Information Processing Letters 110 (2010) 148–152

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Improving practical exact string matching ☆

Branislav Ďurian [a], Jan Holub [b], Hannu Peltola [c], Jorma Tarhio [c,*]

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

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.

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 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 re-

☆ A preliminary version of this paper appeared in Proceedings of ALENEX 2009, Tenth Workshop on Algorithm Engineering and Experiments.

* Corresponding author.
E-mail addresses: branislav.durian@snt.sk (B. Ďurian), jan.holub@fit.cvut.cz (J. Holub), hpeltola@cs.hut.fi (H. Peltola), tarhio@cs.hut.fi (J. Tarhio).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2009.11.010

Algorithm 2.1 (BNDM)

```
for a ∈ Σ do B[a] ← 0 endfor
for j ← 1..m do
    B[p_j] ← B[p_j] | (1 ≪ (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+j}]
        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
```

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 $P = p_1 p_2 \dots p_m$ and a text $T = t_1 t_2 \dots t_n$ 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 $t_i t_{i+1} \dots t_{i+m-1} = p_1 p_2 \dots p_m$. 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$_q$

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. 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

Algorithm 2.2 (SBNDM$_q$)

```
for a ∈ Σ do B[a] ← 0 endfor
for j ← 1..m do
    B[p_j] ← B[p_j] | (1 ≪ (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 = i then
            report occurrence at j + 1
            i ← i + s_0
        endif
    endif
    i ← i + m − q + 1
endwhile
```

Algorithm 2.3 (Computing s_0)

```
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

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

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 $P = \texttt{abcab}$. 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 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 $t_{n+1} \dots t_{n+m}$ 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 > n) return (nmatches);
    nmatches++;
    i++;
  }
}
```

Here $s_0 = 1$ 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. Hyyrö [12] has successfully tried this technique with BNDM. We adopted an approach by Kalsi et al. [13] to SBNDM$_q$. We implemented five versions. SBNDM$_{2b}$ 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 Algorithm 2.4 will then be

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

In SBNDM$_{4b}$ 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 SBNDM$_{6b}$ the value of 6-gram is computed as $g[x_1] \,\&\, (g[x_2] \ll 2) \,\&\, (g[x_3] \ll 4)$. SBNDM$_{8b}$ works in a corresponding manner. From SBNDM$_{4b}$ we made a modified version SBNDM$_{2+2b}$, 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 a 4-gram.

All our SBNDM$_{qb}$ 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 GiB[1] 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 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 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 Lec$q$, 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.

SBNDM$_{2+2b}$ was the fastest among the tested algorithms for English patterns of 5 characters. For longer English patterns SBNDM$_{4b}$ 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, SBNDM$_{2+2b}$ 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 $m = 5$, SBNDM$_{4b}$ for $m = 10$, and SBNDM$_{6b}$ $m = 20$ and 30, respectively. On the binary patterns, FSO was the best for $m = 5$ and 10, and SBNDM$_{8b}$ 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 ($= 2^{10}$) and Gi ($= 2^{30}$) are prefixes of the IEEE 1541 standard.

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

Algorithm	Patterns											
	English				DNA				Binary			
	5	10	20	30	5	10	20	30	5	10	20	30
Shift-Or	668	667	668	668	667	668	668	668	670	669	669	669
FSO	188	188	188	188	195	188	187	188	425	194	189	188
BNDM	385	312	201	146	751	415	229	160	1340	741	389	269
BNDMnr	350	290	183	125	687	376	205	141	1270	683	335	230
$SBNDM_1$	367	301	188	128	714	389	211	146	1257	703	343	233
$SBNDM_2$	145	128	116	90	493	339	198	142	1239	694	339	229
$SBNDM_3$	209	109	81	60	289	180	133	109	1041	668	338	228
$SBNDM_4$	358	128	77	58	377	141	85	68	726	550	320	224
$SBNDM_5$	903	176	83	58	903	179	82	61	958	401	274	209
$SBNDM_6$	–	241	93	64	–	242	92	64	–	350	212	173
$SBNDM_8$	–	502	141	87	–	503	141	87	–	525	165	113
$SBNDM_{2b}$	125	130	117	88	449	296	175	125	1203	647	320	212
$SBNDM_{2+2b}$	105	108	100	76	400	245	124	80	933	532	298	201
$SBNDM_{4b}$	246	92	63	50	236	97	61	53	647	490	283	197
$SBNDM_{6b}$	–	166	77	55	–	145	60	43	–	262	175	141
$SBNDM_{8b}$	–	325	93	66	–	277	77	53	–	300	105	81
Lec_3	419	189	104	80	465	207	117	92	881	514	389	377
Lec_4	698	246	126	92	718	251	127	94	971	397	235	199
Lec_5	–	307	147	105	–	309	147	106	–	385	191	146
Lec_6	–	374	154	110	–	375	154	111	–	414	174	128
Lec_7	–	550	195	125	–	551	195	125	–	572	203	131
BM	330	226	150	121	921	670	568	511	1864	1381	989	871
QS	310	219	150	121	869	726	706	701	1614	1691	1740	1645

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 Lec*q* versions were slightly faster in the 32-bit mode.

4.3. Memory requirements

All the versions of $SBNDM_q$ 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 $SBNDM_{qb}$ requires additional 256 (or 512) KiB. The initialization of each $SBNDM_{qb}$ 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. $SBNDM_q$ did not get similar gain as on x86 processors. However, the best version, $SBNDM_3$ 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

[1] C. Allauzen, M. Crochemore, M. Raffinot, Factor oracle: A new structure for pattern matching, in: SOFSEM'99, Theory and Practice of Informatics, in: Lecture Notes in Comput. Sci., vol. 1725, Springer, Berlin, 1999, pp. 291–306.
[2] R.A. Baeza-Yates, G.H. Gonnet, A new approach to text searching, Comm. ACM 35 (10) (1992) 74–82.
[3] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Comm. ACM 20 (10) (1977) 762–772.
[4] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.
[5] B. Dömölki, An algorithm for syntactic analysis, Comput. Linguist. 3 (1964) 29–46, Hungarian Academy of Science, Budapest.
[6] S. Faro, T. Lecroq, Efficient variants of the backward-oracle-matching algorithm, in: Proc. PSC 2008, The 13th Prague Stringology Conference, 2008, pp. 146–160.
[7] K. Fredriksson, Shift-or string matching with super-alphabets, Inform. Process. Lett. 87 (4) (2003) 201–204.
[8] K. Fredriksson, Sz. Grabowski, Practical and optimal string matching, in: String Processing and Information Retrieval, 12th International Conference, SPIRE 2005, in: Lecture Notes in Comput. Sci., vol. 3772, Springer, Berlin, 2005, pp. 376–387.

[9] L. He, B. Fang, J. Sui, The wide window string matching algorithm, Theoret. Comput. Sci. 332 (1–3) (2005) 391–404.
[10] R.N. Horspool, Practical fast searching in strings, Softw. Pract. Exp. 10 (6) (1980) 501–506.
[11] A. Hume, D.M. Sunday, Fast string searching, Softw. Pract. Exp. 21 (11) (1991) 1221–1248.
[12] H. Hyyrö, Personal communication.
[13] P. Kalsi, H. Peltola, J. Tarhio, Exact string matching algorithms for biological sequences, in: 2nd International Conference on Bioinformatics Research and Development, BIRD 2008, in: Communications in Computer and Information Science, vol. 13, Springer, Berlin, 2008, pp. 417–426.
[14] J.Y. Kim, J. Shawe-Taylor, Fast string matching using an n-gram algorithm, Softw. Pract. Exp. 24 (1) (1994) 79–88.
[15] M.O. Külekci, A method to overcome computer word size limitation in bit-parallel pattern matching, in: Proceedings of the 19th International Symposium on Algorithm and Computation, ISAAC 2008, in: Lecture Notes in Comput. Sci., vol. 5369, Springer, 2008, pp. 496–506.
[16] T. Lecroq, Fast exact string matching algorithms, Inform. Process. Lett. 102 (6) (2007) 229–235.
[17] G. Navarro, NR-grep: A fast and flexible pattern-matching tool, Softw. Pract. Exp. 31 (13) (2001) 1265–1312.
[18] G. Navarro, M. Raffinot, Fast and flexible string matching by combining bit-parallelism and suffix automata, ACM J. Exp. Algorithmics (JEA) 5 (4) (2000).
[19] H. Peltola, J. Tarhio, Alternative algorithms for bit-parallel string matching, in: String Processing and Information Retrieval, 10th International Symposium, SPIRE 2003, in: Lecture Notes in Comput. Sci., vol. 2857, Springer, Berlin, 2003, pp. 80–94.
[20] D.M. Sunday, A very fast substring search algorithm, Comm. ACM 33 (8) (1990) 132–142.