Plegado de proteínas bajo el modelo Hydrophobic–Polar

Formal Methods in Computer Science & Engineering

Design of Bio-inspired Algorithms

Jonathan Carrero

Enero 2020

Índice de contenido

Índice de ilustraciones

1 Introducción

El modelo Hydrophobic–Polar (HP) es un modelo muy simplificado que se utiliza para estudiar los pliegues de las proteínas y que fue propuesto por Ken Dill en 1985. Este modelo deriva de la observación de que las fuerzas hidrófobas entre los residuos de aminoácidos son la fuerza impulsora para que las proteínas se plieguen en su *estado nativo*[1] [**1**].

El objetivo de este documento es entender los conceptos que aparecen en él de manera que sea autocontenido y cualquier lector que no esté familiarizado con conocimientos de biología pueda entender de qué trata.

[1] El estado nativo de una proteína es su forma debidamente plegada, la cual es operativa y funcional.

2 ¿Qué es un aminoácido?

Los aminoácidos son moléculas orgánicas que conforman la base de las proteínas. Están formadas de carbono, oxígeno, hidrógeno y nitrógeno. Entre sus funciones, los aminoácidos ayudan a descomponer los alimentos, al crecimiento o a reparar tejidos corporales, y también pueden ser una fuente de energía.

Existen una gran variedad de aminoácidos, siendo los más importantes: esenciales, no esenciales y condicionales. Además, los aminoácidos pueden ser clasificados de muchas maneras diferentes si atendemos a las propiedades de su cadena, ubicación en el grupo amino, etc.

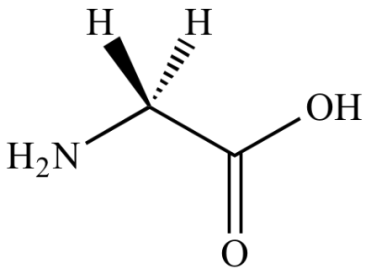

Ilustración 1: Estructura general de un aminoácido.

3 Modelo Hydrophobic–Polar

Como he dicho anteriormente, es un modelo simplificado para examinar los pliegues de proteínas en el espacio. En este modelo, todos los aminoácidos son clasificados únicamente como Hydrophobic (H) o Polar (P). A pesar de que el modelo HP abstrae muchos detalles del plegamiento de proteínas, este sigue siendo un problema NP-Hard en redes 2D y 3D [**2**].

3.1 Definiendo las características

Conviene definir alguna características de este modelo que nos ayudarán a entender mucho major qué es lo que estamos haciendo y cómo lo estamos haciendo.

- Dado el alfabeto {H, P}, una proteína es una secuencia de longitud n, tal como HPPH.

- Para nosotros, la red estará formada por una malla bidimensional, donde cada posición no puede estar ocupada por más de un residuo de aminoácido.
- No se admiten cruces en la secuencia (los caminos que forma la red no pueden cruzarse sobre la malla).
- Cuando dos residuos de aminoácido ocupan celdas vecinas se dice que interaccionan.

Dicho esto, el objetivo es lograr un estado de plegado en el que se haya minimizado la energía total. Esto se consigue maximizando el número de pares adyacentes de aminoácidos de tipo H, ya que, por cada par formado de este tipo, la energía se reduce.

A modo de ejemplo, supongamos por un momento que tenemos la cadena HPPHPPPHPPH. Imaginemos que, tras realizar dos pruebas iniciales, obtenemos los siguientes plegados:

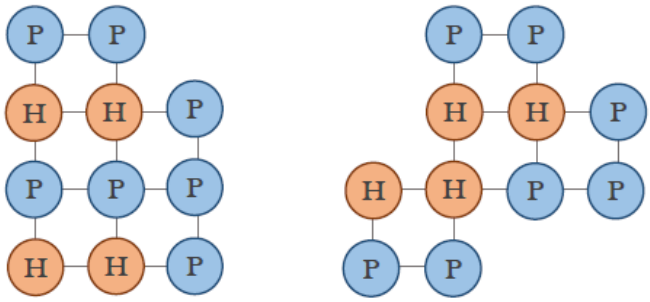

Ilustración 2: Ejemplo de pliegues con el modelo HP.

Si nos fijamos en el pliegue de la izquierda, observamos cómo tan solo ha habido dos enlaces entre aminoácidos de tipo H, mientras que en la figura de la derecha hay tres. Por lo tanto, el pliegue de la derecha sería mejor (recordemos que los pares enlazados de tipo H reducen la energía, que es el objetivo que estamos persiguiendo durante el plegado).

4 Implementación recursiva del modelo

La implementación se ha desarrollado en Java (la razón principal es porque es el lenguaje con el que más he trabajado y además permite una fácil organización a través de sus clases). El Proyecto es muy simple y consta únicamente de tres clases. Tanto el proyecto como la memoria pueden ser consultados en el siguiente enlace de Github: https://github.com/Joncarre/Java-language › *Algoritmos bioinspirados* › *Modelo Hydrophobic Polar*.

Al comenzar la ejecución, la clase *Main* recibe un argumento con la cadena de aminoácidos que se hayan introducido por parámetro. Para configurar los argumentos de entrada en Eclipse es necesario pulsar en *Run* › *Run Configurations…* › *Arguments*. Una vez hecho, aparecerá la siguiente ventana que vemos en la
Ilustración *3* y desde la que podremos escribir nuestra cadena. Cabe destacar que la clase *Main* comprobará si la cadena que se ha introducido es correcta, es decir: si está formada únicamente por caracteres 'H' o 'P'. Si no es así, lanzará una excepción y se detendrá la ejecución del programa.

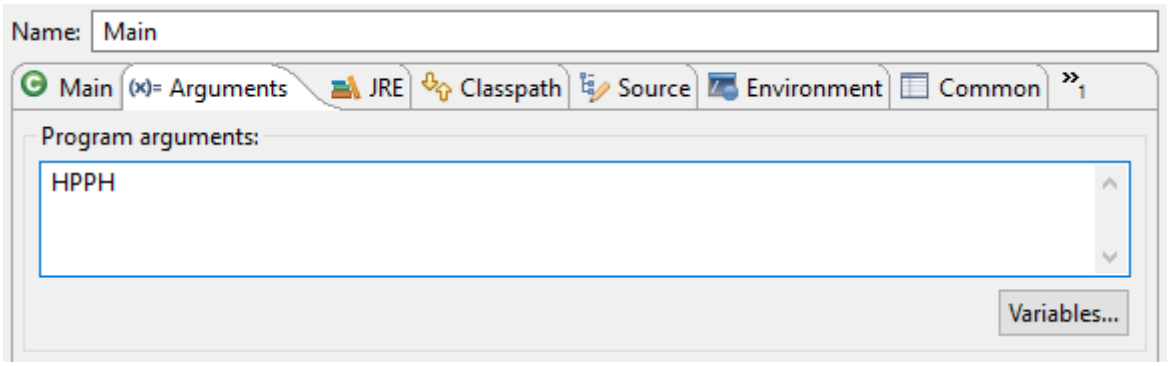

Ilustración 3: Introducir parámetros en Eclipse.

Carece de sentido explicar detalladamente cada uno de los métodos en cada una de las clases pero, a modo de ejemplo, podemos echar un vistazo a uno de los métodos más importantes llamado *getScore*. Dado un pliegue[2] de residuos de aminoácidos, este método asigna una puntuación (un fitness) a dicho pliegue.

[2] A veces uso la palabra "proteína" y a veces la palabra "pliegue", pero ambas palabras significan lo mismo; es el estado final que tiene la cadena de aminoácidos cuando se ha interaccionado con ella y se ha logrado crear una malla que la representa.

```java
/**
 * Calcula la puntuación de un determinado pliegue
 * @return
 */
public int getScore() {
    int score = 0;
    for (int i = 1; i < this.chain.length(); i++) {
        int dx = Math.abs(col[i] - col[i-1]);
        int dy = Math.abs(row[i] - row[i-1]);
        if (dx + dy != 1) {
            return Integer.MIN_VALUE;
        }
    }

   for (int i = 0; i < this.chain.length(); i++) {
        for (int j = i+1; j < this.chain.length(); j++) {
            if (row[i] == row[j] && col[i] == col[j])
                return Integer.MIN_VALUE;
            if (isHydrophobic(i) && isHydrophobic(j)) {
                if (row[i] == row[j] && (col[i] == col[j]+1 || col[j] == col[i]+1))
                    score++;
                else if (col[i] == col[j] && (row[i] == row[j]+1 || row[j] == row[i]+1))
                    score++;
            }
        }
    }
    return score;
}
```

Ilustración 4: Método getScore.

Este método realiza dos labores que son muy importantes a la hora de crear un pliegue:

- El primer bucle *For* determina que los residuos que forman la cadena sean adyecentes entre sí. Si no es así, asigna una baja puntación al pliegue resultante. Si recordamos la explicación del modelo HP, estee es un objetivo importante porque, si los residuos estuvieran esparcidos por la malla sin ningún tipo de adyacencia, no se crearían enlaces de tipo H y, por lo tanto, la energía del pliegue aumentaría considerablemente (cosa que no queremos que ocurra).
- Los siguientes dos bucles *For* anidados comprueban que, dados dos residuos, ambos se encuentren en coordenadas distintas pero adyacentes entre sí. Además, si se forma un enlace de tipo H, aumenta la puntuación asignada a esa proteína en +1.

4.1 Ejemplo de ejecución

Si ejecutamos el programa introducciendo como argumento la cadena HPPH, la cual forma la proteína más sencilla que existe y en la que tan solo aparece

un único enlace de tipo H, el resultado del plegamiento es el que podemos observer en la siguiente ilustración.

```
Tu cadena de aminoácidos es: HPPH
Enlaces de tipo H: 1

HH
PP
```

Ilustración 5: Ejemplo con cadena de longitud 4.

A decir verdad, la representación que se muestra por consola no es muy amigable para el usuario, pero si tratamos de ver el resultado como una malla que se ha construido de manera secuencial (en la que cada residuo ocupa unas determinadas coordenadas) observaremos que, en efecto, tan solo hemos obtenido un enlace de tipo H en el pliegue resultante, el cual aparece marcado en rojo.

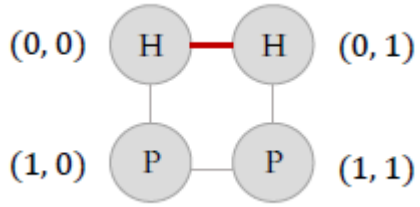

Ilustración 6: Representación secuencial de la cadena HPPH.

4.2 Limitaciones en la implementación

Uno de los puntos más importantes es ver las limitaciones que tiene el programa. Hoy en día, llevar a cabo la tarea de realizar el plegado de proteínas de la manera más eficiente posible es uno de los mayores retos en el campo de la bioquímica. Es por eso que llevar que el esfuerzo en mejorar cada vez más este tipo de programas es considerablemente grande.

Quizá el mayor problema de la implementación esté en la manera en la que se calcula el pliegue a medida que tomamos residuos de la cadena de aminoácidos. Este cálculo, como vimos durante el ejemplo, se realiza de manera totalmente secuencial. Si volvemos a fijarnos en la **Ilustración 6**, observamos que el primer residuo que aparece en la coordenada $(0, 0)$ es de tipo H, a continuación se explora la siguiente fila y se establecen las coordenadas para el primer residuo de tipo P, etc. Esta manera de calcular la malla hace imposible que,

por ejemplo, dada la cadena HPPH, aparezca en la primera posición el residuo P. El programa únicamente explora en las cuatro direcciones (norte, sur, este y oeste) considerando los residuos se manera secuencial. Como podemos imaginar, el número de comprobaciones enseguida crece de manera exponencial, de manera que para calcular un pliegue formado por tan solo 4 residuos de aminoácido, el número de comprobaciones es de $4^3 = 64$.

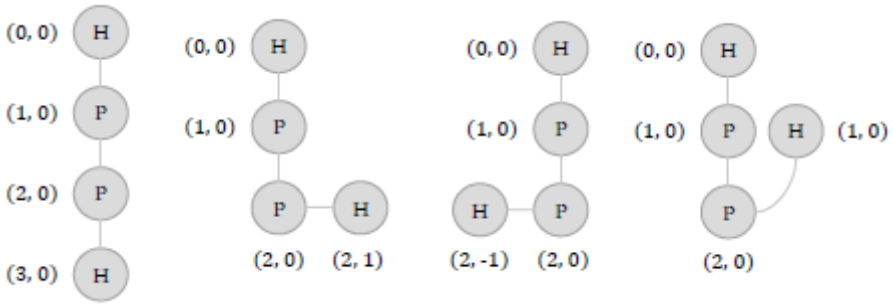

Ilustración 7: Primeras cuatro comprobaciones en la cadena HPPH.

Así, de manera general, temenos que para una cadena de longitud n, el número de comprobaciones es 4^{n-1}, ya que el primer residuo siempre tiene una posición fija (no es necesario saber dónde colocarlo). Tal y como vimos en el método getScore, el último residuo ocupa exactamente las mismas coordenadas que el segundo residuo. Ambos están en las coordenadas **(1, 0)**. Esto hace que el método getScore asigne una puntuación muy baja a este pliegue resultante para así asegurarse de que nunca sale elegido como la mejor proteína que puede ser formada con la cadena HPPH.

Algunas pruebas con cadenas de longitud 15 han tomado alrededor de 1 minuto de ejecución. Sin embargo, al aumentar la cadena a 16 aminoácidos, el tiempo se ha disparado considerablemente hasta los casi 6 minutos debido a las $4^{15} = 1.073.741.824$ comprobaciones. Para terminar, a continuación podemos ver una malla formada por una cadena de longitud 16.

```
Tu cadena de aminoácidos es: HHPHPHHPPPPHPPHP
Enlaces de tipo H: 7

   PPP
  PHHP
  PHHHH
   PPHP
```

Ilustración 8: Ejemplo con cadena de longitud 16.

5 Implementación con algoritmo genético

Hasta ahora hemos visto una primera implementación en la que de manera recursiva se buscaba un plegamiento que maximizase el número de enlaces entre aminoácidos de tipo H. Este proyecto formó parte del proyecto básico de la asignatura Algoritmos Bioinspirados, el cual fue desarrollado durante el mes de noviembre.

Yendo un paso más allá en busca de una implementación más cercana a los objetivos de la asignatura, se ha desarrollado un nuevo proyecto que mezcla algunas ideas de los algoritmos genéticos con optimización por enjambres de partículas. Durante las siguientes secciones vamos a explicar qué partes tiene este nuevo proyecto, su funcionamiento, limitaciones y algunos de las pruebas realizadas. El código del proyecto puede ser consultado en la siguiente dirección: https://github.com/Joncarre/Java-language › *Hydrophobic Polar with GA.*

5.1 Diseño e ideas descartadas

Antes de empezar a pensar incluso en el diseño me tuve que informar de qué era aquello del plegado de proteínas. Sabía en qué consistía el problema a grandes rasgos, pero no tenía ni idea de algo tan básico como saber qué era un aminoácido (aunque esto ya lo expliqué durante el proyecto básico).

La primera idea que fue descartada tras darle muchas vueltas es utilizar un grafo para representar los individuos de la población. Esta idea creo que era más ambiciosa que la final, pero empezaron a surgir muchos problemas. Al principio no parecía tan complejo en cuanto a la representación del grafo o la manera en la que se calcularía el fitness al final de cada evolución de la población (dicho calculo se podría haber hecho con alguno de los algoritmos que vi durante la asignatura de TAIS, como por ejemplo haber utilizado Dijkstra). Esta idea únicamente se componía de nodos, los cuales representaban cada uno de los inviduos de la población (aminoácidos de la cadena de proteínas).

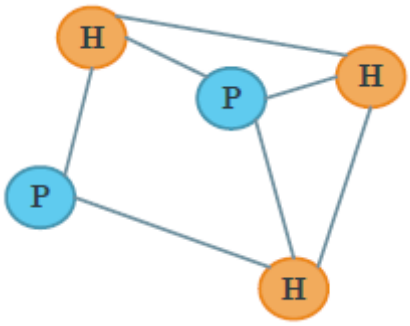

Ilustración 9: Representación del modelo en grafo

El problema que tuve con esta representación fue cuando me di cuenta de que la evolución de la población iba a ser bastante compleja. Para empezar, si un nodo puede tener varios enlaces (e incluso es el propio usuario quien, en principio, debería establecer el número de enlaces que cada individuo va a tener al ser inicializado), ¿cuáles son las reglas que hay que tener en cuenta a la hora de evolucionar la población? Está claro que los enlaces deberían cambiar, pero ¿cómo? Nótese que si los cambiamos arbitrariamente, esto podría dar lugar a muchas composiciones no deseadas y a que la población no converga. Al mismo tiempo, llevar a cabo un control de las reglas sobre cómo debería evolucionar cada individuo no parece tarea sencilla. Supongamos que creamos unas reglas que nos permiten obtener poblaciones compactas[3]. ¿No podría ocurrir que estas reglas sean tan estrictas que acotasen significativamente la manera en la que evoluciona la población? Esto produciría poca variedad en el algoritmo genético, lo cual, en teoría, es una de las principales bases que todo algoritmo genético debe tener.

Otra representación que tampoco se llevó a cabo fue la de utilizar un array en el que cada posición almacenase cuatro direcciones. Cuando empecé a pensar en cómo quería representar la población, caí en la cuenta de que lo más sencillo era simplemente utilizar una matriz en la que cada individuo se pudiese mover en las cuatro direcciones posibles: norte, sur, este y oeste. El inconveniente principal es que esta estructura era un poco difícil de manejar y confusa (además de no ser muy enficente en términos de búsqueda), pues el array de individuos tenía que tener almacenadas las direcciones. Pero ¿por qué la población tiene que almacenar las direcciones de cada individuo? En todo caso, cada individuo debería almacenar sus propias direcciones desde un punto de

[3] Con población compacta me refiero al hecho de que todos los individuos formen un grafo y no haya, por ejemplo, distintos grafos e incluso nodos solitarios.

vista de encapsulación (la información de un determinado individuo no debería mezclarse con la información de los demás). Como he dicho, esta idea fue rápidamente descartada.

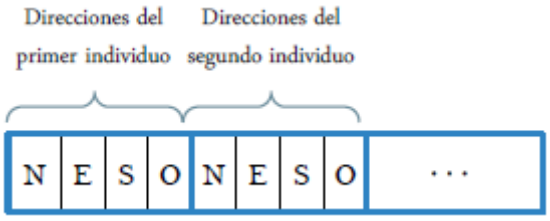

Ilustración 10: Representación del modelo en array

La idea por la que finalmente opté es muy parecida a esta pero teniendo una mejor jerarquía de clases. En las siguientes secciones veremos detenidamente cómo se representa la información que maneja el algoritmo, aunque cabe decir que la solución por la que me decanté fue que cada individuo tuviese su propio vector de direcciones. Al mismo tiempo, los individuos también tienen unas determinadas coordenadas, las cuales son creadas en una clase llamada *Location*.

5.2 Representación de la información

A diferencias del proyecto anterior en el que debíamos configurar la entrada de argumentos en Eclipse, esta vez el programa, al ser ejecutado, lee de fichero la cadena de aminoácidos. Esta cadena, de nuevo, viene representada por caracteres que indican qué tipo de aminoácido estamos leyendo, y finaliza con la lectura de cualquier otro carácter (yo he establecido que se lea el carácter "E" de "End"). Una vez leída, los inviduos son inicializados en una posición aleatoria que se encuentra dentro de las coordenadas de la matriz.

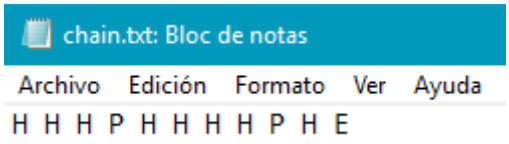

Ilustración 11: Ejemplo de *input*

Como he avanzado, la salida del programa será una matriz que se representará por consola en la que cada aminoácido estará situado en unas determinadas coordenadas. Además, como veremos en seguida, cada uno de los animoácidos tiene cuatro enlaces (cada uno de ellos con una cierta polaridad) que también

aparecerán en la salida por consola. En la **Ilustración 12** podemos observar como aparece la matriz con cada uno de los aminoácidos y sus correspondientes cuatro enlaces (cada uno de ellos con un tipo de polaridad).

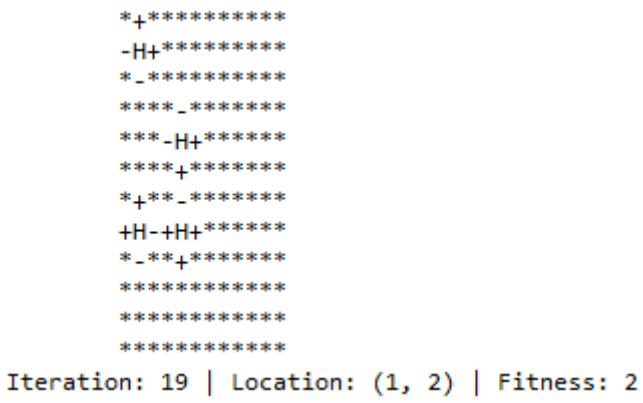

Ilustración 12: Ejemplo de *output*

Nótese que esta matriz tan solo representa la manera en la que se muestra la información al usuario que esté interactuando con el programa, pero interiormente no tiene sentido manejar una matriz cuando podemos representar la cadena de aminoácidos utilizando únicamente un vector de la clase *Vector* que nos proporciona Java.

5.2.1 Parámetros importantes

Como es de suponer, el programa tiene muchos parámetros comunes a cualquier algoritmo genético (tamaño de la población, número de individuos, tamaño de los individuos, número de veces que evoluciona la población, etc.). Mi intentación no es centrarme en estos parámetros porque creo que quedan fuera de esta pequeña memoria, sino más bien nombrar aquellos que son importantes de cara a la resolución del problema.

Dicho esto, la clase *Individual* es la que contiene los parámetros que indican cómo debe comportarse la cadena (desde un punto de vista biológicamente muy básico) a medida que esta evoluciona. Estos parámetros han sido elegidos por mí después de conocer y preguntar algunas nociones sobre qué es lo que ocurre en nuestro cuerpo durante un verdadero plegamiento de proteínas. Por lo tanto, he intentado que el resultado sea lo más cercano posible a lo que sería la realidad y, al mismo tiempo, establecer un escenario "fácil" con el que poder trabajar y poder acoplar a lo que sería el algoritmo genético.

El primero de los parámetros es la temperatura. La temperatura, junto con el pH, polaridad del disolvente y fuerza iónica, es uno de los factores que afectan a la desnaturalización de las proteínas. Sin entrar en muchos detalles, la desnaturalización es el proceso en el que una proteína sufre un cambio estructural (por algunas de las razones que acabo de nombrar) y dicho cambio produce, a su vez, un cambio funcional que puede conllevar una pérdida total de la función biológica. En lo que a nosotros nos concierne, esta temperatura viene reflejada como un atributo de tipo double que se inicializa con un valor entre 30 y 50 (un rango de temperatura que más o menos he considerado factible).

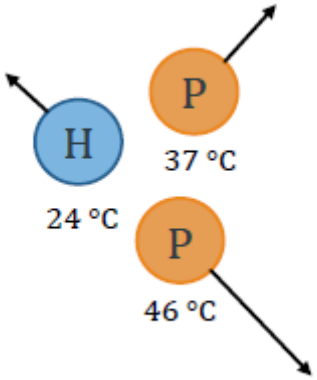

Ilustración 13: Representación de la temperatura

El segundo factor es la localización. La localización de cada aminoácido viene representada por un objeto de tipo *Location*. Esta clase ha sido creada simplemente con la utilidad de almacenar dos valores (las coordenadas necesarias para situarlo en la matriz) en un solo parámetro.

El último de los parámetros es el que más se aleja del modelo real. Digamos que en nuestro cuerpo, cada aminoácido tiene una determinada carga que puede ser positiva o negativa. Esta carga influye enormemente cuando la proteína pasa por las diferentes fases del plegamiento, en el que se forman agrupaciones de aminoácidos para dar lugar a las conocidas, entre otras, hélices alfa o plegamientos beta (durante la estructura secundaria). En mi modelo, no es individuo (aminoácido) el que tiene la polaridad, sino que este posee un vector con cuatro valores que indican cuatro enlaces (norte, sur, este y oeste). Son estos enlaces los que sí poseen polaridad, haciendo que, mientras cada individuo se mueve por la matriz, permitan (o no) unirse a otros individuos adyacentes.

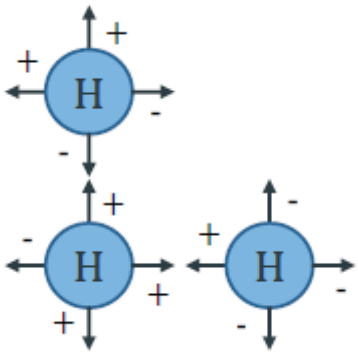

Ilustración 14: Representación de la polaridad

5.3 Cálculo del Fitness

Como todo algoritmo genético, cada individuo de la población necesita almacenar un parámetro fitness que indique su valor (qué tan bueno es) respecto a los demás. Durante este pequeño apartado vamos a ser cómo se calcula este valor.

Cada vez que se ha terminado de evolucionar la población, dos métodos evaluan la matriz y establecen cuánto enlaces de tipo H se han creado. En el modelo antiguo únicamente era necesario tener dayacencia entre dos aminoácidos de tipo H para lograr un enlace pero aquí, queriendo ser un poco más realista, he introducido el último de los parámetros que vimos en la sección anterior. Ahora, ya no es suficiente con tener adyacencia sino que además los enlaces deben tener polaridades distintas. Como he dicho, esta comprobación se realiza al final de cada evolución, estableciendo el nuevo valor fitness para cada uno de los individuos de la población.

```
*+**********
-H+*********
*-**********
****-*******
***-H+******
****+*******
*+**-*******
+H-+H+******
*-**+*******
**********
**********
**********
Iteration: 19 | Location: (1, 2) | Fitness: 2
```

Ilustración 15: Uniones entre individuos

5.4 Fases de la evolución

Como ya sabemos, el objetivo del modelo Hydrofobobic-Polar es maximizar el número de enlaces de tipo H. Esto hace que inevitablemente tengamos que intentar, de alguna manera, que los individuos estén lo más cerca posible los unos de los otros. Ya sabemos que al comienzo de la ejecución, durante la inicialización de la población, cada individuo es puesto en unas determinadas coordenadas aleatorias. Así pues, es lógico pensar que vamos a tener una fase de movimiento tarde o temprano. El objetivo de esta fase no es más que perseguir crear un mayor número de enlaces. A continuación vamos a ver cómo se ha desarrollado esta fase (que ha sido la más compleja del proyecto) y qué elementos influyen en ella.

En primer lugar, debemos darnos cuenta de que hay ciertos elementos a tener en cuenta. Uno de los más importantes es que, de alguna forma, los individuos deberían acercarse cada más al Fittest actual (durante cada evolución de la población). Esta, de hecho, es una de las principales ideas detrás de los algoritmos PSO. Otra de las ideas importantes que hay que tener en cuenta es que en nuestra implementación tan solo vamos a permitir un movimiento en una de las cuatro direcciones: norte, sur, este y oeste. Además, únicamente se avanzará una casilla.

Estos movimiento de los individuos no son totalmente obligatorios durante la evolución ni mucho menos, sino que se ejecutarán según una cierta probabilidad. Dicha probabilidad indicará si se realiza un movimiento y, además, hacía qué dirección.

5.4.1 Cálculo de las probabilidades

La probabilidad de realizar un movimiento para cada uno de los individuos es 1. Es decir: al comienzo de la fase de movimiento (si no hiciéramos nada más), un individuo se movería sí o sí en una de las cuatro direcciones. Lo que haremos será ir restando probabilidad de movimiento para decidir si finalmente se lleva a cabo o no. En esta resta intervienen tres factores (que están directamente realizados con los parámetros que vimos durante la sección **Parámetros importantes**). Cabe destacar que no ha sido fácil elegir estos tres factores (más bien la manera de ir restando las probabilidades): nótese que debe haber un equilibrio entre cuál de ellos tiene más importancia al realizar un movimiento y cuál no. Al mismo tiempo, una resta demasiado extrema podría conllevar que el invididuo nunca se moviese, y tal vez no es eso lo que queremos

en determinadas situaciones (ocurre lo mismo si no tenemos cuidado y restamos un valor muy bajo, podría pasar que los individuos siempre se movieran).

El primero de los factores a tener en cuenta es el fitness. El fitness de un inviduo impacta directamente en la probabilidad que tiene de desplazarse por la matriz. Pensemos por un momento que tal vez no conviene desplazar un individuos que tiene un fitness "lo suficientemente alto" (en este caso, una buena opción sería dejar que el resto de individuos se acercasen a este para lograr un número mayor de enlaces). La manera en la que se resta la probabilidad es la siguiente:

Probabilidad = Probabilidad – (1 – 1/2^n), donde *n* es el fitness del individuo actual.

Otro factor de menor peso es la temperatura. Como vimos durante las secciones anteriores, la temperatura es un factor que influye en la manera en cómo los aminoácidos interactúan entre sí: a mayor temperatura, mayor probabilidad de enlazamiento (en nuestro caso, mayor probabilidad de desplazamiento para continuar buscando más individuos). La resta de esta probabilidad es como sigue:

Probabilidad = Probabilidad – (Probabilidad/Temperatura)

El último factor también tiene una notable importancia ya que se trata de la distancia que hay desde cada una de las cuatro posiciones a las que puede moverse hacía el Fittest. Por ejemplo, es razonable pensar que si hay una menor distancia moviéndose hacia el norte, entonces este movimiento tenga una mayor probabilidad de ser realizados que si se mueve hacía el sur. Así pues, en este caso se realizan cuatro restas distintas, en la que cada una de ellas se calcula de la siguiente manera:

Probabilidad = Probabilidad – (Probabilidad/Resta coordenadas)

Estas sucesivas restas arrojarán en un número comprendido entre 0 y 1, el cuál indicará si el individuo que actualmente está siendo comprobado se moverá (y una vez tomada esa decisión, nuevamente de manera aleatoria se comprobará hacía qué dirección se desplazará, basado en el último de los factores).

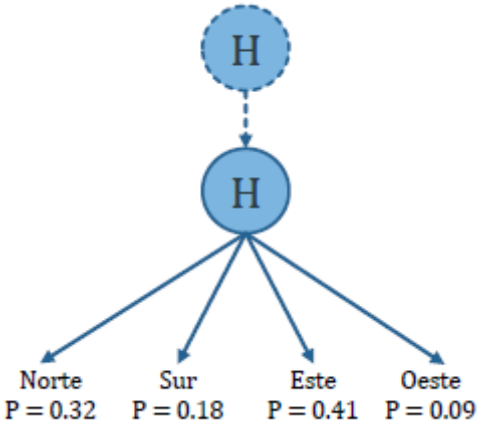

Ilustración 16: Probabilidades para cada dirección

5.4.2 Mutacion

La mutación es otra de las fases importantes durante la evolución en cualquier algoritmo genético. En esta ocasión, estas fases no ocurren en un caso real, sino que simplemente han sido añadidas para dotar al algoritmo de una mayor viveza (y porque creía que era importante que hubiese una fase de mutación durante la evolución de la población, aunque como sabemos esta fase es opcional).

La primera de las mutaciones que se lleva a cabo es sobre el tipo. Existe una cierta probabilidad (muy pequeña) de que un aminoácido pueda cambiar su tipo. He dejado esta probabilidad en 0.05 y, tras realizar algunas pruebas, creo que es un valor aceptable.

Finalmente, la segunda de las mutaciones tiene que ver con la polaridad de los enlaces de cada individuo. Como dijimos, un individuo puede crear enlaces en sus cuatro direcciones pero, para que sea una unión satisfactoria, la polaridad debe ser distinta a la del enlaces de su vecino. Pues bien, estos enlaces tienen una probabilidad de 0.1 de modificar su polaridad durante esta última fase de mutación.

5.4.3 Limitaciones de la implementación

Durante esta última sección dedicada al desarrollo e implementación del proyecto voy a hablar sobre algunas de las limitaciones que he detectado a medida que iba construyendo el proyecto.

Una de las primeras limitaciones es que los individuos no pueden rotar sobre sí mismos, esto es: un individuo no puede "girar" dentro de su posición en la matriz para tratar de buscar nuevas uniones (podría ser, por ejemplo, que debido a la polaridad no haya podido realizar ninguna unión).

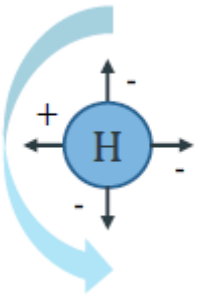

Ilustración 17: Rotación dentro de la matriz

Otra limitación de la implementación es que la matriz que inicialmente se crea por defecto tiene un tamaño $n*n$, donde n es el tamaño de la cadena tomada como *input* (esta limitación podría ser fácilmente modificada, pero a la hora del desarrollo opté por crear la matriz según esta regla).

Como he dicho muchas veces, en un caso real, el aminoácido es el que tiene la polaridad y no "sus enlaces". Nótese que en determinadas ocasiones la tarea de desarrollar proyectos que se asemejen a los casos reales es enormente grande. Así pues, de nuevo, decidí que la polaridad de cada individuo no debía estar en el propio individuo sino en cada uno de sus enlaces (para así poder moverse en una dirección u otra y poder formar nuevas uniones con sus vecinos).

5.5 Ejecución de ejecución

Pasamos a ver la última sección de esta memoria. A continuación se muestra una de las pruebas que se han realizado. Esta prueba ha sido ejecutada con la cadena de aminoácidos de tamaño 4: HHHP, luego la malla resultante es una matriz 4x4.

Si nos fijamos en lal siguiente ilustración, durante la inicialiación de la población cada individuo es colocado de manera totalmente aleatoria en la matriz. Observemos que hay individuos que son vecinos entre sí, pero la polaridad de sus enlaces no permite crear una nueva unión.

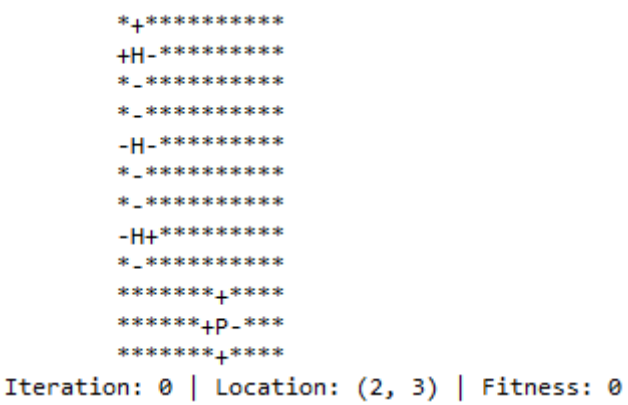

Ilustración 18: Iteración 0 de la población

Durante la iteración número 7, el enlace situado al sur del aminoácido en la posición (0, 1) cambia su polaridad de negativo a positivo. Este cambio produce una unión con su vecino situado al sur, cuyo enlace norte tiene polaridad negativa. En este punto, el algoritmo identifica al invididuo *fittest*.

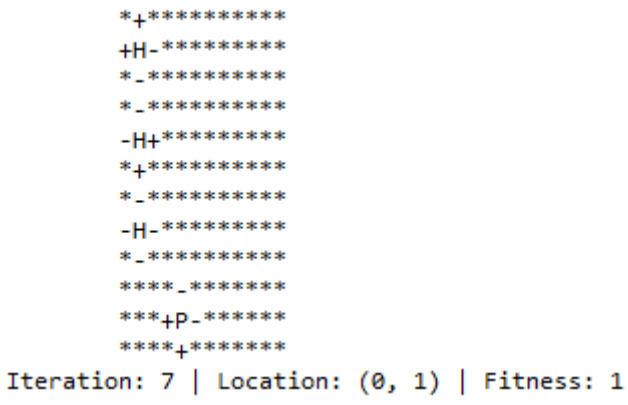

Ilustración 19: Iteración 7 de la población

Una iteración más tarde se produce un cambio significativo en la población: el aminoácido situado en la posición (1, 3) cambia su tipo, pasando a ser un aminoácido de tipo H. Esto conlleva que en las siguientes iteraciones la probabilidad de realizar nuevos enlaces crezca.

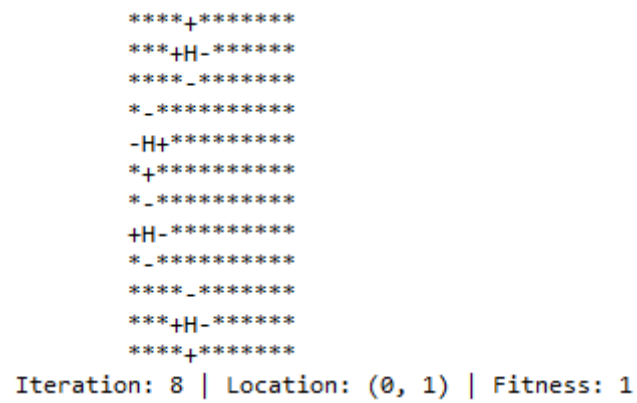

Ilustración 20: Iteración 8 de la población

Como ya se dijo, se ha optado porque el algoritmo no sea elitista. Esto quiere decir que debido a la probabilidad de movimiento que tiene cada individuo (que recordemos es menor cada vez que si fitness crece) o a las mutaciones, el fitness de determinados individuos podría decrecer. Esto es precisamente lo que ocurre durante la iteración 12: el enlace del individuo que cambió su valor en la **Ilustración 19**, vuelve a tener una polaridad negativa, rompiendo así la unión.

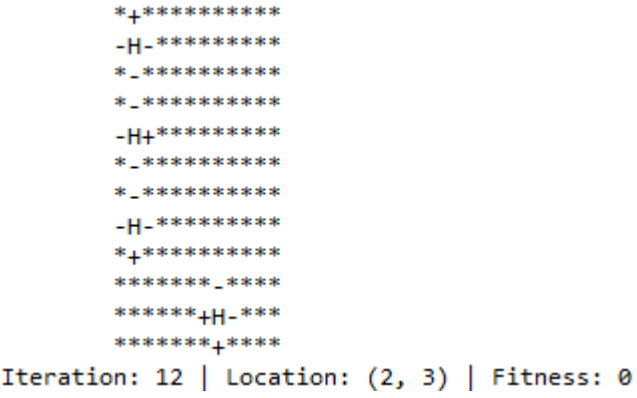

Ilustración 21: Iteración 12 de la población

En la iteración número 17 se produce otro cambio en la polaridad, que esta vez viene producido por el aminoácido situación en la posición (1, 0). Este cambio conlleva que la unión vuelva a tener éxito, aumentando el fitness del indiviuo en +1.

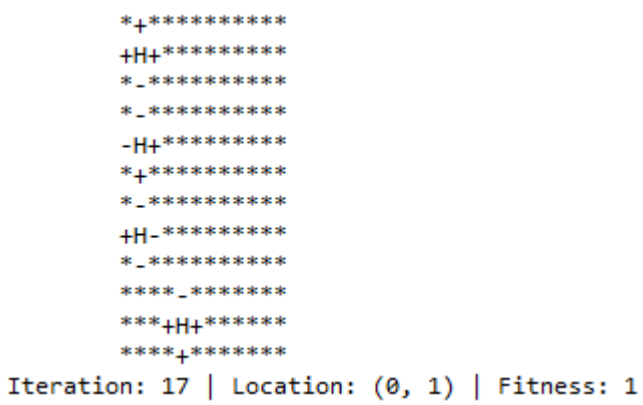

Ilustración 22: Iteración 17 de la población

Después de dos iteraciones más, la tendencia de los aminoácidos a agruparse produce una nueva unión, lo que hace que el fitness aumente hasta 2. El nuevo fittest de la población pasa a ser el individuo situado en la posición (1, 2).

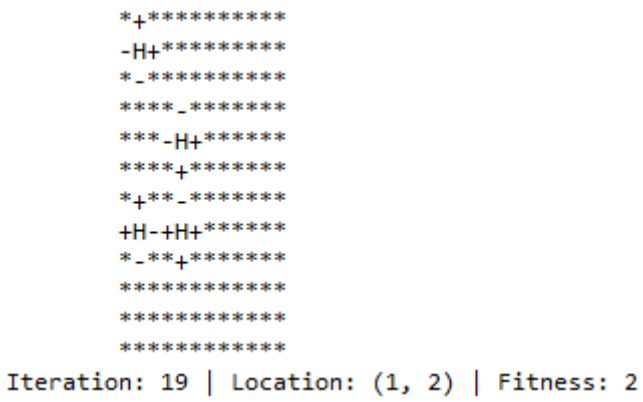

Ilustración 23: Iteración 19 de la población

Si nos fijamos en la última ilustración, todos los aminoácidos han logrado agruparse (aunque es cierto que, debido a la polaridad, no han surgido nuevas uniones).

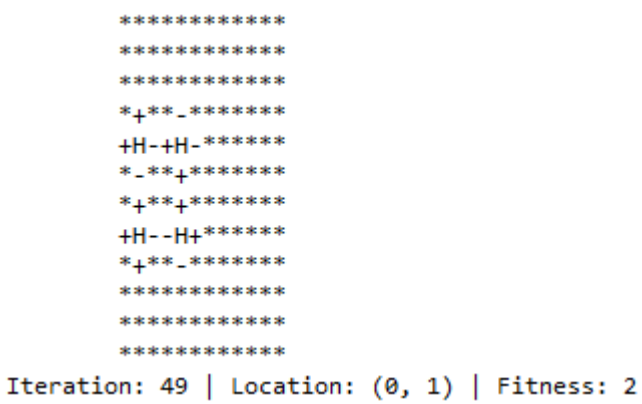

Ilustración 24: Iteración 49 de la población

6 Referencias

[**1**] Dill K. A. "*Theory for the folding and stability of globular proteins*". Biochesmistry. 1985.

[**2**] Crescenzi P. Goldman D. Papadimitriou C. Piccolboni A. Yannakakis M. "*On the complexity of protein folding*". Macromolecules. 1998.