Fast Approximate String Matching with Suffix Arrays and A* Parsing

Philipp Koehn
University of Edinburgh
10 Crichton Street
Edinburgh, EH8 9AB
Scotland, United Kingdom
pkoehn@inf.ed.ac.uk

Jean Senellart
Systran
La Grande Arche
1, Parvis de la Défense
92044 Paris, France
senellart@systran.fr

Abstract

We present a novel exact solution to the approximate string matching problem in the context of translation memories, where a text segment has to be matched against a large corpus, while allowing for errors. We use suffix arrays to detect exact n-gram matches, A* search heuristics to discard matches and A* parsing to validate candidate segments. The method outperforms the canonical baseline by a factor of 100, with average lookup times of 4.3–247ms for a segment in a realistic scenario.

1 Introduction

Approximate string matching is a pervasive problem in natural language processing applications, such as spelling correction, information retrieval, evaluation, and translation memory systems. Often we cannot find an *exact* match for a query string in a large corpus, but we are still interested in an *approximate* match that is similar according to some metric, typically string edit distance.

The problem is of great concern in genetic sequence retrieval. In fact, as we discuss below, most of the work on approximate string matching addresses this task.

We present a new method that was developed in the context of translation memory systems, which are used by human translators. When translating an input sentence (or segment), the human translator may be interested in the translation of a similar segment, which is stored in a translation memory. The term translation memory is roughly equivalent to a parallel corpus: a collection of segments and their translations. The approximate string matching problem for translation memories is to find the source language segment that is most similar to the input.

Note that translation memories are rarely used in recent machine translation research, since that work is driven by open domain translation tasks such as news translation. In the practical commercial world of human translation, however, translation tasks often involve the translation of material that is very similar to prior translations. Consider, for instance, the translation of a manual for an updated product.

This paper defines the problem of approximate string matching (Section 2) and reviews related work (Section 3). Our method uses a suffix array to find n-gram matches (Section 4), principles of A* search to filter the matches, and an A* parsing method to identify the most similar segment match (Section 5.3). It retrieves the most similar fuzzy match in an average time of 4.3–247 milliseconds (Section 6).

2 Approximate String Matching

Let us first formally define approximate string matching. We follow the definition by Navarro (2001):

The problem involves:

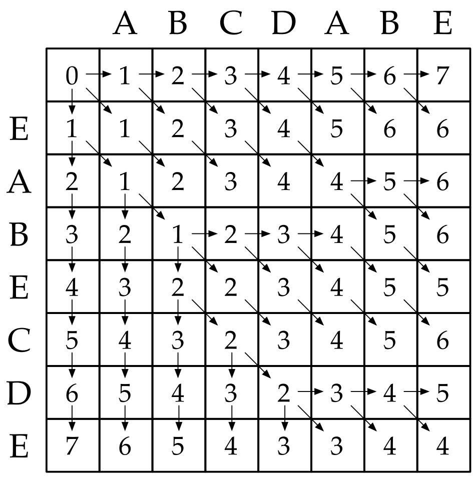
- a finite alphabet Σ of size $|\Sigma| = \sigma$
- a corpus $T \in \Sigma^*$ of length $|T| = n$
- a pattern $P \in \Sigma^*$ of length $|P| = m$
- a distance function $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$
- a maximum error allowed $k \in \mathbb{R}$

The problem is typically defined as: given T,P,$d()$,k, retrieve all text positions j so that there exists an interval $T_{i..j}$ with $d(P, T_{i..j}) \leq k$.

In this paper, we modify the definition of problem, as follows:

Segments: The corpus T tiles into a sequence of segments $\{S_1, ..., S_s\}$ so that

- start$(S_1) = 1$



		A	B	C	D	A	B	E
	0	1	2	3	4	5	6	7
E	1	1	2	3	4	5	6	6
A	2	1	2	3	4	4	5	6
B	3	2	1	2	3	4	5	6
E	4	3	2	2	3	4	5	5
C	5	4	3	2	3	4	5	6
D	6	5	4	3	2	3	4	5
E	7	6	5	4	3	3	4	4

Figure 1: Canonical dynamic programming solution to the approximate string matching problem under a unit cost string edit distance. Each alignment point between the two strings ABCDABE and EABECDE is filled with the minimal cost and a backpointer to its prior alignment point.

- $\text{end}(S_s) = |T|$, and
- $\forall_{1<z\leq s}\text{end}(S_{z-1}) + 1 = \text{start}(S_z)$,

i.e., the segments start at the beginning of the text, end at the end of the text, and follow each other.

Minimum matching cost: Given the sequence of all segments $\{S_1, ..., S_s\}$ as defined above, the minimum matching cost is $c = \min\{d(P, S_z)|S_z \in \{S_1, ..., S_s\}, d(P, S_z) \leq k\}$.

Task: retrieve all segments S_z with distance $d(P, S_z) = c$.

3 Related Work

The canonical method for the approximate string matching problem under the string edit distance metric has been discovered many times. It uses dynamic programming, as illustrated in Figure 1. Given the matrix of possible alignment points $\{(i, j)|0 \leq i \leq |a|, 0 \leq j \leq |b|\}$ between the two strings a and b, we compute for each point the optimal path, with only depends on the directly preceding points, as defined by the deletion, insertion, substitution, and match operations.

The method can be adapted straightforwardly to the problem where the pattern may match at any starting point and end point in the corpus (the general approximate string matching problem), or our problem, where the pattern may be matched against any of the segments of the corpus. The complexity of the algorithm is $O(nm)$ — linear both with respect to the corpus size and the pattern size. Our method has sub-linear complexity with respect to corpus size.

Most of the work in this area deals with genetic sequence retrieval. Biologists search for genes or other DNA sequences consisting of the letters A, C, G, and T in the genetic code of a living being. Our problem differs from this application in various points:

- the lexicon is much larger: 10,000s or more distinct words found in a text vs. four bases
- the allowable error is larger: in translation memories, we may accept up 30% error
- sequences are shorter, typically 10-50 words
- there are natural starting and end points, i.e., segment boundaries
- we search for the best matches, not all matches

The first two differences make our problem harder, while the last three make it easier, and we exploit them in our method.

The main building blocks of approximate string matching methods are corpus indexing, filtering, pattern processing, and improvements to the dynamic programming techniques (Navarro, 2001).

Common corpus indexing methods are suffix trees and suffix arrays. Suffix trees compile the corpus into a trie, which allows the quick retrieval of any substring in the corpus. However, worst case quadratic space requirements are prohibitive for our application. We use suffix arrays (Manber and Myers, 1990), which are described in detail in Section 4.

The idea of filtering is to discard most of the search space and to focus on the promising areas. The method is divided into a filtering and validation stage. The focus is typically on filtering, while standard techniques are used for validating potential matches. Filters based on suffix arrays are commonly used, see for instance work by Kärkkäinen and Na (2007). Our method is partly a filtering method. It gets most of its gains from the filtering stage, although we also optimize validation based on the information gained in the filtering stage.

There are various ways to process the pattern, for instance splitting it into sub-patterns for which exact matching is performed, or compiling it into a finite state machine. Our method utilizes exact matches of sub-patterns.

The dynamic programming techniques can be improved in many ways. To give an example, in the canonical algorithm, we do not need to compute the entire matrix but can focus on the alignment points with the lowest cost.

The only description of a method addressing the approximate string matching problem in translation memories, that we are aware of, is work by Mandreoli et al. (2002), which uses simple filtering methods using a database. They report speeds of *”8 seconds to compare 419 query sentences against 1497 reference sentences”*, and *”1.5 seconds per query sentence”* with a larger corpus, which is a few orders of magnitude slower than our method.

Suffix arrays have been applied to a related problem in machine translation, namely looking up phrases in a word-aligned parallel corpus to compute phrase translation probabilities. Work by Callison-Burch et al. (2005); Zhang and Vogel (2005); McNamee and Mayfield (2006) was extended to so-called hierarchical phrases, essentially phrases with gaps, by Lopez (2007).

4 Suffix Arrays

Our method uses **n-gram matches** between the input segment (the pattern) and the translation memory (the corpus) to identify potential **candidate corpus segments**.

We store the corpus in a suffix array to enable quick lookup. The data structure uses an index of starting positions of all suffixes in the corpus, which is sorted alphabetically (see Figure 2). This allows us to use binary search to find a particular suffix in the corpus.

We sort the index using quick sort, which is O($n \log n$). Our implementation takes a few seconds even for corpora with tens of millions of words. We identify words with integer word ids, so we also sort the index based on these word ids, not actually alphabetically. The suffix array takes up O(n) space, doubling the space requirements for storing the corpus, which is not a problem with modern computers and customary corpora. Searching the array for an n-gram of size g takes O($g \log n$) time.

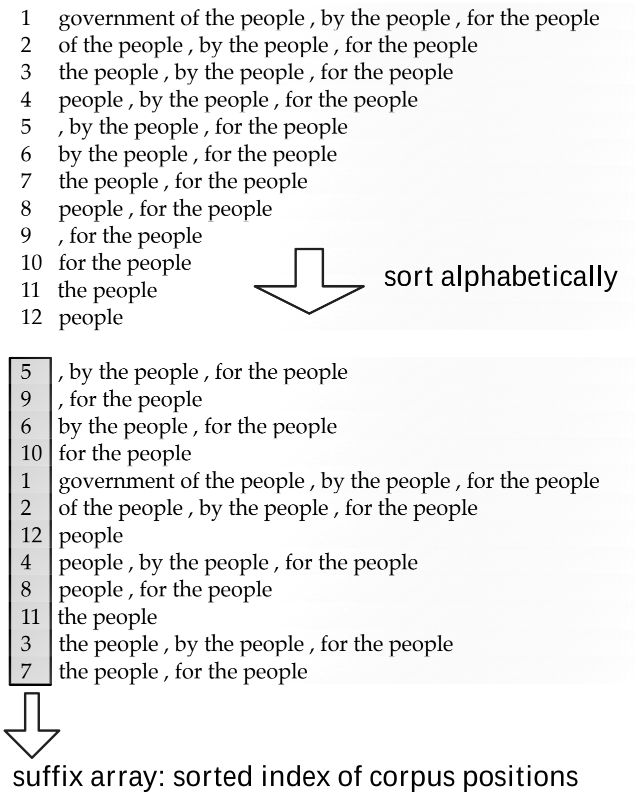

1 government of the people , by the people , for the people
2 of the people , by the people , for the people
3 the people , by the people , for the people
4 people , by the people , for the people
5 , by the people , for the people
6 by the people , for the people
7 the people , for the people
8 people , for the people
9 , for the people
10 for the people
11 the people
12 people
sort alphabetically
5 , by the people , for the people
9 , for the people
6 by the people , for the people
10 for the people
1 government of the people , by the people , for the people
2 of the people , by the people , for the people
12 people
4 people , by the people , for the people
8 people , for the people
11 the people
3 the people , by the people , for the people
7 the people , for the people
suffix array: sorted index of corpus positions

Figure 2: **Suffix array:** When sorting all suffixes of a corpus (here, a 12 word phrase) and keeping track of their corpus positions, a suffix array is created. The suffix array can be used to find a particular suffix by binary search.

Finding all occurrences of an n-gram in the corpus builds on the binary search method to find an existing suffix in the corpus. Matching an n-gram requires that the g words of the n-gram match the first g words of the suffix. First, we attempt to find any match in the corpus (and terminate if we fail to do so). Then we perform two binary searches to find the first and last occurrence of the n-gram. This gives a range of corpus positions, where the n-gram occurs. This range is an interval in the index.

A pattern of m words contains O(m^2) n-grams. Since the binary search for a n-gram of size g takes O($m \log n$) time (and $g \leq m$), looking up all n-grams takes O($m^3 \log n$) in the worst case. However, since when looking up, say, the *trigram* starting at the beginning of the pattern, we can re-use the results of the previous search for the *bigram* starting at the beginning of the pattern. The range of matches for the trigram is a subset of the matches for the bigram. Since, we cannot be sure that the ranges get smaller with longer n-grams, this does not reduce the worst case cost of O($m^3 \log n$), but it does dramatically reduce the cost in the average case to $O(m \log n)$.

5 A* Search

A* search uses a heuristic to estimate the cost of a partial solution to a search problem. A* search requires that the heuristic is *admissible*, i.e. it may over-estimate the cost of a partial solution, but never under-estimate it.

If we find a **ceiling cost** — for instance, by finding a (typically sub-optimal) complete solution to the problem, or by other means — A* search can safely discard all partial solutions that have a worse heuristic cost than the ceiling cost.

The strategy to employ A* search is two-fold: we would like to explore the most likely part of the search space (as indicated by the heuristic cost of partial solutions), and we would like to drive down the ceiling cost to safely discard much of the space.

In our method, we use A* search in two stages: first as a filtering methods to discard n-gram matches and candidate segments, and then as a validation method to compare a pattern against a corpus segment. During both stages we are keenly interested in reducing the ceiling cost.

We initialize the ceiling cost to the maximum cost allowed when matching the pattern against the corpus. In a translation memory application, we are only interest in segments that match at least 70% (or more), i.e. the maximum allowable string edit distance k is $\lceil 0.3m \rceil$.

See Figure 3 for the pseudo code of the algorithm. The rest of this section describes it in detail.

5.1 Match Filtering

Recall that we use the suffix array to generate all n-gram matches between the pattern and the corpus. When we identify such an n-gram match, we record its

- start and end position in the pattern
- start and end position in the corpus segment
- the corpus segment id

We also compute two cost estimates: its **minimum** and **maximum cost**. The minimum cost is the lowest possible cost for matching the pattern against the corpus segment in which the match was found, when everything else goes perfectly fine. The maximum cost assumes that this n-gram is the only match between the pattern and the corpus.

The minimum cost composed of:

- the *difference* between the starting position of the n-gram in the pattern and the starting position of the n-gram in the corpus segment, or 1 if they are the same except if both are at the segment start.
- the *difference* between the number of words after the n-gram in the pattern and the number of words after the n-gram in the corpus segment, or 1 if they are the same except if both are at the segment end.

The maximum cost is composed of:

- the *maximum* of the starting position of the n-gram in the pattern and the starting position of the n-gram in the corpus segment.
- the *maximum* of the number of words after the n-gram in the pattern and the number of words after the n-gram in the corpus segment

If a newly found match has a lower maximum cost than the ceiling cost, we can update the ceiling cost to this value.

If the match has a higher minimum cost than the ceiling cost, we can safely discard the match. Surviving matches are stored in a hash indexed by the corpus segment id.

Note that if we find a corpus segment that is identical to the pattern, it has a maximum cost of 0, invalidating all matches that are not also such full matches.

Also note that if we find, say, a trigram match, we will also find two underlying bigram and three underlying unigram matches. However, we remove such sub-matches from our set of matches, since they cannot be used with a lower cost than the trigram match.

5.2 Length Filtering

The collection of a set of n-gram matches yields a set of candidate corpus segments. We perform an additional filtering step before full validation, i.e., before computing the string edit distance between each corpus segment and the pattern.

```
MAIN:
Input: pattern $p = p_1..p_n$
Output: best matching segments $\mathbf{S}$
ceiling-cost = $\lceil 0.3 \times$p.$length\rceil$
$\mathbf{M}$ = find-matches($p$)
$\mathbf{S}$ = find-segments($p$,$\mathbf{M}$)

FIND-MATCHES:
Input: pattern $p = p_1..p_n$
Output: matches $\mathbf{M}$
for start = 1 .. $p$.length do
    for end = start .. p.length do
        remain = $p$.length $-$ end
        $\mathbf{M}_{\text{start,end}}$ = find-in-suffix-array($p_{\text{start}}..p_{\text{end}}$)
        break if $\mathbf{M}_{\text{start,end}}$ == $\emptyset$
        for all $m \in \mathbf{M}$ do
            $m$.leftmin = $|m$.start $-$ start$|$
            $m$.leftmin = 1 if $m$.leftmin == 0 & start $> 0$
            $m$.rightmin = $|m$.remain $-$ remain$|$
            $m$.rightmin = 1 if $m$.rightmin == 0 & remain $> 0$
            min-cost = $m$.leftmin+$m$.rightmin
            break if min-cost $>$ ceiling-cost
            $m$.leftmax = max($m$.start, start)
            $m$.rightmax = max($m$.remain, remain)
            $m$.pstart = start; $m$.pend = end
            $\mathbf{M} = \mathbf{M} \cup \{m\}$
        end for
    end for
end for

FIND-IN-SUFFIX-ARRAY:
Input: string
Output: matches $\mathbf{N}$
first-match = find first occ. of string in array
last-match = find last occ. of string in array
for index $i$ = first-match .. last-match do
    $m$ = new match()
    $m$.start = $i$.segment-start
    $m$.end = $i$.segment-end
    $m$.length = $i$.segment.length
    $m$.remain = $m$.length $-$ $m$.end
    $m$.segment-id = $i$.segment.id
    $\mathbf{N} = \mathbf{N} \cup \{m\}$
end for

FIND-SEGMENTS:
Input: pattern $p$, matches $\mathbf{M}$
Output: best matching segments $\mathbf{S}$
for all $s : \exists m \in \mathbf{M} : m$.segment-id $= s$.id do
    $a$ = new agenda-item()
    $a$.M = $\{m \in \mathbf{M} : m$.segment-id $= s$.id$\}$
    $a$.sumlength = $\sum_{m \in a.\text{M}} m$.length
    $a$.priority = $-$ $a$.sumlength
    $a$.s = $s$
    $\mathbf{A} = \mathbf{A} \cup \{a\}$
end for
while $a$ = pop($\mathbf{A}$) do
    break if $a$.s.length - $p$.length $>$ ceiling-cost
    break if max($a$.s.length, $p$.length) - $a$.sumlength $>$ ceiling-cost
    cost = parse-validate($a$.s, $a$.M)
    if cost $<$ ceiling-cost then
        ceiling-cost = cost
        $\mathbf{S} = \emptyset$
    end if
    $\mathbf{S} = \mathbf{S} \cup \{a$.s$\}$ if cost == ceiling-cost
end while

PARSE-VALIDATE:
Input: string, $\mathbf{M}$
Output: cost
for all $m_1 \in \mathbf{M}$, $m_2 \in \mathbf{M}$ do
    $\mathbf{A} \cup \{a\}$ if $a$ = combinable($m_1, m_2$)
end for
cost = min $\{m$.leftmax $+$ $m$.rightmax$|m \in \mathbf{M}\}$
while $a$ = pop($\mathbf{A}$) do
    break if $a$.mincost $>$ ceiling-cost
    $mm$ = new match()
    $mm$.leftmin = $a$.m1.leftmin
    $mm$.leftmax = $a$.m1.leftmax
    $mm$.rightmin = $a$.m2.rightmin
    $mm$.rightmax = $a$.m2.rightmax
    $mm$.start = $a$.m1.start; $mm$.end = $a$.m2.end
    $mm$.pstart = $a$.m1.pstart; $mm$.pend = $a$.m2.pend
    $mm$.internal = $a$.m1.internal + $a$.m2.internal + $a$.internal
    cost = min(cost, $mm$.leftmax + $mm$.rightmax + $mm$.internal)
    for all $m \in \mathbf{M}$ do
        $\mathbf{A} = \mathbf{A} \cup \{a\}$ if $a$ = combinable($mm, m$)
    end for
end while

COMBINABLE:
Input: matches $m_1$, $m_2$
Output: agenda item $a$
return null unless $m_1$.end $<$ $m_2$.start
return null unless $m_1$.pend $<$ $m_2$.pstart
$a$.m1 = $m_1$; $a$.m2 = $m_2$
delete = $m_2$.start$-$$m_1$.end-1
insert = $m_2$.pstart$-$$m_1$.pend-1
internal = max(insert,delete)
$a$.internal = internal
$a$.mincost = $m_1$.leftmin + $m_2$.rightmin + internal
$a$.priority = $a$.mincost
```

Figure 3: Pseudo code of the algorithm

Note that a large number of corpus segments is already excluded implicitly by our n-gram match requirements. If the difference in length between the corpus segment and the pattern is larger than the ceiling cost, then none of its n-gram matches has a minimum cost that is lower than the ceiling cost.

We pose another requirement for a corpus segment. The difference between the larger of segment length and pattern match minus sum of the length of all its n-gram matches has to be smaller than the ceiling cost. In other words, in addition to the cost due to length differences between the pattern and the corpus segment, the remainder has to made up by sufficient n-gram matches to not exceed the ceiling cost (optimistically, all of them are used).

5.3 Validation with A* Parsing

At this point, we have a set of candidate corpus segments. We could simply compute string edit distance to validate them and to find the segments with the lowest cost.

However, we also have all n-gram matches for each candidate corpus segment, so we can use a more efficient method for validation. This method draws from ideas in A* parsing. We combine the n-gram matches to build a binary tree that matches the pattern against the corpus segment.

See Figure 4 for an illustration of the process. We first arrange the matches in a chart covering the corpus segment, as it is commonly done in chart parsing. For each match, we record the minimum and maximum cost. The illustration also shows how these costs are composed of left (with respect to segment start) and right (with respect to segment end) matching costs.

Of all the matches in the example, the lowest maximum cost is 6, which hence constitutes the ceiling cost. Note that we may have external information that gives us a lower ceiling cost. Due to the ceiling cost of 6, we can safely discard the match of the first word in the candidate and the last word of the pattern (this match has a cost of 13).

We then proceed to pairwise combine matches into **multi-matches**. In the example, we first combine the match that covers candidate segment positions 2–3 with the match covering positions 5–6. The resulting multi-match inherits the left cost estimate from the first match and the right cost estimate

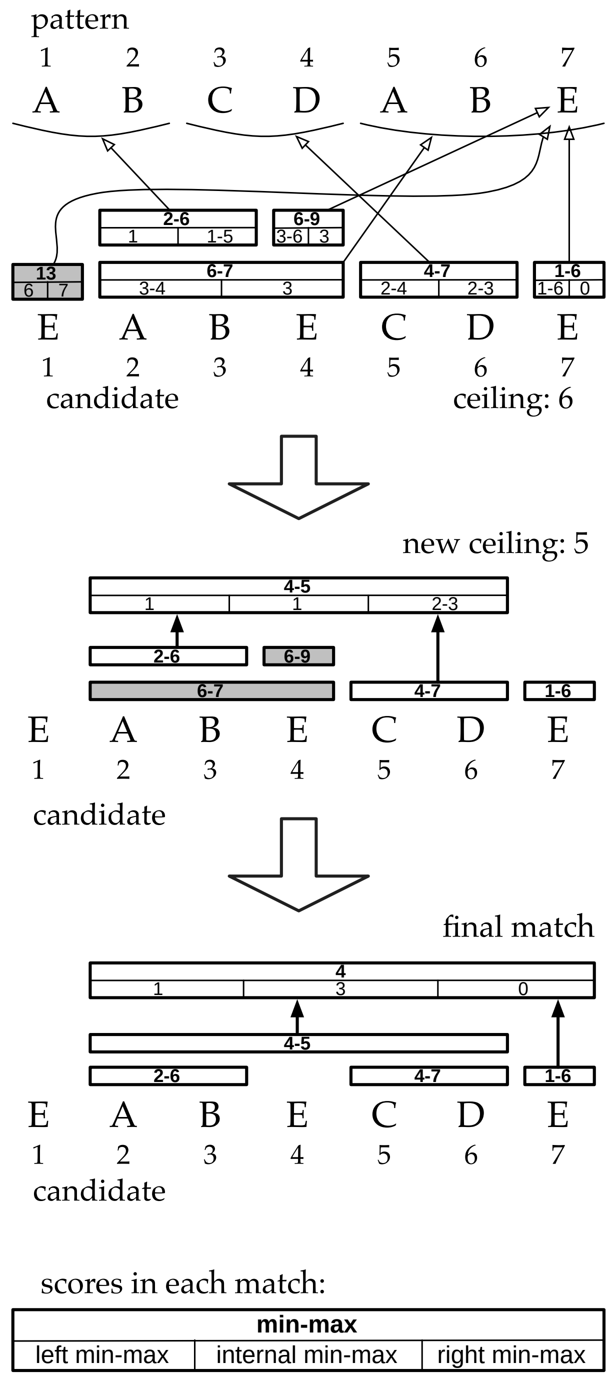


Figure 4: **A* parsing** to match a translation memory segment (candidate) to the input segment (pattern): For each n-gram match, the minimum and maximum cost is recorded. New matches are generated by pairwise combination. The ceiling cost is updated to the lowest maximum cost, which allows the removal of matches with higher minimum cost (indicated by grey boxes).

from the second match. It also incurs an internal cost of 1 due to the insertion of word E (position 4) in the candidate.

Overall, it has a maximum cost estimate of 5. Since the maximum cost estimate of the multi-match is lower than the ceiling cost, we update the ceiling cost. This also triggers the removal of two matches of the chart: the match covering candidate word positions 2–4, and the match covering position 4 have each a minimum cost of 6, and hence cannot possible be used in an optimal edit path.

In the example, the final step is the composition of the multi-match covering positions 2–7, which has a minimum and maximum cost of 4. No other combination of (multi-)matches has a lower minimum cost, so we found an optimal edit path and its string edit distance.

In the description of the example above, we glossed over the exact strategy of combining matches. There are many possible strategies for such a A* parsing method. We implemented a simple bottom up parser that incrementally adds valid multi-matches composed of an increasing number of matches. The parser terminates, when now new surviving matches are generated at a iteration (corresponding to certain number of matches). The canonical A* parsing strategy is best-first parsing with a priority queue.

The theoretical cost of the described A* parsing method is worse than the standard dynamic programming method, but in practice it is faster, except for degenerate cases. The fewer initial matches are found, the higher the speed-ups. For more on this, refer to the experimental section.

5.4 Refinement

The number of unigram matches between a pattern an the corpus is very large. Consider that in a text corpus most segments contain a period. This results in costly generation of matches, and a large number of candidate segments before filtering.

However, given a maximum error of, say, 30%, a candidate segment must have a matching n-gram larger than a unigram, except for very short patterns. Hence, we do not need to generate unigram matches for length filtering.

We gain significant speed-ups by postponing the generation of unigram matches after length filtering, when a much smaller number of segments are left to be considered. By hashing the unigrams in the pattern, we loop through all words in the candidate segment to detect unigram matches, and add them to the set of matches (unless they are subsumed by larger matches). While this implies a linear cost with respect to total length of all surviving candidate segments, it is still much cheaper than generating all unigram matches earlier on.

Acquis	Corpus	Test
segments	1,169,695	4,107
words	23,566,078	128,005
words/seg.	20	31

Product	Corpus	Test
segments	83,461	2,000
words	1,038,762	24,643
words/seg.	12	12

Table 1: Statistics of the corpus used in experiments

Note that when using lower maximum error rates, or when lower ceiling costs are detected, we may also postpone the generate of other small n-gram matches (bigrams, trigrams).

6 Experiments

We carried out experiments using two English data sets: the publicly available JRC-Acquis corpus[1] (Acquis) and a commercial product manual corpus (Product). We use the same test set as Koehn et al. (2009). See Table 1 for basic corpus statistics.

The Acquis corpus is a collection of laws and regulations that apply to all member countries of the European Union. It has more repetitive content than the parallel corpora that are more commonly used in machine translation research. Still, the commercial Product corpus is more representative of the type of data used in translation memory systems. It is much smaller (around a million words), with shorter segments (average 12 words per segments).

See Table 2 for a quantitive analysis of the ratio of approximate matches in the corpus. For the Acquis corpus the ratio of matches ranges from 45% to 60%, and for the Product corpus from 47% to 73%, when

[1] http://wt.jrc.it/lt/Acquis/ (Steinberger et al., 2006)

Max.	Acquis		Product	
error	matches	speed	matches	speed
40%	60%	462ms	73%	5.1ms
30%	54%	247ms	66%	4.3ms
20%	50%	163ms	58%	3.8ms
10%	45%	85ms	47%	3.2ms

Table 2: Number of matches and speed of lookup

Method	Acquis	Product
Baseline	33,776ms	371.2ms
with length filtering	2,965ms	77.1ms
Our method	**247ms**	**4.3ms**
w/o refinement	2,831ms	40.3ms
w/o length filtering	4,414ms	46.2ms
w/o A* parsing	6,079ms	113.0ms

Table 3: **Main results**: Performance of the method and contribution of its steps

varying the error threshold for acceptable segments from 10% to 40%.

Timing experiments were run on a machine with a 64-bit Intel Xeon E5430 2.66GHz CPU. Speed is measured in average time per segment lookup. The speed of our method varies for the different maximum error threshold numbers — in the given range roughly by a factor of 5 for Acquis and 2 for Product. With a lower maximum error more segments can be discarded early in the process.

The main results are shown in Table 3. When using a 30% threshold for accepting approximate matches, our method takes 3.0 milliseconds per input segment for the Acquis corpus, and 4.3ms for the Product corpus. This compares very favorably against the canonical baseline method, the well-known dynamic programming solution to the string edit distance problem (33,776ms and 371.2ms), even if we filter out segments that disqualify due to length differences (2,965ms and 77.1ms).

The table also gives an indication of the contribution of the steps of the method. The refinement of delaying the generation of unigram matches (see Section 5.4) is responsible for reducing the time needed by roughly a factor of ten — without it the time costs increases to 2,831ms and 40.3ms, respectively. If we leave out length filtering (see Section 5.2), time per segment increases to 4,414ms and 46.2ms. Finally, if we carry out the comparison of candidate segments against the input segment without A* parsing, time increases further to 6,079ms and 113.0ms.

Step	Acquis	Product
range finding	29.7ms	2.06ms
match generation	119.0ms	0.69ms
filtering	94.4ms	1.49ms
validation	3.7ms	0.01ms

Table 4: Time per segment for each step of the method

Another way to inspect the time requirements of the various steps of the method is to look at the time spent in each stage. See Table 4 for details. For the Product corpus, most time is spent on range finding and filtering, while for the Acquis corpus match generation dominates..

The validation step takes in both cases very little time. The A* parsing method is twice as fast as the canonical dynamic programming method (taking about 3.7ms vs 0.016ms per input segment in both corpora), but this is clearly not the bottleneck.

Note that there is also the stage of creating the suffix array, which takes about 40 seconds for the Acquis corpus and 2 seconds for the Product corpus. Since this is a one-time pre-processing cost, which can be incurred offline, we did not consider it in this analysis

It is worth pointing out that for the Product corpus, one degenerate case out of the 2000 input segments is responsible for a quarter of the average cost. It takes over 1000ms seconds to complete. The segment consist of a sequence of 120 periods, which are treated as tokens, which generates a very large number of n-gram matches, which requires a large amount of match filtering. Note that such outliers are a concern, but they can be easily detected and properly addressed.

Table 5 gives the average number of segments considered by the method. The number of segments with n-gram matches is 24,258.3 for the Acquis and 580.1 for the Product corpus. Segment filtering reduces these numbers to 77.8 and 6.8. Finally, 25.9 and 3.3 segments on average are consider best matches since they all have the optimal

Segments	Acquis	Product
with matches	24,258.3	580.1
after length filtering	77.8	6.8
best	25.9	3.3

Table 5: Number of corpus segments considered at different steps

edit cost. In our implementation we use string edit distance based on letter matches as the tie-breaker, which adds negligible computational cost.

7 Conclusion and Future Work

We present a novel method for approximate string matching and applied it to the translation memory search problem. The method outperforms the baseline canonical dynamic programming method by a factor of 100 in our experiments. It is an exact solution — it always find the optimal match.

The method involves n-gram lookup from a suffix array over the corpus, match and length filtering based on A* search principles and and A* parsing method for validation.

The largest gains in performance are due to the filtering techniques which leave very few segments for validation. The method may be improved by integrating the A* principle of maintaining and driving down the ceiling cost more tightly. For instance, if large n-gram matches with low maximum error are found, this may eliminate the need to look up smaller n-grams. Or, if we compute the actual scores for the most promising segments early on (for instance, when the longest matching n-gram is found), we may find a lower ceiling leading to reduction in n-gram lookup and finer match filtering.

While the method is very fast for our application (4.3–247ms) and further improvements are not an urgent matter, we would like to extend the method to similar approximate string problems, for instance using edit distances that allow moves or matching against word graphs. Such problems arise in word alignment, machine translation evaluation, and interactive machine translation.

Acknowledgement This work was partly supported by the EuroMatrixPlus project funded by the European Commission (7th Framework Programme).

References

Callison-Burch, C., Bannard, C., and Schroeder, J. (2005). Scaling phrase-based statistical machine translation to larger corpora and longer phrases. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 255–262, Ann Arbor, Michigan. Association for Computational Linguistics.

Kärkkäinen, J. and Na, J. C. (2007). Faster filters for approximate string matching. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–90.

Koehn, P., Birch, A., and Steinberger, R. (2009). 462 machine translation systems for europe. In *Proceedings of the Twelfth Machine Translation Summit (MT Summit XII)*. International Association for Machine Translation.

Lopez, A. (2007). Hierarchical phrase-based translation with suffix arrays. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 976–985.

Manber, U. and Myers, G. (1990). Suffix arrays: A new method for on-line string searches. In *First Annual ACM-SIAM Symposium on Dicrete Algorithms*, pages 319–327.

Mandreoli, F., Martoglia, R., and Tiberio, P. (2002). Searching similar (sub)sentences for example-based machine translation. In *Italian Symposium on Advanced Database Systems (SEBD)*.

McNamee, P. and Mayfield, J. (2006). Translation of multiword expressions using parallel suffix arrays. In *5th Conference of the Association for Machine Translation in the Americas (AMTA)*, Boston, Massachusetts.

Navarro, G. (2001). A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31–88.

Steinberger, R., Pouliquen, B., Widiger, A., Ignat, C., Erjavec, T., Tufis, D., and Varga, D. (2006). The JRC-Acquis: A multilingual aligned parallel corpus with 20+ languages. In *LREC*.

Zhang, Y. and Vogel, S. (2005). An efficient phrase-to-phrase alignment model for arbitrarily long phrase and large corpora. In *Proceedings of the 10th Conference of the European Association for Machine Translation (EAMT)*, Budapest.