Efficient Algorithm for δ - Approximate Jumbled Pattern Matching

Iván Castellanos and Yoan Pinzón

Faculty of Engineering,
National University of Colombia,
Bogotá, Colombia
{iycastellanosm, ypinzon}@unal.edu.co

Abstract. The Jumbled Pattern Matching problem consists on finding substrings which can be permuted to be equal to a given pattern. Similarly the δ - Approximate Jumbled Pattern Matching problem asks for substrings equivalent to a permutation of the given pattern, but allowing a vector of possible errors δ. Here we provide a new efficient solution for the δ - Approximate Jumbled Pattern Matching problem using indexing tables and bit vectors which, according to the experimental results, gives a speed up about $1.5 - 3.5$ times faster than the solution based on Wavelet trees. This speed up depends mainly of the size of the alphabet. Further there are presented some solutions to another problems related to δ - Approximate Jumbled Pattern Matching, as the All Matching problem, where it is necessary to calculate all the occurrences of a given pattern allowing an error in the text, or the Min-Error problem, where the objective is to find the occurrences which are closer to the pattern.

Keywords: Parikh vectors, jumbled pattern matching, bit vectors, bit parallelism

1 Introduction

Stringology or String Matching is one of the most widely studied problems in computer science, due to the extensive many applications where this problem is used. The main idea of this problem is to find patterns in a text. However, there are many different versions of the original String Matching Problem. In this paper we study one of these versions.

The problem of Jumbled Pattern Matching, also known as Parikh [11], Abelian [8] or Permutation Matching [13], was firstly introduced at [14]. This is a variant of the Pattern Matching problem, where instead of looking in a text at a substring identical to the given pattern, the main interest is to find a substring which has a permutation identical to the given pattern [7].

The Jumbled Pattern Matching can be used to solve different problems in bioinformatics, such as alignment [2], interpretation of mass spectometry data [3], SPN discovery [4], gene clusters, repeated pattern discovery, scrabble and table arragement problems [5], to name some. This variant of Pattern Matching has also been used as a filtering step in the approximate Pattern Matching algorithms.

The Jumbled Pattern Matching problem has been already studied through different versions, which can be applied to the same applications of the original Jumbled Pattern Matching. In [6,12] an approximate version of the problem is considered, here what it is important is to calculate the maximal occurrences in a text between two bound queries, each of them in the form of a vector known as Parikh.

More formally, the Parikh vector of a string s with characters from a finite ordered alphabet Σ, denoted $p(s) = (p_1, \ldots, p_\sigma)$, is defined as the vector of frequencies from

Iván Castellanos, Yoan Pinzón: Efficient Algorithm for δ - Approximate Jumbled Pattern Matching, pp. 47–56.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

each character of Σ in s. For the problem of Exact Jumbled Pattern Matching the target is to find either every substring s' of s such that $p(s')$ is equal to a given Parikh vector q (occurrence problem) or if there is at least one substring s' of s such that $p(s')$ is equal to a given Parikh vector q (decision problem).

For the approximate version of the Jumbled String Matching we need in addition to the Parikh vector q a vector of possible errors δ. The occurrence problem from the δ - Approximate Jumbled Pattern Matching consists in finding all the matchings of a pattern q in the text s such that the absolute difference between the occurrence and the pattern is not bigger than the error δ, i.e. (i, j) is a match if for the substring $s' = s_i \cdots s_j$, $|p(s') - q| \leq \delta$. Similarly the decision version of this problem consists in deciding whether q occurs in s allowing the error δ.

Example 1. Consider the alphabet $\Sigma = \{a, b, c\}$, the string $s = ccabbcbaaccbbaaccbabab$ and the query $q = \{3, 1, 3\}$ with $\delta = \{1, 1, 1\}$. It has 5 maximal occurrences, namely $(5, 11)$, $(6, 12)$, $(8, 17)$, $(13, 19)$, and $(14, 21)$ with errors 2, 2, 3, 2 and 3 respectively. in Figure 1 it is showed this example, and the substrings whose corresponding Parikh vector is a maximal match.

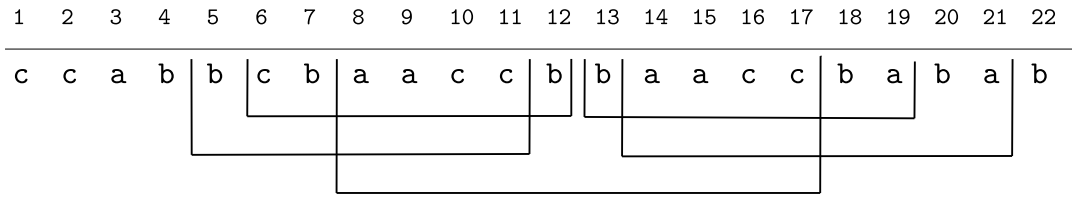

Figure 1. Maximal ocurrences of the query $q = \{3, 1, 3\}$ with $\delta = \{1, 1, 1\}$ in the text *ccabbcbaaccbbaaccbabab*.

Our main contribution is to provide an efficient solution for the δ - Approximate Jumbled Pattern Matching problem. We use different data structures and a new approach based on bit vectors, producing a speed up on previous solutions. Also there are presented some adaptions of the algorithm for another problems that fit in the category of approximate jumbled pattern matching and that to the best of our knowledge, they were not defined before.

The rest of the paper is organized as follows, in section 2 we give some basic definitions and an overview of the used algorithm. In section 3 we present a new implementation of this algorithm, follow by some experimental results in section 4. Finally, the paper is closed with some concluding marks and future work in section 5.

2 Preliminaries

Given a finite ordered alphabet Σ with σ elements, i.e. $\Sigma = \{a_1, \ldots, a_\sigma\}$, $a_1 < \cdots < a_\sigma$ and a string $s \in \Sigma^*$ of length $|s| = n$, i.e., $s = s_1 \cdots s_n$. The Parikh vector of s denoted $p(s) = (p_1, \ldots, p_\sigma)$ counts the multiplicity from each character of Σ in s, i.e. $p_i = |\{j \mid s_j = a_i\}|$ for $i = 1, \ldots, \sigma$, additionally, by $pr(s, i)$ we denote the prefix of s until position i inclusive, i.e. $pr(s, i) = s_1, \ldots, s_i$. If s is clear, it can be used just $pr(i)$, also we represent $p(pr(s, i))$ or $p(pr(i))$ by $prv(s, i)$ or $prv(i)$ respectively. By $s[i, j] = s_i \cdots s_j$ we denote the substring of s from i to j inclusive for $1 \leq i \leq j \leq n$, note that $p(s[i, j]) = prv(j) - prv(i - 1)$.

For a Parikh vector $q \in \mathbb{N}^\sigma$, where $\mathbb{N}$ denotes the set of positive integers including zero, and let $|q| := \sum_{i=1,\ldots,\sigma} q_i$ denote the length of q. It is said that (i, j) is an occurrence of q in s if and only if $p(s[i, j]) = q$. By convention, it is said that the empty string ε occurs in each string once. The problem of deciding whether q occurs in s, known as decision problem, or finding all the occurrences of q in s, known as occurrence problem, is called Jumbled Pattern Matching (JPM) [9].

For two Parikh vectors $p, q \in \mathbb{N}^\sigma$, the binary operations $p \leq q$ and $p + q$ are defined component-wise, i.e. $p \leq q$ if and only if $p_i \leq q_i$ for all $i = 1, \ldots, \sigma$, and $p + q = u$ where $u_i = p_i + q_i$ for $i = 1, \ldots, \sigma$ respectively. Similarly, if $p \geq q$, we set $q - p = v$ where $v_i = q_i - p_i$ for $i = 1, \ldots, \sigma$. Note that for two Parikh vectors p and q it is possible that neither $p \leq q$ nor $q \leq p$. Finally, if the Parikh vector p is greater or equal than the Parikh vector q, it is said that p is a super-Parikh vector of q or also that q is sub-Parikh vector of p.

Let $s \in \Sigma^*$ a text and $u, v \in \mathbb{N}^\sigma$ a pair of given Parikh vectors with $|s| = n$ and $u \leq v$. u, v are called the query bounds. The problem of finding all maximal occurrences in s of some Parikh vector q such that $u \leq q \leq v$ is one version of what is refered to Approximmate Jumbled Pattern Matching (AJPM). An occurrence (i, j) of q is maximal (w.r.t u and v) if neither $(i - 1, j)$ nor $(i, j + 1)$ is an occurrence of some Parikh vector q' such that $u \leq q' \leq v$. The decision version of the problem is where we only want to know whether some q occurs in s satisfying the bounds $u \leq q \leq v$. In the rest of this paper the lower bound u will be denoted as $q - \delta$ and the upper bound v as $q + \delta$.

In both, the exact and the approximate problem, it is possible to determine if there are occurrences of one query in $O(n)$ time using a window approach. However, usually it is of more importance to find occurrences in a text for many queries. Because of that it is necessary to make a preprocessing of the text, in the following sections we assume that K many queries arrive over time. Obviously, all sub-Parikh vectors of s can be precomputed, then stored them (sorted, e.g, lexicograpically) and when a query arrives, binary search can be done to find the occurrences in the text. In this case, preprocessing time is $\Theta(n^2 \log n)$, because the number of Parikh vectors of s is at most $\binom{n}{2} = O(n^2)$ and there are nontrivial strings with quadratic number of Parikh vectors over arbitrary alphabets. With this preprocessing the time for each query is $O(\log n)$ for the decision problem and $O(\log n + M)$ for the occurrence problem where M is the number of occurrences of the query. On the other hand, the storage space is $\Theta(n^2)$ and this is inacceptable in many applications.

2.1 ESR Algorithm

In [6] were introduced the concepts of Expansion, Shrinkage and Refining to move two pointers L and R, our solution is based on this approach. The pointers L and R represent a window pointing the potential positions $i - 1$ and j where it can be found an occurence, clearly these pointers can be moved linearly for each position in the text, however, this is not optimal. The algorithm instead, updates these pointers in jumps, alternating between updates of R and L, in a manner such that many positions are skipped. In addition, because of the way we update the pointers, we can ensure that, every time we have a maximal occurrence (i, j) the pointers will have the values $i - 1$ and j.

For the Expansion phase R is moved, extending the window to the right until its corresponding Parikh vector is a super-Parikh vector of $q - \delta$. At the end of this

phase, it can be possible that the Parikh vector of the substring contained in the window does not satisfy the bounds because is not a sub-Parikh vector of the upper bound $q + \delta$. Consequently, we switch to the Shrinkage phase. For this phase L is moved, shrinking the window from the left until its corresponding Parikh vector is a sub-Parikh vector of $q + \delta$.

After this it might be that the Parikh vector of the string contained in the window is not a super-Parikh vector of $q - \delta$ anymore. In this case, we need to start a new cycle with an Expansion phase.

On the other hand, if after the Shrinkage phase the Parikh vector corresponding to the window is still a super-Parikh vector of $q - \delta$, then the window with positions (L, R) satisfies the condition $|p(s[L + 1, R]) - q| \leq \delta$, i.e., it is a match, although it is not necessarily maximal. In order to make this occurrence a maximal one, it is necessary to enter the Refining phase moving again R, extending the window to the right as long as the Parikh vector of the window is a sub-Parikh vector of $q - \delta$. After this, a match of the query is kept and it is maximal, because if we extend the window to the left or to the right it is not a match anymore. Finally, after reporting the found occurrence, the process is restarted by expanding the window to the right by one character and entering a new cycle starting with a Shrinkage phase.

More formally, we can express these functions as

$$Expand(k, v) := \min\{j \,|\, prv(j) \geq prv(k) + v\} \tag{1}$$

$$Shrink(k, v) := \min\{j \,|\, prv(k) - prv(j) \leq v\} \tag{2}$$

$$Refine(k, v) := \max\{j \,|\, prv(j) - prv(k) \leq v\} \tag{3}$$

In Figure 2 is showed an example of the ESR Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Expand(0,{2,0,2})	c	c	a	b	b	c	b	a	a	c	c	b	b	a
Shrink(8,{4,2,4})	c	c	a	b	b	c	b	a	a	c	c	b	b	a
Expand(5,{2,0,2})	c	c	a	b	b	c	b	a	a	c	c	b	b	a
Refine(5,{4,2,4})	c	c	a	b	b	c	b	a	a	c	c	b	b	a

Figure 2. Results of the first 4 operations using the ESR Algorithm for query $q = \{3, 1, 3\}$ with $\delta = \{1, 1, 1\}$ in the text *ccabbcbaaccbba*.

During the algorithm are used $Expand(L, q-\delta)$, $Shrink(R, q+\delta)$ and $Refine(L, q+\delta)$. In [6] the ESR algorithm is defined and it is mentioned that the complexity of this algorithm depends on how the functions *prv*, *Expand*, *Shrink*, *Refine* and some comparations are implemented and on the number of times these functions are executed or the number of times we update the pointers R and L.

3 Our Work

In the following section, we show a new implementation of the ESR algorithm, using different data structures to those used previously in the literature [6] to solve the

problem of δ - Approximate Jumbled Pattern Matching, namely Wavelet trees. Firstly, $prv(s,\, i)$ is stored in vectors, making this takes linear space on memory and also can be calculated linearly in time, initially $prv(0) = (0_1, \ldots, 0_\sigma)$ and then for $i = 1, \ldots, n$ and $j = 1, \ldots, \sigma$ we have:

$$prv_j(i) = \begin{cases} prv_j(i-1) & s[i] \neq a_j \\ prv_j(i-1) + 1 & s[i] = a_j \end{cases} \tag{4}$$

Clearly, having this table $prv(i)$ can be calculated during the algorithm in constant time.

Theorem 1. *Given a Parikh vector v and a position k we have that for the text s:*

$$Expand(k,\, v) := \max\{i_j \,|\, i_j \, pos \, of \, the \, (v_j + prv_j(k))'th \, occurrence \, of \, j\} \tag{5}$$

$$Shrink(k,\, v) := \max\{i_j \,|\, i_j \, pos \, of \, the \, (prv_j(k) - v_j)'th \, occurrence \, of \, j\} \tag{6}$$

$$Refine(k,\, v) := \min\{i_j \,|\, i_j \, pos \, of \, the \, (v_j + prv_j(k) + 1)'th \, occurrence \, of \, j\} - 1 \tag{7}$$

Proof. To show the equivalence between (1) and (5), first set $R' = \max\{i_j \,|\, i_j \, pos \, of \, the \, (v_j + prv_j(k))'th \, occurrence \, of \, j\}$, then $prv(R') \geq prv(k) + v$, because for every $j = 1, \ldots, \sigma$ we have that $prv_j(R') \leq v_j + prv_j(k)$ due to the definition of R', now without loss of generality assume that $R' = i_1$, then taking any $R'' < R'$ we have that $prv(R'') \not\leq v + prv(k)$ because $prv_1(R'') \not\leq v_1 + prv_1(k)$. So it can be concluded that $R' = \min\{j \,|\, prv(j) \geq prv(k) + v\}$. Analogously the equivalence between (2) and (6) can be proved.

To show the last equivalence, set $R' = \min\{i_j \,|\, i_j \, pos \, of \, the \, (v_j + prv_j(k) + 1)'th \, occurrence \, of \, j\}$, without loss of generality taking $R' = i_1$ then $prv_1(R') - prv_1(k) = v_1 + 1$ and $prv_j(R') - prv_j(k) < v_j + 1$ for $j = 2, \ldots, \sigma$, because of the definition of R'. Then it can be seen that $prv(R') - prv(k) \not\leq v$. However, taking $R' - 1$ we have that $prv_1(R' - 1) - prv_1(k) < v_1 + 1$, so $prv(R' - 1) - prv(k) \leq v$. It proves that $R' - 1 = \max\{j \,|\, prv(j) - prv(k) \leq v\}$.

As a consequence of this theorem, calculating the functions of *Expand*, *Shrink* and *Refine* can be done in $O(\sigma)$ using an inverted index table, Basically, the position of each character in the string is stored, then it can be known which is the position in the text of the $n'th$ occurrence of the character $a \in \Sigma$ in constant time. Note that using this approach with all the elements of the alphabet it can be calculated each position in the text where a Parikh vector can fit, if the $n'th$ occurrence of character $a \in \Sigma$ does not exist, then it means that in the text there are less than $n \, a's$.

For the functions *Expand* and *Shrink* we calculate first the Parikh vectors corresponding to $prv(L) + q - \delta$ and $prv(R) - q + \delta$ then we use the inverted index table to find the minimum position in which every character of these Parikh vectors can fit in the text, finally we calculte the maximum of this positions, because it is necessary that all the characters of the Parikh vectors fit in the text. For the function *Refine* we calculate the Parikh vector $prv(L) + q + \delta$ and then we use the inverted index table to find the positions where the occurrences + 1 are, then we take the minimum and we make -1 to get the last position where the Parikh vector can fit completely.

Obviously, implementing these functions as described above has a time complexity of $\Theta(\sigma)$, which makes the complete algorithm to have a worst case time complexity of $\Theta(\sigma n)$ for a query, however, this happens in a few singular cases. in [6] it is proved that the average time complexity is sublinear when implementing this functions in $\Theta(\sigma)$.

3.1 Parikh vectors as bit vectors

Previously, all the implementations of Parikh vectors had been made using integer vectors. Nevertheless, due to the intrinsic paralellism of bit words, we represent here the Parikh vectors as bit vectors, storing many integers in the same bit word to make faster some operations of the Parikh vectors. For each element in the Parikh vector we use the number of bits that the integer represent and one more to use as a carry. Since it is necessary to know each of the elements of the Parikh vector when using the inverted index table we just use shiftings in order to get this values in constant time.

We use the bit vectors mainly to make a speed up in three operations used on the ESR algorithm: addition, substraction and comparation. These operations were defined component-wise, then what it is done at the end is adding, substracting and comparing all the elements of the Parikh vectors, but in parallel, using the properties of operations on bit words. The addition and the substraction are still being equal to the operations in integers, it is just necessary to verify before substracting, that the result will be non-negative. Moreover, for the comparation the function with bit vectors is:

$$\leq (a, b) := (b \,|\, carries) - a) \,\&\, carries) \neq carries) \tag{8}$$

Basically, we are 'adding' the carries to the element who is suposed to be greater or equal, namely b. In case that $a \leq b$, when substracting a to b, all the carries in the result should be setting to 1, in other case, then at least one of the positions of the carries will be 0. in Figure 3 it is showed an example of the use of bit vectors and the operations of addition and comparation.

Parikh Vectors				Bit Vectors															
				c =	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
a =	3	1	3	a' =	0	0	0	1	1	0	0	0	0	1	0	0	0	1	1
b =	2	2	3	b' =	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1
a + b	5	3	6	a' + b'	0	0	1	0	1	0	0	0	1	1	0	0	1	1	0
				b''= b' | c	1	0	0	1	0	1	0	0	1	0	1	0	0	1	1
				a''=b'' - a'	0	1	1	1	1	1	0	0	0	1	1	0	0	0	0
$a_i \leq b_i$	F	T	T	a'' & c	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
a ≤ b	False			a' ≤ b'	False														

Figure 3. Bit vectors corresponding to the Parikh vectors $a = \{3, 1, 3\}$ and $b = \{2, 2, 3\}$, the carry used is the bit vector c. Highlighted are the results of the binary operations addition and less or equal.

As it can be deduced, using of bit vectors instead of integer Parikh vectors can help to get a better performance of the operations which are executed many times in

a run of the algorithm. Many of the properties of bits arquitecture from computers are used for the algorithm.

3.2 All Matchings

So far it has been showed how to calculate the maximal matchings on a query, however, it can be of interest to find all the possible matches on a text. Therefore, in this section it is showed how the algorithm can be used to make this possible.

First, note that just using the functions *Expand* and *Shrink* the positions $L+1$ and R where the query has possibly a match are obtained, additionally, if after using both functions, $(L+1,\ R)$ is a match, then it is the shortest possible match starting at position $L+1$.

Second, note that the function *Refine* is used in order to make the match a maximal one, then after *Refine* it is known that there are no possible matchings starting at position $L+1$ and ending after position R, this is the longest possible match starting at position $L+1$.

Finally, using this two properties of the functions, we keep track of the position R after the function *Expand*, so if we have a match after *Shrink*, we use *Refine* to get the position R' and all the substrings starting at position $L+1$ and finishing between positions R and R' inclusive are matches, note that $R' - R \leq 2 * |\delta|$. After this we make a new cycle of *Expand* starting at position $L+2$ and ending at position R.

In Figure 4 the pseudocode of the All Matching algorithm is presented.

Input: A Parikh vector q a vector of possible errors δ
Output: A set *Matches* with all the ocurrences of q in s allowing the error δ

```
1: L ← 0, R ← 0, R' ← 0, Matches ← ∅
2: while L < n − |q − δ| and R < n do
3:     if q − δ ≰ p(s[L + 1, R]) then
4:         R ← Expand(L, q − δ)
5:     L ← Shrink(R, q + δ)
6:     if p(s[L + 1, R] ≥ q − δ) then
7:         R' ← Refine(L, q + δ)
8:         for j = R to R' do
9:             add (L + 1, j) to Matches
10:        L ← L + 1
11: return Matches
```

Figure 4. Pseudocode of the algorithm to calculate all the matchings for the δ - Approximate Jumbled Pattern Matching

3.3 Min-Error Matching

In some cases it is not of interest to find the maximal matchings nor all the possible matchings in a text, but the closest matchings to the query q, this is what we call a Min-Error Matching.

More formally, an occurrence $(i,\ j)$ is said to have minimum error, if neither $(i-1,\ j)$, $(i+1,\ j)$, $(i,\ j-1)$ nor $(i,\ j+1)$ is a match or is an occurrence of some Parikh vector q' such that $|q - q'| \leq |p(s_i, \ldots, s_j) - q|$, in other words, if when we extend or shrink the window of the occurrence the error is bigger. this can be made finding all the possible matchings with the algorithm and then calculating the error

of each of them, this would have a cost of $O(2n|\delta|\sigma)$. If $|\delta|$ were of $O(n)$, then the number of possible matchings in worst case would be $O(n^2)$, nevertheless, in practice $|\delta|$ is $O(1)$, so the time worst case running of finding the Min-Error matches is still $O(n\sigma)$.

Example 3. Consider again the alphabet $\Sigma = \{a, b, c\}$ and the query $q = \{3, 1, 3\}$ with $\delta = \{1, 1, 1\}$ over the string $s = ccabbcbaaccbbaaccbabab$. It has 6 min-error occurrences, namely $(6, 11)$, $(8, 12)$, $(8, 14)$, $(9, 16)$, $(11, 17)$, and $(14, 19)$ with errors 1, 2, 2, 1, 2 and 1 respectively. In Figure 5 the results of finding the Maximal Matchings are compared with he results finding the Min-Error Matchings of the query in s.

Max Match	Parikh Vector	Error
(5, 11)	{2, 2, 3}	2
(6, 12)	{2, 2, 3}	2
(8, 17)	{4, 2, 4}	3
(13, 19)	{3, 2, 2}	2
(14, 21)	{4, 2, 2}	3

Min-Err Match	Parikh Vector	Error
(6, 11)	{2, 1, 3}	1
(8, 12)	{2, 1, 2}	2
(8, 14)	{3, 2, 2}	2
(9, 16)	{3, 2, 3}	1
(11, 17)	{2, 2, 3}	2
(14, 19)	{3, 1, 2}	1

Figure 5. Maximal occurrences and Min-Error ocurrences (left and right) of the query $q = \{3, 1, 3\}$ with $\delta = \{1, 1, 1\}$ in the text *ccabbcbaaccbbaaccbabab* and the respectively error of each match.

4 Experimental Results

All the experiments were run on a computer, with an intel i5 1.70 GHz CPU with 4 GB of RAM and running Ubuntu 14.04 LTS 64-Bit. The codes were writen uniformelly in C++ and compiled with Codeblocks.

The datasets were made taking randomly generated texts of different lengths with alphabets of size 2, 4, 8, 26 and 94. Each query was made also random and was tested with diferent possible deltas. Although it is known that random texts and random queries are not closed to the reality, it a good approximation to test the performance in average of the algorithm. [10]

Essentially, we compared our implementation against the implementation based on Wavelet trees using the most efficient implementation of these [15]. For both implementations, the tests were run several times and the relative differences on the average time from each test were taken.

In Figure 6 are presented the results of the tests, here it can be seen the improvement of our algorithm over other solutions, as it can be seen that there are cases where the results are even 3.5 times faster. In addition, it can be seen that the best results are for larger alphabets, this can be clearly deduced because of the advantages of the bit vectors that were used in our implementation of the algorithm.

5 Conclusions

We presented a new implementation of an algorithm to solve the δ - Approximate Jumbled Pattern Matching, this implementation speeds up searchings using tables of

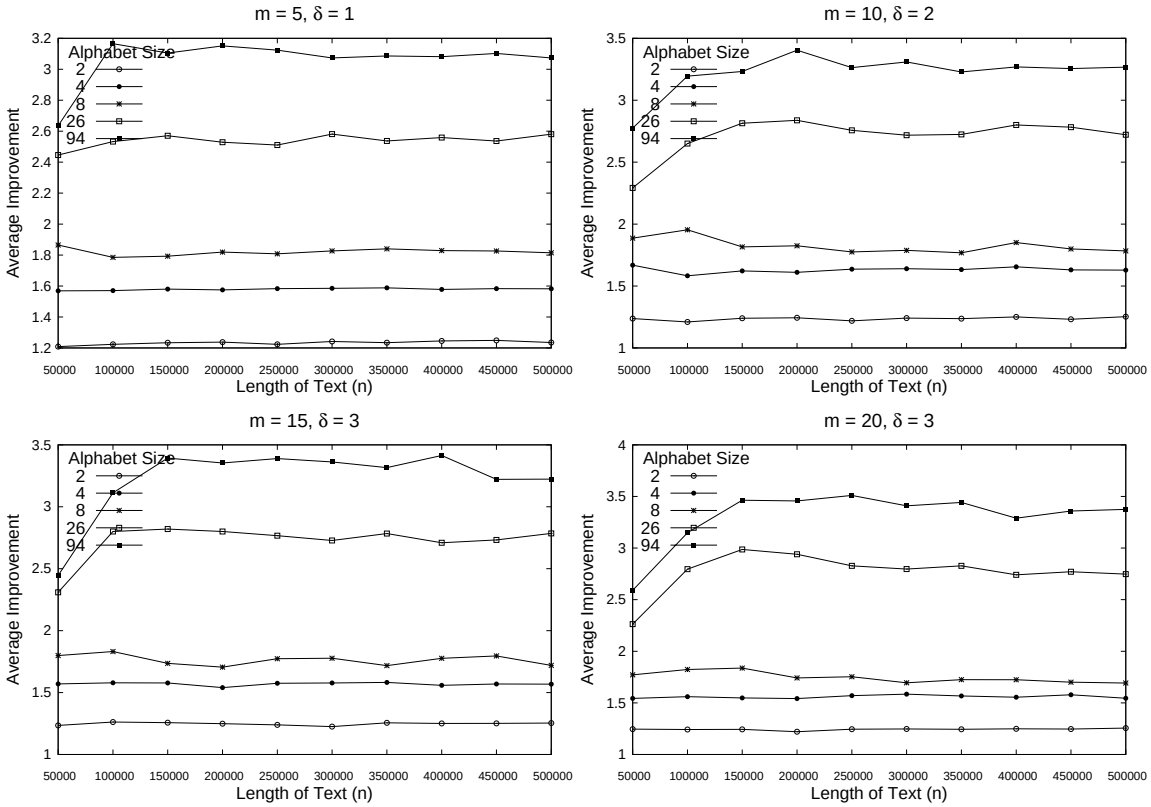

Figure 6. Average relative speed up of our implementation in random texts against efficient Wavelet tree implementation.

indexes and bit vectors. Our solution has a better performance when the alphabet is bigger. Also we showed how the algorithm can be used to calcuate all possible matchings and with it calculate also the Min-Error matchings in a text, though we are interested in a development of an algorithm which can find Min-Error matchings in a text without calculating the error for all possible matchings.

Although for binary alphabets our solution makes a practical improvement on the δ - Approximate Jumbled Pattern Matching problem, several improvements can still be done because of the many properties Parikh vectors have with binary alphabets, and does not have on general alphabets [1,13]. Even though the binary problem has been studied before on the Exact version, it has not been done on the Approximate Version.

References

1. G. Badkobeh, G. Fici, S. Kroon, and Zs. Lipták: *Binary jumbled string matching for highly run-length compressible texts*. Information Processing Letters, 113(17) 2013, pp. 604–608.
2. G. Benson: *Composition Alignment*, in Algorithms in Bioinformatics, vol. 2812 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 447–461.
3. 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.
4. S. Böcker: *Simulating multiplexed SNP discovery rates using baes-specific cleavage and mass spectrometry*, in European Conference on Computational Biology 2006 (ECCB 2006), vol. 23, 2006, pp. e5–e11.

5. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *On Table Arrangements, Scrabble Freaks, and Jumbled Pattern Matching*, in Fun with Algorithms, vol. 6099 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 89–101.
6. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *On Approximate Jumbled Pattern Matching in Strings.* Theory of Computing Systems, 50(1) 2012, pp. 35–51.
7. A. Butman, R. Eres, and G. M. Landau: *Scaled and permuted string matching.* Information Processing Letters, 92(6) 2004, pp. 293–297.
8. D. Cantone and S. Faro: *Efficient online Abelian pattern matching in strings by simulating reactive multi-automata*, in Proceedings of the Prague Stringology Conference 2014, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2014, pp. 30–42.
9. F. Cicalese, G. Fici, and Zs. Lipták: *Searching for jumbled patterns in strings*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2009, pp. 105–117.
10. D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt: *Fast Pattern Matching in Strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
11. L.-K. Lee, M. Lewenstein, and Q. Zhang: *Parikh Matching in the Streaming Model*, in String Processing and Information Retrieval, vol. 7608 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 336–341.
12. J. Mendivelso and Y. Pinzón: *A Novel Approach to Approximate Parikh Matching for Comparing Composition in Biological Sequences*, in Proceedings of the 6th International Conference on Bioinformatics and Computational Biology (BICoB 2014), 2014.
13. T. M. Moosa and M. S. Rahman: *Indexing permutations for binary strings.* Information Processing Letters, 110(18–19) 2010, pp. 795–798.
14. G. Navarro: *Multiple Approximate String Matching by Counting*, in Proc. WSP'97, Carleton University Press, 1997, pp. 125–139.
15. G. Navarro: *Wavelet Trees for All*, in Combinatorial Pattern Matching, vol. 7354 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 2–26.