Bit-Parallel Search Algorithms for Long Patterns*

Branislav Ďurian[1], Hannu Peltola[2], Leena Salmela[3], 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 {hpeltola, tarhio}@cs.hut.fi
[3] Department of Computer Science, University of Helsinki, P.O.B. 68,
FI-00014 University of Helsinki, Finland leena.salmela@cs.helsinki.fi

Abstract. We present three bit-parallel algorithms for exact searching of long patterns. Two algorithms are modifications of the BNDM algorithm and the third one is a filtration method which utilizes locations of q-grams in the pattern. Two algorithms apply a condensed representation of q-grams. Practical experiments show that the new algorithms are competitive with earlier algorithms with or without bit-parallelism. The average time complexity of the algorithms is analyzed. Two of the algorithms are shown to be optimal on average.

Keywords: Bit-parallel, pattern, q-gram, string matching

1 Introduction

String matching [1, 12] is a classical problem of computer science. The basic task is to find all the occurrences of a pattern string in a text string, where both of the strings are drawn from the same alphabet. There are several variations of the problem. In this paper we concentrate on exact matching of long patterns, which has recently gained attention [3, 5, 8–10, 17].

BNDM (Backward Nondeterministic DAWG Matching) [11] is among the best string matching algorithms. It implements a bit-parallel simulation of a nondeterministic automaton. BNDM is known to be efficient for patterns of at most w characters, where w is the register size of the computer, typically 32 or 64. It is straightforward to extend BNDM to handle longer patterns by simulating a virtual long register with registers of size w, but the resulting algorithms are not very efficient. Long BNDM [11], LBNDM [13], BLIM [8], and SABP [17] are faster bit-parallel solutions than the trivial one. However these algorithms are clearly slower than the best solutions (e.g. Lecroq's algorithm [10]) which do not apply bit-parallelism.

In this paper, we present three new bit-parallel algorithms BXS, BQL, and QF, which are in most cases faster than the previous bit-parallel algorithms for patterns longer than the register size. Our algorithms are also competitive with earlier algorithms without bit-parallelism. Our algorithms apply *q-grams*, i.e. q consecutive characters together. Two of the algorithms are partly based

* Supported by Academy of Finland grants 118653 (ALGODAN) and 134287 (IASMP)

on recent q-gram variations [2] of BNDM for short patterns. The third one is based on checking an alignment q-gram by q-gram introduced by Fredriksson and Navarro [4]. Two of the algorithms, BQL and QF, use a condensed representation of q-grams [10, 15] that enables reasonable space requirements. We also analyze the time complexity of the algorithms. BQL and QF are shown to be optimal on average.

We use the following notations. Let $T = t_1 t_2 \dots t_n$ and $P = p_1 p_2 \dots p_m$ be two strings 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. Formally we search for all positions i such that $t_i t_{i+1} \dots t_{i+m-1} = p_1 p_2 \dots p_m$. In the algorithms we use C notations: '|', '&', and '<<' represent bitwise operations OR, AND, and left shift, respectively.

2 Previous algorithms

Because two of our algorithms are partly based on BNDM, we introduce the code of BNDM. After that we will shortly explain the principles of five earlier algorithms for long patterns.

2.1 BNDM

The key idea of BNDM [11] is to simulate a nondeterministic automaton recognizing all the prefixes of the pattern. The automaton is simulated with bit-parallelism even without constructing it.

In BNDM (see Alg. 1) the precomputed table B associates each character with a bit mask expressing its occurrences in the pattern. At each alignment

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

```
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← 0
 2: for j ← 1 to m do
 3:     B[p_j] ← B[p_j] | (1 << (m − j))
    /* Searching */
 4: i ← 0
 5: while i ≤ n − m do
 6:     j ← m; last ← m; D ← (1 << m) − 1
 7:     while D ≠ 0 do
 8:         D ← D & B[t_{i+j}]; j ← j − 1
 9:         if D & (1 << (m − 1)) ≠ 0 then
10:             if j > 0 then
11:                 last ← j
12:             else
13:                 report occurrence at i + 1
14:         D ← D << 1
15:     i ← i + last
```

of the pattern, the algorithm reads the text from right to left until the whole pattern is recognized or the processed text string is not any substring of the pattern. Between alignments, the algorithm shifts the pattern forward to the start position of the longest found prefix of the pattern, or if no prefix is found, over the current alignment. With the bit-parallel Shift-AND technique the algorithm maintains a state vector D, which has one in each position where a substring of the pattern starts such that the substring is a suffix of the processed text window. The standard BNDM works only for patterns which are not longer than w.

The inner while loop of BNDM checks one alignment of the pattern from right to left. In the same time the loop recognizes prefixes of the pattern. The leftmost one of the found prefixes determines the next alignment of the algorithm.

2.2 Algorithms for long patterns

We consider five earlier algorithms. The first one is a modification of BNDM by Navarro and Raffinot [11]. We call it Long BNDM. In this algorithm, a prefix of w characters is searched with the standard BNDM and in the case of a match of that prefix, the rest of the alignment is verified in the BNDM manner in pieces of w characters. The maximum shift is w.

In LBNDM by Peltola and Tarhio [13], the pattern of length m is partitioned into $\lfloor \frac{m}{k} \rfloor$ consecutive pieces, each consisting of $k = \lfloor \frac{m-1}{w} \rfloor + 1$ characters. This division implies k subsequences of the pattern such that the ith sequence takes the ith character of each piece. 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. Each occurrence of the superimposed pattern is a potential match of the original pattern and thus must be verified. The shift of LBNDM is a multiple of k and at most m. LBNDM works efficiently only for large alphabets.

Külekci [8] introduced BLIM which checks w alignments simultaneously. Starting with a vector of ones of length w, the vector is updated with the AND operation with the mask of a text character in turn until the vector becomes zero. The shifting is based on the character immediately following the window. The maximum shift of BLIM is $w + m$. SABP by Zhang et al. [17] is related to BLIM. In SABP, bitvectors are preprocessed in a so called matching matrix.

The Wide Window algorithm (WW) [5] applies two automata in a window of size $2m-1$. WW is not a bit-parallel algorithm like the others in this section. The search begins from the middle of the window. The window is moved m positions forward until a character occurring in the pattern is found and a forward suffix automaton can start. Then the rest of the match is verified with a reverse prefix automaton. Finally the start position is moved past the current window.

3 New algorithms

In this section we will present our new algorithms BXS, BQL, and QF. All the algorithms use q-grams, and we present the pseudocodes for $q = 3$. The value

$q = 3$ has been selected only for the clarity of presentation. In the rest of the paper the variable w' holds the minimum of m and w.

3.1 BXS

Our first algorithm is BXS (BNDMq with eXtended Shift). We first cut the pattern into $\lceil m/w' \rceil$ consecutive pieces of length w' except for the rightmost piece which may be shorter. Then we superimpose these pieces getting a superimposed pattern of length w'. In each position of the *superimposed pattern* a character from any piece (in corresponding position) is accepted. We then use the following modified version of BNDM to search for consecutive occurrences of the superimposed pattern using bit vectors of length w' but still shifting the pattern by up to m positions. We first initialize the B vectors as if we were searching with the standard BNDM for the superimposed pattern. When searching we rotate the bits in D rather than just shifting them to the left as in the standard BNDM. In the standard BNDM the D vector is guaranteed to die (i.e. all bits are 0)

Algorithm 2 $\mathbf{BXS}_3(P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n)$

Require: $m \geq q$
/* Preprocessing */
1: **for all** $c \in \Sigma$ **do** $B[c] \leftarrow 0$ /* 0^w */
2: $w' \leftarrow \min(m, w)$; $x \leftarrow m - (m \bmod w') + w'$
3: **for** $j \leftarrow 1$ **to** m **do**
4: $B[p_j] \leftarrow B[p_j] \mid (1 \ll ((x - j) \bmod w'))$ /* $0^*10^{(x-j) \bmod w'}$ */
5: **for** $c \in \Sigma$ **do**
6: $B1[c] \leftarrow (B[c] \ll 1) \mid 1$
7: $B2[c] \leftarrow (B[c] \ll 2) \mid 3$ /* $(B \ll 2) \mid 0^{w-2}1^2$ */
/* Searching */
8: $i \leftarrow mq1 \leftarrow m - q + 1$ /* now $q = 3$ */
9: **while** $i \leq n - q + 1$ **do**
10: $D \leftarrow B2[t_{i+2}]$ & $B1[t_{i+1}]$ & $B[t_i]$
11: **if** $D \neq 0$ **then**
12: $j \leftarrow i$; $first \leftarrow i - mq1$
13: **repeat**
14: $j \leftarrow j - 1$
15: **if** $D \geq (1 \ll (w' - 1))$ **then** /* is highest bit set */
16: **if** $j > first$ **then**
17: $i \leftarrow j$ /* possible prefix found; sliding backward */
18: **else** /* verify whole match */
19: **if** $t_{first+1}t_{first+2} \cdots t_{first+m} = p_1p_2 \cdots p_m$ **then**
20: report an occurrence at $first + 1$
21: $D \leftarrow (D \ll 1 \mid 1)$ & $B[t_j]$ /* rotating set highest bit */
22: **else**
23: $D \leftarrow (D \ll 1)$ & $B[t_j]$
24: **until** $D = 0$ **or** $j \leq first$
25: $i \leftarrow i + mq1$

after at most m characters are read because the shift operation inserts zeroes to the right. Now we no longer have this guarantee because of rotating bits in D. Therefore we also need to check that we will not read more than m characters in a window and exit the inner loop of BNDM if this is the case. We further note that the w':th bit of D is set whenever the processed suffix of the current alignment matches a prefix of the original pattern. However, it is also set if the suffix of the alignment matches a prefix of a power of the superimposed pattern even if it does not match a prefix of the original pattern. Thus the shifts of the alignment can be unnecessarily short, and if the w':th bit in D is set after reading m characters, we need to verify for an occurrence of the original pattern.

In practise BXS is faster if we utilize q-grams as in BNDMq [2]. In each alignment we first read the last q characters and update D accordingly. To do this efficiently we store shifted values of B into tables Bi. This reduces the maximum shift length to $m-q+1$. Algorithm 2 shows the pseudo code for BXS with this modification. The computation of D on line 10 is different for each q as well as the computation of Bi tables on lines 5–7. Each B or Bi table needs $\sigma \cdot w$ bits.

BXS does not work well when the superimposed pattern is not sensitive enough, i.e. too many different characters are accepted at the same position. This happens when the alphabet is too small or the pattern is too long. Increasing the value of q can help, and another solution is to use only a substring of the pattern when constructing the superimposed pattern. Of course this limits the maximum shift length. It is also possible to relieve this problem by considering a condensed representation of q-grams introduced in the next section.

3.2 BQL

BQL (BNDMq Long) is our second algorithm. BQL increases the effective alphabet size by using overlapping q-grams, e.g. when using 3-grams the pattern "ACCTGGT" is processed as "ACC-CCT-CTG-TGG-GGT". Thus we effectively search for a pattern of $m-q+1$ overlapping q-grams. Similar to BXS we cut the q-gram pattern into $\lceil (m-q+1)/w' \rceil$ pieces and superimpose them. The B vectors of BNDM are then initialized for the superimposed q-gram pattern.

In the search phase we use a modification of Simplified BNDM [13], which allows us to always shift by $m-q+1$ but still only use w' bits for the bit vectors B and D. We divide the text into nonoverlapping windows of length $m-q+1$ and in each window we do a BNDM like scan from right to left. Whenever the highest bit in D is set, we verify all such alignments of the pattern with the text that the prefix of one of the superimposed pieces is aligned with the processed suffix of the window. When the D vector dies, we shift always by $m-q+1$ to move to the next text window. Algorithm 3 shows the pseudo code of the algorithm. The computation of ch on line 12 is different for each q.

We use the following condensed representation of q-grams to reduce the space usage of the B vectors. The parameter s regulates the number of bits reserved for each character in a q-gram, and q-grams are encoded as the $s \cdot q$ lowest bits of shifted sum of bit representations of ASCII characters, see line 12 of Algorithm 3.

Algorithm 3 $\mathbf{BQL}_{3,s}(P = p_1p_2\cdots p_m, T = t_1t_2\cdots t_n)$

Require: $m > q$ and theoretically $q \cdot s < w$

```
    /* Preprocessing */
 1: for ch ← 0 to (2^{q·s} − 1) do
 2:     B[ch] ← 0                                                    /* 0^w */
 3: w' ← min(m, w); mq1 ← m − q + 1; x ← m − (m mod w') + w'
 4: mask ← (1 << (q · s)) − 1                                  /* 0^{w'−q·s} 1^{q·s} */
 5: ch ← 0
 6: for i ← m downto 1 do
 7:     ch ← ((ch << s) + p_i) & mask
 8:     if i ≤ mq1 then
 9:         B[ch] ← B[ch] | (1 << ((x − i) mod w'))
    /* Searching */
10: i ← mq1
11: while i ≤ n − q + 1 do
12:     ch ← (((((t_{i+2}) << s) + t_{i+1}) << s) + t_i) & mask
13:     if (D ← B[ch]) ≠ 0 then
14:         j ← i
15:         repeat
16:             j ← j − 1
17:             if D ≥ (1 << (w' − 1)) then /* is highest bit set */
18:                 for k ← j step down w' while k ≥ i − mq1 do
19:                     if t_{k+1}t_{k+2}···t_{k+m} = p_1p_2···p_m then /* verify match */
20:                         if k + m ≤ n then
21:                             report an occurrence at  k + 1
22:             ch ← ((ch << s) + t_j) & mask
23:             D ← (D << 1) & B[ch]
24:         until D = 0
25:     i ← i + mq1
```

The vector table B thus needs $2^{s \cdot q} \cdot w$ bits. Roughly the value $s = 1$ is suitable for the binary alphabet, $s = 2$ is good for DNA and natural language, and $s \geq 5$ is good for random data of alphabet of 256 characters. A similar representation has earlier been used by Lecroq [10] and in the code of agrep [15].

In the experiments of Section 5, we used the following modification of line 19 before entering the inner verification loop: We made a guard check of the last 2-gram of the pattern. This modification makes the algorithm faster especially on small alphabets or with long patterns.

3.3 QF

Our third algorithm, QF (Q-gram Filtering), is similar to the approximate string matching algorithm by Fredriksson and Navarro [4], which is not a BNDM based algorithm. As preprocessing we store for each *phase* i, $0 \leq i < q$, which q-grams occur in the pattern in that phase, i.e. start at position $i + j \cdot q$ for any j. To store this information we initialize a vector B for each q-gram where the i:th bit is set if the q-gram occurs in phase i in the pattern.

Algorithm 4 $\mathbf{QF}_{3,s}(P = p_1p_2\cdots p_m, T = t_1t_2\cdots t_n)$

Require: $m > q$ and theoretically $q \cdot s < w$

```
    /* Preprocessing */
 1: for ch ← 0 to (2^{q·s} − 1) do /* note that 2^{q·s} = (1 << (q · s)) */
 2:     B[ch] ← 0                                          /* only 0^q needed */
 3: mq1 ← m − q + 1; ch ← 0; mask ← (1 << (q · s)) − 1     /* 0^{w−q·s}1^{q·s} */
 4: for i ← m downto 1 do
 5:     ch ← ((ch << s) + p_i) & mask
 6:     if i ≤ mq1 then
 7:         B[ch] ← B[ch] | (1 << ((m − i) mod q))          /* here q = 3 */
    /* Searching */
 8: i ← mq1
 9: while i ≤ n − q + 1 do
10:     D ← B[((((t_{i+2} << s) + t_{i+1}) << s) + t_i) & mask]
11:     if D ≠ 0 then
12:         j ← i − mq1 + q      /* end of the leftmost q-gram of an alignment */
13:         repeat
14:             i ← i − q
15:         until i ≤ (j − q)   or
                (D ← (D & B[((((t_{i+2} << s) + t_{i+1}) << s) + t_i) & mask])) = 0
16:         if i < j then
17:             i ← j
18:             for k ← j − q + 1 to j do
19:                 if t_k t_{k+1} t_{k+2} ··· t_{k+m−1} = p_1 p_2 p_3 ··· p_m then
20:                     report an occurrence at  k
21:     i ← i + mq1
```

During searching we read consecutive q-grams in a window and keep track of *active* phases, i.e. such phases that all read q-grams occur in that phase in the pattern. This can be done conveniently with bit parallelism. We maintain a vector D where the i:th bit is set if the i:th phase is active. Initially all phases are active and after reading a q-gram G the vector D can be updated using the preprocessed B vectors: $D = D \;\&\; B[G]$. If we have read all the q-grams of the window and some phase is still active, we must verify for an occurrence of the pattern. When the vector D becomes inactive or after verification, we can shift the alignment past the last read q-gram.

To reduce space usage QF applies the same condensed representation of q-grams as BQL. The pseudocode of QF is shown as Algorithm 4. The **or** operator on line 15 is short-circuit OR. The computation of index expression of B on lines 10 and 15 is different for each q. The vector table B needs $q \cdot 2^{q \cdot s}$ bits. This can be a considerably enhancement compared to BQL, especially if B becomes small enough compared to the data cache.

D actually contains the information describing which *phases* are potential, so we would not need to check them all. Use of that information did not improve performance in practice. If tests on line 15 could be made separately with **goto**s, the test in the **if** statement on the next line would become unnecessary.

4 Analysis

The worst case complexity of our algorithms is $\mathcal{O}(mn)$, and the best case complexity is $\mathcal{O}(nq/(m-q))$.

When analyzing the average case complexity of the algorithms, we assume that the characters are statistically independent of each other and the distribution of characters is discrete uniform. Furthermore, we assume that the alphabet is such that the condensed representation of q-grams in BQL and QF produces a uniform distribution. For simplicity we assume in the analysis that w divides m.

When analyzing the average case complexity of BXS, we assume that $q = 1$. The parameter q in BXS is used to gain a practical speedup but it does not affect the asymptotic complexity of the algorithm. On the other hand, we will see that in the BQL and QF algorithms the value of q has a crucial impact on the average case complexity.

BXS. Let $P = p_1 p_2 \dots p_m$ be the pattern. Let us then construct a pattern

$$P' = ([p_1, p_{1+w}, p_{1+2w}, \dots][p_2, p_{2+w}, p_{2+2w}, \dots] \dots)^{\frac{m}{w}},$$

where the square brackets denote a class of characters and exponentiation the repetition of an element. If we now run the standard BNDM algorithm with the pattern P' on a machine where the length of the computer word is long enough, it will read exactly the same characters and perform exactly the same shifts as BXS with the original pattern P run on a machine with word length w.

The average case complexity of BNDM with a pattern containing classes of characters is $\mathcal{O}(n \log_{\bar{\sigma}} m/m)$ where $\bar{\sigma}$ is the inverse of the probability of a class of characters matching a random character. If there are at most m/w characters in a class then this probability is bounded by $m/(w\sigma)$. Note that this bound should be smaller than 1 and thus $m < w\sigma$ must hold. Now the average case complexity of the algorithm becomes

$$\mathcal{O}(n \log_{w\sigma/m} m/m) = \mathcal{O}\left(\frac{n}{m} \cdot \frac{\log_\sigma m}{1 - \log_\sigma \frac{m}{w}}\right).$$

The above result holds for a random pattern and a random text. However, our pattern P' has a repetitive structure with period m/w and is thus not completely random. Still if the text is random, the algorithm actually cannot perform worse with a repetitive pattern than with a random pattern because the probability of a random text substring matching the pattern in any position is in fact lower for the repetitive pattern as it contains fewer unique substrings. Thus the average case complexity of BXS is

$$\mathcal{O}\left(\frac{n}{m} \cdot \frac{\log_\sigma m}{1 - \log_\sigma \frac{m}{w}}\right).$$

An optimal string matching algorithm has the average case time complexity $\mathcal{O}(n \log_\sigma m/m)$ [16] so BXS is worse than optimal by a factor of $1/(1-\log_\sigma(m/w))$.

BQL. BQL processes the text in windows. There are $n/(m-q+1)$ windows. In each window the algorithm first reads the last q-gram of the window. Let us call a window good if the last q-gram of the window does not match the pattern in any position and let all other windows be called bad. In a good window the algorithm reads q characters and then moves on to the next window. Thus the work done by the algorithm in a good window is $\mathcal{O}(q)$. In a bad window the highest bit of the vector D can be set at most w times triggering m/w verifications each time. Each verification can be performed in $\mathcal{O}(m)$ time. Thus the work done by the algorithm in a bad window can be bounded by $\mathcal{O}(w \cdot m/w \cdot m) = \mathcal{O}(m^2)$. The probability that a window is bad is at most $m/2^{sq}$ and therefore the average complexity of the algorithm can be bounded by

$$\mathcal{O}\left(\frac{n}{m-q+1}\left(q + \frac{m}{2^{sq}} \cdot m^2\right)\right) = \mathcal{O}\left(\frac{nq}{m-q+1} + \frac{n}{m-q+1} \cdot \frac{m^3}{2^{sq}}\right).$$

Let us then choose $q = 3\log_{2^s} m$. Then

$$\mathcal{O}\left(\frac{nq}{m-q+1} + \frac{n}{m-q+1} \cdot \frac{m^3}{2^{sq}}\right) = \mathcal{O}\left(\frac{n\log_{2^s} m}{m}\right).$$

If we further choose s so that $2^s = \Theta(\sigma)$, then $\mathcal{O}\left(n\log_{2^s} m/m\right) = \mathcal{O}\left(n\log_\sigma m/m\right)$ and therefore BQL is optimal on average for an appropriate choice of q and s.

QF. The algorithm by Fredriksson and Navarro [4] (FN for short) is designed for multiple approximate string matching. FN is similar to our QF when we set the number of differences $k = 0$ and the number of patterns $r = 1$. There are two differences between the algorithms. QF counts the occurrences of the q different phases of the pattern separately, while FN disregards the phases and only counts how many differences are at least needed to align the read q-grams with the pattern somehow. Secondly, QF uses a condensed representation of the q-grams, while FN uses plain q-grams.

The condensed representation of the q-grams reduces the alphabet size to 2^s. If we assume that the alphabet size is 2^s, then QF never reads more characters in a window than FN. QF stops handling a window when the read q-grams do not match the pattern q-grams in the same phase. FN cannot stop sooner than QF because the read q-grams can be aligned with the pattern with 0 differences if QF has not stopped reading. Both of the algorithms shift the pattern so that the new window is shifted just past the last read q-gram. Because QF never reads less q-grams in a window than FN, it always makes a shift that is at least as long as in FN. Therefore, the average case complexity of QF cannot be worse than the average case complexity of FN, $\mathcal{O}(n\log_\sigma m/m)$ for $k = 0$ and $r = 1$ when $q = \Theta(\log_\sigma m)$. As the alphabet size in QF is 2^s, the average complexity of QF is $\mathcal{O}(n\log_{2^s} m/m)$ when $q = \Theta(\log_{2^s} m)$. Thus the average complexity of QF is $\mathcal{O}(n\log_\sigma m/m)$ if $2^s = \Theta(\sigma)$ and $q = \Theta(\log_{2^s} m) = \Theta(\log_\sigma m)$. This complexity is optimal for exact matching of a single pattern, and thus the analysis gives a tight bound.

5 Experimental comparison

The tests were run on a 2.8 GHz Pentium D (dual core) CPU with 1 GB of memory. Both cores have 16 KB L1 data cache and 1024 KB L2 cache. The computer was running Fedora 8 Linux. All the algorithms were tested in the testing framework of Hume and Sunday [7]. All programs were written in C and compiled with the gcc compiler 4.1.2 producing x86_64 "32-bit" code and using the optimization level `-O3`.

We used four texts of 2 MB in our tests: binary, DNA, English, and uniformly random of 254 characters[4]. The English text is the beginning of the KJV bible. The DNA text is from the genome of fruitfly (*Drosophila Melanogaster*). The binary and random texts were generated randomly. For each text we have pattern sets of lengths 25, 50, 100, 200, 400, 800, and 1600. The 200 patterns in each pattern set are picked randomly from the same data source as the text. Roughly more than half of the patterns appear in the text. The patterns in each pattern set are from non-overlapping positions.

We compared our algorithms with the following algorithms: Long BNDM [11] (the times for $m = 25$ were run with the standard BNDM), LBNDM [13] for English and random, BLIM [8], WW [5], A4 [14] for DNA, Lecroq [10], and EBOM [3]. We made also preliminary tests on SABP [17]. Its performance seems to be similar to that of BLIM. Lecroq's algorithm and A4 are q-gram variants of the Boyer–Moore–Horspool algorithm [6]. EBOM is an efficient implementation of the oracle automaton utilizing 2-grams. Because Lecroq's algorithm (as described in [10]) has at most 256 values of shift, it is not competitive for long patterns. Therefore we implemented another version called Lecroq2 which has 4096 values of shift. Obviously tests with long patterns were done with such a version in [10].

Table 1 shows average times of 200 runs in milliseconds. (To get more accuracy the runs with search times less than 10 ms were repeated 3000 times.) The best times have been boxed. The best values of parameters for each algorithm are given for each pattern set. Generally QF was the fastest and BQL was the second best—especially on longer patterns. In most cases the best value of q for BQL was bigger or equal than for QF. BLIM would work faster with $w = 64$ (i.e. using "64-bit" code) except on binaries and DNA for $m = 25$. Lecroq2 is a considerable improvement compared to the basic version on other data sets than binary when $m \geq 400$. The relatively good performance of Long BNDM on Random_{254} seems to be due to a skip loop. Also WW has a related structure.

The times in Table 1 do not include preprocessing based on the patterns. The preprocessing times were unessential for all other algorithms except BLIM, WW, and EBOM. E.g. for English, their preprocessing times grew (according to pattern length) as follows: BLIM from 94 to 6512, WW from 3 to 675, and EBOM from 15 to 214 milliseconds per pattern set.

[4] Our testing environment allows an alphabet of at most 254 characters. So this is not a limitation of the algorithms.

Table 1. Search times for 200 patterns in milliseconds.

	Algorithm	par.	25	par.	50	par.	100	par.	200	par.	400	par.	800	par.	1600
Binary data	Long BNDM		786		526		526		529		529		531		532
	BLIM		875		534		535		536		536		539		542
	WW		1384		801		468		354		283		210		172
	Lecroq	6	304	7	188	8	125	6	137	6	138	8	110	8	102
	Lecroq2	6	313	6	201	8	143	6	139	6	140	8	116	8	111
	EBOM		784		502		314		241		184		91		79
	BXS	5	708	12	507	14	1022								
	BQL	9,1	322	9,1	173	11,1	117	9,1	148	12,1	69	15,1	16	15,1	8.8
	QF	8,1	282	9,1	145	9,1	107	9,1	133	12,1	68	13,1	14	15,1	8.2
DNA data	Long BNDM		455		298		298		297		298		300		300
	BLIM		518		356		357		359		357		359		363
	WW		679		389		226		200		126		79		70
	Lecroq	3	215	4	147	4	111	3	124	3	131	7	106	6	90
	Lecroq2	4	223	4	149	5	110	4	164	8	89	8	18	8	9.8
	A4	4	228	4	156	4	113	6	183	6	83	6	18	6	10
	EBOM		411		262		160		132		80		44		47
	BXS	4	348	7	225	8	211	14	273						
	BQL	5,3	219	5,3	135	7,2	108	7,2	126	7,2	44	8,2	13	8,2	8.0
	QF	4,3	165	5,3	110	8,2	101	5,3	105	7,2	40	8,2	9.6	8,2	5.8
English data	Long BNDM		309		247		248		249		249		251		253
	LBNDM		389		238		168		144		123		113		138
	BLIM		371		224		207		191		183		176		165
	WW		406		242		162		152		91		47		44
	Lecroq	3	189	3	135	3	106	3	174	4	135	4	85	3	54
	Lecroq2	3	202	3	142	4	107	8	175	8	91	6	17	8	9.5
	EBOM		213		163		126		101		60		33		28
	BXS	3	211	4	129	6	119	3	145	5	70	9	34	12	31
	BQL	4,3	207	4,3	133	7,2	109	4,3	112	7,2	49	7,2	14	14,1	8.5
	QF	3,4	143	3,4	104	8,2	102	3,5	92	4,3	34	8,2	10	8,2	5.8
Random$_{254}$ *data*	Long BNDM		108		102		101		102		102		103		105
	LBNDM		124		106		102		84		37		11		8.0
	BLIM		164		114		104		135		138		120		120
	WW		117		103		103		100		54		16		9.5
	Lecroq	3	179	3	129	3	103	3	175	3	135	4	102	3	94
	Lecroq2	3	192	3	136	3	104	3	174	5	75	4	12	5	8.5
	EBOM		106		99		97		63		22		10		8.9
	BXS	2	99	2	93	1	94	2	74	2	17	4	7.8	4	6.4
	BQL	2,5	145	2,6	100	2,7	99	2,6	94	2,6	25	2,7	7.8	2,7	4.8
	QF	2,6	98	3,4	95	2,8	96	2,8	70	2,6	16	2,8	5.0	2,8	3.1

6 Concluding remarks

We have presented three bit-parallel q-gram algorithms for searching long patterns. The new algorithms are efficient—both in theory and practice. Our ex-

periments show that the new algorithms are in most cases faster than previous bit-parallel algorithms for long patterns. Our algorithms are also competitive with earlier algorithms without bit-parallelism. QF is the best of the algorithms. We showed that it is optimal on average.

References

1. M. Crochemore and W. Rytter: *Jewels of Stringology*. World Scientific Publishing Company, 2002.
2. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: Tuning BNDM with q-Grams. In *Proc. ALENEX09, the Tenth Workshop on Algorithm Engineering and Experiments*: 29–37, 2009.
3. S. Faro and T. Lecroq: Efficient variants of the backward-oracle-matching algorithm. In *Proc. PSC 2008, The 13th Prague Stringology Conference*: 146–160, 2008.
4. K. Fredriksson and G. Navarro: Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics*, **9**(1.4):1–47, 2004.
5. L. He, B. Fang, and J. Sui: The wide window string matching algorithm. *Theoretical Computer Science*, **332**(1–3):391–404, 2005.
6. R.N. Horspool: Practical fast searching in strings. *Software – Practice and Experience*, **10**(6):501–506, 1980.
7. A. Hume and D.M. Sunday: Fast string searching. *Software – Practice and Experience*, **21**(11):1221–1248, 1991.
8. M.O. Külekci: A method to overcome computer word size limitation in bit-parallel pattern matching. In *Proc. ISAAC 2008: 19th International Symposium on Algorithms and Computation*, *Lecture Notes in Computer Science* **5369**:496–506, 2008.
9. M.O. Külekci: Filter based fast matching of long patterns by using SIMD instructions. In *Proc. of the Prague Stringology Conference 2009*. Pages 118–128.
10. T. Lecroq: Fast exact string matching algorithms. *Information Processing Letters*, **102**(6):229–235, 2007.
11. G. Navarro and M. Raffinot: Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics*, **5**(4):1–36, 2000.
12. G. Navarro and M. Raffinot: *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, New York, NY, 2002.
13. H. Peltola and J. Tarhio: Alternative algorithms for bit-parallel string matching. In *Proc. SPIRE'03, 10th International Conference on String Processing and Information Retrieval*, *Lecture Notes in Computer Science* **2857**:80–93, 2003.
14. J. Tarhio and H. Peltola: String matching in the DNA alphabet. *Software – Practice and Experience*, **27**(7):851–861, 1997.
15. S. Wu and U. Manber: Agrep – a fast approximate pattern searching tool. In Proceedings of the Winter USENIX Technical Conference, pages 153–162, 1992.
16. A. C.-C. Yao: The complexity of pattern matching for a random string. *SIAM Journal on Computing*, **8**(3):368–387, 1979.
17. G. Zhang, E. Zhu, L. Mao, and M. Yin: A bit-parallel exact string matching algorithm for small alphabet. In *Proc. FAW2009, Third International Workshop on Frontiers in Algorithmics*, *Lecture Notes in Computer Science* **5598**:336–345, 2009.