VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

MODELLING AND DATA ANALYSIS MASTER'S STUDY PROGRAMME

Master's thesis

Sensitivity Analysis of Online String-Matching Algorithms

Srautinio teksto paieškos algoritmo jautrumo analizė

ALVARO RIVERA MONTANO

Supervisor: Assoc. Prof. Dr. Viktor Skorniakov

Vilnius, 2021

Table of contents

List of images

List of Tables

List of abbreviations

BMH	Boyer-Moore-Horspool
KMP	Knuth–Morris–Pratt
MAE	Mean absolute error
MSE	Mean squared Error
OAT	One-factor-at-a-time
R2	R Squared
RMSE	Root mean squared error

Abstract

Alvaro Rivera Montano, Sensitivity Analysis of Online String-Matching Algorithms: Master's Thesis/ Supervisor Assoc. Prof. Dr. Viktor Skorniakov; Vilnius University, Faculty of Mathematics and Informatics.

String-matching is a well known and studied problem in computer science that allows us to find occurrences of a pattern inside a text. As datasets grow larger every year, it becomes more and more important to know which algorithms work better in specific situations, as there is not a single algorithm that will be well suited for every scenario. The aim of this paper is to analyse a new algorithm proposed by Faro et al.[10], which takes the approach of subsampling the text and patterns before calculating the distance of the elements in the subsample. Sensitivity analysis is used to allow us to test the robustness of the proposed algorithm. An extensive experimental comparison of the parameters, together with a comparison with classical algorithms, was presented. A dataset was generated from the performance measurements applied during the experiments and in addition, some suggestions about case scenarios of when to use the newly proposed algorithm.

Key words : String matching, online string matching, sensitivity analysis, characters distance, characters sampling, experimental evaluation, algorithm comparison, dataset generation.

1 Introduction

String matching is a well known and studied problem in computer science; It consists of finding the occurrences of a pattern x of length m in a text y of length n. Both x and y are built over the same alphabet of size s.

The fields where applications of string matching are being used are quite extensive and include diverse areas such as text processing, speech recognition, information retrieval, among others. For instance in linguistics and literature is used to search for patterns in huge corpus and dictionaries, in biology to search for amino acid sequence patterns[13], and in computer science to search for a large amount of data that are stored in linear files. Therefore there is a constant need for more and faster solutions to the text searching problem.

Offline and online solutions exist depending on if the text or otherwise the pattern is provided first and if an index is required to preprocess the text. Indexes that are realized by data structures are used by the offline solutions, which try to drastically speed up searching by preprocessing the text and building a data structure that allows searching in time proportional to the length of the pattern. For this reason, such kind of problem is known as indexed searching[9]. Among the most efficient solutions to such a problem, we mention those based on suffix trees. However, despite their optimal time performance, the space requirements of full-index data structures, as suffix-trees and suffix-arrays, are from 4 to 20 times the size of the text, and such space requirement is too large for many practical applications.

A more practical solution is the online string matching approach, which utilizes partial or no indexes at all. The Online solution assumes the text is not preprocessed, and thus they need to scan the text online when searching. In this thesis, the focus is on the performance of online string matching algorithms. Various online string-matching algorithms have been proposed throughout the years, being two of the most well known the Knuth-Morris-Pratt algorithm and the Boyer-Moore algorithm, especially with its Horspool variation, this kind of algorithms have followed the approach of the original Naive algorithm but using auxiliary functions to skip some of the characters that are being scanned.

A different approach to the online string machine problem was introduced by[6], which consists in the construction of a succinct sampled version of the text and in the application of any online string matching algorithm directly on the sampled sequence. The drawback of this approach is that any occurrence reported in the sampled-text may require to be verified in the original text. However, a sampled-text approach has a lot of good features: it may be easy to implement, may require little extra space, and may allow fast searching.

More recently[10] extended the works of[6] by introducing the idea of the Character Distance of Pivots algorithm, which shows promising results; however, not many experiments have been conducted to corroborate that this could indeed be a good alternative to more classical string matching algorithms. In this thesis, the focus is on this new online algorithm and its comparison with the standard algorithms of the field.

1.1 Objectives

1.1.1 Aim

The aim of this thesis is to investigate the performance of the online string matching algorithm suggested by Faro et al.[10] to evaluate it in comparison with standard string matching algorithms.

1.1.2 Goals

- To measure the performance of the algorithm in reference to Preprocessing time, Searching Time, and Space requirements.
- To perform experiments (measured primarily by the time of execution) on diverse factors such as language, text size, and pattern length.
- To generate a dataset from the results of the experiments.
- To compare the results obtained from the experiments with the results of standard string matching algorithms.
- To summarize the results obtained from both the parameter experiments as well as the comparison between algorithms.
- To recommend case scenarios when the use of a specific algorithm could be useful.

The novelty of this thesis stems from the fact that the algorithm suggested by Faro et al.[10] is brand new and, consequently, investigations of such kind were not performed; furthermore, an extensive analysis of the subsampling string matching approach is non-existent, and this thesis could be an initial step to lead to further investigations of the topic.

2 Background

In this section, some concepts related to string-matching algorithms are discussed, as well as their categories and use cases to posteriorly introduce the main algorithm that will be analysed in the experimental section of the thesis.

2.1 String Matching Algorithms

Finding all the occurrences of a string in a text is known as string matching, and it is a very important subject that belongs to the domain of text processing. It has been extensively studied and applied in the field of computer science, mainly due to its various practical applications in different areas like information analysis and retrieval. String matching algorithms are often used as part of more complex software systems[8]. Many programs in different fields of science require to match patterns; among these, we can mention information retrieval, biology, and speech recognition.

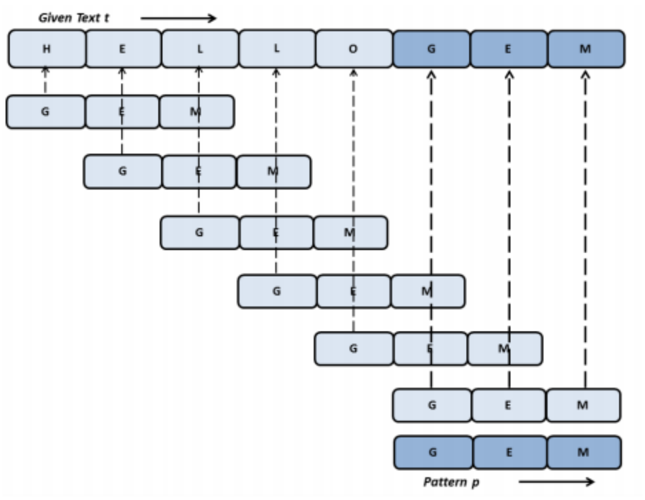


Figure 1: A character comparison string-matching algorithm[11]

2.1.1 Classification of String-Matching Algorithms

Exact string matching algorithms are the most common type of matching algorithms; in this case, the algorithm expects a full match of the string; however, there is another matching alternative, the approximate string matching algorithm, which searches for a substring that is close to the given pattern string, how the closeness degree is calculated depends on the application and complexity of the problem. For this thesis, the focus is on exact string matching.

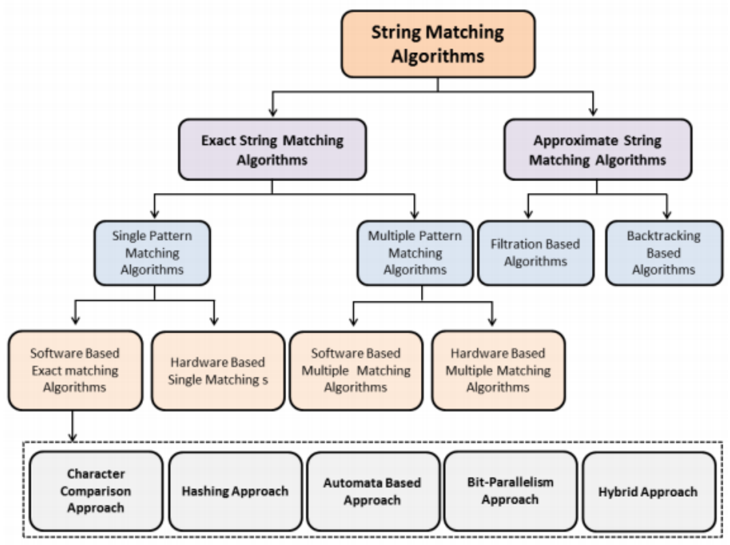

String Matching Algorithms
Exact String Matching Algorithms
Approximate String Matching Algorithms
Single Pattern Matching Algorithms
Multiple Pattern Matching Algorithms
Filtration Based Algorithms
Backtracking Based Algorithms
Software Based Exact matching Algorithms
Hardware Based Single Matching s
Software Based Multiple Matching Algorithms
Hardware Based Multiple Matching Algorithms
Character Comparison Approach
Hashing Approach
Automata Based Approach
Bit-Parallelism Approach
Hybrid Approach

Figure 2: Taxonomy of the string-matching algorithms[11]

An important question in string matching that will affect the type of preferred solution is about which string, the pattern, or the text, is given first. In the case, the text is given first, and a full indexing of it is created then we call this Offline string-matching, otherwise in the case when the pattern is given first, and we can process the pattern at the moment of running time then this case would be considered as Online string matching. In this thesis, we are interested only in analysing this kind of algorithms. Partial indexing of the text for online string matching algorithms[1] is also common, which is the case of the algorithm that will be analysed in this thesis. An example of online string-matching algorithm is the one that utilise the approach of automata like we can see in figure 3 in this case, a finite automaton tries to match the pattern 'ababc' over the alphabet *Sigma*= a,b,c. However, in this work, the focus is on Online algorithms[17] of the most common type, which are the Character Comparison algorithms[7], which compare characters one by one, like the one showed in figure 1 .

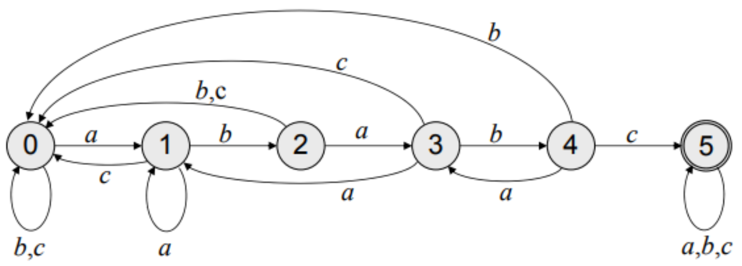


Figure 3: Example of an Automaton

2.2 A Characters Distance Sampling Algorithm

In this section, the main algorithm of this thesis is presented, proposed by Faro[10], extending the work done by Claude[6] with the sampled string matching approach. In the following subsections, the ideas introduced by Faro et al. are explained.

Let y be the input text, of length n, and let x be the input pattern, of length m, both over the alphabet Σ of length σ. We refer to x[i] as the i-th character of the string x starting at position 1, for $1 \leq i \leq m$

Firstly a sampled alphabet $\bar{\Sigma} \subset \Sigma$ is defined; in this algorithm, the sampled alphabet is called the set of pivot characters. The set $\bar{\Sigma}$ in practice is just reduced to a single character[5]; therefore, for the rest of the paper, we can assume that $|\bar{\Sigma}| = 1$ and this set composed of just one character is known as the pivot character.

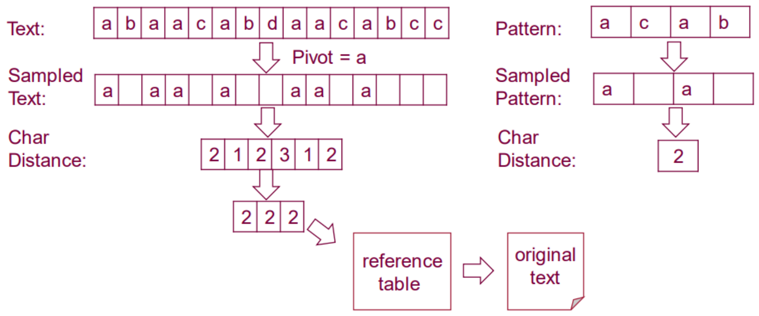


Figure 4: Character Distance Algorithm

In figure 4, we can see a basic example of how the algorithm works; initially, we select a pivot character; after that, the text will be subsampled to only keep the positions of that character. In the selection of the pivot character, one option could be to just rank the frequencies of characters and select one of the most frequent ones. In the example, the pivot is the character 'a'; the same is done with the pattern we want to search. After this, the distances between the occurrences of the characters are kept and stored in an auxiliary table; up to this point is what we call preprocessing time, which is one of the performance values measured in this thesis. This is done once per document unless we changed the pivot character.

After the initial step, we enter into the searching phase, the most important performance measurement, where the algorithm checks if there are occurrences of the distances of the pattern in the distances of the text, in our example, the pattern only has one distance of value 2, and there are three occurrences of that value in the distances of the text. After this, with the help of the auxiliary table, we check the original text only for those three candidate distances using a classical one by one character matching function and this allows us to avoid having to check the whole document.

We now need to define the concept of bounded position sampling, which is the approach used by the algorithm.

Definition 1 (The Bounded Position Sampling) Let y be a text of length n, let $C \subseteq \Sigma$ be the set of pivot characters and let n_c be the number of occurrences of any character of C in the input text y. First we define the position function, $\delta : \{1, \dots, n_c\} \rightarrow \{1, \dots, n\}$ where $\delta(i)$ is the position of the i-th occurrence of any character of C in y. Assume now that k is a given threshold constant. We defne the k-bounded position function, $\delta_k : \{1, \dots, n_c\} \rightarrow \{0, \dots, k-1\}$, where $\delta_k(i)$ is the position (modulus k) of the i-th occurrence of any character of c in y. Formally we have

$$\delta_k(i) = [\delta(i) \mod k], \text{ for each } i = 1, \dots, n_c$$

The k-bounded-position sampled version of y, indicated by $\dot{y}$, is a numeric sequence, of length n_c defined as

$$\dot{y} = \langle \delta_k(1), \delta_k(2), \dots, \delta_k\,(n_c) \rangle$$

Then we have $0 \leq \dot{y}[i] < k$, for each $1 \leq i \leq n_c$

Definition 2 (The Characters Distance Sampling) Let $c \in \Sigma$ be the pivot character, let $n_c \leq n$ be the number of occurrences of the pivot character in the text y and let δ be the position function of y. We define the characters distance function $\Delta(i) = \delta(i+1) - \delta(i)$, for $1 \leq i \leq n_c - 1$, as the distance between two consecutive occurrences of the character c in y. The characters-distance sampled version of the text y is a numeric sequence, indicated by $\bar{y}$, of length $n_c - 1$ defined as:

$$\bar{y} = \langle \Delta(1), \Delta(2), \dots, \Delta\,(n_c - 1) \rangle$$

Then we have:

$$\textstyle\sum_{i=1}^{n_c-1} \Delta(i) \leq n - 1$$

To be able to retrieve the original i-th position $\delta(i)$, of the pivot character, from the i-th element of the k-bounded position sampled text $\dot{y}$, a block-mapping table τ is also maintained which stores the indexes of the last positions of the pivot character in each k-block of the original text, for a given input block size k.

Is assumed that the text y is divided in $\lceil n/k \rceil$ blocks of length k, with the last block containing (n mod k) characters. Then $\tau[i] = j$ if the j-th occurrence of the pivot character in y is also its last occurrence in the i-th block. If the i-th block of y does not contain any occurrence of the pivot character then $r[i]$ is set to be equal to the last position of the pivot character in one of the previous blocks.

2.2.1 Pseudocode

In this section, all the procedures that compose the algorithms are explained, including the auxiliary functions as well as the procedures that are used depending on the number of pivot character occurrences that exist in the input pattern.

```
GET-POSITION($\tau$, $b$, $\dot{y}$, $i$)
    while $\tau[b] < i$ do
        $b \leftarrow b + 1$
    $p \leftarrow (b-1) \times k + \dot{y}[i]$
    return ($p$, $b$)
```

```
COMPUTE-CHARACTER-DISTANCE-SAMPLING($\dot{y}$, $\tau$)
    $\bar{y} \leftarrow \langle\rangle$
    $n_c \leftarrow len(\dot{y})$
    $b \leftarrow 1$
    $(\delta_1, b) \leftarrow$ GET-POSITION($\tau$, $b$, $\dot{y}$, 1)
    for $i \leftarrow 2$ to $n_c$ do
        $(\delta_i, b) \leftarrow$ GET-POSITION($\tau$, $b$, $\dot{y}$, $i$)
        $\bar{y}[i-1] \leftarrow \delta(i) - \delta(i-1)$
    return $\bar{y}$
```

Figure 5: Pseudocode for the functions Get-Position and Compute-Character-Distance

In the figure figure 5, the pseudocode on the left shows the function Get-Position which, computes the index of the block that corresponds to the i-th occurrence of the pivot character in y, in other words, we obtain the position of the character in the original text.

The pseudocode of the function on the right named Compute-Character-Distance-Sampling computes the character distance sampled version of y, which is obtained from $\dot{y}$. So we get $\bar{y}$ which will be use in our main Search function to process the distances between occurrences of the selected pivot character in y.

```
COMPUTE-DISTANCE-SAMPLING($y$, $n$, $\bar{\Sigma}$)
    $\bar{y} \leftarrow \langle\rangle$
    $j \leftarrow 0$
    $p \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
      if $y[i] \in \bar{\Sigma}$ then
        $j \leftarrow j + 1$
        $\bar{y}[j] \leftarrow i - p$
        $p \leftarrow i$
    return ($\bar{y}$, $j$)
```

```
COMPUTE-POSITION-SAMPLING($y$, $n$, $\bar{\Sigma}$, $k$)
    $\dot{y} \leftarrow \langle\rangle$
    $\tau \leftarrow$ a table of $\lceil n/k \rceil$ entries
    for $i \leftarrow 1$ to $\lceil n/k \rceil$ do $\tau[i] \leftarrow 0$
    $j \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
      if $\tau[\lceil i/k \rceil] = 0$ then
        $\tau[\lceil i/k \rceil] \leftarrow \tau[\lceil i/k \rceil - 1]$
      if $y[i] \in \bar{\Sigma}$ then
        $j \leftarrow j + 1$
        $\dot{y}[j] \leftarrow i \mod k$
        $\tau[\lceil i/k \rceil] \leftarrow j$
    return ($\dot{y}$,$j$,$\tau$)
```

Figure 6: Pseudocode for the functions Compute-Distance-Sampling and Compute-Position-Sampling

In the figure figure 6, on the left we have the pseudocode for the function Compute-Distance-Sampling which is used for the construction of the character distance sampling version of a text y. In practice, this is used inside the search functions as a filtering phase to process the distances of the pivot character in the pattern x and is calculated online.

On the right, we have the pseudocode for the function Compute-Position-Sampling, which constructs the character position sampling version of a text y. This is the main function in the preprocessing stage, and the entry point to the workflow of the algorithm, the parameters that need to be provided are the text y, the alphabet from where we will select the pivot character, and the constant k, which is the value for the segment size.

```
VERIFY($x$, $m$, $y$, $s$)
1.    $i \leftarrow 0$
2.    while $i \leq m$ and $x[i] = y[s+i]$ do
3.        $i \leftarrow i + 1$
4.    if $i = m$ then
5.        report occurrence at $s$
```

```
RESTORINGVERIFY($x$, $m$, $y$, $s$, $k$, $\mathcal{D}(x)$, $q_0$)
1.    $q \leftarrow q_0$
2.    $c \leftarrow 0$
3.    for $i \leftarrow s$ to $s + k - 1$ do
4.        $q \leftarrow \delta(q, y[i])$
5.        if $(q = m)$ then
6.            report occurrence at $i - m + 1$
```

Figure 7: Pseudocode for the functions Verify and RestoringVerify

In the figure figure 7, the pseudocode on the left shows the function Verify which, as the name indicates, verifies the occurrences of a pattern x of length m, in a document y, of length n, starting at the provided position s. This function is used inside our main Search procedure to verify if candidate matching distances of a pattern and a text contains also matching characters; it simply compares the substring of the text $y[s \cdots s+m-1]$ and the pattern x, character by character, going from left to right until a mismatch is found or the two strings completely match if so it reports an occurrence at the position s. The time complexity for this function is $O(m)$ in the worst case.

The pseudocode on the right of In the figure figure 7, shows the function RestoringVerify which uses a string matching automaton of x to remember the positions of the text which have been already processed during a previous call yo the Verify function.

```
SEARCH-0($x$, $\dot{y}$, $y$)
$\Delta$    We assume $c$ does not occur in $x$
1.    $m \leftarrow len(x)$
2.    $n_c \leftarrow len(\dot{y})$
3.    $b \leftarrow 1$
4.    $\delta_0 \leftarrow 0$
5.    for $i \leq 2$ to $n_c$ do
6.        $(\delta_i, b) \leftarrow$ GET-POSITION$(\tau, b, \dot{y}, i)$
7.        if $(\delta_i - \delta_{i-1} > m)$ then
8.            search for $x$ in $y[\delta_{i-1} + 1..\delta_i - 1]$
9.    if $(n + 1 - \delta_{n_c} > m)$ then
10.       search for $x$ in $y[\delta_{n_c} + 1..n]$
```

Figure 8: Pseudocode for the function Search-0

Figure figure 8 shows the pseudocode of the function Search-0 for the sampled string matching problem, this function will be called when there are no occurrences of the pivot character c in the pattern x, so m_c equals to zero, in this case, there will be no distances to calculate, so this would be the first kind of exception case of the algorithm. Under this assumption, the algorithm searches for the pattern x in all substrings of the original text, which do not contain the pivot character.

We can identify those substrings by the intervals $[\delta(i)+1\cdots\delta(i+1)-1]$, for each $0 \leq i \leq n_c$ assuming that $\delta(0)=0$ and $\delta\left(n_c+1\right)=n+1$

Specifically, for each $1 \leq i \leq n_c+1$ the algorithm checks if the interval $\delta(i)-\delta(i-1)$ is greater than c. If this is the case then the algorithm searches for x in the substring of the text $y[\delta(i-1)+1\cdots\delta(i)-1]$ using any classical matching algorithm. Otherwise we simply skip this substring as no occurrences of the pattern were found in the searched position. The worst-case and average time complexity of the Search-0 algorithm are $O(n)$ and $O\left(n\log_a m/m\right)$, respectively.

```
SEARCH-1($x$, $\dot{y}$, $y$)
Δ     We assume $c$ occurs once in $x$
1.    $m \leftarrow len(x)$
2.    $n_c \leftarrow len(\dot{y})$
3.    $\alpha \leftarrow \min\{i : x[i] = c\}$
4.    $b \leftarrow 1$
5.    $\delta_0 \leftarrow 0$
6.    $(\delta_1, b) \leftarrow$ GET-POSITION$(\tau, b, \dot{y}, 1)$
7.    for $i \leq 2$ to $n_c$ do
8.        $(\delta_i, b) \leftarrow$ GET-POSITION$(\tau, b, \dot{y}, i)$
9.        if $(\delta_{i-1} - \delta_{i-2} > \alpha - 1$ and $\delta_i - \delta_{i-1} > m - \alpha)$ then
10.           VERIFY($x$, $m$, $y$, $\delta_{i-1} - \alpha + 1$)
11.   if $(\delta_{n_c} - \delta_{n_c-1} > \alpha - 1$ and $n + 1 - \delta_{n_c} > m - \alpha)$ then
12.       VERIFY($x$, $m$, $y$, $\delta_{n_c} - \alpha + 1$)
```

Figure 9: Pseudocode for the function Search-1

Suppose now the case where the pattern x contains a single occurrence of the pivot character c, so that the length of the sampled version of the pattern is still equal to zero. The algorithm efficiently takes advantage of the information precomputed in $\dot{y}$ using the positions of the pivot character c in y as an anchor to locate all candidate occurrences of x.

Figure figure 9 shows the pseudocode of the algorithm which searches for all occurrences of a pattern x, when the pivot character c occurs only once in it. Specifically, let α be the unique position in x which contains the pivot character, we assume that $x[\alpha]=c$ and $x[1\cdots\alpha-1]$ does not contain c. Then, for each $0 \leq i \leq n_c-1$, the algorithm checks if the interval $\delta(i-1)-\delta(i-2)$ is greater than $\alpha-1$ and if the interval $\delta(i)-\delta(i-1)$ is greater than $m-\alpha$. So if that case happens then algorithm will simply check if the substring of the text $y[\delta(i)-\alpha+1\cdots\delta(i)+m-\alpha]$ is equal to the pattern, otherwise the substring is skipped. As before we assume that $\delta(0)=0$ and $\delta\left(n_c+1\right)=n+1$. Finally the last alignment of the pattern in the text is verified separately at the end of the main cycle.

```
SEARCH-2$^+$($x$, $\dot{y}$, $y$)
Δ    We assume $c$ occurs more than once in $x$
1.   $m \leftarrow len(x)$
2.   $\dot{n} \leftarrow len(\dot{y})$
3.   $\alpha \leftarrow \min\{i : x[i] = c\}$
4.   $b \leftarrow 1$
5.   $(\bar{x}, \bar{m}) \leftarrow$ COMPUTE-DISTANCE-SAMPLING$(x, m, \{c\})$
6.   search for $\bar{x}$ in $\bar{y}$ :
7.       for each $i$ such that $\bar{x} = \bar{y}[i..i + \bar{m} - 1]$ do
8.           $(\delta_i, b) \leftarrow$ GET-POSITION$(\tau, b, i)$
9.           VERIFY$(x, m, y, \delta_i - \alpha)$
```

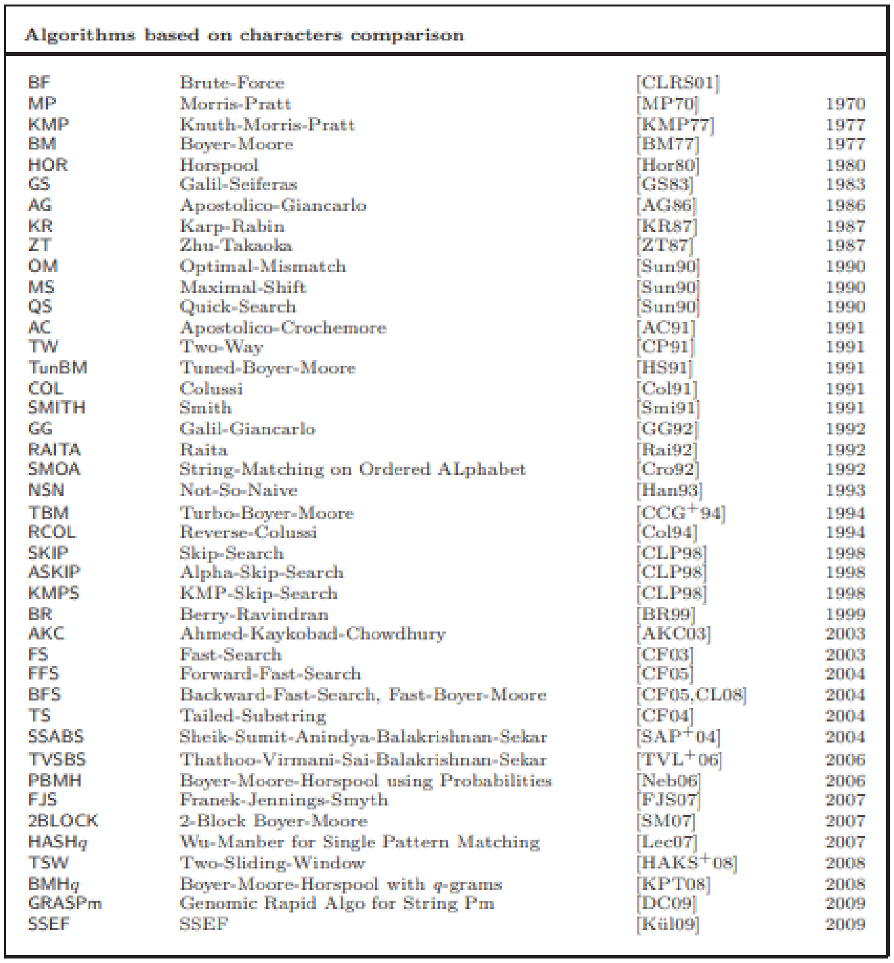
Figure 10: Pseudocode for the function Search-2+

If the number of occurrences of the pivot character in x are 2 or more, i.e. $m_c \geq 2$ and $\bar{m} \geq 1$) then the algorithm uses the sampled text $\bar{y}$ to compute on the fly the character-distance sampled version of y and use it to search for any occurrence of $\bar{x}$. This is used as a filtering phase for locating in y any candidate occurrence of x.

figure 10 shows the pseudocode of the algorithm which searches for all occurrences of a pattern x, when the pivot character c occurs more than once in it. First the character distance sampled version of the pattern $\bar{x}$ is computed. Then the algorithm searches for $\bar{x}$ in $\bar{y}$ using any exact online string matching algorithm. Notice that $\bar{y}$ can be efficiently retrieved online from the sampled text $\dot{y}$. For each occurrence position j of $\bar{x}$ in $\bar{y}$ an additional procedure must be run to check if such occurrence corresponds to a match of the whole pattern x in y . For this purpose the algorithm checks if the substring of the text $y[\delta(j) - \alpha \cdots \delta(j) - \alpha + m - 1]$ is equal to x , where α , as before, is the first position in x where the pivot character occurs. The value of $\delta(j)$ can be obtained in constant time from $\dot{y}[j]$.

2.3 Classic Algorithms

In this section, some traditional string-matching algorithms are explained, initially the basic Naive algorithm, and posteriorly two of the most well known classical algorithms, the Knuth–Morris–Pratt algorithm, and the Boyer-Moore-Horspool. These algorithms will be posteriorly compared with the Character Distance algorithm in the experiments section of the thesis.



Algorithms based on characters comparison			
BF	Brute-Force	[CLRS01]	
MP	Morris-Pratt	[MP70]	1970
KMP	Knuth-Morris-Pratt	[KMP77]	1977
BM	Boyer-Moore	[BM77]	1977
HOR	Horspool	[Hor80]	1980
GS	Galil-Seiferas	[GS83]	1983
AG	Apostolico-Giancarlo	[AG86]	1986
KR	Karp-Rabin	[KR87]	1987
ZT	Zhu-Takaoka	[ZT87]	1987
OM	Optimal-Mismatch	[Sun90]	1990
MS	Maximal-Shift	[Sun90]	1990
QS	Quick-Search	[Sun90]	1990
AC	Apostolico-Crochemore	[AC91]	1991
TW	Two-Way	[CP91]	1991
TunBM	Tuned-Boyer-Moore	[HS91]	1991
COL	Colussi	[Col91]	1991
SMITH	Smith	[Smi91]	1991
GG	Galil-Giancarlo	[GG92]	1992
RAITA	Raita	[Rai92]	1992
SMOA	String-Matching on Ordered ALphabet	[Cro92]	1992
NSN	Not-So-Naive	[Han93]	1993
TBM	Turbo-Boyer-Moore	[CCG$^+$94]	1994
RCOL	Reverse-Colussi	[Col94]	1994
SKIP	Skip-Search	[CLP98]	1998
ASKIP	Alpha-Skip-Search	[CLP98]	1998
KMPS	KMP-Skip-Search	[CLP98]	1998
BR	Berry-Ravindran	[BR99]	1999
AKC	Ahmed-Kaykobad-Chowdhury	[AKC03]	2003
FS	Fast-Search	[CF03]	2003
FFS	Forward-Fast-Search	[CF05]	2004
BFS	Backward-Fast-Search, Fast-Boyer-Moore	[CF05,CL08]	2004
TS	Tailed-Substring	[CF04]	2004
SSABS	Sheik-Sumit-Anindya-Balakrishnan-Sekar	[SAP$^+$04]	2004
TVSBS	Thathoo-Virmani-Sai-Balakrishnan-Sekar	[TVL$^+$06]	2006
PBMH	Boyer-Moore-Horspool using Probabilities	[Neb06]	2006
FJS	Franek-Jennings-Smyth	[FJS07]	2007
2BLOCK	2-Block Boyer-Moore	[SM07]	2007
HASHq	Wu-Manber for Single Pattern Matching	[Lec07]	2007
TSW	Two-Sliding-Window	[HAKS$^+$08]	2008
BMHq	Boyer-Moore-Horspool with q-grams	[KPT08]	2008
GRASPm	Genomic Rapid Algo for String Pm	[DC09]	2009
SSEF	SSEF	[Kül09]	2009

Figure 11: List of comparison based string-matching algorithms[9]

2.3.1 Brute force algorithm

Features

- It doesn't need to preprocess the pattern nor the text.
- The space required by the algorithm is constant.
- The sliding window shift always moves one position to the right.
- Comparisons can be made in any order.
- Searching stage in $O(m \times n)$ time complexity.
- 2n expected text character comparisons.

The naive algorithm, also known as brute force, effectuates the character check at all positions in the text between position one and n, whether an occurrence of the candidate pattern begins there or not. After each attempt, the sliding windows will be shifted to the right by exactly one position. The naive algorithm does not require a preprocessing stage for the text nor the pattern; the required space in memory required by the algorithm will be constant. During the searching stage, the text character comparisons can be effectuated in

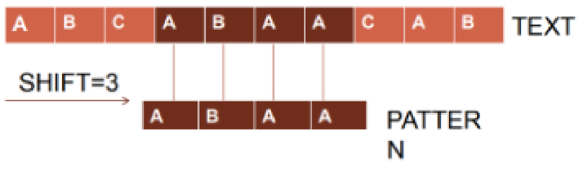


Figure 12: Example of shift and matching in the Naive Algorithm.[18]

any order.

2.3.2 Knuth-Morris-Pratt

Features

- The algorithm performs the comparison from left to right.
- The Preprocessing stage in O(m) space and time complexity.
- Searching stage in O(m+n) time complexity.
- The maximum of character comparisons during the searching stage is 2n-1.

.

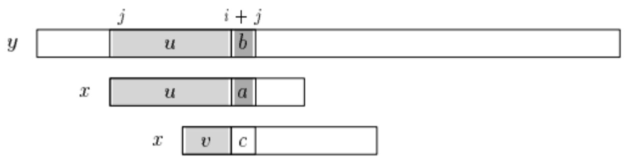


Figure 13: Shift in the Knuth-Morris-Pratt algorithm.[4]

Unlike with the case of the naive algorithm, when the KMP encounters a new mismatch, it performs some analysis on the substring before trying to find its occurrences in the text. Backtracking on the pattern never occurs with this algorithm as it avoids to compare a second time a character that was already involved in another comparison with some character of the pattern.

2.3.3 Boyer-Moore-Horspool algorithm

Features

- Improved and simplified version of the Boyer-Moore algorithm.
- Uses the bad-character shift auxiliary table.
- Easy and straightforward to implement.
- Preprocessing phase in $O(m + \sigma)$ time and $O(\sigma)$ space complexity.
- Searching phase in $O(m \times n)$ time complexity.
- The mean number of comparisons for a single character is between $1/\sigma$ and $2/(\sigma + 1)$

.

a	A	C	G	T
$bmBc[a]$	1	6	2	8

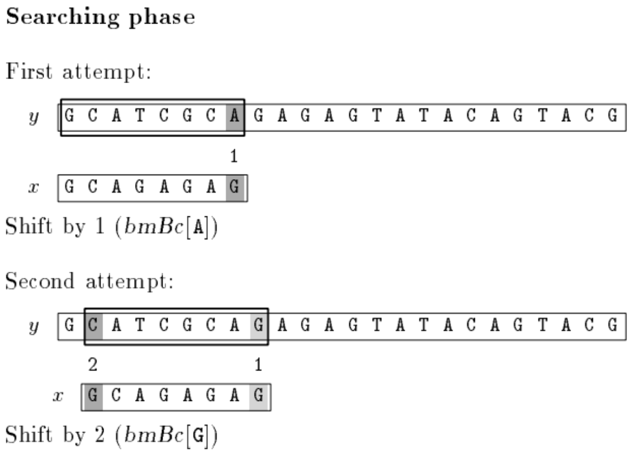


Figure 14: Example of the Searching Phase of the Horspool Algorithm.[3]

The execution time in the case of the Boyer-Moore-Horspool algorithm is linear in relation to the length of the pattern that is being searched. BMH has a lower execution time than many of the popular string-searching algorithms. One of many variations of the Boyer-Moore[2] algorithm, Horspool uses the auxiliary table for bad matches to exclude the positions where the substring can't match the pattern and skip the bad characters; this allows the algorithm to perform better when it can utilize more often the auxiliary table like in the cases of large input patterns, therefore the larger the pattern, the faster the searching process.

3 Experimental Evaluation

In this section, the experimental evaluation of the algorithm and its required steps to accomplish it are explained. After the initial step of implementing the newly proposed algorithm, several text files are collected, and multiple versions of each document are created; posteriorly, those documents are used together with multiple combinations of input parameters to evaluate the performance of the searching algorithm under different case scenarios and to record the outputs to generate a dataset that will be used for data exploration, model fitting, analysis, summarization and finally for performance comparison with other algorithms.

3.1 Character Distance Algorithm Implementation

A publicly available working code for the algorithm didn't exist; therefore, it was required to implement the algorithm from scratch for this thesis, which was done using Python 3. Several functions were implemented following the notes and pseudo code from the original document and posteriorly tested thoroughly using the library Pytest.

Function	**Description**
Search0	Secondary function, it is triggered when there are no instances of the pivot character in the input pattern.
Search1	Secondary function, similar to Search0 is triggered when there is only one instance of the pivot character in the pattern.
Search 2+	The main function, it uses the character distances to find occurrences of the pattern in the text.

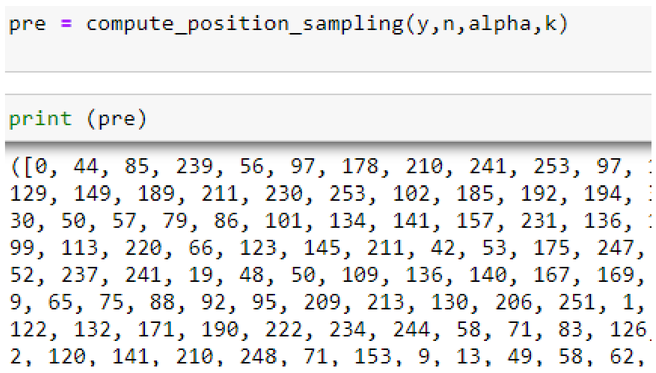
Table 1: Search functions

In table 1 the three searching functions can be seen; the function Search2+ will be used when there are at least two pivot characters present in the pattern; otherwise, the other two functions would be called depending on if there is one or zero occurrences of the pivot character.

Function	**Description**
get_position	Gets the original position in the input text.
compute_position_sampling	Computes the preprocessing step.
compute_distance_sampling	Samples the distances for the pattern.
compute_character_distance_sampling	Computes the Characters Distance sampled version of the text.
Verify	Verifies a match.

Table 2: Auxiliary functions.

In table 2 the auxiliary functions are shown. Some are used to preprocess the text and pattern, others to sample the positions of the characters where the pivot character is present, to restore the location of the character in the original text and to verify matches.



```
pre = compute_position_sampling(y,n,alpha,k)
```

```
print (pre)
```

```
([0, 44, 85, 239, 56, 97, 178, 210, 241, 253, 97,
129, 149, 189, 211, 230, 253, 102, 185, 192, 194,
30, 50, 57, 79, 86, 101, 134, 141, 157, 231, 136,
99, 113, 220, 66, 123, 145, 211, 42, 53, 175, 247,
52, 237, 241, 19, 48, 50, 109, 136, 140, 167, 169,
9, 65, 75, 88, 92, 95, 209, 213, 130, 206, 251, 1,
122, 132, 171, 190, 222, 234, 244, 58, 71, 83, 126
2, 120, 141, 210, 248, 71, 153, 9, 13, 49, 58, 62,
```

Figure 15: Pre processing stage results.

The Preprocessing space required by the Character Distances String-Matching will depend not only on the size of the document but also on the number of occurrences of pivot character; a typical value for the 1st ranked pivot would be around 8 percent of the size of the document.

In figure 15 we can observe an example of a typical output for the preprocessing function computePositionSampling, we can see the array storing the distances between the occurrences of the chosen pivot character, bounded by the segment constant 'k,' which in this case is 256.

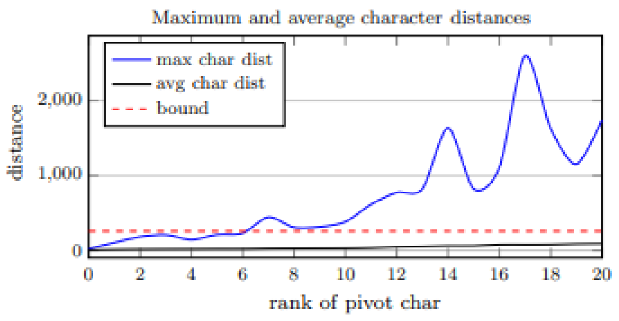


Figure 16: Maximum and average characters distances [10]

The figure 16 shows the average and maximum distances between two consecutive occurrences, computed for the most frequent characters in a text. On the x-axis, we have the characters which are ordered on the base of their rank value; on the y-axis, the searching time. The red line represents the bound k = 256.

3.2 Concepts

A sensitivity analysis of string matching algorithms allows us to test the robustness of the results of a model in the presence of input uncertainty. Some important aspects to take into consideration while measuring the performance of algorithms and that have been applied during the experiments are:

Efficiency measurement: In this case, we need to run the algorithm with the same parameters multiple times to compute the mean of values obtained over those several running times; we do this to minimize errors generated while measuring.

Stability Measurement: It refers to the variation that exists from the average of the running times, is computed as the standard deviation of those evaluated times during the observations. A big number of string matching algorithms seem to present low stability when encountering short patterns and the opposite in the presence of really long ones.

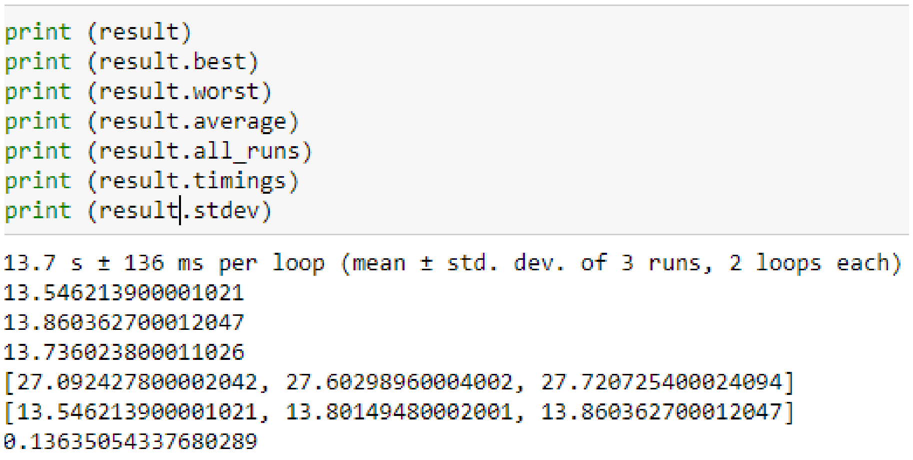
Flexibility measurement: Refers to how well the algorithm can adapt when it encounters different kinds of input data, for example, if it can maintain a good performance while dealing with both small and large texts, different kind of alphabet sizes, input patterns of various lengths, etc. Several algorithms can obtain excellent performance when dealing with a certain specific type of input data, but they underperform in other situations.

3.3 Experiments

The experiments were executed locally on a PC running Intel Core i7 4790 CPU with a @3.6GHz clock speed and 32 Gb of memory ddr3 3200 MHz. The operating system used was Windows 10, and during the experiments, only the regular background processes ran to decrease random variation, the time results were averages of 50 runs. The algorithms presented in the previous section were implemented using the Python programming language.

For the methodology in all tests, a one-factor-at-a-time experimental design was followed[19] to ensure the accuracy and effectiveness of tests and case scenarios testing that aims to simulate realistic user behavior. Comparisons have been performed in terms of pre-processing times, searching times, and space consumption; 50 instances of each set of parameters were generated to capture the variation of the algorithm when induced by the text itself and to be able to summarize the mean performance easily for each combination.

The Python function timeit() was used to obtain the pre-processing and searching times, and the values of the parameters 'number' and 'repeat' were set to -5 and -10 respectively to have more representative values and to comply with the principles of efficiency and stability mentioned before. The number parameter controls how many executions are done for each timing, and it helps to get representative timings as sometimes, at the moment of the specific measurement, some other application running in the background might affect the results. The 'repeat' argument controls how many timings will be produced, and its use is to get accurate statistics.



```
print (result)
print (result.best)
print (result.worst)
print (result.average)
print (result.all_runs)
print (result.timings)
print (result.stdev)
```

```
13.7 s ± 136 ms per loop (mean ± std. dev. of 3 runs, 2 loops each)
13.546213900001021
13.860362700012047
13.736023800011026
[27.092427800002042, 27.60298960004002, 27.720725400024094]
[13.546213900001021, 13.80149480002001, 13.860362700012047]
0.13635054337680289
```

Figure 17: Performance results obtained from the timeit() function

In figure 17 we can see an example of the results of measuring one combination of input parameters; the measurement was effectuated three times with two loops each one, and the values for the best, worst, average, and standard deviation were obtained, as well as the individual measuring values. For the experiments, the average value was taken.

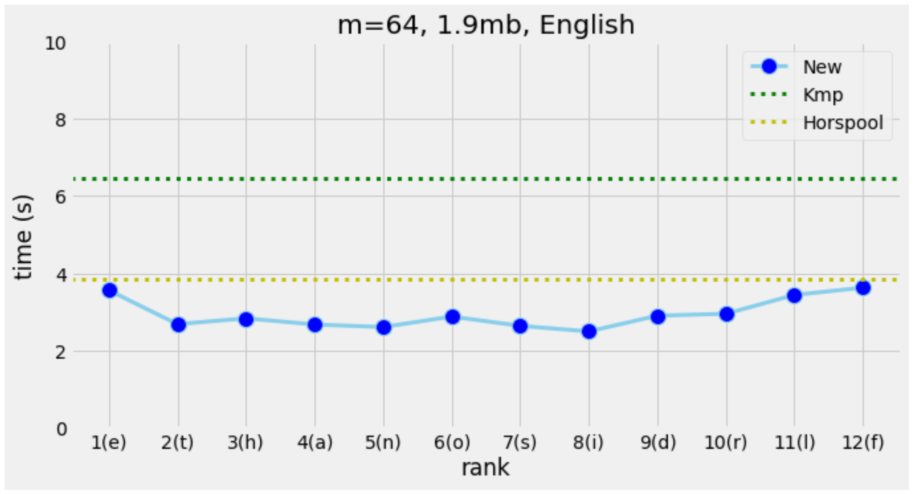


Figure 18: Performance of the Algorithms in the case of long patterns

The figure 18 shows an example of a search time comparison in the case of a long pattern, the comparison is done between pivot characters and also with the KMP and Horspool algorithms.

3.4 Parameters

The parameters selected to test the robustness of the algorithm are:

Text length: Measured in number of characters, at first look is the variable that will most likely affect the searching time the most. Different text files of variate sizes have been selected and are explained in the following subsection.

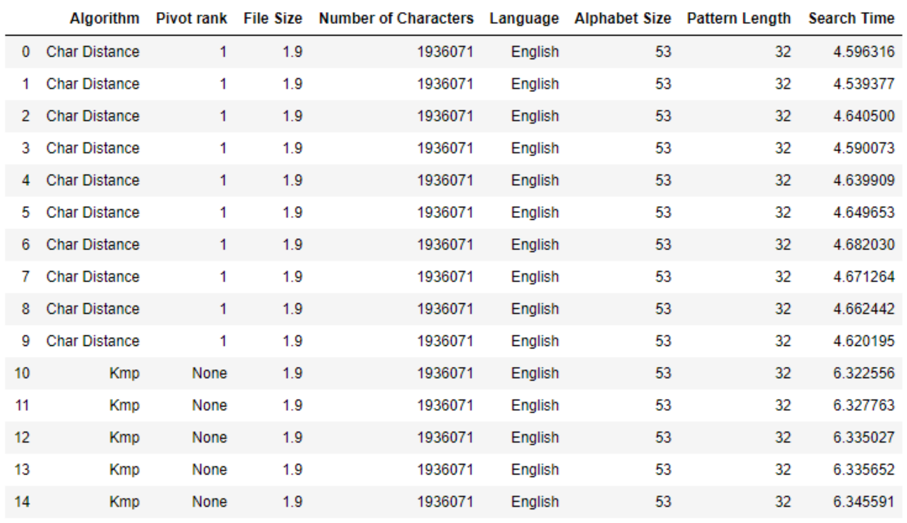
Pattern length: Measured in number of characters, processed online by the algorithm to search for occurrences of the pivot character in it. Several length values are used.

Language: Multiple texts of different genres in 3 different languages are used as input parameters (English, German and Lithuanian).

Alphabet size: Measured by number of unique characters in the text, medium to large sizes of alphabets were used. The alphabet was extracted from each text.

The pivot rank: The top 5 most frequent characters in the text are selected as input values and they are used by the proposed algorithm to calculate de distances between the occurrences of the character in the text.

The segment size: Parameter 'k' used by the characters distance algorithm has been set to 256 for all the experiments as this is the value recommended by the author of the paper for non trivial analyses as it allows to store a character distance of up to 256 in one byte and that way resources are optimized.



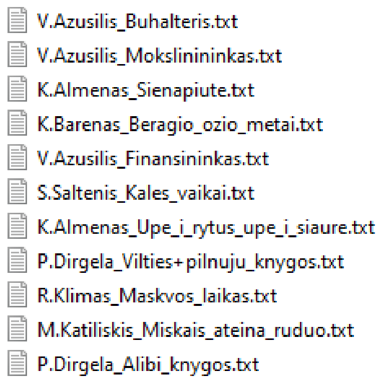
	Algorithm	Pivot rank	File Size	Number of Characters	Language	Alphabet Size	Pattern Length	Search Time
0	Char Distance	1	1.9	1936071	English	53	32	4.596316
1	Char Distance	1	1.9	1936071	English	53	32	4.539377
2	Char Distance	1	1.9	1936071	English	53	32	4.640500
3	Char Distance	1	1.9	1936071	English	53	32	4.590073
4	Char Distance	1	1.9	1936071	English	53	32	4.639909
5	Char Distance	1	1.9	1936071	English	53	32	4.649653
6	Char Distance	1	1.9	1936071	English	53	32	4.682030
7	Char Distance	1	1.9	1936071	English	53	32	4.671264
8	Char Distance	1	1.9	1936071	English	53	32	4.662442
9	Char Distance	1	1.9	1936071	English	53	32	4.620195
10	Kmp	None	1.9	1936071	English	53	32	6.322556
11	Kmp	None	1.9	1936071	English	53	32	6.327763
12	Kmp	None	1.9	1936071	English	53	32	6.335027
13	Kmp	None	1.9	1936071	English	53	32	6.335652
14	Kmp	None	1.9	1936071	English	53	32	6.345591

Figure 19: Dataframe with fixed parameters.

In figure 20 we can see an example of a dataframe showing performance measurements of the searching times with a fixed set of parameters repeated several times for two different algorithms.

3.5 Text Corpora

For this work, two main text repositories were used to obtain the texts from, Project Gutenberg, which is an online repository of ebooks; this was the source for the English and German texts, and the second repository used was epaveldas.lt, which was used to obtain the texts in Lithuanian.



V.Azusilis_Buhalteris.txt
V.Azusilis_Mokslinininkas.txt
K.Almenas_Sienapiute.txt
K.Barenas_Beragio_ozio_metai.txt
V.Azusilis_Finansininkas.txt
S.Saltenis_Kales_vaikai.txt
K.Almenas_Upe_i_rytus_upe_i_siaure.txt
P.Dirgela_Vilties+pilnuju_knygos.txt
R.Klimas_Maskvos_laikas.txt
M.Katiliskis_Miskais_ateina_ruduo.txt
P.Dirgela_Alibi_knygos.txt

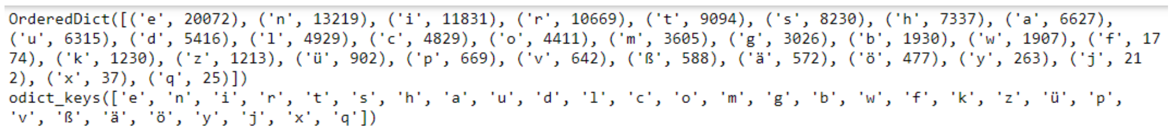
Figure 20: Some of the collected Lithuanian texts

In total, 150 texts were collected, 100 from the Project Gutenberg repository and 50 from the epaveldas.lt repository. Each one of the three languages (English, German, Lithuanian) counts with 50 books. The file sizes of the documents vary between 100kb up to 5Mb, and the alphabet size varies from 77 to 153 (Alphabets also include symbols).

The library Markovify was used to create 50 different versions for each book in the three languages; the goal is to be able to create simulated texts that retain a natural language structure and to produce several instances of the same set of parameters but with different texts.

The top 10 most frequent words of each document for patterns of lengths that range from 2 to 8 were also stored in an extra file to be used posteriorly by the text processing script.

A count of the occurrences of each character in the document is also applied to posteriorly calculate the frequencies for the selection of the pivot characters with higher rank values.



```
OrderedDict([('e', 20072), ('n', 13219), ('i', 11831), ('r', 10669), ('t', 9094), ('s', 8230), ('h', 7337), ('a', 6627),
('u', 6315), ('d', 5416), ('l', 4929), ('c', 4829), ('o', 4411), ('m', 3605), ('g', 3026), ('b', 1930), ('w', 1907), ('f', 17
74), ('k', 1230), ('z', 1213), ('ü', 902), ('p', 669), ('v', 642), ('ß', 588), ('ä', 572), ('ö', 477), ('y', 263), ('j', 21
2), ('x', 37), ('q', 25)])
odict_keys(['e', 'n', 'i', 'r', 't', 's', 'h', 'a', 'u', 'd', 'l', 'c', 'o', 'm', 'g', 'b', 'w', 'f', 'k', 'z', 'ü', 'p',
'v', 'ß', 'ä', 'ö', 'y', 'j', 'x', 'q'])
```

Figure 21: Number of character occurrences in a document.

3.6 Scripting

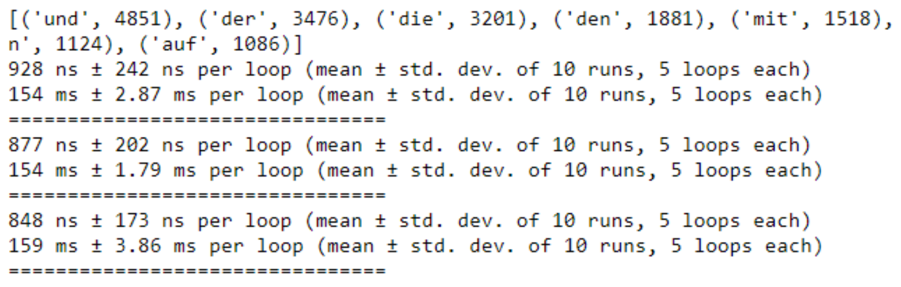
To automatize the process 2 Python scripts were coded. The steps are as follow:

Script 1:

- The script loads the 150 documents (50 per language) one by one.
- For each document 50 variations of the same text are created using the library Markovify.
- For patterns of length 8 or less the top 10 most frequent words are retrieved.
- For longer patterns 10 random patterns are extracted from the text.

Script 2:

- A list of all the input variables and its possible values are feed to the script and the library itertools is used to create a full cross product of all the possible combinations of values.
- The script loads one of the text files generated by the previous script together with the most frequent words and random patterns assigned to that text by the previous script.
- The values of all by one parameter are set to fixed values.
- The function timeit() is called to measure the preprocessing and searching time of the algorithm with respect to the document and the input variables, the values of the parameters 'number' and 'repeat' are set to -5 and -10 respectively. In this stage the required preprocessing space is also stored.
- the script stores the average of values and generates a data-frame row with the output of the function timeit().
- The same step is repeated 50 times for each combination of parameters before proceeding to the next combination.
- The resulting data-frames are merged to firstly generate the individual datasets for each language and posteriorly those are also merged to produce the final dataset of times for the Character Distance string-matching algorithm.



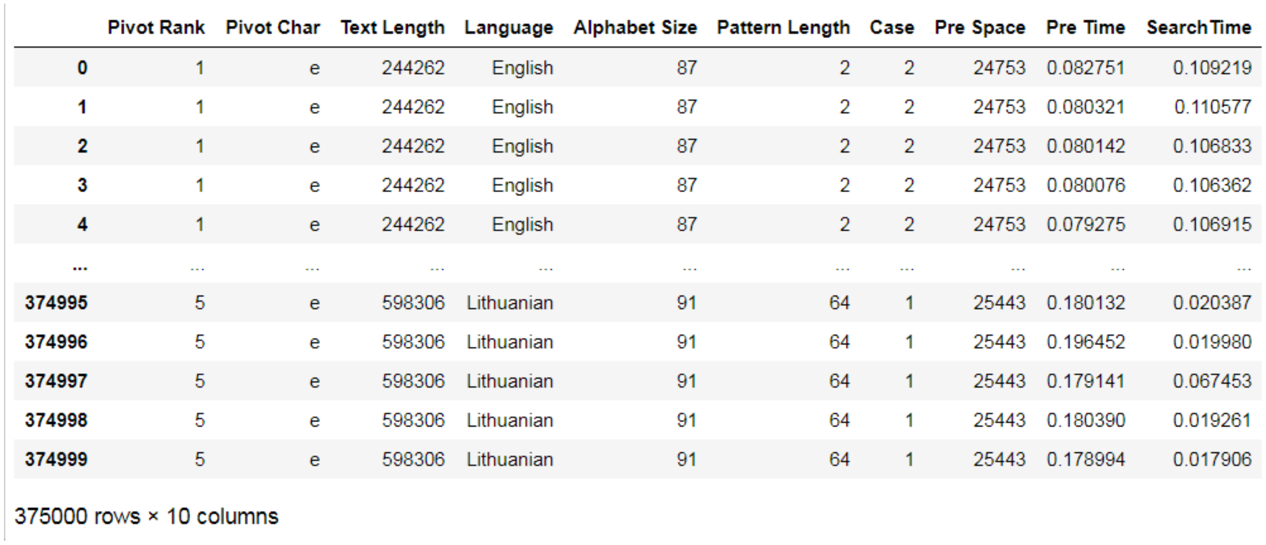
```
[('und', 4851), ('der', 3476), ('die', 3201), ('den', 1881), ('mit', 1518),
n', 1124), ('auf', 1086)]
928 ns ± 242 ns per loop (mean ± std. dev. of 10 runs, 5 loops each)
154 ms ± 2.87 ms per loop (mean ± std. dev. of 10 runs, 5 loops each)
================================
877 ns ± 202 ns per loop (mean ± std. dev. of 10 runs, 5 loops each)
154 ms ± 1.79 ms per loop (mean ± std. dev. of 10 runs, 5 loops each)
================================
848 ns ± 173 ns per loop (mean ± std. dev. of 10 runs, 5 loops each)
159 ms ± 3.86 ms per loop (mean ± std. dev. of 10 runs, 5 loops each)
================================
```

Figure 22: The second script in action

The figure 22 shows the timeit() function measuring the preprocessing and searching times of the most frequent German 3 character-length patterns with respect to the text.

3.7 Dataset

After all the steps mentioned in the previous section a dataset was generated from a full cross product of the available variables and 50 instances of each set of parameter combination were generated to capture the variation of the algorithm induced by the text.



	Pivot Rank	Pivot Char	Text Length	Language	Alphabet Size	Pattern Length	Case	Pre Space	Pre Time	Search Time
0	1	e	244262	English	87	2	2	24753	0.082751	0.109219
1	1	e	244262	English	87	2	2	24753	0.080321	0.110577
2	1	e	244262	English	87	2	2	24753	0.080142	0.106833
3	1	e	244262	English	87	2	2	24753	0.080076	0.106362
4	1	e	244262	English	87	2	2	24753	0.079275	0.106915
...	...	...	...	...	...	...	...	...	...	...
374995	5	e	598306	Lithuanian	91	64	1	25443	0.180132	0.020387
374996	5	e	598306	Lithuanian	91	64	1	25443	0.196452	0.019980
374997	5	e	598306	Lithuanian	91	64	1	25443	0.179141	0.067453
374998	5	e	598306	Lithuanian	91	64	1	25443	0.180390	0.019261
374999	5	e	598306	Lithuanian	91	64	1	25443	0.178994	0.017906

375000 rows × 10 columns

Table 3: Generated Dataset

3.7.1 Variables

Pivot Rank refers to the position in the rank of frequencies of the specific pivot character in the chosen text, the set of possible values is (1..5)

Pivot Char is the character that represents the before mentioned position in the rank of frequencies.

Text Length is the size of the text measured by the number of characters in it.

Language in this case can be one of three values English, Lithuanian, German.

Alphabet size refers to the unique set of characters in the text.

Pattern Length is the length of the input pattern, the possible lengths are (2..8, 16,32,64).

Case indicates when the function that calculates distances is being used (1) otherwise an auxiliary function is being used (2).

Pre Space is the extra space required for the algorithm to construct its partial index and is measured by the number of characters in it.

Pre Time is the preprocessing time required for the algorithm to create its partial index and it is measured in seconds.

Search Time is the time required for the algorithm to search for the matches of the pattern x in y.

3.8 Exploratory Data Analysis

In this subsection, a few explorations to the data will be performed to have a better understanding of its nuances, including plotting the density function, frequency of pivot characters and plots related to preprocessing space, preprocessing time and searching time in relation to some independent variables.

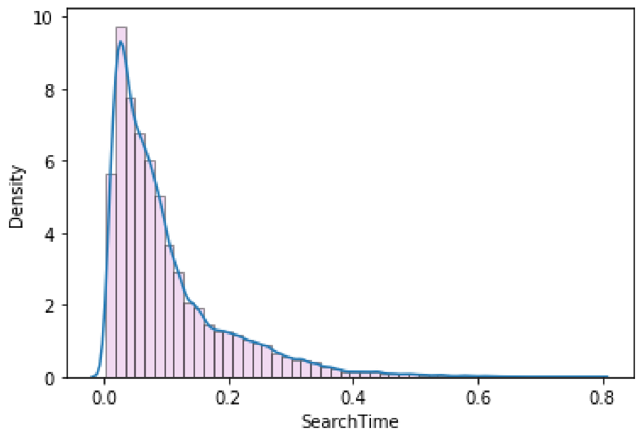


Figure 23: Density function

figure 23 shows the density function for the searching time of our dataset where we can notice most of the values are concentrated in the 0 - 0.2 range, we can also observe some right skewness.

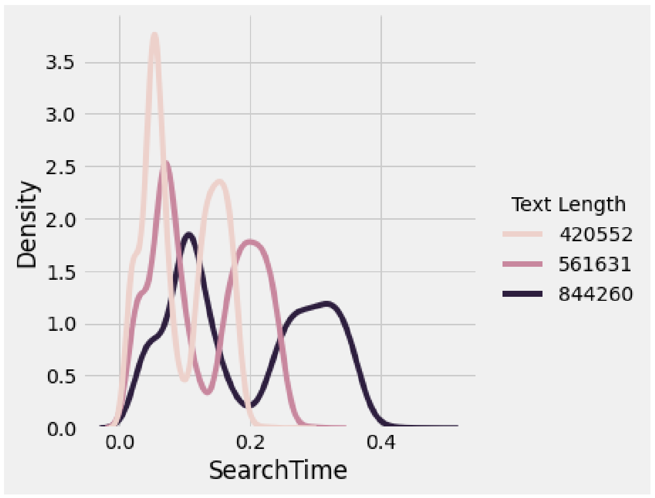
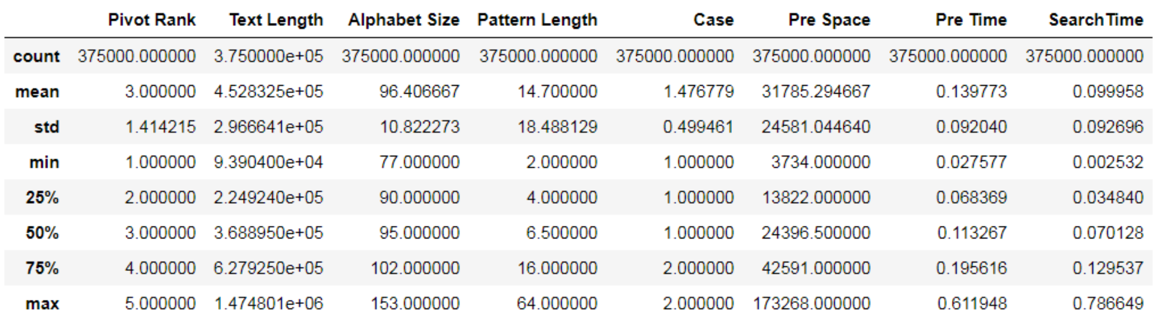


Figure 24: Density function per document

In figure 24 it can be seen that the density function has 2 peaks per each document, the explanation for this is that sometimes we cannot use the primary function of the algorithm as there is no presence of the pivot character in the pattern and therefore we cannot calculate distances so a slower auxiliary function is used instead.



	Pivot Rank	Text Length	Alphabet Size	Pattern Length	Case	Pre Space	Pre Time	SearchTime
count	375000.000000	3.750000e+05	375000.000000	375000.000000	375000.000000	375000.000000	375000.000000	375000.000000
mean	3.000000	4.528325e+05	96.406667	14.700000	1.476779	31785.294667	0.139773	0.099958
std	1.414215	2.966641e+05	10.822273	18.488129	0.499461	24581.044640	0.092040	0.092696
min	1.000000	9.390400e+04	77.000000	2.000000	1.000000	3734.000000	0.027577	0.002532
25%	2.000000	2.249240e+05	90.000000	4.000000	1.000000	13822.000000	0.068369	0.034840
50%	3.000000	3.688950e+05	95.000000	6.500000	1.000000	24396.500000	0.113267	0.070128
75%	4.000000	6.279250e+05	102.000000	16.000000	2.000000	42591.000000	0.195616	0.129537
max	5.000000	1.474801e+06	153.000000	64.000000	2.000000	173268.000000	0.611948	0.786649

Table 4: Descriptive statistics for the dataset

Table 4 shows the descriptive statistics of the dataset, some interesting characteristics can be noticed, for instance the broad margin of difference between the Min and Max searching times, as well as how the mean of the preprocessing time is higher than the mean of the searching time. Another interesting aspect seems to be that the preprocessing memory space needed by the algorithm seems to be of around 7 to 14 percent of the size of the provided text.

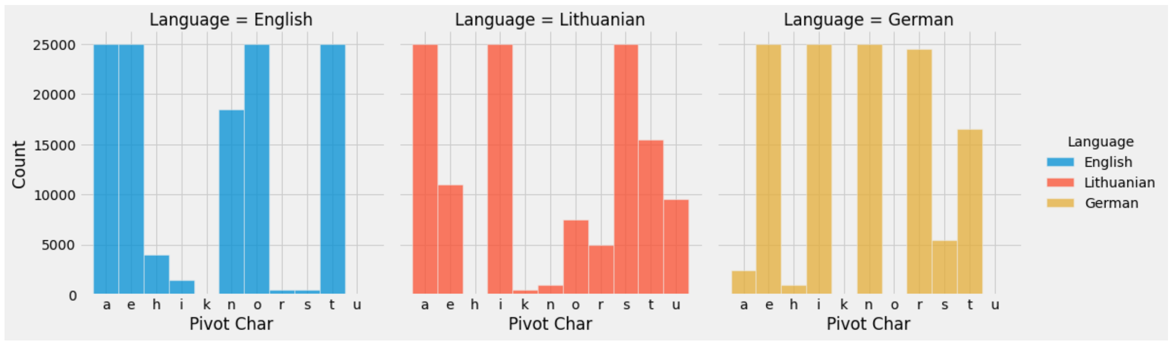


Figure 25: Frequencies of occurrences of the pivot characters by language

In figure 25 we can observe the number of occurrences of the selected pivot characters per language, it can be seen that only 9 characters have been used as pivots in English and in most of the cases the same five characters ('a','e','n','o','t') have been used. In the case of the Lithuanian language we can see that 10 characters have been used, with two of them only been selected in seldom occasions('k' and 'n'). For the German language 8 characters have been used and in most of the occasions it was the same 5 characters ('e','i','n','r','t') .

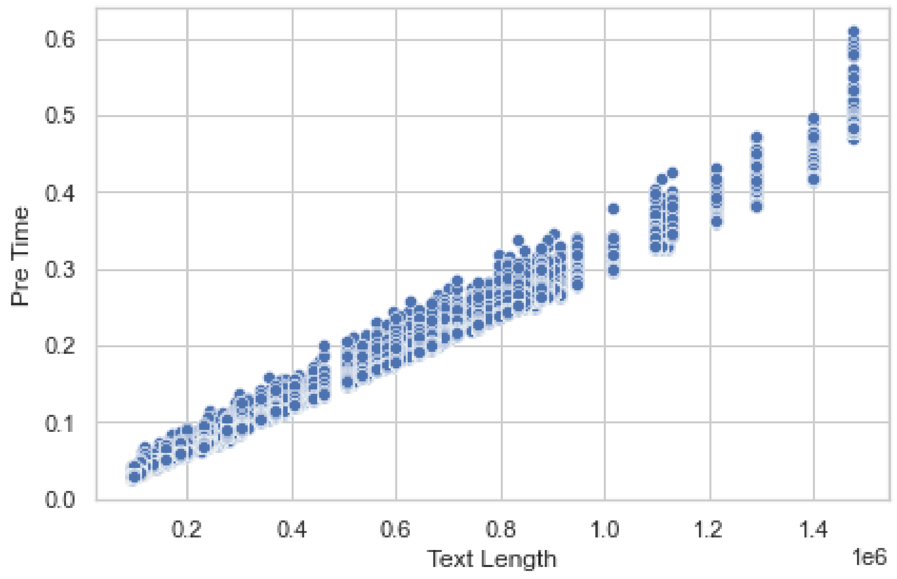


Figure 26: Pre processing time(s) in comparison with the size of the document

In figure 26 we can see the relationship between the text length by number of characters and the preprocessing time in seconds, it can be seen a clear linear relationship between the length and time.

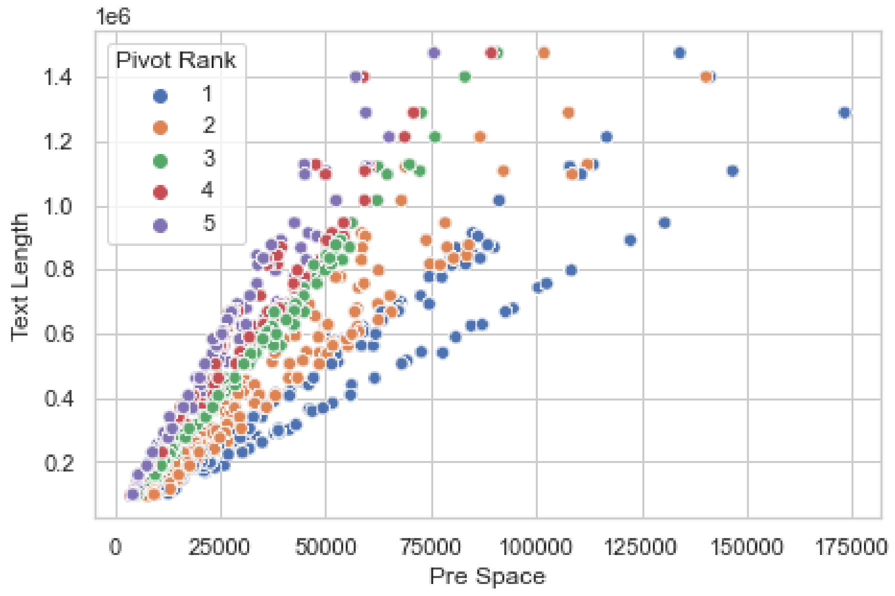


Figure 27: Space in number of characters in comparison with the pivot rank and text length

As we can see on figure 27 , the required space used by the preprocessing step depends mostly on two factors, the text length and the selection of the pivot character specially its rank as if the frequency of the pivot character is higher then more distances will be calculated and therefore more memory space will be required .

	case 1	case 2
Count	196208	178792
Percentage	52.3	47.7
Search Time (avg)	0.0546	0.1496

Table 5: Primary vs Auxiliary function cases

As the proposed algorithm needs to detect more than one instance of the selected pivot character in the pattern to be able to calculate any distance between occurrences of the pivot, a question arises about what to do when there are no instances or only one instance of the pivot character in the pattern, in this case an auxiliary function will be called, this function takes advantage of the preprocessing step but uses a classical approach[7] for the searching process, therefore in the cases where the secondary function is called, the searching times will increase. In the table 5 we can see the number of cases in our dataset where the Primary function (case 1) was used and the number of times when the auxiliary function (case 2) was called. It can be seen that the number of times when each function is called doesn't differ by a big margin and observing the averages of the searching times we can see that the primary function is almost three times as fast as the auxiliary one.

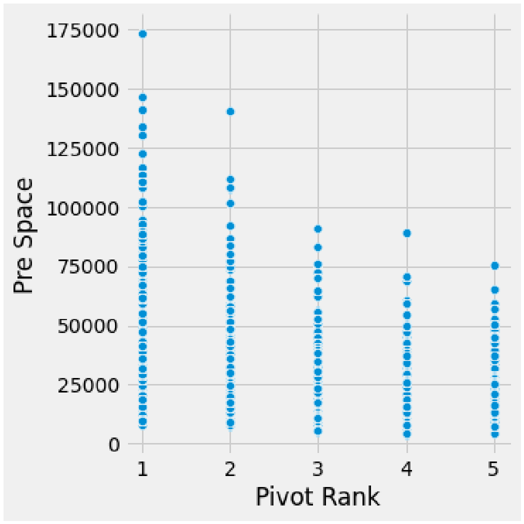


Figure 28: Pre processing space required by pivot character.

In figure 28 we can see that the selection of the pivot character affects the space in memory required by the algorithm. The highest-ranked pivot character requires more space in memory because it is the most frequent character, and that means that there are more occurrences of it in both the pattern and the text and more distances need to be stored.

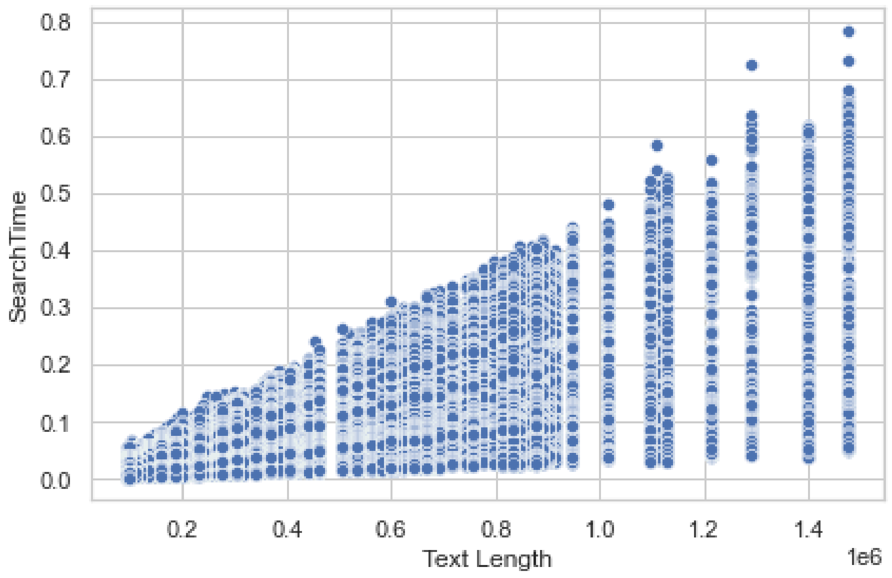


Figure 29: Searching time (s) in comparison with the Text size (chars)

In figure 29 we can observe the searching time in seconds for different text lengths expressed in number of characters, we can see that as the text length increases the searching time also increases in most of the cases however there are many exceptions and therefore probably other factors are involved including the selection of the rank of the pivot character, pattern length, alphabet size, etc.

3.9 Experimental Results

In this section firstly a model that predicts the searching times of the algorithm is explained and posteriorly several case scenarios with different combination of input parameters will be shown[14].

As the data seemed to have linear characteristics, a simple linear regression was tried; however, it was under-fitting; a quadratic polynomial regression was then tried, which fitted quite well to the data; one possible explanation of this can be that because the algorithm uses two procedures depending on if there are enough characters to calculate distances or not, the existence of the second procedure seems to affect the shape of the density function.

3.9.1 Polynomial Regression

Polynomial is a form of regression analysis in which the relationship between the independent variable x and the dependent variable y is modelled as an nth degree polynomial.The formula is as follows:

$$y(x, w) = w_0 + w_1 x + w_2 x^2 + \ldots + w_m x^m = \sum_{j=0}^{m} w_j x^j \tag{1}$$

Where m represents the order of the polynomial and x^j expresses x raised to the power j. The coefficient of the polynomial is represented by w.

In our case the value of the polynomial degree m is specified to be m=2. Then the polynomial becomes a quadratic polynomial(2).

$$y(x, w) = w_0 + w_1 x + w_2 x^2 = \sum_{j=0}^{2} w_j x^j \tag{2}$$

The PolynomialFeatures class of scikit-learn is used to convert the original features into their higher order terms. Then the training is done using Linear Regression.

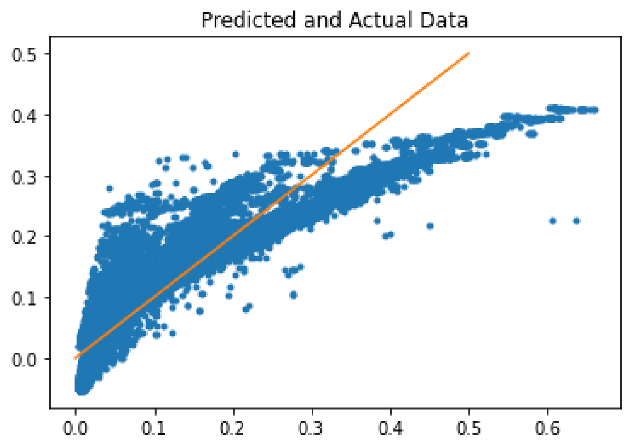


Figure 30: Model with Linear Regression

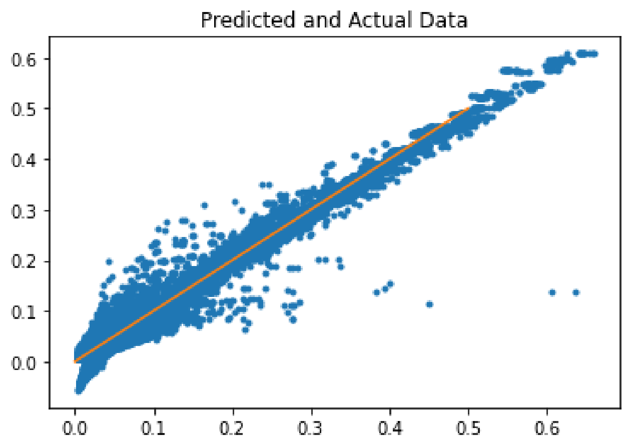


Figure 31: Model with Polynomial Regression

In figure 30 we can see the results from applying linear regression to the data; the model seems to be under-fitting; the opposite can be said about the polynomial regression of degree 2 (quadratic) seen in figure 31 which seems to fit the data quite well and is the model that was chosen.

3.9.2 Metrics

The metrics used to evaluate the model are explained in this section.

$$\text{MAE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{3}$$

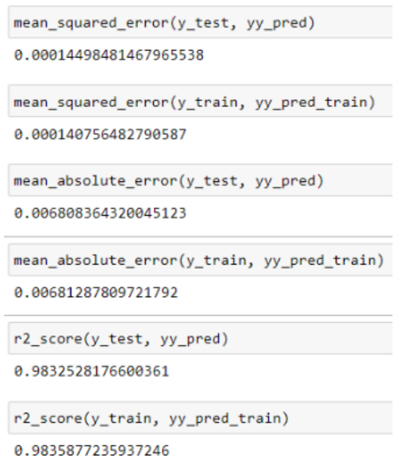
The Mean absolute error or MAE (3) where y_i represents the true value, $\hat{y}_i$ represents the predicted value and n represents the number of values. It averages the absolute differences between predictions and actual values. It gives an idea of how wrong the predictions were. The measure gives an idea of the magnitude of the error, but it does not explain the direction of it (e.g., over or under predicting). The Smaller the value of MAE, the better the accuracy of the predictive model.

$$\mathrm{MSE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y})^2 \tag{4}$$

The Mean Squared Error or MSE (4) is much like the mean absolute error in that it provides a gross idea of the magnitude of error, y_i represents the true value, $\hat{y}_i$ represents the predicted value and n represents the number of values. The Smaller the value of MSE, the better the accuracy of the predictive model.

Taking the square root of the mean squared error converts the units back to the original units of the output variable and can be meaningful for description and presentation. This is called the Root Mean Squared Error (or RMSE).

The R2 (or R Squared) metric provides an indication of the goodness of fit of a set of predictions to the actual values. In statistical literature, this measure is called the coefficient of determination.



```
mean_squared_error(y_test, yy_pred)
0.00014498481467965538

mean_squared_error(y_train, yy_pred_train)
0.000140756482790587

mean_absolute_error(y_test, yy_pred)
0.006808364320045123

mean_absolute_error(y_train, yy_pred_train)
0.00681287809721792

r2_score(y_test, yy_pred)
0.9832528176600361

r2_score(y_train, yy_pred_train)
0.9835877235937246
```

Figure 32: Metrics for the model

In figure figure 32 the metric evaluation results are shown, We can see that in the case of the three metrics the difference between the training error and the testing error is minimum. Mean Squared Error, Mean absolute error and r2 score were applied receiving satisfactory values therefore this model was selected.

3.9.3 Model experiments

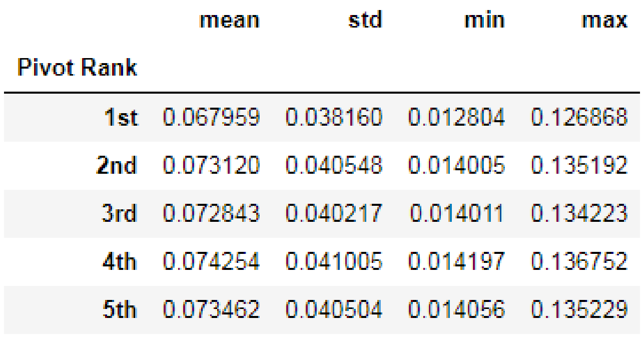
After selecting the model it was used to perform the sensitivity analysis of the robustness of the proposed algorithm, observing how the different input values affect the resulting searching time with previously unseen data.

3.9.4 Case scenarios

A common approach is used for the sensitivity analysis namely changing one-factor-at-a-time (OAT)[19], to see the effect this change produces on the output, therefore sensitivity is measured by monitoring changes in the output after fixing all of the input values with the exception of one.

In this section, several case scenarios[16] will be shown, the values of some of the input variables will be fixed, and the output in terms of the searching time of the algorithm will be observed.

Case scenario 1: Searching time in relation to the rank of the pivot characters with different text sizes in English. Values: Pivot Rank = (1..5), Alphabet Size = 80, Text Length = (100 different text lengths), Language = English, Pattern Length = 8



Pivot Rank	mean	std	min	max
1st	0.067959	0.038160	0.012804	0.126868
2nd	0.073120	0.040548	0.014005	0.135192
3rd	0.072843	0.040217	0.014011	0.134223
4th	0.074254	0.041005	0.014197	0.136752
5th	0.073462	0.040504	0.014056	0.135229

Figure 33: Descriptive statistics for case scenario 1

It can be seen in the figure figure 34, and in the summary of figure figure 33 that in this case scenario, there is an apparent difference in the mean of the searching times mainly between the highest-ranked character and the others; the mean was obtained from 100 values of text sizes, and four were selected to be plotted, in the plot the difference in search time is more noticeable in the largest text documents.

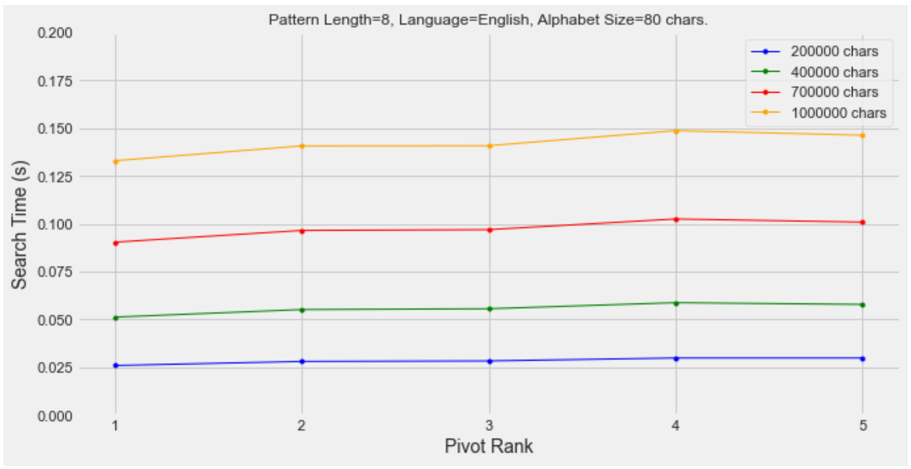
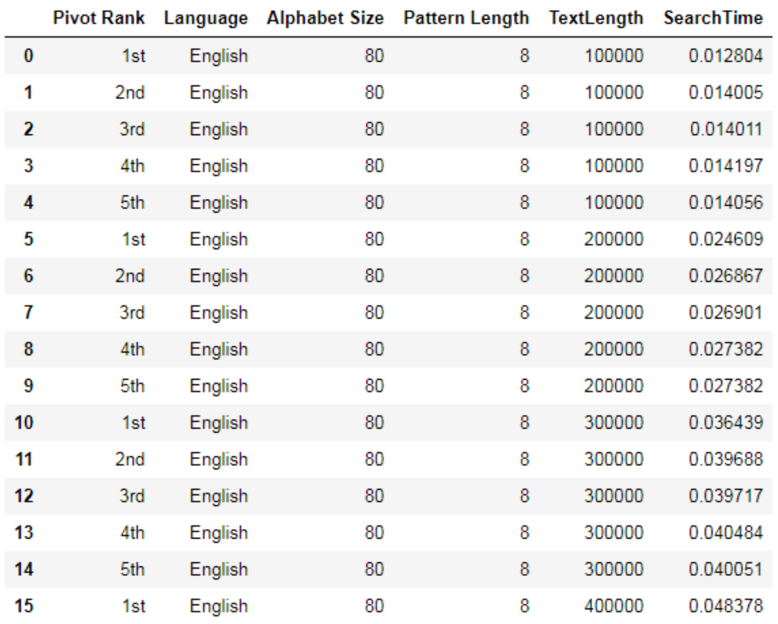


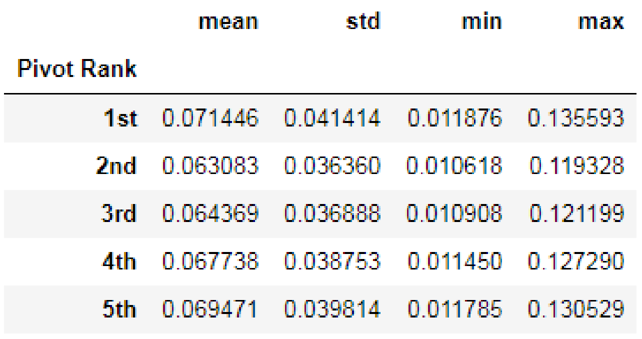
Figure 34: Plot for case scenario 1



	Pivot Rank	Language	Alphabet Size	Pattern Length	TextLength	SearchTime
0	1st	English	80	8	100000	0.012804
1	2nd	English	80	8	100000	0.014005
2	3rd	English	80	8	100000	0.014011
3	4th	English	80	8	100000	0.014197
4	5th	English	80	8	100000	0.014056
5	1st	English	80	8	200000	0.024609
6	2nd	English	80	8	200000	0.026867
7	3rd	English	80	8	200000	0.026901
8	4th	English	80	8	200000	0.027382
9	5th	English	80	8	200000	0.027382
10	1st	English	80	8	300000	0.036439
11	2nd	English	80	8	300000	0.039688
12	3rd	English	80	8	300000	0.039717
13	4th	English	80	8	300000	0.040484
14	5th	English	80	8	300000	0.040051
15	1st	English	80	8	400000	0.048378

Figure 35: Instances of the case scenario 1

Case scenario 2: Searching time in relation to the rank of the pivot characters with different text sizes in German. Values: Pivot Rank = (1..5), Alphabet Size = 80, Text Length = (100 different text lengths), Language = German, Pattern Length = 16



	mean	std	min	max
Pivot Rank				
1st	0.071446	0.041414	0.011876	0.135593
2nd	0.063083	0.036360	0.010618	0.119328
3rd	0.064369	0.036888	0.010908	0.121199
4th	0.067738	0.038753	0.011450	0.127290
5th	0.069471	0.039814	0.011785	0.130529

Figure 36: Descriptive statistics for case scenario 2

In this case, we switch the language to German, and the pattern length to 16, and the rest of the variables will have the same values as the previous case. In the summary of means on figure 36 we can observe that unlike in the previous case, now the highest-ranked character is the slowest; this would be something interesting to investigate further as the reason might be the language, longer pattern length, or both. In figure figure 37 we can see that the second and third pivot characters are better candidates than the typical first one.

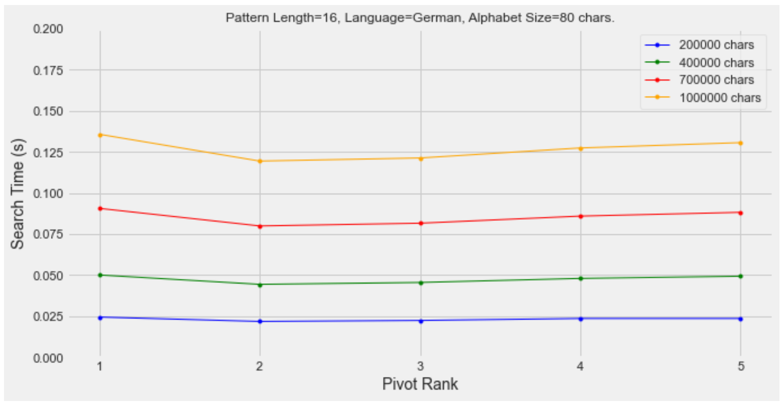
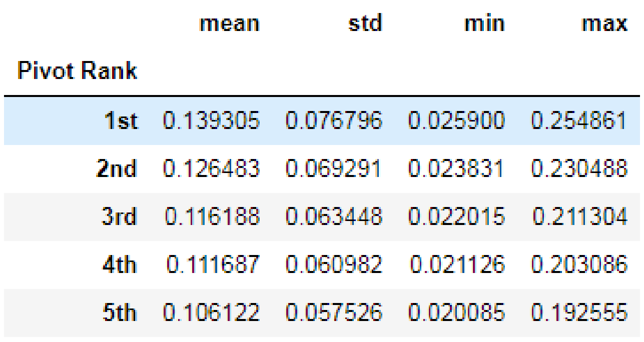


Figure 37: Plot for case scenario 2

Case scenario 3: Searching time in relation to the rank of the pivot characters with different text sizes in Lithuanian. Values: Pivot Rank = (1..5), Alphabet Size = 80, Text Length = (100 different text lengths), Language = Lithuanian, Pattern Length = 2



Pivot Rank	mean	std	min	max
1st	0.139305	0.076796	0.025900	0.254861
2nd	0.126483	0.069291	0.023831	0.230488
3rd	0.116188	0.063448	0.022015	0.211304
4th	0.111687	0.060982	0.021126	0.203086
5th	0.106122	0.057526	0.020085	0.192555

Figure 38: Descriptive statistics for case scenario 3

In this case, we switch the language to Lithuanian, and the pattern length to 2, and the rest of the variables will have the same values as the previous case. In the summary of means on figure 38 we can observe that in this short-length pattern, the 5th pivot character is the best alternative, and once again, the highest-ranked character is the slowest; this is also something that can be investigated further. In figure figure 39 we can see clearly the lower-ranked pivot characters are better candidates, in this case, so it seems apparent that a short length pattern prefers a lower-ranked pivot, and this could be a good hypothesis to investigate.

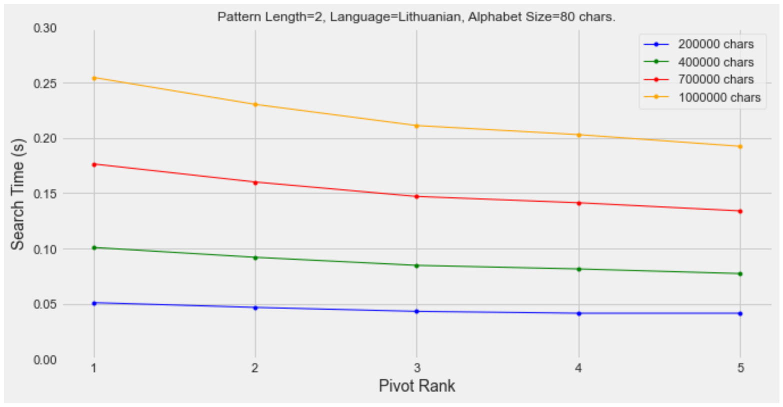


Figure 39: Plot for case scenario 3

Case scenario 4: Searching time in relation to multiple pattern lengths (2..8,16,32,64) and languages (English, German, Lithuanian,) Values: Pivot Rank = 1, Alphabet Size = 120, Text Length = 1000000

In figure 40 we can see clearly how the longer patterns take considerably less time than the shorter ones; also, there is an apparent minor but semi-constant difference between the searching times of the three languages; therefore, it could be an interesting relationship to investigate further.

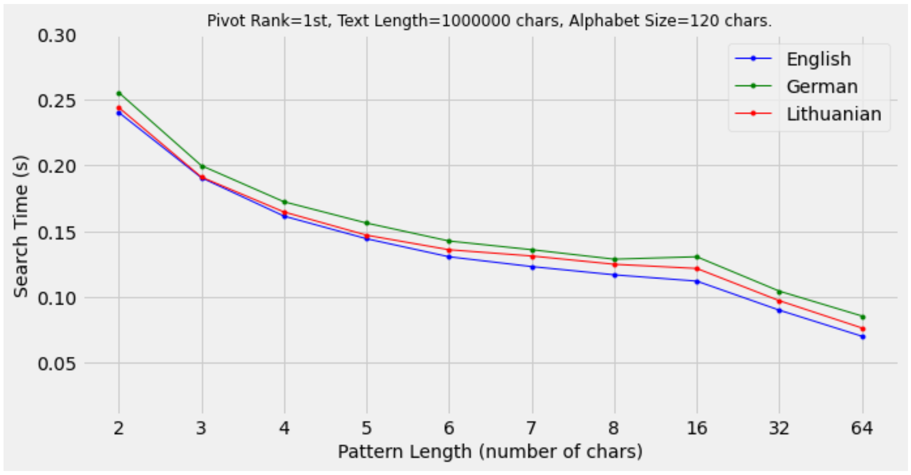


Figure 40: Plot for case scenario 4

Case scenario 5: Searching time in relation to Pattern Length per each language and with 2 different text sizes. Values: Pivot Rank = 1, Alphabet size = 80, Text Length = (1000000, 5000000), Language = (English, German, Lithuanian), Pattern Length = (2..8,16,32,64)

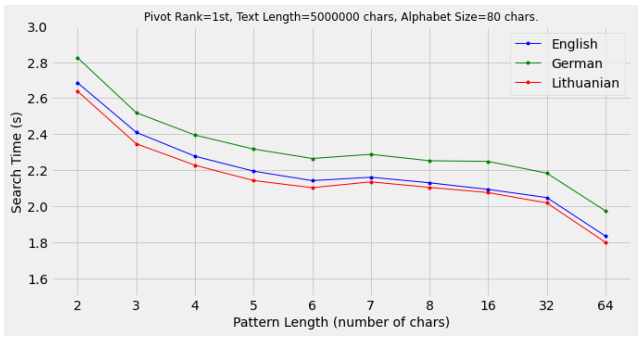


Figure 41: Search time over a text of 1m chars length

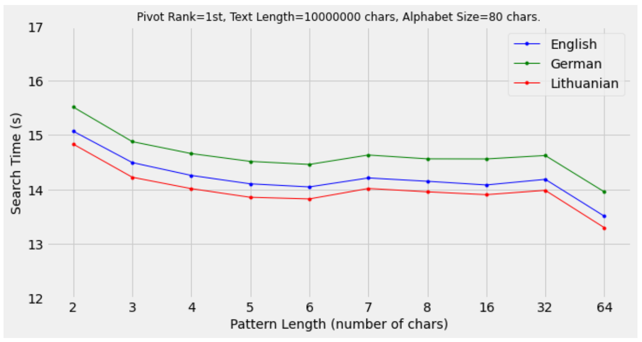


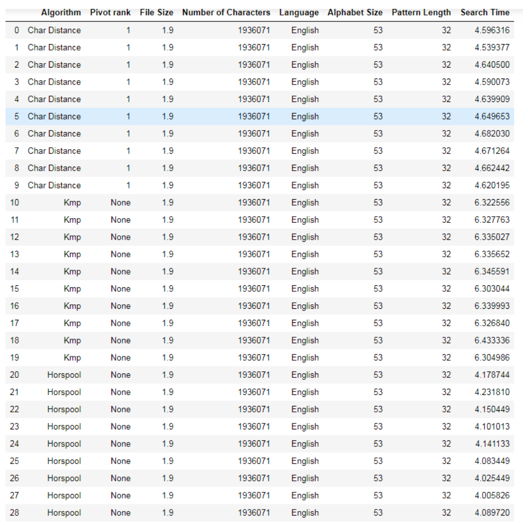
Figure 42: Search time over a text of 5m chars length

In both figure figure 41 and figure 42, we can see that when the pattern length is larger, the searching time is reduced; this is explained by the fact that when we input larger patterns, the chances of having several instances of the pivot characters in it are increased, as well as the chances to have unique combinations of pivot distances which means that the algorithm would skip unwanted combinations. When talking about the language, we can observe that when the text file is larger, the German files start to take a longer Searching time.

These were just a few of the possible combinations that can be explored further; many interesting characteristics were found from the plots and descriptive statistics.

3.10 Algorithms comparison

In this section, the Characters Distance algorithm will be matched with the two classic algorithms mentioned in previous sections of this thesis, which are the Knuth-Morris-Pratt algorithm and the Boyer-Moore-Horspool algorithm. Following the same procedure as with the character distance algorithm, the searching times of both of these algorithms were recorded from experiments, this way at the end, three datasets in total were generated, one for the new algorithm, which consists of 375000 rows and two smaller datasets of 75000 rows one for the Knuth-Morris-Pratt algorithm and one for the Boyer-Moore-Horspool algorithm. The reason for the smaller dataset is because those two algorithms don't require a pivot character. In figure 43 we can see a dataframe with combined instances of the three algorithms.



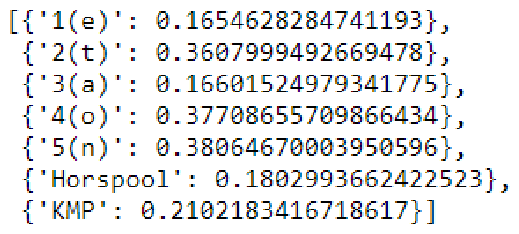
	Algorithm	Pivot rank	File Size	Number of Characters	Language	Alphabet Size	Pattern Length	Search Time
0	Char Distance	1	1.9	1936071	English	53	32	4.596316
1	Char Distance	1	1.9	1936071	English	53	32	4.539377
2	Char Distance	1	1.9	1936071	English	53	32	4.640500
3	Char Distance	1	1.9	1936071	English	53	32	4.590073
4	Char Distance	1	1.9	1936071	English	53	32	4.639909
5	Char Distance	1	1.9	1936071	English	53	32	4.649653
6	Char Distance	1	1.9	1936071	English	53	32	4.682030
7	Char Distance	1	1.9	1936071	English	53	32	4.671264
8	Char Distance	1	1.9	1936071	English	53	32	4.662442
9	Char Distance	1	1.9	1936071	English	53	32	4.620195
10	Kmp	None	1.9	1936071	English	53	32	6.322556
11	Kmp	None	1.9	1936071	English	53	32	6.327763
12	Kmp	None	1.9	1936071	English	53	32	6.335027
13	Kmp	None	1.9	1936071	English	53	32	6.335652
14	Kmp	None	1.9	1936071	English	53	32	6.345591
15	Kmp	None	1.9	1936071	English	53	32	6.303044
16	Kmp	None	1.9	1936071	English	53	32	6.339993
17	Kmp	None	1.9	1936071	English	53	32	6.326840
18	Kmp	None	1.9	1936071	English	53	32	6.433336
19	Kmp	None	1.9	1936071	English	53	32	6.304986
20	Horspool	None	1.9	1936071	English	53	32	4.178744
21	Horspool	None	1.9	1936071	English	53	32	4.231810
22	Horspool	None	1.9	1936071	English	53	32	4.150449
23	Horspool	None	1.9	1936071	English	53	32	4.101013
24	Horspool	None	1.9	1936071	English	53	32	4.141133
25	Horspool	None	1.9	1936071	English	53	32	4.083449
26	Horspool	None	1.9	1936071	English	53	32	4.025449
27	Horspool	None	1.9	1936071	English	53	32	4.005826
28	Horspool	None	1.9	1936071	English	53	32	4.089720

Figure 43: Search imes for the three algorithms

Continuing with the comparisons started in the previous section, we introduce several case scenarios where the values of some of the input variables will be fixed, and the output in terms of the searching time of the algorithm will be observed and compared among the three suggested algorithms.

3.10.1 Case Scenarios

Case scenario 1: Performance of the algorithms in response to the following input values: Text Length = 1212000, Language = English, Pattern Length = 7



```
[{'1(e)': 0.1654628284741193},
 {'2(t)': 0.3607999492669478},
 {'3(a)': 0.16601524979341775},
 {'4(o)': 0.37708655709866434},
 {'5(n)': 0.38064670003950596},
 {'Horspool': 0.1802993662422523},
 {'KMP': 0.2102183416718617}]
```

Figure 44: Case 1 of algorithm comparisons

We can observe in figure 44 that the Character Distance algorithm outperforms the other two algorithms when using the first and third in the rank of the pivots, however when selecting the other pivots the searching time almost doubles in time.

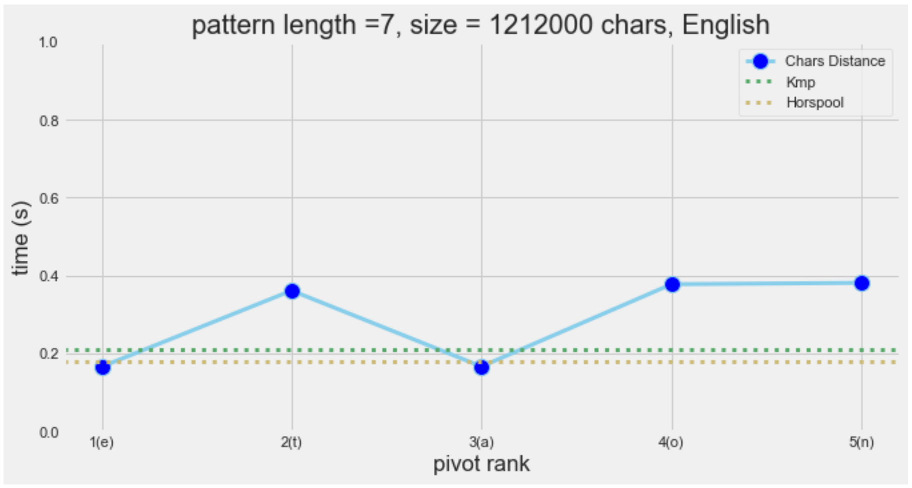


Figure 45: Plot for the case 1 of algorithm comparisons

Case scenario 2: Performance of the algorithms in response to the following input values: Text Length = 210000, Language = German, Pattern Length = 3

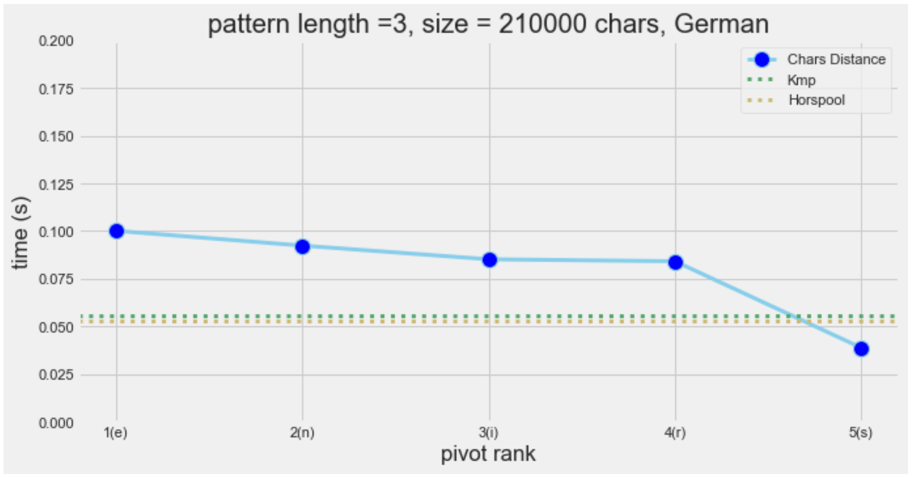
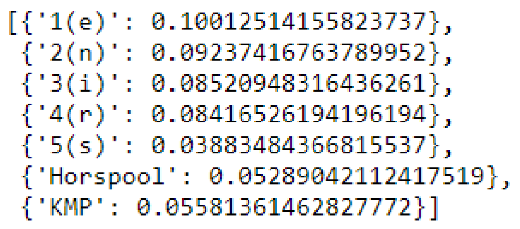


Figure 46: Case 2 of algorithm comparisons

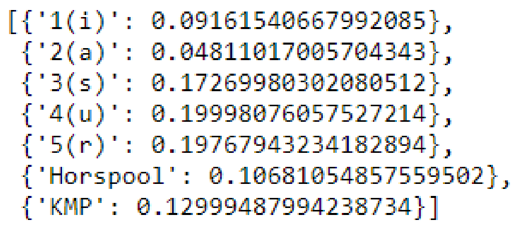
We can observe in figure 47 that the Character Distance algorithm is slower with all but one of the pivot characters, this confirms the presumption that the new algorithm would perform poorly with short patterns as it cannot calculate any distances and must relay on a slower auxiliary function for the searching process.



```
[{'1(e)': 0.10012514155823737},
 {'2(n)': 0.09237416763789952},
 {'3(i)': 0.08520948316436261},
 {'4(r)': 0.08416526194196194},
 {'5(s)': 0.03883484366815537},
 {'Horspool': 0.05289042112417519},
 {'KMP': 0.05581361462827772}]
```

Figure 47: Plot for the case 2 of algorithm comparisons

Case scenario 3: Performance of the algorithms in response to the following input values: Text Length = 700000, Language = Lithuanian, Pattern Length = 10



```
[{'1(i)': 0.09161540667992085},
 {'2(a)': 0.04811017005704343},
 {'3(s)': 0.17269980302080512},
 {'4(u)': 0.19998076057527214},
 {'5(r)': 0.19767943234182894},
 {'Horspool': 0.10681054857559502},
 {'KMP': 0.12999487994238734}]
```

Figure 48: Case 3 of algorithm comparisons

We can observe in figure 49 that the Character Distance algorithm is only faster than the other with the top 2 pivot characters, this seems to a common case with medium length patterns.

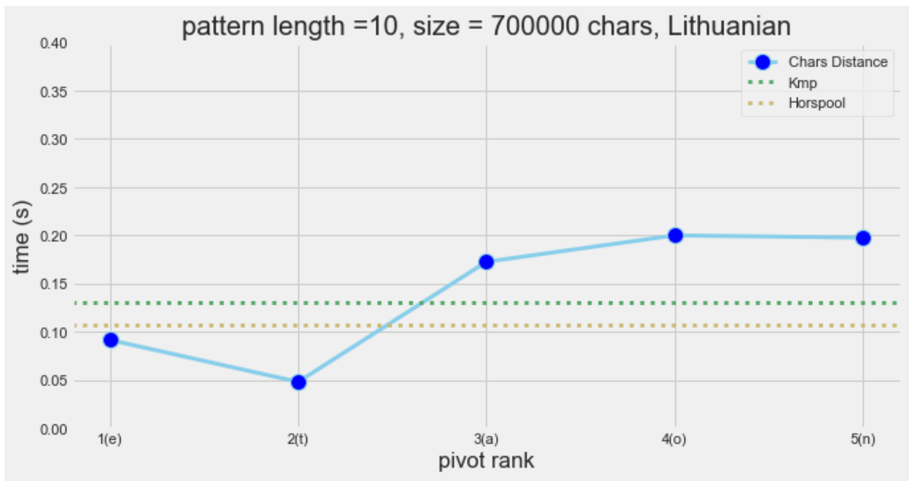


Figure 49: Plot for the case 3 of algorithm comparisons

Case scenario 4: Performance of the algorithms in response to the following input values: Text Length = 860000, Language = English, Pattern Length = 64

We can observe in figure 51 that the Character Distance algorithm is greatly superior to the other algorithms when it encounters really long patterns as the positions of distance are so unique that they allow the algorithm to discard non matching patterns faster.

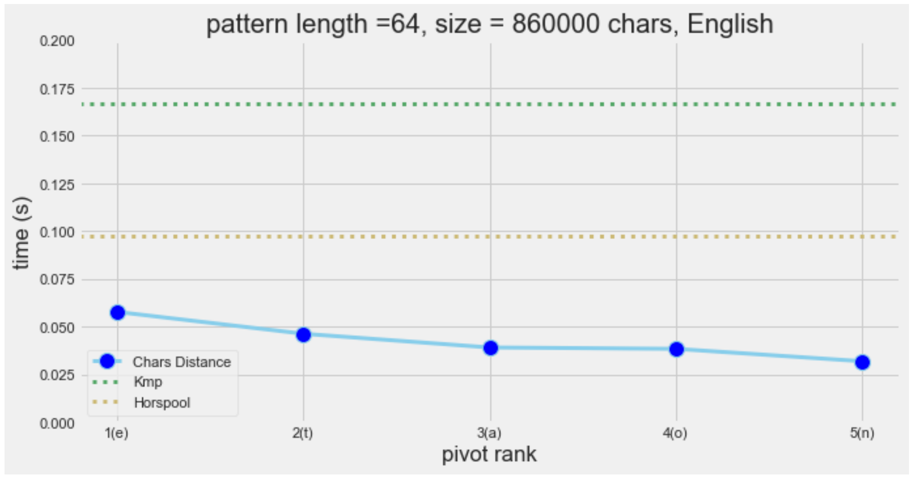
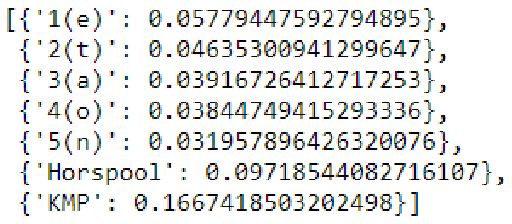


Figure 50: Case 4 of algorithm comparisons

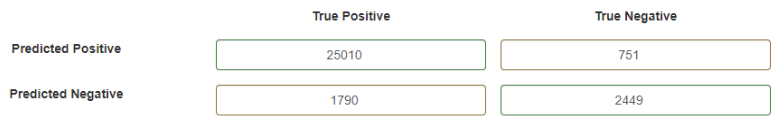


```
[{'1(e)': 0.05779447592794895},
 {'2(t)': 0.04635300941299647},
 {'3(a)': 0.03916726412717253},
 {'4(o)': 0.03844749415293336},
 {'5(n)': 0.03195789642632007б},
 {'Horspool': 0.09718544082716107},
 {'KMP': 0.1667418503202498}]
```

Figure 51: Plot for the case 4 of algorithm comparisons

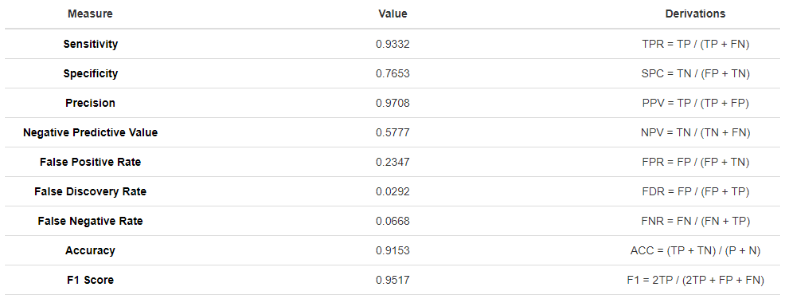
3.10.2 Summary

We can observe that in most cases, the newly proposed algorithm outperforms the classical ones when the most efficient pivot character is selected. A classifier was implemented using logistic regression to compare the proposed Characters Distance algorithm against the Boyer Moore Horspool algorithm; we took 75000 instances from each of the algorithms datasets and combined them for a total of 150k instances divided into a training set of 120k instances and a test set of 30k; the metrics can be seen in figure 52 and figure 53. From the metrics, we can see that the Characters Distance algorithm (True Positive) was of 25010 instances (predicted positive) + 1790 (predicted negative) against the BMH algorithm 2449 instances (predicted negative) + 1790 (predicted positive) of the BMH algorithm. In most cases, the newly proposed algorithm is the recommended option.



	True Positive	True Negative
Predicted Positive	25010	751
Predicted Negative	1790	2449

Figure 52: Confusion Matrix



Measure	Value	Derivations
Sensitivity	0.9332	TPR = TP / (TP + FN)
Specificity	0.7653	SPC = TN / (FP + TN)
Precision	0.9708	PPV = TP / (TP + FP)
Negative Predictive Value	0.5777	NPV = TN / (TN + FN)
False Positive Rate	0.2347	FPR = FP / (FP + TN)
False Discovery Rate	0.0292	FDR = FP / (FP + TP)
False Negative Rate	0.0668	FNR = FN / (FN + TP)
Accuracy	0.9153	ACC = (TP + TN) / (P + N)
F1 Score	0.9517	F1 = 2TP / (2TP + FP + FN)

Figure 53: Metrics for the logistic regression model

When the pivot character is not efficiently selected like in the case of a random selection, then the classical algorithms can outperform the new one.

It would be of great interest to compare the new algorithm with Offline string matching algorithms[15] or algorithms that make use of automata for their matchings as those present better-searching speed performance[12], although with the caveat of having to create a full index of the texts.

4 Conclusions

Performance analysis of the newly proposed online algorithm was presented in this paper, scanning multiple texts and measuring its preprocessing time, space required, and searching time, with multiple values and iterations applied to the parameters text length, pivot character, alphabet size, language, and pattern length. A dataset was created that was used to fit a linear regression model. Comparison of the Character Distance Sampling Algorithm was made against two of the most important classical algorithms in the field, Knuth–Morris–Pratt and Boyer–Moore–Horspool.

Several case scenarios were discussed in this thesis that attempt to be a guideline about when to use different parameter values for the new algorithm depending on the situation and also when to use the new algorithm in comparison with choosing a classical one.

The pivot character is the parameter that presented the most interesting outcomes as it seems to be explaining an underlying structure of the languages; in future works, it would be interesting to try different combinations when selecting the pivot character; in this thesis, the top five most frequent characters in the specific text was the strategy used; however, several other options could be tried, for instance merging two pivots or applying other conditions on the selection besides frequency.

Some important observations are that in cases when the algorithm doesn't encounter any occurrences of the pivot character or only encounters one, the performance of the algorithm is affected drastically as it needs to use a secondary function to complete the searching process; on the other hand, we observed that an ideal setup for the algorithm is when it encounters large patterns with several occurrences of the pivot character as the distances between the pivot characters will be less common and therefore easier to locate.

Another interesting observation is that the preprocessing time is directly connected with both the text size as well as with the rank of the pivot character; choosing a low ranked pivot character will produce a faster pre-processing time as well as occupying less space in memory but will in most cases reduce the searching speed performance.

The parameter that affects the most in relation to searching time is the length of the text as expected; however, we also observed that other aspects such as pattern length, language, and especially which pivot character is selected seem to affect the outcome greatly.

The newly proposed algorithm presented better searching times than both of the other alternatives in most of the cases with the exception of when smaller patterns of less than 4 characters were given. This fact made that the comparisons between the algorithm parameter values itself would be produce more interesting results than the comparison with the other algorithms.

4.1 Future Work

Even though interesting results have been obtained, many ideas that are outside the scope of this thesis could be explored in future works. For instance, more Online String Matching algorithms could be added to the grid of comparisons to have a bigger spectrum of choices, as well to add further parameter combinations such as broadening the selection of languages, utilizing really small alphabets, lengthier patterns, and larger corpora.

One of the key parts of the newly proposed online algorithm, which is subsampling the alphabet, could also be explored further by using a combination of pivot characters to calculate the distances instead of having the pivot ranking of frequencies as the unique criterion for the subsampling.

Focusing on an offline approach could also be another alternative as some of the ideas as subsampling the alphabets could work both with online and offline string matching algorithms. On the other hand, modifying the algorithm to work not only with exact string matching problems but also with approximate methods could greatly increase the number of experiments and the use cases proposed in this thesis.

Finally, applying these ideas into a Big Data setting could be considered as well as those applications could benefit greatly of faster searching times.

References

[1] K. Al-Khamaiseh and S. ALShagarin. A survey of string matching algorithms. *Int. J. Eng. Res. Appl*, 4 (7):144–156, 2014.

[2] D. Cantone and S. Faro. Fast-search algorithms: New efficient variants of the boyer-moore pattern-matching algorithm. *J. Autom. Lang. Comb.*, 10(5/6):589–608, 2005.

[3] C. Charras. Bmh algorithm. https://www-igm.univ-mlv.fr/~lecroq/string/node14.html, 2020.

[4] C. Charras. Kmp algorithm. https://www-igm.univ-mlv.fr/~lecroq/string/node8.html, 2020.

[5] C. Charras, T. Lecrog, and J. D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. pages 55–64, 1998.

[6] F. Claude, G. Navarro, H. Peltola, L. Salmela, and J. Tarhio. String matching with alphabet sampling. *Journal of Discrete Algorithms*, 11:37–50, 2012.

[7] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[8] S. Faro and T. Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Computing Surveys (CSUR)*, 45(2):1–42, 2013.

[9] S. Faro, T. Lecroq, S. Borzi, S. Di Mauro, and A. Maggio. The string matching algorithms research tool. pages 99–111, 2016.

[10] S. Faro, F. P. Marino, and A. Pavone. Efficient online string matching based on characters distance text sampling. *Algorithmica*, 82(11):3390–3412, 2020.

[11] S. I. Hakak, A. Kamsin, P. Shivakumara, G. A. Gilkar, W. Z. Khan, and M. Imran. Exact string matching algorithms: Survey, issues, and future research directions. *IEEE Access*, 7:69614–69637, 2019.

[12] J. Holub, W. F. Smyth, and S. Wang. Fast pattern-matching on indeterminate strings. *Journal of Discrete Algorithms*, 6(1):37–50, 2008.

[13] P. Kalsi, H. Peltola, and J. Tarhio. Comparison of exact string matching algorithms for biological sequences. pages 417–426, 2008.

[14] T. Lecroq. Experimental results on string matching algorithms. *Software: Practice and Experience*, 25(7): 727–765, 1995.

[15] T. Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, 2007.

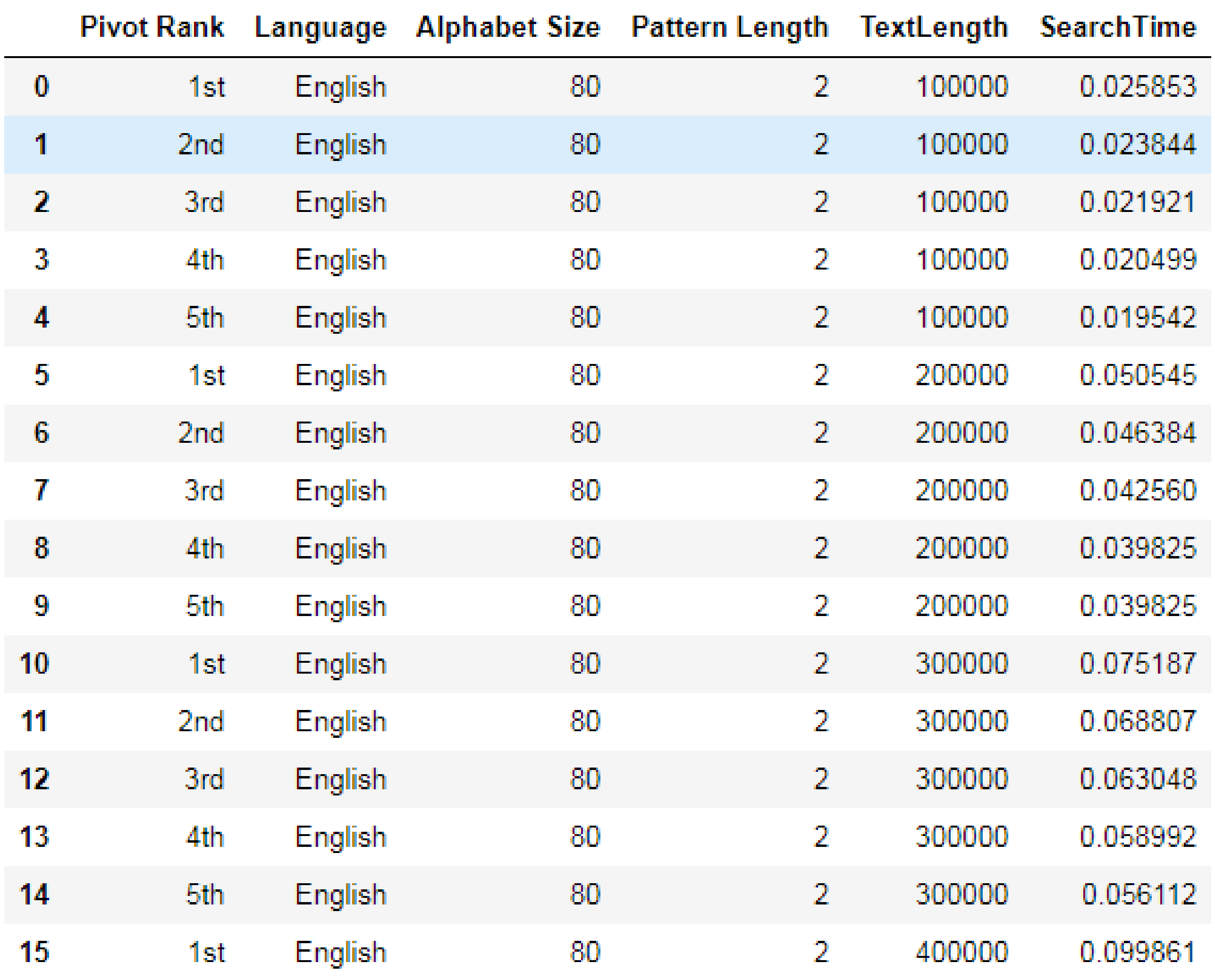
[16] K. Lei, Y. Ma, and Z. Tan. Performance comparison and evaluation of web development technologies in php, python, and node. js. In *2014 IEEE 17th international conference on computational science and engineering*, pages 661–668. IEEE, 2014.

[17] P. D. Michailidis and K. G. Margaritis. On-line string matching algorithms: Survey and experimental results. *International journal of computer mathematics*, 76(4):411–434, 2001.

[18] Ques10. Naive algorithm. https://www.ques10.com/p/41120/the-naive-string-matching-algorithms-1/, 2020.

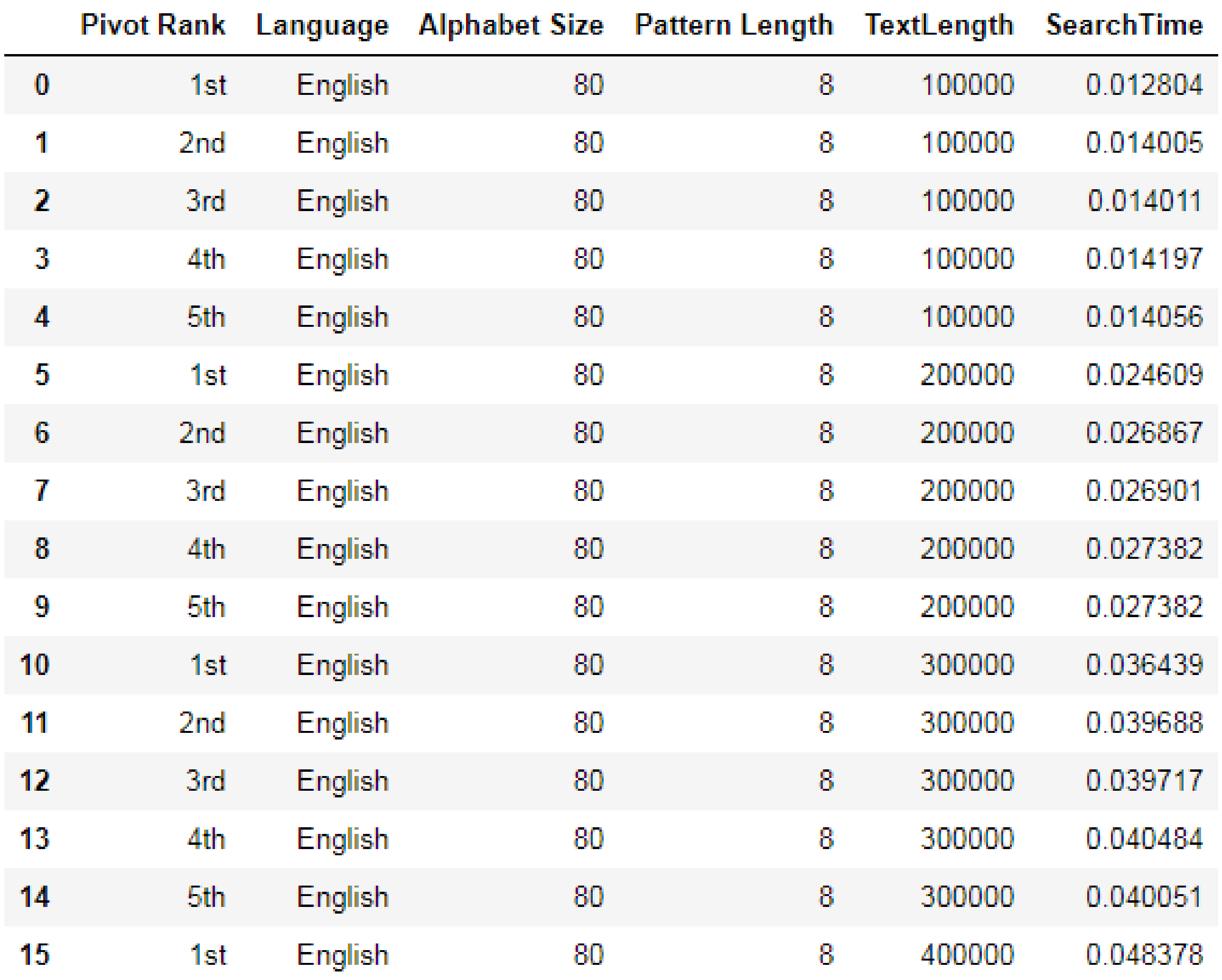
[19] A. Saltelli, M. Ratto, S. Tarantola, F. Campolongo, E. Commission, et al. Sensitivity analysis practices: Strategies for model-based inference. *Reliability Engineering & System Safety*, 91(10-11):1109–1125, 2006.

A Search Time Table for Patterns of length 2 in English



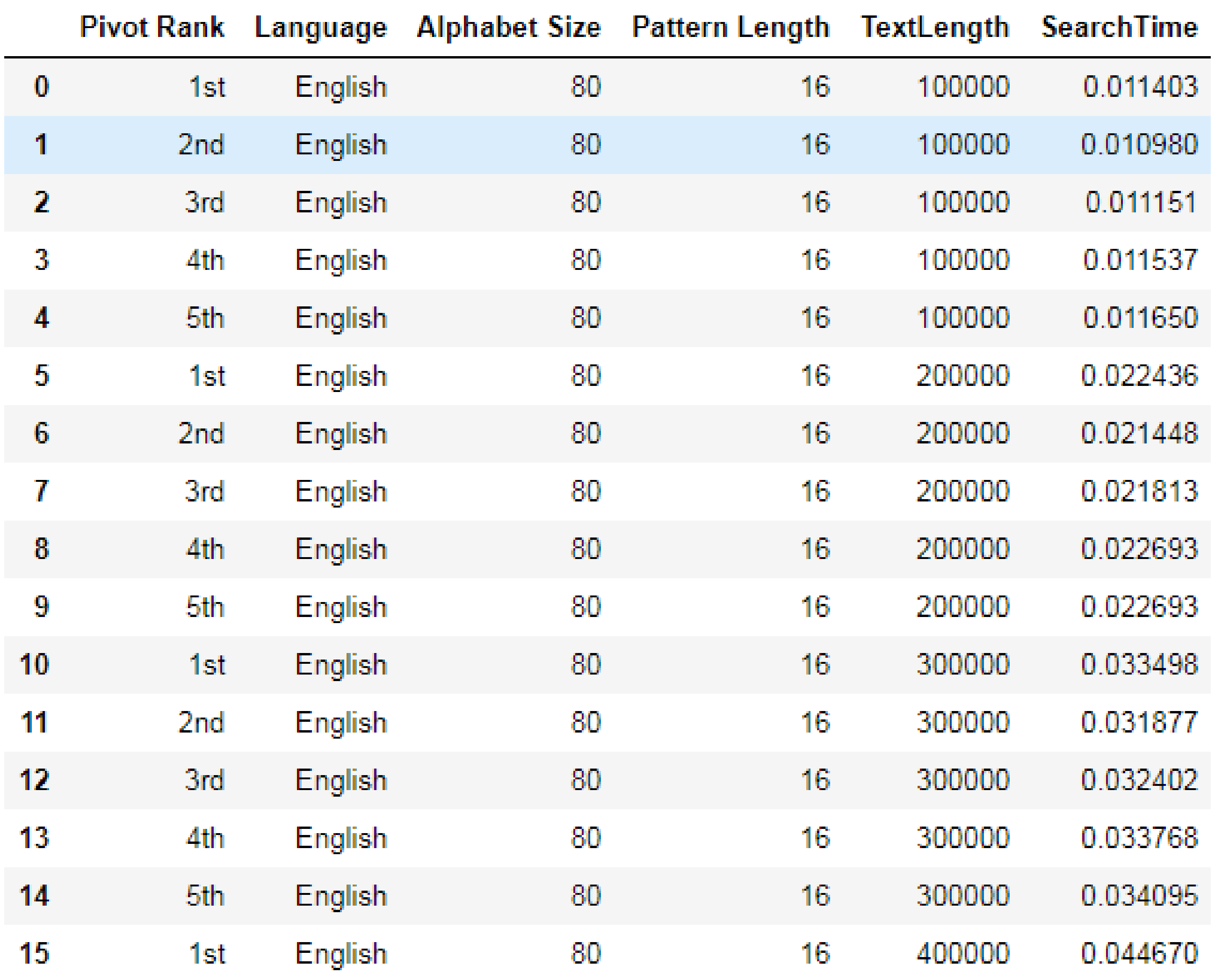
	Pivot Rank	Language	Alphabet Size	Pattern Length	TextLength	SearchTime
0	1st	English	80	2	100000	0.025853
1	2nd	English	80	2	100000	0.023844
2	3rd	English	80	2	100000	0.021921
3	4th	English	80	2	100000	0.020499
4	5th	English	80	2	100000	0.019542
5	1st	English	80	2	200000	0.050545
6	2nd	English	80	2	200000	0.046384
7	3rd	English	80	2	200000	0.042560
8	4th	English	80	2	200000	0.039825
9	5th	English	80	2	200000	0.039825
10	1st	English	80	2	300000	0.075187
11	2nd	English	80	2	300000	0.068807
12	3rd	English	80	2	300000	0.063048
13	4th	English	80	2	300000	0.058992
14	5th	English	80	2	300000	0.056112
15	1st	English	80	2	400000	0.099861

B Search Time Table for Patterns of length 8 in English



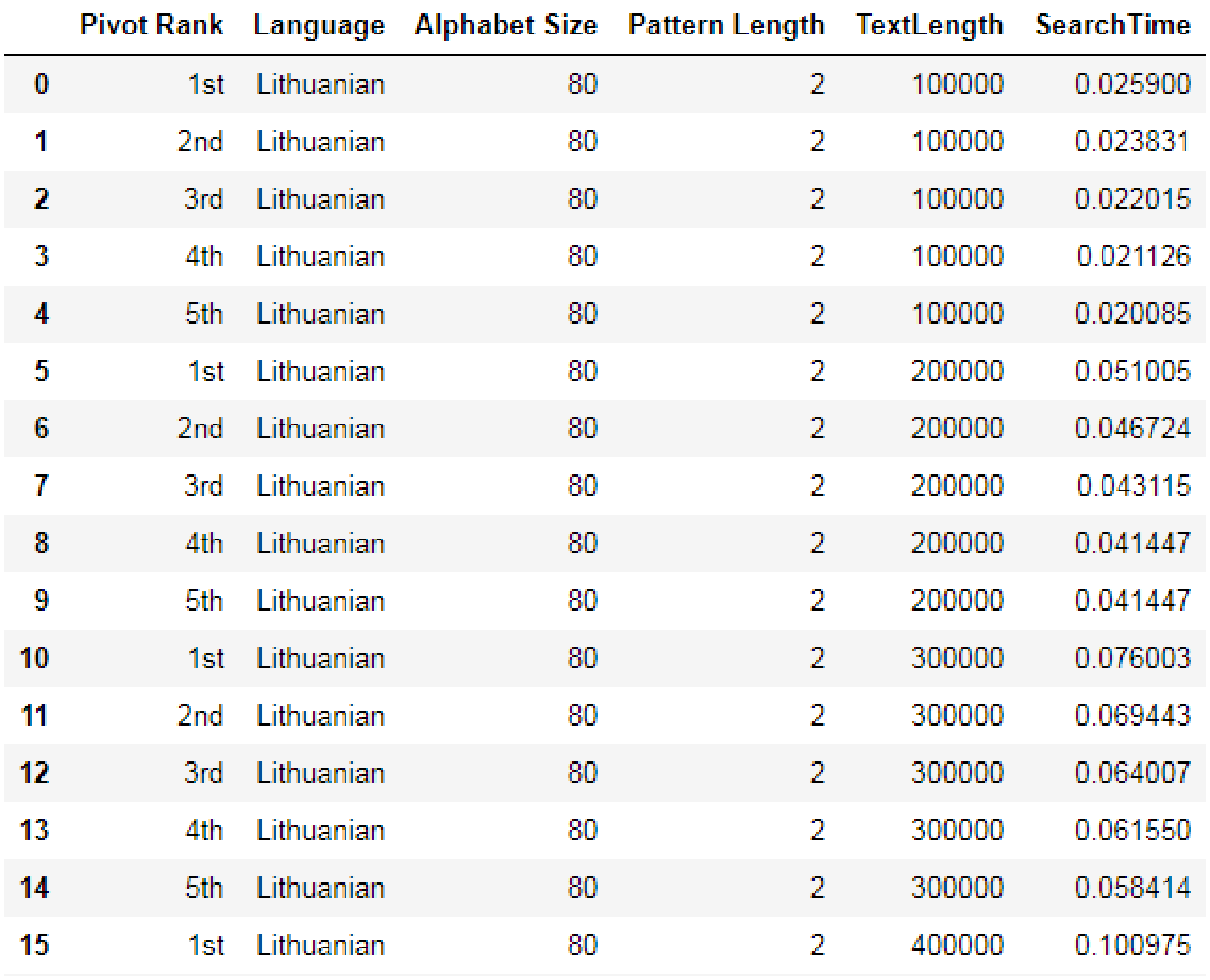
	Pivot Rank	Language	Alphabet Size	Pattern Length	TextLength	SearchTime
0	1st	English	80	8	100000	0.012804
1	2nd	English	80	8	100000	0.014005
2	3rd	English	80	8	100000	0.014011
3	4th	English	80	8	100000	0.014197
4	5th	English	80	8	100000	0.014056
5	1st	English	80	8	200000	0.024609
6	2nd	English	80	8	200000	0.026867
7	3rd	English	80	8	200000	0.026901
8	4th	English	80	8	200000	0.027382
9	5th	English	80	8	200000	0.027382
10	1st	English	80	8	300000	0.036439
11	2nd	English	80	8	300000	0.039688
12	3rd	English	80	8	300000	0.039717
13	4th	English	80	8	300000	0.040484
14	5th	English	80	8	300000	0.040051
15	1st	English	80	8	400000	0.048378

C Search Time Table for Patterns of length 16 in English



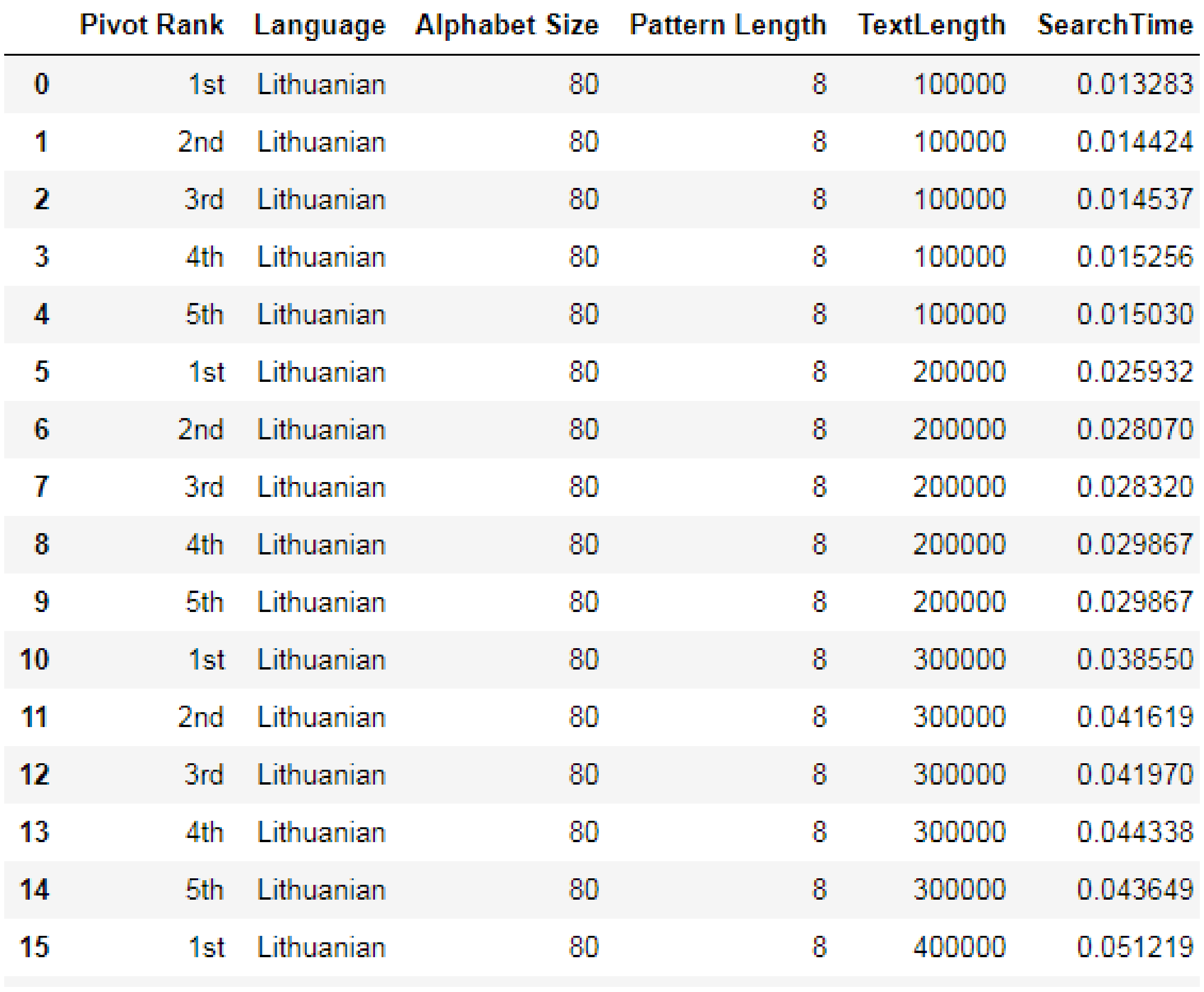
	Pivot Rank	Language	Alphabet Size	Pattern Length	TextLength	SearchTime
0	1st	English	80	16	100000	0.011403
1	2nd	English	80	16	100000	0.010980
2	3rd	English	80	16	100000	0.011151
3	4th	English	80	16	100000	0.011537
4	5th	English	80	16	100000	0.011650
5	1st	English	80	16	200000	0.022436
6	2nd	English	80	16	200000	0.021448
7	3rd	English	80	16	200000	0.021813
8	4th	English	80	16	200000	0.022693
9	5th	English	80	16	200000	0.022693
10	1st	English	80	16	300000	0.033498
11	2nd	English	80	16	300000	0.031877
12	3rd	English	80	16	300000	0.032402
13	4th	English	80	16	300000	0.033768
14	5th	English	80	16	300000	0.034095
15	1st	English	80	16	400000	0.044670

D Search Time Table for Patterns of length 2 in Lithuanian



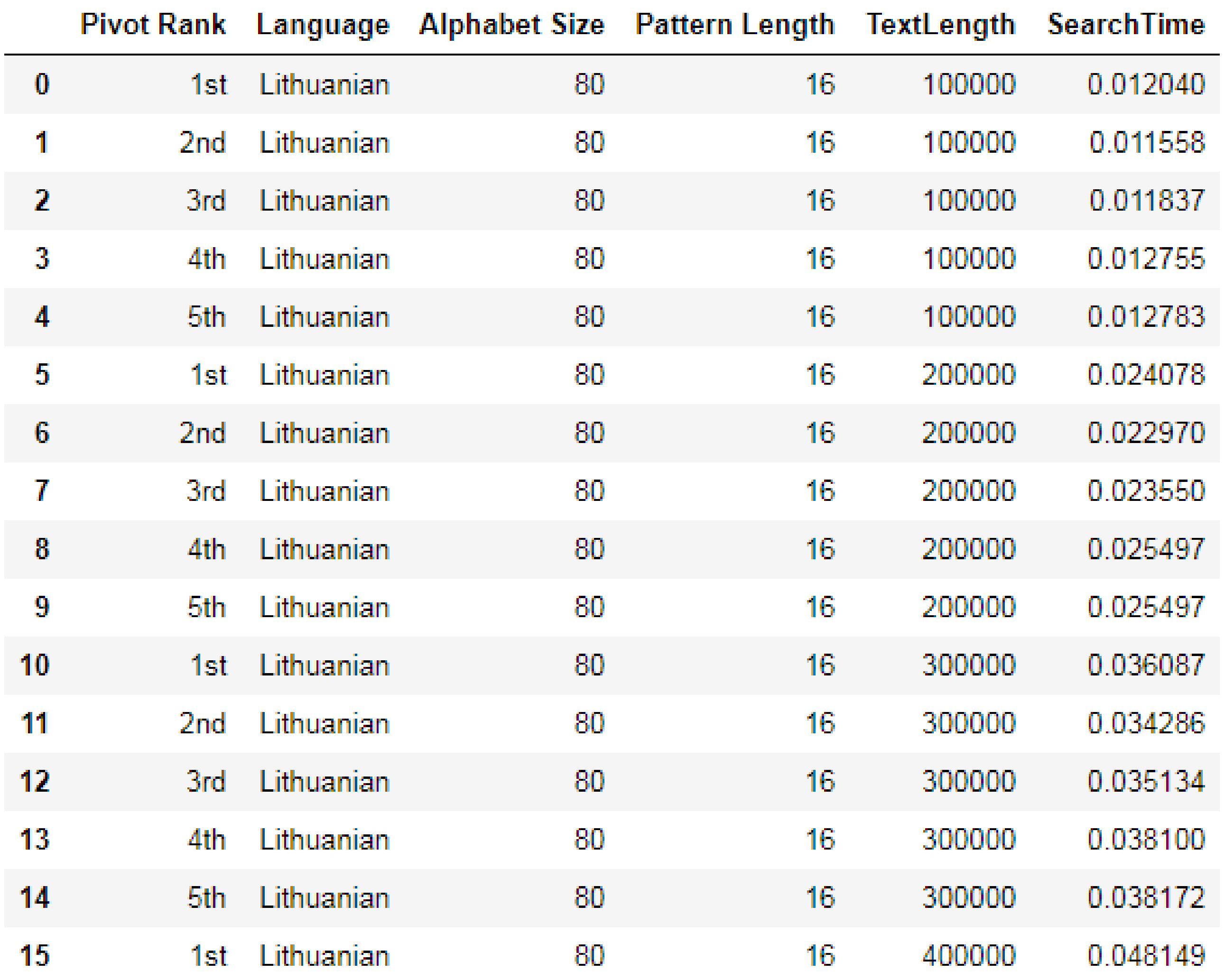
	Pivot Rank	Language	Alphabet Size	Pattern Length	TextLength	SearchTime
0	1st	Lithuanian	80	2	100000	0.025900
1	2nd	Lithuanian	80	2	100000	0.023831
2	3rd	Lithuanian	80	2	100000	0.022015
3	4th	Lithuanian	80	2	100000	0.021126
4	5th	Lithuanian	80	2	100000	0.020085
5	1st	Lithuanian	80	2	200000	0.051005
6	2nd	Lithuanian	80	2	200000	0.046724
7	3rd	Lithuanian	80	2	200000	0.043115
8	4th	Lithuanian	80	2	200000	0.041447
9	5th	Lithuanian	80	2	200000	0.041447
10	1st	Lithuanian	80	2	300000	0.076003
11	2nd	Lithuanian	80	2	300000	0.069443
12	3rd	Lithuanian	80	2	300000	0.064007
13	4th	Lithuanian	80	2	300000	0.061550
14	5th	Lithuanian	80	2	300000	0.058414
15	1st	Lithuanian	80	2	400000	0.100975

E Search Time Table for Patterns of length 8 in Lithuanian



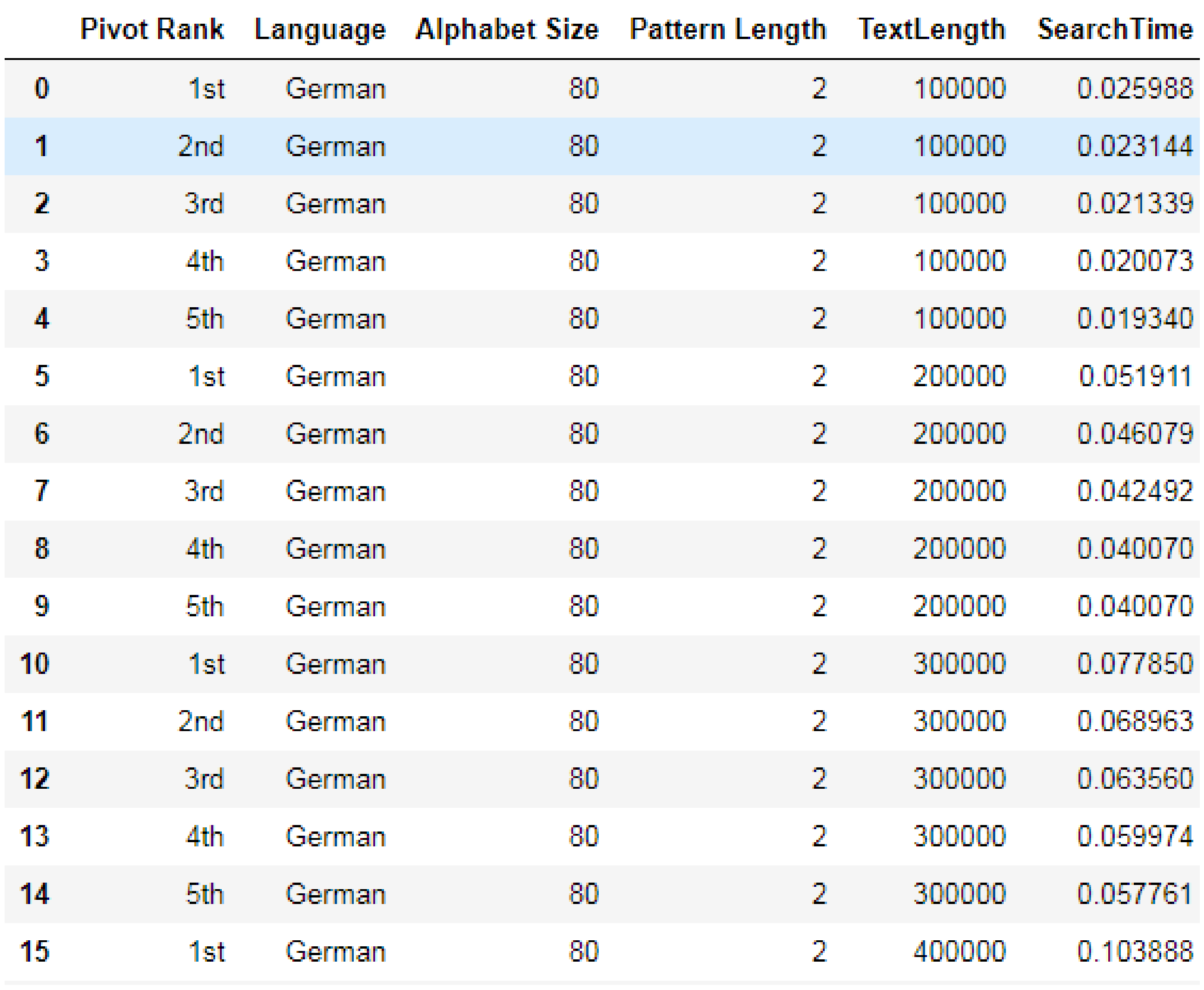
	Pivot Rank	Language	Alphabet Size	Pattern Length	TextLength	SearchTime
0	1st	Lithuanian	80	8	100000	0.013283
1	2nd	Lithuanian	80	8	100000	0.014424
2	3rd	Lithuanian	80	8	100000	0.014537
3	4th	Lithuanian	80	8	100000	0.015256
4	5th	Lithuanian	80	8	100000	0.015030
5	1st	Lithuanian	80	8	200000	0.025932
6	2nd	Lithuanian	80	8	200000	0.028070
7	3rd	Lithuanian	80	8	200000	0.028320
8	4th	Lithuanian	80	8	200000	0.029867
9	5th	Lithuanian	80	8	200000	0.029867
10	1st	Lithuanian	80	8	300000	0.038550
11	2nd	Lithuanian	80	8	300000	0.041619
12	3rd	Lithuanian	80	8	300000	0.041970
13	4th	Lithuanian	80	8	300000	0.044338
14	5th	Lithuanian	80	8	300000	0.043649
15	1st	Lithuanian	80	8	400000	0.051219

F Search Time Table for Patterns of length 16 in Lithuanian



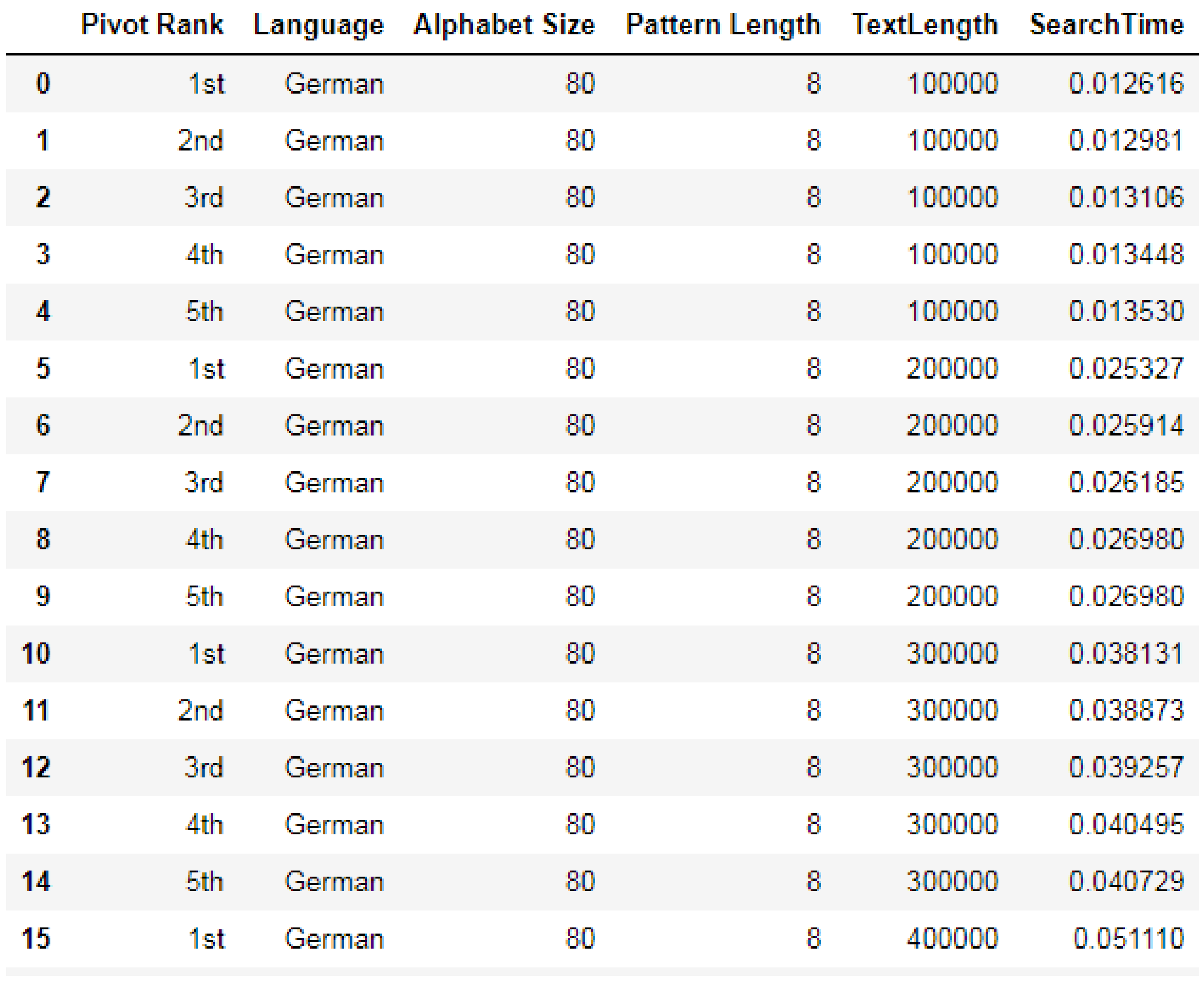
	Pivot Rank	Language	Alphabet Size	Pattern Length	TextLength	SearchTime
0	1st	Lithuanian	80	16	100000	0.012040
1	2nd	Lithuanian	80	16	100000	0.011558
2	3rd	Lithuanian	80	16	100000	0.011837
3	4th	Lithuanian	80	16	100000	0.012755
4	5th	Lithuanian	80	16	100000	0.012783
5	1st	Lithuanian	80	16	200000	0.024078
6	2nd	Lithuanian	80	16	200000	0.022970
7	3rd	Lithuanian	80	16	200000	0.023550
8	4th	Lithuanian	80	16	200000	0.025497
9	5th	Lithuanian	80	16	200000	0.025497
10	1st	Lithuanian	80	16	300000	0.036087
11	2nd	Lithuanian	80	16	300000	0.034286
12	3rd	Lithuanian	80	16	300000	0.035134
13	4th	Lithuanian	80	16	300000	0.038100
14	5th	Lithuanian	80	16	300000	0.038172
15	1st	Lithuanian	80	16	400000	0.048149

G Search Time Table for Patterns of length 2 in German



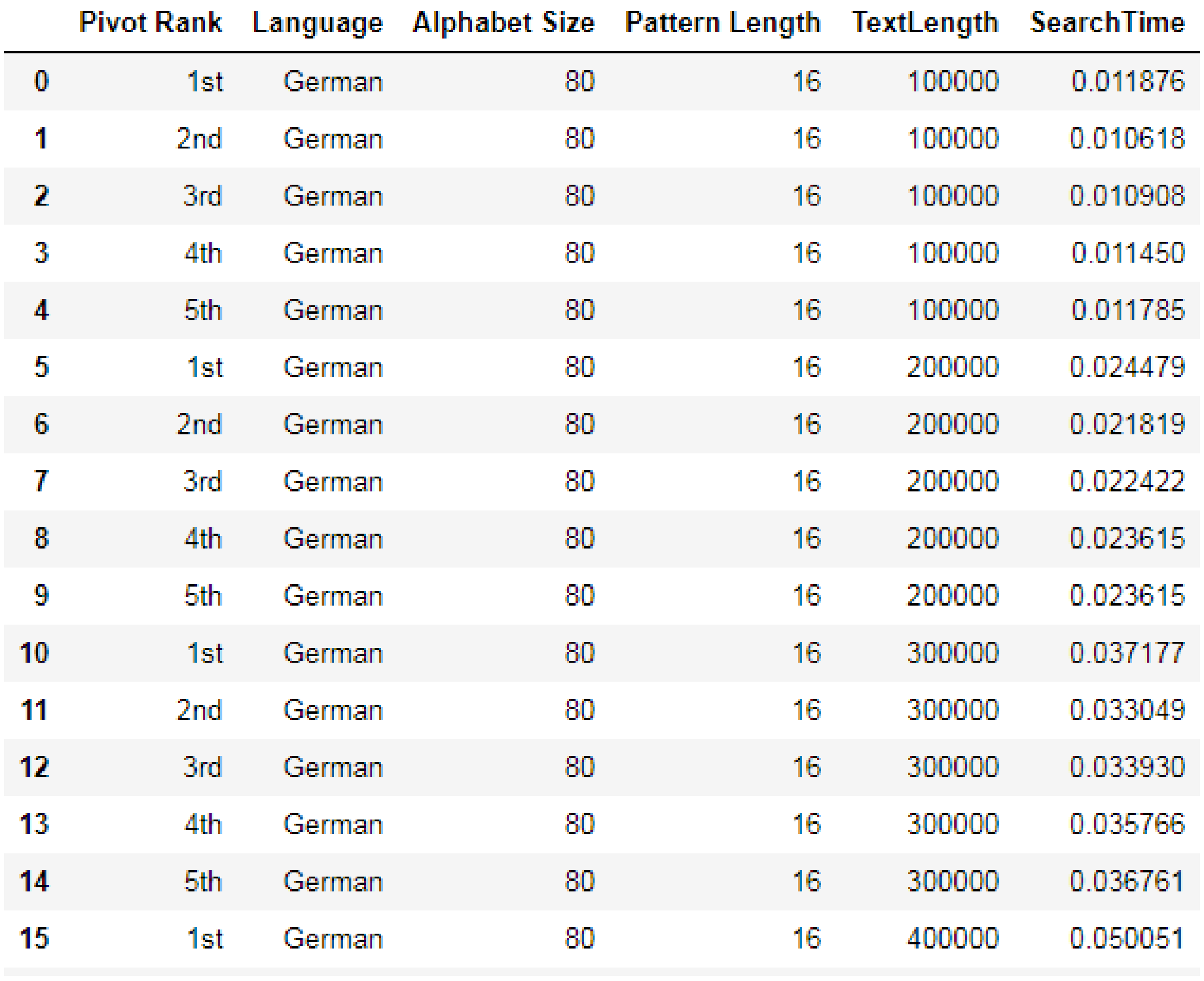
	Pivot Rank	Language	Alphabet Size	Pattern Length	TextLength	SearchTime
0	1st	German	80	2	100000	0.025988
1	2nd	German	80	2	100000	0.023144
2	3rd	German	80	2	100000	0.021339
3	4th	German	80	2	100000	0.020073
4	5th	German	80	2	100000	0.019340
5	1st	German	80	2	200000	0.051911
6	2nd	German	80	2	200000	0.046079
7	3rd	German	80	2	200000	0.042492
8	4th	German	80	2	200000	0.040070
9	5th	German	80	2	200000	0.040070
10	1st	German	80	2	300000	0.077850
11	2nd	German	80	2	300000	0.068963
12	3rd	German	80	2	300000	0.063560
13	4th	German	80	2	300000	0.059974
14	5th	German	80	2	300000	0.057761
15	1st	German	80	2	400000	0.103888

H Search Time Table for Patterns of length 8 in German



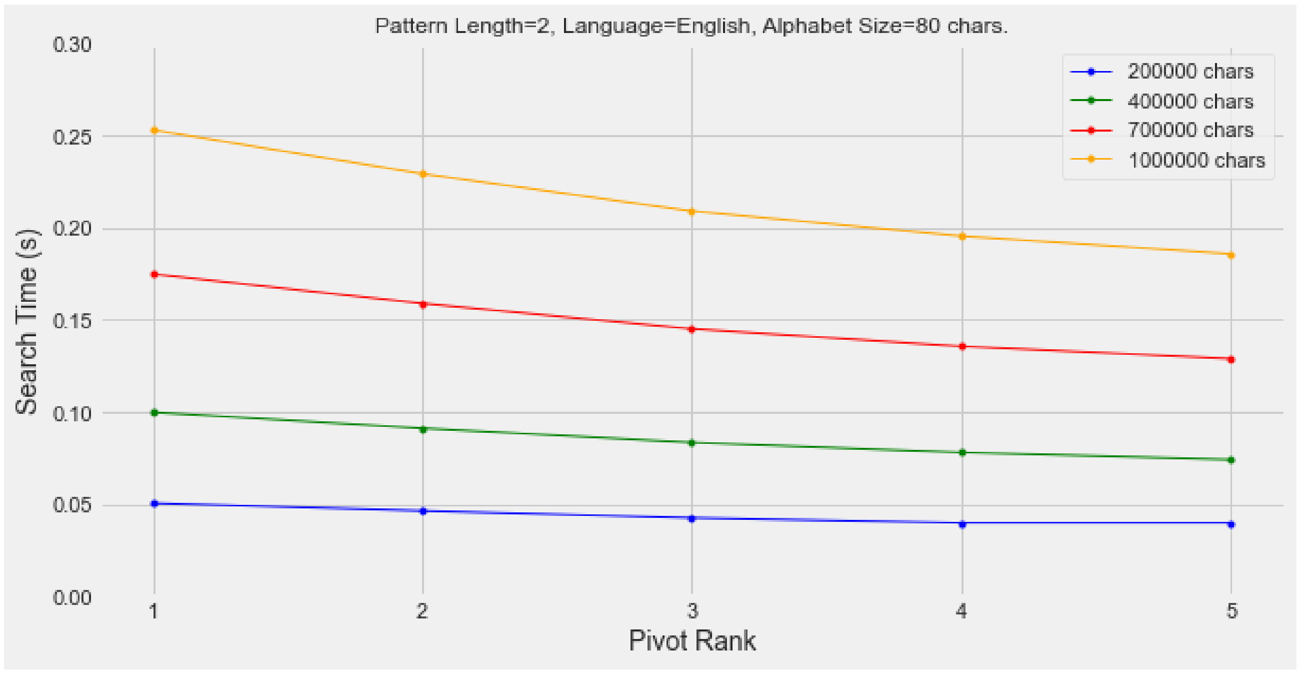
	Pivot Rank	Language	Alphabet Size	Pattern Length	TextLength	SearchTime
0	1st	German	80	8	100000	0.012616
1	2nd	German	80	8	100000	0.012981
2	3rd	German	80	8	100000	0.013106
3	4th	German	80	8	100000	0.013448
4	5th	German	80	8	100000	0.013530
5	1st	German	80	8	200000	0.025327
6	2nd	German	80	8	200000	0.025914
7	3rd	German	80	8	200000	0.026185
8	4th	German	80	8	200000	0.026980
9	5th	German	80	8	200000	0.026980
10	1st	German	80	8	300000	0.038131
11	2nd	German	80	8	300000	0.038873
12	3rd	German	80	8	300000	0.039257
13	4th	German	80	8	300000	0.040495
14	5th	German	80	8	300000	0.040729
15	1st	German	80	8	400000	0.051110

I Search Time Table for Patterns of length 16 in German

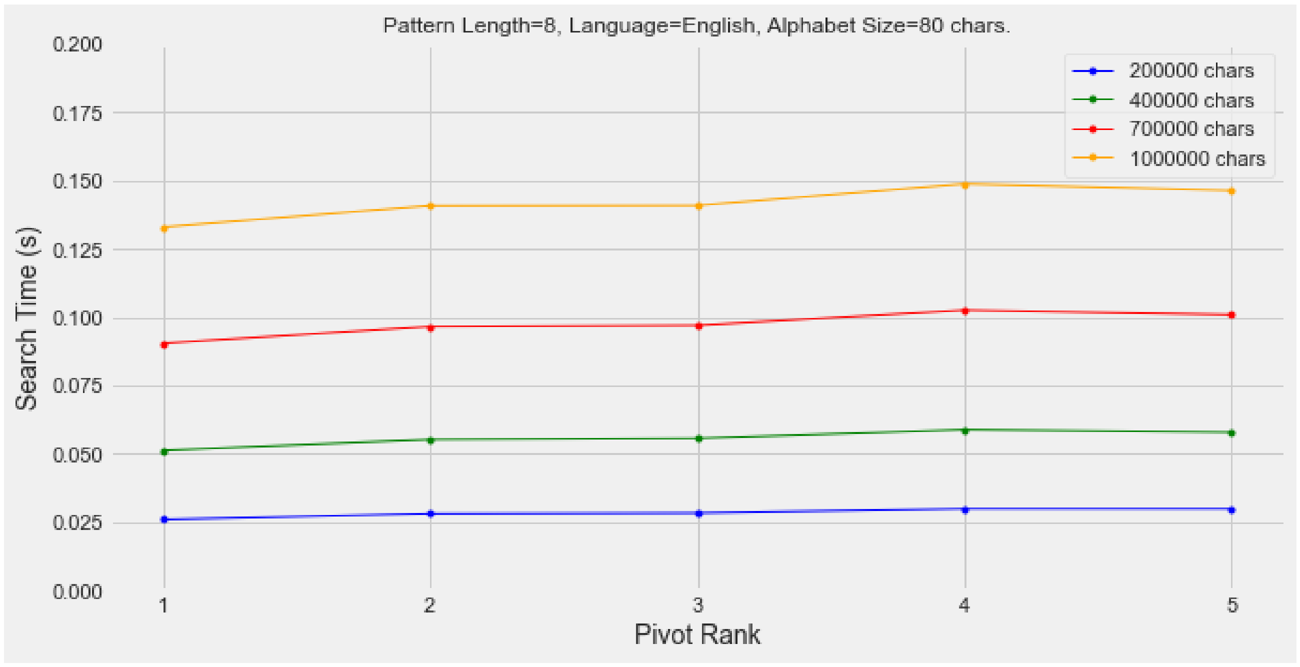


	Pivot Rank	Language	Alphabet Size	Pattern Length	TextLength	SearchTime
0	1st	German	80	16	100000	0.011876
1	2nd	German	80	16	100000	0.010618
2	3rd	German	80	16	100000	0.010908
3	4th	German	80	16	100000	0.011450
4	5th	German	80	16	100000	0.011785
5	1st	German	80	16	200000	0.024479
6	2nd	German	80	16	200000	0.021819
7	3rd	German	80	16	200000	0.022422
8	4th	German	80	16	200000	0.023615
9	5th	German	80	16	200000	0.023615
10	1st	German	80	16	300000	0.037177
11	2nd	German	80	16	300000	0.033049
12	3rd	German	80	16	300000	0.033930
13	4th	German	80	16	300000	0.035766
14	5th	German	80	16	300000	0.036761
15	1st	German	80	16	400000	0.050051

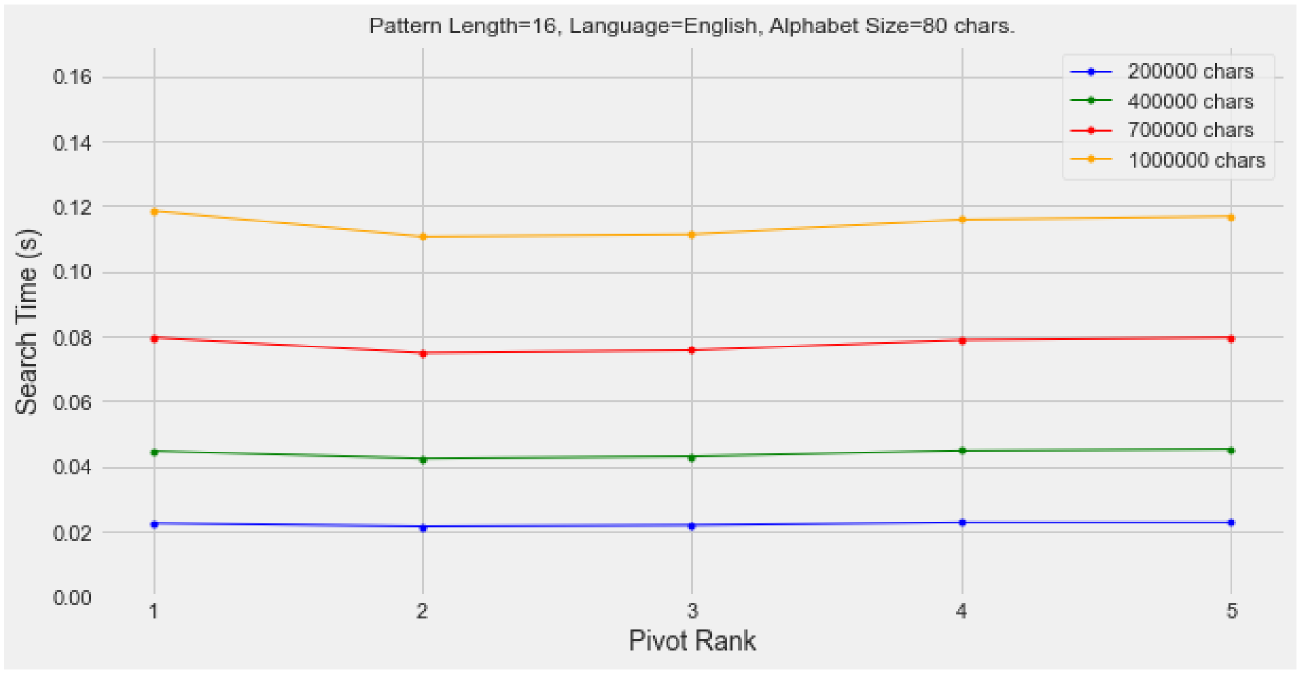
J Plot for Patterns of length 2 in English



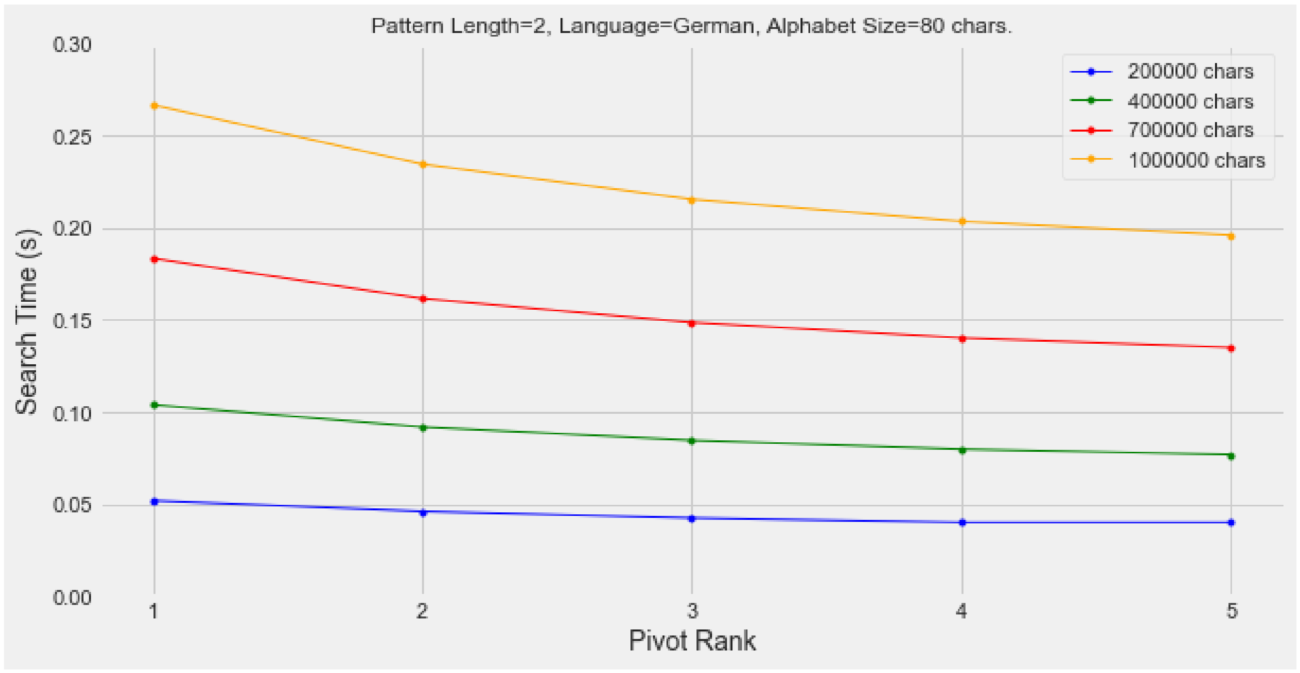
K Plot for Patterns of length 8 in English



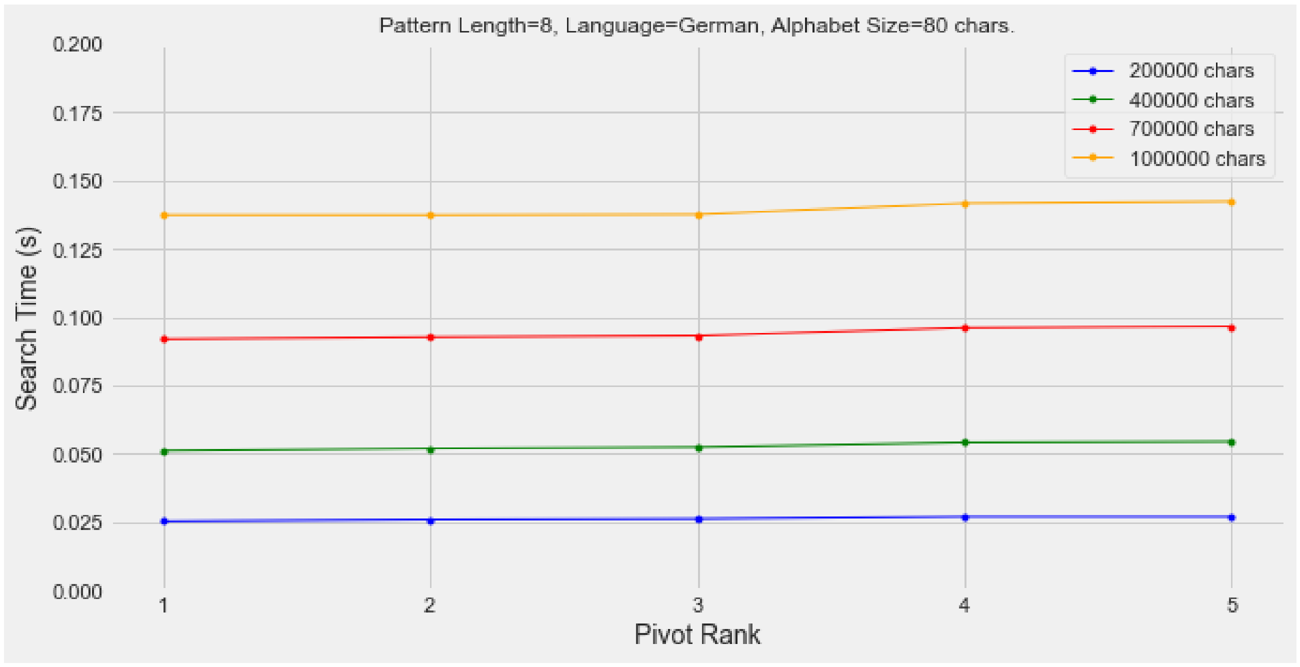
L Plot for Patterns of length 16 in English



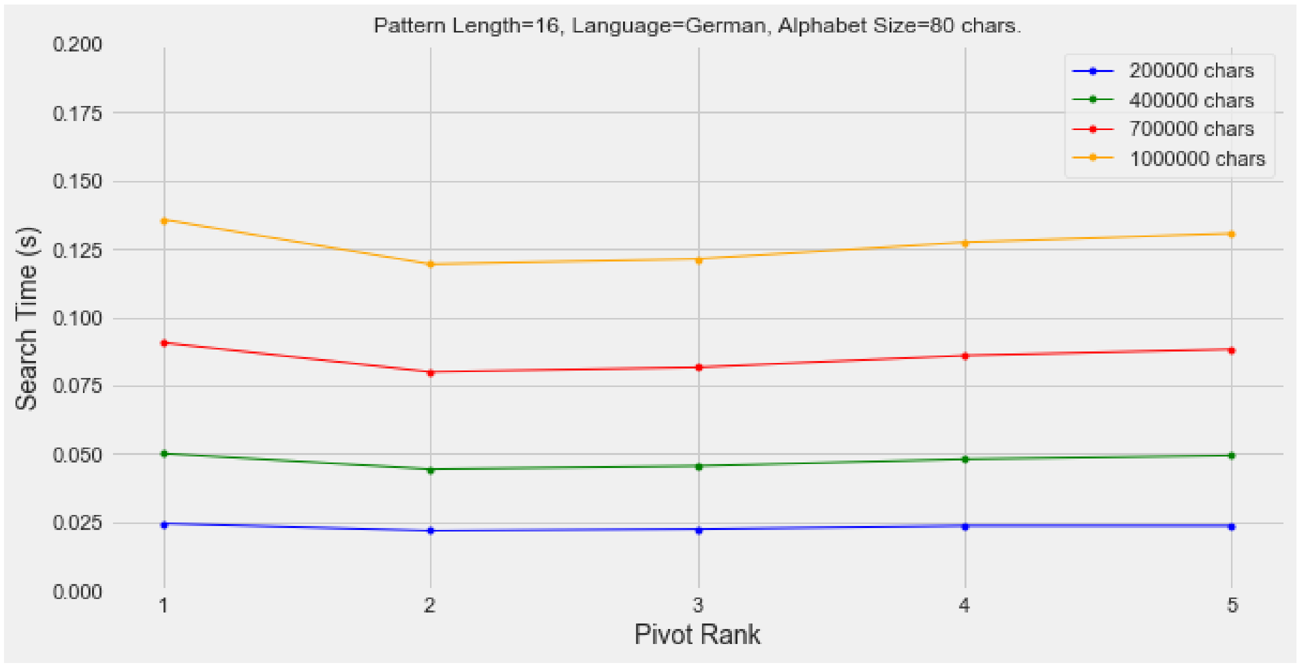
M Plot for Patterns of length 2 in German



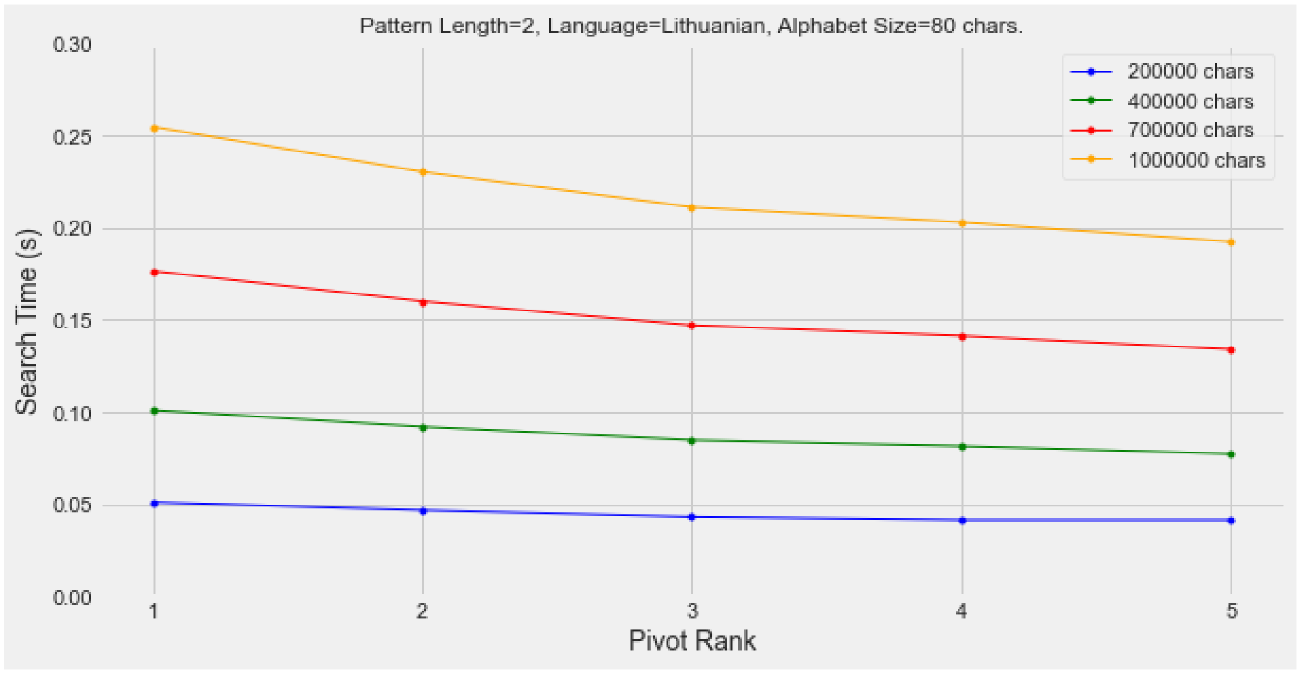
N Plot for Patterns of length 8 in German



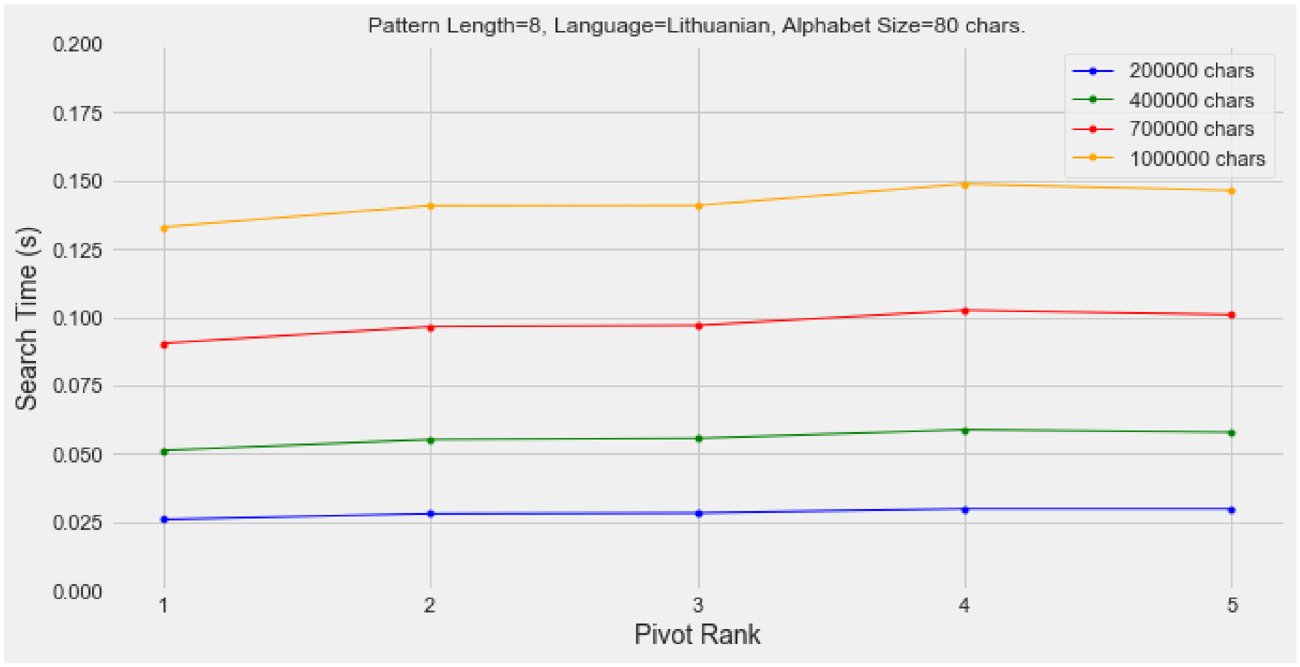
O Plot for Patterns of length 16 in German



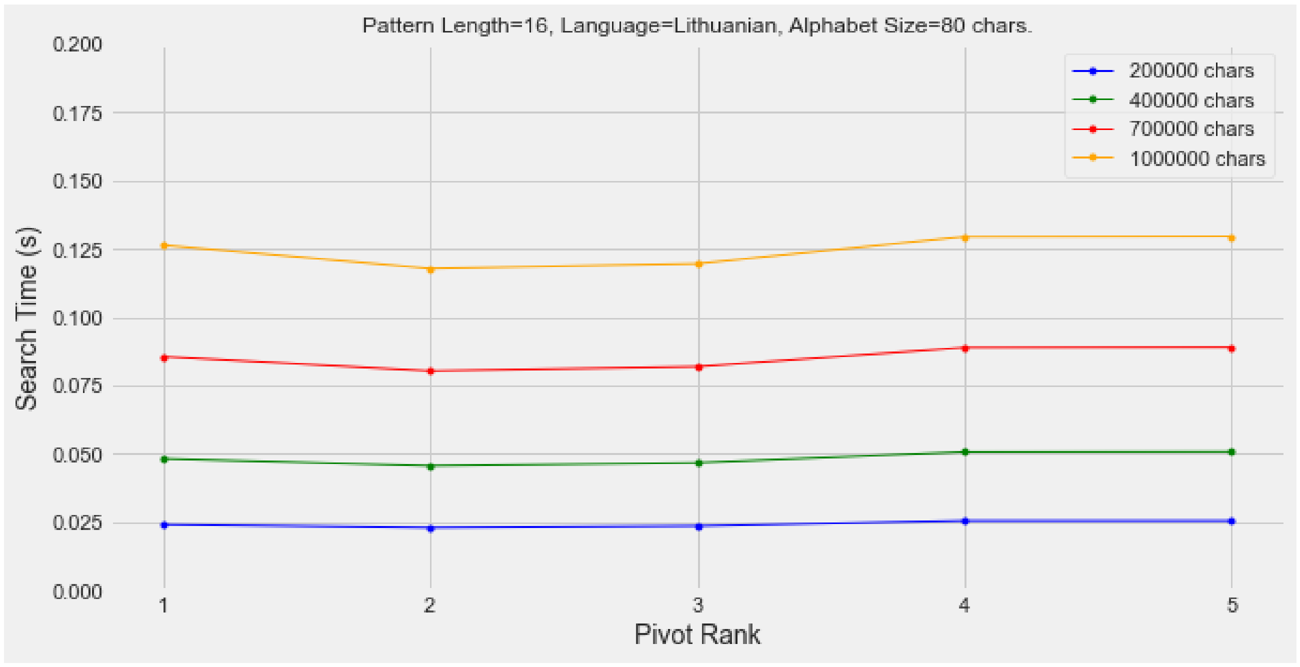
P Plot for Patterns of length 2 in Lithuanian



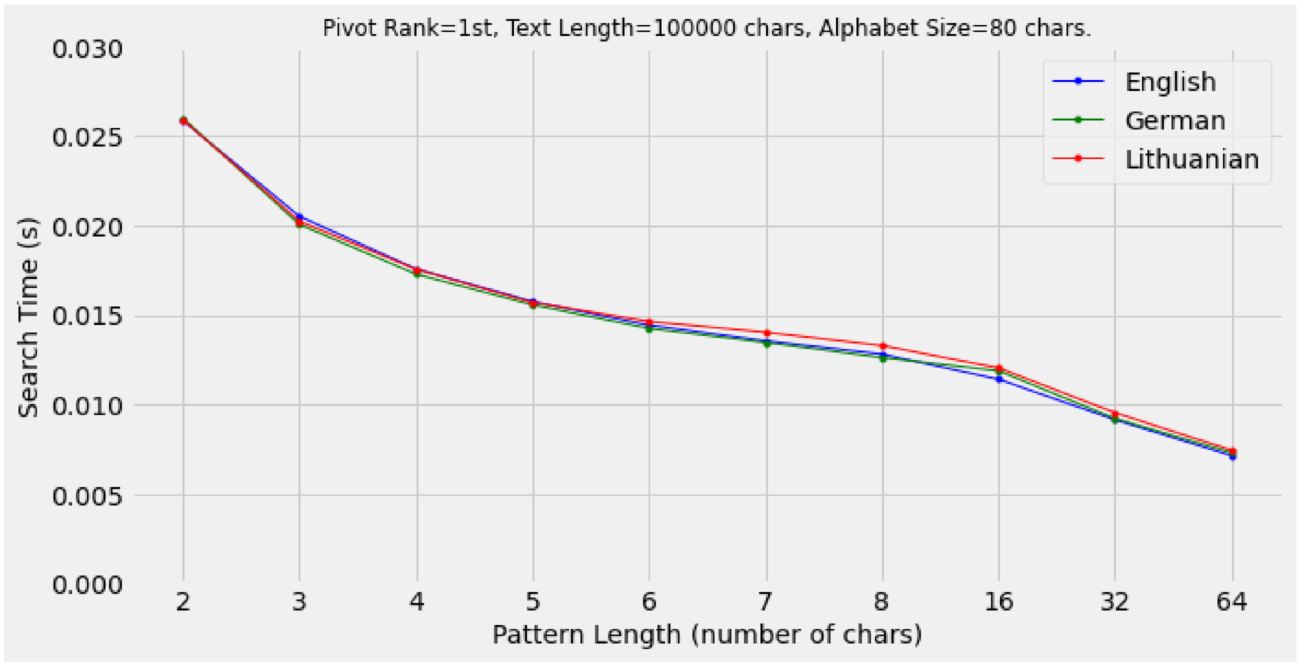
Q Plot for Patterns of length 8 in Lithuanian



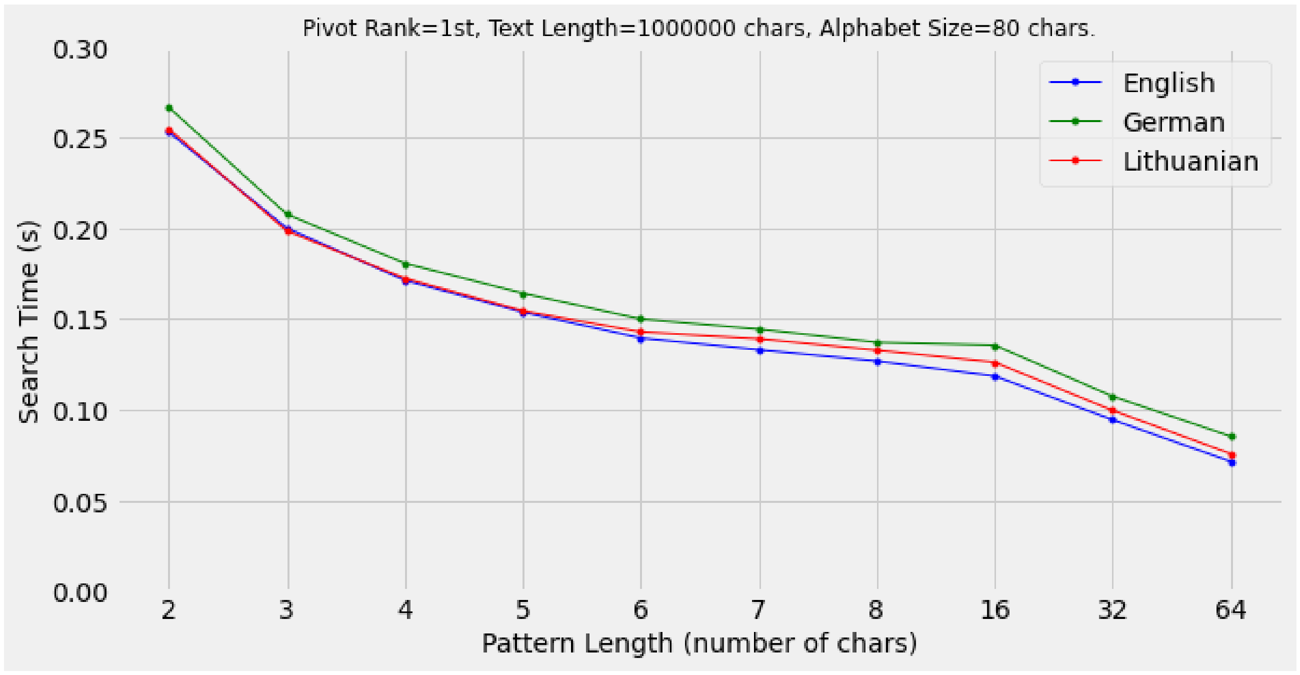
R Plot for Patterns of length 16 in Lithuanian



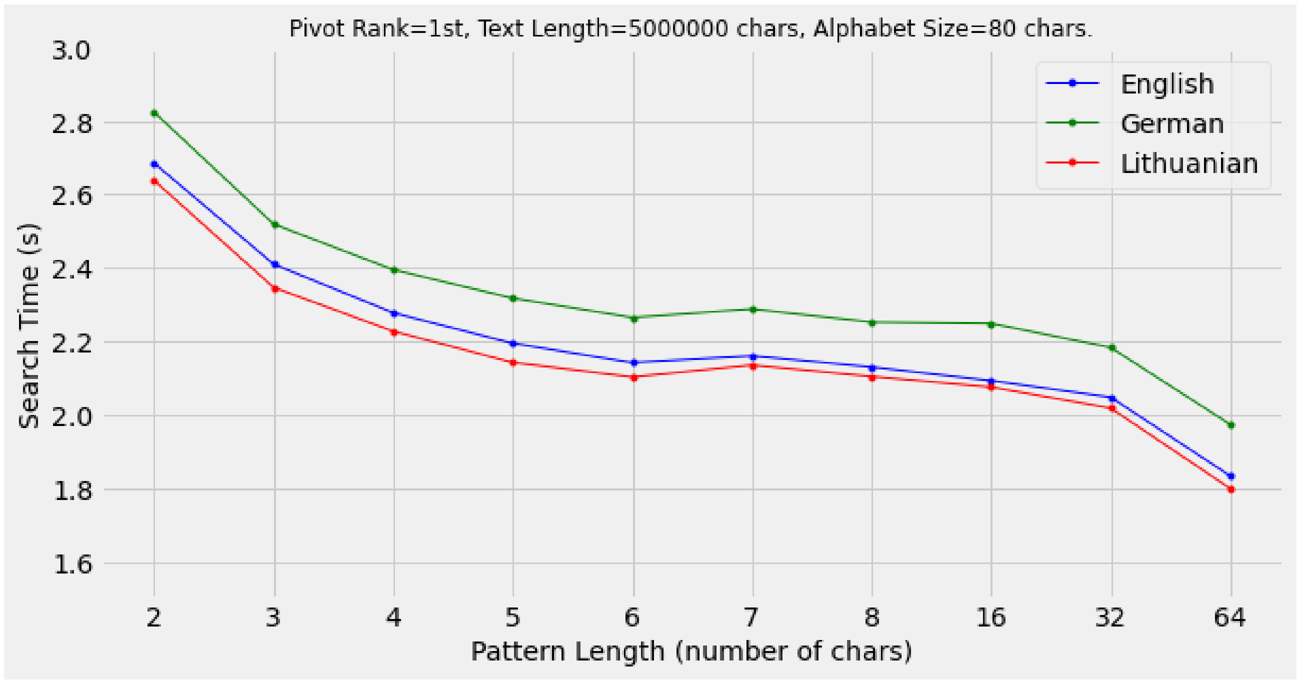
S Plot for Patterns of different lengths, 100k of chars



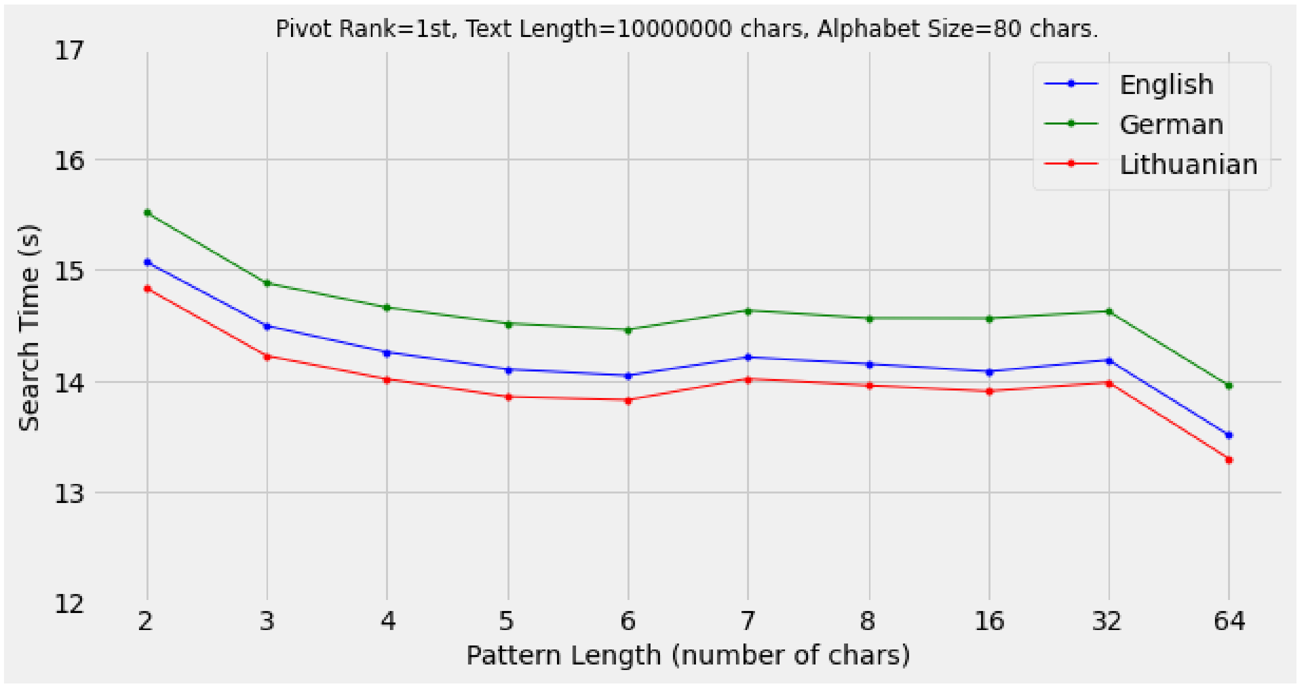
T Plot for Patterns of different lengths, 1m of chars



U Plot for Patterns of different lengths, 5m of chars



V Plot for Patterns of different lengths, 10m of chars

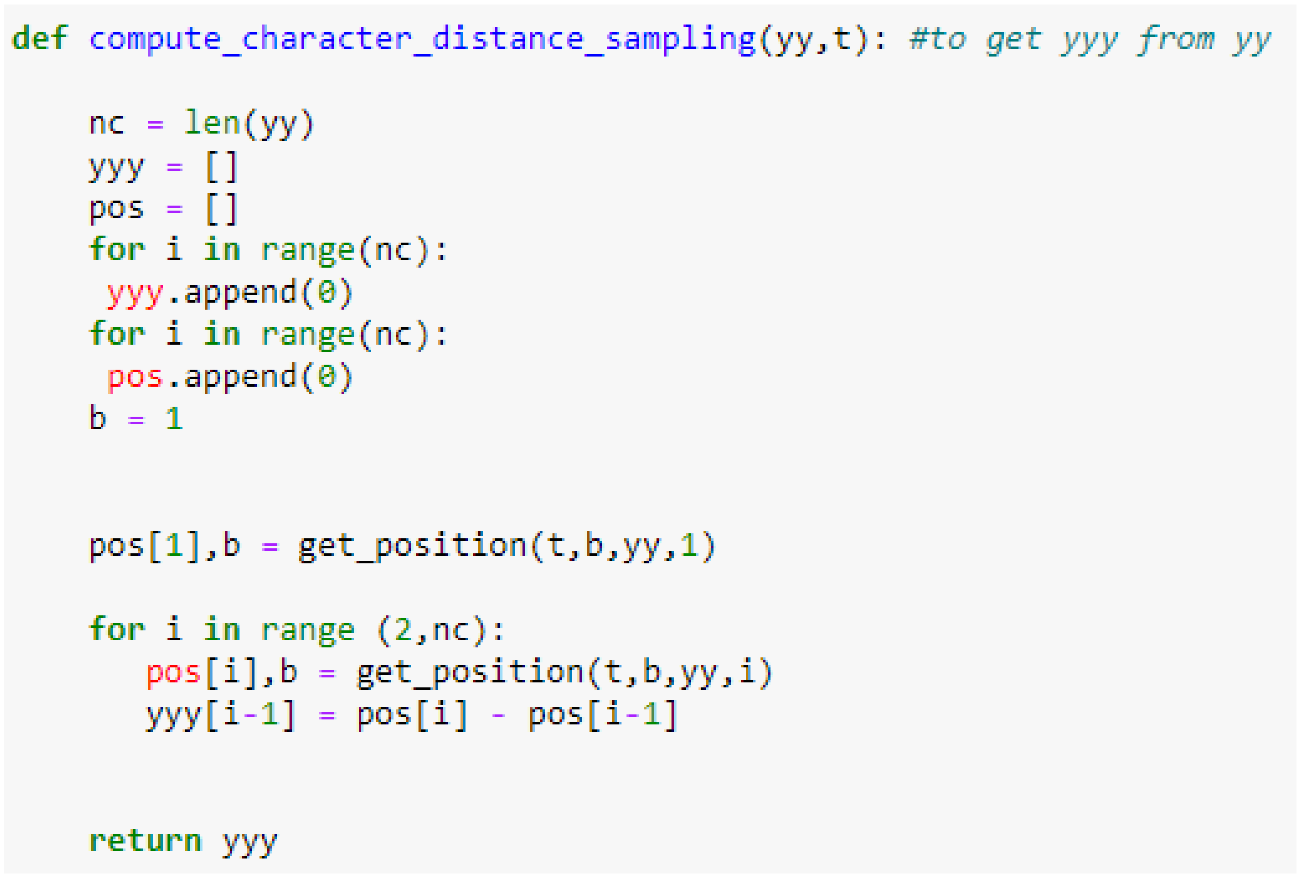


W Auxiliary Functions

```python
def get_position(t,b,yy,i): # get original position in y

    while t[b] < i:
        b = b+1

    p = ((b-1) * k) + yy[i]  # restore the original position using k instead

    return p, b
```

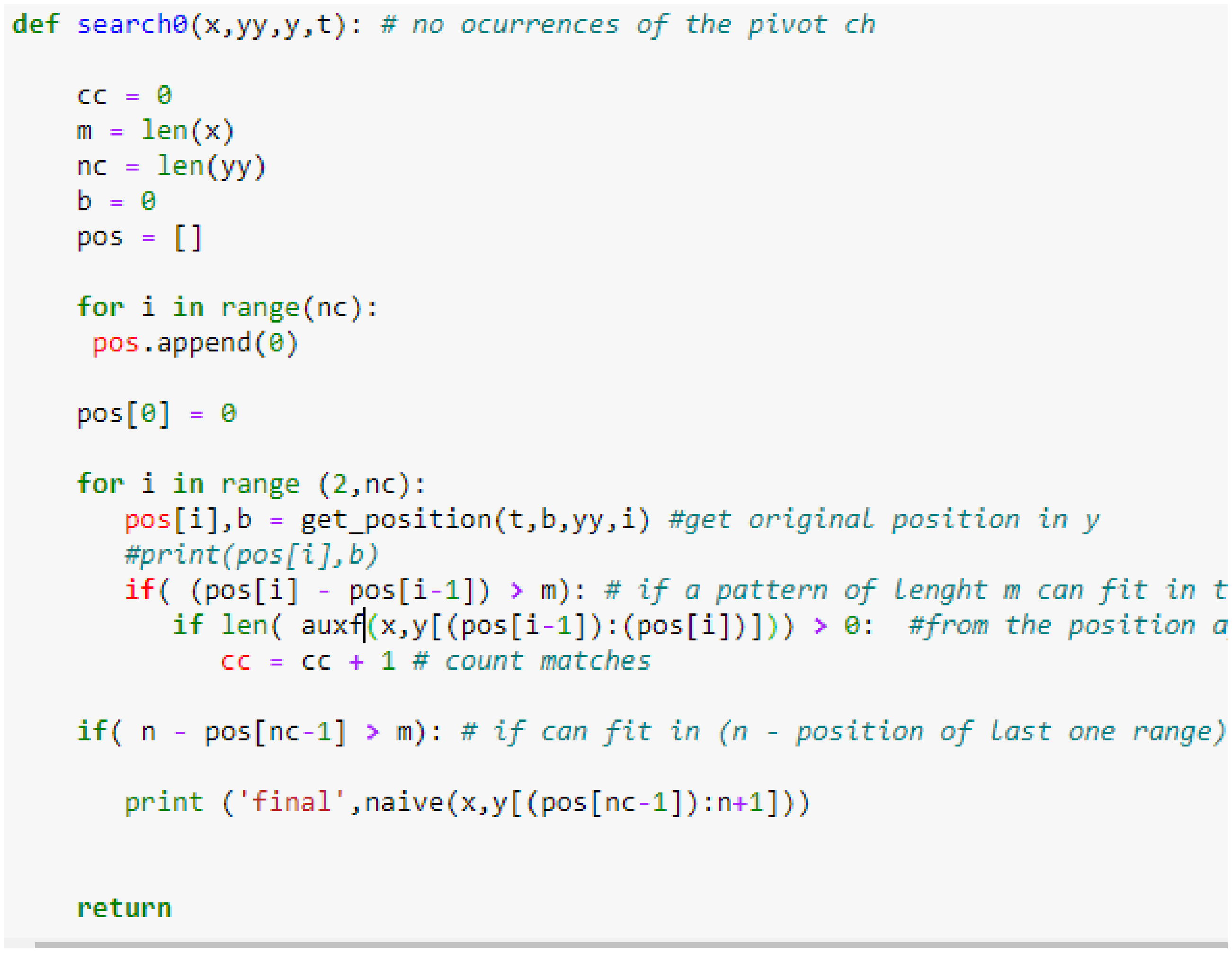
```python
def compute_distance_sampling(y,n,alpha): #used for sampling the pattern

    yyy = []
    j = 0
    p = 0

    for i in range(1,n):
       if y[i] in alpha:
         j= j+1
         yyy.append(i - p)
         p = i



    return yyy
```

X Compute Character Distance Function



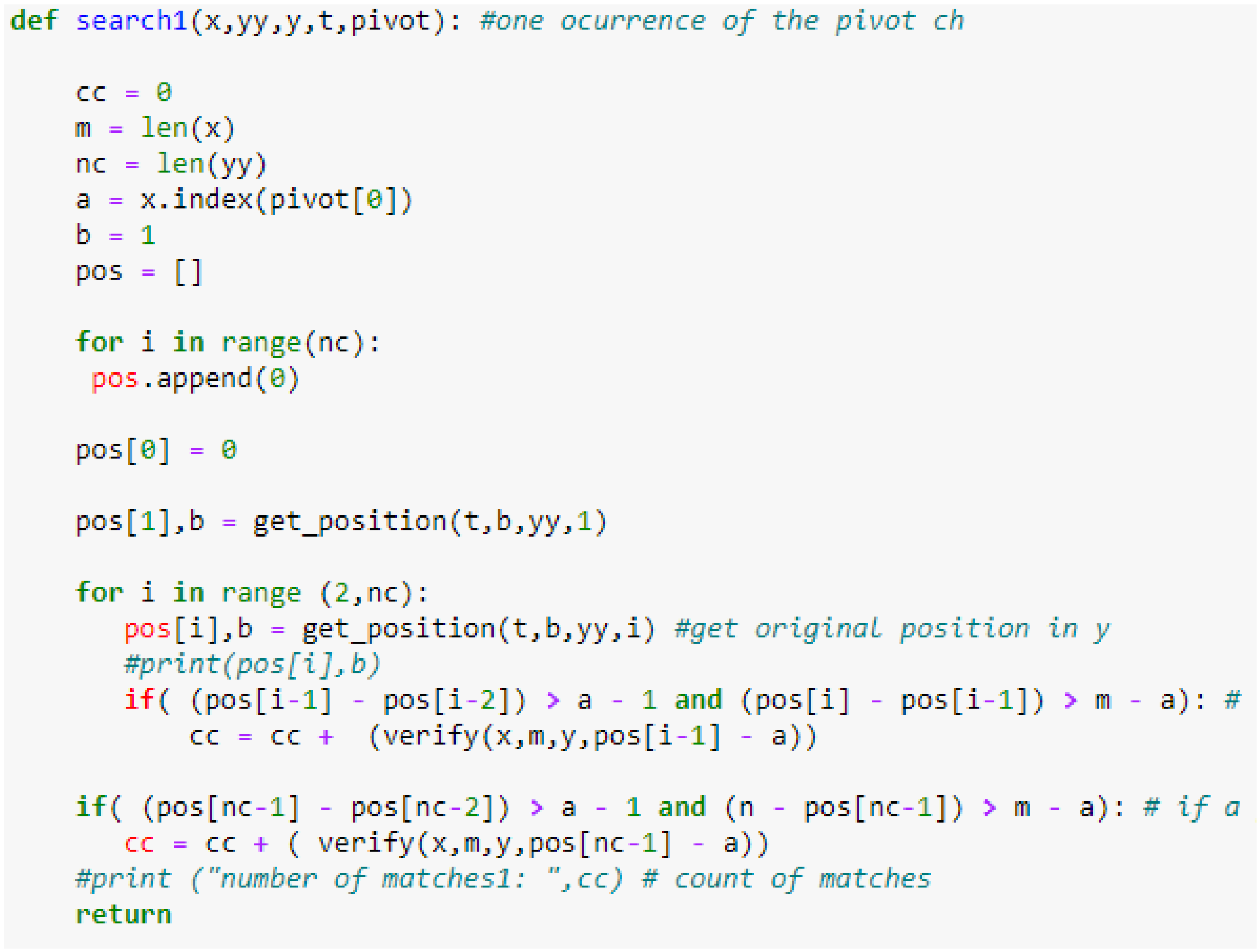
```python
def compute_character_distance_sampling(yy,t): #to get yyy from yy

    nc = len(yy)
    yyy = []
    pos = []
    for i in range(nc):
     yyy.append(0)
    for i in range(nc):
     pos.append(0)
    b = 1


    pos[1],b = get_position(t,b,yy,1)

    for i in range (2,nc):
       pos[i],b = get_position(t,b,yy,i)
       yyy[i-1] = pos[i] - pos[i-1]


    return yyy
```

Y Search0 Function



```
def search0(x,yy,y,t): # no ocurrences of the pivot ch

    cc = 0
    m = len(x)
    nc = len(yy)
    b = 0
    pos = []

    for i in range(nc):
     pos.append(0)

    pos[0] = 0

    for i in range (2,nc):
       pos[i],b = get_position(t,b,yy,i) #get original position in y
       #print(pos[i],b)
       if( (pos[i] - pos[i-1]) > m): # if a pattern of lenght m can fit in t
          if len( auxf(x,y[(pos[i-1]):(pos[i])])) > 0:  #from the position a
             cc = cc + 1 # count matches

    if( n - pos[nc-1] > m): # if can fit in (n - position of last one range)

       print ('final',naive(x,y[(pos[nc-1]):n+1]))


    return
```

Z Search1 Function



```
def search1(x,yy,y,t,pivot): #one ocurrence of the pivot ch

    cc = 0
    m = len(x)
    nc = len(yy)
    a = x.index(pivot[0])
    b = 1
    pos = []

    for i in range(nc):
     pos.append(0)

    pos[0] = 0

    pos[1],b = get_position(t,b,yy,1)

    for i in range (2,nc):
       pos[i],b = get_position(t,b,yy,i) #get original position in y
       #print(pos[i],b)
       if( (pos[i-1] - pos[i-2]) > a - 1 and (pos[i] - pos[i-1]) > m - a): #
           cc = cc +  (verify(x,m,y,pos[i-1] - a))

    if( (pos[nc-1] - pos[nc-2]) > a - 1 and (n - pos[nc-1]) > m - a): # if a
       cc = cc + ( verify(x,m,y,pos[nc-1] - a))
    #print ("number of matches1: ",cc) # count of matches
    return
```