Computing and Informatics, Vol. 36, 2017, 1088–1106, doi: 10.4149/cai_2017_5_1088

A PRACTICAL INDEX FOR APPROXIMATE DICTIONARY MATCHING WITH FEW MISMATCHES

Aleksander Cisłak

Warsaw University of Technology
Faculty of Mathematics and Information Science
ul. Koszykowa 75, 00–662 Warsaw, Poland
e-mail: a.cislak@mini.pw.edu.pl

Szymon Grabowski

Lodz University of Technology
Institute of Applied Computer Science
Al. Politechniki 11, 90–924 Łódź, Poland
e-mail: sgrabow@kis.p.lodz.pl

Abstract. Approximate dictionary matching (checking if a pattern occurs in a collection of strings) is a classic problem with applications in e.g. spellchecking, online catalogs, and web searchers. We present a simple solution called *split index*, which is based on the Dirichlet principle, for matching a keyword with few mismatches, and experimentally show that it offers competitive space-time tradeoffs. Our implementation in the C++ language is focused mostly on data compaction, which is beneficial for the search speed. We compare our solution with other algorithms and we show that it is faster when the Hamming distance is used. Query times in the order of 1 microsecond were reported for one mismatch for a few-megabyte natural language dictionary on a medium-end PC. We also demonstrate that a basic compression technique consisting in q-gram substitution can significantly reduce the index size (up to 50 % of the input text size for the DNA sequences).

Keywords: Approximate dictionary matching, Hamming distance, k-mismatches problem, split index, text indexing

Mathematics Subject Classification 2010: 68W32

1 INTRODUCTION

Dictionary string matching (keyword matching, matching in dictionaries), defined as the task of checking if a query string occurs in a collection of strings given beforehand, is a classic research topic. In recent years, increased interest in *approximate* dictionary matching can be observed, where the query and one of the strings from the dictionary may only be similar in a specified sense rather than equal. Approximate dictionary matching is considered a hard problem, since most useful string similarity measures are non-transitive. Two popular measures include the Hamming distance (later referred to as Ham), which defines the number of mismatching characters between two strings of equal length, and the Levenshtein distance (later referred to as Lev), which defines the minimum number of edits (insertions, deletions, and substitutions) required for transforming one string into another. It is worth noting that matching with mismatches (i.e. using the Hamming distance), which is the main focus of this paper, is a very desired functionality with applications in, i.a., bioinformatics [22, 23], biometrics [13], cheminformatics [16], circuit design [20], and web crawling [26].

As indexes supporting approximate matching tends to grow exponentially in k, the maximum number of allowed errors, it is a worthwhile goal to design efficient indexes for small k values. In this paper, we focus on the problem of dictionary matching with few mismatches (especially one mismatch). Formally, for a collection $\mathcal{D} = \{s_1, \ldots, s_{|\mathcal{D}|}\}$ of $|\mathcal{D}|$ strings (also called words, we use these terms interchangeably) s_i of total length n (where n is the length of the concatenation of all words from the dictionary, i.e. $n = \sum_{i=1}^{|\mathcal{D}|} |s_i|$) over a given alphabet Σ (where $\sigma = |\Sigma|$), $I(\mathcal{D})$ is an approximate dictionary index supporting matching with mismatches, if for any query pattern P of length m ($m = |P|$) it returns all strings s_i from $\mathcal{D}$ such that $Ham(P, s_i) \leqslant k$. The number of occurrences, that is, the number of such matching strings is indicated by *occ*. Throughout this work the substrings are denoted as $S[i_1, i_2]$ (an inclusive range, hence, if $i_1 > i_2$, $S[i_1, i_2]$ is an empty string), and all indexes are 1-based. We also introduce the function $subtr(s, i, l)$, which removes the substring $s[i, i + l - 1]$ from s, i.e. $subtr(s, i, l) = s[1, i - 1]s[i + l, |s|]$ (where $s_1 s_2$ is a concatenation of two strings s_1 and s_2).

2 RELATED WORK

The following section describes the related work algorithms, and the summary of the complexities of notable algorithms is provided in Table 1. If not stated otherwise, the space complexities are expressed in words (in the sense of machine words with size $\Theta(\log n)$).

Solutions for approximate dictionary matching can be basically divided into two classes: the worst-case space and query time oriented, and the heuristical ones. Notable results from the first class include the k-errata trie by Cole et al. [11], which is based on the suffix tree and the longest common prefix structure. It can

be used in various contexts, including full-text and keyword indexing, as well as wildcard matching. For the Hamming distance and dictionary matching, it uses $O\left(n + |\mathcal{D}|\frac{(\log|\mathcal{D}|)^k}{k!}\right)$ space and offers $O\left(m + \frac{(\log|\mathcal{D}|)^k}{k!}\log\log n + occ\right)$ query time (this also holds for the edit distance but with larger constants). This was extended by Tsur [32] who described a structure similar to the one from Cole et al. with time complexity $O(m + \log\log n + occ)$ (for constant k) and $O\left(n^{1+\varepsilon}\right)$ space for a constant $\varepsilon > 0$. For full-text searching with the Hamming distance, Gabriele et al. [17] provided an index with average search time $O(m + occ)$ and $O\left(n\log^l n\right)$ space (for some l). Another theoretical work describing the algorithm which is similar to our split index was given by Shi and Widmayer [31], who obtained $O(n)$ preprocessing time and space complexity, and $O(n)$ expected search time (for a fixed alphabet) if k is bounded by $O(m/\log m)$. They introduced the notion of home strings for a given q-gram, which is the set of strings in $\mathcal{D}$ that contain the q-gram in the exact form (the value of q is set to $m/(k+1)$). In the search phase, they partition P into $k+1$ disjoint q-grams and use a candidate inspection order to speed up finding the matches with up to k edit distance errors.

On the practical front, Bocek et al. [3] provided a generalization of the Mor and Fraenkel [27] algorithm for $k \geqslant 1$ which is called *FastSS*. To check if two strings S_1 and S_2 match with up to k errors, we first delete all possible ordered subsets of k' symbols for all $0 \leqslant k' \leqslant k$ from S_1 and S_2. Then we conclude that S_1 and S_2 *may* be in edit distance of at most k if and only if the intersection of the resulting lists of strings is non-empty (explicit verification is still required). For instance, if $S_1 =$ `abbac` and $k = 2$, then its neighborhood is as follows: `abbac`, `bbac`, `abac`, `abac`, `abbc`, `abba`, `abb`, `aba`, `abc`, `aba`, `abc`, `aac`, `bba`, `bbc`, `bac` and `bac` (some of the resulting strings are repeated and they may be removed). If $S_2 =$ `baxcy`, then its respective neighborhood for $k = 2$ will contain, e.g., the string `bac`, but the following verification will show that S_1 and S_2 are in edit distance greater than 2. If, however, $Lev(S_1, S_2) \leqslant 2$, then it is impossible not to have in the neighborhood of S_2 at least one string from the neighborhood of S_1, hence we will never miss a match. The lookup requires $O\left(k(|s|_a)^k \log(n(|s|_a)^k)\right)$ time (where $|s|_a$ is the average dictionary word length), and the index occupies $O\left(n(|s|_a)^k\right)$ space. Another practical filter was presented by Karch et al. [21] and it improved on the FastSS method. They reduced space requirements and the query time by splitting long words (similarly to FastBlockSS [3] which is a variant of the original method) and storing the neighborhood implicitly with indexes and pointers to original dictionary entries. They claimed to be faster than other approaches such as the aforementioned FastSS and a BK-tree [6]. Recently, Chegrane and Belazzougui [9] described another practical index and they reported better results when compared to Karch et al. Their structure is based on the dictionary by Belazzougui for the edit distance of 1 (see the following subsection). An approximate (in the mathematical sense) data structure for approximate matching which is based on the Bloom filter was also described [25].

A permuterm index is a keyword index which supports queries with one wildcard symbol [19]. The idea is store all rotations of a given word appended with the

terminating character, for instance for the word `text`, the index would consist of the following permuterm vocabulary: `text$, ext$t, xt$te, t$tex, $text`. When it comes to searching, the query is first rotated so that the wildcard appears at the end, and subsequently its prefix is searched for using the index. This could be for example a trie or any other data structure which handles a prefix lookup. The main problem with the standard permuterm index is its space usage, as the number of strings inserted into the data structure is the number of words multiplied by the average string length. Ferragina and Venturini [15] proposed a compressed permuterm index in order to overcome the limitations of the original structure with respect to space. They explored the relation between the permuterm index and the Burrows–Wheeler Transform [7], which is applied to a concatenation of all strings from the input dictionary. Moreover, they provided a modification of the last-to-first (LF) mapping mechanism known from the FM-index (a full-text index by Ferragina and Venturini [14], consult the original article for more information) in order to support the functionality of the permuterm index.

2.1 The 1-Error Problem

It is important to consider methods for detecting a single error, since over 80 % of errors (even up to roughly 95 %) are within $k = 1$ for the edit distance with transpositions [12, 30]. Belazzougui and Venturini [2] presented a compressed index whose space is bounded in terms of the k^{th} order entropy of the indexed dictionary (see Section 3.1 for the description of this kind of entropy metric). It can be based either on perfect hashing, having $O(m + occ)$ query time and $2nH_k + n \cdot o(\log \sigma) + 2|\mathcal{D}| \log |\mathcal{D}|$ space requirements in bits (where H_k denotes the k^{th} order entropy), or on a compressed permuterm index with $O(m \min(m, \log_\sigma n \log \log n) + occ)$ time (when alphabet size $\sigma = \log^c n$ for some constant c) but improved space requirements. The former is a compressed variant of a dictionary presented by Belazzougui [1] which is based on neighborhood generation and occupies $O(n \log \sigma)$ bits of space and can answer queries in $O(m + occ)$ time. Chung et al. [10] showed a theoretical work where external memory is used, and their focus is on I/O operations. They limited the number of these operations to $O(1 + m/(wB) + occ/B)$, where w is the size of the machine word and B is the number of words within a block (a basic unit of I/O), with the space of the proposed structure of $O(n/B)$ blocks. In the category of filters, Mor and Fraenkel [27] described a method which is based on the deletion-only 1-neighborhood.

2.2 The 1-Mismatch Problem

For the 1-mismatch problem (Hamming distance), Yao and Yao [33] described a data structure for binary strings having length m with $O(m \log \log |\mathcal{D}|)$ query time and $O(m \log m|\mathcal{D}|)$ space requirements. This was later improved by Brodal and Gąsieniec [4] with the results of $O(m)$ query time with an index which occupies $O(n)$ space (also for binary strings). This was in turn extended with a structure

with $O(1)$ query time and $O(|\mathcal{D}| \log m)$ space in a cell probe model (where only the number of memory accesses is taken into account) [5]. Another notable example is a recent theoretical work of Chan and Lewenstein [8], who introduced an index with the optimal query time, i.e. $O(m/w + occ)$ which uses additional $O\left(w|\mathcal{D}| \log^{1+\varepsilon} |\mathcal{D}|\right)$ bits of space (beyond the dictionary itself), assuming a constant-size alphabet.

Name	Time Complexity	Space Complexity
Cole et al. [11]	$O\left(m + \frac{(\log \|\mathcal{D}\|)^k}{k!} \log\log n + occ\right)$	$O\left(n + \|\mathcal{D}\| \frac{(\log \|\mathcal{D}\|)^k}{k!}\right)$
Tsur [32]	$O(m + \log\log n + occ)$	$O\left(n^{1+\varepsilon}\right), \varepsilon > 0$
Gabriele et al. [17]	$O(m + occ)$	$O\left(n \log^l n\right)$
Shi and Widmayer [31]	$O(n)$ for $k = O(m/\log m)$ and $\sigma = O(1)$	$O(n)$
Bocek et al. [3]	$O\left(k(\|s\|_a)^k \log\left(n(\|s\|_a)^k\right)\right)$	$O\left(n(\|s\|_a)^k\right)$
1-Error Problem		
Belazzougui and Venturini [2]	$O(m + occ)$	$2nH_k + n \cdot o(\log \sigma)$ $+ 2\|\mathcal{D}\| \log \|\mathcal{D}\|$ bits
Belazzougui [1]	$O(m + occ)$	$O(n \log \sigma)$ bits
Chung et al. [10]	$O(1 + m/(wB) + occ/B)$	$O(n/B)$
1-Mismatch Problem		
Yao and Yao [33]	$O(m \log\log \|\mathcal{D}\|)$	$O(m \log m \|\mathcal{D}\|)$
Brodal and Gąsieniec [4]	$O(m)$	$O(n)$
Chan and Lewenstein [8]	$O(m/w + occ)$	$O\left(w\|\mathcal{D}\| \log^{1+\varepsilon} \|\mathcal{D}\|\right)$ bits

Table 1. Overview of the complexities of notable algorithms for matching in dictionaries. Symbol descriptions and more information on the cited results are located in Sections 1 and 2.

3 OUR ALGORITHM

The algorithm that we are going to present is uncomplicated and based on the Dirichlet principle, ubiquitous in approximate string matching techniques. We partition each string s into $k+1$ disjoint pieces $p_1, \ldots, p_{k+1}$, of average length $|s|/(k+1)$ (hence the name "split index"), and each such piece acts as a key in a hash table H_T. The precise size of each piece p_i of string s is determined using the following formula: $|p_i| = \lfloor |s|/(k+1) \rfloor$ for $i < k+1$ and $|p_{k+1}| = |s| - \sum_{i=1}^{k} |p_i|$. This means that the pieces might be in fact unequal in length, e.g., 2 and 3 for $|s| = 5$ and $k = 1$. The values in H_T are the lists of words which have one of their pieces as the corresponding key. For instance for the word `table` and $k = 1$, we would have a relation `tab` $\rightarrow$ `le` on one list (i.e. `tab` would be the key and `le` would be the value) and `le` $\rightarrow$ `tab` on the other. In this way, every word occurs on exactly $k+1$ lists. This seemingly bloats the space usage, still, in the case of small k the occupied space is acceptable. Moreover, instead of storing full words on the respective lists, we only store their "missing" pieces.

In the case of $k = 1$, we first populate each list with the pieces without the prefix and then with the pieces without the suffix; additionally we store the position on the list (as a 16-bit index) where the latter part begins. In this way, we traverse only a half of a list on average during the search. We also support k larger than 1 – in this case, we ignore the piece order on a list, and we store $\lceil \log_2(k+1) \rceil$ bits with each piece that indicate which part of the word is the list key. Let us note that this approach would also work for $k = 1$, however, it turned out to be less efficient.

As regards the implementation, our focus was on data compactness. In the hash table, we store the buckets which contain word pieces as keys (e.g. `le`) and pointers to the lists which store the missing pieces of the word (e.g. `tab`, `ft`). These pointers are always located right next to the keys, which means that unless we are very unlucky, a specific pointer should already be present in the CPU cache during the bucket traversal. The memory layouts of these substructures are fully contiguous. Successive strings are represented by multiple characters with a prepended 8-bit counter which specifies the length, and the counter with the value 0 indicates the end of the bucket or the list.

Any hash function for strings can be used, and two important considerations are the speed and the number of collisions, since a high number of collisions results in longer buckets, which may in turn have a negative effect on the query time (see Section 4 for further discussion). Figure 1 illustrates the layout of the split index.

The preprocessing stage proceeds as follows (consult Figure 2 in order to see the pseudocode):

1. Duplicate words are removed from the dictionary $\mathcal{D}$.

   The following steps refer to each string s from $\mathcal{D}$:

2. The word s is split into $k + 1$ pieces.
3. For each piece p_i: if $p_i \notin H_T$, we create a new list L_{new} containing the missing pieces (later simply referred to as a missing piece; in the case of $k = 1$, this is always one contiguous piece) $\mathcal{P} = \{p_j : j \in [1, k+1] \wedge j \neq i\}$. This list is then added to the hash table (we append p_i and the pointer to L_{new} to the bucket). Otherwise, if $p_i \in H_T$, we add the missing pieces $\mathcal{P}$ to the already existing list L. The pieces are inserted at relevant positions while keeping the order of the list which is described in previous paragraphs and in Figure 1.

   As regards the search (consult Figure 3 in order to see the pseudocode):

1. The query pattern P is split into $k + 1$ pieces.
2. We search for each piece p_i from the pattern (the prefix and the suffix if $k = 1$): the corresponding list L is retrieved from the hash table or we continue if $p_i \notin H_T$. We traverse each piece l_j from L – here, we always consider only the relevant pieces, e.g., missing suffixes if p_i is a missing prefix (the details of the traversal

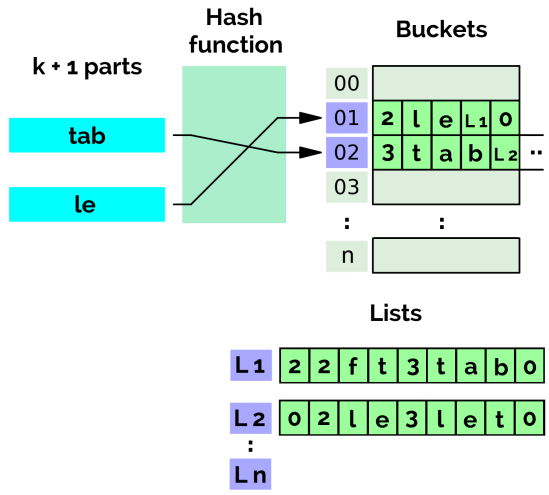

Figure 1. Split index for keyword indexing which shows the insertion of the word `table` for $k = 1$. The index also stores the words `left` and `tablet` (only selected lists containing pieces of these two words are shown), and `L1` and `L2` indicate pointers to the respective lists (L_1 and L_2). The first cell of each list indicates a 1-based word position (i.e. the word count from the left) where the missing prefixes begin ($k = 1$, hence we deal with two parts, namely prefixes and suffixes), and 0 means that the list has only missing suffixes. In this case, we have "2" in L_1 and `left` (missing suffix) and `table` (missing prefix), and "0" in L_2 and `table` (missing suffix) and `tablet` (missing suffix). Adapted from Wikimedia Commons (author: Jorge Stolfi; available at `http://en.wikipedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg`; CC A-SA 3.0).

depend on the layout, consult the preceding paragraphs). If $|l_j| = |P| - |p_i|$, the verification is performed and the result is returned if $Ham(l_j, P_S) \leqslant k$, where P_S is the pattern P with the piece p_i removed.

3. The missing pieces that were retrieved from the lists are combined into one word in order to present the answer.

3.1 Complexity

Let us consider the average string length $|s|_a$, where $|s|_a = \left(\sum_{i=1}^{|\mathcal{D}|} |s_i|\right) / |\mathcal{D}|$. Average time complexity of the preprocessing stage is equal to $O(kn)$, where k is the allowed number of errors, and n is the total input dictionary size (as defined in Section 1). This is because for each word and for each piece p_i we either create a new list for the missing pieces or we can add, if optimized, the pieces to the already existing list in $O(|s|_a)$ time (let us recall that $|\mathcal{D}||s|_a = n$). We assume that adding a new element to the bucket takes constant time on average, and the calculation of all hashes takes

```
Preprocessing(D, k)
1   (* assume that D is a dictionary of words without duplicate entries *)
2   H_T ← []
3   for each s ∈ D do
4       pSize = ⌊|s|/(k + 1)⌋
5       for i ∈ {1, ..., k} do
6           p_i ← s[1 + (i − 1) · pSize, i · pSize]
7           P_S = subtr(s, 1 + (i − 1) · pSize, pSize)
8           if p_i ∉ H_T
9               then H_T[p_i] ← [P_S]
10              else H_T[p_i].append(P_S)
11      (* handling the last piece of the pattern *)
12      p_{k+1} ← s[k · pSize + 1, |s|]
13      P_S = subtr(s, k · pSize + 1, |s| − (k · pSize))
14      if p_{k+1} ∉ H_T
15          then H_T[p_{k+1}] ← [P_S]
16          else H_T[p_{k+1}].append(P_S)
17  return H_T
```

Figure 2. Pseudocode description of the preprocessing procedure of the split index. The function *subtr* refers to substring removal (consult Section 1 for a more precise description of this operation).

$O(n)$ time in total. This is true irrespective of which list layout is used (there are two layouts for $k = 1$ and $k > 1$, see the preceding paragraphs). The occupied space is equal to $O(kn)$ because each piece is stored explicitly on exactly k lists and in exactly 1 bucket.

The average search complexity is $O(kt)$, where t is the average length of the list. We search for each of the $k + 1$ pieces of the pattern P of length $m = |P|$, and when the list corresponding to the piece p is found, it is traversed and at most t verifications are performed. Each verification takes at most $O(\min(m, |s_{max}|))$ time where s_{max} is the longest word in the dictionary[1], but $O(1)$ time on average. Again, we assume that determining the location of the specific list, that is iterating a bucket, takes $O(1)$ time on average. As regards the list, its average length t is higher when there is a higher probability that two words s_1 and s_2 from $\mathcal{D}$ have two pieces of the same length l which match exactly, i.e. $Pr(s_1[i_1, i_1 + l - 1] = s_2[i_2, i_2 + l - 1])$. Since all words are sampled from the same alphabet Σ, t depends on the alphabet size, that is $t = f(\sigma)$. Still, the dependence is rather indirect; in real-world dictionaries which store words from a given language, t will be rather dependent on the k^{th} order entropy of the language. This kind of entropy is similar to the classical Shannon's entropy, however, it takes the context

[1] Or $O(k)$ time, in theory, using the old longest common extension (LCE) based technique from Landau and Vishkin [24], after $O(n \log \sigma)$-time preprocessing.

```
Search(H_T, P, k)
1   pSize ← ⌊|P|/(k+1)⌋
2   for i ∈ {1,...,k} do
3       p_i ← P[1+(i−1)·pSize, i·pSize]
4   p_{k+1} ← P[k·pSize+1, |P|]
5   matchSet ← ∅
6   for i ∈ {1,...,k+1} do
7       if H_T[p_i] = NIL
8           then continue
9       L = H_T[p_i] (* L is a list containing the missing pieces *)
10      if i < k+1
11          then P_S = subtr(P, 1+(i−1)·pSize, pSize)
12          else P_S = subtr(P, k·pSize+1, |P|−(k·pSize))
13      for j ∈ {1,...,|L|} do
14          if |L_j| = |P| − |p_i|
15              then if Ham(L_j, P_S) ⩽ k
16                  then matchSet ← matchSet ∪ L_j
17  return matchSet
```

Figure 3. Pseudocode description of the search procedure of the split index. The function *subtr* refers to substring removal (consult Section 1 for a more precise description of this operation).

of k preceding symbols into account. Let us note that Shannon's entropy corresponds to the case of $k = 0$ (see e.g. Gog [18], Section 2.1.4 for more information).

3.2 Compression

In order to reduce storage requirements, we apply a basic compression technique. We find the most frequent q-grams (i.e. strings of q contiguous symbols; in the literature these are sometimes referred to as n-grams or k-mers) in the word collection and replace their occurrences on the lists with unused symbols, e.g., byte values 128, ..., 255. The value of q can be specified at the preprocessing stage, for instance $q = 2$ and $q = 4$ are reasonable for the English alphabet and the DNA, respectively. Different q values can be also combined depending on the distribution of q-grams in the input text, i.e. we may try all possible combinations of q-grams up to a certain q value and select ones which provide the best compression. In such a case, longer q-grams should be encoded before shorter ones. For example, the word `compression` could be encoded as `#p*s\` using the following substitution list: `com` → `#`, `re` → `*`, `co` → `$`, `om` → `&`, `sion` → `\` (note that not all q-grams from the substitution list are used). Possibly even a recursive approach could be applied, although this would certainly have a substantial impact on the query time.

The space usage could be further reduced by the use of a different character encoding. For the DNA (assuming 4 symbols only) it would be sufficient to use 2 bits per character, and for the basic English alphabet 5 bits. In the latter case there are 26 letters, which in a simplified text can be augmented only with a space character, a few punctuation marks, and a capital letter flag. Such an approach would be also beneficial for space compaction, and it could have a further positive impact on cache usage. The compression naturally reduces the space while increasing the search time, and a sort of a middle ground can be achieved by deciding which additional information to store in the index. This can be for instance the length of an encoded (compressed) piece after decoding, which could eliminate some pieces based on their size without performing the decompression and explicit verification.

4 EXPERIMENTAL RESULTS

Experimental results were obtained on the machine equipped with the Intel i5-3230M processor running at 2.6 GHz and 8 GB DDR3 memory, and the C++ source code was compiled (as a 32-bit version) with clang v. 3.4-1 and run on the Ubuntu 14.04 OS. The source code is publicly available under the following link: `https://github.com/MrAlexSee/SplitIndex`. The data sets that were used in order to obtain the experimental results are listed in Section 6. Each evaluation was run 100 times and the results were averaged as an arithmetic mean.

One of the crucial components of the split index is a hash function. Ideally, we would like to minimize the average length of the bucket (let us recall that we use chaining for collision resolution), however, the hash function should be also relatively fast because during the search it has to be calculated for each of the $k + 1$ pieces of the pattern. We investigated various hash functions, and it turned out that the differences in query times are not negligible, although the average length of the bucket was almost the same in all cases (relative differences were smaller than 1 %). We can see in Table 2 that the fastest function was the xxhash (available on the Internet under the following link: `https://code.google.com/p/xxhash/`), and for this reason it was used for the calculation of other results.

Decreasing the value of the load factor (LF) did not strictly provide a speedup in terms of the query time, as demonstrated in Figure 4. This can be explained by the fact that even though the relative reduction in the number of collisions was substantial, the absolute difference was equal to at most a few collisions per list. Moreover, when the LF was higher, pointers to the lists could be possibly closer to each other, which might have had a positive effect on cache utilization. The best query time was reported for the maximum LF value of 2.0, hence this value was used for the calculation of other results.

In Table 3 we can see a linear increase in the index size and an exponential increase in query time with growing k. Even though we concentrate on $k = 1$ and the most promising results are reported for this case, our index might remain competitive also for higher k values.

Hash Function	Query Time (µs)
xxhash	0.93
sdbm	0.95
FNV1	0.95
FNV1a	0.95
SuperFast	0.96
Murmur3	0.97
City	0.99
FARSH	1.00
SpookyV2	1.04
Farm	1.04

Table 2. Evaluated hash functions and search times per query for the English dictionary of size 2.67 MB and $k = 1$. A list of common English misspellings was used as queries, maximum load factor = 2.0.

k	Query Time (µs)	Index Size (KB)
1	0.51 (0.05)	1 715
2	11.49 (0.53)	2 248
3	62.85 (0.83)	3 078

Table 3. Query time (with standard deviation reported in the parentheses) and index size vs. the error value k for the English language dictionary of size 0.79 MB. A list of common English misspellings was used as queries.

Q-gram substitution coding (the technique described in Section 3.2) provided an expected reduction in index size, at the cost of an increased query time. Q-grams were generated separately for each dictionary $\mathcal{D}$ as a list of 100 q-grams which provided the best compression for $\mathcal{D}$, i.e. they minimized the size of all encoded words, $S_E = \sum_{i=1}^{|\mathcal{D}|} |Enc(s_i)|$. For the English language dictionaries, we also considered using only 2-grams or only 3-grams, and for the DNA only 2-grams (a maximum of 25 2-grams) or 4-grams, since mixing the q-grams of various sizes has a further negative impact on the query time. For the DNA, 5 000 queries were generated randomly by introducing noise into words sampled from dictionary, and their length was equal to the length of the particular word. Up to 3 errors were inserted, each with a 50 % probability. For the English dictionaries we opted for the list of common misspellings, and the results were similar to the case of randomly generated queries.

We can see the speed-to-space relation for the English dictionaries in Figure 5 and for the DNA in Figure 6. In the case of English, using the optimal (from the compression point of view, i.e. minimizing the index size) combination of mixed q-grams provided almost the same index size as using only 2-grams. Substitution coding methods performed better for the DNA (where $\sigma = 5$) because the sequences are more repetitive. Let us note that the compression provided a higher relative decrease in index size with respect to the original text as the size of the dictionary

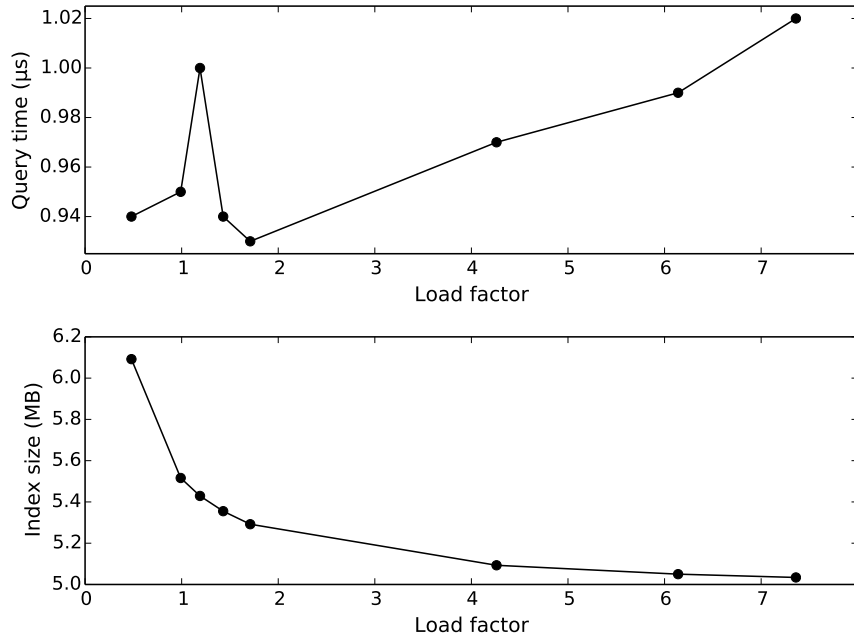

Figure 4. Query time and index size vs. the load factor for the English dictionary of size 2.67 MB and $k = 1$. A list of common English misspellings was used as queries. The value of LF can be higher than 1.0 because we use chaining for collision resolution.

increased. For instance, for the DNA dictionary of size 627.8 MB the compression ratio was equal to 1.93 and the query time was still around 100 µs.

Tested on the English language dictionaries, favorable results were reported when compared to methods proposed by other authors. Others consider the Levenshtein distance as the edit distance, whereas we use the Hamming distance, which puts us at the advantageous position. Still, the provided speedup is significant, and we believe that the more restrictive Hamming distance is also an important measure of practical use. The implementations of other authors are available on the Internet (`http://searchivarius.org/personal/software`; `https://code.google.com/p/compact-approximate-string-dictionary/`, from Boytsov and Chegrane and Belazzougui, respectively). As regards the results reported for the Mor–Fraenkel method and Boytsov's Reduced alphabet neighborhood generation, it was not possible to accurately calculate the size of the index (both implementations by Boytsov), and for this reason we used rough ratios based on index sizes reported by Boytsov for similar dictionary sizes. Let us note that we compare our algorithm with Chegrane and Belazzougui [9], who published better results when compared to Karch et al. [21], who, in turn, claimed to be faster than

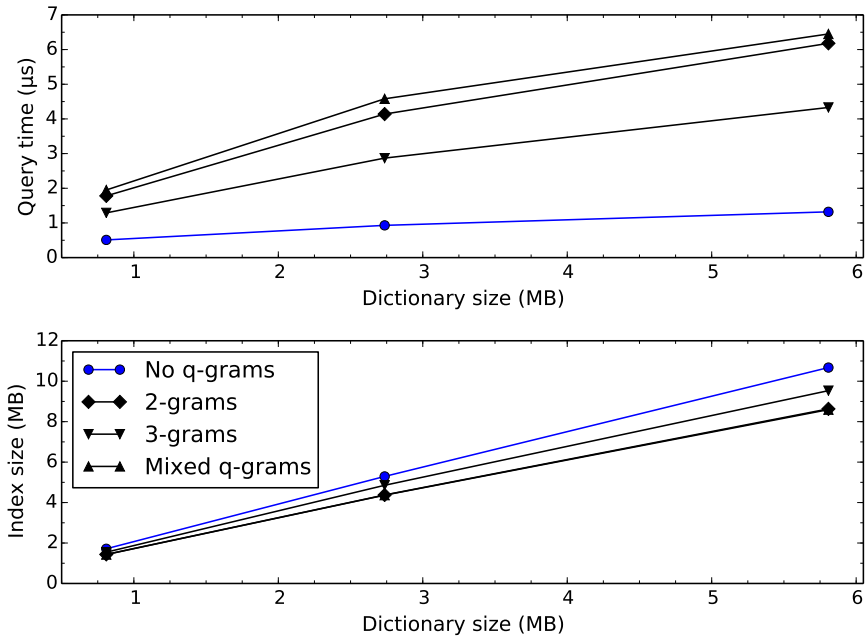

Figure 5. Query time and index size vs. dictionary size for $k = 1$, with and without q-gram coding. Mixed q-grams refer to the combination of q-grams which provided the best compression, and for the three dictionaries these were equal to ([2-, 3-, 4-] grams): $[88, 8, 4]$, $[96, 2, 2]$, and $[94, 4, 2]$, respectively. English dictionaries and the list of common English misspellings were used. Standard deviation values for all time measurements were roughly in the order of 5 % of the reported values.

other state-of-the-art methods. We have not managed to identify any practice-oriented indexes for matching in dictionaries over any fixed alphabet Σ dedicated for the Hamming distance, which could be directly compared to our split index. The times for the brute-force algorithm are not listed, since they were roughly 3 orders of magnitude higher than the ones presented. Consult Figure 7 for details.

We also evaluated different word splitting schemes, for instance for $k = 1$, one could split the word into two pieces of different sizes, e.g., $6 \to (2, 4)$ instead of $6 \to (3, 3)$. However, unbalanced splitting methods caused slower queries when compared to the regular one. As regards Hamming distance calculation, it turned out that a naive implementation (i.e. simply iterating and comparing each character) was the fastest one. The compiler with automatic optimization was simply more efficient than other implementations (e.g. ones based directly on SSE instructions) that we have investigated.

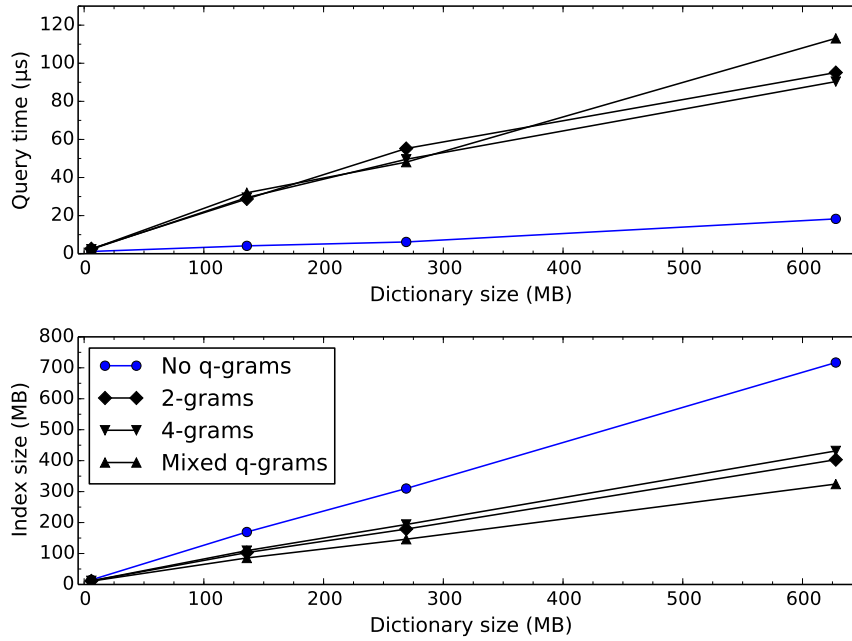

Figure 6. Query time and index size vs. dictionary size for $k = 1$, with and without q-gram coding. Mixed q-grams refer to the combination of q-grams which provided the best compression, and these were equal to ([2-, 3-, 4-] grams): [16, 66, 18] (due to computational constraints, they were calculated only for the first dictionary, but used for all four dictionaries). DNA dictionaries and the randomly generated queries were used. Standard deviation values for all time measurements were roughly in the order of 5 % of the reported values.

5 CONCLUSIONS

We have presented an index for dictionary matching with mismatches, which performed best for the Hamming distance of one. Its functionality could be extended by storing additional information in the lists that contain missing parts of the words. This could be for instance a mapping of words to positions in the document, which would create an inverted index handling approximate matching.

Another useful extension could consist in a dedicated solution for binary alphabets (where $\sigma = 2$). In this case, the characters would be stored as individual bits, which would greatly reduce the overall index size, while also making it possible for more consecutive characters to fit into a single cache line. On the other hand, access to each specific character would be slower (as it would require additional bitwise operations), hence, the effect of such a layout on the query time is rather unclear. This could be combined with the support for fixed-sized queries, since they

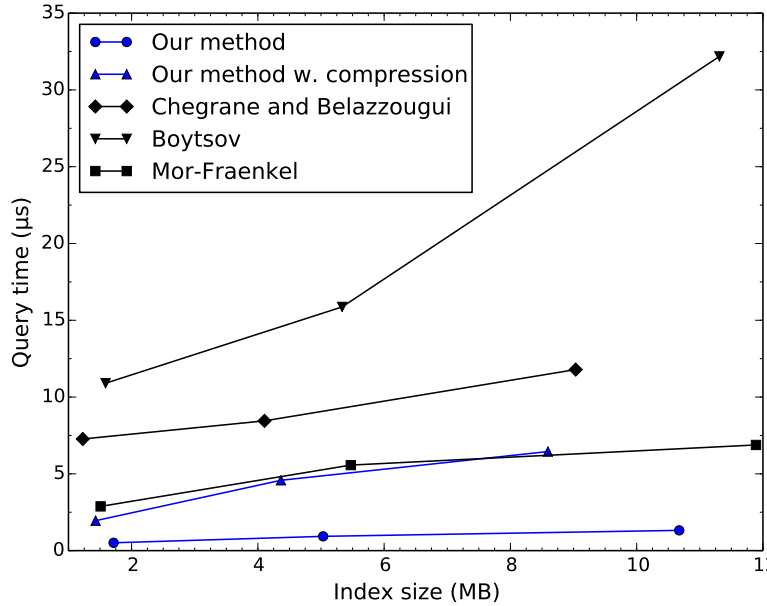

Figure 7. Query time vs. index size for different methods. The method with compression encoded mixed q-grams (consult Section 3.2). We used the Hamming distance, and the other authors used the Levenshtein distance for $k = 1$. English dictionaries of size 0.79 MB, 2.67 MB, and 5.8 MB were used as input, and the list of common English misspellings was used for queries.

are rather common in the case of binary alphabets (where they describe, e.g., a fixed number of states of the elements in a certain system [20]). If the size of all words in a dictionary were known, it would be possible to additionally reduce the space requirements of the index by omitting information regarding each piece length.

The algorithm can be sped up by means of parallelization, since access to the index during the search procedure is read-only. In the most straightforward approach we could simply distribute individual queries between multiple threads. A more fine-grained variation would be to concurrently operate on parts of the word after it has been split up (the number of parts depending on the k parameter), or we could even access in parallel lists which contain candidate prefixes and suffixes. If we had a sufficient amount of threads at our disposal, these approaches could be combined.

In the future, we are going to seek ways to employ the presented technique in full-text indexed approximate pattern matching. One relevant idea can be intermediate partitioning, that is, breaking either the pattern [28] or a q-gram from the text [29]

into few pieces, in order to match against them in an approximate (rather than exact in the traditional partitioning) manner.

Acknowledgement

The work was supported by the Polish National Science Centre under the project DEC-2013/09/B/ST6/03117 (the second author).

6 DATA SETS

The following data sets were used in order to obtain the experimental results:

- iamerican – 0.79 MB, English, available from Linux packages,
- foster – 2.67 MB, English, available at: `http://www.math.sjsu.edu/~foster/dictionary.txt`,
- iamerican-insane – 5.8 MB, English, available from Linux packages,
- DNA – 20-mers extracted from the genome of Drosophila melanogaster (available at: `http://flybase.org/`), sizes: 6.01 MB, 135.89 MB, 262.78 MB, and 627.80 MB,
- A list of common English misspellings – 44.2 KB (4 261 words), available at: `http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines`.

REFERENCES

[1] Belazzougui, D.: Faster and Space-Optimal Edit Distance "1" Dictionary. Combinatorial Pattern Matching (CPM 2009). Springer, Lecture Notes in Computer Science, Vol. 5577, 2009, pp. 154–167, doi: 10.1007/978-3-642-02441-2_14.

[2] Belazzougui, D.—Venturini, R.: Compressed String Dictionary Look-Up with Edit Distance One. Combinatorial Pattern Matching (CPM 2012). Springer, Lecture Notes in Computer Science, Vol. 7354, 2012, pp. 280–292, doi: 10.1007/978-3-642-31265-6_23.

[3] Bocek, T.—Hunt, E.—Stiller, B.—Hecht, F.: Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, Switzerland.

[4] Brodal, G. S.—Gąsieniec, L.: Approximate Dictionary Queries. Combinatorial Pattern Matching (CPM 1996). Springer, Lecture Notes in Computer Science, Vol. 1075, 1996, pp. 65–74, doi: 10.1007/3-540-61258-0_6.

[5] Brodal, G. S.—Venkatesh, S.: Improved Bounds for Dictionary Look-Up with One Error. Information Processing Letters, Vol. 75, 2000, No. 1, pp. 57–59, doi: 10.1016/S0020-0190(00)00079-X.

[6] Burkhard, W. A.—Keller, R. M.: Some Approaches to Best-Match File Searching. Communications of the ACM, Vol. 16, 1973, No. 4, pp. 230–236, doi: 10.1145/362003.362025.

[7] Burrows, M.—Wheeler, D. J.: A Block-Sorting Lossless Data Compression Algorithm. Technical Report 124, Digital, Systems Research Center, Palo Alto, California, 1994.

[8] Chan, T.—Lewenstein, M.: Fast String Dictionary Lookup with One Error. Combinatorial Pattern Matching (CPM 2015). Springer, Lecture Notes in Computer Science, Vol. 9133, 2015, pp. 114–123, doi: 10.1007/978-3-319-19929-0_10.

[9] Chegrane, I.—Belazzougui, D.: Simple, Compact and Robust Approximate String Dictionary. Journal of Discrete Algorithms, Vol. 28, 2014, pp. 49–60, doi: 10.1016/j.jda.2014.08.003.

[10] Chung, C. W.—Tao, Y.—Wang, W.: I/O-Efficient Dictionary Search with One Edit Error. String Processing and Information Retrieval (SPIRE 2014). Springer, Lecture Notes in Computer Science, Vol. 8799, 2014, pp. 191–202, doi: 10.1007/978-3-319-11918-2_19.

[11] Cole, R.—Gottlieb, L. A.—Lewenstein, M.: Dictionary Matching and Indexing with Errors and Don't Cares. Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing, ACM, 2004, pp. 91–100, doi: 10.1145/1007352.1007374.

[12] Damerau, F. J.: A Technique for Computer Detection and Correction of Spelling Errors. Communications of the ACM, Vol. 7, 1964, No. 3, pp. 171–176, doi: 10.1145/363958.363994.

[13] Davida, G. I.—Frankel, Y.—Matt, B. J.: On Enabling Secure Applications Through Off-Line Biometric Identification. Security and Privacy, IEEE, 1998, pp. 148–157.

[14] Ferragina, P.—Manzini, G.: Opportunistic Data Structures with Applications. 41st Annual Symposium on Foundations of Computer Science (FOCS 2000), November 12–14, 2000, Redondo Beach, California, USA, pp. 390–398, doi: 10.1109/SFCS.2000.892127.

[15] Ferragina, P.—Venturini, R.: The Compressed Permuterm Index. ACM Transactions on Algorithms, Vol. 7, 2010, No. 1, Art. No. 10, doi: 10.1145/1868237.1868248.

[16] Flower, D. R.: On the Properties of Bit String-Based Measures of Chemical Similarity. Journal of Chemical Information and Computer Sciences, Vol. 38, 1998, No. 3, pp. 379–386, doi: 10.1021/ci970437z.

[17] Gabriele, A.—Mignosi, F.—Restivo, A.—Sciortino, M.: Indexing Structures for Approximate String Matching. Algorithms and Complexity (CIAC 2003). Springer, Lecture Notes in Computer Science, Vol. 2653, 2003, pp. 140–151, doi: 10.1007/3-540-44849-7_20.

[18] Gog, S.: Compressed Suffix Trees: Design, Construction, and Applications. Ph.D. Thesis, University of Ulm, 2011.

[19] Garfield, E.: The Permuterm Subject Index: An Autobiographical Review. Journal of the American Society for Information Science, Vol. 27, 1976, No. 5, pp. 288–291, doi: 10.1002/asi.4630270504.

[20] Girard, P.—Landrault, C.—Pravossoudovitch, S.—Severac, D.: Reduction of Power Consumption During Test Application by Test Vector Ordering. Electronics Letters, Vol. 33, 1997, No. 21, pp. 1752–1754, doi: 10.1049/el:19971225.

[21] Karch, D.—Luxen, D.—Sanders, P.: Improved Fast Similarity Search in Dictionaries. String Processing and Information Retrieval (SPIRE 2010). Springer, Lecture Notes in Computer Science, Vol. 6393, 2010, pp. 173–178, doi: 10.1007/978-3-642-16321-0_16.

[22] Kurtz, S.—Choudhuri, J. V.—Ohlebusch, E.—Schleiermacher, C.—Stoye, J.—Giegerich, R.: REPuter: The Manifold Applications of Repeat Analysis on a Genomic Scale. Nucleic Acids Research, Vol. 29, 2001, No. 22, pp. 4633–4642, doi: 10.1093/nar/29.22.4633.

[23] Landau, G. M.—Schmidt, J. P.—Sokol, D.: An Algorithm for Approximate Tandem Repeats. Journal of Computational Biology, Vol. 8, 2001, No. 1, pp. 1–18, doi: 10.1089/106652701300099038.

[24] Landau, G. M.—Vishkin, U.: Fast Parallel and Serial Approximate String Matching. Journal of Algorithms, Vol. 10, 1989, No. 2, pp. 157–169, doi: 10.1016/0196-6774(89)90010-2.

[25] Manber, U.—Wu, S.: An Algorithm for Approximate Membership Checking with Application to Password Security. Information Processing Letters, Vol. 50, 1994, No. 4, pp. 191–197, doi: 10.1016/0020-0190(94)00032-8.

[26] Manku, G. S.—Jain, A.—Das Sarma, A.: Detecting Near-Duplicates for Web Crawling. Proceedings of the 16th International Conference on World Wide Web, ACM, 2007, pp. 141–150, doi: 10.1145/1242572.1242592.

[27] Mor, M.—Fraenkel, A. S.: A Hash Code Method for Detecting and Correcting Spelling Errors. Communications of the ACM, Vol. 25, 1982, No. 12, pp. 935–938.

[28] Navarro, G.—Baeza-Yates, R.: A Hybrid Indexing Method for Approximate String Matching. Journal of Discrete Algorithms, Vol. 1, 2000, No. 1, pp. 205–239, Special Issue on Matching Patterns.

[29] Navarro, G.—Sutinen, E.—Tarhio, J.: Indexing Text with Approximate q-Grams. Journal of Discrete Algorithms, Vol. 3, 2005, No. 2, pp. 157–175, doi: 10.1016/j.jda.2004.08.003.

[30] Pollock, J. J.—Zamora, A.: Automatic Spelling Correction in Scientific and Scholarly Text. Communications of the ACM, Vol. 27, 1984, No. 4, pp. 358–368, doi: 10.1145/358027.358048.

[31] Shi, F.—Widmayer, P.: Approximate Multiple String Searching by Clustering. Genome Informatics, Vol. 7, 1996, pp. 33–40.

[32] Tsur, D.: Fast Index for Approximate String Matching. Journal of Discrete Algorithms, Vol. 8, 2010, No. 4, pp. 339–345.

[33] Yao, A. C.—Yao, F. F.: Dictionary Look-Up with Small Errors. Combinatorial Pattern Matching (CPM 1995). Springer, Lecture Notes in Computer Science, Vol. 937, 1995, pp. 387–394, doi: 10.1007/3-540-60044-2_57.

Aleksander Cisłak received his B.Sc. degree in computer science from Lodz University of Technology in 2014 and M.Sc. degree in informatics from TU München in 2015. His research interests include string matching algorithms and applied graph theory. He worked as Assistant at TU München, conducting research in the area of graph-based behavioral malware detection, and he currently works as Assistant at Warsaw University of Technology, with the main focus of his work being the fuzzy cognitive maps (FCMs).

Szymon Grabowski received his M.Sc. degree from the University of Lodz in 1996, Ph.D. degree from AGH-UST in Cracow in 2003, and Habilitation degree from Systems Research Institute in Warsaw in 2011, all in computer science. His former research, including his Ph.D. dissertation, involved nearest neighbor classification methods in pattern recognition, also with applications in image processing. Currently, his main interests are focused on string matching and text indexing algorithms, and data compression. Some of his particular research topics include various approximate string matching problems, compressed text indexes, and XML compression. He has published about 100 papers in journals and conferences. He is currently Professor at the Institute of Applied Computer Science of Lodz University of Technology.