Overview

In this first CS1003 practical, I was asked to write a program that performs spelling correction. To correct the misspelt words, I had to compare them to words (spelt correctly) in a dictionary. To do that, I had to read the external text file words_alpha.txt, and store it in a suitable structure. In order to compare words, I had to decompose them into character *bigram*, and calculate the Jaccard similarity index between two sets of *bigram*. Finally, I had to run different tests to check that my program was fully functional.

I'm proud to say that I've achieved the previously established goals:

- I've imported, read and stored the dictionary
- The program can get the character bigram for a word
- The code is capable of calculating the Jaccard similarity index of two sets
- All tests are successful (see **Testing** for more details)

Design

The project is made up of one Java program, CS1003P1.java, which contains four methods:

- A *main* method that is executed when the program is run.
- A *run* method, which contains the "important" code to the spelling correction.
- A *calculateBigram* method, that takes a word as input and returns its character bigram.
- A *calculateJaccard* method, which computes the Jaccard similarity index between two inputted sets.

I will now describe all four methods in more detail.

In the *main* method body, I've created two objects: a CS1003P1 object called *practical*, and a StringTokenizer object called *st*. To do so, I had to import the StringTokenizer library using the *import java.util.StringTokenizer;* syntax. *st* reads and stores the single parameter inputted by the used when the program is run. As the input is usually a sentence, the delimiter used to separate the different words is the space character. After storing the parameter, I run the *run* method to perform the spelling correction, with *st* as parameter.

```
1   import java.util.StringTokenizer;
2   import java.io.IOException;
3   import java.nio.charset.StandardCharsets;
4   import java.nio.file.Files;
5   import java.nio.file.Path;
6   import java.nio.file.Paths;
7   import java.util.List;
8   import java.util.HashSet;
9
10  public class CS1003P1 {
11      public static void main(String[] args) {
12          CS1003P1 practical = new CS1003P1();
13          StringTokenizer st = new StringTokenizer(args[0], " "); // Read the single parameter
14          practical.run(st);
15      }
```

Figure 1: The imports and the *main* method

The *run* method takes a StringTokenizer object as input (which basically is the single parameter inputted by the user when he/she runs the program), and checks the sentence, corrects any spelling mistakes if necessary, and prints out the result.

The method begins with accessing the dictionary: I created a Path object called *myPath* to store the words_alpha.txt file's path. To do so, I had to import the Path and Paths libraries (see Figure 1). The next step was to store the words in a Java List object: I used the *readAllLines* method from the Files library to create a new List of Strings called *dictionary*. The syntax used is *List<String> dictionary = Files.readAllLines(myPath, StandardCharsets.UTF_8)*, and the libraries involved are StandardCharsets, Files, and List (all imported in Figure 1). However, when executed, this returns an IOException error, which is why I used a *try… catch…* check to catch the IOException error and ignore it.
After reading the text file and storing its data in a suitable data structure, I cycled through every word in the user-inputted sentence. This translates to looking at every different token in the StringTokenizer object. To do so, I employed a *while* loop that runs until *st* doesn't have any tokens left. I then stored the "current" token in a String variable, to facilitate the rest of the code. I then had to check if the token is spelt correctly, a.k.a. if it is already in the dictionary. If that's the case, then no spelling check is necessary, and we can just print the word as it is. If not, then we will have to correct it. I used an *if… else…* conditional statement: *if (dictionary.contains(token.toLowerCase()))* – the token has to be in lower case, as all words in the dictionary are stored in lower case.
If this check returns False, we move to the *else* part of the statement. The first step in correcting the misspelt word is to compute its character bigram. According to the System Specification, a bigram does not accept duplicates. This is why I created a HashSet object called *tokenBigram* to store the token's bigram. I had to import the HashSet library (see Figure 1) and use the *calculateBigram* method defined later (see Figure 3) with the token as input.
After obtaining the token's character bigram, I had to cycle through every word in the dictionary, calculate their bigram, and compare them using the Jaccard similarity index. Beforehand, I created two variables: *max* and *closest*, that store the highest Jaccard similarity index and the closest word respectively. To check every word in *dictionary*, I used the *for (String word : dictionary)* syntax. In this *for* loop, I calculated the word's bigram, storing it in another HashSet object called *wordBigram*, and the Jaccard similarity index with the *calculateJaccard* method (see Figure 4), saving the value in the *similarity* variable. A quick *if* statement follows to check if the obtained similarity index is higher than the previous maximum: if so, we update both the *max* and *closest* variables with the new index, and the new word. We now have the corrected word, and we can print it out!
When the *while* loop stops, we will have checked every word in the user-inputted parameter, corrected it – if necessary – and printed it out. For aesthetics and formatting reasons, we print out an empty line at the end of the code.

```
17  public void run(StringTokenizer st) {
18      Path myPath = Paths.get("words_alpha.txt"); // Access the dictionary
19      try {
20          List<String> dictionary = Files.readAllLines(myPath, StandardCharsets.UTF_8); // Store the dictionary in a List object
21          while (st.hasMoreTokens()) {
22              String token = st.nextToken();
23              if (dictionary.contains(token.toLowerCase())) { // Check if the word is spelt correctly, a.k.a, it exists in the dictionary
24                  System.out.print(token + " ");
25              }
26              else { // Find the most similar string from the dictionary
27                  HashSet<String> tokenBigram = new HashSet<String>(calculateBigram(token));
28                  double max = 0.0;
29                  String closest = "";
30                  for (String word : dictionary) { // Cycle through every word of the dictionary
31                      HashSet<String> wordBigram = new HashSet<String>(calculateBigram(word));
32                      double similarity = calculateJaccard(wordBigram, tokenBigram);
33                      if (similarity > max) {
34                          max = similarity;
35                          closest = word;
36                      }
37                  }
38                  System.out.print(closest + " ");
39              }
40          }
41      }
42      catch (IOException e) { // Line 20 returns an IOException error - we have to ignore it
43          e.printStackTrace();
44      }
45      System.out.println();
46  }
```

Figure 2: The *run* method

One might ask: "Why did you create an extra method, *run*, when you could've put all of this code in the *main* method?" This is a problem that I encountered while doing my CS1003P1 code. At first, I **did not** implement a *run* method, all of the "core" code was in the *main* method. However, to access *calculateBigram* and *calculateJaccard* from the static *main* method, I had to make the other two methods static, which wasn't necessary. Therefore, to resolve this issue, I created an "intermediate" method, *run*.

We now move on to the third method in the CS1003P1.java file: *calculateBigram*. What this method does is relatively explicit: it takes a word (String) as argument, and returns a HashSet object, its bigram.
As required by the System Specification, before calculating the word's bigram, I implemented the top-and-tail method to "improve string similarity search". I simply updated the word by concatenating the "^" symbol at the beginning and the "$" symbol at the end.
To store the bigram, I created an empty HashSet called *bigram*. I then cycled through every pair of characters in the word, converted both characters to Strings, concatenated them and added the pair to the HashSet. When the loop ends, the HashSet *bigram* contains the character bigram of the inputted word, and the method returns the bigram.

```
48      public HashSet<String> calculateBigram(String word) {
49          word = "^" + word + "$"; // Add the top-and-tail
50          HashSet<String> bigram = new HashSet<String>(); // The bigram is a HashSet object, to avoid repeating sequences
51          for (int i=0; i<word.length()-1; i++) { // Cycle through every charcater of the inputted word, except the final one
52              String a = "" + word.charAt(i);
53              String b = "" + word.charAt(i+1);
54              String sequence = a+b;
55              bigram.add(sequence); // The HashSet automatically checks if the added sequence is already in the bigram
56          }
57          return bigram;
58      }
```

Figure 3: The *calculateBigram* method

The fourth and final method implemented in this first CS1003 practical is *calculateJaccard*. It takes two HashSet objects containing Strings, determines their intersection and union, and returns the Jaccard similarity index.
I firstly created two new HashSet objects: *intersection* and *union*. I then had to check which of the two inputted sets is the largest, in order to use the *retainAll* method optimally. I added the largest set to the intersection and removed all elements that aren't common to both sets, to get the intersection. To compute the union, I followed the formula $|A \cup B| = |A| + |B| - |A \cap B|$. However, as *set1* and *set2* are HashSet objects, adding all elements of *set2* to *set1* does not create duplicates. Therefore, we don't have to remove the intersection to get the union. Finally, I computed the Jaccard similarity index by dividing the size of the intersection by the size of the union, and returned that number.

```
60      public double calculateJaccard(HashSet<String> set1, HashSet<String> set2) {
61          HashSet<String> intersection = new HashSet<String>();
62          HashSet<String> union = new HashSet<String>();
63          if (set1.size() > set2.size()) { // Check which set is the largest
64              intersection.addAll(set1);
65              intersection.retainAll(set2); // Only keep the elements that are common to both sets
66          }
67          else {
68              intersection.addAll(set2);
69              intersection.retainAll(set1);
70          }
71          set1.addAll(set2); // Set1 being a HashSet, adding all elements of set2 does not create duplicates
72          union.addAll(set1); // We do not have to remove the intersection to get the union (c.f. formula |A u B| = |A| + |B| - |A n B|)
73          return (double) intersection.size()/union.size();
74      }
75  }
```

Figure 4: The *calculateJaccard* method

Testing

We can divide the testing of my program in five categories:

1. Testing the Jaccard similarity WITHOUT the top-and-tail.
2. Testing the top-and-tail.
3. Testing the character bigram WITH the top-and-tail.
4. Testing the Jaccard similarity WITH the top-and-tail.
5. Finding any flaws in the Jaccard similarity.

The following five tables describe the different tests used to check the previously established task.

1. Testing the Jaccard similarity WITHOUT the top-and-tail					
What is being tested	Name of test method	Pre conditions	Expected output	Actual output	Evidence
"chips" and "crisps"	*testJaccard* and *testBigramNoTopAndTail*	None	0.125	0.125	The *calculateJaccard* method works WITHOUT the top-and-tail
"cheese" and "cheese"			1.0	1.0	
"Ozgur Akgun" and "Alan Dearle"			0.0	0.0	
"speled" and "spelled"			0.833	0.833	

2. Testing the top-and-tail					
What is being tested	Name of test method	Pre conditions	Expected output	Actual output	Evidence
"Cocoa"	*testTopAndTail*	None	^Cocoa$	^Cocoa$	I have implemented the top-and-tail correctly

3. Testing the character bigram WITH the top-and-tail					
What is being tested	Name of test method	Pre conditions	Expected output	Actual output	Evidence
"cocoa"	*calculateBigram*	None	[^c, co, oc, oa, a$]	[^c, co, oc, oa, a$]	The *calculateBigram* is able to generate an inputted word's bigram
"Antoine"			[^A, An, nt, to, oi, in, ne, e$]	[^A, An, nt, to, oi, in, ne, e$]	
"banana"			[^b, ba, an, na, a$]	[^b, ba, an, na, a$]	

4. Testing the Jaccard similarity WITH the top-and-tail					
What is being tested	Name of test method	Pre conditions	Expected output	Actual output	Evidence
"Antoine" and "Anton"	*calculateJaccard*	None	0.4	0.4	The *calculateJaccard* method also works WITH the top-and-tail
"chips" and "crisps"			0.3	0.3	
"cheese" and "cheese"			1.0	1.0	
"Ozgur Akgun" and "Alan Dearle"			0.0	0.0	

5. Finding any flaws in the Jaccard similarity			
What is being tested	Mistake	Expected output	Actual output
"There are no clouds in the ski"	Misspelt "sky"	"There are no clouds in the sky"	"There are no clouds in the ski"
"I love comuter science"	Forgot the "p" in "computer"	"I love computer science"	"I love commuter science"
"They ate they're lunch"	Used the wrong homonym	"They ate their lunch"	"They ate they're lunch"
"Le ski est bleu"	Literally "The ski is blue" in French	"The ski is blue"	"Le ski est bleu"

Evaluation

I am proud to say that my program works: it can correct misspelt words in an inputted sentence. The methods used – implementing the top-and-tail, calculating a word's character bigram, and computing the Jaccard similarity index between two sets – and the final code have been thoroughly tested.

Throughout the testing, I discovered some flaws to the Practical. Firstly, if the misspelt word is already in the dictionary, then the correction won't work. Secondly, the algorithm doesn't take context into account: using the wrong homonym won't be detected as an error. Finally, some errors drastically reduce the Jaccard similarity index, and my program does not pair the misspelt word with its correct version.

Conclusion

All in all, I have found this assignment a bit challenging at first, as I didn't know what to do. I had absolutely no idea how to read external files or access user parameters, and I had no knowledge in HashSet objects. On top of that, I never encountered the Jaccard similarity index

before. But through extensive research and patience, I figured a way to complete my first CS1003 practical.

If I had some extra time, I would've loved to push the challenge even further, by trying to resolve the issues indicated above. I can imagine that there is a myriad of methods to calculate the similarity between two words, and I would've loved to explore some more efficient that the Jaccard similarity index.