Improving practical exact string matching

Branislav Ďurian, Jan Holub, Hannu Peltola, Jorma Tarhio

A S&T Varias s.r.o., Priemyselná 2, SK-01001 Žilina, Slovakia
b Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Kolejní 550/2, CZ-160 00, Prague 6, Czech Republic
c Department of Computer Science and Engineering, Helsinki University of Technology, PO Box 5400, FI-02015 TKK, Finland

ARTICLE INFO
Article history:
Received 23 June 2009
Received in revised form 11 November 2009
Accepted 24 November 2009
Available online 26 November 2009
Communicated by W.-L. Hsu

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 straightforward 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 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.

© 2009 Elsevier B.V. All rights reserved.
sults 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 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 $P = p_1 p_2 \ldots p_m$ and a text $T = t_1 t_2 \ldots t_n$ 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 $t_i t_{i+1} \ldots t_{i+m-1} = p_1 p_2 \ldots p_m$. In the algorithms we use C-like notations: ‘’, ‘&’, and ‘$\ll$’ 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. $\text{SBNDM}_q$

In BNDM [18] (see Algorithm 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. Pelton 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

**Algorithm 2.1 (BNDM)**

```c
for a ∈ $\Sigma$ do $B[a] \leftarrow 0$ endfor
for j ← 1, m do
    $B[p_j] \leftarrow B[p_j] \ll (m - j)$ endfor
i ← 0
while i ≤ n - m do
    j ← m; last ← m; D ← (1 ≤ m) - 1
    while D ≠ 0 do
        D ← $D \& B[t_{i+1}]$
        j ← j - 1
        if D & (1 <= (m - 1)) ≠ 0 then
            if j > 0 then last ← j
            else report occurrence at i + 1 endif
        endif
        D ← D - 1
    endwhile
    i ← I + last
endwhile
```

**Algorithm 2.2 (SBNDM$_q$)**

```c
for a ∈ $\Sigma$ do $B[a] \leftarrow 0$ endfor
for j ← 1, m do
    $B[p_j] \leftarrow B[p_j] \ll (m - j)$ endfor
Compute $s_0$ with Algorithm 2.3
i ← m - q + 1
while i ≤ n - q + 1 do
    D ← $F(i, q)$
    if D ≠ 0 then
        j ← i - (m - q + 1)
        do i ← i - 1
        D ← (D ≤ 1) & $B[t_i]$
        while D ≠ 0
        if j = 1 then
            report occurrence at j + 1
        endif
        i ← i + $s_0$
    endif
    i ← I + m - q + 1
endwhile
```

**Algorithm 2.3 (Computing $s_0$)**

```c
S ← $B[p_m]$; $s_0$ ← m
for i ← m - 1 downto 1 do
    if S & (1 <= (m - 1)) ≠ 0 then $s_0$ ← i endif
S ← (S ≤ 1) & $B[p_i]$
endfor
```

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 and enhanced version of SBNDM applying q-grams. The pseudocode is shown as Algorithm 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 $s_0$, 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 $P = abca$. 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 Algorithm 2.3.

As an example we give a compact C implementation of the main loop of SBNDM$_2$ in Algorithm 2.4. Because of clearness and compactness, this code differs slightly from Algorithm 2.2. The initial value of $i$ is $m$. It is assumed that $t_{n+1} \ldots t_{n+m}$ is a stopper, i.e. a copy of the pattern.
Algorithm 2.4 (SBNDM2.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 > m) return (nmatches);
        nmatches++;
    i++;
}

Here s₀ = 1 is applied. The code computes the number of matches nmatches.

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. Hyrro [12] has successfully tried this technique with BNDM. We adopted an approach by Kalsi et al. [13] to BNDMₙᵣ. We implemented five versions. SBNDM₂b reads a 2-gram as a 16-bit halfword. The value of B[tᵢ] & B[tᵢ₊₁] ≤ 1) is stored to a precomputed table g for each halfword. The second line of Algorithm 2.4 will then be

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

In SBNDM₄ₗ the corresponding value of 4-gram is computed as g[x₁] & g[x₂] ≤ 2) where x₁ and x₂ are the halfwords and g is the same table used in the 2-gram version. In SBNDM₆ₗ the value of 6-gram is computed as g[x₁] & g[x₂] ≤ 2) & (g[x₃] ≤ 4). SBNDM₈ₗ works in a corresponding manner. From SBNDM₆ₗ we made a modified version SBNDM₂₄ₗ, 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 a 4-gram.

All our SBNDMₙₗ 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 (tᵢ₊₁ ≪ 8) + tᵢ and on a big endian (tᵢ ≪ 8) + tᵢ₊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 GiB 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 tests 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 test was generated 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 BNDMnl [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 Leq, the ‘New’ algorithm of Lecroq [16], which is a q-gram variant of Horspool’s algorithm [10]. FSO is our tuned version of the Fast-Shif-Or algorithm [8] with 64-bit bitvectors.

4.1. Behavior with the 64-bit code

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.

SBNDM₂₄ₗ was the fastest among the tested algorithms for English patterns of 5 characters. For longer English patterns SBNDM₄ₗ 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, SBNDM₂₄ₗ 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, SBNDM₄ₗ for m = 10, and SBNDM₆ₗ m = 20 and 30, respectively. On the binary patterns, FSO was the best for m = 5 and 10, and SBNDM₆ₗ 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.

1 Ki = 2³⁰ and Gi = 2³⁰ are prefixes of the IEEE 1541 standard.
4.2. 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.

4.3. 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.

4.4. 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. SBNDM3 did not get similar gain as on x86 processors. However, the best version, SBNDM3 was faster on binary and DNA than the 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