REVISTA COLOMBIANA DE COMPUTACIÓN
Volumen 8, número 2
Págs 79 - 100

Sobre el Longest Common Subsequence: Extensiones y Algoritmos

Wilson Soto* Yoan Pinzón*

Resumen

Dadas dos palabras *x* e *y* sobre un alfabeto finito cualquiera, el problema de la *Longest Common Subsequence* (*LCS*) en castellano Subsecuencia Común Más Larga consiste, como su nombre sugiere, en encontrar cuál es el largo máximo que puede tener una palabra que sea subsecuencia de *x* e *y* simultáneamente. El presente artículo muestra una revisión y análisis de las diferentes extensiones y técnicas algorítmicas más conocidas hasta el momento y que dan solución a este problema.

Palabras clave: *Alineamiento, LCS, subsecuencia común más larga, similitud.*

Abstract

Given to strings x and y over a finite alphabet, the problem of the Longest Common Subsequence (LCS) is to find the longest possible string that is a subsequence of both x and y simultaneously. Here we present a survey and analysis of the different extensions and most common algorithmical techniques known up to now that can solve this problem.

Keywords: *Alignment, LCS, longest common subsequence, Similarity.*

1 Introducción

El *Longest Common Subsequence* (*LCS*) en castellano Subsecuencia Común Más Larga es uno de los principales problemas en el campo de la *stringology* (algoritmos para manipulación de cadenas de caracteres). El LCS es uno de los modelos computacionales más usados, pues sirve como medida de similaridad entre un conjunto de secuencias por medio de su longitud, que se muestra a través de los símbolos que contiene en común. Las aplicaciones que tiene el LCS son entre otras la compresión de datos, reconocimiento sintáctico de patrones, procesamiento XML, recuperación WEB, detección de intrusos y virus, comparación de archivos y bioinformática [36].

* Departamento de Ingeniería de Sistemas e Industrial, Grupo ALGOS-UN, Universidad Nacional de Colombia, Bogotá D.C., Colombia. Email: {wesotof,ypinzon}@unal.edu.co

El presente artículo busca dar una clasificación de los algoritmos desarrollados para dar solución a las extensiones del longest common subsequence desde el punto de vista de las técnicas más comúnmente usadas. Un mapa conceptual que puede resumir la idea básica de este artículo se ve en la Fig. 1. Por ello, en la Sección 2 se presentan los conceptos básicos y la notación necesaria para entender los problemas. La Sección 3 explica las técnicas usadas para dar solución a los problemas, incluyendo los algoritmos y estudios que se han desarrollado para las diferentes extensiones asociadas al problema del longest common subsequence. La Sección 4 pretende dar una visión práctica de donde aparece el problema e intenta descubrir las posibles aplicaciones y sobre todo un campo de investigación. La Sección 5 muestra las necesidades de investigación que son actualmente requeridas y en las que se puede desarrollar un estudio más profundo.

Fig. 1. Mapa conceptual del problema del longest common subsequence

2 El problema del longest common subsequence

Las siguientes definiciones son necesarias para entender este artículo: La distancia entre dos cadenas de caracteres *(strings) x* y *y* es el mínimo costo de operaciones para transformar *x* en *y*. El costo de una secuencia de operaciones es la suma de los costos de las operaciones individuales. Alineamiento significa insertar espacios tal que letras iguales se alineen, evitando que ocurra lo mismo con los espacios, pero aceptándolos al comienzo o final del *string* (*c.f.* Fig. 2).

G	C	A	T	-	A	T	T	-
-	C	A	T	G	-	T	T	G

Fig. 2. Alineamiento

Dado un alineamiento se calcula una medida de similitud, esta medida tiene tres posibilidades: (*i*) de letra-letra, (*ii*) error letra-letra y (*iii*) letra-espacio (o viceversa). Los tipos de alineamiento se resumen en [16]. Algunos de los algoritmos de alineamiento más conocidos son: Needleman y Wunsch (1970), SW (Smith & Waterman, 1981), FASTA (1988) y BLAST (1990) que son relacionados en [9].

La similitud de secuencias tiene diversas aplicaciones, entre otras ensamblaje de fragmentos, agrupamientos, minería de datos, comparación de genomas, análisis fitogénetico, homologías[1] funcionales y estructurales. Entre las medidas de similitud se encuentran la distancia de Hamming, La distancia de Levenshtein, la distancia de Episodio y el Longest Common Subsequence. El LCS permite únicamente inserciones y eliminaciones, todas de costo unitario. El nombre de esta distancia refiere a que esta medida representa la longitud más larga de acoplamiento entre los caracteres que pueden estar entre dos cadenas de caracteres, teniendo en cuenta el orden de las letras, caracteres o símbolos.

La siguiente es la notación formal usada. Sea Σ el Conjunto de elementos llamados símbolos, caracteres o letras. Por ejemplo Σ = {A, C, G, T}. Un *string* o palabra es una secuencia finita de símbolos que pertenece a un alfabeto Σ. Sea la función s de $\{1,\ldots,n\}$ sobre Σ tal que $s = a_1a_2 \ldots a_n$ donde $a_i = s(i) \in \Sigma$ es un *string* sin ningún símbolo o carácter. Σ^n define el conjunto de *strings* o palabras de longitud n. Por ejemplo Σ^3 = {LAS, ESS, FGR, DED, …}. Similarmente, Σ^* se define como el conjunto de strings sobre Σ de cualquier longitud. Por ejemplo Σ^* = {ACT, TAC, TAG, ACGT, CT, AG, ACTG, GG, AA, ε}. $|s|$ indica la longitud de un *string* s. Por ejemplo, para s = AGCATT, $|s| = 6$ o si $s = \varepsilon$, $|s| = 0$. Un prefijo de s se define como el *string* v tal que $s = vt$ para ciertos $t \in \Sigma^*$. Así mismo, un sufijo de s se define como el *string* v tal que $s = tv$ para ciertos $t \in \Sigma^*$. *D*efinimos d: $\Sigma^* \times \Sigma^* \rightarrow$ R como un función de distancia. Igualmente definimos las siguientes operaciones:

- Inserción: Insertar un carácter en un *string.* Por ejemplo la operación de inserción sobre el *string* s = TCA adicionando la letra G, es s' = TCGA.

1 proteínas que tienen un origen evolutivo común

- Eliminación: Eliminar un carácter de un *string.* Por ejemplo la operación de eliminación sobre el *string s* = TCGA eliminando una letra, es *s'* = TGA.

- Sustitución: Reemplazar una letra en un *string.* Por ejemplo la operación de sustitución sobre el *string s* = AATC reemplazando una letra por otra, es *s'* = ATTC.

Con lo anterior, la definición del problema del LCS consiste en: DadoΣ y las secuencias $S_1, S_2 \in \Sigma^*$, $|S_1| = n$ y $|S_2| = m$ $(m \leq n)$, $LCS(S_1, S_2)$ es una secuencia W tal que:

1. $i, 1 \leq i \leq |W|\text{-}1, j, j' : 1 \leq j < j' \leq |S_1|, k, k' : 1 \leq k \leq k' \leq |S_2|$ tal que $W[i] = S_1[j] = S_2[k]$, y $W[i+1] = S_1[j'] = S_2[k']$;
2. $W' \in \Sigma^* : (1)$ y $|W'| > |W|$

El objetivo es en encontrar cuál es el largo máximo que puede tener un *string* que sea subsecuencia de S_1 y S_2 simultáneamente. Dado *S,* un conjunto de secuencias, *s* es una secuencia común de *S* si esta es una subsecuencia de cada secuencia en *S*. La Fig. 3 muestra un ejemplo de algunas de las subsecuencias comunes para dos cadenas S_1 y S_2 con $|\text{LCS}(S_1, S_2)| = 9$.

S_1=	1	0	0	2	3	2	1	1	0	2	2	2	3	3	1	0	0	1
S_2=	0	1	1	1	3	3	0	2	1	2	1	1	0	1	2	1		

sc_1=	1	3	2	1	2	1		
sc_2=	0	1	1	3	3	1		
sc_3=	0	1	1	0	2	2	1	0	1
sc_4=	1	0	2	2	1	1	0	2	1

Fig. 3. Ejemplos de subsecuencias comunes para S_1 y S_2. sc_3 y sc_4 son LCS.

En cuanto a la complejidad, el LCS es un problema que puede ser resuelto en tiempo polinomial con programación dinámica para do s *strings*, o para algún número fijo de *strings,* pero este se convierte en un problema "NP-Hard" en general para un número arbitrario *k* de *strings.* En [24] se dedica un capítulo completo a la complejidad del algoritmo y la relación con la teoría NP que se resume al problema de maximización siguiente: "Dado un lenguaje finito no vacío *X*, encontrar una subsecuencia común de *X* de longitud máxima Algunos trabajos ya han estudiado el tema como [3] [10], que tratan el tema con soluciones heurísticas que dan solución a la subsecuencia común más larga en tiempo polinomial.

2.1 Extensiones del LCS

2.1.1. All Longest Common Subsequence (ALCS)

El problema del *all longest common subsequence* consiste en buscar todas las subsecuencias comunes más largas entre X y Y. Los estudios completos sobre los límites en tiempo de ejecución sobre todos los algoritmos que generan ALCS distintos y ALCS embebidos (posiciones en ambos *strings* para los cuales los caracteres del LCS correspondan) se encuentran en [1]. Una de las variaciones presentadas en este tipo de extensión es el *all semi-local longest common subsequence* [42], donde cada string es comparado contra todos los sufijos del otro *string.*

2.1.2 Length Longest Common Subsequence (LLCS)

La longitud de un longest common subsequence de dos o más strings es una útil medida de su similaridad. La longitud esperada de un LCS es proporcional a la longitud de las secuencias dadas y dependiente del tamaño del alfabeto [20]. La estimación de esta longitud esperada promueve algunos problemas de interés combinatorial.

En [32] algunos de los algoritmos desarrollados para solucionar esta extensión del longest common subsequence tiene que ver con técnicas de programación dinámica, naive recursive (recursión ingenua) y programación dinámica con recursión. Otras investigaciones como en [34] y [35] proponen una solución basada en tipos de algoritmos de paralelismo aplicados a modelos de procesamiento.

2.1.3 Constrained Longest Common Subsequence (CLCS)

Dados los strings S_1, S_2 y P, el problema del CLCS para S_1 y S_2 con respecto a P es buscar un Longest Common Subsequence LCS de S_1 y S_2 tal que P es una subsecuencia de este LCS. En [4] se encuentran algoritmos relacionados con el estudio de este tipo de extensión, mencionando el concepto de grafo de edición, que surge a partir de la solución del longest common subsequence usando la matriz de programación dinámica (ver Fig. 4).

El grafo de edición es un grafo en el que se presentan las celdas como nodos y los vértices como las operaciones, con un tamaño de $(n+1)(m+1)$ puntos (i,j) para $0 \le i \le n$, y $0 \le j \le m$ como vértices. El peso de los arcos corresponde al costo de las operaciones, inserción $(i\text{-}1, j)$, eliminación $(i, j\text{-}1)$ y operaciones de igualación o reemplazo $(i\text{-}1, j\text{-}1)$. La distancia de edición entre los dos strings es la ruta más corta desde el

vértice [0, 0] (esquina superior izquierda de la matriz) al vértice [*m*, *n*] (esquina inferior derecha de la matriz). Para el ejemplo, la distancia es cinco y el LCS(*x*, *z*) = "ACDBC" (ver Fig. 5).

2.1.4 Shortest Common Supersequence (SCS)

El problema del *shortest common supersequence* (SCS) consiste en buscar para un conjunto dado de *strings P*, el string más pequeño el cual es una supersecuencia de cada *string* de *P*. Formalmente el problema se define como dadas dos secuencias *u* y *v*, la secuencia *u* es una supersecuencia de *v* si *v* es una subsecuencia de *u*. Si $u \in \Sigma^*$ es una supersecuencia común de $v_1,\ldots,v_n$, si para cada $i = 1,...,n$, secuencia *u* es una supersecuencia de v_i, entonces *u* es una supersecuencia de longitud mínima si *u* es una supersecuencia común de longitud mínima posible [19].

Aunque este problema es dual con el problema del Longest Sommon Subsequence, muy pocos algoritmos han sido desarrollados, autores como Itoga en 1981 y Fraser e Irving en 1995 han tratado el problema en [43]. En [44] también se aborda este tipo de extensión.

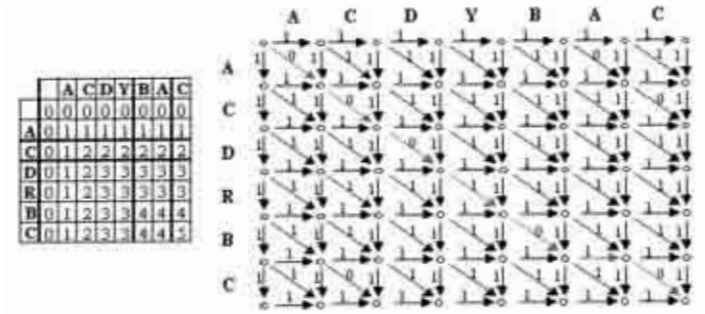

Fig. 4. Matriz de programación dinámica y grafo de edición asociado para *x* = "ACDYBAC" y *z* = "ACDRBC"

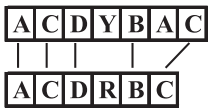

Fig. 5. Longest common subsequence entre *x* = "ACDYBAC" y *z* = "ACDRBC"

2.1.5 Longest Common Increasing Subsequence (LCIS)

Un problema similar al LCS es el problema de *Longest Increasing Subsequence* (LIS) (en castellano, subsecuencia creciente más larga),

el cual consiste en, dada una permutación P de los números 1,2,...,n, encontrar el largo máximo que puede tener una subsecuencia de P que sea creciente. Por ejemplo, si P = (6,2,8,5,7,4,1,9,3) entonces una LIS (también la única) es R = (2,5,7,9). Un problema de LIS se reduce a uno de tipo LCS. Para ello, interpretamos una permutación $R = (R_1,R_2,...,R_n)$ sobre los números 1,2,...,n como la palabra $R_1R_2...R_n$. Es fácil ver que una LIS entre P y Q corresponde a un LCS entre P y Q = (1,2,...,n). Esta variante es ampliamente estudiada en [14] y [15], además mencionan una extensión en particular del *Longest Common Increasing Subsequence*, el LCWIS (*longest common weakly-increasing subsequence*) que considera, un limite y no subsecuencias estrictamente incrementales.

2.1.6 Exemplar Longest Common Subsequence (ELCS)

Es una generalización del problema del longest common subsequence, donde las secuencias de entrada están sobre la unión de dos conjuntos disyuntos de símbolos, un conjunto de símbolos mandatorio y un conjunto de símbolos opcional. Esta extensión tiene un acercamiento heurístico el cual es básicamente una restricción al problema de alineamiento de un *string*. El *exemplar longest common subsequence* es un conjunto S de secuencias sobre un alfabeto $A_o \cup A_m$, donde A_o es el conjunto de símbolos opcionales y A_m es el conjunto de símbolos mandatorios. El conjunto A_o y el conjunto A_m son disyuntos, la salida es un secuencia común más larga de todas las secuencias en S que contienen todos los símbolos mandatorios [10]. Algunas de las versiones del *exemplar longest common subsequence* se ven en la Tabla 1.

Tabla 1. Versiones del ELCS

Nombre Problema	Ocurrencias de símbolos mandatorios	Ocurrencias de símbolos opcionales
ELCS(1,<=1)	exactamente 1	a lo sumo 1
ELCS(1)	exactamente 1	sin restricción
ELCS(>=1,<=1)	al menos 1	a lo sumo 1
ELCS(>=1)	al menos 1	sin restricción

2.1.7 Otras variantes del Longest Common Subsequence

Estas variantes son relacionadas en [30], introduciendo la noción de *gap-constrains* al LCS. Esta extensión ofrece la posibilidad de

manipular gap-constraints entre las igualdades consecutivas en medio de las secuencias y además provee la herramienta de manipular problemas de búsqueda donde no todas las pos[illegible]ne

Dado un *string* $X[1..n] = X[1]\ X[2] \ldots X[n]$ y una subsecuencia $S[1..r] = S[1]\ S[2] \ldots S[r]$ de X, se define una secuencia de correspondencia de X y S, $C(X, S) = C[1]\ C[2] \ldots C[r]$ si es una secuencia estrictamente incremental de enteros tomando desde $[1, n]$ tal que $S[i] = X[C[i]]$ para todo $1 \leq i \leq r$. Dado X y una de sus subsecuencias S, $C(X, S)$ puede no ser única. El LCS con espacio fijo (FIG), se define como buscar la subsecuencia común fija espaciada (*fixed gapped*) de máxima longitud entre dos strings X y Y, si existe una secuencia de correspondencia fija espaciada CFG(k) (X, S) y CFG(k) (Y, S). Una secuencia de correspondencia fija espaciada de un *string* X de longitud n y una subsecuencia S de longitud r, con respecto a un entero dado K, se da, si y solo si tiene $C[i] - C[i\text{-}1] <= K + 1$ para todo $2 \leq i \leq r$. El LCS con espacio elástico (ELAG), se define como buscar la subsecuencia común elástica espaciada (*elastic gapped*) de máxima longitud entre dos *string*s X y Y, si existe una secuencia de correspondencia elástica espaciada CFG(k) (X, S) y CFG(k) (Y, S). Una secuencia de correspondencia elástica de un string X de longitud n y una subsecuencia S de longitud r, con respecto a enteros dados K_1 y K_2, $K_2 > K_1$, se da, si y solo si se tiene $K_1 < C[i] - C[i\text{-}1] \leq K_2 + 1$ para todo $2 \leq i \leq r$. El LCS con espacio rígido fijo (RIFIG), se define como buscar la subsecuencia común dados dos strings $X[1..n] = X[1]\ X[2] \ldots X[n]$ y $Y[1..n] = Y[1]\ Y[2] \ldots Y[n]$ y un entero K y una subsecuencia común $S[1..r] = S[1]\ S[2] \ldots S[r]$ de X y Y, si existe una secuencia de correspondencia espaciada fija CFG(k) (X, S) y CFG(k) (Y, S) tal que para todo $2 \leq i \leq r$. El LCS con espacio rígido elástico (RELAG), se define como buscar la subsecuencia común dados dos strings $X[1..n] = X[1]\ X[2] \ldots X[n]$ y $Y[1..n] = Y[1]\ Y[2] \ldots Y[n]$ y un entero K y una subsecuencia comn $S[1..r] = S[1]\ S[2] \ldots S[r]$ de X y Y, si existe una secuencia de correspondencia espaciada fija CFG(k) (X, S) y CFG(k) (Y, S) tal que para todo $2 \leq i \leq r$.

3 Técnicas de solución y algoritmos

La Tabla 2 muestra una clasificación que reúne los algoritmos realizados para dar solución al problema del LCS y algunas de sus extensiones, relacionando las técnicas usadas. Las técnicas usadas están basadas en la división realizada en [33], las cuales son: Programación Dinámica, Autómatas Finitos, Filtramiento y

3.1 Programación dinámica

En la programación dinámica, la solución propuesta usa un algoritmo que construye una matriz $D[0..m, 0..n]$ donde, $D_{i,j}$ representa el LCS para $x_1..i$ y $z_1..j$. Cada celda es calculada segn los vecinos superior, izquierdo y superior izquierdo de la celda. La matriz D es calculada de la siguiente manera (*c.f.* Fig. 6): $D_{i,j} = 0$ si $i = 0$ o $j = 0$, $D_{i,j} = D[i\text{-}1, j\text{-}1] + 1$ si $x_i = z_j$, $D_{i,j} = \max\{D[i\text{-}1, j], D[i,j\text{-}1]\}$ si $x_i \neq z_j$, y la distancia de edición es la celda $D[m, n]$. Por ejemplo, para los strings x =“CTACTAATA” y z = “ATCGTT”, $|LCS(x, t)| = D(6, 9) = 4$ y para determinar la subsecuencia común más larga entre las cadenas dadas se utiliza los conceptos de frontera y puntos dominantes [8]. La frontera no es más que la delimitación de los números similares dentro de la matriz, como se ve en la Fig. 6 las líneas gruesas y los puntos dominantes son las esquinas de estas fronteras. Para saber cual es la subsecuencia común más larga se recorre desde la esquina de la frontera inferior derecha buscando las esquinas o puntos dominantes que están a la izquierda y por encima al punto dominante anterior. En el ejemplo, empezamos con 4 ([T,T] posición [6,8]) luego pasamos a la frontera 3 ([T,T] posición]5,5[(luego a la 2)[C,C] posición [3,4](y por último a 1 que puede ser)[T,T] posición [2,2](o)A,A posición [1,3]). Entonces el $LCS(x, t)$ = “ACTT” o “TCTT”.

		C	**T**	**A**	**C**	**T**	**A**	**A**	**T**	**A**
		0	0	0	0	0	0	0	0	0
A	0	0	0	(1)	1	1	1	1	1	1
T	0	0	[1]	1	1	2	2	2	2	2
C	0	1	1	1	(2)	2	2	2	2	2
G	0	1	1	1	2	2	2	2	2	2
T	0	1	2	2	2	(3)	3	3	3	3
T	0	1	2	2	2	3	3	3	(4)	4

Fig. 6. Matriz de Programación Dinámica

En este tipo comunes y en consecuencia la más larga. El algoritmo que construye la matriz tiene complejidad en tiempo de ejecución de $O(mn)$ y el espacio requerido es $O(mn)$. El algoritmo de bůsqueda de texto de Hirschberg toma un tiempo de ejecución $O(nl + n \log s)$ y espacio $O(d + n)$ basado en los puntos dominantes [2].

El algoritmo que genera la matriz de programación dinámica es el

Algoritmo 1 DP

```
LCS(a,b) {m=|a|, n=|b|}
for i ← 0 to m do D[i, 0] ← 0
for j ← 0 to n do D[0,j] ← 0
for i ← 1 to m do
    for j ← 1 to n do
        if a_i = b_j then L[i,j] ← L[i-1,j-1] + 1
        else
            if D[i,j-1] > D[i-1,j] then D[i,j] ← D[i,j-1]
            else D[i,j] ← D[i-1,j]
    endfor
endfor
```

Un ejemplo de solución dentro de este tipo de técnica es el algoritmo de solución del ALCS (*all longest common subsequence*) se encuentra el mencionado en [26], con tiempo de ejecución de O(*mn*) y espacio O(*m*).

Algoritmo 2 ALCS – DP

```
LCS(a,b) {m=|a|, n=|b|}
for j ← 0 to n do
    for i ← 0 to m do
        if i = 0 or j = 0 then
            D[i, j] ← 1
        else
            D[i, j] ← 0
            if a_i = b_j then D[i, j] ← D[i - 1, j - 1]
            else
                if L[i - 1, j] = L[i, j] then
                D[i, j] ← D[i, j] + D[i - 1, j] endif
                if L[i, j - 1] = L[i, j] then
                D[i, j] ← D[i, j] + D[i, j - 1] endif
                if L[i - 1, j - 1] = L[i, j] then
                D[i, j] ← D[i, j] - D[i - 1, j - 1] endif
            endif
        endif
    endfor
endfor
```

Una de las extensiones de esta técnica es la técnica *Sparse Dynamic Programming* (programación dinámica esparcida) que se estudia en [6], y básicamente es una técnica diseñada para el mejoramiento de algoritmos en el análisis de secuencias, que funciona dado un conjunto de recurrencias de programación dinámica para calcular la matriz. Se usa un conjunto esparcido de las entradas a la matriz que son de importancia, proporcionando la ventaja de producir algoritmos que tienen tiempo de ejecución dependiente sobre el tamaño del conjunto esparcido mejor que sobre el tamaño de la matriz de programación dinámica.

Tabla 2. Complejidad de algoritmos para solucionar el Longest Common Subsequence y sus extensiones

	TÉCNICA	AÑO	TIEMPO	ESPACIO	AUTOR
LCS	PD	1974	$O(mn)$	$O(mn)$	Wagner - Fisher
	PD	1975	$O(mn)$	$O(n)$	Hirschberg
	PD	1977	$O((n+R) \log n)$	$O(R+n)$	Hunt - Szymanski
	PD	1977	$O(Ln + n \log n)$	$O(Ln)$	Hirschberg
	PD	1977	$O(L(m - L)\log n)$	$O((m - L)^2 + n)$	Hirschberg
	PD	1980	$O(n \max\{1, m/\log n\})$	$O(n^2/\log n)$	Masek - Paterson
	PD	1982	$O(n(m - L))$	$O(m^2)$	Nakatsu - Kambayashi - Yajima
	PD	1984	$O(Lm \log(n/L) + Lm)$	$O(Lm)$	Hsu Du
	PD	1985	$O(Em)$	$O(E \min\{m, E\})$	Ukkonen
	PD	1986	$O(kn)$	$O(n)$	Landau - Vishkin
	PD	1986	$O(n + m \log n + D \log (mn/D))$	$O(R+n)$	Apostolico
	PD	1987	$O(n(m - L))$	$O(n)$	Kumar - Ragan
	PD	1987	$O(Lm + n)$	$O(D + n)$	Apostolico - Guerra
	PD	1990	$O(n(m - L))$	$O(n)$	Wu - Manber - Myers - Miller
	PD	1990	$O(n + \min\{D, Lm\})$	$O(D + n)$	Chin - Poon
	PD	1992	$O(n(m - L))$	$O(n)$	Apostolico- Browne - Guerra
	PD	1992	$O(Lm)$	$O(n)$	Apostolico- Browne - Guerra
	PD	1992	$O(n + D \log \log \min\{D, mn/D\})$	$O(D + m)$	Eppstein - Galil - Giancarlo - Giuseppe
	A	1996	$O(mn/\log n)$	$O(n)$	Wu - Manber - Myers
	A	1995	$O(min(3^m, m(2mt)^k, (k+2)^{m-k}(k+1)!))$	t numero estados	Melichar
	F	1999	$O(kn \log(m)/w)$	-	Baeza Yates - Navarro
	F	1990	$O(n \log \lvert\Sigma P\rvert + pm)$	-	Grossi - Luccio
	PB -NFA	1996	$O(\lceil k(m - k)/w \rceil n)$ peor caso $O(\lceil k^2 / w \rceil n)$ promedio	-	Baeza Yates - Navarro
	PB -PD	1994	$O(nm/\log(\delta) / w)$	-	Wright
	PB -PD	1999	$O(\lceil m/w \rceil n)$peor caso $O(\lceil k / w \rceil n)$promedio	-	Myers
	PD	2006	$O(R \log \log n + n)$	$\theta(max\{R, n\})$	Iliopoulos - Rahman
ALCS	PD	2002	$O(2mn)$	$O(mn)$	Kunkle
	PD	2003	$O(mn)$	$O(m)$	Greenberg
	A	2006	$O(mn/\log(m+n))$	$O(m+n)$	Tiskin
CLCS	PD	1992	-	-	Sankoff
	PD	2005	$O(L[patrón]* n^2m^2)$	$O(L[patrón]*nm)$	Abdullah - Omer
	PD	2005	$O(k(mL + R) + n)$	$O(Rk)$	Deorowicz
	PD	2006	$O(pR \log \log n + n)$	$\theta(max\{R, n\})$	Iliopoulos - Rahman
LLCS	PD	1975	$O(mn)$	$O(m+n)$	Hirschberg
	PD	2002	$O(2^{mn})$	-	Kunkle
	PD	2002	$O(mn)$	-	Kunkle
	PD	2002	$O(mn)$	$\theta(mn)$	Kunkle
	PB	1993	$O(m+n+1)$	$O(m+n)$	Mi - Rahman
LCIS	PD	2004	$O(mn)$	$O(mn)$	Yang - Huang - Chao
	PD	2005	$O((m +nl) \log \log \delta + Sort (m))$	$O(m)$	Brodal - Kaligosi - Katriel - Kutz
LCWIS	PD	2005	$O(m + n \log n)$ para alfabeto de 2 y 3 letras	-	Brodal - Kaligosi - Katriel - Kutz
ELCS	A	2006	-	-	Bonizzoni - Della Vedova - Dondi - Fertin - Vialette
SCS	PD	1995	$O(k)$	-	Dnacik
FIG	PD	2006	$O(n^2 + R \log \log n)$	$\theta(R)$	Iliopoulos - Rahman
ELAG	PD	2006	$O(n^2 + R \log \log n)$	$\theta(max(R,n))$	Iliopoulos - Rahman
RELAG	PD	2006	$O(n^2 + R(k_2 + k_1))$	$\theta(n^2)$	Iliopoulos - Rahman
RLCS	PD	2006	$O(n^2)$	$\theta(n^2)$	Iliopoulos - Rahman
RIFIG	PD	2006	$O(n^2)$	$\theta(n^2)$	Iliopoulos - Rahman

δ= tamaño del alfabeto; A = Autómata Finito No Determinista; ALCS = All Longest Common Subsequence; CLCS = Constrained Longest Common Subsequence; D = Número de puntos dominantes; E = distancia de edición (E = m + n - 2L); ELAG = LCS Elastic Gap; ELCS =Exemplar Longest Common Subsequence; F = Filtramiento; FIG = LCS Fixed Gap; k = Número de errores; l = Longitud del LCIS/LCWIS; L = Longitud del longest common subsequence; LCS = Longest Common Subsequence; LCIS = Longest Common Increasing Subsequence; LCWIS = Longest Common Weakly-Increasing (or non decreasing) Subsequences; LLCS = Length Longest Common Subsequence; m = $L(S_1)$ (m <= n); n = $L(S_2)$; p = longitud del tercer string el cual aplica la restricción; PB = Paralelismo de Bits; PD = Programación Dinámica; R = Numero de igualaciones; RELAG = LCS Rigid Elastic Gap; RIFIG = LCS Rigid Fixed Gap; RLCS = LCS Rigid Gap; SCS = Shorthest Common Supersequence; Sort = Tiempo de orden de cada secuencia de entrada; t = min(m + 1, s); w = longitud de palabra del computador en bits.

3.2 Autómata finito

Un autómata finito es una máquina finita de estados donde para cada par de estados y símbolos de entrada hay una y solo una transición para un nuevo estado, además es determinista si las transiciones son conocidas y dados algunos datos de entrada, se sabe que transición realizará la máquina. El grafo de edición sobre un *string S*, es un autómata finito determinista que tiene la capacidad de reconocer todos los substrings de *S*. Un autómata de sufijos no determinista reconoce algún sufijo para un *string S* (ver Fig. 7).

Ahora consideremos el anterior autómata con errores, donde cada fila denota el número de errores visto (la primera fila cero, la segunda fila uno, etc.). Cada columna representa la igualación a un prefijo de patrón. Las flechas horizontales representan la igualación a un carácter, todas las otras flechas incrementan el número de errores, al moverse a la próxima fila, las flechas verticales insertan un nuevo carácter en el patrón y las flechas diagonales representan el reemplazo o eliminación de un carácter del patrón. La flecha inicial que hace un bucle sobre el primer nodo, indica una igualación para iniciar en cualquier parte en el texto (ver Fig. 8).

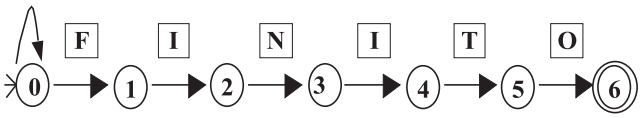

Fig. 7. Autómata finito no determinista que busca exactamente “finito”

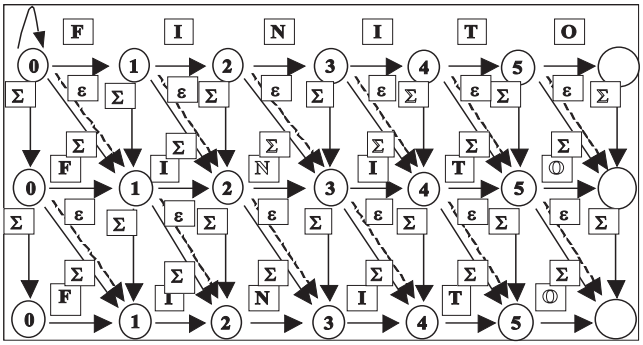

Fig. 8. Autómata no finito determinista que busca el texto “finito” con 2 errores

En [36] se reseña un análisis del autor Melichar de 1995, que mejora la complejidad de espacio y tiempo de preprocesamiento del autómata a la siguiente expresión O(min(3m, m(2mt)k, (k+2)m-k(k+1)!)) donde t = min(m+1,σ), t es el empleo de tiempo en m número de estados. Una de las extensiones del problema del longest common subsequence que usa esta técnica es la exemplar Longest Common Subsequence (ELCS) [10], empleando un DAG (grafo acíclico directo) o autómata finito determinista.

3.3 Filtramiento

Filtramiento o filtración, son algoritmos que filtran el texto rápidamente descartando áreas del texto las cuales no generan igualaciones, o mejor, buscan potenciales igualaciones y luego aplican un algoritmo secuencial para revisar cada candidato. Estos algoritmos están enfocados al caso promedio, y son de mayor funcionalidad en los algoritmos que no realizan toda la inspección de los caracteres del *string.* Esta técnica envuelve dos etapas de proceso. La primera etapa preselecciona un conjunto de posiciones en el texto que son potencialmente similares al patrón. La segunda etapa verifica cada posición potencial usando un método exacto de rechazo potencial de igualaciones con k errores. La preselección utiliza usualmente O(n+p) tiempo, donde n es el coeficiente mas pequeño de los algoritmos de preprocesamiento patrón/texto y p es el número potencial de igualaciones buscadas en la primera etapa del algoritmo. Algunas de las derivaciones de esta técnica son la filtración por tuplas (si un patrón aproximadamente iguala un substring del texto, entonces ellos comparten al menos una l-tupla para un l suficientemente grande) y filtración por composición (si un patrón aproximadamente iguala un substring de un texto, entonces ellos tienen composiciones de letras similares). Es de importancia que la complejidad de los algoritmos desarrollados en este tipo de técnica dependan críticamente de la tasa de r/p, donde r es el número real de igualaciones con k errores y p es el número potencial de igualaciones buscados en la primera etapa del algoritmo. Entre mas grande la taza, mas pequeña es la ejecución de la segunda etapa de filtración del algoritmo. La filtración es útil, para grandes niveles de error donde las áreas de texto a verificar cubren casi todo el texto. La verificación puede ser hecha en una forma jerárquica [36], este concepto puede ser visto en la Fig. 9, en la cual se desea buscar el patrón “ACTGACAA” con cuatro errores en cualquier texto. Si se divide el patrón en cuatro subpatrones con algún error, cualquiera de estos puede ser buscado en el texto. Puede verificarse la búsqueda del patrón completo en un texto desde las hojas hasta la raíz (patrón).

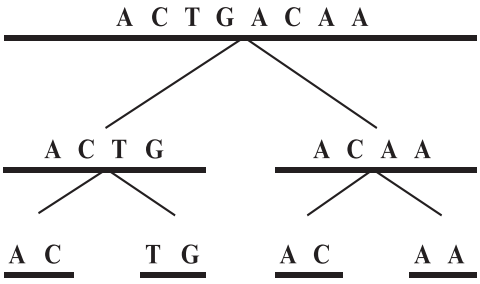

Fig. 9. Verificación jerárquica

3.4 Paralelismo de bits

Son algoritmos basados en el trabajo de paralelismo del computador sobre los bits. Se usan sobre los autómatas finitos no deterministas y la matriz de programación dinámica que solucionan el problema del longest common subsequence [36]. Este funciona específicamente con el número de bits de la palabra del computador. Las actuales arquitecturas funcionan con palabras de 32 o 64 bits de procesamiento, lo que es muy importante en la práctica. Este valor es comúnmente representado como ω. Por ello las operaciones usadas en este tipo de técnica están basadas en las operaciones con los bits, que son: conjunción, disyunción inclusiva, disyunción exclusiva, negación y desplazamiento o corrimiento.

Los algoritmos desarrollados en paralelismo de bits buscan un patrón en un texto simulando la operación de un autómata no determinista y muchos de estos pueden verse como implementaciones de un autómata determinista, pues el uso de esta técnica convierte los autómatas no deterministas a deterministas con la ventaja de ser mas simple, mas rápido y fácil de extender para manipular patrones complejos, pero con la principal desventaja de la limitación impuesta con el tamaño de palabra del computador.

Unos de los algoritmos desarrollados en es tipo de técnica son el BPD (*Bit-Paralellism Diagonals*) que es basado sobre un autómata finito no determinista con el paralelismo de bits por las diagonales cuya complejidad es $O(\lceil km / \omega \rceil n)$ y el BPR (*Bit-Paralellism Rows*) que esta basado también sobre un autómata finito no determinista pero con paralelismo de bits por filas con una complejidad de $O(\lceil m/ \omega \rceil kn)$. Uno de los mejoramientos en los algoritmos de paralelismo de bits [17] es $O(mn \log(\gamma) / \omega)$ en el peor caso y $O(n)$ en el caso promedio.

4 Aplicaciones prácticas

Muchos de los estudios relacionados con la solución del problema del longest common subsequence han mostrado la implementación y posterior experimentación del algoritmo propuesto como en [11], desde las primeras investigaciones como [23], hasta modificaciones en el uso de un sistema para permitir patrones de subsecuencias, como además patrones de substrings en nodos y aceleración en el tiempo de computación [3]. La aplicación práctica de la solución es diverso, entre las más importantes a mencionar están: la búsqueda de patrones en secuencias biológicas, búsqueda y recuperación de texto, comparación de archivos y detección de virus e intrusos.

- **Análisis de secuencias biológicas**: El análisis de secuencias biológicas es de suma importancia pues se puede determinar la evolución de las especies, la cual es dada por la variación genética de los organismos (genes) y por consiguiente las proteínas que los forman. Estas evoluciones de las proteínas están clasificadas en tres formas que son la mutación, duplicación y barajado de dominios. Por ello, la comparación de secuencias (nucleótidos y/o aminoácidos) es una forma de descubrir que partes de las secuencias son más importantes (están más conservadas), además de poder predecir la estructura o la función de las proteínas. Algunos de los trabajos relacionados se encuentran en [37].

- **Búsqueda y recuperación de texto**: La recuperación de texto consiste en buscar la información de mayor importancia en una recopilación de texto, teniendo en cuenta que la tarea básica es la igualdad de strings, y la búsqueda de patrones de texto permite acceder a zonas de interés donde se encuentra la mayor cantidad de información. La necesidad de descubrir y reconocer variantes formas de strings como por ejemplo variantes de la ortografía, errores ortográficos y diferentes transliteraciones incrementan la dificultan en recuperación de información, la búsqueda aproximada de texto en la que se encuentra el longest common subsequence ayudan a solucionar este problema. Un ejemplo importante en este tipo de aplicación, son los buscadores WEB. Uno de los estudios interesantes en este tipo de aplicación esta en [7], donde se analiza la búsqueda sobre texto comprimido. Otro estudio importante en el área esta en [5], que consiste en la compresión de texto usando strings comunes largos en un archivo de texto, implementado el algoritmo de string matching de Karp y Rabin(1987).

- **Comparación de archivos**: La comparación de archivos busca determinar las diferencias entre dos archivos de texto. El resultado de estas comparaciones son típicamente mostradas al usuario, pero también pueden ser usadas para lograr tareas en redes, sistemas de archivos y control de revisión. El programa *diff*[2] tiene un algoritmo capaz de solucionar el problema del longest common subsequence, el cual busca las líneas que no cambian entre los archivos. La eficiencia práctica del algoritmo es porque este se fija en las igualaciones esenciales de los archivos, lo que hace que se reduzca a la subsecuencia común mas larga para algún par de segmentos iniciales de los dos archivos [28].

- **Detección de intrusos y virus**: La aplicación del longest Common Subsequence en la detección de intrusos forma parte de la categoría de detección de intrusos llamado de uso impropio usando *pattern matching*. Este tipo de categoría refiere a intrusiones que siguen un patrón bien definido de ataques que explotan la debilidad en un sistema y software de aplicación. Esta técnica representa el conocimiento acerca del comportamiento malo o inaceptable y busca detectarlos directamente, en contraste a la categoría de detección de intrusos de anomalía. En la categoría de detección de intrusos de uso impropio las firmas de intrusión son usualmente especificadas como una secuencia de eventos y condiciones que conducen a forzar una entrada. Algunos patrones en escenarios de ataques tienden a extraer virus de archivos infectados. Se considera de entrada un flujo de eventos para ser igualado contra el patrón (firma de intrusión). Una de las desventajas de este acercamiento es que este considera únicamente las vulnerabilidades conocidas y por consiguiente no funciona en posteriores intrusiones desconocidas. El marco completo de la aplicación del longest common subsequence en la detección de intrusos y virus se puede observar en [33].

5 Trabajos a futuro

Actualmente los estudios relacionados con el LCS se dirigen a reducir la complejidad de tiempo de ejecución y de espacio por medio de algoritmos y diversas técnicas con el fin de llegar a resultados más eficientes y mejores, donde el problema tiene aplicaciones prácticas. Además, el empleo de otras técnicas para solucionar el problema del Longest Common Subsequence como algoritmos de aprendizaje [13] y otras donde se basan en solución de extensiones del problema, como en [16], basado en una técnica de solución del shortest common

2 Desarrollado en 1970 sobre el sistema operativo Unix en los laboratorios AT&T Bell. El primer prototipo apareció en 1976 por James Hunt

supersequence, son investigaciones que están abiertas. Otros de los trabajos a futuro para el desarrollo de soluciones del problema, comprenden el análisis sobre el poder de algoritmos no basados en la comparación, como los algoritmos basados en bits. Reducción de trabajo para búsqueda de texto como por ejemplo búsqueda múltiple de *strings* con más comprobaciones y algoritmos óptimos no dependientes del texto de entrada. En general la dirección de las investigaciones a futuro están en la combinación de una heurística y un algoritmo longest common subsequence exacto. Lo que se busca es que las dos fases podrían ser ortogonales en el sentido que el trabajo hecho durante el preprocesamiento pueda ser completamente utilizado en el algoritmo exacto.

6 Conclusiones

Según la Tabla 2 que resume la agrupación de algunos de los algoritmos que dan solución al problema del Longest Common Subsequence y sus extensiones, se puede concluir que hay varias investigaciones por realizar en el área, como por ejemplo utilizar técnicas para desarrollar algoritmos en las que posiblemente no se han estudiado. La intención no era reproducir textualmente los análisis de la solución y los algoritmos, la idea era agrupar la mayoría de estos y principalmente lograr una clasificación según las técnicas comunes con algunas de las extensiones encontradas. Así la finalidad era mostrar una visión general del problema y profundizar en el entendimiento del mismo para incentivar la investigación a futuro en esta importante área de la algoritmia. Sin embargo, determinar cual es el mejor de los algoritmos que dan solución al problema del longest common subsequence no es correcto, pues las mismas investigaciones crean ambientes de experimentación propios, lo que lleva a la conclusión que dependiendo de la aplicación práctica se debe recurrir a la solución más apropiada.

Referencias

[1] C. E. R. Alves, E. N. Caceres and S. W. Song, A BSP/CGM algorithm for the all-substrings longest common subsequence problem, In: *Proceedings of the seventeenth International Symposium on Parallel and Distributed Processing* (IPDPS '03), IEEE Computer Society, Washington, USA, pages 57-64, 2003.

[2] A. Apostolico, String editing and longest common subsequences, In: *Handbook of formal languages, linear modeling: background and application*, Springer-Verlag, New York, USA, pages 361-398, 1997.

[3] S. Arikawa, M. Hirao, H. Hoshino, A. Shinohara and M. Takeda, A practical algorithm to find the best subsequences patterns, *Theoretical Computer Science*, 292(2):465-479, 2003.

[4] A. N. Arslan and O. Egecioglu, Algorithms for the Constrained Longest Common Subsequence Problems, *International Journal of Foundations of Computer Science*, 16:1099-1109, 2005.

[5] C. Avendaño, C. Feregrino and G. Navarro, Búsqueda Aproximada Directamente en Texto Comprimido. Technical Report, Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE), México, 2004.

[6] B. S. Baker and R. Giancarlo, Longest Common Subsequence from Fragments via Sparse Dynamic Programming, In: *Proceedings of the sixth Annual European Symposium on Algorithms*, Springer-Verlag, Venice, Italy, pages 79-90, 1998.

[7] J. Bentley and D. Mcilroy, Data Compression using Long Common Strings, In: *Proceedings Data Compression Conference*, Washington, USA, pages 287-295, 1999.

[8] L. Bergroth, H. Hakonen and T. Raita, A Survey of Longest Common Subsequence Algorithms, In: *seventh International Symposium on String Processing Information Retrieval* (SPIRE'00), IEEE Computer Society, Washington, USA, pages 39-48, 2000.

[9] L. Bergroth, H. Hakonen and T. Raita, New Approximation Algorithms for Longest Common Subsequences, In: *String Processing and Information Retrieval* (SPIRE'98). IEEE Computer Society, Washington, USA, 1998.

[10] P. Bonizzoni, G. Della, R. Dondi, G. Fertin and S. Vialette, Exemplar Longest Common Subsequence, In: *Proceedings of second International Workshop on Bioinformatics Research and Applications* (IWBRA 2006), University of Reading, UK, pages 791-798, 2006

[11] P. Bonizzoni, G. Della and G. Mauri, Experimenting an Approximation Algorithm for the LCS, *Discrete Mathemathics*, 110(1):13-24, 2001.

[12] H. S. Booth, S. F. MacNamara, O. M. Nielsen and S. R. Wilson, An Iterative Approach to the Longest Common Subsequence, *Methodology and Computing in Applied Probability*, 6(4):401-421, 2003.

[13] E. Breimer, M. Goldberg and D. Lim, A learning algorithm for the longest common subsequence problem. *Journal of Experimental Algorithmics*, 8(2):147-156, 2003.

[14] G. Brodal, K. Kaligosi, I. Katriel and M. Kutz, Faster Algorithms for Computing Longest Common Increasing Subsequences, In: *Proceedings Combinatorial Pattern Matching*, seventeenth annual symposium, Springer-Verlag, Barcelona, España, pages 330-341, 2006.

[15] K. Chao, C. Huang and I. Yang, A fast algorithm for computing a longest common increasing subsequence, *Information Processing Letters*, 93:249-253, 2005.

[16] C. Charras, T. Lecroq, and J. D. Pehoushek, A Very Fast String Matching Algorithm For Small Alphabets and Long Patterns, In: *Proceedings of the nineth annual symposium on Combinatorial Pattern Matching*, Springer-Verlag, NJ, USA, pages 55-64, 1998.

[17] M. Crochemore, C. S. Iliopoulos, G. Navarro, Y. Pinzon and A. Salinger, Bit-parallel (□, □)-Matching and Suffix Automata, In: *Proceedings tenth International Symposium on String Processing and Information Retrieval* (SPIRE'03), Manaus, Brazil, pages 211-223, 2003.

[18] M. Crochemore and T. Lecroq, Pattern Matching and Text Compression Algorithms, In: *The Computer Science and Engineering Handbook*, CRC Press, Boca Raton, FL, USA, pages 1-49, 2003.

[19] V. Dancik, Common Subsequences and Supersequences and Their expected Length, *Combinatorics, Probability & Computing*, 7(4):365-373, 1998.

[20] V. Dancik, Expected Length of Longest Common Subsequences, PhD thesis, University of Warwick, UK, 1994.

[21] V. Dancik and M. S. Paterson, Longest Common Subsequences, In: *Proceedings of the nineteenth International Symposium on Mathematical Foundations of Computer Science*, Springer-Verlag, London, UK, pages 127-142, 1994.

[22] S. Deorowicz, Fast algorithm for constrained longest common subsequence problem, Technical Report, Silesian University of

Technology, Poland, 2005.

[23] G. Dowling and P. May, Approximate String Matching, *Communications of the ACM*, 12(4):381-402, 1980.

[24] N. Francois, Alignement, sequence consensus, recherche de similarites:complexite et approximabilite, PhD thesis, Universite Montpellier II, France, 2005.

[25] R. Greenberg, Bounds on the Number of Longest Common Subsequences, The Computing Research Repository, Technical Report, cs.DM/0301030, pages 1-13, 2003.

[26] R. Greenberg, Computing the Number of Longest Common Subsequences, The Computing Research Repository, Technical Report, cs.DS/0301034, pages 1-3, 2003.

[27] R. Greenberg, Fast and Simple Computation of All Longest Common Subsequences, Technical Report, Loyola University, USA, 2002.

[28] P. Heckel, A technique for isolating differences between files, *Communications of the ACM*, 21(4):264-268, 1978.

[29] D. S. Hirschberg, A Linear Space Algorithm for Computing Maximal Common Subsequences, *Communications of the ACM*, (18):341-343, 1975.

[30] C. S. Iliopoulos and M. Rahman, New efficient algorithms for LCS and constrained LCS problem, In: *Proceedings of the third Workshop on Algorithms and Complexity in Durham* (ACiD), Durham, UK, September 2007.

[31] C. S. Iliopoulos and M. Rahman, Algorithms for Computing Variants of the Longest Common Subsequence Problem (Extended Abstract), In: *International Symposium on Algorithms and Computation* (ISAAC), Kolkata, India, pages 399-408, 2006.

[32] D. Kunkle, Empirical Complexities of Longest Common Subsequence Algorithms, Technical Report, Rochester Institute of Technology, NY, USA, 2002.

[33] S. Kumar and E. Spafford, A pattern-matching model for intrusion detection, In: *Proceedings seventeenth National Computer*

Security Conference, Baltimore, USA, pages 11-21, 1994.

[34] H. Lin and M. Lu, An Optimal Algorithm for the Longest Common Subsequence Problem, In: *Proceedings of the third IEEE Symposium on Parallel and Distributed Processing*, Los Alamitos, California, pages 630-639, 1991.

[35] M. Lu and C. S. Rahman, A Partition Approach to Find the Length of the Longest Common Subsequence, In: *sixth International Conference on VLSI Design*, Bombay, India, pages 31-36, 1993.

[36] G. Navarro, A Guided Tour To Approximate String Matching, *ACM Computing Surveys*, 33(1):31-88, 2001.

[37]K. Ning, H. K. Ng and L. H. Wai, Finding Patterns in Biological Sequences by Longest Common Subsequences and Shortest Common Subsequences, In: *sixth IEEE Symposium on BioInformatics and BioEngineering* (BIBE), Arlington, Virginia, USA, 2006.

[38] C. Rick, New Algorithms for the Longest Common Subsequence Problem, Technical Report 85123-CS, University of Bonn, Germany, 1994.

[39] D. Sankoff, Matching Sequences under Deletion/Insertion Constraints, *National Academy of Sciences of the USA*, 69:4-6, 1972.

[40] V. S. Singh, H. Wang and R. A. Walker, Solving The Longest Common Subsequence (LCS) Problem Using The Associative Asc Processor With Reconfigurable 2d Mesh, In: *International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, Nevada, USA, 2006.

[41] J. M. Steele, Long common subsequences and the proximity of two random strings, *SIAM Journal on Applied Mathematics*, 14(4):731-737, 1982.

[42] A. Tiskin, All semi-local longest common subsequences in subquadratic time, In: *Computer Science Symposium*, St. Petersburg, Rusia, 2006.

[4] Z. Tronicek, Searching subsequences, PhD thesis, Czech Technical University, Prague, Czech Republic, 2001.

[44] Z. Tronicek, Problems Related to Subsequences and Supersequences, In: *String Processing and Information Retrieval Symposium* (SPIRE)/*International Workshop on Groupware* (CRIWG), pages 199-205, Cancun, Mexico, 1999.

[45] L. Wei and C. Ling, A sity, USA, 2002.