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

Chia Wei Lu\textsuperscript{1}, Chin Lung Lu\textsuperscript{1} and R. C. T. Lee\textsuperscript{1,*}

\textsuperscript{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 \( P = p_{1}p_{2}\ldots p_{n} \) and a text \( T = t_{1}t_{2}\ldots t_{m} \), \( n \geq m \), 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 \( W = T(1, m) = t_{1}t_{2}\ldots t_{m} \) 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 \( P(1,m) = p_{1}p_{2}\ldots p_{n} \), \( D(i) \) 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 \( P = actagtgtagt \) \( D(i) \)'s are also shown in the following table.

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td>a</td>
<td>c</td>
<td>t</td>
<td>a</td>
<td>g</td>
<td>t</td>
<td>g</td>
<td>c</td>
<td>t</td>
<td>a</td>
<td>g</td>
<td>t</td>
</tr>
<tr>
<td>( D(i) )</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

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.
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 \( p_{i,j} \) matches with \( w_{j,i} \) for \( 1 \leq i \leq j-1 \). If we slide the window \( x \) steps to the right, then \( p_{i,j} \) will be aligned with \( w_{j,i} \) for each \( 1 \leq i \leq j-1 \) if \( I[i]-x > 0 \). Note that if \( I[i]-x \leq 0 \), then \( w_{j,i} \) will not be aligned with any character in \( P \). Since \( w_{j,i} = p_{i,j} \) for all \( 1 \leq i \leq j-1 \) and \( w_{j,i} \neq p_{i,j} \), we have \( w_{j,i} = p_{i,j} = p_{i(j),j-i}(i) \) for all \( 1 \leq i \leq j-1 \) if \( I[i]-x > 0 \) and \( w_{j,i} \neq p_{i(j),j-i}(i) \) if \( I[j]-x > 0 \). Therefore, we can decide the value of \( x \) by satisfying \( p_{i,j} = p_{i(j),j-i}(i) \) for all \( 1 \leq i \leq j-1 \) if \( I[i]-x > 0 \) and \( p_{i,j} \neq p_{i(j),j-i}(i) \) 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 \( \Delta \) used in Sunday algorithm [24] is defined as follows. For all \( 1 \leq j \leq m \), \( \Delta[j] \) is the minimum \( m shift \), where \( m shift \) is a positive integer, such that the following two conditions are satisfied:

Condition (1) Either \((I[i]-m shift) < 1 \) or \( p_{i,j} = p_{i(j),j-i}(i) \) for all \( 1 \leq i \leq j-1 \).

Condition (2) Either \((I[i]-m shift) < 1 \) or \( p_{i,j} \neq p_{i(j),j-i}(i) \).

For \( j = m+1 \), \( \Delta[j] \) is the minimum value of \( m shift \) such that either \((I[i]-m shift) < 1 \) or \( p_{i,j} = p_{i(j),j-i}(i) \) for all \( 1 \leq i \leq j-1 \). We denote the values of \( \Delta m = d, \Delta 2 = d, \ldots, \Delta m+1 = d \) by \( \{d, d, \ldots, d\} \).

For the example of \( P = agcca \) and \( I[5,3,2,4,1] \), it can be verified that \( \Delta m = d, \Delta 2 = d, \Delta 3 = d \). Note that if the comparing order is from left to right, i.e. \( I[1,2,\ldots,m] \), \( \Delta \) is equal to the sliding function used in KMP algorithm. If the comparing order is from right to left, i.e. \( I[m, m-1,\ldots,1] \), \( \Delta \) 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 \( \Delta \) can be computed in preprocessing. The algorithm for finding the sliding distance \( \Delta \) 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 \( P = aaaaaa \) and the alphabet \( \Sigma = \{a,c,g,t\} \). The Sunday algorithm will first compare \( p_5 = t \) with \( w_5 \) of the window. If \( p_5 = w_5 \), it then compares \( p_4 \) with \( w_4 \). Suppose that the first mismatch is found at the second comparison, i.e. \( p_4 \neq w_4 \). We then slide the window seven steps to the right. If the first mismatch occurs at the first comparison, i.e. \( p_5 \neq w_5 \), we can slide the window only one step to the right because \( p_5 \neq p_6 \). However, in this example, suppose that we compare \( p_3 \) with \( w_3 \) first and then \( p_6 \) with \( w_6 \). If \( p_3 = w_3 \) and \( p_6 \neq w_6 \), we can slide the window only one step to the right because \( p_3 = p_6 = a \) and \( p_6 \neq p_5 \). If \( p_3 \neq w_3 \) and \( p_6 \neq w_6 \), 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 \( x = \neq y \) is \( 1/4 \) and that of \( x = y \) is \( 3/4 \). Therefore, if we compute \( p_3 \) with \( w_3 \) first and then \( p_6 \) with \( w_6 \), 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 \( p_3 \) with \( w_3 \) first and then \( p_6 \) with \( w_6 \), 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 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 \( (3/4) \cdot (1 + (3/4) \cdot 7 = 33/16 \), and that for comparing \( p_3 \) with \( w_3 \) first and then \( p_6 \) with \( w_6 \) is \( (3/4) \cdot (3/4) \cdot 5 + (3/4) \cdot 1 = 63/16 \). 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 \( \Delta \). Assume that the first mismatch occurs at the position \( I[i] \). This means that \( p_{i,j} = w_{j,i} \) for \( 1 \leq j \leq (i-1) \) and \( p_{i,j} \neq w_{j,i} \). Then we can slide the window \( \Delta[i] \) steps to the right. Let \( \sigma \) be the alphabet size. The probability of \( p_{i,j} = w_{j,i} \) for all \( 1 \leq j \leq (i-1) \) and \( p_{i,j} \neq w_{j,i} \) 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 \( \Delta \), we define a function \( AVGS \) to compute the expected number of steps to slide a window as follows.

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

For example, let \( P = aaaaaa \). Consider the comparing order \( I \) and its sliding distance, denoted by \( \Delta \), which are shown in the following table.
The 31st Workshop on Combinatorial Mathematics and Computation Theory

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>$j$</th>
<th>$I_j$</th>
<th>$\Delta_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Then, by the definition, we have $AVGS(\Delta) = (3/4) \cdot 5 + (1/4) \cdot (3/4) \cdot 1 + (1/4)^2 \cdot (3/4) \cdot 7 + \ldots + (1/4)^6 \cdot (3/4) \cdot 6 + (1/4)^7 \cdot 6 \approx 2.44$. By comparing $AVGS(\Delta)$ with $AVGS(\Delta)$, we may conclude that the comparing order $I_j$ is better than $I_j$ because $AVGS(\Delta) > AVGS(\Delta)$.

If we can find the optimal comparing order $I_{opt}$ such that its sliding distance, denoted by $\Delta_{opt}$, 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 $I_j[4, 3, 2, 1]$ and its sliding distance is $\Delta_j[4, 4, 4, 4]$. If we use this comparing order, we have $AVGS(\Delta) = 1.75$. Then we can use it as a lower bound of $AVGS_{opt}$, denoted by $AVGS_{low}$, that is, $AVGS_{low} = AVGS(\Delta) = 1.75$. 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 $1 \leq j < i$ have been determined. Let $\Delta_{k}$ denote the sliding distance $\Delta$ for setting $I[i] = j$ at level $i$, $\Delta_{k}[k] = \Delta[k]$ for $1 \leq k \leq i$, and $\Delta_{k}[k] = m$ for $i < k \leq m + 1$. 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 \( \Delta_{a,j} \)'s for \( 1 \leq j \leq 4 \). Consider the case where \( p_{i} \) is the first character to be compared. If \( p_{i} \) mismatches with its corresponding character in a window, we can slide the window 2 steps to the right because \( p_{0} = p_{i} = a \). Therefore, we set \( \Delta_{a,0} = 2 \). As for the values of \( \Delta_{a,j}[k]'s \) for \( 1 < k \leq 5 \), we set all of them as \( m = 4 \) so that \( AVGS(\Delta_{a,j}) \) is a upper bound of the \( AVGS' \)'s in this branch. That is, if \( AVGS(\Delta_{a,j}) \) is smaller than or equal to \( AVGS_{w_0} = 1.75 \), 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 \( p_{i} \) is the first compared character, there are three possibilities to choose one of \( p_{1}, p_{2}, \) and \( p_{3} \) as the next compared character. For each possibility, we compute the values of \( \Delta_{b,j}, \Delta_{c,j}, \) and \( \Delta_{d,j} \) 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 \( \Delta \). If \( AVGS(\Delta) > AVGS_{w_0} \), we then set \( AVGS_{w_0} = AVGS(\Delta) \) 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 \( AVGS_{w_0} = 1.75 \). Thus, we set \( AVGS_{w_0} = 1.94 \) 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 \( AVGS_{w_0} = 1.94 \).

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** \( (P, \sigma, LvBound) \): A branch and bound algorithm to find the optimal scanning order

**Input:** A pattern \( P \), alphabet size \( \sigma \) and an integer \( LvBound \).

**Output:** The optimal scanning order \( I_{opt} \) and \( \Delta_{opt} \).

1. Compute the scanning order \( I \) and its shifting function \( \Delta \) used in Sunday’s Algorithm. Set \( AVGS_{opt} = AVGS(\Delta) \).
2. Set \( CheckPos[j] = 0 \) for all \( 1 \leq j \leq m \).
3. \( (I_{opt}, \Delta_{opt}) = \text{FindOpt_branch_and_bound}(1, I, AVGS_{opt}, \sigma, LvBound, CheckPos) \).
4. Return \( I_{opt} \) and its shifting function \( \Delta_{opt} \).

**FindOpt_branch_and_bound** \( (i, I, AVGS_{opt}, \sigma, LvBound, CheckPos) \).

**Input:** An integer \( i \), an integer array \( I \), an integer \( AVGS_{opt} \), alphabet size \( \sigma \), an integer \( LvBound \), and an integer array \( CheckPos \).

**Output:** The optimal scanning order \( I_{opt} \) and its shifting function \( \Delta_{opt} \).

1. if \( i = LvBound \) then /* Termination conditions */
2. for \( j = m \) to \( j = 1 \) do
3. \( \text{If} \ CheckPos[j] = 0, \text{set} I[i] = j, \ CheckPos[j] = 1, \text{and} i = i + 1. \)
4. end for
5. Compute the shifting function \( \Delta \) of \( I \).
6. if \( AVGS(\Delta) > AVGS_{opt} \), set \( AVGS_{opt} = AVGS(\Delta) \), \( I_{opt} = I \), and \( \Delta_{opt} = \Delta \).
7. Return \( (I_{opt}, \Delta_{opt}) \).
8. end if
9. if \( i \leq m \) then
10. For every \( j \), where \( 1 \leq j \leq m \), such that \( CheckPos[j] = 0 \), set \( I'[i] = j \) and compute the shifting function \( \Delta'[1...i] \). Set the values of \( \Delta'[i+1...m+1] = m \).
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.

1: Compute (I, Δ) = Preprocessing of Algorithm 1 (P, σ, LvBound).
2: Set i = 1.
3: while i <= n-m+1 do
4:     Set j = 1.
5:     while j <= m do
6:         if p_{i+j-1} ≠ t_{i+j-1} then exit the inner loop.
7:     else set j = j + 1.
8: end while
9: if j = m+1 then report the position i.
10: Set i = i + Δ[I].
11: end while

3 A Combined Algorithm of Algorithm 1 and HASHq Algorithm

The most recent survey [13] shows that HASHq algorithm [19] is very efficient for small alphabet. In this section, we combine the HASHq 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 Algorithm 1 and the HASHq algorithm.

The HASHq 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 HASHq 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 ≠ 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 ≤ x ≤ 255, the preprocessing constructs a sliding table shift with shift[x] = m−i if there exists p_{i+q},p_{i+q+1},...,p_m which is the rightmost substring of P such that h(p_{i+q+1},...,p_m) = x and shift[x] = m−q, otherwise, where q ≤ i < m. For q ≤ i < m, we compute h(p_{i+q},p_{i+q+1},...,p_m). For
i = m, we let \( x_m = h(p_{m-q}, p_{m-q+1}, \ldots, p_m) \). Then
\[
\text{shift}(x_m) = m - m = 0.
\]
In addition, the preprocessing uses another variable \( shl \) with \( shl = \text{shift}(h(p_{j-q}, p_{j-q+1}, \ldots, p_j)) \) if \( p_{j-q}, p_{j-q+1}, \ldots, p_j \) is the second rightmost substring of \( P \) such that
\[
h(p_{j-q}, p_{j-q+1}, \ldots, p_j) = x_m \text{ and } shl = m - q, \text{ otherwise, where } q \leq j < m.
\]
For a window \( W(1, m) \) in the searching phase, the HASHq first checks if \( \text{shift}(h(w_{m-q}, w_{m-q+1}, \ldots, w_m)) \) 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 \( \text{shift}(h(w_{m-q}, w_{m-q+1}, \ldots, w_m)) \) is not equal to 0, then the HASHq algorithm slides the window \( \text{shift}(h(w_{m-q}, w_{m-q+1}, \ldots, w_m)) \) steps to the right. Otherwise, it compares the characters of the window against those of \( P \) from left to right. After it, the HASHq slides the window \( shl \) steps to the right. Basically, the HASHq algorithm can be considered as a filtering algorithm. It only checks the windows with the value \( \text{shift}(h(w_{m-q}, w_{m-q+1}, \ldots, w_m)) = 0 \). Therefore, it would be very efficient if most of the windows are filtered out.

The HASHq 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 \( shl \) 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 \( shl \) is 1 because \( h(p_{m-q}, p_{m-q+1}, \ldots, p_m) = h(aa) = h(w_{m-q}, w_{m-q+1}, \ldots, p_m) \), where \( m = 8 \) in this example. Consider the window \( W = gcataaaa \), the HASHq 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 \( w_1 = g \) with \( p_1 = a \) and then slides the window to the right by one step since \( shl = 1 \). 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 HASHq algorithm. Suppose that
\[
h(P(m-q+1, m)) = h(W(m-q+1, m))
\]
We first define a new sliding distance \( \Delta_q \). This sliding distance is similar to the sliding distance \( \Delta \) introduced in the previous section. For all \( 1 \leq j \leq m \), \( \Delta_q[j] \) 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 \( 1 \leq i \leq j - 1 \).
2. Either \( |I[j] - mshift| < 1 \) or \( p_{I[j]} \neq p_{(I[j] - mshift)} \).
3. Either \( (m - mshift) < q \) or \( h(p_{m-q+1}, \ldots, p_m) = h(P(m-q+1, \ldots, p_m)) \).

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 \( 1 \leq i \leq j - 1 \).
2. Either \( (m - mshift) < q \) or \( h(p_{m-q+1}, \ldots, p_m) = h(P(m-q+1, \ldots, p_m)) \).

Consider the example that \( P = gcataaaa \) and \( q = 3 \). A comparing order \( I \) and the sliding distance \( \Delta_q \) 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 | 9 |
| \( \Delta_q[j] \) | 1 | 6 | 6 | 6 | 6 | 6 | 6 | 7 | 8 |

In this example, \( \Delta_q[1] = 6 \) because \( mshift = 6 \) is the minimum value to satisfy the required three conditions, as shown as follows.

1. \( |I[1] - 6| = 6 - 6 < 1 \).
2. \( |I[2] - 6| = 1 - 6 < 1 \).
3. \( m - 6 = 8 - 6 < 3 \).

It can be verified that \( \Delta_q[2] = 2 \) since it does not need to meet the condition 3 required by \( \Delta_q \). Note that it is not hard to see that \( \Delta_q[i] \geq \Delta_q[i] \) for all \( 1 \leq i \leq m \).

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 \( A V G S(\Delta_q) \) for the entire pattern. Suppose that \( h(p_{m-q+1}, \ldots, p_m) = h(W_{m-q+1}, \ldots, w_m) \). It means that the substring \( p_{m-q+1}, \ldots, p_m \) may have very high probability to be equal to \( w_{m-q+1}, \ldots, w_m \). If we compare the characters of \( w_{m-q+1}, \ldots, w_m \) 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 HASHq algorithm compares the characters of the window from left to right. Thus, in our combined algorithm, we set \( I[i] = i \) for \( m - q + 1 \leq i \leq m \) and find the optimal comparing order \( I[i] \) for \( 1 \leq i \leq m - q \) such that \( A V G S(\Delta_q) \) is the maximal.

Consider the example with \( P = gcataaaa \) and \( q = 3 \). The optimal comparing order \( I \) and the sliding distance \( \Delta_q \) 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 | 9 |
| \( \Delta_q[j] \) | 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.
searching phase, we first compare \( p_i \) with \( w_i \) first. If \( p_i = w_i \), we then compare \( p_j \) with \( w_j \) and so on. Suppose that the mismatch occurs at the second comparison, i.e., \( p_i \neq w_i \). Then we can slide the window to the right by \( \Delta_i[2] = 6 \) steps. Suppose that the window \( W = g c g t a a a a a . \) Then, we can find the first mismatch occurring at \( w_i \) and hence we can slide the window \( \Delta_i[4] = 6 \) steps to the right. Note that \( \Delta_i[4] = 6 \) is larger than the shift value \( sh = 1 \) which is used in the original HASHq algorithm.

For the patterns with \( sh > (m/2) \), we use the original HASHq algorithm for the searching phase. For the patterns with \( sh \leq (m/2) \), 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 HASHq algorithm, where \( \text{preprocessing\_step\_of\_HASHq} \) is a subroutine used in the HASHq algorithm to find the sliding table \( \text{shift} \) and \( sh_1 \), and \( \text{searching\_step\_of\_HASHq} \) is a subroutine of the HASHq 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 \( \Delta_q \) for \( I[i] \). Set \( \text{AVGS}_w = \text{AVGS}(\Delta_q) \).

Line 2: Set \( \text{CheckPos}[j] = 0 \) for \( 1 \leq i \leq m - q \) and \( \text{CheckPos}[i] = 1 \) for \( m - q + 1 \leq i \leq m \).

Substitute \( \Delta \) by \( \Delta_q \) in the functions \( \text{Preprocessing\_of\_Algorithm\_1} \) and \( \text{FindOpt\_branch\_and\_bound} \).

---

**Algorithm 2** (\( P, T, \sigma, \text{LvBound}, q \))

**Input:** A pattern \( P \), a text string \( T \), alphabet size \( \sigma \), an integer \( \text{LvBound} \) and an integer \( q \).

**Output:** All the occurrences of \( P \) in \( T \).

1: \( \text{(shift, shl)} = \text{preprocessing\_step\_of\_HASHq}(P, q) \).
2: if \( sh > (m/2) \) then do \( \text{searching\_step\_of\_HASHq}(P, T, q, \text{shift, shl}) \) and exit.
3: Compute \( (I, \Delta_q) = \text{Preprocessing of Algorithm 2} (P, \sigma, \text{LvBound}, q) \).
4: Set \( i = 1 \).
5: while \( i \leq n - m + 1 \) do
6: \( \text{sh} = 1 \).
7: while \( \text{sh} \neq 0 \) and \( i \leq n - m + 1 \) do
8: \( \text{sh} = \text{shift}[h(w_{i-m-q}w_{i-m-q+1} \cdots w_{i-m-1})] \).
9: \( i = i + \text{sh} \).
10: end while
11: Set \( j = 1 \).
12: while \( j \leq m \) do
13: if \( P_{j} \neq \text{match[1]} \) then exit the inner loop.
14: else set \( j = j + 1 \).
15: end while
16: if \( j = m + 1 \) then report the position \( i \).
17: Set \( i = i + \Delta_q[j] \).
18: 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 \( \text{LvBound} = 4 \). We first tested the performance of 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.

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

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

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

Springer Berlin Heidelberg.


