A Bit-Parallel Filtering Algorithm to Solve the Approximate String Matching Problem

Chia Shin Ou and R. C. T. Lee
Department of Computer Science
National Tsing Hua University
d9662836@oz.nthu.edu.tw and rctlee@ncnu.edu.tw

Abstract

There are many algorithms to solve the exact string matching problem and the approximate string matching problem. One kind approach of them is so-called filtering approach. This kind approach is used to decrease the searching time in scanning by obtaining the preprocessing. One of the filtering is called Approximate Boyer-Moore Algorithm which is proposed by Tarhio and Ukkonen. The Approximate Boyer-Moore Algorithm spent O((k+c)m) time in preprocessing where c is the size of the alphabet. In this paper, we present a bit-parallel filtering technique which is based on the Approximate Boyer-Moore Algorithm and spents O(cm) time. The result will be useful when we solve the text with multiple patterns or with long pattern.

1 Introduction

Consider the approximate string matching (ASM) problem. The edit distance [12, 14] between two strings A and B, denoted as $ed(A, B)$, is defined as the minimum number of insertions, deletions and substitutions needed to transform string A to string B. The ASM Problem is , given a text $T = t_1t_2\ldots t_n$, $P = p_1p_2\ldots p_m$ and an error bound k, to find all such j that the edit distance between some substring of T ending at position i and P is at most k. The classic algorithm to solve the ASM problem is according to the dynamic programming approach [8] in $\underline{O}(mn)$. Let C be a $m + 1$ by $n + 1$ matrix such that $C(i, j)$ is the minimum edit distance between $p_1p_2\ldots p_j$ and any substring of T ending at position i. The dynamic programming table C can be obtained by the following recursive formula:

$$C(i, j) = \min\begin{cases} C(i-1, j)+1 \\ C(i, j-1)+1 \\ C(i-1, j-1)+\text{ if } t_i = p_j \text{ then } 0 \text{ else } 1 \end{cases},$$

where $0 \le i \le n$ and $1 \le j \le m$, and $C(i, 0) = 0$ for $0 \le i \le n$.

Landau and Vishkin [4], Galil and Park [2], and Ukkonen and Wood [14] also proposed different improved algorithms for [8] in $O(kn)$ time. Myers [6] gave a new algorithm by using the bit-parallel approach to obtain the dynamic programming table which has good performance in practical and can be done in $O(mn/w)$ time.

Filtering algorithm [7,9,10,11] is a good issue to decreasing the processing time in scanning the text. One of the filtering algorithm [11] is Approximate Boyer-Moore (ABM) Algorithm which is used to solve the ASM problem. The ABM Algorithm has good performance in moderate patterns with small error bound k and large alphabet size. In our paper, we proposed a bit-parallel filtering algorithm to improve the ABM algorithm.

2 Approximate Boyer-Moore Algorithm

The Approximate Boyer-Moore (ABM) Algorithm contained two phases: *filtering* and *checking*. The filtering phase is to evaluate every diagonal h of the dynamic programming table C whether entries in the diagonal h need to be computed. After the filtering phase, the value of needless entries of C will be set by ∞. Then, the checking phase is to compute the remained entries of C. The checking phase can be done by using the dynamic programming approach [8]. Hence, in our paper, we just focus the phase in filtering.

The ABM Algorithm, although an approximate string matching algorithm, is related to the exact string matching algorithms, namely Boyer and Moore Algorithm [1] and Horspool Algorithm [3]. In fact, if we consider that the error bound k is equal to 0, the ABM Algorithm reduces to the Horspool Algorithm. Most of them start with some kind of pairwise character comparison. Consider a window $W = w_1w_2\ldots w_m$ of a text string T and a pattern $P = p_1p_2\ldots p_m$. Suppose $w_m \neq p_m$, we know there is no exact matching between P and W and a shifting is now needed. In approximate string matching, we cannot make such a conclusion. Let us consider the case such that T = *acttgta* and P = *acttgt*. If we compare t_7 and p_6, we will find out that $t_7 \neq p_6$. We may correctly conclude that we need at least 1 operation to make $t_7 \neq p_6$. We then compare t_6 with p_5 and we will find out that $t_6 \neq p_5$. However, we cannot claim that $ed(T, P) \geq 2$, because by inserting an a after p_6, we will make P identical to T. In fact, in this case, $ed(T, P) = 1$.

To facilitate our discussion, denote $S(i, j)$ to be the substring $s_i s_{i+1} \ldots s_j$ of S and let us define the k-environment of p_j as given the pattern $P= p_1p_2\ldots p_m$, the k-environment of p_j, denoted as $E_k(j)$, is $P(j–k, j+k)$. Note that each t_i is aligned with a p_j. For instance, t_2 is aligned with p_3. In this case, every t_i is in its corresponding $E_1(j)$ under the condition that $k = 1$. For instance, $t_6 = t$ is in $E_1(7) = tc$ and $t_2 = c$ is in $E_1(3) = ctt$. We can also prove that $ed(T, P) = 1 = k$. In this case, we can see that $t_2 = c$ is not in its corresponding $C(1, 3) = atg$ and $t_4 = t$ is not in $C(1, 5) = gcg$. We can also prove that $ed(T, P) > k = 1$.

Let us assume that we have two strings with the same length and the error bound k is already specified. Then we examine each t_i to see whether t_i is in $C(k, i)$. If one t_i is not in $C(k, i)$, we cannot perform any insertion or deletion to change p_i to t_i. But, we can perform a substitution. If $(k + 1)$ t_i's are not in $C(k, i)$'s, we have to perform $(k + 1)$ substitutions. Thus we have $ed(T, P) > k$. From the above discussion, we can have the following Lemma:

Lemma 1. [] Given an error bound k and two strings $T = t_1t_2\ldots t_m$ and $P = p_1p_2\ldots p_m$, if there are more than k t_i's not in $E(k, i)$'s, $ed(T, P) > k$.

For finding t_i fast, the filtering phase need to precompute table $Bad(j, x)$, $1\leq j \leq m$, $x \in \Sigma$, such that $Bad(j , x) = true$ if and only if x does not appear in k-environment E_j.

Lemma 1 can obviously be used as a filtering scheme. In the following, we shall show a very interesting rule of the ABM Algorithm: the shifting rule of the filtering phase. The shifting is based upon the following rule.

In the ABM Algorithm, the shifting is based upon Horspool algorithm. Consider the suffix $W(m–k, m)$ of the window W, if a shifting is needed, this substring of the new window will be aligned with a corresponding substring $P(i, k)$ in P. According to Boyer and Moore algorithm, there must exist at least one pair of characters in $W(m–k, m)$ and $P(i, k)$ which exactly match with each other.

Consider the case, $W = agtcccta$ and $P = agtcgcta$. Suppose $k = 1$ and we have to shift. After shifting, $W(m – k, m) = W(7, 8) = ta$ will be aligned with $P(i, i+k) = P(i, i+1)$ for some i. Thus we have to make sure that after the shift, among the two characters of $W(7, 8)$, namely t and a, at least one character of them will be exactly matched with its corresponding character in $P(i, i+1)$. If we want to match t, 4 steps are needed. If we want to match a, 7 steps are needed. We choose the minimum of them, which is 4.

To facilitate our shifting mechanism, in the following, we shall give the D_j table of the ABM Algorithm defined as follows: Given an alphabet set $\Sigma = \{x_1, x_2, \ldots , x_\sigma\}$ and pattern P with length m, we create a table, denoted as D_j table of P, containing σ entries where σ is the alphabet size. Each entry stores the location of the rightmost x_i, $1 \leq i \leq \sigma$, in $P(1, m – j)$ counted from location $m – j$, if it exists. If x_i does not exist in $P(1, m – j)$, store m in the entry.

For instance, let $P = gcagagag$ and $\Sigma =\{a, c, g, t\}$. Then the D_j tables, for $j = 1$ and $j = 2$, are as follows:

Table 1. D_1 and D_2 tables for $P = gcagagag$

The D_1 Table

a	c	g	t
1	6	2	8

The D_2 Table

a	c	g	t
2	5	1	8

Consider window $W(1, m)$, $P(1, m)$ and an error bound k. Let $d_j = D_j(t_{m–j+1})$, for $j = 1$ to $j = k + 1$. Then the number of steps needed to shift is $d = min\{d_1, d_2, \ldots , d_{k+1}\}$.

Consider the case, $T = cctcgcaagagc$, $P = gcagagag$ and an error bound $k = 1$. In this case, $W = cctcgcaa$, $d_1 = D_1(w_m) = D_1(\text{a}) = 1$ and $d_2 = D_2(w_{m-1}) = D_2(a) = 2$. Thus the number of shifts is $d = min\{d_1, d_2\} = min\{1, 2\} = 1$. We would shift one step.

It can be easily seen that if we shift less than d steps where d is defined by the shifting rule.

The filtering algorithm is shown in the following:

Algorithm 2 Filtering Phase of Approximate Boyer-Moore Algorithm

Construct the D_j table and the $Bad(j, x)$ for $1 \leq j \leq m$ and $x \in \Sigma$;
$i := m$;
while $i \leq n$ **do begin**
  $r := i$; $j := m$;
  $bad := 0$;
  $d := m$;
  while $j > k$ and $bad \leq k$ **do begin**
    if $j \geq m – k$ **then** $d := min(d, d_k[i, t_r])$;
    if $Bad(j, t_r)$ then $bad := bad + 1$;
    $j := j – 1$; $r := r – 1$;
  end;

if $bad \leq k$ **then**
mark entries $C(0, i - m - k), \ldots, C(0, i - m + k)$;
$i := i + max(k + 1, d)$;**end**

3 Our Bit-Parallel Filtering Algorithm

Our bit-parallel filtering algorithm is based on ABM algorithm. The main idea of filtering phase of ABM algorithm is to determine whether t_i exists in k-environment of p_j or not. To achieve this idea in bit-parallel, we obtained the incident vector IV defined as follows: Given a string $S = s_1s_2\ldots s_n$ and a character x, $IV_S[x] = (IV_S[t_i](1), IV_S[t_i](2), \ldots, IV_S[t_i](n))$ where $IV_S[t_i](j) = 1$ if $s_j = x$ and $IV_S[t_i](j) = 0$ if otherwise. For instance, let $P = aacag$. Then $IV_P[a] = (1,1,0,1,0)$, $IV_P[c] = (0,0,1,0,0)$ and $IV_P[g] = (0,0,0,0,1)$. According to Lemma 1, we redefined our filtering scheme as follows:

Lemma 4: Given an error bound k and two strings $T = t_1t_2\ldots t_m$ and $P = p_1p_2\ldots p_m$, if there are more than k t_i's such that for all $i - k \leq j \leq i + k$, $IV_P[t_i](j) = 0$, $ed(T, P) > k$.

We need bit-parallel operation to check, in vector $IV_P[t_i]$, whether there exists a 1 between locations $i - k$ and $i + k$ in $IV_P[t_i]$. To achieve this, we first define an auxiliary vector $B = (B(1), B(2), \ldots, B(n))$ in which all bits are 0's, except that $B(j) = 1$ for all $i - k \leq j \leq i + k$. Consider the case where $i = 3$, $k = 2$ and $n = 10$. Vector B will be (0, 1, 1, 1, 1, 1, 0, 0, 0, 0).

To detect the existence of 1's between locations $i - k$ and $i + k$ in $IV_P[t_i]$, we can use the logical & operation on vectors $IV_P[t_i]$ and B. If such 1's exists in $IV_P[t_i]$, this operation keeps the bit within locations $i - k$ and $i + k$ in $IV_P[t_i]$. For instance, $IV_P[t_i]$ = (1, 0, 0, 1, 0, 0, 0, 1, 1, 0) and B = (0, 1, 1, 1, 1, 1, 0, 0, 0, 0), ($IV_P[t_i]$ & B) = (0, 0, 0, 1, 0, 0, 0, 0, 0, 0). It shows that $t_i = p_4$. If $IV_P[t_i]$ = (0, 0, 0, 0, 0, 1, 0, 1, 1, 0) and B = (0, 1, 1, 1, 1, 1, 0, 0, 0, 0), ($IV_P[t_i]$ & B) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0). It shows that there is no t_i which exists between locations $i - k$ and $i + k$ in $IV_P[t_i]$. Hence, we can use the bit-parallel operation ($IV_P[t_i]$ & B) to confirm whether t_i exists between locations $i - k$ and $i + k$ in $IV_P[t_i]$.

The bit-parallel filtering algorithm is shown in the following:

Algorithm 2: Bit-parallel Filtering Approach based on Approximate Boyer-Moore Algorithm

Construct the D_j table;
for all $x \in \Sigma$ **do begin**
$B := 1^{k+1}0^{m-k+1}$;
for j = 1 to m **do begin**
if ($IV_P[x]$ & B) = 0^m **then** $Bad(j, x)$ = **true**;
else $Bad(i, x)$ = **false**;
B >> 1;
if $j \leq k$ **then** $B := B$ | 1;
end for
end for
$i := m$;
while $i \leq n$ **do begin**
$r := i$; $j := m$;
bad :=0;
$d := m$;
while $j > k$ and $bad \leq k$ **do begin**
if $j \geq m - k$ **then** $d := min(d, d_k[i, t_r])$;
if $Bad(j, t_r)$ then $bad := bad + 1$;
$j := j - 1$; $r := r - 1$;
end;
if $bad \leq k$ **then**
mark entries $C(0, i - m - k), \ldots, C(0, i - m + k)$;
$i := i + max(k + 1, d)$;
end

5 Conclusion

In this paper, we proposed a bit-parallel filtering algorithm which is based on the Approimate Boyer-Moore Algorithm. Our filtering algorithm can decrease the preprocessing time to $O(cm)$. The result will be useful when we solve the text with multiple patterns or with long pattern and also can be used to determine the article plagiarism.

References

[1] R. Boyer and S. Moore. A fast string searching algorithm. *Communications of the ACM*, **20**, 1977, pp.75-96.

[2] Z. Galil and Z. Park. An improved algorithm for approximate string matching. *SIAM journal on Computing*, **19**, 1990, pp.989-999.

[3] N. Horspool. Practical fast searching in strings. *Software Practice & Experience*, 1980, pp.501-506.

[4] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, **10**, 1989, pp.157-169.

[5] R. C. T. Lee, S. S. Tseng, R. C. Chang, and Y. T. Tsai. Introduction to the Design and Analysis of Algorithms. *McGraw-Hill Education*, 2005.

[6] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, **46**, 1999, pp.395-415.

[7] G. Navarro and R. Baeza-Yates. Very fast and Simple approximate string matching. *Information Processing Letters*, **72**, 1999, pp.65-70.

[8] P. H. Sellers. String Matching with Errors. *Journal of Algorithms*, **20**, 1980, pp.443-453.

[9] E. Sutinen and J. Tarhio. On using q-gram

locations in approximate string matching. *In Proceeding of the 7th Annual Symposium on Combinatorial Pattern Matching*, Springer-Verlag, Berlin, 1995, pp.50-61.

[10] J. Takaoka. Approximate pattern matching with samples. *In Proceedings of ISAAC*. Springer-Verlag, Berlin, 1994, pp.234-242.

[11] J. Tarhio and E. Ukkonen. Approximate Boyer-Moore String Matching, *SIAM Journal on Computing*, **22**, 1993, pp.243-260.

[12] E. Ukkonen. Finding Approximate patterns in strings. *Journal of Algorithms*, **6**, 1985, pp.100-118.

[13] R. Wagner and M. Fischer. The string-tostring correction problem. *Journal of the ACM*, **21**, 1975, pp. 168-173.

[14] S. Wu and U. Manber, Fast Text Searching: Allowing Errors, *Communications of the ACM*, **35**, 1992, pp.83-91.