UPPSALA
UNIVERSITET

Variable Insertion and Search in a Translation Memory

Jakob Berndtsson

Uppsala University
Department of Linguistics and Philology
Språkteknologiprogrammet
(Language Technology Programme)
Bachelor's Thesis in Language Technology, 15 credits

25th August 2015

Supervisors:
Christian Hardmeier, Uppsala University
Sebastian Schleussner, Convertus AB
Aaron Smith, Convertus AB

Abstract

Translation memories are tools that save and re-use translated text for translators and machine translation systems. This thesis investigates how a translation memory can be improved by implementing variables in it. Eight different variable types divided in three categories are successfully implemented, and through tests their usefulness is shown. Half of the variable types are found in many places in the training data of the tests and are therefore considered useful. The other half is rarer in the training data, but is in most cases considered potentially useful.

To test the implementation, the translation memory is used as a part of a machine translation system, and using a smaller course syllabus corpus and a subset of the JRC-Acquis corpus, it is shown that the introduction of variables improves the translation quality. How big the improvement is depends on several factors, with the biggest one being the number of segments having been inserted in the translation memory prior to translation.

Sammandrag

Översättningsminnen är verktyg som sparar och återanvänder text för översättare och översättningssystem. Den här uppsatsen undersöker hur ett översättningsminne kan förbättras genom att implementera variabler i det. Åtta olika variabeltyper uppdelade i tre kategorier implementeras framgångsrikt, och genom tester visas det hur de användbara de är. Hälften av variabeltyperna hittas på många ställen i träningsdatan av testerna och anses därför vara användbara. Resten är ovanligare i träningsdatan, men är i de flesta fall potentiellt användbara.

För att testa implementationen används översättningsminnet som en del av ett maskinöversättningssystem, och med hjälp av en mindre kursplanskorpus och en delmängd av JRC-Acquis-korpusen visas det hur introduktionen av variabler förbättrar översättningskvaliteten. Hur stor förbättringen är beror på flera faktorer, där den största är antalet segment som har lagts in i översättningsminnet innan översättningen sker.

Contents

Acknowledgements

I would like to thank my supervisors, Christian Hardmeier at Uppsala University and Sebastian Schleussner and Aaron Smith at Convertus AB, along with Anna Sågvall Hein at Convertus AB, for helpful feedback and suggestions, both for the writing and the implementation process. I would also like to thank Lina Stadell at Convertus AB, for general help and suggestions during my time at the company. Finally, I would want to thank Monika Hawrylak for proof-reading, encouraging words and massive support during the entire time of the project.

1 Introduction

Imagine a situation where a translator is manually translating a number of documents, each very similar to the other. The translator can then translate every single segment, even if it is identical to a segment already translated. However, this is not a very effective workflow, and in addition it can also lead to unnecessary work for the translator. To make the workflow more effective, the translator could simply manually copy translated segments from an old document to the new one. This might make the workflow a little more effective, but not by much. To streamline the process, maybe it would be good to try to automate the process?

Enter translation memories. A translation memory makes a translator's life easier by saving everything that has been translated, and then re-using translations whenever a segment identical to a saved segment shows up for translation. This can be of great help, but it is still somewhat limited, since the re-use of the translation only happens if the segment up for translation and the saved segment are identical. It could be meaningful to use this approach even if e.g. a single digit is the only thing differing between two segments.

There are some different techniques for re-using text even when segments are not fully identical. One of the techniques is to use variables in the translation memory. Variables are elements like numbers, email addresses, dates or anything that can be replaced with another element of the same type. With variables, segments still need to be identical, but with variable placeholders in the place of elements, segments up for translation can more often be considered identical with the saved segments, and the saved segments can therefore be re-used more often.

Translation memories can be useful not only for human translators, but also for machine translation systems. If a machine translation system translates a text, the post-edited correctly translated segments can be saved to the translation memory, for later re-use. Since a translation memory therefore helps the machine translation system to find correct approved translations, the quality of the translation can benefit from the use of a translation memory.

1.1 Purpose

The purpose of this thesis is to introduce insertions and searches of variables in a translation memory as part of a machine translation system and examine how this affects the translation quality. While previous research has shown that the introduction of similar techniques in a machine translation system improves its translation quality, there seems to be little research in how specifically the

introduction of variables affects the quality.

This is a gap that I am hoping to fill with this thesis. Unlike other relatively technically complex techniques that requires modifications in many different steps in the machine translation workflow, full matching offers a relatively easy solution that can be implemented by only modifying an existing translation memory.

The translation memory used is part of a rule-based machine translation system developed and utilized by the Uppsala-based language technology company Convertus AB. Although the translation memory can be used for other applications as well, the main application for my development and testing is the Convertus Syllabus Translator, which specializes in translating course syllabi, mainly from Swedish to English.

1.2 Outline

Chapter 2 introduces the concept of translation memories, explains how they can be integrated in machine translation systems, and gives a brief overview of the Convertus Syllabus Translator. Chapter 3 presents eight different variable types and describes how the variables were implemented by adding search and insertion functions to the translation memory of the Convertus Syllabus Translator. In chapter 4, several experiments are performed to test how the introduction of variables to a translation memory affects the translation quality. In chapter 5, the test results are the basis of a discussion of the usefulness of the different variable types and the improvements that come from implementing the variables in the translation memory. Finally, chapter 6 concludes the thesis.

2 Background

2.1 Translation Memory

A Translation Memory (TM) is a database commonly used as an aid for human translators, but can similarly also be used as an aid for machine translation. The database contains already translated source–target segment pairs, which are re-used whenever an identical source segment appears for translation. A TM generally has two parts; insertion and search. Insertions are made when source segments have been translated and the translations have been approved by the user. The pairs of source and target segments then get inserted and stored into the TM. The search is made when a new source segment is being translated. If the new segment *matches* (i.e., is similar enough to) the source of one of the segment pairs in the TM, the new segment automatically gets translated to the target of the same segment pair in the TM. When first used, the TM will naturally be empty, and no matches will be found. However, with time more and more segment pairs are stored, and the potential usefulness of the TM increases. In general, a TM can be a helpful tool for translators, given certain types of text. For example, legal and scientific texts are types of text where the translator might benefit from a TM system, due to the repetitive nature of the texts, while the translation of less repetitive texts, like literary texts, benefits less from it (Gow 2003, p. 14).

Gow (2003, pp. 23–24) specifies three types of matches for TM:

- **Perfect matches**: source segment and TM source segment are identical.
- **Full matches**: source segment and TM source segment are identical, except for some variable elements.
- **Fuzzy matches**: source segment and TM source segment are similar enough (according to some similarity metric).

The concept of perfect matches is as trivial as finding out if two text strings are identical, and one might argue that it is enough. However, if one considers the segments in the following example:

(1) *The course in machine translation starts 19 January.*
The course in machine translation starts 31 August.

The segments in example 1 are not identical, thus they are not a perfect match. They are however similar in that they convey almost identical information, with the date being the only difference, and the similarity can be of use for the translator (or machine translation system). To make them match and increase

the recall of the TM, one of the two techniques full and fuzzy matching (explained in sections 2.1.1 and 2.1.2, respectively) can be implemented.

2.1.1 Full matches and variables

To avoid situations in which only source segments completely identical to the TM segment are matched, one could implement full matching, which means that variables (sometimes called *placeables* or *placeable elements*) are used to match segments that are not completely identical. A variable is an element that is replaceable by another, similar element. One common way to do this is to replace the element with some kind of placeholder in the TM (Azzano 2012, p. 13), and then replace the placeholder with another element of the same type.

For example, if the variables in question are numbers and month names and the English to Swedish segment pair

(2) *The course in machine translation starts 19 January.*
Kursen i maskinöversättning startar 19 januari.

is inserted into the TM, it is saved as

(3) *The course in machine translation starts* `NUM MONTH`.
Kursen i maskinöversättning startar `NUM MONTH`.

where `NUM` and `MONTH` are placeholders for a number and a month, respectively. If the segment

(4) *The course in machine translation starts 31 August.*

later shows up for translation, the TM would find a match for the segment in example 3, and the correct translation (*Kursen i maskinöversättning startar 31 augusti.*) can be given without passing through the hands of a translator or machine translation system.

In his PhD thesis, Azzano (2012, p. 38) specifies two different categories of variables, namely *absolute* and *localizable* variables (or in his terminology, *placeable* and *localizable* elements, respectively). Absolute variables refer to elements that are directly transferred from source to target segment, such as URLs and E-mail addresses. Localizable variables refer to elements that are transferred from source to target via a specified locale. This could be applied to numbers (where decimal and thousands separators can vary for different languages) and dates, and an example of the latter would be, from Swedish (ISO 8601) to British English:

(5) 2015-03-09
09/03/2015

In this thesis, a third category of variables, *lexicalizable* variables, is introduced. These variables are words and phrases that like the variables in the other categories act as variable elements. However, the way they are translated is through a lexicon (hence the term *lexicalizable*). Examples of possible lexicalizable variable types are names of months, and other proper nouns that need translation.

2.1.2 Fuzzy matches

In order to accommodate minor changes to documents to be translated, one might implement full matching in the TM, as is mentioned in section 2.1.1. Another way is to introduce fuzzy matching. With fuzzy matching, segments that are similar but not identical will match each other. For example, if the following source segment is in a TM:

(6) *The course in machine translation starts 19 January.*

and the source segment

(7) *The course in machine translation starts 31 August.*

shows up during translation, it *could* be matched by the segment from example 6, if the similarity between the two segments is large enough. If the segments match, both the TM and translator or MT system are used to translate the segment up for translation.

In modern applications, the similarity is most often measured by using the Levenshtein distance (Azzano 2012, pp. 16–17), which counts the number of substitutions, deletions and insertions needed to change one word to another. The character-based Levenshtein distance between examples 6 and 7 would then be 8, and the word-based distance would be 2. Using a definition from Koehn and Senellart (2010) where the Levenshtein distance is divided by the maximized number of characters or words (46 not counting whitespace and 8, respectively for the examples above) and the resulting quotient is subtracted from 1, examples 6 and 7 would get a similarity score (or fuzzy match score) of approximately 83 % using character-based distance and 75 % using word-based distance.

A fuzzy match is a pair of segments with a similarity score theoretically anywhere higher than 0 % and lower than 100 %. However, to avoid noise and mismatched segments, a threshold is usually set. As an example, the default threshold of the popular TM software SDL Trados Studio is set to 70 %.[1] Using that default value and the similarity score definition mentioned in the previous paragraph, examples 6 and 7 would be considered a match, no matter if character-based or word-based Levenshtein distance was used.

For the parts of the two matched source segments that are identical, the target is taken directly from the target segment of the segment pair in the TM. In examples 6 and 7, this would mean that *The course in machine translation starts* is taken directly from the TM. The parts in the segment that do not match (*31 August* in the examples above) are instead given to the translator or MT system to translate. To know which part of the target segment from the TM that needs to be exchanged by the part translated by the translator or MT system, some kind of word alignment needs to be applied as well.

2.1.3 Translation Memories in Machine Translation

TM and machine translation (MT) have long been separated techniques, partly because of differing translation challenges:

[1] `http://producthelp.sdl.com/SDL%20Trados%20Studio/client_en/Edit_View/TMs/EVWorkingwithTMsAbout_Translation_Memory_Matches.htm`

> While TM have addressed the need of translation agencies to produce high-quality translations of often repetitive material, [MT] has set itself the challenge of open domain translations such as news stories and is mostly satisfied with translation quality that is good enough for gisting, i.e., transmitting the meaning of the source text to a target language speaker. (Koehn and Senellart 2010)

However, in later years, translation agencies have started to look at the integration of MT into their translation workflow, as a way of making the translation process more effective.

In several papers (e.g. Koehn and Senellart 2010; Zhechev and van Genabith 2010) it has been shown that the integration of TM into an MT system can be beneficial to the performance of the system. Unlike the full matching used in this thesis, those works use fuzzy matching in the TM, and with similarity scores 80 % or higher, MT with integrated TM outperforms pure MT. Koehn and Senellart (2010) and Zhechev and van Genabith (2010) both utilize a pipelined approach similar to the one described in section 2.1.2. A more advanced method, with the TM matching integrated into the MT decoder is implemented by Wang et al. (2013), outperforming Koehn and Senellart (2010) at all fuzzy match intervals.

2.2 Convertus Syllabus Translator

The Convertus Syllabus Translator (CST) is a rule-based machine translation system with statistical fallback functionality. It is based on the MATS system, developed at the Department of Linguistics at Uppsala University (Sågvall Hein et al. 2003). MATS was developed as a pure rule-based system, but was later extended with the mentioned statistical fallback (Weijnitz et al. 2004). With the foundation of Convertus AB, MATS was further developed as CST, specialized in translating course syllabi, and became available for public use in 2007 (Pettersson 2010).

The workflow, illustrated in figure 2.1, starts with segmentation of the source text, which divides the text into sentences, headlines, list items, and so on. Next, each segment is passed to the TM. If a match is found, the translated segment is retrieved from the TM and sent directly to the reconstruction step, where all the translated segments are combined to form the full target text. If no TM match is found, the segment is translated by the MT system, and the target segment is automatically post-edited and sent to the reconstruction step. When all source segments have been translated (using either TM or MT) and combined, the full target text is sent to manual post-editing, where the segments are corrected by a human translator. After all corrections have been made and the translated text has been approved, all the source and target segment pairs are inserted into the TM, and the finished translated text is delivered to the user.

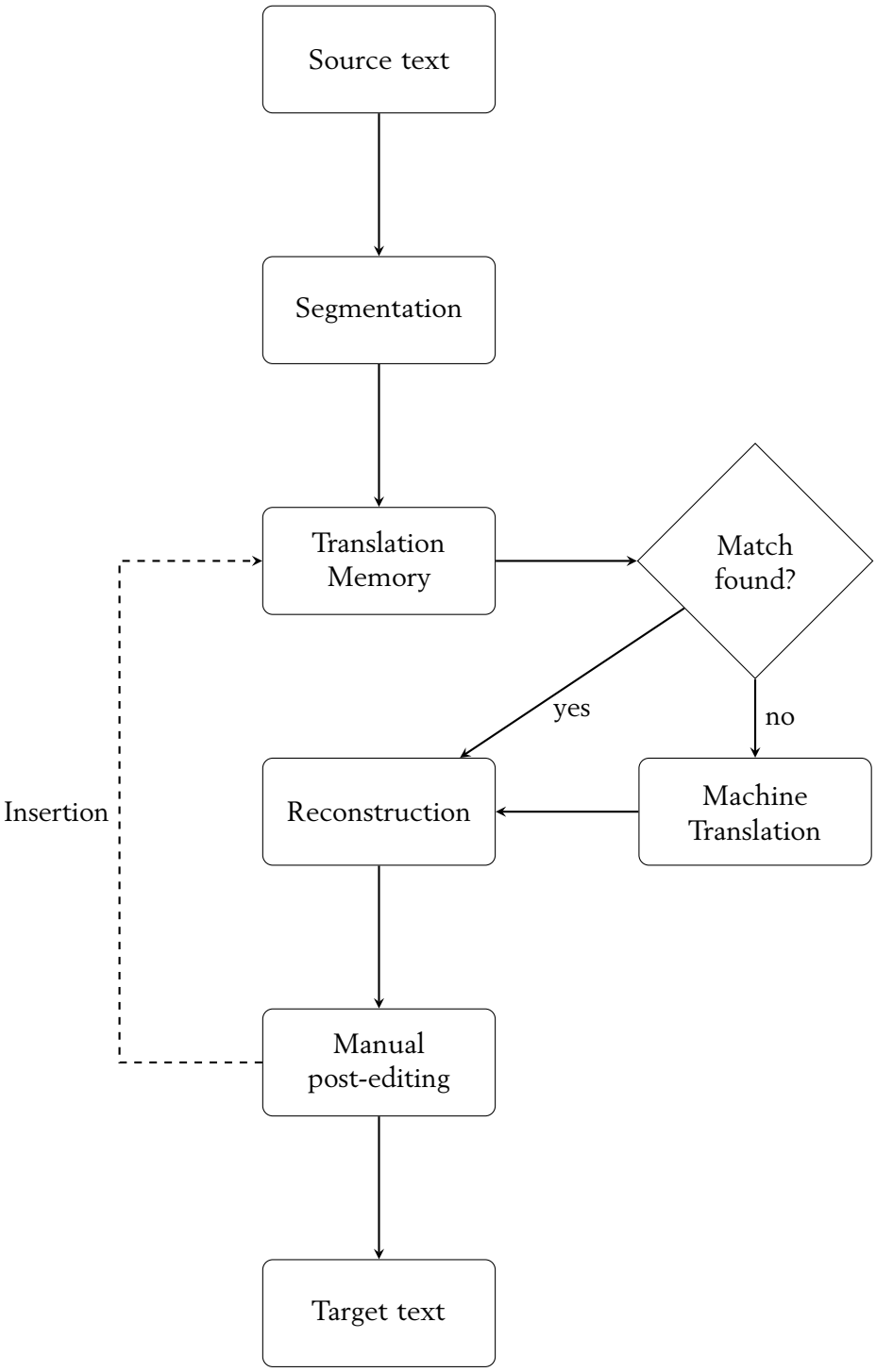

Figure 2.1: A simplified flow chart of the Convertus Syllabus Translator's work flow

3 Implementation

Before the implementation of full matching, the TM of CST could only handle perfect matches, which limits the usefulness of the TM (see section 2.1). To extend the system's functionality, I introduce and implement full matching and eight different types of variables.

At the time of writing this thesis, the languages currently supported by CST are Swedish and English, in both directions. For this reason, and because Swedish and English are the two languages I have the most knowledge of, they are the two languages that the full matching is implemented for. However, there is still some functionality for other languages.[1]

Technically, both variable insertion and search make extensive use of regular expressions. These, along with the development in general, are implemented using Perl, which often is considered unparalleled when it comes to pattern-matching in text (O'Reilly and Smith 1998). The database used for the TM is a MySQL system, which is linked to the Perl modules with the help of the Perl Database Interface (DBI).

3.1 Variables

For both insertion and search, variables are added by replacing the element with a placeholder. The placeholders are in the format `<VARn/>`, where `VAR` is the variable type, and *n* is an identifying number for the variable. The first element of a certain type is given the number 0, the second 1, and so on. For example, with number and month variables, the following segment

(8) *19 January 2015*

will after having the elements replaced by placeholders look like

(9) `<NUM0/> <MONTH0/> <NUM1/>`

My philosophy when defining and implementing the variables is to make sure that as many correct additions of variables as possible are made, but to give priority to precision over recall (i.e., to prefer elements incorrectly not recognized as variables to elements incorrectly recognized as variables). The reason for this is that unlike elements incorrectly not recognized as variables, elements incorrectly recognized as variables can produce incorrect translations. For example, consider the following Swedish-to-English segment pair being inserted into the TM:

[1] The functionality is however limited to languages using the Latin alphabet.

(10) *Får jag åka i maj?*
May I leave in May?

If month variables are incorrectly applied during the insertion into the TM, the segment pair could be stored as

(11) *Får jag åka i* `<MONTH0/>`*?*
`<MONTH0/>` *I leave in May?*

When the new segment *Får jag åka i juli?* is sent through the TM for translation, it will be matched by the source segment in example 11 and produce the incorrect target segment *July I leave in May?*. In comparison, elements incorrectly not recognized as variables can not do this kind of damage: they can only lessen the amount of segments matched by segments in the TM.

For translating the variable elements from source to target language, the same method is used in both insertion and search. However, the method differs between the three different categories of variables. Along with the descriptions of the different variable types of each category, the translation methods for the absolute, localizable and lexicalizable variables are described in sections 3.1.1, 3.1.2 and 3.1.3, respectively.

3.1.1 Absolute variables

Absolute variables are the simplest types of variables, as the values they contain are exactly the same in source and target segments. Thanks to this, no specific language format needs to be used, and the variables can be used in any language, known or unknown.

There are four absolute variable types implemented for this thesis; URLs, email addresses, codes and punctuation.

- **URLs and email addresses**
  The regular expressions used for URLs and email addresses are inspired by the ones presented by Azzano (2012, pp. 149–150 and 154–155). They are both defined with the possibility of letting invalid addresses through. For example, with the URL variable type, an element like `www.p` will be accepted as a valid URL. Since the task is to recognize all valid URLs and email addresses, recall is given a higher priority than with other variable types. Since URLs and email addresses have very specific formats, it is assumed that false placeholder insertions (i.e., insertions of URL or email address placeholders in place of non-URL or email address elements) are highly unlikely, and therefore it is more important to have a more tolerant definition that matches all elements, than having a strict definition that might miss some elements.

- **Codes**
  Codes are defined as an uppercase letter, preceded and followed by zero or more uppercase letters, numbers and dashes. The main motivation behind this variable type is to pick up course codes, but it can also be used for all other codes that fit the definition.

- **Punctuation**
  The punctuation variable type handles sentence-dividing final punctuation, and includes one or more of five common kinds of punctuation: `.`, `!`, `?`, `:` and `;`. The reason for having punctuation as a variable is to make sure that e.g. the segments C*ourse:* and C*ourse.* match. The MT system of CST handles punctuation, but this is no help for the TM, since the TM is placed before the MT system in the translation workflow.

Since the value of an absolute variable is the same in both source and target language, values are simply copied from source to target segment during insertion and search; no modification of the value is needed.

3.1.2 Localizable variables

The values of localizable variables are elements that are translated according to some defined local format. The formats are language-specific, but whenever an unknown language appears (as source or target language), a generic format is used.

- **Dates**
  The date variable type picks up full dates in a numeric format. For Swedish, the ISO 8601 format is used, giving the format as `YYYY-MM-DD`. For English, the format `DD/MM/YYYY` is used. For the generic format, used when a language is not recognized by the system, the ISO 8601 format is used.

- **Numbers**
  In the simplest cases, the number variable type picks up a group of digits. For more advanced cases, thousands and decimal dividers are added, which is also why the variable is localizable, not absolute. For Swedish the format is `I III,FF`, where `I` is the integer part and `F` the fraction part of the decimal number. The same format is also used as the generic format. For the English locale, the format `I,III.FF` is used.

Different methods are used for dates and numbers when changing the format from source to target language. For dates, the different parts of the date (i.e., day, month and year) are individually saved from the source segment, and put into the correct places in the target segment. Unlike dates, no re-ordering of the parts is needed for numbers (i.e., no re-ordering of the integers and fractions), and therefore only the dividers need to be exchanged.

3.1.3 Lexicalizable variables

Lexicalizable variables are translated through a bilingual dictionary, and are, like localizable variables, language-specific. Unlike for the localizable variables, no generic format can be used for the lexicalizable variables when an unknown language is used.

- **Months**
  The values of month variables are simply the names of the different

months, like *January*, *March* and so on. For the English month names, matching is done case-sensitively (with capitalization), and for Swedish month names, matching is done case-insensitively. The case-sensitive matching in English is done to lower the number of incorrect matches with the verbs *may* and *march*.

- **Terms**
  A variable type requested by a customer of Convertus, the term variable is very specific to the domain of course syllabi. It deals with names for school terms, i.e. *spring term* and *autumn term*, along with their definite versions: *the spring term* and *the autumn term*, respectively.

For translating the lexicalizable elements, a bilingual dictionary file is read at the start of the translation of a new text. This file contains the name of the variable type and the source and target elements. An example of a lexicon entry is

(12) MONTH January januari

After the lexicon file has been read, the translation is as easy as replacing the source element (*January*) with the target element (*januari*).

3.2 Insertion

The insertion function, also explained in pseudocode in figure 3.1, starts by iterating over the eight different variable types, looking for occurrences of their respective patterns in the source segment. If a pattern match is found, the respective target language value of the match is generated. For example, if the match in Swedish is the date *2015-01-19*, the English translated value would be *19/01/2015*.

Before the elements recognized as variable elements are replaced by placeholders, a number of tests are made. These tests include determining if the variable is matched also in the target segment, and comparing the target language value acquired with the target pattern for the current variable type. This is especially useful for the punctuation variable type and its segment end anchoring, since if only a check of whether the target language value is present in the segment is done, the punctuation can end up in the wrong place. For example, if one considers the following segment pair:

(13) *Jag talade med fru Robinson häromdagen.*
I spoke to Mrs. Robinson the other day.

If an element is replaced by a placeholder in the target segment without considering the target pattern for the current variable type, the placeholder is simply inserted at the first matched element. Inserting a punctuation variable into the segment pair in example 13 would result in the following incorrect segment pair being inserted in the TM:

(14) *Jag talade med fru Robinson häromdagen*<PUNC0/>
I spoke to Mrs<PUNC0/> *Robinson the other day.*

```
1: function INSERTION(source segment src, target segment trg)
2:     for all variable types do
3:         while src matches source pattern do
4:             if variable type is localizable then
5:                 variable target ← localized source match
6:             else if variable type is lexicalizable then
7:                 variable target ← translated source match
8:             else
9:                 variable target ← source match
10:            end if
11:            if variable type is lexicalizable then
12:                if no. of matches in src ≠ no. of matches in trg then
13:                    next
14:                end if
15:            else if variable type = code then
16:                if trg excludes uppercase letters & spaces then
17:                    next
18:                end if
19:            end if
20:            while trg matches trg pattern do
21:                if trg = variable target then
22:                    substring at target match pos in trg ← placeholder
23:                    substring at source match pos in src ← placeholder
24:                end if
25:            end while
26:        end while
27:    end for
28:    return src, trg
29: end function
```

Figure 3.1: Pseudocode for the insert function.

In addition, a few special cases are dealt with. The first case deals with homographs in segments with lexical variables, as exemplified in examples 10 and 11 in section 3.1. This is solved by not adding the variable if the number of instances of a value is different in the source segment compared to the target segment (e.g., one instance of *maj* vs. two instances of *May* in example 10). The second case deals with false friends. False friends are words that look the same in different languages, but have entirely different meanings. For example, the word *lock*, which means *lock* in English but *lid* in Swedish. This becomes a problem in segments where all letters are in uppercase, since the insertion function then interprets the words as possible code variables. For example, the English to Swedish segment pair

(15) *CHILDREN PLAY IN THE BARN*
BARN LEKER I LADAN

would without any special treatment for cases with false friends be saved in the TM as

(16) *CHILDREN PLAY IN THE* <CODE0/>
<CODE0/> *LEKER I LADAN*

which of course can have negative consequences for later translations. For example, if example 16 is saved in the TM, and the new segment C*HILDREN PLAY IN THE CLASSROOM* turns up, the system would deliver the incorrect target segment C*LASSROOM LEKER I LADAN*. To overcome this problem, no code variables are added in place of a word if the segment contains multiple words and does not contain any lowercase letters. This may not be a perfect solution, but it is an adequate tradeoff, given the improbability of this problem happening in an authentic text.[2]

In the end of the function the segment pair is inserted into the TM. If variables were added to the segments during the insertion, the segments are then inserted in the TM once more, this time without variables. This is done to increase the possibility of finding a match during translation: the more TM entries a segment can match, the greater the possibility that a match will be found.

3.3 Search

The search function is divided in two subfunctions. In the first subfunction, variable elements in the source segment are replaced with placeholders. Following this, the TM database is queried for a segment pair where the TM source segment is identical to the source segment up for translation. If a match is found, the target segment from the database has its placeholders replaced with the correct elements in the second subfunction.

It is necessary to have two subfunctions of the search functions. This is partly because the source segment needs to have its elements replaced by placeholders before the database can be queried, and partly because the target segment needs to be retrieved from the database before the second subfunction can run.

As with the insertion function, both subfunctions of the search function start by iterating over the different variable types. After the iteration they differ greatly, which is further explained in section 3.3.1 and 3.3.2, and in pseudocode in figure 3.2.

3.3.1 Placeholder insertion

In each variable type iteration, all potential variable elements in the source segment are replaced by placeholders. The values replaced by placeholders are saved for later use during the element insertion subfunction (see section 3.3.2). After every variable of a type has been added, the segment with its new placeholders inserted is saved to an array of segment suggestions.

To start with, the segment suggestion array contains the original source segment without placeholders, and various segment suggestions are later added whenever all possible variables of a type have been added to the segment. This means that if the variables added to a segment are (in order) numbers, codes

[2]The problem was found using constructed rather than authentic segments, thus the improbability is assumed.

and punctuation, the segments added to the segment suggestions array will contain the following placeholders:

(17)
- No placeholders
- Numbers
- Numbers and codes
- Numbers, codes and punctuation

Intuitively this might be enough. But if one considers the following Swedish to English segment pair, from one of the corpora from the tests in chapter 4:

(18) *98/78/EG:*
98/78/EC:

After placeholders have replaced elements in the segments during insertion, the segment pair is saved as the following:

(19) `<NUM0/>/<NUM1/>`*/EG*`<PUNC0/>`
`<NUM0/>/<NUM1/>`*/EC*`<PUNC0/>`

In other words, the segments contain number and punctuation variables, but no code variable, even though there is a part that looks like a code. No combination like that is available in the list in example 17, and therefore more segment suggestions need to be added.

Using combinatorics, all possible variable type combinations are calculated and for each combination, the appropriate placeholders are added to the source segment. With segment suggestions containing the variable combinations in example 17 already added, segment suggestions containing the following variable combinations are added in this combinatorics step:

(20)
- Numbers and punctuation
- Codes
- Codes and punctuation
- Punctuation

Only combinations of different variable types are considered, which is a tradeoff. On the one hand, there might e.g. be other code variables somewhere else in the segment that should remain code variables. But on the other hand, if all different combinations of the variables are added, the number of segment suggestions can become too large for efficient performance, since the number of segment suggestions grows exponentially with the number of variables. It was e.g. not uncommon in the testing corpora (see chapter 4) with long segments containing around 15 different numbers. If segment suggestions containing all possible variable combinations of the 15 numbers are added, the number of segment suggestions reaches 32,768, and this number is limited only by the number of variables, which is unknown. However, if only the combinations of different variable types are considered (i.e., that either all numbers or no numbers are replaced by placeholders in the example above), there is a maximum amount of segment suggestions (256).

Additionally, one special case is taken care of. A segment pair like the following is commonly found in the TM after some insertions:

(21) <CODE0/>
<CODE0/>

If an all-uppercase segment like *TRANSLATION* appears for translation from English to Swedish, the search function will suggest that the segment is a code variable, and include the segment <CODE0/> in the segment suggestion list. When querying the database for this segment, a match will be found in the segment pair in example 21. This means that the Swedish target segment incorrectly will be given as *TRANSLATION*. To prevent errors of this kind from happening, no code variables are added to the source segment if the segment contains only uppercase letters, whitespace and/or punctuation.

3.3.2 Element insertion

As mentioned in section 3.3, the database is queried for a target segment in between the two subfunctions of the search function. For the second subfunction, the source and target segments with placeholders, along with the original elements replaced by variables in the placeholder insertion subfunction, are used. In each variable type iteration, the number of variables of the current type in the source segment is checked, and for each variable, the source value is exchanged for the target value, in the way described in sections 3.1, 3.1.2 and 3.1.3. Finally, the current placeholder in the target segment is replaced by the correct element.

Compared to the insertion function and the first subfunction of the search function, this subfunction was the most trivial one, and the only one without any need for solutions of special cases.

3.4 XML entities

In order to avoid situations where parts of any XML entities are caught up by variables and replaced by placeholders in the insertion function and the placeholder insertion subfunction of the search function, the XML entities receive protection when variable elements are replaced by placeholders. This protection replaces XML entities with the placeholder <XML*n*/>, where *n* is an identifying number, before the placeholder substitution is done. After the substitution is done, the XML values are inserted into their correct places again. Without a solution like this, a segment like *Himlen var bl*å. would be changed to *Himlen var bl*&#229<PUNC0/> after replacing variable elements with placeholders, thus losing the last semicolon of the XML entity.

All XML entities in a segment get protection, with one exception: XML entities that are part of the Swedish term variable elements (*vårtermin* eller *hösttermin*). If the TM needs to search for *v*<XML0/>*rtermin* instead of *v*å*rtermin*, no match will be found, and no term variable can be added.

```
1: function PLACEHOLDER INSERTION(source segment src)
2:     org src ← src
3:     push src to segment suggestions
4:     for all variable types do
5:         if variable type = code then
6:             if org src contains only uppercase letters, whitespace and/or
                   punctuation then
7:                 next
8:             end if
9:         end if
10:        while src matches source pattern do
11:            replace variable element with placeholder in src
12:            push variable element to variable sources
13:        end while
14:        push src to segment suggestions
15:    end for
16:    for all variable types found in segment suggestions do
17:        push all variable type combinations to combos
18:        for all combos do
19:            tmp src ← orig src
20:            for all variable types in combo do
21:                replace variable type patterns with placeholders in tmp src
22:            end for
23:            push tmp src to segment suggestions
24:        end for
25:    end for
26:    return segment suggestions
27: end function
```

```
1: function ELEMENT INSERTION(source segment with placeholders src, tar-
   get segment with placeholders trg)
2:     for all variable types do
3:         for no. of variable type placeholders in src do
4:             if variable type is localizable then
5:                 variable target ← localized variable source
6:             else if variable type is lexicalizable then
7:                 variable target ← translated variable source
8:             else
9:                 variable target ← variable source
10:            end if
11:            replace placeholder with variable target in trg
12:        end for
13:    end for
14:    return trg
15: end function
```

Figure 3.2: Pseudocode for the placeholder insertion and element insertion subfunctions of the search function.

4 Experiments

To see how the performance changes when full matching is introduced to CST, some experiments were performed. In short, these experiments consisted of dividing corpora into training and testing data, inserting the training data into the TM, and using the testing data to test the performance.

The corpora used for the experiments are presented in section 4.1, and the testing procedure is explained in detail in section 4.2. Finally, the results of the experiments are presented in section 4.3.

4.1 Corpora

Four different corpora were used for the experiments, all in the same language direction: Swedish to English. Three of them consist of course syllabi in different subject areas: Humanities and Social Studies (`humsam`), Medicine and Pharmacy (`medfarm`) and Science and Technology (`teknat`).

The last corpus is a subset of the Swedish to English part of a version of the

	Segments	Swedish words	English words
`humsam`	1,499	15,029	18,801
`medfarm`	1,500	17,773	22,270
`teknat`	1,500	14,062	17,493
`acquis 1`	5,994	102,065	116,408
`acquis 2`	5,994	100,067	113,712
`acquis 3`	5,994	98,535	112,267
`acquis 4`	5,994	99,764	113,267
`acquis 5`	5,993	101,898	115,610
`acquis 1 sub`	1,500	25,954	29,608

Table 4.1: Statistics of the training corpora.

	Segments	Swedish words	English words
`humsam`	500	4,774	5,978
`medfarm`	500	6,052	7,513
`teknat`	500	4,803	5,961
`acquis`	2,996	50,310	57,478
`acquis sub`	500	8,662	9,991

Table 4.2: Statistics of the testing corpora.

JRC-Acquis corpus (`acquis`) part of the OPUS collection (Tiedemann 2012). The JRC-Acquis corpus consists of legislative texts of the European Union, and of the 792,924 segments in the corpus, 32,965 randomly chosen segments were used for these tests.

After dividing the corpus into testing and training data, the training data was divided into five subsets of roughly the same size (`acquis 1-5`). Additionally, `acquis 1 sub`, a subset of `acquis 1`, was used in order to compare the different types of texts used in the experiments. A subset of the testing data (`acquis sub`) was also used, to investigate the importance of testing data size.

The course syllabus corpora have the advantages of being in the domain for CST and the kind of data a user of this particular system is likely to work with. The disadvantage of these corpora is their size: only around 2,000 segments per corpus, which were divided into training and testing data. The main advantage of the `acquis` corpus is its comparably bigger size and repetitive content, and the main disadvantage is that it consisted of unfamiliar content for CST.

The question of how familiar the system is with the kind of content used in the texts is however of small importance for these tests. The overall performance of the system will obviously be worse for an out-of-domain text, but the focus of this thesis is the TM. Since the TM search is made before the MT decoding in the pipeline (see section 2.2), the actual performance of the isolated TM does not rely on the performance of the entire system.

Statistics for the training and testing corpora are gathered in tables 4.1 and 4.2, respectively.

4.2 Testing procedure

Experiment	Training data	Testing data	Domain
1	`humsam`	`humsam`	`humsam`
2	`medfarm`	`medfarm`	`medfarm`
3	`teknat`	`teknat`	`teknat`
4	`acquis 1 sub`	`acquis sub`	`kursgeneral`
5	`acquis 1 sub`	`acquis`	`kursgeneral`
6	`acquis 1`	`acquis`	`kursgeneral`
7	`acquis 1-2`	`acquis`	`kursgeneral`
8	`acquis 1-3`	`acquis`	`kursgeneral`
9	`acquis 1-4`	`acquis`	`kursgeneral`
10	`acquis 1-5`	`acquis`	`kursgeneral`

Table 4.3: The corpora and domain setting used for each experiment.

The training and testing corpora used for each experiment are described in table 4.3. The training data was first formatted for direct insertion into the TM. In other words, the regular procedure of inserting new entries into the TM of CST, described in section 2.2, was not used here. After the segments had been inserted into the TM, the testing data (or specifically the source language segments of the testing data) was run through CST. For the course syllabus corpora, the MT system was set to a setting corresponding to the respective

subject area of the corpus, and for the `acquis` corpus, the most general setting available (a general course syllabus setting) was used, as is shown in table 4.3.

The results were evaluated using the target language segments of the testing data as gold standard. The evaluation metric used was BLEU, which was calculated with the help of the `multeval` evaluation tool (Clark et al. 2011). Before evaluating, the texts were tokenized using the Sample Tokenizer 1.1, part of the Moses statistical machine translation system.[1]

This entire procedure was done two times for each experiment; one time without variables in the TM, and one time with variables. Using the results of the former system as baseline and comparing them with the results of the latter system, it could be shown how the introduction of full matching affects the performance of CST.

Additionally, CST uses a deterministic MT system. This means that given the same source segment and settings, the target segment that is produced is always the same. Since the translations made by the MT system for a specific corpus are the same during all the tests, only the translations made by the TM will be different between the two testing systems.

4.3 Results

Experiment	`baseline`	`variables`	Increase, %
1	62.21	62.26	0.08
2	63.39	63.47	0.13
3	64.35	64.66	0.48
4	13.89	13.94	0.36
5	14.49	14.70	1.45
6	15.10	15.59	3.25
7	15.47	16.05	3.75
8	15.67	16.30	4.02
9	15.91	16.52	3.83
10	16.11	16.83	4.47

Table 4.4: The BLEU scores of the experiments, along with the percentual increases of the BLEU score between `baseline` and `variables`.

The results of the tests described in section 4.2 are presented in table 4.4, with `baseline` being the system without variables in the TM, and `variables` being the system with variables.

With the smaller corpora used in experiments 1–4, the differences between the systems are very small, with the average increase being approximately 0.26 %. The bigger testing data in experiment 5 made the increase between the two systems go up to 1.45 %, even though the experiment used the same training data as experiment 4. In experiments 6–10 the increase between the two systems generally got bigger as the training data size grew, with the exception of the change of increase between experiments 8 and 9, where the

[1] `http://www.statmt.org/moses/`

increase instead got slightly smaller. The increases for all the experiments are presented along the BLEU scores in table 4.4.

Table 4.5 shows the number of variables inserted into the different training corpora of `variables`. From the `acquis` corpus, only the full corpus (used in experiment 10) is shown. The reason for this is that the results of the subsets does not provide any new valuable information: `acquis 1` have approximately a fifth of the variables in `acquis 1-5` inserted, `acquis 1-2` have approximately two fifths, and so on.

The number of variables in the different corpora gives an idea of the usefulness of the different variable types. The only variable that did not occur at all was the date variable, with the reason being that although dates in the ISO 8601 format were present in the Swedish source text of the training data of the course syllabus corpora, the same format was used in the English target text, and thus no dates were added during the training. And in the `acquis` corpus no dates of the correct formats were present at all during training. The usefulness of the individual variable types is further discussed in section 5.1.

	`humsam`	`medfarm`	`teknat`	`acquis 1-5`
Dates	0	0	0	0
Months	3	0	1	2160
Terms	1	1	4	0
Emails	0	0	0	2
URLs	4	0	8	0
Numbers	224	247	277	29738
Codes	108	159	310	3055
Punctuation	864	924	875	11700

Table 4.5: Occurrences of variables in the different training corpora

With the course syllabus corpora (experiments 1–3) and the full `acquis` corpus (experiment 10), `baseline` and `variables` produced different outputs on a total of 120 segments. These are divided between the corpora as follows:

- Experiment 1: 4 segments
- Experiment 2: 1 segment
- Experiment 3: 6 segments
- Experiment 10: 109 segments

The 11 differing segments from experiments 1–3 are presented in appendix A, while a selection of the differences from experiment 10 is presented and discussed in section 5.2. For the `acquis` corpus, only the results of the full corpus experiment 10 (and not any experiment using only the different subsets) are presented. While other differences were produced using different corpus subsets instead of the entire corpus, I have chosen to look at only the best test results, to limit the amount of data.

Table 4.6 shows how many segments per experiment that have been translated by the TM. While the results are consistent to the results in table 4.4 considering the differences between `baseline` and `variables`, table 4.6 suggests that TM in general is more useful for the `acquis` corpus than for the course syllabus corpora.

When comparing table 4.6 with the number of differing segments, some things might seem odd at a first glance. For example, in experiment 1, 4 and 20 segments were translated by the TM for the `baseline` and `variables` systems, respectively, but only 4 segments differed between the two systems. The explanation for what happened with the other 16 segments is that the MT system in `baseline` and the TM in `variables` in these cases managed to produce identical target segments.

	`baseline`		`variables`	
Experiment	#	%	#	%
1	4	0.80	20	4.00
2	1	0.20	4	0.80
3	7	1.40	17	3.40
4	131	26.20	148	29.60
5	789	26.34	932	31.11
6	878	29.31	1,012	33.78
7	935	31.21	1,060	35.38
8	965	32.21	1,091	36.42
9	985	32.88	1,116	37.25
10	1,001	33.41	1,136	37.92

Table 4.6: The number of TM matches for each test, along with the percentual amount of TM matches in the test data translations.

5 Discussion

The tests made in chapter 4 generally showed a slight increase in translation quality when full matching is introduced to CST. When full matching is introduced in the smaller corpora (experiments 1–5), the increase of the BLEU score is minor, but when it is introduced in the larger corpora experiments (experiments 6–10), the increases are considerably larger. Given the results, the size of the increase depends mostly on the size of the training data, followed by the size of the testing data. The type of text seems to have less importance, at least with the types of text used in the tests of this thesis: the course syllabus corpora and `acquis` corpus produced similar results (given the same amount of training and testing). However, it is worth to notice that TM in general seems to be more useful for the `acquis` corpus.

Considering that more training data means that it is probable that more segments will be matched, it is not surprising that the increase in translation quality is affected by the size of the training data. However, since in translation the quality of a translated document does not depend on the size of the document, it is somewhat unexpected that the system benefits from having bigger testing data. The reason for this could be that having bigger testing data makes it more probable to find matching segments, although this is just a guess.

In this chapter, the usefulness of the different variables types are discussed in section 5.1, some examples of differences between the output of `baseline` and `variables` are explained in section 5.2, and finally a suggestion for improvements of the TM used is given in section 5.3.

5.1 Variable types

The usefulness of the different variable types varied greatly in the experiments in chapter 4. The occurrences of the different variable types in the training data of the corpora, shown in table 4.5, give a hint of their usefulness. As mentioned in section 4.3, no date variables were inserted in the TM. Despite this, it is my belief that the date variable is useful, at least in the case of CST. The reason for this is that when a text containing a date in the correct format for the language is translated, CST delivers the target text with the date translated to the correct format for the language, which will make the date a variable element when inserted into the TM. Thus the date variable is a variable that interacts well with CST.

While the month variable was almost completely absent in the course syllabus corpora, it was one of the most frequent variable types in the `acquis` corpus. This variable might arguably not be of the biggest importance specifically

to CST and the type of texts it usually translates, but in general, month names are words that can occur in all sorts of domains, and I consider the variable type useful.

The term variable is in my opinion the least useful variable type. As mentioned in section 3.1.3, it is a variable specific to the domain of course syllabi, but even in the course syllabus corpora it is not a common variable, with only six occurrences in the 4,499 segments of the training parts of the course syllabus corpora.

Both the email and URL variables were uncommon in the training data. However, as with months, they are words that could appear in many different domains, and therefore have potential usefulness.

The three remaining variable types numbers, codes and punctuation were by far the three most common ones in the training data of all corpora. The number and punctuation variables are very useful, thanks to the fact that numbers can occur in basically any type of text, and that most segments contain some form of ending punctuation.

The code variable is however somewhat problematic. The variable can be very useful, but it is not very stable, and can easily generate incorrect translations. In chapter 3 some special cases for the code variable are handled, but there is a possibility that other scenarios are not foreseen. This possibility exists for all variables, but considering the amount of errors related to the code variable that had to be dealt with during the development, problems are more likely to occur with the code variable than with other variable types.

When it comes to variables it could arguably be a good idea to make them domain-specific. For example, the term variable is unlikely to be found in the JRC-Acquis corpus. Always using all variable types might not cause any damage to the translation, but in terms of runtime, there could be gains found from restricting the number of variable types, since the system then does not need to search for all of them.

5.2 Differences between the two systems

For 109 segments in experiment 10, the two systems produced different results. While the differences between many of these segments are relatively minor (e.g., *of 3 March 2005* vs. *of the 3 March 2005* and *800/1999* vs. *800/1999*,), six of them are presented and discussed in this section. The following abbreviations are used for the examples:

- src: The source segment.
- trg: The gold-standard target segment.
- bas: The output segment of `baseline`.
- var: The output segment of `variables`.

As mentioned in section 4.2, the two systems used the same machine translations for the segments that were not translated through the TM. This means that for all the segments that differed between the systems, at least one of systems generated a translation through the TM. And since all the segments inserted in the TM of `baseline` were also inserted among the segments in the

TM of `variables`, it is reasonable to assume that the system with the segment translated through the TM in most cases (if not all) is `variables`.

(22)
- src: Artikel VI
- trg: Article VI
- bas: Article WE
- var: Article VI

(23)
- src: _BAR_ TOTALSUMMA _BAR_ 13380303 _BAR_ 357784 _BAR_ 13738087 _BAR_
- trg: _BAR_ GRAND TOTAL _BAR_ 13380303 _BAR_ 357784 _BAR_ 13738087 _BAR_
- bas: the TOTALSUMMA of the BAR the BAR 13380303 BAR 357784 BAR 13738087 BAR
- var: _BAR_ GRAND TOTAL _BAR_ 13380303 _BAR_ 357784 _BAR_ 13738087 _BAR_

Most of the time, `variables` offered an improvement over `baseline` to the segments that are different between the two systems. This is particularly visible in examples 22 and 23. In example 22 `baseline` incorrectly analyzed *VI* as an uppercase first-person plural pronoun, instead of a Roman numeral, and translated it to *WE*. On the other hand, the TM of `variables` analyzed the numeral as a code, and produced the correct target segment. In example 23, the target segment produced by `baseline` contains an OOV (out of vocabulary) word and unnecessarily removed underscores, while `variables` produced a segment identical to the gold-standard, thanks to successful matching in the TM.

(24)
- src: med beaktande av Europaparlamentets yttrande (3), och
- trg: Having regard to the opinion of the European Parliament (3),
- bas: considering The European Parliament's statement (3), and
- var: Having regard to the opinion of the European Parliament (3),

(25)
- src: Denna förordning är till alla delar bindande och direkt tillämplig i alla medlemsstater..
- trg: This Regulation shall be binding in its entirety and directly applicable in all Member States.
- bas: This ordinance is to all parts binding and directly applicable in all member states ..
- var: This Regulation shall be binding in its entirety and directly applicable in all Member States.

(26)
- src: (delgivet med nr K(2005) 191)
- trg: (notified under document number C(2005) 191)
- bas: (delgivet with no K(2005) 191)
- var: (notified under document number C(2005) 191)

In other cases, `variables` also produced results identical to the gold-standard. However, the gold-standard target segments are not exactly literal translations of the source segments. The examples in question are examples 24,

25 and 26. In example 24, there is a conjunction present in the end of the source segment which is not present in the gold-standard target segment, in example 25 the source segment ends with two periods whereas the target segment ends with only one period and in example 26 the gold-standard target segment contains some words lacking any equivalent in the source segment. With these conditions it is practically impossible for an MT system to produce a target segment that is identical to the gold-standard.

(27)
- src: Förordning (EEG) nr 1617/93 ändras på följande sätt:
- trg: Regulation (EEC) No 1617/93 is amended as follows:
- bas: (EEC) no 1617/93 be changed in the following ways:
- var: Regulation (EEC) No 1617/93 is hereby amended as follows: 1.

Finally, in example 27, `variables` produced a target segment that is correct (albeit slightly different from the gold-standard target segment), apart from *1.* in the end of the segment, which is not present in the source segment, or any of the other target segments. The reason for this unexpected number and period is likely because of a typo in the TM segment pair matched by the segment in question.

To summarize, the analysis of the experiment 10 examples has shown that while it is not guaranteed that the introduction of full matching will produce correct translations, it is likely that it will make matches in the TM more frequent. And if the segment pairs in the TM are completely incorrect translations, the translations produced by the TM will be completely incorrect.

5.3 Implementing fuzzy matching

One solution to limit the problems of unstable and relatively useless variable types, presented in section 5.1, could be to implement fuzzy matching, introduced in section 2.1.2. With this, some of these variables (e.g., the code and term variables) could be removed, since segments in which they appear instead could be matched by fuzzy matching.

As shown in section 4.3, CST showed some improvements when full matching was introduced, and similarly, as shown in section 2.1.3, when fuzzy matching was introduced to an MT system, the performance of the MT system increased. It is therefore possible that the performance will increase even more when both fuzzy and full matching are implemented. This is a theory that seems to be untested, but since it is common practice in commercial TM software to make use of both techniques (Azzano 2012, pp. 43–44), a reasonable guess is that the combination could be of benefit for the user.

The reason for fuzzy matching not being implemented into the TM of CST is because of the size of the problem. While only a modification of the TM itself is needed for the implementation of full matching, fuzzy matching requires editing of at least the TM, MT system and user interface (to permit the user to set the fuzzy matching threshold). Additionally, some kind of word alignment must be applied to the segments during translation, which further increases the complexity of the issue.

6 Conclusion

In this thesis, full matching has been implemented into a TM part of an MT system, in order to answer the question of how this affects the translation quality. What is clear is that when full matching is introduced to a TM, the translation quality increases. However, the size of the increase depends on some different factors.

The biggest factor is the amount of segments inserted in the data: the more segments inserted, the bigger the chance that a new segment will match something in it. The size of the document up for translation is also a factor of importance, although it is not quite clear why. Lastly the type of text seems to be of less importance. However, this might be because of the properties of the two types of text used in the tests of this thesis: with other types of text the type might be of greater importance.

Having a well-filled TM does not necessarily mean that the translation quality will be high for a new document, even if matches are found for many of its segments. This depends on what has been inserted in the TM: if the segment pairs in the TM are incorrect and/or inconsistent, the translations made by the TM will subsequently follow the same pattern and deliver translations of questionable quality.

Future work could be to implement fuzzy matching alongside the full matching, which might improve the translation quality, compared to having only fuzzy matching or full matching.

Bibliography

Azzano, Dino (2012). 'Placeable and Localizable Elements in Translation Memory Systems'. PhD Thesis. Ludwig Maximilian University of Munich, Center for Information and Language Processing.

Clark, Jonathan H., Chris Dyer, Alon Lavie and Noah A. Smith (2011). 'Better Hypothesis Testing for Statistical Machine Translation: Controlling for Optimizer Instability'. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers - Volume 2*, pp. 176–181.

Gow, Francie (2003). 'Metrics for Evaluating Translation Memory Software'. Master's Thesis. School of Translation and Interpretation, University of Ottawa.

Koehn, Philipp and Jean Senellart (2010). 'Convergence of Translation Memory and Statistical Machine Translation'. In: *AMTA Workshop on MT Research and the Translation Industry*, pp. 21–31.

O'Reilly, Tim and Ben Smith (1998). *The Importance of Perl*. URL: `http://archive.oreilly.com/pub/a/oreilly/perl/news/importance_0498.html` (visited on 27/03/2015).

Pettersson, Eva (2010). *Kursplaneöversättaren - Ett automatiskt översättningsstöd för översättning av akademiska kursplaner från svenska till engelska*. Presentation. Uppsala, Sweden: Convertus AB.

Sågvall Hein, Anna, Eva Forsbom, Per Weijnitz, Ebba Gustavii and Jörg Tiedemann (2003). 'MATS - A Glass Box Machine Translation System'. In: *Proceedings of the Ninth Machine Translation Summit*. New Orleans, USA, pp. 491–493.

Tiedemann, Jörg (2012). 'Parallel Data, Tools and Interfaces in OPUS'. In: *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*. Istanbul, Turkey, pp. 2214–2218.

Wang, Kun, Chenqing Zong and Keh-Yih Su (2013). 'Integrating Translation Memory into Phrase-Based Machine Translation during Decoding'. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*. Sofia, Bulgaria, pp. 11–21.

Weijnitz, Per, Anna Sågvall Hein, Eva Forsbom, Ebba Gustavii, Eva Pettersson and Jörg Tiedemann (2004). 'The machine translation system MATS - past, present & future'. In: *Proceedings of Recent Advances in Scandinavian Machine Translation (RASMAT'04)*. Uppsala, Sweden.

Zhechev, Ventsislav and Josef van Genabith (2010). 'Seeding Statistical Machine Translation with Translation Memory Output through Tree-Based Structural Alignment'. In: *Proceedings of the 4th Workshop on Syntax and Structure in Statistical Translation*. Beijing, China, pp. 43–51.

A Output differences

In this appendix, the segments from experiments 1–3 (the course syllabus corpora) that differed between the output of `baseline` and `variables` are shown. The following abbreviations are used:

- src: The source segment.
- trg: The gold-standard target segment.
- bas: The output segment of `baseline`.
- var: The output segment of `variables`.

Experiment 1 (`humsam`)

1. - src: Delkurs 1.
   - trg: Component 1.
   - bas: Module 1.
   - var: Sub-course 1.

2. - src: Delkurs 3;
   - trg: Module 3;
   - bas: Module 3;
   - var: Sub-course 3;

3. - src: Efter genomförd Delkurs 1 skall studenten:
   - trg: After implemented Module 1, the student should:
   - bas: After completed Module 1, the student should:
   - var: After implemented Module 1, the student should:

4. - src: Efter genomförd delkurs 1 skall studenten:
   - trg: After implemented Module 1, the student should:
   - bas: After completed Module 1, the student should:
   - var: After implemented Module 1, the student should:

Experiment 2 (`medfarm`)

1. - src: Ersätter och motsvarar tidigare kurs 3FX117 Toxikologi, läkemedelsmetabolism och säkerhetsvärdering.
   - trg: Substituting a corresponding earlier course 3FX117 Toxicology, drug metabolism and safety assessment.
   - bas: Substituted and corresponded earlier the 3FX117 of course Toxicology, drug metabolism and saftey assessment.

- var: Substituting a corresponding earlier course 3FX117 Toxicology, drug metabolism and saftey assessment.

Experiment 3 (`teknat`)

1.
   - src: Det får påbörjas tidigast efter uppnådda 24 hp samt då slutbetyg föreligger i relevanta kurser, som berör examensarbetets innehåll
   - trg: [It may be started, at the earliest, after 24 achieved HE credits and when final grades exist in relevant courses that concern the contents of the degree project
   - bas: It may be started at the earliest after achieved 24 credits and then final grade exist in relevant courses that concern the contents of the degree project
   - var: It may be started, at the earliest, after 24 achieved HE credits and when final grades exist in relevant courses that concern the contents of the degree project

2.
   - src: Mål
   - trg: Learning Outcomes
   - bas: Learning outcomes
   - var: Objective

3.
   - src: För studier i årskurs 3:
   - trg: For studies in school year 3:
   - bas: For studies at school year 3:
   - var: For studies in school year 3:

4.
   - src: Produktutveckling, 120 högskolepoäng
   - trg: Product Development, 120 credits
   - bas: Development in production, 120 credits
   - var: Product Development, 120 credits

5.
   - src: Litteraturuppgift 1 hp;
   - trg: Literature assignment 1 hp;
   - bas: Literature assignment 1 credit;
   - var: Literature assignment 1 hp;

6.
   - src: Teori 11 hp;
   - trg: Theory 11 hp;
   - bas: Theory 11 credits;
   - var: Theory 11 HE credits;