Computing and Informatics, Vol. 38, 2019, 367–389, doi: 10.31577/cai_2019_2_367

LIGHTWEIGHT FINGERPRINTS FOR FAST APPROXIMATE KEYWORD MATCHING USING BITWISE OPERATIONS

Aleksander Cisłak, Szymon Grabowski

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

Abstract. We aim to speed up approximate keyword matching with the use of a lightweight, fixed-size block of data for each string, called a fingerprint. These work in a similar way to hash values; however, they can be also used for matching with errors. They store information regarding symbol occurrences using individual bits, and they can be compared against each other with a constant number of bitwise operations. In this way, certain strings can be deduced to be at least within the distance k from each other (using Hamming or Levenshtein distance) without performing an explicit verification. We show experimentally that for a preprocessed collection of strings, fingerprints can provide substantial speedups for $k = 1$, namely over 2.5 times for the Hamming distance and over 30 times for the Levenshtein distance. Tests were conducted on synthetic and real-world English and URL data.

Keywords: Fingerprint, keyword matching, approximate matching, bitwise

Mathematics Subject Classification 2010: 68W32

1 INTRODUCTION

This study deals with strings, that is, finite sequences of symbols. We assume that a string S is 1-indexed, i.e., index 1 refers to the first symbol $S[1]$, index 2 refers to the second symbol $S[2]$, etc. All strings are specified over the same alphabet Σ, with alphabet size $\sigma = |\Sigma|$.

Exact string comparison refers to checking whether two strings S_1 and S_2 of equal length n have the same characters at all corresponding positions. The strings can store, e.g., natural language data or DNA sequences. Assuming that each character occupies 1 byte, calculation of such a comparison takes $O(n)$ time in the worst case; however, the average case is $O(1)$, using no additional memory. Specifically, the complexity of the average case of comparing two strings depends on the alphabet size. Assuming a uniform random symbol distribution, the chance that first two symbols match (i.e., that $S_1[1] = S_2[1]$) is equal to $1/\sigma$, the chance that both first and second symbol pairs match (i.e., that $S_1[1] = S_2[1]$ and $S_1[2] = S_2[2]$) is equal to $1/\sigma^2$, etc. More generally, the probability that there is a match between all characters up to a 1-indexed position i is equal to $1/\sigma^i$.

Nonetheless, it is often faster to compare hash values for two strings (in constant time) and perform an explicit verification only when these hashes are equal to each other. This is particularly true in a situation where one would compare a single string, that is a query (pattern), against a preprocessed collection (dictionary) of strings. The hash-based approach forms the basis of, e.g., the well-known Rabin–Karp [16] algorithm for online exact matching.

Aside from exact matching, there has been a substantial interest in *approximate* string comparison, for instance for spelling suggestions or matching biological data [22, 26, 21]. Approximate string matching defines whether two strings are equal according to a specified similarity metric, and the number of errors is denoted by k in the following text. Two popular measures include:

- the Hamming distance [15] (later referred to as Ham), which defines the number of mismatching characters at corresponding positions between two strings of equal length,
- the Levenshtein distance [17] (also called edit distance, later referred to as Lev), which determines the minimum number of edits (insertions, deletions, and substitutions) required for transforming one string into another.

Hash values cannot be easily used in the approximate context. This work has focused on approximate matching in practice, and we introduce the concept of lightweight *fingerprints*, whose goal is to speed up approximate string comparison. The speedup can be achieved for preprocessed collections of strings, at the cost of a fixed-sized amount of space per each word in the collection. This means that we evaluate fingerprints for a keyword indexing problem, also known as *dictionary matching* or *keyword matching*; see, e.g., [3, 5, 6, 8, 9, 10]. Specifically, in this setting a pattern P is compared against a string collection $\mathcal{D} = \{S_1, \ldots, S_{|\mathcal{D}|}\}$. In the following, the text size is generally denoted by n, and the pattern size is denoted by m (i.e., $|P| = m$).

2 RELATED WORK

The original idea of a string fingerprint, which is also called a "sketch" in selected publications [1], goes back to the work of Rabin and Karp [28, 16]. They used a variant of a hash function called a rolling hash, which can be quickly (incrementally) calculated for each successive substring of the input text, in order to speed up exact online substring matching. This technique was later used also in the context of multiple pattern matching [20, 30] and matching over a two-dimensional text [33]. Bille et al. [4] extended this idea and demonstrated how to construct fingerprints for substrings of a string which is compressed by a context-free grammar. Policriti et al. [25] generalized the classical Rabin–Karp algorithm in order to be used with the Hamming distance.

At the conceptual level, fingerprints may be perceived as a form of lossy compression over the input text, nevertheless, they cannot replace the text – rather, they can be used as additional information. Bar-Yossef et al. [1] show that it is not possible to use only a fingerprint (reducing the text by more than a constant factor) in order to answer a match query. Moreover, they prove that for answering decision queries under the Hamming distance – such that the existence of the pattern in the text with less than k Hamming errors is reported as "a match" and no such occurrence yields the "no match" output – the size of the fingerprint must be $\Omega(n/m)$, where $k = \varepsilon m$, for a fixed $0 < \varepsilon < 1$.

Policriti and Prezza [24] presented a related idea called de Bruijn hash function, where shifting the substring by one character results in a corresponding one-character shift in its hash value. Grabowski and Raniszewski [13] used fingerprints in order to speed up verifying tentative matches in their SamSAMi (sampled suffix array with minimizers) full-text index. Fingerprints, which are concatenations of selected bits taken from a short string, allow them to reject most candidate matches without accessing the indexed text and thus avoiding many cache misses. Recently, fingerprints have been applied to the longest common extension (LCE) problem [27], allowing to solve the LCE queries in logarithmic time in essentially the same space as the input text (replacing the text with a data structure of the same size).

Ramaswamy et al. [29] described a technique called "approximate" fingerprinting; however, it refers to exact pattern matching with false positives rather than matching based on similarity metrics. Fingerprints have also been used for matching at a larger scale, i.a., for determining similarity between audio recordings [7] and files [19]. The term fingerprint has also been used with a different meaning in the domain of string processing, where it refers to the set of distinct characters contained in one of the substrings of a given string, with the ongoing recent work, e.g., a study by Belazzougui et al. [2].

3 FINGERPRINTS

In this section we introduce the notion of a fingerprint, describe its construction and demonstrate how to compare two fingerprints. For a given string S, a fingerprint S'

is constructed as $S' := f(S)$ using a function f which returns a fixed-sized block of data. In particular, for two strings S_1 and S_2, we would like to determine that $Ham(S_1, S_2) > k$ or $Lev(S_1, S_2) > k$ by comparing only fingerprints S'_1 and S'_2 for a given $k \in \mathbb{N}^+$. In other words, fingerprints allow for a quick rejection of a candidate for an approximate match (up to k errors) between two strings.

Fingerprint comparison might be indecisive, i.e., it might not be sufficient to indicate that the above stated inequalities hold. In that case (we explain later when this occurs), we still have to perform an explicit verification on S_1 and S_2, but fingerprints allow for reducing the overall number of such operations. There exists a similarity between fingerprints and hash functions; nonetheless, hash comparison works only in the context of exact matching. Let us clarify that in this work the term fingerprint refers to a short (having at most a few bytes in length) block of data which can be used for the aforementioned approximate matching.

As far as the complexity of a single verification (string comparison) is concerned, the worst case is equal to $O(n)$ for the Hamming distance (considering two strings of equal size n) and $O(k \min(|S_1|, |S_2|))$ for the Levenshtein distance, using Ukkonen's algorithm [32]. Assuming a uniform random alphabet distribution, the average case complexity is equal to $O(k)$ for both metrics.

In our proposal, fingerprints use individual bits in order to store information about symbol frequencies or positions in the string $S[1, n]$. Let $\Sigma' \subseteq \Sigma$ be a subset of the original alphabet with $\sigma' = |\Sigma'|$ denoting its size. We propose the following approaches.

- **Occurrence** (**occ** in short): we store information in each bit that indicates whether a certain symbol from Σ'_{occ} occurs in a string using σ'_{occ} bits in total.
- **Occurrence halved**: the fingerprint refers to occurrences in the first and second halves of S, that is, $S[1, \lfloor n/2 \rfloor]$ and $S[\lfloor n/2 \rfloor + 1, n]$, respectively. We store information whether each of the σ'_{occh} symbols occurs in the first half of S using the first σ'_{occh} bits of the fingerprint, and we store information whether each of the same σ'_{occh} symbols occurs in the second half of S using the second σ'_{occh} bits of the fingerprint. The occurrence halved scheme works only for the Hamming distance.
- **Count**: we store a count (i.e., the number of occurrences) of each symbol using b bits per symbol. The count can be in the range $[0, 2^b - 1]$, where $2^b - 1$ indicates that there are $2^b - 1$ or more occurrences of a given symbol. We use σ'_{count} symbols from Σ'_{count}.
- **Position** (**pos** in short): we can encode information regarding the first (leftmost, i.e., the one with the lowest index) position in S of each symbol from Σ'_{pos} using p bits per symbol, where $p \leq \lceil \log_2 n \rceil$. This position can be in the range $[1, 2^p - 1]$ encoded in the fingerprint as 0-indexed, where index 0 refers to the first symbol, index 1 refers to the second symbol, etc, and the value of $2^p - 1$ indicates that the first occurrence is either at one of the positions from the range $[2^p, n]$ or the symbol does not occur in S (we do not know which one is true). We use

$\sigma'_{pos} \cdot p$ bits in order to encode positions of σ'_{pos} symbols. The remaining bits, e.g., 1 bit for $\sigma'_{pos} = 5$, $p = 3$ and 16 bits per fingerprint, are used in order to store information about the occurrences of additional symbols, in the same fashion as in the occurrence fingerprint which was introduced previously. The position-based scheme works only for the Hamming distance.

Fingerprints can be also differentiated based on the symbols which they refer to. The choice of the specific symbol set is important when it comes to an empirical evaluation and it is discussed in more detail in Section 4. We have identified the following possibilities.

- **Common**: A set of symbols which appear most commonly in a given collection.
- **Rare**: A set of symbols which appear least commonly in a given collection.
- **Mixed**: A mixed set where half of the symbols comes from the common set while the other half comes from the rare set.

3.1 Fingerprint Examples

In the following examples, we constrain ourselves to the variant of 2-byte (16-bit) fingerprints with common letters. Fingerprints could in principle have any size, and the longer the fingerprint, the more information we can store about the character distribution in the string. Still, we regard 2 bytes, which correspond to the size of 2 characters in the original string, to be a desirable compromise between size and performance (consult the following section for experimental results). The choice of common letters is arbitrary at this point and it only serves the purpose of idea illustration.

In the following examples, occurrence fingerprint is constructed using selected 16 most common letters of the English alphabet, namely {`e`, `t`, `a`, `o`, `i`, `n`, `s`, `h`, `r`, `d`, `l`, `c`, `u`, `m`, `w`, `f`} [18, p. 36]. For the occurrence halved and count fingerprints (with $b = 2$ bits per count), we use the first 8 letters from this set. In the case of a position fingerprint (with $p = 3$ bits per letter), we use the first 5 letters for storing their positions and the sixth letter `n` for the last (single) occurrence bit.

Each fingerprint type would be as follows for the word `instance` (spaces are added only for visual presentation):

- **Occurrence**:
  `1 1 1 0 1 1 1 0 0 0 0 1 0 0 0 0`
  The first (leftmost) bit corresponds to the occurrence of the letter `e` (which does occur in the word, hence it is set to `1`), the second bit corresponds to the occurrence of the letter `t`, etc.
- **Occurrence halved**:
  `01 10 01 00 10 11 10 00`
  The first (leftmost) bit corresponds to the occurrence of the letter `e` in the first half of the word, that is `inst`; the second bit corresponds to the occurrence of the

letter `e` in the second half of the word, that is `ance`; the third bit corresponds to the occurrence of the letter `t` in the first half of the word, the fourth bit corresponds to the occurrence of the letter `t` in the second half of the word, etc.

- **Count**:

  `01 01 01 00 01 11 01 00`

  For reasons which will become clear later (see proof of Theorem 1), we use a Gray code [12], in which the 2-bit encodings of numbers $\{0, 1, 2, 3\}$ are `00`, `01`, `11`, and `10`, respectively. The first two (leftmost) bits correspond to the count of the letter `e` (it occurs once, hence the count is `01`, that is 1), the second two bits correspond to the count of the letter `t` (it occurs once, hence the count is `01`, that is 1), etc.

- **Position**:

  `111 011 100 111 000 1`

  The first three (leftmost) bits correspond to the position of the first occurrence of the letter `e` (this 0-indexed position is equal to 7, hence it is set to 111), the second three bits correspond to the position of the first occurrence of the letter `t` (this 0-indexed position is equal to 3, hence it is set to `011`), etc. The last (rightmost) occurrence bit indicates the occurrence of `n`, and since this letter does occur in the input string, this bit is set to `1`.

3.2 Construction

The construction of various fingerprint types is described below. For the description of symbols and types, consult preceding subsections. At the beginning, each bit of the fingerprint is always set to `0`.

- **Occurrence**: Let us remind the reader that the length of the fingerprint is equal to σ'_{occ} for a selected alphabet Σ'_{occ} of letters whose occurrences are stored. A string is iterated characterwise. For each character c, a mask `0x1` is shifted q times to the left, where $q \in \{0, \ldots, \sigma'_{occ} - 1\}$ is a corresponding shift for the character c. In other words, there exists a mapping $c \to q$ for each character $c \in \Sigma'_{occ}$. A natural approach to this mapping is to take the position of a symbol in the alphabet Σ'_{occ} (assuming that the alphabet is ordered). The fingerprint is then `or`-ed with the mask in order to set the bit which corresponds to character c to `1`. For a string of length n, time complexity of this operation is equal to $O(n)$.
- **Occurrence halved**: The fingerprint is constructed in an analogous way to the occurrence approach described above. We start with iterating the first half of the string, setting corresponding bits depending on letter occurrences, and then we iterate the second half of the string, again setting corresponding bits, which are shifted by 1 position with respect to bits set while iterating the first half of the string. Character mapping is adapted accordingly.

- **Count**: A string is again iterated characterwise. The length of the fingerprint is equal to $b \cdot \sigma'_{count}$ for a selected alphabet Σ'_{count} of letters whose counts are stored. Similarly to the occurrence fingerprint, there exists a mapping $c \rightarrow q$ for each character $c \in \Sigma'_{count}$. However, since we need b bits in order to store a count, it holds that $q \in \{0, b, \ldots, \sigma'_{count} - b\}$, assuming that b divides σ'_{count}. A selected bit mask is set and the fingerprint is then `or`-ed with the mask in order to increase the current count of character c which is stored using b bits at positions $\{q, q+1, \ldots, q+b-1\}$.

  Instead of a natural binary encoding, we however use a Gray code, in which the encodings for any pair of successive values (e.g., 1 and 2) differ at a single bit position. To increment a b-bit field, from value i to $i+1$ (where $0 \le i < i+1 < 2^b$), it is sufficient to extract this Gray-encoded field into a machine word W, and perform the operation $W := W \oplus (W \gg 1)$. The lowest b bits of W will then store the Gray-encoded value $i+1$. Naturally, we subsequently need to overwrite the original field with the obtained value from W. All of the above steps can be realized using a few simple bitwise operations. Assuming fixed b, for a string of length n, the time complexity of this operation is equal to $O(n)$.

- **Position**: In the case of position fingerprints, the length of the fingerprint is equal to $\sigma'_{pos} \cdot p$ for a selected alphabet Σ'_{pos} of letters whose positions are stored and a chosen constant p which indicates the number of bits per position. Here, we iterate the alphabet, and for each character $c \in \Sigma'_{pos}$ we search for the first (leftmost) occurrence of c in the string. Each position of such an occurrence is then successively encoded in the fingerprint, or the position *pos* is set to all `1`s if $pos \ge 2^p - 1$. For a string of length n, the time complexity of this operation is equal to $O(n \cdot \sigma'_{pos})$.

3.3 Comparison

We can quickly compare two *occurrence* (or occurrence halved) fingerprints by performing a binary `xor` operation and counting the number of bits which are set in the result (that is, calculating the Hamming weight, H_W). Let us note that H_W can be determined in constant time using a lookup table with $2^{8|S'|}$ entries, where $|S'|$ is the fingerprint size in bytes. We denote the fingerprint distance with F_D, and for occurrence fingerprints $F_D(S'_1, S'_2) = H_W(S'_1 \oplus S'_2)$. In other words, we count the number of mismatching character occurrences which are stored in individual bits.

However, let us note that F_D does not determine the true number of errors. For instance, for $S_1 =$ `run` and $S_2 =$ `ran`, F_D might be equal to 2 (occurrence differences for `a` and `u`) but there is still only one mismatch. On the other extreme, for two strings of length n, where each string consists of a repeated occurrence of one different symbol, F_D might be equal to 1 (or even 0, if the symbols are not included in the fingerprints), but the number of mismatches is n. In general, F_D can be used in order to provide a lower bound on the true number of errors, and the following relation holds (the right-hand side can be calculated quickly using a lookup table, since $0 \leq F_D \leq 8|S'|$):

$$D(S_1, S_2) \geq \lceil F_D(S'_1, S'_2)/2 \rceil, D \in \{Ham, Lev\}. \tag{1}$$

This formula also holds for the *count* fingerprint. Let us observe that with the use of a Gray code, the value of F_D might be underestimated, e.g., for comparing two 4-bit counters storing values 4 and 7, we have $H_W(\texttt{0110} \oplus \texttt{0100}) = H_W(\texttt{0010}) = 1$, however, it is not overestimated. As far as the *position* fingerprint (which is relevant only to the Hamming metric) is concerned, after calculating the `xor` value, we do not compute the Hamming weight, rather, we compare each set of bits (p-gram) which describes a single position. The value of F_D is equal to the number of mismatching p-grams. Similarly to other fingerprint types, these values can be preprocessed and stored in a lookup table in order to reduce calculation time.

The relationship between the fingerprint error and the true number of errors is further explored in Theorem 1 and Theorem 2. In plain words, manipulating a single symbol in either string makes the fingerprint distance grow by at most 2. Let us note that Formula (1) follows as a direct consequence of this statement, with the round-up on the right-hand side resulting from the fact that fingerprint distance might be odd.

Theorem 1. Consider $\mathcal{F} = \{occ, count\}$ and assume a distance function $D \in \{Ham, Lev\}$. For any two strings S_1 and S_2, with their fingerprints S'_1 and S'_2, respectively, and the fingerprint distance between them $F_D(S'_1, S'_2) = f(S'_1, S'_2)$, where $f \in \mathcal{F}$, we have that for any string S_3 such that $D(S_2, S_3) = 1$, the following relation holds: $F_D(S'_1, S'_3) \leq F_D(S'_1, S'_2) + 2$.

Proof. Let us first consider the occurrence fingerprints and Hamming distance (that is $D = Ham$). For this distance, two strings must be of equal length (otherwise the distance is infinite), and we set $|S_1| = |S_2| = n$. The string S_2 can be obtained from S_1 by changing some of its $k = D(S_1, S_2)$ symbols, at positions $1 \leq i_1 < i_2 < \ldots < i_k \leq n$. Let V_0 be an initial copy of S_1 and in k successive steps we transform it into $V_1, V_2, \ldots, V_k = S_2$, by changing one of its symbols at a time. For clarity, we shall modify the symbols in the order of their occurrence in the strings (from left to right). We shall observe how the changes affect the value of $F_D(S'_1, V'_j)$, which is initially (i.e., for $j = 0$) equal to zero.

Consider a j^{th} step, for any $1 \leq j \leq k$. We have four cases:

(i) both $V_{j-1}[i_j] \in \Sigma'$ and $V_j[i_j] \in \Sigma'$,

(ii) both $V_{j-1}[i_j] \notin \Sigma'$ and $V_j[i_j] \notin \Sigma'$,

(iii) $V_{j-1}[i_j] \in \Sigma'$ but $V_j[i_j] \notin \Sigma'$,

(iv) $V_{j-1}[i_j] \notin \Sigma'$ but $V_j[i_j] \in \Sigma'$.

Let us notice that:

- in case (i) $H_W(V'_j) - H_W(V'_{j-1}) \in \{-1, 0, 1\}$, yet since $V_{j-1}[i_j] \neq V_j[i_j]$, we may obtain new mismatches at (at most) two positions of the fingerprints, i.e., $F_D(S'_1, V'_j) - F_D(S'_1, V'_{j-1}) \leq 2$,
- in case (ii) $V'_j = V'_{j-1}$ and thus $F_D(S'_1, V'_j) = F_D(S'_1, V'_{j-1})$,
- in case (iii) $H_W(V'_{j-1}) - H_W(V'_j) \in \{0, 1\}$ and $F_D(S'_1, V'_j) - F_D(S'_1, V'_{j-1}) \leq 1$,
- in case (iv) $H_W(V'_j) - H_W(V'_{j-1}) \in \{0, 1\}$ and $F_D(S'_1, V'_j) - F_D(S'_1, V'_{j-1}) \leq 1$.

From the shown cases and by the triangle inequality we conclude that replacing a symbol with another makes the fingerprint distance grow by at most 2.

Now we change the distance measure to the Levenshtein metric (i.e., we set $D = Lev$). Note that the set of available operations transforming one string into another is extended; not only substitutions are allowed, but also insertions and deletions. The overall reasoning follows the case of Hamming distance, yet we need to consider all three operations. A single substitution in V_j, for a j^{th} step, makes the fingerprint distance grow by at most 2, in the same manner as shown above for the Hamming distance. Inserting a symbol c into V_j (at any position) implies one of three following cases:

(i) $c \notin \Sigma'$, where the fingerprint distance remains unchanged,

(ii) $c \in \Sigma'$ and $c \in S_1$, where again the fingerprint distance does not change, or

(iii) $c \in \Sigma'$ and $c \notin S_1$, where the fingerprint distance grows by 1.

Deleting a symbol c from V_j (at any position) implies one of three following cases:

(i) $c \notin \Sigma'$, where the fingerprint distance remains unchanged (same as for the insert operation),

(ii) $c \in \Sigma'$ and $c \in S_1$, where the fingerprint distance might not change (if V_j contains at least two copies of c) or it might grow by 1, or

(iii) $c \in \Sigma'$ and $c \notin S_1$, where the fingerprint distance might not change or it might decrease by 1. Note, however, that the last case for the delete operation never occurs in an edit script transforming S_1 into S_2 using a minimum number of Levenshtein operations.

Handling $f = count$ is analogous to the presented reasoning for $f = occ$, both for the Hamming and the Levenshtein distance. Note that changing a symbol's count by 1, where the count is stored in a b-bit field, may change up to b bits in natural binary encoding while it changes only 1 bit in Gray encoding, which is why we use the latter representation. □

Theorem 2. Consider $F_D = pos$ and assume a distance function $D = Ham$. For any two strings S_1 and S_2, with their fingerprints S'_1 and S'_2, respectively, we have that for any string S_3 such that $D(S_2, S_3) = 1$, the following relation holds: $F_D(S'_1, S'_3) \leq F_D(S'_1, S'_2) + 2$.

Proof. For the Hamming distance, two strings must be of equal length and let us set $|S_1| = |S_2| = n$. Similarly to the case of occurrence and count fingerprints, the string S_2 can be obtained from S_1 by changing some of its $k = D(S_1, S_2)$ symbols. This proof follows the same logic as presented in proof for Theorem 1. Let us note that the only difference lies in the fact that we compare the first position of a given letter rather than its occurrence. We deal with the same four cases depending on whether a modified letter belongs to Σ', and modifying a single letter may change: in case (i) at most two p-grams (which describe the position of the first occurrence of a given letter), in case (ii) 0 p-grams, in case (iii) at most one p-gram, and in case (iv) at most one p-gram (that is, the result of these two latter cases is equivalent). Again, as before, the number of modified p-grams corresponds directly to the maximum change in fingerprint distance. $\square$

3.4 Storage

Even though the true distance is higher than the fingerprint distance F_D, fingerprints can still be used in order to speed up comparisons because certain strings will be compared (and rejected) in constant time using only a fixed number of fast bitwise operations and array lookups. As mentioned before, we consider a scenario where a number of strings is preprocessed and stored in a collection. Since the construction of a fingerprint for the query string might be time-consuming, fingerprints are useful when the number of strings in a collection is relatively high. When it comes to the space overhead incurred by the fingerprints, for a dictionary $\mathcal{D}$ containing $|\mathcal{D}|$ keywords, it is equal to $O(|\mathcal{D}||S'| + 2^{|S'|} + \sigma)$, $|S'|$ being the (constant) fingerprint size in bytes. This holds since we have to store one constant size fingerprint per keyword together with the lookup tables which are used in order to speed up fingerprint comparison. These tables include one for determining the number of mismatches between two fingerprints (depending on fingerprint type: between occurrences, between counts, etc.) and one for the resulting number of errors (see Formula (1)). Let us note that this overhead is relatively small, especially when the size of each string is large (this is further discussed in the next section).

4 EMPIRICAL STUDY

Experimental results were obtained on the machine equipped with the Intel i7-4930K processor running at 3.4 GHz and 64 GB DDR3 RAM (1.6 GHz, latency timing 9-9-9-27). The source code and a compiled Linux binary executable (using `gcc` 64-bit version 5.4.0) are publicly available under the following link: `https://github.com/MrAlexSee/Fingerprints`. Consult Appendix A for more information regarding the usage of this tool.

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

- **Synthetic data**: 9.0 MB, generated based on English language letter frequencies [18, p. 36], 500 000 words.

- **English insane** (real-world data): 3.18 MB, American English language dictionary, 350 518 words.
- **English 200** (real-world data): 7.41 MB, words extracted from the English 200 collection from the Pizza&Chili index (`http://pizzachili.dcc.uchile.cl/texts/nlang/english.200MB.gz`), 815 935 words. This might be regarded as a kind of a middle-ground between synthetic and real-world data. Sequences were split on any white space and they included only printable characters. These words are usually not actual English language words (they contain, e.g., punctuation marks), however, they appear as part of the English text (hence they might be searched for in practice).
- **URLs** (real-world URL data): 95.02 MB of web addresses, available online: `http://data.law.di.unimi.it/webdata/in-2004/`, 1 382 908 words.

All dictionaries were filtered in order to contain only ASCII characters (given sizes pertain to dictionaries after said filtering, not counting duplicate words or delimiters, $1\,\mathrm{MB} = 10^6\,\mathrm{B}$). All dictionaries and query collections are available directly from the Github repository mentioned above, with the exception of URLs, owing to size limitations.

The number of queries of a given size (letter count) was equal to 10 000, and the number of iterations was set to 100. Each iteration consisted in a single search for each query within the dictionary. All presented results, including searching and construction, refer to single-thread performance, measured as elapsed CPU time. For the calculation of the Hamming distance, a regular loop which compares each consecutive character until k mismatches are found was used. It turned out that this implementation was faster than any other low-level approach (e.g., directly using certain processor instructions from the SSE extension set) when full compiler optimization (level `O3`) was used. For the Levenshtein distance, we used our own implementation based on the optimal calculation of the $2k + 1$ strip and the 2-row window [14]. It turned out to be faster than publicly available implementations, for instance the version from the Edlib library [31] or the SeqAn library [11]. This was probably caused by the fact that we could use the most lightweight solution and thus omit certain layers of abstractions from the libraries, especially since the comparison function was invoked multiple times for relatively short strings.

Queries were extracted randomly from the dictionary and compared against this dictionary. We have also tried distorting the queries by inserting a number of errors. For each query, the number of errors was uniformly sampled in the range $[1, e]$, and the timing results were consistent for any e in the $[1, \ldots, 4]$ range. In the case of English language dictionaries, we have also tested queries which consisted of the most common words extracted from a large corpus of the English language, and identical behavior was observed as in the case of queries which were sampled from the dictionary. This test was performed in order to check whether the words which are more likely to be searched for in practice exhibit the same behavior as other words.

Each fingerprint occupied 2 bytes, since 1-byte fingerprints turned out to be ineffective, and we regarded this as the optimal value with respect to a reasonable keyword size. The mode length in English dictionaries was equal to 8, which means that each fingerprint roughly incurred a 25 % storage penalty on average; however, the mode length in the URL collection was equal to 69, which means that each fingerprint roughly incurred only a 3 % storage penalty on average. Count fingerprints used 2 bits per count, that is, we set $b = 2$ and a natural binary encoding (i.e., not a Gray code). For correctness, in the case of the count variant, multiple mismatches between 2-bit counters were treated as a single mismatch. This was the case since, e.g., the difference between 1 and 2 is equal to 1, but $\texttt{01} \oplus \texttt{10} = \texttt{11}$ (the Hamming weight of such result is equal to 2 and not to 1). Position fingerprints used 3 bits per position, that is, we set $p = 3$ (consult Section 3 for details). Given the selected fingerprint size of 2 bytes (16 bits), these values allow for the use of 8 letters for count fingerprints and 5 letters for position fingerprints, with an extra occurrence bit in the latter case.

In our implementation, fingerprint comparison requires performing one bitwise operation and 2 array lookups, that is, 3 constant operations in total. We analyze the comparison time between two strings using various fingerprint types versus an explicit verification. When the fingerprint comparison was not decisive (i.e., we could not reject the match based solely on the use of fingerprints), a verification consisting in distance calculation was performed and it contributed to the elapsed time. The fingerprint is calculated once per query and it is then reused for the comparison with consecutive keywords. This means that we examine the situation where a single query is compared against a set (dictionary) of keywords.

4.1 Results

Figure 1 demonstrates the results for synthetic English data, which allowed us to check a wide range of word sizes (which occur infrequently or not at all in natural language corpora) for occurrence, count, and position fingerprints. Hamming distance was used as a similarity metric in this case. As described in the previous section, common, mixed, and rare letter sets were selected based on English alphabet letter frequencies. We can observe that the effectiveness of various approaches depends substantially on the word size, and the performance of letter sets also depends on the fingerprint type. The highest speedup was provided by occurrence fingerprints for common letters in the case of words of 10 characters, and it was equal to over 2.5 times with respect to the naive comparison.

In Tables 1 and 2 we present the speedup for $k = 1$ that was achieved for two English dictionaries and the URL data. Word lengths of 8 (in total 48 636 words for English insane and 104 753 words for English 200) and 69 (in total 34 044 words for URLs) were used, which corresponded to mode length values in the tested dictionaries. The speedup S was calculated using the following formula: $S = T_n/T_f$, where T_n refers to the average time required for a naive comparison (i.e., not using fingerprints), and T_f refers to the average time required for comparison using fin-

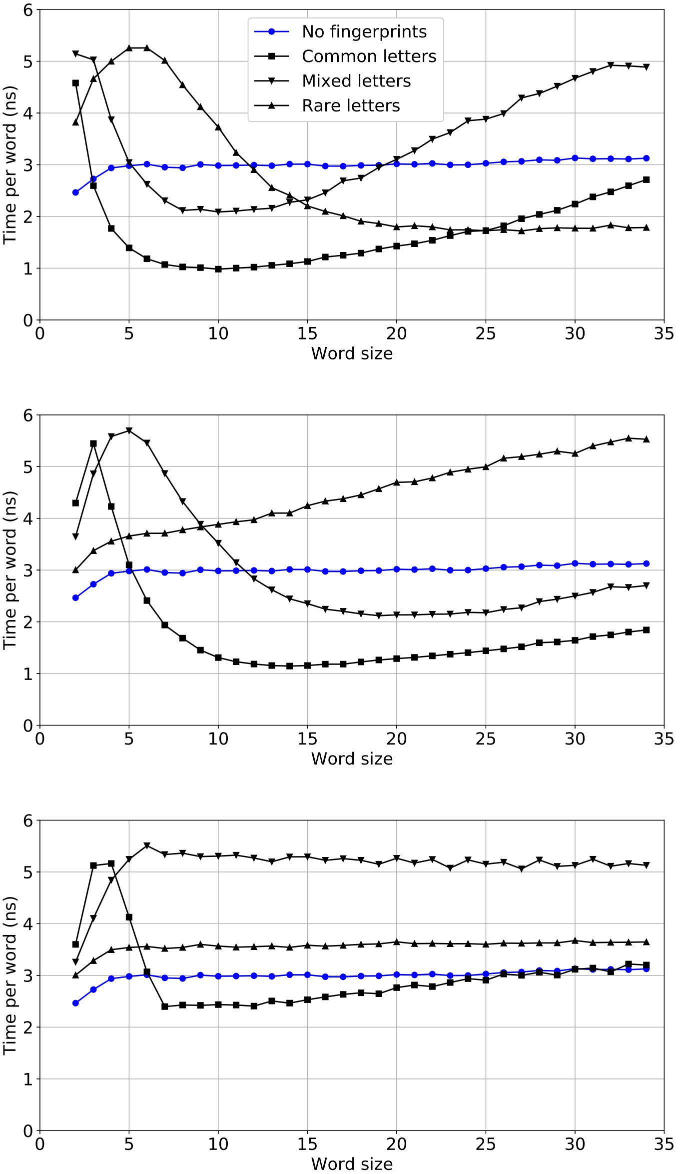

Figure 1. Comparison time vs. word size for 1 mismatch (**Hamming** distance) for **synthetic** data. Words were generated over the English alphabet. Time refers to average comparison time between a single pair of words. The upper figure shows results for **occurrence** fingerprints, the middle figure shows results for **count** fingerprints, and the bottom figure shows results for **position** fingerprints.

gerprints. For instance, 2.0 means that the time required for comparison decreased twofold when fingerprints were used.

A higher speedup in certain cases for the URL data was caused by a higher level of similarity between the data. In particular, the data set comprised some URLs which referred to different resources that were located on the same server. This resulted in certain words sharing a common prefix, requiring a naive algorithm to proceed with checking at least several first characters of each word. Presented results also demonstrate a limitation of our technique, which is apparent in the case of shorter words, where using fingerprints may increase the comparison time. Position fingerprints are not listed for the URL data, since they were completely ineffective due to multiple common prefixes between words and a large word length (almost no words were rejected). Let us also note that Hamming distance results for the English words are consistent with those reported for synthetic English data.

English Insane	Common	Mixed	Rare
Occurrence	2.66	1.45	0.72
Occurrence halved	2.05	0.97	0.69
Count	1.26	0.64	0.72
Position	1.03	0.55	0.80

English 200	Common	Mixed	Rare
Occurrence	2.12	1.03	0.54
Occurrence halved	1.35	0.61	0.75
Count	0.88	0.50	0.77
Position	0.71	0.52	0.82

URLs	Common	Mixed	Rare
Occurrence	1.46	1.19	2.27
Occurrence halved	1.78	1.93	1.53
Count	1.82	1.78	1.38

Table 1. Speedup for various fingerprint types relative to a naive comparison for $k = 1$ using **Hamming** distance for **real-world** data (English and URL dictionaries). Values smaller than 1.0 indicate that there was no speedup and the time required for comparison increased. The results in upper and middle table were calculated for the set of English language words of length 8, and the results in the lower table were calculated for the set of URLs of length 69 (both length values were modes of the word lengths in the respective dictionaries).

In Table 3 we list percentages of words that were rejected for the same data sets for $k = 1$ as a hardware-independent method of comparing our approaches. The rejection rate is naturally positively correlated with the speedup in comparison time. Table 4 presents the construction speed. Time measured during the construction covered (i) creating fingerprints, (ii) storing fingerprints in our custom dynamic container, (iii) storing words from the input dictionary (not counting disk I/O) in the same container. It is assumed that the words in the dictionary are already sorted, a stage which can be easily performed as preprocessing (most available dictionaries are already sorted anyway).

English Insane	Common	Mixed	Rare
Occurrence	33.38	10.19	3.44
Count	8.34	2.95	1.05

English 200	Common	Mixed	Rare
Occurrence	22.62	6.50	1.98
Count	5.11	1.98	1.01

URLs	Common	Mixed	Rare
Occurrence	3.60	2.19	14.35
Count	5.52	5.87	3.06

Table 2. Speedup for various fingerprint types relative to a naive comparison for $k = 1$ using **Levenshtein** distance for **real-world** data (English and URL dictionaries). The results in upper and middle table were calculated for the set of English language words of length 8, and the results in the lower table were calculated for the set of URLs of length 69.

Let us also discuss a related method, namely neighborhood (permutation) generation [23]. For a given pattern P, it consists in constructing all combinations of perturbed words derived from P, whose presence in the dictionary is then checked in an exact manner (using, e.g., a hash table). For instance, using Hamming distance for a word `cat` and English alphabet, one would first check a perturbation using letter `a`: `aat`, `cat` (can be ignored, since it is the same as the pattern), and `caa`, then using letter `b`: `bat`, `cbt`, `cab`, etc.

If the neighborhood size (that is, the count of generated candidates), which depends on the pattern length and the alphabet size, is relatively small and the dictionary size (that is, the total word count) is relatively large, this might turn out to be a promising approach. On the other hand, fingerprints are a more versatile method, which can be used for speeding up a comparison of any two strings. This stands in contrast to only checking for the presence of a string in a dictionary (as is the case for the neighborhood method), and fingerprints can be used for augmenting another data structure (see Section 5 for more information).

For the comparison of neighborhood generation and a fingerprint-based search when querying a dictionary, consult Figure 2. The search was performed for Hamming and Levenshtein metrics with $k = 1$. Time refers to average comparison time between a single pair of words, in the same manner as for fingerprint comparison presented in Figure 1. Both dictionaries were subsampled in order to contain the tested number of words, namely $[2^6, 2^7, \ldots, 2^{16}]$. This consisted in randomly selecting words of size 8, which was the mode length in both dictionaries.

English Insane	Common	Mixed	Rare
Occurrence (Ham, Lev)	98.45 %	92.53 %	75.84 %
Occurrence halved (Ham)	96.72 %	85.30 %	10.12 %
Count (Ham, Lev)	90.55 %	71.37 %	7.01 %
Position (Ham)	87.80 %	50.30 %	0.65 %

English 200	Common	Mixed	Rare
Occurrence (Ham, Lev)	97.22 %	87.73 %	55.07 %
Occurrence halved (Ham)	92.34 %	71.39 %	3.17 %
Count (Ham, Lev)	84.02 %	55.30 %	2.19 %
Position (Ham)	78.47 %	39 42 %	0.23 %

URLs	Common	Mixed	Rare
Occurrence (Ham, Lev)	71.00 %	55.72 %	89.39 %
Occurrence halved (Ham)	80.33 %	84.03 %	73.41 %
Count (Ham, Lev)	80.87 %	80.55 %	67.33 %

Table 3. Percentage of **rejected words** for various fingerprint types for $k = 1$ for **real-world** data (English and URL dictionaries). Rejection means that the true error was determined to be more than k based only on fingerprint comparison. The results in upper and middle table were calculated for the set of English language words of length 8, and the results in the lower table were calculated for the set of URLs of length 69.

English Insane	Common	Mixed	Rare
Occurrence	498.56	499.10	497.62
Occurrence halved	475.70	475.18	474.26
Count	155.14	165.36	316.29
Position	154.35	167.62	196.97

English 200	Common	Mixed	Rare
Occurrence	485.61	485.76	485.61
Occurrence halved	458.59	458.68	461.25
Count	165.13	190.57	335.54
Position	160.57	170.90	198.26

URLs	Common	Mixed	Rare
Occurrence	418.75	419.07	419.05
Occurrence halved	405.91	406.03	405.93
Count	244.99	270.82	357.00

Table 4. **Construction speed** given in MB/s ($1\,\mathrm{MB} = 10^6\,\mathrm{B}$) for various fingerprint types for **real-world** data (English and URL dictionaries). The results in upper and middle table were calculated for the set of English language words of length 8, and the results in the lower table were calculated for the set of URLs of length 69.

The largest value ($2^{16} = 65\,536$) was used only for the English 200 dictionary, since the English insane dictionary did not contain such number of words having 8 characters. The alphabet size is equal to 53 and 94 for English insane and English 200, respectively, and it was accordingly smaller for subsampled dictionaries (e.g., 33 and 56 characters for English dictionaries with 64 words, respectively). All these dictionaries can be inspected at the Github repository. The number of iterations for neighborhood generation was equal to 100 (the same as for the fingerprints).

We can see that fingerprints outperform the neighborhood generation method for smaller dictionaries, up to around 2 orders of magnitude for the smallest one (which is, admittedly, not likely to be used in a real-world scenario). As the dictionary size increases, the gap becomes smaller, with neighborhood generation being faster for the largest subsampled dictionaries. In regard to the Levenshtein distance, on the one hand there exist more combinations which need to be checked by the permutation algorithm (insertions, deletions), on the other hand, Levenshtein distance is also more expensive to calculate when the fingerprint comparison is not decisive. Nevertheless, the neighborhood generation algorithm was slower for the Levenshtein metric when compared to the Hamming metric very roughly by a factor of 5, and fingerprints were relatively slower very roughly by a factor of 2. This meant that for dictionaries where the fingerprint-based approach turned out to be faster, it outperformed neighborhood generation by a wider margin for the Levenshtein than for the Hamming distance.

Let us note that the time required to extract the alphabet from the dictionary (which was needed for generating the neighborhood) did not contribute to the measured elapsed time. The tested implementation of the neighborhood generation method can be also found in the Github repository referenced previously.

In general, the choice of the optimal strategy, viz. fingerprint type, letters data set, and how many bits are used per single counter or position in a fingerprint, depends chiefly on the input data. Larger fingerprints would allow for obtaining a better rejection rate, but this would come at the cost of increased space usage. Once the rejection rate is close to the optimal 100 %, larger fingerprints would provide only a negligible reduction in processing time. In our case, the simplest approach, that is, occurrence fingerprints with common letters, seemed to offer the best performance. Still, we would like to point out that a practical evaluation on a specific data set would be advised in a real-world scenario.

5 CONCLUSIONS

We have evaluated fingerprints in the context of dictionary matching. Still, we would like to emphasize the fact that fingerprints are not a data structure in itself, rather, they are a string augmentation technique which we believe may prove useful in various applications. For instance, they can be used in any data structure which performs multiple internal approximate string comparisons, providing

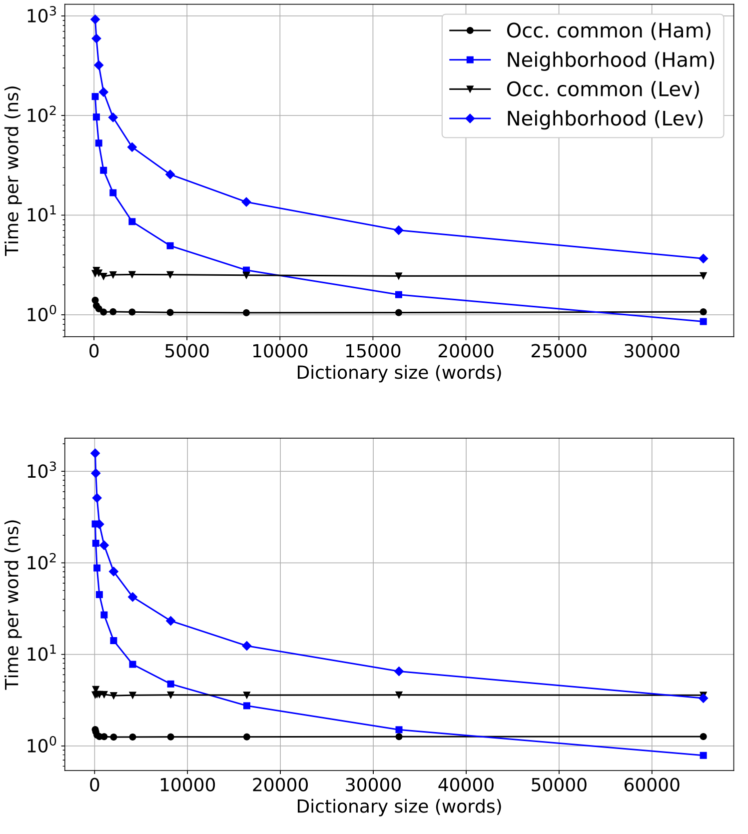

Figure 2. Comparison time vs. dictionary size for 1 error using **fingerprints** (occurrence common variant, which turned out to be the fastest one) and the related **neighborhood generation** method. Results for the English insane dictionary (upper figure) and the English 200 dictionary (lower figure) are presented, both for words of size 8. Note the logarithmic y-scale.

substantial speedups at a modest increase in the occupied space. In particular, for longer strings such as URL sequences the space overhead can be considered negligible.

Fingerprints take advantage of the letter distribution, and for this reason they were not effective for strings sampled over the alphabet with a uniform random distribution. They are also not recommended for the DNA data due to the small size of the alphabet and a large average word size. These two combined properties result in a scenario where each word contains multiple occurrences of each possible letter with a high probability.

In the future, we would like to extend the notion of a fingerprint by encoding information regarding not only single symbol distributions, but rather q-gram distributions. The set of q-grams could be determined either heuristically or using an exhaustive search, and their use might provide a speedup for any real-world data set (possibly including DNA sequences). We believe that it may be also beneficial for processing larger k values. Another possibility lies in combining different fingerprint types for a single word in order to further decrease comparison time at the cost of increased space usage. We also plan to employ fingerprints in order to speed up internal substring comparison in another data structure which we have previously created, namely the split index [9].

Acknowledgement

We thank the anonymous reviewers for their constructive comments which helped us improve the initial version of the manuscript.

A TOOL USAGE

The source code and a compiled Linux binary executable of the `fingerprints` tool are publicly available under the following link: `https://github.com/MrAlexSee/Fingerprints`. This description refers to the release version `v1.3.0` (in order to directly obtain the binary executable use the link: `https://github.com/MrAlexSee/Fingerprints/releases/download/v1.3.0/fingerprints`).

In order to reproduce experiments described in this paper, download the source code, set the path to `Boost` library in the makefile (variable `BOOST_DIR`) and issue a command `make` in the main directory. Alternatively, download directly the aforementioned compiled executable for the Linux operating system.

As mentioned in the chapter on empirical study, dictionaries and corresponding queries can be found in the `data` folder, with the exception of the URLs dictionary, which should be downloaded separately due to size restrictions (the relevant link is provided in Section 4).

In order to test the synthetic data, use the `test_synth.sh` script, and in order to test the real-world data, use the `test_real.sh` script. Both scripts automatically examine all fingerprint and letters type combinations. In order to compare fingerprints performance with a related neighborhood generation method, refer to the folder `related`. A complete list of command-line parameters which can be provided to the executable is located in Table 5.

Short Name	Long Name	Parameter Description
	`--calc-rejection`	calculate percentages of rejected words instead of measuring time
`-d`	`--dump`	dump input files and parameters info with elapsed time and throughput to output file (useful for testing)
	`--dump-construction`	dump fingerprint construction time
`-D`	`--distance arg`	distance metric: `ham` (Hamming), `lev` (Levenshtein) (default = `ham`)
`-f`	`--fingerprint-type arg`	fingerprint type: `none`, `occ` (occurrence), `occhalved` (occurrence halved), `count`, `pos` (position) (default = `occ`)
`-h`	`--help`	display help message
`-i`	`--in-dict-file arg`	input dictionary file path (positional argument 1)
`-I`	`--in-pattern-file arg`	input pattern file path (positional argument 2)
	`--iter arg`	number of iterations per pattern lookup (default = 1)
`-k`	`--approx arg`	perform approximate search (Hamming or Levenshtein) for k errors
`-l`	`--letters-type arg`	letters type: `common`, `mixed`, `rare` (default = `common`)
`-o`	`--out-file arg`	output file path (default = `res.txt`)
`-p`	`--pattern-count arg`	maximum number of patterns read from top of the pattern file
	`--pattern-size arg`	if set, only patterns of this size (letter count) will be read from the pattern file
`-s`	`--separator arg`	input data (dictionary and patterns) separator (default = newline)
`-v`	`--version`	display version info
`-w`	`--word-count arg`	maximum number of words read from top of the dictionary file

Table 5. A complete list of command-line parameters for the `fingerprints` tool

REFERENCES

[1] Bar-Yossef, Z.—Jayram, T. S.—Krauthgamer, R.—Kumar, R.: The Sketching Complexity of Pattern Matching. In: Jansen, K., Khanna, S., Rolim, J. D. P., Ron, D. (Eds.): Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (RANDOM 2004, APPROX 2004). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 3122, 2004, pp. 261–272, doi: 10.1007/978-3-540-27821-4_24.

[2] Belazzougui, D.—Kolpakov, R.—Raffinot, M.: Various Improvements to Text Fingerprinting. Journal of Discrete Algorithms, Vol. 22, 2013, pp. 1–18, doi: 10.1016/j.jda.2013.06.004.

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

[4] Bille, P.—Cording, P. H.—Gørtz, I. L.—Sach, B.—Vildhøj, H. W.—Vind, S.: Fingerprints in Compressed Strings. In: Dehne, F., Solis-Oba, R., Sack, J. R. (Eds.): Algorithms and Data Structures (WADS 2013). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 8037, 2013, pp. 146–157, doi: 10.1007/978-3-642-40104-6_13.

[5] Bocek, T.—Hunt, E.—Stiller, B.: Fast Similarity Search in Large Dictionaries. Technical Report No. ifi-2007.02, Department of Informatics, University of Zurich, 2007.

[6] Brodal, G. S.—Gasieniec, L.: Approximate Dictionary Queries. In: Hirschberg, D., Myers, G. (Eds.): Combinatorial Pattern Matching (CPM 1996). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1075, 1996, pp. 65–74, doi: 10.1007/3-540-61258-0_6.

[7] Cano, P.—Batlle, E.—Kalker, T.—Haitsma, J.: A Review of Audio Fingerprinting. Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology, Vol. 41, 2005, No. 3, pp. 271–284, doi: 10.1007/s11265-005-4151-3.

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

[9] Cisłak, A.—Grabowski, Sz.: A Practical Index for Approximate Dictionary Matching with Few Mismatches. Computing and Informatics, Vol. 36, 2017, pp. 1088–1106, doi: 10.4149/cai_2017_5_1088.

[10] 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, 2004, pp. 91–100, doi: 10.1145/1007352.1007374.

[11] Döring, A.—Weese, D.—Rausch, T.—Reinert, K.: SeqAn an Efficient, Generic C++ Library for Sequence Analysis. BMC Bioinformatics, Vol. 9, 2008, doi: 10.1186/1471-2105-9-11.

[12] Gray, F.: Pulse Code Communication. United States Patent Office Application, Serial No. US785697A, 1953.

[13] Grabowski, Sz.—Raniszewski, M.: Sampling the Suffix Array with Minimizers. In: Iliopoulos, C., Puglisi, S., Yilmaz, E. (Eds.): String Processing and Information Retrieval (SPIRE 2015). Springer, Cham, Lecture Notes in Computer Science, Vol. 9309, 2015, pp. 287–298, doi: 10.1007/978-3-319-23826-5_28.

[14] Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, 1997, doi: 10.1017/CBO9780511574931.

[15] Hamming, R. W.: Error Detecting and Error Correcting Codes. The Bell System Technical Journal, Vol. 29, 1950, No. 2, pp. 147–160, doi: 10.1002/j.1538-7305.1950.tb00463.x.

[16] Karp, R. M.—Rabin, M. O.: Efficient Randomized Pattern-Matching Algorithms. IBM Journal of Research and Development, Vol. 31, 1987, No. 2, pp. 249–260, doi: 10.1147/rd.312.0249.

[17] Levenshtein, V. I.: Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. Soviet Physics Doklady, Vol. 10, 1966, No. 8, pp. 707–710.

[18] Lewand, R. E.: Cryptological Mathematics. Mathematical Association of America, UK edition, 2000.

[19] Manber, U.: Finding Similar Files in a Large File System. Proceedings of the USENIX Winter 1994 Technical Conference (WTEC '94), San Francisco, California, 1994, pp. 1–10.

[20] Muth, R.—Manber, U.: Approximate Multiple Strings Search. In: Hirschberg, D., Myers, G. (Eds.): Combinatorial Pattern Matching (CPM 1996). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1075, 1996, pp. 75–86, doi: 10.1007/3-540-61258-0_7.

[21] Myers, E. W.: A Sublinear Algorithm for Approximate Keyword Searching. Algorithmica, Vol. 12, 1994, No. 4-5, pp. 345–374, doi: 10.1007/BF01185432.

[22] Navarro, G.: A Guided Tour to Approximate String Matching. ACM Computing Surveys (CSUR), Vol. 33, 2001, No. 1, pp. 31–88, doi: 10.1145/375360.375365.

[23] Navarro, G.—Baeza-Yates, R. A.—Sutinen, E.—Tarhio, J.: Indexing Methods for Approximate String Matching. IEEE Data Engineering Bulletin, Vol. 24, 2001, No. 4, pp. 19–27.

[24] Policriti, A.—Prezza, N.: Hashing and Indexing: Succinct DataStructures and Smoothed Analysis. In: Ahn, H. K., Shin, C. S. (Eds.): Algorithms and Computation (ISAAC 2014). Springer, Cham, Lecture Notes in Computer Science, Vol. 8889, 2014, pp. 157–168, doi: 10.1007/978-3-319-13075-0_13.

[25] Policriti, A.—Tomescu, A. I.—Vezzi, F.: A Randomized Numerical Aligner (rNA). Journal of Computer and System Sciences, Vol. 78, 2012, No. 6, pp. 1868–1882, doi: 10.1016/j.jcss.2011.12.007.

[26] 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.

[27] Prezza, N.: In-Place Sparse Suffix Sorting. Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '18), SIAM, 2018, pp. 1496–1508, doi: 10.1137/1.9781611975031.98.

[28] Rabin, M. O.: Fingerprinting by Random Polynomials. Technical Report TR-15-81, Department of Computer Science, Harvard University, 1981.

[29] Ramaswamy, R.—Kencl, L.—Iannaccone, G.: Approximate Fingerprinting to Accelerate Pattern Matching. 6th ACM SIGCOMM Conference on Internet Measurement (IMC '06), ACM, 2006, pp. 301–306, doi: 10.1145/1177080.1177120.

[30] Salmela, L.—Tarhio, J.—Kytöjoki, J.: Multipattern String Matching with Q-Grams. Journal of Experimental Algorithmics, Vol. 11, 2006, Art. No. 1.1, doi: 10.1145/1187436.1187438.

[31] Šošić, M.—Šikić, M.: Edlib: A C/C++ Library for Fast, Exact Sequence Alignment Using Edit Distance. Bioinformatics, Vol. 33, 2017, No. 9, pp. 1394–1395, doi: 10.1093/bioinformatics/btw753.

[32] Ukkonen, E.: Algorithms for Approximate String Matching. Information and Control, Vol. 64, 1985, No. 1-3, pp. 100–118, doi: 10.1016/S0019-9958(85)80046-2.

[33] Zhu, R. F.—Takaoka, T.: A Technique for Two-Dimensional Pattern Matching. Communications of the ACM, Vol. 32, 1989, No. 9, pp. 1110–1120, doi: 10.1145/66451.66459.

Aleksander Cisłak received his B.Sc. degree in computer science from Lodz University of Technology in 2014, M.Sc. degree in informatics from TU München in 2015, and Ph.D. degree in computer science from Warsaw University of Technology in 2019. 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.