An Algorithm to the Nearest Neighbor String Searching Problem

Yi Kung, Shieh and R. C. T. Lee
Department of Computer Science
Nation Tsing Hua University, Hsinchu, Taiwan
d9762814@oz.nthu.edu.tw and rctlee@rctlee.cyberhood.net.tw

Abstract
In this study, we focus on the nearest neighbor string searching problem. The Nearest neighbor string searching problem is roughly, given a text string \( T = t_1t_2 \cdots t_n \) and a pattern string \( P = p_1p_2 \cdots p_m \), find the ending locations of all substrings in \( T \) whose edit distances with respect to \( P \) are the smallest among all substrings of \( T \). Our algorithm is based upon two concepts: (1) In a dynamic programming matrix \( D \) which is used in solve the approximate string matching problem, the difference between two near entries is at most 1. (2) Values on the matrix \( D \) can be computed along the diagonals. Our experimental results showed that our algorithm is efficient.

1. Introduction
The edit distance between two strings \( A \) and \( B \) is the minimum number of edit operations needed to convert \( B \) to \( A \), which is denoted as \( ed(A,B) \) [5]. An edit operation is either an insertion, a deletion or a substitution of a character. \( ed(A,B) \) can be computed by a classic dynamic programming algorithm in time \( O(|A| \times |B|) \) where \( |A| \) is the length of \( A \) [10], [18].

In this paper, for a string \( A = a_1a_2 \cdots a_i \), we define \( A(i) = a_i a_{i+1} \cdots a_j \), where \( 1 \leq i \leq j \leq |A| \).

The definition of the nearest neighbor string searching problem is: we are given a text \( T = t_1t_2 \cdots t_n \) and a pattern \( P = p_1p_2 \cdots p_m \), our job is to find the ending locations of all substrings \( S \) in \( T \) where \( ed(S,P) \) is the smallest among all substrings of \( T \). Each \( S \) is a nearest neighbor of \( P \) with respect to \( T \) and is denoted as \( NN(P,T) \).

We explain the relationship between the nearest neighbor string searching problem and the approximate string matching problem in Section 2. In Section 3, we introduce the alternative dynamic programming approach. Section 4 shows a property which is based upon the approach of Section 3. We introduce our algorithm in Section 5 and its further improvement in Section 6. Our experiments are shown in Section 7.

2. The Relationship between the Nearest Neighbor String Searching Problem and the Approximate String Matching Problem

The definition of the approximate string matching problem is: we are given a text string \( T = t_1t_2 \cdots t_n \), a pattern string \( P = p_1p_2 \cdots p_m \) and an error bound \( k \), and we have to find the all ending locations of the substrings \( S' \) in \( T \) such that \( ed(S,P) \leq k \).

Many algorithms have been proposed to solve this problem [1], [2], [3], [6], [7], [9], [12], [13], [14], [15], [17]. The different between the nearest neighbor string searching problem and the approximate string matching problem is the pre-specified error bound \( k \).

In fact, among these approximate string matching algorithms, the Sellers algorithm is quite unusual because it does not consider \( k \).

Before presenting the Sellers Algorithm, we define some notations. For two strings \( A(i,j) = a_ia_{i+1} \cdots a_j \) and \( B(l,j) = b_{l+1}b_{l+2} \cdots b_j \), let \( S_i \) denote a suffix of \( A(i,j) \) whose edit distance with \( B(l,j) \) is the smallest among all suffix of \( A(i,j) \). Let \( D(i,j) \) be the edit distance between \( S_i \) and \( B(l,j) \).

The Sellers Algorithm computes \( D(i,j) \) through the dynamic programming matrix \( D \) with size \((n+1) \times (m+1)\) by the recurrence:

\[
D(i,j) = \min(D(i-1,j) + 1, D(i,j-1) + 1, D(i-1,j-1) + 1)
\]

subject to the boundary conditions that \( D(0,j) = j \) for all \( j \) and \( D(i,0) = 0 \) for all \( i \). After computing \( D \)-matrix, the smallest value \( D(i,m) \) of the last row in the matrix is \( NN(D,P,T) \) and
location \( i \) of \( T \) is a solution. The algorithm is shown in Algorithm 1. Algorithm 1 can be computed in time \( O(n \times m) \).

**Algorithm 1.** The Sellers Algorithm to solve the nearest neighbor string searching problem.

**Input:** \( T = t_1 t_2 \cdots t_n \) and \( P = p_1 p_2 \cdots p_m \).

**Output:** The ending locations of all nearest neighbors of \( P \) with respect to \( T \).

// Pre-processing phase
for \( i = 0 \) to \( n \) do \( D(i, 0) = 0 \)
for \( j = 0 \) to \( m \) do \( D(0, j) = j \)

// Searching phase
for \( i = 0 \) to \( n \) do
for \( j = 0 \) to \( m \) do
  if \( i = p_j \) then \( eq = 0 \)
  else \( eq = 1 \)
  \( D(i, j) = \min(D(i-1, j)+1, D(i, j-1)+1, \)
  \( D(i-1, j-1)+eq) \)
end for
end for

In the last row of \( D \)-matrix, find and return all locations \( i \) such that \( D(i, m) \) is the smallest value.

For example, \( T = atcaacta \) and \( P = tega \). The matrix \( D \) is shown in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>t</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td>t</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

In Table 1, the minimum value of the last row is \( D(4, m) = D(4, 4) = 1 \). We know that there is a substring ending at location 4 of \( T \) whose edit distance with \( P \) is the smallest value among all substrings of \( T \). Hence, \( NND(P, T) = 1 \) and the solution is location 4. By tracing back, we can find that \( NNN(P, T) = tca \).

The Sellers Algorithm computes all entries of the matrix, and it is quite time-consuming. Thus out research focuses on how to ignore some computations.

We explain an important property of a \( D \)-matrix. That is:

\[ |D(i, j) - D(i+1, j)| \leq 1 \quad (2-1) \]

and \( |D(i, j) - D(i, j+1)| \leq 1 \).

In other words, the difference between two near entries \( D(i, j) \) and \( D(i, j+1) \) is -1, 0 or 1 [1], [3], [6], [13], [14]. We shall call this property the change property. We have the following claim:

**Claim 1:** If \( D(i_1, j_1) = a \), the smallest \( i_2 \), such that \( i_2 > i_1 \) and \( D(i_2, j_1) = b \) for \( b < a \), must equal to \( i_1 + (a - b) \) because it takes at least \( (a - b) \) steps for \( D(i, j) = a \) to decrease to \( b \).

Consider Table 1. Assume that we know \( NND(P, T) = 1 = D(4, m) \). After we obtained \( D(6, m) = 3 \), we can say that \( D(7, m) \) must be larger than \( NND(P, T) = 1 \) by Claim 1.

### 3. The Alternative Dynamic Programming Computation for the Approximate String Matching Problem

The Alternative Dynamic Programming Computation approach is to solve approximate string matching problem [2], [7], [13], [17].

A diagonal \( d \) in a matrix \( D \) is a continuous sequence of locations \( (i, j) \) where \( d = i - j \). For an error \( e \), let \( L_{d,e} \) denote the largest location \( j \) such that \( D(i, j) = e \) on the \( d \)-diagonal. Note that \( L_{d,e} = e \) if the error \( e \) is smaller than the smallest value on \( d \)-diagonal and \( L_{d,e} = m \) if the error \( e \) is larger than or equal to the largest value on \( d \)-diagonal.

Consider Table 1. On diagonal \( d = 2 \), there are five locations: \((2, 0)\), \((3, 1)\), \((4, 2)\), \((5, 3)\) and \((6, 4)\). We can see that \( L_{2,0} = 0 \), \( L_{2,1} = 2 \), \( L_{2,2} = 3 \) and \( L_{2,3} = 4 \). For the error \( e = -1 \), \(-1 \) is smaller than the smallest value on 2-diagonal, and \( L_{2,-1} = -1 \). For the error \( e = 4 \), \( 4 \) is larger than the largest value on 2-diagonal, and \( L_{2,4} = 4 = m \).

Our approach uses a very important property, called the Diagonal Non-decreasing Property.

**The Diagonal Non-Decreasing Property:** The values on a \( d \)-diagonal in a matrix \( D \) are non-decreasing [2], [6], [7], [13], [17]. That is: \( D(i+1, j+1) - D(i, j) \geq 0 \).

Therefore, we have the following claim:

**Claim 2:** If \( D(i, j) = a \) and \( D(i+1, j+1) = b \) on \( d \)-diagonal in a dynamic programming matrix \( D \), \( b - a \) is 0 or 1.
Consider the 2-diagonal of Table 1. \( D(2,0)=0 \) and \( D(3,1)=1 \), and we obtain \( D(3,1)-D(2,0)=1 \). \( D(3,1)=1 \) and \( D(4,2)=1 \), and we obtain \( D(4,2)-D(3,1)=1-1=0 \).

Suppose we want to solve the approximate string matching problem. We are given an error bound \( k \). On a \( d \)-diagonal, after we find \( L_{d,k}=j \), we do not have to compute \( L_{d,k+1} \) anymore based upon the Diagonal Non-decreasing Property. This is why the Alternative Dynamic Programming Computation approach is efficient to solve the approximate string matching problem.

For instance, let \( k=1 \). Again, \( T=atcaata \) and \( P=tagca \). The computation of the \( D \)-matrix is now shown in Table 2.

<table>
<thead>
<tr>
<th>( D )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( c )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( g )</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

In [7], it was pointed out that in order to find \( L_{d,e} \), we have to find \( L_{d-1,e-1} \), \( L_{d,e-1} \) and \( L_{d+1,e-1} \) first. Besides, we may use another important property which can be explained by an example. We know that \( D(1,0)=0 \). We also find that \( T(2,3)=P(1,2) \) and \( t_3=a \neq g = p_3 \). Then we are certain that \( D(3,2)=0 \) and \( D(4,3)=1 \). Therefore, we conclude that \( L_{1,0}=2 \).

This property is used to find every \( L_{d,e} \).

In [7], it was shown that \( L_{d,e} \) can be found by the following procedure:

\[
\begin{align*}
\text{end while} & \quad L_{d,e} = r + col \\
\end{align*}
\]

To understand why this procedure correctly gives \( L_{d,e} \), consult [6] and [7]. The while loop in the above procedure is to determine whether a substring of \( T \) is equal to a substring of \( P \) as we pointed above.

Once we obtain \( L_{1,0}=2 \), on 0-diagonal, from \( D(1,1) \) to \( D(3,3) \) are not computed. This is why the Alternative Dynamic Programming Computation approach is quite efficient.

The Alternative Dynamic Programming Computation approach to solve the approximate string matching problem is shown in Algorithm 2, which obtain the all \( L_{d,e} \) without computing \( D \)-matrix.

Algorithm 2. The Alternative Dynamic Programming Computation approach to solve the approximate string matching problem

**Input:** \( T=t_1t_2\cdots t_n \), \( P=p_1p_2\cdots p_m \) and \( k \)

**Output:** The ending locations of substrings \( S \) of \( T \) such that \( ed(S,P) \leq k \)

\[
\begin{align*}
&// \text{Pre-processing phase} \\
&\text{for } i=1 \text{ to } k+1 \text{ do} \\
&\quad \text{for } j=-1 \text{ to } i-1 \text{ do} \\
&\quad \quad L_{-i,j} = j \\
&\quad \text{end for} \\
&\text{end for} \\
&// \text{Searching phase} \\
&\text{for } c=0 \text{ to } n-m+k+1 \text{ do} \\
&\quad \text{for } e=0 \text{ to } k \text{ do} \\
&\quad \quad d = c - e \\
&\quad \quad r = \min(\max(L_{d-1,e-1}, L_{d,e-1}+1, L_{d+1,e-1}+1), m) \\
&\quad \quad \text{col} = 0 \\
&\quad \quad \text{while } r + \text{col} + d < n \text{ and } r + \text{col} < m \text{ and } t_{r+\text{col}+d+1} = P_{r+\text{col}+1} \text{ do} \\
&\quad \quad \quad \text{col} = \text{col} + 1 \\
&\quad \quad \text{end while} \\
&\quad \quad L_{d,e} = r + \text{col} \\
&\quad \text{if } L_{d,e} = m \text{ then Output location } d + m \text{ is a solution.} \\
&\quad \text{end for} \\
&\text{end for}
\end{align*}
\]

There are many approaches about improving the performance to obtain \( \text{col} \) for every \( L_{d,e} \) [2], [4], [6], [7], [11], [13], [16]. Most of them obtain \( \text{col} \) in time \( O(1) \).

4. The \( L_{d,e} \) Diagonal Triangular Property
From the above discussion, we can see that the Alternative Dynamic Programming Computation approach starts from find $L_{d,0}$ for all diagonal $d$. Then based upon this information, we compute $L_{d,e}$ and so on. In the following, we show that we can focus on any diagonal $d$ and find $L_{d,e}$ on this diagonal. The process is as follows:

1. We start with finding $L_{d,0}$ first. Since $L_{d-1,0} = L_{d,0} = L_{d+1,0} = -1$. We have $r = \max(L_{d-1,0}, L_{d,0}, L_{d+1,0}) = \max(-1, -1, -1) = 0$. Then we find the length $col$ of the longest common prefix between $T(r + d + 1, n)$ and $P(r + 1, m)$. We now have found $L_{d,0} = r + col$.

2. To find $L_{d,1}$, we have to find $L_{d-1,0}, L_{d,0},$ and $L_{d+1,0}$. Among them, $L_{d,0}$ has already been found. $L_{d-1,0}$ and $L_{d+1,0}$ can be found as we pointed out in the previous step.

3. To find $L_{d,2}$, we have to find $L_{d-1,1}, L_{d,1}$ and $L_{d+1,1}$, and so on. In the following, we show that we can approach starts from $L_{d,0}$, $L_{d,1}$ and $L_{d+1,0}$.

We thus have the following property, called the $L_{d,e}$ triangular property.

**The $L_{d,e}$ Diagonal triangular property:** To find $L_{d,e}$, we need only to find $L_{d-1,e-1}$, $L_{d,e-1}$ and $L_{d+1,e-1}$.

To give one example, suppose that we want to find $L_{10,2}$ on 10-diagonal, the values that we need to process is on the line as shown in Figure 1. On 10-diagonal, the current maximum error $e$ that we have obtained is $-1$ and $L_{10,1} = -1$.

![Figure 1. The diagonal $d = 10$ in table $L$.](image)

The finding of $L_{10,0}$: we need only to find the values in the triangular area as shown in Figure 2.

![Figure 2. The triangular area of $L_{10,0}$.](image)

The finding of $L_{10,1}$: we need only to find the values in the triangular area as shown in Figure 3.

![Figure 3. The values in the triangular area of $L_{10,1}$.](image)

The finding of $L_{10,2}$: we need only to find the values in the triangular area as shown in Figure 4.

![Figure 4. The triangular area of $L_{10,2}$.](image)

The procedure `diagonal_comp` is the algorithm to find the largest error on a diagonal $d$. As will be explained later, we may input a parameter `start_error`. This is useful when we already know a current maximum error in this diagonal. Assume that $L_{d,0}, L_{d,1}$, and $L_{d,2}$ have been obtained. The current maximum error that we already know is 3, and we can set `start_error` to 3. If we do not know any value on the $d$-diagonal, we use set `start_error` to be zero.

**Procedure** `diagonal_comp(d,start_error)`

**Input:** $d$ and `start_error`

**Output:** The largest error $e$ on $d$-diagonal $e = start_error$
while \( L_{d,e} \neq m \) do
  \( e = e + 1 \)
  if \( L_{d,e} \) is unknown then
    \( i = 0; j = d - e; j' = d + e \)
    while \( i \neq e \) do
      if \( L_{j,i} \) is unknown then
        \( r_1 = \min(\max(L_{j-1,i-1}), L_{j-1} + 1, L_{j+1,i-1} + 1),m \)
        \( col_1 = 0 \)
      while \( r_1 + col_1 + j < n \) and \( r_1 + col_1 < m \) and \( t_{r_1 + col_1 + j + 1} = p_{r_1 + col_1 + 1} \) then
        \( col_1 = col_1 + 1 \)
      end while
      \( L_{j,i} = r_1 + col_1 \)
    end if
    \( r_2 = \min(\max(L_{j-1,i-1}), L_{j-1} + 1, L_{j+1,i-1} + 1),m \)
    \( col_2 = 0 \)
    while \( r_2 + col_2 + j' < n \) and \( r_2 + col_2 < m \) and \( t_{r_2 + col_2 + j' + 1} = p_{r_2 + col_2 + 1} \) then
      \( col_2 = col_2 + 1 \)
    end while
    \( L_{j,j'} = r_2 + col_2 \)
  end if
  \( i = i + 1; j = j + 1; j' = j' - 1 \);
end while
\( r = \min(\max(L_{d-1,e-1}, L_{d-1} + 1, L_{d+1,e-1} + 1),m) \)
\( col = 0 \)
while \( r + col + d < n \) and \( r + col < m \) and \( t_{r + col + d + 1} = p_{r + col + 1} \) do
  \( col = col + 1 \)
end while
\( L_{d,e} = r + col \)
end if
end while
return \( e \)

For example, \( T = aatcagggcga \) and \( P = ggccgtaca \). By the procedure \textit{diagonal}_\textit{comp} with \textit{start}_\textit{error} = 0, on 2-diagonal, we obtain the result as shown in Table 3.

5. **Our Algorithm to Solve the Nearest Neighbor String Searching Problem**

The neighbor searching problem is defined as follows: We are given a text \( T = t_1t_2 \cdots t_n \) and a pattern \( P = p_1p_2 \cdots p_m \), we have to find all ending locations of substrings \( S \) in \( T \) such that \( ed(S,P) \) is the smallest among all substrings of \( T \).

In Section 2, we indicated that we can use the Sellers algorithm to solve this problem by finding the entire \( D \)-matrix. Essentially, we find \( D(i,m) \) for \( 1 \leq i \leq n \). The smallest \( D(i,m) \) is \( NND(P,T) \). The Sellers algorithm requires the entire \( D \)-matrix with size \( n \times m \), and it is not efficient enough. In the following, we shall present our algorithm which is more efficient than the Sellers algorithm.

There is no pre-specified error bound \( k \) in the nearest neighbor string searching problem. Therefore, the high-level structure of our algorithm is as follows:

1. We initially view the nearest neighbor string searching problem as approximate string matching problem with a temporary error bound \( tk = m \).
2. We solve the approximate string matching problem by using the procedure \textit{diagonal}_\textit{comp} with \textit{start}_\textit{error} = \( tk - 1 \). As soon as we find a \( D(i,m) = a < tk \), we set \( tk = a \).
3. If we find a \( D(i,m) = b > tk \), by using the small change property, we understand that we only have to find \( D(i',m) \) for \( i' = i + (b - tk) \). To find \( D(i',m) \), we again use \textit{diagonal}_\textit{comp} with \textit{start}_\textit{error} = \( tk \).

Assume that on a \( d \)-diagonal, \( D(i,m) = D(d + m,m) = x \). It is not necessary to care about \( L_{d,e} = m \) for \( e' > x \). Hence, let \( se_d \) be the smallest error \( e' \) on \( d \)-diagonal such that \( L_{d,e'} = m \).
On a \( d \)-diagonal, our goal is to find \( s_{e_d} \).

After we have obtained \( L_{d,s_{e_d}} \) on a \( d \)-diagonal, \( s_{e_d} \) has to be compared with \( tk \).

If \( s_{e_d} < tk \), we obtain that \( D(i,m) = D(d + m,m) = s_{e_d} < tk \). We obtain a new temporary error bound \( tk = s_{e_d} \) and have a new temporary solution \( d + m \). The next diagonal that we need to consider is diagonal \((d + 1)\).

If \( s_{e_d} = tk \), \( D(i,m) = D(d + m,m) = s_{e_d} + tk \). Location \( d + m \) is a temporary solution for the error \( tk \). The next diagonal that we need to consider is also diagonal \((d + 1)\).

If \( s_{e_d} > tk \), \( D(i,m) = D(d + m,m) = s_{e_d} + tk \). Let \( dis = s_{e_d} - tk \). By the small change property, we know that \( D(i',m) > tk \) for \( i < i' < t + dis \) as shown in Figure 5. The next diagonal that we need to consider is diagonal \((d + dis)\). This mechanism is called the hopping mechanism.

![Figure 5. The utilization of the small change property](image)

Note that \( tk = m \) initially. We consider diagonal \( d = m \) initially. The high-level description of our algorithm is as follows:

**High-level Description of our Algorithm**

\( tk = m \), \( d = -m \) and \( start\_error = tk - 1 = m - 1 \).

While \( d < n - m \), execute the procedure \( diagonal\_comp \) to obtain \( s_{e_d} \).

If \( s_{e_d} < tk \), \( tk = s_{e_d} \), \( start\_error = tk - 1 \) and \( d = d + 1 \).

If \( s_{e_d} = tk \), \( start\_error = tk - 1 \) and \( d = d + 1 \).

Otherwise, \( dis = s_{e_d} - tk \), \( start\_error = tk \) and \( d = d + dis \).

End of While

6. **A Further Improvement of Our Algorithm**

Our algorithm introduced above will jump \( dis \)'s diagonals on a \( d \)-diagonal when \( s_{e_d} > tk \). On a \( d \)-diagonal, we need to find all \( L_{d,e} \) for \( 0 \leq e \leq s_{e_d} \). Actually, if \( tk \) is found to be very small, we do not have to do that. That is, when we find a \( tk \) small enough, we give up utilizing the hopping mechanism and use an approximate string matching algorithm with \( tk \) as the error bound. We further improved our algorithm by using this idea and the approximate string matching algorithm used is the Alternative Dynamic Programming Computation algorithm with changing error bound.

When the error bound \( tk \) is not small enough, we use the hopping mechanism. When the error bound is quite small, we give up the hopping and utilize the Alternative Dynamic Programming Computation algorithm to solve the nearest neighbor string searching problem with error bound \( tk \) which may be changed.

There is a parameter, called switch value \( SV \) which is rather small and pre-specified by us. If we find a \( tk \) which is smaller than or equal to \( SV \) on a \( d \)-diagonal, we switch to the modified Alternative Dynamic Programming Computation approach to solve the nearest neighbor searching problem starting from \((d + 1)\)-diagonal. Algorithm 4 is for our improved algorithm.

**Algorithm 3.** An improved method to the nearest neighbor string searching problem

**Input:** \( T = t_1 t_2 \ldots t_n \), \( P = p_1 p_2 \ldots p_m \) and \( SV \).

**Output:** The ending locations of the nearest neighbors of \( P \) with respect to \( T \).

// Pre-processing phase

for \( i = 1 \) to \( m + 1 \) do

for \( j = -1 \) to \( i \) do

\( L_{-i,j} = -1 \)

end for

end for

for \( i = 0 \) to \( n + 1 \) do

\( L_{i, -1} = -1 \)

end for

// The part of the hopping mechanism

\( tk = m \); \( num = 0 \); \( d = -m \);

\( start\_error = m - 1 \); \( sw = 0 \);

while \( d \leq n - m \) and \( sw = 0 \) do

\( s_{e_d} = diagonal\_comp(d, start\_error) \)

if \( s_{e_d} < tk \) then

\( solution(0) = m + d \); \( num = 1 \);

\( tk = s_{e_d} \); \( d = d + 1 \);

\( start\_error = tk - 1 \);

if \( tk = SV \) then \( sw = 1 \)

else if \( s_{e_d} = tk \) then

\( solution(num) = m + d \); \( num = num + 1 \);

end if

end while

\( solution(num) = m + d \); \( num = num + 1 \);
\[ d = d + 1; \text{start\_error} = tk - 1; \]
else
\[ d = d + (se_d - tk); \text{start\_error} = tk; \]
end if
end while
// The part of modified Alternative Dynamic Programming Computation approach
for \( c = d \) to \( n - m + tk \) do
for \( e = 0 \) to \( tk \) do
\[ d = c - e \]
\[ r = \max(L_{d-1,c-1}, L_{d,c-1} + 1, L_{d+1,c-1} + 1) \]
\[ \col = 0 \]
while \( r + \col + d < n \) and \( r + \col < m \) and \( t_{r+\col+1} = p_{r+\col+1} \) do
\[ \col = \col + 1 \]
end while
\[ L_{d,e} = r + \col \]
if \( L_{d,e} = m \) then
if \( e < tk \) then
\[ tk = e; \text{solution}(0) = d + m; \]
\[ \num = 1; \]
else if \( e = tk \) then
\[ \text{solution}(
um) = d + m; \]
\[ \num = \num + 1; \]
end if
end if
end if
end for
Output \( \text{solution}(0), \ldots, \text{solution}(\num-1) \)

7. Experiments
Our machine is an AMD Opteron(tm) Processor 6128 (2.0G Hz with 8 cores) and 8GB in main memory. The operation system is FreeBSD 9.1-RELEASE. The compiler is GNU Compiler Collection (gcc) 4.2.1. All the compiled commands utilized the optimization level 3 argument (-O3).

7.1 Experiment 1: DNA Text and Small Distance Patterns
The text is the prefix with length 10 million of a DNA sequence Drosophila taken from NCBI. Four groups of patterns with lengths 30, 40, 50 and 60 were randomly selected from the text string and modified such that their errors are between 1 and 5. Each group contains 1000 patterns and \( SV \) was set to be \( m = \log_\sigma(m + \log_\sigma m) \). The results are shown in Figure 6. It shows that our algorithm is more efficient than the Sellers Algorithm.

7.2 Experiment 2: DNA Text and Random Patterns
The text string is the same with the text in experiment 1. Four groups of patterns with lengths 30, 40, 50 and 60 were randomly generated. Each group contains 1000 patterns and \( SV \) was set to be \( \log_\sigma(m + \log_\sigma m) \). The results are shown in Figure 7. Again, we can see that our algorithm works quite efficiently.

7.3 Experiment 3: Random Text and Small Distance Patterns with \( \sigma = 4 \)
The text was randomly generated with length 10 million. Four groups of patterns with lengths 30, 40, 50 and 60 were randomly selected from the text string and modified such that their errors are between 1 and 5. Each group contains 1000 patterns and \( SV = 7 \). The results are shown in Figure 8.

Figure 6. The Drosophila text and small distance patterns

Figure 7. The Drosophila text and random patterns

Figure 8. Random text and small distance patterns
7.4 Experiment 4: Random Text and Random Patterns with $\sigma = 4$

The text is the same as the text string in experiment 3. Four groups of patterns with lengths 30, 40, 50 and 60 were also randomly generated. Each group contains 1000 patterns and $SV$ was set to be $m \log \sigma (m + \log \sigma m)$. The results are shown in Figure 9.

![Figure 9. Random text and random patterns](image)

8. Conclusion and Future Research

In this paper, we are interested in solving the nearest neighbor search problem which is different from the approximate string matching problem. In the approximate string matching problem, there is a reasonable pre-specified error bound which does not exist in the nearest neighbor string searching problem. We first show the small change property in the $D$-matrix. This special property and computing the values on the diagonals of the $D$-matrix allows us to design an algorithm which may skip some computations. Experimental results show that our algorithm is much more efficient than the Sellers Algorithm.

For future research, let us note that there are many filtering algorithms to solve the approximate string matching problem [1], [3], [6], [9], [12], [15]. Most of them are based upon the pre-specified error bound $k$. We plan to use the counting filtering algorithm to solve our problem because it is not based upon the pre-specified error bound [15].

References