



Nuria Castejón Navarro

Implementación de un programa en lenguaje  
Perl para el ensamblaje de secuencias  
nucleotídicas

Trabajo tutorizado por: Dr. Héctor Candela Antón  
Departamento de Biología Aplicada, Área de Genética

Grado en Biotecnología  
Facultad de Ciencias Experimentales  
Universidad Miguel Hernández de Elche  
Curso 2016-2017

## ÍNDICE DE MATERIAS

<b>1. Resumen.....</b>	<b>4</b>
<b>2. Introducción .....</b>	<b>5</b>
2.1. Métodos de secuenciación de ácidos nucleicos.....	5
2.1.1. Método de Sanger .....	5
2.1.2. Secuenciación mediante hibridación .....	6
2.1.3. Secuenciación masivamente paralela .....	6
2.2. Aproximaciones bioinformáticas para la secuenciación de genomas .....	8
2.2.1. Resecuenciación de genomas previamente conocidos.....	8
2.2.2. Ensamblaje <i>de novo</i> de genomas .....	8
2.2.2.1. La estrategia OLC ( <i>overlap-layout-consensus</i> ).....	9
2.2.2.2. Estrategias basadas en grafos de De Bruijn .....	10
2.2.2.3. Grafos dirigidos de De Bruijn .....	11
2.2.2.4. Grafos bidirigidos de De Bruijn.....	11
<b>3. Antecedentes y Objetivos.....</b>	<b>13</b>
<b>4. Materiales y Métodos .....</b>	<b>14</b>
4.1. Lenguaje de programación.....	14
4.2. Pragmas .....	14
4.2.1. strict .....	14
4.2.2. warnings .....	15
4.2.3. autodie.....	15
4.3. Módulos .....	15
4.3.1. Bio::SeqIO .....	15
4.3.2. Bit::Vector .....	15
4.3.3. GetOptions.....	15
4.4. Subrutinas desarrolladas en este trabajo .....	16
4.4.1. revcom.....	16
4.4.2. to_binary.....	16
4.4.3. to_number.....	16
4.4.4. to_nucleotides.....	16
4.4.5. beginning / end .....	16
4.4.6. next_base .....	17
4.4.7. assemble .....	17
4.5. Implementación de grafos .....	17
4.6. Secuencias utilizadas en este trabajo .....	19
<b>5. Resultados .....</b>	<b>20</b>
5.1. Efecto del valor de $k$ sobre la calidad del ensamblaje.....	20
5.2. Efecto de los errores de secuenciación sobre la calidad del ensamblaje .....	25
5.3. Efecto combinado del valor de $k$ y la cobertura sobre la calidad del ensamblaje.....	27

5.4. Potencial del programa para la visualización de isoformas producidas mediante procesamiento alternativo de los intrones.....	28
<b>6. Discusión.....</b>	<b>30</b>
<b>7. Conclusiones y proyección futura.....</b>	<b>33</b>
<b>8. Bibliografía.....</b>	<b>34</b>
<b>9. Apéndice: Programas de ordenador.....</b>	<b>37</b>
9.1. paso1.pl.....	37
9.2. paso2.pl.....	39
9.3. libreria.pm.....	41
9.4. simulador.pl.....	46

## ÍNDICE DE FIGURAS

<b>Figura 1.-</b> Tipos de aristas en un grafo bidirigido de De Bruijn.....	12
<b>Figura 2.-</b> Implementación de los grafos de De Bruijn bidirigidos en nuestro programa.....	18
<b>Figura 3.-</b> Grafos bidirigidos de los cóntigos obtenidos al ensamblar el genoma del cloroplasto de <i>Arabidopsis thaliana</i> empleando distintos valores de $k$ .....	21
<b>Figura 4.-</b> Grafos bidirigidos de los cóntigos obtenidos al ensamblar el genoma del cloroplasto de <i>Oryza sativa</i> ssp. <i>japonica</i> empleando distintos valores de $k$ .....	22
<b>Figura 5.-</b> Los genomas cloroplásticos de <i>Arabidopsis thaliana</i> y <i>Oryza sativa</i> pueden reconstruirse mediante sendos recorridos del cartero chino.....	23
<b>Figura 6.-</b> Grafos bidirigidos de los cóntigos obtenidos al ensamblar el genoma del cloroplasto de <i>Zea mays</i> empleando distintos valores de $k$ .....	24
<b>Figura 7.-</b> Recorridos del cartero chino a través del grafo obtenido a partir del genoma del cloroplasto de <i>Zea mays</i> .....	25
<b>Figura 8.-</b> Grafo bidirigido del genoma del cloroplasto del <i>Arabidopsis thaliana</i> obtenido para $k = 45$ empleando lecturas simuladas con una tasa de error de $10^{-4}$ .....	26
<b>Figura 9.-</b> Grafos correspondientes al ensamblaje conjunto de 10 clones BAC de <i>Arabidopsis thaliana</i> a partir de lecturas simuladas sin errores.....	27
<b>Figura 10.-</b> Representación mediante un grafo de la estructura de las isoformas del gen AT2G37340 de <i>Arabidopsis thaliana</i> .....	29

## ÍNDICE DE TABLAS

<b>Tabla 1.-</b> Origen y número de acceso de las secuencias obtenidas de GenBank.....	19
<b>Tabla 2.-</b> Efecto de $k$ sobre el número de vértices y aristas de los grafos obtenidos al ensamblar los genomas cloroplásticos de diversas especies.....	20

## 1. Resumen

En este trabajo, hemos creado un programa que permite el ensamblaje *de novo* de secuencias nucleotídicas. El programa ha sido escrito en lenguaje de programación Perl y utiliza una estrategia basada en la representación de la secuencia mediante un grafo bidirigido de De Bruijn. A partir de secuencias nucleotídicas de pequeño tamaño (las lecturas), el programa produce un archivo en formato FastA con las secuencias de los cóntigos producto del ensamblaje y otro en formato dot con una representación gráfica de la relación existente entre dichos cóntigos. Hemos utilizado el programa para evaluar los efectos de varios parámetros (como la longitud de los  $k$ -meros, la cobertura o la tasa de error en las lecturas) sobre la calidad de los ensamblajes resultantes. Además, hemos comprobado que los grafos de De Bruijn son una herramienta interesante para detectar variantes en el procesamiento de los intrones. Por último, consideramos que nuestro programa representa una herramienta docente muy poderosa para que los estudiantes exploren la relación entre la Teoría de Grafos y la Genómica.

Palabras clave: ensamblaje de genomas; grafos de *de Bruijn*; grafos bidirigidos, bioinformática.

In this work, we have written a computer program for the *de novo* assembly of nucleotide sequences. The program has been written using the Perl programming language, and uses a strategy based on bidirected *de Bruijn* graphs. Starting from short nucleotide sequences (the reads), the program produces a FastA file containing the contig sequences and a dot file that allows one to visualize the relationship among the different contigs. We have used our program in order to evaluate the effects of several parameters (such as the  $k$ -mer length, the sequencing coverage or the sequencing error rates) on the quality of the resulting assemblies. In addition, we have found that De Bruijn graphs are an interesting tool that might facilitate the identification of novel splice variants. Finally, we think that our program is a powerful teaching tool that will help the students to investigate the relationships between graph theory and genomics.

Keywords: genome assembly; *de Bruijn* graphs; bi-directed graphs; bioinformatics.

## 2. Introducción

El descubrimiento del ADN como molécula portadora de la información genética supuso un hito para la ciencia, marcando el inicio de una nueva era en el conocimiento de los organismos vivos. Desde el desarrollo de los primeros métodos para determinar la secuencia de nucleótidos del ADN hasta la actualidad se han producido numerosos avances en el campo de la secuenciación han ido acompañados del desarrollo de técnicas bioinformáticas que permiten el ensamblaje de genomas completos a partir de las secuencias parciales generadas (Mardis, 2008). Estas nuevas técnicas facilitan el estudio de los organismos vivos y están conduciendo a una mejor comprensión de los mismos desde el punto de vista genético, genómico y molecular (Pareek *et al.*, 2011; Schuster, 2007). En esta Introducción se presentan los avances más recientes en las tecnologías de secuenciación y en las técnicas de análisis bioinformático que han propiciado la situación actual.

### 2.1. Métodos de secuenciación de ácidos nucleicos

#### 2.1.1. Método de Sanger

La secuenciación de ADN es posible desde 1977, cuando se desarrolló un método para determinar la secuencia nucleotídica de las moléculas de ADN (Sanger *et al.*, 1977). Este método se basa en la síntesis enzimática de fragmentos de ADN de longitud variable mediante la incorporación de 2',3'-didesoxinucleótidos terminadores marcados a la mezcla de reacción. En estas reacciones, la síntesis se inicia a partir de un único cebador, que es complementario de la molécula de ADN cuya secuencia quiere determinarse. La síntesis de los fragmentos se detiene cada vez que se incorpora un terminador, ya que el carbono 3' de los terminadores carece del grupo hidroxilo necesario para que la reacción progrese. Dado que la incorporación de los terminadores a los fragmentos se produce en posiciones al azar, el producto neto de la reacción es una colección de fragmentos de ADN de tamaño variable, cada uno de ellos marcado en su extremo 3'. Si bien inicialmente los terminadores se marcaban con  $^{32}\text{P}$ , actualmente se utilizan marcajes no isotópicos (mediante fluorocromos), que permiten distinguir los cuatro terminadores (ddATP, ddCTP, ddGTP y ddTTP) por la fluorescencia que emiten. Tras separar los fragmentos de la colección mediante electroforesis capilar, la secuencia se determina leyendo los colores de los fragmentos en orden de tamaño.

El método de Sanger es considerado un método de “primera generación” y fue el empleado para determinar la primera secuencia del genoma humano en 2001 (Lander *et al.*, 2001; Venter *et al.*, 2001).

### 2.1.2. Secuenciación mediante hibridación

La secuenciación mediante hibridación (*sequencing by hybridization*; Drmanac *et al.*, 2002) es un método de secuenciación indirecto que describimos aquí por consideraciones históricas. Si bien este método no ha llegado a ser utilizado en la práctica, el fundamento teórico de este método es el mismo que utilizan los programas de ensamblaje de genomas más modernos. El método se basa en la aplicación de técnicas de hibridación de ácidos nucleicos para determinar la secuencia nucleotídica de una molécula de ADN.

En una primera etapa, la molécula de ADN se fragmenta al azar en fragmentos de tamaño  $k$  (denominados  $k$ -meros a lo largo del trabajo), que se marcan con radiactividad o fluorescencia para posibilitar su detección. A continuación, se realiza la hibridación de los fragmentos marcados con una micromatriz (*microarray*) sobre la que se han distribuido regularmente e inmovilizado los  $4^k$  oligonucleótidos posibles de longitud  $k$ . Gracias al marcaje, la hibridación permite teóricamente identificar todos los  $k$ -meros presentes en la molécula de ADN cuya secuencia se desea conocer (Drmanac *et al.*, 1993).

En la práctica, este método conlleva importantes problemas técnicos, debido a las limitaciones de las técnicas de hibridación (que podrían resultar en abundantes falsos positivos y falsos negativos). Además, la saturación de la señal podría impedir la estimación del número de veces que cada  $k$ -mero está presente en la molécula secuenciada (valor denominado *multiplicidad* a lo largo del trabajo). Para valores pequeños de  $k$ , la señal obtenida será demasiado densa como para poder deducir la secuencia correcta a partir de los  $k$ -meros. Para valores muy elevados de  $k$ , resulta experimentalmente imposible disponer de una micromatriz con todas las sondas posibles (Idury y Waterman, 1995).

Los métodos de análisis bioinformático para la reconstrucción de la secuencia del fragmento de ADN a partir de los  $k$ -meros que la constituyen se describen más adelante en esta Introducción (véase la sección 2.2.2.2, de la página 10), ya que representan la base teórica del método de ensamblaje implementado en nuestro programa.

### 2.1.3. Secuenciación masivamente paralela

En los últimos años se han desarrollado nuevos métodos de secuenciación, cuyo éxito reside en que permiten determinar la secuencia de millones de fragmentos de ADN en paralelo y son asequibles a cualquier laboratorio (Metzker, 2010). Estos nuevos métodos se denominan, genéricamente, tecnologías de secuenciación masivamente paralela (*massively parallel sequencing technologies*) o de la siguiente generación (*next-generation sequencing*).

Estos métodos difieren del método de Sanger por su mayor rendimiento y la menor longitud de las lecturas generadas (Zhang *et al.*, 2011). En todos los casos, estas tecnologías permiten secuenciar los genomas con una gran cobertura y a un coste reducido. La reducción de los costes se debe a menudo a la inmovilización de las reacciones en superficie sólida, que permite minimizar la cantidad de reactivos empleados. En este apartado describimos sucintamente tres de las nuevas tecnologías que se encuentran disponibles comercialmente, basadas en distintos principios: la pirosecuenciación, la secuenciación por síntesis con terminadores reversibles, y la secuenciación mediante ligación.

La pirosecuenciación del ADN fue una de las primeras tecnologías utilizadas por la nueva generación de secuenciadores. Como en el método de Sanger, la secuencia se determina en una reacción de síntesis, iniciada a partir de un cebador. Para determinar la secuencia, esta técnica se basa en la detección del pirofosfato que se libera cada vez que se añade un nuevo nucleótido al extremo 3' de la molécula que se sintetiza. El pirofosfato se utiliza como sustrato en reacciones enzimáticas que conducen a la emisión de una señal luminiscente que es detectada por el equipo (Ronaghi *et al.*, 1996; Goldberg *et al.*, 2006). Para distinguir qué nucleótido ha sido incorporado a la molécula, los nucleótidos se añaden por separado, uno distinto en cada ciclo. Esta técnica fue inicialmente comercializada por la empresa *454 Life Sciences*, posteriormente absorbida por Roche (Margulies *et al.*, 2005; Pettersson *et al.*, 2009). La secuenciación de homopolímeros plantea un problema técnico, ya que conlleva la incorporación de múltiples nucleótidos idénticos en un mismo ciclo, dando lugar a un incremento en la señal luminiscente. Las dificultades en la medición precisa de dicha señal dan lugar a errores (inserciones o deleciones) en este tipo de secuencias.

La secuenciación por síntesis (*sequencing-by-synthesis*) es un método semejante al de Sanger. Difiere del mismo en que la polimerización de ADN utiliza terminadores reversibles, que impiden transitoriamente la elongación de la molécula (Bentley, 2006). Esto posibilita que se registre la señal fluorescente emitida por los nucleótidos terminadores incorporados en cada ciclo de la reacción. Este método, comercializado por la compañía Illumina, es posiblemente el más utilizado en la actualidad.

Un último método, el comercializado por la compañía ABI SOLiD, se basa en la ligación de octámeros marcados, de secuencia conocida, para la determinación de la secuencia de ADN.

## 2.2. Aproximaciones bioinformáticas para la secuenciación de genomas

Las estrategias bioinformáticas utilizadas para determinar la secuencia de un genoma completo difieren según se disponga o no de la secuencia de un genoma relacionado (por ejemplo, el de otro individuo de la misma especie) que pueda utilizarse como referencia.

Cuando se dispone de un genoma de referencia, la determinación de la secuencia se realiza alineando las secuencias producidas por un secuenciador automático (denominadas “lecturas”, del inglés *reads*) a la secuencia previamente conocida del genoma de referencia. Esta estrategia se denomina “resecuenciación”. Cuando no se dispone de una secuencia de referencia, la estrategia a seguir se denomina “ensamblaje *de novo*” (Miller *et al.*, 2010). El problema del ensamblaje *de novo* consiste en hallar una reconstrucción plausible de la secuencia completa del genoma, compatible con las lecturas producidas por el secuenciador.

### 2.2.1. Resecuenciación de genomas previamente conocidos

El principal problema de resecuenciación de genomas consiste en alinear las lecturas obtenidas a una secuencia genómica de referencia (Ruffalo *et al.*, 2011). Con el fin de acelerar el alineamiento de las lecturas, los programas empleados para esta labor típicamente realizan un procesamiento previo de la secuencia de referencia. La mayoría de estos programas utilizan algoritmos basados en la transformación de Burrows-Wheeler, una permutación del orden de los caracteres del genoma de referencia que permite que los alineamientos se realicen en tiempo proporcional a la longitud de las lecturas, independientemente del tamaño del genoma al que deban alinearse (Li y Durbin, 2009; Li y Durbin, 2010). Entre estos programas destacan Bowtie2 (Langmead, 2010) y BWA (Li y Durbin, 2009).

### 2.2.2. Ensamblaje *de novo* de genomas

El ensamblaje *de novo* de un genoma consiste en averiguar la secuencia de la que provienen las lecturas. Las estrategias modernas para ensamblar genomas se basan en la representación de la secuencia de las lecturas mediante grafos. Los grafos son un concepto matemático que permite representar relaciones entre distintos elementos. En un grafo, los elementos se representan mediante vértices (*nodes*) y las relaciones entre los mismos se representan mediante aristas (*edges*). Por ejemplo, algunas aplicaciones bioinformáticas utilizan los denominados “grafos de solapamientos” (*overlap graphs*), en los que cada vértice representa una lectura distinta y cada arista representa el solapamiento entre dos lecturas.

En un grafo de solapamientos, la reconstrucción del genoma requiere la identificación de un camino a través del mismo que visite todas las lecturas del grafo. Un camino que recorre un grafo visitando cada vértice una sola vez se denomina “hamiltoniano”. La identificación de caminos hamiltonianos es un problema de tipo NP-completo, denominación que recibe una clase de problemas para los que no se conocen algoritmos eficientes para hallar la solución. Dicho en otras palabras, la solución óptima del problema sólo podría encontrarse comprobando sistemáticamente un número astronómico de posibles soluciones (equivalente al número de permutaciones posibles del orden de los vértices).

Se denomina camino “euleriano” a un recorrido realizado a través del grafo que visita cada arista una sola vez, pudiendo pasar por un mismo vértice tantas veces como sea necesario. A diferencia de lo descrito para los caminos hamiltonianos, sí existen algoritmos eficientes para identificar caminos eulerianos. Como se describe en la sección 2.2.2.2, algunos programas reconstruyen el genoma mediante un recorrido euleriano.

Dado que las moléculas de ADN tienen direccionalidad inherente (por las diferencias entre sus extremos 5' y 3'), las aristas se representan a menudo mediante flechas que conectan un vértice fuente (*source node*) con un vértice destino (*sink node*). Los grafos que contienen este tipo de aristas se denominan “dirigidos” (*directed*). En un grafo de solapamientos dirigido, las flechas indican un solapamiento entre el extremo 3' de la secuencia representada por el vértice fuente y el extremo 5' de la secuencia representada por el vértice de destino.

Una complejidad adicional en la representación de fragmentos del genoma mediante grafos deriva de la naturaleza bicatenaria del ADN, situación en la que un mismo fragmento viene representado por dos vértices (correspondientes a la secuencia de ambas hebras). El uso de grafos “bidirigidos” (en los que ambos extremos de la arista tienen polaridad) permite representar la secuencia de ambas hebras mediante un mismo vértice. La polaridad de las aristas en cada uno de sus extremos permite especificar si los solapamientos tienen lugar entre las secuencias representadas por los vértices o sus complementarias. En este trabajo, las secuencias se han representado mediante un grafo bidirigido.

### **2.2.2.1. La estrategia OLC (*overlap-layout-consensus*)**

La estrategia denominada OLC (del inglés *overlap-layout-consensus*; Ruffalo *et al.*, 2011), que traducimos como “Solapamiento-Disposición-Consenso”, ha sido tradicionalmente utilizada para ensamblar lecturas largas, como las producidas con el método de Sanger. Como su nombre indica, la estrategia consta de tres etapas. En la primera etapa (*overlap*), deben identificarse todos los solapamientos existentes entre las

secuencias de las lecturas y sus complementarias. Dichos solapamientos pueden representarse por medio de un grafo de solapamientos bidirigido, en el que los fragmentos de ADN bicatenario se representan mediante vértices y los solapamientos entre los mismos se representan mediante aristas bidirigidas. En la segunda etapa (*layout*), se determina el orden en que deben visitarse los vértices para reconstruir el genoma. Típicamente el grafo se simplifica eliminando aquellas aristas que aportan información redundante, utilizando para ello la propiedad transitiva de los solapamientos (si  $A \rightarrow B$  y  $B \rightarrow C$ , entonces la arista  $A \rightarrow C$  puede eliminarse). Una vez simplificado, se intenta hallar un recorrido hamiltoniano (para que el recorrido integre información del mayor número posible de lecturas, representadas por vértices) mediante el uso de algoritmos voraces (*greedy*). Por último, se selecciona una secuencia de consenso (*consensus*), a partir de las secuencias individuales de los vértices atravesados (Miller *et al.*, 2010).

Lógicamente, este método funciona mejor para ensamblar genomas pequeños y con lecturas largas. Sin embargo, no se recomienda su uso con lecturas cortas (como las producidas por algunas tecnologías de secuenciación masivamente paralelas) debido al tamaño reducido de los solapamientos.

### 2.2.2.2. Estrategias basadas en grafos de De Bruijn

La estrategia más ampliamente utilizada para realizar el ensamblaje *de novo* de lecturas cortas (como las producidas por la tecnología de secuenciación por síntesis de Illumina) se basa en la representación de las lecturas mediante un grafo de De Bruijn. Por definición, los grafos de este tipo constan de aristas que representan secuencias de  $k$  caracteres. El vértice de origen de la arista representa el prefijo de longitud  $k-1$  de la secuencia de la arista (los  $k-1$  nucleótidos iniciales), y el vértice de destino representa el sufijo de longitud  $k-1$  de la secuencia de la arista (los  $k-1$  nucleótidos finales).

Los programas que utilizan estrategias basadas en grafos de De Bruijn, como Velvet, descomponen las lecturas en secuencias más cortas de longitud  $k$  ( $k$ -meros). Cada lectura de longitud  $L$  puede descomponerse en  $L-k+1$   $k$ -meros distintos, cuyos vértices de origen y de destino vienen determinados inmediatamente por la secuencia del  $k$ -mero. Tras construir un grafo a partir de la descomposición de todas las lecturas en sus correspondientes  $k$ -meros, el genoma se ensambla mediante un recorrido euleriano (para que integre información del mayor número posible de  $k$ -meros, representados en este caso por las aristas).

Destacaremos dos ventajas de esta estrategia respecto de la estrategia OLC: (1) La construcción de un grafo de De Bruijn no requiere la identificación de solapamientos entre las lecturas, que representa una de las tareas que requieren más tiempo en la

estrategia OLC, ya que los vértices del grafo y las aristas entre los mismos se deducen inmediatamente al descomponer las lecturas en  $k$ -meros. (2) Al considerar que la reconstrucción correcta del genoma es la que visita todas las aristas (conforme a la definición de un recorrido euleriano), existen algoritmos eficientes para llevar a cabo la reconstrucción del genoma.

Esta estrategia, hoy utilizada con gran éxito en combinación con las nuevas tecnologías de secuenciación masivamente paralela, fue inicialmente un mero desarrollo teórico asociado al método de secuenciación por hibridación (Drmanac *et al.*, 1989; Zhang *et al.*, 2003), que se describió en la sección 2.1.2.

### 2.2.2.3. Grafos dirigidos de De Bruijn

El programa Velvet utiliza una estrategia basada en la representación de las lecturas mediante grafos dirigidos de De Bruijn (Zerbino *et al.*, 2008). Para afrontar el problema de las secuencias bicatenarias, Velvet adopta las siguientes medidas: (1) El grafo no sólo incorpora las secuencias derivadas de las lecturas, sino también sus complementarias. El programa denomina “semivértices” a los vértices generados a partir de una secuencia y su complementaria. (2) La longitud de las secuencias representadas por los vértices del grafo debe ser un número impar. De este modo se evitan problemas asociados a las secuencias palindrómicas de longitud par, impidiendo que ambos semivértices representen la misma secuencia. (3) La secuencia del genoma se reconstruye mediante dos recorridos simultáneos a través del grafo, uno para cada una de las dos hebras del ADN bicatenario.

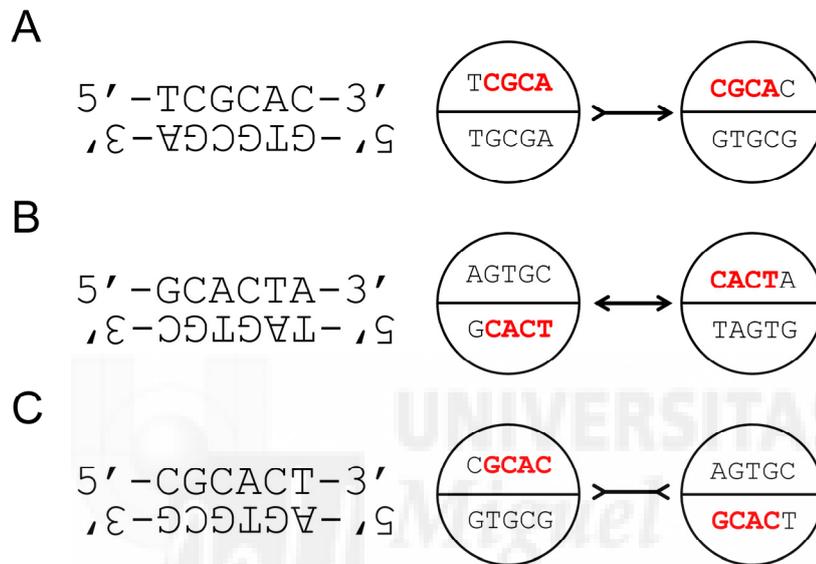
Otros programas que emplean estrategias similares son AByss (Simpson *et al.*, 2009), Oases (Schulz *et al.*, 2012) y SOAPdenovo (Li *et al.*, 2010).

### 2.2.2.4. Grafos bidirigidos de De Bruijn

Los vértices de los grafos bidirigidos de De Bruijn son más adecuados para la representación de genomas de ADN bicatenario. En estos grafos, cada vértice representa a una secuencia de longitud  $k-1$  y a su complementaria. Dado que un mismo vértice representa a dos secuencias distintas, suele adoptarse el criterio de considerar que la secuencia representativa del vértice es la alfabéticamente inferior (la que aparecería antes en una lista ordenada alfabéticamente). Los vértices que siguen esta convención se dice que han sido “normalizados” o que son “canónicos”.

Como muestra la Figura 1, las aristas de los grafos bidirigidos poseen polaridad en ambos extremos. El tipo de punta de flecha elegido en cada caso indica si el solapamiento de los vértices de origen y destino tiene lugar entre las secuencias “canónicas” o sus complementarias. Los posibles solapamientos definen los tres tipos de

aristas (o flechas) representados en la Figura 1. La realización de recorridos a través del grafo debe considerar que los distintos tipos de punta de flecha reflejan si el solapamiento se produce con la secuencia canónica del vértice o su complementaria. En consecuencia, un recorrido que finaliza en la secuencia canónica de un vértice, sólo puede continuar si existe una arista que parta de dicha secuencia (  ). Recíprocamente, un recorrido que finaliza en la secuencia complementaria sólo puede continuar si existe una arista que parta de dicha secuencia complementaria (  ).



**Figura 1.-** Tipos de aristas en un grafo bidirigido de De Bruijn. La figura ilustra los tres tipos de aristas existentes en este tipo de grafo. En cada caso, una secuencia de 6 nucleótidos (mostrada a la izquierda) ha sido descompuesta en 2 vértices que representan secuencias de 5 nucleótidos. Las secuencias de los vértices han sido normalizadas (la secuencia alfabéticamente inferior se muestra en el semicírculo superior). La parte común a la secuencia de los dos vértices se ha destacado en rojo. Nótese que dependiendo de si la secuencia compartida reside en la secuencia canónica o su complementaria se utilizan diferentes tipos de flechas (A-C).

---

### 3. Antecedentes y Objetivos

Las nuevas tecnologías de secuenciación masivamente paralela (*massively parallel sequencing technologies*) posibilitan la secuenciación y análisis de genomas a un precio asequible. Los programas como Velvet representan las lecturas mediante un grafo dirigido de De Bruijn y permiten ensamblar *de novo* la secuencia del genoma del que derivan. En el laboratorio del Dr. Héctor Candela se vienen utilizando rutinariamente programas basados en esta estrategia para realizar ensamblajes *de novo* de genomas orgánulos (del cloroplasto y la mitocondria), así como transcriptomas, de especies de interés agronómico. El objetivo principal de este Trabajo de Fin de Grado ha sido crear un programa de ensamblaje de secuencias siguiendo una estrategia basada en grafos bidirigidos de De Bruijn. Para ello, hemos empleado el lenguaje de programación Perl debido a su amplia utilización en Bioinformática. Este programa será una herramienta docente de gran utilidad ya que hemos incorporado la posibilidad de visualizar gráficamente el grafo resultante de los ensamblajes.

Además de este objetivo principal, hemos abordado los siguientes objetivos particulares, que coinciden con las secciones presentadas en la sección de resultados:

1) Evaluar los efectos del tamaño de la secuencia representada por los vértices sobre la calidad del ensamblaje resultante. Para ello, hemos simulado la secuenciación y ensamblaje de genomas de cloroplastos de varias especies de plantas ejecutando nuestro programa con diversos valores de  $k$ .

2) Evaluar los efectos de la existencia de errores de secuenciación sobre la calidad del ensamblaje resultante. Para ello, hemos simulado la secuenciación y ensamblaje de genomas de cloroplastos con lecturas portadoras de mutaciones.

3) Evaluar el efecto combinado del valor de  $k$  y la cobertura sobre la calidad del ensamblaje resultante. Para ello, hemos simulado la secuenciación y ensamblaje de un conjunto de diez clones BAC (*bacterial artificial chromosomes*) de *Arabidopsis thaliana* utilizando diferentes valores de  $k$  y de cobertura.

4) Evaluar el potencial del programa para la visualización y detección de isoformas del ARN mensajero producidas mediante procesamiento (*splicing*) alternativo de los intrones. Para ello, hemos combinado en un mismo ensamblaje secuencias genómicas y derivadas de ARNm de genes de *Arabidopsis thaliana*.

## 4. Materiales y Métodos

### 4.1. Lenguaje de programación

El lenguaje de programación escogido para implementar el programa fue Perl (*Practical Extraction and Report Language*). El lenguaje de programación Perl fue creado por Larry Wall en 1987. Se trata de un lenguaje interpretado desarrollado inicialmente para la manipulación de textos. En la actualidad este lenguaje se utiliza ampliamente en Bioinformática. Este programa comparte características con otros lenguajes de programación, como C, Shell o AWK. Sus principales características son la facilidad de uso, la disponibilidad de abundantes módulos y su flexibilidad, ya que permite tanto la programación estructurada como la programación orientada a objetos y la programación funcional. Hemos utilizado la versión 5.20.2 del lenguaje Perl.

Hemos evaluado el código obtenido con la herramienta *Perl Critic* (<http://perlcritic.com/>), que permite evaluar el código e identificar problemas en el código de programas escritos en lenguaje Perl. El código ha sido evaluado mediante esta aplicación con diferentes niveles de severidad, comprendidos entre “*gentle*” y “*brutal*”.

### 4.2. Pragmas

Los *pragmas* permiten definir el comportamiento del compilador frente a determinadas circunstancias. Se activan con el comando `use`. Deben especificarse al comienzo del programa para que cumplan correctamente su función. Su uso está muy extendido ya que favorecen el buen funcionamiento de los programas, evitando la comisión de errores o alertando de los mismos y deteniendo la ejecución del programa en curso.

#### 4.2.1. `strict`

`strict` es el *pragma* más comúnmente usado, ya que impone al programador pautas de obligado cumplimiento para el desarrollo del código, como el requisito de declarar las variables del programa con el comando `my` antes de utilizarlas. Este *pragma* permite detectar situaciones de programación insegura, incluyendo variables no inicializadas, referencias irreales, ausencia de comillas, bucles que no se han cerrado correctamente (falta o exceso de corchetes), etc. El uso de `strict` es particularmente importante en programas que manejan muchas variables, en los que podrían existir variables con igual nombre para procesos independientes. El uso de `strict` asegura que las variables son válidas únicamente en el bloque de código para el que han sido definidas, sin afectar globalmente al funcionamiento del programa.

### 4.2.2. warnings

El *pragma* `warnings` permite detectar fallos en el código del programa, impidiendo su ejecución y advirtiéndolo de los fallos encontrados.

### 4.2.3. autodie

El *pragma* `autodie` detiene el programa si alguna de las funciones no se está ejecutando correctamente, es decir, si está devolviendo valores erróneos.

## 4.3. Módulos

Los módulos (*libraries*) de Perl proporcionan nuevas funciones, escritos por otros programadores, que pueden incorporarse a un programa mediante el comando `use`. Por ejemplo, BioPerl es una colección de módulos desarrollados por programadores de todo el mundo que aportan funciones relacionadas con la bioinformática y la genómica. En este trabajo, hemos utilizado los siguientes módulos:

### 4.3.1. Bio::SeqIO

Bio::SeqIO es un módulo perteneciente a BioPerl (<http://search.cpan.org/dist/BioPerl/Bio/SeqIO.pm>) que permite leer y escribir secuencias a partir de/a un archivo en diversos formatos estándar (FastA, Genbank, EMBL y otros).

### 4.3.2. Bit::Vector

Este módulo (<http://search.cpan.org/~stbey/Bit-Vector-7.4/Vector.pod>) contiene funciones para crear y manipular vectores de bits del tamaño deseado. En nuestro programa hemos utilizado funciones de este módulo para crear vectores de bits en los que hemos almacenado la información del grafo.

### 4.3.3. GetOptions

El módulo `Getopt::Long` ([http://search.cpan.org/~chips/perl5.004\\_05/lib/Getopt/Long.pm](http://search.cpan.org/~chips/perl5.004_05/lib/Getopt/Long.pm)) permite especificar las opciones y el valor de los parámetros necesarios para la ejecución del programa en la línea de comandos. De esta forma, cuando el usuario no introduce las opciones imprescindibles para su ejecución, el programa devuelve un mensaje de error e información sobre las opciones que deben especificarse en la línea de comandos.

En el programa `paso1.pl` es necesario especificar los siguientes valores: (1) el nombre de un archivo en formato FastA con la secuencia de las lecturas (`--input`); (2) el de un archivo de salida (`--output`); (3) el tamaño  $k$  seleccionado para el ensamblaje (`--kmer`). En caso de no especificar un valor  $k$ , el programa utiliza  $k=35$  por defecto. A diferencia del criterio seguido en la Introducción de este documento, en el

programa hemos utilizado la misma nomenclatura que Velvet, que utiliza  $k$  para referirse a la longitud de la secuencia representada por los vértices. Las aristas, en consecuencia, poseen longitud  $k+1$  en nuestro programa.

El programa `paso2.pl` requiere: (1) nombre de un archivo de salida (`--out`) y el valor de  $k$  (`--kmer`).

#### 4.4. Subrutinas desarrolladas en este trabajo

En este trabajo hemos generado un módulo, denominado genéricamente `libreria.pm`, que contiene una colección de subrutinas. Las subrutinas son funciones que pueden referenciarse tantas veces como resulte necesario durante la ejecución del programa para realizar las operaciones específicas para las que han sido definidas. Las subrutinas del archivo `libreria.pm` se utilizan en los programas `paso1.pl` y `paso2.pl` con el comando `use`. Se han definido las siguientes subrutinas:

##### 4.4.1. `revcom`

Esta subrutina toma como entrada una secuencia nucleotídica dada y devuelve la secuencia complementaria. Para ello, invierte el orden de los caracteres y los translitera mediante la expresión `~tr/ATCG/TAGC/`.

##### 4.4.2. `to_binary`

Devuelve un número binario a partir de una secuencia de nucleótidos. Para ello, sustituye todos los nucleótidos de una secuencia por un par de dígitos binarios, según las siguientes equivalencias (se indican los comandos utilizados): `~s/A/00/g`, `~s/T/11/g`, `~s/G/10/g` y `~s/C/01/g`;

##### 4.4.3. `to_number`

Transforma una secuencia de nucleótidos a un número para posibilitar la conversión a binario de los datos guardados en memoria.

##### 4.4.4. `to_nucleotides`

Revierde el proceso, recuperando la secuencia de nucleótidos a partir de la secuencia combinada de 0-1.

##### 4.4.5. `beginning / end`

Devuelven información referida a las puntas de flecha de salida de un vértice y llegada al siguiente, de manera que un 0 indica que la flecha apunta hacia fuera del vértice y 1 que apunta hacia dentro del mismo.

#### 4.4.6. next\_base

Devuelve la base que debe añadirse a la secuencia según la información incluida en la arista correspondiente, dependiendo del tipo de solapamiento de las secuencias de origen y de destino.

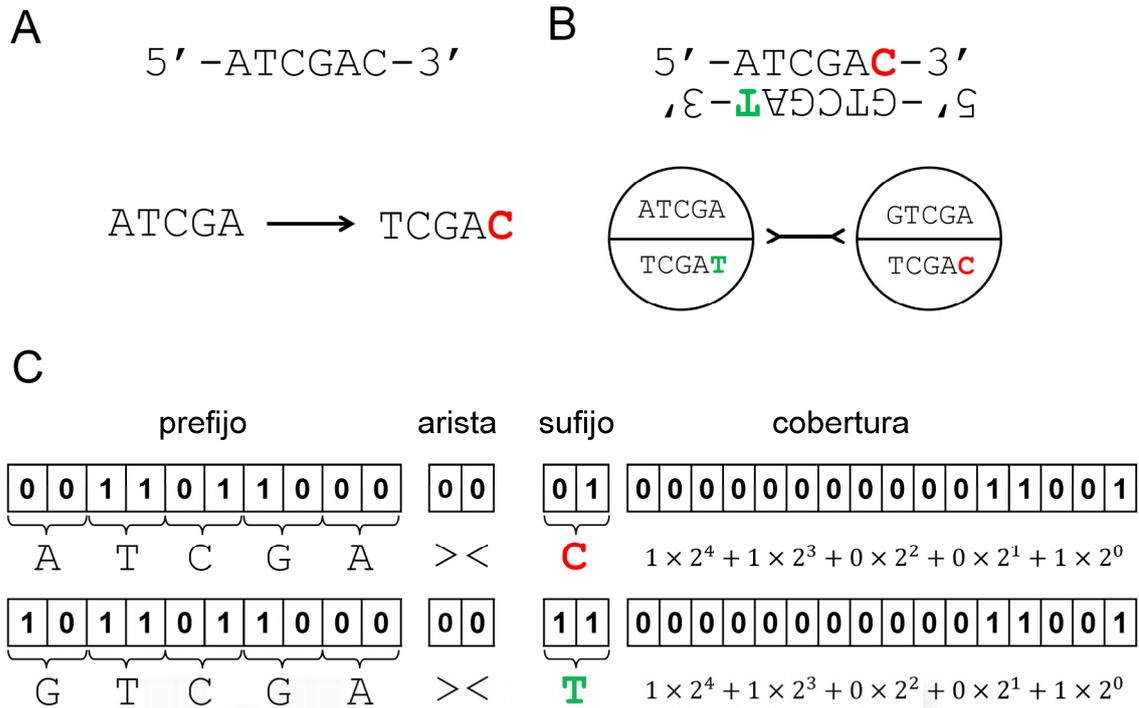
#### 4.4.7. assemble

Es la subrutina más compleja, encargada de llevar a cabo el ensamblaje de la secuencia. Para ello, se selecciona una arista no visitada todavía como punto de partida, desde la que inicia el ensamblaje en ambas direcciones. Cada vez que se visita una arista se marca como visitada. El ensamblaje de la secuencia se detiene cuando se alcanza una bifurcación (puntos donde el recorrido a seguir no puede determinarse sin ambigüedad). El proceso continúa hasta que se han visitado todas las aristas. El programa devuelve las secuencias ensambladas (denominadas “cóntigos”, del inglés *contigs*) en formato FastA, e incluye información sobre el vértice inicial, el vértice final, el tipo de punta de flecha inicial y final y la cobertura (número de veces promedio que cada nucleótido de la secuencia ensamblada está presente en el conjunto de lecturas).

### 4.5. Implementación de grafos

El programa maneja dos tipos de grafos. El primer tipo es un grafo bidirigido de De Bruijn, que se crea a partir de la descomposición de las lecturas en  $k$ -meros. La Figura 2 ilustra el procedimiento seguido para representar cada arista del grafo en la memoria del ordenador y guardarlo en el disco duro. Tras descomponer las lecturas en aristas de longitud  $k+1$ , los vértices de origen y destino (que representan secuencias de longitud  $k$ ) se normalizan (seleccionando la secuencia alfabéticamente menor) y sus secuencias se representan mediante un vector de  $2k$  bits, a razón de 2 bits por nucleótido. La secuencia del vértice de destino puede especificarse con sólo 2 bits adicionales, ya que puede deducirse de la del vértice de origen, con la sola excepción del nucleótido final (representado mediante los 2 bits). Utilizamos 2 bits para especificar el tipo de puntas de flecha al inicio y final de los vértices y 16 bits para indicar el número de veces que la arista está presente en las lecturas (un valor denominado cobertura o multiplicidad).

El segundo tipo de grafos, producido por el programa `paso2.p1`, es una representación del resultado final del ensamblaje. Las Figuras 3 a 10 corresponden a grafos de este tipo, en los que cada arista se corresponde con uno de los cóntigos resultantes del ensamblaje. Cada vértice representa una secuencia bicatenaria de longitud  $k$  compartida por las distintas aristas que inciden sobre el mismo. Podría considerarse que, en este tipo de grafos, los vértices representan solapamientos de longitud  $k$  entre las secuencias representadas por las aristas.



**Figura 2.-** Implementación de los grafos de De Bruijn bidirigidos en nuestro programa. (A) Descomposición de una secuencia de 6 nucleótidos en 2 vértices ( $k=5$ ) conectados mediante una arista dirigida. El nucleótido marcado en rojo es el único que no está presente en el vértice de origen. (B) Representación de la misma secuencia mediante un grafo bidirigido, asumiendo que se trata de ADN bicatenario. Los vértices han sido “normalizados” como se describe en la Introducción. La direccionalidad de las aristas refleja que el solapamiento se produce entre las secuencias complementarias de las canónicas. Los nucleótidos marcados en verde y rojo son los únicos que no pueden deducirse de la secuencia del vértice de origen cuando el recorrido se inicia en el vértice de la derecha o el de la izquierda, respectivamente. (C) Representación binaria de las aristas bidirigidas, según se describe en el texto. Cada arista es representada mediante dos vectores de bits, lo que permite recorrer el grafo en ambas direcciones.

Para representar el grafo de los cóntigos, el programa `paso2.pl` genera un archivo en el formato estándar Dot, utilizado por el software de visualización de grafos Graphviz (<http://www.graphviz.org/Home.php>). Las figuras de los cóntigos han sido generadas a partir de estos archivos mediante los programas `neato`, `circo` y `dot` (Gansner *et al.*, 2006). El formato de los archivos utilizados por Graphviz emplea un lenguaje específico que permite definir la forma, tipo y direccionalidad de las aristas, entre otras muchas características del grafo.

#### 4.6. Secuencias utilizadas en este trabajo

Las secuencias que se han empleado para poner a punto el programa y realizar estudios de su eficacia se han obtenido de la base de datos de Genbank y se muestran en la Tabla 1.

**Tabla 1.-** Origen y número de acceso de las secuencias obtenidas de GenBank.

Especie	Secuencia	Número de acceso	Longitud (pb)
<i>Arabidopsis thaliana</i>	Cloroplasto	NC_000932.1	154.478
<i>Oryza sativa japonica</i>	Cloroplasto	X15901.1	134.525
<i>Zea mays</i>	Cloroplasto	NC_001666.2	140.384
<i>Arabidopsis thaliana</i>	BAC F3C11	AC007167	103.874
<i>Arabidopsis thaliana</i>	BAC F1O13	AC007211	103.889
<i>Arabidopsis thaliana</i>	BAC F5K7	AC006413	106.716
<i>Arabidopsis thaliana</i>	BAC F27C21	AC006527	33.117
<i>Arabidopsis thaliana</i>	BAC F15K19	AC006429	94.875
<i>Arabidopsis thaliana</i>	BAC F23M2	AC007045	117.945
<i>Arabidopsis thaliana</i>	BAC F9B22	AC006528	95.765
<i>Arabidopsis thaliana</i>	BAC F26B6	AC003040	123.360
<i>Arabidopsis thaliana</i>	BAC F12K2	AC006233	88.383
<i>Arabidopsis thaliana</i>	BAC F13M22	AC004684	93.845

## 5. Resultados

### 5.1. Efecto del valor de $k$ sobre la calidad del ensamblaje

Hemos utilizado nuestro programa para estudiar el efecto del valor de  $k$  (tamaño de la secuencia representada por los vértices) sobre la complejidad del grafo resultante. Para nuestro estudio, seleccionamos los genomas de los cloroplastos de la planta modelo *Arabidopsis thaliana* y dos plantas cultivadas, el maíz (*Zea mays*) y el arroz (*Oryza sativa* ssp. *japonica*), cuyas secuencias se hallan depositadas en Genbank (Tabla 1). Los genomas de los cloroplastos son moléculas de ADN bicatenario circular que poseen dos regiones de copia única (denominadas SSC y LSC) separadas por otras tantas repeticiones invertidas de una misma secuencia (denominadas IRa e IRb). Por su topología circular y la presencia de secuencias repetidas, consideramos que los genomas de los cloroplastos representan un modelo excelente para evaluar el funcionamiento de nuestro programa.

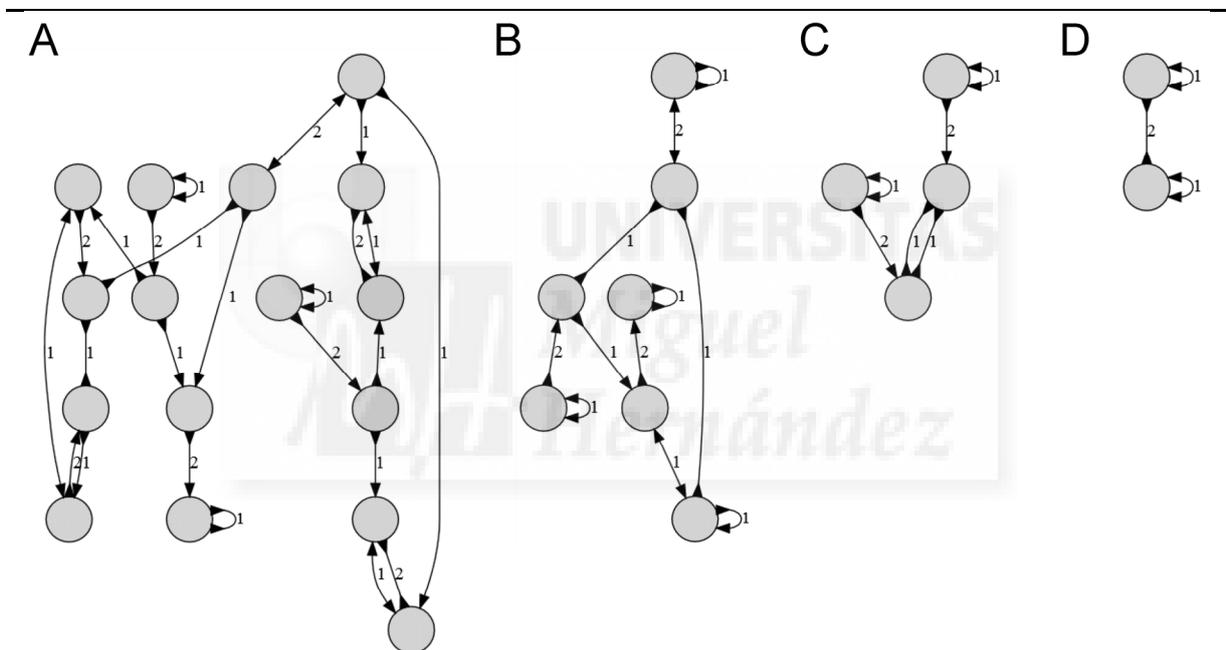
Hemos simulado el ensamblaje de estos tres genomas utilizando valores de  $k$  comprendidos entre 21 y 139. Como resultado de cada ensamblaje, obtuvimos dos archivos: (1) un archivo con la secuencia de los contigs (*contigs*) resultantes (en formato FastA), y (2) otro archivo con la representación del ensamblaje mediante un grafo (en formato Dot). En estos archivos, el número de contigs siempre coincide con el número de aristas del grafo. El número de vértices, sin embargo, depende del número de solapamientos existentes entre los distintos contigs. La Tabla 2 recoge los resultados obtenidos para los tres genomas estudiados. Como puede apreciarse, los valores mayores de  $k$  conducen a grafos con un menor número de aristas y vértices en todos los casos.

**Tabla 2.-** Efecto de  $k$  sobre el número de vértices y aristas de los grafos obtenidos al ensamblar los genomas cloroplásticos de diversas especies.

Especie	Número	Valor de $k$								
		21	25	29	33	37	41	45	53	73
<i>A. thaliana</i>	Vértices	30	16	10	7	6	4	<b>2</b>	=	=
	Aristas	43	22	13	9	7	5	<b>3</b>	=	=
<i>O. sativa</i>	Vértices	44	28	16	12	10	8	6	<b>2</b>	=
	Aristas	66	42	25	18	15	12	9	<b>3</b>	=
<i>Z. mays</i>	Vértices	43	29	13	8	7	6	6	6	<b>4</b>
	Aristas	65	43	20	11	11	9	9	9	<b>6</b>

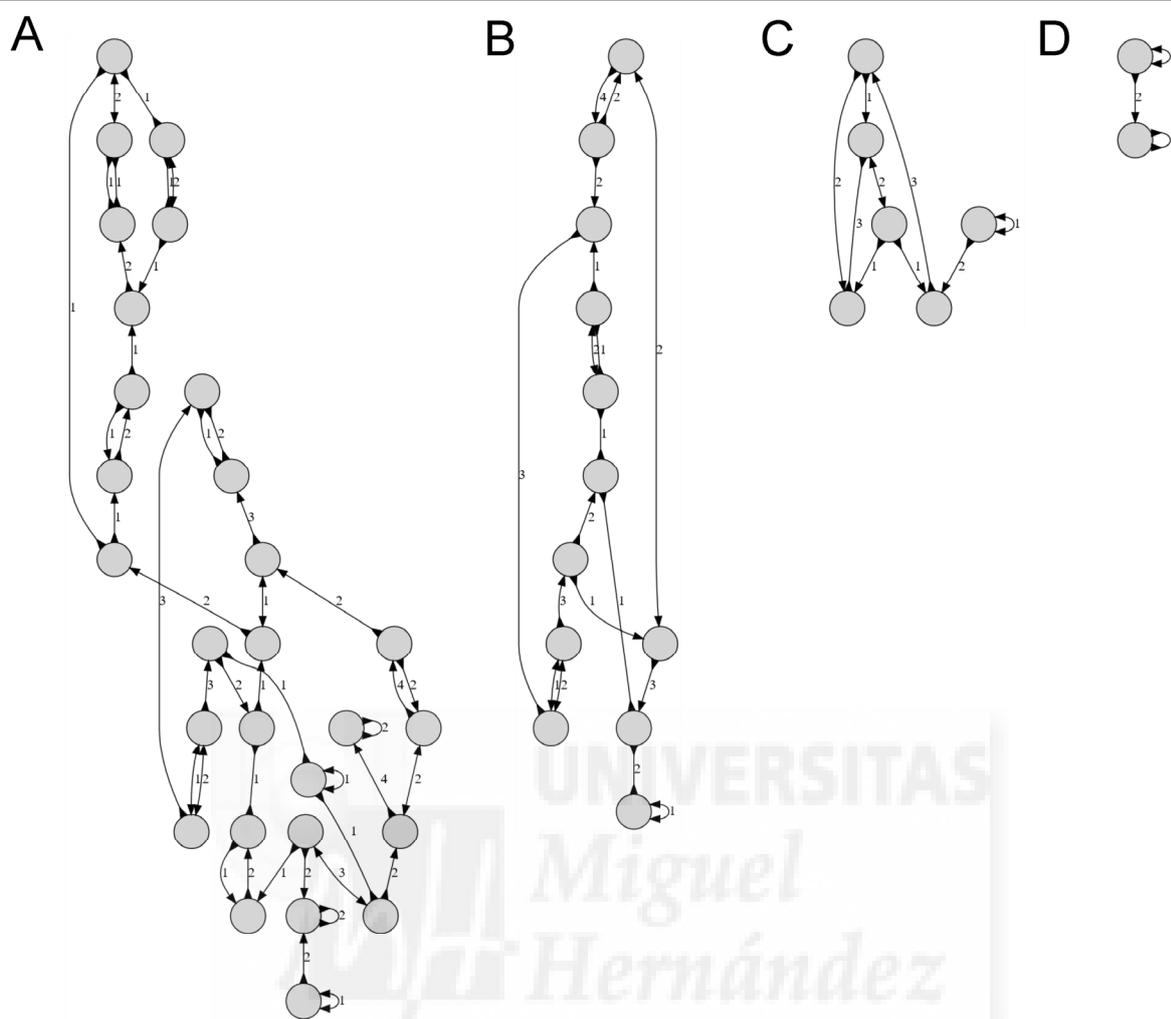
Se indica mediante un signo = aquellos casos en los que el resultado fue idéntico al anterior.

El genoma del cloroplasto de *Arabidopsis thaliana* es una molécula circular de 154.478 pares de bases (pb). La Figura 3 muestra la reducción en la complejidad del grafo obtenida al incrementar el valor de  $k$ . El grafo para  $k=25$  (Figura 3A) consta de 16 vértices conectados entre sí mediante 22 aristas. Cada una de estas aristas representa a uno de los cóntigos obtenidos en el ensamblaje correspondiente. La reconstrucción completa del genoma circular requiere la identificación de un ciclo (camino que comienza y finaliza en el mismo vértice) a través del grafo, resolviendo el denominado “problema del cartero chino” (*Chinese postman’s problem*; problema consistente en visitar cada arista del grafo al menos una vez). Para identificar el ciclo que corresponde a la reconstrucción correcta del genoma es preciso que dicho ciclo visite cada arista el número de veces especificado en su correspondiente etiqueta.



**Figura 3.-** Grafos bidirigidos de los cóntigos obtenidos al ensamblar el genoma del cloroplasto de *Arabidopsis thaliana* empleando distintos valores de  $k$ . (A)  $k = 25$ . (B)  $k = 33$ . (C)  $k = 41$ . (D)  $k = 45$ . Se observa que el número de cóntigos, representados mediante aristas, disminuye en los ensamblajes realizados con tamaños de  $k$ -mero superiores. La etiqueta de cada arista representa el número de veces que la secuencia correspondiente se repite en el genoma.

Para valores de  $k$  iguales o superiores a 45 el grafo obtenido consta únicamente de 2 vértices y 3 aristas (Figura 3D). Las tres secuencias asociadas con estas tres aristas se corresponden exactamente con las dos regiones de copia única y la secuencia repetida. Esta última, como indica la etiqueta de la arista correspondiente, debe ser atravesada dos veces para reconstruir correctamente la secuencia del genoma circular.

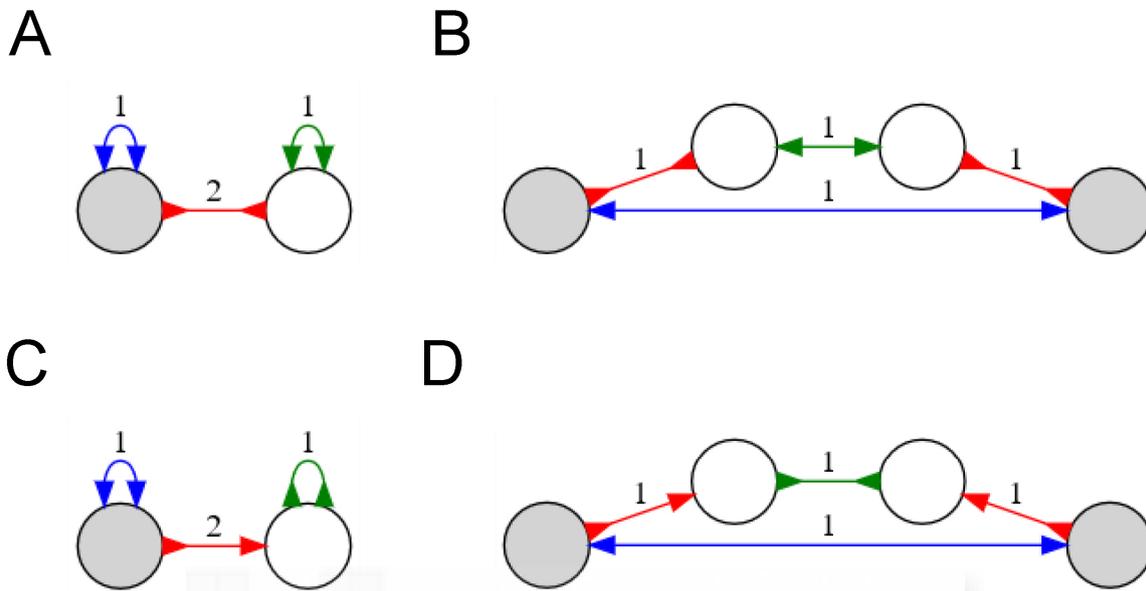


**Figura 4.-** Grafos bidirigidos de los c3ntigos obtenidos al ensamblar el genoma del cloroplasto de *Oryza sativa* ssp. *japonica* empleando distintos valores de  $k$ . (A)  $k = 25$ . (B)  $k = 33$ . (C)  $k = 45$ . (D)  $k = 53$ . Se observa que el n3mero de c3ntigos, representados mediante aristas, disminuye en los ensamblajes realizados con tama3os de  $k$ -mero superiores. La etiqueta de cada arista representa el n3mero de veces que la secuencia correspondiente se repite en el genoma.

Un resultado an3logo se obtuvo para el genoma del arroz, cuyo tama3o es de 134.525 pb (Tabla 2). Como muestra la Figura 4, el grafo obtenido para  $k=25$  consta de 42 aristas (c3ntigos) y 22 v3rtices. El n3mero de aristas y v3rtices disminuye conforme se incrementa  $k$ . El grafo m3s simple, con 3 aristas y 2 v3rtices, se obtuvo para valores de  $k$  iguales o superiores a 53.

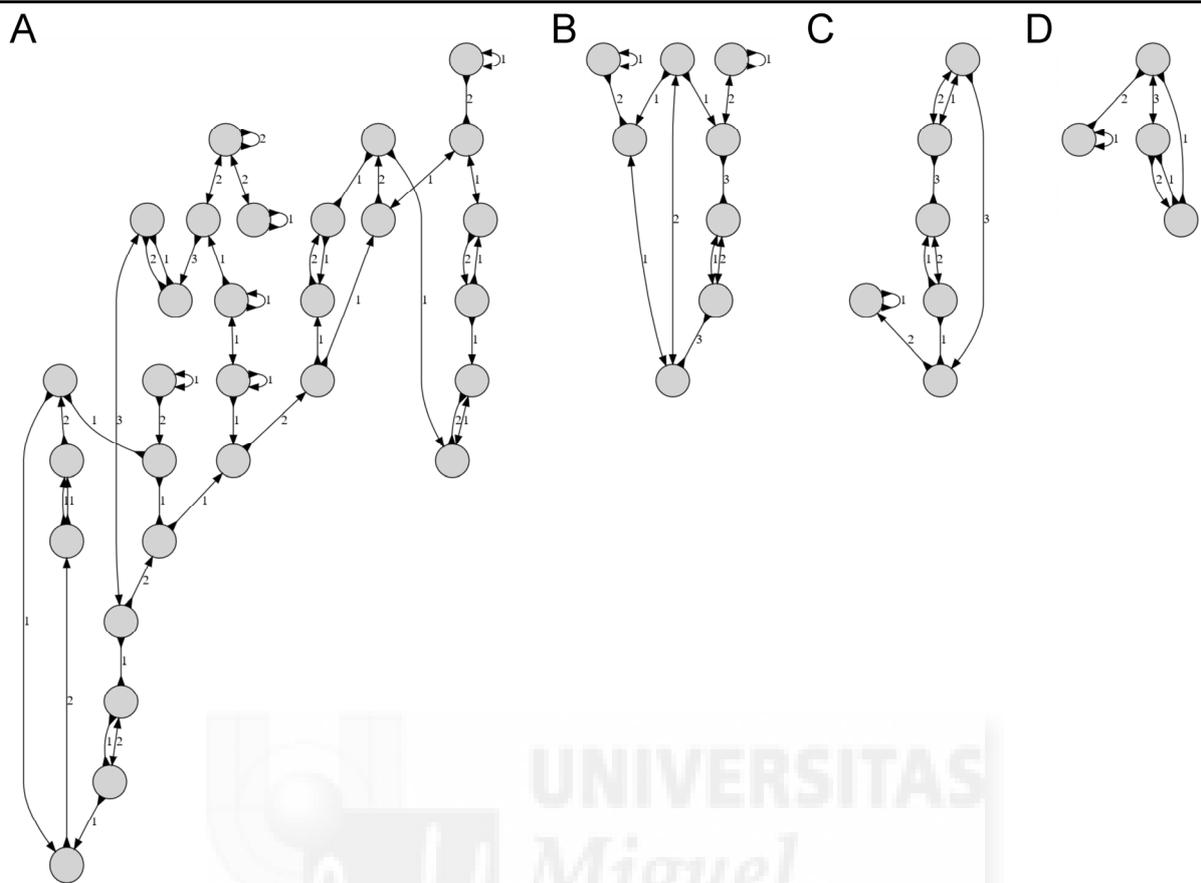
A partir de los grafos m3s simples obtenidos para los genomas de los cloroplastos de *Arabidopsis thaliana* y *Oryza sativa*, la Figura 5 presenta los recorridos c3clicos que deben realizarse para combinar las secuencias de los c3ntigos individuales en una 3nica secuencia, correspondiente al genoma completo del cloroplasto. Como puede apreciarse en la misma figura, dos aristas que indiquen sobre el mismo v3rtice pueden combinarse en un

recorrido si, y sólo si, la arista de entrada al vértice posee la misma orientación que la arista de salida (como por ejemplo,  o .



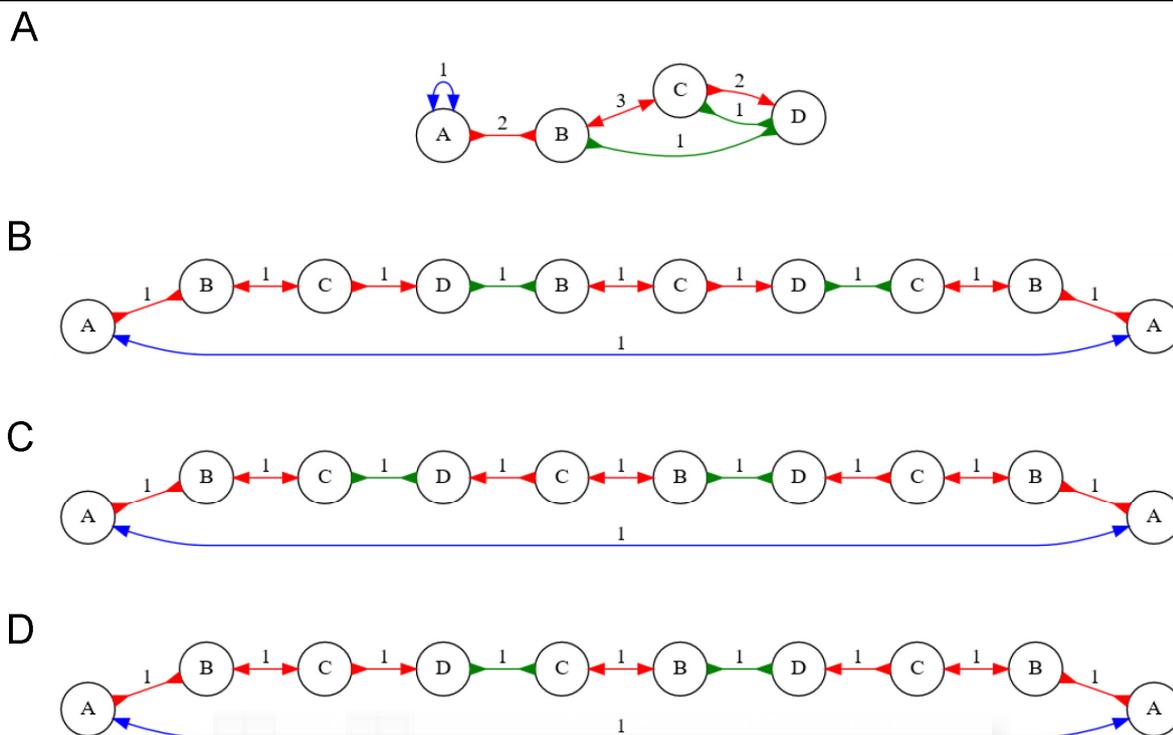
**Figura 5.-** Los genomas cloroplásticos de *Arabidopsis thaliana* y *Oryza sativa* pueden reconstruirse mediante sendos recorridos del cartero chino. (A) Grafo bidirigido del genoma del cloroplasto de *Arabidopsis thaliana* obtenido para  $k = 45$  (véase la Figura 3D). El grafo consta de dos vértices y tres aristas. (B) Recorrido del cartero chino a través del grafo anterior. Este recorrido se corresponde con una molécula de topología circular que contiene dos repeticiones invertidas de la misma secuencia, representada por las aristas de color rojo. Las aristas de color verde y azul representan las dos regiones de copia única presentes en el genoma del cloroplasto. (C) Grafo bidirigido del genoma del cloroplasto de *Oryza sativa* obtenido para  $k = 53$  (véase la Figura 4D). (D) Recorrido del cartero chino a través del grafo anterior. Este recorrido también se corresponde con una molécula de topología circular con dos repeticiones invertidas.

El genoma del cloroplasto del maíz rindió un resultado diferente (Figura 6). El grafo más simple se obtuvo para valores de  $k$  iguales o superiores a 73. Todos los valores mayores utilizados, hasta  $k=139$ , rindieron grafos con idéntica topología. El grafo más simple obtenido consta de 6 aristas y 4 vértices, conectadas según se indica en la Figura 6D. El examen de esta figura indica que la diferente topología se debe a que el genoma contiene tres copias de una secuencia de longitud superior a 139 pb, como indica la etiqueta de una de las aristas. Dado que la longitud del cóntigo correspondiente es de 171 pb, cabe suponer que sólo los valores de  $k$  iguales o superiores a 173 permitirán obtener grafos más simples que el de la Figura 6D.



**Figura 6.-** Grafos bidirigidos de los cóntigos obtenidos al ensamblar el genoma del cloroplasto de *Zea mays* empleando distintos valores de  $k$ . (A)  $k = 25$ . (B)  $k = 33$ . (C)  $k = 45$ . (D)  $k = 73$ . Se observa que el número de cóntigos, representados mediante aristas, disminuye en los ensamblajes realizados con tamaños de  $k$ -mero superiores. La etiqueta de cada arista representa el número de veces que la secuencia correspondiente se repite en el genoma.

El grafo de la Figura 6D contiene tres ciclos, correspondientes a otras tantas reconstrucciones posibles del genoma, que se representan en la Figura 7. Obviamente, sólo una de ellas puede corresponder a la verdadera estructura del genoma del cloroplasto del maíz. Parece probable que la secuencia  $\{A, B, C, D\}$  de vértices, que se visita en ambas orientaciones en el ciclo de la Figura 7D, corresponda a las repeticiones invertidas del genoma. La secuencia combinada de los tres cóntigos (aristas) que conectan dichos vértices posee una longitud de 22.748 pb y se alinea perfectamente con ambas repeticiones invertidas del genoma del maíz (cuyas coordenadas son, respectivamente, 82.353-105.100 y 117.637-140.384 en la secuencia NC\_001666.2).



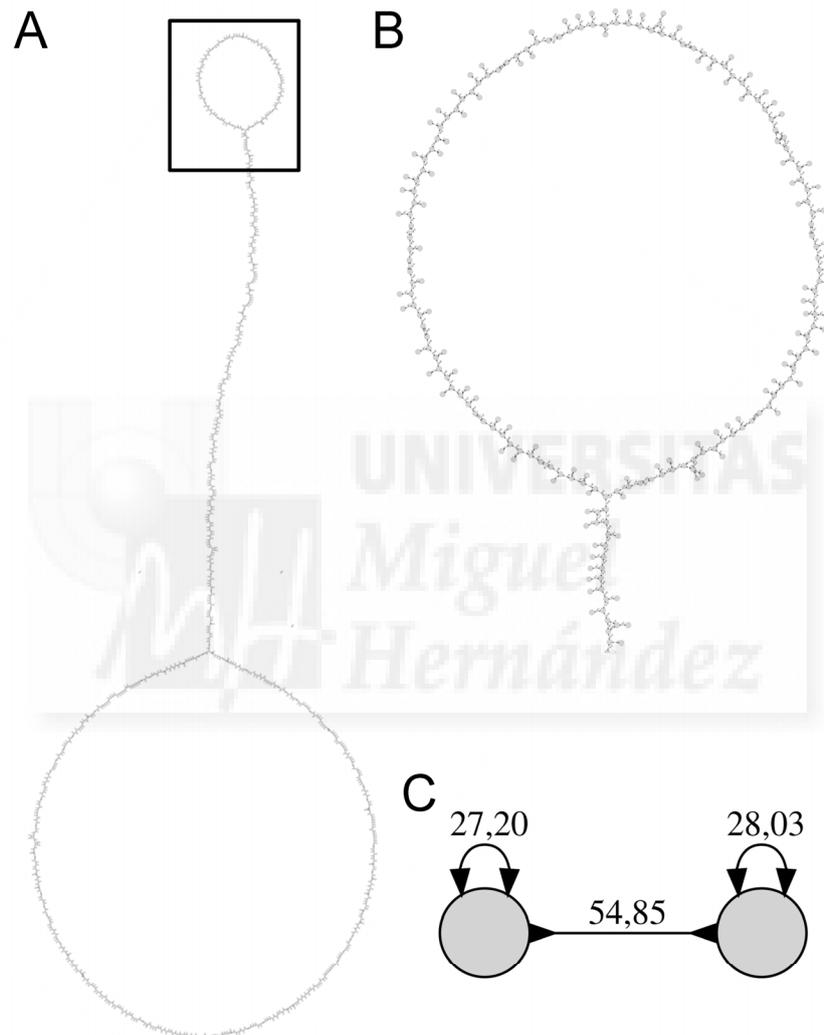
**Figura 7.-** Recorridos del cartero chino a través del grafo obtenido a partir del genoma del cloroplasto de *Zea mays*. (A) Grafo bidirigido del genoma del cloroplasto del maíz obtenido para  $k = 73$  (véase la Figura 6D). El grafo consta de cuatro vértices y cinco aristas. La reconstrucción correcta del genoma requiere identificar un ciclo que visite algunas aristas una sola vez (las indicadas en verde y azul) y otras aristas dos o tres veces (las indicadas en rojo). (B-D) Recorridos del cartero chino a través del grafo anterior. Existen tres recorridos compatibles con una molécula de topología circular. El ciclo mostrado en D contiene dos repeticiones de la secuencia de vértices  $A \rightarrow B \rightarrow C \rightarrow D$ , en posición invertida, cuyo tamaño coincide con el de las repeticiones invertidas del genoma del cloroplasto del maíz.

## 5.2. Efecto de los errores de secuenciación sobre la calidad del ensamblaje

Las lecturas (*reads*) producidas por las tecnologías de secuenciación masivamente paralelas a menudo contienen errores de secuenciación con una frecuencia elevada. Para evaluar el efecto de dichos errores sobre la contigüidad de la secuencia ensamblada, hemos realizado ensamblajes utilizando lecturas simuladas con errores de secuenciación. Para simular las lecturas, hemos utilizado un programa que produce lecturas a partir de posiciones seleccionadas al azar de la secuencia de un genoma en formato FastA. Este programa introduce aleatoriamente sustituciones nucleotídicas en la secuencia de las lecturas conforme a la probabilidad que se le indica.

Hemos simulado la secuenciación del genoma del cloroplasto de *Arabidopsis thaliana* con una cobertura de  $50\times$  y lecturas de 100 nucleótidos. La probabilidad de error en la secuencia de estas lecturas fue de  $10^{-4}$  sustituciones/posición. Hemos ensamblado dichas

lecturas seleccionando  $k=45$ , el valor mínimo que rinde el grafo más simple posible cuando se utilizan lecturas carentes de errores de secuenciación (Figura 3D). El ensamblaje de dichas lecturas produjo un grafo mucho más complejo, con 1.544 vértices y 1.617 aristas (cóntigos), como se muestra en la Figura 8. El principal efecto de las sustituciones nucleotídicas es la aparición de “cabos sueltos” (*hanging tips*), en las posiciones en las que han ocurrido errores de secuenciación.



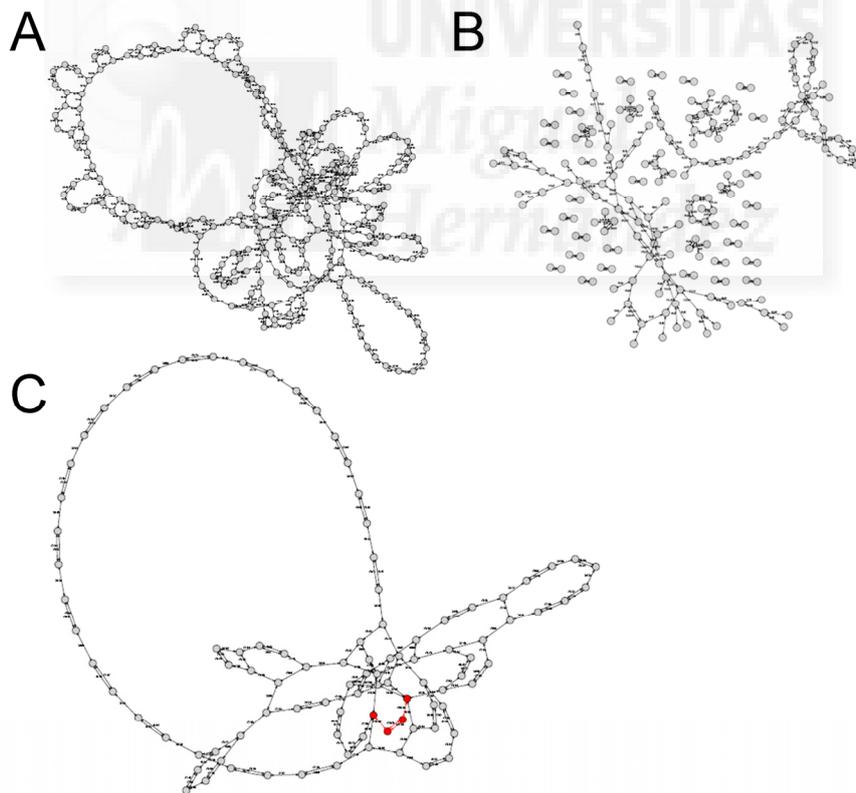
**Figura 8.-** Grafo bidirigido del genoma del cloroplasto del *Arabidopsis thaliana* obtenido para  $k = 45$  empleando lecturas simuladas con una tasa de error de  $10^{-4}$ . (A) Grafo bidirigido de los cóntigos. (B) Detalle del grafo mostrado en A donde se aprecian numerosos cabos sueltos. (C) Grafo más simple obtenido al seleccionar las aristas cuya cobertura es superior a 2.

Para obtener un menor número de cóntigos de mayor longitud, hemos añadido al programa la opción de utilizar únicamente aquellas aristas del grafo de *de Bruijn* cuyos valores de cobertura se hallan en el rango comprendido entre los valores especificados por

las opciones `--mincov` y `--maxcov`. Desechando los vértices cuya cobertura es igual o inferior a 2, el grafo de la Figura 8A se simplificó hasta el grafo mostrado en la Figura 8C, que es idéntico al de la Figura 3D. Las etiquetas de este nuevo grafo indican los valores de cobertura de los tres cóntigos obtenidos. Obsérvese que la eliminación de los cabos sueltos tiene como consecuencia una importante reducción de la cobertura (desde  $50\times$  hasta  $27,20\times$  y  $28,03\times$  en los cóntigos de copia única, y de  $100\times$  hasta  $54,85$  en el cóntigo correspondiente a la repetición invertida).

### 5.3. Efecto combinado del valor de $k$ y la cobertura sobre la calidad del ensamblaje

El grado de contigüidad de un ensamblaje *de novo* requiere que se alcance un equilibrio entre el valor de  $k$  utilizado y la cobertura del experimento de secuenciación. Como hemos visto anteriormente, en condiciones ideales (con cobertura ilimitada), los valores elevados de  $k$  producen grafos con menos vértices y aristas. Con valores de cobertura limitantes, sin embargo, esos mismos valores de  $k$  podrían conducir a cóntigos de menor



**Figura 9.-** Grafos correspondientes al ensamblaje conjunto de 10 clones BAC de *Arabidopsis thaliana* a partir de lecturas simuladas sin errores. (A) Ensamblaje realizado con cobertura  $100\times$  y  $k=61$ . (B) Ensamblaje realizado con cobertura  $100\times$  y  $k=91$ . (C) Ensamblaje realizado con cobertura  $200\times$  y  $k=91$ . Los vértices y aristas indicados en rojo en C permiten ensamblar perfectamente la secuencia del vector pBeloBac-Kan, común a todos los clones BAC.

longitud, debido al menor número de lecturas solapantes (nótese que, para que dos lecturas se solapen, deben compartir al menos  $k$  nucleótidos, condición que sucede con menor frecuencia cuando la cobertura es baja o cuando  $k$  es elevado). Hemos utilizado nuestro programa para simular el ensamblaje de un conjunto de 10 clones BAC de *Arabidopsis thaliana*, cuyas secuencias hemos obtenido de Genbank, con insertos de tamaño comprendido entre 33.117 y 123.360 pb (Tabla 1). Esta simulación nos ha permitido evaluar las condiciones en las que debería realizarse la secuenciación los clones.

Como se observa en la Figura 9, la contigüidad del grafo obtenido depende de la cobertura y del valor de  $k$  seleccionado. Con una cobertura 100× y  $k=61$ , el grafo está completamente conectado (Figura 9A). Con idéntica cobertura y  $k=91$ , el grafo se compone de numerosas *islas* (cántigos conectados entre sí y desconectados del resto), que aparecen como consecuencia de la situación descrita en el párrafo anterior (Figura 9B). La contigüidad del grafo se restablece al incrementar la cobertura a 200×, nivel al que la cantidad de lecturas resulta suficiente para la identificación de todos los solapamientos existentes.

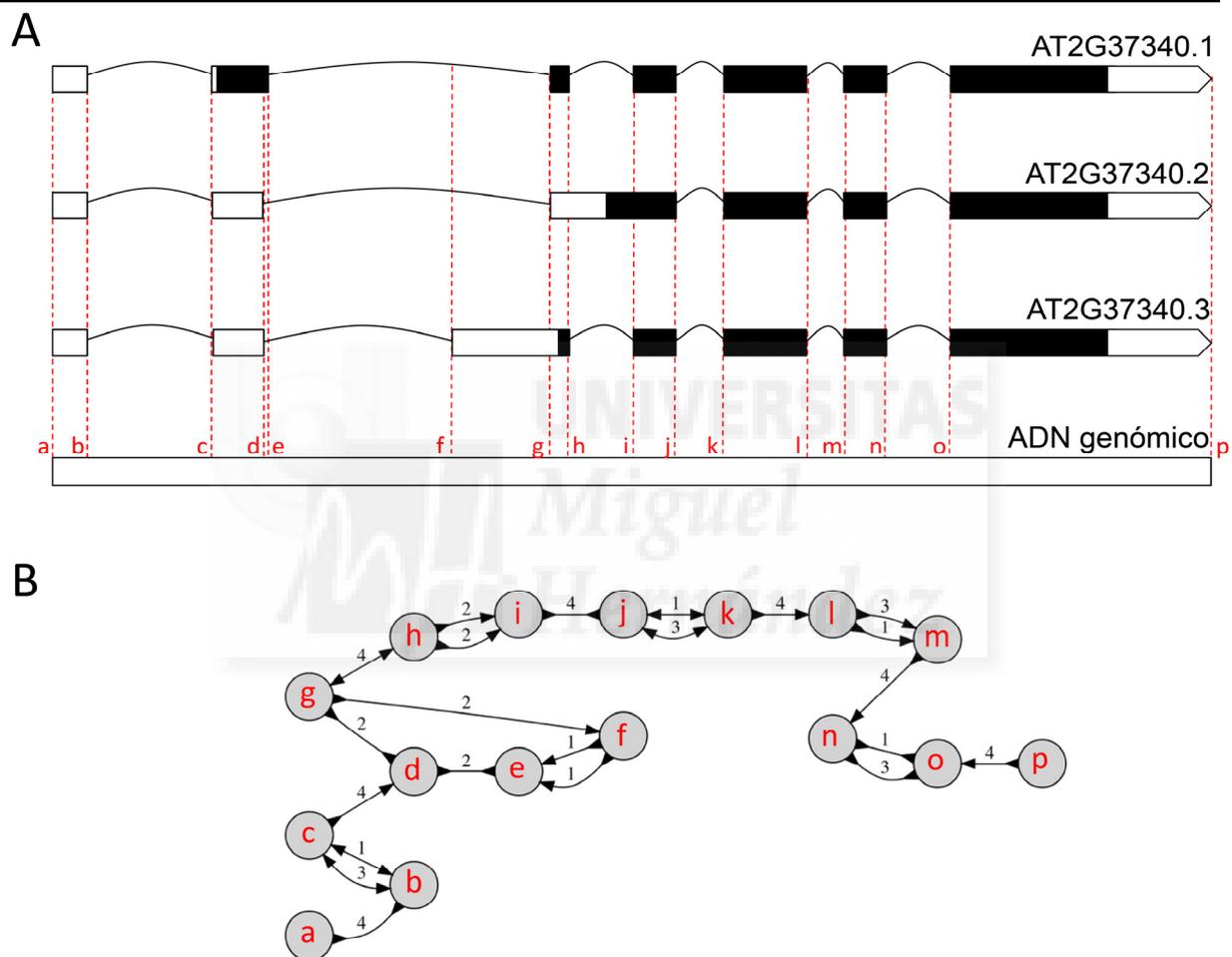
#### **5.4. Potencial del programa para la visualización de isoformas producidas mediante procesamiento alternativo de los intrones**

Para determinar si nuestro programa puede ser utilizado para detectar casos de procesamiento alternativo de los intrones (*splicing* alternativo), hemos realizado ensamblajes utilizando las secuencias genómica y de tres isoformas distintas del gen AT2G37340 de *Arabidopsis thaliana* (Figura 10A), que se describen en la base de datos del TAIR (*The Arabidopsis Information Resource*; <http://www.arabidopsis.org>). Este gen ha sido seleccionado porque sus isoformas ilustran tres situaciones distintas:

- a) Uso alternativo de sitios donantes del *splicing*. El segundo intrón se procesa de modo diferente en las isoformas AT2G37340.1 y AT2G37340.2. La diferencia estriba en el uso de diferentes sitios donantes en ambas isoformas, denominados *e* y *d* en la Figura 10A.
- b) Uso alternativo de sitios aceptores del *splicing*. El segundo intrón se procesa de modo diferente en las isoformas AT2G37340.2 y AT2G37340.3. La diferencia estriba en el uso de diferentes sitios aceptores en ambas isoformas, denominados *g* y *f* en la Figura 10A.
- c) Retención de un intrón. El tercer intrón de la isoforma AT2G37340.1 no se elimina en la isoforma AT2G37340.2. Esto se debe a la no utilización de los sitios *h* e *i* en esta última isoforma.

En la Figura 10B se muestra el grafo producido por nuestro programa a partir de las secuencias genómica y de las tres isoformas del ARN mensajero (ARNm), utilizando un

valor de  $k=41$ . Como se aprecia, este grafo ilustra perfectamente la estructura de las isoformas del gen. En esta representación los intrones se identifican fácilmente por la aparición de aristas adicionales. Por ejemplo, dos aristas diferentes ilustran las dos maneras posibles de conectar los sitios  $n$  y  $o$  del *splicing*: (1) a través de la secuencia del intrón (como sucede en el ADN genómico) y (2) conectando directamente las secuencias de los sitios donante y aceptor (como sucede en las tres isoformas del ARNm). Los diversos caminos que conectan los vértices  $d$  y  $g$  en el grafo permiten determinar, a simple vista, que el gen experimenta procesamiento alternativo de alguno de sus intrones.



**Figura 10.-** Representación mediante un grafo de la estructura de las isoformas del gen AT2G37340 de *Arabidopsis thaliana*. (A) Representación a escala de las tres isoformas (AT2G37340.1, AT2G37340.2 y AT2G37340.3). Las isoformas del ARNm se representan mediante flechas interrumpidas por intrones, que se han representado mediante arcos. La parte coloreada en negro corresponde a la región codificante (CDS; *coding sequence*). (B) Grafo producido por nuestro programa a partir de la secuencia del ADN genómico y las tres isoformas, obtenido para  $k=41$ . Para una mejor comprensión del grafo, hemos indicado mediante letras minúsculas de color rojo la correspondencia entre los vértices del grafo (representados en B) y los límites entre exones e intrones de las distintas isoformas (representadas en A).

## 6. Discusión

En este trabajo, hemos desarrollado un programa de ordenador que permite realizar el ensamblaje *de novo* de secuencias nucleotídicas. Por su flexibilidad, el programa ha sido implementado en Perl, un lenguaje de programación ampliamente utilizado en Bioinformática. Hemos aprovechado ampliamente las librerías de código (*libraries*) disponibles para este lenguaje de programación, que nos han permitido incorporar algunas prestaciones a nuestro programa de manera eficiente, como el uso de vectores de bits (`Bit::Vector`), la lectura de secuencias nucleotídicas en distintos formatos (`Bio::SeqIO`) o la especificación de opciones desde la línea de comandos (`Getopt::Long`). Una ventaja del lenguaje Perl es la portabilidad de los programas: los programas pueden ser ejecutados en cualquier sistema operativo siempre que haya sido instalado previamente un intérprete del lenguaje.

Al igual que otros programas similares, como Velvet (Zerbino y Birney, 2008), nuestro programa se estructura en dos subprogramas. El primero de ellos (`paso1.pl`), análogo al programa `velveth` de Velvet, se encarga de la lectura de los datos y de la representación de la secuencia mediante un grafo de De Bruijn, que se almacena en el disco duro para su reutilización posterior. El segundo subprograma (`paso2.pl`), análogo al programa `velvetg`, recupera el grafo del disco y realiza el ensamblaje de la secuencia. Este proceso se realiza visitando los vértices del grafo y recorriendo el grafo en ambas direcciones hasta que se alcanza una bifurcación. Cada uno de estos recorridos define inequívocamente la secuencia de un cóntigo. Como resultado de la ejecución del programa se obtienen dos archivos utilizando formatos estándar: uno que almacena la secuencia de los cóntigos (en formato FastA) y otro que almacena un grafo que describe los solapamientos existentes entre los mismos (en formato Dot).

Hemos optado por representar la secuencia mediante un grafo bidirigido de De Bruijn. En este tipo de grafo, cada vértice representa simultáneamente a una secuencia de longitud  $k$  y a su complementaria. Por este motivo, los grafos bidirigidos de De Bruijn constituyen una solución particularmente apropiada para representar secuencias derivadas de genomas de ADN bicatenario. Del mismo modo que la mayoría de programas que utilizan estrategias similares, nuestro programa exige que  $k$  sea un número impar, de manera que ninguna secuencia de longitud  $k$  puede ser idéntica a su complementaria. Como se describe en la Figura 2, hemos intentado representar el grafo de manera compacta. Para ello, nuestro programa almacena cada nucleótido utilizando únicamente 2 bits, ya que éste es el mínimo necesario para especificar cuatro estados diferentes (A, C, G y T). En el disco duro, cada arista del grafo se almacena como dos vectores de  $2k + 20$  bits. En el primero de estos vectores, los primeros  $2k$  bits se utilizan para representar la secuencia de  $k$  nucleótidos

asociada al vértice de origen. Por tratarse de un grafo de De Bruijn, el vértice de destino (*sink node*) puede representarse con sólo 2 bits adicionales, ya que los  $k-1$  nucleótidos iniciales de su secuencia están presentes en la secuencia del vértice de origen. La direccionalidad de cada arista se almacena en 2 bits (a razón de 1 bit por extremo). El programa utiliza los 16 bits restantes para almacenar el valor de cobertura asociado a la arista, por lo que la cobertura máxima admitida por el programa es de 65.535 ( $2^{16}-1$ ). En el segundo vector de bits, se intercambian los vértices de origen y de destino, lo que facilita que la arista se utilice en recorridos a través del grafo realizados en una u otra dirección.

Uno de los aspectos más importantes en cualquier ensamblaje *de novo* es la elección de valores adecuados para los parámetros del programa. El valor de  $k$  (la longitud de la secuencia representada por los vértices del grafo) resulta crítico para la obtención de un ensamblaje de calidad. El uso de valores de  $k$  grandes posee la ventaja de rendir grafos con menos ramificaciones, en los que el ensamblaje de la secuencia debe resultar más sencillo. Sin embargo, para una misma cobertura, cada incremento en el valor de  $k$  supone una mayor dificultad para identificar solapamientos de longitud suficiente entre las lecturas. Dado que nuestro programa ofrece la posibilidad de visualizar el grafo resultante, una característica única de nuestro programa es que permite evaluar a simple vista la calidad del ensamblaje obtenido, lo que facilita la elección del valor de  $k$  más idóneo para el ensamblaje de la secuencia.

Con este propósito, hemos realizado ensamblajes en condiciones ideales (en las que cada  $k$ -mero está representado en los datos de entrada tantas veces como aparece en la secuencia a ensamblar) de la secuencia de los genomas cloroplásticos de varias especies de plantas superiores. En estas condiciones, en las que la cobertura no limita el resultado obtenido, hemos comprobado en todos los casos que el número de vértices y aristas del grafo (nótese que el número de aristas del grafo equivale al número de contigios) se reduce progresivamente hasta alcanzar un valor mínimo. Conforme a nuestras expectativas, los grafos más sencillos obtenidos para los genomas del arroz y *Arabidopsis thaliana* constaron únicamente de tres aristas, que se correspondieron con precisión con las dos secuencias de copia única y la secuencia de las repeticiones invertidas de dichos genomas. Las etiquetas que el programa añade a las aristas indican claramente que los genomas completos contienen dos copias de esta última. El resultado obtenido para el maíz fue ligeramente más complejo, con un mayor número de aristas y vértices. El grafo obtenido indica que la secuencia representada por una de las aristas está presente en 3 copias. La longitud de dicha secuencia (171 pb) impone una cota inferior al valor de  $k$  que permitiría obtener un grafo más sencillo. De hecho, recientemente hemos comprobado que con  $k=173$  se obtiene un grafo de topología idéntica a los del arroz y *Arabidopsis thaliana* (resultado no mostrado). Este resultado permite ilustrar asimismo otro aspecto destacable de nuestro programa: la

ausencia de un límite superior al valor de  $k$  (a diferencia, por ejemplo, de Velvet, cuyo valor máximo es 81).

Hemos utilizado nuestro programa para evaluar el efecto de los errores en la determinación de la secuencia sobre la calidad del ensamblaje resultante. Con el fin de simular lecturas con errores, hemos escrito un programa auxiliar (`simulador.pl`) que genera lecturas a partir de una secuencia genómica suministrada en formato FastA. El programa genera lecturas de la longitud especificada, en las que se introducen errores al azar según la probabilidad de error deseada. Las simulaciones realizadas indican que tasas de error tan bajas como  $10^{-4}$  determinan un incremento dramático en el número de vértices y aristas (cóntigos) del grafo final. Hemos comprobado que este problema puede resolverse, al menos en ciertos casos, seleccionando únicamente el subgrafo compuesto por aristas cuya cobertura (multiplicidad) supera un umbral mínimo. El rango de cobertura a considerar puede especificarse fácilmente en el programa `paso2.pl` mediante las opciones `--mincov` y `--maxcov`.

Simulando experimentos de secuenciación de clones BAC con diferente cobertura, hemos comprobado que la relación entre el valor de  $k$  y la cobertura es un aspecto crítico que condiciona el éxito de los experimentos de secuenciación. A menor cobertura, es preciso utilizar un valor de  $k$  capaz de capturar los solapamientos existentes entre las lecturas. En caso contrario, el ensamblaje resultante se encuentra muy fragmentado. Nuestro programa podría utilizarse *a priori* en la planificación de experimentos de secuenciación, para seleccionar estos parámetros empíricamente.

La representación de las diferentes isoformas del ARNm de un gen mediante grafos de De Bruijn nos ha permitido identificar de manera precisa los límites entre intrones y exones. El grafo resultante refleja también con precisión las diferencias estructurales presentes en las distintas isoformas. Estas observaciones sugieren que los grafos de De Bruijn podrían utilizarse de manera sistemática para la detección de *splicing* alternativo en experimentos de transcriptómica.

## 7. Conclusiones y proyección futura

Hemos implementado un programa que permite ensamblar *de novo* secuencias de nucleótidos, siguiendo una estrategia basada en la representación de la secuencia mediante un grafo de De Bruijn bidirigido. Este programa ha sido escrito en lenguaje de programación Perl. Para reducir el espacio necesario para representar el grafo en la memoria del ordenador, el programa almacena las secuencias nucleotídicas utilizando 2 bits por nucleótido.

Utilizando nuestro programa, hemos evaluado el efecto de diversos parámetros sobre la calidad de los ensamblajes resultantes. Los parámetros que hemos considerado han sido: (a) el valor de  $k$ , (b) la cobertura (número promedio de veces que cada nucleótido del genoma ha sido secuenciado en el experimento), y (c) la tasa de error en la determinación de la secuencia.

Hemos determinado que, en condiciones ideales de cobertura, valores mayores de  $k$  conducen a ensamblajes con contigs más largos, que responden mejor a la estructura correcta del genoma.

Hemos comprobado que los errores de secuenciación conducen a un incremento dramático en la complejidad del grafo resultante. Este problema puede resolverse mediante la selección de las aristas del grafo que superan un umbral de cobertura.

Nuestros resultados indican que la relación entre el valor de  $k$  y la cobertura es un aspecto crítico a considerar en los experimentos de secuenciación. Si bien el uso de un valor alto de  $k$  potencialmente conduce a mejores ensamblajes, el resultado puede ser desastroso si se utiliza una cobertura insuficiente.

Consideramos que la representación de los diversos transcritos de un gen mediante un grafo de De Bruijn podría permitir identificar eficazmente nuevos casos de *splicing* alternativo.

Por último, el uso del programa en un contexto docente permitirá que los estudiantes de Bioinformática asimilen mejor conceptos relacionados con el ensamblaje *de novo* de genomas.

En cuanto a la proyección futura del trabajo, pretendemos implementar el programa en un lenguaje de programación diferente, que permita la ejecución más rápida del mismo. El programa en Perl podría ser fácilmente implementado para su utilización a través de un formulario web, utilizando módulos adecuados para ello (CGI).

## 8. Bibliografía

- Bentley, D.R. (2006). Whole-genome re-sequencing. *Curr. Opin. Genet. Dev.* **16**: 545-552.
- Drmanac, R., Labat, I., Brukner, I. y Crkvenjakov, R. (1989). Sequencing of megabase plus DNA by hybridization: Theory and the method. *Genomics* **4**: 114-128.
- Drmanac, R., Drmanac, S., Strezoska, Z., Paunesku, T., Labat, I., Zeremski, M., Snoddy, J., Funkhouser, W.K., Koop, B., Hood, L., *et al.* (1993). DNA sequence determination by hybridization: a strategy for efficient large-scale sequencing. *Science* **260**: 1649-1652.
- Drmanac, R., Drmanac, S., Chui, G., Diaz, R., Hou, A., Jin, H., Jin, P., Kwon, S., Lacy, S., Moeur, B., Shafto, J., Swanson, D., Ukrainczyk, T., Xu, C. y Little, D. (2002). Sequencing by hybridization (SBH): advantages, achievements, and opportunities. *Adv. Biochem. Eng. Biotechnol.* **77**: 75-101.
- Gansner, E., Koutsofios, E. y North, S. (2006). Drawing graphs with dot. Manual de usuario del programa.
- Goldberg, S.M., Johnson, J., Busam, D., Feldblyum, T., Ferriera, S., Friedman, R., Halpern, A., Khouri, H., Kravitz, S.A., Lauro, F.M., Li, K., Rogers, Y., Strausberg, R., Sutton, G., Tallon, L., Thomas, T., Venter, E., Frazier, M., y Venter, J.C. (2006). A Sanger/pyrosequencing hybrid approach for the generation of high-quality draft assemblies of marine microbial genomes. *Proc. Natl. Acad. Sci. USA* **103**: 11240-11245.
- Idury, R.M. y Waterman, M.S. (1995). A new algorithm for DNA sequence assembly. *J. Comput. Biol.* **2**: 291-306.
- Lander, E.S., Linton, L.M., Birren, B., Nusbaum, C., Zody, M.C., Baldwin, J., Devon, K., Dewar, K., Doyle, M., FitzHugh, W., *et al.* (2001). Initial sequencing and analysis of the human genome. *Nature* **409**: 860-921.
- Langmead, B. (2010). Aligning short sequencing reads with Bowtie. *Curr. Protoc. Bioinformatics* **11**: 11.7.
- Li, H. y Durbin, R. (2009). Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* **15**: 1754-1760.
- Li, H. y Durbin, R. (2010). Fast and accurate long read alignment with Burrows-Wheeler transform. *Bioinformatics* **26**: 589-595.
- Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J. y Wang, J. (2010). De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.* **20**: 265-272.

- Mardis, E.R. (2008). The impact of next-generation sequencing technology on genetics. *Trends Genet.* **24**: 133-141.
- Margulies, M., Egholm, M., Altman, W.E., Attiya, S., Bader, J.S., Bemben, L.A., Berka, J., Braverman, M.S., Chen, Y., Chen, Z., Dewell, S.B., Du, L., Fierro, J.M., Gomes, X.V., Goodwin, B.C., He, W., Helgesen, S., He Ho, C., Irzyk, G.P., Jando, S.C., Alenquer, M.L.I., Jarvie, T.P., Jirage, K.B., Kim, J.B., Knight, J.R., Lanza, J.R., Leamon, J.H., Lefkowitz, S.M., Lei, M., Li, J., Lohman, K.L., Lu, H., Makhijani, V.B., McDade, K.E., McKenna, M.P., Myers, E.W., Nickerson, E., Nobile, J.R., Plant, R., Puc, B.P., Ronan, M.T., Roth, G.T., Sarkis, G.J., Simons, J.F., Simpson, J.W., Srinivasan, M., Tartaro, K.R., Tomasz, A., Vogt, K.A., Volkmer, G.A., Wang, S.H., Wang, Y., Weiner, M.P., Yu, P., Begley, R.F., y Rothberg, J.M. (2005). Genome sequencing in open microfabricated high density picoliter reactors. *Nature* **437**: 376-380.
- Metzker, M.L. (2010). Sequencing technologies- the next generation. *Nat. Rev. Genet.* **11**: 31-46.
- Miller, J.R., Koren, S. y Sutton, G. (2010). Assembly algorithms for next-generation sequencing data. *Genomics* **95**: 315-327.
- Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., Flanigan, M.J., Kravitz, S.A., Mobarry, C.M., Reinert, K.H., Remington, K.A., Anson, E.L., Bolanos, R.A., Chou, H.H., Jordan, C.M., Halpern, A.L., Lonardi, S., Beasley, E.M., Brandon, R.C., Chen, L., Dunn, P.J., Lai, Z., Liang, Y., Nusskern, D.R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G.M., Adams, M.D. y Venter, J.C. (2000). A whole-genome assembly of *Drosophila*. *Science* **287**: 2196-2204.
- Pareek, C.S., Smoczynski, R. y Tretyn, A. (2011). Sequencing technologies and genome sequencing. *J. Appl. Genet.* **52**: 413-435.
- Pettersson, E., Lundeberg, J. y Ahmadian, A. (2009). Generations of sequencing technologies. *Genomics* **93**: 105-111.
- Pevzner, P.A., Tang, H. y Waterman, M.S. (2001). An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA* **98**: 9748-9753.
- Ronaghi, M., Karamohamed, S., Pettersson, B., Uhlen, M. y Nyren, P. (1996). Real-time DNA sequencing using detection of pyrophosphate release. *Anal. Biochem.* **242**: 84-89.
- Ruffalo, M., LaFramboise, T. y Koyutürk, M. (2011). Comparative analysis of algorithms for next-generation sequencing read alignment. *Bioinformatics* **27**: 2790-2796.
- Sanger, F., Nicklen, S. y Coulson, A.R. (1977). DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. USA* **74**: 5463-5467.

- Schulz, M.H., Zerbino, D.R., Vingron, M. y Birney, E. (2012). Oases: Robust de novo RNA-seq assembly across the dynamic range of expression levels. *Bioinformatics* **28**: 1086-1092.
- Schuster, S.C. (2007). Next-generation sequencing transforms today's biology. *Nat. Methods* **5**: 16-18.
- Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J. y Birol, I. (2009). ABySS: a parallel assembler for short read sequence data. *Genome Res.* **19**: 1117-1123.
- Tucker, T., Marra, M. y Friedman, J.M. (2009). Massively parallel sequencing: the next big thing in genetic medicine. *Am. J. Hum. Genet.* **85**: 142-154.
- Venter, J.C., Adams, M.D., Myers, E.W., Li, P.W., Mural, R.J., Sutton, G.G., Smith, H.O., Yandell, M., Evans, C.A., Holt, R.A., *et al.* (2001). The sequence of the human genome. *Science* **291**: 1304-1351.
- Xie, Y., Wu, G., Tang, J., Luo, R., Patterson, J., Liu, S., Huang, W., He, G., Gu, S., Li, S., Zhou, X., Lam, T.W., Li, Y., Xu, X., Wong, G.K., y Wang, J. (2014). SOAPdenovo-Trans: de novo transcriptome assembly with short RNA-Seq reads. *Bioinformatics* **30**: 1660-1666.
- Zerbino, D.R. y Birney, E. (2008). Velvet: algorithms for de novo short read assembly using *de Bruijn* graphs. *Genome Res.* **18**: 821-829.
- Zhang, J., Chiodini, R., Badr, A. y Zhang, G. (2011). The impact of next-generation sequencing on genomics. *J. Genet. Genomics* **38**: 95-109.
- Zhang, J.H., Wu, I.Y. y Zhang, X.S. (2003). Reconstruction of DNA sequencing by hybridization. *Bioinformatics* **19**: 14-21.

## 9. Apéndice: Programas de ordenador

### 9.1. paso1.pl

```

package Assembler;
use strict;
use warnings;
use autodie;
use Bit::Vector;
use Bio::SeqIO;
use Getopt::Long;
use libreria;

my $infile;
my $outfile;
my $k = 35;    # kmer length

my $commandline = join ' ', $0, @ARGV;
open my $log_fh, '>>', 'assembler.log';
print {$log_fh} "$commandline\n";
close $log_fh;

if (!GetOptions ('kmer=i'=>\$k, 'input=s' => \$infile, 'output=s' =>\$outfile) or $infile eq "" or $outfile eq "")
{
    usage();
}

if ($k%2 == 0) {
    $k++;
}

my $seqio_object = Bio::SeqIO->new(-file => $infile,
    -format => 'fasta');

open my $fh_output, '>', $outfile;

while (my $seq_object = $seqio_object->next_seq) {
    my $read = $seq_object->seq;
    #Descomponemos la lectura en k-meros de longitud k
    for my $pos (0 .. (length $read) - $k - 1) {
        my $subsequence_1 = substr $read, $pos, $k;
        my $RC_1 = revcom($subsequence_1);
        my ($print_1, $arrow1);
        if ($subsequence_1 lt $RC_1) {
            $print_1 = $subsequence_1;
            $arrow1 = '0';
        } else {
            $print_1 = $RC_1;
            $arrow1 = '1';
        }
        my $subsequence_2 = substr $read, $pos+1, $k;
        my $RC_2 = revcom($subsequence_2);
        my ($print_2, $arrow2);
        if ($subsequence_2 lt $RC_2) {
            $print_2 = $subsequence_2;
            $arrow2 = '1';
        } else {

```

```

    $print_2 = $RC_2;
    $arrow2 = '0';
}
    my $bin_1 = to_binary ($print_1);
    my $bin_2 = to_binary ($print_2);
my ($output1, $output2);
    if ($arrow2 eq '0'){
        $output1 = $bin_1 . $arrow1 . '0' . to_binary(revcom (substr $print_2, 0 , 1));
    } else {
        $output1 = $bin_1 . $arrow1 . '1' . to_binary(substr $print_2, $k-1,1);
    }
    if ($arrow1 eq '0'){
        $output2 = $bin_2 . $arrow2 . '0' . to_binary(revcom (substr $print_1, 0 , 1));
    }else{
        $output2 = $bin_2 . $arrow2 . '1' . to_binary(substr $print_1, $k-1, 1);
    }
    if ($output1 ne $output2) {
        print {$fh_output} $output1."\n";
        print {$fh_output} $output2."\n";
    } else {
        print {$fh_output} $output1."\n";
    }
}
}

close $fh_output;
my $result = `sort -k1,1 $outfile | uniq -c > resultado.tmp`;

my $pattern = 'B' . ($k * 2 + 20);

open my $fh, '<', 'resultado.tmp';
open my $file_out, '>', 'output_final.bin';
binmode $file_out;

while (my $linea = <$fh>) {
    chomp $linea;
    if ($linea =~ m/(S+)\s(S+)/) {
        my $number = $2 . to_number($1, 8);
        print {$file_out} pack $pattern, $number;
    }
}

close $fh;
close $file_out;

sub usage
{
    print "Unknown option: @_ \n" if (@_);
    print "\n$0:\n";
    print "This program prepares a bidirected de Bruijn graph from a set of read sequences.\n\n";
    print "Mandatory options:\n";
    print " --input      STRING  Fasta file with reads\n";
    print " --output      STRING  Name for intermediate files\n";
    print " --kmer        INT     k-mer length\n";
    exit;
}

```

## 9.2. paso2.pl

```

package Assembler;
use strict;
use warnings;
use autodie;
use Bit::Vector;
use Getopt::Long;
use libreria;

my $k = 35;
my $mincov = 0;
my $maxcov = 999999;
my %graph;
my %hash;
my $output;
my $contig_number = 0;

my $commandline = join ' ', $0, @ARGV;
open my $log_fh, '>>', 'assembler.log';
print {$log_fh} "$commandline\n";
close $log_fh;

if (!GetOptions ('kmer=i'=>\$k, 'mincov=i'=>\$mincov, 'maxcov=i'=>\$maxcov, 'output=s'=>\$output) or $output eq "")
{
    usage();
}

if ($k%2 == 0) {
    $k++;
}

my $datos;
my $longitud = $k*2 + 20;
my $longitud_seq = $k*2;
my $patron = 'B'. $longitud;

my $total_bytes = int ($longitud / 8);
if ($longitud % 8 != 0){
    $total_bytes++;
}

open my $datosin, '<', 'output_final.bin';
binmode $datosin;

while(my $nbytes = read $datosin, $datos, $total_bytes){
    my $sequencebin = unpack $patron,$datos;
    my $seq = substr $sequencebin, 0, $longitud_seq;
    my $arrow = substr $sequencebin, $longitud_seq, 4;
    my $multi = substr $sequencebin, $longitud_seq +4, 16;
    if ($multi >= $mincov && $multi <= $maxcov) {
        $graph{$seq}{oct "0b$arrow"}{multiplicity}=oct "0b$multi";
        $graph{$seq}{oct "0b$arrow"}{visited}=0;
    }
}

close $datosin;

```

```

# Opens new file for graph
open my $graph, '>', "$output.dot";

# Prints header for graph file
print {$graph} "strict digraph $output {\n";
print {$graph} "  node [shape=ellipse, style=filled];\n";
print {$graph} "  edge [dir=both];\n";

foreach my $anterior (keys %graph) {
  foreach my $dato_anterior (keys %{$graph{$anterior}}) {
    if ($graph{$anterior}{$dato_anterior}{visited}==0){
      # $graph{$anterior}{$dato_anterior}{visited}=1;
      # my ($a,$b)=alternate_edge($anterior,$dato_anterior, $k);
      # $graph{$a}{$b}{visited}=1;
      $contig_number++;
      my ($first, $first_arrow, $last, $last_arrow, $cover, $contig) = assemble ($anterior, $dato_anterior, $k, \%graph);
      print '>'. $contig_number. "\t". to_nucleotides($first). "\t". $first_arrow. "\t". $last_arrow. "\t". to_nucleotides($last). "\t". $cover. "\n";
      $hash{to_nucleotides($first)}{to_nucleotides($last)} += 1;
      print $contig. "\n";
      my ($arrowtail, $arrowhead);
      if ($first_arrow==0){$arrowtail = 'inv';} else {$arrowtail = 'normal';}
      if ($last_arrow==0){$arrowhead = 'inv';} else {$arrowhead = 'normal';}
      if ($hash{to_nucleotides($first)}{to_nucleotides($last)} > 1 ){
        print {$graph} '      '. to_nucleotides($last). ' -> '. to_nucleotides($first). " [arrowhead=$arrowtail arrowtail=$arrowhead
label=".$cover."];\n";
      } else {
        print {$graph} '      '. to_nucleotides($first). ' -> '. to_nucleotides($last). " [arrowhead=$arrowhead arrowtail=$arrowtail
label=".$cover."];\n";
      }
    }
  }
}
print {$graph} "\n";
close $graph;

sub usage
{
  print "Unknown option: @_ \n" if (@_);
  print "\n$0:\n";
  print "This program uses a bidirected de Bruijn graph to assemble the reads. Two files are produced: a fasta \n";
  print "file with the contigs and a dot file with the graph.\n\n";
  print "Mandatory:\n";
  print " --out      STRING Base name for graph file\n";
  print " --kmer     INT      k-mer length\n";
  exit;
}

```

### 9.3. libreria.pm

```

package Assembler;
use strict;
use warnings;
use Bit::Vector;

#Returns the reverse complement of a nucleotide sequence
sub revcom {
    my $sequence = shift;
    $sequence = reverse $sequence;
    $sequence =~ tr/ATCG/TAGC/;
    return $sequence;
}

#Converts a sequence to a binary value
sub to_binary {
    my $sequence = shift;
    $sequence =~ s/A/00/g;
    $sequence =~ s/T/11/g;
    $sequence =~ s/G/10/g;
    $sequence =~ s/C/01/g;
    return $sequence;
}

sub to_number {
    my $number = shift;
    my $kmer_length = shift;
    my $valor = Bit::Vector->new_Dec ($kmer_length*2, $number);
    $number = $valor->to_Bin();
    return $number;
}

#Converts a bit vector to a nucleotide sequence
sub to_nucleotides {
    my $sequence = "";
    my $number = shift;
    while ($number =~ s/^(1|0)/) {
        if ($1 eq '00') {
            $sequence = $sequence.'A';
        } elsif ($1 eq '01') {
            $sequence = $sequence.'C';
        } elsif ($1 eq '10') {
            $sequence = $sequence.'G';
        } else {
            $sequence = $sequence.'T';
        }
    }
    return $sequence;
}

#Returns the direction of arrow at the beginning of an edge
sub beginning {
    my $number = shift;
    if ($number & 8) {
        return 1;
    } else {

```

```

    return 0;
}
}

#Returns the direction of arrow at the end of an edge
sub end {
    my $number = shift;
    if ($number & 4) {
        return 1;
    } else {
        return 0;
    }
}

#Determines the base to add while moving from a node to the next
sub next_base {
    my $number = shift;
    my $base = $number & 3;
    if ($base == 0) {
        return 'A';
    } elsif ($base == 1) {
        return 'C';
    } elsif ($base == 2) {
        return 'G';
    } else {
        return 'T';
    }
}

sub assemble {
    my ($source_node, $source_edge_info, $kmer_length, $graph_ref) = @_;
    my ($first, $last, $first_arrow, $last_arrow);
    my $multiply = 0;
    my $nedges = 0;
    my $assembly_2;
    my $current_node = $source_node;
    my $current_edge_info = $source_edge_info;
    my $current_node_sequence = to_nucleotides ($current_node);
    my $beginning = beginning ($current_edge_info);

    if ($beginning == 1) {
        $current_node_sequence = revcom ($current_node_sequence);
    }

    my $assembly = $current_node_sequence;

    while ( defined $graph_ref->{$current_node}{$current_edge_info}){
        my $current_node_sequence = to_nucleotides ($current_node);
        my $next_base = next_base ($current_edge_info);
        $assembly = $assembly . $next_base;

        $graph_ref->{$current_node}{$current_edge_info}{visited}=1;
        my ($a,$b)=alternate_edge($current_node,$current_edge_info, $kmer_length);
        # print $a, $b, "\n";
        $graph_ref->{$a}{$b}{visited}=1;

        $nedges++;
        $multiply = $multiply + $graph_ref->{$current_node}{$current_edge_info}{multiplicity};
    }
}

```

```

my $end = end ($current_edge_info);
my $beginning = beginning ($current_edge_info);
if ($beginning == 1) {
    $current_node_sequence = revcom ($current_node_sequence);
}

my $sink_node_sequence = substr ($current_node_sequence, 1,$kmer_length-1) . $next_base;

my $revcom_sink = revcom ($sink_node_sequence);
if ($sink_node_sequence gt $revcom_sink) {
    $sink_node_sequence = $revcom_sink;
}
my $sink_node = to_binary ($sink_node_sequence);

if (defined $graph_ref->{$sink_node}){
    my $counter=0;
my $counter2=0;
    my $sink_edge_info;
    foreach my $value (keys %{$graph_ref->{$sink_node}}){
my $beginning = beginning($value);
        if ($beginning != $end){
            $counter++;
            $sink_edge_info = $value;
        }

my $newedge;
        if ($beginning == 1){
            $newedge = revcom($sink_node_sequence) . next_base($value);
        } else {
            $newedge = $sink_node_sequence . next_base($value);
        }
        if(substr($newedge,0,$kmer_length) eq revcom(substr $newedge,1,$kmer_length)){
            $counter2=$counter2+2;
        } else{
            $counter2++;
        }
    }

    if ($counter ==1 && $counter2 == 2 && $graph_ref->{$sink_node}{$sink_edge_info}{visited}==0){
$current_node = $sink_node;
        $current_edge_info = $sink_edge_info;
    } else{
$first = $sink_node;
        $first_arrow = end($current_edge_info);
        last;
    }
}

($current_node, $current_edge_info) = alternate_edge($source_node, $source_edge_info, $kmer_length);

while (defined $graph_ref->{$current_node}{$current_edge_info}){
    my $current_node_sequence = to_nucleotides ($current_node);
    my $next_base = next_base ($current_edge_info);
    $assembly_2 = $assembly_2 . $next_base;

    $graph_ref->{$current_node}{$current_edge_info}{visited}=1;
}

```

```

my ($a,$b)=alternate_edge($current_node,$current_edge_info, $kmer_length);
$graph_ref->{$a}{$b}{visited}=1;

$nedges++;
$multip = $multip + $graph_ref->{$current_node}{$current_edge_info}{multiplicity};

my $end = end ($current_edge_info);
my $beginning = beginning ($current_edge_info);
  if ($beginning == 1) {
    $current_node_sequence = revcom ($current_node_sequence);
  }
my $sink_node_sequence = substr ($current_node_sequence, 1,$kmer_length-1) . $next_base;
my $revcom_sink = revcom ($sink_node_sequence);
if ($sink_node_sequence gt $revcom_sink) {
  $sink_node_sequence = $revcom_sink;
}
my $sink_node = to_binary ($sink_node_sequence);

if (defined $graph_ref->{$sink_node}){
  my $counter =0;
  my $counter2 =0;
  my $sink_edge_info;
  foreach my $value (keys %{$graph_ref->{$sink_node}}){
    my $beginning = beginning($value);
    if ($beginning != $end){
      $counter++;
      $sink_edge_info = $value;
    }
  }
  my $newedge;
  if ($beginning == 1){
    $newedge = revcom($sink_node_sequence) . next_base($value);
  } else {
    $newedge = $sink_node_sequence . next_base($value);
  }
}
if(substr $newedge,0,$kmer_length eq revcom(substr $newedge,1,$kmer_length)){
  $counter2=$counter2+2;
} else{
  $counter2++;
}
}
if ($counter == 1 && $counter2 == 2 && $graph_ref->{$sink_node}{$sink_edge_info}{visited}==0){
  $current_node = $sink_node;
  $current_edge_info = $sink_edge_info;
} else {
  $last=$sink_node;
  $last_arrow = end($current_edge_info);
  last;
}
}
}

return $first, $first_arrow, $last, $last_arrow, $multip/$nedges, revcom($assembly).substr($assembly_2, 1, length($assembly_2) - 1);
}

sub alternate_edge {
  my ($current_node, $current_edge_info, $kmer_length) = @_;
  my $current_node_sequence = to_nucleotides ($current_node);
  my $beginning = beginning ($current_edge_info);
  if ($beginning == 1) {

```

```

    $current_node_sequence = revcom ($current_node_sequence);
}
my $otherway_node_sequence = substr ($current_node_sequence, 1,($kmer_length-1)) . next_base ($current_edge_info);
my $otherway_edge_info;
my $revcom_otherway = revcom ($otherway_node_sequence);
if ($otherway_node_sequence gt $revcom_otherway) {
    $otherway_node_sequence = $revcom_otherway;
}
$beginning = end($current_edge_info);
my $end = beginning($current_edge_info);
if ($end == 0) {
    $otherway_edge_info = $beginning.'0'.to_binary(revcom(substr to_nucleotides($current_node), 0 , 1));
} else {
    $otherway_edge_info = $beginning.'1'.to_binary (substr to_nucleotides($current_node), $kmer_length-1,1);
}
return to_binary($otherway_node_sequence), oct "0b$otherway_edge_info";
}
1;

```



## 9.4. simulador.pl

```
#!/usr/bin/env perl
use warnings;
use strict;
use Bio::Perl;
use Bio::SeqIO ;
my $file = shift;
my $lonlectura = shift;
my $probability = shift;
my $cobertura = shift;
my $seqio_object = Bio::SeqIO->new(-file => $file,
                                   -format => 'fasta');
my $seq_object = $seqio_object->next_seq;
my $seq = $seq_object->seq;
my $lon = length($seq);
my $nlecturas = int($cobertura * $lon / $lonlectura);
$seq = $seq x 2;
for (my $i=1; $i<=$nlecturas; $i++){
    my $pos = int(rand($lon)) . "\n";
    my $subseq = substr($seq,$pos,$lonlectura);

    # for each position
    for (my $j=0; $j<=$lonlectura-1; $j++){
        if (rand(1)<=$probability) {
            my $nuc = substr($subseq,$j,1);
            my $newnuc = elegir($nuc);
            substr($subseq,$j,1,$newnuc);
        }
    }
    print ">$i\n";
    print $subseq . "\n";
}

sub elegir {
    my $nuc = shift;
    if ($nuc eq 'A') {
        my @letters = ("C","T","G");
        $nuc = $letters[int(rand(3))];
    } elsif ($nuc eq 'T') {
        my @letters = ("C","A","G");
        $nuc = $letters[int(rand(3))];
    } elsif ($nuc eq 'C') {
        my @letters = ("A","T","G");
        $nuc = $letters[int(rand(3))];
    } else {
        my @letters = ("C","T","A");
        $nuc = $letters[int(rand(3))];
    }
}
return $nuc;
}
```