國立中山大學 資訊工程 學系

碩士論文

最長共同子序列與相關問題之回顧

A Survey of the Longest Common Subsequence Problem and Its Related Problems

研究生：陳建儐 撰

指導教授：楊昌彪

中華民國 九十四 年 六 月

A Survey of the Longest Common Subsequence Problem and Its Related Problems

A Thesis

Submitted to the Faculty

of

Department of Computer Science and Engineering

National Sun Yat-sen University

by

Jian-Bein Chen

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

June 29, 2005

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

ABSTRACT

The longest common subsequence (LCS) problem is a classical problem both in combinational optimization and computational biology. During the past few decades, a considerable number of studies have been focused on LCS and its related problems. In this thesis, we shall present a complete survey on LCS and its related problems, and review some efficient algorithms for solving these problems. We shall also give the definition to each problem, present some theorems, and illustrate time complexity and space complexity of the algorithms for solving these problems.

CHAPTER 1

Introduction

Since the LCS (*Longest Common Subsequence*) problem has been proposed in 1974 by Wagner and Fischer [129], much ink has been spent on this subject [8,13,21, 22,22,25,33,38,51,72,91,93,94,97,138]. Until now, LCS is still a hot research topic in many specialized fields, including computer science, molecular biology, mathematics, speech recognition, gas chromatography, bird song analysis and so on. In different areas, LCS will have some differences in problem definition. The purpose of this thesis is to list those problems which are transferred from LCS, and to enumerate some well-known algorithms in each problem.

The purpose of LCS algorithms is to find all matching points and dominate points quickly and efficiency. The LCS can be divided into two categories. One is processing the scoring matrix row by row and the other is contour to contour [61,65].

In different applications, there are many LCS related problem, such as the LIS (*longest increasing subsequence*) problems [23, 47, 137], the *cyclic string correction problem* [26, 49, 73, 86, 100, 111, 122], *the longest arc-preserving common subsequence problem* [67, 68, 81, 82] and *the constrained longest common subsequence problem* [30,104,123]. In these LCS related problems, with some LCS properties, we can find an efficient way to solve these problems.

The edit distance is widely used in many areas, such as computational biology, speech recognition and text retrieval [7, 40, 54, 92, 96, 109, 111, 127]. The *Levenshtein edit distance* has three different edit operations, insertion, deletion and replacement [78]. The LCS is a special case of edit distance with two operations only, which are insertion and deletion. Based on this idea, some other edit operations have also been defined, such as transpose, reversal, move operation and so on [36,70,90].

Sequence alignment is an important problem in computer science, and it can be divided into *pairwise alignment* and *multiple sequences alignment* [25,39,46,74,75, 106,131,134]. When we construct an alignment, there are two different strategies. One is global alignment [6,13,14,135], which focuses on the alignment with full strings. The other is local alignment [34,87,98,120,121,126], which emphasizes the substring of the input strings. In biology, the mutations usually occur in the successive region, not random in the genome. Therefore, we have the alignment problem with affine gap. *Multiple sequence alignment* (MSA) algorithms can be classified into exact algorithm, progressive algorithm and iterative algorithm. For speeding up, many algorithms for solving the multiple sequence alignment are based on pairwise alignment [24,29,48,139].

The rest of this thesis is organized as follows. In Chapter 2, we introduce some well-known dynamic programming algorithms, which solve the LCS problem efficiently. We also give two different approaches for the LCS problem. One is Hunt and Szymanki's algorithm [65] in Section 2.2, and the other is Hirschberg's algorithm [61] in Section 2.3. We discuss some other algorithms about LCS, and give the upper bound for the time complexity of LCS [11,52,53,60,85,108,128,132] in Section 2.4. We give some linear algorithm in Section 2.5. In Section 3.1, we introduce the cyclic string correction problem, and reduce the problem into cyclic longest common subsequence. We give some LIS algorithms in Section 3.2, we introduce the constrained longest common subsequence problem and tow algorithms for it [30,104,123]. In Chapter 4, we give some prosperities in the Levenshtein edit distance problem and list some complexities of the edit distance problem with various edit operations. In the Chapter 4.2, we give a brief introduction in sequence alignment. We introduce the global alignment and local alignment in Sections 5.1 and 5.2, respectively. The pairwise alignment with affine gap is introduced in Section 5.3. We give the three kinds of multiple sequence alignment algorithms and some biology programs in Section 5.4. In Chapter 6, we list some open LCS problems and its related problems, and also give a conclusion of this thesis.

CHAPTER 2

The Longest Common Subsequence Problem

The LCS problem is defined by Wagner and Fischer in 1974 [129]. Since then, many algorithms have been proposed [8,13,21,22,22,25,33,38,51,64,72,91,93,94,97, 136,138]. These algorithms should be categorized into the scoring matrix processing with row by row way and contour to contour way. We will introduce these algorithms one by one, according to its category.

2.1 The Longest Common Subsequence Problem

Let A and B be two strings, where $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$, $n \geq m$. A *subsequence* is obtained by deleting zero or some (not necessarily consecutive) characters of the original string. A *common subsequence* of A and B is a subsequence of both A and B. A longest common subsequence (LCS) of A and B is a common subsequence with maximum length. Generally speaking, the LCS may not be unique. For example, in Figure 2.1, we can see that `AGT` and `ACGAG` are both common subsequences in `AGCTGACG` and `CACAGTAG`. In addition, `ACGAG` is longer than `AGT`, and no other common sequence in `AGCTGACG` and `CACAGTAG` is longer than `ACGAG`. Therefore, `ACGAG` is the longest common subsequence between `AGCTGACG` and `CACAGTAG`.

The LCS problem can be defined as follows:

Definition : Given two strings A and B, where $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$, $n \geq m$, the LCS problem is to defined to find the LCS of A and B.

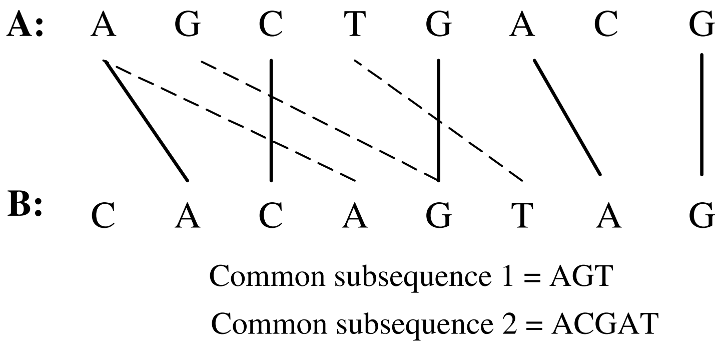


Figure 2.1 LCS subsequences.

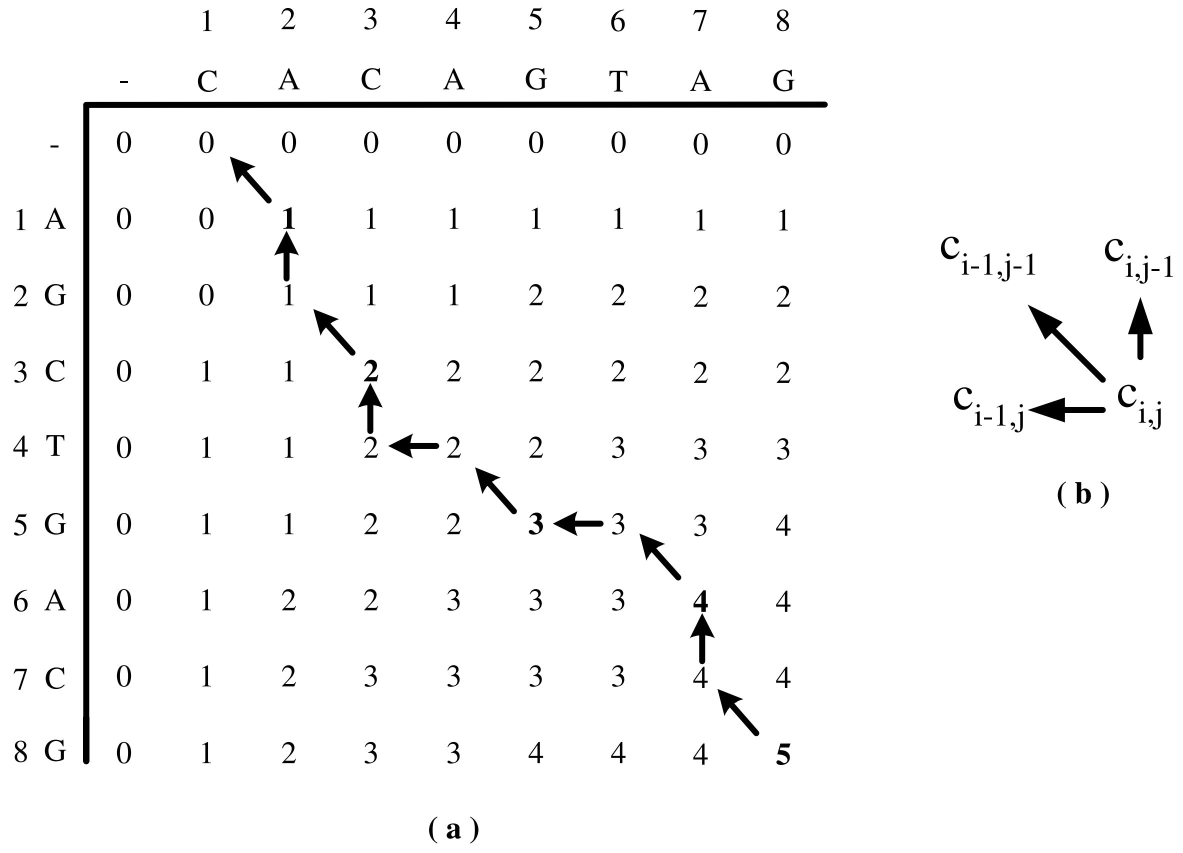
The first kind of the LCS algorithms are to build the scoring matrix row by row, usually using dynamic programming. In 1970, Needleman and Wunsch [98] proposed a well-known method, which uses dynamic programming to calculate an $(m+1) \times (n+1)$ matrix. Let $A_{1,i}$ denote $a_1a_2 \cdots a_i$, $1 \leq i \leq n$, and $B_{1,j}$ denote $b_1b_2 \cdots b_j$, $1 \leq j \leq m$. Let $L(i,j)$ denote the value of the LCS of $A_{1,i}$ and $B_{1,j}$. Each entry $L(i,j)$ will have three cases as follows:

Case 1: Delete a_i. The LCS between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$ is the same as the LCS between $a_1a_2 \ldots a_{i-1}$ and $b_1b_2 \ldots b_j$.

Case 2: Delete b_j. The LCS between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$ is the same as the LCS between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_{j-1}$.

Case 3: Keep the same symbol a_i and b_j. The LCS between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$ is the same as the LCS between $a_1a_2 \ldots a_{i-1}$ and $b_1b_2 \ldots b_{j-1}$ plus the matching score 1.

Therefore, $L(i,j)$ will reduce to the one of the three cases $L(i-1,j)$, $L(i,j-1)$ and $L(i-1,j-1)+1$, and each case can also follow the same conditions to reduce into the three cases with smaller size. Finally, the problem can reduce the initial state



		1	2	3	4	5	6	7	8
	-	C	A	C	A	G	T	A	G
-	0	0	0	0	0	0	0	0	0
1 A	0	0	**1**	1	1	1	1	1	1
2 G	0	0	1	1	1	2	2	2	2
3 C	0	1	1	**2**	2	2	2	2	2
4 T	0	1	1	2	2	2	3	3	3
5 G	0	1	1	2	2	**3**	3	3	4
6 A	0	1	2	2	3	3	3	**4**	4
7 C	0	1	2	3	3	3	3	4	4
8 G	0	1	2	3	3	4	4	4	**5**

Figure 2.2 An example of LCS of two strings. (A) Each entry $L(i,j) = k$ means there is an LCS of length k along the path from (i,j) to $(0,0)$. (B) Three possible directions of the tracking back path.

$L(i,0)$, $L(0,j)$ or $L(0,0)$, whose scores are all 0. We can calculate $L(i,j)$ formally as follows:

$$L(i,0) = 0 \ , \ i \geq 0$$

$$L(0,j) = 0 \ , \ j \geq 0$$

$$L(i,j) = \max \left\{ \begin{array}{l} L(i-1,j) \\ L(i,j-1) \\ L(i-1,j-1)+1 \quad if \ a_i = b_j \end{array} \right\}, \ 1 \leq i \leq n, 1 \leq j \leq m.$$

The advantage of this algorithm is easy to implement and to trace back. It is clear that the time complexity is $O(mn)$. In Figure 2.2, we use the above scoring function to fill the matrix row by row, and use the matching pairs to trace back the LCS. In the figure, we can note that the score can be increased only by a match. Based on this idea, there are two different approaches to find the LCS. First, Hunt and Szymanki's [65] algorithm uses dynamic programming to find those match positions. The second algorithm was proposed by Hirschberg [61], which uses the divide and conquer strategy to find the match positions. In the following two sections, we will introduce these two algorithms in detail.

2.2 Hunt and Szymanki's Algorithm

Hunt and Szymanki [65] performed preprocessing on the input strings, and found out all matches in the input strings first. With the information obtained from the preprocessing, we can compute the matching positions more quickly. The algorithm can be divided in three steps and the time complexity is $(O(r+n)\log n)$, where n is the length of two input strings and r is the total number of matches between the two strings. Let $LCS(A,B)$ denote the length of LCS of strings A and B. In this algorithm, there is a key data structure, the "threshold values" array $T_{i,k}$, where $T_{i,k}$ is the least j value such that $A_{1,i}$ and $B_{1,j}$ have a common subsequence of length k. For example in Figure 2.3, $T[4,3]=6$ means that $LCS(A_{1,4},B_{1,6})$= 3, but $LCS(A_{1,4},B_{1,5}) \leq 2$. In this example $A_{1,4}=$ `AGCT` and $B_{1,6}=$ `CACAGT`, and their LCS length is 3. It can be easily seen that the LCS length of $A_{1,4}$ and $B_{1,5}=$ `CACAG` is only 2.

Hunt and Szymanki's Algorithm

Input: Two strings $A=a_1a_2\cdots a_n$ and $B=b_1b_2\cdots b_n$.

Output: The longest common subsequence of A and B.

```
    element array $A[1:n], B[1:n]$;
    integer array $THRESH[0:n]$;
    list array $MATCHLIST[1:n]$
    pointer array $LINK[1:n]$
    pointer PTR;

1.  for $i = 1$ step 1 until $n$ do
          set $MATCHLIST[i] =< j_1, j_2, \cdots, j_p >$ such that
              $j_1 > j_2 > \cdots > j_p$ and $A[i] = B[j_q]$ for $1 \leq q \leq p$;
2.  $THRESH[0] = 0$;
    for $i = 1$ step 1 until $n$ do
          $THRESH[i] = n + 1$;
    $LINK[0] = null$
3.  for $i = 1$ step 1 until $n$ do
          for $j$ on $MATCHLIST[i]$ do
                begin
                      find $k$ such that $THRESH[k-1] < j \leq THRESH[k]$;
                        if $j < THRESH[k]$ then
                              begin
                                  $THRESH[k] = j$;
                                  $LINK[k] = newnode(i, j, LINK[k-1])$;
                                end;
                end;
```

```
4.  $k$ = largest $k$ such that $THRESH[k] \neq n+1$;
    $PTR = LINK[k]$;
    while $PTR \neq null$ do
        begin
            print $(i,j)$ pair pointerd to by PTR
            advance PTR
        end
```

Here we denote (i,j) as a general pair, and $<i,j>$ as a matching pair where $a_i = b_j$. The set of j-values is $\{j \mid < a_i, b_j >\}$.

In the first step, we record the j-value where a_i matches in B with decreasing order. For example in Figure 2.3, we denote string B = `CACAGTAG` as a sequence $S_B = C1, A2, C3, A4, G5, T6, A7, G8$ with index j. The j-value of A and G in B are $\{2,4,7\}$ and $\{5,8\}$, respectively. Therefore, we sort S_B by lexical order and return the sequence $S_B^{'} = A2, A4, A7, C1, C3, G5, G8, T6$ in $O(n \log n)$ time. We can search any symbol in the sorted list, and this process will take $O(\log n)$ time in binary search. In the worst case, we have to search n distinct symbols in the search process and we need $O(n \log n)$ time. Hence, we sort and search the input string in Step 1, which requires $O(n \log n)$ time and $O(n)$ space. Step 2 initiates the $THRESH$ array with $O(n)$ time, and fills all entries $(n+1)$ as infinite. We show an example in Figure 2.3 to illustrate these steps, the character G appears in column 5 and column 8 for row 2. Hence, $MATCHLIST[2]$ is given by $[8,5]$ to record that $<2,5>$ and $<2,8>$ are matching pairs.

In Step 3, there are two outer for-loops to do all $<i,j>$ with decreasing j with increasing i. Hence, we will execute r loops. In each loop, we add the matching list in our efficient data structure "THRESH" row by row. When we want to add a $<i,j>$ into the $THRESH[i]$ array and to replace $THRESH[i][k]$, it means for all

		0	1	2	3	4	5	6	7	8
			C	A	C	A	G	T	A	G
0		0	0	0	0	0	0	0	0	0
1	A	0	0	1	1	1	1	1	1	1
2	G	0	0	1	1	1	2	2	2	2
3	C	0	1	1	2	2	2	2	2	2
4	T	0	1	1	2	2	2	3	3	3
5	G	0	1	1	2	2	3	3	3	4
6	A	0	1	2	2	3	3	3	4	4
7	C	0	1	2	3	3	3	3	4	4
8	G	0	1	2	3	3	4	4	4	5

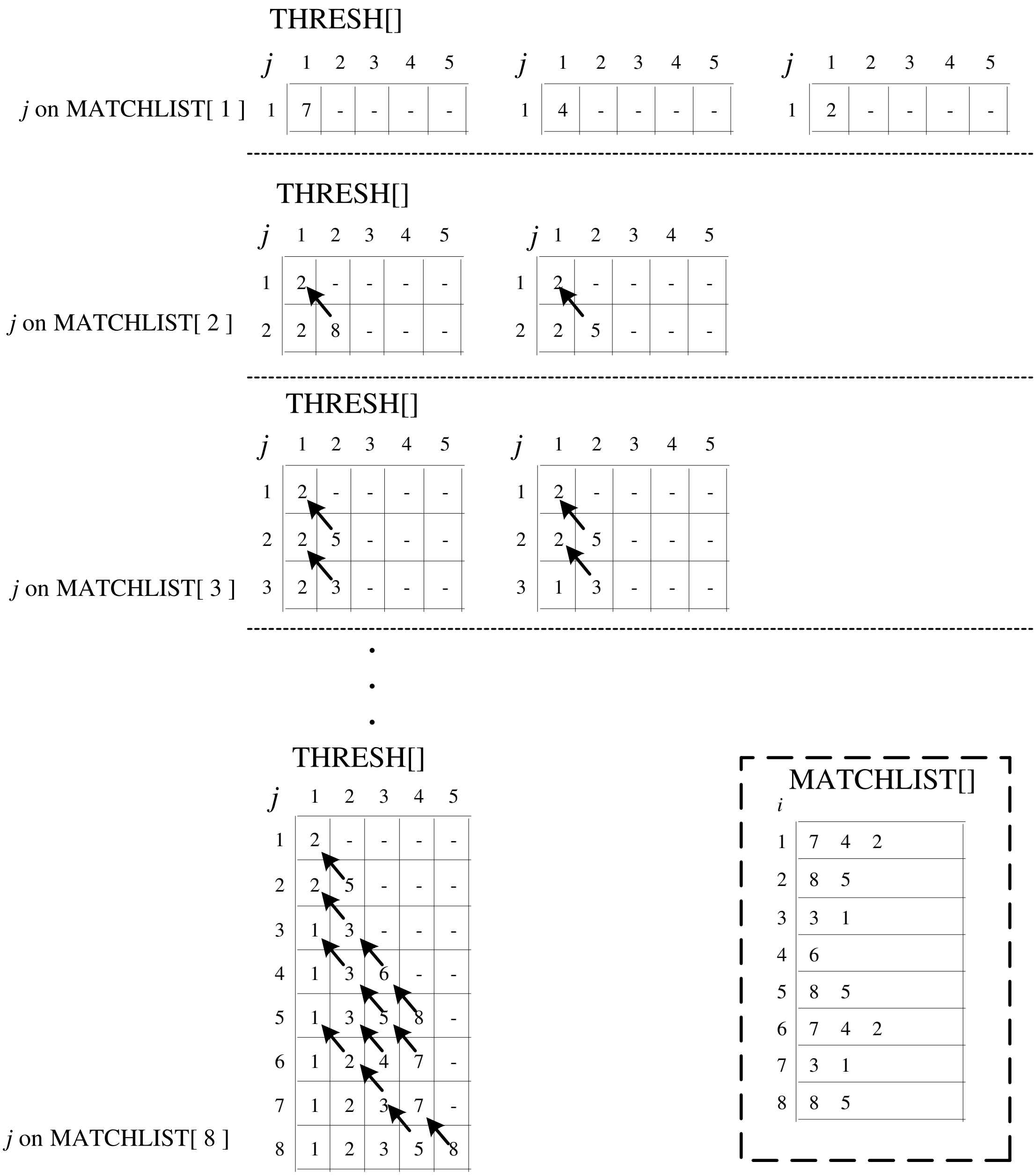
MATCHLIST

i	
1	7 4 2
2	8 5
3	3 1
4	6
5	8 5
6	7 4 2
7	3 1
8	8 5

Figure 2.3 Steps 1 and 2 of Hunt and Szymanski's algorithm.

common subsequences of length k, j is the leftmost one. For example in Figure 2.4, when j is on $MATCHLIST[2]$, we first add element 8 in $THRESH[2][2]$, because $2 \leq 8 \leq infinite$, and add a linked list to $THRESH[1][1]$, it means now we have a common sequence whose length is 1 in score matrix $(2, 1)$, and also have a common subsequence whose length is 2 in the score matrix $(2, 8) \longrightarrow (1, 2)$. Then we add the second element 5 on $MATCHLIST[2]$, $2 \leq 5 \leq 8$, so 5 will replace 8, it means we find a better solution. In step 3, we consider all matches in the scoring matrix so it requires $O(n + r \log n)$ time and at most $O(r)$ list nodes.

In the final step, we follow the maximum column in $THRESH[]$, and trace the LCS according to the linked list, it takes at most $O(n)$ time. In 1987, Apostolico and Guerra [12] used the idea of dominating match to reduce the number of matches which need to be considered.



THRESH[]

j on MATCHLIST[1]

j	1	2	3	4	5
1	7	-	-	-	-

j	1	2	3	4	5
1	4	-	-	-	-

j	1	2	3	4	5
1	2	-	-	-	-

THRESH[]

j on MATCHLIST[2]

j	1	2	3	4	5
1	2	-	-	-	-
2	2	8	-	-	-

j	1	2	3	4	5
1	2	-	-	-	-
2	2	5	-	-	-

THRESH[]

j on MATCHLIST[3]

j	1	2	3	4	5
1	2	-	-	-	-
2	2	5	-	-	-
3	2	3	-	-	-

j	1	2	3	4	5
1	2	-	-	-	-
2	2	5	-	-	-
3	1	3	-	-	-

.
.
.

THRESH[]

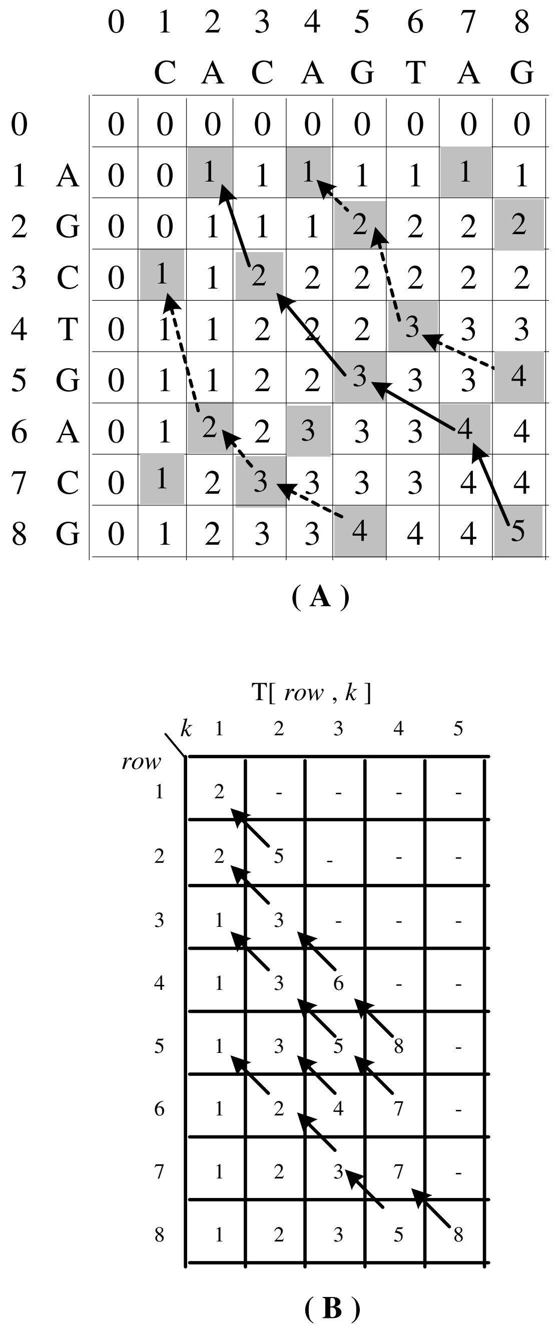
j on MATCHLIST[8]

j	1	2	3	4	5
1	2	-	-	-	-
2	2	5	-	-	-
3	1	3	-	-	-
4	1	3	6	-	-
5	1	3	5	8	-
6	1	2	4	7	-
7	1	2	3	7	-
8	1	2	3	5	8

MATCHLIST[]

i	
1	7 4 2
2	8 5
3	3 1
4	6
5	8 5
6	7 4 2
7	3 1
8	8 5

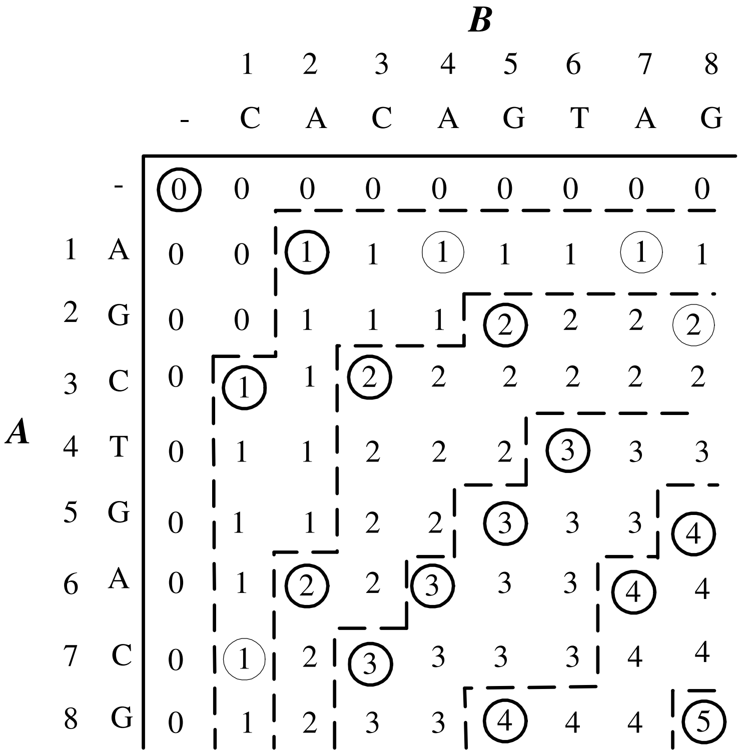
Figure 2.4 Step 3 of Hunt and Szymanski's algorithm.



2.3 Hirschberg's Algorithm

The second kind of approaches is the contour-based approach, which was proposed by Hirschberg in 1977 [61]. Before explaining the algorithm, we shall first give some definitions and properties of the contour. Here we denote $< i, j >$ as the position where $a_i = b_j$, and $L(i, j)$ as the length of the LCS in $A_{1,i}$ and $B_{1,j}$. We call $< i, j >$ a k-candidate, if $a_i = b_j$ and $L(i, j) = k$. According to Theorem 2.1, among all k-candidates, we need only consider these k-candidates which dominate other k-candidates and are not dominated by any other k-candidates. These considered k-candidates are called *minimal k-candidates*. In Figure 2.6, all circled positions are k-candidates, and bold ones are minimal k-candidates. For example, $< 1, 2 >$, $< 1, 4 >$, $< 1, 7 >$, $< 3, 1 >$ and $< 7, 1 >$ are all 1-candidates between `AGCTGACG` and `CACAGTAG`. It is clear that $< 1, 2 >$ and $< 3, 1 >$ are both minimal 1-candidates, because they can dominate all other 1-candidates except for each other. Therefore, all 2-candidates can be obtained by $< 1, 3 >$ and $< 3, 1 >$. The matrix separated by dash lines is called "*contour*", which has its own k-candidate region and can never be crossed to each other. Besides, minimal k-candidates must exist in the corner of the contour, because this is the position which can never be dominated. Hence, one can transfer the LCS problem to finding the set of minimal k-candidates. If we find out all minimal k-candidates and there is no $(k + 1)$-candidate, then the length of the LCS is k, and the LCS path must exist in the set.

Theorem 2.1 *[61] Let $A =< x_1, y_1 >$ and $B =< x_2, y_2 >$ be two k-candidates. If $x_1 \geq y_1$ and $x_2 \geq y_2$, then we can say that A dominate B (B is a superfluous k-candidate) since any $(k + 1)$-candidates generated by B can also be generate by A. Thus, from the set of k-candidates, we need only consider those candidates are minimal under the usual vector ordering. Note that if A and B are both minimal elements, then $x_1 < x_2$ iff $y_2 > y_1$.*



			1	2	3	4	5	6	7	8
		-	C	A	C	A	G	T	A	G
	-	⓪	0	0	0	0	0	0	0	0
1	A	0	0	①	1	①	1	1	①	1
2	G	0	0	1	1	1	②	2	2	②
3	C	0	①	1	②	2	2	2	2	2
4	T	0	1	1	2	2	2	③	3	3
5	G	0	1	1	2	2	③	3	3	④
6	A	0	1	②	2	③	3	3	④	4
7	C	0	①	2	③	3	3	3	4	4
8	G	0	1	2	3	3	④	4	4	⑤

Figure 2.6 Contours in the scoring matrix.

Theorem 2.2 *[61] For $k \geq 1$, $< i, j >$ is a minimal k-candidate iff j is the minimum value such $b_j = a_i$, and $low < j < high$, where high is the minimum j-value of all k-candidates whose i-value is less than i (no upper limit if there is no such k-candidate) and low is the minimum j-value of all $(k-1)$-candidates whose i-value is less than i.*

In this algorithm, $\mid \Sigma \mid = s$, where s is the numbers of distinct symbols in B and each symbol in Σ is encoded from $1, 2, \cdots, s$. Here we denote $NB[\theta]$ as the number of times that symbol θ occurs in string B and $PB[\theta, 1], \cdots, PB[\theta, NB[\theta]]$ as the order list of string B in which symbol θ occurs. For example, in Figure 2.8, there are 4 distinct symbols in string B. Hence, A is encoded to 1, G is encoded to 2, C is encoded to 3, and T is encoded to 4. The symbol G is the second row in the PB array. Therefore, $PB[2, 1] = 5$ and $PB[2, 2] = 8$ means that in string B, the symbol G first occurs in the fifth position, and the second one occurs in the eighth position.

The Hirschberg Algorithm

Input: Two strings $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$.

Output: The longest common subsequence of A and B.

```
    ALGD(m,n,A,B,C,p)
1.  $NB[\theta] = 0$ for $\theta = 1, \ldots, s$
    $PB[\theta,\ 0] = 0$ for $\theta = 1, \ldots, s$
    $PB[0,\ 0] = 0$; $PB[0,\ 1] = 0$
    for $j = 1$ step until $n$ do
    begin
        $NB[b_j] = NB[b_j] + 1$
        $PB[b_j,\ NB[b_j]] = j$
    end
2.  $D[0,\ i] = 0$ for i= 0, …, $m$
    lowcheck = 1
3.  for $k = 1$ step 1 do
    begin
4.      $N[\theta] = NB[\theta]$ for $\theta$ 1, …, s
        $N[0] = 1$
        $FLAG = 0$
        $low = D[k-1,\ lowcheck]$
        $high = n + 1$
```

```
5.          for $i = lowcheck + 1$ step 1 until $m$ do
            begin
6.              while $PB[a_i, N[a_i] - 1] >$ low do $N[a_i] = N[a_i] - 1$
7.              if high $> PB[a_i, N[a_i]] > low$
                    then begin
                        $high = PB[a_i, N[a_i]]$
                        $D[k, i] = high$
                        if $FLAG = 0$ then $\{lowcheck = i,\ FLAG = 1\}$
                    end
                    else $D[k, i]$
8.              if $D[k-1, i] > 0$ then $low = D[k-1, i]$
            end loop of step 5
9.      if $FLAG = 0$ then go to Step 10
        end loop of step 3
10.     $p = k - 1$
        $k = p$
        for $i = m + 1$ step -1 until 0 do
        if $D[k, i] > 0$ then
        begin
            $c_k = a_i$
            $k = k - 1$
        end
```

In Step 1, we record the number of occurrences of symbol θ in B and sort them by their j-values in the PB matrix. This will take $O(n \log s)$ time, where s is the number of distinct symbols in B. Through this step, we can quickly bound the region, from which we can find the next minimal k-candidates.

In Step 2, we use Theorem 2.2 to make a possible region to find out all k-candidates using previous $(k-1)$-candidates. For example, Figure 2.7(a) shows the initial state with $k = 2$, where the gray area is the possible area for the minimal 2-candidates. In Figure 2.7(b), a new bound of gray area is set after we find the first minimal 2-candidate. Therefore, each time we find a new minimal k-candidate, the search area for other k-candidates will become smaller.

In Figure 2.9 and Figure 2.10, we show how each new minimal k-candidate is found. In Step 2, we will do p loops to find out all k-candidates. In each loop, we use PB to quickly locate the specific symbol which has the closest j-value to the lower bound to decrease the region more efficiently. Since it takes $O(n)$ time for a single loop, Step 2 can be done in $O(pn)$ time. After we obtain each minimal k-candidate, the k-candidate will be added to another matrix $D[k, i] = j$, which means the position of common subsequence with length k in row i is j. Finally, we can use matrix D to trace the LCS path, which will take $O(m)$ time. These three steps take $O(n \log s + pn + m)$, therefore the time complexity is $O(n \log s + pn)$, where s is the number of distinct symbols, and p is the length of LCS.

2.4 Some Algorithms of LCS

In recent years, there has been renewal of interest in the lower bound for the complexity of the LCS problem [3, 47, 60]. In 1980, Masek and Paterson [66] used the "Four Russians" technique to prove that if $\mid \Sigma \mid$ is arbitrary or finite, the upper bound of the time complexity for LCS is $O(mn \log \log n / \log n)$ and $O(mn/ \log n)$, respectively. Until now, this still theoretically fastest algorithm for the LCS problem.

Using dynamic programming, the LCS problem can be solved easily in $O(mn)$ time and space, where m and n are the lengths of LCS input strings. Many heuristics algorithms try to reduce the complexity with special inputs, such as matching number, shorter LCS length, small alphabet and so on [9–13, 27, 28, 44, 61, 65, 91, 94, 134]. For example, Hunt and Szymanski's algorithm takes $O(r \log n)$ time and Hirschberg's algorithm takes $O(pn)$ time where r is the number of matching pairs and p is the length of LCS. When r and p are small, these two algorithm are faster than the traditional algorithm. However, in the worst case, r will be n^2 and p will be n, then these two algorithm become worse than the traditionl algorithm.

In recent years, many algorithms try to use bit-vector to speed up LCS, which are word-dependent. These algorithms assume the computer can process one word at a time, where the length of the word is k (normally 32 or 64, depending on the machine). Hence, it only takes $O(1)$ to process a word whose length is less than or equal to k. The advantage of bit-vector algorithm is avoiding input-sensitive and output sensitive. According to this hypothesis, many algorithms were proposed with bit-vector, and the LCS problem can be solved in $O(nm/k)$ time [16, 93]. In 2000, Crochemore et al. gave an $O(n^2/k)$ time and $O(n/k)$ space algorithm [38].

In Summary, we list these algorithms in Table 2.1.

2.5 Linear Space Algorithms

When computer scientists analyze an algorithm, they usually focus on its time complexity and space complexity . Computing the LCS with the Wagner and Fischer's algorithm typically requires $O(m \times n)$ space, where m and n are the lengths of the input strings. This space complexity will become infeasible if n and m are very large. In this section, we will present linear space algorithms for the LCS problem.

Hirschberg proposed a linear space version for the LCS problem [59]. The main idea in this algorithm is to find a break point of the LCS and use divide and conquer method to reconstruct the LCS path. Given two strings $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$, where m and n are the lengths of A and B, respectively. $ALGB(m, n, L)$ is a linear space algorithm and it returns L which is the last row of the original LCS $m \times n$ matrix. The algorithm is as follows [59]:

Year	Author(s)	Time Complexity	Space Complexity
1974	Wagner and Fischer [129]	$O(mn)$	$O(mn)$
1975	Hirschberg [59]	$O(mn)$	$O(n)$
1977	Hunt and Szymanski [65]	$O((n+r)\log n)$	$O(r+n)$
1977	Hirschberg [61]	$O(pn+n\log n)$	$O(pn)$
1977	Hirschberg [61]	$O(L(m-p)\log n)$	$O((m-p)^2+n)$
1980	Masek and Paterson [66]	$O(n\max\{1,m/\log n\})$	$O(n^2/\log n)$
1982	Nakatsu et al. [94]	$O(n(m-p))$	$O(m^2)$
1984	Hsu and Du [10,62]	$O(pm\log(n/p)+pm)$	$O(pm)$
1985	Ukkonen [125]	$O(Em)$	$O(E\min\{m,E\})$
1986	Apostolico [9]	$O(n+m\log n+D\log(mn/D))$	$O(r+m)$
1986	Myers [91]	$O(D(m+n))$	$O(D(m+n))$
1987	Kumar and Rangan [71]	$O(n(m-p))$	$O(n)$
1987	Apostolico and Guerra [12]	$O(pm+n)$	$O(D+n)$
1990	Chin and Poon [27]	$O(n+\min\{D,pm\})$	$O(D+n)$
1992	Apostolico [11]	$O(pm)$	$O(n)$
1992	Eppstein et al. [44]	$O(n+D\log\log\min\{D,mn/D\})$	$O(D+m)$
1999	Rick [108]	$O(\min\{pm,p(n-p)\})$	$O(n)$

Table 2.1 Time and space complexity of algorithms computing LCS(A,B), where $m=\mid A\mid$ and $n=\mid B\mid$, r is the number of matching pairs, and p is the length of a longest common subsequence, $E=m+n-2p=$ edit distance between A and B, D is the number of dominant matches, and k is the number of bits of the word. [102,103,108]

```
    $ALGC(m, n, A, B, C)$
1.  If problem is trivial, solve it:
    if $n = 0$ then $C = \phi$ ($\phi$ is the empty string)
    else if $m = 1$ then if $\exists\ j \leq n$ such that $A(1) = B(j)$
                    then $C = A(1)$
                    else $C = \phi$
2.  Otherwise, spilt problem:
    else begin $i = \lfloor m/2 \rfloor$;
3.  Evaluate $L(i, j)$ and $L * (i, j)$ $[j = 0, 1, \cdots, n]$:
        $ALGB(A_{1,i}, B_{1,n}, L_1)$;
        $ALGB(A_{m,i+1}, B_{n,1}, L_2)$;
4.  Find $j$ such that $L(i, j) + L * (i, j) = L(m, n)$ using problem:
        $M = \max\{L_1 j + L_2(n - j)\}$;
        $k = \min j \textit{ such taht } L1(j) + L2(n - j) = M$;
5.  Solve simpler problems:
        $ALGC(i, k, A_{1,i}, B_{1,k}, C1)$;
        $ALGC(m - i, n - k, A_{i+1,m}, B_{k+1,n}, C2)$;
6.  Give output:
        $C = C1 \mid\mid C2$;
    end
```

The main idea of this algorithm lies in Step 3, which uses $ALGB$ to find the break point. Step 3 can be described in more detail as follows:

1. Use $ALGB(A_{1,m/2}, B_{1,n})$ to calculate the last row L_1 of LCS matrix between $a_1 a_2 \cdots a_{m/2}$ and $b_1 b_2 \cdots b_n$.

2. Use $ALGB(A_{n,m/2+1}, B_{n,1})$ to calculate the last row L_2 of LCS matrix between $a_m a_{m-1} \cdots a_{m/2+1}$ and $b_n b_{n-1} \cdots b_1$.

3. Calculate $L(k) = L_1(k) + L_2(n-k)$ where $L_i(k)$ is the kth element of L_i and $k = 1, 2, \cdots, n$. Hence, the maximum $L(k)$ is the break point.

Take Figure 2.12 for example, Figure 2.12 (a) and (b) compute $L_1 =$ `11223333` and $L_2 =$ `33333221`, respectivly. Afterwards, we can compute the maximum of $L_1 + L_2$, which is 5. It means the LCS must pass through this region and the length of the LCS is 5. After we pick the break point we can cut the original problem to two smaller problems. In Figure 2.13 and Figure 2.14, we can find out the computing area is decreased by using the break point.

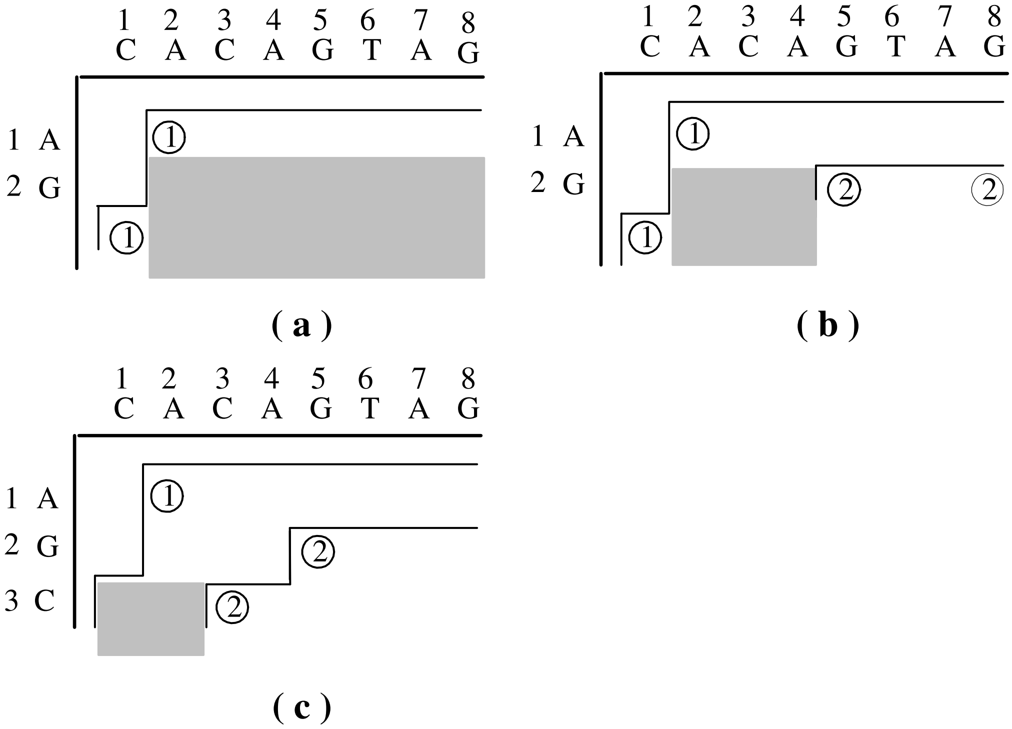
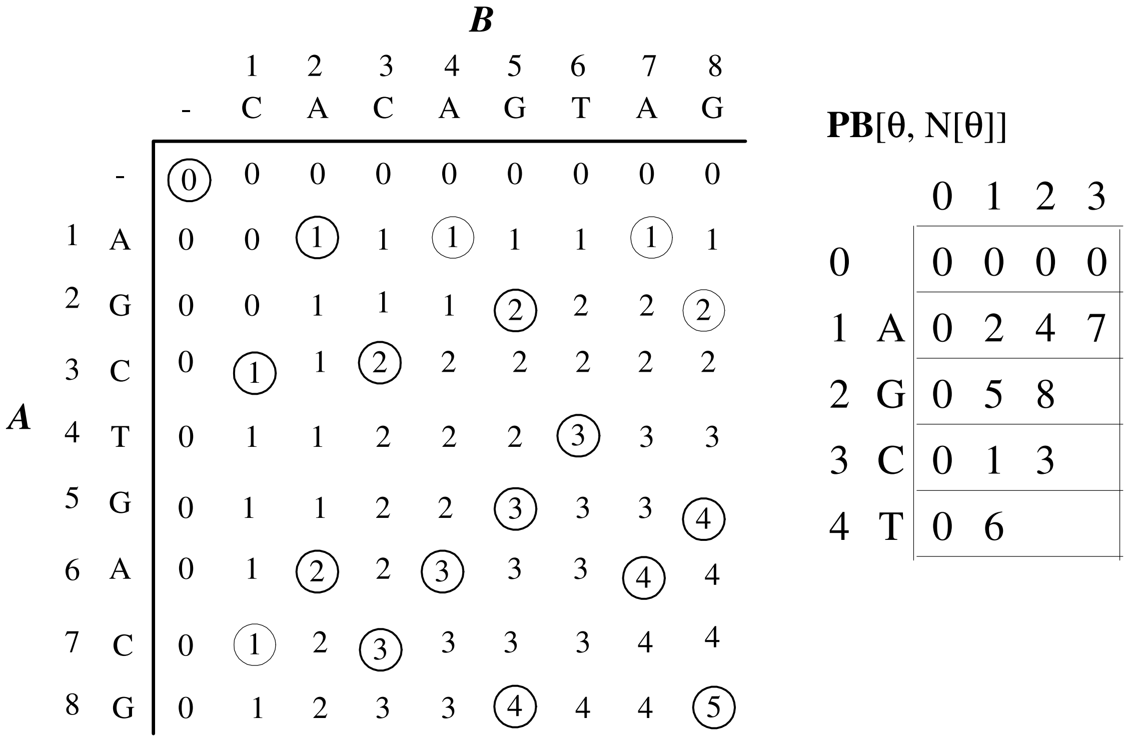


Figure 2.7 Three possible regions in contour.



		B								
			1	2	3	4	5	6	7	8
		-	C	A	C	A	G	T	A	G
	-	0	0	0	0	0	0	0	0	0
1	A	0	0	1	1	1	1	1	1	1
2	G	0	0	1	1	1	2	2	2	2
3	C	0	1	1	2	2	2	2	2	2
A 4	T	0	1	1	2	2	2	3	3	3
5	G	0	1	1	2	2	3	3	3	4
6	A	0	1	2	2	3	3	3	4	4
7	C	0	1	2	3	3	3	3	4	4
8	G	0	1	2	3	3	4	4	4	5

PB[θ, N[θ]]

		0	1	2	3
0		0	0	0	0
1	A	0	2	4	7
2	G	0	5	8	
3	C	0	1	3	
4	T	0	6		

Figure 2.8 Steps 1 and 2 of Hirschberg's algorithm. θ is the number of distinct symbols in B, $N[\theta]$ is the occurrences of θ in B.

PB[θ, N[θ]]

		0	1	2	3
0		0	0	0	0
1	A	0	2	4	7
2	G	0	5	8	
3	C	0	1	3	
4	T	0	6		

D[*k* , *i*]

k = 1

k \ *i*	1	2	3	4	5	6	7	8
1	-	-	-	-	-	-	-	-

i = 0,
high = 9, *low* = 0

k \ *i*	1	2	3	4	5	6	7	8
1	2	-	-	-	-	-	-	-

i = 1,
high = 2, *low* = 0,
lowcheck = 1

	1	2	3	4	5	6	7	8
1	2	0	1	-	-	-	-	-

i = 3,
high = 1, *low* = 0

• • •

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0

---

k = 2

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	-	-	-	-	-	-	-

high = 9, *low* = 2,
lowcheck = 1

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	-	-	-	-	-	-

i = 2,
high = 5, *low* = 2,
lowcheck = 2

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	3	-	-	-	-	-

i = 3
high = 3, *low* = 1

• • •

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	3	0	0	2	0	0

Figure 2.9 Steps 3 $\sim$ 9 of Hirschberg's algorithm when $k = 1$ and 2.

k = 3

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	3	0	0	2	0	0
3	-	-	0	-	-	-	-	-

high = 9, *low* = 5,
lowcheck = 2

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	3	0	0	2	0	0
3	-	-	0	-	-	-	-	-

high = 9, *low* = 3

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	3	0	0	2	0	0
3	-	-	0	6	-	-	-	-

high = 6, *low* = 3,
lowcheck = 4

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	3	0	0	2	0	0
3	-	-	0	6	5	-	-	-

high = 5, *low* = 3

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	3	0	0	2	0	0
3	-	-	0	6	5	4	-	-

high = 4, *low* = 3

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	3	0	0	2	0	0
3	-	-	0	6	5	4	3	-

high = 3, *low* = 3

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	3	0	0	2	0	0
3	-	-	0	6	5	4	3	0

•

•

•

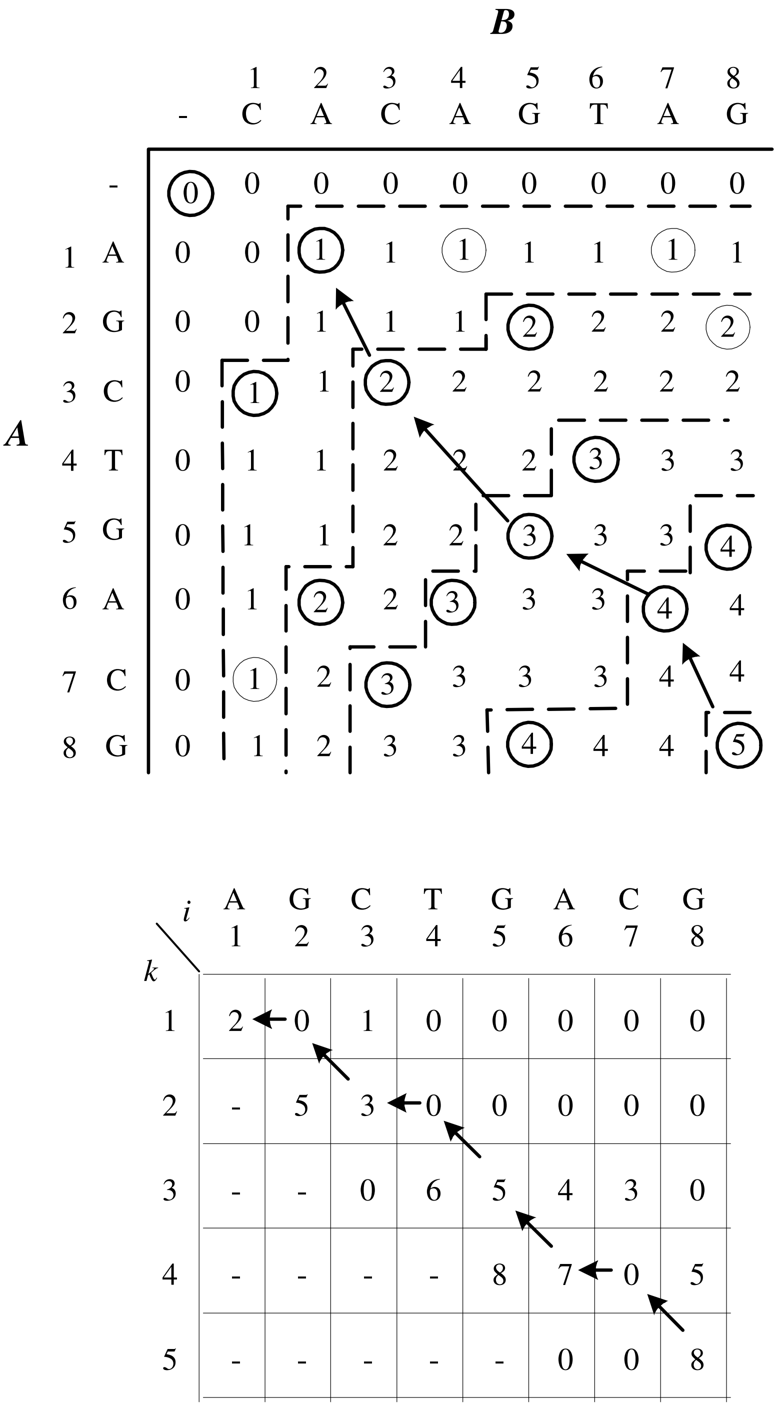
k = 5

	1	2	3	4	5	6	7	8
1	2	0	1	0	0	0	0	0
2	-	5	3	0	0	2	0	0
3	-	-	0	6	5	4	3	0
4	-	-	-	-	8	7	0	4
5	-	-	-	-	-	0	0	8

PB[θ, N[θ]]

		0	1	2	3
0		0	0	0	0
1	A	0	2	4	7
2	G	0	5	8	
3	C	0	1	3	
4	T	0	6		

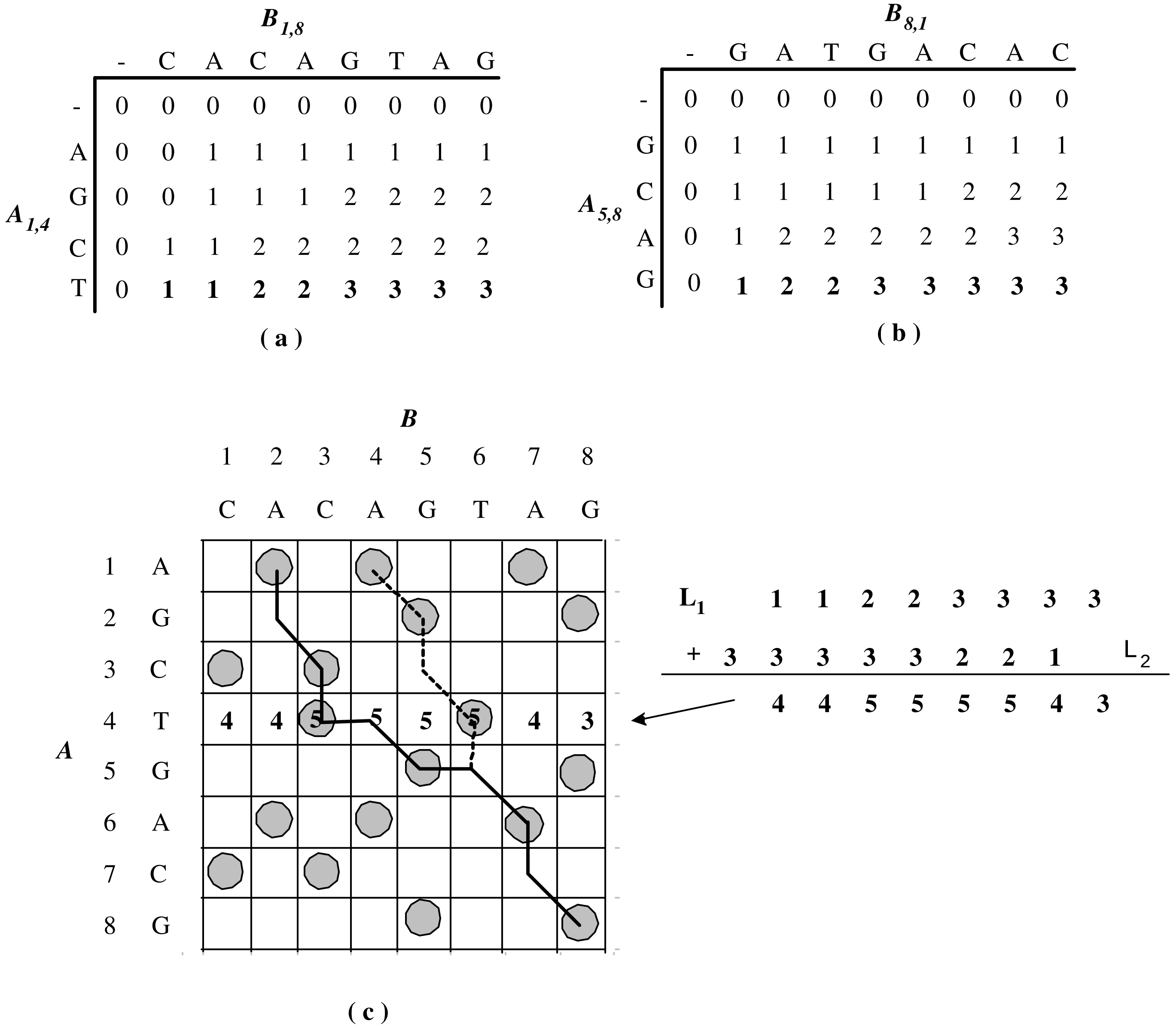
Figure 2.10 Steps 3 ∼ 9 of Hirschberg's algorithm when $k = 3$.



			1	2	3	4	5	6	7	8
		-	C	A	C	A	G	T	A	G
	-	0	0	0	0	0	0	0	0	0
1	A	0	0	1	1	1	1	1	1	1
2	G	0	0	1	1	1	2	2	2	2
3	C	0	1	1	2	2	2	2	2	2
4	T	0	1	1	2	2	2	3	3	3
5	G	0	1	1	2	2	3	3	3	4
6	A	0	1	2	2	3	3	3	4	4
7	C	0	1	2	3	3	3	3	4	4
8	G	0	1	2	3	3	4	4	4	5

k \ i	A 1	G 2	C 3	T 4	G 5	A 6	C 7	G 8
1	2	0	1	0	0	0	0	0
2	-	5	3	0	0	0	0	0
3	-	-	0	6	5	4	3	0
4	-	-	-	-	8	7	0	5
5	-	-	-	-	-	0	0	8

Figure 2.11 Step 10 of Hirschberg's algorithm.



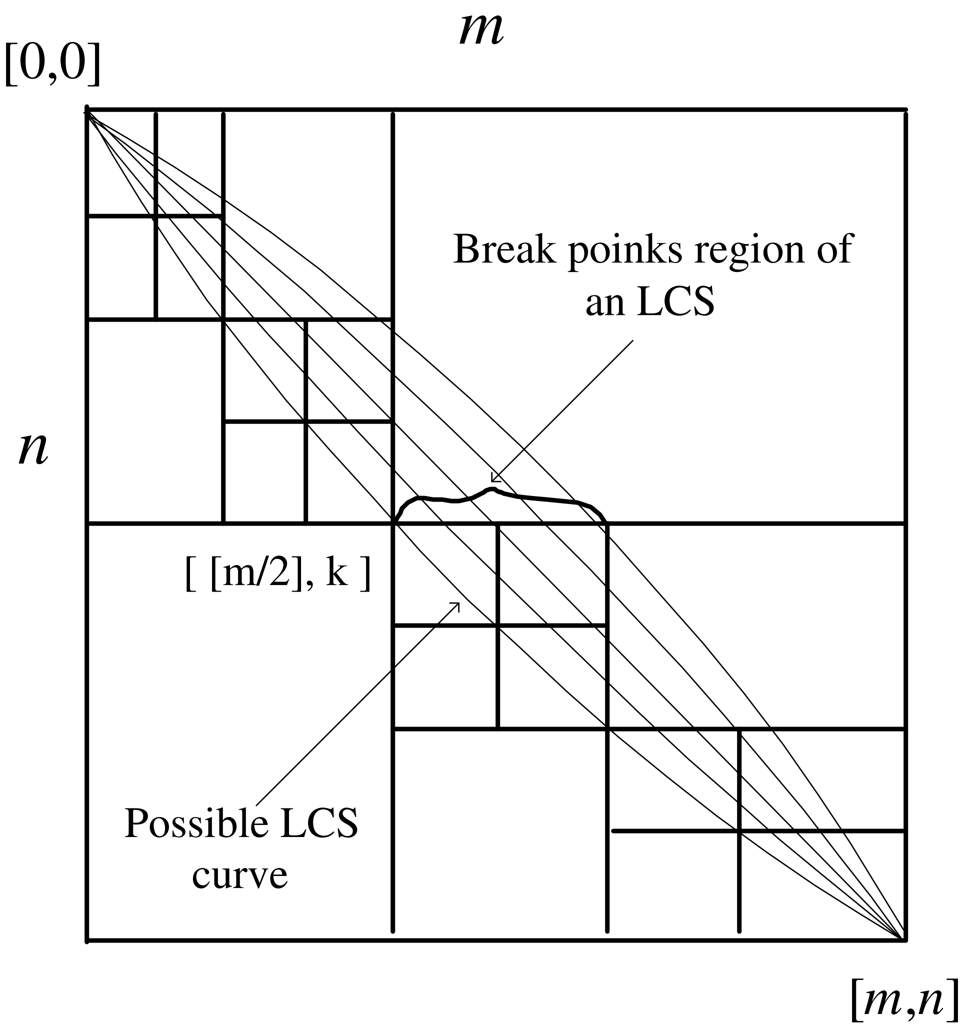


Figure 2.13 The divide and conquer of the LCS algorithm. [37]

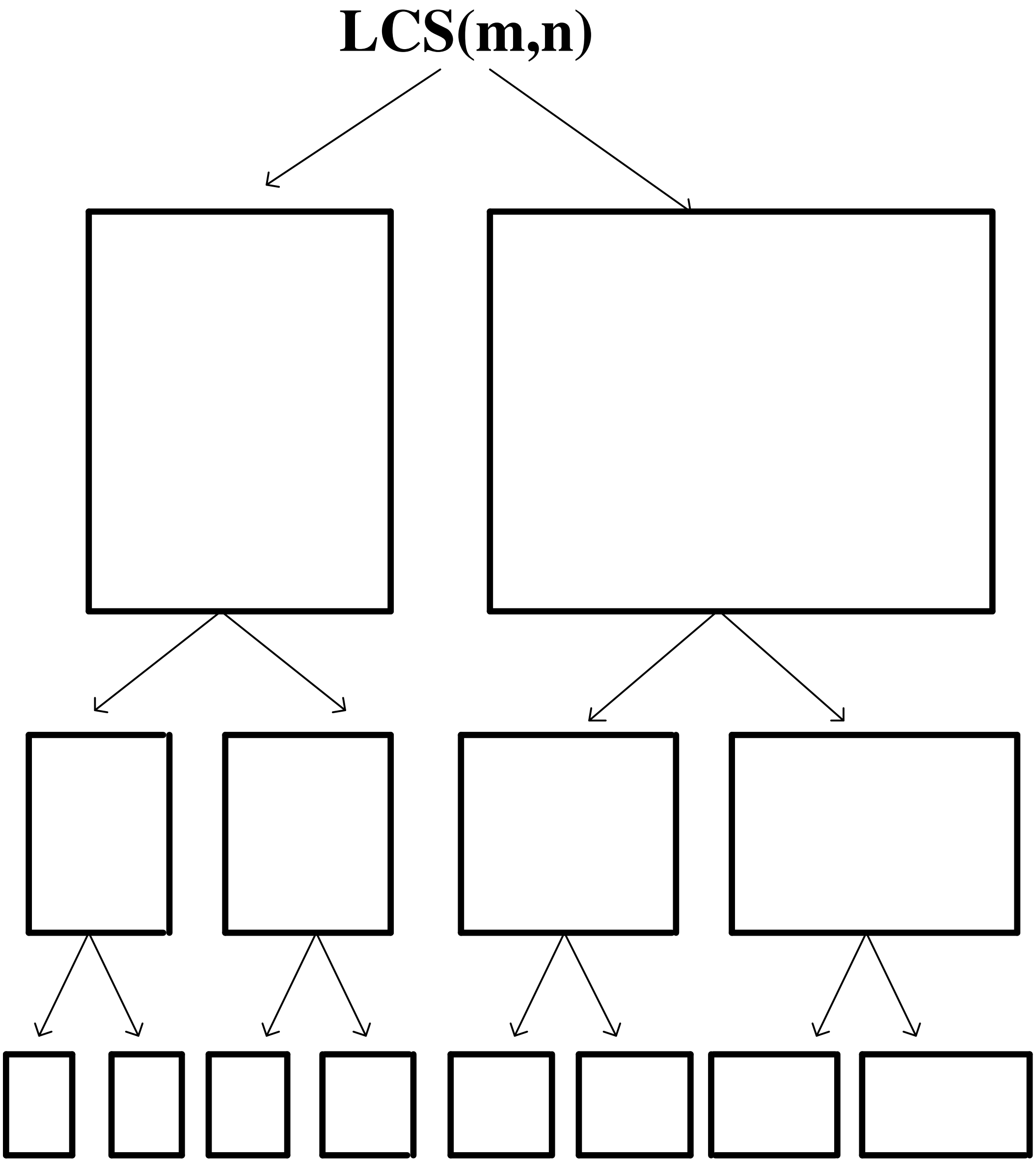


Figure 2.14 The recursive calls of the LCS algorithm. [37]

CHAPTER 3

Related Problems of LCS

The LCS problem can be add some additional conditions will become some LCS related problems. The cyclic string correction problem is add condition in input. The longest increasing subsequence is add condition in the mismatch condition. The constrained longest common subsequence is add condition in the output with a constrained word.

3.1 The Cyclic String Correction Problem

The linear string to string correction problem was defined by Wagner and Fischer in 1974 [129], which can be solved by using a well known dynamic programming algorithm. Using this algorithm, we can obtain the edit distance for any prefix of A and B, which takes $O(mn)$ time and $O(mn)$ space, where m and n are the lengths of input sequences A and B, respectively. In 1990, Maes defined a cyclic string correction problem, whose input are cyclic stings instead of linear ones [84].

We define a *cyclic shift* σ as an operation of shift, which can be written as

$$\sigma(a_1a_2\cdots a_n) = a_2a_3\cdots a_na_1.$$

Given $A = a_1a_2\cdots a_n$, which is a string with length n. Let $\sigma^k(A)$ be the function which repeats the σ function k times and will produce the shifted sequence $a_{k+1}a_{k+2}\cdots\ a_na_1a_2\cdots a_k$. According to the above definition, it is clear that $A = \sigma^n(A)$. Here we denote $[A]$ as a *cyclic string*, which is a set composed by $\sigma^1(A)\bigcup\sigma^2(A)\bigcup\cdots\sigma^{n-1}(A)$. Given two cyclic strings $[A]$ and $[B]$, we can compute the edit distance

$\delta([A],[B]) = min\{\delta(\sigma^k(A), \sigma^l(B)) \mid 1 \leq k \leq n =\mid A \mid, 1 \leq l \leq m =\mid B \mid\}$, where $\delta(A, B)$ is the edit distance of two strings A and B.

In Chapter 4, we will show that the longest common subsequence problem is a special case of the edit distance problem. Therefore, any algorithm that solves the edit distance problem can be easily modified to solves the LCS problem. Hence, here we take the cyclic longest common subsequence problem as an example to explain the cyclic string problem, and show some of its properties. The cyclic longest common subsequence problem can be formalized as follows:

Definition : Given two cyclic strings $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$, $n \geq m$, the *cyclic longest common subsequence problem* is to find $LCS([A],[B]) =$ *max* $\{LCS(\sigma^k(A),\sigma^l(B)) \mid 1 \leq k \leq n =\mid A \mid, 1 \leq l \leq m =\mid B \mid\}$, where $LCS(A, B)$ is the length of longest common subsequence of strings A and B.

Theorem 3.1 *$LCS([A],[B]) = l$, where $1 \leq s \leq n$ and $1 \leq t \leq m$, if and only if $LCS(\sigma^s(A), B) = l$ [84].*

According to Theorem 3.1, we can transfer the cyclic longest common subsequence problem $LCS([A],[B])$ to LCS($[A], B$). In this problem, we can take the original LCS algorithm to solve this problem. If we calculate $LCS(\sigma^1(A), B)$, $LCS(\sigma^2(A), B)$, $\cdots$, $LCS(\sigma^n(A), B)$ one by one, it will take $O(n^2m)$ time.

We can use $(2n+1) \times (m+1)$ space to store $LCS([A], B)$ score matrix, each $LCS[\sigma^k(A), B]$ can be found in the $(k+n) \times (m+1)$ windows. In Maes's algorithm, we compute the value inside the boundaries instead of computing all values in the whole windows.

Theorem 3.2 *Let L_s and L_t be two LCS paths which can be calculated from LCS($\rho^s(A), B$) and LCS($\rho^t(A), B$). If $s \neq t$, then L_s and L_t never cross through each other [84].*

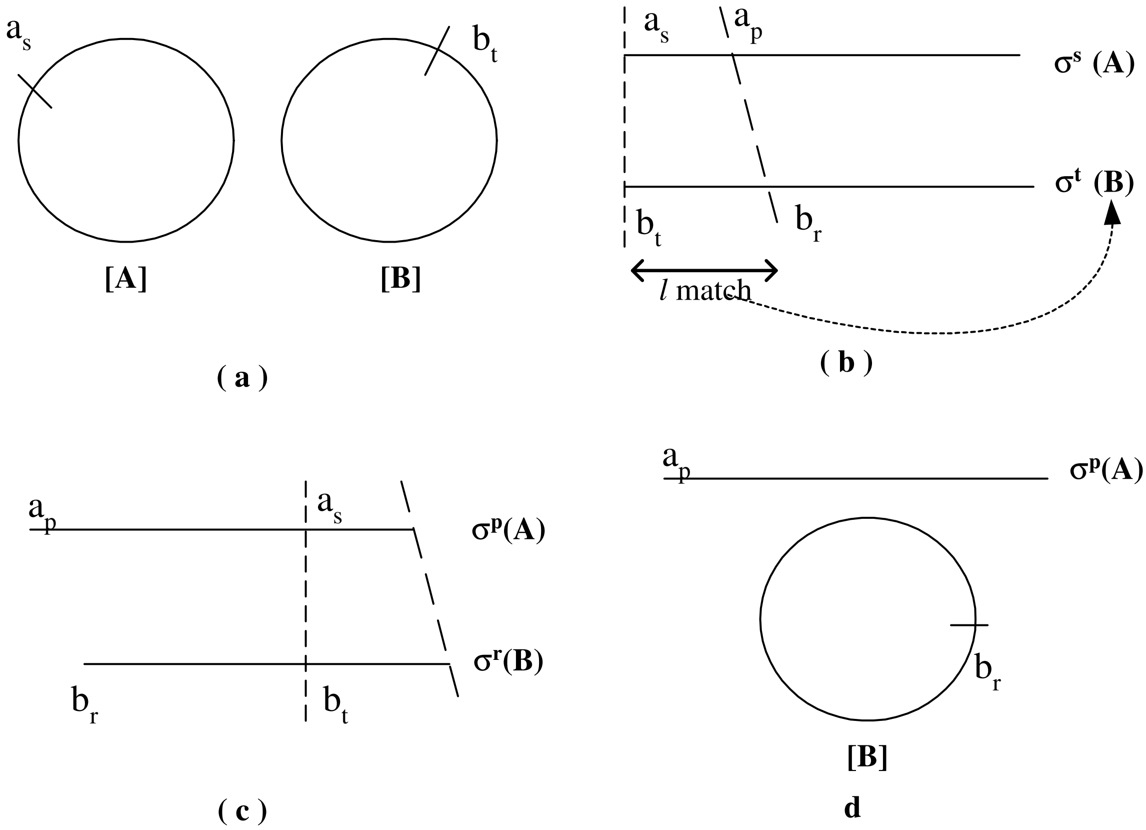


Figure 3.1 An example of transforming two cyclic strings.

Take Figure 3.2 as an example, L_s is the LCS path of $LCS(\sigma^s(A), B)$, and L_t is the LCS path of $LCS(\sigma^t(A), B)$. Let $e1$ and $e2$ denote the bold and dash paths between p_d and P_c, respectively.

(1) If the number of matches on $e1$ is not less than that on $e2$, we can find another LCS path passing through $e1$ in $LCS(\sigma^t(A), B)$ which is better than L_t.

(2) Otherwise, we can find another LCS path passing through $e2$ in $LCS(\sigma^s(A), B)$ which is better than L_s.

Observably, conditions (1) and (2) lead to a contradiction. Therefore, the LCS path in each windows will never cross by each other.

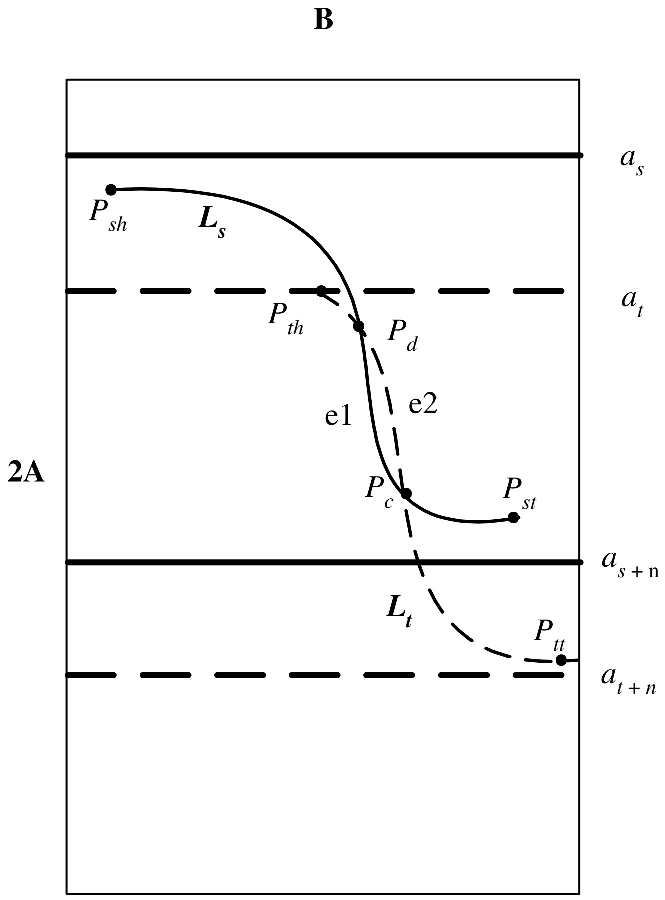


Figure 3.2 An example of LCS crossing.

Maes proposed an algorithm for solving this problem in $O(nm \log n)$ time [84] in 1990. According to Theorem 3.2, we know that any two LCS paths in different windows would never cross. See Figure 3.3, we compute L_0 and L_n as boundaries first. In fact, L_0 is the same as L_n . Then we compute the $L_{\lceil n/2 \rceil}$ by only computing the values of the area bounded by L_0, L_n and the horizontal lines on positions $\lceil n/2 \rceil$ and $\lceil 3n/2 \rceil$. With the new lower boundary $L_{\lceil n/2 \rceil}$, we can find $L_{\lceil n/4 \rceil}$ by only computing the area bounded by L_0 and $L_{\lceil n/2 \rceil}$. Obviously, we can use the divide and conquer method to separate the original problem to the smaller ones. In Figure 3.3, a_k is the kth character of A and the dash line is the boundary of the LCS.

Because LCS paths never cross to each other, we only calculate the region between the boundaries of the windows.

3.2 The Longest Increasing Subsequence Problem

The LIS (*longest increasing subsequence*) problem is well-studied in computer science [23,47] and also has some biology applications [39,109,138].

The longest increasing subsequence problem is to find a longest increasing subsequence of a given sequence of distinct integers $S = x_1x_2 \cdots x_n$. For example, given a sequence S = 3154782, we can see that 3578 and 478 are both increasing subsequences in S. In addition, 3578 is longer than 478, and no other sequence in S is longer than 3578. Therefore, 3578 is the longest increasing subsequence in 3154782.

The LIS problem can be formally defined as follows:

Definition : Given a comparable sequence of n integers $S = x_1x_2 \cdots x_n$, where all integers are distinct, the LIS problem is to find a subsequence $x_{i_1}x_{i_2} \cdots x_{i_k}$ of S, where $x_{i_1} \leq x_{i_2} \leq, \cdots \leq x_{i_k}$ and k is the maximum.

If we have the sorted sequence of the input, then we can find the LIS simply by using the dynamic programming method. We can treat the input as sequence A, and the sorted input as sequence B, then $LCS(A, B)$ will calculate the longest increasing subsequence in A. For example, in Figure3.4, we can show that the LIS of 31547826 is 3578. Hunt and Szymanski [65] presented an algorithm which can solve the LIS problem in $O(n \log n)$ time and its time complexity is reduced to $O(n \log \log n)$ if the van Emde Boas data structure is used. Bespamyatnikh and Segal [23] also used the van Emde Boas data structure to find out all longest increasing subsequences in $O(n \log \log n)$.

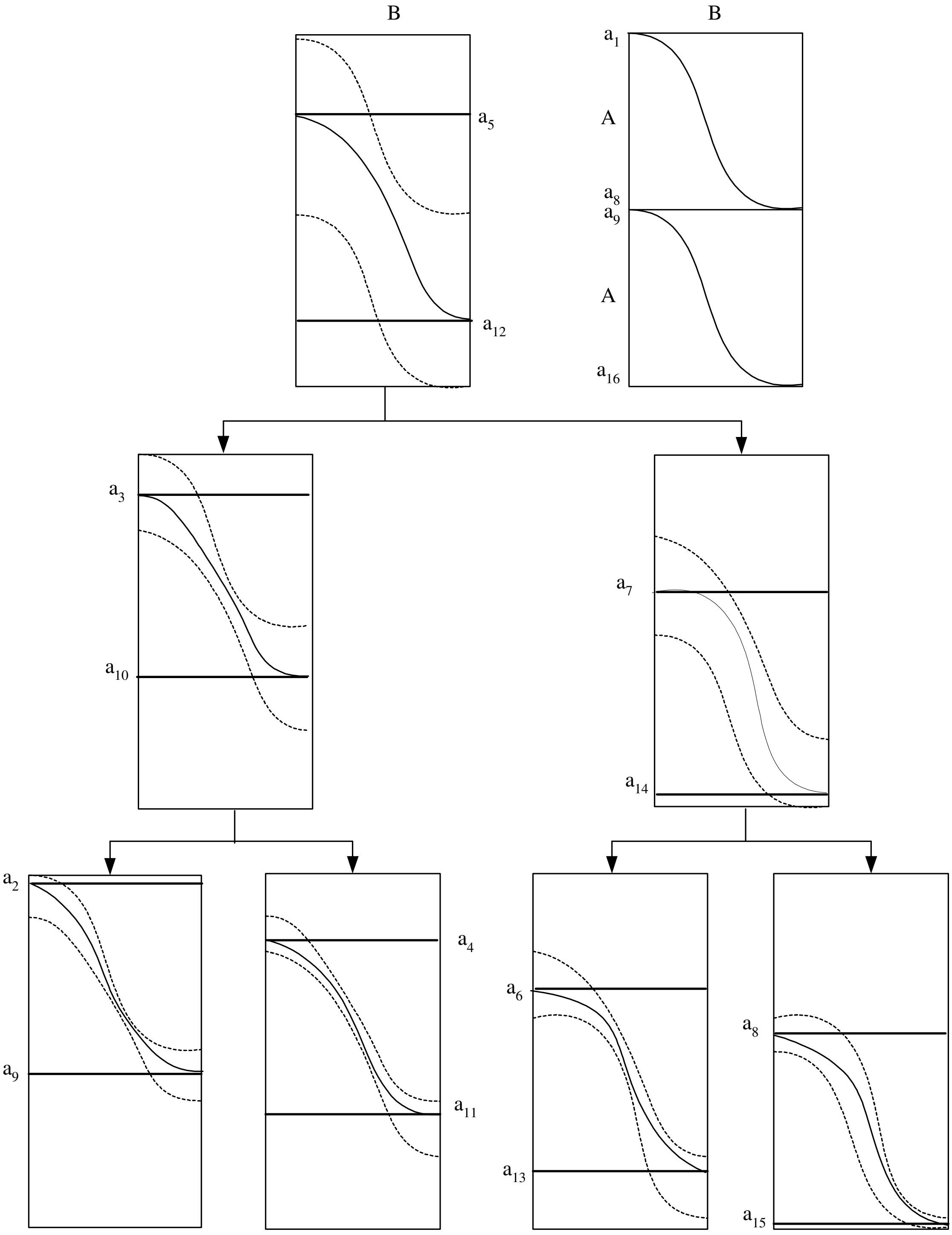
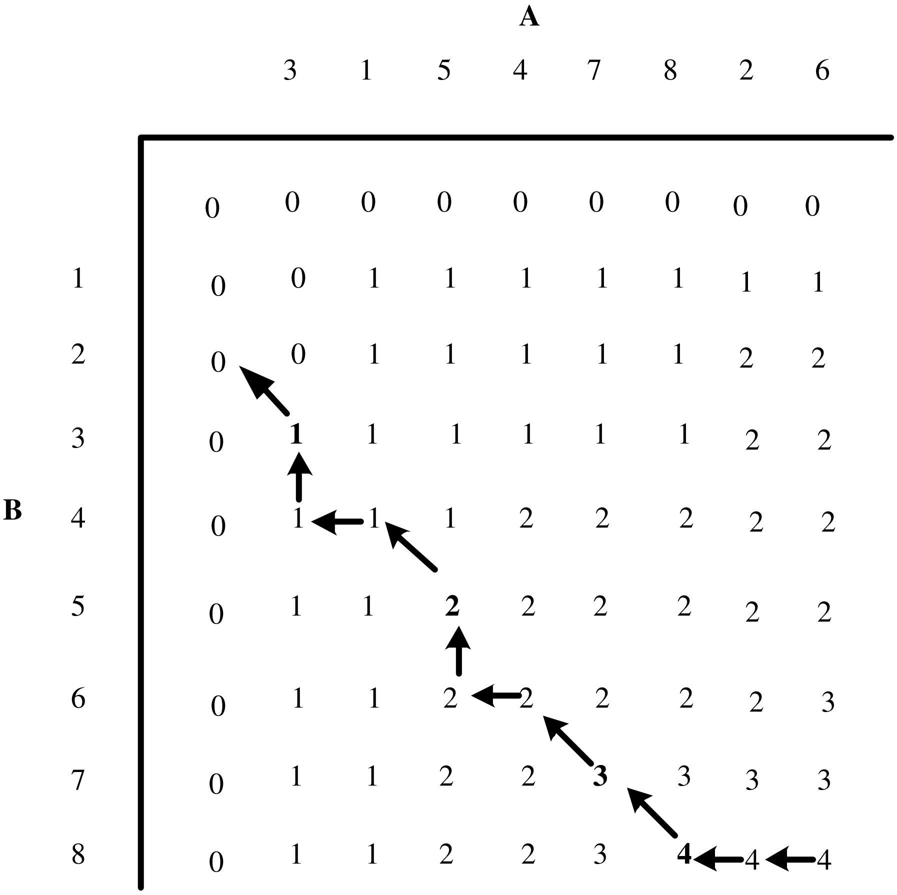


Figure 3.3 The divide and conquer of Maes's algorithm. The a_k is the kth character of A and the dash line is the boundary of the LCS. Here, $A = a_1 a_2 \cdots a_8$. That is, $a_i = a_{8+i}, 1 \leq i \leq 8$.



						A				
			3	1	5	4	7	8	2	6
		0	0	0	0	0	0	0	0	0
	1	0	0	1	1	1	1	1	1	1
	2	0	0	1	1	1	1	1	2	2
	3	0	**1**	1	1	1	1	1	2	2
B	4	0	1	1	1	2	2	2	2	2
	5	0	1	1	**2**	2	2	2	2	2
	6	0	1	1	2	2	2	2	2	3
	7	0	1	1	2	2	**3**	3	3	3
	8	0	1	1	2	2	3	**4**	4	4

Figure 3.4 Solving LIS with the LCS algorithm, $A = \texttt{31547826}$ is the input sequence and $B = \texttt{12345678}$ is the sorted sequence. `3578` is the longest increasing subsequence in this case.

Year	Author(s)	Time Complexity
1961	Schensted [110]	$O(n \log n)$
1977	Hunt and Szymanski [65]	$O(n \log \log n)$
2000	Bespamyatnikh and Segal [23]	$O(n \log \log n))$

Table 3.1 The algorithms for the LIS problem.

Albert et al. [4] gave an algorithm to solve the window LIS problems. Given a sequence $S = x_1x_2 \cdots x_n$ and a window size $w \leq n$, a window of S is defined as a substring $x_{i+1}x_{i+2} \cdots x_{i+w}$ where $0 \leq i \leq n - w$. There are three kinds of problems as follows:

(1) Local Max Value: finding the length k of the longest increasing subsequence in all windows.

(2) Local Max Sequence: finding the LIS in each window.

(3) Global Max Sequence: finding the windows with the longest increasing subsequence among all windows.

Albert et al. [4] takes $O(n \log \log n + OUTPUT)$ time to solve these three problems.

Given two comparable sequences $S = x_1x_2 \cdots x_n$ and $T = y_1y_2 \cdots y_m$. Yang et al. [137] proposed an algorithm to solve the longest common increasing subsequence problem between two comparable sequences in $O(mn)$ time and $O(mn)$ space.

3.3 The Constrained Longest Common Subsequence Problem

The constrained longest common subsequence problem can be defined as follows [104]:

Definition : Given two strings $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$ and a constrained sequence $C = c_1c_2 \cdots c_p$, the constrained longest common subsequence problem (CLCS) is to find a longest common subsequence containing C as its subsequence.

In the original LCS problem, the longest common subsequence is the optimal. However, the CLCS want to find a longest common subsequence containing specific sequence may be shorter than the original LCS. For example, given two strings `CACAGTAG` and `AGCTGACG`, the longest common subsequence is `ACGAG` . However, if we give a constrained sequence `CAC` in addition, then the constrained longest common subsequence is `CACG` which is different greatly from `ACGAG`.

Given two strings $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$ and a constrained sequence $C = c_1c_2 \cdots c_p$. Let $A_{q,r} = a_qa_{q+1} \cdots a_r$ and $B_{s,t} = b_sa_{s+1} \cdots a_t$ are *substrings* in A and B respectively, where $1 \leq q \leq r \leq n$ and $1 \leq s \leq t \leq m$. Let $LCS(q, r, s, t)$ denote the length of LCS between $A_{q,r}$ and $B_{s,t}$. For $1 \leq k \leq p$, $1 \leq i \leq n$ and $1 \leq j \leq m$, let $LCS_k(i, j)$ denote the length of LCS between $A_{1,i}$ and $B_{1,j}$ with $a_i = b_j = c_k$. Because the constrained sequence must be in the output, we can use each constrained sequence character c_k as one break point. Based on this idea, Tsai [123] uses c_k to separate the input string to several individual parts, and calculate the common subsequences among these parts with dynamic programming. The scoring function can be formalize as follows:

$$LCS(A, B) = \max_{1 \leq i \leq n, 1 \leq j \leq n}\{LCS_p(i, j) + LCS(i + 1, n, j + 1, m)\ \}$$

$$LCS_k(i, j) = LCS(1, i - 1, 1, j - 1) + 1\ ,\ where\ k = 1, a_i = b_j = c_k$$

$$L_k(i, j) = \max_{1 \leq x < i, 1 \leq y < j}\{\ L_{k-1}(x, y) + LCS(x + 1, i - 1, y + 1, j - 1) + 1 \quad \}$$
$$,\ where\ a_i = b_j = c_k, 2 \leq k \leq p,\ 1 \leq i \leq n\ and\ 1 \leq j \leq m.$$

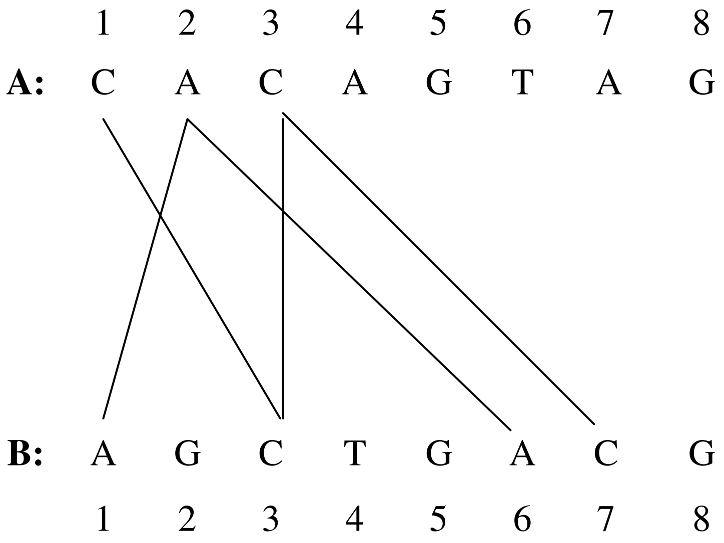


Figure 3.5 Possible break points on A = `CACAGTAG`, B = `AGCTGACG` with the constrained sequence C = `CAC`.

Figure 3.5 shows an example of break points, where the two input strings $A =$ `CACAGTAG` and $B =$ `AGCTGACG`, and the constrained sequence $C =$ `GTA`.

$$
\begin{aligned}
&LCS(A,B) = \ LCS_3(3,7) + \ LCS(4,8,8,8) = \ LCS_3(3,7) \ +1 \\
&LCS_3(3,7) = \ LCS_2(2,6) + 1 \\
&LCS_2(2,6) = \ LCS_1(1,3) + 1 \\
&LCS_1(1,3) = \ 1 \\
&\text{Therefore, } LCS(A,B) = 4
\end{aligned}
$$

Hence, we can summarize that the algorithm takes $O(pm^2n^2)$ time.

Chin et. al [30] also followed the idea of break points and reduce the conditions as follows:

Case 1: $If a_i = b_j = c_k$, the CLCS score between $a_1a_2 \dots a_i$ and $b_1b_2 \dots b_j$ is equal to the LCS score between $a_1a_2 \dots a_{i-1}$ and $b_1b_2 \dots b_{j-1}$ plus the matching score. In addition, $a_1a_2 \dots a_{i-1}$ and $b_1b_2 \dots b_{j-1}$ may contain $c_1c_2 \dots c_{k-1}$.

Case 2: $Ifa_i = b_j$ and $a_i \neq c_k$, the CLCS score between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$ is equal to the LCS score between $a_1a_2 \ldots a_{i-1}$ and $b_1b_2 \ldots b_{j-1}$ plus the matching score. In addition, $a_1a_2 \ldots a_{i-1}$ and $b_1b_2 \ldots b_{j-1}$ may contain $c_1c_2 \ldots c_k$.

Case 3: $Ifa_i \neq b_j$ and $a_i \neq c_k$, the CLCS score between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$ is equal to the LCS score between $a_1a_2 \ldots a_{i-1}$ and $b_1b_2 \ldots b_j$. In addition, $a_1a_2 \ldots a_{i-1}$ and $b_1b_2 \ldots b_j$ may contain $c_1c_2 \ldots c_k$.

Case 4: $Ifa_i \neq b_j$ and $b_j \neq c_k$, the CLCS score between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$ is equal to the LCS score between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_{j-1}$. In addition, $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_{j-1}$ may contain $c_1c_2 \ldots c_k$.

We can formalize the above cases as follows:

$$L_0(i,0) = 0 \ , \ 0 \leq i \leq m$$

$$L_0(0,j) = 0 \ , \ 0 \leq j \leq n$$

$$L_0(i,j) = \max \left\{ \begin{array}{l} L_0(i-1,j) \\ L_0(i,j-1) \\ L_0(i-1,j-1)+1 \quad if \ a_i = b_j \end{array} \right\}, where \ \begin{array}{c} 1 \leq i \leq m \\ 1 \leq j \leq n \end{array}$$

$$L_k(i,0) = -\infty, \ 0 \leq i \leq m \ and \ 1 \leq k \leq p$$

$$L_k(0,j) = -\infty, \ 0 \leq j \leq n \ and \ 1 \leq k \leq p$$

$$L_k(i,j) = \max \left\{ \begin{array}{ll} L_k(i-1,j) & \\ L_k(i,j-1) & \\ L_k(i-1,j-1)+1 & if \ a_i = b_j \\ L_{k-1}(i-1,j-1)+1 & if \ a_i = b_j = c_k \end{array} \right\}, where \ \begin{array}{c} 1 \leq i \leq m \\ 1 \leq j \leq n \\ 1 \leq k \leq p \end{array}$$

Take Figure 3.6 for example, we should compute the original scoring matrix L_0 first, and find the break point in $L_0(1,1)$. In $L_1(3,2)$, $a_3 = C$ and $b_2 = A$ is a mismatch, then $L_1(3,2) = \max\{L_1(2,3)),(L_1(3,1)\} = 1$. In $L_1(3,3))$, $a_3 = C$ and $b_3 = C$ is a match, then $L_1(3,3) = L_0(2,2) + 1 = 2$. We can follow the function to compute k scoring matrices and use the tracing back technique to find the constrained longest common subsequence. We also give an example of tracing back in Figure 3.7.

				1	2	3	4	5	6	7	8
			C	A	C	A	G	T	A	G	
		0	0	0	0	0	0	0	0	0	
1	A	0	0	1	1	1	1	1	1	1	
2	G	0	0	1	1	1	2	2	2	2	
3	C	0	1	1	2	2	2	2	2	2	
4	T	0	1	1	2	2	2	3	3	3	
5	G	0	1	1	2	2	3	3	3	4	
6	A	0	1	2	2	3	3	3	4	4	
7	C	0	1	2	3	3	3	3	4	4	
8	G	0	1	2	3	3	4	4	4	5	

L_0

			1	2	3	4	5	6	7	8
			C	A	C	A	G	T	A	G
		-	-	-	-	-	-	-	-	-
1	A	-	-	-	-	-	-	-	-	-
2	G	-	-	-	-	-	-	-	-	-
3	C	-	1	1	2	2	2	2	2	2
4	T	-	1	1	2	2	2	3	3	3
5	G	-	1	1	2	2	3	3	3	4
6	A	-	1	2	2	3	3	3	4	4
7	C	-	1	2	3	3	3	3	4	4
8	G	-	1	2	3	3	4	4	4	5

L_1

			1	2	3	4	5	6	7	8
			C	A	C	A	G	T	A	G
		-	-	-	-	-	-	-	-	-
1	A	-	-	-	-	-	-	-	-	-
2	G	-	-	-	-	-	-	-	-	-
3	C	-	-	-	-	-	-	-	-	-
4	T	-	-	-	-	-	-	-	-	-
5	G	-	-	-	-	-	-	-	-	-
6	A	-	-	2	2	3	3	3	4	4
7	C	-	-	2	3	3	3	3	4	4
8	G	-	-	2	3	3	4	4	4	5

$\mathbf{L_2}$

			1	2	3	4	5	6	7	8
			C	A	C	A	G	T	A	G
		-	-	-	-	-	-	-	-	-
1	A	-	-	-	-	-	-	-	-	-
2	G	-	-	-	-	-	-	-	-	-
3	C	-	-	-	-	-	-	-	-	-
4	T	-	-	-	-	-	-	-	-	-
5	G	-	-	-	-	-	-	-	-	-
6	A	-	-	-	-	-	-	-	-	-
7	C	-	-	-	3	3	3	3	3	3
8	G	-	-	-	3	3	4	4	4	4

$\mathbf{L_3}$

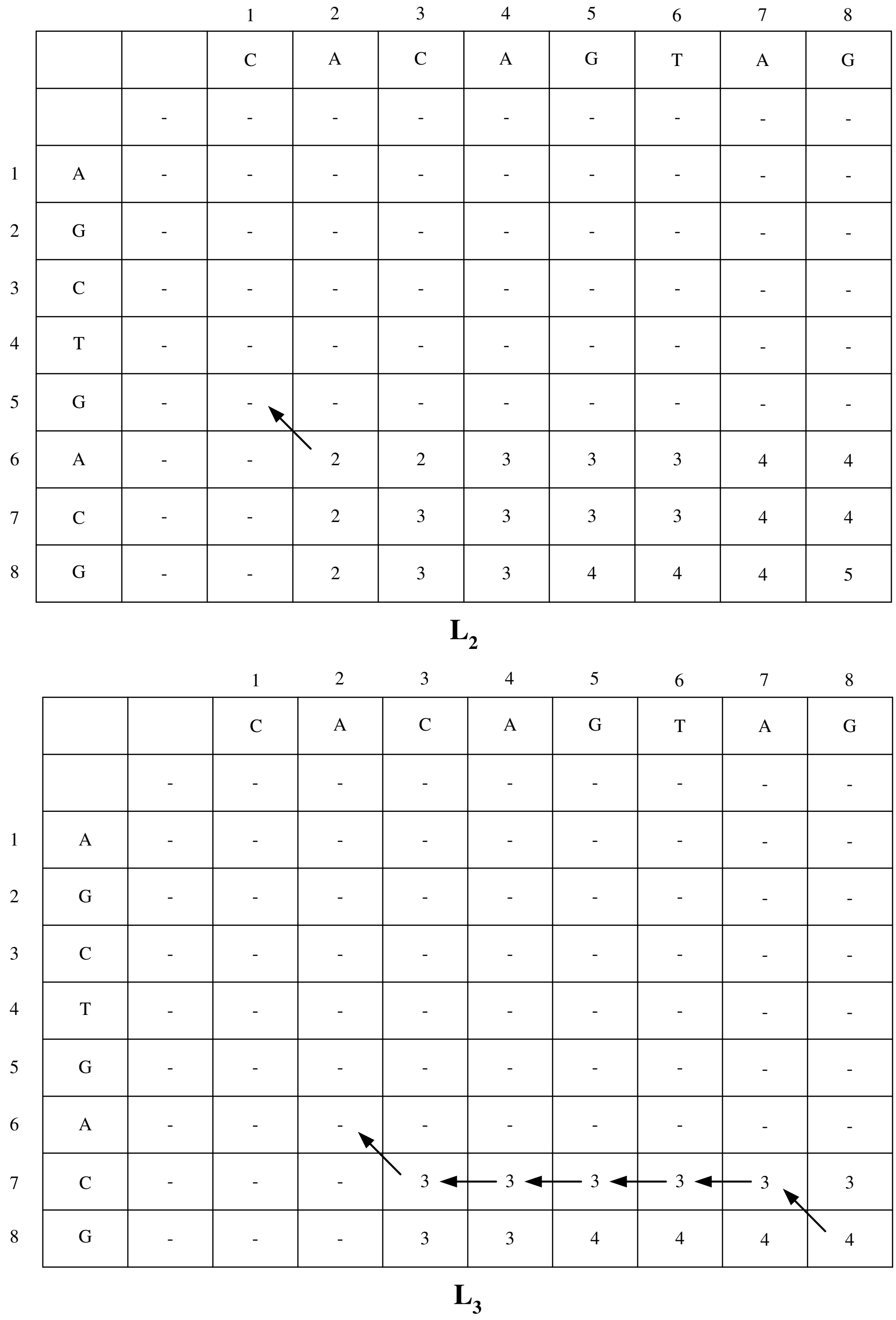
Figure 3.6 The result obtained from two input sequences `CACAGTAG` and `AGCTGACG` with constrained sequence `CAC`. Here, L_k denotes the scoring matrix of an LCS which contains the constrained sequence $c_1c_2\cdots c_k$.

			1	2	3	4	5	6	7	8
			C	A	C	A	G	T	A	G
		0	0	0	0	0	0	0	0	0
1	A	0	0	1	1	1	1	1	1	1
2	G	0	0	1	1	1	2	2	2	2
3	C	0	1	1	2	2	2	2	2	2
4	T	0	1	1	2	2	2	3	3	3
5	G	0	1	1	2	2	3	3	3	4
6	A	0	1	2	2	3	3	3	4	4
7	C	0	1	2	3	3	3	3	4	4
8	G	0	1	2	3	3	4	4	4	5

L_0

			1	2	3	4	5	6	7	8
			C	A	C	A	G	T	A	G
		-	-	-	-	-	-	-	-	-
1	A	-	-	-	-	-	-	-	-	-
2	G	-	-	-	-	-	-	-	-	-
3	C	-	1	1	2	2	2	2	2	2
4	T	-	1	1	2	2	2	3	3	3
5	G	-	1	1	2	2	3	3	3	4
6	A	-	1	2	2	3	3	3	4	4
7	C	-	1	2	3	3	3	3	4	4
8	G	-	1	2	3	3	4	4	4	5

L_1



			1	2	3	4	5	6	7	8
			C	A	C	A	G	T	A	G
		-	-	-	-	-	-	-	-	-
1	A	-	-	-	-	-	-	-	-	-
2	G	-	-	-	-	-	-	-	-	-
3	C	-	-	-	-	-	-	-	-	-
4	T	-	-	-	-	-	-	-	-	-
5	G	-	-	-	-	-	-	-	-	-
6	A	-	-	2	2	3	3	3	4	4
7	C	-	-	2	3	3	3	3	4	4
8	G	-	-	2	3	3	4	4	4	5

L_2

			1	2	3	4	5	6	7	8
			C	A	C	A	G	T	A	G
		-	-	-	-	-	-	-	-	-
1	A	-	-	-	-	-	-	-	-	-
2	G	-	-	-	-	-	-	-	-	-
3	C	-	-	-	-	-	-	-	-	-
4	T	-	-	-	-	-	-	-	-	-
5	G	-	-	-	-	-	-	-	-	-
6	A	-	-	-	-	-	-	-	-	-
7	C	-	-	-	3	3	3	3	3	3
8	G	-	-	-	3	3	4	4	4	4

L_3

Figure 3.7 The tracing path obtained from sequences `CACAGTAG` and `AGCTGACG` with constrained sequence `CAC`.

CHAPTER 4

The Edit Distance Problem

The edit distance is used widely in many areas, such as computational biology, speech recognition and text retrieval [7,40,54,92,96,109,111,127]. In this chapter, we will introduce the Levenshtein edit distance in Section 4.1. There are many different kinds of operations in the edit distance problem. The edit distance has different complexities while adopting different operations. In Section 4.2, we will introduce some different operations in the edit distance problem and list some complexities of them.

4.1 The Edit Distance Problem

The *Levenshtein edit distance* problem was first proposed by Levenshtein in 1965 [78, 79]. A few years later, Wagner and Fischer presented some definitions and properties in their paper [129]. We will follow their definition in this section. Let $A = a_1a_2 \cdots a_n$ be a sequence over an alphabet set Σ, and $A_{i,j} = a_ia_{i+1} \cdots a_j$, where $1 \leq i \leq j \leq n$. Let Λ denote the null string and $A_{i,j} = \phi$ if $i > j$.

An *edit operation* s is an ordered pair $(a, b) \neq (\phi, \phi)$ of characters, which indicate $a \to b$. An edit operation (a, b) called a *deletion* if $b = \phi$, an *insertion* if $a = \phi$, and a *replacement* if $a \neq \phi$ and $b \neq \phi$. An *edit sequence* $S = s_1s_2 \cdots s_m$ is a sequence of edit operations. We denote S as a sequence of strings $A_0A_1 \cdots A_m$, such that $A = A_0$, $B = A_m$, and $A_{i-1} \to A_i$ via s_i for $1 \leq i \leq m$. In short, we can use $s_1s_2 \cdots s_m$ to change string A to string B.

Let $\gamma(s)$ denote the cost function of edit operation s. For an edit sequence S, $\gamma(S) = \sum_{i=1}^{m} \gamma(s_i)$. Therefore, we can denote the *edit distance* $\delta(A, B)$ from string A to string B as the minimal cost for transforming A to B. Formally,

$$\delta(A, B) = min\{\gamma(S) \mid S \text{ is an edit sequence used for transforming } A \text{ to } B\}.$$

Let A and B be strings of lengths n and m, respectively. Let $D(i, j) = \delta(A_{1,i}, B_{1,j})$, for $1 \leq i \leq n, 1 \leq j \leq m$. We have

$$D(i,j) = min \begin{cases} D(i-1, j-1) + \gamma(a_i \rightarrow b_j), \\ D(i-1, j) + \gamma(a_i \rightarrow \phi), \\ D(i, j-1) + \gamma(\phi \rightarrow b_j), \end{cases}$$

where $D(i, j)$ is the optimal edit distance of the first i characters of $A_{1,i} = a_1, a_2, \cdots, a_i and B_{1,j} = b_1, b_2, cdots, b_j$.

The last paired character must be one of the following:

Case 1: Delete a_i. The edit distance between $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$ is equal to the edit distance between $a_1 a_2 \ldots a_{i-1}$ and $b_1 b_2 \ldots b_j$ plus the deleting score.

Case 2: Insert b_j. The edit distance between $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$ is equal to the edit distance between $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_{j-1}$ plus the inserting score.

Case 3: Replacing a_i to b_j. The edit distance between $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$ is equal to the edit distance between $a_1 a_2 \ldots a_{i-1}$ and $b_1 b_2 \ldots b_{j-1}$ plus the replacing score.

Because each $D(i, j)$ is the cost of the minimum cost between $A_{1,i}$ and $B_{1,j}$, and the above three conditions are considered, the $D(i, j)$ should be the minimal cost for transforming $A_{1,i}$ to $B_{1,j}$.

Let T be a *trace* from A to B, where T is any set of integers (i, j) satisfying:

1. $1 \leq i \leq n$ and $1 \leq j \leq m$.

2. For any distinct (i_1, j_1) and (i_2, j_2) in T, (a) $i_1 \neq i_2$ and $j_1 \neq j_2$ (b) $i_1 < j_1$ if and only if $j_1 < j_2$.

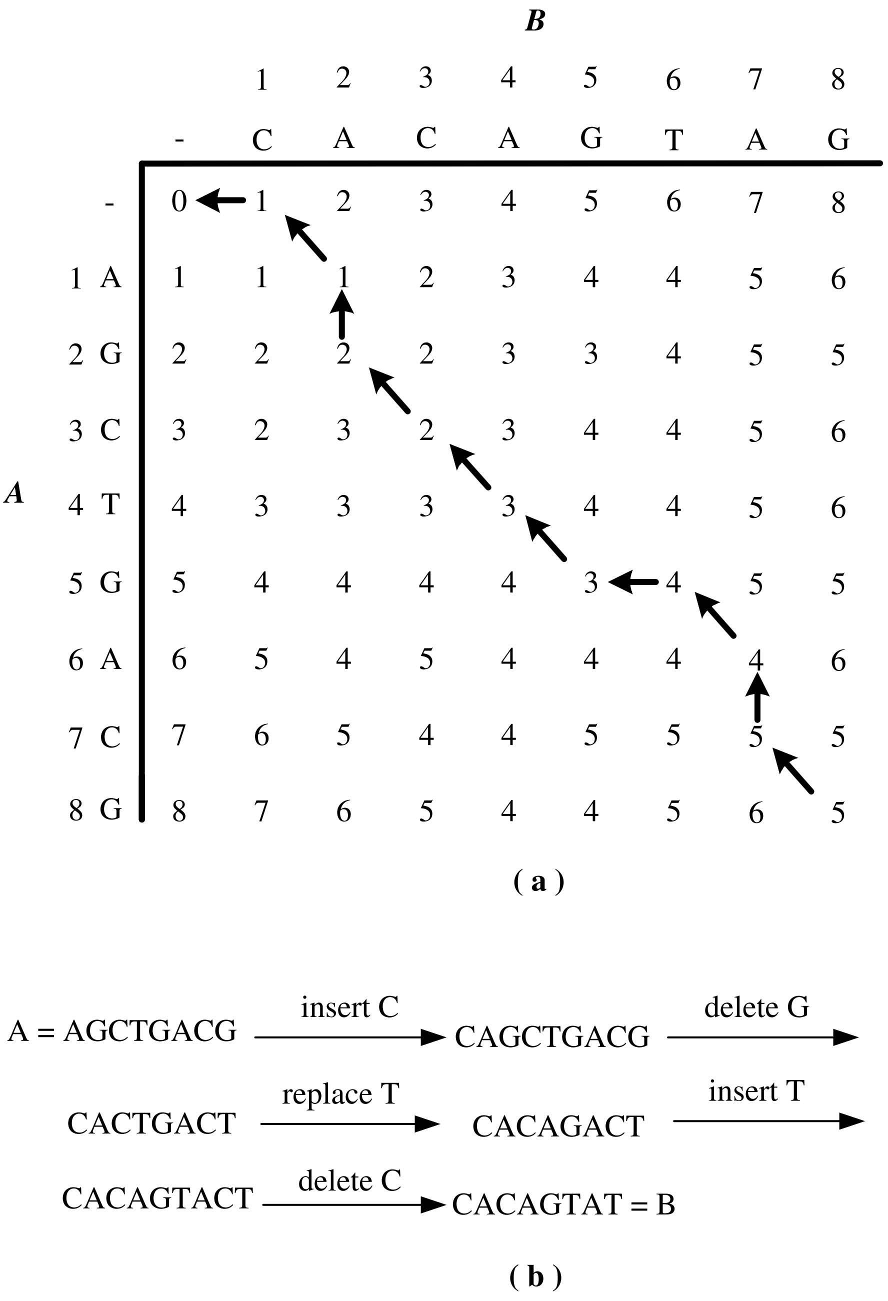
According to above definition, we can make sure that a trace T can reach any position of A and B by condition (1). Also, it must contain at least one line and never be crossed by condition (2).

Therefore, we can obtain a $\delta(A, B) = D(m, n)$. If $D(m, n)$ is found, the trace T can also be determined by tracing back. In Figure 4.1, we show the score matrix between two strings and edit operations to transform one string to the other.

Let both the cost of insertion and deletion be 1, and the the cost of replacement is 2. We have $\delta(A, B) = (n - \rho(A, B)) + (m - \rho(A, B))$, where $\rho(A, B)$ is the longest common subsequence of A and B. This equation means that if a longer common subsequence exits, the edit distance will be smaller. Hence, following the equation $\delta(A, B) = n + m - 2\rho(A, B)$, we can obtain $\rho(A, B) = (n + m - \delta(A, B))/2$. Thus, we can show the longest common subsequence problem is a special case of the edit distance problem.

4.2 Other Operations in the Edit Distance Problem

In the previous section, we have introduced the Levenshtein edit distance. Generally, there are three operations in the Levenshtein edit distance, which are insertion, deletion and replacement. In this section, we will present the edit distance problem which is extended by adding some other different operations, including reversal, transposition and move.



			1	2	3	4	5	6	7	8
		-	C	A	C	A	G	T	A	G
	-	0	1	2	3	4	5	6	7	8
1	A	1	1	1	2	3	4	4	5	6
2	G	2	2	2	2	3	3	4	5	5
3	C	3	2	3	2	3	4	4	5	6
4	T	4	3	3	3	3	4	4	5	6
5	G	5	4	4	4	4	3	4	5	5
6	A	6	5	4	5	4	4	4	4	6
7	C	7	6	5	4	4	5	5	5	5
8	G	8	7	6	5	4	4	5	6	5

(a)

(b)

Figure 4.1 An example for finding the edit distance between two stings.(a) The tracing back in the score matrix between A = `CACAGTAG` and B = `AGCTGACG`. (b) The operations to transform A to B.

Given two string $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$, and let $\delta(A, B)$ be the edit distance function which can transfer A to B. We shall defined some edit distance operations are defined as follows:

1. For a string, an insertion of a character c at position i transforms A to $a_1a_2\cdots a_{i-1}\ c\ a_i\ a_{i+1}\cdots a_n$.
2. For a string, a deletion at position i transforms A to $a_1a_2\cdots a_{i-1}a_{i+1}\cdots a_n$.
3. For a string, a replacement at position i with a character c transforms A to $a_1a_2\cdots a_{i-1}\ \ c\ a_{i+1}\cdots a_n$.
4. For a string, a block move (transposition) with parameters i, j, k where $1 \leq i \leq j \leq k \leq n$, transforms A to $a_1a_2\cdots a_{i-1}a_ja_{j+1}\cdots a_{k-1}a_ia_{i+1}\cdots a_{j-1}a_ka_{k+1}\cdots a_n$.
5. For a string, a reversal with parameters i, j where $1 \leq i \leq j \leq n$, transforms A to $a_1a_2\cdots a_{i-1}a_ja_{j-1}\cdots a_ia_{j+1}\cdots a_n$.

Different models for edit distance may have different complexity in time and space. For example, if the cost of insertion and deletion are both 1, then we can reduce this problem to the LCS problem. In addition, if only transposition operations are allowed, we can reduce it to the sorting problem. The model with block move can be reduced to genome rearrangement problem. In the following, we will introduce these problems one by one.

Definition : Given two strings A and B, the Hamming distance is the number of different characters between two strings.

According to the above definition, we know that the Hamming distance is the same as the edit distance problem if the cost of replacement is set to 1.

Year	Author(s)	Time Complexity
1995	Kececioglu and Sankoff [69]	2-approximation algorithm
1996	Bafna and Pevzner [18]	1.75-approximation algorithm
1998	Christie [31]	1.5-approximation algorithm in $O(n^4)$time
2000	Walter et .al [130]	2.25-approximation algorithm in $O(b^2)$ time

Table 4.1 Some algorithms for sorting by reversals.

Year	Author(s)	Time Complexity
1998	Bafna and Pevzner [17]	2,1.75 and 1.5-approximation algorithms
1998	Christie [32]	1.5-approximation algorithms in $O(n^4)$time
2003	Hartman and Shamir [55]	1.5-approximation algorithm

Table 4.2 Some algorithms for sorting by transpositions.

If the input of the edit distance problem is the permutation of n numbers, then we can reduce it to another well defined problem.

Definition : Given two permutations A and B, the reversal distance problem is to find the minimal reversal operations which can transform A to B.

In the reversal distance problem, Bafna and Pevzner presented a 1.5-approximation algorithm with cycle graph in $O(n^2)$ time [18]. In the same year, Christie presented a different approach with a 1.5-approximation algorithm in $O(n^4)$ time [31].After two years, Walter [130] gave a 2.25- approximation algorithm with a simpler data structure in $O(b^2)$ time, where b is the number of nonconsecutive pairs. We summarize these algorithms In Table 4.1.

Definition : Given two permutations A and B, the transposition distance problem is to find the minimal reversal operations which can transform A to B.

Definition : Given a query string A and a database D with k sequences, The SNN (*sequence nearest neighbors*) problem is to find a string B in D such that

$d(A, B) \leq d(A, S)$ for any other string S in D, where $d(A, B)$ denotes the distance between A and B.

For the SNN problem, Muthukrihan and Sahinalp presented an $O(\log l(\log l)^2)$-approximation algorithm with character edits(insertion, deletion and replacement) and block edits(moves, copies, deletes and reversals), where l is the length of the longest string in D [90]. With only character edits and block reversals, the $d(a, b)$ can be computed in $O(l \log^3 l)$ time [90].

In 2002, Cormode and Muthukrishnan presented an algorithm for solving the edit distance with insertion, deletion, replacement and block move [35]. In this algorithm, they proved that their algorithm is $O(\log n \log n)$-approximation to the optimal algorithm. In the same year, Shapira and Storer gave an $O(\log n)$-approximate algorithm to the same problem [113].

CHAPTER 5

Sequence Alignment

According to the number of input sequences, the sequence alignment can be divided into two categories: *pairwise alignment* and *multiple sequence alignment*. Pairwise sequence alignment is to compare two sequences, while multiple sequence alignment is to perform alignment on a set of input sequences.

The strategy of alignment can be divided into *global alignment* and *local alignment*. When we construct an alignment, there are two ways to present mismatches. One is to align sequences without gaps, such as BLAST and FASTA [7], and the other is to align sequence with gaps, such as Needleman-Wunsch algorithm [98] and Smith-Waterman algorithm [115]. Gapless alignment is widely used in large-scale database search because it is faster. However, gapless alignment may not be sufficient for finding sequences with low similarities.

Some researchers have studied the sequence alignment problem and develop some useful methods [6,7,15,33,46,77,89,108,126,131,139]. In biology, the sequence alignment has many applications [2,5,14,34,39,105,109,112,133].

5.1 Global Alignment

First, we give the global view of the sequence alignment. Global alignment is to align the full sequences of two given sequences, and is a powerful tool to measure similarity. For two sequences $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$, we can use the following scoring rules to measure the alignment.

1. If a_i is aligned with b_j, where $1 \leq i \leq n$ and $1 \leq j \leq m$, then the score increases 2.

2. If a_i is aligned with b_j and $a_i \neq b_j$, where $1 \leq i \leq n$ and $1 \leq j \leq m$, then the score decreases 1.

3. If a_i or b_j is aligned with a blank (gap), where $1 \leq i \leq n$ and $1 \leq j \leq m$, then the score is decreases 1.

We denote $V(i, j)$ as the optimal alignment score between $A_{1,i} = a_1a_2 \cdots a_i$ and $B_{1,j} = b_1b_2 \cdots b_i$. The dynamic programming for solving the sequence alignment problem is as follows:

1. If $a_i = b_j$, then a_i is aligned with b_j and the score is increased by 2. $V(i, j) = V(i-1, j-1) + 2$, where $V(i-1, j-1)$ means the optimal alignment score of $a_1a_2 \cdots a_{i-1}$ and $b_1b_2 \cdots b_{j-1}$.

2. If $a_i \neq b_j$, then there are three cases as follows.

Case 1. a_i is aligned with b_j and the score is decreased by 1. $V(i, j) = V(i-1, j-1) - 1$.

Case 2. a_i is aligned with $'-'$ and the score is decreased by 1. Thus, $V(i, j)$ $= V(i-1, j) - 1$, where $'-'$ denote a gap, which means one character to match with a null character.

Case 3. b_j is aligned with $'-'$ and the score is decreased by 1. In other words, $V(i, j) = V(i, j-1) - 1$.

For the values on the boundary, we have:

$$
\begin{aligned}
V(0,0) &= 0 \\
V(i,0) &= -i, 1 \leq i \leq n \\
V(0,j) &= -j, 1 \leq j \leq m
\end{aligned}
$$

where $V(0,j)$ means that $B_{1,j}$ is aligned to null character. $V(i,0)$ is defined in a similar way. For $i \geq 1$ and $j \geq 1$, we have

$$V(i,j) = \max \begin{cases} \max \begin{cases} V(i-1,j-1)-1 \\ V(i-1,j)-1 & \text{if } \; a_i \neq b_j \\ V(i,j-1)-1 \end{cases} \\ V(i-1,j-1)+2 & \text{if } \; a_i = b_j \end{cases} \tag{5.1}$$

For example, the alignment of A = `CGAAGUC` and B =`AAUGAGCUG` computed by dynamic programming is shown in Figure 5.1 with score -1 for mismatch and score 2 for match. Therefore, we can obtain the optimal alignment by tracing back as shown in Figure 5.2.

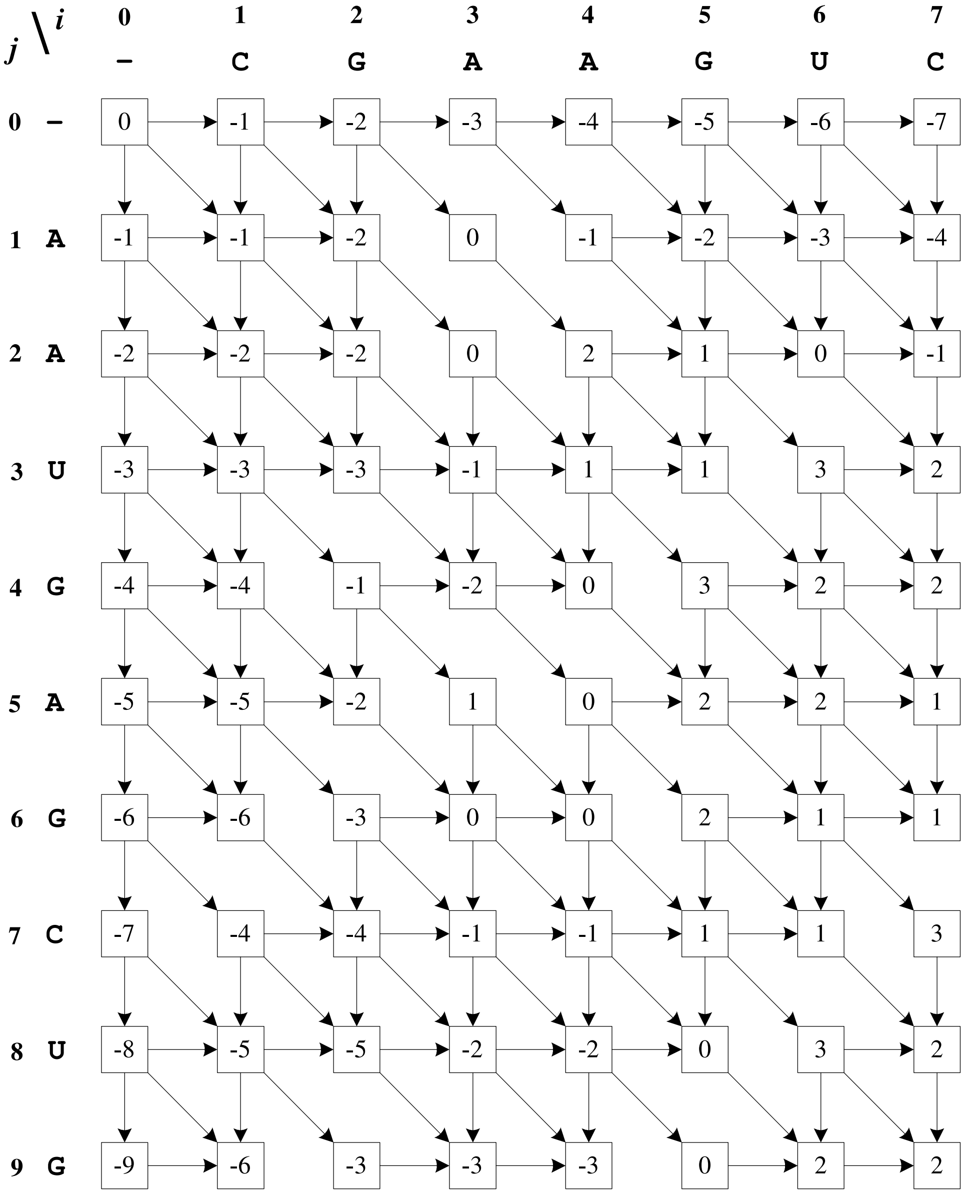
If we consider the gap penalty in addition, we can add some function in Equation 5.1. Let $\sigma(i,-)$ and $\sigma(-,i)$ denote the gap penalties of element i. Hence, we can make a more general equation from Equation 5.1 as follows:

$$\sigma(i,j) = \begin{cases} 2 & \text{if } \; a_i = b_j \\ -1 & \text{if } \; a_i \neq b_j \quad \text{(including } \; a_i =' -' \; \text{ or } \; b_j =' -' \text{)} \end{cases} \tag{5.2}$$

$$V(i,j) = \max \begin{cases} V(i-1,j-1) + \sigma(a_i, b_j) \\ V(i-1,j) + \sigma(a_i, -) \\ V(i,j-1) + \sigma(-, b_j) \end{cases} \tag{5.3}$$

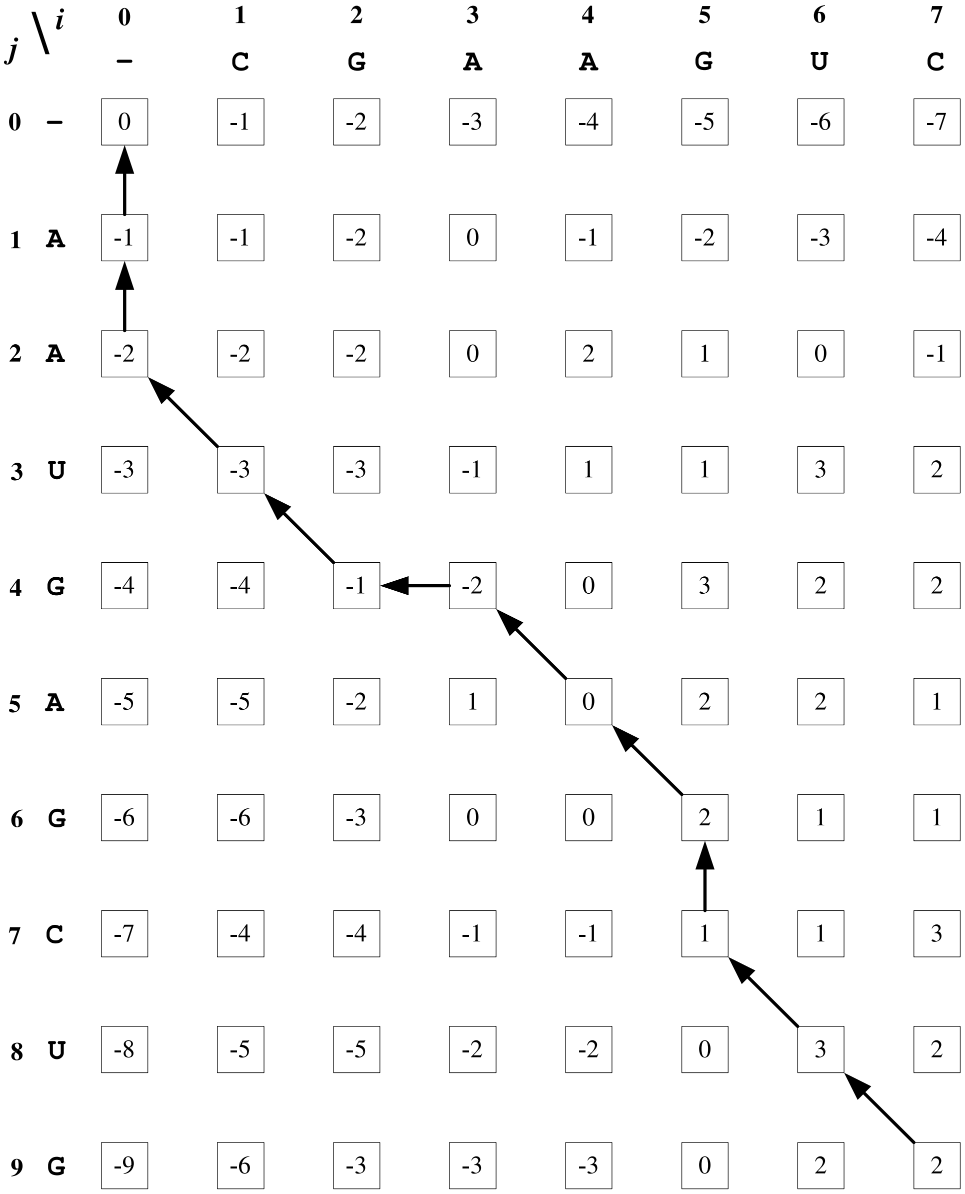
5.2 Local Alignment

In Section 5.1, we discussed the global alignment, which constructs an alignment through the whole input sequences. Global alignment emphasizes the similarity amoung entire input sequences. The other point of view is local alignment, which is stress on the region of the sequence. It emphasize the similarity of the substring of



$j \backslash i$		0	1	2	3	4	5	6	7
		–	C	G	A	A	G	U	C
0	–	0	-1	-2	-3	-4	-5	-6	-7
1	A	-1	-1	-2	0	-1	-2	-3	-4
2	A	-2	-2	-2	0	2	1	0	-1
3	U	-3	-3	-3	-1	1	1	3	2
4	G	-4	-4	-1	-2	0	3	2	2
5	A	-5	-5	-2	1	0	2	2	1
6	G	-6	-6	-3	0	0	2	1	1
7	C	-7	-4	-4	-1	-1	1	1	3
8	U	-8	-5	-5	-2	-2	0	3	2
9	G	-9	-6	-3	-3	-3	0	2	2

Figure 5.1 The optimal score of sequence alignment between B = `CGAAGUC` and A = `AAUGAGCG` obtained by dynamic programming.



$j \backslash i$		0	1	2	3	4	5	6	7
		–	C	G	A	A	G	U	C
0	–	0	-1	-2	-3	-4	-5	-6	-7
1	A	-1	-1	-2	0	-1	-2	-3	-4
2	A	-2	-2	-2	0	2	1	0	-1
3	U	-3	-3	-3	-1	1	1	3	2
4	G	-4	-4	-1	-2	0	3	2	2
5	A	-5	-5	-2	1	0	2	2	1
6	G	-6	-6	-3	0	0	2	1	1
7	C	-7	-4	-4	-1	-1	1	1	3
8	U	-8	-5	-5	-2	-2	0	3	2
9	G	-9	-6	-3	-3	-3	0	2	2

Figure 5.2 The optimal alignment on $A = \texttt{CGAAGUC}$ and $B = \texttt{AAUGAGCUG}$ obtained by tracing back approach.

the input sequences. Local alignment is widely used on protein comparison for finding local similarity. One may apply Smith-Waterman [115]optimal local alignment algorithm to solve it.

Given the two sequences $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$. Let $A_{q,r} = a_qa_{q+1}\cdots a_r$ and $B_{s,t} = b_sa_{s+1}\cdots a_t$ are *substrings* in A and B respectively, where $1 \leq q \leq r \leq n$ and $1 \leq s \leq t \leq m$. The local alignment is to find highest score, which aligned by all possible $A_{q,r}$ and $B_{s,t}$. To achieve such a goal, we define $V(n, m)$ as the score of an optimal local alignment of sequences A and B. According to the score function, the answer to the problem may be quite variant. The dynamic programming for optimal local alignment is similar to that for optimal global alignment. Take two sequences $R =$ CGAAGUC and $R' =$ AAUGAGCUG as an example, three possible alignments are shown as follows:

```
CGAAGU---C--   --CGAAG-UG   --C-CAAGUC--
--AA-UGAGCUG   AAUG-AGCUG   AA-UGA-G-CUG
```

From either biological or computational side, we have to decide which alignment is better. The Smith-Waterman algorithm [115] use the dynamic programming to find the optimal alignment of all the possible subsequences. Let us briefly review the algorithm in the following.

In the local alignment, if the score become negative, then set the score to zero. The initial values on boundary are given as follows:

$$\begin{array}{rcl} V(0,0) & = & 0 \\ V(i,0) & = & 0, 1 \leq i \leq n \\ V(0,j) & = & 0, 1 \leq j \leq m \end{array}$$

For $i > 0$ and $j > 0$, we have

$$\sigma(i,j) = \begin{cases} 2 & \text{if } \ a_i = b_j \\ -1 & \text{if } \ a_i \neq b_j \quad (\text{including } \ a_i =' -' \ \text{ or } \ b_j =' -') \end{cases} \tag{5.4}$$

$$V(i,j) = \max \begin{cases} 0 \\ V(i-1,j-1) + \sigma(a_i, b_j) \\ V(i-1,j) + \sigma(a_i, -) \\ V(i,j-1) + \sigma(-, b_j) \end{cases} \tag{5.5}$$

In global alignment, we compute the scoring matrix from (0,0) to (n,m) and trace the matrix to find the longest path. In the local alignment, we will find a longest path from (q,s) to (r,t), which obtained from the scoring matrix in $A_{q,r}$ and $B_{s,t}$. The edge whose weight is 0 can be treated as a "jump" path from (0,0) to other possible (q,s). Hence, local alignment can be reduced to finding the longest path from $(0,0)$ through (q,s) to (r,t).

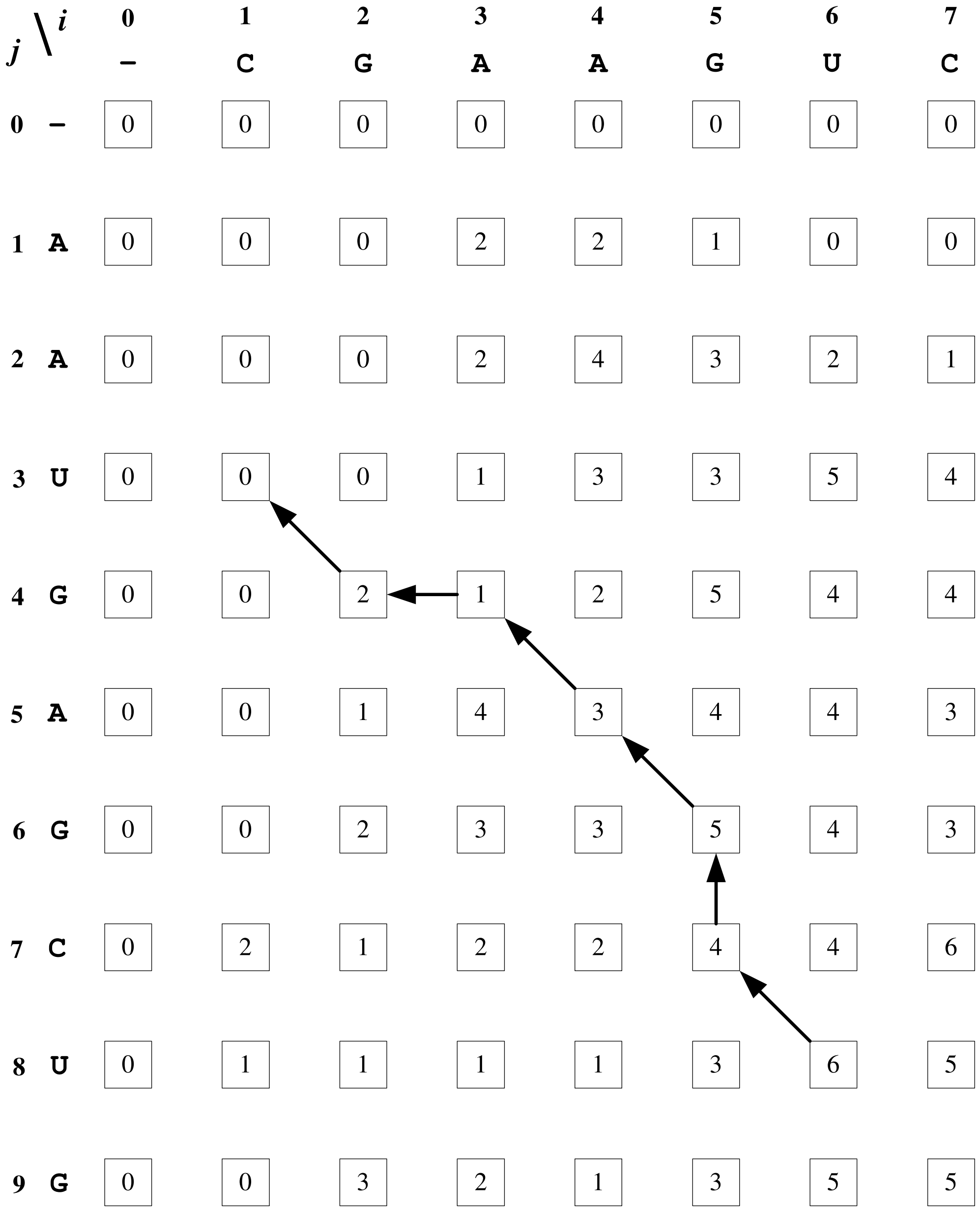
For example, consider two sequences A = `CGAAGUC` and B= `AAUGAGCUG` with pre-defined scoring function σ as defined in Equation 5.4. As shown in Figure 5.3, the optimal local alignment is as follows

```
GAAG-U
G-AGCU
```

The optimal score of the local alignment between A and B is 6.

This optimal local alignment only reports the longest local alignment path. In few years later, the k best non-overlapping paths were proposed [131, 135].

Thus, with this formula, we are able to compute an optimal local alignment generated from the given sore function and gap definition. As the problem changes



$j \backslash i$	0 –	1 C	2 G	3 A	4 A	5 G	6 U	7 C
0 –	0	0	0	0	0	0	0	0
1 A	0	0	0	2	2	1	0	0
2 A	0	0	0	2	4	3	2	1
3 U	0	0	0	1	3	3	5	4
4 G	0	0	2	1	2	5	4	4
5 A	0	0	1	4	3	4	4	3
6 G	0	0	2	3	3	5	4	3
7 C	0	2	1	2	2	4	4	6
8 U	0	1	1	1	1	3	6	5
9 G	0	0	3	2	1	3	5	5

Figure 5.3 The optimal alignment of A=`CGAAGUC` and B=`AAUGAGCUG` by using the dynamic programming.

in different conditions, there are variable score definitions and different constraints. One is the affine gap penalty, which will be discussed in the next section.

5.3 Affine Gap Penalty

In certain applications, we may give different penalty according to different length of gaps. For example, mutations resulting from insertion or deletion of long substrings may be regarded as a single evolutionary event [1, 116, 126]. Therefore, we map this phenomenon to the sequence alignment problem, which is the problem we discuss here.

The penalty in the affine gap model can be divided into two parts, one is the penalty for the gap initialization and the other is for gap extension. In other words, the gap penalty is $G_s + cG_e$ where both G_s and G_e are user-defined constants, with $G_s \geq 0$, $G_e \geq 0$, $c \geq 1$, and c is the number of gaps in the short fragment.

For example, given two sequences A = CGAAGUC and B = AAUGAGCUG, the optimal alignment of these two sequences is as follows:

```
--CGAAG-UC
AAUG-AGCUG
```

There are three gaps in this alignment. An affine gap alignment for the same sequences is considered as follows:

```
C--GAAGUC
AAUGAGCUG
```

We assume the gap penalty as $\sigma(i,-)$ =$\sigma(-,j) = 0$. Suppose $G_s = 4$ and $G_e = 1$, the score of the first alignment is $4\times2-2\times1-2\times(4+1\times1)-1\times(4+2\times1) = -10$ and the score of the second alignment is $3\times2-4\times1-1\times(4+2\times1) = -4$.

Once again, the dynamic programming approach can be used to measure the sequence alignment with affine gap penalty. Here we define the following variable:

1. $V(i,\ j)$ is the score of the optimal alignment of $A = a_1a_2\cdots a_i$ and $B = b_1b_2\cdots b_j$.

2. $V_G(i,j)$ is the score of an optimal sequence alignment of $A = a_1a_2\cdots a_i$ and $B = b_1b_2\cdots b_j$ whose last matched pair is a_i and a_j.

3. $V_F(i,j)$ is the score of the optimal alignment of $A = a_1a_2\cdots a_i$ and $B = b_1b_2\cdots b_j$ whose last matched pair is a_i and a null character.

4. $V_E(i,j)$ is the score of the optimal alignment of $A = a_1a_2\cdots a_i$ and $B = b_1b_2\cdots b_j$ whose last matched pair is a null character and a_j.

Thus, we have base:

$$\begin{aligned}
V(0{,}0) &= 0\\
V(i{,}0) &= -G_s - iG_e,\ 1\le i\le n\\
V(0{,}j) &= -G_s - jG_e,\ 1\le j\le m\\
V_E(i{,}0) &= -\infty\\
V_F(0{,}j) &= -\infty
\end{aligned}$$

Therefore, we could compute the score $V(i,j)$, $1 \le i \le n,\ 1 \le j \le m$, by

$$\begin{aligned}
V(i{,}j) &= max(V_G(i{,}j), V_F(i{,}j), V_E(i{,}j)),\ \text{for i>0,j>0}\\
V_G(i{,}j) &= V(i-1{,}j-1)+\sigma(S_i,T_j),\\
V_F(i{,}j) &= max(V_F(i-1{,}j)-G_e, V(i\text{-}1{,}j)-G_s-G_e)\\
V_E(i{,}j) &= max(V_E(i{,}j-1)-G_e, V(i{,}j-1)-G_s-G_e).
\end{aligned}$$

The equation $V_F(i, j)$ can be understood as choosing the maximum score of adding another null character to an existing gap and starting a new gap.

5.4 Multiple Sequences Alignment

In previous sections, we have already discussed some types of pairwise sequences alignment and introduce some of its applications in biology. With the growth of biological data bases, biologists tend to compare more sequences at the same time. Therefore many tools for multiple sequences alignment have been developed, such as phylogenetic analysis, motif finding, protein structure prediction and so on [63, 80].

In 1989, Lipman presented an optimal alignment for multiple sequence with small number of inputs [83]. Since then, many algorithms have been proposed. These algorithms should be categorized into exact algorithm, progressive algorithm and iterative algorithm. We will introduce these algorithms one by one, according to its category. Table 5.1 list some programs of multiple sequence alignment, which is widely used in biology.

First, for the exact algorithm, Lipman presented a mathematically optimal multiple sequence alignment using dynamic programming [83]. However, it can only be used in small number of short sequences, and the optimal in mathematics may not be feasible in biology. Stoye used a heuristic approach to partition the inputs into several segments, and then applied Lipman's algorithm to get a better result [117].

The second kind of multiple sequence alignment algorithm is progressive algorithm. Generally, progressive algorithms usually take less memory and time than exact algorithms, and can usually be divided into three steps as follows:

Step 1: Compute the distance matrix based on all pairwise sequence alignment.

Table 5.1 Some algorithms of multiple sequence alignment.

Name	Algorithm	Author(s)
Global Multiple Sequence Alignment		
MSA	Exact	Lipman et al. [83]
DCA	Exact	Stoye et al. [117]
OMA	Iterative	Reinert et al. [107]
MULTAL	Progressive	Taylor [118]
MultAlign	Progressive	Barton and Sternberg [20]
Pileup	Progressive	Feng and Doolittle [45]
SAGA	Iterative	Notredame and Higgins [101]
PRPP	Progressive	Gotoh [50]
CLUSTALW	Progressive	Thompson et.al [58, 119]
PRALINE	Progressive	Heringa [57]
Local Multiple Sequence Alignment		
PIMA	Progressive	Smith and Smith [114]
DIALIGN	Iterative	Morgenstern [88]
BLOCK	Iterative	Henikoff [56]
eMOTIF	Iterative	Nevill-Manning et al. [99]
GIBBS	Iterative	Lawrence et al. [76]
HMMER	Iterative	Eddy [43]
MEME	Iterative	Bailey and Elkan [19]

Table 5.2 The ambiguous symbol of IUPAC.

IUPAC Symbol	Meaning	Complement
A	A	T
C	C	G
G	G	C
T	T	A
R	A or G	Y
W	A or T	S
S	C or G	W
Y	C or T	R
K	G or T	M
M	A or C	K
N	A or G or C or T	N

Step 2: Build a guide tree from the distance matrix.

Step 3: Progress the multiple sequence alignment one by one, according to the guide tree.

Hence, we will describe these three steps in more detail and list some differences between progressive algorithms. In Step 1, CLUSTALW uses the approximate algorithm to compute the pairwise score instead of exact match [58,119]. Some algorithms use ambiguous symbols (in Table 5.2) instead of original A, G, C, T in DNA. In Step 2, there are many way to construct the guide tree. MultAlign and Pipeup use UPGMA (*Unweighted Pair-Group Method Using Arithmetic averages*) method to build the tree [20, 45], while CLUSTALW uses the neighbor-joining method to construct the guide tree [58, 119]. In Step 3, many progressive algorithms use dynamic programming to align groups which are constructed by the guide tree.

Finally, we use ACO (*Ant Colony Optimization Algorithm*) to introduce the iterative algorithm. ACO [42] [41] simulates the behavior of ants in the nature world. The computer simulates a group of ants from nest to food, and each ant leaves a special hormone, called pheromone. Pheromone will decrease gradually naturally,

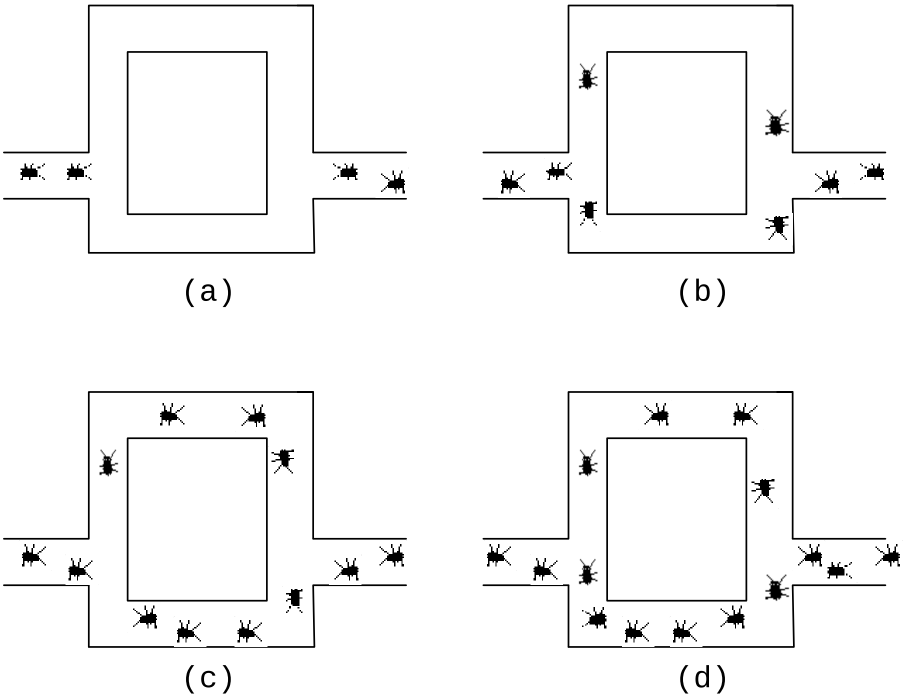


Figure 5.4 (a)The ants move in the divergent point. (b) The ants randomly choose the path. (c) Because the upper path is longer than the lower path, more ants pass through the lower path due to the more pheromone. (d) The ants will follow the pheromone rate to choose the better path, so the shorter path will be taken more times.

avoiding ants from searching a local optimal solution. For a shorter path, ants will pass if through more often than a longer one in a period of time. Finally, most ants will follow the shortest path. There are two prosperities in ACO, one is the position feedback that improves the solution in every generation, the other is the distributed ants that avoid local optimal solution.

The ACO algorithm is as follows:

Step 1: Set parameters and initialize pheromone trails.

Step 2: Each ant completes its trip and constructs a solution according to the trip.

Step 3: Calculate the scores of the all solutions carried by the ants.

Step 4: Based on the score, update the pheromone trails.

Step 5: If the best solution has not been changed after some predefined iterations, terminate the algorithm; otherwise, go to Step 2.

CHAPTER 6

Conclusion

The LCS problem is a classic problem, and it can be applied in many areas. Hence, many algorithms for this problem have been proposed with various approaches and inputs.

Furthermore, we list some open problems here. First, the theoretically fastest algorithm is $mn/\log n$, where m and n are the lengths of the input strings. Therefore, the LCS algorithm should be improved. Second, we can also give some constraints constrains on the LCS problem, including input type, scoring function and output type.

Some open problems on edit distance are listed in this thesis. We can transfer the edit distance problem to another problem by different edit operations. Sorting by transposition and sorting by reversal are still open problems.

In the multiple sequence alignment problems, all three kinds of algorithms are with some advantage and disadvantages. The exact algorithm have highly accuracy. However, it spends too much time and has input limit. The progressive algorithm is more quickly than the exact algorithm, but the strategy to build the guide tree usually biases to some specific inputs. In the alignment algorithm, the optimal alignment in computer may not be the optimum in biology. However, optimum is not far rom the mathematical optimum. Therefore, the *near optimal alignment problem* arises in this area [95, 124].

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] R. A. Abagyan and S. Batalov, "Do aligned sequences share the same fold?," *Journal of Molecular Biology*, Vol. 273, No. 1, pp. 355–368, 1997.

[2] P. Agarwal and D. J. States, "Comparative accuracy of methods for protein sequence similarity search," *Bioinformatics*, Vol. 14, No. 1, pp. 40–47, 1998.

[3] A. V. Aho, D. S. Hirschberg, and J. D. Ullman, "Bounds on the complexity of the longest common subsequence problem," *Journal of the ACM*, Vol. 23, No. 1, pp. 1–12, 1976.

[4] M. H. Albert, A. Golynski, A. M. Hamel, A. López-Ortiz, S. S. Rao, and M. A. Safari, "Longest increasing subsequences in sliding windows," *Theoretical Computer Science*, Vol. 321, No. 2-3, pp. 405–414, 2004.

[5] S. F. Altschul, "A protein alignment scoring system sensitive to all evolutionary distances," *Journal of Molecular Evolution*, Vol. 36, No. 3, pp. 290–300, 1993.

[6] S. F. Altschul and G. Gish, "Local alignment statistics," *Methods Enzymol.*, Vol. 266, pp. 460–480, 1996.

[7] S. F. Altschul, W.Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, Vol. 215, No. 3, pp. 403–410, 1990.

[8] C. E. R. Alves, E. N. Ca'ceres, and S. W. Song, "An all-substrings common subsequence algorithm," *Eletronic Notes in Discrete Mathematics*, Vol. 19, pp. 133–139, 2005.

[9] A. Apostolico, "Improving the wort-case performance of the Hunt-Szymanski strategy for the longest comon subsequence of two string," *Information Processing Letters*, Vol. 23, No. 2, pp. 63–69, 1986.

[10] A. Apostolico, “Remark on the Hsu-Du new algorithm for the longest common subsequence problem,” *Information Processing Letters*, Vol. 25, pp. 235–236q, 1987.

[11] A. Apostolico, S. Browne, and C. Guerra, “Fast linear-space computations of longest common subsequences,” *Theoretical Computer Science*, Vol. 92, pp. 3–17, 1992.

[12] A. Apostolico and C. Guerra, “The longest common subsequence problem revisited,” *Algorithmica*, No. 2, pp. 315–336, 1987.

[13] P. Argo, M. Vingron, and G. Vogt, “Protein sequence comparsion: Methods and significance,” *Protein Engineering*, Vol. 4, No. 4, pp. 375–383, 1991.

[14] P. Argos, “A sensitive procedure to compare amino acid sequences,” *Journal of Molecular Biology*, Vol. 193, No. 2, pp. 385–396, 1987.

[15] T. K. Attwood and D. J. Parry-Smith, *Introduction to Bioinformatics*. Prentice Hall, 1999.

[16] R. Baeza-Yates and G. H. Gonnet, “A new approach to text searching,” *Communications of the ACM*, Vol. 35, No. 10, pp. 74–82, 1992.

[17] V. Bafn and P. Pevzner, “Sorting by transpositions,” *SIAM Journal on Discrete Mathematics*, Vol. 11, No. 2, pp. 224–240, 1998.

[18] V. Bafna and P. Pevzner, “Genome rearrangements and sorting by reversals,” *SIAM Journal of Computing*, Vol. 25, No. 2, pp. 172–289, 1996.

[19] T. L. Bailey and C. Elkan, “The value of prior knowledge in discovering motifs with MEME.,” *Proceeding of International Conference on Intelligent Systems for Molecular Biology*, pp. 21–29, 1995.

[20] G. J. Barton and M. J. E. Sternberg, “A strategy for the rapid multiple alignment of protein sequences: confidence levels from tertiary structure comparisons,” *Journal of Molecular Biology*, Vol. 198, pp. 327–337, 1987.

[21] L. Bergroth, H. Hakonen, and T. Raita, “A survey of longest common subsequence algorithms,” *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE’00)*, pp. 39–48, 2000.

[22] L. Bergroth, H. Hakonen, and T.Raita, “New approximation algorithm for longest common subsequence,” *Proceeding of the 5th of the International Symposium on String Processing information Retrieval*, pp. 32–40, 1998.

[23] S. Bespamyatnikh and M. Segal, "Enumerating longest increasing subsequences and patience sorting," *Information Processing Letters*, Vol. 76, No. 1-2, pp. 7–11, 2000.

[24] P. Bonizzoni and G. D. Vedova, "The complexity of multiple sequence alignment with SP-score that is a metric," *Theoretical Computer Science*, Vol. 259, No. 1, pp. 63–79, 2001.

[25] E. A. Breimer, M. K. Goldberg, and D. T. Lim, "A learning algorithm for the longest common subsequence problem," *Journal of Experimental Algorithmics (JEA)*, Vol. 8, No. 2.1, 2003.

[26] H. Bunke and U. Buhler, "Applications of approximate string matching to 2D shape recognition," *Pattern Recognition*, Vol. 26, No. 12, pp. 1797–1812, 1993.

[27] F. Y. L. Chin and C. K. Poon, "A fast algorithm for computing longest common subsequences of small alphabet size," *Information Processing Letters*, Vol. 13, No. 1, pp. 463–469, 1981.

[28] F. Chin and C. K. Poon, "Performance analysis of some simple heuristics for computing longest common subsequences," *Algorithmica*, Vol. 12, pp. 293–311, 1994.

[29] F. Y. L. Chin, N. L. Ho, T. W. Lam, P. W. H. Wong, and M. Y. chan, "Efficient constrained multiple sequence alignment with performance guarantee," *Journal of Bioinformatics and Computational Biology (JBCB)*, Vol. 3, No. 1, pp. 1–18, 2005.

[30] F. Y. L. Chin, A. D. Santis, A. L. Ferrara, N. L. Ho, and S. K. Kim, "A simple algorithm for the constrained sequence problems," *Information Processing Letters*, Vol. 90, No. 4, pp. 175–179, 2004.

[31] D. A. Christie, "A 3/2-approximation algorithm for sorting by reversals," *in the Proceeding of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 244–252, 1998.

[32] D. A. Christie, "Genome rearrangment problems," *PhD thesis, University of Glasgow, Scotland*, 1998.

[33] V. Chvatal and D. Sankoff, "Longest common subsequences of two random sequences," *Journal of Applied Probability*, Vol. 12, pp. 306–315, 1975.

[34] J. F. Collins, A. F. Coulson, and A. Lyall, "The significance of protein sequence similarities," *Computer Applications in the Biosciences*, Vol. 4, pp. 67–71, 1988.

[35] G. Cormode and S. Muthukrishnan, "The string edit distance matching problem with moves," *In Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA)*, p. 667V676, 2002.

[36] G. Cormode and S. Muthukrishnan, "The string edit distance matching problem with moves," *In Procceding of the 11th Symposium on Discrete Algorithms (SODA'00)*, pp. 197–206, 2000.

[37] M. Crochemore, C. S. Iliopoulos, and Y. J. Pinzon, "Speeding-up Hirschberg and Hunt-szymanski LCS algorithms," *Fundamenta Informaticae*, Vol. 56, No. 1,2, pp. 89–103, 2003.

[38] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid, "A fast and practical bit-vector algorithm for the longest common subsequence problem," *Proceedings of the11th Australasian Workshop On Combinatorial Algorithms*, pp. 75–86, 2000.

[39] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. While, and S. L. Salzberg, "Alignment of whole genomes," *Nucleic Acids Research*, Vol. 27, No. 11, pp. 2369–2376, 1999.

[40] R. Dixon and T. Martin, "Automatic speech and speaker recognition," *IEEE Press*, 1979.

[41] M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, Vol. 1, No. 1, pp. 53–66, 1997.

[42] M. Dorigo, V. Maniezzo, and A. Colorni, "The ant system: Optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics - Part B*, Vol. 26, No. 1, pp. 29–42, 1996.

[43] S. R. Eddy, "Profile hidden markov models," *Bioinformatics*, Vol. 14, pp. 755–763, 1998.

[44] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Ltaliano, "Sparse dynamic promming I: Linear cost functions," *Journal of the ACM*, Vol. 39, pp. 519–545, 1992.

[45] D. F. Feng and R. F. Doolittle, “Progresearchsive sequence alignment as a prerequisite to correct phylogenetic trees,” *Journal of Molecular Evolution*, Vol. 25, pp. 351–360, 1987.

[46] W. M. Fitch and T. F. Smith, “Optimal sequences alignments,” Vol. 80, pp. 1382–1386, 1983.

[47] M. L. Fredman, “On computing the length of longest increasing subsequences,” *Discrete Mathematics*, Vol. 11, No. 1, pp. 29–35, 1975.

[48] M. Gerstein and M. Levitt, “Using iterative dynamic programming to obtain accurate pairwise and multiple alignments of protein structures,” *In Proceedings of the Fourth International Conference on Intelligent Systems in Molecular Biology*, Menlo Park, CA, AAAI Press, 1996.

[49] R. C. Gonzalez and R. E. Woods, *Digital Image Processing.* Addison-Wesley, 1992.

[50] O. Gotoh, “Significant improvement in accuracy of multiple protein sequence alignments by iterative refinements as assessed by reference to structural alignments,” *Journal of Molecular Biology*, Vol. 264, pp. 823–838, 1996.

[51] R. I. Greenberg, “Fast and simple computation of all longest common subsequences,” *Information Processing Letters*, Vol. 1, 2002.

[52] R. I. Greenberg, “Bounds on the number of longest common subsequences,” *Computing Research Repository*, Vol. 2, 2003.

[53] J. Y. Guo and F. K. Hwang, “An almost-linear time and linear space algorithm for the longest common subsequence problem,” *Information Processing Letters*, Vol. 94, No. 3, pp. 131–135, 2005.

[54] D. Gusfield, *Algorithm on Strings,Trees and Sequences.* Cambridge University Press, 1997.

[55] T. Hartman and R. Shamir, “A simpler 1.5-approximation algorithm for sorting by transpositions,” *Combinatorial Pattern Matching 14th Annual Symposium*, pp. 156–169, 2003.

[56] S. Henikoff, “Playing with blocks: some pitfalls of forcing multiple alignments,” *The New Biologist*, Vol. 3, pp. 1148–1154, 1991.

[57] J. Heringa, “Two strategies for sequence comparison: profile-preprocessed and secondary structure-induced multiple alignment,” *Computers and Chemistry*, Vol. 23, pp. 341–364, 1999.

[58] D. G. Higgins and P. M. Sharp, “Clustal: a package for performing multiple sequence alignment on a microcomputer,” *Gene*, Vol. 73, pp. 237–244, 1988.

[59] D. S. Hirschberg, “A linear space algorithm for computing maximal common subsequence,” *Communications of the Association for Computing Machinery*, Vol. 24, pp. 664–675, 1975.

[60] D. S. Hirschberg, “An information theoretic lower bound for the longest common subsequence problem,” *Information Processing Letters*, Vol. 7, No. 1, pp. 40–41, 1978.

[61] D. S. Hirschberg, “Algorithms for the longest common subsequence problem,” *Journal of the ACM*, Vol. 24, pp. 664–675, 1977.

[62] W. J. Hsu and M. W. Du, “New algorithms for the LCS problem,” *Journal of Computer and System Sciences*, Vol. 19, pp. 133–152, 1984.

[63] K.-F. Huang, C.-B. Yang, and K.-T. Tseng, “An efficient algorithm for multiple sequence alignment,” *In the Proceedings of the 19th Workshop on Combinatorial Mathematics and Computation Theory*, Kaoshung, Taiwan, pp. 50–59, 2002.

[64] K.-S. Huang, C.-B. Yang, and K.-T. Tseng, “Fast algorithms for finding the common subsequence of multiple sequences,” *In the Proceedings of 2004 International Computer Symposium*, Taipei, Taiwan, p. 90, 2004.

[65] J. W. Hunt and T. G. Szymanski, “A fast algorithm for computing longest common subsequences,” *Communications of the ACM*, Vol. 20, No. 5, pp. 350–353, 1977.

[66] W. j. Masek and M. S. Peterson, “A faster algorithm computing string edit distance,” *Journal of Computer and System Sciences*, Vol. 20, pp. 18–31, 1980.

[67] T. Jiang, G. Lin, B. Ma, and K. Zhang, “Exact algorithms for the longest common subsequence problem for arc-annotated sequences,” *In Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, No. 1848, pp. 154–165, 2000.

[68] T. Jinge, B. Lin, B. Ma, and K. Zhang, "A general edit distance between RNA structures," *Journal of Computational Biology*, Vol. 9, No. 2, pp. 371–388, 2002.

[69] J. Kececioglu and D. Sankoff, "Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement," *Algorithmica*, Vol. 13, No. 1-2, pp. 180–210, 1995.

[70] P. Kolman, "Approximating reversal distance for strings with bounded number of duplicates in linear time," *30th International Symposium on Mathematical Foundations of Computer Science (MFCS 2005)*, 2005.

[71] S. K. Kumar and C. P. Rangan, "A linear-space algorithm for the LCS problem," *Acta Informatica*, Vol. 24, pp. 353–362, 1987.

[72] G. M. Landau, E. Myers, and M. Ziv-Ukelson, "Two algorithms for LCS consecutive suffix alignment," *15th Combinatorial Pattern Matching Conference (CPM 2004)*, pp. 173–193, 2004.

[73] G. M. Landau, E. W. Myers, and J. P. Schmidt, "Incremental string comparison," *SIAM Journal on Computing*, Vol. 27, No. 2, pp. 557–582, 1998.

[74] G. M. Landau, B. Schieber, and M. Ziv-Ukelson, "Sparse LCS common substring alignment," *Information Processing Letters*, Vol. 88, pp. 259–270, 2003.

[75] G. M. Landau and M. Ziv-Ukelson, "On the common substring alignment problem," *Journal of Algorithms*, Vol. 41, pp. 338–354, 2001.

[76] C. E. Lawrence, S. F. Altschul, M. S. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton, "Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment," *Science*, Vol. 262, pp. 208–214, 1993.

[77] R. C. T. Lee, "Computational biology." http://www.csie.ncnu.edu.tw/, Department of Computer Science and Information Engineering, National Chi-Nan University, 2001.

[78] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Doklady Akademii Nauk SSSR*, Vol. 163, No. 4, pp. 845–848, 1965.

[79] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, Vol. 10, No. 8, pp. 707–710, 1966.

[80] Y.-J. Liao, C.-B. Yang, and S.-H. Shiau, "Motif finding in biological sequences," *In the Proceedings of 2003 Symposium on Digital Life and Internet Technologies*, Tainan, Taiwan, p. 89, 2003.

[81] G.-H. Lin, Z.-Z. Chen, T. Jiang, and J. Wen, "The longest common subsequence problem for sequences with nested arc annotations," *Journal of Computer and System Sciences*, Vol. 65, pp. 465–480, 2002.

[82] G.-H. Lin, B. Ma, and K. Zhang, "Edit distance between two RNA structures," *Annual Conference on Research in Computational Molecular Biology Proceedings of the Fifth Annual International Conference on Computational Biology*, pp. 211–220, 2001.

[83] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu, "A tool for multiple sequence alignment," *Proceedings of the National Academy of Sciences*, Vol. 86, 1989.

[84] M. Maes, "On a cyclic string-to-string correction problem," *Information Processing Letters*, Vol. 35, pp. 73–78, 1990.

[85] D. Maier, "The complexity of some problems on subsequences and supersequences," *Journal of the ACM*, Vol. 25, pp. 322–336, 1978.

[86] A. Marzal and S. Barrachina, "Speeding up the computation of the edit distance for cyclic strings," *In Proceeding of the 15th Internation Conference on Pattern Recoginition*, pp. 895–898, 2000.

[87] J. Modelevsky, "Computer applications in applied genetic engineering," *Advances in Applied Microbiology*, Vol. 30, pp. 169–195, 1984.

[88] B. Morgenstern, A. Dress, and T. Wener, "Multiple DNA and protein sequence based on segment-to-segment comparison," *Proceedings of the National Academy of Sciences*, Vol. 93, pp. 12098–12103, 1996.

[89] D. W. Mount, *Bioinformatics: Sequence and Genome Analysis.* Cold Spring Harbor Laboratory Press, 2001.

[90] S. Muthukrishnan and S. C. Sahinalp, "Approximate nearest neighbors and sequence comparison with block operations," *In the Proceeding of the 32th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 416–424, 2000.

[91] E. W. Myers, "An $O(ND)$ difference algorithm and its variations," *Algorithmica*, No. 1, pp. 251–266, 1986.

[92] E. W. Myers, "An overview of sequence comparison algorithms in molecular biology,"

[93] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM*, Vol. 49, pp. 395–415, 1999.

[94] N. Nakatsu, Y. Kambayashi, and S. Yajima, "A longest common subsequence algorithm suitable for similiar texts," *Acat Informatica*, Vol. 18, pp. 171–179, 1982.

[95] D. Naor and D. L. Brutlag, "On near-optimal alignments of biological sequences," *Journal of Computational Biology*, Vol. 4, pp. 349–366, 1994.

[96] G. Navarro, "A guided tour to approximate string matching," *Computing Surveys*, Vol. 33, No. 1, pp. 31–88, 2001.

[97] G. Navarro and M. Raffinot, "A bit-parallel approach to suffix automata: Fast extended string matching," *Cominatorial Pattern Matching*, pp. 14–33, 1998.

[98] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, Vol. 48, No. 3, pp. 443–453, 1970.

[99] C. G. Nevill-Manning, T. D. Wu, and Brutlag, "Highly specific protein sequence motifs for genome analysis," *Proceeding of the National Academy of Sciences of the United States of America*, Vol. 95, pp. 5865–5871, 1998.

[100] F. Nicolas and E. Rivals, "Longest common subsequence problem for unoriented and cyclic strings," *Laboratoire d'Informatique, Robotique et Microe'lectronique de Montpellier (LIRMM).*

[101] C. Notredame and D. G. Higgins, "SAGA: sequence alignment by genetic algorithm," *Nucleic Acids Research*, Vol. 24, No. 8, pp. 1515–1524, 1996.

[102] M. S. Paterson and V. Dancik, "Upper bonds for the expected length of a longest common subsequence of two binary sequences," *11th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*, Vol. 6, pp. 669–678, 1994.

[103] M. S. Paterson and V. Dancik, "Upper bonds for the expected length of a longest common subsequence of two binary sequences," *Random Structures and Algorithms*, Vol. 6, pp. 449–458, 1995.

[104] C.-L. Peng, "An approach for solving the constrained longest common subsequence problem," *M.S thesis,University of Sun Yat-sen, Kaohsiung, Taiwan*, 2003.

[105] Y.-H. Peng and C.-B. Yang, "The comparison of RNA secondary structures with nested arc-annotation," *In the Proceedings of of the 22nd Workshop on Combinatorial Mathematics and Computation Theory*, Tainan, Taiwan, pp. 107–115, 2005.

[106] D. R. Powell, L. Allison, and T. I. Dix, "A versatile divide and conquer technique for optimal string alignment," *Information Processing Letters*, Vol. 70, pp. 127–139, 1999.

[107] K. Reinert, J. Stoye, and T. Will, "An iterative method for faster sum-of-pair multiple sequence alignment," *Bioinformatics*, Vol. 16, No. 9, pp. 808–814, 2000.

[108] C. Rick, "Simple and fast linear space computation of longest common subsequences," *Information Processing Letters*, Vol. 75, pp. 275–281, 2000.

[109] D. Sankoff and J. Kruskal, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.

[110] C. Schensted, "Longest increasing and decreasing subsequences," *Canadian Journal of Mathematics*, Vol. 13, pp. 179–191, 1961.

[111] P. H. Sellers, "An algorithm for the distance between two finite sequences," *Journal of Combinatoric Theory*, Vol. 16, pp. 253–258, 1974.

[112] J. Setubal and J. Meidanis, *Introduction to Computational Molecular Biology.* PWS Publishing Company, Boston, second ed., 1997.

[113] D. Shapira and J. A. Storer, "Edit distance with move operations," *In Proceedings of the 13th Symposium on Combinatorial Pattern Matching (CPM)*, pp. 85–98, 2002.

[114] R. D. Smith and T. F. Smith, "Pattern-induced multi-sequence alignment (PIMA) algorithm employing secondary structure-dependent gap penalties for use in comparative modelling," *Protein Engineering*, Vol. 5, No. 1, pp. 35–41, 1992.

[115] T. F. Smith and M. S. Waterman, "Comparison of biosequences," *Advances in Applied Mathematics*, Vol. 2, pp. 482–489, 1981.

[116] T. F. Smith, M. S. Waterman, and C. Burks, "The statisical distribution of nucleic acid similarities," *Nucleic Acids Research*, pp. 645–656, 1985.

[117] J. Stoye, V. Moulton, and A. W. Dresearchs, "DCA: An efficient implementation of the Divide-and-Conquer approach to simultaneous multiple sequence alignment," *Computer Applications in Biological Science*, Vol. 13, pp. 625–626, 1997.

[118] W. R. Taylor, "A flexible method to align large numbers of biological sequences," *Journal of Molecular Evolution*, Vol. 28, pp. 161–169, 1988.

[119] J. Thompson, D. Higgins, and T. Gibson, "CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting positionspecific gap penalties and weight matrix choice," *Nucleic Acids Research*, Vol. 22, pp. 4673–4690, 1994.

[120] J. D. Thompson, F. Plewniak, and O. Poch, "A comprehensive comparision of multiple sequence alignment programs," *Nucleic Acids Research*, Vol. 27, No. 13, pp. 2682–2690, 1999.

[121] H. Touzet, "Tree edit distance with gaps," *Information Processing Letters*, Vol. 85, pp. 123–129, 2003.

[122] W. H. Tsai and S. S. Yu, "Attributed string matching with mering for shape recognition," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, Vol. 7, pp. 453–462, 1985.

[123] Y.-T. Tsai, "The constrained longest common subsequence problem," *Information Processing Letters*, Vol. 88, pp. 173–176, 2003.

[124] K.-T. Tseng, C.-B. Yang, and K.-S. Huang, "The better alignment among output alignments," *In Proceedings of the 2005 International Conference on Mathematics and Engineering Techniques in Medecine and Biological Sciences*, Las Vegas, Nevada, USA, pp. 31–37, 2005.

[125] E. Ukkonen, "Algorithms for approximate string matching," *Information and Control*, Vol. 64, pp. 100–118, 1985.

[126] M. Vingron and M. S. Waterman, "Sequence alignment and penalty choice: Review of concepts, case studies and implications," *Journal of Molecular Biology*, Vol. 235, pp. 1–12, 1994.

[127] T. Vintsyuk, "Speech discrcimination by dynamic programming," *Cybernetics*, Vol. 4, pp. 52–58, 1968.

[128] R. A. Wagner, “On the complexity of the extended string-to-string correction problem,” *Annual ACM Symposium on Theory of Computing Proceedings of seventh annual ACM symposium on Theory of computing*, pp. 218–223, 1975.

[129] R. A. Wagner and M. J. Fischer, “The string-to-string correction problem,” *Journal of the ACM*, Vol. 21, No. 1, pp. 168–173, 1974.

[130] M. E. T. Walter, Z. Dias, and J. Meidanis, “A new approach for approximating the tranposition distance,” *In the Proceedings of the Seventh International Symposium on String Processing Information Retrieval*, pp. 199–208, 2000.

[131] M. S. Waterman and M. Eggert, “A new algorithm for best subsequence alignments with application to tRNA-tRNA comparisons,” *Journal of Molecular Biology*, Vol. 197, No. 2, pp. 723–728, 1987.

[132] C. K. Wong and A. K. Chandra, “Bounds of the string editing problem,” *Journal of the ACM*, Vol. 23, pp. 13–16, 1976.

[133] M.-Y. Wu, “RNA secondary structure alignment,” *M.S thesis,University of Sun Yat-sen, Kaohsiung, Taiwan*, 2003.

[134] S. WU, U. Manber, G. Myers, and W. Miller, “An O(NP) sequence comparison algorithm,” *Information Processing Letters*, Vol. 35, pp. 317–323, 1990.

[135] X.Huang, R. Hardison, and W. Miller, “A space-efficient algorithm for local similarities.,” *Computer Applications in Biosciences*, Vol. 6, pp. 373–381, 1990.

[136] C. B. Yang and R. C. T. Lee, “Systolic algorithm for the longest common subsequence problem,” *Journal of the Chinese Institue of Engineers*, Vol. 10, No. 6, pp. 691–699, 1987.

[137] I.-H. Yang, C.-P. Huang, and K.-M. Chao, “A fast algorithm for computing a longest common increasing subsequence,” *Information Processing Letters*, 2004.

[138] H. Zhang, “Alignent of BLAST high-scoring segment pairs based on the longest increasing subsequence algorithm,” *Bioinformatics*, Vol. 19, No. 11, pp. 1391–1396, 2003.

[139] J. Zhu, J. S. Liu, and C. E. Lawrence, “Bayesian adaptive sequence alignment algorithms,” *Bioinformatics*, Vol. 14, pp. 25–39, 1998.