Depth-First Search with Backtracking

COMP 2210 Assignment

Problem Description

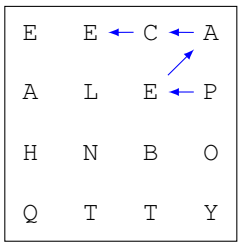
In this assignment, you will implement a version of a word search game much like Boggle[1] and other popular word games. The approach you take to finding words on the board will be a direct application of *depth-first search* with *backtracking*.

The version of 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. (22 words of length 5, 3 words of length 6, and 1 word of length 7 give 22 + 6 + 3 = 31 points.)

[1] `https://en.wikipedia.org/wiki/Boggle`

Implementation Details

Your must implement your solution to the assignment in terms of two provided files: `WordSearchGame.java` and `WordSearchGameFactory.java`, plus one class that you create from scratch all on your own. The interface `WordSearchGame` describes all the behavior that is necessary to play the game. So, we can think of this interface 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.

A brief example client along with the corresponding output are given below. Although the class that implements the `WordSearchGame` interface must be in the same directory, the `ExampleGameClient` code is independent of its name or any other details of the class.

```
public class ExampleGameClient {
   public static void main(String[] args) {
      WordSearchGame game = WordSearchGameFactory.createGame();
      game.loadLexicon("wordfiles/words.txt");
      game.setBoard(new String[]{"E", "E", "C", "A", "A", "L", "E", "P", "H",
                                 "N", "B", "O", "Q", "T", "T", "Y"});
      System.out.print("LENT is on the board at the following positions: ");
      System.out.println(game.isOnBoard("LENT"));
      System.out.print("POPE is not on the board: ");
      System.out.println(game.isOnBoard("POPE"));
      System.out.println("All words of length 6 or more: ");
      System.out.println(game.getAllValidWords(6));
   }
}
```

Output:

```
LENT is on the board at the following positions: [5, 6, 9, 14]
POPE is not on the board: []
All words of length 6 or more:
[ALEPOT, BENTHAL, PELEAN, TOECAP]
```

`WordSearchGame` methods relating to the lexicon

The three methods in the `WordSearchGame` interface that relate to loading and searching the lexicon are `loadLexicon`, `isValidWord`, and `isValidPrefix`. While `loadLexicon` is called only once per game, `isValidWord` and `isValidPrefix` will be called heavily throughout game play. These methods must be efficient if the game is to run with reasonable response times when using large lexicons. The choice of collection or data structure to store the lexicon will determine just how efficient these methods can be. You have two basic choices to represent the lexicon: use a prebuilt collection from the JCF or implement your own custom data structure or collection for the purpose. If you choose the former option, `TreeSet` will offer very good performance and provide very convenient methods. If you choose the latter option, a *trie* will provide even better (time) performance and will be a fun challenge to implement. The choice is completely up to you. A good (and optional) idea would be to develop your solution with a `TreeSet` to store the lexicon and then, if you have plenty of time at the end, make an attempt at building your own trie.

The `loadLexicon` method reads a list of words from a text file and stores each *unique* word in the data structure or collection that you select to represent the lexicon. You will notice that many of the words in the provided lexicon files are in lowercase while the game board is in uppercase. Be sure that the lexicon is loaded and all string comparisons are made in a case-insensitive manner. You will also notice that the provided lexicon files have different content and formats. Using a simple scanner, however, it is possible to use the same code to read in all the provided lexicon files.

Note that the lexicon must be loaded before calling many of the other `WordSearchGame` methods. If any method that is dependent on a lexicon is called before `loadLexicon`, your code must throw an `IllegalStateException`. See the source code documentation for details.

The `isValidWord` method searches the data structure that holds the lexicon for a specified string and indicates whether or not that string is present. If the string is present then it is a valid word. If the string is not present then it is not a valid word.

The `isValidPrefix` method searches the data structure that holds the lexicon to determine if any word in the lexicon begins with the specified string. For the purposes of this method, a string should be considered a prefix of itself. For example, "cat" is a prefix of "cat" as it is of "catalog".

`WordSearchGame` methods relating to the board

The `setBoard` method accepts an array of length N^2 that specifies the content of each position on the $N \times N$ board. The elements of the array are the Strings on the board listed 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. In general, the String at index $p = row \times N + col$ is the content of board position (row, col).

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 show below on the left. The same board is shown below on the right but with each element annotated with its row-major position. Note that the letter `N` is at board position (2, 1), which corresponds to row-major position $2 \times 4 + 1 = 9$.

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

E 0	E 1	C 2	A 3
A 4	L 5	E 6	P 7
H 8	N 9	B 10	O 11
Q 12	T 13	T 14	Y 15

The primary responsibility of the `setBoard` method is to populate the data structure that you choose to represent the board with the contents of the provided array of Strings. Once again, the choice of data structure to represent the board should be made in support of the algorithms that will depend on it (see game play methods below). Two obvious data structure choices include *(a)* just keeping the one-dimensional array of strings as the board representation or *(b)* creating a two-dimensional array of strings to directly represent the two-dimensional board being modeled. Other choices are possible, and the one you pick is at your discretion.

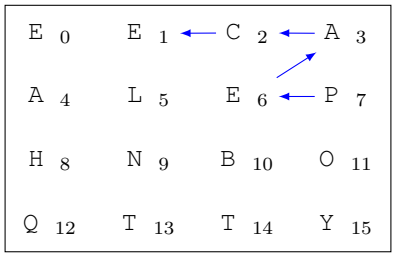
The implementing class of the `WordSearchGame` interface must have a default board. Specifically, the above board should be set as the default so that it is available for game play even if the `setBoard` method has not been called.

The `getBoard` method returns a string representation of the current board 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. This method will not be tested or graded.

`WordSearchGame` methods relating to game play

The methods that implement game play options are `getAllValidWords`, `isOnBoard`, and `getScoreForWords`. The `getAllValidWords` method returns a `SortedSet` of strings containing all words on the board that are of a specified minimum length and can be constructed according to the game rules. If no words can be found, this method returns an empty `SortedSet`.

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 is possible to find the string on the board, this method returns a `List` of `Integers` representing the row-major positions of each substring[2] in the individual board positions used to construct the string. For example, `isOnBoard("PEACE")` would return the list [7, 6, 3, 2, 1].



Both `getAllValidWords` and `isOnBoard` can be implemented using slightly different versions of depth-first search. The specific depth-first algorithms that you implement must be efficient enough for use on large game boards with large lexicons.

The `getScoreForWords` method returns the cumulative score of all the `scorable` words in the given `SortedSet`. To be scorable, a word must (1) have at least the minimum number of characters, (2) be in the current lexicon, and (3) be on the current board. Each scorable word of length $K > M$ is worth $1 + (K - M)$ points, where M is the specified minimum length.

Provided word list files for the lexicon

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.

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.

[2]Since the board positions contain strings and not just single characters, the `List` returned by `isOnBoard` contains strings.