Improved Exact String Matching Algorithms Based upon Selective Matching Order and Branch and Bound Approach

Chia Wei Lu[1], Chin Lung Lu[1] and R. C. T. Lee[1, *]
[1]Department of Computer Science
National Tsing Hua University, Hsinchu City, Taiwan, ROC
d9762807@oz.nthu.edu.tw, cllu@cs.nthu.edu.tw, rctlee@rctlee.cyberhood.net.tw

Abstract

In this paper, we propose two improved algorithms for exact string matching problem, which aims to find all the positions i's in a given text where a given pattern occurs. Our algorithms find the optimal selective comparing order of the pattern so that we could have a better performance in the searching phase. To find the optimal comparing order, we adopt the branch and bound approach. Moreover, our proposed algorithm can be combined with other existing exact string matching algorithms to improve the searching efficiency. The experimental results show that our algorithms indeed have the smallest number of character comparisons when comparing with the other algorithms using different comparing order. Besides, our algorithms are also efficient in the running time as compared with other existing exact string matching algorithms.

1 Introduction

In this paper, we are concerned with the exact string matching problem in which given a pattern $P = p_1 p_2 ... p_m$ and a text $T = t_1 t_2 .. t_n$, $n \geq m$, we are asked to find all occurrences of P in T. Much research has been done for this problem [1-27]. The most remarkable algorithms, KMP [17] and BM [3], have linear searching time in the worst case. Many algorithms [3,11,12,16,19] perform efficiently for large alphabet size, but few algorithms [4,19] preform efficiently for small alphabet size. Since many applications, such as anti-virus and DNA searching, are of small alphabet size, it is also important to design efficient algorithms for these cases. However, it is more difficult to design an efficient algorithm for small alphabet size. Therefore, in this paper, we concentrate on the efficiency of string matching algorithms for small alphabet size. In this paper, we assume that the size of the alphabet is known when we are given P and T. For example, considering the DNA sequence analysis, the size of alphabet is 4.

Considering the brute-force algorithm, we first open the window $W = T(1, m) = t_1 t_2 .. t_m$ in T with size m and try to see whether P exactly matches with this window. It compares the window with P character by character from left to right. If a mismatch is found, slide the window one step to the right and then compare the window $W = T(2, m+1)$ with P. It repeats the processes till the right-most window of T being compared. This approach is an exhaustive search approach because every substring of length m in T is compared, and the time complexity in worst case is $O(mn)$. Nearly all exact string matching algorithms try to avoid such kind of exhaustively searching. In the following, we shall introduce the Sunday algorithm [24] which allows us to avoid an exhaustive search.

The Sunday's Algorithm

In the Sunday algorithm [24], it compares characters of the window with P by using a specified order for every different P. Given a string $P(1, m) = p_1 p_2 \ldots p_m$, $D(i)$ for $1 \leq i \leq m$, is the distance between p_i and the right most character equal to p_i to the left of location i if such a character exists in $P(1, i-1)$; otherwise, $D(i) = i$. For example, suppose that $P = actagtgctagt$. Then its $D(i)$'s are also shown in the following table.

i	1	2	3	4	5	6	7	8	9	10	11	12
P	*a*	*c*	*t*	*a*	*g*	*t*	*g*	*c*	*t*	*a*	*g*	*t*
$D(i)$	1	2	3	3	5	3	2	6	3	6	4	3

Having $D(i)$'s, we may now determine the order of character comparisons. The location with the largest $D(i)$ will be the first one and that with the second largest $D(i)$ will be the second and so on. Let I be an integer array that is a permutation of $\{1, 2, \ldots, m\}$, and $I[j]$ be the location of the *j*-th character in *P* to be compared. That is, for a window $W = w_1 w_2 \ldots w_m$, we compare $p_{I[1]}$ against $w_{I[1]}$ first, and then $p_{I[2]}$ against $w_{I[2]}$ and so on. We denote the values of $I[1] = i_1$, $I[2] = i_2$, …, $I[m] = i_m$ by $I[i_1, i_2, \ldots, i_m]$. The table below gives the order of the pattern in the above table.

j	1	2	3	4	5	6	7	8	9	10	11	12
$I[j]$	10	8	5	11	12	9	6	4	3	7	2	1

Next, we need to compute the number of steps to slide the window to the right when the first mismatch is found. Suppose that we find the first mismatch at $p_{I[j]}$. This means that $p_{I[i]}$ matches with $w_{I[i]}$ for $1 \le i \le j-1$. If we slide the window x steps to the right, then $p_{(I[i]-x)}$ will be aligned with $w_{I[i]}$ for each $1 \le i \le j-1$ if $I[i]-x > 0$. Note that if $I[i]-x \le 0$, then $w_{I[i]}$ will not be aligned with any character in P. Since $w_{I[i]} = p_{I[i]}$ for all $1 \le i \le j-1$ and $w_{I[j]} \ne p_{I[j]}$, we have $w_{I[i]} = p_{I[i]} = p_{(I[i]-x)}$ for all $1 \le i \le j-1$ if $I[i]-x > 0$ and $w_{I[j]} \ne p_{I[j]} \ne p_{(I[j]-x)}$ if $I[j]-x > 0$. Therefore, we can decide the value of x by satisfying $p_{I[i]} = p_{(I[i]-x)}$ for all $1 \le i \le j-1$ if $I[i]-x > 0$ and $p_{I[j]} \ne p_{(I[j]-x)}$ if $I[j]-x > 0$. This can be done in preprocessing. Note that we should have a minimum number of steps to slide the window to satisfy the above condition so that we will not miss any solution. The sliding distance Δ used in Sunday algorithm [24] is defined as follows. For all $1 \le j \le m$, $\Delta[j]$ is the minimum $mshift$, where $mshift$ is a positive integer, such that the following two conditions are satisfied:

Condition (1) Either $(I[i]-mshift) < 1$ or $p_{I[i]} = p_{(I[i]-mshift)}$ for all $1 \le i \le j-1$.

Condition (2) Either $(I[j]-mshift) < 1$ or $p_{I[j]} \ne p_{(I[j]-mshift)}$.

For $j = m+1$, $\Delta[j]$ is the minimum value of $mshift$ such that either $(I[i]-mshift) < 1$ or $p_{I[i]} = p_{(I[i]-mshift)}$ for all $1 \le i \le j-1$. We denote the values of $\Delta[1] = d_1$, $\Delta[2] = d_2$, …, $\Delta[m+1] = d_{m+1}$ by $\Delta[d_1, d_2, \ldots, d_{m+1}]$.

For the example of $P = agcca$ and $I[5, 3, 2, 4, 1]$, it can be verified that $\Delta[1, 4, 4, 4, 4, 4]$. Note that if the comparing order is from left to right, i.e. $I[1, 2, \ldots, m]$, Δ is equal to the sliding function used in KMP algorithm. If the comparing order is from right to left, i.e. $I[m, m-1, \ldots, 1]$, Δ is equal to one of the sliding functions used in BM algorithm. Sunday proposed two algorithms [24] which used different comparing orders, and both of them are better than KMP and BM algorithms. Note that the values of Δ can be computed in preprocessing. The algorithm for finding the sliding distance Δ will not be described in this paper. We refer the readers to [24] for the algorithm. In the following section, we present our algorithm [20] which improved the Sunday algorithm.

2 Our Improved Algorithm

Consider the example with $P = aaaaata$ and the alphabet $\Sigma = \{a, c, g, t\}$. The Sunday algorithm will first compare $p_6 = t$ with w_6 of the window. If $p_6 = w_6$, it then compares p_7 with w_7. Suppose that the first mismatch is found at the second comparison, i.e. $p_7 \ne w_7$. We then slide the window seven steps to the right. If the first mismatch occurs at the first comparison, i.e. $p_6 \ne w_6$, we can slide the window only one step to the right because $p_5 \ne p_6$. However, in this example, suppose that we compare p_5 with w_5 first and then p_6 with w_6. If $p_5 = w_5$ and $p_6 \ne w_6$, we can slide the window only one step to the right because $p_4 = p_5 = a$ and $p_5 \ne p_6$. If $p_5 \ne w_5$, we then can slide the window five steps to the right because the characters in $P(1, 5)$ are all equal to $'a'$. In this example, the alphabet size is 4. Then for two random characters x and y from the alphabet, the probability of $x = y$ is $1/4$ and that of $x \ne y$ is $3/4$. Therefore, if we compare p_6 with w_6 first and then p_7 with w_7, we have $3/4$ probability to slide the window one step to the right, and $(1/4)\cdot(3/4) = 3/16$ probability to slide the window seven steps to the right. However, if we compare p_5 with w_5 first and then p_6 with w_6, we have $3/4$ probability to slide the window with five steps, and $(1/4)\cdot(3/4) = 3/16$ probability to slide the window with one step. If we only consider the cases where the first mismatch occurs at the first and the second comparisons, the expected number of steps to slide a window for Sunday algorithm is $(3/4)\cdot 1 + (3/16)\cdot 7 = 33/16$, and that for comparing p_5 with w_5 first and then p_6 with w_6 is $(3/4)\cdot 5 + (3/16)\cdot 1 = 63/16$. It can be realized that the latter comparing order is better than the former used in Sunday algorithm.

For a comparing order I, we can compute its sliding distance Δ. Assume that the first mismatch occurs at the position $I[i]$. This means that $p_{I[j]} = w_{I[j]}$ for $1 \le j \le (i-1)$ and $p_{I[i]} \ne w_{I[i]}$. Then we can slide the window $\Delta[i]$ steps to the right. Let σ be the alphabet size. The probability of $p_{I[j]} = w_{I[j]}$ for all $1 \le j \le (i-1)$ and $p_{I[i]} \ne w_{I[i]}$ is $(1/\sigma)^{i-1}((\sigma-1)/\sigma)$. Therefore, the probability of sliding the window by $\Delta[i]$ steps is $(1/\sigma)^{i-1}((\sigma-1)/\sigma)$. To measure the goodness of a comparing order and its sliding distance Δ, we define a function $AVGS$ to compute the expected number of steps to slide a window as follows.

$$AVGS(\Delta) = \left(\sum_{i=1}^{m} (1/\sigma)^{i-1}((\sigma-1)/\sigma)\Delta[i] \right) + (1/\sigma)^m \Delta[m+1].$$

For example, let $P = aaaaata$. Consider the comparing order I_1 and its sliding distance, denoted by Δ_1, which are shown in the following table.

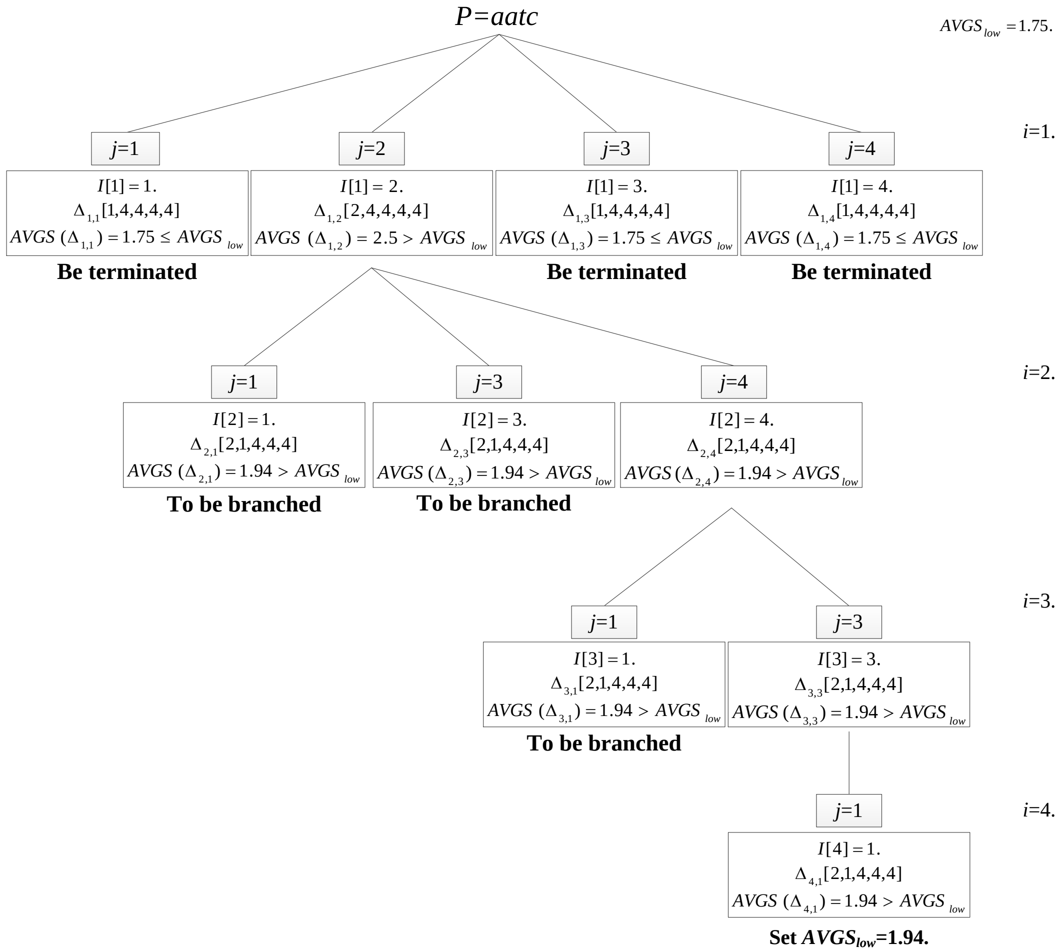

Fig. 1. An iteration of our branch and bound approach to find the optimal comparing order.

j	1	2	3	4	5	6	7	8
$I_1[j]$	5	6	7	4	3	2	1	
$\Delta_1[j]$	5	1	7	6	6	6	6	6

Then, by the definition, we have $AVGS(\Delta_1) = (3/4)\cdot 5 + (1/4)\cdot(3/4)\cdot 1 + (1/4)^2\cdot(3/4)\cdot 7 + \ldots + (1/4)^6\cdot(3/4)\cdot 6 + (1/4)^7\cdot 6 \cong 4.36$. Let us consider another comparing order I_S and its sliding distance, denoted by Δ_S, used in Sunday algorithm as shown in the following table.

j	1	2	3	4	5	6	7	8
$I_S[j]$	6	7	5	4	3	2	1	
$\Delta_S[j]$	1	7	6	6	6	6	6	6

Then we have $AVGS(\Delta_S) = (3/4)\cdot 1 + (1/4)\cdot(3/4)\cdot 7 + (1/4)^2\cdot(3/4)\cdot 6 + \ldots + (1/4)^7\cdot 6 \cong 2.44$. By comparing $AVGS(\Delta_1)$ with $AVGS(\Delta_S)$, we may conclude that the comparing order I_1 is better than I_S because $AVGS(\Delta_1) > AVGS(\Delta_S)$.

If we can find the optimal comparing order I_{OPT} such that its sliding distance, denoted by Δ_{OPT}, has the maximal value of $AVGS_{OPT}$ where $AVGS_{OPT} = AVGS(\Delta_{OPT})$, we would have the best performance in searching phase. However, the number of possible comparing orders is the factorial of m. It is not practical to perform an exhaustive search to find the optimal comparing order. In the following, we give a branch and bound algorithm to efficiently find the optimal comparing order.

Consider the example where $P = aatc$. The comparing order of Sunday algorithm is $I_S[4,3,2,1]$ and its sliding distance is $\Delta_S[1,4,4,4,4]$. If we use this comparing order, we have $AVGS(\Delta_S) = 1.75$. Then we can use it as a lower bound of $AVGS_{OPT}$, denoted by $AVGS_{low}$, that is, $AVGS_{low} = AVGS(\Delta_S) = 1.75$. Next, we use a branch and bound strategy for finding the optimal comparing order as illustrated in Fig. 1.

In our branch and bound approach, we adopt the depth first branching strategy. The nodes at level i of our branch and bound tree represent all the possible values $I[i]$. Therefore, if we reach level i, the values of $I[j]$ for all $1 \le j < i$ have been determined. Let $\Delta_{i,j}$ denote the sliding distance Δ for setting $I[i] = j$ at level i, $\Delta_{i,j}[k] = \Delta[k]$ for $1 \le k \le i$, and $\Delta_{i,j}[k] = m$ for $i < k \le m+1$. Note that for each level i of the branch and bound tree, every node is related to a possible value of $I[i]$. Consider the first level of the tree in the Fig. 1, i.e., $i=1$. Any character of P can be the first one to be compared in a window. We compute the $\Delta_{1,j}$'s for $1 \le j \le 4$. Consider the case where p_2 is the first character to be compared. If p_2 mismatches with its corresponding character in a window, we can slide the window 2 steps to the right because $p_1 = p_2 = a$. Therefore, we set $\Delta_{1,2}[1]=2$. As for the values of $\Delta_{1,2}[k]$'s for $1 < k \le 5$, we set all of them as $m=4$ so that $AVGS(\Delta_{1,2})$ is a upper bound of the $AVGS$'s in this branch. That is, if $AVGS(\Delta_{1,2})$ is smaller than or equal to $AVGS_{low}=1.75$, we then can terminate this branch. In this example, only the branch $I[1]=j=2$ will be branched.

In the second level, $i=2$, under the situation that p_2 is the first compared character, there are three possibilities to choose one of p_1, p_3, and p_4 as the next compared character. For each possibility, we compute the values of $\Delta_{2,1}$, $\Delta_{2,3}$, and $\Delta_{2,4}$ and branch the node whose $AVGS$ value is the largest. We repeat the processes and if it can be branched to the level $i=m$, we then compute the sliding distance Δ. If $AVGS(\Delta) > AVGS_{low}$, we then set $AVGS_{low} = AVGS(\Delta)$ and record this better comparing order I. In this example, we reach the level $i=m$, and the value of $AVGS(\Delta)$ is 1.94 which is larger than $AVGS_{low}=1.75$. Thus, we set $AVGS_{low}=1.94$ and record the comparing order $I[2,4,3,1]$ which is better than that of Sunday algorithm. Note that there are still three nodes to be branched. However, all of the branches will be terminated immediately because their $AVGS$ values are smaller than or equal to $AVGS_{low}=1.94$.

Note that for the very long patterns, it may still take long time to search the optimal comparing order. However, the effectiveness of the higher levels, say level $i>5$, would not be significant. By the definition of $AVGS$, the value do not have significant differences for the different comparing orders of $i>5$. Therefore, in practice, we may use a level bound $LvBound$ to serve as a termination condition, that is, if the branch and bound procedure reaches the level $LvBound$, we terminate it. Our algorithm to find the optimal scanning order is described in *Preprocessing of Algorithm* 1.

***Preprocessing of Algorithm* 1** (P, σ, $LvBound$): A branch and bound algorithm to find the optimal scanning order

Input: A pattern P, alphabet size σ and an integer $LvBound$.
Output: The optimal scanning order I_{OPT} and Δ_{OPT}.

Compute the scanning order I and its shifting function Δ used in Sunday's Algorithm. Set $AVGS_{OPT} = AVGS(\Delta)$.
Set $CheckPos[j]=0$ for all $1 \le j \le m$.
$(I_{OPT}, \Delta_{OPT}) = FindOpt\_branch\_and\_bound(1, I, AVGS_{OPT}, \sigma, LvBound, CheckPos)$.
Return I_{OPT} and its shifting function Δ_{OPT}.

FindOpt_branch_and_bound (i, I, $AVGS_{OPT}$, σ, $LvBound$, $CheckPos$).

Input: An integer i, an integer array I, an integer $AVGS_{OPT}$, alphabet size σ, an integer $LvBound$, and an integer array $CheckPos$.
Output: The optimal scanning order I_{OPT} and its shifting function Δ_{OPT}.

if $i = LvBound$ **then** **/* Termination conditions */**
  for $j=m$ to $j=1$ **do**
    If $CheckPos[j]=0$, set $I[i]=j$, $CheckPos[j]=1$, and $i=i+1$.
  end for
  Compute the shifting function Δ of I.
  If $AVGS(\Delta) > AVGS_{OPT}$, set $AVGS_{OPT} = AVGS(\Delta)$, $I_{OPT} = I$, and $\Delta_{OPT} = \Delta$.
  Return (I_{OPT}, Δ_{OPT}).
end if
if $i \le m$ **then**
  For every j, where $1 \le j \le m$, such that $CheckPos[j]=0$, set $I^j[i]=j$ and compute the shifting function $\Delta^j[1 \ldots i]$. Set the values of $\Delta^j[i+1 \ldots m+1] = m$.

while ($AVGS(\Delta^{j}) > AVGS_{OPT}$ for some j) **do** **/* bound */**
Find the j_{max} such that $AVGS(\Delta^{j_{max}})$ is the largest among all $1 \le j_{max} \le m$ and j_{max} is the largest.
Set $AVGS(\Delta^{j_{\max}}) = 0$ and $CheckPos[j_{\max}] = 1$. **/* branch */**
I_{OPT} =*FindOpt_branch_and_bound*($i+1$, $I^{j_{max}}$, $AVGS_{OPT}$, σ, $LvBound$, $CheckPos$).
Set $CheckPos[j_{\max}] = 0$.
end while
end if
else **/* *i*=*m*+1 */**
Compute the shifting function Δ of I.
If $AVGS(\Delta) > AVGS_{OPT}$, set $AVGS_{OPT} = AVGS(\Delta)$, $I_{OPT} = I$, and $\Delta_{OPT} = \Delta$.
Return (I_{OPT}, Δ_{OPT}).
end else

Our complete algorithm using optimal scanning order for the exact string matching problem is described in *Algorithm* 1.

***Algorithm* 1 (** P, T, σ, $LvBound$ **)**

Input: A pattern P, a text string T, alphabet size σ, and an integer $LvBound$.
Output: All the occurrences of P in T.

Compute $(I, \Delta) =$ *Preprocessing of Algorithm* 1 (P, σ, $LvBound$).
Set $i = 1$.
while $i <= n - m + 1$ **do**
Set $j = 1$.
while $j <= m$ **do**
if $p_{I[j]} \ne t_{i+I[j]-1}$ **then** exit the inner loop.
else set $j = j + 1$.
end while
if $j = m + 1$ **then** report the position i.
Set $i = i + \Delta[j]$.
end while

---

3 A Combined Algorithm of Algorithm 1 and HASHq Algorithm

The most recent survey [13] shows that HASH*q* algorithm [19] is very efficient for small alphabet. In this section, we combine the HASH*q* algorithm with our algorithm proposed in the previous section. As can be seen from our experimental results, the combined algorithm is more efficient than our proposed Algoriehm 1 and the HASH*q* algorithm.

The HASH*q* algorithm is similar to the Horspool algorithm. Given a window, it checks whether a suffix of the window is equal to a suffix of the pattern. If it is not, it slides the window; otherwise, it uses a very simple left-to-right comparison method to determine whether there is an exact match. To check whether a suffix of the window is equal to a suffix of the pattern, the HASH*q* algorithm uses a simple hashing function h to transform a substring with length q into an integer value within 0 and 255. Therefore, if two strings A and B are equal, then $h(A) = h(B)$. But if $h(A) = h(B)$, then it does not imply $A = B$. Thus, if $h(W(m-q+1, m)) = h(P(m-q+1, m))$, we start to determine whether there is an exact match.

The hashing function is to serve as a filtering mechanism. It also can help us to decide the number of steps to slide the window. Suppose that the length of the suffix is q and that i is the largest integer such that $i \ne m$ and $h(W(m-q+1, m)) = h(P(i-q+1, i))$. Then we slide the window to the right by $m - i$ steps. The algorithm performs a pre-processing on the pattern P to derive the sliding table. The sliding table is of length 256. For all $0 \le x \le 255$, the preprocessing constructs a sliding table *shift* with $shift[x] = m - i$ if there exists $p_{i-q+1} p_{i-q+2} \ldots p_i$ which is the rightmost substring of P such that $h(p_{i-q+1} p_{i-q+2} \ldots p_i) = x$ and $shift[x] = m - q$, otherwise, where $q \le i < m$. For $q \le i < m$, we compute $h(p_{i-q+1} p_{i-q+2} \ldots p_i)$. For

$i = m$, we let $x_m = h(p_{m-q+1} p_{m-q+2} \cdots p_m)$. Then $shift[x_m] = m - m = 0$. In addition, the preprocessing uses another variable $sh1$ with $sh1 = shift[h(p_{j-q+1} p_{j-q+2} \cdots p_j)]$ if $p_{j-q+1} p_{j-q+2} \cdots p_j$ is the second rightmost substring of P such that $h(p_{j-q+1} p_{j-q+2} \cdots p_j) = x_m$ and $sh1 = m - q$, otherwise, where $q \le j < m$.

For a window $W(1, m)$ in the searching phase, the HASH*q* first checks if $shift[h(w_{m-q+1} w_{m-q+2} \ldots w_m)]$ is equal to 0 or not. That is, it checks if the hashing value of the suffix with length q of W is equal to the hashing value of the suffix with length q of P. If $shift[h(w_{m-q+1} w_{m-q+2} \ldots w_m)]$ is not equal to 0, then the HASH*q* algorithm slides the window $shift[h(w_{m-q+1} w_{m-q+2} \ldots w_m)]$ steps to the right. Otherwise, it compares the characters of the window against those of P from left to right. After it, the HASH*q* slides the window $sh1$ steps to the right. Basically, the HASH*q* algorithm can be considered as a filtering algorithm. It only checks the windows with the value $shift[h(w_{m-q+1} w_{m-q+2} \ldots w_m)] = 0$. Therefore, it would be very efficient if most of the windows are filtered out.

The HASH*q* algorithm is very good at filtering. But it uses a straightforward algorithm to determine whether $W = P$. It does not consider the order of character comparisons. The value of $sh1$ may be small for some patterns and this makes the sliding of the window inefficient. For example, consider $P = gcataaaa$ and $q = 3$. The value of $sh1$ is 1 because $h(p_{m-q} p_{m-q+1} \cdots p_{m-1})$ = $h(aaa)$ = $h(p_{m-q+1} p_{m-q+2} \cdots p_m)$, where $m = 8$ in this example. Consider the window $W = gcgtaaaa$, the HASH*q* algorithm will compare the characters w_1, w_2 and w_3 with p_1, p_2 and p_3, respectively. It finds a mismatch when comparing $w_3 = g$ with $p_3 = a$ and then slides the window to the right by one step since $sh1 = 1$. If we use the idea of our proposed algorithm in the previous section to find a good comparing order, we may slide the window more steps in this case.

Below, we try to find a good comparing order I as well as a sliding distance to replace the checking step of HASH*q* algorithm. Suppose that $h(P(m-q+1, m)) = h(W(m-q+1, m))$. We first define a new sliding distance Δ_q. This sliding distance is similar to the sliding distance Δ introduced in the previous section. For all $1 \le j \le m$, $\Delta_q[j]$ is the minimum value of *mshift* with satisfying the following three conditions:

(1) Either $(I[i] - mshift) < 1$ or $p_{I[i]} = p_{(I[i]-mshift)}$ for all $1 \le i \le j-1$.
(2) Either $(I[j] - mshift) < 1$ or $p_{I[j]} \ne p_{(I[j]-mshift)}$.
We now add another rule:
(3) Either $(m - mshift) < q$ or $h(p_{m-q+1-mshift} \cdots p_{m-mshift}) = h(p_{m-q+1} \cdots p_m)$.

For $j = m+1$, $\Delta_q[j]$ is the minimum value of *mshift* that satisfies the following conditions.

(1) Either $(I[i] - mshift) < 1$ or $p_{I[i]} = p_{(I[i]-mshift)}$ for all $1 \le i \le j-1$.
(2) Either $(m - mshift) < q$ or $h(p_{m-q+1-mshift} \cdots p_{m-mshift}) = h(p_{m-q+1} \cdots p_m)$.

Consider the example that $P = gcataaaa$ and $q = 3$. A comparing order I and the sliding distance Δ_q are shown in the following.

j	1	2	3	4	5	6	7	8	9
$I[j]$	5	1	2	3	4	6	7	8	
$\Delta_q[j]$	1	6	6	6	6	6	6	7	8

In this example, $\Delta_q[2] = 6$ because $mshift = 6$ is the minimum value to satisfy the required three conditions, as shown as follows.

(1) $I[1] - 6 = 5 - 6 < 1$.
(2) $I[2] - 6 = 1 - 6 < 1$.
(3) $m - 6 = 8 - 6 < q = 3$.

It can be verified that $\Delta[2] = 2$ since it does not need to meet the condition 3 required by Δ_q. Note that it is not hard to see that $\Delta_q[i] \ge \Delta[i]$ for all $1 \le i \le m$.

In this combined algorithm, we do not have to use the branch and bound algorithm introduced in the previous section to find the optimal comparing order I_{OPT} with the largest $AVGS(\Delta_q)$ for the entire pattern. Suppose that $h(p_{m-q+1} p_{m-q+2} \cdots p_m) = h(w_{m-q+1} w_{m-q+2} \ldots w_m)$. It means that the substring $p_{m-q+1} p_{m-q+2} \cdots p_m$ may have very high probability to be equal to $w_{m-q+1} w_{m-q+2} \ldots w_m$. If we compare the characters of $w_{m-q+1} w_{m-q+2} \ldots w_m$ in the very beginning, we would need to compare more characters to find a mismatch if it exists. Note that this is also the reason that the HASH*q* algorithm compares the characters of the window from left to right. Thus, in our combined algorithm, we set $I[i] = i$ for $m-q+1 \le i \le m$ and find the optimal comparing order $I[i]$ for $1 \le i \le m-q$ such that $AVGS(\Delta_q)$ is the maximal.

Consider the example with $P = gcataaaa$ and $q = 3$. The optimal comparing order I and the sliding distance Δ_q used in our combined algorithm are shown in the following.

j	1	2	3	4	5	6	7	8	9
$I[j]$	5	1	2	3	4	6	7	8	
$\Delta_q[j]$	1	6	6	6	6	6	6	7	8

In this example, we first set $I[6] = 6$, $I[7] = 7$, $I[8] = 8$ because $q = 3$, and then find the optimal comparing order for the positions 1 to 5. In the searching phase, we first compare p_5 with w_5 first. If $p_5 = w_5$, we then compare p_1 with w_1 and so on. Suppose that the mismatch occurs at the second comparison, i.e., $p_1 \neq w_1$. Then we can slide the window to the right by $\Delta_q[2]=6$ steps. Suppose that the window $W = gcgtaaaa$. Then, we can find the first mismatch occurring at w_3 and hence we can slide the window $\Delta_q[4]=6$ steps to the right. Note that $\Delta_q[4]=6$ is larger than the shift value $sh1=1$ which is used in the original HASH*q* algorithm.

For the patterns with $sh1 > (m/2)$, we use the original HASH*q* algorithm for the searching phase. For the patterns with $sh1 \leq (m/2)$, we use our optimal comparing order to improve the efficiency for sliding the window.

The following *Algorithm* 2 is the combination of *Algorithm* 1 and the HASH*q* algorithm, where *preprocessing_step_of_HASHq* is a subroutine used in the HASH*q* algorithm to find the sliding table *shift* and *sh*1, and *searching_step_of_HASHq* is a subroutine of the HASH*q* algorithm to perform the searching phase.

The *Preprocessing* of *Algorithm* 2 is obtained from *Preprocessing* of *Algorithm* 1 by the following modifications.

Line 1: Set $I[i]=i$ for $m-q+1 \leq i \leq m$ and then compute the comparing order $I[i]$ for $1 \leq i \leq m-q$ by using the Sunday's algorithm. Compute Δ_q for $I[i]$. Set $AVGS_{low} = AVGS(\Delta_q)$.

Line 2: Set $CheckPos[j]=0$ for $1 \leq i \leq m-q$ and $CheckPos[i]=1$ for $m-q+1 \leq i \leq m$.

Substitute Δ by Δ_q in the functions *Preprocessing* of *Algorithm* 1 and *FindOpt_branch_and_bound*.

***Algorithm* 2 (** P, T, σ, $LvBound$, q **)**

Input: A pattern P, a text string T, alphabet size σ, an integer $LvBound$ and an integer q.
Output: All the occurrences of P in T.

$(shift, sh1) = preprocessing\_step\_of\_HASH_q(P,q)$.
if $sh1 > (m/2)$ **then** do $searching\_step\_of\_HASH_q(P,T,q,shift,sh1)$ and exit.
Compute $(I,\Delta_q) =$ *Preprocessing of Algorithm* 2 (P, σ, $LvBound$, q).
Set $i=1$.
while $i <= n-m+1$ **do**
  Set $sh=1$.
  while $sh \neq 0$ and $i <= n-m+1$ **do**
    Set $sh = shift[h(w_{i+m-q}w_{i+m-q+1}\ldots w_{i+m-1})]$.
    Set $i = i + sh$.
  end while
  Set $j=1$.
  while $j <= m$ **do**
    if $p_{I[j]} \neq t_{i+I[j]-1}$ **then** exit the inner loop.
    else set $j = j+1$.
  end while
  if $j = m+1$ **then** report the position i.
  Set $i = i + \Delta_q[j]$.
end while

4 Experiments

In our experiments, we randomly generated a text T of size $n =$ 1G and patterns of size $m \in \{5, 10, 15, 20, 25, 30, 35, 40\}$ by using different alphabet sizes with $|\Sigma| \in \{2, 4, 8, 26\}$. We tested the performances of Algorithm 1 (Alg1 for short) and Algorithm 2 with $q=3$ (Alg2_H3 for short) by using $LvBound = 4$. We first tested the performance our Algorithm 1 by comparing the number of its character comparison and the total running time (including the pattern preprocessing time and the text searching time) with those obtained by the other algorithms which use the different comparing orders: (1) KMP algorithm [17] (KMP) with left-to-right comparing order. (2) Boyer-Moore algorithm [3] without the bad character rule (BM-bc) that uses right-to-left comparing order. (3) The Sunday's maximal shift algorithm [24] (MS). The experimental results are shown in Table 1. Next, we compared the total running times of our algorithms with those of other algorithms which perform efficiently in practice. The tested algorithms are listed follows. Boyer-Moore algorithm [3] (BM), shift-and algorithm [4] (SA), TVSBS algorithm [27] (TVSBS), EBOM algorithm [12] (EBOM), Horspool algorithm [16] (H80), tuning BNDM algorithm with 2-Grams [11] (SBNDMq2), FJS algorithm [10] (FJS), and HASH*q* algorithm with $q = 3$ [19] (HASH3). The results are shown in Tables 2-3. Note that for these algorithms, we used the C codes which were implemented and used in [13]. The running time was measured by using hardware cycle counter and averaged over 100 random patterns in each experiment.

Table 1. The number of character comparisons (million)/the total running time (sec) for the algorithms with different scanning orders.

Alphabet size	*m*	KMP	BM-bc	MS	Alg1
2	5	1338/1121	860/755	877/796	**795/717**
	15	1329/1107	510/450	492/476	**374/359**
	25	1320/1107	411/366	404/397	**260/262**
	35	1331/1106	356/319	349/351	**216/222**
4	5	1196/905	732/553	747/565	**650/507**
	15	1198/907	531/402	423/320	**336/267**
	25	1197/905	443/337	319/243	**243/195**
	35	1200/908	421/321	297/227	**199/163**
8	5	1108/562	781/397	795/405	**686/359**
	15	1107/561	610/313	479/248	**417/221**
	25	1109/561	534/274	358/188	**308/167**
	35	1111/562	498/257	293/155	**245/135**
26	5	1037/360	898/313	905/315	**839/295**
	15	1038/360	752/264	694/244	**616/218**
	25	1038/359	653/229	553/197	**519/185**
	35	1037/360	624/221	465/167	**435/156**

Table 2. The comparison of total running time (sec) for $|\Sigma| = 2$.

m	5	10	15	20	25	30	35	40
BM	918	678	548	495	445	412	387	377
SA	**366**	359	349	353	365	374	220	210
TVSBS	765	791	807	772	796	835	821	797
EBOM	707	425	305	236	194	166	144	128
H80	970	992	1059	1017	1020	1005	1023	1021
SBNDMq2	641	334	222	**166**	**133**	**114**	**106**	**106**
FJS	914	1102	1142	1090	1134	1151	1132	1127
HASH3	468	269	215	209	195	196	193	198
Alg1	706	465	356	298	258	235	219	211
Alg2_H3	464	**254**	**198**	171	155	147	138	132

Table 3. The comparison of total running time (sec) for $|\Sigma|=4$.

m	5	10	15	20	25	30	35	40
BM	545	403	379	327	328	316	320	303
SA	374	374	374	373	373	373	224	224
TVSBS	386	261	214	188	176	164	165	160
EBOM	244	179	141	115	96	83	74	66
H80	483	351	340	312	322	318	340	328
SBNDMq2	**240**	168	124	97	79	**67**	**64**	64
FJS	595	522	525	491	515	506	514	514
HASH3	288	132	99	84	74	70	69	66
Alg1	492	329	260	221	191	171	159	151
Alg2_H3	287	**131**	**98**	**83**	**74**	68	65	**62**

The experimental results can be summarized as follows:

(1) According to Table 1, our proposed algorithm Alg1, improves the Sunday's maximal shift algorithm (MS) in all cases and is also better than the other algorithms using different comparing orders, such as KMP and BM-bc algorithms.
(2) Our algorithm Alg2_H3 improves the HASH3 algorithm in all cases.
(3) Comparing to other algorithms, our algorithm Alg2_H3 is most efficient for $m=5$ to 15 when $|\Sigma|=2$ and for $m=10$ to 25 when $|\Sigma|=4$. Note that the cases with $m=5$ to 15 and $|\Sigma|=2$ are the most time-consuming.

5 Conclusion and Future Research

In this paper, we proposed a branch and bound algorithm to find the optimal comparing order to minimize the number of character comparisons. Our experimental results have shown that this algorithm indeed has the smallest number of character comparison in all experimental cases, especially when the size of alphabet is small. In addition, we proposed another algorithm by combining our approach of computing an optimal comparing order with the HASH*q* algorithm and showed that this algorithm is most efficient among all of the tested algorithms for some cases. It will be interesting future work to analyze the time complexity of our branch and bound algorithm or to find a polynomial algorithm for finding the optimal comparing order. It would also be interesting to analyze the average-case time complexity of the string matching algorithm using the optimal comparing order.

Reference

[1] Apostolico, A. and Crochemore, M., Optimal canonization of all substrings of a string, Information and Computation, Vol. 95, 1991, pp. 76-95.
[2] Apostolico, A. and Giancarlo, R., The Boyer-Moore-Galil string searching strategies revisited, SIAM Journal on Computing, Vol. 15, 1986, pp. 98-105.
[3] Boyer, R.S. and Moore, J.S., A fast string searching algorithm, Communications of the ACM, Vol. 20, 1977, pp. 762-772.
[4] Baeza-Yates, R. and Gonnet, G. H., A new approach to text searching, Communications of the ACM, Vol.35, No. 10, 1992, pp.74-82.
[5] Baeza-Yates, Ricardo A. and Régnier, Mireille, Average Running Time of the Boyer Moore Horspool Algorithm, Theoretical Computer Science, Vol. 92, 1992, pp.19-31.
[6] Colussi, L., Correctness and efficiency of the pattern matching algorithms, Information and Computation, Vol. 95, No. 2, 1991, pp. 225-251.
[7] Colussi, L., Fastest pattern matching in strings, Journal of Algorithms, Vol. 16, No. 2, 1994, pp. 163-189.
[8] Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W. and Rytter, W., Speeding up two string matching algorithms, Algorithmica, Vol. 12, 1994, pp. 247-267.
[9] Charras, C., Lecroq, T. and Pehoushek, J.D., A very fast string matching algorithm for small alphabets and long patterns, in Proceedings of Combinatorial Pattern Matching, 1998, pp. 55-64,

Springer Berlin Heidelberg.
[10] Franek, Frantisek, Jennings, Christopher G., and Smyth, W.F., A simple fast hybrid pattern-matching algorithm, Journal of Discrete Algorithms, Vol. 5, 2007, pp. 682-695.
[11] Ďurian, B., Holub, J., Peltola, H., and Tarhio, J., Tuning BNDM with q-grams, in Proceedings of ALENEX, pp.29-37.
[12] Faro, S. and Lecroq, T., Efficient variants of the backward-oracle-matching algorithm, International Journal of Foundations of Computer Science, Vol. 20, No. 6, 2009, pp. 967-984.
[13] Faro, S. and Lecroq, T., The exact online string matching problem: A review of the most recent results, ACM Computing Surveys, Vol. 45, 2013, pp. 1-42.
[14] Galil, Z. and Giancarlo, R., On the exact complexity of string matching: upper bounds, SIAM Journal on Computing, Vol. 21, No. 3, 1992, pp. 407-437.
[15] Galil, Z. and Seiferas, J., Time-space optimal string matching, Journal of Computer and System Science, Vol. 26, No. 3, 1983, pp. 280-294.
[16] Horspool, R.N., Practical fast searching in strings, Software - Practice & Experience, Vol. 10, No. 6, 1980, pp. 501-506.
[17] Knuth, D.E., Morris (Jr), J.H. and Pratt, V.R., Fast pattern matching in strings, SIAM Journal on Computing, Vol. 6, No. 2, 1977, pp. 323-350.
[18] Lecroq, T., A variation on the Boyer-Moore algorithm, Theoretical Computer Science, Vol. 92, No. 1, 1992, pp. 119-144.
[19] Lecroq, T., Fast Exact String Matching Algorithms, Information Processing Letters, Vol. 102, 2007, pp. 229-235.
[20] Chia Wei Lu and R. C. T. Lee, An exact string matching algorithm based upon selective matching order and branch and bound approach, Proc. of the 30th Workshop on Combinatorial Mathematics and Computation Theory, 2013, pp. 131-137.
[21] Navarro, G., Nr-grep: a fast and flexible pattern-matching tool, Software - Practice & Experience, Vol. 31, 2001, pp. 1265–1312.
[22] Navarro, G. and Raffinot, M., Fast and flexible string matching by combining bit-parallelism and suffix automata, Journal of Experimental Algorithmics, Vol. 5, No. 4, 2000.
[23] Peltola, H. and Tarhio, J., Alternative algorithms for bit-parallel string matching, in Proceedings of String Processing and Information Retrieval, Vol. 2857, 2003, pp.80-93, Springer Berlin Heidelberg.
[24] Sunday, D.M., A very fast substring search algorithm, Communications of the ACM, Vol. 33, No. 8, 1990, pp. 132-142.
[25] Smith, P.D., Experiments with a very fast substring search algorithm, Software - Practice & Experience, Vol. 21, No. 10, 1991, pp. 1065-1074.
[26] Simon, I., String matching algorithms and automata, in Proceedings of 1st American Workshop on String Processing, R.A. Baeza-Yates and N. Ziviani ed., 1993, pp. 151-157, Universidade Federal de Minas Gerais, Brazil.
[27] Thathoo, R., Virmani, A., Sai Lakshmi, S., Balakrishnan, N. and Sekar, K., TVSBS: A fast exact pattern matching algorithm for biological sequences, Current Science, Vol.91, No. 1, 2006, pp.47-53.