Tuning BNDM with q-Grams

Branislav Ďurian[*] Jan Holub[†] Hannu Peltola[‡] Jorma Tarhio[‡]

Abstract

We develop bit-parallel algorithms for exact string matching. Our algorithms are variations of the BNDM and Shift-Or algorithms. At each alignment the algorithms read a q-gram before testing the state variable. In addition we apply reading a 2-gram in one instruction. Our experiments show that many of the new variations are substantially faster than any previous string matching algorithm on x86 processors for English and DNA data.

1 Introduction

Searching for occurrences of a string pattern in a text is a common task. It is utilized not only in text processing but also in other fields of science where patterns need to be found (e.g. DNA processing, musicology, computer vision). Although the task of exact string matching has been extensively studied since seventies, new algorithms or modifications of the previous ones still appear that slightly improve time needed for searching.

The Boyer–Moore algorithm [2] with its many variations is a widely known solution for exact string matching. Horspool's algorithm (BMH) [10] and Sunday's QS algorithm [20] have been considered examples of efficient variations of the Boyer–Moore algorithm. But because modern processors give favor to straight-forward and bit-parallel algorithms, the advantage of BMH and QS is not any more clear.

An elegant way of reaching the asymptotic optimum average time complexity is the Backward DAWG Matching algorithm (BDM) [3]. However, the algorithm is complicated to implement and it is not fast for many typical text searching tasks. Its asymptotic optimality is exposed only when searching for very long patterns.

More suitable is BNDM (Backward Nondeterministic DAWG Matching) by Navarro and Raffinot [17]. BNDM is a kind of cross of the BDM and Shift-Or [1, 4] algorithms. The idea is similar as in BDM, while instead of building a deterministic automaton, a nondeterministic automaton is simulated even without constructing it. The resulting code applies bit-parallelism and it is efficient and compact.

In this paper we present new variations of the BNDM and Shift-Or algorithms. Our point of view is the practical efficiency of algorithms. These algorithms are an outcome of a long series of experimentation on bit-parallelism. At each alignment our algorithms read and process a q-gram, i.e. a string of q characters, before testing the state variable, which is a bit vector holding partial matches recognized so far. In addition we apply reading a 2-gram in one instruction. We concentrate on tuning the algorithms for x86 processors, and the results may be different on other platforms. Our experiments show that the new algorithms are very efficient on newish x86 and x86_64 processors. For example, the search time of the fastest version is less than 36% of the search time of QS for English patterns of five characters. In addition, the best versions are faster than Shift-Or on short DNA patterns. In particular, our algorithms beat clearly the winner of the recent state-of-the-art comparison [15].

We use the following notations. Let a pattern $P = p_1 p_2 \ldots p_m$ and a text $T = t_1 t_2 \ldots t_n$ be two strings over a finite alphabet Σ. The task of exact string matching is to find all occurrences of P in T. Formally we search for all positions i such that $t_i t_{i+1} \ldots t_{i+m-1} = p_1 p_2 \ldots p_m$. In the algorithms we use C-like notations: '$|$', '&', '$\mathbf{neg}()$', '$\ll$', and '$\gg$' represent bitwise operations OR, AND, one's complement, left shift, and right shift respectively. The register width (or word size informally speaking) of a processor is denoted by w. If not otherwise stated we assume that the rightmost bit of the computer word represents the value $2^0 = 1$.

The rest of the paper is organized as follows. Since our work is based on BNDM, we start with the BNDM algorithm in Section 2. Two variations BNDMq and SBNDMq are introduced in Sections 3 and 4, respectively. In Section 5 we present UFNDMq, which is a q-gram variation of the Shift-Or algorithm. Reading a 2-gram in one instruction is dealt with in Section 6. Section 7 reviews the complexity issues and the results of our experiments before concluding

[*] S&T Varias s.r.o., Priemyselná 2, SK-010 01 Žilina, Slovakia

[†] Department of Computer Science and Engineering, Czech Technical University in Prague, Karlovo nám. 13, CZ-121 35, Prague 2, Czech Republic

[‡] Department of Computer Science and Engineering, Helsinki University of Technology, P.O.B. 5400, FI-02015 HUT, Finland.

remarks in Section 8.

2 BNDM

Let us start with BNDM. Its pseudocode [17] is shown as Alg. 2.1. The precomputed table B associates each character a with a bit mask expressing its locations in the pattern. At each alignment of the pattern, the algorithm reads the text from right to left until the whole pattern is recognized or the processed text string is not any substring of the pattern. Between alignments, the algorithm shifts the pattern forward to the start position of the longest found prefix of the pattern (assigned to *last*), or if no prefix is found, over the current alignment ($last = m$). With the bit-parallel shift-and technique the algorithm maintains a state vector D, which has one in each position where a substring of the pattern starts such that the substring is a suffix of the processed text string. The basic version of BNDM works for patterns which are not longer than w.

Algorithm 2.1 (BNDM)

```
for $a \in \Sigma$ do $B[a] \leftarrow 0$ endfor
for $j \leftarrow 1..m$ do
    $B[p_j] \leftarrow B[p_j] \mid (1 \ll (m-j))$ endfor
$i \leftarrow 0$
while $i \leq n-m$ do
    $j \leftarrow m$; $last \leftarrow m$; $D \leftarrow (1 \ll m) - 1$
    while $D \neq 0$ do
        $D \leftarrow D \mathbin{\&} B[t_{i+j}]$
        $j \leftarrow j - 1$
        if $D \mathbin{\&} (1 \ll (m-1)) \neq 0$ then
            if $j > 0$ then $last \leftarrow j$
            else report occurrence at $i+1$ endif
        endif
        $D \leftarrow D \ll 1$
    endwhile
    $i \leftarrow i + last$
endwhile
```

3 BNDMq

We develop BNDM further. We present a version called BNDMq which at each alignment first reads a q-gram, i.e., q characters, before testing the state vector D. Another difference is a more simple instruction flow when the q-gram is not present in the pattern. This loop has been made as short as possible in order to quickly advance $m - q + 1$ positions in such a case. The pseudocode of BNDMq is shown as Alg. 3.1, where $F(i, q)$ is a shorthand notation for instructions

$$B[t_i] \mathbin{\&} (B[t_{i+1}] \ll 1) \mathbin{\&} \cdots \mathbin{\&} (B[t_{i+q-1}] \ll (q-1)).$$

Note that BNDMq does not have the *last* variable storing the found prefix, but the variable i, which points to the counter position of p_{m-q+1}, is updated directly.

Algorithm 3.1 (BNDMq)

```
for $a \in \Sigma$ do $B[a] \leftarrow 0$ endfor
for $j \leftarrow 1..m$ do
    $B[p_j] \leftarrow B[p_j] \mid (1 \ll (m-j))$ endfor
$i \leftarrow m - q + 1$
while $i \leq n - q + 1$ do
    $D \leftarrow F(i, q)$
    if $D \neq 0$ then
        $j \leftarrow i$
        $first \leftarrow i - (m - q + 1)$
        do
            $j \leftarrow j - 1$
            if $D \geq (1 \ll (m-1))$ then
                if $j > first$ then $i \leftarrow j$
                else report occurrence at $j+1$ endif
            endif
            $D \leftarrow (D \ll 1) \mathbin{\&} B[t_j]$
        while $D \neq 0$
    endif
    $i \leftarrow i + m - q + 1$
endwhile
```

At the implementation level, the test starting the outer while loop can be removed by placing a copy of the pattern as a stopper in the end of the text [11]. Then the end of the text is tested every time an occurrence of the pattern is encountered.

4 SBNDMq

The inner while loop of BNDM checks one alignment of the pattern in the right-to-left order. In the same time the loop recognizes prefixes of the pattern. The leftmost one of the found prefixes determines the next alignment of the algorithm. Peltola and Tarhio [19] presented SBNDM, a simplified version of BNDM. SBNDM does not care of prefixes, but shifts the pattern simply past a mismatch. SBNDM is slightly faster than BNDM especially for short patterns. Independently, Navarro [16] has utilized a similar approach already earlier in the code of his NR-grep.

Next we present SBNDMq, which is a revised version of SBNDM applying q-grams. The pseudocode, which has been developed from BNDMq, is shown as Alg. 4.1.

The inner loops of BNDM and BNDMq contain two tests per a text character. The inner loop of SBNDMq has only one test. This feature was also present in the code of Navarro's NR-grep [16]. When removing the test of j (see Alg. 3.1) the loop runs in the case of a

Algorithm 4.1 (SBNDMq)

```
for $a \in \Sigma$ do $B[a] \leftarrow 0$ endfor
for $j \leftarrow 1..m$ do
    $B[p_j] \leftarrow B[p_j] \mid (1 \ll (m-j))$ endfor
Compute $s_0$ with Alg. 4.2
$i \leftarrow m-q+1$
while $i \le n-q+1$ do
    $D \leftarrow F(i,q)$
    if $D \neq 0$ then
        $j \leftarrow i-(m-q+1)$
        do $i \leftarrow i-1$
            $D \leftarrow (D \ll 1)$ & $B[t_i]$
        while $D \neq 0$
        if $j = i$ then
            report occurrence at $j+1$
            $i \leftarrow i+s_0$
        endif
    endif
    $i \leftarrow i+m-q+1$
endwhile
```

match one position further to the left than in BNDMq. The loop does not go any further, because the $w-m$ leftmost bits of each $B[a]$ are zeros, where w is the word length, and the m rightmost bits of D are zeros because of shifting left for m times. Note that if there is an occurrence of the pattern in the beginning of the text, the algorithm reads the character t_0, which should be accessible or the beginning of the text should be processed otherwise. (Also BNDMq reads t_0 in such a situation. But in the case of BNDMq it can be easily avoided at the implementation level.)

In the case of a match, the shift is s_0, which corresponds to the distance to the leftmost prefix of the pattern in itself. For example, s_0 is three for $P =$ `abcab`. If the proportional number of matches is not high, the algorithm runs equally fast with the conservative value $s_0 = 1$. The computation of s_0 is shown as Alg. 4.2.

Algorithm 4.2 (Computing s_0)

```
$S \leftarrow B[p_m]$; $s_0 \leftarrow m$
for $i \leftarrow m-1$ downto $1$ do
    if $S$ & $(1 \ll (m-1)) \neq 0$ then $s_0 \leftarrow i$ endif
    $S \leftarrow (S \ll 1)$ & $B[p_i]$
endfor
```

As an example we give a compact C implementation of the main loop of BNDM2 in Algorithm 4.3. Because of clearness and compactness, this code differs slightly from Alg. 4.1. The initial value of i is m. It is assumed that $t_{n+1}...t_{n+m}$ is a stopper, i.e. a copy of the pattern. Here $s_0 = 1$ is applied. The code computes the number of matches (`nmatches`).

Algorithm 4.3 (BNDM2.c)

```
while (1) {
  while (!(D = (B[t[i]]<<1)&B[t[i-1]]))
    i += m-1;
  j = i;
  while (D = (D<<1)&B[t[i-2]]) i--;
  i += m-1;
  if (i == j) {
    if (i > n) return (nmatches);
    nmatches++;
    i++;
  }
}
```

5 UFNDMq

Algorithms of BNDM and SBNDM type apply backward matching. The TNDM algorithm [19] (a BNDM variant) uses backward and forward scanning. It makes slightly less accesses to the text than BNDM, but it is slower than BNDM. Here we present a new variation called FNDM (Forward Nondeterministic DAWG Matching) as Alg. 5.1. A preliminary version of FNDM was introduced by Holub and Ďurian [9]. The idea is to read every m:th character x of the text while x does not occur in the pattern. If x is present in the pattern, the corresponding alignments are checked by the naive algorithm. BNDM and its descendants apply the shift-and approach while FNDM uses shift-or.

Algorithm 5.1 (FNDM)

```
for $a \in \Sigma$ do $B[a] \leftarrow$ neg$(0)$ endfor
for $j \leftarrow 1..m$ do
    $B[p_j] \leftarrow B[p_j]$ & neg$(1 \ll (j-1))$ endfor
$i \leftarrow m$
while $i \le n$ do
    $D \leftarrow B[t_i]$
    while $D \neq$ neg$(0)$ do
        if $D <$ (neg$(0) \ll (m-1))$ then
            if $p_1 p_2 \ldots p_{m-1} = t_{i-m+1} t_{i-m+2} \ldots t_{i-1}$
            then report occurrence at $i-m+1$
            endif
        endif
        $i \leftarrow i+1$
        $D \leftarrow (D \ll 1) \mid B[t_i]$
    endwhile
    $i \leftarrow i+m$
endwhile
```

Next we extend FNDM to handle q-grams. Let

$G(i, q)$ be a shorthand notation for instructions

$$B[t_i] \mid (B[t_{i-1}] \ll 1) \mid \cdots \mid (B[t_{i-q+1}] \ll (q-1)).$$

If we replace the first occurrence of $B[t_i]$ in Alg. 5.1 by $G(i, q)$, we get FNDMq.

We will develop FNDMq further. The resulting algorithm is UFNDMq which is given as Alg. 5.2. The letter U stands for upper bits because the algorithm utilizes those in the state vector D. Like FNDM, UFNDMq is a filtration algorithm. A candidate is checked by the naive algorithm only if at least q characters are correct. The reading step is q instead of m or 1 after a candidate has been processed. Checking can be done in any order.

Algorithm 5.2 (UFNDMq)

```
$mask \leftarrow (1 \ll (q-1) - 1)$
for $a \in \Sigma$ do
    $B[a] \leftarrow$ neg$(mask \ll m)$ endfor
for $j \leftarrow 1..m$ do
    $B[p_j] \leftarrow B[p_j]$ & neg$(1 \ll (j-1))$ endfor
$t_{n+1}t_{n+2} \ldots t_{n+m} \leftarrow P$
$i \leftarrow 0$ ; $D \leftarrow$ neg$(0)$
while (1) do
    while $(D \mid mask) =$ neg$(0)$ do
        $i \leftarrow i + m$ ; $D \leftarrow (D \ll m) \mid G(i, q)$
    endwhile
    $F \leftarrow (D \mid (1 \ll (m-1) - 1))$
    if $F$ then
        Scan through unset (=0) upper bits in F
        and check candidates starting
        at corresponding positions
        if end position $> n$ then Return endif
    endif
    $i \leftarrow i + q$ ; $D \leftarrow (D \ll q) \mid G(i, q)$
endwhile
```

Checking is done if any of the highest bits in D is not set. Those bits correspond to candidate positions.

Let us study an example. Let `abcdefgh` be the pattern, and let q be 4. Let us assume that the marked 4-grams have been read.

$$\ldots\mathtt{xx}\overline{\mathtt{xxab}}\mathtt{cdef}\overline{\mathtt{ghxx}}\mathtt{xx}\ldots$$

Then the rightmost bits of D are computed as shown in Fig. 1. So the candidate `abcdefgh` should be checked.

```
x:  ...00011111111
x:  ...100011111111
a:  ...1100011111110
b:  ...11100011111101
--------------------------------
g:  ...1111111100010111111
h:  ...11111111110001111111
x:  ...111111111100011111111
x:  ...1111111111100011111111
--------------------------------
D:  ...1111111111110111111111
```

Figure 1: Computation of D.

Let us consider another example. Let q be 2. When `bc` of an occurrence of the same pattern has been read, i is advanced by 2 until the end of the pattern is recognized.

Notice that unlike the other q-gram algorithms UFNDMq works reasonably also on "undersized" patterns i.e. when $q > m$. Then it must be allowed to access characters before the beginning of text or better by evaluating the first value of D separately. A disadvantage of UFNDMq is that the pattern length is limited by $q + m \leq w$.

6 Reading 2-grams

Some CPU architectures, notably the x86, allow unaligned memory reads of several bytes. This inspired us to try reading several bytes in one instruction, instead of separate character reads. One may argue that it is not fair to apply such multiple reading, because all CPU architectures do not support it. But because of the dominance of the x86 architecture it is reasonable to tune algorithms for that.

Fredriksson [5] was probably the first one who applied reading several bytes simultaneously to string matching. We adopted a similar approach by Kalsi et al. [12] to BNDMq and SBNDMq. We developed three versions for both. BNDM2b/SBNDM2b reads a 2-gram as a 16-bit halfword. The value of $B[t_i]$ & $(B[t_{i+1}] \ll 1)$ is stored to a precomputed table g for each halfword. In BNDM4b/SBNDM4b the corresponding value of 4-gram is computed as $g[x_1]$ & $(g[x_2] \ll 2)$ where x_1 and x_2 are the halfwords and g is the same table used in the 2-gram version. In BNDM6b/SBNDM6b the value of 6-gram is computed as $g[x_1]$ & $(g[x_2] \ll 2)$ & $(g[x_3] \ll 4)$. From SBNDM4b we made a modified version SBNDM2+2b, where a 4-gram is tested in two parts. If the first 2-gram do not exits in the pattern, we can shift $m - 1$ positions instead of $m - 3$ with 4-gram.

Reading more than two bytes simultaneously does not seem to give extra advantage. Based on the tests by Kalsi et al. [12], unaligned memory reads on x86 processors incur a speed penalty of up to 70% when compared with aligned reads. This unfortunately reduces the speed of reading four bytes, because then 75% of the reads are unaligned on average.

Reading 2-grams works readily on some other CPU architectures besides x86. During preprocessing we take care of endianess (the order in which integer values are stored as bytes in the computer memory). The indexing of the table g is different. On a little endian machine the bitvector is stored to $(t_{i+1} \ll 8) + t_i$ and on a big endian machine to $(t_i \ll 8) + t_{i+1}$.

7 Evaluation

Complexity. Providing $m \leq w$, the worst case time complexity of BNDM is $O(mn)$, but the average time complexity is sublinear. The space complexity of BNDM is $O(|\Sigma|)$. It is straightforward to show that BNDMq and SBNDMq inherit these complexities. Also UFNDMq is sublinear on average and $O(mn)$ in the worst case.

There exists a linear time version of BNDM [17], but it is in practice slower on average than the standard version. Therefore we did not develop linear versions of our algorithms.

Experimental results. The tests were run on a 2.8 GHz Pentium D (dual core) CPU with 1 GB of memory. Both cores have 16 KB L1 data cache and 1024 KB L2 cache. The computer was running Fedora 8 Linux. All the algorithms were tested in a testing framework of Hume and Sunday [11]. All programs were written in C and compiled with the gcc compiler 4.1.2 producing x86_64 "32-bit" and "64-bit" code and using the optimization level `-O3`.

The change of the process from one processor core to another empties cache memories with various degree. This would slow down reads from memory and induce annoying variation to the timing of test runs. To avoid it we have used Linux function `sched_setaffinity` to bind the process to only one processor or core.

We used three texts of 1 MB in our tests: English, DNA, and binary. The English text is the beginning of the KJV bible. The DNA text is from Hume and Sunday [11]. The binary text was generated randomly. For each text there were pattern sets of lengths 5, 10, 20, 30, and 50. For DNA and binary, each set contained 200 patterns taken from the same data source as the corresponding text. So every pattern do not necessary occur in the text. For English, each set contained 300 patterns drawn from non-overlapping positions of the text.

The set of tested algorithms include several classical algorithms. Besides Shift-Or [1, 4] we have two versions of BNDM: the original one and the NR-grep variation BNDMnr [16]. BM is the implementation `fast.rev.d12` of Boyer–Moore algorithm by Hume and Sunday [11] which follows original suggestions of Boyer and Moore [2] about maximal efficiency. QS is their implementation `uf.rev.sd1` of Sunday's QS algorithm [20]. KS by Kim and Shawe-Taylor [13] uses a trie of reversed q-grams of the pattern. In the tested implementation q is five. KS was designed only for DNA, and therefore it does not find all English patterns (inaccurate times are marked with a star).

We also tested some new algorithms. Lec is the 'New' algorithm of Lecroq [15], which uses q-grams and hashing. We used 256 as the size of the hash table of Lec. WW-LBNDM is an algorithm developed by He et al. [8] for large alphabets. It examines the text in regions of $2m - 1$ characters, i.e. wide windows (WW). The bit-parallel version was called LBNDM [7]. It is interesting that upper limit for characters examinations is $2n$. BLIM is Külekci's bit-parallel algorithm designed for long patterns. The tested implementation uses 32-bit vectors.

Because a preliminary version of SBNDM2 was already present in Lecroq's tests [15], we show also its run times. It is called SBNDM2x.

Results of test runs are shown in Tables 2 (32-bit) and 1 (64-bit). For the 32-bit case we used pattern sets of lengths 5, 10, 20, and 30. For the 64-bit case we used pattern sets of lengths 5, 10, 20 and 50 (5–30 for English). The times are averages of processor times of 50 runs. The data was in the main memory so that the times do not contain any I/O time. The test environment does not show the locations of occurrences. It only counts the number of occurrences. The three best times for each pattern set has been underlined.

Behavior with the 64-bit code. The speed of BNDMq is very close to that of SBNDMq for $q = 3, ..., 6$. The same is true for English and DNA in the case $q = 2$, but BNDM1 is clearly slower than SBNDM1. It is remarkable that the maximal shift of SBNDM4 is two for patterns of 5 characters (except when s_0 is applied), but the search speed is still reasonably good.

Lec$q$ was not competitive in our tests, e.g. SBNDM4 seems to be faster than Lec3 on other cases than binary patterns of 10 characters. On DNA and English the speed of Lec$q$ slows down, when q increases. Thus Lec3 is the fastest of Lec$q$ versions on those data sets. This behavior differs slightly from the results reported at [15]. On the other hand Lec$q$ works well on binary data.

SBNDM2+2b is the fastest tested algorithm for short English patterns. Its search time is less than half of that of QS. For long English patterns SBNDM4b is the fastest. Versions of Lec$q$ are slower for English patterns than SBNDMq with an equal value of q.

On DNA sequences SBNDM4b is the best for $m \leq 20$, and SBNDM6b is the best for $m > 20$. Observe the good performance of KS on long DNA patterns.

On DNA we tested SBNDM2b and Shift-Or sepa-

Table 1: Search times in milliseconds with the 64-bit code.

patterns →	English				DNA				binary			
↓ algorithm	5	10	20	30	5	10	20	50	5	10	20	50
Shift-Or	1001	1002	1002	1002	667	668	668	668	669	669	669	671
BNDM	568	468	300	224	744	412	229	109	1340	752	398	173
BNDMnr	530	435	272	184	680	375	204	89	1266	682	334	138
SBNDM2x	407	224	180	134	669	370	202	91	1554	774	359	139
WW-LBNDM	681	562	352	248	967	577	330	152	1561	1039	595	273
BNDM1	597	474	289	201	835	459	256	113	1515	877	464	201
BNDM2	222	191	172	137	526	348	206	93	1327	751	388	166
BNDM3	311	165	120	90	300	184	134	80	1081	681	344	146
BNDM4	534	191	113	87	381	145	84	51	772	555	311	138
BNDM5	1354	265	124	90	905	179	85	41	1025	415	273	134
BNDM6	—	373	149	97	—	249	98	44	—	362	205	123
SBNDM1	546	447	283	194	722	398	218	96	1339	722	354	141
SBNDM2	220	194	174	138	500	341	198	92	1233	694	340	139
SBNDM3	311	163	120	90	286	178	132	81	1015	667	337	136
SBNDM4	532	193	115	86	377	141	85	54	716	544	320	137
SBNDM5	1350	264	125	91	903	182	82	43	955	420	276	137
SBNDM6	—	363	142	94	—	242	93	42	—	352	208	124
BNDM2b	208	206	184	141	504	326	191	90	1296	732	381	164
BNDM4b	410	163	108	88	255	98	62	48	650	516	290	130
BNDM6b	—	317	136	96	—	159	65	37	—	275	174	106
SBNDM2b	202	210	183	141	448	295	172	82	1233	657	325	130
SBNDM2+2b	175	174	160	126	399	245	124	56	928	533	299	123
SBNDM4b	402	158	105	81	237	96	58	48	627	488	285	124
SBNDM6b	—	292	134	94	—	145	59	38	—	266	174	105
UFNDM3	282	174	120	96	316	211	143	77	1069	683	407	201
UFNDM5	374	198	117	87	258	140	80	47	521	439	267	124
UFNDM8	505	264	140	102	343	174	95	50	559	220	132	77
Lec3	629	282	158	123	464	207	118	75	882	511	389	372
Lec4	1047	369	189	136	717	250	127	78	971	398	236	179
Lec5	—	460	219	156	—	308	145	75	—	383	191	118
Lec6	—	560	231	162	—	376	154	79	—	417	176	93
Lec7	—	825	289	184	—	552	195	87	—	571	203	90
BM	497	340	228	182	920	667	565	464	1867	1381	989	701
QS	466	330	226	182	869	726	704	702	1618	1688	1739	1728
KS	—	*422	*211	*143	—	267	126	63	—	453	295	162

rately for $m = 2, 3, 4, 5$ (the data is not shown). We tested all the possible combinations of a, c, g, and t. SBNDM2b is faster than Shift-Or for $m \geq 2$, and SBNDM3 and SBNDM4b are still faster than SBNDM2b for $m \geq 4$. This is significant, because Shift-Or is known to be fastest for short DNA [18, Fig. 2.22, p. 39].

On short patterns the extra work of fetching s_0 instead of adding 1 seems to slow down the searching. Even larger values of q than were used in these tests work fast on long patterns.

For binary data, the optimal value of q is higher than for other tested data sets. Lec6 is the fastest for $m = 30$. For short patterns, SBNDM4b is best, and SBNDM6b is the fastest when m is around 10–20. On small alphabets the length of expected shift increases only a little for algorithms using mere the occurrence shift (e.g. QS) when patterns get longer.

The algorithms WW-LBNDM and BLIM were not competitive in our tests. The obvious reason is that they have been designed for problem settings of another kind.

Behavior with the 32-bit code. We ran the same tests using the 32-bit code in our test machine. Interestingly most algorithms (e.g. UFNDM8) were faster in the 64-bit mode while others (e.g. SBNDM2+2b) were faster in the 32-bit mode. Some of the differences are significant. A possible reason is that in 64-bit mode, there are more addressable registers.

SBNDM versions with $q > 3$ became clearly slower for $m = 5$. The noteworthy exceptions were also

Table 2: Search times in milliseconds with the 32-bit code.

patterns→	English				DNA				binary			
↓algorithm	5	10	20	30	5	10	20	30	5	10	20	30
Shift-Or	969	970	968	970	647	645	647	644	647	648	647	648
BNDM	608	462	286	197	835	460	259	183	1546	899	475	328
BNDMnr	551	448	279	188	705	381	209	145	1290	692	337	232
SBNDM2x	498	258	196	142	773	388	208	144	1627	766	348	228
WW-LBNDM	611	505	318	223	884	520	296	208	1473	940	520	363
BLIM	545	390	270	242	652	373	236	227	1247	685	389	379
BNDM1	632	489	302	214	864	479	269	188	1569	902	480	331
BNDM2	316	232	191	148	565	364	212	151	1369	759	393	272
BNDM3	480	230	146	109	409	216	146	120	1200	678	345	242
BNDM4	886	299	154	106	617	212	112	84	997	609	335	234
BNDM5	2086	382	174	116	1390	258	117	79	1508	482	296	219
BNDM6	—	522	197	133	—	351	130	90	—	462	235	184
SBNDM1	548	444	278	192	704	388	210	146	1259	700	344	234
SBNDM2	323	241	194	144	547	347	201	140	1264	678	334	224
SBNDM3	480	228	145	107	401	213	149	121	1115	662	332	223
SBNDM4	884	300	152	106	608	210	108	80	913	593	329	231
SBNDM5	2103	382	176	116	1402	258	117	78	1402	477	295	218
SBNDM6	—	530	200	133	—	355	133	87	—	460	238	187
BNDM2b	234	202	182	139	511	334	196	139	1378	758	393	268
BNDM4b	512	186	110	83	339	125	74	60	711	534	302	213
BNDM6b	—	291	123	88	—	173	69	52	—	286	185	150
SBNDM2b	193	194	173	132	450	299	177	125	1217	660	330	216
SBNDM2+2b	156	161	151	115	404	248	129	84	938	536	299	203
SBNDM4b	366	142	97	77	237	96	64	52	669	487	284	198
SBNDM6b	—	250	113	83	—	143	59	45	—	261	173	141
UFNDM3	928	512	295	204	815	474	284	210	1740	1107	666	486
UFNDM5	1106	554	281	—	745	380	191	—	1041	785	468	—
UFNDM8	1145	664	364	—	792	443	243	—	1176	517	297	—
Lec3	606	275	155	120	449	204	117	93	852	502	383	370
Lec4	980	354	183	135	675	239	122	93	919	385	226	192
Lec5	—	449	211	154	—	303	141	103	—	373	189	144
Lec6	—	559	234	164	—	374	156	111	—	411	176	128
Lec7	—	766	279	180	—	513	186	121	—	530	195	129
BM	444	324	218	171	821	608	516	463	1562	1169	838	737
QS	434	312	215	172	822	691	671	668	1555	1654	1703	1610
KS	—	*333	*160	*117	—	221	98	76	—	436	283	213

SBNDM2b, SBNDM2+2b, SBNDM4b, and SBNDM6b, which were generally faster than with the 64-bit code; especially the fastest times for English and DNA data became better. We repeated this test also with a 1.0 GHz AMD Athlon 64 X2 dual core 5000+ processor, 2 GB of memory, 64 kB L1 cache and 512 kB L2 cache. The relative performance of algorithms remained mostly the same. Moreover, we tested the algorithms in four other computers having a x86 processor (Pentium III or newer). The results were similar.

The 32-bit code of UFNDMq was dramatically slower than the 64-bit code. We tried a newer gcc 4.3.0 compiler, but results were similar. On the other hand the 32-bit code compiled with earlier gcc version 4.1.2 ran about 30% faster. We suspect that the reason for the problem is a compiler bug in optimization. In another computer, the 32-bit codes of UFNDMq compiled with Dev-C++ 4.9.9.2 run relatively faster.

We did also some preliminary testing with the 32-bit version of the FAOSO algorithm [6]. It was slower than the fastest one of our algorithms for all the pattern sets tested. The relatively best result of FAOSO was 571 milliseconds for binaries of five characters, but this did not beat 64-bit UFNDM5. Although FAOSO is fast for short patterns, it is rather unpractical. Namely it has two constant parameters and it is a tedious process to find out the best combination of them for each type of input.

Examined characters. The relative numbers of examined text characters are shown in Table 3. The value 200 means that every character is examined twice on average. The values for BNDMqb and SBNDMqb are not shown, because they are naturally the same as for the basic versions. On the given value of q the number of examined characters is correlated with the search time. When q increases, it is obvious that more characters are read from the text. Table 3 clearly shows how fuzzy the connection between the search time and the number of examined text characters is. For example, SBNDM4 is clearly faster than SBNDM1 on binary patterns of five characters, though it examines substantially more characters.

Memory requirements. All versions of BNDM need occurrence vectors B for each character. They need thus 1 kB (bitvectors of 32 bits) or 2 kB (bitvectors of 64) of memory. Moreover, BNDMqb and SBNDMqb require additional 262 kB (bitvectors of 32) or 524 kB (bitvectors of 64). The initialization of BNDMqb and SBNDMqb takes about 15–20 milliseconds per 200 patterns.

Behavior on a different processor. Although the current market share of x86 processors is over 99%, it is also necessary to try other processors. So we tested the algorithms on Sparc. The results were mixed. The new algorithms BNDMq and SBNDMq did not get similar gain as on x86 processors. However, the best version, SBNDM3 was faster on binary and DNA than old versions of BNDM. We tested also such version of SBNDM2b that never reads 2-grams that cross the word border, which is not allowed in Sparc. However, there was not significant difference between the speed of SBNDM2b and SBNDM2.

8 Concluding remarks

We have presented new variations of the BNDM and Shift-Or algorithms. Our experiments show that several variations are clearly faster than the corresponding original algorithms on x86 processors. Moreover, our algorithms seem to be faster than any previous exact string matching algorithm for English and DNA data on those processors. Therefore our algorithms will be most useful for practitioners[1]. Our algorithms work well also with short patterns which is not typical for algorithms of Boyer–Moore type.

Our algorithms can also be applied to multiple matching and approximate matching. See the book [18] for the basic techniques. Here we described algorithms only for patterns of at most w characters. Next we will work on bit-parallel algoritms for longer patterns in order to compete with BLIM [14]. The LBNDM algorithm [19] is a good starting point.

Acknowledgements

This research has been partially supported by the Ministry of Education, Youth and Sports of Czech Republic, the Czech Science Foundation, and the Academy of Finland. The authors thank K. Fredriksson, M. Külekci, T. Lecroq, and L. He for letting us to test their original codes.

References

[1] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[3] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

[4] B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics*, 3:29–46, 1964. Hungarian Academy of Science, Budapest.

[5] K. Fredriksson. Shift-or string matching with super-alphabets. *Inf. Process. Lett.*, 87(4):201–204, 2003.

[6] K. Fredriksson and Sz. Grabowski. Practical and optimal string matching. In M. Consens and G. Navarro, editors, *String Processing and Information Retrieval, 12th International Conference, SPIRE 2005*, volume 3772 of *Lecture Notes in Computer Science*, pages 376–387, Springer-Verlag, Berlin, 2005.

[7] L. He and B. Fang. Linear nondeterministic dawg string matching algorithm (abstract). In *SPIRE 2004: Proceedings of the 11th International Conference on String Processing and Information Retrieval*, volume 3246 of *Lecture Notes in Computer Science*, pages 70–71. Springer-Verlag, 2004. ISBN 978-3-540-23210-0.

[8] L. He, B. Fang, and J. Sui. The wide window string matching algorithm. *Theor. Comput. Sci.*, 332(1–3):391–404, 2005.

[9] J. Holub and B. Ďurian. Fast variants of bit parallel approach to suffix automata. URL: http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf, 2005. Talk given in The Second Haifa Annual International Stringology Researh Workshop of the Israeli Science Foundation.

[10] R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.

[11] A. Hume and D. M. Sunday. Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.

[12] P. Kalsi, H. Peltola, and J. Tarhio. Exact string matching algorithms for biological sequences. In *2nd International Conference on Bioinformatics Research and Development, BIRD 2008*, number 13 in Communications

[1] The codes of our new algorithms will be made available on the Web.

Table 3: Relative numbers of the examined text characters (100 = all once).

patterns →	English				DNA				binary			
↓ algorithm	5	10	20	30	5	10	20	30	5	10	20	30
Shift-Or	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
BNDM	26.7	16.9	10.2	7.4	44.5	26.2	15.3	11.1	86.8	51.6	29.2	21.4
BNDMnr	28.0	17.9	10.7	7.6	52.1	29.6	16.7	11.9	117.0	65.2	34.3	24.3
SBNDM2x	101.1	30.5	13.9	9.0	114.0	36.9	17.9	12.3	177.5	72.8	35.9	25.0
WW-LBNDM	24.8	16.3	10.1	7.2	37.4	24.2	14.8	10.9	55.9	41.3	26.4	20.0
BNDM1	26.7	16.9	10.2	7.4	44.6	26.2	15.3	11.1	90.0	51.7	29.2	21.4
BNDM2	50.7	23.8	12.4	8.4	59.1	29.1	16.0	11.4	99.0	53.5	29.6	21.6
BNDM3	100.2	38.0	17.5	11.4	102.9	39.6	18.6	12.5	131.1	58.3	30.6	22.0
BNDM4	200.1	57.4	23.9	15.1	201.0	57.7	24.1	15.4	217.5	69.5	32.7	22.7
BNDM5	500.0	83.4	31.4	19.4	500.1	83.5	31.4	19.4	503.1	90.4	36.6	24.1
BNDM6	—	120.0	40.1	24.1	—	120.1	40.0	24.0	—	124.1	42.9	26.8
SBNDM1	27.9	17.9	10.7	7.6	51.8	29.6	16.7	11.9	107.8	64.8	34.3	24.3
SBNDM2	50.9	24.1	12.6	8.5	61.9	30.8	16.8	11.9	109.8	64.8	34.3	24.3
SBNDM3	100.2	38.2	17.6	11.4	103.8	40.3	18.9	12.7	132.4	66.1	34.4	24.3
SBNDM4	200.0	57.4	24.0	15.2	200.8	58.0	24.2	15.4	207.4	74.1	35.3	24.4
SBNDM5	499.8	83.5	31.5	19.4	498.1	83.6	31.4	19.4	459.2	93.0	38.2	25.2
SBNDM6	—	120.0	40.1	24.1	—	120.1	40.1	24.1	—	125.3	43.9	27.4
SBNDM2b	50.9	24.1	12.6	8.5	62.0	30.8	16.8	11.9	109.8	64.8	34.3	24.3
SBNDM2+2b	51.7	25.0	13.4	9.1	70.8	35.8	19.6	13.7	143.9	71.1	35.1	24.4
SBNDM6b	—	120.1	40.1	24.1	—	120.1	40.1	24.1	—	125.3	43.9	27.4
UFNDM3	60.2	30.8	16.2	11.0	64.5	34.3	19.1	13.8	97.8	59.2	36.6	27.3
UFNDM5	100.1	50.1	25.4	17.0	100.6	50.5	25.5	17.2	115.6	63.5	37.0	27.6
UFNDM8	160.0	80.0	40.0	26.7	160.0	80.0	40.0	26.7	162.5	82.7	42.7	29.5
BM	26.4	16.8	10.7	8.2	50.1	36.4	30.7	27.5	103.8	77.1	54.8	47.8
QS	25.8	16.9	10.9	8.5	56.1	46.8	45.6	45.2	148.8	159.4	163.6	155.2

in Computer and Information Science, pages 417–426. Springer-Verlag, Berlin, 2008.

[13] J.Y. Kim and J. Shawe-Taylor. Fast string matching using an n-gram algorithm. *Softw. Pract. Exp.*, 24(1):79–88, 1994.

[14] M. O. Külekci. A method to overcome computer word size limitation in bit-parallel pattern matching. In *ISAAC 2008: Proceedings of the 19th International Symposium on Algorithm and Computation*, volume 5369 of *Lecture Notes in Computer Science*, pages 496–506. Springer, 2008.

[15] T. Lecroq. Fast exact string matching algorithms. *Inf. Process. Lett.*, 102(6):229–235, 2007.

[16] G. Navarro. NR-grep: A fast and flexible pattern-matching tool. *Softw. Pract. Exp.*, 31(13):1265–1312, 2001.

[17] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000.

[18] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7.

[19] H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In *String Processing and Information Retrieval, 10th International Symposium, SPIRE 2003*, volume 2857 of *Lecture Notes in Computer Science*, pages 80–94, Springer-Verlag, Berlin, 2003.

[20] D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.