CS 445 – Data Structures – Assignment#3[1]

Due: Tuesday Nov. 14th@ 11:59pm

All source files plus a completed Assignment Information Sheet zipped into a single .zip file and submitted properly to CourseWeb.

Late submission: Thursday Nov. 16th @11:59pm with 10% penalty per late day

OVERVIEW

Purpose: We have discussed recursion and in particular backtracking algorithms (such as Eight Queens). In this assignment you will get some practice at recursive programming by writing a backtracking algorithm to find phrases in a puzzle.

Idea: "Word search" puzzles are common in newspapers, on restaurant placemats and even in their own collections within books. These vary in format from puzzle to puzzle, but one common format is as follows: A user is presented with a 2-dimensional grid of letters and a list of words. The user must find all of the words from the list within the grid, indicating where they are by drawing ovals around them. Words may be formed in any direction – up, down, left or right (we will not allow diagonals even though most puzzles do allow them) but all of the letters in a word must be in a straight line.

This idea can be extended to allow ***phrases*** to be embedded in the grids. However, requiring an entire phrase to be in a straight line on the board is not practical, as the dimensions of the board would need to be overly large. Therefore, we will still require the letters in each word to be in a straight line, but we are allowed to change direction between words.

For example, we might be given the following 10x10 grid of letters:

```
a b s t r a c t i j
a t a d t d a t a j
t b c d c a g h t j
y b c d a t g h y j
p b c d r a g h p j
e b c d t f l h e j
s a r e s f o h s j
a r e d b f o h a j
r b c d a f c h r j
e r e a l l y r e j
```

and the following phrases:

[1] Assignment adapted from Dr. John Ramirez's CS 445 class.

```
        abstract
        abstract data types
        abstract data types are really cool
        abstract data types are really awesome
```

Upon searching, assuming the grid starts in the upper left corner with position (0,0), and that the row is the first coordinate:

`abstract` would be found in two places:
  [(0,0) to (0,7)] and
  [(8,4) to (1,4)]

`abstract data types` would also be found in two places:
  [(8,4) to (1,4)], [(1,5) to (1,8)], [(2,8) to (6,8)] and
  [(8,4) to (1,4)], [(1,3) to (1,0)], [(2,0) to (6,0)]

`abstract data types are really cool` would be found at:
  [(8,4) to (1,4)], [(1,3) to (1,0)], [(2,0) to (6,0)], [(7,0) to (9,0)], [(9,1) to (9,6)], [(8,6) to (5,6)]

`abstract data types are really awesome` would not be found

DETAILS

Your task is to write a Java program to read in a grid of letters from a file, and then interactively allow a user to enter phrases until he or she wants to quit. For each phrase your program must output whether or not the phrase is found and, if found, specifically where it is located.

Input Details:

The grid of letters for your program will be stored in a text file formatted as follows:
  Line 1: Two integers, R and C, separated by a single blank space. These will represent the number of rows, R and columns, C, in the grid.
  Remaining lines: R lines containing C lower case characters each.

The user input will be phrases of words, with a single space between each word and no punctuation. Each phrase will be entered on a single line. The user may enter either upper or lower case letters, but the string should be converted to lower case before searching the grid. The program will end when the user enters no data for the current phrase (i.e. hits <Enter> without typing any characters beforehand).

Output Details:

If a phrase is not found in the grid the output should simply state that fact.
If a phrase is found in the grid, your program must find **one occurrence of the phrase**, and the output must indicate this fact in two ways:
1) Show the coordinates of each word in the phrase as a pair of (row, column) pairs.
2) Show the grid with the letters of the phrase indicated in upper case

For example, for the grid above and the phrase "abstract data types" your output would be:

```
abstract: (8,4) to (1,4)
data: (1,5) to (1,8)
types: (2,8) to (6,8)
```

```
a b s t r a c t i j
a t a d T D A T A j
t b c d C a g h T j
y b c d A t g h Y j
p b c d R a g h P j
e b c d T f l h E j
s a r e S f o h S j
a r e d B f o h a j
r b c d A f c h r j
e r e a l l y r e j
```

Algorithm Details:
Your search algorithm must be a recursive, backtracking algorithm. Note that you do not need recursion to match the letters within individual words (although you may do this recursively if you prefer). Recursion is necessary when moving from one word to the next – since it is here where you may change direction. To make the program more consistent (and easier to grade), we will have the following requirements for the recursive process:

1) No letter / location on the grid may appear more than one time in any part of a solution.

2) When given a choice of directions, the options must be tried in the following order: right, down, left, up. Given this ordering and the grid above, the solution for "abstract data" would be [(8,4) to (1,4)], [(1,5) to (1,8)] rather than [(8,4) to (1,4)], [(1,3) to (1,0)], since the "right" direction is tried before the "left" direction.

3) If the last letter in a word is at location (i, j), the first letter of the next word must be at one of locations (i, j+1), (i+1, j), (i, j-1), or (i-1, j).

4) The direction chosen to find the first letter of a word is the same direction that must be used for all of the letters of the word. For example, in the grid shown above, for the phrase "abstract data", the following would NOT be a valid solution: [(8,4) to (1,4)], [(1,5) to (4,5)]. This solution is not legal because we proceeded right from the "T" of "abstract" to find the "D" in "data", but then proceeded down to find the remaining letters in "data". Similarly, also in the grid shown above, for the phrase "abstract data types are", the following would NOT be a valid solution: [(8,4) to (1,4)], [(1,3) to (1,0)], [(2,0) to (6,0)], [(7,0) to (7,2)]. This solution is not legal because we proceeded down from the "S" of "types" to find the "A" in "are", but then proceeded right to find the remaining letters in "are".

The idea is that you are building a solution word by word. Each time you complete a word, you can look for the "next" word in any of the four directions – this is where the recursion occurs. If the "next" word cannot be found in any of the directions, you must delete the most recently completed word and backtrack to the previous word.

For example, consider the board above with the search phrase "data types are really". The algorithm first tries to find the word "data" starting at position (0,0). It tries in all four directions and does not succeed. It proceeds through the other starting positions until it finds "data" in locations [(1,3) to (1,0)].

It now recursively tries to find "types" in the following ways:

> Going right to position (1,1) – this will not work since (1,1) is already being used in "data"
>
> Going down to position (2, 0) – this will succeed and "types" is found in locations [(2,0) to (6,0)].
>
> The algorithm now recurses again, this time looking for "are" in the following ways:
>
> > Going right to position (6,1) – this will succeed and "are" is found in locations [(6,1) to (6,3)].
> >
> > The algorithm now recurses again, this time looking for "really" in the following ways:
> >
> > > Going right to position (6,4) – this will not work since character (6,4) is an "s"
> > >
> > > Going down to position (7,3) – this will not work since character (7,3) is a "d"
> > >
> > > Going left to position (6,2) – this will not work since (6,2) is already being used in "are"
> > >
> > > Going up to position (5,3) – this will not work since character (5,3) is a "d"
> > >
> > > Since all directions have been tried, the search for "really" has failed and the algorithm must backtrack.
> > >
> > > The word "are" is removed and the previous search for "are" resumes
> >
> > Going down to position (7,0) – this will succeed and "are" is found in locations [(7,0) to (9,0)].
> >
> > The algorithm now recurses again, this time looking for "really" in the following ways:
> >
> > > Going right to position (9,1) – this will succeed and "really" is found in locations [(9,1) to (9,6)].
> > >
> > > *The last word in the phrase has been found and the algorithm succeeds*

Clearly, more backtracking may be necessary in other situations. I recommend tracing through the process with some example phrases before you start coding your solution.

A non-trivial part of this assignment is keeping track of the "path" of the solution and printing the path out once the solution has been found. You will likely need to use some data structure to store / update this path. Think carefully how you can do this part of the assignment.

Test Files:

I have placed several test files and some sample outputs on CourseWeb. See the Assignment page for these files and make sure that your program works correctly for all of the sample files.

Help:

If you are having trouble with recursion and backtracking, the program FindWord.java may be of help to you. This program finds individual words in a grid of characters using recursion and backtracking. It recurses on individual characters rather than on words, but the recursive process is similar in both programs. See also test file findWord1.txt.

Important Note: The code from FindWord.java to read in and set up the grid can be taken and used directly in your program if you wish. However, you should NOT use the recursive part of FindWord directly in your program since it is not solving the problem you are trying to solve. Further, you should not use this code to find individual words in your grid since FindWord allows directions to change within a single word, whereas in your assignment you are only allowed to change directions between words. See additional comments in the FindWord.java code.

EXTRA CREDIT

Allow your program to recurse diagonally as well as horizontally and vertically. If you do this you will need to make up your own test files to demonstrate that your program works correctly.

SUBMISSION REQUIREMENTS

Make sure you submit all of the following in **a single .zip file**:
1) All source files that you either wrote or utilized (ex: from the Textbook author's files). Call your main program file **Assig3.java**.
2) All data files
3) Your completed Assignment Information Sheet

As with Assignments 1 and 2, the idea from your submission is that your TA can compile and run your programs WITHOUT ANY additional files, so be sure to test them thoroughly before submitting them. **If you use an IDE for development, make sure your program runs from the <u>command line</u> without the IDE before submitting it.** If you cannot get the program working as specified, clearly indicate any changes you made and clearly indicate why, so that the TA can best give you partial credit. 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.

RUBRICS

Please check the grading rubric on CourseWeb.