

BLIM: A New Bit-parallel Pattern Matching Algorithm Overcoming Computer Word Size Limitation

M. Oğuzhan Külekci

Abstract. Bitwise operations are executed very fast in computer architecture. Algorithms aiming to benefit from this intrinsic property can be classified as bit-parallel algorithms. Bit-parallelism has been widely investigated in pattern matching area since the introduction of the Shift-Or algorithm. In the original idea, there were no shift mechanism, and the input pattern length is required to be less than the *computer word size* (W) to benefit from the full power of bit-parallelism. The lack of the shift mechanism was removed by the succeeding algorithms of this genre, but W limitation has not been overcome in an elegant way. This study proposes a new bit-parallel algorithm, given name *BLIM* (bit-parallel length independent matching), for exact pattern matching that does not restrict the input pattern to be shorter than the word size. Multiple pattern case is also addressed, and it is shown that up to computer word size number of patterns, whatever their lengths are, can be searched simultaneously in a single bit-parallel framework. Similar to other algorithms of this genre, BLIM is also capable of handling fixed-length gaps and character classes in the input strings as well. The proposed algorithm is compared with the other alternatives of its class, mainly the shift-or and BNDM variants. Experimental results indicate that BLIM is compatible with the previous bit-parallel algorithms with an additional gain of overcoming the word size limitation.

Keywords. pattern matching, bit-parallelism, string search.

A less extensive version of this study has appeared in ISAAC'08, the 19th International Symposium on Algorithms and Computation [12], and a related talk was given in LSD&LAW'09, London Stringology Days & London Algorithmic Workshop [13].

1. Introduction

Searching for exact or approximate occurrences of single or multiple patterns on text files is one of the fundamentals in computer science. There exists numerous algorithms focusing on various aspects of the general problem [6, 4, 2, 15]. Although the field is well studied for more than three decades, new challenges still appear as a consequence of developments in the related areas. For example current antivirus systems require searching hundreds of thousands virus patterns that necessitates *massive* multiple pattern matching, which was not that much serious before. A similar necessity arises for next generation high-throughput DNA sequencing systems also.

A formal review with a classification of pattern matching algorithms can be found in [5]. In general a window of length usually equal to the pattern size is slid throughout the text, and characters residing in that investigation window are visited in an order for checking a possible match. Following this check operation, the window is moved by an amount computed according to the characters accessed. Shift amount and scan order are usually computed in a preprocessing stage. The performance depends on the speed of checking for a possible match or mismatch, and the amount of shift. Thus, fast detection of mismatches, as mismatches are expected to be much more frequent than the matches, and maximal shifts are the two important points while designing an effective algorithm for any particular case.

Processors execute bitwise operations very fast. Bit-parallel pattern matching algorithms aim to benefit from that intrinsic property of computers for fast comparison of the input pattern against the text. These algorithms are explained in section 2 with more detail. There were two main problems in the original idea (Shift-Or algorithm) introduced by Baeza-Yates&Gonnet [3]. The first one, lack of sift mechanism, was resolved by the succeeding studies on the algorithm. The other one, which restricts the length of the input pattern to be less than the computer word size (W) has not been solved in an elegant way up to now, although there exists some alternatives for dealing with strings longer than W . The situation worsens in multi-pattern case, where the total length of the input patterns is more likely to exceed W than in single pattern case.

This study focuses on overcoming word size limitation in bit-parallel pattern matching. The proposed algorithm, given name *BLIM* (bit-parallel length independent matcher), offers a platform for matching patterns of any length in a unified bit-parallel fashion. Upto W number of patterns, whatever their sizes are, can also be scanned. Similar to other algorithms of its genre, character classes and fixed-length gaps in the input patterns are handled as well. The experiments conducted to compare BLIM with other bit-parallel algorithm indicate that the performance is compatible with the additional capability of searching patterns longer than W .

2. Previous Bit-parallel Algorithms

Let $T = t_0t_1t_2 \dots t_{n-1}$ be the text of length n , $P = p_0p_1p_2 \dots p_{m-1}$ be the pattern of length m that will be scanned on T , and W denotes computer word size. The exact pattern matching task is to find all occurrences of given pattern P in text T . The alphabet of the text will be shown by Σ , and the number of characters in the alphabet by σ .

The initial attempt of using bit-parallelism in pattern matching was by Baeza-Yates&Gonnet [3]. The pseudo code of their *Shift-Or* (SO) is given in Algorithm 1. Despite its linear time complexity, in practice SO was very effective especially for searching moderate patterns on texts composed of small alphabets, which actually fits well for biological sequences. The original algorithm did not include a shift mechanism, but the gain obtained from the speed of the bitwise operations compensated this absence.

The positions of the characters occurring in the input pattern are marked on a computer word in the preprocessing phase. The state vector D is updated with the corresponding mask of the visited character while passing over the text sequentially. The update operation is composed of a shift of D combined with a bitwise *or* operation of the mask of the character, and hence the algorithm is given name shift-or. A match is reported if the D reaches a final state.

Algorithm 1 Shift-Or(P,m,T,n)

```

1: for all  $i \in \Sigma$  do
2:    $B[i] = 1^W$ ;
3: for  $i = 0$  to  $m - 1$  do
4:    $B[P[i]] = B[P[i]] \& 1^{m-i-1}01^i$ ;
5:  $D = 1^W$ ;  $mm = 0^{W-m}10^{m-1}$ ;  $i = 0$ ;
6: while  $i < n$  do
7:    $D = (D \ll 1) | B[T[i]]$ ;
8:   if  $((D \& mm) \neq mm)$  then
9:     Pattern detected beginning at  $T[i]$ ;
```

A similar approach was also proposed by Wu&Manber [20], which later on implemented as *agrep* approximate string search utility. Fredriksson&Grabowski [9, 10] introduced a fast variant of the original SO by loop-unrolling, and also extended the algorithm to be average optimal by integrating a text skipping property. Their resulting algorithms are fast shift-or (FSO), average optimal shift-or (AOSO), and fast average optimal shift-or (FAOSO). It must be noted that the AOSO and FAOSO algorithms require an additional parameter for their text skipping property, which is not straightforward to be computed in practice.

Navarro&Raffinot [14] implemented the nondeterministic backwards directed acyclic word graph in a bit-parallel fashion, which resulted the BNDM algorithm. BNDM includes an efficient shift mechanism, and its performance is much better than SO. The pseudocode of BNDM is given in Algorithm 2.

Peltola&Tarhio [16], and Holub&Durian [11] enhanced the performance of BNDM by simplifying the inner while loop, resulting with the fastest variants of the BNDM family called SBNDM and SBNDM2 respectively. Recently, Durian *et al.* [7] studied BNDM with q-grams.

Algorithm 2 BNDM(P,m,T,n)

```

1: for all  $i \in \Sigma$  do
2:    $B[i] = 0^W$ ;
3: for  $i = 0$  to  $m - 1$  do
4:    $B[P[i]] = B[P[i]] \& 0^{m-i-1}10^i$ ;
5:  $i = 0$ ;
6: while  $i \leq n - m$  do
7:    $j = m$ ;  $last = m$ ;  $D = 0^{W-m}1^m$ ;
8:   while  $D \neq 0$  do
9:      $D = D \& B[T[i + j]]$ ;
10:     $j = j - 1$ ;
11:    if  $D \& 10^{m-1} \neq 0^m$  then
12:      if  $j > 0$  then
13:         $last = j$ ;
14:      else
15:        Pattern detected beginning at  $i + 1$ ;
16:       $D = D \ll 1$ ;
17:       $i = i + last$ ;

```

The common property of both SO and BNDM type algorithms is that (See line 5 on both algorithms) each character $i \in \Sigma$ has a corresponding mask vector on which the locations of its occurrences in the input pattern are marked. In other words, each bit in a mask vector indicates if the pattern has that character at that position. Thus, the input string is required to be less than the computer word size to be efficiently addressed by the bits in the mask. Using more computer words clearly degrades the performance.

Some modifications have been offered to deal with long patterns on SO or BNDM class algorithms. In the original work of SO [3], a part of the long pattern shorter than or equal to W in length is scanned by the original algorithm, and in case of a match, the whole pattern is verified accordingly. This actually turns the algorithm into a filter.

Navarro&Raffinot [14] proposed to partition the pattern into consecutive pieces each of which has W characters except the last one that holds the remaining characters. The original BNDM algorithm is applied on each piece one by one. If the first piece matches, then the next pieces are scanned consecutively again with the original algorithm by shifting the window W characters until a mismatch detected. Note that in this case the maximum shift is W .

	0	1	2	3	4	5	6	7	8	9	10	11
0	a	b	a	a	b							
1		a	b	a	a	b						
2			a	b	a	a	b					
3				a	b	a	a	b				
4					a	b	a	a	b			
5						a	b	a	a	b		
6							a	b	a	a	b	
7								a	b	a	a	b

TABLE 1. The alignment matrix generated for the sample pattern $P = abaab$ with the assumption that computer word size $W = 8$.

Peltola&Tarhio [16] gave an alternative way of dealing with long patterns. They divided the pattern into $\lfloor \frac{m}{k} \rfloor$ pieces, each with length $k = \lfloor \frac{m-1}{w} \rfloor + 1$. The remaining $m - k \cdot \lfloor \frac{m}{k} \rfloor$ are left either at the beginning or at the end of the pattern. These k pieces are superimposed and then searched with the BNDM on the text. In case of a match a verification is performed accordingly. Note that in this case it is possible to make shifts longer than W . The authors state that their proposed method is efficient when the pattern is longer than $2.W$ in length for large alphabets. For small alphabets, using q-grams to generate superalphabets [8] seems a better choice.

All of the proposals concentrating on dealing long patterns are in fact patches of the original underlying SO or BNDM algorithms, and require the original algorithm to be tuned accordingly when the pattern is longer than W . This study introduces a unified way of bit-parallel exact searching for patterns of any lengths. Instead of marking the position of the characters, the bits in a computer word are used in a different way to achieve this goal.

3. BLIM Algorithm

Given a pattern P , let's imagine an alignment matrix A , where each row_i , $0 \leq i < W$, contains the pattern right shifted by i characters. Thus, A has W rows, and $ws = W + m - 1$ columns. Note that, throughout the study ws value will be referred as window size from now on. A sample alignment matrix corresponding to pattern $P = abaab$ is shown in Table 1 assuming computer word size $W = 8$.

The main idea of BLIM is to slide that ws length alignment matrix over the text, check if any possible placements of the input pattern exist in the current window via bitwise operations, and after completing the investigation of the current text portion $T[i \dots i + ws - 1]$ shift the window to the right by an amount

	0	1	2	3	4	5	6	7	8	9	10	11
a	FF	FE	FD	FB	F6	ED	DB	B7	6F	DF	BF	7F
b	FE	FD	FA	F4	E9	D3	A7	4F	AF	3F	7F	FF
c	FE	FC	F8	F0	E0	C1	83	87	8F	1F	3F	7F
d	FE	FC	F8	F0	E0	C1	83	87	8F	1F	3F	7F

TABLE 2. The mask matrix generated for sample pattern $P = abaab$, assuming alphabet $\Sigma = \{a, b, c, d\}$, and computer word size $W = 8$.

that is computed according to the immediate text character $T[i + ws]$ following the current window¹.

When the window is located on text $T[i \dots i + ws - 1]$, BLIM visits the characters $T[i + j]$ ($0 \leq j < ws$) in an order that was previously computed at the preprocessing stage. At each character visit, possible occurrences of the pattern are checked with a bitwise *and* operation by using a mask matrix that was again precomputed. Current window is slid right by the shift amount specified by the text character $T[i + ws]$ after the current investigation is over. Thus, a shift vector of alphabet size needs to be calculated at preprocessing also. In summary, at preprocessing stage BLIM calculates the mask matrix, the shift vector and decides on the scan order according to input pattern. The pseudo-code of BLIM is given in Algorithm 6.

3.1. Preprocessing Stage I: The Mask Matrix

Mask matrix consists of alphabet size σ rows, and ws columns. Each $Mask[ch][pos]$, where $ch \in \Sigma$, and $0 \leq pos < ws$, is a bit vector of W bits as $b_{W-1}b_{W-2} \dots b_1b_0$. The i th bit in $Mask[ch][pos]$ is set to 0, if observing that ch at position pos is not possible for the i character right shifted placement of the input pattern P . Otherwise, it is equal to 1, which indicates observing that ch at that pos does not violate the placement of the i character right shifted version of the pattern. The formal definition for the actual value of the b_i in $Mask[ch][pos]$ is as follows:

$$b_i = 0 \quad , \text{ if } (0 \leq pos - i < m \text{ and } (ch \neq p_{pos-i}))$$

$$b_i = 1 \quad , \text{ otherwise}$$

Algorithm 3 depicts the calculation of the mask matrix, and the example mask matrix generated for the sample pattern $P = abaab$ is shown in Table 2 in hexadecimal notation, assuming the alphabet of the text is $\Sigma = \{a, b, c, d\}$ with a computer word size $W = 8$.

For the sample pattern $P = abaab$, Table 3 depicts the calculation and meanings of bits in $Mask[b][6]$ explicitly. It is seen that right shift of the string by 0, 1, 2, 5, and 7 characters are appropriate, as specified by the corresponding bits

¹Performing shift according to the immediate text character following the current window was first proposed by Sunday in quick search algorithm [17], which is one of the fastest algorithms in BM family.

Bit Index i	7	6	5	4	3	2	1	0
pos- i	-1	0	1	2	3	4	5	6
P_{pos-i}		a	b	a	a	b		
Bit Value	1	0	1	0	0	1	1	1

TABLE 3. Calculation of the bit vector $Mask[b][6] = A7$ for $P = abaab$.

Character	Shift Value
a	9 = $12 - \max(0, 2, 3)$
b	8 = $12 - \max(1, 4)$
c	13 = $12 + 1$
d	13 = $12 + 1$

TABLE 4. Computing the shift values of letters for sample pattern $P = abaab$.

in $Mask[b][6]$. These possible alignments can be viewed on rows 0, 1, 2, 5, and 7 of Table 1. Note that, the observation of b at $T[i + 6]$ does not infer with the alignments beginning at $T[i]$, $T[i + 1]$, and $T[i + 7]$. Thus, the corresponding bits are set to 1.

Algorithm 3 ComputeMaskMatrix(P, m)

```

1: for  $j = 0$  to  $ws - 1$  do
2:   for all  $a \in \Sigma$  do
3:      $Mask[a][j] = 1^W$  ;
4:   for  $i = 0$  to  $W - 1$  do
5:      $tmp = 0^{W-i-1}10^i$  ;
6:     for  $j = 0$  to  $m - 1$  do
7:       for all  $a \in \Sigma$  do
8:          $Mask[a][i + j] \&= \sim tmp$  ;
9:          $Mask[P[j]][i + j] |= tmp$  ;

```

3.2. Preprocessing Stage II: The Shift Vector

BLIM uses the shift mechanism proposed in quick search of Sunday [17]. That is; the immediate text character following the window determines the actual shift amount. If that character is included in the pattern then $Shift[ch] = ws - k$, where $k = \max\{i; p_i = ch\}$, else $Shift[ch] = ws + 1$. On the same example pattern $P = abaab$, remembering that $ws = 12$, and $\Sigma = \{a, b, c, d\}$, the shift values of the letters are given in Table 4.

If one encounters the character a at $T[i + 12]$ while the window is located at $T[i \dots i + 11]$, then the pattern $abaab$ may begin at $T[i + 9]$, $T[i + 10]$, and $T[i + 12]$. Thus, the safe shift of a should be 9. Similarly, if $b = T[i + 12]$, then the shift value

is 8. The characters c and d do not occur in pattern P . Observation of those bad characters at $T[i + 12]$ directly yields the next attempt to begin from $T[i + 13]$, which means a shift value of 13.

Note that such a shift mechanism does not benefit from the characters investigated in the current window. This memoryless shift function may be replaced with a more sophisticated two-dimensional shift matrix such that $Shift[ch][pos]$ would indicate the safe shift amount when one observes ch at position pos . At each character visit, the shift value should be selected as the maximum of the current value and the value of the visited letter. Although theoretically less number of characters are visited in that case, the experiments showed that the time elapsed to update shift value at each character visit downgrades the speed. Thus, it is preferred to use the simple version instead.

Algorithm 4 ComputeShiftVector(P, m)

```

1:  $ws = W + m - 1$ ;
2: for all  $a \in \Sigma$  do
3:    $Shift[a] = ws + 1$ ;
4: for  $j = 0$  to  $m - 1$  do
5:    $Shift[P[j]] = ws - j$ ;

```

3.3. Preprocessing Stage III: Scan Order

Algorithm 5 ComputeScanOrder(m)

```

1:  $i = 0$ ;
2: for  $j = m - 1$  down to 0 do
3:    $k = j$ ;
4:   while  $k < ws$  do
5:      $ScanOrder[i] = k$ ;
6:      $k = k + m$ ;  $i = i + 1$ ;

```

The characters of the ws length window should be visited in such an order that checking all possible alignments in the current window be accomplished with minimum number of character accesses. If we traverse the window from right-to-left or left-to-right, we have to perform unnecessary checks in case of a leftmost or rightmost occurrence of the pattern. Actually, the optimum solution requires calculating the next position after each character visit, which slows down the algorithm considerably.

As a simple and efficient way of computing the scan order, BLIM algorithm checks the characters at positions $m - i, 2m - i, \dots, km - i$, where $km - i < ws$, for $i = 1, 2, \dots, m$ in order. As an example, assuming computer word size $W = 8$, the scan order of the sample pattern $P = abaab$ is 4, 9, 3, 8, 2, 7, 1, 6, 11, 0, 5, 10. The main idea behind this ordering is to investigate the window in such a way that at each character access maximum number of alignments is checked.

3.4. Main Search Loop

Algorithm 6 BLIM(P,m,T,n)

```

1: ComputeMaskMatrix(P,m);
2: ComputeShiftVector(P,m);
3: ComputeScanOrder(m);
4:  $k = i = 0$ ;
5: while  $i < n$  do
6:    $flag = Mask[T[i + ScanOrder[0]]][ScanOrder[0]]$ ;
7:   for  $j = 1$  to  $ws - 1$  do
8:      $flag \&= Mask[T[i + ScanOrder[j]]][ScanOrder[j]]$ ;
9:     if  $flag = 0^W$  then
10:      break;
11:   if  $flag \neq 0^W$  then
12:     for  $j = 0$  to  $W - 1$  do
13:       if  $flag \& 0^{W-j-1} 10^j$  then
14:         Pattern detected beginning at  $T[i + j]$ 
15:    $i = i + Shift[T[i + ws]]$ ;

```

BLIM algorithm, as sketched in Algorithm 6, has a very simple main loop in which the current text portion is investigated very fast by using only the bitwise *and* operator. When the window is located at $T[i \dots i + ws - 1]$, the *flag* variable is first set to the corresponding mask value of the text character $T[i + ScanOrder[0]]$ at position $ScanOrder[0]$. Remember that $ScanOrder[0]$ is always equal to $m - 1$. The next character to be visited is defined by $ScanOrder[1]$. The *and* operation is performed between the current flag and the mask of $T[i + ScanOrder[1]]$ defined for position $ScanOrder[1]$. The traversal of the characters continues till the *flag* becomes 0 or all the characters are visited. If *flag* becomes 0, this implies pattern does not exist on the window. Otherwise, one or more occurrences of the pattern are detected at the investigated text factor. In that case, the 1 bits of the *flag* tells the positions of occurrences exactly, e.g., say if the 3rd bit is 1, then pattern is detected at $T[i + 3]$. After completing the search on the current text factor, the window is slid right by the shift amount of the character at $T[i + ws]$.

A sample run of the BLIM algorithm is depicted in Table 5 assuming that pattern **abaab** is to be searched on a text factor $T[0 \dots 11] = \mathbf{ababaabaabab}$. The corresponding mask values can be reached from Table 2. At the end of that sample run, it is observed that *flag* is 00100100, which indicates pattern is detected at positions $T[2]$ and $T[5]$ conforming to bit indices (from least significant bit to most significant bit) that are 1. Note that after completion of the run, the window will be slid right according to the immediate text character following the window, e.g., the shift is 9, if $T[12] = \mathbf{a}$, or 13 if $T[12] = \mathbf{c}$ or \mathbf{d} .

	j	ScanOrder[j]	Mask[ch][pos]	flag
ababa <u>a</u> baabab	0	4	Mask[a][4] = 11110110	11110110
ababa <u>b</u> aabab	1	9	Mask[b][9] = 00111111	00110110
aba <u>b</u> aabaabab	2	3	Mask[b][3] = 11110100	00110100
ababa <u>a</u> abab	3	8	Mask[a][8] = 01101111	00100100
aba <u>b</u> aabaabab	4	2	Mask[a][2] = 11111101	00100100
ababa <u>a</u> abab	5	7	Mask[a][7] = 10110111	00100100
a <u>b</u> aabaabaabab	6	1	Mask[b][1] = 11111101	00100100
ababa <u>b</u> aabab	7	6	Mask[b][6] = 10100111	00100100
ababa <u>a</u> abab	8	11	Mask[b][11] = 11111111	00100100
<u>a</u> baabaabaabab	9	0	Mask[a][0] = 11111111	00100100
ababa <u>a</u> abab	10	5	Mask[a][5] = 11101101	00100100
a <u>b</u> aabaabaabab	11	1	Mask[b][1] = 11111101	00100100

TABLE 5. A sample run of BLIM on text factor **ababaabaabab** for searching **abaab**.

4. Complexity

The complexity of the mask matrix computation, being the sum of the initialization and the actual computation of the values, is $O(ws.\sigma + W.m.\sigma)$. Remembering $ws = W + m - 1$, this complexity turns into $O(\sigma(W + m - 1 + W.m))$, which approximates to $O(\sigma.m)$ as W is constant. The space requirement for mask matrix is $O(ws.\sigma)$. The computation of the shift vector requires $O(\sigma + m)$ operations and $O(\sigma)$ space. Time and space complexities are both $O(ws)$ for the scan order calculation.

At each attempt of aligning the investigation window over the text, BLIM algorithm inspects $\lceil \frac{ws}{m} \rceil$ characters at least, and ws characters at most. The maximum possible shift is $ws + 1$, where the minimum is W . In the best case the window is slid through the text with maximal shifts, and at each attempt minimum number of text accesses are performed. The time complexity is $O(\lceil \frac{n}{ws+1} \rceil \cdot \lceil \frac{ws}{m} \rceil)$, which actually converges to $O(n/m)$. On the opposite side, worst case occurs with minimal shifts and maximal character inspections, which implies a time complexity of $O(\lceil \frac{n}{W} \rceil \cdot ws)$ approximating $O(n.m/W)$.

For the average case analysis it will be assumed that the characters of the pattern and the text are uniformly distributed, and for the sake of simplicity all the characters of the pattern will be assumed to be distinct. The average time complexity is simply $O(n/\text{average shift})$. (*average character inspection per window*). Thus, we need to calculate average shift (AS) and average character inspection per attempt (ACI).

Shift is computed according to the single character succeeding the window. If that character does not exist in the pattern, shift is $ws + 1 = W + m$, else shift takes a value in between W and $W + m - 1$, e.g. if the character is the last character

of the pattern shift is W , if it is the one before the last, then shift is $W + 1$. Then the average shift computation is as in Equation 4.2.

$$AS = \frac{(\sigma - m).(W + m) + W + (W + 1) + \dots + (W + m - 1)}{\sigma} \quad (4.1)$$

$$= W + m - \frac{m.(m + 1)}{2.\sigma} \quad (4.2)$$

The probability that the input pattern P of length m exists in the ws size window is denoted by H , and is given in 4.4.

$$H = \frac{W.\sigma^{ws-m}}{\sigma^{ws}} \quad (4.3)$$

$$= W.\sigma^{-m} \quad (4.4)$$

When the pattern occurs in the investigated window, then ws number of character access will be performed. If not, then the corresponding number of character visits is in between W and ws , which on the average turns out to be $\frac{ws+W}{2} = W + \frac{m-1}{2}$. Based on these facts, the average character inspection per window may be defined as in Equation 4.8.

$$ACI = H.ws + (1 - H).(W + \frac{m-1}{2}) \quad (4.5)$$

$$= H.W + H.m - H + W + \frac{m-1}{2} - H.W - H.\frac{m-1}{2} \quad (4.6)$$

$$= (H + 1).\frac{m-1}{2} + W \quad (4.7)$$

$$= (W.\sigma^{-m} + 1).\frac{m-1}{2} + W \quad (4.8)$$

The average case time complexity being $O(ACI.\frac{n}{AS})$ is then

$$Avg. Case Complexity = O\left(\frac{n}{W + m - \frac{m.(m+1)}{2.\sigma}} \cdot [(W.\sigma^{-m} + 1).\frac{m-1}{2} + W]\right)$$

5. Handling Character Classes and Fixed Length Gaps

A common and very useful property of bit-parallel pattern matching algorithms is the easy handling of character classes and fixed length gaps in the input patterns. BLIM also deals with these cases as well by doing some changes in the preprocessing.

In line 7 of the algorithm 3, instead of performing *or* operation for a single character $P[j]$, the set of all characters that are let to appear in $P[j]$ can be processed in the same way. If $P[j]$ is defined to be empty, then line 7 should not be executed for it.

Similarly, while computing the shift vector, line 6 in algorithm 4 should be altered to contain all the characters defined for $P[j]$. In case of a gap at $P[j]$, this line will not be executed for this position.

	0	1	2	3	4	5	6
0	a	b	a	a			
1		a	b	a	a		
2			a	b	a	a	
3				a	b	a	a
4	b	b	a				
5		b	b	a			
6			b	b	a		
7				b	b	a	

TABLE 6. The alignment matrix generated for the multi-pattern set $S = \{abaa, bba\}$ assuming $W = 8$.

6. Multiple Pattern Case

Let $S = \{P_0, P_1, \dots, P_{R-1}\}$ be a set of R patterns. The length of each pattern is shown by m_i , for $0 \leq i < R$. The length of the shortest and longest patterns are denoted by $\tilde{m} = \min(m_0, m_1, \dots, m_{R-1})$, and $\hat{m} = \max(m_0, m_1, \dots, m_{R-1})$ respectively. Exact multiple pattern matching is finding the occurrences of patterns from set S on text T .

When compared with the single pattern case, computer word size limitation of the bit-parallel pattern matching algorithms is more severe in multi-pattern case as it tends to be more probable that the total length of multiple patterns will exceed W . If the total length of the patterns does not exceed W , the SO and BNDM variants can handle the situation properly by slight modifications. In other case, superimposing the pattern characters at the same positions yields a filter, and verification is performed wherever this filter is matched on the text.

BLIM offers a more proper way of dealing with multiple patterns in a unified bit-parallel fashion. Up to W number of patterns, whatever their sizes are, can be scanned simultaneously by the multi-pattern adaptation of BLIM.

The window size in multi-pattern case is $\tilde{ws} = Z + \hat{m} - 1$, where $Z = \lfloor \frac{W}{R} \rfloor$. Actually Z interprets the number of right shifted alignments for each pattern in the set. Assuming that the investigation window is aligned at text portion $T[i..i + \tilde{ws} - 1]$, the algorithm will check each pattern beginning at positions i to $i + Z - 1$.

Similar to the alignment matrix given in Table 1 for the single pattern case, Table 6 sketches the alignments of multiple patterns $S = \{abaa, bba\}$ by assuming $W = 8$. Thus, $R = 2$, $\hat{m} = \max\{4, 3\} = 4$, $\tilde{m} = \min\{4, 3\} = 3$, $Z = \lfloor \frac{8}{2} \rfloor = 4$, and $\tilde{ws} = 4 + 4 - 1 = 7$ for the sample set.

The computation of mask matrix for the multi-pattern case is given in Algorithm 7. Actually it is an extension of the single pattern case that passes over all patterns in given set S . Note that if W is not a multiple of R , then the initial $W - R \cdot \lfloor \frac{W}{R} \rfloor$ bits at each mask are set to 0 initially.

Algorithm 7 MComputeMaskMatrix($P_0, P_1, \dots, P_{R-1}, m_0, m_1, \dots, m_{R-1}$)

```

1: for  $j = 0$  to  $\tilde{w}s - 1$  do
2:   for all  $a \in \Sigma$  do
3:      $Mask[a][j] = 0^{W-R.Z}1^{R.Z}$ ; //Initialize mask matrix
4:    $b = 0$ ;
5:   for  $n = 0$  to  $R - 1$  do
6:     for  $j = 0$  to  $Z - 1$  do
7:        $tmp = 0^{W-b-1}10^b$ ;
8:       for  $k = 0$  to  $m_n - 1$  do
9:         for all  $a \in \Sigma$  do
10:           $Mask[a][k + j] \&= \sim tmp$ ;
11:           $Mask[P[n][k]][k + j] |= tmp$ ;
12:         $b = b + 1$ ;

```

In the alignment matrix, the shortest pattern has a corresponding character at the columns from 0 to $\tilde{m} + Z - 2$. The characters succeeding that position should be used for the calculation of the *shift* for safe movement of the window. As we now have more patterns implying more characters, instead of computing the shift based on a single character, it is more beneficial to take 2 characters into consideration. Thus, when the window is aligned with $T[j]$, shift is calculated according to the 2-gram $T[j + \tilde{m} + Z - 1]T[j + \tilde{m} + Z]$, which indicates that the shift is of size σ^2 in multi-pattern case. The pseudocode of calculation is given in Algorithm 8.

Algorithm 8 MComputeShift($P_0, P_1, \dots, P_{R-1}, m_0, m_1, \dots, m_{R-1}$)

```

1:  $v = \tilde{m} + Z - 1$ ;
2: for all  $a \in \Sigma$  and  $b \in \Sigma$  do
3:    $Shift[a][b] = v + 2$ ;
4: for  $i = 0$  to  $R - 1$  do
5:   for  $j = Z$  to  $v + 1$  do
6:     if  $j + m_i = v + 1$  then
7:       for all  $a \in \Sigma$  do
8:          $Shift[P[i][m_i - 1]][a] = \min(Shift[P[i][m_i - 1]][a], j)$ ;
9:     else if  $j = v + 1$  then
10:      for all  $a \in \Sigma$  do
11:         $Shift[a][P[i][0]] = \min(Shift[a][P[i][0]], j)$ ;
12:     else
13:       for all  $a \in \Sigma$  do
14:          $Shift[P[i][v - j]][P[i][v - j + 1]] =$ 
15:            $\min(Shift[P[i][v - j]][P[i][v - j + 1]], j)$ ;

```

The scan order computation for the multi-pattern case is given in Algorithm 9. Actually, the ordering mechanism is preserved for the shortest pattern, and the

remaining positions beyond $\tilde{m} + Z - 2$ are simply concatenated to the list. The corresponding scan order for the sample given in Table 6 is then $ScanOrder = \{2, 5, 1, 4, 0, 3, 6\}$.

Algorithm 9 MComputeScanOrder(m_0, m_1, \dots, m_{R-1})

```

1:  $i = 0$ ;
2: for  $j = \tilde{m} - 1$  down to 0 do
3:    $k = j$ ;
4:   while  $k < \tilde{w}s$  do
5:      $ScanOrder[i] = k$ ;
6:      $k = k + \tilde{m}$ ;  $i = i + 1$ ;
7:    $j = \tilde{m} + Z - 1$ ;
8:   while  $j < \tilde{w}s$  do
9:      $ScanOrder[i] = j$ ;
10:     $j = j + 1$ ;  $i = i + 1$ ;

```

Main search loop for the multi-pattern case is identical with the single pattern case. The bits that are set to 1 in the flag after the completion of the investigation on the current text window indicates which patterns from the set is observed and at which positions. For example, in the sample multi-pattern case, the bits 0, 1, 2, 3 indicate that pattern *abaa* observed beginning from $T[j], T[j+1], T[j+2], T[j+3]$, and similarly bits 4, 5, 6, 7 imply pattern *bba* beginning from the same locations, assuming the window is located at $T[j]$. Thus, a slight modification is required at line 16 in main BLIM skeleton sketched in Algorithm 6.

7. Experimental Results

BLIM is compared with alternative bit-parallel algorithms on natural language and DNA sequence data. The *enwik8* corpus that comprises the first 100MB of wikipedia in English is used in natural language benchmarks, and the 90MB *hs13.chr* file corresponding to the plain ascii form of the human chromosome 13 from Manzini's DNA corpus is used for the tests on DNA sequence.

The alternatives of BLIM are examined in two groups as SO family and BNDM family. SO family contains the algorithms original shift-or [3] and its variants introduced in [10, 9] as fast shift-or (FSO), average optimal shift-or (AOSO) and the fast average optimal shift-or (FAOSO). The BNDM family is composed of original BNDM [14], and its two variants as SBNDM [16], and SBNDM2 [11]. The implementations of are by the original authors of the algorithms except the SO, and BNDM, which are taken from [4]. Note that AOSO and FAOSO algorithms require an additional parameter k (see [10, 9] for details). The best value of k is experimental, and the best reported values from [10] for each length of patterns on DNA and English text are used in the experiments.

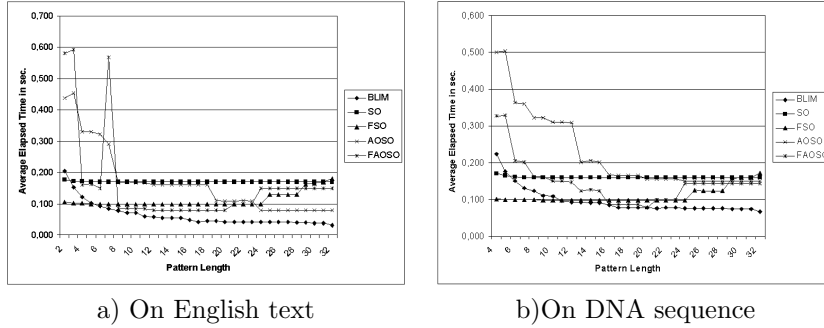


FIGURE 1. BLIM versus SO family.

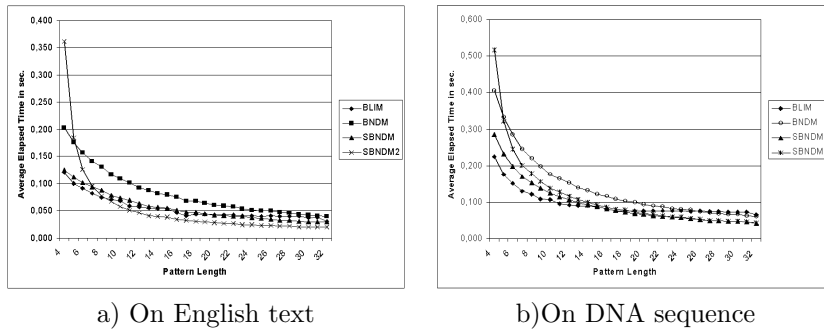


FIGURE 2. BLIM versus BNDM family.

The machine on which the experiments are conducted has a Xeon 2.4 Ghz 32-bit processor with 3GB memory. The codes are compiled with *gcc 4.1.2* with full optimization (*-O3*) turned on running on Gentoo Linux distribution.

At each run of the experiment, 100 patterns for each length of 2 to 32 on natural language text, and 4 to 32 on DNA sequence data are randomly selected from the corresponding sources. Each pattern is scanned via the algorithms for 5 times, and the minimum user time among them is recorded.

Figure 7 depicts the performance of BLIM against the SO family on natural language and DNA data. Similarly, Figure 7 shows the comparison with the BNDM family. Figure 7 indicates the overall performances of algorithms.

On DNA sequence, the performance of FSO is the best for patterns up to length 12, and SBNDM is a better choice for longer ones. When the overall performances are considered, BLIM and SBNDM, being very compatible, are very effective.

On natural language text, FSO is the winner for the short patterns, but now up to length 6. On longer patterns, SBNDM, SBNDM2, and BLIM are effective

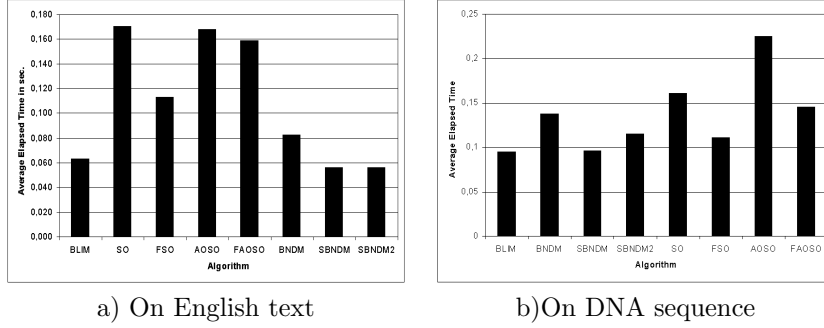


FIGURE 3. Overall performance comparison among bit-parallel pattern matching algorithms.

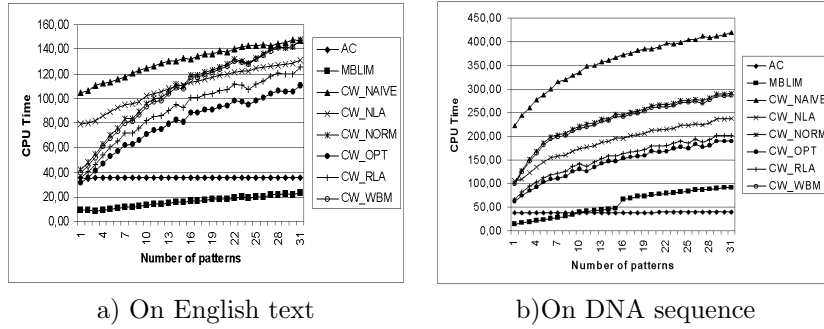


FIGURE 4. Multiple BLIM (MBLIM) versus multiple pattern matching algorithms from SPARE Parts [19] toolkit.

showing very similar performances. On the average, BLIM is slightly worse than SBNDM, and SBNDM2 in this case.

Note that on both DNA and English text, the overall performance of BLIM is very good, although there exists better choices for some specific length intervals. That is mainly because the performance of the BLIM does not vary much with the length, unlike the other algorithms of this genre except SO, and FSO. Thus, when combined with the property of not having a word size limitation, BLIM is a good candidate in practice as a general pattern matching tool.

The multi-pattern BLIM (MBLIM) is also compared with some of the classic multiple pattern matching algorithms via the SPARE parts [19] toolkit. The algorithms from that package are mainly the Aho-Corasick (AC) [1] and Commentz-Walter [18] variants as naive (CW_NAIVE), normal (CW_NORM), optimum (CW_OPT), no look ahead (CW_NLA), right look ahead (CW_RLA), and weak Boyer-Moore (CW_WBM).

Up to 32 patterns of length 2 to 20 on English text, and 4 to 32 on DNA sequence are randomly selected from the corresponding files. With the same settings for the single pattern case, experiments are conducted. The results are depicted in Figure 7. Note that especially on natural language text, the performance of MBLIM is promising.

8. Conclusion

This study introduced a new bit-parallel pattern matching algorithm, given name BLIM, that does not restrict the input patterns to be shorter than the computer word size. Handling character classes and fixed-length gaps along with the multiple pattern case are also examined. Experimental results indicate that the performance of the proposed scheme is compatible with the previous algorithms of this genre with an additional value of overcoming word size limitation.

The main contribution of this study is not a new more fast string matching algorithm, but a new approach identifying the use bits in a different manner. Each bit in the proposed scheme represents an event, and the observations performed during the investigation alters these events according to the precomputed masks. In exact pattern matching problem examined in this study, the events corresponds the alignments of the patterns in a window, and the observations are actually the characters accessed. The future works on the topic will compromise the use of the same model in other problems such as the *constraint satisfaction* or similarity search by *seeds*.

Acknowledgment

The author thanks J. Tarhio, K. Fredriksson, and T. Lecroq for sharing their codes and comments.

References

- [1] A.V. Aho and M.J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18:333–340, June 1975.
- [2] A. Apostolico and Z. Galil, editors. *Pattern Matching Algorithms*. Oxford University Press, 1997.
- [3] R.A. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.
- [4] C. Charras and T. Lecroq. *Handbook of exact string matching algorithms*. King's Collage Publications, 2004.
- [5] L. Cleophas, B.W. Watson, and G. Zwaan. A new taxonomy of sublinear keyword pattern matching algorithms. Computer Science Report 04–07, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, April 2004.

- [6] M. Crochemore and W. Rytter. *Jewels of stringology*. World Scientific Publishing, 2003.
- [7] B. Durian, J. Holub, H. Peltola, and J. Tarhio. Tuning bndm with q-grams. In *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX09)*, pages 29–37, New York City, January 3 2009.
- [8] K. Fredriksson. Faster string matching with super-alphabets. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE'2002)*, LNCS 2476, pages 44–57. Springer-Verlag, 2002.
- [9] K. Fredriksson and Sz. Grabowski. Practical and optimal string matching. In *Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE'2005)*, LNCS 3772, pages 374–385. Springer-Verlag, 2005.
- [10] K. Fredriksson and S. Grabowsky. Average-optimal string matching. *Journal of Discrete Algorithms (Accepted)*, 2009.
- [11] J. Holub and B. Durian. Fast variants of bit parallel approach to suffix automata. Unpublished Lecture, University of Haifa, April 2005.
- [12] M. O. Külekcı. A method to overcome computer word size limitation in bit-parallel pattern matching. In Seok-Hee Hong, Hiroshi Nagamochi, and Takura Fukunaga, editors, *Proceedings of 19th International Symposium on Algorithms and Computation, ISAAC'2008*, volume 5369 of *Lecture Notes in Computer Science*, pages 496–506, Gold Coast, Australia, December 2008. Springer Verlag.
- [13] M. O. Külekcı. Overcoming computer word size limitation in bit-parallel pattern matching. Talk given in LSD&LAW'09, London Stringology Days & London Algorithmic Workshop, King's College, London, UK, February 2009.
- [14] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithms*, 5(4):1–36, 2000.
- [15] G. Navarro and M. Raffinot. *Flexible pattern matching in strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.
- [16] H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In *LNCS 2857, Proceedings of SPIRE'2003*, pages 80–94, 2003.
- [17] D.M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.
- [18] B.W. Watson. A new family of Commentz-Walter-style multiple-keyword pattern matching algorithms. In *Proceedings of The Prague Stringology Club Workshop*, pages 71–76, 2000.
- [19] B.W. Watson and L. Cleophas. SPARE parts: A C++ toolkit for string pattern recognition. *Software – Practice and Experience*, 34:697–710, 2004.
- [20] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

M. Oğuzhan Külekci
TÜBİTAK - UEKAE
P.K. 74
41470 Gebze,Kocaeli
Turkey
e-mail: kulekci@uekae.tubitak.gov.tr