Lab Report 12

15 January, 2019

Justin Voitel,Cong Nguyen-Dinh,Hermes Rapce

Scrabble Cheater - Basic Edition

Task 1.1

As in some Labs before, we will use a BufferedReader(with a FileReader) to read every line in a file until our line variable is null, that indicates that there is no more line to read:

```
public WordList initFromFile(String fileName) {
  String line;
  try {
     BufferedReader bufferedReader = new BufferedReader(new FileReader(fileName));
     while((line = bufferedReader.readLine()) != null) {
        wordList.add(line);
     }
     bufferedReader.close();
  } catch (IOException e) {
     e.printStackTrace();
  }
  return new SimpleWordList();
}
```

For now we will save the lines in a ArrayList of the type String

Task 1.2

To make a string normalized we have to sort its chars. First we save every char of the string in a char Array, by using the String.toCharArray() method. Then we can use the Arrays.sort() method to bring each char in its normalized position and then return a new String Object, by giving the sorted char Array to the constructor:

```
public String getNormalized() {
  char[] chars = word.toCharArray();
  Arrays.sort(chars);
  return new String(chars);
}
```

For the equals() method, we first make an instanceof check if the Object can be casted to a Permutation Object. If so, we cast it to

be a Permutation object (perm1) and returning the result of both normalized Strings in a equality check:

```
public boolean equals(Object obj) {
  if(obj instanceof Permutation){
     Permutation perm1 = (Permutation)obj;
     return this.getNormalized().equals(perm1.getNormalized());
  }else{
     return false;
  }
}
```

When we would run the test cases from PermutationTest, all cases would succeed

Task 2.1

We chose to recursively call a method to add all possible tile rack part combinations into a seperate HashSet of type String:

```
private void permutation(String perm, String word) {
  if (word.isEmpty()) {
     if(wordList.contains(perm+word)){
        validSet.add(perm + word);
     }
  } else {
     for (int i = 0; i < word.length(); i++) {
        permutation(perm + word.charAt(i), word.substring(0, i) + word.substring(i + 1,
word.length()));
     }
  }
}
```

When we first call the method above, we have to pass an empty String for the “perm” attribute(we can see this attribute as a locked/already been through part of the permutation):

```
permutation("", tileRackPart);
```

, whereas we pass the full tile rack part to the “word” attribute. Now we call a for-loop for every letter from the “word” parameter, where we recursively call the same method, where the new perm attribute will be the old perm + the “left” part of the substring of the char position(inclusive the char position). The new “word” attribute will be the “right” part of the substrong from the char position of the loop. This continues until we reached the end of the word string(it will be empty). Now we check if the permutation is valid by checking if it contains in the wordList

Task 2.2 & Task 2.3

We didn´t really understood what we had to do in these tasks. We know our code is not the most optimal and has a exponential growth in time as longer the string gets, but with our capabilities there wasnt a better way to do it

Task 2.4

We added a simple main method to the SimpleWordList Class where we first create a object of itself and then run the initFromFile() method to fill in the sample word data. Then we call the validWordsUsingAllTiles() method with a String of our choice tests have been shown, that our algorithmus needs too long if the length of the string is longer than 7.

For a sample string “gersind”, we could create 2 words from the list:

```
count: 2
valid list: [engirds, dingers]
time: 6.0s
```