ISBN: 978-989-95618-4-7

Managing and Querying a Bilingual Lexicon with Suffix Trees

Jorge Costa[1], Luis Gomes[1], Gabriel Pereira Lopes[1], and Luis M.S. Russo[2]

[1] Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa (FCT/UNL)
Quinta da Torre. 2829–516 Caparica, Portugal
jorge.costa@fct.unl.pt luismsgomes@gmail.com
gpl@fct.unl.pt

[2] Instituto Superior Técnico – Universidade Técnica de Lisboa (IST/UTL)
Av. Rovisco Pais, 1049-001 Lisboa, Portugal
luis.russo@ist.utl.pt

Abstract. In this paper, we propose a bilingual translation lexicon management system based on generalized suffix trees. Bilingual pairs of expressions (single and multi-word) are stored in two separate suffix trees. Bilingual lexicon entries are defined by setting links between two expressions that translate each other, and thus form translation pairs. The system supports insertion and deletion of individual entries (at any time) as well as bulk insertion. The most interesting features of this system are the monolingual and bilingual miscoverage queries which, for a given expression or pair of expressions, indicates which subexpressions (words or multi-word passages) are not covered by the respective monolingual or bilingual lexicon. The miscoverage results are important for flagging out words and multi-words, out of vocabulary. Moreover, the performance of suffix tree based miscoverage queries, when compared with an alternative implementation using suffix arrays, are 10 times faster.

Keywords: Lexica, Lexicon Management, Generalized Suffix Trees, Machine Translation, Cross-Language Information Retrieval

1 Introduction

A bilingual lexicon is a set of pairs of expressions (word and multi-word [3]) of two different languages that are translations of each other, and thus form translation pairs (a bilingual lexicon entry).

[3] The concept of single or multi-word is more linked to Indo-European languages where blank space is usually used as a word separator. This does not apply to languages such as Chinese, where equivalent concept is linked to single or multi-character expressions as the blank character is not used as a separator. And we are not mentioning agglutinative languages such as Finnish or German where compounds have no blank spaces separating agglutinated parts.

When these translation pairs are automatically extracted and give rise to translation tables [11] or bilingual lexicons [1], it may be necessary to manage acquired translations (removing incorrect translations and inserting new ones). And, since bilingual lexicons have many applications (Machine Translation, translation alignment and Cross-Language Information Retrieval, to mention just a few), it is important to provide efficient access to the lexicon.

For a better understanding of what follows, in this paper we call subexpression of an expression a substring of that expression. For example, for "lexicon management system", the subexpressions are "lexicon", "management", "system", "lexicon management", "management system" and "lexicon management system".

We propose a system that supports lexicon management and advanced querying, efficiently. This system can be integrated into existing applications, but more importantly, it provides an innovative functionality, that we called miscoverage, not yet explored to its full potential.

Our management system identifies the subexpressions of an expression [4], that do not (yet) have a translation stored in the lexicon, through the miscoverage queries. Miscoverage can be either monolingual or bilingual. Monolingual miscoverage checks which subexpressions of an expression are not in the corresponding monolingual lexicon. Bilingual miscoverage indicates, for two expressions in different languages, which subexpressions of the pair are not in the lexicon, or which subexpressions do not have a translation in the other language. We present more details and some examples in Section 3.

The miscoverage functions are useful, because they identify which subexpressions are already known as translated and what still needs to be learned as translation, thus guiding the extraction of new (missing) bilingual translation pairs. We have been compiling an English–Portuguese lexicon, by manually validating bilingual pairs automatically extracted from aligned parallel corpora [1, 10]. As the lexicon grows, the miscoverage allow us to spot bilingual pairs that are valuable additions to the lexicon, such as pairs of words or short expressions that were not yet in the lexicon and thus have no coverage, and on the other hand, pairs of expressions (typically longer expressions) that might not be so important because they are covered by existing entries.

As an example, consider that we have just started compiling a bilingual lexicon, containing computer science related expressions, and we have inserted the pair "bilingual"–"bilingue". Next, as we insert the pair "bilingual suffix tree"–"árvore de sufixos bilingue", our system tells us that there is a partial coverage of that pair and, in particular, that the subexpressions "suffix tree"–"árvore de sufixos" are not covered. As a matter of fact, most of the times when we have partial coverage, the uncovered subexpressions are themselves translations of each other and reasonable candidates for integrating the lexicon.

[4] which can be as large as a whole sentence or paragraph

ISBN: 978-989-95618-4-7

In order to efficiently support the management of a bilingual lexicon and inherent operations, we designed a system based on generalized suffix trees [9]. Suffix trees are not commonly used in translation problems. Suffix arrays, full-text indexes, are much more common in the Machine Translation area with several works already published [3, 21], due to its fast access, low memory consumption and easier implementation. In our particular case, the memory consumption was not an issue (the lexicon tested had 200.000 entries), as we wanted to compare the time performance of suffix trees and suffix arrays, prior to tackling memory problems.

To support the particular queries implemented in the experimental context, suffix trees have shown to be a better option (see Section 4). We do not expect that a common lexicon will only have 200.000 entries, but our first approach was to develop a system to manage the entries and return the miscoverage information efficiently, without taking in account the space. To extend our system to larger lexicons and to more than two languages, it is necessary to improve the suffix trees using approaches like compression (see Future Work).

We tested our solution against suffix arrays, to support the benefits of representing a lexicon and defining queries over it, using suffix trees.

In the following sections, we explain the context in which our work is used and present some of the works in the field. Then, we explain in more detail the system implementation and provide some examples and results of a comparison between our system with suffix trees and suffix arrays.

2 Work Context

2.1 Suffix Trees

A suffix tree [9] is a full-text index built on all suffixes of a text $T[1..n]$. A suffix tree has n leaves, each one corresponding to a suffix of T. In Figure 1, we have a suffix tree for the text "abracadabra", with leaf 1 corresponding to suffix "abracadabra", leaf 2 to "bracadabra" and leaf 5 to "cadabra". This numeration defines the suffix represented: leaf 1, for the first suffix (the biggest) and leaf 12 for the smallest suffix (the terminator character). Every text represented in a suffix tree, needs a terminal character to mark the end of the suffix and avoid possible loops. In Figure 1, the parent node of leafs 4, 6 and 11, is an example of the utility of the terminal character, because it allows us to represent the suffix corresponding to the last character of "abracadabra", "a".

Moreover, each tree edge is labeled by a string. The concatenation of these labels, from the root to the leaf, is called path-label. Each path-label coincides with a suffix of T. The leaves are sorted in lexicographic order of the suffixes, like it is shown in Figure 1.

Using the path-label length, every node has a string-depth that is precisely the size of the node's path-label. The node-depth of a node is the existing num-

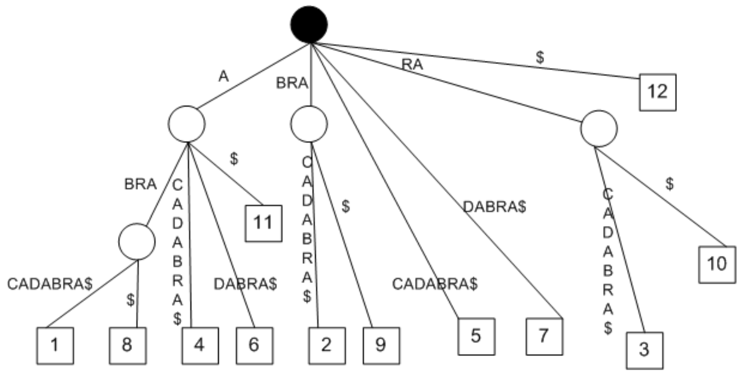


Fig. 1: Suffix tree for "abracadabra", with leaves represented as boxes.

ber of nodes in the path between the root and the node itself. Finally, each internal node has a suffix link, except for the root. A suffix link of an internal node with path-label $x\alpha$, where x is a single character and α a substring (possibly empty), is the node with path-label α. These links prune the search in the case of a mismatch. A mismatch occurs when the next character of the pattern is different from the next character of the edge in which we are descending. This way, it avoids a return to the root, everytime a mismatch occurs.

A suffix tree has $O(n^2)$ nodes [19] and can be built in $O(n)$ time [20, 6].

Generalized Suffix Trees A generalized suffix tree is a suffix tree that is able to index more than one text. Following this idea, we implemented two different generalized suffix trees to manage the bilingual lexicon, because this structure supports more than one expression. Figure 2 shows an example of a generalized suffix tree for the string "magician" and "abracadabra". The different expressions are differentiated in the leaf nodes. In Figure 2, $(1, 2)$ means that it is the leaf for the second bigger suffix of "abracadabra" ("bracadabra"), while $(2, 2)$ refers to the second suffix of "magician" ("agician"). The first numerical reference in the leaf distinguishes different expressions, while the second numerical reference distinguishes the suffixes of the expression of the first reference.

2.2 Suffix Arrays

A suffix array [13, 8] is a permutation of all the suffixes of text T, in which the suffixes are lexicographically sorted. More specifically, a suffix array of a text $T_{1,n}$ is an array $A[1..n]$ of a permutation of the interval $[1, n]$, such that $T_{A[i],n} < T_{A[i+1],n}$, where $<$ stands for a lexicographic order [15]. In Figure 3, there is an example of a suffix array for "abracadabra".

As suffix trees, suffix arrays are also full-text indexes and, much like in suffix trees, they can be used to search for patterns, with the difference that the

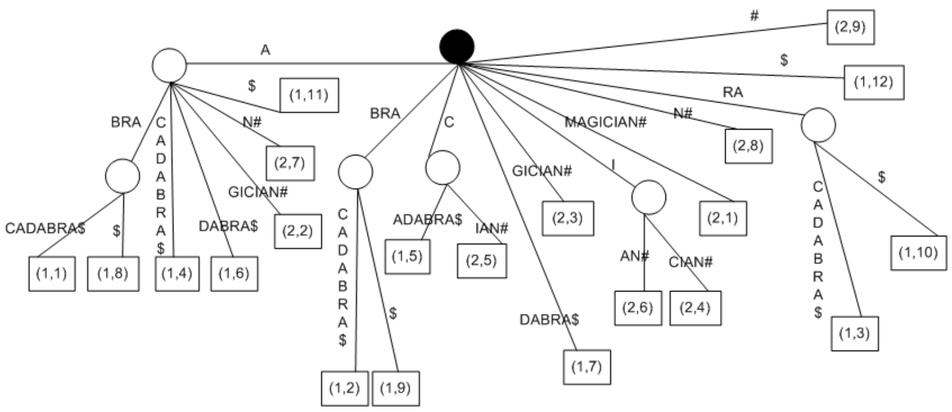


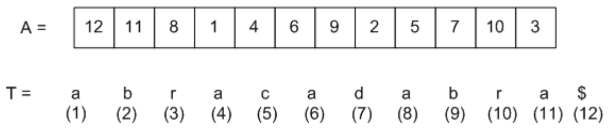
Fig. 2: Generalized suffix tree for "abracadabra" and "magician"

final result is an interval $A[sp, ep]$, in which sp is the starting position and ep the final position. For this reason, suffix arrays were used to compare the performance of the presented management system with suffix trees, especially for the miscoverage queries.

2.3 Bilingual Lexicon and Suffix Trees on Machine Translation

A bilingual lexicon is a core component of any Machine Translation system. In particular, bilingual lexicons are used for decoding [11], aligning parallel texts [7, 16], and extracting translation templates (for Hierarchical Phrase-Based Machine Translation) [12]. Therefore, it is important to provide efficient access to the lexicon in several ways: for decoding, it is necessary to locate lexicon entries in the text to be translated; similarly, for phrase alignment, it is necessary to locate lexicon entries in parallel texts being aligned; and finally, for extracting translation templates it is necessary to locate common sub-expressions among lexicon entries. Using suffix trees, all these operations are performed in optimal time (linear with regard to the size of the input).

In the Introduction Section, we stated that suffix trees are not the most common structure in computational linguistics. The complexity of the construction algorithm and the higher memory consumption are the main factors for not using suffix trees that much. Despite that, the construction is linear and they



A =	12	11	8	1	4	6	9	2	5	7	10	3
T =	a (1)	b (2)	r (3)	a (4)	c (5)	a (6)	d (7)	a (8)	b (9)	r (10)	a (11)	$ (12)

Fig. 3: Suffix Array for "abracadabra$".

ISBN: 978-989-95618-4-7

support complex queries also in linear time [9], which was the focus of our analysis. One of the fields in which suffix trees are frequently used is bioinformatics, where such complex queries are relevant [2].

2.4 Bilingual Suffix Trees

There is an interesting prototype of Bilingual Suffix Trees (BST) [14] that uses generalized suffix trees (GST) to build a corpus of parallel phrases, from comparable non parallel corpora, to extract word unknown translations. A bilingual suffix tree results of matching the GST of a language with the GST of the other language, resulting in a tree with a structure like the one shown in Figure 4.

With the use of a seed bilingual lexicon, the BST defines and stores an alignment between parallel sub-corpora. In Figure 4 we show an example of an alignment, using the bilingual lexicon on the top right, and where the same edge is used to represent parallel expressions from different texts, defining those expressions as aligned. The first edge on the left, leaving the root, shows

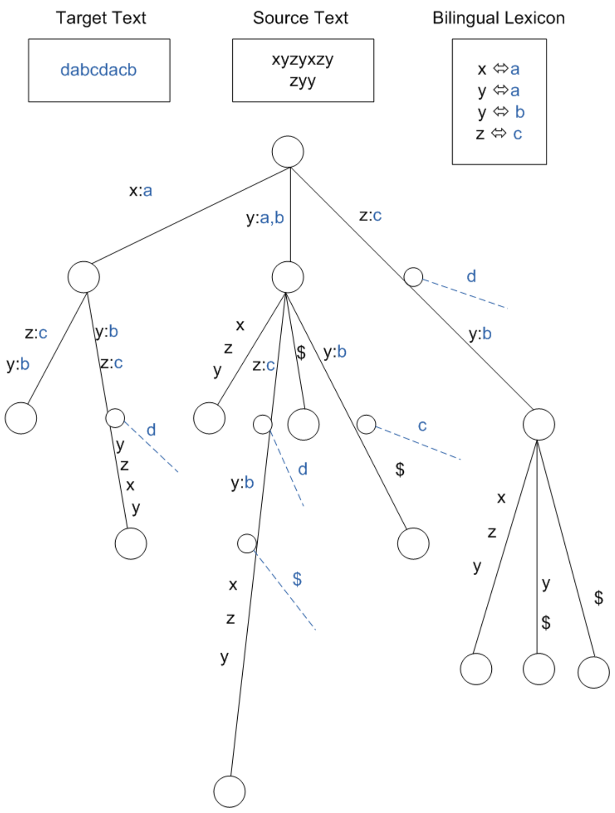


Fig. 4: Example of Bilingual Suffix Tree.

an example of this situation with x from the first corpus, aligned with a from the second corpus.

The BSTs from Munteanu et. al. [14] demonstrate another use of suffix trees in Machine Translation. They are important for the alignment of non parallel corpora, which helps to demonstrate that suffix trees can be a good solution for different machine translation processes, but all the information is in one tree. In our management system, not only we are able to manage and query over a bilingual lexicon, but also define links between translation pairs, using two different GSTs, separating the different languages.

3 Representing the Lexicon

Our representation of the bilingual lexicon uses two generalized suffix trees, one for each language in the system (English and Portuguese in this case), and both are built with an adaptation of the Ukkonen's algorithm [20]. As it is shown in Section 4, suffix links make a significant difference in miscoverage queries and are the main reason for the performance of our implementation.

3.1 Lexicon Management

To represent the expressions in our bilingual lexicon management system, we assume that a word is separated from another word by a blank space. No additional spaces are inserted.

In our management of a bilingual lexicon, we allow adding, removing and linking pairs of expressions. Such pairs may be added either individually (as a single lexicon entry), or as a set. For the case of a single lexicon entry, this means that we consider two expressions that are already a known translation pair, which means that are already linked, or two expressions that are not yet defined as a pair in the lexicon. Following this idea, the most important feature for the management of the lexicon is the correspondence link.

A correspondence link is an abstraction that helps defining a translation pair. We say that two expressions are translation pairs, if they have a correspondence link between them. In Figure 5, we see a representation of the bilingual lexicon in Table 1 using suffix trees, with the correspondence links between the nodes whose path-labels are one side of the translation pair expression. For instance, "bilingual suffix tree" is translated by "árvore de sufixos bilingue", so there is a connection from the respective nodes of both trees. An expression can have more than one link, like "suffix" that can be translated by "sufixo" and "de sufixos". Every expression ends with a terminal character \$, that we always consider as unique even when compared with itself, to distinguish every terminal character of every expression.

The correspondence links are created when we add a pair to the system. If the expressions already exist, then it is only necessary to search for the respective nodes in both trees and add the link between the nodes. If one of the

ISBN: 978-989-95618-4-7

Table 1: Bilingual Lexicon Example

English	**Portuguese**
bilingual	bilingue
bilingual suffix tree	árvore de sufixos bilingue
suffix	de sufixos
suffix	sufixo
suffix tree	árvore de sufixos
tree	árvore

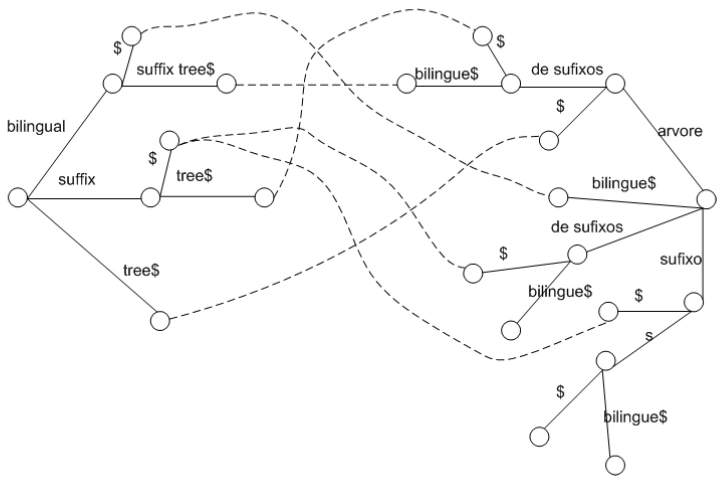


Fig. 5: Correspondence Links

expressions (or both expressions) does not exist, then we change the structure of the tree first, creating the new nodes and labels needed to represent the new expressions, leaving the creation of the link to be done later.

Whenever the tree changes, it is necessary to determine the Depth-First Search (DFS) [5] timestamps for every node. DFS is an algorithm to search a tree, following a methodology of analyzing the nodes by depth. If a node has children, the left child is visited first and so on, until we reach a node without children. Then, we go back and search the siblings of those nodes. The first timestamp marks the first time we reach a node, while the second marks the second time in which that node is visited, after all its descendants were visited. When we add several pairs, we use a simple DFS search in the complete tree. When a single pair is added, then we only look at the neighbor nodes (parent and siblings) and adjust the timestamps locally. The DFS timestamps are used to identify a node and also to determine an ancestor of a node, which is useful for bilingual miscoverage queries and the main reason for using DFS algorithm.

ISBN: 978-989-95618-4-7

Taking $first$ and $second$ as the DFS timestamps of a node, a node X is ancestor of a node Y if: $X.first \leq Y.first$ and $X.second \geq Y.second$. Using these timestamps, finding an ancestor of a node takes constant time, which is one of the main reasons to the bilingual miscoverage performance, explained in the next section in detail.

The removal of a pair only eliminates the link between the elements. The expressions still remain in the lexicon, but without being a translation pair.

3.2 Miscoverage Queries

Miscoverage queries are the main innovation of the bilingual lexicon management system presented. Through these queries, our lexicon can provide useful information to the extraction of translation pairs, using the information already indexed in the lexicon. This will not only benefit the alignment and extraction, but allows a lexicon growth in each iteration, because with new information provided by miscoverage about non known translations in either side of a pair of translation expressions, these processes can also support and guide the acquisition of new and important information to be stored in the lexicon.

In the next subsections, we describe the implementation and give some examples of the miscoverage queries, at monolingual and bilingual level. We will use the bilingual lexicon example in Table 1, as a running example.

Monolingual Miscoverage on a monolingual lexicon is the simplest form of miscoverage. It only analyzes one expression of a single language, leading to a search in only one of the trees of the lexicon. The idea is to determine which subexpressions do not exist in the lexicon. The search is focused on the longest common prefix (descending in the tree for the most consecutive characters possible) we can find. For "bilingual suffix tree" we would descend with "bilingual" and then with "suffix tree", finding these two subexpressions. For "bilingual suffix array", we would descend with "bilingual" and "suffix", but not with "array", meaning that this subexpression is not covered.

The main idea in implementing this operation is to add a blank space, at the beginning of the query, and use the unique terminator character ($) at the end of the query, like in Figure 6 with "suffix tree". Then, we use two integer indexes *j* and *i*. Index *i* is incremented every time we can descend with the next character, while *j* is incremented when we find a mismatch and follow a suffix link. At the end, an expression is covered if its i-th character points to a blank space (or the terminal character) and the j-th character also points to a blank space, with *i* larger than *j*: $i > j$ and $(query[i] = \text{“ ”}$ or $query[i] = \$)$ and $(query[j] = \text{“ ”})$.

Figure 6 shows some examples of different cases, based on the placement of the indexes. In the first example on the left, both indexes point to a blank space, so "suffix" is covered. The second on the left shows "suffix" covered as well, because *j* points to a blank space and *i* pointed to a blank space recently. The third

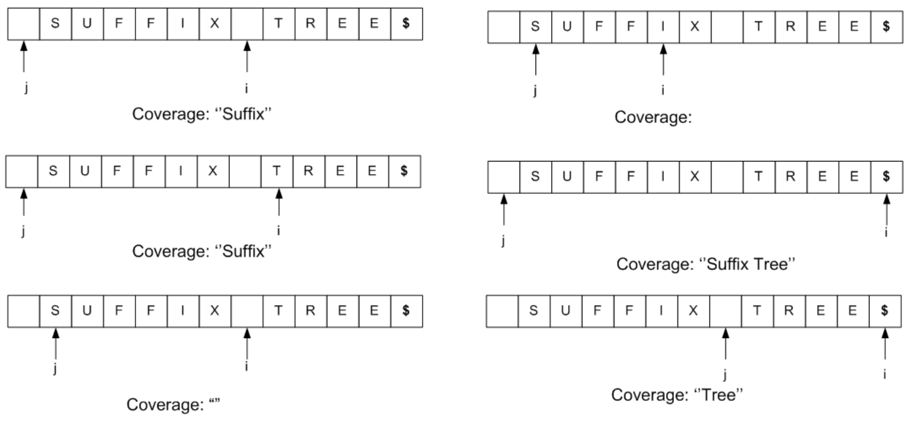


Fig. 6: Examples of monolingual miscoverage situations based on the indexes.

example on the left and the first on the right violate the previous conditions, which means that nothing was found as covered yet. The other two examples follow the restrictions, so the subexpressions between *j* and *i* are covered.

Bilingual Miscoverage is a more complex process. Consider two expressions e_1 and e_2, each one in a different language. First, we determine the monolingual miscoverage of e_1 and e_2, returning two sets, s_1 and s_2, of the nodes whose path-labels are the subexpressions covered. Then, for every node N_1 in s_1, we try to find if one of its correspondence links, points to an ancestor of a node N_2 in s_2. If this happens, then the path-label corresponding to N_1 and N_2 are a translation pair, meaning that those expressions have no bilingual miscoverage, thus are covered.

As an example, considering that we were looking for "suffix link" – "bilingue de sufixos", we were able to find the correspondence link between "suffix" and "de sufixos". The node labeled "suffix" has a correspondence link to the node labeled "de sufixos" (by table 1), which is an ancestor of itself. Thus, these subexpressions have bilingual coverage. The pair "link" – "bilingue" does not have bilingual coverage, because the node for "bilingue" does not have a correspondence link to a node that is ancestor of any node in s_2.

As the first step is similar to the monolingual miscoverage, the main challenge is in the second step. To speed up the analysis of s_1 and s_2, we order s_2 by the first DFS timestamps, using then a binary search. This binary search is used for avoiding nodes that cannot be an ancestor, namely the ones with first DFS timestamps higher than the first timestamp of N_1.

In Table 2 we present some examples of results of the miscoverage queries, based on the lexicon of Table 1. When there is no result presented, means that the expression is fully covered, thus no coverage is found.

ISBN: 978-989-95618-4-7

Table 2: Examples of miscoverage query results

Query	Results
Monolingual miscoverage of “tree”	
Monolingual miscoverage of “bilingual prefix tree”	“prefix”
Monolingual miscoverage of “mono suffix array”	“mono” “array”
Bilingual miscoverage of “red tree” “árvore vermelha”	“red” “vermelha”
Bilingual miscoverage of “bilingual tree” “árvore bilingue”	
Bilingual miscoverage of “linear suffix tree” “árvore de sufixos linear”	“linear” “linear”

4 Results

In this section, we show the time complexity, regarding the operations and construction of the bilingual lexicon. Then, we present some results that compare the solutions in terms of time consumption, with a variable size of the tree and a variable number of operations done in a fixed time interval. With these results, the goal is to demonstrate the benefits of suffix trees when compared with suffix arrays, regarding the implementation of the miscoverage queries.

4.1 Time Complexity

Next, we present the list of variables used to represent the time complexity formula in Table 3, where $|X|$ indicates the length of expression X.

– $\mathbf{e}_A$ – Expression of a language A.
– $\mathbf{e}_B$ – Expression of a language B.
– **e** – Expression in a predefined language.
– **f** – File with the terms to index in the bilingual lexicon.
– **s** – Size of file 'f'.
– $\mathbf{w}_A$ – Number of single words of an expression in the tree for language A.
– $\mathbf{l}_B$ – Number of correspondence links from a node of the tree for language B, to the tree for language A.
– ϵ – Factor that varies from 1 to 2.

In Table 3 we show the time complexities for all main operations. Adding, removing and querying for a pair has the usual linear complexity, based on the length of the expressions. Adding several pairs from a file is fully dependable on the size of the file. The monolingual miscoverage is similar to the querying for a pair, but with only one expression instead of two. The most complex operation is the bilingual miscoverage, with the third expression representing the finding of the pair. l_B represents the starting node set, while the logarithm expression is from finding ancestors. Further to the linear searches for the two expressions, the operation has the factor for analyzing if the results from the search, have known translation pairs, using the ancestor technique.

Table 3: Time Complexities

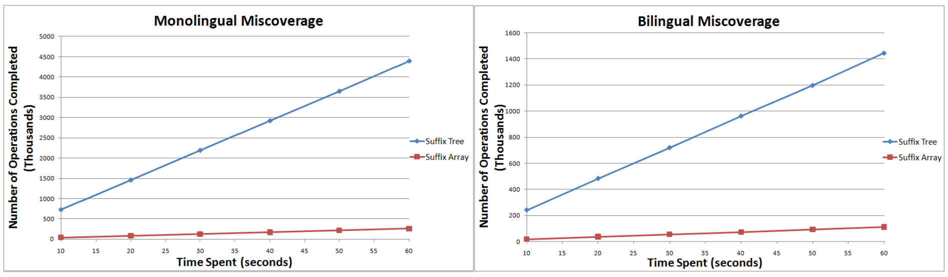
Functionality	Complexity
Add / Remove a single pair to the lexicon	$O(\|e_A\| + \|e_B\|)$
Add multiple pairs to the lexicon from a file	$O(s)$
Check if two expressions form a pair	$O(\|e_A\| + \|e_B\|)$
Monolingual Miscoverage	$O(\|e\|)$
Bilingual Miscoverage	$O(\|e_A\| + \|e_B\| + (l_B^{\epsilon} \times log(w_A)))$

4.2 Comparing Miscoverage with Suffix Trees and Suffix Arrays

One of the biggest challenges on building the bilingual lexicon was to implement the Ukkonen's algorithm. For comparing the performance of our final solution with suffix trees and a solution for the same queries using suffix arrays, we used two approaches. The first compares the performance of the systems, by number of operations done in a fixed time range. The second compares the number of operations done, by varying the size of the tree. The bigger difference in the implementations is the existence of suffix links. With suffix trees, the searches use suffix links, while in the simulated suffix arrays, the search returns to the root, every time it is not possible to descend.

In Figure 7, we present some results of the number of operations completed (yy axis), using both structures, with the time intervals from 10 to 60 seconds in the xx axis. The number of operations grows in both cases, but with suffix trees it is much bigger, especially for a larger range of time, which demonstrates the benefits of the structure for these queries, compared with suffix arrays.

In Table 4, the difference of performance in both structures is still visible, but the number of operations done in 10 seconds, does not vary too much. There is a slight decrease of operations done, when the number of terms in the tree



(a) Monolingual Miscoverage (b) Bilingual Miscoverage

Fig. 7: Number of Miscoverage Operations Done in a Time Range

Table 4: Miscoverage Operations by Tree Size

Number of Elements	Suffix Arrays	Suffix Trees
10.000	25.534	244.326
20.000	24.319	244.895
50.000	22.182	243.884
100.000	20.241	242.987
200.000	19.302	242.471

increases, but it is not a very significant difference, because the search depends almost on the pattern size, and not the tree size. The range for the number of elements in the tree was from 15.000 to 200.000 and the results presented are for the bilingual miscoverage.

By analyzing these results, the improvements that suffix trees provide, when compared to suffix arrays, are 10 times better, which is significant. This happens because of the existence of suffix links, allows the pruning of a search, especially in the miscoverage queries, thus leading to the efficiency demonstrated.

5 Future Work

The focus of our study was the time efficiency of suffix trees, to develop the text mining miscoverage queries, comparing their time performance against suffix arrays. In this study we used a lexicon with 200.000 entries, but for a lexicon with millions of entries, the memory consumption becomes an issue too important to be ignored. Moreover, suffix trees are structures with high space consumption (10 times more than the text), which makes the system inefficient.

In terms of future work, we aim to solve the space issue using compressed indexes [15], namely compressed suffix trees [17, 18, 4]. Using these compressed indexes, the system will consume 90% less memory, without losing the linear characteristics of the suffix trees.

With a system based on compressed indexes, it is our intention to expand it to support 3 or more languages, thus needing 3 or more compressed structures. Having a multilingual lexicon, we need to adapt the miscoverage queries for a multilingual environment. The miscoverage queries will also be improved to provide more complete results and to find miscovered alignment patterns. The correspondence links must also be adapted to support multiple languages. Instead of one correspondence link between trees, the lexicon would have $\frac{N(N-1)}{2}$ links, with N as the number of languages represented.

The implementation of the lexicon opens the possibility of developing higher level applications, like the inference of new translations, using only two aligned expressions and the information from miscoverage. In Figure 8, we know that

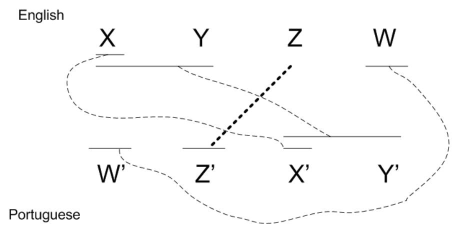


Fig. 8: Inference of the pair Z–Z′ co-occurrence

X, XY and Y are in the English lexicon and that W′, Z′, X′ and X′Y′ are in the Portuguese lexicon, using the monolingual miscoverage. With bilingual miscoverage, we find the pairs X–X′, XY–X′Y′ and W–W′. Using this knowledge, we can infer co-occurrence of the pair Z–Z′, because it is the only part missing and thus support translation extraction. Bilingual miscoverage is also important for the classification of translation pairs [10], because it is an important feature used to that matter.

6 Conclusions

The system implemented accomplishes the goals of finding a way of representing a bilingual lexicon, using suffix trees. These structures are not used so often in linguistics, due to the memory constraints already mentioned in this paper. In this study, we ignored the memory problems to compare our system, in terms of time efficiency with an alternative supported by suffix arrays.

The results were 10 times better, in terms of number of operations done in a period of time, which indicates that suffix trees, when memory is not an issue, are a good solution for managing a bilingual lexicon.

We were able to implement the queries proposed, which benefits the extraction of new translation pairs, thus enabling the lexicon to steadily grow with more translations. This implementation is efficient using suffix trees, which leads to further interesting projects to solve the memory problem, using compressed suffix trees, enabling a solution efficient in terms of high speed querying and low memory consumption.

Acknowledgements The authors would like to acknowledge VIP Access project (Ref. PTDC/PLP/72142/2006) and ISTRION project (Ref. PTDC/EIA-EIA/ 114521/2009) funded by FCT/MCTES that supported the research carried out.

References

1. Aires, J., Lopes, G., Gomes, L.: Phrase translation extraction from aligned parallel corpora using suffix arrays and related structures. Progress in Artificial Intelligence pp. 587–597 (2009)

2. Barsky, M., Stege, U., Thomo, A., Upton, C.: Suffix trees for very large genomic sequences. In: Proceeding of the 18th ACM conference on Information and knowledge management. pp. 1417–1420. ACM (2009)
3. Callison-Burch, C., Bannard, C.: A compact data structure for searchable translation memories. In: EAMT-2005 (2005)
4. Cánovas, R., Navarro, G.: Practical compressed suffix trees. In: Proc. 9th International Symposium on Experimental Algorithms. pp. 94–105. LNCS 6049 (2010)
5. Cormen, T.: Introduction to algorithms. The MIT press (2001), pp. 540–549
6. Farach, M.: Optimal suffix tree construction with large alphabets. In: focs. p. 137. Published by the IEEE Computer Society (1997)
7. Gomes, L., Aires, J., Lopes, G.: Parallel texts alignment. In: New Trends in Artificial Intelligence, 14th Portuguese Conference in Arificial Intelligence, EPIA 2009, Aveiro, October, 2009, Proceedings. pp. 513–524. Universidade de Aveiro (2009)
8. Gonnet, G., Baeza-Yates, R., Snider, T.: Information retrieval: Data structures and algorithms, chapter 3: New indices for text: Pat trees and pat arrays. Prentice Hall, Upper Saddle River, New Jersey 7458, 66–82 (1992)
9. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge Univ Pr (1997)
10. Kavitha, K.M., Gomes, L., Lopes, G.: Using svms for filtering tables for pbsmt. In: EPIA 2011. APPIA, Portuguese Association for Artificial Intelligence (2011)
11. Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., et al.: Moses: Open source toolkit for statistical machine translation. In: Proceedings of the 45th Annual Meeting of the ACL. pp. 177–180. Association for Computational Linguistics (2007)
12. Lopez, A.: Hierarchical phrase-based translation with suffix arrays. In: Proc. of EMNLP-CoNLL. pp. 976–985 (2007)
13. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. In: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms. pp. 319–327. Society for Industrial and Applied Mathematics (1990)
14. Munteanu, D., Marcu, D.: Processing comparable corpora with bilingual suffix trees. In: Proceedings of the ACL-02: Empirical methods in natural language processing-Volume 10. pp. 289–295. Association for Computational Linguistics (2002)
15. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys (CSUR) 39(1), 2 (2007)
16. Och, F., Ney, H.: The alignment template approach to statistical machine translation. Computational Linguistics 30(4), 417–449 (2004)
17. Russo, L., Navarro, G., Oliveira, A.: Fully-compressed suffix trees. ACM Transactions on Algorithms (TALG) (2011), to appear
18. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41(4), 589–607 (2007)
19. Sedgewick, R., Flajolet, P.: An Introduction to the Analysis of Algorithms. Addison-Wesley, Reading (1996)
20. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
21. Yamamoto, M., Church, K.: Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. Computational Linguistics 27(1), 1–30 (2001)