UNIVERSIDAD IBEROAMERICANA

Estudios con Reconocimiento de Validez Oficial por Decreto Presidencial

del 3 de abril de 1981

"ENSAMBLE DE FRAGMENTOS DE ADN USANDO MÉTODOS BASADOS EN TEORÍA DE GRAFOS"

TESIS

Que para obtener el grado de

DOCTOR EN CIENCIAS DE LA INGENIERÍA

P r e s e n t a

JOSE EMILIO QUIROZ IBARRA

Directores: Dr. Guillermo Manuel Mallén Fullerton, Dr. Guillermo Fernández Anaya

Lector: Dr. Javier García García

Ciudad de México 2018

Tabla de contenido

1. Resumen 2
2. Introducción 4
3. Hipótesis y Objetivos 10
4. Fundamentación y Estado del Arte 11
5. Metodología 32
6. Conclusiones y Trabajos futuros 52
7. Referencias bibliográficas 55
8. Anexos 58

1. Resumen

El problema de ensamble de fragmentos de ADN[1], conocido como FAP (Fragment Assembly Problem por sus siglas en ingles), para la determinación de un genoma conlleva un importante problema computacional derivado de la precisión requerida al manipular los fragmentos resultantes del proceso de secuenciación, así como por el alto volumen de datos que se obtiene de los equipos de secuenciación de ADN y finalmente por la forma misma de los datos secuenciados. La generación de un nuevo algoritmo y sus estructuras de datos, capaces de producir resultados en tiempos competitivos, con consumo de recursos computacionales razonables y con cierta garantía de calidad en el genoma resultante es el reto del problema de ensamble de fragmentos de ADN y de el proyecto de investigación que en esta tesis se reporta. En los pasos del proceso de este trabajo de investigación, primero se determinan los traslapes entre fragmentos de ADN, construyendo una estructura de datos del tipo árbol de prefijos, ya que los fragmentos de las muestras de ADN se replican en diferentes posiciones y esto da oportunidad a determinar que una secuencia de bases de un fragmento coincida con otra secuencia de otro fragmento, obteniéndose con esto un sistema de fragmentos traslapados; con estos valores es posible construir un grafo dirigido o dígrafo, el cuál es manejado mediante una estructura de datos del tipo lista de adyacencia. De esta lista, se desarrolla una técnica de recorridos del dígrafo, con la particularidad de localizar los caminos más largos, obteniéndose con esto una serie de fragmentos contiguos, mismos que se identifican como *contigs*, con atributos de calidad que permiten garantizar que el *contig* está ensamblado con valores certeros. El resultado final, es decir los *contigs*, se comparan con los resultados de los ensambladores comerciales Velvet y Edena a partir de la localización de estos *contigs* en un genoma ya documentado y buscando de esta forma un resultado competitivo con respecto a Edena y Velvet. El proceso de desarrollo, pruebas y comparativos se lleva a cabo con la bacteria *Staphylococcus Aureus*, ya que se cuenta con mucha documentación al respecto, aunque también se hicieron algunas pruebas aisladas con la bacteria *Rhodobacteria Sphaeroides*.

[1] *ADN*: Acido Desoxirribonucleico.

2. Introducción

2.1 Qué es el ADN y cómo es el proceso de secuenciación

Todos los seres vivos están compuestos por células; algunos seres vivos u organismos son de una sola célula (unicelular) y otros son de múltiples células (pluricelular). Las células agrupadas forman tejidos y eso es aplicable tanto a animales como a plantas. Cuando se agrupan diferentes tipos de tejidos, se obtiene un órgano, el cuál cumplirá con una función específica en el ser vivo; varios órganos forman un sistema orgánico, como el sistema respiratorio o el sistema digestivo.

En 1838-1839 Schleiden y Schwann proponen la Teoría Celular (Salamanca Gomez, 1962) en la cual se describe esta composición de células para la construcción de tejidos, órganos y sistemas. Todas las células provienen en realidad de otras células por lo que el proceso de reproducción celular es la base de generación de la vida, así como la información que debe heredar una célula a otra durante su generación o reproducción.

Los organismos pueden ser del tipo Procariontes y Eucariontes. Los procariontes son organismos unicelulares sin núcleo, aunque presentan un nucleoide. Los eucariontes son pluricelulares y las células contiene un núcleo verdadero (Curtis, Barnes, Schnek, & Flores, 2006, págs. 20-29). El nucleoide es una región similar a un núcleo, aunque es de forma irregular; en esta se encuentra la información del ADN en el caso de procariontes. El núcleo, propio de organismos eucariontes, tiene una membrana nuclear y el ADN se guarda dentro de un orgánulo. La clasificación sistémica de los organismos[2] parte de los procariontes y los eucariontes. Los procariontes son arqueas y bacterias. Estas células se adaptan al medio ambiente y ensayan diferentes posibles mecanismos para realizar su metabolismo. Eucariontes son todos los animales y vegetales; los eucariontes provienen de los procariontes.

La célula es una masa de protoplasma (citoplasma) rodeado por una membrana y con un núcleo en el interior. En el núcleo están los cromosomas,

[2] La Clasificación Sistémica de la biología se encarga de agrupar y ordenar las especies a partir de su historia evolutiva.

que contiene los genes o unidades de la herencia. Los cromosomas están constituidos por fibras de acido desoxirribonucleico (ADN). Los genes son los responsables de las características estructurales de la célula y por lo mismo, de las características físicas del ser vivo, es decir, el fenotipo. El conjunto de genes de un ser vivo constituye el genotipo, es decir, la clase de la que un organismo es miembro y por extensión su genoma (Barbadilla, s.f.).

La molécula del ADN se compone de pentosas (un tipo de azúcar) y fosfatos de los que pende una base nitrogenada unida a otha hebra de ADN mediante puentes de hidrógeno; las bases nitrogenadas son la adenina (A), timina (T), citosina (C) y la guanina (G). Estas bases tienen átomos de nitrógeno y son del tipo purinas (A, G) y pirimidinas (T, C) (figura 1). La cantidad de pirimidinas es igual al de purinas, es decir A+G =T+C y además A de coecta

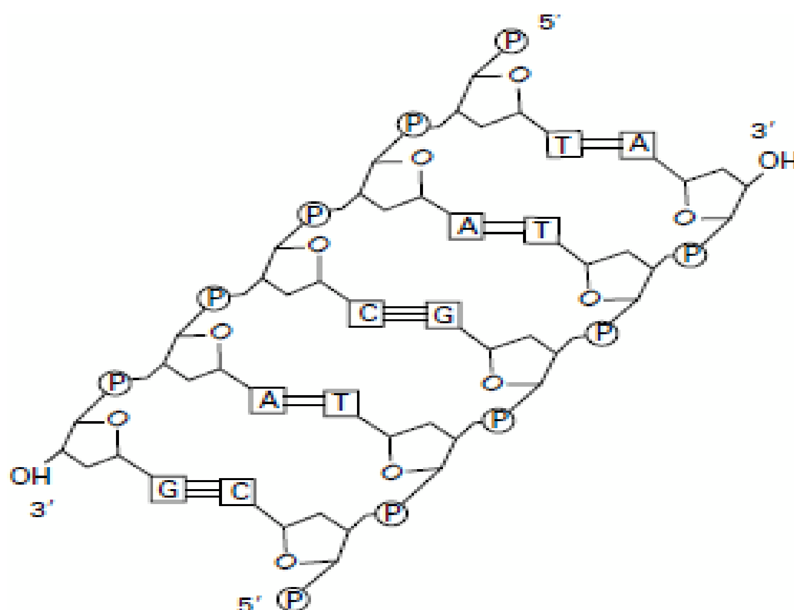


Figura 1. Estructura química del ADN.

Tomado de (Kingston, 2002, pág. fig. 15.1)

siempre con T y C con G (Salamanca Gomez, 1962, págs. 10-12).

En 1953 James Watson y Francis Crick (Watson & Crick, 1953) descubrieron que la molécula del ADN tiene una estructura espacial de doble hélice. Esta doble cadena esta enrollada en un eje a todo lo largo, como una escalera en espiral, en la que los peldaños se unen por pares de bases

nitrogenadas en oposición, es decir A con T y C con G. La dirección de la cadena está determinada por los átomos de carbón 5' y 3' de la pentosa, que indican el inicio y el fin respectivamente. Las hebras corren complementariamente, por lo que, al recorrer la hebra en forma inversa, se tienen los pares complementarios (figura 2).

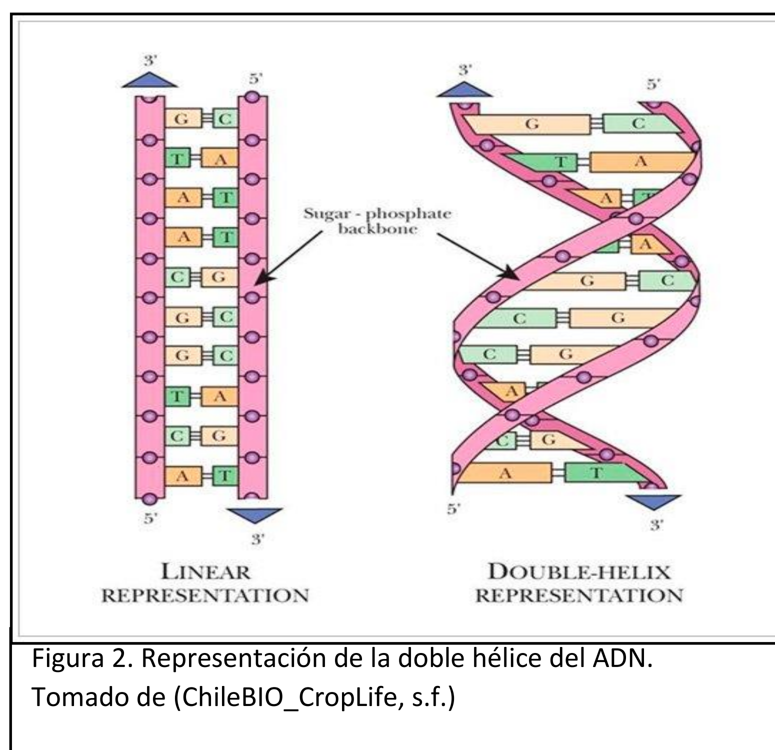


Figura 2. Representación de la doble hélice del ADN. Tomado de (ChileBIO_CropLife, s.f.)

En 1975 Frederik Sanger (Sanger & Coulson, DNA sequencing with chain-terminating inhibitors, 1977) propuso una técnica para secuenciar[3] el ADN que consistía en detectar bandas oscuras en una fina capa de gel, utilizando electroforesis[4]. Sanger propuso cortar la molécula del ADN en puntos específicos de la secuencia utilizando enzimas de restricción (Gingold, 1988). Este método resultaba lento y costoso. Para reducir el tiempo Sanger et al. (Myers Jr, 2016); (Sanger F. , Coulson, Hong, Hill, & Petersen, 1982) propusieron dividir las secuencias en puntos aleatorios, con la desventaja de que este método genera un orden incierto de los fragmentos. Esta técnica,

[3] Secuenciación: proceso para determinar la secuencia de los nucleótidos A, C, G, T en un fragmento de ADN.
[4] Electroforesis: consiste en aplicar una corriente eléctrica a través de un gel que contiene las moléculas, mismas que se desplazan en diferentes direcciones y velocidades en base a su tamaño y carga.

aunque es rápida, genera un problema computacional severo por el desconocimiento del orden de los fragmentos. A este método se le conoce como *Técnica de Shotgun* (Tammi, 2003).

2.2 Problemática computacional

Dado que no se conoce el orden de los fragmentos lo que se hace para la construcción del genoma es generar muchas copias del mismo problema secuenciando los fragmentos obtenidos, es decir, cortando cada copia en diferentes sitios y obteniendo como resultado que una misma base se repetirá en varios fragmentos. La suma de la longitud de los fragmentos dividido por la longitud del problema original es la cobertura. Si se tiene una cobertura suficientemente larga, los fragmentos se traslaparán en los extremos y por lo tanto debería de ser posible, con el resultado de todos los traslapes ordenados, reconstruir la secuencia original del ADN.

Pero en la realidad esto no es tan simple, ya que pueden presentarse traslapes falsos, tanto considerando que la secuenciación no es perfecta como que existe la posibilidad de que la misma secuencia de bases ocurra más de una vez en un genoma. También se pueden presentar fragmentos que no pertenecen al problema original, debido a contaminación[5] de ADN ajeno al problema, así como quimeras[6] generadas durante el proceso de clonación del ADN, ya que se usa una bacteria para la clonación[7] y de aquí se pueden obtener fragmentos con porciones de ADN de dicha bacteria. Otro problema son las repeticiones, que significa que secciones del ADN aparecen cientos de veces ya sea en forma de tándem (pequeñas secuencias repetidas muchas veces una trs otra) o dispersas en diferentes lugares de la secuencia.

Los algoritmos computacionales desarrollados para este proyecto de ensamble de fragmentos han podido resolver algunos de estos problemas, aunque también hay otros que no detectan.

---

[5] Contaminación: la principal causa es por procesar las muestras de ADN sin las condiciones adecuadas de aislamiento.

[6] Quimera: se origina al mezclar células provenientes de diferentes organismos.

[7] Clonación: proceso de generación de copias múltiples de un fragmento de ADN

Por otra parte, el tamaño de los genomas de los organismos es una variante importante, empezando desde pequeñas bacterias o procariontes, desde un millón de bases y hasta los eucariontes de más de 100,000 millones de bases, como se muestra en la figura 3.

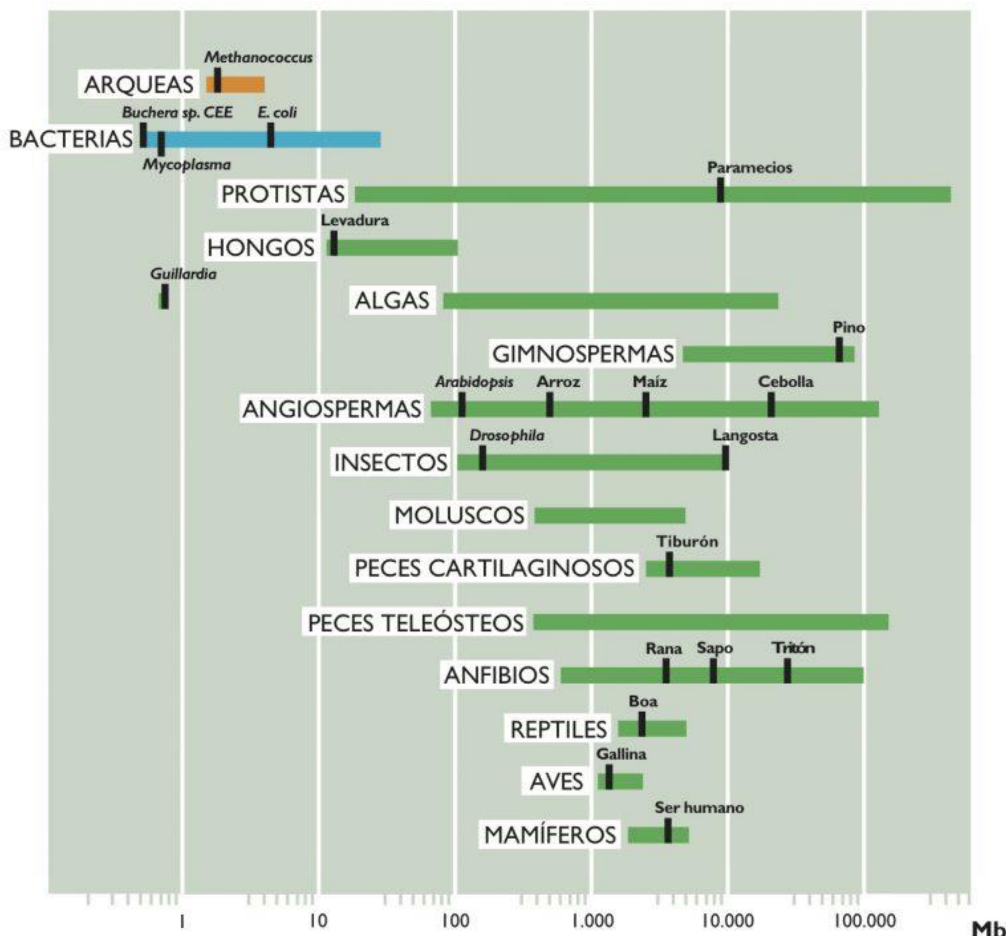


Figura 3. Tamaño en Mbyte del genoma de diferentes especies.
Tomado de (revista ”Métode” de la Universidad de Valencia, Amparo Latorre, Francisco J. Silva, 25/02/2013)

2.3 Antecedentes históricos

En 1943 Frederik Sanger obtuvo la secuencia de una proteína y a partir de esta observación, dedujo que el ADN debería de tener también un orden. En 1953 James Watson y Francis Crick (Watson & Crick, 1953) descubren la estructura de doble hélice del ADN. En 1975, Sanger (Sanger & Coulson, DNA

sequencing with chain-terminating inhibitors, 1977) desarrolló un mecanismo de secuenciación del ADN basado en la replicación.

En 1979 Staden (Staden, 1979) propuso un método de ensamble de los fragmentos de ADN para la obtención del genoma utilizando una computadora, partiendo de que los fragmentos de ADN se producen de múltiples copias del genoma original en una misma región y esto facilita que existan traslapes en los extremos de cada fragmento (figura 4).

```
Fragmento 1:  TTCACTTATTTAAAATCTGGAAGA
Fragmento 2:          TTTAAAATCTGGAAGAAACCTAGG
Ensamble:     TTCACTTATTTAAAATCTGGAAGAAACCTAGG
                      | <=  16  traslapes  => |
```

Figura 4. Ensamble a partir de traslape de fragmentos

En 1990 comienza formalmente el proyecto del genoma humano (Barbadilla, s.f.). En el año 2000 se presenta un borrador inicial; en 2003 se presenta el genoma esencialmente completo y en 2006 se culmina el proyecto con el último cromosoma humano. La importancia de este proyecto fue que diversas instituciones y gobiernos interactuaron y colaboraron para lograr el genoma completo y esto motivó un gran desarrollo en el área de la tecnología computacional aplicada a la genómica.

Weber y Myers (Myers Jr, 2016), posteriormente, propusieron la generación de fragmentos pares o apareados[8] en el proceso de *shotgun*. Tradicionalmente una secuencia inicia en el extremo identificado como 5’ y termina en 3’. Al generar secuencias pares se puede obtener información a

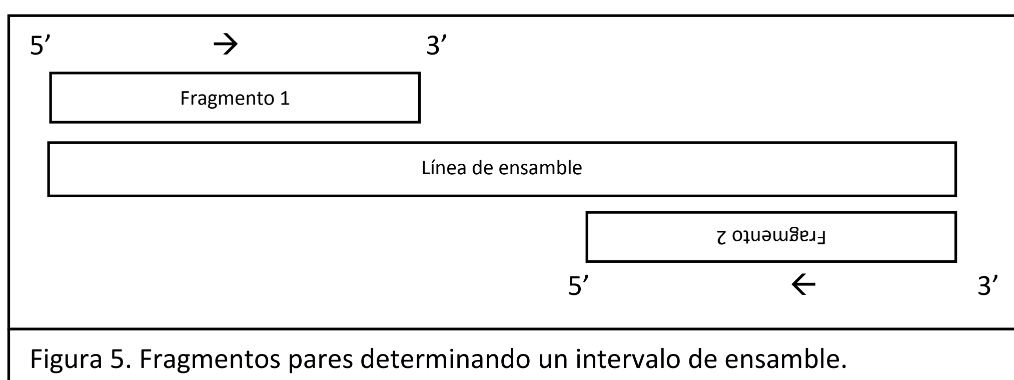


Figura 5. Fragmentos pares determinando un intervalo de ensamble.

[8] *Fragmentos apareados*: dos fragmentos que identifican un intervalo de secuenciación de ADN

partir de 3’ y con esto se obtiene dos fragmentos encontrados que identifican un intervalo de secuenciación (figura 5).

El fragmento que se lee desde 3’ está en inverso complementario por lo que para utilizarlo como delimitador del intervalo deberá convertirse la secuencia (figura 6).

Lectura directa:	A C C G T C G G A T
	| | | | | | | | | |
Lectura inversa Complementaria:	T G G C A G C C T A

Figura 6. Lectura directa y en inverso complementario

La tecnología de secuenciación del ADN ha avanzado mejorando costos y tiempos de procesos. En la actualidad hay bases de datos públicas con información sobre muchos genomas, incluyendo el humano (National Center for Biotechnology Information, s.f.). Steven Salzberg elaboró un estudio comparativo de porgramas ensambladores de fragmentos de ADN (Salzberg, y otros, 2012). Las pruebas que incluye el estudio de Salzberg se hicieron con el resultado de la secuenciación de equipos *Illumina* [9]. Este estudio se reportó en 2012, sin embargo, dado que el paradigma sigue vigente, el estudio sigue siendo relevante. La métrica principal en el estudio fue el N50 [10]. Salzberg trabajó cuatro organismos en la prueba, entre los cuáles se incluyó un *Staphylococcus Aureus*.

3. Hipótesis y Objetivos

Con el análisis de la estructura de los datos resultantes del proceso de secuenciación (fragmentos de ADN), se ha observado que se tiene información que permite identificar el posible traslape entre dichos fragmentos; estos fragmentos se pueden modelar mediante un grafo dirigido[11], el cuál es posible recorrer y a partir de la identificación de las rutas más largas conformar un *contig*[12]; en la medida en que se obtengan *contigs* de mayor longitud, será posible aproximar un genoma completo. A partir de esta conjetura, la

[9] *Illumina*: http://www.illumina.com/, Illumina, Inc., San Diego, CA, USA.
[10] *N50*: longitud más corta cercana a la media del conjunto de *contigs.*
[11] Grafo dirigido: Conjunto de vértices y aristas entre los vértices, orientadas en una dirección.
[12] *Contig*: grupo de fragmentos ensamblados que construyen un elemento contiguo.

descripción de la problemática computacional y la búsqueda de nuevos mecanismos para resolver el problema de ensamble de fragmentos “*FAP*” (por sus siglas en inglés: *Fragment Assembly Problem*) se ha determinado la hipótesis y el objetivo de esta investigación, mismos que se describen a continuación.

Hipótesis:

Es posible aplicar métodos alternativos a los habituales, de la teoría de grafos para el ensamble de fragmentos de ADN, que provea un algoritmo para generar resultados de calidad competitiva y consumo de recursos de cómputo moderado.

Objetivo:

Diseñar un nuevo algoritmo para el ensamble de fragmentos de ADN, basado en teoría de grafos, que genere resultados de buena calidad y que sea rápido y eficiente.

Particularmente se requiere:

1) revisar y documentar el estado del arte en investigación sobre FAP,
2) proponer nuevos modelos de algoritmos,
3) evaluar su complejidad en tiempo y espacio considerando problemas de organismos con varios millones de fragmentos,
4) elaborar los programas que implementan dichos algoritmos,
5) medir y comparar los resultados de ejecución contra otros programas de uso común, aplicando a casos reales de organismos.

4. Fundamentación y Estado del Arte

4.1 Ensamble de genomas de ADN

Existen dos tendencias para el ensamblado del ADN secuenciado: Ensambladores Comparativos y Ensambladores *de novo*. Los Ensambladores Comparativos parten de un genoma conocido que podría resultar similar al que se desea ensamblar; este método consiste en localizar los fragmentos acomodándolos hasta llegar a algún tipo de ensamble. Los Ensambladores *de novo* parten de un proceso de predicción que no hace comparaciones con

algún otro modelo. Las técnicas que se describen en los siguientes párrafos son *de novo*, ya que no tratan de acomodar los fragmentos en base a un genoma conocido.

El desarrollo de algoritmos de ensamble de genomas ha sido relevante a partir de los algoritmos *Greedy* [13] (Bang-Jensen & Yeo, 2004). El inconveniente de esta técnica es que se busca la mejor solución tomando sólo decisiones locales. Al aplicar el algoritmo al problema completo, los resultados no son del todo favorables. El algoritmo consiste en tomar un fragmento y colocarlo en el mejor lugar disponible, valorando la efectividad del resultado. Si el resultado es bueno va a buscar un nuevo fragmento; si el resultado no es bueno se desecha el fragmento y no se vuelve a considerar. Cuando el algoritmo presenta fallas ante ciertas muestras de datos, va a ser considerado como una heurística[14]. Una heurística puede ser un método altamente eficiente, sin embargo, no garantizará llegar a los resultados correctos. Una heurística es parte de la búsqueda de soluciones para problemas complejos y de alto volumen de datos, como es el caso de *FAP*, en el cual puede presentarse cierta pérdida de información sin demeritar el resultado final, aunque se sabe que no se podrá obtener un genoma completo al 100%.

Una de las técnicas más empleadas hoy en día es mediante la construcción de un grafo de *de Bruijn*[15]. Este tipo de grafo es una representación de traslapes entre símbolos. El grafo de *de Bruijn* parte de fragmentos cortos y forma un grafo de *k-mers* (segmentos cortos de bases de tamaño *k*) a partir de los cuales se buscarán los traslapes y la conformación de *contigs*. Esta técnica fue propuesta por Pevzner y Compeau (Compeau, Pevzner, & Tesler, 2011).

---

[13] *Greedy*: técnica algorítmica que trata de generar una solución óptima para una etapa específica del problema.

[14] *Heurística*:

Enfoque para la resolución de problemas, aprendizaje o descubrimiento que emplea un método práctico no garantizado para ser óptimo o perfecto, pero suficiente para los objetivos inmediatos. Alberto Cajal, https://www.lifeder.com/metodo-heuristico/.

Es una regla de sentido común basada en la experiencia en lugar de ser una afirmación matemáticamente probada (Levitin, 2012, pág. 442).

[15] *de Bruijn,* Nicolaas Govert : Holanda 9 July 1918 – 17 February 2012

Una gráfica de *de Bruijn* es un grafo dirigido que representa traslapes entre secuencias de símbolos. En el ensamble de fragmentos de ADN tiene como vértices todas las subsecuencias de longitud *k* (*k-mers*) encontradas en todas las lecturas y una arista por cada *(k+1)-mer* (sub secuencias de longitud *k+1*) presente en las lecturas, entre el prefijo *k-mer* y el sufijo *k-mer* de la subsecuencia. El algoritmo consiste en construir un grafo de *de Bruijn* a partir de los *k-mers* de cada fragmento, como se muestra en la figura 7. El grafo presenta discontinuidades y de cada elemento discontinuo se extraen las secuencias más largas; cada una de estas es un *contig*. Es una práctica común utilizar *k* del orden de 20.

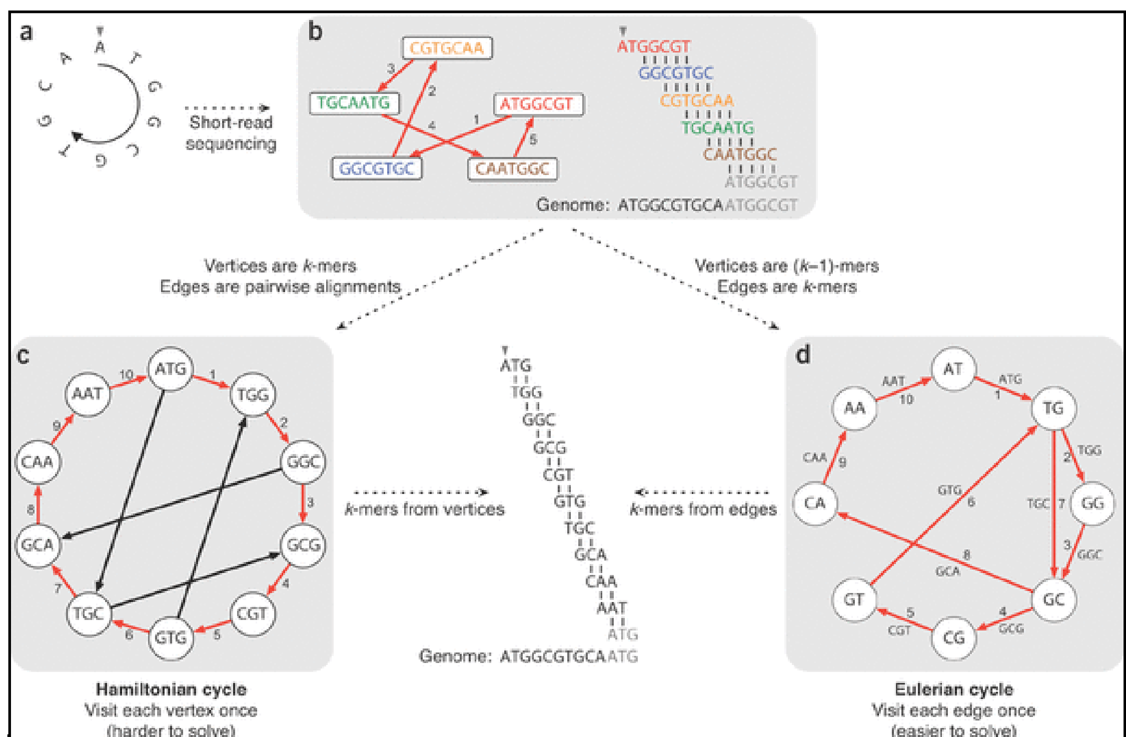


Figura 7. Construcción de un grafo de *de Bruijn*.

Tomado de (revista "Nature": http://www.nature.com/nbt/journal/v29/n11/full/nbt.2023.html)

Una técnica basada en algoritmos genéticos fue propuesta por Parsons (Parsons, Burks, & Forrest, 1993). Otros investigadores también aplicaron metaheurísticas similares a Parsons, pero siempre se obtuvieron pocos *contigs*, del orden de 1000 fragmentos y el resultado del procesamiento por computadora resultaba en tiempos muy largos.

En 2013, Mallén (Mallen-Fullerton & Fernandez-Anaya, 2013) presentó una nueva técnica a partir del "Problema de Agente Viajero" (*PAV* o *TSP* por sus

siglas en inglés, *Traveling Salesman Problem*). Se aplicaron métodos heurísticos formales y algoritmos del *PAV*, obteniendo valores óptimos para los *benchmark*[16] comúnmente usados y por primera vez se resolvió un problema de la vida real utilizando esta clase de optimización.

Aplicando teoría de grafos y fijando apropiadamente las funciones objetivo, (Mallén-Fullerton, Quiroz-Ibarra, Miranda, & Fernández-Anaya, 2015) en 2015 presentamos un nuevo método de ensamble de fragmentos de ADN, desde la perspectiva de un grafo dirigido y buscando la reducción de la complejidad de los algoritmos. En un segundo momento y dando continuidad a la misma problemática, desarrollamos otro nuevo algoritmo (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017) contemplando fragmentos apareados, resultantes de la secuenciación del equipo de *Illumina*. En este nuevo desarrollo se generaron dos estructuras de datos fundamentales: un árbol *Trie* (Ukkonen, 1995) para determinar los traslapes entre fragmentos y un grafo dirigido para generar los recorridos óptimos que producen los *contigs*. En la siguiente sección se detallarán los elementos de teoría de grafos y estructuras de datos necesarios que fundamentan su uso en este último desarrollo.

4.2 Teoría de grafos

Un Grafo *(G)* es un conjunto finito de vértices *(V)* y aristas *(E)* siendo representado por $G = \{V, E\}$ (Johnsonbaugh, 2005). Los vértices están conectados por las aristas por lo que a los extremos de cada arista siempre hay un par de vértices. Un grafo representa un conjunto de relaciones arbitrarias entre objetos. Los objetos son los vértices y las relaciones son las aristas. Los vértices se pueden identificar mediante letras o números y las aristas se pueden identificar por el par de vértices que los une, sin importar como se han identificado. Así por ejemplo $V = \{1,4,5,7,9\}$ y $E = \{(1,4), (1,5), (4,9), (9,7), (7,5)\}$. El dibujo del grafo no tiene una regla específica, sin embargo, se buscará cierta claridad en cuanto a la lectura tradicional de la izquierda hacia la derecha y de

---

[16] *Benchmark*: prueba de rendimiento comparativa. https://es.wikipedia.org/w/index.php?title=Benchmark_(inform%C3%A1tica)&oldid=102059576

arriba hacia abajo, tratando de evitar cruces o solapamientos innecesarios. En la figura 8 se muestra el grafo de las relaciones ejemplificadas en este párrafo.

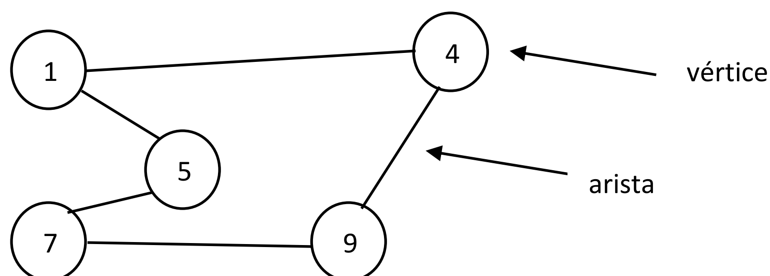


Figura 8. Grafo G con vértices y aristas.

Las aristas pueden ser líneas rectas, curvas, onduladas o de otra geometría, pero deben cumplir con la función de unir dos vértices. Una arista que sale de un vértice puede regresar a este mismo, finalmente une dos vértices; esta arista es un lazo. También puede darse el caso de dos vértices que son conectados en forma múltiple, es decir, más de una arista los une; esta arista es múltiple. En la figura 9 se muestran estas condiciones.

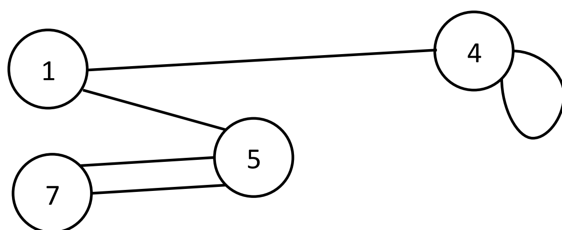


Figura 9. Grafo con arista (4,4) tipo lazo y arista (5,7) múltiple.

Las aristas pueden tener un peso, es decir, el costo de unir a dos vértices. Por ejemplo, en la figura 9 en la arista múltiple, la primera podría tener un costo de 15 y la segunda un costo de 8, lo cual puede significar la distancia, el esfuerzo o el valor de acercamiento, entre otros. En función de la necesidad de representación del grafo, se deberá definir este costo de las aristas. El grafo es una herramienta de análisis de relaciones; en forma gráfica se pueden detectar

una serie de situaciones de interés; pero también se tienen diferentes formas de representación numérica, adecuado para altos volúmenes de relaciones. Las formas numéricas pueden ser: matrices de adyacencias, matrices de incidencias y listas de adyacencias. Las más utilizada es la matriz de adyacencias, pero puede tener ciertas desventajas al existir dispersión de los datos, ya que dejaría muchos huecos sin valores ocupando espacio innecesariamente y desde el punto de vista del procesamiento de datos, esto no representa una buena alternativa de análisis. En caso de dispersión es más conveniente utilizar una lista de adyacencias.

Los grafos pueden ser dirigidos o no-dirigidos; este último es como se muestra en la figura 8, es decir, sin dirección en las aristas. En el caso del grafo dirigido o dígrafo, una arista *(u, v)* une al par de vértices *(u, v)* y se dice que están ordenados, lo que significa que el primer elemento *u* indica el punto de partida y el segundo *v* el destino de la arista. La figura 10 muestra un grafo dirigido o dígrafo.

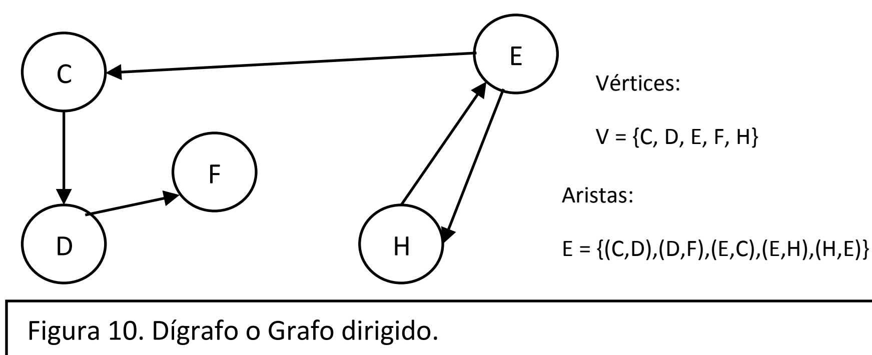


Figura 10. Dígrafo o Grafo dirigido.

El grado *gr(x)* de un vértice *x* (Tutte, 2001) está determinado por el número de aristas que se encuentran en dicho vértice. En un grafo no-dirigido, solo se cuentan las aristas. Por ejemplo en el grafo de la figura 9: $gr(1)=2$, $gr(5)=3$. En un dígrafo se debe distinguir el grado de entrada y de salida con base en las aristas que llegan y las que salen. Por ejemplo, en el grafo de la figura 10 se tiene $gr_e(C)=1$, $gr_s(C)=1$, $gr_e(E)=1$, $gr_s(E)=2$.

La adyacencia de un grafo se puede definir por los vértices o por las aristas. Los vértices son adyacentes cuando los une una arista. Por ejemplo, en la figura 9 los vértices 1 y 5 son adyacentes; los vértices 4 y 5 no son adyacentes. La adyacencia de aristas se da cuando hay en común un vértice.

Por ejemplo, en la figura 10 las aristas *(E, C)* y *(C, D)* son adyacentes; las aristas *(H, E)* y *(D, F)* no son adyacentes.

Una trayectoria en un grafo es una sucesión de vértices adyacentes y en la sucesión, las aristas son todas distintas; el vértice sí puede volver a aparecer. La representación de una trayectoria es $P= (v_0, v_1, v_2, \ldots, v_n)$. La longitud del tramo es n aristas y el número de vértices es $n+1$. En la figura 9 se tiene una trayectoria P= *(4,5,1,7),* la longitud del tramo es 3, ya que se requiere de tres aristas para el recorrido y son 4 vértices de referencia.

Un circuito es una trayectoria que inicia y termina en el mismo vértice, es decir $v_0 = v_n$ y la longitud debe ser mayor o igual a 2. El circuito es simple si todos los vértices involucrados aparecen una sola vez; por ejemplo, en la figura 11 se observa que *P = (1,5,4,1)* es un circuito simple y *P = (1,5,7,5,4,1)* no es un circuito simple, ya que hay repetición de vértices. Un lazo no es un circuito ya que no cumple con la condición de longitud mayor o igual a 2.

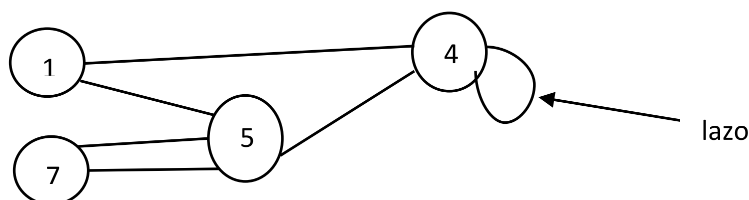


Figura 11. Grafo con circuitos.

Un grafo es conexo si existe un camino para comunicar cualquier par de vértices que forman el grafo. Si no se cumple lo anterior, entonces el grafo es disconexo, ya que está formado por varios pedazos y a cada uno de estos pedazos se le llamará componente. Un dígrafo es fuertemente conexo si existe un camino para comunicar cualquier par de vértices que forman el grafo, como se muestra en la figura 12.a. El dígrafo es solo conexo si cada vértice está conectado con alguno otro sin que necesariamente exista un camino de regreso, como se muestra en la figura 12.b.

Un grafo es completo si se tiene una arista para cualquier pareja de vértices; un grafo es incompleto si se permiten vértices sin arista (un caso

particular de un grafo disconexo) y finalmente un grafo es nulo si solo tiene vértices, sin arista alguna.

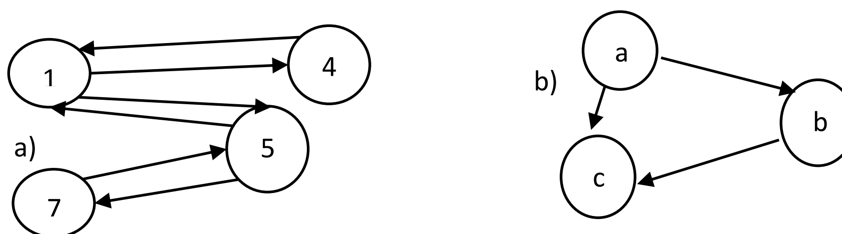


Figura 12. a) Dígrafo fuertemente conexo y b) Dígrafo conexo.

La lista de adyacencia para representar un grafo tiene un formato simple. Tomando como ejemplo el dígrafo de la figura 12.b y suponiendo ciertos valores de peso arbitrarios de las aristas *(a,b)* de 3, *(a,c)* de 1 y *(b,c)* de 8, la lista de adyacencia quedaría como:

a, b, 3

a, c, 1

b, c, 8

4.3 Algoritmos, Estructuras de Datos y Complejidad

Un algoritmo (del matemático persa Al-Juarismi 780-850) es la secuencia de pasos que se deben seguir para resolver un problema. Según el DRAE[17]: "Es el conjunto ordenado y finito de operaciones que permite hallar la solución de un problema".

Los algoritmos son aplicables a cualquier tipo de problemas, por ejemplo, culinarios, de reparaciones mecánicas, de registros contables, de prendido y apagado de una máquina, entre otros. Un algoritmo que se ha formulado correctamente y que se sigue estrictamente, da siempre el resultado esperado. Los algoritmos deben expresarse en los términos de la aplicación que se le da, así, por ejemplo, un algoritmo que dirige una receta de cocina se expresa con

[17] *DRAE*: Diccionario de la Real Academia de la lengua española

términos adecuados a una persona que tiene un conocimiento básico sobre cocina; un algoritmo para registrar un asiento contable tiene los términos adecuados para un profesional en la Contabilidad.

El conjunto de pasos del algoritmo encapsulado[18] genera una abstracción, de tal forma que no es necesario describir todo lo que hay que hacer en una receta de cocina para saber que se está cocinando un determinado platillo. El concepto de abstracción permite hablar del proceso en lo general, sin conocer los detalles o implicaciones.

Las abstracciones deben seguir ciertas reglas que son aplicables desde el exterior del proceso y hacia el interior del algoritmo mismo. Algunas reglas exteriores son: que sea identificable, que se sepa el rango de utilización, que sea posible aislarlo de su contexto para usarlo, que no sea necesario conocer los detalles; este nivel es similar al de una caja negra. Algunas reglas intermedias son: que se conozca lo que se requiere para funcionar, lo que generará como resultado, identificar si es necesario hacer uso de otros algoritmos o componentes. Finalmente, algunas reglas internas o propias del algoritmo son: que tenga solo un punto de entrada, aunque pueda tener varios puntos de salida, si tiene ciclos de repetición, que sean finitos, que se describa con un lenguaje claro y preciso, que sea ejecutable en el medio adecuado al tipo de problema que se resuelve; este nivel es equivalente a una caja blanca.

Aunque los algoritmos son aplicables a cualquier tipo de problema, en lo sucesivo, para efectos de este trabajo, los aplicaremos a problemas computacionales que requieren interacción con los recursos de una computadora y la interacción humana, para ello se deberá transcribir el algoritmo a un lenguaje de programación de computadoras, como *Java*, *C/C++*, *Perl*, *Python*, entre otros. A esta transcripción en el ámbito de la computación se le conoce como implementación. Los algoritmos computacionales se pueden representar mediante pseudocódigo, diagramas de flujo o algún lenguaje de alto nivel como *Python* o *SQL*.

[18] Encapsulado: Aislado del exterior.

Antes de elaborar el algoritmo es muy conveniente organizar el problema, fraccionarlo si es muy grande o complejo (y es suceptible de dividirse) y concentrarse en desarrollar los subproblemas que surjan de este modelo organizativo. Algunas técnicas son los diagramas HIPO (IBM, 1970), Redes de Petri (Petri, 1960), Diagramas de Flujo de Datos (Yourdon, 1986), diagramas de Warnier-Orr (Warnier, 1974), entre los más utilizados. Los subproblemas deben tener características de cohesión (que sean específicos a la tarea que deben resolver), de abstracción (visibles como caja negra) y de acoplamiento (reutilizables fácilmente).

Los algoritmos computacionales tienen instrucciones en secuencia lineal, es decir, una instrucción atrás de otra, tienen secuencias condicionales, es decir, se ejecuta una instrucción siempre y cuando se cumpla alguna condición y ciclos de repetición, es decir, se repite una acción o conjunto de acciones en función de una condición que puede estar generada dentro del mismo conjunto de instrucciones del ciclo de repetición o previamente establecida. A estos tipos de instrucciones se les conoce como Secuencia, Selección y Repetición. Estos tipos se pueden incluir entre ellos para generar soluciones más complejas. El éxito de la computadora precisamente radica en que es capaz de seguir una secuencia sin riesgo a equivocarse, seleccionar con la suficiente precisión y repetir a muy alta velocidad las instrucciones indicadas.

El hecho de contar con un algoritmo no quiere decir que se resuelve un problema o parte de este. Los algoritmos pueden ser más o menos eficientes, más o menos precisos y más o menos confiables. Por estas razones es muy importante documentar el rango de utilización de un algoritmo, es decir, sus límites, sus probables problemas, sus requerimientos de ejecución, el grado de precisión con el que trabaja y lo que se espera de éste. El no contar con suficiente documentación de un algoritmo conduce a que no sea utilizable o utilizarlo con desconfianza, generando esto un problema que puede ser mayor a lo que el mismo algoritmo pretende resolver.

En 1976 Niklaus Wirth (Wirth, 1976) propuso en su libro "*Algorithms + Data Structures = Programs*" que se deben acomodar los datos de tal forma

Figura 13. Lista Ligada simple

que sea óptima su utilización para la solución de un problema. La programación de computadoras se formalizó con este principio. Esto marcó la diferencia con las formas actuales de desarrollo de programas, por lo que el contar con una estructura de datos adecuada al problema y al algoritmo, genera una solución formal, alejada de técnicas artesanales o de gurús de la computación.

Una estructura de datos es una forma de acomodar los datos de un problema a fin de que se apeguen al tipo de solución óptima que se puede diseñar. Hay dos tipos de estructuras de datos: las lineales y las no-lineales. Las lineales parten del concepto de Lista Ligada (figura 13), en la cual un elemento es consecutivo a otro elemento y así sucesivamente. Con este concepto de lista, se generan listas ligadas simples, listas doblemente ligadas, listas circulares, pilas de datos, colas de datos y colas de datos con prioridad.

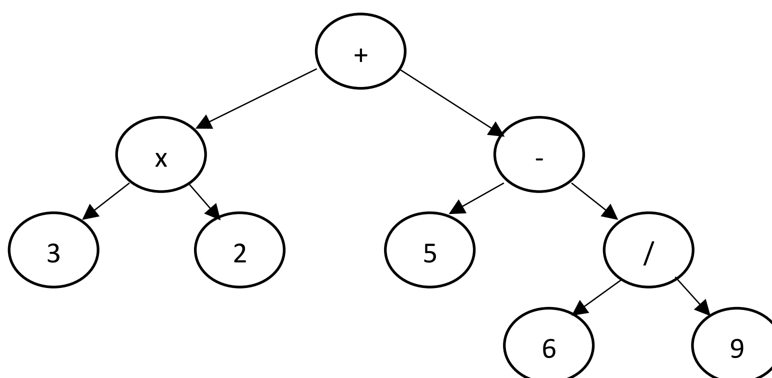


Figura 14. Árbol binario de la expresión: 3 x 2 + 5 – 6 / 9

Las estructuras de datos no-lineales parten del concepto de un árbol y de aquí se derivan los árboles de búsqueda, los árboles binarios, los árboles de expresiones (figura 14), los árboles n-arios, árboles de prefijos (*Trie*) (figura 15) y los grafos, tanto no-dirigidos como dirigidos. En los árboles el principio fundamental es que hay un elemento identificado como raíz, del cual se derivan otros elementos hacia las ramas del árbol y estos elementos pueden ser otras raíces u hojas del árbol, mismas que ya no pueden derivar hacia otros elementos. Los grafos se diferencian de los árboles en que no tienen un nodo raíz del cual se deriva todo el árbol y pueden tener ciclos.

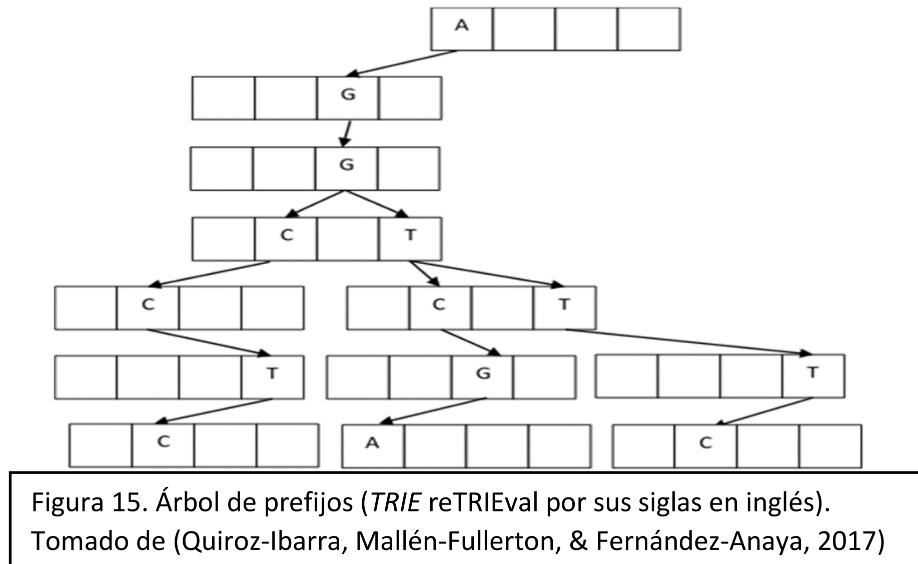


Figura 15. Árbol de prefijos (*TRIE* reTRIEval por sus siglas en inglés). Tomado de (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017)

Un algoritmo se diseña para efectuar cierta operación, pero ¿hace lo que debe hacer, con los recursos adecuados y en el tiempo adecuado? En la medida en que la tecnología nos provee de computadoras más poderosas y económicas se piensa que no importan algunos factores de consumo de poder computacional, mientras se resuelva el problema en cuestión; si el recurso de cómputo no es suficiente, seguramente hay alguna otra tecnología que con alguna diferencia económica pueda resolver el problema; para la condición actual del mundo con sus recursos limitados, esto ya no es una forma adecuada de proceder en la solución de muchos problemas.

Un algoritmo debe tener ciertas cualidades (Levitin, 2012, pág. 14):

- Corrección: que haga lo que deba hacer correctamente
- Eficiencia en términos de tiempo: que lo haga en el menor tiempo posible
- Eficiencia en términos de espacio: que lo haga con el menor consumo de memoria posible
- Simplicidad: que su diseño sea muy fácilmente entendible por una persona
- Generalidad: que pueda ser aplicable en

otros problemas

En una primera implementación no siempre se logran resolver todas estas cualidades por lo que un proceso de rediseño siempre es adecuado. Los aspectos más relevantes en el diseño y rediseño de un algoritmo son los referidos a la eficiencia, en términos de tiempo y espacio y siempre a partir de que el algoritmo es correcto. El estudio de la eficiencia de tiempo y espacio es un análisis de la complejidad del algoritmo.

La definición de complejidad, en términos de Edgar Morín (Morin, 1994, pág. 32) es: "La complejidad es un tejido de constituyentes heterogéneos inseparablemente asociados: presenta la paradoja de lo uno y lo múltiple. …la complejidad es, efectivamente, el tejido de eventos, acciones, interacciones, retroacciones, determinaciones, azares, que constituyen nuestro mundo fenoménico…".

La Complejidad Computacional, en términos de Hartmanis y Hopcroft (Hartmanis & Hopcroft, 1971), es una teoría que busca clasificar los problemas de acuerdo con su dificultad inherente y la relación entre las clases de complejidad. Los dos aspectos de estudio de la complejidad de un algoritmo son el tiempo y el espacio, por lo que la teoría de la complejidad computacional busca evaluar la viabilidad de implementación de un algoritmo por medio de la cuantificación del tiempo y espacio requerido de un algoritmo para funcionar. Se busca llegar a elementos que ayuden a determinar si las funciones algorítmicas son computables o no-computables. Las clases de complejidad se clasifican como sigue (Dean, 2016):

Clase L: Es el conjunto de los problemas de decisión que pueden ser resueltos en espacio log(n) (sin contar el tamaño de la entrada), donde n es el tamaño de la entrada, por una máquina de Turing *determinista* tal que la solución, si existe, es única.

Clase NL: La clase de complejidad NL (espacio logarítmico no determinista) es el conjunto de los problemas de decisión que pueden ser resueltos en espacio log(n) (sin contar el tamaño de la entrada), donde n es el

tamaño de la entrada, por una máquina de Turing *no determinista* tal que la solución, si existe, es única.

Clase P: Son todos aquellos problemas de decisión que pueden ser resueltos en una máquina determinista secuencial en un período de tiempo polinómico en proporción a los datos de entrada.

Clase NP: Es el conjunto de problemas que pueden ser resueltos en tiempo polinómico por una máquina de Turing no determinista. La importancia de esta clase de problemas de decisión es que contiene muchos problemas de búsqueda y de optimización para los que se desea saber si existe una cierta solución o si existe una mejor solución que las conocidas.

Clase NP-Completo: Informalmente, los problemas de NP-completos son los problemas más difíciles de NP, en el sentido de que son los más probables de no encontrarse en P. Los problemas de NP-completos son esos problemas NP-duros que están contenidos en NP, aunque NP-duro pudiera no ser parte de NP.

Para comprender la complejidad del algoritmo considérese la operación de multiplicación siguiente:

```
  123
X 456
-----------------
     738          <- 123 x 6 implica tres multiplicaciones
    615           <- 123 x 5 implica tres multiplicaciones
   492            <- 123 x 4 implica tres multiplicaciones
-------------------<- total de 9 multiplicaciones
 56088            <- implica una suma y 9 multiplicaciones.
```

Ahora efectuando la siguiente multiplicación:

```
            123
          X 4567
------------------
           861            <- 123 x 7 implica tres multiplicaciones
          738             <- 123 x 6 implica tres multiplicaciones
         615              <- 123 x 5 implica tres multiplicaciones
        492               <- 123 x 4 implica tres multiplicaciones
-----------------         <- total de 12 multiplicaciones
       561641             <- implica una suma y 12 multiplicaciones
```

Se puede deducir ahora que el número de multiplicaciones cambia en función de la cantidad de dígitos de cada cifra, es decir, para el primer caso 3x3=9 y para el segundo caso 3x4=12; la suma es una sola operación en todos los casos. La cantidad de operaciones en una multiplicación es entonces *(nxm)+1*, donde *n* es la cantidad de dígitos del multiplicando y *m* es la cantidad de dígitos del multiplicador. Se ha elaborado el análisis del algoritmo de la multiplicación, considerando que las entradas del algoritmo son el multiplicando y el multiplicador y la salida es el resultado de la multiplicación.

Las operaciones de la multiplicación se llevan a cabo sin importar que tipo de computadora es o que sistema operativo tiene o si es de arquitectura de 32 o 64 bits o si se hace con lenguaje *C/C++* o *Java*. Dada la gran oferta de computadoras con diferentes capacidades es difícil pronosticar el tiempo exacto que se lleva una operación, de aquí que lo que se hace es obtener una medida basada en la cantidad de operaciones que deben hacerse. Este es un dato relativo y en realidad representa el grado de complejidad del algoritmo. Es importante aclarar que esta valoración no tiene relación con el grado de esfuerzo realizado para el desarrollo del algoritmo, ya que podrían existir algoritmos con un alto grado de dificultad de desarrollo, pero un buen nivel de eficiencia o viceversa.

La medición de la cantidad de veces que se ejecuta un algoritmo puede resultar una labor tediosa por lo que se recomienda identificar las operaciones más críticas de un algoritmo, cómo contribuyen éstas a la operación total del algoritmo y calcular el número de veces que esta operación se llevaría a cabo. La identificación de estas operaciones críticas no debe representar un problema mayor, ya que suele ser la operación con más consumo de recursos. Por ejemplo, en un algoritmo de ordenamiento se sabe que la operación más crítica es la comparación de llaves, por lo que es esta operación la que hay que valorar. Otro ejemplo es con un algoritmo que hace sumas, restas, multiplicaciones y divisiones; la más crítica de éstas es la división, ya que requiere más pasos, luego está la multiplicación y finalmente la resta y suma como las más simples operaciones. Como conclusión, una vez identificada la

operación más crítica solo resta valorar el número de veces que se va a ejecutar, en función al tamaño de la muestra de datos (*n*).

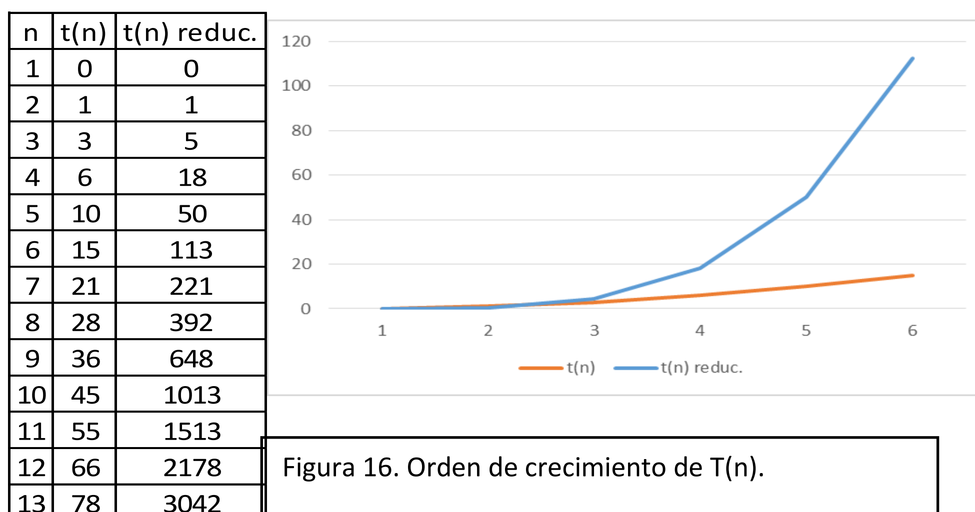
Considérese el siguiente caso (Levitin, 2012): *cop* es el tiempo de ejecución de la operación crítica de un algoritmo; *C(n)* es el número de veces que la operación crítica debe ejecutarse para un algoritmo. Ahora se puede estimar el tiempo total de ejecución como: $T(n) \approx cop * C(n)$. Es importante observar que esta fórmula es una aproximación, ya que solo considera las operaciones más críticas y desecha las no importantes. Esta fórmula da un orden de magnitud basado en el tamaño de la muestra de datos *n* y también da el orden de crecimiento dependiendo de *n*. Por ejemplo, $C(n) = ½\, n\, (n-1)$. Esta función tiene un orden de crecimiento de tal forma que, si *n* se duplica, *C(n)* se cuadruplica aproximadamente tal como se observa en la figura 16. Se puede aproximar entonces que, para valores pequeños de *n*, la fórmula se reduce a:

$$C(n) = ½\, n\, (n-1) = ½\, n^2 - ½\, n \approx ½\, n^2.$$

Y para recalcular *T(n)* con el doble de la muestra de *n*, es decir *2n*, se tiene:

$$T(2n) / T(n) \approx ½(2n)^2 / ½\, n^2 = 4.$$

Este es el factor de crecimiento para valores pequeños de *n*, como lo muestra la figura 16, con *n* menor a 3 y *T(n)* reducida.



n	t(n)	t(n) reduc.
1	0	0
2	1	1
3	3	5
4	6	18
5	10	50
6	15	113
7	21	221
8	28	392
9	36	648
10	45	1013
11	55	1513
12	66	2178
13	78	3042

Figura 16. Orden de crecimiento de T(n).

La parte importante del crecimiento de una función *C(n)* es lo que debe ser representativo del algoritmo y se busca una tabulación que permita identificar los puntos críticos o asíntotas donde el algoritmo va a dejar de ser funcional. En la tabla de la figura 17 se muestran los órdenes de crecimiento para algunas funciones de algoritmos.

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	n!
10	3.321928095	10	33.21928095	100	1000	1024	3628800
1.00E+03	9.965784285	1.00E+03	9965.784285	1E+06	1E+09	1.0715E+301	
1.00E+04	13.28771238	1.00E+04	132877.1238	1E+08	1E+12		
1.00E+05	16.60964047	1.00E+05	1660964.047	1E+10	1E+15		
1.00E+06	19.93156857	1.00E+06	19931568.57	1E+12	1E+18		
1.00E+07	23.25349666	1.00E+07	232534966.6	1E+14	1E+21		

Figura 17. Tabulación de diferentes funciones de crecimiento de T(n).

Es notorio que las funciones exponenciales dejan de ser utilizables más pronto que las logarítmicas, de aquí que un algoritmo, debe buscarse que sea logarítmico, aunque esto no es simple.

Otros aspectos que considerar en la evaluación del tiempo del algoritmo son: el mejor de los casos, el peor de los casos y el caso promedio. Para evaluar estas tres condiciones se describe como ejemplo el algoritmo 1 que busca un valor *k* en un arreglo de elementos *arr*, de tamaño *n* y solamente devuelve si

Algoritmo 1 Búsqueda

// k: valor a localizar
// arr: arreglo de elementos desordenado
// n: tamaño del arreglo

1. *Encontrado ← falso*
2. *i ← 0*
3. *Mientras i < n*
   3.1 Si arr[i] == k
   Encontrado = verdadero
   Salir del ciclo.
   3.2 i ← i +1
4. *Fin-mientras.*

está o no.

En el mejor de los casos *C(n) = 1*, es decir, encuentra el valor *k* en el primer elemento del arreglo.

En el caso promedio, en realidad la búsqueda va a depender de la probabilidad *p* de que el elemento *k* esté en alguna posición intermedia, por lo que si *p* está en el rango *0 ≤ p ≤ 1* y la probabilidad de que *k* esté en cualquier posición del arreglo es la misma, entonces:

$$C(n) = \left(1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}\right) + n(1-p)$$

$$C(n) = \frac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1-p)$$

pero $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

entonces $C(n) = \frac{p}{n}(n(n+1)/2) + n(1-p)$

y finalmente $C(n) = p(n+1)/2 + n(1-p)$

Esta fórmula provee mucha información. Por ejemplo, si *p = 1* la búsqueda es exitosa y la cantidad promedio de comparaciones es *(n + 1) / 2*. Si *p = 0* la búsqueda ha fracasado y la cantidad promedio de comparaciones es *n*, ya que el algoritmo ha inspeccionado todos los datos, es decir, *n*.

El análisis de la eficiencia de un algoritmo se concentra en el orden de crecimiento del conteo de las instrucciones básicas, siendo esto el principal indicador del comportamiento del algoritmo. Para comparar y valorar este orden de crecimiento se utilizan tres notaciones (Cormen, Leiserson, Rivest, & Stein, 2009): *big O* denotada por *O*, *big Omega* denotada por *Ω* y *big theta* denotada por *Θ*. A continuación, se definen estas notaciones.

Retornando al algoritmo 1 en el que se tiene el peor de los casos en *n*, donde *n* es el tamaño de la muestra de datos, y esta misma *n* es el factor que provoca el crecimiento de las operaciones del algoritmo, el tiempo de ejecución del algoritmo es el punto de interés para lo cual se utiliza la notación *Θ(n)*. Una vez que n sea suficientemente grande, el tiempo de ejecución se encontrará en el intervalo $k_1 n$, al menos y a lo más $k_2 n$, como se muestra en la gráfica de la figura 18. Para valores a la izquierda de la línea punteada de la gráfica de la figura 18, no es de interés el comportamiento del algoritmo. Una

vez que se supera la línea punteada el tiempo de ejecución está entre $k_1 n$ y $k_2 n$ *y esto es Θ(n)*.

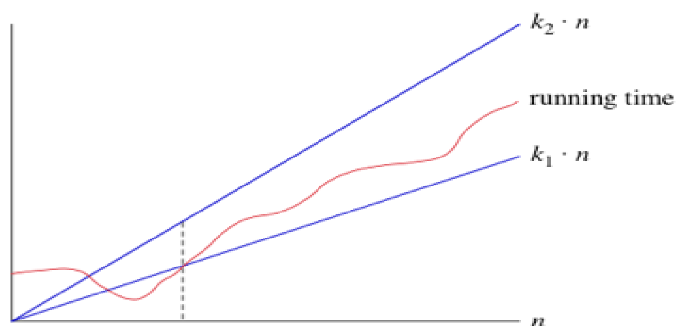


Figura 18. Intervalos de *Θ(n)*.

Tomado de (Cormen & Balkom, s.f.)

Se pueden usar otras funciones de crecimiento de *n* como n^2, *n log n*, y otras (ver figura 17), dependiendo del algoritmo empleado. En la práctica el uso de la notación *Θ(n)* es que no importan las unidades de tiempo. Por ejemplo, si el tiempo de ejecución es de $6n^2 + 100n + 300$ microsegundos o milisegundos, al pasar a notación *Θ*, no se especifica esta unidad, también se descartan los términos $100n + 300$ y solo se dice que es $\Theta(n^2)$.

La notación *big Θ* sirve para acotar hacia arriba y hacia abajo el crecimiento de una función que representa un algoritmo. Es común requerir una notación asintótica que delimite el tiempo de ejecución solo hacia el límite superior. Esta notación es la *big O*.

Si el tiempo de ejecución es *O(f(n))*, el límite superior estará determinado por *k f(n),* para *n* suficientemente grande, es decir, *big O* sirve para cotas superiores asintóticas. Si el tiempo de ejecución es *Θ(f(n))* en una situación particular, entonces también lo es *O(f(n))*.

Si ahora el tiempo de ejecución de un algoritmo es *Ω(f(n))*, entonces al menos existe una cierta cantidad de tiempo; esto representa la notación *big Ω.*

Si un tiempo de ejecución es *Ω(f(n))*, para *n* suficientemente grande, entonces el tiempo de ejecución es por lo menos *k f(n)*, para alguna constante *k*. Se usa la notación *big Ω* para límites asintóticos inferiores.

La mayoría de los algoritmos son del tipo polinomial (Cormen, Leiserson, Rivest, & Stein, 2009, pág. 1048), con muestras de datos del tamaño *n* y en el peor de los casos $O(n^k)$, para alguna constante *k*. Sin embargo, no todos los problemas se pueden resolver en tiempo polinomial. Un algoritmo que se resuelve en tiempo polinomial es tratable (*P*) y un problema que requiere tiempo superpolinomial será considerado como intratable o duro (*NP*). No se ha descubierto a la fecha un algoritmo polinomial *P* para un problema *NP-completo*, por lo que surge la pregunta ¿*P = NP*? Hay muchos problemas del tipo *NP-completos* que tentativamente pueden ser parecidos a un problema *P*, aunque con un análisis más detallado se encontrará que dichos problemas serán efectivamente *NP-completos*.

Un caso de *NP-completo* es la búsqueda en un grafo *G=(V, E)* en tiempo $O(V \cdot E)$ de la ruta más corta entre dos vértices; localizar la ruta más larga también es un problema con alto grado de dificultad. Como conclusión, encontrar una ruta simple con cierto número de aristas es un problema *NP-completo*.

Un segundo caso que es de interés es encontrar un recorrido de Euler en un dígrafo (figura 7), es decir, un ciclo que recorre el grafo pasando por cada arista una sola vez; este recorrido tiene un tiempo *O(E)*. Un ciclo hamiltoniano de un grafo *G=(V, E)* es un recorrido que contiene una sola vez cada vértice. Determinar si un grafo tiene recorrido hamiltoniano (figura 7) es del tipo *NP-completo*.

Resumiendo, se pueden clasificar los problemas en tres grandes clases: *P*, *NP* y *NP-completos* (Cormen, Leiserson, Rivest, & Stein, 2009, pág. 1049). Los problemas del tipo *P* son los que se resuelven en tiempo polinomial, como se había dicho anteriormente. Esto significa que su solución está en el tiempo $O(n^k)$, para alguna constante *k* y donde *n* es el tamaño del problema. Los problemas del tipo *NP* podrían ser verificables en un tiempo polinomial, dependiendo del tamaño del problema. Cualquier problema *P* puede ser también *NP* y no siempre será verificable, es decir $P \subseteq NP$. Sin embargo, no hay certeza de que todo *P* sea propiamente un subconjunto de *NP*. Finalmente, ¿un problema *NP-completo* es un problema *NP-duro*? ¿*NP-duro* es otra clase de problemas? Un problema *NP-completo* es intratable, es decir, no hay forma de obtener una solución polinomial de éste. Las relaciones entre *P*, *NP*, *NP-*

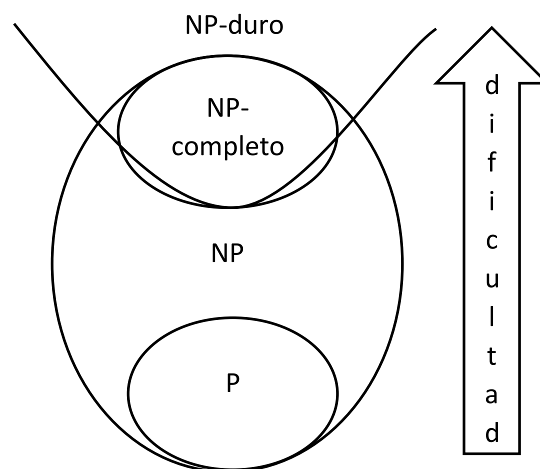


Figura 19. Relación posible entre P, NP, NP-completo y NP-duro

completo y *NP-duro* podrían verse como se muestra en la figura 19.

Los problemas que son extremadamente ineficientes por cada dato de entrada y que no tienen forma de llegar a una polinomial serán considerados como *NP-duros*. Algunos científicos consideran que este tipo de problemas no existen en la realidad, sin embargo, esta conjetura nunca ha sido probada, pero esto podría significar que *NP-duro* no es parte *NP-completo*.

5. Metodología

El proceso del ensamble de fragmentos de ADN requirió de una serie de pasos, los cuáles fueron desarrollados aplicando las estructuras de datos y algoritmos que se describirán en los siguientes subincisos. La intención de esta secuencia de pasos ha sido garantizar un resultado óptimo, en un tiempo de procesamiento razonable. En el diagrama de la figura 20 se muestran los pasos, mismos que serán descritos posteriormente.

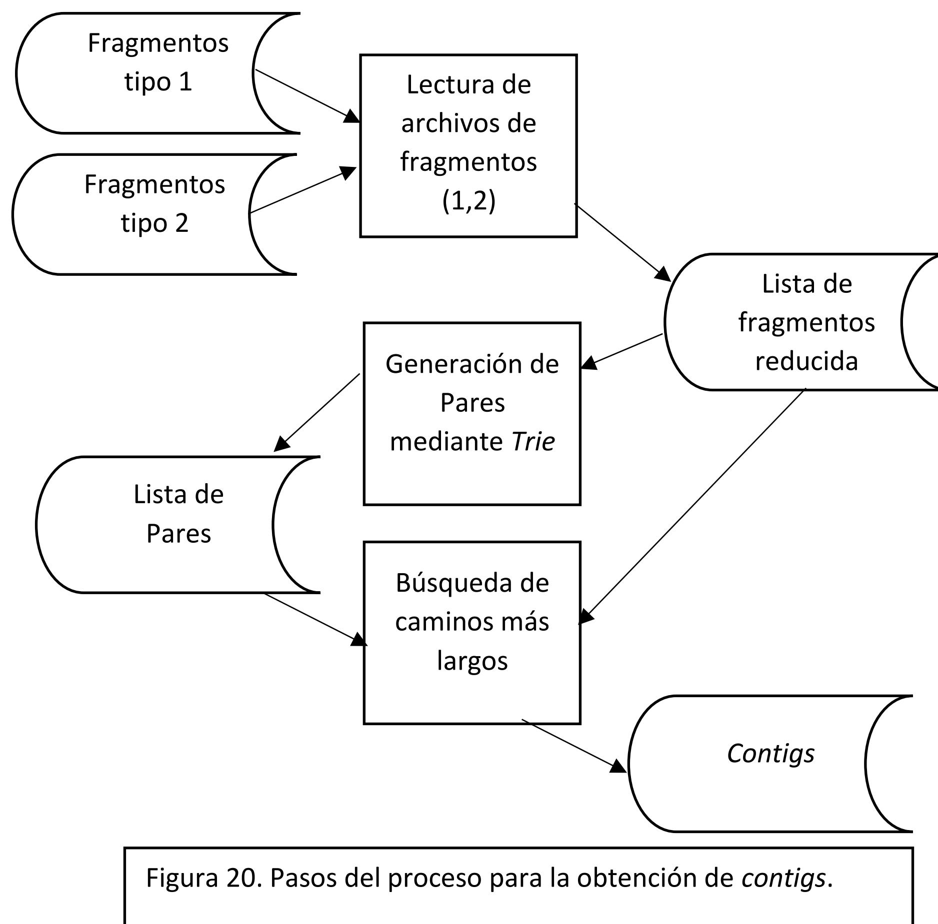


Figura 20. Pasos del proceso para la obtención de *contigs*.

5.1 Archivos: Formato y procesamiento

Los archivos de fragmentos (1,2), la lista de fragmentos reducida, la lista de pares y el archivo de *contigs* se describen a continuación, junto con los detalles del proceso de lectura y grabación.

Archivos de fragmentos.

Estos archivos contienen la información generada por la máquina de *Illumina*. Se recuperan del proceso de la máquina dos archivos. El primero contiene los registros del tipo 1 del intervalo y el segundo contiene los registros del tipo 2, también del intervalo, con un código de identificación común para resolver el intervalo. Una muestra de datos del tipo 1 y 2 se exhiben en la figura 21.

```
@SRR022868.923/1
ATCTACTTACTGGAAGTTTAATTTGAGTAAATTGTTATCCAGTCATTCGTTAGAACTCCTTA
TAGTACTTATACCNNNNNNNNNNNNNNNNNNNNNNNNNN
+
II*II?IIIIIIIIII8;III2I@I;I4E+2'I?'F>>$.II.&0@+)I:,6(,2
#+*#+#*.'('''+$54%$*!!!!!!!!!!!!!!!!!!!!!!!!!!

@SRR022868.1240/1
TCTGCAGTTTCAAACGGGCTTCCAACTGACTCATCTACATATTTATTAGCTTGTTAACTTGA
CATAACACCATGTAACTCTTCCATAGACTCTAAATGCTC
+
D4II+HII(I%III&'II+I+%)+43I')=)86)%&=7+I7-%9&2(.9%&%>&**%"%
&($#8%%*&%&&,3'-#-#.%"4%"%&#$&&###'$#$##&#

@SRR022868.923/2
NNNNNNNNNTNNNNNNNNNNNNNNNNNNNNANNNNNNNNNNNNNNNNNNNNNTNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
+
!!!!!!!!!'!!!!!!!!!!!!!!!!!!!!
@!!!!!!!!!!!!!!!!!!!!!-!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!

@SRR022868.1240/2
AATCGCGAGACAAGACCGAACATTTGAGATTAATGTTTCTGGAGTAAAAAATCATTAAGATA
AGTACACACCTGNGCCGNCATCTACTGACNCNNNNCNTN
+
I,I;4IG%II)6$$*)+)):%*H)8*+$%E?I4&&3+'2-3,2+0#$.%&'#$##"&%#&##
$$%$%#'%$",%!'%""!"'##%&'+$%*!-!!!!#!#!
```

Figura 21. Muestra de registros de fragmentos generados por *Illumina*.

Cada fragmento en realidad consta de cuatro registros. El primero es un código de identificación único, termina con “/1” ya que es el inicio de un intervalo, pero si es el final del intervalo termina con “/2”. El segundo registro son las bases A, C, G, T, N, aunque “N” no es una base y se refiere a un valor indeterminado, por lo que se decidió eliminar cualquier fragmento que presentara una o más “N”. Posteriormente un registro con un carácter “+” y el cuarto registro es la información sobre la calidad de la información. En este trabajo no se está tomando en cuenta dicha información. Para el ejemplo de

datos que se muestra en la figura 21, el registro de bases es de longitud fija de 101 caracteres.

Lista de fragmentos reducida.

Este archivo es una forma compacta de los fragmentos. Se integra en un solo registro de longitud fija por cada fragmento leído, generando los siguientes campos: un número consecutivo iniciando por cero, la posición del registro en el archivo de fragmentos original (*offset*[19]), esto con la intención de localizarlo con un acceso directo y no con una búsqueda secuencial; posteriormente está el código del registro, que puede ser “/1”, “/2” o “/1_inv” o “/2_inv”; estos dos últimos son los inversos complementarios de sus correspondientes tipos 1 y 2 y finalmente las bases. Se filtran los fragmentos que tienen “N” o algún otro carácter diferente a A, C, G, T. También se eliminan de la lectura los registros que muestran una misma base consecutiva cierto número de veces. Conforme se leen los registros tipo 1 y 2 se hace una validación de fragmento duplicado por las 101 bases. Si el fragmento leído es duplicado, no se continúa con la construcción de un inverso complementario. Si el inverso complementario se duplica, solo se omite su grabación. En la figura 22 se muestra un ejemplo de registros de fragmentos reducidos.

```
0000000 0000004936 @SRR022868.1240/1
TCTGCAGTTTCAAACGGGCTTCCAACTGACTCATCTACATATTTATTAGCTTGTTAACTTGA
CATAACACCATGTAACTCTTCCATAGACTCTAAATGCTC
0000001 0000004936 @SRR022868.1240/1_inv
GAGCATTTAGAGTCTATGGAAGAGTTACATGGTGTTATGTCAAGTTAACAAGCTAATAAATA
TGTAGATGAGTCAGTTGGAAGCCCGTTTGAAACTGCAGA
0000002 0000005164 @SRR022868.1263/1
AGCACGTTGTCACTCATTTCGAGATCCTTTTCAACTTGCTTTTTATTCTTTTCGAAATCAGC
TGGTTGAGTAGTTATGAGTTCATTATTTTTATTAGAATA
0000003 0000005164 @SRR022868.1263/1_inv
TATTCTAATAAAAATAATGAACTCATAACTACTCAACCAGCTGATTTCGAAAAGAATAAAAA
GCAAGTTGAAAAGGATCTCGAAATGAGTGACAACGTGCT
0000004 0000005392 @SRR022868.1281/1
ACGTCTTGTTAATCAGTTTGCGTCTTACGATTGGCCACTAGATCTAGCTTTAATTTCTATTC
CAATATTTTCTGTCAATGTGAGCTCAGATGAATTGACAC
```

Figura 22. Muestra de registros de fragmentos reducidos.

[19] *Offset*: desplazamiento de un registro dentro de un archivo. Al conocer esta posición se puede llegar al registro con un acceso directo, calculando la distancia total, en bytes, desde el origen (Wirth, 1976, pág. 23).

Lista de Pares.

Este archivo es la lista de aristas del dígrafo, obtenida a partir del *Trie*. Se construyó una estructura de datos del tipo *Trie*, en el que se colgaron todos los fragmentos que pasaron la validación y sus correspondientes inversos complementarios. Posteriormente, se reinició la lectura de fragmentos y a cada fragmento leído se le resta una posición, se busca una coincidencia contra los fragmentos originales colgados en el *Trie*; si no existe, se resta otra posición y así sucesivamente hasta llegar al límite del valor de traslape. Si no hubo coincidencias, se pasa al siguiente. Al encontrar la primera coincidencia, se graba en el archivo de pares indicando el vértice origen, el vértice destino y el peso del traslape que es la arista misma. Los vértices origen y destino se indican con el valor consecutivo del archivo de fragmentos reducido. En la figura 23 se muestra un ejemplo de registros pares.

```
000000002 001957482 054
000000002 001550273 050
000000003 000612071 098
000000003 000451009 093
000000003 000697341 084
000000003 000323938 084
000000003 002151215 079
000000003 002157862 078
000000003 001253339 059
000000003 000721626 052
000000003 001861264 052
000000003 001404185 051
000000007 000827013 100
000000007 001010952 093
000000007 000333666 089
000000007 001786331 084
000000007 001088225 069
```

Figura 23. Muestra de registros de pares de vértices de la Lista de Aristas.

Archivo de *Contig*s.

Este archivo es el resultado final de todo el proceso. Con los registros pares se construye una estructura de datos del tipo lista de adyacencia múltiple, con una sección de cabeceras y de cada registro de cabecera, que vértices son adyacentes a éste. Esta estructura de datos permite la localización de los recorridos más largos, iniciando por vértices que tiene grado de entrada cero y hasta llegar a los vértices que tienen grado de salida cero. Dado que pueden ser varios con grado de salida cero, se toma aquel que tenga el mayor peso acumulado de los traslapes; esta es la ruta más larga. Esta ruta del dígrafo contiene la información necesaria para acomodar las bases a partir del archivo reducido de fragmentos, construyendo así el *contig*. El archivo se graba en formato FASTA, como se muestra en la figura 24.

```
>contig 6   long. 447
ATAGATCAGCGTAAACAGTAGGGTGTAACGGGAAGAACGGTGCTCCGAAACATGTTGAGAAT
GATGGTTCAGGTTCTGTCACACCACGCTCTGTACCAGCTAATTTAGAAGTGAAACCACTCAA
GAAATGATACATTGCTTGGTCTTTATTTAACTTTGAAATCGGTGGAATAACACCAAATGCAT
CCGCAGTTAAGAAAATAATTGTATTTGGATGTGCTGCTTTAGATGGTACTACAATATTGTCA
ATGTGATTAATTGGATAAGCGGCACGCGTGTTTTCTGTATAACGATTGTCTTCAAAGTCCAC
TGAACCATCTTCTGCAACTACAGTGTTCTCTAAAATTGCACCATATTTGATTGCGTCAAAAA
TCTGTGGTTCTTTTTCTTTGGAAAGATTAATTGCTTTTGCATAGCAGCCACCTTCGATATTA
AAGACCCCGTTTT
>contig 7   long. 602
TAATCGAAAATCGACACCAATCGTTTCATCATCATACGCTAATTCTAATGAAACTTTTGTGT
TCCCCACTGCAACTTCTTTTACTTTATTGGGAATATTTAATAATTGCTGCTGCACTGCTTTT
GGGTTATCGGTACTTATTATGAAATCTAAATCTTTGCTCATTTCTTTAAAACGACGGAAGCT
TCCTGCAGATGAATATTGATCGATATAATTTAATGTATCTATATAATCAATGATTTCTTGAT
TAAGTCTTCTCATTTGATCAATTGGATATCTATCTTTCTTAGCACCAAGTTGTTTCACAGCT
TCTAATATGTTTTGTTCCGTTTTCTTAGCAAATCCGCTTAATTCACTAACTTTTCCATTTTC
ACAAGCAACTTGAAGTGACGCTTTATCAACAATATTCAACTCTTTATATAGCTTAGCAATTT
TCTTGCTTCCAAGTCCTTGAATTTTCAAAAGTGGAATAAGACCTTCCGGAACTTCTTCCTGT
AATTGCTGTAAATACTGAGATTCACCGGTCTCACGGTAATCATTGATTACTTCTGCAACACC
TTTACCAATGCCTTTTAACTCCGTTACATCAGATATTTCATCTA
>contig 8   long. 609
ACTCAATTTTGCAATTGTATTTGCCGTTGCAGGTGCAACAATGATTGCATCTGCCCAATCAC
CTAATGCAATATGCTGTATTTCTGAAGGATTTTCTTCTATAAAAGTATCTGTATAAACAGCA
TTTCGACTTATTGCTTGAAATGCTAATGGTGTCACAAATTTTTGTGCGTGATTCGTTAACAT
AACGCGAACTTCATACCCAGATTGTGTTAACTTACTTGTCAAATCAATTGCTTTATATGCCG
CAATGCCACCTGTAACGGCTAATAATATTTTCTTCATATTCAATCTCCCTTAAATATCACTA
TGACATTTACGCTTTACATCATCATATGCGCACAAATGCTCATTACTTTTTTATAGATACAA
ATTTAGTATTATTATAACATCAATCATTGGATAAACTAAAAAAACACACCTACATAGGTGCG
TTTGATTTGGATATGCCTTGACGTATTTGATGTACGTCTAGCTTCACATATTTTTAATGGTC
GAAACTATTCTTTACCATAATAATCACTTGAAATAACAGGGCGAATTTTACCGTCAGCAATT
TCTTCTAACGCTCTACCAACTGGTTTAAATGAATGATATTCACTTAATAAT
```

Figura 24. Muestra de registros de *contigs* en formato FASTA.

5.2 Árbol de Prefijos, Lista de Pares y Lista de Adyacencia

El árbol de prefijos o *Trie* es una estructura de datos que ofrece, mediante operaciones mínimas, la búsqueda de coincidencias entre fragmentos para generar los vértices del dígrafo. Estas coincidencias son el valor del traslape entre dos fragmentos y con esos elementos se construye el dígrafo, mismo que quedará representado por la lista de pares o lista de aristas. Para la construcción del *Trie* se lee cada fragmento y se busca su localización entre cuatro elementos de cada nodo (A, C, G, T); si el nodo ya existe comparte la información hasta llegar a un punto donde, si coincide al 100% se considera duplicado y si no, se generan nuevas derivaciones de nodos, mismos que podrían servir para compartir con los siguientes fragmentos. En la figura 25 se muestra un ejemplo de *Trie*. En esta figura se ejemplifica la construcción de *Trie* considerando que el primer fragmento es AGGTCGA. Se acomoda a partir del nodo superior que la primera vez está vacío y en la medida en que baja, va generando nuevos nodos. El siguiente fragmento es AGGTTTC. El prefijo AGGT ya existe, así que solo lo recorre y la parte restante del fragmento TTC, genera nuevos nodos. Ahora llega el fragmento AGGCCTC y se acomoda el prefijo AGG; el restante CCTC genera nuevos nodos.

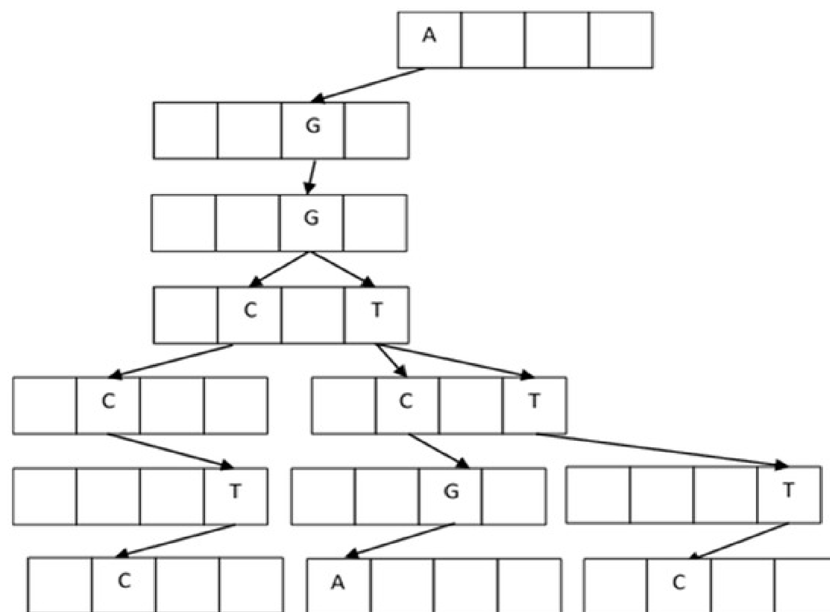


Figura 25. *TRIE*.

Tomado de (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017)

La construcción del *Trie* presenta una eficiencia O(ln), donde l es la longitud del fragmento y n la cantidad total de fragmentos, sin embargo, el número de coincidencias es alto por lo que se gana eficiencia en tiempo y espacio.

El algoritmo para la construcción es en realidad simple, ya que solo se requiere saber si el nodo que necesita la siguiente letra del fragmento está ocupada o no. El algoritmo 2 (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017) expone esta condición.

Algoritmo 2. Construcción del *Trie*.
Tomado de (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017)

1. *For each fragment*
- ***1.1*** *For each letter x from the fragment*
  - ***1.1.1*** *If box node(x) is free:*
    - ***1.1.1.1*** *Take up the box and create new empty node*
  - ***1.1.2*** *Else*
    - ***1.1.2.1*** *Go to next node*

El proceso de búsqueda de traslapes es restando una posición cada vez al fragmento en proceso y buscando en el *Trie* la coincidencia con el resto de las bases. Si se encuentra la coincidencia se genera una arista del grafo con sus vértices de origen y destino y el peso de la arista es el valor de traslape encontrado. Si se ha recorrido todo el fragmento hasta el límite de valor de traslape y no se localiza coincidencia, se desecha el fragmento en proceso. En el peor de los casos la eficiencia es O(l) (donde l es la longitud del fragmento), es decir, se recorrió todo el fragmento y no encontró coincidencia. El algoritmo 3 (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017) expone esta

Algoritmo 3. Búsqueda de un fragmento en el *Trie*.
Tomado de (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017)

1. *For each fragment (l)*
- ***1.1*** *For each fragment (l ← l-1 until overlap limit value)*
  - ***1.1.1*** *For each fragment-letter vs. Trie-node-letter*
  - ***1.1.1.a*** *If equal: continue to next fragment-letter and node*
  - ***1.1.1.b*** *If empty box: finish cycle 1.1*
- ***1.2*** *If end of fragment:*
  - ***1.2.1*** *Drain the branch*
  - ***1.2.2*** *Identify both fragments and create an edge (from, to, overlap)*

condición.

La lista de pares generada del *Trie* es una Lista de Aristas que se convierte en Lista de Adyacencia. Esta estructura de datos es del tipo convencional, y se consideró para su construcción una sección de cabeceras; cada cabecera es un nodo que probablemente tenga derivaciones en el grafo. El número de cabeceras ya se conoce y es el valor máximo de la lista de pares, previamente construido en un archivo del tipo secuencial. Por cada arista leída de la lista de pares, se localiza su vértice origen y se genera un nuevo nodo adyacente con el valor del peso de la arista, quedando entonces ligados todos los vértices adyacentes. El concepto de Lista de Adyacencia se muestra en la figura 26.

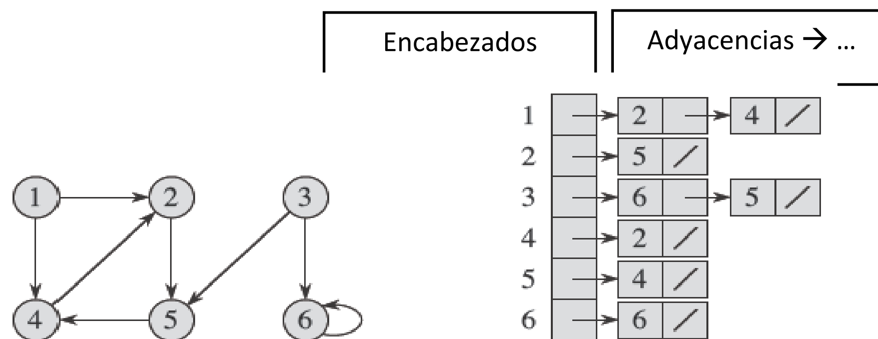


Figura 26. Lista de Adyacencia de un dígrafo.

Adaptado de (Cormen & Balkom, s.f., pág. 590)

5.3 Algoritmos Auxiliares

Los algoritmos y técnicas de manejo de datos que se describen a continuación jugaron un rol importante en la investigación ya que en la búsqueda de soluciones se encontraron situaciones particulares que requirieron de un análisis de datos y planteamiento a probables soluciones. En la versión final del programa de ensamble de fragmentos no se consideraron, sin embargo, los resultados que arrojaron estas técnicas sirvieron de pauta para resolver problemas de los algoritmos principales.

Montículo de pilas (Mallén-Fullerton, Quiroz-Ibarra, Miranda, & Fernández-Anaya, 2015, pág. 3).

Una técnica para resolver el problema de los recorridos en el grafo, resultante del proceso de detección de traslapes, es mediante el árbol de expansión mínima (*MST* minimum spanning tree). Esta técnica va a seleccionar, a partir de un vértice arbitrario del grafo, el camino más largo y una vez encontrado el punto final, a partir de este último vértice, vuelve a buscar el camino más largo. El último destino alcanzado se va a considerar como la "raíz del grafo" y a partir de este punto encontrará caminos múltiples en el grafo, ya sea mediante el algoritmo de Prim (Prim, 1957) o el de Kruskal (Kruskal, 1956). Estas técnicas tienen una complejidad $O(|E| \log |V|)$, donde $|V|$ es la cantidad de vértices del grafo y $|E|$ es la cantidad de aristas del grafo $G=(V, E)$. Para mejorar dicha complejidad se optó por asociar un montículo de pilas a cada vértice y en éste almacenar las adyacencias del vértice. La estructura de datos del tipo Pila muestra un grado de complejidad $O(1)$, tanto para almacenar como para extraer información, ya que el acceso es directo y de una sola instrucción, ganando con esto en eficiencia respecto a los algoritmos de Prim y Kruskal. En el artículo (Mallén-Fullerton, Quiroz-Ibarra, Miranda, & Fernández-Anaya, 2015, págs. 3.1, 3.2, 3.3) se pueden revisar los algoritmos. Dado que la localización de Prim y Kruskal son algoritmos *greedy*, al igual que el *MST*, en la versión final de la programación no se aplicaron, sin embargo, la experiencia aportó nuevas ideas para la implementación y búsqueda de recorridos en el grafo.

Heurística para ciclos (Mallén-Fullerton, Quiroz-Ibarra, Miranda, & Fernández-Anaya, 2015, pág. 2.2).

El resultado del proceso de detección de traslapes puede generar ciclos en el grafo y esto conlleva a un problema en la búsqueda del camino más largo, mismo que conformaría un *contig*. Una vez detectados algunos casos de prueba se optó por una técnica heurística de programación para genera la solución con mejor posibilidad de resolver el problema. En el dígrafo de la figura 27 se muestra un problema de recorridos debido a un ciclo en los vértices 8, 9 y 10. El método heurístico consiste en identificar que al saber que

ya ha pasado por un vértice, deberá regresar a buscar un camino alterno. En un dígrafo esto podría parecer más simple, ya que el mismo flujo de las aristas puede ser una pauta importante para el regreso, hasta lograr un camino que no se ha descubierto. Para el caso de la figura 27, si el recorrido es por los vértices *3→6→9* o *2→5→6→9* o *1→5→6→9* por ser la arista de 24 más larga, va a tomar este camino y llegará a *8→9*, pero *9* ya fue utilizado, por lo que el regreso es *8,10,9* y la única salida disponible en *9* es hacia *11*. Si el recorrido es de *3→8→9→10→8*, pero *8* ya fue utilizado, regresa por *10, 9* y la única salida de nuevo es por *11*.

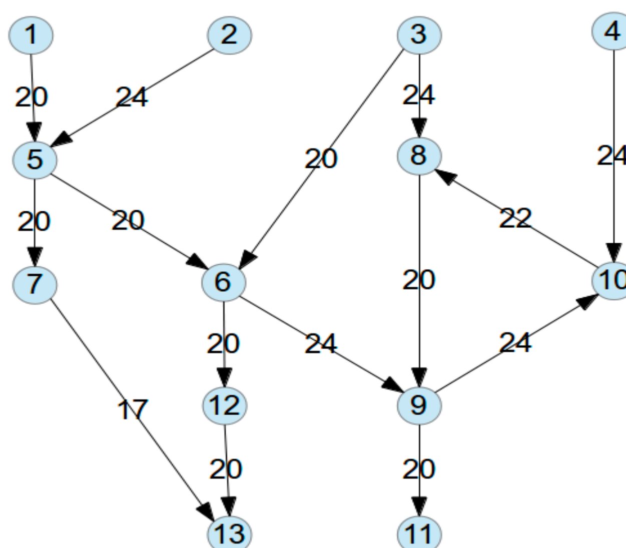


Figura 27. Ejemplo de grafo con ciclo en recorrido.
Tomado de (Mallén-Fullerton, Quiroz-Ibarra, Miranda, & Fernández-Anaya, 2015)

La heurística tiene el inconveniente de verificar cada vez que llega a un vértice si ya fue procesado o no, sin embargo, esta verificación ayudará a asegurar que la búsqueda del recorrido más largo no caiga en una condición de ciclo indefinido.

Transitivas.

En un grafo *G=(V,E)* cualquiera, se pueden encontrar elementos transitivos. Un elemento transitivo se identifica mediante las relaciones *R* de vértices que cumplan con la siguiente proposición:

$$\forall a,b,c \in G: \quad aRb \wedge bRc \rightarrow aRc \qquad \text{(ecuación 5.1)}$$

Esta es la propiedad de transitividad y se lee como:

"*Si a está relacionada con b y b está relacionada con c, entonces a está relacionada con c*". En forma gráfica, estas relaciones se verían como lo indica la figura 28.

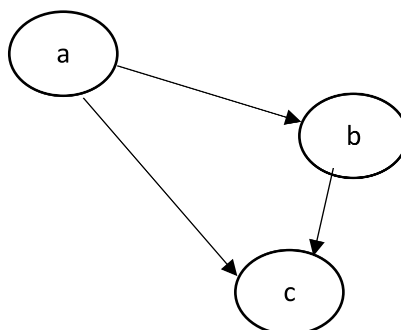


Figura 28. Transitividad. La arista *aRc* es transitiva.

Las aristas transitivas pueden ser un inconveniente para el recorrido del grafo, ya que, si se está buscando el recorrido de camino de mayor peso, una arista transitiva podría generar un camino de menor valor. El algoritmo de búsqueda del camino más largo debe garantizar la mejor ruta, sin embargo, las transitivas generarán más trabajo en el proceso del algoritmo. La eliminación de elementos transitivos puede ahorrar una gran cantidad de trabajo de procesamiento del computador.

En la búsqueda de métodos para identificar los vértices transitivos, se encontró que *SCIDB* (Brown, 2011) ofrecía una forma de análisis de datos para la identificación. *SCIDB* es un manejador de base de datos para análisis de datos científicos, que soporta los formatos matriciales de alto volumen, como lo es nuestro caso de fragmentos de ADN. *SCIDB* es desarrollado por *Paradigm4-Labs* en Waltham, Mas. USA. (http://www.paradigm4.com). Se instaló y configuró *SCIDB* conforme a las instrucciones propias de Paradigm4; éstas se muestran con detalle en el anexo 8.2.

SCIDB maneja un modelo de datos de tipo matricial con características *ACID* (Atomicidad, Consistencia, Aislamiento y Durabilidad/persistencia) y con

un lenguaje muy parecido a *SQL*[20] (del tipo declarativo) para la manipulación y análisis de datos. Es una base de datos computacional, pero no es un *datawarehouse*, no es una base de datos de inteligencia de negocios y tampoco es una base de datos transaccional. Una base de datos relacional está enfocada en el manejo de las transacciones y no soporta ciertos formatos complejos, forzando a los usuarios al manejo de modelos del algebra relacional y reacomodos frecuentes de datos para dar cabida a las expectativas.

Los tipos de datos que maneja *SCIDB* son datos generados por una máquina o son datos científicos o datos espaciales, entre los principales. En el caso de los fragmentos de ADN los datos son generados por la máquina secuenciadora *Illumina*.

SCIDB agrega funcionalidades extras como particiones de datos multidimensionales, indexación implícita, actualización, manejo de versiones y una biblioteca de operaciones matemáticas, de la cual se tomó la función *SPGEMM* (Buluç, 2012) de multiplicación de matrices dispersas.

Para la identificación de los elementos transitivos se optó por el método del cuadrado de la matriz de adyacencia, considerando que una celda con valor diferente de cero que existe en la matriz original y en la matriz resultante del cuadrado de la misma, es una arista transitiva; en la figura 29 se muestra una matriz de adyacencia *A* y el resultado del cuadrado de *A*, es decir A^2.

		a	b	c	d	e
	a	0	1	**1**	0	0
	b	0	0	1	0	0
A =	c	0	0	0	1	**1**
	d	0	0	0	0	1
	e	0	0	0	0	0

		a	b	c	d	e
	a	0	0	**1**	1	1
	b	0	0	0	1	1
AxA =	c	0	0	0	0	**1**
	d	0	0	0	0	0
	e	0	0	0	0	0

Figura 29. Transitividad numérica. Los vértices que tienen valor diferente de cero tanto en **A** como en **AxA**, son transitivos.

[20] *SQL*: Structured
pág. 4).

La matriz de adyacencia es dispersa y de muy alto volumen por lo que efectuar una simple multiplicación de matrices no es una operación eficiente. La función *SPGEMM* de *SCIDB* es propia para multiplicación de matrices dispersas y de muy alto volumen. Con la ejecución de este comando se detectaron las coordenadas de los elementos transitivos, mismos que ahora sería posible eliminarlos de la lista de pares originales, obteniendo con esto una reducción significativa del volumen de datos y de operaciones. En el anexo 8.3 se muestra la secuencia de comandos para la obtención del resultado.

Con el propósito de generar una versión que se integrara al paquete de programación en lenguaje *C++,* se desarrolló un algoritmo con funcionalidades equivalentes a *SPGEMM*, conociendo los resultados que se deberían obtener. El algoritmo parte de la matriz de adyacencia con el principio de la ecuación 5.1 y que se muestra en el algoritmo 4. La eficiencia de este algoritmo es $O(n^3)$.

Algoritmo 4. Transitivos en matriz de adyacencia *M*

```
1 Para i ← 0 hasta N
2           Para j ← 0 hasta  N
3                Si M[i][j]
4                Para k ← 0 hasta N
5                Si M[j][k]
6                     Si M[i][k] ➔ M[i][k] es transitiva;
```

Cabe resaltar que construir una matriz de adyacencia no resultaba conveniente, por la misma característica de dispersión, por lo que se trabajó con la misma lista de pares a modo de lista de adyacencia. El método se ajustó como lo muestra el algoritmo 5 con un árbol binario de búsqueda, recorrido en-orden y búsquedas indexadas. La eficiencia de este algoritmo es $O(n^2 \log n)$, sin embargo requiere la construcción del árbol binario a partir de la lista de pares.

Algoritmo 5. Transitivos en lista de adyacencia L

```
1 Construye Arbol BST ← L (llave=del+al, valor=peso)
2 Recorre BST en-orden
3      para cada BST(del1, al1).valor ≠ 0
4           recorre BST en-orden desde del2 ← al1
5           para cada BST(del2, al2).valor ≠ 0
```

6	*Si BST(del1, al2).valor ≠ 0 ➔ BST(del1, al2) es transitiva;*

Para darle certidumbre al proceso, tanto de *C++* como de *SCIDB*, se elaboró un comparativo de cifras de control y tiempos de ejecución con diferentes tamaños de muestras de los organismos muestra de *Staphylococcus Aureus* y de *Rhodobacteria Sphaeroides*; se tabularon los resultados. El detalle de la programación en *C++* se adjunta en el anexo 8.4. En la figura 30 se muestra el detalle de las cifras de control de un proceso de detección de transitivas mediante el algoritmo 5 con *C++* y mediante operaciones de comandos de *SCIDB* con *SPGEMM*. En ésta puede observarse para el organismo *Staphylococcus Aureus* que existe cierta similitud en los tiempos y las cifras de control coinciden. Para el organismo *Rhodobacteria Sphaeroides* se discrepó fuertemente el comparativo de tiempos, debido al grado de complejidad del organismo y la consecuente saturación de memoria RAM de la computadora. En una valoración escalada de la *Rhodobacteria Sphaeroides* y cantidad adicional de memoria RAM se podría observar un crecimiento uniforme. La generación de pares a partir de los fragmentos originales se obtuvo con el programa *FRAGAPARES.cpp* (Anexo *FAP*).

	Organismo:	*Staphylococcus Aureus*					*Rhodobacteria*
	Fragmentos originales	22,446	89,717	298,193	1,094,711	1,928,438	2,277,948
	Pares obtenidos	507,565	1,719,620	5,794,534	10,995,061	14,860,536	39,090,200
	C++: Transitivos detectados	450,077	1,512,638	5,071,521	9,050,916	10,662,539	34,009,373
	scidb: transitivos detectados	450,077	1,512,638	5,071,521	9,051,113	10,661,720	34,011,340
tiempos (mseg)	construye BST	1,066.66	3,763.78	13,291.60	25,265.10	34,128.70	92,835.70
	elimina Trs	*14,337.80*	*35,076.00*	*125,850.00*	*683,135.00*	*711,887.00*	*637,978.00*
	guarda en archivo secuencial	14,389.10	35,247.00	126,436.00	684,483.00	714,164.00	641,987.00
c++	**total (mseg)**	**15,455.76**	**39,010.78**	**139,727.60**	**709,748.10**	**748,292.70**	**734,822.70**
tiempos (mseg)	carga lista en SCIDB	2,356.00	7,639.00	25,487.00	48,692.00	66,289.00	172,725.00
	convierte lista en matriz de adyacencia	10,000.00	33,600.00	63,322.00	223,138.00	303,897.00	1,569,453.00
	obtiene transitivas con SPGEMM	6,602.00	18,481.00	59,879.00	145,449.00	165,568.00	9,226,241.00
scidb	**total (mseg)**	**18,958.00**	**59,720.00**	**148,688.00**	**417,279.00**	**535,754.00**	**10,968,419.00**

Figura 30. Tabla comparativa *C++* vs *SCIDB*.

En forma gráfica (figura 31) y considerando únicamente las muestras de *Staphylococcus Aureus* se observa el crecimiento proporcional entre ambos procesos, por lo que podemos concluir que existe certeza en los resultados.

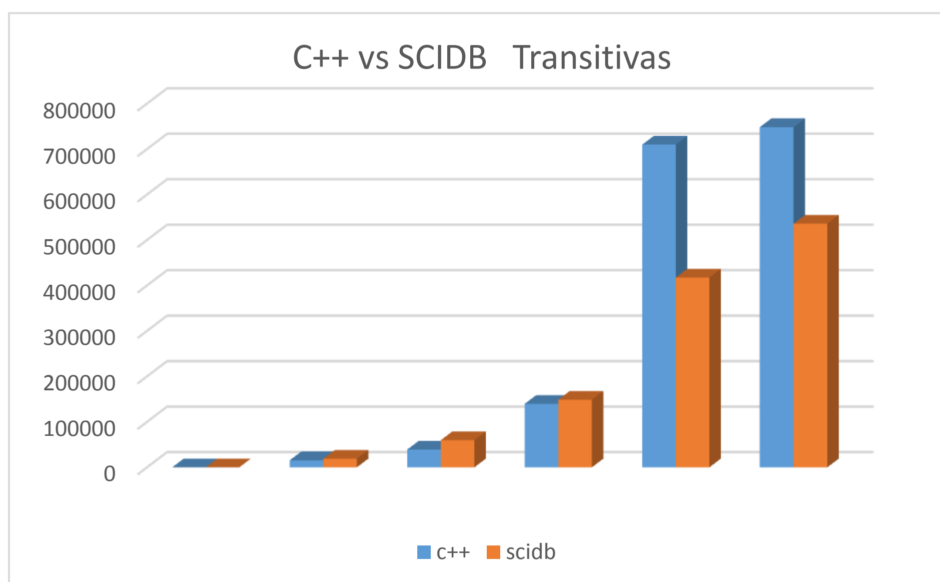


Figura 31. Gráfica comparativa de proceso de detección de Transitivas para el *Staphylococcus Aureus*.

Se efectuaron algunas pruebas aisladas y se comprobó que al eliminar las aristas transitivas se depura el volumen de datos y se gana eficiencia en la generación de resultados, sin alterar el resultado de los *contigs*. Sin embargo, ha quedado pendiente la incorporación de estos algoritmos, ya que, como se muestra en la gráfica de la figura 31 el comportamiento en las primeras muestras del *Staphylococcus Aureus* es eficiente, sin embargo, en la medida en que crece el tamaño de la muestra, se pierde eficiencia. Con respecto a *SCIDB* que conserva una mejor tendencia de velocidad de respuesta. Los algoritmos deberán revisarse antes de incorporarlos al paquete final de programación.

Al contar con la lista de pares cargada en *SCIDB* se tiene una serie de posibles operaciones de análisis disponibles mediante el uso de los mismos comandos de *SCIDB*, ya que la lista de pares es el grafo dirigido que representa a toda la secuenciación del organismo en estudio y el acceso a estos datos permite, con elementos de programación mínimos, generar un sinnúmero de consultas de combinaciones de los elementos pares del grafo para así poder decidir cuáles son las mejores opciones para la manipulación de dicho grafo y llegar a un buen resultado en el ensamble de los fragmentos de ADN.

El uso de *SCIDB* es recomendable para el estudio de la forma de los datos, sus relaciones, su comportamiento, sus volúmenes, variaciones, entre otras cualidades y para determinar características de interés propias de la muestra de datos.

5.4 Identificación de los caminos más largos, conformación de un contig aplicando consensos e intervalos de confirmación

Para la obtención de un *contig*, se deben buscar los caminos más largos a partir de todos los vértices que tengan grado de entrada cero, considerados todos estos como punto de partida de un recorrido del grafo. Se inicia el recorrido marcando el peso acumulado que va dejando cada arista, hasta llegar a un vértice con grado de salida cero. Este es un final del recorrido y ahora a partir de este mismo punto se elabora un recorrido en reversa buscando por dónde había un mejor peso generado en las derivaciones del grafo, hasta llegar nuevamente a un vértice de inicio o de grado de entrada cero. La lista de aristas involucradas en cada recorrido es un *contig*. El algoritmo 6 (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017) muestra la forma de llevar a cabo este recorrido.

Algoritmo 6. Ensamble de *Contigs*.
(tomado de (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017))

```
1. For all elements with Din = 0
        1.1 Dout minus 1
        1.2 For each adjacency element until Dout = 0
                1.2.1 Accumulate weight
        1.3 If accumulated weight > previous weight
                1.3.1 Label maximum path
```

En la medida en que se conforma el *contig* se genera un contador indicando cuantas aristas están participando en el proceso, de tal forma que al terminar, se analiza esta lista de valores y se determina si se cumple con el consenso definido. Hacia los extremos de este *contig* se eliminarán las secciones que no cumplan con el consenso. En la figura 32 (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017) se muestra el efecto del consenso en el proceso del ensamble de cada *contig*.

	T	T	C	A	C	T	T	A	T	T	T	A	A	A	A	T	C	T	G	G	A	A	G	A	A	A	C	C	T	A	G																					
			C	A	C	T	T	A	T	T	T	A	A	A	A	T	C	T	G	G	A	A	G	A	A	A	C	C	T	A	G	G	C	C	T	G	A	A														
						T	T	A	T	T	T	A	A	A	A	T	C	T	G	G	A	A	G	A	A	A	C	C	T	A	G	G	C	C	T	G	A	A	C	G	G	T										
									T	T	T	A	A	A	A	T	C	T	G	G	A	A	G	A	A	A	C	C	T	A	G	G	C	C	T	G	A	A	C	G	G	T	A	A								
												A	A	A	A	T	C	T	G	G	A	A	G	A	A	A	C	C	T	A	G	G	C	C	T	G	A	A	C	G	G	T	A	A	T	T	G					
													A	A	A	T	C	T	G	G	A	A	G	A	A	A	C	C	T	A	G	G	C	C	T	G	A	A	C	G	G	T	A	A	T	T	G	G	G	T	T	T
A:				2				3				5	6	6	6						6	6		6	6	6				6							5	5					3	3								
C:			2		2												6										6	6					5	5					4													
G:																			6	6			6								6	5				5				4	4						2	1	1			
T:	1	1				3	3		4	4	4					6		6												6						5							4			2	2				1	1	1
									T	T	T	A	A	A	A	T	C	T	G	G	A	A	G	A	A	A	C	C	T	A	G	G	C	C	T	G	A	A	C	G	G	T										
	|	<	-	-	-	-	>	|	<	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	>	|	<	-	-	-	-	-	-	>	|
	bases with low consensus							|	bases with good consensus value																																			|	bases with low consensus value								

Figura 32. Efecto del consenso en el ensamble de un *contig*.

Tomado de (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017)

El siguiente paso es garantizar que el resultado del *contig* contenga intervalos delimitados por los registros tipo "/1" y "/2". Al asegurar la existencia de estos intervalos se obtendrá un *contig* de calidad. En la figura 33 se muestran dos casos de *contigs* antes de ser depurados por el proceso de selección basado en los intervalos. Un intervalo que abre y cierra dentro del *contig* genera una confirmación del contenido y se presentarán ciertos traslapes entre los intervalos, generando esto a su vez una continuidad de los fragmentos ensamblados, hasta el final del *contig* o que se presenten discontinuidades de los intervalos.

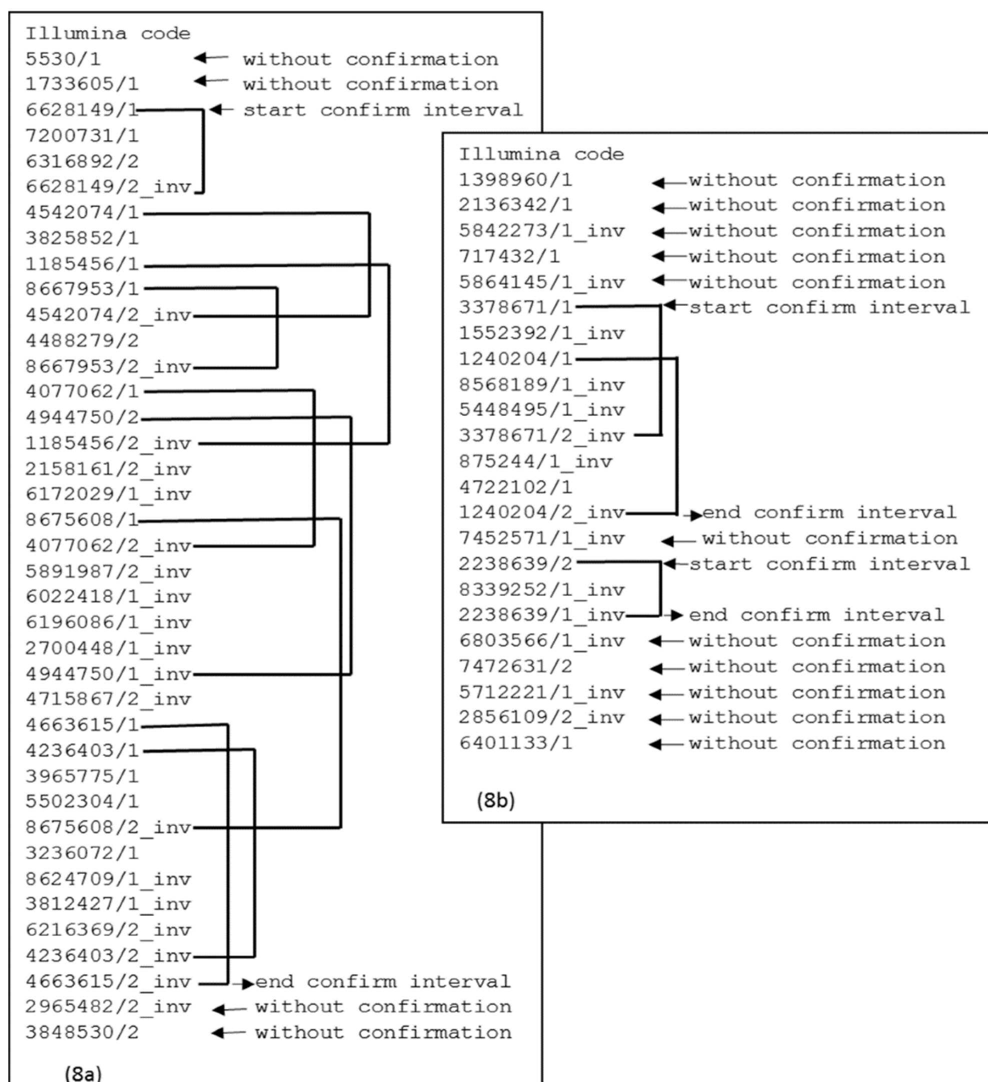


Figura 33. Intervalos de confirmación. En (a) se obtiene un intervalo continuo excepto en los extremos, los cuales se pueden desechar. En (b) hay una interrupción en el *contig* y esto provoca que el *contig* se parta en dos.

Tomado de (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017)

El proceso de generación de intervalos de confirmación genera un nivel muy alto de calidad en los resultados, desafortunadamente se truncaron muchos *contigs* y esto generó una caída en las estadísticas resultantes, en la cual se busca un *contig* lo más grande posible. El proceso de intervalos debe revisarse.

5.5 Evaluación de la calidad de los contigs

En la primera versión de la programación de ensamble de fragmentos, se trabajó con una muestra de *Staphylococcus Aureus* pero con registros de tipo 1 solamente, ya que no se sabía aún que existían los registros apareados (tipos 1

y 2). El reporte de resultados se puede ver en (Mallén-Fullerton, Quiroz-Ibarra, Miranda, & Fernández-Anaya, 2015). En la búsqueda de nuevas muestras de datos, se encontraron los registros apareados. Estos registros se pueden obtener de intervalos lineales o de segmentos circulares. Se decidió trabajar con intervalos lineales, ya que el ensamble obedece a una configuración lineal también. Se localizó una muestra de varios organismos, pero para darle continuidad al trabajo, se decidió también hacer el ensamble y todo el trabajo comparativo con *Staphylococcus Aureus*. La información se obtuvo del sitio web *GAGE* (Genome Assembly Gold-Standard Evaluationes http://gage.cbcb.umd.edu/). En este sitio se reportan los organismos de *Staphylococcus aureus, Rhodobacter sphaeroides, Humano* (chromosome 14) y *Bombus impatiens* (una especie de abeja). Se llevaron a cabo algunas pruebas aisladas también con *Rhodobacter sphaeroides*, para el caso de manejo de aristas transitivas, dado el grado de complejidad de este organismo.

El detalle de los datos del *Staphylococcus Aureus* es: 647,062 fragmentos de longitud fija de 101 bases; la bacteria tiene un genoma con 2,914,007 bases, un plásmido con 27,428 bases y otro plásmido con 3,170 bases. Este mismo juego de datos se aplicó a la programación de *Velvet*(v.1.2.10) (Zerbino, 2011) y a la de *Edena*(v.3.13) (Hernandez, s.f.). *Velvet* trabaja con un grafo de *de Bruijn* por lo que requiere un valor de *k-mer*; se asignó el valor de 31. *Edena* trabaja con traslapes y después de varias ejecuciones, se determinó que el valor óptimo para este juego de datos es de 30 bases. Tanto *Velvet* como *Edena* están preparados para trabajar con los registros apareados. Con la programación propia de este proyecto, se generaron dos resultados. En el primero (*Versión A*) solo incluye los registros apareados, pero no hace operación alguna con los intervalos de confirmación. En el segundo (*Versión B*) se determinó, a partir de los intervalos de confianza, hacer los cortes necesarios. En la *versión B* se determinó eliminar los *contigs* que tuvieran una longitud menor o igual a 1.5 veces el tamaño del fragmento original. *Edena* hace esta misma operación, por lo que obtuvo congruencia con esta medida. Este ajuste aportó una mejoría en el *N50*. También se eliminaron los fragmentos que presentaron repeticiones de una misma base a partir de 15 veces. En la figura 34 se muestran los resultados de estas ejecuciones.

Organismo:		Staphylococcus Aureus		Genoma: 2,914,007		Plasmido1: 27,428		Plasmido2: 3,170	
		Generados			Encontrados en genoma con Perl				
Pro-grama	Para-metro	*Contigs*	Long. Prom.	**N50**	*Contigs*	% vs gene-rados	**N50**	Bases	% vs genoma	**Mummer al 100%**
Velvet	K-mer =31	1,059	2,404	**6349**	907	85.6 %	**5724**	2,153,717	73.1 %	**67.3 %**
Edena	Traslape =30	3,038	796	**1168**	3,020	99.4 %	**1159**	2,380,777	80.8 %	**86.6 %**
Versión A	Traslape =50	2,985	860	**1325**	2,947	98.7 %	**1306**	2,536,435	86.1 %	**84.3 %**
Versión B	Traslape =50 Con interv. de conf.	7,700	368	**418**	7,610	98.8 %	**418**	2,798,144	95.0 %	**93.4 %**

Figura 34. Resultados del proceso de ensamble de *contigs*.

Se aplicaron dos técnicas de localización de los *contigs* en el genoma original. La primera fue con un programa en *Perl* desarrollado en el proyecto; la segunda fue con el programa de uso cotidiano *Mummer* (Kurts, 2004). El programa de *Perl* busca en el genoma original rigurosamente, es decir, tiene que estar el 100% del *contig* en el genoma para que sea válido. El programa de *Mummer*, aunque se tomaron los encontrados al 100%, en la búsqueda puede adaptar alguna condición en los extremos de los *contigs*, por lo que no se obtiene una gran precisión. En esta tabla de resultados (figura 34) se puede observar que en *Versión A* se genera un N50 superior a *Edena* pero al aplicar los intervalos de confianza (*Version B*), se cae el valor. Esto se debió a que se fraccionaron muchos *contigs* al tener discontinuidades de los intervalos. Sin embargo, estos *contigs* generados fueron de muy alta calidad, aunque demasiado cortos y tanto *Mummer* como el programa de Perl obtuvieron el mayor valor de localizaciones en el genoma.

La revista *MDPI,* en la que se publicaron los artículos de este proyecto, solicitó para la segunda publicación (Quiroz-Ibarra, Mallén-Fullerton, &

Fernández-Anaya, 2017), la ejecución de un *benchmark* con el software de *QUAST* (v.4.4) (Center for Algorithmic Biotechnology. St. Petersburg, Russia) (Gurevich, Saveliev, Vyahhi, & Tesler, 2013). Los resultados de *Quast* se muestran en la figura 35.

Organismo:	Staphylococcus aureus						Genoma: 2,903,081 con 2 plasmidos			
		Generados (estadísticas sin referencia)					Encontrados en el genoma (estadísticas con el genoma de referencia)			
Programa	Parametro	*Contigs* Generados	*Contigs* en el ensamble	Total de bases en el ensamble	Total de bases en los *contigs*	**N50 de los *contigs***	**NG50 de los *contigs* encontrados**	Total número de bases alineadas	Fracción del genoma %	***Contigs* encontrados por Mummer en el genoma al 100%**
Velvet	k-mer =31	1,059	693	2,733,950	2,816,990	**6666**	**6,216**	2,733,853	93.942%	**67.3%**
Edena	overlap =30	3,038	1670	2,017,901	2,419,142	**1380**	**938**	2,017,901	69.39%	**86.6%**
Version A	overlap =50	2,985	1701	2,234,717	2,619,179	**1539**	**1,183**	2,233,545	76.326%	**84.3%**
Version B	overlap =50 c/ intervalo de confirm.	20,439	2090	1,455,722	6,016,455	**675**	**500**	1,437,787	25.986%	**93.4%**

Figura 35. Resultados valorados por Quast.
Tomado de (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017)

En el resultado de *Quast* se puede observar que el programa *Version A* supera a *Edena* pero no a *Velvet*. Una consideración importante al respecto es que *Velvet* utiliza el grafo de *de Bruijn* y tanto *Edena* como *Version A* trabajan con los traslapes del grafo y un grafo dirigido. Se agregó la columna resultante de *Mummer* con la intención de no perder las referencias entre una tabla de resultados y otra.

6. Conclusiones y Trabajos futuros

Se plantearon dos técnicas de ensamble de fragmentos de ADN. La primera basada en optimización y heurísticas con resultados certeros, pero con la interrogante sobre dónde partir los *contigs* para que sea garantizable su localización en el genoma original. Se discurrieron varias técnicas y de esto se

derivó el trabajo del segundo método en el que los intervalos de confianza podrían ser una herramienta para el criterio de fraccionamiento del *contig*.

Al aplicar los intervalos de confianza, se garantiza la calidad (Salzberg, y otros, 2012) del *contig* pero ahora es una secuencia muy corta por lo que habrá que revisar esta técnica de manejo de intervalos, buscando que no se corten los *contigs* o buscando como extender los intervalos incluso entre *contigs*. A pesar de este problema de intervalos, la creación de nuevos algoritmos, resultantes de las experiencias de las primeras pruebas de ensamble, permitieron obtener un resultado de mayor calidad (ver figura 34) en el que el *N50* mejora notablemente y supera a *Edena*.

Al plantear los nuevos algoritmos y revisar los anteriores, analizando su grado de complejidad, las estructuras de datos *adhoc* y las implicaciones de la forma de desarrollo se pudo identificar que ciertas operaciones en lenguaje de programación *C/C++* deben ser manipuladas al margen de los módulos de biblioteca que ofrece el mismo lenguaje de programación, ya que estos se construyen con generalidades, mismas que compromete la ejecución óptima.

También se identificaron algunos puntos críticos en los algoritmos, derivados del alto volumen de datos y que obligaron al uso de arquitectura computacional de 64 bits, capaz de soportar esta gran cantidad de datos de forma más eficiente. Esto mismo generó a su vez una limitante en el uso de arquitecturas de 32 bits con límites de memoria RAM y tamaño de archivos de datos.

Se cumplió con el objetivo sobre la obtención de un nuevo algoritmo basado en teoría de grafos, con resultados de alta calidad, eficiencia y rapidez. La parte de calidad es visible en la figura 35 ya que el *N50* de *contigs* generados es alto, aunado esto a los *contigs* encontrados en el genoma. La eficiencia y rapidez, como se ve en la figura 36, reportado en (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017) es, comparando con *Edena*, notablemente mejor. La comparación con *Edena* es efectiva ya que hasta donde se logró documentar, utiliza algunos algoritmos similares a los implementados en este proyecto más algunas heurísticas que se mencionan en la documentación (Hernandez, s.f.), pero no se detallan.

Program	Step	Execution Time (s)
Velvet	Velveth	23.085
	Velvetg	6.397
Edena	Edena–DR	363.676
	Edena-e	2.051
Version A	FragAPares	69.140
	ListAdy	5.002
Version B	FragAPares	69.140
	ListAdyU	5.834

Figura 36. Tiempos de ejecución.
Tomado de (Quiroz-Ibarra, Mallén-Fullerton, & Fernández-Anaya, 2017)

Los algoritmos son mejorables en muchos sentidos, pero se debe vigilar no solo la eficiencia de los algoritmos sino también el resultado que se obtiene. Algunos aspectos de mejora y que son candidatos para trabajos futuros son el manejo de los intervalos de confianza, con una nueva perspectiva a fin de garantizar que el contenido del intervalo esté al 100% de calidad y que sea posible darles continuidad a los diferentes intervalos, buscando con esto un *contig* más grande.

Un elemento que debe ser motivo de análisis y búsqueda de soluciones es el manejo de los valores "N" contenidos en algunos fragmentos entregados por *Illumina*, es decir, que sea posible determinar su valor con precisión o utilizarlas como tipo comodín, ya que se pierden muchas bases que bien podrían ser elementos importantes en el manejo de los intervalos de confianza.

La información sobre calidad en los registros de *Illumina*, no fue considerada, sin embargo, esto podría arrojar la posibilidad de un mejor método de consenso en el conteo vertical de bases ensambladas del *contig*, produciendo así registros de mayor longitud y nuevamente un mejor intervalo de confianza.

La eliminación de elementos transitivos del grafo es un recurso que, aunque se desarrollaron algoritmos y se generaron técnicas con *SCIDB* para garantizar dichos elementos, requiere de un algoritmo incorporado en la

misma etapa de determinación del grafo, ya que de esta forma se puede eliminar un volumen de operaciones y de datos innecesarios. *SCIDB* ofrece ser una herramienta para análisis de datos de alto volumen que puede ayudar a tomar decisiones respecto a cómo implementar mejores algoritmos y estructuras de datos.

En resúmen, las aportaciones de mayor revelancia de este trabajo de investigación son las siguientes:

- Integración de la solución al problema del ensamble de fragmentos de ADN mediante estructuras de datos y algoritmos alternativos, aplicado a organismos reales.
- Desarrollo de los algoritmos con la función objetivo “el camino más largo” en el grafo dirigido, a partir de un vértice con grado de entrada en cero y hasta un vértice con grado de salida en cero, validando la presencia de ciclos en el grafo.
- Identificación de los registros apareados que delimitan un intervalo de secuencias de bases, cuyo contenido se garantiza como parte del genoma.
- Aseguramiento de la calidad de un *contig* en un proceso *de novo* usando los intervalos de confirmación que generan los registros apareados y que se presentan conectados entre sí.
- Con el intervalo de confirmación establecido por los registros apareados y detectando discontinuidades en dicho intervalo, se determina el punto de corte de un *contig* ensamblado erróneamente con traslapes falsos.
- Incorporación de la detección de fragmentos del grafo dirigido del tipo transitivos y su aportación al proceso de ensamble de los *contigs*, afectando a la función objetivo del camino más largo.
- Aplicación de *Mummer* y *QUAST* para la valoración de resultados comparativos con *Edena* y *Velvet* en igualdad de condiciones operativas.

7. Referencias bibliográficas

Bang-Jensen, J., & Yeo, A. (2004). When the greedy algorithm fails. *Discret Optimization*, 121-127.

Barbadilla, A. (s.f.). *Ensayos sobre genética*. Recuperado el 5 de 11 de 2017, de Genotipo y Fenotipo.: http://bioinformatica.uab.es/base/base3.asp?sitio=ensayosgenetica&anar=conceptos&item=genoti

Brown, P. (2011). Big Data and Big Analytics: SciDB is not Hadoop. *Strata conference. Making data work.* Sutton South, UK.: O´Reilly.

Buluç, A. &. (2012). Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, C170-C191.

ChileBIO_CropLife. (s.f.). *El ADN, los genes y el código genético*. (A. G. CropLife, Editor) Recuperado el 10 de 09 de 2017, de http://www.chilebio.cl/wp-content/uploads/2015/07/1.jpg

Compeau, P., Pevzner, P., & Tesler, G. (2011). How to apply de Bruijn graphs to genome assembly. *Nature Biiotechonology*, 987-991.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms.* Cambridge, UK: MIT Press.

Cormen, T., & Balkom, D. (s.f.). *khanacademy.org*. Recuperado el 15 de 10 de 2017, de Khan Academy Notación Asintótica: https://es.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-big-theta-notation

Curtis, H., Barnes, N., Schnek, A., & Flores, G. (2006). *Biología.* Buenos Aires, Argentina: Médica Panamericana.

Date, C. (2000). *An Introduction to Database Systems* (8a. ed.). Addison Wesley Longman.

Dean, W. (2016). *Computational Complexity Theory.* (S. U. Metaphysics Research Lab, Ed.) Recuperado el 04 de 11 de 2017, de Stanford Encyclopedia of Philosophy: https://plato.stanford.edu/entries/computational-complexity/

Gingold, E. (1988). An introduction to genetic engineering. In E. Gingold, *Molecular biology and biotechnology* (pp. 25-46). London, U.K.: Royal Society of Chemistry.

Gurevich, A., Saveliev, V., Vyahhi, N., & Tesler, G. (2013). QUAST: Quality assessment tool for genome assemblies. *Bioinformatics, 29*, 1072–1075.

Hartmanis, J., & Hopcroft, J. E. (1971). An Overview of the Theory of Computational Complexity. *Journal of the ACM*, 444-475.

Hernandez, D. (s.f.). *Edena*. (G. R. Institute, Editor) Obtenido de http://www.genomic.ch/edena.php

Johnsonbaugh, R. (2005). *Matemáticas Discretas.* México: PEARSON EDUCACIÓN.

Kingston, H. M. (2002). *ABC of clinical genetics* (Third Edition ed.). London, England: BMJ Publishing Group.

Kruskal, J. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc., 7*, 48-50.

Kurts, S. (2004). Versatile and open software for comparing large genomes. *Genome Biology, 5*(2).

Levitin, A. (2012). *Introduction to the Design & Analysis of Algortihms, 3rd edition.* Addison-Wesley.

Mallen-Fullerton, G., & Fernandez-Anaya, G. (2013). DNA fragment assembly using optimization. *IEEE Congress on Evolutionary Computation*, (págs. 1570-1577). Cancun, México.

Mallén-Fullerton, G., Quiroz-Ibarra, J., Miranda, A., & Fernández-Anaya, G. (2015). Modified Classical Graph. *Algorithms*, 754-773.

Morin, E. (1994). *Introducción al pensamiento complejo (5a. reimpresión).* Barcelona: Gedisa.

Myers Jr, E. W. (2016). A history of DNA sequence assembly. *it-Information Technology, 58*(3), 126-132.

National Center for Biotechnology Information. (s.f.). *NCBI Genome & Maps*. Recuperado el 21 de noviembre de 2016, de https://www.ncbi.nlm.nih.gov/guide/genomes-maps/

Parsons, R., Burks, C., & Forrest, S. (1993). Genetic Algorithms for DNA Sequence Assembly. *ISMB*, 310-318.

Prim, R. (1957). Shortest connection networks and some generalizations. *Bell Syst. Tech. J., 36*, 1389–1401.

Quiroz-Ibarra, J., Mallén-Fullerton, G., & Fernández-Anaya, G. (2017). DNA Paired Fragment Assembly Using Graph Theory. *Algorithms*, 10,36.

Salamanca Gomez, F. (1962). *Citogenetica Humana.* México, D.F.: Médica Panamericana.

Salzberg, S., Phillippy, A., Zimin, A., Puiu, D., Magoc, T., Koren, S., . . . al., e. (2012). A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, 557-567.

Sanger, F., & Coulson, F. (1977). DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences of the United States of America, 74(12)*, 5463-5467.

Sanger, F., Coulson, A. R., Hong, G. F., Hill, D. F., & Petersen, G. B. (1982). Nucleotide sequence of bacteriophage λ DNA. *Journal of molecular biology, 162*(4), 729-773.

Staden, R. (1979). A strategy of DNA sequencing employing computer programs. *Nucleic acids research, 6*(7), 2601-2610.

Tammi, M. (2003). The principles of shotgun sequencing and automated fragment assembly. (K. I. Center for Genomics and Bioinformatics, Ed.)

Tutte, W. (2001). *Graph Theory.* Waterloo, Ontario, Canada: Cambridge University Press.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 249-260.

Watson, J. D., & Crick, F. H. (1953). A Structure for Deoxyribose Nucleic Acid. *Nature 171*, 737-738. doi:10.1038/171737a0

Wirth, N. (1976). *Algorithms Plus Data Structures.* Prentice-Hall.

Zerbino, D. (2011). *Velvet*. (EML-EBI, Editor) Obtenido de http://www.ebi.ac.uk/~zerbino/velvet/

8. Anexos

8.1 Manual de uso del software

Instrucciones de compilación:

Se deben descargar todos los contenidos en una carpeta claramente establecida en el disco duro local. Los archivos de la carpeta son:

DNAAssembler.jar,
FragAPares_12.cpp,
ListADYDFSEnsam_2.cpp,
EstructParesL.h,
Estructuras12.h.

Para compilar C++ en Windows (64 bits):

>g++ FragAPares_12.cpp -o FragAPares.exe -std=c++11

Nota. Debe estar presente el archivo Estructuras12.h

>g++ ListAdyDFSEnsam_2.cpp -o ListAdyDFSEnsam_2.exe -std=c++11

Nota. Debe estar presente el archivo EstructParesL.h

Para ejecutar desde consola:

>java -jar DNAAssembler.jar

Para compilar C++ en Linux (64 bits):

>g++ FragAPares_12.cpp -o FragAPares.exe -std=c++11

Nota. Debe estar presente el archivo Estructuras12.h

>g++ ListAdyDFSEnsam_2.cpp -o ListAdyDFSEnsam_2.exe -std=c++11

Nota. Debe estar presente el archivo EstructParesL.h

Para ejecutar desde consola:

>java -jar DNAAssembler.jar

8.2 Instalación de SCIDB en Linux

El procedimiento se realizó en una computadora de 64 bits con Ubuntu 14.04 LTE.

1. hacer el superusuario scidb/paradigm4
2. crear carpeta para scidb15 y cargar en esta el archivo *.tar*
3. activar la apertura de terminal desde cualquier sitio: "sudo apt-get install nautilus-open-terminal"
4. expandir el archivo *.tar* con: "tar -xvzf scidb-15.12.1.4cadab5.tar.gz"
5. renombrarse la carpeta como "scidbtrunk"
6. activar los servcios de ssh: "sudo apt-get -y install expect openssh-server openssh-client "
7. reinciar el servicio: "sudo service ssh restart"
8. verificar que este corriendo: "sudo service ssh status"
9. para levantar el servicio:
   "sudo /usr/sbin/update-rc.d ssh defaults"
   "sudo service ssh start"
10. para activar ssh sin passowrd: "ssh-keygen" (dar ENTER en todas las preguntas)
11. para dar permisos de ejecución:
   "chmod 755 ~"
   "chmod 755 ~/.ssh"

----

12. para que los siguientes comandos funcionen, hay que activar el superusuario *root*, como sigue:
- whoami ... dice que usuario eres
- sudo -i .... abre sesión de root .... exit la cierra

localizar /etc/ssh/sshd_config y editarlo como sudo. (conviene con nano...)
comentar la linea: #PermitRootLogin without-password
agregar la linea: PermitRootLogin yes
reinciar ssh... sudo service ssh restart
reinciar ubuntu completo

----

sudo passwd root ... 'para' es el nuevo password de unix
--->> aqui ejecutar las líneas del punto 13. <<---
sudo passwd -l root ... con esto volvemos a bloquear (lock) el usuario *root*
regresar las lineas de sshd_config nuevamente como sudo
reinciar ubuntu completo

----

13. para dar acceso..
cd <dev_dir>/scidbtrunk
deployment/deploy.sh access root "" "" localhost (o 127.0.0.1) ... ok
deployment/deploy.sh access scidb "" "" localhost (o 127.0.0.1) ... ok
14. para verificar el acceso a ssh: "ssh localhost date"
15. cd scidb/scidbtrunk ... deployment/deploy.sh prepare_toolchain localhost
<<< nota: debe funcionar y tarda mucho, descargas múltiples de web>>

----

16. instalacion de postgres: (conserva los metadatos de scidb)
Primero investigar la máscara con ifconfig.. dice 127.0.0.1 y 255.0.0. por lo que la máscara es: 127.0.0.0/8
el comando es:

deployment/deploy.sh prepare_postgresql postgres postgres 127.0.0.0/8 localhost ...ok

----

17. para dar derechos a postgres..
sudo usermod -G scidb -a postgres
chmod g+rx /home/scidb
y para verificar: /usr/bin/sudo -u postgres ls <dev_dir> # the 'postgres' user should have access.
o sea:
/usr/bin/sudo -u postgres ls /home/scidb .... y todo salió bien.

Aquí acaban las tares de preinstalación.

----

1. se modificó .bashrc en home y se agregó: <dev-dir> es scidb
export SCIDB_VER=<your-scidb-version> # i.e. 15.12
export SCIDB_INSTALL_PATH=/home/<dev_dir>/scidbtrunk/stage/install
export SCIDB_BUILD_TYPE=Debug
export PATH=$SCIDB_INSTALL_PATH/bin:$PATH
2. se ejecutó el comando "source .bashrc" ... para que aplique los cambios o se cierra y se abre de nuevo.
se ejecutaron los *echos*:
echo $SCIDB_VER
echo $SCIDB_INSTALL_PATH
echo $PATH .... y todo bien
3. se ejecutaron:
se creó una copia de *scidbtrunk* en */home/scidbtrunk* y desde este se están ejecutando los comandos, ya que se maneja erróneamente la ruta del *path* y otros errores
cd scidbtrunk ... para entrar a las carpetas
./run.py -h # to learn its usage. .... Para nada!
./run.py setup # to configure build directories and *cmake* infrastructure ...(pide borrar "all", le dije <enter>)
./run.py make -j4 # to build the sources, usa cuatro núcleos con el j4 -----ok
./run.py install #pide borrar todas las carpetas y le dije que si (y)! ... pidió cambio de password y le dí <enter> y luego pidió password de sudo, le dí paradigm4.

----

para iniciar scidb: > scidb.py startall mydb
para detener scidb: > scidb.py stopall mydb

8.3 Comandos de la evaluación de SCIDB vs C++

Archivo	fragmentos	pares	organismos	status
1. pares22k.dat	44,893/2 = 22,446	507,565	Saur	ok-scidb
2. pares89k.dat	179,435/2 = 89,717	1,719,620	Saur	ok-scidb
3. pares298k.dat	596,387/2 = 298,193	5,794,534	Saur	ok-scidb
4. pares1094k.dat	2,189,422/2 =1,094,711	10,995,061	Saur	ok-scidb
5. pares1928k.dat	3,856,877/2 =1,928,438	14,860,536	Saur	ok-scidb
6. pares2277k.dat	4,555,897/2 =2,277,948	39,090,200	RBac	ok-scidb

---

Resumen de los pasos ejecutados s partir del archivo de pares (22,446 de la primera muestra):

1. Convertir los archivos .dat a .csv
>tr -s ' ' ',' <pares22k.dat >pares22k.csv

2. Creación en SCIDB del espacio requerido para la lista de pares (507,570 pares de la primera muestra)
>iquery -anq "CREATE ARRAY paresK <del:uint32, al:uint32,tras:double NOT NULL>[x=0:507570,507571,0];"

3. Carga del archivo .csv al espacio de SCIDB
>iquery -anq "load (paresK, '/home/scidb/Documents/pares22k.csv',0,'csv');"

4. Determinación del valor máximo de pares:
(en modo AQL)>select max(del) from paresK;
{i} max
{0} 44892
----

5. Creación del espacio de la matriz de adyacencia
>iquery -anq "create array matrizK<tras:double NOT NULL>[del=0:44893,44894,0, al=0:44893,44894,0];"

6. Carga de la matriz de adyacencia a partir de la lista de pares con un comando de redimensionamiento:
>iquery -anq "store(redimension(paresK, matrizK),matrizK);"

7. Creación de la matriz que conservará el resultado:
>iquery -anq "create array matrizKRM<tras:double NOT NULL>[del=0:44893,44894,0, al=0:44893,44894,0];"

8. Carga del modulo de algebra:
(en modo AFL)>iquery/AFL: load_library('dense_linear_algebra');

9. Calculo de las transitivas:
>iquery -anq "store(spgemm(matrizK, matrizK),matrizKRM);"

10. Reporte de las transitivas:
>iquery -q "select count(*) from matrizKRM as md, matrizK as mr where md.del = mr.del and md.al=mr.al;"
{i} count
{0} 450077 <--- total de transitivas detectadas

Nota: esta secuencia se repitió para todas las muestras de *Staphylococcus Aureus* y *Rhodobacteria Sphaeroides*, como sigue (se expone solo para la primera muestra).

Paso 1. Convertir los archivos .dat a .csv

```
>tr -s ' ' ',' <pares22k.dat >pares22k.csv...(todos los archivos)
>se compactaron los .dat en un tar.gz y se eliminaron los .dat originales
=================================================================
Memoria RAM ocupada: 2.3GBytes
-------------------------------
Paso 2. Iniciar SCIDB
>scidb.py startall mydb (para detenerla: scidb.py stopall mydb)
-------------------------------
Memoria RAM ocupada: 2.4GBytes
-------------------------------
-------------------------------
c++ 22446:
inciado..leidos del archivo: 507565
en el arbol: 507565
tiempo(mseg) de construccion: 1066.06
borrados: 1211190
tiempo(mseg): de eliminar Trs: 14337.8
grabados: 57488
tiempo(mseg): de archivo: 14389.1
tiempo(mseg) total: 15455.2
-------------------------------
Paso 3.1. (para 22446) Se ejecutÃ³ desde comando la creación del espacio de paresK:
time iquery -anq "CREATE ARRAY paresK <del:uint32, al:uint32,tras:double NOT
NULL>[x=0:507570,507571,0];"
>>>>
Query was executed successfully

real	0m0.140s
user	0m0.012s
sys	0m0.000s
----------------------------------
Paso 4: Se cargaron los datos desde csv:
time iquery -anq "load ( paresK, '/home/scidb/Documents/pares22k.csv',0,'csv');"
>>>>
Query was executed successfully

real	0m2.356s
user	0m0.007s
sys	0m0.004s
----------------------------------
Paso 5: Diseño de la matriz de adyacencia:
en iquery en modo AQL:
select max(del) from paresK;
{i} max
{0} 44893
---------
select max(al) from paresK;
{i} max
{0} 44892
---------
Ahora se creará la matriz con los datos anteriores:
time iquery -anq "create array matrizK<tras:double NOT NULL>[del=0:44893,44894,0,
al=0:44893,44894,0];"
>>>>>>
Query was executed successfully

real	0m0.025s
user	0m0.011s
sys	0m0.000s
---------------------------------
```

Paso 6: Carga de la matriz de adyacencia a partir de la lista de pares con un redimensionamiento:
time iquery -anq "store(redimension(paresK, matrizK),matrizK);"
>>>>
Query was executed successfully

real 0m10.000s
user 0m0.009s
sys 0m0.003s

Paso 7: Creación de la matriz que conservará el resultado:
time iquery -anq "create array matrizKRM<tras:double NOT NULL>[del=0:44893,44894,0,
al=0:44893,44894,0];"
>>>>>>>>>>>>
Query was executed successfully

real 0m0.024s
user 0m0.011s
sys 0m0.000s

Paso 8: Calculo de las transitivas:
(primero cargar módulo de algebra) en iquery/AFL: load_library('dense_linear_algebra');
(segundo calcular las transitivas)
time iquery -anq "store(spgemm(matrizK, matrizK),matrizKRM);"
>>>>>
Query was executed successfully

real 0m6.602s
user 0m0.004s
sys 0m0.008s
=========================
.... en caso de error, corrida de prueba:
AFL% store(build(<val:double NOT NULL>[row=0:2,3,0, col=0:1,2,0],'[[(),4],[6,()],[9,10]]',true),A);
AFL% store(build(<val:double NOT NULL>[row=0:1,2,0, col=0:2,3,0],'[[3,(),()],[(),5,()]]',true),B);
AFL% spgemm(A, B);
{row,col} multiply
{0,1} 20
{1,0} 18
{2,0} 27
{2,1} 50
...
drop array A;
drop array B;
AFL% exit; ....
==========================
Paso 9: Reporte de las transitivas:
time iquery -q "select count(*) from matrizKRM as md, matrizK as mr where md.del = mr.del and
md.al=mr.al;"
>>>>>>>>>>>>>>
{i} count
{0} 450077 <--- total de transitivas detectadas

real 0m1.462s
user 0m0.006s
sys 0m0.006s

Paso 10:
borrado de los arreglos:
drop array paresK;
drop array matrizKRM;
drop array matrizK;

Fin del proceso 22K ---------Nota: se detiene el proceso en este punto ya que la descarga a disco tiene otros factores que influyen en el tiempo de respuesta y no se consideraron.

8.4 Programa de eliminación de Transitivas en C/C++

Anexo del programa en C++ que detecta y elimina las transitivas

```
<< archivo .cpp >>
#include "ETStrs.h"
void construyeArb(FILE *inFileP, map<string,int> &arbS);
void muestraArb(map<string,int> &arbS);
void eliminaTrs(map<string,int> &arbS);
void generaPares(FILE *outFileP, map<string,int> &arbS);
const int MAXREC = 25;

int main() {
   clock_t iniciaT = clock(), iniciaG = clock();
   inFileP  = fopen(PARESNVO,"r");

   map<string,int> arbS;
   //-----------------------------------
         construyeArb(inFileP, arbS);
         fclose(inFileP);
         cerr << "tiempo(mseg) de construccion: "<<
               (float)(clock()-iniciaT)*1000.0/CLOCKS_PER_SEC <<endl;
         iniciaT = clock();
         //muestraArb(arbS);

         eliminaTrs(arbS);

         //muestraArb(arbS);
         cerr << "tiempo(mseg): de eliminar Trs: "<<
               (float)(clock()-iniciaT)*1000.0/CLOCKS_PER_SEC <<endl;

         outFileP = fopen (PARESNVST, "w");
         generaPares(outFileP, arbS);
         fclose(outFileP);
         cerr << "tiempo(mseg): de archivo: "<<
               (float)(clock()-iniciaT)*1000.0/CLOCKS_PER_SEC <<endl;
         cerr << "tiempo(mseg) total: "<<
               (float)(clock()-iniciaG)*1000.0/CLOCKS_PER_SEC <<endl;
         //system("pause");
         //getchar();
}
void generaPares(FILE *outFileP, map<string,int> &arbS){
         long int grabados = 0;
         map<string,int>::iterator it = arbS.begin();
         RegPares rp;
         string s;
         for ( ; it != arbS.end(); it++) {
               if (it->second == 0) continue; // esta marcada como transitiva
               s = it->first;
               rp.asignaS(s, it->second);
               fprintf(outFileP, "%s %s %03d\n", rp.del, rp.al, rp.peso);
               grabados++;
         }
         cout << "grabados: " << grabados << endl;
}
void construyeArb(FILE *inFileP, map<string,int> &arbS){
         long int leidos = 0;
         char linea1 [MAXREC];
         fgets(linea1, MAXREC, inFileP);       // lee el primero
         map<string, int>::iterator iter;
```

```
	while (!feof(inFileP)){	//si eof esta sobre la última linea de datos, hay que hacer ultimo
		leidos++;
		regPar.asignaC(linea1);
		//arbS[regPar.genLlave()] = regPar;			// si esta duplicado, se
									reemplaza el valor (no
									conviene)
		arbS.insert(pair<string,int> (regPar.genLlave(), regPar.peso));//si esta duplicado,
										no lo hace..ok!
		fgets(linea1, MAXREC, inFileP);
	}

	cout << "leidos del archivo: " << leidos<<endl;
	cout << "en el arbol: "<< arbS.size() << endl;
}

void eliminaTrs(map<string,int> &arbS) {
	map<string, int>::iterator it1 = arbS.begin(), it2, it3;
	RegPares rpx, rpy;
	string aux, sx, sy;
	long int borrados=0;

	for (; it1 != arbS.end(); it1++){
		//if (it1->second == 0) continue; // ya es una transitiva
		sx=it1->first;			rpx.asignaS(sx, it1->second);
		it2 = it1;
		it2++;
		if (it2 == arbS.end()) continue; // it2 llego al final pero it1 no !
		sy=it2->first;			rpy.asignaS(sy, it2->second);

		while(strcmp(rpx.del,rpy.del)==0){		//checar aqui ---
			//if (it2->second != 0) {
				aux = tostr(rpx.al) + "-" + tostr(rpy.al);
				it3 = arbS.find(aux);
				if (it3 != arbS.end()){
					if(it3->second != 0) {
						it2->second = 0; // peso = 0 la convierte en
									transitiva
						borrados++;
					}
				} else {		// busca al reves
					aux = tostr(rpy.al) + "-" + tostr(rpx.al);
					it3 = arbS.find(aux);
					if (it3 != arbS.end()) {
						if(it3->second != 0) {
							it1->second = 0;// peso = 0 la convierte
										en transitiva

							borrados++;
						}
					}
				}
			it2++;
			if (it2 != arbS.end()){
				sy = it2->first;	rpy.asignaS(sy, it2->second);
			} else
				strcpy(rpy.del,MUNO);
		}		// fin del while
	}
	cout <<"borrados: "<<borrados<<endl;
}
```

```
void muestraArb(map<string,int> &arbS){
	map<string, int>::iterator it = arbS.begin();
	cout << "--> arbol de.."<< arbS.size()<<endl;

	for ( ; it != arbS.end(); it++) {
	   cout << it->first<<" ";
	   cout << it->second<<"\n";
	}
}
///////////////////////////////////////////
<< archivo .h>>
#include <iostream>
#include <map>
#include <set>
#include <iterator>
#include <sstream>
#include <string.h>
#include <stdio.h>
#include <time.h>
using namespace std;

template <typename T> string tostr(const T& t) {	//para concatenar simple
	ostringstream os;
	os << t;
	return os.str();
}
#define PARESNVO "paresNvo.dat"
#define PARESNVST "paresNvst.dat"
#define MUNO "-1"

FILE *inFileP;
FILE *outFileP;

struct RegPares{
	char del[10];
	char  al[10];
	int peso;

	inline string genLlave(){
		return tostr(del) + "-" + tostr(al);
	}
	inline void asignaC(char *tempo)   {
		sscanf(tempo, "%s %s %d", del, al, &peso);
	}
	inline void asignaS(string &tempo, int &p){
		sscanf(tempo.c_str(), "%9s-%9s", del, al);
		peso = p;
	}
} regPar;

/////////////////////////////////
```

8.5 Programa que convierte los fragmentos a pares en C/C++ FragAPares

Archivo *EstructPares12.h* de tipo cabecera de C/C++:

```
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
using namespace std;
// ---------------------------------
#define FRAGNVO "fragNvo.dat" // en desuso
#define PARESNVO "paresNvo.dat" // en desuso
#define PARAMETROS "params.txt"
#define CONSOLA "consola.txt"
#define INICIAL 500000 // tamaño del arreglo esperado en memoria dinámica
#define MAXREC 250
#define FRESPERADOS 2000
#define INVCOMP 1   //0=no genera Inv.Comp. 1=si genera Inv.Comp.<<----
#define CODFAGSZ 50 //longitud maxima de un codigo de identificacion
unsigned int MAX = INICIAL, MAXFR = FRESPERADOS;
// leidos del archivo de parametros
unsigned int FRAGSZ, MAXTRAS, REGPF, FACTORR;
char *FRAGORIGINAL1, *FRAGORIGINAL2;
// ----------------------------------
struct ArrInts {
	int arrI[4];
} *arrTrie=NULL;

// ----------------- estructura para armar archivo de fragmentos nuevo -----------
struct FragNvo{
	int serialFrag;			//    >numero
	long poscFrag;			// en donde esta en el arhcivo original (para que lo quiero?)
	char codigoFrag[CODFAGSZ]; // (codigo(+_inv)) // ajustado a 30, antes de 28, antes de 45
	char *fragFrag;				//se tiene que manejar dinámico porque no se conoce
FRAGSZ

	inline void limpiaFN(){
		serialFrag = poscFrag = 0;
		memset(codigoFrag, 0, CODFAGSZ);
		memset(fragFrag, 0, FRAGSZ);
	}
} fragNvo;
struct ArrFrags {
	int serialAF;
	//char fragAF[FRAGSZ+1];  //se tiene que manejar dinámico porque no se conoce FRAGSZ
	char *fragAF;
} *arrFrags = NULL;
// ---------------------estructura del arhcivo de pares nuevo --------------
struct ArchPares {
	int serial1;
	int serial2;
	int traslape;
} regArchPares;
// -----------------------------------------------------------
inline int letraAnum(char base){
	static const char letras[] = {9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
					9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
					9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
					9,9,9,9,9,0,9,1,9,9,9,2,9,9,9,9,9,9,9,9,
```

```
                                                  9,9,9,9,3};
        return letras[base];
}
static const char nums[] = {'A','C','G','T'};
inline char xor3 (char base) {
        return nums[(letraAnum(base) ^ 3)];

//////////////////////////////////
```

Archivo *FragAPares.cpp* de tipo punto de entrada (main) de C/C++:

```
#include "Estructuras12.h"
void leeParametros();
bool subeAArbol();
void leeYGrabaFragmento();
void armaFragmento(char linea1[], char linea2[], int &contGrabadosF);
void generaInverso(int &);
void muestraArbol();
void arregloCrece();
void arrFragsCrece();
void buscaTraslape();
int  cuentaChars(char ps[], char c);
bool cuentaPegados(char ps[], char *, char *, char *, char *);
int agotaRamas(long int j, const long int &ktras, long int &grabaDos, const long int &leiDos);
long posc;          // localización del registro en el archivo (offset)
unsigned int jmax=0, fmax = 0, kmax = 0;
FILE *ofC;          //consola con cifras de control

int main() {
        clock_t iniciaT = clock();
        ofC = fopen(CONSOLA, "w");
        printf("lee parametros...\n");
        leeParametros();
        //dimensiona arreglos ------------------------------
        fragNvo.fragFrag = (char*) malloc (FRAGSZ+1);//parte de un struct para guardar fragmento
        arrTrie = (struct ArrInts*)calloc(INICIAL,sizeof(struct ArrInts)); // reserva espacio, incia todo
en ceros
        arrFrags = (struct ArrFrags*)malloc(MAXFR * sizeof(struct ArrFrags)); // reserva el espacio
        // -------------------------------------------------
        leeYGrabaFragmento();    //<-- arma arbol trie, genera fragmentos Nuevo y valida
                                 duplicados -- ok
        // -------------------------------------------------
        //muestraArbol();
         float tMseg = ((float)(clock()- iniciaT)/CLOCKS_PER_SEC) * 1000;
        fprintf(ofC,"-->>tiempo de creacion Trie (mseg) %f \n", tMseg);

        buscaTraslape();         //<----- busca en arbol trie para traslapes y genera lista

        tMseg = ((float)(clock()- iniciaT)/CLOCKS_PER_SEC) * 1000;
        fprintf(ofC,"-->>tiempo total (mseg) %f \n", tMseg);
        fclose(ofC);
        printf("termina primera etapa...\n");
        return 0;
}
// va a leer el arhcivo de fragmentos generado y x c/u busca los traslapes
// lo graba en Pares Nuevo
FILE *outfileT;
void leeParametros() {
        FILE *inParams = fopen(PARAMETROS, "r");
```

```
		char linea[250];

		fgets(linea, 100, inParams);		//lee 1a. linea de parametros
		FRAGORIGINAL1 = (char*)malloc(strlen(linea));
		sscanf(linea, "%*s  %s", FRAGORIGINAL1);	//obtiene archivo de fragmentos tipo 1

		fgets(linea, 100, inParams);		//lee 2a. linea de parametros
		FRAGORIGINAL2 = (char*)malloc(strlen(linea));
		sscanf(linea, "%*s %s", FRAGORIGINAL2);	//obtiene archivo de fragmentos tipo 2

		fgets(linea, 100, inParams);		//lee 3a. linea de parametros (arhcivo de contigs)

		fgets(linea, 100, inParams);		//lee 4a. linea de parametros (registros x fragmento 2/4)
		sscanf(linea, "%*s %d", &REGPF);	//obtiene registros por fragmento

		fgets(linea, 100, inParams);		//lee 5a. linea.. valor de traslape
		sscanf(linea, "%*s %d", &MAXTRAS);		//obtiene valor de traslape

		fgets(linea, 100, inParams);		//lee 6a. linea .. valor de consenso

		fgets(linea, 100, inParams);		//lee 7a. linea .. longitud del fragmento
		sscanf(linea, "%*s %d", &FRAGSZ); //obtiene valor de FAGSZ

		fgets(linea, 100, inParams);		//lee 8a. linea .. factor de repeticiones
		sscanf(linea, "%*s %d", &FACTORR);		//obtiene valor de FACTORR

		fclose(inParams);
		fflush(stdout);
	}
	void buscaTraslape()
	{
		printf("\ninicia generacion de pares...\n");
		outfileT = fopen (PARESNVO, "w");
		long int i=1, ktras=0, j=0, leiDos=0, grabaDos=0, ii;// ij=0;

		while(leiDos < fmax)		//fmax dice cuantos fragmentos se generaron
		{
			for (ii=1; ii<FRAGSZ-MAXTRAS+1; ii++)
			{ // cada fragmento es procesado eliminando c/vez una letra
				j = 0; // esta en el nodo raiz
				for (i=ii;i<FRAGSZ; i++)
				{	// i determina la letra que sigue; inicia en la 2a.<<<!
					int inumLe = letraAnum(arrFrags[leiDos].fragAF[i]);
					 // convierte letra a numero 0..3

					if(arrTrie[j].arrI[inumLe] > 0)
					{			// coincide y continua --------------
						j = arrTrie[j].arrI[inumLe];
						ktras++; continue;		// va por otra letra
					}

					if (arrTrie[j].arrI[inumLe] == 0) // casilla vacía --------------------
						break;

				}	// fin del for de letras (fin del framento)
				if (i == FRAGSZ)
				agotaRamas(j, ktras, grabaDos, leiDos); //agotar la secuencia y grabar con
todos los caminos posibles hacia abajo

				ktras = 0;
```

```
                }
                leiDos++;
            }       // fin del while de leidos .......................
            fprintf(ofC,"-->>leidos: %ld \n", leiDos);
            fprintf(ofC,"-->>grabados en pares: %ld \n", grabaDos);
            fclose(outfileT);
        }
        int agotaRamas(long int j, const long int &ktras, long int &grabaDos, const long int &leiDos)
        {
            for (int i=0; i<4; i++)
            {       // recorre cada una de los 4 posibles caminos (ACGT)

                if(arrTrie[j].arrI[i] == 0) continue;  // ok--si es igual a cero, continua

                    if(arrTrie[j].arrI[i] >  0)
                    {     // continua al sig. nodo
                        agotaRamas(arrTrie[j].arrI[i], ktras, grabaDos, leiDos);
                        continue;
                    }
                    else
                    {       // ok-- <0: llego al nodo final de la rama; es un negativo
                    regArchPares.serial1 = arrFrags[leiDos].serialAF;
                    regArchPares.serial2 = -arrTrie[j].arrI[i];
                    regArchPares.traslape = ktras;
                    fprintf(outfileT, "%09d %09d %03d\n",
                        regArchPares.serial1,
regArchPares.serial2,regArchPares.traslape);
                        grabaDos++; i=4;  // en el nodo solo hay un negativo; no debe
seguir ahi
                        return 0;
                }
            } // fin del for de las 4 posibles salidas
            return 0;
        }
        /* lee el fragmento (de 2 en 2) los acomoda en un struct y genera un arbol trie
         * con el nodo validado, graba en archivo y genera el Inverso, lo agrega al arbol
         * y si todo bien, lo graba en el archivo
        */
        void leeYGrabaFragmento(){
            FILE *infile1 = fopen(FRAGORIGINAL1, "r");
            FILE *ofrags  = fopen(FRAGNVO, "w"); // archivo de txto delimitado con espacios, leerlo
igual
            int tipo = 1;
            int contLeidos = 1, contGrabadosF=0, rechazados=0, duplicados=0, rechazaPega = 0;
            char linea1[MAXREC], linea2[MAXREC];

            char As[FACTORR+1];     for (int i=0;i<FACTORR;i++) As[i]='A';//para detectar
repeticiones...
            char Cs[FACTORR+1];     for (int i=0;i<FACTORR;i++) Cs[i]='C';
            char Gs[FACTORR+1];     for (int i=0;i<FACTORR;i++) Gs[i]='G';
            char Ts[FACTORR+1];     for (int i=0;i<FACTORR;i++) Ts[i]='T';
            printf("inicia lectura de fragmentos tipo 1...\n");
        repiteX2:       //reincia el ciclo con el tipo 2
            posc = ftell(infile1);                  //en donde estoy?
            fgets(linea1, MAXREC, infile1);     //codigo del fragmento
            fgets(linea2, MAXREC, infile1);     // fragmento ... siempre esta en pares de registros
            //lee registro "directo" y si tiene errores se brinca el "IC"

            while (!feof(infile1)) {
                //valida que solo haya ACGT
```

```
                if ((cuentaChars(linea2,'A') + cuentaChars(linea2,'C') +
                        cuentaChars(linea2,'T') + cuentaChars(linea2,'G'))  != FRAGSZ)
                        {  rechazados++;   goto nocuenta;} // contiene caracteres diferentes a
ACGT; se anula la lectura

            // valida pegados / repeticiones // -------------------------------------------
        if (cuentaPegados(linea2, As, Cs, Gs, Ts))
           {   rechazaPega++;  goto nocuenta;}

                armaFragmento(linea1, linea2, contGrabadosF); // se ha generado un fragNvo

                if (subeAArbol())
                {           // con el fragNvo y se valida que no sea duplicado
                        fprintf(ofrags, "%07d %010ld %-45s %.*s\n",
                                fragNvo.serialFrag, fragNvo.poscFrag,fragNvo.codigoFrag,
                                FRAGSZ, fragNvo.fragFrag); //graba txt
                        contGrabadosF++; fmax++;
                }
                else {duplicados++; goto nocuenta; }

                if(INVCOMP == 0) goto nocuenta; // con esto se evita que se genere el inverso
complementario *****

                generaInverso(contGrabadosF);     // se genera el inverso complementario

                if (subeAArbol())
                {           // con el fragNvo y se valida que no sea duplicado
                        fprintf(ofrags, "%07d %010ld %-45s %.*s\n",
                                fragNvo.serialFrag, fragNvo.poscFrag, fragNvo.codigoFrag,
                                FRAGSZ, fragNvo.fragFrag); //graba txt
                        contGrabadosF++; fmax++;
                }
        else duplicados++;

            nocuenta:
                if (REGPF == 4)
                {                       //  se brinca 2
              fgets(linea1, MAXREC, infile1); // +
           fgets(linea1, MAXREC, infile1); // info. de calidad
            }
        posc = ftell(infile1);    //donde estoy?
                fgets(linea1, MAXREC, infile1); //codigo del fragmento
                if (!feof(infile1))
                {
                        fgets(linea2, MAXREC, infile1); // frgamneto
                        contLeidos++;
                }
            }           //fin del while (!feof(infile1)) <-----
            fflush(infile1);
            fclose(infile1);
            infile1 = NULL;
            if (tipo == 1)
            {
                tipo = 2;
                FILE *infile2 = fopen(FRAGORIGINAL2, "r");
                printf("inicia lectura de fragmentos tipo 2...\n");
                infile1 = infile2;
                goto repiteX2;
            }
```

```
        fclose(ofrags);
        fprintf(ofC,"-->>Leidos en total: %d\n",contLeidos);
        fprintf(ofC,"-->>Nodos generados: %d\n", kmax);
        fprintf(ofC,"-->>Rechazados X error en bases: %d \n", rechazados);
        fprintf(ofC,"-->>Rechazados X bases duplicadas: %d \n", rechazaPega);
        fprintf(ofC,"-->>Fragmentos Duplicados: %d \n", duplicados);
        fprintf(ofC,"-->>Grabados: %d \n",contGrabadosF-1);
        fprintf(ofC,"-->>En el trie: %d \n", jmax);
}
// -----------------------------------------------------------------------
inline bool subeAArbol(){
        bool noDuplica = false;
        int i=0,j=0,k=0, ib; // i:letraACGT-0123, j:nodoTrie, k:posc.fragmento
        short mxz = strlen(fragNvo.fragFrag);
        if (jmax+4 >= MAX) arregloCrece();

        for(;k<mxz;k++)
        {       // RECORRE CADA LETRA DEL FRAGMENTO con k
                i = letraAnum(fragNvo.fragFrag[k]);//cambia letra x numero --OK
                if (i>=4||i<0) continue;    // error en el caracter convertido ---
                ib = i;
                // tres condiciones: todo es nuevo, ya había datos y se duplica ---
                if (arrTrie[j].arrI[0] == 0 && arrTrie[j].arrI[1] == 0 &&
                        arrTrie[j].arrI[2] == 0 && arrTrie[j].arrI[3] == 0) {
                        arrTrie[j].arrI[i] = j+1;
                        noDuplica = true;
                        jmax++; j++; kmax++;                                   //todo nuevo
                } else
                        if (arrTrie[j].arrI[i] == 0) {
                                arrTrie[j].arrI[i] = jmax+1;
                                noDuplica = true;
                                j = ++jmax;                           // brinca a jmax
                        } else {
                                j = arrTrie[j].arrI[i];              //solo lo sigue
                        }
        }
        if (noDuplica) arrTrie[jmax].arrI[ib] = -fragNvo.serialFrag;

        return noDuplica;
}
void muestraArbol(){
        cout <<"-----------------------------------------------------------------\n";
        for (unsigned int ii=0;ii<=jmax+1;ii++){
      cout <<"\n-->"<<ii<<":";
      for (int jj=0; jj<4; jj++)
        cout << arrTrie[ii].arrI[jj]<<",";
        }
        cout <<endl;
}
//----------------------------------------------------------------
inline void armaFragmento(char linea1[], char linea2[], int &contGrabadosF) {

        fragNvo.limpiaFN();
        short l1=strlen(linea1), l2=strlen(linea2);
        strncpy(fragNvo.codigoFrag, linea1, l1-1);
        fragNvo.serialFrag = contGrabadosF;
        fragNvo.poscFrag = posc;        //este indica la posicion del registro en el archivo
        strncpy(fragNvo.fragFrag, linea2, l2-1);
        fragNvo.fragFrag[l2-1] = '\0';
        //------------------------------
```

```
        // carga arreglo de fragmentos (podría pasarse despues de subir a arbol)
        if (fmax + 1 >= MAXFR)     arrFragsCrece();
        arrFrags[fmax].serialAF = fragNvo.serialFrag;
        //---> aqui hay que definir el espacio en mem.dinamica para fragAF
        arrFrags[fmax].fragAF = (char*) malloc (FRAGSZ+1);
        strcpy(arrFrags[fmax].fragAF,fragNvo.fragFrag);
}
// -------------------------------
inline int cuentaChars(char ps[], char c)
{
        int conta =0;
        char *pt;
        do
        {
                pt=strchr(ps,c);
                conta++;
                ps = pt+1;
        } while (pt != NULL);
        conta--;                        //siempre sobra uno
        return conta;
}
inline bool cuentaPegados(char ps[], char *As, char *Cs, char *Gs, char *Ts)
{
   if (strstr(ps, As) != NULL) return true;
   if (strstr(ps, Cs) != NULL) return true;
   if (strstr(ps, Gs) != NULL) return true;
   if (strstr(ps, Ts) != NULL) return true;
   return false;
}
//-----------------------------------
inline void generaInverso(int &cgf){
        char inver [FRAGSZ];
        int i=0, j=FRAGSZ-1;

        for (;i<FRAGSZ; i++) inver[j-i] = xor3(fragNvo.fragFrag[i]); // obtiene Inv.Comp.
        inver[FRAGSZ] = '\0';
        strcpy(fragNvo.fragFrag, inver);
        strncat(fragNvo.codigoFrag,"_inv",4);
        fragNvo.serialFrag = cgf;
        // carga arreglo de fragmentos (podría pasarse despues de subir a arbol)
        if (fmax + 1 >= MAXFR)     arrFragsCrece();
        arrFrags[fmax].serialAF = fragNvo.serialFrag;
        // ---> aqui hay que definir malloc
        arrFrags[fmax].fragAF = (char*) malloc (FRAGSZ+1);
        strcpy(arrFrags[fmax].fragAF,fragNvo.fragFrag);
        //fmax++;
}
void arregloCrece() {
        arrTrie = (struct ArrInts*)realloc(arrTrie, (MAX*2)*sizeof(struct ArrInts));
        memset(arrTrie+MAX, 0, (MAX)*sizeof(struct ArrInts));
        MAX *= 2;
}
void arrFragsCrece() {
        MAXFR *= 2;
        arrFrags = (struct ArrFrags*)realloc(arrFrags, (MAXFR)*sizeof(struct ArrFrags));
        if (arrFrags != NULL) return;
        printf ("...Error al crecer arrFrags %d", MAXFR);
}
////////////////////////
```

8.6 Programa que convierte los pares a Contigs en C/C++ ListAdy

Archivo *EstructParesL.h* de tipo cabecera de C/C++:

```
#include <iostream>
#include <set>
#include <vector>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

#include <sys/stat.h>
#define _FILE_OFFSET_BITS 64   // ojo, para manejar arhcivo > 4 gb
using namespace std;
// -----------------------
void leeParametros();
void muestraLista(int);
void creaCabeceras();
void muestraListaPS(int);
//------------------------
int creaPilaDin0(int);
void PilaDin0Crece();
void adyacentes(int ultimo, int &qMax);
void recorreReversaMax(int);
void recorreReversa(int);
void creaPilaFrag();
void crecePilaFrag();
bool guardaUltimo(set<string> &, char[50]);
bool verificaIC(set<string> &, char[50]);
// ----------------------------------
#define PILAS 19
#define STFR 5000          //tamaño original de la pila
#define CODFSZ 50
#define PARESNVO "paresNvo.dat" // en desuso
#define FRAGNVO  "fragNvo.dat"   // en desuso
#define PARAMETROS "params.txt"
#define CONSOLA "consola.txt"
#define MAX(a,b)  ((a>b) ? a:b)

//unsigned long maxpa = 0;          //--conserva el limite del arreglo y se ajusta en c/crecimiento
off_t maxpa = 0;
unsigned int maxpila = 5; // determina el valor inicial de las pilas
unsigned int qSize=0;       // es el tamaño limite de la pila
unsigned int stFrSz = STFR; // es el tamaño inicial de una pila de fragmentos
unsigned int KONSENSO, FRAGSZ;
off_t PARSZ;                //longitud del registro del arhcivo de pares (se calcula)
char *ARCONTIGS;            //archivo de salida de contigs
FILE *infileP;              // para el archivo de pares
FILE *infileF;              // para el archivo de fragmentos
FILE *oufileC;              //para el archivo de contigs
FILE *ofC;                              //consola con cifras de control

int *qDin0 = NULL;          // pila para guardar los nodos con Din=0

// ---estructura del archivo de pares binario ---------
struct RegPares
{
        int serial1;
        int serial2;
```

```
	int traslape;

	inline void asigna(char *tempo) // 25 posiciones
	{
		char * pch;
		pch = strtok(tempo, " ");
		serial1 = atoi(pch);
		pch = strtok(NULL, " ");
		serial2 = atoi(pch);
		pch = strtok(NULL, " ");
		traslape = atoi(pch);
	}
} regPar;
// --- estructura de la lista de adyacencia con cabeceras y detalle -----------
struct RegListaAdy
{
   int laAl;			//en cabeceras es Din
   int laTraslape;  //en cabeceras es Dout
   int laIndice;      // dice a donde ir
} *arrLA = NULL;
// ---- estructura de predecesores y sucesores de u nodo ---------------------
struct RegPreSuc			// el nodo es el indice del arreglo, empieza en 0 -----
{
	int predecesor;
	int finalMax;
	int din;
	int dout;
	int pesoAcum;
	char finContig;		//cuando sea 1 es que aqui termina contig
} * listaPS = NULL;
// -------------------------------------------------------
struct RegFrags		// arma pila de fragmentos y se reutiliza como matriz de consensos
{					// el de abajo es el ultimo de la secuencia. el de arriba es
el primero (tr=0)
	int traslape;
	int serial;
	char codigoFrag[CODFSZ];
	//char fragmento[FRAGSZ+1];
	char *fragmento;			//se le asigna espacio al crear la pila o al crecer
} * pilaFrags = NULL;
// -------------------------------------------------------------------------
struct FragNvo{			//estructura del arhcivo de fragmentos, nuevo ------------
	int serialFrag;		//    >numero serial
	char codigoFrag[CODFSZ]; // (codigo(+_inv))
	//char fragFrag[FRAGSZ];
	char *fragFrag;

	inline void asigna(char *tempo) // 76 posiciones o mas
	{
		sscanf(tempo, "%d %*d %s %s", &serialFrag, codigoFrag, fragFrag);
	}

} regFragNvo;
//---------crece con factor de 1.5 el arreglo de la lista Ady.-----------------
void arrLACrece()
{
	maxpa *= 1.5;
	arrLA = (struct RegListaAdy*)realloc(arrLA, maxpa*sizeof(struct RegListaAdy));
	if (arrLA != NULL) return;
	printf("error al crecer arrLA!!");
```

```
}
// ----calcula la cantidad de pares; con esto se hace el arreglo de Lis.Ady.---
off_t registros()
{
   //char tempo[80];
   char tempo[130];
   //fgets(tempo, 80, infileP);
   fgets(tempo, 130, infileP);
   PARSZ = strlen(tempo);
// para linux:
   struct stat st;
   fstat(fileno(infileP), &st);
   off_t sze = st.st_size;
   //printf("___sze: %ld", sze);
   // ------------------------------
   // se coloca en el ultimo, lo lee y es lo que devuelve (del)
   fseek(infileP, sze-PARSZ, SEEK_SET);
            fgets(tempo, PARSZ, infileP);
            regPar.asigna(tempo);
            off_t del = regPar.serial1;
   fseek (infileP, 0, SEEK_SET); // se regresa -al origen del archivo
   //printf("\nel ultimo: %ld ", del);
   return del+1;
}
// agrega las cabeceras de la lista adyacente y deja un maxpa actualizado
void creaCabeceras()
{
   for (unsigned int i=0; i < maxpa; i++)
   {
      arrLA[i].laAl = 0;
      arrLA[i].laTraslape = 0;
      arrLA[i].laIndice = i;
   }
   arrLACrece();    // deja un crecimiento efectuado y maxpa se modifica
}
void creaListaPS(int cab)
{
            for (int i=0; i < cab; i++)
            {
                        listaPS[i].predecesor = -1;
                        listaPS[i].pesoAcum   = 0;
                        listaPS[i].din        = 0;
                        listaPS[i].dout       = 0;
                        listaPS[i].finContig  = '0';
                        listaPS[i].finalMax   = -1;
            }
}
// -----------------------------------------------------------------------
void muestraLista(int limite)
{
   printf("\n .....va a mostrar la lista de Ady.: \n");
   for (int i=0;i<limite; i++)
      printf("\ni:%d ->al:%d  tr:%d  idx:%d  ",i, arrLA[i].laAl, arrLA[i].laTraslape, arrLA[i].laIndice);
}
// -----------------------------------------------------------------------
void muestraListaPS(int limite)
{
   printf("\n .....va a mostrar la lista de Dins.: \n");
   for (int i=0;i<limite; i++)
      printf("\nnodo:%d   \t Di:%d  \t  Do:%d \tPred:%d  \t W:%d  \t FCt:%c  \t FMax:%d",
```

```
                    i,listaPS[i].din,listaPS[i].dout,listaPS[i].predecesor,listaPS[i].pesoAcum,
                     listaPS[i].finContig, listaPS[i].finalMax);
}
//--------------------------------------------------------------------
void adyacentes(int ultimo, int &qMax)
{
        int ady = arrLA[ultimo].laIndice;     //va por la cabecera y ahi el primer adyacente
        if (ady == ultimo)                                        // *** ady es el indice de arrLA
        {
                if (listaPS[ady].dout == 0) listaPS[ady].finContig = '1'; // es fin del contig
                return;                         //no tiene adyacencias y se sale
        }
        int adyC = arrLA[ady].laAl;                      // obtiene la cabecera del adyacente
        int wMax = 0;                                             // *** adyC es el indice de listaPS
        while(ady != ultimo)
        {
                listaPS[adyC].din -=1;                  // resta 1 al Din y calcula el pesoMax
                int tempo = arrLA[ady].laTraslape + listaPS[ultimo].pesoAcum;
                wMax = MAX(listaPS[adyC].pesoAcum, tempo);
                if (wMax > listaPS[adyC].pesoAcum) // si hubo cambios, toma nuevo predecesor
                {                                                 // y cambia el peso
                        listaPS[adyC].pesoAcum = wMax;
                        listaPS[adyC].predecesor = ultimo;
                }
                if (listaPS[adyC].din == 0)  // se agrega a la pila
                {
                        if (qMax+1 == qSize) PilaDin0Crece();
                        qMax++;
                        qDin0[qMax] = adyC;
                }
                ady = arrLA[ady].laIndice;
                adyC = arrLA[ady].laAl;
        }       //<<---- cierra while
}
// --------------------------------------------------------------------------
int creaPilaDin0(int cabeceras)      //crea la pila de nodos con Din = 0 y devuelve cuantos hubo
{
        qDin0 = (int*)malloc(cabeceras * sizeof(int));
        int q = 0;
        for (int i=0; i < cabeceras; i++)
        {
                if (listaPS[i].din==0 && listaPS[i].dout>0)
                {
                        qDin0[q] = i;  q++;
                }
        }
        return q-1;
}
// ------------------------------------------------------------------
void PilaDin0Crece()
{
        qSize *= 1.5;
        qDin0 = (int*)realloc(qDin0, (qSize)*sizeof(int));
}
// ----------------------------------------------------
inline void copiaStr(vector<char> v, char *c, int posc)
{
        int sl = strlen(c);
        int i=posc;
        for (;i<sl;i++)       v[i] = c[i];
```

```
}
/////////////////////////////////
```

Archivo *ListAdyDFSEnsam_2.cpp* de tipo punto de entrada (main) de C/C++:

```
#include "EstructParesL.h"
// ---> esta version muestra el contig cortado como formato FASTA
// pasa del archivo de pares-delim. a un arreglo en mem.dinÃ¡mica
// el archivo esta ordenado por "del-asc"+"traslape-desc"+"al-*"
//------------------------
//------------------------

int main ()
{
   clock_t iniciaT = clock();
   leeParametros();
   ofC = fopen(CONSOLA, "a");
         infileP  = fopen (PARESNVO, "r");        //prepara el arhcivo
         maxpa = registros();
         printf("\nregistros a procesar: %ld... \n", maxpa);
         // crea espacio para arrLA ....
   arrLA = (struct RegListaAdy*)malloc(maxpa * sizeof(struct RegListaAdy)); // reserva el espacio
   if (arrLA == NULL) {printf("error en creacion de espacio de listaLA!!"); return 8;}

         // crea espacio para listaPS ..........
         listaPS = (struct RegPreSuc*)malloc(maxpa * sizeof(struct RegPreSuc)); // reserva el espacio
         if (listaPS == NULL) {printf("error en creacion de espacio de listaPS!!"); return 8;}
         // ---------------------------------------------------------------------------
   off_t cabeceras = maxpa;  // cabeceras es el limite fijo y no va a cambiar
   off_t dondeVa = maxpa;    // a partir de esta posiciÃ²n empieza a agregar
   off_t guardados = 0;
         // ----------------------------------------------------------------
   creaCabeceras();      // crea las cabeceras de todas los nodos de la lista
         creaListaPS(cabeceras);                    // crea los nodos con pred=-1. peso=0,...
         // ----------------------------------------------------------------
   // lee el primero
         char *tempo= new char[PARSZ+1];
         fgets(tempo, PARSZ+1, infileP);      regPar.asigna(tempo);
         long leidos = 0;
         // ciclo de lecturas
         printf("inicia ensamble de contigs ...\n");
   while(!feof(infileP))  //x c/leido, busca su cabecera y se va al ultimo asignado; actualiza
   {
      leidos++;
      //if (leidos % 10000 == 0) printf (" Leidos..: %d", leidos);
      // crea nuevo elemento para ListaAdyacente
      arrLA[dondeVa].laAl        = regPar.serial2;
      arrLA[dondeVa].laTraslape   = regPar.traslape;
      arrLA[dondeVa].laIndice     = arrLA[regPar.serial1].laIndice;
      //actualiza cabecera de lista adyacente
      arrLA[regPar.serial1].laIndice = dondeVa;
      listaPS[regPar.serial2].din++;   // agreag uno al grado de entrada
      listaPS[regPar.serial1].dout++;// agreag uno al grado de salida
      dondeVa++;   guardados++;
      // termina proceso de guardado de listaAdyacente -------------------------
         // lee el siguiente
      fgets(tempo, PARSZ+1, infileP);
      if (!feof(infileP))
      {
```

```
        regPar.asigna(tempo);
          if (dondeVa >= maxpa) arrLACrece();       // si no hay espacio, crece
      }
   }
          delete [] tempo;
          fflush(stdout);
   //muestraLista(cabeceras);
   //muestraLista(guardados+cabeceras);
          //muestraListaPS(cabeceras);
          // ------------------------------------------->>
          // ------------------------------------------->>
          // ---->>inicia recorridos en profundidad ---->>
          int qMax = creaPilaDin0(cabeceras);            //crea el espacio y le inserta los nodos con
Din=0; devuelve cuantos hubo
          qSize = qMax;                   // es el limite de capacidad de la pila
          int ultimo = qDin0[qMax]; // saca el ultimo (primera vez)
          while(qMax >= 0) // toma los elementos a partir del ultimo insertado (es una pila)
          {
                    qMax--;
                    adyacentes(ultimo, qMax);
                    ultimo = qDin0[qMax];      // saca el ultimo
          }
          //muestraListaPS(cabeceras);
          free(arrLA);
          arrLA = NULL;
          //-----------
          recorreReversaMax(cabeceras);     // calcula los MAXs con los marcados=1 (en reversa
desde listaPS)
          //muestraListaPS(cabeceras);
          recorreReversa(cabeceras);                        // genera los contigs basados en el pesoMAx
desde el nodo de inicio
          // fin recorridos en profundidad ------------<<
          //fclose(infileP);
          float tMseg = ((float)(clock()- iniciaT)/CLOCKS_PER_SEC) * 1000;
          fprintf(ofC,"-->>tiempo (mseg) %f \n", tMseg);
          fprintf(ofC,"-->>leidos de arch.B.: %ld \n", leidos);
          fprintf(ofC,"-->>guardados en L.A.: %ld \n", guardados);
          printf("termina etapa 2 ...\n");
          fflush(stdout);
return 0;
}
// -----------------------------------------------
void recorreReversaMax(int cab) // recorre en reversa los marcados con 1
{                                                                                        // y actualiza los iniciales max.
          int fcntgs = 0;
          int cntgsGen = 0;
          for (int i=0; i < cab ; i++)
          {
                    if (listaPS[i].finContig == '1')
                    {
                              int nodoF = i;
                              fcntgs++;
                              int peso=listaPS[i].pesoAcum;
                              while (listaPS[nodoF].predecesor != -1)
                              {
                                                  nodoF = listaPS[nodoF].predecesor;
                              }
                              if (listaPS[nodoF].pesoAcum < peso)
                              {
                                        listaPS[nodoF].pesoAcum = peso;
```

```
                                listaPS[nodoF].finalMax = i;
                                cntgsGen++;
                        }
                }
        }
        fprintf(ofC,"-->>contigs terminales: %d \n", fcntgs);
        fprintf(ofC,"-->>contigs verficados: %d \n", cntgsGen);
        fflush(stdout);
}
//-----------------------------------------------
void recorreReversa(int cab) // recorre los que tiene peso!=0 y finalMax!= -1
{                                                       // y se va a finalMax para recorrer en
reversa
        infileF = fopen(FRAGNVO, "r");      //prepara el arhcivo de fragmentos
        oufileC = fopen(ARCONTIGS, "w"); // prepara el archivo de contigs
        int kelimxK = 0, kelimxD = 0;
        int kcntgs = 1;
        off_t nodoF;
        int pesoActual  =0;
        int fondo = 0;
        int trCalc = 0;
        int trAnte = 0;
        set <string> indice;

        char tempF[200];
        fgets(tempF, 200, infileF);             //lee primer registro
#ifdef _WIN32
        const off_t LSEG = strlen(tempF)+1; // longitud efectiva del registro de fragmentos
#else
        const off_t LSEG = strlen(tempF); // longitud efectiva del registro de fragmentos
#endif
   //printf("\n--long del reg. %ld\n", LSEG);fflush(stdout);

        int iniTR=0, finTR=0, kTR=0;
        char *ensamble = NULL; // string para ensamblar los contigs
        char *ens = NULL;
        creaPilaFrag();                 // crear una pila de fragmentos pilaFrags[] .. regFrags
        regFragNvo.fragFrag = (char*)malloc(FRAGSZ);         //reserva espacio una sola vez (+1?)

        for (unsigned int i=0; i < cab ; i++)
        {
                if (listaPS[i].pesoAcum > 0 && listaPS[i].finalMax > -1)
                {
                        fondo = 0;
                        nodoF = listaPS[i].finalMax;
                        pesoActual = listaPS[i].pesoAcum;
                        // meterlo a la pila en el fondo --> va por el fragmento al arhcivo de
acceso directo
                        while (listaPS[nodoF].predecesor != -1)       // es hasta agotar el camino
del dfs
                        {
                                fseek(infileF, LSEG*nodoF, SEEK_SET);         //<--brinca a la
posicion de nodoF
                                fgets(tempF, LSEG, infileF); regFragNvo.asigna(tempF);//<--lee en
la posicion -del-

                                strcpy(pilaFrags[fondo].fragmento, regFragNvo.fragFrag); // <<-
aqui lo mete a la pila
                                pilaFrags[fondo].traslape = pesoActual;
                                pilaFrags[fondo].serial   = regFragNvo.serialFrag;
```

```
                strcpy(pilaFrags[fondo].codigoFrag,regFragNvo.codigoFrag);
                //-------------
                if (fondo+2 >= stFrSz) crecePilaFrag();
                fondo++;
                nodoF = listaPS[nodoF].predecesor;
                pesoActual = listaPS[nodoF].pesoAcum;
            }
            // procesa el utlimo registro:...........
            fseek(infileF, LSEG*nodoF, SEEK_SET);// <-- brinca a la posicion del nodo
            fgets(tempF, LSEG, infileF);
            regFragNvo.asigna(tempF);//<--lee en la posicion -del-
            strcpy(pilaFrags[fondo].fragmento, regFragNvo.fragFrag);
            pilaFrags[fondo].traslape = 0;        // el que queda hasta arriba no traslapa
            pilaFrags[fondo].serial   = regFragNvo.serialFrag;
            strcpy(pilaFrags[fondo].codigoFrag,regFragNvo.codigoFrag);

            //<<---con fondo se verifica que sea al menos uno mayor a dos veces el
(K)consenso--->>
            if (fondo <= KONSENSO*2)// se elimina si no cumple !!
            {
             kelimxK++; continue;
             }
             // la pila esta bien y completa, aunque de cabeza. al mostrarl al reves,
sale como debe ser.!!
             // ----------------------------------- ahora desde arriba de la pila ...y va
ensamblando
             //antes de hacer operaciones va a verificar que no se duplique como IC
             if(verificaIC(indice, pilaFrags[fondo-1].codigoFrag))   // se elimina si no
cumple
             {
                kelimxD++;  continue;
            }
             trCalc = 0;
             trAnte = -FRAGSZ;
             iniTR = 0; kTR = 0; finTR = 0;

             int lensam = fondo * FRAGSZ * 2;

             ens = (char *) malloc(lensam);
             if (ens     == NULL) {
                printf("error al crear espacio de ens !!!");fflush(stdout);  return; }

    ensamble = (char *) malloc(lensam);
             if (ensamble == NULL) {
                printf("error al crear espacio de ensambles
            !!");fflush(stdout);return; }

             strcpy(ensamble,pilaFrags[fondo].fragmento);        // <<--coloca el
primero y en el while coloca el mismo en 0 la 1a. vez

             while(fondo != 0)   // sacar de la pila mientras fondo != 0 y ensamblar
             {
                trCalc = FRAGSZ - (pilaFrags[fondo].traslape - trAnte);
                // impresion de trabajo vvvv
                //fprintf(oufileC, "\nTr: %d Serial: %d Codigo: %s\n Fr: %s
Calc:%d",
     //   pilaFrags[fondo].traslape, pilaFrags[fondo].serial,
     //   pilaFrags[fondo].codigoFrag, pilaFrags[fondo].fragmento, trCalc);
```

```
                    if (strlen(ensamble) + trCalc*20 >= lensam) //asegura el espacio
de memoria
                    {
                         lensam *= 2.0;
                         ensamble = (char*)realloc(ensamble, lensam);
                         if (ensamble == NULL) {
                              printf("error al crecer espacio de ensambles
                         !!!");              return;   }
                     }

                    strncat(ensamble, pilaFrags[fondo].fragmento+FRAGSZ-trCalc,
trCalc);
                    if (kTR++ <  KONSENSO) iniTR += trCalc; // es lo que le va a quitar
al inicio del contig
                    if (fondo <  KONSENSO-1) finTR += trCalc; // es lo que le va a
quitar al final del contig
                    trAnte = pilaFrags[fondo].traslape;
                    fondo--;
               }    // fin del while; falta el ultimo ..

               trCalc = FRAGSZ - (pilaFrags[fondo].traslape - trAnte);

               // impresion de trabajo vvvv
     //fprintf(oufileC,"\nTr: %d Serial: %d Codigo: %s\n Fr: %s Calc:%d",
     //   pilaFrags[fondo].traslape, pilaFrags[fondo].serial,
     //   pilaFrags[fondo].codigoFrag, pilaFrags[fondo].fragmento, trCalc);

               strncat(ensamble, pilaFrags[fondo].fragmento+FRAGSZ-trCalc, trCalc);
               finTR += trCalc;
               int ls = strlen(ensamble)-iniTR-finTR;

     //servirÃ¡ para validar que no se repite un contig en IC, pero utiliza el penutlimo vs el
segundo
     // para garantizar un buen traslape
     guardaUltimo(indice, pilaFrags[fondo+1].codigoFrag);


               ens[0] = '\0';
               strncat(ens, (ensamble+iniTR), ls);
               int fr60 = 0;
               char *p = ens;
               // imprime contig sin cortes
               //fprintf(oufileC,"\n>contig %d   long. %d  \n%s", kcntgs, ls, ens);
               //------------
               //  Fragmenta el contig de 60 en 60
               fprintf(oufileC,">contig %d long. %d \n", kcntgs, ls);
               do
               {
      fprintf(oufileC,"%.60s\n", p);
      fr60 += 60;
      p = ens + fr60;
               } while (fr60 < ls+1);
               // hasta aqui la fragmentación del contig de 60 en 60
               free(ensamble);
               free(ens);
               kcntgs++;
          }
      }
      fprintf(ofC,"-->> contigs efectivos: %d \n", kcntgs-1);fflush(stdout);
      fprintf(ofC,"-->> eliminados por no cubrir el consenso: %d\n", kelimxK);fflush(stdout);
```

```
            fprintf(ofC,"-->> eliminados por duplicados en I.C.  : %d\n", kelimxD);fflush(stdout);
            fclose(ofC);
            fclose(infileF);
            fclose(oufileC);
}
bool verificaIC(set<string> &indice, char codigo[50])
{
   int lc = strlen(codigo);
   static char *pcd;
   int poscD;
   pcd = strchr(codigo,'/');   // la diagonal esta en esta direccion
   poscD = pcd - codigo;       //saca la posicion por diferencia de apuntadores
   string termina, original=codigo;
   termina.append(codigo, poscD, lc-poscD);
   original.erase(poscD, lc-poscD);
   if (termina == "/1") original.append("/1_inv");
   if (termina == "/2") original.append("/2_inv");
   if (termina == "/1_inv") original.append("/1");
   if (termina == "/2_inv") original.append("/2");
   auto se = indice.find(original);
   if (se != indice.end())
      return true;    // si esta
   else
      return false;    // no esta
}
bool guardaUltimo(set<string> &indice, char codigo[])
{
   int lc = strlen(codigo);
   string codSt = codigo;
   codSt.shrink_to_fit();
   indice.insert(codSt);
   return true;
}
//----------------------------------------------------------------------
void leeParametros()
{
            FILE *inParams = fopen(PARAMETROS, "r");
            char linea[250];

            fgets(linea, 100, inParams);            //lee 1a. linea de parametros Fragmentos 1 - DC
            fgets(linea, 100, inParams);            //lee 2a. linea de parametros Fragmentos 2 - DC

            fgets(linea, 100, inParams);            //lee 3a. linea de parametros (archivo de contigs)
            ARCONTIGS = (char*)malloc(strlen(linea));
            sscanf(linea, "%*s  %s", ARCONTIGS);

            fgets(linea, 100, inParams);            //lee 4a. linea de parametros (regs.x fragmento 2/4)- DC

            fgets(linea, 100, inParams);            //lee 5a. linea.. valor de traslape - DC

            fgets(linea, 100, inParams);            //lee 6a. linea .. valor de consenso
            sscanf(linea, "%*s %d", &KONSENSO);         //obtiene valor de KONSENSO

            fgets(linea, 100, inParams);            //lee 7a. linea .. longitud del fragmento
            sscanf(linea, "%*s %d", &FRAGSZ); //obtiene valor de FRAGSZ

            fgets(linea, 100, inParams);            //lee 8a. linea .. factor de repeticiones - DC

            fclose(inParams);
            fflush(stdout);
```

```
}
// -------------------------------------------------------------------
void creaPilaFrag()		//crea la pila de fragmentos con STFR elementos (50)
{
	stFrSz = STFR;
	pilaFrags = (RegFrags*)malloc(STFR * sizeof(RegFrags));
	//asigna memoria dinamica al fragmento:
	int i=0;
	for (;i<stFrSz;i++) pilaFrags[i].fragmento = (char*)malloc(FRAGSZ+1);
}
void crecePilaFrag()		//crece la pila de fragmentos
{
	int aux = stFrSz;
	stFrSz = stFrSz * 2.0;
	pilaFrags = (RegFrags*)realloc(pilaFrags, (stFrSz)*sizeof(RegFrags));
	//asigna memoria dinamica al fragmento
	int i=aux;
	for (;i<stFrSz;i++) pilaFrags[i].fragmento = (char*)malloc(FRAGSZ+1);
}
////////////////////////////////////////////
```

8.7 Programa que controla la interface gráfica con parámetros en Java MaxP

Archivo *Parametros.java* de tipo cabecera de Java, orientado a eventos:

```
/*
 * Captura de parámetros para el ensamble de
 * fragmentos de ADN.
 * Progama Elaborado por: Emilio Quiroz
 * Mayo/2017
 */
package DNAAssembler;

import java.io.IOException;
import java.io.*;
import java.util.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.regex.Pattern;
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;

/**
 *
 * @author emilio quiroz
 * @date 15/may/2017
 */
public class Parametros extends javax.swing.JFrame {

    /**
     * Creates new form Parametros
     */
    public Parametros() {
        initComponents();
        jLabel11.setVisible(false);
    }
    // variables globales
    String archF1="", archF2="", archCntgs="",
            ruta1="", ruta2="";
    int regPF, valTr, valCons, fragSz, factorR;
    boolean txt1Err=false, txt2Err=false;
    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initComponents() {

        jLabel4 = new javax.swing.JLabel();
        buttonGroup1 = new javax.swing.ButtonGroup();
        jPanel1 = new javax.swing.JPanel();
        jLabel1 = new javax.swing.JLabel();
        jLabel2 = new javax.swing.JLabel();
        jLabel3 = new javax.swing.JLabel();
        txtArchT1 = new javax.swing.JTextField();
        txtArchT2 = new javax.swing.JTextField();
        jLabel5 = new javax.swing.JLabel();
        rbt2Regs = new javax.swing.JRadioButton();
        rbt4Regs = new javax.swing.JRadioButton();
```

```
jLabel6 = new javax.swing.JLabel();
txtValTras = new javax.swing.JTextField();
jLabel7 = new javax.swing.JLabel();
txtValCons = new javax.swing.JTextField();
jLabel8 = new javax.swing.JLabel();
txtArchCntg = new javax.swing.JTextField();
btnProcesar = new javax.swing.JButton();
btnCancelar = new javax.swing.JButton();
jLabel9 = new javax.swing.JLabel();
jSeparator1 = new javax.swing.JSeparator();
btnBuscar1 = new javax.swing.JButton();
btnBuscar2 = new javax.swing.JButton();
jLabel11 = new javax.swing.JLabel();
btnReset = new javax.swing.JButton();
jLabel10 = new javax.swing.JLabel();
txtValFactorR = new javax.swing.JTextField();

jLabel4.setText("jLabel4");

javax.swing.GroupLayout jPanel1Layout = new javax.swing.GroupLayout(jPanel1);
jPanel1.setLayout(jPanel1Layout);
jPanel1Layout.setHorizontalGroup(
   jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
   .addGap(0, 100, Short.MAX_VALUE)
);
jPanel1Layout.setVerticalGroup(
   jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
   .addGap(0, 100, Short.MAX_VALUE)
);

setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
setTitle("MaxP - DNA fragment assembler");
setBounds(new java.awt.Rectangle(0, 0, 600, 800));
setCursor(new java.awt.Cursor(java.awt.Cursor.DEFAULT_CURSOR));
setResizable(false);

jLabel1.setFont(new java.awt.Font("Tahoma", 1, 14)); // NOI18N
jLabel1.setText("Parameters capture for the DNA contigs assembly process");

jLabel2.setText("Type 1 records file:");

jLabel3.setText("Type 2 records file:");

txtArchT1.setEditable(false);
txtArchT1.setText(".fasta / .dat / .seq");
txtArchT1.setToolTipText("archivo de fragmentos tipo 1");
txtArchT1.setCursor(new java.awt.Cursor(java.awt.Cursor.TEXT_CURSOR));

txtArchT2.setEditable(false);
txtArchT2.setText(".fasta / .dat / .seq");
txtArchT2.setFocusable(false);

jLabel5.setText("Fragment per records:");

buttonGroup1.add(rbt2Regs);
rbt2Regs.setText("2 records: code and bases");
rbt2Regs.setToolTipText("registros simples (codigo y bases)");
rbt2Regs.setEnabled(false);
rbt2Regs.setFocusable(false);
rbt2Regs.addActionListener(new java.awt.event.ActionListener() {
```

```
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        rbt2RegsActionPerformed(evt);
    }
});

buttonGroup1.add(rbt4Regs);
rbt4Regs.setText("4 records: code, bases, id, quality");
rbt4Regs.setToolTipText("incluye registro de calidad");
rbt4Regs.setEnabled(false);
rbt4Regs.setFocusable(false);
rbt4Regs.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        rbt4RegsActionPerformed(evt);
    }
});

jLabel6.setText("Overlap value:");

txtValTras.setText("30");
txtValTras.setToolTipText("valor default o indique otra cantidad");
txtValTras.setEnabled(false);
txtValTras.addFocusListener(new java.awt.event.FocusAdapter() {
    public void focusGained(java.awt.event.FocusEvent evt) {
        txtValTrasFocusGained(evt);
    }
    public void focusLost(java.awt.event.FocusEvent evt) {
        txtValTrasFocusLost(evt);
    }
});

jLabel7.setText("Consensus value:");

txtValCons.setText("4");
txtValCons.setToolTipText("valor default o indique otra cantidad");
txtValCons.setEnabled(false);
txtValCons.addFocusListener(new java.awt.event.FocusAdapter() {
    public void focusGained(java.awt.event.FocusEvent evt) {
        txtValConsFocusGained(evt);
    }
    public void focusLost(java.awt.event.FocusEvent evt) {
        txtValConsFocusLost(evt);
    }
});

jLabel8.setText("Contigs file:");

txtArchCntg.setText(".fasta");
txtArchCntg.setToolTipText("se recomienda terminacion fasta");
txtArchCntg.setEnabled(false);
txtArchCntg.addFocusListener(new java.awt.event.FocusAdapter() {
    public void focusGained(java.awt.event.FocusEvent evt) {
        txtArchCntgFocusGained(evt);
    }
    public void focusLost(java.awt.event.FocusEvent evt) {
        txtArchCntgFocusLost(evt);
    }
});

btnProcesar.setText("Process");
btnProcesar.setEnabled(false);
```

```
        btnProcesar.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnProcesarActionPerformed(evt);
            }
        });

        btnCancelar.setText("Cancel");
        btnCancelar.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnCancelarActionPerformed(evt);
            }
        });

        jLabel9.setFont(new java.awt.Font("Tahoma", 1, 11)); // NOI18N
        jLabel9.setForeground(new java.awt.Color(255, 0, 0));

        btnBuscar1.setText("search");
        btnBuscar1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnBuscar1ActionPerformed(evt);
            }
        });

        btnBuscar2.setText("search");
        btnBuscar2.setEnabled(false);
        btnBuscar2.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnBuscar2ActionPerformed(evt);
            }
        });

        jLabel11.setIcon(new
    javax.swing.ImageIcon(getClass().getResource("/DNAAssembler/arena.gif"))); // NOI18N

        btnReset.setText("Restore");
        btnReset.setActionCommand("btnReset");
        btnReset.setEnabled(false);
        btnReset.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnResetActionPerformed(evt);
            }
        });

        jLabel10.setText("Self-overlapped factor:");

        txtValFactorR.setText("29");
        txtValFactorR.setToolTipText("maximo de repeticiones");
        txtValFactorR.setEnabled(false);
        txtValFactorR.addFocusListener(new java.awt.event.FocusAdapter() {
            public void focusGained(java.awt.event.FocusEvent evt) {
                txtValFactorRFocusGained(evt);
            }
            public void focusLost(java.awt.event.FocusEvent evt) {
                txtValFactorRFocusLost(evt);
            }
        });

        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
```

```
        layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup()
          .addContainerGap()
          .addComponent(btnProcesar, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
          .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
          .addComponent(btnReset, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
          .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
          .addComponent(btnCancelar, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
          .addGap(163, 163, 163))
        .addGroup(layout.createSequentialGroup()
          .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
              .addContainerGap()
              .addComponent(jSeparator1, javax.swing.GroupLayout.PREFERRED_SIZE, 336,
javax.swing.GroupLayout.PREFERRED_SIZE))
            .addGroup(layout.createSequentialGroup()
              .addGap(10, 10, 10)
              .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addComponent(jLabel2)
                .addComponent(jLabel3)
                .addComponent(jLabel8))
              .addGap(4, 4, 4)
              .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                .addGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addComponent(txtArchT2, javax.swing.GroupLayout.PREFERRED_SIZE, 110,
javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addComponent(txtArchT1, javax.swing.GroupLayout.PREFERRED_SIZE, 112,
javax.swing.GroupLayout.PREFERRED_SIZE))

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addGroup(layout.createSequentialGroup()
                      .addGap(10, 10, 10)
                      .addComponent(btnBuscar1))
                    .addGroup(layout.createSequentialGroup()
                      .addGap(8, 8, 8)
                      .addComponent(btnBuscar2))))
                .addGroup(layout.createSequentialGroup()
                  .addComponent(txtArchCntg, javax.swing.GroupLayout.PREFERRED_SIZE, 184,
javax.swing.GroupLayout.PREFERRED_SIZE)
                  .addGap(2, 2, 2))))
            .addGroup(layout.createSequentialGroup()
              .addContainerGap()
              .addComponent(jLabel1))
            .addGroup(layout.createSequentialGroup()
              .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                  .addGap(10, 10, 10)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addComponent(jLabel10)
                    .addComponent(jLabel7)
                    .addComponent(jLabel5))
                  .addGap(18, 18, 18)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
```

```
                        .addComponent(txtValTras, javax.swing.GroupLayout.PREFERRED_SIZE, 117,
javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(txtValFactorR, javax.swing.GroupLayout.PREFERRED_SIZE, 116,
javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(txtValCons, javax.swing.GroupLayout.PREFERRED_SIZE, 117,
javax.swing.GroupLayout.PREFERRED_SIZE)))
                    .addGroup(layout.createSequentialGroup()
                        .addGap(136, 136, 136)
                        .addComponent(rbt2Regs))
                    .addGroup(layout.createSequentialGroup()
                        .addGap(136, 136, 136)
                        .addComponent(rbt4Regs))
                    .addGroup(layout.createSequentialGroup()
                        .addGap(10, 10, 10)
                        .addComponent(jLabel6)))
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                .addComponent(jLabel11)))
            .addContainerGap(37, Short.MAX_VALUE))
    );
    layout.setVerticalGroup(
        layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup()
            .addGap(11, 11, 11)
            .addComponent(jLabel1)
            .addGap(23, 23, 23)
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                    .addGap(4, 4, 4)
                    .addComponent(jLabel2))
                .addGroup(layout.createSequentialGroup()
                    .addGap(1, 1, 1)
                    .addComponent(txtArchT1, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addComponent(btnBuscar1))
            .addGap(6, 6, 6)
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                    .addGap(4, 4, 4)
                    .addComponent(jLabel3))
                .addGroup(layout.createSequentialGroup()
                    .addGap(1, 1, 1)
                    .addComponent(txtArchT2, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addComponent(btnBuscar2))
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addComponent(jLabel8)
                .addComponent(txtArchCntg, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
            .addGap(8, 8, 8)
            .addComponent(jSeparator1, javax.swing.GroupLayout.PREFERRED_SIZE, 6,
javax.swing.GroupLayout.PREFERRED_SIZE)
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                        .addComponent(rbt2Regs)
                        .addComponent(jLabel5))
                    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                    .addComponent(rbt4Regs)
```

```
                .addGap(8, 8, 8)
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                  .addComponent(jLabel6)
                  .addComponent(txtValTras, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(3, 3, 3)
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                  .addComponent(jLabel10)
                  .addComponent(txtValFactorR, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(2, 2, 2)
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING,
false)
                  .addComponent(txtValCons)
                  .addComponent(jLabel7, javax.swing.GroupLayout.PREFERRED_SIZE, 20,
javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(31, 31, 31)
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.CENTER)
                  .addComponent(btnProcesar, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                  .addComponent(btnReset, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                  .addComponent(btnCancelar, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                .addGap(49, 49, 49))
              .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup()
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                .addComponent(jLabel11)
                .addGap(118, 118, 118))))
    );

    pack();
  }// </editor-fold>
  private void desactiva(){
    btnBuscar2.setEnabled(false);
    btnProcesar.setEnabled(false);
    btnReset.setEnabled(false);
    rbt2Regs.setEnabled(false);
    rbt4Regs.setEnabled(false);
    txtValCons.setEnabled(false);
    txtValTras.setEnabled(false);
    txtArchCntg.setEnabled(false);
    txtArchT1.setText("");
    txtArchT2.setText("");
    txtArchCntg.setText("");
    rbt2Regs.setSelected(false);
    rbt4Regs.setSelected(false);
    txtValCons.setText("4");
    txtValTras.setText("30");
    txtValFactorR.setText("29");
  }
  private void btnCancelarActionPerformed(java.awt.event.ActionEvent evt) {
    JOptionPane.showMessageDialog(null, "process canceled !");
    System.exit(1);
  }

  private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
    jLabel11.setVisible(true);
```

```
jLabel11.setEnabled(true);
try {
   FileWriter fw=new FileWriter("params.txt");
   fw.write("archF1: " + ruta1+"\n"); //registros 1
   fw.write("archF2: " + ruta2+"\n"); //registros 2
   fw.write("archCntgs: " + archCntgs+"\n");   //resultante de contigs
   fw.write("regPF: " + regPF+"\n");   //registros x fragmento (2/4)
   fw.write("valTr: " + valTr+"\n");   //valor del traslape
   fw.write("valCons: " + valCons+"\n");   //valor del consenso
   fw.write("FragSz: " + fragSz + "\n");   // long. del fragmento
   fw.write("FactorR: " + factorR + "\n"); // factor de repeticiones
   fw.close();
} catch (IOException ioe) {
   System.out.println("error trying to write "+ioe);
}
// <<< aqui van las llamadas a los programas de C/C++>>>>
String sop = System.getProperty("os.name");
try {
   List<String> arg1;
   List<String> arg2;
   if (sop.contains("Win")){
      arg1 = new ArrayList<String>(4);
      arg2 = new ArrayList<String>(4);
      arg1.add("cmd");
      arg1.add("/c");
      arg1.add("start/w");
      arg1.add("FragAPares_12.exe");
      // -------------------------
      arg2.add("cmd");
      arg2.add("/c");
      arg2.add("start/w");
      arg2.add("ListAdyDFSEnsam_2.exe");
   } else { // Linux o Mac
      arg1 = new ArrayList<String>(3);
      arg2 = new ArrayList<String>(3);
      arg1.add("bash");
      arg1.add("-c");
      arg1.add("./FragAPares_12.eje");
      arg2.add("bash");
      arg2.add("-c");
      arg2.add("./ListAdyDFSEnsam_2.eje");
   }
   ProcessBuilder p1,p2;
   p1 = new ProcessBuilder(arg1);
   p2 = new ProcessBuilder(arg2);
   Process pb = p1.start();
   BufferedReader input = new BufferedReader(
         new InputStreamReader(pb.getInputStream()));
   int k=0;
   String linea;
   while ((linea = input.readLine()) != null) {
      System.out.println(linea + " " + ++k);
   }
   // --------------------------------
   pb = p2.start();
   input = new BufferedReader(
         new InputStreamReader(pb.getInputStream()));
   k=0;
   while ((linea = input.readLine()) != null) {
      System.out.println(linea + " " + ++k);
```

```
        }
        System.out.println(k + " waiting 2... " + pb.waitFor());
    } catch (IOException | InterruptedException ex) {
        Logger.getLogger(Parametros.class.getName()).log(Level.SEVERE, null, ex);
    }
    JOptionPane.showMessageDialog(null, "process terminated !");
    System.exit(0);
}
private boolean checa2En2(String arF){
    if (arF.isEmpty()) return false;
    try {
            String leido;
            char idCodigo;
            int contador = 0;
            BufferedReader b1 = new BufferedReader(
                        new FileReader(arF));
            leido = b1.readLine(); // 1a. lectura codigo
            idCodigo = leido.charAt(0); // identifica codigo
            leido = b1.readLine();  // 2a. lectura bases
            boolean contiene;
            while (contador < 4){
                contiene = Pattern.matches("(A|C|G|T|N)+", leido);
                if (contiene) contador++;
                fragSz = leido.length();
                leido = b1.readLine();  //lee codigo de nuevo
                if (leido.charAt(0) == idCodigo){
                    idCodigo = leido.charAt(0);
                    contador++;
                }
                leido = b1.readLine();  // lee bases
            }// fin de while
            b1.close();
            if (contador == 4) return true; //todo bien
        } catch (IOException ioe) {
            System.out.println("problems when checking records:"+ioe);
            return false;
        }// fin de ioexception
    return false;
}
private boolean checa4En4(String arF){
    if (arF.isEmpty()) return false;
    try {
            String leido;
            char idCodigo;
            int contador = 0;
            BufferedReader b1 = new BufferedReader(
                        new FileReader(arF));
            leido = b1.readLine(); // 1a. lectura codigo
            idCodigo = leido.charAt(0); // identifica codigo
            leido = b1.readLine();  // 2a. lectura bases
            boolean contiene;
            while (contador < 4){
                contiene = Pattern.matches("(A|C|G|T|N)+", leido);
                if (contiene) contador++;   //va bien
                fragSz = leido.length();
                b1.readLine();  //lee extra
                b1.readLine();  //lee Q
                leido = b1.readLine();  // lee codigo
                if (leido.charAt(0) == idCodigo){
                    idCodigo = leido.charAt(0);
```

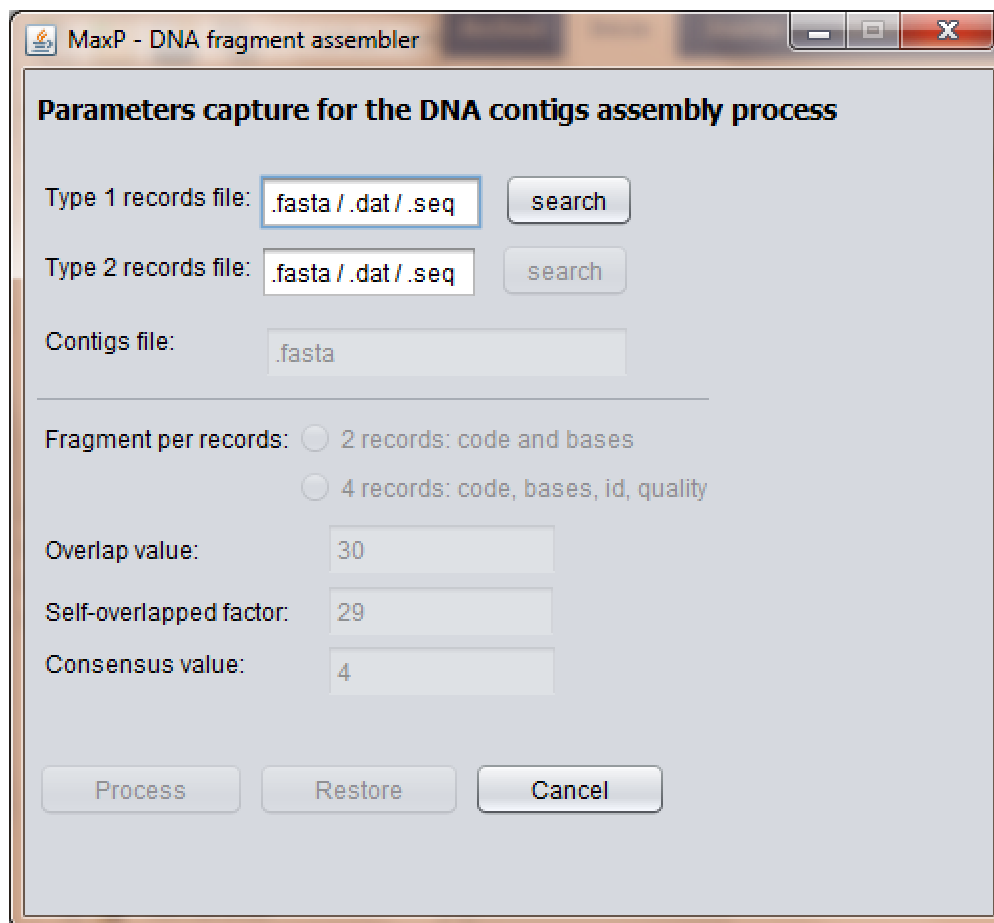
```
                contador++;            //va bien
            }
            leido = b1.readLine();  // lee bases
          }// fin de while
          b1.close();
          if (contador == 4) return true; //todo bien
        } catch (IOException ioe) {
          System.out.println("problems when checking records: "+ioe);
          return false;
        }// fin de ioexception
    return false;
}
private File pintaPanel(int ta) {
    JFileChooser fc = new JFileChooser();
    int retcod = fc.showOpenDialog(jPanel1);
    if (retcod == JFileChooser.APPROVE_OPTION){
        File file = fc.getSelectedFile();
        try {
          if (ta==1)  ruta1 = file.getCanonicalPath();
          else       ruta2 = file.getCanonicalPath();
        } catch (IOException ex) {
          Logger.getLogger(Parametros.class.getName()).log(Level.SEVERE, null, ex);
        }
        return file;
    }else{
        return null;
    }
}
private void btnBuscar1ActionPerformed(java.awt.event.ActionEvent evt) {
    jLabel11.setVisible(true);
    try {
        archF1 = pintaPanel(1).getCanonicalPath();
    } catch (IOException ex) {
        Logger.getLogger(Parametros.class.getName()).log(Level.SEVERE, null, ex);
    }
    if (!archF1.isEmpty()) txtArchT1.setText(archF1);
    jLabel11.setVisible(false);
    btnBuscar2.setEnabled(true);
    btnReset.setEnabled(true);
}
private void btnBuscar2ActionPerformed(java.awt.event.ActionEvent evt) {
    jLabel11.setVisible(true);
    try {
        archF2 = pintaPanel(2).getCanonicalPath();
    } catch (IOException ex) {
        Logger.getLogger(Parametros.class.getName()).log(Level.SEVERE, null, ex);
    }
    if (!archF2.isEmpty()) txtArchT2.setText(archF2);
    jLabel11.setVisible(false);
    txtArchCntg.setEnabled(true);
}
private void rbt2RegsActionPerformed(java.awt.event.ActionEvent evt) {
    // radio button que indica que son registros de 2 en 2
    if (rbt2Regs.isSelected())
        // checar que haya lecturas de 2 en 2 en ambos archivos
        if (checa2En2(archF1) && checa2En2(archF2)){
          regPF = 2;
          txtValTras.setEnabled(true); // esta bien
          txtValTras.setEditable(true);
        }else{
```

```
         txtValTras.setEnabled(false);// esta mal
      }
}
private void rbt4RegsActionPerformed(java.awt.event.ActionEvent evt) {
    // radio button que indica que son registros de 4 en 4
   if (rbt4Regs.isSelected())
      // checar que haya lecturas de 4 en 4 en ambos archivos
      if (checa4En4(archF1) && checa4En4(archF2)){
         regPF=4;
         txtValTras.setEnabled(true); // esta bien;
         txtValTras.setEditable(true);
      }else {
         txtValTras.setEnabled(false); // esta mal
      }
}
private void btnResetActionPerformed(java.awt.event.ActionEvent evt) {
   desactiva();
   //initComponents();
}
private void txtValTrasFocusLost(java.awt.event.FocusEvent evt) {
   try {
      valTr = Integer.parseInt(txtValTras.getText());
      txtValFactorR.setEnabled(true);
   }catch (NumberFormatException nfe) {
      txtValFactorR.setEnabled(false);
      txtValTras.requestFocus();
   }
}
private void txtValTrasFocusGained(java.awt.event.FocusEvent evt) {
   txtValTras.setSelectionStart(0);
   try {
      valTr = Integer.parseInt(txtValTras.getText());
      txtValFactorR.setEnabled(true);
   }catch (NumberFormatException nfe) {
      txtValFactorR.setEnabled(false);
      txtValTras.requestFocus();
   }
}
private void txtValConsFocusGained(java.awt.event.FocusEvent evt) {
   txtValCons.setSelectionStart(0);
   try {
      valCons = Integer.parseInt(txtValCons.getText());
      btnProcesar.setEnabled(true);
      btnProcesar.requestFocus();
   }catch (NumberFormatException nfe) {
      btnProcesar.setEnabled(false);
      txtValCons.requestFocus();
   }
}
private void txtValConsFocusLost(java.awt.event.FocusEvent evt) {
   try {
      valCons = Integer.parseInt(txtValCons.getText());
      btnProcesar.setEnabled(true);
      btnProcesar.requestFocus();
   }catch (NumberFormatException nfe) {
      btnProcesar.setEnabled(false);
      txtValCons.requestFocus();
   }
}
private void txtArchCntgFocusLost(java.awt.event.FocusEvent evt) {
```

```
        archCntgs = txtArchCntg.getText();
        if (archCntgs.contains(" ")) txtArchCntg.requestFocus();
        //una o mas letras, puede haber numeros, un punto, extension
        boolean contiene =Pattern.matches(
              "^[a-zA-Z]+[0-9]*\\.[a-zA-Z]+[0-9]*", archCntgs);
        if (!contiene) txtArchCntg.requestFocus();
        // todo bien !!
        rbt2Regs.setEnabled(true);
        rbt4Regs.setEnabled(true);
    }
    private void txtArchCntgFocusGained(java.awt.event.FocusEvent evt) {
        btnProcesar.setEnabled(false);
    }
    private void txtValFactorRFocusGained(java.awt.event.FocusEvent evt) {
        txtValFactorR.setSelectionStart(0);
        try {
            factorR = Integer.parseInt(txtValFactorR.getText());
            txtValCons.setEnabled(true);
        }catch (NumberFormatException nfe) {
            txtValCons.setEnabled(false);
            txtValFactorR.requestFocus();
        }
    }
    private void txtValFactorRFocusLost(java.awt.event.FocusEvent evt) {
        try {
            factorR = Integer.parseInt(txtValFactorR.getText());
            txtValCons.setEnabled(true);
        }catch (NumberFormatException nfe) {
            txtValCons.setEnabled(false);
            txtValFactorR.requestFocus();
        }
    }
    // -------------------------------------------------------
    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        /* Set the Nimbus look and feel */
        //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
        /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
         * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
         */
        try {
            for (javax.swing.UIManager.LookAndFeelInfo info :
javax.swing.UIManager.getInstalledLookAndFeels()) {
                if ("Nimbus".equals(info.getName())) {
                    javax.swing.UIManager.setLookAndFeel(info.getClassName());
                    break;
                }
            }
        } catch (ClassNotFoundException ex) {

java.util.logging.Logger.getLogger(Parametros.class.getName()).log(java.util.logging.Level.SEVERE,
null, ex);
        } catch (InstantiationException ex) {

java.util.logging.Logger.getLogger(Parametros.class.getName()).log(java.util.logging.Level.SEVERE,
null, ex);
        } catch (IllegalAccessException ex) {
```

```
java.util.logging.Logger.getLogger(Parametros.class.getName()).log(java.util.logging.Level.SEVERE,
null, ex);
        } catch (javax.swing.UnsupportedLookAndFeelException ex) {

java.util.logging.Logger.getLogger(Parametros.class.getName()).log(java.util.logging.Level.SEVERE,
null, ex);
        }
        //</editor-fold>

        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new Parametros().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    private javax.swing.JButton btnBuscar1;
    private javax.swing.JButton btnBuscar2;
    private javax.swing.JButton btnCancelar;
    private javax.swing.JButton btnProcesar;
    private javax.swing.JButton btnReset;
    private javax.swing.ButtonGroup buttonGroup1;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel10;
    private javax.swing.JLabel jLabel11;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JLabel jLabel5;
    private javax.swing.JLabel jLabel6;
    private javax.swing.JLabel jLabel7;
    private javax.swing.JLabel jLabel8;
    private javax.swing.JLabel jLabel9;
    private javax.swing.JPanel jPanel1;
    private javax.swing.JSeparator jSeparator1;
    private javax.swing.JRadioButton rbt2Regs;
    private javax.swing.JRadioButton rbt4Regs;
    private javax.swing.JTextField txtArchCntg;
    private javax.swing.JTextField txtArchT1;
    private javax.swing.JTextField txtArchT2;
    private javax.swing.JTextField txtValCons;
    private javax.swing.JTextField txtValFactorR;
    private javax.swing.JTextField txtValTras;
    // End of variables declaration
}

//////////////////////////////////
```

Interface gráfica de captura de parámetros:



8.8 Publicación 1: Modified Classical Graph Algorithms for the DNA Fragment Assembly Problem

Algorithms **2015**, *8*, 754-773; doi:10.3390/a8030754

OPEN ACCESS

algorithms

ISSN 1999-4893
www.mdpi.com/journal/algorithms

Article

Modified Classical Graph Algorithms for the DNA Fragment Assembly Problem

Guillermo M. Mallén-Fullerton 1,†, J. Emilio Quiroz-Ibarra 2,†, Antonio Miranda 3,† and Guillermo Fernández-Anaya 3,†,*

[1] Engineering Department, Universidad Iberoamericana Ciudad de México, Prolongación Paseo de la
Reforma 880, Lomas de Santa Fe, Distrito Federal 01219, Mexico;
E-Mail: guillermo.mallen@ibero.mx

[2] Universidad Iberoamericana Ciudad de México, DCI, Prolongación Paseo de la Reforma 880, Lomas de Santa Fe, Distrito Federal 01219, Mexico; E-Mail: quirozem@yahoo.com.mx

[3] Physics and Mathematics Department, Universidad Iberoamericana Ciudad de México, Prolongación Paseo de la Reforma 880, Lomas de Santa Fe, Distrito Federal 01219, Mexico; Email: antonio.miranda@ibero.mx

† These authors contributed equally to this work.

* Author to whom correspondence should be addressed; E-Mail: guillermo.fernandez@ibero.mx; Tel.: +52-55-5950-4000.

Academic Editors: Giuseppe Lancia and Alberto Policriti

Received: 24 June 2015 / Accepted: 26 August 2015 / Published: 10 September 2015

Abstract: DNA fragment assembly represents an important challenge to the development of efficient and practical algorithms due to the large number of elements to be assembled. In this study, we present some graph theoretical linear time algorithms to solve the problem. To achieve linear time complexity, a heap with constant time operations was developed, for the special case where the edge weights are integers and do not depend on the problem size. The experiments presented show that modified classical graph theoretical algorithms can solve the DNA fragment assembly problem efficiently.

Keywords: DNA fragment assembly; minimum spanning tree; heap; linear complexity

1. Introduction

Since its discovery by Watson and Crick [1], the importance of DNA to biology, medicine and human kind has been evident. However, we had to wait some time until it was possible to sequence the DNA bases, which was accomplished by Sanger in the mid-1970s [2], by detecting small dark bands on a thin gel layer. This method severely constrains the length of the DNA to be sequenced due to the limited resolution of the dark bands. In order to solve this problem, the DNA is first cut in known places by using restriction enzymes, which will divide the DNA at the point where a specific base sequence is found. The resulting sub-sequences, are divided again, this time using a different restriction enzyme, and this process is repeated until the resulting fragments can be divided in random places yielding sub-fragments that are small enough as to provide the complete sequence of bases using Sanger's method. This process was slow and expensive because every time that a sequence was divided, each resulting subsequence had to be cloned in order to have enough material for the next division. This problem prompted scientists to start dividing sequences in random positions from the beginning, instead of doing it after several divisions, saving time and money [3]. This method, in turn, created a severe computational problem since there is no information about the order of the fragments. If one starts with many copies of the original problem and the derived fragments are sequenced, then each base of the original problem could appear in several fragments. The sum of the lengths of the fragments divided by the length of the original problem is known as its *coverage*. We hope that if we have a large enough coverage, the fragments will overlap in such a way that the sequence at the end of a fragment overlaps with the sequence at the beginning of another one. Hence, in theory, we should be able to use this information to sort the fragments and reconstruct the original DNA sequence.

What really happens is not very simple since, by chance, there are a huge number of false overlaps amongst the fragments. There is also the problem that fragment sequencing is not perfect. Two percent error is common, as well as having fragments that do not belong to the original problem, some due to contamination with DNA foreign to the problem, and others because of the appearance of chimeras from the process of DNA cloning, in which bacteria DNA is used, so that we could get fragments with portions of bacteria DNA. On top of what was said, we have repeats, which are DNA sections that appear tens or hundreds of times one after another (tandem repeats) or in far away parts of the original DNA (interspersed repeats) [4].

As far as we know, at this point there is no good probabilistic model of DNA. Simple, or relatively simple ones do not explain the repeats. For instance, in the real life problem that we will present later in this paper, there are sequences of A bases that are extremely long. Figure 1 show some examples of fragments that come from the sample problem that we will use in Section 4 of this paper and Table 1 shows the probability that it happens considering that among the total number of fragments, the A bases occurs with probability 0.3249. As can be seen from the figure, the number of times that the A chain occurs in the real data is much larger than the expected value according to a simple probabilistic model. Despite these anomalies and some others, some of the parameters that are used in the assembly of fragments, such as the minimum number of bases that an overlap must have to be considered important, have an empirical basis [5].

There is also the problem that in real situations, sections of the original DNA problem have no coverage since fragments are obtained by a random process and it is not possible guarantee that there are fragments from every part of the genome under study. This is the reason why the resulting assembly of all fragments is a set of *contigs* or subsequences and almost never the entire sequence of the problem. It is the job of specialized biologists to fill in the gaps and correct assembly errors made during the automated process.

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAACAAAAACAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAACAAATAAAAAAAC

AAAAAAAAAAAAAAACAAAAAAAAAATAAAAAAAA

ACAAAAAAAAAAAAAAAAACAAAAAAAAAAAAAAA

Figure 1. Some fragments from the *S. aureus* strain MW2 problem.

Table 1. Observed frequency on *n* bases A *vs.* expected value for p = 0.3249 and 5,324,340 fragments.

n	Observed	Probability	Expected Value
15	101	0.000000475	2.528889873
16	74	0.000000147	0.784225487
17	65	0.000000046	0.242640081
18	51	0.000000014	0.074884674
19	40	4.32861E−09	0.023046972
20	39	1.32807E−09	0.007071095
21	18	4.06052E−10	0.002161959
22	28	1.23662E−10	0.000658416
23	14	3.74924E−11	0.000199622
24	15	1.13089E−11	0.000060212
25	22	3.3908E−12	0.000018054
26	15	1.00958E−12	0.000005375
27	22	2.9809E−13	0.000001587
28	11	8.71281E−14	0.000000464
29	15	2.51495E−14	0.000000134
30	10	7.14487E−15	3.80417E−08
31	9	1.98796E−15	1.05846E−08
32	6	5.37563E−16	2.86217E−09
33	4	1.3946E−16	7.42531E−10
34	4	3.38757E−17	1.80366E−10
35	1	7.29107E−18	3.88201E−11

DNA fragment sequencing technology has continued to advance while costs continue to go down. The original *shotgun* technology has been called “long reads” because the fragments would normally have more than 500 bases. Using new technologies, commonly known as Next Generation Sequencing, the cost to sequence a million bases is about $1 [6]. Unfortunately, the length of the fragments is shorter using this newer technology. Today, 150 bases reads are common with the Illumina sequencers. To compensate for the problems derived from the short size of each fragment, coverage of about 40 is needed instead of a coverage of about 10 in the case of long reads. If we consider the number of fragments to be sequenced with newer technologies, we need

to assemble three to four million fragments to sequence a bacterium, while using long reads, only 50,000 fragments were enough. These are really bad news due to the combinatorial nature of the solutions to the problem.

Since the very beginning of DNA fragment sequencing, greedy algorithms have been used; for instance, we pick a random fragment and connect to it the one that has the longest overlap. This process is repeated until no overlapping fragments are available. Nowadays, this technique would be difficult to use, since fragments are too short, which often produces several different fragments with the same overlap size. Another popular technique is the use of De Bruijn graphs [7]. In order to build these graphs, all possible *k-mers* (subsequence of *k* consecutive bases) of each fragment are used to find all possible overlaps among them. Using these graphs, several problems can be solved by using heuristics; for instance, given a *k-mer*, there might be two parallel paths to other *k-mer*, creating what is called a bubble. The bubble is then popped by removing the shortest side or projecting one side onto the other when they have the same length. Short sequences that originate from long ones are eliminated. It is common to use a value of *k* close to 20 in order to obtain the *k-mers*.

An approach based on genetic algorithm optimization was suggested by Parsons in 1993 [8]. Later, other scientists applied similar metaheuristics [9–12], but their results were based on very small benchmarks, the largest had about 1000 fragments, and even though results were improving slowly, they were still far from real problems, including the long reading ones, with tens of thousands of fragments. In 2013, a reduction from fragment DNA sequencing to the Traveling Salesman Problem (TSP) was used [13]. Taking advantage of formal heuristics and algorithms for the TSP, optimal values were found for the most commonly used benchmarks, and, for the first time, a real life problem was solved using optimization.

The remainder of this paper is organized as follows: Section 2 provides the basic ideas on the use of graph theory for the solution of DNA sequencing problems; Section 3 explains those algorithms that are necessary to tackle the problem; Section 4 illustrates the use of our algorithms on real life problem benchmarks; and, in Section 5, we give our conclusions and future work.

2. Use of Graph Theory

2.1. Generalities

In reference [13], a graph theory approach was used by means of the TSP, where the solution obtained from the assembly of fragments is a series of *contigs* rather than a Hamiltonian Path. In TSP, all possible edges of the graph are considered, but when there is no possible connection between two nodes, it is usually represented using a very large weight to prevent it from showing up in the final solution. Applying TSP to DNA fragment assembly might be considered excessive, even though the expected results are obtained, since the fraction of real connections is very small. A graph theoretical approach using other algorithms might give better results.

The size of real life DNA fragment assembly problems is huge, with 35 base fragments and a coverage of 40, we get about 700 million fragments for human chromosome number 1, which has 245 million base pairs. It is known that the DNA fragment assembly problem is NP-hard (Non-deterministic Polynomial time hard), since it can be reduced from the shortest common superstring problem [14]; in practice, we must only use linear time algorithms, even if by doing so we sacrifice correctness and obtain only an approximate solution.

2.2. DNA Fragment Assembly as a Graph

The DNA fragment assembly problem can be transformed into a directed graph: we need to find a sequence of fragments where each one is always the prefix of the next one. We know that DNA

comes in pairs of strands (double helix) that run in opposite directions, where in front of each base of one strand, there is a base of the reverse-complement on the other; for instance, an A in front of a T and a C in front of a G, which clearly means that the information contained in each one of the strands is the same. When DNA fragment overlaps are obtained, it is not known what strand they come from, *i.e.*, every time that an overlap is computed, the strand and its reverse-complement must be considered. This means that a prefix of a normal overlap becomes a suffix of the reverse-complement. At first, it might seem that this case could be represented by a simple non-directed edge between the two fragments. However, this is not so if we consider that each node or fragment that is a reverse-complement must also be a reverse-complement for every edge that connects to it.

As a small example, we present a set of 13 fragments, Table 2, the associated overlap matrix, Table 3 and the resulting graph, Figure 2.

Table 2. Graph example: fragments.

Fragment number	**Sequence**
1	GATGTACTATTTGAAGCTTAT
2	TACTATTTGAAGCTTATTCGG
3	TGAAGCTTATTCGGCCAATCG
4	ATGTACTATTTGATGAAGCTT
5	GCTTATTCGGCCAATCGTACT
6	CAATCGTACTGATGTACTATT
7	CGTACTATTTACTGATGTACA
8	CGGCCAATCGTACTGATGTAC
9	ATGTACTATTTGATGAAGCTT
10	GATGAAGCTTATTCGGCCAAT
11	AAGCTTATCAGTACGTGGAAC
12	ACTATTTACTGATGTACAATA
13	ACAATAGTACATCAGTAAAAA

Table 3. Graph example: overlaps matrix.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
1					20								
2					24								
3						20		24					
4										24			
5						20	20						
6									24			20	
7													17
8									20				
9										24	20		
10								22					
11													
12													20

13

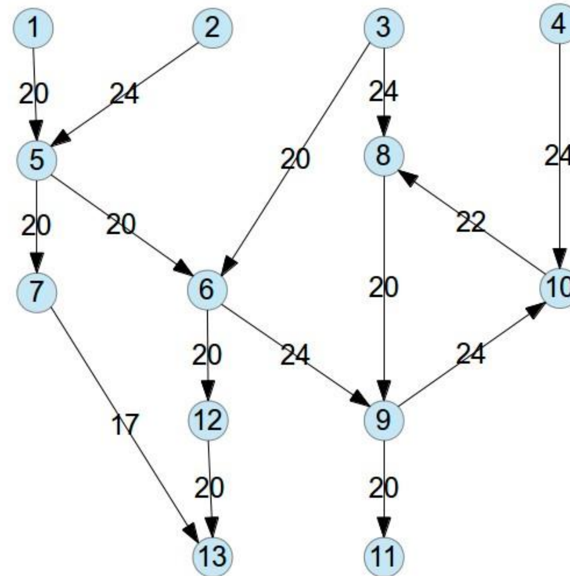


Figure 2. Graph example: graph.

2.3. Objective Function

In the literature, using nature inspired algorithms to solve the DNA fragment assembly problem published after Parsons [8], the associated graph has been considered to be an undirected graph. Other literature considered bi-directed graphs, especially when De Bruijn graphs were proposed. It is common to use a Hamiltonian path, even though sometimes Eulerian paths have been used too [7]. In any case, it is not entirely clear how many times each one of the fragments should be used, since, from the physical point of view, it is feasible for a sequence to repeat in different sections of the DNA, and for identical fragments to come from different places. Thinking that a fragment and it reverse-complement are the same, creates too much complexity from the algorithmic point of view as well as from the programming one, and would only be justified as long as it was strictly required to consider Hamiltonian paths. Hence, we consider each reverse-complement fragment as an independent fragment. Another possibility might be to trust the huge cover that is customary in Next Generation Sequencing (NGS) and not use reverse-complements.

With respect to the objective function for the optimization, in many papers, the sum of the overlaps of each fragment with the next one is used, which has probably been inherited from the use of some greedy algorithms. In addition to this objective function, we propose to maximize the sum of the lengths of the *contigs*. This can be achieved by considering that each time that a new fragment is added at the end of a *contig*, its length is incremented by an amount equal to the length of the fragment minus the overlap length. So we would have the following objective functions:

$$\max F_1 = \sum_{i=1}^{n-1} w(i, i+1) \tag{1}$$

$$\max F_2 = \sum_{i=1}^{n-1} (L - w(i, i+1)) \tag{2}$$

where $w(i, i + 1)$ is the number of overlapped bases between fragment i and the next one in the sequence, n is the number of fragments, and L is the length of the fragment. Notice that Equation (2) can easily be transformed into a minimization:

$$\min F_2' = \sum_{i=1}^{n-1} w(i, i+1) \tag{3}$$

Because

$$F_2 = (n-1)L - F_2' \tag{4}$$

The boundary between *contigs* appears because there are values of $w(i, i + 1)$ that are equal to 0. Since there is usually some contamination with foreign DNA and there might be chimeras, some of the obtained *contigs* do not belong to the original DNA, and, at some point, which is not within the scope of this paper, useful *contigs* must be separated from the useless ones.

In order to obtain each *contig*, an agreement must be reached (computed) for each base, *i.e.*, since we know that there might be redundancy for each base (due to the original cover) and that there might be sequencing errors in the fragments, it is possible for each base in each *contig* to have several versions, from which the most frequent value must be accepted, even if it is the minimum acceptable value. Since most of the errors in the *contigs* are at the ends, due to chemical problems in the sequencing equipment or because of the lack of enough redundancy, it might be necessary to cut them off.

3. Algorithms

3.1. Basic Algorithm for F1 and F2

For the objective function F_1, we propose an algorithm similar to topological sorting. Let $G = (V, E)$ be a directed acyclic graph (DAG) where V is the set of vertices, E is the set of edges and $w(u, v)$ is the weight of edge $u \rightarrow v$. In this graph, the longest path will go from a node that has no edge going in (initial vertex) and those nodes that are connected to it. Notice that there might be several "initial vertices", and that there is no single privileged node. Let $S \subseteq V$ be the set of vertices with in-degree $d_{in}(v) = 0$ and let Q be a stack. The algorithm to determine the longest distance from a given vertex to each of the other vertices is the following (Algorithm 1):

Algorithm 1. Longest distance from initial vertices

1. For each vertex $v \in S$ with $d_{in}(v) = 0$
   1.1. *push(Q,v)* 1.2. *d(v):*=0 1.3. *origin(v):=v*
2. While Q is not empty
   2.1. *u:=pop(Q)*
   2.2. for each vertex v $(u,v) \in E$
      2.2.1. $d_{in}(v) := d_{in}(v) - 1$
      2.2.2. if $d_{in}(v)=0$ 2.2.2.1. *push(Q,v)*
      2.2.2.2. $d(v) := \max_{(x,v)\in E} \left(d(x) + w(x,v)\right)$

2.2.2.3. father(v):=x

This algorithm only works if the graph is acyclic. In real life DNA fragment assembly problems, there are cycles. When this algorithm is applied to a graph with cycles, neither the cycles nor the nodes that are further from the cycles are detected. Later we will discuss how to take care of cycles.

3.2. Constant Time Heap

Aside from a modified version of the algorithm of Section 3.1, there are other ways to tackle the cycle problem. One way, not necessarily the best way, would be to build a minimum spanning tree, even if it ignores the direction of the edges. It is clear that if there are no undirected cycles in the equivalent undirected graph, there should be no directed cycles in its directed version. So we could use one of several algorithms like Prim [15] and Kruskal [16], both with time complexity in $O(|E| \log|V|)$ when using a Fibonacci Heap.

In order to achieve the linear time complexity that we require, we notice that in the case of the DNA fragment sequencing problem, we should use a heap that can insert and extract-min in constant time, thus improving the time complexity of other data structures where the extract-min operation is in $O(\log n)$.

Our heap requires that the ordering values are integers independent from the number of nodes in the graph. In the DNA fragment sequencing problem, the edges of the graph are the fragment overlaps, hence it is an integer value representing the number of bases, and its value is also bounded by $0 < w(u, v) < L$, where L does not depend of the number of fragments, and has a value of at most a few hundred bases. The heap that we propose is based on a *Stack* P_i for each possible value to be introduced. The heap operations are:

Algorithm 2. Stack heap insert and extract min

---

insert x

1. push(Px,x) 2. if x<xmin

2.1. *xmin:=x*
3. if x > xmax

3.1. xmax:=x

extract-min

1. x:=pop(Pxmin)

2. While the stack P_{xmin} is empty and $xmin \leq xmax$

2.1. xmin: = xmin + 1

In the extract-min operations shown, the value that is in the stack corresponding to x_{min} is extracted. If the stack is empty, the next non-empty stack smaller than the maximum value is used.

When the number of stacks is large enough, it is useful to use some other kind of heap, such as a binary or a Fibonacci heap instead of sequential search. In the example that we provide in Section 4, we use no more than 34 stacks ($0 \le x \le 34$), hence sequential search turns out to be faster. The time complexity of the insert operation is constant because Steps 1 through 3.1 always require the same amount of time. In the extract-min operation, Step 1 is of constant time and Step 2, which is only executed when the stack with the minimum values is empty, in the worst case depends on the value x_{max}, and could be linear, in the case of sequential search, or logarithmic if a heap is used. In the case of DNA fragment assembly, millions of edges or nodes will be inserted into the heap, hence the value of x_{max} should be close to a few hundred, which makes the probability of Step 2 executing more than once negligible. In any case, since the time complexity of our heap is independent from the number of nodes or vertices inserted into it, then Prim's algorithm [15] as well as Kruskal's [16] algorithm become linear. Even though our technique is designed to work in the particular case of DNA fragment assembly, it is possible that it could also be used in other problems.

3.3. MST in Linear Time

As mentioned before, when using our heap, Algorithm 2, Prim's algorithm as well as Kruskal's algorithm become linear, and even though it is possible to implement operations such as delete-key and decrease-key in our heap, it is more convenient to allow the edge values to create a cycle and discard them when leaving the heap, since this is more efficient and is also of constant time. Let *M(F, V)* be the MST of graph *G(E, V)* and let *S* be the set of vertices that have been added to the MST at some point during the execution of Prim's algorithm. Here is our version of Prim's algorithm:

Algorithm 3. Prim's algorithm

---

1. For some vertex *u*
   - 1.1. Put *u* in *S*
   - 1.2. For each $(u,v) \in E$
     - 1.2.1. insert *w(u,v)* in the heap
2. While the heap is not empty
   - 2.1. Extract the edge *(u, v)* from the heap
     - 2.1.1. if $u \notin S$
       - 2.1.1. 1.Put *u* in *S* and *(u,v)* in *F*

- 2.1.1.1.1. For each $(u,x) \in E$ insert *w(u,x)* in the heap
- 2.1.2. if $v \notin S$
  - 2.1.1.1.Put *v* in *S* and *(v,u)* in *F*
    - 2.1.2.1.1. For each $(v,x) \in E$ insert *w(v,x)* in the heap

---

Since at most $|E|$ edges will be inserted and extracted in constant time, Algorithm 3 will have a time complexity in $O(|E|)$.

Now, our version of Kruskal's algorithm (Algorithm 4):

Algorithm 4. Kruskal's algorithm

1. Insert *E* in a *heap* in increasing order of *w(u,v)*
2. Create a forest *F* where each vertex is an independent tree
3. While the heap is not empty
   3.1. Extract *(u,v)* from heap
      3.1.1. If *u* and *v* belong to different trees, merge both trees

In order to merge the trees, we use union by rank with path compression [17]. The time complexity of Step 1 is $O(|E|)$, Step 2 is $O(|V|)$. Step 3 is repeated $|E|$ times and every time the following operations are performed: extraction of the minimum value from the heap, a find of two nodes, and in the worst case, the union of two trees. Find is in $O(1)$ and the union is $O(\log^*|E|)$ [17], which for problems with 65,535 edges up to the maximum physical size of DNA fragment assembly problems is 5; therefore, in practice the algorithm is in $O(1)$. The total running time of Kruskal's algorithm using our heap, Algorithm 2, with union by rank and path compression, applied to the DNA fragment assembly problem is in $O(|E| + |V|)$. Once the MST has been computed, its maximum distances are computed using Algorithm 1. In this case, there are no cycles (directed or undirected), so Algorithm 1 works correctly.

3.4. Modification of the Basic Algorithm

There are two kinds of cycles that cause trouble: cycles that are connected to a start node, and disconnected cycles. In both cases, there might be sets of nodes that are created like attachments to nodes of the cycle, but do not belong to them, and are not detected by Algorithm 1. In the case of disconnected cycles, we would need to take any node as a start node, which can require some trial and error since a randomly selected node could be attached to the cycle but not necessarily in the cycle itself. However, disconnected cycles in real life DNA fragment sequencing problems are rare, and when they appear, they are usually very small, two or three nodes, that lead to *contigs* that are too small and that would have been discarded anyway.

A cycle connected to a start node of the graph appears when there are nodes for which, even when paths that go from a start node to another node *u* in the cycle, the value of $d_{in}(u)$ never goes to zero because at least one edge comes from the cycle. When Algorithm 1 is done, we can take one of the nodes that went through Step 2.2.1, but that remained with a final value of $d_{in}(u) > 0$, and insert it into the stack to continue with Step 2 of the algorithm. It is necessary, however, to mark those nodes so that they will not be considered again and to avoid the cycle being traversed more than once. The total distance from a start node to the last processed node in the cycle is the sum of the distance from a start point to the cycle entry point plus the length of the cycle, minus the edge that would close the cycle on the start node in the cycle. So we have that if the selected node is *s*, the previous node in the cycle is *t*, and the total length of the cycle is *C* then:

$$d(t) = d(s) + C - w(t,s) \quad (5)$$

Among the distinct entry nodes to the cycle, we could select one with the longest distance to a start node. However, it does not guarantee that we will always find the longest distance to node *t* since the value of *w(t, s)* changes with the selection of *s* and there could be another cycle entry node with a smaller value of *d(s)* but with a small enough value of *w(t, s)* to give a longest distance. In the case of the DNA fragment assembly problem the values of *w( )* are in a very narrow range, most of the distances to the start node are big and it is uncommon for them to be similar, so that if

our heuristic consists of selecting an s with the longest distance to the start point, we will have a high probability of finding the maximum distance. The modified algorithm (Algorithm 5) is as follows:

Algorithm 5. Modified maximum distance algorithm

---

1. For each vertex $v \in S$ with $d_{in}(v) = 0$
   1.1. push(Q,v)
   1.2. d(v):=0 1.3. origin(v):=v
2. While there are vertices *v* with $d_{in}(v) > 0$
   2.1. While *Q* is not empty
      2.1.1. u:=pop(Q)
      2.1.2. For every vertex *v* where $(u,v) \in E$ and *v* is not marked
         2.1.2.1. $d_{in}(v) := d_{in}(v) - 1$
         2.1.2.2. If $d_{in}(v) = 0$
            2.1.2.2.1. *push(Q,v)*
            2.1.2.2.2. $d(v) := \max_{(x,v) \in E} \left(d(x) + w(x,v)\right)$
            2.1.2.2.3. insert *d(v)* in a *heap*
            2.1.2.2.4. father(v):=x 2.1.2.2.5. origin(v):=origin(x)
   2.2. If the *heap* is not empty
      2.2.1. extract the node *v* from *heap*
         2.2.1.1. If $d_{in}(v) > 0$
            2.2.1.1.1. Make $d_{in}(v) = 0$
            2.2.1.1.2. Insert *v* in the stack
            2.2.1.1.3. Mark the node *v*
            2.2.1.1.4. Go to step 2.1

Notice that Step 2.2.1.1 is required since node *v* could have gone through Step 2.1.1 and already be in a path. In this case, we can use the heap to keep the time complexity of the algorithm linear.

3.5. Assembly Algorithm

The output from Algorithms 1 and 4 is a sequence of directed edges and nodes that allow us to directly build the *contigs*. Considering reverse-complement fragments as independent of the original ones makes the job easier. The reconstruction of the sequence of DNA in the *contig* consists of placing the first fragment, and then adding the overlap-free portion of the next fragment. In our method, we do not need to reach a computed agreement as in other methods [18] since we do not differentiate among overlapped bases from one or other fragment. However,

the ends of the *contig* might have sequencing errors and there is nothing to verify that there are no such errors. The solution is to cut off the ends in such a way that a minimum agreement is reached, *i.e.*, that the bases must appear in at least a given number of fragments.

4. Experiments

In order to test our algorithms, we used several benchmarks that are available for use by researchers [19]. We selected the benchmarks of *S. aureus* MW2 with short reads. The benchmarks that we used have the following number of fragments: 2,278,504, 730,201, 298,194, 89,718 and 22,447. From now on, we will refer to each one of these problems by its number of fragments. In order to have something to compare our results to, we ran the Edena assembler using the same data, Table 4. This comparison has some limitations due to the fact that Edena uses a series of heuristics in order to refine its solution, by eliminating *contigs* that have no real meaning, while our results are those directly generated by our algorithms with no refinement. Refining the raw results from our algorithms might be part of our future research.

Table 4. Edena benchmark results.

		Problem				
Fragments		22,448	89,718	298,194	730,201	2,278,504
Contigs Obtained		186	258	575	837	2046
Total Number of Bases		26,284	102,684	314,120	788,198	2,526,278
Contigs Found	**Contigs Found**	184	256	565	823	1920
	%	98.9	99.2	98.3	98.3	93.8
	Bases Found	26,024	102,249	310,965	777,583	2,463,563
	%	99	99.6	99	98.7	97.5
	N50	188	1247	2911	4132	5250
	Average Length	141.4	399.4	550.4	944.8	1283.10
	Minimum Length	52	52	52	52	52
	Maximum Length	880	7295	10,985	20,521	20,579
Contigs not Found	**Contigs not Found**	2	2	10	14	126
	%	1.1	0.8	1.7	1.7	6.2
	Bases not Found	260	435	3155	10,615	62,715
	%	1	0.4	1	1.3	2.5
	N50	NA	NA	NA	NA	5599
	Average Length	130	217.5	315.5	758.2	497.7
	Minimum Length	99	99	56	56	52
	Maximum Length	161	336	2521	5941	18,158

From each benchmark, we took only the fragment file in FASTA format, and from it we computed the overlaps among all fragments considering that reverse-complement fragments are

also independent fragments (In the original benchmarks, there is a file of overlaps where reverse-complements are not considered as independent fragments). To compute overlaps, we used a *trie* (prefix tree). The computed overlaps are all larger than 20 bases. From these overlaps other sets of test data were obtained: one discarding redundant overlaps because of transitivity, and the other one, computing the MST. In all of the cases, the minimum agreement in Algorithm 5 was 4. The *contigs* obtained in each experiment were searched in the complete genome [20].

In Experiment 1, we used all of the overlaps larger than 20 bases and executed Algorithm 1 for the two objective functions. The results are presented in Tables 5–7.

Table 5. Experiments summary.

			Problem				
Algorithm		**Fragments**	22,448	89,718	298,194	730,201	2,278,504
Edena	**Edena**	**Contigs Found**	184	256	565	823	1920
		%	99.2	98.9	98.3	98.3	93.8
		Bases Found	26,024	102,249	310,965	777,583	2,463,563
		%	99	99.6	99	98.7	97.5
Algorithm 1	**F1 Objective Function, all Overlaps.**	**Contigs Found**	74	101	149	204	480
		%	90.2	88.6	91.4	91.1	94.1
		Bases Found	15,822	70,250	206,826	463,582	1,102,152
		%	81.8	83	86.1	83.9	83.4
Algorithm 1	**F2 Objective Function, all Overlaps**	**Contigs found**	74	102	146	189	450
		%	90.2	89.5	89.6	84.4	88.2
		Bases found	15,515	70,843	189,927	369,001	931,343
		%	81.5	83.8	79	66.8	69.9
Algorithm 1	**F1 Objective Function, no Transitive overlaps**	**Contigs found**	74	102	149	204	481
		%	90.2	89.5	91.4	91.1	93.9
		Bases found	15,806	71,652	206,826	462,774	1,103,587
		%	83.7	84.6	86.1	83.9	82.7
Algorithm 1	**F2 Objective Function, no Transitive Overlaps**	**Contigs found**	75	103	142	181	417
		%	91.5	90.4	87.1	80.8	81.4
		Bases found	15,599	72,245	174,244	320,502	791,376
		%	84	85.4	72.5	58.1	58.9
Algorithm 1	**F1 Objective Function, MST**	**Contigs found**	304	1827	5980	15,443	46,187
		%	91	97	98.4	98.9	95.7
		Bases found	35,980	256,641	815,994	2,081,306	6,208,258
		%	89.8	96.6	98.1	98.7	93.9
Algorithm 5	**F1 Objective function, no Transitive Edges**	**Contigs found**	74	102	150	205	480
		%	90.2	88.7	91.5	91.1	94.1
		Bases found	15,822	70,318	206,927	463,684	1,102,152
		%	81.8	83	86.1	83.9	83.4

Table 6. Algorithm 1, F_1 objective function, all overlaps.

		Problem				
Fragments		22,448	89,718	298,194	730,201	2,278,504
Contigs Obtained		82	114	163	224	510
Total Number of Bases		19,343	84,659	240,314	552,408	1,321,969
Contigs Found	**Contigs Found**	74	101	149	204	480
	%	90.2	88.6	91.4	91.1	94.1
	Bases Found	15,822	70,250	206,826	463,582	1,102,152
	%	81.8	83	86.1	83.9	83.4
	N50	360	1614	3326	4937	4499
	Average Length	213.8	695.5	1388.10	2272.50	2296.20
	Minimum Length	54	58	54	62	53
	Maximum Length	882	7261	10,943	22,745	24,396
Contigs not Found	**Contigs not Found**	8	13	14	20	30
	%	9.8	11.4	8.6	8.9	5.9
	Bases not Found	3521	14,409	33,488	88,826	219,817
	%	18.2	17	13.9	16.1	16.6
	N50	758	2308	3954	7258	11,798
	Average Length	440.1	1108.40	2392.00	4441.30	7327.20
	Minimum Length	95	95	95	181	255
	Maximum Length	826	3703	6809	9542	23,614

Table 7. Algorithm 1, F_2 objective function, all overlaps.

				Problem		
Fragments		22,448	89,718	298,194	730,201	2,278,504
Contigs Obtained		82	114	163	224	510
Total Number of Bases		19,028	84,584	240,280	552,509	1,331,757
Contigs Found	**Contigs Found**	74	102	146	189	450
	%	90.2	89.5	89.6	84.4	88.2
	Bases Found	15,515	70,843	189,927	369,001	931,343
	%	81.5	83.8	79	66.8	69.9
	N50	356	1527	3184	4298	4201
	Average Length	209.7	694.5	1300.90	1952.40	2069.70
	Minimum Length	54	58	54	59	53
	Maximum Length	882	7261	10,943	21,185	16,135

Contigs not Found	**Contigs not Found**	8	12	17	35	60
	%	9.8	10.5	10.4	15.6	11.8
	Bases not Found	3513	13,741	50,353	183,508	400,414
	%	18.5	16.2	21	33.2	30.1
	N50	758	2308	6336	7620	9472
	Average Length	439.1	1145.10	2961.90	5243.10	6673.60
	Minimum Length	95	96	103	181	251
	Maximum Length	826	3703	7626	22,745	24,396

From the DNA fragment assembly point of view, we are interested in large *contigs* with no errors. The most important elements to be computed are the number of bases and the *contigs* that were found in the complete genome, as well as the mean length of each *contig*. When we compare the results obtained by Algorithm 1 for F_1 and F_2 using all overlaps, we find that F_1 is a little bit better than F_2, except in the number of bases found for problem 89,718. In terms of multi objective function optimization, the solution obtained for F_1 is not enough to dominate the solution obtained by F_2, but since it is better in most of the test, we can say that it is almost dominant. In the other experiments, we find a similar situation, where F_1 is almost dominant with respect to F_2.

For Experiment 2, we excluded transitive overlaps and used Algorithm 1 for F_1 and F_2. The results are given in Tables 5, 8 and 9.

Table 8. Algorithm 1, F_1 objective function, no transitive overlaps.

		Problem				
Fragments		22,448	89,718	298,194	730,201	2,278,504
Contigs Obtained		82	114	163	224	512
Total Number of Bases		18,879	84,646	240,314	551,600	1,333,881
Contigs Found	**Contigs Found**	74	102	149	204	481
	%	90.2	89.5	91.4	91.1	93.9
	Bases Found	15,806	71,652	206,826	462,774	1,103,587
	%	83.7	84.6	86.1	83.9	82.7
	N50	360	1527	3326	4937	4499
	Average Length	213.6	702.5	1388.10	2268.50	2294.40
	Minimum Length	54	58	54	62	53
	Maximum Length	882	7261	10,943	22,745	24,396
Contigs not Found	**Contigs not Found**	8	12	14	20	31
	%	9.8	10.5	8.6	8.9	6.1
	Bases not Found	3073	12,994	33,488	88,826	230,294
	%	16.3	15.4	13.9	16.1	17.3
	N50	539	2308	3954	7258	11,657
	Average Length	384.1	1082.80	2392.00	4441.30	7428.80
	Minimum Length	95	95	95	181	255

	Maximum Length	826	3703	6809	9542	23,614

In general, we found that F_1 is better than F_2 without dominating it. Comparing the results where transitive overlaps are excluded (Tables 8 and 9 or Table 5) to those where they are not (Tables 6 and 7 or Table 5), we can see that the results are similar for F_1, hence it is not clear whether it is better to exclude or not to exclude transitive overlaps. There is, however, an obvious advantage of excluding transitive overlaps, which is that the problem size is reduced and requires less computational resources. So, our recommendation is to exclude transitive overlaps.

Comparing our results so far to Edena, we find out that we obtained fewer correct bases and more *contigs* that are not in the genome, but the length of our *contigs* is better.

Table 9. Algorithm 1, F_2 objective function, no transitive overlaps.

		Problem				
	Fragments	22,448	89,718	298,194	730,201	2,278,504
	Contigs Obtained	82	114	163	224	512
	Total Number of Bases	18,579	84,571	240,265	551,701	1,343,637
Contigs Found	**Contigs Found**	75	103	142	181	417
	%	91.5	90.4	87.1	80.8	81.4
	Bases Found	15,599	72,245	174,244	320,502	791,376
	%	84	85.4	72.5	58.1	58.9
	N50	356	1527	2984	3910	4135
	Average Length	208	701.4	1227.10	1770.70	1897.80
	Minimum Length	54	58	54	59	53
	Maximum Length	882	7,261	10,943	10,648	16,135
Contigs not Found	**Contigs not Found**	7	11	21	43	95
	%	8.5	9.6	12.9	19.2	18.6
	Bases not Found	2980	12,326	66,021	231,199	552,261
	%	16	14.6	27.5	41.9	41.1
	N50	539	2308	6336	8135	7554
	Average Length	425.7	1120.50	3143.90	5376.70	5813.30
	Minimum Length	95	96	103	181	251
	Maximum Length	826	3703	10,648	22,745	24,396

Table 10. Algorithm 1, F_1 objective function, Minimum Spanning Tree (MST).

	Problem				
Fragments	22,448	89,718	298,194	730,201	2,278,504
Contigs Obtained	334	1883	6080	15,618	48,263
Total Number of Bases	40,076	265,584	831,449	2,109,070	6,613,747

Contigs Found	Contigs Found	304	1827	5980	15,443	46,187
	%	91	97	98.4	98.9	95.7
	Bases Found	35,980	256,641	815,994	2,081,306	6,208,258
	%	89.8	96.6	98.1	98.7	93.9
	N50	135	165	154	153	153
	Average Length	118.4	140.5	136.5	134.8	134.4
	Minimum Length	62	62	61	57	57
	Maximum Length	446	660	747	894	1152
Contigs not Fund	Contigs not Found	30	56	100	175	2076
	%	9	3	1.6	1.1	4.3
	Bases not Found	4096	8943	15,455	27,764	405,489
	%	10.2	3.4	1.9	1.3	6.1
	N50	154	199	198	194	245
	Average Length	136.5	159.7	154.6	158.7	195.3
	Minimum Length	64	65	65	64	63
	Maximum Length	362	481	481	493	1130

In Experiment 3, we obtained the Minimum Spanning Tree (MST) from all overlaps using our version of Kruskal's algorithm. We only show the results for F_1 (Tables 5 and 10). In this case, we were able to find much more bases in the genome than in previous experiments, including Edena, but the size of our *contigs* is small. There appears to be a tradeoff between the number of bases that can be found and *contig* length.

In other experiments, we used Algorithm 4 without transitive overlaps, obtaining similar results to those of Algorithm 1 for F_1 (F_1 was again better than F_2). Analyzing the *contigs* obtained by the algorithm, we found that if the *contig* is split at the edge that joins the cycle with the path, we have a better chance of finding both *contigs*. Therefore, we modified Algorithm 4 making d(v) = 0 in

Step 2.2.1.1 and repeated Experiment 4 without transitive overlaps. In this case, the objective function F_1 was better than F_2 without being dominant. These results are given in Tables 5 and 11.

Table 11. Algorithm 5, F_1 objective function, no transitive edges.

		Problem				
Fragments		22,448	89,718	298,194	730,201	2,278,504
Contigs Obtained		82	115	164	225	510
Total Number of Bases		19,343	84,727	240,415	552,510	1,321,969
Contigs Found	Contigs Found	74	102	150	205	480
	%	90.2	88.7	91.5	91.1	94.1
	Bases Found	15,822	70,318	206,927	463,684	1,102,152
	%	81.8	83	86.1	83.9	83.4
	N50	360	1614	3326	4937	4499
	Average Length	213.8	689.4	1379.50	2261.90	2296.20

	Minimum Length	54	58	54	62	53
	Maximum Length	882	7261	10,943	22,745	24,369
Contigs not Found	**Contigs not Fund**	8	13	14	20	30
	%	9.8	11.3	8.5	8.9	5.9
	Bases not Found	3521	14,409	33,488	88,826	219,817
	%	18.2	17	13.9	16.1	
	N50	758	2308	3954	7258	11,798
	Average Length	440.1	1108.40	2392.00	4441.30	7327.20
	Minimum Length	95	95	95	181	255
	Maximum Length	826	3703	6809	9542	23,614

In an analysis of those *contigs* that were not found in the genome, we considered the possibility of splitting them in two or more pieces hoping that all of them could be found. There are few cases in which *contigs* should be divided; in problem 730,201, using the modified Algorithm 4 with F_1, there were 513 *contigs* from which 414 were found directly, 76 were split in two to be found, 14 in three, 10 in four or more pieces, and two produced pieces that were too small to be considered. With appropriate rules, it should be possible to find the correct split points in most of the cases, as other assemblers do. We also analyzed residual paths, after removing paths of maximum length, from which we considered the possibility of recovering several long *contings*, which we will do in the future.

5. Conclusions and Future Work

In order to assemble DNA, it is possible to use traditional graph theory concepts being careful to use only linear algorithms due to the large amount of data that is handled. With this idea, we developed a constant time heap, appropriate for this particular case (and maybe to other cases as well), which allowed us to reduce the time complexity of other algorithms, such as Prim's and Kruskal's, making them linear.

If we compare the results of the objective functions F_1 and F_2, even though one does not dominate over the other, better results are obtained using F_1, hence we recommend its use. We also recommend the removal of transitive overlaps because of resource reduction considerations, even though the results of the assembly are almost the same for function F_1. The use of the MST considerably improves the number of correct bases, but the *contigs* obtained in this way are too small. It can be seen from the experiments that there is a tradeoff between *contig* length and the number of bases detected. Comparing our results to those of Edena is not completely objective since we do not apply heuristics after the algorithms are executed in order to refine the solutions, but Edena (as well as the Velvet assembler) do so intensively. In any case, compared to Edena, our method produced, in some cases, less bases but longer *contigs*, while, in other cases, we found more bases, but shorter *contigs*, especially when using MST. Therefore, the problem must be considered as a multi objective optimization, with an objective function that is able to maximize the number of mean *contig* length, giving the user a Pareto Front from which he can obtain the most convenient solution according to his particular criteria.

Our future work includes:

- Developing rules to split *contigs* in such a way that most of the pieces can be found.
- Attempting to recover other *contigs* after extracting the longest ones.

- Implementing a solution to DNA fragment assembly as a multi objective optimization problem.

Acknowledgments

This work has been developed with the support of Universidad Iberoamericana.

Author Contributions

Guillermo Mallén-Fullerton started the idea and developed the algorithms. Emilio Quiroz-Ibarra developed test programs and provided helpful observations about the implementation problems associated with some of the algorithms. Antonio Miranda and Guillermo Fernández-Anaya provided algorithms complexity analysis and observations to improve it. Guillermo Fernández-Anaya collaborated in the experiments design and interpretation and Antonio Miranda wrote the English version of the article. All authors approved the manuscript.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Watson, J.D.; Crick, F.H. Molecular structure of nucleic acids. *Nature* **1953**, *171*, 737–738.
2. Sanger, F.; Nicklen, S.; Coulson, A.R. DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci.* **1977**, *74*, 5463–5467.
3. Staden, R. A strategy of DNA sequencing employing computer programs. *Nucl. Acid. Res.* **1979**, *6*, 2601–2610.
4. Van Belkum, A.; Scherer, S.; van Alphen, L.; Verbrugh, H. Short-sequence DNA repeats in prokaryotic genomes. *Microbiol. Mol. Biol. Rev.* **1998**, *62*, 275–293.
5. Salzberg, S.L.; Phillippy, A.M.; Zimin, A.; Puiu, D.; Magoc, T.; Koren, S.; Yorke, J.A. GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Gen. Res.* **2012**, *22*, 557–567.
6. Shendure, J.; Ji, H. Next-generation DNA sequencing. *Nat. Biotechnol.* **2008**, *26*, 1135–1145.
7. Pevzner, P.A.; Tang, H.; Waterman, M.S. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci.* **2001**, *98*, 9748–9753.
8. Parsons, R.J.; Forrest, S.; Burks, C. Genetic algorithms for DNA sequence assembly. In Proceedings of the First International Conference on Intelligent Systems for Molecular Biology (ISMB), Bethesda, MD, USA, 6–9 July 1993; pp. 310–318.
9. Krause, J.; Cordeiro, J.; Parpinelli, R.S.; Lopes, H.S. A Survey of Swarm Algorithms Applied to Discrete Optimization Problems. In *Swarm Intelligence and Bio-inspired Computation: Theory and Applications*; Elsevier Science Publishers: Amsterdam, The Netherlands, 2013.
10. Alba, E.; Luque, G. A new local search algorithm for the DNA fragment assembly problem. In *Evolutionary Computation in Combinatorial Optimization*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 1–12.
11. Luque, G.; Alba, E. Metaheuristics for the DNA fragment assembly problem. *Int. J. Comput. Intel. Res.* **2005**, *1*, 98–108.

12. Firoz, J.S.; Rahman, M.S.; Saha, T.K. Bee algorithms for solving DNA fragment assembly problem with noisy and noiseless data. In Proceedings of the 14th ACM Annual Conference on Genetic and Evolutionary Computation, Philadelphia, PA, USA, 7–11 July 2012; pp. 201–208.
13. Mallen-Fullerton, G.M.; Fernandez-Anaya, G. DNA fragment assembly using optimization. In Proceedings of the 2013 IEEE Congress on Evolutionary Computation (CEC), Cancun, Mexico, 20–23 June 2013; pp. 1570–1577.
14. Gallant, J.; Maier, D.; Astorer, J. On finding minimal length superstrings. *J. Comput. Syst. Sci.* **1980**, *20*, 50–58.
15. Prim, R.C. Shortest connection networks and some generalizations. *Bell Syst. Tech. J.* **1957**, *36*, 1389–1401.
16. Kruskal, J.B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc.* **1956**, *7*, 48–50.
17. Hopcroft, J.E.; Ullman, J.D. Set merging algorithms. *SIAM J. Comput.* **1973**, *2*, 294–303.
18. Bonfield, J.K.; Smith, K.; Staden, R. A new DNA sequence assembly program. *Nucl. Acid. Res.* **1995**, *23*, 4992–4999.
19. Mallén-Fullerton, G.M.; Hughes, J.A.; Houghten, S.; Fernández-Anaya, G. Benchmark datasets for the DNA fragment assembly problem. *Int. J. Bio-Inspir. Comput.* **2013**, *5*, 384–394.
20. Staphylococcus *aureus* subsp. aureus MW2 DNA, complete genome, GenBank: BA000033.2. Available online: http://www.ncbi.nlm.nih.gov/nuccore/47118312?report=fasta (accessed on 3 June 2015).

© 2015 by the authors; licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/4.0/).

8.9 Publicación 2: DNA Paired Fragment Assembly Using Graph Theory

Article

DNA Paired Fragment Assembly Using Graph Theory

J. Emilio Quiroz-Ibarra 1,*, Guillermo M. Mallén-Fullerton 2 and Guillermo Fernández-Anaya 3

[1] Physics and Mathematics, Universidad Iberoamericana Ciudad de México, Prolongación Paseo de la Reforma 880, Lomas de Santa Fe, Ciudad de México 01219, Mexico

[2] Engineering Department, Universidad Iberoamericana Ciudad de México, Prolongación Paseo de la Reforma 880, Lomas de Santa Fe, Ciudad de México 01219, Mexico; guillermo.mallen@ibero.mx

[3] Physics and Mathematics Department, Universidad Iberoamericana Ciudad de México, Prolongación Paseo de la Reforma 880, Lomas de Santa Fe, Ciudad de México 01219, Mexico; guillermo.fernandez@ibero.mx

* Correspondence: quirozem@yahoo.com.mx; Tel.: +52-1-55-1393-8302

Academic Editor: Andras Farago
Received: 26 January 2017; Accepted: 17 March 2017; Published: 24 March 2017

Abstract: DNA fragment assembly requirements have generated an important computational problem created by their structure and the volume of data. Therefore, it is important to develop algorithms able to produce high-quality information that use computer resources efficiently. Such an algorithm, using graph theory, is introduced in the present article. We first determine the overlaps between DNA fragments, obtaining the edges of a directed graph; with this information, the next step is to construct an adjacency list with some particularities. Using the adjacency list, it is possible to obtain the DNA *contigs* (group of assembled fragments building a contiguous element) using graph theory. We performed a set of experiments on real DNA data and compared our results to those obtained with common assemblers (*Edena* and *Velvet*). Finally, we searched the *contigs* in the original genome, in our results and in those of *Edena* and *Velvet*.

Keywords: DNA fragment assembly; Trie data structure; graph theory algorithms

---

1. Introduction

Each monomer comprising the DNA polymer is formed with a pentose, a phosphate group and one of four nitrogenous bases: adenine, guanine, cytosine and thymine. In 1953, scientists James

Watson and Francis Crick [1] discovered the double-helix spatial structure of the DNA molecule. This double chain is coiled around a single axis, and the strands are attached by hydrogen bridges between pairs of opposite bases. The direction of the polymer chain is determined by the pentose carbon atoms 5^0 and 3^0 (the beginning and ending positions of a DNA strand are denoted as 5^0 and 3^0, respectively; a DNA strand is read in the 5^0 to 3^0 direction, and the complementary strand runs in the opposite direction). The bases in each pair are complementary. The content of adenine (**A**) is the same as the content of thymine (**T**), and the cytosine (**C**) content is the same as that of guanine (**G**) (Figure 1).

In 1975, Frederik Sanger [2] proposed a DNA sequencing technique that involved detecting small dark bands in a thin gel using electrophoresis. Sanger proposed cutting the DNA molecule at specific points in the sequence with restriction enzymes [3]. This method is slow and costly; with

each digestion, the sample must be divided, and each new division must be cloned to obtain a sufficient amount of material. To reduce processing time, Sanger et al. [4,5] proposed splitting the DNA sequences at random points. The disadvantage of this method is that the order of the fragments is unknown, generating an NP-Complete (Non-deterministic polynomial time) [6] computational problem. This method is known as the *shotgun technique*.

Algorithms **2017**, *10*, 36; doi:10.3390/a10020036 www.mdpi.com/journal/algorithms

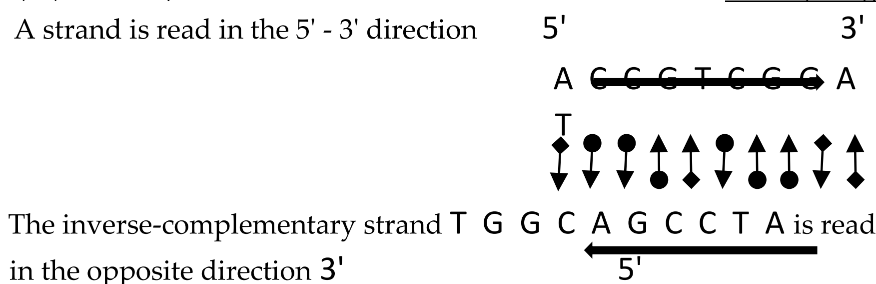


Figure 1. Reading a strand in direct and inverse-complementary.

Rodger Staden [7] proposed a method to assemble the genome using a computer. As the DNA fragments are produced from many copies of the original genome, more than one fragment comes from the same region. The DNA fragments should be processed while looking for overlaps, coincidences in the extremes of the fragments. The total number of bases in the fragments divided by the total number of bases in the complete genome is called the *coverage*. If the *coverage* is high enough, it is possible to rebuild the genome, but it is difficult to obtain high-quality results because false overlaps can be generated due to sequencing errors, sample contamination with foreign DNA and chimeras (the cloning process is carried out using host bacteria, and sometimes the DNA of the sample is concatenated with that of the host bacteria, producing what is known as a *chimera*). In our experiments,

we noticed that some DNA sections might not be sampled and that complete genome reconstruction would therefore not be possible. What can be obtained is a set of *contigs* covering most of the genome, and it is the job of a molecular biologist to assemble the *contigs* using other techniques.

Later, James Weber and Eugene Myers [4] proposed the generation of paired fragments in the *shotgun* process. Traditional sequencing starts from the 5^0 end of each piece of DNA, and only a limited number of bases can be sequenced. In NGS (Next-Generation Sequencing), the number of bases is always the same, yielding fixed-size fragments; however, the DNA pieces sequenced are generally longer. The goal of the paired fragments procedure is to obtain a sequence from the 3^0 end as well as one from the 5^0 end, obtaining two fragments from the same piece of DNA. This would help to establish an interval in the DNA sequence associated with the paired fragments. Unfortunately, sequencing from the 3^0 end is difficult and produces a relatively large number of sequencing errors.

DNA sequencing technology has advanced, decreasing costs and process time. Today, there are databases with information about many genomes, including the human genome and those of many disease-causing microorganisms [8]. Steven Salzberg [9] developed a comparative study of DNA sequence assemblers; tests were carried out with fragment sets obtained using Illumina equipment (*Illumina*: http://www.illumina.com/, Illumina, Inc., San Diego, CA, USA). The fragments' lengths were found by NGS technology to be in the range of 50 to 150 bases [10]. The Salzberg study was developed in 2012; nevertheless, the paradigm remains the same today, and

the Salzberg study is therefore still applicable. The primary metric in the evaluation was N50 (shortest sequence length close to the median of a set of *contigs*). Salzberg employed four organisms in the test, including a *Staphylococcus aureus* problem. These results are particularly interesting because the tests we present in this article have been obtained using a problem with the same bacterium. Salzberg did not include the *Edena assembler* (http://www.genomic.ch/edena.php, Genomic Research Institute, Geneva, Switzerland.), developed by David Hernández [11] in his research. In our comparative study, this assembler is an important reference.

Initially, *greedy* (Algorithmic technique that at each step tries to generate the optimal solution of that step of the problem without considering the rest of the problem) algorithms [12] were used to find the order of the fragments; later, the *de Bruijn* graph [13] was introduced with different values for the *k-mer* (section of k consecutive bases) [13]. Most of the assemblers available today are based on *de Bruijn* graphs. In our comparative study, we include the *Velvet assembler* (http://www.ebi.ac.uk/~zerbino/velvet/, EML-EBI, Cambridge, UK), developed by Zerbino [14], which applies *de Bruijn* graphs.

Parsons et al. [15] proposed an optimization with a genetic algorithm to maximize the sum of the fragment overlaps, but the obtained *contigs* were relatively short and processing speed was low. Later, Mallén-Fullerton and Fernández-Anaya [16] suggested a reduction of the fragments' assembly problem to the traveling salesman problem (TSP) that has been studied extensively. Solution methods with relatively good efficiency exist for the TSP. Applying heuristics and algorithms, they obtained optimal solutions for several commonly used benchmarks, and for the first time, a real-world problem

was solved by using optimization. Using graph theory and setting the appropriate objective functions, Mallén et al. [17] developed a new assembly method from the perspective of a directed graph, looking for the reduction of the algorithm's complexity.

In this publication, taking Mallén et al. [17] as a starting point, we developed a new algorithm working with the *paired fragments* (two records identified by "/1" and "/2", respectively, conforming to an interval of the sequence of DNA) resulting from the sequencing results of the *Illumina equipment.* In our initial tests, we found some *contigs* that were not located in the published genome for the same organism, even though these *contigs* were properly obtained. Empirically, we found that when these *contigs* were split at certain points, all of the pieces could, in most cases, be found in the genome. Using the information from the paired fragments information, we could find the locations where a *contig* should be split to increase the quality of the assembly.

To obtain the overlaps between DNA fragments, we used a *Trie* [17]. Using the overlaps as edges, a directed graph has been obtained, and we could build a set of *contigs* improving the N50 [8] of the previous release [17]. The interval of the paired fragments was a critical factor in our success. In our experiments, we worked with the sequencing data of real-life organisms obtained from *Illumina equipment*, maximizing the lengths of the *contigs* as the objective function.

In Section 2 of this paper, we present the applied graph theory elements, data structures and algorithms. In Section 3, we reveal the developed algorithms that solve the assembly problem. Section 4 sets out the application of this new model to a real-life problem comparing our assembler with other assemblers (*Velvet* and *Edena*), and finally, in Section 5, we present our conclusions and ideas for future work.

2. Graph Theory in the DNA Fragment Assembly Problem

2.1. Generalities

A graph $G = (V, E)$ is a set of vertices V and edges E; the vertices are linked by the edges, and, on each side of the edge, only one vertex exists. Some graphs are non-directional; nevertheless, if the system requires a direction, it will activate an ordered paired of vertices, and the result will be called a directed graph. One paradigm for DNA fragment assembly using overlapping fragments is based on a graph. Several models have been developed, and these have had variable outcomes with respect to the algorithm's complexity and efficiency and the quality of the results in the assembled DNA. The algorithms based on graph theory include *de Bruijn* graphs [13], Eulerian paths [10], Hamiltonian paths [10] and Depth-First search (DFS) [17].

2.2. The Shotgun Technique

In the sequencing process, a sample of DNA is broken into many fragments [10]. Next-Generation Sequencing (NGS) produces very short fragments, all the same size, usually between 25 and 500 bases. The cuts are made in random places, usually producing millions of fragments. Each fragment is sequenced, and the results are stored in a FASTA (plain-text file format used to represent and store genetic information) file format [18]. Figure 2 illustrates the *shotgun technique* [10]. Each fragment overlaps with other fragments; the fragments are the graph vertices, and the number of overlapped bases are the edge weights. In Figure 3, a graph model example is shown.

```
Sequenced info:          TTCACTTATTTAAAATCTGGAAGAAACCTAGG
fragment 1 left cut:     TTCACTTATTTAAAATCTGGAAGA
fragment 2 right cut:         TTTAAAATCTGGAAGAAACCTAGG
Overlap assembly:        TTCACTTATTTAAAATCTGGAAGAAACCTAGG
                              |  <= 16 overlaps =>  |
```

Figure 2. Shows the *shotgun* process for a DNA sequence and the assembly with overlaps.

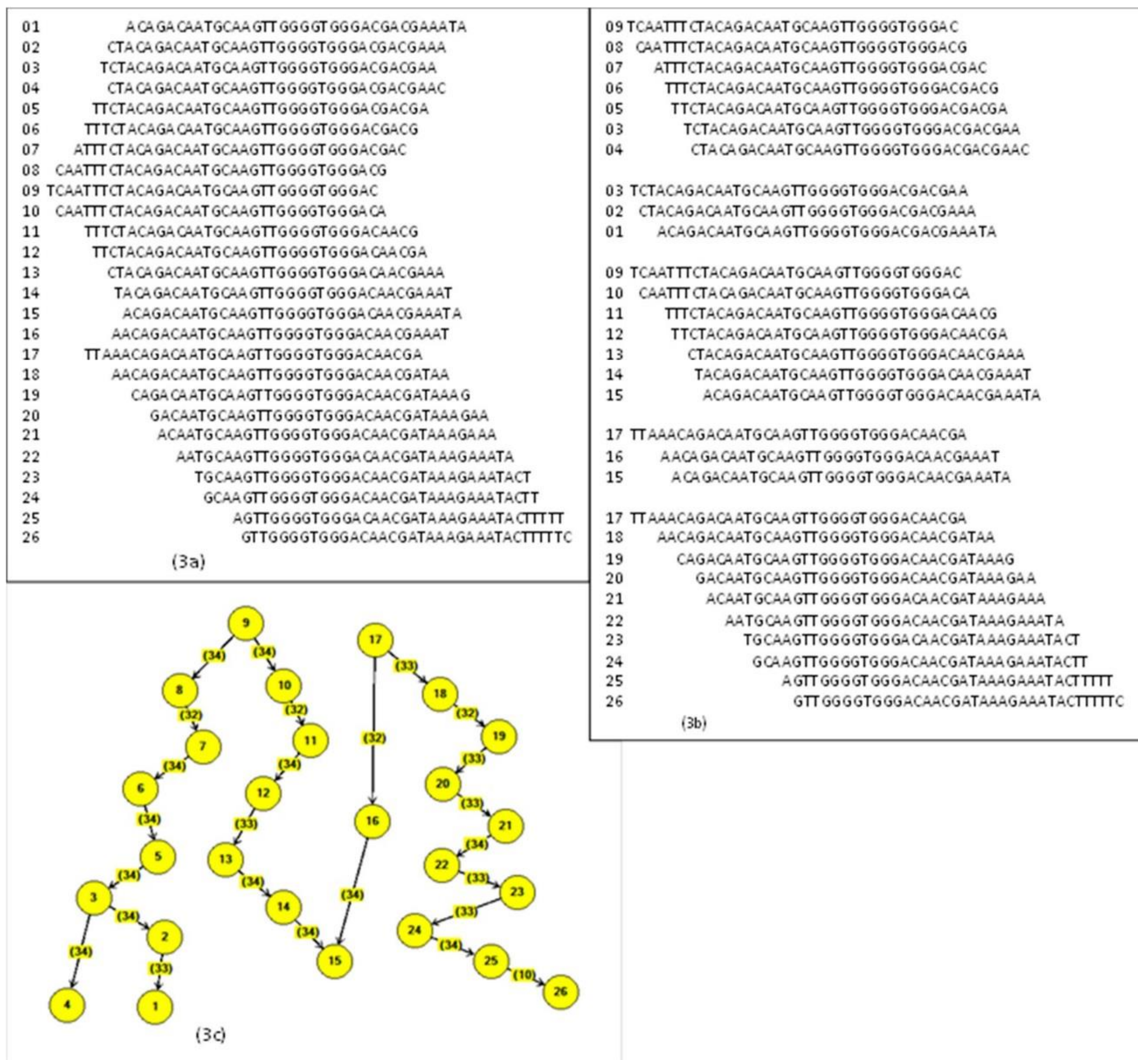


Figure 3. Directed graph from a group of fragments. (**3a**) Semi assembled fragments; (**3b**) Overlapped fragments; (**3c**) Resulting graph from the overlap analysis.

2.3. Pair Generation

The overlaps between fragments are obtained using a *Trie* data structure [19]. With this data structure, it is possible to calculate only the real overlaps without checking all of the possible pairs or the overlaps that are smaller than a predefined length. In the worst-case scenario, the search complexity of the *Trie* is $O(l)$, where l is the fragment length. If n is the number of fragments in the problem, the complexity to obtain all the overlaps is $O(nl)$. In addition, the *Trie* has the advantage of storing no duplicate data [20].

2.4. Adjacency List

A proper way to manage sparse graphs, as was applied in our case, is the *adjacency list* data structure [21]. We found the use of a linked list appropriate for our problem, with a header section as a dictionary. Each header value is a starting point for the adjacency elements stored in the same data structure, which really contains a set of *adjacency lists*. The input and output degrees for each vertex are calculated while the overlaps are loaded. Each overlap has a preceding and a subsequent fragment that determine the direction of the edges of the directed graph. The vertices with zero input degree value are the starting point for a path. The path with a larger sum of weights is a *contig* in our algorithms.

Because of the fragment overlaps, each base in a *contig* can appear in more than one fragment, as shown in Figure 4. The *consensus* is the number of times that a nucleotide appears in the same

position of a *contig*. We considered a minimum *consensus* to accept a base in a *contig*. The base found most frequently in each position of a *contig* is accepted if its frequency is over the minimum *consensus* previously defined. In the example presented in Figure 4, a minimum *consensus* of four is accepted.

```
   T T C A C T T A T T T A A A A T C T G G A A G A A A C C T A G
       C A C T T A T T T A A A A T C T G G A A G A A A C C T A G G C C T G A A
             T T A T T T A A A A T C T G G A A G A A A C C T A G G C C T G A A C G G T
                   T T T A A A A T C T G G A A G A A A C C T A G G C C T G A A C G G T A A
                         A A A A T C T G G A A G A A A C C T A G G C C T G A A C G G T A A T T G
                           A A A T C T G G A A G A A A C C T A G G C C T G A A C G G T A A T T G G G T T T
A:       2       3       5 6 6 6           6 6   6 6 6       6             5 5         3 3
C:     2   2                       6                   6 6         5 5         4
G:                                     6 6     6               6 5       5       4 4           2 1 1
T: 1 1       3 3   4 4 4         6   6                     6           5             4     2 2       1 1 1
                   T T T A A A A T C T G G A A G A A A C C T A G G C C T G A A C G G T
   | < - - - - > | < - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - > | < - - - - - - > |
   bases with low      bases with good consensus value                                  bases with low
   consensus     |                                                                    | consensus value
```

Figure 4. *Consensus* effect in the *contig* assembly.

2.5. Specific Characteristics

Our objective is to obtain the paths that maximize the sum from the weights of the edges contained in the path [17]. Each path starts at a node with zero-input degree and ends at a node with zero-output degree. Notice that several starting nodes can connect to a shared group of nodes. In this case, we keep only the path with the maximum edge sum. From the selected paths, it is possible to assemble a set of *contigs*.

In the assembly process for the *contig*, the *consensus* value [17] must be selected. In the experiments presented in this article, we used a *consensus* of 4. Bases with a *consensus* lower than the specified value are discarded. A *consensus* value greater than or equal to the specified value indicates a good-quality base, as shown in Figure 4. Usually, the lowest *consensus* values are found in the extremes of the *contig*.

To verify the quality of the *contigs*, we searched for them in the original genome. Those not located were separated into several segments, searching for each one in the genome. Some of these fragments were located, while a few were not, indicating that the prediction of the split point was not always accurate. The existence of paired fragments in the *contig* is a guarantee that the content of the interval is correct, and this gave us the criterion for the split point with respect to the interval of the paired fragments.

Figure 5 shows two intervals; the split point of the *contig* is outside the intervals. All of the elements outside of the intervals can be discarded because there is no confirmation with the paired fragments (“/1”, “/2”). It should be noted that the N50 [9] value will be reduced; meanwhile, the new *contigs* are of better quality.

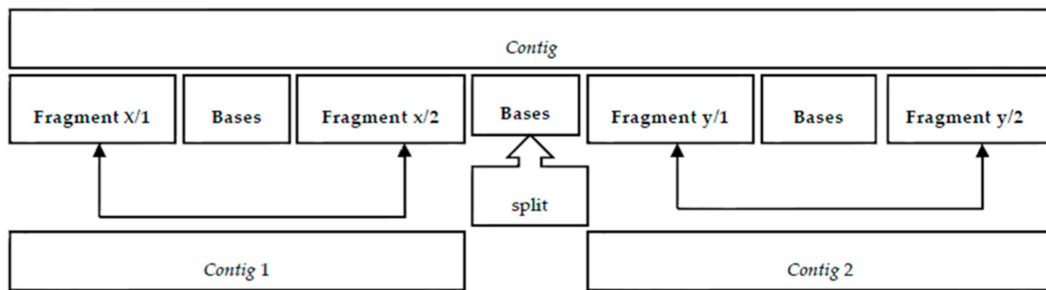


Figure 5. Assembly ensured with the intervals of paired fragments.

3. Algorithms

To resolve the assembly problem, we hereby propose the following sequence (Figure 6):

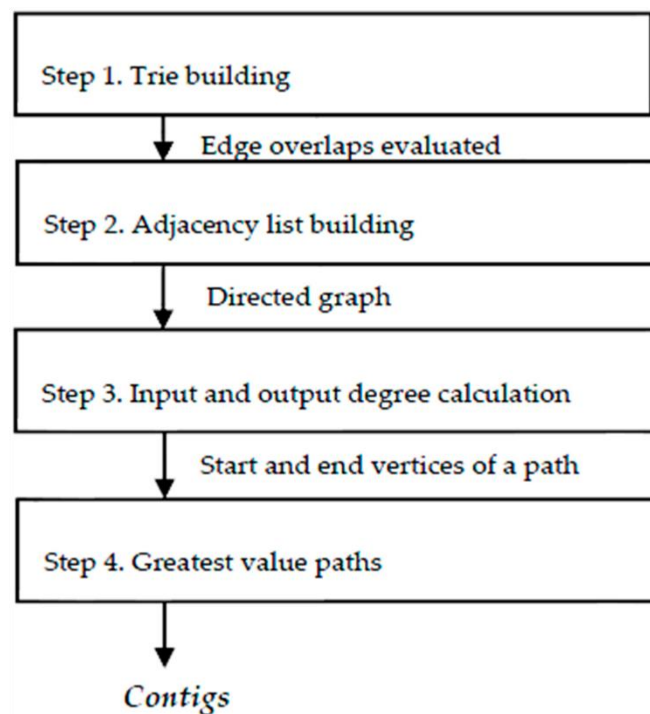


Figure 6. *Contigs* build sequence.

Step 1. Build a *Trie* (prefix tree) [19] to identify the overlapped fragments, trying to get the greatest value from the overlapping while resolving duplicates, fragments without overlap and repetition bases with the same element.

Step 2. From the list of overlapping fragments that comprise a directed graph, an adjacency list is built [21], with which we can calculate the input and output degrees of each vertex.

Step 3. Using the degree input and output information of the vertices, we could detect those that are a head of a path, with zero input degree, and build the path to those with zero output degree, accumulating the overlapping values of each edge. This accumulated value can change if, when on a new path, there is a value greater than the previous intersection of an intermediate vertex of the path.

Step 4. Finally, walking in reverse, beginning with the last vertex, rebuild the route marked with the greatest overlapping values until a vertex of zero input degree is reached. This path is a *contig*.

3.1. Trie

The *Trie* data structure was developed by Fredking [22]. It is a tree structure that forms prefix texts that are converted into a search index for the new texts. The new texts fit with the prefix and complement the branch of the tree with a suffix. In our problem, the prefixes and suffixes are letters from the alphabet: A, C, G, T.

In the building process, the first node with four locations is empty and ready to receive the four elements A, C, G, T. The first fragment arrives and accommodates depending on the letter value. For example, the first string is AGGTCGA, and it goes on creating new blank nodes. The next one arrives with AGGTTTC, and it accommodates from AGGT; the next fragment is AGGCCTC, and now it accommodates from AGG. Figure 7 shows the resulting tree.

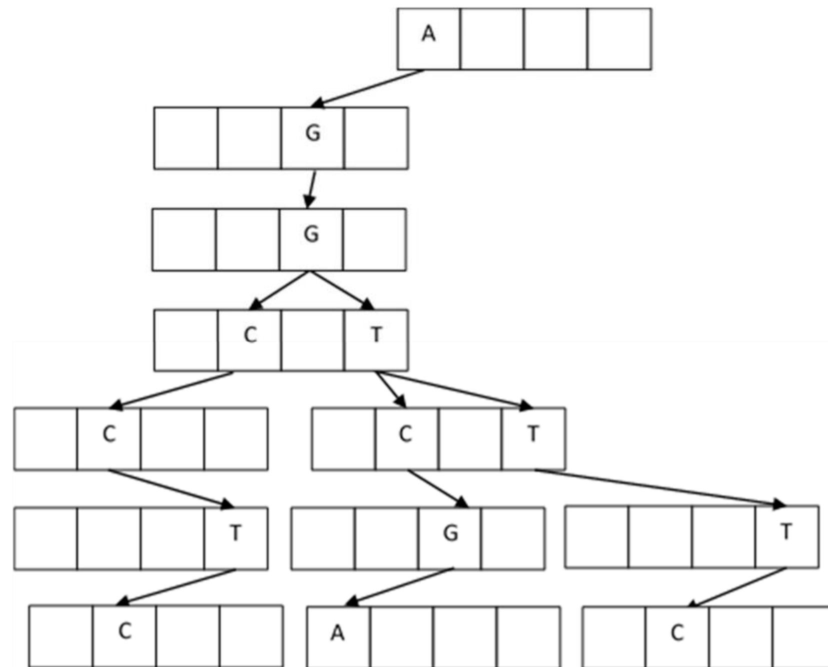


Figure 7. Inserted sequences in the Trie.

The duplicate fragments are easily handled because they do not provide new values to the branches and are eliminated. The construction of the *Trie* is, in the worst case, *O(ln)* [19] when there are no coincidences between the fragments (*l* is the fragment length and *n* is the number of fragments). In our data problem, the number of coincidences is large. Algorithm 1 describes the process of building the *Trie*.

Algorithm 1. Trie construction.

---

1. *For each fragment*
   - 1.1 *For each letter x from the fragment*
     - 1.1.1 *If box node(x) is free:*
       - *1.1.1.1 Take up the box and create new empty node*
     - 1.1.2 *Else*
       - *1.1.2.1 Go to next node*

Algorithm 2 shows the search for the overlapped fragments. In the worst case, it is *O(l)* for a single fragment, where *l* is the length of the fragment, because, at that point, it has completely walked through the branch of the tree.

Algorithm 2. Searching a fragment in the Trie.

1. *For each fragment (l)*
   1.1 *For each fragment (l ← l-1 until overlap limit value)*
      1.1.1 *For each fragment-letter vs. Trie-node-letter*
         1.1.1.a *If equal: continue to next fragment-letter and node*
         1.1.1.b *If empty box: finish cycle 1.1 1.2 If end of fragment:*
      1.2.1 *Drain the branch*
      1.2.2 *Identify both fragments and create an edge (from, to, overlap)*

3.2. Adjacency List

This data structure is built by taking the list of edges. The list is a conventional data structure for each fragment.

3.3. Contig Assembly

In this step, the longest path will be sought, starting with all the vertices with zero input degree, these being the potential starting points of a path that could eventually become a *contig*. The walkthrough is carried out until a vertex with zero output degree is found, while accumulating the overlapping values. Once a path is finished, the walkthrough shifts to the next vertex with zero input degree and the process is repeated. If a node that has been used in another path is detected, the process will evaluate the greatest value and will leave the greater value as the result. Once the complete graph has been processed, each starting node is a *contig*. Algorithm 3 shows the process used to obtain the greatest weight of a path.

Algorithm 3. *Contigs* assembly.

1. *For all elements with Din = 0*
   1.1 *Dout minus 1*
   1.2 *For each adjacency element until Dout = 0*
      1.2.1 *Accumulate weight*
   1.3 *If accumulated weight > previous weight*
      1.3.1 *Label maximum path*

While assembling the branch, a content counter is generated for the column; at the end of the assembly process, the consensus is reviewed with the counter, and it is possible to remove the sections not in compliance with the predefined value.

In the FASTA file containing the original fragments [18], the paired values are identified as "/1" and "/2"; these identifiers will be used to confirm the *contig* by searching the interval inside the *contig*. Algorithm 4 describes how the paired fragments confirm intervals.

Algorithm 4. Confirm intervals.

1. Interval-counter ← 0 (counts the intervals into a contig).
2. For each fragment in the contig

2.1 If type "/1", search type "/2" in the contig

2.1.1 If Found type "/2" in the contig

2.1.1.1 Yes: interval counter = interval counter+1

3. For each fragment in the contig

3.1 If interval counter = 0

3.1.1 If fragment in in-between, cut of the contig
3.1.2 If fragment in extremes, drop the section

Figure 8a shows a case of a *contig* that demonstrates continuity between the first confirmation interval and the subsequent interval. If there is no continuity between the confirmation intervals, then there is a split point, as shown in Figure 8b. This split point will produce two *contigs*. The elements at the extremes are also removed because there is no confirmation interval.

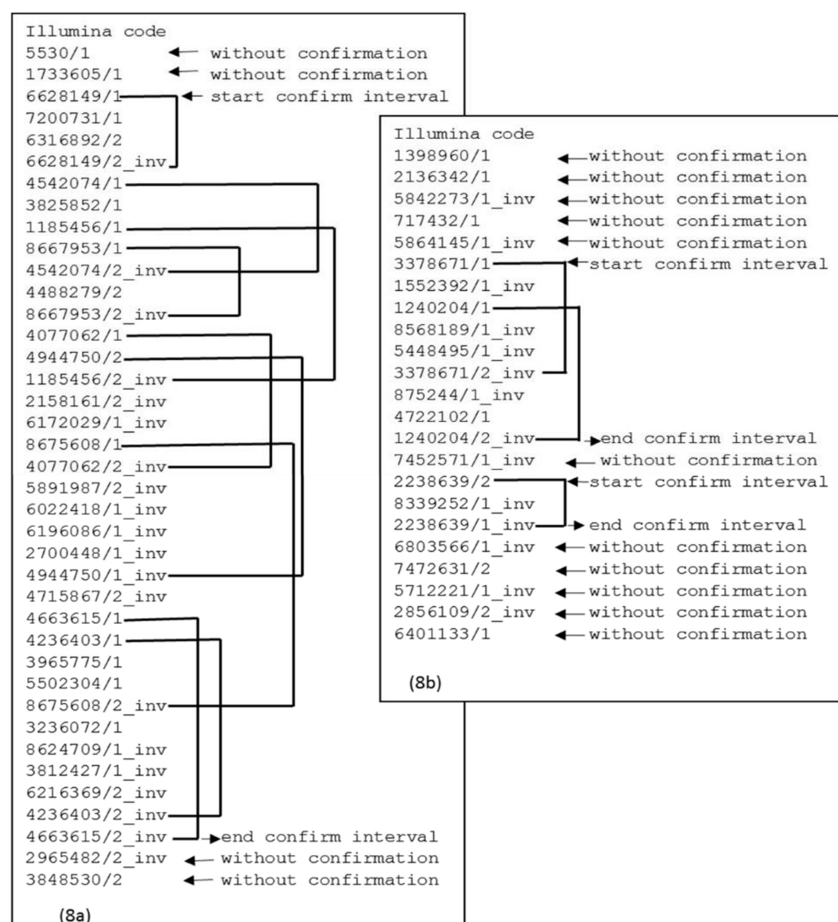


Figure 8. Examples of confirm intervals. (**8a**) *Contig* with confirm interval joined; (**8b**) *Contig* with confirm intervals separated.

4. Experiments

The application of the algorithms was carried out with the *Staphylococcus aureus* problem, taken from GAGE (Genome Assembly Gold-Standard Evaluations, 2011, [9]). The information consisted of 647,062 fixed-length fragments of 101 bases. The bacterium had a genomic chromosome
with 2,903,081 bases, a first plasmid with 27,428 bases and a second plasmid with 3170 bases.

The *Illumina equipment* generates three types of DNA sequences: single record, two records paired in a lineal sample and two records paired in a circular sample. In the paper of Mallén et al. [17], the work

was applied to a single record; in this new approach, we have worked with lineal paired records. These paired records contained the information necessary to confirm intervals in the assembled DNA. The reverse-complementary fragments were generated (inverted sequence exchanging AxT and CxG in reverse order) for both types of records, "/1" and "/2". The correspondence of the pairs is:

Type "/1" paired with the reverse complement of "/2",
Type "/2" paired with the reverse complement of "/1".

The generation of these reverse-complementary pairs generated duplicated *contigs*; they were eliminated from the result.

The *Velvet* release we used was 1.2.10, and we searched for the execution with the best results of N50 [9], varying the value of the *k-mer* [13]. Ultimately, the best value obtained was 31. The *Eden* release we used was 3.13, and, in the same way, we took the best result with different overlap values to obtain the best N50 [9]. The best value was 30 overlapping bases.

Illumina equipment delivers two fragment data files. The first contains the record type "/1" and the second contains the record type "/2". In both *Edena* and *Velvet*, it is possible to define a parameter for how to use these files. During the tests, this was considered.
With our programs (in the C/C++ programming language), we also had different parameter values with respect to overlapping. The best result of N50 [9] was found with 50 bases. In the results, we produced two versions, A and B. In our programs, we merge both data files because, during the initial tests, we found these records in the same file. We deleted the records with "N" values for the bases; this value means that the base is undetermined. In addition, we removed all characters except for the A, C, G, and T values. During the *Trie* process, we removed duplicate fragments, and, finally,

we eliminated repetitions of the same base value with more than 15 occurrences. This decision was based on an analysis of the data as a special case and would not be applicable to other organisms.

The A version of our program was designed for paired records; the B version also included both records, but the confirmation interval was applied to the *contigs*. In this version, we also eliminated the *contigs* with a length shorter than 1.5 times the fragment size. *Edena* also carries out this process, but this decision cannot be modified by the user; it is a consistency requirement. This action resulted in a modest improvement in N50 [9]. Table 1 shows the results of the executions in the "Generated" section. To determine how close we were to obtaining total genome reconstruction, we did a search of every *contig* in the original genome with our results, with those of *Edena* and with those of *Velvet*.

The results presented in Table 1 have been obtained from QUAST (v 4.4. CAB: Center for Algorithmic Biotechnology. St. Petersburg, Russia) [23]. The search was also done with Mummer [24]. The last column of Table 1 shows the result from Mummer, representing the *contigs* that have been found completely in the original genome. This result was applied to *Edena*, *Velvet* and our programs.

We performed all program executions two consecutive times to guarantee an execution time independent of computer dependencies such as those from the cache memory, the hard disk or the processor. With the Linux command "time" in terminal mode, we obtained the real time and

the user time. We took the real time in all the proofs and the lower of the two times from the first and the second executions.

All of the runs were made on the same computer, employing 64 bit Linux with Ubuntu 14.4 Long Term Support (London, UK), 16 gigabytes of RAM, an Intel (Santa Clara, CA, USA) i5-4460 processor @

3.2 GHz × 4 and a hard disk with an EXT4 partition type. The programming language of our programs (versions A and B) was C/C++, with the standard compilation C++ ISO (-std = C++11), and without any optimization. Table 2 shows the execution times for each of the cases tested.

Table 1. Experimental results.

Organism:		Staphylococcus aureus					Genome: 2,903,081 with Two Plasmids			
		Generated (Statistics without Reference)					Found in Genome (Statistics with Genome Reference)			
Program	**Parameter**	***Contigs* Generated**	***Contigs* in the Assembly**	**Total Bases in the Assembly**	**Total Bases in the *Contigs***	**N50 from *Contigs***	**NG50 from *Contigs* Found**	**Total Number of Aligned Bases**	**Genome Fraction %**	***Contigs* Found by Mummer in Genome at 100%**
Velvet	k-mer = 31	1059	693	2,733,950	2,816,990	**6666**	**6216**	2,733,853	93.942%	**67.3%**
Edena	overlap = 30	3038	1670	2,017,901	2,419,142	**1380**	**938**	2,017,901	69.39%	**86.6%**
Version A	overlap = 50	2985	1701	2,234,717	2,619,179	**1539**	**1183**	2,233,545	76.326%	**84.3%**
Version B	overlap = 50 w/confirm interval	20,439	2090	1,455,722	6,016,455	**675**	**500**	1,437,787	25.986%	**93.4%**

Table 2. Execution times.

Program	Step	Execution Time (s)
Velvet	Velveth	23.085
	Velvetg	6.397
Edena	Edena–DR	363.676
	Edena-e	2.051
Version A	FragAPares	69.140
	ListAdy	5.002
Version B	FragAPares	69.140
	ListAdyU	5.834

5. Conclusions

The advantages of our algorithms are described below.

Before applying the confirmation intervals, our N50 [9] was superior to *Edena* but not to *Velvet*. It is clear that *Velvet* uses *de Bruijn* graphs [13], and it contains a rebuilding *contigs* step. The resultant N50 of *Velvet* is better than those of *Edena* and our programs (see Table 3).

Table 3. N50 comparison.

	N50 from Generated *Contigs*
Velvet	6666
Edena	1380
Version A	1539
Version B	675

After applying the confirmation intervals, our N50 [9] for the paired fragments was smaller, but the quality of the *contigs* was greater than those of the other programs, as seen in the percentage of Mummer-located *contigs* (see Table 4).

Table 4. *Contigs* found by Mummer.

	Contigs Found by MUMMER in the Genome at 100%
Velvet	67.3%
Edena	86.6%
Version A	84.3%
Version B	93.4%

The values found by Mummer (see Table 1) do not help *Velvet* (67.3%), and our program yielded the best value (93.4%). If Mummer locates the *contig* completely in the genome, it appears as a 100% value; if Mummer cannot locate the *contig*, it adjusts the difference, splitting or tolerating certain errors and trying to relocate the pieces, reporting these as a percent value less than 100%. These values are not included in Table 1.

Regarding the execution times (see Table 2), in the first step, *Edena* took 6 min to calculate the overlap, and it took the longest time for the samples. In our version, an equivalent process took approximately 1

min, so we concluded that our process is much faster than *Edena*, though that is not the case when compared to *Velvet*. Our program requested approximately 4 gigabytes of RAM to manage the *Trie* [22]. Meanwhile, as the *Trie* grows, it becomes asymptotic; the construction of the *Trie* is rapid, with times on the order of five seconds.

A potential future study could search the "/2" type records outside the *contig* with a "/1" type record without its matching pair. These might generate a *contig* with a greater confirmation interval and, consequently, a better N50 value [9].
To obtain an improvement in the execution time, it would be advisable to eliminate the transitive edges of the graph. With this action, the data volume would be reduced significantly.

The identification of the sequences of the *contigs* could represent valuable information for molecular biologists, and these could probably be obtained by looking for paired fragments that span multiple *contigs*. The records with undetermined values ("N") could probably also contribute some information about the *contig* ordering.

Finally, the scaffolding process (*contigs* assembly) could generate important results for biologists, including a likely new step of detecting overlapping within *contigs* and giving rise to some adjustments in the quality assurance of the assembly.

Acknowledgments: This work has been developed with the support of Universidad Iberoamericana.

Author Contributions: J. Emilio Quiroz-Ibarra developed the algorithms and programs and conceived the idea for the study. Guillermo M. Mallén-Fullerton originated the idea and developed the algorithms. Guillermo Fernández-Anaya provided the formal verification of the algorithms, the complexity analysis and observations to improve them. All authors approved the manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Watson, J.D.; Crick, F.H.C. A Structure for Deoxyribose Nucleic Acid. *Nature* **1953**, *171*, 737–738. [CrossRef] [PubMed]
2. Sanger, F.; Coulson, F. DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. USA* **1977**, *74*, 5463–5467. [CrossRef] [PubMed]
3. Gingold, E. An introduction to genetic engineering. In *Molecular Biology and Biotechnology*; Royal Society of Chemistry: London, UK, 1988; pp. 25–46.
4. Myers, E.W., Jr. A history of DNA sequence assembly. *IT Inf. Technol.* **2016**, *58*, 126–132. [CrossRef]
5. Sanger, F.; Coulson, A.R.; Hong, G.F.; Hill, D.F.; Petersen, G.B. Nucleotide sequence of bacteriophage λ DNA. *J. Mol. Boil.* **1982**, *162*, 729–773. [CrossRef]
6. Myers, E. Toward Simplifying and Accurately. *J. Comput. Biol.* **1995**, *2*, 275–290. [CrossRef] [PubMed]
7. Staden, R. A strategy of DNA sequencing employing computer programs. *Nucleic Acids Res.* **1979**, *6*, 2601–2610. [CrossRef] [PubMed]
8. NCBI Genome&Maps, National Center for Biotechnology Information. Available online: https://www.ncbi.nlm.nih.gov/guide/genomes-maps/ (accessed on 21 November 2016).
9. Salzberg, S.L.; Phillippy, A.M.; Zimin, A.; Puiu, D.; Magoc, T.; Koren, S.; Treangen, T.J.; Schatz, M.C.; Delcher, A.L.; Roberts, M.; et al. A critical evaluation of genome assemblies and assembly algorithms. *Genome Res.* **2012**, *22*, 557–567. [CrossRef] [PubMed]
10. Tammi, M. *The Principles of Shotgun Sequencing and Automated Fragment Assembly*; Center for Genomics and Bioinformatics: Stockholm, Sweden, 2003.

11. Hernandez, D.; Tewhey, R.; Veyrieras, J.B.; Farinelli, L.; Østerås, M.; François, P.; Schrenzel, J. De novo finished 2.8 Mbp *Staphylococcus aureus* genome assembly from 100 bp short and long range paired-end reads. *Bioinformatics* **2014**, *30*, 40–49. [CrossRef] [PubMed]
12. Bang-Jensen, J.; Gutin, G.; Yeo, A. When the greedy algorithm fails. *Discret. Optim.* **2004**, *1*, 121–127. [CrossRef]
13. Compeau, P.E.; Pevzner, P.A.; Tesler, G. How to apply de Bruijn graphs to genome assembly. *Nat. Biotechnol.* **2011**, *29*, 987–991. [CrossRef] [PubMed]
14. Zerbino, D.R.; Birney, A.E. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.* **2008**, *18*, 821–829. [CrossRef] [PubMed]
15. Burks, C.; Forrest, S.; Parsons, R. Genetic Algorithms for DNA Sequence Assembly. *ISMB* **1993**, *1*, 310–318.
16. Mallén-Fullerton, G.M.; Fernandez-Anaya, G. DNA fragment assembly using optimization. In Proceedings of the IEEE Congress on Evolutionary Computation, Cancun, Mexico, 20–23 June 2013; pp. 1570–1577.
17. Mallén-Fullerton, G.M.; Quiroz-Ibarra, J.E.; Miranda, A.; Fernández-Anaya, G. Modified Classical Graph Algorithms for the DNA Fragment Assembly Problem. *Algorithms* **2015**, *8*, 754–773. [CrossRef]
18. Fasta_Formats, DNA Sequence Formats, Genomatix GmbH, 2016. Available online: https://www.genomatix.de/online_help/help/sequence_formats.html#FASTA (accessed on 30 October 2016).
19. Ukkonen, E. On-line construction of suffix trees. *Algorithmica* **1995**, *14*, 249–260. [CrossRef]
20. Gusfield, D. Linear-Time Construction on sufixx trees. In *Algorithms on Strings, Trees and Sequences*; Cambridge University Press: Melbourn, UK, 1997; pp. 94–119.
21. Black, P. Adjacency List, Dictionary of Algorithms and Data Structures, 2008. Available online: https://xlinux.nist.gov/dads/HTML/adjacencyListRep.html (accessed on 4 November 2016).
22. Fredkin, E. Trie memory. *Commun. ACM* **1960**, *3*, 490–499. [CrossRef]
23. Gurevich, A.; Saveliev, V.; Vyahhi, N.; Tesler, G. QUAST: Quality assessment tool for genome assemblies. *Bioinformatics* **2013**, *29*, 1072–1075. [CrossRef] [PubMed]
24. Kurtz, S.; Phillippy, A.; Delcher, A.L.; Smoot, M.; Shumway, M.; Antonescu, C.; Salzberg, S.L. Versatile and open software for comparing large genomes. *Genome Biol.* **2004**, *5*, R12. [CrossRef] [PubMed]

© 2017 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).