

# Sequential Exact String Matching

(1994; Crochemore, Czumaj, Gąsieniec, Jarominek, Lecroq, Plandowski, Rytter)

Maxime Crochemore, Université de Marne-la-Vallée and  
King's College London, [monge.univ-mlv.fr/~mac](mailto:monge.univ-mlv.fr/~mac)

Thierry Lecroq, Université de Rouen, [monge.univ-mlv.fr/~lecroq](mailto:monge.univ-mlv.fr/~lecroq)

*Entry editor: Gonzalo Navarro*

**INDEX TERMS:** string matching, pattern matching, shift function, prefix, suffix, border, period

**SYNONYMS:** Exact pattern matching

## PROBLEM DEFINITION

Given a *pattern string*  $P = p_1p_2 \dots p_m$  and a *text string*  $T = t_1t_2 \dots t_n$ , both being sequences over an alphabet  $\Sigma$  of size  $\sigma$ , the *exact string matching (ESM)* problem is to find one or, more generally, all the text positions where  $P$  occurs in  $T$ , that is, compute the set  $\{j \mid 1 \leq j \leq n - m + 1 \text{ and } P = t_jt_{j+1} \dots t_{j+m-1}\}$ . The pattern is assumed to be given first and is then to be searched in several texts.

Both worst- and average-case complexity are considered. For the latter one assumes that pattern and text are randomly generated by choosing each character uniformly and independently from  $\Sigma$ . For simplicity and practicality the assumption  $m = o(n)$  is set in this entry.

## KEY RESULTS

Most algorithms that solve the ESM problem proceed in two steps: a preprocessing phase of the pattern  $P$  followed by a searching phase over the text  $T$ . The preprocessing phase serves to collect information on the pattern in order to speed up the searching phase.

The searching phase of string-matching algorithms work as follows: they first align the left ends of the pattern and the text, then compare the aligned symbols of the text and the pattern—this specific work is called an attempt or a scan—and after a whole match of the pattern or after a mismatch they shift the pattern to the right. They repeat the same procedure again until the right end of the pattern goes beyond the right end of the text. The scanning part can be viewed as operating on the text through a window, which size is most often the length of the pattern. This processing manner is called the scan and shift mechanism. Different scanning strategies of the window lead to algorithms having specific properties and advantages.

The brute force algorithm for the ESM problem consists in checking if  $P$  occurs at each position  $j$  on  $T$ , with  $1 \leq j \leq n - m + 1$ . It does not need any preprocessing phase. It runs in quadratic time  $O(mn)$  with constant extra space and performs  $O(n)$  character comparisons on average. This is to be compared with the following bounds.

**Theorem 1 (Cole et al. 1995 [3]).** *The minimum number of character comparisons to solve the ESM problem in the worst case is  $\geq n + \frac{9}{4m}(n - m)$ , and can be made  $\leq n + \frac{8}{3(m+1)}(n - m)$ .*

**Theorem 2 (Yao 1979 [15]).** *The ESM problem can be solved in optimal expected time  $O(\frac{\log m}{m} \times n)$ .*

**On-line text parsing** The first linear ESM algorithm appears in the 1970's. The preprocessing phase consists in computing the periods of the pattern prefixes, or equivalently the length of the longest border for all the prefixes of the pattern. A border of a string is both a prefix and a suffix of it distinct from the string itself. Let  $next[i]$  be the length of the longest border of  $p_1 \dots p_{i-1}$ . Consider an attempt at position  $j$ , when the pattern  $p_1 \dots p_m$  is aligned with the segment  $t_j \dots t_{j+m-1}$  of the text. Assume that the first mismatch (during a left to right scan) occurs between symbols  $p_i$  and  $t_{i+j}$  for  $1 \leq i \leq m$ . Then,  $p_1 \dots p_{i-1} = t_j \dots t_{i+j-1} = u$  and  $a = p_i \neq t_{i+j} = b$ . When shifting, it is reasonable to expect that a prefix  $v$  of the pattern matches some suffix of the portion  $u$  of the text. Doing so, after a shift, the comparisons can resume between  $p_{next[i]}$  and  $t_{i+j}$  without missing any occurrence of  $P$  in  $T$  and having to backtrack on the text. There exists two variants, depending on whether  $p_{next[i]}$  has to be different from  $p_i$  or not.

**Theorem 3 (Knuth, Morris and Pratt 1977 [11]).** *The text searching can be done in time  $O(n)$  and space  $O(m)$ . Preprocessing the pattern can be done in time  $O(m)$ .*

The search can be realized using an implementation with successor by default of the deterministic automaton  $\mathcal{D}(P)$  recognizing the language  $\Sigma^*P$ . The size of the implementation is  $O(m)$  independent of the alphabet size, due to the fact that  $\mathcal{D}(P)$  possesses  $m + 1$  states,  $m$  forward arcs, and at most  $m$  backward arcs. Using the automaton for searching a text leads to an algorithm having an efficient delay (maximum time for processing a character of the text).

**Theorem 4 (Hancart 1993 [10]).** *Searching for the pattern  $P$  can be done with a delay of  $O(\min\{\sigma, \log_2 m\})$  letter comparisons.*

Note that for most algorithms the pattern preprocessing is not necessarily done before the text parsing as it can be performed on the fly during the parsing.

**Practically-efficient algorithms** The Boyer-Moore algorithm is among the most efficient ESM algorithms. A simplified version of it, or the entire algorithm, is often implemented in text editors for the search and substitute commands.

The algorithm scans the characters of the window from right to left beginning with its rightmost symbol. In case of a mismatch (or a complete match of the pattern) it uses two precomputed functions to shift the pattern to the right. These two shift functions are called the *bad-character shift* and the *good-suffix shift*. They are based on the following observations. Assume that a mismatch occurs between character  $p_i = a$  of the pattern and character

$t_{i+j} = b$  of the text during an attempt at position  $j$ . Then,  $p_{i+1} \dots p_m = t_{i+j+1} \dots t_{j+m} = u$  and  $p_i \neq t_{i+j}$ . The good-suffix shift consists in aligning the segment  $t_{i+j+1} \dots t_{j+m}$  with its rightmost occurrence in  $P$  that is preceded by a character different from  $p_i$ . Another variant called the *best-suffix shift* consists in aligning the segment  $t_{i+j} \dots t_{j+m}$  with its rightmost occurrence in  $P$ . Both variants can be computed in time and space  $O(m)$  independent of the alphabet size. If there exists no such segment, the shift consists in aligning the longest suffix  $v$  of  $t_{i+j+1} \dots t_{j+m}$  with a matching prefix of  $x$ . The bad-character shift consists in aligning the text character  $t_{i+j}$  with its rightmost occurrence in  $p_1 \dots p_{m-1}$ . If  $t_{i+j}$  does not appear in the pattern, no occurrence of  $P$  in  $T$  can overlap the symbol  $t_{i+j}$ , then the left end of the pattern is aligned with the character at position  $i + j + 1$ . The search can then be done in  $O(n/m)$  in the best case.

**Theorem 5 (Cole 1994 (see [5, 14])).** *During the search for a non-periodic pattern  $P$  of length  $m$  (such that the length of the longest border of  $P$  is less than  $m/2$ ) in a text  $T$  of length  $n$ , the Boyer-Moore algorithm performs at most  $3n$  comparisons between letters of  $P$  and of  $T$ .*

Yao's bound can be reached using an indexing structure for the reverse pattern. This is done by the Reverse Factor algorithm also called BDM (for Backward Dawg Matching).

**Theorem 6 (Crochemore et al. 1994 [4]).** *The search can be done in optimal expected time  $O(\frac{\log m}{m} \times n)$  using the suffix automaton or the suffix tree of the reverse pattern.*

A factor oracle can be used instead of an index structure, this is made possible since the only string of length  $m$  accepted by the factor oracle of a string  $w$  of length  $m$  is  $w$  itself. This is done by the Backward Oracle Matching (BOM) algorithm of Allauzen, Crochemore and Raffinot [1]. Its behaviour in practice is similar to the one of the BDM algorithm.

**Time-space optimal algorithms** Algorithms of this type run in linear time (for both preprocessing and searching) and need only constant space in addition to the inputs.

**Theorem 7 (Galil and Seiferas 1983 [8]).** *The search can be done optimally in time  $O(n)$  and constant extra space.*

After Galil and Seiferas' first solution, other solutions are by Crochemore-Perrin [6] and by Rytter [13]. Algorithms rely on a partition of the pattern in two parts; they first search for the right part of the pattern from left to right, and then, if no mismatch occurs, they search for the left part. The partition can be: the perfect factorization [8], the critical factorization [6], or based on the lexicographically maximum suffix of the pattern [13]. Another solution by Crochemore (see [2]) is a variant of KMP [11]: it computes lower bounds of pattern prefixes periods on the fly and requires no preprocessing.

**Bit-parallel solution** It is possible to use the bit-parallelism technique for ESM.

**Theorem 8 (Baeza-Yates & Gonnet 1992; Wu & Manber 1992 (see [5, 14])).** *If the length  $m$  of the string  $P$  is smaller than the number of bits of a machine word, the preprocessing phase can be done in time and space  $\Theta(\sigma)$ . The searching phase executes in time  $\Theta(n)$ .*

It is even possible to use this bit-parallelism technique to simulate the BDM algorithm. This is realized by the BNDM (Backward Non-deterministic Dawg Matching) algorithm (see [2, 12]).

In practice, when scanning the window from right to left during an attempt, it is sometimes more efficient to only use the bad-character shift. This was first done by the Horspool algorithm (see [2, 12]). Other practical efficient algorithms are the Quick Search by Sunday (see [2, 12]) and the Tuned Boyer-Moore by Hume and Sunday (see [2, 12]).

There exists another method that uses the bit-parallelism technique that is optimal on the average though it consists actually of a filtration method. It considers sparse  $q$ -grams and thus avoids to scan a lot of text positions. It is due to Fredriksson and Grabowski [7].

## APPLICATIONS

The methods which are described here apply to the treatment of the natural language, the treatment and the analysis of the genetic sequences and of musical sequences, the problems of safety related to data flows like virus detection, and the management of the textual data bases, to quote only some immediate applications.

## OPEN PROBLEMS

There remain only a few open problems on this question. It is still unknown if it is possible to design an average optimal time constant space string matching algorithm. The exact size of the Boyer-Moore automaton is still unknown (see [5]).

## EXPERIMENTAL RESULTS

The book of G. Navarro and M. Raffinot [12] is a good introduction and presents an experimental map of ESM algorithms for different alphabet sizes and pattern lengths. Basically, the Shift-Or algorithm is efficient for small alphabets and short patterns, the BNDM algorithm is efficient for medium size alphabets and medium length patterns, the Horspool algorithm is efficient for large alphabets, and the BOM algorithm is efficient for long patterns.

## URL to CODE

The site [monge.univ-mlv.fr/~lecroq/string](http://monge.univ-mlv.fr/~lecroq/string) presents a large number of ESM algorithms (see also [2]). Each algorithm is implemented in C code and a Java applet is given.

## CROSS REFERENCES

*Sequential multiple string matching* is the version where a finite set of patterns is search in a text; *Sequential approximate string matching* is the version where errors are permitted; *Indexed string matching* refers to the case where the text is preprocessed; *Regular expression matching* is the more complex case where  $P$  can be a regular expression.

# RECOMMENDED READING

Further information can be found in the three following books: [5], [9] and [14].

## References

- [1] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. In *SOFSEM'99*, LNCS 1725, pages 291–306, 1999.
- [2] C. Charras and T. Lecroq. *Handbook of exact string matching algorithms*. King's College London Publications, 2004.
- [3] R. Cole, R. Hariharan, M. Paterson, and U. Zwick. Tighter lower bounds on the exact complexity of string matching. *SIAM J. Comput.*, 24(1):30–45, 1995.
- [4] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.
- [5] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
- [6] M. Crochemore and D. Perrin. Two-way string matching. *J. ACM*, 38(3):651–675, 1991.
- [7] K. Fredriksson and S. Grabowski. Practical and optimal string matching. In *Proceedings of SPIRE'2005*, LNCS 3772, pages 374–385, 2005.
- [8] Z. Galil and J. Seiferas. Time-space optimal string matching. *J. Comput. Syst. Sci.*, 26(3):280–294, 1983.
- [9] D. Gusfield. *Algorithms on strings, trees and sequences*. Cambridge University Press, 1997.
- [10] C. Hancart. On Simon's string searching algorithm. *Inf. Process. Lett.*, 47(2):95–99, 1993.
- [11] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.
- [12] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.
- [13] W. Rytter. On maximal suffixes and constant-space linear-time versions of KMP algorithm. *Theor. Comput. Sci.*, 299(1–3):763–774, 2003.
- [14] W. F. Smyth. *Computing Patterns in Strings*. Addison Wesley Longman, 2002.
- [15] A. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8:368–387, 1979.