LA BRIANDAIS TRIE

In Task 1 of this assignment, you were asked to complete a recursive, backtracking solution to filling in the squares of a crossword puzzle. To do this you utilized the DictInterface interface and the MyDictionary class, both of which were provided for you. Unfortunately, the MyDictionary class implements the DictInterface in a somewhat primitive and inefficient way, utilizing a linear search of the dictionary array. This can lead to long run-times when many searches are required (as in the crossword problem!).

Consider the De La Briandais (DLB) tries, which we discussed in lecture and recitation. Since these tries allow a string to be tested as a word or as a prefix in the dictionary in (typically) time proportional to the length of the string, they appear to be a superior way to implement the DictInterface interface. In Task 2 of this assignment you will do the following:

1) Implement a DLB as a class, as discussed in lecture and in recitation. This requires the "nodelets" within the DLB to contain a character field, a child reference and a sibling reference. Your class(es) should be written in reasonably good object-oriented style. You may have a number of methods in your DLB class, but it must minimally implement the DictInterface as specified. Verify that this

implementation works by testing your DLB using the DictTest.java program that I have provided for you on CourseWeb. For consistency in grading and testing, you must call your class DLB and it must be in a file named DLB.java.

2) Modify your main program from Task 1 in the following ways:
    a. Have the type of the DictInterface object as a command-line argument so that the user can run the same program with either a MyDictionary or a DLB. Look over DictTest.java to see how this would be done.
    b. Update your program so that if the DictInterface object is a DLB it will find **ALL** of the solutions rather than just the first solution. This can be done with two simple changes to your main program:
        i. When a solution is output, test again to see the type of the DictInterface object. If the type is MyDictionary, stop the program execution, but if the type is DLB, continue execution.
        ii. Rather than having your backtracking algorithm complete (terminate) once a solution is found, have it backtrack and continue to search for solutions. This is the approach taken in the BoggleSolver.java program in Lab 1. Look over that to get an idea of how to do this.

    Since some of the test files will have thousands and even millions of solutions, in this case do not print out the solutions. At the end of your execution, print out the total number of solutions found. *Don't forget to do this as it will be used in evaluating the correctness of your algorithm.*

```
> java CrosswordB DLB dict8.txt test3b.txt
1
>
```

3) Compare the execution of your crossword solution with the MyDictionary to that using the DLB. Do this by informally recording the amount of time required for the first solution to be found (or for the program to terminate if no solutions exist) with both dictionaries. To informally time the programs, run them while watching your clock/watch/etc. We are not worried about exact times here – just orders of magnitude. Based on your comparisons of the various test files, determine if there is a significant difference in the run-times.
4) For some test files, one or perhaps both versions will take several minutes or perhaps hours or even days. Based on some worst-case analysis and some real timing of the smaller files you should be able to get an idea of how long the larger files will take to run. If a program takes more than a few hours you can abort the execution.
5) Once you have completed your comparison runs, write a short paper (2-3 pages, double-spaced) that summarizes your project in the following ways:
    a. Discuss how you solved the crossword-filling problem in some detail. Include
        i. how you set up the data structures necessary for the problem and
        ii. how your algorithm proceeded.
        iii. Also indicate any coding or debugging issues you faced and how you resolved them. If you were not able to get the program to work correctly, still include your approach and speculate as to what still needs to be corrected.

b. Discuss the differences (if any) between the run-times of the program using the MyDictionary and the program using the DLB. Include the (approximate) run-times for the programs for the various files **in a table**.
c. Include an asymptotic analysis of the worst-case run-time for each version of the program. Some values to consider in this analysis may include:
    i. Number of words in the dictionary
    ii. Number of characters in a word
    iii. Number of possible letters in a crossword location
    iv. Number of crossword locations in the puzzle

   If you were unable to complete the crossword solving program, speculate (using some intelligent guessing) for the actual run-times, but still include the comparison in your paper.

Important Notes for Task 2:

1. For consistency, call your main program CrosswordB.java.
2. When grading your assignments, your TAs may redirect your output to a file so that they can refer to it at a later point. To make sure this will work correctly, make sure that once your search algorithm begins there is NO INPUT or anything that would make your program require any user interaction (since the TA will not see any prompts given that they will be sent to a file rather than the display).
3. Read over the test file page testFiles.html on CourseWeb for important information on the test files and some expected results.
4. Be sure to thoroughly document your code.
5. This assignment was devised so that Task 1 and Task 2 could be implemented independently. If you are unable to complete one task don't let that prevent you from completing the other. Also, be sure to try each and submit something so that you can get some partial credit.

EXTRA CREDIT

If you are interested in some extra credit for this assignment, here are some possibilities:

1. Create a third implementation of the DictInterface and compare its performance to that of the other two implementations. One possibility is to use binary search of a sorted array. However, searching for a prefix is tricky if binary search is used, so be careful with this approach (i.e., make sure it is correct).
2. Do more detailed analysis of the run-times of the two different DictInterface implementations. Add timing to your program and do multiple runs to obtain (reasonably) accurate values of the actual run-times. Create a graph to show how the run-times increase as the crossword board size increases.
3. Add and demonstrate (through a driver program that you write yourself) a delete() method for your DLB.

SUBMISSION REQUIREMENTS

You must submit the following files **individually** to GradeScope:

1) Crossword.java
2) CrosswordB.java
3) DLB.java
4) Any other helper files that you had to add to support your implementation
5) Well written/formatted paper explaining your search algorithm and results (see Task 2 above for details on the paper)
6) Assignment Information Sheet (including compilation and execution information).

The idea from your submission is that your TA can compile and run your programs **from the command line** WITHOUT ANY additional files or changes, so be sure to test it thoroughly before submitting it. If the TA cannot compile or run your submitted code it will be graded as if the program does not work.

If you cannot get the programs working as given, clearly indicate any changes you made and clearly indicate why on your Assignment Information Sheet. You will lose some credit for not getting it to work properly, but getting the main programs to work with modifications is better than not getting them to work at all. A template for the Assignment Information Sheet can be found in the assignment's CourseWeb folder. You do not have to use this template, but your sheet should contain the same information.

Note: If you use an IDE, such as NetBeans, Eclipse, or IntelliJ, to develop your programs, make sure the programs will compile and run on the command-line before submitting – this may require some modifications to your program (e.g., removing package information).

RUBRICS

Please check the grading rubrics on CourseWeb.