Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Jumbled Pattern Matching with Prime Factorization

Ismail Habib

ismail.habib@gmail.com

INTI International University and Colleges

Tin Tin Ting

INTI International University and Colleges

Muhammed Kulekci

Miami University

Research Article

Keywords: Jumbled pattern matching, Abelian pattern matching, String algorithms, Prime encoding, Sliding window algorithms, Exact pattern matching, Text indexing

Posted Date: April 27th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9494714/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Jumbled Pattern Matching with Prime Factorization

Ismail Habib[1*], Tin Tin Ting[1] and Muhammed Kulekci[2]

[1*]Information Systems, INTI International University and Colleges Persiaran Perdana BBN, Putra Nilai, 71800 Nilai, Negeri Sembilan, Malaysia.

[2]Computer Science and Software Engineering, Miami University 501 E. Spring Street, Oxford, OH 45056.

*Corresponding author(s). E-mail(s): ismail.habib@gmail.com;
Contributing authors: tintin.ting@newinti.edu.my;
kulekcmo@miamioh.edu;

Abstract

The jumbled pattern matching (JPM) aims to find all possible permutations of a queried string on a target text. Previous studies mainly depend on Parikh Vector and counting alphabet frequencies. In this study, we present a different approach that is based on the fundamental theory of arithmetic, which states that every number greater than 1 is either a prime or can be represented as a product of primes. By changing the domain of the problem from a frequency counting to the prime factorization, we remove some constraints of the previously proposed algorithms such as the size of the query and the extra size required for frequency storage.

The JPM comes in multiple flavors. The *online version* which takes an input query and searches for all occurrences of that query within the search text. The *offline version* which is also known as Indexed Jumbling String Match, relies on pre-processing the string by creating an index over the search text.

The algorithm presented in this study can be used for both searching in the online version and creating an index in the offline version of Jumbled String Match. This algorithm runs in the worst case scenario in $\mathcal{O}(n)$ for searching in the online version and Index Creation in the offline version. It performs Subsequent searches in the offline version in sub $\mathcal{O}(n)$.

We will also show that the algorithm is truly linear across n where n is the number of characters to be processed unlike the previously presented algorithms and it is independent of the size of the query string.

Keywords: Jumbled pattern matching, Abelian pattern matching, String algorithms, Prime encoding, Sliding window algorithms, Exact pattern matching, Text indexing

1 Introduction

Jumble Pattern Matching has been introduced multiple times under many different names. For instance Jumbled Pattern Matching [1, 2], Abelian Pattern Matching[3, 4], and Permutation Matching [5, 6]. Some of the initial publications in this issue date back to 1979 [7] although they discuss variation of the problem which is set equality the core proposed solution remains within the domain of frequency counting. Informally, two strings are jumbled matched if they contain the same character frequencies, regardless of their order

Although this might seem like a normal string search problem the implementations expand over a vast set of core issues. For instance. it can be used the field of Bioinformatics in genome sequencing and composition where it can help identify meaningful sequences in the DNA and extract common sequences within DNA.

Applications also span Machine Learning specifically in pattern matching where different episodes leading to a specific event can be identified. The existence of an efficient algorithm to identify the commonality among patterns can optimize the process of episode finding in machine learning algorithms.

A third area that is effected with the existence of efficient jumbled pattern matching algorithm is the development of mass spectrometers, which in its core uses jumbled pattern matching to match structures to known molecule mass. The existence of such a matching pattern algorithm can help reduce the time required to identify patterns in huge sets of molecules.

In this paper we will discuss a linear time algorithm that solves the Jumbled Pattern Match problem which runs in $\mathcal{O}(n)$, by looking at two variations of the problem the online version which can be thought of as a stream of text being matched for a certain query, and the offline version which preprocesses the provided text and stores an index of locations that can enhance later queries on the text.

We will provide an algorithm for the online Jumbled Pattern Matching that runs in $\mathcal{O}(n)$

The same algorithm can be used to construct the index for the offline version, however it runs in $\mathcal{O}(n^2)$, for subsequent requests in $\mathcal{O}(1)$

It was proven that the minimum complexity bound is $\Omega(n^{2-\delta})$ for index creating and $\Omega(n^{2-\epsilon})$, where $\epsilon, \delta > 0$[8].

In Section 2 two of this paper we begin with proper definition of the Jumbled Pattern Matching and explain the notation used and the basic building blocks of the algorithm.In Section 3 we provide a survey of the solutions space along with a comparison of the different algorithms developed. The algorithm with different Online and Offline flavours are presented in Section 4. In section 5 we propose the experimental results of implementing this algorithm. Some further enhancements and work is presented in Section 6. Section 7 wraps up the summary of the paper and a conclusion.

2 Preliminaries and Primitives

let alphabet be $\Sigma = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_\sigma\}$
let σ be the cardinality $|\Sigma|$
Let $\mathcal{S} = s_1 s_2 \ldots s_n$ be the characters of the string drawn from the alphabet Σ
let n be the size of the string $|\mathcal{S}|$.
let Q be the query.
let m be the size of the query $|Q|$
let U be a sub-string of $\mathcal{S}$ of size m
Q Jumble matches a sub-string of $\mathcal{S}, U$, can be defined as

Definition 1 $\forall \epsilon_u \in U, \epsilon_q \in Q \{Q \text{ jumble matches } U \rightarrow |U| = |Q| \bigwedge |\epsilon_u| = |\epsilon_q| \bigwedge \epsilon_u = \epsilon_q\}$

It is clear from the above definition 1 that if the frequencies of ϵ's are equal in both string we have a jumble match and hence most of the algorithms proposed to date rely on this concept and implement a variation of the Parikh Vector.

However with a slight variation of the original problem replacing the characters with prime numbers we could reconstruct the solution in a different way that will be much more efficient to compute and results in faster more consistent results.

2.1 Hashing

let $\prod_p = \{p_1, p_2, \ldots p_\sigma\}$ be a set of first σ prime numbers.
Let $G(x)$ be a bijection function that maps each ϵ_i to p_i
let $F(p_1, p_2.., p_i)$ be the function to compute the hash of p_i where p is a prime number.

Definition 2 $F(G(\epsilon_i), G(\epsilon_j)...G(\epsilon_m)) = \prod_{i=1}^{m} G(\epsilon_i)$

Hence Definition 1 can be simplified as

Definition 3 $\{Q \text{ jumble matches } U \rightarrow F(G(U)) = F(G(Q))\}$

2.2 Query

It is clear that in order to query a string in the Online Version of the Jumbling Pattern Matching all that needs to be done is to calculate the product of the characters prime mapping in the query string, $F(G(Q))$ and match it with the characters prime mapping in U, F(G(U)), sliding U over the string S

As for the offline scenario the index is pre-processed by sliding U over the string S generating an Index, hash-table, then any subsequent query has the following three scenarios for subsequent queries.

1) $|Q| = |U|$: To match the indexes all you need to do is find query the hash value directly. This will clearly run in $\mathcal{O}(1)$

2) $|Q| \leq |U|$: To match this case all you need to do is pass over the indices in Hash Table calculating the mod $|U|$. If the mod = 0 then only check the specified indices specified in these Hash Table entries for jumbling pattern match. This will clearly run in sub $\mathcal{O}(n)$

3) $|Q| \geq |U|$: To match this case all you need to do is pass over the indices in Hash Table calculating the mod $|Q|$. If the mod = 0 then only check the specified indices specified in these Hash Table entries for jumbling pattern match. This will clearly run in sub $\mathcal{O}(n)$

A work around to the above procedure to ensure that all subsequent query will be $\mathcal{O}(1)$ is to generate the index for all possible length of $|Q|$ which will cause the pre-processing time to increase to $\mathcal{O}(n^2)$

3 Current Existing Algorithms

Most of the algorithms that currently exist in the field exist focus on the the frequency analysis of the characters through a sliding window over the text, and hence they usually result in a dependency on the size of the sliding window which is the currently the cardinality of the Query window. These algorithms have been classified in [9] as earlier and new methods for Jumbled Pattern Matching. The core difference between the classification here is the dependency on the counting method of the sliding window for the earlier algorithm while the latter method which depend on bit parallelism for achieving optimization in the window.

Another classification that can also be considered is the Exact matching verses estimated pattern match. In the earlier an exact frequency count is needed to achieve a jumble hit while the latter defines a threshold that if met will result in a jumble match.

Another classification that is highly considered is the area of Jumbled Matching is the bit searching verses a finite set of alphabet. In this specific classification the core focus or interest is how the algorithm perceive the input. In the bit search the input is considered as a bit stream for both the query string and input text while the latter focuses on searching the string on building an index over a finite set of alphabet. Therefore the alphabet space has to be previously known.

In our study of the solution space here we will focus on the classification presented in [9].

3.1 Earlier Methods

As we have mentioned previously the traditional methods mainly focus on a sliding window approach that performs a frequency counting matching the frequency count against a frequency count of a sliding window across the input text. Further classifications exist which depend mainly on the direction of analysis within the sliding window, i.e. left to right or right to left analysis. The core reason for this distinction is to give the algorithm a boost in speed while jumping ahead the sliding window in the search text. It is important to note that this trick is effective, however only provides a marginal enhancement. Hence this distinction will not be considered in our review.

3.1.1 Distribution of Character

[10] Originally this paper was presented for approximate string match however this algorithm can be used for Jumbled pattern match as well. The core Idea of the algorithm is depended on pre-processing the count of each character is stored in a Queue. The forward sliding window performs a count of each character in the window resulting in an array of the frequency count of the sliding window. In the Pattern matching phase, which will be called the search phase here after, every time a character is encountered it is enqueued and the count is decremented from the array. Whenever the count of a character in the array reaches 0 the queue is processed by dequeueing the element from the queue if at the end of the processing of matched elements the queue size is within the range of the defined threshold k then a matched is returned.

It is clear to see that this algorithm runs in sub $\mathcal{O}(mn)$, where m is the size of the window size. Since the first loop will need to traverse the whole text and each time a suspected pattern match is reached the sub-string is traversed for pattern match.

3.1.2 Count

[11] This Algorithm is another approximation algorithm that measures the distance between the query string and the sliding window. Unlike the Distribution of Character where the window used can be larger than m in Count algorithm only windows of query size are considered. It works by pre-processing and counting all the elements in the query string. It also creates a variable which includes the negation of the number of elements minus the error threshold $count = -(|Q| - k)$ where k is the accepted mismatches. The algorithm proceeds by incrementing the value of *count* whenever the value of count exceeds 0 that indicates that the number of matches between the query Q and the sub-string U has been met and hence returns an occurrence.

The algorithm clearly runs in $\mathcal{O}(n)$ since the frequencies are consolidated into a single variable of *count* which is compared instead of having to traverse the whole sub-string for a pattern match whenever a pattern match is suspected.

3.1.3 MCount

[11] This was presented in the same research as the count algorithm. The concept is very similar to the above Count algorithm however the counting is not handled through an array it is handled through maintaining a bit vector associated with each character. The count of the character is stored in this bit vector. The most significant bit within the bit vector is used to identify if the number of characters match the number within the query string. If an overflow occurs then then the number of characters have exceeded the *count* variable described in the Count section above. The core idea of the algorithm is similar to the count specified above however the *count* variable is maintained within the bit vector, this helps reduce the number of comparisons required providing a marginal enhancement over the Count algorithm however introduces an extra space complexity of $log|\Sigma|$ bits.

Since there is no major difference this algorithm also runs in $\mathcal{O}(n)$.

3.1.4 Prefix Based Algorithm

[3] This algorithm adds a enhancement over the frequency count to avoid the dependency on the size of the query. This is achieved by avoiding the comparison of each frequency distribution of the sliding window and only comparing the first and last character in the sliding window to avoid the full traversal of the query window. This is a slight variation however it achieved wonderful results by eliminating the dependency over the size of the query window size. This reduces the number of comparisons required and also removes the necessity of maintaining a bit vector as in the Mcount algorithm.

The result is $\mathcal{O}(n)$ without any dependency on the query size, except the first window, and eliminates the extra space required with the MCount algorithm.

3.1.5 Bit-parallel Abelian Matcher

[12] This algorithm similar to the MCount maintains a bit vector associated with the characters in the query string. It also maintains an overflow bit as the MCount however it introduces a variation which allows it to add a further enhancement which is every time an overflow occurs this is an indication that a certain character will result in a mismatch hence it jumps ahead removing all the characters before the next occurrence of the overflowed character. This enhancement allows it to jump multiple steps at a time causing the string traversal to be smaller than the size of the string however there is the same inheritance of the extra space required.

However in order to check for a match both the query frequency count and the window character count has to be checked. This results in $\mathcal{O}(nm)$ where m is the size of the window.

This algorithm introduces the core idea required to one of the practically fastest known algorithms BAM2 which will be introduced later.

3.1.6 Suffix Based Algorithm

[3] Works quite similarly to the above algorithm it does however traverse the window right to left. This variation introduces an enhancement over BAM. It allows us to avoid the processes of counting the number of characters that are needed to be jumped every time an overflow occurs. In essence as soon as an overflow occurs we just need to shift the window directly to the position of the overflow. This location is lost within BAM since the jump position is not the same position as the overflow and hence a look for the first occurrence of the overflowed character has to be researched when an overflow occurs.

Just as the BAM the order of the algorithm is $\mathcal{O}(nm)$.

3.2 New Methods

3.2.1 BAM2

[13] Like the BAM algorithm it creates a bit vector and depends on the overflow of the most significant bit, however it adds a trick to the bit vector that is it stores the occurrences of each two consecutive characters. The window is traversed two characters

at a time. causing the sliding window to be traversed two bits at a time reducing the practical time required however it does introduce the practical complexity of having to handled odd and even size windows independently. That being said the whole input string will have to be traversed and a match will need to traverse a factor of the window size hence the complexity of the algorithm is $\mathcal{O}(nm)$.

3.2.2 Exact Backward for Large alphabets - EBL

[13] This is not the fastest algorithm however it is the most efficient algorithm that can handle large query sizes. It uses the flexibility of frequency count to compare both the search string and query string however before performing the frequency analysis it maintains a bit vector which stores the existence of a character in a query string. This bit vector is compared with the last two characters before performing the frequency analysis if one of the last two characters does not exist the window is advanced the size of the query string.

The bit vector that stores only the existence of characters only introduces a marginal increase in the bit vector. The increase is the size of the alphabet in bits. The complexity just as BAM2 is also $\mathcal{O}(nm)$, however, the 2-gram processing of BAM2 causes it to be practically more efficient that

3.3 Offline Methods

When it comes to the offline version of the Jumbled Pattern search the focus is not on the efficiency of the algorithm for processing however the focus is mainly on the optimization of subsequent queries and compactness of the index[14–19].

Multiple algorithms have been proposed however non of them have exceeded the $\mathcal{O}(n)$ pre-processing time and most of the algorithm rely on effectively storing the Parakh Vector or Frequency count in Arrays, Abelian Trees - which could be considered as Arrays with pointers to objects, hashtables, suffix tress, B-Trees, tier etc.

For a constant $\mathcal{O}(1)$ query time the algorithms require pre-processing time of $\mathcal{O}(n^2)$

It is important to note that [8] has provided the lower bounds possible for query time through a reduction based on the hardness of the 3SUM, which requires $\Omega(n^2-\epsilon)$.

The proposed algorithm does not not perform better in terms of performance for the pre-processing and the constraints specified in [8] still hold however, the stored information is all in the form of Numbers no association with the letter frequencies is required hence through some compression scheme the generated index can achieve low storage requirements. Further work is required in this area.

4 Algorithm

4.1 Online Version

Algorithm 1 Online Jumbled Pattern Matching using Prime Encoding

Require: Text $S[1..n]$, Pattern $Q[1..m]$, Prime mapping π
Ensure: List of starting indices where Q jumble-matches S

```
1:  hq ← 1
2:  for j = 1 to m do
3:      hq ← hq · π(Q[j])
4:  end for
5:  h ← 1
6:  for j = 1 to m do
7:      h ← h · π(S[j])
8:  end for
9:  ilist ← ∅
10: if h = hq then
11:     append 1 to ilist
12: end if
13: for i = m + 1 to n do
14:     h ← h / π(S[i−m])
15:     h ← h · π(S[i])
        ‘‘‘
16:     if h = hq then
17:         append (i − m + 1) to ilist
18:     end if’’’
19: end for
20: return ilist
```

The above algorithm takes three inputs The text to search in S, the query string Q that needs to be matched and a list of Prime Numbers P. P is the size of P is equal to the size of the alphabet Σ. In order to avoid the dependency on the Query size $|Q|$ we only calculate the first set of characters of the size of Q in S. In all the subsequent window traversal due to the nature of the problem domain we can achieve the same result by dividing out the first character of the previous window and multiply with the prime value of the newly introduced character.

In order to clarify this let us take a small example.

Let S = ablewasIereIsawelba

Let P = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101;

Let Q = was = 83*2*67 = 11,122

So we will be looking for values of 11,122

The table 1 below walks through the both while loops that traverse the string

4.2 Analysis of Algorithm - Online Version

From the above algorithm we see that we have two loops the first loop executes one multiplication step the size of the query $\mid Q \mid$. The second loop executes 3 steps the size of the input string $\mid S \mid$ - the size of the query $\mid Q \mid$ hence the total cost of this algorithm is $\mid Q \mid + \mid S \mid - \mid Q \mid$ which reduces to $\mathcal{O}(n)$.

4.3 Offline Version

There is no major difference between the Offline Algorithm and the online algorithm except in storing the results inside a Chained Hash Table. This hash table can then be used for sub $\mathcal{O}(n)$ latter queries.

If later queries are required to run in $\mathcal{O}(1)$ we can pre-process the whole Input S in order of $\mathcal{O}(n^2)$ by calculating all the possible Query Sizes over the length of $|S|$ and store the hashtables hence any subsequent query would run in $\mathcal{O}(1)$.

4.4 Analysis Of Algorithm - Offline Version

It is clear from the above that if the query size matches the index size the algorithm will execute in the same manner as the online version i.e in $\mathcal{O}(n)$.

The case that requires analysis is when the Index size does not match the query size. In such a scenario we have two cases:

i) When the Query Size is larger than the index size.

ii) When the Query size is smaller than the index size.

It is clear that both these cases have the same complexity and hence without the loss of generality it suffices to analyze only one case.

Window	Calculation	Value	Match
abl	2*3*37	222	No
ble	222/2*11	1221	No
lew	1221/3*83	33781	No
ewa	33781/37*2	1826	No
was	1826/11*67	11122	Yes
asl	11122/83*37	4958	No
sle	4958/2*11	27269	No
ler	27269/67*61	24827	No
ere	24827/37*11	7381	No
rel	7381/11*37	24827	No
els	24827/61*67	27269	No
lsa	27269/11*2	4958	No
saw	4958/61*83	11122	Yes
awe	11122/67*37	1826	No
wel	1826/2*61	33781	No
elb	33781/2*3	1221	No
lba	1221/37*2	222	No

Table 1 Algorithm Walkthrough

Algorithm 2 Offline Jumbled Pattern Index Search

Require: Query Q, hashIndex
Ensure: List of matching indices

```
hq ← 1
for c ∈ Q do
    hq ← hq · π(c)
end for
ilist ← ∅
for each (h, positions) in hashIndex do
    if h = hq then
        for each index in positions do
            append index to ilist
        end for
    end if
end for
return ilist
```

Algorithm 3 Offline Jumbled Pattern Index Search

Require: Query Q, index $hashIndex$, pattern length m
Ensure: List of matching indices

```
hq ← 1
for j = 1 to m do
    hq ← hq · π(Q[j])
end for
ilist ← ∅
if hq ∈ hashIndex then
    for each i in hashIndex[hq] do
        append i to ilist
    end for
end if
return ilist
```

5 Experimental Results

In this paper we have performed the experiments on the pizzachili corpus we have concatenated all the files in the corpus and for larger files we have concatenated the files with themselfs http://pizzachili.dcc.uchile.cl/

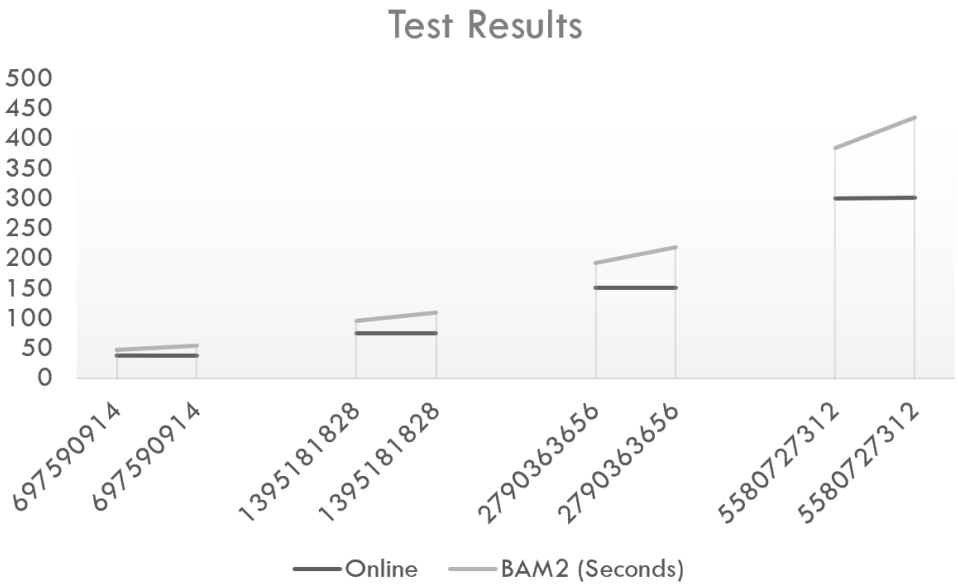
The results of the files can be found below

It is clear from the plotted graph that the proposed algorithm remains constant independent of the size of the query and that can be noticed from the horizontal plotting of each size independent of the query size.

n	Filesize	Online Version (Seconds)	BAM2 (Seconds)	Query Size
697590914	697.5 Megabytes	37.613653	47.88711	6
697590914	697.5 Megabytes	37.5331	54.43521	10
1395181828	1.39 Gigabytes	75.188637	95.77623	6
1395181828	1.39 Gigabytes	75.199969	109.2149	10
2790363656	2.7 Gigabytes	150.406048	191.8442	6
2790363656	2.7 Gigabytes	150.455019	217.7972	10
5580727312	5.5 Gigabytes	299.178479	383.4875	6
5580727312	5.5 Gigabytes	300.883874	435.5323	10

Table 2 Experimental Results

Fig. 1 Experimental results



Another important observation is that the increase across sizes is linear the difference in doubling the size of the input n directly doubles the time required to finalize the processing.

6 Future Work

Although the proposed algorithm does provide significant enhancements in time, it is important to know that there are a few issues that will require detailed studies.

This algorithm depends on products of prime numbers which can grow significantly fast, this limits the size of the query string that can be used. One quick workaround would be to rearrange the prime numbers based on the distribution of frequencies.

The average difference in results needs to be further investigated to ensure that this will yield to promising results.

If however larger query sizes are required a work around could be to use large number libraries. It is important to note though that the fastest large number algorithm for multiplication runs in $\mathcal{O}(n^1.4)$ where n in this case is the number of bits of the products. This needs to be analyzed to measure the effect of such complication on the overall performance of the algorithm.

Another area which needs further investigation is the dynamic alphabet. This situation occurs when we do not know the size of alphabet before hand. There have been multiple algorithms to generate prime numbers and some of these algorithm run in $\mathcal{O}(n)$. This in essence means. Although the n does not refer to the the size of the alphabet which larger alphabet sizes this might cause the pure linearity of the algorithm causing the algorithm to run in $\mathcal{O}(nm)$ where m is the size of the alphabet. This needs further investigation and analysis.

It would also be good if such an algorithm can be tested in practical application such as gnome sequencing and alignment to measure the enhancement in performance over current traditional methods.

7 Conclusion

In this paper we have presented a novel approach to solve the Jumbled Pattern matching problem, instead of focusing on frequency and counting methods that currently exist we show that by a slight modification of the domain where we are solving the problem from the frequency domain to the prime factorization domain we can achieve significantly better results, which are linear across the size of the input and independent across the query size.

Funding

The authors did not receive any specific funding for this work.

References

[1] Burcsi, P., Cicalese, F., Fici, G., Lipták, Z.: On approximate jumbled pattern matching in strings. Theory of Computing Systems **50**(1), 35–51 (2012)

[2] Cicalese, F., Fici, G., Lipták, Z.: Searching for jumbled patterns in strings. In: Stringology, pp. 105–117 (2009)

[3] Ejaz, T.: Abelian pattern matching in strings. PhD thesis, Dortmund University of Technology (2010)

[4] Ejaz, T., Rahmann, S., Stoye, J.: Online abelian pattern matching (2008)

[5] Moosa, T.M., Rahman, M.S.: Sub-quadratic time and linear space data structures for permutation matching in binary strings. Journal of Discrete Algorithms **10**, 5–9 (2012)

[6] Kubica, M., Kulczyński, T., Radoszewski, J., Rytter, W., Waleń, T.: A linear time algorithm for consecutive permutation pattern matching. Information Processing Letters **113**(12), 430–433 (2013)

[7] Hvllot, J.: Associative commutative pattern matching. In: Proceedings of the 6th International Joint Conference on Artificial intelligence-Volume 1, pp. 406–412 (1979)

[8] Amir, A., Chan, T.M., Lewenstein, M., Lewenstein, N.: On hardness of jumbled indexing. In: International Colloquium on Automata, Languages, and Programming, pp. 114–125 (2014). Springer

[9] Ghuman, S.S., Tarhio, J., Chhabra, T.: Improved online algorithms for jumbled matching. Discrete Applied Mathematics (2018)

[10] Grossi, R., Luccio, F.: Simple and efficient string matching with k mismatches. Information Processing Letters **33**(3), 113–120 (1989)

[11] Navarro12, G.: Multiple approximate string matching by counting. In: Fourth South American Workshop on String Processing (WSP 1997), vol. 8, p. 125 (1997). McGill-Queen's Press-MQUP

[12] Cantone, D., Faro, S.: Efficient online abelian pattern matching in strings by simulating reactive multi-automata. In: Stringology, pp. 30–42 (2014)

[13] Chhabra, T., Ghuman, S.S., Tarhio, J.: Tuning algorithms for jumbled matching. In: Stringology, pp. 57–66 (2015)

[14] He, M., Munro, J.I., Rao, S.S.: A categorization theorem on suffix arrays with applications to space efficient text indexes. In: SODA, vol. 5, pp. 23–32 (2005)

[15] Christodoulakis, M., Christou, M.: Abelian concepts in strings: a review. Festschrift for Borivoj Melichar, 19–45 (2012)

[16] Gagie, T., Hermelin, D., Landau, G.M., Weimann, O.: Binary jumbled pattern matching on trees and tree-like structures. Algorithmica **73**(3), 571–588 (2015)

[17] Amir, A., Butman, A., Porat, E.: On the relationship between histogram indexing and block-mass indexing. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **372**(2016), 20130132 (2014)

[18] Badkobeh, G., Fici, G., Kroon, S., Lipták, Z.: Binary jumbled string matching for highly run-length compressible texts. Information Processing Letters **113**(17), 604–608 (2013)

[19] Burcsi, P., Cicalese, F., Fici, G., Lipták, Z.: On table arrangements, scrabble freaks, and jumbled pattern matching. In: International Conference on Fun with Algorithms, pp. 89–101 (2010). Springer