Searching with Extended Guard and Pivot Loop

Waltteri Pakalén[1], Jorma Tarhio[1], and Bruce W. Watson[2]

[1] Department of Computer Science
Aalto University, Finland
[2] Information Science, Centre for AI Research
School for Data-Science & Computational Thinking
Stellenbosch University, South Africa

Abstract We explore practical optimizations on comparison-based exact string matching algorithms. We present a *guard test* that compares q-grams between the pattern and the text before entering the match loop, and evaluate experimentally the benefit of optimization of this kind. As a result, the *Brute Force* algorithm gained most from the guard test, and it became faster than many other algorithms for short patterns. In addition, we present variations of a recent algorithm that uses a special skip loop where a pivot, a selected position of the pattern, is tested at each alignment of the pattern and in case of failure; the pattern is shifted based on the last character of the alignment. The variations include alternatives for the pivot and the shift function. We show the competitiveness of the new algorithm variations by practical experiments.

Keywords: exact string matching, tune-up of algorithms, guard test, skip loop, experimental comparison

1 Introduction

Searching for occurrences of a string pattern in a text is a fundamental task in computer science. It finds use in many domains such as text processing, bioinformatics, computer vision, and intrusion detection. Depending on the problem definition, the pattern occurrences can be exact, approximate, permuted, or any other variation. Here, we consider the *exact online string matching problem*, where the occurrences are exact and the pattern may be preprocessed but not the text. Formally, the problem is defined as follows: given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ both in an alphabet Σ, find all the occurrences (including overlapping ones) of P in T. String matching is an extensively studied problem with over a hundred published algorithms in the literature, see e.g. Faro and Lecroq [15].

Guard test [20,26,27] is a widely used technique to speed-up comparison-based string matching algorithms. The idea is to test certain pattern positions before entering a match loop. Guard test is a representative of a general optimization technique called loop peeling, where a number of iterations are moved in front of the loop. As a result, the computation becomes faster because of fewer loop tests. Original guard tests deal with single characters — here we consider extended guards: q-grams that are substrings of q characters.

Processor (CPU) development has gradually improved the speed of multicharacter reads — especially the penalty for misaligned memory accesses has disappeared. In our earlier paper [32], we applied q-gram guards to the Dead-Zone algorithm [33] and we anticipated that the guard test with a q-gram might improve the performance of some other algorithms as well. In this paper, we show that this is true. Especially, the transformed Brute Force algorithm is faster than many other algorithms for patterns

Waltteri Pakalén, Jorma Tarhio, Bruce W. Watson: Searching with Extended Guard and Pivot Loop, pp. 90–102.
Proceedings of PSC 2021, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06869-4 © Czech Technical University in Prague, Czech Republic

$m \leq 16$. Testing of q-grams as entities has been used before by Faro and Külekci [13], Sharfuddin and Feng [28] as well as Khan [22].

A few years ago, Al-Ssulami [1] introduced an interesting algorithm called SSM (short for Simple String Matching) which utilizes a special skip loop where a pivot, a selected position of the pattern, is tested at each alignment of the pattern and in case of failure, the pattern is shifted based on the last character of the alignment. This algorithm is not widely known, but it contains a unique shift heuristic that is worth observing. In this paper, we introduce several variations of SSM including alternatives for the pivot and the shift function.

Our emphasis is on the practical efficiency of algorithms and we show the competitiveness of the new algorithm variations by practical experiments.

The rest of the paper is organized as follows: Section 2 reviews typical loop structures of exact string matching algorithms. Section 3 presents how guard test with a q-gram is implemented, Section 4 presents the principles of the SSM algorithm, and Section 5 introduces our variations of SSM. Section 6 shows the results of our practical experiments, and the discussion of Section 7 concludes the article.

2 Loops in String Matching

Let us consider Algorithm 1, which is a general model of string matching of Boyer–Moore type [5]. Two phases alternate during execution. The first phase is the while loop in line 3 which goes through the alignments of the pattern. This loop is called a *skip loop* [20], which is aimed at forwarding the pattern quickly rightwards. The variable j is associated with the alignments: t_j is the last character of an alignment. When the skip loop finds an alignment which is a potential occurrence of the pattern, that alignment is checked in second phase in line 4 and the skip loop is resumed after moving the pattern in line 5. In the preprocessing phase executed before Algorithm 1, the shift functions $\mathbf{shift}_1$ and $\mathbf{shift}_2$ are computed based on the pattern.

Algorithm 1

```
1  j ← m − 1
2  while j < n do
3      while condition do j ← j+ shift_1
4      if t_{j−m+1} ··· t_j = P then report occurrence
5      j ← j+ shift_2
```

In line 1, the pattern is placed at the first position of the text. Line 3 can be missing as it is the case in the original Boyer–Moore algorithm [5] or it can be in a reduced form

if **condition** then

where the then-branch contains lines 4 and 5 as in Horspool's algorithm [18]. In order to reduce tests in the skip loop, a copy of the pattern may be concatenated to the end of the text as a stopper. The **condition** of line 3 examines a suffix of the alignment window. In certain algorithms (Cantone and Faro [8], Peltola and Tarhio [25]), even some characters following the alignment are considered. The displacement $\mathbf{shift}_1$ of the pattern is typically a constant [25], but it can be function based on a character or a q-gram at a constant distance (Faro and Lecroq [14]).

The test in line 4 can be implemented as a match loop in various orders: backward, forward, reversed frequency etc. (Hume and Sunday [20]) or with the memcmp

library function. In the algorithms of BNDM type (Ďurian et al. [11]) the match loop does not compare characters but updates a bit vector. Some algorithms (Hume and Sunday [20], Raita [26,27]) contain guard tests which are located between lines 3 and 4.

The displacement $\mathbf{shift}_2$ of the pattern in line 5 can be a constant (Ďurian et al. [11]) or a function (Boyer and Moore [5]). In some algorithms, there is a separate shift in case of a match in line 4 (Ďurian et al. [11]).

The skip loop of Alg. 1 in line 3 has a unique form in the SSM algorithm [1]:

$$\text{while } p_c \neq t_{j-m+1+c} \text{ do } j \leftarrow j + h[t_j]$$

where p_c is a *pivot character* p_c in P, $0 \leq c < m-1$. SSM uses Horspool's [18] shift table h which carries out the bad character heuristic for p_{m-1}. If a character x does not occur in $p_0 \cdots p_{m-2}$, $h[x]$ is m. Otherwise, $h[x]$ is $m-1-i$, where $p_i = x$ is the rightmost occurrence of x in $p_0 \cdots p_{m-2}$. In the following, we call a skip loop of this kind a *pivot loop*. In other algorithms than SSM, the test character of the skip loop is either the last character of the pattern or a character close to the last character. In SSM, the pivot position is selected so that the shift is long after the pivot loop.

3 Extended Guard Test

The match loop is subject to various optimizations that add or move logic from the match loop into a filter. For instance, the Fast-Search algorithm [7] shifts whenever the right-most character of an alignment mismatches. Verifying this mismatch does not require an explicit comparison between the characters. Instead, the mismatch is encoded in a shift table during preprocessing. Thus, the logic is moved from the match loop into a filter that determines if a match loop is necessary to perform based on the results of the shift table lookup.

Similarly, a guard test is a line of optimizations that compares particular character(s) between the pattern and the alignment window to determine whether a match loop is necessary. Hume and Sunday [20] presented a guard test that compares the least frequent character of the pattern (over the alphabet Σ) with the corresponding text character.

More recently, Khan [22] presented a transformation of the match loop on comparison-based algorithms that involves testing q-grams. A similar approach was earlier developed by Sharfuddin and Feng [28] for Horspool's algorithm [18].

On 64-bit processors, reading a q-gram can be performed in one instruction for $q = 2^i$ for $i = 0, 1, 2, 3$. We implemented the extended guard test as follows. The first q-gram of the pattern is stored in a variable during preprocessing. The first q-gram of an alignment window is stored in another variable. If the values of these variables differ, an occurrence is impossible and the match loop is skipped. If they match, the match loop is executed to check for an occurrence. The match loop can now skip the first q characters because the characters already matched in the guard test.

The processor word size limits q to be less than or equal to that size in bytes. Additionally, the pattern cannot be shorter than the q-gram or otherwise the guard test matches characters outside the right ends of the pattern and the alignment window. An implementation can be adapted to run for any pattern length by branching to a non-guard version of the algorithm in the beginning. Lastly, the guard test is not applicable to every comparison-based algorithm. A skip loop or a similar fast

loop [20] may ruin the benefit of a guard test. And some algorithms, such as the original Boyer–Moore algorithm [5] and the Fast-Search algorithm [7], require knowing the position of the first mismatching character in order to shift correctly.

4 Original SSM

4.1 Algorithm

The main idea of the SSM algorithm [1] is to select a pivot character p_c that will allow a long shift in case of found p_c. As far we know, the shift heuristic of SSM is different from earlier algorithms. The pseudocode of SSM is presented as Algorithm 2. Let the pattern P be aligned with the text T so that p_{m-1} is at t_j. As in the model algorithm Algorithm 1, two phases alternate during execution. The first phase is the pivot loop where the pivot p_c is compared with a text character $t_{j-(m-1)+c}$ and Horspool's [18] shift $h[t_j]$ is taken in case of mismatch (line 4). In case of a match, a potential occurrence of P has been found and it is checked in the second phase in a match loop (lines 6–9). After the match loop, P is shifted according to a proprietary shift table s (line 10) and the pivot loop is resumed. In order to be able to stop the pivot loop, a copy of P is placed at t_n as a stopper (line 1).

Algorithm 2: SSM

```
1  place a copy of P at t_n
2  j ← m − 1
3  while j < n do
4      while p_c ≠ t_{j−m+1+c} do j ← j + h[t_j]
5      i ← m − 1
6      while i ≥ 0 and p_i = t_{j−m+1+i} do i ← i − 1
7      if i < 0 then
8          if j < n then report match at k
9          i ← i + 1
10     j ← j + s[i]
```

4.2 Pivot Character and Shift Tables

The pivot character $p_c, 0 \leq c \leq m-2$, is selected to enable a long safe shift in case of a character match at p_c. Note that p_{m-1} is not allowed to be the pivot in SSM. When the pivot loop stops it is known that p_c was found, and the shift table s utilizes this fact. In order to determine p_c, a distance array[1] $d[i]$ is computed where $d[i]$ is the distance of p_i to its next occurrence to the left or $i+1$ if no such an occurrence exists. Formally,

$$d[i] = \begin{cases} \min(i+1, \min\{k > 0 \mid p_i = p_{i-k}\}) & \text{if } i < m-1 \\ 0 & \text{if } i = m-1 \end{cases}$$

Let i with the largest $d[i]$ be c. If there are more than one such indices, the smallest one is selected. In other words, c is $\min\{i \mid d[i] \geq d[j] \text{ for j} = 0, \ldots, \text{m} - 2\}$.

The shift table s applies two heuristics which consider runs, i.e. sequences of equal characters. If $p_j \cdots p_k$ is a run and $p_{j-1} \neq p_j$ or $j = 0$ holds, then $s_1[i]$ is $i - j + 1$

[1] Sunday [30] used a similar construction in his MS algorithm.

for $i = j, \ldots, k$. Informaly, $s_1[i]$ is a shift to get a different character in P at the text position that caused a mismatch. If $p_{j-1} \neq p_j$ and $p_k \neq p_{k+1}$ or $k = m-1$ holds, then $s_2[j-1]$ is $k - j + 2$. Otherwise, $s_2[j]$ is 1. Informaly, $s_2[i]$ shifts the pattern over a complete run. Finally[2], $s[i]$ is $\max(s_1[i], s_2[i], d[c])$. See an example in Table 1, where P is abbbbf and $p_c = p_1$.

	j					k
i	0	1	2	3	4	5
p_i	a	b	b	b	b	f
d	1	2	1	1	1	0
s_1	1	1	2	3	4	1
s_2	5	1	1	1	2	1
s	5	2	2	3	4	2

Table 1. Data structures of SSM for P = abbbbf.

4.3 Remarks

Al-Ssulami's experiments [1] show that SSM is faster than Horspool's algorithm (Hor) [18]. The comparison is a bit unfair because SSM has a skip loop and Hor does not contain one. The time complexity of SSM is $O(mn)$ in the worst case for $P = a^m$ and $T = a^n$. However, SSM works in linear time for several cases that are in $O(mn)$ for many algorithms of Boyer-Moore type, for example $P = a^{m/2-1}ba^{m/2}$ and $T = a^n$.

The example of Table 1 suggests that the s_2 heuristic could be improved. For example, the value of $s_2[2]$ could be 3 with a relaxed definition. However, we will keep the original s in the following.

The HSSMq algorithm by Al-Ssulami et al. [3] uses a q-gram as a pivot. However, the loop structure of HSSMq is different from SSM because the value of the tested q-gram is used for shifting whereas the shift of the pivot loop in SSM is based on another place of the alignment. Thus HSSMq does not contain a similar pivot loop. Moreover, HSSMq has been designed for long patterns in a small alphabet. Al-Ssulami's third algorithm FHASHq [2] applying q-grams has also a different loop structure. We chose FHASH2 to be one of the reference methods in our experimental comparison.

5 Variations of SSM

As far as we know, the pivot loop of SSM is of a new type of skip loop which has not been presented earlier. The pivot loop opens possibilities for variations in the following features:

- the pivot character
- the shift function of the pivot loop
- the shift function of the match loop

In this section, we will present several variations of SSM. Our aim is to improve the performance of SSM on short English patterns. Each alternative of a feature is given a unique letter, which will be concatenated to the name of a variation. For example, the variation SSM-UBC contains the alternatives U, B, and C.

[2] In the original article [1], s is defined as an outcome of an algorithm without s_1 and s_2.

5.1 Variations of Pivot

In SSM the pivot is chosen so that the shift after the match loop could be long. Another principle would be to minimize the number of exits from the pivot loop. Then the least frequent character of P is a good choice. Let F denote this alternative. The least frequent character is utilized in many algorithms [18,20,23,30], mostly in the match loop. Instead of a single character, the least frequent q-gram could be used, but the size of the required frequency table is impractical for large alphabets. Külekci [23] considers even the use of discontinuous q-grams.

The cost of reading a q-gram from the memory for $q = 2, 4$, and 8 is almost equal to the cost of reading a single character in modern processors. Also implementing a match loop as a single call of the library function memcmp is faster in some processors than a character by character loop. We tried the following q-gram pivots.

- $p_{m-4} \dots p_{m-1}$ (alternative U)
- $p_{m-8} \dots p_{m-1}$ (alternative V)
- $p_0 \dots p_3$ and $p_{m-4} \dots p_{m-1}$ using a short circuit (alternative W)
- P with memcmp (alternative M)

The alternatives U and W work for $m \geq 4$ and V for $m \geq 8$. In the alternatives U, V, and W, the match loop checks the remaining characters of P. In the alternative M there is no match loop at all, and the shift in case of a match is $m - o$ where o is the length of the overlap of P with itself.

5.2 Variations of Shift of the Pivot Loop

If the pivot does not match, we know that the current alignment of P does not hold an occurrence. Therefore the shift need not necessarily be based on t_j but Sunday's shift [30] based on t_{j+1} and Berry and Ravindran's shift [4] based on a 2-gram $t_{j+1}t_{j+2}$ can be applied as well. Let S denote Sunday's shift and let B denote Berry and Ravindran's shift with 16-bit reads, i.e. a 2-gram is read in a single operation. Kalsi et al. [21] show that the shift based on $t_j t_{j+1}$ is better than Berry and Ravindran's shift on DNA data. Let X denote this shift with 16-bit reads.

The restriction that p_{m-1} cannot be a pivot is unnecessary. The algorithm becomes slightly faster without this restriction for large alphabets. Especially patterns like $a^{m-1}b$ can be found faster. Let A denote the variation where p_{m-1} is allowed to be a pivot.

5.3 Variations of Shift of the Match Loop

The shift of SSM for patterns like $(ab)^{m/2}$ is shorter than in the Boyer-Moore algorithm [5] because SSM does not apply the good suffix shift. The good suffix shift is easy to preprocess in linear time and to combine it with the original shift table s. Let C denote the combined shift which includes the good suffix shift.

5.4 Special Variations

A guard test of a single character before the match loop is denoted by G.

Horspool [18] presented the SLFC algorithm which searches for occurrences of the least frequent character of P and checks the alignments of P associated with each

found occurrence. We made a variation L, where the shift of the pivot loop is replaced by the strchr library instruction which searches for the next occurrence of the least frequent character of P. From SSM-WB we made a hybrid algorithm SSM-WBZ for English data, where SSM-L is used when the frequency of the least frequent character of P is below a threshold and SSM-WB is used otherwise.

Let us assume that there is a run of k characters in the pattern and the last character of the run has been chosen as the pivot. In this situation, the shift of the pivot loop is "$j = j + k - 1 + h[t_{j+k-1}]$" in the case of Horspool's shift. Sunday's shift and Berry and Ravindran's shift are treated correspondingly. This modification is faster for patterns containing a run of four or more characters, but it is, unfortunately, slower in the average case.

6 Experimental Results

6.1 Guard Test

We added the guard test to several comparison-based string matching algorithms in order to experimentally evaluate its effectiveness. Table 2 lists the selected algorithms. Each algorithm was transformed with possible values of q. That is, on a 64-bit processor, four values 1, 2, 4, and 8 were possible for q.

ASKIP [9]
Brute Force (BF)
BR [4]
GRASPm [10]
HOR [18]
NSN [17]
RAITA [26]
SMITH [29]
TS [6]
TSW [19]

Table 2. Transformed algorithms.

The experiments were run in the SMART framework [16,12], with default configurations (e.g. the text size was 1 MiB). The processor used was Intel Core i7-6500U with 4 MB L3 cache and 16 GB RAM; this CPU has a Skylake microarchitecture and has none of the misaligned access performance penalties found on some other microarchitectures. The operating system was Ubuntu 16.04 LTS.

The base implementations were taken from the repository of SMART [12]. The results are reported as speed-ups: the ratio of running times of the transformed algorithm and the original one.

Tables 3–6 show the speed-ups on English text, genome sequence, rand2, and rand250, respectively, for $q = 8$ when $m > 4$ and for $q = 4$ when $m = 4$. The other q values have been left out because they were either on par with $q = 8$ or slower. However, there were exceptions for $q = 1$ and $q = 4$ (not shown in the tables): $q = 1$ interestingly displayed up to 1.14 speed-up on both BF and TS, and $q = 4$ was almost always neck and neck with $q = 8$, except on rand2, where $q = 8$ was substantially faster.

The larger the alphabet is, the smaller the speed-ups are. This is evident when comparing rand2 and genome to English and rand250. Table 5 for rand2 exhibits

m	4	8	16	32	64	128	256	512	1024
ASKIP	0.99	1.02	1.02	0.97	1.01	1.01	1.00	1.00	0.99
BF	3.20	3.11	3.25	3.38	3.27	3.21	3.22	3.22	3.23
BR	1.19	1.12	1.10	1.07	1.02	1.00	1.02	1.02	1.00
GRASPM	1.01	1.03	1.01	1.04	1.00	0.96	1.02	1.02	1.04
HOR	1.34	1.26	1.16	1.16	1.13	1.09	1.13	1.10	1.02
NSN	1.25	1.29	1.34	1.37	1.35	1.32	1.32	1.31	1.32
RAITA	1.15	1.14	1.10	1.06	1.05	1.03	1.00	1.06	1.06
SMITH	1.35	1.31	1.23	1.19	1.17	1.07	1.02	1.08	1.06
TS	1.20	1.15	1.13	1.07	1.01	1.01	1.00	1.00	1.04
TSW	1.32	1.25	1.18	1.09	1.08	1.00	1.02	1.00	1.00

Table 3. Speed-ups on English text.

m	4	8	16	32	64	128	256	512	1024
ASKIP	1.17	1.14	1.06	1.03	1.04	1.01	1.00	1.00	1.00
BF	6.71	7.10	7.10	7.32	7.22	7.08	7.03	7.32	7.38
BR	1.72	1.61	1.43	1.30	1.26	1.31	1.27	1.27	1.29
GRASPM	1.12	1.15	1.18	1.28	1.41	1.55	1.56	1.50	1.52
HOR	1.88	1.73	1.64	1.63	1.64	1.63	1.64	1.64	1.63
NSN	1.56	1.65	1.71	1.74	1.72	1.66	1.65	1.65	1.65
RAITA	1.65	1.61	1.53	1.54	1.54	1.53	1.54	1.53	1.54
SMITH	1.73	1.61	1.51	1.57	1.56	1.52	1.52	1.52	1.51
TS	1.50	1.52	1.47	1.43	1.42	1.42	1.36	1.34	1.31
TSW	2.17	1.96	1.72	1.56	1.49	1.50	1.51	1.49	1.49

Table 4. Speed-ups on genome sequence.

m	4	8	16	32	64	128	256	512	1024
ASKIP	1.33	1.63	1.47	1.35	1.26	1.14	1.12	1.03	1.01
BF	4.37	9.81	9.38	9.38	9.16	9.14	9.37	9.37	9.26
BR	1.47	2.01	2.05	2.05	2.06	2.05	2.06	2.06	2.06
GRASPM	1.25	1.97	2.53	3.12	3.51	3.50	3.53	3.45	3.39
HOR	1.64	2.29	2.38	2.39	2.40	2.39	2.40	2.40	2.40
NSN	1.54	1.63	1.69	1.72	1.70	1.62	1.62	1.61	1.61
RAITA	1.76	2.19	2.42	2.55	2.55	2.57	2.56	2.57	2.58
SMITH	1.49	1.96	2.02	2.11	2.09	2.06	2.05	2.04	2.05
TS	1.41	1.81	1.83	1.90	1.92	1.92	1.91	1.86	1.85
TSW	1.77	2.66	2.71	2.67	2.70	2.67	2.69	2.70	2.66

Table 5. Speed-ups on rand2.

significant speed-ups, even for some algorithms that normally enter the match loop relatively rarely (e.g. ASKIP). Almost all the transformed algorithms exhibit substantial speed-ups for rand2. Slightly similar observations apply to genome sequence but to a much lesser extent. In this case, for many algorithms the speed-ups are in range of 1.4 to 1.7. On the other hand, rand250 shows almost no speed-up in any case other than BF. RAITA and SMITH exceed a speedup of 1.1 for a few pattern lengths but otherwise a mere few cases reach 1.05. The English text is somewhere between the other texts. It reaches speed-ups of up to 1.3. In many cases, the speed-ups hover around 1.0 to 1.15. For English, the speed-ups additionally seem to decrease for the longest patterns.

The reason that smaller alphabets work better can be found in the probabilities to match a pair of characters between the pattern and the text. Such a pairwise comparison on average is more probable to match as the alphabet size goes down.

m	4	8	16	32	64	128	256	512	1024
ASKIP	1.02	1.01	1.06	1.03	1.04	1.02	1.01	1.01	1.00
BF	2.03	2.03	1.92	1.97	2.03	2.04	2.06	2.04	1.97
BR	0.98	0.98	0.98	1.00	1.00	1.02	1.02	1.00	1.00
GRASPM	1.00	0.99	0.96	1.02	1.00	1.00	1.00	1.04	0.98
HOR	1.06	1.04	1.01	1.04	1.02	1.00	1.00	1.06	1.00
NSN	0.99	1.00	1.04	1.03	1.04	1.01	1.03	1.03	1.03
RAITA	0.99	1.00	1.00	1.00	0.98	1.02	1.15	1.13	1.06
SMITH	1.13	1.11	1.07	1.07	0.98	1.00	1.02	1.00	0.98
TS	1.04	1.01	1.01	1.03	1.02	1.01	0.99	1.03	1.04
TSW	1.02	0.99	1.01	1.00	0.99	1.01	1.00	0.98	1.00

Table 6. Speed-ups on rand250.

Thus, the original algorithms must carry out multiple pairwise comparisons before coming across a mismatch. Meanwhile, the guard test does not care whether one, two, three, or more pairs match between the q-grams. As long as one of them mismatches, the guard test fails altogether. Hence, there are more savings in execution for smaller alphabets. Similar reasoning applies as to why $q = 4$ and $q = 8$ perform neck and neck. A comparison between a pair of four characters mismatches often enough that the difference is nearly negligible between $q = 4$ and $q = 8$.

While we have included BF in the tables, its speed-ups are somewhat incomparable to the other speed-ups because of its nature. BF is evidently going to be the one benefitting the most from such an optimization. However, note that it became quite competent with the guard test. The transformed BF was the fastest algorithm in the test set on English text, genome sequence and rand2 for $m \leq 16$. This is a remarkable result because short patterns are most important in practice. This is also an example of how technology can beat complicated algorithms (see another example [31] of that).

m=4		8		16		32	
BF4	0.74	BF8	0.75	BF8	0.72	SMITH8	0.67
TSW4	1.18	TSW8	0.92	TSW8	0.77	GRASPM8	0.69
TSW	1.56	SMITH8	1.07	SMITH8	0.80	RAITA8	0.69
TS4	1.57	RAITA8	1.11	RAITA8	0.84	TSW8	0.69
SMITH4	1.61	HOR8	1.13	HOR8	0.86	HOR8	0.69
BR4	1.64	TSW	1.15	GRASPM8	0.87	BF8	0.71
RAITA4	1.67	GRASPM8	1.18	GRASPM	0.88	GRASPM	0.72
NSN4	1.71	GRASPM	1.21	TSW	0.91	BR8	0.72
HOR4	1.71	BR8	1.22	BR8	0.91	RAITA	0.73
GRASPM4	1.81	TS8	1.23	RAITA	0.92	ASKIP	0.74
GRASPM	1.82	RAITA	1.26	SMITH	0.98	TSW	0.75
TS	1.88	BR	1.37	HOR	1.00	ASKIP8	0.76
RAITA	1.92	SMITH	1.40	BR	1.00	BR	0.77
BR	1.95	HOR	1.42	ASKIP8	1.02	SMITH	0.80
NSN	2.13	TS	1.42	TS8	1.04	HOR	0.80
SMITH	2.18	NSN8	1.67	ASKIP	1.04	TS8	0.91
HOR	2.29	ASKIP8	1.70	TS	1.17	TS	0.97
BF	2.37	ASKIP	1.73	NSN8	1.70	NSN8	1.70
ASKIP	2.46	NSN	2.15	NSN	2.28	NSN	2.33
ASKIP4	2.49	BF	2.33	BF	2.34	BF	2.40

Table 7. Ranks of algorithms according to average running times of 500 English patterns in milliseconds for $m = 4, 8, 16, 32$.

Besides BF, we did not notice any drastic changes in the algorithm rankings. Table 7 shows algorithm rankings for English for $m = 4, 8, 16, 32$. The suffix 4 or 8 refer to the transformed algorithm with $q = 4$ or $q = 8$, respectively.

In addition, we tried the guard test with the HASH3, HASH5, and HASH8 algorithms [24]. The results did not show any improvement except with rand2. This was expected, because those algorithms contain an already-efficient skip loop. For instance, HASH3 enters the match loop only when the hash value of the last 3-gram of an alignment window is equal to the hash value of the last 3-gram of the pattern. This means that only a very small number of alignments are checked. As for other tested algorithms, the transformed BF was faster than the HASH3, HASH5, and HASH8 algorithms for $m \leq 16$.

Finally, we ran BF8 against all the 195 algorithms in the SMART repository. BF8 was faster than all the others for $m = 8$ on rand2. Its rank was #16 for $m = 8$ on English.

We also ran experiments with BF8b, a variation of BF8, for $8 \leq m \leq 16$. BF8b tests two 8-grams, the first and last 8-gram of an alignment, with a short-circuit AND instead of a match loop. BF8b was about 25% faster than BF8 on rand2.

In order to test the reliability of our results, we repeated the experiments of HOR in the testing environment of Hume & Sunday (HS) [20]. Table 8 shows speed-ups on genome sequence in both HS and SMART. The table shows concretely that $q = 2$ is inferior and that $q = 4$ and $q = 8$ are very similar. Moreover, the differences between HS and SMART are notable with up to a 12 percentage point difference in the speed-ups.

	m	4	8	16	32	64	128	256
HS	HOR2	1.40	1.46	1.43	1.43	1.44	1.42	1.43
	HOR4	1.97	1.80	1.71	1.69	1.71	1.68	1.68
	HOR8	-	1.85	1.72	1.71	1.71	1.70	1.70
SMART	HOR2	1.38	1.39	1.39	1.37	1.38	1.39	1.38
	HOR4	1.88	1.70	1.62	1.63	1.62	1.61	1.63
	HOR8	-	1.73	1.64	1.63	1.64	1.63	1.64

Table 8. Speed-ups on genome sequence in HS and SMART for HOR with $q \in \{2, 4, 8\}$.

Khan [22] applied q-gram reading inside the match loop. Speed-ups he achieved were much smaller than ours.

6.2 Variations of SSM

The experiments with SSM were run on Intel Core i7-4578U. Algorithms were written 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 [20]. We used two texts: English (the KJV Bible, 4.0 MB) and DNA (the genome of E. Coli, 4.6 MB) for testing. The texts were taken from the SMART repository. Sets of patterns of lengths 5, 10, and 20 were randomly taken from both texts. Each set contains 200 patterns.

Table 9 lists the alternatives introduced in Section 4. Table 10 shows the running times of the original SSM together with twelve variations. We tested even more alternatives (e.g. SSM-L) but we do not show their times because they were not competitive. Besides SSM we ran experiments with two other reference methods:

id	Alternative
A	p_{m-1} allowed as a pivot
B	shift based on $t_{j+1}t_{j+2}$
C	SSM shift combined with the good suffix shift
F	the least frequent character of P as a pivot
G	a guard test
L	strchr on a pivot
M	P as a pivot
S	shift based on t_{j+1}
U	$p_{m-4} \dots p_{m-1}$ as a pivot
V	$p_{m-8} \dots p_{m-1}$ as a pivot
W	$p_0 \dots p_3$ and $p_{m-4} \dots p_{m-1}$ as a pivot
X	shift based on $t_j t_{j+1}$
Z	strchr in case of an infrequent character in P

Table 9. Summary of alternative features of SSM.

	English			DNA		
m	5	10	20	5	10	20
SSM	89	51	31	206	133	103
SSM-ASC	79	48	30	199	135	105
SSM-AFSC	68	42	28	208	160	144
SSM-ASGC	67	40	26	206	158	142
SSM-USC	59	40	29	105	91	88
SSM-UBC	49	30	18	79	52	38
SSM-UXC	53	30	17	71	45	32
SSM-VBC	–	29	17	–	51	38
SSM-VXC	–	30	17	–	43	32
SSM-MB	63	37	22	166	105	71
SSM-WB	49	29	18	79	51	38
SSM-WX	51	29	16	68	43	31
SSM-WBZ	39	27	23	–	–	–
SBNDM4	31	11	7	38	16	11
FHASH2	36	29	24	149	94	61

Table 10. Running times (in units of 10 ms) of algorithms for sets of 200 patterns.

SBNDM4 [11] with 16-bit reads, and FHASH2 [2]. SBNDM4 is an example of a simple and efficient algorithm, and FHASH2 is an advanced algorithm co-authored by the developer of SSM.

From Table 10 one can see that SSM-WBZ and SSM-WX are the best variations of SSM. SSM-WBZ is the fastest on English data for $m = 5$ and 10. The character 'm' was used as a threshold for SSM-WBZ. If one selects a more frequent threshold, this algorithm becomes slightly faster for $m = 5$ and slightly slower for $m = 20$. SSM-WX is the fastest on English data for $m = 20$ and on DNA data. SSM-WBZ and SSM-WX are 47–70% faster than the original SSM. However, these variations are much slower than SBNDM4 which in turn is slower than recent SIMD-based algorithms like EPSM [13] (the times of EPSM are not shown here).

Table 10 confirms that the alternative X is faster than B on DNA data. Note that no SSM variation was faster than FHASH2 for English patterns of five characters.

7 Conclusions

We applied a guard test on comparison-based string matching algorithms. The test compares multiple characters between the pattern and an alignment window before the match loop. The guard test led to notable speed-ups as shown with experiments. The Brute Force algorithm benefits most from the guard test, and it was faster than other comparison-based algorithms of the test set for short patterns $m \leq 16$. In addition, the variation BF8 was faster than any of the algorithms in the SMART repository for $m = 8$ on binary text.

Our experiments show that most of the new variations of SSM are faster than the original SSM. Although the pivot loop is an inspiring tool, we learned that it hardly can lead to the level of SBNDM4 in efficiency. The obvious reason is that the pivot loop makes two separate accesses to the text in a round: the pivot and the base of shift. A typical skip loop accesses only the base of shift which consists of a single character or a q-gram. However, the positive results with variations testing q-grams (the alternatives U, V, and W) support the usefulness of q-gram guards.

References

1. A. M. Al-Ssulami: *Hybrid string matching algorithm with a pivot.* J. Information Science, 41(1) 2015, pp. 82–88.
2. A. M. Al-Ssulami and H. Mathkour: *Faster string matching based on hashing and bit-parallelism.* Inf. Process. Lett., 123 2017, pp. 51–55.
3. A. M. Al-Ssulami, H. Mathkour, and M. A. Arafah: *Efficient string matching algorithm for searching large DNA and binary texts.* Int. J. Semantic Web Inf. Syst., 13(4) 2017, pp. 198–220.
4. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Club Workshop 1999, J. Holub and M. Simánek, eds., Prague, Czech Republic, 1999, pp. 16–28.
5. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.
6. D. Cantone and S. Faro: *"It's economy, stupid!": Searching for a substring with constant extra-space complexity*, in Proceedings of Third International Conference on Fun with algorithms, P. Ferragina and R. Grossi, eds., Tuscany, Italy, 2004, pp. 118–131.
7. D. Cantone and S. Faro: *Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm.* Journal of Automata, Languages and Combinatorics, 10(5/6) 2005, pp. 589–608.
8. D. Cantone and S. Faro: *Improved and self-tuned occurrence heuristics.* J. Discrete Algorithms, 28 2014, pp. 73–84.
9. C. Charras, T. Lecroq, and J. Pehoushek: *A very fast string matching algorithm for small alphabets and long patterns*, in CPM 1998: Combinatorial Pattern Matching, M. Farach-Colton, ed., vol. 1448 of Lecture Notes in Computer Science, Piscataway, New Jersey, USA, 1998, Springer, Berlin, Heidelberg, pp. 55–64.
10. S. Deusdado and P. Carvalho: *GRASPm: An efficient algorithm for exact pattern-matching in genomic sequences.* International journal of bioinformatics research and applications, 5(4) 2009, pp. 385–401.
11. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.
12. S. Faro: *SMART.* `https://github.com/smart-tool/smart`, 2016, Commit cd7464526d41396e11912c6a681eddb965e17f58. Accessed 12.6.2020.
13. 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.

14. S. Faro and T. Lecroq: *Efficient variants of the backward-oracle-matching algorithm.* Int. J. Found. Comput. Sci., 20(6) 2009, pp. 967–984.
15. 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.
16. S. Faro, T. Lecroq, S. Borzì, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2016, pp. 99–111.
17. C. Hancart: *Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte*, PhD thesis, University Paris 7, 1993.
18. R. N. Horspool: *Practical fast searching in strings.* Software: Practice and Experience, 10(6) 1980, pp. 501–506.
19. A. Hudaib, R. Al-Khalid, D. Suleiman, M. Itriq, and A. Al-Anani: *A fast pattern matching algorithm with two sliding windows (TSW).* Journal of Computer Science, 4(5) 2008, pp. 393–401.
20. A. Hume and D. Sunday: *Fast string searching.* Software: Practice and Experience, 21(11) 1991, pp. 1221–1248.
21. P. Kalsi, H. Peltola, and J. Tarhio: *Comparison of exact string matching algorithms for biological sequences*, in Bioinformatics Research and Development, Second International Conference, BIRD 2008, Vienna, Austria, July 7-9, 2008, Proceedings, 2008, pp. 417–426.
22. M. A. Khan: *A transformation for optimizing string-matching algorithms for long patterns.* The Computer Journal, 59(12) 2016, pp. 1749–1759.
23. M. O. Külekci: *An empirical analysis of pattern scan order in pattern matching*, in Proceedings of the World Congress on Engineering, WCE 2007, London, UK, 2-4 July, 2007, 2007, pp. 337–341.
24. T. Lecroq: *Fast exact string matching algorithms.* Information Processing Letters, 102(6) 2007, pp. 229–235.
25. H. Peltola and J. Tarhio: *String matching with lookahead.* Discrete Applied Mathematics, 163 2014, pp. 352–360.
26. T. Raita: *Tuning the Boyer-Moore-Horspool string searching algorithm.* Software: Practice and Experience, 22(10) 1992, pp. 879–884.
27. T. Raita: *On guards and symbol dependencies in substring search.* Software: Practice and Experience, 29(11) 1999, pp. 931–941.
28. A. Sharfuddin and X. Feng: *Improving Boyer-Moore-Horspool using machine-words for comparison*, in Proceedings of the 48th Annual Southeast Regional Conference, 2010, Oxford, MS, USA, April 15-17, 2010, H. C. Cunningham, P. Ruth, and N. A. Kraft, eds., ACM, 2010, pp. 1–5.
29. P. D. Smith: *Experiments with a very fast substring search algorithm.* Software: Practice and Experience, 21(10) 1991, pp. 1065–1074.
30. D. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
31. J. Tarhio, J. Holub, and E. Giaquinta: *Technology beats algorithms (in exact string matching).* Software: Practice and Experience, 47(12) 2017, pp. 1877–1885.
32. J. Tarhio and B. W. Watson: *Tune-up for the Dead-Zone algorithm*, in Proceedings of the Prague Stringology Conference 2020, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2020, pp. 160–167.
33. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of Dead-Zone single keyword pattern matching*, in Combinatorial Algorithms, 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19-21, 2012, Revised Selected Papers, S. Arumugam and W. F. Smyth, eds., vol. 7643 of Lecture Notes in Computer Science, Springer, 2012, pp. 236–248.