Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Approximate search for textual information in images of historical manuscripts using simple regular expression queries

DEGREE FINAL WORK

Degree in Computer Engineering

Author: José Andrés Moreno
Tutor: Enrique Vidal Ruiz
Experimental Director: Alejandro Hector Toselli

Course 2019-2020

Acknowledgements

I wanted to thank Enrique, for trusting in me and giving me the opportunity of developing this project at the PRHLT. Also, I wanted to thank Alejandro, for helping me to go through the most technical parts of this project.

Finally, I wanted to thank all the people who have helped me get to this point.

« Mais non jamais, jamais je ne t'oublierai » - Edouard Priem

Resumen

Los archivos históricos así como otras instituciones de patrimonio cultural han estado digitalizando sus colecciones de documentos históricos con el fin de hacerlas accesible a través de Internet al público en general. Sin embargo, la mayor parte de las imágenes de los documentos digitalizados carecen de transcripción, por lo que el acceso a su contenido textual no es posible.

En los últimos años, en el Centro PRHLT de la UPV se ha desarrollado una tecnología para la indexación probabilística de colecciones de estas imágenes (no transcritas). La principal aplicación de estos índices es facilitar la búsqueda de información textual en la colección de imágenes. El sistema de indexación desarrollado genera una tabla en la cual se indexa cada palabra con todas sus posibles localizaciones en el documento. Específicamente, cada entrada de la tabla define una palabra con información de su localización: número de página y posición en la página, y una medida de certeza (o confianza) calculada a partir de la probabilidad de aparición de dicha palabra en esa localización de la imagen.

La disponibilidad de sistemas como este abre un nuevo horizonte en el marco de las humanidades y en particular en el estudio de la historia. No obstante, para una mayor flexibilidad en estas aplicaciones es necesario dotar a los sistemas de búsqueda de capacidades similares a las de los buscadores tradicionales. En particular, se ha permitido a los usuarios formular sus consultas mediante expresiones regulares simples, así como búsquedas aproximadas; es decir, palabras similares a las consultadas. Por ejemplo, para buscar la palabra "France", ejemplos de consultas basadas en expresión regular podrían ser $''Fran.*''$ o $''.*ranc.*''$. Así mismo, para una búsqueda aproximada se podría formular alguna las siguientes consultas: $''Francia''$, $''france''$, $''franc''$, etc.

Durante el proyecto se han explorado diferentes técnicas para realizar búsquedas aproximadas y finalmente se han obtenido resultados favorables tanto a nivel de tiempo como a nivel de consumo de memoria. De esta forma, podemos concluir que se ha logrado ampliar la funcionalidad del sistema con un consumo de recursos moderado.

Palabras clave: Procesado de Imágenes de Texto Manuscrito; Indexación Probabilística de Palabras; Búsqueda y Recuperación de la Información; Expresiones Regulares; Distancia de Edición; Algoritmos Rápidos de Búsqueda

Abstract

Historical archives, as well as other cultural heritage institutes, have been digitizing their collections of historical documents in order to make them accessible via the Internet to the general public. However, most of the images in the digitized documents lack of transcription, so access to their textual content is not possible.

In recent years, technology has been developed at the UPV, in the PRHLT Center for the probabilistic indexing of collections of these images (not transcribed). The main application of these indexes is to facilitate the search for textual information in the image collection. The developed indexing system generates a table in which each word is indexed with all its possible locations in the document. Specifically, each table entry defines a word with information on its location: page number and position on the page, and a measure of certainty (or confidence) calculated from the probability of the appearance of said word in that location of the image.

The availability of systems like this opens a new horizon in the humanities framework and in particular in the study of history. However, for greater flexibility in these applications, it is necessary to provide search systems with capabilities similar to those of traditional search engines. In particular, there is a need to allow users to formulate their queries using simple regular expressions, as well as approximate searches; that is, words similar to those consulted. For example, to search for the word "France", examples of regular expression-based queries might be $"Fran.*"$, or $".*ran.*"$. Likewise, for an approximate search, the following queries could be made: $"Francia"$, $"france"$, $"franc"$, etc.

During this project several techniques to perform approximate search have been explored and finally, we have achieved good results in terms of time performance with reasonable memory consumption. Therefore, we can conclude that we have improved the flexibility of the system with moderate memory usage.

Key words: Handwritten Text Image Processing, Probabilistic Word Indexing, Information Search and Retrieval, Regular Expressions, Edit Distance, Fast Search Algorithms

Contents

Contents **vii**
List of Figures **ix**
List of Tables **x**
List of algorithms **x**

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 1
1.3 Thesis structure 2
2 Description of the initial system **5**
3 Wildcard search **7**
3.1 Online techniques 7
3.1.1 Regular expressions 7
3.2 Offline techniques 8
3.2.1 MARISA trie 8
4 Approximate string search **11**
4.1 Online techniques 12
4.1.1 The dynamic programming algorithm 12
4.1.2 Regular expressions 13
4.2 Offline techniques 13
4.2.1 BK-Tree 13
4.2.2 DAWG levenshtein 16
5 Implementation and demonstrators **19**
5.1 Implementation 19
5.2 Bentham demonstrator 20
6 Experimental results **25**
6.1 Experiments on Bentham GT 26
6.1.1 Wildcard search 26
6.1.2 Approximate string search 30
6.2 Experiments on Bentham 35
6.2.1 Wildcard search 35
6.2.2 Approximate string search 37
7 Conclusions **39**
7.1 Future work 39
7.2 Degree relationship 40
7.3 Transversal competences 40
Bibliography **41**

List of Figures

2.1 Workflow diagram of the system. 5
2.2 Workflow diagram of the KWS and indexing tool phase. 6
2.3 Workflow diagram of the index ingestion phase. 6
2.4 Workflow diagram of the keyword search phase. 6

3.1 Automaton recognising the strings that start with the prefix "OFFEN". . . 7
3.2 Trie storing the words: "PAUL", "MARIA", "MARTHA", "MOTHER". . . . 8
3.3 Patricia trie storing the words: "PAUL", "MARIA", "MARTHA", "MOTHER". 9
3.4 Trie used as suffix tree storing the words: "PAUL", "MARIA", "MARTHA", "MOTHER". 9

4.1 Automaton accepting strings with at most 2 edit operations from "OFFEN". 11
4.2 BK-Tree storing the words: "CAT", "CUT", "MAP", "HOT", "CATS" 14
4.3 The strings "TOP", "TOPS", "TAP" and "TAPS" stored in a trie (left) or in a DAWG (right). 16

5.1 Workflow diagram of wildcard query "offen*". 19
5.2 Workflow diagram of approximate string query "committted$\sim$1". 19
5.3 Screenshot of the HOME level of the Bentham demonstrator 21
5.4 Result of the search "offen*" at HOME level 21
5.5 Result of the search "offen*" at BOX level 22
5.6 Matched spot for query "offen*" on page 14. 22
5.7 Pseudowords that matched the spot on page 14 for query "offen*". 23
5.8 Pseudowords that matched the spot on page 24 for query "offen*". 24
5.9 Pseudowords that matched the spot on page 3 for query "committed 1". . 24

6.1 Number of pseudowords in Bentham GT according to their length. 26
6.2 Elapsed time according to prefix length. 27
6.3 Elapsed time according to suffix length. 27
6.4 Number of pseudowords retrieved according to prefix length. 28
6.5 Number of pseudowords retrieved according to suffix length. 28
6.6 Elapsed time according to prefix length. 29
6.7 Elapsed time according to suffix length. 29
6.8 Elapsed time according to pseudoword length allowing one error 30
6.9 Number of pseudowords retrieved according to query length. 31
6.10 Elapsed time on performing a query according to query length. 32
6.11 Frequency of the number of pseudowords that overlap 33
6.12 Number of unique pseudowords matched according to query length . . . 34
6.13 Number of pseudowords in Bentham according to their length. 35
6.14 Elapsed time according to prefix length. 36
6.15 Elapsed time according to suffix length. 36
6.16 Number of pseudowords retrieved according to prefix length. 37
6.17 Number of pseudowords retrieved according to suffix length. 37
6.18 Number of pseudowords according to query length. 38

6.19 Elapsed time on performing a query according to query length. 38

List of Tables

4.1 Example of table filled for calculating the distance between "ELISSABETH" and "ELISABETHA". 13
4.2 Table calculating levenshtein distance between "TAPS" and "TOP", and table to calculate levenshtein distance between "TAPS" and "TOPS". 16

6.1 Table comparing the efficiency between MARISA and TRE regexp 26
6.2 Table comparing the efficiency of the techniques seen in chapter four. . . . 30
6.3 Recall, misses and false positives of standard search and approximate string search allowing one, two or three edit operations. 32

List of algorithms

4.1 The Wagner–Fischer algorithm. 12
4.2 Insert string into BK-Tree. 14
4.3 Search string in a BK-Tree. 15

CHAPTER 1
Introduction

1.1 Motivation

Since the invention of writing, the humanity has stored and shared knowledge in handwritten documents. Over the centuries, the amount of handwritten documents has increased considerably and therefore, we find massive historical handwritten text collections stored in thousands of kilometers of shelves in archives and libraries. In these documents, we can find relevant information about the history and evolution of our literature, philosophy, politics and the human language. Unfortunately, these documents are useless in this format, as the relevant information for experts is hidden inside the large amount of texts that are archived.

In order to help the experts to discover the relevant information in these documents and making them more accessible, the Pattern Recognition and Human Language technology (PRHLT) at UPV, in collaboration with other research groups all over the world, has developed an initiative to recognize historical handwritten texts, index them and allow searches over them, in a project called tranScriptorium.

As part of the tranScriptorium project, the PRHLT has developed a search engine able to perform queries over the historical documents. This engine allows the classic boolean queries (AND, OR, NOT), sequence queries, and proximity queries.

1.2 Objectives

Despite the success of the system among the users, sometimes it is difficult to find information in the documents for various reasons:

Firstly, the user might be not sure about the spelling of a word. As the human language evolves through time, there might be differences in the spelling between the spelling of the query performed by the user and the spelling of the word at the historical epoch where the text was written.

Secondly, the word that we are looking might be well written on the text, but our system didn't recognize it properly.

These reasons would not allow the user to retrieve the information, notwithstanding it is present on the text. For example, in some historical texts the word "Elisabeth" could be spelled like "Elisabet", "Elisabetha" or "Elisabeta". Given this, the users would like to be able to search words which they are not sure about their spelling. Also, they would like performing searches of variants of a word. Thus, the users would like to perform approximate searches.

Approximate search is the search of the words that match approximately a pattern. In this thesis, we will distinguish between two types of approximate search queries: the wildcard queries and the approximate string queries.

A wildcard query is a query where we find a wildcard, which is a symbol that represents a sequence of zero or more characters. It is very helpful to search variants of a word or when we do not know the correct spelling of a word.

An approximate string query is a query where we specify the number of character insertions, deletions and substitutions that we allow in our search. Therefore, it is very useful to perform queries of words that we are not sure about their spelling.

Both techniques can be modelled through regular expressions, given that a regular expression can be defined as a sequence of characters that conforms a search pattern. The example mentioned before could be modelled as a wildcard query using the regular expression "Elisabet.*", where the "." represents any character and the "*" represents the empty string or a sequence of any length of the last character before him, in this case ".". Using approximate string search and allowing 1 errors, it could be modelled as ".?lisabeth" | "E.?isabeth" | "El.?sabeth" |"Eli.?abeth" | "Elis.?beth" | "Elisa.?eth" | "Elisab.?th" | "Elisabe.?h" | "Elisabet.?". where the symbol "?" represents 0 or 1 occurrences of the preceding symbol, in this case ".".

However, the use of regular expressions as queries implies two main difficulties:

Firstly, the users must learn regular expression syntax in order to perform queries and secondly, regular expressions can be too computationally demanding if they are not used with care. Given that the users only want to perform two concrete cases, wildcard search and approximate string search, of all the possibilities that regular expressions offer, we have developed in this thesis an interface to make the searches easier for them.

In this interface, the wildcard search will be modelled as a query with the "*" symbol, which specifies that in that position there could be a sequence of characters of any length. The approximate string search is modelled with the "$\sim$" symbol and a number after it, which represents the number of errors allowed.

Therefore, the objectives of this project are:

1. Develop the wildcard (prefix, suffix and interfix search) and the approximate string search in our search engine, in order to handle misspellings.
2. Perform these searches in a reasonable time with a reasonable memory usage.
3. Measure the improvement of the approximate search over regular search in the system.

1.3 Thesis structure

In order to explain the whole process of developing the wildcard and approximate search, this thesis is structured in six chapters.

In the second chapter, we will describe in detail the actual search system, as well as some key concepts that will be useful for the rest of the thesis.

In the third and fourth chapter, you will find an explanation of what is wildcard search and an approximate string search respectively, as well as a sum up of the different techniques that have been tried to perform these queries.

In the fifth chapter, you will find details of how the techniques seen in chapters three and four are used in combination with the already existing search engine, in order to

perform approximate searches. Also, you will be able to see a demonstrator working with this software.

In the sixth chapter, you will find the experimental results of the techniques seen in the previous chapters in terms of speed and memory usage. We will also assess the performance of the system using other classical information retrieval measures.

Finally, in the last chapter you will find the conclusions and further work of this project, as well as an explanation of the relationship between the degree and this thesis.

CHAPTER 2
Description of the initial system

In this chapter we will describe in detail the already existing search engine.

In our system, we can distinguish three different phases:

Firstly, the KWS and indexing tool. In this phase we pre-compute the probabilistic indices from the images. Secondly, the ingestion phase, where we create the actual database where our system will perform the queries. This two phases are offline phases, as they are only computed once. Finally, we find the keywords search phase, where the system analyses the queries of the users, searches the relevant information and displays the retrieved images. This last phase must be fast.

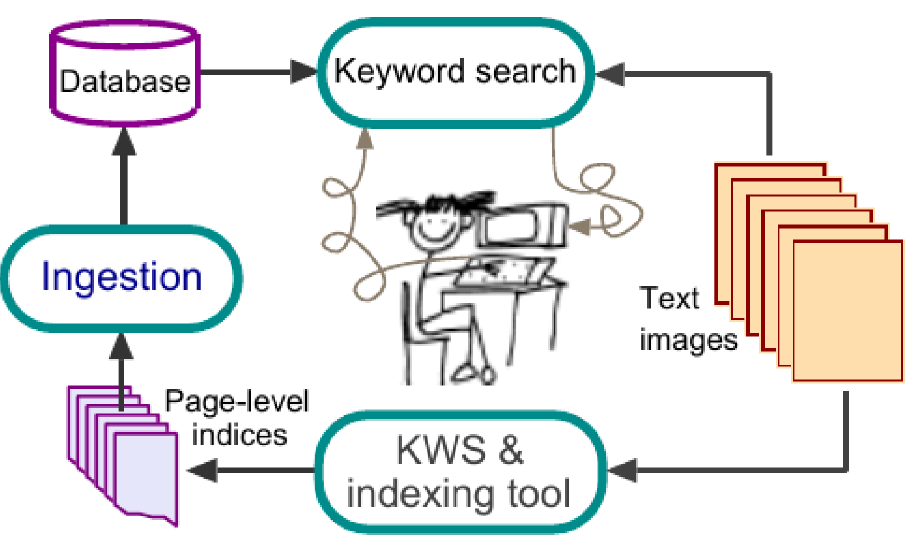

Figure 2.1: Workflow diagram of the system.

In the KWS and indexing tool phase, we use of Keyword Spotting techniques to generate probabilistic indexes. In our system, we require a small dataset of transcribed images to train optical and language models.Then, these models are used in the contextual word recognizer, in order to create intermediate rich structures, such as character and or word latices. Finally, these rich substructures are used by the KWS and indexing process to generate the page-level probabilistic indices.

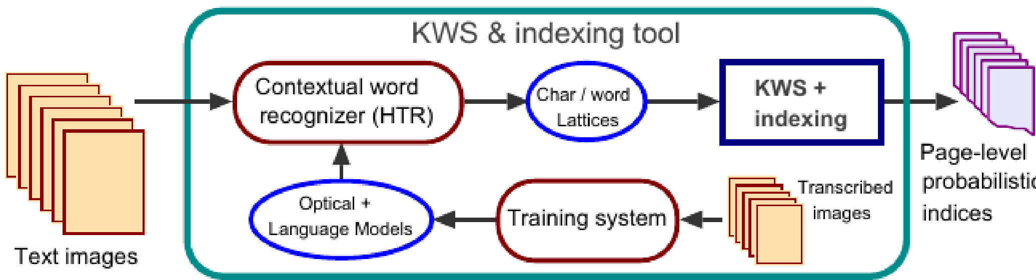

Figure 2.2: Workflow diagram of the KWS and indexing tool phase.

In the ingestion phase, the page-level probabilistic indexes are stored in an adequate data structure that allows to retrieve the information demanded by the user. In this phase, a set of processes are performed on the data: char and diacritics folding, word grouping (index by lemma rather than by word) and trim the index to the desired indexing density. The term density refers to a relevance probability threshold or a number specifying how many spots per page, per image region, or per running word should be indexed.

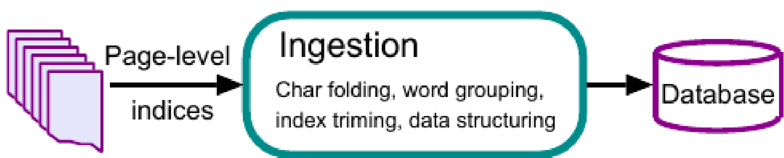

Figure 2.3: Workflow diagram of the index ingestion phase.

In the keyword search phase, the user interacts with the GUI in order to perform a query. The query performed by the user is analysed by the system, which retrieves the relevant information for it. Finally, the system displays the images that correspond to the query made by the user.

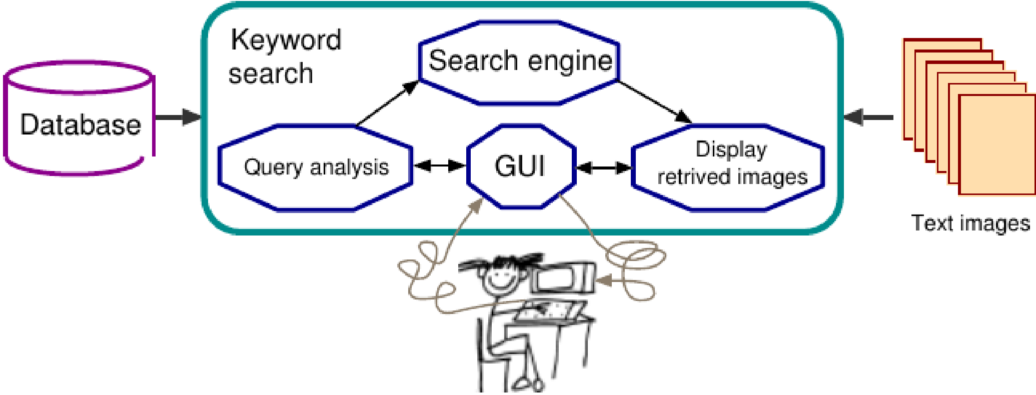

Figure 2.4: Workflow diagram of the keyword search phase.

The work of this thesis will focus on the query analysis process, as we will analyze the approximate query performed by the user, in order to convert it into a regular OR search of the words that match the pattern.

CHAPTER 3
Wildcard search

Wildcard search is a query where we find a wildcard, which is a symbol that represents a sequence of zero or more characters. It is very helpful to search variants of a word or when we do not know the correct spelling of the word. In our system, the chosen symbol is "*". For example, we could perform the query "offen*" to search for "offend", "offense", "offenses", "offender", "offending", etc.

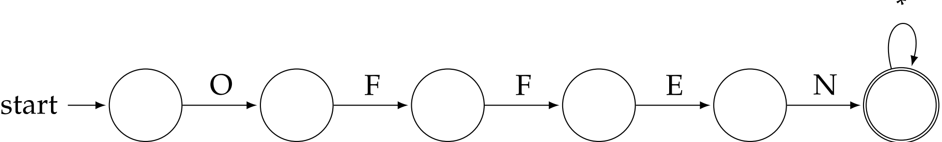

Figure 3.1: Automaton recognising the strings that start with the prefix "OFFEN".

The techniques and data structures used to perform approximate search, can be grouped in two main categories: online and offline. In the online category, the whole search process is done at runtime, unlike in the offline group, where some pre-computation of the vocabulary is done before in order to improve the search performance. On the one hand, we find that the main advantage of the online group is that they do not consume hard disk space for storing any data structure, as all the computation is done at runtime. On the other hand, the offline group does use hard disk space for storing the data into an appropriate data structure. However, it is expected that the offline group has a better time performance, as it has already done some pre-computation.

3.1 Online techniques

3.1.1. Regular expressions

As online method, we have used regular expressions to perform wildcard searches. A regular expression can be defined as a sequence of characters that conforms a search pattern. In combination with string searching algorithms, they are used to retrieve the words that match the pattern.

The library used as regular expressions matching engine is TRE. This library makes use of regular expressions with POSIX syntax. It also allows performing approximate string search. The temporal cost of the matching algorithm employed by TRE is O(M^2N), where M is the length of the regular expression and N is the length of the word against which we are trying to match the pattern.

To perform our wildcard search, we will compare every word of our lexicon with the pattern. Taking this into account, the runtime cost will be O(WM^2N), where W is the number of word that conform the vocabulary.

3.2 Offline techniques

3.2.1. MARISA trie

As offline method, we have used a MARISA trie, which is an specific case of a search trie.

Firstly, a trie, also known as prefix tree, is a search tree where strings with common prefixes are grouped. With this purpose, each edge represents a character of a string and every node represent the string that is formed by the concatenation of the edge labels from the root node to him. The cost of performing a lookup on the trie is O(M), where M is the length of the search string.

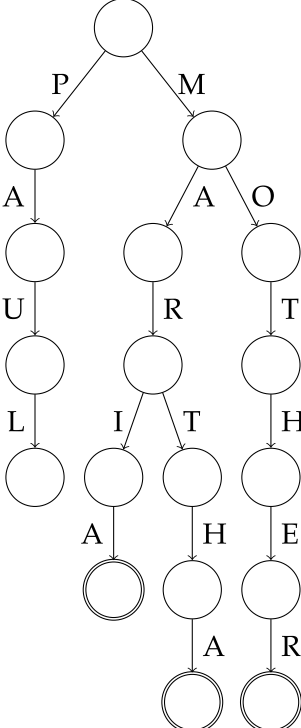

Figure 3.2: Trie storing the words: "PAUL", "MARIA", "MARTHA", "MOTHER".

Secondly, a PATRICIA Tree is a compressed binary trie, where, unlike in the standard trie, the labels of the edges can be strings and not only characters. These strings, in a PATRICIA trie represent which position of the binarized string is going to differentiate between the left and right sub-trees. Moreover, all the nodes that are the only child are merged with their parent. This fact makes our data structure more compact than a standard trie, as it requires less nodes for representing the same words. The cost is O(K) where K is the length of the largest word that its in it.

Finally, a MARISA trie [1] is a static trie that that consists of recursively compressed Patricia tries, where its edge labels are encoded using another Patricia trie.

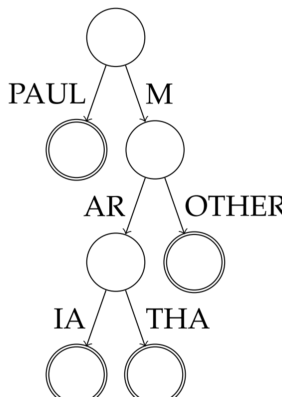

Figure 3.3: Patricia trie storing the words: "PAUL", "MARIA", "MARTHA", "MOTHER".

To perform wildcard search, we have two MARISA tries, one which is the regular prefix tree and another one as a suffix tree. In order to model a suffix tree with a trie, we create it taking as input the reversed dictionary. Given this, if we want to perform a prefix search, we will need to perform the search on the prefix tree. If we want to perform a suffix search, we will have to reverse the word that we want to lookup, perform the search and reverse the words that are retrieved. Finally, if we want to perform an interfix search, we will have to perform the intersection between the result of the prefix and suffix search.

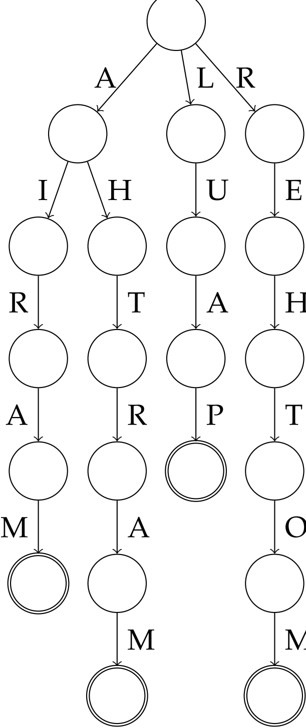

Figure 3.4: Trie used as suffix tree storing the words: "PAUL", "MARIA", "MARTHA", "MOTHER".

CHAPTER 4
Approximate string search

An approximate string query is a query where we specify the number of character operations that we allow in our search. It is very useful when we are not sure about spelling of a word.

As we are using levenshtein distance as distance metric, there are three different character operations:

1. Insertion: Lisabeth → **E**lisabeth (Insert character "E" at the beginning).
2. Deletion : Eli**s**sabeth → Elisabeth (Deletion of the character "s").
3. Substitution : Mari**a** → Mari**o** (substitution of "a" for "o").

These three operations have a cost of 1, unless when the substitution replaces the exact same character, in that case its cost is 0.

In our system, the chosen symbol for denoting this operation is "∼" and it is followed by the number of allowed character operations that we allow. For example, we could perform the query "offen ∼ 2" to find "ofen", "offense", "ofend", etc.

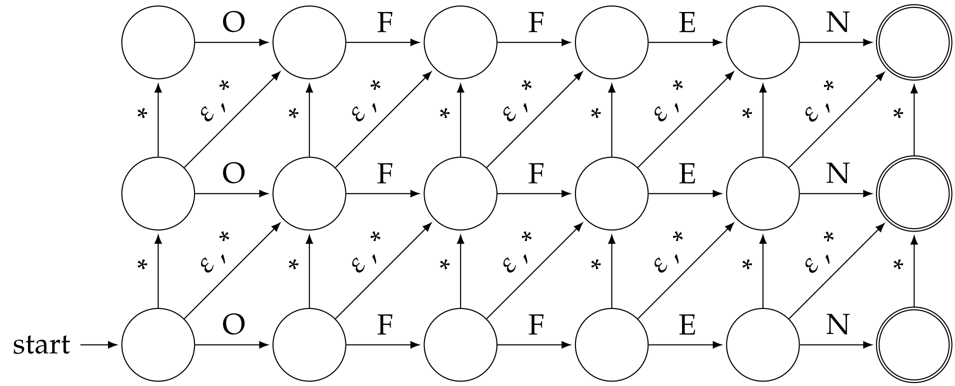

Figure 4.1: Automaton accepting strings with at most 2 edit operations from "OFFEN".

In the figure above, please notice that "ε" represents the empty string and "*" represents any character. Therefore, the horizontal transitions represents a substitution with cost 0, as we are substituting a character by itself, the vertical transitions represent a character insertion, the diagonal transition with "ε" represent a deletion and the diagonal transition with "*" represents a substitution of a different character.

4.1 Online techniques

4.1.1. The dynamic programming algorithm

Firstly, we are going to discuss the dynamic programming approach. Dynamic programming could be defined as an algorithm design technique which splits a problem into subproblems and then it calculates the optimal answer to the them. Once they are calculated, they are stored in order to avoid to recompute them again if there is any overlap. This technique allows us to explore all the possible solutions, avoiding repeating previous computations.

In this problem, we are going to compute the number of edit operations that have been performed to match the ith first characters of string X to the jth characters of string Y that have been matched. The solution presented by the Wagner–Fischer algorithm [3] follows the bottom-up approach, where we start solving the base cases to finally achieve the computation of the minimal edit distance between X and Y, which can be retrieved at table[n][m]. An example of table filled with this algorithm can be seen in table 4.1.

Algorithm 4.1 The Wagner–Fischer algorithm.

```
Let X and Y be the two strings that we want to compare, and n and m their length

for i = 0 to n do
  table[i][0] = i;
end for

for j = 0 to m do
  table[0][j] = j;
end for

for i = 1 to n do
  for j = 1 to m do
    if X[i-1] == Y[j-1] then
      cost = 0
    else
      cost = 1
    end if
    table[i][j] = minimum( 1 + table[i-1][j], 1 + table[i][j-1], cost + table[i-1][j-1])
  end for
end for

return table[n][m]
```

The time cost of this function is O(NM) where N is the length of string X and M is the length of string Y. If we assume that $N \approx M$, we get that the time cost is O(N^2). Moreover, as we are comparing the searched word against the whole vocabulary, the runtime cost will be O(WNM), where W is the number of word that conform the vocabulary.

		E	L	I	S	S	A	B	E	T	H
	0	1	2	3	4	5	6	7	8	9	10
E	1	0	1	2	3	4	5	6	7	8	9
L	2	1	0	1	2	3	4	5	6	7	8
I	3	2	1	0	1	2	3	4	5	6	7
S	4	3	2	1	0	1	2	3	4	5	6
A	5	4	3	2	1	1	1	2	3	4	5
B	6	5	4	3	2	2	2	1	2	3	4
E	7	6	5	4	3	3	3	2	1	2	3
T	8	7	6	5	4	4	4	3	2	1	2
H	9	8	7	6	5	5	5	4	3	2	1
A	10	9	8	7	6	6	5	5	4	3	2

Table 4.1: Example of table filled for calculating the distance between "ELISSABETH" and "ELISABETHA".

4.1.2. Regular expressions

The second online approach employed to perform the approximate string search is regular expressions. As library, we have employed TRE, which has already been discussed in Section 3.1.1.

4.2 Offline techniques

4.2.1. BK-Tree

As first offline technique, we have employed a BK-tree [2], which is a concrete case of metric tree.

Firstly, a metric tree is a tree data structure that index the data in a metric space.

A metric space is a representation space for data that satisfies the next three properties:

1. $d(p,q) = 0 \Leftrightarrow p = q$ (identity of indiscernibles)
2. $d(p,q) = d(q,p)$ (symmetry)
3. $d(p,q) + d(q,r) \geq d(p,r)$ (triangle inequality)

Secondly, a BK-tree is a metric tree designed to retrieve the strings that are in a given distance range from a given string. This data structure was proposed by Burkhard and Keller in 1973, in their paper "Some approaches to best match file searching".

As distance measure between two strings, it uses the well-known levenshtein distance, which is explained at the begining of this chapter. It is suitable for this case as it fulfills the three properties described above.

First, to create the BK-Tree, we will select arbitrarily one pseudoword of our vocabulary as the root of the tree. Then, we will take every pseudoword present in our lexicon and we will calculate its distance to the root string. If there is no child of the root with this number of editions in the tree, we add it with a edge cost equal to the number of editions. If there is already a child with the same number of editions, we compute the edit distance from the child to the pseudoword that we are trying to insert, and we repeat the process described before. This process is described formally in Algorithm 4.2. Also, an example of BK-Tree can be found in Figure 4.2.

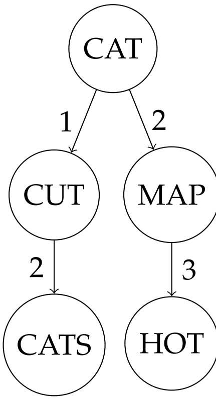

Figure 4.2: BK-Tree storing the words: "CAT", "CUT", "MAP", "HOT", "CATS"

Algorithm 4.2 Insert string into BK-Tree.

Let T be the BK-Tree, w be the word that we want to insert
if T *is empty* **then**
  $T = word$
else
  $currentWord = T.root$
  $children = T.getChildren(currentWord)$
  while $TRUE$ **do**
    $childAtDistance = null$
    $distance = levenshteinDistance(w, currentWord)$
    for *child in children* **do**
      if $levenstheinDistance(child, w) == distance$ **then**
        $childAtDistance = child$
        break
      end if
    end for
    if $childAtDistance == null$ **then**
      $children[distance] = w$
      break
    else
      $currentWord = childAtDistance$
    end if
  end while
end if

Second, in order to perform a query on a BK-Tree, we will take advantage of the triangle inequality property. Firstly, we will calculate the distance between the query and the root node. Next, we will add to the results list the root string if the distance is smaller or equal to the levenshtein distance that has been specified. Then, as the triangle inequality must hold, we know that all the possible results must hold in the interval [d - n, d + n], where "d" denotes the distance between the root node and the query, and "n" denotes the number of edit operations that are allowed. So, we insert the children of the root node that are in the interval described before, into a list of candidate nodes, and we repeat the process performed on the root node over each element of the candidates list. We repeat this process until the candidates list is empty and finally, we return the results list. This process is described formally in Algorithm 4.3.

Algorithm 4.3 Search string in a BK-Tree.

Let T be the BK-Tree, w be the query string and n be the maximum number of edit operations allowed
if *T is empty* **then**
  return []
end if
$candidates = [T.root]$
$result = []$
while *candidates is not empty* **do**
  $candidate = candidates.pop()$
  $children = T.getChildren(candidate)$
  $distance = levenshteinDistance(w, candidate)$
  if $distance \leq n$ **then**
    $result.append(candidate)$
  end if
  $lowerBound = distance - n$
  $upperBound = distance + n$
  for *child in children* **do**
    $childDistance = levenshteinDistance(candidate, child)$
    if $(childDistance \geq lowerBound)$ && $(childDistance \leq upperBound)$ **then**
      $candidates.append(child)$
    end if
  end for
end while
return results

The search cost is O($N * M * log(W)$), where N is the average length of a word in our lexicon, M is the length of the word that is used as query, $log(W)$ is the estimated depth of the BK-Tree, where W is the number of nodes.

4.2.2. DAWG levenshtein

The second offline technique that we have employed is a DAWG allowing levenshtein search.

Firstly, a DAWG (Directed Acyclic Word Graph) is a data structure designed to represent a set of strings. With this purpose, each edge represents a character of a string and each node represents the strings that are formed by all the possible paths from the root to itself. The main difference between a trie and a DAWG is that a DAWG eliminates the prefix, interfix and suffix redundancy, meanwhile the trie only eliminates prefix redundancy. An example can be seen in Figure 4.3.

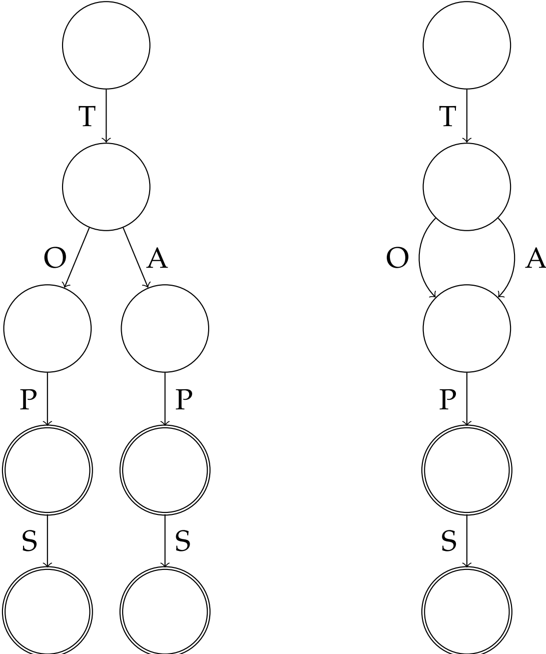

Figure 4.3: The strings "TOP", "TOPS", "TAP" and "TAPS" stored in a trie (left) or in a DAWG (right).

Now, we want to be able to calculate the levenshtein distance in an efficient way, taking advantage of the structure of the DAWG. If we take a look at table 4.2, we can observe that all the rows except the last one are equal to the rows on the left table. This is due to the fact that the word TOP is a prefix of the word TOPS. Thus, we observe that, if we have already calculated the edit distance table of a prefix of a word, to calculate the edit distance of that word we will only have to calculate the remaining rows of the table.

		T	**A**	**P**	**S**
	0	1	2	3	4
T	1	0	1	2	3
O	2	1	1	2	3
P	3	2	2	1	2

		T	**A**	**P**	**S**
	0	1	2	3	4
T	1	0	1	2	3
O	2	1	1	2	3
P	3	2	2	1	2
S	4	3	3	2	1

Table 4.2: Table calculating levenshtein distance between "TAPS" and "TOP", and table to calculate levenshtein distance between "TAPS" and "TOPS".

Taking this information into account, when we want to perform a search in the DAWG, we will calculate the edit distance of every node against the searched word, using the table calculated by the precedent node if any, in order to avoid recomputations [4]. Also, as we are only interested in knowing if a word is within a given edit distance, if there is no position where the cost is within the admitted edit distance in the last row that we have analysed for a branch, we will prune the search, as it will be impossible that any other word of that branch has a smaller or equal cost than the allowed one. The search cost is $O(K * N)$, where K is the largest length of a string in our lexicon and N is the number of nodes in our DAWG.

CHAPTER 5

Implementation and demonstrators

In this section we will show details about the implementation of the project, as well as some real examples of how the system works.

5.1 Implementation

As briefly mentioned in chapter 2, the work of this thesis is focused on the keyword search phase, concretely in the query analysis. Firstly, the system checks if the query that is going to be performed is an approximate search. If that is the case, the system performs the approximate search operation, using the appropriate technique explained in chapter three if wildcard search or in chapter four if approximate string search, and retrieves as result a list with all the strings that match the pattern. Then, the system searches each string of the list and retrieves the images that match with each word. Finally, the web server combines all the images which are retrieved by the index server and show them to the user.

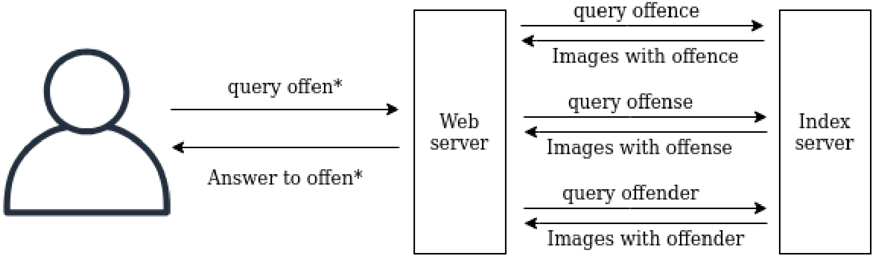

Figure 5.1: Workflow diagram of wildcard query "offen*".

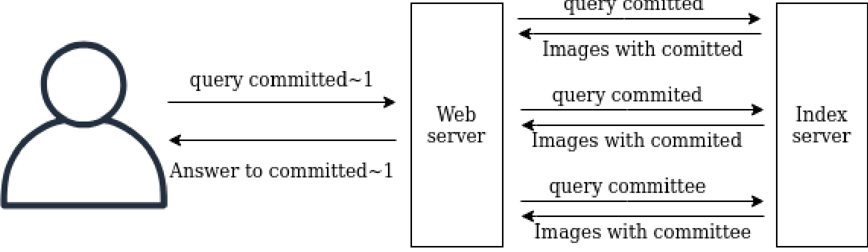

Figure 5.2: Workflow diagram of approximate string query "committted∼1".

In this project, we have implemented the programs to retrieve the lexicon of the collection, detect the approximate searches, handle them using the appropriate library, and transforming the list of words that match the query into an OR search. Also, we have used our own code for the dynamic programming approach explained in section 4.1.1. Finally, we have also developed programs to measure the performance of each of the different techniques explained in the precedent chapters.

With this purpose, the libraries that have been used are:

- TRE [9], as regular expressions library. It has been employed in wildcard search and approximate string search. It is developed in C++ and it is under the BSD 2-clause license.

- libmarisa [10], as MARISA trie library. It allows to perform regular lookups and prefix search. It is developed in C++ and it is under the BSD 2-clause license.

- Bk-Tree [11], as BK-tree library. It is developed in C++ and it is under the GPL-3.0 License.

- dawg-levenshtein [12], as DAWG library. It allows the creation of a DAWG and the approximate string search, using levenshtein distance as metric, over it. As it is inside the "brmson" project, its license is open source.

5.2 Bentham demonstrator

In order to try our implementation, we have adapted an existing demonstrator, which provided full search and retrieval capabilities for probabilistically indexed datasets, but it did not support queries including wildcard or approximate string search.

First, we have made the appropriate changes in the parser, in order to detect the approximate search query. Then, the system calls to the programs explained in the previous section and finally, it receives an OR search with all the words that match the pattern. This query is performed as usual by the search engine.

We have used the Bentham Papers collection as dataset. This collection is formed by 89111 images of manuscripts written in English by the philosopher and reformed Jeremy Bentham (1748-1832) and his secretarial staff.

This collection has been structured in three different search levels: HOME, BOX and PAGE:

The HOME levels corresponds to the set of documents that conform the full collection. The BOX level corresponds to the box where they are archived in the UCL archives. Finally, the PAGE level corresponds to the paper image. In order to prevent the high computational cost that might suppose the use of the wildcard search and approximate string search, some constraints have been added to the queries. In wildcard search, there are required 4 characters (without the "*") at HOME level, 2 at BOX level and 1 at PAGE level. In approximate string search, the difference between the length of the query (without the '∼') and the specified number of edit operations allowed must be greater than 3 at HOME, 1 at BOX level and 0 at image level.

In the search interface the user has the option of choosing a minimum confidence threshold and/or limiting the number or results displayed by page.

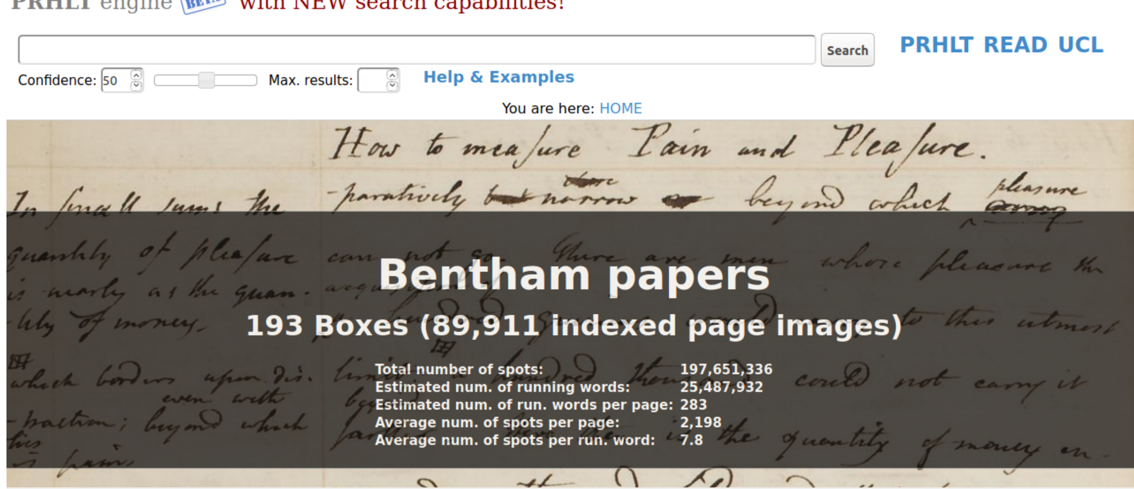

Figure 5.3: Screenshot of the HOME level of the Bentham demonstrator

In the next figures, you will find how a user would interact with the search system when performing a wildcard query. Firstly, the user would perform the query at the HOME level and the system would retrieve the boxes that match the query.

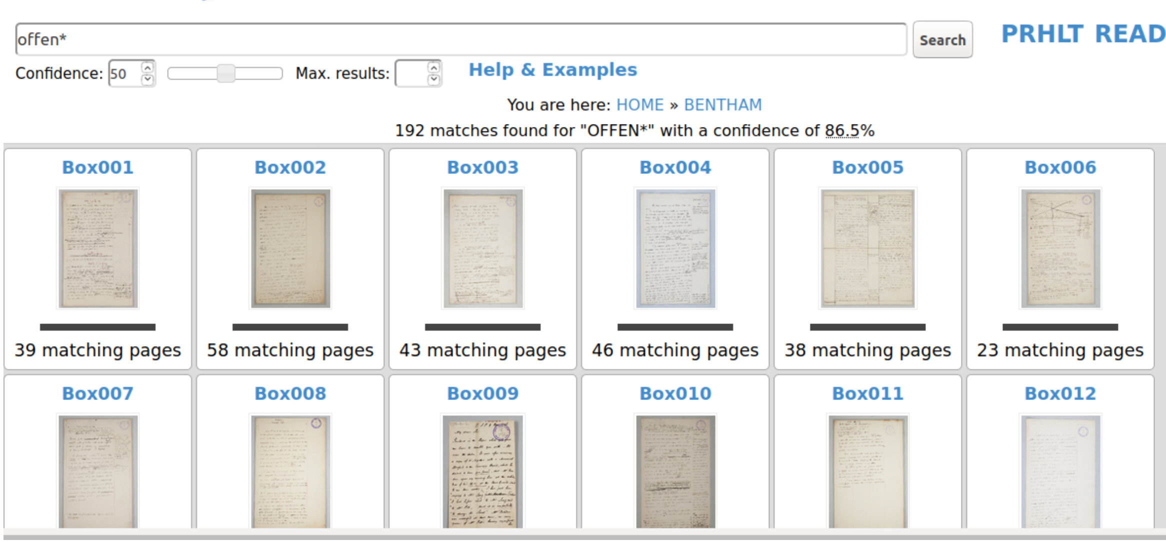

Figure 5.4: Result of the search "offen*" at HOME level

Then, the user would select the box where he wants to search, in this case we select box002.

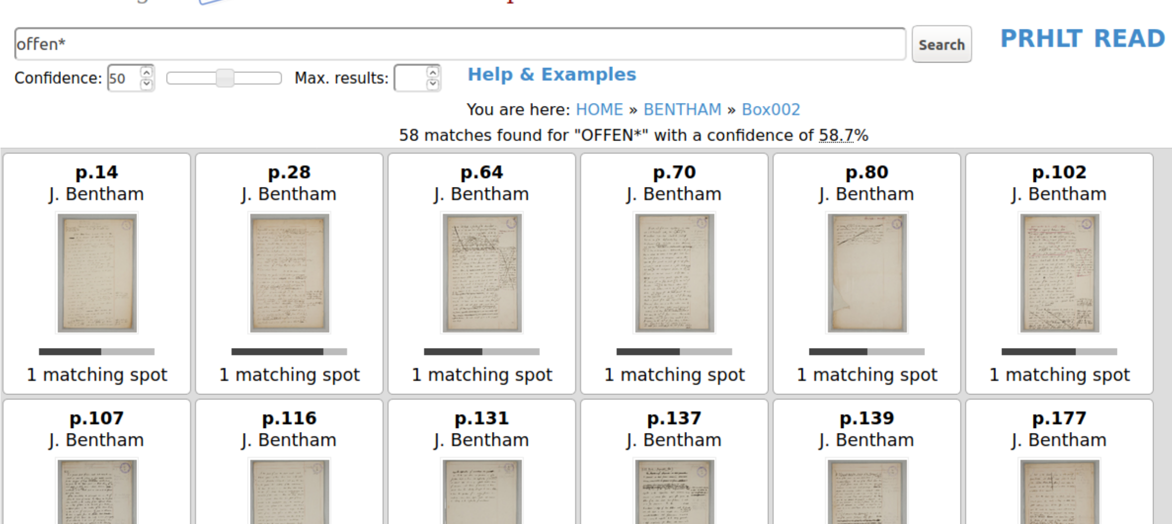

Figure 5.5: Result of the search "offen*" at BOX level

Finally, the user select the page that he wants to see in detail. When the user clicks on a page, the system displays the selected page and highlights the spot or spots that match the query.

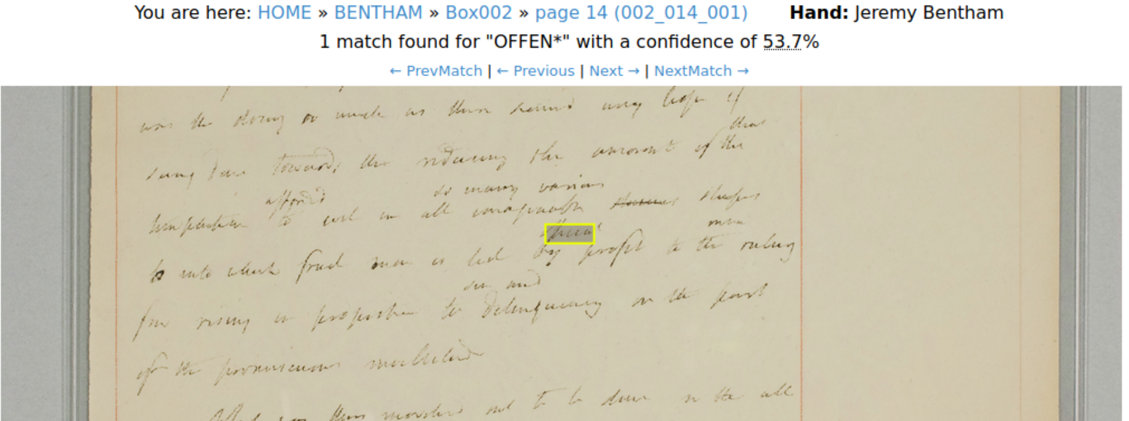

Figure 5.6: Matched spot for query "offen*" on page 14.

If the user clicks on a spot, he can visualize a list of possible pseudowords, ranked by their probability, that could be written on that concrete spot.

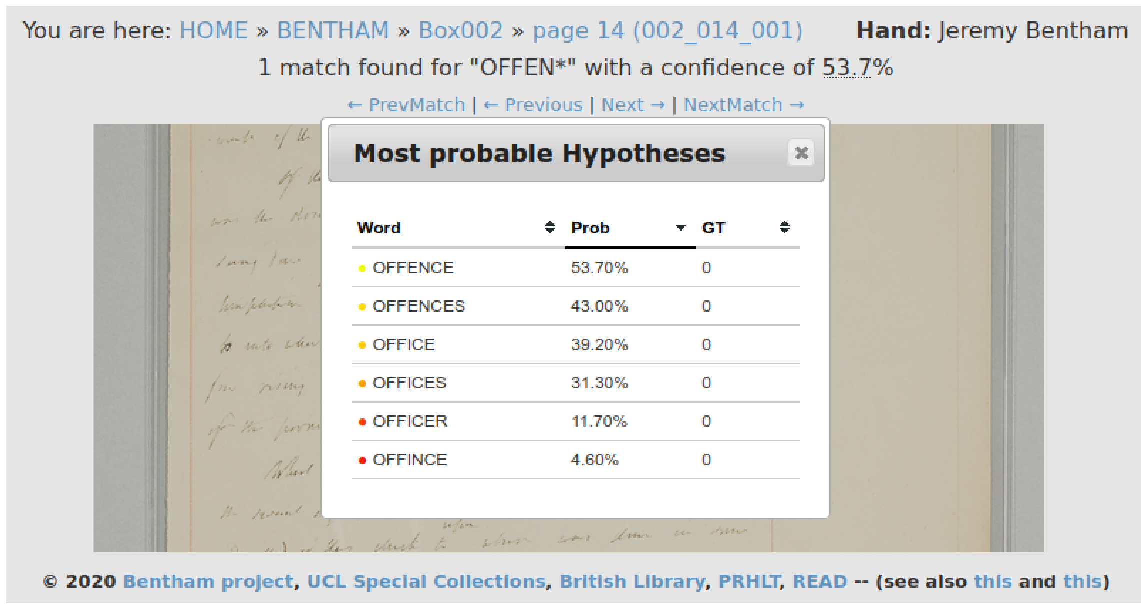

Figure 5.7: Pseudowords that matched the spot on page 14 for query "offen*".

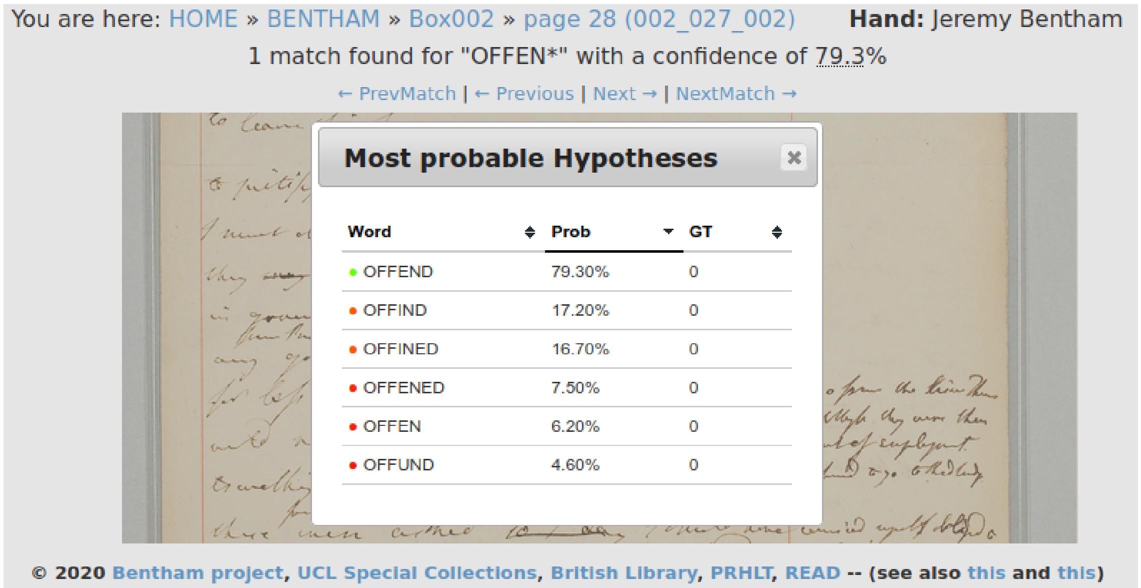

Figure 5.8: Pseudowords that matched the spot on page 24 for query "offen*".

The procedure would be exactly the same for the approximate string search. In the figure below you can observe an example for the query committed∼1.

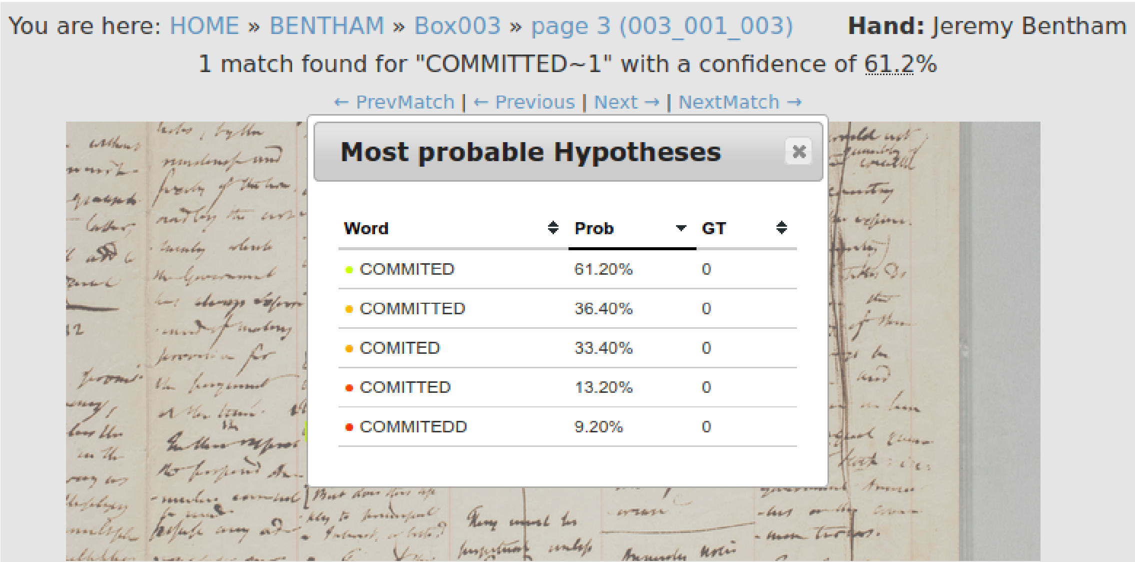

Figure 5.9: Pseudowords that matched the spot on page 3 for query "committed 1".

CHAPTER 6
Experimental results

Firstly, from this collection we are going to use only 357 transcribed page images as experimental dataset, in order to determine which are the optimal techniques for this dataset, as well as to evaluate the performance of our new query system. This reduced dataset is the "Bentham GT" collection. Secondly, we will measure the costs, time and performance of the full Bentham collection, formed by more than 89111 page images.

Please note that the query sets used for the experiments are all the possible prefixes and suffixes for the wildcard experiments and all the pseudowords for the approximate string search experiments. They were conducted on a sample of these query sets for each prefix, suffix or word length with a 95% confidence interval and a 1% margin of error. Moreover, in the plots where you find error bars, they represent two standard deviations of the mean of the experiment. Please note that when you find asymmetry on the errorbars is due to the fact that two times the standard deviation is greater than the average, and then the plotting software truncate them.

6.1 Experiments on Bentham GT

This collection is formed by 357 page images, with 89870 running words, a lexicon size of 6988 words and 5974668 different pseudowords. In figure 6.1 we can observe that most of the pseudowords have a length between four and sixteen.

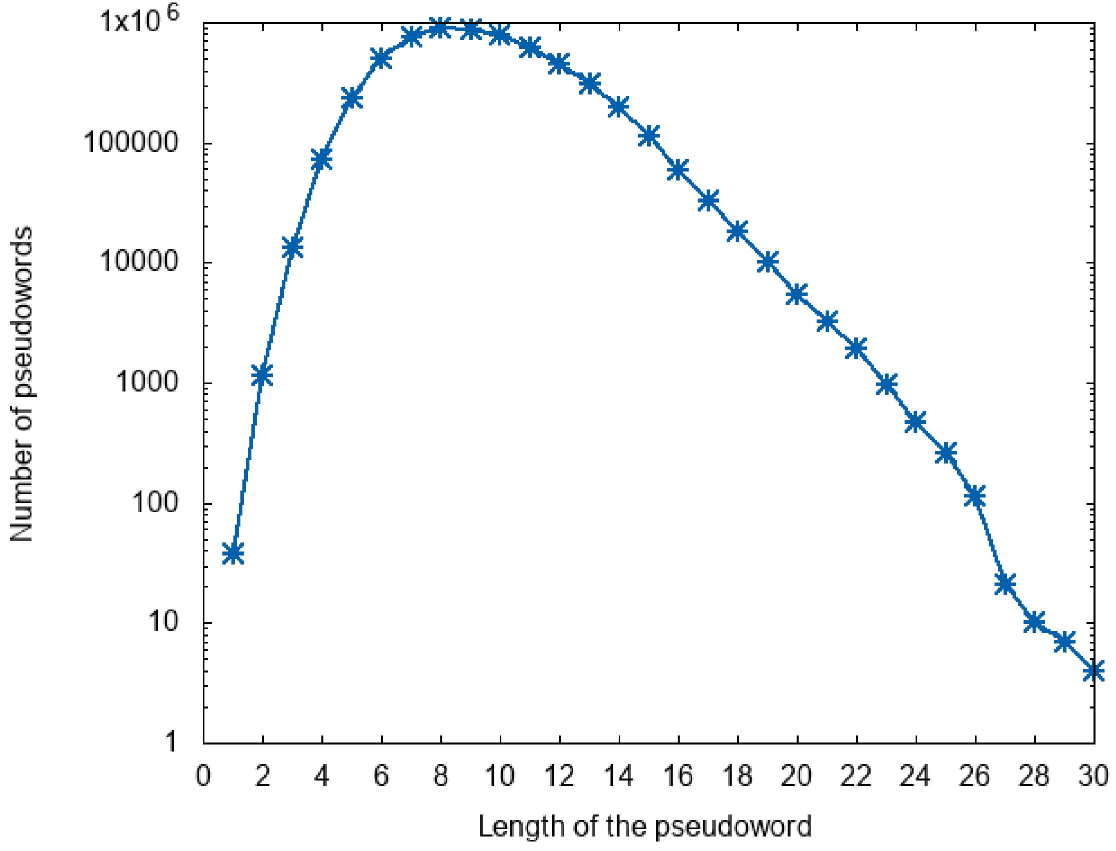

Figure 6.1: Number of pseudowords in Bentham GT according to their length.

6.1.1. Wildcard search

First, we are going compare the different techniques explained in Section 3, in order to determine which of them is the optimal for this collection.

If we take a look at Table 6.1, we can observe that both memory consumption are very reasonable, taking into account that we are storing almost six millions of pseudowords. However, we can observe that the MARISA trie has a better performance, as it consumes only a tiny amount of hard disk and RAM memory.

	MARISA	TRE regexp
Hard Disk space	27.7 MB	-
RAM space	28 MB	135 MB

Table 6.1: Table comparing the efficiency between MARISA and TRE regexp

Next, if we take a look at Figure 6.2 and 6.3, we observe that the MARISA trie performance is much better than the TRE regexp performance. This is due to the fact that the MARISA trie does some pre-computation at index time, meanwhile the TRE regexp does

all the computation at runtime. We can also observe that the elapsed time for TRE regexp and MARISA trie is almost constant, except the case of length 1 for MARISA trie. Taking this information into account, we have chosen as optimal technique the MARISA trie, as it is always faster than TRE regexp.

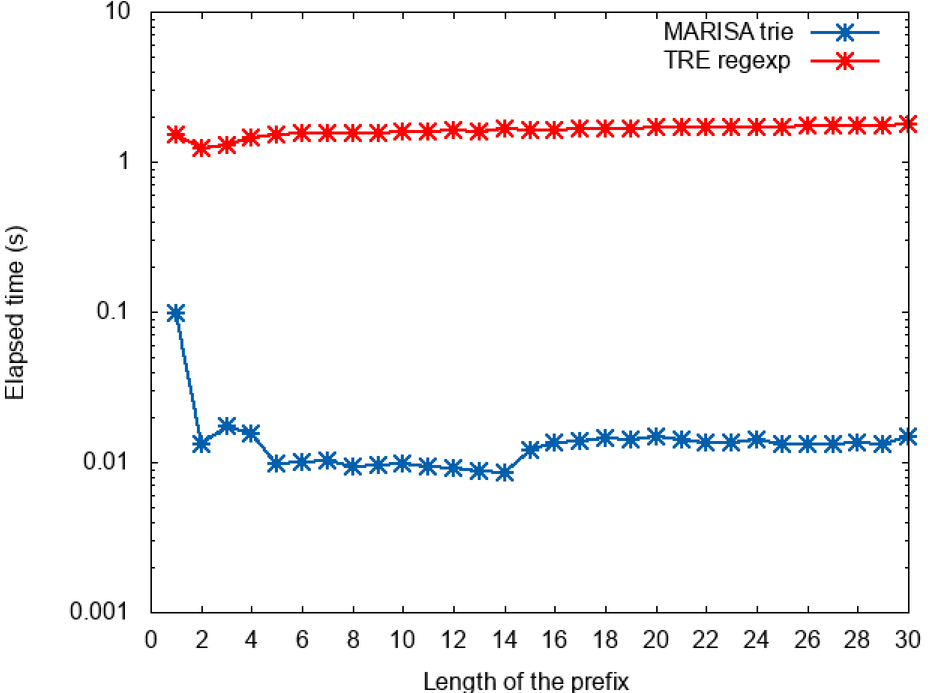

Figure 6.2: Elapsed time according to prefix length.

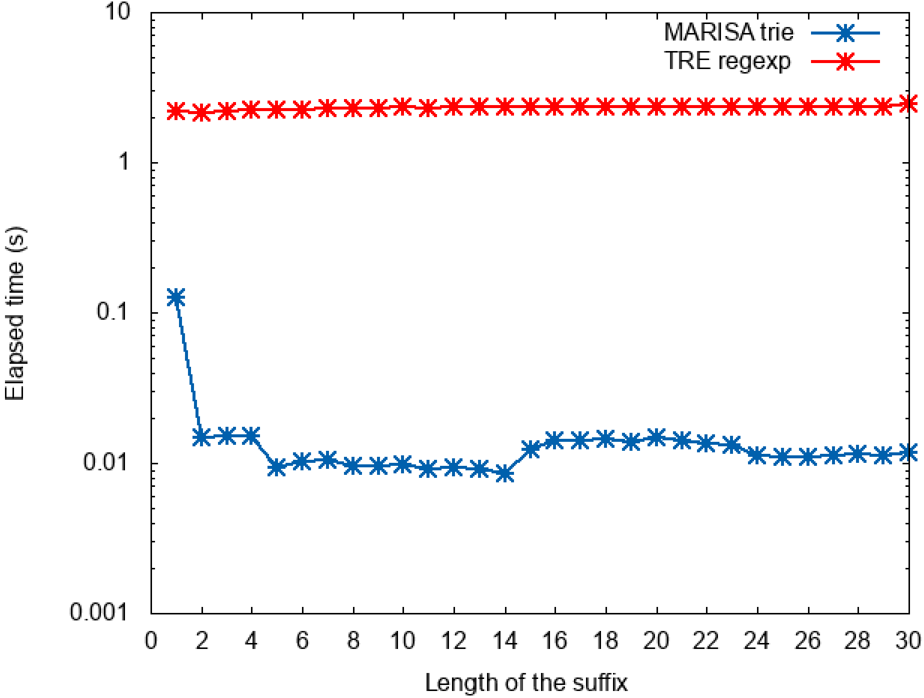

Figure 6.3: Elapsed time according to suffix length.

Now, if we take a look into the figures 6.4 and 6.5, we can observe that, the smaller the prefix or the suffix, the bigger the number of retrieved words is. Furthermore, we can observe that the most informative prefix and suffix length are between the lengths 4 and 8, given that they match a reasonable number of words. The prefixes and suffixes with a length in the intervals 1 to 3 match too many words and the ones with a length larger than 8 match less than 2 words on average, which might be not very informative. Finally, we can observe that there is a huge variability on the number of retrieved words, which means that some prefixes are very rare and others are very common. Please note that, when we make reference to prefix or suffix length of a string, we do not take into account the "*" symbol for the calculation of it.

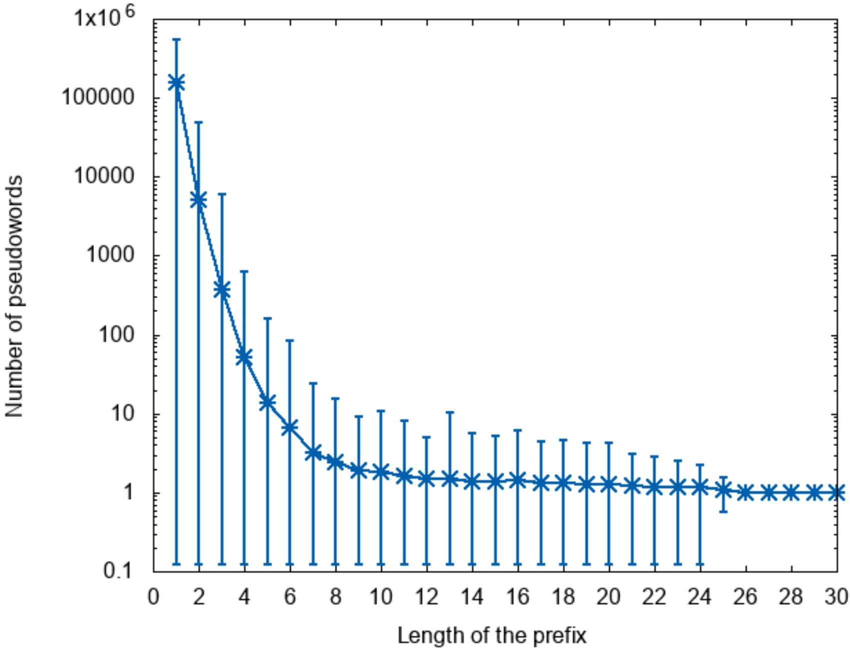

Figure 6.4: Number of pseudowords retrieved according to prefix length.

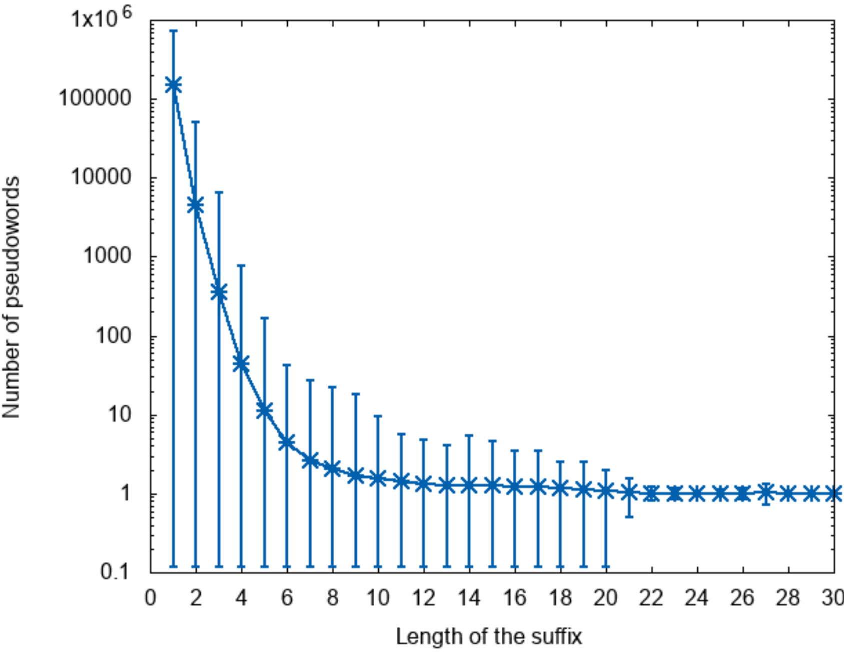

Figure 6.5: Number of pseudowords retrieved according to suffix length.

If we take a look at the time plots, we can see that the query that takes more time is the one with a prefix or suffix length of 1. This fact makes sense as it is the length that matches more pseudowords. The rest of the length queries take a similar time.

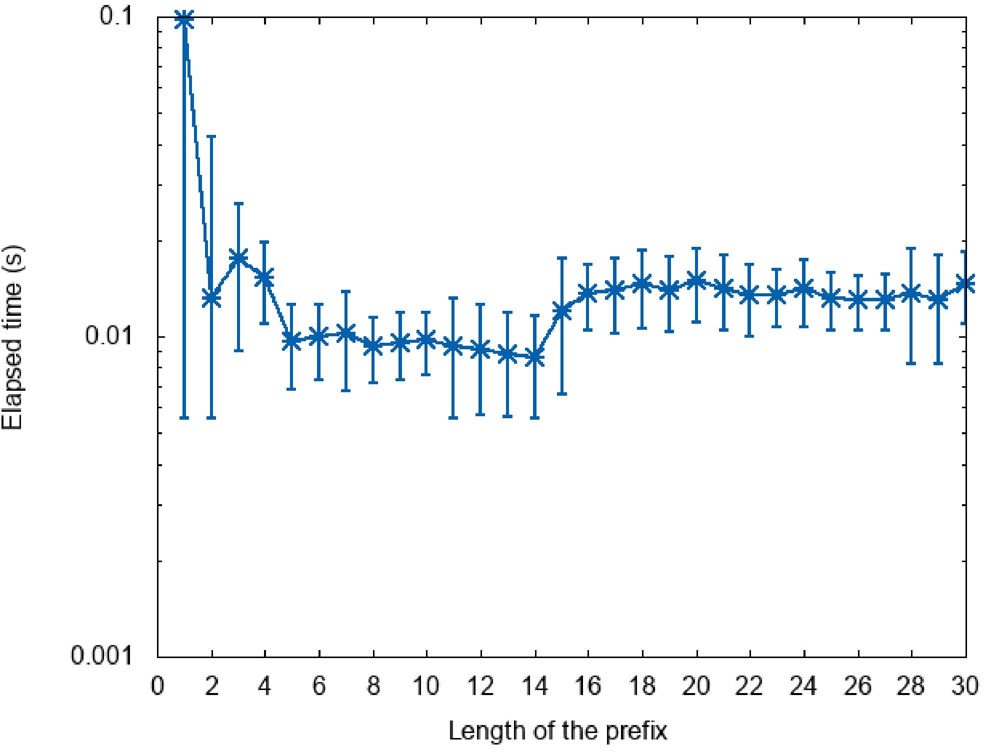

Figure 6.6: Elapsed time according to prefix length.

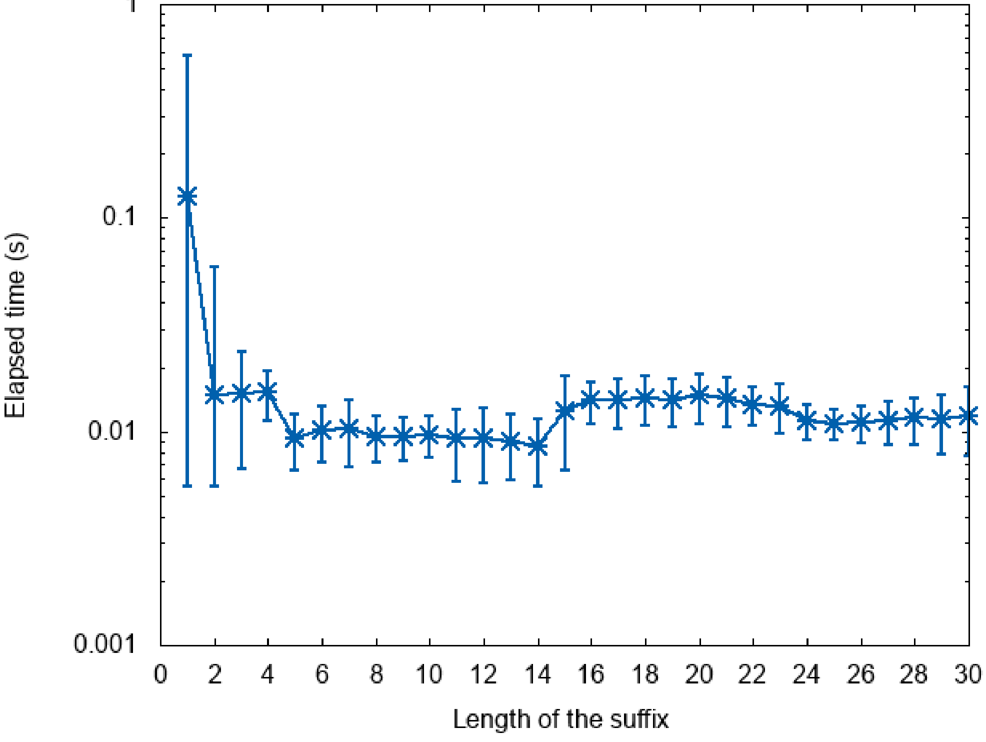

Figure 6.7: Elapsed time according to suffix length.

6.1.2. Approximate string search

First, we are going to compare the different techniques explained in section 4, in order to determine which of them is optimal for this collection.

If we take a look at Table 6.2, we can observe that both memory consumption are very reasonable, taking into account the amount of stored pseudowords. However, we can observe that the best performances are achieved by the dynamic programming algorithm and DAWG levenshtein, as both of them have a RAM consumption of 86MB. Finally, we would like to remark that BK-Tree does not consume hard disk space because the library that we are employing does not have any method to store the created tree. However, the elapsed time to construct the tree is less than one minute, so this approach stills reasonable.

	DP	TRE regexp	BK-Tree	DAWG levensthein
Hard disk space	-	-	-	21,1MB
RAM space	86MB	282,62MB	909,31MB	86MB

Table 6.2: Table comparing the efficiency of the techniques seen in chapter four.

Next, if we take a look at Figure 6.8 we observe that the best performing techniques belong to the offline techniques group, as expected. This is due to the fact that the offline techniques make some pre-computation at index time. From the offline techniques, the best performing technique is the DAWG levenshtein, as it takes less time than the BK-Tree. However, both techniques seem suitable for this collection, as both have a reasonable time performance.

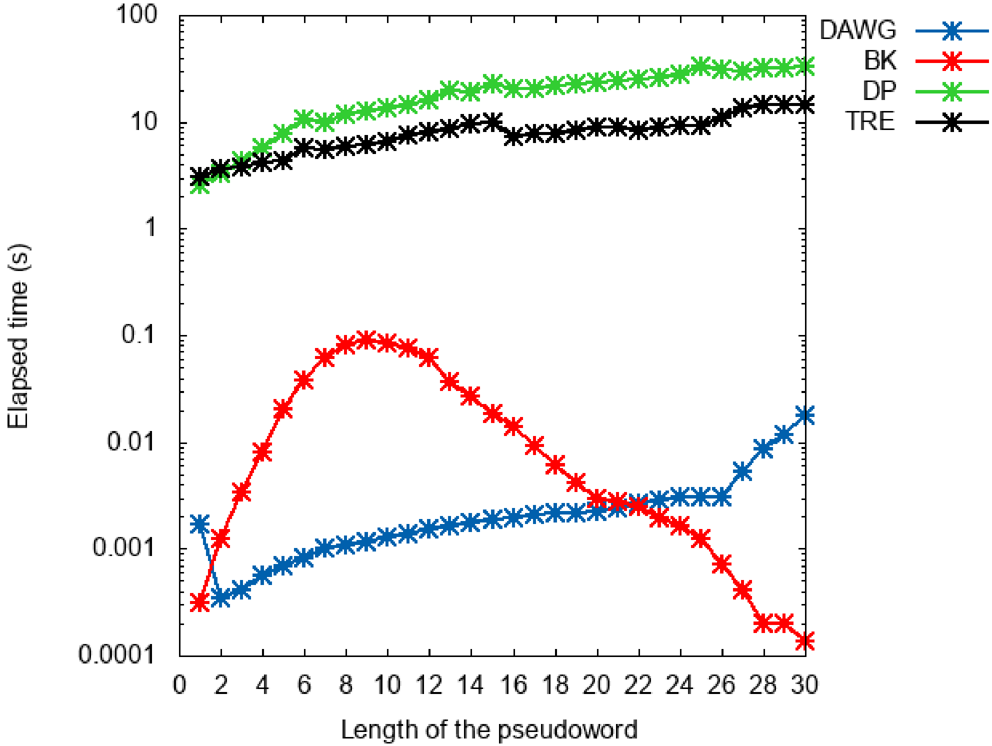

Figure 6.8: Elapsed time according to pseudoword length allowing one error

Taking all this information into account, we can affirm that DAWG levenshtein is the best approach to perform approximate string search, as it is the technique that has the better performance in terms of time and memory consumption.

Finally, if we observe the results obtained in the comparison of the different techniques for wildcard search and approximate string search, we can conclude that the offline techniques outperform the online techniques when we are searching over a large dictionary, as they have a faster time performance than the online techniques and their memory consumption is very reasonable. However, we would like to empathise the fact that online techniques could have very good results with a smaller lexicon and, in the concrete case of the regular expressions, they would also allow to the user to perform every type of query that has been defined in the regular expressions syntax.

Now, if we take a look at the first plot we can observe that the most useful number of allowed errors are 1 and 2, as 3 errors retrieves usually too many words. Also, we can observe that the most informative word lengths are between 4 and 23, as they retrieve a reasonable and similar number of words. In the second plot we can observe that the time to perform a search increases with the number of errors allowed, as it was expected to happen. Finally, we can see that the search time slightly increases with the length of the query.

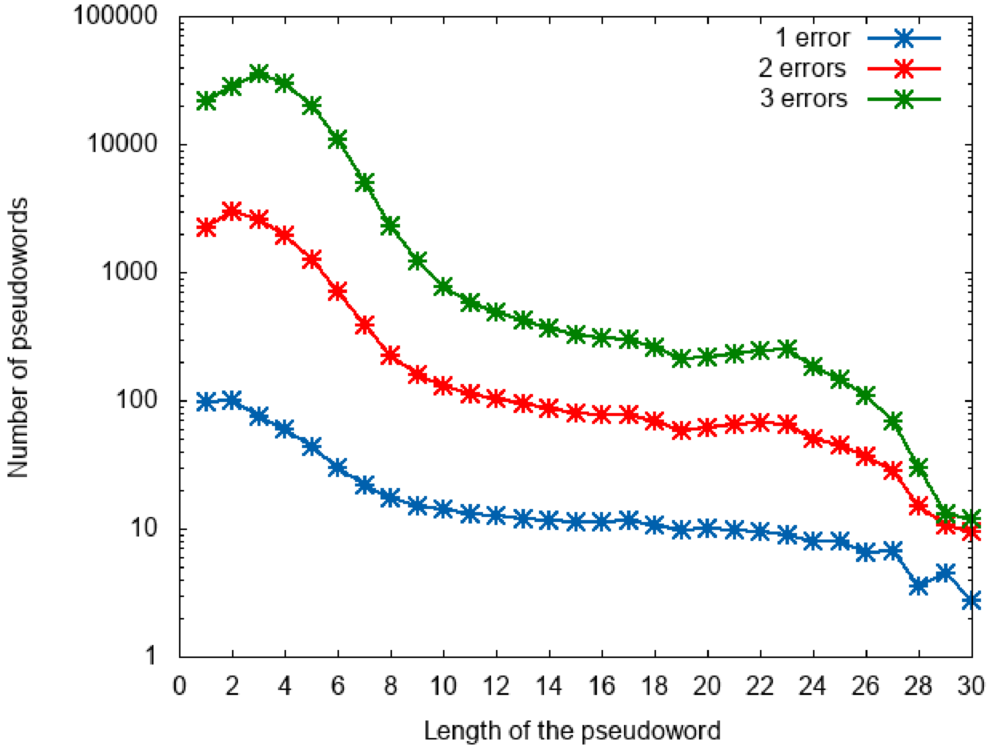

Figure 6.9: Number of pseudowords retrieved according to query length.

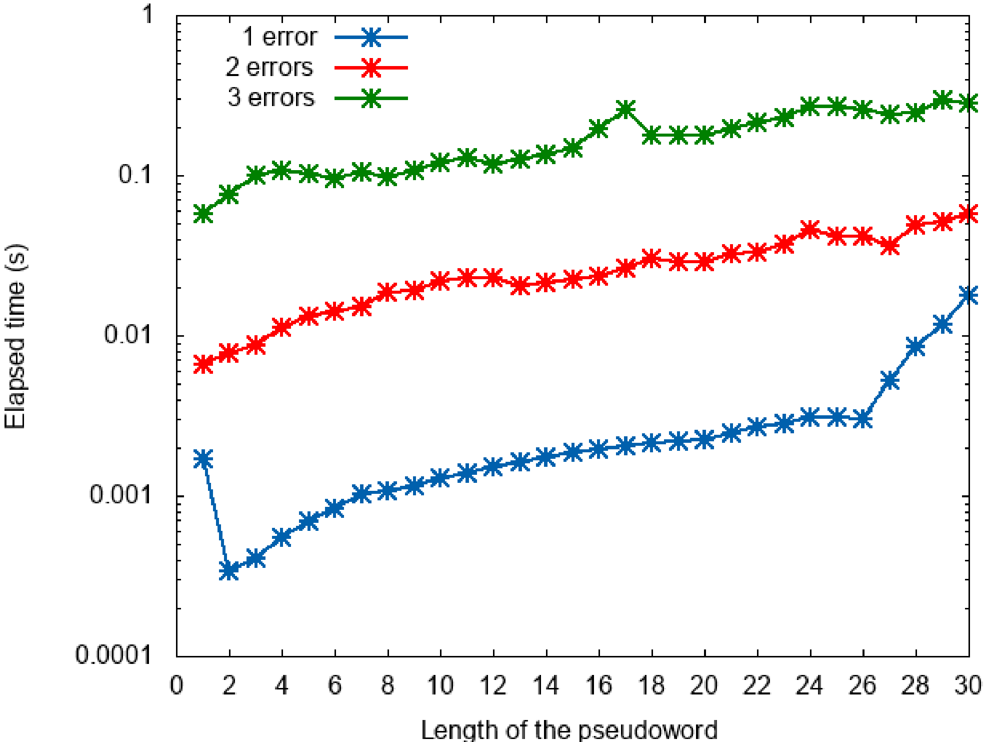

Figure 6.10: Elapsed time on performing a query according to query length.

Now, we will evaluate the approximate string search using some standard information retrieval measures:

Firstly, the precision is the percentage of relevant retrieved documents in the set of retrieved documents. It indicates to which extent retrieved documents are relevant.

Secondly, the recall is the percentage of relevant retrieved documents over the set of all relevant documents. It indicates to which extent relevant documents are retrieved.

Now, as one of the approximate search purposes is to use it when we are not really sure about the spelling of a word, we have used as queryset a list of 652 queries, which are present on the manual transcription of the pages but not on the probabilistic indexes that are provided by our system.

Over this query set, we have measured the recall over this queryset using approximate string search allowing one, two and three errors:

	RCmx	Misses	False positives
STD	0.000	652	2583
SEARCH~1	0.417	380	43496
SEARCH~2	0.701	195	252817
SEARCH~3	0.833	109	1021018

Table 6.3: Recall, misses and false positives of standard search and approximate string search allowing one, two or three edit operations.

Firstly, we observe that the recall of the standard search is 0, as expected. Also, we can observe that, allowing one and two errors, the recall has improved drastically while maintaining a suitable number of false positives. Finally, if we take a look at results when allowing three errors, we observe that we could improve even more the recall, but we would have a huge number of false positives, which is impractical.

Despite the success of approximate string search in this context, there is still one main problem: We give the same probability to each word that has been expanded from the query. For example, if we perform the approximate string search "case 1", we observe that we give the same probability to "cases", the plural of "case", than to other non semantically related words like "cast", "cash", "chase", "came", "cause".

With the purpose of measuring how informative a concrete expansion of a query is, we are going to calculate the overlap between queries. While using this measure, we suppose that if the query match only a few strings, it will probably be more informative than other queries that match more strings.

In the figure Figure 6.11, we can observe that most of the words match 2 words or less, apart from itself, when performing approximate string search allowing 1 error. This fact tells us that most of the queries are probable to be relevant, as the don't overlap with many words.

In order to complement this information, we have repeated this experiment taking into account the length of the query. To make make it more visual, we have added a little perturbation to the plot, in order to avoid the visual overlap. We can observe in Figure 6.12 that the queries with length smaller than three are not very informative, as they usually match more than 2 words. Also, we can observe in general that, the larger the pseudoword is, the less overlap there is.

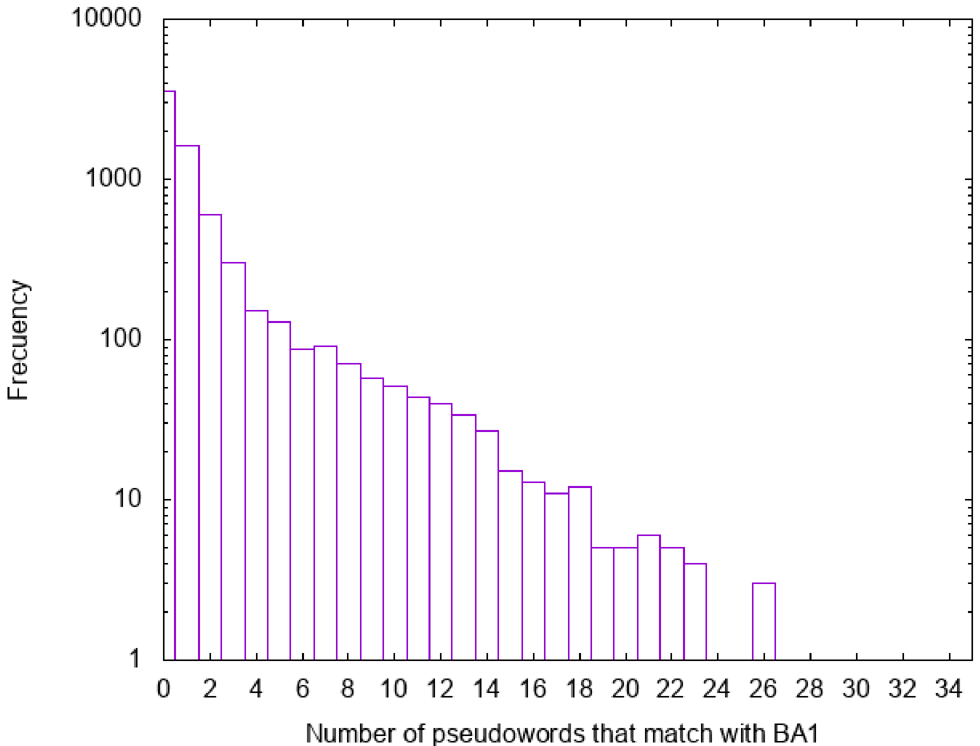

Figure 6.11: Frequency of the number of pseudowords that overlap

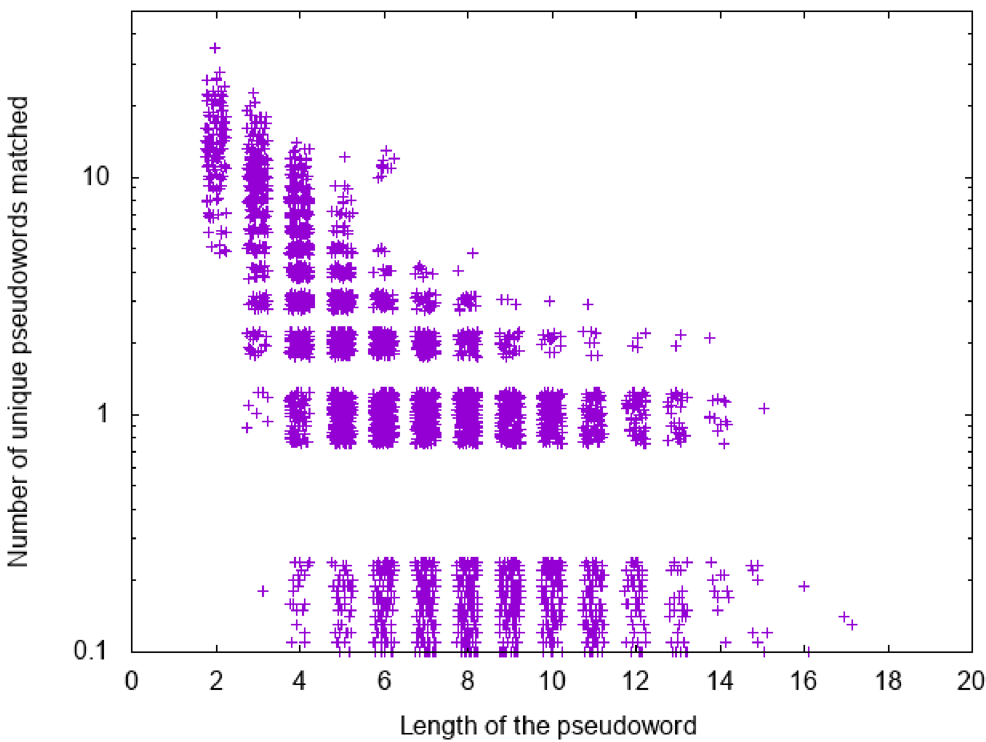

Figure 6.12: Number of unique pseudowords matched according to query length

6.2 Experiments on Bentham

This collection is formed by 89111 page images, with 25487932 running words and 37172635 different pseudowords. In figure 6.13 we can observe that most of the pseudowords have a length between four and sixteen.

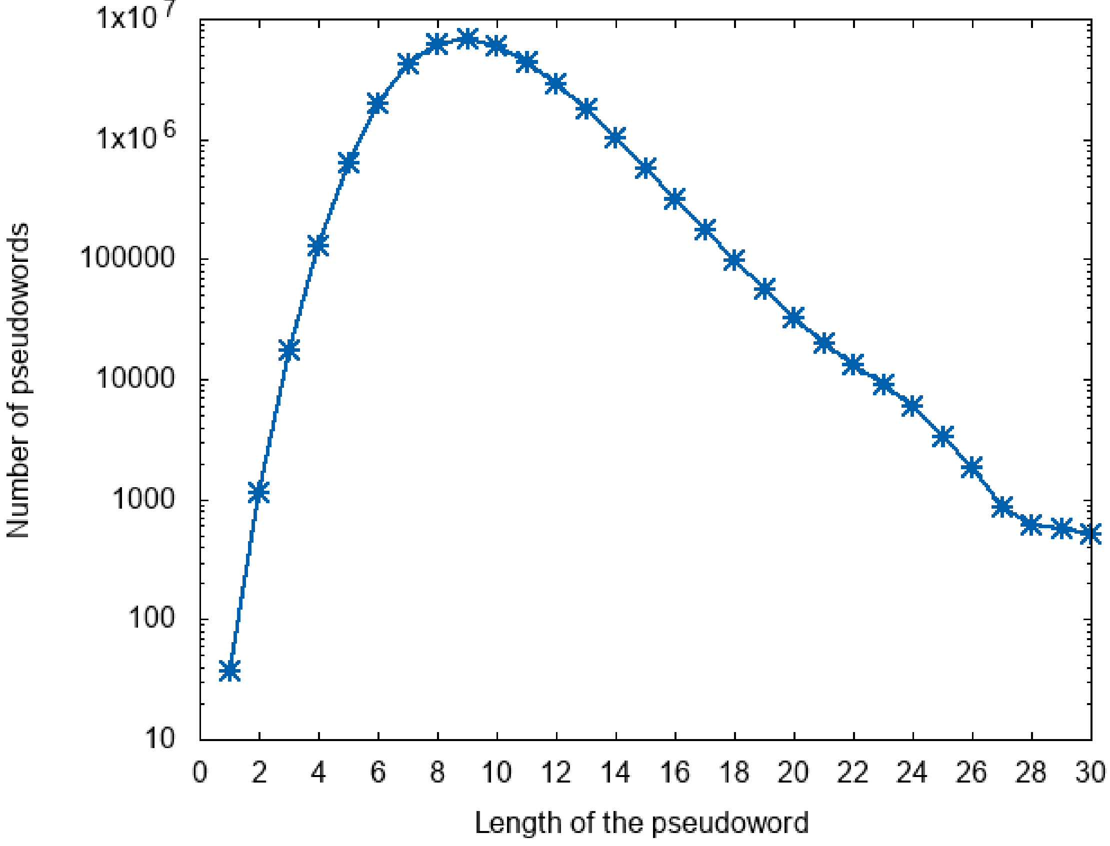

Figure 6.13: Number of pseudowords in Bentham according to their length.

6.2.1. Wildcard search

Firstly, the hard disk space usage of the MARISA trie is 89,6 MB and the RAM usage is 1.2GB. Again, we can affirm that both memory usages are very reasonable, taking into account the number of strings that are stored there.

Secondly, if we take a look at Figures 6.14 and 6.15, we can observe that the shape of the plot is very similar to the observed on the precedent chapter, but with a higher order of magnitude on the y axis. Also, we find again that the query that takes most time is the one with prefix or suffix length 1. Finally, we can affirm that the results of the queries are retrieved in a reasonable time, as it usually takes about 0.1 seconds to retrieved the results.

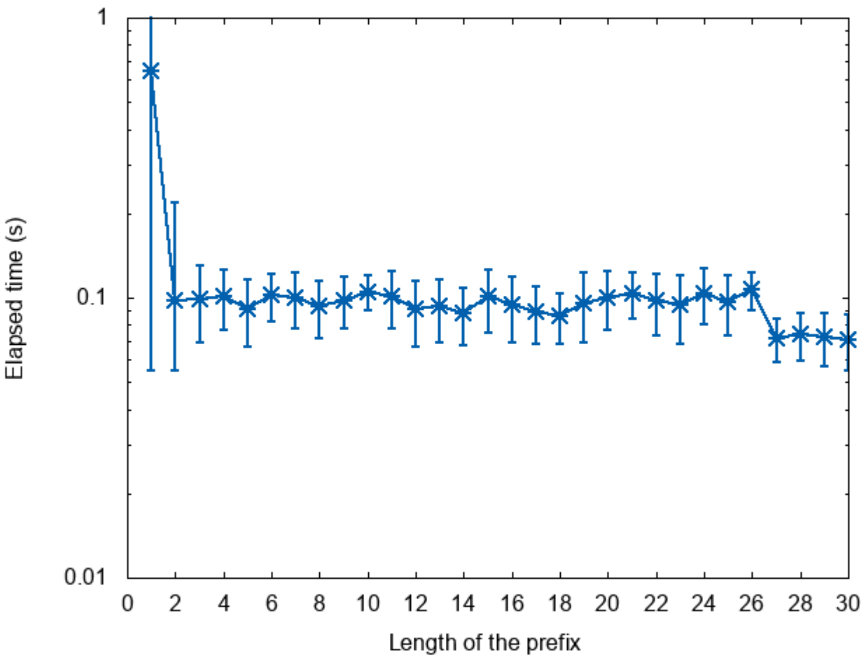

Figure 6.14: Elapsed time according to prefix length.

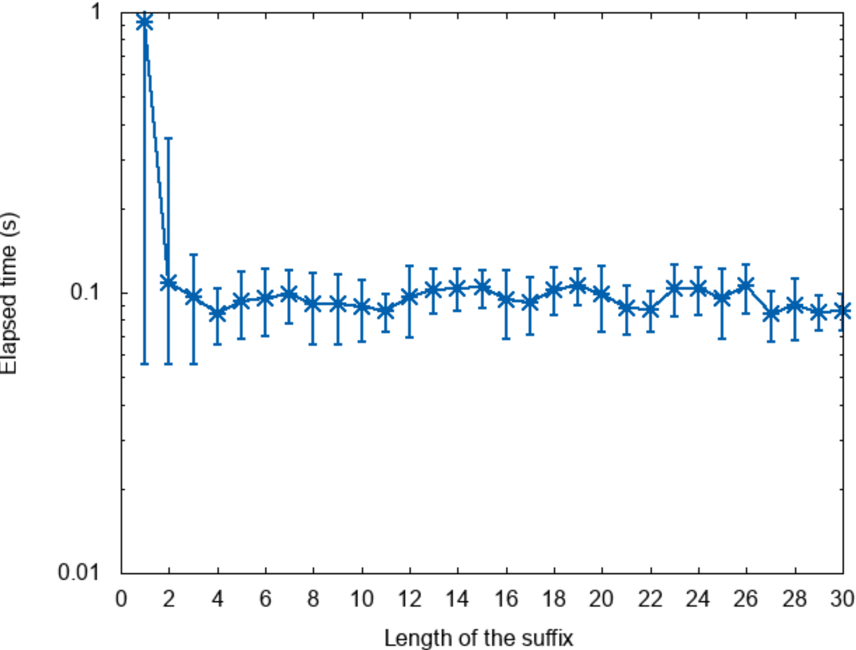

Figure 6.15: Elapsed time according to suffix length.

If we take a look at Figures 6.16 and 6.17, we can observe again that the plot shape is very similar to the one found on the precedent chapter, but with a higher order of magnitude on the y axis.

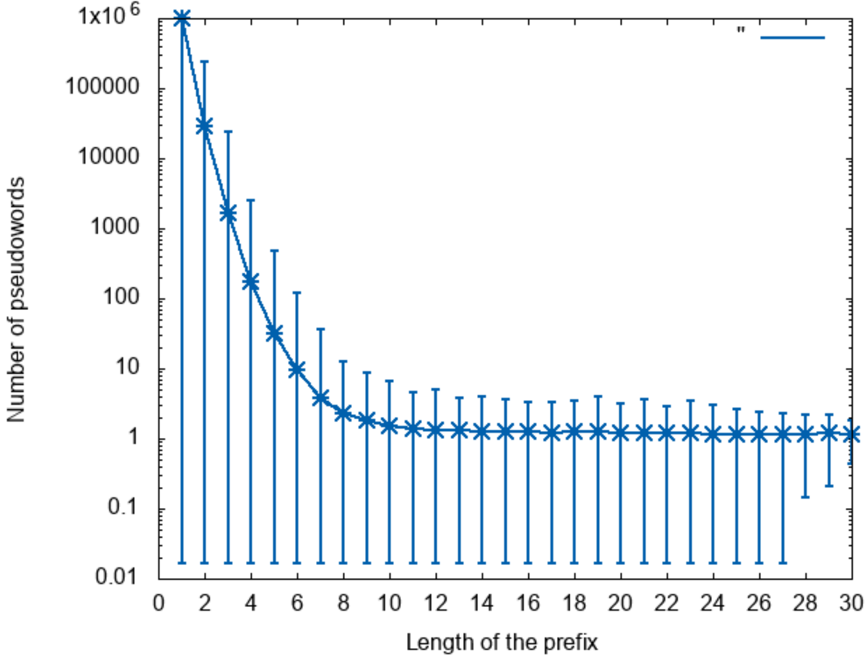

Figure 6.16: Number of pseudowords retrieved according to prefix length.

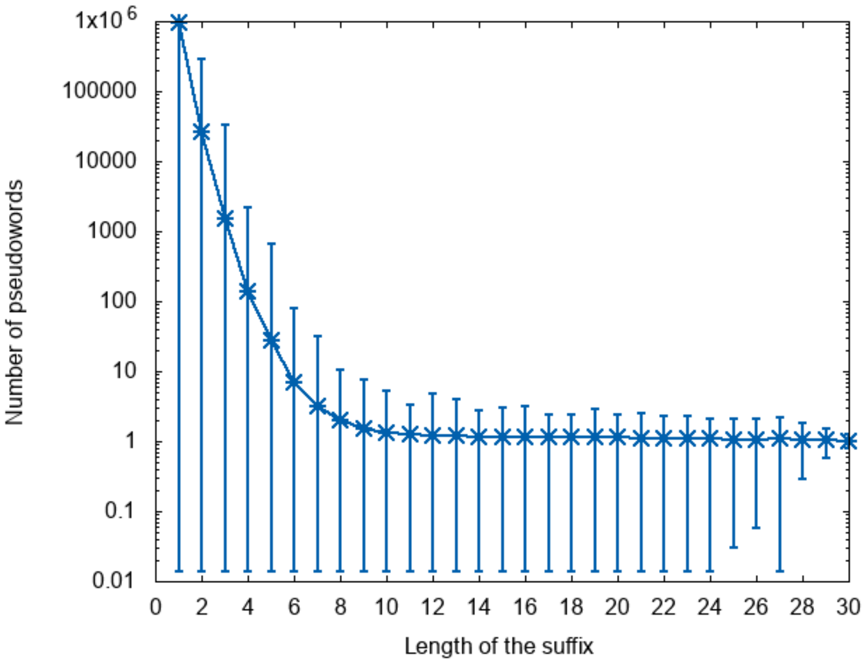

Figure 6.17: Number of pseudowords retrieved according to suffix length.

6.2.2. Approximate string search

Firstly, the hard disk space usage of the DAWG is 169.3 MB and the RAM usage is 3.07GB. Again, we can affirm that both memory usages are very reasonable, taking into account the number of strings that are stored there.

Secondly, if we take a look at the Figure 6.18, we can observe that it has the almost the same shape than the plot presented in the previous section, as it is smoother. Also, in Figure 6.19 we can observe that despite the increase of the lexicon size, we still have very reasonable response times for queries allowing 1 or 2 errors.

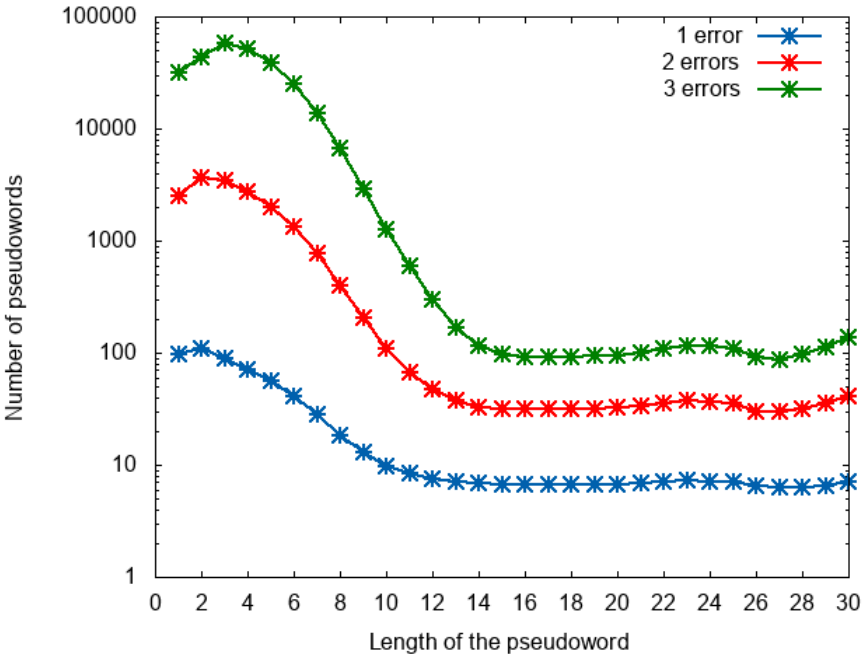

Figure 6.18: Number of pseudowords according to query length.

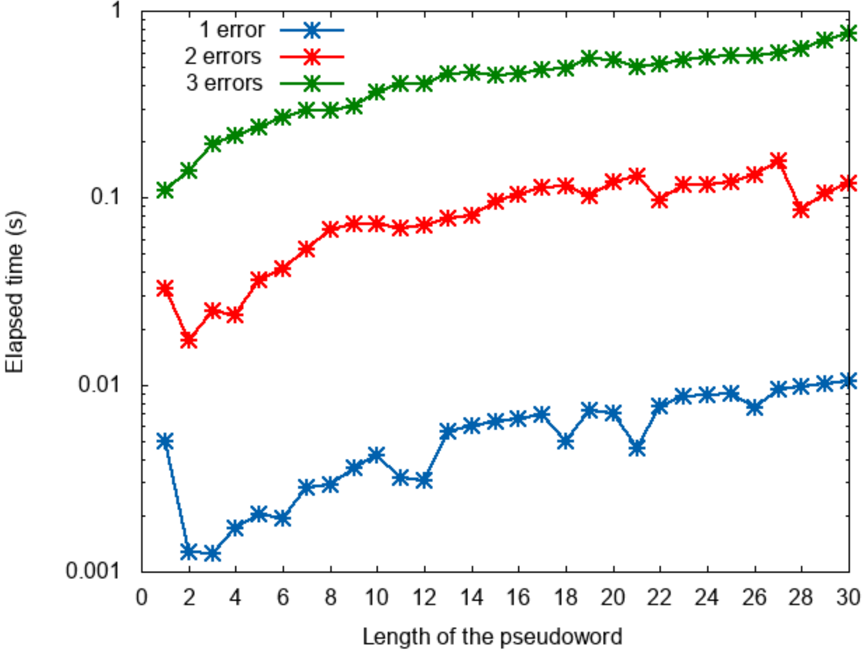

Figure 6.19: Elapsed time on performing a query according to query length.

CHAPTER 7
Conclusions

At the beginning of this project, we had three main objectives: The first one was developing the wildcard and approximate string search in the existing search engine. The second one was to able to perform these queries in a reasonable time with a reasonable memory usage. Finally, the third one was to measure the improvement of approximate search over regular search in the system.

For the first objective, we have made an empirical research of different alternatives that satisfy it. Also, we had to modify the query analysis part of the search engine, in order to convert the approximate query into an OR query formed by the words to which the query has been expanded.

For the second objective, we have chosen the techniques that gave us the better performance in terms of time and memory consumption.

Finally, we have measured how the approximate string search can improve the retrieved results in some cases, as for example when the recall is low.

7.1 Future work

As possible future works for this project we find:

- Measure the impact of the wildcard search in the system over the regular search.
- Implement other approximate search operators, like the "?", which matches 0 or one characters in that position, or the "+", which matches 1 character in that position.
- Change the current search engine by another commercial search engine, as for example SOLR.
- Try other edit distance metrics, as for example the Damerau-Levenshtein distance [5], which considers the transposition of two adjacent characters as one operation.
- Try to calculate custom weights for each edit operation, in order to improve the retrieved results of the system [6].
- Try to estimate the probability of expasion of a query into a concrete word. This could be done continuing with the overlap between queries mesaure that has been proposed in this project. Other alternatives have been proposed in [7] .
- Try other algorithms that take into account phonetics. This could be done using the Methaphone algorithm [8].

7.2 Degree relationship

In this project, I have had the opportunity of practising concepts that I have studied during the degree and the specialization that I have chosen.

Firstly, as my work belongs to a search engine, I have found useful many concepts that I learnt at "Information storage and retrieval systems".

Secondly, the main chapters of the project are closely related to "Data Structures and Algorithms" and "Algorithmics". In these subjects, I have learnt the basis for understanding data structures and algorithms in general, as well as how to measure them and how to choose which is the optimal for each situation.

7.3 Transversal competences

Finally, during the realisation of this project, I have improved on several transversal competences:

- "Comprehension and integration", as I had to understand how the parts that form the search system interacted between them, in order to be able to develop and implement the approximate search in the engine.
- "Application and practical thinking", as I have used quality information sources and cited them properly. Also, I have clearly defined the objectives that we wanted to achieve doing this project.
- "Analyzing and solving problems", as I have tested several techniques in order to determine which is the best one.
- "Ethical, environmental and professional responsibility", as I had to check that all the employed libraries are under a license that allows their free use.
- "Life-long learning", as I had to search different techniques to perform approximate search.
- "Effective communication", as I had to write properly this thesis in English, using the appropriate vocabulary and expressions of a technical academic work.
- "Specific tools", as I had to learn how to code in C++ because it was the language that was used on the existing search engine.

Bibliography

[1] YATA, Susumu. Dictionary compression by nesting prefix/patricia tries. *Proc. 17th Meeting of the Association for Natural Language*. 2011.

[2] BURKHARD, W. A. and KELLER, R. M. Some approaches to best-match file searching. *Communications of the ACM.* 1 April 1973. Vol. 16, no. 4, p. 230–236.

[3] WAGNER, Robert A. and FISCHER, Michael J. The String-to-String Correction Problem. *Journal of the ACM (JACM)*. January 1974. Vol. 21, no. 1, p. 168–173.

[4] S. Hanov. Fast and Easy Levenshtein distance using a Trie, 2011. http://stevehanov.ca/blog/index.php?id=114.

[5] DAMERAU, Fred J. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 1964, vol. 7, no 3, p. 171-176.

[6] HAUSER, Andreas W.; SCHULZ, Klaus U. Unsupervised learning of edit distance weights for retrieving historical spelling variations. *Proceedings of the First Workshop on Finite-State Techniques and Approximate Search.* 2007. p. 1-6.

[7] PUIGCERVER, Joan; TOSELLI, Alejandro H.; VIDAL, Enrique. Querying out-of-vocabulary words in lexicon-based keyword spotting. *Neural Computing and Applications*, 2017, vol. 28, no 9, p. 2373-2382.

[8] PHILIPS, Lawrence. Hanging on the metaphone. *Computer Language*, 1990, vol. 7, no 12, p. 39-43.

[9] TRE — The free and portable approximate regex matching library. https://laurikari.net/tre/

[10] MARISA trie. https://github.com/s-yata/marisa-trie

[11] BK-Tree library. https://github.com/oliversno/BK-Tree

[12] DAWG-levenshtein. https://github.com/brmson/dawg-levenshtein