Energy footprint of the Levenshtein distance computing algorithm

Author :

Salla DIAGNE

16th JANUARY 2016

Table of contents

Introduction

The Levenshtein distance (named after Vladimir Levenshtein) [1] between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other.

Chapter 1

Technical work

1.1 Goal

The goal of this project is to measure energy footprints of different programming languages through execution of programs executing the same task. The results of this experiment can be very beneficial for many people in specific domains, especially in two. First, developers who tend to favour energy-efficiency over performance will know how language to use to produce the most economical program possible. Second, language designers can analyse the metrics and reach conclusions on how a design of a language can be improved or rewritten.

DIAGRAM

1.2 Architecture and design

The design of the program is very simple. In we place ourselves in the context of object-oriented programming (since it is the most understandable form of design by a human), the program is composed of 5 classes (it could even be less than that). Those are depicted in the following UML diagram :

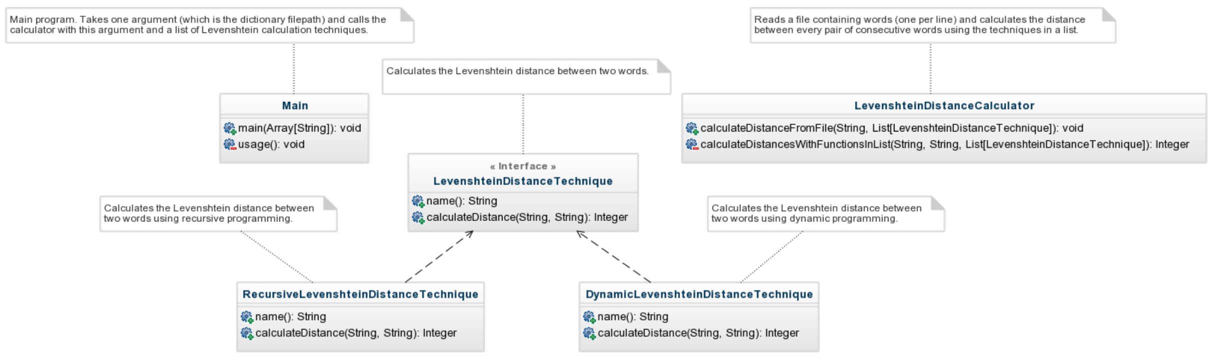

Main program. Takes one argument (which is the dictionary filepath) and calls the calculator with this argument and a list of Levenshtein calculation techniques.
Reads a file containing words (one per line) and calculates the distance between every pair of consecutive words using the techniques in a list.
Calculates the Levenshtein distance between two words.
Main
main(Array[String]): void
usage(): void
LevenshteinDistanceCalculator
calculateDistanceFromFile(String, List[LevenshteinDistanceTechnique]): void
calculateDistancesWithFunctionsInList(String, String, List[LevenshteinDistanceTechnique]): Integer
« Interface »
LevenshteinDistanceTechnique
name(): String
calculateDistance(String, String): Integer
Calculates the Levenshtein distance between two words using recursive programming.
Calculates the Levenshtein distance between two words using dynamic programming.
RecursiveLevenshteinDistanceTechnique
name(): String
calculateDistance(String, String): Integer
DynamicLevenshteinDistanceTechnique
name(): String
calculateDistance(String, String): Integer

Figure 1.1: UML diagram of the algorithm in Scala

1.3 Algorithm

For the credibility of the measures made, the algorithm involves two I/O aspects : file reading, and printing. First, the algorithm performs file reading by retrieving words in a dictionary of words. File reading operations can be an important factor in energy consommation, as well as printing on the standard output.

The pathway of the algorithm is : opening a text file containing thousands of words, read every two consecutive word, print them and print the Levenshtein distances calculated with a recursive technique and a technique using dynamic programming (iterative).

The algorithm is summarized by the following flowchart :

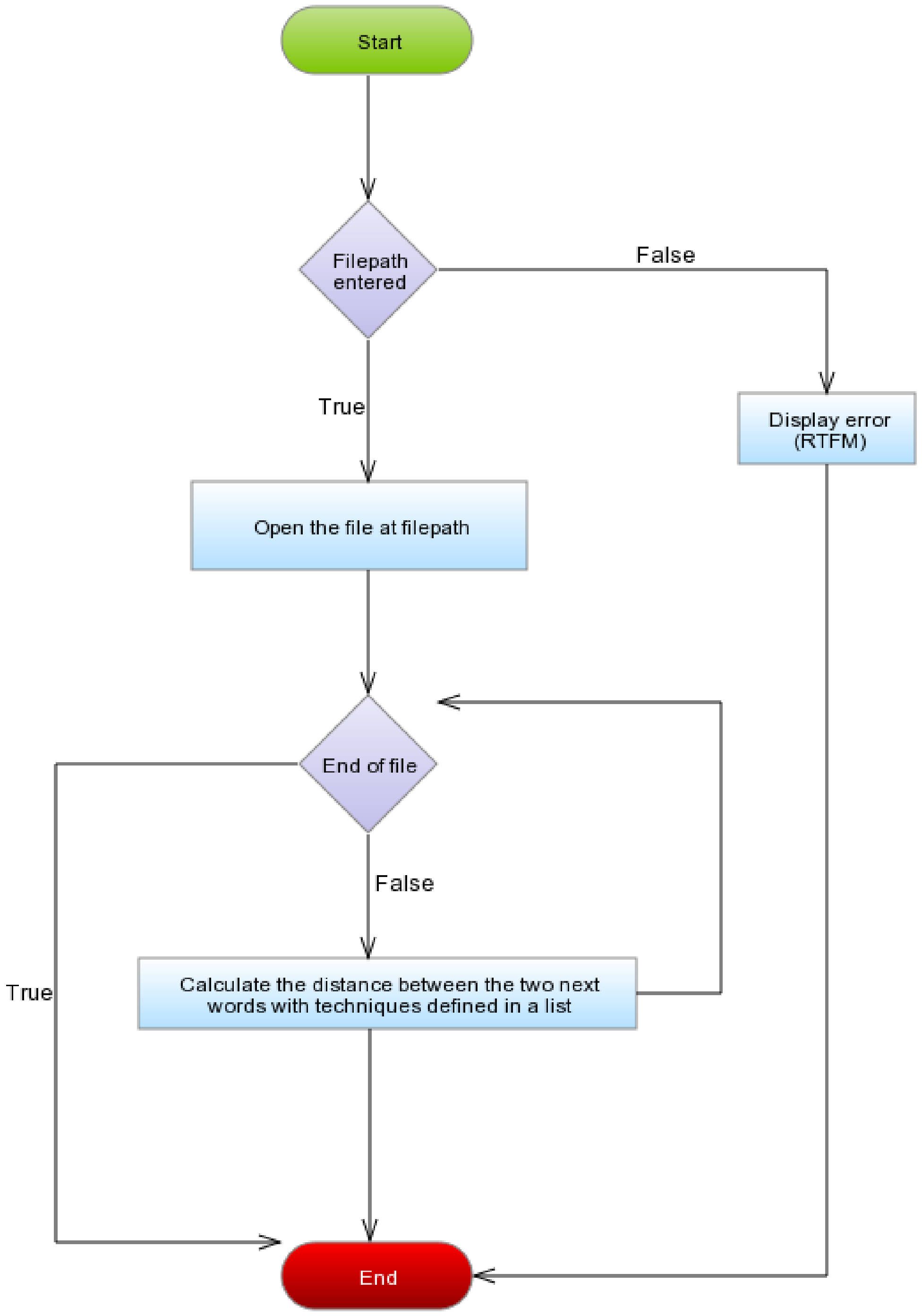

Start
Filepath entered
False
True
Display error (RTFM)
Open the file at filepath
End of file
False
True
Calculate the distance between the two next words with techniques defined in a list
End

Figure 1.2: Flowchart of the algorithm

Two versions of the algorithm have been implemented : a recursive version and a version using dynamic programming (iterative).

Algorithm 1: The recursive version of the Levenshtein distance computing algorithm

```
Function recursiveLevenshteinDistance(word1 : String, word2 : String)
    if word1 == word2 then
        return 0
    lengthWord1 ← word1.length();
    lengthWord2 ← word2.length();
    if lengthWord1 == 0 then
        return lengthWord2
    if lengthWord2 == 0 then
        return lengthWord1
    firstLetterWord1 ← lengthWord1;
    firstLetterWord2 ← lengthWord2;
    subWord2 ← word2.substring(1);
    subWord2 ← word2.substring(1);
    if firstLetterWord1 == firstLetterWord2 then
        return recursiveLevenshteinDistance(subWord1, subWord2)
    else
        return 1 + min(recursiveLevenshteinDistance(subWord1, word2),
          recursiveLevenshteinDistance(word1, subWord2),
          recursiveLevenshteinDistance(subWord1, subWord2)
```

Algorithm 2: The dynamic version of the Levenshtein distance computing algorithm

```
Function dynamicLevenshteinDistance(word1 : String, word2 : String)
    if word1 == word2 then
        return 0
    lengthWord1 ← word1.length();
    lengthWord2 ← word2.length();
    table ← int[lengthWord1 + 1, lengthWord2 + 1];
    if lengthWord1 == 0 then
        return lengthWord2
    if lengthWord2 == 0 then
        return lengthWord1
    for i ← 0 to lengthWord1 do
        for j ← 0 to lengthWord2 do
            if i == 0 then
                table[i][j] = j
            else if j == 0 then
                table[i][j] = i
            else if word1[i - 1] == word2[j - 1] then
                table[i][j] = table[i - 1][j - 1]
            else
                table[i][j] = 1 + min(table[i - 1][j], table[i][j - 1], table[i - 1][j - 1]
    return table[lengthWord1][lengthWord2]
```

1.4 Implementation

The algorithm is implemented in the following languages (in alphabetical order) :

- C
- Go
- Haskell
- Java
- Ocaml
- Python
- Ruby
- Scala

The measurements of energy footprints is done thanks to PowerAPI [2][?], a middleware toolkit for software-defined power meters.

The GitHub repository containing the implementations in the different languages and the results (the files generated by PowerAPI) can be found at the following link : `https://github.com/sallareznov/gc-levenshtein`.

In order to have relevant evaluations, the different program follow the same logic, down to the last semicolon.

1.5 Usage

The utilisation of the implemented programs is quite straightforward. No additional command needs to be entered, because every program has a build file. For example, with the implementation in the Go language, the initial arborescence (the one before compilation) is the following :

```
.
└── src
    ├── levenshtein
    │   ├── dynamic_levenshtein_distance_technique.go
    │   ├── levenshtein_distance_calculator.go
    │   ├── levenshtein_distance_technique.go
    │   └── recursive_levenshtein_distance_technique.go
    ├── levenshtein_test
    │   └── levenshtein_test.go
    └── main
        └── main.go
```

After setting the `$GOPATH` environment variable (needed by the Go compiler to find the module) to the project root folder (`gc-levenshtein/go/`), you have to compile the program by executing the command `go install main`.

Two new folders will be generated : one for the packages (`pkg/`) and another one for the binaries (`main`). In our context, the package is `levenshtein` and the binary is `main`.

```
.
├── bin
│   └── main
├── pkg
│   └── linux_amd64
│       └── levenshtein.a
└── src
    ├── levenshtein
    │   ├── dynamic_levenshtein_distance_technique.go
    │   ├── levenshtein_distance_calculator.go
    │   ├── levenshtein_distance_technique.go
    │   └── recursive_levenshtein_distance_technique.go
    ├── levenshtein_test
    │   └── levenshtein_test.go
    └── main
        └── main.go
```

To run the program, just launch the binary file with the name of the file containing the words as an argument. For example : `./bin/main ../dictionary_EN.txt`. The following output (truncated) will be produced in the standard output :

```
...

Word1 : micropipette
Word2 : microprocessing
Recursive distance : 8
Dynamic distance : 8

Word1 : microprocessor
Word2 : microprocessors
Recursive distance : 1
Dynamic distance : 1

Word1 : microprogram
Word2 : microprogrammed
Recursive distance : 3
Dynamic distance : 3

Word1 : microprogramming
Word2 : microradiographical
Recursive distance : 9
Dynamic distance : 9
```

```
Word1 : microradiographically
Word2 : microradiography
Recursive distance : 5
Dynamic distance : 5

Word1 : micros
Word2 : microscope
Recursive distance : 4
Dynamic distance : 4

...
```

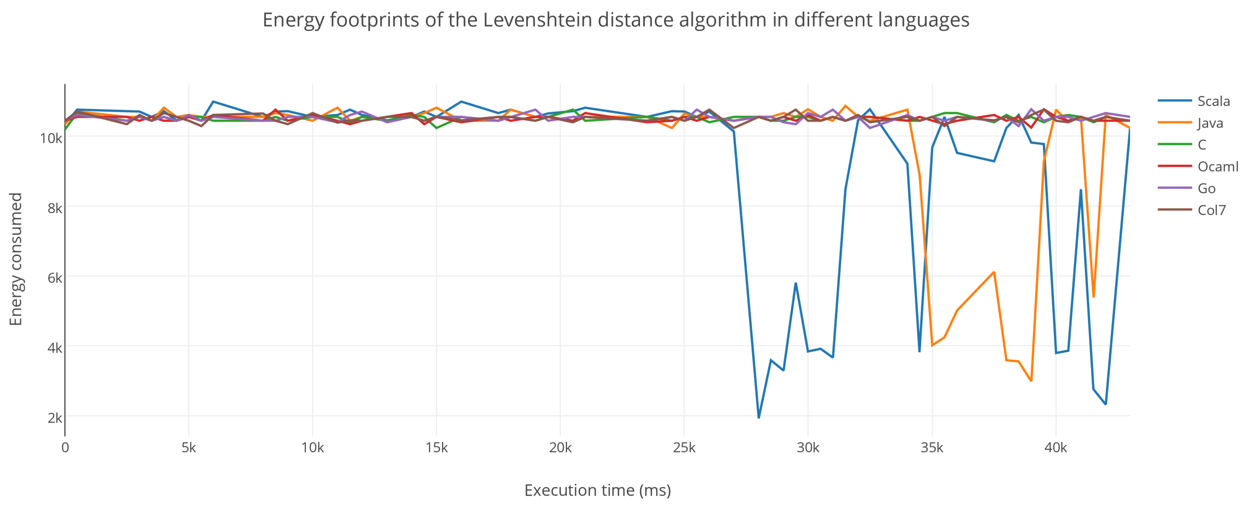
Chapter 2

Evaluation

2.1 Performance

2.2 Validation

The energy footprint of every implemented program has been measured by PowerAPI, and the results obtained are entered in the following graph :



Conclusion

Bibliography

[1] Wikipedia. Levenshtein distance. `https://en.wikipedia.org/wiki/Levenshtein_distance`.

[2] Inria Spirals Team. Powerapi, a middleware toolkit for software-defined power meters.
Official website : `http://powerapi.org`
GitHub : `https://github.com/Spirals-Team/powerapi`
GitHub Wiki : `https://github.com/Spirals-Team/powerapi/wiki`.