UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

FACULTAD DE CIENCIAS SOCIALES Y JURÍDICAS DE ELCHE

GRADO EN ESTADÍSTICA EMPRESARIAL

TRABAJO FIN DE GRADO

4ºCURSO

METAHEURÍSTICAS INSPIRADAS EN LA NATURALEZA

Autor: Giménez Buitrago, Alejandro

Tutora: Antón Sánchez, Laura

RESUMEN

En el presente trabajo de fin de grado hablaremos de los algoritmos metaheurísticos, para qué se utilizan, en qué consiste su comportamiento y los tipos de metaheurísticos que hay en función de cómo consiguen llegar a obtener una solución, centrando nuestro trabajo en aquellos que imitan comportamientos de sistemas naturales. Explicaremos con detalle cómo funciona una de las metaheurísticas inspiradas en la naturaleza más famosas, los algoritmos genéticos y algunas de las variantes que presentan. Expondremos algunas de las metaheurísticas más utilizadas y su funcionamiento básico; hablaremos de los algoritmos inspirados en colonias de hormigas (ACO), de los algoritmos basados en nubes/enjambres de partículas (PSO) y de la técnica que basa su comportamiento en el proceso que sufre un metal sólido al ser introducido en un líquido a altas temperaturas, Simulated-Annealing. Finalmente resolveremos un problema de temática real, el problema del viajante, mediante la utilización de algoritmos genéticos con el Software libre, *RStudio,* más concretamente con una de sus librerías desarrolladas específicamente para la aplicación de algoritmos genéticos, *GA.*

ÍNDICE

1. INTRODUCCIÓN

En la actualidad todo proceso tiene potencial para ser optimizado. Llamamos optimizar a la acción de desarrollar una actividad de la manera más eficiente posible, es decir la optimización consiste en llevar a cabo una tarea de la mejor forma posible basándose en algún aspecto, ya sea tiempo, coste… La optimización es aplicada a cualquier ámbito y a cualquier tipo de problema. Ésta aparece en los casos de minimización de tiempo, coste o de riesgo; o en la maximización de beneficio, calidad… (Westreicher, 2020).

Un gran número de problemáticas actuales de optimización son muy complejas y difíciles de resolver. Son problemáticas que no pueden ser resueltas de una manera exacta en tiempos razonables. A raíz de esto, aparece la forma de resolver estos problemas usando algoritmos aproximados, y aquí es donde aparecen los algoritmos heurísticos.

La heurística abarca el conjunto de técnicas o métodos para resolver un problema. Enfocada a esta serie de problemas trata de aplicar una serie de algoritmos creados específicamente para cada tipo de problema, que buscan la mejor solución posible con un coste de computación y tiempo mínimos.

"Las heurísticas específicas dependen del problema; están diseñadas y son aplicables a un problema particular" (Talbi,2009, XVII).

Los metaheurísticos son un tipo de algoritmos heurísticos que están siendo utilizados para resolver problemas de optimización de todo tipo cuando los métodos exactos no son aplicables. Los metaheurísticos pueden aplicarse a cualquier tipo de problema de optimización. Un algoritmo metaheurístico se puede definir como un proceso iterativo que guía y/o modifica las operaciones y/o soluciones de uno o más algoritmos heurísticos subordinados para producir soluciones de mayor calidad en un tiempo razonable.

Las metaheurísticas consiguen resolver problemas de gran dificultad explorando el espacio de soluciones, reduciendo el tamaño efectivo del espacio y explorando éste de manera eficiente (Talbi,2009).

Las técnicas metaheurísticas se pueden clasificar en 4 grandes grupos dependiendo de la forma en que buscan y/o construyen las soluciones. Aunque hay ciertas metaheurísticas que no pueden clasificarse en ninguno de estos grupos, éstos albergan la gran mayoría de las técnicas. Tenemos las metaheurísticas de búsqueda, constructivas, de relajación y evolutivas:

· Metaheurísticas de búsqueda: Estas técnicas proporcionan estrategias para explorar las estructuras del problema y mediante transformaciones y movimientos recorrer todo el espacio de soluciones en busca de la mejor de ellas.

· Metaheurísticas constructivas: Son técnicas que orientan todo su procedimiento a la obtención de una solución, analizando y seleccionando minuciosamente las componentes de esta.

· Metaheurísticas de relajación: Su método consiste en obtener simplificaciones del problema, para calcular soluciones en problemas menos restrictivos y más sencillos de resolver, y estas soluciones nos ayudan a encontrar la solución al problema original.

· Metaheurísticas evolutivas: Son técnicas que hacen evolucionar de forma simultánea a los valores de un conjunto de soluciones, de forma que, cada iteración nos acerca al óptimo del problema (Vargas et al., 2016).

Entre los metaheurísticos aparecidos destacan aquellos que imitan el comportamiento de sistemas naturales, y son en los que nos centraremos.

Uno de los más utilizados por sus excelentes resultados y por la variedad de problemas a los que se puede aplicar son los algoritmos genéticos que se encuentran dentro del grupo de metaheurísticas evolutivas.

Comenzaremos profundizando acerca de los algoritmos genéticos, hablaremos de las metaheurísticas inspiradas en colonias de hormigas, las metaheurísticas inspiradas en los comportamientos sociales de las bandadas de pájaros, bancos de peces, o colonias de abejas (Algoritmos basados en Nubes de Partículas) y de la metaheurística “Simulated Annealing”, y finalmente diseñaremos un algoritmo genético para resolver un problema de temática real.

2. ALGORITMOS GENÉTICOS

Los algoritmos genéticos al igual que muchos de los algoritmos metaheurísticos, perciben el problema como una búsqueda de la mejor solución en un entorno con infinidad de soluciones.

Éstos son algoritmos cuyos mecanismos de búsqueda imitan un determinado fenómeno natural: la evolución de las especies, basándose en el fenómeno de la selección natural a través de la herencia genética.

"Existen organismos que se reproducen y la progenie hereda características de sus progenitores, existen variaciones de características si el medio ambiente no admite a todos los miembros de una población en crecimiento. Entonces aquellos miembros de la población con características menos adaptadas (según lo determine su medio ambiente) morirán con mayor probabilidad. Entonces aquellos miembros con características mejor adaptadas sobrevivirán más probablemente." Darwin, El origen de las especies (1921).

En la naturaleza, cada una de las especies existentes vive con el constante problema de buscar su propia supervivencia, por lo que cada especie trata de conseguir una mejor adaptación al medio en el que habita, es decir, las especies viven en una búsqueda constante de mejora. Los algoritmos genéticos copian lo que la naturaleza hace.

Los algoritmos genéticos trabajan sobre una población de "individuos" que representan soluciones posibles al problema. A cada uno de estos individuos se le asigna un valor de adecuación que representa cuan de buena es esa solución. De forma semejante a lo que ocurre en nuestro mundo, los individuos se reproducen entre sí para producir nuevas soluciones, "hijos", de forma que cuanto mejor es el valor de adecuación mayor probabilidad poseerá ese individuo de ser seleccionado para la reproducción. Estos hijos, heredan características de cada una de las soluciones reproducidas. Los individuos con un menor valor de adecuación también pueden reproducirse, pero con una probabilidad menor.

2.1. PROCEDIMIENTO

El primer paso que debemos realizar es el de generar nuestra población inicial de soluciones. Cabe indicar que en los algoritmos genéticos en cada iteración poseeremos un conjunto de soluciones.

A continuación, evaluamos la población generada, asignando a cada una de las soluciones su correspondiente valor de adecuación. Una vez evaluadas nuestras soluciones, realizamos el proceso de selección, mediante el cual cada uno de nuestros individuos es copiado un número determinado de veces en función de su valor de adecuación, cuanto mejor sea tendrá un mayor número de copias; por lo que ahora disponemos de una nueva población que sustituye a la que teníamos. Este proceso de selección es el que nos asegura que los individuos con mejor nivel de adecuación participen más activamente en la construcción de la nueva población.

Después, comienza el proceso de cruce. Encontramos dos tipos de algoritmos genéticos dependiendo de como forman una nueva población a través de las generaciones, es decir, como se crea la nueva población después del proceso de cruce. Tenemos los algoritmos genéticos estacionarios y generacionales.

En los algoritmos genéticos estacionarios, los individuos se emparejan de forma aleatoria y cada una de esas parejas dispone de una probabilidad de reproducirse y formar nuevas soluciones, de forma que, si se reproducen, sus "hijos" les reemplazarán en la población; si no se reproducen quedan inalterados en esta iteración. Por lo que la nueva población dispondrá de individuos de distintas generaciones.

En el caso de los algoritmos generacionales, los individuos también se reproducen de la misma forma sólo que, en este caso, los hijos sustituirán a los padres por completo, y la nueva población únicamente estará formada por los hijos, por lo que no habrá interacción entre individuos de distintas generaciones (Soft Computing, 2006).

Al finalizar el proceso de cruce, nuestros individuos tienen la posibilidad de ser mutados, es decir, las soluciones se pueden alterar parcialmente, introduciendo nuevas características en la población o algunas que se habían perdido en el proceso de evolución.

Al terminar el proceso de cruce y mutación, volvemos a evaluar nuestra población y se comprueba si se cumple la condición de terminación. Ésta suele estar basada en un número de iteraciones, en el número de individuos/soluciones evaluadas, en la variabilidad de la población, etc. Esta dependerá del tipo de problema o del objetivo buscado. Si la condición no ha quedado cumplida volvemos al proceso de selección y repetimos la generación de otra nueva población hasta que nuestra condición de terminación se vea cumplida.

El procedimiento de un algoritmo genético básico podría resumirse mediante el esquema mostrado en la Figura 1:

```
Procedimiento Algoritmo Genético

    Generar_población_inicial
    Evaluar_población
    mientras no (condición_terminación)
        Selección_población
        Cruzar_población
        Mutar_población
```

Figura 1. Procedimiento de un algoritmo genético (Maroto et al., 2012).

A continuación, vamos a explicar con más detalle cada uno de los pasos a realizar para aplicar un algoritmo genético con éxito.

2.1.1. GENERAR POBLACIÓN INICIAL Y EVALUACIÓN

Antes de comenzar a aplicar un algoritmo genético, generando la población inicial, necesitamos realizar uno de los aspectos fundamentales para la posterior eficiencia del algoritmo, tenemos que diseñar una codificación para las soluciones. Una correcta codificación de las soluciones es imprescindible para el buen devenir del algoritmo genético, ya que sobre estas soluciones actuarán el resto de los procedimientos.

El algoritmo genético no trabaja sobre las soluciones directamente, éstas deben estar codificadas de forma equivalente al material genético de un individuo.

Cada solución estará compuesta de una serie de parámetros, conocidos como genes, que pueden ponerse uno tras otro formando una cadena de valores, lo que se conoce como cromosoma. Al conjunto de parámetros representados por un cromosoma se denomina genotipo. El genotipo contiene la información necesaria para construir un organismo, denominado fenotipo (Maroto et al., 2012).

Estas cadenas de valores, cromosomas, pueden ser representadas sobre cualquier tipo de alfabeto, codificación binaria, cadenas de enteros, cadenas de números reales, cadenas sin números…

Lo cromosomas pueden no contener la solución del problema, sino parámetros necesarios para el calculo de ésta. Contenga o no la solución directa al problema, hablaremos siempre de soluciones.

Una vez decidido la codificación de las soluciones dependiendo el tipo de problema al que están referidas, el siguiente paso, es el de crear la población inicial de individuos, de un determinado tamaño. La generación de estas soluciones puede realizarse de dos formas distintas: mediante un algoritmo heurístico o de forma aleatoria.

La ventaja principal de realizarlo de forma aleatoria es la rápida generación de las soluciones y la gran variedad de éstas, pero en ocasiones podemos crear soluciones muy mediocres que tarden mucho tiempo en converger en soluciones buenas.

La ventaja principal de realizarlo mediante un algoritmo heurístico es que las soluciones generadas poseen una mayor calidad, por lo que convergen de forma más rápida, pero puede derivar en la generación de una población con poca variedad, lo que es un grave problema por que puede provocar una convergencia prematura, quedando atrapado en un óptimo local, además de que el tiempo de generación mediante algoritmo heurísticos puede ser bastante alto y es posible que no nos convenga.

El siguiente paso es el de crear una función de evaluación que está relacionada con el objetivo buscado en el problema: maximizar beneficios, minimizar tiempo…

Ésta asignará a cada uno de nuestros individuos un valor de adecuación que, como hemos dicho anteriormente, nos indica cuan de buena es la solución respecto a las demás.

Estos son los dos pasos iniciales antes de comenzar con las numerosas iteraciones que tengamos que realizar, los siguientes pasos tendremos que realizarlos hasta que quede satisfecha la condición de terminación.

2.1.2. SELECCIÓN DE LA POBLACIÓN

A continuación, realizamos el proceso de selección de una nueva población. Esta selección representa lo que en la naturaleza es la selección natural de las especies, basada principalmente en que sobreviven los mejores.

En nuestro caso, aquellos individuos con un mayor valor de adecuación serán aquellos preparados para sobrevivir ante cualquier circunstancia, por lo que poseen una mayor probabilidad de sobrevivir y trasmitir su material genético a las próximas generaciones.

Recordamos que, para indicar esta mayor probabilidad de trasmitir su material genético, realizaremos copias de las soluciones generadas en un principio, de forma que los mejores individuos obtendrán un mayor número de copias que los de peor valor de adecuación.

A la hora de realizar el proceso de selección hay dos aspectos muy importantes a tener en cuenta para el buen funcionamiento de nuestro algoritmo genético: la diversidad de nuestra población y la denominada presión del proceso selectivo.

Es de severa importancia dar una oportunidad de reproducirse a aquellos individuos menos buenos, ya que dentro de su material genético pueden contener información útil para la futura solución. Esta idea de preservar también información recogida por individuos menos buenos nos lo marca la presión selectiva. Si el proceso selectivo es muy duro, si sólo consiguen pasarlo aquellos individuos extremadamente preparados para sobrevivir, la población perderá diversidad, y esto puede provocar que nuestro problema converja muy rápido estancándose en un óptimo local. Pero hay que tener cuidado pues si nuestro proceso selectivo es demasiado débil, si cualquier individuo por muy malo que sea su valor de adecuación consigue sobrevivir, sobrevivirán excesivas malas soluciones, y es posible que nunca consiga converger, ya que no iremos mejorando nuestra población, y nunca encontraremos la solución buscada. Por lo que se extrapola que debemos encontrar un compromiso adecuado entre la diversidad de nuestra población y el nivel de presión sobre nuestro proceso selectivo.

2.1.2.1. MÉTODOS DE SELECCIÓN

Hay diversas formas de realizar este proceso de selección, vamos a exponer algunas de ellas:

En primer lugar, tenemos la selección simple o selección por ruleta, en la cual basamos nuestra selección en la simulación de una ruleta, en el que cada solución tiene asignada una proporción de ésta, de forma que las soluciones con mayor valor de adecuación poseen una sección mayor. Una vez construida nuestra ruleta, lanzamos la bola tantas veces como soluciones tenemos, con cada una de las bolas lanzadas, obtendremos una nueva solución, que en su conjunto formarán una nueva población que sustituirá a la anterior, formada en este caso con un número mayor de buenas soluciones.

Luego tenemos el caso del muestreo determinístico, en el cual obtenemos la población buscada mediante el resultado entero del producto de la probabilidad que tienen de aparecer en esta nueva población, según su valor de adecuación, por el tamaño de la población. Cuando realicemos este proceso y la nueva población no se haya completado, debido a que sólo escogimos la parte entera y dejamos de lado la parte fraccionaria, ordenamos la población original en orden decreciente, y vamos escogiendo solución a solución hasta completar la nueva población.

En el muestreo estocástico funciona de forma similar al determinístico en la parte entera, pero difieren en el proceso de asignar las partes fraccionarias. Cuando lo realizamos sin reemplazamiento, la parte fraccionaria se utiliza como la probabilidad de obtener otra copia, así un valor como 3.8, tendrá 3 copias seguras y un 80% de obtener otra copia. Cuando en cambio lo realizamos con reemplazamiento, con las partes fraccionarias nos creamos una ruleta, en las que realizaremos tantos lanzamientos como soluciones falten para completar la nueva población.

Todos los métodos de selección explicados anteriormente presentan el mismo problema, durante las primeras iteraciones habrá individuos con valores de adecuación mucho mayores que otros, lo que provoca que estos individuos comenzarán a formar la gran parte de las poblaciones en muy pocas iteraciones, y como hemos expuesto, esto es un problema debido a la pérdida de diversidad muy temprana lo que nos provocará una convergencia muy prematura y nuestro encallamiento en óptimos locales. Esto provoca que, en muy pocas iteraciones, toda la población disponga de un valor cercano a la media, es decir, no diferenciaremos a los mejores individuos de los peores, y nuestra búsqueda del mejor individuo, se convertirá en una búsqueda a ciegas, debido a qué perderemos la referencia de nuestros mejores individuos.

Debido a esta problemática, pueden realizarse técnicas de escalado, las cuáles nos aseguraran que nuestra población no pierda diversidad de forma temprana, limitando el número de copias que obtienen los mejores individuos en las primeras iteraciones, y que nuestra población sea la “mejor” cuando estemos llegando a las últimas iteraciones, obteniendo los mejores individuos una mayor proporción de copias. Estas limitaciones u otorgación de mayor proporción se realiza mediante la modificación de los valores de adecuación.

Encontramos otros tipos de selección que no presentan el problema expuesto anteriormente, como son los mecanismos basados en el ranking y la selección por competición.

En el primero de ellos, ordenamos los individuos en orden creciente según su valor de adecuación, para asignarles, según su posición, un número de copias, proporcional al lugar del ranking no a su valor de adecuación.

En el segundo de ellos, seleccionamos en cada iteración un número k de individuos, utilizando el método de la ruleta, y de ellos únicamente nos quedamos con el de mayor valor de adecuación; y repetimos esta iteración hasta completar nuestra nueva población. Al incrementar el número de individuos seleccionados en cada iteración nuestra presión selectiva será mayor, por lo que un buen valor de k es 2, de esta forma escogeremos a dos individuos lanzando en dos ocasiones la ruleta y sólo uno de ellos pasará a formar la nueva población.

2.1.3. CRUCE O REPRODUCCIÓN DE LA POBLACIÓN SELECCIONADA

Nos centraremos en la explicación exhaustiva de los algoritmos genéticos estacionarios, es decir, podremos disponer de distintas generaciones acabado el proceso de cruce.

Cuando ya disponemos de nuestra nueva población después de la realización del proceso de selección, comienza el proceso de cruce, en el cual emparejamos de forma aleatoria a todas las soluciones de la población. Todas las parejas que se forman tienen la misma probabilidad de cruce. Si al juntar dos individuos, estos no se reproducen, estos individuos pasarán a formar parte de la nueva población ya que, como ya hemos indicado, pueden convivir diferentes generaciones en la misma población; pero si el cruce se lleva a cabo, estos formarán dos nuevos hijos que heredarán los genes de los dos padres combinándolos, y serán estos los que formen la nueva población, sustituyendo a sus “padres”. Por lo que el tamaño de la población siempre es el mismo.

El valor de la probabilidad de cruce suele estar entre 0.6 y 0.9, ya que debemos pensar que, si no hay cruces, la población será muy parecida a la anterior y podríamos tardar mucho en encontrar la solución buscada.

Debemos ser conscientes de que la combinación de los genes de los padres debe ser beneficiosa, no debe tratarse de una simple combinación de genes sin ningún sentido. El proceso de cruce tiene que estar bien diseñado, y depende de nuestro problema y de

nuestras soluciones codificadas, deberemos aplicar alguna de las numerosas técnicas de cruce desarrolladas hasta el momento.

2.1.3.1. TÉCNICAS DE CRUCE

La más sencilla de todas es el denominado cruce 1-punto. Teniendo las dos soluciones elegidas para el cruce, los padres, elegimos de forma aleatoria un punto de cruce *k*, tal que este punto no sea 0 ni superior al número de longitud de las soluciones, *l*. Una de las soluciones resultantes heredará los *k* primeros genes de uno de los padres y el resto del otro. La otra solución justo al revés, heredando los k primeros genes del padre que terminó el genoma del otro hermano, y el resto del que comenzó.

Hay un pequeño problema realizando el cruce 1-punto, y es que, si los genes que hacían buena a la solución se encontraban en las primeras y últimas posiciones, con el cruce 1-punto, los hijos nunca podrán heredar las características que hacían que sus padres estuvieran capacitados para sobrevivir, es decir, con el cruce 1-punto limitamos el conjunto de características que pueden heredar las nuevas soluciones.

Debido a este problema, se diseñó el cruce 2-puntos. Con esta técnica los puntos *k1* y *k2*, también son generados de forma aleatoria, de forma que $1 \leq k1 < k2 \leq l$. En este caso, la primera de las soluciones heredará los $k1$ primeros genes y los que van desde $k2+1$ hasta l, de uno de los padres, y desde $k1+1$ hasta $k2$ del otro de los padres. El otro de los hijos heredará los genes de forma totalmente inversa a su hermano.

De la misma forma que en la técnica de cruce 1-punto, podemos encontrar numerosas combinaciones de genes que no pueden heredar las nuevas soluciones, por lo que existen técnicas en las que aumentan el número de puntos de cruce, lo que se conoce como cruce multi-punto.

Existe otra técnica de cruce diferente a las anteriores, denominada cruce uniforme. Esta técnica se basa en lo que denominamos máscara de cruce. Generamos aleatoriamente una cadena de igual longitud que nuestras soluciones, compuesta únicamente de unos y ceros, de forma que generamos una máscara de cruce por cada uno de los cruces que realizamos. Para generar la primera de las nuevas soluciones, observamos posición a posición, si en la máscara se encuentra un 1, heredará el gen de uno de los padres y cuando salga 0 el

gen del otro de los padres. Para generar al otro hijo, actuaremos de la misma forma, pero en este caso cuando salga 1 heredará el gen del padre que antes se heredaba cuando había un 0.

Comparando las técnicas cruce 2-puntos y cruce uniforme, no han llegado a la conclusión de que una sea mejor que la otra, depende del tipo de problema y de la codificación de las soluciones.

Hay otros tipos de cruce que están indicados para los problemas en los que los valores de adecuación dependen únicamente del orden en el que aparecen los genes, los problemas tipo permutación. El más famoso de estos tipos de cruce se denomina PMX (Partially Matched Crossover). Un problema muy conocido que presenta esta particularidad es el problema del viajante de comercio (conocido como TSP por sus siglas en inglés, Travelling Salesman Problem), que resolveremos más tarde. En dicho problema el viajante debe visitar n ciudades distintas de manera que minimice la distancia recorrida o el coste. En este caso en particular, las soluciones estarán codificadas de forma que, si "Elche" aparece en la posición i, será la i-esima ciudad que visitemos. En el proceso de cruce de este problema, no se cruza el valor de cada uno de los genes, sino el orden en que aparecen dichos genes, por lo que si aplicáramos cualquiera de los cruces explicados anteriormente nos llevaría a nuevas soluciones no factibles.

Para explicar como funciona este cruce, debemos verlo en un ejemplo.

Veamos la aplicación del cruce PMX representado en la Figura 2. Como en el cruce dos-puntos, el primer paso es generar de forma aleatoria, dos puntos *k1* y k2 de forma que $1 \leq k1 < k2 \leq l$. Cada uno de los hijos hereda de uno de los padres los genes contenidos entre k1 y k2. Supongamos que para nuestro ejemplo *k1*=4 y *k2*=7:

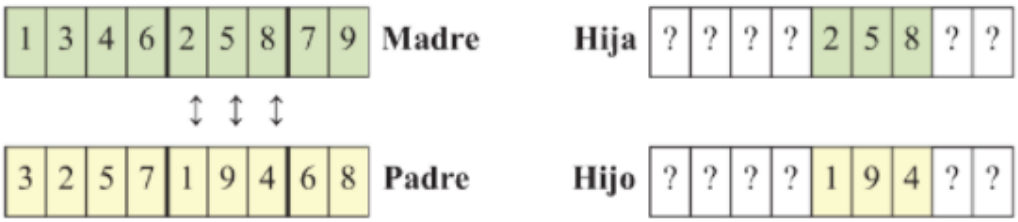

Figura 2. Cruce PMX A (Maroto et al., 2012).

En la imagen podemos observar que hemos fijado unos intercambios:

2 ↔ 1 5 ↔ 9 8 ↔ 4

Estos intercambios los utilizaremos en el tercer paso.

El siguiente paso, representado en la Figura 3, consiste en heredar del padre contrario al heredado en el caso anterior, los genes que todavía quedan libres en la nueva solución, y que no produzcan conflictos con los colocados en el primer paso, es decir, que no aparezca más de una vez el mismo número, ya que no queremos que se visite la misma ciudad dos veces.

Figura 3. Cruce PMX B (Maroto et al., 2012).

Nos queda el tercer y último paso (Figura 4), en el cual debemos rellenar aquellos genes que no se han colocado por estar en conflicto con otros genes. Es aquí donde utilizaremos los intercambios fijados anteriormente que nos ayudarán a heredar el resto de los genes.

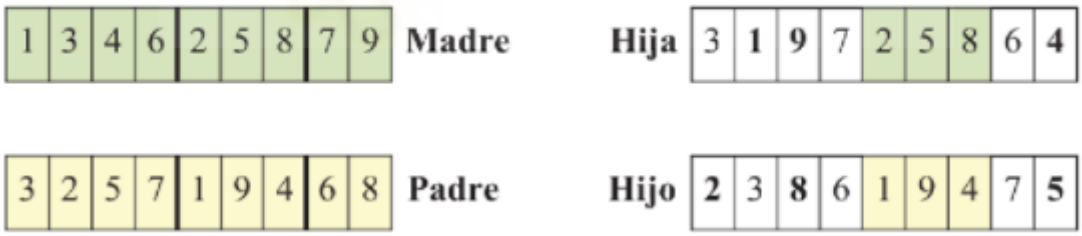

Figura 4. Cruce PMX C (Maroto et al., 2012).

Como podemos observar la hija debía heredar del padre el gen “2”, pero la hija ya había heredado este de la madre, observando los intercambios fijados, lo hemos cambiado por 1. Los mismo ocurre en todos los genes que faltan por colocar y que tendremos que intercambiar siendo acordes a los intercambios fijados (Maroto et al., 2012).

Dentro de los tipos de cruce adecuados para problemas como el TSP, tenemos un tipo de cruce que se asemeja al cruce PMX, el denominado *cruce por orden* (OX). En el cual el primer paso es igual que el cruce PMX, pero difieren en el modo que rellenan los genes

que faltan. En este caso, volviendo a la Figura 2, la hija ahora heredará los genes que faltan en el orden que se encuentran en los genes del padre, es decir, iremos rellenando en el orden: 3-7-1-9-4-6, y de forma igual en el hijo, pero con la madre.

Podemos encontrar otros cruces para este tipo de problemas, como es el cruce PBX o cruce basado en la posición, en el cual se selecciona al azar un subconjunto de posiciones del padre y se copian los valores de esas posiciones y los demás se rellenan con los valores de la madre, y para la realización del otro hijo se procede de la forma contraria, seleccionando al azar las posiciones de la madre y luego se rellenan los espacios vacíos con los valores del padre (Al-Shayea et al., 2020).

Otra operación de cruce adecuado para problemas como el TSP es el cruce de ciclo cruzado (CX) el cuál explicaremos mediante un ejemplo extraído de (Hussain et al., 2017) para entender su funcionamiento. Disponemos de los recorridos de los dos padres seleccionados para cruzarse:

$$P_1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8)$$

$$P_2 = (8\ 5\ 2\ 1\ 3\ 6\ 4\ 7)$$

Para crear a uno de los hijos empezaremos por el padre P_1 y para el otro empezaremos por la madre P_2. Para crear al primero de los hijos empezamos fijándonos en el primer valor del recorrido del padre, por lo que seleccionamos el 1. A partir del primer valor seleccionado empezamos una cadena que parará hasta que volvamos a un valor que ya hayamos colocado, de forma que ahora debemos fijarnos en P_2 para seguir la cadena y colocar en el hijo el valor observado en P_2 pero en la posición donde se encuentra ese valor en P_1, es decir:

$$H_1 = (1\ X\ X\ X\ X\ X\ X\ 8)$$

Lo que hemos realizado ha sido observar qué valor hay en la posición donde hemos colocado el primer valor para el nuevo hijo, observando en el padre distinto al que hemos copiado el valor, el 1 nos ha llevado al 8 de P_2, por lo que buscamos ese valor en P_1 y colocamos ese valor en la posición que se encuentra en P_1.El 1 nos lleva al 8, el 8 al 7 y el 7 al 4, y tenemos que parar debido a que el 4 nos lleva al 1 que ya está colocado.

De forma que nuestro primer hijo queda en la primera parte de esta forma:

$$H_1 = (1\ X\ X\ 4\ X\ X\ 7\ 8)$$

En la segunda parte rellenamos los valores restantes con los valores del otro padre por lo que el hijo 1 se quedará:

$$H_1 = (1\ 5\ 2\ 4\ 3\ 6\ 7\ 8)$$

El otro hijo se quedará de la siguiente forma, comenzando, en el primer valor del otro padre, el 8:

$$H_2 = (8\ \ 2\ 3\ 1\ 5\ 6\ 4\ 7)$$

El problema de este tipo de cruce es que hay veces que los hijos acaban siendo exactamente iguales a los padres.

Hemos explicado dos tipos de cruce, los explicados en primer lugar (*cruce 1-punto*, *cruce 2-puntos* y *cruce multi-punto*) y los explicados en último lugar (*PMX*, *cruce por orden (OX)*, *PBX y cruce de ciclo cruzado (CX*)). La diferencia principal entre estas dos formas de realizar el proceso de cruce es que en las explicadas en primer lugar no ha hecho falta conocer en absoluto el tipo de problema que resolvíamos, en cambio, en las explicadas en último lugar, hemos necesitado conocer el tipo de problema, es decir, hemos incorporado conocimiento específico del problema del viajante de comercio para evitar obtener soluciones no factibles. Por lo que la ventaja principal de incorporar información del problema en el proceso de cruce es que, evitando la generación de soluciones no factibles, reducimos el tiempo necesario para evaluar las soluciones más tarde.

2.1.4. MUTACIÓN

Cuando ya hemos realizado todos los cruces y tenemos nuestra nueva población, en las que tenemos soluciones de generaciones distintas, comienza el proceso de mutación, con el objetivo de introducir cierta variabilidad en nuestra población, introduciendo nuevas características a algunos individuos o introduciendo algunas que se perdieron durante el proceso evolutivo.

"Es decir, la mutación, a diferencia del proceso de cruce, es una especie de búsqueda a ciegas que pretende asegurar que en el espacio de soluciones no exista ningún punto con una probabilidad nula de ser examinado" (Maroto et al.,2012,281).

El proceso de mutación afecta a todos los genes de cada uno de los individuos que forman nuestra población después del proceso de cruce. Todos los genes disponen de la misma probabilidad de ser mutados, esta debe ser muy baja, mucho más baja que la probabilidad

de cruce. El proceso de mutación debe formar cromosomas válidos. El proceso de mutar un gen no es más que el de alterar su valor.

Al igual que el operador de cruce, la mutación también puede incorporar conocimiento específico del problema.

Al finalizar el proceso de mutación, vemos si hemos cumplido nuestra condición de terminación, sino es el caso, volvemos al proceso de selección de una nueva población.

3. METAHEURÍSTICAS INSPIRADAS EN COLONIAS DE HORMIGAS

La información de este apartado está extraída de dos trabajos finales de carrera basados en la aplicación de esta metaheurística (Revuelta, 2015) y (Vázquez, 2016).

Estas técnicas basan su procedimiento en el comportamiento que realizan las hormigas al conseguir alimento para su colonia. Las hormigas salen en busca de alimento, al principio todas siguen caminos aleatorios hacia la comida y las hormigas al caminar van depositando feromonas sobre el camino que realizan. Las hormigas poseen una memoria que les permite recordar el camino por el que han ido y debido a esto el camino de ida y vuelta lo realizan por el mismo camino, por lo que el depósito de feromonas es mayor.

A continuación, salen otras hormigas a recoger alimento, siguiendo algunas de éstas el camino realizado por alguna de las de antes atraídas por las feromonas, pero hay otras que deciden realizar caminos más cortos. Al paso del tiempo, los caminos más cortos dispondrán de un mayor número de feromonas debido a que al ser distancias menores, las hormigas tardan menos en realizar el recorrido dejando mayor número de feromonas pues el viaje se realiza con mayor asiduidad. Las feromonas se evaporan por lo que un camino que no ha sido visitado por hormigas hace tiempo, podría perderse. Por lo que finalmente, todas las hormigas atraídas por una gran cantidad de feromonas realizarán el camino óptimo hasta el alimento.

3.1. ACO-ANT COLONY OPTIMIZATION

Estas técnicas son usadas tanto para resolver problemas donde hay que encontrar caminos de coste mínimos, como para resolver cualquier problema donde se usan grafos para su representación, como es el caso del TSP.

Vamos a explicar cómo diseñar un algoritmo ACO y sus principales pasos básicos, mostrados en la Figura 5, sin introducirnos en las muchas variantes posibles.

Figura 5. Pseudocódigo de un ACO (Revuelta, 2015).

· **Fijar parámetros iniciales y creación de las hormigas artificiales:** En primer lugar, seleccionamos el número de hormigas que poseerá nuestra colonia, m, que dependerá del problema a resolver y su complejidad, a mayor complejidad, mayor número de hormigas. A continuación, debemos fijar un rastro de feromona inicial, τ_0, ya que cada uno de los arcos (i,j) que unen nuestros vértices deben tener una representación de un rastro de feromonas artificial para que nos sirva de guía. Este rastro inicial es constante e idéntico para cada uno de los arcos que disponemos en nuestro problema. Las aristas también pueden tener asociadas un valor que señale el coste/distancia de recorrer un arco, un valor que introduce información específica del problema a resolver, η_{ij}. Es de relativa importancia el coeficiente de evaporación de la feromona, ρ, ya que nos ayudará en nuestro proceso de actualización de la feromona. Por último, debemos fijar un criterio de parada que puede ser cuando se hayan realizado un número de iteraciones o hayamos llegado a un objetivo prefijado.

· **Construcción de la ruta:** Para construir la ruta, es necesario disponer de la probabilidad con la que una hormiga *k* se dirige de un vértice a otro o dicho de otra forma, la probabilidad con la que una hormiga recorre un arco, P_{ij}^{k}, que dependerá del rastro de feromonas presentes en ese arco, τ_{ij}, del parámetro asociado a ese arco que contiene información específica del problema a resolver, η_{ij}, y teniendo en cuenta, en cada uno de los casos, aquellos vértices a los que no podemos ir desde otros vértices, debido a

restricciones o debido a que el problema no sería factible. Por ejemplo, en el caso del TSP, la probabilidad de visitar un vértice ya visitado es 0, pues no queremos volver a pasar por los mismos lugares.

· **Resultados:** Los resultados obtenidos serán el coste/distancia del recorrido realizado por cada hormiga, C^k, y también la secuencia de vértices visitados por cada una de ellas, T^k.

· **Actualización de la feromona:** Este es un paso imprescindible para la efectividad de nuestro algoritmo. Cuando todas las hormigas creadas, m, finalizan su recorrido, aplicamos la regla de actualización del rastro de feromona asociada a cada uno de los arcos. La actualización de la feromona se puede realizar de muchas formas distintas, dependiendo de cada problema específico. A la hora de actualizar las feromonas entra en juego el coeficiente de evaporación del rastro de feromona, ρ. Este coeficiente está comprendido entre 0 y 1, y es constante. Por lo tanto, el nuevo rastro de feromona hallado en el arco (i,j) será: $\tau_{ij}^{t+1} = (1 - \rho) * \tau_{ij}^{t}$. A la fórmula se le añade el rastro de feromonas que las hormigas han dejado tras sí en todos los arcos (i,j) por los que han pasado en su recorrido. En algunos algoritmos el rastro de feromonas que depositan las hormigas cuando recorren un arco es constante por cada una de las hormigas que pasan, para marcar ese arco con un mayor atractivo para las futuras hormigas. En otros algoritmos, la cantidad de feromonas depositadas es proporcional al coste/distancia de los arcos, por lo que a menor coste/distancia mayor cantidad de feromonas depositaremos. Lo que si encontramos en cada uno de los algoritmos de tipo ACO es la fórmula de evaporación de feromonas. La evaporación de feromonas en los vértices impide que nuestro problema converja muy rápido hacia áreas de soluciones sub-óptimas, y favorece la exploración de nuevas áreas de soluciones.

·**Criterio de parada:** Al finalizar la actualización de las feromonas, observamos si hemos cumplido nuestro criterio de parada, si no es así volvemos a la construcción de las rutas y nuestras hormigas volverán a realizar recorridos. Una vez cumplamos el criterio de parada, dispondremos de las soluciones con menor coste/distancia y la secuencia realizada para ello.

4. ALGORITMOS BASADOS EN NUBES/ENJAMBRES DE PARTÍCULAS

Estas metaheurísticas son técnicas de búsqueda que nacieron a partir de la observación del comportamiento del vuelo de algunas bandadas de pájaros o el movimiento en las bandadas de peces. Están basados en la optimización por nubes de partículas, conocida también como Particle Swarm Optimization (PSO).

"El algoritmo se basa en una metáfora social: los individuos que son parte de una sociedad tienen una opinión influenciada por la creencia global compartida por todos los posibles individuos. Cada individuo puede modificar su opinión (o estado) según tres factores: el conocimiento del entorno, los estados por los que has pasado y los estados por los que han pasado los individuos cercanos" Gómez (2008,42).

Esta técnica simula como los enjambres de abejas buscan el polen. Disponemos de un conjunto de abejas (partículas) que van a sobrevolar el espacio (espacio de soluciones) buscando el área con mayor densidad de flores, pues aumentarán la posibilidad de encontrar una mayor cantidad de polen, recordando los lugares donde han encontrado un mayor número de flores y además conociendo cual es la mejor zona encontrada por el conjunto del enjambre.

4.1. PSO. PARTICLE SWARM OPTIMIZATION

El algoritmo PSO trabaja con una población (enjambre o nube) de posibles soluciones candidatas (partículas). Estas se mueven en busca de "comida" con la estrategia de "seguir al ave que está más cerca de la comida". Estas partículas se encuentran siempre en continuo movimiento y nunca se eliminan o mueren. Cada individuo se mueve de forma "individual" por el espacio de búsqueda, recuerdan y comunican la mejor solución que han encontrado, por lo que el movimiento de cada individuo también está guiado por aquellas partículas que han encontrado la mejor solución hasta el momento.

El primer paso en la explicación del algoritmo PSO es explicar la específica anatomía de las partículas, clave en el éxito del PSO.

Cada una de las partículas está compuesta por tres vectores:

· El primer vector es el vector de posición actual de la partícula, x_i, la cual en el inicio se calcula de forma aleatoria.

· El segundo vector contiene la posición que ha generado una mejor solución al problema encontrada por esa partícula hasta este momento, p_i.

· El tercer vector es el vector de velocidad de esa partícula donde se almacena la dirección en la cual se moverá dicha partícula, v_i.

Además, cada individuo dispone de dos valores denominados de formas muy distintas: adaptación, capacidad, aptitud, "fitness" …

· El primero de estos valores almacena el valor de adecuación de la posición actual en la que se encuentra la partícula, es decir, un valor que indica cuan de buena es la posición en la que se encuentra actualmente.

· El segundo de estos valores es el mismo para cada una de las partículas depende el momento en el que nos encontremos, pues almacena el valor de adecuación de la mejor solución encontrada por cualquiera de las partículas de nuestra población.

4.1.1. DESCRIPCIÓN DEL PROCESO ITERATIVO DEL PSO

La Figura 6 muestra el proceso iterativo del algoritmo PSO:

```
Creación enjambre de k partículas con posición y velocidades aleatorias.

Clasificar partículas con el cálculo de adec_x_i y asignamos adec_mejorpos.

Mientras no se cumpla condición de terminación

        Para cada partícula

                Actualizamos nueva posición.

                Cuando se den las condiciones actualizamos mejor posición
                personal y actualizamos mejor global.

                Actualizamos velocidad para siguiente iteración.
```

Figura 6. Pseudocódigo del algoritmo PSO. Elaboración Propia.

El primer paso que debemos hacer es crear la población que dispondrá del número de partículas que decidamos dependiendo del tipo de problema a resolver y de su complejidad, por lo que, para cada una de las partículas realizaremos los siguientes pasos:

· Asignaremos aleatoriamente la posición inicial x_i en la que se encuentre cada partícula con la condición lógica que esté dentro de nuestro espacio de búsqueda de soluciones.

· Asignaremos a p_i la misma posición que hemos hallado aleatoriamente como posición inicial x_i, debido a que la mejor posición encontrada por esa partícula corresponde a la única posición en la que se ha encontrado, la inicial.

· Calcularemos el valor de adecuación de la posición en la que se encuentra la partícula, $adec\_x_i$, y se lo asignaremos a la partícula.

· Iniciamos el valor *adec_mejorpos* que contiene el mejor valor de adecuación correspondiente a la mejor posición encontrada por alguna de las partículas, que al principio será el valor adec_x_1, pues es la primera posición creada, pero durante el proceso de creación de la población cuando aparezca una mejor posición que genere un mejor valor de adecuación, el valor *adec_mejorpos* se actualizará para todas las partículas.

· Generamos de forma aleatoria la velocidad, v_i, de cada una de las partículas.

Una vez creada la población y asignados los parámetros correspondientes a cada una de las partículas comenzarían las interacciones que tienen los siguiente pasos, siempre y cuando no se cumpla la condición de parada:

· Las partículas se desplazan desde la posición en la que se encuentran a una nueva posición del espacio de búsqueda. Al vector de posición que posee cada partícula, x_i, se le añade el vector de velocidad que también posee cada una de las partículas, v_i, para obtener una nueva posición x_i.

· Calculamos el nuevo valor de adecuación correspondiente a la nueva posición x_i, y actualizamos el valor $adec\_x_i$.

· Si el nuevo valor de adecuación $adec\_x_i$ es el mejor de los hasta ahora encontrados por la partícula *i*, actualizamos el p_i con la nueva mejor posición, y si además este valor de adecuación es el mejor valor de adecuación encontrado hasta ahora entre todas las partículas actualizamos también el valor de *adec_mejorpos* presente en todas las partículas con el nuevo valor de adecuación.

· Al final de cada iteración el vector de velocidad v_i es actualizado siguiendo una fórmula matemática en la que reside el éxito de este tipo de algoritmos, ya que regula el

comportamiento y la eficacia del modelo. Entran en esta fórmula 3 parámetros, *q, c* y *s;* los parámetros *c* y *s* controlan los componentes cognitivo y social y el parámetro *q* es un factor de inercia que depende de la iteración en la que estemos; estos tres parámetros controlan el equilibrio necesario entre la exploración del espacio de búsqueda y la explotación del algoritmo.

$$v_i^t = \mathrm{q} * v_i^{t-1} + c * rand * (p_i - x_i) + s * rand * (mejorp - x_i)$$

Es decir, la nueva velocidad para la próxima iteración se calcula como la suma del producto entre el parámetro q y la velocidad de la iteración anterior, el producto entre el parámetro c, un número aleatorio entre 0 y 1 y la resta entre la mejor posición encontrada por la partícula y la posición actual de la partícula y el producto entre el parámetro s, un número aleatorio entre 0 y 1 y la resta entre la mejor posición encontrada por cualquiera de las partículas y la posición actual de la partícula.

Como podemos observar la ecuación para calcular la velocidad posee 3 partes distintas. La primera parte de la ecuación posee el producto entre la velocidad anterior y el parámetro q, lo que significa que el algoritmo PSO posee memoria y un factor de inercia, que evita que la velocidad llegue a ser muy alta además de mejorar el rendimiento del algoritmo al ir reduciendo gradualmente su valor. La segunda parte es la parte cognitiva, pues indica la decisión que tomará la partícula dependiendo de su propia experiencia, ya que mide la distancia entre su mejor posición obtenida y en la que se encuentra ahora. La tercera parte es la parte social, pues indica la decisión que tomará la partícula en función del resto de partículas, ya que mide la distancia entre la mejor posición obtenida por el conjunto de partículas y su posición actual.

La Figura 7 representa como cada una de las partes influye en la nueva posición que tendrá la partícula i en la siguiente iteración.

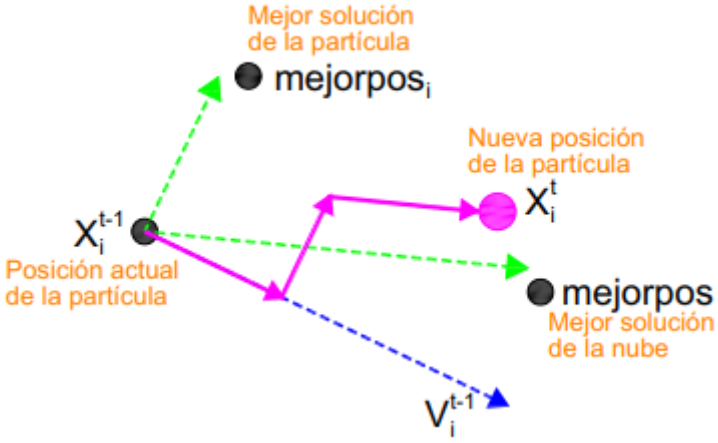

Figura 7. Esquema explicativo del movimiento de una partícula en cada iteración (Gómez, 2008).

Como hemos dicho anteriormente el valor del factor de inercia hace mejorar el algoritmo mediante el calculo de éste con la siguiente fórmula:

$$q^t = q_{max} - \frac{q_{max} - q_{min}}{t_{max}} * t$$

El parámetro q_{max} es el factor de inercia inicial, el parámetro q_{min} es el factor de inercia final, t_{max} es el número máximo de iteraciones que estamos dispuestos a realizar para no superar el esfuerzo computacional deseado y t es el número de iteración actual (Gómez, 2008) y (Aguirre, 2017).

5. SIMULATED-ANNEALING

La técnica metaheurística Simulated Annealing (Recocido Simulado) es una técnica de búsqueda local basada en el proceso que sufre un material sólido cuando es introducido a un líquido a altas temperaturas consiguiendo así su fundición y después es dejado enfriar lentamente hasta su vuelta al estado sólido.

Con la utilización de esta técnica no tratamos de buscar la mejor solución posible a partir de otra, sino que, generamos una solución inicial *s*, a partir de esta, generamos de forma aleatoria otra solución que se encuentre dentro de nuestro espacio de soluciones, s_1, y si s_1 es mejor que *s*, aceptamos a s_1 como nueva solución actual. A pesar de que s_1 sea peor que *s*, existe cierta posibilidad de que la aceptemos como nueva solución. Esta probabilidad de aceptación de s_1 como nueva solución actual cuando sea peor que *s* depende del valor objetivo de dicha solución y del valor del parámetro *t*, que como analogía al proceso en el cual se basa lo denominamos *temperatura*. Cuando llevamos pocas iteraciones, la temperatura es alta, por lo que la probabilidad de aceptación de soluciones peores también lo es, debido a que necesitamos asegurarnos de que exploramos gran parte del espacio de soluciones y que no divergimos muy pronto en un óptimo local. Cuando el proceso avanza, la temperatura disminuye, con lo que la probabilidad de aceptar soluciones peores que las que tenemos disminuye.

La idea de funcionamiento procede del proceso físico del templado de metales, en el cual para conseguir que el metal posea las propiedades deseadas de resistencia o flexibilidad, es necesario controlar la velocidad del proceso de templado, es decir, el tiempo de enfriamiento. Cuando el hierro fundido es enfriado muy despacio tiende a solidificar en una estructura de energía mínima. Con el algoritmo de Simulated Annealing imitamos este proceso de forma, que, al principio, cuando la temperatura es alta, aceptamos casi cualquier movimiento por el espacio de soluciones, es como una búsqueda aleatoria, con el fin de explorar todo el espacio de soluciones. Sin embargo, a medida que el proceso avanza y la temperatura disminuye, la exploración del espacio es sustituida por la explotación, debido a que ahora la búsqueda está guiada por las mejores soluciones, ya que la probabilidad de aceptar soluciones peores a las que tenemos es baja.

Para la realización de esta técnica hay que tener en cuenta la importancia de la elección de la probabilidad de aceptación de una solución peor a la que tenemos, del programa de enfriamiento y del criterio de parada.

En primer lugar, debemos fijar el valor inicial de la temperatura teniendo en cuenta que esta debe ser lo suficientemente alta para conseguir "fundir" todo el sistema, es decir, el peligro al fijar la temperatura inicial es ajustarla a una temperatura baja debido a que la parte de exploración de soluciones la reduciremos considerablemente, por lo que debemos asegurarnos de que es lo bastante alta.

En segundo lugar, debemos definir el programa de enfriamiento. Hay que tener en cuenta que la calidad de las soluciones obtenidas y la velocidad del sistema de enfriamiento suelen ser inversamente proporcionales. Si decidimos que la temperatura disminuya de forma muy rápida, las soluciones no serán las mejores, en cambio si disminuimos la temperatura muy lentamente, obtendremos las mejores soluciones, pero con un costo alto de tiempo de cálculo, por lo que hay que encontrar el punto de equilibrio entre ambas.

A continuación, debemos elegir la probabilidad de aceptación de una solución cuando sea peor de la que disponemos actualmente. La probabilidad de aceptación suele estar sacada, por su analogía con la termodinámica, de la distribución de Boltzmann, que depende de la diferencia de energía y temperatura y presenta la siguiente fórmula:

$$P(s \rightarrow s_1) = \exp\left(-\frac{\Delta F(s)}{T(s)}\right)$$

Donde:

$\Delta F(s)$ = F(s) - F (s_1), son los valores objetivos de las soluciones s y s_1 y T(s) es la temperatura en el momento s.

Por último, debemos fijar la condición de finalización que, dependiendo de nuestro objetivo, tiempo… podemos elegir entre muchas condiciones. Algunas que pueden ser establecidas son las siguientes:

· Cuando la solución no ha mejorado al menos en un % después de N pasos.

· Cuando se hayan realizado un número determinado de iteraciones.

· Cuando la temperatura haya alcanzado un valor mínimo (normalmente 0).

· Cuando se haya encontrado una solución aceptable.

La Figura 8 resume el procedimiento del algoritmo Simulated Annealing:

```
Generar solución inicial s dentro del espacio de soluciones V(s)

T=temperatura inicial

Mientras no se cumpla condición de terminación

        Establecer el conjunto de soluciones accesibles desde s

        Generar s_n

        Si s_n es mejor que s_{n-1}

                s_act=s_n

        Sino

                Calcular P (s_{n-1} → s_n)

                s_act=s_n con probabilidad P

        Actualizar T
```

Figura 8. Pseudocódigo del algoritmo Simulated Annealing (Maroto et al., 2012).

Para comenzar generamos de forma aleatoria o mediante algún heurístico la solución inicial de la que partimos, además de fijar la temperatura inicial del proceso. A continuación, repetiremos los siguientes pasos hasta cumplir con la condición de terminación. El primer paso de toda iteración es generar el conjunto de soluciones accesibles desde la solución actual y se elige de forma aleatoria una de ellas. Si esta solución es mejor que la actual la sustituimos y se convierte en la nueva solución actual, si no, la aceptamos con una probabilidad que tendremos que calcular. El ultimo paso es actualizar la temperatura a una temperatura menor (Maroto et al., 2012) y (Pérez de Vargas, 2015).

6. APLICACIÓN A UN PROBLEMA REAL: PROBLEMA DEL VIAJANTE

A continuación, vamos a aplicar lo explicado a un problema real, al famoso problema del viajante. Los algoritmos genéticos son especialmente adecuados para los problemas de optimización combinatoria, como es el caso del problema del viajante.

En este problema, un viajante debe visitar x lugares y lo que buscamos es la solución óptima que minimice la distancia recorrida por el viajante. Este caso se puede aplicar a la realidad a miles de problemas distintos, como a una gira de un grupo de música, a un técnico que debe visitar 10 casas distintas en la misma ciudad, al repuesto de café de las universidades…

Veamos el problema que vamos a resolver:

Un vendedor/comercial de la empresa Heura Food, debe recorrer las ciudades más importantes de Europa con el fin de enseñar sus productos a los principales supermercados, para que expongan sus productos en sus establecimientos, la empresa está experimentando un crecimiento exponencial por lo que se necesita que se consigan nuevos grandes clientes lo más rápido posible. Nuestro comerciante debe visitar las ciudades de:

Barcelona, Estambul, Londres, Moscú, San Petersburgo, Berlín, Madrid, Kiev, Roma, París, Bucarest, Minsk, Hamburgo, Viena, Varsovia y Budapest.

La sede de Heura Food se encuentra en Barcelona, por lo que debe salir de Barcelona visitar 15 ciudades distintas y volver a Barcelona.

Este problema, que a priori puede parecer sencillo de resolver, tiene $2.092279 * 10^{13}$ posibilidades distintas. Si intentáramos resolver este problema con un modelo de programación lineal con variables binarias, tendríamos un total de 256 variables binarias y 257 restricciones. Este problema no es factible resolverlo mediante técnicas exactas, debido al tiempo computacional, por lo que no queda otra opción que resolverlo con técnicas heurísticas o metaheurísticas. Por lo que nosotros vamos a resolver este problema mediante el diseño de un algoritmo genético.

En primer lugar, extraemos los datos verdaderos de las distancias vía terrestre en automóvil entre las ciudades a visitar. En la Figura 9 podemos observar nuestra matriz de distancias:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	**Matriz de distancias**	Barcelona	Estambul	Londres	Moscú	San Petersbu	Berlín	Madrid	Kiev	Roma	París	Bucarest	Minsk	Hamburgo	Viena	Varsovia	Budapest
2	Barcelona	0	2988	1505	3610	3533	1871	621	3137	1363	1036	2859	2943	1777	1780	2355	1960
3	Estambul	2988	0	3055	2154	2517	2191	3581	1294	2241	2805	637	1819	2481	1562	2176	1327
4	Londres	1505	3055	0	2887	2818	1109	1735	2429	1848	470	2564	2181	940	1489	1646	1738
5	Moscú	3610	2154	2887	0	706	1822	4106	864	3022	2888	1771	718	2101	1936	1260	1821
6	San Petersburgo	3533	2517	2818	706	0	1751	4031	1221	2951	2765	2135	791	2030	1865	1189	1878
7	Berlín	1871	2191	1109	1822	1751	0	2320	1356	1502	1050	1699	1107	289	639	574	872
8	Madrid	621	3581	1735	4106	4031	2320	0	3630	1953	1271	3342	3386	2167	2375	2852	2519
9	Kiev	3137	1294	2429	864	1221	1356	3630	0	2354	2369	915	529	1634	1336	784	1118
10	Roma	1363	2241	1848	3022	2951	1502	1953	2354	0	1435	2038	2307	1660	1097	1766	1216
11	París	1036	2805	470	2888	2765	1050	1271	2369	1435	0	2306	2173	896	1232	1591	1480
12	Bucarest	2859	637	2564	1771	2135	1699	3342	915	2038	2036	0	1336	1997	1073	1273	837
13	Minsk	2943	1819	2181	718	791	1107	3386	529	2307	2173	1336	0	1392	1226	550	1115
14	Hamburgo	1777	2481	940	2101	2030	289	2167	1634	1660	896	1997	1392	0	974	852	1156
15	Viena	1780	1562	1489	1936	1865	639	2375	1336	1097	1232	1073	1226	974	0	680	243
16	Varsovia	2355	2176	1646	1260	1189	574	2852	784	1766	1591	1273	550	852	680	0	858
17	Budapest	1960	1327	1738	1821	1878	872	2519	1118	1216	1480	837	1115	1156	243	858	0
18																	

Figura 9. Matriz de distancias en coche entre las ciudades más habitadas de Europa. Elaboración propia.

6.1. RESOLUCIÓN Y RESULTADOS

Para la resolución de este problema utilizaremos la librería *GA* dentro del software libre *RStudio*. La librería *GA*, nos permite construir el algoritmo genético que deseamos según las características del problema a resolver.

En la Figura 10 queda explicado el formato de la función *ga* con las distintas posibilidades que nos ofrece:

```
ga(type = c("binary", "real-valued", "permutation"),
  fitness, ...,
  min, max, nBits,
  population = gaControl(type)@population,
  selection = gaControl(type)@selection,
  crossover = gaControl(type)@crossover,
  mutation = gaControl(type)@mutation,
  popSize = 50, pcrossover = 0.8, pmutation = 0.1,
  elitism = max(1, round(popSize * 0.05)),
  monitor = gaMonitor, maxiter = 100, run = maxiter,
  maxfitness = -Inf, names = NULL, suggestions, seed)
```

Figura 10. Argumentos de la función ga en R (Scrucca, 2013)

En primer lugar, podemos elegir el tipo de algoritmo genético que vamos a realizar según el tipo de codificación que tendrán nuestras soluciones, se puede elegir entre una codificación del tipo binaria, cadenas de números reales o de tipo permutación, cuando la solución se encuentra en el orden de una lista de objetos o números, que es el caso de nuestro problema a resolver.

A continuación, según el tipo de algoritmo que hayamos elegido la población se generará aleatoriamente, eligiendo nosotros el tamaño de la población que deseamos.

Nos da la oportunidad de elegir el tipo de selección que vamos a realizar para formar la población que va a reproducirse según el tipo de algoritmo. Los comunes a los tres tipos de algoritmo que podemos realizar son la selección basada en el ranking lineal, ranking no lineal, tipo simulación de una ruleta y selección por competición.

También nos ofrece distintos operadores de cruce según el tipo de algoritmo genético. Debido a que es posible dar con soluciones no factibles, para el tipo permutación tenemos la posibilidad de elegir entre los siguientes operadores: operador de ciclo cruzado (CX), Operador PMX, cruce por orden (OX) y el cruce basado en la posición (PBX).

Para finalizar podemos indicar todas las probabilidades, podemos fijar la probabilidad de cruce y de mutación, también podemos aplicar elitismo, eligiendo el número de los mejores individuos que queremos asegurar que permanezcan en la nueva población y nos da la oportunidad de sugerir alguna solución que creamos buena para que sea incluida en la población inicial. Podemos elegir y fijar cualquier condición de terminación que deseemos, desde fijar el número máximo de iteraciones hasta fijar que si en un determinado número de iteraciones consecutivas no mejora nuestro mejor individuo finalicen las iteraciones.

Todos los argumentos tienen un valor prefijado por si no queremos indicar nada, menos la función de evaluación, nuestros datos del problema y el tipo de algoritmo genético que queremos construir. La población tiene un valor prefijado de 50, la probabilidad de cruce está fijada en un 80%, la de mutación en un 10%, el elitismo se aplica en un 5%, siendo el máximo 1 individuo, el máximo número de iteraciones está fijado en 100 y, según el tipo de algoritmo que seleccionemos hay un método distinto prefijado para el proceso de selección, cruce y mutación.

En el caso del tipo permutación, si no deseamos seleccionar ningún método en concreto, el algoritmo realizará la selección basada en el ranking lineal y el proceso de reproducción lo realizará con el cruce por orden (Scrucca, 2013 y 2021).

En este trabajo hemos construido un algoritmo genético tipo permutación con el proceso de selección y cruce predeterminado, con un tamaño de población de 10, un número máximo de iteraciones de 500, con la probabilidad de cruce predeterminada, 0.8, con la

probabilidad de mutación en 0.2 y seleccionamos que no nos vaya informando de los datos en cada una de las iteraciones.

El código utilizado para obtener los siguientes resultados de la aplicación de un algoritmo genético al problema se encuentra en el anexo del trabajo y a continuación vamos a explicar los aspectos más relevantes de los resultados obtenidos.

En la Figura 11 en la parte de arriba encontramos las características básicas del algoritmo aplicado y en la de abajo encontramos los resultados:

```
-- Genetic Algorithm -------------------

GA settings:
Type                  =  permutation
Population size       =  10
Number of generations =  500
Elitism               =  1
Crossover probability =  0.8
Mutation probability  =  0.2

GA results:
Iterations             = 500
Fitness function value = 7.970033e-05
Solution =
     x1 x2 x3 x4 x5 x6 x7 x8 x9 x10  ...  x15 x16
[1,]  9  1  7 10  3 13  6 15 12   5        16  14
```

Figura 11. Resumen de la aplicación de un algoritmo genético a nuestro problema. Elaboración propia.

Podemos observar que hemos realizado las 500 iteraciones fijadas de máximo.

Nuestra función de evaluación de las soluciones se encuentra invertida por lo que cuanto mayor sea el "fitness" de nuestra mejor solución, mejor solución al problema estamos consiguiendo. Nuestro valor "fitness" de la solución es $7.970033x10^{-5}$, por lo que la distancia total que vamos a recorrer con esta solución es 12547 km.

En la Figura 12 queda reflejada la solución y el recorrido a realizar, recordamos que salíamos de Barcelona y teníamos que volver de nuevo a ella por lo que el recorrido sería:

Barcelona→ Madrid→ París→ Londres→ Hamburgo→ Berlín→ Varsovia→ Minsk→ San Petersburgo→ Moscú→ Kiev→ Estambul→ Bucarest→ Budapest→ Viena→ Roma→ Barcelona

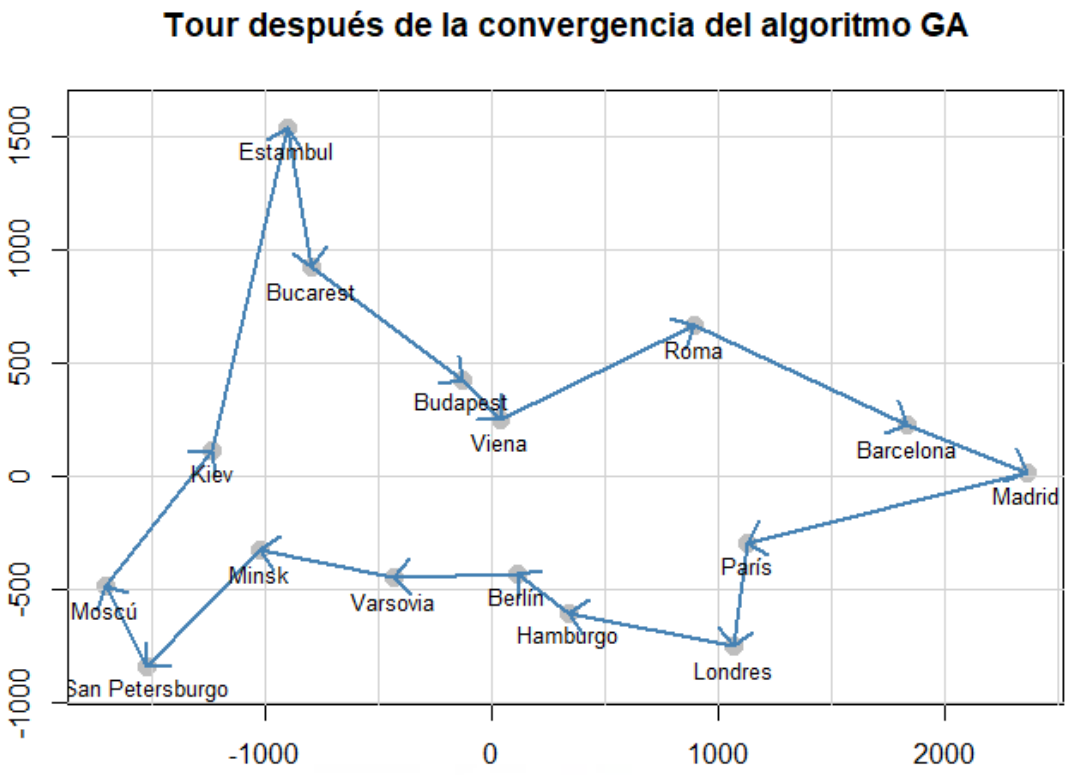

Figura 12. Gráfico de la solución obtenida al aplicar el algoritmo genético. Elaboración propia.

Observamos la colocación de las ciudades que están repartidas por el eje cartesiano en función de las distancias que hay entre unas y otras, donde podemos observar que Viena se encuentra muy cerca del "centro" de Europa si contáramos únicamente con las ciudades más pobladas.

En la Figura 13 podemos observar como ha ido mejorando la mejor solución en cada una de las iteraciones:

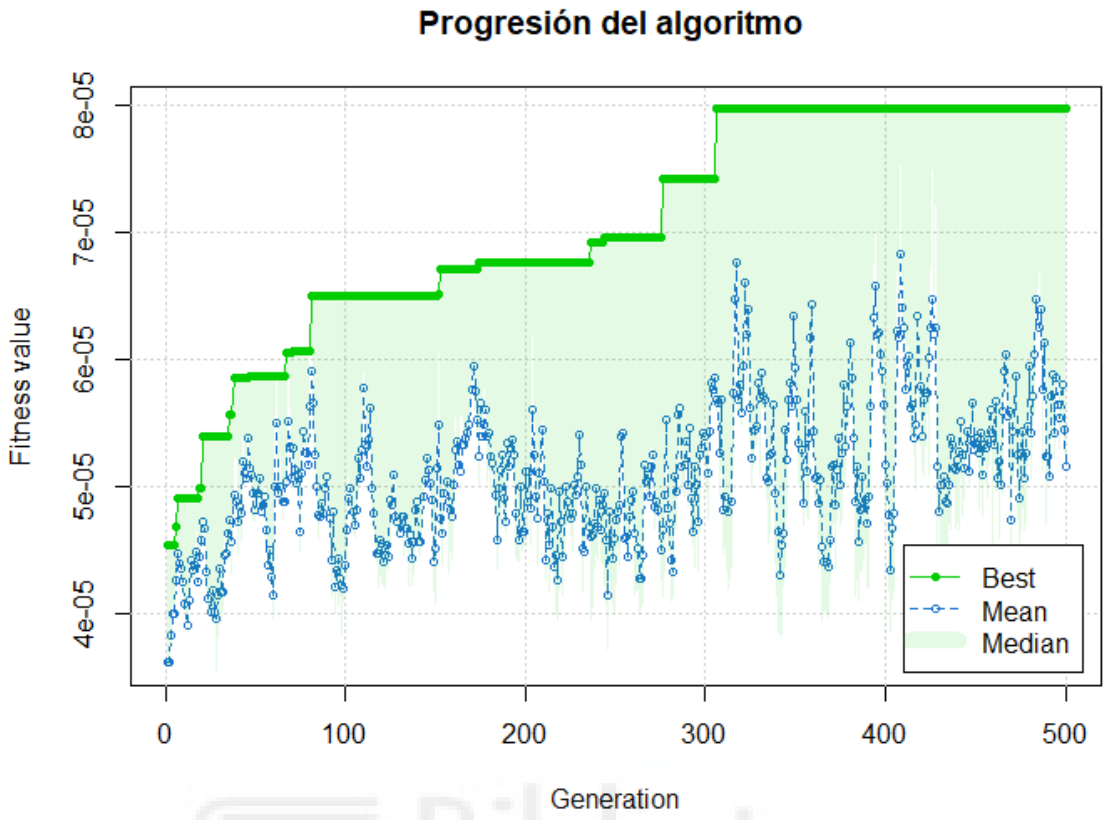

Figura 13. Gráfico que muestra la evolución de la mejor solución a través de las iteraciones. Elaboración propia.

Observamos como un poco más tarde de superar la iteración 300, no volvemos a encontrar una mejor solución.

En la Figura 14 podemos observar el resultado de realizar 100 simulaciones de la aplicación del mismo algoritmo genético realizado anteriormente:

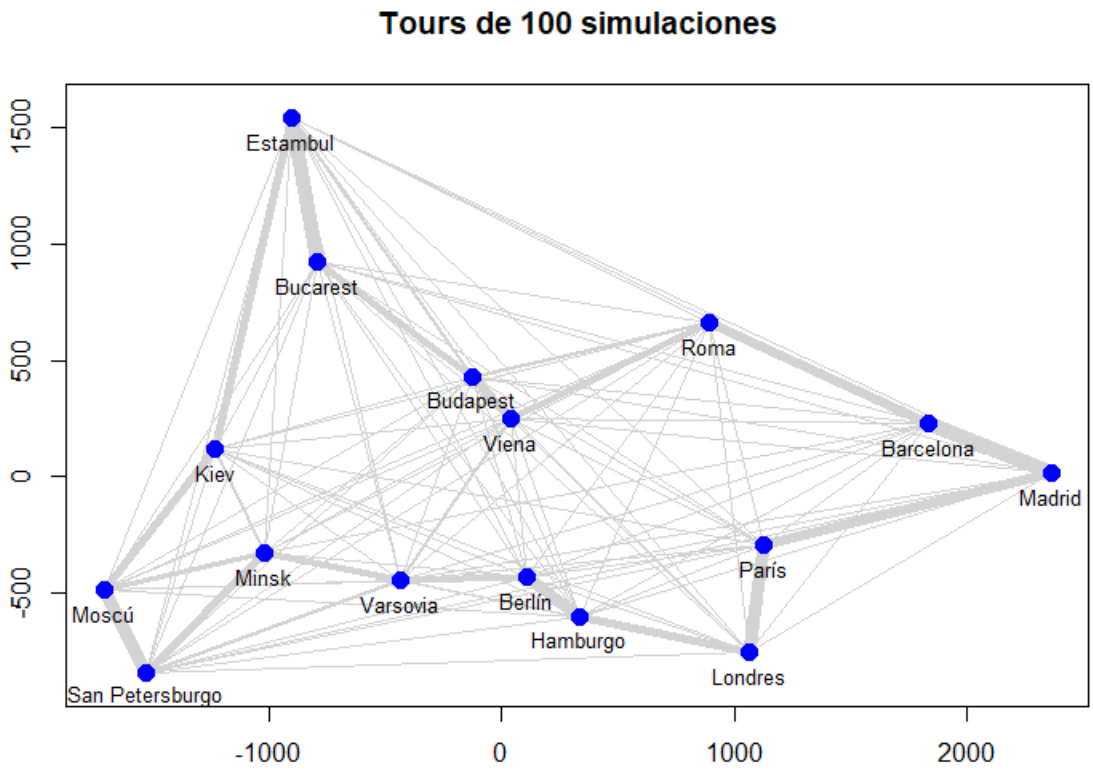

Figura 14. Gráfico de los tours resultantes de 100 simulaciones con el mismo algoritmo genético. Elaboración propia.

Podemos observar que las líneas grises más gruesas, debido a que se solapan las soluciones, nos marcan la misma solución extraída anteriormente, la cual parece que es la mejor solución posible.

Hemos probado distintos algoritmos genéticos al problema y no hemos conseguido mejorar el "fitness" de la mejor solución, pero si hemos conseguido cambiando los métodos de selección y cruce, conseguir un mejor rendimiento del algoritmo pues conseguimos los mismos resultados con menos iteraciones, esto se puede deber a que el algoritmo creado ahora no explora tanto como el otro y va construyendo una población muy buena desde el principio.

Aplicamos el mismo algoritmo de antes, disminuyendo el número de iteraciones y realizando la selección por competición y la reproducción mediante el cruce PMX

En la Figura 15 en la parte de arriba encontramos las características básicas del algoritmo aplicado y en la de abajo encontramos los resultados:

```
-- Genetic Algorithm -------------------

GA settings:
Type                  =  permutation
Population size       =  10
Number of generations =  200
Elitism               =  1
Crossover probability =  0.8
Mutation probability  =  0.2

GA results:
Iterations             = 200
Fitness function value = 7.970033e-05
Solution =
     x1 x2 x3 x4 x5 x6 x7 x8 x9 x10  ...  x15 x16
[1,]  3 10  7  1  9 14 16 11  2   8         6  13
```

Figura 15. Resumen de la aplicación de un algoritmo genético modificado a nuestro problema. Elaboración propia.

Observamos que hemos realizado 200 iteraciones y que hemos llegado al mismo valor "fitness" que en el algoritmo genético anterior.

En la Figura 16 podemos observar que la evolución de este algoritmo genético es mucho más rápida que en el anterior:

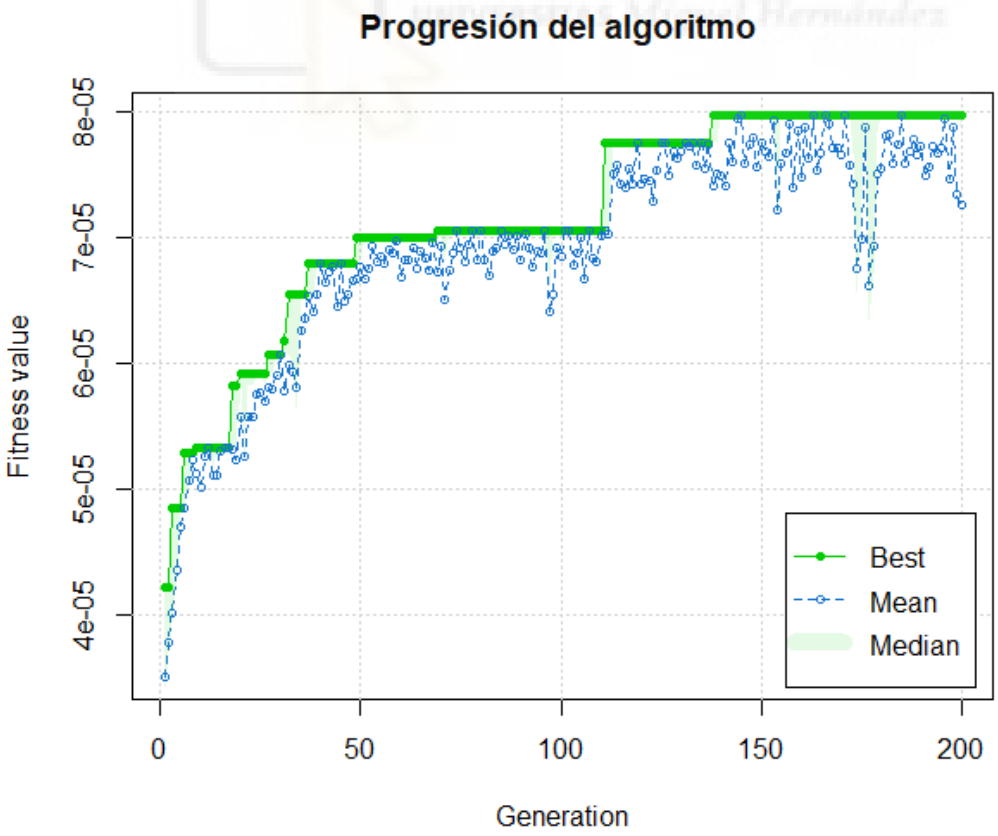

Figura 16. Gráfico que muestra la evolución de la mejor solución a través de las iteraciones del segundo algoritmo aplicado. Elaboración propia.

Podemos observar que en comparación con el gráfico de la Figura 13, en la Figura 16 la media se encuentra bastante constante y cercana al "fitness" de la mejor solución, mientras que la media en la Figura 13 fluctúa mucho entre iteraciones, debido a que estaba explorando por si en algún "rincón" del espacio de soluciones se encontraba la mejor, en cambio, este algoritmo ha ido directamente a la mejor solución por lo que la ha encontrado antes. En este caso no ha sido un problema debido a que hemos llegado a la misma solución, pero en otros tipos de problema el perder varianza desde el principio puede ser un error debido a que podemos estancarnos en óptimos locales no globales.

7. CONCLUSIONES

En nuestro trabajo hemos tratado de plasmar la importancia de la metaheurística en el panorama actual y futuro, consiguiendo encontrar soluciones a problemas imposibles de resolver antes de su aparición, y sus ventajas frente a la heurística. Hemos explicado las diferentes técnicas metaheurísticas que existen en función de la forma en que encuentran/construyen las soluciones a los problemas, centrándonos en aquellas que basan su funcionamiento en comportamientos observados en la naturaleza. Entre todas ellas destaca por su mayor utilización y la gran cantidad de alternativas que presenta, los algoritmos genéticos. Nos hemos centrado en esta técnica metaheurística, explicando su funcionamiento básico y la importancia de cada una de las partes de su procedimiento, así como las alternativas que presentan cada una de ellas para que se ciñan al problema a resolver para asegurarnos de que su efectividad sea máxima, donde si falla una de ellas el algoritmo no será efectivo y no se acercará a obtener un valor óptimo cercano al óptimo global que mediante estas técnicas no podemos asegurar.

Hemos explicado el funcionamiento básico y dónde reside el éxito de las metaheurísticas inspiradas en la naturaleza más utilizadas después de los algoritmos genéticos, aquellas inspiradas en la colonias de hormigas, inspiradas en nubes de partículas e inspiradas en el proceso físico que experimenta un metal cuando es introducido en un líquido a altas temperaturas.

Finalmente hemos plasmado en un ejemplo como aplicar un algoritmo genético para resolver el famoso problema del viajante TSP, dándonos cuenta de que aplicar un algoritmo genético simple con éxito es relativamente sencillo, ya que hay softwares como es el caso de *RStudio*, que han implementado librerías para que su aplicación sea viable para personas con no mucha formación en sus programas, dando además un sinfín de alternativas para resolver una gran cantidad de problemas. Hemos conseguido resolver el problema propuesto, donde nuestro algoritmo genético creado, resuelve el problema en 1 segundo, es decir, sin requerir de un esfuerzo ni computacional ni de tiempo.

El futuro de las metaheurísticas es impresionante e ilusionante, la gran diversidad que presentan y su flexibilidad permiten combinarlas con otras muchas técnicas pudiendo ser aplicadas a cualquier problema que requiera mejorar el esfuerzo de computación y/o la calidad, permitiendo simplificar casi cualquier problema. Actualmente se están implementando metaheurísticas híbridas junto al Deep Learning, Inteligencia Artificial… Me parece una rama que no tiene techo en su unión con otras debido a su diversidad, por lo que la mejora de las metaheurísticas significarán una mejora en otras muchas ramas y permitirán la resolución de problemas muy complejos no resolubles hasta el momento, de forma parecida a lo que ocurre con las matemáticas y las demás ciencias básicas, cualquier descubrimiento ayuda a avanzar al resto.

8. BIBLIOGRAFÍA

· Aguirre, J. (2017). Algoritmo cultural y de nubes de partículas multi-objetivo para evitar colisiones en la gestión de tráfico aéreo. Universidad Politécnica de Madrid.

· Al-Shayea, A., Saleh, M., Alatefi, M. y Ghaleb, M. (2020). Scheduling Two Identical Parallel Machines Subjected to Release Times, Delivery Times and Unavailability Constraints. *Processes 8, no. 9*.

· Darwin, C. (1921). El origen de las especies por medio de la selección natural.

· Gómez, M. (2008). Sistema de generación eléctrica con pila de combustible de óxido sólido alimentado con residuos forestales y su optimización mediante algoritmos basados en nubes de partículas. Universidad Nacional de Educación a Distancia.

· Hussain A., Muhammad Y.S., Nauman Sajid, M., Hussain I., Mohamd Shoukry A. y Gani S. (2017). Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator. Computational Intelligence and Neuroscience.

· karthy1988 (2015). Optimized Delivery Route using Genetic Algorithm: Cost cutting for e-commerce. rpubs.com. https://rpubs.com/karthy1988/TSP_GA

·Maroto, C., Alcaraz, J., Ginestar, C. y Segura M. (2012). Investigación Operativa en Administración y Dirección de Empresas. Valencia: Editorial Universitat Politècnica.

· Pérez de Vargas, B. (2015). Resolución del Problema del Viajante de Comercio (TSP) y su variante con Ventanas de Tiempo (TSPTW) usando métodos heurísticos de búsqueda local. Universidad de Valladolid

· Revuelta, T. (2015). Desarrollo y aplicación de un algoritmo ACO para el problema ATSP. Universidad de Valladolid.

· Scrucca, L. (2013). GA: A PACKAGE FOR GENETIC ALGORITHMS IN R. Università degli Studi di Perugia.

· Scrucca, L. (2021). Genetic Algorithms. cran.r-project.org. https://cran.r-project.org/web/packages/GA/GA.pdf

· Soft Computing and Intelligent Information Systems (2006). Algoritmos Genéticos I. Conceptos Básicos. Bioinformática. Universidad de Granada.

https://sci2s.ugr.es/sites/default/files/files/Teaching/GraduatesCourses/Bioinformatica/Tema%2006%20-%20AGs%20I.pdf.

·somasdhavala (2014). Genetic Algorithm on TSP. rpubs.com. https://rpubs.com/somasdhavala/GAeg

·Talbi, E.-G. (2009). Metaheuristics: From Design to Implementation. Wiley Publishing.

· Vargas, J. y Penit, V. (2016). Estudio y aplicación de metaheurísticas y comparación con métodos exhaustivos. Universidad Complutense de Madrid.

·Vázquez, J. (2016). Aplicación del algoritmo de colonia de hormigas al problema de rutas de reparto con destino móviles. Escuela técnica superior de ingeniería. Universidad de Sevilla.

·Westreicher, G. (24 de mayo de 2020). Optimización. Economipedia.com.

https://economipedia.com/definiciones/optimizacion.html

9. ANEXO

Código utilizado para aplicar un algoritmo genético a nuestro problema y graficar los resultados para conseguir una visualización más clara de estos, extraído de dos artículos de la plataforma Rpubs (karthy1988,2015) y (somasdhavala, 2014).

```
#Solo vamos a necesitar 3 librerías de R, las instalamos y las activamos.

library(readxl)

install.packages('Rcpp')

library(Rcpp)

install.packages("GA")

library("GA")

# Leemos el fichero donde tenemos los datos de las distancias entre las ciudades
seleccionadas.

Matrizdedistancias <- read_excel ("Matriz de distancias.xlsx”, range = "B2:Q17",
col_names = FALSE)

M<-as.matrix(Matrizdedistancias)

colnames(M)=c("Barcelona", "Estambul", "Londres", "Moscú", "San Petersburgo",
"Berlín", "Madrid", "Kiev", "Roma", "París", "Bucarest", "Minsk", "Hamburgo",
"Viena", "Varsovia", "Budapest")

rownames(M)=c("Barcelona","Estambul","Londres","Moscú","San Petersburgo",
"Berlín", "Madrid", "Kiev", "Roma", "París", "Bucarest", "Minsk", "Hamburgo",
"Viena", "Varsovia", "Budapest")

names<-c("Barcelona","Estambul","Londres","Moscú","San Petersburgo", "Berlín",
"Madrid", "Kiev", "Roma", "París", "Bucarest", "Minsk", "Hamburgo", "Viena",
"Varsovia", "Budapest")

set.seed(123)

#Realizamos un tour para calcular la distancia total

tourLength <- function(tour, distMatrix) {
```

```
        tour <- c(tour, tour[1])
        route <- embed(tour, 2)[, 2:1]
        sum(distMatrix[route])}
# La inversa de la distancia total es nuestra función de evaluación o función fitness
tpsFitness <- function(tour, ...) 1/tourLength(tour, ...)
#Utilizaremos el comando GA para resolver el problema.
```

#Al ser un problema donde el resultado es el orden de las ciudades, elegimos el tipo permutación, indicado para problemas que contengan reordenamiento de una lista de objetos. Le especificamos el tamaño de la población, la probabilidad de mutación y le pedimos que no nos muestre los resultados de cada iteración mediante el comando monitor="null", además de fijar el máximo de iteraciones en 500.

```
GA.fit <- ga(type = "permutation", fitness = tpsFitness, distMatrix = M, lower = 1,
        upper = 16, popSize = 10, maxiter = 500, run = 500, pmutation = 0.2,
        monitor = NULL)
#Observamos la resultados
summary(GA.fit)

#Lo graficamos para ver la solución de forma más clara.
mds <- cmdscale(Matrizdedistancias)
x <- mds[, 1]
y <- mds[, 2]
plot(x, y, type = "n", asp = 1, xlab = "", ylab = "", main = "Tour después de la
convergencia del algoritmo GA")
abline(h = pretty(range(x), 10), v = pretty(range(x), 10), col = "light gray")
points(x, y, pch = 16, cex = 1.5, col = "grey")
```

```
tour <- GA.fit@solution[1, ]

tour <- c(tour, tour[1])

n <- length(tour)

arrows(x[tour[-n]], y[tour[-n]], x[tour[-1]], y[tour[-1]], length = 0.15, angle = 25, col =
"steelblue", lwd = 2)

text(x, y, names, cex=0.8)

#Graficamos la evolución del algoritmo genético, para ver cómo ha ido mejorando
nuestra solución a través de las iteraciones.

plot(GA.fit, main = "Progresión del algoritmo ")

#Vamos a crear dos funciones que nos van a permitir graficar muchas simulaciones del
mismo algoritmo genético.

getAdj <- function(tour) {

	n <- length(tour)

	from <- tour[1:(n - 1)]

	to <- tour[2:n]

	m <- n - 1

	A <- matrix(0, m, m)

	A[cbind(from, to)] <- 1

	A <- A + t(A)

	return(A)

}

plot.tour <- function(x, y, A) {

	n <- nrow(A)

	for (ii in seq(2, n)) {

		for (jj in seq(1, ii)) {
```

```
                    w <- A[ii, jj]
                    if (w > 0)
                        lines(x[c(ii, jj)], y[c(ii, jj)], lwd = w, col = "lightgray")
    }
  }
}
n <- length(x)
B <- 100
fitnessMat <- matrix(0, B, 2)
A <- matrix(0, n, n)
#Generamos un bucle para que aplique 100 veces el mismo algoritmo genético aplicado anteriormente para comparar las distintas soluciones que nos van saliendo.
for (b in seq(1, B)) {
        GA.rep <- ga(type = "permutation", fitness = tpsFitness, distMatrix = M,
          lower= 1, upper=16, popSize = 10, maxiter = 500, run = 500,
          pmutation = 0.2, monitor = NULL)

        tour <- GA.rep@solution[1, ]
         tour <- c(tour, tour[1])
        A <- A + getAdj(tour)
}
#Graficamos los caminos obtenidos en las soluciones al aplicar el algoritmo 100 veces.
plot(x, y, type = "n", asp = 1, xlab = "", ylab = "", main = "Tours de 100 simulaciones")
plot.tour(x, y, A * 10/max(A))
points(x, y, pch = 16, cex = 1.5, col = "blue")
```

```
text(x, y - 100, names, cex = 0.8)
```

#Probamos otro algoritmo genético cambiando los métodos de selección y de cruce, además de reducir el número máximo de iteraciones.

```
GA.fit <- ga(type = "permutation", fitness = tpsFitness, distMatrix = M, lower = 1, upper = 16, selection =gaperm_tourSelection , crossover =gaperm_pmxCrossover, popSize = 10, maxiter = 200, run = 200, pmutation = 0.2,monitor = NULL)

summary(GA.fit)

plot(GA.fit, main = "Progresión del algoritmo ")
```