Jorge André Nogueira da Costa
M.Sc

Time and Space Efficient Data Structures for Supporting Machine Translation Tasks

Dissertação para obtenção do Grau de Doutor em Informática

Orientador: José Gabriel Pereira Lopes, Investigador Principal, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa
Co-orientador: Luís Manuel Silveira Russo, Professor Auxiliar, Instituto Superior Técnico – Universidade de Lisboa

Júri:

Presidente: Prof. Doutor Pedro Manuel Corrêa Calvente Barahona
Arguentes: Prof. Doutora Maria Andrea Rodríguez-Tastets
Prof. Doutor Miguel Ángel Martínez Prieto

Vogais: Prof. Doutora Nieves Rodríguez Brisaboa
Prof. Doutor Pedro Abílio Duarte de Medeiros
Prof. Doutor Margarida Paula Neves Mamede
Prof. Doutor José Gabriel Pereira Lopes

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro de 2017

Time and Space Efficient Data Structures for Supporting Machine Translation Tasks

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my three pillars: my parents and Diana

ACKNOWLEDGEMENTS

First, I would like to express my gratitude to my supervisors, Dr. Gabriel Pereira Lopes from Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, and Dr. Luís Manuel Silveira Russo from Instituto Superior Técnico da Universidade de Lisboa. Thank you for all the guidance, insight and help during this research. Our innumerous conversations made me learn a lot, not only in knowledge, but also as a person, and your incredible availability for listening to my concerns and to answer my questions is something that I will always be grateful for.

I also want to express my gratitude to the members of my PhD comittee, Dr. Nieves Brisaboa and Dr. Margarida Mamede, for all the valuable insight they gave me during my research. A special thanks to Dr. Brisaboa for welcoming me in the Laboratorio de Bases de Datos of Universidade da Coruña, to work with amazing people and learn a lot of important information that helped me in the following years of research.

A very warm thank you as well to my colleagues in NOVA-LINCS research center, not only for all the discussions and advises that helped me get out solve problems in my work, but also for all the fun moments we spent together, during and after work. A special thank you to Luís Gomes, for all the knowledge he shared with me, and for all the help in giving me the context that I needed to continue my work. Also, a special thanks to the amazing people that I met like Filipa Peleja and João Casteleiro, and some old friends from past battles, Miguel Domingues, Miguel Lourenço, Tiago Santos and Ricardo Silva, for their crucial support during this research.

I would also like to thank my colleagues and managers in OutSystems, for all the support and understanding in these last two years. Thanks for covering me and always getting my back when I had to leave earlier or get in later, for the purpose of this research.

Finally, I would like to show all my love and appreciation for the three most important people in my life. First, to my mother, the fiercest warrior I have ever seen, and a great source of inspiration for me. Thank you for everything that you taught me and for all the support that you gave me, even when you were going through the hardest battle one person can fight. I would love that you were here to see me finish this huge endeavor, but wherever you are, I hope you feel proud of what I achieved. Second, to my father, which is a role model for me. Thanks for all the support, in particular during a very dark and hard period of our lives and for helping me keep me balanced and focused, when everything seemed to be falling apart. Last, but not least, my love and appreciation to my beloved Diana. Thank you for all the love, patience and support, in particular when everything seemed to be falling apart. You make me a better person everyday and without you, it would be much more difficult to finish this thesis.

ABSTRACT

The amount of digital natural language text collections available nowadays is huge and it has been growing at an exponential rate. All this information can be easily accessed by individuals of several nationalities and cultures. This leads to the development of new and innovative techniques and tools, for processing and indexing these texts, in fields of research such as Machine Translation, Natural Language Processing or Cross-Language Information Retrieval. Over the years, a lot of important work has been developed, using efficient data structures, such as suffix arrays, for fast pattern matching and to determine statistics. However, these data structures require a considerable amount of space, around four times the text size, which is a problem considering the amount of bilingual texts available in so many languages.

This thesis proposal introduces a two-layer bilingual framework based on compact data structures, for indexing parallel texts, translation memories and bilingual lexica, and their alignments, in pairs of two different languages. Besides a word-based suffix array implementation, this thesis proposal presents a solution based on two byte-codes wavelet trees, one for each text, and bitmaps to represent the alignment. Additionally, it introduces a skip-based bilingual search procedure that speeds up the search time response of the framework, for operations over pairs of word, multi-word or discontiguous phrases.

For indexing and querying over aligned parallel corpora, the bilingual framework presents a space consumption around 50% of the alignment-annotated corpora size, against the 160% of the non compressed approach. In terms of search time response, the compressed approach is slower than the one based on suffix arrays as expected. The skip-based bilingual search procedure improves the time response from the original bilingual search algorithm from 1.6x to 2.3x in average. With such space requirements, the framework is able to represent huge amounts of data in main memory, avoiding the considerably slower disk accesses, and to support tasks such as translation, text alignment, word-sense disambiguation or context analysis.

Keywords: Bilingual texts, Byte-codes Wavelet Tree, Suffix Array, Machine Translation, Bilingual Framework, bilingual search, hierarchical phrases, compression.

RESUMO

Hoje em dia, a quantidade de texto em língua natural em suporte digital é enorme e continua a crescer exponencialmente. Esta quantidade de informação pode ser facilmente acedida por pessoas de várias nacionalidades e culturas. Isto levou ao desenvolvimento de novas e inovadoras tecnologias de processamento e indexação de texto, em áreas como a Tradução Automática, Processamento de Língua Natural e Extracção de Informação Multilingue. Ao longo dos anos, foram desenvolvidos projectos importantes, usando estruturas de dados eficientes, como os arrays de sufixos, para pesquisa de padrões e para cálculo de estatísticas. No entanto, estes índices requerem uma representação de espaço considerável, cerca de quatro vezes o tamanho do texto, o que é um problema dada a quantidade de texto disponível em tantas línguas.

Esta proposta de tese introduz uma estrutura bilingue baseada em índices compactos, para representar textos paralelos, memórias de tradução ou léxicos bilingues, e o respectivo alinhamento, em pares de duas línguas distintas. Além de uma implementação usando arrays de sufixos à palavra, esta proposta de tese apresenta uma solução baseada em duas árvores wavelet codificadas ao byte, uma por língua, e *bitmaps* para representar o alinhamento. Adicionalmente, esta proposta introduz um procedimento de pesquisa bilingue baseada em saltos, que acelera o tempo de resposta da estrutura, com suporte para operações de *localização*, $contagem$ e *extracção*, para pares de palavras, sub-frases ou sub-frases discontíguas.

Para indexar e pesquisar sobre texto paralelo alinhado, a estrutura bilingue requer por volta de 50% do tamanho dos textos alinhados, em termos espaço, contra os 160% obtidos com a abordagem não comprimida. Em termos de tempos de resposta, a abordagem comprimida é mais lenta que a abordagem baseada em arrays de sufixos, o que seria expectável. O algoritmo de pesquisa baseado em saltos melhoram o tempo de resposta da estrutura bilingue, desde 1.6x a 2.3x em média. Com estes requisitos de espaço, a estrutura permite representar enormes quantidades de texto em memória, evitando o acesso a disco consideravelmente mais lento, e suportar funcionalidades como tradução, alinhamento, desambiguação ou análise de contexto.

Palavras-chave: Textos bilingues, Árvores Wavelet codificada ao Byte, Arrays de Sufixos, tradução automática, Framework Bilingue, pesquisa bilingue, frases hierárquicas, compressão.

CONTENTS

LIST OF FIGURES

List of Tables

LISTINGS

CHAPTER 1

INTRODUCTION

1.1 Research Statement

This thesis presents the details and results about the research work I developed, on compact data structures for supporting space and time demanding Machine Translation and Natural Language Processing applications. The main objective of this research is to develop a Bilingual Framework to index aligned parallel texts (pairs of texts that are translations of each other) in main memory, for any language pairs, while offering efficient monolingual and bilingual searches over the indexed corpora.

This Framework is not specifically intended to do Machine Translation directly. It is integrated in an interactive workflow of phrase alignment of parallel collections of sentences, extraction of new translation pairs of phrases and human validation of translation candidates. This workflow gathers vital information and continuously learns from it, enabling machine-made translations with higher quality. The functionalities of the proposed Bilingual Framework enable determining statistics and extract contextual information that is relevant for supporting several tasks in this workflow and others, such as training translation engines, computer-assisted translation, word sense disambiguation, etc.

This research builds on previous work using non compact data structures, which resulted in three publications [36–38], including my Masters thesis [36], and resulted in two articles [39, 40], published as book chapters.

1.2 Research Context and Motivation

Translation (man or machine made) is an important mechanism that allows reaching information from a plethora of sources and cultures, available in languages different from our native one, in an easier way for our understanding. Apart from the translation quality, one of the

bigger problems nowadays derives from globalization, requiring translations to be made from any language to any other language. Moreover, more and more translations are needed at an increasing faster rate. To satisfy these translation needs, only machines can help.

In this vein, Machine Translation (MT) is an area that uses software tools to translate, without human intervention, texts from one language to another. Since the fifties [180], MT evolved considerably, in particular with the rise of Statistical Machine Translation. Its machine learning approach to build translation models, based on statistical techniques, enabled machine-made translations of higher quality. A translation model $\mathscr{P}(f|e)$ is the probability of the sentence f to be translated by the sentence e. A conventional representation of a statistical translation model, using for instance a data structure to index the texts that allow building the model in main memory, can be considerably space demanding. With such high space requirements, the size of training data would be heavily restricted, which is a serious limitation to its scalability.

Callison-Burch et. al [28] and Zhang and Vogel [189] introduced an alternative approach that relies on computing the parameters for MT on demand, storing only the training data instead of the whole models. This training data can be sentence aligned parallel corpora, which is two collections of texts in two different languages, with the translation correspondences defined at the sentence level. Representing the mentioned training data reduced the space requirements of MT tasks, making them more dependent on fast pattern matching functionalities.

Besides the Machine Translation task itself, there are other MT related tasks that benefit from these fast pattern matching functionalities, over parallel corpora. Tasks such as extracting and classifying as correct or incorrect new translation pairs [100], computer-assisted translation tools and building bilingual lexica, draw crucial information from parallel corpora. Even outside Machine Translation, other NLP problems such as word-sense disambiguation [128, 168], can use parallel corpora as a source.

Despite its many merits, these approaches only considered phrase-based models, which have some limitations such as the inability of learning reorderings of the phrases that can occur, when translating from one language to another. Later, Chiang [31, 32] introduced a statistical translation approach based on discontiguous phrases to tackle these limitations. On top of this approach, Lopez implemented the extraction of discontiguous translation rules [116], with the same data structure used by Callison-Burch and Zhang and Vogel to support it.

The mentioned solutions are based on Suffix Arrays. Alongside Suffix Trees, these structures are the most important data structures in string matching problems, including fields of research as Bioinformatics, Natural Language Processing (NLP) and Machine Translation. Both data structures fully index a text and all its substrings, meaning that all characters of the text are indexed, and provide fast string matching. The many virtues of Suffix Trees were described by Apostolico [8] and Gusfield [82] and are of particular importance in bioinformatics, namely for analyzing long strings of DNA and proteins.

However, Suffix Trees have space requirements around 10 to 20 times the text size. For example, storing the human genome requires about 700 Megabytes, while a Suffix Tree would require at least 40 Gigabytes to index it. Suffix Arrays [125] simulate the behavior of Suffix Trees, with less space requirements and slower, but still efficient, string matching response. These

characteristics made these structures often used in Information Retrieval [13, 27, 116, 117, 183], but their space requirements are still four times the text size.

1.2.1 Space Problem

There are two alternatives to support pattern matching: 1) sequential pattern matching, which transverses the text sequentially to determine every occurrence of the pattern; 2) indexed pattern matching, which builds a data structure (or index) of the text that allows searching for the pattern without traversing the whole text. As sequential string matching easily becomes impractical for large texts, and the source training data do not change frequently that would lead to expensive costs on maintaining the index. Using adequate data structures is indeed the best choice.

Considering the advantages of using text indexing, with data structures such as Suffix Arrays and Suffix Trees, it is necessary to have enough main memory storage that can maintain and support fast access to the indexed text. However, main memory is not unlimited and, in comparison with secondary storage, its sizes are considerably reduced and more expensive. On the other hand, secondary storage is highly and easily available, but disk access is considerably slower than main memory access [141].

The amount of digital textual information has been rising at an exponential rate. A considerable part of this information consists of natural language text collections, which in 2002 it was estimated that exceeded 30 to 40 times the amount of printed material in history. Considering the collections of texts with millions of words each, in multiple languages, and the space requirements of Suffix Trees and Suffix Arrays, the main memory can be too short for indexing the texts using these techniques.

To add to that, the work described in this thesis is inserted in a context of an interactive workflow of translation alignment and extraction, that builds iteratively a knowledge base to be used in the MT task [4]. This accumulated data is based on parallel text alignment and bilingual lexica, which are sets of pairs of word and multi-word segments in two languages that are correct translations of each other. This workflow includes human validation [69], which contrasts with the more common and fully automatic unsupervised Statistical MT procedures. However, it combines the best features of the machine learning techniques, with the insightful knowledge that linguistics can add to the workflow. The human interaction allows accumulating information to iteratively support further alignments and extractions, resulting in translations with better quality [4], in comparison with Moses Statistical MT system [107].

Considering all the bilingual textual data, or bitexts[1], available for these tasks, from parallel corpora, to bilingual lexica, tackling the high space requirements of MT tasks is the main concern surrounding this research. To solve this space problem, without affecting too much the time response, I describe in this thesis the study I developed on compact data structures, for indexing bilingual textual data in main memory, without requiring disk access, and with direct application in the aforementioned tasks.

[1] Segments of text in two different languages and the translation correspondences between those segments.

1.2.2 Reducing the space

Since the efficiency of Suffix Arrays and Suffix Trees is well know for string matching problems, the first approach in this research was to use compressed version of such structures, Compressed Suffix Arrays and Compressed Suffix Trees [58, 59, 156, 157, 159]. These approaches reduce considerably the space requirements of these structures, with the side effect of having a slowdown on the search time response, since the data is represented in its compressed form.

These compressed data structures are also full-text indexes, just like their uncompressed approaches, meaning that every position of the text is indexed in main memory. However, since the purpose of this research is focused on natural language, indexing every position of the text, may be too much, since in most cases the main unit of any language is the word. This way, we shifted the focus of the research from the character-based indexes, to the word- based data structures, maintaining the purpose of reducing the space requirements, for indexing such amount of textual information.

Inverted indices [12] have been effectively used in indexing natural language text over the years, with focus on the support for Web search engines. These indices are based on a vocabulary, with all the content words in the texts, and a list of the positions in the text where each word in the vocabulary appears. However, searching for multi-word terms require intersections of the posting lists to find the final occurrences, which despite several proposals for improving this matter [11, 14, 163], is not an ideal solution.

Word-based versions of text indices like Suffix Arrays, were important for Information Retrieval [182] and string matching [6], reducing space consumption, by building an index for every word instead of any position in the texts. The Word-based Suffix Array by Ferragina and Fischer [55] required space around $O(n)$ plus the text, with n as the number of words in the text. However, the text still needs to be fully-indexed alongside the index itself, which can still maintain the space problem.

Text compression is an area of research on the rise, where word-based compression techniques [131] achieve compressions ratios of 25%-35% of the text size, while allowing searches up to eight times faster over the compressed text, when compared with the original text [134, 175]. Brisaboa et. al. [22] proposed a reorganization of the bytes in the codified words of the compressed text, following an index-like structure. This reorganization enables search times proportional to the text length, as in typical indexing techniques. With the same amount of space, this reorganization is even more efficient than compressed inverted indices.

The word-based approach was also extended for compressed self-indices [140], with interesting results. A compressed self-index fully-indexes a text without needing it in main memory, and with space requirements proportional to its empirical entropy [126]. The index alone is sufficient for supporting searches and text access. Brisaboa et. al. [23] proposed a self-index based on a word-based layer on top of a Compressed Suffix Array [60], occupying around 35-45% of the text size, and being more efficient than inverted indices.

Any of these compressed alternatives have an access time slowdown when compared to uncompressed data structures. On the other hand, considering the difference between main

and secondary memory access times, it is more beneficial to have large texts indexed in main memory, in compressed form, to reduce and even avoid disk processing, transmission and transfer time [140].

Considering all this information, in this thesis I introduce a Bilingual Framework based on compressed word-based indexes, and additional structures, to index and access bilingual textual information, in main memory, for supporting Statistical Machine Translation methodologies and tasks. These bilingual texts can go from sentence aligned parallel texts, phrase tables or bilingual lexica. So, the main purpose of the Framework is not doing translation per se, but actually offer functionalities over these bilingual texts, to determine statistics that can be relevant to a myriad of MT related tasks.

In that subject, for instance translation models require, among other information, calculating the bilingual co-occurrence frequencies to, in turn, determine translation probabilities and extract translation tables. Language models require determining monolingual occurrences of word and multi-word terms, that appear prior or after other single or multi-word segments. Other applications such as computer-assisted translation tools, require fetching text surrounding an occurrence, for context analysis or word-sense disambiguations [30].

Taking these necessities into consideration, by indexing bilingual texts in main memory, the Bilingual Framework supports MT-related functionalities such as: 1) counting and locating the occurrences of a word or multi-word term, including discontiguous phrases, in one language (monolingual search); 2) counting and locating the co-occurrences of a pair of word or multi-word terms, including discontiguous phrases, in both languages (bilingual search); 3) context extraction of segments of the indexed text, surrounding an occurrence or co-occurrence[2]

1.3 Research Objectives

The broad objective of the research introduced in this document lies on investigating space and time efficient techniques, using compact data structures, to apply them directly in important and relevant NLP and MT problems. Such compressed indexes and techniques enable the support for a space and time efficient Bilingual Framework, for indexing and accessing bilingual textual information in main memory.

Considering the space requirements that indexing this textual information may have, it is crucial to determine the data structures that offer the best space / time tradeoff, for supporting fast access to the data, while maintaining the indices fully represented in main memory. This first objective that I propose to accomplish can be summarized as follows:

> *Objective 1: Develop a compact Bilingual Framework based on word-based compact techniques, for representing bilingual texts in any language pairs, in main memory. The Framework is bound for supporting MT tasks, with functionalities like monolingual and bilingual searches, and also context extraction, with an efficient space / time performance.*

[2] In this document, context extraction refers to fetching the surrounding context of an occurrence in a text. Do not confuse this with the extraction of new translation pairs or candidates, supported by statistics that can also be fetched from the proposed Bilingual Framework.

Reducing the space requirements of the representations of bilingual texts, brings aboard a slowdown directly related to using compressed text and compressed indexes. However, natural language is very rich and even some common difficulties can be transformed into strengths. When dealing with translations from one language to another, there is ambiguity. For instance, the word `plant` can be translated to Portuguese in several different ways, with different meanings, such as `planta` or `fábrica`. This means that in a bilingual search for the pair of terms `plant` <-> `planta`, over aligned parallel corpora for instance, not all the occurrences of `plant` are relevant to determine co-occurrences with `planta`. The same occurs for the pair `plant` <-> `fábrica`. This led me to continue the research with a second major objective for this thesis, that is described as follows:

> *Objective 2: Improve the bilingual search time response of the Bilingual Framework, by using the correspondence between segments to filter non relevant occurrences, that result from ambiguity in translations.*

Up to this point, the Bilingual Framework would only support phrase-based searches, meaning that all the searched terms would be contiguous phrases. However, Hierarchical Machine Translation and its discontiguous phrases with gaps tackled some of the limitations of the Phrase-based Machine Translation [32]. This approach proved to be extremely important for improving machine-made translation quality. Therefore, on the third goal for this research, I propose to extend the functionalities of the Framework for supporting Hierarchical Machine Translation related tasks:

> *Objective 3: Add the support for matching discontiguous phrases to all the functionalities in the Bilingual Framework proposed in Objective 1.*

These objectives led to a deep study of compressed techniques and indexes, as well as algoritms that would improve as most as possible the search time response of the Bilingual Framework. The next Section gives an overview of the organization of this document, as well a slight introduction to the fulfillment of these objectives and the contributions of the research work carried out.

1.4 Research Steps and Contributions

The research developed in the context of this thesis integrates well with the main idea guiding the wide research carried out in the research group where this work is inserted, and it is organized in the following main parts.

1. Develop a Word-based Bilingual Framework for indexing and querying over monotonically aligned parallel corpora in main memory, using compact data structures.

2. Speedup the search time response of the framework, by using the underlying ambiguity in translating a text from one language to another.

3. Support the location of phrases with gaps, such as in Hierarchical Machine Translation, over the compact data structures that support the Framework.

4. Develop a Bilingual Concordancer, a Computer-Aided Translation tool frequently used by human translators, supported by the proposed Bilingual Framework.

The starting point to meet *Objective 1* consisted of understanding how the Bilingual Framework would be structured, considering the requirements and functionalities. To represent bilingual textual data, the Framework has two main layers: Text Layer, to represent the corpora, and the Alignment Layer to represent the parallel text alignment. From this point, it was important to determine the best approach for indexing huge amounts of text. For that purpose, the follow-up work consisted of understanding the several compression techniques and compressed data structures available, from compressed versions of common data structures in string matching[156, 159], to compressed self-indices with direct application in NLP [22, 23].

After studying several alternatives, the Byte-codes Wavelet Tree [22] was the data structure chosen to index the bilingual textual data. These indices have small space requirements, around 30% to 35% of the text size, and they have the capability of searching the compressed indexed text in logarithmic time, as typical indexing techniques, beating the popular compressed inverted indices [43, 169]. In addition, they allow a search to start or resume from any point of the text, and its organization follows the order of the words in the text.

To represent the alignment, we chose an implementation based on bitmaps, with one to two bits to represent the correspondences between segments of a text in one language and their respective translations in the other language.

With this representation, the proposed Bilingual Framework requires around 50% of some tested alignment-annotated parallel corpora sizes, proving to be considerably more space efficient than similar alternatives based on Suffix Arrays [27, 117, 189], with a cost of a natural slowdown for having the data compressed. These results prove that using compressed techniques in such MT tasks can enable indexing amounts of bilingual texts that state of the art approaches could not. This step of the research is detailed in **Chapter 3 – Indexing Aligned Parallel Corpora** and it was presented in the paper [39]:

> Jorge Costa, Luís Gomes, Gabriel P. Lopes, Luís M. S. Russo and Nieves R. Brisaboa. Compact and fast indexes for translation related tasks. Progress in Artificial Intelligence. Published by: Springer. EPIA 2013.

With the space problem solved, the second phase of the research centered on alternatives to improve the bilingual search time response of the Framework. For monolingual searches, there is not much to do as it mainly uses the search algorithms of the data structures. However, in a bilingual environment, there is potential for improving the search time response, by taking into account translation ambiguity.

Considering the correspondences between segments represented in the Alignment Layer, the search procedure is able to jump over, or skip, redundant occurrences of the terms in a bilingual search, which it is denominated as skip-based bilingual search. In a first approach, to

determine the co-occurrences, the search procedure intersected the lists of occurrences of both terms, in a bilingual search, with binary searches. With this new strategy, the bilingual search improves its time response by 1.7 times using the compressed indices, and around 2.5 times, using the version supported by Suffix Arrays, when compared against the original procedure. This improvement does not affect the memory performance of the Framework, making it an important alternative for determining statistics over bilingual texts. This second stage of research is detailed in **Chapter 4 – Phrase-Based Search** and in the paper [40]:

> Jorge Costa, Luís Gomes, Gabriel P. Lopes and Luís M. S. Russo. Improving bilingual search performance using compact full-text indices. Computational Linguistics and Intelligent Text Processing. Published by: Springer. CICLing 2015.

With the support for phrase-based searches fully implemented on the proposed Framework, the third stage of the research focused on locating and counting discontiguous phrases in parallel corpora, which is the basis of the Hierarchical Machine Translation Model [32]. Locating phrases with gaps requires a change in the internal search algorithms of the data structures. There are some approaches using Suffix Arrays [13, 116] for determining hierarchical phrase rules, however, to the best of our knowledge, there are no proposals for matching phrases with gaps in compressed indices.

Using an adaptation of techniques based on Suffix Arrays, combined with the skip-based search algorithm and some known dynamic programming algorithms, such as the *Longest Common Subsequence*, the Bilingual Framework is able to support the same functionalities for hierarchical phrases efficiently, without requiring extra memory consumption. The research step is detailed in **Chapter 5 – Hierarchical Phrase-Based Search**.

The Bilingual Framework that results from these three stages of research, with its space / time performance and the functionalities it offers, can be used as support for several MT and cross-language applications. Locating and counting occurrences and co-occurrences of terms in both languages is important for Machine Translation, as it computes some of the parameters for training a translation engine on demand. This is even more important given the iterative alignment and extraction process mentioned before, explained in further detail in **Chapter 2 – Preliminaries**, to contribute for learning and accumulating new knowledge, that later improves the machine-made translation quality. This gathering of textual knowledge can easily lead to large amounts of textual data to be indexed, making the compression techniques, and the space efficient indexing capabilities it leads to, even more important. The same functionalities are important as well for extraction of pairs of sentences, that are correct translations of each other, by drawing statistics from the known knowledge base.

The characteristics of the Bilingual Framework can also be used for word-sense disambiguation applications [30] and computer-assisted translation tools, such as a Bilingual Concordancer [18]. A Bilingual Concordancer is an important context analysis tool for human translators. This is extremely helpful for post-editing machine-made translations [108] and other related tasks, as it allows the user to look at the context of a translation candidate in the parallel corpus from where it was extracted [5].

The fourth and final stage of the research focuses on supporting a web-based Bilingual Concordancer tool using the proposed Bilingual Framework. This tool enables searching for a term in one language, or a pair of terms in both languages, to locate and count their occurrences or co-occurrences in aligned parallel corpora. These terms can be word and multi-word, with support as well for discontiguous phrases. Besides locating and counting the occurrences, the Concordancer tool also shows the occurrences of the terms in the context where they appear in the parallel corpora, with syntax highlighting to emphasize the occurrences and also for translation spotting. This application is introduced in **Chapter 4 – Phrase-Based Search**.

Chapters 3 to 5 also describe the space and time results respectively from the Bilingual Framework and all its functionalities. Then, **Chapter 6 – Related Work** describes some relevant Related Work, that was studied during the research that led to this thesis. This work involves compressed data structures and also important research being done in MT, using structures such as Suffix Arrays or similar structures. During the description of the Related Work, the Chapter includes some discussion about the choices made for the thesis, when compared with the presented alternatives. The conclusions of the research and possible paths for extending it are presented in **Chapter 7 - Conclusions and Future Work**.

CHAPTER 2

PRELIMINARIES

This Chapter introduces some important notions for the scope of the research work carried out and that is described in this thesis. At first, there is an introduction of Phrase-Based and Hierarchical Phrase-Based Machine Translation, that are directly linked to the contiguous and discontiguous phrase-based searches supported by the proposed Bilingual Framework. This information is followed by an overview of the iterative translation alignment and extraction flow, that results from the work developed by the research group where this work is integrated in. This overview includes a more detailed introduction to parallel text alignment, as the bilingual textual data tested to validate this research consists of aligned parallel texts.

2.1 Data Structures

Machine Translation applications are heavily based on text mining and string matching tasks over large amounts of text. String matching consists of finding the occurrences of a small string, the pattern, on a larger string, the text. There are two alternatives to support string matching: 1) sequential string matching, which traverses the text sequentially to determine every occurrence of the pattern; 2) indexed string matching, which builds a data structure (or index) of the text that allows searching for the pattern without going across the whole text. Indexing the text using data structures is the best option when the texts are too long, making the sequential scan impracticable. And when the text does not change frequently, this avoids expensive costs of building or maintaining the indices. These requirements are commonly met in Machine Translation tasks, thus indexing the text to support such tasks is the best alternative.

Full-text indices are data structures that fully index a text T in main memory, while providing fast substring searches over large text collections. In this Section, I introduce Suffix Trees and Suffix Arrays, the most important and used full-text indices in string matching, from research fields from Bioinformatics to Natural Language Processing.

2.1.1 Suffix Trees

A Suffix Tree [127, 181] for a text T, with length n, is a rooted directed tree with n leaves, where each leaf has a suffix of T as path-label, following a left-to-right lexicographical order. A path-label of a node consists of the concatenation of all the edge labels from the root to the node itself. Each internal node has at least two children and each edge is labeled with a non empty substring of T. Every edge out of a node has a label with a different first character. A Suffix Tree fully represents T and all its suffixes.

Figure 2.1 shows an example of a Suffix Tree for the text T = `abracadabra#`, with the leafs represented as squares and numbered following the position where the respective path-label starts in T. The leaf numbered **1** spells out the label `abracadabra#`, which is the suffix of T starting at position **1**, while the leaf numbered **3** has `racadabra#` as label, the suffix of T that starts at position **3**. From the root, one can notice that no edge coming out of it has a label that starts with a same character. At the end of T, there is a unique terminal character to represent the end of the sequence to be indexed. Without the terminal character, as `abra` is also a prefix of `abracadabra`, the path labeled `abra` would not end in a leaf.

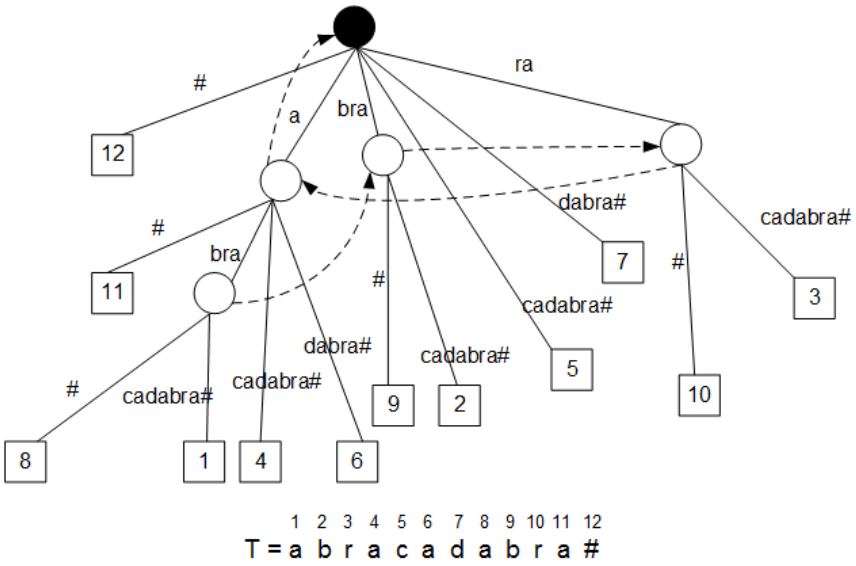

Figure 2.1: Suffix Tree for T =`abracadabra#`

A Suffix Tree can be built in $O(n)$ time [52, 127, 176, 181], while the search time for a pattern P is linear regarding the size of P, $O(|P| + occ)$, where occ is the number of occurrences of pattern P. This is extremely useful for string matching problems, as the time required for matching a pattern in a text depends on the pattern size. For that matter, Suffix Trees are extemely important for many string processing problems, having a myriad of virtues as described by Apostolico [8] and Gusfield [82].

These data structures have been commonly used in bioinformatics [81], supporting tasks that demand analyzing long strings of DNA and proteins efficiently, like retrieving sequences similar to a sample in a gene or protein database, resulting in several important results in the

field. However, using Suffix Trees in bioinformatics also brought to attention the main shortcoming of these structures: their large space requirements. Classical Suffix Trees implementations require $O(n\ log\ n)$ bits, while the indexed string requires $n\ log\ \alpha$ bits, with α as the size of the alphabet. This means that even spaced engineered implementations can be ten (or more) times larger than the indexed text [110], which can be a problem as a Suffix Tree is tailored to be managed in main memory.

The efficiency on building the tree and searching for a pattern in it heavily depends on suffix links. A suffix link is a link between an internal node v and a node w, in which v is labeled `c` α and w is labeled α, where `c` is a single character and α a substring of T. These links allow navigating the edges of the tree, when a mismatch of a character occurs in a string matching task, without requiring going back to the root and try a new path. Following suffix links, we can keep testing for a possibility to descend with the character that originated the mismatch. We only stop this process when the root is reached, or when we find again a match and can continue descending the tree. In Figure 2.1, suffix links are represented by dashed arrows. In the same example, the node labeled `bra` has a link to the node labeled `ra`.

2.1.1.1 Generalized Suffix Trees

A Generalized Suffix Tree is a Suffix Tree for indexing more than one string sequence, T_1, T_2, ..., T_n. Figure 2.2 shows an example of a Generalized Suffix Tree for T_1 =`abracadabra#` and T_2 =`magician$`. The different sequences indexed in the tree are distinguished in the leaf nodes, which are numbered by two integers (i, j), instead of one. Integer i indicates the sequence, while j second indicates the starting position of the path-label in T_i, just as in the Suffix Tree in Figure 2.1. In Figure 2.2, the leaf $(1,2)$ represents the suffix starting of position 2 of T_1, `bracadabra#`, while the leaf $(2,2)$ is labeled by the second suffix of T_2, `agician$`.

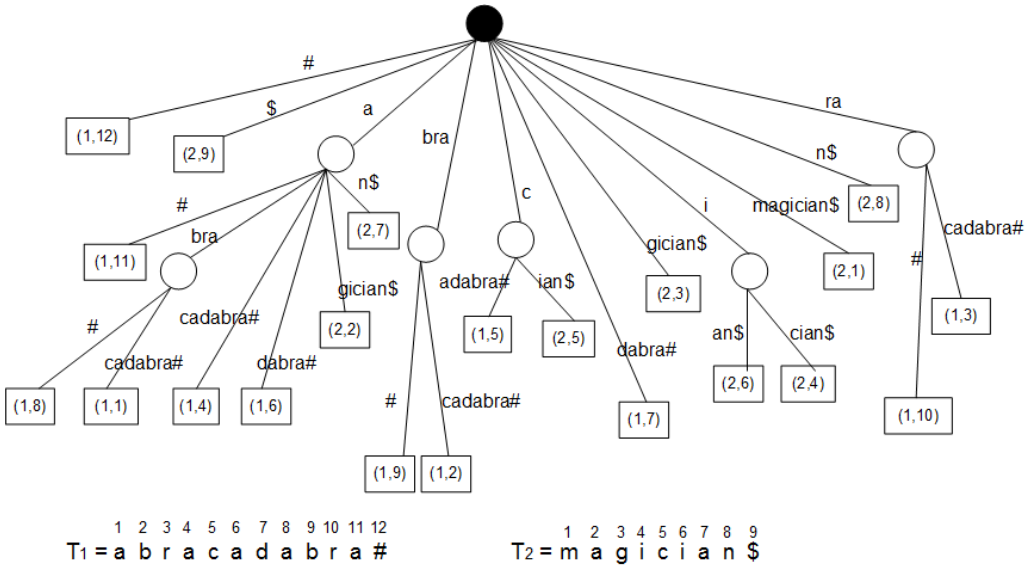

Figure 2.2: Generalized Suffix Tree for T_1 =`abracadabra#` and T_2 =`magician$`

2.1.2 Suffix Arrays

A Suffix Array [125] is a permutation of all the suffixes of a text T, in a way that all the suffixes are lexicographically sorted. More specifically, a Suffix Array $A[1,n]$ contains a permutation of the interval $[1,n]$, where $T_{A[i]} < T_{A[i+1]}$, with $T_{A[i]}$ being the beginning of the suffix of T starting at position $A[i]$ and < indicating a lexicographic order. Figure 2.3 shows an example of a Suffix Array for `abracadabra#`, where each position of the array stores a pointer to the beginning of the respective suffix of T. In this example, $A[3] = 8$ and $T_{A[3]} =$ `abra#`.

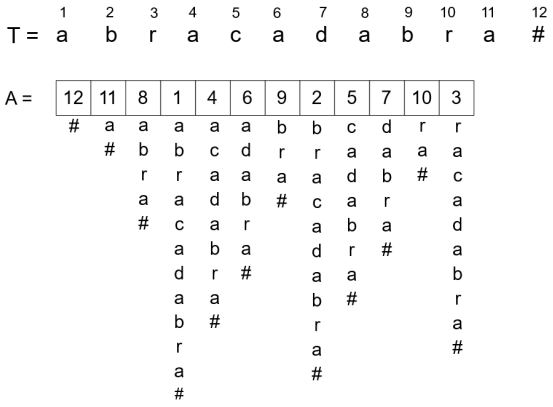

Figure 2.3: Suffix Array for $T =$ `abracadabra#`

A Suffix Array can be built from a Suffix Tree, by collecting the leaves of the tree in a left-to-right order, however it is more practical to build the index directly. Manber and Myers [125] designed the original algorithm to build a Suffix Array in $O(n \log n)$ time. Since then, several algorithms were developped, from non-linear construction algorithms [51, 89, 112, 164], which were considered the best in practice [153], to $O(n)$ time algorithms [93, 96, 102, 103, 145].

Searching for a pattern P of size m in a Suffix Array consists of binary searches, since the suffixes are sorted following a lexicographic order. This results in an interval $A[sp,ep]$, with sp and ep as the pointers to the first and last occurrences of P in A respectively. Every step of the binary search demands a comparison between P and the text starting at the position indicated by the Suffix Array, thus requiring access to the original text, and taking $O(m \log n)$ time.

Suffix Arrays are considerably more space efficient than Suffix Trees and for that matter have been frequently used in Information Retrieval tasks, like finding frequent words in natural language texts. However, they still require a high amount of space in main memory. The size of the Suffix Array is equal to the length of the indexed text, and each position of the array stores pointers to positions of the text, resulting in a space consumption of four times the text size. In addition, a Suffix Array requires the explicit storage of the text, to be accessed during the searches, as every position of the array holds a pointer to the text.

2.1.3 Wavelet Trees

A Wavelet Tree is a balanced search tree where each unique symbol of the text corresponds to a leaf of the tree. Every node in each level of the tree holds a bitmap, with 1s representing the

symbols that descend to its right child and 0s the symbols that descend to the left. Wavelet Trees play a significant role in compressed full-text indexing, due to its bitmap-based representation. Figure 2.4 shows an example of a Wavelet Tree for `abracadabra$`, where the text is just shown for visualization purposes.

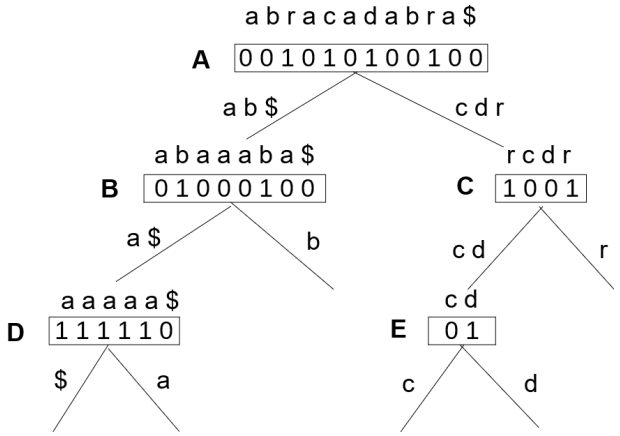

Figure 2.4: Wavelet Tree for `abracadabra$`

The alphabet σ for `abracadabra$` consists of six characters, with $\sigma = \{\$abcdr\}$, following a lexicographic order. It is this lexicographic order of the alphabet that influences the descend to the left and descend to the right in a Wavelet Tree, mentioned above. Picking the example in the Figure, the root node of the tree represents the entire expression `abracadabra$`, with the respective bitmap 001010100100. The 1s mark the positions of the symbols `c`, `d` and `r`, which are the ones on the right half of the sorted alphabet. On the other hand, `$`, `a` and `b` are represented with 0s for belonging to the left half of the alphabet. Continuing this logic, the left child of the Wavelet Tree consists of a representation of `abaaaba$`, the concatenation of all the characters marked as 0 in the previous bitmap, in the root. The right child has a representation of the sequence `rcdr`, with the same logic but for the characters marked as 1.

The structure of the tree follows the same strategy for the following levels. The inner nodes also have bitmaps to represent the sequence in the respective node, following as well the logic to divide the symbols in half for the next level. Going back to the example, node **B** has the bitmap 01000100, as `b` falls to the second half of the alphabet $\sigma = \{\$ab\}$. Following the same logic as before, the left child (**D**) has a representation of `aaaaa$`. The right child, since it only represents a character, is a leaf of the tree and has no more bitmaps. This process is followed recursively until the leaves, i.e. when only a character is left in each iteration, on every branch of the tree.

With such a representation, Wavelet Trees are succinct data structures, that enable representing the text in a reduced space. That happens because the representation based on bitmaps reduce the dependence of the index on the alphabet, from σn to $n \log \sigma$ [78], when compared to a Suffix Array. However, having the representation based on bitmaps, there are some important operations that are relevant to extract information from the index. These binary sequences

are in fact the base of many succinct data structures [140], thus it is important to have some efficient operations to extract information from them, when there is a representation fully based on bits 0s and 1s. These operations are the $rank$ and $select$.

2.1.4 Rank and Select

Given an offset i in a bitmap, $rank_{0/1}(bmap, i)$ counts the number of bits 0 / 1, until position i, while $select_{0/1}(bmap, i)$ determines the position in the bitmap where the i-th occurrence of the bit appears.

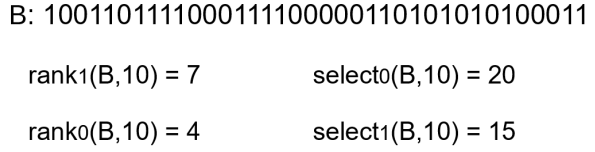

Figure 2.5: *Rank* and *select* examples in a bitmap

These apparently simple operations have been subject of several studies, with the objective of finding efficient strategies to compute them, due to their heavy importance in full-text indices [140] and succinct data structures [124].

These operations were defined in the work of Jacobson [90], alongside an implementation of $rank$ in constant time, using binary trees. The implementation of this approach falls outside the context of this work, but it is strongly based on storing $rank$ results in a two-level structure. The first level stores $rank(i)$ for every i multiple of $\lceil logn \rceil^2$, with n being the size of the bit sequence. The second level stores $rank'(j)$ for every j multiple of $\lceil logn \rceil$, computing $rank$ for subsequences of size $\lceil logn \rceil^2$. The bits without $rank$ information are used as an index, with the number of times bits 1 and 0 appear in the respective subsequence up to that bit. The final result is obtained using table lookups, leading to $rank$ in constant time. However, with this approach, $select$ is computed in $O(loglogn)$, as it is based on binary searches over the table, while the space requirements are $O(n)$.

Several studies followed this one [34, 135], improving the solution and reaching constant time for both $rank$ and $select$ operations, based on additional data structures, just like the original proposed solution [90]. These approaches did not consider the number of bits of the bitmaps, and how many 1s and 0s, and also the statistical properties of the sequence, for instance the positions of the bits 1 in the bitmap. By adding more data structures, the space requirements of these solutions were also higher.

Later, other approaches emerged that focused more on working with compressed binary sequences, which considered then the statistical properties of the bitmaps [150, 155]. Both approaches were based on using blocks, where each block had the number of bits 1 it contains, and an identifier of the block to determine its placement in the sequence. The first approach, by Pagh [150], introduced a first compressed bitmap that supported more functionalities than just accessing the bits, such as the $rank$ function. The second approach, by Raman et. al. [155],

was actually the first one that supported $rank$ and $select$ in constant time for 0 and 1 bits, over compact bitmaps.

A third alternative was introduced later, and continued to be studied afterwards, which was based on gap encodings [60, 80, 124]. This technique is based on encoding the gaps between consecutive bits 1, namely the sequences with bits 0. This approach worked fine, if the number of 1s in the sequence is small. Otherwise, the first two approaches perform better.

The $rank$ operation is important in Wavelet Trees for determining the occurrence of a character c until a given position i, $Occ(c,i)$. Consider the Wavelet Tree in Figure 2.4, to determine $Occ(\mathrm{a},7)$, as a belongs to the first half of the σ alphabet, one must compute $rank_0(bmap_A,7) = 5$ in the root's bitmap. This returns the position of the character in the left child node (**B**), where the procedure continues with the same strategy. Since a still belongs to the first half of the reduced alphabet, this character is represented with 0s, leading to $rank_0(bmap_B,4) = 4$[1]. This process continues with node **D**, with $rank_1(bmap_D,3) = 4$, as a belongs to the second half of the alphabet \$, a, concluding that until position 7, a occurs four times.

2.1.5 Rank and Select adapted to Bytemaps

The $rank$ and $select$ operations were later generalized to be applied on sequences of an arbitrary number of bytes, instead of bits [78, 124]. Given a sequence seq of symbols $S = s_1 s_2 ... s_n$, $rank_S(seq,i)$ counts the number of times symbol S appears in seq until position i, while $select_S(seq,i)$ indicates the position where the i-th occurrence of S appears.

B: b_1 b_2 b_3 b_3 b_2 b_1 b_1 b_3 b_3 b_2 b_1 b_1 b_2 b_2 b_1 b_3

$rank_{b1}(B,10) = 4$ $\quad$ $select_{b2}(B,5) = 11$

$rank_{b2}(B,10) = 3$ $\quad$ $select_{b2}(B,5) = 13$

Figure 2.6: $Rank$ and $select$ examples in a bytemap

This particular scenario is the one applied to the work presented in this thesis, as the compressed approach uses sequences of bytes to represent the text. For that matter, to decompress the textual data, or to get some statistics about a given term or pair of terms, the $rank$ and $select$ are extremely important (more detail in Section 3.3.2).

Some approximations for computing $rank$ and $select$ over bytemaps are based on sequence of bits, using just plain bitmaps or Wavelet Trees. The first approach uses indicator bitmaps for each byte, using the algorithms for computing $rank$ and $select$ in constant time [140], but occupying more space due to having a bitmap per byte. The second approach uses Wavelet Trees, resulting in a balanced binary tree with eight levels, where level i of the tree has the i-th bits of the binary sequence of each byte. The $rank$ is determined exactly as explained before, starting from the root to the leaves. The $select$ is determined the other way

[1] Since the bitmaps are represented starting at position 0, the procedure uses the $rank$ result - 1.

around, starting from the leaf corresponding to the byte being processed, and applying $select$ operations upwards in the tree, until the root [53].

Fariña et. al. [53] presented two approaches directly supporting $rank$ and $select$ over the sequences of bytemaps. The first approach stores absolute counters of the frequency of a byte before a given position at given intervals, losing the constant time for $rank$ and $select$, but avoiding sequential scanning for more than small portions of the bitmap (between intervals). Following this strategy, $rank_b(B,i)$ is computed by determining the number of occurrences of byte b from the beginning of the last block before position i, until i, in a sequential scan, and adding the result to the value in the mentioned block. For $select_b(B,i)$ it is necessary to use binary searches over the values in the blocks, until finding the first value x such as $rank_b(B,x) = i$, and perform a sequential scan over that block alone.

To improve this first approach, the same authors introduced a two-level block structure, with the sequence being divided in sb superblocks, with each superblock being divided in b blocks of size $n/(sb*b)$, similarly to the two-level structure of Jacobson's work [90]. The first level stores the number of occurrences of each byte from the beginning of the bitmap to each superblock. The second level stores the number of occurrences of each byte, from the beginning of the superblock until each block. Following this strategy, $rank_b(B,i)$ is determined by counting the number of occurrences of b from the beginning of the last block before i until position i, plus the values in the block and superblock for byte b. This reduces the response time from $O(n)$ to $O(n/(sb*b))$. For $select_b(B,i)$, there is again a binary search for x, where $rank_b(B,x) = i$, starting in the superblocks and then in the blocks inside the respective superblock, finishing up with a sequential scanning on the right block. The time reduces from $O(log\ b + n/b)$ to $O(log\ sb + log\ b + n/s(sb*b))$.

2.2 Phrase-based Machine Translation

Natural languages are complex systems, with many words having different meanings and obeying to different word order policies, thus making Machine Translation a hard problem. As it is not feasible to give this entire information to a Machine Translation engine as input, such systems must take decisions and determine translation alternatives as correct as possible, from an incomplete knowledge base. Statistical Machine Translation uses statistical decision theory to train translation models, taking the best decisions possible from this limited knowledge base, with the objective of achieving the best machine-made translation quality.

At first, statistical translation models [24, 144, 172, 178] were based on word-to-word correspondences between the source and the target language, which poses several restrictions. Later, phrase-based translation models [106, 146, 147] led to important improvements in Machine Translation by using phrases, taken as sequences of words, as the main unit of translation, instead of words. This approach is able to tackle local reorderings, like the English (EN) - Portuguese (PT) pair `green house` <->`casa verde`, where `green` is translated by `verde` and `house` by `casa`. Additionally, the phrase-based model enables 1 <-> M, M <-> 1 and M <-> M

correspondences between phrases, with M as the number of words in the phrases. The example above demonstrates the M <-> M approach, while the pair `counterclockwise` <->`no sentido contrário aos ponteiros do relógio` shows an example of a 1 <-> M correspondence between phrases.

However, phrase-based models still struggle with phrase reorderings. In Figure 2.7, example **1** shows a case where `is set out below` is translated by `transcreve-se adiante`, in English-Portuguese, but the order of the phrases changes drastically from one language to the other. Other limitation of phrase-based models lie in non common translations, that come from particular constructions of each language. In example **2**, the translation of the phrase `In Portugal and Spain alike` in Portuguese is influenced by the language construction `In...and...alike`. The correct Portuguese translation is `Tanto em Portugal como em Espanha`, which is influenced by the correct translation of `In...and...alike`, which is `Tanto em...como em...`. Translating directly the subphrases could result in a non meaningful phrase.

Considering longer phrases for training the translation engine could be a solution to solve these problems. However, Koehn et. al. [106] found that phrases longer than three words does not improve translation quality in relevant numbers. Other models were proposed over the years, some with no reordering [109, 187], others with phrase reordering independently of their content [106, 146] and others with phrase reorderings taking into account some lexical sensitivity [170, 173, 174], but the problem was not successfully solved.

2.3 Hierarchical Machine Translation

Chiang [32] tackled the limitation of phrase-based translation models by introducing a Hierarchical Phrase-based Translation model, which formally is a synchronous context-free grammar. This model takes into account that languages are hierarchically structured and uses all the strengths of the phrase-based model, such as learning reorderings of words, and capitalizes on that knowledge to learn also phrase reorderings.

A hierarchical phrase contains non variable subphrases, or terminal elements, and gaps, or non terminal elements. A gap is a placeholder in a hierarchical phrase, that can be replaced by any subphrase, within a limited word-based length, or be empty [31]. An occurrence of an hierarchical phrase in a text, must match the non variable subphrases, and any other subphrase that may match the gaps placeholders, inside the limited window. Figure 2.7 shows examples of hierarchical phrases for the phrase pairs mentioned above, with X_1 and X_2 representing gaps, and the non variable subphrases explicitly represented in the hierarchical phrases.

Using gaps as placeholders between non variable subphrases enables learning translations and solving phrase reorderings, that otherwise would be difficult using the phrase-based models. The hierarchical phrase `In` X_1 `and` X_2 `alike` <-> `Tanto em` X_1 `como em` X_2 defines a translation rule in EN-PT that states that `In...and...alike` <->`Tanto em...como em...`, in which the gaps can be replaced by any subphrase that are correct translations of

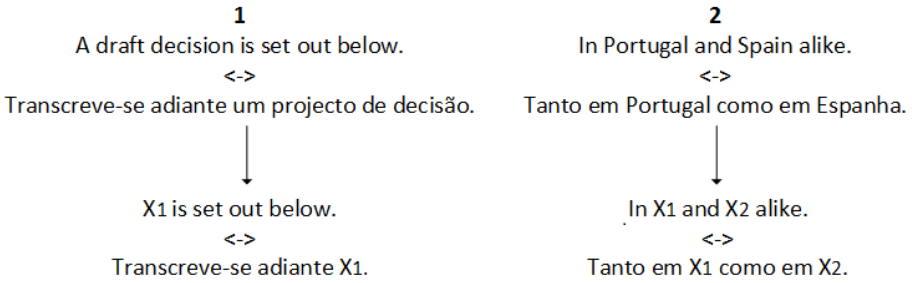

Figure 2.7: Example of phrases in English with their translation in Portuguese and the respective hierarchical phrase pairs

each other. Recovering the example in the previous Section, the gaps X_1 and X_2 would be replaced by `Portugal` <-> `Portugal` and `Spain` <-> `Espanha` respectively.

On the other example, `Transcreve-se adiante` X_1 <-> X_1 `is set out below`, the placement of the gaps show a big change in the order between the original phrase and its translation. This case is tackled successfully by the hierarchical phrase-based model, as the placement of the gaps determine that the subphrase at the end of the English phrase is translated by the subphrase at the beginning of the Portuguese phrase.

This hierarchical representation of phrases with gaps makes this statistical approach superior to the phrase-based alternative, improving translation quality significantly, due to ability of dealing with these changes of order and language constructions.

2.4 Iterative Translation Alignment and Extraction

The work presented in this thesis is inserted in a research context that includes a broader iterative process of four steps: 1) alignment of parallel sentences at phrase level, using accumulated bilingual lexica; 2) extraction of new bilingual translation pairs of word and multi-word phrases, using the aligned texts; 3) classification of the newly extracted translation pairs as correct or incorrect; 4) human validation of extracted and automatically classified translation pairs.

By extracting new translation pairs, that are later classified and validated as correct or incorrect, this workflow is able to learn more information at every iteration, increasing the knowledge base of the system. For instance, with every extraction of correct translation, a bilingual lexicon for that language pair will have more information, leading to the possibility of refining the alignment, in a next iteration, with the new gathered information. With better alignments, better and more accurate extractions can be made, and the process continues iteratively. Using an automatic translator, that takes advantage from this accumulated knowledge, leads to a considerable improvement in machine-made translation quality [4].

This iterative process is not fully automatic, as it uses the input of human translators and linguists, that validate as correct or incorrect the automatically extracted pairs. This human input ensures that all the tasks in the iterative flow improve in a sustainable way, leading to better results in each task and to a significant improvement in Statistical MT quality.

Despite most of the work regarding Machine Translation systems is fully based on automatic approaches, there is some work that looked at human interaction in Machine Translation, showing the benefits that it can bring to improve the machine-made results. Callison-Burch et. al. [26] proposed a Statistical Machine Translation system that uses two levels of human edition to improve the translations produced by the system. Green et. al. [74] proved that the quality of machine-made translation improved over time with post-edition done by human users, reducing as well translation time. Zaidan and Callison-Burch [186] proposed a metric that used as well human input, but with focus on building a database of knowledge for supporting the training stage of a translation engine, instead of focusing in post-edition of translations.

2.4.1 Parallel Text Alignment

Two texts T_1 and T_2, in two different languages L_1 and L_2, are parallel if T_2 is a translation of T_1 in language L_2 and vice-versa. Parallel text alignment is the task of identifying the corresponding segments within parallel texts, that are correct translations of each other. There are different granularities for texts, such as documents, subdocument parts (titles, paragraphs, sentences), and subsentence parts (phrases), which lead to different granularities in the alignment task.

Sentence, or coarse, alignment determines segments of text, like sentences or paragraphs within parallel texts that are translations of each other. This alignment granularity is extremely useful for Statistical Machine Translation, for creating Translation Memories and for usage in Computer-Assisted Translation tools. Bluealign [166] is a state-of-the-art sentence alignment with excellent results for noisy texts, such as texts in PDF format, using the BLEU score to determine sentence similarity [151]. Bluealign is interesting as it uses as well accumulated translation knowledge, in contrast with other aligners that infer translations from the text being aligned, during the alignment process [20, 132, 177], achieving better results for smaller texts.

Using a similar logic, Gomes et. al. [69] introduced an aligner that uses the coverage of a lexicon in a corpus, namely the ratio of text covered by expressions found in the lexicon. This approach was able to take advantage of acquired knowledge obtained from several iterations of the alignment and extraction flow introduced above. Recently, Gomes et. al. [71] introduced a different coverage-based alignment, using only Moses phrase tables [107], without requiring the translation of the texts to be aligned, as in Bluealign.

Figure 2.8 shows an example of a small monotonic coarse alignment, or sentence-based alignment, with three segments. In the scope of this thesis, a parallel coarse alignment has the same number of coarse segments in both languages, regardless of the language. When translating a text from one language to another, there are local changes of order and specific language characteristics that must be taken into account and could make the previous statement false. However, these changes of order are considered just within sentences, or eventually paragraphs at most. In this work, it is not possible to have misalignments with such a coarse granularity [2].

[2] It is possible to have misalignments at a coarse level of granularity, if the texts are extracted from PDF, which can bring hidden characters, or documents with pictures.

English	Portuguese
The European Council and the Council shall be heard by the European Parliament in accordance with the conditions laid down in the Rules of Procedure of the European Council and those of the Council.	O Conselho Europeu e o Conselho são ouvidos por o Parlamento Europeu em as condições previstas em os regulamentos internos de o Conselho Europeu e de o Conselho.
The Commission may attend all the meetings of the European Parliament and shall, at its request, be heard.	A Comissão pode assistir a todas as sessões de o Parlamento Europeu e é ouvida quando assim o solicitar.
It shall reply orally or in writing to questions put to it by the European Parliament or by its members.	A Comissão responde, oralmente ou por escrito, a as questões que lhe sejam colocadas por o Parlamento Europeu ou por os seus membros.

Figure 2.8: Monotonic coarse alignment

Sub-sentence, or fine, alignment determines the correspondence between smaller segments of the parallel texts, such as word and multi-word terms within parallel sentences. Given a bilingual lexicon, the fine alignment process determines all the pairs of word and multi-word terms from the lexicon that co-occurs within the parallel sentences, using that information to determine the finer alignment segments. Gomes [69] introduced as well a sub-sentence alignment, that uses a bilingual lexicon and the coverage measure to align the words and multi-words found in the lexicon, leaving for the extraction task, in the iterative flow, the identification of new and unknown pairs of word and multi-word terms.

Figure 2.9 shows the fine alignment of the second and third coarse segment presented in Figure 2.8, that was obtained at some time. The fine segments have a third column in the middle with the information about the alignment, where *A* indicates a correct alignment, ? an incorrect or unknown alignment, and *P* a punctuation mark. The monotonic fine alignment can originate empty segments, causing an unknown (?) situation. These cases are very common when there are significant changes of order in the texts after the translation, which together with the small length of the fine granularity segments, can cause some misalignments. Just like in the coarse segments and taking into account that the fine alignment considers empty segments, the number of fine segments within every coarse alignment is the same in both sides of the parallel text.

To tackle the empty segments and unknown correspondences in the fine monotonic alignment, Gomes [69] also introduced a subsentence non-monotonic alignment. With this sub-sentence alignment, every known word and multi-word term within a coarse segment have its alignment candidates associated, linking them to their target terms in the other language,

2nd coarse segment

English	Alignment	Portuguese
The	A	A
Commission	A	Comissão
may	A	pode
attend	A	assistir a
all	A	todas
the	A	as
meetings	A	sessões
of	A	de
the	A	o
European Parliament	A	Parlamento Europeu
and	A	e
shall , at its	?	é ouvida quando assim o
request	A	solicitar
, be heard	?	
.	P	.

3rd coarse segment

English	Alignment	Portuguese
	?	A Comissão
It shall reply	A	responde
	?	,
orally	A	oralmente
or	A	ou
in writing	A	por escrito
	?	,
to questions	A	a as questões
	?	que lhe sejam
put	A	colocadas
to it	?	
by	A	por
the	A	o
European Parliament	A	Parlamento Europeu
or	A	ou
by	A	por
its	A	os seus
members	A	membros
.	P	.

Figure 2.9: Monotonic fine alignment

regardless the order they appear in the text. In this research work, the text alignment considered follows the generally correct assumption that the aligned segments follow the same order in both documents, thus being a monotonic text alignment. This does not prevent the Bilingual Framework to be extended to support non monotonic alignment as well, without much effort (see Chapter 7). However at this stage, it only supports monotonic alignments, as in bilingual lexica, translation models or monotonically aligned parallel corpora.

2.4.2 Translation Extraction and Classification

Translation extraction consists of extracting a bilingual lexicon from aligned parallel corpus. This task gets an aligned parallel corpora and produces a set of possible translation word and multi-word pairs, using term occurrence and co-occurrence statistics, determined based on the parallel corpora. Within the iterative process mentioned in the beginning of this Section, the extraction task uses progressively refined alignments, which together with information about correctly and incorrectly previously extracted pairs, enables more quality extractions.

Extracting pairs of words and multi-words commonly uses similarity between words, by identifying cognates [3] with several measures such as Longest Common Subsequence Ratio and Edit-Distance-based similarity [70, 87, 130, 167], or by using hierarchical phrases as extraction templates. These hierarchical phrases, together with a sub-sentence alignment and previously extracted lexica, enable generating new pairs by using the non variable parts of the phrases as context to replace the gaps by word, multi-word and subword possibilities [69].

[3] Words with a common etymology

Within the iterative workflow previously described, the extracted pairs need to be validated as *correct* or *incorrect*, before being added to the bilingual lexicon to feed the next iteration[4]. These entries are manually validated by linguists, which reduces uncertainty and lead to better alignment precision and more accurate extractions. However, the number of entries can be high and human validation is time consuming. For that matter, the entries are automatically classified using a Support Vector Machine trained on a set of manually classified entries, improving validation productivity [98, 100].

2.4.3 Bilingual Framework

In the context of the aforementioned iterative process, it is important to represent the parallel texts in main memory efficiently, for fast access and querying over the corpora. This is extremely relevant for obtaining statistical information, that provides key information and supports the tasks within the iterative process, like the alignment, extraction and classification. For this purpose, the Bilingual Framework requires a support based on efficient data structures to represent the parallel texts and their respective alignments, in any language pair.

Before moving to the details of the Framework implementations, it is important to clarify its structure and how its several components work together. The proposed Bilingual Framework is divided in three main layers in main memory: the Text Layer, with the texts in both languages indexed; the Alignment Layer to represent the segments of the texts that are correct translations of each other; and the Document Layer to represent the location and organization in disk, at the document and directory level, of the several texts that are part of the indexed corpora.

These three components work together, around a generic Framework API that any End-User can use. Figure 2.10 shows a diagram with a simplified structure of the Bilingual Framework. The Figure displays the three mentioned Layers, and the communication between them and the mentioned API, which has a request / response interaction with the End-User.

Besides the different Layers, Figure 2.10 shows as well a representation of an API that is available to be used by any End-User or application, with the functionalities supported by the Framework. The process can be summarized in the following generic steps, with the numeration in Figure 2.10: 1) the End-User queries the Framework, using the End-User API; 2) the Framework gets the occurrences of the terms, in both texts, from the Text Layer; 3) for every occurrence returned in step 2), the Framework queries the Alignment Layer to determine the alignment segment where they occur; 4) also for every occurrence, the Framework gets the respective Document and Directory where they appear, from the Document Layer; 5) the response is sent to the End-User with the results.

This interaction flow depends on the specific operations of the presented Layers, namely steps 2), 3) and 4). These functionalities are detailed in the next Chapters of this thesis, alongside a description of their implementation and how they contribute to the Framework in detail. Binding them all together is the Framework structure, described in the End-User API in Figure 2.11, and its functionalities centered on *creating* / *loading* the Framework and *locating* or

[4]The *incorrect* entries are also added and marked as *rejected*, as it is also useful for refining further alignments

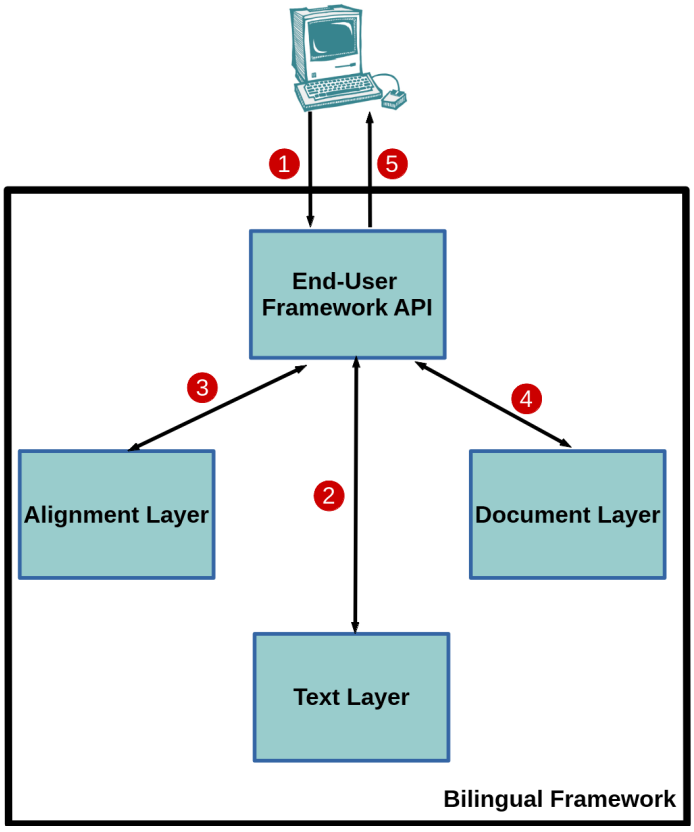

Figure 2.10: Simplified architecture of the Bilingual Framework, with the interaction between the Layers and End-Users

counting the occurrences of a term, or pair of terms, in parallel corpora.

```
struct bil_fram{
    char* langs[2];
    struct t_index * ti[2];
    struct t_api *tapi[2]
    struct a_index * ai;
    struct a_api *aapi;
    struct d_index *di;
}
```

Figure 2.11: Bilingual Text Framework Structure

Figure 2.11 shows a structure that represents the Bilingual Framework structure (**bil_fram**), which has seven fields: two text indices (*ti*), one per language; two text layer APIs (*tapi*), one per language, with the implementation chosen for the Layer; one alignment index (*ai*), to represent the corpora alignment; one alignment Layer API (*aapi*), with its implementation; and the document index (*di*). It easy to infer that these fields correspond to the three Layers of the Framework and its implementations. The APIs are represented next to the indexes, since they allow referencing the implementation of each Layer, as the Text and Alignment Layer can be

supported by different data structures.

Based on this structure, the Bilingual Framework supports the creation of the indexes that implement it, using any word-based data structure that complies to the Framework Layers and its APIs, which are detailed in the following Chapters. This indexes are stored in disk and then loaded to main memory, before any operation over the Framework starts.

Additionally, the Framework supports the functionality to *count* and *locate* co-occurrences of a pair of terms, in two different languages, in the indexed parallel corpora. These functionalities are the core of the Framework and are based on an iterative behavior, where every co-occurrence is returned at a time. This behavior leads to more flexibility for any application that may use it. An example is a bilingual concordancer application with pagination, where not all the occurrences of two terms are returned at a time, but only a subset of those occurrences. In a scenario where each page displays ten results out of a total of 30, the user could change at any time between page one to three, with the next 10 results being calculated on demand, avoiding waiting for the calculation of all 30 results from the start.

The Framework also supports context extraction, where the snippets of text surrounding the occurrences can be returned, considering a parameter window of the user's choosing.

With such operations, the Bilingual Framework can be used or adapted to support a large number of NLP applications, such as the ones that are part of the iterative process of aligment, extraction and classification already described. This Framework can then be used to calculate statistics that are important for all the steps of the iterative process, including co-occurrence frequency of a pair of terms, context extraction, coverage of a lexicon in a corpora, etc.

As mentioned before, the core support for these functionalities lie on the three mentioned Layers. The next Chapter details two implementations for both Text and Alignment layer, and one implementation for the Document Layer. All these approaches use the same generic API and support the same functionalities, meaning that the Framework can be supported by different data structures, depending on the users space and time needs. Chapters 4 and 5 detail the procedures and algorithms for phrase-based and hierarchical phrase-based search, using the data structures that support the three Layers. These procedures are also generic and can be supported by any data structure chosen to support the Framework.

CHAPTER 3

INDEXING ALIGNED PARALLEL CORPORA

This thesis describes a space and time efficient Bilingual Framework to index and search over coarsely and finely aligned parallel corpora for any pair of languages. To index aligned parallel corpora, one must take into account the parallel corpora in two languages and their respective coarse and fine alignments. With that purpose, the Bilingual Framework has three layers: 1) Text Layer, to represent the corpora; 2) Alignment Layer, to represent the parallel text alignment, at a coarse and fine granularity; 3) Document Layer, to represent the location of the different texts, and the directories where they are stored in disk [39].

As introduced at the end of Chapter 2, Figure 2.11 presents the structure that supports the Bilingual Framework, with two text indexes for representing the text and one for representing the alignment, and functionalities that focus on *counting* and *locating* co-occurrences for pairs of word and multi-word terms, over the indexed parallel corpora.

Indexes such as Suffix Trees and Suffix Arrays are full-text indexes, meaning that every position of the text is indexed. This allows searching for any word or multi-word term, starting at any position of the text. Taking into account that a lot of languages have words separated by blank spaces, and the main goal of the framework is actually search for a word or sequence of words, the results need to be filtered to remove the occurrences that are not at word boundaries. For example, searching for `able` in a full-text index would return occurrences of the term in words such as `enable`, `portable`, etc. These occurrences would not be interesting for the final result, because they are actually not occurrences of the term itself. The term is just a suffix of the actual word found, thus these occurrences should not be taken into account.

However, removing these occurrences from the final result can be a time-costly step. On top of this, indexing all the positions of the text would increase the memory consumption, with a possible multiplication factor of five to six times [55], despite the very efficient search time responses they can get. With the functionalities of the Framework being centered on words, the main unit for representing the texts indexed by the Framework is also the word.

This representation greatly influences the Bilingual Framework implementation, namely on the Text Layer, which is where the texts are actually indexed. As Suffix Trees and Suffix Arrays are so important in fields such as Bioinformatics, or Natural Language Processing, a first approach for the Text Layer is a word-based adaptation of one of these structures, the Word-based Suffix Array. This structure is actually used in several Machine Translation related tasks [28, 116, 183].

In a search for a pair of terms, in two different languages, the Text Layer provides the individual occurrences of both terms. On the other hand, the Alignment and Document Layers provide some additional context to these occurrences, on the scope of the whole parallel corpora. As a starting point, we used an implementation for the Alignment Layer based on arrays of integers.

However, these approaches can require more space than the texts themselves. In this research, I studied several compressed indexes and alternatives to support both layers, to reduce the space consumption of the Framework. This resulted in new approaches for supporting these layers: Byte-codes Wavelet Trees [22] for the Text Layer and Bitmaps for the Alignment Layer. Using such data structures reduced considerably the space consumption of the Framework, achieving values smaller than the text size [39].

Through the course of this Chapter, I describe the three Layers of the Framework and how they are implemented. This affects all the functionalities of the Framework, however, in this Chapter, I do not detail any *locate* / *count* functionality. This Chapter focus more on the structure of the Framework itself, namely the data structures that support the layers, thus describing in detail how the **bil_fram** structure, introduced in Figure 2.11, is implemented.

Over the next Sections, I introduce the initial approaches for the Text and Alignment Layers before explaining in detail the strategies we developed to compress the Framework. Then, I introduce the implementation of the Document Layer based on arrays, that I also developed from scratch for this work. Finally, the memory consumption results that we achieved are displayed, using five different English - Portuguese corpora. However, this Chapter actually starts with the pre-processing of the corpora, that is required before actually indexing the texts in the Framework. All the details regarding the search functionalities are described only in the following Chapters.

3.1 Pre-Processing the Corpora

To index parallel corpora of several different languages, one needs to understand the characteristics of each language, their differences, in a way that is possible to create a Framework generic for any language. Disregarding for now more significant language differences, the texts itself have punctuation marks, composite words and other symbols, that also may vary from language to language. This requires a pre-processing stage, before actually index the corpora, to separate all these symbols and punctuation marks, apart from the words.

In many languages, such as the Western languages, the words are separated by special symbols such as blank spaces or the already mentioned punctuation marks. Knowing this, the pre-processing stage parses and identify the words, using the blank spaces as separators to isolate the words. Additionally, the punctuation marks and symbols are also separated in blank

spaces, to be considered as a “word"in the context of the Framework. It is known that some languages, like Mandarin, do not use separators as in the Western languages. In these cases, the texts are pre-parsed and divided in separated segments, which are then considered as words.

Besides the characteristics mentioned above, indexing corpora also requires attention to the casing of the words. Indexing the texts as they are would lead to the searches being influenced by the casing of the words. Considering a search for `index` again, occurrences of `Index` in the corpora would not be returned in the final result, as the word in the text is in upper case, while the search term is in lower case. The same can happen the other way around, with searches for `INDEX` not matching any occurrence of `index` at all in the corpora. This is particularly important for the Text Layer and will be described in more detail in Subsection 3.2.1.

The Bilingual Framework gets two aligned texts, at phrase level, with every line having the pair of phrases in both languages. Here, not only the Framework is indexing the text, but also defining the alignment in the Alignment Layer. As we support multi-grain alignment, whenever this processing reaches a *newline* character, automatically indicates that a paragraph has ended and a new one is about to begin. The Alignment Layer is detailed in Section 3.4.

3.2 Text Layer

The Text Layer of the Bilingual Framework is where the texts of the parallel corpora are represented and managed. Each one of the two languages represented have its own “instance"of the Text Layer, meaning that a corpora in one language does not have any information about, or influence, the corpora in the other language. For that matter, the text layer consists of two text indexes, one for each language, with the respective corpora indexed in main memory.

Besides indexing the texts, the Text Layer supports functionalities such as *counting* the frequencies and *locating* the occurrences of a word or multi-word term in the corpora of the respective language. Figure 3.1 shows the API of the Text Layer, with all its functionalities, implemented by the data structures that support it. The API for the Text Layer evolved during the research period that resulted in this thesis, in particular for the *locate* operation. However, this was the starting point of the Text Layer, which was a key factor for the changes made afterwards. Chapters 4 and 5, describe in detail these changes and the algorithms that support it.

By looking at Figure 3.1, there is a structure *t_api*, that has a set of functions in its body. This structure represents the Text Layer API, introduced first in the previous Chapter, in Figure 2.11, as **struct t_api**. This API always have these six functions, regardless of the data structures used to implement it. With the functions being created within the API, every approach for the Text Layer has to implement this API. Then, the actual implementation of the API that we want to use, can be called directly, meaning that the Framework will automatically use its underlying implementation. For instance, considering a Text Layer supported by Suffix Arrays, by referencing that implementation directly, the Framework would use these functions with the underlying implementation based on Suffix Arrays.

The first three functions in the API focus on the creation and management of the index that supports the Text Layer, for one language. The first one, *create_index*, handles the creation of

```
struct t_index;

struct t_api{
   void create_index (char* filename, char*
      text, int text_len, struct t_api *api);

   struct t_index* load_index(char* filename,
         struct t_api *api);

   void free_index(struct t_index* index);

   int* locate (struct t_index *index, char*
      term, int* num_occs);

   int count(struct t_index* index,
      char* term);

   char* get_snippet (struct t_index *index,
      int off, int len, int* len_bytes);
}
```

Figure 3.1: Initial Text Layer API, for one language

the index, receiving as input the name given to the files that are going to be created (*filename*), to store the index in disk, the *text* to be indexed and its length in words (*text_len*). The second function, *load_index*, receives the name of the files in disk created in the previous function (*filename*), and builds the index in main memory, returning a structure with all its information. This makes the index available in memory and ready to answer any search queries from users or automatic scripts. Finally, the third function, *free_index*, just releases all the structures that the index uses from main memory, disabling it for further usages, until it is loaded again.

The actual index that supports the Text Layer is defined in the structure **t_index**, presented in Figure 3.1. In the presented API, the structure is not yet detailed, because it depends on the implementation used for the Text Layer. This enables maintaining the same generic API, regardless of the data structures and algorithms used to support these functionalities underneath. Any new implementation of the Text Layer, needs also to define the **t_index** structure.

This thesis presents two different implementations for supporting the Text Layer, both word-based: a first approach supported by Word-based Suffix Arrays and a compressed solution, that I adapted to support the Framework's functionalities and behavior. However, before introducing in detail these two implementations, the next Subsection tackles the casing problem, mentioned above in the pre-processing stage.

3.2.1 Case Insensitive Layer

As mentioned before, it is very common that a word may appear in lower case, capitalized or even in upper case throughout a text. In a search environment, all the occurrences of that word should be taken into account, regardless of their casing.

To tackle this problem, we added a Case Insensitive Layer (CIL), that is always present regardless of the implementation used to support the Text Layer. The objective of CIL is to normalize the casing of the corpora indexed in the Framework, by converting every word in

uppercase or capitalized to lower case, before actually indexing the texts. As stated before, this allows every search to be normalized, regardless of the casing used by the users.

This approach leads to having the indexed text always in lower case, which also affects the implementation of the search queries. For that matter, every query to the Framework is also converted to lower case by the CIL, before actually calling the Text Layer functionalities, that implement the search functionalities of the Framework.

With this kind of interaction, the CIL works almost as an intermediary between the actual Text Layer Implementation and the Framework API, not only for creating the indices in lower case, but also in every *count* or *locate* search query. Figure 3.2 shows a representation of this interaction, with the CIL appearing between the Framework and the Text Layer implementation. Being actually an important part of the Text Layer, the Case Insensitive Layer follows its exact same API, depicted in Figure 3.1.

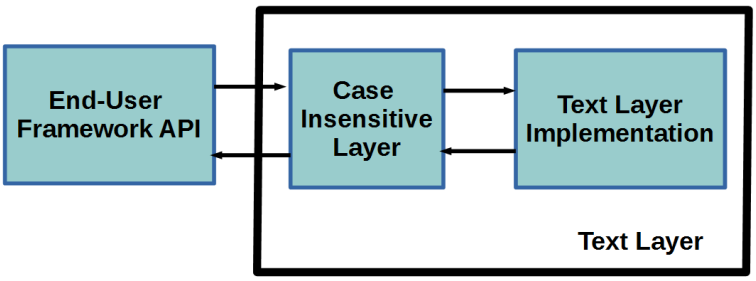

Figure 3.2: Text Layer, with the Case Insensitive Layer

By following the Text Layer API, the CIL has an implementation of its functions. Considering that the main objective of CIL is to normalize the indexed corpora, its main functionality lies on converting the texts to lower case. Additionally, as it also works as an intermediary between the Framework and the Text Layer, as seen in Figure 3.2, several of its functionalities are just calls to the actual Text Layer implementation, with the term in the input converted in lower case. However, the function *get_snippet*, which extracts a snippet of text surrounding an occurrence, requires the original information of the casing, so that it displays the correct formatting of the texts. This indicates that the CIL needs to store the information about the original casing of the words in the text.

Figure 3.3 shows the structure **t_index** used by the Case Insensitive Layer, with the information necessary to support the behavior described above.

```
struct t_index{
    struct t_api *text_layer_api;
    struct t_index *text_layer_index;
    int *lower_case, *upper_case;
}
```

Figure 3.3: Case Insensitive Layer Index

The CIL implementation of **t_index** has four fields. The first two fields, *text_layer_api* and

text_layer_index, represent the Text Layer implementation used by the Framework, corresponding to the rightmost element in Figure 3.2. The first one is the API of the Text Layer, while the second is the actual index implementation used, either based on Suffix Arrays or Byte-codes Wavelet Trees. The **t_index** structure again will depend on the implementation chosen, with its two alternatives being detailed later in this Chapter. The CIL uses these two fields to call the functionalities from the Text Layer, after the pre-processing is done at the level of the corpora and of the terms being searched.

The remaining two fields are two bitmaps, one to represent the lower casing and the other for representing the upper casing. With a bit to represent every word in the text, these bitmaps are used to mark the original casing of each of those words, before being normalized to lower case. This way, if a word of the text is all upper case, the respective bit of bitmap *upper_case* will be 1. If the word is lower case, then the same logic applies for the bitmap *lower_case*. If the word is capitalized[1], both bitmaps will maintain a bit 0 in the respective position. This means that when there is not 1 in any of the bitmaps, for a particular word, then that word is originally capitalized in the text. These bitmaps are specifically important for recovering the original casing of the words in the indexed corpora, as for the *get_snippet* functionality.

As stated before, much of the CIL functionality lies on using the Text Layer directly. However, creating the indices with the text in lower case is not as straightforward. When creating an index, the first step lies on normalizing the text, by converting every word to lower case. Alongside the conversion to lower case, the bitmaps are populated with the respective bits, to represent the original casing considering the logic explained above. At the end, the bitmaps are written in disk and the *create_index* of the Text Layer is called, using already the lower case text.

Algorithm 3.1 shows a simplified version of the procedure to create the indices, in one language, in the Case Insensitive Layer. Much of the nomenclature used are the names defined in the Text Layer API in Figure 3.1. Some of the fields created at the beginning of the function require some explanation. The fields *lower_case* and *upper_case* represent the bitmaps for the original casing. These bitmaps are filled throughout the function, with a bit being added for every word of the text, with the abstract function *add_bit*. At the end, these two bitmaps are written in disk, to be later loaded in main memory, as a part of the CIL index, when *load_index* function is called. Then, a field *lc_text* is also created to hold the text to be indexed in lower case, to be latter sent as parameter of the call to the Text Layer implementation. Throughout the functions, we used some functions such as *to_lowercase*, to transform a string to lower case and *is_lowercase* and *is_uppercase* to check if a word is in lower case or upper case.

Note that the *create_index* function of the Text Layer implementation is called, using the API and the index defined in the structure **t_index**. This is an example of how the CIL works together with the actual Text Layer Implementation. Also denote that its last parameter is *NULL*. This is used because there are no more APIs to call, after the Text Layer implementation's API.

Having the Case Insensitive Layer defined and the word casing problem solved, the following subsections introduce the approaches for the actual implementation of the Text Layer.

[1] A capitalized word has the first character as upper case and the following characters as lower case.

```
void create_index(char* outfile, char* text, int text_len, struct t_api *api){
  int lc = 0;
  int uc = 0;
  int* lower_case;
  int* upper_case;
  char* lc_text;
  for(every token in text){
    lc = is_lowercase(token);
    uc = 0;
    if(!lc){
      uc = is_uppercase(token);
    }
    add_bit(lower_case, lc);
    add_bit(upper_case, uc);
    append(to_lowercase(token), lc_text);
  }
  write_in_disk(filename, lower_case);
  write_in_disk(filename, upper_case);
  api->create_index(filename, lc_text, text_len, NULL);
}
```

Listing 3.1: *create_index* function on the Case Insensitive Layer

3.2.2 A First Approach: Word-based Suffix Arrays

A first approach for supporting the Text Layer consists of Word-based Suffix Arrays. In its original conception, a Suffix Array for a text T is a full-text index, indexing a pointer to every position of T. Figure 2.3, in Section 2.1, shows exactly that for T = `abracadabra#`, where $T[A[1]]$ =`#`, $T[A[2]]$ =`a#`, $T[A[5]]$ =`acadabra#`.

However, for such a Framework as the one detailed in this thesis, indexing every position of the text is unnecessary. Besides that, such approach would lead to a considerable superior amount of required space, and would also impact negatively the search time response.

Word-based text indices became commonly used in Information Retrieval [182] and string-matching studies [6], where several scenarios, like in computational linguistics, the word boundary is enough. A good example of this scenario is the task of calculating the frequency of a word or multi-word term in T [183], where only the word-boundaries are relevant.

The general idea behind a word-based index, for a text T, lies on storing a subset of positions of T, namely the offsets where the words in T begin, instead of all the positions in the text. It is straightforward to build such an index, if one considers the simple approach of building it for every position of the text, and then discard the positions that do not occur at word boundaries. However, this requires $O(|T|)$ construction space, besides the size of text and of the Suffix Array, for an index that will take $O(w)$ of space, with w being the number of words in the text.

Often, the concern regarding full-text indices is related to their time and space performance, however an efficient index may be unuseful if it does not have an efficient construction algorithm [193]. The construction of full-text indices was a concern tackled in relevant work along the years [83], with some important results being achieved for Word-based Suffix Trees [6, 88]. Later, Ferragina and Fischer [55] designed an approach for the construction of a Word-based

Suffix Array, which is the base structure for the first approach for the Text Layer. All these word-based approaches are variations of the respective full-text index construction algorithms, in Suffix Trees [176] and Suffix Arrays [96].

Ferragina and Fischer [55] proposed a Word-based Suffix Array, considering the word as the main unit, taking $O(|T|)$ of construction time and $O(w)$ space in addition to the text T. Consider I as the set of positions where a new word starts in T, and $Suffix_I(T)$ the set of suffixes of T that start in positions I. Generically, a Word-based Suffix Array SA represents all the suffixes in $Suffix_I(T)$, following a lexicographic order. More formally, the Word-based Suffix Array $SA[i..w]$ is a permutation of set I such that $T_{SA[i-1],n} < T_{SA[i],n}$ for every $1 < i \leq w$. This definition is very similar to the classical character-based Suffix Array. As a matter of fact, the word-based version is supported by a classical Suffix Array. However, to store the text at the word boundary, its the indexed text itself that needs to change.

To index text at word boundaries, the first thing to do is to assign an unique integer value, id, to every unique word in the text. This assignment follows the lexicographic order of the words, to keep as well the same order, but now with integers. Replacing every word in T by its id results in the integer sequence SID. As SID is a sequence of characters, it is indexed in a character-based suffix array based, where each character is the id that represents a word of T. Figure 3.4 shows a simplified example of the construction process of a Word-based Suffix Array, with an overview of its several steps.

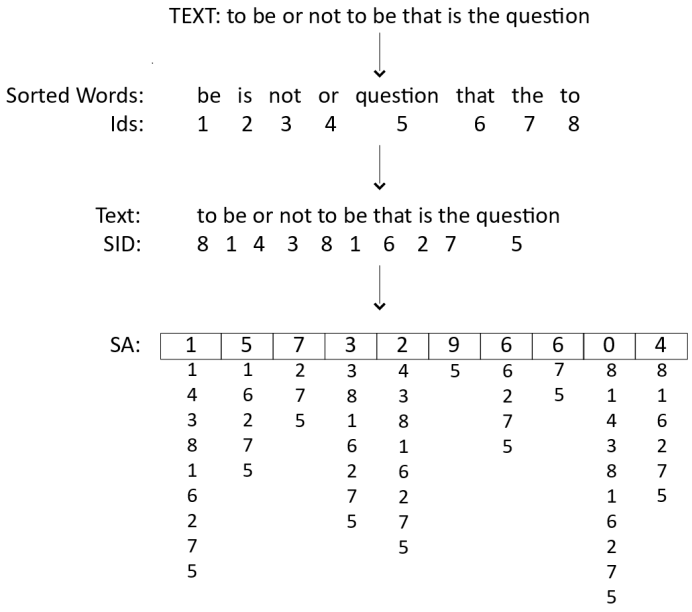

Figure 3.4: Example of a Word-based Suffix Array

First, the words of the text are sorted following a lexicographic order and the id is assigned to every unique word, following that order. In the example in Figure 3.4, the words in the text `to be or not to be that is the question` are sorted lexicographically, resulting in `be is not or question that the to`, with the respective ids respecting the order.

Then, SID is created, by replacing every word of the text to the sequence of ids, in this example $SID = 8143816275$. Then, SID is indexed in a classical, character-based, suffix array.

Ferragina and Fischer's approach uses the set I as a starting point in SA. Then, using the words of the text, a radix-sort is performed over SA, to sort it lexicographically. During this process, the array is "divided"in buckets, where each bucket consists of the suffixes starting with the same word of the text. In Figure 3.4, suffixes `be or not to the be that is the question` and `be that is the question` would be in the same bucket, as both start with the word `be`. Then, SID is built with the bucket-number of each suffix in A. Finally, the suffix array for SID is built, using the linear-time construction algorithms for character-based suffix arrays [96], as they work fine for integer alphabets as well.

In the proposed Framework, the Text Layer uses a slightly different, and more simplified, approach for Word-based Suffix Arrays. At first, the assignment of unique ids is made by traversing the whole text first, without sorting the words, and they are stored in a hash table. During this process, SID is also created. This table will work as the vocabulary for the Suffix Array, where the key is the word, and the value is the respective id. Then, the Suffix Array is built for the sequence SID. This way, SID is indexed in a classical Suffix Array, as any text would be.

Following this strategy, there are three essential features that support the Word-based Suffix Array: the SID (or *text*), the Suffix Array and the vocabulary (*voc*), with the hash table to represent the mapping between the words and the id. Figure 3.5, shows the definition of the structure **t_index**, with the Text Layer approach based on Word-based Suffix Arrays. Besides the three mentioned elements, this structure also has the number of tokens (*nr_tokens*) in the text and the number of bytes (*nr_bytes*) used in memory by the index.

```
struct t_index{
    int nr_tokens;
    int nr_bytes;
    int* text;
    int* suffix_array;
    struct vocabulary *voc;
}
```

Figure 3.5: **t_index** structure for Word-based Suffix Arrays

Algorithm 3.2 shows the general procedure for creating the Text Layer, based on Word-based Suffix Arrays, following the logic described above. The *add_token_voc* function creates an entry in the hash table, with the token and a new id, that is incremented with every new word. If the word already exists, then its previously created id is used. Then, with these ids, SID is progressively created and used to build the classical suffix array at the end. This algorithm is very simple, and it runs in $O(|T| + w)$.

With such a structure for the Word-based Suffix Array, searching for a term in this structure is not very different from the character-based version of the index. Actually, as it uses the full-text Suffix Arrays, the main difference lies in an extra step, that is based on mapping every word in the term to its respective id. After that, binary searches are performed over the Suffix Array

```
void create_index(char* outfile, char* text, int text_len, struct t_api *api){
  for(every token in corpus){
    id = add_token_voc(token, ti->voc);
    append(id, ti->text);
    ti->nr_tokens++;
  }
  ti->create_suffix_array(ti->suffix_array, ti->text);
}
```

Listing 3.2: *create_index* function based on Word-based Suffix Arrays

to determine the interval of occurrences of the term in the indexed corpus. Chapter 4 details the searching procedures used over the Word-based Suffix Arrays approach for the Text Layer.

3.3 Compressing the Text Layer

The text indices presented so far support fast string matching operations, which enable a time efficient implementation of the Text Layer. Apart from the time performance, these word-based data structures lead to a considerable better space performance, when compared to their classical, character-based implementations. These word-based indices can reduce around 20% [55] of the standard suffix arrays / trees requirements of four to 20 times the text size [110, 125, 127].

However, considering the amount of textual information available today, from parallel corpora to the accumulated knowledge introduced in Section 2.4, representing the Text Layer in secondary storage could be inevitable, even with these space savings. On the other hand, indexing is more beneficial for larger texts, as searching the index would be considerably more efficient than direct searches over such large texts. This creates a situation where the biggest advantages of indexes could be wasted due to the primary memory space limitations.

The benefits of text indexing can be compromised if there is not enough storage space to maintain the index in main memory and support fast access to its indexed data. Despite the significant amount of storage available in secondary memory, the disk access is considerably slower when compared with primary memory and CPU caches, which affects the efficiency on accessing the indexed texts. In addition, most indices are not designed to work efficiently in secondary memory [56].

Over the years, several approaches were proposed for disk-based data structures, such as Suffix Trees and Suffix Arrays, with interesting results [41, 45, 56, 94]. In fact, more recent approaches are getting closer, or even surpassing, some compressed indices, for some of its functionalities [61, 68, 92]. However, traversing the Suffix Arrays / Trees through arbitrary paths is a very common necessity, for supporting most of the complex functionalities offered by these structures. Due to the lack of reference locality, secondary storage performs poorly in such task, thus making in memory representation preferable [141]. This led to the study of data compression and the direct application of compressed data structures to support the proposed Bilingual Framework.

3.3.1 Introduction to Compressed Indexes

Compressing a text is a common technique to represent it using less space. To achieve this, the text is transformed to obtain a more space efficient representation, usually using some codification mechanisms, like the Huffman encoding [85]. However, this compressed representation, despite lighter, has a drawback. By having the text codified to save space, it is necessary to decode it to obtain the original text back, which in a search procedure for instance, can make it slower than searching directly over the original text. Nevertheless, comparing the main and secondary memory access times, it is better to store a large text that does not fit in main memory in its compressed form, and then decompress it, than having it stored in disk. And in its compressed form, the text may fit easily in main memory, thus avoiding disk access.

The same logic can be also applied to text indexes. In fact, it makes even more sense to reduce the indexes space consumption, considering the space consumption that these data structures may require and its benefits in terms of accessing and querying the text.

A succinct index enables fast searching functionality using a space proportional to the size of the text, like two times the text size. For instance, the first succinct index [111] achieved $O(n \log \sigma)$ of space, in comparison with the classical indexes, such as Suffix Arrays and Trees, which occupy $O(n \log n)$. A succinct index is indeed a compressed index, as it achieves space proportional to the k-th order entropy of the text. Over the years, several succinct data structures were proposed, from succinct representations of trees [136], to compressed data structures [77] and succinct structures for problems such as range minimum queries [16], document listings [138] and computing the $TF * IDF$ [160].

The entropy of a data source measures the average number of bits that is necessary to encode a symbol. It is often used as an estimate of the performance of data compression algorithms, as it effectively bounds the performance of the strongest compression possible. Entropy is defined in the context of probabilistic models for the mentioned data source. For example, coin flips have an entropy of 1 bit per flip, since you never know the result of the next flip. On the other hand, if you generate a long string of the same characters, you know already what is going to be the next character, thus the entropy is zero. There are several models of entropy, from zero-order model to the k-order model. The zero and first order models consider the characters in a text statistically independent, while the second-order onwards determine a probability of a character to appear, given that the previous character is a given character.

A compressed data structure takes advantage of the regularities of the text to be indexed, obtaining a space consumption proportional to the size of the compressed text. The Burrows-Wheeler Transform (BWT) [25] was an important leap forward for indexing compressed text, as it is a data compression technique that makes the text easier to be compressed [126]. This technique is frequently used in several compressed solutions [3], including bzip. Additionally, there are other techniques and basic notions that are important to save space. An example is that it is not feasible to maintain both index and text in memory, to achieve an efficient space consumption. However, the text and the index are necessary to locate the positions where a term occurs or to display text content. This happens directly with Suffix Arrays and Suffix Trees,

either with their classical and word-based approaches.

This changed completely with the appearance of compressed self-indexes. A self-index is a compact data structure that is able to represent the text in a space proportional to its empirical entropy [126]. Even with such characteristic, these indexes contain enough information to efficiently search and extract any text substring [140]. A self-index is then able to replace entirely the text, thus avoiding the necessity of storing the text as well in main memory, alongside the index. The first approach for a self-index lies on sampling suffix arrays in regular intervals, meaning that only some positions of the suffix array were indexed in main memory, using a technique called backward search to obtain the remaining information [54]. Other alternatives followed it, achieving search time functionalities with almost optimal time [58, 59, 76, 78, 120, 157], and space requirements close to the ones of the compressed texts.

3.3.2 Compressed Text Layer

Thus far, the compressed techniques presented are character-based, just as the standard suffix arrays / trees. However, a word-based approach is more fitting for supporting the Bilingual Framework, which poses some challenges, such as the fact that the vocabulary (words) is considerably large.

Word-based text compression is a technique to represent textual information using considerably less space, by assigning a codeword of bytes to each word of the text. This is achieved by using coding methods, which depending on the one chosen, can lead to compression ratios of 25% to 35% of the original text size. There are compression techniques designed for natural language texts that allow searching the compressed text up to eight times faster than the original text [134, 175], using word-based models [131]. This shows that the compressed text, besides potentially avoiding the slower secondary storage, shows promise of supporting important search mechanisms, with indexing-like characteristics.

Words follow a more biased distribution of frequencies than characters, following the Zipf Law [12, 191]. This Law states that the most frequent word in a corpus occurs twice as often as the second most frequent word, three times as often as the third most frequent word, etc, and it is also applicable to random texts [113]. This distribution makes the texts to be highly compressible at word boundary, with encodings such as Huffman [85] achieving compression ratios close to 25% of the text size.

Huffman encoding is a lossless data compression technique, that assigns codes to characters or words, where the length of the code is directly related with the frequencies of the characters. In this logic, the most frequent character or word is assigned with the smaller code, while the least frequent one gets the largest code. Huffman is a zero-order encoder, meaning that it is a compressor which encodes each symbol regardless of the preceding ones, which makes it work well with a biased distribution of word frequencies.

Other encoding techniques, such as Plain Huffman [133] and Restricted Prefix Byte Codes (RPBC) [42], enable compression ratios around 30%, but a considerable faster decompression. These two approaches differ from Huffman encoding, by using sequences of byte codewords

instead of bits. In fact, that is the main difference between Plain Huffman and the classical version of the encoding. However, as there is no need for manipulating bits, the decompression is much faster. RPBC uses the same Huffman codes as Plain Huffman, but with the first byte of each codeword specifying the number of bytes of the codeword.

Finally, coding techniques such as Tagged Huffman [133], End-Tagged Dense Codes (ETDC) and (s,c)-Dense Codes [21] require 35% of the text size [21], but with the compressed text being self-synchronized. Self-synchronization allows a search to start from a given position of the text, or filter a search from a point in the text, instead of always starting from the beginning. ETDC uses the first bit of each byte to determine if the byte is the last one of its codeword. This makes this encoding dense, as the remaining seven bits of each byte can be used for the combinations of words in the text. This representation enables skipping bytes [19], while searching over the compressed text, which is something that Huffman encoding does not allow directly. Despite the worst compression ratios, self-synchronization is a decisive feature for supporting searches limited to a particular part of a corpus, like a text or paragraph. This leads to a better efficiency in searching and decompressing text, when compared to Plain Huffman or RPBC, as in those cases both procedures would always need to start from the beginning of the text.

3.3.2.1 Byte-Codes Wavelet Tree

Brisaboa et. al. [22] proposed a reordering of the bytes in the codewords of the compressed text, to follow a wavelet tree-like [78] structure, instead of having a sequential representation of the compressed text. This representation enables the compressed text to be always self-synchronized, regardless the encoding used. This is extremely helpful, as the most efficient bytewise encodings can be used, without losing the self-synchronization property. Additionally, this reorganization in a wavelet tree, despite not being technically an index, has some implicit indexing properties, such as the logarithmic search over the compressed text. These representations of the compressed text is known as Byte-Codes Wavelet Tree.

The Byte-codes Wavelet Tree is a multi-ary wavelet tree in which the bytes of the codewords are organized in different nodes, throughout the several levels of the tree. The codewords follow the same principle introduced before. Every word in the text is represented by a codeword of bytes, with the sequence of those words being the compressed text. The codeword for frequent words are smaller than the codewords for the less frequent words. With this strategy, the representation saves space, as the most frequent words take less space.

As the bytes of the codewords are divided through the several levels of the tree, with one byte per level, the tree representation has as many levels as the largest codeword of the compressed text. The first level of the tree, or the root, indexes the first byte of all the codewords of the compressed text. This representation follows the order of the text, with the position i of the root representing the first byte of the codeword of the i^{th} word in the text. The second level of the tree has one node per every unique first byte in the root, representing all the second bytes of the codewords throughout the nodes in the level. Thus, a node $Node_x$ in the second level stores the second bytes of all the codewords whose first byte is x. The same behavior is applied for the

remaining levels of the tree, until reaching the leaves, where the last bytes of the codewords are represented. For instance, on the third level of the tree representation, $Node_{xy}$ stores all the third bytes of the codewords that have x as first byte and y as second byte, and so on for the remaining levels of the tree. This reorganization maintains the order of the words in the text in every node, even in deeper levels of the tree.

Figure 3.6 shows the structure of a Byte-codes Wavelet Tree for the text `to be or not to be that is the question`. Every word in the text has a different codeword, with the more frequent words having smaller codewords, where $b_1 ... b_5$ represent different bytes. For example, the word `question` has the codeword $b_4 b_5 b_2$, while the most frequent word `to` has codeword b_1.

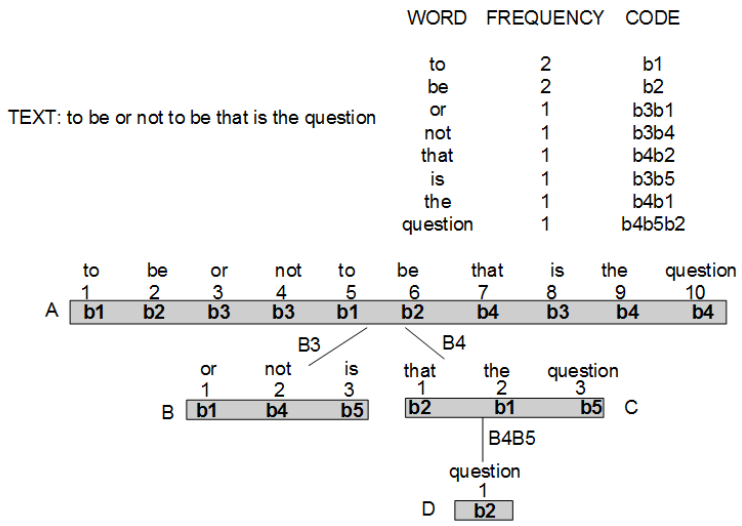

Figure 3.6: Example of a Byte-codes Wavelet Tree

To represent the codeword for `question`, the root node indexes byte b_4 at position 10, since the word appears in that position in the indexed text. Following the branch b_4, one gets to node C, with all the second bytes of the codewords starting with byte b_4. For the particular case of the word `question`, the byte b_5 is indexed in node C, at position 3, the last one of the node, as it is the last word in the text. The same strategy is visible in the following levels. Continuing through branch $b_4 b_5$, the 3rd byte of the word `question`, b_2, appear in D alongside all the third bytes of the codewords that start with $b_4 b_5$. Here, it is the only byte in there as the codeword for `question` is the only one that starts with $b_4 b_5$.

Considering this reorganization of the compressed text and its bytes, all the occurrences of a word in the text are represented in the node of the tree structure that stores the last byte of the respective codeword. Recovering the example of the word `question` in Figure 3.6, node D represents the last bytes of its codeword, meaning that `question` has one occurrence in the text. This does not mean that node D will only have the representation of all the occurrences of `question`, neither that only a leaf has all the occurrences of a word in the text. Node C represents the last bytes of the codeword $b_4 b_1$, for the word `the`, but also the second bytes of all the codewords starting with b_4, and it is not a leaf.

On the other hand, the absolute location of the occurrences in the text are codified in the root of the tree representation, as it is the node that has one byte, the first, of all the words in the text. Besides this, the representation of the bytes follows the order the respective words appear in the text, thus it is possible to determine where an occurrence of a word appears in the text.

This reorganization produces codeword boundaries that enable self-synchronization, regardless the encoding used. Such characteristic allows using the most efficient compressors, such as Plain Huffman or RPBC, which as they are not self-synchronized, both are more space efficient than ETDC for instance. With the self-synchronization present from the reorganization, the less efficiency on searching or decompressing is now solved, thus one can benefit for their less space requirements.

This Byte-codes Wavelet Tree is the compressed approach that I implemented for supporting the Text Layer of the presented Bilingual Framework. To implement the Text Layer based on this index, the structure **t_index** must be defined, just like in the Word-based Suffix Arrays alternative. Figure 3.7 shows my implementation of the structure.

The main structure holds the total bytes (*total_bytes*) that the index takes and the number of tokens of the indexed text (*nr_tokens*). The structure has two additional fields: a vocabulary (*voc*) and the tree topology (*tree*).

```
struct vocabulary{
    char* words;
    int word_size;
    int* word_pointers;
    int nr_diff_words;
    int* sorted_words;
}

struct st_tree{
    int max_level;
    int min_level;
    struct bytemap **bmaps;
    int nr_bmaps;
    int* nr_bmaps_per_level;
    int* start_level;
    int* bmaps_start_offs;
}

struct t_index{
    struct st_tree *tree;
    int total_bytes;
    int nr_tokens;
    struct vocabulary *voc;
}
```

Figure 3.7: **t_index** structure for Byte-codes Wavelet Tree

The vocabulary represents all the unique words of the text, being useful for building the index, but also to extract the actual words from the respective codewords, when necessary. The **vocabulary** structure holds the actual words of the text and the number of different words in the text (*nr_diff_words*). As the words are all represented together, *word_size* holds the largest size

possible for a word in *words*. Finally, the structure holds the pointers for the sequence *words* where each word starts, at field *word_pointers*, and *sorted_words* that holds the location of every word in *word_pointers*, sorted by frequency.

This vocabulary representation simplifies the representation of all the words in the indexed text, by having them all together, with no repetition, in a large sequence of words, then saving the pointers to those words in a separate field. Then, having the words sorted, enables discovering faster in a search, if the word of the searched term exists in the indexed text.

The **st_tree** structure holds the bytemaps where the codewords are rearranged, and how many they are (*nr_bmaps*), alongside the maximum and minimum level of the tree representation. The bytemaps are not divided through nodes, but all the bytes are represented consecutively. The next three fields help defining the tree structure. The first one, *nr_bmaps_per_level*, define the number of bytemaps that every level has, with the root having one bytemap. The second one, *start_level*, represent when each level of the tree begins, in the *bmaps* sequence. Finally, the last one determines where every bytemap, or every node, starts in *bmaps*.

The procedure to create the index is divided in two main steps, with the first one focusing on creating the codes, and the second one based on creating the tree topology, using the codewords. To generate the codes for every word, one must first determine the frequencies of the words in the text. It is important to recover here that the words with higher frequency have smaller codewords, to save space.

To determine the frequency of every word in the text, the creation procedure uses a hash table, with the word as key and its frequency and codeword as value. For every word in the text, the procedure searches for it in the hash table. If it exists, increases its frequency, if it does not exist, creates a new entry in the table for it. At the end, the table is sorted by the frequency of the words. Then, considering their frequency, the codes for every word are generated using Plain Huffman. Finally, the bytemaps are created, defining the tree topology of the structure, with the bytes of the codewords populating the bytemaps[2].

Algorithm 3.3 shows a simplified version of the creation procedure for Byte-codes Wavelet Tree. As explained before, the procedure uses a hash table that stores the frequency and later the codeword for every word in the text. In the first step, the frequency of every word is calculated. After that, the hash table is sorted taking into account the frequency of the words, using the number of unique words *n_words*. The last three methods represent the final steps of the creation procedure. First, the codes are generated taking into account the frequencies of the words, creating the bytemap with all the bytes of all the codewords. This procedure is followed by building the tree structure, using the same strategy presented in the structure **t_index**, with a representation of where each level and each bytemap per node starts. We use the known **bytemap** and **st_tree** structures already used for **t_index**. Finally, the vocabulary is built, just before all the data is stored in disk, making it available for loading later to main memory, populating the index structure.

To locate a word or multi-word term in the text, or counting its frequency, involves $ascend$

[2]The explanation for this algorithm is extensive. For more information check the work by Brisaboa et. al. [22]

```
void create_index(char* outfile, char* text, int text_len, struct t_api *api){
  int nr_words = 0, ht_pos = 0;
  hash_table ht;
  struct vocabulary *voc;
  for(every token in text){
    ht_pos = get_value_from_key(ht, token);
    if(ht_pos == -1){
      ht_pos = add_new_entry(ht, token);
      ht[ht_pos].freq = 0;
      voc->nr_diff_words +=1;
    }
    ht[ht_pos].freq+=1;
  }
  sort_by_frequency_ascending(ht, nr_words);
  struct bytemap** bmaps = generate_codes(ht, nr_words);
  struct st_tree* tree = build_tree(corpus, len(corpus), ht, nr_words, bmaps);
  build_vocabulary(voc);
  save_to_disk(outfile, tree, bmaps, voc, ht);
}
```

Listing 3.3: *create_index* function based on Byte-codes Wavelet Trees

and *descend* the tree representation, using the bytes of the codeword(s) of the term. Considering just a term with one word, the number of its occurrences in the text is the number of the last bytes of its codeword represented in the respective final node. This requires accessing the mentioned node, by using the tree structure mentioned above, and $count$ the number of last bytes of the respective codeword. For example, the word `the` in Figure 3.6 has one occurrence, with node C having just one byte b_1, the second byte of its codeword.

However, to determine the location of a word in the text, it is necessary to *ascend* the tree, from the lower levels to the root, as it is in the first level of the tree structure that the absolute offsets of each word are codified. Consider again the word `the` in Figure 3.6. To determine the only occurrence of the word in the text, the procedure starts at node C, the one with the last byte of its respective codewords. Byte b_1 appears at position two of node C, thus a $select$ for the second b_4, the first byte of the codeword, on node A is made, determining the ocurrence of `the`, at position nine of the text.

In the next Chapter, I describe the search procedures for the compressed Text Layer, with details on how the functionalities offered by the Byte-codes Wavelet Tree help support that.

3.4 Alignment Layer

The Text Layer, introduced in the previous Sections, indexes in main memory the texts, in the two languages, of the parallel corpora represented in the Bilingual Framework. As these texts are indeed parallel, they are actual translations of each other, which is a rich source for information, such as co-occurrence statistics. For that purpose, the Bilingual Framework supports several queries to extract information, not only from the texts, but also from the relationship between them, namely what is translated by what. This leads to the importance of representing which segments of these texts are translations of one another, when that information is available.

Namely, the parallel text alignment needs to be indexed in the Bilingual Framework, to relate the texts together. This way, the alignment between the segments of the texts is represented in the Alignment Layer.

The Alignment Layer supports two granularities of alignment: the coarse alignment, based on sentences, paragraphs or titles, and the fine alignment, based on word or multi-word sub-sentence segments. Its main purpose lies on determining in which segments of the alignment, either coarse or fine, the occurrences of a term, previously determined using the Text Layer, appear. Figure 3.8 shows the API for the Alignment Layer, with all its functionalities and structures.

```
struct si_bounds{
    int begin_offs[2];
    int end_offs[2];
    int begin_fine;
    int end_fine;
}

void create_index (char* outfile, char* text,
    int text_len);

struct si_index* load_index(char* infile, char*
    langs[2]);

void free_index(struct si_index* index);

void locate (struct si_index *index, int off,
    int side, int* coarse, int* fine);

int[2] fine_begin (struct si_index *index, int
    fine);

int fine_begin_one_side (struct si_index
   *index, int fine, int side);

int[2] fine_end (struct si_index *index, int
    fine);

int fine_end_one_side (struct si_index *index,
   int fine, int side);

void coarse_bounds_one_side(struct si_index*
    index, int coarse, int side, struct
    si_bounds* bounds);

void coarse_bounds (struct si_index *index, int
    coarse, struct si_bounds *bounds);
```

Figure 3.8: Alignment Layer API

Just like in the Text Layer, this API also has functions for creating, loading and releasing the indexes supporting the Alignment Layer. Those functions receive as input parameter the generic structure **si_index**, not defined at this stage, because its definition will depend on the implementation used for the Alignment Layer. This thesis presents two different approaches for supporting the parallel corpora alignment: 1) an implementation based on Arrays of Integers that was experimented as first approach; 2) an implementation based on bitmaps, that I created to reduce the space consumption of the Alignment Layer.

The following function, *locate*, determines the fine and coarse segments where an occurrence (*off*) appears, for one of the two languages of the parallel corpora (*side*). The following functions of the API determine the bounds of the fine or coarse segments of the alignment, namely in which absolute offset of the corpora they start and end. The functions *fine_begin* and *fine_end* return the offsets where the fine segment starts and ends, for both languages of the parallel corpora, while *fine_begin_one_side* and *fine_end_one_side* do the same but for just one *side* of the parallel corpora. The same strategy is applied to the coarse segments of the alignment, with the functions *coarse_bounds* and *coarse_bounds_one_side* returning the offsets where the segments begin and end.

To hold this information, the structure **si_bounds** was created with the beginning and ending offsets of the coarse segment, in both languages. Additionally, the structure holds the first and last fine segments within the coarse segment represented.

3.4.1 Arrays of Integers

A first approach to represent the alignment is based on Arrays of Integers. This approach is simple, as it focus on using standard arrays, in which every position of the array represents an offset of where the alignment segment starts. To represent the two granularities, the Alignment Layer uses several arrays: one per language to represent the coarse alignment, one per language for the fine alignment and one to relate the two granularities.

For the coarse alignment, the Alignment Layer uses two arrays, one for each language, with both following the exact same behavior. Each position of these two arrays stores the word-based offset, where the segments begin in the corpora. This means that the first position of the array has the starting position of the first coarse segment, the second has the starting position of the following segment, and so on.

The fine alignment is represented in a similar fashion, using as well two Arrays of Integers, but here, every position in the arrays consider the word-based offset within a coarse segment, instead of the absolute offset. This means that the first fine segment within a coarse segment, has always offset 0, as it is the first, while the second's starting offset is equal to the word-based length of the first. When a coarse segment ends and another one begins, the process restarts and a new fine segment with offset 0 is added. The reason for choosing this non direct representation, in opposition to use the absolute offset, lies on the high number of fine segments available and the possible sizes that the corpora may have. It is important to notice that a fine segment can be as small as a word or a couple of words, thus the number of fine segments in a huge corpora can also easily be huge. Besides that, an absolute offset in such corpora can have a very large integer (or long) number. Thus, representing such amount of fine segments, with such large numbers, would lead to unnecessary high space requirements.

This strategy enables a two byte representation for integers, which goes from 0 to 65535, for the fine alignment arrays, as the considered coarse segments do not span for more than this number of words. This enables saving space, especially considering the alternative of using eight bytes per integer, for representing absolute offsets. For the coarse alignment, this

scenario does not raise a problem, since there are considerable more fine segments than coarse segments. Thus, the coarse alignment representation can use larger integers, to represent the absolute offsets. Figure 3.9 shows an example of the bitmaps for the alignment presented in Section 2.4.1, in Figure 2.8 and Figure 2.9.

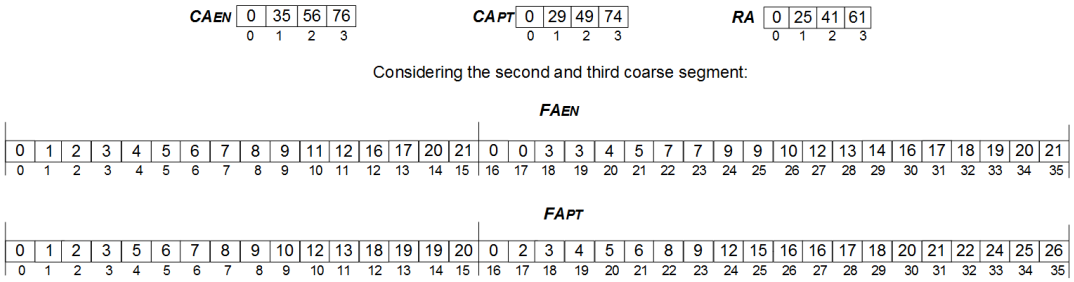

Figure 3.9: Alignment Layer supported by Arrays of Integers

In Figure 3.9, CA_{EN} and CA_{PT} represent the arrays for the coarse alignment and FA_{EN} and FA_{PT} the arrays for the fine alignment. CA_{EN} has four positions, the three coarse segments of the alignment, plus a fourth one to mark the end of the last segment. The first segment starts at position 0, while the second starts at position 35, meaning that the first segment has 35 words[3]. For the fine alignment, FA_{EN} has one position per fine segment, starting as well at position 0. As the first segment has a length of 1, the second segment starts at offset 1, thus $FA_{EN}[1] = 1$. The second segment has also length 1, resulting in $FA_{EN}[2] = 2$, and so on. Notice that for space purposes, only the fine segments for the second and third coarse segments are represented.

As mentioned earlier, at the start of every coarse segment, the fine segment offsets restart at 0. This occurs at the 16th segment, where $FA_{EN}[16] = 0$, meaning all the fine segments of the 2nd coarse segment are already represented, and the segments for the 3rd one will start from that position onwards. This first segment happens to be empty, thus having length 0, as one can notice in Figure 2.9. This approach based on the arrays tackle those situations, by maintaining the same offset for the following segment, thus leading to $FA_{EN}[17] = 0$. Then, $FA_{EN}[18] = 3$, as the second fine segment of the third coarse segments has a length of three words. This same logic is followed also for CA_{PT} and FA_{PT}.

With the fine segment representation depending on the coarse alignment, it is necessary to link the two alignment granularities. In the context of this work, the considered text alignments have the same number of coarse and fine segments in both languages, thus $|CA_{EN}| = |CA_{PT}|$ and $|FA_{EN}| = |FA_{PT}|$[4]. Taking this into account, the Alignment Layer has an extra array, RA, that relates the fine segments with the coarse segments. Each position of RA indicates in which fine segment of the alignment, the new coarse segment starts. In Figure 3.9, $RA[1] = 25$ and $RA[2] = 41$, meaning that the second coarse segment starts at the 25th fine segment and ends at the 40th, with the third coarse segment starting at the 41st fine segment of the alignment. The arrays for the fine alignment demonstrate that, since the second coarse segment has 16 fine segments and the third 20, with the first segment not being represented (with 25 fine segments).

[3] Do not forget that we consider punctuation marks and other tokens for words.
[4] Empty segments also count as valid segments.

This makes *RA* the connection between the two granularities, being essential for searching over the Alignment Layer, which will be described in detail in the next Chapter.

Figure 3.10 shows the structure of the Alignment Layer approach based on arrays of integers. This structure follows the previous explanation with two arrays for the coarse alignment (*coarse_offs1* and *coarse_offs2*), two arrays for the fine alignment (*fine_offs1* and *fine_offs2*) and the array *RA* (*r_array*). Besides these arrays, the structure holds the number of fine (*nfine*) and coarse (*ncoarse*) segments.

```
struct s_index{
    int* coarse_offs1;
    int* fine_offs1;
    int* coarse_offs2;
    int* fine_offs2;
    int* r_array;
    int nfine;
    int ncoarse;
}
```

Figure 3.10: Structure **s_index** for the Alignment Layer based on Arrays of Integers

The construction of these arrays follow the order of the text, thus the segments of the alignment in CA_{EN} and CA_{PT} follow an ascending order, meaning that every position forward in the array has an higher absolute offset. To support this logic, the arrays are built progressively using every pair of aligned fine segments, through the processing of the parallel corpora to be indexed by the Bilingual Framework.

For every pair of fine segments, their start offsets are added to *coarse_offs1* and *coarse_offs2*, if the fine segments correspond to the start of a new coarse segment. Then, consider *off_1* and *off_2* as the start offsets of the fine segments and *current_coarse_off_1* and *current_coarse_off_2* as the offsets of the beginning of the current coarse segment being processed. The values *off_1* - *current_coarse_off_1* and *off_2* - *current_coarse_off_2* are then added to the arrays *fine_offs1* and *fine_offs2*, to represent the fine segments. These offsets are relative to the beginning of the coarse segment they are on, instead of the absolute ones.

The last step of this procedure lies on populating the extra array that relates the fine with the coarse granularity. For every pair of fine segments in the corpora, a number is incremented to represent the number of aligned fine segments being processed. Then, when the procedure is in a situation of the beginning of a new coarse segment, the accumulated number is added to *r_array*, defining the number of fine segments between this new coarse segment and the last.

Note that the explanation above just pretends to explain how the procedure is done. In practice, the offsets are not added immediately to the arrays. In fact, the arrays are only created when the indexes are loaded to main memory. On the creation process, the data is stored directly in disk. Algorithm 3.4 shows a simplified version of this procedure. In this procedure, the *add_offset* function represents the addition of a new offset to a given array. Besides this,

the values *off_1* and *off_2* represent the word-based absolute offsets of the fine segments, *current_coarse_off_1* and *current_coarse_off_2* the starting offset of the previous coarse segment and *rel_off* the number used to populate the extra *r_array*.

```
void create_index(char* outfile, char* text, int text_len){
  int off_1 = 0;
  int off_2 = 0;
  int current_coarse_off_1 = 0;
  int current_coarse_off_2 = 0;
  int rel_off = 0;
  for(every pair of fine segments in the parallel corpora seg_1 and seg_2){
    if(new coarse segment){
      add_offset(index->coarse_offs1, off_1);
      current_coarse_off_1 = off_1;
      add_offset(index->coarse_offs2, off_2);
      current_coarse_off_2 = off_2;
      add_offset(index->r_array, rel_off);
    }
    add_offset(index->fine_offs1, off_1 - current_coarse_off_1);
    add_offset(index->fine_offs2, off_2 - last_coarse_off_2);
    rel_off +=1;
    off_1 += word_len(seg_1);
    off_2 += word_len(seg_2);
  }
}
```

Listing 3.4: *create_index* for the Alignment Layer supported by Arrays of Integers

3.4.2 Bitmaps

Representing the offsets may require a considerable amount of space, in particular for huge texts, with a lot of coarse, and specially fine segments. Similarly to what was done with the Text Layer, we designed an alternative that requires less space, by using bits, instead of integers, to represent the alignment.

The second approach for implementing the Alignment Layer is based on bitmaps. Similarly to the previous approach based on arrays, the bitmaps also support the coarse and fine alignment. For enabling this, the Alignment Layer consists of two bitmaps, one per language, to represent the fine alignment, and one to represent the coarse granularity.

To represent the fine alignment, each of the bitmaps use the same strategy. For every new fine segment in the corpora, a bit 1 is added to the bitmaps to represent its beginning. This bit is then followed by w bits 0, with w being length of the fine segment in terms of words. If the fine segment is empty, then no bits 0 are added. This process is repeated until the end of the fine segments, ending as well with a bit 1 to signal the end of the last fine segment. The same process is repeated for the bitmap that indexes the fine segments in the other language.

For the coarse segment, the process is similar, but instead of using the length of the segment in words, it uses the length in fine segments. This way, for every new coarse segment, a bit 1 is added to the bitmap representing its beginning, just like for the fine alignment. Then, the bit is followed by f bits 0, with f being the number of fine segments within the coarse segment.

Using the fine segments makes the representation of the coarse segments more space efficient. As there are much less fine segments than words, and one bit is used per word / segment, representing the coarse segments based on the fine segments leads to a smaller bitmap. With this representation, and since the parallel corpora has the same number of coarse segments, only one bitmap is needed to index this granularity.

Figure 3.11 shows an example of the bitmaps, following the same alignment than Figure 3.9, with the fine-grained bitmaps only mapping to the second and third coarse segments.

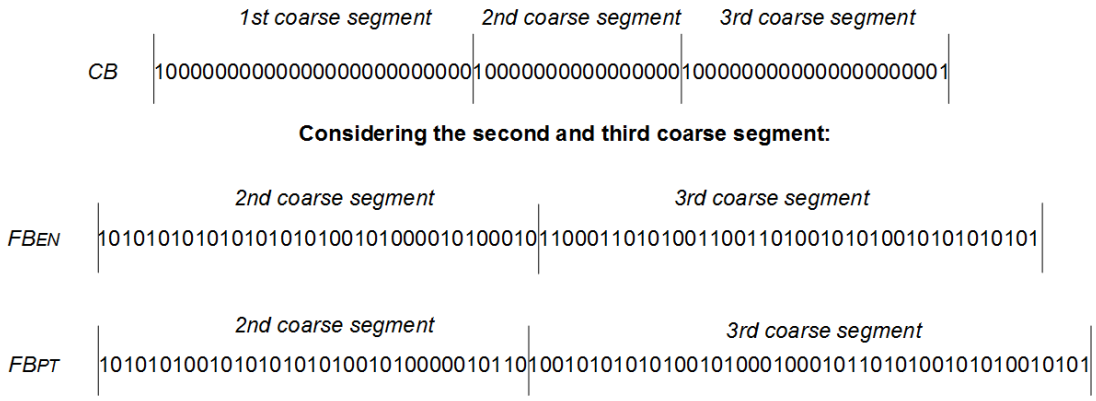

Figure 3.11: Alignment Layer supported by Bitmaps

In Figure 3.11, CB represents the bitmap for the coarse alignment and FB_{EN} and FB_{PT} the bitmaps for the fine alignment. FB_{EN} represents a first fine segment with one word, as there is a first bit 1 followed by a bit 0. The second fine segment has as well one word, and so on. Two consecutive bits 1 in the bitmap indicate an empty segment, or a fine segment with no words, thus no bits 0 are added. With CB the logic is similar but with each bit 0 representing a fine segment. With this logic, the second coarse segment, after the second bit 1 in the bitmap, has 17 fine segments which are represented by seventeen bits 0.

This representation does not require additional bitmaps, like the approach based on arrays does, as CB is already based on fine segments. This way, one can determine in which fine segment an occurrence exists, and then use that segment to determine the respective coarse segment, in CB. This is done by using *rank* and *select* operations over the bitmaps [73].

Figure 3.12 shows the definition of the structure **s_index** for the Alignment Layer supported by bitmaps. The structure is simple and follows the aforementioned description, with the three bitmaps *fine1*, *fine2* and *coarse*, to represent the fine and coarse granularities. Additionally, the structure keeps the number of words in both *fine* bitmaps (*nwords1* and *nwords2*), the number of fine segments (*nfine*) and the number of coarse segments (*ncoarse*).

To build the Alignment Layer based on Bitmaps, the strategy is similar to the one used for the Arrays of Integers. The bitmaps are built progressively, by using every pair of aligned fine segments of the parallel corpora. Also similarly to the approach based on arrays of integers, the bitmaps are not built directly when creating the structure for the Alignment Layer. The data is directly stored to disk and the structure itself is rebuild when loading the Bilingual Framework to main memory.

```
struct s_index{
    int* fine1;
    int nwords1;
    int* fine2;
    int nwords2;
    int* coarse;
    int nfine;
    int ncoarse;
}
```

Figure 3.12: Structure **s_index** for the Alignment Layer based on Bitmaps

For every fine segment processed from the parallel corpora, the bitmaps *fine1* and *fine* are populated with a number of zeros that correspond to the word-based length of the segments. Consider the offsets *off1* and *off2* as the starting offsets of two aligned fine segments. Bitmap *fine1* is populated with *off1* - *lastoff1* zeros, with *lastoff1* being the starting offset of the previous fine segment processed. The same logic is applied to the other language and therefore *fine2*. After all the zeros are added, a bit one is added to mark the end of the fine segment and the beginning of a new one, if there are more.

Additionally, after updating the *fine* bitmaps, a bit zero is added to the *coarse* bitmap, meaning that the current coarse segments have one more fine segment. By doing this for every aligned pair in the corpora, all the fine segments inside the coarse segment are accounted for. Whenever a new coarse segment starts, meaning that its first pair of fine segments are being processed, a bit one is added to *coarse*. This bit represents the end of a coarse segment and the beginning of a new one, if there are more.

Algorithm 3.5 shows an overview of the procedure described above, with *off1* and *off2* being the starting offset of the two fine segments, and *lastoff1* and *lastoff2* the offset of the previous two processed segments. For the first pair of segments, a bit one is added to every bitmap, with the function *bitmap_push*. Note that the bitmaps are not yet created, so the bits are added directly to disk. For the remaining pair of segments, the same *bitmap_push* function is used to represent the adding of a bit to the respective bitmap.

In these last two Sections, I presented the two alternatives for the Text Layer and Alignment Layer, with focus on their structures and creation algorithms. Later in this Chapter, I present the memory consumption results for these four approaches. For the search time response of the Framework, the next Chapter presents all the results, after explaining the monolingual and bilingual search procedures.

3.5 Document Layer

As explained before, the Text and Alignment Layers are used together to correctly determine the occurrences and co-occurrences of a term, or pair of terms, in the indexed parallel corpora. The Alignment Layer not only helps filtering the calculation of the occurrences of a term in the Text

```
void create_index(char* outfile, char* text, int text_len){
  int off1 = 0;
  int off2 = 0;
  int lastoff1 = 0;
  int lastoff2 = 0;
  for(every pair of fine segments in the parallel corpora seg_1 and seg_2){
    if(off1 ==0 || off2 = 0){
      bitmap_push(index->fine1, 1);
      bitmap_push(index->fine2, 1);
      if(new coarse segment){
        bitmap_push(index->coarse, 1);
      }
    }else{
      while(lastoff1 < off1){
        bitmap_push(index->fine1, 0);
        lastoff1++;
      }
      while(lastoff2 < off2){
        bitmap_push(index->fine2, 0);
        lastoff2++;
      }
      bitmap_push(index->fine1, 1);
      bitmap_push(index->fine2, 1);
      bitmap_push(index->coarse, 0);
      if(new coarse segment){
        bitmap_push(index->coarse, 1);
      }
    }
  }
}
```

Listing 3.5: *create_index* for the Alignment Layer supported by Bitmaps

Layer, as it also determines the position in the alignment of that occurrence, using the offset returned in the Text Layer.

However, by indexing parallel corpora, the Bilingual Framework may represent a set of several parallel texts, or documents, in both supported languages. These documents do not require to be necessarily stored in the same disk path, meaning that the documents must be stored in different directories, in secondary storage. Taking this scenario in consideration, to filter a search to a certain document, or directory with a set of documents in it, instead of the whole corpora, the Text and Alignment Layer do not suffice. Even if one wants to obtain information about a particular document (or documents), there is not enough knowledge represented in the existing Layers to accomplish that.

To tackle this limitation, we designed a Document Layer to emulate the secondary storage location of the texts that are part of the indexed parallel corpora. Consider a “tree"of directories, with a root directory *Dir_A* that has three other directories inside it, *Dir_B*, *Dir_C* and *Dir_D*, and several documents divided through these different directories. To represent a structure similar to this in main memory, we used the length of the documents, namely their number of coarse segments.

Consider Figure 3.13 with the four aforementioned directories and ten documents inside

them, alongside the length of every document on the right. With such structure of directories and documents, we defined arrays of integers to represent the relative location of every document, and directory, starting from the root directory, and following a lexicographic order. Every position in the arrays will have the first coarse segments of the documents, and directories, following the lexicographic order, meaning that the document / directory with the lexicographically lower name will be represented first in the arrays.

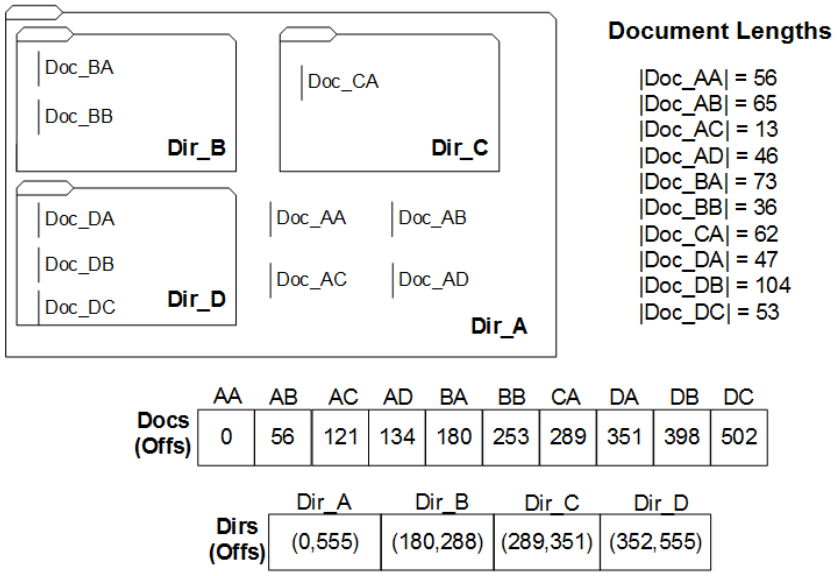

Figure 3.13: Document Layer example

Considering the example in Figure 3.13, array **Docs** represent the documents location in the path of directories. Doc_{AA} is the first document, following the lexicographic order, in the root directory, Dir_A, thus the first position of **Docs** represents the start of Doc_{AA} at coarse segment 0. Then, the second document, Doc_{AB}, is represented in the second position of the array, with offset 56, meaning that the 56th coarse segment of the corpora is the first one of this document. This representation considers that the next document starts immediately after the previous one ends. Here, Doc_{AB} starts after the last offset of Doc_{AA}. The same logic is followed for the remaining documents in the root directory.

After all the documents in a directory are represented in the array, the same logic is applied to the remaining subdirectories, and their documents, following as well the same lexicographic order. Dir_A has three directories in it, Dir_B, Dir_C and Dir_D. With all the documents in Dir_A already processed, the next ones in **Docs** are Doc_{BA} and Doc_{BB}. Doc_{BA}'s offset is the accumulated length of all the documents, in terms of coarse segments, in Dir_A, thus $|Doc_{AA}|+|Doc_{AB}|+|Doc_{AC}|+|Doc_{AD}| = 56+65+13+46 = 180$. The same logic is applied when afterwards for the documents in Dir_C and Dir_D, until all the documents are represented.

Besides representing the documents, the Document Layer also uses arrays of integers to represent the directories. The representation of the directories follows the same strategy used for the documents. The main difference lies on the information stored in the arrays. For the directories, the starting offset are not enough, as it is important to determine when a directory

ends. For the documents, that info is not necessary, because the document is the lowest unit for this Layer. For the directories, it would not be possible to know when a directory starts or ends, just with their starting offsets.

Starting from the root directory, Dir_A, the first position in array **Dirs** has the offset 0, as it is the root directory, and the offset 555 to represent the end of the directory. The ending offset at 555 is the total length of all the documents in this example. Being the root directory, Dir_A only “ends"at the last offset of the last document of the last subdirectory, following the lexicographic order. The same strategy is followed for the next directories. Dir_B starts when the first document inside it starts as well, thus offset 180, and ends at offset $180+|Doc_{BA}|+|Doc_{BB}| = 288$. The ending offset this time only considers the documents inside Dir_B as no subdirectory exists inside it. The remaining directories follow the same strategy, always considering the coarse segments.

Figure 3.14 shows the set of functions that the Document Layer offers, divided in three major types: 1) creating / loading / destroying the Layer; 2) finding the document / directory where an offset appears; 3) determine the disk path of a document / directory.

```
struct di_index_builder* new_builder(char*
    root, char* langs[2]);

void builder_add_doc(struct di_index_builder*
    builder, int coarse, char* docpath);

void builder_close(struct di_index_builder*
    builder, int coarse);

struct di_index* load_index(char* root, char*
    langs[2]);

void free_index(struct di_index *index);

void locate(struct di_index *index, int coarse,
    int* dir, int* doc);

void doc_bounds(struct di_index *index, int
    doc, int *begin, int *end);

void dir_bounds(struct di_index *index, int
    dir, int *begin, int *end);

char* get_docname(struct di_index *index, int
    doc);

char* get_dirname(struct di_index *index, int
    dir);
```

Figure 3.14: Document Layer API

The first set of functions handle the creation and loading of the structure that supports the Document Layer. While processing the corpora to be indexed, the Bilingual Framework consumes one document or parallel text at a time. This leads to using a continuous creation workflow, in which a **builder** is used, to store one document at a time in the Layer. The three main functionalities of the Layer are: *new_builder*, to create the **builder** for the parallel corpora

under the directory *root*; *builder_add_doc* to add a new document with coarse segment $coarse$ to the Layer; and *builder_close* to end the creation procedure, closing with the last segment ($coarse$) of the last document under *root*.

The following function, *locate*, determines the directory (*dir*) and document (*doc*) of the given $coarse$ segment. Then, by giving a certain document, *doc_bounds* determines the *begin* and *end* absolute offset boundaries, within the whole parallel corpora, while *dir_bounds* determines the same boundaries, but for a giving directory *dir*.

Finally, the last set of functions focus on obtaining the path of a document or directory, under the root folder. Given a document *doc* or a directory *dir*, the Layer returns the complete path name to that directory or document. This functionality is used to obtain info about a document, or directory, where an occurrence appears, within the parallel corpora.

Knowing the concept about this Layer, the structure **d_index** in Figure 3.15, has the index representation, with all the data structures necessary to support it. The structure starts with *dir_begin_offs* and *dir_end_offs* to represent the starting and ending offsets of the directories, while the array *doc_offs* represents the offsets for the documents, as introduced in Figure 3.13. For representing the paths and the names of the documents, the structure has a field *names*, with all the names of documents and directories concatenated together. The structure has three additional sequences of integers, *dir_path_pointers* and *doc_name_pointers*, to point to the names of the directories and documents in *names*, and *paths* for the beginning of a path.

```
struct d_index{
    int* dir_begin_offs;
    int* dir_end_offs;
    int nr_dirs;
    int* doc_offs;
    int nr_docs;
    int* dir_path_pointers;
    int nr_dir_path;
    int* doc_name_pointers;
    int nr_doc_names;
    int* paths;
    int nr_paths;
    char* names;
    int nr_names;
}
```

Figure 3.15: Document Index

To create the Document Layer, there are some additional structures that the **builder** structure has, an hash table and a stack, that help in two particular important steps of the creation. The **builder** approach forces the creation procedure to have an iterative behavior, thus indexing one document at a time, while they are being processed on creating the Framework.

For every document processed, the first step is to process the document that is being indexed. The name of the document, which includes the whole path, is split and then added, if not there already, to the string of names (the field *names* in the **di_index** structure). Also the new path is added to the *paths* field and the pointer for the location of this document in *paths* is

added to the field *doc_name_pointers*. This is done with the help of the hash table, which holds all the pointers to every document or directory indexed. If it already exists, the same pointer is used, otherwise a new one is created and added to the hash table.

Then, the second step focuses on placing this document within the directory it belongs. The **builder** structure holds a stack that helps supporting this behavior. Whenever a document is being processed, since the Document Layer uses the whole path, the document name also includes the parent directories. This means that we know for every document, its exact placement in disk. For that matter, the parent directories are pushed into a stack. When a new document comes, if it is in the same directory, the document is just added, without any change. If it is in a different directory, then the directories are popped from the stack, and the new directory is pushed. Figure 3.16 shows an example of this procedure, recapping the directories and documents presented in Figure 3.13. When Doc_{CA} is added, all the documents inside Dir_B have been processed already. The path for Doc_{CA} includes Dir_C and Dir_A as parent directories, thus Dir_C is popped from the stack and Dir_C is added. This same behavior is applied for every depth of path. When a directory is popped, the field *dir_end_offs* is filled with the ending offset. In opposite direction, when a directory is pushed to the stack, the field *dir_begin_offs* is filled with the beginning offset.

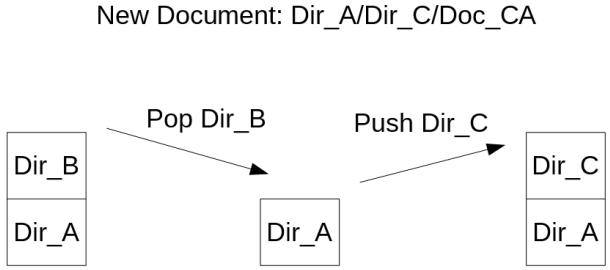

Figure 3.16: Using a stack to build the Document Layer

Now knowing the behavior supported by the stack, it is understandable why the *builder_close* function has a *coarse* parameter. This is actual the last coarse segment of the whole corpora, and it is going to be used in the *dir_end_offs* field, namely to “close"all the directories that are still in the stack. For instance, in Figure 3.13, Dir_D and Dir_A end exactly at the same offset, because there are no more directories inside Dir_A, after Dir_D.

For the Document Layer, there is no listing of the creation algorithm, as it is complex to present here fully, and would become non helpful if listed too generically. The three functions used to create the Layer write directly in disk the info relative to the fields that are later loaded for the structure **di_index**, including the pointers to the actual names of the documents.

This indexing of the documents and directories in main memory, alongside the texts and alignment, have the objective of enabling the extraction of information about the texts that are part of the corpora. One example is the extraction of the actual name and path where the text is represented in disk.

The functionalities of the Bilingual Framework are based on these three Layers, in particular

to the way they work together to achieve the desired results. These Layers are not only a source of information, but they also can to filter the scope of a search, if desired. For instance, we can have a search for a term, or pair of terms in two languages, that actually is filtered to a paragraph, or to a set of texts inside a directory in disk, by using the Alignment and Document Layers.

The following Chapter tackles the search functionalities of the Bilingual Framework, with detail on how the Layers work together and contribute to achieve the final goals. These functionalities are centered on counting and locating occurrences in both sides of the corpora, but also extract context surrounding an occurrence of a term.

3.6 Results

In this Section, the results are focused on the structure of the Bilingual Framework, in particular for the space it requires in main memory, and for the time it takes to build all the structures that support the Framework. The results do not consider the Document Layer, as most of the tests were made with just one corpora, sentence aligned, and with sentences phrase aligned.

In this thesis, the experiments done were based on two language pairs, English (EN) - Portuguese (PT) and German (DE) - Portuguese (PT). In the first language pair, we considered five aligned parallel corpora: 1) EUconst [171] with size of 1.6 Megabytes (Mb); 2) EMEA [171] with 196 Mb; 3) Europarl [105] with 394 Mb; 4) DGT[5] with 804 Mb; and 5) Eurlex[6] with 1126 Mb. In DE-PT, we considered three aligned parallel corpora: 1) EMEA with size 208 Mb; 2) Europarl with size of 428 Mb; 3) DGT with size of 788 Mb.

These sizes consider both corpora in the two languages, together with the monotonic alignment. The texts were aligned using the tool developed by Gomes et. al. [72], using knowledge accumulated from previous iterations of the alignment flow, and no stop words were removed. These parallel texts were then indexed by the Bilingual Framework and work as basis not only for the following results, but also for other experiments further presented in this thesis.

Tables 3.1 and 3.2 summarizes some information about the bilingual corpora used in the tests for the Framework. Besides the total sizes, and sizes of the corpora per language, in Megabytes, it also displays the number of tokens (words and symbols) per corpora, per language, and the number of coarse and fine alignment segments. Both this information is displayed in millions (M). In the Tables there is also information about the combination of all the corpora together, that is also used for testing.

The sizes of the individual corpora in the three languages do not add up to meet the total size presented in the Tables. However, one cannot forget that this corpora is alignment annotated, meaning that besides the texts, there is a representation of the alignment as well requiring space. In Table 3.2 the number of aligned segments is also for the coarse and fine alignment. As the lexicon for DE-PT language pair is smaller, the number of alignments is also smaller.

[5] http://ipsc.jrc.ec.europa.eu/index.php?id=197
[6] http://eur-lex.europa.eu/en/index.htm

Table 3.1: Information about the English and Portuguese bilingual corpora

Corpora	Total Size (Mb)	Size (Mb)		Tokens (M)		Aligned Segments (M)	
		EN	PT	EN	PT	Coarse	Fine
EUconst	1.6	0.687	0.727	0.124	0.133	0.007	0.084
EMEA	196	75.011	84.240	14.241	15.769	0.917	13.112
Europarl	394	160.223	178.484	29.871	32.640	1.010	23.205
DGT	806	316.623	356.866	59.413	67.975	1.635	51.206
Eurlex	1126	436.756	491.548	79.848	91.797	2.767	69.423
All	2523.6	989.300	1111.864	183.497	208.314	7.336	157.030

Table 3.2: Information about the German and Portuguese bilingual corpora

Corpora	Total Size (Mb)	DE Size (Mb)	DE Tokens (M)	Aligned Segments (M)	
				Coarse	Fine
EMEA	208	91.304	13.915	1.102	10.919
Europarl	428	196.587	30.603	1.108	22.434
DGT	788	355.420	52.934	2.797	42.118

3.6.1 Memory Consumption

The first experiments performed with the Bilingual Framework, focus on its space requirements, with an analysis of the memory consumption for all the combinations of the Text Layer and the Alignment Layer implementations. Tables 3.3 and 3.4 show an overview of the memory consumption of the Bilingual Framework in Megabytes (Mb), for EN-PT and DE-PT. The experiments were made with all the possible combinations between the Text and Alignment implementations, with Word-based Suffix Arrays (WSA) and Byte-codes Wavelet Trees (WT) for the Text Layer, and Bitmaps (Bmap) and Arrays of Integers (ArrInt) for the Alignment Layer.

The results also include the Document Layer and the Case Insensitive Layer, in the total amount displayed. The results were measured as well for all the corpora at the same time, with and without EMEA, in EN-PT, which is the corpora that covers a topic different from the other four. Both Tables also repeat the size of the corpora, for easier interpretation of the results.

The results in Table 3.3 demonstrate that the less space consuming approach is the combination between Byte-codes Wavelet Trees and Bitmaps. For any of the parallel corpora, including the smaller one (Euconst), this approach takes less space than the corpora + alignment size. These results are expected because it is the result from the study of compressed alternatives for both Layers. On the opposite side, all the approaches based on Word-based Suffix Arrays require more space than the parallel corpora size, which is also expected, due to the nature of the structure. Still, the light implementation of the index, and its word-based implementation, make it considerably smaller than the 4x the text size, by its character-based version.

Table 3.4 shows the results for the same set of tests for the DE-PT language pair, following the

Table 3.3: Memory consumption results in the EN-PT language pair

Corpora	Size (Mb)	WSA (Mb)		WT (Mb)	
		Bmap	ArrInt	Bmap	ArrInt
EUconst	1.6	2.576	2.903	0.940	1.268
EMEA	196	263.744	315.123	99.969	151.348
Europarl	394	546.542	628.601	202.747	284.806
DGT	806	1117.707	1305.400	418.667	606.361
Eurlex	1126	1503.145	1748.265	568.369	813.489
All	2523.6	3421.573	3988.152	1306.178	1872.758

Table 3.4: Memory consumption results in the DE-PT language pair

Corpora	Size (Mb)	WSA (Mb)		WT (Mb)	
		Bmap	ArrInt	Bmap	ArrInt
EMEA	217	278.690	324.349	108.731	154.390
Europarl	448	588.197	668.006	222.551	302.360
DGT	825	1062.608	1221.960	410.159	569.511

same behavior of the results for the EN-PT pair. This shows how the Bilingual Framework has the same space requirements, regardless of the language pair used. Other languages could be tested here as well, but these two language pairs were chosen for being two of language pairs with more work done on the research group. Even between these two, there is a big difference, with the EN-PT being much more advanced. However, the DE-PT also brings another challenges, such as the compound (and long) words.

To better understand the memory results, Tables 3.5 and 3.6 show the memory consumption ratios of every implementation of the Framework. These ratios were determined using a division between the index space consumption and the parallel corpora size.

Table 3.5: Memory consumption ratios for the EN-PT language pair

Corpora	Size (Mb)	WSA		WT	
		Bmap	ArrInt	Bmap	ArrInt
EUconst	1.6	1.61x	1.81x	0.59x	0.79x
EMEA	196	1.35x	1.61x	0.51x	0.77x
Europarl	394	1.39x	1.60x	0.51x	0.72x
DGT	806	1.39x	1.62x	0.52x	0.75x
Eurlex	1126	1.33x	1.55x	0.50x	0.72x
All	2523.6	1.36x	1.58x	0.52x	0.74x

The ratios reinforce that the compact approaches for the Text and Alignment Layers reduce considerably the space requirements of the Framework. The implementation based on WTs and

Table 3.6: Memory consumption ratios for the DE-PT language pair

Corpora	Size (Mb)	WSA		WT	
		Bmap	ArrInt	Bmap	ArrInt
EMEA	217	1.28x	1.49x	0.50x	0.71x
Europarl	448	1.31x	1.49x	0.50x	0.67x
DGT	825	1.29x	1.48x	0.50x	0.69x

Bmaps have a memory consumption ratio around 50% of the aligned parallel corpora size in EN-PT, for the bigger corpora, Eurlex, and slightly above that value for the other corpora. For the language pair DE-PT the results are similar, with space requirements of 50%, for DGT, the largest corpora tested for this pair[7]. These space consumption ratios are result of the compressed approach of the data structures that support the Layers of the Framework.

On the other hand, the combination between WSA and ArrInt, require around 1.55x - 1.81x of the text size in EN-PT, and around 1.5x in the three corpora tested for DE-PT. These results make this approach the most space consuming version of the Bilingual Framework tested in the scope of this research. In fact, the results that demonstrate that every approach based on Arrays of Integers, require more space than the actual parallel corpora size, even combining it with the Bitmap approach for the Alignment Layer.

These space requirements are not surprising due to the nature of the data structures. Byte-codes Wavelet Trees index the compressed text, thus they require less space than the original text size. Besides, this structure offers indexing and searching capabilities over the compressed text, without needing the text alongside the index. Additionally, the nature of this representation, enables encoding algorithms that lead to smaller space consumption, as the self-synchronization property is not lost, like the Plain Huffman. However, considering the space requirements of the original Suffix Array, based on characters, the Word-based Suffix Array alternative shows efficient space consumption, lower than twice the text size. That is due to its lightweight structure, based on words, and with a vocabulary that does not demand a lot of space in memory. However, it is still considerably higher than the compressed alternative for the Text Layer.

Additionally, by analyzing the parallel corpora, it is clear that the tendency is for the consumption ratio to decrease, which can be seen when comparing the results obtained from EUconst to Eurlex. This variation of the ratios happens because larger texts have more repetitions, which benefits the compression algorithms and the vocabulary representation, for the non-compressed data structures. The decreasing variation only has two exceptions. The first one is Europarl, as it is a parallel corpora extremely rich in repetitions, despite being smaller than DGT and Eurlex. The second exception is when All the corpora are considered, since it is a mix of all the texts, each one with different subjects and with different and unrelated terms, thus being influenced by the compressed ratios of all of them. In particular, EUconst, Eurlex and

[7] In the research context where this work is inserted on, the different language pairs evolve at different paces, thus it is not expected that every corpora is being worked on for every pair

DGT are related to legislation, while EMEA has more medical terms and Europarl has transcripts of speeches in the European Parliament.

For the Alignment Layer, the same behavior is seen for the approach based on the Bitmaps, when compared with the solution based on Arrays of Integers. Analyzing the compression ratios, we can see that they mostly decrease around 17 to 26%, in both language pairs, just by changing the Alignment Layer implementation from the Arrays to the Bitmaps. That smaller space consumption is caused by the more compact representation of the segments, based on three bitmaps, against the five arrays of integers of the alternative, where every integer that represents and alignment segment requires from 8 to 32 bits.

Table 3.7 and 3.8 show the space consumption of the Text Layer in detail, including the Case Insensitive Layer (CIL), in Megabytes. The Table shows as well the size of the texts in the corpora and the alignment, separately, for better understanding the space requirements of the Bilingual Framework. The Document Layer is not depicted in the Table, as it is only relevant for the case of **All** the parallel corpora being indexed together. In that case, considering that every parallel corpora is in its own directory, within a root directory, the Document Layer proves to require a minimum amount of space, since for the five documents and six directories, takes 333 bytes.

Table 3.7: Memory consumption results of the Text Layer, in Megabytes, for EN-PT

Corpora	**Texts Size**	**Memory Consumption**			**Consumption ratios**		
		WSA	**WT**	**CIL**	**WSA**	**WT**	**CIL**
EUconst	1.414	2.412	0.776	0.065	1.70x	0.55x	0.05x
EMEA	159.251	242.526	78.751	7.502	1.52x	0.49x	0.05x
Europarl	338.707	505.131	161.336	15.627	1.49x	0.48x	0.05x
DGT	673.489	1030.708	331.669	31.847	1.53x	0.49x	0.05x
Eurlex	928.304	1385.782	451.006	42.912	1.49x	0.49x	0.05x
All	2101.164	3154.418	1039.024	97.953	1.5x	0.49x	0.05x

Table 3.8: Memory consumption results of the Text Layer, in Megabytes, for DE-PT

Corpora	**Texts Size**	**Memory Consumption**			**Consumption ratios**		
		WSA	**WT**	**CIL**	**WSA**	**WT**	**CIL**
EMEA	188.912	266.091	96.132	7.876	1.41x	0.51x	0.04x
Europarl	394.183	562.167	196.522	16.679	1.43x	0.50x	0.04x
DGT	721.572	1014.660	362.210	29.796	1.41x	0.50x	0.04x

Analyzing the results from the Text Layer, it is even more visible the difference in space requirements of Word-based Suffix Arrays and Byte-codes Wavelet Trees. WTs require considerably less space than Word-based Suffix Arrays, requiring around 49% of the texts size, for Eurlex, in EN-PT, and 50% in DE-PT. With WSA's that space consumption ratio is around 1.49x the total size of both texts, also for Eurlex in EN-PT, and 1.41x in DE-PT. For the remaining parallel

corpora, the results follow similar results, with a slight higher ratio for smaller texts, except for Europarl, as explained earlier.

These results also demonstrate that the Case Insensitive Layer has a slight impact on the total memory consumption, requiring around 5% of the total size of the indexed texts, in EN-PT. In the other language pair, DE-PT, similar results can be seen with the CIL, for DGT, taking around 4% of the total text index size. With these minor ratios, we decided that this implementation of the CIL meets the space efficient requirements, for our purpose of normalizing the casing of the Framework, considering the advantages that it brings. For that matter, no studies were pursued to compress this Case Insensitive Layer.

Tables 3.9 and 3.10 show the memory consumption for the Alignment Layer, in Megabytes, alongside the memory consumption ratios obtained with Arrays of Integers and Bitmaps, considering the size of the alignment.

Table 3.9: Memory consumption results of the Alignment Layer, in Megabytes, for EN-PT

Corpora	**Texts Size**	**Memory Consumption**		**Consumption ratios**	
		Bmap	**ArrInt**	**Bmap**	**ArrInt**
EUconst	0.186	0.089	0.416	0.48x	2.24x
EMEA	36.749	12.077	63.456	0.33x	1.73x
Europarl	55.293	22.883	104.942	0.41x	1.90x
DGT	132.511	48.751	236.444	0.37x	1.78x
Eurlex	197.696	65.774	310.894	0.33x	1.57x
All	422.436	149.572	716.151	0.35x	1.70x

Table 3.10: Memory consumption results of the Alignment Layer, in Megabytes, for DE-PT

Corpora	**Texts Size**	**Memory Consumption**		**Consumption ratios**	
		Bmap	**ArrInt**	**Bmap**	**ArrInt**
EMEA	28.088	11.234	56.893	0.40x	2.03x
Europarl	53.817	23.225	103.034	0.43x	1.91x
DGT	103.428	42.683	202.035	0.41x	1.95x

Regarding the Alignment Layer, the difference between implementations is easily visible now in more detail, with the approach based on Bitmaps requiring much less space than the alternative based on Arrays of Integers. Recovering the total size of the parallel corpora in Table 3.9, for Eurlex, in EN-PT, the Alignment Layer supported by Bitmaps require around 33% its size. On the other hand, the alternative based on Arrays of Integers require around 1.57x the size of the alignment, in main memory. For the other parallel corpora, except for EUconst, the ratios do not vary that much, with results around 33% to 48% of the alignment size.

For the DE-PT language pair, the numbers do not differ that much. For DGT, the Bitmaps take 41% of the size of the alignment, while Arrays of Integers take around 1.95x. The difference

in space requirements between the two language pairs, result from the alignments being at different stages of evolution, with the DE-PT being less complete as the EN-PT, thus making it smaller. However, since all the segments of the alignment are represented anyway, that space needs to be considered for the Alignment Layer, thus the higher ratios.

The Alignment Layer does not make the Bilingual Framework space inefficient. Despite none of the approaches tested is actually supported by a compressed index, the representation based on bitmaps, achieve efficient space requirements, even considering that every segment of the alignment requires several bits to represent it. With such results, we decided not to try to compress the Alignment Layer even further. As will be seen in the next Chapter, when the data is compressed, the access time slows down, thus to avoid making the response time of the Framework slower, we continued with the Bitmap approach for the most space efficient representation of the Alignment Layer.

Finally, by looking at all the results, it is possible to determine that the space consumption ratios for EUconst are considerably higher. That happens because the size of the corpora is very small, making it less compressible, as there is not so much repetition and regularity. Therefore, I included this corpora in the tests, in EN-PT, so that this difference was noticed, concluding that for smaller corpora, the compression indices are not necessary. For the other corpora, regardless the language pair, the gains in space are undoubtedly visible.

3.6.2 Time for Creating the Indexes

In the previous Section, I detailed the results for the memory consumption of the Framework, including the space requirements per Layer. However, it is not just the space that is relevant when a data structure is chosen to support a Framework such as the one presented in this thesis. The access time and construction time of the structures are also important. In this Section, I then present the time consumption for building the Framework. The experiments were developed using a machine with 16 Gigabytes Memory RAM, a Intel Core I7, 3.4 GHz processor, using Ubuntu Linux Kernel 3.2.0.

Tables 3.11 and 3.12 show the indexing time results, for the approaches based on Word-based Suffix Arrays and Arrays of Integers, for the Text and Alignment Layer respectively. The results are presented in milliseconds (ms) and are measured from the beginning to the end of the procedure to create the index. Besides the total amount of time spent on building the Framework, the Table also shows more specific results for the Text Index, the Case Insensitive Layer (CIL) and the Alignment, also in ms. The column that shows the results for the Text Index, consider just the creation of the data structures in both languages, without including the CIL. The time consumption for creating the CIL is presented in the next column of the Tables, with the sum between the time spent building the Layer for both languages. Finally, the column with the results from the Alignment consider parsing it from the aligned parallel corpora, plus the creation of the data structures that support the Alignment Layer. The Tables present an average of the indexing time, for all the columns displayed.

The results in Tables 3.11 and 3.12 show that the combination between WSA and ArrInt

Table 3.11: Indexing time results, for Word-based Suffix Arrays and Arrays of Integers, in EN-PT

Corpora	Text Index (ms)	CIL (ms)	Alignment (ms)	Total (ms)
EUconst	69.306	106.090	43.209	219.967
EMEA	9682.450	12219.007	4978.347	27410.137
Europarl	20432.115	26454.773	10090.002	58248.771
DGT	48006.035	55719.712	21498.545	127320.282
Eurlex	65018.269	74894.055	29069.396	171887.022
All	167071.792	160375.104	70657.646	401597.897

Table 3.12: Indexing time results, for Word-based Suffix Arrays and Arrays of Integers, in DE-PT

Corpora	Text Index (ms)	CIL (ms)	Alignment (ms)	Total (ms)
EMEA	10306.182	15492.546	5703.496	31962.847
Europarl	20294.396	31632.757	12856.319	64570.989
DGT	49942.184	62189.643	21237.753	134304.154

has an efficient indexing time performance. For EN-PT, building the Framework takes around 401.5 seconds (401597 ms), for the combination of all the corpora. It is relevant to note again that the text and alignment indexes of the Framework are immediately saved in disk, which also influences the indexing time, since writing in disk is an expensive operation. However, for completeness and fairness these results include that as well, as it is in fact the real time spent on building the Framework. For DE-PT, the results follow the same behavior as for EN-PT.

From the more specific results, it is possible to infer that the most time consuming Layer to be indexed is, by far, the Text Layer, as expected, since indexing the texts is a more expensive operation than indexing just the alignment segments. For Word-based Suffix Arrays, it is necessary to create a vocabulary, convert the text considering the vocabulary, and then index it, which takes a considerable amount of time. However, drilling down on the Text Layer, we can see that CIL actually takes even more time than the actual index to be built. This demonstrates that the process for transforming the text to lower case is actually expensive, since the entire text needs to be traversed and then transformed. The only exception is the results for the combination of **All** the corpora, in EN-PT, where actually the time for building the text indexes is higher than the time for building the CIL. Our interpretation is that this has a tendency to happen for a group of several corpora, where their total size is already significant and thus more time consuming to create the data structures that index it.

Tables 3.13 and 3.14 show the same indexing time results, but for Byte-codes Wavelet Trees and Bitmaps, also in milliseconds. This Table does not repeat the results for CIL, since the implementation does not change. Just like in the previous Tables, these results present an average, including the total indexing time, thus it is normal that the numbers do not match with precision, namely the exact time difference from one approach to another.

Table 3.13: Indexing time results, for Byte-codes Wavelet Trees and Bitmaps, in EN-PT

Corpora	**Text Index (ms)**	**Alignment (ms)**	**Total (ms)**
EUconst	29.809	43.572	181.269
EMEA	3008.814	4600.213	20128.638
Europarl	6635.239	9779.283	43748.495
DGT	12985.722	19730.674	88266.405
Eurlex	17757.077	26813.552	120337.833
All	42908.234	66436.808	271217.792

Table 3.14: Indexing time results, for Byte-codes Wavelet Trees and Bitmaps, in DE-PT

Corpora	**Text Index (ms)**	**Alignment (ms)**	**Total (ms)**
EMEA	3380.722	5236.139	24346.478
Europarl	7451.068	11740.627	51654.449
DGT	13571.382	19497.815	97106.997

The results in Tables 3.13 and 3.14 show an opposite behavior, when compared with the results obtained with the combination between WSAs and ArrInts. Here, creating the Alignment Layer actually takes more time than creating the Text Layer. That difference lies on the very efficient building algorithm for the Byte-codes Wavelet Tree, which despite compressing the text, is highly optimized and creates the tree topology, without actually building a data structure per se, like in Word-based Suffix Arrays. For that matter, the indexing time with the Text Layer based on Byte-codes Wavelet Trees is lower than the one obtained by the uncompressed alternative, with Word-based Suffix Arrays.

Considering the Alignment Layer alone, the difference is actually quite slim, when comparing with the alternative based on Arrays of Integers. The indexing times of both approaches are similar, despite for larger corpora, that difference keeps slightly increasing.

These results make this combination, with the most compressed indexes, faster on creating the structure for the Framework, with a visible performance improvement for every corpora, in both language pairs. Again, the smaller corpora present smaller improvements, as their size is not enough to be able to observe an improvement as high as the larger corpora, such as in Eurlex or DGT.

In general, both implementations chosen for the Layers of the Bilingual Framework present an efficient indexing time and space consumption, in particular the one based on Byte-codes Wavelet Trees + Bitmaps, which require around 50% of the whole parallel corpora + alignment size, and are actually faster on building the Framework.

In the next Chapter, I introduce the phrase-based search algorithms for the Bilingual Framework, and the time response that all these alternatives present.

CHAPTER 4

PHRASE-BASED SEARCH

Chapter 3 described the structure of the proposed Bilingual Framework, with an in-depth explanation of the Text and Alignment Layers, and the data structures that support them. Knowing the structure that supports the Framework, this Chapter details its phrase-based functionalities: *counting* and *locating* all the occurrences of one, or two word and multi-word terms in the indexed parallel corpora.

These functionalities are based on phrase-based bilingual searches that locate co-occurrences of two terms in different languages, over parallel corpora. A phrase-based search finds the occurrences of terms in a corpora, with the terms being either single words or multi-word phrases, with phrases being substrings of potentially unlimited size, in number of words. As explained earlier in this Thesis, a phrase in this context does not mean a phrase in syntactic theory, but more of a word or a set of contiguous words.

The phrase-based translation models [106, 146, 174] advanced the state of the art in Machine Translation, by using phrases as the basic unit of translation, instead of words. As introduced in Chapter 2, the usage of phrases allowed models to learn local word re-orderings in sentences of two different languages, alongside translation of multi-word terms. This led to models more sensitive to the local context and to the differences in languages, and thus more powerful for translation, leading to more robust translations, specially for substrings common enough to be observed in the training of the translation engine. These models are extensively used in Statistical Machine Translation, thus it is extremely important to be able to create them from parallel corpora, or even draw statistics from them.

The phrase-based bilingual search can then be helpful to determine these statistics [106, 189], such as the co-occurrence frequency of a pair of terms, in two different languages, over the indexed parallel corpora. Considering the Machine Translation task itself, as the higher the co-occurrence frequency is, the higher the probability for two phrase terms to be correct translations of each other is. Additionally, these statistics, the location of the co-occurrences

in the corpora and the surrounding context where they co-occur, are considerably relevant for supporting the extraction of new and unknown translation pairs in a given language pair, for context analysis and word-sense disambiguation tasks, among others.

As it is known, two languages can be completely different in terms of grammar, order and may even use different alphabets (e.g. Hindi, Mandarin, Russian, Arabic, etc). This heterogeneity makes the task of supporting these functionalities a not easy task, in particular if we are concerned about their time and space performance. Thus, to support these functionalities, the Bilingual Framework uses a strong coordination between the Text Layer and the Alignment Layer, with the objective of offering the best efficiency possible.

Having parallel corpora in any language pair indexed in the presented Bilingual Framework, the following Sections present our proposed implementations for phrase-based bilingual searches, using the data structures that support the Framework. The next Section in specific describes a first and standard approach for the bilingual search, detailing the monolingual searches in each language on both Text Layer approaches, and the coordination with the Alignment Layer to determine the co-occurrences.

4.1 Bilingual Search

A phrase-based bilingual search determines the co-occurrence frequency, or location, of a pair of word or multi-word terms, X <->Y, in a sentence aligned parallel corpora, with X being a term in one language and Y its translation in the other language.

Considering the language heterogeneity mentioned above, determining if two occurrences in different languages co-occur in the scope of the parallel corpora is not an easy task. A co-occurrence is only statistically valid, meaning that it is only accounted for as a valid co-occurrence of two terms, if their individual occurrences in the respective texts appear close enough to each other. This means that, at least, they should occur in two parallel sentences, translation of each other. Consider Figure 4.1 with three sentences extracted from the English - Portuguese corpora EUconst [171], and the words `Commission` and `Commissão` highlighted.

In the second sentence, there is one occurrence of each word of the pair, `Commission` and `Commissão`, that may be considered nearby in the texts, since each of them occur in the second sentence, which are parallel. On the other hand, the occurrence of `Commissão` in the third sentence is already too far apart to be considered a valid co-occurrence, and there is no match for `Commission` in the second paragraph of the English text, as it is not parallel to the third sentence in Portuguese. As we assume that text is aligned at the sentence level, determining the co-occurrences of two terms must take into account this granularity of alignment.

The best way to determine if the occurrences of two terms actually co-occur, in the parallel corpora, is to use the parallel text alignment. As a result, an occurrence of X co-occurs with an occurrence of Y when both appear in aligned segments of the sentence aligned parallel corpora. Considering $coarse(X)$ and $fine(X)$ the coarse segment where X appears, two occurrences of X and Y at offsets off_X and off_Y respectively, must meet the following conditions to co-occur:

English	Portuguese
The European Council and the Council shall be heard by the European Parliament in accordance with the conditions laid down in the Rules of Procedure of the European Council and those of the Council.	O Conselho Europeu e o Conselho são ouvidos por o Parlamento Europeu em as condições previstas em os regulamentos internos de o Conselho Europeu e de o Conselho.
The **Commission** may attend all the meetings of the European Parliament and shall, at its request, be heard.	A **Comissão** pode assistir a todas as sessões de o Parlamento Europeu e é ouvida quando assim o solicitar.
It shall reply orally or in writing to questions put to it by the European Parliament or by its members.	A **Comissão** responde, oralmente ou por escrito, a as questões que lhe sejam colocadas por o Parlamento Europeu ou por os seus membros.

Figure 4.1: Three aligned coarse segments in English and Portuguese, with the words `Commission` and `Commissão` highlighted

Condition 1: $coarse(off\_X) = coarse(off\_Y)$

Condition 2: $fine(off\_X) - d \leq fine(off\_Y) \leq fine(off\_X + |X|) + d$

where d is the maximum distance in fine segments allowed for term X and Y, to co-occur at positions off_X and off_Y of the corpora.

Condition 1 states that an occurrence of *X*, at position *off_X*, must appear in an aligned coarse segment with an occurrence of *Y*, at position *off_Y*. This guarantees that no co-occurrence of the pair *X* and *Y* happens outside the border of a sentence. This Condition itself would be sufficient to discard that the second occurrence of `Commissão` does not co-occur with the occurrence of `Commission` in the example of Figure 4.1.

However, a sentence can be long. Therefore, two terms that appear in aligned sentences can also be considered too far away from each other to be considered a valid co-occurrence. This leads to *Condition 2*, which verifies, within a sentence, if the “distance"between two terms is small enough to consider the occurrences a valid co-occurrence. For that purpose, the Condition uses the fine granularity of the alignment. *Condition 2* states that *off_X* and *off_Y* are a valid co-occurrence, if the fine segments where they appear do not appear further than a variable, but pre-established, distance d. This distance can be set by the user of the Framework, to measure how close it requires the co-occurrences to be in the parallel corpora.

Returning to Figure 4.1, considering the occurrences of the two terms at the second aligned

sentences, `Commission` appears at position 36 of the English text and `Commissão` at position 30 of the Portuguese text. These numbers consider the absolute offsets of the corpora, meaning that the counting is made from the very beginning of the texts. These two numbers are quite close enough, in particular if we consider that naturally there will be more words in one text than in the other, due to the differences in the languages. However, these offsets were determined by just only considering a snippet of a corpora considerably bigger. With a bigger scope, like the whole indexed corpora, the discrepancy in offsets will be much likely higher, in particular further down in the texts. This means that using the absolute offsets would be misleading to determine co-occurrences, since two terms may appear too distant, when they appear in aligned segments of the alignment.

With the fine segments of the alignment, this problem can be solved, since despite the small length of the fine segments, which can have an one word length, a large amount of these segments are aligned. Thus, they make a good metric to measure the distance between terms in two different languages.

Figure 4.2 shows an abstract example of some occurrences, and their lengths, throughout a coarse segment, presented in the Figure finely aligned. Every dotted line represents a fine segment of the alignment, with the continuous and highlighted lines representing five occurrences in the corpora, tagged from A to E. The occurrences A, B and C are on the English side, while D and E are occurrences of the Portuguese term in the search.

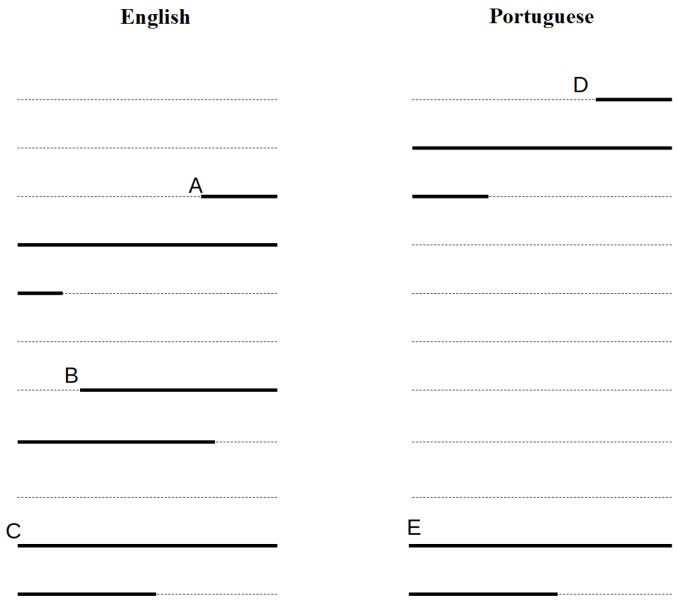

Figure 4.2: Abstract representation of occurrences of terms in two languages, from A to E

In Figure 4.2, occurrence A appears in the third aligned fine segment of the English sentence, while occurrence D appears in the first one. For $d = 2$, these occurrences would co-occur, since $fine(D) - fine(A) = 2$, thus meeting *Condition 2*. The same scenario happens for the occurrences C and E, with $fine(C) - fine(E) = 0$, while occurrence B does not co-occur with D or E as it is too far apart from them, $fine(B) - fine(E) = 4$ and $fine(D) - fine(B) = 7$.

Going back to *Condition 2*, one can notice that it also includes the end of an occurrence, $off\_X + |X|$, using the length of the term. Fine segments can be very small, possibly even smaller than the term being searched[1]. This leads to having occurrences that can span over more than one fine segment, as Figure 4.2 shows. Depending on the value set for d, if we considered just the starting offset of the occurrence, the bilingual search procedure would possibly disregard some valid possibilities for co-occurrence. To include this factor in the bilingual search, the distance d takes into account the length of the occurrences, or their endings.

Recovering the same example in Figure 4.2, for $d = 1$, occurrences A and D would not co-occur, if we just considered the starting offset of the occurrences, because the starting offsets appear two fine segments apart. However, the occurrences even span to the third aligned fine segments, in both languages. Therefore, it would be interesting to consider these two occurrences as a valid co-occurrence. Considering *Condition 2*, and the usage of the offset where the occurrence ends, then we have a match, since $fine(A) - fine(D) + |D| = 0$, thus meeting *Condition 2* for $d = 1$.

The next Section introduces a first approach for supporting the bilingual search. Either one of the Conditions introduced above are key to the whole process, regarding the implementations used for the Layers of the Framework.

4.2 Bilingual Search Procedure: A First Approach

To support a bilingual search for a pair $X <-> Y$, any procedure needs to consider the Text Layer, for obtaining the individual occurrences of the terms, and the Alignment Layer for verifying if they meet *Condition 1* and *Condition 2* criteria. To get the individual occurrences of X and Y, the search algorithms of the text indexes can be used. As this step considers each language apart, it is denominated monolingual search.

A phrase-based monolingual search consists of counting or locating a word or multi-word phrase term X, in a given language, in the corpora of the same language. This is the most straightforward search operation supported by the Bilingual Framework, as it is almost fully based on the *locate* and *count* functionalities of the data structures that support the Text Layer. Considering the operation to **locate** all the occurrences of term X, the search procedure selects the text index that represents the corpora in the respective language, returning the set of all the occurrences of the term in that corpora.

4.2.1 Getting the occurrences in a Word-based Suffix Array

A Word-based Suffix Array indexes all the positions of the text, at word boundary, following a lexicographical order. For that matter, all the occurrences of a term will be represented in a contiguous interval of the Suffix Array. Considering this characteristic of Suffix Arrays, the monolingual search uses a bisect algorithm, or binary search, to determine the begin and end of the interval, where all the occurrences of the search term are indexed.

[1] A fine segment in one of the languages can consist of a single word, or it can even be empty (see Figure 2.9)

However, to have the Suffix Arrays indexing the corpus at word boundaries, every word in the texts is represented by an unique integer identifier. This mapping from text to integer is defined in a *vocabulary*, supported by a hash table, which maps the word to the respective integer. So, to enable the matching in the Suffix Array, the first step is based on converting every word, or token, of the searched term to its respective integer *id*, using the *vocabulary*, to have the same representation of the data structure. This results in a set of *ids* that will be used for determining the location of the occurrences of the searched term.

Then, the next step of the search focuses on determining the occurrences of the term in the respective corpora. For that, two binary searches are performed. One is performed over the entire Suffix Array, from the first to the last position, to include every word of the indexed corpus in the search. This results in the beginning position $begin$ of the interval. Then, a second binary search is performed, from the position $begin$ to the end of the Suffix Array, to determine the end of the interval of occurrences. After the interval is calculated, then the final list of occurrences is filled with all the positions in the Suffix Array, within the calculated interval.

Consider again the Word-based Suffix Array example in Figure 4.3. Searching for `to be` would result in the interval [8,9], the two last positions of the Suffix Array. This shows that there are two occurrences of the term in `to be or not to be that is the question`, at positions 0 and 4 of the text, the values stored in positions 8 and 9 of the array.

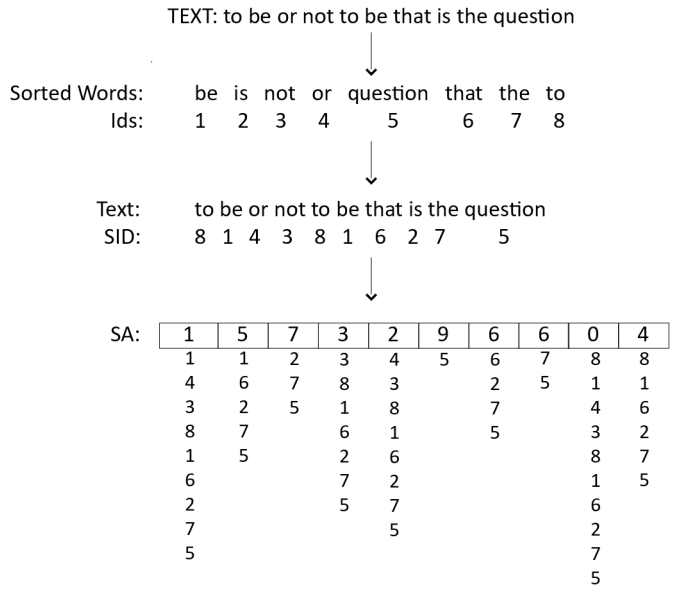

Figure 4.3: Example of a Word-based Suffix Array

Algorithm 4.1 shows the *locate* procedure for Word-based Suffix Arrays. This procedure uses the index and the term being searched, to return the number and list of occurrences. The algorithm starts by transforming the searched term to its respective *ids*, followed by the calculation of the interval of occurrences using *bisect*, and ending with the population of the list of occurrences with the occurrences in the interval. The *bisect* algorithm performs a binary search over the Suffix Array, and for every position visited, it uses the integer representation of the indexed

text (*text* field in the **t_index** structure), to determine if there is a match with the searched term. This is the base implementation of the Word-based Suffix Arrays *locate* function and it has a time complexity of $O(m * log\, k)$ in the worst case, with m being the word size of the term and k the number of words in the text.

```
int* locate(struct t_index *index, char* term, int *num_occs){
  int ntoks = 0;
  int *ids, occs;
  for(every token in term){
    ids[ntoks] = get_id(index->voc, token);
    ntoks++;
  }
  int beg = bisect(ids,ntoks,index->suffix_array,index->text,0,index->len);
  int end = bisect(ids,ntoks,index->suffix_array,index->text,beg,index->len);
  *num_occs = end - beg;
  if (*num_occs = 0){
    return occs;
  }
  occs = calloc(*num_occs, sizeof(int));
  for(i = beg; i < end; i++){
    occs[i] = index->sa[i];
  }
  return occs;
}
```

Listing 4.1: *locate* search procedure for Word-based Suffix Arrays

4.2.2 Getting the occurrences in a Byte-codes Wavelet Tree

A Byte-codes Wavelet Tree represents each word in the indexed text with a codeword of bytes, following the order they appear in the text. The codewords then follow a tree structure, where the root has the first byte of all the codewords, while the following nodes on the lower levels will have the remaining bytes of the codewords represented.

Similarly to what happens with Word-based Suffix Arrays, the first step of the monolingual search is to determine the respective codeword for every word in the searched term. That is done as well using a vocabulary that is built based on the Plain Huffman encoding algorithm, storing the words by the ascending order of their frequency in the text. Right after getting the codewords, the search procedure also determines the nodes where the bytes of the codewords are represented. In fact, as the tree topology is saved in memory, with the information about where all the bytemaps that correspond to the nodes start, it is possible to get the bytemaps needed on the search from the start. The implementation of this vocabulary and of these procedures fall outside of the scope of this thesis.

Figure 4.4 recovers the example of a Byte-codes Wavelet Tree of the previous Chapter. In the example, the word `question` has three codewords, $b_4 b_5 b_2$, thus the three nodes where the bytes are represented would be calculated (node A, C and D). Considering the searched term `the question`, this would be done for both words in the term, resulting in the bytes of the codewords of the two words and the respective nodes where the bytes are represented.

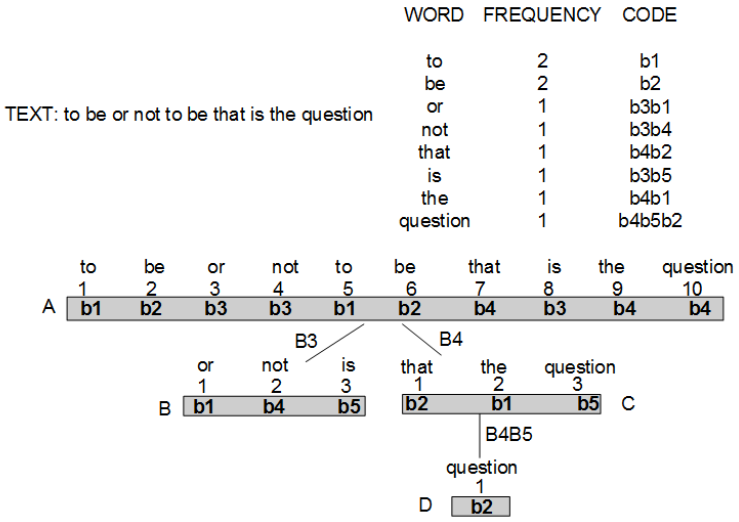

Figure 4.4: Example of a Byte-codes Wavelet Tree

The search in a Byte-codes Wavelet Tree is centered on a word. While in Suffix Arrays, one could search for the interval of occurrences considering a whole term t, multi-word or not, with this structure the search must be done based on one of the words of t. For that matter, the word chosen to be the "anchor"for the search would be the least frequent word, lf, of t. In Figure 4.4, considering t to be `be or`, the search would be based on the word `or`, as it is the least frequent word of t.

To determine the location of an occurrence of a word, the procedure is started with the last byte of its codeword, thus on the lowest level of its codeword representation. For the term `question`, with codeword $b_4b_5b_2$, it would be node D. To determine the location of the first, and here the only, occurrence of `question` in the text, the procedure looks to the upper level, node C, and does a $select_{b_5}(C,1) = 3$. This means that the first (and only) occurrence of `question` is represented in the third position of node C, the one with the 2nd byte of the codeword. This procedure continues until reaching the root, using always the result of the $select$ from the previous node. Continuing the example for the term `question`, for the next level the procedure does $select_{b_4}(A,3) = 10$. As node A is the root, the term `question` appears at position 10 of the indexed text.

This is the standard procedure to determine the location of a word in the text, in a Byte-codes Wavelet Tree, regardless of the height of the tree. If the word just has a codeword with one byte, the same logic is applied directly to the root.

For a multi-word term t, the processed described above is performed based on the least frequent word, lf, of the term. Starting from the last byte of the codeword of lf, the algorithm $ascends$ the tree structure using $select$ operations, until determining the position of an occurrence in the root. For instance, for the term `be or`, in node B the procedure does a $select_{b_1}(B,1) = 1$, to determine where in the node is located the first occurrence of the 2nd byte of the codeword, b_1. Expanding this up to node A, the procedure continues with $select_{b_3}(A,1) = 3$, meaning that the first (and only) occurrence of `or` appears in the 3rd position

of the text. Now, since the searched term is `be or`, it is necessary to determine if immediately before the occurrence of `or`, there is an occurrence of `be`. Since `or` occurs at position three, then `be` would need to appear at position two. As $A[2] = b_2$, which is the first and only byte of the codeword for `be`, there is a positive match and `be or` occurs at position two of the text.

In a scenario where the other word does not have just one byte, like `be`, the search procedure would *descend* the tree structure, using the bytes of the remaining codewords of the searched term. Consider that the searched term now is `that is` and that the procedure already determined that `is` at position eight of the text. To determine if `that` precedes it, the first byte is verified first, with $A[7] = b_4$, which matches the first byte of the codeword. This is done for all the remaining words of the term, with the same verification being performed for the respective position of the words in the term. Then, the process continues for the following levels, using $rank$. Continuing with the term `that`, to determine where its second byte may be in the second level of the tree, a $rank_{b_4}(A,7) = 1$ is performed, concluding that the occurrence of the first byte of `that` is the first occurrence of b_4 in the root. Since we know what is the node on level two that corresponds to the second bytes after b_4, we access node C on position one, which is the result from the $rank$ and check if $C[1] = b_2$. Since that is true, and there are no more bytes in the codeword, it is possible to conclude that there is an occurrence of `that is` at position seven of the text.

To implement this behavior, I adapted the original Byte-codes Wavelet Tree code and created an additional structure, next to the **t_index** structure presented in the previous Chapter, to hold the relevant data for the search procedure to work. Figure 4.5 shows the structure definition. The **wt_search** structure holds the codewords for the searched term, *codes*, which also requires an additional structure, the less frequent word in the searched term, *lf*, its number of occurrences in the text, *lf_nr_occs*, and the position of the leaf of the occurrence being processed, *leaf_pos*.

```
struct code{
    char bytes[4];
    int branches[4];
    int len;
}

struct wt_search{
    struct code *codes;
    int nr_codes;
    int lf;
    int lf_nr_occs;
    int leaf_pos;
}
```

Figure 4.5: **wt_search** structure used in the Byte-codes Wavelet Tree monolingual search

The structure **code** has the number of *bytes* of the codeword, the pointers to where the bytemaps for the respective nodes start (*branches*) and the length (*len*) of the codeword. In this implementation of the Byte-codes Wavelet Tree, no codeword can have more than four bytes. Regarding the *branches* field, it is important to remember that the bytes in the Wavelet Tree are represented in a single bytemap, which then have some markers indicating the tree topology,

namely where a node / level starts and ends. The *branches* field points to where each node of the tree structure is represented in the bytemaps.

Algorithm 4.2 shows the generic procedure for determining the codewords in the Byte-codes Wavelet Tree. The function *get_vocabulary_positions* fetches the positions where the words in the searched term are in the vocabulary. Then, for every word in the text, the procedure fetches the codewords of every word, initializing the respective structure **code**, including its *bytes* and *branches*. During this process, the least frequent word of the searched term is chosen, using the position in the vocabulary of the words, since the vocabulary is sorted by frequency. Having the least frequent word calculated, then the procedure determines the number of occurrences in the indexed text of that term. That is done using the *rank* operation, in the last node of where the codeword is represented, with its last code, since the last node where a word is represented has all the occurrences of that word. To get to the last node, the procedure uses the *branches* field of the structure **code**, while for the last code it uses the *bytes* field.

```
struct wt_search* calculate_codewords(struct t_index *index, char* term){
  struct wt_search *search;
  int nr_positions;
  int* voc_positions = get_vocabulary_positions(index->voc, term,
    &nr_positions);
  search->nr_codes = nr_positions;
  int lf_word = MAX_INT;
  for (i = 0; i < search->nr_codes; i++){
    search->codes[i] = get_codeword(voc_positions[i], index);
    if (voc_positions[i] < lf_word){
      lf_word = voc_positions[i];
      search->lf = i;
    }
  }
  code c = search->codes[search->lf];
  search->lf_nr_occs = rank(index->tree->bmaps[c.branches[c.len-1]],
    size(index->tree->bmaps[c.branches[c.len-1]]) - 1, c.bytes[c.len-1]);
  return search;
}
```

Listing 4.2: Getting the codewords of a term in Byte-codes Wavelet Tree

Algorithm 4.3 shows the second part of the search procedure, where the actual occurrences are calculated. The procedure *get_occurrences* uses the **wt_search** structure and returns one occurrence of the term, namely its absolute offset in the indexed corpora.

To determine an occurrence of the term, the procedure is based on its least frequent word, starting from the node of the tree that has all the occurrences of the last byte of the respective codeword. For that matter, for every position of the last byte in the node, *leaf_pos*, the procedure ascends to the root, using the *select* function for the byte of the codeword respective to the level of the tree being processed. On reaching the root, the first bytes of all the codewords of the terms are verified, using the *chk_root_bytes* function. Then, the remaining words of the term are verified, by analyzing if the other bytes of those words match the respective positions in the respective nodes, using *rank* in the *chk_other_bytes* function. If one byte does not match, the procedure stops, as there is no occurrence of the term in that position.

```
int chk_root_bytes(struct bytemap **bmaps, struct wt_search *search, int pos){
  char* start = bmaps[0][pos - search->lf];
  for (i = 0; i < search->nr_codes; i++){
    if (start[i] != search->codes[i].bytes[0]){
      return 0;
    }
  }
  return 1;
}

int chk_other_bytes(struct bytemap **bmaps,struct wt_search* search,int pos){
  int i = 0, stop;
  while (i < search->nr_codes){
    stop = 0;
    pos = pos - search->lf + i;
    code c = search->codes[i];
    for (j = 1; j < c.len, j++){
      pos = rank(bmaps[c.branches[j-1], pos, c.bytes[j-1]) - 1;
      if (bmaps[c.branches[j]][pos] !=c.bytes[k]){
        stop = 1;
        break;
      }
    }
    if (stop){
      break;
    }
    i++;
  }
  return i == search->nr_codes;
}

int get_occurrence(struct bytemap **bmaps, struct wt_search *search){
  int pos = search->leaf_pos;
  while (search->leaf_pos < search->lf_nr_occs){
    pos = search->leaf_pos;
    code lf_code = search->codes[search->lf];
    for (i = lf_code.len - 1; i >= 0; i--) {
      pos = select(bmaps[lf_code.branches[i]), pos + 1, lf_code.bytes[i]);
    }
    search->leaf_pos++;
    if(chk_root_bytes(bmaps,search,pos) && chk_other_bytes(bmaps,search,pos)){
      return pos - search->lf_pos;
    }
  }
  return -1;
}
```

Listing 4.3: Getting an occurrence of a term in Byte-codes Wavelet Tree

The functions presented in the previous two Algorithms are the base of the *locate* function supported by Byte-codes Wavelet Trees. Algorithm 4.4 shows the *locate* procedure in pseudo-code using the *calculate_codewords* and *get_occurrence* functions. This procedure runs until no more occurrences of a term exist, returning them in a list. This process as a whole is considerably more complex than the alternative using Suffix Arrays, because the words are all codified in codewords, to save space. To decode them, it is necessary to use the $select$ and $rank$ operations, over the tree topology that represents the indexed compressed text.

```
int *locate (struct t_index *index, char *term, int *num_occs){
  struct search_t *search = calculate_codewords(index, term);
  int *occs = malloc(search->lf_occs * sizeof(int));
  int pos = 0, i = 0;
  while (pos != -1){
    pos = get_occurrence(bmaps, search);
    occs[i] = pos;
    i++;
  }
  *num_occs = i - 1;
  return occs;
}
```

Listing 4.4: *locate* procedure in Byte-codes Wavelet Tree

4.2.3 Calculating the co-occurrences

The monolingual search procedures described above return all the occurrences of a term in its respective corpora. Considering a pair X <->Y, the occurrences of X and Y are calculated first, using the Text Layer and its indices. However, to determine the co-occurrences of the pair, the sets of occurrences returned in both languages need to be filtered, to determine the pairs of occurrences that match *Condition 1* and *Condition 2*, and form valid co-occurrences.

A first approach to determine the co-occurrences, from the set of occurrences of X and Y, set_X and set_Y respectively, would involve traversing all the occurrences in set_X and set_Y. To improve that process, we started by defining a strategy based on binary searches. Taking into account that an occurrence of X cannot co-occur with more than one occurrence of Y, and vice-versa, it is trivial to infer that the number of co-occurrences will be, at most, equal to the number of occurrences of the less frequent term of the pair.

Consider that X has less occurrences than Y in the parallel corpora. To determine the co-occurrences, the bilingual search procedure traverses the smaller set of occurrences obtained from the Text Layer, set_X in this example, and for every occurrence in set_X, determine if there is an occurrence in set_Y that co-occur with it. That is done using binary searches over the larger set of occurrences, set_Y [39]. To perform binary searches, the set of occurrences needs to be sorted by order of appearance in the text, which means that in case of the implementation based on Suffix Arrays, the set needs to be sorted.

As discussed above, the texts in the two languages can be considerably different, given the different language characteristics. For that matter, *Condition 1* and *Condition 2* use the alignment information to determine if a co-occurrence is valid or not. Thus, for every occurrence processed in set_X and set_Y, the Alignment Layer is used to determine the coarse and fine segments where those occurrences appear. These coarse and fine segments are also used as search parameters for the binary searches over the smallest set of occurrences, influencing when the procedure continues to the left or to the right.

Algorithm 4.5 represents a simplified version of the first approach for the bilingual search, based on binary searches, with the definition of the functions *init_search* and *next_match* of the Bilingual Framework.

```
struct results_iter{
  int *occs[2];
  int nr_occs[2];
  int num_bilingual_matches;
  int pos;
  int linear;
  int bsect;
  int window;
}

struct results_iter *init_search (struct bil_fram * fram, char *term1, char *
    term2, int window){
  struct results_iter *it;
  it->nr_occs[0] = strlen(term1);
  it->nr_occs[1] = strlen(term2);
  it->occs[0] = fram->tapi[0]->locate(fram->ti[0], term1, &(it->nr_occs[0]));
  it->occs[1] = fram->tapi[1]->locate(fram->ti[1], term2, &(it->nr_occs[1]));
  it->pos = 0; it->window = window;
  if (it->nr_occs[0] > it->nr_occs[1]){
    it->linear = 1;
    it->bsect = 0;
  }else{
    it->linear = 0;
    it->bsect = 1;
  }
  sort(it->occs[it->bsect]);
  return it;
}

int next_match(struct bil_fram *fram, struct results_iter *it, int *off1, int
    *off2){
  int pos = 0, has_match = 0;
  while (has_match == 0 && pos < it->nr_occs[it->linear]){
    int occ_linear = it->occs[pos++];
    int coarse, fine;
    fram->aapi->locate(fram->ai, occ_linear, it->linear, &coarse, &fine);
    struct si_bounds bounds;
    fram->aapi->coarse_bounds(fram->ai, coarse, &bounds);
    int bis_pos = bisect(it->occs[it->bsect], bounds.begin_offs[it->bsect], 0,
         it->nr_occs[it->bsect]);
    int occ_bisect = it->occs[it->bsect][bis_pos];
    if (check_conditions(fram, it, occ_linear, occ_bisect)){
      *off1 = occ_linear;
      *off2 = occ_bisect;
      has_match = 1;
      it->num_bilingual_matches++;
    }
  }
  return has_match;
}
```

Listing 4.5: *init_search* and *next_match* procedures of the Bilingual Framework

Before the definition of the functions, it is important to introduce the structure **results_iter**, that enable the iterative behavior of the Framework, as introduced at the end of Chapter 2. This structure holds two arrays of occurrences (*occs*), one per term, the number of occurrences of the terms in each language (*nr_occs*) and the number of co-occurrences (*num_bilingual_matches*).

Additionally, the structure holds the position *pos* of the array that is traversed in a linear fashion and the limit *window* used by *Condition 2*. Finally, to represent which set of occurrences will be traversed in a linear fashion, the structure holds which side is going to follow that behavior, and the other side that is going to support binary searches (*bsect*). Here, by side, we mean one of the two languages of the parallel corpora.

The function *init_search* determines the individual occurrences of the two terms in the bilingual search, using the APIs of the implementations chosen for the Text Layer, defined in the structure **bil_fram**. Since each co-occurrence is determined one by one, using the binary search procedure, this function already determines all the individual occurrences, leaving the finding of the co-occurrences for the *next_match* procedure. Before finishing, this function determines which set of occurrences will be processed via binary searches, the bisect side, *bsect*, and the set that is going to be traversed in a linear fashion, the *linear* side. The bisect side is finally sorted by the order that the occurrences appear in the text.

Having the occurrences calculated, the *next_match* checks them to determine which ones match *Condition 1* and *Condition 2*. While there is no match yet, and there are still occurrences to process on the linear side, the procedure moves to the next occurrence on that side, thus in the smallest set of individual occurrences. Then, using the Alignment Layer, the procedure determines the alignment segments where that occurrence appears, and the bounds of the determined coarse segment, using *coarse_bounds*. This function returns the offsets where a coarse segment start and end, in both languages. This information is decisive to control the binary search, since we want to find occurrences on the other language within the same aligned coarse segments. For that purpose, the offset where the coarse segment starts in the bisect side, *coarse_bounds.begin_offs[it->bisect]*, is used on the binary search.

After finding the first occurrence on the bisect side that appears within the coarse segment, the function *check_conditions* determines if there is any co-occurrence. Within this function, the occurrence found on the bisect side is matched with the occurrence on the linear side, to try to meet the Conditions. Regardless of if the occurrences meet the Conditions or not, and since the bisect side is sorted by the order of the text, the following consecutive occurrences on the bisect side are checked, until getting to an occurrence that falls outside of the determined coarse segment. The occurrence that is closer to the occurrence on the linear side, is the one chosen to co-occur.

From the algorithm presented, it is possible to identify that the binary searches on the bisect side are always performed over the entire the set of occurrences of the terms. The reason for this lies on the Suffix Arrays and Byte-codes Wavelet Trees returning their set of occurrences in different orders, lexicographically and following the order of the text respectively. To avoid sorting both sets of occurrences, which can be considerable long for common terms in large corpora, we decided to implement the binary search considering that on the linear side, the occurrences are not sorted by any particular order. Thus, an occurrence on the linear side can have a smaller, or larger, offset than the previous one in the set.

This bilingual search procedure heavily depends on the Alignment Layer, to determine the coarse and fine segments where the occurrences appear. In particular, the *locate* operation

influences greatly the binary search that supports the procedure, as it returns the coarse and fine segment when an occurrence appears. These values are what *Condition 1* and *Condition 2* use to determine the co-occurrences.

As the Alignment Layer has two possible implementations, arrays of integers and bitmaps, the next subsections describe the two approaches for determining those alignment segments.

4.2.3.1 Using the Alignment Layer with Arrays of Integers

This Alignment Layer implementation is supported by five arrays of integers, two for the coarse segments, two for the fine segments and one to relate both. This approach uses binary searches to determine the coarse and fine segments where an offset, *off*, occurs. The value of *off* is calculated using the *locate* function of the Text Layer, introduced above. Figure 4.6 recovers the example of the Alignment Layer based on Arrays of Integers, presented in the previous Chapter.

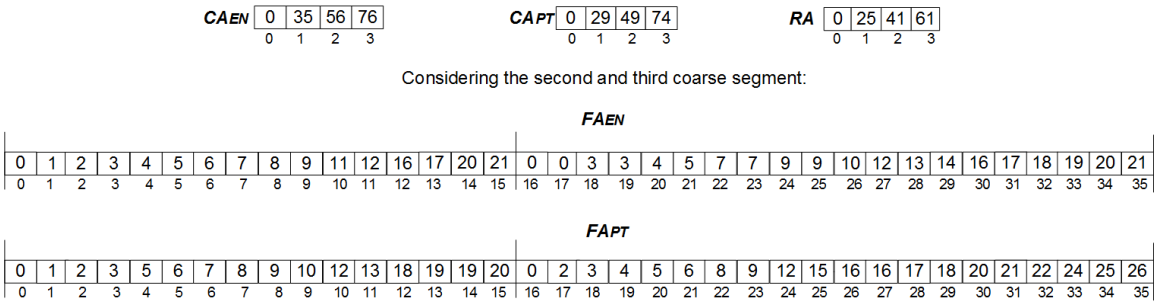

Figure 4.6: Alignment Layer supported by Arrays of Integers

Consider an example of a term in English with an occurrence at $off = 52$. To implement the $locate$ operation, the first step is to determine the coarse segment, *coarse*, where *off* appears in the alignment. To calculate the segment the procedure executes $coarse = bisect(CA_{EN}, 52, 0, ncoarse) = 1$, with *ncoarse* as the total number of coarse segments and *bisect* as the binary search function. The binary search is performed over the entire array, from 0 to *ncoarse*, and since the *bisect* is zero based, the occurrence appears in the second coarse segment of the alignment.

To determine the *fine* segment, the procedure continues with the arrays that hold the fine alignment representation. However, as the fine segments are represented based on the coarse alignment, first the procedure looks at RA array to determine the boundaries of the second coarse segment. Thus, $RA[coarse] = RA[1] = 25$ and $RA[coarse+1] = RA[2] = 41$, meaning that the second coarse segment occurs between the 25th and 41st fine segments.

As a side note, in Figure 4.6, the arrays for the fine segments have the first segment omitted, so this interval actually refers to the first sequence in the arrays (from position 0 to 15). However, for example purposes, I will refer to the Figure interval hereon, 0 to 15, which map to positions 25 to 41. Similarly, the fine segment determined in the next step is also dependent on the number of fine segments within the 1st coarse segment, which is 25. For that matter, it is necessary to add 25 to the value determined in this example, to obtain the actual fine segment.

The procedure continues with a binary search performed over the array for fine segments on the English side, FA_{EN}, between the calculated interval. As $CA_{EN}[1] = 35$, the fine segment is determined by $bisect(FA_{EN}, 52-35, 0, 15) = 13$. Since the fine segments are represented based on the coarse segments, the procedure adjusts the binary search to consider the absolute offset of the occurrence, 52, minus the offset where the coarse segment begins, 35.

Following the previous note, $fine = 13$ is actually limited by the absence of the 1st coarse segment from the fine bitmaps. This means that the actual $fine$ segment would be $RA[1]+13 = 38$, as $RA[1]$ holds the total number of fine segments within the 1st coarse segment. This extra calculation is not necessary to perform in the original algorithm, since the alignment would be properly represented in the respective arrays.

Algorithm 4.6 shows this procedure more formally, using the **s_index** structure and the $bisect$ operation that supports the binary search. It follows the process introduced above, for one language in the Framework, using *rel_begin* and *rel_end* to represent the RA interval. In a bilingual search, this process would be repeated as well for the second language.

```
void locate(struct s_index *index, int off, int side, int *fine, int *coarse){
  *coarse = bisect(off1, index->coarse_offs1, 0, index->ncoarse);
  int rel_begin = index->r_array[*coarse];
  int rel_end = index->r_array[*coarse + 1];
  *fine = bisect(off1 -  index->coarse_offs1[*coarse], index->fine_offs1,
      rel_begin, rel_end);
}
```

Listing 4.6: *locate* function based on Arrays of Integers

4.2.3.2 Using the Alignment Layer with Bitmaps

This approach for the Alignment Layer is based on three bitmaps, two for the fine segments and one for the coarse alignment. To support the $locate$ operation, the $rank$ and $select$ functions are used over the bits, to determine the coarse and fine segment where an occurrence appears. Figure 4.7 also recovers the example of the Alignment Layer based on bitmaps, introduced in the previous Chapter.

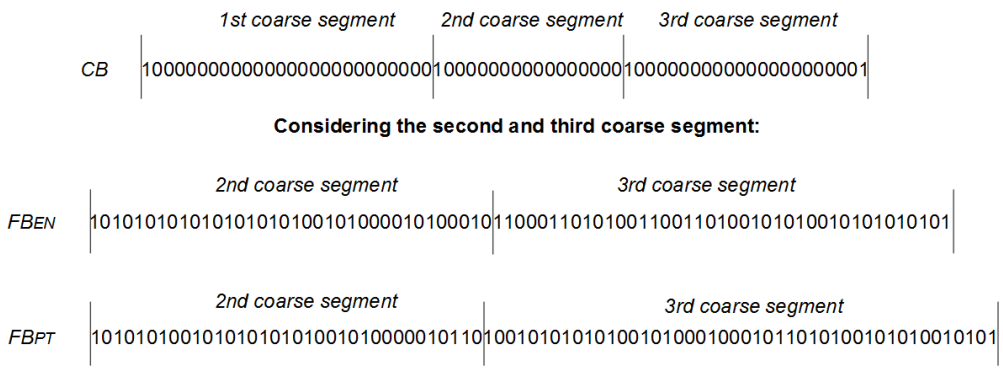

Figure 4.7: Alignment Layer supported by bitmaps

Consider the same example with an English term occurring at the offset $off = 52$. The $locate$ procedure starts with calculating the $fine$ segment where off appears. Since the bitmaps for the fine alignment are defined based on the absolute offsets of the text, where each 0 represents a word or symbol in the text, the first step is determining where off appears in the English bitmap. Just as with the example for the Arrays of Integers, Figure 4.7 omits the first coarse segment in the fine alignment representation, which means that the example is influenced by the number of words / symbols in the first coarse segment, which is 35 as seen in Figure 4.6. This means that for the example $off = 52 - 35$, while in a real world scenario would still be 52. To determine the fine segment, first the procedure does $select_0(FB_{EN}, off) = 30$, meaning that the off-th zero in the bitmap occurs at the 30th bit of FB_{EN}.

Knowing the position in the bitmap where off appears, the procedure counts the number of fine segments until that position, with $fine = rank_1(FB_{EN}, 29) = 13$, meaning that the occurrence appears at the 13th fine segment.

To determine the coarse segment, since the corresponding bitmap is based on the number of fine segments per coarse segment, the procedure follows a similar behavior, but using $fine$ instead of off. Taking into consideration that the first coarse segment has 25 fine segments, then here $fine = 25 + 13 = 38$. Thus, the procedure does $select_0(CB, 38) = 40$, meaning that the 38th fine segment appears at the 40th bit of CB. Then, we perform $coarse = rank_1(CB, 40) = 2$, meaning that off appears at the 2nd coarse segment of the alignment.

Despite needing less data structures as the other alternative for the Alignment Layer, this bitmap-based approach requires more operations to determine the $fine$ and $coarse$ segments, as the information is codified in bits, while with the arrays it is directly available in the structures.

Algorithm 4.7 shows the $locate$ function based on bitmaps, which uses the $rank$ and $select$ operations as described earlier. This algorithm also describes the steps just for one language, with the second following the exact same strategy. The procedure uses two auxiliary values, off_pos to hold the position in the bitmap for the fine segments where off appears, and $fine\_pos$ to hold the position in the bitmap for the coarse segment where $fine$ appears.

```
void locate(struct s_index *index, int off, int side, int *fine, int *coarse){
  int off_pos = select(index->fine1, off, 0);
  *fine = rank(index->fine1, off_pos, 1);
  int fine_pos = select(index->coarse, *fine, 0);
  *coarse = rank(index->coarse, fine_pos, 1);
}
```

Listing 4.7: *locate* function based on bitmaps

4.2.4 Using the Document Layer

Without direct connection to the bilingual search, the Document Layer filters a search over a specific document, or documents within a directory, that are part of the parallel corpora. Additionally, it can also be queried for information about the documents where a specific searched term appears. The *locate* operation of the Document Layer API returns the document and

directory where an occurrence at offset *off* appears, by using its alignment placement, more specifically, its coarse segment. Additionally, the *get_docname* and *get_dirname* operations return the name of the document or directory, respectively, determined in the *locate* operation.

Figure 4.8 recovers the example of the Document Layer from the previous Chapter, with four directories and ten documents. The documents and directories are represented with arrays of integers, where the directories, despite the representation in the Figure, are represented with two arrays: one for the beginning offsets and one for the ending offsets.

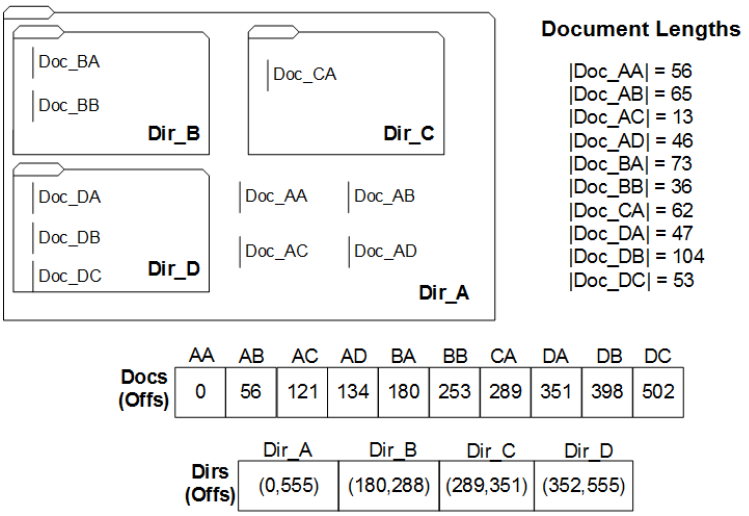

Figure 4.8: Document Layer

To determine the document or directory where an offset *off* occurs, the array where the documents are represented is processed using binary searches. For example, consider an offset *off* that appears at the coarse segment $coarse = 290$. Going through the *Docs* array using a binary search, the procedure determines that the occurrence appears at document *Doc_CA*. Using again binary searches, going through the array with the beginning offsets of each directory, we would get to the directory *Dir_C*, which by analyzing Figure 4.8, has *Doc_CA* inside it.

This approach for a bilingual search considers all the individual occurrences found on both texts. Even considering that binary searches are performed, all the occurrences in one side are processed, and for each one, a binary search is performed on the occurrences of the other language. Then, per every occurrence processed, the Alignment Layer is queried to obtain the information from the alignment. This is a complex procedure, that may take some time to return the co-occurrences of a pair of terms. The next Section introduces a speed-up that we designed to improve the bilingual search time response of the Bilingual Framework.

4.3 Skip-based Bilingual Search Procedure

In parallel corpora, with texts in different languages, it is not uncommon to encounter several difficulties for the task of alignment such as changes of order of the words or sentences, different language and morphological constructions, but also ambiguity. For example, the English

term `plant` can be translated to Portuguese in several different ways, with different meanings, such as: 1) `planta`, which can mean a *plant* in nature, like a flower, tree, etc, or a *plant* in architecture, like a sketch and blueprint of a building or room; 2) `fábrica` or `instalação`, as in a factory or power plant; 3) `semear`, as in cultivating crops in agriculture; etc.

Although ambiguity can bring difficulties to Machine (and even man-made) Translation tasks, it actually has some characteristics that can be used as an asset, in particular for a bilingual search. Considering a bilingual search for the EN-PT pair `plant` <->`fábrica` over an aligned parallel corpus, it would not be uncommon to find occurrences of `plant` co-occurring with other of its possible translations, like `planta` or `instalação`. For this particular search pair, the occurrences of `plant` would not be relevant to the result of the bilingual search, as only the co-occurrences with `fábrica` would matter. This information can thus be used to improve the response speed of a bilingual search.

It is known that for two occurrences of a pair of terms to co-occur, in a bilingual search, they must appear in aligned coarse segments, as defined in *Condition 1*. The bilingual search procedure can capitalize on this Condition, and on the mentioned ambiguity, to skip occurrences of the terms, that do not appear coarsely aligned with occurrences of the other term in the search pair, thus not meeting *Condition 1* [40].

Figure 4.9 shows an abstract representation of seven aligned coarse segments of an English-Portuguese parallel corpus. Throughout the aligned segments, there are occurrences of the terms `plant`, in English, and `fábrica`, `planta` and `instalação` in Portuguese. Additionally, there are some occurrences of `factory` in English represented as well, which is also a correct English translation for `fábrica`. The occurrences are located in several aligned coarse segments from $coarse_1$ to $coarse_7$, in positions (A) to (N).

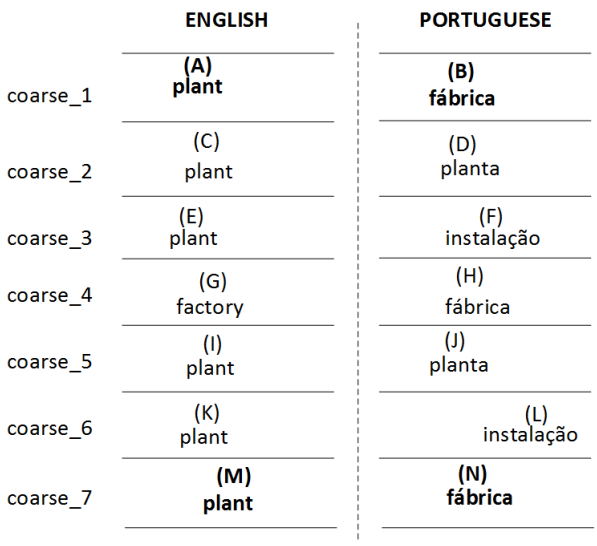

Figure 4.9: Representation of seven coarse segments with several occurrences divided through the segments. The two co-occurrences of `plant` <->`fábrica` are depicted in bold font

Considering the bilingual search pair `plant` <->`fábrica`, Figure 4.9 shows two possible

co-occurrences in $coarse_1$ and $coarse_7$. The remaining four occurrences of `plant` and occurrence of `fábrica` do not appear in aligned coarse segments, thus possibly co-occurring with different terms. Taking this into account, we designed a strategy to skip several of these occurrences that we know would not for sure be apart of a valid co-occurrence for the searched term. By skipping occurrences, the bilingual search time response can improve and return the co-occurrences faster. We denominated this procedure as skip-based bilingual search.

The main idea behind skipping an occurrence in this bilingual search approach is based on *Condition 1*. For instance, occurrence (*H*) in $coarse_4$ does not have a respective occurrence of `plant` in the aligned $coarse_4$. The same thing also happens in English, with occurrences (*C*), (*E*), (*I*) and (*K*) not appearing in aligned sentences with any occurrence of `fábrica`. This makes the skip-based bilingual search heavily dependent on *Condition 1*, and thus also on the Alignment Layer. Being the skip-based bilingual search based on *Condition 1*, consider $coarse(off)$ as the coarse segment where off appears. This value is key to the bilingual search procedure, as it enables filtering out the occurrences that are non relevant to its result.

Consider the example in Figure 4.9 and the search pair *X* <->*Y*, where *X* = `plant` and *Y* = `fábrica`. A skip-based bilingual search for the pair *X* <->*Y* would begin with $off_X = (A)$ and $off_Y = (B)$. As $coarse(off_X) = coarse(off_Y)$, *Condition 1* is met. Then, *Condition 2* is verified to determine if the occurrences are a valid bilingual co-occurrence of the pair[2].

Continuing the procedure to the next occurrences of both terms, would result in $off_X = (C)$ and $off_Y = (H)$. These occurrences appear in non-aligned coarse segments, as $coarse(C) = coarse_2$ and $coarse(H) = coarse_4$. As $coarse_2 \neq coarse_4$, *Condition 1* is not met and there is no possibility of existing a co-occurrence with (*C*) and (*H*). At this point, and considering the text alignment, it is possible to infer that no occurrence of *X* will co-occur with an occurrence of *Y* before $coarse_4$. Here, the bilingual search procedure can skip all the occurrences of `plant` until the first occurrence of the term that appears after the beginning of $coarse_4$.

This is the key logic behind the skip-based bilingual search, that avoids processing occurrences of a term, such as (*E*) in this example, and moving directly to the next possible occurrence of the term that possibly enables a co-occurrence. Going back to the example, the skip results in $off_X = (I)$, with $coarse(I) = coarse_5$, which is the first occurrence of *X* after $coarse_4$. However, $off_Y = (H)$ and $coarse(H) = coarse_4$ and $coarse_5 > coarse_4$, meaning that *Condition 1* is still not met.

As the previous step demonstrated, this skip-based approach does not guarantee that both occurrences of the terms will appear in aligned segments after the skip. When this occurs, the procedure continues to follow the same logic, thus skipping more occurrences until getting to two occurrences on the same aligned coarse segments. However, now it is the occurrence of `fábrica` that appears behind in the aligned corpora, with $coarse_4 < coarse_5$. This means that the skip now is made on the Portuguese side, for the first occurrence of `fábrica` after the beginning of $coarse_5$. The skip results in $off_Y = (N)$ at segment $coarse_7$.

[2] In this example, the main interest lies on using *Condition 1* to apply jumps and not verifying if the occurrences meet *Condition 2*

As the occurrences are not aligned yet, the procedure focuses again on the English side, as $coarse_5 < coarse_7$, and skips to the first occurrence of `plant` after the beginning of $coarse_7$, resulting in $off_X = (M)$. Finally, $coarse(off_X) = coarse(off_Y)$, *Condition 1* is met and there is a possible co-occurrence, that is verified with *Condition 2*. As there are no more occurrences of the terms, the bilingual search procedure ends, with the two possible co-occurrences found.

In this small example, the bilingual search procedure skipped entirely the occurrences (*E*) and (*K*) of `plant`, while it did not checked *Condition 2* for occurrences (*C*) and (*I*) of `plant` and occurrence (*H*) of `fábrica`. This already shows how this bilingual search procedure enables speeding up the bilingual search time response, by not processing some occurrences that are not relevant to the bilingual search being executed. Transporting this to larger examples, with parallel corpora of Gigabytes, the ambiguity would be even larger and these skips would be more often. The Results Section of this Chapter shows the number of occurrences skipped in some example parallel corpora.

4.3.1 Occurrences within the same coarse segment

In a bilingual search procedure, it is not uncommon to find a scenario where more than one occurrence match *Condition 1*, within aligned coarse segments. The example above only showed a scenario where that does not occur, however it is something that the skip-based bilingual search supports. Consider Figure 4.10, with a deviation from the previous example, showing three occurrences of `plant` and two occurrences of `fábrica` in $coarse_7$.

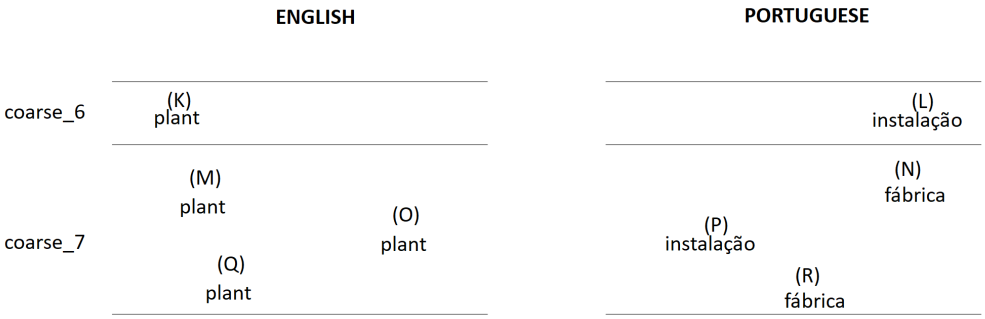

Figure 4.10: Representation of several occurrences within aligned coarse segments

The example in Figure 4.10 shows that there are several occurrences of `plant` and `fábrica` in $coarse_7$, all meeting *Condition 1*. This makes a combination of six possible co-occurrence possibilities, just within this $coarse_7$. In these cases, all these occurrences need to be verified, to determine which ones meet *Condition 2*. If there is a case when more than one occurrence, in one language, meet *Condition 2* with an occurrence of the other language, then the bilingual search procedure chooses the closest ones, based on the aligned fine segments between them. For instance, if occurrence (*O*) meets both Conditions with occurrences (*N*) and (*R*), then the one closest to it would form the co-occurrence. No occurrence can be part of more than one co-occurrence pair.

In this scenario, when more than one occurrence of a term appears within the same coarse segment, the skip-based search procedure gathers all the occurrences inside the aligned sentences, from both sides of the aligned corpora, and then selects the pairs of occurrences that meet *Condition 2*. If an occurrence of term Y, off_Y, meets *Condition 2* with two occurrences of a term X, off_{X1} and off_{X2}, then the closest occurrences are chosen, using $min(fine(off_{X1}) - fine(off_Y), fine(off_{X2}) - fine(off_Y))$. This expression enables measuring the distance between two occurrences, in two different texts, using the fine alignment segments.

To implement this behavior in the Bilingual Framework, there are some changes that need to be done. Thus far, the Text Layer returns all the occurrences of a term at once, while this approach requires a more iterative behavior, where an occurrence is returned at a time.

4.3.2 Adapting the Text Layer

To support the iterative nature of the skip-based search procedure, the first change that was necessary to perform was at the level of the Text Layer API. For that, I replaced the *locate* function for three new functions, *start_locate*, *locate_next* and *end_locate*. Figure 4.11 shows the newer version for the Text Layer API, considering this iterative approach.

```
struct t_index;
struct t_occ_iter;

struct ti_api{
    void create_index (char* outfile, char*
        text, int text_len, struct ti_api *api);

    struct t_index* load_index(char* infile,
        struct ti_api *api);

    void free_index(struct t_index* index);

    struct t_occ_iter* start_locate (struct
        t_index *index, char* term, int *start,
        int end);

    int locate_next(struct t_index* index,
        struct t_occ_iter* it, int *skip);

    void end_locate(struct t_index* index,
        struct t_occ_iter* it);

    char* get_snippet (struct t_index *index,
        int off, int len, int* len_bytes);
}
```

Figure 4.11: Second version of the Text Layer API

These three new functions added to the API can also replace the $count$ function, since now the search will follow an iterative behavior, returning one occurrence at a time. The $count$ behavior can be easily supported by the new three functions in the API.

To begin the search for a term, the function *start_locate* initializes all the data that is needed for that search be performed. Besides the index and the term being searched for, the function

also uses an input, *start*, that enables starting a search for a certain offset of the text, instead of always from the beginning. It also has an *end* field to determine, if desired, a last offset of the text allowed to bound the search. This function returns the iterator that will work as basis for the entire search procedure, as well as the offset where the actual search starts. For instance, consider that a new search is triggered, starting at offset 100 of the indexed text. If the first occurrence of the searched term is at offset 120, then the *start* input would return that value. The *end_locate* functions ends the search by freeing the iterator from main memory.

The function *locate_next* returns the next occurrence of the term, in the context of the search started before. To accomplish that, it uses the index and the iterator, as well as an additional input, *skip*. This input works similarly to the *start* input of the previous function, in the sense that determines from where the search must resume. This value is extremely important for the skip-based search procedure, since it controls the search according to the skips made. It is important to understand that this *skip* field does not have the occurrence that is returned in the current iteration, but the next possible occurrence of the term instead, that is used by the procedure to determine where it is going to skip to, in the next iteration.

All these three functions use an iterator, which is represented in this Framework by the structure **t_occ_iter**. This structure is not yet defined in Figure 4.11, because it depends on the implementation of the Text Layer. In the next Subsections, I introduce the modifications on both implementations of the Text Layer, to take into account the iterative behavior of the skip-based search, including the respective definitions of the structure **t_occ_iter**.

4.3.2.1 Adapting the search over Word-based Suffix Arrays

The Word-based Suffix Array is an index that represents the suffixes of the indexed text following a lexicographic order, being the sets of occurrences following the same order. The skip-based bilingual search depends heavily on skips that follow the order of the text, thus making this lexicographic order not useful. Considering this fact, we decided to apply the skip-based search procedure, exclusively to the final part of the bilingual search, when the two sets of occurrences are being processed to determine the co-occurrences.

To implement the new Text Layer API, the first step is to define the structure **t_occ_iter**, with the necessary fields to implement the iterative behavior over the Word-based Suffix Array. Figure 4.12 shows the definition of the structure for the Word-based Suffix Arrays. Since the occurrences are returned following a lexicographic order, the structure holds the set of occurrences (*occs*), pre-calculated already, alongside the number of occurrences (*nr_occs*). Since the search now follows a purely iterative behavior, the structure stores the current position, *cur_pos*, to keep track of the last occurrence processed in *occs*. Additionally, the structure holds the position of the next occurrence in *occs*, *next_pos*. This value is key for the skips, since it helps controlling where the next valid occurrences in both languages are, thus being decisive for calculating to which place of the aligned corpora the search skips to. The structure also holds the number of words of the term and, finally, the interval that delimits the search. This interval determines that the search is valid between the starting offset, *search_begin*, and the ending

offset, *search_end*.

```
struct t_occ_iter{
    int* occs;
    int nr_occs;
    int cur_pos;
    int next_occ;
    int nr_tokens;
    int search_begin;
    int search_end;
}
```

Figure 4.12: Structure **t_occ_iter** in Word-based Suffix Arrays

Applying the skip-based search procedure on Word-based Suffix Arrays does not require a change on the process of determining the sets of occurrences. When the search for a term in the respective language starts, the set of occurrences is then determined, without any filters of the skips, due to the text being indexed following a lexicographical order. To apply the skips, both sets of occurrences are sorted following the order of the text. This is the first change when comparing with the alternative based on binary searches, where only one set was sorted.

Algorithm 4.8 shows the *start_locate* function supported by Word-based Suffix Arrays. The first part of this function is very similar to the previous implementation of the *locate* function, with the calculation of the *ids*, using the vocabulary, followed by determining the interval of occurrences in the Suffix Array, using binary searches. After the sets of occurrences are stored in the iterator structure, the occurrences are sorted in an ascending order, to match the order they appear in the text.

Then, if the *start* input is not zero, meaning that the search will start from a given offset and not from the beginning, a new binary search is done, this time over the set of occurrences, to find the position in the array where that offset, or the one immediately after it, appears. This field allows keeping track of where the search is.

Finally, and since this procedure supports the skip-based bilingual search strategy, the *next_occ* field is determined. This field is crucial for determining where the bilingual search procedure skips to, since it will indicate the offset of the next occurrence of the term. Thus, if the occurrence at position *cur_pos* appears before the established *end* of the search interval, the *next_occ* field is set to *it->occs[it->cur_pos]*. This way, the field holds the offset of the next valid occurrence.

Having the sets of individual occurrences calculated, the skips are applied directly to the sets of occurrences pre-calculated in the *start_locate* method, to attempt avoiding processing all the occurrences of at least one language, the linear side. In the iterative behavior of the skip-based bilingual search, whenever a new individual occurrence of a term in one language is processed, the bilingual search procedure sends as input the offset of the skip, to allow skipping over the non-relevant occurrences in between. Binary searches are still used over the sets of occurrences, but filtered by the positions of the skips, instead of using the whole text. In the Text Layer API, that is tackled in the *locate_next* function, with the *skip* input.

```
struct t_occ_iter* start_locate(struct t_index *index, char* term, int *start,
     int end){
  struct t_occ_iter *it;
  int ntoks = 0, num_occs = 0;
  int *ids, occs;
  for(every token in term){
    ids[ntoks] = get_id(index->voc, token);
    ntoks++;
  }
  int beg = bisect(ids,ntoks,index->suffix_array,index->text,0,index->len);
  int end = bisect(ids,ntoks,index->suffix_array,index->text,beg,index->len);
  num_occs = end - beg;
  if (num_occs == 0){
    *start = MAX_INT;
    return it;
  }
  for(i = beg; i < end; i++){
    it->occs[i] = index->sa[i];
  }
  it->nr_tokens = strlen(term);
  it->nr_occs = num_occs;
  it->search_begin = *start;
  it->search_end = end;
  sort(it->occs, it->nr_occs);
  if (*start == 0){
    it->cur_pos = 0;
  }else{
    it->cur_pos = bisect(*start, it->occs, 0, it->nr_occs) - 1;
  }
  if (it->occs[it->cur_pos] + it->nr_tokens < end){
    it->next_occ = *start = it->occs[it->cur_pos];
  }else{
    it->next_occ = *start = MAX_INT;
  }
  return it;
}
```

Listing 4.8: *start_locate* function on Word-based Suffix Arrays

After the occurrence is determined, the possible next occurrence of the term needs to be calculated, to be used in the calculation of the following skip. If the next occurrence in the set *occs* of the iterator structure is still within the search interval, then that is the next occurrence of the term. In the next iteration, if the skip is the same offset than that next occurrence, no binary search is needed, since the occurrence was already calculated.

Algorithm 4.9 shows the *locate_next* procedure for Word-based Suffix Arrays. The algorithm starts by checking if the *skip* input appears within the search interval defined in the *start_locate*. If it does, then we check if it is equal to the field *next_pos* of the iterator structure. If it is, it means that it is the next occurrence in the set *occs*. If it is not, then the binary search is performed to get the position in *occs* of the next occurrence, after the *skip* position. Then, the new *skip* value is determined, with the next occurrence in *occs*.

Even considering that the skip-based bilingual search still requires calculating all the occurrences of a term, that process is quite fast with Suffix Arrays. The most time consuming

```
int locate_next(struct t_index *index, struct t_occ_iter *it, int *skip){
  if (*skip > it->search_end || *skip < it->search_begin){
    *skip = it->next_pos = MAX_INT;
    return -1;
  }
  if (*skip != it->next_pos){
    it->cur_pos = bisect(*skip, it->occs, 0, it->nr_occs);
    if (it->cur_pos = it->nr_occs){
      *skip = it->next_pos = MAX_INT;
      return -1;
    }
  }
  int off = it->occs[it->cur_pos++];
  if (it->cur_pos == it->nr_occs){
    *skip = it->next_pos = MAX_INT;
  }else{
    *skip = it->next_pos = it->occs[it->cur_pos];
  }
  return off;
}
```

Listing 4.9: *locate_next* function on Word-based Suffix Arrays

part of the bilingual search is determine the co-occurrences. Even considering that both sets of occurrences need to be sorted and several binary searches are made on both sides, no set is traversed in a linear fashion. This enables saving time, as we demonstrate later in this Chapter.

4.3.2.2 Adapting the search in Byte-coded Wavelet Trees

The Byte-codes Wavelet Tree represents the words of the indexed text in a compressed fashion, following the order they appear in the text. That makes already an important difference for the skip-based bilingual search, when compared with the alternative based on Suffix Arrays. With this approach, all the occurrences of a term also follow the order of the text.

This order is decisive for supporting the skip-based bilingual search, since no sorting is required for the sets of occurrences. Considering that the text is already represented following the desired order, with Byte-codes Wavelet Trees, the skips can be directly applied over the index, instead of calculating all the occurrences. Considering that this is a compressed index, and the occurrences of a term are not as directly accessible as in a Suffix Array, this is of extreme importance, since it saves processing time.

In the Byte-codes Wavelet Tree, the representation of the words requires accessing two different nodes of the tree: the root, for determining the absolute offsets where the words appear, and the node where the final byte of the respective codeword is represented. Whenever a new position of the skip is calculated, the monolingual search procedure $descends$ the tree with that position, to determine the occurrence in the respective last node of the tree. With that occurrence, then $ascends$ again, to determine the actual absolute offset of the possible next occurrence, after the skip.

Another major difference from the Byte-codes Wavelet Tree to the Word-based Suffix Array alternative, lie on the next occurrence that is determined to help the next skip. Due to the way

the Byte-codes Wavelet Tree is structured, and due to the search heavily centered on the least frequent word of the searched term, the position of the next occurrence that is calculated, is in fact the position of the next occurrence of the least frequent word. We could, at that moment, also check if the remaining words of the term appeared alongside the occurrence of the least frequent, to ensure that in fact there was an occurrence of the term in that position. However, since the position of the least frequent word is sufficient to guide the skip-based procedure, we opted for not doing that to avoid also processing overhead.

Similarly to the Suffix Arrays, the first step for implementing the new Text Layer API with Byte-codes Wavelet Tree, is the definition of the structure **t_occ_iter**. Figure 4.13 shows the definition of this structure for Byte-codes Wavelet Trees. The structure has the codewords of the words in the searched term (*codes*), the number of codewords (*nr_codes*), the position in the term of its least frequent word (*lf*), the number of occurrences of the least frequent word (*lf_nr_occs*) and the position in the leaf of the occurrence being processed (*leaf_pos*). Additionally, the structure has a field to represent the position in the root of the occurrence being processed (*root_pos*). These two fields are decisive for the *ascending* and *descending* behavior explained above, as they help controlling which occurrence is being processed and which occurrence will be returned for the following skip.

```
struct t_occ_iter{
    int root_pos;
    struct code *codes;
    int nr_codes;
    int lf;
    int lf_nr_occs;
    int leaf_pos;
}
```

Figure 4.13: Structure **t_occ_iter** in Byte-codes Wavelet Trees

When the search for a term starts in a Byte-codes Wavelet Tree, the only values that are determined are the positions, in the root and in the leaf, where the first possible occurrence of the term appears. The position at the root is then used by the next iteration, to help the skip-based bilingual search on determining in which position the search will continue.

Algorithm 4.10 shows a simplified version of the *start_locate* function supported by Byte-codes Wavelet Trees. The first steps, besides initializing the iterator structure, determine the codewords of the words in the searched term are calculated, much like in the original *locate* functionality, with the usage of the vocabulary and the finding of the least frequent word in the searched term. Then, using *rank*, the number of occurrences of the least frequent word is determined and saved in the structure, in the field *lf_nr_occs*.

The final steps of this procedure lie on calculating the *leaf_pos* and *root_pos* fields. The first one, corresponds on a cursor located at the node where the last byte of the codeword of the least frequent word on the term, that is going to be used for the iterative behavior. If there is no interval restrictions for the search, that position starts at zero, to start from the very first occurrence of the word. However, if the search starts from a specific offset, instead of the

```
int ascend_by_pos(struct bytemap **bmaps, struct t_occ_iter *it, int pos){
  struct code lf_code = it->codes[it->lf];
  for (int i = lf_code.len - 1; i >= 0; i--){
    pos = select(bmaps[lf_code.branches[i]], pos+1, lf_code.bytes[i]);
  }
  return pos;
}

int descend_by_pos(struct bytemap **bmaps, struct t_occ_iter *it, int pos){
  struct code lf_code = it->codes[it->lf];
  for (int i = 0; i < lf_code.len; i++){
    pos = rank(bmaps[lf_code.branches[i]], pos, lf_code.bytes[i]) - 1;
  }
  return pos;
}

struct t_occ_iter* start_locate(struct t_index *index, char* term, int *start,
     int end){
  struct t_occ_iter *it = malloc(sizeof(struct t_occ_iter));
  int *voc_positions = get_vocabulary_positions(index->voc, term, &(it->
      nr_codes));
  int lf_word = MAX_INT;
  for (i = 0; i < it->nr_codes; i++){
    it->codes[i] = get_codeword(voc_positions[i], index);
    if (voc_positions[i] < lf_word){
      lf_word = voc_positions[i];
      it->lf = i;
    }
  }
  struct code c = it->codes[it->lf];
  it->lf_nr_occs = rank(index->tree->bmaps[c.branches[c.len-1]],
    size(index->tree->bmaps[c.branches[c.len-1]]) - 1, c.bytes[c.len-1]);
  if (*start != 0){
    it->leaf_pos = descend_by_pos(index->tree->bmaps, it, *start);
  }
  it->root_pos = ascend_by_pos(index->tree->bmaps, it, it->leaf_pos);
  *start = it->root_pos - it->lf;
  return it;
}
```

Listing 4.10: *start_locate* function on Byte-codes Wavelet Trees

beginning, the procedure needs to determine the respective *leaf_pos*, at the last node of the respective codeword. For that, the procedure $descends$ the tree using $rank$, in an auxiliary function that I denominated *descend_by_pos*, from the root to the last node.

Having the *leaf_pos* calculated, to determine the respective position for that occurrence in the root, leads to $ascending$ the tree, starting at *leaf_pos*. To implement that process, I created an auxiliary function, *ascend_by_pos*, that uses the *select* procedure, from the last node to the root, to get progressively the position in the upper node. With this calculation, the function ends with the information of the occurrence of the least frequent word in the searched term, which is going to be later used to calculate the position of the next skip.

During the skip-based bilingual search, whenever a new individual occurrence of a term is asked for, the Byte-codes Wavelet Tree implementation determines the next valid occurrence

of the term, considering the position of the skip. This means that some occurrences of the term may not be processed at all, for a particular search, which differs from the implementation based on Word-based Suffix Arrays. However, these skips also lead to some extra work over the tree structure of the Byte-codes Wavelet Tree.

Whenever a new individual occurrence is calculated, if the position of the text given by the skip is the same as the one predicted, stored in the field *root_pos*, then that position can be used directly. However, since this position is an occurrence of the least frequent word in the text, the procedure verifies if the remaining words occur around it, following the correct placement of the words in the term. This process is exactly the same as the one used in the original *locate* procedure, introduced above in this Chapter.

If the position of the skip is different from the value of *root_pos*, then we need to $descend$ the tree by that given position, to find the new corresponding *leaf_pos* in the last node of the respective codeword. Since the position of the skip is new, it is important to discover the first occurrence of the term, after that skip offset. Thus, we need to find the respective occurrence in the “leaf"node, of that occurrence. After finding that position, the same process is applied of $ascending$ the tree to find the respective absolute offset of that occurrence, the *root_pos*. Then, since this is only an occurrence of the least frequent word in the term, the remaining words of the term are verified, to determine if it matches an occurrence of the full term.

Algorithm 4.11 shows the *locate_next* procedure for Byte-codes Wavelet Tree. The function *update_root_pos*, $ascends$ the tree to determine the position of an occurrence of a word at the root. The function *chk_occurrence* checks if the position determined in the previous function matches an occurrence of the term, with the help of two previously introduced functions to verify the remaining words of the searched term at the root, *chk_root_bytes*, and at the other levels, *chk_other_bytes*. These functions vary slightly, since they now use the iterator structure as input, but the implementation is kept the same. If the occurrence matches, then the next possible occurrence of the term is determined, for the next iteration of the skip-based search procedure. If not, the procedure keeps trying until an occurrence is found, or there are no more occurrences to process.

As previously stated, these modifications prevent processing some occurrences that otherwise could be returned in the previous implementation of the bilingual search. This improves the bilingual search processing time, as demonstrated in the Results Section of this Chapter. However, it is also important to understand that additional $ascends$ and $descends$ in the tree are made, which also is a factor to consider in the final performance of the whole procedure.

4.3.3 Skip-based Bilingual Search on the Bilingual Framework

Thus far, the modifications for supporting the skip-based bilingual search, focused on the Text Layer. That happens because the Alignment and Document Layers do not require any change. The skips are applied over the texts, using their absolute offsets, thus the representation of the alignment and of the location of the texts in disk can maintain their implementations.

```
void update_root_pos(struct t_index *index, struct t_occ_iter *it, int *pos){
  if (it->leaf_pos == it->lf_nr_occs){
    *pos = it->next_pos = MAX_INT;
  }else{
    it->root_pos = ascend_by_pos(index, it, it->leaf_pos);
    *pos -= it->lf;
  }
}

int chk_occurrence(struct bytemap **bmaps, struct ti_occ_iter *it, int occ){
  return (chk_root_bytes(bmaps,it,occ) && chk_other_bytes(bmaps,it,occ));
}

int locate_next(struct t_index *index, struct t_occ_iter *it, int *skip){
  *skip += it->lf;
  if (*skip != it->root_pos){
    it->leaf_pos = descend_by_pos(index, it, *skip);
    update_root_pos(index, it, skip);
  }
  while (it->leaf_pos < it->lf_nr_occs && !chk_occurrence(index->tree->bmaps,
      it, it->root_pos)){
    it->leaf_pos++;
    update_root_pos(index, it, skip);
  }
  if (it->leaf_pos >= it->lf_nr_occs)
    return -1;
  int off = it->root_pos - it->lf;
  it->leaf_pos++;
  update_root_pos(index, it, skip);
  return off;
}
```

Listing 4.11: *locate_next* function on Byte-codes Wavelet Tree

However, it is important to recover that the Alignment Layer is crucial for the skip-based search procedure, to determine the skips and check *Condition 1* and *Condition 2*.

Algorithm 4.12 shows the main procedure, in pseudo-code, of the skip-based bilingual search. As the main behavior of the bilingual search changed, to support the skips, the first modification lies on the **results_iter** structure. On the original approach, the sets of occurrences were stored, alongside a pointer to one of the sets, to guide the iteration over it.

This new version of the structure, holds two iterator structures, **t_occ_iter**, one for each language, used by the Text Layer (*its*). Additionally, this structure holds the information about the individual occurrences of co-occurrence, with its offsets (*off*), and the *fine* and *coarse* segments where the occurrences appear, alongside the coarse boundaries. Then, instead of holding the linear and bisect side, as in the original procedure, the structure holds one side, the *skip_sid*, which represents the language with the occurrence behind in the alignment. Recovering the explanation of the skip-based search procedure, when two occurrences in both languages do not appear in aligned coarse segments, it is the one behind that has a skip directly applied to it. Also related to the skips is the *next_pos* field, which represents the position of the next possible occurrences of both terms, to decide the next skip. Finally, the structure maintains the delimiting *window* to verify *Condition 2*, the number of co-occurrences found (*num_bilingual_matches*)

```
struct results_iter{
  struct t_occ_iter *its[2];
  int off[2];
  int fine[2];
  int coarse[2];
  struct si_bounds coarse_bounds;
  int num_matches[2];
  int num_bilingual_matches;
  int window;
  int skip_side;
  int next_pos[2];
}

int next_bilingual_occurrence(struct bil_fram *fram, struct results_iter *it){
  if (it->next_pos[0] == MAX_INT || it->next_pos[1] == MAX_INT){
    return 0;
  }
  int found_match = 0; int other_side;
  while (!found_match){
    if (!next_monolingual_occurrence(fram, it, 0)){
      return 0;
    }
    if (!next_monolingual_occurrence(fram, it, 1)){
      return 0;
    }
    while(it->coarse[0] != it->coarse[1]){
      if (it->coarse[0] < it->coarse[1]){
        it->skip_side = 0;
      }else{
        it->skip_side = 1;
      }
      other_side = !it->skip_side;
      int coarse_ahead = it->coarse[other_side];
      fram->aapi->coarse_bounds_one_side(fram->ai, coarse_ahead, other_side,
          &(it->coarse_bounds));
      it->next_pos[it->skip_side] = max(it->next_pos[it->skip_side], it->
          coarse_bounds.begin_offs[it->skip_side]);
      if (!next_monolingual_occurrence(fram, it, skip_side)){
        return 0;
      }
    }
    int other_side = !it->skip_side;
    int *occs = get_occs_in_coarse(fram, it, other_side);
    found_match = find_best_match(fram, it, it->skip_side);
  }
  if (found_match){
    it->num_bilingual_matches++;
  }
  return found_match;
}
```

Listing 4.12: Skip-based bilingual search procedure

and the number of individual matches, in each language (*num_matches*).

This procedure starts by getting two new individual occurrences from each language, with the function *next_monolingual_occurrence*. This function determines an occurrence in a language, the fine and coarse segments where it appears, and the coarse segment bounds, in terms

of absolute offsets. This function will be detailed further ahead in this Section.

Having the two occurrences in each language calculated, *Condition 1* is verified, by comparing their two coarse segments. While *Condition 1* is not matched, the procedure determines the side that appears behind, namely the one with the smaller coarse segment. After knowing which occurrence is ahead, $coarse\_ahead$, we determine on the side that is behind, the offset bounds for the coarse segment aligned with $coarse\_ahead$. This falls on the description of this skip-based procedure, since we know that no co-occurrence will be matched before $coarse\_ahead$, in order for both individual occurrences to meet *Condition 1*. Then, the maximum between the beginning offset of *coarse_ahead* and the already determined *next_pos* from that side is chosen to be the position of the skip. With that, the new occurrence on that side is calculated. The process is repeated until *Condition 1* is met, or there are no more occurrences.

When *Condition 1* is met, all the occurrences of a term, in one language l_1 and within that coarse segment of the alignment, are determined and stored in a list. This step is executed with the help of the *get_occs_in_coarse* function. Then, for every occurrence on the other language, l_2, also within the aligned coarse segment, the procedure determines the best match, considering the limit *window* and *Condition 2*. This step is performed with *find_best_match* function, which iterates over the occurrences in l_2, that appear within the coarse segment being processed, and for every occurrence checks *Condition 2* with the occurrences in l_1. If more than one pair of occurrences matches *Condition 2*, the function returns as co-occurrence the ones that are closer, regarding the *window*.

Algorithm 4.13 shows the *next_monolingual_occurrence* procedure, heavily used in the skip-based bilingual search. This function calculates an individual occurrence of a term in one language, alongside the $fine$ and $coarse$ segments. The first step of this function determines the next occurrence of the term in the given *side*, through the *locate_next* function of the Text Layer, and using the *next_pos* field as the already calculated position of the skip. If an occurrence is found, then the *locate* function of the Alignment Layer is used to get the alignment segments where that occurrence appears. This function, despite simple, is critical for the skip-based bilingual search, because it does not only determine the next occurrences of the terms and where in the alignment they appear, but also contributes to the calculation of the next skips.

```
int next_monolingual_occurrence(struct bil_fram *fr, struct results_iter *it,
    int side){
  int off = fr->tapi[side]->locate_next(fr->ti[side], it->its[side], &(it->
      next_pos[side]));
  if (off != -1){
    it->offs[side] = off;
    it->num_matches[side]++;
    fr->aapi->locate(fr->ai, off, &(it->coarse[side]), &(it->fine[side]));
    return 1;
  }
  return 0;
}
```

Listing 4.13: Function to determine the next occurrence in one language

Applying the skip-based bilingual search procedure to any other data structure, would most certainly require changes on its own *locate* procedures. Those changes, as seen here with Word-based Suffix Arrays and Byte-codes Wavelet Trees, depend heavily on the data structure itself. For the latter, since the index is represented following the order of the text, and it is self-synchronized, the skips are applied directly to the index, as opposite to the lexicographic order of the Suffix Arrays.

However, as we demonstrate in the next Section, regardless of the choice, the bilingual search time requirements reduce considerably with this approach, given the number of occurrences that are not processed, to obtain the exact same number of co-occurrences, when compared with the original bilingual search algorithm. On top of that, this skip-based bilingual search procedure does not demand extra space consumption, as it only uses the information reported by the Text and Alignment Layers.

4.4 Results

After analyzing the space consumption of the Bilingual Framework, this Section focus on its search time response efficiency, considering the several approaches presented for the Text and Alignment Layer. The experiments consist of measuring the phrase-based monolingual and bilingual search time response, using Word-based Suffix Arrays and Byte-codes Wavelet Trees, for the Text Layer, and Bitmaps and Arrays of Integers, for the Alignment Layer. The measures were made for both bilingual search approaches, the first one based on binary searches and the second one based on the skips. Both algorithms require the same space requirements, described in Chapter 3, thus the memory consumption is not measured again.

To measure the search time response of the Bilingual Framework, 700000 translation pairs in EN-PT were automatically extracted [69, 72], and classified as correct, using the aligned parallel corpora presented in Section 3.6 as source texts. For the DE-PT language pair, 274747 translation pairs were extracted using the same methods. These entries were classified and validated as correct by human translators, under the two-year research project ISTRION[3], funded by FCT/MEC. Each one of these entries consists of two terms that are correct translations of each other, with variable length of one to eight words / tokens. After being classified as correct, they were later added to the EN-PT and DE-PT bilingual lexica used for the experiments. The experiments were developed using a machine with 16 Gigabytes Memory RAM, a Intel Core I7, 3.4 GHz processor, with Ubuntu Linux Kernel 3.2.0.

4.4.1 Monolingual Search Results

Table 4.1 presents the monolingual search time results for the Bilingual Framework, considering the four possible combinations, using the two approaches for the Text Layer and for the Alignment Layer. For these results, the 700000 entries in English and Portuguese were used. Additionally, the Table shows the number of occurrences returned for each corpora, per language,

[3] ref. PTDC/EIA-EIA/114521/2009

after the processing was done, in thousands, and the impact ratio on using the compressed indexes against the Suffix Arrays approach.

Table 4.1: Monolingual search time results, in seconds, and the number of occurrences found in thousands, for English and Portuguese

Corpora	**Nr. Occurrences**		**Time WSA**		**Time WT**		**Ratios**	
	EN	**PT**	**EN**	**PT**	**EN**	**PT**	**EN**	**PT**
EUconst	1121.2	1092.5	3	2	2	2	0.67x	1x
EMEA	81289.8	85482.4	11	13	201	174	18.3x	13.4x
Europarl	237974.4	230621.4	59	42	1574	1443	26.7x	34.4x
DGT	456850.9	476641.8	73	102	8401	6178	115.08x	60.57x
Eurlex	594720.5	642567.8	114	127	14246	12150	124.96x	95.67x

By analyzing the monolingual results, the WSA shows a much faster time response than WTs, as expected, with the difference in time being aggravated with the increasing of the size of the corpora and the number of occurrences processed. However, it is important to say that this comparison is unfair, since one index is compressed, while the other is not. Adding to that, we need to understand that these results are the accumulation after all the search pairs were processed, which causes even more impact.

The results for the Word-based Suffix Array show that the increasing of the number of occurrences returned has an influence on the processing time. With Europarl, there are more occurrences on the English side than on the Portuguese side, thus the monolingual search takes more time in English. On the other hand, for Eurlex, the situation changes, and the Portuguese side becomes slower since it has more occurrences to report. That results from the monolingual search being dependent on processing the list of occurrences, within a calculated interval of the Suffix Array, thus the more are the occurrences, the more time the search consumes.

With Byte-codes Wavelet Tree the behavior is not the same. Analyzing the result for DGT and Eurlex, and even considering that the Portuguese side has more occurrences than the English side, the search takes more time for English. The cause for that to happen is related with the ambiguity when translating a word from English to Portuguese. There are expressions in English that can be translated differently in Portuguese, and vice-versa, thus those terms appear in different times in the lexica, with a different translation in Portuguese. Thus, there are cases where the same term in English is processed several times, since it is part of different pairs. However, since there are more occurrences in Portuguese, it means that we have highly frequent terms, namely single word terms, resulting in more occurrences, but spanned throughout a smaller number of terms, when compared with the English side. This way, it is faster to process a term with a lot of occurrences, for instance 4000, than ten terms with 400 occurrences each, resulting in the faster time response for the Portuguese side, in Byte-codes Wavelet Trees.

At this point, the Alignment Layer does not have any influence on the monolingual search time results, since the alignment is not required for returning the occurrences of a term in their language. The Alignment Layer has a deep influence in the bilingual search time response, as

demonstrated next.

4.4.2 Bilingual Search Results: First Approach

Tables 4.2 and 4.3 present the bilingual search time results for the Framework, considering the first approach based on binary searches. These bilingual searches focused on *counting* the number of co-occurrences of each pair, and were supported by the four different alternatives of the Bilingual Framework, with the two implementations of the Text Layer and the two of the Alignment Layer. To obtain these results, we removed from the 700000 EN-PT pairs, the ones that do not have any occurrence on the respective corpora being processed. This means that any translation pair, where both terms do not have any occurrence from the corpora, are removed from the list. This way, Table 4.2 also shows the number of pairs processed and the number of co-occurrences (**Co-occs**) found, in each corpora, in thousands. For DE-PT, since the number of pairs is smaller, we did not apply this filter, considering always all the pairs in the lexicon. Table 4.3 shows the number of co-occurrences found, and the time spent to obtain those co-occurrences, but not the number of searches made, since it is always 274747 pairs. Both tables show the slowdown ratios from using Bitmaps, instead of Arrays of Integers, for supporting the Alignment Layer.

Table 4.2: Bilingual search time results, in EN-PT

Corpora	**Nr. Searches**	**Co-occs**	**Time WSA**			**Time WT**		
			Bmap	**ArrInt**	**Ratio**	**Bmap**	**ArrInt**	**Ratio**
EUconst	132.072	201.0	3	2	1.5x	5	3	1.67x
EMEA	190.600	10946.5	271	62	4.37x	635	425	1.49x
Europarl	455.122	23513.7	940	217	4.33x	3944	3196	1.23x
DGT	502.769	58762.5	1967	444	4.43x	16484	14945	1.10x
Eurlex	553.279	79388.7	2684	591	4.54x	29373	27239	1.08x

Table 4.3: Bilingual search time results, in DE-PT

Corpora	**Co-occs**	**Time WSA**			**Time WT**		
		Bmap	**ArrInt**	**Ratio**	**Bmap**	**ArrInt**	**Ratio**
EMEA	6133.261	169	59	2.86x	395	279	1.42x
Europarl	16475.135	450	112	4.02x	2703	1756	1.54x
DGT	28207.182	839	231	3.63x	4892	4289	1.14x

Analyzing the bilingual search time results, we can conclude that the solution supported by WSA + ArrInt is the fastest, while the slowest is the solution based on WT + Bmaps. These time response values are expected, since the WT + Bmaps approach use the two most compressed alternatives for the Text and Alignment Layer respectively. Since these compressed data structures represent the data in an indirect way, to save space, extracting information from them requires

more time. On the other hand, the approach based on WSA + ArrInt uses the two non compact approaches for the Text and Alignment Layers, thus having fastest search time responses.

Specifying the approaches for the Alignment Layer, the Arrays of Integers are more time efficient than the Bitmaps. Considering the results for Eurlex, in EN-PT, when the Text Layer is implemented by WSAs, the approach with Arrays of Integers is 4.54x faster. For DE-PT, this efficiency can also be verified in DGT, with a 3.6x faster performance with the WSA + ArrInt combination. However, looking at the results for the same corpora, but using WTs for the Text Layer, the difference is much lower. For EN-PT, in Eurlex, the difference is around 1.4x, while for DE-PT, with DGT, the difference is 1.14x. This proves that with WT, the main weight of the search lies on the text index, rather than in the Alignment Layer. That is actually visible by checking that the differences in time are similar to both Text Layer approaches. For instance, in DE-PT, with WSA, the approach based on Bitmaps is around 600 seconds slower (839-231), just like with WTs (4892-4289). However, as the time consumption is considerably larger with the compressed indexes, the ratios demonstrating the difference between the Alignment Layer representation are smaller, as the Text Layer time response has much more impact on the overall results.

The reason for the difference between the Alignment representations, lies on the more compressed nature of the Bitmaps. To extract information about the coarse and fine segments where an occurrence appears, with Bitmaps it is necessary to query with the bitmaps with $rank$ and $select$ operations, which is more time consuming than performing binary searches directly over the Suffix Arrays.

4.4.3 Bilingual Search Results: Skip-based Procedure

Tables 4.4 and 4.5 presents the bilingual search time results using the skip-based approach, for the Framework with the Alignment Layer supported by Arrays of Integers. Furthermore, they present the improvement ratios (**Imp. Ratio**) of the search time response, in comparison with the results presented in Tables 4.2 and 4.3. Similarly, Tables 4.6 and 4.7 show the same results, using the Alignment Layer supported by Bitmaps. Tables 4.4 and 4.6 also show the total number of occurrences of the terms that were processed, for the same number of searched pairs, to return the exact same number of co-occurrences as displayed in Tables 4.2 and 4.3. Tables 4.6 and 4.7 do not show these values, as they are the same. The skip-based bilingual search results are again presented in seconds and the number of occurrences in thousands.

By analyzing the results from the four Tables, it is noticeable that regardless the solution used to support the Bilingual Framework, and the language pair, using the skip-based approach improves considerably its bilingual search time response. This improvement is more visible for larger corpora and also for highly repetitive texts, like Europarl and Eurlex, and non existent for small corpora such as EUconst. For larger or more repetitive corpora, the cases of ambiguity and the number of occurrences of the terms tend to be higher, thus benefiting more the behavior of the skips, resulting in more significant time savings.

The improvement ratios displayed in the Tables clearly demonstrate how the skip-based bilingual search is always more efficient. Considering all the gathered results in EN-PT, the

Table 4.4: Skip-based search time results in seconds and number of occurrences in thousands, with the Alignment Layer based on Arrays, for EN-PT

Corpora	Nr. Occurrences		Time WSA			Time WT		
	EN	PT	Skip	Base	Imp. Ratio	Skip	Base	Imp. Ratio
EUconst	704.5	717.3	2	2	1x	3	3	1x
EMEA	33321.7	39184.0	34	62	1.82x	274	425	1.55x
Europarl	69587.1	75332.9	99	217	2.19x	1943	3196	1.64x
DGT	132491.3	144504.2	192	444	2.31x	8907	14945	1.68x
Eurlex	179240.8	197528.2	256	591	2.31x	16471	27239	1.66x

Table 4.5: Skip-based search time results in seconds and number of occurrences in thousands, with the Alignment Layer based on Arrays, for DE-PT

Corpora	Nr. Occurrences		Time WSA			Time WT		
	DE	PT	Skip	Base	Imp. Ratio	Skip	Base	Imp. Ratio
EMEA	14577.554	18761.867	39	59	1.51x	214	279	1.3x
Europarl	41072.747	48910.233	60	112	1.87x	1152	1756	1.52x
DGT	61084.590	76655.437	135	231	1.71x	2614	4289	1.64x

Table 4.6: Skip-based search time results in seconds, with the Alignment Layer based on Bitmaps, for EN-PT

Corpora	Time WSA			Time WT		
	Skip	Base	Imp. Ratio	Skip	Base	Imp. Ratio
EUconst	3	3	1x	4	4	1x
EMEA	147	271	1.44x	390	635	1.63x
Europarl	366	940	2.57x	2220	3944	1.78x
DGT	706	1967	2.79x	9309	16484	1.78x
Eurlex	955	2684	2.81x	16689	29373	1.76x

Table 4.7: Skip-based search time results in seconds, with the Alignment Layer based on Bitmaps, for DE-PT

Corpora	Time WSA			Time WT		
	Skip	Base	Imp. Ratio	Skip	Base	Imp. Ratio
EMEA	78	169	2.17x	345	395	1.15x
Europarl	174	450	2.59x	1695	2703	1.59x
DGT	301	839	2.79x	3422	4892	1.43x

improvement ratio is larger for the approach based on WSAs, when compared with WTs, regardless of the chosen Alignment Layer implementation. Using Arrays of Integers, the improvement

ratios show around 2.3x faster bilingual search response time with WSAs and around 1.66x with WTs, for the Eurlex corpora in EN-PT. In DE-PT, for DGT, the improvements ratios are of 1.71x and 1.64x respectively for WSAs and WTs, showing slightly smaller ratios due to the less number of pairs searched, thus less ambiguity. These results may seem surprising, since the skip-based approach is more directly applied to the Byte-codes Wavelet Tree, while with in Word-based Suffix Arrays the skips are only applied to the final sets of occurrences. However, with WTs every skip requires extra *descends* and *ascends* in the tree structure, to decode the position of the skip and understand which occurrence of the term is matched by that position. Thus, even considering the number of occurrences that we avoided processing, these extra operations also need to be accounted for and have a big influence on the general performance of the algorithm. Nevertheless, the improvements in search time response are significant.

Considering just the results with the Alignment Layer supported by Bitmaps, presented in Tables 4.6 and 4.7, one may see that the improvements are even bigger with this approach, achieving 2.81x faster results for WSAs and 1.78x for WTs, considering again the Eurlex corpora, in EN-PT. For the DE-PT pair, the results follow the same logic than the ones for EN-PT, with improvement ratios around 2.79x with WSAs and 1.43x with WTs, for the corpora DGT.

The cause for these improvements, when compared with the approach based on Arrays of Integers, lies on the heavier nature of the Bitmaps. As with the skip-based search procedure, less occurrences are processed, thus less queries to the Alignment Layer are made, the bigger is the impact on the time response. The results with WTs do not display that behavior so clearly, due to the heavier nature of the text index, as explained before, which consume most of the search response time. However, there are exceptions. With the pair DE-PT, it is noticeable that the approaches based on Arrays of Integers present a better improvement ratio, when using Byte-codes Wavelet Trees. This shows that when using the compressed version of the Text Layer, only in cases where the number of occurrences saved is considerably high, just like in EN-PT, the improvements with Bitmaps are higher than with Arrays of Integers. Otherwise, it is the text index time consumption that has the major influence on the search time response, and thus the improvements can actually be smaller with Bitmaps, as it is a slower alignment representation than Arrays of Integers.

Regardless of the variation of the improvement ratio in all the experiments made, the skip-based bilingual search always speeds up the bilingual search in the proposed Framework. The main reason behind these improvements can be seen on the number of individual occurrences processed, by both bilingual search approaches. Tables 4.8 and 4.9 show those numbers in more detail, displaying the number of individual occurrences non relevant to the bilingual search, in thousands, and thus skipped by the proposed skip-based search procedure. These numbers are followed by the reduction ratio of occurrences processed from one method to the other.

In EN-PT, for Eurlex, DGT and Europarl, the number of occurrences processed reduces around 70%, for both languages, to find the same number of co-occurrences. Even for smaller corpora, such as EUconst, the number of processed occurrences is reduced by 35-37%. In DE-PT, the reduction is more visible in Portuguese, with ratios from 63% to 68%. However, in German, these ratios do not go over 55%, which are the lowest ones from the three languages tested.

Table 4.8: Number of occurrences processed with the skip-based search time results, in thousands, for EN-PT

Corpora	Nr. Occurrences (Bisect)		Nr. Occurrences (Skips)		Reduction Ratio	
	EN	PT	EN	PT	EN	PT
EUconst	1121.2	1092.5	704.5	717.3	37%	35%
EMEA	81289.8	85482.4	33321.7	39184.0	59%	54%
Europarl	237974.4	230621.4	69587.1	75332.9	71%	67%
DGT	456850.9	476641.8	132491.3	144504.2	71%	70%
Eurlex	594720.5	642567.8	179240.8	197528.2	70%	69%

Table 4.9: Number of occurrences processed with the skip-based search time results, in thousands, for DE-PT

Corpora	Nr. Occurrences (Bisect)		Nr. Occurrences (Skips)		Reduction Ratio	
	DE	PT	DE	PT	EN	PT
EMEA	30124.349	50780.848	14577.554	18761.867	52%	63%
Europarl	85929.709	135269.655	41072.747	48910.233	52%	64%
DGT	137121.001	237766.090	61084.590	76655.437	55%	68%

This shows that the level of ambiguity in such texts is very high, with terms having more than one possible translation in the same corpora, making some of their occurrences non relevant to some bilingual searches. For DE-PT, the ambiguity is lower, which alongside having a lexicon in DE-PT not as advanced as the EN-PT lexicon, explains that the number of skipped occurrences is smaller. Despite the reduction ratios displayed in DE-PT do not follow as much the ones obtained with the EN-PT language pair, it is visible how the skip-based bilingual search procedure can save unnecessary processing time.

For a deeper understanding of the skip-based bilingual search performance, Figures 4.14 and 4.15 show the evolution of the search time response, for the 10000, 30000, 50000, 70000 and 100000 most frequent EN-PT pairs of terms, in Europarl, DGT and Eurlex. These most frequent pairs were extracted per corpora, to make sure that the tests were made using the terms with more occurrences on the specific corpora. This means that the 10000 most frequent pairs in Europarl, are probably different from the 10000 most frequent pairs in Eurlex.

The values presented in the Figures show the weight of the most frequent pairs of terms in the overall bilingual search time performance of the Framework. For the approaches supported by WSA, it is clear how much the most frequent terms impact the overall performance of the Framework. With Eurlex, and using the Arrays of Integers in the Alignment Layer, the 10000 most frequent entries take around 131 seconds to be processed, which is 51% of the global time spent to calculate all the 553279 entries co-occurrences, 256 seconds. Moving on to the 100000 most common entries, the same combination WSA + ArrInt takes 194 seconds to determine the co-occurrences of those entries, which is 76% of the total time (256 seconds) for all entries.

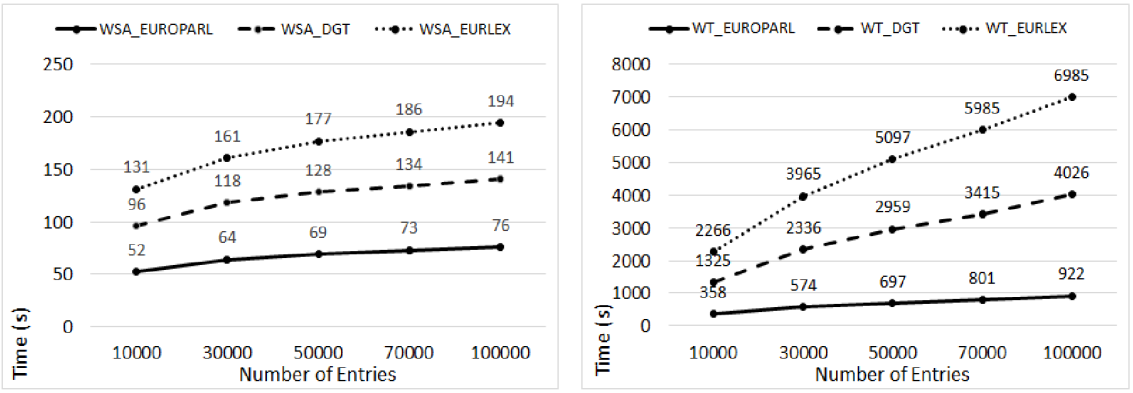

Figure 4.14: Bilingual search time results for the most frequent entries on Europarl, DGT and Eurlex, using WSA and WT with Arrays of Integers to represent the alignment

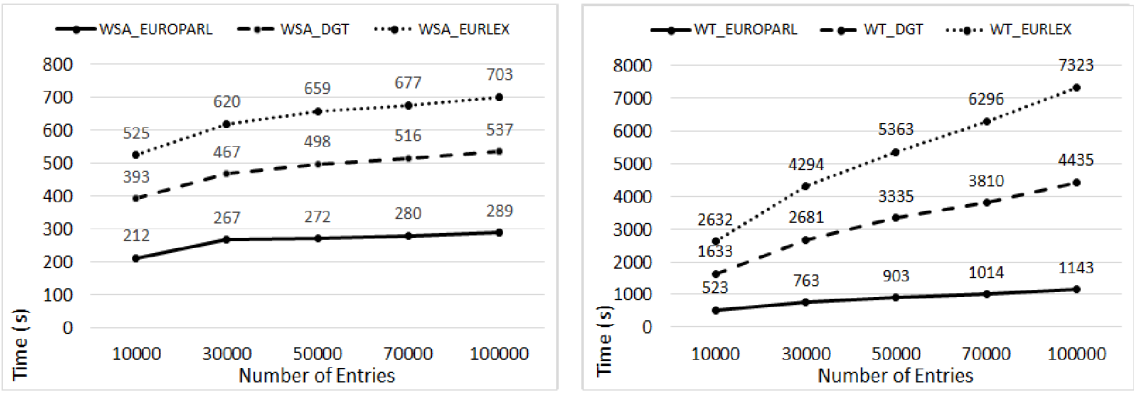

Figure 4.15: Bilingual search time results for the most frequent entries on Europarl, DGT and Eurlex, using WSA and WT with Bitmaps to represent the alignment

With Bitmaps, the same behavior is also visible, even considering other corpora. With DGT, the 10000 most frequent entries take 393 seconds, which is 55% of the total time to process all the bilingual entries, 706 seconds. With the 100000 most common entries, the Framework takes 537 seconds to process them, making it 76% of the total time (706 seconds).

The previous percentages show that the most frequent entries have a huge weight on the approaches based on Word-based Suffix Arrays, since the list of occurrences returned by the indexes will be bigger. For determining the interval of occurrences in the Suffix Arrays this may not be that relevant, but for processing the list of occurrences it is, since much less occurrences will be analyzed to match *Condition 1* and *2*.

With Byte-codes Wavelet Trees the situation is slightly different. These percentages show that the most frequent entries, despite also taking a considerable amount of the search response time, do not have the same weight as with WSAs. For instance, consider the 10000 most frequent entries for DGT. With Arrays of Integers supporting the alignment, those entries are processed in 1325 seconds, which is 15% of the total search time response for all occurrences, 8907 seconds. With Bitmaps, the ratio is around 17%, for the same corpora, with a response of 1633 seconds, out of the total 9309 seconds. Considering that 1000 entries are around 2% of the total

entries processed for Eurlex, it is visible that these most frequent entries have an impact on the overall performance. However, comparing it with the impact in Word-based Suffix Arrays, it is considerably smaller. For the same corpora, processing the 100000 most frequent entries takes around 45% of the total time, 4026 seconds against the total 8907 seconds, with Arrays of Integers, and 47% with Bitmaps, 4435 seconds from the total 9309 seconds.

The justification for the difference in impact verified between the data structures, lies again on their structure and behavior. Since searching for a term in a Byte-codes Wavelet Tree is by itself considerably heavier than with Suffix Arrays, even the terms that are not as frequent have a big impact on the total search time. This is the reason why the 1000 most frequent entries take more than 50% of the total time in WSAs, but only 15% with WTs.

Considering all the results presented in this Section, using WTs to support the Framework, makes it considerably slower, when compared with WSA. The compressed structure of WTs, with all the words codified to save space, demands more time to find and decode a term in the indexed corpora. Even taking into account the improvements of the skip-based bilingual search, they are smaller when compared with the approach based on Word-based Suffix Arrays. Considering that with Word-based Suffix Arrays, the skips are applied to the sets of occurrences, while with Byte-codes Wavelet Trees the occurrences are actually skipped on the tree structure, these results could be surprising. However, they actually show how time consuming it is to restart the search from a given point, in the tree structure of the compressed index.

Nevertheless, this slowdown is typical and already expected with compressed indices, and it depends on the level of compression rate. Data structures with higher compression ratios, and thus less space requirements, will have slower search time responses, since the textual data is deeply codified in the index. The same scenario is visible with the implementations of the Alignment Layer, where the approach based on Bitmaps requires more time than the one based on Arrays. However, considering the index compression ratio, this solution is important to index huge corpora in main memory, since it can easily avoid the slower access to secondary storage.

4.5 Bilingual Concordancer Application

For testing the Bilingual Framework functionalities in a real world application, we applied the Bilingual Framework, with compressed data structures and the skip-based bilingual search, to a bilingual concordancer application.

Bilingual concordancing is a context analysis tool [18] that works over large aligned parallel corpora, and answers user queries in real time. A concordancer tool allows an user to search for a single or pair of word and multi-word terms, in one or in both languages of the texts indexed by the Framework. These searches provide the user the location and frequency of occurrences, or co-occurrences, of the searched terms, alongside the context surrounding the occurrences. This context is determined by the data structures that support the Text Layer, whenever a valid occurrence or co-occurrence is found, for one or the two languages.

Figure 4.16 shows an example of a bilingual concordancer search for the pair of terms

`reserve assets`<->`activos de reserva`, in the tool supported by the Bilingual Framework. The Figure shows three co-occurrences of the pair highlighted, alongside the context in the sentences surrounding each occurrence.

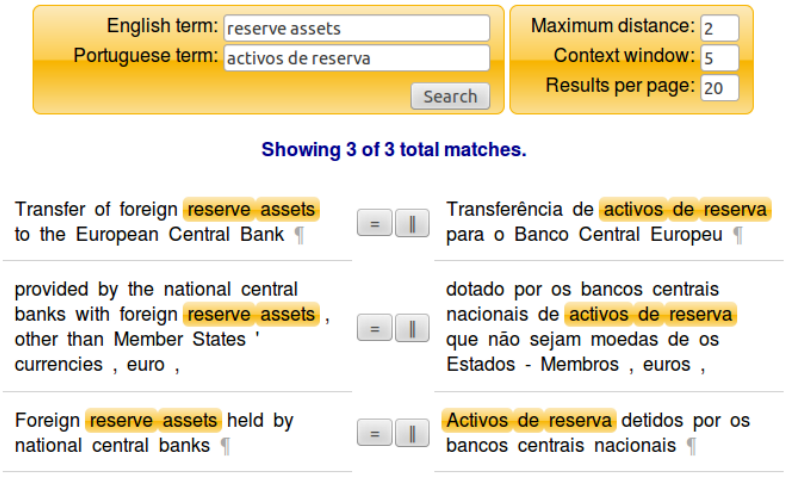

Figure 4.16: Example of a bilingual concordancer search, in English and Portuguese

A tool such as the bilingual concordancer, is often used by human translators, as it is a powerful mechanism to guarantee that a translation is indeed correct. As mentioned earlier in this thesis, the highest the co-occurrence frequency between two terms in two different languages, the highest is the probability of those two terms being correct translations of each other. Besides that, the surrounding text that is extracted provides important information for determining if a translation is valid or not, since it provides contextual information to the user. For that matter, a concordancer tool is a good auxiliary tool for checking, or discovering, a correct translation of a given term.

Since a concordancer tool is mainly designed for human users, we need to consider that they do not like to wait long for obtaining the answers to the searches made. According to [143], response times up to 0.1 seconds are perceived as instantaneous by users, which is something that such a tool should provide. Besides this time requirements, by using potentially huge amounts of parallel corpora, the space problem is as well presents, which makes the space / time trade-off of extreme importance for such a tool.

Considering the results displayed so far by the Bilingual Framework, not only in terms of space consumption, but also regarding the monolingual and bilingual search time performance, a bilingual concordancer tool demands fit well to be supported by the Framework.

4.5.1 Extracting the snippets

To provide interesting context to the user, extracting a snippet of a text where a term occurs does not focus just on the occurrence itself, but also on the text surrounding it. For that, as shown in Figure 4.16, there is a field to input a context window value, alongside the maximum distance allowed for *Condition 2*, for two occurrences to co-occur.

After an occurrence, or co-occurrence is determined, if the concordancer tool is running, then the pieces of text, or snippets, are extracted and displayed to the user. For this tool, we considered that the snippet begins at the fine segment where the occurrence appears, minus the window chosen for the context, also measured in fine segments. The snippet then ends at the fine segment where the occurrence ends, plus the window. These values are calculated using the *locate*, *fine_begin* and *fine_end* of the Alignment Layer. Then, with the bounds calculated, the *get_snippet* function of the Text Layer is called to get the text to be displayed to the user. Algorithm 4.14 shows the Bilingual Framework procedure for extracting a snippet.

```
char* extract_snippet(struct bil_fram *fr, int off, int len, int fine, int
    side, int window, int *len_bytes){
  struct s_bounds sbounds;
  sbounds.begin_fine = fine - window;
  int final_off = off + len;
  int final_fine, final_coarse;
  fr->aapi->locate(fr->ai, final_off, side, &final_coarse, &final_fine);
  sbounds.end_fine = final_fine + window;
  fr->aapi->fine_begin(fr->ai, sbounds.begin_fine, sbounds.begin_offs);
  fr->aapi->fine_end(fr->ai, sbounds.end_fine, sbounds.end_offs);
  int slen = sbounds.end_offs[side] - sbounds.begin_offs[side];
  return fr->tapi[side]->get_snippet(fr->ti[side], sbounds.begin_offs[side],
      slen, &len_bytes);
}
```

Listing 4.14: Extract snippet procedure

The *get_snippet* function of the Text Layer extracts the snippet of text from the indexes. It is important to recover that word-based data structures have a special representation of the text, to allow the word as main unit, instead of the character. For that matter, the same way the text was converted to be indexed by the structures, for this function, they need to be converted back to the original text.

For the Word-based Suffix Arrays, the integer *ids* need to be converted back to words, by using the vocabulary used for building the structure. With Byte-codes Wavelet Trees, the same logic is followed, but here for the codewords of bytes, requiring also access to a vocabulary to get back the original words. This process involves complex procedures around the vocabulary representation, in particular for the compressed index, thus it will not be detailed in this thesis.

An additional scenario that needs to be taken into account, is that all the corpora indexed by the Framework is in lower case. However, when using the *get_snippet* function, the objective is to fetch the snippet with its original casing. For that purpose, the Case Insensitive Layer needs to convert the words in the snippet to capital or upper case, when applied, using the bitmaps created while building the indexes.

Algorithm 4.15 shows the generic procedure of the *get_snippet* function of the Case Insensitive Layer. It uses as well the nomenclature of the Text Layer API and the structure **ti_index**. The procedure extracts the desired snippet of text from the Text Layer, and then, for every word in that snippet, recover its original casing.

All these functionalities described during this Chapter are based on word and multi-word

```
char* get_snippet(struct t_index *index, int off, int len, int *len_bytes){
  char *lc_snippet = index->text_layer_api->get_snippet(index->
      text_layer_index, off, len, len_bytes);
  char *snippet = init_snippet(*len_bytes);
  int first = 1;
  For(every token in lc_snippet){
    if (first){
      first = 0;
    }else{
      append(snippet, "␣");
    }
    if (get_bit(index->lower_case, off) == 1){
      append(snippet, token);
    }else if (get_bit(index->upper_case, off) == 1){
      append(snippet, to_upper(token));
    }else{
      append(snippet, to_capitalizd(token));
    }
  }
  return snippet;
}
```

Listing 4.15: *get_snippet* function of the Case Insensitive Layer

phrase terms. In the next Chapter, I introduce an extension of the Bilingual Framework to support hierarchical phrase terms, in monolingual and bilingual searches.

CHAPTER 5

HIERARCHICAL PHRASE-BASED SEARCH

Hierarchical Phrase-based Translation improved Machine Translation quality by using phrases containing gaps in translation rules, instead of word and multi-word contiguous phrases. In this thesis, these phrases with gaps are denominated as hierarchical phrases. These improvements result from the structure of these hierarchical phrases, that by using variable gaps between contiguous subphrases, allow the translation engines to learn reorderings that would not be possible with contiguous phrases. When translating from one language to the other, these changes of order often occur and they can be extreme, if the languages are considerably different.

Figure 5.1 shows several examples of pairs of hierarchical phrases, that are correct translations of each other in English and Portuguese, alongside an example of a correct match of the phrases. In the first case, the phrase has two gaps that are replaced by `Portugal` and `Spain`, in English, and `Portugal` and `Espanha`, in Portuguese. The other cases show scenarios where there are changes of order when translating from English to Portuguese, such as `global market value` being translated by `valor global de mercado`. In this example, the gap indicates that whatever textual content appears after `global market` may be translated by the content that appears before `global de mercado`, in Portuguese. The third and final case, marks a more difficult construction, where in Portuguese `completou` is translation of `has completed`, but the existence of an adverb, like `already`, creates a different order on the word placement.

Taking the benefits of hierarchical phrases to Machine Translation, we extended the Bilingual Framework to support its *locate*, *count* and *extract* functionalities for terms with gaps.

5.1 Hierarchical Phrases in the Bilingual Framework

In this thesis, a generic hierarchical phrase term contains a set of n contiguous subphrases, denominated as literals, and $n+1$ gaps following the structure $\$_1\ L_1\ \$_2\ L_2\ ...\ \$_n\ L_n\$_{n+1}$, with \$

Hierarchical Phrase	Matching Example
In $1 and $2 alike <-> Tanto em $1 como em $2	In Portugal and Spain alike <-> Tanto em Portugal como em Espanha
global market $1 <-> $1 global de mercado	global market value <-> valor global de mercado
has $1 completed <-> $1 completou	has already completed <-> já completou

Figure 5.1: Three examples of hierarchical phrase pairs in English and Portuguese

representing a gap and $n \leq 5$. Each gap can have different lengths and variable content, within the same term, and can possibly be empty, meaning the existence of a gap with no content.

Considering this variable nature of the gaps, in terms of content and length, and the structure of the hierarchical phrases, it is important to define some rules for the matching them in the corpora. Otherwise, we could get occurrences of these hierarchical phrase terms of excessively large lengths, with the gaps having statistically irrelevant content.

Taking that into account, for this work, we defined a first constraint that states that no term with gaps can be matched over a fixed number of fine segments of the alignment, meaning that, at most, no content of a gap can span for more than that number of fine segments. This restriction is applied regardless of the position of the literals and gaps within a hierarchical phrase. This number is a parameter that can vary considering the requirements of the application.

To filter even more the occurrences returned on a search for a hierarchical phrase term, an additional constraint states that no match for a hierarchical phrase can exceed the boundaries of a coarse segment, or sentence, of the alignment. This means that a match cannot start in one coarse segment and end in another coarse segment.

Figure 5.2 recovers the fine alignment example introduced earlier in this thesis. Considering the example `put $ by`, on the fine alignment on the right, it has an occurrence at position nine of the English side, matching the text `put to it by`. Here, the gap content just takes one fine segment, and the occurrence appears within the same coarse segment of the alignment, thus being a valid individual occurrence of the term. On the other hand, the other possible match of the term, `put to it by the European Parliament or by`, starts at the same offset, but has an excessive length, where the gap spans over five fine segments.

Matching hierarchical phrase terms in the parallel corpora is not the same as doing so for contiguous phrase terms. The nature of these terms requires some changes in the Bilingual Framework, so that it can answer to searches based on hierarchical phrase terms, alongside the phrase-based search.

2nd coarse segment

English	Alignment	Portuguese
The	A	A
Commission	A	Comissão
may	A	pode
attend	A	assistir a
all	A	todas
the	A	as
meetings	A	sessões
of	A	de
the	A	o
European Parliament	A	Parlamento Europeu
and	A	e
shall , at its	?	é ouvida quando assim o
request	A	solicitar
, be heard	?	
.	P	.

3rd coarse segment

English	Alignment	Portuguese
	?	A Comissão
It shall reply	A	responde
	?	,
orally	A	oralmente
or	A	ou
in writing	A	por escrito
	?	,
to questions	A	a as questões
	?	que lhe sejam
put	A	colocadas
to it	?	
by	A	por
the	A	o
European Parliament	A	Parlamento Europeu
or	A	ou
by	A	por
its	A	os seus
members	A	membros
.	P	.

Figure 5.2: English - Portuguese fine alignment example

Regarding the bilingual search, there is no need to adapt the skip-based algorithm, as *Condition 1* and *Condition 2* can still be used to determine co-occurrences and the skips can also be used to save time. The main change lies on the monolingual search of the Bilingual Framework, thus over the Text Layer. This way, the data structures supporting this Layer need to be adapted to support determining occurrences of terms with gaps of variable content.

5.2 Text Layer: Matching Hierarchical Phrases

With the skip-based bilingual search, the Framework adapted an iterative approach, where every occurrence of the searched term is returned at a time. This behavior led to some significant changes over the Text Layer, that are still valid for the hierarchical phrases. In fact, the iterative behavior does not change, since the searches are still made with one term, or a pair of terms.

However, with the introduction of gapped terms, there are some restrictions that need to be followed and were presented earlier in this Chapter, namely the maximum length of the gaps and the occurrence being fully contained within a coarse segment. For that matter, when returning the next occurrence of a term in the iteration, it is important to enforce those restrictions, since the gaps can vary and it is too optimistic to assume that those restrictions are not violated, without explicitly checking them.

Figure 5.3 shows the final version of the Text Layer API, with some slight changes to consider hierarchical phrase terms in the search. To apply the matching restrictions to the next occurrence returned by the search, the *locate_next* function adds an interval to its parameters. This interval is defined by the structure **t_interval**, which has a *start* and an *end* offset. This

way, every time the algorithm calls for the next occurrence in a particular language, this interval shows the boundaries of where it should appear in the text. Considering an occurrence of a hierarchical phrase term at offset off of the text, with length len, this means that a valid occurrence of the term must meet the following Condition:

$$interval.start \leq off \ \& \ off + len \leq interval.end$$

where $interval.start$ and $interval.end$ indicate the parameter interval.

Additionally, there is another structure, **t_occ**, which represents a new occurrence of a phrase, with two fields: the offset and the length of the occurrence. This structure was added because with hierarchical phrases, the length of the occurrences, for the same term, may easily vary. This is applied directly to the output of the *locate_next* function.

Finally, at the *start_locate* function, there are two additional values returned from the function, the *has_start* and *has_end*. These two values indicate if the term has a gap at the beginning and at the end.

```
struct t_index;
struct t_occ_iter;

struct t_occ{
    int off;
    int len;
}

struct t_interval{
    int start;
    int end;
}

struct ti_api{
    void create_index (char* outfile, char*
        text, int text_len, struct ti_api *api);

    struct t_index* load_index(char* infile,
        struct ti_api *api);

    void free_index(struct t_index* index);

    struct t_occ_iter* start_locate (struct
        t_index *index, char* term, int *start,
        int end, int *has_start, int *has_end);

    struct t_occ* locate_next(struct t_index*
        index, struct t_occ_iter* it, struct
        t_interval interval, int *skip);

    void end_locate(struct t_index* index,
        struct t_occ_iter* it);

    char* get_snippet (struct t_index *index,
        int off, int len, int* len_bytes);
}
```

Figure 5.3: Final version of the Text Layer API

This change in the API does not affect the phrase-based searches, and the main search procedures of the Word-based Suffix Arrays and Byte-codes Wavelet Trees, as presented in the previous Chapter. Nevertheless, there is a new input parameter, the interval, that needs to be

considered for the phrase-based search as well. In the phrase-based bilingual searches, the new interval parameter is set to the coarse segment boundaries of the *skip* offset, stating that no occurrence of the term can span over two different coarse segments. This way, the *skip* position is used to determine the boundaries where a valid occurrence of a term must appear, in the corpora of its respective language. In a bilingual search, if no occurrence appears within those boundaries, it means that no co-occurrence will be found in there, thus a new *skip* position needs to be calculated.

To match hierarchical phrase terms in the corpora, the search procedures of the data structures that support the Text Layer need to be adapted, as now the indexes need to consider terms that have gaps with variable content. As presented earlier in this thesis, the Text Layer is supported by two different data structures. Each one has its own very distinct characteristics, thus both approaches will require changes to support finding terms with gaps. To approach this problem, I divided the hierarchical phrases in two separate types: 1) terms that only have gaps at the beginning and/or ending, such as \$ L_1 \$; 2) terms that have gaps between literals, regardless of having gaps as well at the beginning or ending of the phrase, such as \$ L_1 \$ L_2 \$.

The first type of hierarchical phrase terms is a special case. Since these terms only have gaps at its beginning or ending, we know that its literal is a contiguous subphrase, without any other gap in between. For instance, in Figure 5.1, the pair `global market $ <-> $ global de mercado` are examples of such hierarchical phrase terms. This means that the occurrences of such term with gaps, will always match the exact same occurrences of its contiguous literal. Recovering the fine alignment in Figure 5.2, the occurrences of the term `$ European Parliament`, would match the two occurrences of `$ European Parliament`, since there is no gap other gap between the literals to add additional variable content.

However, the gaps influence the starting offset of the occurrences, and also their length. Thus, for every occurrence of the contiguous literal, the *locate_next* procedure of the Text Layer considers the placement of the gaps to determine the length and the starting offset of the occurrence. If the gap appears at the end, then the starting offset matches the one returned by the indices, while if it appears in the beginning, it is the starting offset, minus the maximum length of the gap allowed by the limit window. In both cases, the length of the occurrence is the size of the contiguous literal, plus the maximum length possible for the gap.

Returning to the example in Figure 5.2, consider the term `$ European Parliament` and a maximum gap length of three fine segments. The first step would be to determine the occurrences of `European Parliament`, at offset nine of the coarse segment on the left, and at offset 14 on the coarse segment on the right. Considering the gap at the beginning, the occurrences of `$ European Parliament` appear at position six, on the left, and at position ten, on the right, corresponding to three fine segments of the window. Regarding the length, the first occurrence has a length of five words, `$ meetings of the European Parliament`, while the second a length of six words `$ to it by the European Parliament`.

The example above opens also a new scenario that is only considered while processing terms with gaps. Figure 5.4 shows a representation of a coarse segment, with six occurrences of literals L_1, L_2 and L_3. Consider the search for the term L_1 \$ L_2 \$ L_3, and that all the occurrences comply

to the window restrictions of the search. In this scenario, there would be four occurrences of the term, at offsets (1),(4),(6), (1),(5),(6), (3),(4),(6) and (3),(5),(6). This means, just for one occurrence of one literal, such as L_3, there can be several occurrences of the term.

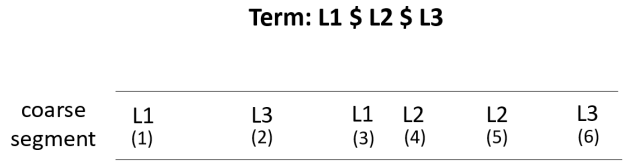

Figure 5.4: Example of occurrences of the literals L_1, L_2 and L_3 in a sentence of the alignment

Following this strategy, the monolingual procedures of the data structures used for the Text Layer, do not need to be changed, as the search will be based on the literal part of the term, which is a phrase as any term supported by the Framework thus far.

For the second type of hierarchical phrase terms, with the gaps appearing between sub-phrase literals, the scenario is different. Here, matching the literals is not enough, since the gaps can have variable content, of variable length. The phrase-based algorithm is designed for matching only contiguous phrases, thus we need a new strategy to solve this problem. This new strategy needs to be applied over the indexes that support the Text Layer, meaning that depending on the index chosen, the search procedure of the index needs to be adapted to support hierarchical phrase terms.

The data structures chosen for supporting the Text Layer are very different. The Word-based Suffix Arrays use integers to represent the text at a word boundaries, over a classical Suffix Array, following a lexicographic order. On the other hand, Byte-codes Wavelet Trees has the words represented with codewords of bytes, following a tree structure and the order of the text. The next Sections describe the two proposed algorithms for locating hierarchical phrases in the corpora, using these two data structures.

5.3 Using Word-Based Suffix Arrays

The text indexed in a Suffix Array follows a lexicographic order, meaning that all the suffixes starting with the same word appear consecutively in the index. Thus, when searching for a word or multi-word term, all its occurrences appear in a consecutive interval of the array.

Suffix Arrays are the most common data structure to support location and extraction of hierarchical phrases over corpora [13, 116]. In particular, the work developed by Baltescu and Blunsom [13], used a strategy that took the advantage of these intervals to support the search for hierarchical phrase terms. In their approach, they trasverse a list of occurrences of a known part of the hierarchical phrase, in the Suffix Array, and check if the remaining literals of the phrase appear within a nearby window in the text. For example, consider the phrase `in $ and $ alike`. When knowing the list of occurrences of `in $ and`, their search procedure would go through all the occurrences of this term, for every occurrence in the list, checks `alike`

appeared after those occurrences, within a limit window. In the next Chapter, this proposal will be explained in more detail.

For our approach based on Word-based Suffix Arrays, we followed a very similar strategy to the one designed by Baltescu and Blunsom. Considering the hierarchical phrase term L_1 \$ L_2 L_3, our strategy selects the less frequent contiguous subphrase literal of the term in the text, for instance L_2 L_3, and determine the list of occurrences of that subphrase in the text. Then, we traverse that list, and for every occurrence, determine if L_1 appears within the pre-set delimiting interval, by accessing directly the text.

This procedure is valid for terms with any number of gaps, following the exact same behavior for the remaining literals. Consider the hierarchical phrase term L_1 \$ L_2 L_3 \$ L_4. After the occurrences for L_1 \$ L_2 L_3 have been found, using the logic mentioned above, the procedure will continue with the list of occurrences of that term, traverse it, and for every occurrence check if L_4 appears within the limit interval, by again accessing the text.

This means that this procedure always follows the same strategy. It starts with the list of occurrences of L_2 L_3, the least frequent term, then checks the literals that precede it, if any, in this case L_1. Since there is a gap between the literals, the limit window is used to ensure that only occurrences of L_1 within that window are returned. Then, having the new list of occurrences, the same check is made for the other literal, to determine which occurrences have L_4 nearby, using the same limit window.

Figure 5.5 shows an example of a fragment of a Word-based Suffix Array, with four suffixes starting at the word `provided`, extracted from the four snippets of the text above, and occurring at offset off_A to off_D. The suffixes are represented with the words itself, instead of the *id* representation, for better understanding the example.

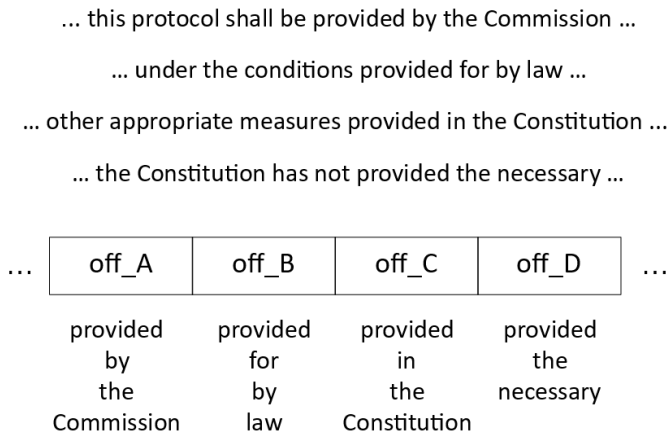

Figure 5.5: Example of a fragment of a Word-based Suffix Array, with four suffixes starting with the word `provided`

Consider a search for the term `has $ provided` and that `provided` is the least frequent literal in the term. The procedure starts by determining the set of occurrences of `provided`, resulting in the four positions of the Array in the Figure. Then, for every position in the interval,

the procedure accesses the text to determine if `has` appears within the limit window. For example purposes, consider that a gap cannot have more than three words in length.

Accessing the first position of the interval, leads the procedure to *off_A*, the occurrence of `provided` at the first snippet of text. Since `has` appears before `provided`, the procedure goes through the text backwards, until finding an occurrence of `has`, or reaching the limit window, in this case, three words. Counting three words backwards from `provided`, we get `protocol shall be`, so no occurrence of `has` exists.

The same thing is done for the remaining positions in the interval, with the same result, except for the last one. In the last position of the Suffix Array interval, there is an occurrence of `has`, one word away from `provided`, thus `has not provided` is a valid occurrence of `has $ provided`, being then returned in the final result.

This strategy is simple an effective, since it is based on the less frequent literal of the hierarchical phrase, thus reducing as most as possible the starting list of occurrences that is traversed. Then, the window delimits the linear processing of the text, backwards and forwards. Additionally, when there is more than one gap, the list of occurrences does not increase. In the worst case, it maintains its size, but the tendency is for the number of occurrences to decrease. Returning to the example in Figure 5.5, if there was an additional gap after `has $ provided`, the list of occurrences would have reduced from four occurrences to one, as a starting point for the next verification.

Since there is a space / time tradeoff concern regarding the Bilingual Framework, we decided to not have any kind of caching mechanisms, to store previously computed list of occurrences of hierarchical phrases. Such an approach would be helpful and would most likely make the search more efficient, in terms of time, but it would also demand more space. For that purpose, and since the procedure always starts with the less frequent literals, the Framework does not use any caching mechanism, thus it does not require any additional space to support the search for hierarchical phrases.

This procedure so far was explained in the context of returning all the occurrences at once. However, due to the skip-based bilingual search, the Text Layer has an iterative behavior, returning one occurrence at a time. To implement this behavior over the Word-based Suffix Arrays, the first change to be done lies on the iterator structure, **t_occ_iter**. This change is only relative to the terms with gaps search, as the phrase-based search maintains the same logic as presented in the previous Chapter.

Figure 5.6 shows the structure for supporting the terms with gaps. The structure shares four fields with the one for the phrase-based search. The first two are the list of occurrences, *occs*, for representing the occurrences of the least frequent literal of the term, and its length, *nr_occs*. Then, it holds the position in the list being processed, *cur_pos*, and the next occurrence of the least frequent literal, for the following skip, *next_occ*. Here, the next occurrences does not have the exact same meaning as in the phrase-based search, since for terms with gaps, this field holds the next occurrence of the least frequent literal in the term. However, it serves the same purpose for the skip-based bilingual search, since the skips just need a guideline for the next iterations of the search. Finally, the last field uses an additional structure, **t_search**, for other

information relevant to the hierarchical phrase-based monolingual search.

```
struct t_literal{
    int *ids;
    int nr_tokens;
}

struct t_window{
    struct t_occ *window_occs;
    int nr_window_occs;
    int cursor;
}

struct t_search{
    struct t_literal **literals;
    struct t_window *window;
    int nr_literals;
    int lf_literal;
}

struct t_occ_iter{
    int* occs;
    int nr_occs;
    int cur_pos;
    int next_occ;
    struct t_search* search;
}
```

Figure 5.6: **t_occ_iter** structure for the hierarchical search in Word-based Suffix Arrays

The structure **t_search** has four fields. The first one represents the literals of the term with gaps, meaning all the contiguous subphrases between gaps. This field also uses an additional structure, **t_literal**, that holds the *ids* of the literal and the number of words / tokens that it has. Then, back at the **t_search** structure, there are additional fields for representing the number of literals in a term with gaps, *nr_literals*, and the position of least frequent literal in the term, *lf_literal*. Finally, there is an additional field related to the window of the search, using the additional structure, **t_window**.

This field and the new structure store a list of occurrences within an interval window of the search. This way, we can hold all the occurrences within a window, to process them one at a time later in the bilingual search, as explained ahead in this Chapter. To represent this behavior, the structure has three fields: a list of occurrences, with offset and length, *window_occs*, the number of occurrences, *nr_window_occs*, and the *cursor* to hold the occurrence being processed.

These structures are then used in the search procedure, that has some changes, at the level of the *start_locate* and *locate_next* methods. Algorithm 5.1 shows the body of the *start_locate* procedure, for supporting terms with gaps. The procedure is quite similar to the one for the phrase-based search. However, as explained earlier, the list of occurrences used as a starting point is of the lowest frequent literal, instead of the whole phrase. For simplification purposes, the algorithm does not show the initialization of some of the fields.

Since these terms have contiguous subphrase literals, separated by gaps, the main difference on this new procedure lies precisely on the processing of the interval of occurrences. In

the phrase-based search, the procedure started by getting the integer ids of the words in the searched term, using the vocabulary, to then proceed to determine the interval of occurrences in the Suffix Array. Here, the logic is similar, but it is done for every literal of the hierarchical phrase term. It starts by determining the literals of the term, using the function *calc_lits*, and then determines the ids for every single one of them, as well as the occurrences in the indexed text. Then, the interval of occurrences of the literal with the less number of occurrences is saved, to work as the starting point for the remainder of the search procedure.

```
struct t_occ_iter* start_locate(struct t_index *index, char* term, int *start,
     int end, int *has_start, int *has_end){
  char **lits = calc_lits(term, &it->search->nr_literals, has_start, has_end);
  int ntokens = 0, num_occs = 0,  min_begin, min_end;
  it->nr_occs = MAX_INT;
  for (int i = 0; i < it->search->nr_literals; i++){
    for(every token in literals[i]){
      it->search->literals[i]->ids[ntokens] = get_id(index->voc, token);
      it->search->literals[i]->nr_tokens++;
    }
    int begin = bisect(it->search->literals[i]->ids, it->search->literals[i]->
        nr_tokens, index->suffix_array, index->text, 0, index->len);
    int end = bisect(it->search->literals[i]->ids, it->search->literals[i]->
        nr_tokens, index->suffix_array, index->text, begin, index->len);
    int freq = end - begin;
    if (freq == 0){
      *start = MAX_INT;
      return it;
    }
    if (freq < it->nr_occs){
      min_begin = begin; min_end = end;
      it->nr_occs = freq;
      it->search->lf_literal = i;
    }
  }
  if (min_end - min_begin == 0){
    *start = MAX_INT;
    return it;
  }
  for(i = min_begin; i < min_end; i++){
    it->occs[i] = index->sa[i];
  }
  sort(it->occs, it->nr_occs);
  if (*start == 0){
    it->cur_pos = 0;
  }else{
    it->cur_pos = bisect(*start, it->occs, 0, it->nr_occs) - 1;
  }
  if (it->occs[it->cur_pos] + it->search->literals[it->lf_literal]->nr_tokens
      < end){
    it->next_occ = *start = it->occs[it->cur_pos];
  }else{
    it->next_occ = *start = MAX_INT;
  }
  return it;
}
```

Listing 5.1: *start_locate* function on Word-based Suffix Arrays for terms with gaps

After the search is started, the search procedure asks for a new occurrence of the term at a time, following the iterative behavior explained earlier. For that, it uses the *locate_next* function, which for hierarchical phrases it follows the logic of going through the list of occurrences of the less frequent literal of the term, and determine if the remaining literals appear within the limit interval in the text. This is done progressively, for every gap between two literals, until all the gaps have been processed.

Algorithm 5.2 shows the procedure for determining the next occurrence of a term with gaps. It starts by determining the occurrence of the less frequent literal of the term, that is closer to the position given by the skip, just like in the phrase-based search. However, now the occurrence of the less frequent literal to be processed need to occur within the input interval, that bounds the occurrences of the terms to a particular area of the corpora. If this occurrence does not appear within the interval, it means that the boundaries need to be recalculated and a new interval needs to be found, around that occurrence. For instance, considering an interval with $start = 300$ and $end = 350$, with a $skip = 320$, with these values being absolute offsets of the text. If the occurrence closest to the skip position occurs at offset 360, then a new interval needs to be determined, to allow occurrences of the searched term, around that position, considering the maximum limit window for the gaps.

The procedure uses a logic based on a window, where every interval processed is considered a window in this context. Whenever an occurrence, or set of occurrences, of a term is found within an interval, those occurrences are represented using the **t_window** structure. This is necessary, since the occurrences are returned one at a time, thus if there is more than one valid occurrence within the same interval, we need to make sure that all of them are returned. The procedure then checks if there is a window still opened, meaning that there are still occurrences to process. If there are not, then a new window is created, with all the occurrences of the term in that interval, if any. If there is a window created, then the procedure checks if there are still occurrences to process within it.

In both cases, if there is an occurrence to process, the function assigns the offset and the length of the next occurrence to processed within the window, using the *cursor* field. If there are more to process after this one, then the next position of the skip is set to the offset of that occurrence. If there is no occurrence to process within the interval, or if the window has no more occurrences left, then the procedure sets the skip position to the next occurrence of the less frequent literal of the term, similarly to what happened in the phrase-based search.

Processing the occurrences within an interval, or window is the core of the hierarchical phrase search. That is done with the function *determine_occs_within_window*, with the procedure for determining an occurrence, or set of occurrences of a term with gaps, within an interval, using Word-based Suffix Arrays.

The first part of the *determine_occs_within_window* function determines the positions in the list of occurrences of the least frequent literal of the search interval. Then, the procedure moves to the actual processing of the occurrences. There are three different scenarios: 1) if the least frequent literal is the first one of the term; 2) if the least frequent literal is the last one of the term; 3) if the least frequent literal appears in the middle of the term.

```
struct t_occ* locate_next(struct t_index *index, struct t_occ_iter *it, struct
     t_interval interval, int *skip){
  if (*skip != it->next_pos){
    it->cur_pos = bisect(*skip, it->occs, 0, it->nr_occs);
    if (it->cur_pos == it->nr_occs){
      *skip = it->next_pos = MAX_INT;
      return NULL;
    }
  }
  if (it->occs[it->cur_pos] > interval.end){
    it->next_pos = *skip = it->occs[it->cur_pos];
  }
  int has_occ;
  struct t_occ *occ;
  if (it->search->window == NULL){
    has_occ = determine_occs_within_window(index->text, it, interval);
  }else{
    has_occ = it->search->window->cursor < it->search->window->nr_window_occs;
  }
  if (has_occ){
    occ->off=it->search->window->window_occs[it->search->window->cursor].off;
    occ->len=it->search->window->window_occs[it->search->window->cursor].len;
    it->search->window->cursor++;
    if (it->search->window->cursor < it->search->window->nr_window_occs){
      it->next_pos = *skip = it->search->window->window_occs[it->search->
          window->cursor].off;
      return occ;
    }
  }
  it->cur_pos++;
  if (iter->cur_pos > it->nr_occs){
    *skip = it->next_pos = MAX_INT;
  }else{
    *skip = it->next_pos = it->occs[it->cur_pos];
  }
  it->search->window = NULL;
  return occ;
}
```

Listing 5.2: *locate_next* function on Word-based Suffix Arrays for terms with gaps

In the first case, the procedure starts with the list of occurrences of the least frequent literal and checks if the next literal in the term appears within the limit interval. This results in a new list of occurrences for that part of the term. Considering the term L_1 \$ L_2 \$ L_3, with L_1 as the least frequent literal, the first iteration of this procedure would return the occurrences of L_1 \$ L_2. That list would be the starting point of the next iteration, to determine if L_3 appears within the search limit window. The final list of occurrences is stored in the *window_occs* field of the **t_window** structure.

The same logic is applied for the other two cases. When the least frequent literal is at the end, then the procedure is the same, but starting from the last literal to the first. The last case combines both, starting from the least frequent literal to the end of the term, to continue then with that temporary result backwards, until the beginning of the term. Considering the same example, with L_2 as the least frequent literal, the procedure would start by determining the

occurrences of L_2 \$ L_3 and then continue backwards to find the occurrences of L_1 \$ L_2 \$ L_3, following the same strategy.

The crucial part of this procedure is finding an occurrence of a literal, before or after another one, within a limit window. This is repeated until all the gaps are matched, using two functions: *match_forward* and *match_backwards*. Both functions use the list of temporary occurrences, calculated up to that point of the search, and then determine if the next literal appears within the limit window. That is done with the *cmp* function, which compares two sequences of integer ids to determine if they are equal, directly over the indexed text. For that comparison, the sequence of identifiers of each literal is used. Whenever an occurrence is found, it is added to a new list of occurrences. Algorithm 5.3 shows the two procedures in more detail.

Consider again that the occurrences of L_1 \$ L_2 were already determined. To find out which occurrences of that subphrase have L_3 appearing afterwards, in the *match_forward* function, the procedure starts by the offset of every occurrence, plus its length. The length is necessary because L_3 can only appear after the occurrence of L_2, thus after the end of L_1 \$ L_2. And then, the comparison with *cmp* is made until the end of the window. This process is repeated always, for every occurrence of the hierarchical phrase L_1 \$ L_2.

The same logic is applied for the function *match_backwards*. For this case, considering that the occurrences of L_2 \$ L_3 are already determined, and the procedure is trying to determine in which cases L_1 appears right before. It is known that, at most, every occurrence of L_1 must end in the position immediately after the occurrence of L_2 \$ L_3 starts. Thus, the procedure checks for an occurrence of L_1 from the beginning of the search interval, until that position.

This procedure traverses linearly several sets of occurrences, but usually, the lists typically decrease considerably in number of occurrences, or there is a literal that is much less frequent that can be used as an efficient starting point for the search.

5.4 Using Byte-codes Wavelet Trees

The text representation on Byte-codes Wavelet Trees is considerably different from the one presented for Suffix Arrays. Here, the words are represented following the order they appear in the text, represented by codewords of bytes, and represented throughout several levels of the tree structure of the index. This radical change between structures, makes the strategy used for Word-based Suffix Arrays, not applicable for searching hierarchical phrases over Wavelet Trees. Additionally, and as far as we know, there is no work published to tackle this kind of search over compressed data structures, and certainly not over these compressed indexes.

The phrase-based search in a Byte-codes Wavelet Tree is heavily based on *descending* and *ascending* the tree structure, anchored by the less frequent word of the phrase in the text. Then, after an occurrence of that less frequent word is found, the procedure then checks if the remaining words of the term appear before or after the occurrence of the less frequent word, depending on their placement in the term.

For the hierarchical phrase-based search, the same logic can be applied, with the search being supported by the least frequent word of the term. Here we can see one big difference to

```
int match_forward(int *text,struct t_occ_iter *it,int beg_pos,int end_pos,int
    literal,int end,struct t_occ *st_occs,int nr_occs,struct t_occ *new_occs){
  struct t_literal *lit = it->search->literals[literal];
  int new_nr_occs = nr_occs;
  for (int i = beg_pos; i < end_pos; i++){
    int start = st_occs[i].off + st_occs[i].len;
    int ind = new_nr_occs;
    while (start <= end){
      if (cmp(lit->ids, text + start, lit->nr_tokens) == 0){
        new_occs[ind].off = off;
        new_occs[ind].len = lit->nr_tokens + (start - off);
        start += lit->nr_tokens;
        ind++;
      }else{
        start++;
      }
    }
    new_nr_occs = ind;
  }
  if (literal == it->lf_literal){
    it->cur_pos += ind;
  }
  return new_nr_occs;
}

int match_backwards(int *text,struct t_occ_iter *it,int beg_pos,int end_pos,
    int literal,int begin,struct t_occ *st_occs,int nr_occs,struct t_occ *
    new_occs){
  struct t_literal *lit = it->search->literals[literal];
  int new_nr_occs = nr_occs;
  for (int i = beg_pos; i < end_pos; i++){
    int end = st_occs[i].off - 1;
    int ind = new_nr_occs;
    while (start <= end){
      if (cmp(lit->ids, text + begin, lit->nr_tokens) == 0){
        new_occs[ind].off = begin;
        new_occs[ind].len = (st_occs[i].off + st_occs[i].len) - begin;
        begin += lit->nr_tokens;
        ind++;
      }else{
        begin++;
      }
    }
    new_nr_occs = ind;
  }
  if (literal == it->lf_literal){
    it->cur_pos += ind;
  }
  return new_nr_occs;
}
```

Listing 5.3: Procedures to find occurrences in the text in a forward and backward direction

the approach based on Word-based Suffix Arrays, since the search is anchored on the words of the hierarchical phrase, instead of the literals. The difference comes from the nature of the data structures, and their search procedures. For the Trees, even for phrase-based searches, it is not possible to determine directly a set of occurrences of a whole subphrase element.

Consider the hierarchical phrase term W_1 \$ W_2 \$ W_3, with W_1, W_2 and W_3 being words, and not literals, with W_3 being the less frequent word in the text and W_2 the most frequent one. At the beginning of a search, the procedure starts by determining the next occurrence of the least frequent word in the term, W_3. Until this part, there are no changes from the phrase-based search algorithm, with the usage of $rank$ and $select$ to $ascend$ and $descend$ in the tree, for the least frequent word.

Figure 5.7 shows a representation of coarse fine segments, with occurrences of words W_1, W_2 and W_3, numbered from (1) to (22). Consider that every occurrence, within the coarse segments, appear within the limit window to match a hierarchical phrase term occurrence. Picking from what was explained above, searching for W_1 \$ W_2 \$ W_3, would start with looking for the next occurrence of W_3, which occurs immediately at offset (2), at $coarse1$.

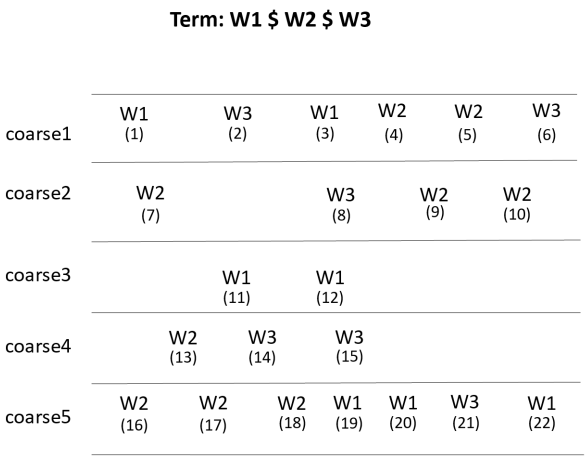

Figure 5.7: Occurrences of the words of the term W_1 \$ W_2 \$ W_3, over five coarse segments

Having an occurrence of W_3 in $coarse1$, then the search proceeds for the second less frequent word, W_1, within $coarse1$, leading to offset (1). Finally, the same process is repeated for W_2, with an occurrence found at offset (4). This means that all the words occur within the same search interval, thus it is possible to have one occurrence of the term. At this point, at offsets (1), (4) and (2) for W_1, W_2 and W_3 respectively, there is no occurrence for the term, since the words do not appear following the correct order. Moving to the next occurrence of W_3, at position (6) of the text, creates a valid occurrence of the term, at offsets (1), (4) and (6).

Looking at Figure 5.7, it is easy to determine that moving to the next occurrence of W_3 would lead to a valid occurrence of the term W_1 \$ W_2 \$ W_3. However, moving for the next occurrence of a word, can actually lead to some occurrences of the term being missed. For example, moving to the next occurrence of W_1, would skip all the occurrences of the hierarchical phrase starting at offset (1). Additionally, due to the nature of the terms with gaps, one occurrence of a word can be part of several different occurrences of a term. For instance, the occurrence W_3 at offset (6), as demonstrated in Figure 5.4, is a part of four different occurrences of the term W_1 \$ W_2 \$ W_3: (1), (4), (6); (1), (5), (6); (3), (4), (6); (3), (5), (6).

For that matter, when the search procedure determines that there is at least one occurrence

of every word within the search interval, it needs to determine all the possible occurrences of the hierarchical phrase term in the interval. This way, we do not miss any occurrence, before moving to the next coarse segment. To accomplish that, we use a known algorithm called Longest Common Subsequence.

5.4.1 Processing occurrences within an interval window

To determine all the occurrences of a term with gaps, within an interval window, we need to find the occurrences of its words in the interval, following the order they appear in the term. Consider that the search interval corresponds to the following section of words in the corpora, separated by blank spaces: $E\ C\ A\ B\ D\ F\ B\ C$. Searching for the term $A\ \$\ B\ \$\ C$ would lead to two occurrences, with the words at positions (3), (4) and (8) and at positions (3), (7) and (8).

This is not a standard substring matching problem, since the matches of a hierarchical phrase term in the text are commonly not contiguous. Thus, we need to find the similarity between a term Y, and the section of the text in the search interval, X, by determining a subsequence of words in X, where the words in X appear by the same order. That problem is the Longest Common Subsequence (LCS). Given two sequences, Y and X, LCS determines the longest subsequence of Y, consecutive or not, that matches a subsequence of X, or the entire X. In the example above, LCS would find the sequence $A\ B\ D\ F\ B\ C$, for $Y = A\ \$\ B\ \$\ C$.

The Longest Common Subsequence problem is often used to compare sequences of DNA in bioinformatics, to determine the similarity between two sequences of DNA, and it is usually solved using dynamic programming [35]. To accomplish that, the procedure uses two sequences $X = X_1X_2...X_m$ and $Y = Y_1Y_2...Y_n$, and two tables. The first table, $length[0..m, 0..n]$, stores the length of the LCS up to a particular position. The final cell of this table, $length[m,n]$ returns the length of the Longest Common Subsequence between X and Y. The second table, $path[1..m, 1..n]$, stores the path followed by the algorithm, while calculating the LCS, which is extremely helpful to extract the subsequence found at the end.

Figure 5.8, extracted from the book *Introduction to Algorithms* by Cormen [35], shows an example for $X = ABCBDAB$ and $Y = BDCABA$, with $length$ being represented with the integers in every cell of the table, and $path$ with the arrows in the cells.

To determine the Longest Common Subsequence between X and Y, we need to build the *length* and *path* tables. This algorithm starts by filling the first row and column of table *length* with zeros, since no occurrence of any word has been matched yet. Then, the algorithm proceeds in a row-major order, meaning that every row of the *length* and *path* tables are processed at a time, from left to right, before moving to the next one. Thus, the algorithm compares every word of X at a time, with all the words in Y, storing in the tables the appropriate value that results from those comparisons.

Every comparison made in the LCS algorithm has three different cases. Consider X_i the word of X at position i and Y_j the word of Y at position j. If $X_i = Y_j$, meaning that the words match, then $length[i,j] = length[i-1, j-1] + 1$ and $path[i,j] =$ " $\nwarrow$ ". This is visible at $i = 2$ and $j = 1$ of the example in Figure 5.8, when there is a match in two sequences of the word

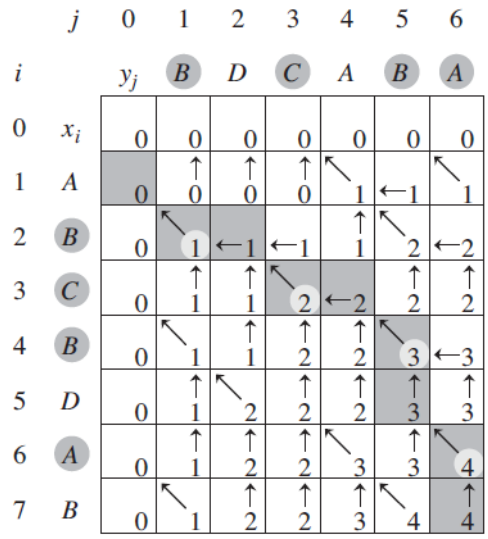

Figure 5.8: Longest Common Subsequence Algorithm

B, leading to $length[1,0]+1=1$. Then, there are two different cases when there is no match between the words. In the first one, if $length[i-1,j] \geq length[i,j-1]$, then $length[i,j] = length[i-1,j]$ and $path[i,j] =$ " $\uparrow$ ". At positions $i=5$ and $j=5$ in Figure 5.8, this case is visible as the word D in X does not match the word B in Y, but the cell above, $length[4,5]=3$, has a bigger value than the cell on the left, $length[5,4]=2$. This case often occurs when there is a mismatch after a match in the previous word. The third and final case covers all the other possibilities, resulting in $length[i,j] = length[i,j-1]$ and $path[i,j] =$ " $\leftarrow$ ", which occurs when there is a mismatch after a match in the previous column. In Figure 5.8, that happens with $i=3$ and $j=4$.

Algorithm 5.4 shows the generic procedure for building these two tables. The first part initializes the first row and column of the *length* table. Then, the actual cells of both tables are filled, according to the strategy explained above. The running time of this algorithm is $O(m \times n)$.

With the tables already built, the next step of this procedure is to determine the Longest Common Subsequence. To accomplish that, there is a recursive algorithm that goes across the *path* table, until reaching its first row or column, by following the arrows in the table. Starting from the last row and column, whenever the procedure finds a $\nwarrow$ in the table, at the cell $[i,j]$, it means that $X[i] = Y[j]$, thus $X[i]$ is an element of the LCS, adding it to the returned sequence. When there is a $\uparrow$ or a $\leftarrow$, then $X[i] \neq Y[j]$, meaning that $X[i]$ is not part of the LCS. In these cases, $X[i]$ is not added to the final sequence, and the procedure continues to the cell above or at the left, respectively. Following this logic, in Figure 5.8, the LCS would be $BCBA$.

Algorithm 5.5 shows the generic procedure for finding the LCS using the *path* table, with m and n being the length of the sequences X and Y. As it is visible, the procedure is recursive and follows the path as explained above. When there is a match between the sequences, then the element is added to the final LCS sequence, returned when the procedure gets to the first row or column of the table. This algorithm takes $O(m+n)$ time.

```
void lcs(int* X, int *Y){
  int m = len(X);
  int n = len(Y);
  int **length = init_table(m,n);
  char **path = init_table(m,n);
  int i, j;
  for (i = 1; i < m; i++){
    length[i,0] = 0;
  }
  for (j = 0; i < n; j++){
    length[0,j] = 0;
  }
  for (i = 1; i < m; i++){
    for (j = 1; i < n; j++){
      if (X[i] == Y[j]){
        length[i,j] = length[i-1,j-1] + 1;
        path[i,j] = "↖";
      }else if (length[i-1,j] ≥ length[i,j-1]){
        length[i,j] = length[i-1,j];
        path[i,j] = "↑";
      }else{
        length[i,j] = length[i,j-1];
        path[i,j] = "←";
      }
    }
  }
}
```

Listing 5.4: Longest Common Subsequence algorithm: building the tables

```
void find_lcs(char **path, int *X,  int m, int n, char *lcs){
  if (m == 0 || n == 0){
    return lcs;
  }
  if (path[m,n] == "↖"){
    find_lcs(path, X, m-1, n-1);
    append (lcs, X[n]);
  }else if (path[m,n] == "↑"){
    find_lcs(path, X, m-1, n);
  }else{
    find_lcs(path, X, m, n-1);
  }
}
```

Listing 5.5: Finding the Longest Common Subsequence

The described procedures find the Longest Common Subsequence between two sequences. In the context of the Bilingual Framework, these sequences would be of words, thus every element of the LCS would be a word. However, the purpose of the search with gaps in the Bilingual Framework is not of determining one Longest Common Subsequence, but all the occurrences of a sequence Y in a larger sequence X, contiguous or not.

For that matter, the search procedure for terms with gaps uses an adaptation of the LCS problem, based on two main differences. First, the sequence Y, or the hierarchical phrase term being searched for, must match completely the sequence X, or the section of the text

within the search interval window. Thus, every word in Y must match in X, contiguously or discontiguously. In Figure 5.8, if $Y = B\$C\$B\$A$, then there would be a valid occurrence of the gapped term Y in X at position two, with length six.

The second change lies on the number of matches reported. With the original LCS procedure, only the longest sequence would be returned. For the proposed hierarchical phrase-based search, we are interested in all the possible matches of a sequence Y in the section of the text X. This means that the behavior of the algorithm needs to change slightly, since it was designed just to return one match.

The base LCS algorithm has three cases, as introduced above, with the first case being when the two elements of the sequences, X_i and Y_i, are equal. When this happens, the *path* table is filled with an $\nwarrow$, which symbolizes a match. However, to return all the occurrences of X in Y, this case is not enough. This way, we created a new subcase that allows, when a match happens, to start a new branch in the *path* table, that would allow two paths continue simultaneously, with the opportunity of returning two different occurrences.

Figure 5.9 shows an example of the LCS algorithm, to build the *length* and *path* tables, with the changes needed to support the hierarchical phrase-based search in the Bilingual Framework. This example covers the sequences $X = DABDBA$ and $Y = A\$B\$C$, with the Figure omitting the gaps for simplification purposes. Considering the hierarchical phrase search supported by the Framework, the sequence $Y = A\$B\$C$ has two occurrences in X. The first occurrence appears at $i = 2$, $i = 3$ and $i = 6$, while the second at $i = 2$, $i = 5$ and $i = 6$, with these positions being the matches of the elements of X in Y.

To enable finding these two occurrences, the *path* table is built differently. By analyzing Figure 5.9, at the cell $i = 5$ and $y = 2$, there are two arrows, one $\nwarrow$ and one $\uparrow$. This happens because there is a new match with the element B, while there is still a path “opened"with A and the first occurrence of B, thus marking that the first path is still eligible, while a new one can branch from the original one. This way, when determining all the occurrences of Y in X, when the procedure visits that cell, it proceeds both ways, following the two possible paths created in the *path* table.

More formally, the new subcase for the LCS procedure is defined as follows: if $X_i = Y_i$ and $length[i-1][j] \geq length[i][j]$, then a $\uparrow$ is added to the *path* table in that cell, next to the $\nwarrow$.

Finally, since the searched terms have literals and gaps, every element of the sequences tested with the LCS algorithm is a literal, with one or more words. To determine if a literal in Y matches, or not, a literal in X, the common procedure to match a literal in Byte-codes Wavelet Trees is used, based on $ascending$ and $descending$ the tree structure.

To support this behavior, the iterator structure of the Byte-codes Wavelet Tree, used for the phrase-based search, needs to be changed. Figure 5.10 shows the new implementation of the structure, and the other auxiliary structures used by the Text Layer.

The new version of the structure **t_occ_iter** has two fields: the *root_pos* with the current position in the root in every iteration, and a new structure, **t_search**. This new structure varies from phrase-based to hierarchical phrase-based terms. For the first ones, the structure holds

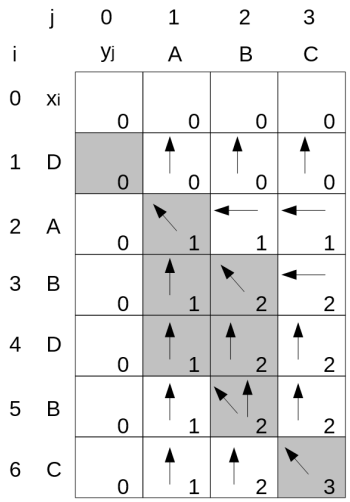

Figure 5.9: Example of the modified LCS algorithm, for the Bilingual Framework

```
struct t_window{
    int start;
    int end;
    struct t_occ **occs;
    int nr_occs;
    int cursor;
    int** length_table;
    int** path_table;
}

struct t_word{
    struct code cod;
    int nr_occs;
    int leaf_pos;
    int root_pos;
}

struct t_search{
    struct t_word *words;
    struct t_window *window;
    int nr_words;
    int lf_word;
    int *literals;
    int nr_literals;
}

struct t_occ_iter{
    int root_pos;
    struct t_search *search;
}
```

Figure 5.10: **t_occ_iter** structure for the hierarchical search in Byte-codes Wavelet Trees

the same fields presented in the previous Chapter. For the terms with gaps, the structure is defined as well in Figure 5.10.

The structure **t_search** represents important fields necessary for the processing of the hierarchical phrase-based search. First, it holds all the *words* in the searched term, with an array of

elements of the type **t_word**. This structure has the codeword of a words (*cod*), the number of its occurrences (*nr_occs*) in the indexed text, and the current positions in the leaf and root, *leaf_pos* and *root_pos* respectively. Going back to the **t_search** structure, the other fields are the number of words in the term (*nr_words*), the least frequent word of the term in the text (*lf_word*), the *literals* of the term and the how many literals the term has (*nr_literals*). This representation of the literals, with a sequence of integers, holds the length of each literal in number of words. For instance, considering the term $AB\$C$, the sequence *literals* would have two positions [2, 1], meaning that the first literal has two words, while the second literal has just one word.

Additionally, the structure has a last field, *window*, to represent the LCS window explained above. This field is used when the LCS procedure is executed over a window of the corpora. It has a *start* and an *end* position to delimit the window, the tables *length_table* and *path_table*, the occurrences found *occs* within the window, alongside the number of occurrences, *nr_occs*. Finally, there is a field *cursor* that represents which occurrence within the window is being processed, to support the iterative behavior.

Algorithm 5.6 presents the procedure to create the *path* and *length* tables, adapted from the original LCS procedure. To represent the four path cases, including the new subcase, we use a representation based on integers. A $\leftarrow$ is represented by -1, a $\uparrow$ by 1, $\nwarrow$ alone by 1 and the combination between the $\nwarrow$ and $\uparrow$ by 2, which is the new subcase. The procedure uses the structure **t_search**, the length of the tables m and n and the bytemaps of the tree structure to determine the matches.

The procedure starts just like the original LCS procedure, with the initialization of the tables and the first row and column of the table *length_table*. Then, for every pair of $[i, j]$, the procedure verifies if there is a match between the literal of the searched term in the text, using the *check_literal_codes* function. This function performs if a phrase-based literal exists in the text, using the procedure introduced in the previous Chapter. Then, if the length of the match is the same as the length of the literal, it means that a match was found, thus a 1 is added to the *path_table* and the length of the sequence increases by 1.

The next step is the verification of the new subcase. If the position above of *length_table* is greater or equal to the current position being processed, it means that a new branch should be created, thus the position in the *path_table* is increased by 1, resulting in the value 2 that represents the $\nwarrow$ and $\uparrow$ case. If there is no match, then the procedure resumes as the original one, adding 0s ($\uparrow$) or -1s ($\leftarrow$), depending on the case.

Algorithm 5.7 shows the procedure to determine all the occurrences of the term in the window of the text, using the tables determined in Algorithm 5.6. As the original algorithm, it uses a recursive behavior, with two main differences. First, instead of building the Longest Common Subsequence, whenever the recursive procedure reaches the first column, checks if the number of elements matched is the same as the number of literals in the searched term. If it is, then the occurrence at position m of the window is saved in *occs*, alongside the length of the occurrence. This is done using the auxiliary function *save_occurrence*. If it is not, the procedure finishes without adding a new occurrence. Then, the recursion is built by following the same cases of the original procedure. The main difference lies on the new subcase, which continues

```
void build_tables(struct t_search *search, int m, int n, struct bytemap **
    bmaps){
  struct t_window *w = search->window;
  w->length_table = init_table(m,n);
  w->path_table = init_table(m,n);
  int i, j;
  for (i = 1; i < m; i++){
    w->length_table[i,0] = 0;
  }
  for (j = 0; i < n; j++){
    w->length_table[0,j] = 0;
  }
  for (i = 1; i < m; i++){
    for (j = 1; i < n; j++){
      int lit_len = search->literals[j-1];
      int match_len = check_literal_codes(search->toks, w->start + i - 1,
          bmaps);
      if (match_len == lit_len){
        w->length_table[i,j] = w->length_table[i-1,j-1] + 1;
        w->path_table[i,j] = 1;
        if (w->length_table[i-1][j] ≥ w->length_table[i][j]){
          w->path_table[i,j]++;
        }
      }else if (w->length_table[i-1,j] ≥ w->length_table[i,j-1]){
        w->length_table[i,j] = w->length_table[i-1,j];
        w->path_table[i,j] = 0;
      }else{
        w->length_table[i,j] = w->length_table[i,j-1];
        w->path_table[i,j] = -1;
      }
    }
  }
}
```

Listing 5.6: Adapting the LCS procedure to build the *length* and *path* tables

the path upwards as well, alongside continuing diagonally, as the original procedure does when a $\nwarrow$ is found.

5.4.2 Proceding to the next occurrence

After all the occurrences within an interval are processed, the search procedure continues to determine the next occurrence of the term. Figure 5.11 recovers the example being followed before, for the search term W_1 \$ W_2 \$ W_3, and with the occurrences of the words in the term spread out through five coarse segments. Since the procedure stopped with all the occurrences appearing within a search interval processed, a new occurrence of the least frequent word in the term needs to be found.

Whenever the search algorithm reaches such a state, the less frequent word is recalculated again, since after processing a search window, or after a skip, the less frequent word can be different from that point of the corpora onwards. In Figure 5.11, from $coarse2$ onwards there are five occurrences of W_1, six of W_2 and four of W_3, thus W_3 is still the less frequent word. Thus, the algorithm proceeds for the next occurrence of W_3, which appears at $coarse2$, at offset (8).

```
void find_sequences(struct t_window *w, int m, int n, int occ_len, int
    nr_literals, int elements_matched){
  if (n == 0){
    if (elements_matched == nr_literals){
      save_occurrence(w->occs, w->cursor, m, occ_len);
    }
    return;
  }
  if (m == 0){
    return;
  }
  if (w->path_table[m,n] >= 1){
    find_sequences(w, m-1, n-1, occ_len-1, nr_literals, elements_matched + 1);
    if (w->path_table[m,n] == 2){
      find_sequences(w, m-1, n , occ_len+1, nr_literals, elements_matched);
    }
  }else if (path[m,n] == 0){
    find_sequences(w, m-1, n, occ_len+1, nr_literals, elements_matched);
  }else{
    find_sequences(w, m, n-1, occ_len, nr_literals, elements_matched);
  }
}
```

Listing 5.7: Finding the matched sequences

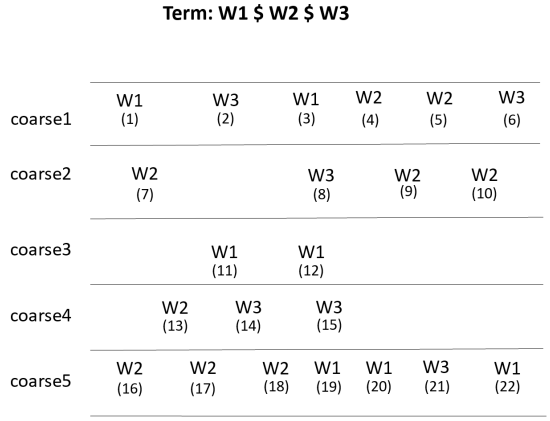

Figure 5.11: Occurrences of the words of the term W_1 \$ W_2 \$ W_3, over five coarse segments

This new position of the word serves as base for creating the next search window, where all the other words of the term need to appear, to find at least one valid occurrence. To determine if the other words appear within the same window, the algorithm continues for W_1, the next less frequent word, obtaining its next occurrence at $coarse3$.

At this stage, there is already a mismatch, since W_1 appears outside the $coarse2$ window. This is an example of a second scenario that may happen, when the next occurrence of one word in the searched term, appears outside the limit search window. Here, in opposition to what happens after going through all the occurrences inside a window, the search procedure continues by using the occurrence of the term ahead in the corpora as anchor for a new search window. In this particular example, that is the case of W_1. This means, that the search procedure

will attempt to find an occurrence of W_2 and W_3, within the interval where W_1 appears, in this case $coarse3$, starting by the less frequent of the two.

The search continues by finding the next occurrence of W_3 in, or after, the window bounded by $coarse3$, resulting in offset (14), at $coarse4$. Again, the occurrence of this word appears outside the window that was set, thus no occurrence of the term will be found within $coarse3$.

The same logic applied above continues, with W_3 being the anchor now, and with the procedure continuing with the next occurrence of W_1 in, or after, $coarse4$, resulting in offset (19), at $coarse5$. Since the two occurrences appear in different sentences, the process is repeated for W_3, getting to position (21). Here, both occurrences of the words appear within the same window, thus the first occurrence of W_2 is determined within $coarse5$, resulting in offset (16).

Up to this point, the search procedure is at the same situation that it was at $coarse1$, meaning that the Longest Common Subsequence algorithm runs. However, no occurrence of the term W_1 \$ W_2 \$ W_3 is found in this segment, since there is no set of occurrences of the words that would match the order of the words in the term.

Considering that this procedure will be used in the context of the skip-based bilingual search, the search windows that open throughout the processing of a term with gaps are heavily influenced by the skips. This means that when a new skip position is given to the monolingual search, the search window is defined around that same position.

This search procedure on Byte-codes Wavelet Trees mixes the phrase-based procedure of this data structure, by *ascending* and *descending* the tree by the least frequent word, with the skip-based search logic of skipping occurrences of the words of the hierarchical phrase term. In the previous example, that happened with the occurrences of W_2 at $coarse2$ and $coarse4$.

This behavior is implemented in the *start_locate* and *locate_next* functions of the Text Layer API, now slightly changed to support the search for terms with gaps, as presented in Figure 5.3. First, Algorithm 5.8 shows the generic procedure to get the *locate_next* function, heavily based on auxiliary functions. Using the *skip* position passed as parameter to the function, the procedure checks if the position appears within the boundaries of the search interval. If it does, then checks if the skip position matches any occurrence of the term, using the *check_occurrence* function. If not, determines a next occurrence with the *get_new_occurrence* function. Before finishing the procedure, either if an occurrence was found or not, the procedure determines the next position of the skip, using the *get_next_skip* function. Over the next paragraphs, we will analyze these steps in more detail.

The Algorithm 5.10 shows the *check_occurrence* procedure, used in the *locate_next* function. This procedure checks if the skip position, passed as parameter to the *locate_next*, appears within the interval of an opened window.

The procedure starts by checking if the *skip* position matches the position at the root, *root_pos*, that was calculated in the previous iteration, as the potential position for a new occurrence. If it is not, then it means that the skip led the search to a position ahead in the text, as in the phrase-based search, and a new position needs to be determined. To accomplish that, as explained above, the procedure determines the least frequent word from that point, using the also displayed *get_lf_word* function, and finds the new *root_pos* for that skip position. That

```
struct t_occ* locate_next(struct t_index *index, struct t_occ_iter *it, struct
     t_interval interval, int *skip){
  if (*skip < interval.start || *skip > interval.end){
    return NULL;
  }
  struct t_occ *occ;
  if (check_occurrence(it, index->tree->bmaps, occ, *skip, interval)){
    get_next_skip(index->tree->bmaps, it, skip, interval.end, 1);
    return occ;
  }
  if (get_next_occurrence(it, index->tree->bmaps, occ, interval)){
    get_next_skip(index->tree->bmaps, it, skip, interval.end, 1);
    return occ;
  }else{
    get_next_skip(index->tree->bmaps, it, skip, interval.end, 0);
    return NULL;
  }
}
```

Listing 5.8: *locate_next* function for the hierarchical phrase-based search supported by Byte-codes Wavelet Trees

is done by $descending$ the tree structure with the *skip* position, finding the respective occurrence in the last node of the codeword of the least frequent word, and then $ascend$ back to the root, to find the actual occurrence in the text.

This process is quite similar to the one used in the phrase-based search, with the difference that the least frequent word can vary depending on the position of the text where we are. This means that the least frequent word is determined every time there is the need to adjust the search to a new skip position. The *get_lf_word* function analyzes how many occurrences each word has left to process, returning the one with less occurrences left.

```
int check_occurrence(struct t_occ_iter *it, struct bytemap **bmaps, struct
    t_occ *occ, int skip, struct t_interval interval){
  if (skip != it->root_pos){
    int lf_word = get_lf_word(it->search);
    it->root_pos = new_word_pos(it->search->words[lf_word], bmaps, skip);
    it->search->lf_word = lf_word;
    return 0;
  }
  if (it->search->window == NULL){
    return 0;
  }
  if (it->search->window->cursor < it->search->window->nr_occs){
    int ind = it->search->window->cursor;
    occ->off = it->search->window->start+it->search->window->occs[ind].off;
    occ->len = it->search->window->occs[ind].len;
    return 1;
  }
  return 0;
}
```

Listing 5.9: Procedure to check an occurrence of a hierarchical phrase-based term

Then, if the skip position actually matches the one that was determined in the previous

iteration, meaning that there was no actual skip, the procedure checks if there is a window open, with occurrences to return. If it is, then the next occurrence is returned. Here, the algorithm uses the occurrences inside a search window, calculated using the adaptation of the Longest Common Subsequence. Otherwise, the window is destroyed. This way, we make sure that all the occurrences of the term found inside a particular search window are considered for the skip-based bilingual search.

When there was no more occurrences to process in the window, the *locate_next* procedure tries to determine a next occurrence for the searched term. That is done with the *get_next_occurrence* procedure, presented in Algorithm 5.10.

```
int get_next_occurrence(struct t_occ_iter *it, struct bytemap **bmaps, struct
    t_occ *occ, struct t_interval interval){
  if (check_words_within_interval(it->search, bmaps, interval){
    create_window(it->search, bmaps, interval);
  }
  if (it->search->window != NULL){
    int ind = it->search->window->cursor;
    occ->off = it->search->window->start+it->search->window->occs[ind].off;
    occ->len = it->search->window->occs[ind].len;
    return 1;
  }
  return 0;
}
```

Listing 5.10: Procedure to get the next occurrence of a hierarchical phrase-based term

In this procedure, the first step is to determine if every word in the searched term appears within the specified interval, in the *check_words_within_interval*. This interval is calculated using the position given by the skip, by considering the maximum allowed length of the term. In this step, the procedure determines the new occurrences of every word in the searched term, except for the least frequent one, which was determined already by the skip. If the position of a word is already within the search interval, we proceed for the next one. If it is not, the algorithm determines a new position for the word, *new_word_pos*, starting from the beginning of the interval. This process is done by $descending$ in the tree by that given position, to determine the respective leaf position, and then $ascend$ again to the root to determine the absolute offset of that occurrence. If the absolute offset appears outside the interval, there would be no occurrence of the term inside that window.

Back to the *get_next_occurrence* function, if the algorithm determines that all the words in the term appear within the search window, then the LCS-based procedure will run to determine the occurrences of the term within the window. Then, when those occurrences are found, the first one will be returned by the function.

An important part for determining the next occurrence of a searched term, is to define the LCS-based procedure that determines all the occurrences of the term, within a search window. As mentioned earlier in this Chapter, this window is extremely important to bound the occurrences of the terms with gaps, to avoid matching occurrences that would not be statistically valuable for Machine Translation. For that matter, when all the words of the searched term

appear within the window, the LCS-based procedure is executed. Algorithm 5.11 shows the procedure that accomplishes that task.

```
int create_window(struct t_search * search, struct bytemap **bmaps, struct
    t_interval interval){
  search->window->start = interval.start;
  search->window->end = interval.end;
  int m = search->window->end - search->window->start;
  int n = search->nr_literals;
  build_tables(search, m, n, bmaps);
  if (search->window->length_table[m][n] != search->nr_literals){
    search = NULL;
  }else{
    search->window->occs = init_array(m*m);
    search->window->cursor = 0;
    find_sequences(search->window, m, n, 0, search->nr_literals, 0);
    search->window->nr_occs = search->window->cursor;
    search->window->cursor = 0;
  }
  free_tables(search->window, m);
}
```

Listing 5.11: Procedure to create a search window for the LCS-based procedure

Within this function, the window structure that supports the LCS-based procedure is created, with the bounds of the search interval. Then, the tables are built, using the *build_tables* procedure. When the *length* and *path* tables are built, the next step verifies if the last position of the *length* table matches the number of words in the searched term. If it does, it means that we found at least one occurrence of the term in the corpora, thus the *find_sequences* procedure is used to find and store all the occurrences found within the window. Otherwise, it means that there was no match of the term in the area of the indexed text defined by the window.

In the *locate_next* function, regardless if a new occurrence is found or not within a bounded window of the text, the position for the next skip needs to be calculated, for the skip-based bilingual search. That is done in the *get_next_skip* function presented in Algorithm 5.12.

This procedure starts by determining if there are any occurrences in an opened window to process, and if all the words in the term still have occurrences left in the text, from that position onwards. If that is not the case, the search ends, since there are no more occurrences to return.

There are two relevant cases tackled in this function. In the first case, if there are still occurrences within the opened window to process, the position of the next skip will be the position of that next occurrence in the list calculated by the LCS-based procedure. If not, then it means that the search window is fully processed, thus we need to determine, for every word, if there are any occurrences after the end of the processed interval. That is done by $descending$ the tree using the position of the end of the interval. If that is the case, then the least frequent word at that point is determined, the procedure $ascends$ the tree with that word, and the resulting position at the root is the position for the next skip.

In the second case, there was no occurrence found, meaning that there are occurrences of words appearing outside the expected search window. If the position at the root of the least

```
int get_next_skip(struct bytemap **bmaps, struct t_occ_iter *it, int end, int
    has_occ){
  if (has_occ){
    if (it->search->window->cursor < it->search->window->nr_words){
      it->root_pos = it->search->window->start + it->search->window->occs[it->
          search->window->cursor];
    }else{
      for (int i = 0; i < it->search->nr_words; i++){
        struct t_word *word = it->search->words[i];
        word->leaf_pos = descend_by_word(bmaps, it->search->window->end, word)
            + 1;
        if (word->leaf_pos ≥ word->nr_occs){
          it->root_pos = MAX_INT;
          break;
        }
        if (i == it->search->nr_words){
          it->search->lf_word = get_lf_token(it->search);
          struct t_word *word = it->search->words[it->search->lf_word];
          it->root_pos = ascend_by_word(bmaps, word->leaf_pos, word);
        }
      }
    }else{
      if (it->root_pos < end){
        int word_ahead = determine_word_ahead(it->search);
        struct t_word *lf_word = it->search->words[it->search->lf_word];
        lf_word->leaf_pos++;
        if (lf_word->leaf_pos ≥ lf_word->nr_occs){
          it->root_pos = MAX_INT;
        }
        lf_word->root_pos = ascend_by_word(bmaps, lf_word->leaf_pos, lf_word);
        if (lf_word->root_pos < it->search->words[word_ahead]->root_pos){
          it->root_pos = it->search->words[word_ahead]->root_pos;
          it->search->lf_word = word_ahead;
        }else{
          it->root_pos = lf_word->root_pos;
        }
      }
    }
    return it->root_pos;
}
```

Listing 5.12: Procedure to get the position of the next skip for the hierarchical phrase-based search supported by Byte-codes Wavelet Trees

frequent word is after the search interval, then that is the position of the next skip. If it is still within the interval, it means that there is at least one occurrence that did not appear within the expected window. Thus, the procedure finds the word with the occurrence that is ahead in the text, as explained before in this Section, with the function *determine_word_ahead*. Then, the procedure determines the next position of the least frequent word, if any, as well in the root. This is done, as usual, using the $descend$ and $ascend$ in the tree structure. The position of the least frequent word and the position of the word ahead are compared, and the biggest one will be the position of the next skip, for the next iteration of the search.

The *start_locate* function, despite not changing too much, also has some slight changes. Algorithm 5.13 shows the procedure of this function, for supporting hierarchical phrase-based

searches, with some minor details missing.

```
struct t_occ_iter* start_locate(struct t_index *index, char* term, int *start,
     int end, int *has_start, int *has_end){
  int *voc_positions = get_vocabulary_positions(index->voc, term, it->search->
      literals, &it->search->nr_literals, &it->search->nr_words, has_start,
      has_end);
  if (!nr_literals){
    *start = it->root_pos = MAX_INT;
    return it;
  }
  int lf_nr_occs = MAX_INT;
  it->search->words = init_array(it->search->nr_words);
  for (i = 0; i < it->search->nr_words; i++){
    it->search->words[i].cod = get_codeword(voc_positions[i], index);
    struct code c = it->search->words[i].cod;
    it->search->words[i].leaf_pos = 0;
    it->search->words[i].nr_occs = rank(index->tree->bmaps[c.branches[c.len
        -1]], size(index->tree->bmaps[c.branches[c.len-1]])-1, c.bytes[c.len
        -1]);
    if (it->search->words[i].nr_occs < lf_word){
      lf_nr_occs = it->search->words[i].nr_occs;
      it->search->lf_word = i;
    }
  }
  if (lf_occs == 0){
    *start = it->root_pos = MAX_INT;
    return it;
  }
  if (*start != 0){
    it->leaf_pos = descend_by_pos(index, it, *start);
  }
  *start = it->root_pos = ascend_by_pos(index, it, it->leaf_pos);
  return it;
}
```

Listing 5.13: *start_locate* function for the hierarchical phrase-based search supported by Bytecodes Wavelet Trees

Due to the nature of the search, the first difference from this procedure to the phrase-based one, lies on the *get_vocabulary_positions* function, that not only finds the words of the term in the vocabulary, as it also determines the literals in the term. Then, if there is at least one literal, the codeword of each word is determined as in the phrase-based procedure, but then the occurrences of every word in the text are determined with the *rank* function. With the phrase-based search this only happened with the least frequent word, but here, due to the nature of the algorithm, we need that value for every word of the searched term. Then, the least frequent word of the term is determined, as well as the position of its first occurrence in the root, to be used as the starting point for the first iteration of the search.

5.5 Skip-based bilingual search for hierarchical phrase terms

As demonstrated in the previous Chapter, the skip-based bilingual search is extremely important for the phrase-based search, with a considerable number of individual occurrences of the

terms being skipped, for returning the exact same amount of bilingual co-occurrences. For hierarchical phrases, where the number of occurrences tends to increase due to the variable nature of the gaps, this skip-based search is even more important, since more individual occurrences of the terms are expected.

To support the skip-based bilingual search for hierarchical searches, the procedures above need to be repeated for both languages. Just like in the phrase-based search, the skip-based procedure is agnostic of the data structures used for the Text Layer. In fact, the skip-based bilingual search is exactly the same for supporting the phrase-based and the hierarchical phrase-based search. To be able to accomplish that, the procedure just needs some slight modifications.

In the previous subsections, I described the changes to the Text Layer necessary to support the search for terms with gaps. For the Alignment Layer, much like for the phrase-based search, it is not necessary to apply any changes to its structure and behavior. This means that despite being a crucial part of the skip-based bilingual search and of the Bilingual Framework, the original Alignment Layer definition is able to support all the evolutions in functionality and behavior of the Framework.

The monolingual hierarchical phrase-based search is highly based on a window, calculated in every iteration of the search, considering the positions provided by the skip and the maximum length allowed for every term. Therefore, before requesting to the Text Layer the next occurrence of a given term, the procedure needs to determine the search window. That is done with a query to the Alignment Layer.

This way, using the position given by the skip, where the search will resume, the Alignment Layer determines the fine and coarse segment of the alignment where that position occurs. Then, using the bounds of the coarse segment and the maximum length of the term allowed, the interval is determined and used in the Text Layer to filter the search for the next occurrence within the calculated window. The actual bilingual search procedure maintains its structure and logic, with a slight difference. Since now the bounds of the coarse segments where the occurrences appear are determined before actually calculating the occurrence itself, there is no need to do that when the skips are being determined.

Algorithm 5.14 shows the part of the bilingual search procedure that determines the next monolingual occurrence, in a given language. This is the part of the bilingual search that actually changes, since the skip-based procedure stays the same as in the phrase-based search. It also shows the structure **results_iter** with three new fields: *has_start* to indicate if there is a gap at the beginning of the term; *has_end* for the same purpose but for a gap at the end of the term; and *search_window* with the search interval to match the term.

The procedure first picks the skip position, *next_pos*, for that particular language and determines where that offset appears in the alignment. If the coarse segment calculated is new, namely different from the field *current_coarse*, the new bounds are determined using the function *define_search_interval*. Since there can be several occurrences of the term within a coarse segment, this avoids determining its bounds every time a new occurrence is requested, when the coarse segment is still the same.

Then, the interval is determined and the new occurrence is determined with the Text Layer's

```
struct results_iter{
  struct t_occ_iter *its[2];
  struct t_occ occs[2];
  int fine[2];
  int coarse[2];
  struct si_bounds coarse_bounds;
  int num_bilingual_matches;
  int num_matches[2];
  int window;
  int skip_side;
  int next_pos[2];
  struct t_interval search_window;
  int has_start;
  int has_end;
}

int next_monolingual_occurrence(struct bil_fram *fr, struct results_iter *it,
    int side){
  int coarse, fine;
  while(!(it->next_pos[side] != MAX_INT)){
    int next_pos = it->next_pos[side];
    fr->aapi->locate(fr->ai, next_pos, &coarse, &fine);
    fr->aapi->coarse_bounds_one_side(fr->ai,coarse,side,&(it->coarse_bounds));
    int start, end;
    start = define_search_interval(fr, it, side, fine, &end);
    it->search_window.start = start;
    it->search_window.end = end;
    struct t_occ *occ=fr->tapi[side]->locate_next(fr->ti[side],it->its[side],
        it->search_window,&(it->next_pos[side]));
    if (occ != NULL){
      it->occs[side].off = occ.off;
      it->occs[side].len = occ.len;
      it->num_matches[side]++;
      if (occ.off != next_pos){
        fr->aapi->locate(fr->ai, occ.off, &coarse, &side);
      }
      it->coarse[side] = coarse;
      it->fine[side] = fine;
      return 1;
    }
  }
  return 0;
}
```

Listing 5.14: Function to determine the next occurrence of a term with gaps in one language

locate_next function. If the offset calculated was not the one expected by the skip, then the respective alignment information is determined to complete the match.

To calculate the bounds of the window, function *define_search_interval* uses the Alignment Layer, namely the function that returns the offset where a fine segment begins. This fine segment already considers the maximum allowed window, thus it is the first fine segment where the occurrence of the hierarchical phrase can start. The only exception is when this fine segment falls outside a sentence boundary. In that case, the beginning of the window is the beginning of the sentence, or coarse segment.

The same logic is applied for the end of the window, but this time considering the fine

segment of the skip position, plus the maximum size of the window. Again, if that offset crosses the coarse segment boundaries, the window ends with the last position of that coarse segment in the text.

This procedure is repeated until there are no more occurrences to return of a given pair of terms. In the next Section, we present the monolingual and bilingual search time response for the hierarchical phrase-based search. These results were determined for both approaches of the Text and Alignment Layer, and also focus on the number of individual occurrences of the terms, for different limit windows, were skipped.

5.6 Results

Considering the changes made to the search procedure to support the matching of hierarchical phrases, this Section shows the hierarchical phrase-based search time response of the Framework. For these experiments, we used the same data structures for the Text and Alignment Layers, the Word-based Suffix Arrays (WSA) and the Byte-coded Wavelet Trees (WT), for the texts, and Arrays of Integers (ArrInt) and Bitmaps (Bmaps), for the alignment.

As explained before, the Bilingual Framework does not require any additional data structure in its core to support the hierarchical phrase-based search, apart from the ones used in runtime to determine the occurrences. This means that just like the skip-based search procedure, the hierarchical phrase-based search does not increase the Framework's space requirements.

To measure the search query time response for the hierarchical phrases, 430355 pairs of hierarchical phrases were extracted in English (EN) and Portuguese (PT), using the same five corpora used in the previous experiments: EUconst, EMEA, Europarl, DGT and Eurlex. The hierarchical phrases extracted only have one gap, either in English or Portuguese, with the gap appearing anywhere in the phrase, including at the beginning and at the end. All these pairs are correct translations of each other, with the gap on both sides indicating variable content that must not invalidate the pair as a correct translation. For these experiments, we did not include de DE-PT pair, since it is not as advanced as the EN-PT, thus the number of hierarchical phrases extracted is considerably smaller. The experiments were made using the same machine with 16 Gigabytes Memory RAM, a Intel Core I7, 3.4 GHz processor, using Ubuntu Linux Kernel 3.2.0.

Since the changes to the Bilingual Framework are just at monolingual level, the first experiments were made over each language independently. For that purpose, I filtered the hierarchical phrases in the 430355 pairs and removed the repetitions, since one hierarchical phrase can have more than one translation in the other language. For instance, `fair $ system` can be translated in Portuguese by `sistema $ justo` or `sistema $ equitativo`. For example, `fair health system` <-> `sistema de saúde justo` and `fair trade system` <-> `sistema de comércio equitativo` are two pairs that would match the hierarchical phrase. This pre-processing step generated two sets of hierarchical phrases without repetitions, one per language, enabling that the phrases searched, are all unique, resulting in 194260 English and 255695 Portuguese hierarchical phrases.

Table 5.1 shows the hierarchical phrase-based search time results for the Bilingual Framework implementation using Arrays of Integers for the alignment, alongside the number of occurrences returned, per corpora. For these experiments, we set the maximum window for matching gaps of three fine segments. Table 5.2 displays the same results, with the alignment being represented by Bitmaps. In this Table, we do not repeat the number of occurrences, as they are the same as the ones displayed in Table 5.1.

Table 5.1: Monolingual search time results for the hierarchical search procedure, with a limit window of three fine segments, using Arrays of Integers to support the Alignment Layer

Corpora	**Nr. Occurrences**		**Time WSA (s)**		**Time BOWT (s)**	
	EN	**PT**	**EN**	**PT**	**EN**	**PT**
EUconst	173.610	227.588	6	7	4	5
EMEA	6910.065	11785.553	38	42	288	346
Europarl	21537.297	31883.074	158	137	1936	2032
DGT	45113.590	73729.861	304	329	7062	7174
Eurlex	59503.496	100908.275	423	469	11419	12918

Table 5.2: Monolingual search time results for the hierarchical search procedure, with a limit window of three fine segments, using Bitmaps to support the Alignment Layer

Corpora	**Time WSA (s)**		**Time BOWT (s)**	
	EN	**PT**	**EN**	**PT**
EUconst	6	7	5	6
EMEA	81	88	406	487
Europarl	333	290	2450	2520
DGT	695	714	8046	8207
Eurlex	943	959	12775	14349

The results in Tables 5.1 and 5.2 demonstrate again the difference, in the search time response, of the approaches based on Suffix Arrays and on compressed indices. Except for EUconst, which is the smaller corpora, all the approaches based on Byte-codes Wavelet Trees or Bitmaps, either in PT or in EN, consume much more time than the uncompressed alternatives.

In comparison with the monolingual phrase-based results, the hierarchical search procedure proves to be less efficient. This conclusion is clear when comparing with the monolingual phrase-based search, where for instance the approach based on Word-based Suffix Arrays and Arrays of Integers takes four times more for Eurlex, in English. This is even more relevant considering that the number of searches, and thus number of occurrences found in the phrase-based scenario is considerably larger. On the other hand, with the Byte-codes Wavelet Trees, the difference is not that high, but still it is noticeable how, for less number of searches, the algorithm is much more time consuming to match phrases with gaps. This behavior was expected, since the search procedure is much more complex than the phrase-based one.

Even comparing these hierarchical phrase-based search results with the bilingual skip-based search for phrases, in Tables 4.4 and 4.6, we can understand how heavy this procedure is. The skip-based bilingual search takes more time using the compressed approaches, but for the uncompressed approaches it is even faster than the hierarchical phrase-based search. And even in the compressed version of the Framework, one could expect a bigger difference in times, since the bilingual search determines occurrences of two elements, in two different languages, thus two different corpora.

Analyzing the results per language, we can conclude that the Portuguese side is more time consuming for most of the cases, as it also has more occurrences returned than the English side. This is visible for all cases, except for the approaches based on Word-based Suffix Arrays, in Europarl. This corpora is the one with more repetitions from the five, since it is based on transcriptions of speeches, where similar phrases are being repeated throughout the texts. Despite there are more occurrences in the Portuguese side, there is more repetition in the English side, since several terms or phrases in English that are highly repetitive, can be translated differently in Portuguese. If we go back to the behavior in Word-based Suffix Arrays, the list of occurrences of a phrase is found all at once, with two binary searches being performed over the Suffix Array. For the hierarchical phrase-based search, the higher is the number of occurrences in the list, the more time consuming is the procedure to match the gaps, which is what happens in Europarl. In Byte-codes Wavelet Trees this is not visible, since the procedure is more agnostic of that fact, being more dependent on the number of occurrences overall, instead of the number of occurrences of a particular phrase.

Figures 5.12 and 5.13 show a variation of the search time response of the Framework, for the limit window of two, three, four and five fine segments, in English and Portuguese respectively. These results were measured for Europarl, DGT and Eurlex, using Arrays of Integers for the Alignment Layer.

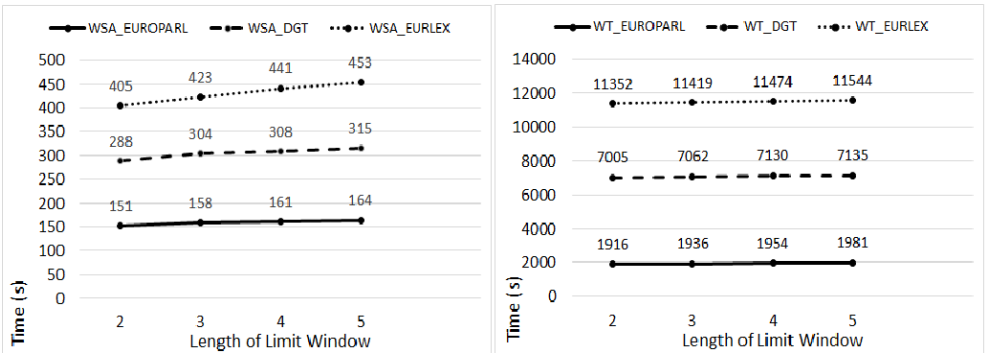

Figure 5.12: Hierarchical phrase-based search time response with Arrays of Integers supporting the Alignment Layer, in English, for multiple gap limit windows

The variations of time with the changes of the maximum gap limit window follow an ascending order, from the smaller window to the larger one. It is not surprising that for all the corpora, and both implementations of the Text Layer, the search time increases with the number of fine segments where the gap can be matched, as it leads to more occurrences processed and more

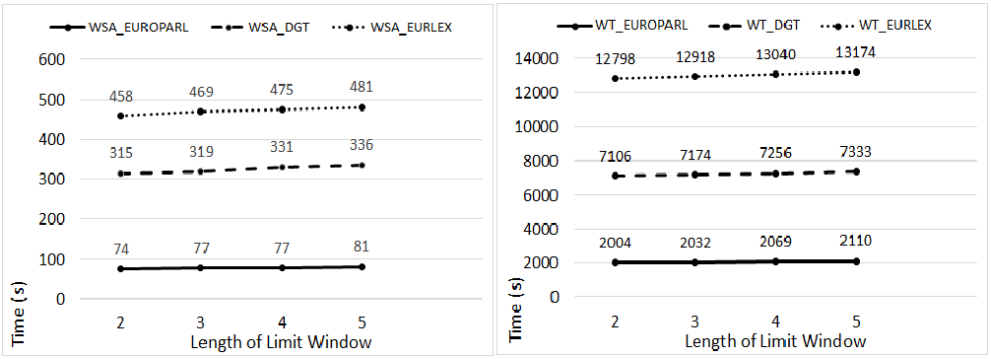

Figure 5.13: Hierarchical phrase-based search time response with Arrays of Integers supporting the Alignment Layer, in Portuguese, for multiple gap limit windows

verifications done. However, in both implementations of the Text Layer, the difference is not that big, which shows that any of the approaches to match hierarchical phrases, is not heavily influenced by the maximum length.

For Word-based Suffix Arrays, the window where the text is queried to determine if a literal appears before, or after, a match increases with the size of the window. In Byte-codes Wavelet Trees, the same thing also happens with the LCS window. However, it is still the procedure as a whole which is time consuming, thus making the increase of the limit window a factor with considerable less impact.

Figures 5.14 and 5.15 show the same search time variation, for the same corpora, but using Bitmaps to support the Alignment Layer.

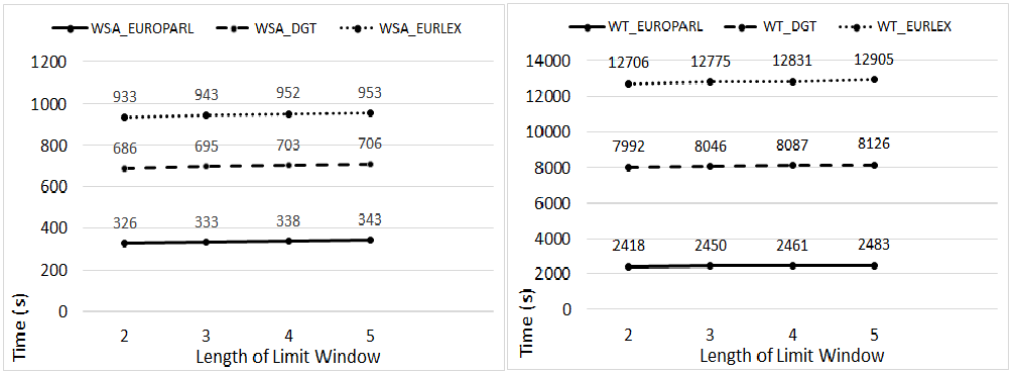

Figure 5.14: Hierarchical phrase-based search time response with Bitmaps supporting the Alignment Layer, in English, for multiple gap limit windows

With Bitmaps supporting the Alignment Layer, the results follow the same ascending variation, for all corpora, and both approaches for the Text Layer, just like with Arrays of Integers. Again, the difference in the length of the limit window does not have a big impact on the time response, since the variation is not considerable.

Figure 5.16 shows the number of occurrences found for all the experiments presented in the previous charts. The number of occurrences are presented in thousands, with the ones found in the English side presented on the chart on the left, and the ones found in the Portuguese side

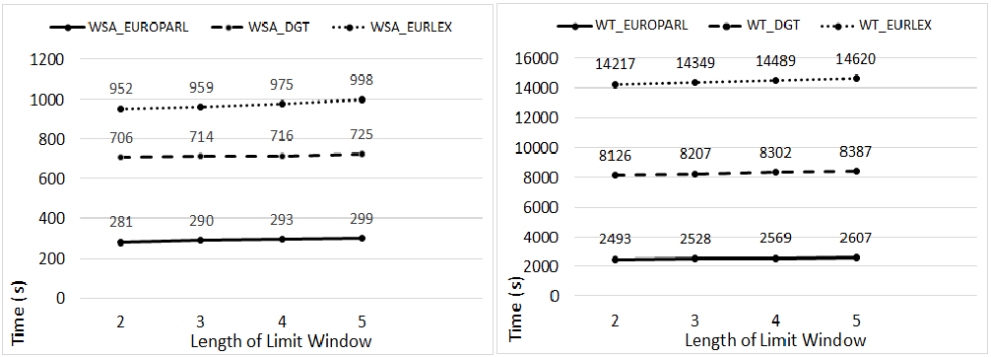

Figure 5.15: Hierarchical phrase-based search time response with Bitmaps supporting the Alignment Layer, in Portuguese, for multiple gap limit windows

presented on the chart on the right.

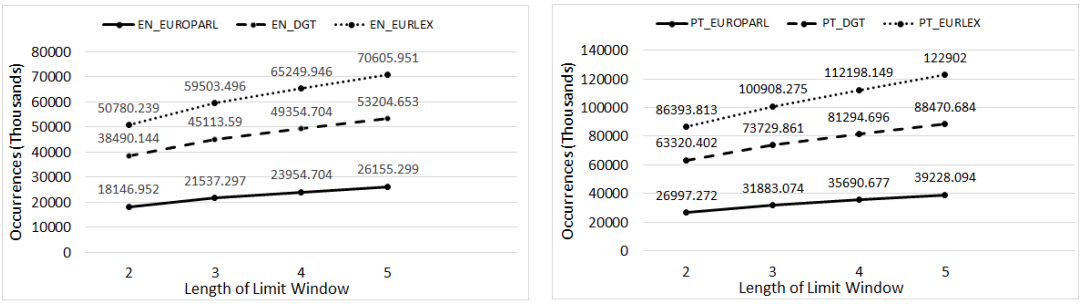

Figure 5.16: Number of occurrences found in the hierarchical phrase-based searches, for multiple gap limit windows

As expected, with the increase of the limit window, the number of occurrences returned is bigger, for the same number of searches made. That happens since with a bigger window, the possibilities for matching the literals also increases, thus also increasing the number of occurrences returned. At this point, we are not considered if the content of the gap has actually any statistical value, which would be an important influence for the size of the window, but just analyzing the effect of the window size.

Comparing these results with the number of occurrences returned by the monolingual phrase-based search, it is interesting to see that the phrase-based finds more occurrences. For instance, in the phrase-based search, 700000 searches for Eurlex, in English, returned around 595 million occurrences, while here, 194260 searches returned more than 70 million occurrences. This means that the phrase-based monolingual search has an average of 850 occurrences per search, while the hierarchical phrase-based shows an average of 360. Because of the gaps, one could assume that there would be more occurrences of the terms. However it is important to consider that the phrase-based search includes very frequent words such as `the`, `a` or `of`. A hierarchical phrase is a more complex language construction, thus the number of highly frequent terms, such as the ones mentioned, are considerably more rare.

Nevertheless, the number of occurrences found is still high. And since the hierarchical phrase-based search is considerably slower than the phrase-based, skipping occurrences in the

bilingual search can be even more relevant. As explained earlier, the skip-based search procedure has the same behavior, regardless if the term is phrase-based or if it has gaps. Table 5.3 shows the difference between the number of occurrences found (Total Occs), in total, against the ones processed in the bilingual search (Occs Processed), alongside the co-occurrences found (Co-Occs), this time in millions.

It is important to mention here, that the monolingual search numbers presented in this Section, refer only to unique terms. This means that no term is searched twice. However, in a bilingual search, since a term can be translated differently in the other language, there are some pairs which repeat one term, in one language. This way, we measured the number of occurrences for all the 430355 terms on both sides of the pairs, regardless if they are repeated or not, to get the total number of occurrences processed. Next to the number of total number of occurrences and the number of occurrences processed in the bilingual search, Table 5.3 shows the reduction ratio. This ratio considers the number of occurrences reduced from the total occurrences to the ones processed by the bilingual search.

Table 5.3: Number of occurrences and co-occurrences found, in millions, using the skip-based bilingual search, on hierarchical phrases.

Corpora	**Total Occs**		**Occs Processed**		**Reduction Ratio**		**Co-Occs**
	EN	**PT**	**EN**	**PT**	**EN**	**PT**	
EUconst	7.747	11.463	3.045	5.439	61%	53%	1.230
EMEA	507.986	883.105	173.871	231.535	66%	74%	43.775
Europarl	1667.872	2538.775	482.104	596.977	71%	76%	172.246
DGT	3137.428	5232.395	915.151	1218.978	71%	77%	416.390
Eurlex	4205.043	8366.443	2080.524	2838.349	51%	66%	1180.801

The reduction ratios in Table 5.3 show how important is the skip-based approach for the bilingual search, considering the number of occurrences saved. For DGT and Europarl, the reduction ratio, either in Portuguese or English, is bigger than 70%, which shows how impactful the skip-based strategy is. Considering that the terms can have billions of occurrences, as seen in Table 5.3, and that the hierarchical phrase-based search, this reduction is even more relevant.

Surprisingly, with the larger corpora, Eurlex, the reduction ratio is smaller. In English is even smaller than with Euconst, while in Portuguese, the reduction ratio is smaller than with EMEA, Europarl and DGT. This does not mean that the skip-based search does not have a big impact for Eurlex, since a 50% of reduction ratio is extremely relevant, but being the larger corpora, the ambiguity was expected to be higher. However, 49% of the English occurrences need to be processed to find all the co-occurrences of the pairs for Eurlex, meaning that the ambiguity, for these pairs, is not as higher in Eurlex.

Table 5.4 shows the search time results for the 430355 pairs for the four combinations of data structures that support the Bilingual Framework. The Table does not display any comparison with the phrase-based bilingual search results, because the testing data is different.

By analyzing the results of the hierarchical phrase-based search, it is clear how much more

Table 5.4: Bilingual search time results for the hierarchical phrase-based search procedure

Corpora	Time WSA (s)		Time BOWT (s)	
	ArrInt	Bmap	ArrInt	Bmap
EUconst	15	24	23	36
EMEA	507	1148	1748	2775
Europarl	1634	3586	8291	11504
DGT	3021	7535	26663	32902
Eurlex	14269	26255	48290	55216

time consuming is this process, when compared with the results from the phrase-based search, presented in Tables 4.4 and 4.6. For Eurlex, considering the fastest approach, based on WSAs + ArrInts, the hierarchical phrase-based search takes around 56x (14269 / 256) more time than the phrase-based, for less searches.

It is important to keep in mind that the phrase-based and hierarchical phrase-based searches were experimented with different search pairs, thus the comparison is not fair. Nevertheless, this gives an idea of how heavier is the hierarchical phrase-based search, not only because of the complexity of the algorithm, but also because of the number of occurrences found. This is visible in Table 5.3, with the number of occurrences processed being considerably higher than with the phrase-based search, even considering the different testing data.

With Byte-codes Wavelet Trees, it is interesting to see that the difference against phrase-based search is much lower. Again for Eurlex, using ArrInts for the Alignment Layer, the hierarchical phrase-based search takes around 3x more time than the phrase-based search (48290 / 16471). This reinforces the idea that the major influence for the search time response in this approach is still the index itself, and not as much the search procedure. Since the data is compressed, the several necessary *ascends* and *descends* in the tree structure to decode it, has the major impact on the phrase-based and hierarchical phrase-based search.

Considering this heavier search procedure, the skip-based approach is extremely important, as demonstrated earlier in this chapter, due to the high number of occurrences that were skipped, to return the same number of occurrences. Considering the high number of occurrences that phrases with gaps may have, it is then decisive to use the skip-based search to avoid having bilingual search times even higher.

These results do not influence the memory consumption of the Framework. Some additional structures could be used to improve the search time response for the hierarchical phrase-based search, but that option would increase the space requirements of the Framework.

CHAPTER 6

RELATED WORK

In the previous Chapters, I described the Bilingual Framework implementation, with all its Layers and search procedures, and the algorithms that support them. During this research, several data structures and different alternatives were studied to understand the best directions for the work being developed.

In this Chapter, I describe some relevant work to the research made in the context of this thesis. First, I describe the data structures and compressed techniques studied, from the character-based data structures to the word-based approaches. Then, I introduce some important and relevant work in Machine Translation, using similar data structures, or other forms of word-based indexing such as inverted indexes.

On the course of this research, the first data structures studied were character-based indexes, which were ultimately not used in the Framework. However, they were a crucial part of the path that guided the research to the Word-based strategy, in particular because it followed the character-based approach of the Master's Thesis [36, 37], based on Suffix Trees and Suffix Arrays.

6.1 Compressed Full-Text Indexing

Full-text indexing enables fast search for small sequences of texts over larger collections of texts, such as parallel corpora. Two examples of such structures are Suffix Arrays [125] and Suffix Trees [127, 181]. Suffix Arrays have been commonly used in Statistical Machine Translation [27, 28, 49, 67, 116, 129, 189], while Suffix Trees have being used often in Bioinformatics [44, 81, 86].

However, as mentioned already in this document, these indexes require a large amount of space in memory, which for huge text collections, with sizes in the order of the Gigabyte or more, may become too restrictive. Using these structures may lead to using disk, which is considerably slower than accessing main memory, despite their considerably larger sizes [140].

Given the relation between primary and secondary access times, it is preferable to store

huge amounts of text in main memory, in its compressed form, to reduce the disk transfer time to a minimum [140, 141]. Several proposals were made over the years to create space efficient indexing strategies, for representing such large texts in main memory, resulting in succinct indexes [84, 111, 120], compressed indexes [58, 59, 76, 156, 157, 159] and self-indexes [1, 48, 54, 65]. The most commonly used data structure on these compression studies was the Suffix Array.

Before presenting some of the studied compressed indexes, the following subsections introduce some of the basic notions and techniques that led to the development of index compression, in particular for the case of Suffix Arrays.

6.1.1 Sampling a Suffix Array

A responsible for the large space requirements of full-text indexes, such as Suffix Arrays, lies on the necessity of having the index and the text in main memory. The main technique to avoid this scenario is by sampling the index, at regular text position intervals, instead of indexing all the positions of the Suffix Array.

However, there are some things that we need to ensure, namely how to find all the text positions, $A[i]$, in Suffix Array A, since only a sample of $A[i]$ entries are represented in main memory. This problem is solved using a technique called backward search [54].

The backward search is a different approach for searching a pattern, from the last character to the first, with a backward step (LF) being defined as $LF(i) = i'$, with $A[i'] = A[i] - 1$, with i and i' being positions of the Suffix Array. Figure 6.1 shows an example of a Suffix Array, for the sequence `abracadabra$`, with a representation of the backward search for `abr`.

Backward Step 1: "r"

i	A[i]	TA[i]-1	suffix TA[i],n
1:	12	a	$
2:	11	r	a$
3:	8	d	abra$
4:	1	$	abracadabra$
5:	4	r	acadabra$
6:	6	c	adabra$
7:	9	a	bra$
8:	2	a	bracadabra$
9:	5	a	cadabra$
10:	7	a	dabra$
11:	10	b	ra$
12:	3	b	racadabra$

Backward Step 2: "br"

i	A[i]	TA[i]-1	suffix TA[i],n
1:	12	a	$
2:	11	r	a$
3:	8	d	abra$
4:	1	$	abracadabra$
5:	4	r	acadabra$
6:	6	c	adabra$
7:	9	a	bra$
8:	2	a	bracadabra$
9:	5	a	cadabra$
10:	7	a	dabra$
11:	10	b	ra$
12:	3	b	racadabra$

Backward Step 3: "abr"

i	A[i]	TA[i]-1	suffix TA[i],n
1:	12	a	$
2:	11	r	a$
3:	8	d	abra$
4:	1	$	abracadabra$
5:	4	r	acadabra$
6:	6	c	adabra$
7:	9	a	bra$
8:	2	a	bracadabra$
9:	5	a	cadabra$
10:	7	a	dabra$
11:	10	b	ra$
12:	3	b	racadabra$

Figure 6.1: Suffix Array for sequence `abracadabra$`, with the backward search for the sequence `abr`

In Figure 6.1, the first step lies on determining all the suffixes in the Array that start with `r`, which appear at positions $i = [11, 12]$. This is determined with the help of an array C, indexed by the characters in the text, $C[S_i]$, which holds the number of characters in the text lexicographically smaller than the character S_i. In this example, $C[r] = 10$, thus the interval of the suffixes

starting by r starts at $C[r]+1$. Since r is the lexicographically larger character, the end of the interval corresponds to the end of the Suffix Array.

To find br, it is important to look for the column $T_{A[i]-1}$, in Figure 6.1. At position $i = 11$, $T_{A[i]-1}$ = b, meaning that the character that precedes the suffix is b. This means that since $A[11,12]$ represents the suffixes that start with r, then the suffixes that start with br must appear in $A[11]-1$ and $A[12]-1$.

By analyzing Figure 6.1, we can see that all the suffixes that precede the ones that start with r, start with the character b, which are the ones we are looking for. To calculate that, we need to determine $C[b] = 6$ and how many occurrences of b appear before $i = 11$ and after $i = 12$, in the column $T_{A[i]-1}$. Those numbers are 0 and 2, meaning that the suffixes starting with b, before r, appear at positions $C[b]+1+0$ and $C[b]+2$, thus $[7,8]$. The same procedure is repeated for the character a.

The step presented above is denominated *LF*. Considering the column $T_{A[i]-1}$ as the string $S1,n$, this step can be generalized as: $LF(i) = C[S_i] + Occ(S_i, i)$, where $Occ(S_i, i)$ is the query to determine the number of occurrences of the character S_i, until position i of S. To compute this query, we can use bitmaps B^c, where $B^c[i] = 1$, if and only if $S_i = c$, for every character c, and then apply the $rank$ function on top of them. The $rank$ computation can be done efficiently using Wavelet Trees [78].

Considering this backward search operation, to find the text position correspondent to $A[i]$, if i is not sampled, several backward searches can be performed, $A[i]-1$, $A[i]-2$, etc. This process is repeated until finding a sampling position $pos = A[i]-k$. This then points to a position of the text indexed by the Suffix Array, thus the answer to the search would be $pos+k$, with k being the number of backward steps made. Going back to Figure 6.1, considering that we wanted to find $A[12]$, which was not sampled, the procedure would perform $LF(12) = 8$. If $i = 8$ was not sampled as well, we would continue with $LF(8) = 4$. Considering that position as sampled, the result for $A[12]$ would be $A[4]+2 = 3$.

The choice for how many positions are sampled is a tradeoff for space and time, since more positions sampled makes the index faster, while less positions sampled makes the index smaller. This way, the data structure becomes a self-index, since a Wavelet Tree to compute the $rank$ and some auxiliary arrays are sufficient. The Suffix Array and the text are not needed. However, this does not make the Suffix Array a compressed index yet.

6.1.2 Compressing the Suffix Array

Another important technique for compression is the inverse of function $LF(i)$, denoted Ψ, which enables scanning the text from left to right [76, 157], from suffix $T_{A[i]}$ to suffix $T_{A[i]+1}$. Function $\Psi : [1,n] \rightarrow [1,n]$ is defined so that, for all $1 \leq i \leq n$, $A[\Psi(i)] = A[i]+1$, excepting for $A[1] = n$ where $A[\Psi(1)] = 1$, with n as the size of the text.

Figure 6.2 shows a representation of a Suffix Array for text abracadabra$, with its Ψ values. In this example, $\Psi(3) = 7$, as the suffix following abra$ is bra$, located at position $i = 7$ of Suffix Array A.

i	A[i]	suffix $T_{A[i],n}$	Ψ
1:	12	$	4
2:	11	a$	1
3:	8	abra$	7
4:	1	abracadabra$	8
5:	4	acadabra$	9
6:	6	adabra$	10
7:	9	bra$	11
8:	2	bracadabra$	12
9:	5	cadabra$	6
10:	7	dabra$	3
11:	10	ra$	2
12:	3	racadabra$	5

Figure 6.2: Suffix Array with Ψ values for sequence `abracadabra$`

The function Ψ matches a subsequence in a larger sequence of text, by being used recursively: i, $\Psi[i]$, $\Psi[\Psi[i]]$, etc, which point to $T_{A[i]}$, $T_{A[i]+1}$, $T_{A[i]+2}$, etc. To find the characters where these Ψ values point to, the $rank$ function can again be used. Since we do not have the text directly accessible, the procedure builds bitmaps where each 1 represents that a suffix in A changed the first character. In the example of Figure 6.2, the bitmap would be $B = 110000101110$, and the character pointed by A[i] would be $rank_1(B, i)$ [140].

Such integer sequences can be compressed using gap encoding methods, which is extremely important for compressing the indexes, and are commonly used for compressing inverted indexes [182]. Some compressed Suffix Arrays are also based on this technique [76, 120].

6.1.3 Self-Repetitions

Going back to Figure 6.2, it is possible to determine that $A[8] = A[4] + 1$, $A[9] = A[5] + 1$ and $A[10] = A[6] + 1$, thus $A[8,10] = A[4,6] + 1$, where $A[4,6]$ represents suffixes only started with `a` and $A[8,10]$ represents the same suffixes without that first character `a`. This regularity means that for each pair of consecutive suffixes `aX` and `aY` in $A[4,6]$, the suffixes `X` and `Y` are contiguous in $A[8,10]$. This leads to the definition of self-repetition.

A self-repetition is an interval $[i, i+l]$ of $[1,n]$, such that it exists another interval $[j, j+l]$, that satisfies $A[j+r] = A[i+r] + 1$ for all $0 \leq r \leq l$ [120, 121]. The regularity in a Suffix Array is given by the number of self-repetitions needed to cover the whole Array [122, 123], which is an extremely important factor to the compressibility of Suffix Arrays for a given text.

Function Ψ is used in most Compressed Suffix Arrays, as there are several properties of this function that enables compression. The first one, as mentioned before, states that Ψ monotonically increases in the areas of the Suffix Array that have suffixes that start with the same character [76]. Thus, $\Psi(i) < \Psi(i+1)$, whenever $T_{A[i]} = T_{A[i+1]}$. This means that when Ψ values are listed from $i = 1$ to $i = n$, it is easy to find increasing sequences of integers, as Figure 6.2 demonstrates with surrounding rectangles.

Then, inside a self-repetition $A[j+r] = A[i+r] + 1$, $\Psi(i+r) = j+r$ throughout the interval. In Figure 6.2, consider the interval $A[7,10] = [A[3]+1, A[4]+1, A[5]+1, A[6]+1]$. If we analyze the

column with the Ψ values, we get $\Psi(3) = 7$, $\Psi(4) = 8$, $\Psi(5) = 9$ and $\Psi(6) = 10$, thus demonstrating the validity of this rule.

This leads to the definition of runs. A run in Ψ is the maximal interval $[i, i+l]$ in Ψ where $\Psi(i+r+1) - \Psi(i+r) = 1$, for every $0 \leq r < l$ [122, 123]. This concludes that the number of self-repetitions that covers a Suffix Array is the number of runs in its Ψ function.

All these techniques and properties of the Suffix Array enables compressing the data structure itself, but also avoid the need of having the full representation of the text in main memory, leading also to the notion of self-indexes. Next, I present the initial compressed data structures studied, Compressed Suffix Arrays and Compressed Suffix Trees, to support the proposed Bilingual Framework.

6.1.4 Compressed Suffix Arrays

Compressed Suffix Arrays (CSA) are data structures that reduce the space consumption of standard Suffix Arrays. While standard Suffix Arrays need $n\ log\ n$ bits, where n is the text size, this approach only requires $O(n\ log(|A|))$ bits, with $|A|$ as the alphabet size and n much larger than $|A|$. In other hand, the access time increases from constant time to $O(log^{\varepsilon}(n))$, with $0 \leq \varepsilon \leq 1$.

Over the years, several approaches were developed for Compressed Suffix Arrays, using different techniques and strategies, to take advantage of the Suffix Array regularities mentioned above [54, 57, 60, 75]. Compressed Suffix Arrays were first introduced by Mäkinen [120, 121] (Mak-CSA) and Grossi and Vitter [76, 77] (GV-CSA), both with different mechanisms for exploring the Suffix Arrays regularities.

MAK-CSA is a minimized array, where self-repetitions $A[i, i+l]$ are replaced by a link to the corresponding area $A[j, j+l]$, where $A[j+r] = A[i+r] + 1$ for all $0 \leq r \leq l$. This makes this approach the first compressed full-text index created.

6.1.4.1 GV-CSA

GV-CSA is a succinct index that provides access to the Suffix Array, $A[i]$, without storing it, keeping the main search algorithm of searching over Suffix Arrays, based on binary searches. For that, the text T is maintained explicitly in main memory. This approach for Compressed Suffix Arrays uses a hierarchical decomposition of the Suffix Array based on the function Ψ.

First, bit array $B^0[1, n]$ is defined as $B^0[i] = 1$ if and only if $A[i]$ is even. Consider $\Psi_0[1, n/2]$ to contain the values $\Psi(i)$ for arguments i where $B^0[i] = 0$. Also consider that $A_1[1, n/2]$ is the subsequence of $A_0[1, n]$ with the even $A_0[i]$ values, divided by 2. Then, A can be represented using only Ψ_0, B^0 and A_1 (see Figure 6.3).

To get $A[i]$, first check if $B^0[i] = 1$. If so, then $A[i]$ is in A_1, divided by two. To know the exact position, we have to know how many 1's are in B^0 from the beginning to position i, using *rank*. Then, we apply the following formula: $A[i] = 2 \times A_1[rank_1(B^0, i)]$. If $B^0[i] = 0$, then $A[i]$ is odd and is not represented in A_1. However, $A[i] + 1 = A[\Psi(i)]$ is even and is represented in A_1. Thus, $A[i] = A[\Psi(i)] - 1$, where $A[\Psi(i)] = A[\Psi_0[rank_0(B^0, i)]]$, since Ψ_0 collects only the Ψ values where $B^0[i] = 0$.

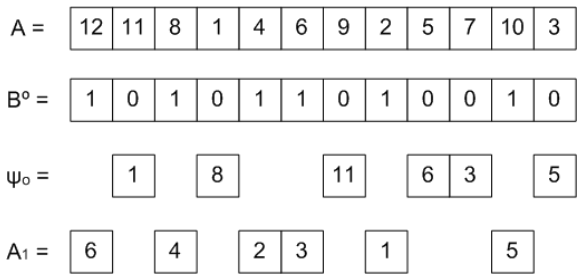

Figure 6.3: Structure of the first hierarchical level of GV-CSA for `abracadabra$`

Figure 6.3 shows an example of this algorithm. To obtain $A[9]$, $B^0[9] = 0$, so it is not represented in A_1. Thus, to find $A[9]$, we have to determine $\Psi_0(9)$, which is $\Psi_0[rank_0(B^0,9)] = \Psi_0[4] = 6$. Now we need $A[6]$, which leads to $B^0[6] = 1$ and $A_1[rank_1(B^0,6)] = A_1[4] = 3$. Thus, $A[6] = 2 \times 3 = 6$ and $A[9] = A[6] - 1 = 5$.

This idea can be used recursively. Instead of representing A_1, we can have B^2, Ψ_2 and A_2, until A_h is small enough to be represented explicitly. Typically $h = [log\ log\ n]$, to use $O(log\ n)$ bits for each of the $n/2^h$ entries of A_h. The B^l arrays add $2n$ bits and the additional rank structures $O(n)$ bits. The problem to be solved is to represent the Ψ_l arrays.

Both these approaches for Compressed Suffix Arrays are not self-indexes, as both explicitly maintain the text T in memory. The following approaches solve this problem, transforming the Compressed Suffix Array to a self-index.

6.1.4.2 Sadakane's Compressed Suffix Array (SAD-CSA)

Sadakane [60, 157] converted Grossi and Vitter's approach into a self-index and optimized it, by representing the Array and the text using the full function Ψ and auxiliary structures.

Sad-CSA gives access to any prefix of $T_{A[i],n}$, instead of access to $A[i]$. To represent A and T, this approach needs the Ψ function and some extra bit vectors. To find a pattern P in T, it is used the vector F, represented in Figure 6.4. Vector F consists of the first characters of all the suffixes in lexicographic order. $T_{A[i]}$ is the first character of the suffix pointed by $A[i]$, so $T_{A[i]} = F_i$. Then, $T_{A[i]+1} = T_{A[\Psi(i)]}$. Using this method, we can continue extracting further characters, until the matching for P is over.

To find $T_{A[i]}$ efficiently, the procedure uses a bit vector $newF_{1,n}$, so that $newF_{1,n} = 1$ if $i = 1$ or $F_1 \neq F_{i-1}$, and a sequence of characters, $charT$, where the distinct characters of T are concatenated following an alphabetic order. Then, $T_{A[i]} = charT[rank_1(newF,i)]$, which can be computed in constant time.

An example is shown in Figure 6.4, with all the mentioned structures represented for the text `abracadabra$`. To obtain $T_{A[8],n}$, we compute the value of $charT[rank_1(newF,8)] = charT[3] =$ `b`. Then, $\Psi(8) = 12$, which means that the second character of the pattern to be found is $charT[rank_1(newF,12)] = charT[6] =$ `r`. We continue the process to $\Psi(12) = 5$, which leads to the third character being $charT[rank_1(newF,5)] = charT[2] =$ `a`, and so on. Until this point we found `bra`. If we continued the example until n, we would be able to find `bra$`, the

8th suffix of `abracadabra$`, namely $T_{A[8],n}$.

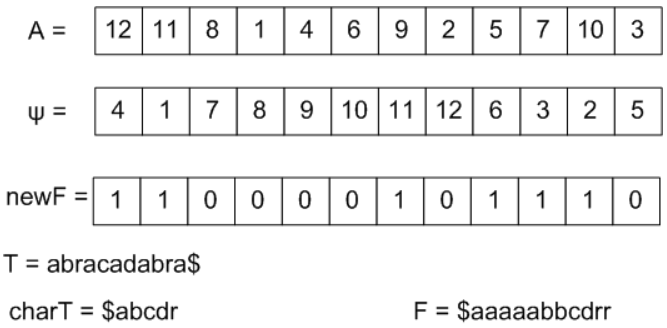

Figure 6.4: Main components of Sad-CSA structure for `abracadabra$`

So far, the CSA uses $n+O(n)+\sigma\ log\sigma$ bits of space for the two previously introduced vectors ($newF$ and $charT$), plus the Ψ representation. However, Sad-CSA does not give direct access to $A[i]$, thus needing additional structures for solving a location query. For that matter, Sadakane used the Grossi and Vitter hierarchical structure, without using the text, and with Ψ instead of Ψ_0. Grossi and Vitter later arranged a slightly different solution [79], using as well the Ψ increasing nature, which is not relevant to this thesis, but which required even less space than the previous GV-CSA implementation and than SAD-CST.

6.1.5 Compressed Suffix Trees

In terms of space, indexing a text in a Suffix Tree is worse than indexing only the text in memory. The text occupies $n\ log(|\alpha|)$ bits, while the Tree occupies $n\ log(n)$ [159], with α as the alphabet and n as the text size, and with α being much smaller than n. A naive implementation of Suffix Trees requires 20 bytes per character, while an optimized one takes 10 bytes [110].

Suffix Arrays reduce the space requirements, but they cannot carry out all the Suffix Trees functionalities. Enhanced Suffix Arrays [2] recover the suffix trees full functionality, which leads to the basis notion of Compressed Suffix Trees (CST). Since Compressed Suffix Arrays support operations such as $A[i]$, $\Psi[i]$, a Compressed Suffix Tree is in fact a Compressed Suffix Array, with some additional content to encode the tree topology, such as the nodes and leafs, and the longest common prefix (LCP) information.

Some approaches were made for CSTs [77, 137], but important characteristics of the classical Suffix Trees were missing, like suffix links between internal nodes, internal node depths and the lowest common ancestor (*lca*) between two nodes. The first proposal of CST with full functionality was made by Sadakane (Sad-CST) [159] and was followed by several important proposals [62, 63, 148, 156].

Every single one of these alternatives have different space performances, leading as well to different search time performances. The most common thread with all these approaches, besides reducing considerably the space consumption, occupying less space than the text size, is that the most compressed is the solution, the slower it gets responding to searches.

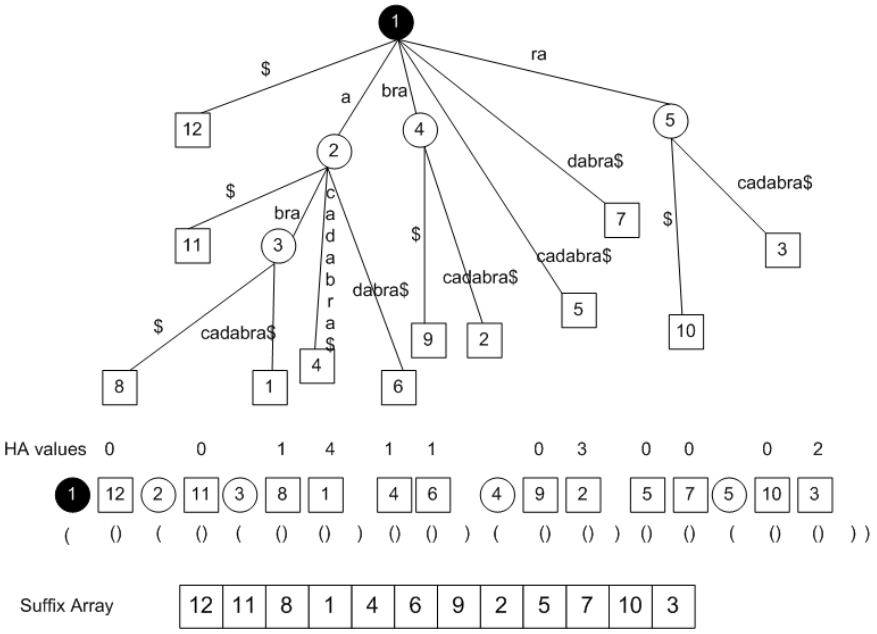

Figure 6.5: Parenthesis representation and HA values for `abracadabra$`

6.1.5.1 Sad-CST

Using Compressed Suffix Arrays, Sadakane achieved a solution with a size of $|CSA| + 6n + o(n)$ bits, where $|CSA|$ is the size of the Compressed Suffix Array. In terms of time complexity, this implementation supports the efficient operations of Suffix Trees, with only a slowdown of $polylog(n)$ [1] [159].

Besides CSAs, Sadakane used a balanced parenthesis encoding of a tree [136, 137] and an Height Array [97]. The balanced trees can be encoded in $2m + O(m)$ bits, with m as the number of nodes. In a preorder transversal of the tree, an open parenthesis is created for the first time the node is visited. Then, the subtrees of the node are traversed, writing a close parenthesis. Thus, every node is represented by a pair of parenthesis, more precisely, by the opening parenthesis since it is possible to find the closing parenthesis in constant time [136]. As a Suffix Tree has $2n-1$ nodes, n leaves and $n-1$ internal levels at most, the balanced tree can be encoded in $4n + O(n)$ bits [159].

The Height Array (HA) stores a set of lengths of LCP information between two consecutive suffixes, or two consecutive leaf labels. With this auxiliary structure and the Compressed Suffix Array, it is possible to simulate a bottom-up traversal of a Suffix Tree [97].

Figure 6.5 shows an example of HA values and the parenthesis representation. The closing parenthesis of the root, node 1 in Figure 6.5, only appears at the end, after all the nodes (which are all descendants of the root) are visited. The closing parenthesis of node 2 comes after node 3 and leafs 11, 8, 1, 4 and 6 are visited. The HA values are determined using the LCP value between a certain suffix and the previous one. For instance, suffix 9, in position seven of the

[1] A polylogarithmic function in n is a polynomial in the logarithm of n, meaning $O(x^c)$, thus growing slower than any positive exponent.

CSA, and suffix 6 have no common prefix, so $HA[7] = 0$. However, suffix 2, in position eight of the CSA, and suffix 9 have `bra` as a common prefix, thus $HA[8] = 3$.

To represent the LCP information, it is necessary to define data structures for HA, such that $HA[i]$ can be computed in constant time. This is possible with a bit vector of $2n + O(n)$ bits, leading to a total storing of $|A| + 2n + O(n)$ bits [159]. Adding the tree encoding, the length of the LCP between suffixes $T_{A[i]}$ and $T_{A[j]}$, for any i and j, can be computed in $O(t_A)$, with t_A as the time to compute an element in the Suffix Array, using a structure of size $|A| + 6n + O(n)$ bits. The values of LCP can be extracted from HA using the *rank* and *select* functions.

6.1.5.2 Fully-compressed Suffix Tree

Russo et. al. [156] broke the $O(n)$ extra-bits barrier of Sadakane's approach, representing a Suffix Tree on top of a CSA based on FM-Index [59], with all the navigational operations of Suffix Trees, needing only a sublinear extra space, namely $O(n\ log(|A|))$ extra bits. In this proposal, they got rid of the parenthesis, using instead the Suffix Array interval corresponding to the Suffix Tree node. In Figure 6.5, node 3 would be represented by the interval $A[3,4]$, which corresponds to suffix 8 and suffix 1, its corresponding children.

Russo et. al. used a special kind of sampling of Suffix Tree nodes, which was pioneered in sparse Suffix Trees [95], based on Lempel-Ziv parsing [1, 58, 111, 139]. Sampling some suffix tree nodes, most operations in this fully compressed suffix tree (FCST) approach are defined by using suffix links towards a sampled node, retrieve its information, and transforming it as returning to the original node, avoiding storing all suffix tree nodes.

Despite the improving in space, Sadakane's representation is faster for most operations, except the important operation to get the child of a node [156]. Additionally, a fully Compressed Suffix Tree must be built from an uncompressed one, which limits their applicability, due to the need of a large main memory available (about 10 times larger), before ending with a fully compressed representation.

Following this FCST approach, Fischer et. al. (FMN-CST) [63] proved that the HA array in Sad-CST is compressible, while supporting constant-time *rank* and *select*. Following the previous idea of eliminating the parenthesis and using Suffix Array intervals, we can obtain a more space efficient CST than Sadakane's approach. For navigating the tree without its balanced tree representation, Fischer et. al stated that three operations are needed: $RMQ(i, j)$ that returns the position of the minimum in $LCP[i, j]$; $PSV(i)$ that returns the last value smaller than $LCP[i]$ in $LCP[1, i-1]$; $NSV(i)$ that returns the first value smaller than $LCP[i]$ in $LCP[i+1, n]$, which can be computed in constant time, with additional $O(n)$ bits on top of the LCP representation.

Cánovas et. al. [29] implemented this CST, where two main problems were tackled: represent LCP efficiently, inspired on the succinct tree representation [161] by Sadakane and Navarro, and compute efficiently the operations RMQ, PSV and NSV over the LCP representation. This resulted in a CST faster than SAD-CST in some operations, with FMN-CST space performance.

6.1.5.3 CST++

In the work developed by Russo et. al. [156] and Fischer et. al. [63] the space consumption of Compressed Suffix Trees is $O(n)$, while in the approach made by Ohlebusch et. al. [148] (OG-CST) the space consumption is $2n$. Most of its operations, like string depth of a node, parent of a node, suffix link, among others, have temporal complexity superior to 1, which was achieved by Sadakane [159] in most of them, except for child, string depth and leaf label.

Ohlebusch et. al. [149] proposed another approach for Compressed Suffix Trees, which has a space consumption of $3n$, but has the same time efficiency as Sadakane's approach. The approach is based on OG-CST, with the use of an enhanced balanced parentheses representation (BPR), similar to the one in Figure 6.5, for the operations RMQ, PSV and NSV. The BPR leads to a construction algorithm just based on the Suffix Array, like stated by Ohlebusch and Gog [148], but their approach lacks some information, like treating the right child and the right sibling the same way. The enhanced BPR [149] solves this problem, by using an additional structure of length $n+2$, which leads to an additional space consumption $(3n)$. However, this enhanced BPR helps reaching a better time efficiency on most of the operations, while having a better space consumption than Sadakane's approach, which needs up to $4n+O(n)$ bits.

6.2 Word-based Compressed Indexes

All these approaches for CSAs and CSTs solved the space problem of its respective classical structures, while attempting to obtain the least slowdown possible, which is an inevitable tradeoff of the compressed indexes. However, all these indexes have the character as main unit. In many situations, in particular with natural language, full-text indexing can be too much for what is actually necessary, causing negative impacts on query time response and space performance.

In many natural languages, the words are separated by spaces or punctuation signs. Even in languages where that does not happen, such as Mandarin, there are parsers that can create a separator between words. Thus, indexing every position of the text is not necessary, since we can use the word boundaries.

Since the proposed Bilingual Framework has the goal of indexing natural language, these character-based indexes are not the most adequate to represent it, despite being very powerful. This way, we guided the research to data structures whose main unit is the word, such as the Word-based Suffix Arrays and the Byte-codes Wavelet Tree. Besides that, other indexes were studied and considered during the research that led to the Bilingual Framework.

6.2.1 Inverted Indexes

An inverted index is a word-based data structure that indexes collections of texts, with the main objective of speeding searches over the words in the text. The structure of the index is based on a vocabulary of words, with every unique word in the text, and their list of occurrences in the indexed text. Figure 6.6 shows a simple example for the text `to be or not to be`.

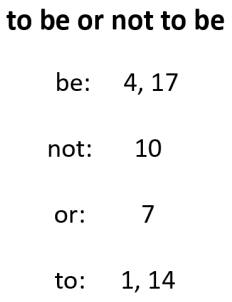

Figure 6.6: Inverted Index for the text `to be or not to be`

Each word in the vocabulary, $\Sigma = \{\texttt{be}, \texttt{not}, \texttt{or}, \texttt{to}\}$, has the list of the character-based positions of its occurrences in the text, in an ascending order. To find the occurrences of a word, it is just a matter of finding the words in the vocabulary, using binary searches, and get its list of occurrences. These data structures are commonly used in search engine indexing, with the index being build with the list of documents per word.

Compression techniques have also been applied to inverted indices, encoding each occurrences list of the index using gap encodings [50, 182], which represent with bits the differences between adjacent elements in the list. The text on the index can be compressed as well, using for instance Huffman coding [85], with every word being represented by bits.

To save extra space, the text can be divided into blocks, with the posting lists pointing to the blocks where the word appears, resulting in the block-addressing inverted index [9]. This saves space, since there will be less blocks than text positions. On top of this, this representation can be compressed using gap encoding methods, since the list of occurrences appears in an increasing order [142]. To find the occurrences of a word, a search is made over the vocabulary of words, which retrieves the list of blocks where the word exists, which then needs to be sequentially processed to find all the occurrences. In these compressed structures, the size of the block is decisive for the space / time tradeoff, where it is better to choose a text compression method that offers faster searches than the Huffman encoding [21, 142]. These compressed inverted indices, alongside the compressed text, require around 30% to 40% of the text size.

Over the years, other approaches for inverted indices arose, using different forms of compression, [7, 184], with its benefits also being studied and proven important [192]. These compact indexes are commonly used in text retrieval frameworks such as Lucene[2], and were proven to have a better performance than compact Suffix Arrays [158] for queries with more than 5,000 occurrences [152]. Lucene is a Java-based information retrieval software used for full text indexing, just like the proposed compressed indices, with efficient searching capability and with space consumption around 30% of text size.

Comparing these space efficient inverted indexes with the Byte-codes Wavelet Tree (WT), shows that the latter is more convenient and efficient to use, either in space requirements and search time response. The Wavelet Tree approach not only requires slightly less space than

[2] http://lucene.apache.org/core/

inverted indexes, compared in similar configurations, but also presents faster results for word and phrase-based searches, regardless their frequency, and until terms with eight words of length [22]. Besides that, it also presents faster results for the *extract* function, very useful for the concordancer application.

Additionally, for functionalities such as the skip-based bilingual search or the hierarchical phrase-based search, it is interesting to have the possibility of skipping to further positions of the text, or even determining directly which words appear nearby an offset. Since the Byte-codes Wavelet Tree follow the order of the text, and allow creating search windows around an offset, much like in the case of Suffix Arrays as well, makes it extremely helpful for the context of these functionalities.

There are some interesting work that studies the usage of main memories of interconnected processors [43, 163, 169], however Brisaboa et. al. [22] demonstrated that by requiring less space, more cache of those processors can be used. This not only is helpful, since caching speeds up the answer for any queries to the index, but also avoids communication between more processors than what it is strictly necessary.

A well known concordancer application, called TransSearch [119], is based on Lucene, offering support for searches over word and multi-word terms, and even discontiguous terms. This concordancer had several improvements over the years, like the support for a word-based alignment, turning the application into a translation search engine [18] as well. The concordancer supported by the presented Bilingual Framework benefits from the favorable comparison between Byte-codes Wavelet Trees and the compact inverted indices that support Lucene, with smaller space consumption and faster search time response. Additionally, representing the text completely, in its compressed form, it offers a more flexible support for terms with gaps, due to the information it indexes and that can be extracted between occurrences of literals.

6.2.2 Word-Based Compressed Suffix Array

Brisaboa et. al. [23] proposed a compressed self-index over a sequence of words of a natural language text, considering the words of the text as the main unit. Since the dependence between consecutive words in a text is significant in natural language [15], the scenario is extremely interesting for obtaining good compression rates.

Building a self-index to words in natural language leads to having alphabets that are considerably larger, which immediately rules out some well known approaches [59, 78], that depend considerably on the size of the vocabulary, $O(\sigma^k)$. A text with n words, usually has a vocabulary of size $\sigma = O(n^\beta)$, with β around 0.5, which means that the numbers can go up easily.

A self-index that does not have any dependence on the alphabet is SAD-CSA [60]. For that matter, Brisaboa et. al. used this Compressed Suffix Array to represent the text, as a sequence of words and separators.

The Word-based Compressed Suffix Array (WCSA) has two layers. Every word or separator of the text is mapped to an integer id. With these ids, a sequence SID is built with every word being replaced by its respective id, much like in the word-based uncompressed suffix array.

SID is then self-indexed in an integer-based CSA (iCSA), being the sequence later discarded. Apart from the CSA, a vocabulary array *VA* is created with the correspondences between the words and the respective *ids*, following an alphabetical order. This word-based compressed index achieves compression results close to many natural language text compressors (without indexation), occupying 35-40% of the text size. Figure 6.7 shows an example of a WCSA for the text `to be or not to be`.

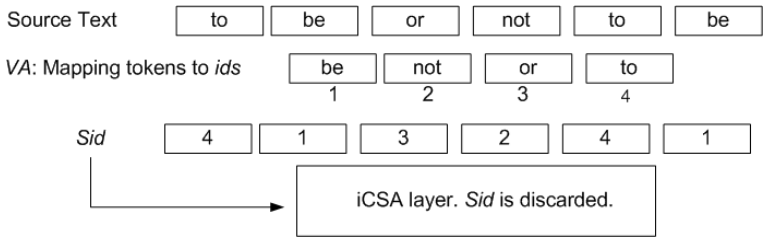

Figure 6.7: General Structure of a Word-Based Compressed Suffix Array

The iCSA layer provides the typical self-indexing operations like *count*, *locate* and *extract*, but for integer patterns. Thus, when querying for a string pattern, it is necessary to find the respective *ids* of each token of the pattern in *VA*, via binary searches [23]. Then, with all the integers obtained from *VA* concatenated, we can use the mentioned functionalities of the iCSA layer, to find the location of the patterns or counting how many times they occur in the text. The main idea behind this procedure does not differ much from the Word-based Suffix Arrays used in the Bilingual Framework, with the main difference lying on the usage of a Compressed Suffix Array in place of a standard, non-compressed one.

The characteristics of this compressed self-index, would also make it as a possible choice for supporting the proposed Bilingual Framework. In fact, this approach and the Byte-codes Wavelet Trees present a very similar space performance and both present better performance than the compressed inverted indexes.

In comparison with block-address inverted indexes, WCSAs perform better in almost all the experimented scenarios. Both data structures were compared with similar space requirements, 45% and 38% of the text size. Considering the latter space requirements, WCSAs perform better than inverted indexes, for *locating* words and phrases, from a number of occurrences from 1 to more than 10000. WCSAs also present better time response for *locating* multi-word terms of two to eight words, and for *extracting* snippets of text. Only when the memory consumption increases to 45%, is when the inverted indexes start to show competitive results. But even then, they only beat WCSAs when *extracting* words with more than 10000 occurrences [23].

When compared with WTs, WCSAs lose for *locating* and *extracting* words in the text. However, they outperform Wavelet Trees when *locating* and *extracting* multi-word terms [23]. With these last two alternatives requiring around the same amount of space, WTs were chosen because of its organization following the order of text, instead of a lexicographic one. This greatly benefits the skip-based bilingual search approach, as the search can resume from any point of the text, using the position given by the jump.

With WCSAs, as it happens as well with the uncompressed version of the Word-based Suffix Array, all the occurrences must be calculated anyway, with the jumps being applied later to the set of occurrences of the given search term. It is true that the results demonstrate that there are more gains with Suffix Arrays, than with the compressed trees. However, it is important to recognize that WCSAs are also compressed indexes, thus it is not as fast and direct to obtain the list of occurrences as in the uncompressed version. This lexicographic order also requires a sorting of the occurrences for the order they appear in the text. As an alternative, the basic approach using binary searches could be used, however it would be an unfair comparison as the number of occurrences processed would be larger.

This does not mean that WCSA would be a worst choice for supporting the Bilingual Framework. We chose Byte-codes Wavelet Trees due to the characteristics of the indices, and for constraints of time it was not possible to implement all the functionalities as well with WCSA. In any case, WCSA would be an interesting alternative to further experiments, to determine its performance on supporting the Bilingual Framework.

6.3 Data Structures in Machine Translation

Suffix Arrays are a common data structure used in Natural Language Processing (NLP) tasks and in Machine Translation (MT) related Tasks, due to their ability to fully index the text and provide fast search functionalities over the textual information. From the most well known full-text indexes, Suffix Trees have been used more frequently on fields related with Bioinformatics and DNA sequencing.

Yamamoto and Church [183] used Suffix Arrays for determining term and document frequency of n-grams over large corpora. This work takes advantage of the structure of the Suffix Arrays, thus it uses the binary searches to find the interval of occurrences of a word or multi-word term in the text, thus determining as well its frequency. Additionally, the lcp (longest common prefix) information is used to determine the frequency of all n-grams in a document, taking advantage of the common prefixes between similar substrings.

This is one of the most important research studies done using Suffix Arrays, since this information is extremely valuable to draw statistics of a term in a particular text. These statistics are then used to build the statistical models that are the core of Statistical Machine Translation [107], besides the known impact it has as a weighting factor in information retrieval or text mining applications [162].

Callison-Burch et.al. [28] used also Suffix Arrays for developing searchable translation memories, allowing searches for phrases over translation archives. This work indexes sentence aligned parallel corpora, with one Suffix Array for every side of the corpora, and an additional index to associate the sentence pairs with their word alignments.

On the same year, Callison-Burch et. al. [27] designed a data structure for phrase-based Statistical Machine Translation, for retrieving long phrases, while using less memory than other approaches based on lookup tables [104]. This work followed the studies that demonstrated that phrase-based Machine Translation [106] made considerable advances, when compared

with the word-based approach [24]. To support the retrieving of phrases, the structure designed was also based on a Suffix Array for every side of the parallel corpora, an index to make the correspondence between sentences in one language and the other and the respective alignment for every sentence pair.

Zhang et. al. [189] also used Suffix Arrays to determine phrase-to-phrase alignment models, for arbitrary long phrases and for every large bilingual corpora. Instead of indexing the translation models, Zhang et. al. opted by indexing the corpora and extract the phrase translations on the fly for each testing sentence [106, 179, 190], or even in disk [188]. This method uses an efficient search procedure, where the prefixes of the phrases being searched for are used to narrow down the search interval, into progressively smaller intervals. This strategy made this approach time and space efficient, while achieving significantly better translation quality [189].

McNamee and Mayfield [129] present an approach for inducing translation equivalent of multi-word terms, using heuristics based on counting the occurrences of those multi-word string in parallel corpora. For that, they use Suffix Arrays and the lcp information, such as the approach by Yamamoto and Church [183], to determine important counts for finding translation equivalents. The procedure starts from sentence aligned corpora built on two Suffix Arrays, and finds sentences containing the searched term st, in one language, and the matching sentences on the other language. Then, those sentences are indexed in a new Suffix Array, using Yamamoto and Church procedure, and the Dice scores are calculated, to return the best ranked alternatives to translate the sentences with st.

All this work based on Suffix Arrays demonstrate the importance of this structure in Machine Translation, with its characteristics and functionalities being extremely important to support these tasks. As demonstrated as well in this thesis, Suffix Arrays provide a fast functionality, for searching for phrase-based and hierarchical phrase-based terms over parallel corpora.

These data structures are space demanding, which led to the study developed in the context of this thesis. Compressed data structures can also be used in the context of Machine Translation, answering to phrase-based queries on demand, or indexing the translation models, since the space requirements of the index allow it.

6.3.1 Hierarchical Machine Translation

Lopez [116] introduced an on the fly lookup algorithm for hierarchical phrase-based matching, following the work by Callison-Burch et. al. [28] and Zhang and Vogel [189] for phrase-based translation models, using Suffix Arrays. As the occurrences of the literals in a discontiguous phrase may not appear consecutively, as seen in the previous Chapter, Lopez started by proposing a strategy to determine the intersection between those occurrences, after studying other alternatives [154]. That strategy, called collocation problem, has three major cases: 1) when two literals are frequent, the intersection is precomputed in a single pass over the text and stored in a cache; 2) when one literal is frequent and the other is not, the proposal uses a double binary search [10], whose time complexity is heavily influenced by the less frequent literal; 3) when both literals are rare, a linear scanning of the lists sufices, as they are small.

For the first case, a precomputation of the most frequent collocations is made, with a single pass over the text, using the most frequent literals. During this pass, the procedure determines the collocations of the most common patterns, with the succeeding patterns in the text, within the maximum length allowed. Also, for scenarios where $L_1\$L_2$, with L_1 and L_2 being two of the most frequent literals, the list of occurrences is also precomputed.

In the second case, the double binary search procedure takes advantage of the less frequent term, by using a similar approach to the binary search-based bilingual search used for the Framework. To use this approach, it is mandatory that the sets of occurrences of both patterns need to be sorted. To supports this behavior, Lopez used Rahman et. al. [154] algorithm for n-grams and a precomputed inverted index for unigrams.

Since a hierarchical phrase term potentially has more occurrences than a standard phrase-based one, Lopez reduced the number of computations made in the algorithm, by extending Zhang and Vogel algorithm [189]. This algorithm takes advantage that any region of a Suffix Array that has the suffixes prefixed by L_1L_2 is a subset of the interval containing the suffixes prefixed by L_2. Following the same strategy, Lopez used this approach to restrict the binary searches mentioned in the previous paragraph. This means that if a search by L_2 fails, we do not need to search for L_1L_2.

This strategy can be extended as well for hierarchical phrase terms. A text only have occurrences of the phrase $L_1L_2L_3$, if there are occurrences of the prefix L_1L_2 and the suffix L_2L_3. This is not much different in the phrase-based search, as what was described above. However, it can be extended for the hierarchical phrase-based search. A hierarchical phrase term, $L_1L_2\$L_3L_4$, only exists in a text if its maximal prefix, $L_1L_2\$$, and maximal suffix, $\$L_3L_4$, exist. Lopez went even further, and used $L_1L_2\$L_3$, the maximal prefix, or $L_2\$L_3L_4$, the maximal suffix, as basis of the search. This reduces the number of occurrences in both sets for the intersection, since the occurrences of such subphrases is likely to be smaller than $L_1L_2\$$ for instance.

Finally, on top of that, this approach uses a mechanism of cache, supported by prefix trees [185]. A prefix tree indexes a text, where each node represents a prefix of the text, which is obtained by concatenating all the words in the path from the root to the node. This cache indexes a mix of phrases that are frequent or difficult to compute, alongside precomputed phrases in previous searches. Prefix trees offer this flexibility, with the ability of marking paths as active or inactive (e.g., to be considered in a search or not), and also fast access for the maximal prefix or maximal suffix of a hierarchical phrase, if needed. Lopez [117] used this algorithm to support translation by computing translation rules only as needed, instead of pre-calculating the complete model, already with the support for hierarchical phrases. This approach requires four times the text size in main memory.

A few years later, Baltescu and Blunsom [13] presented an algorithm for extracting hierarchical phrase-based rules from parallel corpora, using Suffix Arrays, that improved the speed and space consumption over the work proposed by Lopez [116]. Their option for Suffix Arrays resulted from the study by Schwartz and Callison-Burch [165], that concluded that Suffix Arrays perform better than the disk-based approaches [188].

This approach uses a caching layer on top of the Suffix Array to store the set of matches

for every searched phrase. This way, when searching for a phrase $L_1 L_2 L_3$, first the procedure verifies if it is cached, if it is not, then it checks for $L_1 L_2$ and $L_2 L_3$, to determine if it's still possible to save work on processing the complete set of occurrences.

Baltescu and Blunsom considered three different cases. Consider α and β as hierarchical phrases, with α having only one gap and with β being already computed and cached. The first case has $\alpha = \beta$ \$ or $\alpha =$ \$ β. In these cases, since the gaps are at the beginning or end, they are ignored and thus the occurrences of the phrase α are the same as the previously computed phrase β.

In the second case, $\alpha = L_1\ \beta\ L_2\ L_3$. Here, the algorithm selects the occurrences of the phrase β, and iterates over its set of occurrences. Then, for every occurrence in that set, it determines the ones that have L_1 immediately before β, and occurrences of L_2 and L_3 immediately afterwards, by accessing the text. In this verification, the order of the literals in the rule α must be always applied.

In the third and final case, $\alpha = L_1\ \beta$ \$ L_3, where there is a gap after the phrase $L_1\ \beta$. To determine the occurrences of L_3 after the gap, the procedure iterates over the set of occurrences of $L_1\ \beta$, and for every occurrence, it accesses the text to determine if L_3 appears under a limit interval. Baltescu et. al. [13] follow the experimental results obtained by Lopez [118], which shows that translation rules than span more than 15 words have no effect. This rule helps define the limit interval for matching the gap.

This approach is over four times faster than Lopez solution, as it does not use double binary search and does not require sorting, with the cost of a slight increase of memory consumption.

Our implementation of the search for hierarchical phrase-based terms, based Suffix Arrays, uses a very similar approach to this one. In particular, the idea of traversing the list of occurrences of one part of the term, like the less frequent literal, and then determine if the other literals appeared around it.

The algorithm presented in this thesis is simpler. Baltescu and Blunsom approach uses a cache to store the occurrences of the most common phrases in the text, and also the previously processed sets of occurrences, using prefix trees. This approach saves time on processing searches, namely for terms with gaps that have some part of it already preprocessed and cached. The search procedure supported in the proposed Bilingual Framework does not use any caching mechanism. Such support can be space demanding and the focus of this study was to analyze the behavior and performance of the data structures and the functionalities they support, in this context of indexed bilingual texts. In particular, considering that the Framework can also use compressed data structures, such as Byte-codes Wavelet Trees.

Nevertheless, there are several approaches for a cache that could be added to the structure of the Framework, in particular to store references to the occurrences of the most frequent words in the index. However, the space requirements should not increase too much, to avoid losing what is gained by using succinct structures, thus the space / time tradeoff should be planned in detail.

An idea shared by Lopez and Baltescu and Blunson regards the maximum length of 15 word per phrase, when matching a phrase with gaps. In the proposed Bilingual Framework,

we consider the number of fine segments that the term can span across, instead of using the number of words. In any case, the idea is similar, since we also believe that matching terms with gaps can easily match uninteresting results. On top of this, we bound to a match to appear within a coarse segment, since we believe it would make little sense that a match would cross over sentences or paragraphs. However, this parameter can be set and changed depending on the window wanted to extract the matches.

This brings also an important difference from the Bilingual Framework, when compared with these two presented above. Having the bilingual context, instead of just considering one language, brings advantages to the hierarchical phrase-based search. Not only allows filtering the size of the matched occurrences, as explained above, using the alignment, as also allows applying the skip-based bilingual search, to avoid processing unnecessary occurrences of the searched phrases with gaps.

As far as we know, there are no approaches for matching hierarchical phrase-based terms in compressed data structures, so in this thesis we have a first approach for supporting it. As expected, this approach requires much less space than the approach based on Suffix Arrays, but it is also considerably slower. We hope it is a first step for future work that may be explored and published using these succinct data structures.

6.3.2 Machine Translation-related work based on Suffix Arrays

Li et. al [115] proposed an open source toolkit for Statistical Machine Translation, Joshua, in which its grammar extraction is based on the work developed by Callison-Burch et. al. [28] and Lopez [116]. This means that Suffix Arrays are key to this toolkit, as it is the data structure that is used by those two approaches. Joshua also uses paralel and distributed decoding, following on the work by Li and Khudanpur [114], for scaling purposes of the toolkit.

Later, Dyer et. al. [49] proposed cdec, another Framework for decoding, aligning and training SMT models, phrase-based and hierarchical phrase-based. This approach tackles two limitations of previous approaches [107, 115]. In the first one, cdec uncouples the implementation with the translation and language models, and also pruning algorithms, by supporting general scoring, pruning and alignment algorithms. This enables using different models and different model types. In the second one, cdec supports any parameterization, with millions of sparse features, enabling training any model type, with any training criteria [17, 33].

Denkowski et. al. [47] proposed cdec Realtime, a Framework for building Machine Translation systems that learn from post-editor feedback, built over cdec [49]. This work comes from a trend of integrating automatic translation on assisting human translation workflows, with the objective of reducing the human translation effort [46]. This work enables users to translate texts with the help of a Machine Translation system, which in turn also learns from user feedback and improves translation quality, and is based on Suffix Arrays.

More recently, Germann [66] proposed an implementation of a phrase table for Moses, with the functionality of computing phrase table entries for phrase-based Statistical Machine Translation. This approach distinguishes itself by doing this computation on demand, by sampling

and indexed parallel corpora, with Suffix Arrays. This approach has been used in Hierarchical Machine Translation before [116], but it took some time to be adapted for the phrase-based approach, due to some concerns about loss of translation speed and quality. Germann, in this work, is able to prove that these concerns were wrong, by presenting an implementation based on Suffix Arrays, which outperforms disk-based implementations [188], in terms of speed, without losing translation quality.

The implementation of this work, in the Moses Statistical Machine Translation System, was also proposed by Germann [67], with a detailed description of how the approach is combined with the system. This work concluded that sampling phrase tables with Suffix Arrays makes the tables much faster to build, offer flexibility in the choice of feature functions used and they have a lower memory consumption.

The proposed Bilingual Framework could be used as well in the context of the work presented in this subsection, based on Suffix Arrays. With its ability of indexing and querying over parallel corpora, our proposal can support several Statistical Machine Translation related tasks, from the training, to the decoding, by determining relevant statistics, but also tasks extraction of new translation pairs and context analysis.

Some of these approaches use parallel mechanisms, to speedup their procedures, which is an interesting idea for the future of the Bilingual Framework. With both structures used for supporting the Framework, it is possible to add a parallel search implementation, including with the skip-based bilingual search. It would be a matter of dividing the processing in several intervals of the text.

In the data structures, there is recent work on parallelism [64, 91] on Suffix Trees and Arrays, being the work by Fischer et. al. [64] valid for uncompressed and compressed Suffix Arrays. This makes it eligible as well for Compressed Suffix Trees, which are based on Compressed Suffix Arrays. We believe it would be also possible to implement this parallelism over Byte-codes Wavelet Trees, by having several searches starting in different places of the text, in parallel, taking advantage of the self-synchronization nature of the index.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This thesis describes a full-text Bilingual Framework based on compact data structures, for managing and querying bilingual texts in main memory, such as collections of parallel corpora, bilingual lexica, etc. The proposed Framework supports any language pair, using the word as the main unit of the text, and considering the blank space as a separator. This work appears in a context of an iterative workflow of alignment and extraction, that learns information from the previous iterations, together with human input, to apply it on the following iterations and to the Machine Translation task. Thus, it was important to design a Framework that was able to calculate relevant statistics for these tasks in an efficient manner, while being able to index large amounts of textual data in main memory.

Indexing bilingual texts require not just the textual information in both languages, but also the text alignment, taken as established translation correspondences, as it connects the phrases in one language to their respective translation. For that matter, the Bilingual Framework has a Text Layer to index the corpora in the two languages, supported by word-based text indexes, and an Alignment Layer, to represent the correspondences between sentences, or phrases (and subphrases), of the parallel corpora.

At the beginning of the research work described in this thesis, three main objectives were set to be accomplished: 1) implementing the Bilingual Framework, for managing and querying large amounts of bilingual texts in main memory, using compact data structures, to avoid disk storage and access; 2) Speedup the support for $locate$ and $count$ searches over bilingual texts, for efficiently support MT and NLP tasks, such as translation, alignment and extraction; 3) Support the matching for discontiguous phrases in all the functionalities of the Framework.

To accomplish Objective 1, we studied the use of compact data structures to support the Bilingual Framework, either for representing the texts and the alignment. This study led to the implementation of a space efficient approach based on Byte-codes Wavelet Trees, for indexing

the texts, and bitmaps, for representing the alignment. A Byte-codes Wavelet Tree is a word-based representation of compressed text, which requires less space in main memory than the indexed text. The bitmaps enable representing the multi-grain alignment of the indexed texts using bits. As far as we know, there is not any approach that uses compressed data structures for indexing parallel corpora, including the multi-grain alignment information.

The combination of these approaches led to space requirements around 50% of the size of the alignment-annotated texts, in the English-Portuguese and German-Portuguese pairs. These space requirements are very promising, since they allow indexing large amounts of textual data, in different languages, without needing slower disk accesses. In the context of the iterative process of alignment and extraction, these numbers are even more important, due to the accumulated information that may be necessary to index. As alternative, this thesis also introduces a solution based on Word-based Suffix Arrays, for indexing the texts, and Arrays of Integers, for representing the alignment, requiring around 1.6 times the size of the parallel corpora, in the experimented language pairs.

At this stage, the Bilingual Framework supported monolingual and bilingual searches for word and multi-word terms. A monolingual search determines the occurrences of a term in one language, while the bilingual search determines the co-occurrences of a pair of terms, in two languages. The solution based on Word-based Suffix Arrays and Arrays of Integers presents considerable faster results than the most compressed alternative, as it has the indexed text directly accessible, without being necessary to decode it. On the other hand, the space consumption of this approach increases significantly, which can greatly limit the amount of textual data indexed by the Framework.

Knowing that this slowdown is a side effect of using compact indexes, the second Objective of this thesis was focused on speeding up the Framework search response, as most as possible. Since the monolingual search depends on the search algorithms of the data structures used, the speedup was applied to the bilingual search. Originally, the bilingual search procedure was based on binary searches, over the sets of occurrences of both terms, to calculate the co-occurrences. To save time, and meet the second Objective of the thesis, we designed an approach that would avoid processing all the occurrences of the terms, by using the underlying ambiguity that exists when translating from one language to another.

Since a term in one language can be translated by different terms in the other language, we used the alignment information to consider just terms that appear aligned, or in nearby segments, as part of a valid co-occurrence. This led to the skip-based bilingual search information, that using the mentioned information about the alignment, skips over non relevant occurrences of the searched terms, to just consider nearby occurrences. This search procedure improves the bilingual search time, in average, from 1.6x to 2.8x in English - Portuguese, and 1.15x to 2.79x in German-Portuguese, avoiding processing around 52%-71% of the individual occurrences, to obtain the same number of co-occurrences.

With the Bilingual Framework supporting phrase-based searches, we can directly apply its functionalities on Phrase-based Machine Translation related tasks. However, over the last years, Hierarchical Machine Translation emerged as an extension of the phrase-based approach,

which improved the translation quality. A hierarchical phrase is a discontiguous phrase, with variable gaps appearing between subphrases, that can be replaced by any other subphrase. This structure allows matching reorderings of phrases, which was a limitation of the phrase-based model. Thus, the third objective of this thesis focused on extending the functionalities of the Bilingual Framework to add support for hierarchical phrase-based searches.

This extension influences the monolingual search of the Framework, where the data structures that index the texts must determine the occurrences of phrases with gaps, against the usual contiguous phrases. We proposed supporting this behavior by adapting some known algorithms, such as the Longest Common Subsequence for the Byte-codes Wavelet Trees, and recent work developed on extracting hierarchical phrases for Word-based Suffix Arrays. Regardless of these changes in the structure, the memory consumption was not affected, thus meeting the objective of reducing the space requirements of the parallel corpora representation.

The results demonstrate that the monolingual search for hierarchical phrases is considerably slower than the phrase-based implementation, due to the heavier search procedures for matching the gaps. Another factor lies on the variable nature of the gaps, which can match any subphrase, within a limit window, increase considerably the number of occurrences returned.

These factors make the skip-based bilingual search extremely important, since it avoids processing a lot of occurrences that are not relevant to a bilingual search. The results demonstrate that the skip-based search reduces, at least, 51% of the occurrences processed in English - Portuguese, with the maximum reduction reaching 77%. Considering that these experiments have occurrences in the order of billions, and the search procedure is naturally slower when compared to the phrase-based, these reduction ratios are extremely important.

As far as we know, this is the first attempt of matching hierarchical phrases using compressed data structures. With Word-based Suffix Arrays, we adapted an already existing approach [13], without using any auxiliary data structures, to avoid increasing the space requirements of the Framework. This choice was weighted with the already fast search time response of the structure, as seen in the results.

With the three objectives for this thesis met, the Bilingual Framework can be useful for several applications related to Machine Translation and Natural Language Applications, as the entire textual information is represented in main memory. Functionalities such the calculation and location of occurrence and co-occurrence frequencies, alongside context extraction, are crucial for a large set of applications in the mentioned fields of research. Among these tasks are translation alignment, automatic extraction, machine translation, word-sense disambiguation, context analysis, classification and validation of machine-made translations. A first application that was implemented over the presented Bilingual Framework was a bilingual concordancer, an online context analysis tool, useful for human translation and validation.

The results shown in this thesis state that the goals proposed in the research were met, making the Bilingual Framework an useful tool for any of the mentioned tasks. Besides this thesis, the space and phrase-based search time requirements for the Framework were published in two conference publications [39, 40].

7.1 Future Work

As mentioned in the previous Chapter of this thesis, due to time constraints, there were some experiments left to be done, that would be of great relevance for this research.

A first one would be the implementation of the Bilingual Framework, using Word-based Compressed Suffix Arrays [23] for indexing the texts. Despite the index being represented following the lexicographic order, which would disable the direct application of the skips, present faster $locating$ and $extracting$ of phrases.

Then, for the hierarchical phrase-based search, the use of caching mechanisms, such as in the approaches by Lopez [116] and Baltescu et. al. [13], could be an additional experiment to be done. This could speed up the matching for hierarchical phrases, but the space consumption would increase. Nevertheless, it would be interesting to study the space / time tradeoff of those additions to the structure of the Framework.

This research is also a starting point for additional work that can be developed, on top of the proposed Bilingual Framework. Some of this work may be accomplished with the current structure of the Framework, while others imply changing slightly the structure based on indexing monotonic parallel text alignment. The following subsection introduces some of this relevant work for this research.

7.1.1 Linking the gaps of hierarchical phrases in two languages

In this thesis, I presented a procedure for finding the individual occurrences of hierarchical phrase-based terms over parallel corpora, in the respective language, using the proposed Bilingual Framework. Additionally, since there can be many occurrences due to the variable nature of the terms, we used the skip-based bilingual search to skip irrelevant occurrences for a particular search.

The skip-based bilingual search procedure uses the alignment to filter the occurrences of pairs of hierarchical phrase terms, to find their co-occurrences. However, since these terms have gaps, there is no guarantee, except from the alignment, that the two occurrences that meet the Conditions to be a valid co-occurrence, are actually correct translations of the searched terms. Besides that, we only have the maximum length for an occurrence as the base for finding matches. With this method, there is no direct way to guarantee that the content of a gap in one language, is a translation of the content of the other gap in the other language.

To enable this behavior, the search procedure needs to be more demanding at the level of the gaps. Since the literals are non variable parts of the term, the original procedure already guarantees that the occurrences found match those literals. Thus far, the gaps were matched exclusively using a maximum limit window, which can still be applied, alongside the skip-based bilingual search. To filter even further the co-occurrences returned, we propose, as future work, the implementation of the correspondences between the gaps on the terms, in different languages, in the Bilingual Framework.

Consider the phrases `In $1 and $2 alike` <- >`Tanto em $1 como em $2`, in English and Portuguese. These phrases determine that the content of the first gap, in English, is translated by the content of the first gap in Portuguese, and the same for the second gap. This way, after a possible co-occurrence of two terms with gaps is found, the procedure could find out if the gaps are correct translations of each other. If the gaps $\$_1$ are correct translation of each other, as well as the gaps $\$_2$, in both occurrences found, then the there would be a valid co-occurrence. For example, `In Portugal and Spain alike`, is translated by `Tanto em Portugal como em Espanha`, in the English - Portuguese language pair. With the correspondence between gaps, we know that `Portugal` <- >`Portugal` and `Spain` <- >`Espanha`.

To implement this strategy, the Bilingual Framework needs to have information about the known correct translations, in a pair of languages, to determine if the content of two gaps are correct translations of each other. That information can be found in a bilingual lexicon, for the same language pair. A bilingual lexicon is a set of pairs of word or multi-word terms, in two different languages, that are correct translations of each other. Figure 7.1 shows an example of a snippet of a bilingual lexicon in English - Portuguese, with the terms in the same line being correct translations of each other. For example, `European` can be translated in Portuguese in several different ways, being four of the many alternatives represented in the lexicon.

English	**Portuguese**
Commission	Comissão
....	...
European	Europeia
European	Europeias
European	Europeu
European	Europeus
European Parliament	Parlamento Europeu
European Union	União Europeia
...	...
Suffix Tree	Árvore de Sufixos
Tree	Árvore
...	...

Figure 7.1: Snippet of a bilingual lexicon in English - Portuguese

By looking at Figure 7.1, it is easy to find similarities between the structure of a bilingual lexicon with parallel text alignment. In fact, a bilingual lexicon is no more than a monotonic fine-grained alignment, which can be indexed directly by the Bilingual Framework. This way, having the lexicon indexed, for the correspondent language pair, after finding two aligned occurrences, the Framework can query the indexed lexicon, to determine if the content of the gaps are correct translations of each other.

This addition of textual information to the Bilingual Framework can be applied directly to the Text Layer and Alignment Layer, taking advantage of the same search functionalities, and thus search time response. However, it will increase the memory consumption of the Framework, thus making the approach based on compact data structures even more important.

7.1.2 Non-monotonic alignment

After completing the proposed research work, the Bilingual Framework is able to index bilingual texts that are monotonically aligned. For translation memories, sentence-level aligned parallel corpora or bilingual lexica this is enough. However, subphrase-level aligned parallel corpora, such as the ones used in the context of this work, can easily lead to crossed alignments, which may not be correctly tackled by monotonic alignments.

An example of such case is the one depicted in Figure 7.2. With the English phrase `sole control over`, being translated by the Portuguese phrase `o controlo exclusivo de`, where `control over` is translated by `o controlo ... de`. In a monotonic alignment with such finer granularity, no correspondence would be found for these last phrases. The only option would be the whole complete pair `sole control over` <- >`o controlo exclusivo de`. An even more difficult situation would be the phrases `rubber products manufacturing market`, which is translated in Portuguese by `mercado de a indústria transformadora de os produtos de borracha`. However, as Figure 7.2 depicts, the correspondences between the subphrases of those phrases cross heavily between each other, with `rubber` being translated by `borracha` and `market` by `mercado`.

These examples of “misalignments"are easily tackled by non-monotonic alignments, which do not expect the text to be aligned in a monotonic fashion, thus allowing these cross alignments. Representing such kind of alignments would bring benefits on the functionalities of the Bilingual Framework. This representation would enable matching known translations of phrases in both languages, regardless of the order they appear in the corpora. This could be helpful for all the functionalities of the Framework, and also could help on the idea for future work, of matching the gaps between two hierarchical phrase-based terms.

Gomes [69] has recently proposed a non monotonic alignment representation, and aligner tool, that would be an interesting add-on to the Bilingual Framework. This strategy allows an easy representation of crossed alignments, but also enables the representation of all the alignments that a word is part of within a sentence. Figure 7.3 depicts an example also given by Gomes, that shows these groups of alignments, where a word or phrase is part of. In this example, `the room` is aligned with `a sala`, but also `room` is aligned with `espaço`.

Following this grouping of words and phrases, it is possible to create a structure that marks the alignment groups where a word or phrase is part of, regardless of the order. By indexing that info, when finding a phrase in the text, one can easily obtain all the alignments on which the phrase is part of. This can greatly benefit the calculation of co-occurrences, since we can immediately determine, if both terms in the search appear aligned.

This would require changing the Alignment Layer of the proposed Bilingual Framework, by adding an additional structure to represent this type of alignment. It is a fair assessment to state that this new alignment representation would lead to an impact in the space requirements of the Framework, by increasing it. On the other hand, it would be interesting to be able to have a more complete mapping from one language to another, with the possibility of improving functionalities and support new ones.

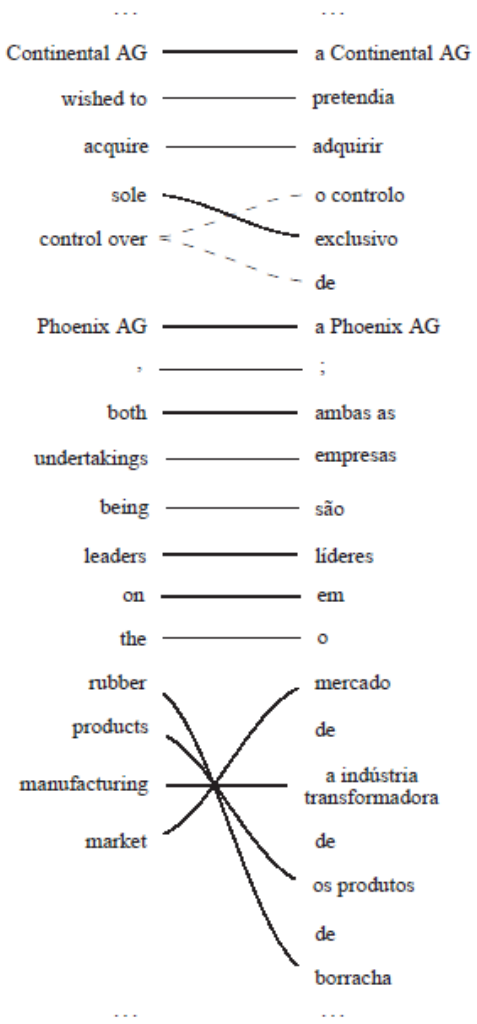

Figure 7.2: Example of misalignments

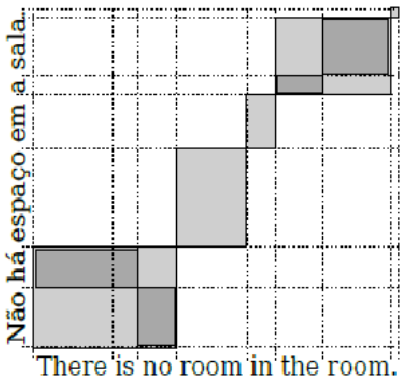

Figure 7.3: Example of new alignment representation

7.1.3 Crossing the word boundary

Recent work analyzed the idea of using stems and suffixes of words to generate new translations [99, 101]. A bilingual lexicon can be easily not complete, as it does not contain all the possible translation pairs between a language pair. The extraction algorithms that try to determine new pairs of translations rely on the parallel corpora, thus the terms that are not on the corpora, are not extracted, unless by using machine learning.

Table 7.1 shows the translation in Portuguese, for the word `ensure` and its forms (`ensures`, `ensured` and `ensuring`). Any of the word forms have plenty of different translations in Portuguese, however there are certain translations missing, such as `ensure` <->`garantam`, `ensured` <- >`garantiu`, and many more. Those translations are probably missing, since they do not appear in the source parallel corpora used for the extraction.

Table 7.1: Snippet of an English - Portuguese biligual lexicon, for the word forms of `ensure`

English	**Portuguese**			
ensure	assegurar asseguram assegurem	zelar	garantir garantem	permitir permitam permitem
ensures	assegura assegure		garanta garante	permita permite
ensured	asseguradas assegurados assegurado assegurou asseguraram		garantidas garantidos garantido	
ensuring	assegurando		garantindo	permitindo

From Table 7.1, it is easy to understand that the translations for the English terms have the same suffixes. For `ensure`, the translations have suffixes such as `am` or `em`, while for `ensured`, the translations have suffixes such as `adas`, `ados`, or `aram`. The same logic applies for the stems of the words. Again for the examples of `ensure` and `ensured`, the translations share the stems `assegur`, `garant`, `permit` and `zel`.

This knowledge can lead to the generation of new translations, such as `ensure` <->`zelam` or `ensured` <->`zelou`, by using the common suffixes and stems from the other translations. Just by analyzing this small example, it is clear how studying this is extremely relevant for Machine Translation, since it allows generating new translations that can increase the quality of the machine-made translation.

For generating these translations, it would be crucial to query for information regarding these stems and suffixes. The Bilingual Framework is word-based, which is enough for the word-based functionalities, but for supporting such queries, it would be ineficient. For that matter, we believe that it would be extremely interesting to extend the Framework, to support the indexation of the texts at the stem and suffix level. Since the word-based approach is based

on blank spaces, or other forms of separators, the same idea could be applied to represent these language structures, by creating a special marker that would work as separator.

Gomes [69] designed an approach that not only supports hierarchical phrases, but also patterns such as $1/ing <->que $1/am. These patterns would represent phrase translations as ensuring <->que asseguram, ensuring <->que garantam or ensuring <->que permitam. In these cases, "/"syntactically means that we have a stem represented by a gap, thus replaceable by variable content, followed by a suffix. Here, again there are gaps that can be capitalized to represent the several pairs introduced in Table 7.1, among others. But, as the number of stems and suffixes in one language is much smaller than the number of words, indexing based on suffixes or prefixes, can lead to further compression rates.

BIBLIOGRAPHY

[1] "A Compressed Self-index Using a Ziv-lempel Dictionary." In: *Proceedings of the 13th International Conference on String Processing and Information Retrieval*. SPIRE'06. Glasgow, UK: Springer-Verlag, 2006, pp. 163–180. ISBN: 3-540-45774-7, 978-3-540-45774-9.

[2] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. "Replacing Suffix Trees with Enhanced Suffix Arrays." In: *Journal of Discrete Algorithms* 2.1 (Mar. 2004), pp. 53–86. ISSN: 1570-8667.

[3] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. 1st ed. Springer Publishing Company, Incorporated, 2008. ISBN: 0387789081, 9780387789088.

[4] J. Aires. "Genuine phrase-based statistical machine translation with supervision." Doctoral dissertation. Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2015.

[5] J. Aires, G. P. Lopes, and L. Gomes. "Phrase Translation Extraction from Aligned Parallel Corpora Using Suffix Arrays and Related Structures." In: *Progress in Artificial Intelligence: 14th Portuguese Conference on Artificial Intelligence, EPIA 2009, Aveiro, Portugal, October 12-15, 2009. Proceedings*. Ed. by L. S. Lopes, N. Lau, P. Mariano, and L. M. Rocha. Springer Berlin Heidelberg, 2009, pp. 587–597. ISBN: 978-3-642-04686-5.

[6] A. Andersson, N. J. Larsson, and K. Swanson. "Suffix trees on words." In: *Combinatorial Pattern Matching: 7th Annual Symposium, CPM 96 Laguna Beach, California, June 10–12, 1996 Proceedings*. Ed. by D. Hirschberg and G. Myers. Springer Berlin Heidelberg, 1996, pp. 102–115. ISBN: 978-3-540-68390-2.

[7] V. N. Anh and A. Moffat. "Index Compression Using Fixed Binary Codewords." In: *Proceedings of the 15th Australasian Database Conference - Volume 27*. ADC '04. Australian Computer Society, Inc., 2004, pp. 61–67.

[8] A. Apostolico. "The Myriad Virtues of Subword Trees." In: *Combinatorial Algorithms on Words*. Ed. by A. Apostolico and Z. Galil. Springer Berlin Heidelberg, 1985, pp. 85–96. ISBN: 978-3-642-82456-2.

[9] R. Baeza-Yates and G. Navarro. "Block-Addressing Indices for Approximate Text Retrieval." In: *J. of the American Society for Information Science (JASIS)* 51.1 (Jan. 2000), pp. 69–82.

[10] R. Baeza-Yates. "A Fast Set Intersection Algorithm for Sorted Sequences." In: *Combinatorial Pattern Matching: 15th Annual Symposium, CPM 2004, Istanbul, Turkey, July 5-7, 2004. Proceedings*. Ed. by S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusoz. Springer Berlin Heidelberg, 2004, pp. 400–408. ISBN: 978-3-540-27801-6.

[11] R. Baeza-Yates and A. Salinger. "Experimental Analysis of a Fast Intersection Algorithm for Sorted Sequences." In: *String Processing and Information Retrieval: 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005. Proceedings*. Ed. by M. Consens and G. Navarro. Springer Berlin Heidelberg, 2005, pp. 13–24. ISBN: 978-3-540-32241-2.

[12] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 020139829X.

[13] P. Baltescu and P. Blunsom. "A Fast and Simple Online Synchronous Context Free Grammar Extractor." In: *The Prague Bulletin of Mathematical Linguistics* 102.1 (2014), pp. 17–26.

[14] J. Barbay, A. López-Ortiz, and T. Lu. "Faster Adaptive Set Intersections for Text Searching." In: *Experimental Algorithms: 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006. Proceedings*. Ed. by C. Àlvarez and M. Serna. Springer Berlin Heidelberg, 2006, pp. 146–157. ISBN: 978-3-540-34598-5.

[15] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, Inc., 1990. ISBN: 0-13-911991-4.

[16] M. A. Bender and M. Farach-Colton. "The LCA Problem Revisited." In: *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*. LATIN '00. Springer-Verlag, 2000, pp. 88–94. ISBN: 3-540-67306-7.

[17] P. Blunsom, T. Cohn, and M. Osborne. "A Discriminative Latent Variable Model for Statistical Machine Translation." In: *ACL 2008, Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics, June 15-20, 2008, Columbus, Ohio, USA*. 2008, pp. 200–208.

[18] J. Bourdaillet, S. Huet, P. Langlais, and G. Lapalme. "TransSearch: From a Bilingual Concordancer to a Translation Finder." In: *Machine Translation* 24.3-4 (Dec. 2010), pp. 241–271. ISSN: 0922-6567.

[19] R. S. Boyer and J. S. Moore. "A Fast String Searching Algorithm." In: *Communications of the ACM* 20.10 (Oct. 1977), pp. 762–772. ISSN: 0001-0782.

[20] F. Braune and A. Fraser. "Improved Unsupervised Sentence Alignment for Symmetrical and Asymmetrical Parallel Corpora." In: *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*. COLING '10. Beijing, China: Association for Computational Linguistics, 2010, pp. 81–89.

[21] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. "Lightweight Natural Language Text Compression." In: *Information Retrieval* 10.1 (Jan. 2007), pp. 1–33. ISSN: 1386-4564.

[22] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. "Reorganizing Compressed Text." In: *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '08. ACM, 2008, pp. 139–146. ISBN: 978-1-60558-164-4.

[23] N. R. Brisaboa, A. Fariña, G. Navarro, A. S. Places, and E. Rodríguez. "Self-indexing Natural Language." In: *String Processing and Information Retrieval: 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10-12, 2008. Proceedings*. Ed. by A. Amir, A. Turpin, and A. Moffat. Springer Berlin Heidelberg, 2009, pp. 121–132. ISBN: 978-3-540-89097-3.

[24] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer. "The Mathematics of Statistical Machine Translation: Parameter Estimation." In: *Computational Linguistics* 19.2 (June 1993), pp. 263–311.

[25] M. Burrows and D. Wheeler. "A Block-sorting Lossless Data Compression Algorithm." In: A block-sorting lossless data compression algorithm n.º 124 (1994).

[26] C. Callison-Burch, C. Bannard, and J. Schroeder. "Improved statistical translation through editing." In: *EAMT-2004 Workshop*. 2004.

[27] C. Callison-Burch, C. Bannard, and J. Schroeder. "A Compact Data Structure for Searchable Translation Memories." In: *European Association for Machine Translation*. 2005.

[28] C. Callison-Burch, C. Bannard, and J. Schroeder. "Scaling Phrase-based Statistical Machine Translation to Larger Corpora and Longer Phrases." In: *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. ACL '05. Association for Computational Linguistics, 2005, pp. 255–262.

[29] R. Cánovas and G. Navarro. "Practical Compressed Suffix Trees." In: *Proceedings of the 9th International Conference on Experimental Algorithms*. SEA'10. Springer-Verlag, 2010, pp. 94–105. ISBN: 3-642-13192-1, 978-3-642-13192-9.

[30] J. Casteleiro, J. F. da Silva, and G. P. Lopes. "Cross-Lingual Word Sense Clustering for Sense Disambiguation." In: *Progress in Artificial Intelligence: 17th Portuguese Conference on Artificial Intelligence, EPIA 2015, Coimbra, Portugal, September 8-11, 2015. Proceedings*. Ed. by F. Pereira, P. Machado, E. Costa, and A. Cardoso. Springer International Publishing, 2015, pp. 747–758.

[31] D. Chiang. "A Hierarchical Phrase-based Model for Statistical Machine Translation." In: *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. ACL '05. Ann Arbor, Michigan: Association for Computational Linguistics, 2005, pp. 263–270.

[32] D. Chiang. "Hierarchical Phrase-Based Translation." In: *Comput. Linguist.* 33.2 (June 2007), pp. 201–228. ISSN: 0891-2017.

[33] D. Chiang, K. Knight, and W. Wang. "11,001 New Features for Statistical Machine Translation." In: *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. NAACL '09. Boulder, Colorado: Association for Computational Linguistics, 2009, pp. 218–226. ISBN: 978-1-932432-41-1.

[34] D. R. Clark. "Compact Pat Trees." UMI Order No. GAXNQ-21335. Doctoral dissertation. Waterloo, Ontario, Canada, 1998.

[35] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001. ISBN: 0070131511.

[36] J. Costa. "Estruturas de dados para representação de um léxico bilingue." Master's thesis. Faculdade de Ciências e Tecnologia – Universidade Nova de Lisboa, 2010.

[37] J. Costa, L. Gomes, G. P. L. Lopes, and L. M. Russo. "Managing and querying a bilingual lexicon with suffix trees." In: *Proceedings of the 15th Portuguese Conference in Arificial Intelligence, EPIA 2011*. APPIA. 2011, pp. 675–689.

[38] J. Costa, G. P. Lopes, L. Gomes, and L. M. S. Russo. "Representing a Bilingual Lexicon with Suffix Trees." In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. TaiChung, Taiwan: ACM, 2011, pp. 1164–1165. ISBN: 978-1-4503-0113-8.

[39] J. Costa, L. Gomes, G. P. Lopes, L. M. S. Russo, and N. R. Brisaboa. "Compact and Fast Indexes for Translation Related Tasks." In: *Progress in Artificial Intelligence: 16th Portuguese Conference on Artificial Intelligence, EPIA 2013, Angra do Heroísmo, Azores, Portugal, September 9-12, 2013. Proceedings*. Ed. by L. Correia, L. P. Reis, and J. Cascalho. Springer Berlin Heidelberg, 2013, pp. 504–515. ISBN: 978-3-642-40669-0.

[40] J. Costa, L. Gomes, G. P. Lopes, and L. M. S. Russo. "Improving Bilingual Search Performance Using Compact Full-Text Indices." In: *Computational Linguistics and Intelligent Text Processing: 16th International Conference, CICLing 2015, Cairo, Egypt, April 14-20, 2015, Proceedings, Part I*. Ed. by A. Gelbukh. Springer International Publishing, 2015, pp. 582–595.

[41] Crauser and Ferragina. "A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory." In: *Algorithmica* 32.1 (Jan. 2002), pp. 1–35. ISSN: 0178-4617.

[42] J. S. Culpepper and A. Moffat. "Enhanced Byte Codes with Restricted Prefix Properties." In: *Proceedings of the 12th International Conference on String Processing and Information Retrieval*. SPIRE'05. Buenos Aires, Argentina: Springer-Verlag, 2005, pp. 1–12. ISBN: 3-540-29740-5, 978-3-540-29740-6.

[43] J. S. Culpepper and A. Moffat. "Compact Set Representation for Information Retrieval." In: *Proceedings of the 14th International Conference on String Processing and Information Retrieval*. SPIRE'07. Santiago, Chile: Springer-Verlag, 2007, pp. 137–148. ISBN: 3-540-75529-2, 978-3-540-75529-6.

[44] A. L. Delcher, S. L. Salzberg, and A. M. Phillippy. "Using MUMmer to identify similar regions in large sequence sets." In: *Current Protocols in Bioinformatics* (2003), pp. 10–3.

[45] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. "Better External Memory Suffix Array Construction." In: *Exponential Algorithmics* 12 (Aug. 2008), pp. 1–24. ISSN: 1084-6654.

[46] M. Denkowski, C. Dyer, and A. Lavie. "Learning from post-editing: Online model adaptation for statistical machine translation." In: *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics* (2014), pp. 395–404.

[47] M. Denkowski, A. Lavie, I. Lacruz, and C. Dyer. "Real time adaptive machine translation for post-editing with cdec and TransCenter." In: *Proceedings of the Workshop on Humans and Computer-assisted Translation*. 2014, pp. 72–77.

[48] H. H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. "Fast Relative Lempel-ziv Self-index for Similar Sequences." In: FAW-AAIM'12 (2012), pp. 291–302.

[49] C. Dyer, J. Weese, H. Setiawan, A. Lopez, F. Ture, V. Eidelman, J. Ganitkevitch, P. Blunsom, and P. Resnik. "cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models." In: *Proceedings of the ACL 2010 System Demonstrations*. Association for Computational Linguistics. 2010, pp. 7–12.

[50] P. Elias. "Universal Codeword Sets and Representations of the Integers." In: *IEEE Trans. Inf. Theor.* 21.2 (Sept. 2006), pp. 194–203. ISSN: 0018-9448.

[51] "Engineering a Lightweight Suffix Array ConstructionAlgorithm." In: *Algorithmica* 40.1 (2004), pp. 33–50. ISSN: 1432-0541.

[52] M. Farach. "Optimal Suffix Tree Construction with Large Alphabets." In: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*. FOCS '97. Washington, DC, USA, 1997, pp. 137–. ISBN: 0-8186-8197-7.

[53] A. Fariña, S. Ladra, O. Pedreira, and A. S. Places. "Rank and Select for Succinct Data Structures." In: *Electronic Notes in Theoretical Computer Science* 236 (Apr. 2009), pp. 131–145. ISSN: 1571-0661.

[54] P. Ferragina and G. Manzini. "Opportunistic Data Structures with Applications." In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. FOCS '00. IEEE Computer Society, 2000, pp. 390–. ISBN: 0-7695-0850-2.

[55] P. Ferragina and J. Fischer. "Suffix Arrays on Words." In: *Proceedings of the 18th Annual Conference on Combinatorial Pattern Matching*. CPM'07. London, Canada: Springer-Verlag, 2007, pp. 328–339. ISBN: 3-540-73436-8, 978-3-540-73436-9.

[56] P. Ferragina and R. Grossi. "The String B-tree: A New Data Structure for String Search in External Memory and Its Applications." In: *J. ACM* 46.2 (Mar. 1999), pp. 236–280. ISSN: 0004-5411.

[57] P. Ferragina and G. Manzini. "An Experimental Study of an Opportunistic Index." In: *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '01. Washington, D.C., USA: Society for Industrial and Applied Mathematics, 2001, pp. 269–278. ISBN: 0-89871-490-7.

[58] P. Ferragina and G. Manzini. "Indexing Compressed Text." In: *Journal of the ACM* 52.4 (July 2005), pp. 552–581. ISSN: 0004-5411.

[59] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. "Compressed Representations of Sequences and Full-text Indexes." In: *ACM Transactions on Algorithms* 3.2 (May 2007). ISSN: 1549-6325.

[60] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. "Compressed representations of sequences and full-text indexes." In: *ACM Transactions on Algorithms (TALG)* 3.2 (2007), p. 20.

[61] P. Ferragina, T. Gagie, and G. Manzini. "Lightweight Data Indexing and Compression in External Memory." In: *Algorithmica* 63.3 (July 2012), pp. 707–730. ISSN: 0178-4617.

[62] J. Fischer. "Wee lcp." In: *Information Processing Letters* 110.8-9 (2010), pp. 317–320.

[63] J. Fischer, V. Mäkinen, and G. Navarro. "Faster Entropy-bounded Compressed Suffix Trees." In: *Theoretical Computer Science* 410.51 (Nov. 2009), pp. 5354–5364. ISSN: 0304-3975.

[64] J. Fischer, D. Köppl, and F. Kurpicz. "On the Benefit of Merging Suffix Array Intervals for Parallel Pattern Matching." In: *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, Tel Aviv, Israel*. 2016, 26:1–26:11.

[65] T. Gagie, P. Gawrychowski, J. Kärkkäinen, and Y. Nekrich. "A Faster Grammar-based Self-index." In: LATA'12 (2012), pp. 240–251.

[66] U. Germann. "Dynamic phrase tables for machine translation in an interactive post-editing scenario." In: *Proceedings of the Workshop on interactive and adaptive machine translation*. Vancouver, BC, Canada, 2014, pp. 20–31.

[67] U. Germann. "Sampling phrase tables for the moses statistical machine translation system." In: *The Prague Bulletin of Mathematical Linguistics* 104.1 (2015), pp. 39–50.

[68] S. Gog, A. Moffat, J. S. Culpepper, A. Turpin, and A. Wirth. "Large-Scale Pattern Search Using Reduced-Space On-Disk Suffix Arrays." In: *IEEE Transactions on Knowledge and Data Engineering* 26.8 (2014), pp. 1918–1931.

[69] L. Gomes. "Translation and alignment extraction within a human-machine interactive workflow." To be published. Doctoral dissertation. Faculdade de Ciências e Tecnologia – Universidade Nova de Lisboa, 2017.

[70] L. Gomes and G. P. Lopes. "Measuring Spelling Similarity for Cognate Identification." In: *Proceedings of the 15th Portugese Conference on Progress in Artificial Intelligence*. EPIA'11. Lisbon, Portugal: Springer-Verlag, 2011, pp. 624–633. ISBN: 978-3-642-24768-2.

[71] L. Gomes and G. Pereira Lopes. "First Steps Towards Coverage-Based Document Alignment." In: *Proceedings of the First Conference on Machine Translation*. Berlin, Germany: Association for Computational Linguistics, 2016, pp. 697–702.

[72] L. Gomes, J. Aires, and G. P. Lopes. "Parallel Texts Alignment." In: *New Trends in Artificial Intelligence — 14th Portuguese Conference on Artificial Intelligence, EPIA 2009*. Aveiro, Portugal: Universidade de Aveiro, 2009, pp. 513–524.

[73] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. "Practical implementation of rank and select queries." In: *In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*. 2005, pp. 27–38.

[74] S. Green, J. Heer, and C. D. Manning. "The Efficacy of Human Post-editing for Language Translation." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '13. ACM, 2013, pp. 439–448. ISBN: 978-1-4503-1899-0.

[75] R Grossi, A Gupta, and J. Vitter. "Higher order entropy analysis of compressed suffix arrays." In: *DIMACS Workshop on Data Compression in Networks and Applications*. 2003, pp. 841–850.

[76] R. Grossi and J. S. Vitter. "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching (Extended Abstract)." In: *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*. STOC '00. Portland, Oregon, USA: ACM, 2000, pp. 397–406. ISBN: 1-58113-184-4.

[77] R. Grossi and J. S. Vitter. "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching." In: *SIAM Journal on Computing* 35.2 (Aug. 2005), pp. 378–407. ISSN: 0097-5397.

[78] R. Grossi, A. Gupta, and J. S. Vitter. "High-order Entropy-compressed Text Indexes." In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '03. Baltimore, Maryland: Society for Industrial and Applied Mathematics, 2003, pp. 841–850. ISBN: 0-89871-538-5.

[79] R. Grossi, A. Gupta, and J. S. Vitter. "When indexing equals compression: Experiments with compressing suffix arrays and applications." In: *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2004, pp. 636–645.

[80] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. "Compressed Data Structures: Dictionaries and Data-aware Measures." In: vol. 387. 3. Essex, UK: Elsevier Science Publishers Ltd., Nov. 2007, pp. 313–331.

[81] D. Gusfield. "Suffix trees (and relatives) come of age in bioinformatics." In: *Bioinformatics Conference, 2002. Proceedings. IEEE Computer Society*. IEEE. 2002, p. 3.

[82] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. New York, NY, USA: Cambridge University Press, 1997. ISBN: 0-521-58519-8.

[83] W.-K. Hon, K. Sadakane, and W.-K. Sung. "Breaking a Time-and-Space Barrier in Constructing Full-Text Indices." In: *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 251–260. ISBN: 0-7695-2040-5.

[84] W.-K. Hon, T.-W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. "Succinct Index for Dynamic Dictionary Matching." In: *Proceedings of the 20th International Symposium on Algorithms and Computation*. ISAAC '09. Honolulu, Hawaii: Springer-Verlag, 2009, pp. 1034–1043. ISBN: 978-3-642-10630-9.

[85] D. A. Huffman et al. "A method for the construction of minimum redundancy codes." In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[86] M. Höhl, S. Kurtz, and E. Ohlebusch. "Efficient multiple genome alignment." In: *Bioinformatics* 18 (2002), S312–S320.

[87] T. Ildefonso and G. P. Lopes. "Longest Sorted Sequence Algorithm for Parallel Text Alignment." In: *Computer Aided Systems Theory – EUROCAST 2005: 10th International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, February 7 – 11, 2005, Revised Selected Papers*. Ed. by R. Moreno Díaz, F. Pichler, and A. Quesada Arencibia. Springer Berlin Heidelberg, 2005, pp. 81–90. ISBN: 978-3-540-31829-3.

[88] S. Inenaga and M. Takeda. "On-Line Linear-Time Construction of Word Suffix Trees." In: *Combinatorial Pattern Matching: 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006. Proceedings*. Ed. by M. Lewenstein and G. Valiente. Springer Berlin Heidelberg, 2006, pp. 60–71. ISBN: 978-3-540-35461-1.

[89] H. Itoh and H. Tanaka. "An Efficient Method for in Memory Construction of Suffix Arrays." In: *Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*. SPIRE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 81–88. ISBN: 0-7695-0268-7.

[90] G. J. Jacobson. "Succinct Static Data Structures." AAI8918056. Doctoral dissertation. Pittsburgh, PA, USA, 1988.

[91] M. Jekovec and A. Brodnik. "Parallel Query in the Suffix Tree." In: *CoRR* abs/1509.06167 (2015).

[92] J. Kärkkäinen and D. Kempa. "Engineering a Lightweight External Memory Suffix Array Construction Algorithm." In: *Mathematics in Computer Science* 11.2 (2017), pp. 137–149. ISSN: 1661-8289.

[93] J. Kärkkäinen and P. Sanders. "Simple Linear Work Suffix Array Construction." In: *Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30 – July 4, 2003 Proceedings*. Ed. by J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 943–955. ISBN: 978-3-540-45061-0.

[94] J. Kärkkäinen and S Srinivasa Rao. "Full-text indexes in external memory." In: *Algorithms for Memory Hierarchies* (2003), pp. 149–170.

[95] J. Kärkkäinen and E. Ukkonen. "Sparse Suffix Trees." In: COCOON '96 (1996), pp. 219–230.

[96] J. Kärkkäinen, P. Sanders, and S. Burkhardt. "Linear Work Suffix Array Construction." In: *Journal of the ACM* 53.6 (Nov. 2006), pp. 918–936. ISSN: 0004-5411.

[97] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications." In: *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*. CPM '01. London, UK: Springer-Verlag, 2001, pp. 181–192. ISBN: 3-540-42271-4.

[98] K. M. Kavitha, L. Gomes, J. Aires, and G. P. Lopes. "Classification and Selection of Translation Candidates for Parallel Corpora Alignment." In: *Progress in Artificial Intelligence: 17th Portuguese Conference on Artificial Intelligence, EPIA 2015, Coimbra, Portugal, September 8-11, 2015. Proceedings*. Ed. by F. Pereira, P. Machado, E. Costa, and A. Cardoso. Springer International Publishing, 2015, pp. 723–734. ISBN: 978-3-319-23485-4.

[99] K. M. Kavitha, L. Gomes, and J. G. Lopes. "Learning Clusters of Bilingual Suffixes Using Bilingual Translation Lexicon." In: *Proceedings of the Third International Conference on Mining Intelligence and Knowledge Exploration - Volume 9468*. MIKE 2015. Hyderabad, India: Springer-Verlag New York, Inc., 2015, pp. 607–615. ISBN: 978-3-319-26831-6.

[100] K. M. Kavitha. "Augmenting Translation Lexica by Learning Generalised Translation Patterns." Doctoral dissertation. Caparica, Portugal, 2017.

[101] K. M. Kavitha, L. Gomes, and J. G. P. Lopes. "Identification of Bilingual Suffix Classes for Classification and Translation Generation." In: *Advances in Artificial Intelligence – IBERAMIA 2014: 14th Ibero-American Conference on AI, Santiago de Chile, Chile, November 24-27, 2014, Proceedings*. Ed. by A. L. Bazzan and K. Pichara. Springer International Publishing, 2014, pp. 154–166. ISBN: 978-3-319-12027-0.

[102] D. K. Kim and H. Park. "A New Compressed Suffix Tree Supporting Fast Search and Its Construction Algorithm Using Optimal Working Space." In: *Proceedings of the 16th Annual Conference on Combinatorial Pattern Matching*. CPM'05. Jeju Island, Korea: Springer-Verlag, 2005, pp. 33–44. ISBN: 3-540-26201-6, 978-3-540-26201-5.

[103] P. Ko and S. Aluru. “Space Efficient Linear Time Construction of Suffix Arrays.” In: *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*. CPM'03. Morelia, Michoacán, Mexico: Springer-Verlag, 2003, pp. 200–210. ISBN: 3-540-40311-6.

[104] P. Koehn. “Pharaoh: A Beam Search Decoder for Phrase-Based Statistical Machine Translation Models.” In: (2004). Ed. by R. E. Frederking and K. B. Taylor, pp. 115–124.

[105] P. Koehn. “Europarl: A Parallel Corpus for Statistical Machine Translation.” In: *Conference Proceedings: the tenth Machine Translation Summit*. Phuket, Thailand: AAMT, 2005, pp. 79–86.

[106] P. Koehn, F. J. Och, and D. Marcu. “Statistical phrase-based translation.” In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*. Association for Computational Linguistics. 2003, pp. 48–54.

[107] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst. “Moses: Open Source Toolkit for Statistical Machine Translation.” In: *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*. ACL '07. Prague, Czech Republic: Association for Computational Linguistics, 2007, pp. 177–180.

[108] G. Kremer, M. Hartung, S. Padó, and S. Riezler. “Statistical Machine Translation Support Improves Human Adjective Translation.” In: *Translation: Computation, Corpora, Cognition* 2.1 (2012).

[109] S. Kumar, Y. Deng, and W. Byrne. “A weighted finite state transducer translation template model for statistical machine translation.” In: *Natural Language Engineering* 12.01 (2006), pp. 35–75.

[110] S. Kurtz. “Reducing the Space Requirement of Suffix Trees.” In: *Software Practice & Experience* 29.13 (Nov. 1999), pp. 1149–1171. ISSN: 0038-0644.

[111] J. Kärkkäinen and E. Ukkonen. “Lempel-Ziv parsing and sublinear-size index structures for string matching (Extended Abstract).” In: *Proceedings on the 3rd South American Workshop on String Processing (WSP'96)*. Carleton University Press, 1996, pp. 141–155.

[112] N. J. Larsson and K. Sadakane. “Faster Suffix Sorting.” In: *Theoretical Computer Science* 387.3 (Nov. 2007), pp. 258–272. ISSN: 0304-3975.

[113] W. Li. “Random Texts Exhibit Zipf'S-law-like Word Frequency Distribution.” In: *IEEE Transactions on Information Theory* 38.6 (Sept. 2006), pp. 1842–1845. ISSN: 0018-9448.

[114] Z. Li and S. Khudanpur. “Large-scale discriminative n-gram language models for statistical machine translation.” In: *Proceedings of 8th Biennial Conference of the Association for Machine Translation in the Americas*. 2008, pp. 133–142.

[115] Z. Li, C. Callison-Burch, C. Dyer, J. Ganitkevitch, S. Khudanpur, L. Schwartz, W. N. G. Thornton, J. Weese, and O. F. Zaidan. "Joshua: An Open Source Toolkit for Parsing-based Machine Translation." In: *Proceedings of the Fourth Workshop on Statistical Machine Translation*. StatMT '09. Athens, Greece: Association for Computational Linguistics, 2009, pp. 135–139.

[116] A. Lopez. "Hierarchical Phrase-Based Translation with Suffix Arrays." In: *Proceedings of 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. 2007, pp. 976–985.

[117] A. Lopez. "Tera-scale Translation Models via Pattern Matching." In: *Proceedings of the 22Nd International Conference on Computational Linguistics - Volume 1*. COLING '08. Manchester, United Kingdom: Association for Computational Linguistics, 2008, pp. 505–512. ISBN: 978-1-905593-44-6.

[118] A. D. Lopez. "Machine Translation by Pattern Matching." AAI3307829. Doctoral dissertation. College Park, MD, USA, 2008. ISBN: 978-0-549-57255-8.

[119] E. Macklovitch, M. Simard, and P. Langlais. "TransSearch: A Free Translation Memory on the World Wide Web." In: *Second International Conference On Language Resources and Evaluation*. Vol. 3. Athens, Greece, 2000, pp. 1201–1208.

[120] V. Mäkinen. "Compact Suffix Array." In: *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*. COM '00. London, UK: Springer-Verlag, 2000, pp. 305–319. ISBN: 3-540-67633-3.

[121] V. Mäkinen. "Compact Suffix Array: A Space-efficient Full-text Index." In: *Fundamenta Informaticae: Special issue on computing patterns in strings* 56.1,2 (Oct. 2002), pp. 191–210. ISSN: 0169-2968.

[122] V. Mäkinen and G. Navarro. "Compressed Compact Suffix Arrays." In: *Combinatorial Pattern Matching: 15th Annual Symposium, CPM 2004, Istanbul, Turkey, July 5-7, 2004. Proceedings*. Ed. by S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusoz. Springer Berlin Heidelberg, 2004, pp. 420–433. ISBN: 978-3-540-27801-6.

[123] V. Mäkinen and G. Navarro. "Succinct Suffix Arrays Based on Run-length Encoding." In: vol. 12. 1. Finland: Publishing Association Nordic Journal of Computing, Mar. 2005, pp. 40–66.

[124] V. Mäkinen and G. Navarro. "Rank and Select Revisited and Extended." In: *Theoretical Computer Science* 387.3 (Nov. 2007), pp. 332–347. ISSN: 0304-3975.

[125] U. Manber and G. Myers. "Suffix Arrays: A New Method for On-line String Searches." In: *SIAM Journal on Computing* 22.5 (Oct. 1993), pp. 935–948. ISSN: 0097-5397.

[126] G. Manzini. "An Analysis of the Burrows&Mdash;Wheeler Transform." In: *Journal of the ACM* 48.3 (May 2001), pp. 407–430. ISSN: 0004-5411.

[127] E. M. McCreight. "A Space-Economical Suffix Tree Construction Algorithm." In: *Journal of the ACM* 23.2 (Apr. 1976), pp. 262–272. ISSN: 0004-5411.

[128] T. McEnery, R. Xiao, and Y. Tono. *Corpus-based Language Studies: An Advanced Resource Book*. Routledge applied linguistics. Routledge, 2006. ISBN: 9780415286220.

[129] P. McNamee and J. Mayfield. "Translation of multiword expressions using parallel suffix arrays." In: *5th Conference of the Association for Machine Translation in the Americas (AMTA), Boston, Massachusetts*. 2006.

[130] I. D. Melamed. "Bitext Maps and Alignment via Pattern Recognition." In: *Computational Linguistics* 25.1 (Mar. 1999), pp. 107–130. ISSN: 0891-2017.

[131] A. Moffat. "Word-based Text Compression." In: *Software Practice & Experience* 19.2 (Feb. 1989), pp. 185–198. ISSN: 0038-0644.

[132] R. C. Moore. *Fast and Accurate Sentence Alignment of Bilingual Corpora*. AMTA '02. London, UK: Springer-Verlag, 2002, pp. 135–144. ISBN: 3-540-44282-0.

[133] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. "Fast and Flexible Word Searching on Compressed Text." In: *ACM Transactions on Information Systems* 18.2 (Apr. 2000), pp. 113–139. ISSN: 1046-8188.

[134] E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. "Fast Searching on Compressed Text Allowing Errors." In: *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '98. Melbourne, Australia: ACM, 1998, pp. 298–306. ISBN: 1-58113-015-5.

[135] J. I. Munro. "Tables." In: *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*. London, UK: Springer-Verlag, 1996, pp. 37–42. ISBN: 3-540-62034-6.

[136] J. I. Munro and V. Raman. "Succinct Representation of Balanced Parentheses and Static Trees." In: *SIAM Journal on Computing* 31.3 (Mar. 2002), pp. 762–776. ISSN: 0097-5397.

[137] J. I. Munro, V. Raman, and S. Rao. "Space Efficient Suffix Trees." In: *Journal of Algorithms* 39.2 (May 2001), pp. 205–222. ISSN: 0196-6774.

[138] S. Muthukrishnan. "Efficient Algorithms for Document Retrieval Problems." In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 657–666. ISBN: 0-89871-513-X.

[139] G. Navarro. "Indexing Text Using the Ziv-Lempel Trie." In: *Journal of Discrete Algorithms* 2.1 (Mar. 2004), pp. 87–114. ISSN: 1570-8667.

[140] G. Navarro and V. Mäkinen. "Compressed Full-text Indexes." In: *ACM Computing Surveys (CSUR)* 39.1 (Apr. 2007). ISSN: 0360-0300.

[141] G. Navarro and A. O. n. Pereira. "Faster Compressed Suffix Trees for Repetitive Collections." In: *Journal of Experimental Algorithmics* 21 (Jan. 2016), 1.8:1–1.8:38. ISSN: 1084-6654.

[142] G. Navarro, E. S. De Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. "Adding Compression to Block Addressing Inverted Indexes." In: *Information Retrieval* 3.1 (July 2000), pp. 49–77. ISSN: 1386-4564.

[143] J. Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 0125184050.

[144] S. Nießen, S. Vogel, H. Ney, and C. Tillmann. "A DP based search algorithm for statistical machine translation." In: *Proceedings of the 17th international conference on Computational linguistics-Volume 2*. Association for Computational Linguistics. 1998, pp. 960–967.

[145] G. Nong, S. Zhang, and W. H. Chan. "Two Efficient Algorithms for Linear Time Suffix Array Construction." In: *IEEE Transactions on Computers* 60.10 (Oct. 2011), pp. 1471–1484. ISSN: 0018-9340.

[146] F. J. Och and H. Ney. "The Alignment Template Approach to Statistical Machine Translation." In: *Computational Linguistics* 30.4 (Dec. 2004), pp. 417–449. ISSN: 0891-2017.

[147] F. J. Och, C. Tillmann, H. Ney, et al. "Improved alignment models for statistical machine translation." In: *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*. Association for Computational Linguistics, 1999, pp. 20–28.

[148] E. Ohlebusch and S. Gog. "A Compressed Enhanced Suffix Array Supporting Fast String Matching." In: *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*. SPIRE '09. Saariselkä, Finland: Springer-Verlag, 2009, pp. 51–62. ISBN: 978-3-642-03783-2.

[149] E. Ohlebusch, J. Fischer, and S. Gog. "CST++." In: *Proceedings of the 17th International Conference on String Processing and Information Retrieval*. SPIRE'10. Los Cabos, Mexico: Springer-Verlag, 2010, pp. 322–333. ISBN: 3-642-16320-3, 978-3-642-16320-3.

[150] R. Pagh. "Low Redundancy in Static Dictionaries with O(1) Worst Case Lookup Time." In: *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*. ICAL '99. London, UK, UK: Springer-Verlag, 1999, pp. 595–604. ISBN: 3-540-66224-3.

[151] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. "BLEU: A Method for Automatic Evaluation of Machine Translation." In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL '02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318.

[152] S. J. Puglisi, W. F. Smyth, and A. Turpin. "Inverted Files Versus Suffix Arrays for Locating Patterns in Primary Memory." In: *Proceedings of the 13th International Conference on String Processing and Information Retrieval*. SPIRE'06. Glasgow, UK: Springer-Verlag, 2006, pp. 122–133. ISBN: 3-540-45774-7, 978-3-540-45774-9.

[153] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. "A Taxonomy of Suffix Array Construction Algorithms." In: vol. 39. 2. New York, NY, USA: ACM, July 2007.

[154] M. S. Rahman, C. S. Iliopoulos, I. Lee, M. Mohamed, and W. F. Smyth. "Finding Patterns with Variable Length Gaps or Don't Cares." In: *Computing and Combinatorics: 12th Annual International Conference, COCOON 2006, Taipei, Taiwan, August 15-18, 2006. Proceedings*. Ed. by D. Z. Chen and D. T. Lee. Berlin, Heidelberg: Springer, 2006, pp. 146–155. ISBN: 978-3-540-36926-4.

[155] R. Raman, V. Raman, and S. S. Rao. "Succinct Indexable Dictionaries with Applications to Encoding K-ary Trees and Multisets." In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 233–242. ISBN: 0-89871-513-X.

[156] L. M. S. Russo, G. Navarro, and A. L. Oliveira. "Fully Compressed Suffix Trees." In: *ACM Transactions on Algorithms (TALG)* 7.4 (Sept. 2011), 53:1–53:34. ISSN: 1549-6325.

[157] K. Sadakane. "Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array." In: *Proceedings of the 11th International Conference on Algorithms and Computation*. ISAAC '00. London, UK: Springer-Verlag, 2000, pp. 410–421. ISBN: 3-540-41255-7.

[158] K. Sadakane. "Succinct Representations of Lcp Information and Improvements in the Compressed Suffix Arrays." In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 225–232. ISBN: 0-89871-513-X.

[159] K. Sadakane. "Compressed Suffix Trees with Full Functionality." In: *Theory of Computing Systems* 41.4 (Dec. 2007), pp. 589–607. ISSN: 1432-4350.

[160] K. Sadakane. "Succinct Data Structures for Flexible Text Retrieval Systems." In: *Journal of Discrete Algorithms* 5.1 (Mar. 2007), pp. 12–22. ISSN: 1570-8667.

[161] K. Sadakane and G. Navarro. "Fully-functional Succinct Trees." In: *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '10. Austin, Texas: Society for Industrial and Applied Mathematics, 2010, pp. 134–149. ISBN: 978-0-898716-98-6.

[162] G. Salton and C. Buckley. "Term-weighting Approaches in Automatic Text Retrieval." In: *Information Processing and Management* 24.5 (Aug. 1988), pp. 513–523. ISSN: 0306-4573.

[163] P. Sanders and F. Transier. "Intersection in Integer Inverted Indices." In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 71–83.

[164] K.-B. Schürmann and J. Stoye. "An incomplex algorithm for fast suffix array construction." In: *Software: Practice and Experience* 37.3 (2007), pp. 309–329.

[165] L. Schwartz and C. Callison-Burch. "Hierarchical phrase-based grammar extraction in joshua." In: *The Prague Bulletin of Mathematical Linguistics* 93 (2010), pp. 157–166.

[166] R. Sennrich and M. Volk. "MT-based sentence alignment for OCR-generated parallel texts." In: *The Ninth Conference of the Association for Machine Translation in the Americas (AMTA 2010)*. Denver, Colorado, 2010.

[167] M. Simard, G. F. Foster, and P. Isabelle. "Using Cognates to Align Sentences in Bilingual Corpora." In: *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Distributed Computing - Volume 2*. CASCON '93. Toronto, Ontario, Canada: IBM Press, 1993, pp. 1071–1082.

[168] M. Stevenson and Y. Wilks. "Word-sense disambiguation." In: *The Oxford Handbook of Computational Linguistics* (2003), pp. 249–265.

[169] T. Strohman and W. B. Croft. "Efficient Document Retrieval in Main Memory." In: *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '07. Amsterdam, The Netherlands: ACM, 2007, pp. 175–182. ISBN: 978-1-59593-597-7.

[170] I. Thayer, E. Ettelaie, K. Knight, D. Marcu, D. S. Munteanu, F. J. Och, and Q. Tipu. "The ISI/USC MT system." In: *International Workshop on Spoken Language Translation (IWSLT)*. 2004, pp. 59–60.

[171] J. Tiedemann. "News from OPUS-A collection of multilingual parallel corpora with tools and interfaces." In: *Recent Advances in Natural Language Processing*. Vol. 5. 2009, pp. 237–248.

[172] C. Tillmann, S. Vogel, H. Ney, and A. Zubiaga. "A DP Based Search Using Monotone Alignments in Statistical Translation." In: *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*. ACL '98. Madrid, Spain: Association for Computational Linguistics, 1997, pp. 289–296.

[173] C. Tillmann. "A Unigram Orientation Model for Statistical Machine Translation." In: *Proceedings of HLT-NAACL 2004: Short Papers*. Boston, Massachusetts: Association for Computational Linguistics, 2004, pp. 101–104. ISBN: 1-932432-24-8.

[174] C. Tillmann and T. Zhang. "A Localized Prediction Model for Statistical Machine Translation." In: *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. ACL '05. Ann Arbor, Michigan: Association for Computational Linguistics, 2005, pp. 557–564.

[175] A Turpin and A Moffat. "Fast file search using text compression." In: *Australian Computer Science Communications* 19 (1997), pp. 1–8.

[176] E. Ukkonen. "On-line Construction of Suffix Trees." In: vol. 14. 3. Secaucus, NJ, USA: Springer-Verlag New York, Inc., Sept. 1995, pp. 249–260.

[177] D. Varga, P. Halácsy, A. Kornai, V. Nagy, L. Németh, and V. Trón. "Parallel corpora for medium density languages." In: *Recent Advances in Natural Language Processing IV (RANLP)* 292 (2005), pp. 247–258.

[178] S. Vogel, H. Ney, and C. Tillmann. "HMM-based Word Alignment in Statistical Translation." In: *Proceedings of the 16th Conference on Computational Linguistics - Volume 2*. COLING '96. Copenhagen, Denmark: Association for Computational Linguistics, 1996, pp. 836–841.

[179] S. Vogel, Y. Zhang, F. Huang, A. Tribble, A. Venugopal, B. Zhao, and A. Waibel. "The CMU statistical machine translation system." In: *Proceedings of MT Summit*. Vol. 9. 2003, pp. 54–61.

[180] W. Weaver. "Translation." In: *Machine translation of languages* 14 (1955), pp. 15–23.

[181] P. Weiner. "Linear Pattern Matching Algorithms." In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*. SWAT '73. Washington, DC, USA: IEEE Computer Society, 1973, pp. 1–11.

[182] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999. ISBN: 9781558605701.

[183] M. Yamamoto and K. W. Church. "Using Suffix Arrays to Compute Term Frequency and Document Frequency for All Substrings in a Corpus." In: *Computational Linguistics* 27.1 (Mar. 2001), pp. 1–30. ISSN: 0891-2017.

[184] H. Yan, S. Ding, and T. Suel. "Inverted Index Compression and Query Processing with Optimized Document Ordering." In: *Proceedings of the 18th International Conference on World Wide Web*. WWW '09. Madrid, Spain: ACM, 2009, pp. 401–410. ISBN: 978-1-60558-487-4.

[185] N. Yazdani and P. S. Min. "Prefix trees: new efficient data structures for matching strings of different lengths." In: *International Symposium on Database Engineering and Applications*. IEEE. Grenoble,France, 2001, pp. 76–85.

[186] O. F. Zaidan and C. Callison-Burch. "Feasibility of Human-in-the-loop Minimum Error Rate Training." In: *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1 - Volume 1*. EMNLP '09. Singapore: Association for Computational Linguistics, 2009, pp. 52–61. ISBN: 978-1-932432-59-6.

[187] R. Zens and H. Ney. "Improvements in Phrase-Based Statistical Machine Translation." In: *Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*. Boston, MA, USA: Association for Computational Linguistics, 2004, pp. 257–264.

[188] R. Zens and H. Ney. "Efficient Phrase-Table Representation for Machine Translation with Applications to Online MT and Speech Translation." In: *HLT-NAACL*. 2007, pp. 492–499.

[189] Y. Zhang and S. Vogel. "An efficient phrase-to-phrase alignment model for arbitrarily long phrase and large corpora." In: *Proceedings of the 10th Conference of the European Association for Machine Translation*. Citeseer. 2005, pp. 294–301.

[190] Y. Zhang, S. Vogel, and A. Waibel. "Integrated phrase segmentation and alignment algorithm for statistical machine translation." In: *Proceedings of the 2003 International Conference on Natural Language Processing and Knowledge Engineering*. IEEE. 2003, pp. 567–573.

[191] G. Zipf. *Human behavior and the principle of least effort: an introduction to human ecology*. Addison-Wesley Press, 1949.

[192] J. Zobel and A. Moffat. "Inverted Files for Text Search Engines." In: *ACM Computing Surveys* 38.2 (July 2006). ISSN: 0360-0300.

[193] J. Zobel, A. Moffat, and K. Ramamohanarao. "Guidelines for Presentation and Comparison of Indexing Techniques." In: *ACM SIGMOD Record* 25.3 (Sept. 1996), pp. 10–15. ISSN: 0163-5808.