Improved Two-Way Bit-parallel Search*

Branislav Ďurian[1], Tamanna Chhabra[2], Sukhpal Singh Ghuman[2], Tommi Hirvola[2], Hannu Peltola[2], and Jorma Tarhio[2]

[1] S&T Slovakia s.r.o., Priemyselná 2, SK-010 01 Žilina, Slovakia
`Branislav.Durian@snt.sk`
[2] Department of Computer Science and Engineering, Aalto University
P.O.B. 15400, FI-00076 Aalto, Finland
{`Tamanna.Chhabra, Suhkpal.Ghuman, Tommi.Hirvola`}`@aalto.fi`,
`hpeltola@cs.hut.fi`, `Jorma.Tarhio@aalto.fi`

Abstract. New bit-parallel algorithms for exact and approximate string matching are introduced. TSO is a two-way Shift-Or algorithm, TSA is a two-way Shift-And algorithm, and TSAdd is a two-way Shift-Add algorithm. Tuned Shift-Add is a minimalist improvement to the original Shift-Add algorithm. TSO and TSA are for exact string matching, while TSAdd and tuned Shift-Add are for approximate string matching with k mismatches. TSO and TSA are shown to be linear in the worst case and sublinear in the average case. Practical experiments show that the new algorithms are competitive with earlier algorithms.

1 Introduction

String matching can be classified broadly as exact string matching and approximate string matching. In this paper, we consider both types. Let $T = t_1t_2 \cdots t_n$ and $P = p_1p_2 \cdots p_m$ be text and pattern respectively, over a finite alphabet Σ of size σ. The task of exact string matching is to find all occurrences of the pattern P in the text T, i.e. all positions i such that $t_it_{i+1} \cdots t_{i+m-1} = p_1p_2 \cdots p_m$. Approximate string matching [14] has several variations. In this paper, we consider only the k mismatches variation, where the task is to find all the occurrences of P with at most k mismatches, where $0 \leq k < m$ holds.

We will present new sublinear variations of the widely known Shift-Or, Shift-And, and Shift-Add algorithms [3,19] which apply bit-parallelism. The key idea of the most of these algorithms is a two-way loop of j where text characters t_{i-j} and t_{i+j} are handled together. Our algorithms are linear in the worst case. Practical experiments show that the new algorithms with q-grams, loop unrolling, or with a greedy skip loop are competitive with earlier algorithms of same type.

All our algorithms utilize bit manipulation heavily. We use the following notations of the C programming language: '&', '|', '<<', and '>>'. These represent bitwise operations AND, OR, left shift, and right shift, respectively. Parenthesis and extra space has been used to clarify the correct evaluation order in pseudocodes. Let w be the register width (or word size informally speaking) of a processor, typically 32 or 64.

2 Previous algorithms

This section describes the previous solutions for exact and approximate string matching. First, we illustrate previous algorithms for exact matching which include Shift-Or and its variants like BNDM (Backward Nondeterministic DAWG Matching),

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

Branislav Ďurian, Tamanna Chhabra, Sukhpal Singh Ghuman, Tommi Hirvola, Hannu Peltola, Jorma Tarhio: Improved Two-Way Bit-parallel Search, pp. 71–83.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

TNDM (Two-way Nondeterministic DAWG Matching), LNDM (Linear Nondeterministic DAWG Matching), FSO (Fast Shift-Or) and FAOSO (Fast Average Optimal Shift-Or. Then the algorithms for approximate string matching are presented which cover Shift-Add and AOSA (Average Optimal Shift-Add).

2.1 Shift-Or and its variations

The Shift-Or algorithm [3] was the first string matching algorithm applying bit-parallelism. Processing of the algorithm can be interpreted as simulation of an automaton. The update operations to all states are identical. Operands in the algorithm are bit-vectors and the essential bit-vector containing the state of the automaton is called the state vector. The state vector is updated with the bit-shift and OR operations. FSO (Fast Shift-Or) [7] is a fast variation of the Shift-Or algorithm, and FAOSO (Fast Average Optimal Shift-Or) [7] is a sublinear variation of that algorithm.

BNDM [15] (Backward Nondeterministic DAWG Matching) is the bit-parallel simulation of an earlier algorithm called BDM (Backward DAWG Matching). BDM scans the alignment window from right to left and skips characters using a suffix automaton, which is made deterministic during preprocessing. BNDM, instead, simulates the nondeterministic automaton using bit-parallelism. BNDM applies the Shift-And method [19], which utilizes the bit-shift and AND operations.

TNDM (Two-way Nondeterministic DAWG Matching) [17] is a variation of BNDM applying two-way scanning. Our new algorithms are related to the Wide-Window algorithm [11] and its bit-parallel variations [4,11,10]. The LNDM (Linear Nondeterministic DAWG Matching) algorithm [10] is a two-way Shift-And algorithm with sequential symmetric scanning. The pseudocode of the LNDM is given as Alg. 1. The precomputed occurrence vector table B associates each character of the alphabet with a bit mask expressing occurrences of that character in the pattern P. We use table B for this purpose in all algorithms presented in this paper. In LNDM, the alignment window is shifted with fixed steps of m. Starting from the mth character of window the text characters are examined moving leftwards. The bitvector L becomes zero, when a mismatch is detected or (m shifts has been made while) m characters have

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

Require: $m \le w$
/* Preprocessing */
for all $c \in \Sigma$ **do** $B[c] \leftarrow 0$
for $i \leftarrow 1$ **to** m **do**
    $B[p_i] \leftarrow B[p_i] \mid 1 << (i-1)$ /* $\mid 0^{m-i}\,1\,0^{i-1}$ */
/* Searching */
for $i \leftarrow m$ **step** m **while** $i \le n$ **do**
    $l \leftarrow 0;\ r \leftarrow 0;\ L \leftarrow\ (\sim 0) >> (w-m);\ R \leftarrow 0$ /* $L \leftarrow 1^m;\ R \leftarrow 0^m$ */
    while $L \neq 0$ **do**
        $L \leftarrow L \,\&\, B[t_{i-l}]$
        $l \leftarrow l+1$
        $(LR) \leftarrow (LR) >> 1$
    $R \leftarrow R >> (m-l)$
    while $R \neq 0$ **do**
        $r \leftarrow r+1$
        if $R \,\&\, \big(1 << (m-1)\big) \neq 0$ **then** report occurrence
        $R \leftarrow (R << 1) \,\&\, B[t_{i+r}]$

been examined. The notation (LR) means the bitvector which is concatenated from two m bits long bitvectors L and R[1]. Next examining continues rightwards from the $m+1$ character of window. Simultaneously it is easy to notice the matches. In our two-way algorithms, these two scans are combined (into one scan). The characteristic feature in two-way algorithms is that the first characters bring plenty information to the state vector, but the last ones quite little.

2.2 Algorithms for the k-mismatches problem

Shift-Add [3, Fig. 8] is a bit-parallel algorithm for the k-mismatches problem. A vector of m states is used to represent the state of the search. A *field* of L bits is used for presenting each of the m states. The minimum value of L is $\lceil \log_2(k+1) \rceil + 1$. In the original Shift-Add the state i denotes the state of the search between the positions $1, \ldots, i$ of the pattern and positions $j-i+1, \ldots, j$ of the text, where j is the current position in the text.

A slightly more efficient variation of Shift-Add is (in the average case only) AOSA (Average Optimal Shift-Add) [7].

Galil and Giancarlo [9] presented a method for solving the k mismatches string matching problem in $\mathcal{O}(nk)$ time with constant time longest common extension (LCE) queries between P and T. Abrahamson [1] improved this for the case $\sqrt{(m \log m)} < k$ by giving an $\mathcal{O}(n\sqrt{m \log m})$ time algorithm based on convolutions. The asymptotically fastest algorithm known to date is given by Amir et al. [2], which achieves the worst-case time complexity of $\mathcal{O}(n\sqrt{k \log k})$. These algorithms are interesting in a theoretical sense, but in practice they perform worse than the trivial algorithm for reasonable values of m and k due to the heavy LCE and convolution operations. Hence we have the need for developing fast practical algorithms for string matching with k mismatches.

3 TSO and TSA

3.1 TSO

At first we introduce a new Two-way Shift-Or algorithm, TSO for short. The pseudocode of TSO is given as Alg. 2. TSO uses the same occurrence vectors B for characters as the original Shift-Or. The outer loop traverses the text with a fixed step of m characters. At each step i, an alignment window $t_{i-m+1}, \ldots, t_{i+m-1}$ is inspected. The positions $t_i, \ldots, t_{i+m-1}$ correspond to the end positions of possible matches and at the same time, to the positions of the state vector D. Inspection starts at the character t_i, and it proceeds with a pair of characters t_{i-j} and t_{i+j} until corresponding bits in D become 1^m or $j = m$ holds. Note that the two consecutive loops of LNDM are combined in TSO into one loop (lines 8–10 of Alg. 2). When the actually used bits in bit-vectors are seated to the highest order bits, in TSO the testing of the state vector D is slightly faster than in elsewhere.

Moreover one bit in D stays zero for each occurrence of the pattern in the inner loop on lines 8–10. The zero bits are switched to set bits on line 12. The count of set

[1] So in the right shift the lowest bit of L becomes the highest bit of R. Note that generally this is different from how e.g. gcc compiler handles this way variables of uint64_t type in x86 architecture in 32-bit mode. See also [12, p. 35].

Algorithm 2 **TSO = Two-way Shift-Or**$(P = p_1 p_2 \cdots p_m,\ T = t_1 t_2 \cdots t_n)$

Require: $m \le w$
/* Preprocessing */
1: $mask \leftarrow \sim 0 << (w - m)$ /* $= 1^m\, 0^{w-m}$ */
2: **for all** $c \in \Sigma$ **do** $B[c] \leftarrow mask$
3: **for** $i \leftarrow 1$ **to** m **do** /* Lowest bits remain 0 */
4: $\quad B[p_i] \leftarrow B[p_i]\ \&\ \sim\big(1 << (w - m + i - 1)\big)$ /* & $1^{m-i}\, 0\, 1^{w-m+i-2}$ */
/* Searching */
5: $matches \leftarrow 0$
6: **for** $i \leftarrow m$ **step** m **while** $i \le n$ **do**
7: $\quad D \leftarrow B[t_i];\ \ j \leftarrow 1$
8: $\quad$ **while** $D < mask$ **and** $j < m$ **do**
9: $\quad\quad D \leftarrow D \mid (B[t_{i-j}] << j) \mid (B[t_{i+j}] >> j)$ /* no need for additional masking */
10: $\quad\quad j \leftarrow j + 1$
11: $\quad$ **if** $D < mask$ **then** /* Garbage is in the lowest bits */
12: $\quad\quad E \leftarrow\ (\sim D)\ \&\ mask$
13: $\quad\quad matches \leftarrow matches + popcount(E)$

bits is then calculated with the popcount[2] function [12] on line 13. An easy realization of popcount is the following:

$$\textbf{while } E > 0 \textbf{ do } matches \leftarrow matches + 1;\ \ E \leftarrow (E - 1) \& E$$

This requires $\mathcal{O}(s)$ time in total where s is the number of occurrences. If the locations of occurrences need to be printed out, $\mathcal{O}(m)$ time is needed for every alignment window holding at least one match.

Alg. 2 works correctly when $n \bmod m = m - 1$ holds. If access to $t_{n+1}, \ldots$ is allowed and some character—e.g. 255—does not appear in P, assignment of stopper $t_{n+1} \leftarrow 255$ makes the algorithm work also for other values of n. Another easy way of handling the end of the text is to use Shift-Or algorithm, because same occurrence vectors are disposable.

In Figure 1, there is an example of the execution of TSO for $P =$ `abcab` and $T = \cdots$`xabcabcabx`$\cdots$.

3.2 TSA

Shift-And is a dual method of Shift-Or. Therefore it is fairly straight-forward to modify TSO to a Two-way Shift-And algorithm, TSA for short. The pseudocode of TSA is given as Alg. 3.

In TSA, $B[t_{i-j}]$ and $B[t_{i+j}]$ are brought to state vector on line 8. For example, let $B[t_{i-2}]$ and $B[t_{i+2}]$ be 1010 and 1011, respectively. (In this example and in the subsequent examples all numbers are binary numbers.) Then the corresponding padded bit strings are $((1010+1) << 2) - 1 = 101011$ and $(1011 >> 2) \,|\, 1111 << 2 = 111110$.

Original Shift-Or/Shift-And examines every text character once. Therefore its practical performance is extremely insensitive to the input data. Two-way algorithms check text in alignment windows of m consecutive text positions. A mismatch can be detected immediately based on the first examined text character. In the best case the performance can be $\Theta(n/m)$. On the other hand, if a match is in any position in

[2] Population count, popcount, counts the number of 1-bits in a register or word. On many computers it is a machine instruction; e.g. in Sparc, and in x86_64 processors in AMDs SSE4a extensions and in Intel's SSE4.2 instruction set extension.

```
P = abcab      B[a] = 10110
               B[b] = 01101
               B[c] = 11011
               B[x] = 11111


T = ··· x a b c a b c a b x ···

                a              D = 10110
j = 1         c                   11011
                  b                  01101
                               D = 10110
j = 2       b                    01101
                    c                  11011
                               D = 10110
j = 3     a                     10110
                      a                 10110
                               D = 10110
j = 4   x                      11111
                        b                01101
                               D = 10110
                               E = 01001
                                    ^  ^
                                   2 matches
```

Figure 1. Example of work made in the inner loop of TSO.

Algorithm 3 TSA = Two-way Shift-And$(P = p_1p_2\cdots p_m, T = t_1t_2\cdots t_n)$

Require: $m \leq w$
    /* Preprocessing */
1: **for all** $c \in \Sigma$ **do** $B[c] \leftarrow 0$
2: **for** $i \leftarrow 1$ **to** m **do**
3:     $B[p_i] \leftarrow B[p_i] \mid 1 << (m - i)$ /* $\mid 0^{i-1}10^{m-i}$ */
    /* Searching */
4: $matches \leftarrow 0$
5: **for** $i \leftarrow m$ **step** m **while** $i \leq n$ **do**
6:     $D \leftarrow B[t_i]$; $j \leftarrow 1$
7:     **while** $(D > 0)$ **and** $(j < m)$ **do** /* alternatively $D \neq 0$ */
8:         $D \leftarrow D \mathbin{\&} (((B[t_{i-j}] + 1) << j) - 1) \mathbin{\&} ((B[t_{i+j}] >> j) \mid$
            $(((\sim 0) >> (w - m)) << (m - j)))$ /* $(1^m << (m - j)))$ */
9:         $j \leftarrow j + 1$
10:     **if** $D > 0$ **then** /* alternatively $D \neq 0$ */
11:         $matches \leftarrow matches + popcount(D)$

---

the window, or if the mismatch is detected based on two last examined characters, then $2m - 1$ characters need to be examined. So in the worst case all text characters except the last characters in each alignment window are examined twice.

3.3 Practical optimizations

In modern processors, loop unrolling often improves the speed of bit-parallel searching algorithms [5]. In the case of TSO and TSA, it means that 3, 5, 7, or 9 characters are read in the beginning of the inner loop instead of a single character. We denote these versions by TSOx and TSAx, where x is the number of characters read in the beginning; x is odd. Line 7 of TSO3 is the following:

7: $D \leftarrow (B[t_{i-1}] << 1) \mid B[t_i] \mid (B[t_{i+1}] >> 1);\;\; j \leftarrow\; 2$

Moreover, the shifted values $B[a] << 1$ and $B[a] >> 1$ can be stored to precomputed arrays in order to speed up access.

Many string searching algorithms apply a so called skip loop, which is used for fast scanning before entering the matching phase. The skip loop can be called greedy, if it handles two alignment windows at the same time [18]. Let us denote

$$(B[t_{i-1}] << 1) \,|\, B[t_i] \,|\, (B[t_{i+1}] >> 1)$$

in TSO3 by $f(3, i)$. If the programming language has the short-circuit AND command, then we can use the following greedy skip loop enabling steps of $2m$ in TSO3:

$$\textbf{while}\ f(3, i) = mask\ \&\&\ f(3, i + m) = mask\ \textbf{do}\ i \leftarrow i + 2 \cdot m$$

Because && is the short-circuit AND, the second condition is evaluated only if the first condition holds. The resulting version of TSO3 is denoted by GTSO3. (Initial G comes from greedy. GTSA3 is formed in a corresponding way.)

3.4 Analysis

We will show that TSO is linear in the worst case and sublinear in the average case. For simplicity we assume in the analysis that $m \le w$ holds and w is divisible by m.

The outer loop of TSO is executed n/m times. In each round, the inner loop is executed at most $m - 1$ times. The most trivial implementation of popcount requires $\mathcal{O}(m)$ time. So the total time in the worst case is $\mathcal{O}(nm/m) = \mathcal{O}(n)$.

When analyzing the average case complexity of TSO, we assume that the characters in P and T are statistically independent of each other and the distribution of characters is discrete uniform. We consider the time complexity as the number of read characters.

In each window, TSO reads $1 + 2k$ characters, $0 \le k \le m - 1$, where k depends on the window. Let us consider algorithms TSOr, $r = 1, 2, 3, \ldots$, such that TSOr reads an r-gram in the window before entering the inner loop. For odd r, TSOr was described in the previous section. For even r, TSOr is modified from TSO$(r-1)$ by reading $t_{i-r/2}$ before entering the inner loop. It is clear that TSOr_2 reads at least as many characters as TSOr_1, if $r_2 > r_1$ holds. Let us consider TSOr as a filtering algorithm. The reading of an r-gram and computing D for it belong to filtration and the rest of the computation is considered as verification. The verification probability is $(m - r + 1)/\sigma^r$. The verification cost is in the worst case $\mathcal{O}(m)$, but only $\mathcal{O}(1)$ on average. The total number of read characters is rn/m in filtration. When we select r to be $\log_\sigma m$, TSOr is sublinear. Because TSOr never reads less characters than TSO1 = TSO, we conclude that also TSO is sublinear.

In other words, the time complexity of TSO is optimal $\mathcal{O}(n \log_\sigma m/m)$ with a proper choice of r for $m = \mathcal{O}(w)$ and $\mathcal{O}(n \log_\sigma m/w)$ for larger m.

The time complexity of preprocessing of TSO is $\mathcal{O}(m+\sigma)$. Because of the similarity of TSO and TSA, TSA has the same time complexities as TSO. The space requirement of both algorithms is $\mathcal{O}(\sigma)$.

4 Variations of Shift-Add

4.1 Two-way Shift-Add

The basic idea in Shift-Add algorithm is to simultaneously evaluate the number of mismatches in each inside field using L bits. The highest bit in each field is an overflow bit, which is used in preventing the error count rolling to the next field. The original Shift-Add algorithm actually used two state vectors, *State* and *Overflow* which were shifted L bits forward. Opposite this, two-way approach in exact matching is successful due to simple (one statement) analogy to the one-way algorithm (Shift-Or, Shift-And). Such an improved (one statement) Shift-Add is introduced in the next section.

The core problem is addition; there can be up to m mismatches. When in some position k errors is reached, we should stop addition into it. In the occurrence vector array, $B[\,]$, only the lowest bit in each field may be set. The key trick is to use the overflow bits in the state vector D. We take the logical AND operation between the applied occurrence vector and the $L-1$ right shifted complemented state vector D. Then the complemented overflow bits and the possibly set bits in the occurrence vector are aligned, and addition happens only when there is no overflow.

This idea is applied in the Two-way Shift-Add$q$ The limitation of Two-way Shift-Add on error level $k=0$ is that each field needs 2 bits.

When bit-vectors are aligned to the lowest order bits, the unessential bits in the right shifted occurrence vector fall off immediately, and in the right shifted ones they do not disturb because bit-vectors are unsigned.

On line 12 the shown form is required with character classes [16, p. 78]; otherwise also substraction works. The form of line 14 depends on q as before. Notice that there can happen larger overflows, but as long as $k \le q$ it does not matter; otherwise we need a larger value for L. Then the minimum value of L is $\lceil \log_2(q+1) \rceil + 1$.

Algorithm 4 Two-way Shift-Add$q(P = p_1p_2\cdots p_m,\ T = t_1t_2\cdots t_n,\ k)$

Require: $m \cdot L \le w$ and $L \ge \max\left\{2, \lceil \log_2(\max\{k,q\}+1)\rceil + 1\right\}$ and $m > (q+1)$ `div` 2

/* Preprocessing */

1: $mask \leftarrow 0$
2: **for** $i \leftarrow 1$ **to** m **do**
3: $\quad mask \leftarrow (mask \ll L) \mid \big((1 \ll (L-1)) - k\big)$
4: **for all** $c \in \Sigma$ **do** $BW[c] \leftarrow mask$
5: $mask \leftarrow 0$
6: **for** $i \leftarrow 1$ **to** m **do**
7: $\quad mask \leftarrow (mask \ll L) \mid 1$
8: **for all** $c \in \Sigma$ **do** $B[c] \leftarrow mask$ /* $mask = (0^{L-1}\,1_2)^{m-1}$ */
9: $mask \leftarrow mask \ll (L-1)$ /* $mask = (1\,0_2^{L-1})^{m-1}$ */
10: **for** $i \leftarrow 1$ **to** m **do**
11: $\quad BW[p_i] \leftarrow BW[p_i] - \big(1 \ll L\cdot(i-1)\big)$
12: $\quad B[p_i] \leftarrow B[p_i]\ \&\ \sim\!\big(1 \ll L\cdot(i-1)\big)$ /* $-\ \big(1 \ll L\cdot(i-1)\big)$ also works normally */

/* Searching */

13: **for** $i \leftarrow m$ **step** m **while** $i \le n$ **do**
14: $\quad D \leftarrow BW[t_i]\ +\ (B[t_{i-1}] \ll L)\ +\ (B[t_{i+1}] \gg L)$ /* this one is for $q=3$ */
15: $\quad j \leftarrow (q{+}1)$ `div` 2 /* integer division – values of q are odd */
16: $\quad$ **while** $j < m$ **and** $(\sim\! D)\ \&\ mask$ **do**
17: $\quad\quad D \leftarrow D\ +\ \big(\sim\! D \gg (L-1)\big)\ \&\ B[t_{i-j}] \ll (L\cdot j)$
$\quad\quad\quad +\ \big(\sim\! D \gg (L-1)\big)\ \&\ B[t_{i+j}] \gg (L\cdot j)$
18: $\quad\quad j \leftarrow j+1$
19: $\quad E \leftarrow (\sim\! D)\ \&\ mask$
20: $\quad$ **while** E **do**
21: $\quad\quad$ report an occurrence /* shifting of E is not needed */
22: $\quad\quad E \leftarrow E\ \&\ (E-1)$ /* turning off rightmost 1-bit */

Figure 2 shows an example how Two-way Shift-Add finds a match. Unrelevant bits are not shown; they are all zeros. On each field (of L bits) in D the highest bit is an overflow bit, which indicates that there is no match on the corresponding text position. Vertical lines limit the computing area having interesting bit fields.

T	=		a b a d a c a d c $\cdots$		
P	=		b a c a c		
k	=		1		
L	=		3		One bit unnecessarily large
$B[\mathrm{a}]$	=		001 000 001 000 001		Shown order of bit fields corresponds to P backwards
$B[\mathrm{b}]$	=		001 001 001 001 000		Occurrences = 000
$B[\mathrm{c}]$	=		000 001 000 001 001		
$B[\mathrm{d}]$	=		001 001 001 001 001		As all other characters that do not appear in P
$BW[\mathrm{a}]$	=		011 010 011 010 011		Again P backwards
$BW[\mathrm{b}]$	=		011 011 011 011 010		011 minus number of errors still allowed
$BW[\mathrm{c}]$	=		010 011 010 011 011		
$BW[\mathrm{d}]$	=		011 011 011 011 011		
$BW[t_5] = BW[\mathrm{a}]$	=		011 010 011 010 011		Starting to check next m positions
$+B[t_4] = B[\mathrm{d}] << 3$	=	001	001 001 001 001		
$+B[t_6] = B[\mathrm{c}] >> 3$	=		000 001 000 001	001	Starting with $q = 3$ characters
$= D$	=		100 011 101 011 100		Note that overflow depends on q
$+B[t_3] = B[\mathrm{a}] << 6$	=	000	001 000 001		Only lowest bits in fields may be set
$\& \sim D >> (L-1)$	=		0 1 0 1 0		So only the overflow bit is relevant on each field
$+B[t_7] = B[\mathrm{a}] >> 6$	=		001 000 001	000$\cdots$	
$\& \sim D >> (L-1)$	=		0 1 0 1 0		
$= D$	=		100 011 101 011 100		Second and fourth position look promising
$+B[t_2] = B[\mathrm{b}] << 9$	=	001	001 000		
$\& \sim D >> (L-1)$	=		0 1 0 1 0		
$+B[t_8] = B[\mathrm{d}] >> 9$	=		001 001	001$\cdots$	
$\& \sim D >> (L-1)$	=		0 1 0 1 0		
$= D$	=		100 011 101 100 100		Overflow also in fourth position
$+B[t_1] = B[\mathrm{b}] << 12$	=	001	000		
$\& \sim D >> (L-1)$	=		0 1 0 0 0		
$+B[t_9] = B[\mathrm{c}] >> 12$	=		000	001$\cdots$	Last characters give only little information
$\& \sim D >> (L-1)$	=		0 1 0 0 0		
$= D$	=		100 011 101 100 100		
E	=		0 1 0 0 0		Match in second position

Figure 2. Example of checking m positions in Two-way Shift-Add.

4.2 Analysis

The worst case analysis is similar to the analysis of TSO/TSA given in subsection 3.4. For simplicity we assume in the analysis that $m \leq w$ holds and w is divisible by m. The outer loop of TSAdd$q$ is executed n/m times, and in each iteration $\mathcal{O}(m)$ text characters are read and $\mathcal{O}(m)$ occurrences are reported. Thus, the total time complexity is $\mathcal{O}(n/m) \cdot \mathcal{O}(m + m) = \mathcal{O}(n)$ for the worst case.

On the average case TSAdd$q$ is sublinear. It can been seen from the test results where the search time decreases when m gets larger.

4.3 Tuned Shift-Add

Algorithm 5 is Tuned Shift-Add. It is a minimalist version of Shift-Add algorithm. If bitvectors fit into computer register, the worst- and average-case complexity of the original Shift-Add algorithm $\mathcal{O}(n)$ [3, p. 75]; also Tuned Shift-Add is linear. The original Shift-Add algorithm is using an *overflow* vector in addition to the state vector (here D). The essential difference between the original Shift-Add algorithm and the Tuned Shift-Add is the state update. Using the same variable naming as in the Tuned Shift-Add the line 11 in Tuned Shift-Add was in original Shift-Add as follows. (Overflow bits are in the *ovmask*; only the highest bit in each bit field is set.)

$D \leftarrow \big((D \ll L) \ + \ BW[t_i]\big) \ \& \ mask2$
$overflow \leftarrow \big((overflow \ll L) \mid (D \ \& \ ovmask)\big) \& \ mask2$
$D \leftarrow D \ \& \ \sim ovmask$ /* clears overflow bits */

Algorithm 5 Tuned Shift-Add($P = p_1 p_2 \cdots p_m$, $T = t_1 t_2 \cdots t_n$, k)

Require: $m \cdot L \leq w$ and $L \geq \max\{2, \lceil \log_2(k+1) \rceil + 1\}$
/* Preprocessing */
1: $mask \leftarrow 0$
2: **for** $i \leftarrow 1$ **to** m **do**
3: $\quad mask \leftarrow (mask \ll L) \mid 1$
4: **for all** $c \in \Sigma$ **do** $B[c] \leftarrow mask$
5: **for** $i \leftarrow 1$ **to** m **do**
6: $\quad B[p_i] \leftarrow B[p_i] \ \& \ \sim\big(1 \ll L \cdot (i-1)\big)$ /* $-\big(1 \ll L \cdot (i-1)\big)$ also works normally */
7: $mask \leftarrow 1 \ll (L \cdot m - 1)$
8: $Xmask \leftarrow \big(1 \ll (L-1)\big) - (k+1)$
/* Searching */
9: $D \leftarrow \sim 0$ /* $= 1_2^w$ */
10: **for** $i \leftarrow 1$ **to** n **do**
11: $\quad D \leftarrow \big((D \ll L) \mid Xmask\big) + \big(B[t_i] \ \& \ (\sim(D \ll 1))\big)$
12: $\quad$ **if** $(D \& mask) = 0$ **then**
13: $\quad\quad$ report an occurrence ending at i

5 Experiments

The tests were run on Intel Core i7-860 2.8GHz, 4 cores, with 16GiB memory; L2 cache is 256KiB / core and L3 cache: 8MiB. The computer is running Ubuntu 12.04 LTS, and has gcc 4.6.3 C compiler. Programs were written in the *C* programming language and compiled with gcc compiler using `-O3` optimization level. All the algorithms were implemented and tested in the testing framework[3] of Hume and Sunday [13]. New

[3] Hume and Sunday test framework allows directly and precisely measure preprocessing times. Test pattern can be selected as considered appropriate. This kind testing method where each algorithm is coded and separately ensures that the tested algorithms can not affect to each other by placement of data structures in memory and data cache. We have tested the search speed of e.g. Sunday's algorithm and various Boyer–Moore variations with implementations made by others. Thus we believe that implementations enclosed in Hume and Sunday test framework are very efficient. This kind of comparison makes it also possible to learn coding of efficient implementations. We encourage everybody to make comparisons with different implementations of same and similar algorithms.

algorithms were compared with the following earlier algorithms: Shift-Or[4] (SO) [3], FSO [7], FAOSO [7], BNDM [15], and LNDM [10]. The given run times of FAOSO are based on the best possible parameter combination for each text and pattern length. We have only 32 bit version of FAOSO, but all other tested algorithms were using 64-bit bit-vectors. For longer patterns than roughly 20 characters there are algorithms [6] which are faster than ones used in here. The results for pattern lengths are shown to demonstrate the behavior of the new algorithms.

We did not test the variations [4] of the Wide-Window algorithm [11], because according to the original experiments [4], these algorithms are only slightly better than BNDM. In addition, they require $m \leq w/2$.

In the test runs we used three texts: binary, DNA, and English, the size of each is 2 MB. The English text is the prefix of the KJV Bible. The binary text is a random text in the alphabet of two characters. The DNA text is from the genome of fruitfly (*Drosophila melanogaster*). Sets of patterns of various lengths were randomly taken from each text. Each set contains 200 patterns.

Data	Algorithm	2	4	8	12	16	20	30	40	50	60
Binary	SO	$\boxed{465}$	$\boxed{465}$	465	465	465	465	466	469	466	465
	FSO	1406	707	$\boxed{268}$	$\boxed{241}$	$\boxed{234}$	234	235	236	235	—
	FAOSO	3522	1728	859	745	695	469	372	239	263	239
	BNDM	1892	1579	1059	723	554	452	316	246	201	171
	LNDM	2814	2291	1573	1166	925	767	544	421	346	294
	TSA	1999	1501	927	641	491	399	276	215	177	152
	TSO	1565	1129	673	455	344	279	188	142	$\boxed{114}$	$\boxed{96.4}$
	TSO3	1429	1158	718	502	385	316	219	172	142	122
	TSO5	1704	911	632	462	359	297	207	161	135	116
	TSO9	1881	771	473	342	272	$\boxed{229}$	$\boxed{172}$	$\boxed{141}$	121	109
	GTSO3	1409	1165	719	499	381	313	217	169	139	121
	GTSA3	1529	1281	819	571	441	362	252	195	163	141

Table 1. Search time of algorithms (in milliseconds) for binary data

Tables 1–3 show the search times in milliseconds for these data sets. Before measuring the CPU time usage, the text and the pattern set were loaded to the main memory, and so the execution times do not contain I/O time. The results were obtained as an average of 100 runs. During repeated tests, the variation in timings was about 1 percent. The best execution times have been put in boxes. Overall, TSO9, TSO5 and GTSO3 appears to be the fastest for binary, DNA and English data respectively.

Table 1 presents run times for binary data. SO is the winner for $m \leq 4$, FSO for $8 \leq m \leq 16$, TSO9 for $20 \leq m \leq 50$, and TSO9 for $m \geq 60$. Table 2 presents run times for DNA data. FAOSO is the winner for $m = 2$, FSO for $m = 4$, TSO5 for $8 \leq m \leq 40$, and TSO9 for $m \geq 50$. Table 3 presents run times for English data. FSO is the winner for $m \leq 4$, GTSO3 for $8 \leq m \leq 16$, GTSA3 for $m = 20$, and TSO5 for $m \geq 30$.

[4] The performance of the Shift-Or algorithm is insensitive to the pattern length (when $m \leq w$) and also to the input data as long as the number of the matches is relative moderate. The relative speed of some algorithm compared to the speed of Shift-Or on given data and pattern length is suitable for comparing tests with similar data. This relative speed is useful for comparing roughly performance of exact string matching algorithms with different text lengths and processors even in different papers.

Data	Algorithm	2	4	8	12	16	20	30	40	50	60
Dna	SO	464	465	465	464	465	465	465	469	465	465
	FSO	709	$\boxed{272}$	235	235	234	235	235	234	235	—
	FAOSO	$\boxed{372}$	639	524	331	311	212	185	216	217	213
	BNDM	1496	984	548	385	302	248	175	134	111	93.4
	LNDM	2255	1438	843	609	481	398	281	219	181	154
	TSA	1481	869	498	355	285	241	179	143	119	103
	TSO	1353	757	364	243	192	161	117	90.9	74.6	62.7
	TSO3	758	491	295	225	189	164	128	106	89.3	79.8
	TSO5	992	401	$\boxed{215}$	$\boxed{153}$	$\boxed{121}$	$\boxed{102}$	$\boxed{78.1}$	$\boxed{65.6}$	60.9	56.1
	TSO9	1217	465	242	168	131	109	81.1	66.4	$\boxed{57.6}$	$\boxed{52.3}$
	GTSO3	753	474	289	223	191	167	132	107	92.4	74.7
	GTSA3	747	486	296	228	193	169	135	111	96.9	85.3

Table 2. Search time of algorithms (in milliseconds) for DNA data

Data	Algorithm	2	4	8	12	16	20	30	40	50	60
English	SO	465	465	465	465	464	465	464	464	465	465
	FSO	$\boxed{328}$	$\boxed{246}$	235	234	234	234	234	234	232	—
	FAOSO	1165	307	167	156	142	141	198	199	195	198
	BNDM	651	505	342	252	198	164	115	93.3	78.0	68.3
	LNDM	1398	903	561	412	326	272	194	154	126	109
	TSA	1243	652	348	245	195	168	121	99.7	87.1	78.4
	TSO	701	518	328	231	176	141	92.3	69.5	56.6	48.9
	TSO3	485	274	159	121	104	89.1	72.9	64.8	59.1	56.1
	TSO5	701	341	184	132	105	88.9	$\boxed{67.1}$	$\boxed{58.9}$	$\boxed{54.6}$	$\boxed{49.6}$
	TSO9	924	448	235	165	128	107	79.3	64.6	57.7	52.4
	GTSO3	449	249	$\boxed{149}$	$\boxed{115}$	$\boxed{96.5}$	86.6	71.8	63.4	57.6	52.8
	GTSA3	441	252	151	116	97.4	$\boxed{86.4}$	72.1	65.1	58.2	53.7

Table 3. Search time of algorithms (in milliseconds) for English data

5.1 Experiments for *k*-mismatches problem

For the k-mismatch problem we tested the following algorithms: Shift-Add (SAdd), Two-way Shift-Add with q-values 1, 3, and 5 (TSAdd-1, TSAdd-3, TSAdd-5), Tuned Shift-Add (TuSAdd), Average Optimal Shift-Add (AOSA), and CMFN. CMFN is a sublinear multi-pattern algorithm by Fredriksson and Navarro [8], and it is also suitable for approximate circular pattern matching problem.

The text files are same as before. The binary pattern set for $m = 5$ contains only 32 patterns, all different. To make the results comparable with other pattern sets containing 200 patterns, the timings have been multiplied with 200/32. The results were obtained as an average of 300 runs.

Programs were written in the *C* programming language and compiled with gcc compiler using `-O2` optimization level. During preliminary tests we noticed performance decrease in AOSA, which seems to be related to the optimization level in gcc compiler. For example on error level $k = 1$ and optimization `-O2` the search speed was 22%–52% faster than with here used `-O3`.

Tables 4–6 represent the results for the k-mismatches problem.

In our tests the Tuned Shift-Add was faster than the original Shift-Add. Both seem to suffer from relatively large number of occurrences. On $k = 1$ TSAdd-3 showed best performance on all other data set except on 5 nucleotide long DNA patterns. (This

	m	TSAdd-1	TSAdd-3	TuSAdd	SAdd	AOSA	CMFN
English	5	177	$\boxed{137}$	149	231	229	880
	10	98	$\boxed{77}$	145	228	115	270
	20	53	$\boxed{43}$	145	228	51	113
	30	37	$\boxed{30}$	145	228	38	93
DNA	5	226	225	$\boxed{165}$	246	267	2770
	10	136	$\boxed{114}$	145	228	164	1420
	20	69	$\boxed{58}$	145	228	92	1810
	30	47	$\boxed{39}$	145	227	62	3083
Bin	5	333	$\boxed{167}$	625	937	937	1062
	10	167	$\boxed{77}$	603	966	966	440
	20	83	$\boxed{39}$	600	947	467	240
	30	57	$\boxed{30}$	593	943	317	140

Table 4. Search times of algorithms (in milliseconds) for $k = 1$.

	m	TSAdd-1	TSAdd-3	TSAdd-5	TuSAdd	SAdd	AOSA	CMFN
English	5	238	201	186	$\boxed{161}$	245	253	2807
	10	124	107	$\boxed{101}$	145	230	137	533
	20	65	56	$\boxed{51}$	147	216	73	223
DNA	5	322	280	268	$\boxed{255}$	339	497	4203
	10	176	158	151	$\boxed{147}$	239	225	3183
	20	88	79	$\boxed{69}$	146	214	113	3563
Bin	5	354	$\boxed{146}$	270	625	958	937	5688
	10	167	$\boxed{73}$	127	642	962	947	800
	20	82	$\boxed{46}$	67	611	941	470	350

Table 5. Search times of algorithms (in milliseconds) for $k = 2$.

	m	TSAdd-1	TSAdd-3	TSAdd-5	TuSAdd	SAdd	AOSA	CMFN
English	5	299	259	247	$\boxed{209}$	291	377	3936
	10	155	137	133	$\boxed{145}$	236	297	1128
	20	78	70	$\boxed{67}$	145	217	107	292
DNA	5	357	316	$\boxed{310}$	447	536	1073	4290
	10	215	196	194	$\boxed{151}$	241	238	4900
	20	108	99	$\boxed{98}$	148	215	128	5293
Bin	5	333	$\boxed{146}$	250	604	937	917	5524
	10	160	$\boxed{77}$	120	580	910	893	808
	20	83	$\boxed{42}$	61	580	917	450	300

Table 6. Search times of algorithms (in milliseconds) for $k = 3$.

test was rerun, but results remained about the same.) TSAdd-3 was best on all tests using binary text. On English and DNA texts for $k = 2$ and $k = 3$ TSAdd and TuSAdd were the best.

To our surprise CMFN was not competitive in these tests. The macro `bitvector` was defined `unsigned long long`, but we suspect that some other compilation parameter was unoptimal.

6 Concluding remarks

We have presented two new bit-parallel algorithms based on Shift-Or/Shift-And and Shift-Add techniques for exact string matching. The compact form of these algorithms is an outcome of a long series of experimentation on bit-parallelism. The new algorithms and their tuned versions are efficient both in theory and practice. They run in linear time in the worst case and in sublinear time in the average case. Our experiments show that the best ones of the new algorithms are in most cases faster than the previous algorithms of the same type.

References

1. K. Abrahamson: *Generalized string matching*. SIAM Journal on Computing, 16(6) 1987, pp. 1039–1051.
2. A. Amir, M. Lewenstein, and E. Porat: *Faster algorithms for string matching with k mismatches*. Journal of Algorithms, 50(2) 2004, pp. 257–275.
3. R. Baeza-Yates and G. Gonnet: *A new approach to text searching*. Communications of the ACM, 35(10) 1992, pp. 74–82.
4. D. Cantone, S. Faro, and E. Giaquinta: *Bit-(parallelism)^2: Getting to the next level of parallelism*, in Fun with Algorithms, 5th International Conference, FUN 2010, June 2-4, 2010. Proceedings, P. Boldi and L. Gargano, eds., vol. 6099 of LNCS, Springer, 2010, pp. 166–177.
5. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching*. Information Processing Letters, 110(4) 2010, pp. 148–152.
6. B. Durian, H. Peltola, L. Salmela, and J. Tarhio: *Bit-parallel search algorithms for long patterns*, in Experimental Algorithms, 9th International Symposium, SEA 2010, May 20-22, 2010. Proceedings, P. Festa, ed., vol. 6049 of LNCS, Springer, 2010, pp. 129–140.
7. K. Fredriksson and S. Grabowski: *Practical and optimal string matching*, in International Symposium on String Processing and Information Retrieval, SPIRE, LNCS, vol. 12, 2005.
8. K. Fredriksson and G. Navarro: *Average-optimal single and multiple approximate string matching*. ACM Journal of Experimental Algorithmics, 9 2004.
9. Z. Galil and R. Giancarlo: *Improved string matching with k mismatches*. SIGACT NEWS 62, 17(4) 1986, pp. 52–54.
10. L. He and B. Fang: *Linear nondeterministic dawg string matching algorithm*, in String Processing and Information Retrieval, 11th International Conference, SPIRE 2004, October 5-8, 2004, Proceedings, A. Apostolico and M. Melucci, eds., vol. 3246 of LNCS, Springer, 2004, pp. 70–71.
11. L. He, B. Fang, and J. Sui: *The wide window string matching algorithm*. Theoretical Computer Science, 332 2005.
12. J. Henry and S. Warren: *Hacker's Delight*, Addison-Wesley, 2003.
13. A. Hume and D. Sunday: *Fast string searching*. Software—Practice and Experience, 21(11) 1991, pp. 1221–1248.
14. G. Navarro: *A guided tour to approximate string matching*. ACM Computing Surveys, 33(1) 2001, pp. 31–88.
15. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata*. ACM Journal of Experimental Algorithmics, 5(4) 2000.
16. G. Navarro and M. Raffinot: *Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.
17. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in International Symposium on String Processing and Information Retrieval, SPIRE, LNCS, vol. 10, 2003, pp. 80–93.
18. H. Peltola and J. Tarhio: *String matching with lookahead*. Discrete Applied Mathematics, 163(3) 2014, pp. 352–360.
19. S. Wu and U. Manber: *Fast text searching allowing errors*. Communications of the ACM, 35(10) 1992, p. 83.