UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de un analizador de biosecuencias: Análisis de mutaciones en nucleótidos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Martín Zamuner Antón

Director: José María Sempere Luna

2015-2016

Resumen

En el presente Trabajo Fin de Grado se desarrollará un analizador de secuencias de carácter bioquímico (fundamentalmente, secuencias de ADN y de ARN) para la búsqueda de posibles mutaciones debidas a la modificación, eliminación e inserción de nucleótidos. El marco teórico de este trabajo se basa en la utilización de gramáticas formales (incontextuales), autómatas finitos y algoritmos de análisis eficientes previamente formulados. El analizador se desarrollará en *Java* y se pretende obtener una herramienta que trabaje en modo *standalone*. Posteriormente, se evaluará su posible integración en un *website* para su ejecución en red.

Palabras clave: Bioinformática, Algorítmica, Lenguajes Formales, Alineamientos.

Abstract

The aim of this thesis is to develop a tool for analyzing biochemical sequences (specially DNA and RNA sequences) in order to find potential mutations caused by the modification, deletion or insertion of nucleotides. The theoretical framework of this project is based on the usage of context-free grammars, finite-state automata, and efficient analysis algorithms previously formulated. Said tool is to be developed using *Java* and it should work as standalone software. Subsequently, its integration in a website for use in a network will be evaluated.

Keywords: Bioinformatics, Algorithmics, Formal Languages, Aligments.

Índice general

1. Introducción

En el desarrollo del trabajo de investigación en el campo de la biología y, más concretamente, el área de estudio de la genética, se requiere la capacidad de analizar secuencias de proteínas o de ADN. Esto se debe a que uno de los descubrimientos más importantes de la ciencia moderna es el hecho de que todos los seres vivos utilizan genes similares para proveer sus distintas funciones biológicas.

La complejidad de estas estructuras moleculares, junto con la inmensa cantidad en la que son encontradas, hacen que este proceso sea prácticamente inabordable usando los métodos de análisis tradicionalmente ejecutados por seres humanos.

Este tipo de problemas es el objetivo perfecto para la automatización que las máquinas de cómputo proveen. Sin embargo, incluso con la ayuda de computadoras de alto rendimiento se necesitaron 50 años desde el descubrimiento de la estructura del ADN en 1953 hasta que la tecnología permitió la secuenciación de un genoma humano completo en 2003.

Es en este contexto que surge la idea de este trabajo; desarrollar un analizador de biosecuencias que permita encontrar patrones y resaltar diferencias entre cadenas de ADN y ARN.

Concretamente, el objetivo central del proyecto es brindar al usuario la capacidad de alinear dos cadenas de forma óptima. Esto es importante ya que si dos secuencias pueden ser alineadas, existe una alta probabilidad de que compartan una relación evolutiva.

2. Marco teórico

Para ser capaces tanto de entender el funcionamiento de la herramienta desarrollada como de describir los algoritmos que la sustentan, es importante tener una serie de conocimientos teóricos básicos.

Empezaremos estudiando los lenguajes formales, ya que constituyen la piedra fundamental del trabajo, para luego adentrarnos en los conceptos que sustentan la noción de alineamiento entre dos cadenas.

2.1. Lenguajes formales

En esta sección, basada íntegramente en [3], [1] y [2], se introducirán los conceptos de *alfabeto* (un conjunto de símbolos), *cadena de caracteres* (una secuencia de símbolos de un alfabeto) y *lenguaje* (un conjunto de cadenas de caracteres de un mismo alfabeto), así como la notación utilizada para describirlos y las operaciones que es posible realizar con ellos.

2.1.1. Alfabetos

Un *alfabeto* es un conjunto de símbolos finito y no vacío cuyos elementos son llamados *símbolos* o *letras*. Convencionalmente, utilizamos la letra griega Σ para designar un alfabeto. Entre los alfabetos más comunes se incluyen los siguientes:

1. $\Sigma = \{0, 1\}$, el alfabeto *binario*.
2. $\Sigma = \{a, b, \ldots, z\}$, el conjunto de todas las letras minúsculas.
3. El conjunto de todos los caracteres *ASCII* o el conjunto de todos los caracteres *ASCII* imprimibles.

2.1.2. Cadenas de caracteres

Una *cadena de caracteres*, o *palabra*, es una secuencia finita de símbolos seleccionados de algún alfabeto. Para simplificar su escritura, podemos obviar la notación comúnmente utilizada para representar secuencias y escribir una cadena como la yuxtaposición de los símbolos que la componen. De esta manera, $(0, 1, 1, 0, 1)$ y 01101 representan la misma cadena del alfabeto binario $\Sigma = \{0, 1\}$. La cadena 111 es otra cadena de dicho alfabeto.

La *cadena vacía* es aquella cadena que presenta cero apariciones de símbolos. Esta cadena, designada por ε, es una cadena que puede construirse en cualquier alfabeto.

Longitud de una cadena

Suele resultar útil clasificar las cadenas por su *longitud*, es decir, el número de posiciones ocupadas por símbolos dentro de la cadena. Por ejemplo, 01101 tiene una longitud de 5. Es habitual decir que la longitud de una cadena es igual al «número de símbolos» que contiene; esta proposición, aunque se suela aceptar coloquialmente, no es estrictamente correcta. Así, en la cadena 01101 sólo hay dos símbolos, 0 y 1, aunque tiene cinco *posiciones* para los mismos y su longitud es igual a 5. Sin embargo, generalmente podremos utilizar la expresión «número de símbolos» cuando realmente a lo que se esta haciendo referencia es al «número de posiciones».

La notación estándar para indicar la longitud de una cadena v es $|v|$. Por ejemplo, $|011| = 3$ y $|\varepsilon| = 0$.

Si Σ es un alfabeto, podemos expresar el conjunto de todas las cadenas de una determinada longitud de dicho alfabeto utilizando una notación exponencial. Definimos Σ^k para que sea el conjunto de las cadenas de longitud k, tales que cada uno de los símbolos de las mismas pertenece a Σ.

Por convenio, el conjunto de todas las cadenas de un alfabeto Σ se designa mediante Σ^*. Por ejemplo, $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$.

Concatenación de cadenas

También nos resultará útil la capacidad de hacer referencia al símbolo que ocupa una posición determinada de la cadena. Definimos $w[i]$ para $i = 0, 1, \dots, |w|-1$, como el símbolo que ocupa la posición indicada por el índice i en la cadena w. Por convenio, los índices empiezan en 0. Por ejemplo, $w[3] = 0$ dada la cadena $w = 01101$.

Sean x e y dos cadenas. Entonces, xy denota la *concatenación* de x e y, es decir, la cadena formada por una copia de x seguida de una copia de y. Dicho de manera más precisa, si x es la cadena compuesta por i símbolos $x = a_1 a_2 \cdots a_i$ e y es la cadena formada por j símbolos $y = b_1 b_2 \cdots b_j$, entonces xy es la cadena de longitud $i + j$: $xy = a_1 a_2 \cdots a_i b_1 b_2 \cdots b_j$.

2.1.3. Lenguajes

Un conjunto de cadenas, todas ellas seleccionadas de un Σ^*, donde Σ es un determinado alfabeto se denomina *lenguaje*. Si Σ es un alfabeto y $L \subseteq \Sigma^*$, entonces L es un *lenguaje de* Σ. Observe que un lenguaje de Σ no necesita incluir cadenas con todos los símbolos de Σ, ya que una vez que hemos establecido que L es un lenguaje de Σ, también sabemos que es un lenguaje de cualquier alfabeto que sea un superconjunto de Σ.

La elección del término *lenguaje* puede parecer extraña. Sin embargo, los lenguajes habituales pueden interpretarse como conjuntos de cadenas. Un ejemplo sería

el inglés, donde la colección de las palabras inglesas correctas es un conjunto de cadenas del alfabeto que consta de todas las letras. Otro ejemplo es el lenguaje *Java*, o cualquier otro lenguaje de programación, donde los programas correctos son un subconjunto de las posibles cadenas que pueden formarse a partir del alfabeto del lenguaje. Este alfabeto es un subconjunto de los caracteres *ASCII*. El alfabeto en concreto puede diferir ligeramente entre diferentes lenguajes de programación, aunque generalmente incluye las letras mayúsculas y minúsculas, los dígitos, los caracteres de puntuación y los símbolos matemáticos.

Algunos ejemplos de lenguajes son los siguientes:

1. El lenguaje de todas las cadenas que constan de n ceros seguidos de n unos para cualquier $n \geq 0$: $\{\varepsilon, 01, 0011, 000111, \dots\}$.
2. El conjunto de cadenas formadas por el mismo número de ceros que de unos: $\{\varepsilon, 01, 10, 0011, 0101, 1001, \dots\}$.
3. El conjunto de números binarios cuyo valor es un número primo: $\{10, 11, 101, 111, 1011, \dots\}$.
4. Σ^* es un lenguaje para cualquier alfabeto Σ.
5. $\emptyset$, el lenguaje vacío, es un lenguaje de cualquier alfabeto.
6. $\{\varepsilon\}$, el lenguaje que consta sólo de la cadena vacía, también es un lenguaje de cualquier alfabeto. Observe que $\emptyset \neq \{\varepsilon\}$; el primero no contiene ninguna cadena y el segundo sólo tiene una cadena.

Notación de conjuntos

Es habitual describir un lenguaje utilizando una «descripción de conjuntos»:

$$\{w \mid \text{algo acerca de } w\}$$

Esta expresión se lee «el conjunto de palabras w tal que (lo que se dice acerca de w a la derecha de la barra vertical)». Algunos ejemplos son:

1. $\{w \mid w$ consta de un número igual de ceros que de unos$\}$.
2. $\{w \mid w$ es un número binario que es primo$\}$.
3. $\{w \mid w$ es un programa *Java* sintácticamente correcto$\}$.

También es habitual reemplazar w por alguna expresión con parámetros y describir las cadenas del lenguaje estableciendo condiciones sobre los parámetros. He aquí algunos ejemplos:

1. $\{0^n 1^n \mid n \geq 1\}$.
2. $\{0^i 1^j \mid 0 \leq i \leq j\}$.

La única restricción importante sobre lo que puede ser un lenguaje es que todos los alfabetos son finitos. De este modo, los lenguajes, aunque pueden tener un número infinito de cadenas, están restringidos a que dichas cadenas estén formadas por los símbolos definidos por un alfabeto finito y prefijado.

2.2. Alineamientos

Los alineamientos constituyen uno de los procesos usados comúnmente para comparar cadenas de texto, permitiéndonos visualizar el parecido entre estas.

Se utilizan en bioinformática para representar y comparar secuencias de ADN y ARN, o cadenas de proteínas. El objetivo es identificar regiones de similitud que puedan ser consecuencia de relaciones funcionales, estructurales o evolutivas entra las secuencias.

2.2.1. Distancias

La idea formal de alineamiento está basada en los conceptos de distancia o similitud entre las cadenas.

Para formalizar estas ideas nos interesa definir la noción de parecido o similitud entre dos cadenas x e y de longitudes m y n, respectivamente. Decimos que una función $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$ es una distancia en Σ^* si se satisfacen las siguientes propiedades para todo $u, v \in \Sigma^*$:

- Positividad: $d(u, v) \geq 0$.
- Separabilidad: $d(u, v) = 0$ si y sólo si $u = v$.
- Simetría: $d(u, v) = d(v, u)$.
- Desigualdad triangular: $d(u, v) \leq d(u, w) + d(w, v)$ para todo $w \in \Sigma^*$.

2.2.2. Operaciones de edición

Las distancias con las que trabajaremos se definen a partir de operaciones que transforman una cadena x en otra cadena y. Trataremos con tres tipos de operaciones elementales, llamadas operaciones de edición:

- Sustitución de una letra en una posición determinada de x por otra letra de y.
- Eliminación de una letra en una posición determinada de x.
- Inserción de una letra de y en una posición determinada de x.

Asignamos un coste (un valor entero positivo) a cada una de las operaciones. Para $a, b \in \Sigma$, definimos:

- $Sub(a, b)$: coste de sustituir la letra b por la letra a.
- $Del(a)$: coste de eliminar la letra a.
- $Ins(b)$: coste de insertar la letra b.

Se asume implícitamente que estos costes son independientes de la posición en la que se lleva a cabo la operación.

2.2.3. Distancia de edición

A partir de estos costes elementales, definimos

$$Lev(x, y) = \text{mín}\{\text{coste de } \phi : \phi \in \Phi_{x,y}\},$$

donde $\Phi_{x,y}$ es el conjunto de secuencias de operaciones de edición elementales que transforman x en y, y el coste de un elemento $\phi \in \Phi_{x,y}$ es la suma de los costes de las operaciones de edición de la secuencia ϕ. La función Lev es una distancia en Σ^* y se le suele llamar *distancia de edición*.

2.2.4. Definición

Un *alineamiento* entre dos cadenas $x, y \in \Sigma^*$, cuyas longitudes son m y n, es una forma de visualizar sus similitudes. Formalmente, un alineamiento entre x e y es una cadena z del alfabeto de pares de letras. Concretamente,

$$(\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}),$$

cuya proyección en el primer componente es x y su proyección en el segundo componente es y. Por tanto, si z es un alineamiento de longitud p entre x e y, tenemos

$$\begin{aligned} z &= (\bar{x_0}, \bar{y_0})(\bar{x_1}, \bar{y_2}) \dots (\bar{x}_{p-1}, \bar{y}_{p-1}), \\ x &= \bar{x_0}\bar{x_1} \dots \bar{x}_{p-1}, \\ y &= \bar{y_0}\bar{y_1} \dots \bar{y}_{p-1}, \end{aligned}$$

con $\bar{x}_i \in \Sigma \cup \{\epsilon\}$ e $\bar{y}_i \in \Sigma \cup \{\epsilon\}$ para $i = 0, 1, \dots, p-1$.

Para mayor comodidad, utilizaremos la siguiente notación para representar un alineamiento:

$$\begin{pmatrix} \bar{x_0} & \bar{x_1} & \dots & \bar{x}_{p-1} \\ \bar{y_0} & \bar{y_1} & \dots & \bar{y}_{p-1} \end{pmatrix}$$

3. Algoritmos

Construyendo sobre la teoría vista en el capítulo anterior, podremos definir y entender los algoritmos que funcionan como motor de la herramienta desarrollada en este trabajo. La totalidad de los algoritmos tratados proviene del capítulo 7 de [1].

3.1. Alineamientos óptimos

En esta sección se presenta el método en el que se basa el cálculo de un alineamiento óptimo entre dos cadenas de texto. El proceso utiliza una técnica central para las ciencias computacionales llamada programación dinámica. Esta técnica consiste en memorizar los resultados de cálculos intermedios con el fin de evitar tener que volver a calcularlos.

La producción de un alineamiento entre dos cadenas x e y se basa en el cálculo de la distancia de edición entre dos palabras. Tiene sentido, por tanto, empezar explicando cómo llevar a cabo este proceso.

3.1.1. Cálculo de la distancia de edición

Dadas dos cadenas $x, y \in A^*$ de longitudes m y n, definimos una tabla T de $m+1$ filas y $n+1$ columnas de la siguiente forma

$$T[i,j] = Lev(x[0 \dots i], y[0 \dots j])$$

para $i = -1, 0, \dots, m-1$ y $j = -1, 0, \dots, n-1$.

Esta tabla será la base de todos los algoritmos que veremos en este capítulo. Para empezar, observemos que $T[i,j]$ es el coste mínimo de un camino empezando en el vértice $(-1,-1)$ y acabando en (i,j) en el grafo de edición $G(x,y)$. Otro resultado que salta a la vista es que $T[m-1, n-1]$ será igual a la distancia de edición entre x e y; esto es, $T[m-1, n-1] = Lev(x,y)$.

El algoritmo Generic-DP ejecuta el cálculo de la tabla T, que guardaremos para futuros algoritmos, y retorna, además, la distancia de edición entre las dos palabras que requiere como parámetros.

1: **function** GENERIC-DP(x, m, y, n)
2: $T[-1,-1] \leftarrow 0$
3: **for** $i \leftarrow 0$ to $m-1$ **do**
4: $T[i,-1] \leftarrow T[i-1,-1] + Del(x[i])$
5: **for** $j \leftarrow 0$ to $n-1$ **do**
6: $T[-1,j] \leftarrow T[-1,j-1] + Ins(y[j])$
7: **for** $i \leftarrow 0$ to $m-1$ **do**
8: $$T[i,j] \leftarrow \text{mín} \begin{cases} T[i-1,j-1] + Sub(x[i], y[j]) \\ T[i-1,j] + Del(x[i]) \\ T[i,j-1] + Ins(y[j]) \end{cases}$$
9: **return** $T[m-1,n-1]$

A continuación se muestra la tabla T generada al ejecutar el algoritmo GENERIC-DP con parámetros $x =$ `EAWACQGKL`, $y =$ `ERDAWCQPGKWY`.

T	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	E	R	D	A	W	C	Q	P	G	K	W	Y
-1	$x[i]$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	E	1	0	1	2	3	4	5	6	7	8	9	10	11
1	A	2	1	2	3	2	3	4	5	6	7	8	9	10
2	W	3	2	3	4	3	2	3	4	5	6	7	8	9
3	A	4	3	4	5	4	3	4	5	6	7	8	9	10
4	C	5	4	5	6	5	4	3	4	5	6	7	8	9
5	Q	6	5	6	7	6	5	4	3	4	5	6	7	8
6	G	7	6	7	8	7	6	5	4	5	4	5	6	7
7	K	8	7	8	9	8	7	6	5	6	5	4	5	6
8	L	9	8	9	10	9	8	7	6	7	6	5	6	7

En esta tabla se pueden observar varios resultados interesantes. Para empezar, el valor de retorno del algoritmo, $T[8,11] = 7$. Como se ha indicado anteriormente, este valor coincide con la distancia de edición entre las dos palabras.

También se han marcado en la tabla los caminos de coste mínimo entre las posiciones $[-1,-1]$ y $[8,11]$. Esto nos permite determinar los tres alineamientos óptimos entre las dos cadenas utilizadas como parámetros del algoritmo:

$$\begin{pmatrix} \texttt{E} & \texttt{-} & \texttt{-} & \texttt{A} & \texttt{W} & \texttt{A} & \texttt{C} & \texttt{Q} & \texttt{-} & \texttt{G} & \texttt{K} & \texttt{-} & \texttt{-} & \texttt{L} \\ \texttt{E} & \texttt{R} & \texttt{D} & \texttt{A} & \texttt{W} & \texttt{-} & \texttt{C} & \texttt{Q} & \texttt{P} & \texttt{G} & \texttt{K} & \texttt{W} & \texttt{Y} & \texttt{-} \end{pmatrix}$$

$$\begin{pmatrix} \texttt{E} & \texttt{-} & \texttt{-} & \texttt{A} & \texttt{W} & \texttt{A} & \texttt{C} & \texttt{Q} & \texttt{-} & \texttt{G} & \texttt{K} & \texttt{-} & \texttt{L} & \texttt{-} \\ \texttt{E} & \texttt{R} & \texttt{D} & \texttt{A} & \texttt{W} & \texttt{-} & \texttt{C} & \texttt{Q} & \texttt{P} & \texttt{G} & \texttt{K} & \texttt{W} & \texttt{-} & \texttt{Y} \end{pmatrix}$$

$$\begin{pmatrix} \texttt{E} & \texttt{-} & \texttt{-} & \texttt{A} & \texttt{W} & \texttt{A} & \texttt{C} & \texttt{Q} & \texttt{-} & \texttt{G} & \texttt{K} & \texttt{L} & \texttt{-} & \texttt{-} \\ \texttt{E} & \texttt{R} & \texttt{D} & \texttt{A} & \texttt{W} & \texttt{-} & \texttt{C} & \texttt{Q} & \texttt{P} & \texttt{G} & \texttt{K} & \texttt{-} & \texttt{W} & \texttt{Y} \end{pmatrix}$$

Coste computacional

La razón por la que la programación dinámica es una técnica tan utilizada es que nos permite reducir el coste temporal de un algoritmo. La relación de recurrencia en la que se basa el algoritmo Generic-DP produciría un coste temporal exponencial si decidiéramos implementarla en un algoritmo sin ninguna consideración por su crecimiento asintótico.

Sin embargo, el algoritmo Generic-DP, aplicado a dos cadenas de longitudes m y n, tiene un coste temporal $O(m \times n)$ y un coste espacial $O(\text{mín}\{m, n\})$.

Esto se debe a que el cálculo del valor para cada posición de la tabla T depende únicamente de las tres posiciones vecinas y tiene un coste temporal constante. Una vez inicializada la tabla, lo que tiene un coste temporal $O(m+n)$, se llevan a cabo $m \times n$ cálculos de este tipo, dando como resultado el coste temporal mencionado anteriormente. El hecho de que dependa únicamente de las tres posiciones vecinas significa, también, que solo hará falta almacenar dos columnas (o filas) en un momento dado, lo que resulta en el coste espacial indicado.

3.1.2. Cálculo de un alineamiento óptimo

Como hemos visto anteriormente, el algoritmo Generic-DP solo calcula el coste de transformar la cadena x en y. Si deseamos obtener una secuencia de operaciones de edición que produzca tal transformación, o su alineamiento correspondiente, tendremos que encontrar los caminos en la tabla T que partan desde la posición final $[m-1, n-1]$ hacia la posición inicial $[-1, -1]$.

Supongamos que nos encontramos en la celda $[i, j]$. Ya que lo que nos interesa es retroceder en la dirección de la posición inicial, procederemos a visitar las celdas vecinas ($[i-1, j-1]$, $[i-1, j]$ y $[i, j-1]$), identificando la posición cuyo valor asociado produce $T[i, j]$. Repetiremos este proceso iterativamente hasta llegar a la celda $[-1, -1]$. El algoritmo One-Alignment, cuyo código se muestra más adelante, implementa este método, produciendo así un alineamiento óptimo.

La validez de este proceso depende de la noción de *arco activo* del grafo de edición $G(x, y)$. Un arco activo es aquel que puede ser considerado para la producción de un alineamiento óptimo. Formalmente, diremos que el arco $((i', j'), (i, j))$, etiquetado como (a, b), es activo cuando

$$\begin{aligned} T[i,j] &= T[i',j'] + Sub(a,b) \text{ si } i - i' = j - j' = 1, \\ T[i,j] &= T[i',j'] + Del(a) \text{ si } i - i' = 1 \text{ y } j = j', \\ T[i,j] &= T[i',j'] + Ins(b) \text{ si } i = i' \text{ y } j - j' = 1, \end{aligned}$$

para $i, i' \in \{-1, 0, \ldots, m-1\}$, $j, j' \in \{-1, 0, \ldots, n-1\}$ y $a, b \in \Sigma$.

Utilizando este concepto, One-Alignment parte del vértice final del grafo $G(x, y)$, subiendo a través de los arcos activos y finalizando cuando encuentra el vértice inicial $(-1, -1)$. Se asume que la variable z del algoritmo es una cadena del alfabeto $(\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\})$, y que en dicho alfabeto la concatenación se hace componente a componente.

```
1: function One-Alignment(x, m, y, n, T)
2:     z ← (ϵ, ϵ)
3:     (i, j) ← (m − 1, n − 1)
4:     while i ≠ −1 and j ≠ −1 do
5:         if T[i, j] = T[i − 1, j − 1] + Sub(x[i], y[j]) then
6:             z ← (x[i], y[j]) · z
7:             (i, j) ← (i − 1, j − 1)
8:         else if T[i, j] = T[i − 1, j] + Del(x[i]) then
9:             z ← (x[i], ϵ) · z
10:            i ← i − 1
11:        else
12:            z ← (ϵ, y[j]) · z
13:            j ← j − 1
14:    while i ≠ −1 do
15:        z ← (x[i], ϵ) · z
16:        i ← i − 1
17:    while j ≠ −1 do
18:        z ← (ϵ, y[j]) · z
19:        j ← j − 1
20:    return z
```

Una consideración importante sobre este algoritmo es que el orden en que han sido escritas las tres instrucciones condicionales es arbitrario. Existen, así pues, $3! = 6$ posibles formas de escribir las líneas 5-13, cada una favoreciendo un camino distinto en el grafo $G(x, y)$.

Esto significa, por tanto, que cada una de estas variaciones producirá un alineamiento óptimo distinto, debido a que estamos seleccionando un solo camino en el grafo. En la próxima sección se verá cómo generar todos los posibles alineamientos óptimos.

Coste computacional

El algoritmo One-Alignment tiene un coste temporal y espacial $O(m + n)$.

Obsérvese que todas las operaciones significativas del algoritmo acaban decrementando las variables i y/o j, las cuales varían desde $m - 1$ y $n - 1$, respectivamente, hasta -1. Esto resulta en el coste temporal mencionado, $O(m+n)$. El coste espacial es consecuencia del almacenamiento de los dos componentes del alineamiento.

3.1.3. Cálculo de todos los alineamientos óptimos

El siguiente algoritmo produce todos los alineamientos óptimos entre dos cadenas x e y. Simplemente inicia un proceso de llamadas recursivas al procedimiento Al, cuyo código aparece en la siguiente página y en el cual se asume que las variables T, x e y son accesibles globalmente.

```
1: function Alignments(x, m, y, n, T)
2:     Al(m − 1, n − 1, (ϵ, ϵ))
```

```
1: function AL(i, j, z)
2:     if i ≠ −1 and j ≠ −1
         and T[i, j] = T[i − 1, j − 1] + Sub(x[i], y[j]) then
3:         AL(i − 1, j − 1, (x[i], y[j]) · z)
4:     if i ≠ −1
         and T[i, j] = T[i − 1, j] + Del(x[i]) then
5:         AL(i − 1, j, (x[i], ϵ) · z)
6:     if j ≠ −1
         and T[i, j] = T[i, j − 1] + Ins(y[j]) then
7:         AL(i, j − 1, (ϵ, y[j]) · z)
8:     if i = −1 and j = −1 then
9:         señalizar que z es un alineamiento óptimo
```

Al ser un algoritmo recursivo, cada nueva llamada al procedimiento AL crea una rama que, a su vez, puede desembocar en otras llamadas. Nótese que en este algoritmo, a diferencia de ONE-ALIGNMENT, todos los condicionales son comprobados. Esto significa que se recorrerán todos los posibles caminos de coste mínimo en el grafo $G(x, y)$.

Cuando una rama de llamadas alcance el vértice inicial del grafo (línea 8), sabremos que la secuencia de transformaciones seguida es un alineamiento óptimo y lo señalizaremos como tal. En este contexto, señalizar es una forma genérica de indicar el fin de un proceso, y se deja a la implementación la decisión de qué hacer con el resultado.

Coste computacional

El algoritmo ALIGMENTS tiene un coste temporal $O(m + n)$.

Aunque al ser un procedimiento recursivo pueda resultar menos claro a primera vista, obsérvese que su comportamiento es idéntico al del algoritmo ONE-ALIGNMENT. Todas las llamadas recursivas decrementan las variables i y/o j, las cuales varían desde $m-1$ y $n-1$, respectivamente, hasta -1. La diferencia respecto al algoritmo de la sección anterior es que ahora recorremos todos los posibles caminos en el grafo $G(x, y)$. Por tanto, el coste temporal será proporcional a la suma de las longitudes de todos los alineamientos producidos.

3.1.4. Cálculo de un alineamiento óptimo con huecos

Todos los algoritmos vistos hasta este punto hacen uso de la tabla T generada por GENERIC-DP. Dicho procedimiento no permite la penalización de secuencias de huecos. Dado que este es uno de los objetivos del proyecto, veamos cómo adaptarlo para cumplir este requisito.

Necesitaremos tres tablas: D, I y T. La tabla T será idéntica a la vista anteriormente. El valor $D[i, j]$ indicará el coste de un alineamiento óptimo entre $x[0 \dots i]$ e $y[0 \dots j]$, en el que se acaba eliminando letras de x. El valor $I[i, j]$ será similar, pero acabando con inserciones de letras de y.

Utilizaremos, además, una función afín para determinar la penalización que asignaremos a una secuencia concreta de huecos. Esta función será de la forma

$$gap(k) = g + h \times (k - 1)$$

siendo g y h constantes enteras positivas. Esto nos permite penalizar la apertura de un hueco con una cantidad g y penalizar de forma distinta la extensión de un hueco con una cantidad h. Normalmente, elegiremos valores tales que $h < g$.

El algoritmo Gap realiza el cálculo de estas tres tablas haciendo uso de la función $gap(k)$.

```
1:  function Gap(x, m, y, n)
2:      for i ← −1 to m − 1 do
3:          (D[i, −1], I[i, −1]) ← (∞, ∞)
4:      T[−1, −1] ← 0
5:      T[0, −1] ← g
6:      for i ← 1 to m − 1 do
7:          T[i, −1] ← T[i − 1, −1] + h
8:      T[−1, 0] ← g
9:      for j ← 1 to n − 1 do
10:         T[−1, j] ← T[−1, j − 1] + h
11:     for j ← 0 to n − 1 do
12:         (D[−1, j], I[−1, j]) ← (∞, ∞)
13:         for i ← 0 to m − 1 do
14:             D[i, j] ← mín{D[i − 1, j] + h, T[i − 1, j] + g}
15:             I[i, j] ← mín{I[i, j − 1] + h, T[i, j − 1] + g}
16:             t ← T[i − 1, j − 1] + Sub(x[i], y[j])
17:             T[i, j] ← mín{t, D[i, j], I[i, j]}
18:     return T[m − 1, n − 1]
```

Nótese que el algoritmo Generic-DP era un caso especial de Gap en el que $g = h$ y la función $gap(k)$ es lineal con el número de huecos.

Coste computacional

El algoritmo Gap tiene un coste temporal y espacial $O(m \times n)$.

Una de las razones por las que se decidió que la función $gap(k)$ fuera afín es que reduce el coste temporal del algoritmo. Si no existiera esta restricción el coste sería $O(m \times n \times (m+n))$. Sin embargo, aplicando esta restricción nos queda un algoritmo analíticamente idéntico a Generic-DP.

3.2. Subsecuencias comunes maximales

En esta sección se tratará el cálculo de subsecuencias de longitud maximal comunes a dos cadenas de texto. De la misma forma que hicimos con los alineamientos óptimos, utilizaremos programación dinámica para reducir el coste temporal de los algoritmos.

Empezaremos explicando cómo calcular la longitud de las subsecuencias comunes maximales.

3.2.1. Cálculo de la distancia de subsecuencia

Dadas dos cadenas $x, y \in A^*$ de longitudes m y n, definimos una tabla bidimensional S de $m+1$ filas y $n+1$ columnas, para $i = -1, 0, \ldots, m-1$ y $j = -1, 0, \ldots, n-1$, de la siguiente forma

$$S[i,j] = \begin{cases} 0 & \text{si } i = -1 \text{ o } j = -1, \\ lcs(x[0 \ldots i], y[0 \ldots j]) & \text{si no.} \end{cases}$$

Esta tabla será la base del resto de algoritmos que veremos en este capítulo. Por definición, vemos que $S[m-1, n-1]$ será igual a la distancia de subsecuencia entre x e y; esto es, $S[m-1, n-1] = lcs(x, y)$.

LCS-Simple ejecuta el cálculo de la tabla S que guardaremos para ser utilizado por el algoritmo que veremos en la próxima sección, y retorna, además, la longitud de las subsecuencias maximales comunes a x e y.

```
1:  function LCS-Simple(x, m, y, n)
2:      for i ← −1 to m − 1 do
3:          S[i, −1] ← 0
4:      for j ← 0 to n − 1 do
5:          S[−1, j] ← 0
6:          for i ← 0 to m − 1 do
7:              if x[i] = y[j] then
8:                  S[i, j] ← S[i − 1, j − 1] + 1
9:              else
10:                 S[i, j] ← máx{S[i − 1, j], S[i, j − 1]}
11:     return S[m − 1, n − 1]
```

A continuación se muestra la tabla S generada al ejecutar el algoritmo LCS-Simple con parámetros $x =$ `AGCTGA`, $y =$ `CAGATCAGAG`.

S	j	-1	0	1	2	3	4	5	6	7	8	9
i		$y[j]$	C	A	G	A	T	C	A	G	A	G
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0
0	A	0	0	1	1	1	1	1	1	1	1	1
1	G	0	0	1	2	2	2	2	2	2	2	2
2	C	0	1	1	2	2	2	3	3	3	3	3
3	T	0	1	1	2	2	3	3	3	3	3	3
4	G	0	1	1	2	2	3	3	3	4	4	4
5	A	0	1	2	2	3	3	3	4	4	5	5

Obsérvese el valor de retorno del algoritmo, $S[5,9] = 5$. Como se ha indicado anteriormente, este valor coincide con la distancia de subsecuencia entre las dos palabras. También se han marcado en la tabla los caminos de coste máximo entre las posiciones $[-1,-1]$ y $[5,9]$. Estos caminos serán usados por el algoritmo que veremos en la sección siguiente para determinar una subsecuencia común maximal.

Debido a que se utilizarán estos mismos ejemplos en el próximo capítulo como pruebas para la herramienta desarrollada, listamos, además, los cuatro alineamientos óptimos entre las cadenas x e y:

$$\begin{pmatrix} - & A & G & C & - & T & - & - & G & A & - \\ C & A & G & - & A & T & C & A & G & A & G \end{pmatrix}$$

$$\begin{pmatrix} - & A & G & - & - & C & T & - & G & A & - \\ C & A & G & A & T & C & - & A & G & A & G \end{pmatrix}$$

$$\begin{pmatrix} - & A & G & - & C & T & - & - & G & A & - \\ C & A & G & A & - & T & C & A & G & A & G \end{pmatrix}$$

$$\begin{pmatrix} - & A & G & - & - & C & - & T & G & A & - \\ C & A & G & A & T & C & A & - & G & A & G \end{pmatrix}$$

Coste computacional

El algoritmo LCS-Simple tiene un coste temporal y espacial $O(m \times n)$.

De forma similar al caso de la tabla T vista anteriormente, el cálculo del valor para cada posición de la tabla S depende únicamente de las tres posiciones vecinas y tiene un coste temporal constante. Una vez inicializada la tabla, lo que tiene un coste temporal $O(m+n)$, se llevan a cabo $m \times n$ cálculos de este tipo, dando como resultado el coste temporal mencionado.

El coste espacial está determinado por el tamaño de la tabla S, $m \times n$ celdas.

3.2.2. Cálculo de una subsecuencia común maximal

Habiendo calculado la tabla S, es posible encontrar una subsecuencia común maximal entre las cadenas x e y retrocediendo desde la posición final $[m-1, n-1]$ hasta la posición inicial $[-1,-1]$ a lo largo de un camino de peso máximo. Cada una de las celdas por las que pasemos que cumpla la condición $x[i] = y[j]$ nos llevará a añadir una letra a la subsecuencia producida.

El algoritmo One-LCS lleva a cabo este procedimiento, requiriendo como parámetro la tabla S, además de las cadenas x e y.

```
1: function ONE-LCS(x, m, y, n, S)
2:     z ← ϵ
3:     (i, j) ← (m − 1, n − 1)
4:     while i ≠ −1 and j ≠ −1 do
5:         if x[i] = y[j] then
6:             z ← x[i] · z
7:             (i, j) ← (i − 1, j − 1)
8:         else if S[i − 1, j] > S[i, j − 1] then
9:             i ← i − 1
10:        else
11:            j ← j − 1
12:    return z
```

La ejecución del algoritmo ONE-LCS con parámetros $x =$ `AGCTGA`, $y =$ `CAGATCAGAG` produce la subsecuencia `AGTGA`.

Coste computacional

El algoritmo ONE-LCS tiene un coste temporal $O(m + n)$ y un coste espacial $O(\text{mín}\{m, n\})$.

Todas las operaciones significativas del algoritmo acaban decrementando las variables i y/o j, las cuales varían desde $m - 1$ y $n - 1$, respectivamente, hasta -1. Esto resulta en el coste temporal mencionado, $O(m+n)$. El coste espacial es consecuencia del almacenamiento de la subsecuencia generada.

4. Herramienta desarrollada

En este capítulo se hará una descripción exhaustiva de la aplicación desarrollada como objetivo central de este trabajo.

Se comenzará explicando algunos de los detalles y decisiones de diseño de la implementación realizada. A continuación, se irán tratando una a una las funciones que ofrece el programa, así como la interfaz de usuario que se provee.

4.1. Detalles de la implementación

La aplicación ha sido desarrollada utilizando el lenguaje de programación *Java*, junto con el conjunto de librerías gráficas *JavaFX*. Una de las grandes ventajas de este lenguaje es que permite la creación de programas ejecutables en cualquier sistema operativo. Además, una aplicación *Java* se puede transformar, con unas pocas modificaciones, en un *Java Applet* capaz de ser embebido en una página web, lo cual es uno de los posibles usos que se le darán a la herramienta.

4.1.1. *Maven*

Desde el comienzo del desarrollo, se utilizó *Maven* como soporte. *Apache Maven* es una herramienta para la automatización de los procesos de compilación, ejecución de *tests* y mantenimiento de un proyecto [6].

Uno de los objetivos principales de *Maven* es «proveer un sistema de construcción uniforme». Esto se consigue imponiendo una estructura de ficheros común a todos los proyectos construidos utilizando esta herramienta, la cual se puede ver en la figura 4.1.

Maven facilita, a su vez, la ejecución de *tests*. El uso de *tests* es una práctica fundamental para el desarrollo de código de calidad, por lo que también es algo en lo que se puso énfasis desde un principio.

Concretamente, los *tests* fueron escritos usando la librería para pruebas unitarias *JUnit*.

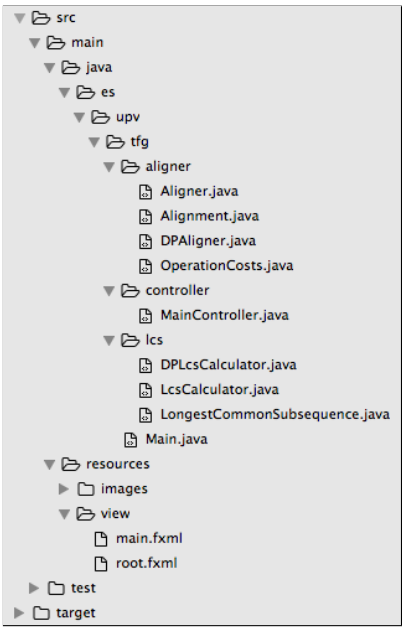


Figura 4.1: Estructura de ficheros del proyecto

4.1.2. *JavaFX*

Como se mencionó anteriormente, para el desarrollo de la interfaz gráfica se hizo uso del conjunto de librerías *JavaFX* [7].

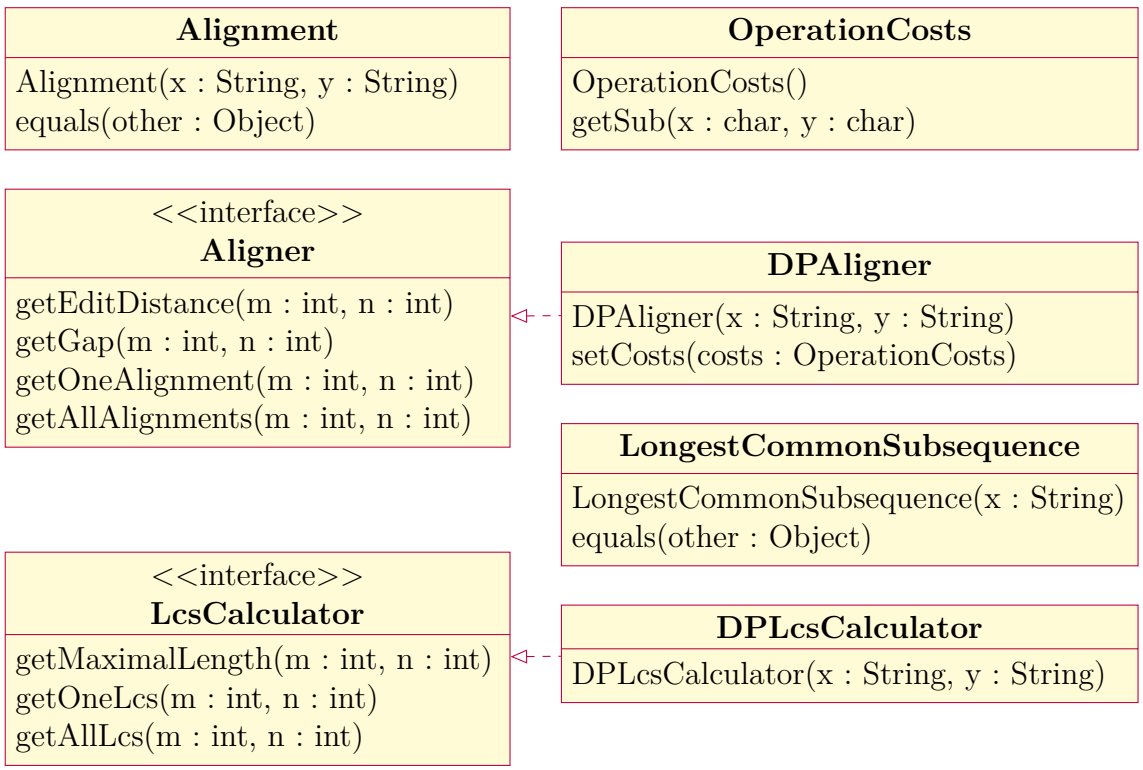
En la práctica, esto significa desarrollar la aplicación entorno a los conceptos de *modelo*, *vista* y *controlador*, lo cual conforma el patrón arquitectónico *MVC*. Los *modelos* administran los datos y la lógica del programa, mientras que las *vistas* se encargan de representar de forma gráfica la información de salida. Por último, los *controladores* funcionan de enlace entre los otros dos elementos.

Dado que la interfaz gráfica se basa en una única ventana, como se verá en la siguiente sección, se puede observar en la figura 4.1 que se tiene un único *controlador*. En él se encuentran los métodos encargados de responder a las acciones del usuario, haciendo uso de las dos *vistas* que se tienen en el directorio `resources` ▸ `views`. Todos los ficheros listados en los directorios `aligner` y `lcs` componen nuestros *modelos*.

4.1.3. Diagrama de clases

Adentrándonos en el diseño de los *modelos*, observamos la utilización de otro patrón arquitectónico: la *programación basada en interfaces*. Este patrón consiste en la abstracción de los servicios que provee una clase, en la forma de una interfaz con la que interactuaremos.

De esta forma, se consigue separar la firma de una clase de su implementación, permitiendo al cliente de una clase intercambiar distintas implementaciones y, por tanto, incrementando la elasticidad y la mantenibilidad del sistema.



Por razones de espacio, en el diagrama de clases se han omitido algunos métodos poco relevantes, como los *getters* de `OperationCosts`.

Como se puede observar, se tienen dos interfaces centrales: `Aligner` y `LcsCalculator`. Estas interfaces definen los métodos que utilizaremos para realizar los cálculos en nuestra aplicación. Además, se proveen implementaciones para cada una de ellas: `DPAligner` y `DPLcsCalculator`. El prefijo `DP` indica que dichas implementaciones utilizan programación dinámica, siguiendo los algoritmos descritos en el capítulo anterior.

Lo interesante de este enfoque es que nos permite añadir otra implementación que utilice algoritmos distintos o esté paralelizada para sacar provecho de un *cluster*, por ejemplo.

4.1.4. Idioma utilizado

Tanto el código como la interfaz de usuario están en inglés. Existen varias razones para ello.

Para empezar, el inglés es el idioma por defecto en nuestro campo. Si se desea maximizar el uso potencial que se le dé a una aplicación, esta debe ser escrita en inglés. Así mismo, la literatura referente a los conceptos utilizados parece coincidir en la utilización de la terminología inglesa.

Por último, siendo de menor importancia, es una preferencia personal del autor.

4.2. Interfaz de usuario

Uno de los aspectos que se consideraron más importantes en el desarrollo de la aplicación fue la interfaz de usuario. Una buena herramienta debe ser intuitiva, permitiendo al usuario centrarse en la tarea que desea realizar. Esto es especialmente importante cuando el perfil del usuario objetivo no es necesariamente técnico.

También se consideró importante que todos los elementos necesarios para la ejecución del análisis fueran fácilmente accesibles y estuvieran siempre a la vista.

Siguiendo estas premisas, se decidió implementar una interfaz gráfica limpia, basada en una única ventana. En la figura 4.2, se puede ver una captura de esta ventana, en la que se han anotado seis puntos de interés.

El punto 1 marca la barra de menús. Nos permitirá acceder a ciertas funciones auxiliares, como las acciones «deshacer» y «rehacer», y la ventana de «ayuda». Los puntos 2 y 3 señalan las cajas de texto en las que se introducirán las palabras que queremos analizar. El punto 4 indica el botón que pulsaremos para realizar el análisis de las palabras introducidas, cuyo resultado se mostrará en la tabla señalada por el punto 5.

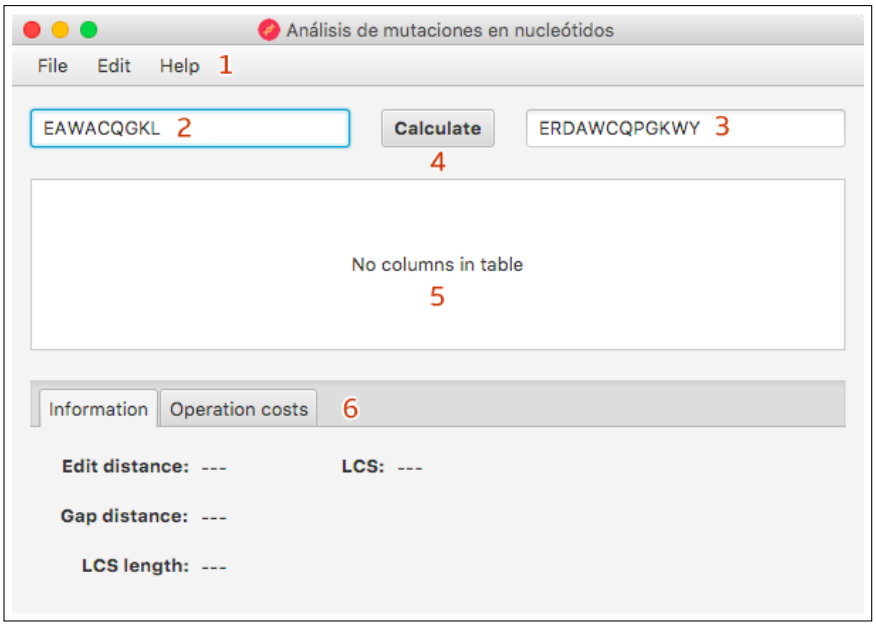


Figura 4.2: Ventana inicial de la aplicación

Finalmente, el punto 6 resalta el panel de pestañas. La primera pestaña, activada en la captura de pantalla, muestra información resultante del análisis realizado. La segunda pestaña permite al usuario modificar los costes de operación utilizados en el análisis, como veremos a continuación.

4.2.1. Costes de operación

Veamos ahora el primer ejemplo de la ejecución del análisis con parámetros $x =$ `EAWACQGKL` e $y =$ `ERDAWCQPGKWY`, como se vio en la sección 3.1.1.

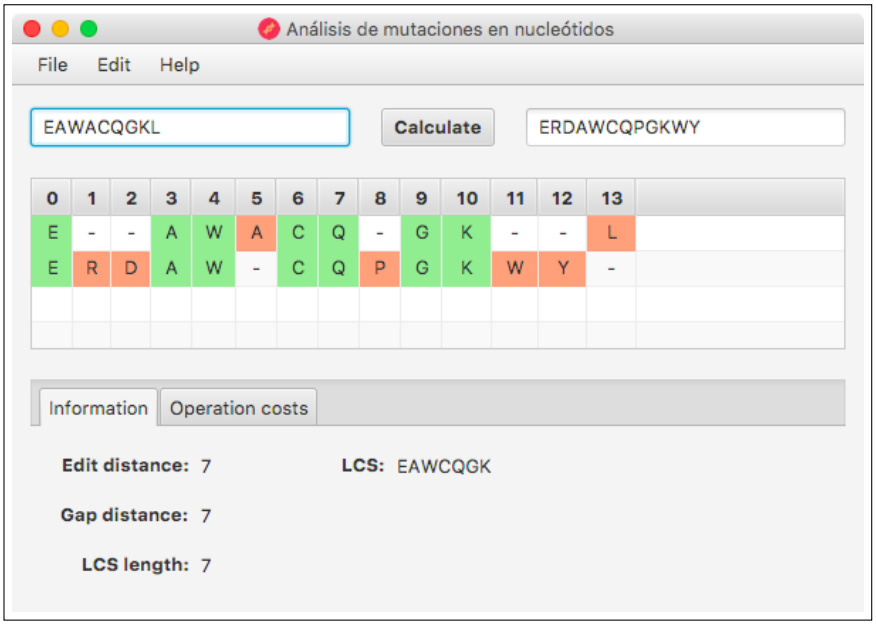


Figura 4.3: Ejemplo de la ejecución del análisis

Como se puede observar en la figura 4.3, tras pulsar el botón «Calculate», se rellenan los distintos elementos de la aplicación con el resultado. En la tabla central se puede ver un alineamiento. Para facilitar al usuario su lectura, las posiciones en las que se cumple $x[i] = y[i]$ se marcan en verde, mientras que las demás se marcan en rojo.

En la pestaña de información se provee la distancia de edición, la distancia de edición con penalización de huecos, la distancia de subsecuencia y una subsecuencia común maximal.

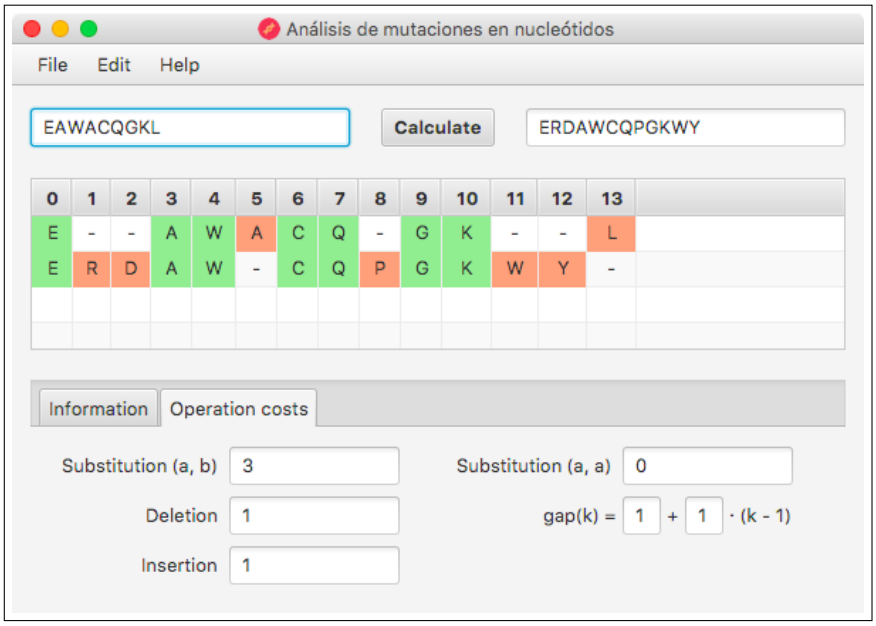


Figura 4.4: Costes de operación utilizados en el ejemplo anterior

Como se comentó en el capítulo de teoría, los resultados obtenidos dependen de los costes de operación configurados. La segunda pestaña del panel inferior, apreciable en la figura 4.4, nos permite modificar estos valores.

Nótese que la función de sustitución está separada en dos casos: sustitución de una letra por otra letra distinta y sustitución de una letra por sí misma. Además, podremos asignar valores a los parámetros g y h, respectivamente, de la función $gap(k)$, lo que se tratará con más profundidad a continuación.

4.2.2. Penalización de huecos

La aplicación permite al usuario penalizar la creación y extensión de huecos en un alineamiento. Para ello, solo hará falta modificar, en la pestaña de costes de operación, la función $gap(k)$ utilizada en el análisis.

En la figura 4.5 se puede ver una captura de la herramienta tras la ejecución del análisis con parámetros $x =$ `EAWACQGKL` e $y =$ `ERDAWCQPGKWY`, habiendo asignado costes distintos a los de los ejemplos anteriores.

Tal y como se vio en la sección 3.1.4, el parámetro g de la función $gap(k)$ penaliza la formación de nuevos huecos, mientras que el parámetro h penaliza la extensión de huecos existentes.

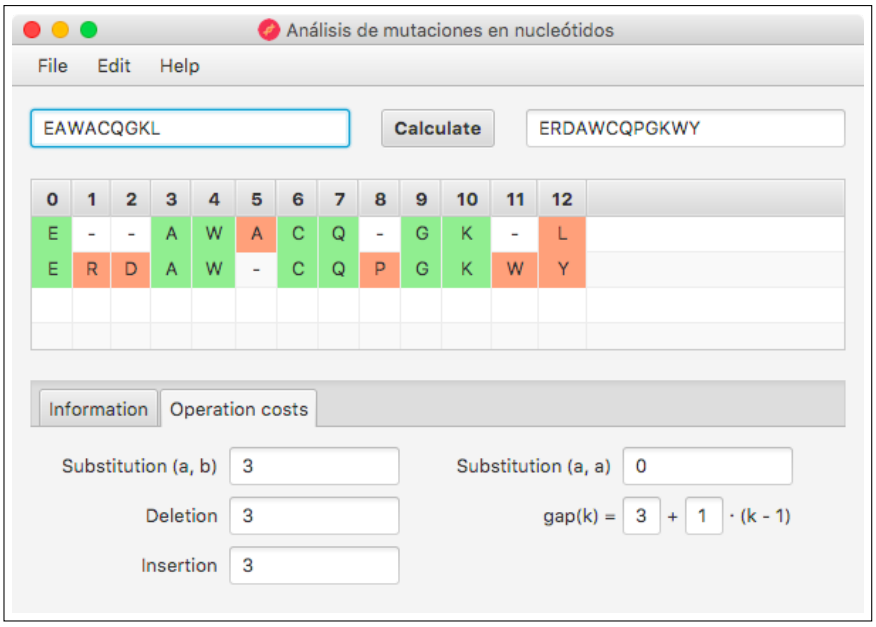


Figura 4.5: Nueva ejecución del ejemplo anterior penalizando la formación de huecos

Tiene sentido, por tanto, que al aumentar el valor del parámetro g en la figura 4.5 respecto a los vistos en la figura 4.4, el número de huecos se vea reducido en una unidad.

4.2.3. Modificación manual de un alineamiento

En la figura 4.6 se puede apreciar otro ejemplo de la ejecución del análisis con parámetros $x =$ `AGCTGA`, $y =$ `CAGATCAGAG`, visto en la sección 3.2.1.

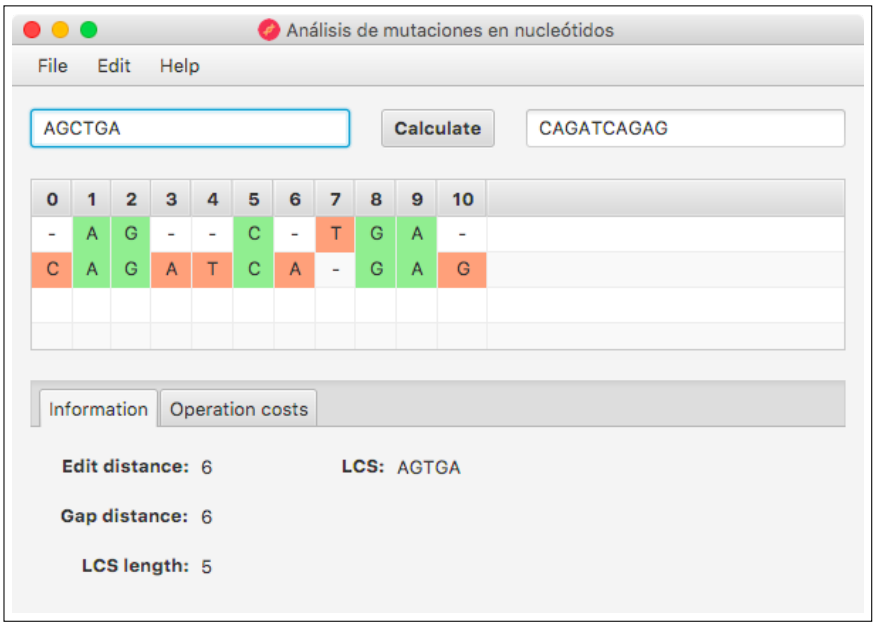


Figura 4.6: Ejemplo de la ejecución del análisis

En dicho ejemplo se ve el uso que se puede hacer de esta herramienta para calcular

alineamientos óptimos entre secuencias de bases nitrogenadas, lo que constituye el objetivo principal del proyecto.

Es habitual en el trabajo de un biólogo, sin embargo, la necesidad de modificar un alineamiento utilizando su conocimiento específico. Esto le permite intentar encontrar relaciones más distantes entre las secuencias que evidencien diferencias funcionales o evolutivas.

Esto se puede hacer en el programa siguiendo los siguientes pasos:

1. Seleccionar una letra de la primera secuencia.
2. Mantener presionada la tecla ⌘ (o ctrl en GNU/Linux y Windows).
3. Pulsar sobre una letra de la segunda secuencia.

En la figura 4.7 se ha modificado el alineamiento producido en la figura 4.6, alineando los símbolos `T` de las dos secuencias.

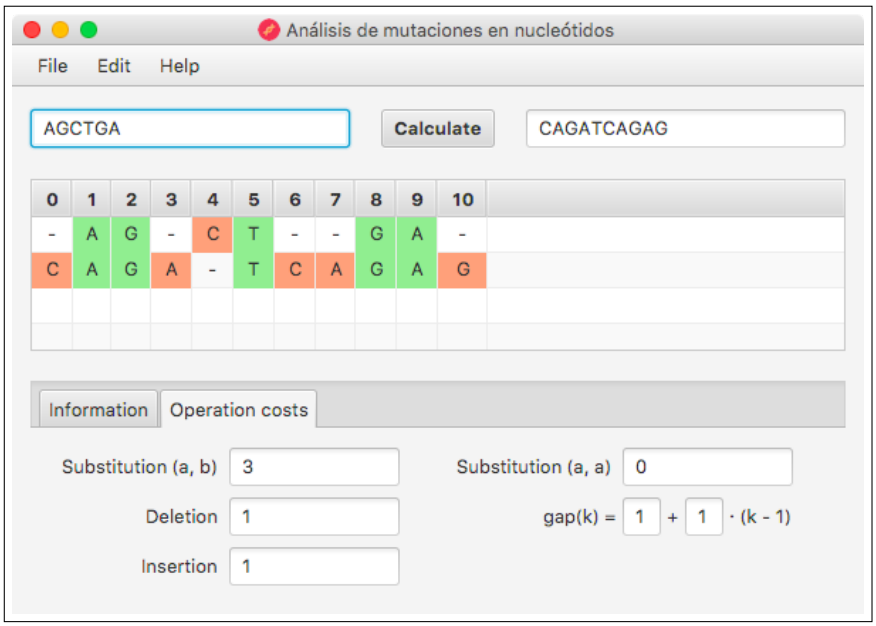


Figura 4.7: Nueva ejecución del ejemplo anterior modificando el alineamiento

Simplemente se parten las secuencias en las posiciones seleccionadas, calculando los alineamientos óptimos de los dos trozos resultantes. Finalmente, se concatenan y se muestran al usuario.

5. Conclusiones

En este último capítulo se hará un breve análisis del trabajo realizado, valorando el cumplimiento de los objetivos del proyecto. Además, se ofrecerá una lista de posibles mejoras y trabajos futuros que se consideran de interés.

5.1. Análisis del trabajo realizado

Llegado este punto, tiene sentido plantearse si el trabajo realizado cumple los objetivos del proyecto descritos en la introducción, satisfaciendo su motivación inicial: «desarrollar un analizador de biosecuencias que permita encontrar patrones y resaltar diferencias entre cadenas de ADN y ARN».

La herramienta desarrollada permite al usuario introducir dos secuencias para ser analizadas utilizando los costes de operación configurados, dando como resultado sus distancias de edición con y sin penalización de huecos, así como la distancia de subsecuencia y una subsecuencia común maximal. También se provee una forma práctica de visualizar sus alineamientos óptimos, permitiendo introducir modificaciones al alineamiento producido.

Se cumple, por tanto, el objetivo principal de «brindar al usuario la capacidad de alinear dos cadenas de forma óptima».

5.2. Posibles mejoras y trabajos futuros

En el resumen se hace referencia a una evaluación del proyecto para una futura integración en un *website* para su ejecución en red. Si bien es una decisión que no depende directamente del autor, se tuvo en cuenta esta posibilidad en el proceso de diseño de la aplicación para facilitar dicho uso.

Otra posibilidad que se dejó abierta durante el desarrollo, tal y como se comentó en la sección 4.1.3, es la paralelización de los algoritmos utilizados. Los microprocesadores modernos proveen al programador múltiples núcleos en los que ejecutar sus programas. Dado que la velocidad de ejecución de una aplicación tiene un gran impacto sobre su utilidad, cualquier mejora en rendimiento que se consiga en una nueva implementación de las clases `Aligner` y `LcsCalculator` será bienvenida.

A pesar de ser una herramienta útil que cumple los objetivos planteados al inicio del proyecto, existe una gran diferencia en las capacidades de la aplicación desarrollada

en comparación con los programas que se utilizan habitualmente en el campo. Así pues, existen diversas funciones que se podrían añadir en futuras expansiones.

Una de las más interesantes es la capacidad de producir alineamientos entre múltiples secuencias de ADN. Este es un problema mucho más complejo que la producción de alineamientos entre dos cadenas, pero es también más útil.

Bibliografía

[1] Maxime Crochemore, Christophe Hancart y Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[2] Gonzalo Navarro y Mathieu Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[3] John E. Hopcroft, Rajeev Motwani y Jeffrey D. Ullman. *Teoría de autómatas, lenguajes y computación*. Pearson Educación S.A., 2007.

[4] David W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2004.

[5] Wikipefia. *Sequence Alignment*. URL: `https://en.wikipedia.org/wiki/Sequence_alignment`.

[6] The Apache Software Foundation. *Maven - Introduction*. URL: `https://maven.apache.org/what-is-maven.html`.

[7] Oracle. *JavaFX Overview*. URL: `http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784`.