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≤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 occurrences, 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≤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 aligned with the left end of the text s. One compares successively p₁ with s₁, s₂,...,sₙ-m+1. If there exists no match of p₁ with sᵢ, i=1,2,...,n-m+1 then ‘p is not in s’ and the process is terminated. Now let si be the first occurence of p₁. Then one compares respectively p₂ with sᵢ₊₁, p₃ with sᵢ₊₂,...,pₘ with sᵢ₊ₘ₋₁. If all pⱼ match with sᵢ₊ⱼ, 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 sᵢ that matches with pⱼ the last character which has produced a mismatch, the window sliding to the right.

An algorithm to determine the first occurrence 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 p₁, p₂, ..., pₘ with corresponding s₁, s₂,...,sₘ. If there exists all matches then ’p is in s’. The process of searching is resumed with p₁ and sₙ₊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 p₁, p₂,..., pₘ respectively match with s₁, s₂,..., sₘ but p₁ is the first character of the pattern p which produces a mismatch of sᵢ. Then one searches the first occurence of pⱼ in the right part of sᵢ between sᵢ₊₁ and sₙ₊ₘ₋₁. If pⱼ is not in this substring then ‘p is not in s’ and the process stops. Therefore a new search of p continues with p₁ and not with p₁ as in a brute-force algorithm presented,for example, in [10] and in [12] at page 60.

Now let sᵢ be the occurence of p₁ in this substring. One verifies if the right part of p₁ that is p₁₋₁p₁₋₂...pₘ match with the right part of sᵢ that is sᵢ₊₁sᵢ₊₂...sᵢ₊ₘ₋₁. There appear the situations:
1)if the right parts mismatch then let the index k be, 1≤k≤m-j, for which pⱼ₊₁≠sᵢ₊k and then
the process of searching will be resumed with the new index values \(i:=i+k+1\) for \(s\) and \(j:=j+k\) for \(p\).

2) if the right parts match then one compares the left parts of \(p\) and \(s\). 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 \(s\) and \(j:=1\) for \(p\) in this order;

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

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

The complete 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=1;
      //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
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 assignment rule of inference

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

ii) the conditional rule of inference

```
a) { P∧ B} s {Q}          b) { P∧ B} s1 {Q}
P∧∼B⇒Q                             P∧∼B} s2  {Q}
--------------------------                  ------------------
{P} if B then s {Q}            {P} if B then s1 else
                                          s2{Q}
```

iii) the loop rule of inference

```
a){ inv∧B} s { inv}              b){ inv∧B} s1 { inv}
-------------------------------      ----------------------
{ inv}while B do s            { inv∧ B} { inv∧ B}
{ inv∧∼ B}                             { inv∧ 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=(p1,p2,...,pm)∧(s1,s2,...,sn)∧n≥m>0∧
  ∀i∈{1,2,...,n}:si are characters∧∀ j∈{1,2,...,m}:pj are characters}
f:= false; i:=1; j:=1;
```
loop
  {inv:1\leq i\leq m}
  while (j=m) and (p(j)=s(i)) do
   {inv:\forall i\in\{1,2,...,j-1\}: p_i=s_i \land 1\leq j \leq m+1}
   j:=j+1; i:=i+1
   goto 1
  end while
  goto 1
end loop

end loop

Number of comparisons. Theoretically, 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=s_m but p_{m+1} \neq s_m. In this case one compares successively p_m with s_m, s_{m+1}, s_{m+2}, ..., s_{n-1}, which are n-m characters and for every, p_m=s_{i=m+1}, p_m=s_{i=m+1}, ...,n. Then one assumes that for every i, i=m+1,m+2,...,n the left neighbours of s_i are p_1,s_2,...,p_{i-1} which for every i are p_{i-1}=s_{i-1}, h_{i-1}=1,2,...,m-2 but p_{i-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, N_{max}= n. For m=2, N_{max}= 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) do
      if (p(j)=s(i+j)) then
        j:=j+1
      end if
      k:=j+1
    end while
    k:=k+1
  end while
  goto 1
end algorithm BF.
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 considered 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 respectively. 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 improved if the neighbours of p_j and s_i are compared in a single loop with the index k between 1 and m.

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

### References:


