Variations of Forward-SBNDM*

Hannu Peltola and Jorma Tarhio

Department of Computer Science and Engineering,
Aalto University, P.O.B. 15400, FI-00076 Aalto, Finland
{hannu.peltola,jorma.tarhio}@aalto.fi

Abstract. Forward-SBNDM is a recently introduced variation of the BNDM algorithm for exact string matching. Forward-SBNDM reads a text character following an alignment of the pattern. We present a generalization of this lookahead idea and apply it to SBNDMq for $q \geq 3$. As a result we get several new variations of SBNDMq. We introduce a greedy skip loop for SBNDM2. In addition, we tune up our algorithms and the reference algorithms with 2-byte read. According to our experiments, the best of the new variations are in several cases faster than the winners of recent algorithm comparisons.

Keywords: string matching, BNDM, 2-byte read, q-grams

1 Introduction

After the advent of the Shift-Or [2] algorithm, bit-parallel string matching methods have gained more and more interest. The BNDM (Backward Nondeterministic DAWG Matching) algorithm [17] is a nice example of an elegant, compact, and efficient piece of code for exact string matching. BNDM simulates the nondeterministic finite automaton of the reverse pattern even without constructing the actual automaton.

SBNDM2 [6,11] is a simplified variation of BNDM. SBNDM2 starts processing of an alignment by reading two characters. Recently Faro and Lecroq [7] introduced Forward-SBNDM, a lookahead version of the SBNDM2 algorithm. In this paper we present a generalization of the lookahead idea and give new variations of SBNDMq [6] for $q \geq 3$. SBNDMq starts processing of an alignment by reading q characters. In addition, we introduce a greedy skip loop for SBNDM2. Our point of view is practical efficiency of exact string matching algorithms. According to our experiments, the best of the new variations are in several cases faster than the winners of recent algorithm comparisons [6,9].

We use the following notations. Let a pattern $P = p_1 p_2 \cdots p_m$ and a text $T = t_1 t_2 \cdots t_n$ 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 $t_i t_{i+1} \cdots t_{i+m-1} = p_1 p_2 \cdots p_m$. In the pseudocode of the algorithms we use some notations of the programming language C: '|', '&', '~', '<<', and '>>' 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.

The rest of the paper is organized as follows. Since our work is based on SBNDMq and Forward-SBNDM, we start with presenting these algorithms in Section 2. In Section 3 we generalize Forward-SBNDM with wider lookahead and longer q-grams. In Section 4 the greedy skip loop is presented. Section 5 reviews the results of our experiments before concluding remarks in Section 6.

* Supported by the Academy of Finland (grant 134287).

Hannu Peltola, Jorma Tarhio: Variations of Forward-SBNDM, pp. 3–14.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2

2 Previous algorithms

2.1 SBNDM*q*

SBNDMq [6] is a variation of SBNDM [18], a simplified version of BNDM, applying q-grams. The pseudocode is shown as Alg. 1. $F(i, q)$ on line 6 is a shorthand notation for the expression

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

Algorithm 1 SBNDMq ($P = p_1 p_2 \cdots p_m, T = t_1 t_2 \cdots t_n$)

```
 1: for all c ∈ Σ do B[c] ← 0
 2: for j ← 1 to m do
 3:     B[p_j] ← B[p_j] | (1 << (m − j))
 4: i ← m − q + 1
 5: while i ≤ n − q + 1 do
 6:     D ← F(i, q)
 7:     if D ≠ 0 then
 8:         j ← i − (m − q + 1)
 9:         repeat
10:             i ← i − 1
11:             D ← (D << 1) & B[t_i]
12:         until D = 0
13:         if j = i then
14:             report occurrence at j + 1
15:             i ← i + s_0
16:     i ← i + m − q + 1
```

At each alignment, SBNDMq first reads q characters $t_i, \ldots, t_{i+q-1}$ before testing the state vector D. If D is zero, this q-gram (i.e., the string of q characters) is not a factor (i.e. a substring) of P, and then the pattern can be shifted forward $m - q + 1$ positions. If D is not zero, a single character at a time is read to the left until the suffix $t_i \cdots t_{j+m}$ of the alignment is not any more a factor of P. If $t_i \cdots t_{j+m}$ is not a factor of P and $i > j$ holds, the pattern is shifted forward and the next alignment starts at t_{i+1}.

In the original BNDM, the inner loop also recognizes prefixes of the pattern. The leftmost one of the found prefixes determines the next alignment of BNDM. SBNDMq does not care of prefixes, but shifts the pattern simply past the text character which nullifies D.

When an occurrence of the pattern is found, the shift is s_0, which corresponds to the distance to the leftmost prefix of the pattern in itself and which is easily computed from the pattern (see [6]). We skip the details, because a conservative value $s_0 = 1$ works well in practice. In the subsequent algorithms of this paper we use the value $s_0 = 1$.

2.2 Forward-SBNDM

Forward-SBNDM, a lookahead version of SBNDM2, was introduced by Faro and Lecroq [7]. The idea of the algorithm is the following. As in SBNDM2, a 2-gram x_1x_2 is read before testing the state vector D. In SBNDM2, x_1x_2 is matched with the end of the pattern. In Forward-SBNDM, only x_1 is matched with the end of the pattern,

Algorithm 2 Forward-SBNDM ($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
Require: 1 ≤ m < w
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← 1
 2: for j ← 1 to m do
 3:       B[p_j] ← B[p_j] | (1 << (m − j + 1))
    /* Searching */
 4: i ← m
 5: while i ≤ n do
 6:       D ← (B[t_{i+1}] << 1) & B[t_i]
 7:       if D ≠ 0 then
 8:           j ← i
 9:           repeat
10:               i ← i − 1
11:               D ← (D << 1) & B[t_i]
12:           until D = 0
13:           i ← i + m − 1
14:           if j = i then
15:               report occurrence at j + 1
16:               i ← i + 1
17:       i ← i + m
```

and x_2 is a lookahead character. By lookahead characters we mean the text characters immediately following the current alignment. Note that $B[x_2]$ can nullify several bits of D, and therefore x_2 enables longer shifts. The pseudocode of Forward-SBNDM is shown as Alg. 2.

After reading x_1x_2 in Forward-SBNDM there are three possibilities to proceed. (i) If x_1x_2 is a factor of P, reading continues leftwards. (ii) If x_1x_2 is not a factor of P and if x_1 matches the last character of P, reading continues leftwards. The extra set bit in the end of B vectors ensures that the state vector D does not get nullified in this case. (iii) If x_1x_2 is not a factor of P and if x_1 does not match the last character of P, then D becomes zero and the pattern is shifted m positions and shift is one longer than in SBNDM2.

Because the length of the occurrence vector B of each character is $m + 1$ in Forward-SBNDM, the upper limit for the pattern length is thus $w - 1$. The extra bit is the rightmost one, and its value is always one, because the lookahead character is not allowed to interfere with recognition of a valid occurrence of P.

In a way Forward-SBNDM is a cross of SBNDM2 and Sunday's QS [19]. QS was the first algorithm to use a lookahead character for shifting. Another famous algorithm using two lookahead characters is by Berry and Ravindran [3].

3 Generalization: Forward-SBNDM*q*

Ďurian et al. [5,6] reported that SBNDMq is efficient also for $q > 2$ on modern processors, although the number of read characters increases with q. This increment can be considerable in the case of short patterns, but this straightforward method is faster on average than SBNDM in most cases. Based on this observation we decided to examine whether a longer lookahead than one as in Forward-SBNDM would be beneficial for SBNDMq. So based on SBNDMq we constructed Forward-SBNDM(q, f), where the lookahead f can be any integer between 0 and $q - 1$. Our preliminary experiments

convinced us that longer lookaheads would be beneficial. The pseudocode is given as Alg. 3.

Algorithm 3 Forward-SBNDM(q, f) ($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
Require: q − f ≤ m ≤ w − f and 0 ≤ f < q
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← (~0) >> (w − f)          /* 1^f */
 2: for j ← 1 to m do
 3:     B[p_j] ← B[p_j] | (1 << (m − j + f))
    /* Searching */
 4: i ← m − q + f
 5: while i ≤ n − q + 1 do
 6:     D ← F(i, q)
 7:     if D ≠ 0 then
 8:         j ← i − (m − q + f + 1)
 9:         repeat
10:             i ← i − 1
11:             D ← (D << 1) & B[t_i]
12:         until D = 0
13:         if j = i then
14:             report occurrence at j + 1
15:             i ← i + 1
16:     i ← i + m − q + f + 1
```

Note that Forward-SBNDM(q,0) is in practice the same as SBNMDq if $s_0 = 1$ is selected. If we keep $f - q$ in a precomputed variable, then even the search part of Forward-SBNDM(q, f) is independent of the value of f. Note also that Forward-SBNDM$(2, 1)$ corresponds to the original Forward-SBNDM.

Because the length of the occurrence vector B of each character is $m + f$, the upper limit for the pattern length is thus $w - f$. In addition it is required that $0 < q - f \leq m$. When changing q, only line 6 needs to be updated. Note that like SBNDMq, Forward-SBNDM(q, f) may read a few characters beyond the text (line 6) and also one character before the text (line 11).

Providing $m \leq w$, the worst case time complexity of BNDM is $\mathcal{O}(mn)$, but the average time complexity is sublinear. The space complexity of BNDM is $\mathcal{O}(|\Sigma|)$. It is straightforward to show that Forward-SBNDM(q, f) inherits these complexities when $m \leq w - f$.

Let $t_i \cdots t_{i+q-1} = x_1 \cdots x_{q-f} y_1 \cdots y_f$ be the q-gram read on line 6. As in the case of Forward-SBNDM, there are three possibilities to proceed. (i) If $x_1 \cdots x_{q-f} y_1 \cdots y_f$ is a factor of P, reading continues leftwards. (ii) If $x_1 \cdots x_{q-f} y_1 \cdots y_f$ is not a factor of P and if $x_1 \cdots x_{q-f}$ matches the suffix of P, reading continues leftwards. The extra set bits in the end of B vectors ensure that the state vector D does not get nullified. (iii) If $x_1 \cdots x_{q-f} y_1 \cdots y_f$ is not a factor of P and if $x_1 \cdots x_{q-f}$ does not match the suffix of P, then the pattern is shifted and the next alignment ends at t_{i+m}. The shift is $m - q + f + 1$, which is f positions more than in SBNDMq.

The disadvantage of Forward-SBNDM(q, f) is that there is a larger risk to fall to the slow loop on lines 8–15, because the probability $F(i, x)$ to be nonzero is higher for $x = q - f$ than for $x = q$.

Example. Let P be `abcdefgh`. The maximal shifts of SBNDM2 and SBNDM3 are 7 and 6, respectively. The maximal shift of Forward-SBNDM(3,2) is 7. Let us consider

a text $T =$ `...xabcdey...` If SBNDM2 reads a 2-gram `de`, it scans back until `x`. If Forward-SBNDM(3,2) reads 3-gram `dey`, it immediately skips 7 positions onwards, because `d` is not a suffix of P and `dey` is not a factor of P.

Variation. The way how f lookahead characters are handled takes f low order bits in the state vector D, which reduces the maximal length of the pattern. This could be circumvented by using on line 6 a distinct occurrence vector table C_k (corresponding to B) for each of the q text positions. Then $F(i, q)$ is interpreted as

$$C_1[t_i] \,\&\, C_2[t_{i+1}] \,\&\, \cdots \,\&\, C_q[t_{i+q-1}],$$

where $C_k[x] = ((B[x] \ll f) + 2^f - 1) \gg (q - k)$ where B is B of SBNDMq as well as on line 11. Note that $2^f - 1$ ensures that the f lowest order bits are set. The justification for deleting the f high order bits by a left shift in the computation of $C_k[x]$ is that they are not needed in the algorithm, because we can assume that $q < w/2$ holds.

Implementation note. In the C language the right operand of a shift operation must be shorter than the width of the left operand. Therefore on line 1 of Alg. 3, shifting has to be made in two parts or handled e.g. with if clause, when $f = 0$.

4 Greedy skip loop

Many string matching algorithms apply so called skip loop, which is used for fast scanning before entering the matching phase. E.g. a basic skip loop of SBNDM is the following:

$$\textbf{while } B[t_i] = 0 \textbf{ do } i \leftarrow i + m.$$

Faro and Lecroq [7,8] introduce several interesting variations of skip loop. In the variation (originally for an algorithm of SBNDM2 type)

$$\textbf{while } B[t_i] = 0 \textbf{ do } i \leftarrow i + d[t_{i+m}] \qquad (1)$$

the maximal step is $2m$, where d is a shift table based on the bad character heuristics a.k.a. the occurrence heuristics. We tried several variations of (1), but we did not succeed improving the speed of our algorithms in our test setting.

Here we present a new type of skip loop for SBNDM2. We call it greedy, because in some cases it reads lookahead characters that it does not utilize. The pseudocode is given as Alg. 4.

Two 2-grams are read in the skip loop. If both do not appear in P, the shift is $2m - 2$. If the former appears in P, the latter is not read (the operator && denotes a short-circuit AND) and the computation proceeds as in SBNDM2. If only the latter 2-gram $t_k t_{k+1}$ appears in P, the next operation is a shift of $m - 2$. This means that the new former 2-gram is $t_{k-1} t_k$. Here also a shift of $m - 1$ would be possible, but that alternative is a bit slower in practice, because we already know that $t_k t_{k+1}$ is a factor of P.

It would be straightforward to generalize the greedy loop for SBNDMq. Instead of reading two 2-grams, the loop may hold reading of two q-grams or a q-gram and a 2-gram.

The form of the greedy skip loop is based on the observation that the cost of side assignments is very small. We tried several variations of the greedy loop on several processors. Unfortunately no variation was clearly the best.

Algorithm 4 Greedy-SBNDM2 ($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
Require: 1 ≤ m < w
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← 0
 2: for j ← 1 to m do
 3:       B[p_j] ← B[p_j] | (1 << (m − j))
    /* Searching */
 4: i ← m − 1
 5: while i ≤ n do
 6:       while (D ← ((B[t_{i+1}] << 1) & B[t_i])) = 0 &&
          ((B[t_{i+m}] << 1) & B[t_{i+m−1}]) = 0 do
 7:           i ← i + 2m − 2
 8:       if D ≠ 0 then
 9:           j ← i
10:           repeat
11:               i ← i − 1
12:               D ← (D << 1) & B[t_i]
13:           until D = 0
14:           i ← i + m − 1
15:           if j = i then
16:               report occurrence at j + 1
17:               i ← i + 1
18:       else
19:           i ← i + m − 2
```

5 Experimental results

We implemented Greedy-SBNDM2 and Forward-SBNDM(q, f) versions up to $q \leq 8$ and for $f \leq \min\{q-1, 5\}$. For efficiency f and q were compile time constants. For each variation we implemented two versions. The standard version corresponds otherwise to the pseudocode, but the test of the outer loop was eliminated and a copy of the pattern was placed as a stopper after the last text character t_n. The b-version applies 2-byte read, where two bytes are read with one instruction. As a result a part of bit shifts was moved to preprocessing as explained below. Otherwise the search part of the b-version is identical with the corresponding standard version.

2-byte read. Reading several bytes at a time is a well-known technique. Fredriksson [10] was probably the first who analyzed its advantage. A string matching algorithm applying 2-byte read is in practice much faster than the traditional version. In some cases the speedup becomes close to two, which is the theoretical limit. The cost of reading one or two bytes is almost the same on most x86 processors. Only crossing a word border causes small overhead [14]. A noteworthy additional advantage is the possibility to move computation from the scanning phase to preprocessing. When applying 2-byte read in an algorithm of BNDM type, we replace a C language expression `B[t[i]] & (B[t[i+1]]<<1)` by `B2[*(uint16_t*)(t+i)]`, where `(uint16_t*)` is a typecast and `t+i` is a reference (pointer) to the character `t[i]`. The table `B2` is computed during preprocessing. When processing a 4-gram, it is advantageous to process it as two separate 2-byte reads (see [6,14] for details) in order to decrease the penalty of crossing word borders. The same holds for even larger values of q.

Unaligned 2-byte reads work also on some other CPU architectures than x86. During preprocessing we take care of endianess (the order in which integer values are stored as bytes in the computer memory). Let `x` and `y` be two succes-

sive characters. The indexing of the table `B2` is different. On a little endian machine (like x86) `B2[(y<<8)+x]=B[x]&(B[y]<<1)` and on a big endian machine `B2[(x<<8)+y]=B[x]&(B[y]<<1)` is applied. If you regard 2-byte read as a machine level thing, you may accept a lighter version applying only the array of 2-byte integers. Depending on the input, `B2[(t[i+1]<<8)+t[i]]` is slightly faster than the original expression in many x86 processors.

Reference algorithms. In addition to variations of SBNDMq we tested four other algorithms:

- BR [3] by Berry and Ravindran,
- EBOM [7] by Faro and Lecroq,
- Hash3 [16] (originally New3) by Lecroq, and
- BMH2 [20,14], a 2-gram variation of the Horspool algorithm [12].

We updated each algorithm with a stopper handling and made a b-version in the same way explained above for Forward-SBNDM(q, f).

Concerning BMH2, many researchers have worked out related variations [1,15,20,21]. The basic idea has been mentioned already in the original article of Boyer and Moore [4]. BR is a cross of BMH2 and Sunday's QS algorithm [19]. In BMH2 the shift is based on the last 2-gram of the text window aligned with the pattern, whereas BR applies the 2-gram locating two positions further to the right. EBOM is an efficient implementation of the oracle automaton utilizing 2-grams.

Because Hash3 applies a 3-gram, the application of 2-byte read is a bit different. The statements

```
h = text[i-2];
h = ((h<<1) + text[i-1]);
h = ((h<<1) + text[i]);
```

are replaced by

```
h = d2[*(uint16_t*)(text+i-2)]+text[i];
```

BMH2 and BR are examples of old algorithms. BR was among the first algorithms to discredit the connection with the number of character reads and efficiency. EBOM and Hash3 are the winners of several test cases in a recent comparison [9].

We use the shorthands FSB and GSB for Forward-SBNDM and Greedy-SBNDM, respectively. FSB(q, f)b for odd q was implemented so that the q-gram is processed using $(q-1)/2$ consecutive 2-byte reads followed by one 1-byte read. Because FSB(q,0) is in practice the same as SBNMDq, $q = 2, 3, \ldots$, the former ones also serve as reference methods, because the latter ones are among the best in our recent comparison [6].

Computers and test setting. We run the main tests on two computers. The first one was IBM ThinkPad X61s having Intel Core 2 Duo Processor L7300 (32 KiB L1 data cache). The test environment was Cygwin. The second computer was a Dell Precision T1500 containing Intel Core i7-860 2.8 GHz CPU (8 KiB L1 data cache/core) running with the 64-bit Ubuntu kernel 2.6.35-30. The programs in C were compiled with the gcc compiler version 3.4.4 in IBM and 4.4.5 in Dell to run either in the 32-bit mode or in the 64-bit mode (only in Dell) using the optimization level `-O3`.

In the main tests we used three texts: English (4 MB), DNA (2 MB), and binary (2 MB). The English text was the KJV Bible. Sets of patterns of various lengths were

randomly taken from each text. Each set contained 200 patterns. Neither end of the English patterns was aligned with boundaries of English words.

All the algorithms were tested in a testing framework of Hume and Sunday [13]. The data was in the main memory so that I/O time had no effect to speed measurements. The search speeds shown are averages of 100 runs (if not otherwise told). Accuracy of the results is about 1 %. For organizational reasons the test sets of ThinkPad X61s and Dell T1500 were not identical.

With 32-bit bitvectors the maximum pattern length for FSB(*,3) is 29. Therefore some results of FSB(4,3) for length 30 are missing.

Text 1: English. The search speeds on English data are shown in Tables 1 and 2. The best speed for each pattern set has been boxed. Both GSB2 and EBOM were among the fastest standard algorithms for $m \leq 15$. Also FSB(3,1) (not tested for Table 1), FSB(4,0), FSB(4,1), and FSB(4,2) worked well for longer patterns. Among the b-versions GSB2b was good for short patterns. FSB(4,0)b, FSB(4,1)b, and FSB(4,2)b were excellent for $m \geq 7$.

As explained in Section 3, FSB(4,f), $f > 0$, was developed from SBNDM4 $\simeq$ FSB(4,0). For most values of m, one of FSB(4,f) was faster than FSB(4,0). The same was true for the b-versions, but the gain on Dell extended further. Note that for $m = 4$, FSB(4,0) and FSB(4,0)b process the whole pattern in the outer loop of the algorithm, and shift is always one! As explained in Section 4, GSB2 was developed from SBNDM2 $\simeq$ FSB(2,0). GSB2 was faster than FSB(2,0) for short patterns. The same was true for the b-versions, but the gain on ThinkPad extended further.

Note that FSB(2,1) $\simeq$ original Forward-SBNDM was slower than SBNDM2 $\simeq$ FSB(2,0). (The same was true for the b-versions.) We made an additional test with an alphabet of 128 characters in order to verify that FSB(2,1) is faster than FSB(2,0) in a text of a larger alphabet as shown in [9].

Relative speedup of 2-byte read is shown in Table 3. Numbers are arithmetic means of the speed ratios calculated from the data of Table 2. The overall average speedup was 1.52 in this test set. The speedup was the biggest for $m = 4$ and decreased as patterns get longer. Note that two of the algorithms went over the limit of two, possibly due to advantageous pipelining.

patterns→	4	7	10	15	20	30	4	7	10	15	20	30
↓algorithm	standard version						b-version with 2-byte read					
GSB2	0.74	1.22	1.52	1.84	2.03	2.42	1.26	1.80	2.00	2.24	2.44	2.92
FSB(2,0)	0.71	1.16	1.43	1.74	1.95	2.30	1.06	1.57	1.80	2.05	2.25	2.67
FSB(2,1)	0.66	0.99	1.23	1.56	1.82	2.25	0.79	1.17	1.40	1.74	2.02	2.47
FSB(4,0)	0.16	0.60	1.02	1.68	2.25	3.23	0.34	1.27	2.05	3.13	3.78	4.71
FSB(4,1)	0.31	0.73	1.15	1.78	2.34	3.27	0.63	1.49	2.18	3.17	3.78	4.73
FSB(4,2)	0.43	0.85	1.23	1.81	2.40	3.29	0.85	1.61	2.26	3.11	3.78	4.59
FSB(4,3)	0.48	0.81	1.12	1.64	2.12	–	0.72	1.21	1.63	2.31	2.88	–
BMH2	0.38	0.63	0.86	1.13	1.39	1.76	0.71	1.18	1.64	2.20	2.66	3.19
BR	0.57	0.83	1.07	1.42	1.88	2.40	0.68	0.98	1.23	1.74	2.16	2.67
Hash3	0.19	0.47	0.73	1.18	1.59	2.31	0.22	0.55	0.85	1.36	1.82	2.59
EBOM	0.84	1.26	1.50	1.74	1.89	2.17	1.15	1.62	1.77	1.96	2.10	2.39

Table 1. Searching speed of algorithms GB/s (per a single pattern) using English text and patterns on ThinkPad X61s.

patterns→	4	7	10	15	20	30	4	7	10	15	20	30
↓algorithm	standard version						b-version with 2-byte read					
GSB2	1.41	2.23	2.67	3.23	3.68	4.35	2.15	3.11	3.59	4.13	4.47	5.21
FSB(2,0)	1.24	2.04	2.60	3.24	3.76	4.55	1.99	2.97	3.50	4.09	4.49	5.29
FSB(2,1)	1.12	1.71	2.15	2.79	3.29	4.08	1.52	2.20	2.68	3.34	3.92	4.65
FSB(3,1)	1.03	1.92	2.67	3.77	4.69	6.21	1.86	3.29	4.35	5.69	6.68	7.94
FSB(4,0)	.300	1.16	1.97	3.22	4.33	6.37	.568	2.13	3.51	5.43	7.05	9.51
FSB(4,1)	.565	1.37	2.11	3.28	4.35	6.34	1.42	3.26	4.78	7.02	8.57	10.4
FSB(4,2)	.802	1.56	2.26	3.35	4.44	6.28	1.86	3.48	4.80	6.73	8.28	10.1
FSB(4,3)	.831	1.40	1.95	2.85	3.79	–	1.32	2.16	2.93	4.24	5.51	–
BMH2	.710	1.18	1.60	2.16	2.75	3.52	1.34	2.27	3.13	4.37	5.48	7.03
BR	1.09	1.59	2.06	2.88	3.65	4.78	1.24	1.81	2.35	3.27	4.19	5.43
Hash3	.414	1.02	1.60	2.53	3.39	5.01	.436	1.07	1.67	2.64	3.53	5.23
EBOM	1.23	1.99	2.48	3.07	3.49	4.15	1.60	2.42	2.91	3.45	3.84	4.51

Table 2. Searching speed of algorithms GB/s (per a single pattern) using English text and patterns on Dell T1500 in 32-bit mode using 32 bit bitvectors.

algorithm	speedup
GSB2	1.32
FSB(2,0)	1.34
FSB(2,1)	1.24
FSB(3,1)	1.56
FSB(4,0)	1.72
FSB(4,1)	2.15
FSB(4,2)	2.03
FSB(4,3)	1.51
BMH2	1.96
BR	1.14
Hash3	1.05
EBOM	1.17

Table 3. Average speedup of 2-byte read based on Table 2.

Text 2: DNA. The search speeds are shown in Tables 4 and 5. On DNA data, larger values of q were better than on natural language. On the other hand the probability to fall to the slow loop, i.e. the inner loop of an algorithm, increases with f.

According to Table 4 GSB2 was slightly faster than FSB(2,0) in every case, and FSB(4,1) was better than FSB(4,0) for short patterns. Otherwise the lookahead versions of FSB(4,0) were not significantly better than FSB(4,0). FSB(2,1) was faster than FSB(2,0) for longer patterns, but neither of them was then competitive with faster algorithms.

Table 5 shows that the lookahead versions were in many cases clearly faster than the versions without lookahead for $m = 10, 20, 30$.

Text 3: Binary. The search speeds are shown in Tables 6 and 7. Large values of q were good with binary data.

The relatively good performance of FSB(4,3) in Table 6 is surprising. With FSB(4,3) only one text character comes from the alignment, and therefore the probability to fall to the slow loop is quite high. Among the standard versions, BMH2 was the fastest for $m = 4$.

Results in Table 7 indicate that 8-grams worked best for $m \geq 20$, and lookahead characters gave clear advantage only for $m = 10$.

patterns→	4	7	10	15	20	30	4	7	10	15	20	30
↓algorithm	standard version						b-version with 2-byte read					
GSB2	0.33	0.49	0.62	0.85	1.10	1.58	0.44	0.59	0.72	0.95	1.24	1.76
FSB(2.0)	0.33	0.47	0.60	0.83	1.06	1.50	0.40	0.55	0.69	0.93	1.21	1.69
FSB(2.1)	0.28	0.46	0.63	0.88	1.14	1.57	0.30	0.49	0.67	0.96	1.22	1.72
FSB(4.0)	0.16	0.57	0.94	1.50	1.94	2.64	0.34	1.17	1.81	2.64	3.04	3.69
FSB(4.1)	0.28	0.66	1.01	1.51	1.93	2.53	0.52	1.19	1.75	2.45	2.86	3.43
FSB(4.2)	0.32	0.63	0.94	1.35	1.72	2.32	0.46	1.01	1.49	2.04	2.48	3.05
FSB(4.3)	0.23	0.44	0.65	1.01	1.27	–	0.30	0.57	0.83	1.27	1.54	–
BMH2	0.32	0.51	0.64	0.84	0.94	1.10	0.48	0.74	0.86	1.20	1.34	1.54
BR	0.25	0.34	0.39	0.54	0.59	0.68	0.27	0.37	0.46	0.59	0.65	0.74
Hash3	0.17	0.41	0.65	1.00	1.31	1.80	0.21	0.50	0.79	1.22	1.59	2.12
EBOM	0.34	0.44	0.54	0.70	0.88	1.20	0.37	0.48	0.58	0.75	0.93	1.27

Table 4. Searching speed of algorithms GB/s (per a single pattern) using DNA text and patterns on ThinkPad X61s.

patterns→	10	20	30	40	50	60	10	20	30	40	50	60
↓algorithm	standard version						b-version with 2-byte read					
GSB2	1.19	2.01	2.92	3.80	4.69	5.60	1.22	2.14	3.08	3.91	4.90	5.69
FSB(4,0)	2.25	4.42	5.72	6.62	7.37	8.26	3.42	5.79	6.91	7.96	8.68	9.66
FSB(4,1)	2.25	4.19	5.37	6.40	7.13	7.88	3.41	5.65	6.66	7.78	8.53	9.48
FSB(5,0)	1.81	4.46	6.55	8.37	9.53	10.9	2.77	6.27	8.52	10.5	11.8	13.2
FSB(5,1)	2.04	4.59	6.63	8.34	9.51	10.7	3.05	6.40	8.61	10.4	11.9	13.3
FSB(6,0)	1.26	3.62	5.76	7.77	9.44	11.0	2.49	6.88	10.2	12.6	14.5	16.6
FSB(6,1)	1.55	3.95	6.18	8.17	9.69	11.2	2.92	7.16	10.3	12.8	14.5	16.8
FSB(6,2)	1.76	4.11	6.29	8.20	9.72	11.2	3.20	7.14	10.2	12.5	14.4	16.1
FSB(6,3)	1.86	4.07	6.05	7.89	9.31	10.8	3.07	6.59	9.17	11.4	13.5	15.2
FSB(7,0)	.882	3.02	5.04	7.00	8.68	10.2	1.56	5.25	8.53	11.5	13.1	15.2
FSB(7,1)	1.10	3.23	5.21	7.16	8.89	10.3	1.93	5.60	8.96	11.6	13.2	15.0
FSB(7,2)	1.31	3.38	5.34	7.23	8.86	10.4	2.27	5.81	9.06	11.6	13.4	15.1
FSB(7,3)	1.49	3.54	5.46	7.33	8.93	10.4	2.57	6.04	9.13	11.5	13.1	14.7
BMH2	1.27	1.89	2.15	2.41	2.45	2.60	1.89	2.79	3.16	3.55	3.59	3.81
BR	.805	1.11	1.26	1.43	1.44	1.50	.859	1.20	1.35	1.53	1.55	1.60
Hash3	1.35	2.72	3.67	4.41	4.93	5.41	1.36	2.90	3.97	4.83	5.39	5.93
EBOM	1.09	1.76	2.38	2.99	3.51	4.09	1.13	1.84	2.46	3.08	3.66	4.20

Table 5. Searching speed of algorithms GB/s (per a single pattern) using DNA text and patterns on Dell T1500. Speeds are averages of 300 runs with 64-bit code.

patterns→	4	7	10	15	20	30	4	7	10	15	20	30
↓algorithm	standard version						b-version with 2-byte read					
GSB2	.127	.213	.301	.457	.588	.917	.156	.238	.322	.485	.578	.946
FSB(2,0)	.137	.210	.289	.434	.561	.878	.127	.211	.299	.458	.578	.937
FSB(2,1)	.139	.219	.309	.465	.600	.905	.125	.218	.319	.484	.645	.962
FSB(4,0)	.131	.249	.334	.470	.606	.886	.244	.321	.426	.571	.721	1.03
FSB(4,1)	.146	.243	.337	.481	.621	.902	.178	.298	.414	.574	.736	1.05
FSB(4,2)	.128	.228	.327	.479	.631	.918	.145	.264	.393	.579	.750	1.07
FSB(4,3)	.124	.222	.323	.492	.652	–	.121	.239	.366	.548	.724	–
BMH2	.164	.187	.199	.208	.215	.212	.207	.231	.247	.246	.267	.263
BR	.115	.120	.130	.127	.130	.129	.129	.136	.132	.140	.140	.139
Hash3	.110	.204	.265	.331	.356	.384	.124	.226	.294	.367	.394	.425
EBOM	.122	.175	.231	.331	.430	.618	.120	.180	.237	.335	.437	.633

Table 6. Searching speed of algorithms GB/s (per a single pattern) using binary text and patterns on ThinkPad X61s.

patterns→	10	20	30	40	50	60	10	20	30	40	50	60
↓algorithm	standard version						b-version with 2-byte read					
GSB2	.586	1.15	1.68	2.20	2.70	3.20	.574	1.15	1.70	2.24	2.77	3.29
FSB(4,0)	.661	1.16	1.73	2.31	2.88	3.44	.739	1.30	1.91	2.53	3.09	3.68
FSB(4,1)	.632	1.19	1.76	2.34	2.90	3.46	.722	1.34	1.96	2.57	3.16	3.72
FSB(5,0)	.846	1.32	1.76	2.29	2.81	3.36	.978	1.48	1.98	2.59	3.19	3.78
FSB(5,1)	.815	1.31	1.76	2.28	2.82	3.33	.945	1.48	2.03	2.63	3.24	3.84
FSB(6,0)	.870	1.66	2.14	2.60	3.03	3.48	1.27	2.12	2.59	3.02	3.52	4.03
FSB(6,1)	.960	1.73	2.19	2.63	3.07	3.53	1.29	2.10	2.56	3.07	3.57	4.03
FSB(6,2)	.909	1.64	2.14	2.59	3.06	3.54	1.16	1.97	2.49	2.99	3.53	4.09
FSB(6,3)	.776	1.46	1.99	2.50	3.01	3.53	.935	1.73	2.31	2.89	3.49	4.07
FSB(7,0)	.755	1.99	2.80	3.39	3.86	4.29	1.22	2.85	3.72	4.33	4.69	5.13
FSB(7,1)	.874	2.03	2.77	3.39	3.84	4.32	1.38	2.87	3.73	4.29	4.69	5.04
FSB(7,2)	.935	1.99	2.71	3.32	3.82	4.26	1.40	2.74	3.54	4.14	4.50	4.95
FSB(7,3)	.883	1.83	2.53	3.14	3.65	4.16	1.24	2.45	3.20	3.81	4.26	4.78
FSB(7,4)	.787	1.58	2.25	2.84	3.40	3.91	.970	1.96	2.71	3.35	3.88	4.41
FSB(8,0)	.543	2.02	3.14	4.06	4.76	5.36	1.05	3.45	4.99	6.00	6.49	7.16
FSB(8,1)	.696	2.10	3.18	4.05	4.77	5.35	1.30	3.53	5.01	5.93	6.56	7.14
FSB(8,2)	.812	2.14	3.17	4.00	4.71	5.28	1.45	3.49	4.90	5.84	6.39	6.90
BMH2	.369	.375	.373	.371	.383	.384	.496	.502	.500	.497	.511	.513
BR	.228	.214	.232	.223	.222	.233	.244	.229	.248	.239	.237	.248
Hash3	.610	.820	.834	.796	.832	.865	.613	.826	.847	.843	.872	.874
EBOM	.422	.767	1.09	1.37	1.64	1.87	.436	.781	1.11	1.39	1.68	1.91

Table 7. Searching speed of algorithms GB/s (per a single pattern) using binary text and patterns on Dell T1500. Speeds are averages of 300 runs with 64-bit code.

Other processors. We tested the algorithms also in several other computers having a x86 processor (Pentium III or newer). The relative performance of the algorithms was mostly similar. The only exception was Atom N450, on which BMH2b was a clear winner.

Memory usage and preprocessing time. All b-versions using 2-byte read require additional 262 kB (bitvectors of 32) or 524 kB (bitvectors of 64). The initialization of the additional table takes about 15–20 milliseconds per 200 patterns. Preprocessing of Forward-SBNDM(q, f) is more laborious when $f > 0$. In our tests the preprocessing time increased at most 6 %.

6 Concluding remarks

For long we believed that the tuned algorithms of Hume and Sunday [13] were the final solution for exact string matching of natural language. Only long patterns offered space for improvement. But the development of processor technology changed the situation: new algorithms, especially those applying bit-parallelism, can be much faster than the old ones.

In this paper, we have presented a generalization of the Forward-SBNDM algorithm and introduced the Greedy-SBNDM2 algorithm. We have shown that the new algorithms are competitive for several pattern lengths in three types of text. Generally the number of lookahead characters f has smaller influence than the q-gram size. Lookahead characters can appreciably increase the shift length in the case of pattern lengths $q - f \leq m \leq 3q$ and thus give significant improvement to the search speed.

In addition we tested the effect of 2-byte read. The speedup of 2-byte read varied from a few percents to over two. It is clear that 2-byte read should be used whenever it is possible.

When comparing the search speed of two string matching algorithms, several factors affect the result: processor, compiler, stage of tuning, text, pattern. Even a small change in the pattern may switch the order of the algorithms. Thus there is no absolute truth which algorithm is better. Because the continuing development of processor and compiler technologies, it is also difficult to anticipate, how present algorithms manage after a few years. We have experienced several times how the speed order of old algorithms has changed when switching to a new computer.

References

1. R. Baeza-Yates: Improved string searching. *Softw. Pract. Exp.*, 19(3):257–271, 1989.
2. R. Baeza-Yates and G. Gonnet: A new approach to text searching. Commun. ACM 35(10):74–82, 1992.
3. T. Berry and S. Ravindran: A fast string matching algorithm and experimental results. Proc. of the Prague Stringology Club Workshop '99, Czech Technical University, Prague, Czech Republic, Collaborative Report DC-99-05, pp. 16–28, 1999.
4. R. S. Boyer and J. S. Moore: A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
5. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: Tuning BNDM with q-grams. In *Proc. ALENEX09, Tenth Workshop on Algorithm Engineering and Experiments*: 29–37, 2009.
6. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: Improving practical exact string matching. *Information Processing Letters* 110(4):148–152, 2010.
7. S. Faro and T. Lecroq: Efficient variants of the backward-oracle-matching algorithm. *International Journal of Foundations of Computer Science* 20(6): 967–984, 2009.
8. S. Faro and T. Lecroq: An efficient matching algorithm for encoded DNA sequences and binary strings. In *Proc. CPM 2009, Combinatorial Pattern Matching, 20th Annual Symposium*, LNCS 5577: 106–115, Springer, 2009.
9. S. Faro and T. Lecroq: The exact string matching problem: a comprehensive experimental evaluation. CoRR abs/1012.2547, 2010.
10. K. Fredriksson: Shift-or string matching with super-alphabets. *Information Processing Letters*, 87(4):201–204, 2003.
11. J. Holub and B. Ďurian: Fast variants of bit parallel approach to suffix automata. Presentation in: *The Second Haifa Annual International Stringology Research Workshop.*
12. R. N. Horspool: Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.
13. A. Hume and D. M. Sunday: Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.
14. P. Kalsi, H. Peltola, and J. Tarhio: Exact string matching algorithms for biological sequences. In *Proc. BIRD 2008, 2nd International Conference on Bioinformatics Research and Development*, Communications in Computer and Information Science 13:417–426, Springer, 2008.
15. J. Y. Kim and J. Shawe-Taylor: Fast string matching using an n-gram algorithm. *Softw. Pract. Exp.*, 24(1):79–88, 1994.
16. T. Lecroq: Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, 2007.
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. H. Peltola and J. Tarhio: Alternative algorithms for bit-parallel string matching. In *Proc. SPIRE'03, 10th International Conference on String Processing and Information Retrieval*, *Lecture Notes in Computer Science* 2857:80–93, 2003.
19. D. M. Sunday: A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.
20. J. Tarhio and H. Peltola: String matching in the DNA alphabet. *Softw. Pract. Exp.*, 27(7):851–861, 1997.
21. R. F. Zhu and T. Takaoka: On improving the average case of the Boyer–Moore string matching algorithm. *Journal of Information Processing*, 10(3):173–177, 1987.