Counting Distinct Words in A File Using an HashTable

Pedro Marques, 92926

***Resumo* – Este relatório apresenta a contagem de palavras distintas numa obra literária, nomeadamente Oliver Twist de Charles Dickens. De tal forma, foi utilizada uma HashTable implementada em python para averiguar quais palavras existem no ficheiro. Foram também realizados testes com outras linguas, nomeadamente Holandês, Alemão, Inglês e Francês.**

***Abstract* -This report follows the count of distinct words in a given literary work, more specifically, Charles Dickens's Oliver Twist. As such, was implemented a HashTable in python3 in order to count them. This test also involved different languages of the same work, including Dutch, German, French, and English.**

I. INTRODUCTION

In this work, we explore the usage of HashTables as efficient data structures. Furthermore, we explore how their performance can be improved in terms of avoiding term collision and accurately storing (key,value) pairs. Using Charles Dickens' work Oliver Twist, we conducted tests counting the number of distinct words in the file, for different languages and using different hashing methods.

II. HASH TABLES

An HashTable is a datastructure that uses an hashing function to map a (key,value) pair to a position in an array. Because it uses an hash function to produce this index, insertion and retrieval operations are fast.

In python, an equivalent structure is used, called a dictionary.

A. Hashing

Hashing consists in transforming a specific given *key* into an integer value. This integer value can be either positive or negative, according to the used method.

Python's native hash function is very fast and reliable and can operate over any object, as long as it has a hash value. The hash value of a given integer x is x itself.

Many hashing functions were also developed with different complexity.

B. Collisions

As previously mentioned, the hash function transforms a given *key* object into a specific integer value. This value needs to be mapped to the size of the HashTable, in order to produce an available index. One of the ways of achieving this is by simply using the remainder of the division between the hash value and the table size as the key index.

Problems arise when two different keys are mapped into the same position in the table. This is called a collision.

Although rare, collisions are possible and there are many ways of dealing with them. One of the simplest ways is *Linear Probing*, which simply consists of finding the next closest free index and inserting the key there. However, in this work, as requested, we will not be dealing with collision resolution

III. SOLUTION

A. The HashTable Class

Based on the dictionary implementation in python, the HashTable class inside the HashTable.py script presents some of its core functions.

This table supports two hashing methods, one is the default *hash* function in python. The second one is a polynomial rolling hash function as presented below.

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

Figure 1 – Polynomial Rolling Hash Function (PRHF) of a string *s* of length *n*, and given some positive integers *m* and *p*

The implemented structure takes into consideration some parameters:

- *SIZE_FACTOR*: Table size multiplicator factor. Default is 1.5
- *Method*: Hashing method to be used. Default is 'PyHash'
- *p*: integer value used in one of the hashing methods

Obviously, the implemented code presents and hashing method *hash(key)*, inserting method *put(key,value)* and a retrieval method *get(key).*

Furthermore, the structure also supports the following operations:

- *contains(key)*: indicates if a key is possibly or not in the hash table
- *length():* returns the number of keys
- *get_keys():* returns an array of all inserted keys
- *get_values():* returns an array of all values
- *items():* returns tuples that map inserted keys with the respective values

As previously mentioned, this structure is able to save (key, value) pairs. In our case, since we only want to count the distinct words, we will only insert keys with a value of 1.

It is also important to notice that some different keys might be mapped to the same index in the structure, because we are not dealing with the collisions, as requested.

B. *The LiteraryFile Class*

This class is simply used in order to retrieve the words from each text file, so that the HashTable can understand them.

As such, given a work W in a certain language L, this class presents the following methods:

- *read_file():* for each line in W, it applies the normalize_text() function and yields its result
- *normalize_text(string):* for a given string, removes all punctuation and splits the string into words, removing stop-words for the given language L
- *max_number_words():* estimates the maximum number of words in file, by multiplying the number of lines by the maximum number of words in a line of the file

As such, using this class, we can quickly turn a work into an array of words to be processed.

C. *Counting the Distinct Words*

Now that we have the data and the hash table defined, the script inside distinct_words.py is responsible for calling all methods and determining the amount of different words.

Some parameters can be given to this script, such as:

- *-f Filename:* Name/path to the work to be analyzed
- *-hash HashingMethod*: defines the hashing method to be used ('PyHash' or 'PRHF')
- *-s SIZE_FACTOR*: defines the SIZE_FACTOR variable for the HashTable structure

All these arguments are optional. If the filename is bot given, it will analyze every work inside the *texts/* folder. The default hashing method is python's hash function ('PyHash') and the default SIZE_FACTOR is 1.5.

The *count_distinct()* function is responsible for the counting. As such, it returns the exact number of distinct words using an exact count as well as the number of distinct words using the HashTable structure.

```
File texts/OliverTwist_german.txt
Estimated Count 16992
Real Count 17143
Number of Collisions 151
```

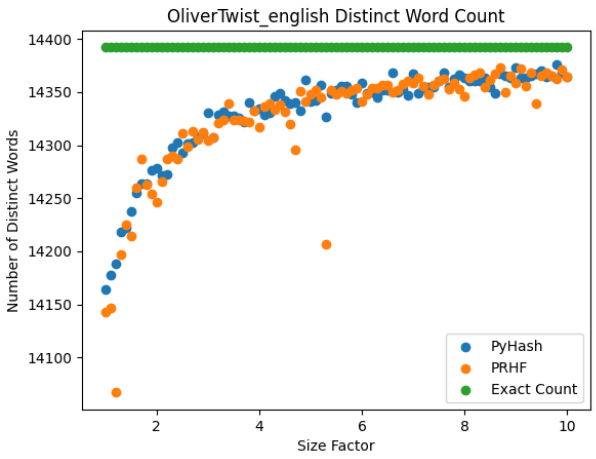
Figure 5 – Results of The Counting Function

It also utilizes the max_number_words() function from the LiteraryFile class in order to calculate the size of the HashTable, which will be the result of the multiplication of the SIZE_FACTOR and the estimated maxNumberWords.

IV. RESULTS

Experiments were conducted with both different hashing methods, different SIZE_FACTORs, and different languages. The size factor ranged from 1.0 to 10.0, with increments of 0.1

In the figures below, the green line represents the exact count value, whereas the remaining scatter plots represent the distinct word count using the hash table, for each method hashing method.



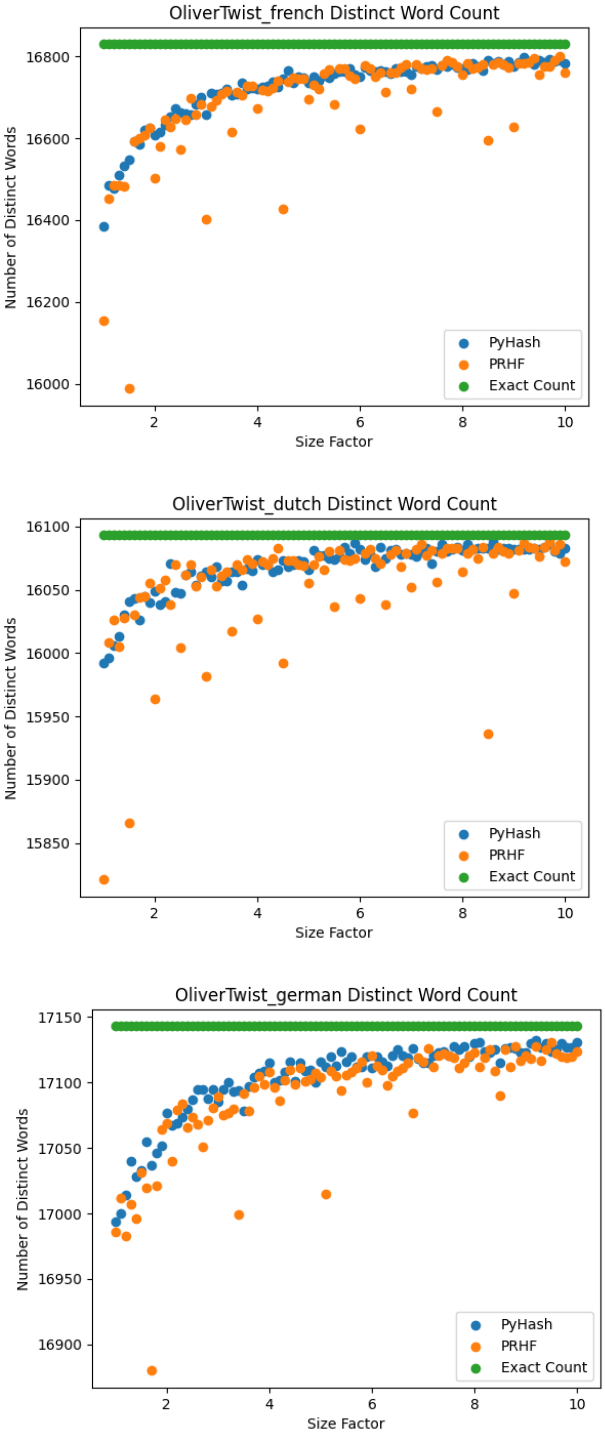
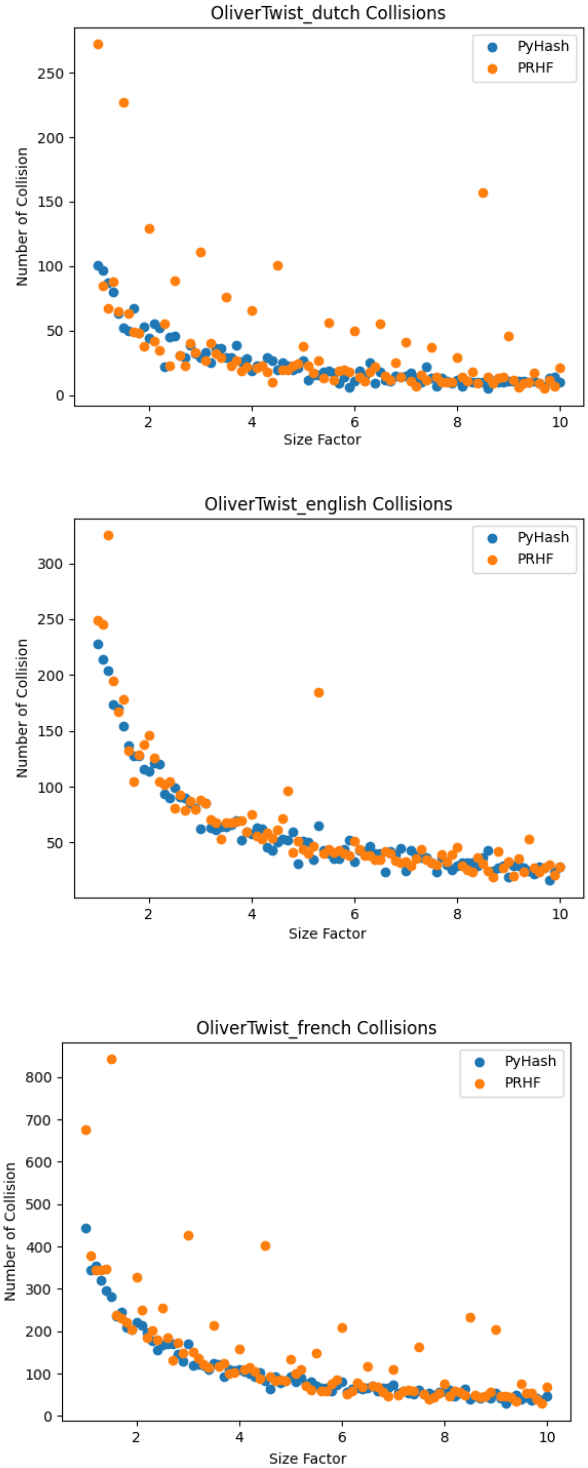


Figure 6 – Distinct Word Counts for Each Language, using Python's Hashing function and the created Polinomial Rolling Hash Function vs Exact Count

From the previous figures, it is obvious that the bigger the Size Factor, the more exact was the counter. Furthermore, we can actually differentiate the performance between both hashing methods. The exact counter has a different value from the HashTable structure, because of the occurrence of collisions, which just replace entires in the structure. As such, we can conclude that Python's Hash function is best at avoiding collisions, i.e. mapping different keys to the same index.

Below, we can actually visualize how the number of collisions varies with different Size Factors.



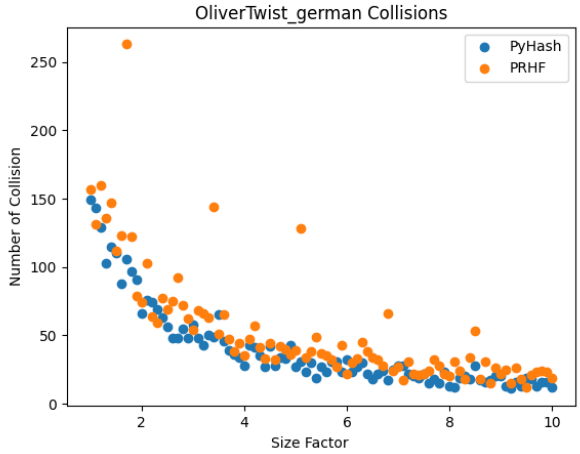


Figure 7 – Number of collisions related to the Size Factor of the HashTable

As previously expected, the number of collisions is smaller with bigger Size Factor values, hence the more precise counter value.

Furthermore, another interesting fact was that different languages report a different number of distinct words.

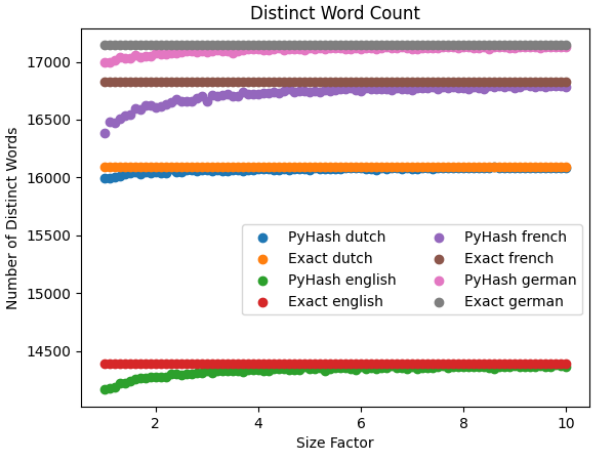


Figure 8 – Number of Distinct Words Per Language

As visualized, while French produced the biggest difference for lower Size Factor values, Dutch provided the closest values.

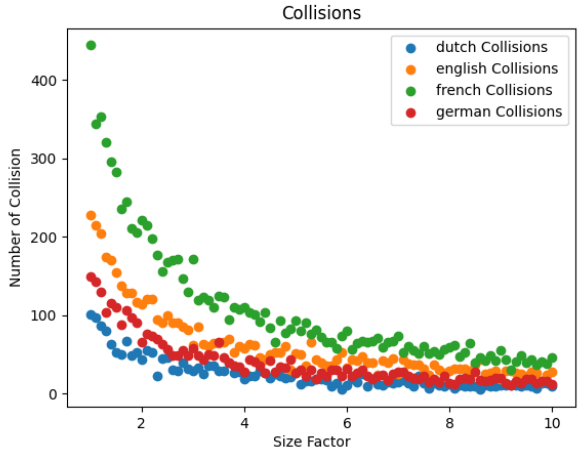


Figure 9 – Number of Collisions Words Per Language

The figure above further illustrates the previous point, with the French work producing the highest number of collisions, and the Dutch producing the lowest.

IV. CONCLUSION

With this work, it was evident that the HashTables consist of efficient and memory-saving structures to keep (key, value) pairs. Furthermore, it was also evident the need to implement collision resolution with these structures as well as a good hash function.

REFERENCES

[1] Algorithms for Competitive Programming, “String Hashing”, (https://cp-algorithms.com/string/string-hashing.html).

[2] TutorialsPoint, "Python - Hash Function", (https://www.tutorialspoint.com/python_data_structure/python_hash_table.htm).