A novel method for finding and scoring valuable translation memory repetitions

Rafał Jaworski

Adam Mickiewicz University
Umultowska 87, 61-614 Poznań, Poland
rjawor@amu.edu.pl

Abstract

This article describes a novel technique in Computer-Aided Translation (CAT) which is meant to be a new generation of translation memory lookup. It combines the benefits of a regular translation memory search (sometimes referred to as fuzzy sentence matching) with the power of a concordance searcher. Input to the search algorithm is a whole sentence, while the output consists of a list of pairwise disjoint fragments from the memory, which cover the input sentence. These fragments may come from different sentences in the memory. As there may exist more than one coverage of the input sentence by the fragments, a specially designed coverage scoring algorithm is implemented. A crucial feature of every CAT mechanism is time performance. For that reason, the implementation of the new search technique uses a custom crafted version of a well known data structure – suffix array.

1. Introduction

Recent decade has shown a significant rise of interest in enhancing the process of human translation with the help of a computer. Computer-Aided Translation has proved to increase the productivity of human translation. As new techniques are constantly being developed, one remains the most fundamental and irreplaceable – translation memory searching. With the use of a translation memory a translator can reuse previously performed translations, which decreases the amount of work and improves consistency of resulting translations.

However, regular translation memory searching scheme involves using the whole sentence as a pattern and retrieving the most similar sentences from the memory. This may result in omitting valuable fragments. For instance, let us suppose that the search pattern is a sentence S, and the memory contains a sentence C_1SC_2, where C_1 and C_2 are contexts of considerable length. This match is either retrieved with a low score or not returned at all.

In order to overcome this shortcoming, this article introduces a new translation memory search algorithm, Concordia, inspired by another CAT technique – concordance searching. The main difference between Concordia and a standard concordancer is the fact that Concordia search uses the whole sentence as a pattern and returns all fragments that cover it. The search algorithm is based on a suffix array index in order to ensure fast lookup times.

Similar research on novel translation memory searching was presented in (Planas, 2005). The SIMILIS software introduced in this work finds syntactical chunks in a translation memory. This, however, requires syntactical analysis of both the search query and the sentences in the translation memory.

Section 2. presents the suffix array data structure, the foundation of the Concordia algorithm. Author's translation memory indexing algorithm is described in Section 3. Section 4. looks into the Concordia search algorithm and presents its speed evaluation. Conclusions and future work plans are listed in Section 5.

2. Suffix arrays

Suffix arrays were introduced back in 1990's (see (Manber and Myers, 1990) and (Nagao and Mori, 1994)). Several years later they were applied to the problem of approximate searching in DNA sequences. This had a vast influence on reviving the research on approximate matching problem in the years 2000-2010. State-of-art solutions in this area are offline searchers, based on indexes.

The survey (Navarro et al., 2000) presents various methods of representing, generating and searching indexes, including methods based on the idea of a suffix array. The article (Ghodsi, 2006) describes an example search algorithm using the suffix array.

2.1. Idea

Suffix array is a data structure designed to hold information about the text to be searched (denoted T). In order to present the structure of a suffix array, let us consider the following example.

Let T ='antananarivo'. This text should be seen as an array of letters, for instance $T[0]$ ='a' and $T[2]$ ='t'.

The suffix array S of the text T is defined as an array of integers corresponding to starting positions of all suffixes of T, sorted in lexicographical order. It is shown in Table 1. For instance, $S[2] = 5$ denotes that lexicographically third suffix (0-based index 2) is the suffix missing the first 5 letters: 'anarivo'.

Finding a pattern P of length m in the text T of length n using the suffix array S is done by performing two binary searches. The first one is done to find the starting position of the searched pattern, the second – to find the end position. According to (Manber and Myers, 1990) this operation consumes $\mathcal{O}(m \cdot \log n)$ time, assuming that comparing suffixes takes m time on average.

However, Manber and Myers in their article of 1990 already suggested ways of improving the search times. The following sections lists selected modern, state-of-art variations of the original idea of suffix arrays.

2.2. Variations

Variations of the classic idea of suffix arrays were developed to enhance the properties of this data structure.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
S[i]	12	3	5	0	7	9	11	4	6	1	2	8	10

Table 1: Suffix array for example text

The biggest impact was put on decreasing the size of the suffix array, hence the research on compressed suffix arrays. Some of these solutions are described below.

2.2.1. Succinct SA

Succinct Suffix Arrays, described in (Makinen and Navarro, 2005), use the ideas of an FM-index and Huffman trees. According to the authors, the size of the index build for the text T of the size n is bounded by $n(H_0(T) + 1)(1 + \mathcal{O}(1))$ bits, where $H_0(T)$ is the 0-th order empirical entropy of T. Counting of the occurrences of a pattern of length p in the text takes $\mathcal{O}(pH_0)$ time.

2.2.2. Compressed SA

Compressed Suffix Arrays (CSA) is a group of algorithms aimed at reducing the size of a suffix array by the means of compression (see (Grossi and Vitter, 2006)). The CSA transforms a suffix array $S[1..n]$ into a sequence of numbers $\psi(i)$ such that $S[\psi(i)] = S[i] + 1$. The properties of the function ψ allow for its compressed representation. Index of this type requires $O(nH_0(T) + n \log \log \sigma)$ space, where σ is the size of the alphabet.

3. Concordia index

This section presents the author's solution for indexing a translation memory. The index is based on suffix array and other auxiliary data structures.

3.1. Overview

Main operations performed on the index are the following:

- $addToIndex(sentence, id)$ This method is used to add a sentence to the index along with its unique id. The id is treated as additional information about the sentence and is then retrieved from the index by the search algorithm. This is useful in a standard scenario, where sentences are stored in a database or a text file, where the id is the line number. Within the addToIndex method the sentence is tokenized and from this point forward treated as a word sequence.

- $generateIndex()$ After adding all the sentences to the index the generateIndex method should be called in order to compute the suffix array for the needs of the fast lookup index. This operation may take some time depending on the number of sentences in the index. Nevertheless, its length rarely exceeds one minute (in reported experiments with 2 million sentences the index generation took 6-7 seconds).

- $simpleSearch(pattern)$ Basic search method takes a text fragment, tokenizes it and tries to locate all its occurrences in the index. The return value is a list of matches, each holding information about the id of the sentence containing the fragment and an offset of this fragment in this sentence.

- $concordiaSearch(pattern)$ The unique search method returns the longest fragments from the index that cover the search pattern.

Functioning of the index is best illustrated by the following simple search example. Suppose that the index contains the following sentences:

- *Novel methods were used to measure the system success rates.* **id=23**

- *Various statistics, including the school success rate, were reported.* **id=12**

- *The research is still ongoing.* **id=259**

Note that the id's of the sentences are not consecutive as there is no such requirement. Let us now search for the fragment "success rate" in the example index. The returned results are: [sentence id = 23, position in sentence (offset) = 8] and [sentence id = 12, offset = 5].

Returned results allow for quick location of the contexts in which the phrase "success rate" appeared. Note also that the system returned the result $(23, 8)$ ("**success rates**") even though the word "rate" was in plural in the index.

Author's index incorporates the idea of a suffix array described in Section 2. and is aided by two auxiliary data structures – the **hashed index** and **markers array**. During the operation of the system, i.e. when the searches are performed, all three structures are loaded in RAM.

When a new sentence is added to the index via the aforementioned $addToIndex$ method, the following operations are performed:

1. tokenize the sentence
2. lemmatize each token
3. convert each token to numeric value according to a dynamically created map (called dictionary)

Lemmatizing each word and replacing it with a code results in a situation, where even large text corpora require relatively few codes. For example, research of this phenomenon presented in (Jaworski, 2013) reported that a corpus of 3 593 227 tokens required only 17 001 codes. In this situation each word could be stored in just 2 bytes, which significantly reduces space complexity.

The following sections will explain in detail the data structures used by the index.

Novel	methods	were	used	to	measure	the	system	success	rates
novel	method	be	use	to	measure	the	system	success	rate
1	2	3	4	5	6	7	8	9	10

Various	statistics	including	the	school	success	rate	were	reported
various	statistic	include	the	school	success	rate	be	report
11	12	13	7	14	9	10	3	15

The	research	is	still	ongoing
the	research	be	still	ongoing
7	16	3	17	18

Table 2: Hashed sentences

3.2. Index – hashed index

Hashed index is an array of sentences in the index. The sentences are stored as code lists. For example, let us compute the hashed index for the sentences of the example index. Results of this process for all three example sentences are shown in Table 2.

Code lists obtained from the sentences are then concatenated in order to form the hashed index. A special code (referred to as EOS – end of sentence) is used as sentence separator. The hashed index is used as the "text" (denoted T) for the suffix array.

3.3. Index – markers array

When a fragment is retrieved from the index with the help of a suffix array, its position is returned as the search result. However, this position is relative to the "text", stored as the hashed index. For example, if we searched for the fragment "success rate", as in previous examples, we would obtain the information about two hits: one at position 8, and the other at position 16 (mind that the positions are 0-based and EOS characters count as single text positions and any punctuation is omitted).

This result does not contain information about the id of the sentence where the fragment was found nor the offset of the fragment in this sentence. Naturally, this information is retrievable from the hash index alone. However, operation of that kind would require searching the hashed index in at least $\mathcal{O}(n)$ time in order to determine which sentence contains the given position. In addition, this would only return the ordinal number of the sentence, not its id, since this information is not stored in the hashed index.

In order to overcome these difficulties, a simple, yet effective data structure was introduced. Markers array is used to store information about the sentence id and offset of each word. Technically it is an array of integers of the length equal to the length of the hashed index. Each integer in the markers array stores both the sentence id and the offset of the corresponding word in the hashed index. Current implementation uses 4-byte integers, where 3 bytes are assigned to store the sentence id and 1 byte is used for the offset. This means the index can store up to 16 777 216 sentences, each no longer than 255 characters (one position is reserved for the EOS character). For example, the pair: $id = (342)_{10} = (101010110)_2$, $offset = (27)_{10} = (11011)_2$ is stored as the integer: $(10101011011011)_2 = (10971)_{10}$.

Even though the markers array is not free from redundancy, the cost space occupied by this data structure is affordable on modern computers. The benefits of its influence on speeding up the search process are much more significant.

3.4. Index – suffix array

The last element of the index is a generated suffix array. It is constructed after the hashed index is complete. Technically, this structure is a classic suffix array for the hashed index. As stated in Section 3.2., hashed index plays the role of the "text" (T), whose "letters" are dictionary codes of words.

Algorithm used for construction of the suffix array is an implementation of classic construction algorithm proposed by Manber and Myers, running in $\mathcal{O}(n \cdot \log n)$ time. It differs significantly from the naive approach (generating suffixes, sorting them and reading their positions) which runs in $\mathcal{O}(n^2 \cdot \log n)$.

Sorted suffixes for the example hashed index result in the following suffix array: [0, 1, 2, 18, 23, 3, 4, 5, 6, 14, 21, 7, 16, 8, 17, 9, 11, 12, 13, 15, 19, 22, 24, 25, 26, 20, 10].

3.5. Simple searching

Searching of the index is done according to the classic suffix array search procedure described in Section 2.1. In order to make this possible, an input search phrase must first undergo the same procedure as a sentence when being added to the index (lemmatizing and coding).

Let us demonstrate the search on the same example search pattern presented in Section 3.1. We are searching for the pattern "success rate" in the example index. The pattern is tokenized and lemmatized, thus transformed into a sequence of lemmas: 'success' 'rate'. These lemmas are then encoded using the dictionary generated during the creation of the hashed index. As a result, the search pattern has the form '9 10'.

By searching in the suffix array with the help of the hashed index we know that the phrase "success rate" can be found in the source text at positions 16 and 8. In fact, we also know that this phrase is present only at these positions, as follows from suffix array properties. However, we expect that the final search results will be given in

the form [*sentence id, offset*]. For that we need to check the markers array. In this example $M[8] = (23, 8)$ and $M[16] = (12, 5)$. This corresponds to the expected final results.

3.6. Big alphabet problem

It must be mentioned that the approach presented above causes a difficulty. Normally, suffix arrays are used to index regular texts over a standard ASCII alphabet. In these cases the size of the alphabet is 255 characters, which means that every character can be stored in just one byte. Classic suffix array implementations depend heavily on the small size of the alphabet and are very sensitive to this property. Meanwhile, the approach presented above uses 4-byte integer values as characters, which brings the alphabet size to 4 294 967 295.

There are search indexes designed to deal with even bigger alphabets (see (Ferragina et al., 2004)). However, in practice, the implementations of classic suffix arrays prove more effective. Therefore, in order to deal with the problem of big alphabet, the following solution was implemented. The 4-byte characters in the text (hashed index) are divided into 4 separate bytes. Each of the bytes is then treated as a separate char. Thus, we obtain a text which is 4 times longer, but consists of considerably shorter characters.

4. Translation memory lookup algorithm – Concordia

4.1. Algorithm details

Concordia search internally uses the simple search procedure but serves for a more complicated purpose. It is aimed at finding the longest matches from the index that cover the search pattern. Such match is called "matched pattern fragment". Then, out of all matched pattern fragments, the best pattern overlay is computed.

Pattern overlay is a set of matched pattern fragments which do not intersect with each other. Best pattern overlay is an overlay that matches the most of the pattern with the fewest number of fragments. The score for this best overlay is computed according to the following procedure. The score is a real number between 0 and 1, where 0 indicates, that the pattern is not covered at all (i.e. not a single word from this pattern is found in the index). The score 1 represents the perfect match – pattern is covered completely by just one fragment, which means that the pattern is found in the index as one of the examples. The formula used to compute the best overlay score is shown below:

$$s = \sum_{f \in O} \frac{len(f)}{len(p)} \cdot \frac{\log len(f) + 1}{\log len(p) + 1} \qquad (1)$$

where:
s – score
f – fragment
O – overlay
p – pattern
$len(f), len(p)$ – number of words in fragment and pattern respectively.

According to the above formula, each fragment covering the pattern is assigned base score equaling the relation of its length to the length of the whole pattern. This concept is taken from the classic Jaccard index. However, this base score is modified by the second factor, which assumes the value 1 when the fragment covers the pattern completely, but decreases significantly, when the fragment is shorter. For that reason, if we consider a situation where the whole pattern is covered with two continuous fragments, such overlay is not given the score 1.

The search procedure itself first tokenizes the input sentence and replaces its words with codes in a manner similar to the $addToIndex$ method. Let us denote the length of input sequence s by m. Then, the following steps are performed:

1. For all i from 0 to $m - 1$
   (a) take the i-th suffix of s
   (b) find those fragments in the index that have the longest common prefix with this suffix
   (c) add at most three of such fragments to the resulting matched pattern fragments set
2. Among the fragments in the matched pattern fragments set find a subset of non-overlapping fragments that maximize the score.

The step 1b) is performed by multiple suffix array binary searches, which first search only the first word of the suffix, then the first two and so on. Each of these steps narrows a continuos fragment of the suffix array, containing those fragments from the translation memory, which start with the first words of the searched suffix.

As a result, a list of matched pattern fragments is obtained. However, these fragments might overlap. For that reason step 2. is performed, which follows an idea of beam search.

4.2. Concordia search example

Let us consider an example illustrating the Concordia search procedure. Let the index contain the following sentences:

- Alice has a cat **id=56**
- Alice has a dog **id=23**
- New test product has a mistake **id=321**
- This is just testing and it has nothing to do with the above **id=14**

The results of Concordia searching for pattern: "Our new test product has nothing to do with computers" are presented in Table 3. Best overlay: $[1, 5]; [5, 9]$, score = 0.53695.

These results list all the longest matched pattern fragments. The longest is $[4, 9]$ (length 5, as the end index is exclusive) which corresponds to the pattern fragment "has nothing to do with", found in the sentence 14 at offset 7. However, this longest fragment was not chosen to the best overlay. The best overlay are two fragments of length

interval	sentence id	offset
$[4, 9]$	14	6
$[1, 5]$	321	0
$[5, 9]$	14	7
$[2, 5]$	321	1
$[6, 9]$	14	8
$[3, 5]$	321	2
$[7, 9]$	14	9
$[8, 9]$	14	10

Table 3: Concordia search results

4: $[1, 5]$ "new test product has" and $[5, 9]$ "nothing to do with". Notice that if the fragment $[4, 9]$ had been chosen to the overlay, it would have eliminated the $[1, 5]$ fragment.

The score of such overlay is 0.53695, which can be considered as quite satisfactory to serve as an aid for a translator.

Concordia search thus makes up for standard translation memory lookup shortcomings. If the above search was performed using standard techniques, it would probably return the results: "New test product has a mistake" and "This is just testing and it has nothing to do with the above" with low scores. These low-scored matches would be discarded by the CAT system (as falling below a given threshold) or ignored by the translator because of insufficient similarity to the pattern. Concordia search results, on the other hand, draw the translator's attention to the coverage of specific fragments of the pattern.

4.3. Speed evaluation

As the Concordia algorithm has so far only been implemented as a search library and not a full CAT system, only speed tests could be run. The speed alone, however, can determine the usability of the solution. It is desired (if not required) that search results are returned to the user in less than a second. For that reason, exhaustive speed test were run on the Concordia library.

Tests were run on a personal computer(1.70GHz CPU, 3GB RAM). The test corpus used as the text to search was taken from the JRC-Acquis corpus (Steinberger et al., 2006). During the test, the Polish version of this corpus was used. The corpus contained 1 917 637 sentences with 20 062 518 words and 139 378 685 characters. Adding all these sentences to index took 6min 18.306s, while generating the suffix array – 6.876s. These figures are very optimistic, considering the fact that real-world translation memories are usually one order of magnitude smaller.

Concordia search times are even more promising. The search experiment consisted in selecting four 10 000 sentences large portions of the JRC-Acquis corpus and using them as Concordia search patterns. The search times for the portions were: 2.475s, 2.325s, 2.575s, 2.293s which is roughly 4000 searches per second.

5. Conclusions and future work

The Concordia search algorithm is designed to be the new generation of translation memory searching. Aside from the unique feature of finding the best overlay of the search pattern, Concordia does not fail to recognize the 100% matches, which is a key property of a translation memory searcher. The new technique proved to perform fast even on large data sets.

Immediate future plans include implementing a CAT system making use of this technique, in order to conduct extensive usability tests.

Other future plans consist in using Concordia searching to perform pre-translation document analysis. Such analysis is performed in order to determine the cost of human translation of a document. Concordia scoring mechanism could be used to compute the total coverage of the document by the translation memory and the searching algorithm could provide data necessary for the analysis in short time.

6. References

Ferragina, Paolo, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro, 2004. Alphabet-friendly fm-index. *In Proc. SPIRE 2004, pp. 150-160, LNCS 3246*.

Ghodsi, Mohammadreza, 2006. Approximate string matching using backtracking over suffix arrays.

Grossi, Roberto and Jeffrey Vitter, 2006. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing, 35(2), pp. 378-407*.

Jaworski, Rafał, 2013. Anubis – speeding up computer-aided translation. *Computational Linguistics – Applications, Studies in Computational Intelligence vol. 458, Springer-Verlag*.

Makinen, Veli and Gonzalo Navarro, 2005. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12(1) pp. 40-66*.

Manber, Udi and Gene Myers, 1990. Suffix arrays: a new method for on-line string searches. *First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319–327*.

Nagao, Makoto and Shinsuke Mori, 1994. A new method of n-gram statistics for large number of n and automatic extraction of words and phrases from large text data of japanese. In *Proceedings of the 15th Conference on Computational Linguistics - Volume 1*, COLING '94. Stroudsburg, PA, USA: Association for Computational Linguistics.

Navarro, Gonzalo, Ricardo Baeza-Yates, Erkki Sutinen, and Jorma Tarhio, 2000. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24:2001.

Planas, Emmanuel, 2005. Similis: Second-generation translation memory software. In *Proceedings of the International Conference Translating and the Computer 27, 24-25 November 2005*.

Steinberger, Ralf, Bruno Pouliquen, Anna Widiger, Camelia Ignat, Tomaz Erjavec, Dan Tufis, and Daniel Varga, 2006. The jrc-acquis: A multilingual aligned parallel corpus with 20+ languages. *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC'2006)*.