Searching of all Occurences of a Word in a String

OCTAVIAN DOGARU and ROXANA DOGARU
West University of Timisoara
Bd.V.Parvan,nr.4, Timisoara,1900,
ROMANIA

Abstract. This paper presents a string search algorithm. The method searches to find all occurences of a word p of m characters in a string s of n characters, $0<m\leq n$. The upper bound of the number of comparisons to determine that p is not in s, in the most unfavourable case, is m(n-m+1).
Key words: string, pattern, searching, all occurences, algorithm.
CSCC'99 Proceedings: - Pages 1503-1507

1 Introduction

Let s[1..n] be a string of n characters and p[1..m] a word or pattern of m characters, $0<m\leq n$, and the task is to find all occurences of p in s. The word and the string are both built on the same alphabet Σ.

There are a lot of algorithms which approach the problem of finding all occurences of a pattern in a text. Many of them are based of the precompiling of the pattern p. Perhaps the Knuth-Morris-Pratt[9] and Boyer-Moore[1] algorithms are the most known. In the paper [2] and the references cited there are presented algorithms of linear time complexity with small constants. They use the idea of precompiling of the pattern p.

In a brute-force algorithm(BF) for string search initially, the pattern p is alligned with the left end of the text s. One compares successively p_1 with s_i, i=1,2,...,n-m+1. If there exists no match of p_1 with s_i, i=1,2, ...,n-m+1 then 'p is not in s' and the process is terminated. Now let s_i be the first occurence of p_1. Then one compares respectively p_2 with s_{i+1}, p_3 with s_{i+2},...,p_m with s_{i+m-1}. If all p_j match with s_{i+j-1}, j=1,2,...,m then p is the first occurence of p in s and the process of searching p in the rest of s is resumed.

2 The main result

Starting from the method presented above we propose the following algorithm of string search. In fact, it is an algorithm which examines text characters only in a window of size m, the length of pattern, which contains the character s_i that matches with p_j the last character which has produced a mismatch, the window sliding to the right.

An algorithm to determine the first occurence of p in s, based on the same idea, has been presented in [7].

Our algorithm has a time complexity of m(n-m+1) in the most unfavourable case and it does not use a suplimentary array. It is different of one presented in [6].

With p and s aligned to the left ends our method is the following.

First one compares successively p_1, p_2, ..., p_m with corresponding s_1,s_2,...,s_m. If there exists all matches then 'p is in s'. The process of searching is resumed with p_1 and s_{m+1} if the latest exists. In this algorithm, once the occurence of p exists in s, the window is shifted to right over the text exactly with m characters. We give this interpretation because, if a word is found in a text it may be erased if one wishes that.

In the process of comparison at above step, we assume that p_1, p_2,..., p_{j-1} respectively match with s_1, s_2,... ,s_{j-1} but p_j is the first character of the pattern p which produces a mismatch of s_j. Then one searches the first occurence of p_j in the right part of s_j between s_{j+1} and s_{n-m+j}. If p_j is not in this substring then 'p is not in s' and the process stops. Therefore a new search of p continues with p_j and not with p_1 as in a brute-force algorithm presented,for example, in [10] and in [12] at page 60.

Now let s_i be the occurence of p_j in this substring. One verifies if the right part of p_j that is $p_{j+1}p_{j+2}...p_m$ match with the right part of s_i, that is $s_{i+1}s_{i+2}...s_{i+m-j}$. There appear the situations:

1)if the right parts mismatch then let the index k be, $1\leq k\leq m-j$, for which $p_{j+k}\neq s_{i+k}$ and then the process of searching will be resumed with the new index values i:=i+k+1 for s_i and j:=j+k for p_j this is a slide to right of lenght k;

2)if the right parts match then one compares the left parts of p_j and s_i. There are the cases:

2i)if the left parts match too then p is in s and the process of searching is resumed with i:=i+m-j+1 for s_i and j:=1 for p_j in this order;

2ii)if the left parts mismatch then the process of searching p_j is resumed with the same j and s_{i+1}.

If i>n-m+j then the process stops.

The complet method described above is written now as the procedure OD1, presented in SPARKS language, slight modified(this language is described in [8]). It is following

```
procedure OD1(m,n,p,s)
integer m,n,i,j,k; boolean f; char p(1:m), s(1:n)
f:= false; i:=1; j:=1;
loop
  while (j<=m) and (p(j)=s(i)  do j:=j+1;i:=i+1
  repeat;
  if j>m  then write('p is in s, i=',i-j+1);
               f:=true; j:=1; cycle
  endif;
       //there exists j such that p(j)≠ s(i)//
 1:i:=i+1
  while (i<=n-m+j) and (p(j)<>s(i)) do i:=i+1
  repeat
  if i>n-m+j  then exit endif;
          //there exists i such that p(j)=s(i)//
          //one tests the neighbours of p(j) with //
          // the neighbours of s(i) //
  k:=1;
  while (k<=m-j) and (p(j+k)=s(i+k)) do
           k:=k+1
  repeat
  if k<=m-j
      then //right parts mismatch, it exists k //
           //such  that p(j+k) ≠ s(i+k)//
                j:=j+k; i:=i+k;  goto 1
      else //the right parts of p(j) and s(i) match,//
          // one verifies the left parts //
           k:=1;
           while (k<j) and (p(k)=s(i-j+k)) do
                 k:=k+1
           repeat
           if k=j then //left parts match too//
                 write('p is in s, i=',i-j+1);
                 f:= true;
                 i:=i+m-j+1; j:=1; cycle
                 else //right parts match  but left//
                    //parts mismatch,//
                 //there exists k such that p(k) ≠ s(i-j+k)//
                     goto 1 //new searching of p(j)//
                        //is resumed //
           endif
  endif
until i>n-m+j  repeat;
if not f  then write('p is not in s') endif
endOD1.
```

In SPARKS language the statement **cycle** causes a transfer of control to the closing phrase of the innermost iteration statement which contains it and the command **exit** causes a transfer of control to the first statement after the innermost looping statement which contains it.

Theorem 1. ***The algorithm OD1 works correctly***.

Proof. The partial correctness of this algorithm can be shown by developing a proof table where we shall insert a set of assertions between the statements of the program and starting from the preconditions one arrives to the postconditions. The justifications are based on the application of logical equivalences and the rules of inference to the sequence of Pascal statements[8 endif]. These are:

i) the *assignement* rule of inference

{ P(e) } v:=e { P(v) }

ii) the *conditional* rule of inference

a) $\dfrac{\{P \wedge B\}\ s\ \{Q\} \qquad P \wedge \sim B \Rightarrow Q}{\{P\}\ \text{if B then s}\ \{Q\}}$ b) $\dfrac{\{P \wedge B\}\ s1\ \{Q\} \qquad \{P \wedge \sim B\}\ s2\ \{Q\}}{\{P\}\ \text{if B then s1 else s2}\ \{Q\}}$

iii) the *loop* rule of inference

a) $\dfrac{\{inv \wedge B\}\ s\ \{inv\}}{\{inv\}\text{while B do s}\{inv \wedge \sim B\}}$ b) $\dfrac{\{inv \wedge B\}s\{inv\}}{\{inv\}\text{repeat s until B}\{inv \wedge B\}}$

where P,Q denote propositions, B-Boolean expression, inv-the invariant of the loop and s, s1, s2-are statements.

```
procedure OD1( m,n,p,s)
integer m,n,i,j,k; boolean f; char p(1:m),s:(1:n);
```

{ pre:input=$(p_1,p_2,...,p_m)\wedge(s_1,s_2,...,s_n)\wedge n\geq m>0\wedge$ $\forall i\in\{1,2,...,n\}$:$s_i$ are characters$\wedge\forall$ $j\in\{1,2,...,m\}$:p_j are characters}

f:= **false**; i:=1; j:=1;

```
loop
   {inv:1≤j≤m}
   while (j<=m) and (p(j)=s(i)  do
      {inv:∀h∈{1,2,...,j-1}:p_h=s_h∧1≤j,i≤m+1}
            j:=j+1;i:=i+1
   repeat
{∀h∈{1,2,...,j-1}:p_h=s_h ∧(j>m∨p_j ≠s_i}
   if j>m  then write('p is in s, i=',i-j+1);
               f:=true; j:=1;
         {output=i-j+1}{j>m∧f=true ∧ j=1}
                  cycle
   endif;
{f=false∧j≤m∧ p_j≠s_i}
   1:i:=i+1
         {(1≤i≤n-m+j∧ p_j ≠ s_i) ∨ (i>n-m+j)}
   while (i<=n-m+j) and (p(j)<>s(i)) do
         {inv:p_j ≠s_{i-1} ∧ i≤n-m+j}
         i:=i+1
   repeat;
{(p_j≠s_{i-1} ∧ i≤n-m+j)∧((i>n-m+j)∨(i≤n-
   m+j∧p_j=s_i))}
   if (i>n-m+j) then {p_j ≠s_{n-m+j}}
                     exit
   endif;
        {i≤n-m+j∧p_j ≠s_{i-1} ∧pj=si}
      k:=1;
      while (k<=m-j) and (p(j+k)=s(i+k)) do
   {inv:∀h∈{1,2,...,k-1}:p_{j+h}=s_{i+h}  ∧
1≤k≤m-j+1}
            k:=k+1
      repeat;
{∀h∈{1,2,...,k-1}:p_{j+h} =s_{i+h} ∧
(1≤k≤m-j+1)∧((k>m-j ∨ p_{j+k} ≠s_{i+k})}

      if k<=m-j  then
{∃k∈{1,...,m-j}:p_{j+k} ≠s_{i+k}}

            j:=j+k; i:=i+k;  goto 1
      else
{∀k∈{1,...,m-j}:p_{j+k}=s_{i+k}  ∧ i ≤n-m+j∧1≤j≤m}
            k:=1;
         while (k<j) and (p(k)=s(i-j+k))  do
      {inv: ∀h∈{1,2,...,k-1}:p_h=s_{i-j+h} ∧1≤k≤j}
               k:=k+1
         repeat;
  {∀k∈{1,2,...,j-1}:p_k=s_{i-j+k}∧(k=j)∨( p_k≠s_{i-j+k}}
         if k=j  then
{∀k∈{1,2,...,m}:p_k=s_{i-j+k} ∧1≤j≤m∧1≤i≤n-m+j}
            write('p is in s, i=',i-j+1); f:=true;
                i:=i-j+m+1; j:=1; cycle
                   else
{∃ k∈{1,2,...,j-1}:p_k≠s_{i-j+k} ∧1≤ j≤m∧1≤i≤n-m+j}
                         goto 1
             endif
      endif
 until i>n-m+j  repeat;
          {1≤ j ≤m+1∧ i>n-m+j}
             {f=true ∨ f=false}
if not f  then write('p is not in s') endif
          {post: output=θ}
endOD1
```

Number of comparisons. Teoretically, to find that 'p is not in s', in the most unfavourable case, without loss in generality, we suppose that j=m, that is, $p_1=s_1$, $p_2=s_2$, ...,$p_{m-1}=s_{m-1}$ but $p_m\neq s_m$. In this case one compares successivelly p_m with $s_{m+1},s_{m+2},...,s_n$ which are n-m characters and for every, $p_m=s_i$,i=m+1,m+2, ...,n. Then one assumes that for every i, i=m+1,m+2,...,n the left neighbours of s_i and p_m are $p_h=s_{i-m+h}$, h=1,2,...,m-2 but $p_{m-1}\neq s_{i-1}$. There exists m-1 left neighbours. Hence the maximum number of comparisons, in the worst case, is

$$N_{max} = m+m(n-m) = m(n-m+1).$$

This is the complexity of our algorithm. For m=1, Nmax= n. For m=2, Nmax= 2n-2 that is the KMP complexity[9]. For m=3 or m=4 it is obtained BM complexity[1].

3. Profiling

We realized a comparison between a classical brute-force algorithm(BF) presented in [10] and [12] and the OD1 algorithm for different values of m and the value of n=7000. We have used the following version of BF(brute-force) algorithm presented in [10]:

```
ALGORITHM BF;
 begin
    i:=0
   while i<=n-m  do
       j:=1;
      while (j<=m) and (p(j)=s(i+j)) do
         j:=j+1
      repeat;
      if j>m  then write(i+1)  endif;
      i:=i+1;
  repeat
 endBF.
```

The experiments have been realized on alphabets of two sizes for every of the two methods: OD1 and BF. One used a program written in Turbo Pascal 7.0.

Alphabet of size 94

The alphabet Σ is composed of 94 different characters. The length of the text s was n=7000 characters.

There was generated sequences of m and n integer random numbers between 33 and 126 and then p and s have been considerated the ASCII character arrays corresponding to these sequences of numbers. For the same s one has calculated the average of the times found for 100 repetitions for every value of m. The results are presented in the Table 1. For m>3 the algorithm OD1 is faster than BF algorithm.

Alphabet of size 26

For Σ made of 26 characters, Σ={a..z}, with the same consideration on s and p the results are presented in the Table 2. For this alphabet, for m>3, the OD1 is faster than BF algorithm.

4 Conclusions.

In these two cases, the alphabets are great, of 94 and 26 characters res-pectively. Excepting the cases where m=2 and m=3, that is, the patterns have a very little lengths, the average running times of the algorithm OD1 are more little than the average running times of the algorithm BF therefore OD1 algorithm is faster than BF algorithm. It remains to analyse the running times and for other sizes of alphabets.

This method may be slight improuved if the neighbours of p_j and s_i are compared in a single loop with the index k between 1 and m.

Table 1. Running times for an alphabet of size 94

m	**OD1**	**BF**	**m**	**OD1**	**BF**
2	0.61	0.26	20	**0.28**	0.38
3	0.40	0.34	40	**0.15**	0.49
4	**0.28**	0.31	80	**0.22**	0.27
5	**0.36**	0.36	160	**0.28**	0.56
6	**0.23**	0.41	320	**0.27**	0.39
7	**0.27**	0.44	500	**0.27**	0.33
8	**0.24**	0.54	1000	**0.33**	0.38
9	**0.22**	0.50	2000	**0.26**	0.27
10	**0.26**	0.44	4000	**0.16**	0.27

Table 2. Running times for an alphabet of size 26

m	**OD1**	**BF**	**m**	**OD1**	**BF**
2	3.82	0.46	20	**0.28**	0.38
3	0.62	0.38	40	**0.15**	0.49
4	**0.28**	0.46	80	**0.22**	0.27
5	**0.21**	0.49	160	**0.28**	0.56
6	**0.35**	0.38	320	**0.27**	0.39
7	**0.33**	0.44	500	**0.27**	0.33
8	**0.23**	0.49	1000	**0.33**	0.38
9	**0.28**	0.43	2000	**0.26**	0.27
10	**0.28**	0.45	4000	**0.16**	0.27

References:

[1] R.S.Boyer, J.S.Moore, A fast string searching algorithm, *Com. ACM*, 20, 10(1977), 762-772

[2] R. Cole, R. Hariharan, Tighter Bounds on The Exact Complexity of String Matching, *Procc.33rd Symp.on Foundation of Computer Sci*., (1992), 600-609

[3] R. Cole, Tight Bounds on Complexity of the Boyer-Moore string matching algorithm, *SIAM J. Computing*, 23, 5(1994), 1075-1091

[4] R. Cole, R. Hariharan, M. Paterson, U.Zwick, Tighter Lower Bounds on the exact Complexity of String Matching, *SIAM J. Computing*, 34, 1(1995), 30-45

[5] M. Crochemore, T.Lecroq, Tight bunds on complexity of the Apostolico-Giancarlo algorithm, *Information Processing Letters*, 63,1997, 195-203

[6] O. Dogaru, On all occurences of a word in a text, *Proceedings on Conference PSCW'' 98 (Prague Stringology Club Workshop)*, September 3-4, 1998, 51-57, Prague, Czech Republic

[7] O. Dogaru, A pattern matching algorithm, *Proceedings on the Conference CSC'98 (Circuits, Systems and Computers)*, October 26-28, 1998, 2, 884-887, Pireus, Greece

[8] E. Horowitz, S. Sahni, Foundamentals of Computer Algorithm, *Computer Science Press*, 1983, 626 pp

[9] D. E. Knuth, J. H. Morris, V. R. Pratt, Fast pattern matching in string, *SIAM J. Comput*, 6, 2(1977), 323-449

[10] T. Lecroq, Experimental Results on String Matching Algorithm, *Software-Practice and Experience*, 25(7), 727-765, 1995

[11] A. B. Tucker, W. J. Bradley, R. D. Cupper, D. K. Garnick, Fundamentals of Computing I, *McGRAW-HILL, INC*, 1992, 400 pp
[12] N. Wirth, Algorithm and Data Structures, *Prentice-Hall*, N.J.(1986), 288 pp