***Pseudo-Código;*** *Análisis de estrategias.*

Analizaremos estrategias posibles a utilizar para la resolución del alineamiento de secuencias. Posterior a esto determinaremos una ruta de acción para finalmente crear un pseudo código que nos sirva de guía.

***MSA y DCA.***

El MSA está basado en la programación dinámica. Para alinear dos secuencias de 300 aa, el algoritmo de programación dinámica utiliza una matriz bidimensional, por lo que el número de operaciones que hay que hacer es de 3002. Para N secuencias se utiliza una matriz N-dimensional, con 300N operaciones. Debido a esto, necesita una gran cantidad de recursos computacionales y mucho tiempo. En la práctica apenas se utiliza, solo si N = 3 o si las secuencias son cortas, 6 < N < 8.

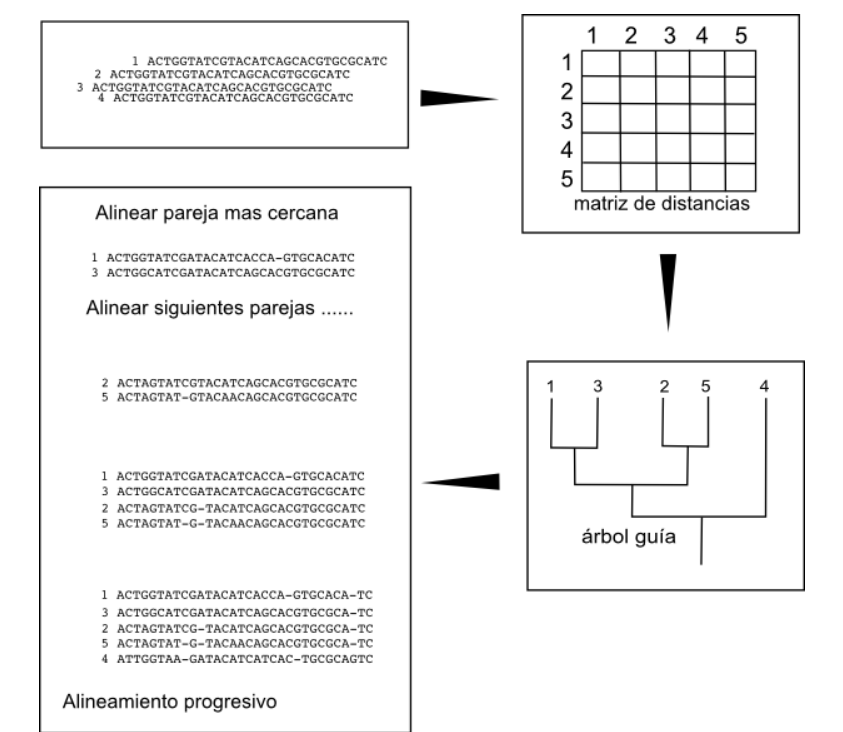
El algoritmo Divide y Vencerás (DCA en inglés) consiste en dividir cada secuencia en dos subsecuencias en un punto cercano al medio. El proceso se repite hasta que las secuencias son lo suficientemente cortas (según un umbral fijado L). A partir de estas subsecuencias se realizan alineamientos por medio del algoritmo de programación dinámica, y después se concatenan para generar el AMS de las secuencias original

***Algoritmo de construcción progresiva.***

La mayoría de los programas relacionados al tema se basan en esta estrategia; El método consiste en primero realizar alineamientos de dos en dos. A partir de estos alineamientos se construye una matriz de distancias entre las secuencias y un árbol guía basado en estas distancias. Mediante este árbol podemos encontrar las parejas de secuencias más similares.

Al alineamiento del par de secuencias más similar se va añadiendo el resto de las secuencias o alineamientos por el orden determinado por el árbol guía.

Nota importante: No nos garantiza que el alineamiento sea el mejor posible, pero es capaz de encontrar una solución óptima eficaz.



***Aproximaciones iterativas (SAGA).***

Esta estrategia calcula una solución subóptima mediante un alineamiento progresivo, luego de esto modifican el alineamiento mediante programación dinámica hasta que la solución converge. En otros alineamientos de progresión normal al cometer un error ya no se puede corregir, pero este método soluciona esto.

El método que analizar es SAGA , que emplea el algoritmo genético. Este algoritmo simula una especie de selección natural. Consta de 4 etapas:

1. Inicialización (G0): se escriben las secuencias (una en cada fila), y cada una se desplaza hacia la derecha un número aleatorio de posiciones. Se incluyen huecos en los extremos para que al final todas tengan la misma longitud.

2. Evaluación mediante una función objetiva: se valora cada secuencia siguiendo un criterio o función objetiva (OF). Cuanto mejor es el alineamiento, mayor es la puntuación. A partir de los alineamientos G0 se obtiene la “descendencia” esperada. De la G0 inicial, se mantiene el 50% en la siguiente generación G1, la otra mitad se sustituye por la descendencia de los alineamientos más aptos (según el criterio seleccionado).

3. Reproducción (mutaciones y recombinaciones): para generar la descendencia, se modifican los alineamientos parentales con mutaciones o recombinaciones, siguiendo la función objetiva, y el 50% con mayor puntuación es el que pasa a la siguiente generación. Las mutaciones (introducción de un hueco, no cambio de aminoácidos) no cambian la secuencia porque se arruinaría el alineamiento. Para introducir los huecos se generan primero dos subgrupos y en cada grupo se añaden huecos en dos posiciones aleatorias. A estos huecos se les pueden realizar más modificaciones, como desplazar los dos, o dividirlos en vertical/horizontal y mover sólo uno de ellos. Cuando hay recombinación se necesitan dos progenitores, a partir de los cuales se genera un primer hijo con la parte izquierda del parental 1 y la parte derecha del parental 2 y el segundo hijo con la parte derecha del parental 1 y la parte izquierda del parental 2.

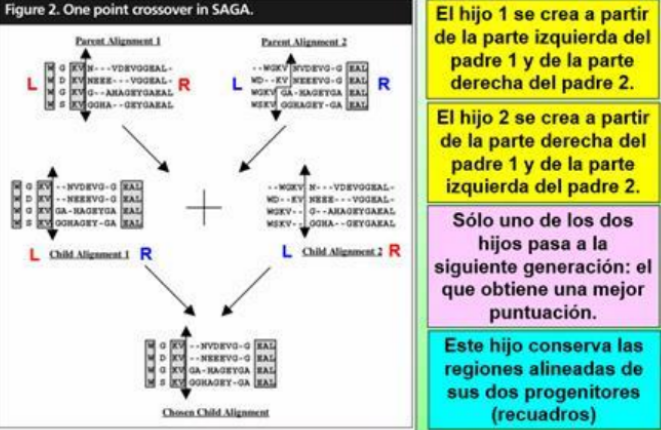
4. Finalización: la señal de parada del algoritmo depende del criterio fijado, que puede ser

una puntuación o un número determinado de generaciones (normalmente 100).

4. Finalización: la señal de parada del algoritmo depende del criterio fijado, que puede ser

una puntuación o un número determinado de generaciones (normalmente 100).

4. Finalización: la señal de parada del algoritmo depende del criterio fijado, que puede ser una puntuación o un número determinado de generaciones (normalmente 100).



***Análisis***

Luego de haber analizado todos los métodos llegue a la conclusión de que el método de MSA no es conveniente para el proyecto debido a la limitación que presenta, por otra parte, el DCA parece prometedor por lo que quedara como segunda estrategia en caso de que la primera no funcione correctamente. El algoritmo de construcción progresiva también se ve prometedor pero debido a lo complejo que suena programarlo no se usara. Por último, el seleccionado fue el método iterativo SAGA, esto debido a que se logra comprender relativamente fácilmente.

***Pseudo Código***

Nota 1: Para la creación de este Código en Python se necesitará la biblioteca de Random.

Proceso AlineamientoSAGA

# Paso 1: Lo primero será la inicialización y los parámetros de entrada.

Var Secuencia1 String # Aquí se manejan solo dos secuencias, pero la idea es que sean mas

Var Secuencia 2 String

Var TamañoPoblacion Int

Var rango\_mutacion Float

Var Generaciones Int # Define la cantidad de generaciones que hará el metodo

Var Poblacion = [Generar\_Alineamiento\_Random(Secuencia1,Secuencia2) Para \_

Hasta TamañoPoblacion.]

# Estará llamando otro proceso que no es parte del proceso principal.

# Se inicia el ciclo principal del método SAGA

Para generation Hasta Generaciones Hacer

# Este es el Paso 2, el proceso de Calcular es externo al P. Principal

Var fitness\_scores = [Calcular\_Fitness(alineamiento) Para alineamiento Hasta Poblacion]

# Paso 3.1: Selección de padres.

# Usando el método choices se toman de población las que generan mayor

# puntuacion

Var padres\_seleccionados = random.choices(Poblacion, weight =fitness\_score, k=2)

# Paso 3.2: Cruzamiento. Contiene proceso externo.

Var hijo1, hijo2 = cruzamiento(padres\_seleccionados[0], padres\_seleccionados[1])

#Paso 3.3: Mutación. Contiene proceso externo.

hijo1 = mutar(hijo1, rango\_mutacion)

hijo2 = mutar(hijo2, rango\_mutacion)

# Paso 3.4: Reemplazar menos aptos.

Población[fitness\_scores.index(mínimo(fitness\_scores))] = hijo1

Población[fitness\_scores.index(mínimo(fitness\_scores))] = hijo2

FinPara

# Paso 4: Selección del mejor alineamiento de la ultima generación

mejor\_alineamiento = max(Población, key = calculate\_fitness)

Leer “Mejor alineamiento:” + mejor\_alineamiento

Leer “Puntaje de Aptitud:” + calculate\_fitness(best\_aligment)

FinProceso

# Este proceso es el encargado de ir generar aleatoriamente los recorridos del arreglo.

# Es importante ver que este está diseñado para solo dos secuencias, por lo que podría sufrir cambios

# drásticos a la hora de meter más secuencias.

Proceso Generar\_Alineamiento\_Random(Secuencia1, Secuencia2)

Var Alineamiento Arreglo

Dimensión Alineamiento (2,0) # Crea el arreglo para las secuencias

Para i = 0 Hasta Longitud(Secuencia1) Hacer # Arreglo que recorre toda la secuencia 1

Si Var Random Proceso Random < 0.5 Entonces # Probabilidad de que inserte GAP o no

Alineamiento[0] += ‘-‘ # Se agrega GAP

Alineamiento[1] += Secuencia2[i]

Sino

Alineamiento[0] += Secuencia1[i]

Alineamiento[1] += Secuencia2[i]

FinSi

FinPara

Regresa Alineamiento

FinProceso

# Proceso de paso 2

Proceso Calcular\_Fitness (alineamiento)

Var score = 0 (Int) # Puntuacion

Para i Hasta Longitud(alineamiento[0]) Hacer Recorre el arreglo 1

Si alineamiento [0][i] == alineamiento[1][i] Entonces # Saber coincidencia

score += 1

FinSi

Si alineamiento [0][i] == ‘-‘ O alineamiento[1][i] == ‘-‘ Entonces # Detecta GAP’S

score -= 2

Sino # No hay coincidencia

score -= 1

FinSi

FinPara

FinProceso

# Proceso de cruzamiento del paso 3.

Proceso cruzamiento (padre1, padre2)

punto\_cruce = random.randint(0, Longitud(padre1[0])) # Num. Aleatorio entre 0 y Len del Padre

hijo1 = [padre1[0][punto\_cruce] + padre2[0][punto\_cruce],

padre1[1][punto\_cruce] + padre2[1][punto\_cruce]]

hijo2 = [padre2[0][punto\_cruce] + padre1[0][punto\_cruce],

padre2[1][punto\_cruce] + padre1[1][punto\_cruce]]

Regresa hijo1, hijo2 # Hijo1 toma las coincidencias de la izquierda del Padre1 e hijo2 toma las de

# Padre 2

FinProceso

# Proceso de mutación del paso 3.

Proceso mutación (alineamiento, rango\_mutacion)

Var alineamiento\_mutado Arreglo

alineamiento\_mutado Dimensiones [‘’,’’]

Para i Hasta Longitud(alineamiento[0]) Hacer

Si random.random() < rango\_mutacion Entonces # Ingresar mutación aleatoria

Si random.random() < 0.5 Entonces

alineamiento\_mutado[0] += ‘-‘

alineamiento\_mutado[1] += alineamiento[1][i]

Sino

alineamiento\_mutado[0] += alineamiento[0][i]

alineamiento\_mutado[1] += ‘-‘

FinSi

Sino

alineamiento\_mutado[0] += alineamiento[0][i]

alineamiento\_mutado[1] += alineamiento[1][i]

FinSi

FinPara

Regresar alineamiento\_mutado

FinProceso

*Estrategias de verificación.*

Para verificar que tan bien se ejecutó el código se realizara una comparación usando la biblioteca BioPython que trae consigo un método para alinear secuencias. Usando dos secuencias (el limite de este código) idénticas se verán las similitudes y diferencias para determinar que tan bien funciono y ver si proseguir con los resultados y complicaciones presentadas.

El pseudo-codigo se pasará a Python para esta prueba