



# On the Practical Power of Automata in Pattern Matching

Ora Amir<sup>1,2</sup> · Amihood Amir<sup>1,3</sup> · Aviezri Fraenkel<sup>4</sup> · David Sarné<sup>1</sup>

Received: 12 July 2023 / Accepted: 2 February 2024  
© The Author(s) 2024

## Abstract

Many papers in the intersection of theoretical and applied algorithms show that the simple, asymptotically less efficient algorithm, performs better than the best complex theoretical algorithms on random data or in specialized “real world” applications. This paper considers the Knuth–Morris–Pratt automaton, and shows a counter-intuitive practical result. The classical pattern matching paradigm is that of seeking occurrences of one string—the pattern, in another—the text, where both strings are drawn from an alphabet set  $\Sigma$ . Assuming the text length is  $n$  and the pattern length is  $m$ , this problem can naively be solved in time  $O(nm)$ . In Knuth, Morris and Pratt’s seminal paper of 1977, an automaton, was developed that allows solving this problem in time  $O(n)$  for any alphabet. This automaton, which we will refer to as the *KMP-automaton*, has proven useful in solving many other problems. A notable example is the *parameterized pattern matching* model. In this model, a consistent renaming of symbols from  $\Sigma$  is allowed in a match. The parameterized matching paradigm has proven useful in problems in software engineering, computer vision, and other applications. It has long been believed that for texts where the symbols are uniformly random, the naive algorithm will perform as well as the KMP algorithm. In this paper, we examine the practical efficiency of the KMP algorithm versus the naive algorithm on a randomly generated text. We analyze the time under various parameters, such as alphabet size, pattern length, and the distribution of pattern occurrences in the text. We do this for both the original exact matching problem and parameterized matching. While the folklore wisdom is vindicated by these findings for the *exact matching case*, surprisingly, the KMP algorithm *always* works significantly faster than the naive in the parameterized matching case. We check this hypothesis for DNA texts and image data and observe a similar behavior as in the random text. We also show a very structured exact matching case where the automaton is much more efficient.

**Keywords** Simple algorithms · Uniformly random text · Knuth–Morris–Pratt automaton · Parameterized matching.

---

This article is part of the topical collection “String Processing and Combinatorial Algorithms” guest edited by Simone Faro.

---

✉ Amihood Amir  
amir@cs.biu.ac.il

Ora Amir  
oramir70@gmail.com

Aviezri Fraenkel  
aviezri.fraenkel@weizmann.ac.il

David Sarné  
sarned@cs.biu.ac.il

<sup>1</sup> Department of Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel

<sup>2</sup> Camtek, Migdal HaEmek, Israel

<sup>3</sup> College of Computing, Georgia Tech, Atlanta, GA 30332, USA

<sup>4</sup> Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, 76100 Rehovot, Israel

## Introduction

Algorithms design has two, almost distinct, tracks. The theoretical track devotes itself to ingenious algorithms and data structures on idealized problem versions, that offer efficient asymptotic complexity. The applications track solves real problems with their specialized inputs and the need for concrete fast solutions. These cross purposes may lead to different approaches for the solution to the same problem.

There is some research at the intersection of these two tracks. The nature of this research has, traditionally, two purposes:

1. Experimental tests that show superiority of a simple, asymptotically less efficient algorithm, when applied to random or concrete data (see e.g., [14, 15, 20–23, 25, 30, 33, 36]).

- Experimental tests that try to pinpoint the size or type of data where the theoretically sophisticated algorithm does perform better. [6]

The main goal of this paper is to present a case where the naive algorithm performs worse than the sophisticated algorithm even for small-sized uniformly random data. We also show a few concrete “real world” applications where this is the case. The results were counter-intuitive to us and to many pattern matching researchers. Yet, when we analyzed these results, the simple theoretical explanation was, indeed, extremely convincing. So much so that we all were a bit stupefied as to why our intuition was “wrong” to begin with. The data structure we study is the Knuth–Morris–Pratt automaton, or the KMP automaton [32].

The KMP automaton is one of the most well-known data structures in Computer science. It allows solving the *exact string matching problem* in linear time. The exact string matching problem has input text  $T$  of length  $n$  and pattern  $P$  of length  $m$ , where the strings are composed of symbols from a given alphabet  $\Sigma$ . The output is all text locations where the pattern occurs in the text. The naive way of solving the exact string matching problem takes time  $O(nm)$ . This can be achieved by sliding the pattern to start at every text location, and comparing each of its elements to the corresponding text symbol. Using the KMP automaton, this problem can be solved in time  $O(n)$ . In fact, analysis of the algorithm shows that at most  $2n$  comparisons need to be done.

It has long been known in the folklore<sup>1</sup> that if the text is composed of uniformly random alphabet symbols, the naive algorithm’s time is also linear. This belief is bolstered by the fact that the naive algorithm’s mean number of comparisons for text and pattern over a binary alphabet is bounded by

$$n \left( 1 + \sum_{i=1}^m \frac{i}{2^i} \right) \text{ which is bounded by } 2n \text{ comparisons.}$$

The number of comparisons in the KMP algorithm is also bounded by  $2n$ . However, because control in the naive algorithm is much simpler, then it may be practically faster than the KMP algorithm.

The last few decades have prompted the evolution of pattern matching from a combinatorial solution of the exact string matching problem [24, 32] to an area concerned with approximate matching of various relationships motivated by computational molecular biology, computer vision, and complex searches in digitized and distributed multimedia

libraries [8, 19]. An important type of non-exact matching is the *parameterized matching* problem which was introduced by Baker [11, 12]. Her main motivation lay in software maintenance, where program fragments are to be considered “identical” even if variable names are different. Therefore, strings under this model are comprised of symbols from two disjoint sets  $\Sigma$  and  $\Pi$  containing *fixed symbols* and *parameter symbols*, respectively. In this paradigm, one seeks *parameterized occurrences*, i.e., exact occurrences up to renaming of the parameter symbols of the pattern string in the respective text location. This renaming is a bijection  $b : \Pi \rightarrow \Pi$ . An optimal algorithm for exact parameterized matching appeared in [5]. It makes use of the KMP automaton for a linear-time solution over fixed finite alphabet  $\Sigma$ . Approximate parameterized matching was investigated in [9, 11, 27]. Idury and Schäffer [29] considered multiple matching of parameterized patterns.

Parameterized matching has proven useful in other contexts as well. An interesting problem is searching for images (e.g., [3, 10, 38]). Assume, for example, that we are seeking a given icon in any possible color map. If the colors were fixed, then this is exact two-dimensional pattern matching [2]. However, if the color map is different the exact matching algorithm would not find the pattern. Parameterized two dimensional search is precisely what is needed. If, in addition, one is also willing to lose resolution, then a two dimensional function matching search should be used, where the renaming function is not necessarily a bijection [1, 7]. Another degenerate parameterized condition appears in DNA matching. Because of the base pair bonding, exchanging A with T and C with G, in both text and pattern, produces a match [28].

Parameterized matching can also be naively done in time  $O(nm)$ . Based on our intuition for exact matching, it is expected that here, too, the naive algorithm is competitive with the KMP automaton-based algorithm of [5] in a randomly generated text.

In this paper, we investigate the practical efficiency of the automaton-based algorithm versus the naive algorithm both in exact and parameterized matching. We consider the following parameters: pattern length, alphabet size, and distribution of pattern occurrences in the text. Our findings are that, indeed, the naive algorithm is faster than the automaton algorithm in practically all settings of the *exact matching problem*. However, it was counter-intuitive to see that the automaton algorithm is *always* more effective than the naive algorithm for *parameterized matching* over randomly generated texts. We analyze the reason for this difference.

Having established that the randomness of the text is what made the naive algorithm so efficient for exact matching. We, therefore, ran the comparison in a very structured artificial text, and the automaton algorithm was a clear winner.

Having understood the practical behavior of the naive versus automaton algorithm over randomly generated texts,

<sup>1</sup> The second author heard this for the first time from Uzi Vishkin in 1985. Since then this belief has been mentioned, in many occasions, by various researchers in the community.

we were curious if there were “real” texts with a similar phenomenon. We did two case studies. We ran the same experiments over DNA texts and observed a similar behavior as that of a randomly generated text. We also ran these experiments over image data and observed the same results.

### Problem Definition

We begin with basic definitions and notation generally following [17].

Let  $S = s_1s_2 \dots s_n$  be a string of length  $|S| = n$  over an ordered alphabet  $\Sigma$ . By  $\epsilon$  we denote an empty string. For two positions  $i$  and  $j$  on  $S$ , we denote by  $S[i..j] = s_i \dots s_j$  the factor (sometimes called *substring*) of  $S$  that begins at position  $i$  and ends at position  $j$  (it equals  $\epsilon$  if  $j < i$ ). A *prefix* of  $S$  is a factor that begins at position 1 ( $S[1..j]$ ) and a *suffix* is a factor that ends at position  $n$  ( $S[i..n]$ ).

The exact string matching problem is defined as follows:

**Definition 1 (Exact String Matching)** Let  $\Sigma$  be an alphabet,  $T = t_1 \dots t_n$  the text and  $P = p_1 \dots p_m$  the pattern,  $t_i, p_j \in \Sigma$ ,  $i = 1, \dots, n; j = 1, \dots, m$ . The exact string matching problem is:

input: text  $T$  and pattern  $P$ .

output: All indices  $j$ ,  $j \in \{1, \dots, n - m + 1\}$  such that

$$t_{j+i-1} = p_i, \text{ for } i = 1, \dots, m$$

We simplify Baker’s definition of parameterized pattern matching.

**Definition 2 (Parameterized-Matching)** Let  $\Sigma$ ,  $T$  and  $P$  be as in Definition 1. We say that  $P$  parameterize-matches or simply  $p$ -matches  $T$  in location  $j$  if  $p_i \cong t_{j+i-1}$ ,  $i = 1, \dots, m$ , where  $p_i \cong t_j$  if and only if the following condition holds:

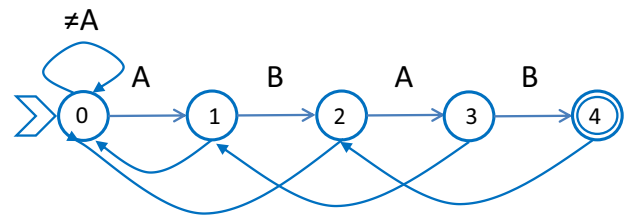
for every  $k = 1, \dots, i - 1$ ,  $p_i = p_{i-k}$  if and only if

$$t_j = t_{j-k}.$$

The  $p$ -matching problem is to determine all  $p$ -matches of  $P$  in  $T$ .

If two strings  $S_1$  and  $S_2$  have the same length  $m$  then they are said to parameterize-match or simply  $p$ -match if  $s_{1_i} \cong s_{2_i}$  for all  $i \in \{1, \dots, m\}$ .

Intuitively, the matching relation  $\cong$  captures the notion of one-to-one mapping between the alphabet symbols. Specifically, the condition in the definition of  $\cong$  ensures that there exists a bijection between the symbols from  $\Sigma$  in the pattern and those in the overlapping text, when they  $p$ -match. The relation  $\cong$  has been defined by [5] in a manner suitable for computing the bijection.



$P=ABAB$   
 $|P|=4$   
 $\Sigma=\{A,B\}$   
 $\delta_s =$  all arrows with a symbol above them.  $\delta_f =$  all other arrows.

Fig. 1 Automaton example

**Example:** The string  $ABABCCBA$  parameterize matches the string  $XYXYZZYX$ . The reason is that if we consider the bijection  $\beta : \{A, B, C\} \rightarrow \{X, Y, Z\}$  defined by  $A \xrightarrow{\beta} X$ ,  $B \xrightarrow{\beta} Y$ ,  $C \xrightarrow{\beta} Z$ , then we get  $\beta(ABABCCBA) = XYXYZZYX$ . This explains the requirement in Def. 2, where two symbols match if they also match in all their previous occurrences.

Of course, the alphabet bijection need not be as extreme as bijection  $\beta$  above. String  $ABABC-CBA$  also parameterize matches  $BABACCAB$ , because of bijection  $\gamma : \{A, B, C\} \rightarrow \{A, B, C\}$  defined as:  $A \xrightarrow{\gamma} B$ ,  $B \xrightarrow{\gamma} A$ ,  $C \xrightarrow{\gamma} C$ .

For completeness, we define the *KMP automaton*.

**Definition 3** Let  $P = p_1 \dots p_m$  be a string over alphabet  $\Sigma$ . The *KMP automaton* of  $P$  is a 5-tuple  $(Q, \Sigma, \delta_s, \delta_f, q_0, q_a)$ , where  $Q = \{0, \dots, m\}$  is the set of states,  $\Sigma$  is the alphabet,  $\delta_s : Q \rightarrow Q$  is the success function,  $\delta_f : Q \rightarrow Q$  is the failure function,  $q_0 = 0$  is the start state and  $q_a = m$  is the accepting state.

The success function is defined as follows:

$$\delta_s(i) = i + 1, i = 0, \dots, m - 1 \text{ and}$$

$$\delta_s(0) = 0$$

The failure function is defined as follows:

Denote by  $\ell(S)$  the length of the longest proper prefix of string  $S$  (i.e., excluding the entire string  $S$ ) which is also a suffix of  $S$ .

$$\delta_f(i) = \ell(P[1..i]), \text{ for } i = 1, \dots, m.$$

For an example of the KMP automaton see Fig. 1.

**Theorem 1 [32]** The KMP automaton can be constructed in time  $O(m)$ .

## The Exact String Matching Problem

The *Knuth–Morris–Pratt (KMP) search algorithm* uses the KMP automaton in the following manner:

```

return not equal
end
Compare( $p_i, p_j$ )

```

### Variables:

$pointer_t$  points to indices in the text.  $pointer_p$  points to indices in the pattern.

### Initialization:

set pointer  $pointer_t$  to 1. set pointer  $pointer_p$  to 0.

### Main Loop:

```

While  $pointer_t \leq n - m + 1$  do:
  If  $t_{pointer_t} = \delta_s(pointer_p)$  then do:
     $pointer_t \leftarrow pointer_t + 1$ 
     $pointer_p \leftarrow \delta_f(pointer_p)$ 
    If  $pointer_p = m - 1$  then do:
      output "pattern occurrence ends in text location  $pointer_t$ ".
       $pointer_p \leftarrow \delta_f(m)$ 
    enddo
  enddo

  else ( $t_{pointer_t} \neq \delta_s(pointer_p)$ ) do:
    if  $pointer_p = 0$  then  $pointer_t \leftarrow pointer_t + 1$ 
    else  $pointer_p \leftarrow \delta_f(pointer_p)$ 
  enddo
  go to beginning of while loop
endwhile

```

**Theorem 2** [32] *The time for the KMP search algorithm is  $O(n)$ . In fact, it does not exceed  $2n$  comparisons.*

```

if ( $A[i] = i$  or  $i - A[i] \geq j$ ) and  $p_j \neq p_1, \dots, p_{j-1}$  then
return equal
if  $i - A[i] < j$  and  $p_j = p_{j-i+A[i]}$  then return equal
return not equal
end

```

## The Parameterized Matching Problem

Amir, Farach, and Muthukrishnan [5] achieved an optimal time algorithm for parameterized string matching by a modification of the KMP algorithm. In fact, the algorithm is exactly the KMP algorithm, however, every equality comparison " $x = y$ " is replaced by " $x \cong y$ " as defined as follows.

### Implementation of " $x \cong y$ "

Construct table  $A[1], \dots, A[m]$  where  $A[i] =$  the largest  $k$ ,  $1 \leq k < i$ , such that  $p_k = p_i$ . If no such  $k$  exists then  $A[i] = i$ .

The following subroutines compute " $p_i \cong t_j$ " for  $j \geq i$ , and " $p_i \cong p_j$ " for  $j \leq i$ .

Compare( $p_i, t_j$ )

if  $A[i] = i$  and  $t_j \neq t_{j-1}, \dots, t_{j-i+1}$  then return *equal*

if  $A[i] \neq i$  and  $t_j = t_{j-i+A[i]}$  then return *equal*

**Theorem 3** [5] *The  $p$ -matching problem can be solved in  $O(n \log \sigma)$  time, where  $\sigma = \min(m, |\Sigma|)$ .*

**Proof** The table  $A$  can be constructed in  $O(m \log \sigma)$  time as follows: scan the pattern left to right keeping track of the distinct symbols from  $\Sigma$  in the pattern in a balanced tree, along with the last occurrence of each such symbol in the portion of the pattern scanned thus far. When the symbol at location  $i$  is scanned, look up this symbol in the tree for the immediately preceding occurrence; that gives  $A[i]$ .

Compare can clearly be implemented in time  $O(\log \sigma)$ . For the case  $A[i] \neq i$ , the comparison can be done in time  $O(1)$ . When scanning the text from left to right, keep the last  $m$  symbols in a balanced tree. The check  $t_j \neq t_{j-1}, \dots, t_{j-i+1}$  in Compare( $p_i, t_j$ ) can be performed in  $O(\log \sigma)$  time using this information. Similarly, Compare( $p_i, p_j$ ) can be performed

using  $A[i]$ . Therefore, the automaton construction in KMP algorithm with every equality comparison “ $x = y$ ” replaced by “ $x \cong y$ ” takes time  $O(m \log \sigma)$  and the text scanning takes time  $O(n \log \sigma)$ , giving a total of  $O(n \log \sigma)$  time.

As for the algorithm’s correctness, Amir, Farach and Muthukrishnan showed that the failure link in automaton node  $i$  produces the largest prefix of  $p_1 \dots p_i$  that  $p$ -matches the suffix of  $p_1 \dots p_i$ .  $\square$

## Our Experiments

Our implementation was written in C++. The platform was Dell latitude 7490 with intel core i7 - 8650U, 32 GB RAM, with 8 MB cache. The running time was computed using the chrono high-resolution clock. The time for a tick of that clock is one nanosecond. The random strings were generated using the random Python package.

We implemented the naive algorithm for exact string matching and for parameterized matching. The same code was used for both, except for the implementation of the equivalence relation for parameterized matching, as described above. This required implementing the  $A$  array. We also implemented the KMP algorithm for exact string matching, and used the same algorithm for parameterized matching. The only difference was the implementation of the equivalence parameterized matching relation.

The text length  $n$  was 1,000,000 symbols. Theoretically, since both the automaton and naive algorithm are sequential and only consider a window of the pattern length, it would have been sufficient to run the experiment on a text of size twice the pattern [4]. However, for the sake of measurement resolution we opted for a large text. Yet the size of 1,000,000 comfortably fits in the cache, and thus we avoid the caching issue. In general, any searching algorithm for *patterns* of length less than 4MB would fit in the cache if appropriately constructed in the manner of [4]. Therefore our decision gives as accurate a solution as possible.

We ran patterns of lengths  $m = 32, 64, 128, 256, 512$ , and 1024. The alphabet sizes tested were  $|\Sigma| = 2, 4, 6, 8, 10, 20, 40, 80, 160, 320$ . For each size, 10 tests were run, for a total of 600 tests.

**Methodology:** We generated a uniformly random text of length 1,000,000. If the pattern would also be randomly generated, then it would be unlikely to appear in the text. However, when seeking a pattern in the text, one assumes that the pattern occurs in the text. An example would be searching for a sequence in the DNA. When seeking a sequence, one expects to find it but just does not know where. Additionally, we considered the common case where one does not expect many occurrences

of the pattern in the text. Consequently, we planted 100 occurrences of the pattern in the text at uniformly random locations. The final text length was always 1,000,000. The reason for inserting 100 pattern occurrences is the following. We do not expect many occurrences, and a 100 occurrences in a million-length text means that less than 0.1% of the text has pattern occurrences. On the other hand, it is sufficient to introduce the option of actually following all elements of the pattern 100 times. This would make a difference in both algorithms. They would both work faster if there were no occurrences at all. There are many possible ways of planting 100 copies of the pattern at random locations. Our method was randomly generating the indices of the planted copies. In order to avoid overlap, we generated 100 indices in a range from 1 to  $1,000,000 - L$ , where  $L$  is the total length of planted patterns. We inserted a pattern copy at each of these indices, starting from the smallest. For each such insertion, we added the pattern length to each of the remaining indices.

We also implemented a variation where half of the pattern occurrences were in the last quarter of the text. For each alphabet size and pattern length we generated 10 tests and considered the average result of all 10 tests. It should be noted that from a theoretical point of view, the location of the pattern should not make a difference. We tested the different options in order to verify that this is, indeed, the case.

The software code can be found at <https://github.com/Ora70/Automaton>.

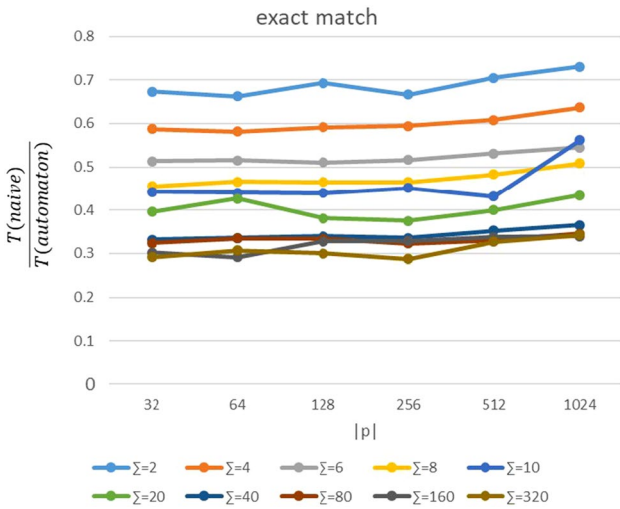
## Exact Matching

### Results

Tables 3 and 4 in the Appendix show the alphabet size, the pattern length, the average of the running times of the naive algorithm for the 10 tests, the average of the running time of the KMP algorithm for the 10 tests, and the ratio of the naive algorithm running time over the KMP algorithm running time. Any ratio value *below* 1 means that the *naive algorithm is faster*. A *small* value indicates a *better* performance of the naive algorithm. Any value above 1 indicates that the KMP algorithm is faster than the naive algorithm. The larger the number, the better the performance.

To enable a clearer understanding of the results, we present them below in graph form. The following graphs show the results of our tests for the different pattern lengths. In Figs. 2 and 3, the  $x$ -axis is the pattern size. The  $y$ -axis is the ratio of the naive algorithm running time to the KMP algorithm running time. The different colors depict alphabet sizes. In Fig. 2, the patterns were inserted at random, whereas in Fig. 4 the patterns appear at the last half of the text.



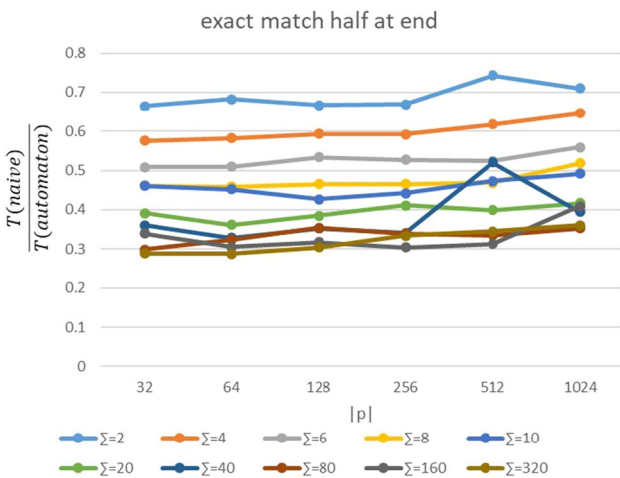


**Fig. 2** Performance in the Exact Matching case, pattern occurrences distributed uniformly random

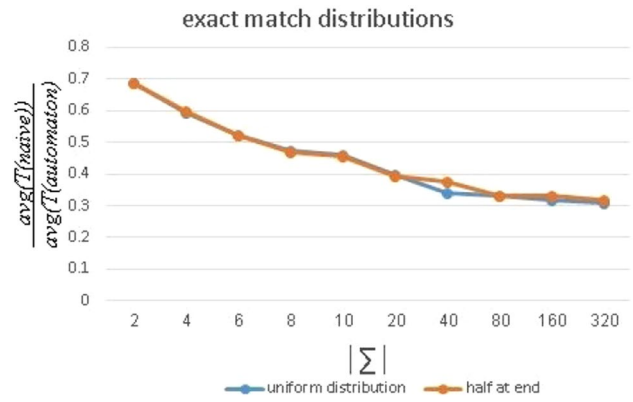
To better see the effect of the pattern distribution in the text, Fig. 4 maps, *on the same graph*, both cases. In this graph, the *x-axis* is the average running time of *all* pattern lengths per alphabet size, and the *y-axis* is the ratio of the naive algorithm running time to the KMP algorithm running time. The results of the uniformly random distribution are mapped in one color, and the results of all pattern occurrences in the last half of the text are mapped in another.

We note the following phenomena:

1. The naive algorithm *performs better* than the automaton algorithm. Of the 600 tests we ran, there were only 3 instances where the KMP algorithm performed better



**Fig. 3** Performance in the Exact Matching case, pattern occurrences congregated at end of text



**Fig. 4** Comparison of average performance of uniform pattern distribution versus pattern occurrences congregated at end of text

than the naive, and all were subsumed by the average. In the vast majority of cases the naive algorithm was superior by far.

2. The naive algorithm performs relatively better for larger alphabets.
3. For a fixed alphabet size, there is a slight increase in the naive/KMP ratio, as the pattern length increases.
4. The distribution of the pattern occurrences in the text does not seem to make a change in performance.

An analysis of these implementation behaviors appears in the next subsection.

### Analysis

We analyze all four results noted above.

**Better Performance of the Naive Algorithm** We have seen that the mean number of comparisons of the naive algorithm for binary alphabets is bounded by

$$n \left( 1 + \sum_{i=1}^m \frac{i}{2^i} \right) \text{ which is bounded by } 2n \text{ comparisons.}$$

The running time of the KMP algorithm is also bounded by  $O(2n)$ . However, the control of the KMP algorithm is more complex than that of the naive algorithm, which would indicate a constant ratio in favor of the naive algorithm. However, when the KMP algorithm encounters a mismatch it follows the failure link, which avoids the need to re-check a larger substring. Thus, for longer length patterns, where there are more possibilities of following the failure links for longer distances, there is a lessening advantage of the naive algorithm.

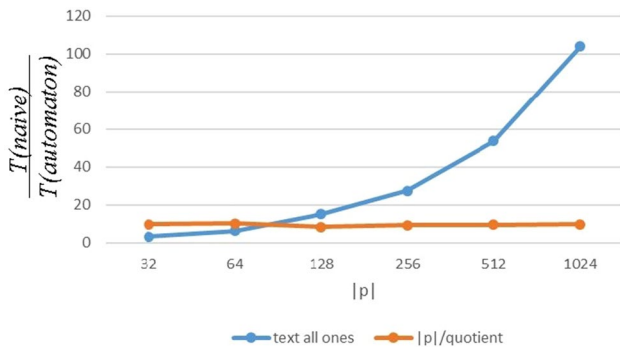


Fig. 5 Performance in the exact matching case, periodic text

**Better Performance of the Naive Algorithm for Larger Alphabets** This is fairly clear when we realize that the mean performance of the naive algorithm for alphabet of size  $k$  is:

$$n \left( 1 + \sum_{i=1}^m \frac{i}{k^i} \right) = n + n \frac{k}{(k-1)^2} \text{ comparisons.}$$

This is clearly decreasing the larger the alphabet size. However, the repetitive traversal of the failure link, even in cases where there is no equality in the comparison check, will still raise the relative running time of the KMP algorithm. Here too, the longer the pattern length, the more failure link traversals of the KMP, and thus less overall comparisons, which slightly decreases the advantage of the naive algorithm.

**The Distribution of Pattern Occurrences in the Text** If the pattern is not periodic, and if the patterns are not too frequent in the text, then there will be at most one pattern in a text substring of length  $2m$ . In these circumstances, there is really no effect to the distribution of the pattern in the text. We would expect a difference if the pattern is long with a small period. Indeed, an extreme such case is tested in Sect. 5.1.3.

**A Very Structured Example**

All previous analyses point to the conviction that the more times a prefix of the pattern appears in the text, and the more periodic the pattern, the better will be the performance of the KMP algorithm. The most extreme case would be of text  $A^n$  ( $A$  concatenated  $n$  times), and pattern  $A^{m-1}B$ . Indeed the results of this case appear in Fig. 5.

Theoretical analysis of the naive algorithm predicts that we will have  $nm$  comparisons, where  $n$  is the text length and  $m$  is the pattern length. The KMP algorithm will have  $2n$  comparisons, for any pattern length. Thus the ratio  $q$  of naive to KMP will be  $O(\frac{m}{2})$ . In fact, when we plot  $\frac{m}{q}$  we get twice

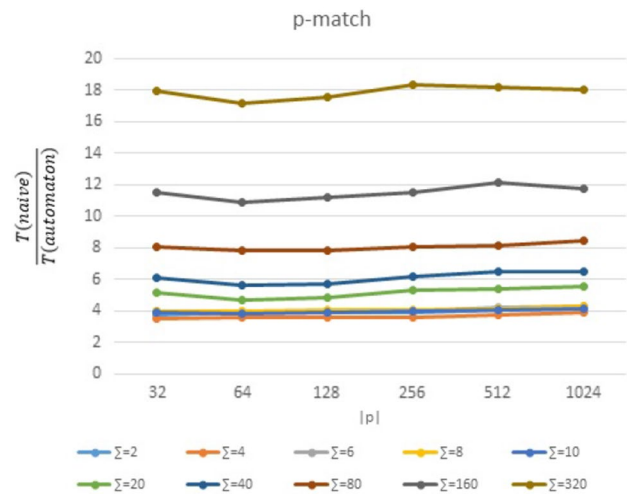


Fig. 6 Performance in the parameterized matching case, pattern occurrences distributed uniformly random

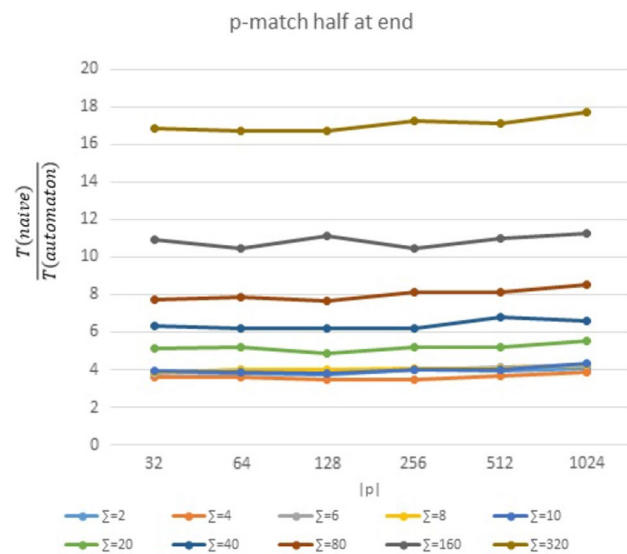


Fig. 7 Performance in the parameterized matching case, pattern occurrences congregated at end of text

the cost of the control of the KMP algorithm. This is shown in Fig. 5 to be 5.

**Parameterized Matching**

**Results**

The exact matching results behaved roughly in the manner we expected. The surprise came in the parameterized matching case. Below are the results of our tests. It should be remarked that, as in the classical exact matching case, we compared the naive algorithm with a straight-forward

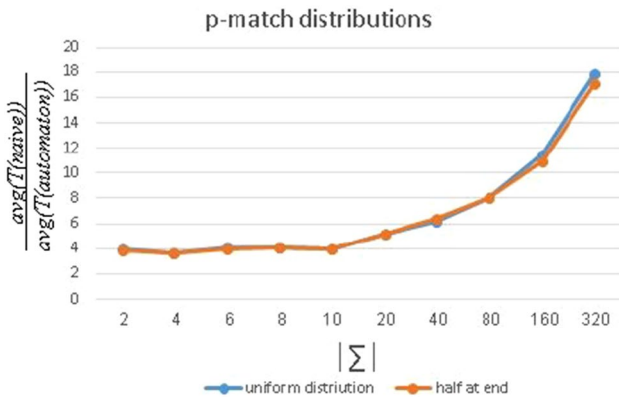


Fig. 8 Comparison of average performance of uniform pattern distribution versus pattern occurrences congregated at end of text

automaton implementation. There exist sophisticated parameterized matching implementations that are sublinear on random texts [13, 26]. Yet for our tests we used the straightforward automaton-based algorithm, and it was sufficient for achieving a clear-cut superiority over the naive algorithm. As in the exact matching case, the tables show the alphabet size, the pattern length, the average of the running times of the naive algorithm for the 10 tests, the average of the running time of the automaton-based algorithm for the 10 tests, and the ratio  $q$  of the naive algorithm running time over the automaton-based algorithm running time. Any ratio value above 1 means that the automaton-based algorithm is faster. A large value indicates a better performance of the automaton-based algorithm.

The following graphs show the results of our tests for the different pattern lengths. The  $x$ -axis is the pattern size. The  $y$ -axis is the ratio of the naive algorithm running time to the automaton-based algorithm running time. The different colors depict alphabet sizes. To better see the effect of the pattern distribution in the text, we also map, on the same graph, both cases. In this graph, the  $x$ -axis is the average running time of all pattern lengths per alphabet size, and the  $y$ -axis is the ratio of the naive algorithm running time to the automaton-based algorithm running time. The results of the uniformly random distribution are mapped in one color, and the results of all pattern occurrences in the last half of the text are mapped in another.

The parameterized matching results appear in Tables 5 and 6 in the appendix. Figures 6 and 7 map the results of the parameterized matching comparisons for the case where the patterns were inserted at random versus the case where the patterns appear at the last half of the text. In Fig. 8, we map at the same graph the average results of both the cases

where the patterns appear at the text uniformly at random, and where the patterns appear at the last half of the text.

The results are very different from the exact matching case. We note the following phenomena:

1. The automaton-based algorithm always performs significantly better than the naive algorithm.
2. The automaton-based algorithm performs relatively better for larger alphabets.
3. For a fixed alphabet size, the pattern length does not seem to make much difference.
4. The distribution of the pattern occurrences in the text does not seem to make a change in performance.

An analysis of these implementation behaviors and an explanation of the seemingly opposite results from the exact matching case appear in the next subsection.

### Analysis

We analyze all four results noted above.

**Better Performance of the Automaton-based Algorithm** We have established that the mean number of comparisons for the naive algorithm in size  $k$  alphabet is

$$n \left( 1 + \sum_{i=1}^m \frac{i}{k^i} \right) = n + n \frac{k}{(k-1)^2} \text{ comparisons.}$$

However, when it comes to parameterized matching, any order of the alphabet symbols is a match, thus if one consider a pattern prefix of length  $i$  where all  $k$  alphabet symbols occur, then the mean number of comparisons is to be multiplied by  $k!$ . We explain this at greater length:

When doing exact matching, the probability that the first pattern symbol equals the text symbol is  $\frac{1}{k}$ . However, in parameterized matching, the first element always matches the text element, so that probability needs to be multiplied by  $k$ . This indeed gives us probability 1 for matching the first pattern symbol. As long as the first symbol appears in a run, the probability of matching it in the text is  $\frac{1}{k}$  per location. But once a new alphabet symbol appears in the pattern, the probability of a match is  $\frac{(k-1)}{k}$ . In general, when the  $j$ th alphabet symbol appears for the first time in the pattern, its probability of matching the text is  $\frac{(k+1-j)}{k}$ . All other pattern symbols (i.e., those that have appeared previously) have matching probability  $\frac{1}{k}$ .

This explains why the probability of matching a prefix of length  $j$  where all symbols of the alphabet occur is



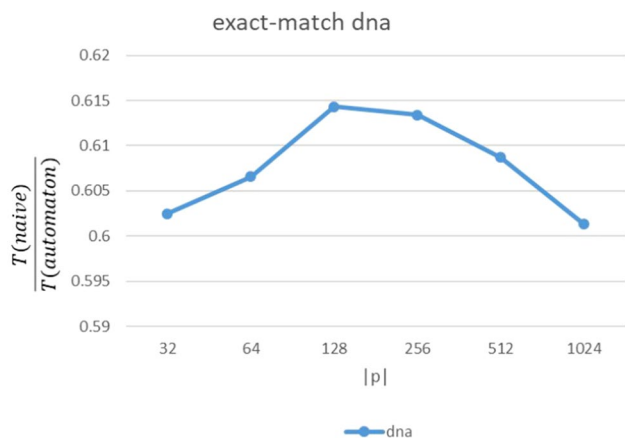


Fig. 9 Performance in the exact matching case on DNA sequences

$$k! \sum_{i=1}^j \frac{i}{k^i}$$

By the discussion above, the probability of matching a prefix of length  $j$  where  $\ell_j$  different alphabet symbols occur is

$$P_{\ell_j} \sum_{i=1}^j \frac{i}{k^i}$$

, where

$$P_{\ell_j} = \prod_{r=0}^{\ell_j-1} k - r$$

This bounds the mean number of comparisons for a size  $k$  alphabet by:

$$n(1 + \sum_{i=1}^m \frac{iP_{\ell_i}}{k^i})$$

Therefore, for size 2 alphabet we get  $3n$  comparisons, and the number rises exponentially with the alphabet size. Also, the automaton-based algorithms is constant at  $2n$  comparisons. Even for a size 2 alphabet, the number of comparisons in the naive algorithm is greater by 50% than in the automaton-based algorithm. Note, also, that because of the need to find the last parameterized match, the control mechanism even of the naive algorithm, is more complex. This results in a superior performance of the automaton-based algorithm even for small alphabets.

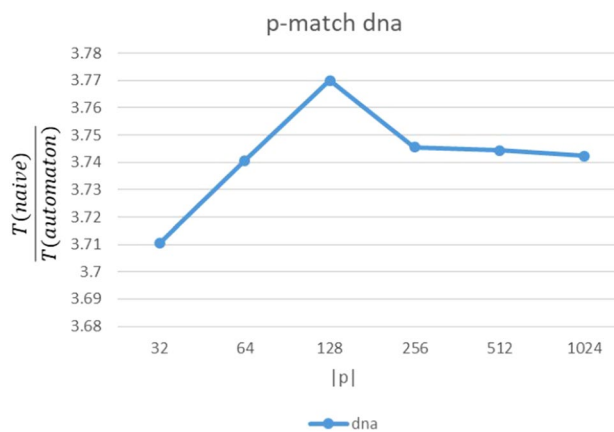


Fig. 10 Performance in the parameterized matching case on DNA Sequences

Of course, the larger the alphabet, the better the performance of the automaton-based algorithm.

**Pattern Length** The pattern length does not play a role in the automaton-based algorithm, where the number of comparisons is always bounded by  $2n$ . In the naive case, the multiplication of the factorial of the alphabet size is so overwhelming that it dominates the pattern length. For example, note that for an extremely large alphabet, there would be a leading prefix of different alphabet symbols. That prefix will always be traversed by the naive algorithm. The larger the alphabet, the longer will be the mean length of that prefix.

**Pattern Distribution** As in the exact matching case, for a non-periodic pattern that does not appear too many times, the distribution of occurrences will have no effect on the complexity.

### DNA Data

Having understood the behavior of the naive algorithm and the automaton-based algorithm in randomly generated texts, the natural question is are there any “real” texts for which the naive algorithm performs better than the automaton-based algorithm.

We performed the same experiments on DNA data. The experimental setting was identical to that of the randomly generated texts with the following differences:

1. The DNA of the fruit fly, *Drosophila melanogaster* is 143.7 MB long. We extracted 10 subsequences of length 1,000,000 each, as FASTA data from the NIH National Library of Medicine, National Center for Biotechnol-



Fig. 11 *lena*

ogy Information. We ran a test for each of the six pattern lengths  $\ell = 32, 64, 128, 256, 512, 1024$ , on each of the 10 DNA subsequences, and noted the average. In order to be as close to “real life” as possible, we assume that only one copy of the pattern occurs in the text, thus the pattern was chosen to be the first  $\ell$  symbols in the sequence. This also has the advantage of having a pattern with a “similar” structure to the text.

2. The alphabet size is 4, due to the four bases in DNA sequence.

Figures 9 and 10 show the ratio between the average running time of the naive algorithm and the automaton based algorithm. As in the uniformly random text we see that for the exact matching case the ratio is less than 1, i.e., the naive algorithm is faster, whereas in the parameterized matching case, the ratio is more than 1, indicating that the automaton based algorithm is faster.

**Table 1** Implementation results—exact matching—Lena

Patt. length	Naive	KMP	$\frac{\text{Naive}}{\text{KMP}}$
32	77.4	226.3	0.3391
64	71.1	225.9	0.3391
128	71.3	223.3	0.3193
256	71.8	223.8	0.3207
512	71.1	226.3	0.3145
1024	70.9	226	0.3139

**Table 2** Implementation results—parameterized matching—lena

Patt. length	Naive	KMP	$\frac{\text{Naive}}{\text{KMP}}$
32	3857.0	666.8	5.8601
64	3728.6	674.3	5.6162
128	3777.6	696.1	5.5168
256	3909.2	697.1	5.6889
512	3848.8	699.9	5.5504
1024	3936.9	700.3	5.6946

## Image Data

For our case study we used the famous “lena” image (see Fig. 11), that is a historic benchmark for many computer vision algorithms. We converted the lena image to a gray level image, and did the search on the pixel level. The conversion was done via the python code at <https://colab.research.google.com/drive/1JLCezhduNtCqYO4nMTKLIwi0AaFejsZW?usp=sharing/>.

The text size was 65536. The pattern sizes are the same six: 32, 64, 128, 256, 512, and 1024. For each size, 10 different patterns were run, chosen from the text, and the average running time was used.

The results can be seen in Tables 1 and 2 below. They are similar to the randomized data and DNA data cases - the automaton is significantly more efficient in parameterized matching, while the naive algorithm is faster in exact matching.

## Conclusions

The folk wisdom has been that simple algorithms generally outperform sophisticated algorithms over uniformly random data. In particular, the naive string matching algorithm will outperform the automaton-based algorithm for uniformly random texts. Indeed this turns out to be the case for *exact matching*. This study shows that this is not the case for *parameterized matching*, where the automaton-based algorithm *always outperforms* the naive algorithm. This advantage is clear and is impressively better the larger the alphabets. The study also shows that the automaton algorithm is clearly superior for searches over DNA data, and image data.

The conclusion to take away from this study is that one should not automatically assume that the naive algorithm is better. In string matching, the matching relation should be analyzed. There are various matchings for which an automaton-based algorithm exists. We considered here parameterized matching, but other matchings, such as ordered

matching [16, 18, 31], or Cartesian tree matching [34, 35, 37], can also be solved by automaton-based methods. In a practical application it is worthwhile spending some time considering the type of matching one is using. It may turn out to be that the automaton-based algorithm will perform significantly better than the naive, even for uniformly random texts. Alternately, even nonuniformly random data may be such that the naive algorithm performs better than the automaton based algorithm for exact matching.

An open problem is to compare the search time in DNA data to the search time in uniformly random data. While it

is clear that DNA data are not uniformly random, it would be interesting to devise an experimental setting to compare search efficiency in both types of strings. A similar study for image data would be interesting.

### Appendix

see Tables 3, 4, 5 and 6.

**Table 3** Implementation results—exact matching, patterns uniformly distributed

$ \Sigma $	Patt. length	Naive	KMP	$\frac{\text{Naive}}{\text{KMP}}$	$ \Sigma $	Patt. length	Naive	KMP	$\frac{\text{Naive}}{\text{KMP}}$		
2	32	4514.1	6712.5	0.6729	4	32	3174.2	5409.9	0.5879		
	64	4449.3	6727.8	0.6623		64	3167.8	5428.3	0.5818		
	128	4697.3	6764.3	0.693		128	3136.8	5293.0	0.5917		
	256	4522.9	6814.2	0.6666		256	3109.7	5228.2	0.5942		
	512	4764.7	6734.7	0.7051		512	3108.8	5110.5	0.608		
	1024	4521.4	6188.7	0.7304		1024	3141.1	4928.7	0.6368		
	32	2225.1	4331.2	0.5139		8	32	1771.8	3903.4	0.4553	
	64	2199.2	4263.2	0.5157			64	1794.5	3852.4	0.4659	
	128	2180.9	4270.6	0.5108	128		1764.0	3789.7	0.4654		
	256	2169.2	4201.4	0.5163	256		1766.5	3798.4	0.4652		
	512	2193.2	4128.4	0.5314	512		1771.9	3670.6	0.4827		
	1024	2238.7	4110.1	0.5455	1024		1827.3	3596.6	0.5085		
	10	32	1593.1	3598.9	0.4427		20	32	1312.0	3309.2	0.396
		64	1578.3	3586.4	0.44			64	1428.6	3297.7	0.4269
		128	1564.8	3563.9	0.4391	128		1252.7	3264.9	0.3817	
		256	1594.5	3531.6	0.4516	256		1187.4	3161.3	0.375	
512		1554.3	3626.0	0.4317	512	1281.7		3166.8	0.4		
1024		1892.5	3380.0	0.5619	1024	1274.6		2923.1	0.4347		
40	32	943.9	2846.7	0.3316	80	32	898.1	2758.7	0.3242		
	64	964.3	2869.3	0.3358		64	938.4	2777.9	0.335		
	128	972.5	2852.5	0.3401		128	946.7	2824.5	0.3336		
	256	952.6	2835.3	0.3363		256	875.9	2709.0	0.323		
	512	975.4	2769.0	0.3523		512	875.8	2653.9	0.3302		
	1024	970.5	2655.4	0.3655		1024	899.6	2605.0	0.346		
160	32	810.9	2686.1	0.302	320	32	790.3	2712.0	0.2916		
	64	794.0	2733.1	0.2918		64	833.4	2711.1	0.3074		
	128	922.2	2771.1	0.3281		128	803.3	2676.3	0.3005		
	256	899.2	2700.6	0.3285		256	785.2	2743.0	0.2877		
	512	897.8	2635.6	0.3374		512	878.5	2690.4	0.3269		
	1024	861.6	2534.9	0.3399		1024	883.8	2563.6	0.3427		

**Table 4** Implementation results—exact matching, patterns at end

$ \Sigma $	Patt. length	Naive	KMP	$\frac{\text{Naive}}{\text{KMP}}$	$ \Sigma $	Patt. length	Naive	KMP	$\frac{\text{Naive}}{\text{KMP}}$
2	32	4613.3	6931.1	0.6649	4	32	3091.7	5362.9	0.5759
	64	4570.1	6695.7	0.6824		64	3203.2	5499.5	0.5819
	128	4462.8	6702.2	0.667		128	3190.4	5373.6	0.5933
	256	4441.5	6644.9	0.6692		256	3200.3	5413.1	0.5924
	512	4786.4	6441.1	0.744		512	3305.2	5340.0	0.6176
	1024	4493.8	6360.6	0.7105		1024	3322.4	5125.8	0.6469
6	32	2374.7	4638.6	0.509	8	32	1836.3	3978.1	0.4616
	64	2336.6	4586.8	0.5093		64	1804.2	3930.2	0.4589
	128	2467.1	4597.0	0.534		128	1816.9	3908.6	0.465
	256	2350.4	4453.1	0.5274		256	1802.8	3875.2	0.4655
	512	2306.2	4447.2	0.5243		512	1792.0	3832.8	0.4684
	1024	2411.2	4302.9	0.5597		1024	1889.1	3640.7	0.5183
10	32	1741.8	3762.0	0.4608	20	32	1242.4	3173.7	0.3916
	64	1719.8	3772.8	0.4528		64	1173.5	3251.9	0.3615
	128	1616.5	3800.2	0.4264		128	1286.4	3302.4	0.3847
	256	1685.1	3814.7	0.4424		256	1334.3	3234.5	0.411
	512	1774.0	3724.7	0.4737		512	1231.7	3090.4	0.399
	1024	1727.8	3484.3	0.4922		1024	1263.8	3031.5	0.4168
40	32	1108.6	3048.3	0.3606	80	32	867.4	2912.6	0.2988
	64	1014.5	3084.3	0.3283		64	941.2	2912.8	0.3248
	128	1142.9	3210.4	0.3533		128	1023.5	2872.7	0.3546
	256	1026.3	3005.2	0.3413		256	1005.4	2949.3	0.3397
	512	1503.7	2930.9	0.5205		512	956.0	2852.1	0.3355
	1024	1170.1	2926.9	0.3951		1024	954.3	2701.8	0.3532
160	32	981.8	2855.0	0.3393	320	32	769.6	2662.8	0.2894
	64	863.6	2818.4	0.3061		64	771.8	2681.5	0.2882
	128	908.6	2842.8	0.3178		128	799.5	2627.0	0.304
	256	851.2	2796.4	0.3047		256	917.9	2722.0	0.3345
	512	909.6	2917.1	0.313		512	967.3	2757.1	0.3455
	1024	1174.9	2815.9	0.4093		1024	951.2	2601.3	0.3604

**Table 5** Implementation results—parameterized Matching, patterns uniformly distributed

$ \Sigma $	Patt. length	Naive	KMP	$\frac{\text{Naive}}{\text{KMP}}$	$ \Sigma $	Patt. length	Naive	KMP	$\frac{\text{Naive}}{\text{KMP}}$
2	32	25738.0	6871.8	3.7655	4	32	26104.6	7489.6	3.5351
	64	25996.5	6761.4	3.8593		64	26734.4	7538.6	3.5998
	128	26080.5	6780.8	3.8571		128	26281.4	7370.8	3.6136
	256	26269.7	6688.6	3.934		256	26204.3	7361.0	3.6062
	512	26004.0	6440.3	4.0456		512	26169.6	7123.6	3.71
6	1024	26456.0	6277.9	4.2167	8	1024	26570.9	6863.1	3.924
	32	26213.2	6818.3	3.96		32	26863.5	7229.3	3.9411
	64	26244.3	7022.8	3.8621		64	27010.3	7258.5	3.9394
	128	26130.3	6879.7	3.9429		128	26965.3	7067.4	4.0336
	256	26141.2	6778.1	3.987		256	26918.8	7099.7	4.0304
10	512	26212.3	6460.7	4.1752	20	512	27211.8	6888.9	4.1592
	1024	26171.5	6312.7	4.2986		1024	27406.5	6698.6	4.3042
	32	28663.6	7629.8	3.8967		32	28539.6	5832.4	5.1463
	64	28787.8	7787.6	3.8351		64	28543.3	6329.9	4.6772
	128	28629.8	7664.8	3.8775		128	28254.3	6041.4	4.8694
40	256	28647.0	7478.5	3.99	80	256	28526.7	5733.2	5.2725
	512	28843.4	7406.5	4.0576		512	28326.8	5546.4	5.3728
	1024	28516.9	7074.3	4.1282		1024	28457.7	5433.1	5.5292
	32	33994.8	5708.6	6.0731		32	42524.1	5292.8	8.0792
	64	33826.0	6076.9	5.6046		64	41425.9	5340.1	7.8236
160	128	33971.3	5994.7	5.7342	320	128	41547.1	5387.7	7.8057
	256	33740.9	5544.9	6.2016		256	41489.1	5269.7	8.0644
	512	34501.6	5411.8	6.5045		512	41615.2	5189.5	8.165
	1024	34172.0	5353.9	6.496		1024	42184.8	5067.8	8.478
	32	54881.0	4789.375	11.5167		32	70360.0	3919.7	17.9046
	64	56750.0	5222.7	10.8806		64	75533.8	4456.5	17.1093
	128	57775.6	5212.2	11.2048		128	75098.4	4284.8	17.4987
	256	56719.3	4953.4	11.5		256	77763.7	4238.4	18.328
	512	58276.6	4793.2	12.1498		512	75922.3	4181.3	18.1751
	1024	57331.2	4913.2	11.7029		1024	76831.3	4366.4	17.989



**Table 6** Implementation results—parameterized matching, patterns at end

$ \Sigma $	Patt. length	Naive	KMP	Naive KMP	$ \Sigma $	Patt. length	Naive	KMP	Naive KMP
2	32	26063.4	6801.4	3.8439	4	32	26616.5	7505.3	3.61
	64	26285.3	6878.0	3.828		64	26571.7	7443.4	3.6226
	128	26053.8	7047.4	3.7047		128	26385.6	7829.9	3.4449
	256	26612.5	6671.7	3.996		256	26236.1	7649.5	3.4807
	512	26501.7	6764.8	3.9329		512	26660.5	7356.9	3.6748
6	1024	26397.8	6506.4	4.0685	8	1024	26667.6	7038.6	3.8591
	32	26312.4	7071.4	3.828		32	27246.5	7421.6	3.8491
	64	26733.2	6924.6	3.9976		64	27046.1	7185.0	3.9748
	128	26470.1	7067.1	3.8636		128	27117.6	7170.1	4.009
	256	26346.3	6701.1	4.0218		256	27154.8	7089.7	4.04
10	512	26610.6	6682.2	4.117	20	512	26901.8	6998.0	4.0791
	1024	26632.3	6399.8	4.2563		1024	27227.5	6667.8	4.2963
	32	29612.8	7759.5	3.9578		32	29588.6	6153.9	5.0995
	64	28948.9	7748.2	3.8873		64	29393.4	6010.3	5.1754
	128	29305.1	7925.5	3.829		128	29498.7	6312.8	4.8688
40	256	29457.3	7619.7	4.0189	80	256	29659.5	5966.3	5.1945
	512	29650.7	7836.9	3.9754		512	29067.8	5802.3	5.226
	1024	30742.0	7421.5	4.3099		1024	28922.4	5455.1	5.5624
	32	34179.7	5577.9	6.2968		32	41534.5	5441.2	7.6963
	64	34385.3	5723.2	6.2199		64	41907.7	5373.0	7.8299
160	128	34951.8	5758.1	6.1685	320	128	41709.3	5474.4	7.6894
	256	36703.8	6033.8	6.216		256	41900.3	5211.0	8.1372
	512	37417.4	5682.4	6.7656		512	41753.4	5196.7	8.1023
	1024	35190.1	5488.2	6.5708		1024	43312.9	5074.3	8.5567
	32	52173.125	4773.875	10.9192		32	67440.5	3981.8	16.8561
160	64	54173.0	5176.9	10.4658	320	64	71874.8	4294.4	16.7108
	128	56313.9	5032.4	11.1442		128	72359.4	4315.2	16.725
	256	54897.4	5257.0	10.433		256	72268.3	4179.1	17.2654
	512	55123.0	5025.2	11.0268		512	72729.5	4234.7	17.1366
	1024	55603.0	4915.0	11.26		1024	73777.8	4152.8	17.7372

**Acknowledgements** The authors are grateful to Gonzalo Navarro and Solon Pissis for their helpful suggestions.

**Funding** Open access funding provided by Bar-Ilan University. Amihud Amir: Partly supported by ISF Grant 1475/18 and BSF Grant 2018141.

## Declarations

**Conflict of interest** On behalf of all authors, the corresponding author states that there is no conflict of interest.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will

need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

1. Amir A, Aumann A, Lewenstein M, Porat E. Function matching. *SIAM J Comput.* 2006;35(5):1007–22.
2. Amir A, Benson G, Farach M. An alphabet independent approach to two dimensional pattern matching. *SIAM J Comp.* 1994;23(2):313–23.
3. Amir A, Church KW, Dar E. Separable attributes: a technique for solving the submatrices character count problem. In: *Proceedings of 13th ACM-SIAM symposium on discrete algorithms (SODA)*; 2002. pp. 400–401.
4. Amir A, Farach M. Efficient 2-dimensional approximate matching of half-rectangular figures. *Inf Comput.* 1995;118(1):1–11.
5. Amir A, Farach M, Muthukrishnan S. Alphabet dependence in parameterized matching. *Inf Process Lett.* 1994;49:111–5.
6. Amir A, Levy A, Reuveni L. The practical efficiency of convolutions in pattern matching algorithms. *Fundam Inform.* 2008;84(1):1–15.

7. Amir A, Nor I. Generalized function matching. *J Discret Algorithm*. 2007;5(3):514–23.
8. Apostolico A, Galil Z (editors). *Pattern Matching Algorithms*. Oxford University Press; 1997.
9. Apostolico A, Lewenstein M, Erdős P. Parameterized matching with mismatches. *J Discret Algorithm*. 2007;5(1):135–40.
10. Babu GP, Mehtre BM, Kankanhalli MS. Color indexing for efficient image retrieval. *Multimed Tools Appl*. 1995;1(4):327–48.
11. Baker BS. Parameterized pattern matching: algorithms and applications. *J Comput Syst Sci*. 1996;52(1):28–42.
12. Baker BS. Parameterized duplication in strings: algorithms and an application to software maintenance. *SIAM J Comput*. 1997;26(5):1343–62.
13. Baker Brenda S. Parameterized pattern matching by Boyer–Moore-type algorithms. In: *Proceedings 6th annual ACM-SIAM symposium on discrete algorithms (SODA)*, ACM/SIAM; 1995, pp. 541–550.
14. Barton C, Iliopoulos CS, Pissis SP. Fast algorithms for approximate circular string matching. *Algorithm Mol Biol*. 2014;9:9.
15. Baumstark N, Gog S, Heuer T, Labeit J. Practical range minimum queries revisited. In: *16th Int'l symposium on experimental algorithms, (SEA)*, vol. 75, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik; 2017, pp. 12:1–12:16.
16. Cho S, Na JC, Park K, Sim JS. Fast order-preserving pattern matching. In: *Proceedings 7th conference combinatorial optimization and applications COCOA*, vol. 8287, *Lecture Notes in Computer Science*, Springer; 2013, pp. 295–305.
17. Crochemore M, Hancart C, Lecroq T. *Algorithms on strings*. Cambridge University Press; 2007.
18. Crochemore M, Iliopoulos C.S, Kociumaka T, Kubica M, Langiu A, Pissis SP, Radoszewski J, Rytter W, Walen T. Order-preserving incomplete suffix trees and order-preserving indexes. In: *Proceedings 20th international symposium on string processing and information retrieval (SPIRE)*, vol. 8214, LNCS, Springer; 2013, pp. 84–95.
19. Crochemore M, Rytter W. *Text algorithms*. Oxford University Press; 1994.
20. Dantzig GB. Linear programming. *Oper Res*. 2002;50(1):42–7.
21. Dinklage P, Fischer J, Herlez A. Engineering predecessor data structures for dynamic integer sets. In: *19th Int'l symposium on experimental algorithms, (SEA)*, vol. 190, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik; 2021, pp. 7:1–7:19.
22. Dinklage P, Fischer J, Herlez A, Kociumaka T, Kurpicz F. Practical performance of space efficient data structures for longest common extensions. In: *28th annual European symposium on algorithms, (ESA)*, vol. 173, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik; 2020, pp. 39:1–39:20.
23. Ferrada H, Navarro G. Improved range minimum queries. *J Discr Algorithm*. 2017;43:72–80.
24. Fischer M.J, Paterson M.S. String matching and other products. In: Karp RM, editor. *Complexity of computation*, SIAM-AMS Proceedings; 1974, vol. 7, pp. 113–125.
25. Franek F, Islam ASMS, Rahman MS, Smyth WF. Algorithms to compute the lyndon array. In: Holub J, Zdárek J, editors, *Proceedings of prague stringology Conference*; 2016, pp. 172–184
26. Fredriksson K, Mozgovoy M. Efficient parameterized string matching. *Inf Process Lett*. 2006;100(3):91–6.
27. Hazay C, Lewenstein M, Sokol D. Approximate parameterized matching. In: *Proceedings of 12th annual European symposium on algorithms (ESA 2004)*; 2004. pp. 414–425.
28. Holub J, Smyth WF, Wang S. Fast pattern-matching on indeterminate strings. *J Discr Algorithm*. 2008;6(1):37–50.
29. Idury R.M, Schäffer AA. Multiple matching of parameterized patterns. In: *Proceedings of 5th combinatorial pattern matching (CPM)*, vol. 807, LNCS, Springer-Verlag; 1994, pp. 226–239.
30. Ilie L, Navarro G, Tinta L. The longest common extension problem revisited and applications to approximate string searching. *J Discr Algorithm*. 2010;8(4):418–28.
31. Kim J, Amir A, Na J.C, Park K., Sim J.S. On representations of ternary order relations in numeric strings. In: *Proceedings of 2nd international conference on algorithms for big data (ICABD)*, vol. 1146, *CEUR Workshop Proceedings*; 2014. pp. 46–52.
32. Knuth DE, Morris JH, Pratt VR. Fast pattern matching in strings. *SIAM J Comp*. 1977;6:323–50.
33. Loukides G, Pissis SP. Bidirectional string anchors: a new string sampling mechanism. In: *29th annual European symposium on algorithms, (ESA)*, vol. 204, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik; 2021, pp. 64:1–64:21.
34. Park SG, Amir A, Landau GM, Park K. Cartesian Tree Matching and Indexing. In: Nadia P and Pissis SP, editors, *Proceedings 30th symposium on combinatorial pattern matching (CPM)*, vol. 128, *leibniz international proceedings in informatics (LIPIcs)*; 2019, pp. 16:1–16:14.
35. Park SG, Bataa M, Amir A, Landau GM, Park K. Finding patterns and periods in cartesian tree matching. *Theor Comput Sci*. 2020;845:181–97.
36. Puglisi SJ, Smyth WF, Turpin A. Inverted files versus suffix arrays for locating patterns in primary memory. In: *Proceedings of 13th international symposium on string processing and information retrieval (SPIRE)*, vol. 4209, LNCS, Springer; 2006, pp. 122–133.
37. Song S, Gu G, Ryu C, Faro S, Lecroq T, Park K. Fast algorithms for single and multiple pattern cartesian tree matching. *Theor Comput Sci*. 2021;849:47–63.
38. Swain M, Ballard D. Color indexing. *Int J Comput Vis*. 1991;7(1):11–32.

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.