December 3, 2019

Jean-Phillippe Durand

40052679

Concordia University

TEXT CORRECTOR PROJECT REPORT

COEN 352

# Table of Content

# Problem Description:

The goal of the project is to develop a simple text corrector. The final solution shall focus on typographical and orthograph errors. The user shall provide a word bank using a plain text file (.txt), which will be used by the program as a dictionary.

The input file used as a dictionary will contain an unknown number of words; each word (separated by a whitespace character). To prevent long computing times when analyzing the text to be corrected, the dictionary will be stored into a binary search tree (BST). Due to it’s low time complexity in the order of O(LogN), where N is the number of words stored in the BST, the BST will prevent the program from slowing down noticeably when inputting dictionaries containing a large number of words.

The BST created from the user inputted dictionary will constitute the foundation of the text corrector, as all correction operations will require searching in the dictionary for the validity of the word, or the closest match.

The corrector has three (3) functions. It will correct word separation in the case where the user forgot to add space between two know (to the dictionary) words. It will then correct words containing a mutated character (e.g. d=ctionary will be corrected to dictionary) and warn the user of the correction by providing him with the initial word and its correction. For the last step, the corrector will analyse the frequency of know words in the text and output a list of the words found, sorted by their frequency and then alphabetically.

# PROBLEM BREAKDOWN:

**TEXT CORRECTOR WBS**

# SOLUTION & DESIGN:

This section will cover the design and solution for each part of the project; starting with the main function.

## Solution Overview:

A screenshot of a cell phone

Description automatically generated

*Figure 1: High-level Diagram of Information Processing Activities*

The main function of the program takes care of the setup for each processing activity. The program starts by storing the words found in the dictionary.txt file into the Red-Black BST.

All three parts of the project are treated independently, and use their own algorithms, however, all three parts uses the word frequency analysis algorithm. These three parts are run alphabetically and will be covered with more details in the next sections.

## Word Frequency:

The word frequency was designed to meet the objectives of part A of the project but is also used in part B and C after the respective algorithms are run to correct the initial inputted text.

Pseudo-code:

**procedure** AnalyseFile()

**foreach** (*Word* in *InputText*)

**if** Dictionary.contains(*Word*) **then**

**if !***WordHashMap*.contains(*Word*) **then**

*WordHashMap*.put(*Word*, 1)

**end if**

**else then**

*WordHashMap*.put(*Word*, *Value*++)

**end else**

**end if**

**end foreach**

PrintOutputFile(*WordHashMap*, *FileName*)

**end procedure**

**procedure** PrintOutputFile(HashMap *FoundWords*, String *FileName*)

FoundWords.sort()

**foreach**(Pair in FoundWords)

Outputfile.append(*Pair.key* + “ “ + *Pair.value* + “\n”)

**end foreach**

**end procedure**

**procedure** Sort()

**if** *a.value* > *b.value* **then return** 1

**else if** *a.value* < *b.value* **then return** -1

**else if** *a.key* > *b.key* **then return** 1

**else return -**1

**end procedure**

## Whitespace Corrector

The whitespace corrector’s algorithm searches each unknown word to the dictionary for a substring of a word contained in the dictionary. The algorithm only allows for one split; therefore, the best solution will be a split creating to known substrings. If there is no split resulting in two known words, the algorithm will weight each split containing one known word and pick the one with the biggest weight. The weight of a split is determined by the length of the word found. If the unknown substring is smaller than 5 it will be tested against a second dictionary of common words in the English language; words like “the”, “he”, “she” and more. If the word is not found, points will be deducted, if it is found the weight will increase substantially. **RUNTIME**

Pseudo-code:

**procedure** CorrectWhiteSpaces()

**foreach** (word in Input)

**if**(!Dictionary.contains(word)) **then**

CorrectedWords = AnalyseWord(word);

Print(CorrectedWords);

**end if**

**end foreach**

**end procedure**

**procedure** String[] AnalyseWord(String iWord)

String wWord = iWord.toLowerCase()

HashMap PossibleSplits

**for**(int i = wWord.length/2; i < wWord.length; i++)

leftSS = wWord.substring(0, i);

rightSS = wWord.substring(i);

leftHit = Dictionary.contains(leftSS);

rightHit = Dictionary.contains(rightSS);

**if**(leftHit && rightHit) **then**

wOutput[0] = leftSS;

wOutput[1] = rightSS;

return wOutput;

**end if**

**else if** (leftHit && i < wWord.length - 1) **then**

PossibleSplits.put(leftSS, i);

**end else if**

**else if** (rightHit && i < wWord.length - 1) **then**

PossibleSplits.put(rightSS, i);

**end else if**

**if**(j>1) **then**

--j;

leftSS = wWord.substring(0, j);

rightSS = wWord.substring(j);

leftHit = Dictionary.contains(leftSS);

rightHit = Dictionary.contains(rightSS);

**if**(leftHit && rightHit) **then**

wOutput[0] = leftSS;

wOutput[1] = rightSS;

return wOutput;

**end if**

**else if** (leftHit && j > 1) **then**

PossibleSplits.put(leftSS, j);

**end else if**

**else if** (rightHit && j > 1) **then**

PossibleSplits.put(rightSS, j);

**end else if**

**end if**

**end forloop**

**if**(!PossibleSplits.isEmpty()) wOutput = PickBestSplit(PossibleSplits, wWord);

**return** wOutput;

**end procedure**

**procedure** String[] PickBestSplit(HashMap<String, Integer> iOptions, String iWord)

String[] wOutput = new String[2];

int MaxWeight = 0;

int tempWeight, tempSplit;

int SplitIndex = iWord.length();

String SS1, SS2;

boolean SS1Hit;

**for**( Map.Entry<String, Integer> entry : iOptions.entrySet())

tempSplit = entry.getValue();

SS1 = iWord.substring(0, tempSplit);

SS2 = iWord.substring(tempSplit);

SS1Hit = (Dictionary.contains(SS1));

**if**(SS1Hit) **then**

tempWeight = SS1.length() - 2;

**if** (CommonWords.containsKey(SS1) && (!(SS2.length() < 6) || CommonWords.containsKey(SS2))) **then**

tempWeight += CommonWords.get(SS1);

**end if**

**if** (SS2.length() < 4 && !CommonWords.containsKey(SS2)) **then**

tempWeight -= 6;

**end if**

**end if**

**else then**

tempWeight = SS2.length() - 2;

**if**(SS2.length() >=3) **then**

**if** (SS2.substring(SS2.length() - 3).equalsIgnoreCase("ing")) **then**

tempWeight -= 6;

**end if**

**end if**

**if** (CommonWords.containsKey(SS2) && !((SS1.length() < 5) || CommonWords.containsKey(SS1))) **then**

tempWeight += CommonWords.get(SS2);

**end if**

**if** (SS1.length() < 4 && !CommonWords.containsKey(SS1)) **then**

tempWeight -= 6;

**end if**

**end if**

**if**(tempWeight > MaxWeight)

MaxWeight = tempWeight;

SplitIndex = tempSplit;

**end if**

**end if**

wOutput[0] = iWord.substring(0, SplitIndex);

wOutput[1] = iWord.substring(SplitIndex);

**if**(MaxWeight <= 0 || wOutput[0].lastIndexOf('-') == (wOutput[0].length() - 1) || wOutput[1].indexOf('-') == 0) **then**

wOutput[0] = iWord;

wOutput[1] = "";

**end if**

**return** wOutput;

**end procedure**

## Mutated Character Corrector

The mutated character corrector utilizes the Levenshtein distance method to measure the distance between to strings. The Levenshtein distance algorithm finds the minimum amount of insertion, deletion or swaps required to modify the one input string into the reference string. By modifying this algorithm to only allow swaps, when comparing a word unknown to the dictionary to a word included in the dictionary, it can be assumed that the unknown word has been mutated from the reference word in the dictionary and thus, will be replaced in the text. **TRY INSERTION AND DELETION**

Pseudo-code:

**procedure** CorrectMutatedChars()

File file = new File(FilePath);

String[] test = FilePath.split(Matcher.quoteReplacement(System.getProperty("file.separator")));

String FileName = test[test.length-1];

String sts;

String CorrectedWord = "";

**foreach**( word in inputFile)

String wWord = word.toLowerCase();

**if**(!Dictionary.contains(word) && !Dictionary.contains(wWord) && !CommonWords.containsKey(wWord)) **then**

Dictionary.BSTIterator();

**while**(Dictionary.hasNext())

String wKey = Dictionary.next();

**if**(wKey.length() == wWord.length() && wKey.length() > 1) **then**

**if**(LevenshteinDistance.getLevenshteinDistance(wKey, wWord) == 1) **then**

CorrectedWord = wKey;

break;

**end if**

**end if**

**end while**

**if**(CorrectedWord != "") **then**

outSuggestions.println(word +", " + CorrectedWord);

outCorrectedText.append(CorrectedWord + " ");

**end if**

**else then** outCorrectedText.append(word + " ");

**end if**

**else then** outCorrectedText.append(word + " ");

CorrectedWord = "";

**end foreach**

**end procedure**

# Results and Analysis

# User Manual:

The Text Corrector has three functions; using a user provided dictionary, the program can analyse the usage frequency of the words contained in the dictionary in each input text. It can divide unknown words by adding a whitespace to possibly correct a forgotten whitespace between one or two words contained in the dictionary. Finally, the corrector can correct a mutated character within a word known to the dictionary.

SETUP:

1. In the source folder of the text corrector, locate the InputFiles folder.
2. Copy the dictionary you wish to use in the InputFiles folder. Make sure each word is separate by one (1) whitespace character.
3. Copy the texts you wish to analyse or correct in one of the three subfolders (CleanText, RemovedSpaces, MutatedChars). Each folder represents one of the functions described above the setup.
4. Using your preferred IDE, open the source files located in the SourceFiles folder.
5. Run main.java.
6. The results will be saved in the OutputFiles folder.