International Journal of Foundations of Computer Science
Vol. 16, No. 6 (2005) 1099–1109

World Scientific
www.worldscientific.com

ALGORITHMS FOR THE CONSTRAINED LONGEST COMMON SUBSEQUENCE PROBLEMS*

ABDULLAH N. ARSLAN
Department of Computer Science
University of Vermont
Burlington, VT 05405, USA
aarslan@cs.uvm.edu

and

ÖMER EĞECIOĞLU†
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
omer@cs.ucsb.edu

Received 25 October 2004
Accepted 25 February 2005
Communicated by J. Holub

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 which contains P as a subsequence. 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 $O(drnm)$.

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 having maximum possible length. The *lcs* problem has many applications, and it has been studied extensively, see for example [1, 5, 2, 4, 6, 8].

*A preliminary version of this work was presented at Prague Stringology Conference (PSC'04), Prague, August 2004.

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

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 $0 \le i \le n$, and $0 \le j \le m$ 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. For all i,j, $0 < i \le n$, and $0 < j \le m$, to vertex $(0,j)$ there is an arc from its neighbor at $(0,j-1)$, to vertex $(i,0)$ there is an arc from its neighbor at $(i-1,0)$, and to vertex (i,j) when $i > 0$ and $j > 0$ there are incoming arcs from its neighbors at $(i-1,j),(i,j-1)$, and $(i-1,j-1)$. Each vertex (i,j) corresponds to a pair of prefixes $S_1[1..i]$, and $S_2[1..j]$. Horizontal, vertical, and diagonal arcs represent, respectively, insert, delete, and either substitute or match operations on S_1. The *lcs* length calculation counts the number of matches on the paths from vertex $(0,0)$ to (n,m), and the problem aims to maximize this number. When $n = m$ the time complexity lower bound for the problem is $\Omega(n^2)$ 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 $O(n \,.\, \max\{1, m/\log n\})$ [6]. 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 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* [7]. For example, for $S_1 =$ bbaba, and $S_2 =$ abbaa, bbaa is an (unrestricted) *lcs* for S_1 and S_2, and aba is an *lcs* for S_1 and S_2 with respect to $P =$ ab, as shown in Figure 1.

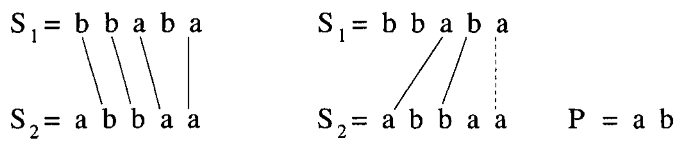

Figure 1: For $S_1 =$ bbaba, and $S_2 =$ abbaa, the length of an *lcs* is 4 (left). When constrained to contain $P =$ ab as a subsequence, the length of an *lcs* drops to 3 (right).

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 [7].

Let n, m, r denote the lengths of the strings S_1, S_2, and P, respectively. For the constrained longest common subsequence problem Tsai [7] gave a dynamic programming formulation that yields an algorithm whose time complexity is $O(rn^2m^2)$. The solution requires *lcs* computations with respect to $P[1..k]$ for all k between all substrings of S_1 and all substrings of S_2. In this paper we present a different dynamic programming formulation with which we improve the time complexity of the problem down to $O(rnm)$. In our solution *lcs* computations are only required between the prefixes of S_1 and S_2. 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 $O(drnm)$. 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 ith 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.

We denote by $L_{i,j,k}$ the length of an *lcs* of $S_1[1..i]$ and $S_2[1..j]$ such that the *lcs* contains $P[1..k]$ as a subsequence. That is, $L_{i,j,k}$ is the optimum value of the constrained *lcs* problem for $S_1[1..i]$, $S_2[1..j]$, and $P[1..k]$. We calculate all $L_{i,j,k}$ 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*

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

where

$$L'_{i,j,k} = \max\{L''_{i,j,k},\ L'''_{i,j,k}\} \tag{2}$$

and

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

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

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 claim that for all $L_{i,j,k}$ defined in (1), $L_{i,j,k}$ is the length of an *lcs* of $S_1[1..i]$ and $S_2[1..j]$ with respect to $P[1..k]$. We prove this by induction. The induction is based on an ordering of (i, j, k). The calculation of $L_{i,j,k}$ uses values $L_{i,j-1,k}$, $L_{i-1,j,k}$, $L_{i-1,j-1,k-1}$, and $L_{i-1,j-1,k}$. Therefore (i, j, k) must come after $(i, j-1, k)$, $(i-1, j, k)$, $(i-1, j-1, k-1)$ and $(i-1, j-1, k)$ in the ordering. This ordering can be generated by three nested loops where k is the loop counter of the outermost loop, and i, and j are loop counters of the inner loops.

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 lengths $L''_{i,j,k}$ and $L'''_{i,j,k}$ in the statement of the theorem keep track of the two further possibilities for lcs lengths described in item 3.

In the base case: when $k = 0$ for all i, j (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

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

which is the classical dynamic programming formulation for the ordinary lcs between S_1 and S_2 [8]. Therefore the base case holds.

Assume by way of induction that $L_{i,j-1,k}$, $L_{i-1,j,k}$, $L_{i-1,j-1,k-1}$, and $L_{i-1,j-1,k}$ defined by (1) are the optimum lengths obtained at the neighboring nodes of (i, j, k) for the corresponding constrained lcs problems. Next we consider the calculation of $L_{i,j,k}$.

For every node (i, j) in the edit graph, we define a *path* at node (i, j) as a simple path which starts at node $(0, 0)$, ends at node (i, j), and which includes at least one matching arc. A path with respect to $P[1..k]$ includes a sequence of matching diagonal arcs ending at $k \geq 1$ distinct nodes $(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)$ such that for all ℓ, $1 \leq \ell \leq k$, $S_1[a_\ell] = S_2[b_\ell] = P[\ell]$. 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 either with 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 with respect 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* on 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* on the resulting path with respect to $P[1..k]$ ending at node (i, j) becomes $L'''_{i,j,k} = 1 + L_{i-1,j-1,k}$.

Since $L'_{i,j,k} = \max\{L''_{i,j,k}, L'''_{i,j,k}\}$ in (2), the value $L'_{i,j,k}$ 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 $P[1..k]$. We calculate, and store in $L_{i,j,k}$ the maximum *#match* on such paths. Now consider the structure of an *lcs* path with respect to $P[1..k]$ ending at node (i, j). As typical in dynamic programming formulations, we consider the possible cases of the last arc on such a path to obtain $L_{i,j,k} \leq \max\{L'_{i,j,k}, L_{i,j-1,k}, L_{i-1,j,k}\}$. Therefore $L_{i,j,k}$ is the length of an *lcs* of $S_1[1..i]$ and $S_2[1..j]$ that contains $P[1..k]$ as a subsequence, and this concludes the proof of the theorem. □

Example: Figure 2 shows the contents of the dynamic programming tables for $S_1 = \texttt{bbaba}$, and $S_2 = \texttt{abbaa}$, and $P = \texttt{ab}$ 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.

	b	b	a	b	a
a	0	0	1	1	1
b	1	1	1	2	2
b	1	2	2	2	2
a	1	2	3	3	3
a	1	2	3	3	4

$k = 0$

	b	b	a	b	a
a	0	0	1	1	1
b	0	0	1	2	2
b	0	0	1	2	2
a	0	0	3	3	3
a	0	0	3	3	4

$k = 1$

	b	b	a	b	a
a	0	0	0	0	0
b	0	0	0	2	2
b	0	0	0	2	2
a	0	0	0	2	3
a	0	0	0	2	3

$k = 2$

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

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 an actual constrained lcs (i.e. an lcs of S_1 and S_2 that contains P as a subsequence) is desired then we can carry the lcs information for each k along with the calculations. The resulting space complexity is $O(rm^2)$ since each lcs of S_1 and S_2 is of length $O(m)$. On any constrained lcs for each k, if we keep track of only the match points (i', j') where $S_1[i'] = S_2[j'] = P[u]$, $1 \leq u \leq k$, then the space complexity can be reduced to $O(r^2 m)$. In this case, a constrained lcs for $k = r$ needs to be recovered using an additional step at the end that performs ordinary lcs computations for S_1 and S_2 to connect the consecutive match points. This step requires $O(m)$ space.

Remark: Space complexity can further be improved by applying a technique used in a linear space unconstrained lcs algorithm [4]. We can compute, instead of an entire lcs for each k, a middle vertex $(n/2, j)$ (assume for simplicity that n is even) at which an lcs path with respect to $P[1..k]$ passes. This can be done in $O(rm)$ space, and we can compute for all k the constrained lcs length $L_{n/2,j,k}$ from vertex $(0,0)$ to vertex $(n/2, j)$, and the constrained lcs length from $(n/2, j)$ to (n, m). The latter is done in the reverse edit graph by calculating the constrained lcs length from (n, m) to $(n/2, j)$, hence we denote it by $L^{reverse}_{n/2,j,l}$ for $0 \leq \ell \leq k$. Then for every k,

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

is the constrained lcs length for k, and its calculation identifies a middle vertex which we store as part of the constrained lcs if it is located at a position where a symbol of S_1 and a symbol of S_2 match. After the middle vertex $(n/2, j)$ for every k is found, the problem of finding a constrained lcs from $(0,0)$ to (n, m) can be solved in two parts: find a constrained lcs from $(0,0)$ to $(n/2, j)$, and find a constrained 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 an lcs of S_1 and S_2 with respect to P can be obtained using $O(rm)$ space. The time complexity remains $O(rnm)$ because n is halved each time, and the computations involve in total $O(nm)$ vertices in the edit graph, 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. The 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 $L_{i,j,k,t}$ denote the length of an lcs of $S_1[1..i]$ 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 $S_1 = \mathtt{bbaba}$, $S_2 = \mathtt{abbaa}$ and $P = \mathtt{ab}$. We have calculated before that the length of an lcs of S_1 and S_2 with respect to P is 3. Thus $L_{5,5,2,0} = 3$. On the other hand the lcs $\mathtt{bbaa}$ of S_1 and S_2 contains the subsequence $\mathtt{a}$ whose edit distance from P is one. Therefore $L_{5,5,2,1} = 4$.

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

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

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

where

$$L'_{i,j,k,t} = \max\{L''_{i,j,k,t},\ L'''_{i,j,k,t},\ L''''_{i,j,k,t}\} \tag{4}$$

where

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

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

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

where

$$\begin{aligned} D_{i,j,k,t} &= \begin{cases} L_{i,j,k-1,t-1} & \textit{if } t \ge 1 \\ 0 & \textit{otherwise} \end{cases} \\ X_{i,j,k,t} &= \begin{cases} L_{i,j,k-1,t-1} & \textit{if } t \ge 1 \textit{ and } S_1[i] = S_2[j] \textit{ and } S_1[i] \ne P[k] \\ 0 & \textit{otherwise} \end{cases} \\ I_{i,j,k,t} &= \begin{cases} L_{i,j,k,t-1} & \textit{if } t \ge 1 \textit{ and } S_1[i] = S_2[j] \\ 0 & \textit{otherwise} \end{cases} \end{aligned}$$

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 for all $L_{i,j,k,t}$ defined in (3), $L_{i,j,k,t}$ is the optimum length for any common subsequence of $S_1[1..i]$ and $S_2[1..j]$ such that the common subsequence contains a subsequence whose edit distance is t from $P[1..k]$. We prove this by induction. The induction is based on an ordering of (i, j, k, t). The calculation of $L_{i,j,k,t}$ uses $L_{i,j-1,k,t}$, $L_{i-1,j,k,t}$, $L_{i-1,j-1,k-1,t}$, $L_{i-1,j-1,k,t}$, $L_{i,j,k-1,t-1}$, and $L_{i,j,k,t-1}$. Therefore (i, j, k, t) must come after $(i, j-1, k, t)$, $(i-1, j, k, t)$, $(i-1, j-1, k-1, t)$, $(i-1, j-1, k, t)$, $(i, j, k-1, t-1)$, and $(i, j, k, t-1)$ in the ordering. This ordering can be generated by four nested loops where t, k, i, j are the loop counters, respectively, of the loops from outer to inner.

In the base case: when $t = 0$ for all i, j, k the formulation becomes the same formulation as in Theorem 1, since now *lcs*'s are required to contain P itself as a subsequence. Therefore, the correctness of this case follows from Theorem 1.

Assume by way of induction that $L_{i,j-1,k,t}$, $L_{i-1,j,k,t}$, $L_{i-1,j-1,k-1,t}$, $L_{i-1,j-1,k,t}$, $L_{i,j,k-1,t-1}$ and $L_{i,j,k,t-1}$ defined by (3) are the *lcs* lengths for the corresponding edit distance constrained *lcs* (sub)problems at the neighboring nodes of (i, j, k, t). Next we consider the calculation of $L_{i,j,k,t}$.

Our solution uses the following observation: let *cs* be a subsequence of a common subsequence of S_1 and S_2. The minimum simple edit distance between *cs* and P can be calculated using insert, delete, and substitute operations in P, and using no operations in *cs*. This is because transforming *cs* to P, and transforming P to *cs* require the same number of edit operations by symmetry. 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.

For all node (i, j) in the edit graph, 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 $(a_1, b_1), (a_2, b_2), \cdots, (a_l, b_l)$ such that the edit distance between the string $S_1[a_1]S_1[a_2] \ldots S_1[a_l]$ $(= S_2[b_1]S_2[b_2] \ldots S_2[b_l])$, 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* on 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 extended by applying an edit operation in P. We can extend 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 one. Therefore, we use $D_{i,j,k,t} = L_{i,j,k-1,t-1}$, and take it into account in $L''''_{i,j,k,t}$. We can extend an edit path at node (i,j) at distance $t-1$ from $P[1..k-1]$ by substituting $S_1[i]$ for $P[k]$ if $S_1[i] = S_2[j]$ and $S_1[i] \neq P[k]$. Then on the resulting edit path *#match* remains the same, and the distance increases by one. Therefore, we use $X_{i,j,k,t} = L_{i,j,k-1,t-1}$ if $S_1[i] = S_2[j]$ and $S_1[i] \neq P[k]$, and take it into account in $L''''_{i,j,k,t}$. We can also extend an edit path at node (i,j) at distance $t-1$ from $P[1..k]$ by inserting $S_1[i]$ in P after position k if $S_1[i] = S_2[j]$. Then on the resulting edit path *#match* remains the same, and the distance increases by one. Therefore, we use $I_{i,j,k,t} = L_{i,j,k,t-1}$ if $S_1[i] = S_2[j]$, and take it into account in $L''''_{i,j,k,t}$. Combining all these $L''''_{i,j,k,t} = \max\{D_{i,j,k,t},\ X_{i,j,k,t},\ I_{i,j,k,t}\}$.

Since $L'_{i,j,k,t} = \max\{L''_{i,j,k,t},\ L'''_{i,j,k,t},\ L''''_{i,j,k,t}\}$ in (4), $L'_{i,j,k,t}$ 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 $P[1..k]$. We calculate, and store in $L_{i,j,k,t}$ maximum $\#match$ on such 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 $L_{i,j,k,t} \leq \max\{L'_{i,j,k,t}, L_{i,j-1,k,t}, L_{i-1,j,k,t}\}$. Hence $L_{i,j,k,t}$ is the length of an *lcs* of $S_1[1..i]$ and $S_2[1..j]$ that contains a subsequence whose edit distance is t from $P[1..k]$. This concludes the proof of the theorem. □

All $L_{i,j,k,t}$ can be computed in $O(drnm)$ 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 edit distance constrained *lcs* (i.e. an *lcs* of S_1 and S_2 that contains a subsequence whose edit distance from P is t) is desired then we can carry the *lcs* information for every k and t along with the calculations. This requires $O(drm^2)$ space. On any edit distance constrained *lcs* for each k and t, if we keep track of only the match points (i',j') where $S_1[i'] = S_2[j'] = P[u]$, $1 \leq u \leq k$, then the space complexity can be reduced to $O(dr^2m)$. In this case, an edit distance constrained *lcs* for $k=r$ needs to be recovered using an additional step at the end that performs ordinary *lcs* computations for S_1 and S_2 to connect the consecutive match points using in total $O(m)$ space. The *lcs* obtained this way is optimal for the edit distance constrained *lcs* problem because the problem definition uses the simple edit distance.

Remark: Space complexity can further be improved by applying the technique we used in our first algorithm. We can compute, instead of the entire edit distance constrained *lcs* for each k, and t, a middle vertex $(n/2,j)$ (assume for simplicity that n is even) at which an optimal 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 an optimal edit path from vertex $(0,0)$ to vertex $(n/2,j)$, and $\#match$ on an 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 $L^{reverse}_{n/2,j,k-l,t-u}$, can be calculated in the reverse edit graph. Then for all k,t,

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

is the optimum $\#match$ for k,t, and its calculation identifies a middle vertex which we store as part of the edit distance constrained *lcs* if it is at the position where a symbol of S_1 and a symbol of S_2 match. After the middle vertex $(n/2,j)$ on an optimal edit path for every k,t is found, the problem of finding an edit distance constrained *lcs* can be solved in two parts: find an edit distanced constrained *lcs* from $(0,0)$ to $(n/2,j)$, and find an edit distance constrained *lcs* from $(n/2,j)$ to (n,m) for all k,t. These two subproblems can be solved recursively. As a result an edit distance constrained *lcs* 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 [7]. We also extended the problem definition to use edit distances, and presented an $O(drnm)$ time algorithm for the resulting edit distance constrained *lcs* problem. This algorithm reduces to the ordinary constrained *lcs* problem for $d = 1$.

Acknowledgement

During the preparation of this paper Chin et al. [3] independently presented an $O(nmr)$-time algorithm for the constrained longest common subsequence problem. Their result is based on reducing the problem to a special case of the multiple sequence alignment problem of S_1, S_2, and P which can be solved in time $O(nmr)$.

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. F. Y.L. Chin, A. D. Santis, A. L. Ferrara, N. L. Ho, S. K. Kim. A simple algorithm for the constrained sequence problems. *Information Processing Letters* Vol. 90, pp. 175-179, 2004.
4. D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of ACM*, 18:341–343, 1975.
5. D.S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24:664–675, 1977.
6. W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *J. Comput. System Sci.*, 20:18–31, 1980.
7. Yin-Te Tsai. The constrained common sequence problem. *Information Processing Letters*, 88:173–176, 2003.
8. R.A. Wagner and M.J. Fisher. The string-to-string correction problem. *J. ACM*, 21:168–173, 1974.