

---

# Fast Pattern Matching via $k$ -bit Filtering Based Text Decomposition

M. OĞUZHAN KÜLEKCI<sup>1</sup>, JEFFREY SCOTT VITTER<sup>2</sup> AND BOJIAN XU<sup>3</sup>

<sup>1</sup> *National Research Institute of Electronics & Cryptology, TURKEY*

<sup>2</sup> *Department of Electrical Engineering and Computer Science, University of Kansas, USA*

<sup>3</sup> *The Information and Telecommunication Technology Center, University of Kansas, USA*

*Email: kulekci@uekae.tubitak.gov.tr*

---

This study explores an alternative way of storing text files to answer exact match queries faster. We decompose the original file into two parts as filter and payload. The filter part contains the most informative  $k$  bits of each byte, and the remaining bits of the bytes are concatenated in the order of appearance to generate the payload. We refer to this structure as  $k$ -bit filtered format. When an input pattern is to be searched on the  $k$ -bit filtered structure, the same decomposition is performed on the pattern. The  $k$  bits from each byte of the pattern form the pattern filter bit sequence, and the rest is the payload. The pattern filter is first scanned on the filter part of the file. At each match position detected in the filter part, the pattern payload is verified against the corresponding location in the payload part of the text. Thus, instead of searching an  $m$ -byte pattern on an  $n$ -byte text, first  $k \cdot m$  bits are scanned on  $k \cdot n$  bits, followed by a verification of  $(8 - k) \cdot m$  bits on the respective locations of the matching positions. Experiments conducted on natural language texts, plain ASCII DNA sequences, and random byte sequences showed that the search performance with the proposed scheme is on average two times faster than the tested best exact pattern matching algorithms. The highest gain is obtained on plain ASCII DNA sequences. We also developed an effective bitwise pattern matching algorithm of possible independent interest within this study.

*Keywords: Pattern Matching, String Search, Filtering, File Structure*

*Received 15 July 2010; revised 09 October 2010*

---

## 1. INTRODUCTION

A file is a sequence of bytes stored in digital media, such as hard disks, DVDs, or flash memories. Currently text is stored on digital media as it would be written to an ordinary paper notebook. The main motivation of this study is to explore an alternative way of storing text files that will result in a boost in search performance, with the restriction that the size of the converted file does not exceed the original one.

Exact string matching, which is simply finding all the occurrences of a given pattern on a text, is one of the deeply studied problems in computer science. The topic may be investigated in two classes regarding to using an index structure or not over the text during the search process [1].

When the pattern to be queried is given before the text arrives, the input pattern is preprocessed, but the text on which it will be searched has no such prior processing. As an example, let's assume we are given a person's name, and need to check if that name exists in the passenger lists of some flights. Since the passenger lists are arriving dynamically and the name we are

looking for is static, we are supposed to preprocess the name string so that we can perform a speedy scan on a later arriving text. There exist many algorithms [2] devoted to improving efficiency for particular cases regarding the size of the alphabet and the type of the text (natural language, biological sequences, random files) along with the lengths of the queried patterns.

On the other side, if the text is available before the run time of the search process, then it can be preprocessed in such a way that a received query is answered in a time cost proportional to the pattern's length and the number of its occurrences by building an index data structure, which is based on mainly subword graphs, suffix trees, and suffix arrays [3], over the content of the text. A typical example of the case is the search engines, which are indexing the available data beforehand and locating the queried patterns as they arrive via that index. Although the pattern length is also a factor of search performance, the point here is that the answer theoretically gets ready as soon as the reading of the pattern is finished.

The cost of the gain via indexing is the large space

consumption [4] of those structures in addition to the heavy procedure of their constructions in practice.

This study attempts to enhance pattern matching performance without creating an index, but storing the files in a different format, named *k-bit filtered file* format. The conversion of an ordinary file into *k-bit filtered* format is a light operation compared with the constructions of the indexing structures, and the resultant file is exactly of the *same size* of the original one. Experiments showed that the search performance on these converted files is on average twice faster than the tested best exact pattern matching algorithms.

In pattern matching, filtering approaches are powerful tools. Following the long standing success of the Wu and Manber's algorithm [5] especially on approximate single/multiple pattern matching on natural language texts, recently Lecroq [6] proposed the *q-hash* algorithm of this genre, which is shown to be very effective especially on small alphabets.

Filtering algorithms compute the hash of the input pattern and create other necessary structures accordingly such as the shift table at the preprocessing stage. While scanning the text, the hash of the investigated window is computed, and a full verification of the pattern is performed if that value coincides with the previously calculated hash of the pattern. Filtering is an elegant way of searching as calculating the hash via an easy-to-compute function is much more speedy than comparing the pattern with the text.

The main idea of this study is to move the most informative bits of the bytes to the beginning of the file and then use those bits as a filter. When a query is received, the corresponding bit sequence of the bytes of the input pattern is first scanned on that filter, and matching positions are verified with the rest of the bits accordingly.

The work presented in this paper is a bit-oriented approach rather than the classical byte-oriented approaches, so bit vector matching is in central of the whole study. Matching on bit sequences instead of byte sequences has been explored in previous studies. Klein and Ben-Nissan [7] adapted the Boyer-Moore algorithm [8] on bit vectors. Kim *et al.* [9] proposed the FED algorithm especially for searching on 2-bit encoded DNA sequences, but being applicable to any bit sequence search also. Faro and Lecroq introduced various algorithms named as BSKS [10], BHM [10], and the most recent BFL [11] algorithm, which is an adaptation of BNDM [12]. Although it is possible to integrate any efficient algorithm of this genre into the proposed filtering scheme, a variant of the Külekci's BLIM [13] algorithm is introduced and used within this study to fulfill the requirements of the filtering scheme in a fast and flexible manner.

The outline of this paper is as follows. Section 2 introduces the *k-bit filtered file* format. Section 3 covers the search process on the new format including the preprocessing steps, decomposition of the input pattern,

and how filtering and verification operations are achieved. Experimental comparisons with alternative algorithms are discussed in section 4 followed by the conclusion.

## 2. *K-BIT FILTERED FILE* FORMAT

Let file  $F$  of size  $n$  bytes be denoted by  $F = s_1s_2s_3 \dots s_n$ , where each byte  $s_i$ ,  $1 \leq i \leq n$ , is composed of eight bits as  $s_i = b_1^i b_2^i \dots b_8^i$ .

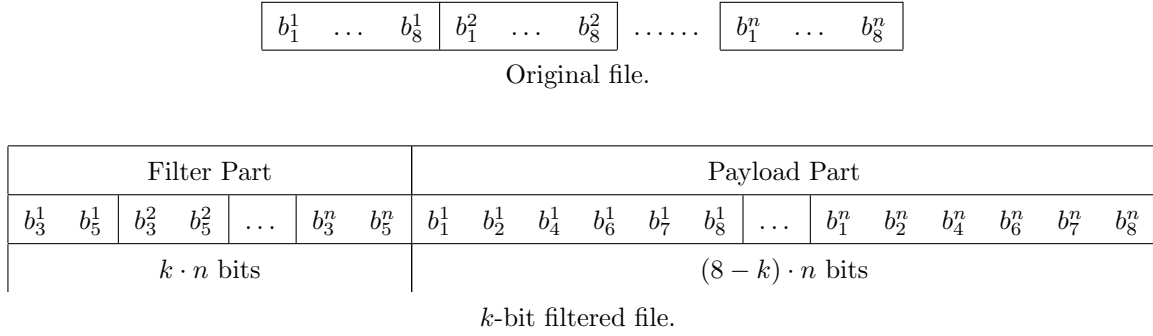
The input representing the number of bits that will be extracted from each byte is denoted by  $k$ , and let  $R$  contain the indices of the most informative  $k$  bits. Selection process of those most informative bits will be discussed below soon. The bits corresponding to the indices given in  $R$  are extracted from each byte  $s_i$  and are stored as a sequence of bits preserving the order of appearance. This bit stream is placed at the beginning of the new file, and the remaining bits of each  $s_i$  are concatenated to this stream again preserving the order of appearance in the original file. Figure 1 denotes the conversion of a sample file into its *k-bit filtered* format, assuming  $k = 2$  and the indices of the most informative  $k$  bits are  $R = \{3, 5\}$ .

The  $i^{th}$  character  $s_i$ ,  $1 \leq i \leq n$ , in the original file has its corresponding  $k$ -bit filter beginning at bit position  $(i - 1) \cdot k + 1$  on the respective  $k$ -bit filtered file. Its remaining bits begin at bit position  $[n \cdot k + (i - 1) \cdot (8 - k) + 1]$ , since there exist  $k$  bits of each of all  $n$  bytes at the beginning, and  $(8 - k)$  bits of each of the previous  $(i - 1)$  bytes preceding it.

The first step while converting a file into its *k-bit filtered* format is to find out the indices of the most informative  $k$  bits among the bytes of that file. It is a known fact from the information theory [14] that the information carried by a sequence is inversely proportional to its compression ratio. In other words, the amount of information on a stream can be measured by compressing the stream and taking the inverse of the compression ratio.

With that in mind, let's assume eight sequences, each of which is composed according to the bits appearing at positions one to eight of each byte. When these eight sequences are individually compressed and sorted according to their sizes in descending order, this order also represents the amount of the information content contained in the corresponding bit streams. For example, if the descending order of the sizes of the eight compressed bit sequences is obtained as  $\{5, 3, 1, 6, 8, 7, 2, 4\}$ , then the most informative bit in each byte is the fifth, and the next most informative one is the third. If  $k = 2$  is given, then the third and fifth bits of each byte are moved to the beginning of the file according to the scheme shown in figure 1.

It is also possible to select the indices of the  $k$  bits via some other techniques, such as simply counting the 0/1 ratio, or measuring the variances of the bit sequences, or even choosing  $k$  random indices. In fact, within the



**FIGURE 1.** Sample file  $F$  is converted to its  $k$ -bit filtered file format, assuming  $k = 2$  and  $R = \{3, 5\}$ .

proposed methodology, each byte in the file is down sampled to  $k$  bits, and if the selected bits are not the most informative ones, the cost will be a higher number of verification requests during the search process. Thus, finding the most informative bits is an important step, and using the compressed sizes as an indicator makes the filter powerful, although it adds an extra time and space complexity caused by the used compression algorithm.

The time required for converting an ordinary file into its  $k$ -bit filtered format is the total time cost for the most informative  $k$  bits selection process and the actual replacement process. Since the movement of the bits is a linear-time operation, and if a linear-time compression scheme is used for selecting the  $k$  bits, the whole process requires two passes over the original file. In the first pass the bit selection is done, and in the second phase the actual bit movement is performed.

### 3. SEARCHING PATTERNS ON A $K$ -BIT FILTERED FILE

In practice, an input pattern  $P$  of length  $m$  can be viewed as a sequence of bytes  $p_1 p_1 \dots p_m$ , where each  $p_i$ ,  $1 \leq i \leq m$ , is a sequence of eight bits,  $p_i = t_1^i t_2^i \dots t_8^i$ . Given the pattern  $P$  and the set  $R$  of  $k$  indices, the search process begins with decomposing the pattern into two pieces as the pattern filter  $PF$  and the pattern payload  $PL$ .  $PF$  is formed by extracting and concatenating the bits at positions  $R[1] \dots R[k]$  from each  $p_i$ , for  $i = 1$  to  $m$ . Similarly, the concatenation of the remaining bits other than the filter ones forms the  $PL$ . Figure 2 depicts this decomposition process on a sample with the assumption that  $k = 2$  and  $R = \{3, 5\}$ .

Following this segmentation,  $PF$  is searched on the filter part of the  $k$ -bit filtered file, which occupies the initial  $k \cdot n$  bits. In case of possible matches at some appropriate bit positions, the location of the corresponding payload is computed and the  $PL$  is verified against the value at that position.

If  $PF$  is found to begin at bit position  $f = k \cdot h + 1$  on the  $k$ -bit filtered file, which means there may be a possible match with the  $(h + 1)^{th}$  character of the

original file, then  $PL$  should to be verified on the corresponding bit position  $l = k \cdot n + h \cdot (8 - k) + 1$ . The calculation of the  $l$  comes from the fact that the payload part of the file begins after the  $(k \cdot n)$ -bit filter part, and to reach the payload of the  $(h + 1)^{th}$  byte, one needs to pass over the payloads of the previous  $h$  characters, which makes a total of  $h \cdot (8 - k)$  bits.

#### 3.1. Preprocessing for possible alignments of pattern payload $PL$

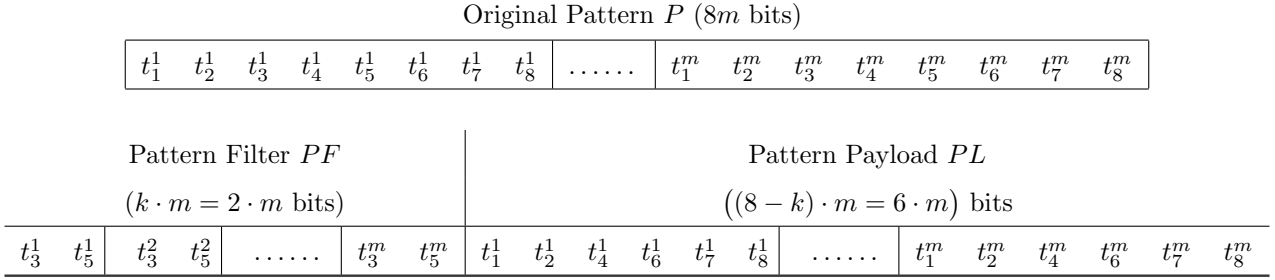
The position of the byte containing the  $l^{th}$  bit can be simply computed with  $byteid = \lceil l/8 \rceil$ .  $PL$  will be verified on that byte beginning from bit position  $8 - (byteid \cdot 8 - l)$ . As that bit position has a potential to take any value from one to eight, prior to actual scanning, all eight possible placements of the  $PL$  are compiled into a table, which is similar to the one used in the FED algorithm[9].

Let  $PL_i$  denote the  $PL$  starting at bit position  $i$ ,  $1 \leq i \leq 8$ , on a byte. Each  $PL_i$  is represented by a four-tuple defined as  $[M_f^i, M_l^i, D^i, C_i]$ . The mask for the first byte of verification sequence is  $M_f^i$  that includes 0s at bit positions smaller than  $i$  and 1s for the others. For example, if  $i = 3$ , meaning  $PL$  begins at the third leftmost bit, the corresponding  $M_f^3 = 00111111$ . Similarly,  $M_l^i$  is the last byte mask. The bit position where  $PL$  ends is  $x = 8 - ((i + m - 1) \bmod 8)$ . Thus, bits coming after  $x$  should be 0, and the rest are to be 1, e.g., assuming  $i = 3$  and  $m = 10$ , we have  $M_l^3 = 11110000$ .  $D_i$  is an array of  $C_i$  bytes holding the actual values of verification.

Figure 3 shows the  $PL_i$  table computed for a sample  $PL = 01100101101110$ . Note that the bold 0s depict the padded bits, which do not have an effect on the verification process, since they will be suppressed by the masks  $M_f^i$  and  $M_l^i$ .

#### 3.2. Matching on the filter part

In  $k$ -bit filtered file structure, appropriate bit positions, on which the pattern filter  $PF$  should be matched, depend on the value  $k$ . While scanning the  $(k \cdot m)$ -bit  $PF$  on the initial  $k \cdot n$  bits of the file,  $PF$  can only



**FIGURE 2.** Decomposing an input pattern  $P$  into filter  $PF$  and payload  $PL$ , assuming  $k = 2$  and  $R = \{3, 5\}$ .

$i$	$M_f^i$	$D_i$			$M_l^i$	$C_i$
1	11111111	01100101	10111000		11111100	2
2	01111111	00110010	11011100		11111110	2
3	00111111	00011001	01101110		11111111	2
4	00011111	00001100	10110111	00000000	10000000	3
5	00001111	00000110	01011011	10000000	11000000	3
6	00000111	00000011	00101101	11000000	11100000	3
7	00000011	00000001	10010110	11100000	11110000	3
8	00000001	00000000	11001011	01110000	11111000	3

**FIGURE 3.** The  $PL_i$  list of the sample  $PL = 01100101101110$ .

begin at a bit position  $f$  such that  $f = k \cdot h + 1$ , for some integer  $h$ ,  $0 \leq h < n$ , since the  $k$ -bit filter corresponding to the  $(h+1)^{th}$  byte of the original file begins at position  $f$ . As an example, if  $k = 2$ ,  $PF$  must be searched at bit positions  $1, 3, 5, \dots, k \cdot (n - 1) + 1$ . Aiming to fulfill this restriction in a fast and flexible way, a variant of the Külekci's BLIM algorithm [13] is adapted in this study.

The given pattern filter  $PF$  can begin at bit positions one to eight in a byte. Some of those beginnings are not appropriate depending on the value of  $k$ . Let's investigate the case for  $k = 1$ , which makes every position possible. Figure 4 shows all possible alignments of the sample  $PF = 11100100001101$ , where X denotes the don't care bits that can take the value of both 1 and 0. The length of the window to be investigated for  $PF$  is  $wl = 1 + \lceil (m - 1)/8 \rceil = 3$  bytes.

A mask matrix is computed according to the alignments of  $PF$ . Each  $Mask[c][p]$  is composed of eight bits, where  $i^{th}$  bit represents whether  $PF$  starting at bit position  $i$  is appropriate when byte  $c$ ,  $0 \leq c < 255$ , is observed at byte position  $p$ ,  $1 \leq p \leq wl$ , in the current window. For example, according to the sample given in figure 4, the occurrence of bit sequence 01010111 on the first byte of the investigation window fits with the alignments beginning from bit positions 6, 7, and 8. Thus, the corresponding mask is  $Mask[01010111][1] = 00000111$ .

Following the construction of the mask matrix, a bitwise AND operation between the mask and a value computed according to  $k$  is performed, since some of the alignments are not appropriate according to the value of  $k$ . This value is simply marking the possible places

---

**Algorithm 1** SearchOnKbitFilteredFile( $P, m, T, n$ )

---

- 1: Decompose  $P$  into  $PF$  and  $PL$ ;
  - 2: Calculate  $Mask$  for  $PF$ ;
  - 3: Calculate  $Shift$  for  $PF$ ;
  - 4:  $filter\_end \leftarrow \lceil (k \cdot n)/8 \rceil$ ;
  - 5:  $i \leftarrow 1$ ;
  - 6: **while** ( $i \leq filter\_end$ ) **do**
  - 7:    $flag \leftarrow Mask[T[i]][1]$ ;
  - 8:   **for** ( $j \leftarrow 2$ ;  $flag$  and  $j < (wl + 1)$ ;  $j++$ ) **do**
  - 9:      $flag \leftarrow flag \text{ AND } Mask[T[i + j]][j]$ ;
  - 10:   **if** ( $flag$ ) **then**
  - 11:     **for** ( $b \leftarrow 1$ ;  $b < 9$ ;  $b++$ ) **do**
  - 12:       **if** (bit  $b$  is set in the  $flag$ ) **then**
  - 13:          $pos \leftarrow (i - 1) \cdot 8 + b$ ;
  - 14:         **if** (VerifyPL( $pos$ ) = true) **then**
  - 15:         pattern is detected!
  - 16:   **repeat**
  - 17:      $i \leftarrow i + Shift[T[i + wl - 1]]$ ;
  - 18:   **until** ( $Mask[T[i + 1][2]] \neq 0$ ) or ( $i > filter\_end$ )
- 

of occurrence. As an example, each  $Mask[c][p]$  is ANDed with 10101010, if  $k = 2$ , and with 10001000, if  $k = 4$ .

A shift table is also computed based on the Horspool's shift mechanism [15]. The last byte of the investigation window determines how many bytes the window will be slid to the right for the next attempt. Continuing with the same example above, the shift amount is 1, 2, or 3, if one observes bit sequences 01000011, 11100100, and 00000000 respectively in the last byte of the investigation window.

	1								2								3							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
1	1	1	1	0	0	1	0	0	0	0	1	1	0	1	X	X	X	X	X	X	X	X	X	X
2	X	1	1	1	0	0	1	0	0	0	0	1	1	0	1	X	X	X	X	X	X	X	X	X
3	X	X	1	1	1	0	0	1	0	0	0	0	1	1	0	1	X	X	X	X	X	X	X	X
4	X	X	X	1	1	1	0	0	1	0	0	0	0	1	1	0	1	X	X	X	X	X	X	X
5	X	X	X	X	1	1	1	0	0	1	0	0	0	0	1	1	0	1	X	X	X	X	X	X
6	X	X	X	X	X	1	1	1	0	0	1	0	0	0	0	1	1	0	1	X	X	X	X	X
7	X	X	X	X	X	X	1	1	1	0	0	1	0	0	0	0	1	1	0	1	X	X	X	X
8	X	X	X	X	X	X	X	1	1	1	0	0	1	0	0	0	0	1	1	0	1	X	X	X

FIGURE 4. Possible alignments of sample  $PF = 11100100001101$ .

The main search operation on a  $k$ -bit filtered file structure is sketched in algorithm 1. The inner *repeat* loop performs a speedy shift mechanism. After the detection of the shift amount, prior to beginning a whole scan of the current window, it simply checks the mask of the second byte, as this position includes less don't cares especially on long patterns. If the corresponding mask is zero, then the algorithm simply continues with shifting.

When a match is detected in the filter part, the bits of the *flag* is investigated to find the exact bit position of the hit. The verification routine is called next to compare the pattern payload  $PL$  in the corresponding position of the file's payload part. If verification returns true, then the occurrence of the queried pattern is reported.

The worst case time complexity of searching  $k \cdot m$  bits on  $k \cdot n$  bits is  $O(\lceil (k \cdot n) / 8 \rceil \cdot wl)$ . This is the case where on each attempt all the bytes involved in the  $wl$ -length window is checked, followed by one-character right shift all the time. In the best case, each alignment of the investigation window accesses only one character, and the shift amount is always maximal being equal to  $wl$ , so the complexity is  $O(\lceil (k \cdot n) / 8 \rceil / wl)$ . The preprocessing of mask creation is quadratic as it requires to traverse a two-dimensional matrix, and shift table construction is just linear of the alphabet size.

The total time needed by the whole search process is simply the time required to search  $k \cdot m$  bits on  $k \cdot n$  bits plus the number of verification requests multiplied by the verification time. When the length of the pattern filter  $PF$  increases, the number of verification requests decreases as it is less probable to observe a long binary sequence. Thus, the time converges to the scanning duration of the  $PF$  filter on the file filter part.

### 3.3. Verification on the payload part

The verification of the  $PL$  on the payload of the  $k$ -bit filtered file is performed by using the  $PL_i$  structures computed in preprocessing. Assuming a match is reported by the filter on the  $f^{th}$  bit, the process is depicted in algorithm 2. First the actual bit position  $l$ , on which the  $PL$  is expected to begin is calculated. The

byte address of  $l$ , and the bit number on that byte are compiled into  $a$  and  $i$  respectively. Note that  $i$  specifies the  $PL_i$  that fits to the current alignment of  $PL$ . As the first and last bytes of  $D_i$  most probably include some don't cares, first the bytes in between are compared. If all matched, the first and last bytes are masked with  $MF_i$  and  $ML_i$ , and are checked against the values in  $D_i[1]$  and  $D_i[C_i]$  respectively. In case of a success at the end of that operation, the existence of pattern  $P$  is reported.

---

#### Algorithm 2 VerifyPL( $f$ )

---

```

1:  $h \leftarrow (f - 1) / k$ ;
2:  $l \leftarrow k \cdot n + h \cdot (8 - k) + 1$ ;
3:  $a \leftarrow \lceil l / 8 \rceil$ ;
4:  $i \leftarrow 8 - (a \cdot 8 - 1)$ ;
5:  $q \leftarrow 2$ ;
6: while ( $q < C_i$  and  $D_i[q] = T[a + q - 1]$ ) do
7:    $q++$ ;
8: if ( $q = C_i - 1$ ) then
9:   if ( $(T[a] \text{ AND } MF_i) = D_i[1]$ ) then
10:    if ( $(T[a + C_i - 1] \text{ AND } ML_i) = D_i[C_i]$ ) then
11:      return true;
12: return false;
```

---

## 4. EXPERIMENTAL RESULTS

Tests were conducted on natural language texts, plain ASCII DNA sequences, and random sequences with uniform probability distribution. The *enwik8*<sup>4</sup> corpus and Manzini's DNA corpus<sup>5</sup> are the sources of natural language and DNA sequences used in the experiments. Random files are generated via the standard `rand()` library function of C with `srand(time())` seeds.

During the experiments for a given text, the text file is saved in proposed  $k$ -bit filtered format for  $k = 1, 2, 4$ . The selection of the most informative bit-positions are performed according to the compression ratios of

<sup>4</sup>The *enwik8.txt* file is the subject of the Hutter Prize compression competition and can be downloaded from <http://prize.hutter1.net>.

<sup>5</sup><http://web.unipmn.it/manzini/danacorporus>

Pattern Length	Ordinary Files						<i>k</i> -bit filtered files		
	q-hash	BLIM	BOM2	BSOM2	BM	QS	1-bit	2-bit	4-bit
5	0.590	0.265	0.329	0.462	0.256	0.217	0.372	<b>0.143</b>	0.148
10	0.524	0.154	0.211	0.291	0.144	0.137	0.093	<b>0.076</b>	0.092
15	0.202	0.122	0.150	0.202	0.109	0.101	0.062	<b>0.059</b>	0.069
20	0.128	0.101	0.126	0.170	0.094	0.093	<b>0.044</b>	0.049	0.056
25	0.094	0.100	0.107	0.142	0.088	0.087	<b>0.042</b>	0.050	0.049
30	0.077	0.092	0.091	0.117	0.079	0.076	<b>0.033</b>	0.036	0.041
35	0.065	0.056	0.080	0.102	0.070	0.070	0.032	<b>0.031</b>	0.039
40	0.057	0.056	0.074	0.095	0.071	0.072	<b>0.028</b>	<b>0.028</b>	0.035
45	0.052	0.056	0.069	0.085	0.066	0.066	<b>0.024</b>	0.033	0.033
50	0.050	0.057	0.064	0.078	0.068	0.064	0.032	<b>0.025</b>	0.031

(a) Average user time on natural language texts of 20MB

5	0.866	0.689	0.947	1.346	0.973	0.940	0.576	<b>0.210</b>	0.360
10	0.753	0.404	0.497	0.712	0.674	0.753	0.139	<b>0.109</b>	0.198
15	0.289	0.331	0.369	0.512	0.558	0.662	<b>0.085</b>	<b>0.085</b>	0.193
20	0.183	0.253	0.286	0.405	0.536	0.647	<b>0.063</b>	0.070	0.144
25	0.136	0.250	0.240	0.335	0.461	0.589	<b>0.059</b>	0.070	0.146
30	0.110	0.241	0.204	0.280	0.449	0.537	<b>0.048</b>	0.052	0.113
35	0.093	0.177	0.187	0.257	0.524	0.809	0.046	<b>0.045</b>	0.142
40	0.082	0.172	0.169	0.224	0.454	0.618	<b>0.038</b>	0.041	0.109
45	0.076	0.167	0.153	0.206	0.441	0.651	<b>0.035</b>	0.049	0.121
50	0.071	0.166	0.143	0.190	0.432	0.690	0.043	<b>0.036</b>	0.107

(b) Average user time on plain ASCII DNA sequences of 30MB

5	0.883	0.213	0.249	0.392	0.289	0.239	0.524	0.209	<b>0.207</b>
10	0.782	0.125	0.140	0.215	0.156	0.138	0.143	<b>0.113</b>	0.134
15	0.301	0.096	0.106	0.155	0.111	0.101	0.088	<b>0.087</b>	0.099
20	0.190	0.077	0.087	0.124	0.090	0.082	<b>0.067</b>	0.071	0.079
25	0.141	0.074	0.079	0.107	0.080	0.074	<b>0.061</b>	0.072	0.068
30	0.113	0.070	0.072	0.095	0.072	0.069	<b>0.050</b>	0.052	0.058
35	0.096	0.064	0.069	0.088	0.069	0.067	0.049	<b>0.046</b>	0.054
40	0.085	0.064	0.066	0.081	0.067	0.067	<b>0.040</b>	0.042	0.049
45	0.078	0.065	0.064	0.078	0.066	0.065	<b>0.036</b>	0.048	0.046
50	0.074	0.065	0.063	0.074	0.065	0.064	0.046	<b>0.037</b>	0.042

(c) Average user time on random byte sequences of 30MB

**FIGURE 5.** Comparison of pattern matching performance between ordinary files and *k*-bit filtered files via average user times measured in milliseconds during the experiments.

the individual bit-streams of the file as explained in section 2. Note that in this process we used the *gzip* tool, which is a standard dictionary-based compression software. Since *gzip* operates on bytes, we stored the bit-sequences as a sequence of characters instead of storing them as bit-arrays.

Sample patterns of length 5 to 50 are randomly selected from the ordinary file. Each pattern is scanned on the input file by the Boyer-Moore [8] (BM), Sunday's quick search[16] (QS), Lecroq's *q*-hash [6] (*q*-hash), backward oracle/suffix oracle matching [17] (BOM2/BSOM2), and Külekci's bit-parallel BLIM [13] algorithms.

Same patterns are scanned on the 1-bit, 2-bit, and 4-bit filtered files of the source files with the proposed

scheme. Each sample pattern is searched on files five times, and for each length 20 random patterns are used. The experiment is repeated several times on several ordinary files of the same length. The user times are measured via the `getrusage` system call.

Tests were conducted on a machine having a 64-bit Intel Xeon processor with 3GB memory, and best efforts were deployed for the implementation of the algorithms. The compiler used was gcc 4.3.1 on Gentoo Linux 2.6.25.9-101.

Figure 5 lists the average of the measurements. Best performances are marked in bold.

Search speed is on average doubled on *k*-bit filtered files for all lengths. It is observed that the gain is more significant on DNA sequences, as on short patterns

up to length 20, matching via  $k$ -bit filtering is more than three times faster when compared with the best performing classical algorithms included in this study.

Note that as the filter bit length  $k$  increases, so does the distinguishing power. On the other side, using more bits enlarges the length of the file filter part ( $k \cdot n$  bits), which in turn slows down the pattern filter matching. In this trade-off, the results indicate that  $k = 4$  is a bad choice, and up to length 15, selection of  $k = 2$  seems better. Just one bit filter ( $k = 1$ ) is in general more speedy than the best resulting algorithm, except the cases of very short patterns on natural language and random texts.

## 5. CONCLUSION

This study focused on exploring an alternative way of storing files other than the indexing structures for faster search and proposed  $k$ -bit filtering for this aim. As a future research point, it is also possible to create an index over the  $k$ -bit filtered files as well, which may improve the efficiency more. Moreover, the compressed pattern matching [18] techniques are also applicable on the proposed format.

The most informative  $k$  bits of each byte for a predefined  $k$  value are moved to the beginning of the file, which generates a  $k \cdot n$  bit sequence for an  $n$ -byte file. The remaining bits, named as the payload of the file during the study, are concatenated keeping the order of appearance.

When an input pattern of length  $m$  is to be scanned, the corresponding  $k$ -bits from the characters of the pattern are extracted and that  $(k \cdot m)$ -bit vector is searched on the initial  $(k \cdot n)$ -bit filter part of the file. In case of a match in the filter at some bit position, the respective payload position on the file is computed and verified with the payload of the pattern.

Experiments conducted on natural language texts, plain ASCII DNA sequences, and random byte texts indicated that exact matching performance is doubled on average when files are stored in  $k$ -bit filtered format, even for  $k = 1$ . Selecting  $k = 2$  causes an improvement on short patterns as expected since the distinguishing power is incremented by more bits, but setting  $k = 4$  worsens the performance. It is also observed that the highest gain is obtained on 1-bit filtered DNA sequences, especially on patterns shorter than 20 bases.

## REFERENCES

- [1] Apostolico, A. and Galil, Z. (eds.) (1997) *Pattern Matching Algorithms*. Oxford University Press.
- [2] Charras, C. and Lecroq, T. (2004) *Handbook of exact string matching algorithms*. King's Collage Publications.
- [3] Crochemore, M. and Rytter, W. (2003) *Jewels of stringology*. World Scientific Publishing.
- [4] Grossi, R. and Vitter, J. (2005) Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, **35**, 378–407.
- [5] Wu, S. and Manber, U. (1992) Agrep – a fast approximate pattern-matching tool. *Proceedings of the Winter 1992 USENIX Technical Conference*, San Francisco, California, USA, pp. 153–162.
- [6] Lecroq, T. (2007) Fast exact string matching algorithms. *Information Processing Letters*, **102**, 229–235.
- [7] Klein, S. T. and Ben-Nissan, M. (2007) Accelerating boyer moore searches on binary texts. *Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA 2007)*, Lecture Notes in Computer Science, **4783**, pp. 130–143. Springer Verlag.
- [8] Boyer, R. and Moore, J. (1977) A fast string searching algorithm. *Communications of the ACM*, **20**, 762–772.
- [9] Kim, J., Kim, E., and Park, K. (2007) Fast matching method for dna sequences. *Proceedings of the First International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE 2007)*, Lecture Notes in Computer Science, **4614**, pp. 271–281. Springer Verlag.
- [10] Faro, S. and Lecroq, T. (2009) Efficient pattern matching on binary strings. *35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2009)*, Spindleruv Mlyn, Czech Republic. Poster.
- [11] Faro, S. and Lecroq, T. (2009) An efficient matching algorithm for encoded dna sequences and binary strings. *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, Lille, France Lecture Notes in Computer Science. Springer Verlag.
- [12] Navarro, G. and Raffinot, M. (2000) Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithms*, **5**, 1–36.
- [13] Külekcı, M. O. (2008) A method to overcome computer word size limitation in bit-parallel pattern matching. *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC 2008)*, Gold Coast, Australia, December, Lecture Notes in Computer Science, **5369**, pp. 496–506. Springer Verlag.
- [14] Shannon, C. E. (1948) A mathematical theory of communication. *Bell System Technical Journal*, **27**.
- [15] Horspool, N. (1980) Practical fast searching in strings. *Software – Practice and Experience*, **10**, 501–506.
- [16] Sunday, D. (1990) A very fast substring search algorithm. *Communications of the ACM*, **33**, 132–142.
- [17] Allauzen, C., Crochemore, M., and Raffinot, M. (1999) Factor oracle: A new structure for pattern matching. *Proceedings of the 26th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 1999)*, Lecture Notes in Computer Science, **1725**, pp. 291–306. Springer Verlag.
- [18] Takeda, M., Shibata, Y., Matsumoto, T., Kida, T., Shinohara, A., Fukamachi, S., Shinora, T., and Arikawa, S. (2001) Speeding up string pattern matching by text compression: The dawn of a new era. *Transactions of Information Processing Society of Japan*, **42**, 370–384.