A best-first anagram hashing filter for approximate string matching with generalized edit distance

Malin AHLBERG   Gerlof BOUMA
Språkbanken / Department of Swedish
University of Gothenburg
malin.ahlberg@gu.se, gerlof.bouma@gu.se

ABSTRACT
This paper presents an efficient method for approximate string matching against a lexicon. We define a filter that for each source word selects a small set of target lexical entries, from which the best match is then selected using generalized edit distance, where edit operations can be assigned an arbitrary weight. The filter combines a specialized hash function with best-first search. Our work extends and improves upon a previously proposed hash-based filter, developed for matching with uniform-weight edit distance. We evaluate an approximate matching system implemented with the new best-first filter, by conducting several experiments on a historical corpus and a set of weighted rules taken from the literature. We present running times and discuss how performance varies using different stopping criteria and target lexica. The results show that the filter is suitable for large rule sets and million word corpora, and encourage further development.

KEYWORDS: Approximate string matching, generalized edit distance, anagram hash, spelling variation, historical corpora.

1 Introduction

A common task in text processing is to match tokens in running text to a dictionary, for instance to see if we recognize a token as an existing word or to retrieve further information about the token, like part-of-speech or distributional statistics. Such matching may be approximate: the dictionary entry that we are looking for might use a slightly different spelling than the token at hand. Examples include words containing typos, but also errors introduced by optical character recognition or spelling variation due to the lack of a standardized orthography. Historical corpora, which have gained a lot of interest recently, are an example application where spelling variation is the rule rather than the exception.

A much researched family of approaches to the approximate matching task is based on edit distance, that is, the number of string manipulation operations needed to change a source into a target. For instance, we could count that dictionary entry as a match, whose edit distance from the source token is minimal. If we allow single character insertion, deletion and substitution as edit operations, we end up with the well-known Levenshtein distance. Edit distance calculation is relatively costly, but there is a wide range of literature on efficient algorithms for approximate dictionary matching with Levenshtein and similar distances (Boytsov, 2011, for an overview).

Generalized edit distance refers to a variant in which we allow arbitrary costs for edits. For instance, for processing historical text, we may wish to make the substitution *þ→th* cheaper than *e→th*. The costs can be linguistically motivated, come from empirically estimated probabilities, etcetera. Instead of minimization of the number of operations. the goal is now minimization of the sum of edit costs. In this paper, we propose an adaptation of the *anagram hashing filter* (Reynaert, 2011), an algorithm for efficient matching with the minimum edit distance criterion. Our proposal facilitates approximate matching with minimum *generalized* edit distance. In a short case study on real world material, we highlight different performance aspects of the filter, demonstrate improvement over the original, and show that under reasonable parameter settings our experimental implementation is able to process corpora fast enough for off-line tasks.

2 Anagram hashing

Edit distance calculation with dynamic programming takes time proportional to the product of the source and target lengths. Approximately matching all tokens in a corpus to a fixed lexicon using edit distance is thus potentially very expensive. The naive approach calculates the edit distance for each word in the corpus to each word in the lexicon, which quickly becomes infeasible. To improve this situation, Reynaert (2011) proposes an *anagram hashing filter* to prefilter the lexicon. For each source word, the filter removes target entries that are guaranteed to lie beyond a certain edit distance. Expensive exact calculations then only need to be performed on a small selection of lexical entries. The technique is efficient enough to support spelling correction of multi-million token corpora in a few hours.

The anagram hashing algorithm assigns hash values to strings using a character based function. Characters in the alphabet are assigned an integer value and the hash value of a string is simply the sum of its character values each raised to some predefined constant. Conveniently, edit operations can be performed directly on these hash values: deleting a character means subtracting its hash value from the source's hash value, insertion corresponds to addition and substitution is a combination of both. This carries over to more general operations involving n-grams rather than just single characters. To illustrate, given *kamel* and the substitution *ka → ca* we can obtain the anagram hash value of *camel*. We pick 5 as the power constant.

$$\begin{array}{llrl} \textit{kamel} & 11^5 + 1^5 + 13^5 + 5^5 + 12^5 & 784302 & \\ \textit{ka} \rightarrow \textit{ca} & -(11^5 + 1^5) \ + \ (3^5 + 1^5) & -160808 & \\ \cline{3-3} & & & + \\ \textit{camel} & & 623494 & (= 3^5 + 1^5 + 13^5 + 5^5 + 12^5) \end{array}$$

Hash values for the target vocabulary and all edit operations are precalculated. Given a source string, the anagram hashing filter then creates all permutations up to a small number of edits. Only target entries whose hash values occur in this set of permutations are submitted for exact edit distance calculation. The anagram hash loses information about the order of characters in a string. Therefore, the actual edit distance between source and candidate target may be higher than estimated by the filter. This optimism is what necessitates performing exact calculations.

In the past, researchers have shown the need for approximate matching with weighted edits (Brill and Moore 2000; Toutanova and Moore 2002 for spelling correction, Hauser et al. 2007; Jurish 2010; Adesam et al. 2012 for historical document processing). As said, we then minimize the sum of edit weights rather than the number of edits. Although these two quantities correlate, there are bound to be divergences. Used in this context, the anagram hashing filter may wrongly exclude good targets when they involve many cheap edits. That is, the filter is no longer optimistic with respect to the exact distance. An approximate solution would be to increase the number of permutations the filter goes through, but given the exponential growth of the permutation set for each added numerical edit, this is ineffective, especially when dealing with thousands of weighted rules.

Given a fixed maximum edit distance and rules that have positive cost, the space of hash permutations that has to be explored is finite but possibly extremely large. A proper adaptation of the anagram hash filter should move information about the cost of each individual edit into the filter, so that as little of this permutation space as possible is explored.

3 Best-first anagram hashing

We incorporate weighted rules into the anagram hashing filter by performing a best-first search of the hash permutation space up to a certain cost cutoff – rather than an exhaustive search of all possible hashes up to a given number of edits – returning a stream of anagram hash permutations of ever increasing cost. The cost estimate is based upon the weights of the substitution rules. As before, this estimate is optimistic. The search stops when the search tree is empty because the cutoff point is reached. Alternatively, we may collect top k lists, in which case we can stop when the cost of the current hash permutation as estimated by the filter is higher than the exact cost of the kth-best match found thus far.

We will assume that all edits have the form of n-gram substitutions (but see Sect. 5). The filter estimates so called *restricted* generalized edit distance (Boytsov, 2011), where substitutions never overlap. Pseudocode for the search algorithm is given in Fig. 1. Three further specifications of the procedure outlined in the preceding paragraph where implemented to constrain search space. These are:

1. Although each vertex in the search tree can have as many children as there are applicable substitution rules, we keep the frontier of active vertices to a minimum by **implicitly binarizing the search tree**. At each iteration, we expand the vertex with the lowest cost. Instead of creating a whole new generation at once, we only add the best child (a conjunctive expansion, lines 8–12 in Fig. 1). In addition, we add the best younger sibling (disjunctive expansion, lines 13–18). Each vertex holds a pointer into the sorted list of

Given an anagram hash H for the input word, a maximal cost C, and an ordered list of substitution rules R with associated hash updates, costs and bitmaps.
Returns a stream of anagram hash permutations of increasing cost.

```
1  $q \leftarrow$ new priority queue
2  $u \leftarrow \langle \ldots, \text{rule} = \text{after last rule in} R, \ldots \rangle$   (an infertile dummy vertex)
3  $v \leftarrow \langle \text{cost} = 0, \text{hash} = H, \text{bitmaps} = \{0\}, \text{rule} = \text{first rule in} R, \text{parent} = u \rangle$
4  add $v$ to $q$ with priority $v_{\text{cost}}$
5  **while not** $q$ is empty **do**
6      $v \leftarrow$ next item from $q$
7      **yield** $v_{\text{hash}}$ and $v_{\text{cost}}$
8      **if** $r \leftarrow$ next rule from $v_{\text{rule}}$ where $v_{\text{cost}} + r_{\text{cost}} \leq C$ and $\exists x \in v_{\text{bitmaps}} y \in r_{\text{bitmaps}} [x \text{ AND } y = 0]$
       **then**
9          point $v_{\text{rule}}$ at the position in $R$ behind $r$
10         $w \leftarrow \langle v_{\text{cost}} + r_{\text{cost}}, v_{\text{hash}} + r_{\text{hash}}, \{x \text{ OR } y \mid x \in v_{\text{bitmaps}} \wedge y \in r_{\text{bitmaps}} \wedge x \text{ AND } y = 0\}, r, v \rangle$
11         add $w$ to $q$ with priority $w_{\text{cost}}$
12     **end if**
13     $u \leftarrow v_{\text{parent}}$
14     **if** $r \leftarrow$ next rule from $u_{\text{rule}}$ where $u_{\text{cost}} + r_{\text{cost}} \leq C$ and $\exists x \in u_{\text{bitmaps}} y \in r_{\text{bitmaps}} [x \text{ AND } y = 0]$
       **then**
15         point $u_{\text{rule}}$ at the position in $R$ behind $r$
16         $w \leftarrow \langle u_{\text{cost}} + r_{\text{cost}}, u_{\text{hash}} + r_{\text{hash}}, \{x \text{ OR } y \mid x \in u_{\text{bitmaps}} \wedge y \in r_{\text{bitmaps}} \wedge x \text{ AND } y = 0\}, r, u \rangle$
17         add $w$ to $q$ with priority $w_{\text{cost}}$
18     **end if**
19 **end while**
```

Figure 1: Best first exploration of the anagram hashing search space

rules to know what the next rule to consider is. A fertile vertex is one whose rule pointer is not at the end of the rule list. In order to create siblings, we keep around and update parent vertices for as long as they are fertile.

2. We compact the rule set by **fusing substitutions with identical anagram hash updates**. The same substitution can target several positions in the same source (*e*→*a* in *th<u>e</u>atr<u>e</u>*), two substitutions may be the same but for unchanged context material (*e*→*a* and *tre*→*tra*), substitutions can be anagrams of each other (*th*→*t*, *ht*→*t*), or two substitutions may accidentally give the same hash update. All such cases are gathered into one effective rule. The associated cost is that of the cheapest constituting substitution, so that substitution fusion does not lead to overestimation of the actual edit distance.

3. We prune the search tree by **keeping track of overlap between substitutions**. In restricted edit distance, substitutions that target the same source characters cannot be combined. Thus, when creating a vertex, we need not consider the more expensive rules that overlap with any of the cheaper rules used for vertex' ancestors. We use bitmaps to record which source characters have been used in the creation of a vertex and to mark the characters appearing in a substitution's left hand side. Substitution fusing means that vertices and rules have sets of bitmaps, which represent disjunctions of substitutions. A rule's overlap with previous substitutions is checked using bitwise AND (line 8 respectively

line 14). Updating the bitmap set uses bitwise OR (line 10 respectively line 16). To illustrate this update (we show strings rather than hashes for readability):

start with *theatre*: {0000000}
apply $e \rightarrow a$: {0010000,0000001} to get *thaatre*, *theatra*: {0010000,0000001}
apply *tre*→*te*: {0000111} to get *thaate*: {0010111}.

When fusing substitutions into one rule, only the most general bitmaps are included in its bitmap set, because they determine its combinatorial possibilities. For instance, putting two substitutions 01100 and 00100 into one rule gives {00100}.[1]

4 Experiments

To get an impression of the effectiveness of the best first anagram hash filter, we performed approximate string matching experiments using a real world data set (Adesam et al., 2012). In total, there are 6045 weighted substitution rules of the form $n \rightarrow m$, where n and m are uni-, bi- or trigrams, possibly of unequal length. The corpus is made up of a collection of Old Swedish texts, comprising 162k types in 3Mln tokens. As lexicon we first use a combination of three Old Swedish dictionaries with 54k entries in total and later the corpus itself.

The approximate matching program is implemented in Python, including the dynamic programming code for the exact edit distance calculations, but excluding the search algorithm given in Fig. 1, which was implemented in Cython (Bradshaw et al.). The experiments ran on a single core of 2.7 Ghz Intel CoreT i7, with 4 GByte of memory running 32-bit Linux. Across experiments, the implementation used about 1GByte at most for the search space.

We used an integer representation of the cost of a vertex during search, corresponding to 6 decimal places precision. This, in combination with the fixed cutoff point and the fact that vertex cost never decreases during the search, allows us to use a very simple implementation of the priority queue based on an array of linked lists. Each index in the array points to the vertices of that cost. Profiling suggests that the filter accounts for 20–60% of running time, depending on the experiment.

We ran the whole corpus against the dictionaries of Old Swedish, with different stopping criteria. The dictionary experiments are summarized in Table 1. Since the substitution costs are in the 0.2–0.9 range, cutoff levels of 1.5 and 2.0 respectively allow matches that require more than just 1 or 2 edits. The number of rules submitted to the filter is fixed by the corpus and the set of substitutions. On average, 194 rules, fused from 450 applicable substitutions, per type were input to the filter.

With cutoff 1.5 and stopping after the best match, matching all 162k types against the 54k entry dictionary takes 106m49s, a throughput of 25 types/s, or 85 tokens/s.[2] The top slice in Table 1 gives the distribution of matches. We find 0.7 matches per type, taking up to 6 edits, but most of them occurring 1 or 2 edits away. On average, the filter produces 21k hash permutations (including possible duplicates), of which an average of 22 hashes (no duplicates) are found in the lexicon's hash table and thus trigger exact edit distance calculation. Looking for the top 3

[1]One might consider doing the same during the search process, for each vertex expansion. Testing has shown that the overhead incurred is too high to increase performance

[2]We matched the corpus by type. To give an impression of speeds when applied to running text, token throughput is calculated by taking the weighted average over per type processing times, with weights corresponding to token occurrences of that type.

	Edits								
	0	1	2	3	4	5	6	7	Total
cutoff 1.5, best match									
#	8887	41020	46173	19279	2216	86	2		117663
%	7.6	34.9	39.2	16.4	1.9	0.1	<.1		
Σ%	7.6	42.4	81.7	98.0	99.9	100.0	100.0		
cutoff 1.5, top 3									
#	8 887	81 153	140 298	67 638	7 547	258	6		305 787
%	2.9	26.5	45.9	22.1	2.5	0.1	<.1		
Σ%	2.9	29.4	75.3	97.4	99.9	100.0	100.0		
cutoff 1.5, full search									
#	8 887	129 517	1 294 345	1 339 331	97 440	1 826	7		2 871 353
%	0.3	4.5	45.1	46.6	3.4	<.1	<.1		
Σ%	0.3	4.8	49.9	96.5	99.9	100.0	100.0		
cutoff 2.0, top 3									
#	8 887	81 199	140 388	103 580	37 492	4 035	162	1	375 744
%	2.4	21.6	37.4	27.6	10.0	1.1	<.1	<.1	
Σ%	2.4	24.0	61.3	88.9	98.9	100.0	100.0	100.0	

Table 1: Matching against the dictionary with different stopping criteria. Distribution in terms of number of required edits per cutoff. #: counts, %: proportion, Σ%: cumulative proportion.

matches at cutoff 1.5 takes 209m36s (13 types/s, 41 tokens/s). As the second slice in table 1 shows, we now find 1.8 matches per type. The filter returns 32k hash permutations, of which 51 are found in the lexicon's hash table, on average.

We also asked for all matches at cutoff 1.5. Running time now more than doubles to 491m52s, or 5.5 types/s. The filter only accounts for about 1/5th of the running time. The filter creates 46k permutations, of which 219 trigger exact edit distance calculations, on average. These exact calculations dominate running time completely in this experiment and explain the slow down. The impact of the exact calculation cost is most clearly seen in the token throughput, which counter-intuitively is *lower* than the type throughput at 4.4 tokens/s. We can explain this from the high lexical neighbourhood density of short words, which also are frequent words. For instance, for types of length 3 and 4, we submit on average over 500 targets for exact calculation, more than double the total average, even though the number of hash permutations generated is below average for these words.

The distribution of matches for this exhaustive run is in the third slice in Table 1. The exhaustive search finds almost 18 matches per type at this cutoff level. The greater proportion of matches is found at a higher number of edits compared to earlier experiments. This is expected, because there is some correlation between cost and the number of edits. Note that to capture 99% of the matches under cutoff 1.5, an exhaustive run with the original anagram hashing filter would have to examine up to four edits; in the order of $194^4 \approx 1.4 \cdot 10^9$ hash permutations on our rule set. Merely generating this number of permutations would be prohibitively expensive, let alone considering the subset of lexicon matches for exact calculation.

We ran two further experiments with higher cutoff points. First, we looked for the top 3 matches with cutoff 2.0. Processing took 1992m27s, giving a throughput of 1.4 types/s, 7.9 tokens/s – a dramatic slow down compared to the cutoff 1.5, top 3 experiment, considering we only find about 25% more matches (bottom slice, Table 1). The explanation lies in the number of hash permutations returned by the filter, which increased 20-fold to 640k,[3] of which 217 hashes per type are found in the lexicon, on average. At these cost levels, the density of the search space increases as the estimated cost increases. That is, there are many more hash permutations in the 1.5–2.0 range than there are in the 1.0–1.5 range.

Secondly and finally, we used the corpus itself as the vocabulary, to study the impact of a larger lexicon size on performance. There is no direct effect on the filter itself, but we can expect a higher proportion of hash permutations to exist in the lexicon. On the one hand, this will trigger more of the expensive exact calculations. On the other, this also means we may be able to stop earlier on average if we use the top 3 stopping criterion. With cutoff 2.0, top 3,[4] running time shows that the early stopping effect dominates: 823m46s, 3.3 types/s, 30 tokens/s.

We have seen that the best-first anagram hash filter allows us to efficiently search for approximate matches in a fixed vocabulary when the distance function is defined by weighted substitution rules rather than by the number of edit operations. Even in our experimental implementation, which has important bottlenecks in the Python code, we are able to achieve throughput high enough to enable the processing of medium-large corpora using a large rule set. Although the filter itself is fixed by the source string, the number of applicable substitutions and the cutoff point, we note that overall performance is very dependent on the size of, and distribution of items in, the lexicon.

5 Comparison to related work

There is a very rich literature on approximate matching against a fixed vocabulary, but work on approximate matching using edit distance tends to focus on unweighted edits (Boytsov, 2011). We will briefly compare our proposal against the implementations used by the research mentioned in Sect. 2, that motivated the use of generalized edit distance.

As mentioned, in its original form the anagram hashing filter only considers hash permutations within a fixed number of edits, irrespective the weight of the operation. Reynaert (2011), however, does apply a more fine-grained ranking to candidate targets that pass the anagram hashing filter, in a way simulating weighted edits. Although effective in that case, the limits to this approach were explained in Sec.2. Until now, we have focused on substitutions, but like the original anagram hash filter, our best-first adaptation can easily incorporate insertions and deletions. Like with substitutions, the bitmap for deletions (substitution with ϵ) marks the deleted characters as used. The bitmap for insertions would be 0, as they always apply (0 AND $x = 0$) and do not change the source bitmap (0 OR $x = x$).

The spelling correction method presented in Brill and Moore (2000) relies on weighted substitution rules much like the ones we used in our case study. The authors report best-match

[3]The maximum number of examined permutations was 40Mln, incurred for a falsely segmented token. Such false tokens are long and almost guaranteed to lead to a search through the whole space up to the cutoff point as they do not have any suitable matches. As we do not look at the *quality* of the matches in this paper, we have left these pathological cases in.

[4]We ignored the first match, effectively running top 4, since the first and best match is now always the word itself. To compare, only 2.4 of matches against the dictionary are at 0 edits distance (see Table 1, bottom slice).

processing speeds of 20 types/s from an implementation using trie-based precompilation of the lexicon and the weighted substitutions. An interesting aspect to this work is the differential weighting of substitution rules depending on where they apply in the string. A quick way to incorporate this into our setup is to use the lowest weight for a rule in the filter, and differentiate according to position in the exact distance calculation stage.

A more recent proposal using approximate matching with weighted edits is Jurish (2010, and refs. therein). Formally, the matching is defined as a composition of three weighted finite state transducers, representing the input word, the edit operations and the lexicon. Instead of actually compiling the resulting transducer – which would be too resource intensive – cheapest paths through the pipeline are calculated on-line using an adapted Dijkstra-algorithm. The author reports processing speeds of 50 tokens/s for Levenshtein distance-based approximate matching. Considerably faster processing is reported for smaller sets of specialized rewrite rules, although it is unclear how the technique would scale when using a very large rule set like in our experiments. However, a real advantage of representing the vocabulary as an automaton is the ability to handle vocabularies of non-finite size, by modeling productive compounding in the automaton. As far as we can see, an anagram hash filter-based approach is not able to accommodate non-finite target vocabularies.

A well-known application for computer-supported orthographic normalization is VARD2, which employs an ensemble of techniques to match tokens against a fixed vocabulary. The authors also use Levenshtein distance, although they consider it to be too computationally expensive to apply to the whole dictionary (Baron and Rayson, 2009, Sect. 3). As Reynaert's (2011) and our work shows, this need not be the case. Of course, if a hybrid approach improves the accuracy of the automatic normalization, our method could well be part of such an ensemble.

Conclusions

We have presented a best first anagram hashing filter for generalized edit distance matching. The algorithm prefilters a lexicon to narrow down the search for a best match given a set of weighted edit operations. Several experiments have shown the efficiency of the filter and the way it interacts with the rest of the system, such as the size and layout of the lexicon, and parameters setting the maximum cost and the number of matches desired. Even though the system's bottleneck is implemented in Python and we tested using a large rule set, we achieve throughput at reasonable parameter settings good enough for off-line processing of million word corpora. At the most advantageous settings, we are able to process 25 types/s, corresponding to 85 tokens/s. The fact that there are enough opportunities left for easy improvements in running time makes these results especially encouraging.

Algorithmic improvements on the filter itself to be researched include the way dependencies between rules are tracked. As it is, the calculation of the Cartesian product of bitmap sets when checking for rule applicability causes considerable overhead. A further topic for investigation is ways to extend the filter to unrestricted edit distance, so that edits on the source string may feed each other, or, similarly, to allow edits with a context specification that is not formally part of the substitution.

Acknowledgments

This work was carried out in the context of the Center for Language Technology, of Gothenburg University and Chalmers University of Technology. The authors thank Yvonne Adesam for feedback and for collaboration on the Old Swedish data.

References

Adesam, Y., Ahlberg, M., and Bouma, G. (2012). *bokstaffua, bokstaffwa, bokstafwa, bokstaua, bokstawa...* Towards lexical link-up for a corpus of Old Swedish. In Declerck, T., Krenn, B., and Mörth, K., editors, *Proceedings of LTHist 2012*.

Baron, A. and Rayson, P. (2009). Automatic standardisation of texts containing spelling variation: How much training data do you need? In *Proceedings of the Corpus Linguistics Conference*, Lancaster.

Boytsov, L. (2011). Indexing methods for approximate dictionary searching: Comparative analysis. *Journal Experimental Algorithmics*, 16(1):1–91.

Bradshaw, R., Behnel, S., Seljebotn, D. S., Ewing, G., et al. The Cython compiler. Software. `http://cython.org`.

Brill, E. and Moore, R. C. (2000). An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics*, pages 286–293, Hong Kong. Association for Computational Linguistics.

Hauser, A., Heller, M., Leiss, E., Schulz, K., and Wanzeck, C. (2007). Information access to historical documents from the early new high german period. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-2007) Workshop on Analytics for Noisy Unstructured Text Data*, Hyderabad, India.

Jurish, B. (2010). Comparing canonicalizations of historical German text. In *Proceedings of the 11th Meeting of the ACL Special Interest Group on Computational Morphology and Phonology*, pages 72–77, Uppsala, Sweden. Association for Computational Linguistics.

Reynaert, M. (2011). Character confusion versus focus word-based correction of spelling and OCR variants in corpora. *International Journal on Document Analysis and Recognition*, 14:173–187. 10.1007/s10032-010-0133-5.

Toutanova, K. and Moore, R. (2002). Pronunciation modeling for improved spelling correction. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 144–151, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.