Tuning Algorithms for Jumbled Matching

Tamanna Chhabra, Sukhpal Singh Ghuman, and Jorma Tarhio

Department of Computer Science
Aalto University
P.O. Box 15400, FI-00076 Aalto, Finland
firstname.lastname@aalto.fi

Abstract. We consider the problem of jumbled matching where the objective is to find all permuted occurrences of a pattern in a text. Besides exact matching we study approximate matching where each occurrence is allowed to contain at most k wrong or superfluous characters. We present online algorithms applying bit-parallelism to both types of jumbled matching. Most of our algorithms are variations of earlier algorithms. We show by practical experiments that our algorithms are competitive with the previous solutions.

Keywords: jumbled matching, Abelian matching, permutation matching, approximate string matching, comparison of algorithms, counting algorithm, algorithm engineering

1 Introduction

String matching [17] is a common problem in Computer Science. Let $T = t_1t_2 \cdots t_n$ and $P = p_1p_2 \cdots p_m$ be text and pattern respectively, over a finite alphabet Σ of size σ. The task of exact string matching is to find all the occurrences of P in T, i.e. all positions i such that $t_it_{i+1} \cdots t_{i+m-1} = p_1p_2 \cdots p_m$. In approximate string matching the objective is to find all substrings of T whose edit distance to P is at most k or which have at most k mismatches with P, where $0 \leq k < m$.

Jumbled matching [5,7] (also known as Abelian matching or permutation matching) is an interesting variation of string matching. The task is to find all substrings of T which are permutations of P. For instance, a permutation of `abcb` occurs in `cdfbbacda`. Jumbled matching can be formalized with Parikh vectors [19]. The Parikh vector $p(S)$ of a string S over a finite ordered alphabet is defined as the vector of multiplicities of the characters, for example $p(S)$ is (1,2,1,0) for $S =$ `abcb` in $\Sigma = \{a, b, c, d\}$.

Jumbled matching has applications in many areas such as alignment of strings [1], SPN discovery [3], discovery of repeated patterns [10], in the interpretation of mass spectrometry data [2]. In case of discovery of repeated patterns [10], jumbled matching algorithms can be used to solve the problem of local alignment of genes. Also the problem of matching of protein domain clusters [10] can be solved by these algorithms as the clusters have common functionality even though appear in different orders. In the field of interpretation of mass spectrometer [2], permutation matching is used to find the strings having the same spectra. Mass spectra are simulated for every potential sequence and comparing the resulting simulated spectra against the measured mass spectrum. In addition to that, permutation matching can be used in SNP [3] and mutation discovery which are based on base-specific cleavage of DNA or RNA and mass spectrometry. Composition alignment [1] is a process of matching strings whether they have equal length and same nucleotide content.

Tamanna Chhabra, Sukhpal Singh Ghuman, Jorma Tarhio: Tuning Algorithms for Jumbled Matching, pp. 57–66.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

Simple counting solutions [13,15,16] for jumbled matching work in linear time. The idea is to scan the text forward while maintaining counts of characters in a sliding alignment window of T. Originally, these counting algorithms were developed as filtration methods for online approximate string matching, but they recognize jumbled patterns as a side-effect when no errors are allowed. Also many other algorithms [4,6,9,12] have been introduced for jumbled matching. In this paper, we introduce new algorithms for jumbled matching and compare their efficiency with the previous solutions. Most of our algorithms are variations of earlier algorithms.

Besides traditional jumbled matching, we also consider approximate jumbled matching. We define an approximate permutation as follows. The string P' is a k-approximate permutation of P, $0 \leq k < m$, if $|P'| = |P| = m$ holds and

$$\sum_{c \in set(P')} \max(cc(P', c) - cc(P, c), 0) \leq k,$$

where $set(P')$ is the set of characters in P' and $cc(u, c)$ is the number of occurrences of a character c in a string u. Our definition of approximate jumbled matching is different from the one presented by Burcsi et al. [5]. Ejaz [9] considers also other cost models. Note that according to our definition, a 0-approximate permutation is an exact permutation. We will present linear and sublinear algorithms for both exact and approximate jumbled matching. By sublinear we mean those algorithms which are on average able to skip a part of the text.

In the pseudocodes of the algorithms, we use C-like notations '&' and '>>' representing bitwise operations AND and right shift, respectively. The size of the computer word is denoted by w. All the bitvectors and bit masks contain w bits.

Our main emphasis is on the practical efficiency of the algorithms which is demonstrated by experimental results. In almost every tested case, the best of our algorithms was faster than any previous solution, and in many cases even doubling the speed of the best previous solution.

The paper is organized as follows. Section 2 describes previous solutions for jumbled matching, Section 3 presents our solutions, Section 4 presents and discusses the results of practical experiments, and Section 5 concludes the article.

2 Previous solutions

Grossi & Luccio's and Navarro's solutions [13,15,16] are based on the frequency of characters occurring in the pattern and in an alignment window. These methods solve this problem in linear time. Navarro's counting algorithm is based on a sliding window approach. Alg. 1 presents the main loop of Navarro's algorithm called Count in the following. The variable C holds the additive inverse of the number of wrong or extra characters in an alignment window of m characters. The initial value of C is $-m$. The array A maintains the character counts of the alignment window such that $A[c]$ is $cc(t_{i-m+1} \cdots t_i, c) - cc(P, c)$. Before the main loop, the character counts of the first alignment window are collected to A, and the variable C is updated respectively.

Alg. 1 (The main loop of Count)
while $i \leq n$ do
  if $C \geq 0$ then report occurrence
  $A[t_{i-m}] \leftarrow A[t_{i-m}] + 1$
  if $A[t_{i-m}] > 0$ then $C \leftarrow C - 1$
  if $A[t_i] > 0$ then $C \leftarrow C + 1$
  $A[t_i] \leftarrow A[t_i] - 1$
  $i \leftarrow i + 1$

Grossi & Luccio's solution maintains a queue of characters which grows with acceptable characters until the length is m which means that an occurrence of P has been found. Another counting algorithm has been proposed by Grabowski et al. [12].

In addition to exact jumbled matching, Navarro's and Grossi & Luccio's algorithms can directly be applied to approximate jumbled matching as well, because a match candidate for the k mismatches problem is a k-approximate permutation of P. The initial value of C in Count is $k - m$ in the approximate case.

Besides a single pattern algorithm, Navarro [16] also presented a multipattern variation for patterns of equal length. Alg. 2 shows the main loop of this algorithm called Mcount in the following. Each pattern has a count variable (or a bin) of its own, and a field of $d + 1$ bits is allocated for it in D, a bitvector of w bits. The bitvector $E[c]$ holds a field of $d+1$ bits for each pattern. The initial values of fields in $E[c]$ and D are $2^d + cc(P_j, c) - 1$ and $2^d - (m - k)$, respectively, for the jth pattern. Before the main loop, the character counts of the first alignment window are collected to E, and D is updated respectively. During scanning, the value for the field of $E[c]$ for the jth pattern is $2^d + cc(P_j, c) - cc(t_{i-m+1} \cdots t_i, c) - 1$. The bit mask F holds one in the most significant bit of every field in D. The bit mask I holds one in the least significant bit of every field in $E[c]$. The operation $(E[c] >> d)\,\&\,I$ extracts the most significant bits of $E[c]$. The condition $D\,\&\,F \neq 0$ means that at least one overflow bit is set in D, i.e. $m - k$ acceptable characters of at least one pattern have been found. In the case of a single pattern, this is enough to recognize an occurrence. In the case of two or more patterns, a verification step is needed because the condition $D\,\&\,F \neq 0$ does not specify which pattern matches.

Alg. 2 (The main loop of Mcount)
while $i \leq n$ do
  if $D\,\&\,F \neq 0$ then verify occurrence
  $c \leftarrow t_{i-m}$
  $E[c] \leftarrow E[c] + I$
  $D \leftarrow D - (E[c] >> d)\,\&\,I$
  $c \leftarrow t_i$
  $D \leftarrow D + (E[c] >> d)\,\&\,I$
  $E[c] \leftarrow E[c] - I$
  $i \leftarrow i + 1$

Cantone and Faro [6] presented the BAM algorithm (Bit-parallel Abelian Matcher) which applies bit-parallelism and backward scanning of the alignment window. Alg. 3 shows the main loop of BAM. A field of $g(c) = \lceil \log cc(P, c) \rceil + 2$ bits is reserved for each character c appearing in P. As in Mcount, the most significant bit of each field is a kind of overflow bit. The initial value of the field is $2^{g(c)-1} - cc(P, c) - 1$ which means that $cc(P, c) + 1$ occurrences trigger the overflow bit. The adaptive width of bit fields make possible to handle longer patterns than a fixed width. Moreover, there

is a special field of one bit for characters not present in P. The bit mask $M[c]$ holds one in the least significant bit of the field of character c. The bit mask I holds the initial values of the fields, and the bit mask F holds one at each overflow bit.

Alg. 3 (The main loop of BAM)
while $i \leq n - m$ do
  $D \leftarrow I$; $j \leftarrow i + m - 1$
  while $j \geq i$ do
    $D \leftarrow D + M[t_j]$
    if $D \,\&\, F \neq 0$ then break
    $j \leftarrow j - 1$
  if $j < i$ then
    report occurrence
    $i \leftarrow i + 1$
  else $i \leftarrow j + 1$

Ejaz [9] proposed several algorithms for jumbled matching. One of them utilizes backward scanning of the alignment window. Moreover, Burcsi et al. [4] introduced a light indexing approach with linear construction time and with sublinear expected query time.

3 New solutions

We have designed various solutions for the exact and approximate jumbled matching. We explain them in the following subsections. Most of the algorithms are variations of Count, Mcount, or BAM.

3.1 Variations of BAM

If the pattern is long, w bits is not enough to hold a distinct bin for each character appearing in the pattern. We made BAMs, a variation of BAM where some bins are shared. In BAMs, characters for bins are selected circularly from the pattern in the right-to-left order. This is a kind of alphabet reduction. Then instead of "report occurrence" in Alg. 3, each match candidate should be verified.

Then we present two other algorithms that are modifications of BAM (Alg. 3). Alg. 4 is approximate BAM (ABAM for short) and Alg. 5 is enhanced BAM with 2-grams (BAM2 for short), respectively. In ABAM, $F[c]$ is the overflow bit of character c. The variable C counts errors. The width of the field for the character c is $\lceil \log(\max(cc(P, c), k)) \rceil + 2$. The width of the field for characters not present in P is $\lceil \log k \rceil + 2$. $M[c]$ and I are the same as in BAM. The enxpression "if $D \,\&\, F[t_j] \neq 0$ then 1 else 0" can be implemented as $(D \,\&\, F[t_j]) \,\&\&\, 1$ in C.

Alg. 4 (The main loop of ABAM)
while $i \leq n - m$ do
  $D \leftarrow I$; $C \leftarrow 0$; $j \leftarrow i + m - 1$
  while $j \geq i$ do
    $D \leftarrow D + M[t_j]$
    $C \leftarrow C +$ (if $D \,\&\, F[t_j] \neq 0$ then 1 else 0)
    if $C > k$ then break
    $j \leftarrow j - 1$
  if $j < i$ then
    report occurrence
    $i \leftarrow i + 1$
  else $i \leftarrow j + 1$

Alg. 5 shows the main loop of BAM2 for patterns of even length. BAM2 handles a 2-gram at a time. BAM2 has a separate loop for patterns of even and odd lengths. The loop for patterns of odd length has two lines more because the remaining leftmost character of the alignment window must be handled in a different way. Typically q-grams are used in string matching to process the right end of the alignment window. BAM2 processes the whole window with 2-grams (except the leftmost character in the case of odd m). This is beneficial because the alignment window is scanned on average further to the left in jumbled matching than in ordinary string matching. Moreover, 2-grams instead of single characters are read in our implementation of BAM2.

Alg. 5 (The main loop of BAM2)
while $i \leq n - m$ do
  $j \leftarrow i + m - 3$
  $D \leftarrow I + M_2[t_{j+1}, t_{j+2}]$
  do
    $D \leftarrow D + M_2[t_{j-1}, t_j]$
    if $D \,\&\, F = 0$ then break
    $j \leftarrow j - 2$
  until $j \geq i$
  if $j < i$ then
    report occurrence
    $i \leftarrow i + 1$
  else $i \leftarrow j$

BAM2 reads four characters before testing D. As a consequence, the minimum width of a bit field is four bits instead of two. The width of the field for characters not present in P is three bits. The array M_2 is precomputed as follows: $M_2[c_1, c_2] = M[c_1] + M[c_2]$.

For small alphabets we use BAM2 as presented in Alg. 5. For large alphabets we use BAM2 with the same bin sharing technique as applied in BAMs.

3.2 Other variations

Alg. 6 presents the main loop of EBL (short for "Exact Backward for Large alphabets"). EBL is based on SBNDM2 [8], which is a sublinear bit-parallel algorithm for exact string matching. Instead of representing occurrence vectors of characters, the array B states if the character c is present in the pattern: $B[c] = 1$ if c is present, otherwise $B[c] = 0$. As in SBNDM2, two characters are read before the first test in an alignment window. The update step of the state variable D is simply $D = D \,\&\, B[t_{i+j-1}]$. When the alignment window contains only acceptable characters,

the window is a match candidate, which should be verified. Whenever a forbidden text character is found, the alignment window is moved forward over that text position.

Alg. 6 (The main loop of EBL)
while $i \leq n - m$ do
  $j \leftarrow m - 1$
  $D = B[t_{i+j}] \,\&\, B[t_{i+j+1}]$
  while $D \neq 0$ and $j > 0$ do
    $D \leftarrow D \,\&\, B[t_{i+j-1}]$
    $j \leftarrow j - 1$
  if $D = 1$ then verify occurrence
  $i \leftarrow i + j + 1$

Alg. 7 presents the main loop of EFS (short for "Exact Forward for Small alphabets"). Like in Count and other algorithms of forward type, the first alignment window is processed before the main loop. The bitvector D has a field of d bits[1] initially $2^{d-1} - cc(P, c) - 1$ for each character c appearing in P. The characters not in P have a joint field of one bit. Like in BAM, D is tested with a mask F which has one at each overflow bit. $M[c]$ is a bit mask having one at the least significant bit of the field of character c.

Alg. 7 (The main loop of EFS)
while $i \leq n$ do
  if $D \,\&\, F = 0$ then report occurrence
  $D \leftarrow D + M[t_i] - M[t_{i-m}]$
  $i \leftarrow i + 1$

Alg. 8 presents the main loop of AFL (short for "Approximate Forward for Large alphabets"). AFL is a modification of Mcount tuned for a single pattern. The array E is the same as in Mcount in the case of a single pattern as well as the offset d. The initial value of the counter C is $k - m$. Like in the other algorithms of forward type, the first alignment window is processed before the main loop.

Alg. 8 (The main loop of AFL)
while $i \leq n$ do
  if $C \geq 0$ then report occurrence
  $E[t_{i-m}] \leftarrow E[t_{i-m}] + 1$
  $C \leftarrow C + (E[t_i] >> d) - (E[t_{i-m}] >> d)$
  $E[t_i] \leftarrow E[t_i] - 1$
  $i \leftarrow i + 1$

Alg. 9 presents the main loop of ABS (short for "Approximate Backward for Small alphabets"). The bitvector D holding the counters (or bins) of characters is initialized for each alignment window. D has a field of d bits initially $2^{d-1} - cc(P, c) - 1$ for each character c appearing in P and a joint field for characters not in P. The offset $o[c]$ is used to move the overflow bit of the corresponding counter to the right end of a word. $M[c]$ is a bit mask having one at the least significant bit of the field of character c.

[1] All the algorithms of Section 3.2 were implemented before the appearance of [6]. So we use here bit fields of fixed width. When shared bins are used, the benefit of adaptive width is smaller than without them.

Alg. 9 (The main loop of ABS)
while $i \leq n$ do
  $D \leftarrow I$; $C \leftarrow 0$; $j \leftarrow i - m$
  while $C \leq k$ and $i > j$ do
    $D \leftarrow D + M[t_i]$
    $C \leftarrow C + (D >> o[t_i]) \,\&\, 1$
    $i \leftarrow i - 1$
  if $C \leq k$ then report occurrence
  $i \leftarrow i + m + 1$

ABL (short for "Approximate Forward for Large alphabets") is a slight modification of ABS. If there are not enough bins for all the characters of the pattern, we apply the same sharing technique as in BAMs. Then instead of "report occurrence" on the last but one line of Alg. 9, each match candidate should be verified.

4 Experiments

m	k	Count	Mcount	BAM	BAMs	BAM2a	ABAM	EBL	AFL	ABL
5	0	2.370	1.960	1.183	1.206	0.749	1.420	0.739	1.781	1.482
10	0	2.370	1.960	0.861	0.863	0.297	1.021	0.638	1.778	1.067
20	0	2.376	1.962	0.564	0.582	0.247	0.689	0.544	1.779	0.701
30	0	2.373	1.959	0.449	0.427	0.261	0.514	0.488	1.778	0.514
50	0	2.377	1.958	—	0.301	0.234	—	0.524	1.778	0.413
100	0	2.378	1.964	—	0.204	0.157	—	1.360	1.779	0.360
5	1	2.373	1.960	—	—	—	3.500	—	1.779	4.231
10	1	2.373	1.963	—	—	—	1.844	—	1.783	2.230
20	1	2.377	1.968	—	—	—	1.073	—	1.777	1.257
30	1	2.377	1.961	—	—	—	—	—	1.779	0.978
50	1	2.374	1.960	—	—	—	—	—	1.774	0.780
100	1	2.378	1.961	—	—	—	—	—	1.778	0.736
5	2	2.373	1.961	—	—	—	6.763	—	1.779	9.438
10	2	2.370	1.961	—	—	—	3.372	—	1.777	5.070
20	2	2.376	1.964	—	—	—	1.554	—	1.778	2.510
30	2	2.374	1.966	—	—	—	—	—	1.779	1.944
50	2	2.380	1.960	—	—	—	—	—	1.779	1.582
100	2	2.379	1.964	—	—	—	—	—	1.779	1.596
5	3	2.370	1.964	—	—	—	8.698	—	1.781	14.790
10	3	2.374	1.959	—	—	—	5.840	—	1.780	11.043
20	3	2.376	1.956	—	—	—	—	—	1.779	5.747
30	3	2.376	1.958	—	—	—	—	—	1.780	4.604
50	3	2.374	1.961	—	—	—	—	—	1.779	3.563
100	3	2.379	1.962	—	—	—	—	—	1.779	3.520

Table 1. Execution times of algorithms (in seconds) for English data.

The tests were run on Intel 2.70 GHz i7 processor with 16 GB of memory. All the algorithms were implemented in C and run in the 64-bit mode in the testing framework of Hume and Sunday [14]. Three types of texts were used for testing: English and protein representing large alphabets as well as DNA representing small alphabets.

m	k	Count	Mcount	BAM	BAM2	ABAM	EFS	AFL
5	0	2.724	2.321	3.279	1.559	3.987	1.138	2.150
10	0	2.722	2.326	2.851	1.761	3.511	1.118	2.151
20	0	2.721	2.324	2.419	1.626	3.184	1.118	2.154
30	0	2.722	2.330	2.091	1.430	2.902	1.126	2.159
50	0	2.720	2.324	2.060	1.297	3.074	1.117	2.153
100	0	2.727	2.327	2.240	1.276	3.632	1.111	2.160
5	1	2.723	2.378	—	—	8.250	—	2.154
10	1	2.721	2.326	—	—	7.483	—	2.144
20	1	2.718	2.323	—	—	6.318	—	2.154
30	1	2.719	2.330	—	—	5.204	—	2.160
50	1	2.721	2.323	—	—	4.833	—	2.158
100	1	2.719	2.324	—	—	4.841	—	2.158
5	2	2.720	2.322	—	—	9.907	—	2.146
10	2	2.720	2.324	—	—	11.593	—	2.157
20	2	2.724	2.326	—	—	10.857	—	2.146
30	2	2.723	2.329	—	—	8.836	—	2.159
50	2	2.721	2.323	—	—	7.762	—	2.158
100	2	2.712	2.324	—	—	6.734	—	2.153
5	3	2.727	2.322	—	—	8.638	—	2.154
10	3	2.720	2.324	—	—	14.146	—	2.154
20	3	2.723	2.323	—	—	15.888	—	2.154
30	3	2.712	2.327	—	—	13.558	—	2.159
50	3	2.724	2.322	—	—	11.582	—	2.154
100	3	2.720	2.324	—	—	9.443	—	2.153

Table 2. Execution times of algorithms (in seconds) for DNA data.

The English text is the KJV Bible (3.9 MB), the DNA text is 4.4 MB long and the protein text is 3.6 MB long. All the texts were taken from the Smart corpus [11]. For each text we had six sets of 200 patterns with lengths: $m = 5, 10, 20, 30, 50$, and 100.

As reference methods we used Count and Mcount [16] as well as BAM [6]. Mcount was run only with a single pattern. We have not shown the results of Grossi & Luccio's algorithm [13] because it was clearly slower than Count. Likewise, we did not test GFG [12] and SBA [9], because they were mostly slower than BAM in tests of [6].

Tables 1, 2 and 3 represent the average execution times in seconds for English, DNA, and protein data respectively, for $k = 0, 1, 2$, and 3. The results were obtained as an average of nine runs. The best time for each combination of m and k has been boxed. An empty cell means that 64 bits was not enough to hold the necessary counters at least for one of the 200 patterns.

From the results for English data in Table 1, it can be seen that EBL is fastest for shorter pattern length and BAM2a is fastest for remaining pattern lengths in the exact case ($k = 0$). For $k = 1, 2$, AFL is the fastest for short patterns and ABL for long patterns. As an exception, ABAM is fastest for $k = 1$ and $m = 20$. For $k = 3$, AFL is the fastest. Note that EBL gets its best time for $m = 30$. Its speed is decreasing for longer patterns because longer patterns produce more false matches which increase verification time.

From the results for DNA data in Table 2, it can be seen that EFS is clearly the fastest in the exact case and AFL in the approximate case. EFS works in a double speed when compared with the previous algorithms. Observe also that BAM2 is faster than BAM, even with a wide margin.

m	k	Count	Mcount	BAM	BAMs	BAM2a	ABAM	EBL	AFL	ABL
5	0	1.928	1.596	0.711	0.733	0.581	0.853	0.471	1.451	0.909
10	0	1.932	1.601	0.591	0.611	0.191	0.702	0.481	1.452	0.764
20	0	1.934	1.599	0.427	0.582	0.168	0.521	0.662	1.451	0.582
30	0	1.934	1.592	0.321	0.331	0.196	0.426	1.939	1.451	0.542
50	0	1.934	1.598	—	0.254	0.198	—	—	1.451	0.691
100	0	1.934	1.598	—	0.247	0.197	—	—	1.449	0.231
5	1	1.931	1.599	—	—	—	2.131	—	1.451	2.346
10	1	1.933	1.602	—	—	—	1.233	—	1.452	1.461
20	1	1.931	1.597	—	—	—	0.791	—	1.451	1.071
30	1	1.931	1.602	—	—	—	0.630	—	1.451	1.116
50	1	1.938	1.598	—	—	—	—	—	1.451	1.591
100	1	1.932	1.602	—	—	—	—	—	1.449	3.449
5	2	1.932	1.602	—	—	—	4.351	—	1.454	5.179
10	2	1.929	1.598	—	—	—	2.077	—	1.451	2.826
20	2	1.933	1.606	—	—	—	1.104	—	1.448	2.067
30	2	1.938	1.649	—	—	—	—	—	1.451	2.301
50	2	1.938	1.598	—	—	—	—	—	1.448	3.396
100	2	1.938	1.599	—	—	—	—	—	1.451	3.163
5	3	1.931	1.603	—	—	—	6.466	—	1.454	8.754
10	3	1.931	1.601	—	—	—	3.377	—	1.456	5.907
20	3	1.933	1.598	—	—	—	—	—	1.453	4.338
30	3	1.936	1.599	—	—	—	—	—	1.453	4.756
50	3	1.939	1.601	—	—	—	—	—	1.444	6.737
100	3	1.937	1.598	—	—	—	—	—	1.453	10.737

Table 3. Execution times of algorithms (in seconds) for protein data.

The results in Table 3 for protein data do not differ much from Table 1. EBL was very slow for $m > 30$ (results not shown), because then the number of forbidden characters is low.

The current implementation of ABAM does not contain shared bins. The test results suggest that ABAM with shared bins could be the winner with some new parameter combinations.

For all types of data, Mcount is considerably faster than Count. The obvious reason is that the main loop of Mcount contains only one if statement whereas the main loop of Count contains three. Relatively, the conditional instructions are slow in modern processors.

5 Concluding remarks

We introduced new variations jumbled matching algorithms. All the forward algorithms are clearly linear. The speed of their approximate versions do not depend on the value of k. It is not difficult to show that the backward algorithms are sublinear on average for small k and large m. The experimental results show that our algorithms are competitive with previous solutions. In almost every tested case, the best of our algorithms was faster than any previous solution, and in many cases even doubling the speed of the best previous solution. Especially the technique of shared bins showed to be useful for jumbled matching. We believe that there is still room to improve our results. E.g. more sophisticated character selection for shared bins may lead to faster solutions.

References

1. G. Benson: *Composition alignment*, in Proc. of the 3rd International Workshop on Algorithms in Bioinformatics 2003, pp. 447–461.
2. S. Böcker: *Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt.* Journal of Computational Biology, 11 (6) 2004, pp. 1110–1134.
3. S. Böcker: *Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry.* Bioinformatics, 23 (2) 2007, pp. 5–12.
4. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *Algorithms for jumbled pattern matching in strings.* Int. J. Found. Comput. Sci. 23 (2) 2012, pp. 357–374.
5. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *On approximate jumbled pattern matching in strings.* Theory Comput. Syst. (MST) 50 (1) 2012, pp. 35–51.
6. D. Cantone and S. Faro: *Efficient online Abelian pattern matching in strings by simulating reactive multi-automata*, in J. Holub and J. Žďárek, eds., Proc. PSC 2014, pp. 30–42.
7. F. Cicalese, G. Fici, and Zs. Lipták: *Searching for jumbled patterns in strings*, in J. Holub and J. Žďárek, eds., Proc. PSC 2009, pp. 105–117.
8. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Information Processing Letters 110 (4) 2010, pp. 148–152.
9. E. Ejaz: *Abelian pattern matching in strings.* Ph.D. Thesis, Dortmund University of Technology 2010, `http://d-nb.info/1007019956`.
10. R. Eres, G. M. Landau, and L. Parida: *Permutation pattern discovery in biosequences.* Journal of Computational Biology, 11 (6) 2004, pp. 1050–1060.
11. S. Faro and T. Leqroc: *Smart: string matching algorithms research tool*, 2015, `http://www.dmi.unict.it/~faro/smart/`
12. S. Grabowski, S. Faro, and E. Giaquinta: *String matching with inversions and translocations in linear average time (most of the time).* Information Processing Letters 111 (11) 2011, pp. 516–520.
13. R. Grossi and F. Luccio: *Simple and efficient string matching with k mismatches.* Information Processing Letters 33 (3) 1989, pp. 113–120.
14. A. Hume and D. Sunday: *Fast string searching.* Software–Practice and Experience, 21 (11) 1991, pp. 1221–1248.
15. P. Jokinen, J. Tarhio, and E. Ukkonen: *A comparison of approximate string matching algorithms.* Software–Practice and Experience 26 (12) 1996, pp. 1439–1458.
16. G. Navarro: *Multiple approximate string matching by counting*, in R. Baeza-Yates, ed., Proc. 4th South American Workshop on String Processing 1997, pp. 125–139.
17. G. Navarro and M. Raffinot: *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, New York, NY, 2002.
18. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in Proc. 10th International Symposium on String Processing and Information Retrieval, vol. 2857 of Lecture Notes in Computer Science, 2003, pp. 80–93.
19. A. Salomaa: *Counting (scattered) subwords.* Bulletin of the European Association for Theoretical Computer Science (EATCS) 81, 2003, pp. 165–179.