Improved String Matching with k Mismatches

Zvi Galil [1,2] and Raffaele Giancarlo [2]

Given a text string $t_0 t_1 ... t_{n-1}$, a pattern string $p_0 p_1 ... p_{m-1}$ and an integer k, $k \leq m \leq n$, we are interested in finding all occurrences of the pattern in the text with at most k mismatches, i.e. with at most k locations in which the pattern and the text have different symbols.

Recently, an efficient algorithm for such a problem has been devised by [2]. Its time performance is $O(k(m\log m+n))$ and it uses $O(k(m+n))$ space. Here we present a compact version of their algorithm, achieving a time performance of $O(m\log m+kn)$ for general alphabets and of $O(m+kn)$ for alphabets whose size is fixed. Our algorithm uses $O(m)$ space. The data structure that we use is the *suffix tree* [1] of the pattern modified in order to support the *static lowest common ancestor* algorithm (*LCA* for short) given in [5]. In more recent versions of [2] the authors used the suffix tree for the problem of string matching with k differences [4] but not for the problem of string matching with k mismatches [3].

The *suffix tree* T of the string $p_0 p_1 ... p_{m-1}$ is a digital search tree of $O(m)$ nodes containing all suffixes of that string. Each leaf of T is labeled with a distinct integer j so that the path from the root of T to leaf (labeled) j corresponds to suffix $p_j p_{j+1} ... p_{m-1}$. A fast algorithm for the construction of T is reported in [1] and it takes $O(m\log m)$ time for general alphabet. When the alphabet has a fixed size, it takes $O(m)$. The LCA algorithm is used to quickly find the length of the longest common prefix between suffixes $p_i ... p_{m-1}$ and $p_j ... p_{m-1}$. Indeed, we assume that the query $LCA(T,i,j)$ returns such a length. We recall that the *LCA* algorithm is linear in the number of queries [5]. The algorithm can be informally described as follows.

The algorithm tests, in increasing order, all positions of the text in order to locate occurrences of the pattern in the text with at most k mismatches. Let i_{new} be the current position to be tested and let $A_{new}[1,k+1]$ be an array in which the (up to) first $k+1$ mismatching text positions are stored. Namely, $A_{new}[s] = r$ if $t_r \neq p_{r-i_{new}}$ is the *s-th* mismatch between $t_{i_{new}} ... t_{i_{new}+m-1}$ and $p_0 ... p_{m-1}$. Let $i_{old} < i_{new}$ be such that (up to at most $k+1$ mismatches) $t_{i_{old}} ... t_j = p_0 ... p_{j-i_{old}}$, with j *maximal*. Let $A_{old}[1,k+1]$ be an array containing, in increasing order, the S mismatching positions between $t_{i_{old}} ... t_j$ and $p_0 ... p_{j-i_{old}}$. A_{old} for i_{old} is as A_{new} for i_{new}.

In order to test an occurrence of $p_0 ... p_{m-1}$ at i_{new}, we first compare the string $t_{i_{new}} ... t_j$ with $p_0 ... p_{j-i_{new}}$ by using T (the suffix tree of the pattern) and the S positions in A_{old} (**Procedure MERGE**). Indeed, assume that the first $q \leq k$ mismatches between $t_{i_{new}} ... t_j$ and $p_0 ... p_{j-i_{new}}$ have been found by using the first $s-1 < S$ entries of A_{old}, i.e. it has been found that prefixes $t_{i_{new}} ... t_{i-1}$ and $p_0 ... p_{i-i_{new}-1}$ have q mismatching positions. Then, $A_{old}[s]$ can be used to locate the $q+1$-*st* mismatch, i.e. the first mismatch between suffixes $t_i ... t_j$ and $p_{i-i_{new}} ... p_{j-i_{new}}$. Let $l = LCA(T,i-i_{new},i-i_{old})$ and notice that $i \leq A_{old}[s]$. The following three cases may arise:

[1]Department of Computer Science, Tel-Aviv University

[2]Department of Computer Science, Columbia University. Work supported in part by NSF Grants MCS-830-3139 and DCR-85-11713

1. $i+l<A_{old}[s]$. By noting that $A_{old}[s]$ is the first mismatch between $t_i...t_j$ and $p_{i-i_{old}}...p_{j-i_{old}}$ and that $p_{i-i_{new}+l} \neq p_{i-i_{old}+l}$, with $p_{i-i_{new}}...p_{i-i_{new}+l-1} = p_{i-i_{old}}...p_{i-i_{old}+l-1}$, we can conclude that $t_{i+l} \neq p_{i-i_{new}+l}$. Thus, $i+l$ can be stored in $A_{new}[q+1]$ and $A_{old}[s]$ is still usable to detect mismatches (and we do not change s).
2. $i+l=A_{old}[s]$. An analysis similar to (1) shows that $t_{A_{old}[s]}$ must be compared with $p_{i-i_{new}+l}$. $A_{old}[s]$ is stored in $A_{new}[q+1]$ if there is a mismatch. In any case, we increment s ($A_{old}[s]$ is useless from now on) and we continue looking for mismatches starting at $i = A_{old}[s]+1$.
3. $i+l>A_{old}[s]$. An analysis similar to (1) and (2) shows that $t_{A_{old}[s]} \neq p_{A_{old}[s]-i_{new}}$. Thus, $A_{old}[s]$ can be stored in $A_{new}[q+1]$ and we increment s ($A_{old}[s]$ is useless from now on).

Assume now that $Q \leq k$ mismatches have been detected at the time $A_{old}[S]$ becomes useless. If $A_{old}[S]<j$, we still have to compare $t_{A_{old}[S]+1}...t_j$ with $p_{A_{old}[S]-i_{new}+1}...p_{j-i_{new}}$. Observing that $t_{A_{old}[S]+1}...t_j$ matches a suffix of $p_0...p_{j-i_{old}}$, this can be efficiently done by using the *suffix tree* and the *LCA* algorithm (second **while** loop in **Procedure MERGE**).

As soon as it is guaranteed that only $Q \leq k$ mismatches exist between $t_{i_{new}}...t_j$ and $p_0...p_{j-i_{new}}$, it can be directly checked whether $t_{j+1}...t_{i_{new}+m-1}$ and $p_{j-i_{new}+1}...p_{m-1}$ have at most $k-Q$ mismatching positions (**Procedure EXTEND**). In such a case i_{new} is an occurrence of the pattern in the text.

Complexity: It is easily seen that **Procedure EXTEND** scans each character of the text at most once contributing for $O(n)$ time. On the other hand, **Procedure MERGE** takes time proportional to the size of $A_{old}+A_{new}$, that is at most $2k+2$. Since it can be called at most $O(n)$ times, **Procedure MERGE** contributes for $O(kn)$ time. The construction of the *suffix tree* takes $O(m\log m)$ for general alphabets and $O(m)$ for alphabets whose size is fixed [1]. Thus, the total time is, as claimed, $O(m\log m+kn)$ for general alphabets and $O(m+kn)$ for alphabets of fixed size. The space required by the algorithm is $O(m+k) = O(m)$, since the *suffix tree* requires $O(m)$ space.

----

1. P. Weiner. Linear Pattern Matching Algorithms. Symposium on Switching and Automata Theory, IEEE, 1973, pp. 1-11.

2. G.M. Landau and U. Vishkin. Efficient String Matching in the Presence of Errors. Symposium on Fundations of Computer Science, IEEE, October, 1986, pp. 126-136.

3. G.M. Landau and U.Vishkin. "Efficient String Matching with k Mismatches". *Theoretical Computer Science, to appear*

4. G.M. Landau and U. Vishkin. Efficient String Matching with k Differences. Dept. of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel, September, 1985.

5. H.T. Harel and R.E. Tarjan. " Fast Algorithms for Finding Nearest Common Ancestors". *SIAM J. on Computing 13*, 2 (1984), 338-355.

```
PROCEDURE MAIN
begin
 $i_{old} = 0; j = 0; S = 0;$
 for $i_{new} = 0$ to $n-m$ do
    begin
     $q = 0$;  if $i_{new} < j$ then   MERGE($i_{new}, i_{old}, j, S, q$);
     if $q < k+1$ then  [EXTEND($i_{new}, j, q$);  $i_{old} = i_{new}$; $A_{old} = A_{new}$; $S = q$;]
     if $q < k+1$ then print($i_{new}, A_{old}$);
                    {$i_{new}$ is an occurrence of pattern in text; mismatching positions are in $A_{old}$}
    end
 end {MAIN})

PROCEDURE EXTEND($i_{new}, j, q$)
begin
 while ($q < k+1$)  and  ($j-i_{new} < m$)  do
   begin
   $j = j+1$;   if  $t_j \neq p_{j-i_{new}}$  then $q = q+1$; $A_{new}[q] = j$;
   end
end {EXTEND}


PROCEDURE MERGE($i_{new}, i_{old}, j, S, q$)
begin
$s = 1; i = i_{new}$;
while  $i \leq A_{old}[S]$  and  $q \leq k$  do
  begin
  $l = LCA(T, i-i_{new}, i-i_{old})$;
    begin-case
    1. $i+l < A_{old}[s]$ :   $q = q+1$; $A_{new}[q] = i+l$; $i = i+l+1$;

    2. $i+l = A_{old}[s]$ :    if  $t_{A_{old}[s]} \neq p_{i-i_{new}+l}$  then   [$q = q+1$; $A_{new}[q] = A_{old}[s]$]
                          $i = A_{old}[s]+1$; $s = s+1$;


    3. $i+l > A_{old}[s]$ :  $q = q+1$; $A_{new}[q] = A_{old}[s]$; $i = A_{old}[s]+1$; $s = s+1$;

    end-case

  end

while  $q \leq k$  do
  begin
   $l = LCA(T, i-i_{new}, i-i_{old})$
   if  $i+l \leq j$  then  $q = q+1$;  $A_{new}[q] = i+l$;  $i = i+l+1$;
                 else return
  end

end {MERGE}
```