2008-05-28

Ranking Search Results for Translated Content

Brian Edwin Hawkins
Brigham Young University - Provo

RANKING SEARCH RESULTS FOR TRANSLATED CONTENT

by

Brian E. Hawkins

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2008

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Brian E. Hawkins

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Christophe Giraud-Carrier, Chair
Date	Sean Warnick
Date	Eric Mercer

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Brian E. Hawkins in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Christophe Giraud-Carrier Chair, Graduate Committee

Accepted for the Department

Date	Parris K. Egbert Graduate Coordinator

Accepted for the College

Date	Thomas W. Sederberg Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

RANKING SEARCH RESULTS FOR TRANSLATED CONTENT

Brian E. Hawkins

Department of Computer Science

Master of Science

Translation Memory (TM) is a valuable tool that helps human translators in doing their job. TM consists of a collection of previously translated texts, called translation units, that may prove useful in the translation of new text. The main problem faced by translators who wish to take advantage of TM is that, although search tools do exist, there is no standardized way of effectively ranking search results. This thesis proposes a method for ranking TM search results together with a novel approach to efficiently finding common substrings that is used in the ranking process.

ACKNOWLEDGMENTS

I would first like to thank my wife for her patience and help as I completed this thesis. I would like to specially thank Dr. Giraud-Carrier for his time and willingness to work with me in completing this work.

I would like to thanks all of my committee. Over the past few months I have learned a lot from the comments and suggestions I received and I am extremely grateful. Thank you all for helping me get to this point.

Contents

Chapter 1

Background

In the language translation service industry human translators have a variety of tools to help aid in the translation process. Human translators use such tools as language dictionaries and machine translations, but the most coveted by content owners is translation memory. Translation memory (TM) is a collection of previously translated texts called translation units. Each translation unit is made up of segments for each language the text is translated into. A segment can be as small as one word or as large as an entire document, but the average size tends to be around one or two sentences.

TM often comes in the form of an XML file with anywhere from a few hundred to as many as a million segments. Human translators use search tools to search through TM while translating new documents. translators try to leverage the existing translated content, but the search tools that are available tend to be inconsistent in ranking search results.

Human translators require different results than most search tools provide. Modern data retrieval techniques tend to focus on finding documents that are similar in content to the search query [3]. For example if I search Google for “I like to run in the Boston Marathon”, I get search hits talking about the Boston Marathon or the Boston Athletic Association. These types of search hits from Google are useless to translators. When a translator searches for “I like to run in the Boston Marathon”, they are not looking for information about the Boston Marathon, they are looking for the exact phrase “I like to run in the Boston Marathon” or some portion of that phrase. When searching through a TM, a translator would like hits for “I like to run” or just “Boston Marathon”, if they are not able to get the entire phrase matched. Partial results aid the translator in piecing together the translation. The problem is that modern ranking techniques such as the Vector Space Model (VSM)[1] [8] do not consider sub phrase matching as part of the ranking algorithm.

Translators want search hits that help them to translate the document. For a translator, the ideal search result is an exact match of the text being translated. Exact matches can be automatically inserted by the translation tool. Unfortunately exact match search results are not very frequent. The problem then becomes to provide the translator enough partial match results so as to aid in the translation of the text.

Historically, a common practice for ranking TM search results is to use the *edit distance metric* [12]. The *edit distance metric* (EDM) is the number of add/delete/modify operations that are required to turn one string into another. Many TM searching tools use the EDM to determine the similarity between the text being translated and the segments found in the TM. One of the flaws with the EDM is when the exact text lies within a segment along with other texts. For example if the translator is searching for “I like to run in the Boston Marathon” and the search hit finds “I like to run in the Boston Marathon, as it keeps me healthy”. The EDM would mark it low because of all of the deletes. Another case to consider when ranking is when the search text is fragmented within the segment like this “**I like to run in the** spring as it gets me ready for the **Boston Marathon**”. This segment needs to be ranked higher than each individual substring alone but lower than an exact match.

[1]In the VSM a phrase is represented by a vector that is created using frequency values of the terms found in the phrase. Similarity between phrases can be found by calculating the cosine of the angle between the two vectors.

It should be noted that although in the above example the substrings “I like to run in the” and “Boston Marathon” appear in order in the search hit, the order of the substrings is not considered.

Simard and Langlais [7] show that by using sub-phrase matching instead of EDM the search tool can provide better results to the translators. The search tool developed in [7] looks for the longest match within the segment, but does not provide a means for ranking the results.

Ranking results is a difficult problem and no two translator search tools do it the same. To solve this problem my solution adopts a formula from the fuzzy set model. The formula in the fuzzy set model is used to determine similarity between two documents. In my case I will use the formula to assign a rank to each segment based on the number and size of each common substring.

In order to efficiently rank results for translators in a timely manner, my solution requires a fast algorithm for finding *all common substrings*. In this paper I propose a solution to the ranking problem as well as introduce a novel algorithm that is able to find matching substrings in linear time for natural language sentences.

Chapter 2

Problem Formulation

Consider some finite alphabet of symbols A. A *string* is an ordered collection of symbols from A given by $s^n := \{(s[1], s[2], \ldots, s[n]) : s[i] \in A, i = 1, \ldots n\}$ and where $n = |s^n|$ is called the *length* of the string.

A set of all strings of length n is given by S^n. The set of all possible strings is given by $S := \cup_{i=1}^{\infty} S^i$. A corpus Υ is a subset of S.

As a clarification, we use the subscript $s_j \in S$ to denote a particular string in S, whereas $s_j[i]$ is the i^{th} symbol in the string s_j and $s_j^n \in S^n$ to be a particular string in S^n. We use the notation $s^l[i,j](s_a, s_b)$ to denote a common substring of length l that begins in the i^{th} position of string s_a and at the j^{th} position of string s_b, however we may omit the notation (s_a, s_b) when the two strings are inferred from the context.

Given two strings s_a and s_b, a common substring is a string $s^l[i,j]$ such that

$$s^l[i,j] := (\{s_a[i], s_a[i+1], \ldots, s_a[i+l-1]\} = \{s_b[j], s_b[j+1], \ldots, s_b[j+l-1]\}).$$

For our purposes we wish to use only the maximal common substrings. A maximal substring at position (i,j) is defined as

$$\bar{s}^l[i,j] := s^l[i,j] : (s_a[i-1] \neq s_b[j-1]) \wedge (s_a[i+l] \neq s_b[j+l]).$$

Let the set $\bar{S}_{a,b}$ be a set of all maximal common substrings between the strings s_a and s_b such that

$$\bar{S}_{a,b} := \{\bar{S}^l[i,j] : 1 \leq l \leq min(|s_a|, |s_b|)\}$$

Given a function $f(s_a, s_b) : s_a \times s_b \rightarrow [0,1]$ that ranks the similarity of the string s_b to s_a, the optimal ranking problem is: given a corpus Υ and string $s_a \in S$, where s_a is an arbitrary string and $s_b \in \Upsilon$, we want to find $max_{s_b \in \Upsilon} f(s_a, s_b)$.

In this work, we consider the function $f(s_a, s_b)$ given by

$$f(s_a, s_b) := (1 - \prod_{s_i \in \bar{S}_{a,b}} (1 - \frac{|s_i|}{|s_a|})).$$

Our definition of $f(s_a, s_b)$ is a directed measure of the similarity of s_b to s_a. By our definition of similarity the string s_b is said to be similar to s_a if it contains all or parts of the string s_a. The string s_b can be similar to s_a, but the opposite may not be true, especially if the string s_b is much larger than s_a. For example say s_a is "I like to run" and s_b is "I like to run in the Boston Marathon". In the previous example s_b is very similar to s_a because it contains the entire string s_a, but s_a is not as similar to s_b because it contains only part of the string of s_b.

As part of this thesis we want to understand how this problem using the measure f corresponds to the results of a human survey. Let the function $\tilde{f}(s_a, s_b)$ will be the result of the human survey. We will survey several translators and have them rank some corpus Υ' and compare those results to f using the following metric. Our metric for measuring the similarity of $f(s_a, s_b)$ and $\tilde{f}(s_a, s_b)$ will consider the top five results using f as compared to the top five results using $\tilde{f}$. Let $T(s_i)$ be a function that returns the number of translators that picked the string s_i. Let $\tilde{f}Score$

be defined as

$$\tilde{f}Score := \sum_{i=1}^{5} T(s_i) : (\forall i : T(s_i) \geq T(s_{i+1}))$$

We will then compute the $fScore$ where

$$fScore := \sum_{i=1}^{5} T(s_i) : (\forall i : f(s_a, s_i) \geq f(s_a, s_{i+1}))$$

The similarity of f with respect to $\tilde{f}$ is calculated as

$$u(f, \tilde{f}) = \frac{fScore}{\tilde{f}Score} \rightarrow [0, 1]$$

Chapter 3

Related Work

The related work is broken into three sections. In Section 3.1 we introduce some other problems that are similar to ours and are often confused with the problems we face in this work.

In Section 3.2 we introduce two solutions for the longest common substring. Finally in Section 3.3 we will introduce some algorithms and concepts that we have adapted to help solve our problem.

3.1 Similar Problems/Research

TM is often used in research, but is not known to be associated with ranking problems. TM is widely used in Example Based Machine Translations (EBMT) [10, 5]. EBMT uses TM to train machine translation engines in attempts to enable the computers to translate in a way that reflects human translators. The goal of EBMT is similar to ours in that the purpose is to provide humans better translations. Our approach, however, is to provide better translations through accurate ranking of TM instead of providing a machine translation.

One of the problems faced when attempting to rank translation search hits is that of finding all of the common substrings between two strings s_m and s_n. This is very similar to the longest common substring problem. The only difference being that we want all of the substrings instead of just the longest.

Another similar problem is that of finding the longest common subsequence. The longest common substring tries to find a common substring that is contiguous in both s_m and s_n. The longest common subsequence however is not concerned with whether the subsequence is contiguous or not. Hunt and Szymanski [6] solved the subsequence problem using an algorithm that runs in $O(nlog(n))$ time. When searching TM we are only concerned with words that are together so that they have the same semantic meaning. Requiring that the words are together rules out the usage of finding common subsequences.

3.2 Related Works

From our research no one has proposed a solution to ranking search hits from TM. We believe our solution to be the first published method for ranking TM search hits in a way that matches what human translators are expecting.

Part of our problem deals with finding the common substrings between s_m and s_n. The problem of finding common substrings has been addressed and we will cover the two main solutions in this section.

3.2.1 Dynamic Programming

Dynamic programming is a technique for solving problems that can be broken down into a set of smaller problems. Programming a solution to these types of problems can be easily done with recursion or nested loops. The downside of dynamic programming solutions is that they tend to run in quadratic time.

As mentioned earlier a problem with the dynamic programming solution is that each time a comparison is done both strings are considered in the computation. Pre-processing one of the strings and using it over and over is not an option. Being able to pre-process one of the strings is important as we compare one string with several segments from the TM in order to rank them for the translator.

3.2.2 Generalized Suffix Tree

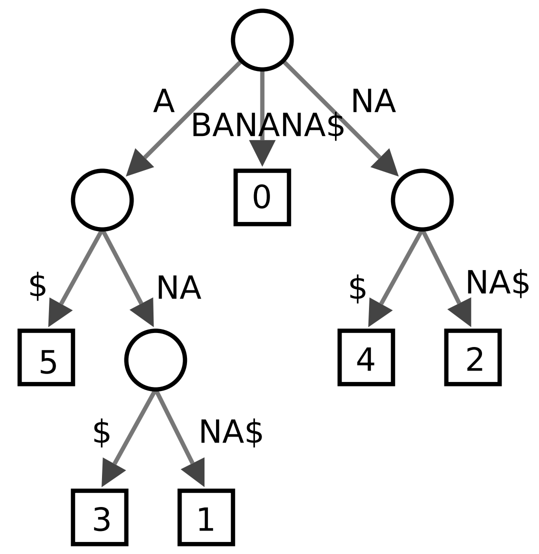

Figure 3.1: Suffix tree of the word "BANANA". Often a special character is appended to the end to mark the end of the string, in this case the string is appended with "$".

A suffix tree is a tree where each branch off of the root represents a suffix of the word or phrase contained within the tree. A suffix tree of the word "BANANA" is shown in Figure 3.1. A suffix tree differs from the suffix trie[1] in that nodes with only one child are collapsed together. The collapsing creates edges with more than one symbol in the label. Collapsing the nodes saves space and enables the tree to be built in linear time. Most often, a symbol is a single character from the phrase, but it does not have to be.

Suffix trees are commonly used for indexing and solving string searching problems [4, 9]. Most people think of the Boyer Moore algorithm [2] when they hear of string searching. The Boyer Moore algorithm is well suited for finding exact matches, but when the length of the potential match is unknown the algorithm does not do well. The suffix tree is well designed for finding partial matches and is often used in

[1]The word trie comes from the word retrieval and it refers to a specifically built tree that is used for data retrieval.

regular expression matching. Suffix trees provide fast look up times and can be built in linear time using non trivial algorithms [4, 11].

A good usage for a suffix tree is searching for a substring. For the tree shown in Figure 3.1 say we are searching for the substring "NAN". Because all possible suffixes of the tree start at the root the search for "NAN" begins there. From the root we see that the substring "NAN" starts in the branch on the right. The edge labeled "NA" contains the first two letters of "NAN", then continuing down the tree the lower edge labeled "NA$" contains the last "N". Because all suffixes start at the root the look up is done in linear time with respect to the size of the substring.

When comparing two strings to find common substrings between them, a suffix tree can be built from the first. A look up is then performed for each symbol in the second string and common substrings are discovered as matches are found.

Our solution for finding the common substrings can also run in linear time, in our applications context and is easier to implement. We will discuss our solution further in Section 5.2.

The following section covers other work that we have borrowed ideas and concepts from to help in solving our problem.

3.3 Fuzzy Set Equation

In fuzzy set theory each member of a set has a degree of membership instead of either included or excluded. The degree of membership is defined as a real value in the interval $[0, 1]$, where 0 is totally excluded and 1 is totally included. The fuzzy set model [1] for Information Retrieval expands on the idea of fuzzy set membership to define the similarity between two documents. In order to define the similarity between two documents a term-term correlation matrix is created. The term-term correlation value is calculated using Equation 3.1. In Equation 3.1, $c_{i,l}$ represents the correlation factor between two terms k_i and k_l where n_i is the number of documents that contain the term k_i and n_l is the number of documents that contain the term k_l and $n_{i,l}$ is the number of documents that contain both terms.

The degree of similarity between two documents is determined by the degree of membership of each term in one document when compared to the terms of the second document. The degree of membership $\mu_{i,j}$ of a term k_i in a given document is calculated as shown in Equation 3.2.

$$c_{i,l} = \frac{n_{i,l}}{n_i + n_l - n_{i,l}} \tag{3.1}$$

$$\mu_{i,j} = 1 - \prod_{k_l \in d_j} (1 - c_{i,l}) \tag{3.2}$$

The degree of membership $\mu_{i,j}$ is calculated as the complement of a negated algebraic product. The term $c_{i,l}$ is the correlation factor obtained from the correlation matrix for the two terms k_i and k_l.

When the correlation $c_{i,l} = 1$ it does not matter as to the correlation of other terms as the degree of membership will be 1. Inversely, if the correlation of two terms is 0 then it does not adversely effect the value of μ.

If we think of the rank of a search hit as synonymous to a degree of membership then this equation becomes very useful for calculating rank. A μ value of 1 is the same as a 100% match. A μ value of 0 is the same as no match. We will discuss other advantages of this equation further in Section 6.1.

Chapter 4

Thesis Statement

Fast accurate ranking of search hits from Translation Memory can be achieved through a combination of efficient domain-specific common substring matching and effective ranking based on a fuzzy similarity function.

Chapter 5

All Common Substrings Algorithm

In our solution we focus on ranking the results provided by searching a TM loaded into the Lucene search engine. Lucene is an open source search engine that is written in Java. Lucene is very fast and does an adequate job of returning an initial set of results that we can use our ranking technique on. Lucene ranks results using the popular VSM [8] model where the weights are provided by using TF/IDF (term frequency/inverse document frequency). Using the result set from Lucene as a starting point works well as it provides search hits with the most terms as well as the rare terms. Terms that are less frequent in the corpus have a higher chance of being harder to translate, because they are not commonly translated and therefore could be more important to the translator.

When discussing search results from TM the query represents the text that is begin translated. The search hits are segments within the TM that are found to have matching words according to the VSM ranking.

The rest of this chapter is laid out as follows: We first introduce, in Section 5.1, a method for query representation that will be used throughout the paper. We then introduce, in Section 5.2, a novel technique for finding partial matches in the search hit. In Section 6.1 we introduce the ranking algorithm and finally in Section 6.2 we describe the filtering technique used to find distinct search results.

A	0, 3
B	1
C	2, 4
D	5

Table 5.1: Contents of $Hash_m$

5.1 String representation

For brevity throughout this paper we will use the following short hand to represent strings of text. The string “I like to read books and read magazines” can be reduced to “A B C D E F D G”. Each letter of the abbreviation is arbitrarily chosen to represent a unique word in the string. The letter D shows up twice, once for each occurrence of the word “read”. This way strings with repeated words can easily be represented without the need to come up with an actual example.

5.2 Partial phrase matching

An important part of our solution is to be able to quickly find common substrings in both the query and the search hits. In this section we describe a novel algorithm for quickly locating all common substrings. The algorithm used here is also applicable to the longest common substring problem [4, 9].

Given two strings s_m and s_n, the common substrings can be represented as $s^l[i,j](s_m, s_n)$. For example take the following strings s_m and s_n where $s_m = \{\mathbf{A}\,\mathbf{B}\,\mathbf{C}\,\mathbf{D}\,E\,\mathbf{B}\,\mathbf{C}\,F\}$ and $s_n = \{G\,\mathbf{A}\,\mathbf{B}\,\mathbf{C}\,\mathbf{D}\,G\,H\}$. The common substrings between s_m and s_n are $s^4[1,2] = \{A\,B\,C\,D\}$ and $s^2[6,3] = \{B\,C\}$.

Our solution is based upon the fact that when matching, terms of s_m and s_n are placed in a matrix (see Figure 5.1), the common substrings form contiguous lines along the diagonals of the matrix. We will locate the common substrings by placing matching terms into queues that represent each of the diagonals. In Figure 5.2 we show a diagram of the matrix with each diagonal labeled.

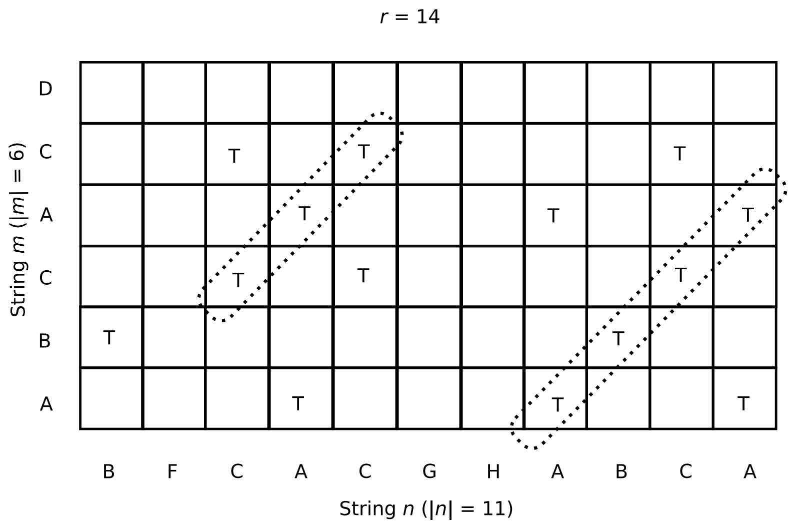

Figure 5.1: Matrix showing matching substrings between strings s_m and s_n.

We start by placing each term of s_m into a hash table along with its position in s_m. We will refer to this hash table as $Hash_m$. For example we will use the string from Figure 5.1 where $m = \{A\,B\,C\,A\,C\,D\}$. After placing the contents of s_m into $Hash_m$ the table will contain the values shown in Table 5.1.

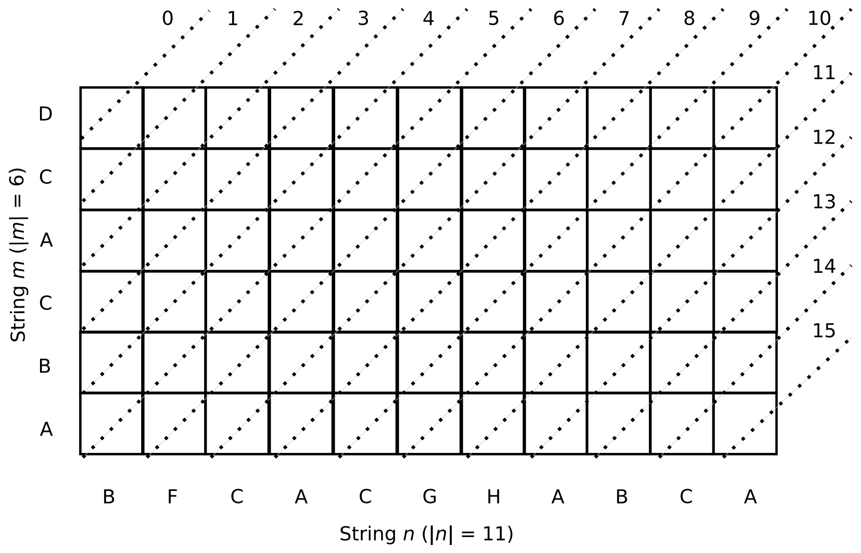

Figure 5.2: Matrix of s_m and s_n with each diagonal numbered.

$$Q = |s_m| - Pos_m + Pos_n - 1 \tag{5.1}$$

Next we consider each term t in s_n. For each t we perform a look up in $Hash_m$ to determine if t exists in s_m. For each occurrence of t in $Hash_m$ we place an entry into the appropriate queue based on Equation 5.1. To understand this process let us consider the first term in s_n, "B". For the first term "B", $Pos_n = 0$ and $Hash_m$ returns $Pos_m = 1$. To calculate which queue term "B" belongs to we input the positions of "B" in s_n and s_m into Equation 5.1 and we get

$$Q = 6 - 1 + 0 - 1 = 4$$

The number of queues that must be maintained at any given point in time while processing s_n is dependent upon the amount of duplicate terms in s_m. In the type of data for this project, duplicate terms tend to be very low, so instead of creating all queues up front we only maintain the queues (the active queues) into which we have placed entries. Maintaining the queues is done by placing the active queues into a hash table, $Hash_q$.

The process of considering each term t_i in s_n starts by performing a look up in $Hash_m$ for the term t_i. If t_i exists in $Hash_m$ then an entry is made into the appropriate queue, as described above. As an entry is added to a queue, the queue is flagged as "updated". After processing t_i all queues that are not flagged as "updated" are removed and placed into a saved list $Save_q$. Queues that are flagged "updated" have the flag removed, but remain in $Hash_q$. After processing all terms in s_n, $Save_q$ will contain a list of all maximal common substrings between s_m and s_n.

The following pseudo code outlines the process described above.

```
**for all** terms $t$ in $s_m$ **do** {Step 1: place values of $s_m$ into hash table}
  $Pos_m(t) \rightarrow Hash_m$
**end for**
**for all** terms $t$ in $s_n$ **do** {Step 2: process each term in $s_n$}
  **for all** $t \in Hash_m$ **do** {Step 2.1}
    Compute the diagonal $i$
```

```
      Insert $t$ into $Q_i$
      Flag $Q_i$ as updated
    end for
    for all active queues $Q_a$ do {Step 2.2}
      if Last term in $s_n$ then
        Remove $Q_a$ and place in $Save_q$
      else if $Q_a$ not updated then
        Remove $Q_a$ and place in $Save_q$
      else
        Remove updated flag
      end if
    end for
  end for
  for all active queues $Q_a$ do {Step 3}
    Remove $Q_a$ and place in $Save_q$
  end for
```

We will prove that the algorithm runs in quadratic time and then show that with our data set the algorithm runs in linear time. We will start by looking at each step in the above code and examine the complexity thereof. In Step 1, the values of s_m are placed into the hash table $Hash_m$. Inserting into a hash table can be done in constant amortized time so this loop can be performed in $O(m)$ time.

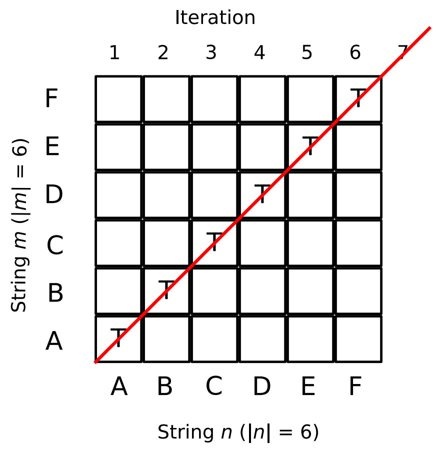

Figure 5.3: Best case scenario. The red lines indicate which are the active queues and in what iteration they are processed. Only one queue is processed in each iteration.

Step 2. The outer loop in this step will run for each term in s_n and performs a look up in $Hash_m$. The first inner loop (Step 2.1) only runs when the term t is found in $Hash_m$. The contents of step 2.1 computes the diagonal on which the correlation

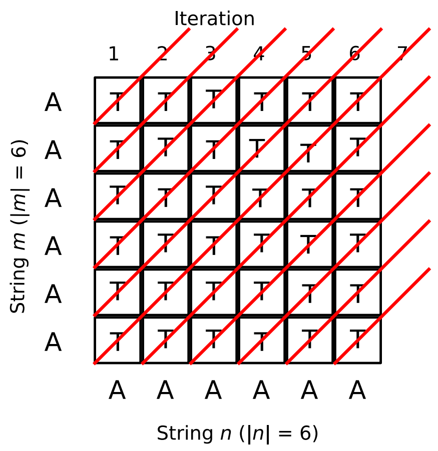

Figure 5.4: Worst case scenario. The red lines indicate which are the active queues and in what iteration they are processed. In each iteration, except the first and last, $m + 1$ active queues are processed.

occurs, performs an insert into a queue and marks the queue as updated, all of which can be done in constant time. In the worst case the loop in step 2.1 will run a total of m times or $O(m)$. The second inner loop (Step 2.2) will run as many times as there are active queues. In the best case for Step 2.2 each correlated term is placed into the same queue as in Figure 5.3 and the loop runs once. In the worst case (See Figure 5.4) the number of active queues can be as many as $m + 1$, so this loop will run in $O(m)$. It follows that Step 2 will run, in the worst case, in $O(m \cdot n)$ time.

Step 3 can be considered as the last iteration of Step 2.2. Step 3 is the cleanup step to remove any remaining active queues and can process as many as m queues. The run time for Step 3 is $O(m)$.

The total running time can be stated as Step 1 $O(m)$ + Step 2 $O(m \cdot n)$ + Step 3 $O(m)$ or the whole thing is $O(m \cdot n)$ and therefore quadratic in the worst case. Looking closer at the algorithm we can see that run time of Step 2.1, 2.2 and 3 are not directly dependent up on m and n but upon the number of correlating terms between s_m and s_n. In Figure 5.1, Figure 5.3 and Figure 5.4 we have represented correlating terms by placing a T in the table. We define the number of T's as $r := |\{\forall(i, j) : s_m[i] = s_n[j]\}|$. In the case of Figure 5.1 $r = 14$. In the best case where s_m an s_n have no correlating terms $r = 0$, but in the worst case where s_m and s_n contain the same repeated term then $r = m \cdot n$. We can restate the speed of Step 2 as $O(n + r)$, even if $r = 0$ the outer loop will run at least n times. The entire process can then be stated as $O(r)$.

Given the nature of our data we can put some bounds on r. One of the bounds we can place is based on the most repeated term in s_m. In general, sentences have very few words that are repeated, if they do these words are usually articles such as "the" and "a". Some odd exceptions to this are the sentences "Buffalo Buffalo, buffalo Buffalo Buffalo, Buffalo Buffalo buffalo" and "James, while John had had 'had', had had 'had had'; 'had had' had had a better effect on the teacher". Fortunately such silly sentences are rare. By analyzing the sentences in our sample TM we find that the most frequent term in a single segment is 'the' and it is used 13 times. From this we can say that $r \leq 13 \cdot n$.

We can still do better if we multiply r by n/n so we get $n \cdot r/n$. The ratio r/n is a value we can empirically sample from our data set, we will define this ratio as $\bar{r} := r/n$. Using the 75 sample search results used in our survey - which we have deemed as good results - the average value of $\bar{r}$ is 0.6382955. Only 9 of the 75 results have a $\bar{r}$ value of greater than 1.

As mentioned above, the total running time of our search algorithm can be stated as $O(r)$. Since, in our data set, $\bar{r} \leq 1$, we have that the total running time is $O(r) = O(\bar{r} \cdot n) = O(n)$, which is linear.

5.3 All Common Substring Speed Comparison

We have shown that for our data set, our common substring algorithm can run in linear time. In this section we will examine some empirical speed comparisons between our algorithm and the suffix tree. We devised three tests that grow the size of the strings being compared. In all three comparisons the timing shown is the result of looping the algorithm one thousand times.

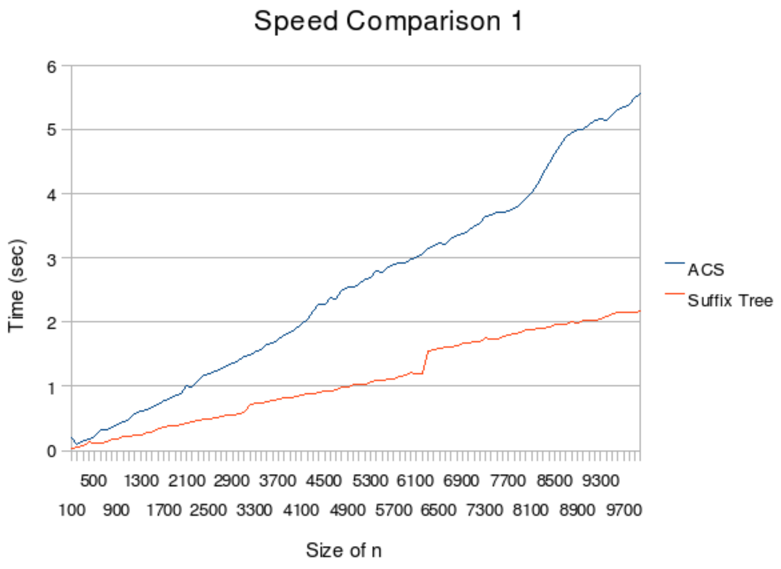

Figure 5.5: Comparison one

In the first comparison (Figure 5.5) we keep the size of s_m constant at 50. To be consistent with our data set we keep the ratio $\bar{r} = r/n = 1$. With both algorithms there are two phases to the comparison. The first phase is that of setting up the data (reading s_m) and the second phase is that of reading s_n and finding substrings. In the first comparison we hold s_m constant so time required to setup the algorithms is constant. The first comparison shows the suffix tree to run in about half the time as our algorithm. This makes sense if we understand how the two algorithms work.

In the second phase our algorithm allocates objects and data structures while building each queue. The suffix tree on the other hand is just reading the tree. In the first comparison the difference between allocating memory and just reading data is apparent.

In the second comparison (Figure 5.6) we keep the size of s_m and s_n equal and again $\bar{r} = 1$. The second comparison includes the time required to setup the data. For our algorithm the time to setup the data is much faster than the suffix tree. In the second comparison the setup time balances out the search time for the substrings and both algorithms run at about the same speed.

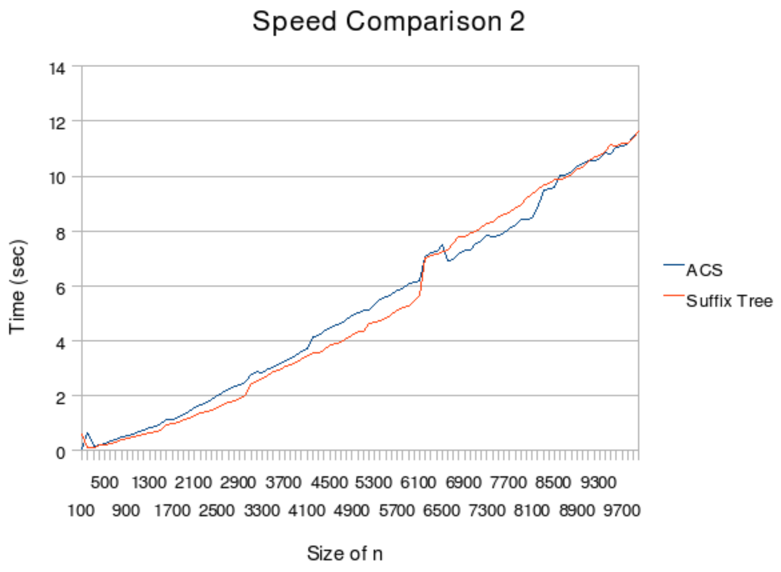

Figure 5.6: Comparison two

The last comparison (Figure 5.7) is the same as the second except we set $\bar{r} = 3$. The third comparison shows a slight curve as our algorithm begins to run in quadratic time whereas the suffix tree continues to run in linear time.

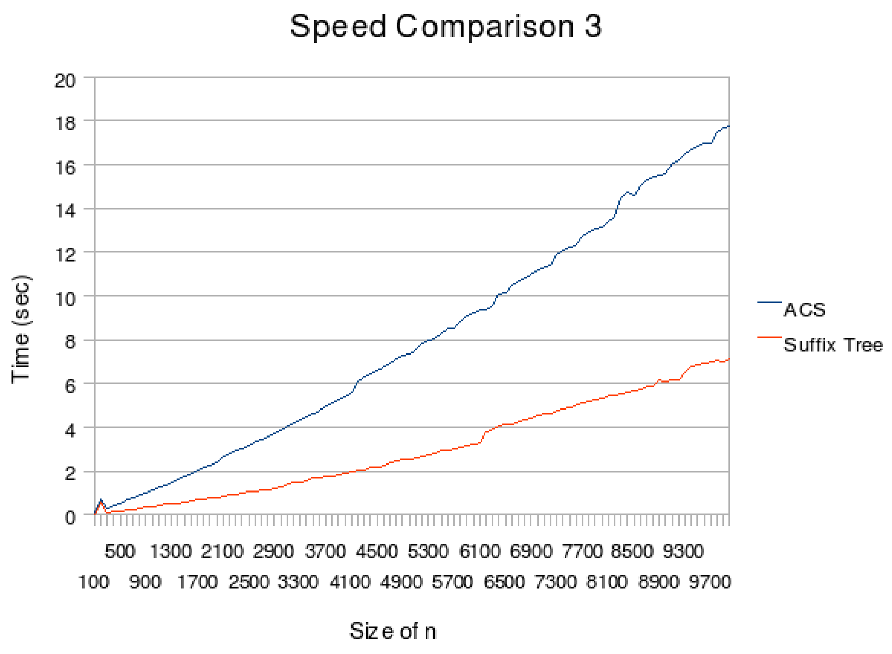

Figure 5.7: Comparison three

Chapter 6

TM Search Results Ranking

In this chapter we look at our ranking technique and compare our results to the results of a user study.

6.1 Ranking

Ranking is performed by looking at the length of all partial matches within the search hit. To calculate the ranking of a particular search hit we use a formula similar to what was introduced in Section 3.3. In our equation instead of calculating μ we will calculate the rank of the search hit as a value bound in the interval [0,1].

Our equation is modified slightly from that in Equation 3.2. Instead of the correlation factor $c_{i,l}$, we use the ratio of the lengths[1] of the substring S_i to the string s_m. The equation (Equation 6.1) looks at all substrings S_i within the list of substrings found to be common between s_m and s_n ($Save_q$).

$$Rank(s_m, s_n) := (1 - \prod_{s_i \in Save_q} (1 - \frac{|s_i|}{|s_m|})) \tag{6.1}$$

We can use Equation 6.1 to calculate the rank of the following two strings. Assume $m = \{A\,B\,C\,D\,E\,F\}$ and $n = \{A\,B\,C\,X\,Y\,C\,D\,E\,F\}$. There are two common substrings between s_m and s_n of lengths 3 and 4. The rank of the search hit s_n is

[1]The length of the string is calculated as the number of words in the string.

calculated as $1 - ((1 - 3/6) \cdot (1 - 4/6)) = 0.83$. String s_n is an 83% match of string s_m.

Some of the nice features of this equation are that the size of s_n is irrelevant to the rank. All substrings are accounted for in the rank as long as the exact match is not found. If string s_m shows up in its entirety in s_n then the equation returns a rank of 1 regardless of other possible substring matches.

Based on results from our user study we did modify our ranking algorithm in the following ways. Common substrings of length 1 were not included in the rank. Stop words that either begin or end a common substring are not considered as part of the length of the substring. Theses modifications tended to be more consistent with the results we obtained from our survey. The list of stop words used can be found in Appendix A.

6.2 Filtering

The filtering phase is where hits with duplicate terms are removed. This process is to ensure the top hits will all help to translate the document. The filtering process starts with the highest ranked result from the previous ranking phase. If the search hit has a rank of 1 (exact match) it is skipped and not included in the filtering stage. Starting with the highest non 1 ranked hit, the terms in the query are marked. Each query term is marked if that term shows up in a search hit. The process continues to the next lower ranked search hit. Query terms that are marked are removed from the search hit's ranking calculation, thus changing the ranking of the search hit. This process continues through all of the ranked hits. Some hits may get filtered more than once.

The process of filtering removes search hits that match the same portion of the query. After filtering, the translator is presented with distinct search hits that offer the greatest variety in translations for the query. For example take the query "I like to run" and lets say the phrase "I like to" is translated in 50 different documents and the term "run" only shows up once. The filtering process will eliminate the duplicate translations of "I like to" and allows the translation of "run" to come to the top of the search results.

6.3 Ranking Survey Results

To validate our ranking algorithm we chose a large TM consisting of over 100,000 segments and loaded the TM into the Lucene search engine using the standard analyzer. The standard Lucene analyzer searches for text using first a boolean algorithm and then ranks the hits using the VSM. The VSM ranking provided the baseline for our ranking test.

We then chose three segments from the TM to use as our test searches. To narrow down the list of possible segments we searched the TM using every segment withing the TM and ranked them using our algorithm. Using segments from the TM that we are searching did return one exact match which we threw out. The reason for using segments from the TM which we were searching is that we have better chances of getting good hits, because segments within a TM tend to be similar to each other. To narrow down the list of segments to choose for the survey we set certain criteria on the search results through trial and error. The criteria that we settled on for narrowing the list is that the 5th search hit ranked above a 40 (on a scale from 0 to 100) and the 17th search hit ranked a zero, this criteria ensured that we have a certain percentage of good hits and a certain percentage of bad hits within the survey.

After the segments were chosen the search hits for each segment were put in a random order and presented to the translator. Each translator was presented the search hits in the same random order. The translator was then asked to choose the top five search hits that would be most useful with aiding in the translation of the provided segment. The translators were only presented with the English version of the searches and the target language was not specified. The survey was timed so that later we could identify those who didn't take the time to do the survey right and their responses could be thrown out.

We learned some interesting things about how far a translator will look to find a good search hit. Good hits that were further down in the survey tended to get overlooked as well as longer search hits. Unfortunately we also had several responses that appeared random and added noise to the results. Each survey was timed and we found that most results done in less then 60 seconds were inconsistent and random in nature. When considering the results we first threw out all responses that were done in less than one minute. Even after filtering the results we still have a bit of noise to contend with, for example in every case, except hit 14 of survey 3, all hits were chosen by at least one translator.

For each survey we present a precision table (Table 6.1, Table 6.2 and Table 6.3). The precision table shows accuracy in the ranking of our algorithm and Lucene's VSM algorithm when compared to the results of the survey. Although not a perfect metric for the kind of results presented here, the precision table does help to show how our ranking algorithm and Lucene's VSM compare to the survey results.

Also included is a graph of the results for each survey (Figure 6.1, Figure 6.2 and Figure 6.3). The X axis represents the search hit as ranked by Lucene. For the survey results the Y axis is the percent of the translators that chose that particular search hit as one of their five responses and for our ranking results the Y axis represents the rank of the search hit.

In the first survey (See Appendix B, Figure 6.1 and Table 6.1) the clear winners are hits 1, 2 and 19. Hit 19 is a good example of how our ranking can improve search hits returned to translators, this hit was ranked very low by Lucene but is ranked high by both our method and the translators. The precision graph for the second survey

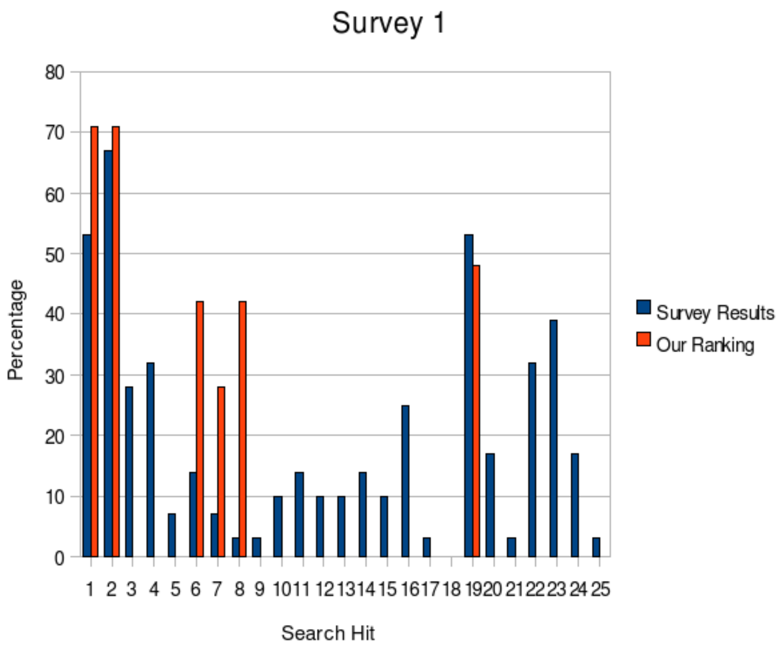

Figure 6.1: Survey one

Rank	ACS	Lucene
1	78%	78%
2	100%	100%
3	100%	85%
4	88%	85%
5	78%	76%

Table 6.1: Precision table for Survey 1

(Table 6.1) shows the loss of precision by Lucene because of hit 19. Our ranking, however placed hit 19 in third place.

The second survey (See Appendix B, Figure 6.2 and Table 6.2) is a good example of how the VSM in Lucene does not work well for ranking search hits for translators. In the second survey hits 3, 9, 10 and 24 are the clear favorites, three of which were ranked highest by our ranking method. Also in the second survey the precision (see Table 6.2) of our ranking is much better than Lucene.

The third survey (See Appendix B, Figure 6.3 and Table 6.3) results shows a case where both our algorithm and Lucene are in very close agreement with what the

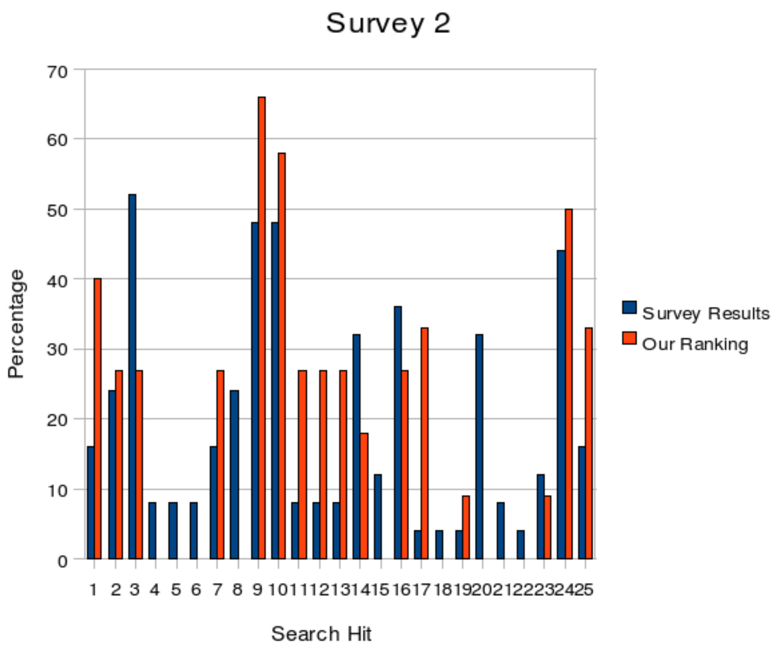

Figure 6.2: Survey two

Rank	ACS	Lucene
1	92%	30%
2	95%	40%
3	94%	62%
4	81%	52%
5	70%	47%

Table 6.2: Precision table for Survey 2

translators picked. Even though our ranking is not always better than Lucene's we are not any worse.

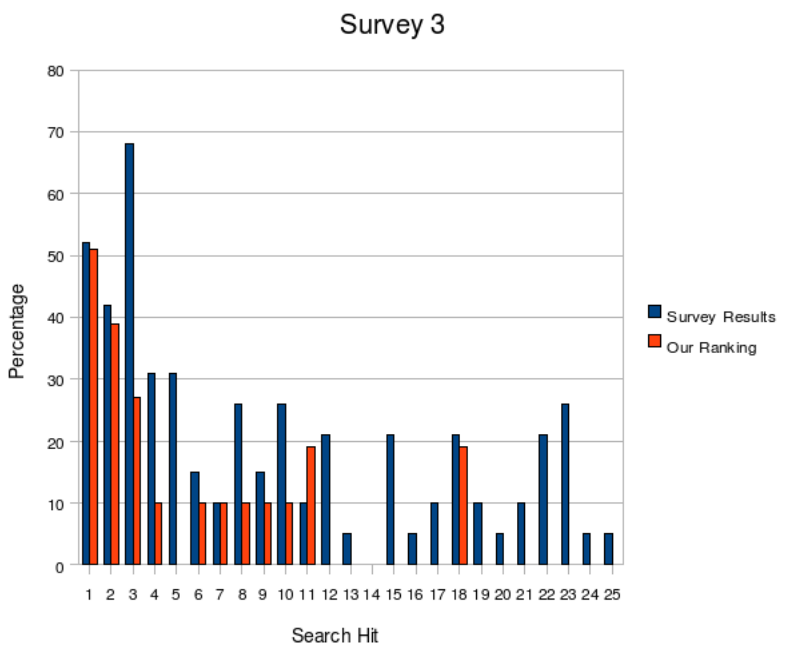

Figure 6.3: Survey three

Rank	ACS	Lucene
1	76%	76%
2	78%	78%
3	100%	100%
4	94%	100%
5	86%	100%

Table 6.3: Precision table for Survey 3

Chapter 7

Conclusions

In this thesis we have presented a simple and effective means for finding common substrings and ranking search hits from Translation Memory. Although our new algorithm for finding common substrings (ACS) is not faster than the suffix tree, it does have some distinct advantages. The ACS algorithm has a quicker initialization time than that of the suffix tree and it is much simpler to implement. In certain circumstances the ACS algorithm will run as fast as the suffix tree. The ACS algorithm is well suited for comparing strings that have few common substrings.

We have shown that our ranking technique is able to identify good hits that were missed by the commonly used VSM ranking. Our ability to identify good search hits and rank them appropriately will greatly aid a translator's effectiveness as translators will not spend the time to dig through search hits in order to find a good translation. We also observed that translators could greatly benefit from the TM tool highlighting matching terms in the search hit. Highlighting search terms will help when good hits are found within long segments.

Appendix A

Stop Word List

The following is a list of stop words used to modify the ranking of segments. This list was obtained from a speculated list of stop words that Google uses.

i, a, about, an, are, and, as, at, be, by, com, de, en, for, from, how, in, is, it, la, of, on, or, that, the, this, to, was, what, when, where, who, will, with, und, the, www

Appendix B

Surveys

	Regenerate–Update modified part and assembly dimensions.
Hit Num	Search Hit
25	For Part–The part can make external references to other parts that are in the same subassembly object anywhere in the assembly as the part being modified, to subassemblies and their subcomponents that are in the same subassembly as the part being modified, and to the skeleton model of the subassembly to which the part being modified belongs.
10	Part–Erase dimensions for a part at the selected assembly location.
23	In an assembly drawing, you can show and erase assembly and part dimensions.
4	To Edit Dimensions of a Part in an Assembly
22	After you have modified dimensions of the weld using Mod Dim in the MOD WELD menu or dimensions of the assembly, regenerate the model with the Regenerate command in the WELDING menu.
6	When an assembly feature has intersecting components, they are named (part and assembly are modified):
9	d#–Dimensions in Part or Assembly mode.
2	RegenerateUpdates modified part and assembly dimensions (also reassembles after exploding).
16	Mod Part-Allows modification of the assembly component dimensions.
21	All of the pipelines in the active assembly filter out existing pipe solids and allow the part name and Start Part to be modified.
18	Ad#–Driven dimensions in Part, Assembly, or Drawing mode.
24	For example, if the specified model is a part, the system shows only dimensions of the part in the assembly.
3	To Modify Dimensions of a Part in an Assembly
11	Lock Modified Dimensions-Lock or unlock modified dimensions.
5	The assembly instance in the assembly family table is modified to include the part instance.
20	The dimensions, attributes, scheme, and so on of read-only features can not be modified and are not regenerated when the part is regenerated.
13	The on-screen display always accurately presents the default dimensional tolerances for the object being modified, whether that object is a part, an assembly, or a part being modified in the context of an assembly.
12	The system creates the feature in the object (part, skeleton, or assembly) currently being modified.
1	Edit > RegenerateUpdates modified part and assembly dimensions.
14	The dimensions for locations that can be modified are:
15	The welding feature dimensions are modified.
7	The dimensions can be section, part, or assembly dimensions.
17	rd#–Reference dimensions in part or top-level assembly.
8	Showing Dimensions in Part and Assembly Modes
19	Regenerate–Opens the PRT TO REGEN menu on which you can update modified objects and assembly dimensions.

Table B.1: Survey 1. Results are shown in the order as they were presented in the survey.

	Mechanica deactivates the Display Options tab and the Display Locations tabs.
9	The Display Options tab displays some or all of these items depending on the design study and the selections you make on the Quantity tab and the Display Options tab:
16	Use the Display Options tab on the Result Window Definition dialog box to determine the appearance of your results window display.
19	Using the Grids and Settings tabs, specify the display.
11	Then select Contour from the Common Settings section of the Display Options tab.
7	When you select Animate on the Display Options tab, these options become available:
13	Select the Display Options tab.
6	The tabs display the following fields:
20	Display Location Tab
14	Display Options
8	Display Location tab - Select specific locations on your model to display in the results window.
3	Selecting one of these display types determines the options available on the Display Options tab:
24	The display options available on the Display Options tab vary depending on the display type you choose from the Display Type option menu.
4	Use the Options and Preview tabs to define what and when the bend notes display.
17	When you select Fringe from the Display Type option menu, these items become available on the Display Options tab:
21	Click the Drawing Display tab.
5	Use the Options and Preview tabs to define when and what symbols will display.
18	Display Arrows - Use the options on this tab to select measures and input loads.
1	When you select P-Level from the Quantity menu, the Display Options and Display Locations tabs are unavailable.
25	When you select Vectors from the Display Type option menu, the options available on the Display Options tab become specific to vector result window displays.
12	Select Deformed from the Common Settings section of the Display Options tab.
2	Display Options Tab
10	When you select Graph from the Display Type option menu, the Display Options tab and the Display Location tab become unavailable.
23	Click the Display tab.
22	Graph Display Tab
15	Use the Graph Location area on the Quantity tab to select specific locations for your graph result window display.

Table B.2: Survey 2. Results are shown in the order as they were presented in the survey.

	The reference is displayed in the collector on the dialog.
19	Depth1 Reference Collector
12	Direction reference summary collector
22	The second direction reference collector.
23	Direction reference collector
15	Placement Reference Collector
25	Depth box and Reference collector
1	The view name is displayed in the reference collector on the dialog.
5	Reference collector
14	Depth2 Reference Collector
3	The reference name is displayed on the dialog box.
16	Direction Reference collector
8	This command is available if a transition collector is displayed.
9	The selected edge is displayed in the Edge collector.
4	The selected reference (surface or datum plane) is displayed in the From collector.
2	Reference status is displayed in the collector and in the Troubleshooter dialog box.
11	The name of the selected subassembly is displayed in the Subassembly Reference area of the dialog box.
6	The reference you select appears in the References collector in the DATUM PLANE dialog box.
7	Depth Reference CollectorActivates the Depth Reference collector located on the dialog bar and on the Shape slide-up panel enabling you to select a depth reference.
18	A missing reference is marked in the collector.
13	Collector Reference Commands
17	Primary reference collector
21	Reference Plane collector
24	The first direction reference collector.
10	The selected entity is displayed in the Surface collector.
20	Angle box/Reference collector

Table B.3: Survey 3. Results are shown in the order as they were presented in the survey.

Bibliography

[1] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

> ANNOTATION: In chapter two of this book it covers different kinds of information retrieval modeling. Covered in chapter two is the fuzzy set model. The fuzzy set model uses a formula to compute the similarity between two documents by looking at the similarity of each term in the documents. The fuzzy set formula was adapted and used in my project to rank search hits from a TM.

[2] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20:762–772, 1977.

> ANNOTATION: This paper introduces the Boyer Moore string searching algorithm. The Boyer Moore algorithm is an extremely fast method for finding strings, but it doest not perform well well trying to find common substrings. When searching for common substrings a common brute force method is to find a single word and then grow the search to see how big the substring is. The Boyer Moore algorithm could be used to find the first word but would not work well as the search string grows to find a match.

[3] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. pages 613–622, Hong Kong, Hong Kong, 2001. ACM.

> ANNOTATION: This document shows several different ranking techniques used on the web. All of the ranking schemes in this work look at finding similarity between the query and the search hit. This is unlike the ranking performed in this work where ranking is based on sub string matches.

[4] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and*. Cambridge University, 1997.

ANNOTATION: This book contains five chapters on the construction and use of suffix trees. The book covers the longest commons substring problem as well as covers constructing a suffix tree in linear time using Ukkonen's algorithm.

[5] G. Hodász, T. Grobler, and B. Kis. Translation memory as a robust example-based translation system. *Proceedings of the Ninth EAMT workshop*, pages 82–89, 2004.

ANNOTATION: This paper introduces a technique where by Translation Memory (TM) segments are broken up into sub-sentence unites according to rules based on syntax. The sub-sentences are then used to build up translations of larger sentences when exact matches are not found. The proposed solution is very language specific, but is being aided by advancements in grammar acquisition methods that will speed up the grammar preparation for additional languages.

[6] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20:350–353, 1977.

ANNOTATION: Hunt Szymanski algorithm

[7] Philippe Langlais Michel Simard. *Sub-sentential Exploitation of Translation Memories.* EAMT Geneve, 2001.

ANNOTATION: This paper talks about using substring matches when searching TM. The paper breaks phrases into chunks based on a language model. Each chunk is then searched for in the TM. Substring searching finds better matches then using existing edit distance metric

[8] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18:613–620, 1975.

ANNOTATION: This is the first paper I could find that talks about the VSM. The vector space model or VSM is a ranking technique that determines how similar one phrase is to another. The technique is very fast to compute and offers a good indexing and ranking technique for determining similarity.

[9] Graham A. Stephen. *String Searching Algorithms.* World Scientific, 1994.

ANNOTATION: This book describes several string searching techniques. The algorithm of most interest to this work is the solutions to the longest common substring problem. Chapter three of this book covers string distance and common sequences.

[10] Eiichiro Sumita. Example-based machine translation using dp-matching between word sequences. pages 1–8, Toulouse, France, 2001. Association for Computational Linguistics.

ANNOTATION: This paper introduces an approach to machine translation where by results are taken from a Translation Memory (TM) and combined with results from a bilingual dictionary and thesaurus to produce a translation of the content. This paper is a good example of Example Based Machine Translation and exemplifies the use of TM in current research.

[11] Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

ANNOTATION: This paper presents Ukkonen's suffix tree algorithm. Ukkonen's algorithm provides a way to build a suffix tree *on-line*. The ability to build the tree *on-line* means it can build the tree using the tokens in the string in the same order as they appear. Ukkonen's algorithm can build a suffix tree in linear time.

[12] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21:168–173, 1974.

ANNOTATION: This is the paper that introduces the edit distance metric. The edit distance metric is the number of edit operations to transform one string into another string. The number of edits are used to determine the similarity between the two strings.