Subsecuencia Común más Larga en Múltiples Secuencias mediante Medidas de la Información

Ross Mary Sáenz Lesmes

Universidad Nacional de Colombia
Facultad de Ciencias
Departamento de Matemáticas
Bogotá D.C, Colombia
2018

Subsecuencia Común más Larga en Múltiples Secuencias mediante Medidas de la Información

Ross Mary Sáenz Lesmes

Trabajo final de Maestría de grado presentado como requisito parcial para optar al título de:
Magister en Ciencias - Matemáticas Aplicadas

Director:
Profesor Humberto Sarria Zapata

Universidad Nacional de Colombia
Facultad de Ciencias
Departamento de Matemáticas
Bogotá D.C, Colombia
2018

A mi familia, especialmente a mi papá y mi pequeño.

Resumen

El problema de la Subsecuencia Común más Larga de Múltiples Secuencias (SCLM), ha sido ampliamente estudiado en Ciencias de la Computación desde hace más de 40 años, motivado principalmente por sus diversas aplicaciones en Bioinformática. Este problema consiste en calcular una subsecuencia de longitud máxima, común a un conjunto de secuencias dado. En este trabajo se propone un algoritmo heurístico capaz de aproximar una o varias soluciones al problema SCLM utilizando la entropía de Shannon como una medida de la información para determinar los alineamientos que generen las mejores aproximaciones a la solución del problema.

Palabras Clave: Algoritmo, heurístico, secuencia, subsecuencia común más larga, SCLM, entropía, alineamiento.

Abstract

The problem of the Longest Common Multiple Sequence (MLCS), has been widely studied in Computer Science for more than 40 years, mainly motivated by its diverse applications in Bioinformatics. This problem consists in calculating a subsequence of maximum length, common to a set of given sequences. In this paper we propose a heuristic algorithm able to approximate one or several solutions to the MLCS problem using the Shannon entropy as a measure of the information to determine the alignments that generate the best approximations to the solution of the problem.

Key words: Algorithm, heuristic, sequence, longest common subsequence, MLCS, entropy, alignment.

Índice general

Introducción

El problema de encontrar una subsecuencia común más larga de una colección de secuencias dadas es uno de los problemas más interesantes y actuales de las Ciencias de la Computación; su solución tiene aplicaciones en: Bioinformática, para la alineación de secuencias de ADN; comparación de archivos, para determinar plagio; detección de archivos, para identificar múltiples archivos defectuosos pero con información común; búsqueda de patrones y contenidos en la Red (Carlos et al., 2004; Ning et al., 2006; Huang et al., 2004; Bozkaya et al., 1997; Alfred, 2014).

En este trabajo se denota (SCL) para referirse al caso particular de colecciones conformadas únicamente por un par de secuencias y (SCLM) para tres o más secuencias, conocido también como Subsecuencia Común Más Larga de Múltiples Secuencias.

El problema SCL se ha estudiado y desarrollado desde hace más de 40 años. En 1974, Wagner y Fischer (Wagner and Fischer, 1974) presentaron una solución al problema para pares de secuencias utilizando Programación Dinámica con una complejidad de $O(mn)$, donde m y n son las longitudes de las secuencias involucradas. Posteriormente, Daniel S. Hirschberg (Hirschberg, 1977) propuso una solución con una complejidad de $O(pn+nlogn)$, donde n es la longitud de la secuencia de entrada más grande y p es la longitud de la subsecuencia común más larga. Se han propuesto otras alternativas de solución de este problema que hacen uso de Autómatas Finitos, Puntos Dominantes, entre otros (Forero, 2010; Iliopoulos and Rahman, 2008; Gorbenko and Popov, 2012).
En 1984, W. Hsu y M. Du, formularon un algoritmo para hallar la solución al problema SCL en un conjunto de $k-$secuencias con longitudes $l_1, l_2, \cdots, l_k$ basado en Programación Dinámica con tiempo de ejecución $O(kl_1l_2 \cdots l_k)$, pero debido a su alto costo computacional se hace inviable para las aplicaciones (Hsu and Du, 1984).

En 1992, Koji Hakata e Hiroshi Imai (Hakata and Imai, 1992) propusieron un algoritmo para hallar una solución a SCLM en una colección de secuencias de caracteres basado en punto dominante, restringiendo la dimensión del alfabeto. En 1998, plantearon una aproximación a la solución de SCLM empleando Programación Dinámica, Punto Dominante y Geometría Máxima (Hakata and Imai, 1998).
Otros métodos de solución se han planteado, basados en punto dominante y paralelismo, tal es el caso del propuesto por Dmitry Korkin, quien dió inicio a esta técnica en 2001 (Korkin, 2001). Así mismo, estudios recientes evidencian estrategias conjuntas, por ejemplo, en Yanni Li, Hui Liy Tihua Duan(Li et al., 2016).

Un problema subyacente a SCL y SCLM es el alineamiento de secuencias (Bereg and Zhu, 2005). De igual forma, éste se ha estudiado para pares y múltiples secuencias mediante diversas técnicas. En el caso específico de pares de secuencias, el algoritmo más destacado es el Neddleman-Wunch (Needleman and Wunsch, 1970; Flouri et al., 2015), el cual genera alineamientos óptimos de forma global y usa Programación Dinámica. En la actualidad éste se sigue implementando en Bioinformática (Fakirah et al., 2015; Feng and Gao, 2015; Boes, 2014). Para el caso de alineamiento múltiple existen una variedad de métodos de aproximación a la solución, sin embargo los más implementados son los iterativos y los progresivos (Löytynoja and Goldman, 2005; Löytynoja and Milinkovitch, 2003; Wallace et al., 2005).

Entre los desarrollos de Software más importantes para alineamientos de pares y múltiples secuencias están: BLAST (Basic Local Alignment Search Tool); FLAST (búsqueda local de k-tuplas); ClustalW, MUSCLE, T-Coffee, entre otros.

En este sentido, el propósito de este trabajo consiste en abordar el problema de la subsecuencia común más larga, para colecciones con dos o más secuencias haciendo uso de una medida de la información conocida como la Entropía de Shannon, con el fin de alinear secuencias mediante el cálculo de la entropía de los caracteres. Esta técnica no ha sido utilizada en ningún estudio previo, por lo tanto este trabajo presenta una forma original de abordar el problema.

Es preciso señalar que este problema es de complejidad NP-Hard (Maier, 1978; Blin et al., 2012), por tal razón en este trabajo se plantea un algoritmo heurístico que permite calcular

en tiempo razonable, una solución aproximada al problema.

El documento se encuentra organizado de la siguiente manera, el capítulo 1, contiene algunas definiciones importantes como son, subsecuencia, subsecuencia común más larga, alineamiento y entropía. El capítulo 2, contiene algunos de los métodos más importantes desarrollados hasta el momento, para aproximar una solución al problema de la subsecuencia común más larga. En el capítulo 3, presentamos el algoritmo por etapas, cada una de estas etapas contiene el seudo-código en lenguaje Haskell y algunos ejemplos ilustrativos. En el capítulo 4, calculamos el orden de la complejidad del algoritmo, y exponemos algunos ejemplos, junto con algunas pruebas de rendimiento del algoritmo. Finalmente, planteamos algunas conclusiones y recomendaciones acerca del algoritmo.

1 Preliminares

En esta sección se presentan algunas definiciones y conceptos necesarios para el desarrollo de este trabajo. Se describe formalmente las nociones de subsecuencia, subsecuencia común más larga entre dos o más secuencias, el significado de alineamiento y alineamiento múltiple y por último la definición de entropía como una medida de la información.

1.1. Subsecuencia Común Más Larga (SCL)

Un *alfabeto* es un conjunto no vacío y finito, cuyos elementos son símbolos llamados letras, el cual se denota con Σ, el cardinal del alfabeto Σ se representa por $|\Sigma|$.

Una *secuencia* sobre Σ es un arreglo ordenado de elementos de Σ con longitud finita. La cadena vacía o nula es aquella que no contiene símbolos y se representa por ε. La longitud de una secuencia A se denota por $|A|$ y está definida por:

$$|A| = \begin{cases} n, & A = a_1 a_2 ... a_n \\ 0, & A = \varepsilon \end{cases}$$

Por ejemplo, sea $\Sigma = \{A, B, C, D\}$, algunas secuencias de este alfabeto son: $AABCD$, AA, $DBAC$, cuyas longitudes son, 5, 2, y 4 respectivamente.

Sean $A = a_1 a_2 ... a_n$ y $C = c_1 c_2 ... c_r$ dos secuencias sobre un alfabeto Σ, C es una *subsecuencia* de A, si $r \leq n$ y existe una función monótona creciente $F : [r] \to [n]$, donde $[n] = \{1, 2, \cdots, n\}$ y $[r] = \{1, 2, \cdots, r\}$, tal que $F(i) = k$ si $c_i = a_k$ para $k \in [n]$. En otras palabras, una subsecuencia C es un subconjunto ordenado de caracteres de la secuencia A no necesariamente consecutivos.

Así mismo, sean A y B dos secuencias sobre el alfabeto Σ, C es una *subsecuencia común* de A y B si C es subsecuencia de A y B. Se dice que C es una subsecuencia común más larga de A y B, si $|C| \geq \left|C'\right|$ para toda subsecuencia común C' de A y B.

1.2. Subsecuencia Común más Larga de Múltiples Secuencias (SCLM)

En general, dada una colección de secuencias $\{s_1, s_2, \cdots, s_n\}$ con longitudes $\{l_1, l_2, \cdots, l_n\}$ respectivamente sobre un alfabeto finito Σ, una secuencia común de esta colección es una subsecuencia C sobre el mismo alfabeto Σ, común a cada s_r para todo $r \in \{1, 2, \cdots, n\}$. C es una subsecuencia común más larga, si C tiene longitud máxima.

Por ejemplo, dadas las secuencias $s_1 = CGCGTA$, $s_2 = AGCCGTAC$ y $s_3 = AGTCCGT$, sobre $\Sigma = \{A, C, G, T\}$. Las subsecuencias comunes de mayor longitud de la colección $\{s_1, s_2, s_3\}$, pertenecen al conjunto: $SCLM(s_1, s_2, s_3) = \{CCGT, GCGT\}$.

1.3. Alineamiento

De manera informal un *alineamiento* entre un par de secuencias de caracteres es una correspondencia entre los caracteres de las secuencias dadas, de tal manera que se presente cierta cantidad de coincidencias, por ejemplo, una alineación entre $\{abcd, acc\}$:

$$\begin{array}{ccccc} a & b & c & d & - \\ a & - & c & - & c \end{array} \tag{1.1}$$

Los espacios o huecos presentes en el alineamiento se denominan gaps y se representan por el símbolo " $-$ " o " ".

Otro alineamiento podría ser:

$$\begin{array}{cccccc} - & - & a & b & c & d \\ a & c & - & - & c & - \end{array}$$

Sin embargo, el mejor alineamiento posible es el dado en (1.1), ya que éste tiene la mayor cantidad de coincidencias. De manera formal, un alineamiento se define de la siguiente forma. Sea $A = a_1a_2...a_n$ y $B = b_1b_2...b_r$ secuencias definidas en Σ, donde $r \leq n$. Un alineamiento entre la secuencia A y B es una matriz $M_{2\times m}$ con elementos en el alfabeto $\Sigma \cup \{-\}$, donde $n \leq m \leq r + n$, tal que:

- La primera fila de la matriz M contiene los caracteres de A y la segunda los de B en orden (por conveniencia se toma en la primera fila la secuencia con mayor longitud).
- Cada columna de la matriz contiene por lo menos un símbolo del alfabeto Σ (Orlova, 2010). Es decir, las columnas de la matriz están formadas por parejas de la forma: $\begin{pmatrix} a_i \\ - \end{pmatrix}$, $\begin{pmatrix} - \\ b_j \end{pmatrix}$ y $\begin{pmatrix} a_i \\ b_j \end{pmatrix}$ si $a_i = b_j$, donde $1 \leq i \leq n$ y $1 \leq j \leq r$.

Vale la pena señalar que para el ejemplo en (1.1), si no se hubiesen introducido gaps, no existiría un alineamiento entre las dos secuencias. En general, los gaps permiten una mejor distribución de los caracteres con el fin de que el alineamiento contenga el mayor número de coincidencias.

En términos generales, *un alineamiento múltiple* corresponde al alineamiento simultáneo de tres o más secuencias. Es decir, el alineamiento para $\{s_1, s_2, \cdots, s_n\}$ sobre un alfabeto Σ es una matriz $M_{n\times m}$, de modo que:

- Cada entrada de la matriz M pertenece a $\Sigma \cup \{-\}$.
- m cumple con $max\{|s_1|, |s_2|, \cdots, |s_n|\} \leq m \leq \Sigma_{i=1}^{n}|s_i|$.
- Toda columna de M contiene por lo menos un símbolo de Σ.
- La fila $k-$ésima de la matriz M contiene los caracteres ordenados de una secuencia s_k, separados por g_k gaps, donde $g_k = max\{|s_1|, |s_2|, \cdots, |s_n|\} - |s_k|$, donde $1 \leq k \leq n$.

Se define *un alineamiento múltiple de* $\{s_1, s_2, \cdots, s_n\}$ *con g gaps*, como un alineamiento múltiple de $\{s_1, s_2, \cdots, s_n\}$ con $m = l_{max}$, donde $l_{max} = max\{|s_1|, |s_2|, \cdots, |s_n|\} + g$.

Por ejemplo, un alineamiento múltiple entre el conjunto de secuencias $\{abcd, acc, acd, ad\}$ con 1 gap, está dado por:

$$\begin{array}{ccccc} a & b & c & d & - \\ a & - & c & - & c \\ a & - & c & d & - \\ a & - & - & d & - \end{array}$$

Note que 1 gap le corresponde a la secuencia de mayor longitud, es decir a la secuencia *abcd*, 2 gaps para *acc* y *acd*, 3 gaps para *ad*.

1.4. Entropía

A continuación se define la función de entropía de Shannon como una medida de la información, ésta mide el grado de desorden o incertidumbre que se genera en los resultados de un experimento cualquiera.

Sea X una variable aleatoria con distribución de probabilidad $P(X)$, si X toma valores en $\{x_1, x_2, \cdots, x_n\}$, la entropía $H(X)$ está definida por:

$$H\left(X\right) = -\sum_{i=1}^{n} p\left(x_i\right) log_2 p\left(x_i\right)$$

(Yeung, 2012; Cover and Thomas, 2012)

2 Estado del Arte

La subsecuencia común más larga para pares y múltiples secuencias es un problema que ha sido estudiado desde distintos enfoques, para el caso particular de dos secuencias se han propuesto algoritmos bastante eficientes que resuelven el problema en tiempo polinomial, sin embargo, para el caso de múltiples secuencias la gran mayoría de algoritmos existentes están constituidos por métodos heurísticos que con cierto grado de certeza aproximan una solución. Los esfuerzos por generar alternativas de solución a este problema, han sido motivados por las diversas e interesantes áreas de aplicación, principalmente en Bioinformática para el análisis de secuencias biológicas.

2.1. Técnicas de Solución

Algunas de las técnicas más utilizadas en la solución al problema de la subsecuencia común más larga para pares de secuencias son: punto dominante, autómatas finitos, y divide y vencerás, basados en programación dinámica.

2.1.1. Programación dinámica

Dadas dos secuencias de caracteres $X = x_1 \dots x_n$, $Y = y_1 \dots y_m$, los algoritmos existentes para calcular la subsecuencia común más larga de X y Y utiliza la técnica de programación dinámica como método para reducir los tiempos de ejecución mediante el almacenamiento de una matriz de puntuaciones L de tamaño $(m+1) \times (n+1)$, donde n y m son las longitudes de las secuencias X y Y respectivamente. Los elementos $L(i, j)$ de la matriz

están definidas por los valores de las posiciones vecinas anteriores, es decir, el valor de cada entrada depende de las entrada izquierda, superior y superior izquierda. Las puntuaciones de la matriz se obtienen de la siguiente manera:

$$L(i,j) = \begin{cases} 0 & i = 0 \ o \ j = 0 \\ L(i-1, j-1) + 1 & x_i = y_j \\ \text{máx}\{L(i-1,j), L(i,j-1)\} & x_i \neq y_j \end{cases} .$$

La longitud de la subsecuencia común más larga, estará determinada por el valor contenido en la posición inferior derecha de L, es decir, $|SCL(X,Y)| = L(m,n)$.

2.1.2. Punto Dominante

El método de punto dominante fue introducido por Hirschberg en 1977 y ha sido sucesivamente implementado, no solo en el caso de SCL sino también en SCLM. En 1998, Hakata e Imai diseñaron un algoritmo basado en punto dominante para el caso específico de tres secuencias con una complejidad de $\mathcal{O}(ns + Ds\log(s))$, donde n es la longitud de las secuencias, s es el cardinal del alfabeto y D es el número de puntos dominantes (Hakata and Imai, 1992). Posteriormente este método fue extendido para un conjunto arbitrario de secuencias (Hakata and Imai, 1998).

Intuitivamente, un punto dominante de la matriz de puntuaciones L es una entrada cuyo valor es estrictamente mayor a todos sus vecinos anteriores. Por ejemplo (Wang et al., 2009), para las secuencias $X = GTAATCTAAC$, $Y = GATTACA$, la matriz de puntuaciones L se muestra en la Figura 2.1, los puntos dominantes de la matriz son señalados por círculos, mientras que los puntos señalados por cuadrados son algunos puntos no dominantes.

Para construir la subsecuencia común más larga recorremos la matriz L desde la parte inferior derecha, posición (m,n) hasta $(0,0)$ pasando por los puntos dominantes sucesivos de manera que se logre el camino más largo. Así, de la Figura 2.2 podemos inferir

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

Figura 2.1: Puntos dominantes y puntos no dominantes de L

que las subsecuencias comunes más largas son $GATTAC$ o $GATTAA$, observe que la $|SCL(GTAATCTAAC, GATTACA)| = L(7, 10) = 6$.

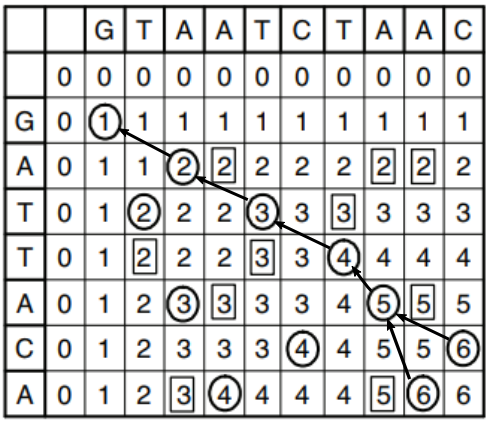

Figura 2.2: Construcción de la subsecuencia común más larga

2.1.3. Autómatas finitos

El método de Autómatas finitos se basa en la matriz de puntuación L que genera la técnica de programación dinámica, representada como un grafo, de tal forma que los nodos son las celdas de la matriz, los arcos representan las operaciones, inserción, eliminación y reemplazo.

Un grafo acíclico directo es una clase de autómata finito que representa los sufijos de una secuencia dada, en los cuales cada arco esta etiquetado con un carácter o símbolo. Los caracteres o símbolos a lo largo de una ruta desde la raíz a un componen la subsecuencia cuyo nodo representa. En esta técnica se busca la ruta más corta ubicada entre la posición $(0, 0)$ y la posición (m, n) de la matriz L. El costo total de la ruta más corta en este grafo, representa el mínimo número de operaciones necesarias para transformar la secuencia X en la Y y la longitud de la subsecuencia más larga es el número de coincidencias o diagonales del grafo (Soto and Pinzón, 2010).

Por ejemplo (Jones and Pevzner, 2004), la Figura 2.3 muestra el grafo para las secuencias ATCGTAC, ATCGTAC. Las flechas oscuras muestran la ruta más corta entre la posición $(0, 0)$ y $(7, 7)$. Observe que el número de arcos diagonales indican las coincidencias de las secuencias. Luego, $|SCL(ATCGTAC, ATCGTAC)| = 5$.

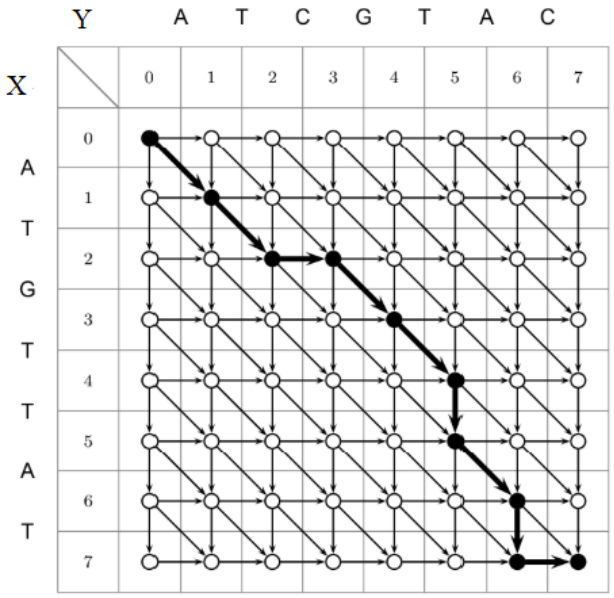

Figura 2.3: Grafo

2.1.4. Aproximaciones para SCLM

La gran mayoría de los métodos que aproximan la solución para el problema SCLM son de tipo heurístico, dentro de los métodos implementados con mejores resultados están los algoritmos en paralelo basados en programación dinámica y punto dominante. Los métodos en paralelo particionan los datos iniciales y distribuyen el algoritmo mediante procesos o capaz, generando al final una combinación o sincronización en cada una de ellas. Este algoritmo mejora la velocidad de computo, sin embargo, la sincronización de cada capa hace que pierda eficiencia. Algunos de los algoritmos más reconocidos con esta técnica se muestran en el siguiente cuadro.

Año	**Autor**	**Complejidad**
2001	Dmitry Korkin	$\mathcal{O}(nsd + \|D\|sd(\log^{d-3} n + \log^{d-2} s))$
2006	Yixin Chen, Andrew Wan and Wei Liu	$\mathcal{O}(\|SCL(X_1, X_2, ..., X_n)\|)$
2009	Qingguo Wang, Dmitry Korkin and Yi Shang	$\mathcal{O}(n \cdot \|\Sigma\| \cdot d + \|D\| \cdot \|\Sigma\| \cdot d \cdot (\log^{d-2} n + \log^{d-2} \|\Sigma\|))$
2010	Qingguo Wang, Mian Pan, Yi Shang and Dmitry Korkin	$\mathcal{O}(dn^2 + k\|\Sigma\|dn)$
2016	Yanni Li, Hui Li, Tihua Duan, Sheng Wang, Zhi Wang and Yang Cheng	$\mathcal{O}(d\|\Sigma\|n + d\|V\| + \|E\|)$

Cuadro 2.1: Algoritmos en paralelo

2.2. Problemas cercanos

Algunos problemas de interés en Ciencias de la Computación semejantes a SCL y no menos importantes son: subsecuencia común más larga restringida, subsecuencia creciente más larga, supersecuencia común más corta, alineamiento de dos o más secuencias. Veamos en qué consiste grosso modo cada uno de ellos. El problema de la subsecuencia común más larga restringida respecto a la secuencia P, consiste en buscar una subsecuencia común

más larga de dos secuencias X, Y, tal que P sea subsecuencia de ésta. Los algoritmos que dan solución a este problema se basan principalmente en programación dinámica, algunos de los algoritmos más relevantes son, Yin-Te Tsai con una complejidad de $\mathcal{O}(m^2 \cdot n^2 \cdot r)$, Peng Chao-Li en $\mathcal{O}(m \cdot n \cdot r)$, Wan Wei-Lun en $\mathcal{O}(m \cdot n \cdot r)$, donde n, m son las longitudes de las secuencias X y Y respectivamente y r es la longitud de la secuencia P (Soto and Pinzón, 2010).

La subsecuencia común creciente más larga, es un problema que consiste en hallar una subsecuencia creciente con longitud máxima para una secuencia A de n enteros positivos. Diferentes alternativas de solución a este problema se han propuesto, tal es el caso de I-Hsuan Yang, Chien-Pin Huang, Kun-Mao Chao, quienes propusieron en 2005 un algoritmo en $\mathcal{O}(mn)$, donde m, n son las longitudes de las secuencias (Yang et al., 2005) y de Christophe Cérin, Catherine Dufourd, Jean-Fréderic Myoupo, quienes en 1993 plantearon un método en paralelo con una complejidad de $\mathcal{O}(n)$ (Cérin et al., 1993).

El problema de la supersecuencia común más corta, consiste en hallar una secuencia de longitud mínima para un conjunto de secuencias A, tal que cada secuencia en A sea subsecuencia de ésta, este problema es NP-Completo. A pesar de que este problema es semejante al problema SCLM, los algoritmos que se han propuesto en este sentido no son numerosos, algunos de ellos se encuentran en, (Turner, 1989), (Irving and Fraser, 1993), (Fraser and Irving, 1995).

El alineamiento de dos secuencias, es un problema estudiado desde inicios de los años 1970 para comparar o analizar las similitudes de las secuencias . La solución se basa en programación dinámica mediante el algoritmo de Neddleman-Wunch, éste determina el alineamiento con la mayor cantidad de coincidencias posibles de dos secuencias y es implementado para el caso de alineamientos globales, el tiempo de ejecución es de $\mathcal{O}(n \cdot m)$, donde n, m son las longitudes de las secuencias. Por otro lado, el alineamiento es un problema que ha sido generalizado para múltiples secuencias, el cual consiste en optimizar la cantidad de coincidencias en la alineación para un conjunto de secuencias dado. Los métodos que solucionan este problema se clasifican en exhaustivos y heurísticos o aproximados. Los métodos exhaustivos para alinear un conjunto de $n-$ secuencias tienen una complejidad del orden de $\mathcal{O}(2^n \prod_{i=1}^{n} |s_i|)$, por consiguiente se utilizan para un número reducido de secuencias,

regularmente menores de diez. Por otro lado, los métodos aproximados se clasifican en progresivos e iterativos. Dentro de los métodos progresivos más conocidos están ClustalW y el T-Coffe. Mientras que el método iterativo más popular es el Muscle.

3 Algoritmo

A continuación se presenta un algoritmo heurístico que genera una aproximación a la subsecuencia común más larga de una colección de secuencias S. El algoritmo tiene varias etapas, en seguida se describe de manera breve cada una de ellas, posteriormente se ahondará en los detalles.

1. Dada una colección de secuencias $S = \{s_1, s_2, \cdots, s_n\}$ y un número de gaps g, se introduce un número de gaps g_k para cada s_k en S, donde $g_k = max\{|s_1|, |s_2|, \cdots, |s_n|\} + g - |s_k|$, con $1 \leq k \leq n$. A partir de la cantidad de gaps que se introducen en cada secuencia s se genera una nube de secuencias para cada s en S.

2. Mediante ciertas funciones se cuenta el número de secuencias que tienen un símbolo $\mathbf{x}$ en la posición i, para $1 \leq i \leq max\{|s_1|, |s_2|, \ldots, |s_n|\} + g$, entre todas las secuencias de la nube que se forman con cada secuencia y su respectivo número de gaps.

3. Con este conteo se construye una matriz de frecuencias de aparición de cada símbolo en una posición determinada. Esta matriz proporciona información sobre la distribución de los caracteres de las secuencias en S.

4. De acuerdo con la matriz de frecuencias determinada en el paso anterior, se construye una colección de Secuencias Patrón P constituidas por los símbolos con mayor frecuencia de aparición por posición.

5. Utilizando la matriz de frecuencias se calcula la entropía de cada símbolo y cada posición.

6. Se alinea cada secuencia s en S con cada secuencia patrón p en P y se escoge como mejor aproximación las subsecuencias resultantes comunes a toda secuencia s en S.

En cada una de las etapas del algoritmo se incluye una implementación en lenguaje de programación Haskell. Este lenguaje se caracteriza por ser un lenguaje puramente funcional es decir, efectúa todas sus operaciones mediante evaluación de funciones matemáticas que evitan el cambio de estado y la mutación de variables. Además, tiene funciones para manejar secuencias de caracteres de forma ágil y eficiente.

3.1. Nube de secuencias y combinatoria

A continuación se presenta por medio de un ejemplo cómo se genera la nube de secuencias por cada secuencia en la colección. Para comenzar se analiza la siguiente situación: dada una secuencia $s = a_1 a_2 \dots a_l$. ¿ Cuáles son todas las formas posibles de organizar los caracteres $a_1, a_2, \dots, a_l$ en $l + g$ lugares, de modo que los caracteres preserven el orden de aparición en s?.

El Cuadro 3.1 muestra todas las formas posibles de organizar la secuencia *aba* con dos gaps.

1	**2**	**3**	**4**	**5**
a	b	a		
a	b		a	
a		b	a	
a	b			a
a			b	a
a		b		a
	a	b	a	
	a	b		a
	a		b	a
		a	b	a

Cuadro 3.1: Nube de secuencias de *aba* con 2 gaps.

En general, una matriz que contiene todas las formas posibles de organizar la secuencia s

con g gaps se denomina *nube de secuencias de s con g gaps*. Observe que el número de formas de organizar los caracteres $a_1, a_2, \ldots, a_l$ en $l+g$ lugares está dado por $\binom{l+g}{g}$. En particular, el ejemplo del Cuadro 3.1 tiene $\binom{5}{2} = 10$ formas distintas.

Por otro lado, es necesario saber cuántas secuencias en la nube de la secuencia s con g gaps, tienen al elemento a_j en la posición i, con $1 \leq i \leq l+g$. La respuesta a esta pregunta viene dada por la siguiente función:

$$comb(l, g, i, j) = \left\{ \begin{array}{cc} \binom{i-1}{j-1} \cdot \binom{l+g-i}{l-j} & \text{si } i \geq j \ , \ g+j \geq i \\ 0 & \text{en otro caso} \end{array} \right\}.$$

Para el ejemplo del Cuadro 3.1, el número de secuencias que contienen al elemento $a_2 = b$ en la posición 3 de la nube de secuencias es $\binom{2}{1}\binom{2}{1} = 4$. El siguiente código en Haskell implementa la función anterior:

```
fcomb  :: Int -> Int -> Int -> Int -> Int
fcomb  l g  j i = if(i>=j && g+j>= i) then (binom (i-1) (j-1))*(binom a b)
                  else 0
                  where (a, b) =  (l+ g -i, l-j)
```

Observe que puede darse el caso en que algunos caracteres de la secuencia s correspondan al mismo símbolo, entonces es natural preguntarse, ¿Cuántas secuencias de la nube de secuencias de s con g gaps, tienen el símbolo $\mathbf{x}$ en la posición i?. Para resolver esta pregunta utilizamos la función *comb* en cada una de las posiciones donde aparezca el símbolo $\mathbf{x}$ en la secuencia s. En otras palabras, si $a_{j_1}, a_{j_2}, \ldots, a_{j_c}$, con $1 \leq j_1 < j_2 \cdots < j_c \leq l$, los caracteres de s con símbolo $\mathbf{x}$, entonces el número de secuencias en la nube de s con g gaps que tienen el símbolo $\mathbf{x}$ en la posición i, está determinado por la siguiente función:

$$combAcumulada(l, g, \mathbf{x}, i) := \sum_{t=1}^{c} comb(l, g, j_t, i).$$

El código que implementa esta función es:

```
--Devuelve las posiciones de s donde se encuentra el simbolo x
posiciones :: Eq a => a -> [a] -> [Int]
```

```
posiciones x s = [i | (x',i) <- zip s [0..n], x == x']
           where n= length s - 1

{- Calcula la sumatoria de com(l,g,j,i) sobre las
posiciones de s donde aparece el símbolo x -}
combAcumulada :: Int -> Int  -> Int -> Int
combAcumulada  l g x i =sumFila [comb l g j i |j <- posiciones x s ]
```

3.2. Matriz Combinatoria de S

Sea Σ el alfabeto de símbolos en S y m el cardinal de Σ; además, sean $l_1, l_2, \ldots, l_n$ las longitudes de $s_1, s_2, \ldots, s_n$ respectivamente, se considera $l_{max} = max\{|s_1|, |s_2|, \cdots, |s_n|\} + g$. Así, para cada s en S, la *Matriz Combinatoria* de s está definida por la matriz de m filas y l_{max} columnas, cuyas filas son indexadas por los símbolos de Σ en orden alfabético, y las columnas por las posiciones 1 hasta l_{max}, de tal manera que la entrada $(\mathbf{x}, j)$ de la matriz, contiene el número de secuencias en la nube de s con $l_{max} - |s|$ gaps que tienen el símbolo $\mathbf{x}$ en la posición j, donde $1 \leq j \leq l_{max}$, es decir $combAcumulada(l, g, \mathbf{x}, j)$. Por último, se dice que la *Matriz Combinatoria* de S es la suma de las matrices combinatorias de cada s en S.

Por ejemplo, sea $S = \{abab, bcca, cab\}$ y $g = 2$, las matrices combinatorias de cada s en S se muestran a continuación.

Para $s_1 = abab$, la nube de secuencias tiene 2 gaps, luego la matriz combinatoria está dada por:

	1	**2**	**3**	**4**	**5**	**6**
a	10	4	4	6	6	0
b	0	6	6	4	4	10
c	0	0	0	0	0	0

La nube de secuencias para $s_2 = bcca$ contiene 2 gaps, luego la matriz combinatoria está dada por:

	1	2	3	4	5	6
a	0	0	0	1	4	10
b	10	4	1	0	0	0
c	0	6	9	9	6	0

De igual manera, para la secuencia $s_3 = cab$ la nube de secuencias tiene 3 gaps, luego la matriz combinatoria es:

	1	2	3	4	5	6
a	0	4	6	6	4	0
b	0	0	1	3	6	10
c	10	6	3	1	0	0

La suma de las matrices anteriores corresponde a la matriz combinatoria de $S = \{abab, bcca, cab\}$ con $g = 2$ y está dada por:

	1	2	3	4	5	6
a	10	8	10	13	14	10
b	10	10	8	7	10	20
c	10	12	12	10	6	0

Cuadro 3.2: Matriz Combinatoria

El siguiente conjunto de funciones en Haskell forman un algorítmo que determina la matriz combinatoria para una colección de secuencias S.

```
--Posiciones donde aparece el k-ésimo símbolo de xt en xs
indicesCaracter :: String -> String -> Int ->  [Int]
indicesCaracter xs xt k =  posiciones ((nub (sort xt )) !! k )  xs

--Devuelve una lista de símbolos distintos de una colección de secuencias
dConcat:: [String] -> String
dConcat xr = nub ( concat xr )
```

```
--Genera la matriz combinatoria de una secuencia S_r
matrizCombinatoria :: String -> [String] -> Int ->  Array (Int ,Int) Int
matrizCombinatoria sr s g = array ((1,1), (length (dConcat s),
((length sr)+gs))) [((i,j),  combinatoriaAcumulada  gs (length xs)
(indicesCaracter sr (dConcat s) (i-1))  (j-1) )
| i <- [1..(length (dConcat s))], j<-[1..((length sr)+gs)]]
                    where gs = longMayor s + g - length sr

--Genera la matriz combinatoria de una colección de secuencias S
matrizFinal :: [String] -> Int ->  Array (Int,Int) Int
matrizFinal s g =sumaMatrices [matrizCombinatoria sr s g | sr <- s]
```

3.3. Secuencias Patrón

Sea $M = [m_{\mathbf{x}j}]$ la matriz combinatoria de S, para una columna j de la matriz M se define la frecuencia máxima de j, $frecMax(j)$, como:

$$frecMax(j) := \max_{\mathbf{x}\in\Sigma} m_{\mathbf{x}j}.$$

Es decir, el máximo de los valores en la j-ésima columna.

Una *Secuencia Patrón* de M es una secuencia de longitud l_{max} con símbolos en el alfabeto Σ, donde el j-ésimo caracter de la secuencia corresponde a un símbolo $\mathbf{a}$, tal que $frecMax(j) = m_{\mathbf{a}j}$.

Sea $P = \{p_1, p_2, \cdots, p_w\}$ el conjunto de todas las secuencias patrón de M. Para la matriz combinatoria de el Cuadro 3.2, el conjunto de secuencias patrón es

$$P = \{accaab, bccaab, cccaab\}.$$

Observe que para el ejemplo anterior, se obtienen tres secuencias patrón, ya que la frecuencia máxima de la columna uno ocurre en los símbolos $\{a, b, c\}$.

A continuación se muestra un código que calcula el conjunto de las secuencias patrón de una matriz combinatoria M.

```
-- Devuelve los caracteres que aparecen en las posiciones dadas.
simbPosiciones :: String -> [Int] -> [Char]
simbPosiciones sr nums = [alfabeto(sr) !! (num -1)| num <- nums]

-- Devuelve los símbolos correspondientes a los valores máximos de cada columna
simbMaximos :: [String] -> Int -> [String]
simbMaximos s g =[simbPosiciones (dConcat s) t|t<-maxColumnas (matrizFinal s g)]

-- productCart "ab" ["da", "cd" ] = ["ada", "acd" , "bda" , "bcd"]
productCart :: String -> [String] -> [String]
productCart x s = [ [a] ++ b | a<- x ,  b <- s]

-- productsCart ["ab","xy"] = ["ax","ay","bx","by"]
productsCart :: [String] -> [String]
productsCart [] = error "Lista vacia"
productsCart [x] = [[a] | a <- x ]
productsCart (x:s) = productCart x y
             where y =  productsCart s

-- Calcula la colección de secuencias patrón según la frecuencia de aparición
secuenciasPatron :: [String]-> Int -> [String]
secuenciasPatron s g = productsCart(simbMaximos s g)
```

3.4. Entropía de símbolos y posiciones

Dada una matriz combinatoria M de la colección de secuencias S y un símbolo $\mathbf{x}$ en el alfabeto Σ, la entropía de un símbolo $\mathbf{x}$ se define como:

$$H(\mathbf{x}) := -\sum_{j=1}^{l_{max}} P(\mathbf{x}, j) \cdot \log_2(P(\mathbf{x}, j)).$$

Donde $P(\mathbf{x}, j)$ es la probabilidad de que dada una ocurrencia del símbolo $\mathbf{x}$ en una secuencia, ésta se encuentre en la posición j. Observe que la probabilidad $P(\mathbf{x}, j)$ está dada por la expresión $P(\mathbf{x}, j) = \frac{m_{\mathbf{x}j}}{\sum_{j=1}^{l_{max}} m_{\mathbf{x}j}}$, luego la entropía de $\mathbf{x}$ es:

$$H(\mathbf{x}) = -\sum_{j=1}^{l_{max}} \frac{m_{\mathbf{x}j}}{\sum_{j=1}^{l_{max}} m_{\mathbf{x}j}} \log_2 \left(\frac{m_{\mathbf{x}j}}{\sum_{j=1}^{l_{max}} m_{\mathbf{x}j}} \right).$$

De forma análoga, se define la entropía de la posición j, $H(j)$, como:

$$H(j) := -\sum_{\mathbf{y} \in \Sigma} \tilde{P}(\mathbf{y}, j) \cdot \log_2(\tilde{P}(\mathbf{y}, j)).$$

Donde $\tilde{P}(\mathbf{y}, j)$ representa la probabilidad de que en la posición j aparezca el símbolo $\mathbf{y}$, observe igualmente que la probabilidad $\tilde{P}(\mathbf{y}, j)$ esta dada por la expresión $\tilde{P}(\mathbf{y}, j) = \frac{m_{\mathbf{y}j}}{\sum_{\mathbf{y} \in \Sigma} m_{\mathbf{y}j}}$, por tanto la entropía quedará escrita como

$$H(j) = -\sum_{\mathbf{y} \in \Sigma} \frac{m_{\mathbf{y}j}}{\sum_{\mathbf{y} \in \Sigma} m_{\mathbf{y}j}} \log_2 \left(\frac{m_{\mathbf{y}j}}{\sum_{\mathbf{y} \in \Sigma} m_{\mathbf{y}j}} \right).$$

La Figura 3.1 ilustra la nube secuencias y la matriz combinatoria M para $S = \{aa, ab\}$ con un gap, entonces $P(\mathbf{a}, 1)$ es igual a 4/9 mientras que $\tilde{P}(\mathbf{a}, 1) = 1$, por tanto $H(1) = 0$ y $H(\mathbf{a}) = 1{,}53$.

1	**2**	**3**
a	a	
a		a
	a	a
a	b	
a		b
	a	b

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

Figura 3.1: Nube de secuencias y Matriz combinatoria.

El código que calcula los valores de entropía para los símbolos y posiciones es:

```
--Calcula el logaritmo a cada número de un vector.
vectorLog ::  [Float] ->  [Float]
vectorLog b = map log2 b

--Calcula la Frecuencia Relativa de una lista de números.
vectorProb ::  [Float] -> [Float]
vectorProb  nums = map (/b) nums
                                  where b = sum nums

--Devuelve el valor de la entropía para una lista de probabilidades.
vectorEntrop :: [Float] -> Float
vectorEntrop x = - sum( zipWith (*) (vectorProb  x)  (vectorLog (vectorProb  x)) )

--Devuelve la entropía de una fila (Entropía Símbolo).
entropFila :: Array (Int,Int) Float -> Int -> Float
entropFila matriz n = vectorEntrop [matriz ! (n,j) | j <- [1..jHi]]
 where ((iLo,jLo),(iHi,jHi)) = bounds matriz

--Devuelve la entropía de una columna (Entropía posición).
entropColumna :: Array (Int,Int) Float -> Int -> Float
entropColumna matriz n = vectorEntrop [matriz ! (i,n) | i <- [1..iHi]]
 where ((iLo,jLo),(iHi,jHi)) = bounds matriz
```

3.5. Alineamiento de una secuencia s con una secuencia patrón p

En esta parte del algoritmo se detalla el proceso de alineamiento de las secuencias patrón con las secuencias de entrada que determinan un conjunto de subsecuencias que aproximan a la solución de la subsecuencia común más larga de S. En efecto, cada secuencia s en S es alineada con cada secuencia patrón p en P, el resultado de cada alineamiento será

considerado como una aproximación al problema de la subsecuencia común más larga.

El proceso de alineamiento usa como criterio el valor de la entropía de cada símbolo en Σ, el cual definirá el orden en el que se alinean los símbolos de cada secuencia, conjuntamente con el criterio de la posición con menor entropía.

Dada una secuencia $s \in S$ y una secuencia patrón $p \in P$, se describe el proceso de alineación de s con p de forma general y a su vez se ilustra cada uno de los pasos con un ejemplo.

Paso 1

Inicialmente se construye una matriz llamada la matriz de alineamiento de s con p, cuyas dimensiones son, dos filas y l_{max} columnas. Donde la primera fila contiene los caracteres de la secuencia patrón, y la segunda fila contendrá números enteros. Sin embargo, los valores iniciales de la segunda fila son ceros. De ahora en adelante esta matriz se denota por $A = [a_{ij}]$, los valores de sus entradas se irán modificando a medida que se completa el alineamiento.

Para empezar, es necesario definir algunas funciones adicionales para el algoritmo. Primero, la función *primerBloque* toma una lista de número enteros y devuelve el primer conjunto de números consecutivos de la lista.

```
-- primerBloque [1,2,3,8,9,11] ->  [1,2,3]
primerBloque :: [Int] -> [Int]
primerBloque xs = takeWhile ( ==1 ) [xs!!(i+1) - xs!!(i) |
i <- [0..((length xs) -2) ]]
```

La función $bloque(\sigma, h)$ devuelve el primer bloque de columnas mayores o iguales a h de la matriz A, tales que contengan al símbolo σ en la primera fila y cero en la segunda. Es decir,

$$bloque(\sigma, h) = primerBloque(\{t : 1 \leq t \leq l_{max}, a_{1t} = \sigma, a_{2t} = 0, t \geq h\})$$

```
{-Devuelve las posiciones donde aparece el primer bloque de un símbolo
a partir de h -}
```

```
bloque :: Array (Int,Int) Int -> Char ->Int -> [Int]
bloque A sigma h =  if a == [] then []
                    else [a!!0 + i| i<- [0.. length (primerBloque a)]]
                where a = [j| j <- [1..jHi] , A ! (2,j)  == -1 ,
                           A ! (1,j) == (ord sigma) , j>= h]
                      ((iLo,jLo),(iHi,jHi)) = bounds A
```

Paso 2

Se ordenan los símbolos del alfabeto Σ de mayor a menor según su valor de entropía, este alfabeto ordenado se denota con Σ_0, el cual define el orden en que los caracteres de s son alineados con p.

Paso 3

Se toma el primer elemento $\sigma \in \Sigma_0$ y se selecciona las posiciones en donde aparece el símbolo σ en s, a esta lista la denotamos por $C = \{c_1, c_2, \cdots, c_k\}$ donde C esta ordenado de forma creciente. En caso de que C sea vacío, se aplica el paso 4, de otro modo se define d como $d := (d_1, d_2, \cdots, d_k)$, con valor inicial $d = (0, \ldots, 0)$.

Para cada c_i en C (recorrido en orden ascendente), se aplica alguno de los siguientes casos:

1. Si $\{t : d_t \neq 0\} = \emptyset$ y $bloque(\sigma, 1) = \emptyset$, se continúa el algoritmo sin alinear símbolo.

2. Si $\{t : d_t \neq 0\} = \emptyset$ y $bloque(\sigma, 1) \neq \emptyset$, se define j_i como el menor número $j_i \in bloque(\sigma, 1)$ tal que

   $$H(j_i) = \min_{t \in bloque(\sigma,1)} H(t)$$

   Luego, considerando $a_{2j_i} = c_i$, el conjunto $\{a_{2t} : 1 \leq t \leq l_{max}, a_{2t} \neq 0\}$ está en orden creciente, entonces se actualiza la entrada a_{2j_i} de la matriz A con c_i y $d_i = c_i$. De otro modo, $a_{2j_i} = 0$ y $d_i = 0$.

3. Si $\{t : d_t \neq 0\} \neq \emptyset$ y $bloque(\sigma, 1) = \emptyset$, se continúa el algoritmo sin alinear símbolo.

4. Si $\{t : d_t \neq 0\} \neq \emptyset$ y $bloque(\sigma, 1) \neq \emptyset$, sea $i^* := max\{t : d_t \neq 0\}$.

   Se debe analizar el conjunto $bloque(\sigma, j_{i^*} + c_i - c_{i^*})$, si éste es vacío, se continúa el algoritmo sin alinear símbolo, de lo contrario se define j_i como el menor $j_i \in bloque(\sigma, j_{i^*} + c_i - c_{i^*})$ tal que

   $$H(j_i) = \min_{t \in bloque(\sigma, j_{i^*} + c_i - c_{i^*})} H(t)$$

   Luego, si se considera $a_{2j_i} = c_i$, el conjunto $\{a_{2t} : 1 \leq t \leq l_{max}, a_{2t} \neq 0\}$ está en orden creciente, se actualiza la entrada a_{2j_i} de la matriz A con c_i y $d_i = c_i$. De otro modo, $a_{2j_i} = 0$ y $d_i = 0$.

Paso 4

Repetir el paso 3 tomando Σ_0 como $\Sigma_0 - \sigma$. Este procedimiento se realiza hasta que Σ_0 sea vacío.

Paso 5

Por último, se cambia cada número no nulo de la segunda fila de A por el caracter correspondiente a la columna, es decir, $a_{2t} = a_{1t}$, mientras que las posiciones nulas se dejan en blanco. Por lo anterior, la secuencia final resultante del alineamiento de s con p es la secuencia de caracteres que aparece en la segunda fila de A.

Ejemplo de alineación de s con p

Sea $S = \{abab, bcca, cab\}$ y $g = 2$, $l_{max} = 6$. El desarrollo del algoritmo para la alineación de $s = abab$ con una secuencia patrón $p = accaab$ es el siguiente.

- Paso 1.

  La matriz inicial A de alineamiento de s con p es:

a	**c**	**c**	**a**	**a**	**b**
0	0	0	0	0	0

- Paso 2.

  El alfabeto de S es $\Sigma = \{a, b, c\}$, teniendo en cuenta el Cuadro 3.3 que muestra la entropía de cada símbolo, se concluye que $\Sigma_0 = \{a, b, c\}$ es decir, se alinean primero todas las $a's$, luego las $b's$ y por último las $c's$.

	1	**2**	**3**	**4**	**5**	**6**	$H(\sigma)$
a	10	8	10	13	14	10	2.568
b	10	10	8	7	10	20	2.488
c	10	12	12	10	6	0	2.284
$H(j)$	1.585	1.566	1.566	1.541	1.506	0.918	

Cuadro 3.3: Matriz Combinatoria con Entropías

- Paso 3. El primer símbolo de Σ_0 es $\sigma = a$. El conjunto de posiciones en donde aparece a en s es $C = \{1, 3\}$ y $d = (0, 0)$.

  Para cada c_i en C se aplican los casos del 1 al 4 según corresponda:

  Para $c_1 = 1$, note que $\{t : d_t \neq 0\} = \emptyset$ y $bloque(a, 1) = \{1\}$. Por consiguiente, se aplica caso 2. Entonces, $j_1 \in \{1\}$ así que, $j_1 = 1$ y $a_{2_{j_1}} = a_{21}$. Suponiendo que $a_{21} = 1$ entonces, el conjunto $\{a_{2t} : 1 \leq t \leq l_{max}, a_{2t} \neq 0\}$ es creciente, así $d = (1, 0)$ y se escribe $a_{21} = 1$ en la matriz A:

a	**c**	**c**	**a**	**a**	**b**
1	0	0	0	0	0

  Para $c_2 = 3$, se ve que $\{t : d_t \neq 0\} = \{1\}$ y $bloque(a, 1) = \{4, 5\}$, por lo tanto, se aplica el caso 4.

  Como $i^* = max\{t : d_t \neq 0\} = 1$, entonces $bloque(a, j_{i^*} + c_i - c_{i^*}) = bloque(a, 3) = \{4, 5\}$, luego $j_2 = 5$ por ser la posición con menor entropía.

Se supone que $a_{25} = 3$ luego, el conjunto $\{a_{2t} : 1 \leq t \leq l_{max}, a_{2t} \neq 0\}$ está en orden creciente, así $d = (1, 3)$ y se escribe $a_{25} = 3$ en la matriz A:

a	**c**	**c**	**a**	**a**	**b**
1	0	0	0	3	0

- Paso 4. Regresa al Paso 3 con $\Sigma_0 = \{b, c\}$.
- Paso 3. Ahora el primer símbolo de Σ_0 es $\sigma = b$. El conjunto de posiciones en donde aparece b en s es $C = \{2, 4\}$, se inicia $d = (0, 0)$.

  Para cada c_i en C se aplican los casos del 1 al 4 según corresponda:

  Para $c_1 = 2$ note que $\{t : d_t \neq 0\} = \emptyset$ y $bloque(b, 1) = \{6\}$ por lo tanto, se aplica el caso 2. Entonces, $j_1 \in \{6\}$, así que $j_1 = 6$. Se supone que $a_{26} = 2$, así pues el conjunto $\{a_{2t} : 1 \leq t \leq l_{max}, a_{2t} \neq 0\} = \{1, 3, 2\}$ no está en orden creciente, en consecuencia $d = (0, 0)$ y la matriz A no se modifica:

a	**c**	**c**	**a**	**a**	**b**
1	0	0	0	3	0

  Para $c_2 = 4$, se ve que $\{t : d_t \neq 0\} = \emptyset$ y $bloque(b, 1) = \{6\}$, luego se aplica caso 2. Entonces $j_2 \in \{6\}$ así que $j_2 = 6$. Se supone que $a_{26} = 4$, por tanto, el conjunto $\{a_{2t} : 1 \leq t \leq l_{max}, a_{2t} \neq 0\} = \{1, 3, 4\}$ es creciente, en consecuencia $d = (0, 4)$ y se escribe $a_{26} = 4$ en la matriz A:

a	**c**	**c**	**a**	**a**	**b**
1	0	0	0	3	4

- Paso 4. Vuelve al Paso 3 con $\Sigma_0 = \{c\}$.
- Paso 3. El primer símbolo de Σ_0 es $\sigma = c$. El conjunto de posiciones donde aparece c en s es vacío, luego vuelve al Paso 4.
- Paso 4. Ahora $\Sigma_0 = \emptyset$, entonces vamos al Paso 5.

- Paso 5. Se cambia la segunda fila de la matriz A por sus respectivos símbolos:

a	**c**	**c**	**a**	**a**	**b**
a				a	b

La subsecuencia común obtenida del alineamiento es *aab*.

El siguiente código alinea una secuencia s con una secuencia patrón p.

```
--Devuelve la última columna donde aparece el símbolo sigma
ultimaColChar :: Array(Int, Int)Int -> Char -> Int
ultimaColChar _A sigma = if a == [] then -1
                         else a !! 0
           where a= reverse [j| j<-[1..jHi ], _A !(2,j)/= -2,
           _A!(2,j) /= -1, _A!(1,j) == ord (sigma) ]
                      ((iLo,jLo),(iHi,jHi)) = bounds _A

--Devuelve la columna donde inicia cada búsqueda para ci
colInicial ::  Array(Int, Int)Int -> Char -> Int -> Int
colInicial _A sigma ci = if anteriorCol == -1 then 1
                     else anteriorCol +(ci-_A !(2,anteriorCol))
                where anteriorCol= ultimaColChar _A sigma


{- Indice de la columna con menor entropía en el primer bloque
desocupado, donde ci es la posición del símbolo en la secuencia-}
indiceAparicion :: Array(Int,Int) Int -> Char -> [Float] ->Int -> Int
indiceAparicion _A sigma entropias ci= if ( pB ) == [] then -1
                                  else [pB !!i|i<-[0..(length vEnt - 1)],
                    (vEnt!!i)== minimum  vEnt ] !! 0
                 where vEnt =  [entropias !! (i-1) |i<- pB]
                       pB = bloque _A sigma (colInicial _A sigma ci)
```

```
-- Devuelve verdadero si la lista está ordenada en forma creciente
creciente :: (Ord a) => [a] -> Bool
creciente [] = True
creciente [x] = True
creciente (x:y:xs) = (x < y) && creciente (y:xs)


-- Alinea ci de C en A,  en la posición con menor entropía del bloque
alinearSimbolo :: Array (Int,Int) Int -> Char -> Int -> [Float] ->
Array (Int,Int) Int
alinearSimbolo _A sigma ci entropias =
   if (ind ) == -1 then _A
   else if (creciente (segundaFila m)) then m // [((2,i),-2)|
   i <- [1..ind-1], _A ! (1,i) == ord sigma , _A ! (2,i) == -1  ]
   else _A
            where m = _A // [((2,ind), ci)]
                  ind = (indiceAparicion _A sigma entropias ci)

--Define el orden de alineación de cada caracter de sr según entropía

indicesOrdenados :: [String] -> Int -> String  -> [Int]
indicesOrdenados  s g sr = [t| i <- [0.. (length c -1) ], t <- (c !! i) ]
  where c = (indicesOrd s g sr)


--Alinea sr con p
alinear :: Array (Int,Int) Int -> [String] -> Int -> String ->
[Float] -> Array (Int,Int) Int
alinear _A s g sr entropias = foldl (\mt n ->  alinearSimbolo mt
          (sr !! n ) n entropias) _A (indicesOrdenados2 s g sr)

 -- Devuelve la secuencia resultante del alineamiento de sr con p
```

```
secAlineamiento ::  Array (Int,Int) Int -> String
secAlineamiento _A = [chr (_A ! (1,j)) | j <- [1..jHi], _A ! (2,j) /= -1 ,
                _A ! (2,j) /= -2  ]
          where ((iLo,jLo),(iHi,jHi)) = bounds _A
```

3.6. Alineamiento de S con P

En la sección anterior se presentó el algoritmo para alinear una secuencia s de S con una secuencia patrón p de P. En esta sección se realiza el alineamiento de cada una de las secuencias de S con cada una de las secuencias de P. Es decir, para cada s de $S = \{s_1, \ldots, s_n\}$ y cada secuencia patrón p de $P = \{p_1, \ldots, p_w\}$, se realiza el alineamiento como en la sección 3.5. Los resultados de estos alineamientos se presentan dentro de un conjunto de matrices, de tal forma que la matriz k-ésima contiene en su i-ésima fila la secuencia final del alienamiento de s_i con p_k; se denota con U al conjunto de todas las secuencias resultantes de los alineamientos de s_i con p_k, $1 \leq i \leq n$ y $1 \leq k \leq w$.

Por ejemplo, para $S = \{abab, bcca, cab\}$ con $g = 2$, el conjunto de secuencias patrón es $P = \{accaab, cccaab, bccaab\}$. Entonces los alineamientos resultantes se muestran en las matrices del Cuadro 3.4. El conjunto de las secuencias resultantes de los alineamientos es $U = \{aab, a, ab, cca, cab, bcca, ca\}$.

a	**c**	**c**	**a**	**a**	**b**
a				a	b
a					
a					b

c	**c**	**c**	**a**	**a**	**b**
				a	b
	c	c		a	
	c			a	b

b	**c**	**c**	**a**	**a**	**b**
				a	
b	c	c		a	
	c			a	

Cuadro 3.4: Matrices de alineamiento de S con P

```
-- Calcula el conjunto U de secuencias resultantes de alinear S con P
secsAlineamientos :: [String] -> Int -> [String]
secsAlineamientos xr g =nub [solucion(alinearF patron xr g secuencia)|
                     patron <- (secuenciasPatron xr g), secuencia <- xr ]
```

3.7. Aproximación Final

Finalmente, el algoritmo escoge como mejor aproximación al problema de la Subsecuencia Común Más Larga de S al subconjunto F de secuencias de U que son subsecuencias comunes de S y tienen longitud máxima, formalmente, $F = \{\bar{s} \in U | \bar{s} \in SC(S), \forall \tilde{s} \in SC(S)(\tilde{s} \in U \rightarrow |\tilde{s}| \leq |\bar{s}|)\}$, donde $SC(S)$ representa el conjunto de todas subsecuencias comunes de S.

Para $S = \{abab, bcca, cab\}$ y $g = 2$, el conjunto resultante del alineamiento de S con P es $U = \{aab, a, ab, cca, cab, bcca, ca\}$, de las cuales tan solo la secuencia a es subsecuencia común de S, entonces el conjunto final de aproximaciones al problema de la subsecuencia común más larga de S generado por el algoritmo es $F = \{a\}$. Observe que en efecto la secuencia a es una solución al problema, sin embargo, no es la única solución, ya que b también lo es.

El siguiente código calcula la aproximación final

```
--Determina si una secuencia es subsecuencia de otra.
subsec :: Eq a => [a] -> [a] -> Bool
[]     `subsec` _      = True
(_:_ ) `subsec` []     = False
(a:as) `subsec` (b:bs) = (if a == b then as else a:as) `subsec` bs

{-Determina si una secuencia es subsecuencia común de un
conjunto de secuencias-}
subsecComun :: String -> [String] -> Bool
[]     `subsecComun` _  = True
u      `subsecComun` []     = True
u      `subsecComun` (a:as) = (u `subsec` a) && (u `subsecComun` as)

--Toma las secuencias de U con mayor longitud
mayorLongitud :: [String] -> [String]
mayorLongitud _U = filter (\x -> length x == maxU) _U
```

```
                    where maxU = maximum (map (length) _U)

--Calcula las aproximaciones finales
soluciones :: [String] -> [String] -> [String]
soluciones _U _S = mayorLongitud [u| u <- _U , subsecComun u _S  == True  ]
```

4 Análisis del algoritmo

4.1. Complejidad

Sea n el número de secuencias de S con alfabeto Σ, m la longitud de la secuencia más larga, más el número de gaps (i.e. $m = l_{max}$). Para realizar el análisis de complejidad del algoritmo, se calcula la complejidad para cada una de las secciones que lo componen mediante la notación $\mathcal{O}$ grande, la complejidad final quedará en términos de n, m y $|\Sigma|$.

Matriz Combinatoria

Observe que para construir la matriz combinatoria de una sola secuencia, es necesario realizar $|\Sigma| \cdot (m)$ combinatorias. Además, calcular cada combinatoria es de orden $\mathcal{O}(m^m)$, entonces la complejidad para construir la matriz es $\mathcal{O}(|\Sigma| \cdot m \cdot m^m) = \mathcal{O}(|\Sigma| \cdot m^{m+1})$, por ende, construir la matriz combinatoria para toda s en S es $\mathcal{O}(|\Sigma| \cdot m^{m+1} \cdot n)$. Finalmente la suma de todas las matrices combinatorias es de orden $\mathcal{O}(|\Sigma| \cdot m \cdot n)$. Por consiguiente, la complejidad de esta sección esta dada por: $\mathcal{O}(|\Sigma| \cdot m^{m+1} \cdot n + |\Sigma| \cdot m \cdot n) = \mathcal{O}(|\Sigma| \cdot m^{m+1} \cdot n)$.

Secuencias Patrón

Teniendo en cuenta que el caso donde se requiere el mayor número de operaciones ocurre cuando cada entrada de la matriz combinatoria es máximo en su columna, se tiene que la complejidad para hallar las secuencias patrón es de orden $\mathcal{O}(|\Sigma|^m)$.

Entropía por símbolos y posiciones

Calcular la entropía de los símbolos en Σ y posiciones tiene complejidad polinomial, la cual resulta imperceptible para la complejidad del algoritmo completo.

Alineamiento de s con p

Inicialmente el algoritmo ordena el alfabeto Σ de mayor a menor entropía, esto se hace en $\mathcal{O}(|\Sigma|^2)$, posteriormente alinea una secuencia $s \in S$ con una secuencia patrón $p \in P$, de tal forma que toma cada caracter de s (de acuerdo al orden) y lo alinea según las condiciones del algoritmo, para ello, debe verificar las posiciones en p donde aparece este caracter con menor entropía y revisar si la segunda fila de la matriz de alineamiento de s con p está ordenada de forma creciente, estas operaciones requieren $\mathcal{O}(m^2)$.

Alineamiento de S con P

En el alineamiento de S con P se realizan $n \cdot |\Sigma|^{m+1}$ alineamientos, como cada alineamiento de s con p toma $\mathcal{O}(m^2)$ entonces alinear S con P requiere tiempo de ejecución del orden $\mathcal{O}(m^2 \cdot n \cdot |\Sigma|^{m+1})$.

Aproximación final

Para determinar las soluciones finales, se ha supuesto el caso en el que los alineamientos de S con P son todos distintos. Decidir si una secuencia es subsecuencia de otra se requieren $2 \cdot m$ operaciones, luego es de orden $\mathcal{O}(m)$. Esta operación debe efectuarse tantas como parejas de secuencias patrón y secuencias en S hayan, por $|S|$, es decir $|\Sigma|^m \cdot n^2$, luego la complejidad de esta parte del algoritmo es $\mathcal{O}(|\Sigma|^m \cdot m \cdot n^2)$.

En conclusión, sumando los resultados de la complejidad hallados en cada una de las secciones previas se obtiene $\mathcal{O}(|\Sigma| \cdot m^{m+1} \cdot n) + \mathcal{O}(|\Sigma|^m) + \mathcal{O}(|\Sigma|^2) + \mathcal{O}(m^2 \cdot n \cdot |\Sigma|^{m+1}) + \mathcal{O}(|\Sigma|^m \cdot m \cdot n^2)$, simplificando, se tiene que la complejidad de todo el algoritmo es:

$$\mathcal{O}(|\Sigma| \cdot m^{m+1} \cdot n^2 + m^2 \cdot n \cdot |\Sigma|^{m+1})$$

.

Con el fin de mejorar el rendimiento del algoritmo se propone utilizar una tabla de combinatorias de rápido acceso que almacena todas los valores de la combinatoria $\binom{n}{k}$ hasta un n fijo, esto equivale a tomar un alfabeto Σ de tamaño fijo y restringir la longitud máxima m de las secuencias de S, de esta manera se obtiene que la complejidad del algoritmo es del orden de $\mathcal{O}(n^2 + n) = \mathcal{O}(n^2)$.

4.2. Ejemplos

A continuación se presentan algunos ejemplos donde se ilustra el procedimiento expuesto para hallar una aproximación a la subsecuencia común más larga de un conjunto S.

1. Sea $S = \{bccdaa, abcd, cbcdda, bcdd\}$ y $g = 3$.

Matriz Combinatoria

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	$H(\sigma)$
a	56	35	20	10	8	18	42	77	112	2.756
b	112	112	86	56	32	16	6	0	0	2.423
c	56	77	112	121	100	65	36	21	0	2.836
d	0	0	6	37	84	125	140	126	112	2.544
$H(j)$	1.5	1.448	1.481	1.609	1.623	1.552	1.440	1.317	1.0	

Secuencias Patrón

El conjunto de secuencias patrón que se obtiene es $P = \{bbcccddda, bbcccdddd\}$. Observe que se obtienen dos secuencias patrón ya que en la posición nueve hay un empate de frecuencias de aparición entre **a** y **d**.

Alineamientos

En primer lugar, se alinea cada una de las secuencias de $S = \{bccdaa, abcd, cbcdda, bcdd\}$ con la secuencia patrón *bbcccddda*.

b	**b**	**c**	**c**	**c**	**d**	**d**	**d**	**a**
	b	c	c				d	a
	b	c					d	
		c		c			d	a
	b	c					d	

En segundo lugar, se alinea cada una de las secuencias de $S = \{bccdaa, abcd, cbcdda, bcdd\}$ con la secuencia patrón *bbcccdddd*.

b	**b**	**c**	**c**	**c**	**d**	**d**	**d**	**d**
	b	c	c					d
	b	c						d
		c		c				d
	b	c						d

Por lo anterior, el conjunto de secuencias resultantes de los alineamientos es $U = \{bccda, bcd,\ ccda, bccd, ccd\}$, de las cuales se toma como mejores aproximaciones las secuencias de U con mayor longitud que son comunes a cada una de las secuencias de S, así que el conjunto final de aproximaciones es $F = \{bcd\}$. En efecto, la aproximación dada en F es una solución al problema.

2. Sea $S = \{babbacd, cddaab, aabcad\}$ y $g = 6$.

Matriz Combinatoria

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	$H(\sigma)$
a	792	1254	1176	918	716	686	837	1093	1324	1386	1170	660	0	3.539
b	924	462	540	798	1008	1058	932	686	420	246	252	462	792	3.571
c	792	462	252	162	168	238	342	456	560	630	624	462	0	3.434
d	0	330	540	630	616	526	397	273	204	246	462	924	1716	3.309
$H(j)$	1.581	1.784	1.799	1.812	1.804	1.832	1.872	1.823	1.67	1.631	1.795	1.937	0.9	

Secuencias Patrón

El conjunto de secuencias patrón que se obtiene es $P = \{baaabbbaaaadd\}$.

Alineamientos

Se alinea cada una de las secuencias de $S = \{babbacd, cddaab, aabcad\}$ con la secuencia patrón *baaabbbaaaadd*, como se muestra en la figura.

b	**a**	**a**	**a**	**b**	**b**	**b**	**a**	**a**	**a**	a	**d**	**d**
b	a			b	b				a			d
b												
b	a											d

De modo que el conjunto de secuencias resultantes de los alineamientos es $U = \{babbad, b, bad\}$, de las cuales la aproximación dada por el algoritmo es $F = \{b\}$. Cabe resaltar que a pesar de ser b una subsecuencia común de $\{babbacd, cddaab, aabcad\}$, no es una subsecuencia común más larga de $\{babbacd, cddaab, aabcad\}$, puesto que las soluciones son $\{cd, ab, aa\}$.

3. Sea $S = \{atatcg, ttgacc, gttcaac, gcactt\}$ y $g = 3$.

Matriz Combinatoria

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	$H(\sigma)$
a	126	56	91	141	171	174	153	112	56	0	3.072
c	0	70	70	80	110	146	177	203	224	210	3.042
g	210	84	63	67	61	41	21	21	56	126	3.004
t	126	252	238	174	120	101	111	126	126	126	3.247
$H(j)$	1.539	1.706	1.759	1.895	1.915	1.845	1.755	1.731	1.756	1.539	

Secuencias Patrón

El conjunto de secuencias patrón está dado por $P = \{gtttaacccc\}$.

Alineamiento

El alineamiento de cada una de las secuencias de $\{atatcg, ttgacc, gttcaac, gcactt\}$ con la secuencia patrón *gtttaacccc* lo presentamos en la siguiente figura.

g	**t**	**t**	**t**	**a**	**a**	**c**	**c**	**c**	**c**
	t		t						c
	t	t			a				c
g	t	t			a				c
g	t	t							

Así, el conjunto de secuencias resultantes del alineamiento es $U = \{ttc, ttac, gttac, gtt\}$, como ninguna secuencia de U es subsecuencia común de $\{atatcg, ttgacc, gttcaac, gcactt\}$, entonces la aproximación que genera el algoritmo es $F = \{\ \}$. Sin embargo, las subsecuencias comunes de longitud máxima del conjunto $\{atatcg, ttgacc, gttcaac, gcactt\}$ son *tt* y *ac*. Observe que a pesar de que el algoritmo no genera una aproximación óptima, algunas de las secuencias de U contienen secuencias cercanas a las soluciones. Por ejemplo, *ttc* y *gtt* difieren en un solo símbolo de la solución *tt*.

4.3. Resultados experimentales

En esta sección se presentan algunos resultados de una serie de pruebas efectuadas por el algoritmo planteado. Las pruebas se realizaron de la siguiente manera, primero, se ejecutó el algoritmo para treinta pares de secuencias distintas con el fin de medir el grado de efectividad del algoritmo para encontrar una solución al problema SCL. Asimismo, se realizaron pruebas para el problema SCLM con 5, 10, 15 y 20 secuencias, cada una de ellas consistiendo de treinta conjuntos distintos, con el objetivo de determinar una estimación del nivel de efectividad para aproximar la solución en el caso de múltiples secuencias.

Para cada uno de los casos mencionados se calculó el porcentaje de aciertos, porcentaje de errores, tiempo de ejecución promedio y tasa de efectividad, la cual está definida como $|SC|/|SCL|$, donde $|SC|$ denota la longitud de la subsecuencia obtenida y $|SCL|$ la longitud de la subsecuencia común más larga, finalmente se halló el promedio de ésta. La tasa de efectividad mide cuán cercana o alejada es la aproximación dada por el algoritmo de la solución óptima.

En el Cuadro 4.1 se muestran los datos obtenidos de las pruebas realizadas. Observe que, si bien la tasa más alta de efectividad y porcentaje de aciertos ocurre en el caso de dos secuencias, la tasa de efectividad disminuye significativamente para el problema SCLM de cinco o más secuencias. Sin embargo, a partir de cinco secuencias la tasa de efectividad no tiene grandes cambios.

Es preciso destacar que el tiempo de ejecución promedio es mayor para el problema SCL debido a que se presentan empates en la matriz combinatoria aumentando el número de secuencias patrón que genera el algoritmo y por tanto el tiempo total. Por el contrario, el tiempo de ejecución para hallar una aproximación al problema de SCLM de cinco y diez secuencias es menor comparado con el tiempo que emplea el algoritmo en los conjuntos de quince y veinte secuencias. No obstante, para el problema SCLM de quince y veinte secuencias el tiempo de ejecución no aumenta significativamente.

Número de secuencias	Aciertos (%)	Errores (%)	$\frac{\lvert SC\rvert}{\lvert SCL\rvert}$(%)	Tiempo (s)
2	73	27	82,9	2,85
5	60	40	54,8	1,38
10	47	53	52,7	1,48
15	57	43	52,2	3,22
20	47	56	56,1	3,91

Cuadro 4.1: Resumen de resultados

En total se realizaron 150 muestras con secuencias aleatorias, de las cuales se obtuvo 85 respuestas que son solución del problema SCLM. No se realizaron pruebas con cadenas de longitud mayor a 23, ya que el cálculo de la combinatoria para cadenas de mayor longitud desbordaría el tipo de dato usado.

5 Comentarios finales

El problema de la subsecuencia común más larga de múltiples secuencias es un problema antiguo y ampliamente estudiado, con aplicaciones en distintos campos del conocimiento. También, es conocido por ser un problema de tipo NP-Hard, es decir calcular una solución resulta ser muy costosa computacionalmente, por tanto, es necesario plantear algoritmos heurísticos que determinen aproximaciones al problema, y que requieran menor tiempo de ejecución.

El resultado de este trabajo de Maestría es un algoritmo heurístico que aproxima la solución al problema de la subsecuencia común más larga de múltiples secuencias, utilizando la entropía de Shannon como criterio de alineación. El algoritmo fue implementado en lenguaje de programación Haskell, motivado principalmente por la variedad de funciones propias del lenguaje para la manipulación de secuencias de caracteres.

Este algoritmo tiene algunas ventajas y desventajas, por ejemplo, se observa que el algoritmo se comporta bien con respecto a la cantidad de secuencias de S, es decir, al aumentar el número de secuencias de entrada, el tiempo de ejecución del algoritmo no aumentará significativamente. En este sentido, restringiendo la longitud de la secuencia más larga en el conjunto S y el cardinal del alfabeto Σ, se concluye que el algoritmo tiene una complejidad polinomial de orden cuadrático. Sin embargo, si no se consideran estas restricciones la complejidad del algoritmo es exponencial.

Por otro lado, aunque el algoritmo puede arrojar aproximaciones que no son solución del problema, éste generará subsecuencias comunes de S. Es importante señalar que el conjunto U de secuencias resultantes de los alineamientos de S con P contiene secuencias con información relevante a la solución del problema. Por tal razón, se sugiere en un trabajo

futuro estudiar la información contenida en el conjunto U para refinar las aproximaciones generadas por el algoritmo. De igual manera, se sugiere como posible trabajo futuro un algoritmo que combine medidas de la información, junto con la técnica de puntos dominantes, con el objetivo de mejorar la efectividad del algoritmo original.

Bibliografía

Alfred, V. (2014). Algorithms for finding patterns in strings. *Algorithms and Complexity*, 1:255–300.

Bereg, S. and Zhu, B. (2005). Rna multiple structural alignment with longest common subsequences. In *International Computing and Combinatorics Conference*, pages 32–41. Springer.

Blin, G., Bulteau, L., Jiang, M., Tejada, P. J., and Vialette, S. (2012). *Hardness of Longest Common Subsequence for Sequences with Bounded Run-Lengths*, pages 138–148. Springer Berlin Heidelberg, Berlin, Heidelberg.

Boes, O. (2014). *Improving the Needleman-Wunsch algorithm with the DynaMine predictor*. PhD thesis, Institute of Bioinformatics.

Bozkaya, T., Yazdani, N., and Özsoyoğlu, M. (1997). Matching and indexing sequences of different lengths. In *Proceedings of the sixth international conference on Information and knowledge management*, pages 128–135. ACM.

Carlos, A. P., Claudia, F. U., and Gonzalo, N. B. (2004). Búsqueda aproximada directamente en texto comprimido'. Technical report, Technical report, Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE), México.

Cérin, C., Dufourd, C., and Myoupo, J.-F. (1993). An efficient parallel solution for the longest increasing subsequence problem. In *Computing and Information, 1993. Proceedings ICCI'93., Fifth International Conference on*, pages 220–224. IEEE.

Cover, T. M. and Thomas, J. A. (2012). *Elements of information theory*. John Wiley & Sons.

Fakirah, M., Shehab, M. A., Jararweh, Y., and Al-Ayyoub, M. (2015). Accelerating needleman-wunsch global alignment algorithm with gpus. In *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*, pages 1–5.

Feng, B. and Gao, J. (2015). Distributed parallel needleman-wunsch algorithm on heterogeneous cluster system. In *2015 International Conference on Network and Information Systems for Computers*, pages 358–361.

Flouri, T., Kobert, K., Rognes, T., and Stamatakis, A. (2015). Are all global alignment algorithms and implementations correct? *bioRxiv*, page 031500.

Forero, W. E. S. (2010). Estudio comparativo de algoritmos para el problema de la subsecuencia común más larga restringida / comparative study of algorithms for the constrained longest common subsequence problem. Magíster en ciencias de la Ingeniería de sistemas y computación.

Fraser, C. B. and Irving, R. W. (1995). Approximation algorithms for the shortest common supersequence. *Nordic Journal of Computing*, 2(3):303–325.

Gorbenko, A. and Popov, V. (2012). On the longest common subsequence problem. *Applied Mathematical Sciences*, 6(116):5781–5787.

Hakata, K. and Imai, H. (1992). The longest common subsequence problem for small alphabet size between many strings. *Algorithms and Computation*, pages 469–478.

Hakata, K. and Imai, H. (1998). Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima. *Optimization Methods and Software*, 10(2):233–260.

Hirschberg, D. S. (1977). Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675.

Hsu, W. and Du, M. (1984). Computing a longest common subsequence for a set of strings. *BIT Numerical Mathematics*, 24(1):45–59.

Huang, K., Yang, C.-B., Tseng, K.-T., et al. (2004). Fast algorithms for finding the common subsequence of multiple sequences. In *Proceedings of the International Computer Symposium*, pages 1006–1011.

Iliopoulos, C. S. and Rahman, M. S. (2008). Algorithms for computing variants of the longest common subsequence problem. *Theoretical Computer Science*, 395(2-3):255–267.

Irving, R. W. and Fraser, C. B. (1993). On the worst-case behaviour of some approximation algorithms for the shortest common supersequence of k strings. In *Annual Symposium on Combinatorial Pattern Matching*, pages 63–73. Springer.

Jones, N. C. and Pevzner, P. (2004). *An introduction to bioinformatics algorithms*. MIT press.

Korkin, D. (2001). A new dominant point-based parallel algorithm for multiple longest common subsequence problem. *Technical Report TR01-148, Univ. of New Brunswick, Tech. Rep.*

Li, Y., Li, H., Duan, T., Wang, S., Wang, Z., and Cheng, Y. (2016). A real linear and parallel multiple longest common subsequences (mlcs) algorithm. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1725–1734. ACM.

Löytynoja, A. and Goldman, N. (2005). An algorithm for progressive multiple alignment of sequences with insertions. *Proceedings of the National academy of sciences of the United States of America*, 102(30):10557–10562.

Löytynoja, A. and Milinkovitch, M. C. (2003). A hidden markov model for progressive multiple alignment. *Bioinformatics*, 19(12):1505–1513.

Maier, D. (1978). The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, 25(2):322–336.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453.

Ning, K., Ng, H. K., and Leong, H. W. (2006). Finding patterns in biological sequences by longest common subsequencesand shortest common supersequences. In *Sixth IEEE Symposium on BioInformatics and BioEngineering (BIBE'06)*, pages 53–60.

Orlova, T. A. (2010). *Mathematical Models, Algorithms, and Statistics of Sequence Alignment*. PhD thesis, University of South Carolina.

Soto, W. and Pinzón, Y. (2010). Sobre el longest common subsequence: Extensiones y algoritmos. *Revista Colombiana de Computación-RCC*, 8(2).

Turner, J. S. (1989). Approximation algorithms for the shortest common superstring problem. *Information and computation*, 83(1):1–20.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173.

Wallace, I. M., Higgins, D. G., et al. (2005). Evaluation of iterative alignment algorithms for multiple alignment. *Bioinformatics*, 21(8):1408–1414.

Wang, Q., Korkin, D., and Shang, Y. (2009). Efficient dominant point algorithms for the multiple longest common subsequence (mlcs) problem. In *IJCAI*, pages 1494–1500. Citeseer.

Yang, I.-H., Huang, C.-P., and Chao, K.-M. (2005). A fast algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 93(5):249–253.

Yeung, R. W. (2012). *A first course in information theory*. Springer Science & Business Media.

Anexo código

```haskell
import Data.List
import Data.Array
import Data.Ratio
import Data.Char

import System.IO
import Control.Monad
import Text.Tabular
import Text.Html (renderHtml, stringToHtml, (+++))
import qualified Text.Tabular.Html as H
import System.Directory
import qualified Text.Tabular.Latex as L
import Data.List.Split
import qualified Text.Tabular.SimpleText as S

--Suma de números de un vector.

sumFila :: (Num a ) => [a] -> a
sumFila [] = 0
sumFila [x] = x
sumFila (x:xs) = x + sumTail
  where sumTail = sumFila xs
```

```
binom n k = product [max (k+1) (n-k+1) .. n] `div` product [1 .. min k (n-k)]

{-Devuelve la combinatoria de un caracter en la posición j-ésima
dado un número de gaps-}.

fcombinatoria :: Int -> Int -> Int -> Int -> Int
fcombinatoria g l j i = if (i < j) then 0
                        else if a< 0 then 0
                        else (binom i (i-j))*(binom b a )
        where (b,a) = ( g+l-i-1 , g+j-i)

fcomb  :: Int -> Int -> Int -> Int -> Int
fcomb  l g j i = if(i>j && g+j >= i ) then (binom (i-1) (j-1))*(binom a b )
                            else 0
                    where (a, b) =  (l+ g -i, l-j)

--suma la combinatoria de un símbolo en las j-ésimas posiciones.
combinatoriaAcumulada :: Int -> Int -> [Int] -> Int -> Int
combinatoriaAcumulada g l vec i =sumFila [fcombinatoria g l j i |  j <- vec ]

{-Dada una cadena de caracteres, devuelve la cadena en orden alfabético
sin repeticiones-}
conjuntoletras :: String -> String
conjuntoletras xs = [ x | x <- nub ( sort  xs) ]

{- Devuelve uno si un símbolo se encuentra en una posición
determinada de una secuencia-}
condicion:: String-> Int -> Int -> Int
condicion xs i j = if conjuntoletras xs !! i == xs !! j  then 1 else 0

--Número de símbolos distintos de la secuencia
longitud :: String -> Int
```

```
longitud xs  = length [x | x <- nub xs]

--posiciones donde se encuentra un símbolo x
posiciones :: Eq a => a -> [a] -> [Int]
posiciones x xs = [i | (x',i) <- zip xs [0..n], x == x']
           where n= length xs - 1


--ordena la lista que genera la matriz,  por renglones
printArray arr a g =  unlines [unwords [show (arr ! (x, y))|
   y <- [1..((length a)+g)]] |x <- [1..(longitud a)]]


--suma las filas y columnas de la matriz y números reales
sumFMatriz :: Array (Int, Int) Int -> Int -> Int
sumFMatriz matrix i = sumFila [matrix ! ( i , j) | j <-[1..jHi] ]
 where ((iLo,jLo),(iHi,jHi)) = bounds matrix

sumFilasMatriz :: Array (Int, Int) Int -> [Int]
sumFilasMatriz matrix = [sumFila [matrix ! ( i , j) | j <-[1..jHi]] |
    i <- [1..iHi] ]
 where ((iLo,jLo),(iHi,jHi)) = bounds matrix

sumColsMatriz :: Array (Int, Int) Int -> [Int]
sumColsMatriz matrix = [sumFila [matrix ! ( i , j) | i <-[1..iHi]] |
    j <- [1..jHi] ]
 where ((iLo,jLo),(iHi,jHi)) = bounds matrix

-- convierte una matriz de valores enteros a una matriz de valores flotantes.

matrizCombFloat ::  Array (Int ,Int) Int ->  Array (Int ,Int) Float
matrizCombFloat matriz = array  (bounds matriz) [((i,j),  fromIntegral
```

```
(matriz ! (i,j))) |  i <-[1..iHi] , j <- [1..jHi]  ]
 where ((iLo,jLo),(iHi,jHi)) = bounds matriz

--Cálculo de la entropía-------------------------------------

--Calcula la función logaritmo en base dos,  previa para la entropía
log2 :: Float -> Float
log2 0 = 0
log2 x = log x/log 2

--Calcula el log a cada numero de un vector
vectorLog ::  [Float] ->  [Float]
vectorLog b = map log2 b

--Calcula la probabilidad a cada número de una lista dada la suma total
vectorProb ::  [Float] -> [Float]
vectorProb  nums = map (/b) nums
                     where b = sum nums

--Devuelve el valor de la entropía para una lista de números.
vectorEntrop :: [Float] -> Float
vectorEntrop x = - sum( zipWith (*) (vectorProb  x) (vectorLog (vectorProb  x)))

--Devuelve la entropía de una fila de una matriz.
entropFila :: Array (Int,Int) Float -> Int -> Float
entropFila matriz n = vectorEntrop [matriz ! (n,j) | j <- [1..jHi]]
 where ((iLo,jLo),(iHi,jHi)) = bounds matriz

--Devuelve la entropía de una columna de una matriz
entropColumna :: Array (Int,Int) Float -> Int -> Float
entropColumna matriz n = vectorEntrop [matriz ! (i,n) | i <- [1..iHi]]
 where ((iLo,jLo),(iHi,jHi)) = bounds matriz
```

```
--Calculo de matrices

{-Dadas dos cadenas y gaps, devuelve una lista donde aparecen
los símbolos de la cadena xs en xt-}

indicesCaracter :: String -> String -> Int ->  [Int]
indicesCaracter xs xt k =  posiciones ((nub (sort xt )) !! k )  xs

{-Concatena las cadenas de la lista de strings xr y  devuelve
un string con los símbolos no repetidos de las cadenas de entrada-}
dConcat:: [String] -> String
dConcat xr = nub ( concat xr )

--Calcula la longitud mayor de las cadenas que componen una lista.
longMayor :: [String] -> Int
longMayor xr =maximum ( map length xr)

{-Dada una cadena de caracteres, una lista de cadenas que la contenga y  un
número de gaps, devuelve la matriz con posibles combinaciones de cada símbolo en
las j-ésimas posiciones de la cadena con mayor longitud más gaps -}
matrizCombinatoria :: String -> [String] -> Int ->  Array (Int ,Int) Int
matrizCombinatoria xs xr g = array ((1,1), (length (dConcat xr),
((length xs)+gs))) [((i,j),  combinatoriaAcumulada  gs (length xs)
(indicesCaracter xs (dConcat xr) (i-1))  (j-1) ) |
   i <- [1..(length (dConcat xr))],j<-[1..((length xs)+gs)]]
                  where gs = longMayor xr + g - length xs

--Suma dos matrices
sumaMat :: Array (Int,Int) Int -> Array (Int,Int) Int ->  Array (Int,Int) Int
sumaMat mat1 mat2 = array (bounds mat1) [((i,j), mat1!(i,j) + mat2!(i,j)) |
   (i,j) <- range (bounds mat1)]
```

```
-- suma de matrices que se generan en el paso anterior
sumaMatrices :: [Array (Int,Int) Int] -> Array (Int,Int) Int
sumaMatrices [] = error "Lista vacia"
sumaMatrices [x] = x
sumaMatrices (x:xs) = sumaMat x y
             where y = sumaMatrices xs

--Devuelve la matriz final dada una lista de cadenas con número de gaps
matrizFinal :: [String] -> Int ->  Array (Int,Int) Int
matrizFinal xr g =sumaMatrices [matrizCombinatoria xs xr g | xs <- xr]

--impresión-----------------------------------------

printArray2 arr xr g =  unlines [unwords [show (arr ! (x, y)) |
  y <- [1.. (longMayor xr + g)]] |x <- [1..(length (dConcat xr))]]

tablaFinal xr g = putStrLn(printArray2 (matrizFinal xr g) xr g)

--Segunda parte del Algoritmo.

--Primero se calcula la secuencia patrón
columnaMatriz :: Array (Int,Int) Int -> Int  -> [Int]
columnaMatriz mat col  = [mat ! (i,col)| i <- [1..iHi ]]
 where ((iLo,jLo),(iHi,jHi)) = bounds mat

filaMatriz :: Array (Int,Int) Int -> Int  -> [Int]
filaMatriz mat fil  = [mat ! (fil,j)| j <- [1..jHi ]]
 where ((iLo,jLo),(iHi,jHi)) = bounds mat

-- Caracteres que aparecen en las posiciones dadas por nums
simbPosiciones :: String -> [Int] -> [Char]
```

```
simbPosiciones xs nums = [conjuntoletras(xs) !! (num -1)| num <- nums]

-- Calcula frecuencia máxima de aparición
maxColumna :: [Int] -> [Int]
maxColumna xs = [i |i <- [1..length xs] , (xs !! (i-1)) == maximum xs ]

-- Calcula la frecuencia máxima de aparición de cada columna
maxColumnas ::  Array (Int,Int) Int -> [[Int]]
maxColumnas mat = [maxColumna (columnaMatriz mat j) |  j <-[1..jHi] ]
 where ((iLo,jLo),(iHi,jHi)) = bounds mat

--Calcular  secuencias patrón
simbMaximos :: [String] -> Int -> [String]
simbMaximos xt g = [simbPosiciones (dConcat xt) t | t <-
                   maxColumnas (matrizFinal xt g)]

-- productCart "ab" ["da", "cd" ] = ["ada", "acd" , "bda" , "bcd"]

productCart :: String -> [String] -> [String]
productCart x xs = [ [a] ++ b | a<- x ,  b <- xs]

-- productsCart ["ab","xy"] = ["ax","ay","bx","by"]

productsCart :: [String] -> [String]
productsCart [] = error "Lista vacia"
productsCart [x] = [[a] | a <- x ]
productsCart (x:xs) = productCart x y
             where y =  productsCart xs

-- Calcula la colección de sequencias patrón según la frecuencia de aparición.

secuenciasPatron :: [String]-> Int -> [String]
```

```
secuenciasPatron xs g = productsCart(simbMaximos xs g)

-- Final secuencias patrón

-- Tabular datos
tabla ::  [String]-> Int -> Table String String String
tabla xr g = Table (Group SingleLine (map Header ([[x] | x <- sort r ]
`union` [ "  " ])) ) (Group SingleLine ( map Header ([show x| x <- [1..jHi]]
`union` ["Entropia"] )))([map show (filaMatriz mat i) `union` [show(entropFila
(matrizCombFloat mat) i ) ]    | i <- [1.. (length r)]  ]
`union` [ [show (entropColumna (matrizCombFloat mat) j) | j <- [1..jHi] ]])
      where mat = matrizFinal xr g
                  jHi = (longMayor xr) + g
            r = (dConcat xr)
-- Alinear las secuencias
-- Ordenar vector entropia por filas de mayor a menor con su respectivo símbolo

ordTupla:: ((Char, Float)  -> (Char,Float) -> Ordering)
ordTupla (a,n) (b,m) | n < m = LT
                     | otherwise = GT

ordTuplas:: [(Char,Float)] -> [(Char,Float)]
ordTuplas xs = reverse (sortBy ordTupla xs)

vectEntropFilas :: Array (Int,Int) Int -> [Float]
vectEntropFilas mat  = [entropFila (matrizCombFloat mat) i |i <- [1..iHi]]
               where  ((iLo,jLo),(iHi,jHi)) = bounds mat

{-Pares formados por los símbolos de entrada y la entropía por filas
de forma ordenada de mayor a menor-}
paresEntropia ::  [String]-> Int -> [(Char,Float)]
paresEntropia xr g = ordTuplas(zip  (sort (dConcat xr)) (vectEntropFilas
```

```
 (matrizFinal xr g)))

--Indices con orden de alineación (entropía) MUY IMPORTANTE
indicesOrdenados :: [String] -> Int -> String  -> [[Int]]
indicesOrdenados xr g xs = [posiciones x xs | (x,a) <- (paresEntropia xr g) ]


matPatron2 :: String -> Array (Int,Int) Int
matPatron2 xs = array ((1,1),(2, length xs)) [((i,j),
   if i == 1 then ord(xs!!(j-1)) else   -1 )| i<-[1,2], j <- [1..(length xs)] ]

{-Devuelve la posición del símbolo x  en la primera aparición
disponible de la matriz patron-}
indiceAparicion2 ::  Array (Int,Int) Int -> Char ->  Int
indiceAparicion2 mat x = if a == [] then -1
                         else a !! 0
  where a = [j| j <- [1..jHi] , mat ! (2,j)  == -1 , mat ! (1,j) == (ord x)]
        ((iLo,jLo),(iHi,jHi)) = bounds mat

-- Alinea un símbolo en la matriz patrón
alinearSimbolo2 :: Array (Int,Int) Int -> Char -> Int ->  Array (Int,Int) Int
alinearSimbolo2 mat x n = if indiceAparicion2 mat x == -1 then mat else
mat // [((2,indiceAparicion2 mat x), n)]

{- Devuelve la longitud del primer bloque de números consecutivos
[1,2,3,8,9,11] -> length [1,2,3] == 3-}
longitudBloque :: [Int] -> Int
longitudBloque xs = length(takeWhile ( ==1 ) [xs!!(i+1) - xs!!(i) |
  i <- [0..((length xs) -2) ]])
--Devuelve la última columna donde aparece el símbolo
ultimaColChar :: Array(Int, Int)Int -> Char -> Int
ultimaColChar mat x = if a == [] then -1
```

```
                    else a !! 0
 where a= reverse [j| j<-[1..jHi ], mat !(2,j)/= -2, mat!(2,j) /= -1,
                  mat!(1,j) == ord (x) ]
       ((iLo,jLo),(iHi,jHi)) = bounds mat

--Devuelve la columna donde inicia cada búsqueda
colInicial ::  Array(Int, Int)Int -> Char -> Int -> Int
colInicial mat x n = if anteriorCol == -1 then 1
                     else anteriorCol +(n-mat !(2,anteriorCol))
                where anteriorCol= ultimaColChar mat x

--Devuelve los indices donde aparece el primer bloque de un símbolo
primerBloque :: Array (Int,Int) Int -> Char ->Int -> [Int]
primerBloque mat x colInicioBusqueda =  if a == [] then []
                      else [a!!0 + i| i<- [0.. longitudBloque a]]
               where a = [j| j <- [1..jHi] , mat ! (2,j)  == -1 ,
        mat ! (1,j) == (ord x) , j>= colInicioBusqueda   ]
                     ((iLo,jLo),(iHi,jHi)) = bounds mat

-- lista de entropía por columnas
entropColumnas :: [String] -> Int -> [Float]
entropColumnas xr g = [entropColumna (matrizCombFloat mat) j | j <-[1..jHi] ]
              where mat =  matrizFinal xr g
                           ((iLo,jLo),(iHi,jHi)) = bounds mat

{- Indice de la columna con menor entropía en el primer bloque desocupado,
donde n es la posición del símbolo en la secuencia-}
indiceAparicion3 :: Array(Int,Int) Int -> Char -> [Float] ->Int -> Int
indiceAparicion3 mat x entropias n  =
 if ( pB ) == [] then -1
 else [pB !! i|i<- [0..(length vEnt - 1) ],(vEnt !! i) == minimum vEnt ] !! 0
               where vEnt =  [entropias !! (i-1) |i<- pB]
```

```
                    pB = primerBloque mat x (colInicial mat x n)

-- Devuelve verdadero si la lista esta ordenada en forma creciente
checkList :: (Ord a) => [a] -> Bool
checkList [] = True
checkList [x] = True
checkList (x:y:xs) = (x < y) && checkList (y:xs)

--Ejemplo: [[2], [0,3],[1]]  -> [2,0,3,1]
indicesOrdenados2 :: [String] -> Int -> String  -> [Int]
indicesOrdenados2  xr g xs = [t| i <- [0.. (length c -1) ], t <- (c !! i) ]
  where c = (indicesOrdenados xr g xs)

-- Alinea la cadena sin revisar orden
alinearParcial :: Array (Int,Int) Int -> [String] -> Int ->
String -> Array (Int,Int) Int
alinearParcial mat xr g xs = foldl (\mt n -> alinearSimbolo2 mt (xs !! n)n )
mat (indicesOrdenados2 xr g xs)

-- Devuelve la segunda fila de la matriz patrón
segundaFila ::  Array (Int,Int) Int -> [Int]
segundaFila mat = [ mat !( 2,j) | j <- [1..jHi], ( mat !( 2,j)) /= -1 ,
( mat !( 2,j)) /= -2   ]
        where ((iLo,jLo),(iHi,jHi)) = bounds mat


{- Alinea un símbolo en la matriz patrón con orden,
donde n es la posición del símbolo en la secuencia de entrada-}
alinearSimbolo3 :: Array (Int,Int) Int -> Char -> Int ->  Array (Int,Int) Int
alinearSimbolo3 mat x n = if (indiceAparicion2 mat x) == -1 then mat
                          else if (checkList (segundaFila m)) then m
                          else mat
```

```
                                where m = mat // [((2,indiceAparicion2 mat x), n)]

{- Alinea símbolo en la matriz patrón considerando orden,
en la posición con menor entropía del bloque,donde n es la
posición del símbolo que se va a alinear-}
alinearSimbolo4 :: Array (Int,Int) Int -> Char -> Int ->
[Float] ->  Array (Int,Int) Int
alinearSimbolo4 mat x n entropias =
        if (ind ) == -1 then mat
        else if (checkList(segundaFila m)) then m //[((2,i),-2)|i <- [1..ind-1],
    mat ! (1,i) == ord x , mat ! (2,i) == -1  ]
        else mat
   where m = mat // [((2,ind), n)]
         ind = (indiceAparicion3 mat x entropias n)

{- Alinea una cadena en la matriz patron considerando orden
y menor entropía en el bloque-}
alinear :: Array (Int,Int) Int -> [String] -> Int -> String ->
[Float] -> Array (Int,Int) Int
alinear mat xr g xs entropias = foldl (\mt n -> alinearSimbolo4
mt (xs !! n ) n entropias) mat (indicesOrdenados2 xr g xs)

{-Devuelve la matriz con los alineamientos hechos.-}
alinearF :: String -> [String] -> Int -> String -> Array (Int,Int) Int
alinearF patron xr g xs = alinear (matPatron2 patron) xr g xs (entropColumnas xr g)


--Tabla para mostrar alineamientos y datos

tablaAlineacion :: Array (Int,Int) Int  -> Table String String String
tablaAlineacion mat  = Table   (Group SingleLine [Header " "])
(Group SingleLine [Header [chr (mat ! (1,j))]| j<- [1..jHi] ])
```

```
[ [if ((mat ! (2,j) == -1) || (mat ! (2,j) == -2) ) then " " else
[chr (mat ! (1,j))] | j <- [1..jHi]] ]
 where ((iLo,jLo),(iHi,jHi)) = bounds mat

-- Devuelve la string solución
solucion ::  Array (Int,Int) Int -> String
solucion mat = [chr (mat ! (1,j)) | j <- [1..jHi], mat ! (2,j) /= -1 ,
mat ! (2,j) /= -2  ]
         where ((iLo,jLo),(iHi,jHi)) = bounds mat

-- Lista de soluciones eliminando las soluciones repetidas.
listaSoluciones :: [String] -> Int -> [String]
listaSoluciones xr g =nub [solucion(alinearF patron xr g secuencia)|
patron <- (secuenciasPatron xr g), secuencia <- xr ]

--Para Imprimir las soluciones que genera una secuencia patrón en una tabla

filaSol :: Array (Int,Int) Int -> [Int]
filaSol mat = [mat ! (2,j) | j <- [1..jHi]]
        where ((iLo,jLo),(iHi,jHi)) = bounds mat

-- Lista de la segunda fila de la matriz con alinemiento
filasSol :: String -> [String] -> Int -> [[Int]]
filasSol patron xr g =[filaSol(alinearF patron xr g secuencia)|secuencia <- xr]

--Determina si una secuencia es subsecuencia de otra.
subsec :: Eq a => [a] -> [a] -> Bool
[]     `subsec` _      = True
(_:_ ) `subsec` []     = False
(a:as) `subsec` (b:bs) = (if a == b then as else a:as) `subsec` bs

--Determina si una secuencia es subsecuencia común de un conjunto de secuencias
```

```
subsecComun :: String -> [String] -> Bool
[]     `subsecComun` _   = True
u      `subsecComun` []      = True
u      `subsecComun` (a:as) = (u `subsec` a) && (u `subsecComun` as)


--Toma las subecuencias de U de mayor longitud.
mayorLongitud :: [String] -> [String]
mayorLongitud _U = filter (\x -> length x == maxU) _U
                          where maxU = maximum (map (length) _U)

--Calcula las soluciones finales
soluciones :: [String] -> [String] -> [String]
soluciones _U _S = mayorLongitud [u| u <- _U , subsecComun u _S  == True  ]

-- Tabla de alineamiento de una patrón con las secuencias de entrada.
tablaAlineacionP ::  String -> [String] -> Int  -> Table String String String
tablaAlineacionP patron xr g  = Table   (Group SingleLine [Header " "|
 i <- filas ])  (Group SingleLine [Header [j]| j<- patron])
 [[if ( (filas !! i)!! j == -1) || ((filas !! i) !! j == -2) then " "
 else [(patron !! j)]|j <- [0..length patron -1] ] | i <- [0..length xr-1] ]
         where filas = filasSol patron xr g

main :: IO ()
main  =
 do

    putStrLn "Ingrese las secuencias separadas por un espacio"
    cadenas <- getLine
    putStrLn "Ingrese el numero de gaps"
    gaps <- getLine
    writeFile ("Datos.html") $ renderHtml $
```

```
  H.css H.defaultCss +++ H.render stringToHtml stringToHtml stringToHtml
  (tabla (splitOn " " cadenas) (read gaps :: Int))
writeFile "Datos.tex" $ L.render id id id (tabla (splitOn " " cadenas)
(read gaps :: Int))
writeFile "Datos.tab" $ S.render " " id id id (tabla (splitOn " " cadenas)
(read gaps :: Int))
let patrones = (secuenciasPatron (splitOn " " cadenas) (read gaps :: Int))
let xr = (splitOn " " cadenas)
let g = (read gaps :: Int)

zipWithM_ (writeFile) [ x ++ ".html" | x <- patrones ] [renderHtml $ H.css
H.defaultCss +++ H.render stringToHtml stringToHtml stringToHtml
(tablaAlineacionP x xr g) | x <- patrones ]
putStrLn $ "Archivos creados "
putStrLn "Secuencias patron: "
mapM_ putStrLn (patrones)
let listaSol = listaSoluciones xr g
putStr (show(listaSol))
putStrLn " "
putStrLn  "Aproximaciones "
putStr (show (soluciones listaSol xr) )
putStrLn " "
```