Accelerating Boyer Moore Searches on Binary Texts

Shmuel T. Klein Miri Kopel Ben-Nissan

Department of Computer Science, Bar Ilan University, Ramat-Gan 52900, Israel
Tel: (972–3) 531 8865 Email: {tomi,kopel}@cs.biu.ac.il

Abstract

The Boyer and Moore (BM) pattern matching algorithm is considered as one of the best, but its performance is reduced on binary data. Yet, searching in binary texts has important applications, such as compressed matching. The paper shows how, by means of some pre-computed tables, one may implement the BM algorithm also for the binary case without referring to bits, and processing only entire blocks such as bytes or words, thereby significantly reducing the number of comparisons. Empirical comparisons show that the new variant performs better than regular binary BM and even than BDM.

Key words: Boyer-Moore, BDM, Pattern matching, binary texts, compressed matching
PACS:

1. Introduction

One of the important applications of automata theory is to Pattern Matching. Indeed, many matching methods can be reformulated in terms finding an automaton with certain properties, as, e.g., the KMP algorithm [11], and several variations of the Boyer and Moore (BM) method [2]. The Backward DAWG Match (BDM) algorithm uses a suffix automaton and runs in optimal sublinear average time [5]. This papers deals with an extension of the BM algorithm to binary data, which has important applications, for example matching in compressed texts. For the ease of description, we shall stick to the usual pattern matching vocabulary.

BM uses two independent heuristics for shifting the pattern forward, denoted by $delta_1$ and $delta_2$ in their original paper. $delta_1$ shifts the pattern according to the character in the text that caused the mismatch: if the character $T[i]$ does not appear at all in P, the pattern can be shifted by its full length. In the binary case, however, the character $T[i]$

is either 1 or 0, and both will most probably appear in P, even close to its right end, except for a very restricted set of patterns. Thus $delta_1$ will rarely let the pattern to be shifted by more than just a few bits.

The second shift function, $delta_2$, assigns a value for each possible position j of the mismatch in P, noting that if $P[j]$ is the first element from the right that does not match, then the suffix $P[j+1]\cdots P[m]$ does, so one may look for a reoccurrence of this suffix in the pattern. More precisely, one looks for the reoccurrence of this suffix which is not preceded by $P[j]$, or, if such an occurrence is not found, one looks for the occurrence of the longest suffix of the suffix $P[j+1]\cdots P[m]$ which is a prefix of P. While $delta_2$ is reported to add only marginally to the performance in the general case, it does in fact all the job on binary data, as $delta_2 \geq delta_1$ in the binary case.

There is, however, an additional problem with using BM on binary input: it forces the programmer to deal at the bit-level, which is more complicated and time consuming. Packing the bits into blocks and processing byte per byte is not a solution, as the location of the pattern in the text is not necessarily byte aligned, in particular when the binary text at hand is the compressed form of some input text. This led to the idea of using 256-ary Huffman codes as compression scheme, especially for very large alphabets for which the loss relative to the optimal binary variant is small [14]. To enable compressed matching, the first bit of each byte serves as *tag*, which is used to identify the last byte of each codeword, thereby reducing the order of the Huffman tree to 128-ary. These *Tagged Huffman codes* have then been replaced by the better (s,c)-*Dense codes* in [3].

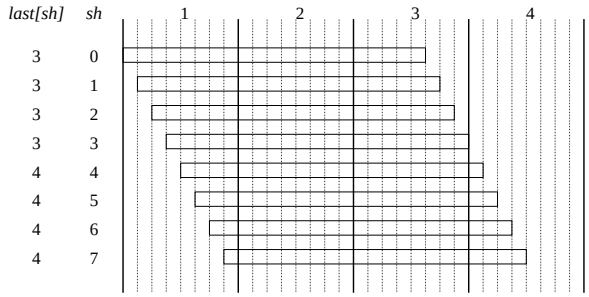
We show in this paper how to apply the BM algorithm even in the binary case by treating only full blocks of k bits at a time, typically, bytes, half-words or words, that is $k = 8$, 16 or 32. This takes advantage of the fact that such k-bit blocks can be processed at the cost of a single operation. The idea is to eliminate any reference to bits and to proceed, by means of some pre-computed tables, block by block, similarly to the fast decoding method of binary Huffman encoded texts, first suggested in [4]. A similar approach has been taken in [7], and the special case of BM compressed matching for LZ encoded texts is treated in [15,16], and for BWT compression in [1]. The details of the algorithm are presented in the next section, an analysis is given in Section 3, and Section 4 brings empirical comparisons.

2. A high-level binary BM variant

Once the block size k is fixed, all references to both text and pattern will only be to entire blocks of k bits. For the ease of description, we shall refer to a k-bit block as a *byte*, though larger values than $k = 8$ could be supported as well.

Let $\mathsf{Text}[i]$ and $\mathsf{Pat}[i]$ denote, respectively, the i-th byte of the text and of the pattern, starting for $i = 1$ with both text and pattern aligned at the leftmost bit of the first byte. When we need to refer to the individual bits rather than the bytes, we shall use the notation $T[i]$ and $P[i]$. Since the lengths in bits of both text and pattern are not necessarily multiples of k, the last bytes may be only partially defined. In fact, we shall use a sequence of several copies of the pattern: using a shift parameter sh, with $0 \leq sh < k$, denote by $\mathsf{Pat}[sh, i]$ the i-th byte of the pattern after an initial shift of sh bits to the

right. Figure 1 visualizes these definitions for $k = 8$ and a pattern of length $m = 21$. The bold vertical bars indicate the byte boundaries. Let $\mathsf{last}[sh]$ be the index of the last byte of the pattern that has been shifted sh bits, so formally $\mathsf{last}[sh] = \lceil (sh + m)/k \rceil$. In our example in Figure 1, $\mathsf{last}[sh] = 3$ for $0 \le sh \le 3$, and $\mathsf{last}[sh] = 4$ for $4 \le sh \le 7$. Figure 1 also includes a Table Correct, to be explained below.



Correct$[sh, j]$	1	2	3
0	13	5	–
1	14	6	–
2	15	7	–
3	16	8	–
4	17	9	1
5	18	10	2
6	19	11	3
7	20	12	4

FIGURE 1: *Schematic representation of shifted copies of the pattern*

The algorithm starts by comparing the last byte of the pattern, of which possibly only a proper prefix belongs really to P, with the corresponding byte of the text. Let sl be the length in bits of the suffix of P in this last byte, that is, $sl = 1 + (m - 1) \bmod k$, so that $1 \le sl \le k$. The variable sl will hold this suffix length throughout the program, but sl may change because of the initial shift sh. When processing the first or last byte of P, the bit positions not belonging to the pattern have to be neutralized by means of pre-computed masks. Define $\mathsf{Mask}[sh, j]$ as a binary mask of length k in which a bit is set to 1 if and only if the corresponding bit of $\mathsf{Pat}[sh, j]$ belongs to P. For our example of Figure 1, $\mathsf{Mask}[0, 3] = 11111000$, $\mathsf{Mask}[0, 2] = 11111111$ and $\mathsf{Mask}[2, 1] = 00111111$.

In the original BM algorithm, the comparisons are between characters, which either match or not. In this binary variant, more information is available in case of a mismatch, namely, the length of the (possibly empty) suffix of the mismatching byte that still did match. For example, the characters `b` and `j` do not match, but their corresponding ASCII encodings, 01100010 and 01101010 are not completely unrelated and share a common suffix of length 3. In fact, they share even more common bits, but we are interested in the longest common suffix, because BM scans from right to left up to the first mismatch, which in this example would be at the 4-th bit from the right.

A way to get the length of the longest matching suffix is to replace comparisons by Xoring. If $C = A$ XOR B, then $C = 0$ if and only if $A = B$, but if $A \ne B$, then the length of the longest common suffix of A and B is the length of the longest run of zeros at the right end of C. In the above example, `b` XOR `j` $= 00001000$ and the requested length is 3. Let $\mathsf{NRZ}[x]$ be this number of rightmost zeros for a given binary string x of length k. To speedup the computation, we shall keep NRZ as a pre-computed table of size 2^k. This is just 256 for $k = 8$, but for $k = 16$ or even 32, such a table might be too large. Not that requesting enough RAM to store a 2^{16} size table is unreasonable, but with growing k, there will be many cache misses, and they may have a strong impact on the performance. Certainly 2^{32} seems too large, so for larger k, it might pay to compute the function on the fly, still using a table $\mathsf{NRZ}[x]$ for single bytes. For example, for $k = 32$, denoting the

four bytes of x by $x_1, \ldots, x_4$ and the rightmost two bytes by x_{34}, the function could be defined by

```
if x34 = 0 then                                  else
    if x2 = 0 then return 24 + NRZ[x1]               if x4 = 0 then return 8 + NRZ[x3]
    else           return 16 + NRZ[x2]               else           return NRZ[x4]
```

It was mentioned in the introduction that the $delta_1$ heuristic of the original BM algorithm will rarely be useful in the binary case, but it is possible to use an extension of the idea as follows. Refer to Figure 2 for an example of a typical scenario. Suppose the pattern has just been positioned at its current location, being shifted sh bits to the right from a byte boundary and extending sl bits into its rightmost byte, which corresponds to byte i of the text. The first comparison at this position will be between $\mathsf{Text}[i]$ and $\mathsf{Pat}[sh, \mathsf{last}[sh]]$, using the appropriate mask to cancel the rightmost bits of the latter. Suppose we got a mismatch. The original BM $delta_1$ heuristic would ask where the rightmost occurrence of the bit in the text causing the mismatch (which, in the binary case, is the complement of the rightmost bit of P) can be found in P. In the byte oriented version on the other hand, since several bits have been compared already, we know what the first sl bits of $\mathsf{Text}[i]$ are, so one can check whether these bits, corresponding to the shaded area in the Text of the upper part of Figure 2, reoccur somewhere in the pattern. This is similar to the idea underlying the original definition of $delta_2$. If yes, we seek the rightmost reoccurrence, which, in our example, is the identically shaded area in P. The pattern can thus be shifted forward so as to align these matching substrings, as can be seen in the bottom line of the example. If the bits do not reoccur, the pattern can be shifted by its full length. The lower part of Figure 2 will be referred to below, when dealing with $delta_2$.

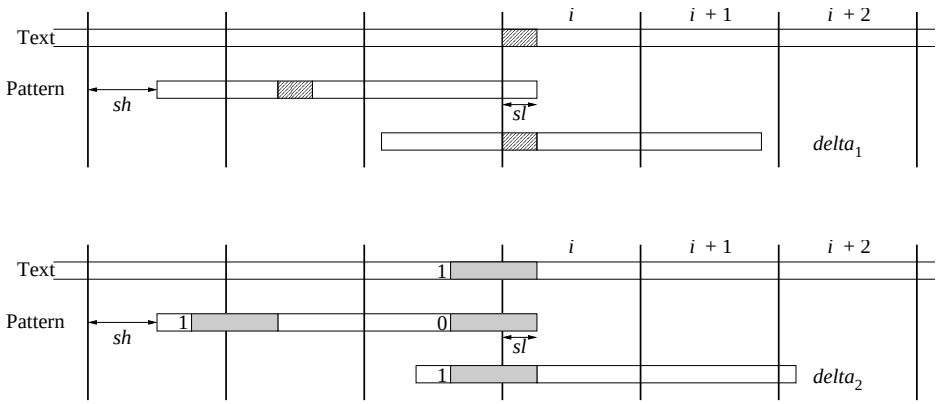


FIGURE 2: *Typical pattern matching scenarios with binary data: $delta_1$ and $delta_2$*

2.1. *Definition of $delta_1$*

The amount of shifts for all the possible cases can be computed in advance. We define a sequence of tables $delta_1[sl, B]$, one table for each possible suffix length sl, with $1 \leq sl \leq k$ and $0 \leq B < 2^{sl}$. $delta_1[sl, B] = m - \ell$, where ℓ is the index of the last bit of the rightmost occurrence in P of the sl-bit binary representation of B. For example, if $sl = 4$ and $P = 0010101011101101$, where the rightmost occurrence of $B = 5 = 0101$ is underlined, then $delta_1[4, 5] = 16 - 9 = 7$. In fact, this definition needs a similar amendment as that of $delta_2$ in the original algorithm: if the sl-bit binary representation of B does not occur as a substring of P, but some proper suffix of it appears as a prefix of P, this should still be counted as a plausible reoccurrence. For example, using $sl = 5$ and the same P but letting $B = 17 = 10001$, one sees that B does not occur in P, but its suffix 001 is a prefix of P. The pattern can thus not be shifted by its full length, and $delta_1[5, 17] = 13$ rather than 16. The tables are first initialized with m in each entry; then the prefixes of P are taken care of by:

```
for r ⟵ 1 to sl − 1
    for all values of B such that the prefix P[1] ··· P[r] is a suffix of B
        delta1[sl, B] ⟵ m − r;
```

finally, the other values are filled in by scanning the pattern left to right:

```
for t ⟵ 1 to m − sl
    B ⟵ P[t] ··· P[t + sl − 1]
    delta1[sl, B] ⟵ m − t − sl + 1,
```

where the assignment of a bit-string of length sl to B should be understood as considering this string as the binary representation of some number, which is assigned to B.

The formal binary BM algorithm is given in Figure 3 (it contains some details to be explained below). Each iteration of the while loop starting in line 3 corresponds to checking if there is a match at the given position i in the text. The pointer to the pattern is initialized in each iteration to the last byte, and the suffix length is calculated. In the main comparison loop in lines 6–8, the variable *indic* serves as indicator if there has been a match. If not, and the pattern has not yet been found, the else clause starting at line 10 calculates the new position of the pointer i to the text. Lines 11–15 and 16–17 deal, respectively, with Δ_1 and Δ_2, which are based on BM's original $delta_1$ and $delta_2$ functions. In line 18, the new bit position of the text pointer is evaluated by adding the maximum possible shift to *lastbit*, which is throughout the bit-index of the last bit in the text that was aligned with the last bit of the pattern. The new byte oriented values of i and sh are then derived accordingly from the newly calculated bit-position in lines 19–20. Note that contrarily to the original BM algorithm, it is possible that the byte index i is not increased after a mismatch (but then the suffix length sl is), in case the shift is not large enough to cause the crossing of a byte boundary.

BLOCKED BOYER-MOORE MATCHING

```
1   i ⟵ last[0]        sh ⟵ 0
2   lastbit ⟵ m
3   while i ≤ ⌈n/k⌉   // length of text in bytes
4       j ⟵ last[sh]
5       sl ⟵ 1 + ((m + sh − 1) mod k)
6       while j > 0 AND (indic ⟵ (Text[i] XOR Pat[sh, j]) AND Mask[sh, j]) = 0
7           i ⟵ i − 1
8           j ⟵ j − 1
9       if j = 0      print "match at k ∗ i + sh"
10      else
11          if j = last[sh]
12              B ⟵ (Text[i] / 2^(k−sl)) mod 2^K
13              Δ1 ⟵ delta1[min(sl, K), B]
14          else
15              Δ1 ⟵ delta1[K, Text[i] mod 2^K] − Correct[sh, j]
16          matchlen ⟵ sl + NRZ[indic] + (last[sh] − j − 1) ∗ k
17          Δ2 ⟵ delta2[m − matchlen]
18          lastbit ⟵ lastbit + max(Δ1, Δ2)
19          i ⟵ ⌈lastbit / k⌉
20          sh ⟵ (lastbit − m) mod k
```

FIGURE 3: *Formal BM algorithm for binary data*

If in the first comparison in line 6, between $\mathsf{Text}[i]$ and $\mathsf{Pat}[sh, \mathsf{last}[sh]]$, one gets a match, the preceding bytes of both text and pattern are inspected. This continues either until the pattern is found (line 9), or until a mismatch stops this iteration. In the original BM algorithm, a single $delta_1$ table could be used, regardless of whether the mismatch occurred in the last byte of the pattern or not, because $delta_1$ was in fact the amount by which the current pointer to the text i could be increased, rather than the number of characters the pattern could be shifted. In this binary variant, on the other side, $delta_1$ will only hold the number of bits to shift the pattern, because the increase in i, if there is one at all, has to be calculated. Therefore, if the mismatch is not at the last byte of P, $delta_1$ has to be corrected by subtracting the number of bits i has been moved to the left during the comparison at the current position of the pattern. This correction is constant for a given pair (sh, j), so a table $\mathsf{Correct}[sh, j]$ can be prepared in advance. The table corresponding to the example in Figure 1 is shown at the right side of the figure. It is not defined for $j = \mathsf{last}[sh]$, and for the other values it is given by

$$\mathsf{Correct}[sh, j] = sl + (\mathsf{last}[sh] - 1 - j) \times k.$$

The combined size of all the $delta_1$ tables is $\sum_{sl=1}^{k} 2^{sl} = 2^{k+1} - 2$, which seems reasonable for $k = 8$ or even 16, but not for $k = 32$. One can adapt the algorithm to choose the desired time/space tradeoff by introducing a new parameter $K \leq k$, representing

the maximal number of mismatching bits taken into account for deciding by how much to move the pattern. If the mismatch is in the first iteration, that is, between $\mathsf{Text}[i]$ and $\mathsf{Pat}[sh, \mathsf{last}[sh]]$, and if $sl > K$, then only the K rightmost bits of the sl first bits of $\mathsf{Text}[i]$ are taken into account (implemented by the division and the mod function in line 12), for the other iterations, the reference is to the K rightmost bits of the current byte of the text (line 15). This reduces the total sizes of the tables to 2^{K+1} at the cost of sometimes shifting the pattern less than could be done if the full length mismatch had been considered. The possible values of K are not bound to be multiples of 8 and the choice of K is governed solely by the available space.

2.2. *Definition of $delta_2$*

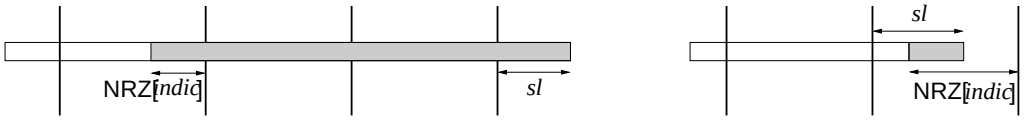
The $delta_2$ table is a function of the length of the matching suffix. This length, denoted by *matchlen*, consists, in case the mismatch did not occur at the last byte of the pattern, of three parts:

(i) the length of the suffix of the pattern in the last byte;
(ii) the lengths of the full matching bytes;
(iii) the length of the matching suffix of the byte that finally caused the mismatch.

The first item is sl, the second is a multiple of k and the last is given by $\mathsf{NRZ}[indic]$. This general case is represented in the left part of Figure 4, where the matching part of the pattern appears in grey. For the special case in which the mismatch occurred already at the first comparison, depicted in the right part of Figure 4, the length of the match is the length of the matching suffix of the sl-bit prefix of $\mathsf{Pat}[sh, \mathsf{last}[sh]]$. This is $\mathsf{NRZ}[indic]$, from which the length of the suffix of the byte not belonging to P, $k - sl$, should be subtracted. One can thus describe the length in both cases by the formula

$$\mathsf{NRZ}[indic] + sl + k \times r,$$

where $r = -1$ if the mismatch is at the last byte, $r = 0$ if it is at the second byte from the right, $r = 1$ for the third, etc. In general, one gets that $r = \mathsf{last}[sh] - j - 1$, where j is the index of the current byte of the pattern at which the mismatch occurred.



Mismatch at the 4th byte *Mismatch at the last byte*

FIGURE 4: *Examples of mismatch lengths for the definition of $delta_2$*

To be consistent with the original BM algorithm, the parameter of $delta_2$ is not the length of the matching suffix, but the index of the first bit from the right that did not match, which is $m - matchlen$. Figure 5 brings two examples of $delta_2$ tables. The upper one corresponds to the example pattern above $P = 0010101011101101$. The values are filled following the original algorithm, except that as above for $delta_1$, the values stored will be the number of bits the pattern can be shifted to the right, and not the number of

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Pat[j]	0	0	1	0	1	0	1	0	1	1	1	0	1	1	0	1
$delta_2[j]$	16	16	16	16	16	16	16	16	16	16	16	3	7	13	2	1

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Pat[j]	1	0	1	0	1	0	1	0	1	1	1	0	1	1	0	1
$delta_2[j]$	13	13	13	13	13	13	13	13	13	13	13	3	7	15	2	1

FIGURE 5: *Examples of $delta_2$*

bits the text pointer can be moved. For example, position 12 corresponds to a matching suffix 1101 and a mismatch at the 0-bit preceding this suffix. We are thus looking for a reoccurrence of this suffix preceded by the complement of the bit that caused the mismatch.

The situation is schematically represented in the lower part of Figure 2, where the greyed suffix of the pattern, preceded by a zero, appears again earlier in the pattern, but preceded there by a 1. The next line in the figure shows how the pattern can be moved so as to align the matching substrings. In the present example, we seek for 11101 in P, which can be found in positions 9–13; the possible shift, which should align the rightmost reoccurrence with the current position of the suffix, in therefore in this case $16 - 13 = 3$. The lower part of Figure 5 shows how the $delta_2$ table changes with a slight change in the pattern: the first bit of P has been changed to a 1. Consider, e.g., position 11, corresponding to the suffix 01101; there is no occurrence of 001101 in P, so a priori, the possible shift should be of length 16, but a suffix of the suffix, 101, appears as prefix in P, so the possible shift is only of length 13.

Note that all the operations in the algorithm in Figure 3 refer to entire bytes of either the text or the pattern and there is no processing of individual bits. Moreover, multiplications and divisions, which generally are more time consuming, are all with powers of 2 and can thus be translated to the more economical shifts by the compiler.

Summarizing, the byte oriented search algorithm makes use of the following tables, all of which are prepared in advance as functions of the pattern only: Mask and Correct, both with k lines and $\lceil m/k \rceil$ columns, so using about m bytes each, $delta_2$ with m entries, last with k entries, NRZ with 2^8 entries, even if $k > 8$, and $delta_1$ of size 2^{K+1}, where K is a parameter of our choice, enabling a time/space tradeoff: increasing K by 1 doubles the size of the $delta_1$ tables, but also increases the expected size of the shift after a mismatch, reducing the expected processing time. Summing it up, we need $3m + k + 256 + 2^{K+1}$ entries, and each can be stored in a single byte (assuming the length m of the pattern is less than 256 bits; for longer patterns, more than one byte is needed for each entry). For example for $m = 100$, using $k = 8$, all the tables together need just slightly more than 1K, and even if we use $k = 32$, thereby quadrupling the number of bits processed in a single operation, the overhead is below 9K if we choose $K = 12$.

3. Time analysis

To compare the number of comparisons of the k-bit block algorithm suggested here with the regular binary BM algorithm, we shall assume the following probabilistic model. The distribution of zeros and ones in the input string is like in a randomly generated one, that is, the probability of occurrence in the text of any binary string of length ℓ is $2^{-\ell}$, and it is independent of the other substrings. This is a reasonable assumption if the binary BM is to be applied on compressed text. For the special case where compression is done using Huffman coding, such "randomness" has been shown to hold in [9], subject to some additional constraints, but in fact any reasonable compression scheme produces output that is quite close to random: if it would not, the remaining redundancy could be removed by applying another compression round on the already compressed text. We moreover assume that this randomness also holds for the input pattern to be searched for.

Let us first concentrate on the binary, unblocked, case, and suppose that the index into the text has just been moved forward to a new location i. The algorithm will now compare $T[i-j]$ with $P[m-j]$ for $j = 0, 1, \ldots$ until a mismatch will allow us to move i forward again. The expected number of comparisons up to and including the mismatch will be

$$1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + \cdots = \sum_{j=1}^{m} \frac{j}{2^j} = 2 - 2^{-(m-1)} - m2^{-m} \simeq 2.$$

This follows from the fact that if p denotes the probability of a zero, then the probability of a bit of the text matching a bit of the pattern is $p^2 + (1-p)^2$, and the probability of a mismatch is $2p(1-p)$; in our case, both probabilities are $\frac{1}{2}$, so the probability of having the first mismatch at the jth trial is $\frac{1}{2}\left(\frac{1}{2}\right)^{j-1} = 2^{-j}$.

For the blocked case, denote by NK the random variable giving the number of comparisons between consecutive mismatches. We then have

$$E(NK) = \sum_{i=1}^{\lceil m/k \rceil} P(NK \geq i) = 1 + \sum_{i=1}^{\lceil m/k \rceil} P(\text{match at first } i-1 \text{ comparisons}).$$

The latter probability depends on the length sl of the suffix in the rightmost k-bit block and is $2^{-sl} + 2^{-(k+sl)} + 2^{-(2k+sl)} + \cdots$. Averaging over the possible values of sl, which all appear with the same probability, we get

$$E(NK) = 1 + \frac{1}{k} \sum_{sl=1}^{k} \sum_{t=1}^{\lceil m/k \rceil} 2^{-(tk+sl)} < 1 + \frac{1}{k} \sum_{sl=1}^{k} 2^{-sl} \sum_{t=1}^{\infty} \left(2^{-k}\right)^t = 1 + \frac{\left[1 - 2^{-k}\right]}{k(1-2^{-k})} = 1 + \frac{1}{k}.$$

Let $M$and M' denote, respectively for the unblocked and k-bit blocked variants, the expected number of bits the pattern can be shifted after a mismatch is detected. In fact, if both algorithms would use the same $delta_1$ and $delta_2$ functions, one would have $M = M'$, as they depend only on the length of the longest matching suffix of the pattern at each position, and not on whether this suffix has been detected bit by bit or in blocks. Indeed, both algorithms use the same $delta_2$, but the block variant exploits the fact that several bits have been compared in a single operation to increase, in certain cases, the

jump defined by $delta_1$. Our empirical tests show that about 75% of the jumps are due to the revised $delta_1$ in the blocked variant. It therefore follows that $M' \geq M$. If n is the length of the text in bits, the expected number of comparisons for the unblocked case is thus $\frac{2n}{M}$, which is larger than the corresponding number for the blocked case, $\frac{(1+\frac{1}{k})n}{M'}$.

To evaluate M, assume that the first mismatch occurs at the jth trial, $j \geq 1$. The value of $delta_2$ for the rightmost bit of the pattern is usually 1, unless the pattern has ℓ identical characters as suffix, in which case $delta_2$ of the last ℓ positions is ℓ. So if $j = 1$, the expected shift by $delta_2$ is $\sum \frac{\ell}{2^{-\ell}} \simeq 2$. For $j > 1$, the probability of a suffix of length $j-1$ to reoccur with complemented preceding bit is 2^{-j}, thus the expected position of this reoccurrence is 2^j bits to the left, but since the shift is limited, the expected shift caused by $delta_2$ will be $\min(2^j, m)$. Since the probability of having the first mismatch at the jth trial is 2^{-j}, we get as expected size of the shift

$$\frac{1}{2}\,2 + \sum_{j=2}^{m} 2^{-j} \min(2^j, m) = 1 + (\log m - 1) + m \sum_{j=\log m+1}^{m} 2^{-j} = \log m + m2^{-m} - 1 \simeq \log m,$$

for large enough m, where the logarithm is with base 2.

4. Experiments

To get some empirical results, we ran the following tests on the English Bible (2.55 MB, King James Version) and the World Factbook 1992 file of the Gutenberg Project (1.49 MB). Both files were Huffman encoded according to their character distributions. The patterns were chosen as substrings of lengths 10 to 500, starting at 200 randomly chosen positions in encoded string, and for each length, the 200 obtained values were averaged. The results are displayed in the plots of Figures 6 to 9, giving the averaged values as function of the length of the pattern m.

Figure 6 gives the average number of comparisons between consecutive shifts. As expected, this number is smaller for the blocked variant than for bit by bit processing, though also for the latter, the obtained values were smaller than 2. The plot in Figure 7 shows the values M and M'. M exhibits indeed logarithmic behavior, and the values of M' can be seen to grow at a faster rate. This may be due to the extended definition of $delta_1$ for the blocked algorithm, which has a dominant influence on the shift size.

The next plots compare the behavior of the blocked and bitwise binary BM variants also with BDM and Turbo-BDM [6,5], two suffix automaton pattern matching techniques, which are among the best alternatives to BM. Figure 8 plots the expected number of comparisons for every 1000 bits passed of the text, showing a significant reduction with the blocked (highlighted as the bolder line) versus the bitwise variant, and improving even on both BDM and Turbo-BDM, which are indistinguishable on these plots. The left part of Figure 8 corresponds to the Bible and the right part to the World Factbook. Typical values, for patterns of length 200 for the Bible file, were 104.0 comparisons for bitwise BM, 43.6 for BDM, 43.5 for TurboBDM and 16.1 for blocked BM. All graphs show decreasing functions of the pattern length, as expected.

The last graphs, in Figure 9, are timing results measured on an Intel Dual Core CPU T7200, 2.00 GHz, with 2GB of RAM. To neutralize the influence of the location of the

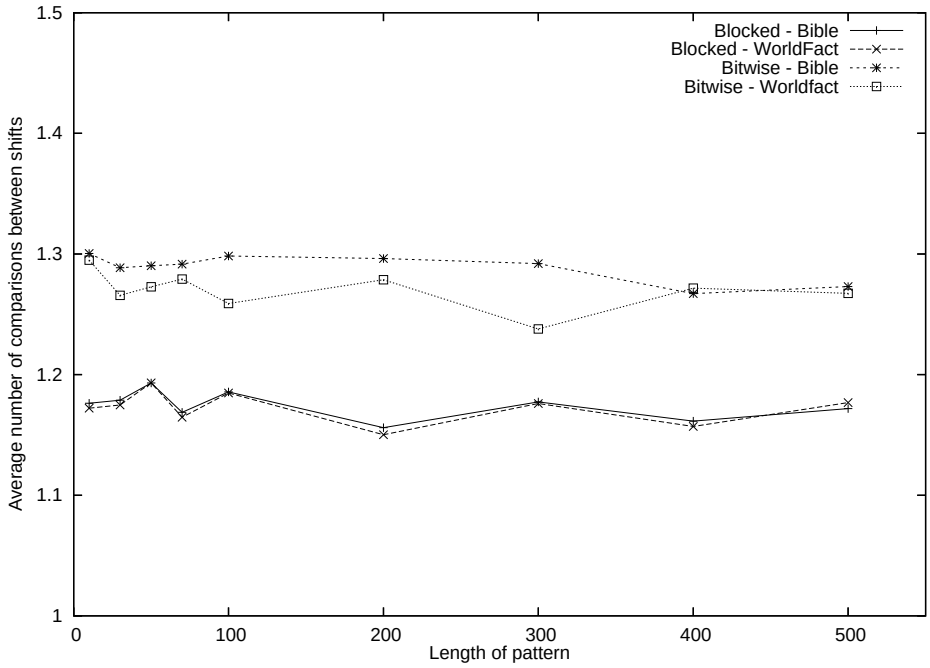


FIGURE 6: *Average number of comparisons between shifts*

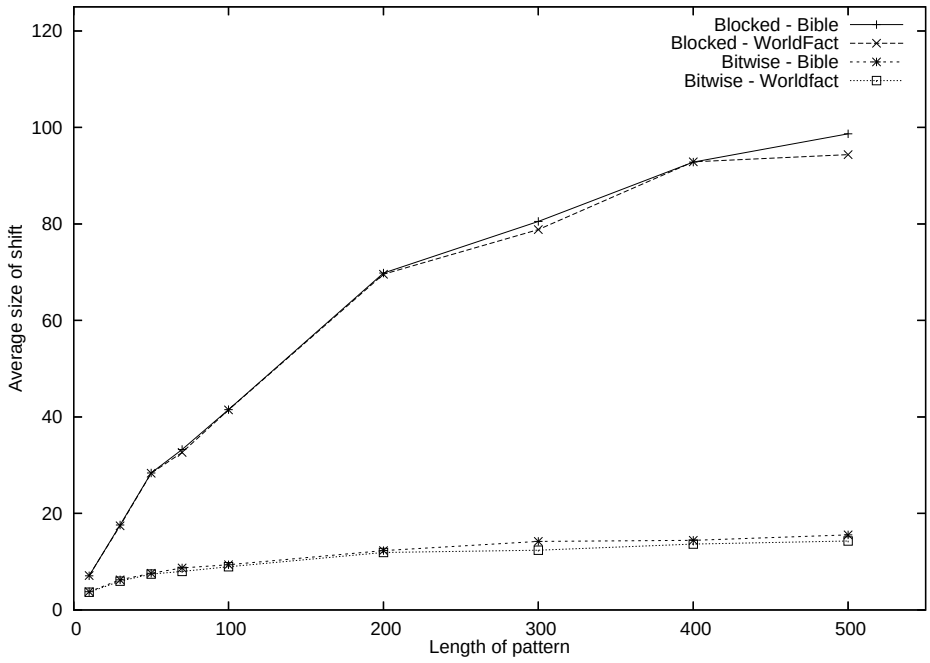


FIGURE 7: *Expected size of shift after mismatch*

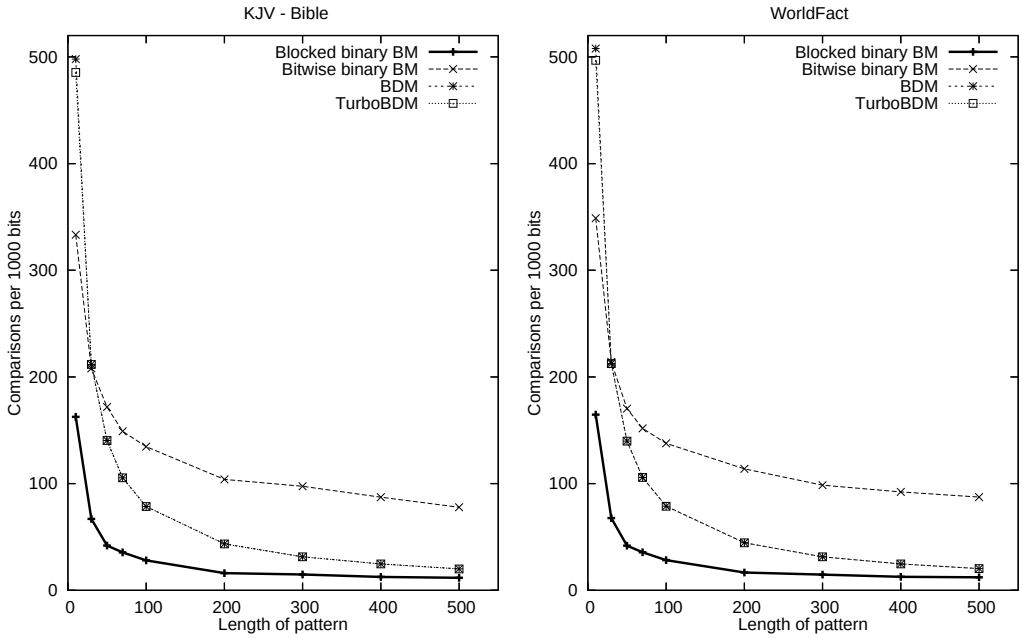


FIGURE 8: *Expected number of comparisons for 1000 bits*

pattern on the search time, all the occurrences have been searched for, that is, the time is that of a full scan of the text. As above, the figure has two parts, the left one for the Bible and the right one for the World Factbook and the lines for the blocked BM variant are emphasized. Values are in seconds. Here too the blocked BM gives consistently better times than the bitwise variant and also than the two BDM algorithms, with typical values, again for patterns of length 200 for the Bible file: 0.138 seconds for bitwise BM, 0.040 for TurboBDM, 0.027 for BDM and 0.008 for blocked BM. As expected. the functions are decreasing, and exhibit similar shapes as the graphs for comparisons of Figure 8. The values for the World Factbook are lower than the corresponding ones for the Bible because the latter is a larger file.

5. Conclusion

A variant of the Boyer Moore pattern matching algorithm has been presented, which is suitable in case when both text and pattern are over a binary alphabet, and nevertheless do not use any bit manipulations. Using a set of tables that are prepared in advance and are independent of the text, the algorithm addresses only entire bytes or words. The expected reduction in time and number of comparisons comes at the cost of a space overhead of 1–10K, which can generally be tolerated.

In fact, one could think that the same ideas could also be used to get a similar reduction in the regular ASCII case: instead of comparing the characters one by one, they could be processed by groups of four (i.e., words instead of bytes). But in practice on natural language text, most of the mismatches occur at the first comparison of each new position

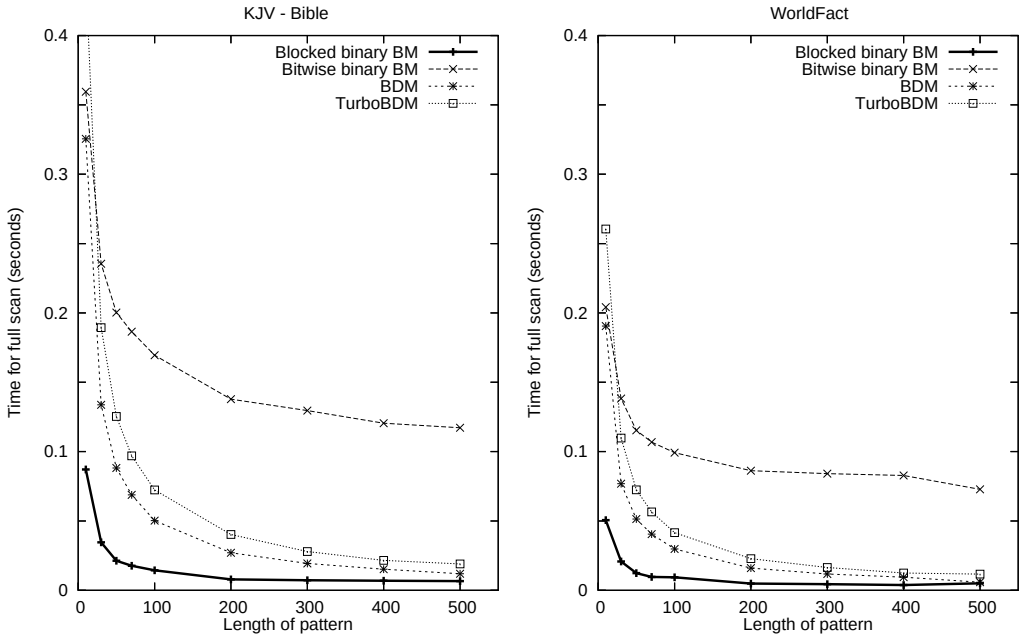


FIGURE 9: *Time to locate all the occurrences of the pattern*

(which has led to the Horspool variant [8]), so the overhead of the word-decoding procedure might in such cases exceed the expected gain. However, for other kinds of ASCII encoded texts, for example those dealt with in biological applications, the extension of BM to work with more than a single character at the same time might well be profitable.

References

[1] BELL T., POWELL M., MUKHERJEE A., ADJEROH D., Searching BWT Compressed Text with the Boyer-Moore Algorithm and Binary Search, *Proc. Data Compression Conf. DCC–02*, Snowbird, Utah (2002) 112–121.

[2] BOYER R.S., MOORE J.S., A fast string searching algorithm, *Comm. ACM* **20** (1977) 762–772.

[3] BRISABOA N.R., FARIÑA A., NAVARRO G., ESTELLER M.F., (S,C)-dense coding: an optimized compression code for natural language text databases, *Proc. Symposium on String Processing and Information Retrieval SPIRE'03*, *LNCS* **2857**, Springer Verlag (2003) 122–136.

[4] CHOUEKA Y., KLEIN S.T., PERL Y., Efficient Variants of Huffman Codes in High Level Languages, *Proc. 8-th ACM-SIGIR Conf.,* Montreal (1985) 122–130.

[5] CROCHEMORE M., CZUMAJ A., GASIENIEC L., JAROMINEK S., LECROQ T., PLANDOWSKI W., RYTTER W., Speeding Up Two String-Matching Algorithms, *Algorithmica* **12**(4/5) (1994) 247–267.

[6] CROCHEMORE M., RYTTER W., *Text algorithms*, Oxford University Press (1994).

[7] FREDRIKSSON K., Faster String Matching with Super-Alphabets, *Proc. Symposium on String Processing and Information Retrieval SPIRE'02*, *LNCS* **2476**, Springer

Verlag (2002) 44–57.
[8] Horspool R.N., Practical fast searching in strings, *Software — Practice and Experience* **10** (1980) 501–506.
[9] Klein S.T., Bookstein A., Deerwester S., Storing Text Retrieval Systems on cd-rom: Compression and Encryption Considerations, *ACM Trans. on Information Systems* **7** (1989) 230–245.
[10] Klein S.T., Shapira D., Pattern matching in Huffman encoded texts, *Information Processing & Management* **41**(4) (2005) 829–841.
[11] Knuth D.E., Morris J.H., Pratt V.R., Fast Pattern Matching in Strings, *SIAM J. Comp.* **6** (1977) 323–350.
[12] Moffat A., Word-based text compression *Software — Practice and Experience* **19** (1989) 185–198.
[13] Moffat A., Zobel J., Sharman N., Text compression for dynamic document databases, *IEEE Transactions on Knowledge and Data Engineering* **9** (1997) 302–313.
[14] de Moura E.S., Navarro G., Ziviani N., Baeza-Yates R., Fast and flexible word searching on compressed text, *ACM Trans. on Information Systems* **18** (2000) 113–139.
[15] Navarro G., Tarhio J.,, Boyer-Moore String Matching over Ziv-Lempel Compressed Text, *Proc. 11th Annual Symposium on Combinatorial Pattern Matching CPM–00*, *LNCS* **1848**, Springer Verlag (2000) 166–180.
[16] Shibata Y., Matsumoto T., Takeda M., Shinohara A., Arikawa S., A Boyer-Moore Type Algorithm for Compressed Pattern Matching, *Proc. 11th Annual Symposium on Combinatorial Pattern Matching CPM–00*, *LNCS* **1848**, Springer Verlag (2000) 181–194.