Searching Long Patterns with BNDM

Jorma Tarhio

Department of Computer Science
Aalto University, Finland
firstname.lastname@aalto.fi

Abstract. We present new algorithms for exact string matching of long patterns. Our algorithms read q-grams at constant distances and are variations of the simplified BNDM algorithm. We demonstrate the competitiveness of our solutions through practical experiments. Many of our algorithms were faster than previous methods for English and DNA patterns between 400 and 50,000 in length. Our algorithms were still better when the preprocessing time was taken into account or when the patterns were taken from a different text of the same type.

Keywords: exact string matching, q-gram, fingerprint, experimental comparison of algorithms

1 Introduction

String matching is a widely studied problem in Computer Science. Dozens of algorithms [5] have been developed for searching text patterns. Recently, attention has been paid to searching for long patterns [1, 3, 6–8, 12], and we will introduce new algorithms for this purpose.

One of the fundamental string matching algorithms is the Backward Nondeterministic DAWG Matching algorithm (BNDM) by Navarro and Raffinot [13, 14]. BNDM is a so-called bit-parallel algorithm, and it works for patterns with up to w characters, where w is the size of a computer word, typically 64. BNDM simulates a nondeterministic automaton without explicitly constructing it. A trivial extension of BNDM searches for the w characters long prefix of a pattern. Each extended occurrence of the prefix is checked for a potential match of the original pattern This BNDM version is quite efficient for patterns with up to $2w$ characters but its speed does not increase as patterns become longer than that. The same is true with the multi-word variation of BNDM [14].

Long BNDM (LBNDM) by Peltola and Tarhio [16] is another attempt to find long patterns with BNDM. A pattern is partitioned into approximately m/w consecutive segments, and each segment corresponds to a position in the transformed pattern that accepts any character of the segment at its position. LBNDM is only good for large alphabets. In this paper, we will introduce new variations of BNDM for long patterns. We extend the idea of LBNDM to handle q-grams instead of single characters. In fact, this variation was already proposed in [16], but nobody has so far developed it further.

The previous papers [1, 3, 6–8, 12] dealing with string matching algorithms for long patterns do not mention any specific application. However, such algorithms could be used to search long DNA sequences and recognize large target code pieces in software packages.

Our main focus is on demonstrating the practical efficiency of the new algorithms. Many of our algorithms were faster than previous methods for a wide range of pattern

lengths from 400 to 50,000. Our algorithms were still better when the preprocessing time was included or when the patterns were taken from another text.

The rest of the paper is organized as follows: Section 2 presents background information. Section 3 introduces our new algorithms, and Section 4 discusses implementation details. Section 5 describes the results of our practical experiments, and Section 6 concludes the article.

2 Background

Suppose we are given a finite alphabet Σ, a text $T = t_0 \cdots t_{n-1}$ of length n, and a pattern $P = p_0 \cdots p_{m-1}$ of length m. In the string matching problem, the task is to find all the occurrences of P in T, that is, all possible i such that $t_{i+j} = p_j$ holds for all $0 \leq j < m$.

This problem can be extended considering indeterminate strings, which means that each t_i and p_j is a non-empty set of characters belonging to Σ. In the matching problem of indeterminate strings, the task is to find all the occurrences of P in T, that is, all possible i such that $t_{i+j} \cap p_j \neq \emptyset$ holds for all $0 \leq j < m$. As a special case, P or T is required to contain only singletons.

A q-gram is a continuous string of q characters. Most string matching algorithms move a window of m positions along the text from left to right and try to match the pattern with the m-gram in that window. This window is called an alignment window.

The new variations of BNDM, which we will present in Section 3, are based on Simplified BNDM (SBNDM) [16], which is a variation of BNDM without prefix recognition. SBNDM is able to solve the matching problem of normal strings as well as indeterminate strings. Algorithm 1 is a slightly modified version of SBNDM. The operators &, |, and $<<$ denote bit-parallel and, or, and left shift, respectively. Our bit-level considerations follow big-endianness.

Algorithm 1: SBNDM

```
 1. for c ∈ Σ do B[c] ← 0
 2. for i ← 0 to m − 1 do B[p_i] ← B[p_i] | (1 << (m − i − 1))
 3. i ← m − 1
 4. while i < n do
 5.    d ← B[t_i]
 6.    if d = 0 then i ← i + m
 7.    else
 8.       j ← i
 9.       do i ← i − 1
10.          d ← (d << 1) & B[t_i]
11.       until d = 0
12.       i ← i + m
13.       if i = j then
14.          report occurrence at i − m + 1
15.          i ← i + 1
```

An array B holds the bit vectors of characters in Σ. The width of the bit vectors is w, the size of a computer word. The last character t_i of the alignment window $t_{i-m+1} \ldots t_i$ is read in line 5. If t_i does not appear in P, then $B[t_i]$ is zero and the

window is moved m positions forward. Other characters of the alignment window are read in line 10. The location i of the last character of the window is saved to the variable j in line 8. An occurrence is found when $i + m = j$ holds after the loop of line 9, and the pointer i is advanced by one in line 15. Alternatively, an increment of $m - x$ could be used, where x is the overlap of the pattern with itself, but the practical gain of the latter approach is marginal for long patterns.

With a change in preprocessing, Algorithm 1 works also for indeterminate strings. Line 2 should be replaced by two lines

2a. for $i \leftarrow 0$ to $m-1$ do for $c \in p_i$ do $B[c] \leftarrow B[c] \mid (1 \ll (m-i-1))$
2b. for each character C do for $i \leftarrow 1$ to k_C do $B[C] \leftarrow B[C] \mid B[c_i]$

where the character C is a set containing characters $c_1, \ldots c_{k_C}$ where $k_C > 1$. See [14] for more details.

The QF algorithm by Ďurian et al. [3] for long patterns is based on filtering q-grams. QF is related to the approximate string matching algorithm by Fredriksson and Navarro [9]. During preprocessing, the q-grams of the pattern are divided into q subsets according to the starting position. So the q-grams starting at $p_{i+j \cdot q}$ for any j, $0 \leq i < q$, belong to the same subset. These subsets correspond to phases or offsets of the pattern—only q-grams of the same phase can occur together. To store this information, a bit vector B is initialized for each q-gram such that the i:th bit is set if the q-gram occurs in phase i in the pattern. During searching, consecutive q-grams in the alignment window are read backward and active phases are observed.

3 New Algorithms

We will introduce the Sparse SBNDM algorithm, which is a generalization of the LBNDM algorithm, and its relaxed variation.

3.1 Sparse SBNDM

LBNDM [16] considers single characters while Sparse SBNDM, SSB for short, works on fingerprints of q-grams. A fingerprint or a hash value of a q-gram is a bit vector formed from the q-gram. Like LBNDM, SSB is essentially a filtering algorithm: it produces potential matches which are then checked.

The pattern P is partitioned in r consecutive segments starting from the right end, each consisting of a positions. If $r \cdot a = b < m$ holds, then the prefix of $m - b$ positions is left over. The integers r and a are selected so that a is small and $r \leq w$ and $m-(q-1)-r \cdot a < a$ are valid. A superimposed pattern P' of length r is formed so that each position of P' corresponds to a segment and a set of the fingerprints of the q-grams ending within that segment. The idea is to read q-grams backward at fixed distances in an alignment window, i.e. q-grams ending at $m-1, m-1-a, m-1-2a$, etc. until a mismatch is found or all the q-grams match. So, this is a kind of sampling scheme. The fingerprints of these q-grams are further interpreted as characters and segments as character positions in P', and the algorithm works like the standard SBNDM in matching indeterminate strings. So the first part of the algorithm is a filter, which produces potential matches of the suffix $p_{m-b} \cdots p_{m-1}$. Extensions of these potential matches must then be checked. In this model, $b = r \cdot a$ is the maximal shift of the alignment window.

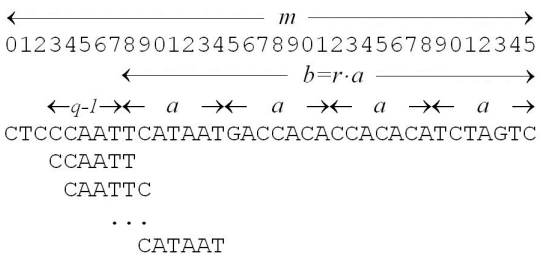

Figure 1. The division of a pattern into segments, an example, $m = 36, q = 6, a = 7$.

Figure 1 illustrates the model of segment partition. Since we have $q = 6$ and $a = 7$ in this example, each segment holds seven 6-grams[1]. For example, the leftmost segment $p_8 \cdots p_{14}$ = `TCATAAT` holds the 6-grams `CCAATT`, `CAATTC`, `AATTCA`, `ATTCAT`, `TTCATA`, `TCATAA`, and `CATAAT` having their last character in that segment.

Algorithm 2: SSB, preprocessing

```
1.  for i ← 0 to 2^q' − 1 do B[i] ← 0
2.  if m ≤ w^2 then x ← 1 else x ← 0
3.  a ← ⌊(m − q + 1 − x)/w⌋ + x; r ← ⌊(m − q + 1)/a⌋
4.  if r > w then r ← w
5.  y ← w − r; b ← r · a; i ← m − 1
6.  while i > m − b − q do
7.     for j ← 1 to a do
8.        d ← F(P, i, q)
9.        B[d] ← B[d] | 1 << y
10.       i ← i − 1
11.    y ← y + 1
```

Algorithm 2 shows the preprocessing of the pattern in SSB. B is an array of bit vectors. The width of a bit vector in B is w. The number of entries $2^{q'}$ depends on q and implementation details (see Section 4). $F(P, i, q)$ computes the fingerprint of the q-gram $p_{i-q+1} \cdots p_i$.

Algorithm 3 is the search phase of SSB. It follows the structure of Algorithm 1. In the same way as in preprocessing, $F(T, i, q)$ computes the fingerprint of the q-gram $t_{i-q+1} \cdots t_i$.

In Algorithm 3, the last q-gram of the window is read in line 4 and other q-grams in line 9 because of loop optimization. Typically, the first read does not match in the case of long q-grams. When a potential match is found, m-grams ending at $t_i, \ldots, t_{i+a-1}$ are checked.

The pseudocode of SSB does not fix, how the fingerprint of a q-gram is formed or computed. In principle, any method used in the literature or any new one can be used. In Section 4 we specify those fingerprint methods used in our practical experiments.

SSB is related to the QF algorithm [3]. In QF, the q-grams starting at $p_{i+j \cdot q}$ for any j belong to the same subset, while in SSB, q-grams ending at a segment belong to the same subset. When scanning backward an alignment window, QF accepts q-grams occurring together in the same phase at any order, while SSB respects their order

[1] The values $q = 6$ and $a = 7$ have been chosen for demonstration purposes and they do not reflect actual values used in the algorithm.

Algorithm 3: SSB, search

```
 1. preprocess P with Algorithm 2
 2. i ← m − 1
 3. while i < n do
 4.    d ← B[F(T, i, q)]
 5.    if d = 0 then i ← i + b
 6.    else
 7.       j ← i
 8.       do i ← i − a
 9.          d ← (d << 1) & B[F(T, i, q)]
10.       until d = 0
11.       i ← i + b
12.       if i = j then
13.          for k = 0 to a − 1 do
14.             if i + k < n then check match at i − m + k
15.          i ← i + a
```

and keeps track of segments in turn. SSB has approximately w subsets of q-grams, but QF has only q subsets and q is typically at most 20. Thus SSB will on average observe earlier than QF, whether an alignment window does not contain a match.

3.2 Relaxed Search

Because the proportion of preprocessing time increases when patterns get longer, we examined ways to make preprocessing lighter. In SSB, each segment has q-grams of its own. We developed a relaxed variation, Relaxed SSB, RSSB for short, where any q-gram of all segments is accepted at any phase of the search. So the array B can now be a Boolean array. Line 9 of preprocessing (Algorithm 2) is changed to

$$9. \quad B[d] \leftarrow 1$$

and lines 7 and 11 can be removed. The loop in line 8 of searching (Algorithm 3) is changed to

$$\begin{aligned} 8. &\quad \text{do } i \leftarrow i - a \\ 9. &\quad\quad d \leftarrow B[F(T, i, q)] \\ 10. &\quad \text{until } d = 0 \text{ or } i > j - b \end{aligned}$$

In SSB, fingerprints are associated with segments, and they can be called local. Fingerprints in RSSB can be called global. Global fingerprints have been used in many earlier algorithms, like in Horspool's algorithm [10] and its q-gram variations. In a manner, LEQ [17] and Takaoka's algorithm [18] use local and global q-grams in a similar way in approximate string matching as SSB and RSSB utilize fingerprints.

4 Implementation Details

When a potential match is found in SSB or RSSB, it should be checked. Actually, not only one candidate is verified but all the m-grams ending at $t_i, \ldots, t_{i+a-1}$, where t_i is the last character of the alignment window. At the implementation level, we use

two techniques. At first the eight-character prefix is compared to the corresponding prefix of the pattern. When the prefixes match, the rest of the candidate is checked with the C library function `memcmp`.

The test $i + k < n$ in line 14 of Algorithm 3 is only intended for processing the end of the text, so it can slow down the search a bit. One option would be handling the end of the text with another algorithm. Alternatively, this test can be removed, if the algorithm is allowed to access characters t_i for $i = n, \ldots, n + a - 2$. In our implementation, the character code 0 is assigned to t_n. If the pattern can contain character code 0, then $x \neq p_{m-1}$ can be assigned to each t_i for $i = n, \ldots, n + a - 2$.

While scanning backward the alignment window, the original BNDM keeps track of the last detected prefix of the pattern. LBNDM has also this feature. We decided to omit that in SSB and RSSB, because its advantage is marginal in the case of long patterns, and extra bookkeeping slows down the algorithm. We tested LBNDM with and without prefix bookkeeping. The latter approach was faster for patterns longer than 500 characters.

In SSB, $b = r \cdot a$ is the maximal shift of the alignment window. However, the heuristic choice in the preprocessing of SSB does not necessarily produce the largest possible b, although the residual $m - b - q + 1$ is always smaller than a. For example, Algorithm 2 sets $r = 59$ and $a = 15$ for $m = 900$ in the case $q = 2$ producing $b = 885$ although the choice $r = 56$ and $a = 16$ yields a longer shift of $b = 896$. However, a longer shift does not necessarily mean that the algorithm becomes faster, because a smaller r can reduce the detection ability of the algorithm. We did not include adjusting b to our final codes, because its gain was marginal on average and even negative in some cases in our preliminary experiments.

In RSSB, r, the number of segments, does not depend on w, and so also larger (or smaller) values of r could be used. However, we decided to keep the same r as in SSB, because it seemed to work well in practice.

For our experiments on SSB, we selected three fingerprint methods associated with the values 2, 13, and 16 of q. The latter two methods were applied to RSSB as well. Because we tested only a few other fingerprint methods, the selected ones are not necessarily the best possible for SSB and RSSB. In the case of SSB2, we used a 2-gram directly as a bit vector of 16 bits. In the case of SSB13, we applied the fingerprint used in QF131 [3]:

$$((\cdots((p_i \ll 1) + p_{i-1} \ll 1) + \cdots + p_{i-11} \ll 1) + p_{i-12}) \;\&\; (2^{13} - 1)$$

which is computed incrementally[2] in preprocessing. In the case of SSB16, we used the SIMD intrinsic function `_mm_movemask_epi8` to transform a 16-gram to a bit vector of 16 bits. The SSEF algorithm [12] applies the same technique. The function `_mm_movemask_epi8` returns a bit vector consisting of the topmost bit of each character of the argument. By shifting the argument bitwise by six to the left, the resulting bit vector contains the second bit from the right for each character, which is a good choice for DNA, because that bit is 1 for C and G, and 0 for A and T resulting an even distribution. For English text, the first bit from the right would be a slightly better choice. Because the observed difference in search times was less than one percent in our preliminary tests, we used the second bit also for English.

[2] Details of an incremental version of the function F for Algorithm 2 are not shown.

5 Experimental Results

We present experimental results in order to compare the behavior of our algorithms against the best known solutions in the literature for searching long patterns.

5.1 Setting

The experiments were run on Intel Core i7-4578U with 16 GB RAM. Our algorithms were implemented[3] in the C programming language and compiled with gcc 5.4.0 using the O3 optimization level. Testing was done in the framework of Hume and Sunday [11]. We used two texts: English (the KJV Bible, 4.0 MB) and DNA (the genome of E. Coli, 4.6 MB). Because the cache size of the processor is 4 MB, we extended both texts to 12 MB by concatenating multiple copies to avoid cache interference with running times [15]. Also, we avoided interference caused by shared memory [15]. Sets of patterns of lengths $m = 16 \cdot 5^i$ for $i = 0, \ldots, 5$ were randomly taken from both texts. Each set contains 100 patterns. The times shown in the tables are averages of at least 100 repeated runs. The times under 100 ms are averages of 1000 repeated runs. The accuracy of the results is about two digits, although more digits are shown in the tables.

5.2 Algorithms

Besides SSB2, SSB13, SSB16, RSSB13, and RSSB16, we show running times for the following four algorithms:

- SSEF [12]
- QF131 [3]
- BRAM7 [7]
- UFM10 [8]

In addition, we tested PBNDM [6], BQL [3], BXS [3], and multi-word BNDM [14], but they were not competitive, and we did not include their results. We were unable to get SSEF2 [1] to work properly. QF131, BRAM7, and UFM10 have variations working better in the lower end of the range of m in our experiments, but we wanted to concentrate on the best variation for $m = 50{,}000$.

5.3 English

Table 1 shows the search times in milliseconds for 100 English patterns. In Table 1 as well as in the subsequent tables, the best time for each m has been underlined as well as the times at most 20 % more than the best time. The algorithms with underlined times can be considered good methods.

In addition to the algorithms mentioned above, we show running times for the following three string matching algorithms in this experiment:

- EPSM [4]
- SBNDM6b [2]
- LBNDM [16]

The first two are examples of general purpose algorithms. EPSM is one of the fastest algorithms for short or medium-size patterns. SBNDM6b is a tuned variation of SBNDM showing almost constant speed for $m > 2w$.

[3] The codes are available at https://users.aalto.fi/tarhio/codes/.

Table 1. Search times in milliseconds for 100 English patterns in 12 MB English text.

Algorithm \ m	16	80	400	2,000	10,000	50,000
EPSM	1,233	765	214	100	173	250
SBNDM6b	1,519	864	874	877	907	1,025
LBNDM	6,284	1,896	1,122	770	1,954	7,392
SSEF	—	1,030	276	81	64	100
QF131	8,447	1,043	441	67	41	148
BRAM7	3,798	1,032	457	104	126	650
UFM10	6,254	1,405	394	62	46	121
SSB2	2,131	1,195	306	84	44	90
SSB13	8,301	1,053	447	65	33	69
SSB16	16,744	1,050	166	33	23	66
RSSB13	8,896	1,284	450	65	48	4,875
RSSB16	14,084	767	166	32	25	76

EPSM was the best for $m = 16$ and RSSB16 as well as EPSM for $m = 80$. SSB2 was the best for $m = 16$ among the algorithms for long patterns. SSB16 and RSSB16 were the best for $400 \leq m \leq 10{,}000$. SSB13 and SSB16 were the best for $m = 50{,}000$. Note that the speed of RSSB13 collapses at $m = 50{,}000$ because global fingerprints match too often. SSB13 worked faster than QF131 for $m \geq 10{,}000$, although the fingerprints are computed in the same way. LBNDM was not competitive at all in this experiment. The size of the effective alphabet in English is too small for LBNDM. Figure 2 illustrates the search times of five selected algorithms given in Table 1.

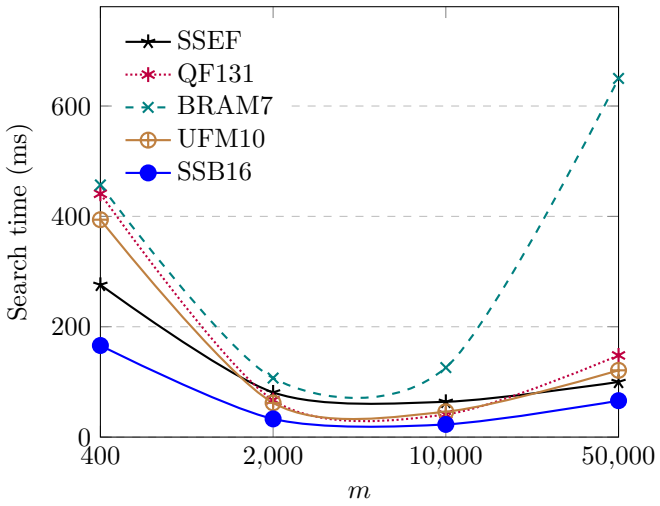

Figure 2. Search times of five algorithms for 100 English patterns in 12 MB English text.

5.4 DNA with Preprocessing

Table 2 shows the search, preprocessing, and total times in milliseconds for 100 DNA patterns. Let us first consider the plain search times. BRAM7 was the best for $m = 16$.

SSB16 and RSSB16 were good for $m \geq 80$. In addition, SSEF was good for $m \geq 2{,}000$ and SSB13 for $m = 50{,}000$. The times of SSB2 and RSSB13 are not shown, because they were not competitive.

In the case of long patterns, the preprocessing time cannot be completely ignored. If the length of the text is at most a few megabytes, the preprocessing time may dominate the total search time. Therefore we measured both preprocessing and search times for DNA. In total times BRAM7 was the best for $m = 16$. RSSB16 was the best for $80 \leq m \leq 2{,}000$ and SSB13 for $m = 10{,}000$ and $50{,}000$. Moreover, SSB16 was good for $m = 400$. Overall, algorithms SSEF, UFM10, and SSB16 suffered the most, when preprocessing was included in the search time. However, it should be kept in mind that the preprocessing of patterns has not necessarily been optimized in the reference methods. Figure 3 illustrates the total search times of five algorithms given in Table 2.

Table 2. Search, preprocessing, and total times in milliseconds for 100 DNA patterns in 12 MB DNA text.

	Algorithm \ m	16	80	400	2,000	10,000	50,000
Search	SSEF	—	1,005	226	33	24	65
	QF131	8,553	1,056	455	70	39	130
	BRAM7	3,656	1,073	531	163	177	550
	UFM10	5,426	1,100	404	80	56	107
	SSB13	8,218	1,056	453	70	32	62
	SSB16	18,708	961	176	28	20	57
	RSSB16	14,361	804	165	28	20	65
Prepro-	SSEF	—	9	10	19	74	456
cessing	QF131	2	2	2	6	26	124
	BRAM7	16	17	18	21	42	128
	UFM10	33	33	34	49	178	702
	SSB13	2	3	3	4	13	57
	SSB16	16	16	17	21	42	149
	RSSB16	2	2	3	7	28	134
Total	SSEF	—	1,014	236	52	99	521
	QF131	8,555	1,058	457	76	65	254
	BRAM7	3,672	1,090	549	184	219	678
	UFM10	5,459	1,133	438	129	234	809
	SSB13	8,220	1,059	456	74	45	119
	SSB16	18,724	977	193	49	62	206
	RSSB16	14,363	806	168	35	48	199

5.5 Maximal Speed

In the case of long patterns, the handling of occurrences of a pattern may interfere with search times. Therefore we made an experiment with fewer occurrences. We divided the DNA text (the genome of E. Coli, 4.6 MB) into two parts with portions 1/3 and 2/3. We picked the patterns from the first part and used the second part as the search text extended to 12 MB. The results are shown in Table 3. BRAM7 was again the best for $m = 16$. SSB16 and RSSB16 were the best for $m \geq 80$. When adding the preprocessing times, which were almost identical to those shown in Table 2, RSSB16 was still better than the others for $m \geq 80$. The speed-up of SSB16 and RSSB16 is

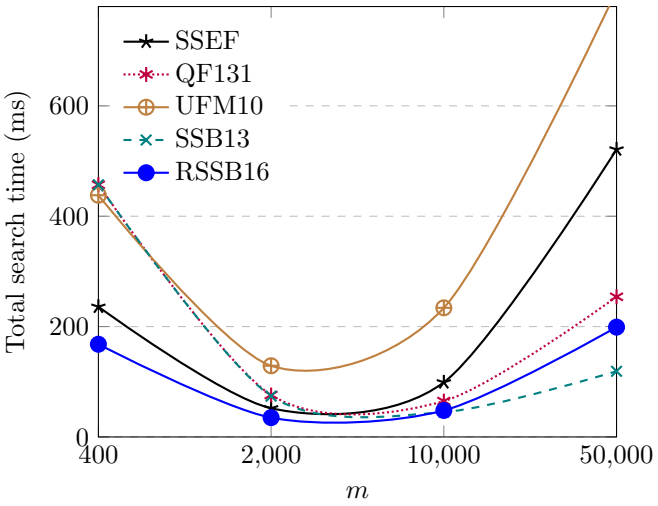

Figure 3. Total search times of five algorithms for 100 DNA patterns in 12 MB DNA text.

astonishing for $m = 50{,}000$ when comparing the search times of Table 2 and 3. On the other hand, this reveals that checking is a bottleneck in running times of our new algorithms.

Table 3. Search times in milliseconds for 100 patterns in 12 MB DNA text. Patterns were taken from another DNA text. The speed-up value is the search time for $m = 50{,}000$ in Table 2 divided by the corresponding time.

Algorithm \ m	16	80	400	2,000	10,000	50,000	speed-up
SSEF	—	1,006	219	30	13	6	11
QF131	8,546	1,088	456	64	16	5	26
BRAM7	3,615	1,049	523	151	61	114	5
UFM10	5,411	1,079	396	82	45	30	4
SSB13	8,211	1,073	449	66	19	3	21
SSB16	18,620	969	177	25	8	1	57
RSSB16	14.220	728	163	25	8	1	65

5.6 Observations

We tested SSB and RSSB variations, which limit the range of checking based on the q-grams of the last segment. This modification gave 5–15 % faster search times but only in the case of m = 50,0000. For other tested lengths of patterns, these new variants were slower or equal.

BNDM and SBNDM can solve the matching problem of indeterminate strings. A nice feature is that SSB2 used in the experiments inherits this characteristic when preprocessing is slightly modified.

In addition, we tested SSB variations, which apply the C data type `__uint128_t` simulating the case $w = 128$. However, these variations were slower than standard variations in our experiments.

6 Concluding Remarks

We introduced a new algorithm SSB and its relaxed variation RSSB for exact string matching of long patterns. In SSB, fingerprints of q-grams are used as characters. SSB is fairly compact and it is a natural extension of BNDM for processing long patterns.

We demonstrated the competitiveness of our solutions through practical experiments. Many of our variations were faster than previous methods for English and DNA patterns between 400 and 50,000 in length. The best ones of our algorithms were still better when the preprocessing time was taken into account or when the patterns were taken from another text of the same type.

Our experiments showed the sublinear behavior of SSB and RSSB. Although increasing q enables efficient search for still longer patterns, processing longer fingerprints requires more time and reduces the gain.

The model of SSB can serve as a platform for new algorithms. For example, we are considering combining the algorithms SSB and QF to make the checking of potential matches lighter.

References

1. M. A. Aydogmus and M. O. Külekci: *Optimizing packed string matching on AVX2 platform*, in High Performance Computing for Computational Science — VECPAR 2018 — 13th International Conference, São Pedro, Brazil, September 17-19, 2018, Revised Selected Papers, H. Senger, O. Marques, R. E. Garcia, T. P. de Brito, R. Iope, S. L. Stanzani, and V. Gil-Costa, eds., vol. 11333 of Lecture Notes in Computer Science, Springer, 2018, pp. 45–61.
2. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.
3. B. Durian, H. Peltola, L. Salmela, and J. Tarhio: *Bit-parallel search algorithms for long patterns*, in Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings, P. Festa, ed., vol. 6049 of Lecture Notes in Computer Science, Springer, 2010, pp. 129–140.
4. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013, 2013, pp. 113–121.
5. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, pp. 13:1–13:42.
6. S. Faro and S. Scafiti: *Pruned BNDM: extending the bit-parallel suffix automata to large strings*, in Proceedings of the 22nd Italian Conference on Theoretical Computer Science, Bologna, Italy, September 13-15, 2021, C. S. Coen and I. Salvo, eds., vol. 3072 of CEUR Workshop Proceedings, CEUR-WS.org, 2021, pp. 328–340.
7. S. Faro and S. Scafiti: *The range automaton: An efficient approach to text-searching*, in Combinatorics on Words - 13th International Conference, WORDS 2021, Rouen, France, September 13-17, 2021, Proceedings, T. Lecroq and S. Puzynina, eds., vol. 12847 of Lecture Notes in Computer Science, Springer, 2021, pp. 91–103.
8. S. Faro and S. Scafiti: *Compact suffix automata representations for searching long patterns.* Theor. Comput. Sci., 940(Part) 2023, pp. 254–268.

9. K. Fredriksson and G. Navarro: *Average-optimal single and multiple approximate string matching.* ACM J. Exp. Algorithmics, 9 2004.

10. R. N. Horspool: *Practical fast searching in strings.* Software: Practice and Experience, 10(6) 1980, pp. 501–506.

11. A. Hume and D. Sunday: *Fast string searching.* Software: Practice and Experience, 21(11) 1991, pp. 1221–1248.

12. M. O. Külekci: *Filter based fast matching of long patterns by using SIMD instructions*, in Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, August 31 - September 2, 2009, J. Holub and J. Zdárek, eds., Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2009, pp. 118–128.

13. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, M. Farach-Colton, ed., Berlin, Heidelberg, 1998, Springer-Verlag, pp. 14–33.

14. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata.* ACM J. Exp. Algorithmics, 5 2000, p. 4.

15. W. Pakalén, H. Peltola, J. Tarhio, and B. W. Watson: *Pitfalls of algorithm comparison*, in Prague Stringology Conference 2021, Prague, Czech Republic, August 30-31, 2021, J. Holub and J. Zdárek, eds., Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2021, pp. 16–29.

16. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in String Processing and Information Retrieval. SPIRE 2003, M. A. Nascimento, E. S. de Moura, and A. L. Oliveira, eds., vol. 2857 of Lecture Notes in Computer Science, Manaus, Brazil, 2003, Springer, Berlin, Heidelberg, pp. 80–94.

17. E. Sutinen and J. Tarhio: *Approximate string matching with ordered q-grams.* Nord. J. Comput., 11(4) 2004, pp. 321–343.

18. T. Takaoka: *Approximate pattern matching with samples*, in Algorithms and Computation, 5th International Symposium, ISAAC '94, Beijing, P. R. China, August 25-27, 1994, Proceedings, D. Du and X. Zhang, eds., vol. 834 of Lecture Notes in Computer Science, Springer, 1994, pp. 234–242.