Universidad Autónoma de Madrid
Escuela politécnica superior

Doble Grado en Matemáticas e Informática

Trabajo Fin de Grado

IMPLEMENTACIÓN Y ANÁLISIS DE ALGORITMOS DE ALINEACIÓN PARA DATOS DE NEXT GENERATION SEQUENCING (NGS)

Autor: Daniel Giménez Llorente
Tutora: Irene Rodríguez Luján
Ponente: José Ramón Dorronsoro Ibero

JUNIO 2016

IMPLEMENTACIÓN Y ANÁLISIS DE ALGORITMOS DE ALINEACIÓN PARA DATOS DE NEXT GENERATION SEQUENCING (NGS)

Autor: Daniel Giménez Llorente
Tutora: Irene Rodríguez Luján
Ponente: José Ramón Dorronsoro Ibero

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
JUNIO 2016

Resumen

El coste del proceso de secuenciación de los genomas de los seres vivos, se ha reducido en gran cantidad en los últimos diez años debido a la aplicación de nuevas técnicas de secuenciación denominadas *Next Generation Sequencing* (NGS). Esta situación ha propiciado la aparición de muchos alineadores que nos permiten, dada una serie de secuencias provenientes de la secuenciación NGS, conseguir hallar la posición en el genoma del que proceden usando un genoma de referencia. Sin embargo es difícil escoger cuál es el alineador que mejor se puede adaptar a cada problema, dada la dificultad de encontrar comparaciones justas entre alineadores en términos de efectividad en la alineación y coste computacional. Además, dentro de un mismo alineador el correcto ajuste de los metaparámetros que definen cada uno de los alineadores es otro problema al que también se enfrentan normalmente los bioinformáticos. Estas dificultades se ven acentuadas por el hecho de que estos alineadores son comúnmente usados como cajas negras dada su complejidad y la dificultad de encontrar descripciones y análisis detallados de los mismos.

El objetivo de este TFG es el análisis teórico e implementación de tres alineadores para su posterior comparación, determinando en qué casos es mejor optar por la utilización de unos u otros. Adicionalmente, se proporciona un análisis de la influencia de sus metaparámetros en el rendimiento de cada alineador. Se han escogido Bowtie, BWA y BWT-SW, para cubrir tanto alineadores de secuencias cortas como largas, así como alineadores únicamente tolerantes a mutaciones y alineadores tolerantes a mutaciones y huecos. Estos alineadores usan una estructura común, FM-Index, que les permite realizar búsquedas en el genoma de referencia de forma óptima a través de la transformada de Burrows-Wheeler y propiedades. Este TFG ofrece una descripción detallada del funcionamiento de cada uno de los algoritmos, que ha sido posible tras el análisis de cada uno de los artículos científicos donde se encuentran definidos los mismos. A continuación, tanto los alineadores como el FM-Index se han implementado en C++, y su correcto funcionamiento se ha verificado mediante una serie de pruebas unitarias de caja blanca y caja negra.

Una vez implementados, se usó el programa ART para la simulación de un generador de secuencias NGS. Este programa recibe como parámetros la tecnología a simular y otros valores que permiten controlar la longitud de la secuencia de salida y las probabilidades de mutaciones y huecos. Se ha comparado el comportamiento de los tres alineadores para diferentes valores de estos parámetros en términos de tiempos de ejecución y tasas de acierto, haciendo variar a su vez los metaparámetros de cada alineador.

Las contribuciones derivadas de este TFG son un estudio detallado del funcionamiento de algunos de los algoritmos de alineación más utilizados, que pretende completar la literatura existente, una implementación sencilla de los mismos que facilita su mejor comprensión y comparación, y un análisis cuantitativo y comparativo de estos alineadores que permite determinar la naturaleza de los problemas de alineación para los cuales cada uno de ellos es más adecuado.

Palabras Clave Bowtie, BWA, BWT-SW, transformada de Burrows-Wheeler, alineadores, genoma.

Abstract

The cost of sequencing a living being's genome has been greatly reduced due to the advent of Next Generation Secuencing (NGS) techniques. This situation has led to the apparition of many new aligners that can find the position of a given NGS sequence in a reference genome. However, it is difficult to choose the aligner that best adapts to a problem given the shortage of fair comparisons between aligners in terms of alignment effectiveness and computational cost. Besides, another problem commonly faced by bioinformaticians is the correct adjustment of aligner's metaparameters. These difficulties are even greater considering that these aligners are commonly used as black-boxes due to their complexity and the lack of a detailed description and analysis of them.

The objective of this Masters Thesis is to provide a theoretical analysis and implementation of three aligners that will serve to compare them, showing which is the best algorithm for each type of NGS sequencing problem. Additionally, this work includes an analysis to determine the influence of aligners' metaparameters in their performance. In order to cover both long and short sequence aligners as well as aligners considering either only mutations (mismatches) or both mutations and gaps, three different aligners, namely Bowtie, BWA and BWT-SW, have been selected for the analysis. These aligners have a common structure, the FM-Index, that allows optimal searching in the reference genome with low memory consumption by using the Burrows-Wheeler transform and its properties. This Masters Thesis offers a description of the hidden details of every algorithm, which has been possible through the thorough study of scientific papers where these algorithms are proposed. Both the FM-Index and the aligners were implemented in C++, and their proper functioning was verified by black and white box unit testing.

Once the aligners were implemented, ART software was used to simulate NGS sequences. This software receives as parameters the NGS technology to simulate and other values to control sequences' length, number of mismatches, and gap probability. The behaviour of these three aligners for different values of these parameters has been compared in terms of execution time and hit rate, varying every aligner's metaparameters as well.

The outcomes of this Masters Thesis are a detailed study of some of the most used alignment algorithms based on the FM-Index, which intends to complement existing literature; a simple implementation of these aligners, which favors their comprehension and comparison; and a quantitative comparative analysis, which allows us to conclude when each aligner is more suitable than others for an specific sequencing problem.

Keywords Bowtie, BWA, BWT-SW, Burrows-Wheeler transform, aligners, genome.

Agradecimientos

En primer lugar quiero agradecer a Irene Rodríguez Luján por haber aceptado ser mi tutora en este TFG. No solo por haberme dado ideas que terminaron siendo la temática de este trabajo, si no por toda la ayuda prestada y el apoyo que me ha ofrecido a lo largo del desempeño del mismo. Sin ella, nada de esto se habría llevado a cabo.

También el Centro de Computación Científica (CCC) que ha permito el uso de sus máquinas para poder ejecutar toda la carga de pruebas de este TFG.

Además, debo dar las gracias a todos mis compañeros de carrera, que han estado conmigo durante cinco años y han creado lazos que son difíciles de romper.

Por último quiero agradecérselo a mi familia, sobre todo a mis padres y a mi hermana Irene. Gracias a ellos, que siempre me han animado e impulsado, he llegado hasta aquí y siempre les estaré agradecidos por ello.

Índice general

Índice de Figuras

Índice de Tablas

Glosario de acrónimos

- **NGS**: Next Generetion Sequencing
- **TFG**: Trabajo de Fin de Grado
- **BWA**: Burrows Wheeler Alignment
- **BWT-SW**: Burrows Wheeler Transform Smith Waterman
- **NCBI**: Centro Nacional para la Información Biotecnológica
- **ADN**: Ácido Desoxirribonucleico
- **ARN**: Ácido Ribonucleico
- **miARN**: micro ARN
- **SW**: Smith Waterman
- **NW**: Needleman-Wunsch
- **SA**: Suffix Array
- **DAWG**: Grafo acíclico dirigido

1 Introducción

1.1. Motivación del proyecto

La bioinformática es un campo multidisciplinar que utiliza tecnologías de información con la finalidad de responder a preguntas de índole biológico. Según la definición del Centro Nacional para la Información Biotecnológica (NCBI por sus siglas en inglés): "Bioinformática es un campo de la ciencia en el cual confluyen varias disciplinas tales como: biología, computación y tecnología de la información. El fin último de este campo es facilitar el descubrimiento de nuevas ideas biológicas así como crear perspectivas globales a partir de las cuales se puedan discernir principios unificadores en biología"[6].

Una de las principales vías de investigación en este campo incluye la secuenciación, mapeo y ensamblado de los genomas. Gracias a estas técnicas, se puede determinar la información genética que forma la base de todos los seres vivos

Este trabajo se va a centrar en el problema de mapeo o alineación. Éste consiste en comparar cada una las lecturas provenientes del proceso de secuenciación con una secuencia de referencia, como un genoma, previamente conocida. Cabe destacar la diferencia que hay entre el alineamiento y el ensamblado, ya que en éste último se usan las lecturas para reconstruir el fragmento de la secuencia del que procedían sin conocer la secuencia (genoma) de referencia. De forma intuitiva, alinear es rehacer un puzle, teniendo una imagen donde poder fijarse mientras que al ensamblar, no se dispone de ninguna información de referencia, y las piezas del puzle se van ensamblando en base a su similitud. El ensamblaje se usa principalmente para la reconstrucción de un genoma mientras que la alineación se suele utilizar para obtener la expresión diferencial de genes entre diferentes individuos o condiciones experimentales.

Un alineador es una herramienta que nos permite mapear las millones de lecturas cortas que se generan en la secuenciación e intentar determinar su posición en una genoma que posee un gran número de nucleótidos. Actualmente, gran parte del uso de alineadores de secuencias procedentes de *Next Generation Sequencing* se realiza con poco conocimiento sobre el mismo. Se trata a la herramienta como una caja negra que recibe secuencias y devuelve sus posiciones en el genoma. Además, hay poca información del método de implementación de cada algoritmo, incurriendo aún más en el sentido de esa caja negra. Debido a esto, la selección del alineador que más se ajuste a un problema, o, la selección de sus propios metaparámetros no es ni mucho menos sencilla [7]. Hay más de 60 alineadores diferentes y cada uno tiene sus particularidades en

términos, por ejemplo, de implementación -diferentes lenguajes- o formato de los datos [8]. El ser capaz de escoger escoger aquel alineador que sea más conveniente y entender a qué se debe esto es crítico, ya que permitirá optimizar tiempo y resultados.

1.2. Objetivos

El principal objetivo de este trabajo, es arrojar cierto grado de luz a algunos de estos algoritmos. Lo ideal hubiese sido poder estudiar y compararlos todos, pero eso no cabe dentro del marco de un TFG. Para ello se han escogido tres herramientas basadas en una estructura común, el FM-Index. Por tanto, este TFG se centra en tratar de explicar el funcionamiento de los algoritmos de estas herramientas y luego llevar a cabo una implementación de cada una de ellas para poder compararlas en términos de su efectividad en la alineación de diferentes tipos de lecturas NGS y en términos de su coste computacional. No se contempla como objetivo la implementación de la completa funcionalidad de cada de ellas, ya que probablemente ese trabajo sobrepasaría el asignado a un TFG. Sin embargo, si se han implementado versiones representativas de los algoritmos que nos permiten compararlos de una manera relativamente justa.

Las comparaciones actuales entre alineadores poseen principalmente dos problemas: por un lado, un sesgo derivado de la autoría de cada herramienta, ya que las comparaciones se realizan normalmente en los artículos de las nuevas herramientas; y, por otro lado, la falta de transparencia en la implementación, lo que hace difícil realizar comparaciones justas y realmente cuantificar cuál de los algoritmos en más eficiente en sí, más allá de la implementación utilizada.

1.3. Metodología y plan de trabajo

Esta memoria se divide en siete capítulos además de este capítulo introductorio, estructurados conceptualmente en cinco bloques. El primer bloque correspondiente al capítulo 2 aborda una descripción del estado del arte en algoritmos de alineación basados en secuencias NGS, centrándose en las características generales que permiten clasificar estos alineadores en diferentes grupos.

El segundo bloque se corresponde con el capítulo 3 dedicado al FM-index, donde se describe esta estructura de datos y la implementación realizada para ser usada como base de los alineadores estudiados en los siguientes capítulos. En este capítulo se explican las propiedades del FM-Index y como pueden ser usadas para resolver parte del problema de la alineación. Finalmente se describen los puntos claves de la implementación del FM-Index que acompaña a este TFG.

El siguiente bloque abarca los capítulos 5, 6 y 7 referentes a cada uno alineadores que se van a analizar. Los alineadores estudiados son: Bowtie, BWA y por último BWT-SW. La estructura de cada uno de estos capítulos es muy similar a la del capítulo 3 dedicado al FM-index: primero se explican los detalles del algoritmo para, seguidamente, presentar la implementación que se ha llevado a cabo

El siguiente bloque (capítulo 7) presenta las pruebas llevadas a cabo sobre los diferentes alineadores, así como los resultados obtenidos. En él se describen las pruebas realizadas y las diferentes comparaciones entre algoritmos en términos de precisión y tiempo de ejecución.

En último lugar se encuentra la sección de conclusiones y trabajo futuro, donde se resumen las principales conclusiones derivadas de este TFG y se plantean líneas de trabajo futuro.

2

Algoritmos de alineación. Estado del arte

2.1. Introducción

Los algoritmos de alineación son aquellos que nos permiten alinear un genoma tras el proceso de secuenciación, donde gracias a técnicas bioquímicas, podemos determinar el orden de los nucleótidos en una molécula de ADN. Los alineamientos de secuencias en bioinfórmatica son muy útiles, ya que nos permiten entre otras cosas, identificar similitudes entre secuencias o producir árboles filogenéticos (que muestran las relaciones evolutivas entre especies).

Sin embargo, los algoritmos de alineación están presentes también en otros campos fuera del ámbito biológico. Es el caso del procesamiento del lenguaje natural, donde gracias a estos algoritmos se pueden producir versiones lingüísticas de pruebas matemáticas generadas por ordenador [9]. En el campo de la lingüística histórica y comparativa, se han estado usando para automatizar el método comparativo a través del cual los lingüistas reconstruían lenguajes [10], o incluso en el campo del marketing analizando series temporales de compras[11].

2.2. Secuenciación de alto rendimiento o next-generation (NGS)

El término de *next-generation sequencing* (secuenciación de alto rendimiento o NGS) se usa para describir una serie de tecnologías modernas de secuenciación entre las que se encuentran: Illumina (Solexa) sequencing, Roche 454 sequencing o SOLiD sequencing.

Estas recientes tecnologías nos permiten secuenciar el ADN y el ARN de forma mucho más rápida y con menor coste en comparación con las tecnologías anteriores como microarrays. Si nos fijamos en la figura 2.2, el precio de secuenciación del genoma humano ha descendido en los últimos años desde los 10 millones en 2007 a poco más de 1000 dólares, gracias a la irrupción de las tecnologías NGS.

Un uso común de estas, es la generación de secuencias cortas de ADN o ARN, que nos permiten determinar la expresión diferencial de los genes. Gracias a esto podemos determinar genes que producen ciertas enfermedades comprobando los niveles de expresión entre células sanas y enfermas.

2.3. Alineación de secuencias

El problema de la alineación de alto rendimiento se puede definir de la siguiente manera. Dada una serie de secuencias Q (producidas por teconologías NGS), una cadena o genoma de referencia, y un umbral k, encontrar todas aquellas subcadenas de ese genoma que cumplen ciertas restricciones (principalmente de calidad de la lectura) y su distancia a las secuencias Q sea menor que el umbral k para una función distancia definida.

En este proceso de alineación hay que tener en cuenta diferentes aspectos. El genoma que se usa como referencia y el genoma que se secuencia no tienen por qué ser iguales. Gracias a la evolución, se han producido mutaciones en los genomas que nos proporcionan gran variabilidad genética. En la figura 2.1 se muestran las diferentes alteraciones que puede sufrir la cadena de nucleótidos. A todo esto hay que añadirle los posibles errores que se hayan podido producir en la secuenciación. Por tanto, el objetivo de la alineación será encontrar la posición del genoma de la que se ha obtenido la secuencia, teniendo a su vez en cuenta todas estas posibles variaciones.

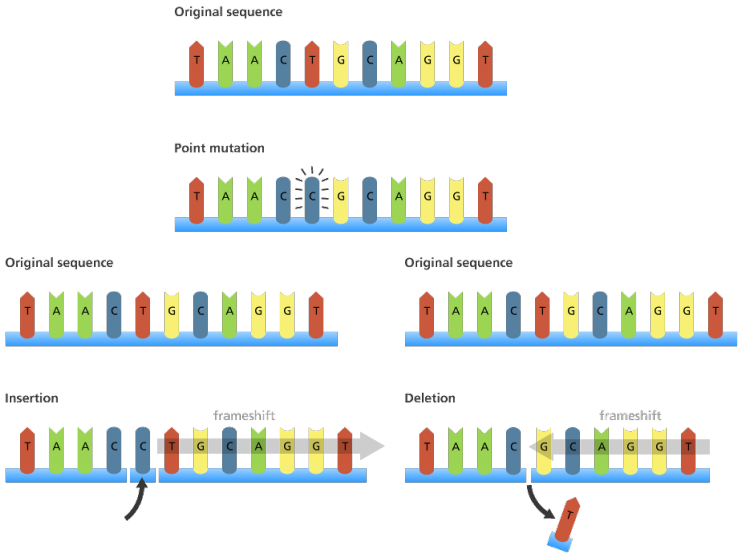


Figura 2.1: Ejemplo de tipos de alteraciones de una secuencia de ADN. Las mutaciones puntuales se corresponden con *mismatches* mientras que las otras dos se corresponden con *gaps* (también conocidos como indels o huecos) en la secuencia del genoma. Imágenes provenientes del Genome Research Limited [1].

Actualmente hay más de 60 alineadores diferentes, y uno de los asuntos más importantes es escoger aquel alineador que está adaptado al problema que queremos resolver. El abaratamiento (figura 2.2), rapidez y precisión de las tecnologías NGS provocaron una explosión de algoritmos de alineación a partir de 2008, por lo que se favoreció la creación de nuevos alineadores tal y como se muestra en la figura 2.3.

Los alineadores se clasifican en base a diferentes criterios. Podemos diferenciar los alineadores a un nivel muestral, es decir teniendo en cuenta las muestras que recibe. El tipo de secuencias o la longitud de éstas serían los parámetros a considerar. Por ejemplo, las secuencias procedentes de miARN (micro ARN) suelen tener longitudes pequeñas (16-30 bases). Por ello, los alineadores candidatos deben manejar de forma óptima este tipo de datos. Por el contrario, otras tecnologías proporcionan secuencias de más de 1000 bases, donde esos alineadores no serían los adecuados.

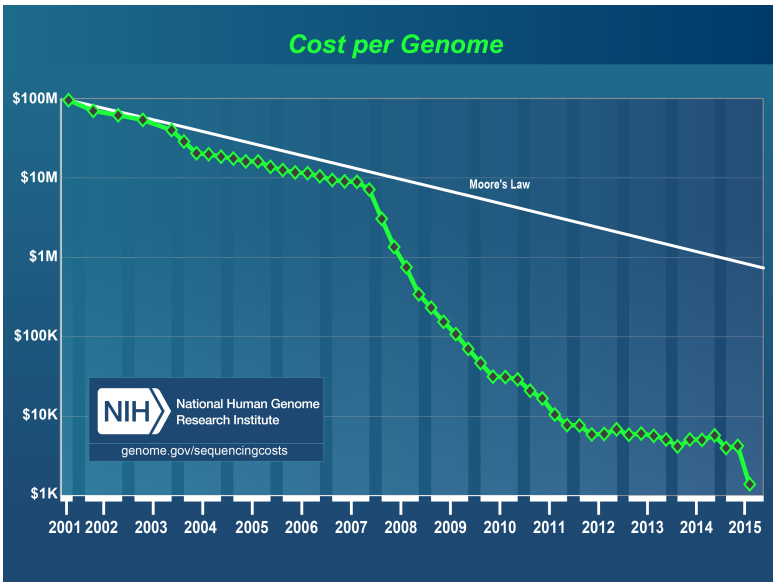


Figura 2.2: Coste de secuenciación del genoma humano [2].

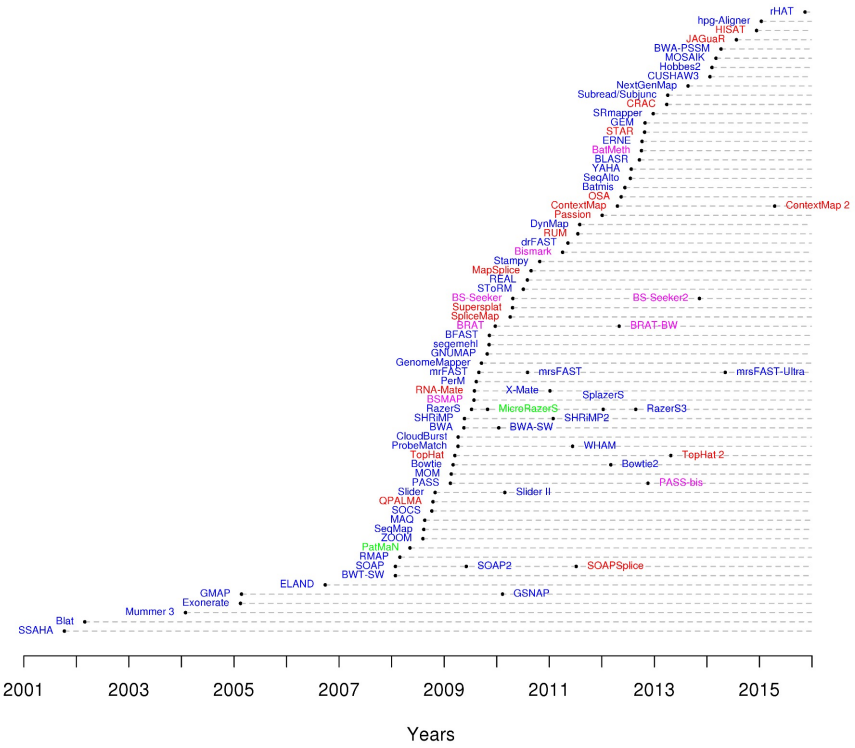


Figura 2.3: *Timeline* de alineadores desde el 2001.Los alineadores de ADN están en azul, los de ARN en rojo, miARN en verde y los de bisulfito en morado.Los puntos grises indican alineadores relacionados [3].

Por otra parte también podemos clasificar los alineadores en base a diferentes aspectos en su diseño e implementación [12]:

- **Indexado**. Existen diferentos métodos de indexar el genoma para llevar a cabo la alineación. Algunos utilizan estructuras de datos como tablas hash [13] para indexar y escanear la secuencia. Aplicando diferentes máscaras y funciones hash al genoma y a las secuencias,

se puede llegar a generar una estructura que nos permita el indexado para compararlas entre sí. Otro caso sería el uso de arrays de sufijos, que nos permiten un rápido alineamiento ordenando todos los sufijos de una secuencia. La ventaja de usar arrays de sufijos es que nos permite agrupar aquellas subcadenas iguales del ADN y compararlas una solo vez con la secuencia. Uno de los ejemplos más representativos del array de sufijos es la estructura FM-Index, que a partir de la transformada de Burrows-Wheeler nos genera un indexado y un método para recorrer eficientemente el genoma en búsqueda de coincidencias con determinada secuencia. Además, hay algunos alineadores que usan el indexado de las lecturas a mapear, luego existen multitud de algoritmos en base al tipo de indexado y la fuente del mismo (genoma y/o secuencias).

- **Semillas**. Una semilla se corresponde con una subcadena de la secuencia donde se permite un número máximo de *mismatches* l. Si dividimos esa subcadena en $l + 1$ fragmentos, al menos uno debe alinear perfectamente con el genoma. Esta idea es usada por algunos alineadores, que dividen en la cadena en $l + k$ fragmentos y obligan a que k alineen perfectamente. Si aumentamos el número de fragmentos que deben coincidir, aumentamos la probabilidad de encontrar la posición real de la secuencia en el genoma. También se pueden generar varias semillas de la secuencia y luego aplicar métodos probabilísticos para unirlos y obtener su posición.

- **Algoritmo de alineación**. Determinados métodos usan las semillas para detectar las zonas del genoma donde más se asemejan a las secuencias, pero no determinan su posición exacta. Para ello se pueden aplicar diferentes métodos. Por ejemplo usar aproximaciones basadas en programación dinámica como el algoritmo de Smith Waterman (SW) [14] o el Needleman-Wunsch (NW)[15], que nos proporcionan alineamientos locales y globales respectivamente. El alineamiento global se usa cuando las dos cadenas que se quieren alinear son similares y de igual tamaño, mientras que las búsqueda local identifica zonas de semejanza entre las secuencia y subcadenas del genoma (en este caso).

Los algoritmos escogidos para el análisis han sido Bowtie, BWA y BWT-SW. El Bowtie y el BWA son alineadores pensados para secuencias cortas, mientras que el BWT-SW está diseñado para secuencias largas. Además, el Bowtie solo permite errores de *Mismatches* mientras que los otros dos permiten también *Gaps*. De esta forma podemos comprobar en la sección 7.2 como, dependiendo del tipo de secuencia, algunos alineadores tendrán mejores resultados que otros.

Alineador	Número de citas	*Mismatches*	*Gaps*	Calidad
Bowtie	6928	Sí	No	Sí
BWA	7064	Sí	Sí	Sí
BWT-SW	99	Sí	Sí	No

Cuadro 2.1: Características de Bowtie, BWA y BWT-SW [3].

3
FM Index

3.1. Definición

El FM-Index [16] es una estructura de datos que nos permite el indexado y la búsqueda en cadenas de caracteres ocupando un espacio proporcional a su entropía. Fue desarrollada por Paolo Ferragina and Giovanni Manzini que la definen como una estructura de datos oportunista que nos permite el indexado de grandes cadenas de texto donde se pueden buscar eficientemente subcadenas teniendo una huella de memoria reducida. Surge de la combinación de la transformada de Burrows-Wheeler junto con las propiedades del array de sufijos; gracias a los cuales podemos realizar la búsqueda de una subcadena $W[1, p]$ en tiempo $O(p)$ en el peor de los casos, independientemente del tamaño de la cadena donde realicemos la búsqueda.

3.2. Transformada de Burrows-Wheeler

La transformada de Burrows-Wheeler (BWT)[17] es un algoritmo cuyo objetivo inicial era ser usado como técnica de compresión de datos en algunos programas como el bzip2 [18]. Consiste en una transformación de la cadena original en otra mediante el uso de las diferentes posibles rotaciones de la cadena, en la que la cadena transformada tiene los mismos caracteres que la original, pero permutados en cualquier orden. Es un proceso reversible muy útil cuando hay varias subcadenas iguales dentro de la cadena original. Esta situación produce que los caracteres se repitan varias veces seguidas en la transformada, lo cual es fácilmente aprovechable por un algoritmo compresor.

3.2.1. Algoritmo de creación de la Transformada de Burrows-Wheeler

Vamos a denotar como T la cadena que queremos transformar siendo $m = |T|$ su longitud. Para la creación de la BWT se tiene que definir también un caracter de terminación, que preceda lexicográficamente al alfabeto que pueda usar nuestra cadena. Como nos vamos a centrar en cadenas de ADN (que contienen solo cuatro letras A, C, G, T que hacen referencia a los cuatro tipos de nucleótidos) podemos tomar como caracter de terminación el '$'. De esta forma tenemos

$\$ < A < C < G < T$. El '$' es buen carater de terminación porque precede a todos los caracteres alfanuméricos.

Para la creación de la BWT se tienen que llevar a cabo las siguientes acciones:

1. Se añade al final de la cadena el símbolo de terminación.
2. Se crean todas las rotaciones posibles de la cadena a procesar y se ordenan lexicográficamente obteniendo una tabla única para cada cadena.
3. La última columna, que se corresponde con el último caracter de cada rotación, forma la transformada de Burrows-Wheeler.

Vamos a definir como M la matriz ordenada lexicográficamente, L la cadena de la transformada y F la primera columna de la matriz. Para ver el funcionamiento de este algoritmo tomamos como cadena que queremos transformar "$ACATGCA$", el resultado del algoritmo se muestra en la Tabla 3.1.

Cadena original	Rotaciones	Rotaciones Ordenadas (M)	F	BWT (L)
ACATGCA$	ACATGCA$	$ACATGCA	$	A
	$ACATGCA	A$ACATGC	A	C
	A$ACATGC	ACATGCA$	A	$
	CA$ACATG	ATGCA$AC	A	C
	GCA$ACAT	CA$ACATG	C	G
	TGCA$ACA	CATGCA$A	C	A
	ATGCA$AC	GCA$ACAT	G	T
	CATGCA$A	TGCA$ACA	T	A

Cuadro 3.1: Tabla de la matriz de la transformada de Burrows-Wheeler

3.2.2. Transformación inversa

La transformación inversa, desde la transformada de Burrows-Wheeler a la cadena original depende de dos teoremas relacionados con la matriz M:

Teorema 1 *Dada la $i-ésima$ fila de M; su último caracter, que coincide con $L[i]$, precede en el texto original a $F[i]$.*

Basta con fijarse que la fila posee un orden cíclico y todas sus filas son la misma cadena desplazada. Si tomamos una fila de la matriz de la tabla 3.1, podemos ver como el último caracter siempre precede al primero en la cadena original.

Teorema 2 ***Left-Mapping.*** *Sea $L[i] = c$ y $r_i = Occ(c, i)$ el número de veces que se repite c en $L[0, i]$, que se corresponde con la subcadena de L desde la posición 0 hasta i. Tomamos $M[j]$ como la $r_{i-ésima}$ fila dentro de las que empiezan por c. Entonces el caracter $L[i]$ se corresponde con el caracter de la primera columna en la posición j, es decir, $F[j]$ (esto se denomina LF-mapping, donde $LF[i] = j$).*

El teorema tiene un interpretación sencilla. La primera 'A' de la transformada de BW se corresponde en la cadena original con la primera 'A' de F. Lo mismo con la segunda, la tercera.., y el resto de caracteres. Un ejemplo, si volvemos a la matriz M de la figura 3.1 y nos fijamos en la segunda 'C' de la BWT (la cuarta fila) vemos que se corresponde con el segundo caracter de la

cadena original. Por otro lado, si nos fijamos en la segunda 'C' de F, se observa que es el mismo caracter de la cadena.

Para hacer la transformación inversa, que a partir de la BWT nos devuelve la cadena original, solo es necesaria la BWT, la cual primero se procesa para rellenar algunas estructuras que son usadas para revertir la permutación:

1. Se crea el array $C[1...|\Sigma|]$ (siendo Σ el lenguaje) guardando en $C[c]$ el número total de caracteres menores que c en la BWT según el orden lexicográfico. Es decir, si quisiéramos hallar C['G'], tendríamos que contar cuántas 'A' y cuántas 'C' tiene la BWT, ya que son los dos caracteres menores que 'G' (no se cuenta el \$).

2. Se usa *LF-mapping.*

   $LF[i] = C[L[i]] + Occ(L[i], i)$, siendo $Occ(L[i], i)$ el número de ocurrencias de $L[i]$ en el prefijo $L[0, i]$.

   El objetivo de *LF-Mapping* es, dado un caracter $c = L[i]$ en una posición i de la BWT hallar que fila ocupa en F. Como están ordenadas lexicográficamente, sabemos que las filas que comienzan por c está contiguas y empiezan en la fila $C[c]$ (por como se definió en la paso anterior). ¿Pero cuál de todas esas filas se corresponde con c? Como sabemos que la primera c en la BWT es la primera de F y así sucesivamente, tenemos que sumar a $C[c]$ el número de veces que haya salido c en $L[0, i]$.

3. Tenemos una nueva fila $j = LF[i]$, donde el primer caracter era el usado para hacer LF. Por el teorema 1, sabemos que el caracter $L[j]$ precede en el texto a $F[j]$, luego hemos avanzado una posición hacia el principio de la cadena original. Si repetimos este proceso, iremos reconstruyendo la cadena hasta llegar al principio. Este procedimiento esta detallado en el Algoritmo 1.

$s \leftarrow 0$;
$T \leftarrow$ "";
$u \leftarrow |L| - 1$;
while $L[s] \neq \$$ **do**
  $T[u] \leftarrow L[s]$;
  $s \leftarrow LF[s]$;
  $u \leftarrow u - 1$;
end
return T;

Algorithm 1: UNPERMUTE. Algoritmo para deshacer la transformada de Burrows-Wheeler. T es el genoma mientras que L es la BWT de T.

La ejecución del Algoritmo 1 permite recuperar la cadena original empezando por el final de la misma, de ahí que sea un proceso de *backtracking.* Para ver el como se desarrolla el algoritmo, vamos a usar la BWT hallada antes para recuperar la cadena original T. La transformada es $L =$ "$AC\$CGATA$". Para revertir el proceso, primero tenemos que rellenar las tablas $Occ(L[i], i)$ y $C[c]$ tal y como se describió anteriormente. El resultado se detalla en la tabla 3.2.

Después solo queda seguir el algoritmo como se muestra en la figura 3.1. Se empieza en la posición 0 de la BWT que se corresponde con una 'A'. Esta 'A' es el último caracter de la cadena original. Como es la primera 'A' de la BWT, sabemos que se corresponde con la primera A de F, que está en la fila 1. Por el teorema 1 sabemos que L[1] precede en la cadena a F[1]. En este caso tendríamos que la 'C' precede a la 'A' en la cadena original. Esta 'C' es el penúltimo caracter de la cadena original.

i	$L[i]$	$Occ('A', i)$	$Occ('C', i)$	$Occ('G', i)$	$Occ('T', i)$
0	A	1	0	0	0
1	C	1	1	0	0
2	$	1	1	0	0
3	C	1	2	0	0
4	G	1	2	1	0
5	A	2	2	1	0
6	T	2	2	1	1
7	A	3	2	1	1

c	$C[c]$
A	1
C	4
G	6
T	7

Cuadro 3.2: Tablas $Occ(c, i)$ y $C[c]$ para el cálculo de la inversa de la transformada de Burrows-Wheeler.

Después repetimos el algoritmo con esa 'C'. Como es la primera 'C' de la BWT se corresponde con la primera 'C' de F ;y así sucesivamente se va recuperando la cadena original "$ACATGCA$".

Figura 3.1: Algoritmo UNPERMUTE de la cadena "$AC\$CGATA$"

3.3. Estructura de datos FM-Index

El procedimiento descrito en el apartado anterior, nos permitía recuperar una cadena partiendo de su transformada de Burrows Wheeler, pero podemos extender las propiedades que tiene esta transformada para realizar otras operaciones más complejas. En particular, esta sección se centrará en el uso de la transformada BWT para la búsqueda de una secuencia dentro de una cadena, ya que es el problema a resolver por los algoritmos de alineación que tratan de alinear las lecturas cortas procedentes de la secuenciación NGS (W) con un genoma de referencia (T).

La matriz M se corresponde con un array de sufijos, ya que todas sus filas son rotaciones de una misma cadena y dispuestas en orden lexicográfico de tal forma que la subcadena formada

por los caracteres de cada fila hasta el $ es un sufijo de la cadena original T. Además, todos los posibles sufijos se corresponden con alguna fila. Si queremos buscar la subcadena W dentro de T, podemos observar que, en la matriz M asociada a T, aquellas filas que tienen como prefijo a W se encuentran contiguas (gracias a que están ordenadas lexicográficamente). De esta forma, si W está en T existe un intervalo de filas que cumple que todas ellas tienen a W como prefijo. Definimos:

$sp(W) = \text{mín}\{i : W \text{ sea un prefijo de } M[i]\}$
$ep(W) = \text{máx}\{i : W \text{ sea un prefijo de } M[i]\}$

Entonces queda definido un intervalo de filas de M que recibe el apelativo de SA (Suffix Array). $SA(sp, ep)$ será el intervalo de filas de M que contienen a W como prefijo. Por ejemplo, si tomamos $W =$ "CA" en la matriz M de la Tabla 3.1, tendríamos que $sp(W) = 4$ y $ep(W) = 5$. Luego las filas que cumplen tener a W como prefijo se corresponden con $SA(4, 5)$.

3.3.1. Búsqueda exacta

Dado un caracter del alfabeto $c \in \Sigma$ y $Occ(c, i)$ el número de ocurrencias de c en $L[0, i]$ Ferragina y Manzani demostraron que si W es una subcadena de T se deben verificr las siguientes igualdades:

1. $sp(cW) = C[c] + Occ(c, sp(W) - 1) + 1$,
2. $ep(cW) = C[c]+Occ(c, ep(W))$, siendo cW la cadena formada al concatenar el caracter c al comienzo de W.

La cadena cW estará contenida dentro de la cadena original T si y solo si $sp(cW) \leq ep(cW)$. Gracias a esto, podemos comprobar si una cadena W está contenida en otra en tiempo $O(|W|)$. Cada vez que buscamos un nuevo caracter de W, recalculamos los índices sp y ep y vemos si sigue contenida en W. Si W no estuviese contenida, habría un caracter en W que haría que $sp > ep$, luego en este caso el tiempo sería menor a $O(|W|)$. Por otra parte, si W sí estuviese contenida en T, en $|W|$ pasos lo verificaríamos. Además, cabe destacar que la forma de recorrer este estructura simula el recorrido de un árbol de prefijos. La búsqueda exacta se detalla en el en el Algoritmo 2.

$c \leftarrow W[p]$;
$sp \leftarrow C[c]$;
$ep \leftarrow C[c + 1] - 1$;
$i \leftarrow p - 1$;
while $sp \leq ep$ *or* $i \geq 0$ **do**
  $c \leftarrow W[i]$;
  $sp \leftarrow C[c] + \text{Occ}(c, sp - 1) + 1$;
  $ep \leftarrow C[c] + \text{Occ}(c, ep)$;
end
return sp,ep;

Algorithm 2: EXACTMATCH. Algoritmo para hallar los índices del SA de una determinada secuencia $W[0, p]$

En la figura 3.2 se representa como varían los índices sp y ep al buscar la cadena $W =$ "CGA" en la cadena "$ACATGCA$" de los ejemplos. Primero se inicializan sp y ep al rango de filas que empiezan por el último caracter de W, en este caso 'A'.

$sp = 1$

$ep = 3$

Después el algoritmo se va desplazando hacia la izquierda de la secuencia W actualizando los índices sp y ep a partir de las tablas 3.2. En este caso, al existir la cadena W en T, el algoritmo termina cuando se ha recorrido de derecha a izquierda W y, por tanto, se verifica que $sp \leq ep$ en todos los pasos.

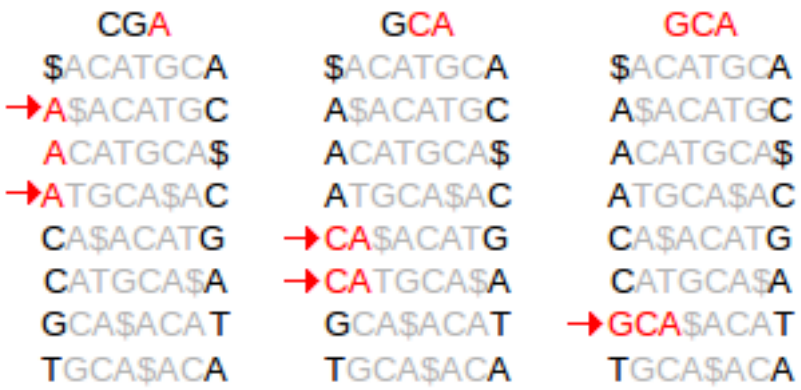


Figura 3.2: Ejemplo de búsqueda exacta de la subcadena "CGA" en la cadena "$ACATGCA$" usando el Algoritmo 2.

3.3.2. El FM-Index en el problema de la alineación genómica

El FM-Index estaba diseñado principalmente para la compresión de textos, pero diferentes campos de la bioinformática también le han dado un nuevo uso. Por ejemplo se ha usado esta estructura para elaborar la cuenta de oligómeros [19] o para llevar a cabo el diseño de pruebas para matrices de microarrays de ADN [20].

Dado el ámbito de este trabajo, esta sección se centrará en el análisis y descripción del uso del FM-Index en el problema de la alineación de genomas. El algoritmo de búsqueda exacta basado en el FM-Index descrito en la Sección 3.3.1, muestra cómo es posible determinar la existencia o no de una secuencia dentro de una cadena en un tiempo de ejecución lineal con la longitud de la secuencia a buscar. Esto es muy conveniente para el problema que queremos abordar: queremos buscar todas las posiciones del genoma donde cierta secuencia coincide. Sin embargo la salida del algoritmo de EXACTMATCH nos devuelve solo los índices de las filas de la matriz que contienen como prefijo la secuencia dada, pero no sabemos con cuál posición del genoma se corresponden.

Una primera solución a este problema sería usar *backtracking* para hallar la posición. Si recordamos, en la Sección 3.2.2, podíamos reconstruir la cadena original a partir de la BWT. Si empezamos en la primera fila recuperaríamos la cadena entera, pero si lo hacemos en cualquier otra, recuperaríamos solo el prefijo de la cadena original que llega hasta ese caracter. Si podemos hallar ese prefijo, nos basta con saber su longitud para saber cuál es su posición en el genoma. Diciéndolo de otra forma, su posición es igual al número de veces que se realiza *backtracking* hasta llegar al principio de la cadena original. Sin embargo esta solución no es nada óptima en tiempo computacional, ya que calcular la posición de una fila requiere un número de *backtracks* que es lineal con la longitud del genoma de referencia, que puede ser extremadamente grande. Por ejemplo, el genoma humano consta de aproximadamente tres mil millones de nucleótidos.

Otra posible solución sería, mientras se está creando el índice, precalcular y guardar de forma paralela a la BWT un array que contenga para cada posición de la cadena BWT su posición correspondiente en el genoma. Este método no es óptimo en coste de memoria. En el caso del

ser humano, cuyo genoma ocupa unos 3GB se necesitarían unos 12GB para almacenar solo las posiciones, ya que el espacio necesario para guardarlas es ,siendo n la longitud de T, proporcional a $n \log_2(n)$.

La implementación final se encuentra a caballo de estas dos soluciones. En vez de guardar todo el array de posiciones, solo se guarda una porción de ellas, cada cierto número k. Si la fila de la que quiere hallar su posición es múltiplo de k entonces se tendría el resultado directo; si no, se realizaría recursión usando *LF-mapping* hasta llegar a una fila que si sea múltiplo de k. Si se aumenta el número k se disminuye la huella de memoria, pero aumenta el tiempo de ejecución debido a la recursión y viceversa.

Más concretamente, cuando la fila de la que se quiere hallar la posición no es múltiplo de k se tiene que hacer *backtracking*. Por cada *backtrack* que se realice, se desplaza una posición a la izquierda en el genoma. Así, cuando llegamos a una fila múltiplo de k, la posición inicial sería la posición de esta última fila más el número de *backtracks* realizados. En la figura 3.3 se observa las posiciones almacenadas en el array de posiciones para un $k = 3$.

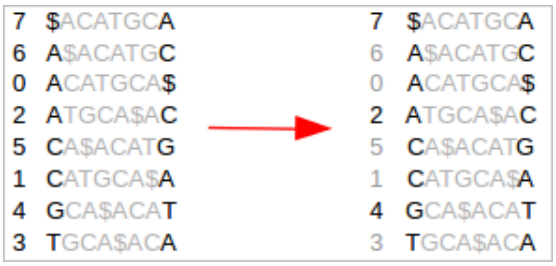


Figura 3.3: Posiciones almacenadas en el FM-Index para un $k = 3$ y la cadena "$ACATGCA$"

3.4. Implementación

La implementación del FM-Index e ha realizado en C++ (aunque casi todo el código es C nativo), para poder aprovechar al máximo la velocidad y el control sobre la memoria; los principales factores en el problema de la alineación. Esta estructura es común en las tres herramientas que se expondrán en los capítulos 4, 5y 6. Esto hace que la comparación sea más justa ya que las tres tienen la misma estructura aunque la utilicen de diferentes formas.

El FM-Index es una estructura formada por los siguientes campos:

- `bwt`: Se corresponde con la transformada de Burrows-Wheeler. En vez de tener los caracteres de la BWT guardados directamente, estos primero sufren una conversión. Como solo hay cuatro caracteres (nucleótidos) podemos usar dos bits para codificar cada uno, de tal forma que se pasa de ocupar un nucleótido por byte a cuatro nucleótidos por byte; reduciendo así el uso de memoria. Debido a esto, se han implementado funciones que nos permiten obtener dada una posición de un byte, el nucleótido que contiene y viceversa.

- `positionDollar`: La BWT realmente no sólo contiene los cuatro caracteres de los nucleótidos, sino además en alguna posición debe estar el caracter de terminación(en nuestro caso el '$'). Si lo hubiéramos guardado en la estructura anterior, hubiésemos necesitado un bit más, luego es conveniente no cambiar la codificación y guardar su posición aparte.

- `ranking`: Se corresponde con al array $C[c]$. Contiene para cada caracter (A,C,G,T) el número de caracteres menores que él dentro de la BWT. Se corresponde con la Tabla 3.2.

- positions: Es un array que contiene para cada fila de la BWT su posición real en el genoma. Como se explicó en la sección 3.3.2, no se almacenan las posiciones en el genoma de todos los caracteres de la BWT, si no que solo se guarda un subconjunto de las mismas. En nuestro caso se ha escogido guardar aquellas las filas con módulo 0 en el anillo $\mathbb{Z}_{32}$ (la misma política que el Bowtie original) [4].

- checkpoints: Se corresponde con $Occ(c, i)$ siendo c un nucleótido. Es una tabla parecida a la que se muestra en la Tabla 3.2. En ella se almacena para cada posición i de la BWT, cuántos caracteres de cada tipo han salido. Al igual que el campo anterior, guardar cada fila es muy costoso, ya que tendríamos que guardar cuatro valores para cada caracter de la BWT. Por tanto, se adopta una estrategia similar al caso anterior y se guarda el valor de $Occ(c, i)$ para aquellas filas i que tienen módulo 0 en el anillo $\mathbb{Z}_{128}$. Si la fila la cual queremos saber sus ocurrencias no es múltiplo de 128, basta con contar cuántas veces aparece el caracter c en el recorrido hasta el múltiplo de 128 más cercano y sumar/restar este valor a $Occ(c, i)$.

- nNuc: El número de nucleótidos del genoma de referencia.

Para rellenar esta estructura a partir de un genoma de referencia se necesita primero calcular la BWT. Para ello primero se crea un array P tal que $P[i] = i, \forall i = 0, .., |T|$. Después se ordena este array usando `qsort`, y como función de comparación una función que compara lexicográficamente las subcadenas del genoma que empiezan por las dos posiciones a comparar. De esta forma, P contiene la permutación ordenada de las rotaciones, verificándose T[P[i]]=BWT[i]. Gracias a este array se rellena toda la estructura del FM-Index. Una vez hecho esto, se libera el array con todas las posiciones y la cadena que tiene el texto original.

También se han implementado funciones para guardar y cargar el FM-Index, de tal forma que no sea necesario generarlo siempre que se quiera usar, sino que será suficiente con leer la estructura de datos descrita anteriormente.

Como se discutió en el capítulo 2, este TFG se centra en el análisis, implementación y comparativa de tres de los algoritmos de alineación más populares basados en el FM-Index. Por tanto la correcta comprensión de esta estructura de datos es pieza fundamental para abordar satisfactoriamente el análisis e imlementación de los alineadores Bowtie, BWA, BWT-SW de los que tratan los siguientes capítulos. El código de las funciones principales se encuentra en el Anexo B.

4

Bowtie

4.1. Definición

Bowtie es un alineador de secuencias cortas definido por sus creadores como un alineador ultrarrápido y eficiente en memoria [4]. Fue desarollado por el equipo de Ben Langmead en el Center for Bioinformatics and Computational Biology, en el Institute for Advanced Computer Studies,de la Universidad de Maryland, publicado en 2008. Surgió de la necesidad de encontrar un método más rápido para alinear secuencias cortas procedentes de secuenciadores de última generación, ya que el tiempo de ejecución de estos alineadores en ese momento era demasiado elevado.

Bowtie usa una forma nueva y diferente de indexado. En contra de las tecnologías que usaban métodos de tablas hash, Bowtie hace uso del FM-Index y por consiguiente de la transformada de Burrows Wheeler. Gracias a esta estructura es capaz de reducir considerablemente el tiempo de ejecución en comparación con otros alineadores anteriores como el MAQ (35 veces más lento que Bowtie) o SOAP (unas 300 veces más costoso computacionalmente que Bowtie)[4]. Además la estructura se encuentra optimizada de tal forma que la huella de memoria no sea excesivamente grande, aproximadamente de 1,3GB para el genoma humano.

4.2. Alineación de secuencias cortas

En esta sección se va explicar el uso del FM-Index para la búsqueda de secuencias en el genoma. La búsqueda exacta de una secuencia se puede realizar usando el algoritmo básico del FM-Index, que, de forma recursiva y en tiempo de ejecución lineal con la longitud de la secuencia, nos da si existen o no posiciones en el genoma que nos coincidan con ella.

La búsqueda exacta es insuficiente para resolver el problema de la alineación genómica. Esto se debe a que el genoma que usamos de referencia puede no coincidir exactamente con el genoma del que proceden las secuencias. Esto se puede deber a variabilidad genética o a la variabilidad técnica (fallos en la secuenciación), tal y como se discutió en la Sección 2.3. Bowtie, solo es capaz de lidiar con fallos de *mismatches* (debidos a mutaciones o errores). Estos fallos se producen cuando el genoma y la secuencia difieren en los nucleótidos en algunas posiciones. Sin embargo

no es capaz de detectar *gaps* en la secuencia (inserciones o borrados de nucleótidos). Además Bowtie está diseñado para un funcionamiento óptimo con un número máximo de tres errores.

Para llevar a cabo la búsqueda inexacta, primero hay que tener en cuenta la información contenida en los *reads* o secuencias. Éstos, además de la cadena secuenciada contienen una cadena de igual longitud que la anterior que contiene los valores de calidad de cada nucleótido secuenciado[21]. Estos números denominados *Phred Quality Scores* o *q-scores*, Q, se relacionan de forma logarítmica con la probabilidad P de que la base sea incorrecta a través de la siguiente fórmula: $Q = -10log_{10}P$. Dentro de las secuencias que contengan el mismo número de errores, será más valida la que tenga una mayor calidad.

Phred Quality Score	Probabilidad de base incorrecta	Porcentaje de precisión
10	1 in 10	90 %
20	1 in 100	99 %
30	1 in 1000	99.9 %
40	1 in 10,000	99.99 %
50	1 in 100,000	99.999 %
60	1 in 1,000,000	99.9999 %

Cuadro 4.1: Correspondencia de los *Phred Quality Score* (Q) y la probabilidad de error en la secuenciación.

4.2.1. Búsqueda inexacta

La búsqueda inexacta empieza igual que la búsqueda exacta con la diferencia de que, cuando el algoritmo llega a un intervalo vacío (SA con sp<ep, es decir, no hay coincidencia exacta con el genoma) se selecciona una base anterior ya examinada y se altera su valor, convirtiéndola en un *mismatch* en esa posición. De esta forma el algoritmo puede continuar y, debido a que se ha retrocedido en la búsqueda, a este paso se le denomina *backtrack*. El *backtrack* solo se aplica si ese caso está contemplado en la política de alineación del usuario; por ejemplo, si el usuario ha indicado un máximo de dos errores y el algoritmo ya ha aplicado dos veces el *backtracking*, no volvería a hacerlo una tercera vez. En la figura 4.1 se observa como se realiza *backtracking* para llegar a un intervalo no vacío.

Esta política de poda es insuficiente debido a que se puede generar un exceso *backtracking*. Si se produce el *backtracking* en los primeros niveles de la secuencia se llega a producir elevado coste en tiempo de ejecución ya que se explorarían todas las posibles posiciones donde podría producir el *mismatch*. Para evitar estas situaciones, Bowtie usa un doble indexado: crea un FM-Index con la cadena original (*forward index*) y otro FM-Index con la cadena al invertida (*mirror index*). En los siguientes párrafos se describe el uso de estos dos índices para diversas casuísticas en función al número de *mismatches* que se desea permitir en la alineación de la secuencia con el genoma de referencia. Es el momento de recordar que el algoritmo de búsqueda del FM-Index hace que nos movamos en el genoma de derecha a izquierda. Por el contrario, cuando se trabaja con el mirror index obtenido a partir de la cadena invertida, el algoritmo de búsqueda del FM-Index produce un recorrido de izquierda a derecha del genoma.

Fase 1. Un solo *mismatch* como máximo

Dada una secuencia de tamaño m, los alineamientos válidos en esta fase corresponderían a uno de los siguientes tres casos excluyentes: (i) un alineamiento exacto, (ii) un error en la primera mitad de la secuencia, o (iii) un error en la segunda mitad de la secuencia. Estas posibilidades

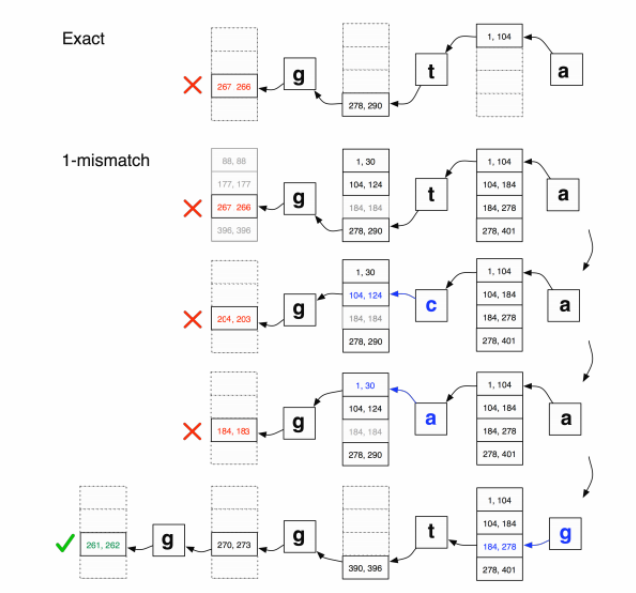


Figura 4.1: Algoritmo de búsqueda inexacta de Bowtie. Cada par de números hace referencia al SA correspondiente. Cuando se produce que $sp > ep$ no hay búsqueda exacta y se realiza backtracking [4].

se reflejan en la figura 4.2, donde la secuencia es la cadena "gccg...agca". Los dos primeros casos (i) y (ii) caen dentro del Caso 1 (0-1 errores en la primera mitad de la secuencia "gccg..", 0 en la segunda "..agca") , mientras que el caso (iii) cae en el Caso 2 (0 errores en la primera parte "gccg.."" y 1 error en la segunda).

Para tratar el Caso 1 basta con usar el *forward index* obligando a realizar una búsqueda exacta al principio del algoritmo. Así, como el *forward index* nos permite buscar desde el final de la secuencia hacia hacia el principio, se empezará con una búsqueda exacta en la parte final de la secuencia"..agca", que es donde no se pueden producir errores. Cuando se llega a la mitad de la secuencia, se cambia a una búsqueda con *backtracking* ya que en esta mitad sí puede haber *mismatches*.

En el Caso 2, el error se localiza en la segunda mitad de la secuencia. Si usáramos el *forward index*, como se mueve de derecha a izquierda buscando la secuencia, se calcularían todas las posibles posiciones donde podría estar el *mismatch* en la segunda mitad, y luego para cada una de ellas habría que realizar la búsqueda exacta en la primera mitad. Sin embargo, al usar el *mirror index*, y dar la vuelta a la secuencia (que sería "accga...gccg"), se pasa a tener lo mismo que en el Caso 1; primero una búsqueda exacta en la parte final de la secuencia invertida (es decir, la parte inicial de la cadena original), para luego cambiar a una búsqueda con *backtracking*. Obsérvese que esta estrategia reduce el coste computacional del algoritmo, ya que se limita el número de casos a explorar.

Fase 2. Un máximo de 2 *mismatches*

El problema se complica cuando se permiten hasta dos errores. Esta fase es más vulnerable al exceso de backtracking, ya que si hay dos errores, el número de backtracks será proporcional al producto de los backtracks de cada error en el peor de los casos. Los diferentes escenarios considerados en esta fase se muestran en la Figura 4.3. Esta fase es incremental a la Fase 1, ya que usa el Caso 1 y el Caso 2 de la anterior fase pero con algunas modificaciones. El Caso 1 de la Fase 2 permite hasta dos errores en la primera parte de la secuencia pero ninguno en la

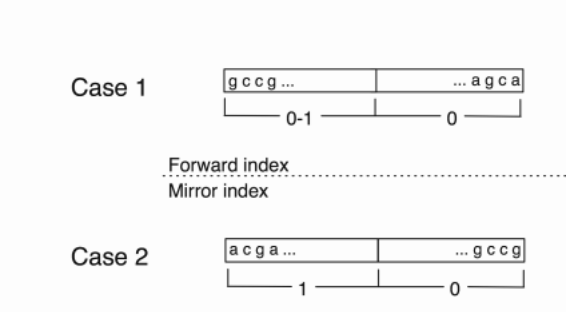


Figura 4.2: Fase 1.Búsqueda 1 mismatch como máximo [5].

trasera, mientras que en el Caso 2 obliga a tener 1 o 2 errores en la parte trasera y ninguno en la delantera.

Además se incluye un caso nuevo, el Caso 3, explorado con el *forward index* que se corresponde con tener un error en la parte delantera y otro en la trasera. Este caso es relativamente

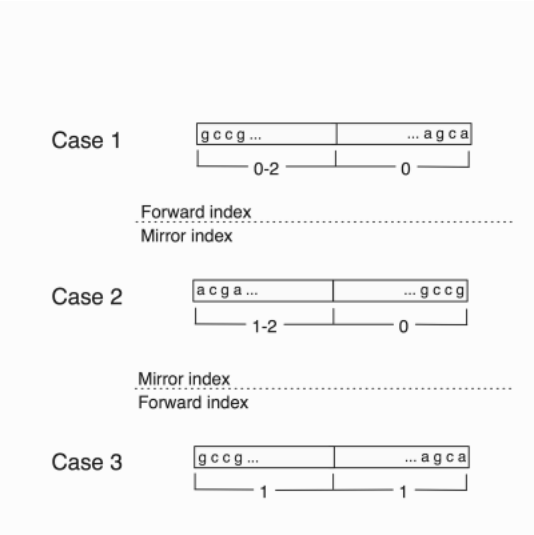


Figura 4.3: Fase 2. Búsqueda de como máximo dos errores [5].

sencillo, pero también es el que puede producir el mayor número de *backtracks*. Se puede observar como estos tres casos cubren todas las posiblidades de tener dos o menos *mismatches*.

Fase 3. Un máximo de 3 *mismatches*

Por último en esta fase se permiten hasta 3 errores. Al igual que en el fase anterior, esta fase hace uso de los casos ya definidos en la Fase 2, pero con algunas diferencias. En la Figura 4.4. se pueden observar todos los casos de la Fase 3, en los que además se indica el uso del forward index o el mirror index.

En el Caso 1 se permiten hasta 3 errores en la parte delantera de la secuencia de referencia y 0 *mismatches* en la trasera. El Caso 2 por su parte, permite de 1 a 3 errores en la parte trasera de la secuencia y ninguno en la parte delantera. El Caso 3 también se ve modificado con respecto a la Fase 2 permitiendo 1 o 2 errores en la parte delantera y obligando a que haya un *mismatch* en la segunda. Por último se introduce un caso nuevo, Caso 4 en el hay 1 error en la parte delantera de la secuencia y 2 en la trasera. Este último caso vuelve a usar el *mirror index* en vez del *forward*, ¿a qué se debe esto? A que es más eficiente buscar primero en la parte

que contiene menos errores. Al usar el mirror, se recorre la cadena original ("gccg...acga") de izquierda a derecha, por lo que se trataría primero la parte de la secuencia con un solo error, reduciendo así el número de *backtracks* y, por tanto, el coste computacional del algoritmo.

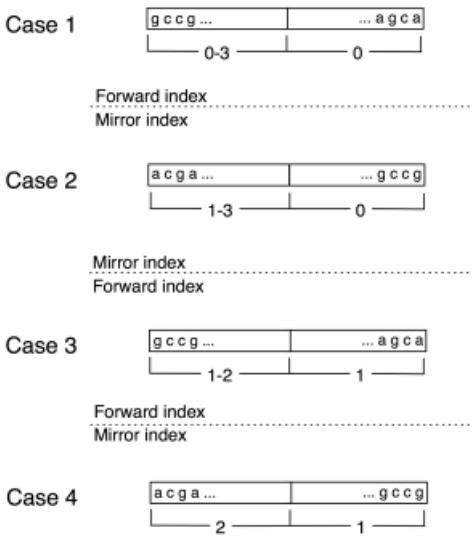


Figura 4.4: Fase 3.Búsqueda de tres *mismatches* [5].

4.2.2. Búsqueda *Maq-like*

Las estrategias descritas hasta el momento tienen unas políticas basadas solamente en el número de errores. Sin embargo el algoritmo Bowtie añade la política del alineador MAQ [22] a la suya propia, la cual considera el valor de los *quality score* de la secuencia. De primeras, MAQ calcula la suma de los *quality score* de cada una de las posiciones donde se hayan producido los errores y descarta aquellas secuencias que superen un umbral, que por defecto es 70. Esto tiene sentido, ya que cuanto más alto sea el *quality score* menor es la probabilidad de que hubiera un error en esa posición, y por lo tanto que se generase esa secuencia. Por ejemplo, si se tuviera una alineación con dos errores, y la suma de las calidades en esas posiciones fuese 60, sería un alineamiento válido; sin embargo, si el quality score en cada una de las posiciones mutadas fuese 40, la alineación se descartaría ya que su suma sería mayor que el umbral. Cuanto mayor sea el valor en una posición, menos posible es que se haya producido el error ahí. Además MAQ permite hasta 3 errores en las primeras 28 bases de la secuencia, que reciben el nombre de semilla. Puede haber cualquier número de errores fuera de la semilla, pero éstos siguen sumando sus *quality score* de tal forma que se puede descartar una secuencia cuando supera el umbral. El sentido biológico de esto es que, en la etapa de la secuenciación, la probablidad de cometer errores es acumulativa y aumenta según se va avanzando en la secuenciación [23].

Debido a estas diferencias, Bowtie tiene que sufrir ciertas modificaciones para poder tener en cuenta el *quality score* y la semilla. Para un funcionamiento rápido y efectivo, tiene que tener en cuenta la suma de los scores donde se han ido produciendo los errores. Así, si cuando se realiza la búsqueda inexacta se supera el umbral, se descarta ese camino del *backtracking* sin tener que haberlo explorado totalmente. Además, la semilla es más restrictiva que el resto, luego para una búsqueda efectiva, es prioritario buscar aquellas posiciones donde efectivamente la semilla se encuentre en el genoma (con el número de errores que se permita), para luego extenderla al resto de la secuencia. Así que primero se trata de alinear la semilla y luego se relaja la búsqueda en el

resto de la secuencia, permitiendo cualquier número de errores pero añadiendo el *quality score* de aquellas posiciones en las que no coincidan las bases.

Para incorporar estos criterios a los casos analizados en la Sección 4.2.2, hay que modificar estos casos de tal forma que el paso primero y fundamental sea la alineación de la semilla, para posteriormente ocuparse del resto de nucleótidos de la secuencia. El problema surge cuando el índice usado para tratar la semilla no sirve para extenderla. Imaginemos una secuencia donde se usa el *forward index* para encontrar la posición de la semilla. Una vez aplicado esto, obtenemos las posiciones del genoma donde se alinea la semilla, pero nos falta por combrobar si coinciden también el resto de la secuencia que no es la semilla. Como no tenemos el genoma original (recordar que éste se liberó una vez construído el FM-Index con el fin de reducir el coste en memoria), no podemos comparar directamente con él para ver si coincide, pero lo que sí podemos hacer es tomar esa subcadena que hemos alineado con el genoma, y buscarla en sentido contrario con el *mirror*. Como se puede observar en la figura 4.5, en aquellos casos que se use el *forward index*, después hay que llevar a cabo una búsqueda exacta con la cadena modificada incluyendo los errores usando el *mirror index* hasta el final de la semilla. Y así tendríamos la posición y los índices donde empieza la parte que no es semilla.

Figura 4.5: Alineación mediante Bowtie de la parte que no es semilla. Una vez se tienen los índices donde empieza esta parte, se va comparando cada nucleótido con el genoma incrementando la suma de los *score* cuando haya discordancia (*mismatch*).

Por último, en las figuras 4.6 y 4.7 se pueden observar la representación de todos los casos de cada tipo de búsqueda que realiza Bowtie cuando se aplican políticas MAQ-like.

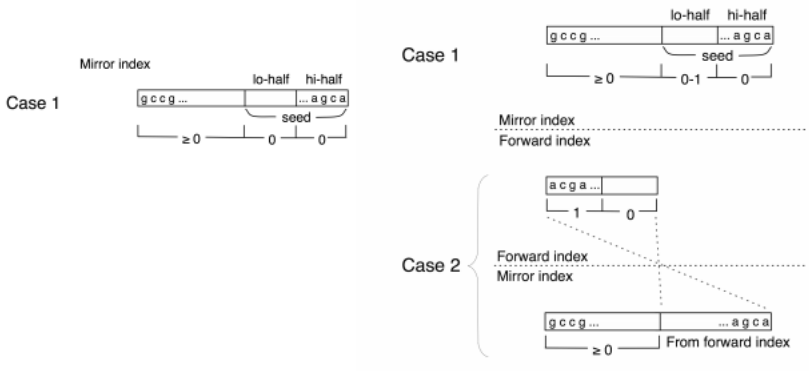


Figura 4.6: Casos explorados para la búsqueda de 0 y 1 *mismatch* usando la política MAQ sobre la secuencia "acga..gccg" [5].

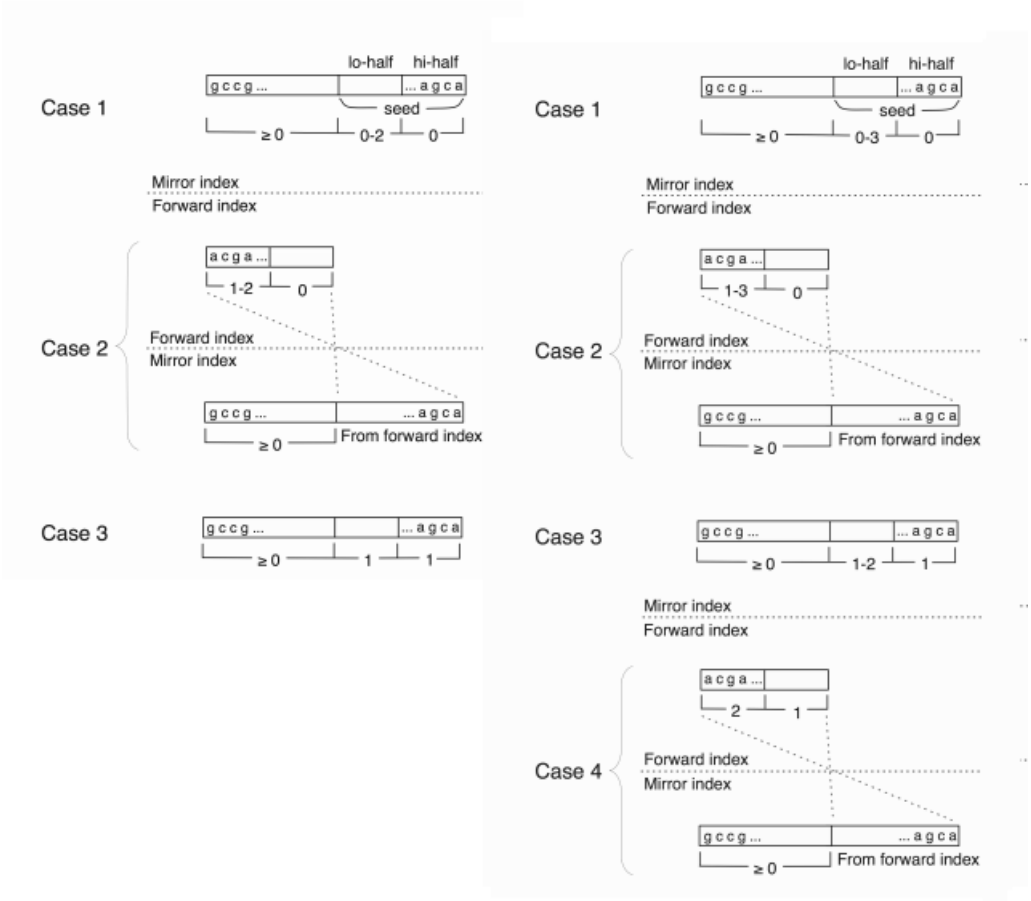


Figura 4.7: Casos explorados para la búsqueda de 2 y 3 *mismatches* usando la política MAQ sobre la secuencia "acga..gccg" [5].

Por lo tanto Bowtie es un alineador diseñado para funcionar de forma muy rápida en secuencias cortas con tres errores como máximo. Trata por separado cada caso de tal forma que optimiza cada uno de ellos.

4.3. Diseño

La implementación del Bowtie se ha realizado en C++, para poder usar ciertas comodidades que tiene este lenguaje, principalmente tratamiento de cadenas, pero casi la mayoría del código es C puro. Primero hay que definir la estructura Bowtie, que contiene:

- `forwardFmi`: Se corresponde con el *forward index*, que nos permite movernos de derecha a izquierda buscando la secuencia en el genoma.
- `mirrorFmi`: Se corresponde con el *mirror index*, que nos permite movernos de izquierda a derecha.
- `nNuc`: El número de nucleótidos del genoma. Tiene este valor guardado porque es necesario tenerlo en cuenta en algunas funciones internas.

El primer paso del algoritmo Bowtie es la generación de los FM-index forward y mirror. Para ello, Bowtie recibe la cadena del genoma de referencia. A esta cadena le añade el símbolo de

terminación y se crea el *forward index*. Después se invierte la cadena (dejando inalterado el '$') y se crea el *mirror index*.

Una vez creada esta estructura, Bowtie recibe secuencias y devuelve el mejor resultado posible de alineación, siempre que cumpla la política de alineación. La implementación de Bowtie proporcionada por sus autores [4], incluye diferentes modos de ejecución. La que se ha implementado se corresponde con la opción del Bowtie "- -best", que aunque es un poco más lenta, revisa todo el genoma en busca de la mejor alineación posible. Se ha optado por esta implementación porque es la que va a permitir comparar de una forma más coherente los diferentes alineadores.

Como se ha explicado antes, el número y el modo en el que se explorar los casos dependen del número de *mismatches* que permita el usuario al ejecutar el programa. La parte del códigos que implementa esta funcionalidad es la siguiente, contenida en la función inexactMatchBowtie:

```
switch (mismatch) {
    case 0:
        case1(bowtieFMI, patternReverse, read, 0, set);
        break;
    case 1:
        case1(bowtieFMI, patternReverse, read, 1, set);
        case2(bowtieFMI, pattern, read, 1, set);
        break;
    case 2:
        case1(bowtieFMI, patternReverse, read, 2, set);
        case2(bowtieFMI, pattern, read, 2, set);
        case3(bowtieFMI, patternReverse, read, 2, set);
        break;
    case 3:
        case1(bowtieFMI, patternReverse, read, 3, set);
        case2(bowtieFMI, pattern, read, 3, set);
        case3(bowtieFMI, patternReverse, read, 3, set);
        case4(bowtieFMI, pattern, read, 3, set);
        break;
```

Como se puede observar, los casos coinciden con los casos de las figuras 4.6 y 4.7, y cada caso recibe la secuencia o la secuencia inversa según vaya a usar el *forward* o el *mirror index*. La variable "set"va acumulando todas las alineaciones válidas en cada uno de los casos. Por último se ordenan todas ellas según el *quality score*, de tal forma que solo se devuelve al usuario el mínimo de ellas. Cabe recordar que el *quality score* es solo de las posiciones donde se realizó un *backtrack*, es decir, donde se producen los errores. Esto hace que si una secuencia tiene mayor calidad que otra, significa que los errores en esa secuencia son más improbables, por lo que es más díficil que se cometieran en esa cadena los *mismatches*.

Además de la función anterior, cabe mencionar la función que realiza la búsqueda inexacta. Esta función es recursiva y recibe un flag que le indica que tipo de búsqueda tiene que realizar. Puede ser exacta, que obligue a que tenga un error, que acepte hasta tres, etc.

Por último también se han creado las funciones para guardar y cargar la estructura del Bowtie, de tal forma que no sea necesario volver a calcular los índices, ahorrando bastante tiempo. La implementación detallada de las principales funciones se encuentra en el Anexo C.

5 BWA

5.1. Definición

Burrows Wheeler Alignment [24] (BWA) es una herramienta de alineamiento del genoma desarrollada por Heng Li y Richard Durbin en 2009. Su principal cualidad es que permite alinear secuencias cortas frente a un genoma de referencia, pero permitiendo que se puedan producir tanto errores como *gaps* (huecos) en la secuencia. Recordemos que Bowtie, solamente era capaz de alinear correctamente aquellas secuencias sinque no tenían huecos.

Como su nombre indica BWA, está basado en la transformada de Burrows-Wheeler y en el FM-Index. La idea general de este algoritmo es recorrer de arriba a abajo un árbol de prefijos mediante el FM-Index, teniendo una baja huella de memoria.

Para la búsqueda inexacta, BWA devuelve aquellas secuencias que tienen una distancia de edición menor que un valor umbral k. Además, al usar el FM-Index, todas las subsecuencias del genoma iguales se encuentran agrupadas en un mismo camino, de forma que solo son comparadas una vez.

5.2. Alineación de secuencias

5.2.1. Árbol de prefijos

Un árbol de prefijos de un cadena T, es un árbol donde cada arista hace referencia a un caracter y la cadena formada por la concatenación de los símbolos de las aristas desde las hojas hasta el nodo raíz representa un prefijo de T. Esta estructura es muy útil ya que podemos realizar una correspondencia entre el FM-Index y los diferentes *suffix arrays* (SA) con el árbol de prefijos. Para ello basta con crear el árbol de prefijos y rellenar los nodos con el SA correspondiente a la subcadena formada desde ese nodo hasta la raíz.

Como ejemplo, la figura 5.1 muestra el árbol de prefijos correspondiente a la cadena "$ACATGCA$" usada en los ejemplos de los capítulos anteriores. Los prefijos de esta cadena son:

$$\begin{cases} \text{“}A\text{”} \\ \text{“}AC\text{”} \\ \text{“}ACA\text{”} \\ \text{“}ACAT\text{”} \\ \text{“}ACATG\text{”} \\ \text{“}ACATGC\text{”} \\ \text{“}ACATGCA\text{”} \end{cases} \quad \text{Que se corresponden con los caminos desde las hojas a la raíz}$$

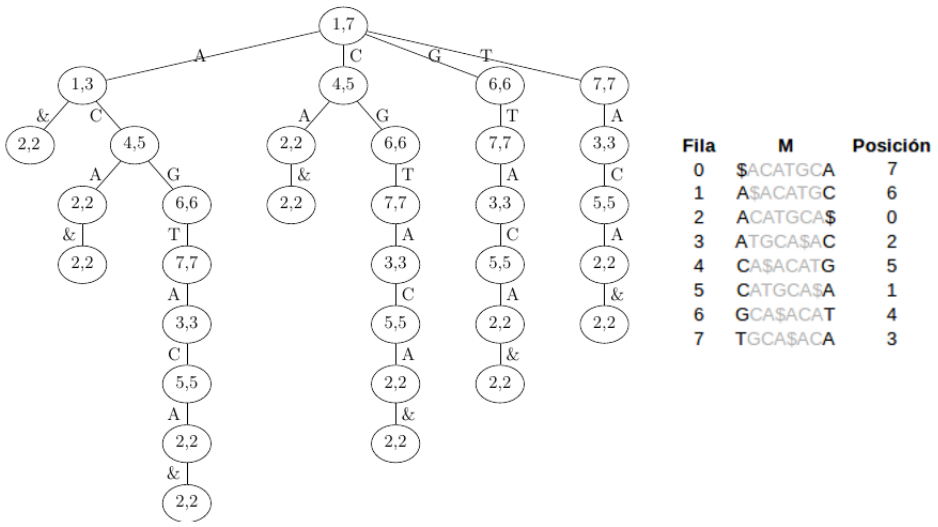


Figura 5.1: Árbol de prefijos de la cadena ”$ACATGCA$”. Cada nodo contiene el intervalo SA de la cadena formada por la concatenación de caracteres desde ese nodo hasta la raíz. El símbolo ’&’ hace referencia al principio de la cadena.

5.2.2. Búsqueda exacta e inexacta

El BWA se basa al igual que Bowtie en la búsqueda a partir de *backtracking* en una secuencia. Cuando no se permiten ni errores ni *gaps*, su búsqueda exacta coincide con el algoritmo descrito en el capítulo del FM-Index Sección 3.3.1.

Búsqueda inexacta

Siguiendo la notación empleada hasta el momento, sea W una secuencia que se quiere alinear con un genoma T. El algoritmo empleado por el BWA permite buscar todos los intervalos SA de aquellas cadenas de T que coincidan con W permitiendo hasta k diferencias, que pueden ser *gaps* o *mismatches*.

Para ello primero se necesita un array D donde $D[i]$ es el límite inferior del número de diferencias en $W[0, i]$. Si se divide la secuencia en pequeños fragmentos y se buscan en el genoma, todos aquellos que no alineen perfectamente producirán un error, y la suma de todos estos errores será el mínimo número de errores que necesite la secuencia original para alinearse. Cuanto mejor se estime D, menor será el espacio de búsqueda y más eficiente sera el algoritmo. La creación del array D se detalla en el Algoritmo 3.

```
z ← 0;
j ← 0;
for (i = 0 to |W| − 1) do
    if W[j, i] no es subcadena de T then
        z ← z + 1;
        j ← i + 1;
    end
    D(i) ← z ;
end
return D;
```

Algorithm 3: CALCULATED. Algoritmo que nos permite hallar el array D a partir de la secuencia W.

Para realizar este algoritmo de forma óptima, se va a hacer uso del *mirror index*, que nos permitía recorrer el genoma de izquierda a derecha. En el Algoritmo 3, se comprueba primero si el prefijo de W hasta i se encuentra en el genoma. Se empieza en un solo caracter y se va aumentando el prefijo hasta que éste no esté en el genoma. Si se produce esta situación, indica que se necesita un error en ese prefijo para que haya alineación. Después se repite esto mismo suponiendo que la cadena empieza en el siguiente caracter donde se produjo el error.Cuando termina el algoritmo, tendremos un array que indicará cuantos errores se necesitan como mínimo para alinear la secuencia.

Para realizar la búsqueda inexacta, primero se calculan los índices *forward* y *mirror*, para los cuales se obtienen las estructuras $Occ(c, i)$ i $C(c)$ definidos en el capítulo 3.2.2. Creado estos índices, se define la función de búsqueda inexacta, la cual primero calcula el array D y después llama a la función de búsqueda inexacta recurrente.

La función recurrente, descrita en el Algoritmo 5 recibe diferentes parámetros. La secuencia que se busca, la posición en la secuencia, el número de errores/huecos k definido por el usuario, y por último sp y ep que son los valores del intervalo SA.

```
if z < D(i) then
    return ∅
end
if i < 0 then
    return [sp, ep]
end
I ← ∅;        Se inicializa el conjunto vacío;
[1]I ← I ∪ INEXRECUR(W, i − 1, z − 1, sp, ep);
foreach b ∈ {A, C, G, T} do
    sp ← C(b) + Occ(b, sp − 1) + 1;
    lep ← C(b) + Occ(b, ep);
    if k ≤ l then
        [2]I ← I ∪ INEXRECUR(W, i, z − 1, sp, ep);
        if b = W[i] then
            [3]I ← I ∪ INEXRECUR(W, i − 1, z, sp, ep);
        else
            [4]I ← I ∪ INEXRECUR(W, i − 1, z − 1, sp, ep);
        end
    end
end
return I
```

Algorithm 4: INEXRECUR. Algoritmo realiza la búsqueda inexacta del BWA a partir de la recursión.

Va almacenando en el conjunto I todos intervalos de SA que cumplen las restricciones. Así, después de cada llamada recursiva, ésta devuelve un conjunto de índices que se une con el de la función. Si el número de errores que se permiten en algún momento de la ejecución es menor que el límite inferior proporcionado por D, entonces esa camino del *backtracking* no se puede encontrar en el genoma. Por el contrario, si se llega al principio de la secuencia es que sí se ha encontrado una alineación, por lo que se devuelven los índices del SA correspondiente.

Las llamadas recursivas dentro de la función cubren cada uno de los casos que se pueden producir. En el Algoritmo 5, las llamadas que contienen un ^{1} o ^{2} se corresponden con casos de *gaps*.

- Primera llamada a la recursión ^{1}. Como se puede observar, se llama a la función disminuyendo en 1 tanto la posición como el numero de errores/huecos. Este caso se corresponde con un *insertion* en la secuencia, es decir; se supone que al secuenciar se añadió un nucleótido en esta posición. No se acualizan *sp* y *ep*, ya que no nos movemos en el genoma.
- Segunda llamada a la recursión ^{2}, donde se resta 1 al número de errores/huecos pero la posición se mantiene intacta. Este caso se corresponde con un *deletion*. Si es así, debemos avanzar en el genoma para cada nucleótido, pero sin avanzar en la secuencia. Como está dentro del bucle, se realizará una llamada por cada nucleótido actualizando los índices *sp* y *ep*.
- Tercera llamada a la recursión ^{3}. Se correponde con el caso de una concordancia. El nucleótido de la secuencia coincide con el del genoma, por lo que se continúa la búsqueda actualizando los índices *sp* y *ep* y disminuyendo en uno la posición en la secuencia.
- Cuarta llamada a la recursión ^{4}. Se corresponde con un mismatch. Se calcula el k y l del caracter del mismatch y se sigue avanzando en la búsqueda, restando 1 a la posición i y al número máximo de errores/huecos.

En este algoritmo se prueban todas las posibles combinaciones que se puedan dar en el genoma teniendo en cuenta la restricción impuesta por el usuario del máximo k de errores/huecos, para hallar siempre la mejor. Además se usa parte de la política de poda del MAQ. Cada vez que se produce un error o hueco, se suma el *quality score* de esa posición teniendo en cuenta que, si supera el umbral 70, se desecha ese camino.

5.3. Diseño

Las funciones del BWA se han implementado en C++ para un mayor control de memoria. La estructura principal del BWA tiene los siguientes campos:

- `forwardFmi`: Se corresponde con el *forward index*, que nos permite movernos de derecha a izquierda buscando la secuencia en el genoma.
- `mirrorFmi`: Se corresponde con el *mirror index*, que nos permite movernos de izquierda a derecha buscando la secuencia en el genoma.
- `nNuc`: El número de nucleótidos del genoma. Tiene este valor guardado porque es necesario tenerlo en cuenta en algunas funciones internas.
- `gap`: Es un boolean que indica si el BWA tiene que realizar la búsqueda solo de mismatches o también incluir *gaps* (huecos).

Además se ha llevado a cabo la implementación de la función para calcular el array D y la búsqueda inexacta recurrente. Las funciones más destacadas se encuentran en el Anexo D.

6
BWT-SW

6.1. Definición

El BWT-SW (Burrows-Wheeler Transform Smith Waterman)[25] es una herramienta que explota las posibilidades de la BWT unida a la programación dinámica, creando una herramienta capaz encontrar todas las alineaciones locales de secuencias largas. BWA y Bowtie son herramientas diseñadas para búscar secuencias cortas de (35-100bp) en el genoma, sin embargo el BWT-SW está diseñado para secuencias mayores.

Para encontrar todas las alineaciones BWT-SW se hace uso del algoritmo de Smith and Waterman(SW) de programación dinámica. Esta aproximación, es demasiado costosa en tiempo de ejecución, ya que requeriría recorrerse todo el genoma para encontrar la mejor alineación. Un ejemplo concreto; el algoritmo de SW tarda 15h en alinear una sola secuencia de 1000 caracteres en el genoma humano. Por ello, los creadores del BWT-SW aprovecharon las propiedades de la BWT y del FM-Index para implementar un algoritmo más óptimo en tiempo.

6.2. Indexado y programación dinámica

Sea T una cadena de n caracteres (el genoma) y sea W una secuencia de m caracteres (*read*). Una aproximación bastante "ingenua" para buscar todos los alineamientos locales sería examinar todas las subcadenas de T de longitud $c \times m$ (donde c es una constante que depende del modelo) e intentar alinearlas con W. Este método implicaría revisar todo el genoma en busca de encontrar una subcadena que satisfaga un mínimo de calidad, lo cual tiene un orden de tiempo de $O(n \times m \times (c \times m))$ resultante de la longitud del genoma y la matriz de programación dinámica.

6.2.1. Algoritmo del BWT-SW

BWT-SW utiliza un indexado del genoma y otro de la secuencia a buscar, de tal forma que se puedan comparar al mismo tiempo aquellas subcadenas iguales. Hay que recordar que el FM-Index representa un árbol de prefijos, y el movernos con el *backtracking* en el FM-Index se

corresponde con recorrer de arriba abajo el árbol de prefijos. Además el árbol que representa la secuencia W se ve luego compactado en un grafo acíclico dirigido DAWG (Direct Acyclic Word Graph). El DAWG junto con el indexado del genoma, servirán de base para llevar a cabo un algoritmo de programación dinámica que encuentre las posibles alineaciones locales de la secuencia W en T, tal y como es esplicará en la Sección 6.2.3.

La versión de los autores del BWT-SW, no usa la estructura del DAWG para la comparación entre el árbol del genoma y las secuencias. El DAWG se usa en el alineador BWA-SW de los autores de BWA cuyo algoritmo está basado en el BWT-SW y optimiza sus tiempos de ejecución [26]. Por lo tanto, aunque se use el algoritmo y la poda del BWT-SW, su forma de implementación es un híbrido entre los dos: BWA-SW y BWT-SW.

Al tratar de alinear el genoma con la secuencia, se pueden producir diferentes casos:

- Si $T[i] == W[i]$ se produce una coincidencia, es decir, el par está alineado.
- Si $T[i] \neq W[i]$ se produce un *mismatch*.
- Si $T[i]$ se alinea con un espacio en la secuencia W, es decir, que se ha producido un *deletion*.
- Si $W[i]$ se alinea con un espacio en T,se ha producido un *insertion* en la secuencia.

Solo el primero de todos tiene un valor positivo a. El *mismatch* tiene un valor negativo b mientras que los otros dos casos se corresponden con *gaps*, y tienen un valor negativo proporcional a la longitud del hueco $g+rs$ siendo r la longitud del *gap* y con g y s negativos. Los valores habituales de estas variables son $a = 1$, $b = -3$, g$=-5$ y $s = -2$ [25].

6.2.2. Árbol de prefijos y DAWG

Como se definió en la sección 5.2.1 un árbol de prefijos de una cadena T es una estructura en la cual cada arista hace referencia a un caracter y la concatenación de todos los caracteres desde una hoja hasta la raíz se corresponde con un prefijo de T. La concatenación de los símbolos desde cualquier nodo hasta la raíz siempre es una subcadena de T, lo que nos permite definir el valor del nodo como el intervalo del SA que contiene esa subcadena. Se puede dar el caso en el que dos nodos diferentes tengan el mismo intervalo, ya que pueden representar los mismos prefijos pero de distinta longitud. En la figura 6.1 se observa como hay bastantes nodos que contienen el mismo intervalo.

El DAWG se crea a partir del árbol de prefijos, colapsando aquellos nodos que tengan el mismo intervalo. Además, cada nodo, al haber sido colapsado, puede llegar a representar diferentes subcadenas con el mismo prefijo.

6.2.3. Programación dinámica

Consideremos un árbol de prefijos asociado al genoma T y un DAWG que proviene de colapsar el árbol del prefijos de la secuencia W. El algoritmo de programación dinámica para la mejor alineación de W a T es el siguiente. Tomamos G_{uv}, D_{uv} y I_{uv} tres matrices, donde u hace referencia a los nodos del árbol del genoma y v a los nodos del DAWG de la secuencia. La matriz G_{uv} nos va a almacenar el mejor *score* de entre todas las posibles cadenas representadas en v con respecto al nodo del genoma u. Por otra parte las matrices D_{uv}, I_{uv} van a tener el mejor *score* posible teniendo en cuenta que se ha producido un *deletion* o un *insertion* en el nodo v. Como los nodos v del DAWG pueden representar varias cadenas, cada una de ellas tendrá su propio *score* en cada matriz y se cogerá siempre el mejor.

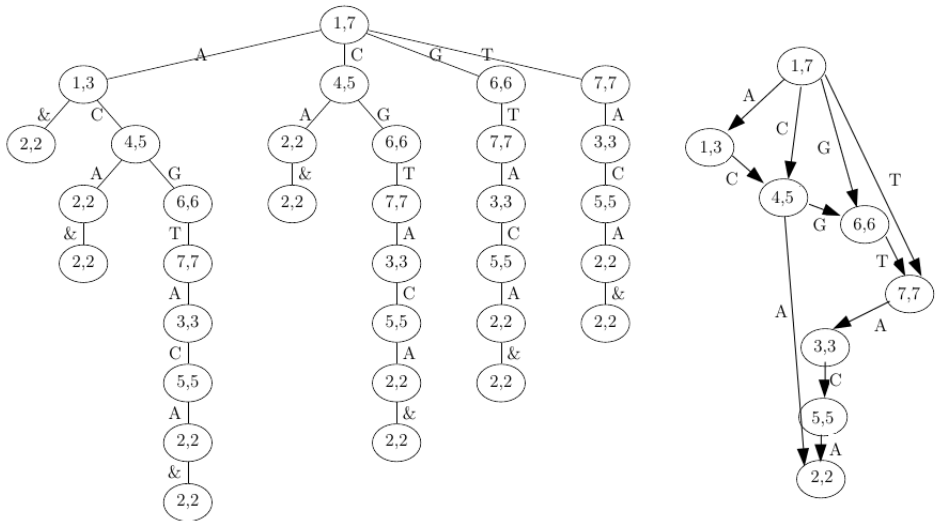


Figura 6.1: Árbol de prefijos de la cadena "$ACATGCA$" con el correspondiente DAWG.

Los valores de inicialización del algoritmo de programación dinámica son:

$G_{u_0 v} = 0$, siendo u_0 el nodo raíz del genoma.

$G_{uv_0} = g + is$, siendo v_0 el nodo raíz del DAWG de la secuencia, $u \neq u_0$ e i la profundidad en el árbol del genoma

$D_{u_0 v} = -\infty$

$I_{uv_0} = -\infty$

Luego, dado un nodo v, se calcula para cada uno de sus padres v':

$I_{uv|v'} = \text{máx}\{I_{uv'}, G_{uv'} + g\} + s$

$D_{uv|v'} = \text{máx}\{D_{u'v}, G_{u'v} + g)\} + s$

$G_{uv|v'} = \text{máx}\{G_{u'v'} + S(u', u; v', v), I_{u'v'}, D_{u'v'}, 0\}$

donde u'es el padre de u en T y la función $S(u', u; v', v)$ devuelve a o b dependiendo de si el caracter en la arista (u', u) coincide o no con el de (v', v)respectivamente. Hay que recordar que el valor g es el coste por empezar un *gap* mientras que s es el coste por cada unidad de longitud del *gap*. Por lo tanto, cuando se halla el valor de $I_{uv|v'}$ y $D_{uv|v'}$ se tiene en cuenta que si cogemos el valor de G, hay que añadirle el coste por iniciar un nuevo *gap* más luego el coste por una unidad de longitud, mientras que si se coge el valor de D o I, solo se añade el coste por unidad ya que se está extendiendo el hueco, no creándolo. Gracias a esto, podemos hallar el nodo v^* que es el padre de v que tenga un mayor valor, es decir, una mayor semejanza al genoma y calcular G_{uv}, D_{uv} y I_{uv} con la fórmula:

$v^* = argmax(G_{uv|v'} : v' \in pred(v))$

$(G_{uv}, D_{uv}, I_{uv}) = (G_{uv|v^*}, D_{uv|v^*}, I_{uv|v^*})$

Después de inicializar las matrices, se rellena cada valor teniendo en cuenta siempre los valores de sus predecesores. Debido a ello, para poder hallar el valor de un par de nodos, sus padres deben de haber sido visitados. En el caso del árbol de prefijos del genoma, esto no es ningún problema, ya que se parte de la raíz hacia las hojas y cada nodo solo tiene un padre, luego para llegar al hijo tienes que pasar obligatoriamente por el padre. Sin embargo en el DAWG esto no es así. Al haber colapsado varios nodos en uno, pueden existir nodos con varios padres, por lo que tendremos que recorrer este grafo en postorden inverso [27], para asegurarnos que todos los padres están visitados antes de llegar al hijo.

El algoritmo del BWT-SW recorre los estructuras en un bucle anidado. En el bucle exterior se encuentra el árbol del genoma que se recorre con una búsqueda en profundidad del árbol, mientras que en el bucle interior se recorre el DAWG en postorden inverso. Esto tiene una ventaja de memoria, ya que cada vez que bajamos un nivel del árbol de prefijos, se corresponde con añadir una fila a nuestra matriz de programación dinámica; y cuando subimos en el árbol, podemos desechar esa fila y liberar su memoria. Si recorriérmos el árbol al revés, en las filas tendríamos los nodos del DAWG y en las columnas el genoma.

6.2.4. Adaptación de la programación dinámica a la poda

El algoritmo descrito en la sección anterior, compara todas las posibles combinaciones entre el árbol y la secuencia. Sin embargo, gran parte de esas comparaciones no son necesarias, por lo que se va a implementar una poda para no recorrerlas. Para la realización de la poda vamos a definir primero unos conceptos y teoremas:

Definición 1 *Alineación sin significado. Sea A un alineamiento entre la subcadena $X = T[h..i]$ de T e $Y = W[j..k]$ una subcadena de la secuencia W. Si A alinea el prefijo $X' = X[h..h']$ con el prefijo $Y' = Y[j..j']$ con un score menor o igual que cero se dice que A es una alineación sin significado. Si no, se dice que es un alineamiento con significado.*

Lema 1 *Suponemos que A es un alineamiento sin significado entre un subcadena $X[h..i]$ e $Y[j..k]$ con un score positivo C. Entonces existe un alineamiento con significado entre el sufijo $X' = X[i'..i]$ de X y el sufijo $Y' = Y[k'..k]$ de Y con un score de al menos C.*

Las tablas de programación dinámica, nunca van a usar valores menores o iguales a 0, luego podemos modificar la programación dinámica para computar solo el mejor score posible entre los alineamientos con significado. Las ecuaciones quedarían:

$$D_{uv} = \begin{cases} \text{máx}\{D_{u'} + s, G_{u'v} + g + s\}, & \text{si } D_{u'v} > 0 \text{ y } G_{u'v} > 0 \\ D_{u'v} + s, & \text{si solo } D_{u'v} > 0 \\ G_{u'v} + g + s & \text{si solo } G_{u'v} > 0 \\ \infty, & \text{en otro caso} \end{cases}$$

$$I_{uv} = \begin{cases} \text{máx}\{I_{uv^*} + s, G_{uv^*} + g + s\}, & \text{si } I_{uv^*} > 0 \text{ y } G_{uv^*} > 0 \\ I_{uv^*} + s, & \text{si solo } I_{uv^*} > 0 \\ G_{uv^*} + g + s & \text{si solo } G_{uv^*} > 0 \\ \infty, & \text{en otro caso} \end{cases}$$

$$G'_{uv} = \begin{cases} \text{máx}\{G_{u'v^*} + S(u', u; v^*, v)\}, & \text{si } G_{u'v^*} > 0 \\ \infty, & \text{en otro caso} \end{cases}$$

$$G_{uv} = \text{máx}(G'_{uv}, D_{uv}, I_{uv})$$

Por último el programa guarda el par de nodos que contienen el mayor *score*. Así, gracias al nodo del árbol de prefijos del genoma, podemos usar el intervalo *sp* y *ep* para hallar la posición en el genoma de la secuencia.

6.3. Diseño

Como el resto de secciones, la implementación ha sido realizada en C++. La estructura básica contiene los siguientes campos:

- `fmi`: Se corresponde con el FM-Index del genoma. Es el árbol de prefijos que vamos a recorrer con una búsqueda en profundidad.
- `nNuc`: Número de nucleótidos del genoma.

La estructura es bastante sencilla, pero se ve complementada con el tratamiento e indexado que reciben las secuencias, tal y como se detalla en la siguiente sección.

6.3.1. Implementación de la búsqueda y programación dinámica

Lo primero de todo es crear el DAWG que represente la secuencia W. Para ello primero se crea una estructura `Node`, que será cada uno de los nodos del grafo y contendrá la siguiente información:

- `childs`: Es un puntero que guarda todos los nodos descendientes de él.
- `NChilds`: Número de descendientes del nodo.
- `SA`: Valor del nodo. Hace referencia a los dos índices del intervalo del Suffix Array.
- `word`: Caracter de las aristas que llevan a ese nodo. En único, ya que todos los padres de este nodo, para llegar a este hijo, lo hacen a través del mismo caracter.
- `vis`: En un flag que nos permite saber si el nodo ha sido visitado.
- `fathers`: Es un puntero que nos indica la posición que ocupan sus padres en el array del DAWG. Estos valores se explican en la estructura del DAWG.
- `Nfathers`: Número de padres.

Una vez definido cada uno de los campos que contienen los nodos, pasamos a definir las campos propios del DAWG:

- `root`: Se corresponde con un puntero que apunta a la nodo raíz del grafo. Desde aquí se puede desplegar el grafo.
- `arrayNodes`: Array que contiene todos los nodos del grafo en postorden inverso. De esta forma, para cualquier nodo en una posición i en este array, sus padres están en una posición j con $0 \leq j < i$. Si recordamos, en la estructura nodal, el array de padres no se apuntan a los nodos, si no que es un puntero a enteros que se corresponde con la posición en este array.
- `length`: Número de nodos del DAWG.

Para crear el DAWG, primero se va recorriendo todo el árbol de la secuencia generando primero su FM-Index. Según se va recorriendo se van obteniendo los distintos SA y creando los nodos, añadiendo los padres y los hijos que se vayan creando. Una vez hecho esto, se recorre el grafo en postorden guardando en una pila las direcciones de memoria de los nodos. Así, al sacarlos de la pila los podemos guardar en el array del DAWG el orden inverso del que se guardaron, es decir, postorden inverso.

Una vez creado el DAWG y con el FM-Index del genoma, se puede empezar la programación dinámica. Para ello se inicializan las matrices y se van rellenando según se recorren los árboles. Otra ventaja de haber guardado los índices en el array de los padres de un nodo en vez de

guardar su posición de memoria es que, al mirar los padres de un nodo, directamente tenemos la columna que hace referencia a cada padre. Por otra parte, el padre de un nodo del árbol de prefijos, es la fila inmediatamente superior.

Mientras se recorre el bucle anidado, se guarda aquel SA del genoma cuyo nodo tenga el mayor *score*. De esta forma, cuando termina, tenemos la posición de la alineación. El código de las funciones más destacadas del BWT-SW se encuentran en el Anexo E.

7
Pruebas y Resultados

7.1. Pruebas unitarias

Para poder llevar a cabo la comparativa de los alineadores, primero se han tenido que realizar diferentes tipos de pruebas para asegurar su correcto funcionamiento. Se han llevado a cabo pruebas unitarias de caja blanca y pruebas unitarias de caja negra. Además se ha utilizado valgrind en todos los módulos para comprobar la ausencia de errores y la buena liberación de memoria. La memoria es un punto crítico en estos algoritmos, ya que un mal manejo de ella o una mala liberación puede producir que se consuma toda la memoria del ordenador. Por ejemplo, en el algoritmo BWT-SW si no se libera cada fila de las matrices cuando se asciende en el árbol de prefijos, se podrían llegar a reservar 4^m nodos en el peor de los casos siendo m la longitud de la secuencia. Teniendo en cuenta que se realizan pruebas con $m = 100$ se observa que puede ser totalmente inmanejable para un ordenador independientemente de lo que ocupe el nodo.

7.1.1. Pruebas unitarias de caja blanca

Este tipo de pruebas han sido las más significativas y se realizan después de la implementación de cada módulo. La parte principal y más compleja de este trabajo ha sido la implementación de los distintos algoritmos. Para observar el correcto funcionamiento de éstos, los más indicado son este tipo de pruebas. Las pruebas de caja blanca utilizadas han sido en genomas generados a mano de pequeña longitud, y secuencias preparadas para probar cada uno de los casos:

- FM-Index. Las pruebas consistían en generar un "genoma"pequeño, de unos 10 nucleótidos y crear a mano su FM-Index. Una vez hecho esto, se iba comprobando que la estructura se iba generando bien y sin errores.
- Bowtie: Este alineador funciona a través de una serie de casos que se han de ir ejecutando de forma secuencial, además de ir alternando los diferentes tipos de índices. Se han creado pruebas de caja blanca para verificar el funcionamiento de cada caso dependiendo del número de errores que introdujera el usuario.
- BWA. En este alineador, se han creado pruebas de caja blanca para observar los diferentes caminos producidos por la recursión.

- BWT-SW. En él, las pruebas de caja blanca se realizaron observando los valores de las matrices mientras se ejecutaba el algoritmo.

7.1.2. Pruebas de caja negra

Por último se han aplicado pruebas de caja negra fabricando pequeños ejemplos y comparando la salida con el resultado esperado

7.2. Comparación de alineadores

Para llevar a cabo la comparativa entre los tres alineadores implementados, se ha utilizado un simulador de secuencias NGS. La ventaja que ofrece el uso de datos simulados en lugar de datos reales, es que se dispone de información sobre la procedencia real de la secuencia; es decir, la posición del genoma de referencia de la que se ha obtenido. De esta forma, se puede cuantificar el rendimiento del alineador en base al porcentaje de aciertos que lleva a cabo cada uno y a su tiempo de ejecución.

Para la generación de las secuencias simuladas se ha utilizado la herramienta ART, capaz de simular secuencias NGS de las tres principales plataformas comerciales: 454, Illumina, y SOLiD [28]. La elección de esta herramienta se justifica por la facilidad que se tiene para controlar los dos tipos de errores en la secuenciación: *mismatches* y *indels*. Se han generado simulaciones de lecturas del genoma utilizando para los *quality scores* distribuciones empíricas obtenidas de la tecnología Illumina y el sistema de secuenciado MiSeq v3. Se ha escogido este sistema puesto que es el que permitía explorar un rango mayor de longitudes de secuencia (hasta 250 nucleótidos por secuencia), lo cual es especialmente útil para comparar alineadores de lecturas cortas (Bowtie y BWA) frente a alineadores de lecturas largas (BWT-SW). Por un lado, para controlar el número de *mismatches*, esta herramienta ofrece un parámetro (*mismatch shift*) que permite desplazar los *quality scores* de cada nucleótido en cada secuencia un determinado número entero (positivo o negativo). Recordando que estos *quality scores* estaban directamente relacionados con la probabilidad de error en la secuenciación, este parámetro permite controlar el grado de mutaciones en la simulación: cuanto mayor sea el valor de los quality scores, menor será la probabilidad de mutación. Por otro lado los *gaps* se controlan mediante una distribución binomial $B(m, p)$ con m la longitud de la secuencia y $p = indel \times 0.005$, siendo el parámetro *indel* el que se ha utilizado para controlar esta probabilidad. Para cada configuración de los parámetros *mismatch* e *indel*, se han generado entre 100 y 150 secuencias. Este número viene controlado por un parámetro que determina el grado de cobertura del genoma, y , por tanto, es dependiente de la longitud de las secuencias, no siendo sencillo su ajuste para obtener en todos los casos el mismo número de secuencias. En cualquier caso, todos los resultados que se muestran a continuación se han normalizado por el número de lecturas, y no se ven afectados por esta variabilidad.

El parámetro *mismatch shift* se ha barrido en el rango de valores -14, -18, -16, ..., 30, mientras que el parámetro *indel* ha tomado valores en $indel = 0, 1, 2, ..., 10$. También se ha modificado la longitud de los reads para tomar los siguientes valores 50, 80, 100, 150, 200, 250. Estos rangos de valores para los parámetros se han establecido con el fin de obtener diversas casuísticas en la generación de secuencias tal y como muestra el figura 7.1. Como puede observarse, para valores del parámetro *indel* igual a 0, no existen inserciones ni borrados, y valores muy altos de *mismatch shift* apenas generan permutaciones, puesto que los quality scores asociados a cada nucléotido serán muy altos (poca probabilidad de error).

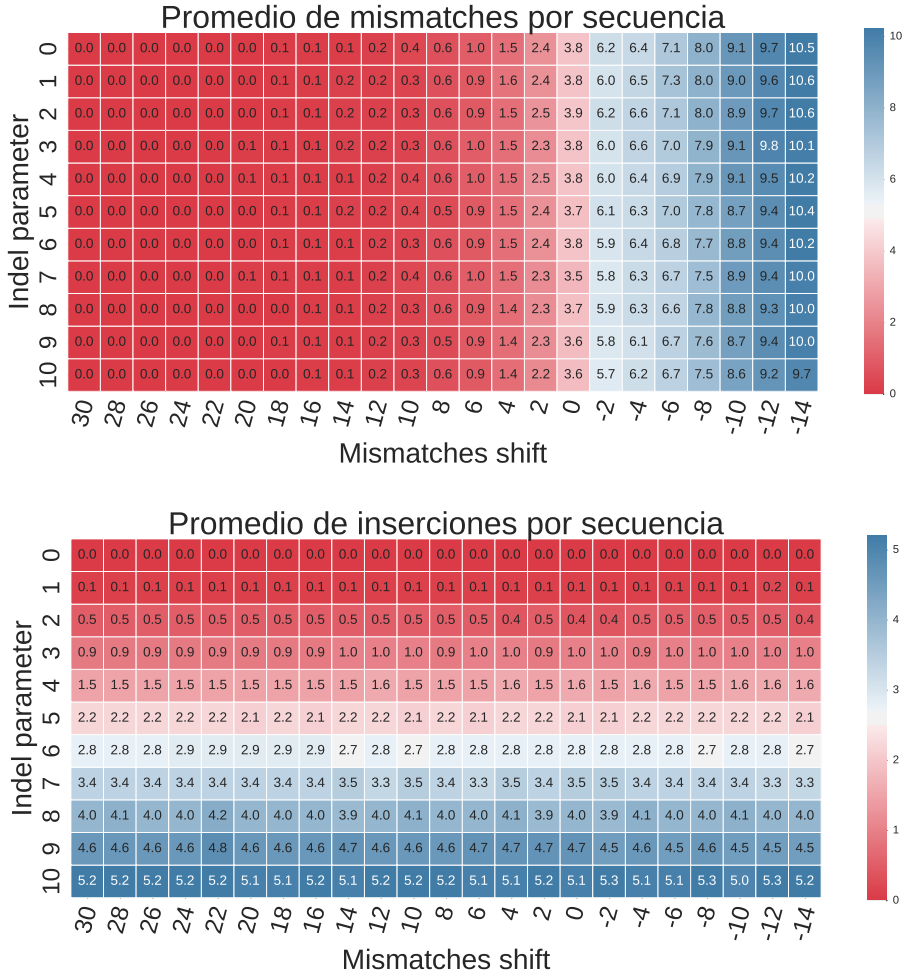


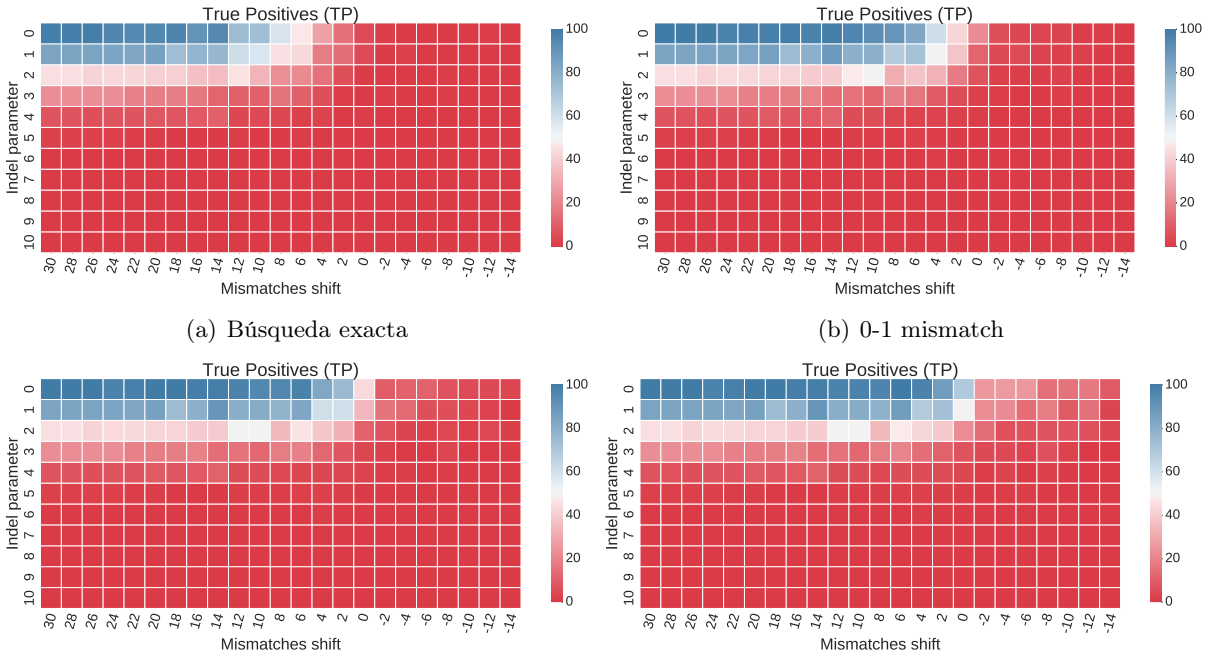
Figura 7.1: Tablas con la media de *mismatches* e inserciones en las secuencias según se varían los parámetros *mismatch shift* e *indel*.

7.2.1. Porcentajes de acierto

En esta sección se van a comparar los alineadores por su tasa de acierto (*True Positives*) dependiendo de los parámetros que controlan los *mismatches* y los *indels* (o *gaps*). Para realizar una comparación justa, se ha fijado la longitud de las secuencias a 100 nucleótidos, valor que está entre medias de secuencias cortas y largas para no beneficiar a ninguno. Se ha ejecutado Bowtie cambiando el número máximo de errores que recibe, desde cero hasta tres, pudiéndose observar en la figura 7.2 el porcentaje de acierto de cada uno.

Si recordamos la figura anterior, se producen de 0 a 3 *mismatches* cuando el parámetro *mismatches shift* se encuentra entre los valores 0-30. Además se observa en la figura que el parámetro *indel* no tiene ningún efecto sobre el número de errores. En la comparativa de la figura 7.2, se muestra como, según se van permitiendo más errores a Bowtie, aumenta la tasa de acierto. La figura (a) se corresponde con una búsqueda exacta, por lo que cuando las secuencias tienen 1 error de media (*mismatches shift* = 6) no la alinearán bien. Sin embargo, si comparamos este caso con el (d), donde se permite a Bowtie un máximo de tres errores se observa que empieza a fallar cuando el parámetro *mismatches shift* vale 0 (que se corresponde con 3 errores según la tabla).

También cabe destacar que Bowtie no es capaz de tratar *gaps*, de ahí que no se produzcan variaciones significativas entre las cuatro imágenes cuando se mueve el parámetro de *indel*.



(a) Búsqueda exacta

(b) 0-1 mismatch

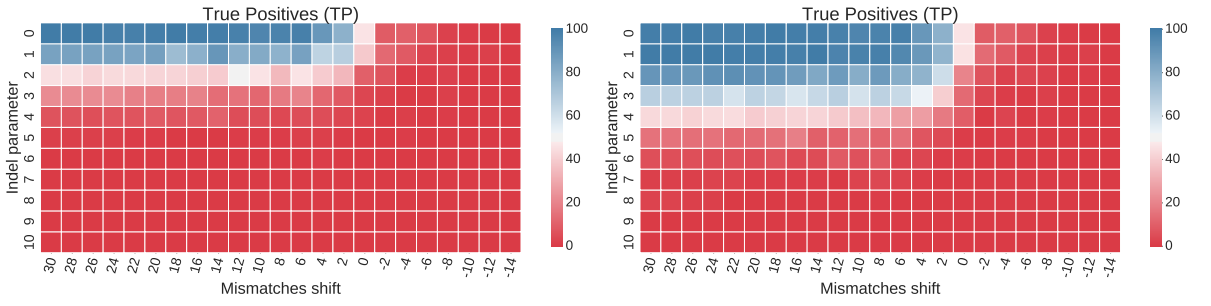
(c) 0-2 mismatches

(d) 0-3 mismatches

Figura 7.2: Tasa de acierto del alineador Bowtie con secuencias de longitud 100

Una vez observado el comportamiento del Bowtie, se ha comparado BWA. Este alineador nos permite ejecutar dos tipos de búsqueda: solo *mismatches* o incluyendo además huecos. En el primer caso, los resultados son muy similares a los del Bowtie (por ejemplo, veáse figura 7.2 (d) versus figura 7.3 (a)). Al no permitir *gaps*, sus tasas de acierto coinciden ya que ambos revisan el genoma en busca de la posición donde se producen menos errores; y en caso de igualdad, usan la política MAQ para decidir a través de los *quality score*(Sección 4.2.2).

En la figura 7.3 se han comparado el BWA para tres o menos mismatches con huecos y sin ellos. Se aprecia en la figura como el alineador con *gaps* sí se ve afectado al modificar el parámetro *indel*. Su tasa de acierto es mejor, ya que es capaz de detectar *insertions* y *deletions* en las secuencias aunque eso recaiga en un aumento del tiempo computacional del que se hablará en la siguiente sección.



(a) Sin gaps

(b) Con gaps

Figura 7.3: Tasa de acierto del alineador BWA con 0-3 mismatches y con secuencias de longitud 100

Por último falta añadir a la comparativa el alineador BWT-SW. Al contrario que los otros dos alineadores, BWT-SW no posee restricciones ni en el número de errores ni de *gaps*, ya que busca la mejor alineación local en el genoma, independientemente de su distancia de edición. En la figura 7.4 se muestra la tasa de acierto de este alineador. BWT-SW es mucho más robusto

que los dos anteriores, ya que tiene una tasa de acierto bastante alta para la mayor parte de las combinaciones de parámetros. Cabe destacar que, si sólo nos fijamos en los *mismatches*, el BWT-SW empieza a fallar cuando el parámetro *mismatch shift* es igual a 0; lo mismo que pasaba con Bowtie y BWA, la diferencia es que el BWT-SW permite una mejor búsqueda de *gaps*.

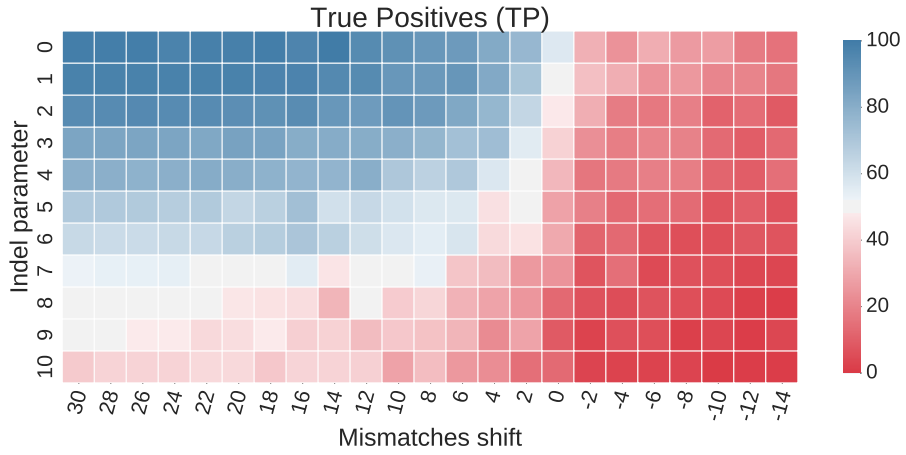


Figura 7.4: Tasa de acierto del alineador BWT-SW con secuencias de longitud 100

7.2.2. Tiempo de ejecución

En esta sección se va a tratar la diferencia del tiempo de ejecución de cada uno de los alineadores, dependiendo del tipo de secuencias.

En la figura 7.5, se observan los tiempos de ejecución de todos los alineadores utilizados. Según aumenta la longitud, es lógico que aumente el tiempo de alineación por secuencia, ya que tiene que explorar más casos. En esta gráfica se dejaron fijos los dos parámetros *mismatches shift* e *indel*.

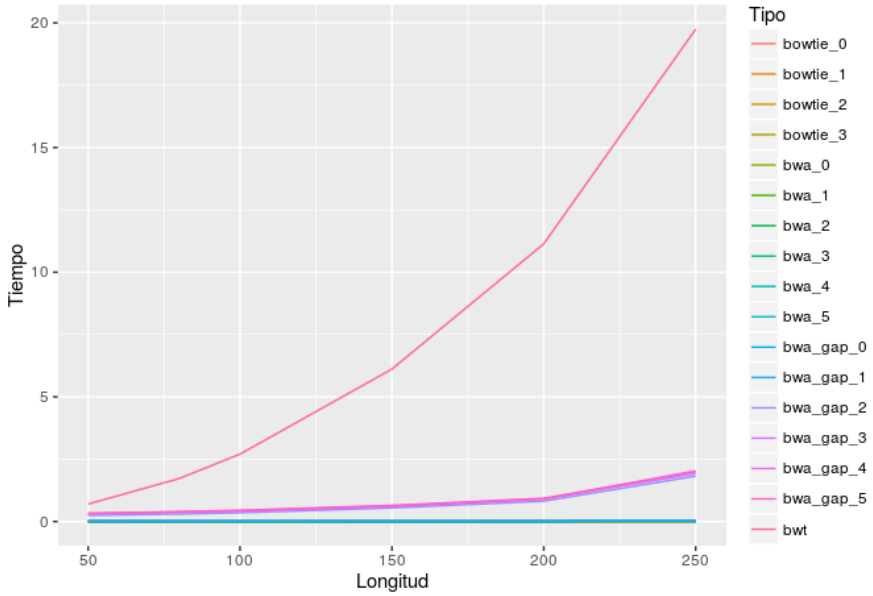


Figura 7.5: Tiempo de ejecución en la alineación de una secuencia según su longitud

El BWT-SW es el que más tiempo tarda en alinear una secuencia, debido a que hace una

exploración más exhaustiva. Por otra parte, el BWA con *gaps* también tiene un tiempo bastante mayor que el Bowtie o BWA sin huecos ya que realiza más llamadas recursivas (Sección 5.2.2).

Con estos datos se podría pensar que el BWT-SW es mucho más lento que el BWA con *gaps*, sin embargo no se está teniendo en cuenta que el BWA impone un umbral de 70 en la suma de los *quality score* de las posiciones incorrectas (polẗitica Maq-like) que, cuando es superado, descarta esa secuencia. Como las pruebas se hicieron con aproximadamente 3 *mismatches* por secuencia (figura 7.1), los *quality score* no era muy bajos y facilmente se superaba el umbral, reduciendo el tiempo de ejecución. Sin embargo, se probó el BWA con *gaps* en secuencias con muchos errores donde se observaba que al aumentar la longitud, superaba con creces el tiempo del BWT-SW.

Se ha comparado también como afecta al tiempo de ejecución el número de *mismatches* que posee la secuencia. En la figura 7.6 se ha representado el logaritmo del tiempo de ejecución según se varía el parámetro *mismatch shift*. Se ha usado el logaritmo del tiempo, para igualar un poco la escala y que sea más visual la comparación entre alineadores. No se ha incluido el BWA con *gaps* porque no producía ninguna diferencia con respecto a la versión de BWA sin *gaps*.

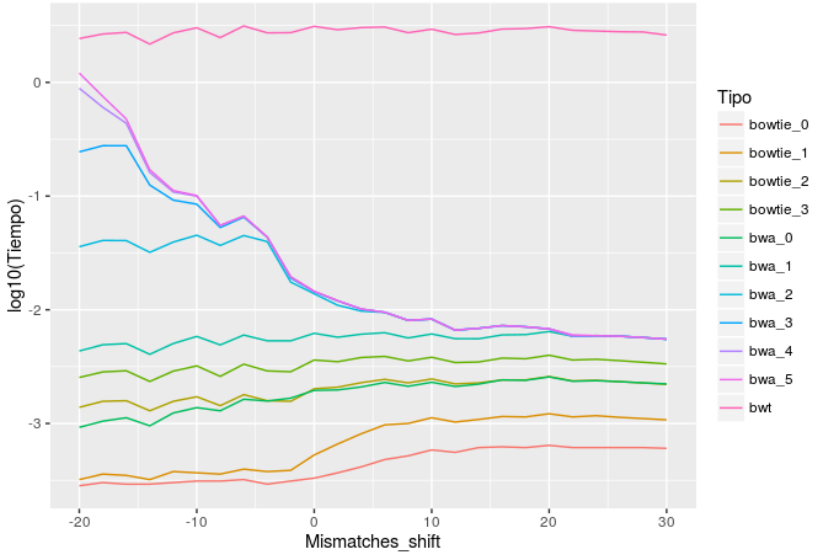


Figura 7.6: Tiempo de ejecución en la alineación de una secuencia de 100 nucleótidos según el número de errores. A mayor valor de *mismatches shift* menor número de errores en la secuencia.

Se obtienen unos resultados muy interesantes. El algoritmo de BWT-SW es independiente del número de errores de la cadena. Esto se debe a que la mayor parte de su tiempo de ejecución es derivado de recorrer el árbol de prefijos del genoma, no de la secuencia. Por otra parte, el BWA tiene un elevado coste cuando las lecturas son muy malas (valores negativos en el eje X). Como se explicó en la figura anterior, si tiene *scores* muy bajos, le cuesta superar el umbral de la poda por lo que hace muchas más recursiones. Por último se observa que el Bowtie con búsqueda exacta va más rápido en secuencias con errores que en secuencias sin ellas. Esto se debe a que si la secuencia tiene errores y no está en el genoma, el algoritmo no llega a recorrer la secuencia entera con el algoritmo EXACTMATCH (Sección 3.3.1) y desecha la cadena; mientras que si la secuencia no tiene errores, realiza el algoritmo completo lo que conlleva un tiempo mayor.

Con respecto al análisis del tiempo frente al parámetro *indel*, todos los alineadores resultaron invariantes al ir cambiando el valor del *indel*. BWT-SW va mejor en lecturas largas porque en estas secuencias se produce un mayor número de *gaps*. Bowtie es el más rápido en la búsqueda de *mismatches* en secuencias cortas, mientras que BWA-SW va bien en búsquedas con *indels* en secuencias cortas.

8

Conclusiones y trabajo futuro

8.1. Conclusiones

Este TFG ha permitido la creación de una herramienta que implementa tres algoritmos de alineación de secuencias NGS basados en el FM-Index. Para llevarlo a cabo ha supuesto un esfuerzo de lectura y comprensión de todo el problema de la secuenciación y de la alineación, así como otros conceptos bioinformáticos necesarios para poder implementar los algoritmos.

Se ha seguido un desarrollo por etapas, en el que cada etapa se ha centrado en cada uno de los alineadores, teniendo su fase de implementación y su fase de pruebas.

Todo el proyecto se ha realizado en C++, pero la representación de resultados se ha hecho a partir de scripts de shell y awk, que recogían la salida del programa y la procesaban. Después se ha usado R, en concreto la librería `ggplot2` para la generación de gráficas y la librería `seaborn` de Python para la creación de los mapas de calor (*heatmaps*).

Durante el transcurso del Trabajo de Fin de Grado, se ha puesto en práctica gran parte del conocimiento adquirido durante la carrera, tanto en el Grado en Matemáticas como en el Grado en Ingeniería Informática. Al ser un trabajo de computación, la asignatura que más ha influido en la finalización de este trabajo ha sido *Programación II*, donde se proporciona una base de programación en C indispensable para la implementación de estos algoritmos. También los conocimientos que brinda la asignatura de *Análisis de algoritmos* han sido fundamentales para el correcto desarrollo de este trabajo, puesto que la idea general de este proyecto era el crear una comparación entre los distintos algoritmos de alineación escogidos. En general, casi todo el material del Grado en Ingeniería Informática ha estado presente en una mayor o menor medida. Respecto a los conocimientos propios del Grado en Matemáticas, su aplicación en este TFG puede no resultar tan evidente; sin embargo, han resultado cruciales para su creación. Las gráficas creadas en R proceden de los conocimientos que se adquieren en *Estadística*. También hay que tener en cuenta que toda la información de los alineadores procedía de artículos científicos que, por supuesto, estaban en inglés. El estar acostumbrado a este tipo de lecturas gracias a las asignaturas de matemáticas ha facilitado sin duda el trabajo.

Gracias a la creación de este TFG, se han llegado a varios resultados significativos. En primer lugar, no hay estudios tan rigurosos como este en el que se comparen los algoritmos en base a una implementación común. Segundo, los resultados coinciden con lo que se esperaría en la teoría

y permiten establecer unas directrices a la hora de elegir un algoritmo de alineación u otro en base al tipo de problema a tratar.

La experiencia de la elaboración de este proyecto ha sido muy gratificante. No solo por la ampliación de conocimientos adquirida en la realización de este TFG, si no también por todos aquellos conocimientos que desconocía y que se han adquirido después de tiempo de búsqueda y asimilación. Es el caso de la programación dinámica o toda la materia relacionada con la bioinformática. Además ha sido muy gratificante el darse cuenta de que estos cinco años de estudio han dado sus frutos y permiten abordar satisfactoriamente problemas de la envergadura de este TFG.

8.2. Trabajo futuro

Este proyecto tiene diferentes vías de trabajo futuro, donde se podría usar todo el conocimiento adquirido para dar lugar a diversas mejoras, algunas de las cuales se describen a continuación.

Añadir nuevos alineadores a la comparativa

En este trabajo solo se han tratado tres alineadores, basados todos en el FM-Index. Un trabajo futuro sería incorporar otros alineadores que estén basados en otras estructuras o métodos como tablas hash o alineadores probabilísticos.

Mejora de los algoritmos implementados

Tanto Bowtie como BWA, tienen diferentes formas de ejecutar sus rutinas. La que se ha implementado en este TFG revisa todo el genoma para encontrar la mejor alineación. Se podría añadir el resto de funcionalidades para unas versiones más completas.

Incorporación de diversas métricas de comparación

En este trabajo las comparaciones se han realizado en base a simulaciones de secuencias NGS. Sin embargo, en la práctica, no se dispone de una "realidad" sobre la que comparar y, por tanto, las medidas de los verdaderos positivos, falsos positivos, falsos negativos y verdaderos negativos no son triviales. En este sentido, han aparecido algunos artículos científicos en los últimos años que tratan de abordar este problema tanto para la comparación entre alineadores como para la comparación del mismo alineador con diferentes metaparámetros [29]. Una posible línea de trabajo futuro sería incorporar algunas de estas métricas a la herramienta desarrollada con el fin de poder hacer comparaciones entre alineadores sin necesidad de recurrir a simulaciones.

Bibliografía

[1] Genome Search Limited. http://www.sanger.ac.uk/about/who-we-are/sanger-institute/genome-research-limited.

[2] National Human Genome Research Institute. https://www.genome.gov/27565109/the-cost-of-sequencing-a-human-genome/.

[3] HTS Mappers. http://www.ebi.ac.uk/~nf/hts_mappers/.

[4] Mihai Pop Ben Langmead, Cole Trapnell and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology 2009*, 2009.

[5] Benjamin Thomas Langmead. *Highly Scalable Short Read Alignment with the Burrows-Wheeler Transform and Cloud Computing*. PhD thesis, 2009.

[6] NCBI. http://www.ncbi.nlm.nih.gov/.

[7] Hoang Thanh V. Robinson Michael L. Tsonis Panagiotis A. Liang Chun Kumar, Praveen Kumar Raj. Cadbure: A generic tool to evaluate the performance of spliced aligners on rna-seq data. *Scientific Reports 5, Article number: 13443*, 2015.

[8] Alvis Brazma Nuno A. Fonseca, Johan Rung and John C. Marioni. Tools for mapping high-throughput sequencing data. *Bioinformatics (2012)*, 2012.

[9] Lillian Lee Regina Barzilay. Bootstrapping lexical choice via multiple-sequence alignment. *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP):*, pages 164–171, 2002.

[10] Grzegorz Kondrak. Algorithms for language reconstruction. 2002.

[11] A. Prinzie and Dirk Van den Poel. Incorporating sequential information into traditional classification models by using an element/position- sensitive sam. 2005.

[12] Alvis Brazma John C. Marioni Nuno A. Fonseca, Johan Rung. Supplement for tools for mapping high-throughput sequencing data. 2012.

[13] Maurer and Lewis. Hash table methods. *ACM Computing Surveys (CSUR) , 7*, pages 5–19, 1975.

[14] Smith TF and Waterman MS. Identification of common molecular subsequences. *J Mol Biol. 147*, 1981.

[15] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology (Elsevier) 48*, 1970.

[16] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 390, 2000.

[17] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[18] Julian Seward. bzip2 and libbzip2, version 1.0. 5: A program and library for data compression. *URL http://www.bzip.orgnce*, 2007.

[19] Jacob T. Schwartz John Healy, Elizabeth E. Thomas and Michael Wigler. Annotating large genomes with exact word matches. *Genome Res 2003*, 2003.

[20] JLippert RA. Space-efficient whole genome comparisons with burrowswheeler transforms. *J Comput Biol 2005*, 2005.

[21] Green P Ewing B. Base-calling of automated sequencer traces using phred. ii. error probabilities. *Genome Res 1998*, 1998.

[22] Heng Li, Jue Ruan, and Richard Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851–1858, 2008.

[23] et al. Hillier, LaDeana W. Whole-genome sequencing and variant discovery in c. elegans. *Nature methods 5.2 (2008)*, pages 183–188, 2008.

[24] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 2009.

[25] S. L. Tam C. K. Wong T. W. Lam, W. K. Sung and S. M. Yiu. Compressed indexing and local alignment of dna. 2007.

[26] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows?wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[27] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[28] Weichun Huang, Leping Li, Jason R Myers, and Gabor T Marth. Art: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, 2012.

[29] Praveen Kumar Raj Kumar, Thanh V Hoang, Michael L Robinson, Panagiotis A Tsonis, and Chun Liang. Cadbure: A generic tool to evaluate the performance of spliced aligners on rna-seq data. *Scientific reports*, 5, 2015.

A

Manual de utilización

Para la compilación del proyecto, basta con acceder al código fuente y realizar:

`make all`

Con esta instrucción se realiza un `clean` de los ficheros y después se recompilan, generando un ejecutable "aligners" en la carpeta "bin".

Para visualizar las opciones de ejecución basta con usar el comando:

`./aligners -help` que imprime la ayuda para la utilización de la herramienta.

```
[ INFO ] Usage:
             aligners -m MODE -i INPUT -o OUTPUT -r READS    [-n MISMATCHES -s SEED -k FILE -l ]
         Options:
             -m, --mode:            Aligner mode [bowtie, bwa, bwt].
             -i, --input-file:      File with the genome sequence.
             -o, --ouput-file:      File to write the alignments.
             -r, --reads-file:      File with the reads.
             -n, --mismatches:      Number os mismatches.
             -k, --keep:            Save the structure in the file specified.
             -l, --load:            Reads the structure in the input file.
             -g, --gap:             Enable gap mode (only with bwa mode).
             -s, --seed:            Size of the seed.
             -h, --help:            Shows the different options and arguments.
```

De esta forma se puede seleccionar el modo a ejecutar y los diferentes parámetros que recibe.

B

Funciones FM-Index

En este anexo se incluye el código de las funciones más importantes a la hora de la creación del FM-Index, incluyendo su estructura.

```
#define pCheckPoint 128
#define pSuffixArray 32
typedef struct Fmi_
{
long long * ranking;
long long ** checkpoints;
long long positionDollar;
u_int8_t * bwt;
long long * positions;
long long  nNuc;

} Fmi;
long long * setPositions(long long * ranking, long long nNuc);
Fmi * createFMI(const char *text,long long nNuc);
void updateCheckpoints(char c, long long pos,long long ** checkpoints);
void updateRanking(char c,long long * ranking);
long long getMinRank(char c,long long * ranking);
long long getMaxRank(char c,long long * ranking);
char getCharacterBWT(Fmi fmi, long long i);
long long getCheckpoints(Fmi * fmi, long long pos, char c);
void pathString(long long pos,Fmi * fmi,int length,char *chain);
long long getPosition(Fmi fmi, long long i);
void printPosition(Fmi fmi, long long i, FILE * f);

void removeFMI(Fmi * fmi);
void saveFMI(FILE * f , Fmi *fmi);
Fmi * loadFMI(FILE * f );
```

```
Fmi * createFMI(const char *text, long long nNuc) {
        Fmi * fmi = (Fmi *) malloc(sizeof (Fmi));
        originalChain = text;
        long long *positions = sortPositions(nNuc);

        fmi->ranking = (long long*) malloc(sizeof (long long)*4);
        for (int i = 0; i < 4; i++) {
                fmi->ranking[i] = 0;
        }
        fmi->positionDollar = 0;
        fmi->checkpoints = (long long**) malloc(sizeof (long long*)*4);
        for (int i = 0; i < 4; i++) {
                fmi->checkpoints[i] = (long long *) malloc(sizeof (long long)*((n
        }
        fmi->positions = setPositions(positions, nNuc);
        fmi->nNuc = nNuc + 1;
        long long * check = (long long *) malloc(4 * sizeof (long long));
        for (int i = 0; i < 4; i++) {
                check[i] = 0;
        }
        long long i, cont = 0, numByte = 0;
        long long size;
        char c;
        u_int8_t byte;
        int positionInByte;
        size = (long long) ceil(nNuc / 4.0);
        fmi->bwt = (u_int8_t *) malloc(size);
        memset(fmi->bwt, 0, size);
        for (i = 0; i < nNuc + 1; i++) {
                if (positions[i] != 0) {
                        c = text[positions[i] - 1];
                        updateRanking(c, fmi->ranking);
                        check[getIntegerNuc(c)]++;
                        if (i % pCheckPoint == 0)
                                updateCheckpoints(check, i, fmi->checkpoints);
                        numByte = cont / 4.0;
                        positionInByte = cont % 4;
                        byte = getConversion(c, positionInByte);
                        fmi->bwt[numByte] = fmi->bwt[numByte] | byte;
                        cont++;
        } else {
                if (i % pCheckPoint == 0)
                        updateCheckpoints(check, i, fmi->checkpoints);
                fmi->positionDollar = i;
                }
        }
        free(check);
        free(positions);
        return fmi;
        }
```

C

Funciones Bowtie

```
#define MAX_QUALITY 70

typedef struct BowtieFmi_
{
        Fmi * forwardFmi;
        Fmi * mirrorFmi;
        long long nNuc;

} BowtieFmi;

void saveBowtieFMI(char * file , BowtieFmi *bowtieFmi);
BowtieFmi * loadBowtieFMI(char * file ,int seed);
BowtieFmi * createBowtieFMI(char * file , int seed);
void updateSpEp(Fmi *fmi, std::string pattern ,int length ,long long *sp,
        long long *ep);
void exactMatchBowtie(BowtieFmi *bowtieFMI,std::string pattern);
Ranges * caseRecurrent(Fmi *fmi, std::string pattern ,int * qualitySeq ,
                int pos ,int minPos, int mismatches, int quality ,
                long long sp, long long ep, int flagExact);
void case1(BowtieFmi *fmiBow, std::string pattern ,
        Read read, int mismatches ,AlignmentSet * set);
void case2(BowtieFmi *fmiBow, std::string pattern ,
        Read read, int mismatches ,AlignmentSet * set);
void case3(BowtieFmi *fmiBow, std::string pattern ,
        Read read ,int mismatches ,AlignmentSet * set);
void case4(BowtieFmi *fmiBow, std::string pattern ,
        Read read ,int mismatches ,AlignmentSet * set);
Alignment * createAlignmentBowtie(Fmi *fmi, std::string pattern ,
        int * qualitySeq , std::string text , long long pos , int quality ,
        long long startAlign ,char * id);
void inexactMatchBowtie(BowtieFmi *bowtieFMI, Read read, int mismatch);
void removeBowtieFMI(BowtieFmi * bowtieFMI);
```

```
void inexactMatchBowtie(BowtieFmi *bowtieFMI, Read read, int mismatch) {
        string pattern;
        pattern=string(read.sequence);
        string patternReverse(pattern);
        AlignmentSet *set = createAlignmentSet();

        reverse(patternReverse.begin(), patternReverse.end());
        switch (mismatch) {
                case 0:
                        case1(bowtieFMI, patternReverse, read, 0, set);
                        break;
                case 1:
                        case1(bowtieFMI, patternReverse, read, 1, set);
                        case2(bowtieFMI, pattern, read, 1, set);
                        break;
                case 2:
                        case1(bowtieFMI, patternReverse, read, 2, set);
                        case2(bowtieFMI, pattern, read, 2, set);
                        case3(bowtieFMI, patternReverse, read, 2, set);
                        break;
                case 3:
                        case1(bowtieFMI, patternReverse, read, 3, set);
                        case2(bowtieFMI, pattern, read, 3, set);
                        case3(bowtieFMI, patternReverse, read, 3, set);
                        case4(bowtieFMI, pattern, read, 3, set);
                break;

        }
        fprintAlignmentSet(NULL, set);
        destroyAlignmentSet(set);



}
```

D

Funciones BWA

```
typedef struct BwaFmi_
{
        Fmi * forwardFmi;
        Fmi * mirrorFmi;
        long long nNuc;
        int gap;

} BwaFmi;
void saveBwaFMI(char * file , BwaFmi *bwaFmi);
BwaFmi * loadBwaFMI(char * file , int gap);
Ranges * inexactMatchingRec(const char * pattern , int i,int * qualities ,int q, in
void inexactMatching(Read read , int z, BwaFmi bwaFmi);
BwaFmi * createBwaFMI(char * file , int gap);
void removeBwaFmi(BwaFmi * bwaFmi) ;
int * calculateD(char * pattern , BwaFmi  bwaFmi);



int * calculateD(char * pattern , BwaFmi bwaFmi) {
        long long k, l;
        char c;
        int z = 0,flag;
        unsigned int i;
        int * d = (int *) malloc(strlen(pattern) * sizeof (int));
        k = 0;
        l = bwaFmi.nNuc;
        for (i = 0; i <= (strlen(pattern) - 1); i++) {
                flag = 0;
                c = pattern[i];
                if(c=='N') {
                        flag = 1;
                }
```

```
                else{
                        k = getMinRank(c, bwaFmi.mirrorFmi->ranking) +
                                getCheckpoints(bwaFmi.mirrorFmi, k, c) - 1;
                        l = getMinRank(c, bwaFmi.mirrorFmi->ranking) +
                                getCheckpoints(bwaFmi.mirrorFmi, l, c) - 1;
                }
                if (k > l || flag ==1) {
                        k = 0;
                        l = bwaFmi.nNuc;
                        z++;
                }
                d[i] = z;
        }
        return d;
}
```

E

Funciones BWT-SW

```
typedef struct BwtSW_
{
        Fmi * fmi;
        long long nNuc;

} BwtSW;

typedef struct RowDP_
{
        int * G;
        int * D;
        int * I;

}RowDP;

BwtSW * createBwtSW(char * file);
void removeBwtSW(BwtSW * bwtSW) ;
void saveBwtSW(char * file , BwtSW *bwtSW);
BwtSW * loadBwtSW(char * file);
void alignReadBwtSW(BwtSW * bwtSW, Read * read);

void alignReadBwtRec(BwtSW * bwtSW, Dawg * dawg, long long sp, long long ep, int
RowDP * createRowDP(int length);
void removeRowDP(RowDP * r,int depth);
void removeRowDPTree(RowDP *r);
```

```
void alignReadBwtSW(BwtSW * bwtSW, Read * read){
        Dawg * dawg;
        int **I,**D,**G;
        int i;
        int max= -999;
        int length;
        long long sp= 0;
        long long ep =0;
        long long l;
        dawg = createDawg(read->sequence);
        length = strlen(read->sequence);
        long long * sa = (long long *) malloc(sizeof(long long)*2);
        I = (int **)malloc(sizeof(int*)*4*length);
        D = (int **)malloc(sizeof(int*)*4*length);
        G = (int **)malloc(sizeof(int*)*4*length);

        I[0]=(int *)malloc(sizeof(int)* dawg->length);
        D[0]=(int *)malloc(sizeof(int)* dawg->length);
        G[0]=(int *)malloc(sizeof(int)* dawg->length);

        for(i = 0; i< dawg->length;i++){
                I[0][i] = -999;
                G[0][i] = 0;
                D[0][i] = -999;
        }
        RowDP *r = createRowDP(dawg->length,1);
        G[1] = r->G;
        I[1] = r->I;
        D[1] = r->D;
        free(r);
        alignReadBwtRec(bwtSW, dawg, sp, ep, I, D, G,1,sa,&max);
        for(l=sa[0]+1 ; l<=sa[1];l++){
                Alignment *a=createAlignment(getPosition(*(bwtSW->fmi),l )+1,max
                printAlignment(a);
                destroyAlignment(a);
        }
        free(I[0]);
        free(D[0]);
        free(G[0]);
        free(I);
        free(D);
        free(G);
        free(sa);
        destroyDawg(dawg);
        fflush(stdout);

}
```