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. Some algorithms are based on the use of automata or combinatorial properties of strings and are implemented to preprocess the pattern and solve the first kind of problem, whereas we are interested in the second kind, which is generally referred as online matching. Recently, we have presented a comprehensive overview of all online string matching algorithms that appeared in literature up to 2010.

In 1991 Hume and Sunday presented an efficient framework for testing string matching algorithms. This framework has been extensively used in the field of string searching algorithms, and in recent years it has been extended to support other string processing challenges.
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 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, together with a comprehensive evaluation of almost all exact string matching algorithms 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 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 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 evaluation performed with Smart, and describe the directions of future works.

---

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/smart-tool.
2 The Smart Tool in Short

SMART is an open source software which provides a standard framework for researchers to design, evaluate and understand string matching algorithms. Moreover it provides a wide corpus of text buffers downloaded at the web page \texttt{http://www.dmi.unict.it/~faro/smart/}. It is licensed under the GNU general public license\footnote{\texttt{http://www.gnu.org/licenses/gpl.html}}.

SMART is written in the C language and can be compiled in any operating system with a standard \texttt{gcc} compiler. Thus it requires the system to allow the allocation of shared memory. The default size of the text 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): \textit{characters comparison}, \textit{deterministic automata}, \textit{bit parallelism} and \textit{packed string matching}. Classical approaches to the problem make use of \textit{comparisons between characters} or perform transitions on some kinds of \textit{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 $\omega$, 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. Thus the solutions which exploit the word RAM model are based on \textit{bit-parallelism} technique or on the \textit{packed string matching} technique.

An almost comprehensive list of algorithms implemented in the preliminary version of SMART (more than 85) can be found in \cite{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 \cite{8}.

- Three bit-parallel algorithms for exact searching of long patterns appeared in \cite{7}. Two algorithms are modifications of the BNDM \cite{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 \cite{12} algorithm presented in \cite{23}. The first generalizes the algorithm by using $q$-grams while the second introduces also a $q$-gram lookahead approach.
• 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 $O(n)$-time packed string matching algorithm [2] which runs in optimal $O(n/\alpha)$-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].


• 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 the 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].

Figure 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 that 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 automatically. It is interesting to observe also that almost 50% of solutions in the last ten years are based on bit-parallelism, and it seems that such number is still increasing.
<table>
<thead>
<tr>
<th></th>
<th>String Matching Algorithm</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brute-Force (BF)</td>
<td>No date</td>
</tr>
<tr>
<td>2</td>
<td>Deterministic-Finite-Automaton (DFA)</td>
<td>1970</td>
</tr>
<tr>
<td>3</td>
<td>Apostolico-Giancarlo (AG)</td>
<td>1986</td>
</tr>
<tr>
<td>4</td>
<td>Karp-Rabin (KR)</td>
<td>1987</td>
</tr>
<tr>
<td>5</td>
<td>Zhu-Takaoka (ZT)</td>
<td>1990</td>
</tr>
<tr>
<td>6</td>
<td>Shift-Or (SO)</td>
<td>1992</td>
</tr>
<tr>
<td>7</td>
<td>Shift-And (SA)</td>
<td>1993</td>
</tr>
<tr>
<td>8</td>
<td>Reverse-Factor (RF)</td>
<td>1994</td>
</tr>
<tr>
<td>9</td>
<td>Not-So-Naive (NSN)</td>
<td>1995</td>
</tr>
<tr>
<td>10</td>
<td>Shift-Or (SO)</td>
<td>1996</td>
</tr>
<tr>
<td>11</td>
<td>Shift-And (SA)</td>
<td>1997</td>
</tr>
<tr>
<td>12</td>
<td>Biweak-DAWG-Matching (BDM)</td>
<td>1998</td>
</tr>
<tr>
<td>13</td>
<td>BMHq</td>
<td>1999</td>
</tr>
<tr>
<td>14</td>
<td>BNDM with Extended Shifts (BXS)</td>
<td>2000</td>
</tr>
<tr>
<td>15</td>
<td>BNDMq Long (BQL)</td>
<td>2001</td>
</tr>
<tr>
<td>16</td>
<td>Fast-Search using Multiple Windows (FSW)</td>
<td>2002</td>
</tr>
<tr>
<td>17</td>
<td>Enhanced ERS-A (ERSA)</td>
<td>2003</td>
</tr>
<tr>
<td>18</td>
<td>Improved Occurrence Heuristics (IOH)</td>
<td>2004</td>
</tr>
<tr>
<td>19</td>
<td>Jumping Occurrence Heuristics (JOH)</td>
<td>2005</td>
</tr>
<tr>
<td>20</td>
<td>Exact Packed String Matching (EPMS)</td>
<td>2006</td>
</tr>
<tr>
<td>21</td>
<td>Improved Two-Way Shift-And (TSA)</td>
<td>2007</td>
</tr>
<tr>
<td>22</td>
<td>Improved Two-Way Shift-And using q-grams (TSAq)</td>
<td>2008</td>
</tr>
<tr>
<td>23</td>
<td>Simple String Matching (SSM)</td>
<td>2009</td>
</tr>
<tr>
<td>24</td>
<td>Quantum Leap Quick-Search (QLQS)</td>
<td>2010</td>
</tr>
<tr>
<td>25</td>
<td>Enhanced ERS-A (EERSA)</td>
<td>2011</td>
</tr>
<tr>
<td>26</td>
<td>Four Sliding Windows (FSW)</td>
<td>2012</td>
</tr>
<tr>
<td>27</td>
<td>Enhanced Two Sliding Windows (ETSWS)</td>
<td>2013</td>
</tr>
<tr>
<td>28</td>
<td>Improved Occurrence Heuristics (WOH)</td>
<td>2014</td>
</tr>
<tr>
<td>29</td>
<td>Jumping Occurrence Heuristics (JOHm)</td>
<td>2015</td>
</tr>
<tr>
<td>30</td>
<td>Improved Two-Way Shift-And using q-grams (TSAq)</td>
<td>2016</td>
</tr>
<tr>
<td>31</td>
<td>Simple String Matching (SSM)</td>
<td>2017</td>
</tr>
<tr>
<td>32</td>
<td>Quantum Leap Quick-Search (QLQS)</td>
<td>2018</td>
</tr>
<tr>
<td>33</td>
<td>Enhanced ERS-A (EERSA)</td>
<td>2019</td>
</tr>
<tr>
<td>34</td>
<td>Four Sliding Windows (FSW)</td>
<td>2020</td>
</tr>
<tr>
<td>35</td>
<td>Improved Occurrence Heuristics (IOH)</td>
<td>2021</td>
</tr>
</tbody>
</table>

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.
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 are drawn from an alphabet of size $\sigma$, 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 $O(n \log \sigma / w)$ 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, depending on the pattern length and on the alphabet size. According to Horspool 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 information 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 algorithm, 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 \(-\text{dif}\).

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 and can 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 text files containing
the experimental results computed during the test as well as the description 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 sub-
directory 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 \(-\text{text}\) 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
\(-\text{tsize}\), followed by an integer value which indicate the dimension, in Mbytes, which
will be used (for ex. \texttt{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.

SMART provides the following set of 15 corpora.

(i) \texttt{englishTexts}, a set of english texts (6.1 MB) over an alphabet of 94 charact ers.

It includes two text of size 3.9 MB and 2.4 MB, respectively.

(ii) \texttt{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) \texttt{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.

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.
Figure 2. A portion of the HTML output produced by SMART. Experimental data are reported in tabular and 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.

(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 the 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) **randσ**, 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, files (ii), (iii) and (iv) are from the Gutenberg project. Files in (vi) are from the Protein Corpus. Finally, files in (vii) are from the Johann Sebastian Bach MIDI Page.

In addition, the Smart tool provides a simple way for adding new corpora to the default set. This can be done by introducing a new sub-directory named with the name of the corpus containing a set of files 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

```c
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 presents 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 directory `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”.

### 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 engines to render webpages and execute JavaScript code. Figure 4 shows a screenshot of the SmartGUI where reporting the average
```c
#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.

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.

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.
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.

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 the 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 increase when the length of the pattern increases. Almost all efficient algorithms are based on a sliding window approach whose size is at most as long as the length of the pattern. This behavior is evident, for instance, in 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 automata show drastic decrease in performances in the case of longer patterns. This is due to the size explosion of the underlying automaton and of the corresponding 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. However, algorithms are designed to be extremely efficient in the case of long patterns, thus this class of algorithms can be hard to adapt in flexibility.

Some string matching algorithms turn out to maintain performances for short patterns. This is the case of the SSECP algorithm whose performances remain good regardless of the length of the pattern increases. This is true also of the EPSM algorithm whose flexibility is obtained only by combining different algorithms, depending on the length of the pattern.

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 various ways 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 Kulekci and Dr. Arseny Kapoulckine for helpful comments and suggestions in improving the SMART frame-
work. We wish also to thank the authors of several string matching algorithms for having provided their codes for inclusion in SMART.

References


