ISSN 1870-4069

Algoritmo genético aplicado al alineamiento múltiple de secuencias genéticas

Luz Andrea Garcia Sena, David Israel Perez Valerio,
Adriana Berenice Maldonado Garcia, Ernesto Ríos-Willars

Universidad Autónoma de Coahuila,
Facultad de Sistemas,
Coahuila

{andrea_sena, davidvalerio, berenice.maldonado, riose}@uadec.edu.mx

Resumen. El Problema del Alineamiento Múltiple de Secuencias es uno de los procesos fundamentales para la bioinformática. Este artículo aborda un análisis detallado sobre el funcionamiento de tres distintas versiones de un algoritmo aplicadas al Alineamiento múltiple de secuencias genéticas, dichas versiones tienen el objetivo de mejorar los alineamientos a partir de los procesos de mutación y reproducir de individuos con secuencias genéticas de la mundial base de datos NCBI. Inicialmente se dispone de una primera versión del algoritmo utilizada como base o punto de partida para la creación de sus dos variantes posteriores, cuyo propósito de estas dos versiones es optimizar la calidad de los resultados del algoritmo inicial estableciendo las modificaciones necesarias para mejorar la calidad de los individuos iniciales y la forma en la que se reproducen para crear nuevas generaciones de individuos más perfeccionadas y mejor alineadas. En este artículo se da a conocer la estructura y el funcionamiento de estas tres versiones con el propósito de realizar un análisis para obtener los resultados más destacables de acuerdo con las pruebas realizadas en cada versión.

Palabras clave: Alineamiento múltiple de secuencias, MSA, bioinformática, algoritmo genético.

A Genetic Algorithm for the Multiple Sequence Alignment Problem

Abstract. The Multiple Sequence Alignment Problem is one of the fundamental processes for bioinformatics. This article deals with a detailed analysis of the operation of three different versions of an algorithm applied to the Multiple Alignment of Genetic Sequences, these versions have the objective of improving the alignments from the mutation and reproduction processes of individuals with genetic sequences from the global NCBI database. Initially, a first version of the algorithm is available as a basis or starting point for the creation of its two subsequent variants, whose purpose of these two versions is to optimize the quality of the results of the initial algorithm by establishing the necessary modifications to improve the quality of the initial individuals and the way in which they reproduce to create new generations of individuals that are more

perfected and better aligned. This article presents the structure and operation of these three versions in order to carry out an analysis to obtain the most outstanding results according to the tests carried out in each version.

Keywords: Multiple sequence alignment, MSA, bioinformatics, genetic algorithm.

1. Introducción

A partir de finales de los años 80 en adelante, el término "bioinformática" ha sido mayormente empleado para describir métodos computacionales destinados al análisis comparativo de datos genómicos. No obstante, su definición original abarcaba un ámbito más amplio, siendo concebida como el estudio de los procesos informáticos dentro de sistemas bióticos [1]. En los organismos, el ADN despliega su función como el portador del material genético, transmitiendo así la información hereditaria de una generación a otra. Todos los seres vivos divergen a lo largo del tiempo a partir de un ancestro común, evolucionando mediante cambios en su ADN [2]. Por tanto, la capacidad de secuenciar el ADN de un organismo se define como un requisito esencial en la investigación biológica.

Diversas investigaciones han centrado su objetivo en desarrollar herramientas para llevar a cabo la secuenciación del ADN. Anteriormente, este proceso se limitaba a secuenciar unas pocas decenas o cientos de nucleótidos de forma simultánea. Sin embargo, en la actualidad, la secuenciación del ADN se realiza mediante máquinas de alto rendimiento que pueden secuenciar miles de millones de bases en un solo día. En el ámbito de la biología computacional, el alineamiento de secuencias de ADN, ARN o proteínas emerge como una tarea esencial y recurrente. Este proceso implica la comparación de un conjunto de secuencias para identificar las regiones donde coinciden y donde difieren entre sí.

El alineamiento múltiple de secuencias reviste una importancia fundamental en la bioinformática y la biología computacional, ya que proporciona perspectivas sobre la estructura y función de las biomoléculas [3]. El algoritmo genético representa un método de búsqueda heurística adaptativa, fundamentado en los principios de la genética de poblaciones. Su introducción se atribuye a John Holland en los inicios de la década de 1970. Este algoritmo se caracteriza por ser un enfoque de búsqueda *bioinspirado* a partir de la mecánica de la selección natural y los procesos genéticos [4]. Los algoritmos evolutivos se emplean para abordar problemas que carecen de una solución eficiente y bien definida.

Este enfoque se utiliza en la resolución de problemas de optimización y en modelado y simulación, donde se recurre a la aleatoriedad. Los Algoritmos Genéticos (GA) representan una solución para la población de candidatos (conocidos como individuos, organismos o genotipos) en problemas de optimización, avanzando hacia opciones más efectivas. Cada candidato presenta un conjunto de características (los genes o fenotipo) que pueden evolucionar y cambiar; la evolución comienza con una población de individuos aleatorios y se desarrolla de manera iterativa, considerando la población como base para cada reproducción.

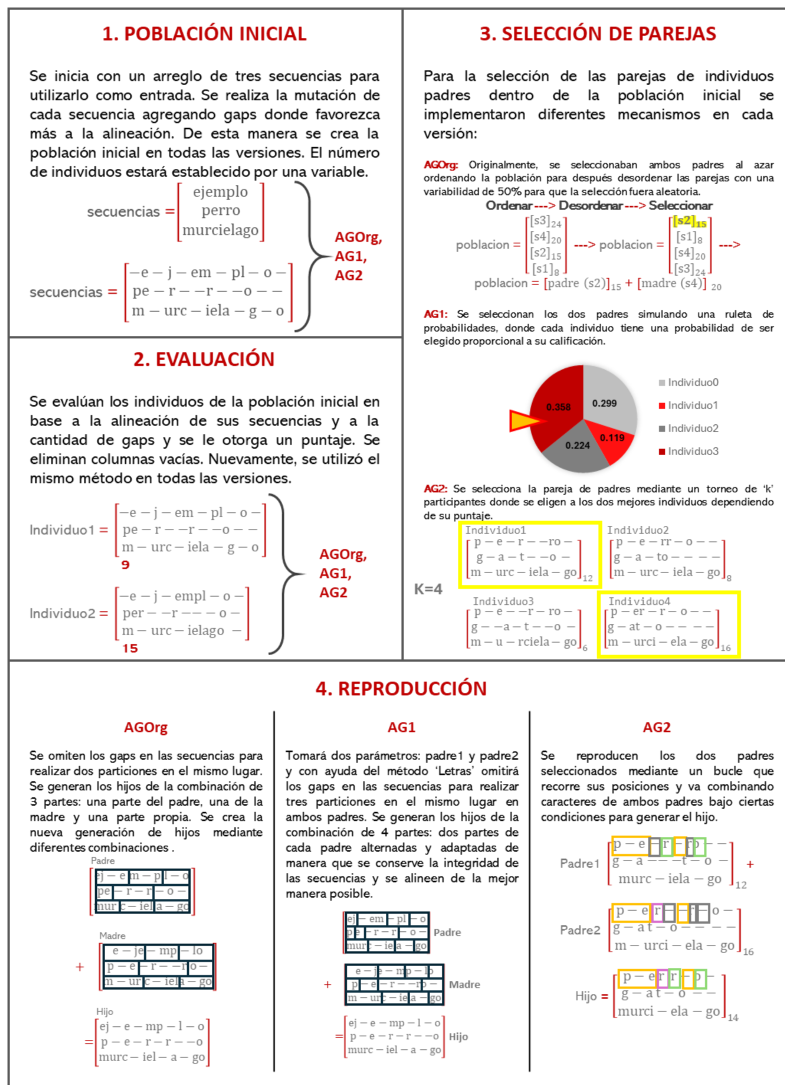

Fig. 1. Infografía comparativa sobre el funcionamiento de los mecanismos principales en las tres versiones.

En cada generación, se evalúa la aptitud (fitness) de todos los individuos, siendo esta aptitud usualmente el valor de la característica objetivo que se está optimizando. Cuando los individuos son lo suficientemente aptos, se seleccionan probabilísticamente de la población existente y sus genes se modifican para crear una nueva generación de individuos (recombinados y potencialmente mutados al azar). El AG continúa este proceso hasta que se ha alcanzado un número máximo de generaciones o de satisfacción [5]. Se destaca que lenguajes de alto nivel como Python o Perl suelen ser más eficientes en el procesamiento de listas o cadenas en comparación con C/C++/Java [4].

No obstante, el éxito práctico de estos algoritmos aún se encuentra en fase de investigación. La principal razón radica en la aleatoriedad inherente a estos algoritmos en el proceso de toma de decisiones.

Esta aleatoriedad introduce una complejidad adicional tanto en la ejecución como en la comprensión del proceso durante la aplicación de los algoritmos [6].

2. Planteamiento del problema

Cuando hay dos secuencias que pueden acomodar un número limitado de inserciones de huecos, el número de alineaciones aumenta a medida que aumenta la longitud de las secuencias. El número de alineaciones potenciales para un par de secuencias m y n, con 'k' inserciones (m,n) [7], se puede calcular usando la ecuación (1):

$$f = \sum_{k=0}^{\min(m,n)} \frac{(m+n-k)!}{k!\,(m-k)!\,(n-k)!} \,. \tag{1}$$

Cuando se intenta resolver el problema de alineamiento múltiple de secuencias, utilizando un enfoque de fuerza bruta, el problema se vuelve NP-completo. Por el contrario, la programación dinámica tiene una complejidad de O(LN), donde L es la longitud de la secuencia y N es el número de secuencias [8]. Los investigadores tienen como objetivo mejorar la eficiencia de este problema a través de enfoques heurísticos y metaheurísticos e implementaciones paralelas para reducir los costos computacionales [9].

Se requiere el desarrollo de un algoritmo de alineamiento múltiple de secuencias genéticas con el propósito de procesar dicho alineamiento, utilizando los métodos más idóneos, secuencias de caracteres, principalmente provenientes de paquetes de software diseñados para el alineamiento de secuencias de ADN y proteínas en formato FASTA. Este algoritmo debe ser capaz de operar eficientemente con cadenas de caracteres de cualquier longitud.

3. Materiales y métodos

Para el desarrollo del algoritmo original (AGOrg) realizado en el lenguaje de programación Java, así como la segunda versión (AG1), en cambio, para la segunda variante optimizada (AG2), se hizo una migración al lenguaje de programación Python, considerando que esto podría contribuir a su optimización.

Para todos los procesamientos se comparan las métricas de *tiempo* y *fitness* alcanzado por el algoritmo. Los tres algoritmos genéticos se publican en la plataforma GitHub para futuras referencias en el link github.com/luzAndreaGs/Algoritmosde AlineamientoMultipledeSecuencias. Los experimentos necesarios se llevaron a cabo utilizando un equipo portátil con un procesador Ryzen 5, una tarjeta gráfica AMD Radeon, un disco de estado sólido de 256GB y 12GB de memoria RAM. Para la realización de estos mismos experimentos, se emplearon nueve archivos de secuencias FASTA de secuencias genéticas de ADN como entrada, obtenidas de la base de datos NCBI [10]:

Tabla 1. Promedios de los resultados de fitness y tiempo para cada algoritmo en cada grupo de secuencias.

Grupo	Fitness			Tiempo		
	AGOrg	**AG1**	**AG2**	**AGOrg**	**AG1**	**AG2**
C1	8480.4	8608.8	8609.8	14	14	11.9
C2	1402	1404.6	1407	4.1857	3.8158	2.8608
C3	50268.43	50203.4	50304.13	224.08	149.89	83.02684

- Bacterial Escherichia coli strain con 2297 bases de longitud.
- Bacterial Enterococcus faecalis strain con 2805 bases de longitud.
- Bacterial Porphyromonas gingivalis strain con 1196 bases de longitud.
- Bacterial Bifidobacterium longum con 590 bases de longitud.
- Bacterial Helicobacter pylori isolate con 315 bases de longitud.
- Bacterial Staphylococcus aureus con 305 bases de longitud.
- Gen Humano MAPK3 con 9116 bases de longitud.
- Gen Humano MAPK13 con 14009 bases de longitud.
- Gen Humano PRKACA con 26075 bases de longitud.

Las secuencias se organizaron en tres grupos para realizar las pruebas del funcionamiento de los algoritmos: el primer grupo está compuesto por las secuencias bacteriales Escherichia coli strain, Enterococcus faecalis strain y Porphyromonas gingivalis strain; el segundo grupo se compone de las secuencias bacteriales Bifidobacterium longum, Helicobacter pylori isolate y Staphylococcus aureus; finalmente el tercer grupo de secuencias se compone de genes humanos MAPK3, MAPK13 y PRKACA.

4. Algoritmos propuestos

Se describen los algoritmos propuestos con los que se realizó una comparativa de desempeño para el alineamiento de las secuencias descritas. **Versión inicial del algoritmo (AGOrg):** El algoritmo se compone de cuatro clases: “Individuo”, “Generación”, “Archivos” y “Algoritmo”. Aquí la explicación del trabajo que tienen cada una de las clases y sus respectivos métodos:

Individuo. Hace el trabajo completo para manipular y mutar un arreglo de secuencias de caracteres (que convierte en una matriz) para alinearlas de manera que tengan la misma longitud y estén ordenadas en columnas.

- *Calificar().* Asigna puntuaciones para las secuencias basándose en su alineación de las letras y la presencia de gaps.
- *Mutar().* Realiza la mutación de las secuencias: primero, busca si hay gaps en cada secuencia y determina qué gaps eliminar y cuáles agregar.
- *ModificarGaps().* Realiza dichas modificaciones.
- *Alinear().* Alinea las secuencias de manera que todas tengan la misma longitud, esto lo hace agregando gaps al final de cada secuencia de ser necesario.

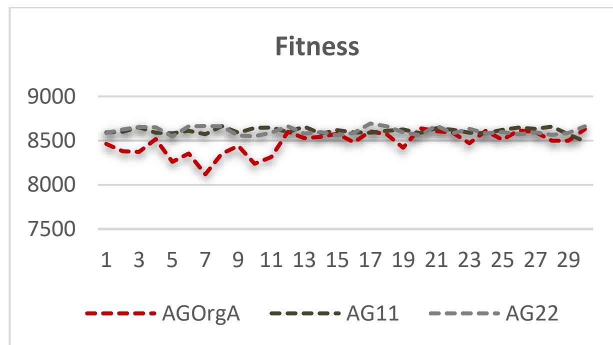

Fig. 2. Gráfica que representa y compara el puntaje del fitness de cada corrida en cada una de las versiones del primer grupo de secuencias.

– *Eliminar().* Después de haber mutado el arreglo de secuencias elimina las columnas que solo contienen gaps.

Generación. Se encarga de manejar las generaciones de individuos, donde cada uno de ellos representa una posible solución al problema de alineamiento de secuencias. La clase consta de una variable (tipo int) que actualmente está inicializada con el número 4, representa el número de individuos por generación y se puede modificar por cualquier número múltiplo de 4 debido a que cada pareja tiene 4 hijos.

– *Ordenar().* Acomoda la población de individuos de acuerdo con su puntuación (de mayor a menor) utilizando el algoritmo básico de burbuja.
– *DesordenarParejas().* Desacomoda aleatoriamente las parejas de individuos para crear una variabilidad del 0.50 al momento de seleccionar la pareja de padres.
– *Letras().* Cuenta únicamente las letras (ignorando los gaps) de cada una de las secuencias, itera sobre cada una de ellas para posteriormente realizar las particiones en la misma letra de cada secuencia en ambos padres.
– *Hijo().* Crea un nuevo individuo (hijo) a partir de dos individuos padres con dos particiones en cada uno (se dividen en 3 secciones) en la misma posición con ayuda del método Letras(). El hijo hereda y combina dos secciones de las secuencias de los padres (una de cada uno) y de acuerdo con ellas adapta una tercera parte que es propia de el mismo hijo.
– *Reproducir().* Ordena la población de individuos, luego desordena las parejas y finalmente reproduce una nueva generación de individuos a partir de los padres de la generación actual.

Archivos. Esta clase se encarga de leer archivos de texto. Utiliza un objeto de Scanner para leer el archivo línea por línea, omite la primera línea, luego lee el resto del archivo y concatena cada línea en la cadena txt. Si ocurre una excepción *FileNotFoundException*, el método imprime un mensaje de error.

Algoritmo. Es la clase que contiene el método principal (main) para ejecutar el algoritmo de ordenamiento de secuencias. Comienza instanciando un objeto de la clase *Archivos* o llamando a un arreglo de Strings llamado secuencias que contiene las secuencias de caracteres que se utilizaran como entrada del algoritmo; en caso de instanciar archivos, se declaran variables con las rutas de los archivos que se van a utilizar en el arreglo de secuencias. Instancia la clase *Generación* para crear un objeto de esta clase pasando el arreglo como argumento del constructor, esto inicializa la primera generación de individuos con las secuencias dadas.

Llama al método *Ordenar* con el objeto de *Generación* creado para ordenar la población inicial de individuos. La muestra en pantalla previamente ordenada y separando por un espacio los caracteres de las secuencias de cada individuo. Después hay un bucle que se ejecuta 10 veces. En cada iteración del bucle se lleva a cabo la reproducción (método *Reproducir*) del objeto de *Generación*, seguido del método para ordenar la nueva población.

Finalmente se imprime la población actual de individuos en cada bucle de reproducción con el respectivo puntaje de cada uno. De esta manera trabajan en conjunto las cuatro clases componentes del algoritmo original. Dentro de las clases, se detectó que el algoritmo únicamente hace su trabajo completo para un arreglo de tres secuencias, al querer utilizar más de tres secuencias no realiza la reproducción debido a un error en el método implementado, esto nos llevó a la conclusión de que sería un buen punto de mejora. Esta versión inicial fue hecha por una alumna de noveno grado (actualmente) de la Facultad de Sistemas de la Universidad Autónoma de Coahuila quien autorizó compartir el código alumnos de un grado inferior y utilizarlo para realizar las mejoras posteriores.

4.1 Segunda versión del algoritmo (Ag1)

En esta versión de la optimización del algoritmo, se trabajaron principalmente los siguientes puntos:

1. Correcto funcionamiento de la reproducción considerando cualquier cantidad de secuencias en los individuos.
2. Mejorar la calidad en la selección de los individuos padres.
3. Realizar más particiones en los individuos padres.
4. Implementar un nuevo método para asegurar que los individuos hijos mantienen íntegras sus secuencias.

Para trabajar la mayoría de estos objetivos, se modificó de manera considerable la clase "Generación" debido a que en ella se encuentran los métodos para la reproducción. A continuación, se explican los métodos que sufrieron cambios.

Constructor. El constructor declara una matriz *x* de caracteres con una longitud igual a la cantidad de elementos en el array de entrada (*entrada.length*), la matriz tendrá el mismo número de filas que la longitud del array de entrada. Itera sobre cada cadena en el array de entrada,cada cadena, se utiliza el método *toCharArray()* para convertirla en un array de caracteres. El array de caracteres resultante se asigna a la fila correspondiente en la matriz *x*. Se itera sobre cada índice *i* desde 0 hasta *num - 1*, donde *num* es el número de individuos en la población (anteriormente definido con el valor de

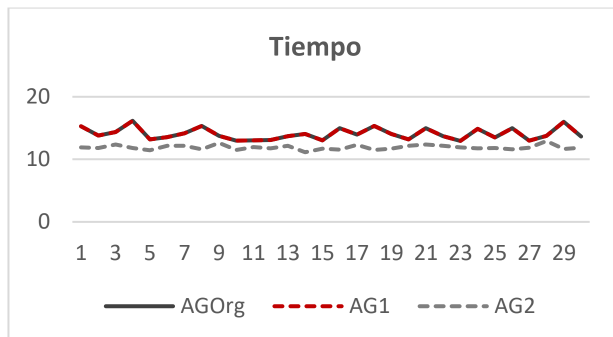

Fig. 3. Gráfica que representa y compara el tiempo de cada corrida en cada una de las versiones del primer grupo de secuencias.

4). Para cada índice *i*, se crea un nuevo objeto de la clase Individuo, pasando la matriz *x* como argumento al constructor de Individuo.

Esto crea un nuevo individuo en la población inicial utilizando las secuencias de caracteres de la matriz *x*. No hay una gran diferencia con la versión original, en resumen, el constructor ahora solo crea la matriz una sola vez y la utiliza repetidas ocasiones para crear los individuos de la población inicial a diferencia de la versión original, que crea la matriz repetidas veces a la par de los individuos dentro del bucle, lo que podría resultar un poco confuso al analizar el código.

SeleccionPorRuleta. Se calcula la sumatotal de las calificaciones de todos los individuos y en base a ella las probabilidades de cada individuo de ser seleccionado, esto se hace iterando sobre los individuos dividiendo su calificación entre la suma total. Se selecciona un numero aleatorio para la reproducción con ayuda de la función random.nextDouble() de la clase “Random” de java, que selecciona un valor aleatorio en el rango [0,1) es decir, que incluye el 0 pero excluye el 1. Se acumulan las probabilidades de selección de cada individuo en un valor acumulado. A medida que se recorre la lista de individuos, se compara el valor aleatorio generado con las probabilidades acumuladas.

Cuando el valor aleatorio es menor o igual que una de las probabilidades acumuladas, se selecciona el individuo correspondiente. Si ningún individuo cumple esta condición (por ejemplo, si el valor aleatorio es mayor que todas las probabilidades acumuladas), se selecciona un individuo aleatorio de la población.

Estas modificaciones hacen más certera la probabilidad de selección de un individuo sea proporcional a su calificación, lo que significa que los individuos con mejores calificaciones tendrán una mayor probabilidad de ser seleccionados lo que dará como resultado individuos hijos de mejor calidad.

Hijo. Toma dos variables como parámetros: *padre1* y *padre2*. Se inicializan las variables *Letras()* para contar el número de letras en cada parte, *n* para no perder el índice en el arreglo de las secuencias del hijo y *secHijo* para almacenar las partes de la secuencia del hijo (su tamaño depende de la cantidad de secuencias que hay en los individuos). Se declara un arreglo de tres particiones para definir los límites de las cuatro partes en las que estarán divididas las secuencias, todas tendrán los límites en la

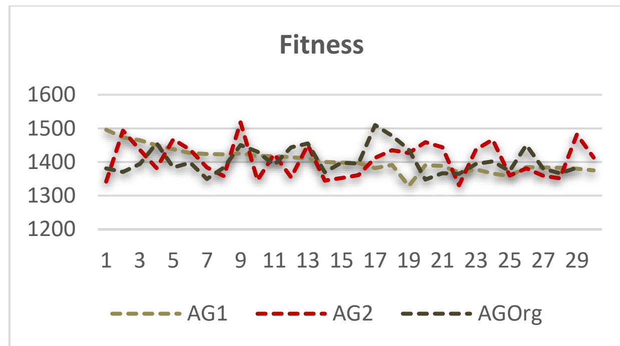

Fig. 4. Gráfica que representa y compara el puntaje del fitness de cada corrida en cada una de las versiones del segundo grupo de secuencias.

misma letra o el mismo lugar gracias al método *Letras()* como en la versión original del algoritmo.

Se inicia un bucle que itera sobre cada secuencia de los individuos padres considerando cualquier cantidad de secuencias y ya no únicamente tres para posteriormente recorrer las cuatro partes de cada secuencia y combinarlas de manera que el hijo hereda la primera y tercera parte del hijo del *padre1* y la segunda y cuarta parte del *padre2* de acuerdo con los límites establecidos. Se guarda la secuencia del hijo en el arreglo *secHijo*. Se convierte el arreglo *secHijo* en un arreglo de caracteres y. Se crea y devuelve un nuevo objeto Individuo con las secuencias del hijo. Este método divide las secuencias de los individuos padres en cuatro partes y las combina para formar la secuencia del hijo. Recorre las partes de todas las secuencias de los padres de manera simultánea

Reproducir. Se crea un nuevo arreglo de individuos llamado hijos con la misma longitud que la población actual (*num*). Se inicia un bucle for que se ejecutará *num* veces, donde i representa el índice de cada hijo que se va a crear. Dentro del bucle, se seleccionan aleatoriamente dos individuos padres llamando al método *seleccionPorRuleta().* Elige un individuo de la población actual donde la probabilidad de selección está proporcionalmente relacionada con la calificación de cada individuo.

Es por ello que ya no hace falta ordenar la población para seleccionar la pareja de padres. Se llama al método *Hijo()* para crear un nuevo individuo hijo a partir de los dos individuos padres seleccionados. El individuo hijo resultante se agrega al arreglo de hijos en la posición i. Una vez que se han creado todos los hijos, el arreglo de la población actual (población) se actualiza con el arreglo de hijos, lo que reemplaza la población anterior con la nueva generación de individuos.

verificarSecuencia. Se crea una variable temporal *sb* para modificar las secuencias de los individuos sin necesidad de crear otro objeto que construye las secuencias de los hijos sin los gaps. Itera sobre cada secuencia en el arreglo de los individuos hijos y dentro de este bucle, hay otro que itera sobre cada caraacter y mientras no sean gaps, se agregan a la variable temporal. Esta variable se convierte a una cadena de caracteres llamada *secuenciaHijo* y se inicializa una variable booleana integridad como true, que indicará si la secuencia del hijo coincide con las secuencias de la población.

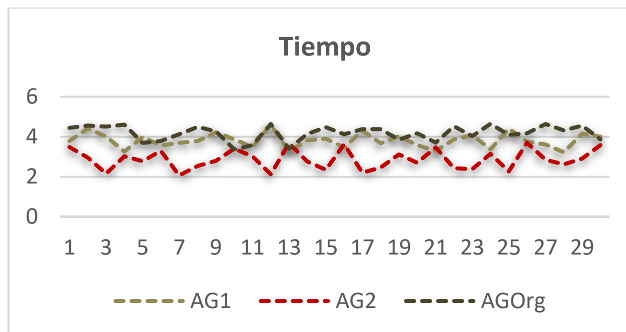

Fig. 5. Gráfica que representa y compara el tiempo de cada corrida en cada una de las versiones del segundo grupo de secuencias.

Se crea otra variable temporal s*ecuenciaOriginal* que se construye de la misma manera que la anterior pero ahora utilizando un individuo de la población inicial. Se comparan ambas variables y si son iguales, la variable integridad permanece como true, si no son iguales, la variable integridad se establece en false. Se devuelve un mensaje indicando si la verificación fue exitosa o si se encontró un error en la secuencia del hijo. Este método se manda a llamar en la clase principal para mostrar el mensaje correspondiente a cada individuo hijo debajo de su calificación.

4.2 Tercera versión del algoritmo (AG2)

Como propuesta de mejora en esta versión, surgió la idea de que el algoritmo se transportara a otro lenguaje de programación (Python) para conocer cómo sería su funcionamiento y realizar cambios para la optimización del algoritmo. Naturalmente se sabe que Python tiende a ser un lenguaje conocido por su facilidad de uso y su rápida iteración en el desarrollo, lo que puede favorecer a la diferencia de rendimiento.

Funciona igualmente con un arreglo de secuencias de caracteres, pueden ser palabras o archivos de texto. Primero almacena las secuencias en una lista de listas mediante el constructor de la clase 'Individuo' y procede a aplicar las mutaciones a cada secuencia, éste y los procesos de modificación de gaps, alineación y la asignación del puntaje se hacen de la misma manera que en el algoritmo original, siguiendo las mismas reglas para la calificación. En la clase *Generacion* se realizaron los cambios más significativos, a continuación se explican parte por parte:

Constructor. En él se establece un valor para el número de individuos que habrá en la población esto permite que se adapte a 'n' cantidad de secuencias, se crea la población inicial con una lista tomando de base las secuencias existentes de la siguiente manera: por cada índice del número de secuencias se crea un objeto de la clase 'Individuo' y se almacenan en la lista de población, además en el constructor se agregó un atributo que guarda las secuencias originales.

alinear_secuencias(). Se añadió este nuevo método que se encarga de mantener todas las secuencias con la misma longitud agregando gaps al final de las palabras que lo requieran dentro de la población inicial con el objetivo de facilitar sus procesos de comparación y reproducción.

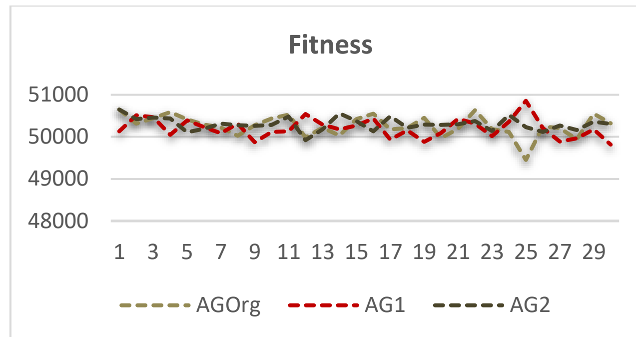

Fig. 6. Gráfica que representa y compara el puntaje del fitness de cada corrida en cada una de las versiones del tercer grupo de secuencias.

selección_torneo(). La selección de los padres ya no se hace de manera aleatoria, sino que ahora los padres se eligen con ayuda este método que selecciona un grupo de 'n' de individuos de la población inicial y los devuelve como participantes del torneo donde los que tienen mayor puntaje, tienen más posibilidades de ganar.

reproducción(). Primero se alinean las secuencias de la población, se selecciona una pareja de padres entre los 4 individuos de la población inicial seleccionando uno por uno, excluyendo el que ya fue seleccionado, genera cuatro hijos de estos padres en dos bucles de dos hijos cada uno donde el hijo1 toma los parámetros de los padres en el mismo orden que se eligieron y el hijo2 de forma invertida, se cambian los roles para introducir variabilidad en la descendencia y evitar sesgos o patrones que puedan surgir si siempre se utilizan los mismos padres en el mismo orden; la nueva población es la de los hijos.

hijo(). Sigue la lógica para reproducir los nuevos individuos: alinea los dos padres agregando gaps al final de cada secuencia de manera que ambos tengan la misma longitud, después va comparando los padres carácter por carácter para decidir de qué manera se van a estructurar a los hijos en base a algunas condiciones: a) Si ambos padres tienen el mismo carácter o también si el carácter del padre1 es un gap y en la misma posición el carácter del padre2 es alfabético, el hijo hereda el carácter del padre2 y b) En otro caso, el hijo hereda el carácter del padre1.

validar_secuencias(). Verifica que no haya modificaciones en las secuencias de los hijos, toma estas secuencias (*hijos*), las convierte en secuencias de caracteres alfabéticos únicamente (es decir, elimina sus gaps) y va comparando los elementos o secuencias de cada lista con las originales, utiliza una variable booleana que ayuda a tomar una decisión: si encuentra alguna discrepancia en ellas (False) quiere decir que las secuencias se están alterando al reproducirse, de lo contario, se mantienen integras (True). Por último, se manda a llamar este método en la clase principal 'Algoritmo' con el objeto de la clase *Generacion* dentro del bucle de reproducción tomando los elementos de las secuencias de la nueva población y comparándolas con las secuencias originales, imprime debajo de cada individuo hijo el resultado de su verificación: *True = "Individuo verificado correctamente", False = "Individuo no válido".* La figura 1 muestra el proceso de cuatro fases para los tres algoritmos desarrollados.

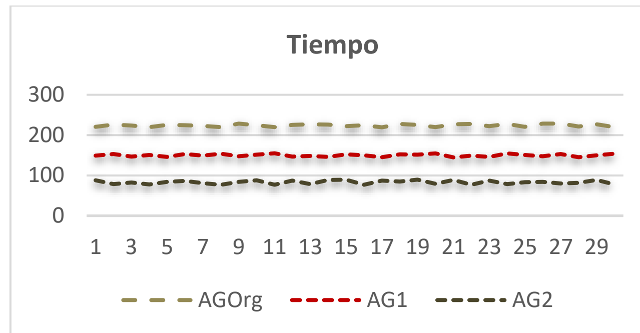

Fig. 7. Gráfica que representa y compara el tiempo de cada corrida en cada una de las versiones del tercer grupo de secuencias.

5. Experimento

Para realizar la comparación entre el funcionamiento de las tres versiones del algoritmo, se ejecutó un experimento en el cual se efectúan treinta corridas de cada uno de los algoritmos bajo los mismos parámetros en todas las versiones para que los resultados sean viables estadísticamente para comparativa. La población inicial es de 52 individuos (se debía utilizar una cifra que sea múltiplo de 4 ya que en la versión original se crean 4 individuos hijos por generación), un número máximo de gaps posibles a añadir de 50 para el proceso de mutación, así mismo 10 generaciones de reproducción. En esas treinta corridas se pretende evaluar el tiempo total de ejecución, el fitness de cada una y el promedio de los resultados (de fitness y tiempo en segundos) de cada versión.

6. Resultados

Se realizaron cálculos de promedios para los resultados de fitness y tiempo como comparativa entre los tres algoritmos en el proceso de los tres conjuntos de secuencias como se indica en la tabla 1. Así mismo se realizaron pruebas Anova de un factor para identificar diferencias significativas entre dichos resultados con el uso del software Minitab16. S describen estos resultados en gráficas comparativas de la Fig. 2 y Fig. 3. Al analizar los resultados obtenidos, podemos ver que efectivamente según las estadísticas los resultados de ambas versiones optimizadas (AG1 y AG2) mejoraron considerablemente en cuanto a puntaje y tiempo en comparación con la versión original, esto se describe en figuras 2 y 3. Respecto del tiempo, la prueba Anova indicó un valor F=23.97 con valor P=0.00 en la comparativa entre los tres algoritmos. Respecto del fitness, la prueba Anova indicó un valor F=23.97 y valor P=0.000 en la comparativa de los tres algoritmos. Por lo que aceptamos que existen diferencias con un nivel de confianza de 95.

Las figuras 4 y 5 representan los resultados de tiempo y fitness para el segundo conjunto. Respecto del tiempo en el resultado de los tres algoritmos, la prueba Anova

indicó un valor F= 78.57 y valor P=0.000 con nivel de confianza de 95, por lo que asumimos que existen diferencias. En la comparativa del fitness para los tres algoritmos, la prueba Anova indicó un valor F= 0.27 y valor P=0.763 con el mismo nivel de confianza. Por lo que asumimos que no existen diferencias.

Las figuras 6 y 7 representan los resultados de fitness y tiempo para los tres algoritmos en el tercer grupo de secuencias. Respecto del tiempo, en la comparativa de los tres algoritmos la prueba Anova indicó un valor F= 11052.68 y valor P=0.000 con nivel de confianza de 95. Por lo que asumimos diferencias significativas. Respecto del fitness, la prueba Anova indicó un valor F= 1.62 y valor P=0.203 con nivel de confianza de 95. Por lo que asumimos que no existen diferencias.

7. Conclusiones

Respecto de la métrica fitness, de acuerdo los resultados obtenidos y con los gráficos, podemos aseverar que, estadísticamente las versiones optimizadas (AG1 y AG2) mejoraron los resultados de manera exitosa en comparación con los que se obtuvieron en el algoritmo original (AGOrg) y de hecho, en ambas optimizaciones los resultaos fueron bastante similares aun estando implementados en diferentes lenguajes de programación (Java y Python), lo que quiere decir que las estrategias de selección y reproducción cumplieron con su propósito de manera exitosa aumentando la calidad de individuos en las nuevas generaciones. Con el tiempo de ejecución, ocurren cosas interesantes.

Podemos observar que el tiempo promedio de AGOrg y AG1 es casi idéntico y dichas versiones fueron desarrolladas en Java, se sabe que comparado con otros lenguajes de programación, Java es un poco más lento, concluyendo con que a pesar de ser una versión mejorada y que nos arroja mejores resultados en las generaciones (fitness), AG1 no presentó alguna ventaja en el tiempo de ejecución del algoritmo, en cambio AG2 si demostró tener ventaja si valoramos también el tiempo, esto puede atribuirse a que está desarrollado en Phyton y actualmente, Python está entre los lenguajes de programación más rápidos, superando a casi todos los demás en el mundo de la programación.

En resumen, podemos decir que definitivamente el rendimiento de ambas versiones AG1 y AG2 alcanzaron las expectativas deseadas y superaron los estándares que se obtuvieron con AGOrg, sin embargo, AG2 está por encima de AG1 ya que pese a la aleatoriedad que siempre existe en este tipo de algoritmos de ordenamiento múltiple de secuencias, éste logró perfeccionar los resultados de manera muy similar o un poco mejor, y además en menor tiempo. Como trabajo a futuro se visualiza la implementación de nuevas estrategias en el proceso de mutación para los diferentes algoritmos aquí planteados, así como incrementar el número de secuencias genéticas en conjuntos de prueba que permitan una comparativa más amplia y estadísticamente más rigurosa. También se visualiza la incorporación de nuevos recursos de hardware que permitan la experimentación con mayores niveles de complejidad para el algoritmo genético.

Agradecimientos. Los autores desean reconocer a la Facultad de Sistemas de la Universidad Autónoma de Coahuila por el apoyo en la realización de este trabajo.

Referencias

1. Hogeweg, P.: Las raíces de la bioinformática en la biología teórica. PLoS Computational Biologics, pp. 7 (2011)
2. Chowdhury, B., Garai, G.: A Review on Multiple Sequence Alignment from the Perspective of Genetic Algorithm. Genomics, vol. 109, no. 5–6, pp. 419–431 (2017) DOI: 10.1016/j.ygeno.2017.06.007.
3. Kaya, M., Sarhan, A., Alhajj, R.: Multiple Sequence Alignment with Affine Gap by Using Multi-objective Genetic Algorithm. Computer Methods and Programs in Biomedicine, vol. 114, no. 1, pp. 38–49 (2014) DOI: 10.1016/j.cmpb.2014.01.013.
4. Kumar, M., Husian, M., Upreti, N., Gupta, D.: Genetic Algorithm: Review and Application. Electrical Engineering eJournal, pp. 2–5 (2010)
5. Alam, T., Qamar, S., Dixit, A., Benaida, M.: Genetic Algorithm: Reviews, Implementations, and Applications. CompSciRN: Computer Principles, pp. 1–9 (2020). DOI: 10.48550/arXiv.2007.12673.
6. Hanif, M.K., Talib, R., Awais, M., Saeed, M.Y., Sarwar, U.: Comparison of Bioinspired Computation and Optimization Techniques. Current Science, vol. 115, no. 3, pp. 450–453 (2018)
7. Waterman, M.S.: Introduction to Computational Biology: Maps, Sequences and Genomes. Chapman and Hall/CRC (2018)
8. Steiner, G.: On the Complexity of Dynamic Programming for Sequencing Problems with Precedence Constraints. Annals of Operations Research, vol. 26, no. 1–4, pp. 103–123 (1990). DOI: 10.1007/bf02248587.
9. Almanza-Ruiz, S.H., Chavoya, A., Duran-Limon, H.A.: Parallel Protein Multiple Sequence Alignment Approaches: A Systematic Literature Review. The Journal of Supercomputing, vol. 79, no. 2, pp. 1201–1234 (2022). DOI: 10.1007/s11227-022-04697-9.
10. Sayers, E.W., Bolton, E.E., Brister, J.R., Canese, K., Chan, J., Comeau, D.C., Connor, R., Funk, K., Kelly, C., Kim, S., Madej, T., Marchler-Bauer, A., Lanczycki, C., Lathrop, S., Lu, Z., Thibaud-Nissen, F., Murphy, T., Phan, L., Skripchenko, Y., Tse, T., Wang, J., Williams, R., Trawick, B.W., Pruitt, K.D., Sherry, S.T.: Database Resources of the National Center for Biotechnology Information. Nucleic Acids Research, vol. 50, no. D1, pp. D20–D26 (2021). DOI: 10.1093/nar/gkab1112.