Universidad de Valladolid

FACULTAD DE CIENCIAS

TRABAJO FIN DE GRADO

Grado en Matemáticas

ALGORITMOS HEURÍSTICOS

Autor:
Pablo Proy de Santiago

Tutor:
Víctor Gatón Bustillo

Año
2023-2024

A mi familia, por acompañarme en este camino y creer en mí. En especial a mis padres, por ser mi ejemplo a seguir y mi apoyo incondicional.

Índice general

Introducción

En el ámbito de la optimización matemática, una heurística es una técnica utilizada para resolver problemas complejos y de gran escala cuando los métodos tradicionales no son lo suficientemente precisos o lo suficientemente rápidos proporcionando una solución óptima. Para ello, se inspiran en la toma de decisiones basándose en distintos factores, como las restricciones del problema en cuestión. Por lo tanto, los algoritmos heurísticos son una clase de algoritmos matemáticos que utilizan diversas técnicas heurísticas para la resolución de problemas complejos, como por ejemplo problemas de optimización industrial o problemas NP-completos.

Mi interés particular por esta clase de algoritmos y sus diversas aplicaciones en problemas cotidianos, han motivado la realización de este Trabajo de Fin de Grado. En concreto, en esta memoria, se describen detalladamente varios algoritmos heurísticos que posteriormente se aplican a juegos combinatorios clásicos para luego analizar los resultados obtenidos. Cabe destacar, que los análisis realizados no se tratan de análisis de eficiencia, sino de análisis cualitativos, de manera que se intenta comprender en lineas generales el comportamiento de los algoritmos heurísticos presentados.

Por último, mencionar que todos todos los algoritmos diseñados se han programado con *MATLAB* (versión R2022b). Todos los experimentos se han realizado en una máquina con las siguientes especificaciones técnicas: procesador *Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz*, memoria RAM *8Gb* y sistema operativo *Windows* de 64 bits. Todo aquel interesado puede solicitar el código de los programas utilizados al autor.

Capítulo 1

Algoritmos Genéticos

En este primer Capítulo, se tratan brevemente los algoritmos evolutivos para, posteriormente, centrarse en los algoritmos genéticos. Principalmente, se introducen los aspectos más importantes de los algoritmos genéticos, así como sus ventajas e inconvenientes. Finalmente, en la última sección del Capítulo, se desarrolla y resuelve un ejemplo práctico con el objetivo de ilustrar los conceptos presentados.

1.1. Algoritmos evolutivos

Los algoritmos evolutivos son técnicas de optimización complejas que emulan la selección natural y los procesos de evolución biológica para obtener soluciones a diversos problemas. Los procesos y teorías evolutivas en los cuales se basan se podrían resumir como sigue: dada una población de una cierta especie, los individuos mejor adaptados al medio tendrán mayor probabilidad de sobrevivir y, por ende, más posibilidades de reproducirse, transmitiendo así a su descendencia las características que les permitieron sobrevivir.

El desarrollo de la teoría de estos algoritmos se vio altamente favorecida en las décadas de 1960 y 1970 por la aparición de calculadoras más potentes y computadoras con una mayor capacidad de cálculo. Fue en esta época cuando surgieron de manera independiente los tres principales enfoques sobre este tipo de algoritmos. Estos enfoques son: las *estrategias evolutivas*, la *programación evolutiva* y los *algoritmos genéticos* (ver [2]).

Con el paso de los años, dichos enfoques fueron modificados para adaptarse a las peculiaridades de los problemas a los que se aplicaban. Además, la publicación del libro *"Genetic Algorithms in Search, Optimization and Machine Learning"* por D.E. Goldberg en 1989 dio gran popularidad a los algoritmos genéticos, hasta tal punto que muchos autores utilizan el término *Algoritmo Genético* para designar a cualquier algoritmo basado en procesos evolutivos.

1.2. Descripción del algoritmo genético

Los algoritmos genéticos son, por lo tanto, una clase de algoritmos evolutivos. Más concretamente, se podrían definir como técnicas de búsqueda basadas en la selección natural y la genética (ver [3]) que,

a diferencia de los métodos de optimización clásicos, se centran en la codificación de los parámetros del problema más que en su naturaleza. En un principio, este tipo de algoritmos fue desarrollado para resolver problemas de optimización con dominios complejos, entendiéndose por dominios complejos aquellos en los cuales el espacio de búsqueda es enorme o aquellos con una función objetivo que posee numerosos óptimos locales, discontinuidades o ruido (ver [8]). En particular, estos algoritmos son muy útiles para resolver problemas de optimización combinatoria, pues este tipo de problemas suele tener un gran espacio de búsqueda.

A continuación, se introducen las primeras consideraciones a tener en cuenta a la hora de resolver un problema utilizando este tipo de algoritmos.

Los algoritmos genéticos representan las variables de decisión de un problema mediante secuencias o vectores de longitud finita, formados por valores de las distintas variables del problema en cuestión (ver Figura 1). Siguiendo la analogía con los procesos evolutivos, dichas secuencias reciben el nombre de *cromosomas* y los parámetros que las conforman se llaman *genes*.

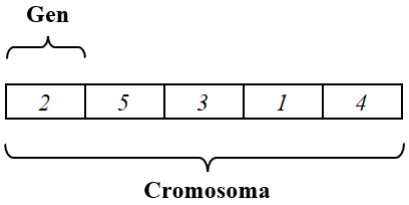

Figura 1: Representación de una variable de decisión siguiendo la analogía con los procesos evolutivos.

La cuantificación de la valía de las soluciones posee una gran importancia, pues permitirá elegir las mejores soluciones para así implementar la selección natural y evolucionar hacia unas nuevas presumiblemente de mayor calidad. Para medir dicha valía, se suelen emplear funciones objetivo que modelicen de la mejor manera posible el problema tratado, aunque también se podrían utilizar funciones subjetivas que dependiesen del criterio del usuario.

Otro aspecto relevante de esta clase de algoritmos es la definición de la población. En contraposición con las técnicas de optimización tradicionales, los algoritmos genéticos necesitan la existencia de un conjunto de soluciones candidatas, que recibe el nombre de *población*. El tamaño de esta población es variable y es normalmente elegido por el usuario. Su tamaño es sumamente importante, pues una población pequeña podría derivar en una pronta convergencia del algoritmo, mientras que una población excesivamente grande podría incurrir en un alto coste computacional.

Tras realizar la primera fase de modelización del problema, es decir una vez adaptado a la representación cromosómica y elegido el método de cuantificación de las soluciones, da comienzo la fase principal de los algoritmos genéticos: la fase de evolución. La evolución puede variar en función del problema o requerir de una mayor complejidad para satisfacer ciertos requisitos en las soluciones. Sin embargo, un esquema común como el presentado en [3] sirve como punto de partida para la inmensa mayoría de ocasiones. Dicho esquema se compone de los siguientes puntos:

1. **Inicialización:** En esta primera etapa, se genera la población inicial de posibles soluciones. Normalmente este proceso se realiza de forma aleatoria, aunque se podría utilizar (si se dispone de ella) información relevante sobre el problema para tratar de agilizar la búsqueda del algoritmo.

2. **Evaluación:** En esta fase se evalúan las soluciones que forman la población, ya sea inicial o de descendencia, para calificarlas en función del criterio elegido, normalmente una función objetivo.

3. **Selección:** Mediante la selección, se escogen las soluciones con una mayor aptitud con el fin de que dichas soluciones perduren, dándoles así preferencia sobre soluciones de menor calidad.

4. **Cruce:** Se combinan las características de al menos dos soluciones preseleccionadas, que reciben el nombre de *progenitores* o *padres*, para obtener nuevas soluciones, llamadas *descendencia* o *hijos*, que se espera sean de una mejor calidad.

5. **Mutación:** Tras generar cada descendiente, se decide con una cierta probabilidad si dicha solución candidata es modificada o no.

6. **Reemplazamiento:** Una vez generada la población de descendencia mediante el cruce y la mutación, se reemplaza la población original por esta nueva.

Los pasos 2 a 6 se pueden repetir tantas veces como se desee o hasta que se alcance cierta condición de finalización. En ocasiones, la condición en cuestión es un número máximo de iteraciones, también llamadas *generaciones*; un tiempo máximo de computación, si se esta utilizando una computadora; la obtención de una solución de cierta calidad...

1.3. Operadores evolutivos fundamentales

En la Sección anterior, se presentó un esquema básico para la creación de un algoritmo genético. Dentro del esquema evolutivo hay varios puntos cruciales para cualquier algoritmo genético, comúnmente conocidos como operadores fundamentales de los algoritmos genéticos. Estos puntos claves son los métodos de selección, cruce, mutación y reemplazamiento.

Los operadores fundamentales juegan un importante papel en la modelización del problema, en especial el cruce y la mutación. Esto se debe a que en la inmensa mayoría de algoritmos genéticos, estos dos operadores son los únicos operadores de variación, es decir los operadores encargados de producir nuevas soluciones e introducir diversidad a la población.

A priori, no se conoce cuáles son los operadores que mejores resultados proporcionan, por lo que habitualmente se prueba la efectividad de diferentes combinaciones de métodos hasta encontrar la más apta. Este proceso se puede ver agilizado si previamente se ha estudiado en profundidad el problema tratado y la primera fase de la modelización se ha realizado correctamente.

A continuación, se detallan los operadores fundamentales y sus distintas variantes.

1.3.1. Selección

El primer operador fundamental y la primera fase relevante del proceso evolutivo en los algoritmos genéticos es la selección. Su principal cometido es el de elegir las soluciones con las mejores aptitudes, con el fin de que puedan generar descendientes aun más aptos.

Un concepto de suma importancia a la hora de elegir un método de selección es la *presión de selección*, que se puede definir de varias formas, una de ellas la siguiente: Asumiendo que la mejor solución de la población se reproducirá más y más rápido que las demás, hasta que en cierta generación la población esté formada únicamente por copias de dicha solución, se define el tiempo de dominio t_d como el tiempo esperado que transcurre hasta que suceda lo anterior. La presión de selección es inversamente proporcional a t_d (ver [2]).

Por lo tanto, la elección de un método que conlleve una alta presión de selección podría derivar en una *convergencia prematura* del algoritmo. Esta convergencia se produce cuando una solución, llamada superindividuo, que generalmente es cercana al óptimo, se reproduce mucho más rápido que las demás. De esta forma, la búsqueda se convierte en una búsqueda local alrededor de dicho superindividuo, alcanzándose frecuentemente máximos (o mínimos) locales en vez de absolutos.

La mayoría de los métodos de selección se pueden clasificar en dos subgrupos, métodos de selección proporcionales a la aptitud y métodos de selección ordinal (ver [2], [3] y [9]).

A continuación se detallan los principales métodos de selección proporcionales a la aptitud de los individuos.

- **Selección de ruleta:** La idea fundamental de este método es asignar a cada solución de la población una ranura de una ruleta circular. Además, el tamaño de la ranura será proporcional a la aptitud de la solución, medida en términos de la función objetivo. Esta ruleta sesgada se girará tantas veces como soluciones se deseen seleccionar.

  Una manera sencilla de implementar este método es la siguiente. Dada una función objetivo f y una población de n soluciones candidatas, se evalúa la aptitud de dichas soluciones y se denota el resultado por f_i para $i = 1, 2, ..., n$. Después, se calcula la probabilidad p_i de seleccionar la solución i, es decir el tamaño de la ranura de i. De esta forma se obtiene que $p_i = f_i / \sum_{j=1}^{n} f_j$ para cada $i = 1, 2, ..., n$. A continuación, se calcula la probabilidad acumulada q_i para cada solución, es decir $q_i = \sum_{j=1}^{i} p_j$. Finalmente, se genera un número aleatorio $r \in (0, 1]$, de tal forma que si $r < q_1$ la solución elegida es la 1, y si $q_{i-1} < r \leq q_i$ entonces la solución elegida es la i.

  La selección de ruleta tiene una alta varianza, por lo que es posible, para un número finito de generaciones, que una solución con una buena evaluación nunca sea seleccionada. También podría suceder que, en casos extremos, una solución de baja calidad fuese seleccionada tantas veces como descendientes hubiese en una población. Una posible forma de contrarrestar este último fenómeno es tomar una población lo suficientemente grande.

- **Selección estocástica universal:** La selección estocástica universal surge con el objetivo de reducir la varianza de la selección de ruleta. Al igual que en la selección de ruleta, a cada solución candidata se le asigna una ranura de tamaño proporcional a su aptitud. Sin embargo, en este caso no se gira la ruleta m veces para elegir m soluciones, sino que se gira una única vez en la que se eligen los m individuos. Para ello se dota a la ruleta de m punteros equiespaciados.

  La manera mas sencilla de implementar este método en casos prácticos es la siguiente. En vez de representar las soluciones como ranuras de una ruleta circular sesgada, se representan como zonas de un segmento. El tamaño de estas zonas seguirá siendo proporcional a la aptitud de cada

solución y habrá tantas zonas como soluciones en la población. Para escoger las m soluciones en un solo "*giro*", se elige aleatoriamente un punto del segmento a partir del cual se generan $m-1$ punteros equiespaciados. Finalmente, se eligen las soluciones que poseen al menos uno de los punteros en su zona. Si en la zona de una solución hubiese más de un puntero, esta solución se elige tantas veces como punteros haya.

En este caso, puesto que la varianza de este método es menor, la mejor solución será seleccionada con total certeza siempre que $m \geq n$, donde n representa el tamaño de la población y m el número de soluciones seleccionadas, ya que la amplitud del intervalo del mejor (o uno de ellos, si hubiese varios) individuo p_i, definido de igual forma que en el caso de la selección de ruleta, está acotado inferiormente por $1/n$.

En la Figura 2 se muestra un ejemplo de este método de selección, en el que se ha tomado una distancia aleatoria con respecto al inicio del segmento a partir de la cual se han situado siete punteros equiespaciados. De esta forma, se seleccionan los individuos 8, 5 y 4 una vez y los individuos 10 y 9 dos veces.

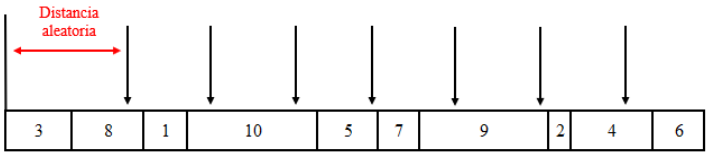

Figura 2: Selección de siete individuos usando la selección estocástica universal.

Dentro de la selección ordinal, destacan los siguientes métodos.

- **Selección por torneo:** La selección por torneo surge como alternativa a los métodos de selección proporcionales a la aptitud. Este método, en su variante más simple, consiste en elegir k soluciones al azar de la población, ya sea con o sin reemplazamiento, que se enfrentarán en un torneo. La solución más apta ganará el torneo y será elegida como solución parental. Por lo tanto, serán necesarios n torneos para elegir n soluciones parentales. Al igual que en métodos anteriores, la aptitud de las soluciones se mide mediante una función objetivo.

  Normalmente el valor de k es igual a 2. En estos casos, algunos autores llaman al método *torneo estocástico*.

- **Selección por truncamiento:** Este método de selección es uno de los más simples y fáciles de implementar en la inmensa mayoría de problemas, por lo que generalmente es la primera opción en la búsqueda del método de selección óptimo.

  La selección por truncamiento consiste en seleccionar las m mejores soluciones, donde m suele ser un parámetro a elección del usuario. Habitualmente, en vez de especificar el número exacto de padres a seleccionar, esta cantidad se expresa mediante un porcentaje sobre el total de la población. Si tras elegir m soluciones parentales se generan m_d nuevas soluciones, cada padre generará m_d/m soluciones descendientes. Nuevamente, para elegir las mejores soluciones se debe evaluar cada uno de los individuos de la población.

Pese a ser un método de fácil implementación, su mayor inconveniente es la alta presión de selección, que podría producir una convergencia prematura, especialmente cuando el tamaño de la población es grande.

1.3.2. Cruce

Una vez seleccionados los individuos que generarán la descendencia, hay que decidir el método de cruce o *crossover* que se utilizará. Este es uno de los puntos cruciales en cualquier algoritmo genético, pues la elección de un buen método puede significar la obtención de una solución próxima a la óptima en un tiempo razonablemente bajo.

Generalmente, en la mayoría de los métodos de recombinación intervienen únicamente dos soluciones parentales que dan lugar a dos descendientes. Sin embargo, en la recombinación pueden intervenir más de dos soluciones o incluso toda la población y pueden generarse uno o más de dos descendientes. En esta Sección, todos los métodos estudiados producirán dos descendientes a partir de dos padres.

Tras seleccionar dos progenitores, estos se recombinarán con cierta probabilidad p_c, conocida como probabilidad de cruce. Es decir, se genera aleatoriamente un número $r \in [0, 1]$, si $r \leq p_c$ se realiza la recombinación mientras que si $r > p_c$ las soluciones descendientes son idénticas a las parentales. Puesto que en la mayoría de algoritmos genéticos el cruce es el principal método de variación de las soluciones, la probabilidad de cruce suele ser igual a 1, de forma que siempre se lleva a cabo la recombinación.

A continuación, se desarrollan varios métodos de cruce en su configuración más genérica (ver [2] y [3]), que pueden ser modificados en función del problema estudiado.

- **Cruce de k puntos:** En este método, dadas dos soluciones parentales, se eligen aleatoriamente k puntos de cruce que serán iguales en ambos padres. Además, dichos puntos de cruce deben ser distintos entre si y nunca la última o primera posición del cromosoma. Una vez elegidos los k puntos, se intercambian los genes de los padres entre cada par de puntos de cruce, generándose así dos nuevas soluciones.

  Normalmente, este método se utiliza estableciendo 1 o 2 puntos de cruce, como podemos ver en la Figura 3. En este ejemplo los puntos de cruce se han representado con trazo discontinuo rojo.

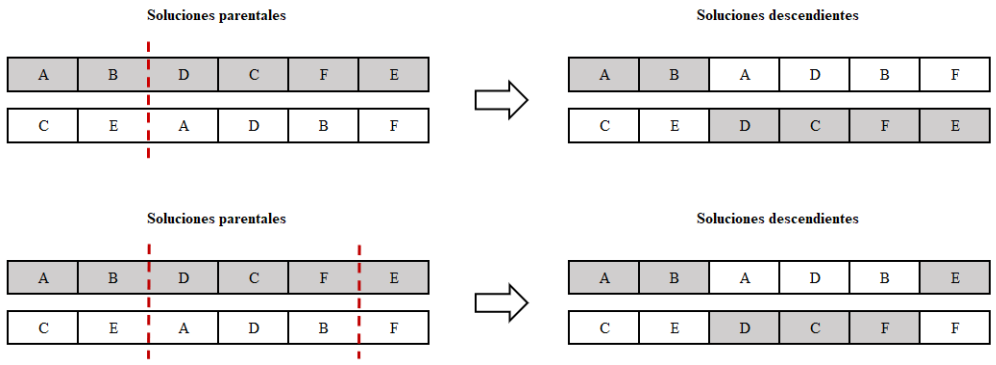

Figura 3: Arriba, cruce de 1 punto. Abajo, cruce de 2 puntos.

- **Cruce uniforme:** En este caso, dadas dos soluciones parentales se intercambian los genes de cada

una de ellas con cierta probabilidad p, originándose dos soluciones descendientes. Normalmente, el valor de p se fija en 0.5. En la Figura 4 se puede ver un ejemplo de este método donde, aleatoriamente, se obtuvo que se intercambiaban los genes 3, 5 y 6.

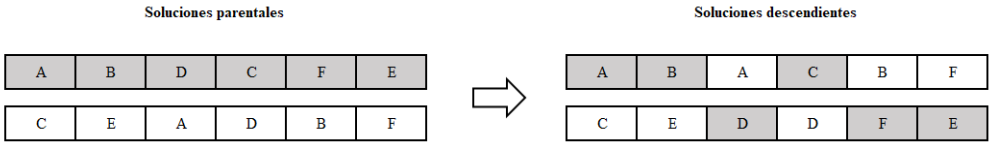

Figura 4: Cruce uniforme para $p = 0.5$.

Los dos anteriores métodos de recombinación, pese a ser los más simples y fáciles de implementar, no son válidos para todo tipo de problemas. En particular, no son válidos en problemas cuyas soluciones o cromosomas representan permutaciones de n elementos, pues frecuentemente se genera descendencia incoherente con las características del problema estudiado.

Un claro ejemplo de este tipo de problemas surge en el *Problema del Viajante*. En este problema, se deben visitar n ciudades distintas una única vez, por lo que las soluciones candidatas (que indican el orden de visita de las ciudades) se corresponden con las posibles permutaciones de las n ciudades (ver Sección 1.5.1 para más detalles). Por lo tanto, si el método de cruce elegido fuese el cruce de k puntos o el cruce uniforme, sería altamente probable obtener una solución descendiente en la cual se visitase una ciudad más de una vez, incumpliéndose así la restricción principal del problema.

En la Figura 5 se puede ver un ejemplo en el que utilizando el cruce de 1 punto se obtiene un descendiente que visita dos veces las ciudades 3 y 1, y otro descendiente que visita dos veces las ciudades 5 y 2.

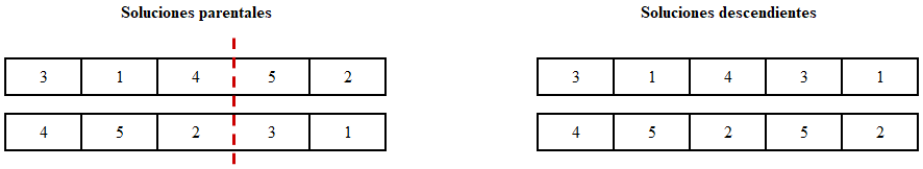

Figura 5: Ejemplo de cruce no válido de 1 punto para el *Problema del Viajante* con $n = 5$ ciudades.

Por lo tanto, si el problema tratado tiene sus soluciones codificadas en forma de permutaciones y se desea utilizar alguno de los dos anteriores métodos, hay que aplicar mecanismos de corrección específicos para corregir las posibles soluciones incoherentes. Sin embargo, este proceso puede requerir mecanismos complejos o muy costosos en términos computacionales. Por ende, la mejor opción es utilizar métodos de cruce especialmente diseñados para este tipo de problemas, los cuales siempre generan soluciones válidas. Cabe destacar que, pese a estar especialmente diseñados para problemas con soluciones codificadas en forma de permutación, también se pueden aplicar a otro tipo de problemas.

A continuación, se presenta alguno de estos métodos:

- **Cruce uniforme ordenado:** Dadas dos soluciones parentales P_1 y P_2, se genera aleatoriamente un vector plantilla con tantas posiciones como genes posean los padres. Dicho vector plantilla

contendrá únicamente los valores 0 y 1. Este método de recombinación, generará dos soluciones descendientes H_1 y H_2.

Para generar H_1 se copian los genes de P_1, manteniendo la posición, que se corresponden con un 1 en el vector plantilla. Sin embargo, H_1 estará incompleto, por lo que se toman los genes de P_1 que se corresponden con el valor 0 del vector plantilla y se ordenan según el orden de aparición en P_2, para posteriormente rellenar con ellos las posiciones vacías de H_1. El procedimiento para generar H_2 es análogo, basta intercambiar P_1 por P_2 en la explicación anterior.

En la Figura 6 se puede ver un ejemplo de este método de recombinación. En dicho ejemplo, para la creación de la solución H_1, los genes que se copian de P_1 haciendo uso del vector plantilla son *A, C, F*. Los genes sobrantes *B, D, E*, se ordenan según el orden de aparición en P_2, resultando en la lista *E, D, B*. Finalmente, se colocan en ese orden en las posiciones vacías de H_1.

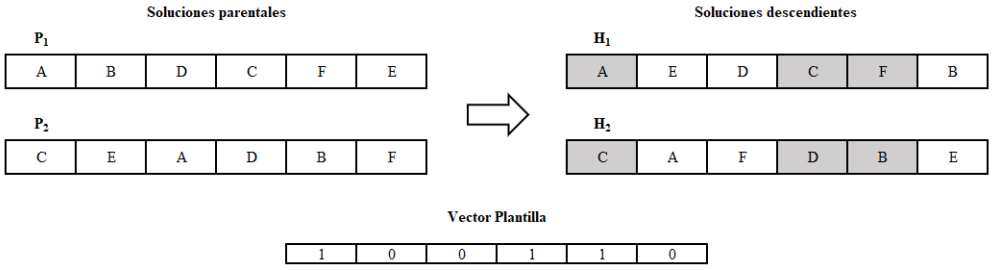

Figura 6: Cruce uniforme ordenado.

- **Cruce ordenado:** El cruce ordenado o cruce basado en el orden, no es más que una modificación del cruce uniforme ordenado. Nuevamente, la creación de las soluciones descendientes es simétrica, por lo que únicamente se detallará el caso de una de ellas. Sin embargo, el proceso de recombinación es ligeramente diferente.

  Dadas dos soluciones parentales P_1 y P_2, se escogen aleatoriamente dos puntos de cruce distintos de la primera y la última posición y nunca iguales. Para la obtención de H_1, se copian los genes de P_1 que están entre los dos puntos de cruce, manteniéndose la posición que tenían en P_1. Posteriormente, se toman los genes de P_1 que no se encuentran entre las posiciones de cruce y se ordenan por orden de aparición en P_2. Finalmente, dichos genes ya ordenados se copian en las posiciones vacías de H_1 comenzando por aquellas que se encuentran después del segundo punto de cruce.

  En la Figura 7 se muestra un ejemplo práctico de este método. Para obtener H_1, se han copiado los genes *D, C, F*, que son los cuales se encuentran entre los dos puntos de cruce de P_1. Tras ordenar según el orden de aparición en P_2 el resto de genes de P_1, se copian en H_1 empezando por las posiciones a la derecha del segundo punto de cruce. En este caso, los genes restantes de P_1 ya ordenados son *E, A, G, B*, que se copian en las posiciones 6, 7, 1 y 2 de H_1 respectivamente.

- **Cruce parcialmente ordenado:** Este método de recombinación se caracteriza por preservar órdenes dentro del cromosoma. Además, es el método más utilizado en problemas cuyas soluciones se codifican en forma de permutación.

  Para generar las dos soluciones descendientes H_1 y H_2 a partir de los padres P_1 y P_2, se elige en primer lugar dos puntos de cruce aleatorios de igual forma que en métodos anteriores. A

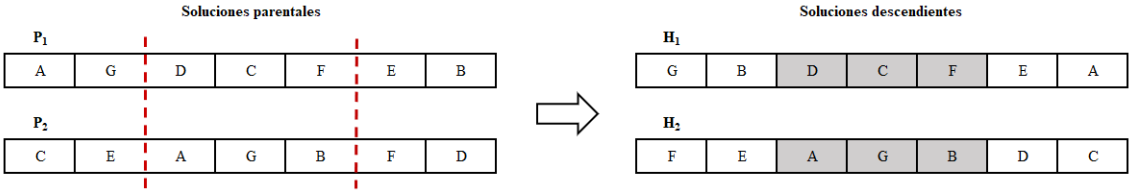

Figura 7: Cruce ordenado.

continuación, se establece una correspondencia entre los genes de ambos padres comprendidos entre los puntos de cruce. Sea X uno de los genes de P_1 comprendidos entre dichos puntos de cruce y supóngase que se corresponde con el gen Y de P_2. Entonces, en H_1 se copia el gen Y en la posición que ocupa X en P_1 y el gen X en la posición que ocupa Y en P_1. Por otro lado, en H_2 se copia el gen X en la posición que ocupa Y en P_2 y el gen Y en la posición que ocupa X en P_2. Este proceso se lleva a cabo con todos los genes comprendidos entre ambos puntos de cruce, manteniéndose los genes parentales no intercambiados en sus posiciones originales, y generándose así las dos nuevas soluciones H_1 y H_2.

En la Figura 8 se representa un ejemplo de este método. Dados dos padres P_1 y P_2 y elegidos aleatoriamente los dos puntos de cruce, se procede a recombinar los genes de ambos padres para generar la descendencia. En P_1, entre los puntos de cruce, se encuentran los genes *D, C, F*, que se corresponden con los genes *A, G, B* de P_2 respectivamente. Por lo tanto, los genes D y A, C y G, F y B se intercambian tanto en la solución parental P_1 como en P_2, originándose simultáneamente los descendientes H_1 y H_2.

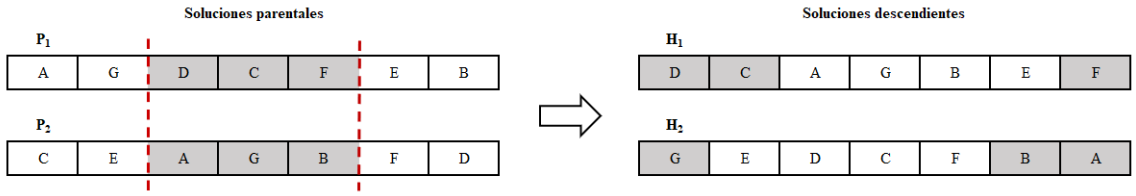

Figura 8: Cruce parcialmente ordenado.

- **Cruce cíclico:** Dadas dos soluciones parentales P_1 y P_2 se puede construir una permutación

$$\tau = \begin{pmatrix} P_1(1) & ... & P_1(n) \\ P_2(1) & ... & P_2(n) \end{pmatrix}$$

de n elementos, donde $P_1(i)$ y $P_2(i)$ representan el gen almacenado en la posición i de cada padre para $i = 1, 2, ..., n$.

Para obtener la solución H_1 se procede de la siguiente manera. En primer lugar, se toma $P_1(1)$, es decir el primer gen de P_1, y se forma su ciclo σ siguiendo la ruta dada por la matriz de permutación τ. A continuación, los genes que forman el ciclo σ, se copian en H_1 en la misma posición que ocupan en P_1. Finalmente, si en H_1 quedasen posiciones vacías tras copiar el ciclo σ, se llenarían con los genes correspondientes de P_2. La creación de H_2 es análoga, basta con

definir τ de la siguiente forma

$$\tau = \begin{pmatrix} P_2(1) & ... & P_2(n) \\ P_1(1) & ... & P_1(n) \end{pmatrix}$$

y seguir el mismo proceso.

En la Figura 9 se puede ver un ejemplo de este método. En concreto, se detalla la creación de H_1 dadas las soluciones parentales P_1 y P_2. En este caso, el primer gen de P_1 es D. Fácilmente se observa que a partir de este gen se forma el ciclo $\sigma = (D, E, H, B)$. Por lo tanto, los genes *D, E, H, B* se copian en las posiciones 1, 5, 3 y 4 de H_1 respectivamente. Las posiciones restantes de H_1 (posiciones 2, 6, 7, 8) se completan con los genes correspondientes de P_2, es decir los genes *G, A, F, C*.

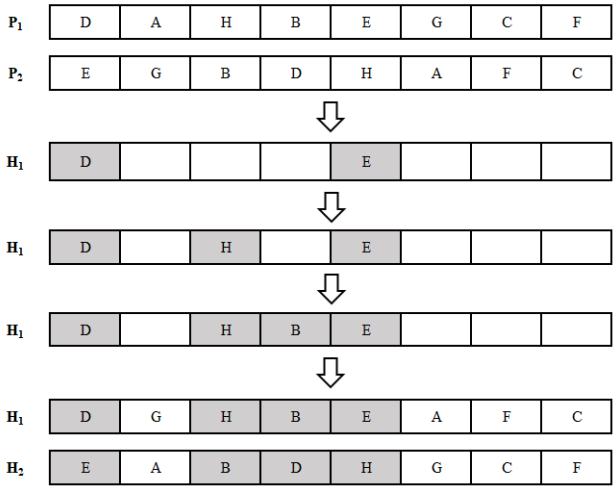

Figura 9: Cruce cíclico. Pasos para la obtención de H_1 (siendo el caso de H_2 análogo).

Además de los anteriores métodos de cruce específicos para problemas con soluciones codificadas en forma de permutación, existe otro método ampliamente utilizado en el *Problema del Viajante*. Este método recibe el nombre de *cruce heurístico*. A grandes rasgos, se puede considerar como un caso particular del cruce cíclico en el que en vez de formar un ciclo siguiendo como patrón una cierta matriz de permutación, este se forma tratando de minimizar el coste final. En la Sección 1.5.2 se detallará el funcionamiento de este método aplicándolo al *Problema del Viajante*.

1.3.3. Mutación

El proceso de mutación consiste sencillamente en modificar, bajo una cierta probabilidad, un individuo para generar uno nuevo que lo reemplazará. Es decir, se realiza una búsqueda local alrededor de la solución que va a mutar. La probabilidad de mutación δ, también denotada por p_m, proporciona la tasa de individuos mutados en una población.

En la mayoría de implementaciones de algoritmos genéticos, se ha realizado previamente un proceso de recombinación, por lo que la mutación se considera un operador de variación secundario. Sin embargo, en las estrategias evolutivas este es el principal operador de variación y, en ocasiones, el único (en este caso, la tasa de mutación sería del 100 %, es decir $\delta = 1$).

Volviendo a los algoritmos genéticos, el principal cometido de la mutación es asegurar diversidad en la población, ayudando así a explorar el espacio de búsqueda. Además, es de gran utilidad cuando las soluciones están próximas al máximo (mínimo) absoluto pues, al modificarse levemente dichas soluciones, se podría alcanzar exactamente el máximo (mínimo) absoluto del problema en cuestión.

En estos casos la probabilidad de mutación es variable, siendo un parámetro a elección del usuario. Una tasa de mutación elevada asegura diversidad en la población, pero si es demasiado alta podría conllevar efectos negativos, pues las soluciones mutadas distarían mucho de la solución que las generó, aumentando el coste computacional y pudiéndose retrasar la convergencia hacia una solución óptima. Típicamente δ varía entre 0.01 y 0.1 (ver [2]).

Como sucede con los métodos de cruce, existen numerosas técnicas de mutación. A continuación se presentan las más habituales:

- **Intercambio de posiciones:** Suponiendo que las soluciones del problema están representadas mediante cromosomas, este método de mutación consiste en escoger dos o más posiciones de un cromosoma e intercambiar los genes que contienen. Normalmente, el número de posiciones que se intercambia es dos y la manera de elegirlas suele ser al azar, aunque también se podría especificar concretamente las posiciones que se desea permutar. En la Figura 10 se muestra un ejemplo de este método.

Figura 10: Mutación por intercambio de las posiciones 1 y 5.

- **Inversión de bits:** Este tipo de mutación es habitual en problemas donde los cromosomas están codificados en forma binaria, es decir sus genes únicamente pueden tomar los valores 0 y 1. Esta técnica consiste en recorrer completamente el cromosoma cambiando los genes "0" por "1" y viceversa. En la Figura 11 se ilustra un ejemplo de este tipo de mutación.

  Este método, se podría adaptar a problemas cuyos cromosomas no estén codificados de forma binaria. Para ello, se recorrería el cromosoma en busca de los genes con valor "a" y se sustituiría por el valor "b". En este caso los valores a y b serían parámetros del algoritmo.

Figura 11: Mutación por inversión de bits.

Al igual que con lo sucedido durante el cruce, el método de mutación se debe elegir con precaución, pues se podrían originar soluciones incoherentes con la naturaleza del problema estudiado.

1.3.4. Reemplazamiento

Una vez llevado a cabo el cruce y la mutación, se debe decidir como acoplar la población de descendencia a la población parental. Este acoplamiento tiene como único objetivo asegurar la mayor calidad posible

de la población, con vistas a refinar aun más las soluciones que se crearán en próximas generaciones. En [2] y [3] se mencionan los principales métodos de reemplazamiento, que son los siguientes:

- **Reemplazamiento generacional**: Este mecanismo es el más simple de todos, pues consiste únicamente en sustituir toda la población parental por la población de descendencia. Es decir, la población parental de la generación i, estará formada en su totalidad por las soluciones creadas en la generación i, o lo que es lo mismo por la población de descendencia de la generación i.
- **Reemplazamiento de estado estable:** En este caso, se escogen n soluciones formadas en la fase de cruce para sustituir n soluciones de la población. Generalmente, el número n de soluciones eliminadas y sustituidas es un parámetro de entrada en el algoritmo. Además, el valor de n suele ser menor estricto que el número de individuos que forman la población. Si en la situación anterior se diese la igualdad, se trataría del caso de reemplazamiento generacional.

  En este método, se debe indicar el criterio usado para eliminar e introducir las n soluciones. Frecuentemente se recurre a criterios de calidad, sustituyéndose los n miembros de la población parental de menor calidad por los n individuos de mayor calidad de la población de descendencia; criterios de aleatoriedad, eligiendo aleatoriamente las n soluciones a reemplazar; o criterios de diversidad sustituyéndose las soluciones parentales que han intervenido en el cruce por soluciones descendientes.

A continuación, se detallan otros dos métodos de reemplazamiento ampliamente utilizados que se podrían considerar casos particulares del reemplazamiento de estado estable.

- **Reemplazamiento de estado estable sin repetición:** Este método es idéntico al de estado estable, salvo por el hecho de que, antes de introducir una solución, se verifica que no exista una igual en la población parental.
- **Reemplazamiento elitista:** Este método elimina y sustituye n soluciones de la población parental siguiendo el criterio de calidad presentado en el reemplazamiento de estado estable.

Inicialmente, en la gran mayoría de problemas se elige el reemplazamiento generacional, pues es sencillo de implementar y tampoco requiere añadir condicionantes o parámetros adicionales.

Sin embargo, el principal inconveniente de este método es que las soluciones descendientes no siempre son más aptas que las soluciones parentales, fallando así el principio de evolución que busca una mejora de los individuos en cada generación. Para solucionar este inconveniente, se utilizan los métodos de reemplazamiento de estado estable y sus variantes.

Utilizar el reemplazamiento de estado estable sin repetición proporciona una mayor variedad en las soluciones que forman la población, explorándose en mayor medida el espacio de búsqueda. No obstante, la única forma de garantizar una mejora en la calidad generacional es emplear el reemplazamiento elitista. De esta forma se asegura la prevalencia de las $n_{pob} - n$ mejores soluciones parentales, donde n_{pob} representa el tamaño de la población en cada generación y n el número de soluciones sustituidas.

El reemplazamiento elitista se postula como uno de los mejores métodos de reemplazamiento, pues mejora notablemente el rendimiento del algoritmo e introduce a la vez diversidad en la población.

Pese a todo, su principal inconveniente es la posibilidad de una convergencia prematura, es decir considerar como máximo (mínimo) absoluto del problema un máximo (mínimo) relativo y centrarse en una búsqueda local alrededor de dicha solución.

Por lo tanto, no existe un método de reemplazamiento mejor que los demás, pues la decisión de emplear uno u otro depende principalmente de las características del problema a resolver o de las condiciones de calidad (medida en términos de la función objetivo) que se espera satisfaga el resultado final porporcionado por el algoritmo genético.

1.4. Convergencia de los algoritmos genéticos

Para el desarrollo de esta Sección, y el planteamiento de los siguientes resultados teóricos, se considera un problema de optimización genérico modelizado de la siguiente forma.

Sean $b \in \mathbb{B}^l = \{0,1\}^l$ y $f(b)$ una función no constante donde $0 < f(b) < \infty$ para todo $b \in \mathbb{B}$. Supóngase (ver [4]) que se desea resolver el problema máx $\left\{f(b) : b \in \mathbb{B}^l\right\}$ mediante un algoritmo genético, donde cada $b \in \mathbb{B}^l$ es un individuo y sus componentes son los distintos genes. Sea n el tamaño permitido de la población (generalmente $n \ll 2^l$) para la ejecución del algoritmo genético y sea $t = 0, 1, 2, ...$ el número de generaciones calculadas.

Dada una generación concreta t del algoritmo genético, la población de dicha generación puede representarse por $M(t) \in \mathbb{B}_{n\times l}$, donde $\mathbb{B}_{n\times l}$ representa las matrices de tamaño $n \times l$ con valores en $\{0,1\}$ y donde cada una de las filas de $M(t)$ representa a un individuo de la generación t y las distintas columnas a los diferentes genes de cada individuo de la población.

Definiendo la proyección $\pi_k(M(t)) = (m_{k,1}, ..., m_{k,l})$ donde $k \in \{1, ..., n\}$ y teniendo en cuenta la notación anterior, se puede definir la convergencia de un algoritmo genético de la siguiente forma (ver [4]).

Definición 1.1. *Sea* $Z_t = \text{máx}\left\{f\left(\pi_k\left(M(t)\right)\right) : k = 1, ..., n\right\}$ *una sucesión de variables aleatorias que representan la mejor aptitud dentro de la población de la generación* t*. Entonces se dice que un algoritmo genético converge hacia el óptimo global, si y solo si,*

$$\lim_{t\to\infty} P\{Z_t = f^*\} = 1,$$

donde $f^* = \text{máx}\left\{f(b) : b \in \mathbb{B}^l\right\}$ *es el valor óptimo global del problema.*

Cabe destacar, que se desconoce una teoría global de convergencia para los algoritmos genéticos, y el estudio de la misma se centra más en las propiedades que deben satisfacer los operadores fundamentales (en especial los de selección, cruce y mutación) para garantizar dicha convergencia (ver [2]).

Por ejemplo, dado un algoritmo genético para resolver el problema planteado en esta Sección, que emplee la selección de ruleta, la mutación por inversión de bits y un método de cruce cualquiera, se verifican, utilizando un análisis de cadenas de Markov, los siguientes enunciados:

Teorema 1.1. *El algoritmo genético con las anteriores características que mantenga la mejor solución encontrada a lo largo del tiempo (en cada generación) después de la selección, converge hacia el óptimo global.*

Teorema 1.2. *El algoritmo genético con las anteriores características que mantenga la mejor solución encontrada a lo largo del tiempo (en cada generación) antes de la selección, converge hacia el óptimo global.*

La demostración de ambos teoremas se puede consultar en [4].

Nótese que los *Teoremas 1.1 y 1.2* únicamente imponen que la mejor solución encontrada hasta el momento se mantenga a lo largo del tiempo, lo cual no implica que dicha solución intervenga en el cruce. Además, no se ha especificado un método de cruce en concreto, ya que no afecta al estudio de la convergencia (ver [4]).

Conviene señalar que, en el mismo artículo se prueba que, aunque el tiempo de transición esperado entre el estado inicial i y otro cualquiera j es finito independientemente de los estados i y j, si no se mantiene el mejor individuo a lo largo del tiempo el algoritmo genético no converge hacia el óptimo global.

Es decir, si se permite que el algoritmo genético descrito funcione indefinidamente y no se almacena el mejor individuo de cada generación, aunque la solución óptima aparezca infinitas veces a lo largo de distintas generaciones, la probabilidad de que, habiendo estado presente en alguna generación, se pierda en generaciones posteriores, es siempre estrictamente positiva.

Finalmente, en [5] se presentan más condiciones de convergencia para estos algoritmos en espacios más generales y en [11] se analizan otras versiones concretas de algoritmos genéticos. También se pueden encontrar variaciones, análisis empíricos y discusiones teóricas de algoritmos genéticos aplicados al *Problema del Viajante* en [6].

1.5. Aplicación: El Problema del Viajante

Para concluir este capítulo, se resuelve un problema práctico diseñando un algoritmo genético con el fin de ejemplificar los conceptos estudiados. El problema a resolver es el *Problema del Viajante*.

Posteriormente, se analizan los resultados y se realizan modificaciones al algoritmo para estudiar su repercusión.

1.5.1. Descripción y particularidades

El *Problema del Viajante* es un problema clásico de optimización combinatoria ampliamente estudiado y con numerosas variantes.

El enunciado que se utiliza es el siguiente: *Un viajero desea visitar una serie de ciudades, todas ellas distintas e interconectadas dos a dos, pasando una única vez por cada ciudad y de manera que la distancia total recorrida sea mínima. Hállese la solución o soluciones que minimicen dicha distancia.*

En un principio puede parecer un problema sencillo, pues con un número pequeño de ciudades la solución se puede hallar calculando la distancia recorrida en cada una de las posibles rutas. Suponiendo que n es el numero de ciudades a visitar, se tendrían $n!$ combinaciones posibles, pero como el sentido en el que se recorren las ciudades es indiferente, el espacio de búsqueda se reduce a $\frac{n!}{2}$ rutas. Por lo

tanto, si se desean visitar 4 ciudades es factible comprobar las 12 posibles rutas, mientras que si se visitasen 20 ciudades, se deberían analizar $1.2 \cdot 10^{18}$ rutas aproximadamente, lo cual es inviable, pues el espacio de búsqueda crece factorialmente.

Otro aspecto imperante es conocer la distancia entre cada una de las ciudades. Comúnmente, estas distancias se representan en forma matricial y dicha matriz recibe el nombre de matriz de distancias. En el problema del viajante clásico, la distancia entre dos ciudades no depende de la ciudad de partida, por lo que tendremos una matriz de distancias simétrica. Sin embargo, algunas variantes eliminan esta condición, incrementan el coste de ciertos viajes o desconectan ciertas ciudades, aumentando la complejidad del problema original.

1.5.2. Modelización e implementación del algoritmo genético

En primer lugar se definen los parámetros de entrada del algoritmo, que serán los siguientes:

- n: Número de ciudades que se desean visitar.
- n_p: Tamaño de la población, medida en individuos, en cada generación.
- n_g: Número de generaciones.
- δ: Probabilidad de mutación.

Tras fijar los parámetros anteriores, se debe modelizar el problema de tal manera que sea posible aplicar el algoritmo genético.

En este caso, los cromosomas se corresponden con las rutas y las ciudades con los genes. A cada una de las n ciudades se le asigna un numero natural del 1 al n, de modo que las diferentes rutas se representan como vectores fila de tamaño n cuyas entradas son los n primeros números naturales. Cada uno de los vectores se interpreta como posible solución al problema.

Puesto que se trata de un problema académico, para un primer ejemplo se supone que las n ciudades se corresponden con los vértices del polígono regular de n lados inscrito en la circunferencia unidad. Por lo tanto, las coordenadas de cada ciudad se obtienen de la siguiente forma:

$$\boldsymbol{c_j} = \left(\cos\left(\frac{2\pi j}{n}\right), \text{sen}\left(\frac{2\pi j}{n}\right)\right),$$

donde $j = 1, 2, ..., n$ y $\boldsymbol{c_j} \in \mathbb{R}^2$ se corresponde con cada una de las ciudades. En la Figura 12, se puede ver la representación gráfica de las ciudades y en líneas discontinuas los caminos entre ellas para $n = 5$.

La matriz de distancias M es en este caso una matriz cuadrada simétrica de tamaño n. La distancia entre cada ciudad se calcula mediante la distancia euclídea d_2 y se almacena en $M = (m_{ij})$ de forma que $m_{ij} = d_2(\boldsymbol{c_i}, \boldsymbol{c_j})$. Es decir, la posición (i, j) de la matriz almacena la distancia entre las ciudades $\boldsymbol{c_i}$ y $\boldsymbol{c_j}$ y la diagonal de M estará formada por ceros.

La *población inicial*, que sirve de punto de partida para la fase evolutiva del algoritmo, se elige de forma aleatoria. Es decir, esta población está formada por n_p rutas, correspondiéndose cada una de ellas con una permutación (generada aleatoriamente) de exactamente n ciudades.

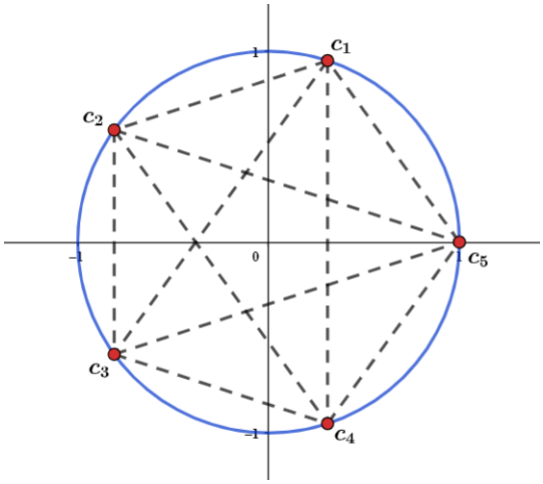

Figura 12: Representación de $n = 5$ ciudades y sus posibles conexiones (línea discontinua).

Por último, falta definir una función objetivo. En este caso la función objetivo $f : \mathbb{N}^n \longrightarrow \mathbb{R}^+$, donde $\mathbb{R}^+$ representa a los números reales positivos, será la suma de las distancias entre cada una de las n ciudades de una ruta $\boldsymbol{r} = (r_1, r_2, ..., r_n) \in \mathbb{N}^n$ dada, es decir:

$$f(\boldsymbol{r}) = \sum_{i=1}^{n-1} m_{r_i, r_{i+1}}.$$

Elegir esta función objetivo, la distancia euclídea y, sobre todo, la representación de las ciudades como los vértices de un polígono regular, permite conocer de antemano la solución óptima del problema.

Proposición 1.1. *El problema del viajante con n ciudades, la distancia euclídea, la representación y la función objetivo presentadas con anterioridad, poseerá n soluciones óptimas que consistirán en recorrer en orden y en cualquier sentido los vértices del polígono regular empezando en cualquiera de ellos.*

Demostración: Fijado un valor n, se construye el polígono regular de n lados inscrito en la circunferencia unidad. Sea l la distancia euclídea entre dos vértices consecutivos de dicho polígono, que será la misma sean cuales sean los vértices tomados por definición de polígono regular.

Sea ahora d la distancia euclídea entre dos vértices no consecutivos de un polígono regular, es decir la longitud de una de sus diagonales. Nuevamente por propiedades de los polígonos regulares, la longitud de cualquiera de sus diagonales es mayor que la longitud de sus lados. Por ende, cualquier suma de lados y diagonales del polígono regular con exactamente $n - 1$ términos en la que intervenga al menos una diagonal, será mayor que el perímetro menos un lado de dicho polígono. Es decir, $(n-1)\cdot l < m_1 \cdot l + m_2 \cdot d$ para $m_1, m_2 \in \mathbb{N}$ cumpliéndose $0 \leq m_1 \leq n-2$, $1 \leq m_2 \leq n-1$ y $m_1 + m_2 = n - 1$.

Por lo tanto, la ruta que menor distancia recorre consiste en seguir en orden los vértices del polígono regular de n lados y dicha distancia mínima se corresponderá con el perímetro menos un lado del polígono, es decir $(n-1)\cdot l$. Además, como el problema del viajante no especifica ningún vértice de inicio, habrá $2n$ formas de llevar a cabo el recorrido. Puesto que tampoco se incluye el sentido en el que se recorren las rutas, únicamente se consideran la mitad de las rutas. De esta forma se concluye que

el problema del viajante con estas características posee n soluciones óptimas distintas de $n!/2$ rutas posibles.

□

En la Figura 13, se representa con trazo continuo rojo el camino que sigue una posible ruta óptima suponiendo que $n = 5$.

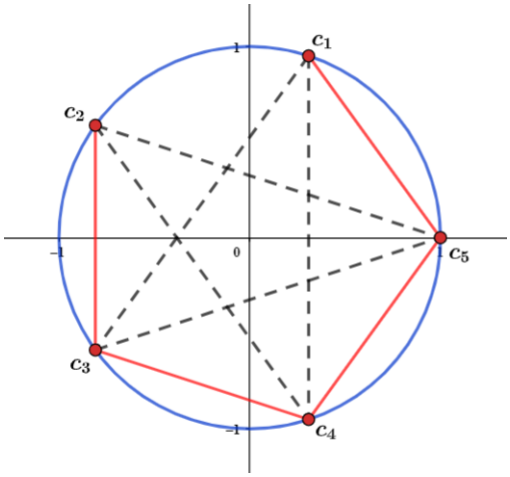

Figura 13: Una ruta óptima para $n = 5$.

Para finalizar la modelización del problema y comenzar con el proceso evolutivo del algoritmo genético, se han de elegir los operadores fundamentales. A continuación, se enumeran los métodos utilizados inicialmente y el motivo de su elección.

La *selección por truncamiento* se ha elegido por su fácil y rápida implementación. En concreto, este método seleccionará el 40 % de las mejores rutas de la población que posteriormente intervendrán en el cruce.

Como se comentó en las Secciones 1.3.2 y 1.3.3, no todos los métodos de cruce y mutación son válidos para los problemas con soluciones codificadas en forma de permutación. Por ese motivo, se han elegido métodos específicamente diseñados para este tipo de problemas, en concreto el *cruce heurístico* y la *mutación por intercambio de dos posiciones*.

Quizás, la fase más relevante del algoritmo genético que se ha diseñado es el cruce. El cruce heurístico empleado genera un único descendiente a partir de dos rutas parentales. Por lo tanto, en primer lugar se eligen aleatoriamente dos rutas P_1 y P_2 entre aquellas previamente seleccionadas en la fase de selección y se procede de la siguiente manera:

1. Se elige aleatoriamente una ciudad c entre las n posibles y se copia en la primera posición de la ruta descendiente H_1.

2. Sean c_1 y c_2 las ciudades que visitan P_1 y P_2 respectivamente inmediatamente después de c. Si solo uno de los padres visita una ciudad que H_1 ya ha visitado, se copia en la primera posición vacía de H_1 la ciudad que visita el otro padre. Si H_1 ya ha visitado tanto c_1 como c_2 se copia en la primera posición vacía de H_1 una ciudad aleatoria entre las n disponibles que aún no haya visitado.

3. Si H_1 aún no ha visitado c_1 ni c_2, se calcula la distancia (en este caso la distancia euclídea) entre la ciudad c y dichas ciudades. Es decir, $D_1 = d_2(c, c_1)$ y $D_2 = d_2(c, c_2)$. Nótese que dichas distancias se encuentran en la matriz de distancias M en las posiciones (c, c_1) y (c, c_2) respectivamente.

4. Finalmente se comparan las distnacias D_1 y D_2. Si $D_1 < D_2$ entonces se copia en la primera posición vacía de H_1 la ciudad c_1, si $D_2 < D_1$ se copia c_2 y si $D_1 = D_2$ se copia al azar c_1 o c_2.

Debido a la interconexión de todas las ciudades, por simplicidad se considera que las rutas son cíclicas, de manera que si la ciudad c ocupa la última posición de alguna de las rutas parentales, la siguiente ciudad que se visitaría sería la primera ciudad de la ruta en cuestión.

Para generar por completo H_1, se repiten $n-1$ veces los pasos 2 a 4. Después de cada iteración se toma c como la última ciudad que haya visitado H_1. Este proceso se repite tantas veces como soluciones descendientes se deseen originar, eligiendo dos nuevos progenitores cada vez.

Por otro lado, en la *mutación por intercambio de dos posiciones* se eligen, con probabilidad δ, dos posiciones de una ruta descendiente aleatoriamente y se intercambia su contenido, es decir se cambia aleatoriamente el orden en el que se visitan dos ciudades. De esta forma, se puede evitar la aparición de subrutas recurrentes, añadiéndose diversidad a la población.

El método de reemplazamiento escogido inicialmente es el *reemplazamiento elitista*. Este tipo de reemplazamiento se ha elegido con el objetivo de preservar rutas parentales de calidad y añadir diversidad a la población. El reemplazamiento elitista utilizado consiste formar la población de la siguiente generación con el 40 % de las mejores rutas parentales y con el mejor 60 % de las rutas descendientes.

Una vez alcanzado el número de generaciones n_g prefijado, el algoritmo devuelve una matriz *MR* de tamaño $n_g \times n$ en la que se almacena la mejor ruta de cada generación, un vector fila *costegen* de tamaño n_g donde se almacena la distancia total viajada en la mejor ruta de cada generación, el número real *tiempo* que almacena el tiempo de computación transcurrido hasta alcanzar el número de generaciones máximo y el valor *mincost*, que es la distancia total recorrida por una de las soluciones óptimas del problema. Puesto que las soluciones óptimas se conocen de antemano, este valor se puede calcular fácilmente y es de utilidad para verificar el funcionamiento del algoritmo genético y comprobar la calidad de las soluciones proporcionadas.

1.5.3. Análisis de resultados

Para concluir esta Sección, se analizan y comentan los resultados obtenidos al resolver el *Problema del Viajante* mediante el algoritmo genético propuesto. En primer lugar, se ejecuta el algoritmo para un número de ciudades bajo y se comenta brevemente el resultado. Posteriormente, se ejecuta para un número de ciudades más alto y se analizan los posibles inconvenientes derivados de los operadores fundamentales elegidos.

En la Figura 14, se muestra la gráfica que proporciona el algoritmo para los parámetros de entrada $n = 10$, $n_p = 30$, $n_g = 5$ y $\delta = 0.1$. En dicha gráfica se representa el coste (medido en términos de la distancia euclídea) de la mejor ruta de cada generación y con trazo discontinuo rojo el coste óptimo, es decir la distancia que recorrería cualquiera de las n rutas óptimas.

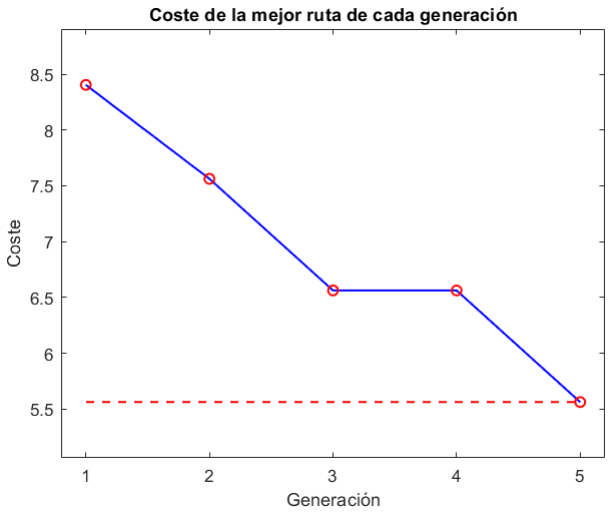

Figura 14: Evolución del coste de la mejor ruta de cada generación para $n = 10$, $n_p = 30$, $n_g = 5$ y $\delta = 0.1$.

Además de la gráfica, el algoritmo también proporciona los siguientes datos:

$$MR = \begin{bmatrix} 9 & 10 & 1 & 2 & 5 & 4 & 3 & 7 & 8 & 6 \\ 8 & 9 & 10 & 1 & 2 & 5 & 4 & 3 & 6 & 7 \\ 7 & 8 & 9 & 10 & 1 & 2 & 3 & 6 & 5 & 4 \\ 7 & 8 & 9 & 10 & 1 & 2 & 3 & 6 & 5 & 4 \\ 5 & 6 & 7 & 8 & 9 & 10 & 1 & 2 & 3 & 4 \end{bmatrix}$$

$$costegen = (8.4039, 7.5623, 6.5623, 6.5623, 5.5623)$$

$$tiempo = 0.0287$$

$$mincost = 5.5623$$

A la vista de la gráfica y los resultados, se obtienen varias conclusiones. En primer lugar, el decrecimiento monótono de la gráfica indica una mejora (o al menos un no empeoramiento) en la calidad de las rutas en las sucesivas generaciones, satisfaciéndose así el principal objetivo de cualquier algoritmo genético. Además, se alcanza una solución óptima en la última generación, que se corresponde con la ruta que parte de la ciudad 5 y recorre en orden y sentido horario el resto de vértices del decágono regular. Si a todo esto se le suma el bajo tiempo de computación empleado, se puede concluir que el algoritmo diseñado promete ser eficiente para esta versión del *Problema del Viajante*. Sin embargo, una única ejecución del algoritmo no es representativa, pues en otros casos puede no obtenerse una solución óptima o aumentar o disminuir el tiempo de computación considerablemente.

A continuación se muestra el contenido de la matriz parental P que almacena las rutas que se reproducen en cada generación. En particular se presenta la matriz P de la generación $n_g = 5$:

$$P = \begin{bmatrix} 5 & 6 & 7 & 8 & 9 & 10 & 1 & 2 & 3 & 4 \\ 7 & 8 & 9 & 10 & 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 1 & 2 & 3 & 6 & 5 & 4 \\ 7 & 8 & 9 & 10 & 1 & 2 & 3 & 6 & 5 & 4 \\ 7 & 8 & 9 & 10 & 1 & 2 & 3 & 6 & 5 & 4 \\ 6 & 5 & 7 & 8 & 9 & 10 & 1 & 2 & 4 & 3 \\ 6 & 5 & 4 & 3 & 7 & 8 & 9 & 10 & 1 & 2 \\ 6 & 5 & 4 & 3 & 7 & 8 & 9 & 10 & 1 & 2 \\ 4 & 3 & 6 & 5 & 7 & 8 & 9 & 10 & 1 & 2 \\ 3 & 4 & 2 & 5 & 6 & 7 & 8 & 9 & 10 & 1 \\ 2 & 5 & 6 & 7 & 8 & 9 & 10 & 1 & 3 & 4 \\ 8 & 7 & 5 & 6 & 4 & 3 & 2 & 1 & 9 & 10 \end{bmatrix}$$

En la matriz P se observa que varios de los progenitores seleccionados son la misma ruta o tienen características genéticas casi idénticas, lo cual podría producir una convergencia prematura del algoritmo diseñado y un desperdicio de tiempo de computación en la reproducción de individuos iguales o muy similares. Esta convergencia prematura podría provocar que el algoritmo fuese incapaz de mejorar las soluciones de una generación a otra o que solo fuese posible a través de la mutación, desperdiciándose esfuerzo computacional. La convergencia temprana podría verse reforzada por la elección de los operadores fundamentales, en especial por el método de selección y de reemplazamiento, pues quizás son demasiado elitistas y no introducen la diversidad suficiente en la población.

A priori, si el algoritmo alcanzase una de las soluciones óptimas el elitismo no sería un problema, pero para comprobarlo, y buscar una solución si fuese necesario, se aplica el algoritmo genético diseñado a un *Problema del Viajante* con parámetros de entrada más desafiantes computacionalmente. Esta vez, el algoritmo genético se ejecutará cinco veces y los datos de salida que proporcionará, y que serán analizados, son el tiempo de computación empleado en cada ejecución *tiempoit* y el coste de la mejor ruta de cada generación para cada ejecución *costegenit*.

Eligiendo los parámetros de entrada $n = 20$, $n_p = 10$, $n_g = 20$ y $\delta = 0.1$ y ejecutando el algoritmo cinco veces, se obtiene la gráfica representada en la Figura 15. El tiempo medio de computación de las cinco ejecuciones ha sido 0.0308 segundos.

Analizando la Figura 15 (izquierda), se observan dos hechos relevantes. El primero de ellos es la no obtención de una ruta óptima en ninguna de las ejecuciones, lo cual es un problema relativo, pues el objetivo de los algoritmos genéticos no es proporcionar siempre una solución óptima, si no una solución lo suficientemente cercana al óptimo en un tiempo razonablemente bajo. Otro problema que se observa, es la convergencia prematura en todas las ejecuciones, lo cual sugiere que el alto elitismo es un problema. La forma más sencilla de intentar solucionar esto sería modificar alguno de los parámetros de entrada, como el tamaño de la población o el número de generaciones, lo cual incrementaría el coste computacional. Además, normalmente no se conoce qué valores son los adecuados para dichos parámetros ni si mejorarán o empeorarán el rendimiento del algoritmo. Otra opción factible es modificar la tasa de mutación.

En la Figura 15 (derecha), se presenta la gráfica obtenida cambiando únicamente la tasa de mutación,

que aumenta hasta un 90 %. En este caso se sigue sin obtener una ruta óptima, pero quizá se ha mitigado en cierto modo la convergencia prematura. El tiempo de computación medio ha sido de 0.0319 segundos, que es muy similar al anterior.

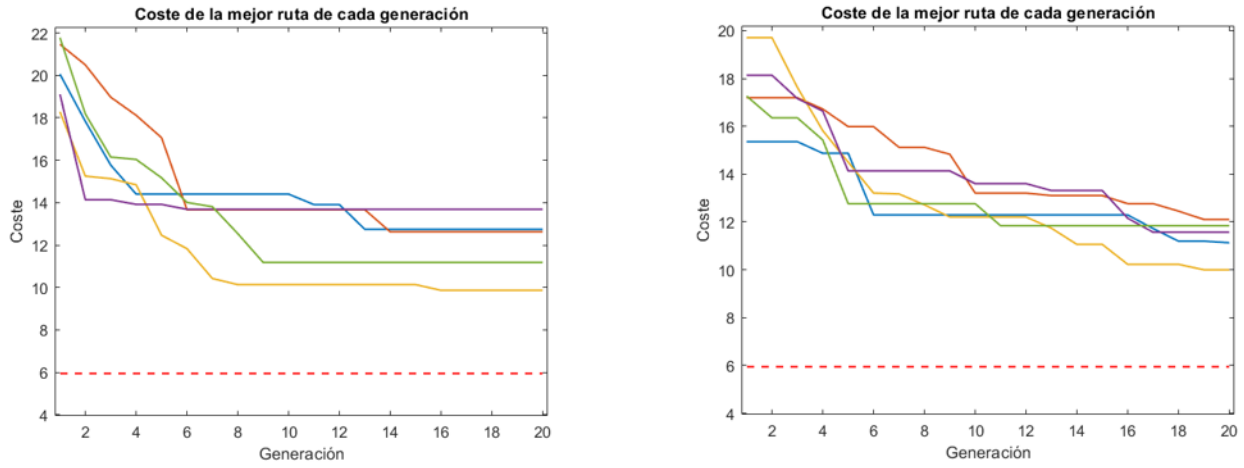

Figura 15: A la izquierda, se representa la evolución del coste de la mejor ruta de cada generación con una tasa de mutación del 10 %. A la derecha, se representa la evolución del coste de la mejor ruta de cada generación con una tasa de mutación del 90 %

No obstante, cuando el número de ciudades, el tamaño de la población o el número de generaciones es elevado, incrementar la tasa de mutación puede acarrear un aumento considerable del tiempo de computación. Por ejemplo, tomando los parámetros $n = 100$, $n_p = 300$, $n_g = 150$ y $\delta = 0.1$ y ejecutando el algoritmo cinco veces, la media de tiempos de computación obtenida es cercana a 77.9215 segundos, mientras que si se realiza el mismo proceso con $\delta = 0.9$ la media asciende a 93.9159 segundos.

Por ello, para evitar la convergencia prematura producida por el alto elitismo del algoritmo diseñado, se modificará el algoritmo cambiando alguno de los operadores fundamentales problemáticos, que en este caso parecen ser la selección por truncamiento y el reemplazamiento elitista.

A continuación, en la Figura 16 (izquierda) se muestra la gráfica obtenida al ejecutar cinco veces el algoritmo genético diseñado originalmente y en la Figura 16 (derecha) la gráfica obtenida al ejecutar cinco veces una nueva versión del algoritmo, ambos con parámetros de entrada $n = 100$, $n_p = 300$, $n_g = 150$ y $\delta = 0.2$. La versión modificada del algoritmo utiliza el método de reemplazamiento generacional, manteniendo el resto de operadores invariables con respecto al algoritmo original, es decir se mantiene la selección por truncamiento, el cruce heurístico y la mutación por intercambio de dos posiciones. Además, con el objetivo de reducir el tiempo medio de computación y la eficiencia del algoritmo, se han tensorizado las fases de cruce y mutación en la implementación. Es decir, para cada una de las n_g generaciones se han calculado simultáneamente las n_p rutas descendientes (en vez de crearlas de una en una mediante un proceso iterativo) y de igual forma se mutan, con cierta probabilidad δ, todas ellas a la vez.

Con el algoritmo original, el tiempo medio de computación ha sido de 82.8996 segundos y el coste de la mejor ruta obtenida globalmente ha sido 10.4660, mientras que utilizando la nueva versión el tiempo medio de computación ha sido de 7.0493 segundos y el coste de la mejor ruta global de 6.2193, que coincide con el óptimo. Como es evidente, el tiempo de computación medio del algoritmo modificado

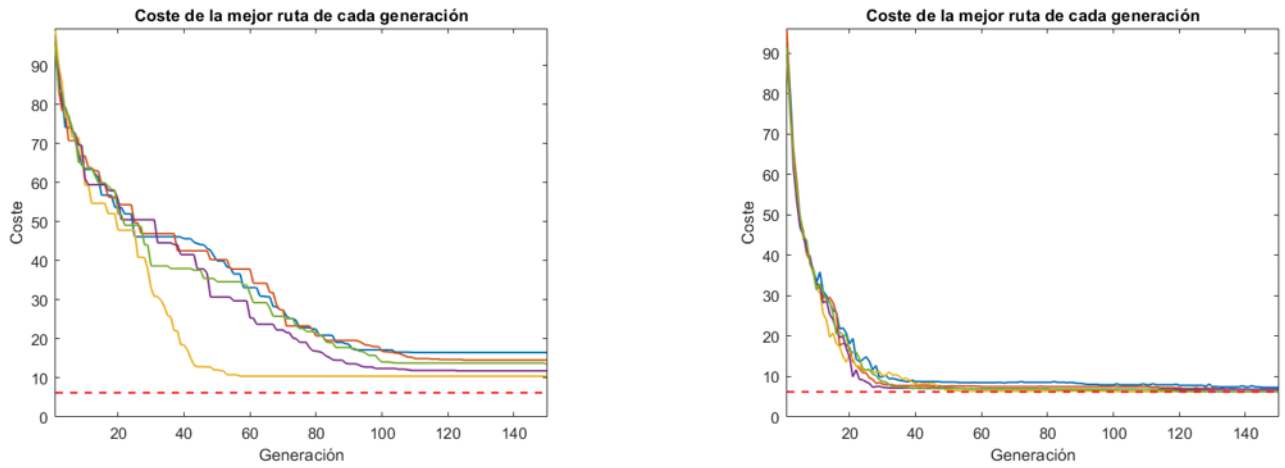

Figura 16: A la izquierda, se representa la evolución del coste de la mejor ruta de cada generación para cada una de las ejecuciones del algoritmo genético original. A la derecha, se representa la evolución del coste de la mejor ruta de cada generación para cada una de las ejecuciones del algoritmo genético modificado con reemplazamiento generacional.

es muy inferior al empleado por el algoritmo original, en concreto menos de una décima parte. Esto se debe, en gran parte, a la nueva implementación del cruce y la mutación y, en menor medida, al cambio en el método de reemplazamiento. Por lo tanto, la comparación de ambas versiones se centrará principalmente en la calidad de las soluciones proporcionadas, ya que claramente el algoritmo modificado es muy superior en términos de tiempo de computación.

Analizando conjuntamente ambas gráficas, se observa como todas las funciones generadas por el algoritmo original son monótonas decrecientes, mientras que en el caso del algoritmo modificado no lo son. Esto se debe al cambio en el método de reemplazamiento, ya que el reemplazamiento elitista asegura que la calidad de la mejor ruta de cada generación será igual o mejor que la calidad de la mejor ruta de la generación anterior, mientras que el reemplazamiento generacional forma la población de cada generación únicamente con rutas descendientes, pudiéndose perder individuos de mejor calidad al no prevalecer ninguna ruta de generaciones previas.

En primer lugar, como se observa en la Figura 16, el algoritmo modificado parece haber solventado solventado por completo el problema de convergencia prematura producido por el alto elitismo. Esta convergencia prematura, es claramente visible en el caso del algoritmo original en todas las ejecuciones alrededor de la generación 100.

El único inconveniente que podría surgir con el algoritmo modificado, sería abandonar soluciones óptimas una vez alcanzadas, como se mencionó en la Sección 1.4. Al no asegurarse la supervivencia de la mejor ruta entre generaciones, no se puede asegurar la convergencia hacia el óptimo global. Se pueden observar fenómenos similares a los presentados en la Sección 1.4, es decir se alcanza una solución óptima en un tiempo finito y también, aunque se haya alcanzado, se puede perder en generaciones posteriores. Debido a lo anterior, en la implementación práctica de este tipo de algoritmos es común almacenar de alguna manera la mejor solución encontrada a lo largo del proceso, de forma que si se abandonase no se perdería por completo.

Finalmente, el algoritmo genético modificado, que utiliza el reemplazamiento generacional, parece ser

mejor alternativa al algoritmo genético originalmente diseñado, que emplea el reemplazamiento elitista, para este *Problema del Viajante* en concreto. Además, cabe destacar la importancia de una implementación eficiente del algoritmo, por ejemplo utilizando tensorización o paralelizando el problema con multiprocesadores, ya que puede suponer un ahorro significativo en tiempo de computación.

Capítulo 2

Algoritmos de colonia de hormigas

En este Capítulo, se introduce brevemente el concepto de inteligencia de enjambre para luego desarrollar en profundidad una de sus ramas, los algoritmos de optimización de colonia de hormigas. Además, al igual que en el primer Capítulo, se utiliza el *Problema del Viajante* con el fin de ejemplificar los conceptos estudiados y comparar el resultado con el obtenido aplicando el algoritmo genético.

2.1. Inteligencia de enjambre

La *inteligencia de enjambre* (ver [1]) es un campo dentro de la informática que se dedica al estudio y diseño de métodos eficientes para resolver diversos problemas. El término "*enjambre*" implica la agrupación de múltiples individuos que presentan características de estocasticidad, aleatoriedad y desorden. Por otro lado, el concepto de "*inteligencia*" sugiere que el método utilizado para abordar los problemas es, de alguna manera, exitoso. Aunque la organización de un enjambre puede variar significativamente, la característica principal e indispensable en cualquier método basado en inteligencia de enjambre es la interacción entre sus individuos. Dependiendo del enfoque adoptado, el enjambre pueden estar formado por individuos animados, computacionales o matemáticos; pueden representar insectos, aves o seres humanos; y pueden ser tanto reales como ficticios.

Entre los distintos enfoques, desataca aquel que se inspira en aspectos y comportamientos biológicos de colonias de insectos, bandadas de aves y diferentes agrupaciones de seres vivos. Alguno de los aspectos más influyentes en este enfoque son la auto-organización, la estigmergia, el control descentralizado y la heterarquía densa (ver [2]). La estigmergia se entiende como una forma de comunicación a través de modificaciones en el entorno o como interacciones sociales indirectas entre individuos de una misma especie. Por otro lado, la heterarquía es un concepto biológico que hace referencia a la organización de insectos sociales. En particular, la heterarquía es una forma de organización en la que cualquier individuo puede influir en los demás y no existe ninguno con más poder que el resto, en contra posición con la organización jerárquica. El adjetivo denso implica una alta relación entre los individuos.

Frecuentemente, los problemas de optimización y búsqueda compleja se abordan mediante algoritmos basados en la inteligencia de enjambre. Es aquí donde desatacan la *optimización de colonia de hormigas* y la *optimización de enjambre de partículas*. En concreto, la optimización de colonia de hormigas se suele aplicar en problemas de optimización combinatoria y la optimización de enjambre de partículas

en la búsqueda de máximos y mínimos de funciones.

La optimización de colonia de hormigas es, por lo tanto, un tipo de algoritmo heurístico típicamente utilizado para resolver problemas de optimización combinatoria. La idea principal que motivó el desarrollo de estos algoritmos y les dio nombre, es la capacidad que poseen los insectos sociales y, en especial las hormigas, para resolver problemas complejos de forma natural. Numerosos estudios biológicos se centraron en comprender cómo las colonias de hormigas resolvían problemas extremadamente complicados para un único individuo basándose en la experiencia colectiva. Uno de los experimentos biológicos más importantes y precursor de los algoritmos de optimización de colonia de hormigas es el *experimento del doble puente* (ver [3]). Su propósito era estudiar la comunicación indirecta de las hormigas mediante las feromonas (sustancias químicas liberadas por ciertos seres vivos que influyen en el comportamiento de los individuos de la misma especie) y su impacto en la obtención del camino más corto hasta una fuente de alimentos.

Dicho experimento consistía en unir mediante dos puentes distintos un hormiguero de hormigas de la especie *iridomyrmex humilis* con una fuente de comida, siendo uno de los puentes más corto que el otro, y estudiar el comportamiento observado. Tras numerosas repeticiones del experimento, se notó que el puente de menor longitud casi siempre era preferido frente al más largo, pese a que ningún individuo poseía una visión global de los caminos. La razón de este suceso está estrechamente relacionada con las feromonas desprendidas por las hormigas, pues aquellas que eligen el puente más corto llegarán antes a la fuente de alimentos y regresarán por ese mismo puente al detectar el rastro de feromonas que han desprendido. De esta manera, en el puente de menor longitud se acumulan más rápido las feromonas, convirtiéndose con el paso del tiempo en el camino preferido al poseer una mayor concentración de feromonas.

En la Figura 17 se representa de manera esquemática este experimento, donde las hormigas se corresponden con los puntos y las feromonas con los trazos discontinuos.

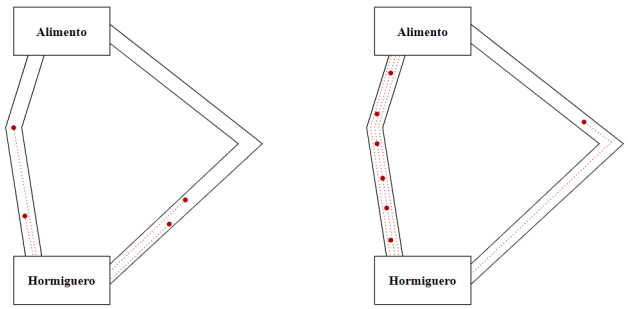

Figura 17: Experimento del doble puente.

2.2. Optimización de colonia de hormigas

Los algoritmos de optimización de colonia de hormigas se diseñan imitando, en mayor o menor medida, los comportamientos de las hormigas para así obtener soluciones eficientes a los problemas en los que se aplican. Uno de los aspectos fundamentales de estos algoritmos es dotar a las hormigas artificiales de un método constructivo que las permita generar soluciones mediante una secuencia de decisiones

probabilísticas, de manera que cada decisión crea soluciones parciales que son ensambladas hasta obtener una solución completa. La secuencia de decisiones necesaria para generar completamente una solución puede ser interpretada como un *camino* en un grafo de decisiones, al cual algunos autores denominan *grafo de construcción*.

Por lo tanto, el objetivo principal de los algoritmos de colonia de hormigas es permitir a las hormigas artificiales construir caminos a través del grafo de construcción que se espera deriven en soluciones eficientes. Para ello, se lleva a cabo un proceso iterativo en el cual las buenas soluciones encontradas en una iteración guiarán a las hormigas de futuras iteraciones. Esto es posible gracias a las feromonas artificiales, pues a las hormigas que han encontrado una buena solución se les permite marcar el camino recorrido en el grafo. Este rastro de feromonas artificiales permitirá a las hormigas buscar, en el futuro, nuevos caminos cercanos a aquellos que conducen a buenas soluciones.

2.2.1. Modelización del algoritmo

Como ya se ha comentado, los algoritmos de colonia de hormigas típicamente se aplican a problemas de optimización combinatoria, por lo que la modelización y aspectos relevantes que se presentan en esta sección se centran en este tipo de problemas. Cabe destacar que la modelización puede variar en función del problema tratado, por lo que los conceptos que a continuación se desarrollan tratan de ser lo más genéricos posible.

El primer paso, necesario en la modelización de cualquier algoritmo de este tipo, es la representación del problema. Para ello es preciso generar un espacio de búsqueda, que estará formado por todas las posibles soluciones, y elegir una función objetivo que evalúe la calidad de cada una de las soluciones candidatas. Al interpretar cada solución como una secuencia formada por distintos componentes, el espacio de búsqueda se puede representar mediante un grafo de construcción $G = (V, E)$ donde el conjunto de nodos V representa a las componentes de las soluciones y el conjunto de aristas E las conexiones posibles entre dichos nodos. Además se debe definir la forma de actualización de los rastros de feromonas y una estrategia heurística, que proporcionará información presumiblemente útil en la construcción de soluciones.

Los rastros de feromonas funcionan como memoria adaptativa de las hormigas, guiando a futuros individuos hacia buenas soluciones, mientras que la información heurística ayuda a realizar buenas conexiones entre las componentes de una solución, basándose en información proporcionada por un buen análisis de las características del problema o por hormigas de iteraciones previas.

El segundo aspecto crucial es decidir cómo se van a comportar las hormigas artificiales. Los movimientos de las hormigas se pueden interpretar como un proceso estocástico donde se construyen soluciones a partir del grafo G. Además, el movimiento está normalmente influido por los rastros de feromonas y por información heurística. La implementación de los rastros de feromonas y la información heurística en el algoritmo está altamente condicionada por la representación del problema y su naturaleza. Generalmente, para problemas de permutación esta información se almacena en matrices, y la probabilidad de que cierta hormiga conecte una componente $v \in V$ con otra cualquiera $w \in V$, viene dada por

$$p_{vw} = \frac{\tau_{vw}^{\alpha} \cdot \eta_{vw}^{\beta}}{\sum_{t \in V} \tau_{vt}^{\alpha} \cdot \eta_{vt}^{\beta}} \qquad \forall w \in V,$$

donde τ y η representan las matrices de feromonas e información heurística respectivamente y los parámetros α y β la influencia de dicha información en la decisión. A priori, $\alpha, \beta \in \mathbb{R}$ son parámetros ajustables por el usuario pero, en la práctica, se suele utilizar $\alpha, \beta \geq 0$.

También, se supone que las hormigas artificiales tienen cierta memoria para almacenar el camino recorrido y que cada una de ellas construye al menos una solución completa. Por lo tanto, las hormigas se mueven siguiendo reglas de decisión probabilísticas en función de los rastros de feromonas, la información heurística y las restricciones del problema. Habitualmente, las restricciones del problema se tienen en cuenta o bien evitando los movimientos que conduzcan a soluciones no factibles o bien penalizando dichas soluciones.

Finalmente, otro aspecto de suma importancia para este tipo de algoritmos es la evaporación de feromonas. Este fenómeno produce una reducción gradual y global de las feromonas, lo cual se traduce en una pérdida de memoria en las hormigas y en un aumento en la exploración del espacio de búsqueda. El principal objetivo de la evaporación es evitar la convergencia prematura del algoritmo, es decir evitar caer en máximos (o mínimos) locales.

Una vez hechas las consideraciones anteriores, puede dar comienzo el proceso iterativo que permitirá construir las soluciones. Para ello, se presenta un esquema general que sirve como punto de partida para diseñar la gran mayoría de algoritmos de colonia de hormigas (ver [3]):

1. **Inicialización:** Se inicializan los rastros de feromonas y los valores heurísticos que poseerá cada una de las posibles conexiones de E, así como la elección del número de hormigas que formarán la colonia artificial.
2. **Construcción:** Se permite a cada una de las hormigas de la colonia construir al menos una solución teniendo en cuenta los rastros de feromonas, los valores heurísticos y las restricciones del problema estudiado.
3. **Actualización de feromonas:** Se evalúan todas las soluciones generadas por las hormigas y se incrementan en una cierta cantidad o porcentaje las feromonas de las conexiones que derivan en soluciones de una calidad prefijada.
4. **Evaporación:** Se lleva a cabo la evaporación de feromonas, reduciendo un cierto porcentaje la cantidad de feromonas existentes en cada una de las conexiones de E.

Los pasos 2 a 4 se repiten hasta que se cumple un criterio de detención elegido por el usuario. Los criterios más comunes son alcanzar un número máximo de iteraciones, obtener una solución de una calidad prefijada u obtener la misma solución durante cierto número de iteraciones (ver [3]). Otra posibilidad sería utilizar conjuntamente varios de estos criterios.

2.2.2. Características relevantes

La eficiencia de un algoritmo de colonia de hormigas, radica principalmente en una correcta modelización de las fases del esquema genérico presentado en la subsección anterior. Si bien un algoritmo eficiente se podría obtener utilizando unos parámetros de entrada adecuados, esta combinación normalmente se desconoce y está estrechamente relacionada con el problema estudiado. Por lo tanto,

para optimizar al máximo la eficiencia del algoritmo se implementan distintos métodos y variantes que afectan a las fases del proceso iterativo, principalmente mediante las feromonas y la información heurística.

Los rastros de feromonas y la información heurística, en este tipo de algoritmos, ayudan a introducir diversidad en las soluciones y a aprovechar el conocimiento previo o adquirido sobre el problema, respectivamente. Por un lado, aumentar la influencia de las feromonas y la información heurística en la elección de los movimientos, puede agilizar y dirigir la búsqueda hacia soluciones óptimas en un menor tiempo de computación. No obstante, una influencia excesiva de estos factores podría producir una convergencia prematura del algoritmo.

En contraposición, la exploración del espacio de búsqueda del problema se ve favorecida al disminuir la influencia de las feromonas y la información heurística. Por ejemplo, al reducir la influencia de las feromonas, los movimientos de las hormigas de futuras iteraciones no estarán tan condicionados por este factor, siendo más probable obtener soluciones con una mayor diversidad en sus componentes y pudiéndose evitar estancamientos en máximos (o mínimos) locales. Otra forma de controlar fácilmente la exploración del espacio de búsqueda, es aumentar o disminuir la tasa de evaporación de feromonas. Aumentar la evaporación favorece la exploración, mientras que reducirla produce el efecto contrario.

La influencia de los rastros de feromonas y la información heurística en el algoritmo se suele controlar mediante parámetros, normalmente denotados por α y β. Estos parámetros pueden ser elegidos por el usuario al comienzo del proceso iterativo o ser ajustados dinámicamente por el algoritmo en función de resultados previos. La implementación de estos parámetros en la construcción de soluciones, se detalla en la Subsección 2.3.1. Como se menciona en [2], el ajuste de estos parámetros debe realizarse conjuntamente, para así tener un control más preciso sobre la exploración del espacio de búsqueda o la explotación del conocimiento previo del problema.

Además de los mecanismos ya mencionados, existen otros métodos que ayudan a diseñar algoritmos eficientes controlando la influencia de los rastros de feromonas. A continuación, se presenta alguno de estos métodos, que por lo general se centran en la actualización de los rastros de feromonas (ver [3]):

- **Actualización basada en la calidad:** En esta variante, las k mejores soluciones refuerzan con feromonas los *caminos* que conducen hacia ellas, donde $k < m$ y m representa el número total de soluciones construidas en cada iteración. Normalmente, la cantidad de feromonas que aportan estas k soluciones es proporcional a su calidad.

- **Actualización elitista:** En este método de actualización, únicamente la mejor solución generada en cada iteración puede actualizar el rastro de feromonas del *camino* que ha derivado en ella. La cantidad en la que aumenta el rastro de feromonas se puede elegir de diversas formas, se puede elegir una cantidad aleatoria que varíe de una iteración a otra, se puede prefijar al inicio del algoritmo o puede ser proporcional a la calidad de la solución.

- **Actualización de mejor-peor solución:** En este caso se actualizan los rastros de feromonas que conducen a buenas soluciones, para lo cual se puede emplear alguno de los dos métodos anteriores, pero también se disminuye la cantidad de feromonas de los *caminos* que conducen a soluciones de poca calidad. Esta disminución de feromonas es independiente del proceso de

evaporación, y se puede aplicar únicamente a la peor solución de cada iteración o a las k' peores rutas, donde $k' < m$.

Sin embargo, el principal inconveniente de este método es la posible pérdida de soluciones eficientes, pues aunque una solución sea de mala calidad puede contener componentes o *subcaminos* que podrían conducir a soluciones óptimas y, al reducir las feromonas asociadas a dichos componentes, sería menos probable explorar esas soluciones potencialmente óptimas.

- **Actualización de máximo y mínimo:** Este método es quizás uno de los más completos, pues además de afectar a la actualización de feromonas también afecta a la inicialización de las mismas. En primer lugar, la cantidad de feromonas que posee cualquier *camino* en cualquier iteración varía entre un valor mínimo y uno máximo. Todos los *caminos* se inicializan con el valor máximo de feromonas y posteriormente únicamente la mejor solución de cada iteración actualiza los rastros de feromonas. Esta actualización es inversamente proporcional a la calidad de la solución, de manera que cuanto mayor sea su calidad menor será el aumento de feromonas.

Pese a que los métodos de actualización basados en la calidad y los elitistas son los más sencillos de implementar y pueden reducir considerablemente el tiempo de computación total, se deben utilizar con precaución ya que podrían causar una convergencia prematura del algoritmo. No obstante, existe una variante que ayuda a evitar este problema y también se puede implementar en otros métodos. Dicha variante, consiste en reinicializar todos los rastros de feromonas si se detecta cierto grado de estancamiento en las soluciones, lo cual da lugar a una mayor exploración del espacio de búsqueda.

Finalmente, alguna otra variante del algoritmo de colonia de hormigas canónico no basada en la actualización de feromonas, se presenta en [3]. Entre dichas variantes destacan la implementación de listas de movimientos candidatos o preferidos, la utilización de feromonas repelentes o la mejora local de soluciones.

2.2.3. Convergencia

Al igual que con los algoritmos genéticos, se desconoce una teoría global de convergencia para los algoritmos de colonia de hormigas, por lo que el estudio de este fenómeno se centra principalmente en las propiedades y características que deben cumplirse para garantizar la convergencia.

Pese a lo anterior, existe una variante de la optimización de colonia de hormigas, llamada *Sistema de Hormigas Basado en Grafos* cuya convergencia ha sido probada. Una de las diferencias entre esta variante y los principales algoritmos de optimización de colonia de hormigas, radica en la actualización de feromonas (ver [2]). No obstante, puesto que esta variante dista bastante de los algoritmos de colonia de hormigas generales, se prueba la convergencia de dichos algoritmos bajo ciertas premisas para problemas de optimización formulados de la siguiente forma (ver [12]).

Sea (S, f, Ω) un problema de minimización donde S representa al conjunto de todas las soluciones candidatas, f es la función objetivo que evalúa cada una de las soluciones y Ω es el conjunto de las restricciones del problema que define las soluciones candidatas factibles. Este problema de minimización, se puede modelizar utilizando el grafo de construcción $G = (V, E, T)$, donde T denota un vector que recopila los rastros de feromonas τ asociados a cada una de las posibles conexiones. Recuérdese

que τ_{ij} contiene las feromonas asociadas al desplazamiento de la componente i a la componente j.

Sea (ver [12]) un algoritmo de colonia de hormigas diseñado para resolver el problema (S, f, Ω), donde cada una de las hormigas conecta las componentes $v, w \in V$ con probabilidad

$$p_{vw} = \frac{\tau_{vw}^{\alpha}}{\sum_{t \in V} \tau_{vt}^{\alpha}} \qquad \forall w \in V,$$

y la actualización de feromonas se lleva a cabo evaporando, en primer lugar, las feromonas asociadas a cada una de las conexiones de E, en función de cierta tasa de evaporación $\rho \in (0, 1)$, y aumentando únicamente las feromonas de las conexiones que derivan en la mejor solución que se haya encontrado hasta el momento. Nótese que en este algoritmo no se está haciendo uso de ninguna información heurística.

Un algoritmo de colonia de hormigas y un problema de minimización como los indicados, cumplen (utilizando que, debido a la evaporación, el nivel máximo de feromonas está acotado asintóticamente (ver [12])), que

Teorema 2.1. *Sea $P^*(t)$ la probabilidad de que el algoritmo de colonia de hormigas encuentre al menos una vez una solución óptima en las primeras t iteraciones. Entonces, para $\epsilon > 0$ suficientemente pequeño y t suficientemente grande, se cumple que $P^*(t) \geq 1 - \epsilon$ y asintóticamente $\lim_{t \to \infty} P^*(t) = 1$.*

La demostración de este enunciado, se encuentra en *Theorem 1* de [12].

El *Teorema 2.1.* afirma que se obtiene una solución óptima con una probabilidad arbitrariamente cercana a 1 si se emplea el tiempo suficiente (que generalmente es desconocido). Además, una vez hallada una solución óptima, en [12] también se prueba que se puede establecer una estimación del límite inferior de la probabilidad con la que una hormiga construirá una solución óptima con posterioridad.

Por último, señalar que en [12] se indica que el *Teorema 2.1.* generalmente se puede extender a todos aquellos algoritmos de colonia de hormigas donde la probabilidad $P(s)$ de construir una solución siempre sea mayor que una constante pequeña $\epsilon > 0$. En particular, este teorema se cumple para aquellos algoritmos de colonia de hormigas en los que se establezca una cantidad mínima de feromonas τ_{min} para cualquier conexión, se limite la cantidad de feromonas que pueden desprender cada una de las hormigas al finalizar la iteración y se permita que los rastros de feromonas se evaporen con el tiempo.

2.3. Aplicación: El Problema del Viajante

Para concluir este Capítulo, se diseña un algoritmo de colonia de hormigas que se utilizará para resolver el *Problema del Viajante* y analizar el funcionamiento e importancia de los conceptos introducidos previamente. El algoritmo diseñado se inspira en los presentados en [2] y [3].

Tanto el enunciado del problema, como su representación y la función objetivo utilizadas, son las mismas que se utilizaron para el ejemplo del algoritmo genético. Es decir, se representan las n ciudades como los vértices de un polígono regular de n lados inscrito en la circunferencia unidad y se toma la función objetivo $f : \mathbb{N}^n \longrightarrow \mathbb{R}^+$ dada por

$$f(\boldsymbol{r}) = \sum_{i=1}^{n-1} m_{r_i, r_{i+1}},$$

para cierta ruta $\boldsymbol{r} = (r_1, r_2, ..., r_n) \in \mathbb{N}^n$. Además, esta modelización permite calcular fácilmente la matriz de distancias M y, como se enuncia en la *Proposición 1.1*, se conocen previamente las soluciones óptimas del problema.

Para más detalles sobre esta primera fase de modelización del problema, consultar las Subsecciones 1.5.1 y 1.5.2.

2.3.1. Diseño del algoritmo

El primer paso en el diseño del algoritmo es la elección de los parámetros de entrada, que en este caso son:

- n: Número de ciudades que se desean visitar.
- n_h: Número de hormigas que forman la colonia artificial en cada iteración.
- *iter*: Número de iteraciones máximas.
- *evap*: Proporción de feromonas evaporadas globalmente en cada iteración.
- *aumen*: Cantidad de feromonas añadidas a las rutas de buena calidad construidas.
- α: Parámetro que determina la influencia de las feromonas en la elección de cada conexión.
- β: Parámetro que determina la influencia de la información heurística en la elección de cada conexión.

Una vez elegidos los anteriores parámetros, se inicializan los rastros de feromonas y la información heurística. Como el *Problema del Viajante* se ha modelizado como un problema de permutación, estos datos se pueden almacenar fácilmente en dos matrices.

En la matriz de feromonas $\tau = (\tau_{ij})$, se almacena la cantidad de feromonas que posee cada una de las posibles conexiones entre las n ciudades del problema. Es decir, la posición (i, j) de τ contiene la cantidad τ_{ij} de feromonas asociadas al camino entre las ciudades i y j. En este caso, la matriz de feromonas τ se ha inicializado tomando aleatoriamente valores del intervalo $[0, 1)$.

Debido a las particularidades de este problema y a que la distancia entre cada una de las n ciudades es fácilmente calculable y viene dada por la matriz de distancias $M = (m_{ij})$, una estrategia heurística eficiente consiste en favorecer los movimientos de las hormigas hacia las ciudades más cercanas a aquella en la que se encuentran en cada momento (ver [3]). De esta forma, la matriz de información heurística $\eta = (\eta_{ij})$ se define como $\eta_{ij} = 1/m_{ij}$ para todo $i, j \in \{1, 2, ..., n\}$, $i \neq j$.

Nótese que las matrices τ y η son cuadradas, simétricas y sus diagonales son nulas.

Tras la inicialización de los parámetros, da comienzo la fase de construcción de soluciones, donde se ha decidido que cada una de las n_h hormigas de cada iteración construya únicamente una ruta. Además,

se define la construcción de las soluciones como un proceso iterativo que se lleva a cabo de la siguiente forma. En primer lugar, se define un conjunto C, que contiene cada una de las n ciudades que se ha de visitar, y se elige una de ellas aleatoriamente, que será la ciudad de partida. Una vez elegida esa primera ciudad, se elimina del conjunto C y se decide en qué orden se visitan las $n-1$ ciudades restantes necesarias para completar una ruta, es decir una solución completa. Estas decisiones se toman de una en una en función de la siguiente distribución de probabilidad discreta

$$p_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{t \in C} \tau_{it}^{\alpha} \cdot \eta_{it}^{\beta}} \qquad \forall j \in C,$$

donde i representa la ciudad en la que se encuentra la hormiga en cada momento y j cada una de las ciudades que aun no se han visitado. Tras visitar una ciudad esta se elimina del conjunto C, evitándose la construcción de soluciones incoherentes. Este proceso se repite tantas veces como hormigas haya en cada iteración, construyéndose así n_h soluciones.

Como se comentó en la Subsección 2.2.2, variando el valor de los parámetros α y β, se controla la influencia tanto de la intensidad del rastro de feromonas entre las ciudades i y j como de la distancia, entre ellas, en el valor de probabilidad p_{ij} de visitar la ciudad j inmediatamente después de la ciudad i. Sin embargo, se desconoce la magnitud de los parámetros α y β o la relación que debe existir entre ambos para obtener buenas soluciones, pues es un aspecto muy dependiente del problema y la modelización del mismo. Por lo tanto, la combinación que maximiza la eficiencia del algoritmo se suele obtener experimentalmente.

Por otro lado, el método de actualización de feromonas implementado es la *actualización elitista*. En concreto, en este algoritmo la actualización de τ se lleva a cabo aumentando una cantidad fija el número de feromonas, siendo dicha cantidad proporcionada por el usuario y denotada por *aumen*. Por ejemplo, suponiendo que la mejor solución de cierta iteración (en términos de la función objetivo f) es $r^* = (r_1^*, r_2^*, ..., r_n^*)$, se aumentan las feromonas de los *caminos* que han conducido a dicha solución, es decir

$$\tau_{r_i^* r_{i+1}^*} = \tau_{r_i^* r_{i+1}^*} + aumen,$$

para $i = 1, 2, ..., n-1$.

El último aspecto de interés en el diseño de este algoritmo, es la evaporación de feromonas. Esta evaporación se produce de forma global, afectando a todos los caminos posibles entre las n ciudades, y viene dada por una tasa fija *evap* proporcionada por el usuario. En particular, la disminución de feromonas se refleja en τ de la siguiente forma:

$$\tau_{ij} = \tau_{ij} \cdot (1 - evap),$$

donde $i, j = 1, 2, ..., n$.

Una vez transcurrido el número máximo de iteraciones *iter* proporcionado por el usuario, el algoritmo devuelve una matriz *MR* de tamaño $iter \times n$ donde se almacena la mejor solución construida en cada iteración, un vector fila *costeiter* de *iter* posiciones donde se almacena el coste de la mejor ruta de cada iteración, la variable *tiempo* que se corresponde con el tiempo de computación empleando por el

algoritmo, la variable *mincost* que contiene el coste de una de las rutas óptimas y una gráfica donde se representa la evolución del coste de la mejor ruta construida en cada iteración.

2.3.2. Análisis de resultados

Para concluir este Capítulo, se analizan los resultados obtenidos al aplicar el algoritmo diseñado al *Problema del Viajante*. También se comparan los resultados con los obtenidos aplicando el algoritmo genético presentado en el primer Capítulo, con el fin de analizar el comportamiento cualitativo de estas dos formas distintas de resolver un mismo problema. En concreto se, comparará con el algoritmo genético modificado, ya que era la versión que mejores resultados proporcionaba (ver Subseción 1.5.3 para más detalles sobre esta versión).

Pese a que el algoritmo proporciona numerosos datos de salida, el análisis de los resultados se centra únicamente en la evolución de los costes frente a las iteraciones y el tiempo de computación empleado. Además, el algoritmo se ejecutará más de una vez con cada combinación de parámetros de entrada elegida, de manera que los resultados serán más representativos.

Tras ejecutar el algoritmo un total de cinco veces para los parámetros de entrada $n = 100$, $n_h = 50$, $iter = 30$, $evap = 0.2$, $aumen = 2$, $\alpha = 2$ y $\beta = 3$, se obtiene la gráfica representada en la Figura 18 (izquierda), con un coste mínimo global de 7.0976 (relativamente cercano a 6.2193, que es el coste óptimo) y un tiempo de computación medio de 19.3033 segundos. Comparando estos resultados con los obtenidos utilizando el algoritmo genético modificado, para el mismo número de ciudades, se observa que el coste de las rutas obtenidas con el algoritmo de colonia de hormigas es ligeramente superior, pues en el caso del algoritmo genético se alcanzaban rutas óptimas. En lo referente al tiempo medio de computación, este es más elevado en el caso del algoritmo de colonia de hormigas. No obstante, la diferencia entre ambos no es muy notable, y conviene señalar que, en este algoritmo de hormigas, la trayectoria de cada hormiga se ha calculado de forma independiente.

Por otro lado, cabe destacar que los valores de α y β elegidos son relativamente bajos, por lo que se estaría fomentando la exploración del espacio de búsqueda (ver Subsección 2.2.2). Aumentando el valor de β hasta 6 y manteniendo el resto de parámetros de entrada invariables, se puede observar en la gráfica de la Figura 18 (derecha) que, en este caso, los costes de las mejores rutas construidas en las iteraciones de cada ejecución están mucho más próximos al óptimo, incluso en alguna de las ejecuciones se alcanza.

El único inconveniente que surge al aumentar el valor de β, puede ser la convergencia prematura del algoritmo en alguna de las ejecuciones. Para solucionar este problema, se podría cambiar el método de actualización de feromonas, ya que una modificación de los parámetros *evap* o α podrían producir una pérdida de calidad en las rutas construidas y una ampliación del espacio de búsqueda del problema. No obstante, la posibilidad de esta convergencia temprana es asumible, pues en este caso se produce hacia rutas de muy buena calidad y el tiempo medio de computación es muy reducido.

La gran mejoría en la calidad de las rutas construidas al aumentar únicamente el valor de β, está estrechamente relacionada con la representación del problema y la estrategia heurística escogidas. Puesto que el valor de la información heurística entre dos ciudades i y j se ha definido como el inverso de la distancia entre dichas ciudades, ese valor h_{ij} será máximo cuando i y j se encuentren posicionadas

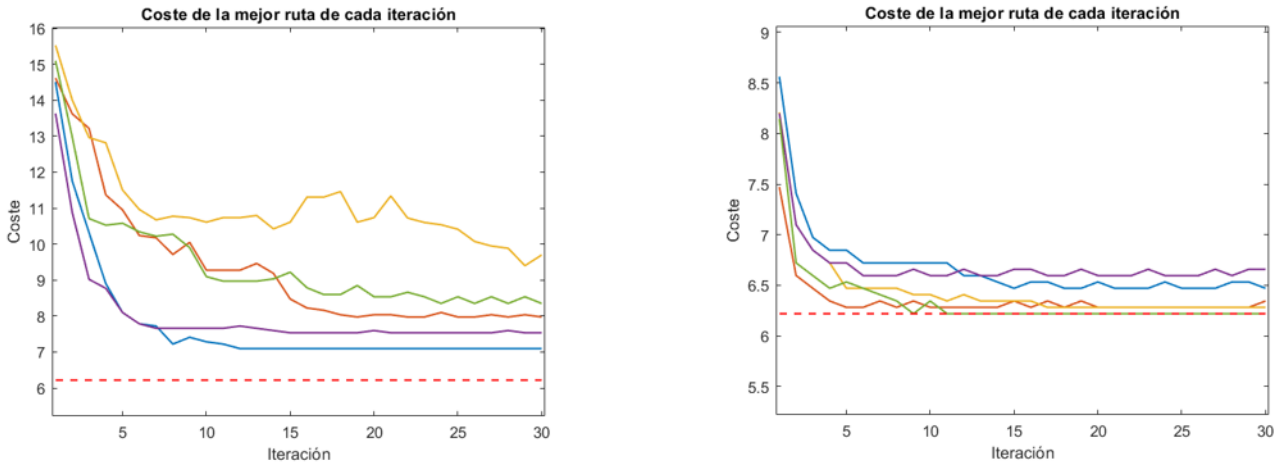

Figura 18: A la izquierda, evolución del coste de la mejor de cada iteración para cada ejecución tomando $\beta = 3$. A la izquierda, evolución del coste de la mejor de cada iteración para cada ejecución tomando $\beta = 6$.

en vértices adyacentes del polígono regular de n lados, ya que la longitud de los lados del polígono regular es menor que la de cualquiera de sus diagonales. De esta forma, dejando de lado la influencia de los rastros de feromonas, cuando las ciudades i y j sean adyacentes, la probabilidad p_{ij} de visitar j justo después de i es mayor que la probabilidad de visitar cualquier otra ciudad no adyacente, pues al ser fijo el denominador de p_{ij}, cuanto mayor sea su numerador mayor será el valor final de p_{ij}. Por lo tanto, cuando el valor de β aumenta lo suficiente, la probabilidad de elegir el desplazamiento de una ciudad a una de sus ciudades adyacentes es muy superior, ya que el numerador de p_{ij} será ligeramente mayor al denominador, siendo por tanto p_{ij} un valor muy próximo a 1 y siendo la probabilidad de desplazarse hacia ciudades no adyacentes casi nula. De esta forma, cuando el valor de β sea lo suficientemente grande, la probabilidad de que las hormigas construyan rutas que recorran el perímetro del polígono regular de n lados es muy elevada. En la Figura 19 se representa la gran mejoría en la calidad de las soluciones al aumentar lo suficiente β, donde se observa que en casi todas las ejecuciones el algoritmo converge hacia el óptimo en las primeras iteraciones.

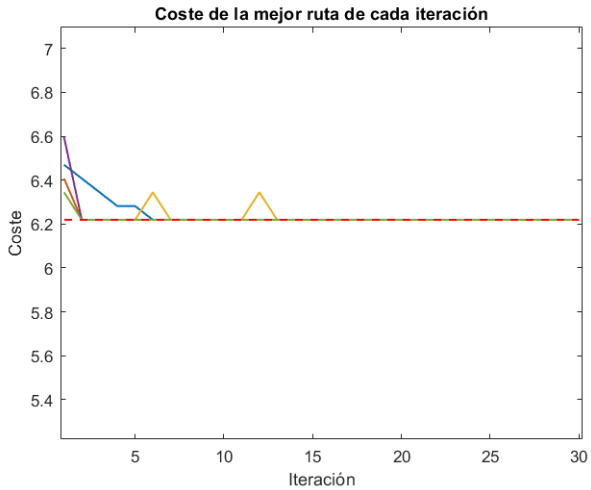

Figura 19: Resultado de ejecutar cinco veces el algoritmo para los parámetros $n = 100$, $n_h = 50$, $iter = 30$, $evap = 0.2$, $aumen = 2$, $\alpha = 2$ y $\beta = 9$.

Además, se ha comprobado tras numerosas ejecuciones que en este caso concreto del *Problema del Viajante*, la mejoría en la calidad de las soluciones se produce si β es mucho más grande que α, o lo que es lo mismo, si únicamente se aumenta el valor de β.

Finalmente, conviene analizar si esto ocurre siempre o ha sido fruto de la disposición particular de las ciudades.

2.3.3. Cambio en la disposición de las ciudades: Malla cuadrada

Con el fin de verificar si el algoritmo de colonia de hormigas es superior al genético solamente en esta configuración del *Problema del Viajante* o lo es en general, independientemente de la ubicación de las ciudades, se plantea otra disposición y se comparan nuevamente los resultados de ambos algoritmos.

Por comodidad en la implementación, el número de ciudades n se toma de forma que sea un cuadrado perfecto, pues dichas ciudades se representarán en una malla cuadrada. Por lo tanto, las coordenadas de cada una de las n ciudades se pueden interpretar como la posición que ocupan en una matriz cuadrada de $\sqrt{n} \times \sqrt{n}$ posiciones. Además, esta forma de distribuir las ciudades, permite conocer las rutas óptimas, que serán aquellas que no contengan ningún desplazamiento en diagonal ni realicen movimientos hacia ciudades no contiguas, y el coste óptimo, que sera $n - 1$. En la Figura 20, se representa a la izquierda la malla cuadrada para $n = 49$ ciudades y a la derecha una posible ruta óptima para dicha representación y número de ciudades.

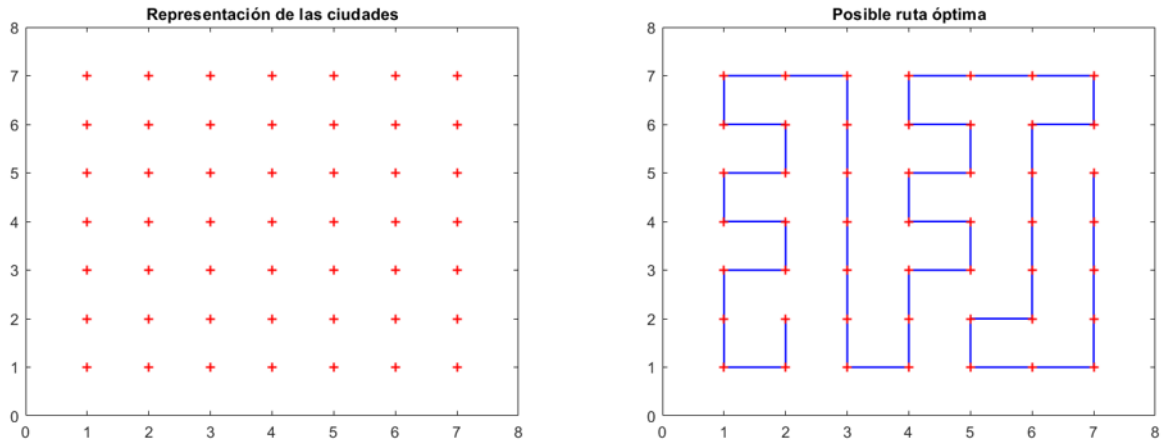

Figura 20: A la izquierda, distribución de $n = 49$ ciudades en una malla cuadrada. A la derecha, representación de una posible solución óptima para dicha distribución utilizando el algoritmo de colonia de hormigas.

El resto de la modelización del problema se mantiene igual que antes, es decir se utiliza la distancia euclídea y la misma función objetivo f.

A continuación, se analizan los resultados de cada algoritmo para cada tipo de representación y después se comparan entre ellos. Para una mejor visualización y comparación de las gráficas, los costes proporcionados tanto por el algoritmo genético modificado como por el algoritmo de colonia de hormigas, se normalizan mediante una transformación lineal que asigna el valor 1 a la peor ruta obtenida en cada ejecución y el valor 0 a las rutas óptimas.

En la Figura 21, se representan las gráficas obtenidas al ejecutar el algoritmo genético modificado

cinco veces, a la izquierda con las ciudades dispuestas en un polígono regular y a la derecha en una malla cuadrada, con los parámetros de entrada $n = 100$, $n_p = 300$, $n_g = 50$ y $\delta = 0.2$. Recuérdese que el algoritmo genético empleaba la selección por truncamiento, el cruce heurístico, la mutación por intercambio de dos posiciones y el reemplazamiento generacional. Utilizando la representación de las ciudades como vértices del polígono regular de n lados, el coste de la mejor ruta global ha sido de 6.2821 (siendo el óptimo 6.2193) y el tiempo medio de computación de 2.3630 segundos, mientras que para la representación de las ciudades en forma de malla cuadrada, dicho coste ha sido de 102.4787 (siendo el óptimo 99) y el tiempo medio de computación de 2.4195 segundos.

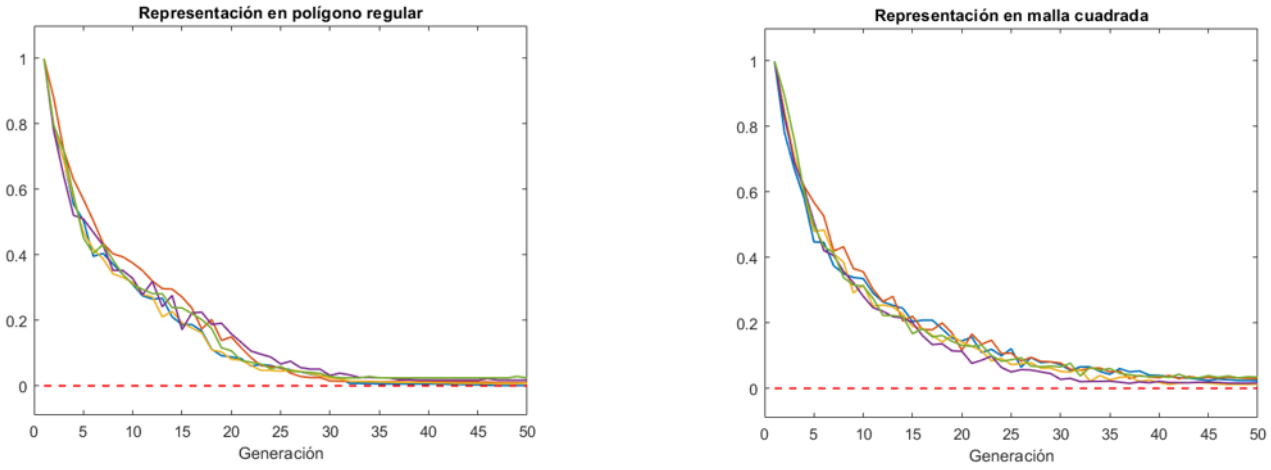

Figura 21: A la izquierda, evolución del coste normalizado de la mejor ruta de cada generación, para cada una de las ejecuciones, utilizando la disposición de polígono regular de $n = 100$ lados. A la derecha, evolución del coste normalizado de la mejor ruta de cada generación, para cada una de las ejecuciones, utilizando la disposición de malla cuadrada 10×10.

En la Figura 22, se representan las gráficas obtenidas al ejecutar cinco veces el algoritmo de colonia de hormigas diseñado en la Sección 2.3.1. Los parámetros de entrada utilizados son $n = 100$, $n_h = 50$, $iter = 30$, $evap = 0.2$, $aumen = 2$, $\alpha = 2$ y $\beta = 6$. Para la representación de las ciudades en forma de polígono regular (izquierda), el coste de la mejor ruta global ha sido el coste óptimo, es decir 6.2193 y el tiempo medio de computación ha sido de 20.7297 segundos. En cambio, utilizando la representación de las ciudades en forma de malla cuadrada (derecha), el coste de la mejor ruta global ha sido de 103.3137 (siendo el óptimo 99) y el tiempo medio 16.2267 segundos.

Analizando las gráficas obtenidas con el algoritmo genético (Figura 21), se observa que para ambas representaciones el algoritmo proporciona soluciones muy buenas alrededor de la generación 35. Además, en ambos casos el tiempo de computación es muy reducido. Por otra parte, el algoritmo de colonia de hormigas proporciona soluciones óptimas o muy cercanas al óptimo en un tiempo razonable cuando la disposición de las ciudades coincide con los vértices del polígono regular de n lados, pero la calidad de las mismas disminuye al utilizar una disposición en forma de malla cuadrada.

Por tanto, los experimentos realizados parecen indicar que, en el caso del algoritmo de colonia de hormigas (Figura 22), la calidad de las soluciones obtenidas varía en función de la disposición de las ciudades que se emplea. Utilizando la disposición de las ciudades en forma de polígono regular, se obtienen soluciones de muy buena calidad, o incluso óptimas, en un número reducido de iteraciones (alrededor de la quinta iteración), mientras que en el caso de la disposición en forma de malla cuadrada,

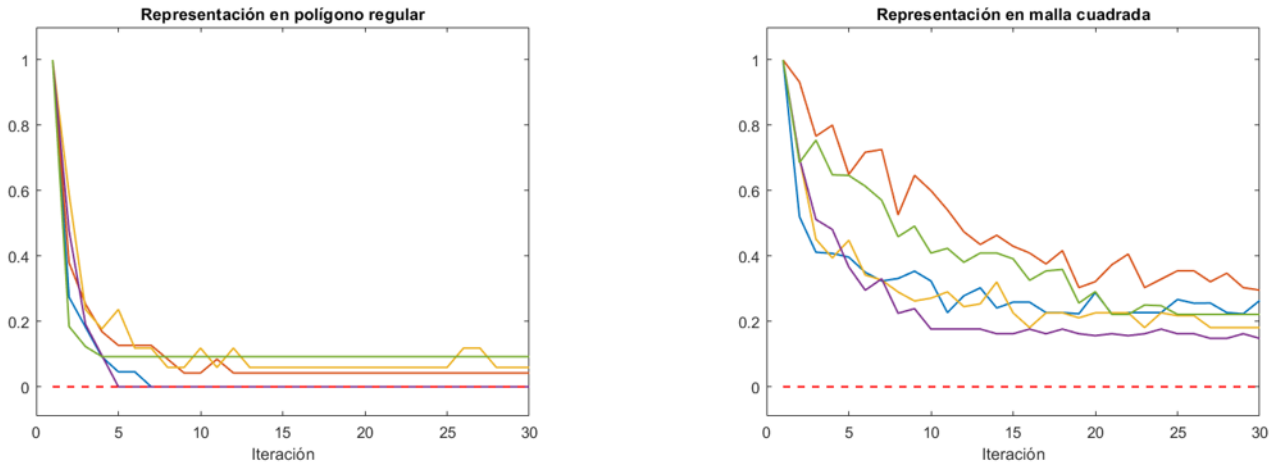

Figura 22: A la izquierda, evolución del coste normalizado de la mejor ruta de cada iteración, para cada una de las ejecuciones, utilizando la disposición de polígono regular de $n = 100$ lados. A la derecha, evolución del coste normalizado de la mejor ruta de cada iteración, para cada una de las ejecuciones, utilizando la disposición de malla cuadrada 10×10.

la calidad de las mismas disminuye considerablemente.

Sin embargo, lo anterior no quiere decir que el algoritmo de colonia de hormigas no pueda llegar a proporcionar buenas soluciones utilizando la disposición en forma de malla cuadrada, sino que el valor adecuado de los parámetros α y β parece ser dependiente de la ubicación concreta de las ciudades. Por el contrario, el comportamiento cualitativo del algoritmo genético en ambas disposiciones (Figura 21) es muy similar, lo cual parece indicar que el rendimiento del algoritmo diseñado es independiente de la ubicación concreta de las ciudades.

Capítulo 3

Aplicaciones de los algoritmos heurísticos en la resolución de juegos

En este Capítulo, se diseñan un algoritmo genético y un algoritmo de colonia de hormigas para implementarlos respectivamente en dos juegos diferentes: el *Sudoku* y el *Nurikabe*. Estos algoritmos se desarrollan fundamentalmente utilizando los conceptos teóricos presentados en los capítulos anteriores y la literatura disponible al respecto. Cabe destacar que el objetivo de este Capítulo es la resolución, no la generación de dichos problemas.

3.1. Algoritmo genético: Aplicación al *Sudoku*

En primer lugar, se desarrolla brevemente la historia y reglas del juego combinatorio *Sudoku* para, posteriormente, diseñar un algoritmo genético y analizar los resultados obtenidos tras su implementación.

3.1.1. Descripción del juego

Sudoku (ver [13] y [16]) es un rompecabezas combinatorio basado en la lógica. Pese a su origen incierto, se cree que fue desarrollado a finales de la década de 1970 en Estados Unidos, aunque fue popularizado por Japón en la década de 1980 y en el resto del mundo a principios del siglo XXI.

El rompecabezas clásico *Sudoku*, o por su traducción del japonés *Número Único*, consiste en rellenar una cuadrícula de tamaño 9×9 dividida en 9 cajas o bloques de tamaño 3×3 cumpliéndose que cada fila, columna y bloque de la cuadrícula deben contener una única vez todos los números naturales del 1 al 9. Inicialmente, algunas celdas vienen ya rellenas con números, que han sido calculados. Recientemente, se ha demostrado que son necesarios al menos 17 números dados inicialmente para que la solución del rompecabezas sea única (ver [16]).

En particular, un rompecabezas *Sudoku* es un caso concreto de un cuadrado latino, pues además de no repetir ningún número entre 1 y 9 en cada una de las filas y columnas, tampoco se repiten en ninguno de sus 9 bloques.

En la Figura 23 se representa un rompecabezas *Sudoku* y su solución.

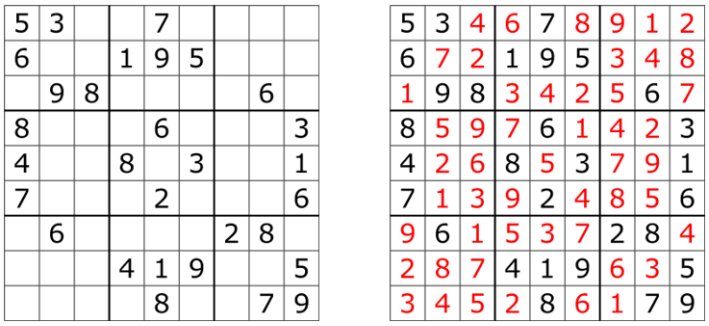

Figura 23: A la izquierda, representación inicial de un rompecabezas *Sudoku* clásico. A la derecha, su solución. Fuente: *Imagen de Wikipedia*. (c.c.). https://en.wikipedia.org/wiki/Sudoku.

3.1.2. Diseño del algoritmo

En el diseño del algoritmo genético para este problema, se han utilizado principalmente los conceptos introducidos en el Capítulo 1 y un algoritmo basado en el presentado en [13]. Sin embargo, se han implementado ciertas variaciones para explorar y comparar distintas alternativas.

En primer lugar, los parámetros de entrada del algoritmo son los siguientes:

- S: *Sudoku* que se desea resolver.
- n_p: Tamaño de la población para cada una de las generaciones.
- n_g: Número de generaciones.
- δ: Tasa de mutación.

El *Sudoku* S a resolver, se corresponde con una matriz cuadrada de 9×9 posiciones. Por lo tanto, se consideran soluciones candidatas de S todas aquellas matrices de tamaño 9×9 cuyas entradas sean únicamente números naturales del 1 al 9 y que contengan los mismos números iniciales, y exactamente en las mismas posiciones, que S. Nótese que en esta primera definición de solución candidata, se permiten configuraciones que incumplan reglas del *Sudoku*.

En lo referente a la representación cromosómica del *Sudoku* (ver Figura 24), se ha decidido representar tanto S como las soluciones candidatas en forma de vectores fila. Cada vector fila estará formado por la concatenación de las 9 filas del *Sudoku* de arriba hacia abajo.

Por lo tanto, las soluciones candidatas serán vectores fila de 81 posiciones (*genes*) que contendrán únicamente números naturales del 1 al 9. El vector que representa al *Sudoku* a resolver S, contendrá ceros en todas las posiciones no dadas inicialmente, es decir en las posiciones a completar.

En cuanto a la población inicial, esta se inicializa de forma semialeatoria, pues se impone a cada uno de sus n_p individuos la regla de no repetición de ningún numero natural entre el 1 y el 9 en una misma fila del *Sudoku*. Para ello, la inicialización de cada vector candidato o solución se realiza identificando en cada fila del *Sudoku* los números aún por usar y colocándolos aleatoriamente en las posiciones vacías,

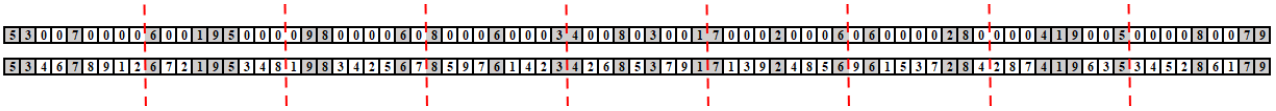

Figura 24: Arriba, representación en forma de vector fila del *Sudoku* de la Figura 23. Abajo, su solución representada como vector fila. Las líneas discontinuas marcan el inicio y fin de cada fila del *Sudoku*.

que se corresponden con las posiciones nulas de la misma fila de S, permaneciendo así inalterados los números dados inicialmente en S. Por ejemplo, en la primera fila del *Sudoku* (Figura 23) faltan por usar los números $\{1, 2, 4, 6, 8, 9\}$. Estos números se colocarán aleatoriamente en las posiciones $\{3, 4, 6, 7, 8, 9\}$ del vector candidato o solución (Figura 24 (Arriba)). Con el resto de filas, se procede de forma análoga.

El principal motivo por el cual se impone el cumplimiento de esta regla, es facilitar la búsqueda de la solución óptima, pues a priori el algoritmo parte de una población inicial con buenas características, lo cual ayuda a simplificar la función objetivo y por ende el proceso evolutivo del algoritmo.

Para finalizar la primera fase de modelización, que precede a la fase evolutiva del algoritmo genético, se debe elegir una función objetivo adecuada. En este caso, dicha función verifica el cumplimiento de las reglas del *Sudoku* y penaliza a aquellas soluciones que las incumplan. A continuación, se define la función objetivo f y sus componentes siguiendo criterios similares a los de [13].

La numeración de los subbloques 3×3 utilizada se muestra en la Figura 25.

1	2	3
4	5	6
7	8	9

Figura 25: Numeración de un subbloque 3×3 de un *Sudoku*.

Sean B_k y C_k dos conjuntos que denotan respectivamente los subbloques y columnas *k-ésimas* del *Sudoku*. Sea el conjunto $N = \{n \in \mathbb{N} : 1 \leq n \leq 9\}$ y $\boldsymbol{x} = (x_1, x_2, ..., x_{81})$ una solución candidata. Haciendo un pequeño abuso de notación, sean:

$$B_k(\boldsymbol{x}) = \{x_j \in \boldsymbol{x} : j \in B_k\},$$

$$C_k(\boldsymbol{x}) = \{x_j \in \boldsymbol{x} : j \in C_k\}.$$

Se define

$$g_1^{(k)}(\boldsymbol{x}) = |N \setminus C_k(\boldsymbol{x})| \quad k \in \{1, 2, ...9\},$$

$$g_2^{(k)}(\boldsymbol{x}) = |N \setminus B_k(\boldsymbol{x})| \quad k \in \{1, 2, ...9\}.$$

Nótese que $g_1^{(k)}(\boldsymbol{x})$ calcula cuántos números no se utilizan en la *k-ésima* columna de $\boldsymbol{x}$ (es decir, verifica el cumplimiento de la regla sobre la columna *k-ésima*) y $g_2^{(k)}(\boldsymbol{x})$ calcula cuántos números no se utilizan en el *k-ésimo* subbloque 3×3 de $\boldsymbol{x}$ (es decir, comprueba el cumplimiento de la regla sobre el subbloque *k-ésimo*), donde $|\cdot|$ denota el cardinal de un conjunto.

Finalmente, como primer candidato a función objetivo f se toma

$$f(\boldsymbol{x}) = \sum_{k=1}^{9} g_1^{(k)}(\boldsymbol{x}) + \sum_{k=1}^{9} g_2^{(k)}(\boldsymbol{x}).$$

Esta función objetivo permite identificar fácilmente la solución óptima del problema, que será aquella con coste nulo. Como se puede observar, en la función objetivo f no verifica el cumplimiento de la regla de no repetición de números por filas, ya que siempre que se utilicen la inicialización de la población descrita en esta Sección y los métodos de cruce y mutación que a continuación se describen, dicha regla siempre se cumplirá.

A continuación se describen con detalle los operadores fundamentales que han sido elegidos para llevar a cabo la fase evolutiva del algoritmo genético.

El método de selección elegido es la *selección por truncamiento*. En particular se ha elegido la selección por sorteo de 2 individuos, dentro de los mejores de la población. El principal motivo de esta elección ha sido la facilidad en la implementación.

Para generar la descendencia en cada una de las n_g generaciones, el método de cruce escogido es una variante del *cruce de dos puntos*. Esta variante consiste en elegir aleatoriamente dos padres de entre las soluciones candidatas preseleccionadas mediante el operador de selección y dos puntos de cruce, que serán los mismo para ambos padres. A diferencia del cruce de dos puntos genérico, los puntos de cruce se eligen en el conjunto de posiciones que marcan el inicio o fin de las filas del *Sudoku*, es decir en el conjunto de posiciones de la forma $\{9i + 1 : i = 0, 1, ..., 8\}$. Posteriormente, se intercambian los *genes* situados entre dichos puntos entre ambos padres, generándose dos nuevos individuos en cada apareamiento, por lo que únicamente serán necesarios $n_p/2$ apareamientos para crear n_p soluciones candidatas. Por lo tanto, por comodidad se supondrá que el parámetro de entrada n_p es par. Además, puesto que los puntos de cruce son idénticos en ambos padres, la posición de los números dados inicialmente en S permanece constante.

En este caso, el método de mutación utilizado es una variante de la *mutación por intercambio de tres posiciones*. Para implementar esta variante, se elige aleatoriamente una fila del individuo a mutar, de la cual se toman tres posiciones aleatorias que no contengan números dados inicialmente en S, es decir tres posiciones que se correspondan con posiciones nulas del *Sudoku* S. Finalmente, el intercambio de las posiciones elegidas se lleva a cabo cíclicamente, es decir suponiendo que las posiciones a intercambiar son $(1, 2, 3)$, el orden de las mismas tras el intercambio será $(3, 1, 2)$.

Como ya se adelantó al definir la función objetivo f, la elección de estas variantes del cruce de dos puntos y de la mutación por intercambio de posiciones, junto con la inicialización de la población empleada, garantizan automáticamente el cumplimiento de la regla del *Sudoku* sobre las filas. Esto se debe por una parte a que el método de cruce intercambia una o más filas completas entre las soluciones parentales, y, por otra parte a que la mutación, si se produjese, afecta a las posiciones de una única fila. De esta forma, con estos métodos se consigue mantener la no repetición de números en cada una de las filas del *Sudoku*.

Por último el reemplazamiento empleado es el *reemplazamiento elitista*. En particular, la población de cada generación creada usando este reemplazamiento elitista estará formada en un 30 % por soluciones

parentales y en un 70 % por las mejores soluciones descendientes creadas en la fase de cruce. La implementación de este método tiene como objetivo la preservación de soluciones candidatas de buena calidad entre las sucesivas generaciones.

Al igual que en los algoritmos descritos en anteriores capítulos, el algoritmo genético diseñado proporciona una serie de parámetros de salida que serán utilizados para analizar su desempeño. Dichos parámetros son una matriz *MS* de dimensiones $n_g \times 81$ donde se almacena la mejor solución candidata hallada en cada generación, un vector fila *costegen* de n_g posiciones donde se guarda el coste de la mejor solución de cada generación, la variable *tiempo* que representa el tiempo de computación empleado durante la ejecución y una gráfica que representa la evolución del coste de la mejor solución obtenida en cada generación.

3.1.3. Primer análisis de resultados

Para analizar tanto el desempeño del algoritmo genético diseñado como los resultados obtenidos, se ha aplicado a diversos *Sudokus* etiquetados como fáciles.

Al igual que en anteriores capítulos, el análisis de los resultados se centra principalmente en la evolución del coste frente a las generaciones.

Tras aplicar el algoritmo genético a ciertos *Sudokus* de dificultad fácil un total de cinco veces, con los parámetros de entrada $n_p = 600$, $n_g = 200$ y $\delta = 0.8$, se han obtenido la gráficas similares a la representada en la Figura 26, donde el coste mínimo global es de 6 y el tiempo medio de compilación ha sido 106.2432 segundos.

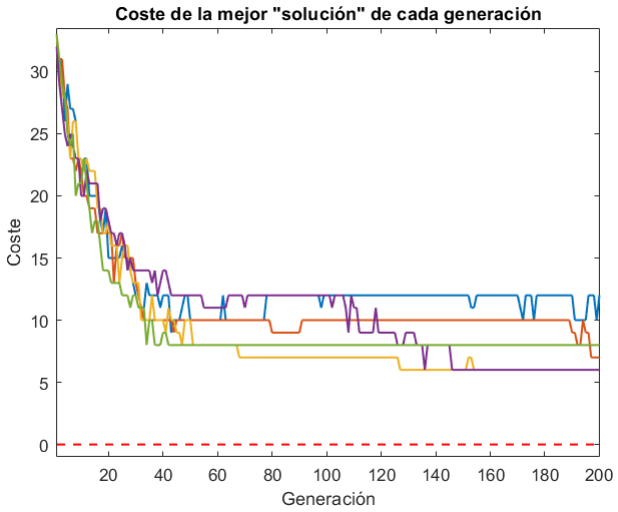

Figura 26: Evolución del coste de la mejor solución generada en cada iteración para el *Sudoku* de dificultad fácil y parámetros de entrada $S =$ Sudoku_Facil.mat, $n_p = 600$, $n_g = 200$ y $\delta = 0.8$.

Se observa una convergencia prematura en varias ejecuciones, en especial a partir de la generación 60. Además, analizando los individuos que forman la población de la última generación de cada ejecución, se comprueba que la gran mayoría de ellos son superindividuos o copias de estos, es decir soluciones que se corresponden con mínimos locales hacia los cuales converge el algoritmo. Este fenómeno, podría verse potenciado por una elección demasiado elitista de los operadores fundamentales, ya que pese a

utilizarse una alta tasa de mutación (que en principio debería ayudar en la exploración del espacio de búsqueda), la convergencia temprana parece ser recurrente.

3.1.4. Segundo análisis de resultados

Para tratar de solventar el problema anterior y poder tener la posibilidad de alcanzar soluciones óptimas, se modifica el algoritmo genético diseñado. En particular, se modifican la función objetivo y el método de reemplazamiento y se introducen pequeñas modificaciones tanto en el método de cruce como en el de mutación.

Siguiendo la línea de estudio de [13] y buscando penalizar en mayor medida aquellas soluciones donde se repitan números en columnas o subbloques 3×3, se implementan las siguientes modificaciones en la función objetivo f. Utilizando la notación presentada en la Sección 3.1.2, se definen para $k \in \{1, 2, ...9\}$ las funciones $g_3^{(k)}$, $g_4^{(k)}$, $g_5^{(k)}$ y $g_6^{(k)}$ como

$$g_3^{(k)}(\boldsymbol{x}) = \left| 45 - \sum_{j \in C_k} x_j \right|, \qquad g_5^{(k)}(\boldsymbol{x}) = \left| 45 - \sum_{j \in B_k} x_j \right|,$$

$$g_4^{(k)}(\boldsymbol{x}) = \left| 9! - \prod_{j \in C_k} x_j \right|, \qquad g_6^{(k)}(\boldsymbol{x}) = \left| 9! - \prod_{j \in B_k} x_j \right|,$$

donde en este caso $|\cdot|$ denota al valor absoluto.

Las funciones $g_3^{(k)}$ y $g_4^{(k)}$ calculan la diferencia entre el valor de la suma y el producto para cada columna de la solución óptima y la suma y el producto de cada fila de la solución $\boldsymbol{x}$. De forma similar, las funciones $g_5^{(k)}$ y $g_6^{(k)}$ realizan los mismos cálculos pero aplicados a cada uno de los subbloques 3×3 de la solución $\boldsymbol{x}$.

Finalmente, la función objetivo f tomada ha sido

$$f(\boldsymbol{x}) = 5000 \left(\sum_{j=1}^{9} \left(g_1^{(k)}(\boldsymbol{x}) + g_2^{(k)}(\boldsymbol{x}) \right) \right) + 10 \left(\sum_{j=1}^{9} \left(g_3^{(k)}(\boldsymbol{x}) + g_5^{(k)}(\boldsymbol{x}) \right) \right) + \sum_{j=1}^{9} \left(\sqrt{g_4^{(k)}(\boldsymbol{x})} + \sqrt{g_6^{(k)}(\boldsymbol{x})} \right),$$

donde los pesos se han ajustado experimentalmente.

En cuanto al método de selección, este se simplifica considerablemente, pues se utilizará una selección puramente aleatoria, donde cada par de padres que interviene en un apareamiento se selecciona directamente y de forma aleatoria de la población de la generación anterior.

El método de cruce sigue siendo el cruce de 2 puntos. Sin embargo, se introduce cierta variante que depende del usuario, pues este indicará (mediante el parámetro de entrada *numfilint*) el número de filas completas que desea se intercambien en los apareamientos, siendo estas elegidas aleatoriamente.

Por último, se describen conjuntamente los cambios introducidos en el método de mutación y reemplazamiento, ya que en este caso están relacionados. Dada una generación $i \in \{1, 2..., n_g\}$ concreta, para

formar la población de dicha generación se toman en primer lugar las soluciones generadas mediante el cruce y todos los individuos de la generación anterior, es decir de la generación $i-1$. Posteriormente, se muta mediante la mutación por intercambio de dos posiciones (en las posiciones no fijas) cada uno de los anteriores $2n_p$ individuos *nummut* veces (siendo $nummut \geq 1$ un parámetro de entrada proporcionado por el usuario), preservando siempre una copia de los individuos antes de la mutación. Por lo tanto, tras las *nummut* mutaciones, se tendrán $2n_p \cdot nummut$ soluciones candidatas. Finalmente, sobre dicho conjunto de soluciones, se aplica un método de reemplazamiento elitista sin repetición, en el cual se eliminan las soluciones repetidas y se forma la población de la generación i con las n_p mejores soluciones restantes.

Nótese que no será necesario proporcionar una tasa de mutación δ, ya que esta se aplicará al menos una vez.

Aplicando esta nueva versión del algoritmo genético a *Sudokus* de dificultad fácil, con parámetros de entrada $n_p = 600$, $n_g = 200$, $nfilint = 1$ y $nummut = 2$ se han obtenido la gráficas similares a la de la Figura 27, con coste mínimo global que coincide con el óptimo y un tiempo medio de computación de 46.0840 segundos.

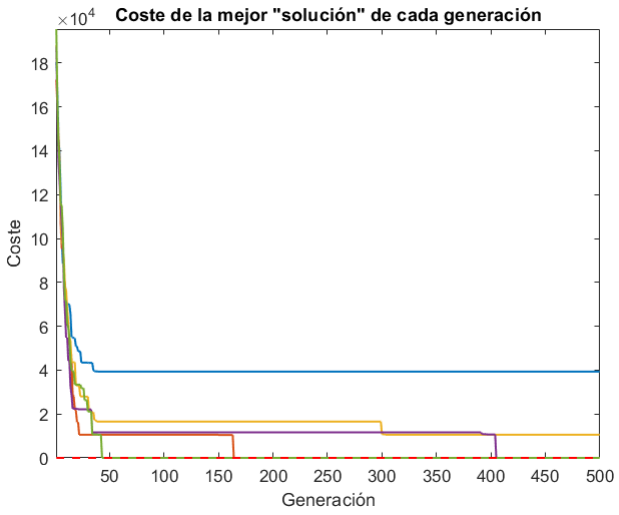

Figura 27: Evolución del coste de la mejor solución generada en cada iteración para un *Sudoku* de dificultad fácil y parámetros de entrada $n_p = 600$, $n_g = 200$, $nfilint = 1$ y $nummut = 2$.

Como se puede apreciar en la Figura 27, se ha alcanzado la solución óptima en tres ejecuciones distintas. Sin embargo, la convergencia prematura del algoritmo sigue latente. Una posible explicación a la obtención de la solución óptima utilizando esta nueva versión, podría ser que mutar *numut* veces todas las soluciones candidatas de cada generación, podría llevar a la solución óptima, ya que las soluciones correspondientes a mínimos locales suelen diferir de la óptima en pocas posiciones, por lo que aplicando repetidas veces la mutación aumentarían las posibilidades de intercambiar los números problemáticos y abandonar así el mínimo local.

No obstante, este algoritmo modificado parece hallar con relativa sencillez y frecuencia la solución óptima de *Sudokus* fáciles, por lo que se aplica también a *Sudokus* difíciles con el objetivo de analizar su desempeño. En concreto, se han utilizado exactamente los mismos parámetros que en el caso del *Sudoku* fácil, es decir $n_p = 600$, $n_g = 200$, $nfilint = 1$ y $nummut = 2$, obteniéndose gráficas similares

a la representada en la Figura 28, donde el coste mínimo global ha sido de 11231 y el tiempo medio de computación de 123.0502 segundos.

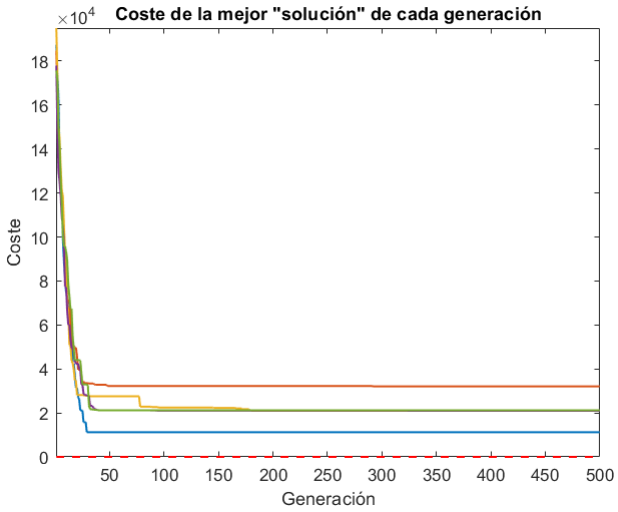

Figura 28: Evolución del coste de la mejor solución generada en cada iteración para un *Sudoku* de dificultad difícil y parámetros de entrada $n_p = 600$, $n_g = 200$, $nfilint = 1$ y $nummut = 2$.

En el caso de *Sudokus* difíciles, no siempre proporciona la solución óptima. A esto último hay que sumarle la persistencia de la convergencia prematura, que en esta ocasión se produce hacia soluciones de peor calidad respecto a aquellas hacia las que se producía en *Sudokus* fáciles. Por lo tanto, esto último podría indicar que el algoritmo genético modificado implementado, con estos parámetros de entrada en particular, tiende a proporcionar soluciones de peor calidad a medida que la dificultad de los *Sudokus* a los cuales se aplica aumenta. De hehcho, en [13] se pretendía hacer uso de su algoritmo para clasificar *Sudokus* en función de su dificultad.

Finalmente, la convergencia prematura que se presenta en la resolución de ambos *Sudokus*, parece la causa principal de la no obtención de soluciones óptimas. Además, en aquellos casos en los que se alcanza el coste óptimo son necesarios parámetros de gran magnitud (valores altos de n_p y n_g). Por lo tanto, un aspecto que se podría estudiar y tratar de mejorar en futuros trabajos sería alterar el método de mutación empleado, ya que podría ser la mejor alternativa para abandonar los mínimos relativos y poder alcanzar el mínimo absoluto. En particular, sería interesante la implementación de un método de mutación que incorporase algún tipo de información heurística sobre qué posiciones pueden ser las erróneas para poder aplicar la mutación a zonas más correctas y maximizar las posibilidades de obtener la solución óptima.

3.2. Algoritmo de colonia de hormigas: Aplicación al *Nurikabe*

En esta Sección, se diseña un algoritmo de colonia de hormigas que se utiliza para resolver el juego de lógica *Nurikabe*. Además de diseñar el algoritmo, se analizan los resultados, se comentan posibles áreas de mejoras y se evalúa el desempeño global de este tipo de algoritmos en este problema.

3.2.1. Descripción del juego

Nurikabe (ver [10] y [15]) es un rompecabezas de origen japonés que fue presentado por primera vez en el año 1991 por la editorial de juegos de lógica Nikoli. El nombre del juego proviene del folclore japonés, donde *Nurikabe* representa un muro invisible que bloquea caminos y obliga a los viajeros a rodearlos. Inspirándose en dicho concepto del folclore, el objetivo del juego es construir un *muro* que divida un espacio dado.

El espacio de juego o tablero del *Nurikabe*, se representa mediante una cuadrícula rectangular de dimensiones variables con celdas blancas, conteniendo alguna de ellas números naturales. Una solución al rompecabezas consiste en colorear en negro las celdas blancas que no contengan número siguiendo las siguientes normas:

1. Las celdas negras deben formar una región continua llamada *muro*.
2. El *muro* no puede contener regiones coloreadas de tamaño 2×2 o más.
3. Cada celda blanca numerada debe formar una única región blanca disjunta, denominada *isla*, cuyo tamaño (en términos de celdas) debe ser igual al número indicado en la celda numerada. De esta regla se deduce que cada celda blanca debe pertenecer a única *isla* y que cada *isla* debe contener una única celda numerada.

En la Figura 29 se representa un tablero de *Nurikabe* y su solución.

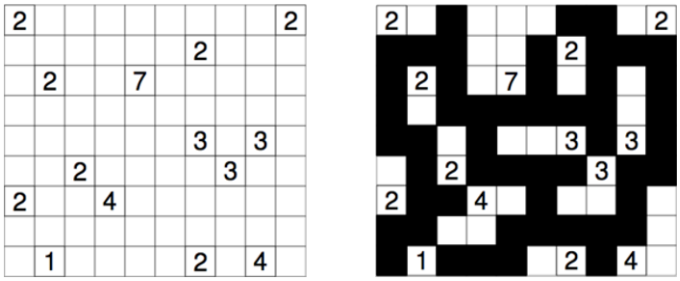

Figura 29: A la izquierda, tablero inicial de *Nurikabe* de tamaño 10×9. A la derecha, su solución. Fuente: *Imagen de Wikipedia*. (c.c.). https://en.wikipedia.org/wiki/Nurikabe_(puzzle).

3.2.2. Diseño del algoritmo

Nuevamente, el algoritmo de colonia de hormigas diseñado, utiliza gran parte de las técnicas y conceptos introducidos en el Capitulo 2. Además, en lineas generales se sigue el esquema presentado en el algoritmo de [10].

En primer lugar, los parámetros de entrada del algoritmo son los presentados a continuación:

- N: *Nurikabe* que se desea resolver.
- n_h: Número de hormigas artificiales en cada iteración.
- $iter$: Número de iteraciones.

- q_0: Parámetro perteneciente a $[0, 1]$ que define la probabilidad relativa de que cada hormiga realice movimientos utilizando uno de dos métodos disponibles.
- ρ: Parámetro de evaporación estándar perteneciente a $[0, 1]$ utilizado en la actualización de los rastros de feromonas.
- *evap*: Parámetro que indica la proporción de feromonas evaporadas.

A diferencia del procedimiento habitual de resolución de un *Nurikabe*, que consiste en construir el muro de celdas negras respetando las reglas, el algoritmo diseñado en esta Sección buscará la solución óptima expandiendo progresivamente cada una de las islas que posee el puzzle dado. Por lo tanto, la representación del tablero inicial, que sirve de punto de partida al algoritmo de colonia de hormigas, se realiza de la siguiente forma.

Dada una matriz N de dimensiones $n \times m$, correspondiente con el tablero inicial de un *Nurikabe*, se colorearan todas sus celdas de negro, a excepción de las *islas primitivas*, es decir las celdas numeradas. Dicha coloración, se realizará almacenando en todas las celdas de N no numeradas el valor -1. De esta forma, el muro del *Nurikabe* estará formado por todas las celdas de N a excepción de las *islas primitivas*. Por otro lado, la expansión de las *islas primitivas* se realiza cambiando el valor de las celdas de -1 a 0, es decir las islas estarán compuestas por celdas con valor mayor o igual a 0.

En definitiva, utilizando esta representación, una solución candidata del *Nurikabe* N se corresponderá con cualquier matriz de dimensión $n \times m$ cuyas entradas son -1, 0, o el valor de cada *isla primitiva*, manteniéndose las *islas primitivas* en la misma posición que ocupan en el tablero inicial. En la Figura 30 se presenta un ejemplo de la representación descrita.

2	-1	-1	-1	-1	-1	-1	-1	-1	**2**
-1	-1	-1	-1	-1	-1	**2**	-1	-1	-1
-1	**2**	-1	-1	**7**	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	**3**	-1	**3**	-1
-1	-1	**2**	-1	-1	-1	-1	**3**	-1	-1
2	-1	-1	**4**	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	**1**	-1	-1	-1	-1	**2**	-1	**4**	-1

2	0	-1	0	0	0	-1	-1	0	**2**
-1	-1	-1	0	0	-1	**2**	-1	-1	-1
-1	**2**	-1	0	**7**	-1	0	-1	0	-1
-1	0	-1	-1	-1	-1	-1	-1	0	-1
-1	-1	0	-1	0	0	**3**	-1	**3**	-1
0	-1	**2**	-1	-1	-1	-1	**3**	-1	-1
2	-1	-1	**4**	0	-1	0	0	-1	0
-1	-1	0	0	-1	-1	-1	-1	-1	0
-1	**1**	-1	-1	-1	0	**2**	-1	**4**	0

Figura 30: A la izquierda, tablero inicial del *Nurikabe* de la Figura 29 ilustrado según la representación descrita. A la derecha, solución del mismo *Nurikabe* representado de igual forma. Para una mejor identificación de las celdas, se han sombreado ligeramente las correspondientes al *muro*.

Una vez elegida la representación del puzzle, se define la función objetivo f que evaluará la calidad de las soluciones candidatas construidas y penalizará aquellas que incumplan alguna regla. Dicha función objetivo y sus componentes, se describen detalladamente a continuación.

Dada una solución candidata $X = (x_{ij})$ para $i = 1, 2, ..., n$ y $j = 1, 2, ..., m$, se define el conjunto

$$B(X) = \{x \in X : x = 0\},$$

que se corresponde con las celdas nulas de X. Además se define

$$BL(X) = \left\{ M \text{ submatriz } 2 \times 2 \text{ de } X : M = \begin{pmatrix} -1 & -1 \\ -1 & -1 \end{pmatrix} \right\},$$

que se corresponde con todas las submatrices de dimensión 2×2 de X cuyas entradas son todas iguales a -1. Además, suponiendo que el conjunto $I = \{v_1, v_2, ..., v_n : n \in \mathbb{N}\}$ representa el valor de cada una de las *islas primitivas* de N, se define la siguiente función:

$$g(X) = \sum_{v \in I} v - |B(X)|,$$

que calcula la diferencia entre la suma del valor de todas las *islas primitivas* de N (es decir, el área total que se espera que ocupen las islas de N) y la suma del número de celdas nulas de la solución candidata X (es decir, el área total que ocupan las islas construidas en la solución X). En este caso, $|\cdot|$ denota el cardinal de un conjunto.

Finalmente, la función objetivo f se define de la siguiente forma:

$$f(X) = g(X) + |BL(X)|.$$

Esta función comprueba que el área total de islas construidas se corresponda con el área total esperada y penaliza aquellas soluciones que contengan regiones de tamaño 2×2 en su muro. Además, permite identificar con facilidad la solución óptima, que será aquella con coste nulo. Sin embargo, esta función objetivo no verifica el cumplimiento de todas las reglas, ya que además de las ya mencionadas, las islas deben ser disjuntas y el muro debe ser continuo. No obstante, como se verá a continuación, el cumplimiento de estas dos reglas se impone directamente en la construcción de soluciones.

Una vez fijada la representación y elegida la función objetivo f, se debe elegir la implementación adecuada de las principales fases de que componen el proceso iterativo del algoritmo. En este caso, se ha decidido que el algoritmo de colonia de hormigas no disponga de ningún tipo de información heurística, por lo que los rastros de feromonas son el único factor que influye en los movimientos de las hormigas. Además, tampoco se dispondrá de un método de evaporación global de feromonas, aunque si se utilizará un método local estrechamente relacionado con la actualización de feromonas, que se realizará tanto local como globalmente. A continuación, se detallan los aspectos de mayor relevancia del algoritmo diseñado.

Para este algoritmo, los rastros de feromonas están asociados a las componentes del problema, es decir a todas las casillas no numeradas de N. Dichas feromonas se representan mediante la matriz de feromonas $\tau = (\tau_{ij})$ con $i = 1, 2..., n$ y $i = 1, 2..., m$, donde la posición (i, j) contiene el rastro de feromonas asociado a la celda (i, j) de N. Antes de comenzar con el proceso iterativo de construcción de soluciones, τ se inicializa tomando valores reales aleatorios del intervalo $(0, 1)$ y se almacena en τ_0 y τ_{max} el valor mínimo y máximo global de τ respectivamente.

Durante la fase de construcción de soluciones, se supone que cada una de las n_h hormigas de cada iteración construye únicamente una solución candidata. Esta construcción se lleva a cabo mediante el siguiente proceso iterativo.

Dada una hormiga h, se elige aleatoriamente una *isla primitiva* v del conjunto I y se almacenan en el conjunto C las coordenadas de todas las celdas horizontal y verticalmente adyacentes a todas las celdas que forman la isla, en este caso únicamente la celda v. El conjunto C recibe el nombre de conjunto de celdas candidatas, y está compuesto por las celdas susceptibles de añadirse a la isla originada por

v, por lo que se deben eliminar de C aquellas que unan diferentes islas horizontal o verticalmente o desconecten el muro de la misma manera. Esta eliminación de celdas, garantiza el cumplimiento de las dos reglas que faltaban por verificar.

Tras eliminar las celdas de C que derivarían en soluciones de N incorrectas, se elige la celda $\boldsymbol{c}^* \in C$ hacia la cual se expandirá la isla. Esta elección se puede llevar a cabo de dos formas, la primera de ellas tomando $\boldsymbol{c}^*$ como la celda de C con mayor concentración de feromonas, es decir

$$\boldsymbol{c}^* = \underset{\boldsymbol{c} \in C}{\operatorname{argmax}} \ \{\tau_{\boldsymbol{c}}\},$$

o eligiendo la celda $\boldsymbol{c}^*$ en función de la siguiente distribución de probabilidad discreta

$$p_{\boldsymbol{c}} = \frac{\tau_{\boldsymbol{c}}}{\sum_{\boldsymbol{k} \in C} \tau_{\boldsymbol{k}}},$$

donde $p_{\boldsymbol{c}}$ denota la probabilidad de elegir la celda $\boldsymbol{c} \in C$.

La aplicación de uno u otro de estos dos métodos de selección, se decide mediante un número real $q \in [0, 1]$ tomado aleatoriamente y el parámetro q_0, pues

$$\boldsymbol{c}^* = \begin{cases} \underset{\boldsymbol{c} \in C}{\operatorname{argmax}} \ \{\tau_{\boldsymbol{c}}\} & \text{si} \quad q < q_0 \\ \\ p_{\boldsymbol{c}} & \text{si} \quad q \geq q_0 \end{cases}$$

Una vez elegida la celda candidata $\boldsymbol{c}^*$ que se añade a la isla v en construcción, se realiza la actualización de feromonas local de la siguiente forma

$$\tau_{\boldsymbol{c}^*} = (1 - \xi)\tau_{\boldsymbol{c}^*} + \xi\tau_0,$$

es decir, se incrementa la concentración de feromonas de la celda $\boldsymbol{c}^*$ que se acaba de añadir a la isla v. Para el algoritmo diseñado, se supone $\xi = 0.1$ (ver [10]).

Este proceso iterativo se repite hasta que el tamaño de la isla generada por v es igual al valor de v o hasta que en algún caso $C = \emptyset$, donde se entiende que C es el conjunto de celdas adyacentes a cualquier celda que forme parte de la isla y que no unan islas ni desconecten el muro de N. Nótese que si en ninguna de las iteraciones del proceso de construcción se tiene que $C = \emptyset$, se habrán añadido a la isla v exactamente $v - 1$ celdas, ya que la celda correspondiente a la *isla primitiva* v también forma parte de la isla final.

Finalmente, una vez que la hormiga h ha terminado la construcción de la isla v, se desplaza aleatoriamente a otra *isla primitiva* de I que no haya sido elegida previamente y se construye la isla correspondiente utilizando el mismo proceso iterativo. Este procedimiento se repite hasta que la hormiga h haya construido todas las islas a partir de las *islas primitivas* de I.

Cuando las n_h hormigas han construido las soluciones correspondientes, al final de cada una de las *iter* iteraciones, se efectúa la actualización global de feromonas. Esta actualización, solamente afecta a las celdas que contienen el valor 0, es decir aquellas pertenecientes a una isla excluyendo las *islas primitivas*.

Para la implementación de este método, se evalúa la calidad de cada una de las n_h soluciones candidatas, que se corresponderá con el valor f_j para $j = 1, 2, ...n_h$, y se calcula la cantidad de feromonas que agregaría cada una de las n_h soluciones como $\Delta\tau_j = 1/f_j$. La actualización de feromonas, únicamente sucede si para algún $j \in \{1, 2, ..., n_h\}$ se cumple $\Delta\tau_j > \tau_{max}$, luego $\tau_{max} = \Delta\tau_j$ y se realizaría mediante la siguiente operación

$$\tau_{\boldsymbol{b}} = (1 - \rho) \cdot \tau_{\boldsymbol{b}} + \rho \cdot \tau_{max},$$

donde $\boldsymbol{b} \in B^*$ se corresponde con las coordenadas de cada una de las celdas que forman las islas de la solución j construida, exceptuando las *islas primitivas*. En caso de que existieran $i, j \in \{1, 2, ..., n_h\}$ con $i \neq j$ y $\Delta\tau_i, \Delta\tau_j > \tau_{max}$, entonces $\tau_{max} = \text{máx}\{\Delta\tau_i, \Delta\tau_j\}$.

Como se mencionó anteriormente, este algoritmo de colonia de hormigas carece de evaporación global de feromonas. Sin embargo, se implementa una evaporación local que afecta únicamente al valor de mayor concentración de feromonas τ_{max}, cuyo objetivo es evitar una posible convergencia prematura del algoritmo, ya que la no evaporación de este valor sumada a la posibilidad de incurrir en movimientos excesivamente elitistas en la construcción de las soluciones (es decir, cuando $q_0 \to 1$) podría provocar estancamientos en mínimos locales del problema. En particular, esta evaporación siempre se lleva a cabo, independientemente de la previa actualización global de feromonas, de la siguiente forma:

$$\tau_{max} = (1 - evap) \cdot \tau_{max},$$

Tras alcanzar el número máximo de iteraciones *iter* prefijado, el algoritmo proporciona una serie de parámetros de salida, en concreto un vector de matrices *MS* del mismo tamaño que N, que contiene la mejor solución construida en cada iteración, un vector fila *costeiter* de *iter* posiciones que almacena el coste de la mejor solución construida en cada iteración, la variable *tiempo* que representa el tiempo de computación empleado y una gráfica donde se representa la evolución del coste de la mejor solución a lo largo de las iteraciones.

3.2.3. Análisis de resultados

Para analizar y comparar el desempeño del algoritmo de colonia de hormigas a la hora de resolver diferentes *Nurikabe*, se ha aplicado a diversos puzles. Generalmente los puzles son etiquetados por tamaño y dificultad, y, aunque ambas características están relacionadas no se entrará a hacer dicha categorización.

El algoritmo diseñado, se aplica un total de cinco veces a cada puzle, que se proporciona en cada caso con la representación descrita en la Sección anterior ya aplicada para una mayor comodidad en la implementación.

En primer lugar, al aplicar el algoritmo a *Nurikabes* fáciles de tamaño 7×7 con los parámetros de entrada $n_h = 5$, $iter = 5$, $q_0 = 0.8$, $\rho = 0.2$ y $evap = 0.1$, se obtienen gráficas similares a las de la Figura 31, con un coste mínimo global de 0 y un tiempo de computación medio de 0.4103 segundos. Como se puede observar en la gráfica, en todas las ejecuciones se alcanza la solución óptima rápidamente.

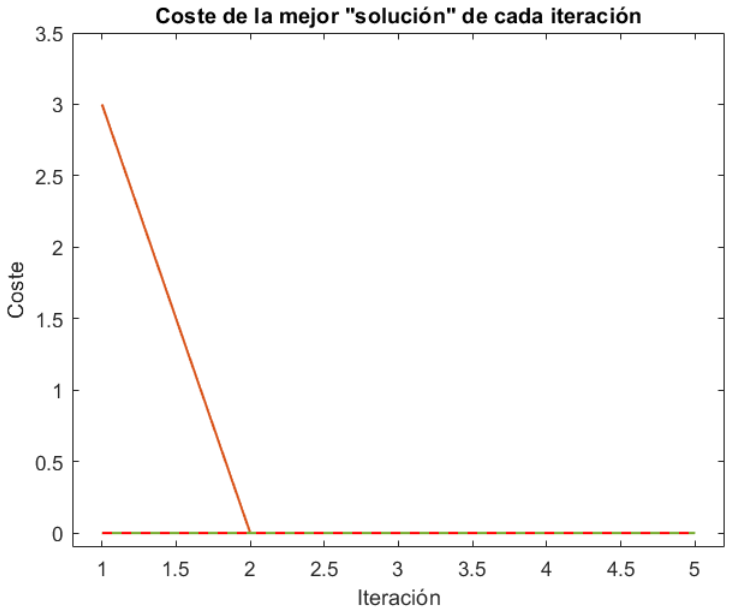

Figura 31: Evolución del coste de la mejor solución candidata construida en cada iteración de cada ejecución para *Nurikabes* fáciles 7×7 y los parámetros de entrada $n_h = 5$, $iter = 5$, $q_0 = 0.8$, $\rho = 0.2$ y $evap = 0.1$.

Para resolver *Nurikabes* difíciles de dimensión 10×10 se han utilizado los parámetros de entrada $n_h = 10$, $iter = 25$, $q_0 = 0.8$, $\rho = 0.2$ y $evap = 0.1$, obteniéndose resultados similares a los de la Figura 32, un coste mínimo global de 0 y un tiempo medio de computación de 21.34 segundos.

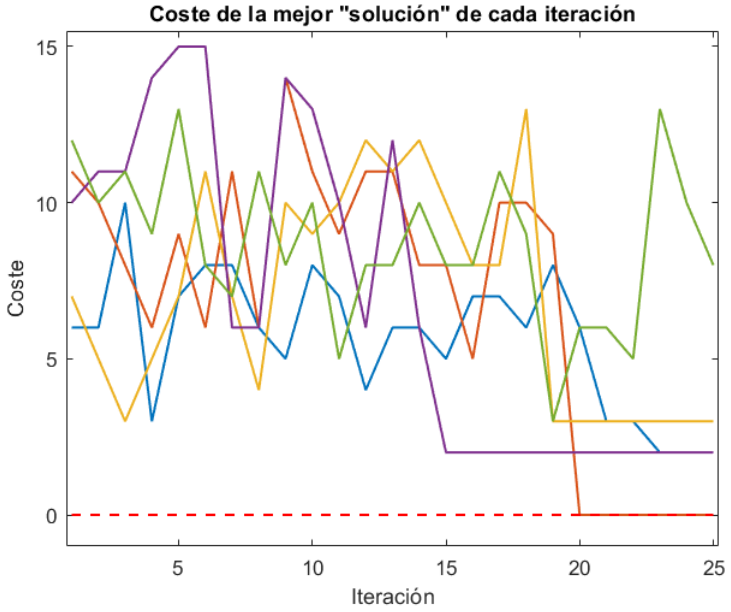

Figura 32: A la izquierda, evolución del coste de la mejor solución candidata construida en cada iteración de cada ejecución para *Nurikabes* difíciles 10×10 y los parámetros de entrada $n_h = 10$, $iter = 25$, $q_0 = 0.8$, $\rho = 0.2$ y $evap = 0.1$.

Para la resolución de *Nurikabes* difíciles de dimensión 15×15 se han empleado los parámetros $n_h = 10$, $iter = 60$, $q_0 = 0.8$, $\rho = 0.2$ y $evap = 0.1$, con los cuales se ha obtenido resultados similares a los de la

Figura 33. El coste óptimo ha sido de 0 y el tiempo medio de ejecución de 51.7299 segundos.

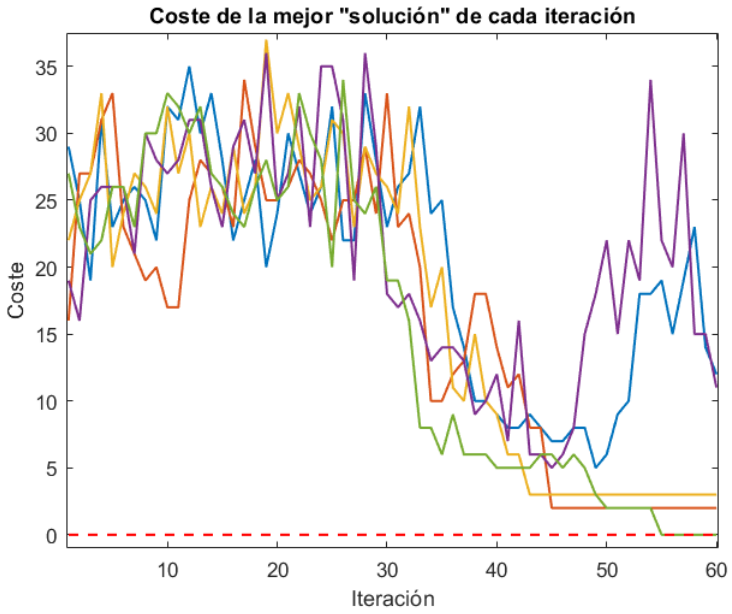

Figura 33: A la izquierda, evolución del coste de la mejor solución candidata construida en cada iteración de cada ejecución para *Nurikabes* difíciles 15×15 y los parámetros de entrada $n_h = 10$, $iter = 60$, $q_0 = 0.8$, $\rho = 0.2$ y $evap = 0.1$.

Como se observa en las gráficas obtenidas para *Nurikabes* de dimensiones 10×10 (Figura 32) y *Nurikabes* 15×15 (Figura 33) ambos etiquetados como difíciles, el algoritmo de colonia de hormigas sigue alcanzando la solución óptima. Sin embargo, a diferencia de lo ocurrido con los puzles de dificultad fácil (Figura 31), dicha solución se alcanza en un mayor número de ejecuciones.

En la gráfica del *Nurikabe* difícil 10×10 (Figura 32), se observa una convergencia prematura en varias de las ejecuciones que no alcanzan el coste óptimo, lo cual se podría tratar de solucionar disminuyendo el valor de q_0 y aumentando el de $evap$, para fomentar así la exploración nuevas soluciones candidatas y evitar, en la medida de lo posible, un algoritmo excesivamente elitista. No obstante, modificar estos parámetros no garantiza una mejora en la calidad de las soluciones, ni la obtención de soluciones óptimas con cierta asiduidad y la posibilidad de una convergencia prematura no parece un problema excesivamente crítico.

Observando la gráfica del *Nurikabe* difícil 15×15 (Figura 33), se aprecia un empeoramiento en la calidad de las soluciones. El algoritmo únicamente construye la solución óptima en una de las ejecuciones, mientras que en otras dos alcanza mínimos locales. Dichos mínimos locales, al igual que en el caso del anterior puzle, podrían ser asumibles, pues se encuentran muy próximos al coste óptimo y podrían indicar que las soluciones construidas difieren en muy pocas celdas de la solución óptima.

Por lo tanto, analizando conjuntamente los resultados obtenidos, se puede concluir que el algoritmo de colonia de hormigas diseñado parece ser eficiente en términos cualitativos para este tamaño y dificultad de puzles, en particular para aquellos de tamaño reducido y baja dificultad. Por otra parte, tanto el coste de la mejor solución construida como el tiempo medio de computación, aumentan al elevarse la dimensión y la dificultad del *Nurikabe*, lo cual podría indicar cierta conexión entre el comportamiento del algoritmo empleado y las características del puzle (dificultad y tamaño). Por último, señalar que

la combinación de parámetros adecuada para cada caso se ha determinado heurísticamente. Esto último, ha ayudado a conjeturar que, manteniendo un número de hormigas relativamente bajo, una alta predisposición a elegir el método de movimiento más elitista y unos parámetros de evaporación de feromonas contenidos, el algoritmo parece capaz de alcanzar con cierta frecuencia soluciones muy próximas, o incluso iguales, a la óptima variando únicamente el número de iteraciones.

Capítulo 4

Otros algoritmos

Para concluir la memoria, en este último Capítulo se presentan y desarrollan brevemente otros tipos de algoritmos. En concreto se introducen la *búsqueda tabú* y las *redes neuronales artificiales*.

4.1. Búsqueda tabú

En primer lugar, se introduce brevemente el funcionamiento de la búsqueda tabú para, posteriormente, describir alguna de sus características más relevantes. Finalmente, se presenta un esquema básico para implementar en la práctica este tipo de algoritmos.

4.1.1. Descripción del algoritmo

La búsqueda tabú, es un tipo de algoritmo heurístico basado en métodos de búsqueda local ampliamente utilizado en problemas de optimización combinatoria complejos. Este tipo de algoritmos ganó popularidad en la década de los años 80, cuando se desarrolló la teoría sobre algoritmos de enfriamiento simulado, pues proporcionó un nuevo enfoque sobre cómo explorar de una forma aleatoria, pero en cierto modo controlada, el espacio de soluciones (ver [3]).

En particular, la búsqueda tabú básica se basa en el enfoque de escalada de colinas o, por su traducción al inglés, *hill climbing*. Este enfoque consiste en un proceso de búsqueda local que, partiendo de una solución factible, mejora progresivamente dicha solución mediante ligeras modificaciones locales, de manera que en cada iteración se obtiene una mejor solución muy similar a la anterior hasta no ser posible obtener una mejora, es decir hasta que se alcance un mínimo local o, con suerte, global (ver [3]). Sin embargo, la búsqueda tabú implementa una serie de aspectos cuyo objetivo es permitir al algoritmo abandonar posibles mínimos locales alcanzados. Para llevar a cabo este cometido, la búsqueda tabú está dotada de cierta inteligencia, que utiliza como principal recurso la memoria tanto a corto, como a largo plazo. Por lo tanto, los algoritmos de búsqueda tabú continúan la búsqueda tras alcanzar mínimos locales, utilizando listas de soluciones candidatas o *listas tabú* (entre otros recursos), cuyo objetivo es evitar el regreso a soluciones ya visitadas sin incurrir en una gran pérdida de eficiencia.

Los conceptos fundamentales de la búsqueda tabú básica son la definición de un entorno de soluciones y una función objetivo f adecuados, la utilización de los distintos tipos de memorias y la correcta

aplicación de las listas tabú. A continuación, se desarrolla cada uno de estos conceptos aplicándolos a un problema de optimización genérico dado por

$$\min_{s \in S} f(s),$$

donde S denota al espacio de búsqueda del problema y f la función objetivo. Si se tratase de un problema de maximización, tras una simple adaptación los conceptos seguirían siendo válidos.

4.1.2. Entorno, movimientos y evaluación de soluciones

En primer lugar, para poder definir el entorno de una solución se debe considerar el espacio de búsqueda S. Habitualmente, en primera instancia el espacio de búsqueda se define como el conjunto de todas las soluciones factibles del problema tratado. Una vez definido el espacio de búsqueda, el entorno de una solución $s \in S$ se puede definir como un subconjunto $N(s)$ de S donde $N(s)$ representa el conjunto de todas las soluciones de S resultantes al aplicar una única transformación local a s. Por lo tanto, en cada iteración del proceso de búsqueda existirá un entorno $N(s_a)$ correspondiente a la solución actual s_a.

Pese a que las soluciones candidatas que conforman un entorno $N(s)$ son obtenidas tras aplicar una única transformación local, pueden existir numerosas transformaciones locales, llamadas *movimientos*, susceptibles de ser aplicadas. De esta forma, considerando M como el conjunto de todos los posibles movimientos, se puede caracterizar formalmente el entorno de una solución $s \in S$ como el conjunto $N(s) = \{\bar{s} = s \oplus m : m \in M\}$.

Tanto la definición, como la complejidad del conjunto de movimientos, son aspectos altamente dependientes del problema estudiado y cruciales en cualquier búsqueda tabú, por lo que si se dispone del conocimiento adecuado sobre el problema y sus particularidades se podría mejorar notablemente el proceso de búsqueda y, por con siguiente, la eficiencia del algoritmo. No obstante, en el caso de problemas que admiten una modelización de sus soluciones en forma de permutación, como puede ser el *Problema del Viajante*, existen ciertos movimientos básicos que pueden servir de punto de partida en el diseño de algoritmos de búsqueda tabú. Algunos de estos movimientos elementales son: el intercambio de elementos de una solución, la trasposición de soluciones y el desplazamiento de un elemento, es decir tomar un elemento de la solución y recolocarlo en una posición concreta, desplazando el resto de elementos a partir de dicha posición. Este último tipo de movimiento resulta ser bastante eficiente en problemas de planificación y diseño de horarios (ver [2]).

Si bien el espacio de búsqueda de un problema y el conjunto de movimientos M se suelen definir de tal forma que no existan, ni se puedan alcanzar mediante movimientos, soluciones no factibles, en ciertos problemas puede ser ventajoso o necesario eliminar dichas restricciones y permitir visitar soluciones incoherentes (ver [3]). La eliminación total o parcial de las restricciones del problema, proporciona un espacio de búsqueda mayor, lo cual favorece la exploración de distintas soluciones y ayuda a simplificar los entornos de las soluciones. Sin embargo, la eliminación de restricciones se debe acompañar de penalizaciones por el incumplimiento de restricciones. Estas penalizaciones pueden ser constantes, proporcionales a la importancia relativa de la restricción incumplida o variables dinámicamente durante el proceso de búsqueda.

Finalmente, la evaluación de las soluciones se realiza mediante la función objetivo f. Implícitamente, esta evaluación puede proporcionar información empírica sobre calidad de los movimientos de M, ayudando a cuantificar su valía en el problema estudiado. Por otro lado, la evaluación de las soluciones, a menudo no se realiza directamente, pues podría acarrear un alto coste computacional. Dada una solución $s \in S$ y su entorno $N(s)$, la evaluación de la solución candidata $\bar{s} \in N(s)$ dada por $\bar{s} = s \oplus m$ se realizaría mediante la operación $\Delta(s, m) = f(\bar{s}) - f(s) = f(s \oplus m) - f(s)$, ya que este cálculo puede ser más eficiente que la evaluación directa $f(\bar{s})$ debido a posibles simplificaciones algebraicas (ver [2]).

Por lo general, al igual que en la mayoría de algoritmos de búsqueda local, en la búsqueda tabú no se evalúan todas las soluciones candidatas de $N(s)$ en cada iteración, ya que se supone que dado un subconjunto $\bar{N}(s) \subset N(s)$, el algoritmo seleccionará con cierta inteligencia una de ellas. El subconjunto $\bar{N}(s)$ de soluciones que se evaluarán, llamado *lista de candidatos*, se puede elegir de diversas formas, siendo las más populares las siguientes (ver [2] y [3]):

- **Probabilísticamente:** En este caso $\bar{N}(s)$ se corresponde con una muestra aleatoria de $N(s)$ de tamaño reducido. Este método es el más sencillo y puede favorecer la exploración del espacio de búsqueda y la relajación de otros factores relevantes del algoritmo, como pueden ser las *listas tabú* (ver Subsección 4.1.3). Sin embargo, el principal inconveniente que surge al tomar $\bar{N}(s)$ como una muestra aleatoria de $N(s)$, es la posible pérdida de soluciones de gran calidad.

- **En función de los movimientos de M:** Si el conjunto de movimientos M permanece invariante durante todo el proceso de búsqueda, se puede dividir el conjunto $N(s)$ en subconjuntos en función del movimiento que se haya aplicado y evaluar únicamente uno de ellos en cada iteración.

- **En función de la calidad:** Bajo la suposición de que un movimiento de buena calidad para cierta solución seguirá siéndolo para soluciones similares a ella (ver [2]), se puede definir la lista de candidatos como un conjunto de soluciones de $N(s)$ ordenadas decrecientemente en función de su calidad. Esta ordenación se realiza en cierta iteración del proceso de búsqueda y, en futuras iteraciones, solo se consideraran las soluciones de mayor calidad. El orden de la lista de candidatos variará ligeramente con el paso de las iteraciones, a medida que se vayan introduciendo en ella soluciones cada vez más diferentes, por lo que serán necesarias evaluaciones periódicas totales del conjunto $N(s)$.

4.1.3. Memoria a corto plazo: *Listas tabú*

Como se mencionó con anterioridad, la principal diferencia de la búsqueda tabú con los algoritmos de búsqueda local básicos, es la utilización de una cierta capacidad de memoria.

En primer lugar, la primera idea que surge al tratar de aplicar una memoria a corto plazo en algoritmos de búsqueda local es la de verificar si una solución ya ha sido visitada previamente. No obstante, esta idea resulta muy costosa en términos computacionales y poco práctica en problemas complejos, pues la prohibición de regresar a soluciones ya visitadas con anterioridad podría imposibilitar la exploración de ciertas regiones del espacio de soluciones o incluso situaciones en las que el algoritmo finalice la búsqueda antes de tiempo por la imposibilidad de visitar nuevas soluciones. Por lo tanto, como alternativa a este método poco práctico surgen las *listas tabú*.

Las listas tabú implementan una memoria a corto plazo para evitar caer en la repetición de patrones durante el proceso de búsqueda. Este objetivo se puede lograr de dos formas distintas, evitando regresar a soluciones soluciones visitadas recientemente o evitando aplicar movimientos que reviertan movimientos recientes. La cantidad de información relativa a estas listas almacenada en la memoria a corto plazo es fija y finita, por lo que al ser más costoso y complejo el almacenamiento de soluciones completas, normalmente las listas tabú almacenan información sobre los últimos movimientos aplicados a las soluciones y prohíben sus movimientos inversos.

Para una correcta implementación de listas tabú de movimientos, debe ser posible alcanzar la solución óptima del problema desde cualquier solución inicial aplicando movimientos de M y, por simplicidad, se supone que cualquier movimiento $m \in M$ tiene inverso $m^{-1} \in M$. Puesto que este tipo de listas tabú limita la aplicación de un movimiento inverso durante un número pequeño de iteraciones, es posible que una vez transcurrido dicho número de iteraciones se aplique alguno de los movimientos inversos tabú. Sin embargo, si esto sucediese se espera que la solución haya sido modificada de tal forma que sea muy difícil volver a una solución ya visitada. En caso de que esto no fuese así, se espera que la nueva versión de lista de movimientos tabú sea diferente y pueda dirigir la búsqueda hacia regiones del espacio de búsqueda distintas (ver [2]). Por último, cabe la posibilidad de aplicar varias listas tabú en un mismo problema, en particular una por cada tipo de movimiento posible, lo cual puede ser beneficioso en ciertos casos para el proceso de búsqueda.

Para ilustrar el funcionamiento de estas listas de movimientos tabú, se utiliza un caso simplificado del *Problema del Viajante*. Suponiendo que en una ruta se ha se ha modificando el orden en el que se visita la ciudad A, por ejemplo ha pasado de ser la segunda ciudad visitada a ser la quinta, varios movimientos que se podrían considerar tabú serían: volver a cambiar el orden de visita de A de quinta a segunda posición, considerar de nuevo que A es la segunda ciudad visitada o cambiar el orden de visita de A de quinta ciudad visitada a cualquier otra posición.

No obstante, en ocasiones las listas tabú prohíben soluciones de una calidad superior a todas aquellas alcanzadas anteriormente, por lo que en estos casos los algoritmos suelen ignorar el movimiento tabú que impida obtener dicha solución. A dicho movimiento habitualmente se le conoce como *movimiento de aspiración*. Cabe mencionar, que este no es el único criterio de aspiración existente, aunque si el más simple y popular.

Por último, se debe tener en cuenta la influencia del tamaño de las listas de movimientos tabú en el proceso de búsqueda del algoritmo. Por una parte, considerar un pequeño número de movimientos tabú provoca una focalización en ocasiones excesiva del algoritmo en una zona concreta del espacio de búsqueda, en detrimento de la exploración del mismo. En contraposición, utilizar un número elevado de movimientos tabú aumenta las posibilidades de encontrar soluciones de mejor calidad, pero un número excesivamente elevado podría suponer no alcanzar ni explorar ciertos mínimos locales. Por lo tanto, con el objetivo de beneficiarse de las ventajas de tener listas de movimientos tabú de distintos tamaños, se suelen modificar dinámicamente el numero de movimientos que pueden albergar durante el proceso de búsqueda. La elección dinámica del tamaño de las listas tabú puede dirigir rápidamente la búsqueda hacia soluciones de gran calidad, demostrando ser un recurso que produce algoritmos más eficientes que aquellos que utilizan listas de tamaño fijo (ver [2]).

4.1.4. Memoria a largo plazo

Dentro de los mecanismos que emplean memoria a largo plazo en la búsqueda tabú, destacan la *memoria basada en la frecuencia de movimientos* y la *memoria que obliga a realizar cierto movimiento* (ver [2]). El funcionamiento de ambas técnicas se describe para el caso particular en el que el conjunto de movimientos M del problema permanece invariable durante todo el proceso iterativo de búsqueda.

La *memoria basada en la frecuencia de movimientos*, generalmente penaliza las soluciones que se hayan obtenido mediante movimientos utilizados frecuentemente. Su principal objetivo es garantizar diversidad en la búsqueda sin necesidad de incurrir en una prohibición excesiva de movimientos, lo cual acarrearía un aumento del tamaño de las listas tabú y, por consiguiente, un aumento en el coste derivado del uso de la memoria a corto plazo. Entre las formas de penalización más comunes, se encuentran la prohibición total de aquellos movimientos de M que hayan alcanzado una frecuencia de uso máximo prefijada y las técnicas de penalización proporcionales a la frecuencia de uso del movimiento a partir del cual se ha obtenido cada solución. Esta última técnica parece ser más ventajosa, ya que pese a penalizar ciertos movimientos nunca los prohíbe categóricamente.

Equivalentemente a la duración de la memoria a corto plazo, en el caso de las técnicas de memoria basadas en la frecuencia, es necesario ajustar adecuadamente la penalización añadida. Dado un movimiento $m \in M$ y su frecuencia de uso ν_m en cierta iteración del proceso de búsqueda, la penalización añadida al coste de las soluciones obtenidas mediante m se puede realizar de maneara proporcional a ν_m o incluso añadiendo simplemente la penalización ν_m^2 (ver [2]). Sin embargo, en situaciones en las que el entorno $N(s)$ de una solución $s \in S$ sea demasiado grande, las frecuencias asociadas a cada movimiento de M tienden a ser más pequeñas, por lo que sería aconsejable completar la penalización añadiendo factores que dependan del tamaño de $N(s)$ o incluso de M, de forma que ninguna penalización pueda llegar a ser nula. Al igual que en la memoria a corto plazo, en este tipo de memoria también conviene tener en cuenta el criterio de aspiración, de manera que no se desechen soluciones de excelente calidad.

A pesar de que la memoria basada en la frecuencia de movimientos se ha descrito para problemas donde el tamaño del conjunto de movimientos M permanece invariable, es posible generalizar estos conceptos al caso en el que M varía durante la búsqueda. En este caso, en vez de almacenar en la memoria a largo plazo la frecuencia de uso de cada movimiento, se almacenaría la frecuencia de uso de ciertas características de cada movimiento.

Por otro lado, la *memoria que obliga a realizar cierto movimiento* es una técnica de memoria a largo plazo mucho más simple que la anterior, ya que únicamente consiste en aplicar forzosamente un movimiento $m \in M$ que lleva un cierto número de iteraciones sin utilizarse, independientemente de su frecuencia de uso global o su repercusión en la calidad de la solución resultante. Esta técnica puede servir de gran ayuda en problemas con espacios de búsqueda grandes, espacios de búsqueda donde las soluciones correspondientes a mínimos locales se encuentran muy dispersas o incluso para abandonar mínimos locales en los que se ha centrado en exceso la búsqueda.

4.1.5. Implementación general del algoritmo

Finalmente, en esta subsección se describe brevemente un esquema básico, similar al presentado en [3], que puede utilizarse como punto de partida en el diseño de algoritmos de búsqueda tabú. En particular, el esquema general está diseñado para un problema de optimización

$$\min_{s \in S} f(s),$$

donde f denota a la función objetivo del problema y S al espacio de búsqueda considerado. Se utilizará la siguiente notación: s solución actual, s^* mejor solución encontrada hasta el momento, f^* evaluación de s^* mediante la función objetivo, $N(s)$ entorno de s, T lista tabú, T_ν memoria a largo plazo, $\bar{N}(s)$ lista de candidatos de la solución s.

El esquema general es el siguiente:

1. Se inicializan los parámetros, es decir se elige aleatoriamente una solución $s_0 \in S$ y realizan las asignaciones $s = s^* = s_0$ y $f^* = f(s_0)$. Además, se supone $T = \emptyset$ y $T_\nu = \emptyset$.

2. Se calcula la lista de candidatos $\bar{N}(s)$ a partir del entorno $N(s)$, la lista tabú T y la memoria a largo plazo T_ν si fuese necesario.

3. Se elige s en $\bar{N}(s)$ mediante el criterio proporcionado por el usuario. Si $f(s) < f^*$ entonces se toma s como la nueva mejor solución encontrada, es decir $s^* = s$.

4. Se actualizan la lista tabú T y la memoria a largo plazo T_ν en función de las técnicas que se hayan decidido implementar en cada caso.

Los pasos 2, 3 y 4 se corresponde con el proceso iterativo de búsqueda de soluciones, por lo que estos mismos se repetirán hasta que el algoritmo llegue a la condición de finalización, que debe ser especificada por el usuario. De forma similar a otros algoritmos heurísticos presentados en esta memoria, entre las condiciones de finalización de la búsqueda más habituales se encuentran alcanzar un número de iteraciones máximo, realizar un número de iteraciones prefijado sin obtener mejoras en la solución u obtener una solución de cierta calidad.

Nótese que en este esquema se hace uso de memoria a corto plazo (mediante listas tabú) y a largo plazo, con el objetivo de que el esquema sea lo más genérico posible. En la práctica, la decisión de utilizar ambos tipos de memoria o no está muy ligada a las necesidades de cada problema, al igual que la definición del conjunto de movimientos M, razón por la que no se especifica en este esquema.

4.2. Redes neuronales artificiales

En esta Sección, se describen brevemente los algoritmos de redes neuronales artificiales y su relación con la biología. Posteriormente, se desarrolla alguno de los conceptos más relevantes de la teoría general de este tipo de algoritmos.

4.2.1. Descripción

Los algoritmos de redes neuronales artificiales, son un tipo de algoritmos inspirados en la estructura y funcionamiento del cerebro humano, capaces de aprender y adaptarse a la resolución problemas de diversas características y una alta complejidad. En consecuencia, para comprender el funcionamiento primitivo de este tipo de algoritmos, se debe estudiar brevemente la naturaleza de las redes neuronales biológicas.

A grandes rasgos, el cerebro humano está compuesto por millones de neuronas interconectadas entre si, donde cada una de ellas realiza tareas simples que habitualmente se corresponden con dar respuesta a estímulos recibidos. Sin embargo, la alta interconexión de estas células, permite llevar a cabo tareas de una alta complejidad de manera prácticamente inmediata. En lineas generales, una neurona esta compuesta por un cuerpo celular o soma, múltiples ramificaciones o dendritas, que permiten la recepción de información procedente de otras neuronas y una prolongación más o menos larga, llamada axón que se encarga del envío de señales nerviosas. Cuando el estímulo total que recibe una neurona supera cierto umbral, esta se activa y envía un nuevo estimulo a las neuronas contiguas.

Por lo tanto, las redes neuronales artificiales se componen de neuronas artificiales con un funcionamiento análogo al biológico, es decir poseen un cuerpo neuronal artificial llamado *nodo* encargado del procesamiento de la información recibida y del envío de señales a otras neuronas artificiales y conexiones que se encargan de transmitir señales a otras neuronas artificiales. Normalmente, dichas conexiones son unidireccionales y tienen asociados ciertos pesos que afectan de diversas formas a las señales enviadas o recibidas. En los algoritmos de redes neuronales, únicamente existen dos estados posibles para cada neurona, que son el estado de *activación* y el de *desactivación*. Sin embargo, en cierto casos es conveniente considerar un tercer estado *desconocido* (ver [1]).

A continuación, se detallan alguno de los conceptos más relevantes y comunes en la mayoría de algoritmos de redes neuronales artificiales.

4.2.2. Organización de las redes neuronales y características de los nodos

Por lo general, las redes neuronales artificiales se organizan en capas, en particular en capas de entrada, capas de salida y capas ocultas, también llamadas simplemente capas (de las cuales puede haber varias). El primer paso que se debe realizar en la modelización de algoritmos de redes neuronales, es el de decidir cuántas capas habrá en la red y de qué tipo, cuántos nodos habrá en cada capa y el tipo de conexiones se que establecerá entre los nodos. En ciertos casos (ver [7]), la elección de estos factores se puede apoyar en otros algoritmos, como por ejemplo los algoritmos genéticos, para tratar de realizar la mejor elección posible.

En función del tipo de conexión entre nodos elegida, una red neuronal artificial se puede clasificar como *red de avance* o *red de retroalimentación*. Las redes de avance son aquellas en las cuales las conexiones son unidireccionales, por lo que la señal se transfiere en una única dirección. Por el contrario, las redes de retroalimentación no son unidireccionales, es decir es posible que una conexión de salida de un nodo sea la conexión de entrada a un nodo anterior al actual, de forma que la señal puede transferirse cíclicamente.

Debido su relativa simplicidad y a que son uno de los tipos de redes más utilizados, se emplea la red neuronal de perceptrones de una capa para ilustrar el funcionamiento básico de un nodo. En particular, puesto que esta es una red de avance, cada una de sus neuronas artificiales o *perceptrones* tiene un número finito de conexiones de entrada y una única conexión de salida.

Por lo tanto, dado un *perceptrón*, sus componentes son el nodo y un número finito de conexiones que vienen asociadas a un peso proporcional a la relevancia de la conexión (ver Figura 34). Para este tipo de neuronas artificiales, si al calcular la suma ponderada de las señales recibidas se supera el umbral T del nodo, este se activa y envía la señal mediante la función de transferencia f. Este proceso, se suele representar mediante la siguiente expresión:

$$y = f\left(\sum_{i=1}^{n}(w_i x_i) - T\right),$$

donde y denota la conexión de salida del nodo, x_i cada una de las conexiones de entrada y w_i los pesos asociados a cada conexión de recepción para $i = 1, 2, ..., n$.

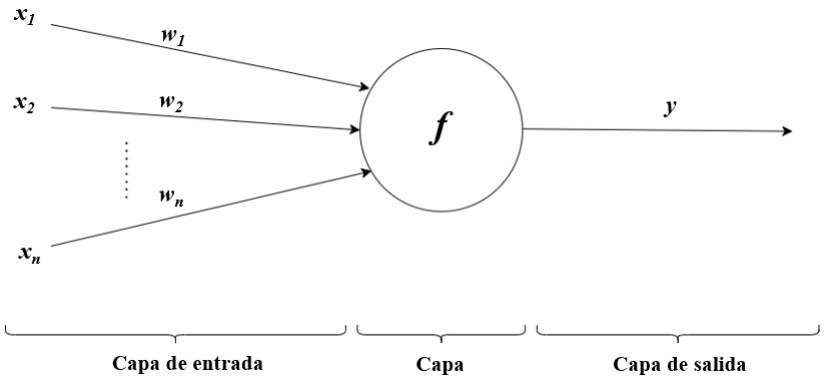

Figura 34: Representación de los componentes de un perceptrón de una red neuronal con una sola capa y n conexiones de entrada.

La definición de la función de transferencia, es un aspecto muy dependiente del problema tratado y del tipo de algoritmo que se desee diseñar. No obstante, puesto que la gran mayoría de problemas no son linealmente separables, las funciones de transferencia no lineales suelen ser de mayor utilidad (ver [7]). Alguna de las funciones de transferencia no lineales más utilizadas para este tipo de redes neuronales son la función escalón

$$y = \begin{cases} 1 & \text{si} \quad \sum_{i=1}^{n} w_i x_i > T \\ \\ 0 & \text{si} \quad \sum_{i=1}^{n} w_i x_i < T \end{cases}$$

o la función sigmoide

$$f(x) = \frac{1}{1 + e^{-x}}.$$

Una vez descrito el funcionamiento de un perceptrón, se puede generalizar al caso de una red neuronal de una capa con m perceptrones. Usando la misma notación y suponiendo que todos los perceptrones

poseen n conexiones de entrada, los pesos asociados a cada una de ellas se pueden representar mediante la siguiente matriz de pesos

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{pmatrix},$$

donde cada entrada $w_{i,j}$ indica el peso asociado a la conexión de entrada $j \in \{1, 2, ..., n\}$ del perceptrón $i \in \{1, 2, ..., m\}$. Suponiendo que el vector $T = (T_1, T_2, ...T_m)$ contiene el umbral de activación para cada uno de los m perceptrones y considerando f como la función de transmisión, el proceso de activación se generaliza de la siguiente forma

$$y_i = f\left(\sum_{j=1}^{n} (w_{i,j}x_j) - T_i\right) \quad \forall i \in \{1, 2, ..., m\}.$$

Mencionar, que la generalización de este procedimiento al caso de redes con varias capas es inmediato, basta con considerar la salida de la capa anterior como la entrada en la siguiente capa.

4.2.3. Aprendizaje y entrenamiento

La fase crítica en el diseño de los algoritmos de redes neuronales, es la fase de entrenamiento o aprendizaje, cuyo objetivo es conseguir que la red diseñada sea capaz de imitar comportamientos o modelar y resolver problemas complejos. En términos de redes neuronales artificiales, el concepto de aprendizaje hace referencia al ajuste de los pesos asociados a las conexiones de cada uno de los nodos de la red, en particular este ajuste se realiza con el objetivo de que los algoritmos proporcionen soluciones lo más precisas posibles. En función de la tipología del aprendizaje, este se puede dividir en dos categorías: el *aprendizaje supervisado* y el *aprendizaje no supervisado*.

En el caso del aprendizaje supervisado, los algoritmos de entrenamiento proporcionan a las redes neuronales un problema de entrenamiento mediante conjuntos de datos formados por distintas señales de entrada y salida así como soluciones o *respuestas* óptimas. Por lo tanto, en este caso se busca minimizar el error entre la respuesta proporcionada por el algoritmo y las respuestas óptimas proporcionadas. Cabe destacar la importancia del conjunto de datos de entrenamiento, pues debe ser lo más representativo posible del tipo de problemas a los cuales se aplicará la red neuronal, ya que de lo contrario la red podría proporcionar resultados erróneos o poco precisos. En resumen, las redes neuronales que utilizan métodos de aprendizaje supervisado, deben ser entrenadas mediante conjuntos de datos dados hasta que proporcionen resultados óptimos o muy similares a ellos con cierta frecuencia, proceso para el cual se ajustan los pesos asociados a las conexiones. De esta forma, se espera que la red diseñada proporcione respuestas de calidad a problemas con características similares a las del problema de entrenamiento usado.

Por el contrario, en el aprendizaje no supervisado la red neuronal no recibe conjuntos de datos de entrenamiento, por lo que este tipo de redes intenta, mediante repetidas implementaciones, descubrir

autónomamente tendencias y patrones en los datos del problema donde se aplican para así proporcionar respuestas adecuadas.

La elección del tipo de aprendizaje, es un aspecto de la modelización muy dependiente del tipo de red que se desea diseñar y la tipología de los problemas a los que se aplicará. Sin embargo, hay procedimientos de aprendizaje supervisado versátiles que se pueden emplear en una amplia gama de problemas y que sirven de base para algoritmos de aprendizaje más complejos. Entre estas técnicas de aprendizaje destacan las siguientes (ver [7]):

- **Métodos de corrección de errores:** Dada una red neuronal con k capas, n nodos en cada capa y en la cual se asume por simplicidad que todos los nodos tienen una única conexión de salida y m conexiones de entrada, se denota mediante y_j^k el estado o señal enviada por el nodo $j \in \{1, 2, ..., n\}$ en la *k-ésima* capa. Suponiendo que la señal óptima correspondiente a dicho nodo para cualquier iteración se corresponde con el valor y_j^*, se puede definir (ver [7]) una función de error básica como

  $$\xi_j^k = y_j^k - y_j^*,$$

  es decir, ξ_j^k denota el error en la señal enviada por el nodo j en la iteración k. De esta forma, suponiendo que la matriz de pesos W se ha definido como en 4.2.2 y que por tanto W^k se corresponde con la matriz de pesos de la *k-ésima* capa, el ajuste de los pesos correspondientes a la conexiones del nodo j se determina mediante

  $$w_{j,t}^{k+1} = w_{j,t}^k - \lambda \xi_j^k \qquad \forall t \in \{1, 2, ..., m\},$$

  donde $\lambda \in (0, +\infty)$ es un parámetro constante llamado *ratio de aprendizaje*. Es decir, cuanto mayor sean el error cometido por el nodo j, menos importancia se le dará a sus conexiones de entrada en la siguiente iteración, por lo que disminuirá la probabilidad de que dicho nodo se active.

  Este ajuste de pesos, se realizará hasta que la red neuronal en cuestión converja. Cabe destacar, la dependencia entre la convergencia de la red neuronal y el valor de la constante λ (ver [7]).

- **Métodos de vecino más cercano:** En este caso, dado un conjunto de datos de entrenamiento E finito, se almacena explícitamente en la memoria de la red neuronal que se desea entrenar. Dado una dato de prueba $x_j \in E$, su *vecino más próximo* $x^* \in E$ viene dado por la expresión

  $$x^* = \min_{j \in \{1,2,...,n\}} d(x^* - x_j),$$

  donde d es una distancia y n el tamaño del conjunto de entrenamiento utilizado, es decir $n = |E|$. De esta forma, el algoritmo de aprendizaje de vecino más cercano, asumirá que el dato de prueba x_t tendrá características muy similares, o incluso idénticas a las de x^*.

  Puesto que este método se basa principalmente en la similitud de los componentes del espacio del problema, es ampliamente utilizado en redes diseñadas para resolver problemas de clasificación. No obstante, el gran inconveniente de este método es el posible incremento del error relativo al aumentar la dimensión del problema (ver [7]).

Cabe destacar que, además de estos algoritmos de aprendizaje, existen muchos otros tipos y variantes más o menos complejas en función de su cometido. Alguno de los algoritmos de aprendizaje supervisado más relevantes son los algoritmos de *retropropagación*, de *minimización del error cuadrático medio* y el *algoritmo de Hopfield*. En caso del aprendizaje no supervisado, cabe destacar el método de *k-medias* y la *agrupación de Kohonen*. El desarrollo de todos estos algoritmos se puede encontrar en [1].

Finalmente, mencionar el interés personal por la aplicación de redes neuronales en la resolución de juegos, en particular en el ajedrez, por lo que aplicar esta técnica de forma similar a la presentada en [14] sería un posible área de estudio para futuros trabajos.

Bibliografía

[1] Albert Y. Zomaya. *Handbook of Nature-Inspired and Innovative Computing* Springer, 2006.

[2] Dréo, Pétrowski, Siarry & Taillard. *Metaheuristics for Hard Optimization. Methods and Case Studies.* Springer, 2006.

[3] Edmund K. Burke & Graham Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques.* Springer, 2005.

[4] Günter Rudolph. *Convergence analysis of Canonical Genetic Algorithms.* IEEE, 1994.

[5] Günter Rudolph. *Convergence of Genetic Algorithms in General Search Spaces.* IEEE, 1996.

[6] Huai-Kuang Tsai, Jinn-Moon Yang, Yuan-Fang Tsai & Cheng-Yan Kao. *An Evolutionary Algorithm for Large Traveling Salesman Problems.* IEEE, 2004.

[7] Jinming Zou, Yi Han & Sung-Sau So. *Overview of Artificial Neural Networks.* In David J. Livingstone *Artificial Neural Networks. Methods and Applications.* Springer, 2008.

[8] John H. Miller. *The coevolution of the automata in the repeated prisoner's dilema.* Journal of Economic Behavior & Organizarion. Elsevier, 1996.

[9] Kanchan Rani & Vikas Kumar. *Solving Travelling Salesman Problem Using Genetic Algorithm Based on Heuristic Crossover and Mutation Operator.* International Journal of Research in Engineering and Technology, 2014.

[10] Martyn Amos, Matthew Crossley & Huw LLoyd. *Solving Nurikabe with Ant Colony Optimization (Extended version).* ResearchGate, 2019.

[11] R. R. Sharapov & A. V. Lapshin. *Convergence of Genetic Algorithms.* Springer, 2005.

[12] Thomas Stützle & Marco Dorigo. *A Short Convergence Proof for a Class of ACO Algorithms.* IEEE Transactions on Evolutionary Computation, 2002.

[13] Timo Mantere & Janne Koljonen. *Solving and Rating Sudoku Puzzles with Genetic Algorithms.* ResearchGate, 2006.

[14] Vikrant Chole & Vijay Gadicha. *Hybrid Fly Optimization Tuned Artificial Neural Network for AI-based Chess Playing System.* Springer, 2022.

[15] Wikipedia. *Nurikabe.* (19/01/2024). https://en.wikipedia.org/wiki/Nurikabe_(puzzle).

[16] Wikipedia. *Sudoku.* (19/01/2024) https://en.wikipedia.org/wiki/Sudoku.