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, Hannu.Peltola, 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 $T = t_1 t_2 \cdots t_n$ and $P = p_1 p_2 \cdots p_m$ 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 $t_i t_{i+1} \cdots t_{i+m-1} = p_1 p_2 \cdots p_m$. Approximate string matching [13] 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 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 fosters the previous solutions for exact and approximate string matching. First, we illustrate previous algorithms for exact matching which includes Shift-Or and its variants like BNDM (Backward Nondeterministic DAWG Matching),

[⋆] Supported by the Academy of Finland (grant 134287).

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 that covers 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 [14] (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) [16] 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 [5, 11, 10]. LNDM (Linear Nondeterministic DAWG Matching) algorithm [10] is a two-way Shift-And algorithm with sequential symmetric scanning. The pseudocode of LNDM is given as Alg. 1. In LNDM, the alignment window is shifted with fixed steps of m. The window is first scanned leftwards and then rightwards. In our algorithms, these two scans are combined (into one scan).

Algorithm 1 LNDM $(P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n)$

```
    /* Preprocessing */
1: for $c \in \Sigma$ do $B[c] \leftarrow 0$
2: for $j \leftarrow 1$ to $m$ do $B[p_j] \leftarrow B[p_j] \mid 0^{m-j}10^{j-1}$
    /* Searching */
3: for $i \leftarrow m$ step $m$ to $n$ do
4:     $l \leftarrow 0;\ r \leftarrow 0;\ L \leftarrow 1^m;\ R \leftarrow 0^m$
5:     while $L \neq 0^m$ do $L \leftarrow L \,\&\, B[t_{i-l}];\ l \leftarrow l + 1;\ (LR) \leftarrow (LR) >> 1$
6:     $R \leftarrow R >> (m - l)$
7:     while $R \neq 0^m$ do
8:         $r \leftarrow r + 1$
9:         if $R \,\&\, 10^{m-1} \neq 0^m$ then output occurrence
10:        $R \leftarrow (R << 1) \,\&\, B[t_{i+r}]$
```

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 $\lceil \log_2(k + 1) \rceil + 1$. The state i denotes the state of the search between the positions $1, \cdots, i$ of the pattern and positions $j-i+1, \cdots, j$ 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 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). In TSO, the testing of the state vector D is slightly faster, when the bit-vectors are seated to the highest order bits.

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

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

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 bits is then calculated with the popcount[1] function [18] on line 13. An easy realization of popcount is the following:

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

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 the character 255 does appear in P, a guard $t_{n+1} \leftarrow 255$ makes the algorithm work also for other values of n. Otherwise the end of the text must be handled in a different way.

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

T = ...xabcabcabx...
           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 TSO.

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

3.2 TSA

Because Shift-And is a dual method of Shift-Or, 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 padded on line 7. For example, let $B[t_{i-2}]$ and $B[t_{i+2}]$ be 1010_2 and 1011_2, respectively. Then the corresponding padded bit strings are $((1010+1) << 2) - 1 = 101011$ and $(1011 >> 2) \,|\, 1111 << 2 = 111110$.

[1] 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.

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

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

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 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.

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.

3.3 Practical optimizations

In modern processors, loop unrolling often improves the speed of bit-parallel searching algorithms [6]. 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 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 [17]. Let us denote

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

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 $m \leq w$ 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, $0 \leq k \leq m-1$, where k depends on the window. Let us consider algorithms TSOr, $r = 1, 2, 3, ...$, 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 rest of 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 $\mathcal{O}(n \log_\sigma /w)$ 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 k-mismatches problem

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 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) corresponding 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 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 Two-way Shift-Add$q$[2] algorithm shown as Algorithm 4. The limitation of Two-way Shift-Add on error level $k = 0$ is that each field needs 2 bits.

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 $L \ge \max\{2, \lceil \log_2(\max\{k, q\} + 1) \rceil + 1\}$ and $m > (q + 1)$ div 2
    /* Preprocessing */
 1: $mask \leftarrow 0$
 2: for $i \leftarrow 1$ to $m$ do
 3:     $mask \leftarrow (mask << L) \mid ((1 << (L - 1)) - k)$
 4: for all $c \in \Sigma$ do $BW[c] \leftarrow mask$
 5: $mask \leftarrow 0$
 6: for $i \leftarrow 1$ to $m$ do
 7:     $mask \leftarrow (mask << L) \mid 1$
 8: for all $c \in \Sigma$ do $B[c] \leftarrow mask$                 /* $mask = (0^{L-1}\,1_2)^{m-1}$ */
 9: $mask \leftarrow mask << (L - 1)$                               /* $mask = (1\,0_2^{L-1})^{m-1}$ */
10: for $i \leftarrow 1$ to $m$ do
11:     $BW[p_i] \leftarrow BW[p_i] - (1 << L \cdot (i - 1))$
12:     $B[p_i] \leftarrow B[p_i]$ & $\sim(1 << L \cdot (i - 1))$    /* $-\,(1 << L \cdot (i - 1))$ also works normally */
    /* Searching */
13: for $i \leftarrow m$ step $m$ while $i \le n$ do
14:     $D \leftarrow BW[t_i] + (B[t_{i-1}] << L) + (B[t_{i+1}] >> L)$      /* this one is for $q = 3$ */
15:     $j \leftarrow (q + 1)$ div 2                    /* integer division – values of $q$ are odd */
16:     while $j < m$ and $(\sim D)$ & $mask$ do
17:         $D \leftarrow D + (\sim D >> (L - 1))$ & $B[t_{i-j}] << (L \cdot j)$
                    $+ (\sim D >> (L - 1))$ & $B[t_{i+j}] >> (L \cdot j)$
18:         $j \leftarrow j + 1$
19:     $E \leftarrow (\sim D)$ & $mask$
20:     while $E$ do
21:         report an occurrence
22:         $E \leftarrow E$ & $(E - 1)$                          /* turning off rightmost 1-bit */
```

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

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.

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 overflow bit, which indicates that there is no match on corresponding text position. Vertical lines limit the computing area having interesting bit fields.

4.2 Tuned Shift-Add

Algorithm 5 is Tuned Shift-Add. It is a minimalist version of Shift-Add algorithm, and so it is linear. The original Shift-Add algorithm is using an *overflow* vector in addition to the state vector (here D). The essential difference between original Shift-Add algorithm and the Tuned Shift-Add is the state update. Using this variable naming

[2] It is obvious that Two-way Shift-Add is sublinear. We will include the exact analysis to the final version.

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
$B[\text{a}]$	=		001 000 001 000 001		Corresponds P backwards
$B[\text{b}]$	=		001 001 001 001 000		Occurrences = 0
$B[\text{c}]$	=		000 001 000 001 001		
$B[\text{d}]$	=		001 001 001 001 001		As all other characters that do not appear in P
$BW[\text{a}]$	=		011 010 011 010 011		Again P backwards
$BW[\text{b}]$	=		011 011 011 011 010		011 minus number of errors still allowed
$BW[\text{c}]$	=		010 011 010 011 011		
$BW[\text{d}]$	=		011 011 011 011 011		
$BW[t_5] = BW[\text{a}]$	=		011 010 011 010 011		Starting to check next m positions
$+B[t_4] = B[\text{d}] << 3$	=	001	001 001 001 001		
$+B[t_6] = B[\text{c}] >> 3$	=		000 001 000 001	001	Starting with $q = 3$ characters
$= D$	=		100 011 101 011 100		Note that overflow depends on q
$+B[t_3] = B[\text{a}] << 6$	=	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
$+B[t_7] = B[\text{a}] >> 6$	=		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
$+B[t_2] = B[\text{b}] << 9$	=	001	001 000		
$\& \sim D >> (L-1)$	=		0 1 0 1 0		
$+B[t_8] = B[\text{d}] >> 9$	=		001 001	001$\cdots$	
$\& \sim D >> (L-1)$	=		0 1 0 1 0		
$= D$	=		100 011 101 100 100		Overflow also in fourth position
$+B[t_1] = B[\text{b}] << 12$	=	001	000		
$\& \sim D >> (L-1)$	=		0 1 0 0 0		
$+B[t_9] = B[\text{c}] >> 12$	=		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.

the line 11 in Tuned Shift-Add was in Shift-Add

```
    $D \leftarrow ((D << L)\ +\ BW[t_i])\ \&\ mask2$
    $overflow \leftarrow ((overflow << L)\ |\ (D\ \&\ ovmask))\&\ mask2$
    $D \leftarrow D\ \&\ \sim ovmask$
```

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 of Hume and Sunday [12]. New algorithms were compared with the following earlier algorithms: Shift-Or (SO) [3], FSO [7], FAOSO [7], BNDM [14], and LNDM [10]. The given run times of FAOSO

Algorithm 5 **Tuned Shift-Add**$(P = p_1p_2 \cdots p_m,\ T = t_1t_2 \cdots t_n,\ k)$

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

are based on the best possible parameter combination for each text and pattern length. All tested algorithms were using 64-bit bit-vectors. The results for patterns longer than 32 are not yet available for all algorithms. We will include them in final paper.

We did not test the variations [5] of the Wide-Window algorithm [11], because according to the original experiments [5], these algorithms are only slightly better than BNDM. In addition, they require $m \le w/2$.

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.

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

Data	Algorithm	2	4	8	12	16	20	30
Binary	SO	$\boxed{465}$	$\boxed{463}$	463	467	466	462	462
	FSO	1421	742	$\boxed{276}$	$\boxed{245}$	246	247	243
	FAOSO	3515	1743	853	747	689	469	374
	BNDM	1998	1678	1106	772	579	472	332
	LNDM	2942	2404	1668	1233	984	822	564
	TSA	1761	1422	884	612	476	395	274
	TSO	1583	1154	676	461	351	283	192
	TSO3	1421	1204	757	528	403	331	231
	TSO5	1661	926	656	483	378	316	229
	TSO9	2161	912	421	294	$\boxed{229}$	$\boxed{189}$	$\boxed{137}$
	GTSO3	1338	1126	722	503	389	234	225
	GTSA3	1272	1218	796	563	437	360	247

Tables 1–3 show the search times in milliseconds for these data sets. Before measuring time, 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 200 runs. During repeated tests, the variation in timings was about 1 percent.

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

Data	Algorithm	2	4	8	12	16	20	30
Dna	SO	$\boxed{468}$	465	463	478	479	468	462
	FSO	733	$\boxed{285}$	245	251	242	244	254
	FAOSO	2523	699	424	331	310	211	185
	BNDM	1563	1031	565	402	314	261	177
	LNDM	2212	1543	888	638	503	418	292
	TSA	1212	734	422	308	246	213	161
	TSO	1123	659	346	235	185	153	111
	TSO3	758	512	302	231	194	167	132
	TSO5	992	415	$\boxed{219}$	$\boxed{156}$	$\boxed{124}$	$\boxed{103}$	$\boxed{80.5}$
	TSO9	1452	585	301	204	158	131	90.4
	GTSO3	754	492	299	222	192	168	132
	GTSA3	673	473	284	226	184	162	124

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

Data	Algorithm	2	4	8	12	16	20	30
English	SO	475	475	474	476	474	469	475
	FSO	$\boxed{334}$	252	240	239	239	239	239
	FAOSO	1213	306	174	158	149	142	198
	BNDM	649	504	342	252	197	164	116
	LNDM	1344	887	561	421	322	273	193
	TSA	887	462	265	189	156	132	99.7
	TSO	713	522	348	234	178	146	90.3
	TSO3	482	273	161	119	101	89.1	73.2
	TSO5	692	342	186	132	109	89.2	67.3
	TSO9	1143	559	289	200	156	128	92.2
	GTSO3	448	256	152	118	97.6	87.2	73.3
	GTSA3	425	$\boxed{243}$	$\boxed{147}$	$\boxed{111}$	$\boxed{95.1}$	$\boxed{85.3}$	$\boxed{70.6}$

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 $8 \leq m \leq 12$, and TSO9 for $m \geq 16$. Table 2 presents run times for DNA data. SO is the winner for $m = 2$, FSO for $m = 4$, and TSO5 for $m \geq 8$. Table 3 presents run times for English data. SO is the winner for $m = 2$ and GTSO3 for $m \geq 4$.

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. It is a sublinear multi-pattern algorithm by Fredriksson and Navarro [8]. (CMFN is also suitable for approximate circular pattern matching problem.)

The text file are same for DNA and English. For binary alphabet it was (by mistake) 131072 characters long.

On two letter alphabet there are only 32 different 5 character strings. Thus the binary pattern set for $m = 5$ contains only 32 patterns. To make the results comparable with other pattern sets containing 200 patterns, the timings have been multiplied with 200/32.

Test results for the k-mismatches problem are in tables 4–6.

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

	m	SAdd	TSAdd-1	TSAdd-3	TuSAdd	AOSA
English	5	259	189	143	156	491
	10	240	103	83	152	245
	20	243	57	46	152	102
	30	239	39	32	152	74
DNA	5	270	282	238	174	523
	10	250	143	121	154	290
	20	233	75	62	155	177
	30	226	50	42	151	127
Bin	5	113	91	78	88	236
	10	67	62	56	43	136
	20	63	33	29	40	73
	30	60	22	20	40	52

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

	m	SAdd	TSAdd-1	TSAdd-3	TSAdd-5	TuSAdd	AOSA
English	5	251	244	211	202	166	499
	10	236	128	111	106	152	258
	20	222	67	58	58	152	132
DNA	5	346	332	293	296	261	700
	10	245	185	164	167	149	478
	20	220	93	83	84	149	240
Bin	5	142	100	88	81	138	331
	10	77	73	65	65	55	166
	20	64	41	38	34	39	125

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

	m	SAdd	TSAdd-1	TSAdd-3	TSAdd-5	TuSAdd	AOSA
English	5	291	300	260	267	209	587
	10	237	155	137	139	146	388
	20	215	78	70	72	145	180
DNA	5	537	357	317	304	449	1105
	10	242	216	197	194	150	476
	20	215	108	99	96	145	247
Bin	5	119	94	81	75	84	325
	10	104	79	69	68	81	242
	20	62	46	43	40	38	123

In our tests the Tuned Shift-Add was faster than the original Shift-Add. Both seem to suffer from relatively large number of occurrences.

During tests we noticed performance decrease in AOSA, which seems to be related to the optimization level. We plan to to make some additional tests on different versions of gcc compiler.

CMFN was not competitive in these tests, and therefore the results are not shown here.

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. Abrahamson, K.R.: Generalized string matching. SIAM Journal on Computing 16 (6), 1039–1051 (1987).
2. Amir, A., Lewenstein, M., and Porat, E: Faster algorithms for string matching with k mismatches. J. Algorithms 50 (2), 257–275 (2004).
3. Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. Communications of the ACM 35(10), 74–82 (1992)
4. Boyer, R.S., Moore, J S.: A fast string searching algorithm. Communications of the ACM 20(10), 762–772 (1977)
5. Cantone, D., Faro, S., Giaquinta, E.: Bit-Parallelism^2: getting to the next level of parallelism. In: Proc. FUN 2010, the 5th International Conference. LNCS, vol. 6099, pp. 166–177 (2010)
6. Ďurian, B., Holub, J., Peltola, H., Tarhio, J.: Improving practical exact string matching. Inf. Process. Lett. 110(4), 148–152 (2010)
7. Fredriksson, K., Grabowski, Sz.: Practical and optimal string matching. In: Proc. SPIRE 2005, the 12th International Conference. LNCS, vol 3772, pp. 376–387 (2005)
8. Fredriksson, K., Navarro, N.: Average-optimal single and multiple approximate string matching. ACM Journal of Experimental Algorithmics 9, article 1.4 (2004)
9. Galil, Z., Giancarlo, R: Improved string matching with k mismatches. ACM SIGACT News 17 (4), 52–54 (1986).
10. He L., Fang B.: Linear nondeterministic DAWG string matching algorithm. In: Proc. SPIRE 2004, the 11th International Conference. LNCS, vol. 3246, pp. 70–71 (2004)
11. He L., Fang B., Sui J.: The wide window string matching algorithm. Theoretical Computer Science 332(1-3), 391–404 (2005)
12. Hume, A., Sunday D.: Fast string searching. Software – Practice and Experience 21(11), 1221–1248 (1991)
13. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. 33(1), 31–88 (2001)
14. Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. ACM Journal of Experimental Algorithmics (JEA) 5(4), 36 pages (2000)
15. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences. Cambridge University Press (2002)
16. Peltola, H., Tarhio, J.: Alternative algorithms for bit-parallel string matching. In: Proc. SPIRE 2003, the 10th International Symposium. LNCS, vol. 2857, pp. 80–94 (2003)
17. Peltola, H., Tarhio, J.: String matching with lookahead. Discrete Applied Mathematics 163(3), 352–360 (2014)
18. Warren, H.S., Jr.: Hacker's Delight. First printing, Addison-Wesley, (2003)
19. Wu, S., Manber, U.: Fast text searching allowing errors. Communications of the ACM 35(10), 83–91 (1992)