Improved Fast Similarity Search in Dictionaries [⋆]

Daniel Karch, Dennis Luxen, and Peter Sanders

Karlsruhe Institute of Technology
danielkarch@gmail.com, {luxen, sanders}@kit.edu

Abstract. We engineer an algorithm to solve the approximate dictionary matching problem. Given a list of words $\mathcal{W}$, maximum distance d fixed at preprocessing time and a query word q, we would like to retrieve all words from $\mathcal{W}$ that can be transformed into q with d or less edit operations. We present data structures that support fault tolerant queries by generating an index. On top of that, we present a generalization of the method that eases memory consumption and preprocessing time significantly. At the same time, running times of queries are virtually unaffected. We are able to match in lists of hundreds of thousands of words and beyond within microseconds for reasonable distances.

1 Introduction and Previous Results

The problem of searching approximate of matches in a dictionary arises in many fields. Most common is the search for the so called best match. The problem has many applications. For example, Google's 'Did you mean' feature catches typos in search queries. But in some settings, the uncertainty is higher and therefore one is not interested in the best match, but also in other matches which are still within a certain distance from the query. An interesting application is a geocoding application that maps perhaps misspelled locations descriptions to geocoordinates.

Each word is represented by a string of characters over a finite alphabet Σ. The Levenshtein distance [1] $ed(a, b)$ defines a metric between two words $a, b \in \Sigma^*$ and is used in this work to compute the distance between two words.

The most trivial algorithm to solve the problem is scanning sequentially through the input list and noting the best match(es) at each entry. The running time is obvious and consists of a linear number of distance computations and searching an entire directory on a standard desktop computer takes only a few seconds even for dictionaries up to a few hundred thousand or even a million entries. But in many settings this is too much, because queries arrive in a high frequency. For example, a web search engine only has a few milliseconds to process a single request and does not have the time to do exhaustive searching in a large dictionary.

[⋆] Partially supported by DFG Grant 933/5-1. A short version of the paper was presented at the 17th Symposium on String Processing and Information Retrieval (Spire 2010). The authors would like to thank Johannes Fischer for the fruitful discussions and the anonymous referees for the detailed reviews.

Since these distance computations are rather expensive, it is natural to find an algorithm that does not compare the input to the entire dictionary, but only a few entries. A so-called *filter* represents a criterion to quickly discard large portions of the search space.

The exploitation of the underlying metric space implied by the edit distance [2] is easy. The set of words is partitioned by the distance of each element to a more or less carefully chosen and perhaps random pivot element. By computing the distance to the pivot, the search space is pruned using the triangle inequality. However, this approach has limited effect, e.g. in natural language dictionaries. Distances of most dictionary elements to the pivot lie in a small range and pruning has a limited effect.

To cope with the limitations, different schemes were introduced from using multiple pivots to tree-like data structures. The oldest of such trees is the BK-tree data structure proposed by Burkhard and Keller [3], which is built recursively. A root is selected whose subtrees are identified by distance values to the root. The i-th subtree consists of elements of the dictionary at distance i to the root. The subtrees are recursively built until the number of elements in a subtree is below some threshold. Again, the triangle inequality is used to branch into or cut any subtrees. A candidate set of possible matches is built by the union of all leaves that are reached by the tree traversal. A rather weak result is that BK-trees and its refinements need $O(n^\alpha)$, $0 < \alpha < 1$, comparisons and node traversals on average [2] for a dictionary of n entries. See Chávez et al. 's publication [4] for a survey.

The general problem of approximately matching words can be further refined into two categories, namely matching elements from a set of words or matching arbitrary patterns in strings [2]. As usual, in high dimensional search problems there is a severe space-time trade-off. Cole et al. [5] give a solution for the dictionary matching problem using $O(n \log^d n)$ space and answer a query in $O(m \cdot \log \log n + occ)$ for a dictionary of size n, query length m, edit distance d. Here, *occ* is the number of occurrences of the pattern. Mihov and Schulz [6] present a sophisticated but complicated method to solve the problem with universal Levenshtein automata. Russo et al. [7] propose a compressed index that performs well for $d = 1, 2, 3$, but needs several seconds to perform queries for larger d. The best known linear space solution needs $O(m^{d-1} \log n \log \log n + occ)$ query time [8] for error $d \geq 2$. However, this solution is fairly complicated and involves large constant factors, and to our knowledge there aren't any implementations yet. Furthermore, any of the general-purpose approximate string matching algorithms have to be adapted to perform dictionary matching: Either the query has to be adapted to ensure that only complete words are found, or special characters have to be introduced to mark the start and end of a dictionary entry.

More practically oriented work has focused on filtering algorithms that take linear space, but these do not have strong worst case performance guarantees. Kärkkäinen and Na [9,10] report on a linear space data structure that supports substring search, but has much larger query times compared to our result. Ukkonen [11] investigated suffix trees as a building block to solve the problem.

Likewise, Cobbs [12] gives a data structure based on suffix trees with linear time preprocessing for a fixed size alphabet for searching fixed patterns. Queries to the data structure can be answered in time $O(mq + occ)$, where m is the length of the pattern, $q \leq n$ and again occ is the number of occurrences.

A technique involving so called q-grams is popular among practitioners. But it generally works for the Hamming distance only. q-grams are sub-words of length q and the q-gram distance (or similarity) is defined by the number of q-grams two words share. A generalization of this technique are gapped q-grams. Taking q letters from a word as before and introducing *don't care* defines a pattern instead of sub-word. These don't care positions are then called gaps. In [13] it is shown that one-gapped q-grams can be extended to obey the edit distance metric. One of the major difficulties of gapped q-grams is the computation of a threshold which is the smallest number of matching q-grams between a pattern and a text. Most experimental work focuses on finding this threshold, e.g. [14,15].

For more information on approximate string matching see [9,16,17,18].

To speed up edit distance computation itself, research focused on simple and practical bit-vector algorithms [19]. Words of character length n with d or fewer differences can be matched in $O(nmd/w)$, where w is the word size of the machine an m the length of a query. This is done by computing the bit representation of the current state-set of the k-difference automaton. The running time was further improved to (nm/w) [20] and further refinements [20] yield an $O(dn/w)$ expected-time algorithm for arbitrary large m.

The remaining parts of this paper are structured as follows. Section 2 gives an introduction into the neighborhood relation on strings that we exploit. It is followed by an discussion of our experimental results in Section 4. Finally, Section 5 draws conclusions and identifies future work.

2 Approximate Dictionary Matching

[ps: changed notation! $d \to \delta$, $t \to d$ to be compatible with what follows.] Our method can be seen as an implementation of a general approach to approximate matching known as *(lossless) filtering.* This can be formalized as follows: Given a set $\mathcal{S}$ of words over a finite alphabet Σ, a metric $\delta : \Sigma^* \times \Sigma^* \to \mathbb{R}_0$, and an error threshold d, a preprocessing algorithm produces a data structure that allows fast evaluation of a function $F : \Sigma^* \to \mathcal{P}(\mathcal{S})$. For a query word $q \in \Sigma^*$, $F(q)$ computes a set of candidate words from $\mathcal{S}$ such that the set of approximate matches $\{s \in \mathcal{S} : \delta(q, s) \leq d\}$ is a subset of $F(q)$. ?

Deletion Neighborhood. We improve a filtering technique called *Fast Similarity Search (FastSS)* [21] which is a generalization of a single error method proposed by Mor and Fraenkel [22].

For integer d and a word $w \in \Sigma^*$ the *d-(deletion-)neighborhood* $\mathcal{N}_d(w)$ is defined as the set of all subwords of w with exactly d deleted positions. Each element of $\mathcal{N}_d(w)$ is called a *residual string.* Furthermore, a string w is called originating string for residual r if and only if $r \in \mathcal{N}_d(w)$. We obtain a lossless filter for a set of words $\mathcal{S}$ by precomputing the d-neighborhoods of strings in $\mathcal{S}$. As a filtering function, we obtain $F(q) = \{s \in \mathcal{S} : \mathcal{N}_d(s) \cap \mathcal{N}_d(q) \neq \emptyset\}$.

? The correctness of this definition follows from the following Lemma:[Beweis vorgezogen]

Lemma 1. *If two words $u, v \in \Sigma^*$ are within a distance d from each other, then there exists a word w which has length at least $|u| - d$ and consists of letters from u and v in their original order. Assume that u is at least as long as v.*

We use the concept of *Ordered Edit Sequences* [16] to show the claim. Our proof is simpler and more intuitive than the proof from [21].

Proof. Recall that the edit distance is said to be the minimal number of edit operations to transform one word $u \in \Sigma^*$ into another $v \in \Sigma^*$. The set of operations available for any single transformation are $op = \{\mathsf{ins}, \mathsf{del}, \mathsf{chg}\} : \Sigma \cup \{\epsilon\} \rightarrow \Sigma \cup \{\epsilon\}$ with $v = op_d(op_{d-1}(\ldots(op_1(u))\ldots))$. The sequence $\rho(u, v) = (op_1, op_2, \ldots, op_d)$ is called edit sequence and we call it ordered if the operations are applied from left to right. We define $pos(\cdot)$ to give the position of an operation within the edit sequence. In other words $\forall i : (pos(op_i) \leq pos(op_{i+1}))$. By definition of the edit distance metric there exists an edit sequence of minimal length. Now, we can show Lemma 1. Since $ed(u, v) \leq d$ it follows that the length of a minimal ordered edit sequence is at most d, which means $|\rho_{min}(u, v)| \leq d$ is the length of a minimal edit sequence. This implies that v is changed at no more than d positions. By deleting these at most d positions from v, we get a string w, which has length at least $|u| - d$ and preserves the letter ordering from u and v. □

Basic Data Structure. A static index data structure is generated in a precomputation phase that can be queried during an on-line phase. We insert a number of values into a hash table that is part of our data structure. The structure utilizes the hash table to store pointers to originating dictionary entries at the hash values of residual strings. If any hash value has more than one originating dictionary entry then the corresponding pointers are stored in a list. Figure 1 sketches the internal structure of the index.

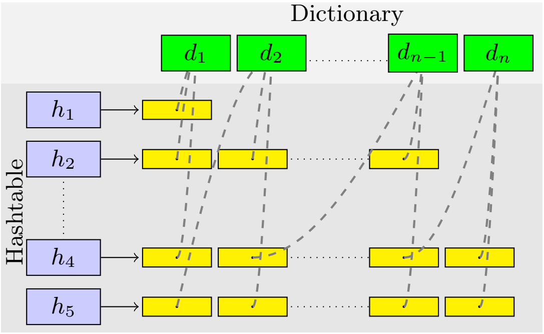

Fig. 1. Approximate string matching data structure.

Query. For an input query q and maximum distance d, the corresponding d-neighborhood and its hash values are computed. If any element of the query's residuals is also an element of the data structure then the pointers to the originating dictionary entries give a set of candidates. Each of those might be an approximate match. Once the candidate set is completely built, it is searched exhaustively by computing the edit distance of each candidate to the query. By removing all elements from the candidate set whose distance is larger than the threshold d we get the set of all dictionary members that are at most a distance d away from query q. Perhaps there exists an additional order on the candidates stemming from the application. The algorithm can be adapted to not only return the best match, but also a list of those candidates that are sufficiently close.

Precomputation. We compute the d-neighborhood of each element of the input dictionary and insert the resulting information into our index data structure. Doing this precomputation naively and storing all residual strings in a data structure takes up an enormous amount of space. Instead, we use hashing and reduce each element of the residual neighborhood into an integer number. We insert pointers to the originating dictionary entries into the hash table at the respective hash values of all residual strings. Therefore, only constant space is needed per residual string regardless of the length of that string. We now present an improvement to the algorithm.

Algorithmic Generalization. We limit the number of elements that are inserted into the index while staying lossless. To do so, we split long input words in half, compute the residual strings with half the number of errors, and adapt the query algorithm, which will be explained in this Section. See Section 4 for an analysis of the threshold value m, which indicates whether or not to split a word. Instead of generating $\binom{|s|}{d}$ hash values we insert only

$$\binom{|s|}{\lfloor \frac{d}{2} \rfloor} + \binom{|s|}{\lceil \frac{d}{2} \rceil}$$

values for a split dictionary entry s. The *generalized d-neighborhood* of $w' \in \Sigma^*$ is the set of residuals that is found by computing all combinations of $\lceil \frac{d}{2} \rceil$ deleted characters for the first and second half of w'.

The generation of the index is simple. But we have to pay some extra care at query time, because insertions and deletions that transform words w into w' can take place at arbitrary positions. As a consequence, we can not rely on the length of a query q to decide whether it has been split or not. Instead of splitting a query q of length l at a fixed position, it is split several times in half at positions in the interval of $\lceil \frac{l}{2} \rceil \pm \lceil \frac{d}{2} \rceil$. Also, the allowed error is halved. If the length of an input word is within $m \pm d$ then the index is also searched for the non-split string.

Consider these definitions. Let $w \in \Sigma^*$ be an entry of dictionary D and d the maximum allowed error. Let $u = p(w)$ and $v = s(w)$ denote the first and second half of the split word w. Prefixes u and suffix v are indexed, while q is the query.

Any query q is split at several positions as explained above and we define $\mathcal{P}(w)$ to be the set of first and $\mathcal{S}(w)$ to be the set of second halves. Our method is still correct since we can show the existence of a common residual string for either the prefix or the suffix of a split query word by the following Lemma.

Lemma 2. *Let $q \in \Sigma^*, w = uv$ with $ed(w, q) \leq d$. Consider $\mathcal{P}(q)$ $(\mathcal{S}(q))$ to be the set of $\lceil \frac{d}{2} \rceil$ many prefixes (suffixes) of q that are generated for each query to the index. Then there exists at least one pair (p', s') with $p' \in \mathcal{P}(q)$, $s' \in \mathcal{S}(q)$ and $p' \circ s' = q$ of prefix-suffix-elements for which either $ed\,(u, p') \leq \lceil d/2 \rceil$ or $ed\,(v, s') \leq \lceil d/2 \rceil$. It suffices to test the split positions from the interval $\lceil \frac{|q|}{2} \rceil \pm \lceil \frac{|d|}{2} \rceil$ to find that pair.*

Proof. Consider the edit sequence S that transforms w into q and that has length at most d, s.t. $ed(w, q) \leq d$. String w is split at position $\lceil \frac{|w|}{2} \rceil$ into $w = p \circ s$. Note that the lengths of p and s differ at most 1. Sequence S is applied to $w = p \circ s$ and yields $q = p' \circ s'$. Hence, either $ed(p, p') \leq \lceil \frac{d}{2} \rceil$ or $ed(s, s') \leq \lceil \frac{d}{2} \rceil$ or both. The algorithm has to split query q exactly into p' and s' to guarentee that a match is found. Assume that it doesn't suffice to test the interval $\lceil \frac{|q|}{2} \rceil \pm \lceil \frac{|d|}{2} \rceil$ to find the correct splitting position. Then p' is either shorter than $\lceil \frac{m}{2} \rceil - \lceil \frac{d}{2} \rceil$ or longer than $\lceil \frac{m}{2} \rceil + \lceil \frac{d}{2} \rceil$. Assume $|p'| < \lceil \frac{m}{2} \rceil - \lceil \frac{d}{2} \rceil$. Then
$\Rightarrow |s'| > \lceil \frac{m}{2} \rceil + \lceil \frac{d}{2} \rceil$
$\Rightarrow |p'| + \lceil \frac{d}{2} \rceil < \frac{m}{2} < |s'| - \lceil \frac{d}{2} \rceil$
$\Leftrightarrow |p'| + \lceil \frac{d}{2} \rceil < |s'| - \frac{d}{2} \Leftrightarrow |p'| < |s'| - 2 \cdot \lceil \frac{d}{2} \rceil \Leftrightarrow |p'| - |s'| < -2 \cdot \lceil \frac{d}{2} \rceil$
$\Leftrightarrow |s'| - |p'| > 2 \cdot \lceil \frac{d}{2} \rceil$
This implies that the lengths of s' and p' differ by more than $2 \cdot \lceil \frac{d}{2} \rceil$. But then edit sequence S has to be longer than $2 \cdot \lceil \frac{d}{2} \rceil$ operations, because length difference is a lower bound for edit distance. The other case for $|p'| > \lceil \frac{m}{2} \rceil + \lceil \frac{d}{2} \rceil$ follows by the same line of argumentation. □

Wu and Manber [23] use partitioning into $d+1$ pieces to match one of the pieces with no error, while Navarro and Baeza-Yates [24] gave a recursive partitioning scheme for fast on-line approximate string matching.

See Section 4 for an experimental analysis of the generalization that shows it uses half the space than our implementation of the original algorithm and maintains stable query performance.

3 Analysis

Our variant makes heavy use of hashing as we argued before and we analyze the penalty of our approach coming from hash collisions. First, consider the case that we do not split the input string, which resembles the original method.

For each dictionary entry of length ℓ, we insert at most $\binom{\ell}{d}$ constant size entries into the hash table. The hash table needs $O(1)$ space per element since the bit size of each entry is of constant size. Note that for $d = 1$ we obtain overall linear space because $O(\ell)$ constant size hash table entries are stored for a dictionary entry of size ℓ.

We resort to average case analysis for the query time using the following model: Consider a dictionary of n words drawn uniformly at random from Σ^ℓ and an arbitrary query word q of length ℓ. In real world inputs, we have a mix of words with different lengths. However, a query of length ℓ will mostly return candidates of length ℓ for random inputs. Hence, there is no need to postulate anything on the distribution of lengths – we just analyze the system for each length separately.

Assume an order in which the residuals of a word can be generated. Consider the 0/1 random variable X_{ijk} that has value one iff the i-th residual of query q is equal to the j-th residual of the input word k. The total number of residuals that need to be considered is bounded by

$$X := \sum_{i=1}^{\binom{\ell}{d}} \sum_{j=1}^{\binom{\ell}{d}} \sum_{k=1}^{n} X_{ijk}.$$

This is an overestimation of the actual number of residuals to be considered since by deleting different sets of characters we might arrive at the same residual. However, for not too small Σ this only happens rarely[was "this is only slightly" ???]. Let σ denote the size of the alphabet actually used. We have $P[X_{ijk} = 1] = 1/\sigma^{\ell-d} = \sigma^{d-\ell}$. Hence, using the linearity of expectation, we get an expected value of ?

$$\mathbf{E}[X] = n\binom{\ell}{d}^2 \sigma^{d-\ell} \tag{1}$$

This gives the number of residuals we have to consider. The number of actual distance computations may be smaller since several residuals of q may match several residuals of a dictionary entry s_k, but we will compute the distance $d(s_k, q)$ only once.

An interesting consequence of (1) is that, on average, we can expect a speedup over the naive algorithm that is independent of the size of the input dictionary. By applying the Markov inequality, we can estimate an upper bound of the probability that the expected number is not a fraction of n. Let c be a constant > 0.

$$P\left[X \geq \frac{n}{c}\right] \leq c^{-1}\binom{\ell}{d}^2 \sigma^{d-\ell} \ . \tag{2}$$

See Section 4, where we experimentally analyze the behavior of the algorithm for varying splitting parameters.

4 Experimental Results

Implementation Details. We implemented the data structure, the construction and query algorithms in C++ using GCC Compiler version 4.3.2. We hashed all

dictionary	no. elements	avg. length	size [MiB]
mobydick	37 924	9	0.31
town	47 339	10	0.49
english	213 557	10	2.20
wikipedia	1 812 365	9	17.06

Table 1. Basic information on our dictionaries.

residual strings with the built-in hash function of the Boost library v1.36 to a 32-Bit Integer and chained with a simple linear congruence.

The exhaustive search of the candidate set is done by a simple implementation of the Levenshtein distance. It computes a band of width $2d + 1$ only. This way we compute the distance exactly only if it is smaller than d and return otherwise as soon as we get a certificate that the distance is larger than d. Since we need $O(1)$ to fill a cell in the distance table, we can verify a candidate in $O(d \cdot l)$, where l is the length of the shorter word. In the experiments it took less than a microsecond to verify any single candidate.

Environment. All of our tests were conducted on a single core of a Intel Xeon X5550 CPU, running a version 2.6.27 Linux kernel. We compare the performance of our optimizations against our own implementation only for reasons of fairness.

Test Instances. The sizes of the dictionaries used in the experiments range between about 38 000 and 1.8 million entries (see Table 4). All results were averaged over a number of queries of perturbed dictionary entries. The word list *mobydick* consists of the distinct words from Melvilles classic novel, the *town* dictionary consists of German town names extracted from the OpenStreetMap project[1] in February 2009, the *english* dictionary is an extract of words from Webster's English Dictionary and the *wikipedia* dictionary is the list of pairwise distinct words from all english Wikipedia[2] titles as of February 2009. Table 4 lists element count and average word length of each test data set.

4.1 Splitting Parameter

Preprocessing Space. We analyze the amount of distinct residuals that are generated for each value of $m \in 1, \dots, 30$ and the average duration of a single query against this index. To do so, we averaged over 1 000 randomized queries. Both value $m = 1$ and $m = 30$ resemble worst cases. We present the results in the plots of Figure 2 for edit distance 3. Other distances show similar behavior. Note that we omitted the lower and upper values of m for clearer arrangement, because for the values $1, \dots, 5$ $(20, \dots, 30)$ nearly all (none) strings get split. We present selected plots that show the experiments. Note the logarithmic scales for query times. In all the experiments we see that there is a trade-off between the

[1] `http://www.openstreetmap.org/`

[2] `http://www.wikipedia.org`

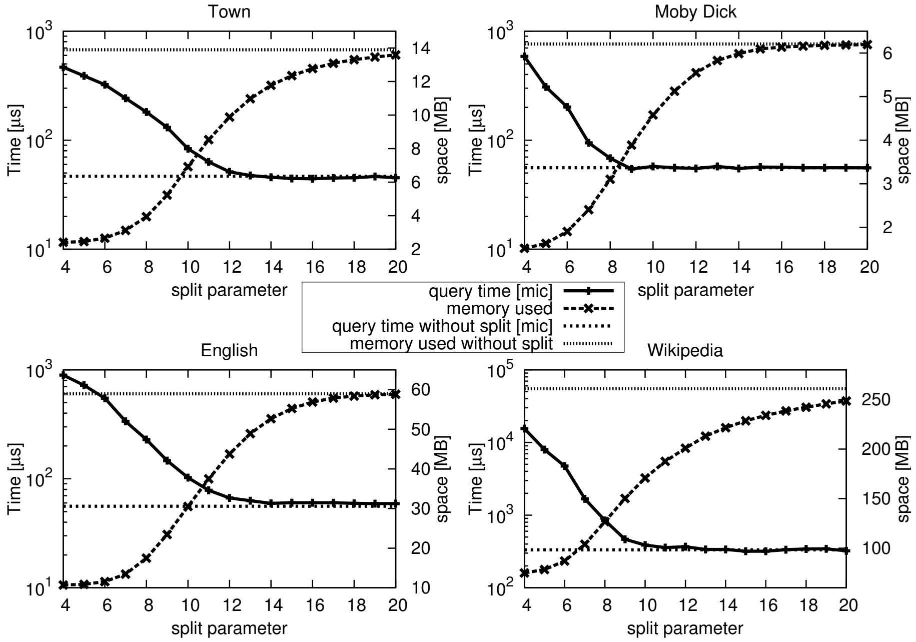

Fig. 2. Analysis of the Splitting Parameter for $d = 2$

memory consumption and average query time. The split parameter functions as an adjusting value to choose between size of the index and query performance. Our analysis shows that the index size can be halved by degrading the speed of an average query within acceptable limit only. Especially, when splitting is restricted to those dictionary entries whose length is larger than the average, we can halve the memory consumption of the index. The query performance is virtually unaffected.

Preprocessing Time. We investigated preprocessing times with and without splitting parameter set. The preprocessing was run for values $d = 0, \ldots, 4$ on all of our data sets. Figure 3 reports on the numbers.

The preprocessing is roughly ten times faster for reasonable values of the splitting parameter than without any splitting. Mainly this is because we do not store any additional information besides pointers to dictionary entries.

Query Performance. We conducted experiments on each list for maximum distances of $d = \{0, \ldots, 4\}$ to test the query performance for varying number of allowed errors. For natural language dictionaries a distance of $d = 3$ is already large and larger distances deliver matches that already look arbitrary. During each query we generated the candidate set, verified each member of the set and reported a best match found. Each test run picked 1 000 elements from the dictionary and introduced up to d errors at random. The splitting parameter is set

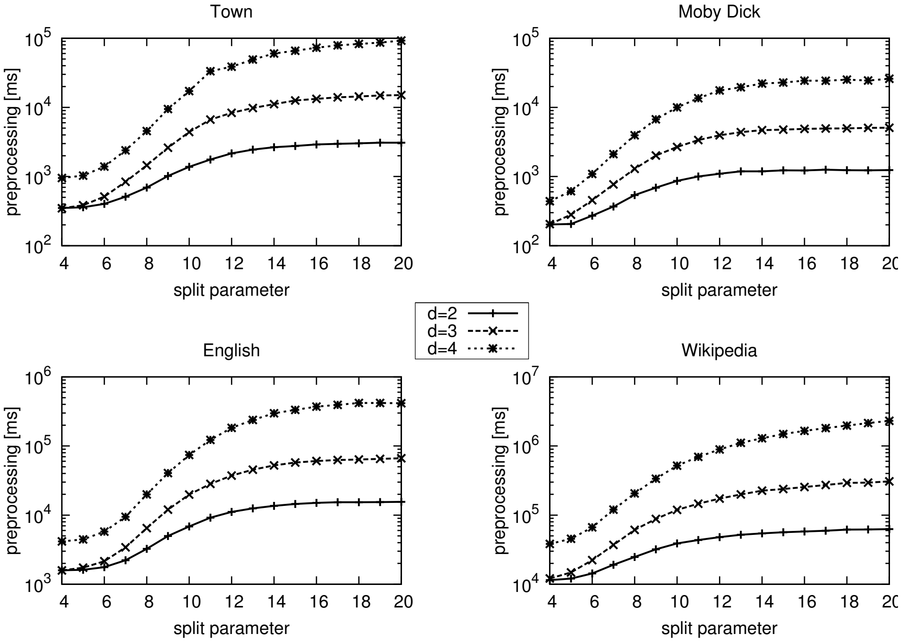

Fig. 3. Analysis of the preprocessing in relation to the splitting parameter.

to $m = 10$. The query times and search space sizes are averaged. Tables 2 and 3 report on these experiments.

The *query* column shows the time for the actual query in microseconds and *cand set* is the number of elements in the candidate set on average. We see the expected rise in the number of candidates that have to be verified by the algorithm. We briefly compared the observed number of collisions against the expected number from our analysis in Section 3. The observed number was always lower as the expected one since our analysis is an overestimate of the actual collision rate. In some cases we observed the order of a magnitude less collisions than expected.

When looking at our result and the original experiments of Bocek et al. [21] in Table 4 we see that our implementation performs better by about an order of magnitude in all important areas. Although we know that our numbers were measured on different hardware, they give an impression on the performance. The experiments were run on the same random dictionary of 10 000 words. Note that the case of $m = \infty$ corresponds to Bocek et al. 's algorithm. They proposed several improvements that either perform fast or have low space consumption but not both at the same time. Since the results of the experiments are only available as plots we have to estimate the values. We did so in a benevolent way and compare the best of their values in each category against our implementation with and without splitting. We see one potential source of performance problems with our experiments as we tested on dictionaries with rather short words that

	Mobydick		Town		English		Wikipedia	
d	mem	proc	mem	proc	mem	proc	mem	proc
0	0.25	0.061	0.46	0.156	2.36	0.886	14.41	7.131
1	1.33	0.320	1.79	0.576	8.55	3.450	55.84	32.287
2	4.57	1.272	6.91	2.483	30.49	12.596	170.79	107.289
3	9.78	4.044	15.18	7.458	61.37	36.309	342.18	270.506
4	16.09	14.647	27.20	28.144	105.75	117.970	603.35	922.521

Table 2. Preprocessing: *Mem* is the size of the index in [MiB], *proc* the duration [s].

	Mobydick		Town		English		Wikipedia	
	query	cand	query	cand	query	cand	query	cand
d	[μs]	set	[μs]	set	[μs]	set	[μs]	set
0	2	1	0	2	0	1	1	1
1	5	5	8	9	8	6	34	25
2	84	61	99	99	122	46	502	702
3	553	606	644	613	644	502	7 019	9 900
4	2 974	3 376	7 250	3 720	7 250	4 520	$55{\cdot}10^3$	$65{\cdot}10^3$

Table 3. Query: *query* is the average time for a single query in microseconds, *cand set* the average cardinality of the candidate set.

	$m = \infty$	$m = 10$	Best of Bocek et al.	BK-tree
preprocessing [ms]	2 649	**349**	5 000 - 7500	**183**
avg. query [μs]	114	**18**	100–$200{\cdot}10^3$	935
dictionary size [MiB]	9.8	**1.5**	20	**0.25**

Table 4. Comparison Against Existing Experiments, best results bold and BK-tree for reference.

have similar sizes. The higher the allowed error distance d is, the shorter residual strings get. This leads to longer indices lists in the hash table, because it is more likely that two distinct words will have common residual strings. This also explains the larger number of candidates for higher values of d.

An experimental evaluation of BK-trees [25] and several variants reports on the size of the search space that is visited depending on the allowed error distance. Those experiments were conducted on a set of 100 000 English words and report on a nearly linear growth of the visited search space going up from 5% for edit distance 0 to slightly more than 40% for a distance of 4. The size of the visited search space in our experiments is always less than 1% and much less than the search space size for the best BK-tree variant [25]. We were able to confirm the high number of candidates with our own BK-tree implementation. Table 5 reports on selected numbers of those experiments for the largest and smallest of the dictionaries.

	Mobydick		Wikipedia	
	query	cand	query	cand
d	[μs]	set	[μs]	set
1	198	197	1 258	1 184
2	3 586	4 127	$94{\cdot}10^3$	$116{\cdot}10^3$
3	8 722	$10{\cdot}10^3$	$374{\cdot}10^3$	$486{\cdot}10^3$
4	13 083	$15{\cdot}10^3$	$862{\cdot}10^3$	$802{\cdot}10^3$

Table 5. Selected numbers on the performance of BK-trees.

The number of candidates in BK-trees is high even for small allowed error distances. Thus the filtering effect of the metric space is quite low.

5 Conclusions and future work

We improved a method for approximate string matching in a dictionary. We developed algorithmic optimizations that provide a tuning parameter to choose between space consumption and running time while having overall lower preprocessing duration. Additionally, the performance has been validated experimentally by comparison against BK-trees and the baseline version of FastSS.

We see possibilities to speed up the verification of the candidate set using bit-parallelism [26] and SIMD instructions of current processors. This technique has been successfully used by [27]. However, only about half of the time of the algorithm is actually spent in the verification phase with the computation of the edit distance. Likewise there might be opportunities to speed up the precomputation, in particular, using fast, incremental computations of hash functions and using parallelization. On the other hand, it might be interesting to use data compression techniques to further reduce the storage requirements.

References

1. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady **10** (1966) 707–710
2. Baeza-Yates, R., Navarro, G.: Fast approximate string matching in a dictionary. In: SPIRE. (1998)
3. Burkhard, W.A., Keller, R.M.: Some approaches to best-match file searching. Commun. ACM **16** (1973) 230–236
4. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. **33** (2001) 273–321
5. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: 36th ACM Symposium on Theory of Computing. (2004)
6. Mihov, S., Schulz, K.U.: Fast approximate search in large dictionaries. Comput. Linguist. **30** (2004) 451–477
7. Russo, L.M.S., Navarro, G., Oliveira, A.L., Morales, P.: Approximate string matching with compressed indexes. Algorithms 2 **3** (2009) 1105–1136

8. Chan, H.L., Lam, T.W., Sung, W.K., Tam, S.L., Wong, S.S.: Compressed indexes for approximate string matching. Algorithmica (2008)
9. Kärkkäinen, J., Na, J.C.: Faster filters for approximate string matching. In: ALENEX, SIAM (2007)
10. Ukkonen, E.: Approximate string-matching with q-grams and maximal matches. Theor. Comput. Sci. **92** (1992) 191–211
11. Ukkonen, E.: Approximate string matching over suffix trees. In: CPM 1993. Volume 684 of LNCS., Springer-Verlag (1993) 228–242
12. Cobbs, A.L.: Fast approximate matching using suffix trees. In: Proceedings of the 6th Annual Combinatorial Pattern Matching Symposium (CPM'95). (1995)
13. Burkhardt, S., Kärkkäinen, J.: One-gapped q-gram filters for levenshtein distance. In: CPM. Volume 2373 of LNCS., Springer (2002)
14. Kärkkäinen, J.: Computing the threshold for q-gram filters. In: Proceedings of the 8th Scandinavian Workshop on Algorithm Theory, Springer (2002)
15. Burkhardt, S., Kärkkäinen, J.: Better filtering with gapped q-grams. In: Fundamenta Informaticae. (2001)
16. Maaß, M.G., Nowak, J.: Text indexing with errors. Journal of Discrete Algorithms **5** (2007) Selected papers from CPM 2005.
17. Maaß, M.G., Nowak, J.: A new method for approximate indexing and dictionary lookup with one error. Inf. Process. Lett. **96** (2005) 185–191
18. Gollapudi, S., Panigrahy, R.: A dictionary for approximate string search and longest prefix search. In: CIKM, ACM (2006)
19. Wu, S., Manber, U.: Agrep – a fast approximate pattern-matching tool. In: Proceedings USENIX Winter 1992 Technical Conference. (1992)
20. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. J. ACM **46** (1999) 395–415
21. Bocek, T., Hunt, E., Stiller, B.: Fast similarity search in large dictionaries. Technical report, Universität Zürich (2007) http://fastss.csg.uzh.ch/.
22. Mor, M., Fraenkel, A.S.: A hash code method for detecting and correcting spelling errors. Commun. ACM **25** (1982) 935–938
23. Wu, S., Manber, U.: Fast text searching: allowing errors. Commun. ACM **35** (1992) 83–91
24. Navarro, G., Baeza-Yates, R.: Improving an algorithm for approximate pattern matching. Algorithmica **30** (1998) 473–502
25. Motwani, G., Nair, S.G.: Search efficiency in indexing structures for similarity searching. CoRR **cs.DB/0403014** (2004)
26. Hyyrö, H., Fredriksson, K., Navarro, G.: Increased bit-parallelism for approximate string matching. ACM Journal of Experimental Algorithmics **10** (2005)
27. Fredriksson, K.: Engineering efficient metric indexes. Pattern Recogn. Lett. **28** (2007) 75–84