Assignment 6: Search with Backtracking

Assigned: Wednesday, November 5, 2014
Due: Monday, November 17, 2014, 11:59 P.M.
Type: Individual

Problem Description

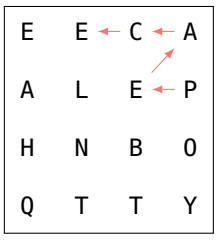
In this assignment, you will implement a version of a word search game much like Boggle and other popular word games. Many games like this are available as apps in the App Store and/or Google Play. In fact, a 2210 alum transformed a similar assignment into an iOS app and made it available on the App Store (http://appstore.com/wareaglewords). So do your work well and might want to publish it for the world!

The game that you will implement is played on a square board according to the following rules.

1. Each position on the board contains one or more uppercase letters.
2. Words are formed by joining the contents of adjacent positions on the board.
3. Positions may be joined horizontally, vertically, or diagonally, and the board does not wrap around.
4. No position on the board may be used more than once within any one word.
5. A specified minimum word length (number of letters) is required for all valid words.
6. A specified lexicon is used to define the set of all legal words.

Below, from left to right, is *(i)* a sample 4 × 4 board with single letters in each position, *(ii)* a sequence of positions forming the word PEACE, and *(iii)* the list of all words with a minimum length of 5 found on the board using the words in the standard Unix /usr/share/dict/words file as the lexicon.

E	E	C	A
A	L	E	P
H	N	B	O
Q	T	T	Y




```
ALBEE  ALCAE   ALEPOT
ANELE  BECAP   BELAH
BELEE  BENTHAL BENTY
BLENT  CAPEL   CAPOT
CENTO  CLEAN   ELEAN
LEANT  LENTH   LENTO
NEELE  PEACE   PEELE
PELEAN PENAL   THANE
TOECAP TOPEE
```

The following words are in that lexicon but *not* on the board: PLACE (rule 2), POPE (rule 4), PALE (rule 3). Although the word BOY can be constructed according to rules 2 and 3, if it does not appear in the lexicon being used (rule 6) or if it does not meet the specified minimum word length (rule 5), it would not be considered a word.

For a word to be *valid* it must be constructed in accordance with all six rules above. A score for each valid word is calculated as follows: one point for the minimum number of characters, and one point for each character beyond the minimum number. Thus, the set of all words of length five or more from the board above would earn a score of 31.

Implementation Details

You are provided with two Java files that you must use to develop your solution: `WordSearchGame.java` and `WordSearchGameFactory.java`. The files are presented in brief below and their methods are discussed in separate sections.

```
public interface WordSearchGame {
   void loadLexicon(String fileName);
   void setBoard(String[] letterArray);
   String getBoard();
   SortedSet<String> getAllValidWords(int minimumWordLength);
   int getScoreForWords(SortedSet<String> words, int minimumWordLength);
   boolean isValidWord(String wordToCheck);
   boolean isValidPrefix(String prefixToCheck);
   List<Integer> isOnBoard(String wordToCheck);
}
```

```
public class WordSearchGameFactory {
   public static WordSearchGame createGame() {
      // return an instance of your game playing engine
   }
}
```

The interface `WordSearchGame` describes all the behavior that is necessary to play the game. So, we can think of it as the specification for a game engine. You must develop your own game engine that meets this specification; that is, you must write a class that implements the `WordSearchGame` interface. You can name this class anything you want and you can add as many additional methods as you like.

The `WordSearchGameFactory` is a class with a single *factory* method for creating game engines. You must modify the `createGame` method to return an instance of your class that implements the `WordSearchGame` interface. Factory classes like this are convenient ways of completely separating an implementation (your class) from a specification (the provided interface). So, the test suite used for grading can be written completely in terms of the interface and without any knowledge of the specific classes used in the implementation.

Lexicon Operations

Three of the methods in the `WordSearchGame` interface relate to loading and searching a lexicon. If the game is to run with reasonable response times with large lexicons, all three methods must be efficient. Of the three, however, `isValidWord` and `isValidPrefix` will see heavy use by the game and are the most important to optimize. With this in mind, you must choose an appropriate data structure to efficiently support how your game engine uses these three methods.

The `loadLexiconn` method loads the lexicon from a file of strings into the data structure that you select. The `isValidWord` method searches this data structure for a particular word, and the `isValidPrefix` method searches this data structure to determine if any word in the lexicon begins with the specified string.

You will notice that many of the words in the provided lexicon files (below) are in lowercase while the game board is in uppercase. Be sure that you address this mis-match, since uppercase letters are considered less than lowercase letters for comparison purposes. That is, 'A' $\neq$ 'a'.

Board Operations

The `setBoard` method accepts a String array of length N^2 that specifies the layout of letters on the $N \times N$ board. The elements of the array specify the letters found on the board in *row-major* order. Thus, the elements in the array from index 0 to $N^2 - 1$ correspond to the positions on the board from left to right, top to bottom. The element at index 0 stores the contents of the board position (0,0) - the upper left corner - and the element at index $N^2 - 1$ stores the contents of the board position (N - 1, N - 1) - the bottom right corner. For example, the array `a = ["E", "E", "C" "A", "A", "L", "E", "P", "H", "N", "B", "O", "Q", "T", "T", "Y"]` would correspond to the board below.

E	E	C	A
A	L	E	P
H	N	B	O
Q	T	T	Y

From this array of strings, you must construct a data structure that represents the board in a way that efficiently supports your board search algorithms. Once again, the choice of data structure should be made in support of the algorithms that require its use.

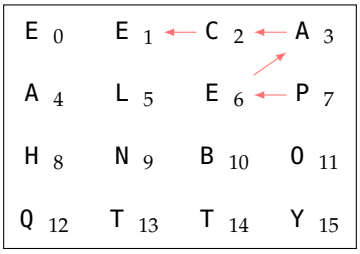
The implementing class of the `WordSearchGame` interface must have a default board. Specifically, the above board should be set as the default so that it is available for game play even if the `setBoard` method has not been called.

The `getBoard` method returns a string representation of the current board that would be suitable for printing to standard out (i.e., as an argument to `System.out.println`). There is no particular format required. Choose the format that is most helpful to you.

Word Search Operations

The `getAllValidWords` method returns a `SortedSet` of strings containing all words on the board that can be constructed according to the specified game rules. If no words can be found, this method returns null. The search algorithm that you use to find all valid words must be efficient enough for use on large game boards with large lexicons. You will lose points for excessively inefficient algorithms.

The `isOnBoard` method takes a string argument and determines if that string can be found on the board. There is no requirement that the string be in the lexicon or have a minimum length. There is, however, the requirement that it be constructed according to rules 2, 3, and 4. If it isn't possible to find the word on the board, this method returns null. If it is possible to find the word on the board, this method returns a `List` of `Integers` representing the sequence of positions on the board used to form the word, using the same mapping from integers to positions required by `setBoard` (row-major). For example, the board below has each position annotated with the associated integer and thus the `isOnBoard` method would return the sequence 7, 6, 3, 2, 1 for the word PEACE.



The `getScoreForWords` methods returns the cumulative score of all the valid words in the given set.

Lexicon Files

You have been provided with several word list files for creating lexicons.

- `CSW12.txt` - 270,163 unique words, used in international Scrabble tournaments
- `OWL.txt` - 167,964 unique words, used in North American Scrabble tournaments
- `words.txt` - 234,371 unique words, provided with Unix distributions
- `words_medium.txt` - 172,823 unique words, subset of the Unix list
- `words_small.txt` - 19,912 unique words, small subset of the Unix list

Your solution should run efficiently with each of these files used for the lexicon.

Submission

Submit your solution via WebCAT no later than the date and time specified. Turn in the `WordSearchGameFactory` class and all classes that you create. **Do not** turn in the `WordSearchGame` interface and **do not** turn in any lexicon files.

Acknowledgements

Word search games of various sorts are popular CS 2 assignments because they bring together several important topics all in one place. This incarnation of the word search problem owes thanks to (at least): Julie Zelenski, Owen Astrachan, and Mike Smith.