
# Energy footprint of the Levenshtein distance computing algorithm

*Author :*

Salla DIAGNE

16$^{th}$ January 2016

# Table of contents

# Introduction

# Chapter 1

# Technical work

## 1.1 Goal

My goal 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.

## 1.2 Overview

## 1.3 Architecture and design

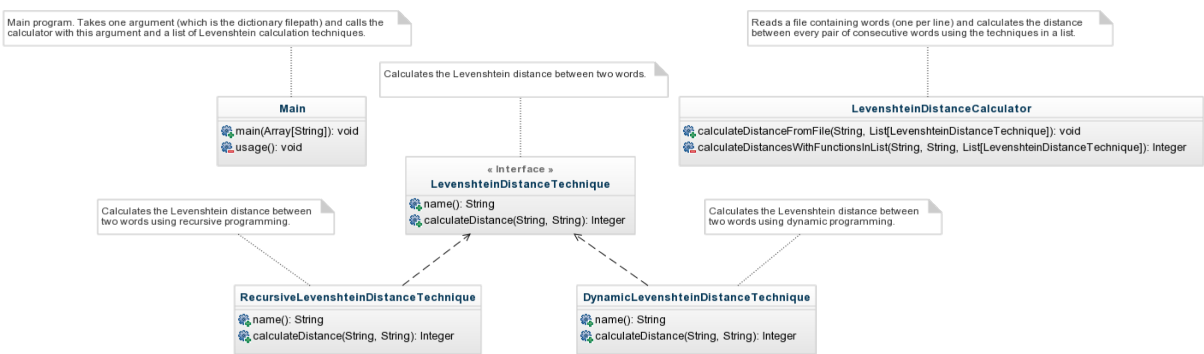


Figure 1.1: UML diagram of the algorithm in Scala

## 1.4 Algorithm

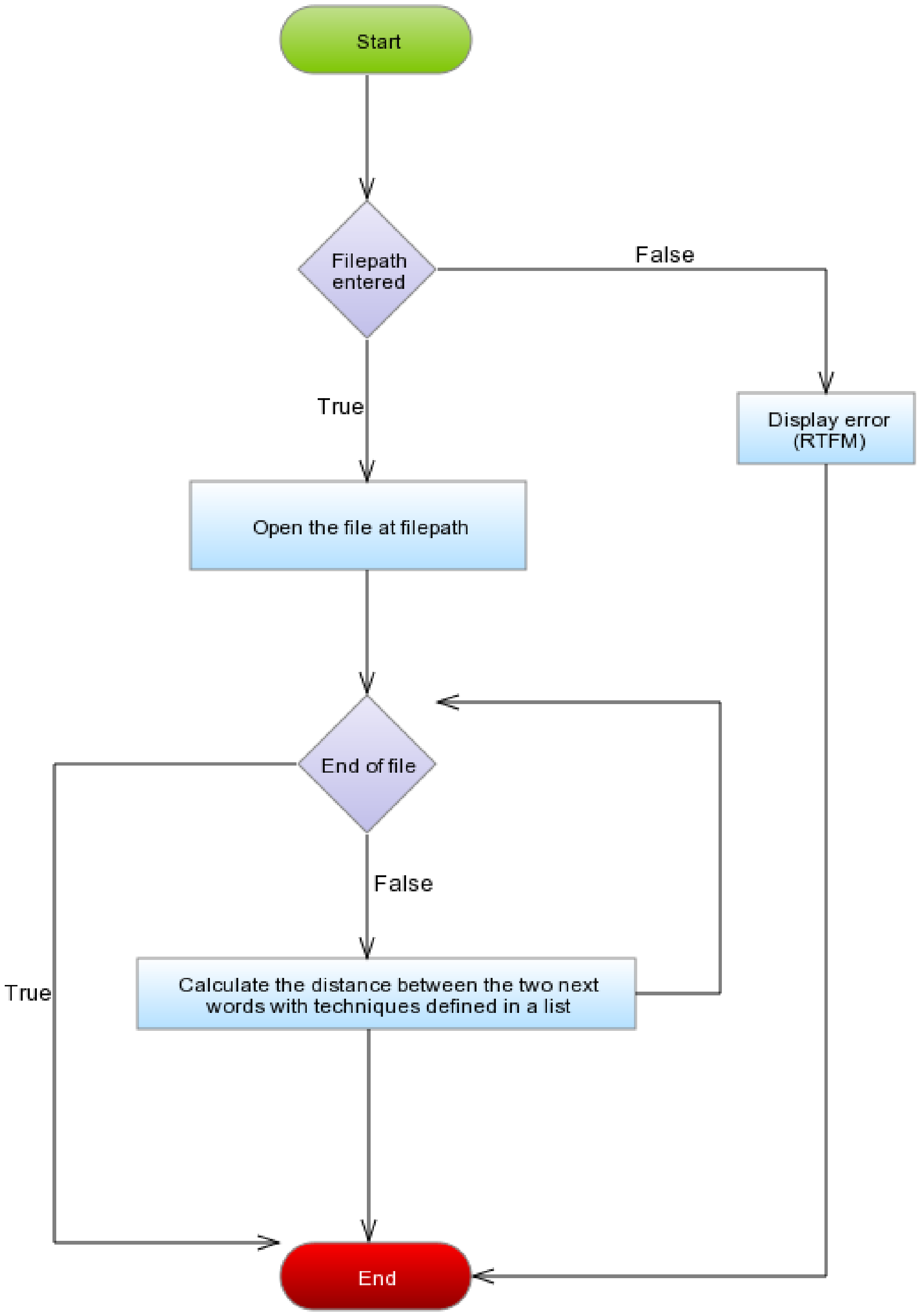


Figure 1.2: Flowchart of the algorithm

**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]
```

## 1.5 Implementation

PowerAPI [1]

## 1.6 Usage

# Chapter 2

# Evaluation

## 2.1 Performance

| Language | Execution Time |
|---|---|
| C | 1 |
| C++ | 1 |
| Go | 1 |
| Haskell | 1 |
| Java | 1 |
| Ocaml | 5m29s |
| Perl (?) | 1 |
| Python | 1 |
| Ruby | 1 |
| Rust | 1 |
| Scala | 45s |
| Smalltalk | 1 |

Figure 2.1: Execution time of the algorithm in the implemented languages

## 2.2 Ease of use

## 2.3 Validation

# Conclusion

# Bibliography

[1] 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.