A linear time string matching algorithm on average with efficient text storage

T. Berry and S. Ravindran

Department of Computer Science
Liverpool John Moores University
Byrom Street
Liverpool
United Kingdom

e-mail: T.Berry@livjm.ac.uk,S.Ravindran@livjm.ac.uk

Abstract. In this paper we describe an algorithm to search for a pattern in an efficiently stored text. The method used to store the text tranforms it to $\frac{\lceil \log_2 \sigma \rceil}{8}$ of its original size, where σ is the size of the alphabet set Σ. We prove that the algorithm takes linear time on average. We compare the new algorithm with some existing string matching algorithms by experimentation.

1 Introduction

String matching and Compression are two widely studied areas in computer science [5]. String matching is detecting a pattern P of length m in a larger text T of length n over an alphabet set Σ of size σ. Compression involves transforming a string into a new string which contains the same information but whose length is as small as possible. These two areas naturally lead to compressed string matching, i.e. searching for a pattern in a compressed text. This method will save both space and time.

In this paper we describe a string matching algorithm to search a pattern in a efficiently stored text. We can reduce the size of any given text according to the size of the alphabet being used. This is useful as although the cost of memory is reducing, the sizes of text databases are growing exponentially.

In Section 2 we describe our storage method that will transform the text to $\frac{\lceil \log_2 \sigma \rceil}{8}$ of its original size. This method is compared with other well-known compression algorithms by experiments in Section 3.

Section 4 describes a novel string matching algorithm on a text that is stored by using the method in Section 2. In Section 5 we prove that on average this algorithm takes $O(n + m)$ time. Our algorithm is compared with other well known string matching algorithms by experiments in Section 6.

2 Efficient storage of a text

We assume that the size of the alphabet set, σ, is in the range $1 \leq \sigma \leq 128$ and that we are representing each character in the alphabet with one byte. There are redundant bits in each byte as we only need $\lceil \log_2 \sigma \rceil$ bits to represent a character.

After we replace the characters in a text with $\lceil \log_2 \sigma \rceil$ bits, it is possible to replace eight consecutive bits in the binary text with its corresponding ASCII character. These eight consecutive bits are called a *block*. The decimal value of a block is the *code* of the block. This representation will reduce the storage space to $\frac{\lceil \log_2 \sigma \rceil}{8} n$, where n is the size of the original text.

For example, consider $\Sigma = \{A, B, C, D, E\}$ and $T = CACDABEB$. This text T of eight characters can be represented with three characters $T' = A1a$. First we represent the characters with $A = 000, B = 001, C = 010, D = 011$ and $E = 100$. This will give the binary representation of the text T:

$$\mathbf{0}1000001\mathbf{0}0110000\mathbf{0}1100001$$

The first bit in each block are shown in bold font. The codes for the text blocks are 65, 48 and 95 and their corresponding ASCII characters are 'A', '1', and 'a' respectively.

3 Comparison with existing compression algorithms

The method described in the last section is not compression as in the literature but does reduce the size of the original text. In this section we compare the well known text compression methods, Huffman encoding [8] and Lempel-Ziv encoding [9, 13, 14] with our method.

The Huffman encoding determines the length of the bit representation of the characters according to their frequency. It assigns smaller codes to high frequency characters and larger codes to low frequency characters.

In Lempel-Ziv (LZ) encoding [14] the file may be compressed to less than $\lceil \log_2 \sigma \rceil$ bits per character but requires re-occurring strings. Each of the repeated strings and each of the characters in the alphabet are represent by 12 bits. The gains from this method are reliant on there being enough repeated strings to counter the 12 bits which are used to represent each of the compressed strings.

The LZ encoding and its derivative LZW encoding [13] are used in UNIX utilities, compress and gzip. Another variation of LZ encoding (NR) is described in [9].

Table 1 (see Appendix) shows that our storage method is comparable to these methods. Although our method is not very good for text files with large alphabets. The method is competive for DNA, Recombinant DNA (RDNA) and hexadecimal files. Note that the main purpose of this paper is not compression, but for the searching of a pattern in a compressed file.

4 Searching in a text with efficient storage

In this section we describe an algorithm to find all exact occurrences of a pattern in a text. Here we assume that the text is stored as described in Section 2 and $\sigma \leq 128$. We describe the algorithm for $\sigma = 2$, we will see later that the algorithm can be easily adapted for $\sigma > 2$.

A substring of the pattern may overlap between consecutive text-blocks and a pattern may start in a text-block at any one of eight positions. During the search we need to look whether a prefix (or suffix) of a pattern is a suffix (or prefix) of a

text-block. Due to this problem we have to consider eight different expressions. Each expression is made up of pattern-blocks of length eight bits. There will be $m + 7$ pattern-blocks, where m is the length of a pattern.

For a pattern $P_1P_2..\ P_m$ we can construct the expressions as shown in Figure 1. Here we consider the case for m mod $8 = 0$. We number the pattern-blocks starting from 0 at the top left corner to $m+6$ in the bottom right corner as shown in brackets. The wildcard character N represents either 0 or 1, and $P_{i..j}$ represents $P_i..P_{j-1}P_j$, for $1 \leq i < j \leq m$.

Exp0:	NNNNNNNP_1	(0)	.....	$P_{m-14..m-7}$	(m-8)	$P_{m-6..m}$N	(m)
Exp1:	NNNNNN$P_{1..2}$	(1)	.....	$P_{m-13..m-6}$	(m-7)	$P_{m-5..m}$NN	$(m+1)$
Exp2:	NNNNN$P_{1..3}$	(2)	.....	$P_{m-12..m-5}$	(m-6)	$P_{m-4..m}$NNN	$(m+2)$
Exp3:	NNNN$P_{1..4}$	(3)	.....	$P_{m-11..m-4}$	(m-5)	$P_{m-3..m}$NNNN	$(m+3)$
Exp4:	NNN$P_{1..5}$	(4)	.....	$P_{m-10..m-3}$	(m-4)	$P_{m-2..m}$NNNNN	$(m+4)$
Exp5:	NN$P_{1..6}$	(5)	.....	$P_{m-9..m-2}$	(m-3)	$P_{m-1..m}$NNNNNN	$(m+5)$
Exp6:	N$P_{1..7}$	(6)	.....	$P_{m-8..m-1}$	(m-2)	P_mNNNNNNN	$(m+6)$
Exp7:	$P_{1..8}$	(7)	.....	$P_{m-7..m}$	(m-1)		

Figure 1: Expressions for a pattern $P_1P_2..\ P_m$ when m mod $8 = 0$.

The naive algorithm will compare a text-block with the first pattern-blocks in each expression. If any of these pattern-blocks matched with the text-block, we need to compare the consecutive text-blocks with the rest of the pattern-blocks in the expression.

Our algorithm first constructs a table called the *Block-Table*. The Block-Table has 256 columns and $m + 7$ rows as there are 256 possible blocks in a text and $m+7$ is the number of pattern-blocks we need to consider. The table is initialised to 0. The $(i, j)^{th}$ entry in the table is defined as follows, where i, $0 \leq i \leq m + 6$, is the pattern-block number and j, $0 \leq j \leq 255$, is the code for a block. Suppose that the pattern-block does not have a wildcard character, the $(i, j)^{th}$ entry is 1, if the code for pattern-block i is equal to j. If there is one or more wild cards in the pattern-block, we consider all the possible blocks. For example, if the i^{th} pattern-block is NN111000, the $(i, j)^{th}$ entry is equal to 1 for all j, where j is the code for 00111000, 01111000, 10111000 or 11111000.

For each expression we only have to compare one pattern-block with a text-block, and if these two match then we compare the rest of the pattern-blocks in the expression with the corresponding text-blocks. We choose a pattern-block (from each expression) which has the minimum number of possibilities of matching with a text-block. We build the *Order-Table* of dimensions 8 by $\lceil \frac{m+7}{8} \rceil$ which contains the order in which to examine the pattern-blocks for each expression. For each pattern-block the number of possibilities of matching a text-block can be found by adding the values in the row of the pattern-block in the Block-Table.

¿From these we construct a *Search-Table* of dimensions 8×256, and it is initialised to -1. In the first row of the Search-Table, we enter pattern-block numbers from the first column of the Order-Table. If j is the code for these pattern-blocks, we enter the pattern-block numbers at the j^{th} column, for all j, $0 \leq j \leq 255$. A column number may be the code for more than one of the chosen pattern-blocks. This is because a text-block can match pattern-blocks from more than one expression. As there are

only eight expressions we need a maximum of eight rows. For example, the chosen pattern-blocks, 110011NN and NN001100, will both match the block 11001100. We enter the pattern-blocks (110011NN and NN001100) numbers in the first and second rows respectively of the column k, where k is the code for 11001100.

We begin the search at the begining of the text and compare the text-blocks with chosen pattern-blocks in the Search-Table. We check the j^{th} column in the Search-Table, where j is the code of the text-block. If the entry is -1 then we check the next text-block. Otherwise we know that the text-block is in the pattern. We compare the rest of the pattern-blocks in the expression with the corresponding text-blocks until either full match or mismatch is found using the Block-Table and Order-Table. Before we move to the next text-block, we check if the entry in the next row of the Search-Table is -1. We repeat this process if the entry is not -1, otherwise we check the next text-block.

If $\sigma > 2$, we have to convert the pattern into a binary string by mapping the characters into $\lceil \log_2 \sigma \rceil$ bits as we did in Section 2. Here we don't have to consider all the expressions. This is because in the pattern-blocks 0, 1, .. , 7 (from expressions 0 to 7 respectively) the pattern starts at positions 7, 6, .. , 0 respectively (see Figure 1). The positions are numbered from left to right in a pattern-block.

We can show that for all σ, in a comparison we need at most $\lceil \frac{8}{\lceil \log_2 \sigma \rceil} \rceil$ expressions. There are two cases which depend on whether $\frac{8}{\lceil \log_2 \sigma \rceil}$ is an integer.

Suppose $\frac{8}{\lceil \log_2 \sigma \rceil}$ is an integer then we have the following case. For example, if an alphabet is represented by two bits in the compressed file (i.e. $\sigma = 3$ or 4) then a pattern can only start at even positions in the text-blocks. So in this case we only need to consider expressions 1, 3, 5 and 7.

Suppose $\frac{8}{\lceil \log_2 \sigma \rceil}$ is not an integer then we have the following case. For example if we are using 3 bits (i.e. $5 \leq \sigma \leq 8$) to represent an alphabet, then we need all the eight expressions. But in any comparison we need at most three expressions. Consider three consecutive text-blocks. Without loss of generality assume that the binary representation of a character starting at position 0 in the first of these three blocks. Then a pattern can start at positions 0, 3, or 6 in the first text-block, positions 1,4 or 7 in the second text-block or positions 2 or 5 in the third text-block. For the first text-block we need to consider the expressions 7, 4 and 1. For the second text-block we need to consider the expressions 0, 3 and 6. For the third text-block we need to consider the expressions 2 and 5.

5 The average running time

The pre-processing of our algorithm takes $O(m)$ time, as the Block-Table, Order-Table and the Search-Table can be constructed in $O(m)$ time. The worst case for the search will take $O(mn)$ time. In this section we will show that the algorithm performs on average at most $2n$ comparisons. ¿From this we can say that the average running time of the algorithm is $O(n + m)$. We also justify this with experiments at the end of this section.

At the end of the previous section we showed that we need to consider all eight expressions only when $\sigma = 2$. First we prove that the average number of comparisons for this worst case.

There are only 256 possible different blocks. If we assume that each of the 256 blocks occurs in a text with equal frequency, then we have the following lemma. Let $\Gamma_{PB}(j)$ be the probability of a pattern-block j matches a text-block.

Lemma 1: $\Gamma_{PB}(j) = \frac{1}{2^{8-w}}$, where w is the number of wildcard character N in the pattern-block.

Recall that when we compare a text-block with a pattern-block, we choose a pattern-block (from each expression) which has the minimum number of possibilities of matching with a text-block (i.e. the pattern-block with minimum number of wildcard character N). If any of these pattern-blocks matches with the text-block, then we choose the pattern-block with the minimum number of wild cards among the remaining pattern-blocks in the expression. In an attempt, for each expression we repeat this step until either a full match or mismatch is found.

For example, consider the expressions for $m = 34$. Figure 2 shows the values of w in a pattern-block for each expression (pattern-block numbers are in brackets).

Exp0:	7 (0)	0 (8)	0 (16)	0 (24)	0 (32)	7 (40)
Exp1:	6 (1)	0 (9)	0 (17)	0 (25)	0 (33)	
Exp2:	5 (2)	0 (10)	0 (18)	0 (26)	1 (34)	
Exp3:	4 (3)	0 (11)	0 (19)	0 (27)	2 (35)	
Exp4:	3 (4)	0 (12)	0 (20)	0 (28)	3 (36)	
Exp5:	2 (5)	0 (13)	0 (21)	0 (29)	4 (37)	
Exp6:	1 (6)	0 (14)	0 (22)	0 (30)	5 (38)	
Exp7:	0 (7)	0 (15)	0 (23)	0 (31)	6 (39)	

Figure 2: The number of wildcards in pattern-blocks for $m = 34$

There are three columns with all zeros which are the first three columns in the Order-Table. In general, for all m, if $m \bmod 8 \neq 7$, there are $\lambda = \lfloor \frac{m-7}{8} \rfloor$ number of columns will have all zeros. If $m \bmod 8 = 7$, and $m \geq 15$ we will have $\lambda - 1$ columns with all zeros, and the remaining one with seven zeros in a column and the eighth zero in another column. For example, Figure 3 shows the number of wildcards in pattern-blocks for $m = 23$ (i.e. $m \bmod 8 = 7$.). We can see that there is one (i.e. $\lambda - 1$) column which is the second column with all zeros. The remaining column of all zeros is the fourth column with seven zeros and the eighth zero is in the first column (shown in bold font).

Exp0:	7 (0)	0 (8)	0 (16)	2 (24)
Exp1:	6 (1)	0 (9)	0 (17)	3 (25)
Exp2:	5 (2)	0 (10)	0 (18)	4 (26)
Exp3:	4 (3)	0 (11)	0 (19)	5 (27)
Exp4:	3 (4)	0 (12)	0 (20)	6 (28)
Exp5:	2 (5)	0 (13)	0 (21)	7 (29)
Exp6:	1 (6)	0 (14)	0 (22)	
Exp7:	**0 (7)**	0 (15)	1 (23)	

Figure 3: The number of wildcards in pattern-blocks for $m = 23$

¿From this observation we have Lemma 2. Let Φ_i be the probability of i number of pattern-blocks matching with the text-blocks in an expression at an attempt. In

other words Φ_i is the probability of the algorithm making *at least* $i+1$ comparisons at an attempt.

Lemma 2: For all m and $\sigma = 2$, $1 \leq i \leq \lambda$, $\Phi_i = 8 \times \frac{1}{256^i}$, where $\lambda = \lfloor \frac{m-7}{8} \rfloor$.

Proof: For all m, each expression has λ number of pattern-blocks with $w = 0$. At an attempt, we can choose pattern-blocks with $w = 0$ from each of the eight expressions for the first λ comparisons. From Lemma 1 we have $\Gamma_{PB}(j) = 1/256$ if $w = 0$. In an attempt we will have the $i+1^{th}$ comparison only if i number of pattern-blocks in an expression matches the corresponding text-blocks. The probability of i matches for an expression is $\frac{1}{256^i}$ and there are eight expressions and so Φ_i is $\frac{8}{256^i}$, $1 \leq i \leq \lambda$. □

In an attempt, for $2 \leq m \leq 9$ and $10 \leq m \leq 14$ we have at most 2 and 3 comparisons respectively. Hence we only need to know the values of Φ_1 for $2 \leq m \leq 9$, and Φ_1 and Φ_2 for $10 \leq m \leq 14$. We can calculate these values easily. For example, the following shows the values of w in a pattern-block for each expression (pattern-block numbers are in brackets) for $m = 10$. First we will select the pattern-blocks 8 to 11 and 4 to 7.

Exp0:	7	(0)	0	(8)	7	(16)
Exp1:	6	(1)	0	(9)		
Exp2:	5	(2)	1	(10)		
Exp3:	4	(3)	2	(11)		
Exp4:	3	(4)	3	(12)		
Exp5:	2	(5)	4	(13)		
Exp6:	1	(6)	5	(14)		
Exp7:	0	(7)	6	(15)		

Figure 4: The number of wildcards in pattern-blocks for $m = 10$

$$\begin{aligned}
\Phi_1 &= \Gamma_{PB}(8) + \Gamma_{PB}(9) + \Gamma_{PB}(10) + \Gamma_{PB}(11) + \Gamma_{PB}(4) + \Gamma_{PB}(5) + \Gamma_{PB}(6) + \Gamma_{PB}(7) \\
&= \frac{1}{8^{8-0}} + \frac{1}{8^{8-0}} + \frac{1}{8^{8-1}} + \frac{1}{8^{8-2}} + \frac{1}{8^{8-3}} + \frac{1}{8^{8-2}} + \frac{1}{8^{8-1}} + \frac{1}{8^{8-0}} \text{ (Lemma 1)} \\
&= 1/256 + 1/256 + 1/128 + 1/64 + 1/32 + 1/64 + 1/128 + 1/256 \\
&= 23/256
\end{aligned}$$

For Φ_2 we only need to consider the first expression. We can have at least 3 comparisons, iff pattern-blocks 8 and (assume we select) 0 match with the corresponding text-blocks.

$$\begin{aligned}
\Phi_2 &= \Gamma_{PB}(8) \times \Gamma_{PB}(0) \\
&= \frac{1}{8^{8-0}} \times \frac{1}{8^{8-7}} \text{ (Lemma 1)} \\
&= 1/256 \times 1/2 \\
&= 1/512
\end{aligned}$$

In an attempt, for all $m \geq 15$, after λ comparisons the pattern-blocks which have not yet been compared will be similar to the expressions for patterns of length m',

$7 \leq m' \leq 14$, where $m' = (m \bmod 8) + 8$ if $m \bmod 8 \neq 7$. Otherwise $m' = 7$. In other words, if we remove all the λ columns with all zeros from the expressions of pattern length $m \geq 15$, the number of wildcards in pattern-blocks will be the same as in the expressions of pattern length m'. For example, if we remove (i.e. λ) columns of all zeros from the number of wildcards in pattern-blocks, for $m = 34$ (see Figure 2), we will get the number of wildcards in pattern-blocks, for $m' = 10$ (see Figure 4) as in Figure 5.

Exp0:	7 (0)	0 (32)	7 (40)
Exp1:	6 (1)	0 (33)	
Exp2:	5 (2)	1 (34)	
Exp3:	4 (3)	2 (35)	
Exp4:	3 (4)	3 (36)	
Exp5:	2 (5)	4 (37)	
Exp6:	1 (6)	5 (38)	
Exp7:	0 (7)	6 (39)	

Figure 5: The number of wildcards in pattern-blocks, for $m' = 10$

Note that in any attempt for all m, we can have at most $\lambda + 1$ matches before we make the last comparison, if $m \bmod 8 = 0$, 1 or 7, otherwise $\lambda + 2$. For $m > 15$, we need to know $\Phi_{\lambda+1}$ and $\Phi_{\lambda+2}$. From the above observation we can calculate these values from the values of Φ_1 and Φ_2 for m, $7 \leq m \leq 14$. From these base values we can have the following Lemma. Note that $\lambda = 0$ for all $m \leq 14$.

Lemma 3: For $m \geq 7$,

$$\begin{aligned} \Phi_{\lambda+1} &= (1/256)^{\lambda} \times \alpha_b \text{ and} \\ \Phi_{\lambda+2} &= (1/256)^{\lambda} \times \beta_b, \end{aligned}$$

where α_b and β_b are the values of b^{th} base case in the first and second columns in the table below respectively and $b = m \bmod 8$.

base case	α	β
0	11/64	
1	15/128	
2	23/256	1/512
3	1/16	1/512
4	13/256	1/512
5	5/128	3/2048
6	9/256	5/4096
7	7/32	

Let Ψ_i be the probability of making *exactly* i comparisons at an attempt. Using Φ_i we can have an equation for Ψ_i:

$$\Phi_i = \Psi_{i+1} + \Psi_{i+2} + \cdots$$

This gives

$$\Psi_i = \Phi_{i-1} - \Phi_i$$

We know that we will make at least one comparison in every attempt. So Φ_0 is 1.

For all m and $\sigma = 2$, the maximum number of comparisons in any attempt is $\mu = \lceil \frac{m+7}{8} \rceil$, which is equal to $\lambda + 2$ if $m \bmod 8 = 0$, 1 or 7, otherwise $\lambda + 3$. So Φ_i is 0 for all $i \geq \mu$. This gives:

$$\begin{aligned} \Psi_1 &= 1 - \Phi_1 \\ \Psi_i &= \Phi_{i-1} - \Phi_i, 2 \leq i \leq \mu - 1 \\ \Psi_\mu &= \Phi_{\mu-1} \end{aligned}$$

Lemma 4: For $\sigma = 2$, the total number of comparisons, Ψ_{Total}, is less than or equal to $2n'$ on average, where n' is the number of text-blocks in the text.

Proof:
$$\begin{aligned} \Psi_{Total} &= n' \times \sum_{i=1}^{\mu} i \times \Psi_i \\ &= n' \times ((1 - \Phi_1) + 2(\Phi_1 - \Phi_2) + \cdots + \mu - 1(\Phi_{\mu-2} - \Phi_{\mu-1}) + \mu\Phi_{\mu-1}) \\ &= n' \times (1 + \Phi_1 + \cdots + \Phi_{\mu-2} + \Phi_{\mu-1}) \\ &= n' \times (1 + \sum_{i=1}^{\lambda} \frac{8}{256^i} + \Phi_{\lambda+1} + \Phi_{\lambda+2}) \text{ (Lemma 2)} \\ &\leq 2n' \end{aligned}$$

This is because $\sum_{i=1}^{\lambda} \frac{8}{256^i} + \Phi_{\lambda+1} + \Phi_{\lambda+2} \leq 1$ (Lemmas 2 and 3) □

Lemma 5: For $\sigma > 2$, the total number of comparisons, Ψ_{Total}, is $O(n)$, where n is the size of the original text.

Proof: The probability of more than one comparison in an attempt is $\Phi_1 + \cdots + \Phi_{\mu-2} + \Phi_{\mu-1}$ (see Lemma 4), where $\mu = \lceil \frac{m\lceil \log_2 \sigma \rceil + 7}{8} \rceil$. Note that $m\lceil \log_2 \sigma \rceil$ is the length of the pattern when we convert it into a binary string. We show in the last section that in an attempt we only need to consider a maximum of $\lceil \frac{8}{\lceil \log_2 \sigma \rceil} \rceil$ expressions when $\sigma > 2$. Hence, for $\sigma > 2$, $\Phi_1 + \cdots + \Phi_{\mu-2} + \Phi_{\mu-1}$ is less than the value given for $\sigma = 2$. □

From these Lemmas we have the following Theorem.

Theorem: The average running time of our algorithm is $O(n + m)$.

To show this is also true in practice we counted the number of comparisons by running our algorithm. Table 2 in the Appendix shows the estimated number of comparisons (Ψ_{Total}) and the actual number of comparisons. We used the same texts for each σ as in Table 1 (Section 3). For each pattern length we use 100 random patterns. The actual number of comparisons in the table is the total number of comparisons divided by the number of patterns of that length. The pattern length given in Table 2 is the length of the original pattern.

6 Comparison with existing string matching algorithms

In this section we compare the existing string matching algorithms with our algorithm, the BRS algorithm. There are a number of strings matching algorithms available in the literature. We have chosen seven of them, BR, BM, HOR, QS, RAI, SMI, RF and NR algorithms which can be found in [1, 2, 7, 12, 10, 11, 6, 9] respectively. The first six algorithms were found to be fast in [1]. Animations of these algorithms can be found at [4] and more information about the algorithms can be found in [3].

The experiments were carried for all the algorithms on an un-compressed text, except for our BRS algorithm and the NR algorithm [9]. The text used for these experiments was the same text as in Table 1 (Section 3). The patterns used in these experiments are generated randomly. For each σ and m, we tested 100 patterns and we measured the total (user) time (including pre-computation time) in seconds to search for all 100 patterns. We repeat each test 10 times and take the average. We used an Intel 486-DX2-66 processor based machine with 8 megabytes of RAM and a 100 megabyte hard drive running S.u.S.E. Linux 5.2 to conduct the experiments. All the algorithms were coded in C. The results of the experiments are in the Appendix (Tables 4 to 8).

7 Conclusions

The method described in Section 2 to store a text will reduce the original text size to $\frac{\lceil \log_2 \sigma \rceil}{8} n$. Although this method is not compression as in the literature, it reduces the space and it is comparable with the existing methods.

The main aim of this paper is string matching in a compressed text. Our string matching algorithm compares two blocks, checks whether a prefix (or suffix) of a block is a suffix (or prefix) of the other block. This takes constant time and uses byte processing. In practice, byte processing is much faster than bit processing because bit shifting and masking operations are not necessary at search time. We prove that the average time taken by our algorithm is $O(n + m)$. We also justified our average running time by experiments.

Using our algorithm one can keep texts (with an alphabet of $2 \leq \sigma \leq 128$ characters) compressed indefinitely and perform the search for a pattern. These methods will save both time and space. The experimental results show that our algorithm is more efficient than the existing algorithms for $\sigma \leq 16$. Texts with such a small alphabet are DNA, RDNA and hexadecimal files. One can improve our algorithm so that it performs well for large alphabet sets.

References

[1] Berry T., Ravindran S., *"A fast string matching algorithm and experimental results"*, Prague Stringology Club Workshop '99, 1999.

[2] Boyer R. S., Moore J. S., *"A fast string searching algorithm"*, Communications of the ACM, 23(5), pp 1075-1091, 1977.

[3] Charras C., Lecroq T., 1997, Exact string matching, available at: `http://www-igm.univ-mlv.fr/~lecroq/string.ps`

[4] Charras C., Lecroq T., 1998, Exact string matching animation in JAVA available at: `http://www-igm.univ-mlv.fr/~lecroq/string/`

[5] Crochemore M., Rytter W., *"Text algorithms"*, Oxford University Press, 1994.

[6] Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., LeCroq, T., Plandowki, W., Rytter, W., *"Speeding up two string matching algorithms"*, Algorithmica 12(4/5), pp 247-267, 1994.

[7] Horspool R. N., *"Practical fast searching in strings"*, Software Practice and Experience, 10(6), pp 501-506, 1980.

[8] Huffman D.A., *"A method for the construction of minimum redundancy codes"*, Proceedings of the Institute of Radio Engineers, 40, pp 1098-1101, 1951.

[9] Navarro G., Raffinot M., *"A general practical approach to pattern matching over Ziv-Lempel compressed text"*, Proceedings of Combinatorial Pattern Matching 99, Lecture Notes in Computer Science, 1645, pp 14-36, 1999. .

[10] Raita T., *"Tuning the Boyer-Moore-Horspool string searching algorithm"*, Software Practice and Experience, 22(10), pp 879-884, 1992.

[11] Smith P. D., *"Experiments with a very fast substring search algorithm"*, Software Practice and Experience 21(10), pp 1065-1074,1991.

[12] Sunday D. M., *"A very fast substring search algorithm"*, Communications of the ACM. 33(8), pp 132-142, 1990.

[13] Welch T. A., *"A technique for high-performance data compression"*, IEEE Computer, 17(6), pp 8-19, 1984.

[14] Ziv J., Lempel A., *"A universal algorithm for sequential data compression"*, IEEE Trans. On Information Theory, IT-23, 1978.

Appendix

σ	Our method	Huffman	Compress	Gzip	NR
2	62500	62500	71579	79644	121110
3	125000	104107	110629	118776	178706
4	125000	125000	136945	146402	215764
5	187500	149935	161641	168813	244192
8	187500	187500	209053	211543	297634
9	250000	201313	223571	226617	310964
16	250000	250000	288546	285834	373658
17	312500	257293	294476	290854	377491
32	312500	312500	367527	330150	449265
33	375000	316232	370975	332592	451570
64	375000	375000	461069	378224	493981

Table 1: Compressed text sizes for a random text of 500,000 bytes.

alphabet of 2			alphabet of 4			alphabet of 8		
Pat Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual
5	85938	85413	2	156250	156258	2	207031	255959
10	68237	68276	4	135742	136513	4	190795	191710
20	64556	64446	8	127288	126999	6	189632	189931
30	64460	64460	12	126962	126962	8	189462	189537
40	64460	64467	18	126960	126962	12	189460	189898
50	64460	64473	24	126960	126962	16	189460	189551

Table 2: Estimated versus actual number of comparisons of our BRS algorithm

alphabet of 16			alphabet of 32			alphabet of 64		
Pat. Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual
2	260742	265331	2	318237	322155	2	378296	378678
4	252288	252013	3	314507	314567	3	377132	376990
6	251960	251956	4	314556	314581	4	376962	376980
8	251960	251962	6	314460	314509	5	376962	376965
10	251960	251957	8	314460	314297	7	376960	376482
12	251960	251959	10	314460	314348	9	376960	376503

Table 3: Estimated versus actual number of comparisons of our BRS algorithm

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
5	14.2	29.1	31.3	31.5	31.1	28.7	31.8	32.6	30.7
10	5.3	27.0	24.7	30.9	31.0	27.7	31.4	22.0	30.5
20	4.4	27.3	20.4	28.8	32.4	26.6	31.0	18.2	29.5
30	4.2	27.3	18.3	31.2	31.2	28.0	31.4	16.0	27.5
40	4.2	28.3	17.3	29.7	31.3	27.9	30.7	13.5	28.5
50	5.2	26.5	16.4	30.5	30.0	27.7	31.1	15.0	28.4

Table 4: Search times for $\sigma = 2$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
4	8.6	15.3	20.5	20.9	20.3	20.2	23.8	21.3	21.6
8	5.3	13.1	17.6	18.1	19.3	18.7	19.5	17.3	20.7
12	5.7	12.5	19.3	18.8	18.7	18.0	18.3	15.3	17.6
16	5.7	12.9	17.4	15.8	17.4	17.3	17.7	13.6	18.4
20	5.7	12.0	17.2	18.5	17.6	17.9	18.5	14.1	20.5
24	5.7	12.5	16.7	17.7	18.6	16.6	18.1	12.7	20.2

Table 5: Search times for $\sigma = 4$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
3	9.9	15.7	22.6	18.6	17.6	16.9	19.3	18.0	28.9
4	14.0	17.0	25.8	21.2	18.7	21.1	21.3	18.9	27.6
6	8.6	13.7	19.7	16.9	16.0	16.5	16.0	15.8	23.5
10	8.5	12.7	15.7	14.1	15.1	14.9	15.1	14.1	25.5
14	8.7	12.0	15.7	12.4	14.2	13.3	13.8	13.0	25.2
18	8.4	11.1	15.0	13.6	14.0	12.7	13.4	13.2	25.5

Table 6: Search times for $\sigma = 8$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
2	14.1	19.4	33.0	25.8	21.0	24.4	24.4	19.6	33.4
4	9.8	15.2	21.7	17.8	17.0	17.9	17.7	16.2	31.5
6	9.8	13.4	16.6	14.0	14.6	13.6	15.0	13.4	31.5
8	9.7	12.3	16.2	14.4	13.9	13.2	13.8	13.2	27.7
10	9.7	12.1	14.2	13.2	13.6	12.3	13.0	13.6	29.6
12	9.9	11.1	14.3	13.0	12.3	13.0	13.4	12.9	31.1

Table 7: Search times for $\sigma = 16$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
2	66.5	18.6	33.0	23.6	19.5	23.8	22.3	19.0	39.1
3	42.1	16.3	24.0	20.2	17.7	19.8	20.3	16.6	40.1
4	31.9	14.8	21.2	17.1	15.5	15.4	17.5	15.0	37.2
6	39.7	12.3	17.5	13.2	14.7	14.6	15.5	14.1	36.2
8	37.9	12.3	15.5	13.4	13.4	13.2	13.9	13.5	38.8
10	48.2	11.5	15.0	12.4	11.8	12.5	13.7	13.0	34.2

Table 8: Search times for $\sigma = 32$