ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221580191

Alternative Algorithms for Bit-Parallel String Matching

Conference Paper *in* Lecture Notes in Computer Science · October 2003

DOI: 10.1007/978-3-540-39984-1_7 · Source: DBLP

CITATIONS
85

READS
421

2 authors, including:

Jorma Tarhio
Aalto University

129 PUBLICATIONS **2,619** CITATIONS

SEE PROFILE

All content following this page was uploaded by Jorma Tarhio on 25 October 2014.

The user has requested enhancement of the downloaded file.

Alternative Algorithms for Bit-Parallel String Matching

Hannu Peltola and Jorma Tarhio

Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 5400, FIN-02015 HUT, Finland
{hpeltola, tarhio}@cs.hut.fi

Abstract. We consider bit-parallel algorithms of Boyer-Moore type for exact string matching. We introduce a two-way modification of the BNDM algorithm. If the text character aligned with the end of the pattern is a mismatch, we continue by examining text characters after the alignment. Besides this two-way variation, we present a simplified version of BNDM without prefix search and an algorithm scheme for long patterns. We also study a different bit-parallel algorithm, which keeps the history of examined characters in a bit-vector and where shifting is based on this bit-vector. We report experiments where we compared the new algorithms with existing ones. The simplified BNDM is the most promising of the new algorithms in practice.

1 Introduction

Searching for occurrences of string patterns is a common problem in many applications. Various good solutions have been presented during years. The most efficient solutions in practice are based on the Boyer-Moore algorithm [2]. We consider several Boyer-Moore modifications applying bit-parallelism for exact string matching.

Three of our algorithms are based on the Backward Nondeterministic DAWG Matching (BNDM) algorithm by Navarro and Raffinot [8, 9]. The BNDM algorithm itself has been developed from the Backward DAWG Matching (BDM) algorithm [3]. BNDM is a fascinating string matching algorithm. The nice feature of BNDM is that it simulates a nondeterministic automaton without explicitly constructing it.

The first of our algorithms is a two-way modification of BNDM. We call it Two-way Nondeterministic DAWG Matching or TNDM for short. If the text character aligned with the end of the pattern is a mismatch, BNDM scans backwards in the text if the conflicting character occurs elsewhere in the pattern. In such a situation TNDM will scan forward, i.e. it continues by examining text characters after the alignment.

In addition to TNDM we consider two other variations of BNDM. The first one has lighter shift computation than BNDM. This algorithm is called Simplified BNDM or SBNDM for short. The second one is a technique to handle long patterns with BNDM. This variation is called Long BNDM or LBNDM for short.

After a shift, most algorithms of Boyer-Moore type forget totally the examined text characters of the previous alignment. We present a new algorithm, which maintains a bit-vector telling those positions where an occurrence of the pattern cannot end in order to transfer information from an alignment to subsequent alignments. We base shifting on this bit-vector. The problem of computation of shift reduces to searching for the rightmost zero in a bit-vector. This algorithm is called Shift-Vector Matching or SVM for short. Although SVM is a kind of brute force approach, it is surprisingly efficient, because it fetches less text characters than other tested algorithms.

Navarro and Raffinot [9, p. 14] also use the method of searching for a certain bit in a bit-vector describing the state of the search. However, they consider only one alignment at a time and they initialize the bit-vector for each alignment. In SVM we initialize the bit-vector only once and so we are able to exchange information between alignments. SVM searches only for complete matches and does not recognize substrings of the pattern like BNDM.

We tested the algorithms with English, DNA, and binary texts on several machines. We measured numbers of fetched characters and execution times. No algorithm was the best in every individual test. However, the simplified BNDM is the most promising of the new algorithms in practice.

2 Two-way variant of BNDM

Let us consider TNDM, our first modification of BNDM, in detail. Let a *text* $T = t_1 \cdots t_n$ and a *pattern* $P = p_1 \cdots p_m$ be strings of an alphabet Σ. Let us consider the first comparison of an alignment of the pattern: t_i vs. p_m. There are three cases:

1. $t_i = p_m$;
2. $t_i \neq p_m$ and t_i occurs elsewhere in P, i.e. there exists $j \neq m$ such that $t_i = p_j$;
3. t_i does not occur in P.

The new algorithm works as BNDM in Cases 1 and 3, but the operation is different in Case 2, where the standard BNDM continues examining backwards until it finds a substring that does not occur in the pattern or it reaches the beginning of the pattern. TNDM will scan *forward* in Case 2. Our experiments indicate that this change of direction will decrease the number of examined characters. In a way this approach is related to Sunday's idea [10] for using the text position immediately to the right of an alignment for determining shift in the Boyer-Moore algorithm.

In Case 2, the next text characters that are fetched are not needed for checking a potential match in the BNDM algorithm, but they are only used for computing the shift. Because $t_i \neq p_m$ holds, we know that there will be a shift forward anyway before the next occurrence is found. The idea of TNDM is to examine text characters forward one by one until it finds the first k such that the string $t_i \cdots t_k$ does not appear in P or $t_i \cdots t_k$ forms a suffix of P. In the former case we can shift beyond the previous alignment of the pattern.

Checking whether the examined characters form a suffix of the pattern, is made by building the identical bit-vector as in BNDM, but in the reverse order. Note, that the bit-vector is built with AND operations which are commutative. So we can build it in any order—especially in the reverse order. Instead of shifting the bit-vector describing the state, we shift the bit-vectors of characters. Thus if we find a suffix, we continue to examine backwards starting from the text position $i - 1$. This is done by resuming the standard BNDM operation.

We use the following notations. The length of a computer word is denoted as w. A bit mask of s bits is represented as $b_s \cdots b_1$. The most significant bit is on the left. Exponentiation stands for bit repetition (e.g. $10^21 = 1001$). The C-like notations are used for bit operations: "`|`" bitwise (inclusive) OR, "`&`" bitwise AND, "`~`" one's complement, "`<<`" bitwise shift to the left with zero padding, and "`>>`" bitwise shift to the right with zero padding. For the shift operations, the first operand is unsigned and the second operand must be non-negative and less than w.

The pseudo-code of TNDM for $m \leq w$ is shown as Algorithm 1. It is straightforward to extend the algorithm for longer patterns in the same way as BNDM, see [9]. Because BNDM is a bit-parallel implementation of BDM, it is possible to make a two-way modification of BDM.

To be able to resume efficiently examining backwards, i.e. jumping in the middle of the main loop of BNDM, we preprocess the possible values of the variable *last* of BNDM for the suffixes of the pattern. With *last*, BNDM keeps track of the starting position of the next potential occurrence P. By updating the state vector in a clever way during the forward phase, we keep it ready for the backward phase.

In preprocessing the values of *last* are computed with the BNDM algorithm as if there were a full occurrence of the pattern in the text. Algorithm 2 shows the pseudo-code where the values of *last* are stored in the array *restore*. We demonstrate the execution of TNDM with an example in Table 1.

Our experiments indicate that TNDM examines less characters than BNDM on the average. There are two reasons for that. Let $t_i \cdots t_k$ be the string examined during the forward phase.

– When $t_i \cdots t_k$ is a suffix of P, we shift the pattern to that suffix. The suffix need not to be reexamined for a possible match ending at t_k. (If BNDM finds a prefix $t_h \cdots t_i$, that prefix may be reexamined for a possible match starting at t_h.)
– If $p_1 \neq p_m$ and $t_i = p_1$ hold, TNDM may make a shift one position longer than BNDM.

It is not difficult to find examples where TNDM examines more characters than BNDM. However, there is always a dual case where the situation is vice versa. Basically BNDM searches for a substring $t_h \cdots t_i$ and TNDM for

Algorithm 1 TNDM.

```
TNDM($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)
          /* Preprocessing */
for $c \in \Sigma$ do $B[c] \leftarrow 0^m$
for $j \in 1 \ldots m$ do $B[p_j] \leftarrow B[p_j] \mid 0^{j-1}10^{m-j}$
Init_shift($P$, $restore[\,]$)
          /* Searching */
$epos \leftarrow m$
while $epos \leq n$ do
    $i \leftarrow 0$; $last \leftarrow m$
    $D \leftarrow B[t_{epos}]$
    if $(D\&1) = 0$ then /* when $D \neq B[p_m]$, */
        do      /* forward scan for suffix of pattern */
            $i \leftarrow i + 1$
            $D \leftarrow D\&(B[t_{epos+i}] << i)$
        until $D \neq 0^m$ and $D\&10^i = 0^m$
        if $D = 0^m$ then /* already $last \leftarrow m$ */
            goto Over
        $epos \leftarrow epos + i$; $last \leftarrow restore[i]$
    do /* variation of BNDM */
        $i \leftarrow i + 1$
        if $D\&10^{m-1} \neq 0^m$ then
            if $i < m$ then $last \leftarrow m - i$
            else report an occurrence at $epos - m + 1$; goto Over
        $D \leftarrow D << 1$
        $D \leftarrow D\&B[t_{epos-i}]$
    until $D \neq 0^m$
    Over:
    $epos \leftarrow epos + last$
```

Algorithm 2 Initialization of the array *restore*.

```
Init_shift($P = p_1p_2 \cdots p_m, restore[\,]$)
$D \leftarrow 1^m$
$last \leftarrow m$
for $i \leftarrow m$ downto 1 do
    $D \leftarrow D\&B[p_i]$
    if $D\&10^{m-1} \neq 0^m$ then
        if $i > 0$ then $last \leftarrow i$
    $restore[m - i + 1] \leftarrow last$
    $D \leftarrow D << 1$
```

Table 1. Simulation of TNDM. $P = \texttt{ATCGA}$; $T = \texttt{GCATCATGATCGAATCAG}\cdots$; Bit-vectors B: $B[\texttt{A}] = 10001$, $B[\texttt{C}] = 00100$, $B[\texttt{G}] = 00010$, $B[\texttt{T}] = 01000$. The last fetched character has been underlined.

Text window	Line	D	i	*epos*	*last*	Explanation
GCAT$\underline{\texttt{C}}$ATGA···	8	00100	0	5	5	The lowest bit is 0; continue to line 9.
GCATC$\underline{\texttt{A}}$TGA···	12	00000	1	5	5	$D = 0$; leave the loop and proceed with lines 13, 14, 24, 25, and 5–.
···ATGA$\underline{\texttt{T}}$CGAy···	8	01000	0	10	5	The lowest bit is 0; continue to line 9.
···ATGAT$\underline{\texttt{C}}$GAA···	12	01000	1	10	5	$D \neq 0$ **and** $D\&10 = 0$; continue to line 9.
···ATGATC$\underline{\texttt{G}}$AA···	12	01000	2	10	5	$D \neq 0$ **and** $D\&100 = 0$; continue to line 9.
···ATGATCG$\underline{\texttt{A}}$A···	12	01000	3	10	5	$D \neq 0$ **and** $D\&1000 = 1$; leave the loop and continue to lines 13, and 15–.
···ATCG$\underline{\texttt{A}}$A···	16	01000	3	13	4	A suffix is found; *epos* and *last* are updated; the scanning direction changes.
···ATCG$\underline{\texttt{A}}$A···	18	01000	4	13	4	$D\&10000 = 0$; not interesting, proceed with lines 21, 22, and 23.
···$\underline{\texttt{A}}$TCGAA···	23	10000	4	13	4	$D \neq 0$; proceed with lines 16, 17, and 18.
···$\underline{\texttt{A}}$TCGAA···	18	10000	5	13	4	$D\&10000 = 1$; something interesting! A prefix or a match?
···AT$\underline{\texttt{G}}$ATCGAA···	20	10000	5	13	4	$i = m(= 5)$; the else branch reports an occurence at 9; continue to lines 24, 25, and 5–.
···AATC$\underline{\texttt{A}}$G···	8	10001	0	17	5	The lowest bit is 1; continue from line 16 with BNDM.
···AATC$\underline{\texttt{A}}$G···	18	10001	1	17	5	$D\&10000 \neq 0$; a prefix or a match?
···AATC$\underline{\texttt{A}}$G···	19	10001	1	17	4	$i < m(= 5)$; it was a prefix, update *last*.
···AAT$\underline{\texttt{C}}$AG···	23	00000	1	17	4	$D = 0$; continue to lines 24 and 25.
···$\underline{\texttt{C}}$AG··	25	00000	1	21	4	$D = 0$; continue to line 5.

a substring $t_i \cdots t_k$ which do not appear in P. Depending on the proportion $(k-i)/(i-h)$ either algorithm has gain.

Further enhancements. If the last character examined does not occur in P while scanning forward, we are able to shift pattern entirely over it. This can be done by adding the following line to TNDM:

```
13.5            if $B[t_{epos+i}] = 0$ then $last \leftarrow i + m$
```

This test is computationally light, because after a forward scan only t_k of $t_i \cdots t_k$ can be missing from the pattern. The test clearly reduces the number of fetched characters. However, the test is beneficial only for alphabets large enough. In Section 5 we call TNDM with this test TNDMa.

In TNDM we scan forward when t_i is not p_m and t_i occurs elsewhere in P. This can be generalized as follows. If the backward phase has encountered $v = t_h \cdots t_i$ such that v is not a suffix of P but v appears elsewhere in P, we will scan forward starting from t_{i+1}. We expect this modification would improve TNDM a bit in the case of small alphabets.

Implementation remarks. Note that on lines 9–12 the algorithm may address at most $m-1$ characters past t_n, the last character of text. This can be prevented by adding the following test on line 8:

```
8.      if $(D\&1) = 0$ and $epos + m - 1 \leq n$ then
```

The other possibility is to ignore spurious suffix and change line 13 in the following way:

```
13.           if $D = 0^m$ or $epos + i > n$ then
```

and allow references to $t_{n+1}, \ldots, t_{n+m-1}$. The third solution is to store to t_{n+1} a stopper character, e.g. null, which do not appear in any pattern.

If the interesting bits do not use the whole word i.e. $m < w$, then one has to be careful with tests like '$D \neq 0^k$'. As the result of a shift some set bits may move beside the interesting area of bit-vector, and tests cannot be simplified to form '$D \neq 0$'. If the interesting bits are located on that edge in the shifting direction, the uninteresting bits fall off during shift. Navarro and Raffinot use this trick successfully in their implementation of BNDM. In the pseudo-code of TNDM, all tests with 0^m can be simplified without extra masking.

Complexity. We consider only patterns than are at most w characters long. The preprocessing time is $O(m+|\Sigma|)$. The worst case complexity of TNDM is clearly $O(nm)$. The average case complexity of BNDM (and BDM) is $O(n \log_{|\Sigma|} m/m)$. It is not difficult to see that the same is true for TNDM.

3 Other variations of BNDM

3.1 SBNDM

When BNDM finds $t_h \cdots t_i$ which is a match or do not appear in P, there are two options for shifting. Let j be the smallest index such that $h < j \leq i$ holds and $t_j \cdots t_i$ is a prefix of P. Then the next alignment starts at t_j. If there is no such prefix, then the next alignment starts at $i+1$.

In SBNDM we shift as in BNDM in the case of a match. But if $t_h \cdots t_i$ do not appear in P, we skip the examining of prefixes and set $h+1$ to be the start position of the next alignment. Naturally this reduces the average length of shift, but on the other hand the innermost loop of the algorithm becomes simpler. Our experiments show that SBNDM is most often faster than BNDM.

The pseudo-code of SBNDM is shown as Algorithm 3. The table B is initialized as in BNDM and TNDM. In the case of a complete match, the shift is s_0, which is easy to precompute as *restore*[1], see Alg. 2. In other words, s_0 equals to $m - x$ where x is the length of the longest prefix of P, which is also a suffix of P.

Algorithm 3 Simplified BNDM.

```
SBNDM($P = p_1 p_2 \cdots p_m, T = t_1 t_2 \cdots t_n$)
1.  initialize $B$ and $s_0$
2.  $pos \leftarrow 0$
3.  while $pos \le n - m$ do
4.      $D \leftarrow 1^m; j \leftarrow m$
5.      do
6.          $D \leftarrow (D << 1) \& B[t_{pos+j}]$
7.          $j \leftarrow j - 1$
8.      until $D = 0^m$ or $j = 0$
9.      if $D \neq 0^m$ then
10.         report an occurrence at $pos$
11.         $pos \leftarrow pos + s_0$
12.     else $pos \leftarrow pos + j + 1$
```

Note that it is possible to leave out the test of j on line 8, because D becomes always zero after m bitwise shifts, but this version will need to examine one extra character after each match (immediately to the left of a match).

3.2 LBNDM

Navarro and Raffinot [9, p. 12] introduced also a method of searching for patterns longer than w. They partitioned the pattern in consecutive subpatterns. All the subpatterns have w characters except possibly the rightmost one which gets the remaining characters. The leftmost subpattern is searched with the standard BNDM algorithm. Only when the match of the leftmost subpattern is found, the rest of an alignment is examined. The maximum shift is w.

We introduce another approach called LBNDM for long patterns. LBNDM is able to make shifts longer than w. The pattern is partitioned in $\lfloor \frac{m}{k} \rfloor$ consecutive parts, each consisting of $k = \lfloor \frac{m-1}{w} \rfloor + 1$ characters. The $m - k\lfloor \frac{m}{k} \rfloor$ remaining character positions are left to either end of the pattern (or to both ends). This division implies k subsequences of the pattern such that the ith sequence takes the ith character of each part. The idea is to search first the superimposed pattern of these sequences so that only every kth character is examined. This filtration phase is done with the standard BNDM algorithm. Each occurrence of the superimposed pattern is a potential match of the original pattern and thus must be verified.

Note that the shifts of the LBNDM are multiples of k. To get a real advantage of shifts longer than in the approach of Navarro and Raffinot, the pattern length should be at least about two times w. On the other hand, this implies $k \geq 3$, which on DNA data turns out to be quite high. In the case of a small alphabet a feasible solution could be to use q-grams instead of single characters, see [7].

4 Shift-Vector Matching

One problem of Boyer-Moore type algorithms is that they do not remember the tried text positions of previous alignments. When the shift is shorter than the pattern length m, some alignments of the pattern may be tried in vain. In the following we will introduce an algorithm with partial memory. The key idea is simple: We maintain a bit-vector, called a *shift-vector*, which tells those positions where an occurrence of the pattern can or cannot end. This approach makes it possible to base shifting on this shift-vector and to manage without any shift table.

While moving the pattern forward and shifting the shift-vector, the old knowledge of already handled positions goes off from the shift-vector. Then the bit corresponding to the end of the pattern must be the highest or the lowest bit. We chose the lowest one, because then masking on some processors is slightly faster. (Often the fastest way to load the specific bit-mask 1 to a register is loading the constant 1 with some instruction, which is not referring to the memory.) This decision implies that the shifting direction is to the right. The new bits entering to a bit-vector during a bitwise shift are zeros, and therefore it is natural to use the convention where zero denotes a text position not yet rejected.

In preprocessing we create bit-vectors representing the characters of an alphabet. These bit-vectors have the zero bit on every position where that character occurs in the pattern, and one elsewhere. So the characters that do not appear in the pattern have the bit-vector $0^{w-m}1^m$. Note that the essential parts of these vectors are complements of those used in TNDM.

We keep track of possible end positions of the pattern in the shift-vector S. It is simply updated by taking OR with the bit-vector corresponding to text character aligned with last character of the pattern. If the lowest bit in S is one, a match cannot end here and we can shift the pattern. The length of the shift is simply got by searching the lowest zero bit in S which is above the lowest position. In addition to shifting the pattern, we also shift bits in S with the same number of positions to the right.

If the lowest bit in S is zero, i.e. p_m has been found, we have to continue checking for the match. Our first implementation had a classical pairwise comparison of pattern and text characters. In addition, S was updated with all characters that were fetched during verifying of alignments. Of course the scope of text characters is relative to the end of pattern. To correctly update S with bit-vectors of text characters, that are aligned with pattern, their values have to be shifted to the right depending how far the are from the end of pattern: $S \leftarrow S \mid (C[t_{epos-j}] >> j)$. Because the lowest bit remains zero as long as a mismatch has not been found, we could remove the pairwise comparison. Text characters that are on the left-hand side of alignment give less information for shifting than those, which are close to the right end of alignment. That is why we chose to check, if there is a match, in the reverse order, i.e. from right to left.

Algorithm 4 SVM.

```
SVM($P = p_1 p_2 \cdots p_m, T = t_1 t_2 \cdots t_n$)
          /* Preprocessing */
for $c \in \Sigma$ do $C[c] \leftarrow 1^m$
for $j \in 1 \ldots m$ do $C[p_j] \leftarrow C[p_j] \& 1^{j-1}01^{m-j}$
          /* Searching */
$epos \leftarrow m$; $S \leftarrow 0$
while $epos \leq n$ do
    $S \leftarrow S \mid C[t_{epos}]$
    $j \leftarrow 1$
    while $(S \& 1) = 0$ do
        if $j \geq m$ then
            report an occurrence at $epos + 1 - m$
            goto Over
        $S \leftarrow S \mid (C[t_{epos-j}] >> j)$
        $j \leftarrow j + 1$
    Over:
    $last \leftarrow$ BSF(˜$(S >> 1)$)+1
    $S \leftarrow S >> last$
    $epos \leftarrow epos + last$
```

The pseudo-code of SVM for $m \leq w$ is shown as Algorithm 3. To our knowledge, this approach of shifting has not been studied before. The function BSF scans the bits in the operand bit-vector starting from lowest bit and searches for the first set bit. The function returns the number of zero bits before first set bit. In the end of this section we discuss various implementation alternatives of BSF.

We demonstrate the execution of SVM with an example in Table 2. After a long shift, the shift-vector S becomes zero or almost zero. Then the subsequent shift is more likely shorter. Fortunately after a short shift, there will normally be several ones in S, and so the subsequent shift will likely be longer again. E.g.

after reading G on the second row of the example, S becomes full of ones enabling a full shift of 5 positions.

In SVM there is an obvious trade-off between the number of fetched characters and searching for a set bit in the shift-vector. The run times depend on the relative speed of these functions. It is straightforward to extend this algorithm for longer patterns.

Complexity. Let us assume that $m \leq w$ holds and BSF is $O(1)$. Then the preprocessing time is $O(m+|\Sigma|)$ and the worst case complexity is clearly $O(nm)$. SVM is sublinear on the average, because at the same alignment it fetches the same text characters as the Boyer-Moore-Horspool algorithm [5] (assuming the right-to-left examining order) and can never make a shorter shift than that algorithm, which is known to be sublinear on the average. If a constant time BSF is not available, there will be an extra work of $\log m$ or $\log w$ for each alignment.

Table 2. Simulation of SVM. P = ATCGA; T = GCAGCTATCGAG$\cdots$; Bit-vectors C: C[A] = 01110, C[C] = 11011, C[G] = 11101, C[T] = 10111. The last fetched character has been underlined. The snapshots correspond to lines 8 and 14.

Text	S	j	*epos*	*last*
GCAG$\underline{\texttt{C}}$TGATCGAG$\cdots$	11011	1	5	2
GCAGCT$\underline{\texttt{G}}$ATCGAG$\cdots$	11111	1	7	5
GCAGCTGATCG$\underline{\texttt{A}}$G$\cdots$	01110	1	12	(5)
GCAGCTGATC$\underline{\texttt{G}}$AG$\cdots$	01110	2	12	(5)
GCAGCTGAT$\underline{\texttt{C}}$GAG$\cdots$	01110	3	12	(5)
GCAGCTGA$\underline{\texttt{T}}$CGAG$\cdots$	01110	4	12	(5)
GCAGCTG$\underline{\texttt{A}}$TCGAG$\cdots$	01110	5	12	5

Search for the lowest zero bit

From the previously examined characters we usually know some positions where the pattern cannot end. All these positions with reference to the end of the pattern have the corresponding bit set in the shift-vector S of SVM. The lowest bit represents the current position. To get the length of the next shift of the pattern, one has to find the rightmost zero bit in S. Alternatively one can complement the bits and search for lowest set bit.

There are several possibilities for searching of the rightmost set bit. Below we consider five alternatives: BSF-0, ..., BSF-4. If we first shift the contents of the word one position to the right and if we are using unsigned variables in C, we get zero padding and there will always exist at least one zero bit.

BSF-0. Many computer architectures have instructions for scanning bits; for example Intel's x86 has instructions for scanning both forward (*Bit Scan Forward*) and backward (*Bit Scan Reverse*). A suitable implementation can be found in Arndt's collection [1] of x86 inline asm versions of various functions as function `asm_bsf`. If this kind of instruction is available, using it gives the fastest performance.

BSF-1. The simplest way to seek for the lowest zero bit goes by shifting the word bit position by bit position to the right and testing the lowest bit. If lowest zero bit can be found with few iterations—e.g. when m is small—the performance is acceptable. Navarro and Raffinot [9] have used this technique in their implementation of BM_BNDM and TurboBNDM. The relative performance decreases while the average length of shift increases.

BSF-2. Search for the lowest set bit becomes easier, if we assume that at most one bit is set. This can be achieved with an expression $x \,\&\, -x$.

If at most one bit is set, it is possible to apply bit masking: we divide different sized groups in the bit-vector to an upper and lower half. If some even bit is set, then we can increase the bit number by one. If some even bit-pair is set, then we can increase the bit number by two. If some upper half of byte is set, then we can increase the bit number by four; etc. Finding the set bit this way requires $\log w$ tests with different bit masks. This idea is presented in the function `lowest_bit_idx` of Arndt [1].

The masking method described above requires large (i.e. w bits wide) bit masks. If they are built from smaller pieces, their construction takes considerably work. One can expect large variation in performance depending on compiler and computer architecture.

BSF-3. This approach is similar to the previous one. Now we clear all other but the lowest set bit. If a number after shifting l positions to the right is not zero, it is obvious, that the only set bit is higher than l bits. The search goes most efficiently by halving: first $w/2$ bits, then $w/4$ bits, etc. The shifting could be made also to the left, but this way the optimizer of the compiler can produce more efficient code by reusing the results of the shifts. Examining the last byte goes faster and easier with the table lookup from precomputed constant array. Together $\log \frac{w}{8}$ shifting tests are needed. Actually the same holds also for m $\lceil \log \frac{m}{8} \rceil$ because one can tailor the routine for different pattern lengths. Relative performance improves clearly when patterns get longer.

BSF-4. We can also utilize the fact that at least one bit is set. The basic idea is that when we shift to the left and the result is zero, we can conclude that the lowest set bit was in the part that fell off. Because we try to isolate the lowest bit to smaller and smaller area, for the next step we have to shift the bit-vector to the right every time the result after shifting is zero. Examining the last byte is made with table lookup from precomputed constant array.

Typically SVM moves a couple of shorter shifts and then longer one, usually the maximum m. In our experimental tests we used the version BSF-0 utilizing the asm function.

Implementation remark. When $m = w$, the value of *last* may become w, which is too large for the shift on line 15. Then the shifting must be made in two parts. This can be made efficiently with the following changes:

```
13.5    $S \leftarrow S >> 1$
14.     $\mathit{last} \leftarrow \mathrm{BSF}(\tilde{}S)$
15.     $S \leftarrow S >> \mathit{last}$
16.     $\mathit{epos} \leftarrow \mathit{epos} + \mathit{last} + 1$
```

5 Experimental results

We tested BNDM, TNDM, SBNDM and SVM together with three older algorithms TBM, GD2, and BMH4, which we know to be fast. The code of BNDM is a courtesy from G. Navarro. TBM (`uf.fwd.md2`) and GD2 (`uf.rev.gd2`) are from Hume and Sunday [6]. TBM is recommended for general purpose use [6]. The performance of GD2 [6, p. 1244] on DNA data has drawn attention quite rarely. BMH4 is a 4-gram version [11] of the Boyer-Moore-Horspool algorithm [5] tuned for DNA data. BMH4 was so poor with the English text that we do not show those figures.

BNDM and SBNDM have a skip loop that could be classified as `uf1` [6]: a while loop is skipping with m positions for characters not present in the pattern.

Our test framework is a modification of the one made by Hume and Sunday [6]. The main experiments were carried out in a Linux workstation with 256 MB (PC133A SDRAM) memory, AMD Athlon Thunderbird 1000 MHz processor, 64 KB L1 cache for both data and instructions, and 256 KB L2 cache on die. The algorithms were compiled with the GNU C compiler version 2.95.4 with optimization flag `-O3`. All the bit-parallel algorithms used 32 bit bit-vectors.

We ran tests with three types of texts of 1 MB: English, DNA, and binary. The English text is the beginning of the KJV bible. The DNA text is from Hume and Sunday [6]. The binary data was generated randomly. For DNA and binary we used 200 patterns of four lengths drawn from same data source as the corresponding text. So every pattern do not necessary occur in the text. For English we used 500 words of various lengths as patterns.

Table 3 shows the proportions of fetched characters. SVM is a clear winner. TNDM and TNDMa also inspect less characters than BNDM. TBM fetches much more characters than others in the case of small alphabets. Note that GD2 and TBM contain a skip loop (`uf3`), which surely worsens their results with the binary alphabet.

Table 4 shows the search times in seconds. The figures are averages of 15 runs. While repeating tests, the variation in search times was less than 2%. BMH4 is the fastest for binary and DNA data. On the English text TBM is the best. TNDM and SVM are slightly slower than BNDM. SBNDM is on the average

Table 3. Proportions of fetched characters.

Σ	m	BNDM	GD2	TBM	BMH4	TNDM	TNDMa	SBNDM	SVM
Binary	5	.868	.983	1.55	.139	.763	.763	1.05	.750
	10	.516	.732	1.63	.675	.492	.492	.647	.445
	20	.292	.520	1.55	.421	.286	.286	.343	.250
	30	.214	.453	1.55	.357	.211	.211	.243	.178
DNA	5	.445	.481	.614	.887	.425	.413	.517	.376
	10	.262	.332	.488	.428	.258	.256	.296	.217
	20	.153	.253	.473	.214	.153	.153	.167	.126
	30	.111	.212	.461	.145	.111	.111	.119	.091
English	2–16	.205	.202	.210	–	.205	.199	.213	.190

Table 4. Run times of the algorithms.

Σ	m	BNDM	GD2	TBM	BMH4	TNDM	SBNDM	SVM
Binary	5	3.23	2.80	3.07	1.83	3.31	2.89	3.24
	10	2.04	2.23	3.40	1.18	2.17	1.76	2.19
	20	1.30	1.79	3.26	0.99	1.49	1.10	1.56
	30	0.96	1.68	3.22	0.91	1.16	0.91	1.25
DNA	5	1.81	1.69	1.65	1.20	2.12	1.67	2.10
	10	1.28	1.39	1.45	0.85	1.55	1.12	1.53
	20	0.91	1.18	1.41	0.82	1.06	0.86	1.04
	30	0.78	1.08	1.40	0.71	0.87	0.75	0.89
English	2–16	2.69	2.45	2.39	–	2.97	2.45	3.39

10% faster than BNDM. TNDMa (times not shown) was slightly slower than TNDM.

It is interesting that the performance of GD2 does not seem to improve as fast as bit-parallel algorithms while patterns get longer. On GD2 the preprocessing times were 0.0013–0.0038 seconds; on all others it was less than 0.0007 seconds.

We also tested the bit-parallel algorithms with 64 bits in our workstation with AMD Athlon CPU. They are about 50% slower mainly because the machine has a 32 bit ALU and the bit operations using 64 bits need two registers.

The performance of various BSF versions depends a lot of the compiler, the computer architecture, and the size of bit-vectors. Besides AMD Athlon we tested BSF on the following configurations: Sun Enterprise 450 (4 UltraSPARC-II 400MHz processors, 2048 MB main memory with 4 MB Ecache, Solaris 8) with Sun WorkShop 6 update 2 C 5.3 and gcc 3.2.1 compilers, and Digital Personal Workstation 433au (Alpha 21164A-2 (EV56 433 MHz) processor, 256 MB main memory; OSF1 V5.1 [Tru64 UNIX]) with Compaq C V6.5-011 and gcc 3.3 compilers.

Table 5 shows the relative performance of SVM with various BSF versions on the English text, where BSF-4 is used as a reference version (so its relative performance is 1). Smaller values denote faster performance.

Table 5. Relative performance of various BSF versions

CPU	AMD Athlon		UltraSPARC-II				Alpha 21164A-2	
Compiler/ w	gcc/32	gcc/64	Sun/32	gcc/32	Sun/64	gcc/64	Compaq(64)	gcc(64)
BSF-0	0.888	–	–	–	–	–	–	–
BSF-1	1.001	0.817	1.346	1.191	1.146	0.972	0.730	1.036
BSF-2	1.044	1.157	1.420	1.126	1.211	1.348	1.011	1.217
BSF-3	0.999	0.999	0.992	0.999	1.123	1.109	1.005	1.001

With longer patterns performance of BSF-1 got worse. Although Ultra-Sparc-II has 64-bit instructions, use of bit-vectors of 64 bits showed to be more than 60% slower than with 32 bits. The Alpha Compaq C-compiler produced code that worked rather slowly with BSF-2, BSF-3, and BSF-4.

We compared LBNDM with the approach by Navarro and Raffinot with long patterns on computers of different kind. There was a large variation in their relative performance. LBNDM proved to be clearly faster in the English text for $m = 70, \ldots, 180$.

The basic test set was run on several other computer architectures. The results were diversed. We review some deviations. On a 500 MHz Celeron, SBNDM was 15% faster than the second best algorithm in the English test. Other Intel processors gave similar results. On 400 MHz PowerPC G4, TNDM was faster than BNDM in the DNA tests. The compiler may affect the performance of a certain algorithm: On the Alpha workstation both the compilers genarate equally fast code for BNDM, but native compiler generates 30% slower code for TNBM than gcc in the case of the English text.

6 Concluding remarks

We introduced four bit-parallel algorithms for exact string matching: TNDM, SBNDM, LBNDM, and SVM. The algorithms are relatively compact. The algorithms can be extended to sets of strings (multiple pattern matching) and classes of characters in a similar way as BNDM [9]. It might be possible to combine SVM with Vishkin's sampling method [12]. At the monent we are working on a combination method of SBNDM and TNDM.

SBNDM showed to be the best of the new algorithms. As it often happens, algorithms with tight loops are also efficient. SBNDM is a good candidate for implementing grep especially in Intel-based machines.

References

1. J. Arndt: Jörgs useful and ugly BIT WIZARDRY page. *URL:* http://www.jjj.de/bitwizardry/bitwizardrypage.html.
2. R.S. Boyer and J.S. Moore: A Fast String Searching Algorithm. *Communications of the ACM*, **20**(10):762–772, 1977.
3. M. Crochemore and W. Rytter: *Text algorithms.* Oxford University Press, 1994.
4. Z. Galil: On Improving the Worst Case Running Time of the Boyer–Moore String Matching Algorithm. *Communications of the ACM*, **22**(9):505–508, 1979.
5. R.N. Horspool: Practical Fast Searching in Strings. *Software — Practice and Experience*, **10**(6):501–506, 1980.
6. A. Hume and D. Sunday: Fast String Searching. *Software — Practice and Experience*, **21**(11):1221–1248, 1991.
7. J. Kytöjoki, L. Salmela, and J. Tarhio: Tuning string matching for huge pattern sets. In *Proc CPM '03, Lecture Notes in Computer Science* **2676**:211–224, 2003.
8. G. Navarro and M. Raffinot: A Bit-parallel Approach to Suffix Automata: Fast Extended String Matching. In *Proc CPM '98, Lecture Notes in Computer Science* **1448**:14–33, 1998.
9. G. Navarro and M. Raffinot: Fast and Flexible String Matching by Combining Bit-parallelism and Suffix automata. *ACM Journal of Experimental Algorithms*, **5**(4):1–36, 2000.
10. D.M. Sunday: A Very Fast Substring Search Algorithm. *Communications of the ACM*, **33**(8):132–142, 1990.
11. J. Tarhio and H. Peltola: String Matching in the DNA Alphabet. *Software — Practice and Experience*, **27**(7):851–861, 1997.
12. U. Vishkin: Deterministic Sampling — a New Technique for Fast Pattern Matching. *SIAM Journal of Computing*, **20**(1):22–40, 1991.

View publication stats