Improving Practical Exact String Matching*

Branislav Říbarov† Jan Holub‡ Hannu Peltola§ Jorma Tarhio§

Abstract

We present improved variations of the BNDM algorithm for exact string matching. At each alignment our bit-parallel algorithms process a q-gram before testing the state variable. In addition we apply reading a 2-gram in one instruction. Our point of view is practical efficiency of algorithms. Our experiments show that the new variations are faster than earlier algorithms in many cases.

Keywords: algorithms, string matching, bit-parallelism, experimental comparison

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 improve time needed for searching.

The Boyer–Moore algorithm [3] with its many variations is a widely known solution for exact string matching. Horspool's algorithm [10] and Sunday’s Quick Search algorithm (QS) [11, 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 the classical algorithms is not any more clear.

An elegant way of reaching the asymptotic optimum average time complexity is the Backward DAWG Matching algorithm (BDM) [4]. 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. The Backward Oracle Matching algorithm [1, 6], a simplified version of BDM, is in practice faster. Another faster variation is BNDM (Backward Nondeterministic DAWG Matching) by Navarro and Raffinot [18]. BNDM is a kind of cross of the BDM and Shift-Or [2, 5] algorithms. The idea is similar as in BDM, while instead of building a deterministic automaton, a nondeterministic automaton is simulated with bit-parallelism even without constructing it.

In this paper we present new variations of the BNDM algorithm. Our point of view is practical efficiency of algorithms. At each alignment of the pattern 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 35% of that of

---

*A preliminary version of this paper appeared in Proceedings of ALENEX 2009, Tenth Workshop on Algorithm Engineering and Experiments.
†Sk&T Varias s.r.o., Priemyselná 2, SK-010 01 Žilina, Slovakia; branislav.ribarov@nt.sk
‡Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Kolejná 550/2, CZ-160 00, Prague 6, Czech Republic; jan.holub@fit.cvut.cz
§Department of Computer Science and Engineering, Helsinki University of Technology, P.O.B. 5400, FI-02015 TKK, Finland; {hpeltola, tarhio}@cs.hut.fi

1
QS for English patterns of five characters. In particular, our algorithms beat clearly the winner of the recent state-of-the-art comparison [16]. The gain of our algorithms is large for patterns of lengths 5–30 which are the most interesting in practice.

We use the following notations. Let a pattern \( P = p_1p_2...p_m \) and a text \( T = t_1t_2...t_n \) be two strings over a finite alphabet \( \Sigma \). The task of exact string matching is to find all occurrences of \( P \) in \( T \). Formally we search for all positions \( i \) such that \( t_i t_{i+1}...t_{i+m-1} = p_1p_2...p_m \). In the algorithms we use C-like notations: ‘|’, ‘&’, and, ‘≪’ represent bitwise operations OR, AND, and left shift respectively. The register width (or informally speaking word size) of a processor, typically 32 or 64, is denoted by \( w \).

2 SBNDM

Algorithm 2.1 (BNDM)

```plaintext
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 \& B[t_{i+j}]
        j \leftarrow j - 1
        if \ D \& (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
```

Algorithm 2.2 (SBNDMq)

```plaintext
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. 2.3
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 - (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
```

2
In BNDM [18] (see Alg. 2.1) the precomputed table \( B \) associates each character 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, or if no prefix is found, over the current alignment. 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 standard BNDM works only for patterns which are not longer than \( w \).

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 explicitly care of prefixes, but shifts the pattern simply over the text character which caused \( D \) to become zero. In practice SBNDM is slightly faster than BNDM especially for short patterns [19]. Independently, Navarro [17] has already earlier utilized a similar approach in the code of his NR-grep.

SBNDM\(_q\) is a revised version of SBNDM applying \( q \)-grams. The pseudocode is shown as Alg. 2.2, where \( F(i, q) \) is a shorthand notation for instructions
\[
B[t_i] \& (B[t_{i+1}] \ll 1) \& \cdots \& (B[t_{i+q-1}] \ll (q - 1)).
\]

The inner loop of BNDM contain two tests per a text character. The inner loop of SBNDM\(_q\) has only one test. When removing the test of \( j \), the loop runs in the case of a match one position further to the left than in BNDM. The loop does not go any further, because the \( w - m \) leftmost bits of each \( B[a] \) are zeros, and the \( m \) rightmost bits of \( D \) are zeros because of shifting left for \( m \) bits. 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.

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 \( P = \text{abcab} \). If the proportional number of matches is not high, the algorithm runs in practice equally fast or even faster with the conservative value \( s_0 = 1 \). The computation of \( s_0 \) is shown as Alg. 2.3.

**Algorithm 2.3 (Computing \( s_0 \))**

\[
S \leftarrow B[p_m]; s_0 \leftarrow m
\]

\[
\text{for } i \leftarrow m - 1 \text{ downto } 1 \text{ do}
\]

\[
\text{if } S \& (1 \ll (m - 1)) \neq 0 \text{ then } s_0 \leftarrow i \text{ endif}
\]

\[
S \leftarrow (S \ll 1) \& B[p_i]
\]

\[
\text{endfor}
\]

As an example we give a compact C implementation of the main loop of SBNDM\(_q\) in Alg. 2.4. Because of clearness and compactness, this code differs slightly from Alg. 2.2. 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 (\texttt{matches}).

### 3 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 [7] was probably the first one who applied reading several bytes simultaneously to string matching. Hyvrö [12] has successfully tried this technique with BNDM. We adopted an
Algorithm 2.4 (SBNDM2.c)

```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-2;
    if (i == j)
        if (i > n) return (nmatches);
        nmatches++;
    i++;
}
```

approach by Kalsi et al. [13] to SBNDMq. We implemented five versions. 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. The second line of Alg. 2.4 will then be

```c
while (!(D = g[*((uint16_t*)(t+i-1))]))
```

In 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 SBNDM6b the value of 6-gram is computed as $g[x_1] \& (g[x_2] \ll 2) \& (g[x_3] \ll 4)$. SBNDM8b works in a corresponding manner. 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.

All our SBNDMq versions apply 2-gram reading. Reading more than two bytes simultaneously does not seem to give extra advantage. Based on the tests by Kalsi et al. [13], crossing the 32-bit border incur a speed penalty of up to 70% to memory reads on x86 processors. This reduces the speed of reading four bytes, because then 75% of the reads cross the border on average.

Reading 2-grams works readily on some other CPU architectures besides x86. During preprocessing one should 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 index is $(t_{i+1} \ll 8) + t_i$ and on a big endian $(t_i \ll 8) + t_{i+1}$, respectively.

4 Experimental results

The tests were run on a 2.8 GHz Pentium D CPU (dual core, family 15, model 4) with 1 GiB1 of memory. Both cores have 16 KiB L1 data cache and 1024 KiB 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 using the optimization level -O3 with the gcc compiler 4.1.2 producing x86_64 “64-bit” code. In the tests only one core was used. The size of bitvectors was 32.

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 and patterns are from Hume and Sunday [11]. The binary text was written randomly. For each text there were pattern sets of lengths 5, 10, 20, and 30. All the pattern sets contained 200 patterns taken from the same data source as the corresponding text. So every pattern do not necessary occur in the text.

The set of tested algorithms include several classical algorithms. Besides Shift-Or [2, 5] we have two versions of BNDM: the original one and the NR-grep variation BNDMnr [17]. BM is the implementation fast.rev.d12 of Boyer–Moore algorithm by Hume and Sunday [11]. QS is their implementation uf.rev.sd1 of Sunday’s QS algorithm [20]. We also tested Lecq, the ‘New’

---

1 Ki (=2^10) and Gi (=2^30) are prefixes of the IEEE 1541 standard.
algorithm of Lecroq [16], which is a q-gram variant of Horspool’s algorithm [10]. FSO is our tuned version of the Fast-Shift-Or algorithm [8] with 64-bit bitvectors.

The results of the test runs are shown in Table 1. The times are averages of the processor times of 100 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 best time for each pattern set has been boxed.

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

<table>
<thead>
<tr>
<th>patterns—algorithm</th>
<th>English</th>
<th>DNA</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 10 20 30</td>
<td>5 10 20 30</td>
<td>5 10 20 30</td>
</tr>
<tr>
<td>Shift-Or</td>
<td>668 667 668 668</td>
<td>667 668 668 668</td>
<td>670 669 669 669</td>
</tr>
<tr>
<td>FSO</td>
<td>188 188 188</td>
<td>195 188 187 188</td>
<td>125 194 189 188</td>
</tr>
<tr>
<td>BNDM</td>
<td>385 312 201 146</td>
<td>751 415 229 160</td>
<td>1340 741 389 269</td>
</tr>
<tr>
<td>BNDMar</td>
<td>350 290 183 125</td>
<td>687 376 205 141</td>
<td>1270 682 335 230</td>
</tr>
<tr>
<td>SBNDM1</td>
<td>367 301 188 128</td>
<td>714 389 211 146</td>
<td>1257 763 343 233</td>
</tr>
<tr>
<td>SBNDM2</td>
<td>145 128 116 91</td>
<td>493 339 198 142</td>
<td>1239 694 339 229</td>
</tr>
<tr>
<td>SBNDM3</td>
<td>209 109 81 60</td>
<td>289 180 133 109</td>
<td>1041 668 338 228</td>
</tr>
<tr>
<td>SBNDM4</td>
<td>358 128 77 29</td>
<td>377 141 85 68</td>
<td>726 550 320 224</td>
</tr>
<tr>
<td>SBNDM5</td>
<td>903 176 55 58</td>
<td>903 179 82 61</td>
<td>958 401 274 209</td>
</tr>
<tr>
<td>SBNDM6</td>
<td>— 241 93 64</td>
<td>— 242 92 64</td>
<td>— 350 212 173</td>
</tr>
<tr>
<td>SBNDM8</td>
<td>— 502 141 87</td>
<td>— 503 141 87</td>
<td>— 525 165 113</td>
</tr>
<tr>
<td>SBNDM2+2b</td>
<td>125 130 117 88</td>
<td>449 296 175 125</td>
<td>1203 647 320 212</td>
</tr>
<tr>
<td>SBNDM4+2b</td>
<td>105 108 100 76</td>
<td>400 245 124 80</td>
<td>933 532 298 201</td>
</tr>
<tr>
<td>SBNDM4b</td>
<td>246 92 63 50</td>
<td>236 97 60 53</td>
<td>650 490 283 197</td>
</tr>
<tr>
<td>SBNDM6b</td>
<td>— 166 77 55</td>
<td>— 145 60 43</td>
<td>— 262 175 141</td>
</tr>
<tr>
<td>SBNDM8b</td>
<td>— 325 93 66</td>
<td>— 277 77 53</td>
<td>— 300 105 81</td>
</tr>
<tr>
<td>Lec3</td>
<td>419 189 104 80</td>
<td>465 207 117 92</td>
<td>881 514 389 377</td>
</tr>
<tr>
<td>Lec4</td>
<td>698 246 126 92</td>
<td>718 251 127 94</td>
<td>971 397 235 199</td>
</tr>
<tr>
<td>Lec5</td>
<td>— 307 147 105</td>
<td>— 309 147 106</td>
<td>— 385 191 146</td>
</tr>
<tr>
<td>Lec6</td>
<td>— 374 154 110</td>
<td>— 375 154 111</td>
<td>— 414 174 128</td>
</tr>
<tr>
<td>Lec7</td>
<td>— 550 130 125</td>
<td>— 551 195 125</td>
<td>— 572 203 131</td>
</tr>
<tr>
<td>BM</td>
<td>330 226 150 121</td>
<td>921 670 568 511</td>
<td>1864 1381 989 871</td>
</tr>
<tr>
<td>QS</td>
<td>310 219 150 121</td>
<td>869 726 706 701</td>
<td>1614 1691 1740 1645</td>
</tr>
</tbody>
</table>

SBNDM2+2b was the fastest among the tested algorithms for English patterns of 5 characters. For longer English patterns SBNDM4b was the fastest. Because our English patterns contain spaces, we ran separate tests (the times are not shown) on the fixed width pattern sets of Hume and Sunday [11] without spaces. In this test, SBNDM2+2b was the fastest for pattern lengths 4–13. Its search time was in the range of 35–67% of that of QS.

On the DNA patterns, FSO was the best for \( m = 5 \), SBNDM4b for \( m = 10 \), and SBNDM6b for \( m = 20 \) and 30, respectively. On the binary patterns, FSO was the best for \( m = 5 \) and 10, and SBNDM8b was the best for \( m = 20 \) and 30, respectively.

We did also some testing with the FAOSO algorithm [8]. It was slower than the fastest one of our algorithms for all the pattern sets tested. In addition we tested several other algorithms [9, 14, 15], but they were not among the best ones for any pattern set.

**Behavior with the 32-bit code.** We ran the same tests using the 32-bit code in our test machine. Most algorithms were faster in the 64-bit mode while e.g. the Lecq versions were slightly faster in the 32-bit mode.

**Memory requirements.** All the versions of SBNDMq need occurrence vectors \( B \) for each character. They need thus 1 KiB (while using 32-bit bitvectors and 2 KiB with 64-bit bitvectors).
memory. Moreover, each SBNDMqb requires additional 256 (or 512) KiB. The initialization of each SBNDMqb takes about 14–15 milliseconds per 200 patterns.

**Behavior on a different processor.** We tested the algorithms in six other computers having a x86 processor (Pentium III or newer). The relative performance of algorithms was mostly the same. The only exception was Atom N270, on which the relative speed of the new algorithms was slower.

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. 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.

5 Concluding remarks

We have presented new variations of the BNDM algorithm. Our experiments show that most variations are clearly faster than the original BNDM on x86 processors. Moreover, our algorithms seem to be faster than any previous exact string matching algorithm for many cases on those processors. Therefore our algorithms will be most useful for practitioners. Our algorithms work well even with short patterns which is not typical for algorithms of Boyer–Moore type.

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.

References


