Exact string matching animation in Java[1]

Christian Charras – Thierry Lecroq

LIR (Laboratoire d'Informatique de Rouen) and
ABISS (Atelier Biologie Informatique Statistiques Socio-linguistique)
Faculté des Sciences et des Techniques
Université de Rouen
76128 Mont Saint-Aignan Cedex, France

e-mail: {Christian.Charras,Thierry.Lecroq}@dir.univ-rouen.fr

Abstract. We present an animation in Java for exact string matching algorithms [4]. This system provides a framework to animate in a very straightforward way any string matching algorithm which uses characters comparisons. Already 27 string matching algorithms have been animated with this system. It is a good tool to understand all these algorithms.

Key words: Exact string matching, animation, Java

1 Introduction

Pattern matching is a very important field in computer science as much from a theoretical viewpoint as from a practical one. It occurs for instance in text processing, speech recognition, information retrieval, and computational biology. It also provides challenging theoretical problems. For a large number of programs, the techniques used to match a pattern constitute a high percentage of their total work. It is then important to design very efficient algorithms. Understanding the existing algorithms is helpful to achieve this goal. It seemed to us very interesting to offer a tool to visualize easily string matching algorithms.

String matching is a special case of pattern matching where the pattern is set up by a finite sequence of characters. It consists in finding one, or more generally, all the occurrences of a word x of length m in a text y of length n. Both x and y are built over the same alphabet Σ.

The best way to understand how a string matching algorithm works is to imagine that there is a window on the text. This window has the same length as the word x. It is first aligned with the left end of the text y, then the string matching algorithm scans if the symbols of the window match the symbols of the word (this specific work is called an *attempt*). After each attempt, the window (and the word) is shifted to the right over the text until it goes beyond the end of the text. A string matching algorithm is then a succession of attempts and shifts. The aim of an efficient algorithm is to minimize the work done during each attempt and to maximize the length of the

[1]This work was partially supported by the project "Informatique et Génomes" of the french CNRS.

shifts. To achieve this, most of the string matching algorithms preprocess the word before the searching phase. All the different string matchings algorithms differ both in the way they compute the attempts (from left to right, from right to left or from other specific orders) and in the way they compute the shifts.

Numerous solutions to the string matching problem have been designed (see [6] and [11]). The two most famous are the Knuth-Morris-Pratt algorithm [9] and the Boyer-Moore [2]. There exist then a large number of algorithms using various methods. It is interesting to have a tool to understand them. There exist some general-purpose visualization systems (see [10] and [3]). These systems have been developped for X Window. Such a system, running for X Window and dedicated to string matching, has been developed by Baeza-Yates and Fuentes [1]. Some specialized systems are accessible directly through the World Wide Web (see [7], [8], [12] and [13]). All of these systems enable only to vizualize the very classical string matching algorithms and usually they do not permit to keep trace of the history of the search phase. Our system offers to the users the possibility to follow the running of a large choice of string matching algorithms very easily through the World Wide Web.

This article is organized as follows: Section 2 described how the system operates and Section 3 described how the system is written and how to animate a new string matching algorithm.

2 The environment

The user can choose among the 27 string matching algorithms already implemented. For each algorithms there is a button (see Figure 1). If one clicks on a button, a window appears (see Figure 2). In that window the user can then enter a text and a word (default text is `gcatcgcagagagtatacagtacg` and default word is `gcagagag`). The text and the word alphabet is restricted to the lower case letters. A button enables then the user to start the search and another button to stop it at any time. The search phase is then shown attempt by attempt: for one attempt the text is displayed and the word, which all characters are materialized by a dot, is aligned with the relevant position in the text. In each attempt the different character comparisons are shown in the following way:

- matches are shown by displaying the word letter in upper case;
- mismatches are shown by displaying the word letter in lower case.

An occurrence of the word in the text is shown by displaying the corresponding text characters in upper case. At the end of the search phase, the system gives the number of attempts and the number of character comparisons performed during the search phase (see Figure 2).

3 The model

The system is written in Java. It is dedicated mainly to exact string matching algorithms but is easily extensible to a large family of algorithms. Moreover it is completely straightforward to implement any string matching algorithms providing that it is written in a specific way.

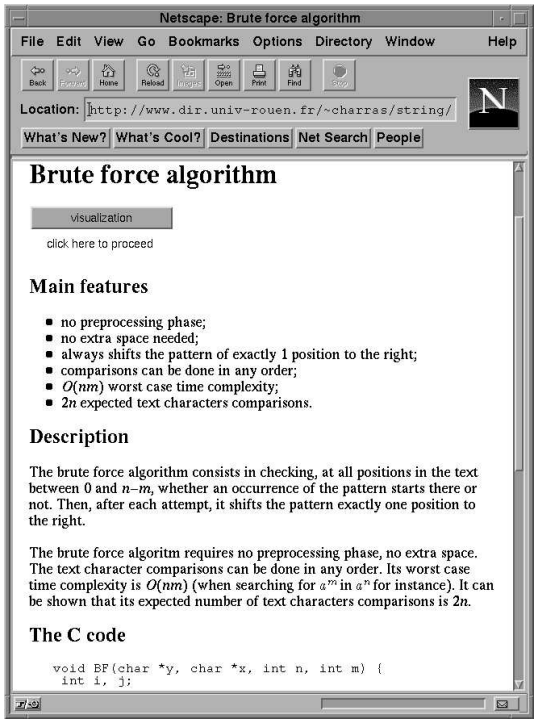

Figure 1: The visualization button for the Brute Force algorithm.

Let us first describe the different structures used by the system. The different buttons for each string matching algorithms are dealt by a class called `AppletButton2` which inheritance graph is shown Figure 3 (`AppletButtoni` is an applet with i inputs for $0 \leq i \leq 3$).

The windows displaying the search phases are dealt by a class which name is `ProgramTextWindow2` which inheritance graph is shown Figure 4.

All the different string matching algorithms inherit of a class which name is `ProgramSPM` which inheritance graph is given Figure 5.

In `ProgramSPM` the different following methods are declared:

- `showAttemptAt(i)`: this method displays m dots below the position i in the text;
- `EqCharsAt(i,j)`: this method tests, and displays the word character accordingly, if there is a match between characters y_i and x_j;
- `NotEqCharsAt(i,j)`: this method tests, and displays the word character accordingly, if there is a mismatch between characters y_i and x_j;
- `showMatch(i)`: this method displays a full match of the word at position i in the text;
- `showComparisons()`: this method displays the number of attempts and the number of character comparisons at the end of the search phase.

The word x, its length m, the text y and its length n are all attributes of the class `programSPM`.

Figure 2: The window for the Brute Force algorithm after a run.

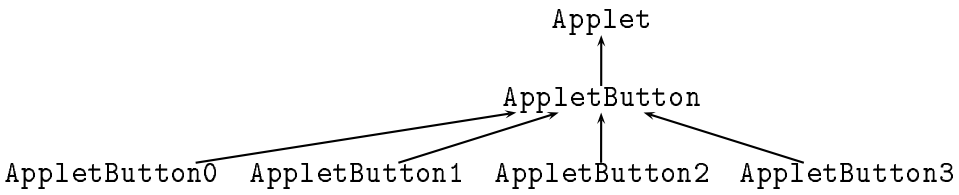

Figure 3: Inheritance graph of the button class.

The animation of a string matching algorithm is very easy if the begin of each attempt (`showAttemptAt`), the character comparisons (`EqCharsAt` or `NotEqCharsAt`) and the report of a full occurrence (`showMatch`) are clearly identified and separated from any other instruction.

Thus the translation of the Brute Force string matching algorithm (see Figure 6) is very straightforward (see Figure 7).

And for a more complicated algorithm as for the Colussi algorithm [5] it is as simple (see Figure 8 and 9).

4 Concluding Remarks

We have presented a system which is able to animate exact string matching algorithms. A demo package is available at the following address:

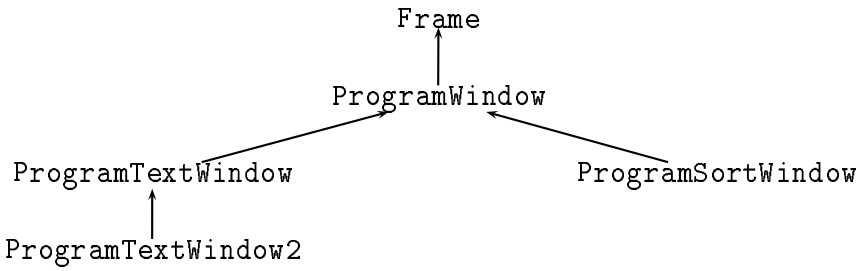

Figure 4: Inheritance graph of the window class.

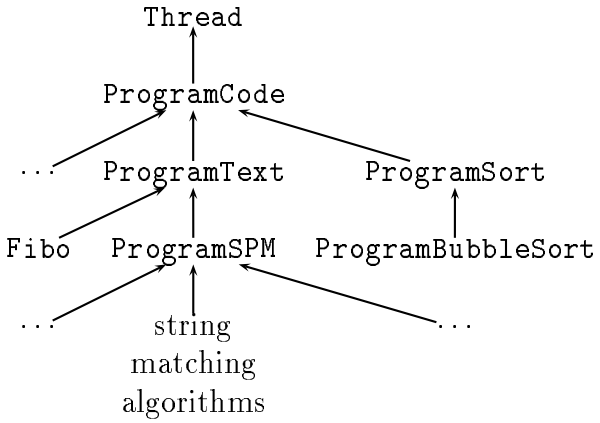

Figure 5: Graph inheritance for the string matching algorithms.

```
void BF(char *y, char *x, int n, int m) {
   int i, j;

   for (i=0; i <= n-m; i++) {
      j=0;
      while (j < m && y[i+j] == x[j]) j++;
      if (j >= m) OUTPUT(i);
   }
  }
```

Figure 6: Brute Force string matching algorithm in C.

```
import lirdir.aptk.InterruptionException;
import lirdir.progtext.ProgramSPM;

public final class ProgramBruteForce extends ProgramSPM {

    public void MAIN() throws InterruptionException{
        int i, j;
        for (i=0; i <= n-m; i++) {
            showAttemptAt(i);
            j = 0;
            while (j < m && EqCharsAt(i+j,j)) j++;
           if (j >= m) showMatch(i);
        }
        showComparisons();
    }
}
```

Figure 7: Brute Force string matching algorithm in Java.

```
   void COLUSSI(char *y, char *x, int n, int m)
   {
    int i, j, right, last, nd, h[XSIZE], next[XSIZE], shift[XSIZE];

    PRE_COLUSSI(x, m, h, next, shift, &nd);

    /* Searching */
    i=0;
    right=0;
    last=-1;
    while (i <= n-m) {
       j=right;
       while (j < m && last < i+h[j] && y[i+h[j]] == x[h[j]]) j++;
       if (j >= m || last >= i+h[j]) {
          OUTPUT(i);
          j=m;
       }
       if (j > nd) last=i+m-1;
       i+=shift[j];
       right=next[j];
    }
   }
```

Figure 8: Colussi string matching algorithm in C.

```
import lirdir.aptk.InterruptionException;
import lirdir.progtext.ProgramSPM;

public final class ProgramColussi extends ProgramSPM {

    public void MAIN() throws InterruptionException {
      int i, j, right, last, nd;
      int h[] = new int[m+1];
      int next[] = new int[m+1];
      int shift[] = new int[m+1];
      nd = PRE_COLUSSI(h, next, shift);
      /* Searching */
      i=0;
      right=0;
      last=-1;
      while (i <= n-m) {
          showAttemptAt(i);
          j=right;
          while (j < m && last < i+h[j] && EqCharsAt(i+h[j],h[j])) j++;
          if (j >= m || last >= i+h[j]) {
             showMatch(i);
             j=m;
          }
          if (j > nd) last=i+m-1;
          i+=shift[j];
          right=next[j];
      }
      showComparisons();
    }
}
```

Figure 9: Colussi string matching algorithm in Java.

`ftp.dir.univ-rouen.fr/pub/ESMAJ/esmaj.zip`
and can be consulted at
`http://www.dir.univ-rouen.fr/~charras/esmaj/`.

We have shown how it is easily possible to animate new string matching algorithms providing that they are written in a given form. This system can easily be extended to animate other class of algorithms. It seems quite obvious that animating approximate string matching algorithms would just need a few efforts. Some sort algorithms and some graph algorithms have already been animated with the same principles.

References

[1] R.A. Baeza-Yates and L.O. Fuentes. A framework to animate string algorithms. *Inform. Process. Lett.*, 59(5):241–244, 1996.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.

[3] M.H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *Proceedings of the IEEE Workshop on Visual Languages*, 1991.

[4] C. Charras and T. Lecroq. Exact string matching algorithms, 1996.
URL:`http://www.dir.univ-rouen.fr/~charras/string/`

[5] L. Colussi. Correctness and efficiency of the pattern matching algorithms. *Inform. Comput.*, 95(2):225–251, 1991.

[6] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[7] A. Cássia Rossi de Almeida. Smaa: string matching algorithm animation.
URL:`http://www.dcc.ufmg.br/~cassia/english_version_smaa.html`

[8] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.
URL:`http://www.cgc.cs.jhu.edu/~goodrich/dsa/11strings/demos/pattern/`

[9] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[10] J. T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Trans. Comput.*, 23(9):27–39, 1990.

[11] G. A. Stephen. *String searching algorithms*. World Scientific Press, 1994.

[12] M. Takeda. Demonstration of naive, KMP, and BM pattern matching algorithms, and their variations.
URL:`http://www.i.kyushu-u.ac.jp/~takeda/PM_DEMO/e.html`

[13] K. A. Zaman. Illustrated pattern matching.
URL:`http://www.cs.columbia.edu/~zkazi/proj.html`