Estudio del costo temporal de los algoritmos aproximados de alineación de secuencias

Gabriel H. Filipowicz
Xavier Messeguer

Report LSI-01-3-T

"ESTUDIO DEL COSTO TEMPORAL DE LOS ALGORITMOS APROXIMADOS DE ALINEACIÓN DE SECUENCIAS"

AUTOR: ***GABRIEL H. FILIPOWICZ***

DIRECTOR: ***XAVIER MESSEGUER***

- Barcelona, Abril de 2001-

1. INTRODUCCIÓN

En este informe se pueden distinguir dos grandes bloques:

- Detalle de conceptos teóricos
- Tareas de investigación

Los conceptos teóricos que se explican en los primeros capítulos son la base que permiten la comprensión de las actividades de investigación. Se incorporan en el informe de beca porque el becario realizó como parte de las actividades una traducción del TFC " Algoritmos paralelos de ensamble de secuencias de DNA" de David Rivas Roig, escrito en lengua Catalana, y por lo tanto se ha incorporado este material al presente escrito. Esta traducción no es textual, sino que incorpora ciertas adaptaciones puntuales que facilitan la comprensión de los conceptos en la lengua Castellana.

Ya en el bloque de investigación se detallan las actividades de evaluación de algoritmos y los resultados y conclusiones obtenidas.

Es importante resaltar que hasta la fecha no se han reportado tareas de investigación sobre el tema particular de este trabajo, por lo que los resultados obtenidos son de una gran ayuda para la comprensión del funcionamiento de estos algoritmos.

2. COMPARACIÓN DE SECUENCIAS

2.1 INTRODUCCIÓN

La comparación de secuencias es una operación básica que se utiliza en la mayoría de las manipulaciones más complejas vinculadas con la biología molecular. Esta operación consiste en determinar qué partes de las secuencias que se comparan coinciden y cuáles no. Este concepto sencillo da lugar a una multitud de problemas diferentes que requieren de la utilización de algoritmos y estructuras de datos complejos para encontrar una solución de manera eficiente. Algunos de estos problemas son:

- Dadas dos secuencias largas de longitud similar y sabiendo que difieren nada más en algunas pocas bases aisladas, determinar cuáles son estas diferencias, y dónde se encuentran.

```
GGATTAACTCCTTGA-CGGCTTTAAGCC
GGATTA-CTCCTTGATCGGCTTTAAGCC
```

Alineación Global

- Dadas dos secuencias, una de longitud mucho más grande que la otra, encontrar la subcadena de la cadena más larga que se asemeja más a la cadena más corta.

```
GGATTAACTCCTTGATCGGCTTTAAGCC
--------TCCTTG-TCGG---------
```

Alineación local

- Determinar si existe un prefijo de una de las secuencias que coincida con un sufijo de la otra de manera que las dos secuencias se puedan enganchar.

```
GGATTAACTCCTTG-TCGGC--------
---------CCTTGATCGGCTTTAAGCC
```

Alineación semiglobal

2.2 DEFINICIONES

Establecer una correspondencia de todas las bases de una secuencia con las bases de la otra, de manera que no quede ninguna base sin aparear, requiere que la longitud de las dos secuencias sea exactamente la misma. Esta circunstancia no se suele dar en la práctica. Por lo tanto debemos modificar la longitud de una secuencia según convenga, introduciendo al alfabeto de secuencias un signo que denominaremos gap y que representaremos con un guión (-). Los gaps podrán formar parte de cualquier secuencia, ya sea como prefijo, sufijo o entre dos nucleótidos, con la siguiente condición:

No permitiremos la alineación de dos gaps en la misma posición, ya que si se da esta circunstancia podríamos eliminarlos conservando la igualdad de longitudes sin afectar a la alineación del resto de las bases.

Alineación:

La *alineación* de dos secuencias consiste en establecer una correspondencia entre los nucleótidos de las secuencias. Cuando la alineación involucra dos secuencias, se denomina alineación simple. Cuando involucra más secuencias se denomina alineación múltiple.

Formalmente decimos que, dadas dos secuencias s_1 y s_2, se define un alineamiento entre las secuencias encontrando dos secuencias s_1^* y s_2^* tales que:

1. s_i^* se obtiene de s_i, con la eventual inserción de gaps.
2. $|s_1^*| = |s_2^*|$
3. Si $s_1^*[i] = gap$, entonces $s_2^*[i] \neq gap$, y viceversa.

Similitud:

La *similitud* es la medida de la semejanza entre dos secuencias. Dada una alineación entre dos secuencias, determinamos una puntuación asignando a las coincidencias (MATCH), otra a las diferencias (MISMATCH) y otra a la alineación de una base con un gap. Por ejemplo, 1; -1; -2, respectivamente. No existe una forma determinada de calcular la similitud, sino que es necesario utilizar la más conveniente en cada caso. El método explicado en el ejemplo es que tiene el menor costo computacional.

2.3 ALINEACIÓN GLOBAL

El objetivo de este apartado es presentar métodos para encontrar la mejor alineación global entre dos secuencias. El concepto de mejor alineación dependerá del tipo de problema con el que trabajemos.

2.3.1 ALGORITMO DE PROGRAMACIÓN DINÁMICA

Es evidente la necesidad de algoritmos eficientes que puedan ser ejecutados sobre computadoras. Por eso comenzaremos presentando el algoritmo básico de alineación que utiliza la técnica de programación dinámica.

La programación dinámica es una técnica que se aplica sobre algoritmos que cumplen con una serie de requisitos:

- El problema podrá ser descompuesto en problemas más pequeños.
- La solución óptima de un subproblema es una solución óptima del problema.
- Las soluciones óptimas del subproblema se pueden combinar para obtener la solución del problema original.

En el caso de la alineación de secuencias, la aplicación de programación dinámica es fundamentada sobre la propiedad que permite calcular la alineación de dos secuencias a partir de la puntuación obtenida en comparar los prefijos de estas secuencias.

Sean s y t dos secuencias de longitud m y n respectivamente. Existen (m+1) posibles prefijos de s y (n+1) prefijos de t, incluida la cadena vacía λ. Por lo tanto, el número total de posibles apareamientos de prefijos de s y de t es (m+1)(n+1). Para cada una de estos apareamientos se encontrará la mejor alineación. El resultado de esta alineación puede ser almacenado en una tabla de dimensiones (m+1)(n+1) donde la posición (i,j) contiene la puntuación correspondiente a la alineación de los prefijos s[1..i] y t[1..j].

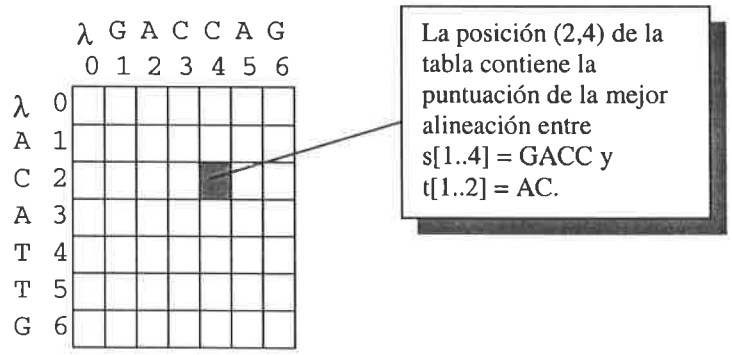


***Fig. 1**. Tabla que utilizamos para calcular la alineación entre dos secuencias*

Utilizando programación dinámica, lo que tenemos que hacer es construir la solución del problema comenzando por resolver los subproblemas más sencillos. Estos subproblemas los combinaremos sucesivamente a fin de encontrar la solución del problema completo. El primer nivel de resolución es el que corresponde a la alineación de λ con todos los prefijos. Estas son los subproblemas más sencillos y no dependen de ningún otro resultado. Al resto de los subproblemas los resolvemos combinando las soluciones asociadas con cada subproblema que se han almacenado en la tabla. La solución al problema de alineación global es la correspondiente a los prefijos s[1..m] y t[1..n], que queda almacenada en la posición (m,n) de la tabla.

Podemos calcular de manera inmediata las entradas correspondientes a las alineaciones de la cadena vacía con todos los prefijos de s y t como –2k, donde k es la longitud de la cadena no vacía y (-2) es la puntuación que se le otorga a un gap. Esto es porque sólo existe una posible alineación entre λ y una cadena cualquiera que consiste en aparear todas las bases de la secuencia no vacía con gaps.

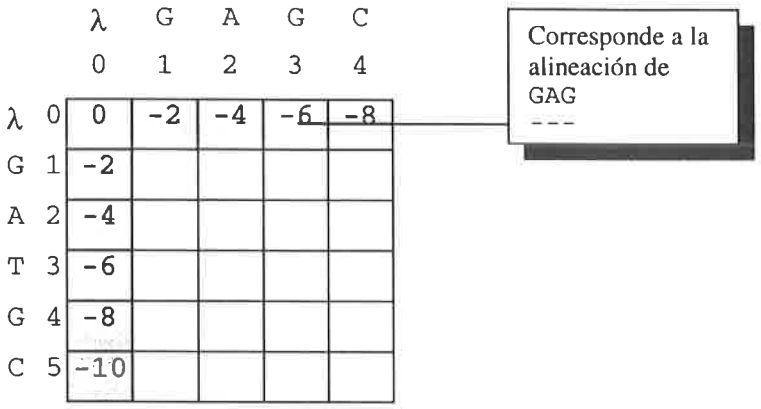


Fig. 2. *Inicialización de la tabla*

Para calcular la puntuación a partir de las puntuaciones de los prefijos previamente calculados y almacenados en la tabla, se utiliza la siguiente propiedad.

Podemos calcular la alineación entre s[1..i] y t[1..j] considerando las alineaciones entre

1. s[1..i-1] y t[1..j-1]
2. s[1..i] y t[1..j-1]
3. s[1..i-1] y t[1..j]

Esto es debido a que existen nada más tres maneras de obtener la alineación de s[1..i] y t[1..j] a partir de los prefijos:

1. Alinear s[1..i] y t[1..j-1] y aparear t[j] con un gap.
2. Alinear s[1..i-1] y t[1..j] y aparear s[i] con un gap.
3. Alinear s[1..i-1] y t[1..j-1] y aparear s[i] con t[i].

Una cuarta posibilidad sería alinear s[1..i] y t[1..j] y aparear dos gaps, pero esto no está permitido.

Si denominamos a la tabla de resultados como A, definiremos cada una de sus posiciones con la ecuación:

$$A[i,j] = \text{máx}\begin{cases} A[i,j.1]-g \\ A[i-1,j-1]+p(i,j) \\ A[i-1,j]-g \end{cases}, \text{ donde g = 2 y } p(i,j) = \begin{cases} 1 si s[i] = t[j] \\ -1 si s[i] \neq t[j] \end{cases}$$

A partir de la tabla inicializada tal como se muestra en la figura XXX, aplicamos la ecuación XXX para llenar el resto de las posiciones comenzando por la posición (1,1) que se calcula a partir de (0,0), (0,1), (1,0). Continuando hasta completar la fila 1 y sucesivamente calculando el resto de las filas hasta m.

		λ	G	A	G	C
		0	1	2	3	4
λ	0	0	-2	-4	-6	-8
G	1	-2	1	-1	-3	-5
A	2	-4	-1	2	0	-2
T	3	-6	-3	0	1	-1
G	4	-8	-5	-2	1	0
C	5	-10	-7	-4	-1	2

Puntuación de la mejor alineación global

Fig. 3. *Aspecto de la tabla una vez completada*

Una vez completa la matriz, obtenemos el resultado de la alineación de dos secuencias en la posición (m,n). En este ejemplo, la puntuación de la alineación es 2.

A continuación se visualiza en términos formales el algoritmo de programación dinámica para encontrar la mejor alineación global.

```
Algoritmo AlineacionGlobal
  Entrada: secuencias s y t
  Salida: similitud global entre s y t
  Variables: A, tabla numérica de dimensión |s|x|t|

  m = |s|
  n = |t|
  para i = 0 hasta m hacer
    A[i,0] = i * g
  para j = 0 hasta n hacer
    A[0,j] = j * g
  para i = 1 hasta m hacer
```

```
    para j = 1 hasta n hacer
      A[i,j] = max ( A[i-1, j] + g,
                     A[i-1, j-1] + p(i,j),
                     A[i, j-1] + g )
Retornar A[m,n]
```

Calcular la complejidad temporal o el tiempo de ejecución de este algoritmo es lo mismo que calcular el tiempo necesario para llenar la matriz de $(m+1)\cdot(n+1)$ posiciones. Si consideramos la operación de adición y la de encontrar el máximo de tres números dados como operaciones elementales, es decir, de costo igual a una unidad temporal, el costo de llenar la matriz es de $(m+1)\cdot(n+1)\cdot 1 = m\cdot n + n + m + 1$. Para obtener el costo en el caso más general y para cualquier longitud de la entrada, nada más nos quedaremos con los términos de mayor orden y expresaremos la complejidad del algoritmo como O(mn). Si asumimos que la longitud de las dos cadenas es la misma, la expresión del orden de magnitud se transforma en $O(n^2)$. Por esto es que decimos que es un algoritmo cuadrático.

3. ALGORITMOS APROXIMADOS

3.1 INTRODUCCIÓN

A pesar de estas limitaciones, los algoritmos heurísticos son útiles porque reducen sustancialmente el tiempo de ejecución respecto del algoritmo correcto. Los algoritmos aproximados son aquellos que no garantizan que se encontrará la solución del problema a resolver. Es posible que un algoritmo heurístico devuelva la solución óptima, pero en muchas ocasiones devolverá una solución no óptima o será incapaz de retornar alguna solución. Se presentan dos algoritmos heurísticos básicos para calcular la similitud de dos secuencias. Ambos reducen el costo temporal del algoritmo de programación dinámica, llegando a ser prácticamente lineal en algunos casos. De esta manera, el tiempo de ejecución para entradas de longitud considerable se ve notablemente reducido. Estos algoritmos sirven sólo para calcular la similitud entre las dos secuencias y no para encontrar la alineación óptima. Así, resultan de gran utilidad para resolver problemas de acoplamiento de secuencias, ya que en este tipo de problemas es necesario comprobar todos los posibles acoplamientos entre los fragmentos que se den como entrada del problema. Al fijar en número de fragmentos como n, da lugar a

$$\frac{n \cdot (n-1)}{2}$$

posibles acoplamientos entre ellos, que deben ser verificados. Un problema que recibe como entrada 10.000 fragmentos de ADN, requiere $10.000 \cdot 9.999 / 2 = 49.995.000$ ejecuciones del algoritmo de comparación. Si suponemos un caso ideal en el que todos estos fragmentos se acoplen entre ellos para formar una única cadena larga de ADN, observemos que sólo 9.999 acoplamientos son válidos, y es necesario que sean reconstruidos. El resto de las comparaciones darán un resultado negativo. Es por esto que bastará con calcular la similitud entre todos los pares de secuencias con el algoritmo heurístico y buscar las secuencias alineadas con el algoritmo de costo cuadrático sólo en los casos en que el algoritmo aproximado de costo lineal haya dado una similitud elevada.

3.2. ALGORITMO FASTA

Para comenzar, definiremos algunos conceptos necesarios para comprender el algoritmo FASTA. Posteriormente es descripto el funcionamiento del algoritmo.

3.2.1. DEFINICIONES

En este apartado se introducen las definiciones y conceptos básicos necesarios para seguir el capítulo y que no se incluyeron en capítulos preliminares.

k-tuplo:

Dada una secuencia de ADN de longitud más grande o igual a k, definiremos como k-tuplo (k-vocablo) a toda subcadena de s de longitud exactamente igual a k. Así, la secuencia AGTAGAC consta de 5 3-mots, que son AGT, GTA, TAG, GAC, AGA.

Segmento:

Dadas dos secuencias s y t, definimos un segmento de s y t como la alineación global de una subcadena de s con una subcadena de t. Por lo tanto, dadas s = GATCT y t = TTCTAG, la alineación siguiente

ATCT

-TCT

formada por ATCT y TCT, subcadenas de s y t respectivamente, es un segmento de s y t.

k-segmento:

Cuando un segmento está formado por un k-tuplo de s y un k-tuplo de t y, además, no contiene un gap, diremos que se trata de un k-segmento. La alineación mostrada en la definición anterior no sería un k-segmento porque la longitud de los dos tuplos que lo componen es diferente. En cambio, la alineación

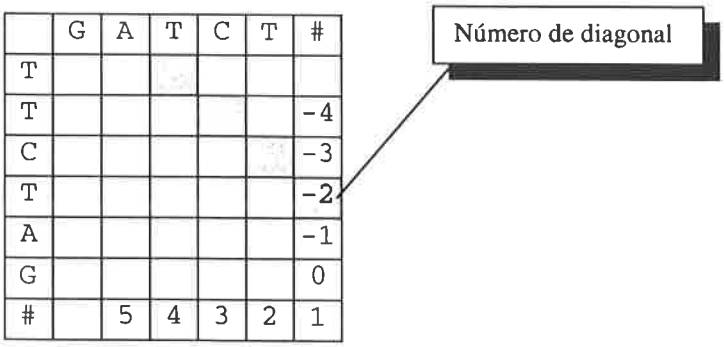
ATCT

TTCT

si es un 4-segmento de s y t.

Segmento sobre la diagonal:

Diremos que un segmento se encuentra sobre la diagonal d cuando el k-tuplo que pertenece a la secuencia s y comienza a partir de una posición i y el k-tuplo que pertenece a t a partir de la posición j son tales que i – j = d. En el ejemplo, ATCT comienza a partir de s[2] y TTCT a partir de t[1]. Por lo tanto, el 4-segmento se encuentra sobre la diagonal 2 – 1 = 1. Así, lo veremos gráficamente si consideramos la tabla que utilizaremos en el algoritmo de programación dinámica, y allí señalaremos las coincidencias entre los nucleótidos que conformen el 4-sermento.



	G	A	T	C	T	#
T						
T						-4
C						-3
T						-2
A						-1
G						0
#		5	4	3	2	1

***Fig. 4**. Representación gráfica de un 4-segmento sobre una diagonal*

k-segmento coincidente:

Definiremos un k-segmento como coincidente si y sólo si dos tuplos que lo componen son iguales en todos sus nucleótidos. Un ejemplo de k-segmentos coincidentes par las secuencias s y t serían:

TCT

TCT

que se encuentran en la diagonal (-1) y es tal que todos los nucleótidos coinciden.

3.2.2. DESCRIPCIÓN DEL ALGORITMO

Este primer algoritmo está basado en la familia de los algoritmos FAST, que se utilizan para la búsqueda de secuencias en grandes bases de datos. Se basa en que las coincidencias entre dos secuencias que son similares se encuentren en forma de pares de segmentos, es decir, subsecuencias de nucleótidos de los cuales coincidan exactamente, y no como nucleótidos aislados coincidentes. Entonces, esta heurística no busca coincidencias nucleótido a nucleótido, sino que agrupa los residuos y hace una búsqueda de grano grueso. Estos grupos de residuos reciben el nombre de w-meros, w-vocablos o w-mots, donde w es el número de residuos que lo conforman, llamado k-tup.

La heurística se divide en dos etapas que describiremos a continuación:

Etapa 1:

En primer lugar construiremos una tabla de consulta para la secuencia de entrada s. Pondremos como medida de las palabras k-tup = 2. Esta medida da lugar a 4^2=16 posibles 2-tuplos diferentes. La tabla de consulta T tendrá una entrada para cada una de estas palabras. Llenaremos las posiciones recorriendo la secuencia s y para cada palabra p de medida 2 en s insertaremos en la posición de T correspondiente a la palabra la posición dentro de s en la que está p.

```
Algoritmo ConstrucciónLookup
  Entrada: secuencia de ADN s, k-tup w
  Salida: tabla de hash C
  Variables: i

  para i desde 1 hasta |s|-w hacer
    añadir_valor(C[s[i..i+w]],i)

  retornar C
```

Como ejemplo, construiremos la tabla de consulta de la secuencia X = AATACGTAGCCTA que comparamos con Y = ATACCTACACCCTACCT. La primera 2-upla es AA en la posición 1 de X, entonces insertamos en la posición de la tabla correspondiente a AA el valor 1. Repetimos la operación para la siguiente 2-tuplo y seguimos hasta el final de X.

2-tupla	AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
Pos	1	4	8	2		10	5	11	9	6	3, 7, 12					

Fig. 5. *Tabla de consulta para una secuencia de ADN*

Etapa 2:

Compararemos la secuencia X con la secuencia Y. Para cada 2-upla de la secuencia Y buscaremos en la tabla de consulta que hemos construido en la etapa anterior si X contiene una 2-upla coincidente. Si es el caso, hay una coincidencia entre X e Y de dos residuos consecutivos. En el ejemplo, la primera palabra de Y a tratar es AT. La tabla de consulta muestra que en X, AT aparece en la posición 2. Por lo tanto, coinciden las posiciones X[2] = A con Y[1] = A y X[2] = T con Y[2] = T. En la siguiente figura, se muestra la tabla completa de coincidencias que se obtienen cuando se concluye esta segunda etapa.

			Y															
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
			A	T	A	C	C	T	A	C	A	C	C	C	T	A	C	C	T
X	1	A																	
	2	A	1																
	3	T		1				1							1				
	4	A			1				1		1					1			
	5	C																	
	6	G																	
	7	T		1				1							1				
	8	A																	
	9	G																	
	10	C				1						1	1				1		
	11	C					1							1				1	
	12	T		1				1							1				
	13	A																	

Fig. 6. *Tabla completa de coincidencias de nucleótidos entre dos secuencias X; Y*

Ahora, sumando para cada diagonal las coincidencias que hemos localizado, obtenemos el resultado final que se puede ver en la figura XXX, donde se muestran las puntuaciones obtenidas para aquellas diagonales en las cuales se encuentra alguna coincidencia. Identificamos las diagonales con la resta de los índices de las posiciones que la forman, así la diagonal d está formada por las posiciones (i,j) tales que i - j = d. Por ejemplo, la diagonal formada por (1,8), (2,9), (3,10) y todas las posiciones (i,j) de la tabla tales que i – j = -7 se identifican como la diagonal –7.

Diagonal	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10
Puntuación	2	0	0	0	4	2	0	0	0	4	2	4	0	3	0	2	2	0	0	0	3

Fig. 7. *Puntuación obtenida por las diagonales*

3.2.3. COMPLEJIDAD TEMPORAL

La primera etapa, la construcción de la tabla de consulta, se calcula en un tiempo de O((n-w)·k), donde w es el k-tup, (n-w) es el número de palabras de longitud w y k es el costo de una operación de inserción en la tabla. Aplicando la definición de notación O, tenemos

$$T : O((n-w)\cdot k) = O(n)$$

La segunda etapa, en la que buscamos los segmentos coincidentes de longitud w, tenemos un costo proporcional al número de segmentos de longitud w coincidentes que encontremos entre las dos secuencias. Si consideramos que las secuencias son equiprobables, es decir, que los símbolos A, C, G y T aparecen en la misma proporción, el número de coincidencias en términos medios está dado por la siguiente expresión

$$\frac{(m-n)\cdot(n-w)}{4^w}$$

que tiene $T: O(n^2/4^w)$. Este costo se puede reducir aumentando el valor del denominador, es decir, incrementando el valor de w o, lo que es lo mismo, aumentando el valor del k-tup con el que se trabaje. El costo global de la heurística es el mayor de las dos etapas y es

$$T: O(\text{máx}(n, n^2/4^w))$$

Si operamos con esta expresión, encontramos que si

$$n = 4^w \Leftrightarrow w = \log_4 n$$

obtenemos

$$T \approx O(n)$$

De esta manera podemos esperar una complejidad de este algoritmo de una forma aproximadamente lineal, adoptando un valor apropiado de w.

3.3 ALGORITMO BESTD

El algoritmo Best Diagonal o BestD es similar al propuesto en el Proyecto de Final de Carrera, *"Superposición de secuencias de ADN"*, realizado por Miquel Angel Martínez bajo la dirección de Xavier Messeguer en el Dto. Lenguajes y Sistemas Informáticos de la UPC [Mar00].

Este algoritmo aproximado se basa en la observación de que en el algoritmo de programación dinámica las alineaciones óptimas, cuando son relevantes, aparecen más o menos ajustadas a alguna de las diagonales de la tabla. Las desviaciones respecto a esta diagonal son mínimas cuando la alineación es buena, ya que mientras los nucleótidos de dos secuencias coincidan, el camino óptimo sigue estrictamente una diagonal. El hecho de que una alineación se desvíe de una diagonal implica la presencia de un gran número de gaps y, en consecuencia, baja similitud entre las dos secuencias.

3.3.1. PRESENTACIÓN DEL ALGORITMO:

El algoritmo procede de la misma manera que el algoritmo cuadrático de programación dinámica que ya hemos descripto, pero limita la parte de la tabla que explora a la zona en donde se concentra la mayor parte de las coincidencias entre las dos secuencias. De esta manera orienta la búsqueda a la región de la tabla donde es posible encontrar la alineación óptima con una probabilidad más alta, debido a que habitualmente la puntuación que se asigna a las coincidencias

es más alta que la que se asigna a las diferencias. Obviamente, la complejidad del algoritmo se reduce con respecto al algoritmo de programación dinámica en la misma proporción que la diferencia en la medida de la fracción de la tabla que se explore.

La medida de las diagonales de la tabla es, como máximo, la de la más larga de las dos secuencias. En notación O, podemos afirmar que es O(mín(m,n)), donde m y n son las longitudes de las secuencias, o bien O(n) si m y n son del mismo orden de magnitud. La medida de la banda a explorar consiste en un cierto número k de estas diagonales y, por lo tanto, la medida será $k \cdot O(n) = O(k \cdot n)$, siendo k el número de diagonales o amplitud de la banda. Pero en notación O se desprecian las constantes que multiplican, concluyendo que la medida, y en consecuencia el costo de exploración de esta banda, es de O(n).

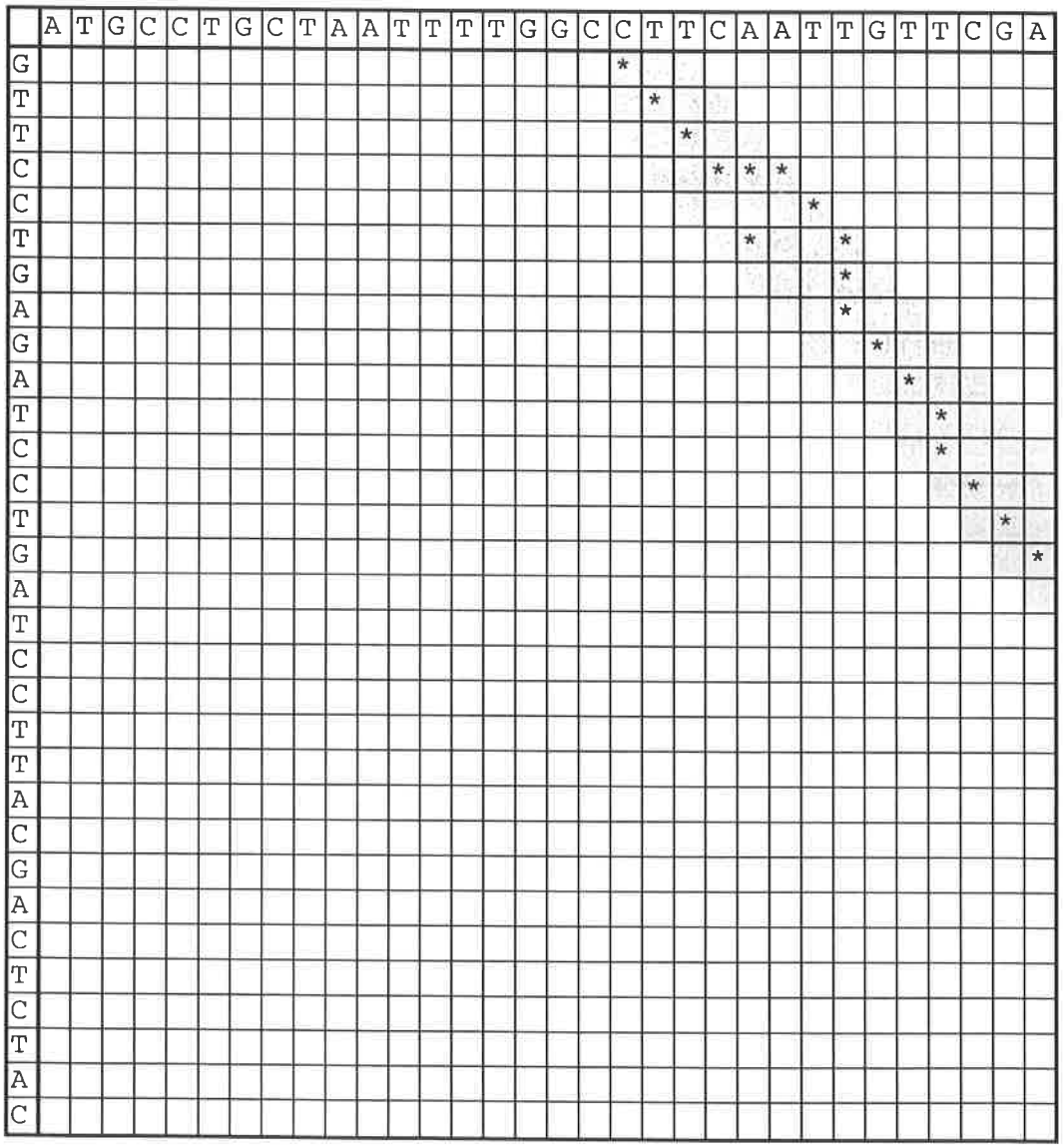


Fig. 8. *Ejemplo de alineación semiglobal donde se ve la zona de la tabla donde se localizan las coincidencias entre las secuencias. (el contenido de la tabla es sólo ilustrativo)*

El procedimiento descripto anteriormente por la heurística nos conduce al planteo de un nuevo problema: determinar cuál es la banda o grupo de diagonales que conviene explorar. Pero este problema se resuelve en tiempo lineal en el algoritmo FASTA que hemos descripto anteriormente. Utilizaremos este algoritmo aproximado para obtener una cuenta de la cantidad de coincidencias de nucleótidos por diagonales.

3.4. ALGORITMO BESTC

La heurística de este algoritmo está basada en la misma idea que hemos descripto en el algoritmo anterior, al menos en su primera etapa. El propósito del algoritmo es calcular la similitud de la alineación semiglobal de dos secuencias de entrada s y t. El nombre BestC proviene de Best Connect, dado que el algoritmo calcula la alineación entre dos secuencias a partir de la conexión de las mejores alineaciones locales.

3.4.1. PRESENTACIÓN DEL ALGORITMO:

La idea principal del algoritmo es conectar los diferentes segmentos coincidentes (alineaciones locales) entre las dos secuencias que se comparen para encontrar la alineación global óptima. El algoritmo FastA encuentra los segmentos coincidentes entre las dos secuencias, siempre que tengan una longitud más grande que el valor del parámetro k-tup que podemos fijar. El algoritmo BestD utiliza esta información para determinar una zona de la tabla en la cual buscar la mejor alineación con una probabilidad más alta de encontrar la mejor posible. En cambio BestC intenta conectar los segmentos coincidentes encontrados previamente para construir segmentos más largos.

Es necesario observar que las mejores alineaciones semiglobales son aquellas que más se ajusten a alguna de las diagonales de la tabla T. A diferencia de la alineación global, en la que se busca una alineación ajustada a la diagonal principal de T, en la alineación semiglobal son útiles cualesquiera de las diagonales de T. A mayor longitud de la diagonal a la cual se ajusta la alineación, más sólida o robusta será esta alineación, es decir, es más grande la longitud de los segmentos prefijo - sufijo coincidentes entre s y t.

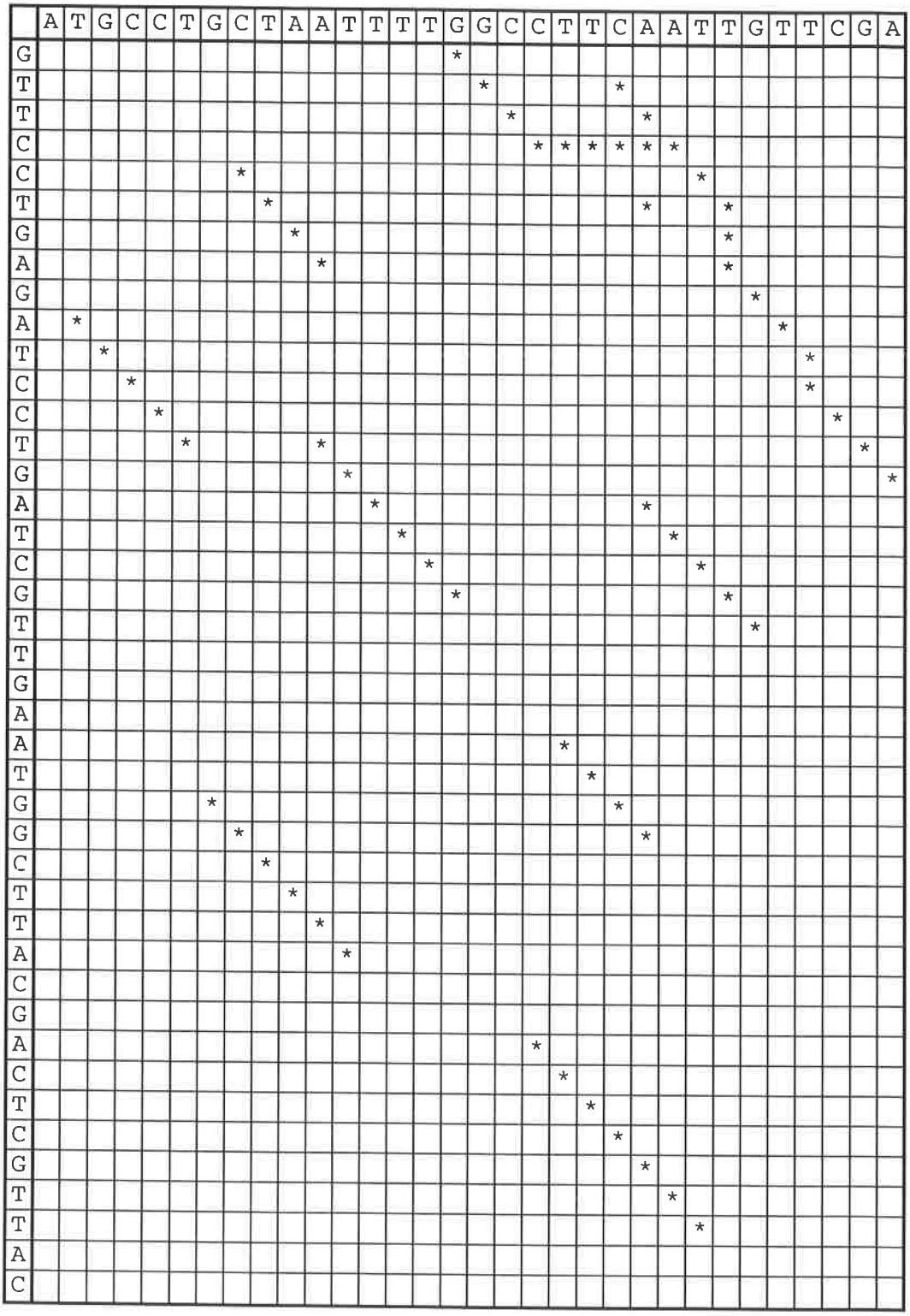


Fig. 8. *Conexión de diferentes segmentos para encontrar una alineación. (el contenido de la tabla es sólo ilustrativo)*

4. EL TRABAJO DE INVESTIGACIÓN

4.1. INTRODUCCIÓN

En los apartados anteriores hemos descrito de forma general los algoritmos de programación dinámica y los aproximados. Del análisis de los algoritmos aproximados se puede observar que existe una etapa que es común a todos, la etapa de construcción de la tabla T. La construcción de esta tabla es la base de los algoritmos aproximados, ya que posteriormente cada uno de ellos aprovecha esta información de una forma diferente, pero todos comienzan con su confección.

Por tal motivo, es importante analizar el comportamiento del algoritmo de construcción de esta tabla T, y la influencia de algunos parámetros sobre la información que quedará disponible para el procesamiento particular de cada algoritmo.

Así, decidimos evaluar la influencia del parámetro k-tup en las características de la tabla T, y en consecuencia de la información disponible para que los algoritmos aproximados puedan construir las alineaciones.

El objetivo principal de esta investigación es encontrar una ley que describa la influencia del parámetro k-tup sobre el costo computacional del algoritmo. Es razonable pensar que si se aumenta el valor del k-tup, será menor el tiempo que se necesita para la confección de la tabla T porque se requiere de menor cantidad de información. Por otra parte, a medida que se aumente el valor del k-tup también se perderá especificidad en cada comparación, pudiendo empeorar la calidad de la solución obtenida. Por lo tanto, es importante lograr un equilibrio entre la calidad de la solución y el tiempo requerido para obtenerla. En este trabajo se le propone al becario encontrar una expresión que describa la forma de la influencia del k-tup sobre el costo temporal, para cuantificar lo que hemos descripto como razonable o esperado.

La idea básica del trabajo consiste en generar una secuencias aleatoria de bases de una longitud determinada, duplicarla, y a esa copia producirle algunas mutaciones puntuales con una cierta probabilidad de mutación. De esta forma, se realizarán diferentes pruebas para comprobar la respuesta de los algoritmos ante situaciones con distintas tazas de mutación.

4.2. DESARROLLO DE HERRAMIENTAS

Si bien en el entorno ALGGEN, localizado en la página web *"www.lsi.upc.es/~alggen.html"*, cuenta con varios programas desarrollados para el área de bioinformática, no existe un programa que sirva como herramienta para realizar esta investigación. Por tal motivo, se han desarrollado herramientas específicas para el análisis de este tipo de situaciones.

Las herramientas desarrolladas están orientadas al análisis de dos tipo de experiencias:

1. Influencia del k-tup sobre la morfología de la matriz T
2. Influencia del k-tup sobre el costo temporal del algoritmo

Estos programas fueron desarrollados en MatLab 5, y pueden enumerarse las siguientes funciones:

- `BaseAlAzar()`
- `SecuenciaAlAzar(n)`
- `Mutar(s, im)`
- `MatrizT(s1, s1, k-tup)`

`BaseAlAzar()`

Descripción: Genera una base al azar, con distribución de probabilidad uniforme.

Entrada: ninguna

Salida: `'A' o 'G' o 'T' o 'C'`

`SecuenciaAlAzar(n)`

Descripción: Genera una secuencia de longitud **`n`** de bases al azar, con distribución de probabilidad uniforme.

Entrada: **`n`**

Salida: secuencia al azar

`Mutar(s, im)`

Descripción: Genera una secuencia que difiere de **`s`** sólo en las mutaciones, las cuales se producen con una probabilidad dada por el índice de mutación **`im`**

Entrada: **`s, im`**

Salida: secuencia mutada

MatrizT(s1, s2, k-tup)

Descripción: Genera la matriz T comparando **s1** y **s2**, tomando el **k-tup** indicado. La matriz que se devuelve tiene '**x**' indicando coincidencias y espacio donde no las hay.

Entrada: **s1, s2, k-tup**

Salida: matriz T

Estas herramientas son suficientes para el análisis de la experiencia 1, pero se necesitan más herramientas para la situación 2. Se han desarrollado entonces:

MatrizT1(s1, s2, k-tup)

Descripción: Genera la matriz T comparando **s1** y **s2**, tomando el **k-tup** indicado. La matriz que se devuelve tiene '**1**' indicando coincidencias y '**0**' donde no las hay.

Entrada: **s1, s2, k-tup**

Salida: matriz T

4.3. La experimentación

4.3.1. Experiencia 1

Para determinar la morfología de la matriz T se utilizó el siguiente procedimiento:

- Generar dos secuencias que sólo difieran entre sí en unas pocas mutaciones.
- Aplicarles a estas secuencias la función MatrizT y observar cómo varía la forma de la matriz al variar el k-tup.
- Repetir el experimento para diferentes índices de mutación.

4.3.1.1. Resultados

Por razones de simplicidad en la explicación e interpretación de los resultados, se ilustra un ejemplo de dos secuencias de 30 bases de longitud, son un índice de mutación del 10% aprox. y para algunos valores del parámetro k-tup.

Las secuencias son:

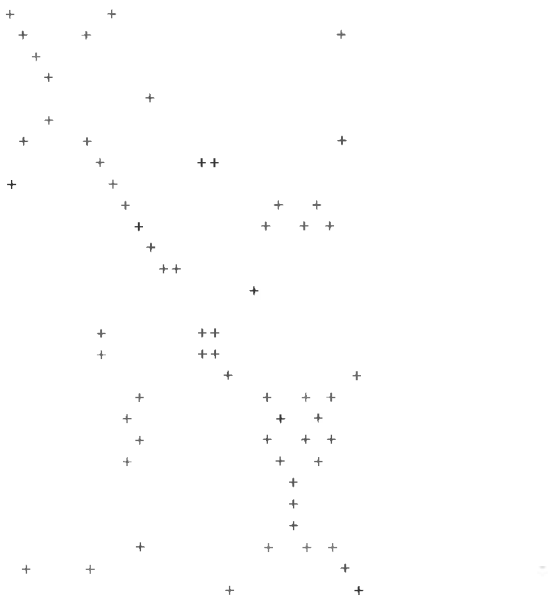
```
s1 = ACATCTCAACGCTTGAAAGCGCGGGGCAGA
          x        x    x    x
s2 = ACATCCCAACGCTTTAAAGTGCGGCGCAGA
```

Los resultados obtenidos para diferentes valores de k-tup son:

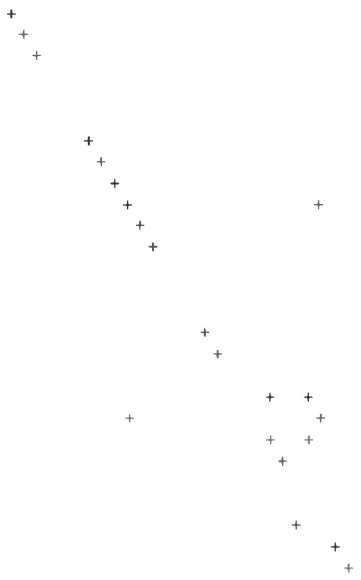
k-tup = 1

	A	C	A	T	C	C	C	A	A	C	G	C	T	T	T	A	A	A	G	T	G	C	G	G	C	G	C	A	G	A
A	+		+					+	+							+	+	+										+		+
C		+			+	+	+			+		+										+			+		+			
A	+		+					+	+							+	+	+										+		+
T				+									+	+	+					+										
C		+			+	+	+			+		+										+			+		+			
T				+									+	+	+					+										
C		+			+	+	+			+		+										+			+		+			
A	+		+					+	+							+	+	+										+		+
A	+		+					+	+							+	+	+										+		+
C		+			+	+	+			+		+										+			+		+			
G											+								+		+		+	+		+			+	
C		+			+	+	+			+		+										+			+		+			
T				+									+	+	+					+										
T				+									+	+	+					+										
G											+								+		+		+	+		+			+	
A	+		+					+	+							+	+	+										+		+
A	+		+					+	+							+	+	+										+		+
A	+		+					+	+							+	+	+										+		+
G											+								+		+		+	+		+			+	
C		+			+	+	+			+		+										+			+		+			
G											+								+		+		+	+		+			+	
C		+			+	+	+			+		+										+			+		+			
G											+								+		+		+	+		+			+	
G											+								+		+		+	+		+			+	
G											+								+		+		+	+		+			+	
G											+								+		+		+	+		+			+	
C		+			+	+	+			+		+										+			+		+			
A	+		+					+	+							+	+	+										+		+
G											+								+		+		+	+		+			+	
A	+		+					+	+							+	+	+										+		+

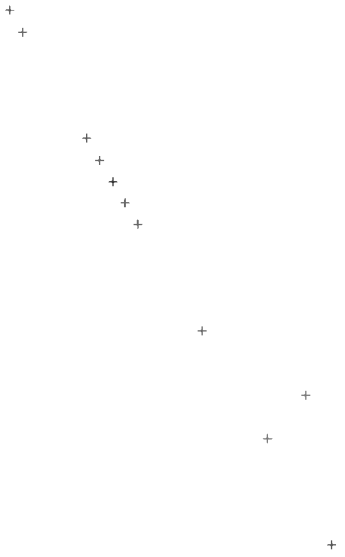
k-tup = 2



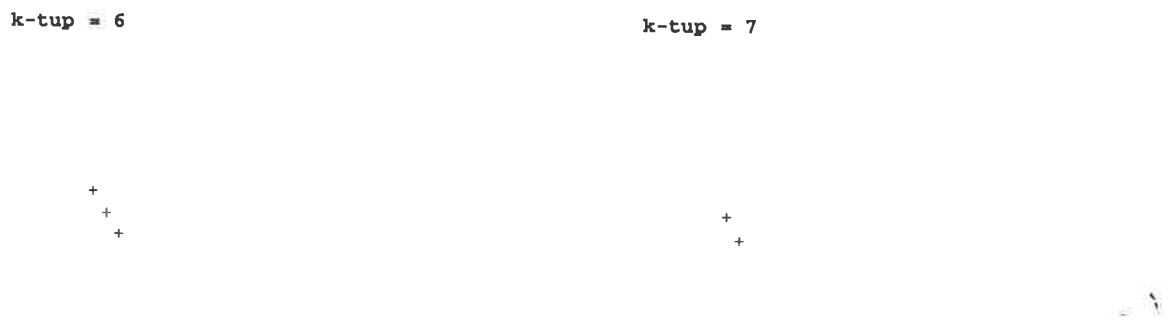
k-tup = 3



k-tup = 4



k-tup = 5



Para valores de k-tup mayores o iguales a 9, la matriz T está vacía.

Este mismo experimento fue realizado para diferentes tazas de mutación y cada uno de estos casos a su vez para diferentes valores de k-tup.

4.3.1.2. DISCUSIÓN DE LOS RESULTADOS

Si consideramos que una taza de mutación del 10% indica que una de cada diez bases estará cambiada, es de esperarse que cada diez bases aparezca una interrupción en las marcas pertenecientes a la diagonal principal. Así, si incrementamos en valor del k-tup, las subcadenas que se comparan entre sí son cada vez más largas y sólo dejará de marcar en la diagonal principal aquella subcadena que contenga la mutación. Pero cuando la longitud de la subcadena, o sea el k-tup, se acerca a la distancia esperada entre dos mutaciones, es de esperarse que la matriz no contenga coincidencias, porque todas las subcadenas contienen mutaciones que hacen que no existan coincidencias.

Lo real es que una taza de mutación del 10% no indica estrictamente que una de cada diez bases estarán cambiadas, y es por esto que es un valor de k-tup = 9 y no k-tup = 10 el que hace que la matriz quede totalmente vacía.

Podría decirse entonces, como una regla general, que el valor del k-tup no debe superar el la frecuencia de aparición de mutaciones, pero si puede ser menor.

Por otra parte, vemos que para valores reducidos del k-tup la matriz T contiene muchas marcas, lo cual eleva el costo computacional de los algoritmos. Pero se puede observar que existe un rango de valores de k-tup que hacen que el tiempo de cómputo sea reducido, y a la vez no se

pierde la información relevante para el algoritmo de reconstrucción. Obsérvese que si bien la tabla para k-tup = 3 tiene menos marcas que la del k-tup = 2, no deja de mostrar claramente que sobre la diagonal principal existe una alineación importante. De esta forma, vemos que un valor apropiado del k-tup nos puede ayudar a reducir el costo temporal, sin perder la información que los algoritmos de reconstrucción necesitan.

4.3.2. EXPERIENCIA 2

Para determinar el costo temporal del algoritmo se consideró que éste estará directamente relacionado con la cantidad de coincidencias que aparezcan en la tabla T, debido a que para cada una de las marcas en la tabla, se realizará una operación costosa que dependerá del algoritmo, pero que en general es una suma. Así, la cantidad de posiciones marcadas como coincidencias en la matriz T se utiliza como indicador del costo temporal.

El procedimiento de experimentación sería entonces:

- Generar dos secuencias que sólo difieran entre sí en unas pocas mutaciones.
- Aplicarles a estas secuencias la función MatrizT1 y observar cómo varía la cantidad de '`1`' en la matriz al variar el k-tup.
- Repetir el experimento para diferentes índices de mutación.

4.3.2.1. RESULTADOS

Los resultados que se muestran fueron obtenidos para secuencias de 100 nucleótidos, y para cada valor de índice de mutación se realizaron 100 experiencias.

Las gráficas obtenidas se muestran en el Anexo I.

4.3.2.2. DISCUSIÓN DE LOS RESULTADOS

Observando el comportamiento de la curva que describe la gráfica de la cantidad de puntos que contiene la matriz T en función del parámetro k-tup, vemos que se aproxima a una forma exponencial negativa. La curva se ha ajustado correctamente con una función de la forma:

$$y = k e^{-ax} + C$$

Para cada caso en particular, dado el valor de la longitud de las secuencias, se pueden obtener los valores particulares de k, a y C.

Para este caso particular estudiado, vemos que a partir de un k-tup = 4 aprox., no se obtiene un beneficio significativo en cuanto a la reducción del costo, al incrementar el k-tup. Esto puede sugerir un límite razonable al momento de decidir incrementar el valor de este parámetro.

5. CONCLUSIONES

De los resultados anteriormente expuestos, y relacionando las dos experiencias, podemos observar que existe un límite razonable para el valor del k-tup, que debe ser establecido para cada caso, pero que no es excesivamente elevado, debido al comportamiento exponencial decreciente. Así, sabemos por la Experiencia 1 que no es conveniente elevar el k-tup porque se eliminan muchos puntos útiles de la matriz T, aunque no es conveniente que sea demasiado bajo, porque la matriz queda demasiado cargada de marcas. Pero además de un análisis morfológico, podemos arribar a la misma conclusión observando la Experiencia 2, que es cuantitativa, y muestra claramente que no se justifica elevar el k-tup por encima de un determinado valor porque no se obtiene un beneficio en costo temporal.

Por lo tanto, sabemos que existe un rango muy estrecho de valores de k-tup que son realmente útiles, porque se tiene un costo temporal aceptable, y además se entrega a los algoritmos aproximados una matriz lo suficientemente clara como para obtener buenos resultados.

Si deseamos obtener un costo lineal, sabemos que el k-tup está determinado por:

$$k - tup = \log_4 n$$

y para n = 100 es un valor de k-tup entre 3 y 4. Así, vemos que el valor de costo lineal es gráficamente un buen valor de k-tup, porque entrega una matriz limpia y tiene un buen costo temporal.

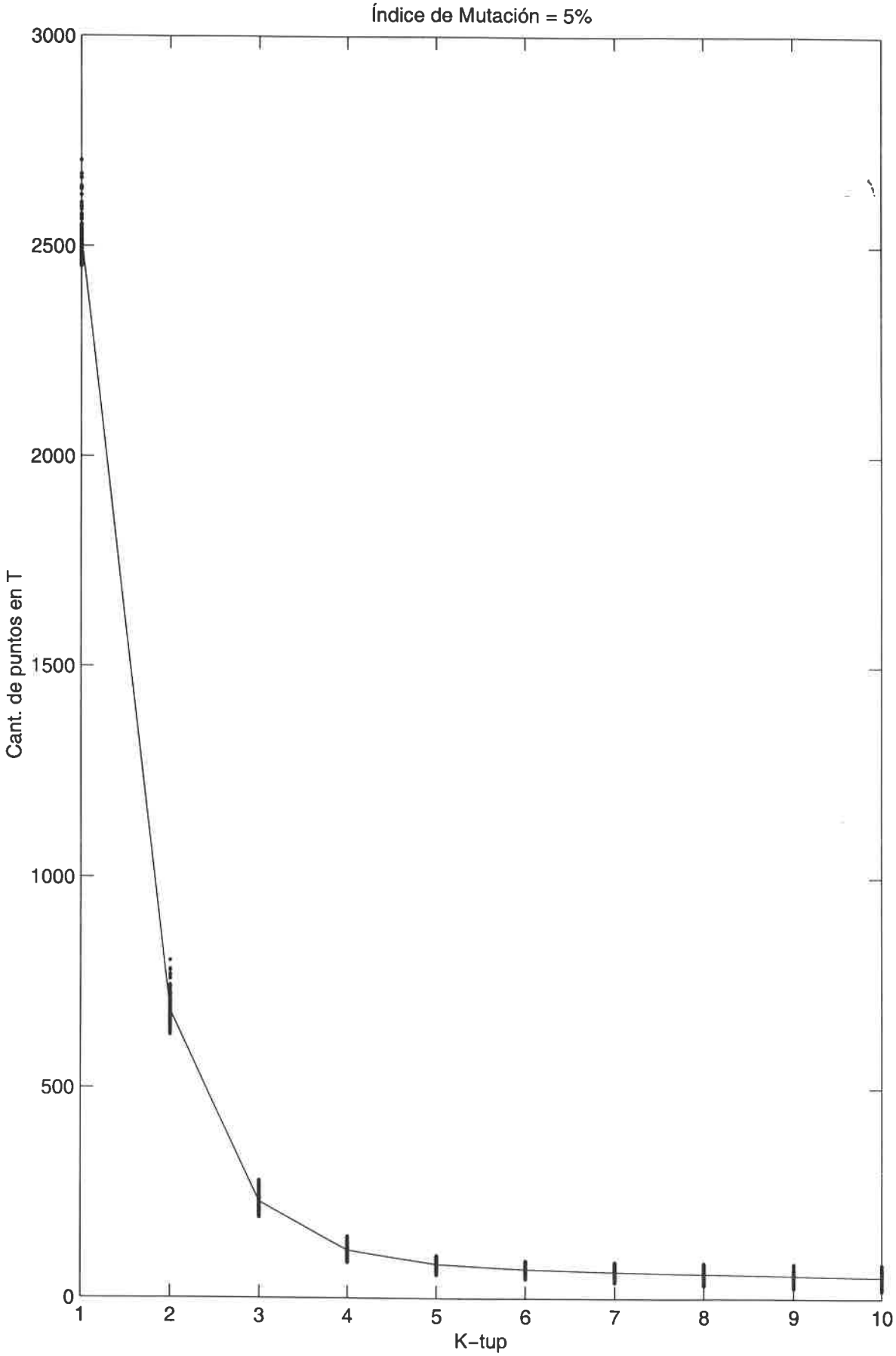
6. BIBLIOGRAFÍA

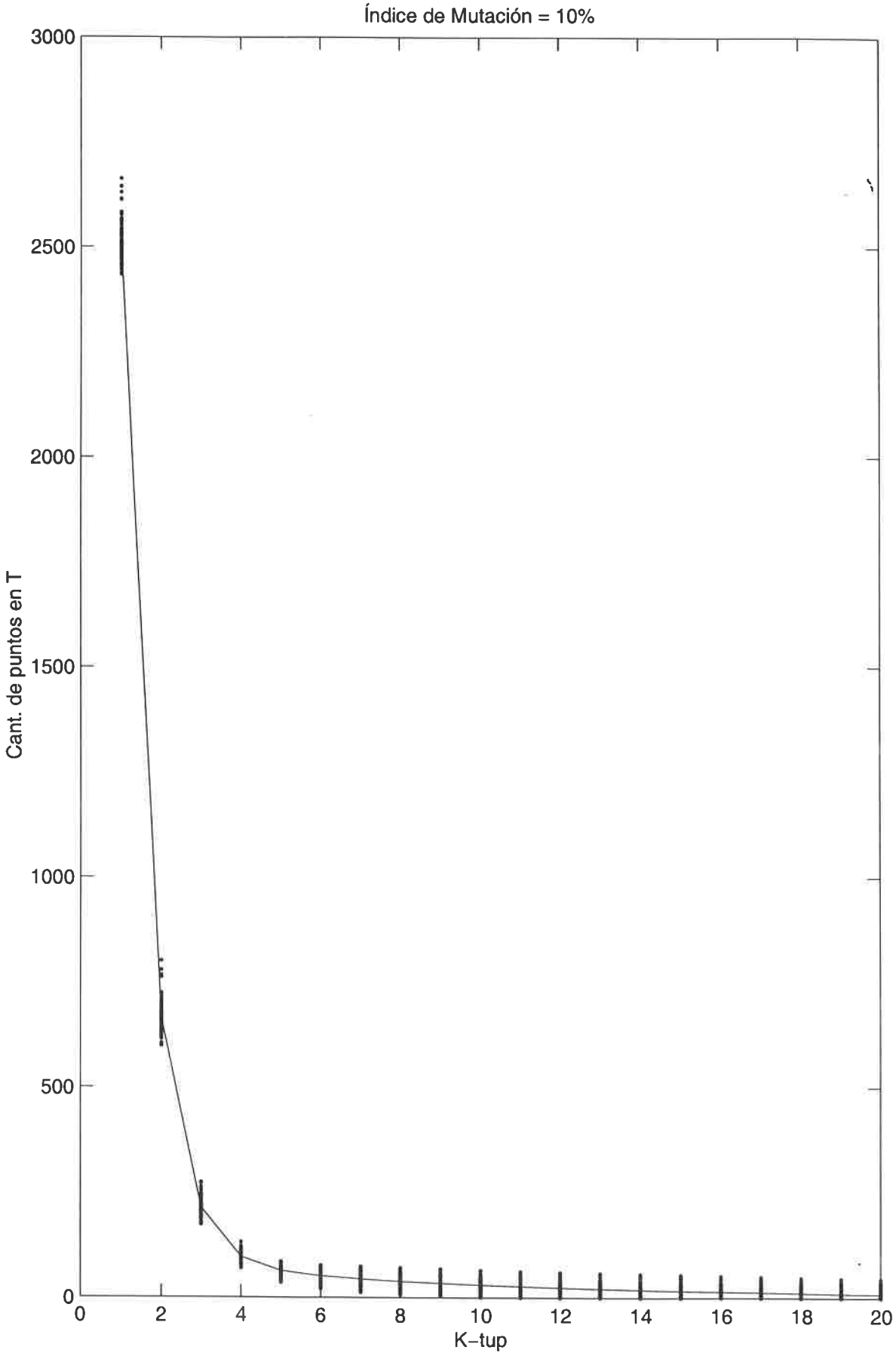
- ***Introduction to Computational Molecular Biology***
  Setubal/Meidanis – Books/Cole Publishing Company – 1997

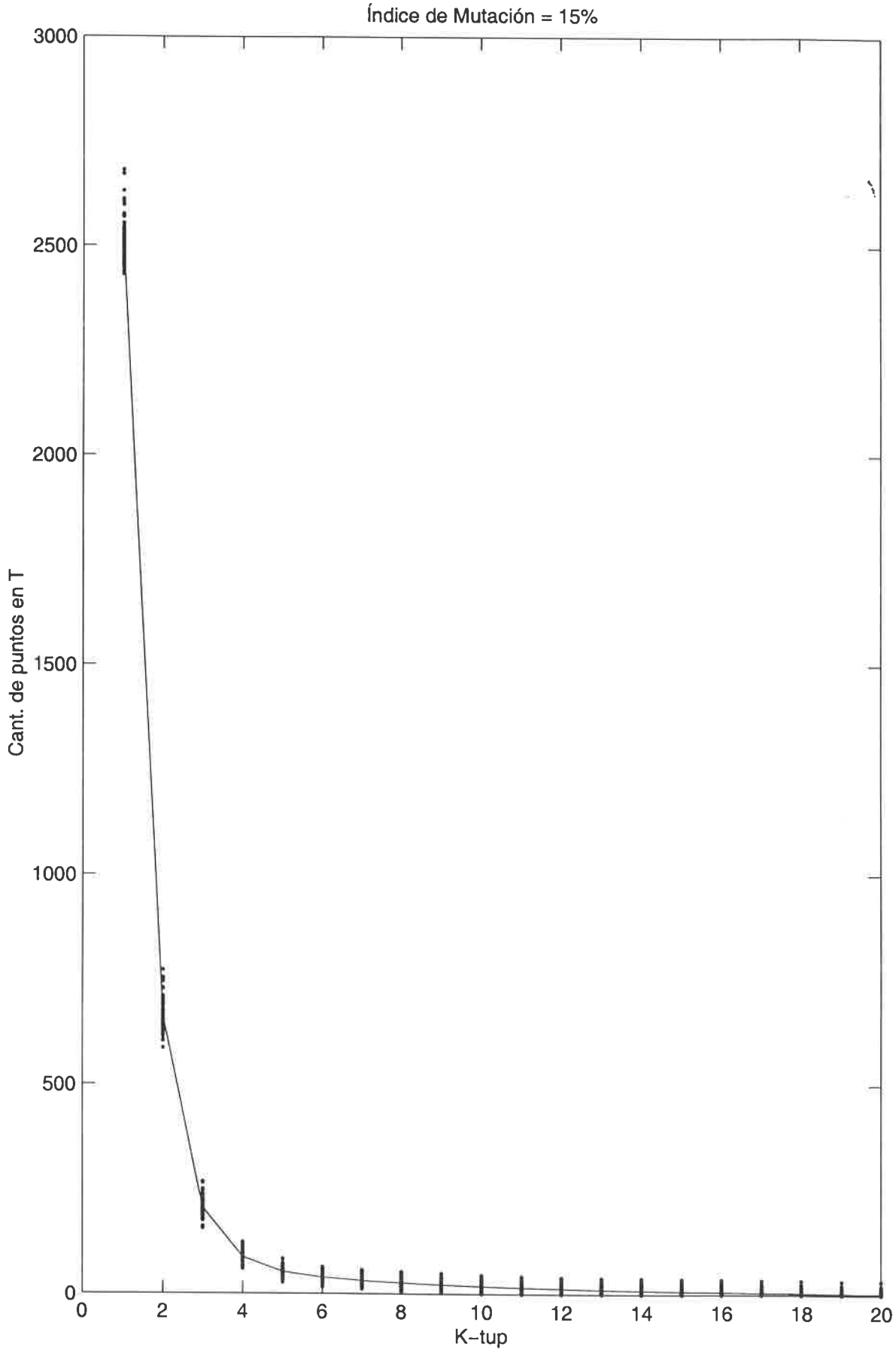
- ***Introduction to Computacional Biology***
  Michael Waterman – Chapman & Hall – 1995

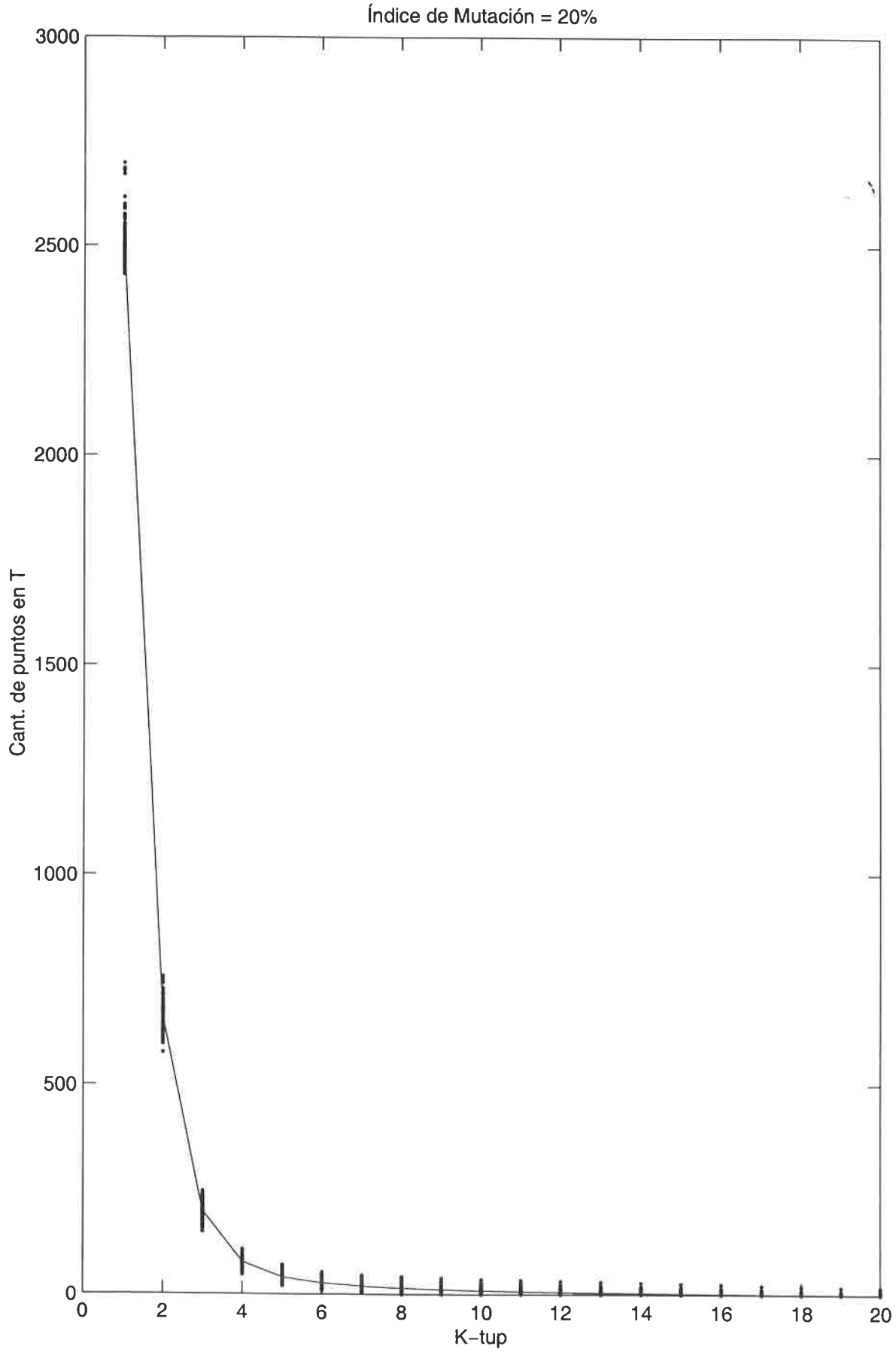
- ***Algoritmos paralelos de ensable de secuencias de DNA***
  David Rivas Roig – TFC - UPC

ANEXO I


Índice de Mutación = 5%
3000
2500
2000
1500
1000
500
0
Cant. de puntos en T
1
2
3
4
5
6
7
8
9
10
K-tup


Índice de Mutación = 10%
3000
2500
2000
1500
1000
500
0
Cant. de puntos en T
0
2
4
6
8
10
12
14
16
18
20
K-tup


Índice de Mutación = 15%
3000
2500
2000
1500
1000
500
0
Cant. de puntos en T
0
2
4
6
8
10
12
14
16
18
20
K-tup


Índice de Mutación = 20%
3000
2500
2000
1500
1000
500
0
Cant. de puntos en T
0
2
4
6
8
10
12
14
16
18
20
K-tup