The String Matching Algorithms Research Tool

S. Faro[†], T. Lecroq[‡], S. Borzì[†], S. Di Mauro[†], and A. Maggio[†]

[†]Università di Catania, Viale A.Doria n.6, 95125 Catania, Italy
[‡]Université de Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France

Abstract. String matching is the problem of finding all occurrences of a given pattern in a given text. It is an extensively studied problem in computer science because of its direct application to several areas such as text, image and signal processing, speech analysis and recognition, data compression, information retrieval, computational biology and chemistry. Since 1970 more than 85 string matching algorithms have been proposed, and more than 50% of them in the last ten years.
In this paper we present Smart, an efficient and flexible tool designed for developing, testing, comparing and evaluating string matching algorithms. It also provides the most comprehensive survey of online exact single string matching algorithms together with a set of corpora available for testing purposes.

Keywords. string matching, text processing, design and analysis of algorithms, testing framework, algorithms survey, experimental evaluation.

1 Introduction

String matching is a very important subject in the wider domain of text processing. It consists in finding *all* occurrences of a given pattern in a given text. It is an extensively studied problem in computer science, mainly due to its direct applications in many areas related with information retrieval and information analysis. String matching algorithms are also basic components used in implementations of practical software existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science. Finally they also play an important role in theoretical computer science by providing challenging problems.

Applications require two kinds of solutions depending on which string, the pattern or the text, is given first. Algorithms based on the use of automata or combinatorial properties of strings are commonly implemented to preprocess the pattern and solve the first kind of problem. In this paper we are interested in this kind of problem, which is generally referred as *online* string matching. Recently Faro and Lecroq presented a comprehensive survey [17] of almost all online string matching algorithms appeared in literature up to 2010.

In 1991 Hume and Sunday presented an efficient framework [20] for testing string matching algorithms. It was developed in the C programming language and has been extensively used in the field during the last few decades. The authors compiled their framework using the stringsearch package[1] including the implementations of 37 string matching algorithms. Although their tool is very useful and simple to be used it presents some questionable points.

First of all, it was designed in order to maintain the preprocessing and the searching phase as separate functions. Although this allows an accurate measurement of

[1] `http://hackage.haskell.org/package/stringsearch-0.3.6.4`

the preprocessing time, it needs also the pattern to be copied to some local buffer, affecting the time measurement with an additional overhead which becomes negligible only if the length of the text is large enough. In addition all data related with the pattern are stored in common structure, thus guaranteeing a similar behavior of the algorithms. However it may slightly increase the access time to pattern information during the searching phase.

A useful feature of the tool from Hume and Sunday is that all algorithms are implemented in separate files. However their compilation is not independent from the whole framework. Thus when testing several algorithms there is always a risk that they meddle with each other.

We also noticed that the use of the `signed char` type does not always work properly, because indexing of arrays does not work with negative values. This could be avoided by forcing the cast to the `unsigned char` type all values which are used as indexes.

In this paper we introduce a new version of Smart (String Matching Algorithms Research Tool), an efficient and flexible framework designed for developing, testing, comparing and evaluating string matching algorithms. It includes most of the features which characterize the tool by Hume and Sunday, but includes a lot of other useful and interesting improvements. It allows the user to completely customize the testing environment, adding new algorithms and testing them for correctness, without the need of recompiling it. Moreover it includes the implementation of more than 120 algorithms, divided into more than 300 variants, and a large set of corpora, divided into categories, which can be used inside the testing environment. Finally it has been designed in order to provide a fair comparison between different solutions, thanks also to a large variety of experimental observations.

In May 2010 a preliminary version of Smart was released to the scientific community and referred in a technical report [13] were the authors discussed a comprehensive evaluation of almost all exact string matching algorithms[2] known up to 2010. In the last six years it was used many times to perform experimental evaluation (see for instance [18, 2, 23, 15, 10]) and to test the performances of new algorithms.

One of the most interesting features of the new version of Smart is a practical and useful graphical interface which works over the standard framework and allows the use of all functionalities of the tool.

The source file of Smart, together with a detailed description and documentation, can be found at `http://www.dmi.unict.it/~faro/smart/`. It is distributed using GitHub[3] at `https://github.com/smart-tool`.

The paper is organized as follows. In Section 2 we give a brief description of the Smart tool, including its main features and the principles of fair testing which are at the basis of the framework. We give also a list of all implemented algorithms and the corpora which are included in Smart. Then in Section 4 we give a brief survey of the last comprehensive experimental evaluation performed with Smart, and describe the directions of future works in Section 5.

[2] The preliminary version of Smart included 85 different algorithms, divided in 130 variants.
[3] The GitHub web page is accessible at `https://github.com`

2 The Smart Tool in Short

Smart is an open source software which provides a standard framework for researchers in string matching. It helps users to test, design, evaluate and understand existing solutions for the exact string matching problem. Moreover it provides the implementation of (almost) all string matching algorithms and a wide corpus of text buffers. The Smart source code can be downloaded at the web page `http://www.dmi.unict.it/~faro/smart/`. It is released under the GNU general public license[4].

Smart is written in the C language and can be compiled in any operating system with a standard `gcc` compiler. The tool uses shared memory for storing the text. Thus Smart requires the system to allow the allocation of shared memory. The default size of the text is 1MB, which is small enough to be supported by any system. However if one wants to use Smart for testing algorithms on larger texts, system settings for shared memory must be checked.

In the following sections we briefly describe Smart's main features.

2.1 Implemented Algorithms

In the last 40 years tens of string matching algorithms (and even a larger number of variants thereof) have been proposed. Smart provides the implementation of more than 300 variants and more than 120 different algorithms. This number is on the rise thanks to the continuous contributions of the community.

All implemented algorithms can be divided into four classes (but further classifications are possible): *characters comparison*, *deterministic automata*, *bit parallelism* and *packed string matching*. Classical approaches to the problem make use of *comparisons between characters* or perform transitions on some kinds of *deterministic automata*. However in the last two decades a lot of work has been made in order to exploit the power of the word RAM model of computation to speed-up classical string matching algorithms. In this model, the computer operates on words of length ω, thus blocks of characters are read and processed at once. This means that usual arithmetic and logic operations on the words all take one unit of time. Most of the solutions which exploit the word RAM model are based on the *bit-parallelism* technique or on the *packed string matching* technique.

An almost comprehensive list of all algorithms implemented in the preliminary version of Smart (more than 85) can be found in [13, 17]. In the present release of the software we have extended such list introducing some additional variants of previous solutions and the following new algorithms presented between 2010 and 2016. The comprehensive list of algorithms implemented in the new version of Smart (more than 120) can be found in [8].

- Three bit-parallel algorithms for exact searching of long patterns appeared in [7]. Two algorithms are modifications of the BNDM [22] algorithm and the third one is a filtration method which utilizes locations of q-grams in the pattern. Two algorithms apply a condensed representation of q-grams.
- Two generalizations of the Forward-SBNDM [12] algorithm presented in [23]. The first generalizes the algorithm by using q-grams while the second introduces also a q-gram lookahead approach.

[4] http://www.gnu.org/licenses/gpl.html

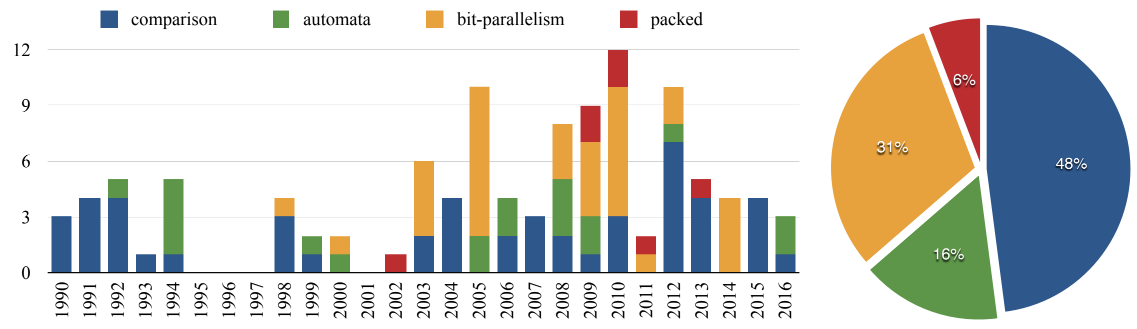


Figure 1: (On the left) The temporal distribution of algorithms proposed in the last 26 years (1990-2016) and (on the right) the percentage of all algorithms up to 2016.

- A solution based on a factorization of the pattern and on a particular encoding of the suffix automaton [16], which turns out to lead to longer shifts than that proposed by other known solutions which make use of suffix automata.
- A constant-space $\mathcal{O}(n)$-time packed string matching algorithm [2] which runs in optimal $\mathcal{O}(n/\alpha)$-time, where $\alpha = w/\log\sigma$, and even in real-time.
- Several variants of previous solutions obtained by using a general approach to string matching based on multiple sliding text-windows [14].
- A very fast string matching algorithm [11] for short patterns, which uses specialized word-size packed string matching instructions, based on the Intel streaming SIMD extensions (SSE) technology.
- Two algorithms, presented in [24], based on a combination of the Boyer-Moore [3] and Horspool [19] algorithms. It takes the maximum shift proposed by the two occurrence heuristics.
- Three improvements of the standard occurrence heuristics [4, 5].
- An improvement of Quick-Search algorithm [25] which improves the shift performed by the occurrence heuristics by computing the shift to left performed by the reverse of the pattern at a given fixed distance from the current window.
- A combination of Skip-Search and the Hash$q$ algorithms which computes buckets of positions for the fingerprint of each q-gram in the pattern. It was presented in [9].
- Improved versions of the Shift-Or and Shift-And algorithms [1] using a two way scan of the window and q-grams. They were presented in [6].

Fig.1 presents the temporal distribution of algorithms proposed in the last 26 years (1990-2016) and the percentage of algorithms belonging to each class, up to 2016. Observe that the number of proposed solutions have doubled in the last ten years, demonstrating the increasing interest in this issue.

The class of algorithms based on comparison of characters is the wider class and consists of almost 50% of all solutions. Also automata play a very important role in the design of efficient string matching algorithms and have been developed to design algorithms which have optimal sub-linear performance on average. Almost 20% of all algorithms in SMART are based on automata.

Bit-parallelism [1] takes advantage of the intrinsic parallelism of the bit operations automata. It is interesting to observe also that almost 50% of solutions in the last

Table 1: The list of those string matching algorithms implemented in SMART which were published 1970-2016. Each algorithm is associated to its acronym used in Smart.

1.	Brute-Force (BF)	no date	64.	Improved LDM 2 (ILDM2)	
2.	Deterministic-Finite-Automaton (DFA)		65.	Franek-Jennings-Smyth (FJS)	2007
3.	Morris-Pratt (MP)	1970	66.	2-Block Boyer-Moore (2BLOCK)	
4.	Knuth-Morris-Pratt (KMP)	1977	67.	Wu-Manber for Single S.M. (HASHq)	
5.	Boyer-Moore (BM)		68.	Horspool with q-grams (BMHq)	2008
6.	Horspool (HOR)	1980	69.	Two Sliding Windows (TSW)	
7.	Galil-Seiferas (GS)	1981	70.	Extended BOM (EBOM)	
8.	Apostolico-Giancarlo (AG)	1986	71.	Forward BOM (FBOM)	
9.	Karp-Rabin (KR)	1987	72.	Succint BDM (SBDM)	
10.	Zhu-Takaoka (ZT)		73.	Forward BNDM (FBNDM)	
11.	Shift-Or (SO)	1989	74.	Forward Simplified BNDM (FSBNDM)	
12.	Shift-And (SA)		75.	Bit-Parallel Length Invariant (BLIM)	
13.	Quick-Search (QS)	1990	76.	Genomic Rapid Algo for S.M. (GRASPm)	2009
14.	Optimal-Mismatch (OM)		77.	Simplified Extended BOM (SEBOM)	
15.	Maximal-Shift (MS)		78.	Simplified Forward BOM (SFBOM)	
16.	Apostolico-Crochemore (AC)	1991	79.	BNDM with q-grams (BNDMq)	
17.	Two-Way (TW)		80.	Simplified BNDM with q-grams (SBNDMq)	
18.	Tuned-Boyer-Moore (TunBM)		81.	FNDM with q-grams (UFNDMq)	
19.	Colussi (COL)		82.	Small Alphabet Bit-Parallel (SABP)	
20.	Smith (SMITH)		83.	Packed String Search (PSS)	
21.	Galil-Giancarlo (GG)	1992	84.	Streaming SIMD Extensions Filter (SSEF)	
22.	Raita (RAITA)		85.	Bounded Boyer-Moore (BBM)	2010
23.	S.M. on Ordered ALphabet (SMOA)		86.	Bounded Fast-Search (BFS)	
24.	Turbo-Boyer-Moore (TBM)		87.	Bounded Forward-Fast-Search (BFFS)	
25.	Reverse-Factor (RF)		88.	BNDM with Extended Shifts (BXS)	
26.	Not-So-Naive (NSN)	1993	89.	BNDMq Long (BQL)	
27.	Reverse-Colussi (RCOL)	1994	90.	Q-Gram Filtering (QF)	
28.	Simon (SIM)		91.	Bit-Parallel^2 Wide-Window (BP2WW)	
29.	Turbo-Reverse-Factor (TRF)		92.	Bit-Parallel Wide-Window^2 (BPWW2)	
30.	Forward-DAWG-Matching (FDM)		93.	Factorized Shift-And (KSA)	
31.	Backward-DAWG-Matching (BDM)		94.	Factorized BNDM (KBNDM)	
32.	Skip-Search (SKIP)	1998	95.	Packed Belazzougui (PB)	
33.	Alpha-Skip-Search (ASKIP)		96.	Packed Belazzougui-Raffinot (PBR)	
34.	Knuth-Morris-Pratt Skip-Search (KMPS)		97.	FSBNDM with q-grams (FSBNDMqf)	2011
35.	Nondeterministic BDM (BNDM)		98.	Packed Crochemore-Perrin (SSECP)	
36.	Berry-Ravindran (BR)	1999	99.	Fast-Search using Multiple Windows (FSw)	2012
37.	Backward-Oracle-Matching (BOM)		100.	TVSBS with Multiple Windows (TVSBSw)	
38.	Double Forward DAWG Matching (DFDM)	2000	101.	Max Shift Boyer-Moore (MSBM)	
39.	BNDM for Long patterns (BNDML)		102.	Max Shift Horspool (MSH)	
40.	Super Alphabet Simulation (SAS)	2002	103.	Enhanced Two Sliding Windows (ETSW)	
41.	Ahmed-Kaykobad-Chowdhury (AKC)	2003	104.	Hash$q$ using Multiple Hashing (MHASHq)	
42.	Fast-Search (FS)		105.	Enhanced Berry-Ravindran (RSA)	
43.	Simplified BNDM (SBNDM)		106.	Backward SNR DAWG Matching (BSDM)	
44.	Two-Way NDM (TNDM)		107.	Multiple Windows SBNDM (SBNDMw)	
45.	Long patterns BNDM (LBNDM)		108.	Multiple Windows FSBNDM (FSBNDMw)	
46.	Shift Vector Matching (SVM)		109.	Enhanced RS-A (ERSA)	2013
47.	Forward-Fast-Search (FFS)	2004	110.	Improved Occurrence Heuristics (IOM)	
48.	Backward-Fast-Search (BFS)		111.	Worst Occurrence Heuristics (WOM)	
49.	Tailed-Substring (TS)		112.	Jumping Occurrence Heuristics (JOM)	
50.	Sheik *et al.* (SSABS)		113.	Exact Packed String Matching (EPSM)	
51.	Wide Window (WW)	2005	114.	Improved Two-Way Shift-And (TSA)	2014
52.	Linear DAWG Matching (LDM)		115.	Improved Two-Way Shift-Or (TSO)	
53.	BNDM with loop-unrolling (BNDM2)		116.	Two-Way Shift-And using q-grams (TSAq)	
54.	SBNDM with loop-unrolling (SBNDM2)		117.	Two-Way Shift-Or using q-grams (TSOq)	
55.	BNDM with Horspool Shift (BNDMBMH)		118.	Simple String Matching (SSM)	2015
56.	Horspool with BNDM test (BMHBNDM)		119.	Quantum Leap Quick-Search (QLQS)	
57.	Forward NDM (FNDM)		120.	Enhanced ERS-A (EERSA)	
58.	Bit parallel Wide Window (BWW)		121.	Four Sliding Windows (FSW)	
59.	Average Optimal Shift-Or (AOSO)		122.	Skip-Search using q-grams (SKIPq)	2016
60.	Fast Average Optimal Shift-Or (FAOSO)		123.	BSDM with q-grams (BSDMqx)	
61.	Thathoo *et al.* (TVSBS)	2006	124.	BSDMqx multiple windows (BSDMqxw)	
62.	Horspool using Probabilities (PBMH)				
63.	Improved LDM (ILDM1)				

ten years (and 31% all along) are based on bit-parallelism, and it seems that such number follows an increasing trend.

In packed string matching, multiple characters are packed into one larger word, so that the characters can be compared in bulk rather than individually. In this context, if the characters of a string are drawn from an alphabet of size σ, then $\lfloor w/\log\sigma\rfloor$ different characters fit in a single word, using $\lfloor\log\sigma\rfloor$ bits per characters. Although algorithms in this class appeared in the last four years they turn out to be among the fastest solutions [21, 10], reaching in some cases the optimal $\mathcal{O}(n\log\sigma/w)$ time complexity [2].

2.2 Algorithm Testing and Evaluation

The main command provided by the tool, `smart` indeed, is used for running experimental tests. The experimental settings could be almost completely customized. The easiest way to use SMART is to run a single search for a custom pattern and a custom text. To this purpose one should use the `-simple` parameter followed by the pattern and the text. Otherwise it is possible to select the corpus which will be used to compute the experimental results by typing the parameter `-text` followed by the name of the selected corpus (for ex. `smart -text genome`). It is also possible to select more than one corpus by typing the name of the corpora, separated by a dash symbol (for ex. `smart -text genome-protein`). You can also type the parameter `-text all` in order to run experimental tests for all corpora, in which case the corpora will be processed one after another.

For each input file, SMART generates sets of r patterns of fixed length, randomly extracted from the text, where the length of the patterns ranges over the set of values $\{2^k \mid 1 \leq k \leq 12\}$, so that running times can be easily reported in a log-scale plot. The value r is set to 500 by default, but it is allowed to use the parameter `-pset` in order to modify the size of the set of patterns generated by the tool (for ex. you can type `smart -text genome -pset 100`). You can also use the parameter `-short` in order to perform experimental tests on short patterns, whose length ranges over the set of values $\{2+\ell \mid 0 \leq \ell \leq 30\}$. If necessary, it is also allowed to restrict the pattern's length to a given range by using the parameter `-plen` and indicating an upper bound and a lower bound for such lengths (for ex. you can type the command `smart -text genome -plen 8 64`).

Many algorithms are very slow under particular conditions. In order to avoid excessive running times during the experimental evaluation it is possible to set a *time bound* which cannot be exceeded by any single run. This can be done by using the parameter `-tb`, followed by a value expressed in milliseconds. By default such bound is set to 300 ms.

The SMART tool has been developed in order to follow the principles of fair testing in string matching. In particular the experimental testing is based on the following features:

Algorithm verification
The tool verifies that all tested algorithms work properly. This verification is done by counting the number of matches returned by the procedure and testing whether the search stops properly at the end of the text. Since all searched patterns are always randomly extracted from the text, it is guaranteed that the number of occurrences

is always equal or greater than 1. It is not uncommon that some algorithms do not work for specific input parameters. For instance a q-gram based algorithm does not work for patterns shorter than q characters. The framework also provides a control mechanism able to distinguish a malfunctioning from a not working instance, in which case an error message is returned by the tool.

Clean time measuring
The SMART tool has been designed in order to rule out from time measurements all disturbing events. To this purpose the reading of data (text and patterns) belongs to an outer part of the test setting, so that times spent to reading is excluded from time measurements. Moreover printing of matches is not performed during time measurement, since printing produces also an additional overhead, which is partly unsynchronized. Finally the framework has been developed in order that the time measurement itself does not disturb the work of algorithms.

Fair comparison
Since in SMART the measuring is focused on performance, the tested algorithms have been implemented in a uniform way, using the same standard for processing string characters, compare them, performing automata transitions, computing matches and for more other common tasks. Then SMART uses the `-O3` level of optimization, which is the highest optimization level. All algorithms are available online and can be analyzed and improved by the community. During time measurement all tested algorithms share the same input data in order to allow a fair comparison. Moreover the SMART tool has been designed in order to ensure that no residual information in cache, during multiple executions of the same algorithm, may be used in the next attempt affecting the time measurement.

Algorithm preprocessing
It has never been clear in the literature if preprocessing should be included in the measurements. The work done in preprocessing is only a proportion of the whole task, and it may depend on the pattern length and on the alphabet size. According to Horspool [19] in string matching the timings do not include the work of initializing tables, however for an *online* algorithm, preprocessing time is spent to compute useful informations which are then used for speeding up the search process. Moreover, it is also true that in most cases the longer is the preprocessing time the faster is the searching. This line of reasoning finds its borderline case in an *offline* algorithms, where a preprocessing of the text leads to an extremely fast searching phase. For this reason the SMART tool has been developed in order to include the preprocessing time in the performance measurement.

However the tool can be set up in order to separate time measurements in searching and preprocessing times. This can be done by using the parameter `-pre`. By default SMART produces only a single time measurement which includes preprocessing and searching time.

Stability measurement
It is also useful to find out how accurately repeatable the results are. To this purpose using only average running times easily hides important details. It is common, in fact, that for some algorithms running times move away from the mean value. This is what in general we associate with the *stability* of a given algorithm: the smaller is the

standard deviation of the running times the superior is its stability. Thus the SMART tool can be set up in order to compute also standard deviation values of running times (use the parameter `-std`). In addition the tool also allows the computation of the best and worst running times obtained during the experimental evaluation (use the parameter `-dif`).

2.3 Output Formats

The SMART tool associates to any experimental test a unique alphanumeric code on 13 characters, beginning with the prefix `EXP` and followed by a string of 10 numbers computed from the timestamp. At the end of the execution of an experimental test SMART stores experimental data in the directory `results/EXPCODE`, where `EXPCODE` is the unique code associated with the experimental test. Files containing experimental data are named with the name of the corpus which has been selected. The system can store experimental data in different formats: simple text, LaTeX, xml, html and php format.

Files in xml format report data in a structured way suitable to be processed or included in other documents, while html files present data in a tabular format. An additional `index.html` file is generated which contains the list of all html pages containing the experimental results computed during the test. Figure 2 shows a portion of the HTML output produced by SMART.

Finally, LaTeX files can be generated to make easy the inclusion of tables containing the experimental results in LaTeX source files.

2.4 Text Corpora

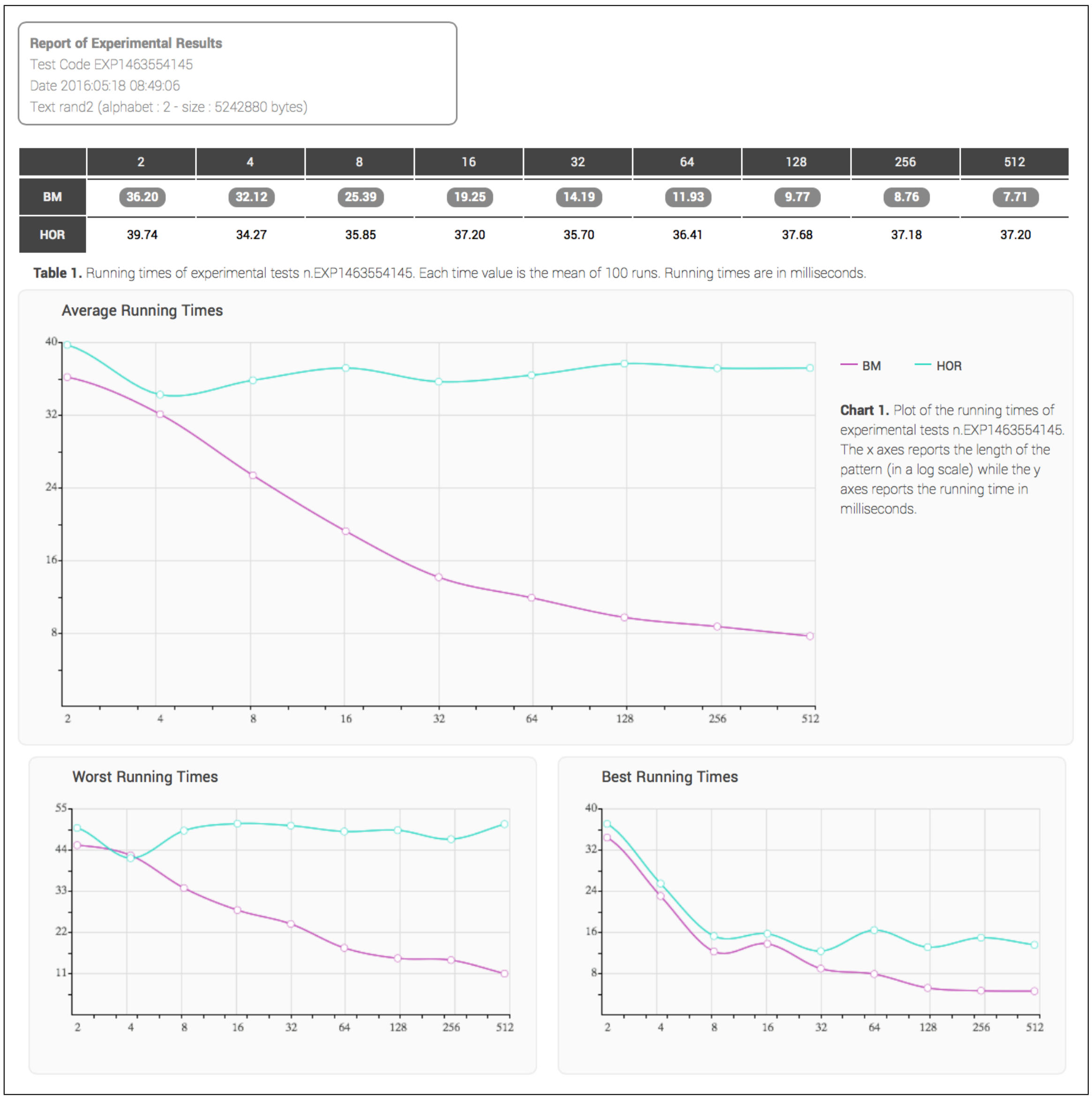
SMART comes with a set of corpora which can be selected in for running experimental results. The corpora are stored in a directory named "`data`", and each corpus consists of a set of texts stored in a sub-directory with the same name of the corpus. Each subdirectory contains and index file (`index.txt`) containing the names and a description of all the files contained in the corpus. It is possible to select the corpus which will be used to compute the experimental results by typing the parameter `-text` followed by the name of the selected corpus.

The SMART tool allows to set an upper bound dimension of the text size used during the experimental results. By default this upper bound dimension is set to 1MB. This means that (at most) the first 1MB of the selected corpus will be used for testing. The default upper bound dimension can be changed by using the parameter `-tsize`, followed by an integer value which indicate the dimension, in Mbytes, which will be used (for ex. `smart -text genome -tsize 5`).

Then all files listed in the index will be loaded in a text buffer, one by one, until the upper bound is reached. If the upper bound is larger than the whole size of the corpus the list of files is processed again in order to fill the whole buffer[5]. In details, SMART provides the following set of 15 corpora.

(i) `englishTexts`, a set of english texts (6.1 MB) over an alphabet of 94 characters. It includes two text of size 3.9 MB and 2.4 MB, respectively.

[5] Note that during the experimental evaluation the text buffer is stored in shared memory, thus if you set the upper bound to a value K MB it is necessary to ascertain your system allows the allocation of at least K MB of shared memory


Report of Experimental Results
Test Code EXP1463554145
Date 2016:05:18 08:49:06
Text rand2 (alphabet : 2 - size : 5242880 bytes)

	2	4	8	16	32	64	128	256	512
BM	36.20	32.12	25.39	19.25	14.19	11.93	9.77	8.76	7.71
HOR	39.74	34.27	35.85	37.20	35.70	36.41	37.68	37.18	37.20

Table 1. Running times of experimental tests n.EXP1463554145. Each time value is the mean of 100 runs. Running times are in milliseconds.

Chart 1. Plot of the running times of experimental tests n.EXP1463554145. The x axes reports the length of the pattern (in a log scale) while the y axes reports the running time in milliseconds.

Figure 2: A portion of the HTML output produced by Smart. Experimental data are reported in tabular and in graphical form. Here we observe average running times, worst running times and best running times of Horspool and Boyer-Moore compared on a Rand2 text buffer.

(ii) `italianTexts`, a set of italian texts (5 MB) over an alphabet of 120 characters. It includes seven texts whose size ranges from 281 KB to 1.5 MB.

(iii) `frenchTexts`, a set of french texts (6.6 MB) over an alphabet of 119 characters. It includes seven texts whose size ranges from 631 KB to 1.2 MB.

(iv) `chineseTexts`, a set of chinese texts (5.7 MB) over an alphabet of 160 characters. It includes five texts whose size ranges from 745 KB to 2.3 MB.

(v) `genome`, a set of DNA sequences (4.4 MB) over an alphabet of 4 characters. It includes Complete genome of the E. Coli bacterium (4.4 MB).

(vi) `protein`, a set of protein sequences (3.1 MB) over an alphabet of 20 characters. It includes a protein sequence from the Human genome (3.1 MB).

(vii) `midimusic`, a set of midi sequences (2.7 MB) over an alphabet of 117 characters. It includes 206 midi files on Johann Sebastian Bach work (1685-1750) whose size ranges from 4 KB to 205 KB.

(viii) `rand`σ, random texts (5 MB) over an alphabet of size σ with a uniform distribution, where σ ranges over the values $\{2, 4, 8, 16, 32, 64, 128, 256\}$.

Files (i) and (v) are from the Large Canterbury Corpus[6], files (ii), (iii) and (iv) are from the Gutenberg project[7] while file (vi) is from the Protein Corpus[8]. Finally files in (vii) are from the Johann Sebastian Bach Midi Page[9].

In addition the SMART tool provides a simple way for adding new corpora to the default set. This can be done by simply introducing a new sub-directory named with the name of the corpus, and containing the set of selected files together with an index file listing their names.

2.5 Adding New Algorithms

The SMART tool is not only a framework for testing all known string matching algorithms. It provides also an easy and fast way for assisting researchers to develop and test new efficient algorithms. It is possible to add new string matching algorithms to SMART, testing them for correctness and compare their efficiency against the previous solutions.

The following few requirements must be guaranteed: a new algorithm must be implemented in the `C` programming language and must include the header file "`include/main.h`". The main method must be defined as

```
int search(unsigned char *x, int m, unsigned char *y, int n)
```

where `x` maintains the pattern, `y` maintains the text, while `m` and `n` are their lengths, respectively. The method must return the number of occurrences of the pattern in the text. In addition, if the algorithm does not run under particular conditions (for instance when the length of the pattern is less than a given value), it is required the algorithm to return the value -1.

Since preprocessing time is computed separately from searching time, it is required to arrange the code concerning the preprocessing phase between the macros `BEGIN_PREPROCESSING` and `END_PREPROCESSING`, while the code concerning the searching phase must be arranged between the macros `BEGIN_SEARCHING` and `END_SEARCHING`. Figure 3 present the `C` code of the Horspool algorithm in a format suitable for inclusion in SMART.

Before compiling the `C` file, copy the header file `main.h` (which is stored in the folder `source/algos/include`) in the same directory. Then put the compiled binary file in the directory "`source/bin`". Before running a new experimental setting you can test the correctness of your algorithm by executing the command "`./test algoname`", where `algoname` is the name of the binary file of your algorithm. Then it will be possible to include the new algorithm in SMART by typing the command "`./select -add algoname`".

[6] `http://www.data-compression.info/Corpora/CanterburyCorpus/`
[7] `http://www.gutenberg.org`
[8] `http://data-compression.info/Corpora/ProteinCorpus/`
[9] `http://www.bachcentral.com`

```
#include "include/define.h"
#include "include/main.h"

int search(unsigned char *P, int m, unsigned char *T, int n) {
          int i, s, count, hbc[SIGMA];

          BEGIN_PREPROCESSING
          for(i=0;i<SIGMA;i++) hbc[i] = m;
          for(i=0;i<m-1;i++) hbc[P[i]] = m-i-1;
          END_PREPROCESSING

          BEGIN_SEARCHING
          s = 0;
          count = 0;
          while(s <= n-m) {
                    i = 0;
                    while( i<m && P[i]==T[s+i] ) i++;
                    if( i==m ) count++;
                    s += hbc[T[s+m-1]];
          }
          END_SEARCHING
          return count;
}
```

Figure 3: The C code of the Horspool algorithm for string matching.

3 A Graphical User Interface

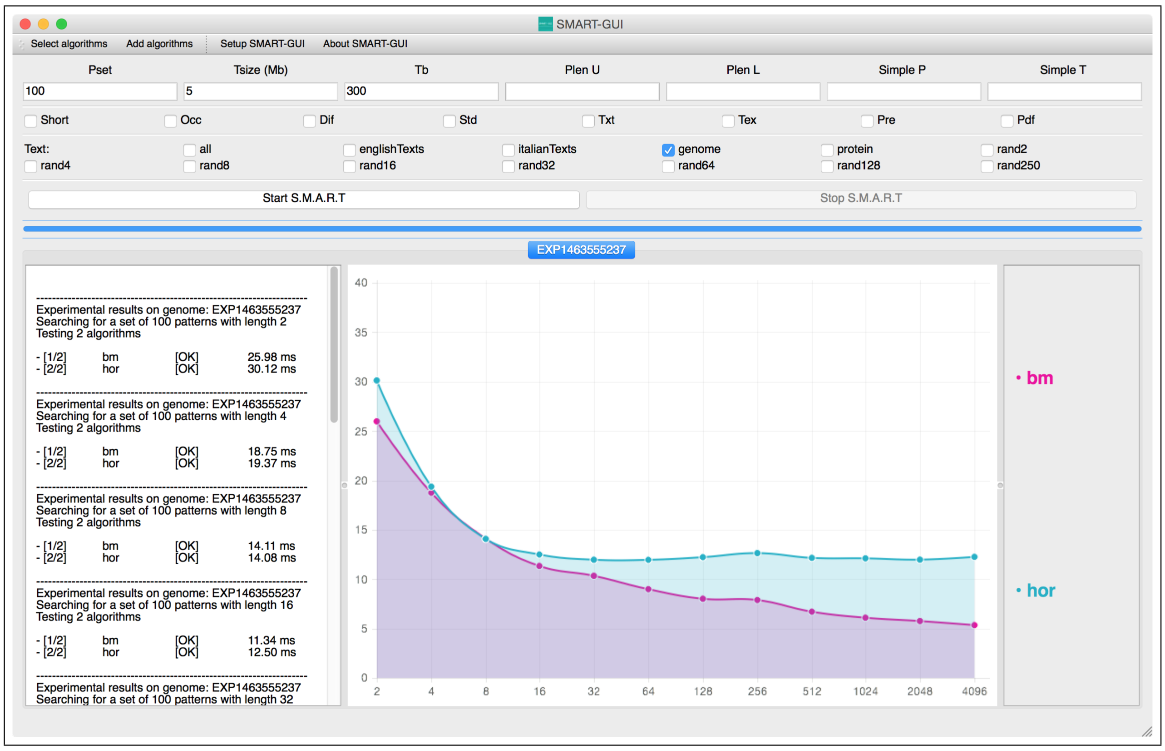
The new version of Smart comes with a useful Graphical User Interface (SmartGUI) which could be used for running experimental results. It is implemented in `C++` using the Qt WebKit, one of the major engine to render webpages and execute JavaScript code. Figure 4 shows a screenshot of the SmartGUI where reporting the average running times of the Horspool and Boyer-Moore algorithms when compared on a genome sequence.

The central part of SmartGUI is dedicated to report graphs of running times. The text output is reported next to the graphs, giving a familiar feedback to users which execute Smart using the terminal. When several experimental evaluations are executed, SmartGUI organizes the graphs using tabs.

SmartGUI has been developed to make easier the use of Smart. It allows to view in real-time all the results of the experimental evaluations and to compare the single algorithms of the experiments through the tabs.

In addition SmartGUI makes easy the customization of the tests through input text parameters and checkbox to choose the output format (txt, LaTeX, pdf, etc.) and the text buffers. It also allows to select algorithms or add new algorithms and run tests on it. During any test SmartGUI shows the status through a progress bar. When a single experimental evaluation ends SmartGUI shows more statistics through a web view opening the html format results.

Executable binaries of SmartGUI are available at the Smart web page for Windows, Linux and Mac. It can be downloaded and installed in any folder of your computer, even different from your main Smart folder. At the first execution of SmartGUI it is necessary to link the main Smart folder using the setup procedure available at the menu button `setup smart gui`.



Pset	Tsize (Mb)	Tb	Plen U	Plen L	Simple P	Simple T
100	5	300				

☐ Short ☐ Occ ☐ Dif ☐ Std ☐ Txt ☐ Tex ☐ Pre ☐ Pdf

Text: ☐ all ☐ englishTexts ☐ italianTexts [✓ genome] ☐ protein ☐ rand2
☐ rand4 ☐ rand8 ☐ rand16 ☐ rand32 ☐ rand64 ☐ rand128 ☐ rand250

```
-----------------------------------------------------------------
Experimental results on genome: EXP1463555237
Searching for a set of 100 patterns with length 2
Testing 2 algorithms

- [1/2]      bm          [OK]         25.98 ms
- [2/2]      hor         [OK]         30.12 ms

-----------------------------------------------------------------
Experimental results on genome: EXP1463555237
Searching for a set of 100 patterns with length 4
Testing 2 algorithms

- [1/2]      bm          [OK]         18.75 ms
- [2/2]      hor         [OK]         19.37 ms

-----------------------------------------------------------------
Experimental results on genome: EXP1463555237
Searching for a set of 100 patterns with length 8
Testing 2 algorithms

- [1/2]      bm          [OK]         14.11 ms
- [2/2]      hor         [OK]         14.08 ms

-----------------------------------------------------------------
Experimental results on genome: EXP1463555237
Searching for a set of 100 patterns with length 16
Testing 2 algorithms

- [1/2]      bm          [OK]         11.34 ms
- [2/2]      hor         [OK]         12.50 ms

-----------------------------------------------------------------
Experimental results on genome: EXP1463555237
Searching for a set of 100 patterns with length 32
```

Figure 4: A screenshot of the Graphical User Interface of SMART. Here we observe average running times of Horspool and Boyer-Moore algorithms when compared on a genome sequence.

4 Experimental Evaluation

In this section we discuss experimental results which could be performed using SMART. The details of the experimental evaluation can be analyzed at the SMART web page (`http://www.dmi.unict.it/~faro/smart/`).

A recent survey [13] already presents an extensive experimental evaluation of almost all string matching algorithms used for searching in different texts. The authors used SMART for computing the experimental results, indeed.

Other more recent experimental evaluations using the SMART tool appeared in [16, 14, 2, 11] where new efficient algorithms were presented.

Thus in this section we do not give another extensive experimental evaluation of string matching algorithms, but we focus our attention on new general experimental observations which SMART allows to do. In particular SMART allows to analyze string matching algorithms from three different points of view: their efficiency, their stability and their flexibility.

Efficiency Measurement

The efficiency of an algorithm is evaluated in SMART by computing the mean of running times over a large set of attempts.

Most string matching algorithms are characterized by a performance plot with a decreasing trend. Thus on average the performances increases when the length of the pattern increases. Almost all efficient algorithms, in fact, are based on a sliding window approach whose shift is at most long as the length of the pattern. This behavior is evident, for instance, in Figure 2 and in Figure 4, where we show the experimental results of the well known Horspool [19] and Boyer-Moore [3] algorithms, which are comparison based algorithms based on the occurrence heuristic. Also automata show such decreasing trend, however almost all of them show a decrease in performances in the case of longer patterns. This is due to the size explosion of the underlying automaton and of the correspondent preprocessing time.

Algorithms based on q-grams are quite efficient on average. In general their performance increases when the length of the pattern increases or when the value of q increases. However, on the other hand in the case of short patterns their performances drastically decreases when the value of q increases.

Stability Measurement

In Smart the stability of an algorithm is computed as the standard deviation of running times observed during the evaluation. Such value shows how much variation exists from the average, i.e. the mean of the running times. A low standard deviation indicates that the running times tend to be very close to the mean, underlying a high stability of the algorithm. On the other hand a high standard deviation indicates that the running times are spread out over a large range of values, thus indicating a low stability. It turns out from our observations that almost all algorithms have a low stability for short patterns while their stability increases when the length of the pattern increases. Such behavior becomes more evident for larger alphabets.

Sometimes an opposite behavior can be observed when searching on texts over a small alphabet like DNA sequences. This is the case, for instance, of some comparison based algorithms based on the occurrence heuristic whose stability decreases when the length of the pattern gets shorter. This trend can be explained by the reduced combination of strings which could be obtained when the pattern is short and the alphabet is small.

Flexibility Measurement

Flexibility is used as an attribute of various types of systems. In the field of string matching, it refers to algorithms that can adapt when changes in the input data occur. Thus a string matching algorithm can be considered flexible when, for instance, it maintains good performances for both short and long patterns, or in the case of both small and large alphabets.

As already observed most string matching algorithms obtain good performances only in the case of long patterns sacrificing their performance for short ones. This is a common behavior, for instance, for all algorithm which make use of a sliding window approach. Such approach allows the pattern to slide along the text by performing subsequent shifts. Each shift can be at most as long as the length of the pattern. It turns out that statistically the shift increases when the length of the pattern increases, or when the size of the alphabet increases. Although bit-parallel algorithms are designed to be extremely efficient in the case of long patterns, also this class of algorithms suffers of a lack in flexibility.

Only packed string matching algorithms turn out to have good performances for short patterns. This is the case of the SSECP algorithm [2] whose performances degrade when the length of the pattern increases. It is also the case of the EPSM algorithm [11], whose flexibility is obtained only by combining different algorithms, depending on the length of the pattern.

5 Conclusions, Future Works and Acknowledgements

We presented SMART (`http://www.dmi.unict.it/~faro/smart/`) a flexible testing and evaluation tool for single exact string matching algorithms. It contains the implementation of almost all string matching algorithms appeared since 1970 up to 2016. The tool helps researchers in the filed in various way and we encourage them to contribute to the project by providing their own code for testing. Many improvements are possible to enhance SMART, including its adjustment in order to work with 128 bit processors.

We wish to thank Prof. Jorma Tarhio, Dr. M. Oğuzhan Külekci and Dr. Arseny Kapoulkine for helpful comments and suggestions in improving the SMART framework. We wish also to thanks the authors of several string matching algorithms for having provided their codes for inclusion in SMART.

References

1. Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
2. Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. Optimal packed string matching. In Supratik Chakraborty and Amit Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India*, volume 13 of *LIPIcs*, pages 423–432. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.
3. Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
4. Domenico Cantone and Simone Faro. Improved and self-tuned occurrence heuristics. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013*, pages 92–106. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2013.
5. Domenico Cantone and Simone Faro. Improved and self-tuned occurrence heuristics. *J. Discrete Algorithms*, 28:73–84, 2014.
6. Branislav Durian, Tamanna Chhabra, Sukhpal Singh Ghuman, Tommi Hirvola, Hannu Peltola, and Jorma Tarhio. Improved two-way bit-parallel search. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014*, pages 71–83. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014.
7. Branislav Durian, Hannu Peltola, Leena Salmela, and Jorma Tarhio. Bit-parallel search algorithms for long patterns. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, volume 6049 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 2010.
8. Simone Faro. Exact online string matching bibliography. *CoRR*, abs/1605.05067, 2016.
9. Simone Faro. A very fast string matching algorithm based on condensed alphabets. In *Algorithmic Aspects in Information and Management - 10th International Conference, AAIM 2016. Proceedings*, volume 9778 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2016.
10. Simone Faro and M. Oguzhan Külekci. Fast multiple string matching using streaming SIMD extensions technology. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, volume 7608 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 2012.

11. Simone Faro and M. Oguzhan Külekci. Fast packed string matching for short patterns. In Peter Sanders and Norbert Zeh, editors, *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013*, pages 113–121. SIAM, 2013.

12. Simone Faro and Thierry Lecroq. Efficient variants of the backward-oracle-matching algorithm. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2008, Prague, Czech Republic, September 1-3, 2008*, pages 146–160. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2008.

13. Simone Faro and Thierry Lecroq. The exact string matching problem: a comprehensive experimental evaluation. *CoRR*, abs/1012.2547, 2010.

14. Simone Faro and Thierry Lecroq. Fast searching in biological sequences using multiple hash functions. In *12th IEEE International Conference on Bioinformatics & Bioengineering, BIBE 2012, Larnaca, Cyprus, November 11-13, 2012*, pages 175–180. IEEE Computer Society, 2012.

15. Simone Faro and Thierry Lecroq. A fast suffix automata based algorithm for exact online string matching. In Nelma Moreira and Rogério Reis, editors, *Implementation and Application of Automata - 17th International Conference, CIAA 2012, Porto, Portugal, July 17-20, 2012. Proceedings*, volume 7381 of *Lecture Notes in Computer Science*, pages 149–158. Springer, 2012.

16. Simone Faro and Thierry Lecroq. A multiple sliding windows approach to speed up string matching algorithms. In Ralf Klasing, editor, *Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings*, volume 7276 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 2012.

17. Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2):13, 2013.

18. F. Hongbo, Y. Nianmin, and M. Haifeng. A practical and average optimal string matching algorithm based on Lecroq. In *IEEE Internet Computing for Science and Engineering*, pages 57–63, 2010.

19. R. Nigel Horspool. Practical fast searching in strings. *Softw., Pract. Exper.*, 10(6):501–506, 1980.

20. Andrew Hume and Daniel Sunday. Fast string searching. In *Proceedings of the Summer 1991 USENIX Conference, Nashville, TE, USA, June 1991*, pages 221–234. USENIX Association, 1991.

21. M. Oguzhan Külekci. Filter based fast matching of long patterns by using SIMD instructions. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, August 31 - September 2, 2009*, pages 118–128. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2009.

22. Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In Martin Farach-Colton, editor, *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20-22, 1998, Proceedings*, volume 1448 of *Lecture Notes in Computer Science*, pages 14–33. Springer, 1998.

23. Hannu Peltola and Jorma Tarhio. Variations of forward-sbndm. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2011, Prague, Czech Republic, August 29-31, 2011*, pages 3–14. Prague Stringology Club, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2011.

24. Mohammed Sahli and Tetsuo Shibuya. Max-shift BM and max-shift horspool: Practical fast exact string matching algorithms. *JIP*, 20(2):419–425, 2012.

25. Daniel Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.