A STATISTICAL TEXT PATTERN MATCHING ALGORITHM

Hussien Hassan Ali
Alexandria University, Faculty of Engineering,
Computers and Automatic Control Department.

ABSTRACT

We present a statistical pattern matching algorithm for text that can utilize the knowledge of the probability distribution of the alphabetic characters in the text under consideration. The algorithm has a pre-processing phase of time complexity on the worst case of order m log m - where m is the length of the pattern - and has a searching phase that on the average does not examine every character in the target text (i.e.sublinear). We develop an upper bound for the number of comparisons in the average case and show that the number of comparisons for this algorithm is less than or equal that of the Boyer-Moore algorithm.

1. INTRODUCTION

The problem of text pattern matching is an old well-known problem which can be summarized as: Given a string of characters S and another target string of characters T, we need to detect the occurrence(s) of S in the target T. The algorithms used to solve this problem are used heavily in information systems for automatic indexing and text categorization tasks, word processors, and all text processing applications. The availability of large amounts of text material in modern information systems and the still unsatisfactory results of query methods that depend on index terms [1], [2] lead to a demand for practical and efficient algorithms for pattern matching.

There are many algorithms in the literature that deal with this problem, for a detailed survey see [3], and [4]. However, two algorithms seem to be the most efficient [4]:The algorithm presented in [5] and the algorithm presented in [6].Throughout this paper, these two algorithms will be referred to as KMP (Knuth-Morris-Pratt) and BM (Boyer-Moore) respectively. Both algorithms use the same basic idea of utilizing the knowledge of the pattern S to enhance the searching process and both have the same worst-case complexity O(n) and a pre-processing time of O(m), where n is the length of the target T and m is the length of the pattern string S. However, they assume nothing about the target text T and do not utilize any information about it.

With modern information systems and the availability of whole text documents in a machine-processable format, a new horizon for text processing is opened. For example, the statistical properties of the characters, words, and other components of the text can be easily calculated and used to enhance the performance of the various operations [7], [8], [9]. In this paper, we propose an algorithm that uses the knowledge of the pattern S and the probability distribution of the characters in the target string T to further enhance the expected number of character comparisons.

The rest of the paper is organized as follows: In section 2, the background and some details of the KMP and BM algorithms are discussed in order to set the stage for the new algorithm. The new statistical algorithm is presented in section 3. The complexity and performance of the new algorithm together with a comparison with other algorithms is presented in section 4. The conclusion is given in section 5 and then the references are given.

2. BACKGROUND

Suppose we have the string pattern $S = s_1 s_2 \ldots s_m$, and the text string $T = t_1 t_2 \ldots t_n$; where $n \geq m$. The most straightforward way to find whether the pattern S occurs in the text T can be summarized as follows:

1- The characters in S are compared against the corresponding characters in T.

2- If a mismatch occurs, shift the string S to the right by one position (i.e. after the first shift, s_1 will correspond to t_2) and repeat the two steps until either the string S is found or the end of the string T is reached.

This method is slow and it needs O(mn) character comparisons in the worst case. Notice that each character of T may be processed more than once (up to m times).

[5] proposed an algorithm (KMP) which is linear and needs only O(m+n) comparisons. The basic idea is to construct a deterministic finite state automaton using the pattern S. Then the characters of T are processed one at a time from left-to-right. The machine will enter an accepting state if it recognizes the pattern S. The machine is constructed such that it 'remembers' the longest head of S that matches with the recent processed characters of T. Whenever a mismatch occurs, the pattern S is shifted to the right (possibly more than one position) such that the next character of T will be compared with the appropriate character of S. However, it has to inspect all the characters of T until a match is found or the end of T is reached. So, effectively, the KMP algorithm skips to the next character of T each time whether a match or mismatch occurs and does not reprocess any already processed characters. The construction of the deterministic finite state automaton is done in a preprocessing phase and needs O(m) operations.

The BM algorithm [6] has basically the same idea as the KMP algorithm but allows skipping by more than one character when a mismatch occurs. This is done by comparing the characters of S against the corresponding characters of T in right-to-left order while the pattern S is shifted to the right. The comparison starts from the last character s_m, then s_{m-1}, ... and so on. When a mismatch occurs, the pattern S is shifted to the right by an appropriate amount (up to m) and the comparison starts again from s_m. The choice to start comparison from right-to-left, in contrast with the KMP method, gives the major gain of performance of BM algorithm over the KMP algorithm. For example, when a mismatching occurs and it is found that the character just read from T does not belong to S at all, the pattern S can be shifted to the right m positions. So, on the average, the BM algorithm does not have to inspect all the characters in T. Again, a preprocessing operation of O(m) is required in order to calculate the appropriate skip distances.

3. STATISTICAL ALGORITHM

In this algorithm we utilize the knowledge of probability distribution of the characters within the T to minimize the expected number of comparisons. discussed in the background section, the K algorithm always skips to the next character of whether a match or a mismatch occurs. On the o hand, the BM algorithm may skip greater distance to m) if a mismatch occurs. So the idea is to utilize knowledge of some statistical properties of the alpha characters in T and S to maximize the two follow factors:

1- The expected number of mismatching that can oc during the search process.
2- The expected skip-distance value taken whe mismatch occurs.

The first factor can be maximized by consider characters in S with the smallest probability occurrence in T. We start the matching operation w a character in S having the lowest probability and matching occurs, then we continue with the charac having the next larger probability and so on. T second factor can be maximized by choosing sequence of characters in the pattern S in the matchi operation to be near to the right end of S and included as a substring in the rest of S. These t requirements may be conflicting in general, si characters with small probability need not be near t right end of S.

Now, suppose that the probability distributi function p(.) of the alphabet characters is known. T general idea is to construct a cost function, using t pattern string S, that represents the expecte skip-distance for each character position in S and ta into account the effect of both factors above. B optimizing this function with respect to the characte positions, the position which makes the functio maximum will produce the minimum expected numbe of character comparisons.

Assume that there is a one to one mapping fro alphabet character set A onto the set of intege $I=[1,|A|]$, and let τ be a random variable whos values are drawn from the set I. For convenience, w will say the value of t is the character corresponding t the integer value in I, since these values are in one t one correspondence. The probability distributio function of τ is the same as p(.). We will consider the text T= $t_1 t_2 \ldots t_n$ as a realization of the sequence of n identical independent random variables $\tau_1\ \tau_2 \ldots \tau_n$. In reality, these random variables are not totally independent. For example, in the English language, the character 'q' will most probably be followed by the character 'u'. However, the assumption of independence will keep the computations reasonably simple. Otherwise, we will have to compute the joint distributions of every different sequence of characters of length less than m.

Let $d_i(\tau)$ be a function of the random variable τ that gives the skip-distance (number of positions the pattern string S is shifted to the right) when comparing τ with the character s_i in S. $d_i(\tau)$ is also a random variable with integer values in the range [0, m]. So the expected value of $d_i(\tau)$ is given by summing over all the characters t in the alphabet. i.e.

$$E(d_i(t)) = S_{t\epsilon A}\ d_i(\tau{=}t)p(t)$$

So during the search, at each step we could seek the position s_i (from the remaining characters of S) which make the above expected skip-distance maximum. In other words, the objective cost function will be :

$$\delta = \mathrm{Max}_i\ (E(d_i(\tau)))$$

$$= \mathrm{Max}_i(\sum_{t \epsilon A} d_i(\tau = t)p(t))$$

In order to compute the function δ we have to calculate $d_i(\tau{=}t)$ for each i and for each character in the alphabet. One approach, suggest using a technique similar to that one used in the pre-processing phase of KMP algorithm or BM algorithm. For example, the BM algorithm provides two heuristic functions called "s-pointer incrementing functions" to calculate the maximum possible increment. These two functions are defined using a similar finite state automaton (as in KMP method) that recognize the suffix substrings of S. However, this approach assumes that the searching process proceeds sequentially starting from the chosen character until a mismatch occur or the whole S string is scanned successfully. Apparently this approach cannot be applied directly to our method since we may start from any position within the string S and it is not necessary that the characters in S are ordered in the way that makes δ optimum.

Conceptually, there is no difference at all between comparing two strings (character-wise) in the order of their characters and comparing them in any other order as long as we preserve the correspondence between the characters. With this in mind, we can map the original pattern string S into a new one S' so that the order of characters in S' will represent the expected skip-distances in increasing order. i.e., the first character (from left) in S' will correspond to the character in the position i of S which yields the minimum expected skip-distance and the last character in S' will correspond to the character position in S which yields the maximum skip-distance. Having done this, we can apply a similar procedure as the one in BM to compute the skip-distances. Of course, searching the text string T will be modified to consider the new relative positions of the corresponding characters.

Although conceptually correct, the above mapping will make the computation of $d_i(\tau{=}t)$ practically unmanageable with combinatorial explosion. The value of $d_i(\tau{=}t)$ depends on the relative position with other characters in S. Since we do not know yet the mapping from S to S', we will not be able to calculate the skip-distance. In other words, in order to construct the finite state automata to determine the skip-distances, we have to have a fixed pre-known character order. This means that we have to try every possible sequence of characters of S (all mapping functions from S onto S') which is m! in order to determine the optimal arrangement.

To get around this problem, a sub optimal skip-distance will be searched for. Instead of computing $d_i(\tau{=}t)$ with a finite state automata that 'remembers' all the current matched characters so far, we will decide how much to shift the string S based only on the current position and the character just read from the text T (the realization of τ). This way, we can use the original pattern string S to compute the skip distances efficiently and then construct the mapping function to create S'. The computation of $d_i(\tau{=}t)$ based on the current position is relatively easy and can be carried out in O(m) operations as shown next.

In the remainder of this section, we formally present the above ideas and the proposed algorithm. The algorithm consists of two phases: The preprocessing *phase* and the searching phase. In the pre-processing phase, the skip-distance and the δ function are computed and the necessary data structure is created. The input to this phase is just the pattern string S. The searching phase uses the resulting data structure from pre-processing and the text string T to locate the pattern inside T.

pre-processing

The pre-processing phase is responsible for creating the correct data structure that will be used in the matching process. The main task is to calculate the skip distances $d_i(\tau=t)$ for all values of i and t. Suppose that when comparing t with s_i a mismatch occurred. Then the pattern S can be shifted to the right so that to align the character t with the first occurrence $s_j = t$ to the left of s_i. i.e. j is the greatest non-negative integer less than i such that $s_j = t$. So the skip distance in this case is (i-j).

Let A be the alphabet character set. Given the pattern string $S = s_1 s_2 \ldots s_m$, the following procedure -written in a Pascal-like pseudo code- will construct the matrix D[i, t] whose element $d_{i,t}$ represents the skip-distance that must be taken when comparing the character at the i^{th} position (s_i) with a character t from T. The dimension of the matrix D is mx|A|.

```
procedure calculate-skip-distance-matrix
begin {Initialization:}
  D[i, t] := 0  for all i and t εA
  i := 1;
  while i ≤ m  do {Mark the positions:}
  begin  D[i, si] := i;
        i := i + 1
  end;
  for all tε A do {Calculate the skip distances:}
  begin value := 0; j := 1;
    while j ≤ m do
      begin  if D[j,t]≠0 then D[j,t]:= 0, value := j
             else if value ≠ 0 then  D[j,t]:= j-value
                              else  D[j, t] := j;
        j := j+1
        end
    end
end
```

Example 1:

Suppose that A = {a,b,c,d} and S = " bccabc" t the output matrix of the procedure 'calculate-s distance-matrix' will be as follows. Notice t columns are corresponding to the characters in A the rows are corresponding to the characters in S we use the characters in S themselves instead of t indices for convenience.

	a	b	c	d
b	1	0	1	1
c	2	1	0	2
c	3	2	0	3
a	0	3	1	4
b	1	0	2	5
c	2	1	0	6

Example 2:

Suppose that A = {a,b,c,d} and S = " bcabdc" t the output matrix of the proced 'calculate-skip-distance-matrix' will be as follows:

	a	b	c	d
b	1	0	1	1
c	2	1	0	2
a	0	2	1	3
b	1	0	2	4
d	2	1	3	0
c	3	2	0	1

Now we have to compute the expected skip-distan to consider the probability distribution of the charac in A. The following procedure is a di implementation to do this. The input is an array (holds the probability of each character in A) and previously computed skip-distance matrix D[i, t]. T output of the procedure is the array v[] which holds expected skip-distances corresponding to each charac in S.

```
procedure compute-expected-skip-distance-vector
begin
  for i = 1 to m do
    begin  v[i] := 0;
      for all tεA do v[i]:= v[i] + p[t]*D[i,t]
    end
end
```

Example 3:

Consider the skip-distance matrices calculated in examples 1 and 2 above and suppose that the probability distribution of {a,b,c,d} is {0.1, 0.2, 0.4, 0.3} respectively. Then the expected skip-distance vector of S = "bccabc" is [0.8, 1.0,1.6, 2.1, <u>2.4</u>, 2.2] and the expected skip-distance vector of S = " bcabdc" is [0.8, 1.0, 1.7, <u>2.0</u>, 1.6, 1.0]. The maximum expected distances are underlined and occurred at positions 5 for the string S= "bccabc" and at position 4 for the string S = " bcabdc".

Having computed the expected skip-distances, we can decide about the sequence of which the characters in S will be matched against the text T. To fully utilize our knowledge about the skip distances we can do the matching sequence in the order of decreasing values of the expected skip-distances. This will require sorting the expected skip-distance vector v[]. Let us define the *matching sequence* (or *M-sequence)* to denote the sequence of indices of the characters in S in the same order as they will be matched against the text T.

Example 4:

Consider the calculated expected distances in Example 3 for S = "bccabc" which are[0.8,1.0,1.6, 2.1, <u>2.4</u>, 2.2]. After sorting, the new vector is [2.4, 2.2, 2.1, 1.6, 1.0, 0.8] yielding the M-sequence [5, 6, 4, 3, 2, 1]. i.e. the matching operation will start with the 5^{th} character of S "b" then the 6^{th} character "c" and so on.

Suppose that this sorting operation is done and we get the M-sequence. We can modify the skip matrix D[i,t] to further increase the skip distances by taking into consideration the knowledge of previously matched characters. For example, consider Example 4 above and suppose that during the comparison, the character "b" in the 5^{th} position is already matched but the next position (character "c" in 6^{th} position) is mismatched with an "a" character. Now according to the element $d_{6,a}$ in the corresponding matrix D (Example 1), we should shift the string S by 2 positions, but we can actually improve this distance if we recognize that the previously matched "b" (at the 5^{th} position) should move 4 positions in order to align with the first "b" in S. So the final step in the pre-processing is to improve the matrix D.

The following procedure modifies the matrix D to improve the skip-distances. The inputs to the procedure are the original D matrix and the M-sequence. The output is a modified matrix D' such that $d'_{i,t} \geq d_{i,t}$ for all i and t. The improvement is done in two steps:

- The forward improvement in which we keep track of the largest skip distance of all the characters already matched and
- The backward improvement in which the right-most occurrences of the different characters in S are checked. If all characters to the right of this occurrence are matched and the distance to the right end is larger than the recorded distance then the larger is recorded instead.

In the following procedure, a temporary Boolean array "R[0..m]" is used to mark the right-most occurrences of the different characters in S.

```
procedure improve-skip-distance-matrix
begin
{Initialize the temporary array and mark the next same
char occurrence to the left}
  for i := 0 to m do R[i] := false;
  for i := m down to 2 do R[m-D[m,s_j]] := true;
 { Start backward improvement }
  k := M-sequence[m];
  for i := m down to 2 do
  begin
   j:=M-sequence[i];
   if j ≥ k and R[j] then
     begin  for all (t ∈A) ≠ s_j do if (m-j) ≥ D[j,t]
                                  then D[j, t] := (m-j+1);
          k := j
          end
     end;
 { Start forward improvement }

 {Initialize skip by the distance between the 1st char in
      M-sequence and its left similar character}
 j :=M-sequence[1];
 if j >1 then skip := D[j-1, s_j]+1 else skip := 1;
 for i=2 to m do
 begin
   j:=M-sequence[i]
   for all (t ∈A) ≠ s_j do if skip > D[j, t]
                             then D[j,t] := skip;
 if j > 1 and skip ≤ D[j-1,s_j] then skip:=
          D[j-1,s_j]+1;
          {Update skip for next iteration}
  end
end;
```

Example 5:

Applying the above procedure to the matrix D in Example 1 and with the resultant M-sequence [5, 6, 4, 3, 2, 1] from Example 4, we get the following new matrix D6:

	a	b	c	d
b	4	0	4	6
c	4	4	0	6
c	4	4	0	6
a	0	4	4	6
b	1	0	2	6
c	4	4	0	6

Notice that this matrix dictates that after successfully matching the first character in the M-sequence, which is the "b" in the 5th position of S, any mismatching will cause skipping by at least 4 positions. In fact this is the optimal skip-distance since the substring "bc" is both in the head and the tail of the pattern S = "bccabc".

Searching phase:

Now we are ready for the searching phase. The input to this phase is the M-sequence, the improved skip-distance matrix D, the target text string T and the pattern string S. The output will be a pointer "found" indicating whether the pattern S is a substring of T or not. If the pattern is not in T, the value of "found" will be zero, otherwise, its value will be the position of the first occurrence in T. The following procedure is an implementation of that searching phase:

```
procedure search-target-text
begin
r := 0;                  { Pointer to target text T}
i := 1;                  { Pointer to M-sequence elements}
found := 0;
while ( r ≤ n-m and found = 0 ) do
   begin
  k := M-sequence[i];
  if t_{r+k} = s_k then i:= i+1 {Matching: increment i}
  else begin  r := r + D[k, t_{r+k}];
      i:= 1 {Mismatching: skip and reset i}
    end
  if i > m then found := r +1
    end
end;
```

4. COMPLEXITY AND PERFORMANC ANALYSIS

The time complexity of the pre-processing phase i O(m log m) in the worst case since the complexity each component is as follows:

1- procedure calculate-skip-distance:
The initialization takes |A|.m constant steps, th marking stage takes m constant steps, and the la stage of calculation takes |A|.m constant steps Since |A| is constant, the total complexity of th procedure is O(m).

2- procedure compute-expected-skip-distance-vector: The time complexity of this procedure is |A|.m i.e. O(m).

3- Sorting the elements in the expected skip-distanc vector is of O(m log m).

4- procedure improve-skip-distance-matrix:

- The initialization takes 2m constant steps,
- The backward improvement takes |A|.m consta steps in the worst case,
- The forward improvement takes |A|.m consta steps.

So, the total time complexity of the procedure i O(m) in the worst case.

The calculation of the time complexity of th searching procedure is a little bit more complicat specially in the worst case. The while loop, controlle by the variable r, has an inner loop which i controlled by the variable i. r is incremented when mismatch occur and at the same time, i is reset to 1 The exit from the loop is either when i = m or whe r > n-m. The worst case occurs when r is alway incremented by a small value (e.g. 1) relative to m while i is always approaching m before resetting. B the construction of the D[] matrix usually maximiz both the probability of mismatching (and hence, res i) and the skip distances of the odd characters that ma be near the left end of S through the backwa improvement (and hence, r is incremented by th maximum allowable distance). So, both two conditio for worst case can not exist together.

However the average case is relatively easy t calculate. Let ξ_i be the average skip distance whe the character s_i mismatch with a character from T, an let $p_i = p(s_i)$, the probability of the character s_i. W will calculate the average number of iterations insid the while loop by multiplying the average number o times the variable i is incremented and the averag number of times the variable r is incremented as follows:

Ave. # of iterations = Ave. # of increments of i x Ave. # of increments of r

Ave. # of increments of r = (n-m)/ Ave. skip distance . (1)

$$\text{Ave. skip distance} = (1-p_1)\ \xi_1 + p_1(1-p_2)\ \xi_2 + p_1 p_2(1-p_3)\ \xi_3 + \ldots + p_1 p_2 \cdots p_{m-1}(1-p_m)\ \xi_m \geq (1-p_1)\ \xi_1$$

then substituting in (1) we get

$$\text{Ave. \# of increments of r} \leq (n-m)/\ (1-p_1)\ \xi_1 \qquad (2)$$

Ave. # of increments of i

$$\begin{aligned}
&= p_1(1-p_2) + 2\,p_1p_2(1-p_3) + \ldots + (m-1)p_1p_2\ldots(1-p_m) + mp_1p_2\cdots p_m \\
&= p_1 - p_1p_2 + 2p_1p_2 - 2p_1p_2p_3 + \ldots + mp_1p_2\cdots p_m \\
&= p_1 + p_1p_2 + p_1p_2p_3 + \ldots + p_1p_2\cdots p_m \\
&= p_1 + p_1(p_2 + p_2p_3 + \ldots + p_2p_3\cdots p_m) \\
&\leq p_1 + p_1(p_2 + p_3 + \ldots + p_m) \\
&\leq p_1 + p_1(1-p_1) \\
&\leq 2p_1 \qquad (3)
\end{aligned}$$

From (2) and (3) above,
Average # of iterations inside the while loop

$$\leq 2p_1\ (n-m)/\ (1-p_1)\ \xi_1 \qquad (4)$$

For small values of p_1 the # of iterations inside the while loop will be small, and approaches the best case which is equal to m. For larger values of p_1, the # of iterations approach the worst case which is equal to mn.

Comparisons

The pre-processing phase of this algorithm is of O(m log m) in contrast to the pre-processing phases of both KMP and BM algorithms which are of O(m). However there is a relatively large constant - in all algorithms - which is |A|. So, for practical values of m less than $2^{|A|}$, the statistical algorithm's pre-processing is of the same complexity as the other two algorithms.

The average number of character comparisons during the searching phase in the statistical algorithm is less than or equal to that of the BM algorithm. This can be seen from the inequality (4), since the statistical algorithm maximizes ξ_1 and minimizes p_1. On the other hand, the BM algorithm is chancy in the sense that it may perform good if the last character in S (s_m) happens to have a low probability and good skip-distance. In general, the statistical algorithm will outperform, on the average, both the BM and KMP algorithms when there are some characters with low probability (such as 'q' in the English text).

5. CONCLUSION AND FUTURE RESEARCH

The availability of large amounts of text material in modern information systems and the still unsatisfactory results of query methods that depend on index terms and other text processing applications lead to a demand for a practical and efficient algorithms for pattern matching. We present a statistical pattern matching algorithm for text that can utilize the knowledge of the probability distribution of the alphabetic characters in the text under consideration. The algorithm is essentially a modification of the KMP algorithm and the BM algorithm. The algorithm has a pre-processing phase of time complexity in the worst case of order m log m - where m is the length of the pattern - and has a searching phase that on the average does not examine every character in the target text (sublinear). We show that the number of comparisons is in general less than or equal to the number of comparisons in the BM algorithm and the performance is boosted when the pattern contains characters with low probability of occurrence.

Experimental studies are needed to assert the given result and to show the actual gain over other methods in different language environment such as English and Arabic. Also, the possibility of implementing the algorithm as special hardware element is now considered.

6. REFERENCES

[1] Su, Louise T. "An Investigation to Find appropriate measures for evaluating interactive information retrieval", *Ph.D. Rutgers State University*, 1991.

[2] Ekmekcioglu, F.C., Robertson, A.M. & Willett P. "Effectiveness of query expansion in ranked-output document retrieval systems". J. *Information Science*, No. 18, 1992.

[3] Standish, T.A. *Data Structure Techniques*. Addison-Wesley, 1980.

[4] Faloutsos, Christos "Access Methods for Text", *Computing Surveys*, Vol.17, No. 1 March 1985.

[5] Knuth, D.E., Morris,J.H.&Pratt,V.R. "Fast Pattern Matching in Strings". *SIAM J. Computer* Vol.6, No.2, June 1977.

[6] Boyer, R.S. & Moore, J.S. "A Fast String Searching Algorithm", *CACM*, Vol.20, No. 10, Oct. 1977.

[7] Yannakoudakis, E.J. & Angelidakis, G. ' insight into the entropy and redundancy of english dictionary" *IEEE Trans . on Pa Analysis and Machine Intelligence*, Vol.10, N p:960-970, Nov 1988.u

[8] Johansson, S. & Hofland, K. *Frequency Anal of English vocabulary and grammer*, Vol. Clarendon Press - Oxfored, 1989 (reprint 199

[9] M. Mrayati "Statistical Studies in Ar linguistic". *In Computers and the Ar Language*; P. Mackey (ed) Hemisphere Publish Corporation, 1990.