Lab 8: Word Search[1]

Due: 16 November 2016

In this lab, you are going to write a program to solve Word Search puzzles. This will give you more experience writing Java code, but also allow you to explore algorithm efficiency issues. A Word Search puzzle presents the user with a simple two-dimensional array of characters, in which they need to find words. In some variations, the user is also presented with a list of words to find in the puzzle. The words may be horizontal, vertical, or diagonal and in any direction (for a total of eight directions). Consider the following puzzle:

T H A S
W A T S
O A H G
F G D T

It contains, for example, the words **two**, **fat**, and **that**, the last of which begins at (3,3) and goes diagonally to (0,0). Note that although this puzzle has four rows and four columns, that will not always be the case. In this lab, the goal will be to find all the words from a dictionary that appear in a given puzzle.

Algorithms

Naive algorithms are generally the first thing you think of when you encounter a new problem. The most obvious way to solve a puzzle in this lab is to use brute force, i.e. for each word in the dictionary, check to see if it appears anywhere in the puzzle:

```
for each word W in the dictionary
     for each row R in the puzzle
          for each column C in the puzzle
               for each direction D
                    check for W at row R, column C in direction D
```

Since there are 8 directions, this algorithm requires 8WRC checks. For a typical puzzle you might find in a magazine, you would expect about 40 words and a 16x16 grid. That results in about 82,000 checks. No problem there, but we're interested in a variation where we find *all possible words* in the grid, i.e. anything that is in the dictionary. In the dictionary you will be using, the number of words is about 173,000, which would result in about 354,000,000 checks. If the grid were doubled (32x32), it would take about 1,400,000,000 checks. Now we're talking about large numbers. Being impatient, we'd like our algorithm to solve this instantly, certainly in less than a second. To do this we'll need to be a little smarter. Most of the words in the dictionary will not be in a given puzzle, so we're wasting a lot of time by trying to find every

[1] This lab was created by Eric Chown for a previous Data Structures class, and was based on an example by Mark Allen Weiss.

dictionary word in the puzzle. Instead, let's check to see whether the character strings that are in the actual puzzle are words in the dictionary:

```
for each row R in the puzzle
     for each column C in the puzzle
          for each direction D
               for each word length L
                    check if L chars starting at R,C
                    in direction D form a word
```

The new algorithm rearranges the loop to avoid searching for every word from the dictionary. Instead, if focuses on what is actually in the puzzle. If for example, the longest word in the list were 20, then the algorithm would need 160RC checks. For our 32x32 case this is roughly 164,000 checks, a huge improvement over our naive algorithm. However, this raises a new problem— we must now decide if the L characters we are looking at are actually in the word list. If we were to use a simple linear search we'd be in trouble all over again, e.g. our 173,000 words would have to be searched each time so we'd be right back up to a huge number. Fortunately there is a better way. Assuming our dictionary of words is sorted, we can use binary search! Even for our huge dictionary, we have only about 17 (lg 173,000) comparisons per check. That would make our total roughly 2,800,000 comparisons. This is about 400 times better than the naive algorithm.

But, we can do even better. Suppose we are searching from some location in some direction on the board and we have the partial character sequence **qx**. There are no English words that begin with **qx**. That means that there is no point in looking at longer sequences that extend **qx**. We can terminate the search from that location in that direction immediately. Here's a new version of our algorithm:

```
for each row R in the puzzle
     for each column C in the puzzle
          for each direction D
               for each word length L
                    check if L chars starting at R,C
                    in direction D form a word
                    if they do not form a prefix,
                         break out of the innermost loop
```

We can check for a prefix using a modified version of the binary search algorithm. In this version we compare the prefix we are looking for to the beginning of a string in the dictionary. For example, if we are looking for the prefix "**que**" and are at the part of the dictionary with the word "**quest**" we would compare "**que**" to the first three letters of "**quest**" and we would find that they are equal. So, in this case, we would determine that "**que**" is, indeed, a prefix.

The Assignment

Your program should be capable of solving the Word Search problem for general puzzles. It should:

1. Get the dictionary file name, create the dictionary, and print its size.

2. Get the puzzle file name, read the puzzle, print its dimensions, and print the puzzle (in upper-case, but characterss in the array your algorithms actually use should be lower-case, since words in the dictionary are lower-case).

3. Ask for and read in the minimum word length to search for.

4. Conduct each search and display the results of each, including coordinates of words found, number of words found, and solution time.

Sample output for a run with puzzle file **p2.txt** (which I have given you) and a minimum word length of 8 appears below.

TestWordSearch

I have provided a **TestWordSearch** class, which is complete ***and should not be changed***.

WordSearch

The **WordSearch** class will hold the current instance of a puzzle. Your class should provide three ways to solve the puzzle corresponding to the three algorithms described above. I have provided a **WordSearch** class that contains some code, along with descriptions of the methods you should implement. ***You cannot change any of this unless you clear it with me first.***

Dictionary

You can use your **Dictionary** class from the Boggle lab. All you need to do is add an **isPrefix** method to check if a given string is a prefix of a word in the dictionary. The dictionary I have given you, **d.txt**, is the same dictionary you used for Boggle.

Sample Output

Here is sample output for a run with the dictionary and puzzle file **p2.txt** that I have given you and a minimum word length of 8:

Welcome to Word Search!

Enter dictionary file name without the ".txt" extension: d
Dictionary size is 172824"

Please enter the puzzle file name without the ".txt" extension: p2

The Board:
```
O B H E N L X I U Q O U C K V
O T F C R S I S V Z U K O O O
Q C F J Y E E M O P L Y F C U
L M W K N A R R A Y T S J Q D
D Q V B P Z O A J W J M P M V
N Y Q V L E N G T H A Y W C M
F L S W X B N L Z T A R U A D
H U C S U B S T R I N G R S O
Q P N R K N R I Z N M G S U P
R M Z C N D X C T N O X T S O
V X A R T R M A E R T S R L L
Y E D J X I H X P J S Y I S Q
N D W V G Y O E H F K T N T E
F N K V M B I N Z C M Q G K A
Y I E L E C T R O N I C Y X A
```

Read puzzle with 15 rows and 15 columns
Minimum word length: 8

Using naive approach:
Found "electron" at (14,2) to (14,9)
Found "electronic" at (14,2) to (14,11)
Found "function" at (6,0) to (13,7)
Found "substring" at (7,3) to (7,11)
Found 4 matches
Solution time: 0.168043736 seconds.

Using binary search approach:
Found "function" at (6,0) to (13,7)
Found "substring" at (7,3) to (7,11)
Found "electron" at (14,2) to (14,9)
Found "electronic" at (14,2) to (14,11)
Found 4 matches

Solution time: 0.007857373 seconds.

Using binary search approach with prefixes:
Found "function" at (6,0) to (13,7)
Found "substring" at (7,3) to (7,11)
Found "electron" at (14,2) to (14,9)
Found "electronic" at (14,2) to (14,11)
Found 4 matches
Solution time: 0.006635879 seconds.

Note that for small puzzles the difference between the second and third methods can be very small. In fact, in one of my runs on the puzzle above, the third method actually took *longer!* Note that the difference will become more pronounced as the minimum word length becomes smaller, because there will be more strings in the puzzle that you are looking for in the dictionary. This brings up two points that can have a significant impact on your run times:

1) In the naive approach, do not bother to look for a word from the dictionary in the puzzle if it is less than the minimum word length.

2) In the other two approaches, do not check to see whether a string of characters in the puzzle is in the dictionary until its length is at least the minimum word length.

Also, it is not necessary that your runtimes be the same as mine. The important thing is that the relative run times of the three methods be similar.

You should work in a modular fashion. Test things individually. I particularly recommend that you test your **isPrefix** method thoroughly before you try to implement the three search algorithms When you do start implementing the three algorithms, ***do your testing with small files,*** e.g. a dictionary of about 10-12 words and a very small puzzle, say 4 rows and 4 columns.

The file **p2.txt** contains the puzzle solved. I have also given you a file containing the output of a run with the same puzzle with a minimum word length of 6. Although I have provided you with a sample puzzle file, ***your program should work with any reasonably sized file***, where “reasonable” means up to 50 rows and 50 columns.

Submitting Your Program

Please submit your program in the usual way on BlackBoard and turn in a hardcopy.