

# Algorithms for the Constrained Longest Common Subsequence Problems

Abdullah N. Arslan<sup>1</sup> and Ömer Egecioğlu<sup>2\*</sup>

<sup>1</sup> Department of Computer Science  
University of Vermont  
Burlington, VT 05405, USA  
e-mail: aarslan@cs.uvm.edu

<sup>2</sup> Department of Computer Science  
University of California, Santa Barbara  
Santa Barbara, CA 93106, USA  
e-mail: omer@cs.ucsb.edu

**Abstract.** Given strings  $S_1, S_2$ , and  $P$ , the constrained longest common subsequence problem for  $S_1$  and  $S_2$  with respect to  $P$  is to find a longest common subsequence  $lcs$  of  $S_1$  and  $S_2$  such that  $P$  is a subsequence of this  $lcs$ . We present an algorithm which improves the time complexity of the problem from the previously known  $O(rn^2m^2)$  to  $O(rnm)$  where  $r, n$ , and  $m$  are the lengths of  $P, S_1$ , and  $S_2$ , respectively. As a generalization of this, we extend the definition of the problem so that the  $lcs$  sought contains a subsequence whose edit distance from  $P$  is less than a given parameter  $d$ . For the latter problem, we propose an algorithm whose time complexity is  $O(d r n m)$ .

**Keywords:** Longest common subsequence, constrained subsequence, edit distance, dynamic programming.

## 1 Introduction

A subsequence of a string  $S$  is obtained by deleting zero or more symbols of  $S$ . The *longest common subsequence* ( $lcs$ ) problem for two strings is to find a common subsequence in both strings having maximum possible length. The  $lcs$  problem has many applications, and it has been studied extensively, see for example [1, 4, 2, 3, 5, 7]. The problem has a simple dynamic programming formulation. To compute an  $lcs$  between two strings of lengths  $n$ , and  $m$ , we use the *edit graph*. The edit graph is a directed acyclic graph having  $(n+1)(m+1)$  lattice points  $(i, j)$  for  $0 \leq i \leq n$ , and  $0 \leq j \leq m$  as vertices. Vertex  $(0, 0)$  appears at the top-left corner, and the vertex  $(n, m)$  is at the bottom-right corner of this rectangular grid. To vertex  $(i, j)$  there are incoming arcs from its neighbors at  $(i-1, j)$ ,  $(i, j-1)$ , and  $(i-1, j-1)$  which represent, respectively, insert, delete, and either substitute or match operations. The  $lcs$  calculation counts the number of matches on the paths from vertex  $(0, 0)$  to  $(n, m)$ , and the problem aims to maximize this number. The time complexity lower bound

---

\*Work done in part while on sabbatical at Sabanci University, Istanbul, Turkey during 2003-2004.

for the problem is  $\Omega(n^2)$  for  $n \geq m$  if the elementary operations are “equal/unequal”, and the alphabet size is unrestricted [1]. If the alphabet is fixed the best known time complexity is  $O(n^2/\log n)$  when  $n = m$  [5]. A survey of practical *lcs* algorithms can be found in [2].

Given strings  $S_1$ ,  $S_2$ , and  $P$ , the constrained longest common subsequence problem [6] for  $S_1$  and  $S_2$  with respect to  $P$  is to find a longest common subsequence *lcs* of  $S_1$  and  $S_2$  such that  $P$  is a subsequence of this *lcs*. For example, for  $S_1 = \text{bbaba}$ , and  $S_2 = \text{abbaa}$ ,  $\text{bbaa}$  is an (unrestricted) *lcs* for  $S_1$  and  $S_2$ , and  $\text{aba}$  is an *lcs* for  $S_1$  and  $S_2$  with respect to  $P = \text{ab}$ , as shown in Figure 1.

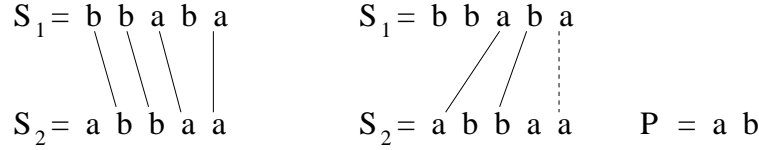


Figure 1: For  $S_1 = \text{bbaba}$ , and  $S_2 = \text{abbaa}$ , the length of an *lcs* is 4 (left). When constrained to contain  $P = \text{ab}$  as a subsequence, the length of an *lcs* drops to 3 (right).

The problem is motivated by practical applications: For example in the computation of the homology of two biological sequences it is important to take into account a common specific or putative structure [6].

Let  $n, m, r$  denote the lengths of the strings  $S_1, S_2$ , and  $P$ , respectively. Tsai [6] gave a dynamic programming formulation for the constrained longest common subsequence problem and a resulting algorithm whose time complexity is  $O(rn^2m^2)$ . In this paper we present a different dynamic programming formulation with which we improve the time complexity of the problem down to  $O(rnm)$ . We achieve improved results by changing the order of the dimensions in the formulation. We also extend the definition of the problem so that the *lcs* sought is forced to contain a subsequence whose *edit distance* from  $P$  is less than a given positive integer parameter  $d$ . For this latter problem we propose an algorithm whose time complexity is  $O(d r n m)$ . Taking  $d = 1$  specializes to the original constrained *lcs* problem as this choice of  $d$  forces the subsequence to contain  $P$  itself. We describe these results in section 2.

## 2 Algorithms

Let  $|S_1| = n$ ,  $|S_2| = m$  with  $n \geq m$ , and  $|P| = r$ . Let  $S[i]$  denote the  $i$ th symbol of string  $S$ . Let  $S[i..j] = S[i]S[i+1] \cdots S[j]$  be the substring of consecutive letters in  $S$  from position  $i$  to position  $j$  inclusive for  $i \leq j$ , and the empty string otherwise.

Denote by  $L_{i,j,k}$  the length of an *lcs* for  $S_1[1..i]$  and  $S_2[1..j]$  with respect to  $P[1..k]$ . This simply means that the common subsequence is constrained to contain  $P$  as a subsequence in turn. We calculate the values  $L_{i,j,k}$  by a dynamic programming formulation. Then  $L_{n,m,r}$  is the length of an *lcs* of  $S_1$  and  $S_2$  containing  $P$  as a subsequence.

**Theorem 1** For all  $i, j, k$ ,  $1 \leq i \leq n$ ,  $1 \leq j \leq m$ ,  $0 \leq k \leq r$ ,  $L_{i,j,k}$  satisfies

$$L_{i,j,k} = \max\{L'_{i,j,k}, L_{i,j-1,k}, L_{i-1,j,k}\} \quad (1)$$

where

$$L'_{i,j,k} = \max\{L''_{i,j,k}, L'''_{i,j,k}\} \quad (2)$$

and

$$L''_{i,j,k} = \begin{cases} 1 + L_{i-1,j-1,k-1} & \text{if } (k = 1 \text{ or } (k > 1 \text{ and } L_{i-1,j-1,k-1} > 0)) \\ & \text{and } S_1[i] = S_2[j] = P[k] \\ 0 & \text{otherwise} \end{cases}$$

$$L'''_{i,j,k} = \begin{cases} 1 + L_{i-1,j-1,k} & \text{if } (k = 0 \text{ or } L_{i-1,j-1,k} > 0) \text{ and } S_1[i] = S_2[j] \\ 0 & \text{otherwise} \end{cases}$$

with boundary conditions  $L_{i,0,k} = 0$ ,  $L_{0,j,k} = 0$ , for all  $i, j, k$ ,  $0 \leq i \leq n$ ,  $0 \leq j \leq m$ ,  $0 \leq k \leq r$ .

**Proof** We prove the correctness of our formulation by induction on  $k$  for all  $i, j$ .

We will consider all possible ways of obtaining an *lcs* with respect to  $P[1..k]$  at any node  $i, j$ . Essentially there are three cases to consider:

1. An *lcs* ending at the node  $(i, j - 1)$  is extended with the horizontal arc  $((i, j - 1), (i, j))$  ending at node  $(i, j)$ ,
2. An *lcs* ending at  $(i - 1, j)$  is extended with the vertical arc  $((i - 1, j), (i, j))$  ending at node  $(i, j)$ ,
3. An *lcs* ending at node  $(i - 1, j - 1)$  is extended with the diagonal arc  $((i - 1, j - 1), (i, j))$  ending at node  $(i, j)$ . In this case we distinguish between subcases depending on whether the diagonal arc is a matching for the given strings along with the pattern, or is a matching for the given strings only at the current indices.

The possible *lcs* extensions referred to in items 1 and 2 above are accounted for by  $L_{i,j-1,k}$  and  $L_{i-1,j,k}$  respectively in the statement of the theorem. The quantities  $L''_{i,j,k}$  and  $L'''_{i,j,k}$  in the statement of the theorem keep track of the two further possibilities described in item 3.

In the base case: when  $k = 0$  (i.e. when  $P$  is the empty string)  $L''_{i,j,k}$  is identically 0. Therefore  $L'_{i,j,k} = L'''_{i,j,k}$  in (2). Since  $k = 0$ , the conjunction in the definition of  $L'''_{i,j,k}$  is always satisfied. We see that putting  $L_{i,j} = L_{i,j,0}$ , (1) becomes

$$L_{i,j} = \max\{L'_{i,j}, L_{i,j-1}, L_{i-1,j}\}$$

where

$$L'_{i,j} = \begin{cases} 1 + L_{i-1,j-1} & \text{if } S_1[i] = S_2[j] \\ 0 & \text{otherwise} \end{cases}$$

which is the classical dynamic programming formulation for the ordinary *lcs* between  $S_1$  and  $S_2$  [7].

Assume that for  $k - 1$  ( $k \geq 1$ ),  $L_{i,j,k-1}$  computed by (1) is the length of an *lcs* for  $S_1[1..i]$  and  $S_2[1..j]$  with respect to  $P[1..k - 1]$  for all  $i, j$  and consider the calculation of  $L_{i,j,k}$  when  $k > 1$ .

We define a *path* at node  $(i, j)$  as a simple path in the edit graph which includes at least one matching arc, starts at node  $(0, 0)$ , and ends at node  $(i, j)$ . A path with

respect to  $P[1..k]$  includes matching diagonal arcs ending at a sequence of  $k \geq 1$  distinct nodes  $(a_1, b_1), (a_2, b_2), \dots, (a_k, b_k)$  such that for all  $\ell$ ,  $1 \leq \ell \leq k$ ,  $S_1[a_\ell] = S_2[b_\ell] = P[\ell]$ . We define  $\#match$  on a path as the number of matches between the symbols of  $S_1$ , and  $S_2$ , not necessarily involving symbols in  $P$ . An *lcs path* with respect to  $P[1..k]$  ending at node  $(i, j)$  is a path with respect to  $P[1..k]$  ending at node  $(i, j)$  with maximum  $\#match$ . Thus  $L_{i,j,k}$  is  $\#match$  on an *lcs path* at node  $(i, j)$  with respect to  $P[1..k]$ . Evidently  $\#match = \#match(i, j, k)$  is a function of the indices  $i, j, k$ . We will omit these parameters when they are clear from the context.

We can extend any *lcs path* with respect to  $P[1..k]$  ending at node  $(i, j - 1)$  with the horizontal arc  $((i, j - 1), (i, j))$  to obtain a path with respect to  $P[1..k]$  ending at node  $(i, j)$ . Such an extension does not change  $\#match$  on the path, and  $L_{i,j,k} \geq L_{i,j-1,k}$ .

Similarly we can extend any *lcs path* with respect to  $P[1..k]$  ending at node  $(i - 1, j)$  with the vertical arc  $((i - 1, j), (i, j))$  to obtain a path with respect to  $P[1..k]$  ending at node  $(i, j)$ . This extension does not change  $\#match$  on the path either, and  $L_{i,j,k} \geq L_{i-1,j,k}$ . Therefore,  $L_{i,j,k} \geq \max\{L_{i,j-1,k}, L_{i-1,j,k}\}$ .

By using a matching arc  $((i - 1, j - 1), (i, j))$ , we can obtain paths with respect to  $P[1..k]$  at node  $(i, j)$  by extending *lcs paths* with either respect to  $P[1..k - 1]$ , or with respect to  $P[1..k]$  ending at node  $(i - 1, j - 1)$ . These two possibilities are accounted for by  $L''_{i,j,k}$  and  $L'''_{i,j,k}$  in the dynamic programming formulation, respectively.

First consider *lcs paths* with respect to  $P[1..k - 1]$  ending at node  $(i - 1, j - 1)$ . We will show that  $L''_{i,j,k}$  stores the maximum  $\#match$  on paths obtained at node  $(i, j)$  by extending these paths.

If  $S_1[i] = S_2[j] = P[k]$  then: If  $k = 1$  then this is the first time the letter  $P[1]$  appears as a matching arc on a path ending at node  $(i, j)$  since we are considering *lcs paths* with respect to  $P[1..k - 1]$  ending at node  $(i - 1, j - 1)$  and  $S_1[i] = S_2[j] = P[1]$ . Therefore, the *lcs length* relative to  $P[1]$  at  $(i, j)$  is  $L''_{i,j,1} = 1 + L_{i-1,j-1,0}$ , which is one more than the length of an ordinary *lcs* between  $S_1[1..i - 1]$  and  $S_2[1..j - 1]$ . If  $k > 1$  and if there is an *lcs path* with respect to  $P[1..k - 1]$  ending at node  $(i - 1, j - 1)$  (i.e. if  $L_{i-1,j-1,k-1} > 0$ ) then we can extend this path with a new match, and  $\#match$  in the resulting path ending at node  $(i, j)$  becomes  $L''_{i,j,k} = 1 + L_{i-1,j-1,k-1}$ .

Next we consider *lcs paths* with respect to  $P[1..k]$  ending at node  $(i - 1, j - 1)$ . We will show that  $L'''_{i,j,k}$  stores the maximum  $\#match$  on paths obtained at node  $(i, j)$  by extending these paths.

If  $S_1[i] = S_2[j]$  then: Since the  $k = 0$  case is considered earlier in the base case of the induction, we only consider the case when  $k > 1$ . If there is an *lcs path* with respect to  $P[1..k]$  ending at node  $(i - 1, j - 1)$  (i.e. if  $L_{i-1,j-1,k} > 0$ ) then we can extend this path by adding a new match (which does not involve  $P$ ), and  $\#match$  in the resulting path relative to  $P[1..k]$  ending at node  $(i, j)$  becomes  $L'''_{i,j,k} = 1 + L_{i-1,j-1,k}$ .

After setting  $L'_{i,j,k} = \max\{L''_{i,j,k}, L'''_{i,j,k}\}$ , the quantity  $L'_{i,j,k}$  is equal to the maximum  $\#match$  on paths with respect to  $P[1..k]$  ending at node  $(i, j)$  ending with the arc  $((i - 1, j - 1), (i, j))$ . If there is no such path then  $L'_{i,j,k} = 0$ . Therefore  $L_{i,j,k} \geq \max\{L'_{i,j,k}, L_{i,j-1,k}, L_{i-1,j,k}\}$ .

From all possible *lcs paths* ending at neighboring nodes of  $(i, j)$  we can find their extensions ending at node  $(i, j)$ , and we can obtain an *lcs path* ending at node  $(i, j)$  with respect to  $P[1..k]$  for all  $k$ . We calculate, and store in  $L_{i,j,k}$  such *lcs lengths*. Now consider the structure of an *lcs path* with respect to  $P[1..k]$  ending at node  $(i, j)$ . As

	<i>b</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>
<i>a</i>	0	0	1	1	1
<i>b</i>	1	1	1	2	2
<i>b</i>	1	2	2	2	2
<i>a</i>	1	2	3	3	3
<i>a</i>	1	2	3	3	4

$k = 0$

	<i>b</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>
<i>a</i>	0	0	1	1	1
<i>b</i>	0	0	1	2	2
<i>b</i>	0	0	1	2	2
<i>a</i>	0	0	3	3	3
<i>a</i>	0	0	3	3	4

$k = 1$

	<i>b</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>
<i>a</i>	0	0	0	0	0
<i>b</i>	0	0	0	2	2
<i>b</i>	0	0	0	2	2
<i>a</i>	0	0	0	2	3
<i>a</i>	0	0	0	2	3

$k = 2$

Figure 2: For  $S_1 = \text{abbaa}$ ,  $S_2 = \text{bbaba}$ , and  $P = \text{ab}$ , the tables of values  $L_{i,j,k}$  = the length of an *lcs* for  $S_1[1..i]$  and  $S_2[1..j]$  with respect to  $P[1..k]$ .

typical in dynamic programming formulations, we consider the possible cases of the last arc on such a path to obtain  $L_{i,j,k} \leq \max\{L'_{i,j,k}, L_{i,j-1,k}, L_{i-1,j,k}\}$  which proves the theorem. •

*Example:* Figure 2 shows the contents of the dynamic programming tables for  $S_1 = \text{bbaba}$ , and  $S_2 = \text{abbaa}$ , and  $P = \text{ab}$  for  $k = 0, 1, 2$ . For  $k = 0$ , the calculated values are simply the ordinary dynamic programming *lcs* table for  $S_1$  and  $S_2$ .

All  $L_{i,j,k}$  can be computed in  $O(rnm)$  time, using  $O(rm)$  space using the formulation in Theorem 1 by noting that we only need rows  $i-1$ , and  $i$  during the calculations at row  $i$ . If actual *lcs* is desired then we can carry the *lcs* information for each  $k$  along with the calculations. This requires  $O(rnm)$  space. By keeping track, on *lcs* for each  $k$ , of only the match points  $(i', j')$  of  $P[u]$  for all  $u$ ,  $1 \leq u \leq r$ , the space complexity can be reduced to  $O(r^2m)$ . In this case, the *lcs* for  $k = r$  needs to be recovered using ordinary *lcs* computations to connect the consecutive match points.

*Remark:* Space complexity can further be improved by applying a technique used in unconstrained *lcs* computation [3]. We can compute, instead of the entire *lcs* for each  $k$ , middle vertex  $(n/2, j)$  (assume for simplicity that  $n$  is even) at which an *lcs* with respect to  $P[1..k]$  passes. This can be done in  $O(rm)$  space, and we can compute for all  $k$  the *lcs* length  $L_{n/2,j,k}$  from vertex  $(0, 0)$  to vertex  $(n/2, j)$ , and *lcs* length from  $(n/2, j)$  to  $(n, m)$ . The latter is done in the reverse edit graph by calculating *lcs* from  $(n, m)$  to  $(n/2, j)$ , hence we denote it by  $L_{n/2,j,l}^{\text{reverse}}$  for  $0 \leq \ell \leq k$ . Then for every  $k$ ,

$$\max_{j, 0 \leq \ell \leq k} L_{n/2,j,l} + L_{n/2,j,k-\ell}^{\text{reverse}}$$

is the *lcs* length for  $k$ , and it identifies a middle vertex. After the middle vertex  $(n/2, j)$  on *lcs* for every  $k$  is found, the problem of finding the *lcs* from  $(0, 0)$  to  $(n, m)$  can be solved in two parts: find the *lcs* from  $(0, 0)$  to  $(n/2, j)$ , and find the *lcs* from  $(n/2, j)$  to  $(n, m)$  for all  $k$ . These two subproblems can be solved recursively by finding the middle points. This way *lcs* can be obtained using  $O(rm)$  space. The time complexity remains  $O(rnm)$  because  $n$  is halved each time, and the area (in terms of number of vertices) covered in the edit graph is  $O(nm)$ , and at each vertex the total time spent is  $O(r)$ .

Next we propose a generalization of the constrained longest common subsequence problem. Given strings  $S_1, S_2$ , and  $P$ , and a positive integer  $d$  the *edit distance*

constrained longest common subsequence problem for  $S_1$  and  $S_2$  with respect to string  $P$ , and distance  $d$  is to find a longest common subsequence  $lcs$  of  $S_1$  and  $S_2$  such that this  $lcs$  has a subsequence whose edit distance from  $P$  is smaller than  $d$ . Edit distance between two strings is the minimum number of edit operations required to transform one string to the other. The edit operations are insert, delete, and substitute.

Let  $L_{i,j,k,t}$  be the length of an  $lcs$  for  $S_1[1..i]$  and  $S_2[1..j]$  such that the common subsequence contains a subsequence whose edit distance from  $P[1..k]$  is exactly  $t$ .

*Example:* Suppose  $S_1 = \text{bbaba}$ ,  $S_2 = \text{abbaa}$  and  $P = \text{ab}$ . We have calculated before that the length of an  $lcs$  for  $S_1$  and  $S_2$  relative to  $P$  is 3. Thus  $L_{5,5,2,0} = 3$ . On the other hand the  $lcs$   $\text{bbaa}$  of  $S_1$  and  $S_2$  contains the subsequence  $\text{a}$ , which is edit distance 1 away from  $P$ . Therefore  $L_{5,5,2,1} = 4$ .

We calculate all  $L_{i,j,k,t}$  by a dynamic programming formulation. Optimal value of the edit distance constrained  $lcs$  problem is  $\max_{0 \leq t < d} L_{n,m,r,t}$ .

**Theorem 2** For all  $i, j, k, t$ ,  $1 \leq i \leq n$ ,  $1 \leq j \leq m$ ,  $0 \leq k \leq r$ ,  $0 \leq t < d$ ,  $L_{i,j,k,t}$  satisfies

$$L_{i,j,k,t} = \max\{L'_{i,j,k,t}, L_{i,j-1,k,t}, L_{i-1,j,k,t}\} \quad (3)$$

where

$$L'_{i,j,k,t} = \max\{L''_{i,j,k,t}, L'''_{i,j,k,t}, L''''_{i,j,k,t}\} \quad (4)$$

where

$$L''_{i,j,k,t} = \begin{cases} 1 + L_{i-1,j-1,k-1,t} & \text{if } ((k = 1 \text{ and } t = 0) \text{ or} \\ & (k > 1 \text{ and } L_{i-1,j-1,k-1,t} > 0)) \\ & \text{and } S_1[i] = S_2[j] = P[k] \\ 0 & \text{otherwise} \end{cases}$$

$$L'''_{i,j,k,t} = \begin{cases} 1 + L_{i-1,j-1,0,0} & \text{if } (k = 0 \text{ and } t = 1) \text{ and } S_1[i] = S_2[j] \\ 1 + L_{i-1,j-1,k,t} & \text{else if } (k = 0 \text{ or } L_{i-1,j-1,k,t} > 0) \\ & \text{and } S_1[i] = S_2[j] \\ 0 & \text{otherwise} \end{cases}$$

$$L''''_{i,j,k,t} = \max\{D_{i,j,k,t}, X_{i,j,k,t}, I_{i,j,k,t}\} \quad (5)$$

where

$$D_{i,j,k,t} = \begin{cases} L_{i,j,k-1,t-1} & \text{if } t \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

$$X_{i,j,k,t} = \begin{cases} L_{i,j,k-1,t-1} & \text{if } t \geq 1 \text{ and } S_1[i] = S_2[j] \\ 0 & \text{otherwise} \end{cases}$$

$$I_{i,j,k,t} = \begin{cases} L_{i,j,k,t-1} & \text{if } t \geq 1 \text{ and } S_1[i] = S_2[j] \\ 0 & \text{otherwise} \end{cases}$$

with boundary conditions  $L_{i,0,k,0} = 0$ ,  $L_{0,j,k,0} = 0$ , for all  $i, j, k$ ,  $0 \leq i \leq n$ ,  $0 \leq j \leq m$ ,  $0 \leq k \leq r$ .



**Proof** We claim that  $L_{i,j,k,t}$  is the optimum length for any  $lcs$  for  $S_1[1..i]$  and  $S_2[1..j]$  such that the  $lcs$  contains a subsequence whose edit distance is  $t$  from  $P[1..k]$ . We prove the correctness of our formulation by induction on  $t$  for all  $i, j, k$ .

In the base case: when  $t = 0$  the formulation becomes the same formulation as in Theorem 1, since now the  $lcs$  is required to contain  $P$  itself as a subsequence. Therefore, the correctness of this case follows from Theorem 1.

Assume that for  $t - 1$  ( $t \geq 1$ ), for all  $i, j, k$ ,  $L_{i,j,k,t-1}$  is the optimum length for any  $lcs$  for  $S_1[1..i]$  and  $S_2[1..j]$  such that the  $lcs$  contains a subsequence whose edit distance is  $t$  from  $P[1..k]$ . Consider the calculation of  $L_{i,j,k,t}$  for all  $i, j, k$  when  $t > 1$ .

Our formulation uses the following observation: Let  $cs$  be a subsequence of an  $lcs$  of  $S_1$  and  $S_2$ . The minimum edit distance between  $cs$  and  $P$  can be calculated using insert, delete, and substitute operations in  $P$ , and using no operations in  $cs$ . To see this consider the edit operations between the symbols in  $cs$ , and in  $P$ . If an edit distance calculation deletes a symbol  $s$  in  $cs$ , we can instead insert the symbol  $s$  in  $P$ ; if a minimum edit distance calculation inserts a symbol  $s$  in  $cs$ , we can instead delete the symbol  $s$  in  $P$ ; and if a minimum edit distance calculation substitutes a symbol  $s'$  for  $s$  in  $cs$ , we can instead substitute a symbol  $s$  for  $s'$  in  $P$  to obtain the same edit distance.

We define an *edit path* at node  $(i, j)$  at distance  $t$  from  $P[1..k]$  as a simple path from node  $(0, 0)$  to node  $(i, j)$ , which includes a sequence of  $l \geq 1$  distinct nodes  $(a_1, b_1), (a_2, b_2), \dots, (a_l, b_l)$  such that the edit distance between the string  $S_1[a_1]S_2[a_2] \dots S_1[a_l]$  ( $= S_2[b_1] S_2[b_2] \dots S_2[b_l]$ ), and  $P[1..k]$  is exactly  $t$ . We define  $\#match$  on a given edit path to node  $(i, j)$  as the number of matching diagonal arcs on the path between the symbols in  $S_1[1..i]$ , and the symbols in  $S_2[1..j]$ , not necessarily involving matches in  $P$ . An optimal edit path at node  $(i, j)$  at distance  $t$  from  $P[1..k]$  is an edit path at node  $(i, j)$  at distance  $t$  from  $P[1..k]$  with maximum  $\#match$ . Thus  $L_{i,j,k,t}$  is  $\#match$  on an optimal edit path at node  $(i, j)$  at distance  $t$  from  $P[1..k]$ . In this case,  $\#match = \#match(i, j, k, t)$  is a function of the indices  $i, j, k, t$ , but we omit these parameters when they are clear from the context.

We can extend any optimal edit path at node  $(i, j - 1)$  at distance  $t$  from  $P[1..k]$  with the horizontal arc  $((i, j - 1), (i, j))$  to obtain an edit path at node  $(i, j)$  at distance  $t$  from  $P[1..k]$ . Such an extension does not change  $\#match$  on the resulting edit path, and  $L_{i,j,k,t} \geq L_{i,j-1,k,t}$ .

Similarly we can extend any optimal edit path at node  $(i - 1, j)$  at distance  $t$  from  $P[1..k]$  with the vertical arc  $((i - 1, j), (i, j))$  to obtain an edit path at node  $(i, j)$  at distance  $t$  from  $P[1..k]$ . This extension does not change  $\#match$  on the resulting edit path, and  $L_{i,j,k,t} \geq L_{i-1,j,k,t}$ . Therefore,  $L_{i,j,k,t} \geq \max\{L_{i,j-1,k,t}, L_{i-1,j,k,t}\}$ .

By using a matching arc  $((i - 1, j - 1), (i, j))$ , we can obtain edit paths at node  $(i, j)$  at distance  $t$  from  $P[1..k]$  by extending optimal edit paths at node  $(i - 1, j - 1)$  at distance  $t - 1$ , or  $t$  from  $P[1..k - 1]$ , or  $P[1..k]$ .

First consider optimal edit paths at node  $(i - 1, j - 1)$  at distance  $t$  from  $P[1..k - 1]$ . We will show that  $L''_{i,j,k,t}$  stores the maximum  $\#match$  obtained at node  $(i, j)$  by extending these edit paths.

If  $S_1[i] = S_2[j] = P[k]$  then: We do not need to consider the case when  $k = 1$  and  $t = 0$  since  $t = 0$  case is considered in the base case of the induction. If  $k > 1$  and if there is an optimal edit path at node  $(i, j)$  at distance  $t$  from  $P[1..k]$  (i.e. if

$L_{i-1,j-1,k-1,t} > 0$ ) then we can extend this edit path with a new match, and  $\#match$  on the resulting edit path at node  $(i, j)$  at distance  $t$  from  $P[1..k]$  becomes  $L''_{i,j,k,t} = 1 + L_{i-1,j-1,k-1,t}$ .

Next we consider optimal edit paths at node  $(i-1, j-1)$  at distance  $t$  from  $P[1..k]$ . We will show that  $L'''_{i,j,k,t}$  stores the maximum  $\#match$  obtained at node  $(i, j)$  by extending these edit paths.

If  $S_1[i] = S_2[j]$  then: If  $k = 0$  and  $t = 1$  then: We can extend an *lcs* path ending at node  $(i-1, j-1)$  with respect to  $P[1..k]$  with a match. In this case,  $\#match$  in the resulting edit path is one more than  $L_{i-1,j-1,0,0}$ . Therefore,  $L'''_{i,j,0,1} = 1 + L_{i-1,j-1,0,0}$ . Otherwise if  $k = 0$  then we can extend an optimal edit path at node  $(i-1, j-1)$  at distance  $t$  from  $P[1..k]$  with a match, and  $\#match$  on the resulting edit path is  $L'''_{i,j,k,t} = 1 + L_{i-1,j-1,k,t}$ .

Any edit path at node  $(i, j)$  at distance  $t-1$  from  $P[1..k-1]$ , or  $P[1..k]$  can be modified by applying an edit operation in  $P$ . We can modify an edit path at node  $(i, j)$  at distance  $t-1$  from  $P[1..k-1]$  by deleting  $P[k]$ . Then on the resulting edit path  $\#match$  remains the same, and the distance increases by 1. Therefore, we set  $D_{i,j,k,t} = L_{i,j,k-1,t-1}$ , and take it into account in  $L''''_{i,j,k,t}$ . We can modify an edit path at node  $(i, j)$  at distance  $t-1$  from  $P[1..k-1]$  by substituting  $S_1[i] = S_2[j]$  for  $P[k]$ . Then on the resulting edit path  $\#match$  remains the same, and the distance increases by 1. Therefore, we set  $X_{i,j,k,t} = L_{i,j,k-1,t-1}$  if  $S_1[i] = S_2[j]$ , and take it into account in  $L''''_{i,j,k,t}$ . We can also modify an edit path at node  $(i, j)$  at distance  $t-1$  from  $P[1..k]$  by inserting  $S_1[i] = S_2[j]$  in  $P$  after position  $k$ . Then on the resulting edit path  $\#match$  remains the same, and the distance increases by 1. Therefore, we set  $I_{i,j,k,t} = L_{i,j,k,t-1}$  if  $S_1[i] = S_2[j]$ , and take it into account in  $L''''_{i,j,k,t}$ . Combining all these  $L''''_{i,j,k,t} = \max\{D_{i,j,k,t}, X_{i,j,k,t}, I_{i,j,k,t}\}$ .

After setting  $L'_{i,j,k,t} = \max\{L''_{i,j,k,t}, L'''_{i,j,k,t}, L''''_{i,j,k,t}\}$ ,  $L'_{i,j,k,t}$  stores the maximum  $\#match$  on edit paths at node  $(i, j)$  at distance  $t$  from  $P[1..k]$  whose last arc is  $((i-1, j-1), (i, j))$ . If there is no such edit path then  $L'_{i,j,k,t} = 0$ .

From all possible optimal edit paths at neighboring nodes of  $(i, j)$  we can obtain their extensions ending at node  $(i, j)$ , and we can find an optimal edit path at node  $(i, j)$  at distance  $t$  from  $P[1..k]$  for all  $k, t$ . We calculate, and store in  $L_{i,j,k,t}$  maximum  $\#match$  in such optimal edit paths. Considering the possible cases of the last arc on an optimal edit path at node  $(i, j)$  at distance  $t$  from  $P[1..k]$  we also have  $L_{i,j,k,t} \leq \max\{L'_{i,j,k,t}, L_{i,j-1,k,t}, L_{i-1,j,k,t}\}$ . This concludes the proof of the theorem. •

All  $L_{n,m,r,t}$  for  $t = 0, 1, \dots, d-1$  can be computed in  $O(dnrm)$  time, and using  $O(drm)$  space using the formulation in Theorem 2 by noting that we only need rows  $i-1$ , and  $i$  during the calculations at row  $i$ . If an actual optimal edit path is desired then we can carry the edit path information for every  $k$  and  $t$  along with the calculations. This requires  $O(dnrm)$  space since edit paths can be of length  $O(n)$ .

If we store match points (where the symbols of  $S_1$ ,  $S_2$ , and  $P$  match) on these edit paths then we can reduce the required space to  $O(dr^2m)$ . In this case, the optimal edit path of the problem needs to be recovered using ordinary *lcs* computations to connect the consecutive match points.

*Remark:* Space complexity can further be improved by using the technique we used in our first algorithm. We can compute, instead of the entire edit path for each  $k$ , and  $t$ , a middle vertex  $(n/2, j)$  (assume for simplicity that  $n$  is even) at which an edit path at distance  $t$  from  $P[1..k]$  passes. This can be done in  $O(drm)$  space, and we



can compute for all  $k$ , and  $t$ ,  $\#match$   $L_{n/2,j,l,u}$  on optimal edit path from vertex  $(0, 0)$  to vertex  $(n/2, j)$ , and  $\#match$  on optimal edit path from  $(n/2, j)$  to  $(n, m)$  where  $0 \leq \ell \leq k$ , and  $0 \leq u \leq t$ . The latter, denoted by  $L_{n/2,j,k-\ell,t-u}^{reverse}$ , can be calculated in the reverse edit graph. Then for all  $k, t$ ,

$$\max_{j, 0 \leq \ell \leq k, 0 \leq u \leq t} L_{n/2,j,l,u} + L_{n/2,j,k-\ell,t-u}^{reverse}$$

is the optimum  $\#match$  for  $k, t$ , and it identifies a middle vertex. After the middle vertex  $(n/2, j)$  on optimal edit path for every  $k, t$  is found, the problem of finding an optimal edit path from  $(0, 0)$  to  $(n, m)$  can be solved in two parts: find an optimal edit path from  $(0, 0)$  to  $(n/2, j)$ , and find an optimal edit path from  $(n/2, j)$  to  $(n, m)$  for all  $k, t$ . These two subproblems can be solved recursively. As a result an optimal edit path can be obtained using  $O(drm)$  space. The time complexity remains  $O(rnm)$  because  $n$  is halved each time, and the area (in terms of number of vertices) covered in the edit graph is  $O(nm)$ , and at each vertex the total time spent is  $O(dr)$ .

### 3 Conclusion

We have improved the time complexity of the constrained *lcs* problem from  $O(rn^2m^2)$  to  $O(rnm)$  where  $n$ , and  $m$  are the lengths of the given strings, and  $r$  is the pattern length. This improvement is achieved by a dynamic programming formulation which is different from what was proposed in [6]. In our formulation, the dimensions are ordered differently. We also extended the problem definition to use edit distances, and presented an  $O(drn)$  time algorithm for the resulting edit distance constrained *lcs* problem.

### References

- [1] A.V. Aho, D.S. Hirschberg, and J.D. Ullman. Bounds on the complexity of the longest common subsequence problem. *J. ACM*, 23(1):1–12, 1976.
- [2] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. *SPIRE, A Coruna, Spain*, pp. 39–48, 2000.
- [3] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of ACM*, 18:341–343, 1975.
- [4] D.S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24:664–675, 1977.
- [5] W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *J. Comput. System Sci.*, 20:18–31, 1980.
- [6] Yin-Te Tsai. The constrained common sequence problem. *Information Processing Letters*, 88:173–176, 2003.
- [7] R.A. Wagner and M.J. Fisher. The string-to-string correction problem. *J. ACM*, 21:168–173, 1974.