Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de un analizador de secuencias de ADN obtenidas por recombinación

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Jose Bonora Soriano

Tutor: Jose Maria Sempere Luna

2014 - 2015

Resumen

El objetivo de este trabajo es el diseño e implementación de una aplicación capaz de reconocer y dar solución al problema de recombinación de las cadenas de ADN, es decir, establecer si a partir de unas cadenas madre de ADN, o partes de estas, sería posible formar otra cadena dada. Y en caso de que la solución sea afirmativa, formular cómo ha sido dada. Se implementara usando un autómata finito no determinista y técnicas de programación dinámica. Todo esto, en un entorno gráfico diseñado para facilitar al usuario su uso y compatible con el formato FASTA y formatos estándar.

Palabras clave: ADN, recombinación genética, FASTA, AFN, autómata.

Abstract

The aim of this work is the design and implementation of a software which is able to recognize and solve a problem of recombination of DNA molecule, in other words, establish if with a mother DNA molecules would be possible to form a new DNA molecule. And if the solution is affirmative, return the path of that recombination. It is implemented using a nondeterministic finite automaton and dynamic programming techniques. All this with a graphical environment designed to facilitate the user their use and compatible with the FASTA format and standard format.

Keywords: DNA, genetic recombination, FASTA, NFA, automaton.

Tabla de contenidos

1. Introducción

Este trabajo partió de la idea de diseñar e implementar una aplicación que fuera capaz de reconocer cadenas del material genético ácido desoxirribonucleico (ADN) creadas a partir de una recombinación de otras cadenas de ADN. Ese fue el objetivo inicial, aunque durante el desarrollo del trabajo se planteó no solo reconocer una posible recombinación, sino que además debería dar, en caso de que existiese, la solución de cómo ha sido formada. Además habría que implementar una interfaz gráfica en JAVA para poder ser utilizado en cualquier máquina que soporte esta plataforma software.

Primero hemos de centrarnos en el problema de recombinación ADN, ¿qué es?, ¿en que se basa? y ¿cómo podemos tratar el problema a nivel computacional? Para ello se explicará fundamentos de biología relacionados con el ADN, cómo se ha tratado en artículos científicos y formas de representar una cadena de ADN. Una vez solucionado esto, el siguiente paso es mostrar las herramientas y estrategias de programación que se han usado para poder crear el programa que cumpla los objetivos propuestos; haciendo hincapié en la explicación de autómatas finitos no deterministas (AFN) y en estrategias de programación dinámica. Terminado este paso se expondrán los motivos de la estructura de la interfaz visual, ya que esta viene definida por la necesidad de que pueda ser integrada en otro programa mayor, y el funcionamiento para que un usuario medio sea capaz de utilizarlo. Para finalizar se expondrá las conclusiones del funcionamiento de la aplicación basándose en sus características y resultados, dejando paso a posibles mejoras o ampliaciones de este.

2. Recombinación del ADN

Para podernos plantear el problema de recombinación del ADN, y en concreto un tipo de este, debemos tener en cuenta conceptos previos para encabezar dicho problema. Empezando por la definición del mismo ADN, toda la información relevante a este apartado ha sido sacada de [1] y [2].

El ADN es una molécula con estructura de doble cadena helicoidal formada por ácidos nucleícos. Esta molécula es la encargada de almacenar la información genética de los seres vivos dependiendo de la secuencia de nucleótidos que contenga. Estos nucleótidos están formados por un monosacárido, en este caso una pentosa la desoxirribosa, una base nitrogenada y un ácido fosfórico. Dicha base nitrogenada podrá ser adenina (A), timina (T), citosina (C) o guanina (G), según cual sea la base nitrogenada que contenga el nucleótido se codificara en la secuencia de nucleótidos dicho nucleótido. Los nucleótidos de una misma hebra están unidos por enlaces fosfodiéster creados a partir del ácido fosfórico, es decir un enlace del tipo covalente y por lo tanto crean una fuerte unión. Las dos hebras se mantienen unidas gracias a los puentes de hidrogeno entre las bases nitrogenadas de los nucleótidos. Dichos enlaces son más débiles y por lo tanto el ADN obtiene la capacidad de poder desenrollar las dos hebras en situaciones que lo requieran. Otra característica que ofrece esta estructura es que, para que se creen los puentes de hidrogeno, deben de colocarse las dos hebras en direcciones opuestas. Teniendo como referencia la posición que ocupan los carbonos en la pentosa podemos decir que una hebra sigue la dirección 5’ - 3’ y la otra 3’ - 5’. Además las dos hebras de la cadena son complementarias en cuanto a su codificación, ya que la base nitrogenada sólo creará el enlace con su complementaria, en el caso del ADN, A es complementaria con T y C con G, por esto el ADN es una cadena antiparalela.

El proceso de recombinación genética es aquel por el cual una molécula de material genético es cortada para que se adhiera a esta otra molécula con un material genético diferente al de la molécula original. En concreto se ha trabajado con ADN cuya finalidad a nivel biológico es una fuente de variación y selección en el material genético.

Existen dos tipos de recombinación, la recombinación homóloga y la recombinación específica del sitio. La primera se produce entre dos regiones del ADN las cuales tienen secuencias homólogas; normalmente son secuencias grandes, el método que se aplica en este tipo de recombinación produce una ruptura o sinapsis en la cual se crea un entrecruzamiento de las dos regiones, el resultado de este proceso es un intercambio de material genético. En la segunda, la que tiene más relevancia en este trabajo, actúan elementos genéticos móviles. Estos elementos tienen una zona de unión compartida con la cadena a la que se van a juntar, dicha zona se suele llamar diana y suele ser una zona de solo unos pares de bases, pero el material genético transportado no tiene porque ser homologo; de ahí la diferencia con la primera.

Mirando en profundidad la recombinación específica de sitio, nos encontramos con la capacidad de transportar una secuencia de una cadena de ADN a otra cadena siempre y cuando haya esa zona diana. Aunque para realizar este proceso necesitamos

unas enzimas llamadas transportasas, las cuales cortaran los extremos del elemento transponible para colocarlos en el ADN con la zona diana. Aunque en verdad la secuencia transponible nunca queda libre, ya que durante el proceso en el que actúan las enzimas siempre estará ligada a alguna de las dos cadenas. Destacar que, como se trata de ADN, todas las secuencias transportadas tendrán su inversa que se transportará también. Este tipo de recombinación en biología molecular es la que hace referencia a la recombinación artificial de ADN creando el ADN recombinante.

El ADN se puede encontrar en diferentes estados según la fase en la que se encuentre la célula, además de tener cuatro estructuras dependiendo de esos estados. La estructura primaria es la más simple pues es solo la secuencia de nucleótidos de la cadena y es la usada en este trabajo, ya que a la hora de representarla computacionalmente podemos hacerlo como una secuencia de caracteres A, T, C y G; representando cada carácter a la codificación de los nucleótidos como en la estructura primaria.

Además, para tratar con dichas cadenas de ADN utilizaremos un formato extendido en el ámbito de la bioinformática como es el formato FASTA, ya que es compatible con la codificación expresada en el párrafo anterior y dispone de grandes bases de datos de secuencias de nucleótidos, como por ejemplo GenBank.

El formato FASTA es un formato de fichero en texto plano, el cual es utilizado para la representación de cadenas de ácidos nucleídos, es decir no solo ADN pues otras moléculas como el ácido ribonucleico (ARN) están formadas por nucleótidos. Este formato también permite incluir nombres de secuencias y comentarios, usuales en bases de datos como en GenBank.

En un fichero FASTA se puede introducir una o más cadenas (siempre encabezado por el carácter “>”), a continuación el nombre y comentarios si los tuviese en la misma línea; y en la siguiente línea empezaría la secuencia codificada de nucleótidos, siendo normal como máximo 80 caracteres por línea. Un ejemplo de un fichero en formato FASTA sería el de la figura 1.

```
>gi|372098992|ref|NT_039500.8| Mus musculus strain C57BL/6J chromosome 10 genomic contig,
GRCm38.p3 C57BL/6J MMCHR10_CTG5_2
AAGTTTTTCAAACTCGATCACTCATGGTCTCTTGAGAAACTTGGAGATGTGATGTAGGTCTCTGTGTGAA
CCGTTTTTCTGCTGGTTCTGTCTCTCATCTCTCATGGATGTCATAAACATCCTCTGCTACTAGTTGTTGG
GAACTTATAAAAGGGGGTACTGGGCAAAGGGGGAGGGAGACCCACACCCCACCAGAGTTTTACCTATCCT
CTGGTCTGTCAGGAGAGAGAGAAAGAGAGAGAGAGGGAGAGAGCTATCATACACTGTCCATTCATTCCTG
GGTGGGCATTCAAGCCTCTGACCCCCTCTTCAGGGGATGGCCAAGGAGCCATGGGATCCCTGGACCTACT
TTGCTAAAGCCACCAGCGTATAAGAATGAGGGATGAGGGAAGAGGTTCCCAACACCTGTGAGTGTGTGTG
CAGTGGATCTTGACAAGCAGAGACTCTCTATGAGAGAGTGTCAATGAGGTTTTAGAGCTTTACTATAGAA
TGCAGAGGGAAAGAGAGAAGGTAGAAAGAGAGTGACAGAGACAGAGAGAAGGCTAGAGAGAAAGAGATTA
AGAGGGCAAGAGGAGAGGAGAGGAGAGGAGAGGAGAGGAGAGGAGAGGAGAGGAGAGGAGAGGAG
```

Figura 1

3. Autómatas finitos y algoritmos utilizados

3.1. Conceptos sobre autómatas

Ahora que se ha establecido la descripción del problema y se tienen han proporcionado los conceptos básicos, se explicará con que herramientas conceptuales se ha llevado a cabo el diseño y la implementación de la aplicación.

La idea principal de la que se partió fue del análisis de un trabajo previo [4] en el cual se propone crear un AFN, a partir de las cadenas madre de ADN y sus patrones de corte, como un autómata capaz de reconocer cualquier palabra del lenguaje creada por la combinación de los segmentos entre patrones de las cadenas madre.

Una vez creado dicho autómata, podremos darle como entrada la cadena que queremos saber si puede ser formada o no. Es decir, si el autómata reconoce esa cadena como una palabra del lenguaje descrito por el autómata querrá decir que es recombinante, y si no lo reconoce no puede ser formada. Además si se reconociese como una palabra del lenguaje deberíamos devolver las soluciones de cómo podría formarse, en otras palabras, qué segmentos de cada cadena madre han intervenido y en qué orden. Para ello deberemos guardar el camino tomado por el autómata y luego tomar las soluciones posibles.

Para comprender el funcionamiento empezaremos explicando que es y cómo funciona un AFN. Un autómata finito no determinista (A, Q, δ, q_0, F) es una máquina abstracta que recibe una entrada discreta para producir mediante cómputos una salida de aceptación o negación. Está compuesto por un alfabeto A, un conjunto de estados finito Q, una función de transición δ, un estado inicial q_0 y un conjunto de estados finales F. Un rasgo característico de este tipo de autómata es que puede tener para un mismo estado diferentes transiciones para una misma entrada. Además la entrada del autómata tiene que pertenecer a A.

El funcionamiento de este se basa en que cuando reciba una entrada según esta y el estado en el que se encuentra actualmente la máquina, siendo el estado inicial el marcado por q_0, transitara a otro estado de la propia máquina; esta transición solo será efectiva si se encuentra la función de transición. Una vez se haya leído por completo la entrada, si la máquina se encuentra en un estado final querrá decir que esa entrada es aceptada por el autómata.

Un ejemplo de AFN que reconoce palabras de un lenguaje, por ejemplo un lenguaje que esté formado por palabras compuestas solo por el carácter a, sería el siguiente:

A = {a,b}, Q = { q_1, q_2}, q_0 ={ q_1 } y F = {q_0}. Mientras que la función de transición vendría dada por la tabla de la figura 2. El autómata puede ser representado gráficamente por un diagrama de estados. Donde los estados son círculos con el

nombre asociado a dicho estado, las flechas indican la transición entre estados con la entrada que la produce, el estado inicial viene dado por el circulo que tiene la transición que no tiene un estado como salida y los estados finales son aquellos que contienen un circulo en el interior del estado. La representación del autómata anterior vendría dada por la figura 3.

	a	b
q_1	q_1	q_2
q_2	-	-

Figura 2

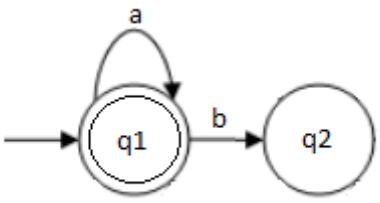

Figura 3

El lenguaje que queremos reconocer con dicho autómata es un lenguaje de tipo *splicing*, este tipo de lenguajes son definidos por un *splicing system*. Estos sistemas fueron introducidos por Tom Head, en 1987, como un modelo matemático capaz de simular el funcionamiento de recombinación del ADN. Como hemos abstraído de [3] y [4].

Un *splicing system* (A, I, B, C) está formado por un alfabeto A, un conjunto de cadenas iniciales I (las cadenas madre), y dos conjuntos finitos B y C de 3-tuplas de la forma (c, x, d). Estas 3-tuplas se llaman patrones izquierdos los de B y patrones derechos los de C. El lenguaje generado por este sistema es un lenguaje que contiene todas las cadenas de I y todas las que se puedan obtener por adición de palabras ucxfq y pexdv, siendo (c, x, d) y (e, x, f) patrones del mismo lado. Estando c, x y d contenidos en A*; siendo * una operación unaria que se aplica sobre un conjunto de símbolos o caracteres en los cuales estará incluida la cadena vacía (λ), y representa el conjunto de las cadenas que se pueden formar tomando cualquier número de cadenas del conjunto inicial, posiblemente con repeticiones, y concatenándolas entre sí.

El tipo de *splicing system* en el nos basamos es el *simple splicing system* (SH), que está compuesto por (A, I, R), siendo A el alfabeto, I las cadenas iniciales y R las reglas del sistema. Pero añadimos un nuevo concepto: los *maximal firm subwords*. Decimos que una palabra es *maximal firm subword* cuando no contiene ninguna ocurrencia de las reglas R. A continuación, mostramos ciertos ejemplos de autómatas formados por este tipo de sistemas.

En este ejemplo vemos que A = {a,b}, I = {aabaaaaabaaabaa} y R = {b}. Los *maximal firm subwords* en este caso serian {aa, aaaaa, aaa}. Podemos ver la representación del autómata en la figura 4.

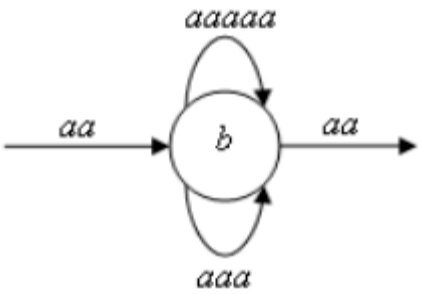

Figura 4

Otro ejemplo un poco más complejo sería A = {a,b}, I = {abaacaaaaaba, abaaabaaacaaba} y R = {b, c}. Los *maximal firm subwords* en este caso serian {a, aa, aaaaa, aaa}. Podemos ver la representación del autómata en la figura 5.

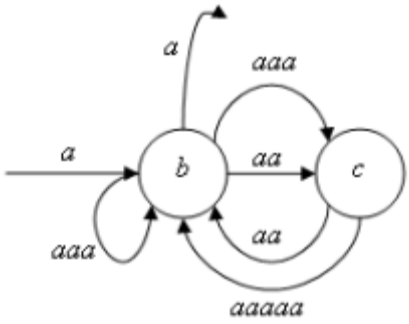

Figura 5

3.2. Algoritmo de construcción del autómata

El algoritmo encargado de crear el AFN tratara de obtener cada uno de los componentes del AFN, donde el alfabeto A = {A, T, C, G}, Q serán los estados de la máquina y se corresponderá a un estado por cada patrón que tengamos, otro para el inicial (ini) y un último (fin) sumidero de todos los finales de cadena, q_0 = {ini}, F = {fin} y por último δ vendrá dado por los *maximal firm subword* de las cadenas de I y los patrones de los extremos a cada *maximal firm subword*. Siendo el estado actual el patrón izquierdo, la entrada el *maximal firm subword,* y el estado al que transita el patrón derecho. Por lo tanto nuestro algoritmo deberá obtener dicha información antes de crearlo.

Para obtener los *maximal firm subwords* de las cadena de I, localizaremos los patrones en las cadenas de I e iremos obteniendo las subcadenas entre los patrones. Aunque deberemos tratar casos especiales en los cuales los patrones están embebidos en otros o se solapan, estos casos el algoritmo funciona de diferente forma y se explicaran con detalle más adelante.

Lo primero que hará el algoritmo será crear un array llamado R2 de listas con la estructura S2, que se puede observar en el anexo. El array tendrá el tamaño del número de patrones de corte más uno, ya que tomaremos como patrón también el inicio de la cadena. Cada lista hará referencia a un patrón y contendrán los índices de donde se encuentran dichos patrones en la cadena que se analice de I. Una vez tengamos localizados donde están, nos dispondremos a sacar los *maximal firm subwords,* según en qué caso nos encontremos.

La parte del algoritmo que calcula los *maximal firm subwords* se basa en tener dos arrays, min_1 y min_2, con las posiciones mínimas de cada patrón en min_1 y en min_2 la posición siguiente si lo hubiera.

Caso 1: El caso simple sería para una cadena de I = {TACCCAGTA} y R = {TA, AG} en min_1 contendría para TA el valor 0 pues esta al principio y para AG el valor 5, ahora en min_2 deberemos poner los mismos valores salvo los del valor más pequeño de min_1, en este caso los de TA; quedando el valor 7 para TA y el valor 5 para AG. Ahora simplemente sabemos que, desde el mínimo valor de min_1 más la longitud del patrón hasta el mínimo valor de min_2, es un *maximal firm subwords*; es decir, desde 2 hasta 5 dando como resultado CCC. Y además sabemos las zonas diana, siendo la izquierda TA y la derecha AG. Una vez hecho esto, seguiremos iterando el mismo proceso recalculando los vectores min_1 y min_2 con los siguientes valores de R2, si los hubiera, hasta finalizar la cadena I.

Caso 2: Este sería el caso en cual la cadena no empieza por un patrón. En este caso se añade un nuevo patrón a los vectores min_1 y min_2 llamado ini este tiene por defecto de R2 los valores 0 y el tamaño de cadena para que, en caso de que no exista ningún patrón en I el *maximal firm subword* sea la cadena I entera, y en caso de que si lo haya, actuara como en el caso simple.

Caso 3: Este sería el caso en el cual el patrón izquierdo tenga embebido o solapado otro patrón. Para una cadena de I = {TAGCCAGTAG} y R = {TAG, AG} nos encontramos con que en min_1 tenemos para TAG el valor 0 y para AG el valor 1, y en min_2 para TAG el valor 7 y para AG el valor 1. Por lo tanto si ejecutamos el caso simple nos daría que el *maximal firm subword* es CCAG (cosa errónea pues contiene un patrón). Para solucionar esto, a la hora de calcular los valores de min_2, no solo actualizaremos el del mínimo de min_1 sino que también aquellos que tengan un valor menor al mínimo de min_1 más la longitud del patrón de ese mínimo, en este caso 0 más 3; por lo tanto actualizaríamos el valor del patrón de AG de min_2 a 5. Dando así como *maximal firm subword* CC con patrón izquierdo TAG y derecho AG.

Caso 4: Este sería el caso en el cual el patrón derecho tenga embebido o solapado otro patrón. Para una cadena de I = {AGCCTAG} y R = {TAG, AG} nos encontramos con que en min_1 tenemos para TAG el valor 4 y para AG el valor 0, y en min_2 para TAG el valor 4 y para AG el valor 5. Por lo tanto si ejecutamos el caso simple nos daría que el *maximal firm subword* es CC, pero no solo es ese pues CCT también debería serlo. Para solucionar esto no solo cogeremos desde el mínimo de min_1 hasta el mínimo de min_2, sino a todos aquellos de min_2 que sean menores que el que el mínimo de min_2 más la longitud del patrón de min_2. En este caso el min_2 de AG lo cumple, pues 5 es menor que 4 más 3. Dando así además como *maximal firm subword* CCT con patrón izquierdo AG y derecho AG.

Esta parte del algoritmo se ejecutará para cada cadena de I, y los *maximal firm subword* se almacenaran en un array con el mismo tamaño que R2 de listas enlazadas con la estructura S, que está disponible en el anexo. Cada posición de este arrary hará referencia a los patrones izquierdos y, según estos, se añadirán los *maximal firm subword* encontrados con su patrón derecho, también guardaremos de que cadena madre de I han sido encontrados.

Una vez calculada la función de transición y los demás componentes del AFN diremos que ya hemos creado nuestro AFN para una entrada (I, R). En la figura 6 podemos ver un ejemplo para el caso:

I = {CGATCCCAGATATTAF, TACCC} y R = {TA, GAT, GATA, AG}.

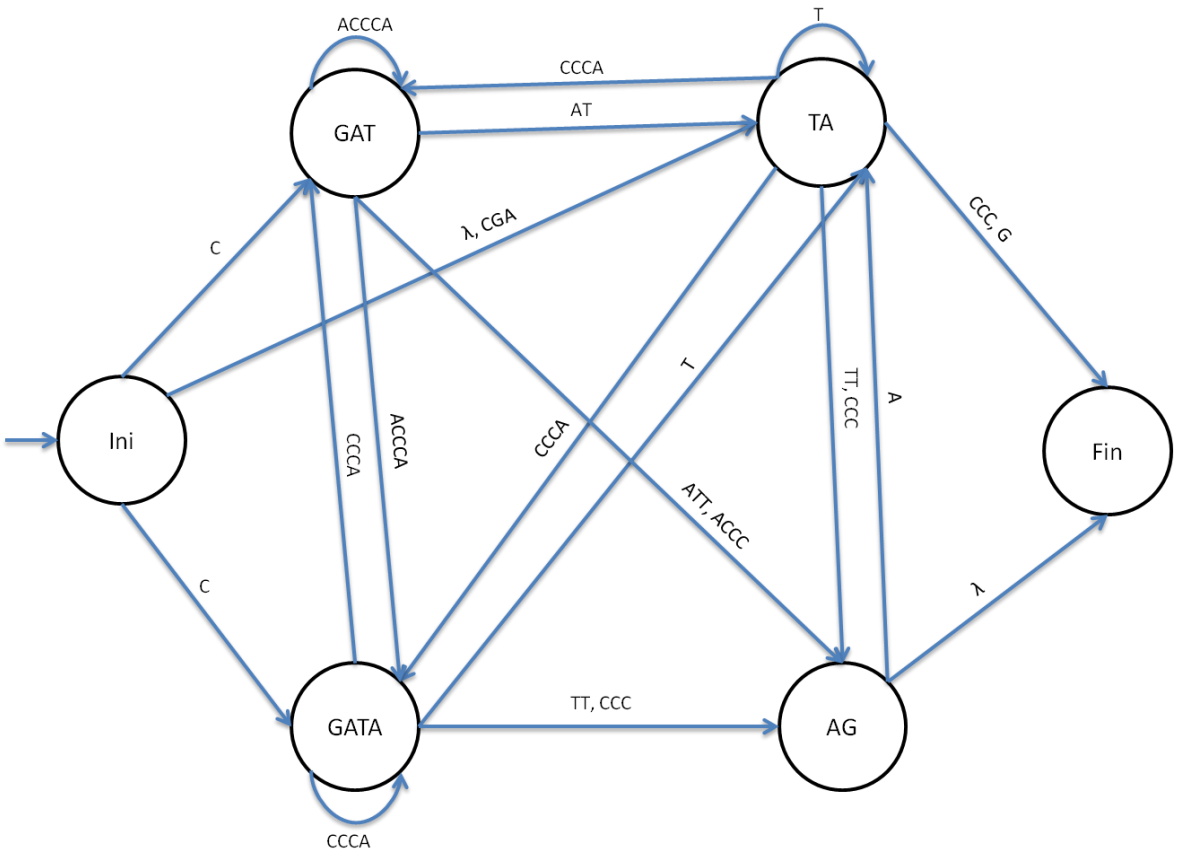

Figura 6

3.3. Análisis mediante el AFN

Creado el AFN ahora podremos pasarle cadenas de ADN para ver si estas pueden ser creadas por recombinación. Para saberlo calcularemos el resultado representándolo en un grafo multietapa. Para ello utilizaremos una matriz N*M, donde N son el número de estados del grafo y M la longitud de la cadena de entrada más uno. Los valores que añadiremos a esta matriz serán del tipo V2, que se puede consultar en el anexo. Se inicializara esta matriz en el estado inicial, y se recorrerá la matriz por

columnas viendo si existe o no un objeto V2. Si es así se intentara aplicar alguna de las transiciones que existen para el estado en el que se encuentre, si no hay entrada posible para ese estado simplemente esa rama del grafo terminara, si existe, aplicara la transición e irá a la posición de la matriz avanzando a la fila asignada al estado nuevo y avanzará x posiciones de columna; siendo x el valor actual más la longitud del *maximal firm subword* asociado a esa transición y del patrón derecho. En esa nueva posición se creará un nuevo objeto V2, el cual guardara la posición anterior de la matriz y de que cadena madre era esa transición, esto es así para poder devolver el camino que realiza el grafo como veremos más adelante. Además V2 es una lista enlazada para poder ver si diferentes caminos del grafo confluyen en uno mismo.

Recorrida toda la matriz, lo que determinará si se ha llegado a una solución es si en la fila que representa el estado Fin y la última columna existe algún objeto V2. Si es así, querrá decir que la recombinación es posible pues existe una combinación de estados del AFN para crear esa cadena. En caso de que no lo haya, simplemente el resultado será que no hay posibilidad de recombinación. Un ejemplo para el AFN creado en la figura 8 con entrada TACCCAGATAG sería el de la figura 7.

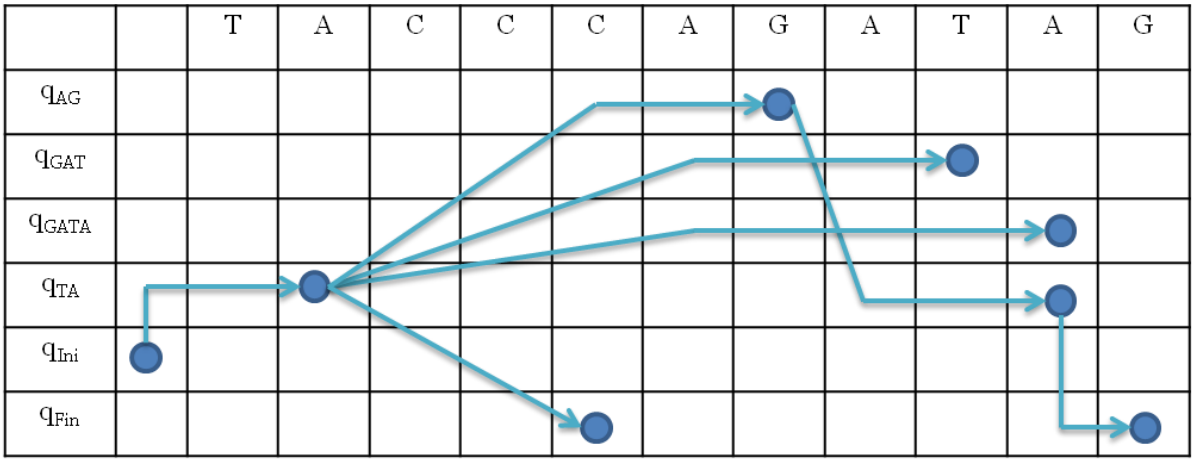

Figura 7

Si aplicamos esta estrategia de programación dinámica solo podemos devolver si existe o no solución, sin embargo al ir guardando en cada nodo del grafo la posición en la matriz del nodo anterior podemos recuperar el camino de la rama que ha generado una solución. Y no solo eso, además sabemos la procedencia de las partes de la cadena generada. Una característica de este grafo es que a un nodo de la solución se puede llegar de diferentes caminos, por lo tanto, cuando lo recorramos hacia atrás buscando las soluciones deberemos guardarnos la posición de la matriz donde se almacenen al menos más de un objeto V2 para volver a recorrer dicha rama del grafo y devolver esa solución también. Un ejemplo de esto se daría para la entrada del AFN de la figura 6 CGATATTAG como vemos en la figura 8, el cual tiene hasta 6 soluciones.

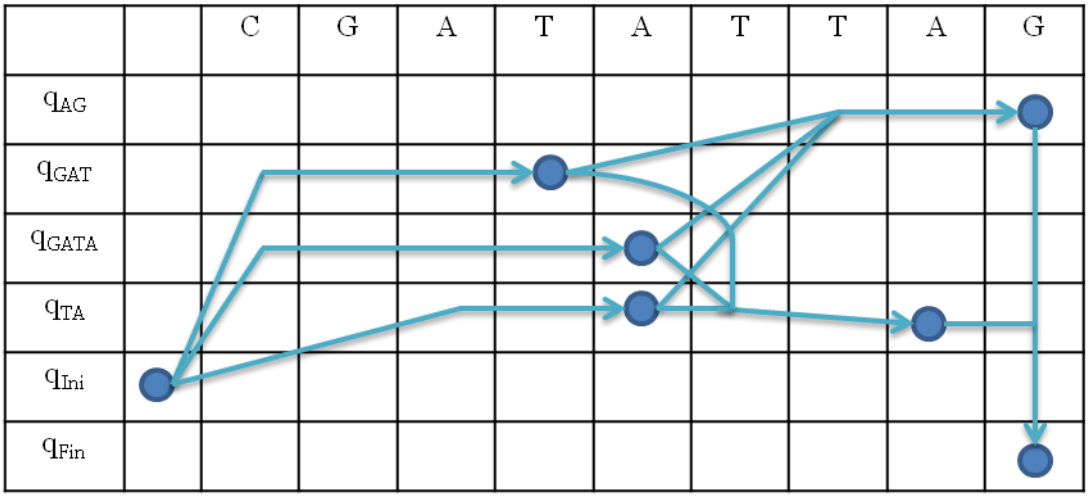

Figura 8

4. Interfaz de la aplicación

La interfaz de esta aplicación ha sido pensada para ser incluida dentro de un sistema mayor, por lo tanto ha sido diseñada pensando en esto. Nuestra interfaz dispondrá de una sola ventana, la cual estará visiblemente separada en tres zonas: la primera es la zona de adición de cadenas madre y patrones, la segunda es la zona donde se genera y visiona el AFN creado, y la tercera zona es la de visionado de la solución. Como se puede observar en la figura 9.

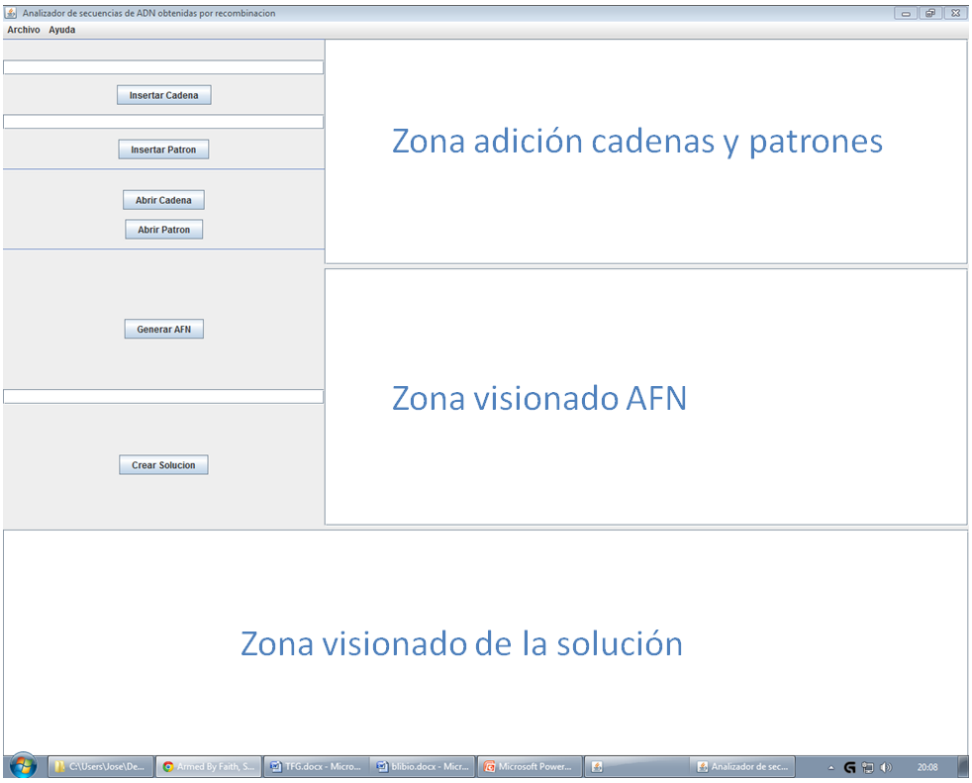

Figura 9

En la primera zona dispondremos de la capacidad de añadir cadenas madre y patrones tanto manualmente insertándolas en las líneas de texto y como pulsando los botones insertar cadena o insertar patrón, según lo que el usuario quiera. También dispondremos de otro método para añadirlas como selección de un fichero FASTA con los botones abrir cadena y abrir patrón, los cuales abrirán una ventana con el explorador del sistema, como el de la figura 10, para seleccionar el fichero. Dichos ficheros pueden contener una o más cadenas.

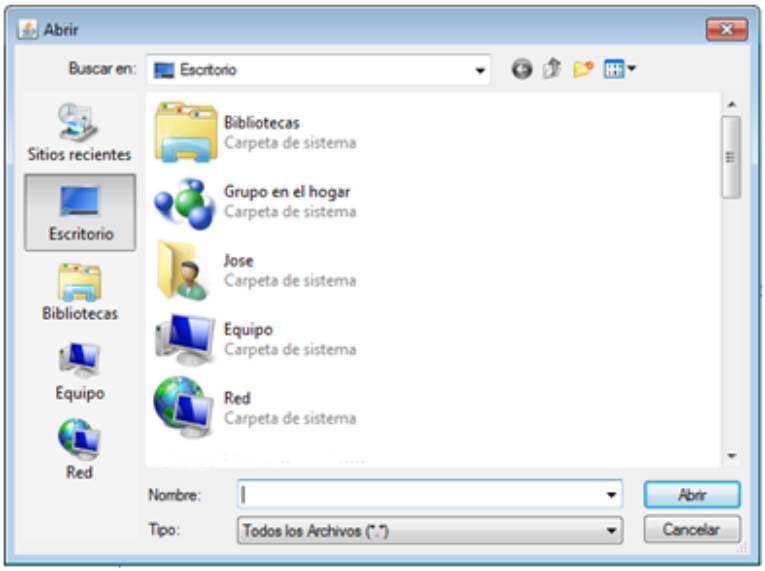

Figura 10

En la figura 11 podemos observar un ejemplo completo de cómo funciona el programa para el ejemplo I = { TACC, CGATACCCAGATATTAG }, R = {AG, TA, GATA, GAT} y entrada del AFN TACCCAGATAG.

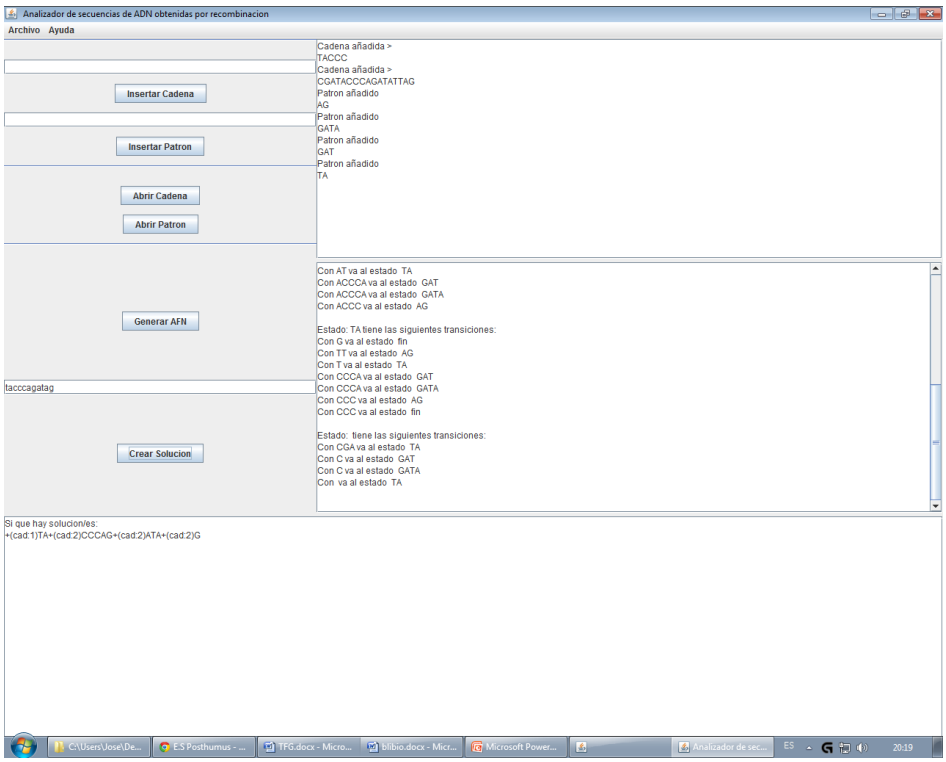

Figura 11

Como último punto destacar las opciones en el menú Archivo de borrar el contenido actual y guardar la solución actual en un fichero por defecto llamado “solución.txt”.

5. Conclusiones

En este trabajo se ha cumplido con el objetivo propuesto en la introducción, pues se ha diseñado e implementado una aplicación capaz de analizar cadenas de ADN obtenidas por recombinación y añadiendo cual es la solución a esa recombinación junto con una interfaz de usuario ergonómica y fácil de usar.

En cuanto a costes nos encontramos con que, en el algoritmo de creación del AFN, primero se recorren todas la cadenas de I buscando los patrones de R, es decir un coste de $\Theta(|I| * i * |R|)$; siendo |I| el número de cadenas madre, i la longitud de estas y |R| el número de patrones. Luego debemos encontrar para cada cadena de I los *maximal firm subwords* con coste $\Omega(i * |R|^2)$ y $O(|R|^2)$ para cada cadena de I. En la fase de análisis, para crear la matriz de programación dinámica debemos recorrerla por completo por columnas, por lo tanto será $\Theta(|Q|*M)$; siendo |Q| el número de estados y M la longitud de la cadena de entrada más uno. Finalmente tenemos que recuperar las soluciones con un coste $\Omega(S * M)$ y O(S), siendo S el número de soluciones obtenidas.

Aunque el proyecto ha dando como resultado una aplicación funcional y que cumple los requisitos establecidos, podríamos detallar algunas mejoras para este. Un ejemplo en cuanto a coste sería paralelizar el bucle que busca los *maximal firm subwords*, ya que en este para cada cadena de I el código está hecho para que no dependa de las iteraciones anteriores. Por lo tanto fácil de ejecutar en paralelo obteniendo una mejora de rendimiento equivalente a numI en esta zona del código. Otra mejora sería mejorar la interfaz a la hora de devolver un resultado incluyendo una librería grafica para devolver la cadena solución gráficamente y representada por colores distintos por cada trozo de ADN recombinado, y no en texto plano como se hace ahora. También se podrían mejorar las funciones del programa agregando una opción que pudiera guardar y cargar los AFN creados para su uso posterior en otras ejecuciones.

6. Anexo

A continuación definiremos las estructuras que hemos utilizado a la hora de implementar nuestra aplicación.

Lista enlazada S2:

```
public class S2 {

    private int index;
    private S2 sig;

    public S2(int index, S2 sig) {
        this.index = index;
        this.sig = sig;
    }

    public S2 getSig() {
        return sig;
    }

    public void setSig(S2 sig) {
        this.sig = sig;
    }

    public int getIndex() {
        return index;
    }
}
```

Lista enlazada S:

```
public class S {
    private S sig;
    private String sufijo;
    private String derecha;
    private int cadMadre;

    public S(String sufijo, String derecha, S sig, int cad) {
        this.sufijo = sufijo;
        this.derecha = derecha;
        this.sig = sig;
        this.cadMadre = cad;
    }

    public S getSig() {
        return sig;
    }
    public String getSufijo() {
        return sufijo;
    }
    public String getDerecha() {
        return derecha;
    }
    public int getCadMadre() {
        return cadMadre;
    }
}
```

Lista enlazada V2:

```
public class V2 {

    private int xant;
    private int yant;
    private int cadMadre;
    private V2 sig;

    public V2(int xant, int yant, int cadMadre, V2 sig) {
        this.xant = xant;
        this.yant = yant;
        this.cadMadre = cadMadre;
        this.sig = sig;
    }

    public int getXAnt() {
        return xant;
    }

    public int getYAnt() {
        return yant;
    }
    public V2 getSig() {
        return sig;
    }
    public int getCadMadre(){
        return cadMadre;
    }

    public void setSig(V2 sig){
        this.sig = sig;
    }
}
```

7. Bibliografía

[1] Biología Molecular. Fundamentos y aplicaciones. 2009. Beas, C., Ortuño, D., Armendáriz, J. McGraw-Hill.

[2] Molecular Biology of the Cell. 2002. Alberts B, Johnson A, Lewis J, et al.

[3] Regular languages, regular grammars and automata in splicing systems. AIP Conf. Proc. 1522, 856 (2013). Nurhidaya Mohamad Jan, Wan Heng Fong and Nor Haniza Sarmin.

[4] Recognition of Simple Splicing Systems using SH-Automaton. 2008. Fong Wan Heng, Nor Haniza Sarmin, Zuwairie Ibrahim. Journal of Fundamental Sciences (Volume 4, No. 2, 2008, Pages 337 to 342)