Trabajo Fin de Grado
en Ingeniería de las Tecnologías Industriales

Análisis y resolución del Problema del Viajante aplicando el Algoritmo Genético en Python

Autor: Guillermo Moreno García

Tutor: José Miguel León Blanco

Dpto. de Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2025

Trabajo Fin de Grado
en Ingeniería de las Tecnologías Industriales

Análisis y resolución del Problema del Viajante aplicando el Algoritmo Genético en Python

Autor:
Guillermo Moreno García

Tutor:
José Miguel León Blanco
Profesor Contratado Doctor

Dpto. de Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2025

Trabajo Fin de Grado: Análisis y resolución del Problema del Viajante aplicando el Algoritmo Genético en Python

Autor: Guillermo Moreno García

Tutor: José Miguel León Blanco

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2025

El Secretario del Tribunal

A mi familia

A mis maestros

A mis amigos

Agradecimientos

Quiero agradecer a mi familia su apoyo constante durante todo este proceso. Principalmente me gustaría destacar la insistencia de mi padre sin el cual este trabajo posiblemente no se habría llevado a cabo. Por soportar largas tardes de estudio y dedicación durante muchas semanas. También, me gustaría dar las gracias a José Miguel por su dedicación y su amabilidad siempre que la he necesitado. Por último, agradecer a mis amigos por su compañía y motivación en los momentos clave.

Guillermo Moreno García

Sevilla, 2025

Resumen

En el mundo actual, donde la logística forma una parte fundamental de la mayoría de los sectores industriales y empresariales, el conseguir optimizar rutas y minimizar los costes de transporte se vuelve un desafío de vital importancia. El Problema del Viajante sirve de base para tratar todas estas cuestiones.

Este Trabajo de Fin de Grado, en adelante TFG, se centra en la resolución de un *Travelling Salesman Problem*, mediante la aplicación de un algoritmo genético aprovechando la flexibilidad que nos ofrece a la hora de programar el lenguaje Python.

El principal objetivo es analizar este problema y su facilidad para escalarlo a un sinfín de situaciones actuales. Donde se va a comparar y parametrizar las diferentes opciones que existen dentro de este.

Abstract

In today's world, where logistics is a fundamental part of most industrial and business sectors, optimising routes and minimising transport costs is a vital challenge. The traveller problem serves as a basis for addressing all these issues.

This Final Degree Project, hereinafter TFG, focuses on the resolution of a Travelling Salesman Problem, through the application of a genetic algorithm taking advantage of the flexibility offered by the Python language when programming.

The main objective is to analyse this problem and how easy it is to scale it to a wide range of current situations. Where we are going to compare and parameterise the different options that exist within this.

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice **xiv**

Índice de Tablas **xviii**

Índice de Figuras **xviii**

Notación **xx**

1 Introducción **1**

2 Análisis teórico **3**

2.1 Dificultad del TSP *3*

2.2 Aplicaciones del TSP *4*

2.3 Formulación Matemática *4*

2.4 Variantes del problema *6*

3 Métodos de resolución **9**

3.1 Algoritmos exactos *9*

3.1.1 Algoritmo de Fuerza Bruta 9

3.1.2 Algoritmo Branch and Bound 9

3.2 Algoritmos heurísticos *10*

3.2.1 Nearest Neighbor 10

3.2.2 Nearest Insertion 10

3.2.3 K-Optimal Algorithm 10

3.3 Algoritmos metaheurísticos *11*

3.3.1 Búsqueda Local 11

3.3.2 Algoritmos Bioinspirados 13

4 Algoritmos seleccionados **17**

4.1 Algoritmo Genético *17*

4.2 Cheapest Insertion *19*

5 Programación del algoritmo **22**

5.1 Fitness *23*

5.2 Población inicial *23*

5.3 Selección y reproducción *23*
5.4 Cruce *24*
5.5 Mutación *24*
5.6 Ejecución *24*
5.7 Parametrización *25*

6 Resultados obtenidos y conclusión **27**
6.1 Resolución óptima con nuestro algoritmo *27*
6.2 Análisis comparativo *29*
6.2.1 Parámetros de selección 29
6.2.2 Tasa de mutación 32
6.2.3 Conclusiones 36

Referencias **38**

Anexo **43**

ÍNDICE DE TABLAS

Tabla 1. Coordenadas de la instancia R101.50. 26
Tabla 2. Parámetros analizados. 35

ÍNDICE DE FIGURAS

Figura 1. Formación de subcircuitos. 5
Figura 2. Ejemplo de mejora del resultado con el intercambio de aristas. 11
Figura 3. Salida por pantalla de las variables del problema. 27
Figura 4. Representación de la mejor ruta calculada del TSP. 28
Figura 5. Impresión por pantalla de la ruta y el valor final. 28
Figura 6. Representación de la solución para *TamTorneo=2* 28
Figura 7. Representación de la solución para *TamTorneo=8*. 29
Figura 8. Representación de la solución para *TamTorneo=4.* 29
Figura 9. Representación de la solución para *Suavizacion = 0.1*. 30
Figura 10. Representación de la solución para *Suavizacion = 1.* 30
Figura 11. Representación de la solución para *Suavizacion = 10*. 30
Figura 12. Representación de la solución para *Suavizacion = 0.1* con una población de 1000 individuos 31
Figura 13. Representación de la solución para *TasaMutacionInicial = 0,02*. 31
Figura 14. Representación de la solución para *TasaMutacionInicial = 0,25.* 32
Figura 15. Representación de la solución para *TasaMutacionInicial = 0,50.* 32
Figura 16. Representación de la evolución de la tasa de mutación con la adaptación lineal. 33
Figura 17. Representación de la evolución de la tasa de mutación con la adaptación cuadrática. 33
Figura 18. Representación de la evolución de la tasa de mutación con la adaptación logarítmica. 33
Figura 19. Representación de la solución con adaptación lineal de la tasa de mutación. 34
Figura 20. Representación de la solución con adaptación cuadrática de la tasa de mutación. 34
Figura 21. Representación de la solución con adaptación cuadrática de la tasa de mutación. 34

Notación

TFG	Trabajo de Fin de Grado
TSP	Traveling Salesman Problem
TSPm	Traveling Salesman Problem con múltiples visitas
GTSP	Traveling Salesman Problem generalizado
mTSP	Traveling Salesman Problem con múltiples viajeros
TSPTW	Traveling Salesman Problem con ventanas temporales
NN	Nearest Neighbor
SS	Scatter Search
PSO	Particle Swarm Optimization
ACO	Ant Colony Optimization

1 INTRODUCCIÓN

El Problema del Viajante es uno de los problemas computacionales más complejos del mundo actual. Puesto que, está relacionado con uno de los siete problemas del milenio cuya resolución está premiada con un millón de dólares estadounidenses por el *Clay Mathematics Institute* [1]. Entraremos en mayor detalle en el capítulo 2.1.

Este problema, que será nombrado como Problema del Viajante, Travelling Salesman Problem o TSP de forma indistinta a lo largo del documento, consiste en encontrar la ruta más corta y, al mismo tiempo, la más eficiente, para llegar a un destino.

Es un problema aparentemente sencillo, puesto que es una labor que realizamos todos a diario al desplazarnos, el buscar el camino más corto hacia nuestro destino. Pero que, según profundizamos en él, nos daremos cuenta de la complejidad de este.

La estructura del documento será la siguiente:

- *Capítulo 1: Introducción.* Breve descripción del objeto y alcance del trabajo.
- *Capítulo 2: Análisis Teórico.* Se detallan las principales características de los problemas TSP y sus variantes. Mostrando su formulación matemática.
- *Capítulo 3: Métodos de resolución.* Explicación de los diferentes métodos de resolución del problema. Exactos, heurísticos y metaheurísticos.
- *Capítulo 4: Algoritmos Seleccionados.* Se determinan los algoritmos empleados para resolver nuestro problema en concreto.
- *Capítulo 5: Programación del algoritmo.* Análisis de cómo se ha programado y resuelto nuestro problema de forma computacional.
- *Capítulo 6: Resultados obtenidos y conclusión.* Se expone el estudio y las conclusiones sobre la modificación de los parámetros del algoritmo y los resultados obtenidos.

2 ANÁLISIS TEÓRICO

En este capítulo, se aborda la evolución histórica y el desarrollo teórico del *Travelling Salesman Problem* [2], siendo este uno de los problemas más representativos del campo de la optimización y la matemática computacional. El TSP surge de la necesidad de optimizar el transporte y la distribución de mercancías al producirse la Revolución Industrial del siglo XIX y XX, la cual propició una apertura de mercados aumentando los desplazamientos tanto regionalmente como internacionalmente.

Las primeras menciones al problema datan de un libro alemán de 1832 dirigido a vendedores, el cual incluía ejemplos de rutas comerciales por Alemania y Suiza. Sin embargo, este escrito no poseía un enfoque matemático. Fueron los matemáticos William Rowan Hamilton y Thomas Kirkman los que establecieron los primeros fundamentos teóricos en 1857, con su famoso juego: "The Icosian Game" [3].

Este juego consiste en recorrer todos los vértices de un grafo sin repetir ninguno volviendo al punto inicial, siendo esto la búsqueda de un ciclo hamiltoniano, matemáticamente hablando.

Pese a estos jóvenes comienzos, no sería hasta la década del 1930 cuando el matemático austriaco Karl Menger comenzó su investigación con problemas similares a lo que hoy entendemos como el Problema del Viajante. A estos problemas los denominó como el *Problema del Mensajero*. Aplicaba el algoritmo de la *Fuerza Bruta* que calcula la distancia mínima para conectar un conjunto de puntos probando todas las combinaciones posibles. Técnica que rápidamente mostró grandes limitaciones al aplicarla en escalas mayores.

Años más tarde, en la década de los años 40, Merrill M. Flood estudió la optimización de la ruta de autobuses escolares, siendo este uno de los primeros ejemplos de aplicación del TSP. Aunque el concepto de *Travelling Salesman Problem* lo introdujo Hassler Whitney catedrático de la Universidad de Princeton.

Desde entonces, el estudio del TSP se ha convertido en un modelo fundamental para el desarrollo de métodos y algoritmos de optimización aplicables a numerosos problemas de la industria moderna.

A lo largo de este capítulo, exploraremos el Problema del Viajante en profundidad, analizando sus principales características, su evolución histórica y algunas de sus aplicaciones más relevantes.

2.1 Dificultad del TSP

Tras conocer los orígenes del problema, vamos a analizar la complejidad de este [4]. El TSP pertenece a la clase de problemas NP-Hard. Para entenderlo mejor, primero debemos indagar en los Problemas P y NP.

Los problemas de clase P son problemas que se pueden resolver en un tiempo polinómico por una máquina determinista. Teniendo en cuenta que un algoritmo tiene un tiempo polinómico, si su tiempo de ejecución puede definirse con una función polinómica del tamaño de la entrada. Es decir, este tipo de problemas no son muy complejos y se resuelven "rápidamente" en términos computacionales.

Los problemas NP a diferencia de los primeros, no tienen por qué ser resueltos en un tiempo polinómico. Pero conociendo su solución, esta sí puede ser verificada en un tiempo polinomial.

El TSP se encuentra dentro de los problemas NP-Hard, lo que significa que es al menos tan complejo como cualquier problema en NP, aunque no necesariamente pertenece a esta clase. Ya que, puede que su solución no sea verificable en un tiempo polinómico.

Esto implica que, aunque para verificar si una potencial solución del Problema del Viajero es la óptima puede resultar sencillo para casos muy pequeños, no existe ningún algoritmo eficiente conocido para comprobar todos los casos. Esta situación se debe al crecimiento exponencial de las posibles soluciones a medida que se aumenta el número de nodos, puesto que la cantidad de rutas posibles aumenta factorialmente.

Por lo que, cuando tenemos un TSP con grandes instancias la resolución exacta se vuelve

computacionalmente imposible. Sin embargo, los avances en el desarrollo de microprocesadores más potentes han permitido mejorar el desempeño de los algoritmos que aproximan soluciones. Surgiendo métodos heurísticos y metaheurísticos que permiten obtener soluciones aproximadas en un tiempo razonable. Nos adentraremos en ello más adelante.

2.2 Aplicaciones del TSP

Pese a su enfoque inicial en rutas comerciales, el Problema del Viajante tiene una gran variedad de aplicaciones en distintos campos en la actualidad. En este punto se presentan algunos de los más destacados:

Uno de los principales campos con aplicaciones más directas y numerosas del TSP es la *logística*, donde prácticamente la totalidad de las empresas de paquetería o de servicios de entrega de comida a domicilio emplean algoritmos modelados como un TSP para organizar los envíos y ahorrar tiempo y costes en los transportes. También, es ampliamente empleado en empresas de transportes escolares y laborables. Incluso algunas agencias de viajes y agentes comerciales optimizan los puntos que deben de visitar aplicando algún algoritmo de resolución del TSP. [5]

En el ámbito de la *industria*, aunque en menor medida que en el caso anterior, el TSP contribuye a la reducción de costos en diversas tareas. Destaca principalmente su empleo en la secuenciación de tareas, con la aplicación de algoritmos heurísticos en cadenas de montaje y entornos de fabricación. También, tiene un papel relevante en la programación de algoritmos que guían un láser a través de los puntos que deben ser cortados, empleándose en la creación de microprocesadores y de placas de circuito.

Por último, el TSP se ha aplicado en la *organización de datos* en grupo (*clusters*). En este contexto, ayuda a organizar elementos similares mediante una adaptación del problema. Buscando maximizar la similitud entre los datos, facilitando la agrupación y un análisis estructurado.

2.3 Formulación Matemática

Existen diversas formulaciones para el problema del TSP, se han propuesto hasta ocho versiones diferentes [6]. A continuación, se expondrá una de las más comunes de este.

El Problema del Viajante se puede resumir como un problema de optimización lineal donde se busca recorrer todos los nodos existentes una única vez minimizando la suma total de las distancias entre un nodo y el siguiente del camino, así se define la función objetivo.

También, se aplican una serie de restricciones. Concretamente dos para impedir que se repita el paso por un nodo ya visitado, una para regular la entrada y otra para la salida. Y con el propósito de evitar la formación de bucles para asegurar el paso por todos los nodos del sistema tenemos tres opciones que veremos próximamente.

El conjunto que forman la función objetivo y las restricciones constituyen la base del TSP, antes de proceder con su representación matemática, se deben definir algunos conceptos clave que se emplearan posteriormente:

- Un *grafo* $\mathrm{G} = (\mathrm{N}, \mathrm{A})$ consiste en un conjunto N de elementos denominados vértices o nodos, y un conjunto A cuyos elementos se denominan arcos o aristas. Cada arco representa una conexión directa entre dos nodos del conjunto N.
- Un *grafo orientado o dirigido* es aquel en el que los arcos son pares ordenados; es decir, un arco (i, j) parte del nodo i y llega al nodo j.
- Un *grafo no orientado o no dirigido* es un grafo en el cual (i, j) y (j, i) representan el mismo arco.
- Un *grafo completo* es un grafo (ya sea dirigido o no) en el que todas las aristas posibles están presentes.
- Un *camino* es una secuencia de aristas en la que cada vértice es único y no se repite.

- Un *circuito* es una secuencia de aristas donde el primer y el último vértice son el mismo, sin que existan vértices repetidos entre ellos.

Con estas definiciones en mente, el problema del TSP se puede formular en términos de teoría de grafos de la siguiente forma: sea $G = (N, A)$ un grafo completo, donde $N = \{1, 2, \dots, n\}$ representa el conjunto de nodos o vértices, y A es el conjunto de arcos. Los nodos $i = 2, \dots, n$ corresponden a los clientes que se deben visitar, mientras que el nodo 1 se considera tanto el punto de inicio como de finalización (ciudad de origen y destino) el cual podríamos considerar como almacén. Cada arco (i, j) tiene asociado un valor positivo d_{ij}, que indica la distancia entre los vértices i y j.

Definiendo las variables binarias de decisión x_{ij} para cada par $(i, j) \in$ A, donde estas toman el valor de 1 si el arco (i, j) forma parte de la solución y 0 en caso contrario, el problema consiste en minimizar la siguiente función objetivo:

$$\min \sum_{i}^{n} \sum_{j}^{n} d_{ij}\, x_{ij}$$

Sujeto a las siguientes restricciones, que aseguran que únicamente se pase una vez por cada nodo:

1. Restricción de entrada única: Para cada nodo j, solo se puede llegar desde un nodo i:

$$\sum_{i} x_{ij} = 1 \ \ \forall j$$

2. Restricción de salida única: Para cada nodo i, solo puede salir hacia un nodo j:

$$\sum_{j} x_{ij} = 1 \ \ \forall i$$

Estas restricciones aunque son necesarias, no son suficientes para delimitar el problema, puesto que se pueden dar lugar a subcircuitos, como se puede observar en la Figura 1. Donde a pesar de cumplir las dos restricciones vistas no se recorren todos los nodos. Véase que $x_{16} = x_{65} = x_{51} = x_{24} = x_{43} = x_{32} = 1$, por lo que no se viola ninguna de las restricciones.

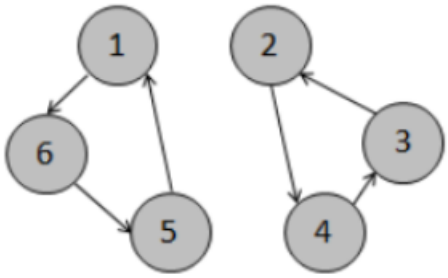

Figura 1. Formación de subcircuitos.

Para que el modelo nos ofrezca una solución viable, es necesario incorporar una restricción adicional que elimine los subcircuitos. Para ello, tenemos tres alternativas:

- *Restricciones de eliminación de subcircuitos*: Este enfoque consiste en añadir una condición que garantice que al menos un arco salga de cualquier subcircuito. Esto se expresa como:

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 1 \quad \forall S \subset N, S \neq \emptyset, S \neq N$$

Esta restricción asegura que cualquier subconjunto de nodos *S* será abandonado al menos una vez. El principal inconveniente es que se requiere un gran número de restricciones. Si $|N| = n$, se necesitan alrededor de $2^n - 2$ restricciones de este tipo. En total serían $2^n + 2^n - 2$

restricciones: $2^n - 2$ para romper subcircuitos, n para la restricción de entrada y n para la de salida de cada nodo.

En la práctica, este problema se suele resolver con procedimientos de generación de restricciones. Con la idea de incluir únicamente las restricciones que se necesiten y no todas al inicio.

- *Restricciones de Tucker, Miller y Zemlin* [7]: Otro método para evitar la formación de subcircuitos es mediante la introducción de nuevas variables de decisión al problema. En este caso, se definen las variables u_i, $\forall i \in N$, que representan la posición en la secuencia en que se visita cada nodo i. Para el nodo de origen, se fija el valor de u_0 en 1, ya que es el primer nodo en ser visitado. Luego, se añaden $n - 1$ variables auxiliares continuas, u_1, que tomarán valores entre 2 y n, además de las siguientes $(n-1)^2$ restricciones, que aseguran la no formación de subcircuitos:

$$C - u_j + n\, x_{\{ij\}} \leq n - 1$$
$$2 \leq i \neq j \leq n$$

- *Problema auxiliar de redes*: Esta última estrategia consiste en utilizar, además de las variables binarias propias del problema, algunas variables de flujo y formular un problema de flujo en redes con la siguiente estructura:

$$\min \sum_i^n \sum_j^n d_{ij}\, x_{ij}$$
$$\sum_i x_{ij} = 1 \ \forall j$$
$$\sum_{j=1} x_{ij} = 1 \ \forall i$$
$$x_{ij} \in \{0,1\} \forall i$$
$$\sum_j y_{ij} - \sum_j y_{ji} = b_i \qquad i = 1, \dots, n$$
$$0 \leq y_{ij} \leq (n-1) x_{ij}$$

En el primer nodo, asignamos la oferta $n - 1$, es decir, $b_i = n - 1$ para los demás nodos asignamos una demanda de 1, es decir, $b_1 = -1$ para $i = 2, \dots, n$. De esta forma, garantizamos que todos los nodos estarán interconectados, ya que las unidades de flujo deben llegar a cada nodo específico. Además, debido a las restricciones de asignación, cada nodo tendrá un solo arco de salida y otro de entrada, lo cual asegura que la solución sea de en un único circuito.

2.4 Variantes del problema

Existen multitud de variantes del *Travelling Salesman Problem* propiciado por su gran adaptabilidad a una amplia serie de situaciones y problemas específicos los cuales han ido surgiendo con el desarrollo de nuevas necesidades. Algunas de las más relevantes se han recopilado en un artículo Abraham Punnen y Gregory Gutin [8], de las cuales se procede a destacar las siguientes:

- *MAX TSP*: En este caso y siguiendo la idea contraria al TSP original, el objetivo es encontrar un circuito hamiltoniano cuyo coste sea el máximo posible. [9]

- *TSP con cuello de botella*: Aquí se busca encontrar la solución en la cual se minimice el mayor coste de una arista. Es decir que arista de mayor tamaño del sistema sea del menor tamaño posible. [10]
- *TSPm, TSP con múltiples* visitas: Similar al TSP convencional, pero sin la restricción de que cada nodo solo sea visitado una vez. [11]
- *TSP agrupado*: Los nodos o ciudades se agrupan en conjuntos, y el objetivo es encontrar un circuito de forma que las ciudades dentro de un mismo grupo sean visitadas consecutivamente. [12]
- *GTSP, TSP generalizado*: Al igual que en el caso anterior, los nodos se dividen en grupos, pero en esta ocasión el objetivo es encontrar un circuito hamiltoniano de coste mínimo que visite únicamente un nodo de cada conjunto. [13]
- *mTSP, TSP con múltiples viajeros*: En esta variante, se tiene más de un viajante lo cual permite repartir la visita de los nodos, esto permite reducir el tiempo necesario para completar la visita de todos los nodos. [14]
- *TSPTW con ventanas de tiempo*: En este problema, cada nodo tiene un tiempo establecido de visita, donde se marca un instante de llegada y un tiempo de salida, por lo que solo se puede pasar por esa ciudad en el rango temporal establecido. [15]

3 MÉTODOS DE RESOLUCIÓN

Desde el comienzo de la investigación sobre el Problema del Viajante, se han desarrollado varios métodos para obtener soluciones de forma más rápida y precisa posible. Comenzando desde el método de la fuerza bruta utilizado por Karl Menger, como vimos en el capítulo *Análisis teórico,* hasta la aplicación en la actualidad de complejos algoritmos metaheurísticos.

Para entenderlos mejor, procedemos a analizar en esta sección los principales métodos de resolución que se han empleado y se emplean para resolver el TSP.

3.1 Algoritmos exactos

Estos algoritmos fueron los primeros en ser empleados. Tienen como objetivo encontrar la solución óptima al problema, es decir, las rutas en las que todos los nodos sean visitados una sola vez. En el caso que nos ocupa, como se ha mencionado previamente, al tratarse de un problema de complejidad NP-Hard, su implementación se ve restringida en cierto modo. Sin embargo, dentro de estos métodos, algunos son más eficientes y tienen un mayor alcance que otros.

3.1.1 Algoritmo de Fuerza Bruta

Probablemente, el más sencillo y robusto de los algoritmos que se analizarán, debido a su facilidad de aplicación y a que en un principio nos ofrecen soluciones exactas. El método evalúa todas las posibles combinaciones y selecciona la mejor solución por comparación directa. Su principal debilidad es que su rango de cálculo se limita a un pequeño número de nodos, ya que el número de combinaciones aumenta a un ritmo de $\frac{(N-1)!}{2}$. Cuando el número de nodos incrementa, el uso de este algoritmo se vuelve inviable debido a su crecimiento factorial [16].

Dado que la mayoría de las situaciones en las que se aplica el TSP involucran una mayor cantidad de nodos de la que puede manejar el algoritmo de fuerza bruta, este enfoque tiene una utilidad práctica limitada y, salvo casos concretos, no es muy aplicado en la actualidad.

3.1.2 Algoritmo Branch and Bound

El algoritmo de *Branch and Bound* [17], o de *Ramificación y Acotación*, tiene su origen en los trabajos de Dantzig, Fulkerson y Johnson realizados entre 1954 y 1959. Esta es la técnica más común para resolver los problemas de programación entera, donde las variables de decisión deben ser valores enteros.

Este enfoque busca resolver problemas complejos dividiendo el problema en partes más pequeñas y facilitando su evaluación, la fase de ramificación, y una vez que acotamos la solución óptima podemos comparar y descartar aquellos subconjuntos que no ofrecen una mejora, esta es la conocida como fase de acotación.

Más tarde, en los años 80, se desarrolló una mejora conocida como *Branch and Cut* [18], o Ramificación y Corte en español. Esta técnica incorpora el uso de planos de corte, lo que facilita el excluir las soluciones no viables, al mejorar las aproximaciones.

Esto redujo el tiempo computacional a la hora de emplear el algoritmo. Aun así, debido a las dimensiones con las que se suele trabajar en el TSP, este método se vuelve prácticamente inoperante, puesto que el crecimiento exponencial del conjunto de soluciones supera la capacidad de computación del método. Por ello, se desarrollaron alternativas que ofrecen mejores resultados.

3.2 Algoritmos heurísticos

Con el objetivo de reducir el coste computacional, que es el principal factor limitante en los algoritmos exactos, surgen las heurísticas. Las cuales, a pesar de no garantizar hallar la solución óptima, sí que ofrecen una alternativa aceptable con un mucho menor tiempo computacional

En esta sección vamos a analizar algunas de las heurísticas más conocidas y empleadas actualmente. Cabe destacar que la heurística *Cheapest Insertion* no se encuentra en esta selección, puesto que será detallada en el capítulo *Nuestros Algoritmos,* ya que forma parte del programa diseñado para resolver el Problema del Viajante en este TFG.

3.2.1 Nearest Neighbor

Una de las heurísticas más simples empleadas para resolver el TSP es la denominada *heurística del vecino más próximo,* o como es mayoritariamente conocida en su versión en inglés *Nearest Neighbor*, NN [19]. Este algoritmo y el *Nearest Insertion*, que se expondrá a continuación, son heurísticas constructivas. Es decir, que generalmente se emplean como búsqueda para una solución inicial del problema para posteriormente aplicar una heurística de mejora.

El NN propone que el "viajante" inicie su recorrido en una ciudad y, a partir de ahí, se dirija a la ciudad más cercana que aún no haya visitado. Este proceso se repite hasta que se hayan recorrido todas las ciudades, momento en el cual se regresa al punto de partida.

Un aspecto a tener en cuenta es que, al construir la solución, el algoritmo suele comenzar de manera eficiente seleccionando aristas de menor coste o distancia. Sin embargo, al final del proceso, pueden quedar ciudades cuya conexión requiera aristas de mayor distancia. Este fenómeno, que se conoce como "miopía" del algoritmo, surge porque en cada paso se elige la mejor opción inmediata sin considerar cómo podría afectar las decisiones futuras. La cuál es su mayor desventaja.

Para implementar este algoritmo, puede seguirse el esquema:

1. Seleccionar un nodo inicial de forma arbitraria, denotado como j.
2. Establecer $l = j$ y definir el conjunto $T = \{1, 2, \ldots, n\} \setminus \{j\}$
3. Mientras $T \neq 0$:
    - Identificar $j \in T$ tal que $d_{lj} = min\{d_{ij} | i \; e \; T\}$
    - Conectar l con j.
    - Actualizar $T = T \setminus \{j\}$ y establecer $l = j$.
4. Conectar el nodo l con el nodo inicial para completar la ruta.

3.2.2 Nearest Insertion

Este método sigue la misma lógica del algoritmo *Nearest Neighbor*. Se comienza con un ciclo inicial compuesto por unos pocos nodos. El algoritmo busca insertar el nodo que genere el menor incremento en el coste total del recorrido. Es decir, identifica la ciudad más próxima a cualquiera de las ciudades que ya forman parte del ciclo y la incorpora. Este proceso de inserción se repite de manera iterativa, ampliando gradualmente el ciclo inicial hasta que se incluye a todas las ciudades en el recorrido final [20].

Ambas heurísticas vistas pertenecen al grupo de *Greeady Algorithm* [21] que engloba algoritmos como estos dos presentados donde se busca partiendo de un nodo inicial, insertar el siguiente que posea menor distancia o coste sin repetir ninguno en la secuencia.

3.2.3 K-Optimal Algorithm

Este algoritmo pertenece al grupo de las heurísticas de mejora, esto quiere decir que necesita de una

solución inicial del problema para a partir de ella optimizarla.

Su funcionamiento se basa en la sustitución de *k* arcos de un recorrido por otros diferentes. El problema se considera k-óptimo cuando no es posible obtener una ruta con menor coste o distancia mediante el intercambio de *k* arcos. Pese a los avances en la capacidad computacional, este análisis solo se suele realizar para el caso de 2-opt, es decir, k igual a 2. [22] Puesto que, para *k* mayores el tiempo de cálculo se dispara.

La heurística 2-opt se centra en analizar las intersecciones entre aristas y evaluar si la sustitución de estas por otras dos aristas que no se crucen produce una mejora del resultado final.

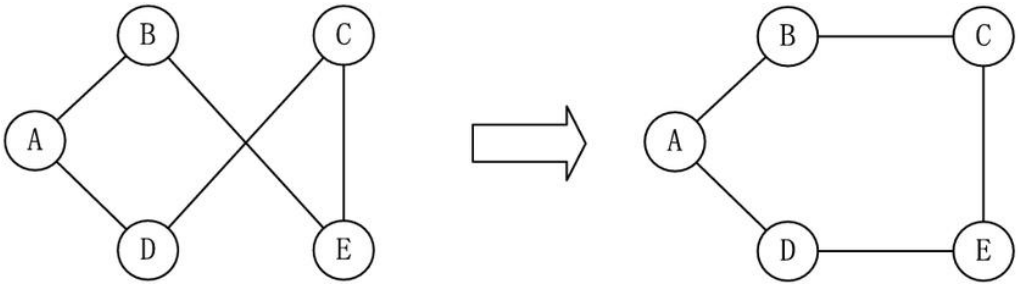

Figura 2. Ejemplo de mejora del resultado con el intercambio de aristas

3.3 Algoritmos metaheurísticos

Tras haber analizado las heurísticas más conocidas, nos adentramos en un siguiente nivel dentro de la complejidad de estos algoritmos, las metaheurísticas. Estas surgen como una evolución de las heurísticas que exploran de manera más eficiente espacios de búsqueda complejos, siendo más adaptables y dinámicas.

Las metaheurísticas son una de las técnicas más estudiadas y empleadas en la actualidad, gracias a su facilidad para ser adaptadas a una gran variedad de situaciones y problemas en la práctica. Como ya vimos con las heurísticas, estos modelos, pese a no conseguir la solución óptima, sí que nos ofrecen buenos resultados, mejores incluso que los obtenidos con las heurísticas, con un bajo coste computacional.

Por lo general, siguen algoritmos probabilísticos, lo que restringe la optimalidad. Se ejecutan de forma iterativa y se suelen complementar con otros algoritmos de optimización para acotar, acelerar el proceso y evitar quedar estancados en óptimos locales.

Estas técnicas de optimización se pueden dividir en dos grandes subgrupos que son los de *Búsqueda Local* [23] y los *Bioinspirados* [24]. Comenzamos a analizar el primero de ellos.

3.3.1 Búsqueda Local

Estos métodos se emplean cuando el tamaño del espacio de soluciones es de gran tamaño y debido al tiempo que se tardaría no se puede realizar una exploración que lo recorra completamente para hallar la solución óptima. Para reducir el espacio de búsqueda se emplean diferentes algoritmos que descartan rutas que no mejoren la solución actual.

El principal inconveniente que presentan estos algoritmos es que se pueden desechar caminos que conduzcan a buenas soluciones, o que se produzca un estancamiento en diversos escenarios: en máximos locales, donde no existen vecinos con mejores valores; en mesetas, donde los vecinos tienen el mismo coste sin progreso claro; o en crestas u hombros, donde los vecinos combinan opciones de costes peores y mejores, dificultando el avance.

3.3.1.1 Hill climbing

Esta metaheurística también conocida como *Escalada*, es el primero de los algoritmos de *búsqueda local* que vamos a analizar en este TFG [25].

El algoritmo de *Escalada* es un método de mejora iterativa donde, partiendo de un estado inicial, en

cada iteración, se analizan las soluciones cercanas al punto de partida y tras evaluarlas se escoge la que mejor resultado ofrezca. La evaluación se realiza con la aplicación de una función heurística a definir previamente.

Existen dos variantes principales. La *escalada simple*, su enfoque consiste en aplicar a la solución actual un operador que la mejore. En cuanto se identifica un estado válido que mejora al actual se utiliza, descartando el anterior. Por otro lado, la *escalada por máxima pendiente* analiza todas las soluciones posibles en cada iteración y selecciona la que ofrece una mejora mayor. A diferencia de la versión simple que para al encontrar la primera solución con mejora, este método evalúa todas las opciones antes de decidir.

La principal ventaja del algoritmo es que reduce el uso de memoria, ya que no guarda un registro del proceso de selección y descarte de vecinos. Sin embargo, además de las limitaciones generales de los algoritmos de *búsqueda local*, no ofrece un buen desempeño cuando se emplea en problemas TSP con restricciones adicionales. Puesto que no posee mecanismos internos que garanticen el cumplimiento de todas las restricciones durante la ejecución de la metaheurística. Esta limitación, aunque sucede en los demás algoritmos, es especialmente acentuada en el *Hill Climbing* debido a su enfoque ascendente y su falta de elementos probabilísticos que añadan aleatoriedad al sistema.

Para llevar a cabo este algoritmo es fundamental establecer ciertos aspectos clave. En primer lugar, es necesario crear una representación de los diferentes estados que contemple todas las posibles soluciones del problema en cuestión para luego definir una función objetivo que evalúe la calidad de cada solución propuesta; también, se hace imprescindible contar desde el inicio una solución inicial que puede ser generada de manera aleatoria o a través de heurísticas; por último, resultará esencial disponer de un generador de soluciones cercanas al estado actual que permitan explorar diversas alternativas, incluyendo cierta dosis de aleatoriedad, para ampliar las posibilidades de búsqueda.

Si bien la *Escalada* es una estrategia conveniente y efectiva en la resolución de problemas complejos, dependerá en gran medida de la heurística utilizada para determinar su velocidad y eficiencia.

3.3.1.2 Simulated Annealing

El *Simulated Annealing* [26], o enfriamiento simulado, pertenece también al grupo de los algoritmos de *búsqueda local*, el cual se inspira en el proceso físico del recocido, donde un material se calienta a altas temperaturas y se enfría de forma controlada para mejorar sus propiedades físicas. Siendo este, un enfoque muy adaptable al ámbito de la optimización como una técnica que explora soluciones de manera iterativa y controlada, con el objetivo de aproximarse al mejor resultado posible.

Este método se comienza con una solución inicial y un valor alto de temperatura. A partir de ahí, se generan nuevas soluciones modificando ligeramente la actual. Estas nuevas soluciones se evalúan, y si ofrece una mejora, se selecciona como solución; en caso contrario, se puede aceptar según una probabilidad que disminuye conforme avanza el proceso, controlada por la temperatura. Este mecanismo facilita al algoritmo escapar de óptimos locales.

La temperatura, que es el elemento central de este enfoque, se reduce gradualmente siguiendo un esquema de enfriamiento. En las primeras etapas, cuando la temperatura es alta, se permite mayor exploración ya que se pueden producir cambios que no mejoren la solución actual. A medida que disminuye, el algoritmo se enfoca en las zonas con un mejor desempeño dentro del espacio de búsqueda. Este proceso se detiene cuando se alcanza el criterio de parada, que puede ser un número máximo de iteraciones o una temperatura mínima.

3.3.1.3 Búsqueda Tabú

La *Búsqueda Tabú* [27] es una metaheurística que destaca por su capacidad para escapar de óptimos locales, gracias al empleo de una memoria. Esta característica la diferencia del resto de algoritmos ya vistos hasta ahora, puesto que almacena en una lista "tabú" las soluciones potencialmente buenas, eliminándolas temporalmente del espacio de búsqueda, lo que permite al algoritmo evitar bucles y diversificar.

Al igual que las demás metaheurísticas de esta sección, se comienza con una solución inicial y se generan nuevas soluciones al evaluar los nodos próximos a esta. Se selecciona la mejor solución entre ellas, siempre y cuando esta alternativa no esté en la lista tabú, aunque se pueden aceptar soluciones tabú prometedoras según reglas de *aspiración*. Tras cada iteración, la solución escogida se agrega a la lista tabú,

mientras que las más antiguas se eliminan, para mantener el tamaño de la lista constante.

Es de gran relevancia en el problema el diseñar adecuadamente parámetros como el tamaño de la lista, las condiciones de aspiración y los criterios de parada para lograr un buen desempeño del algoritmo. También, es común encontrar, junto al algoritmo, métodos que diversifican las regiones a explorar, cuando el progreso se estanca, trabajando conjuntamente.

3.3.1.4 Búsqueda Dispersa

El algoritmo de Búsqueda Dispersa [28] (Scatter Search, SS) no es una metaheurística que encaje en el grupo de *búsqueda local* al cien por cien, a pesar de ello sí que comparte características como el uso de operadores para transformar soluciones y el empleo de memoria para almacenar y combinar resultados, similar a la *Búsqueda Tabú*, buscando mejorar soluciones existentes.

La primera descripción del método fue publicada en 1977 por Fred Glover donde se establecen las bases del algoritmo. Que se creó para abordar problemas de optimización complejos mediante la combinación sistemática de soluciones. A diferencia de otros métodos que dependen de la aleatoriedad, el SS utiliza estrategias que aprovechan tanto la diversidad como la calidad de las soluciones. Veamos cómo funciona.

Inicialmente, se genera un conjunto de soluciones variadas mediante técnicas de diversificación controlada. Es decir, las soluciones generadas no se centran en una única región del espacio exclusivamente, asegurándonos así de que se da cierta diversidad en el grupo y garantizando un estudio de la gran mayoría del espacio de búsqueda. Tras esto, se analizan aplicando heurísticas o métodos exactos.

Las soluciones con mejor desempeño son almacenadas en un conjunto de referencia (RefSet), que se actualiza de manera constante. Este conjunto actúa como base para la creación de nuevas soluciones, que surgen de la combinación de las alternativas del RefSet tratando de aprovechar las mejores soluciones.

3.3.2 Algoritmos Bioinspirados

Las primeras metaheurísticas que hemos analizado hasta ahora se incluyen dentro de los algoritmos de *Búsqueda Local*. En adelante, vamos a indagar en los *algoritmos Bioinspirados* que como su propio nombre indica se basan en comportamientos y patrones que podemos encontrar en la naturaleza.

La metaheurística del *Algoritmo Genético*, que se encuadra dentro de este subgrupo de metaheurísticas, será detallada en el siguiente capítulo al igual que la heurística *Cheapest Insertion*.

3.3.2.1 Particle Swarm Optimization

El *Particle Swarm Optimization* [29], abreviado como PSO, es una metaheurística que se inspira en el comportamiento colectivo de sistemas naturales, como bandadas de aves o bancos de peces entre otros.

Este método emplea un grupo de partículas que representan posibles soluciones. Estas se van desplazando por el espacio de búsqueda, siendo influencias tanto por su experiencia personal como por la del conjunto de partículas. El factor determinante es procurar el equilibrio entre la exploración de nuevas áreas y la explotación de las zonas que ofrecen mejores resultados.

Cada partícula tiene una posición y velocidad que se actualizan combinando tres factores: su movimiento previo; la atracción hacia su mejor posición conocida, basada en su experiencia hasta ese instante; y la atracción hacia la mejor posición global determinada por el grupo. Como en la mayoría de los algoritmos analizados, este proceso iterativo se continúa hasta alcanzar un criterio de parada.

Si aplicamos el PSO a nuestro *Travelling Salesman Problem*, podemos adaptar las partículas para

representar rutas como permutaciones. Cada partícula equivale a un recorrido, y su calidad se mide por la distancia total del mismo. Las rutas se ajustan utilizando técnicas como el intercambio en la posición de la ruta de las ciudades basándonos en las actualizaciones de velocidad.

3.3.2.2 Colonia de hormigas

El algoritmo de optimización por *colonia de hormigas* [30], también conocido en inglés como *Ant Colony Optimization* o simplemente ACO, es una metaheurística basada en el comportamiento colectivo de las hormigas en la vida real. Reflejando el proceso de búsqueda que realizan al buscar fuentes de alimento. Este método se centra en la generación de soluciones a través de la influencia de las feromonas. Donde cada hormiga, con su propio camino, construye una solución individual guiándose de las feromonas que van depositando otras hormigas a su paso y con la influencia de la función heurística que se establezca.

En este contexto, una hormiga se define como un individuo que construye su propia solución de forma autónoma. Según avanza interactúa con el entorno detectando y depositando feromonas en los caminos que considera prometedores.

El algoritmo comienza con un conjunto de hormigas explorando el espacio de búsqueda. A medida que se avanzan, seleccionan hacia donde se dirigen basándose en dos factores: la cantidad de feromonas que encuentran en el camino, siendo mayor su influencia según crece la cantidad de estas; y la evaluación de las soluciones que ofrece el método de resolución escogido, por lo general una heurística. Las hormigas depositan feromonas en las soluciones que generan, y estas feromonas se evaporan gradualmente con el tiempo para evitar la convergencia prematura en soluciones subóptimas.

El proceso se repite iterativamente, permitiendo que las hormigas refinen colectivamente sus soluciones a medida que la información de las mejores rutas se acumula gracias a los rastros de feromonas. Algunos de los parámetros clave son la tasa de evaporación de las feromonas y el peso relativo de la evaluación heurística.

Este método es uno de los más empleados a la hora de resolver problemas de tipo TSP, ya que gracias a su gran balance entre la exploración de la región de búsqueda y su explotación de las mejores soluciones permite obtener resultados cercanos al optimo en problemas complejos y de grandes dimensiones.

4 ALGORITMOS SELECCIONADOS

Tras haber indagado en los principales métodos de resolución que se han utilizado y varios se siguen empleando a día de hoy. Como ya se comentó en el anterior capítulo, en este se explicarán los algoritmos del *Algoritmo Genético* y el *Cheapest Insertion* que son los empleados para resolver el Problema del Viajante que analizaremos en el quinto capítulo.

4.1 Algoritmo Genético

Para resolver el Problema del Viajante se ha seleccionado la metaheurística del *Algoritmo Genético* [31][32], puesto que es uno de los algoritmos más estudiados y aplicados en la actualidad, con usos como la optimización de producción multicriterio, la optimización de redes logísticas (caso que se asemeja bastante a nuestro problema) o el diseño automatizado de componentes y equipamiento industrial entre otros.

A finales de los 50 y principios de los años 60 aparecen los primeros ejemplos de lo que hoy podemos llamar algoritmos genéticos. Nils Aall Barricelli fue uno de los pioneros, simulando la evolución por ordenador en 1954, y junto a Alex Fraser, que en 1957, comenzó con la investigación de la selección natural. En los orígenes de este algoritmo también destaca Hans-Joachim Bremermann que contribuyó en los 60 con técnicas de recombinación, mutación y selección en problemas de optimización. [33]

Aunque nombres ilustres como Ingo Rechenberg, Hans-Paul y Lawrence J. Fogel contribuyeron con el desarrollo de estrategias y de la programación evolutiva, hay que destacar el papel de John Henry Holland. Quien con su libro *Adaptation in Natural and Artificial Systems* en 1975, fue el primero en presentar sistemática y rigurosamente el concepto de sistemas digitales adaptativos utilizando las operaciones de mutación, selección y el cruzamiento, simulando el proceso de la evolución biológica como estrategia para resolver problemas.

Tras haber visto los orígenes históricos del *Algoritmo Genético*, vamos a indagar un poco más sobre qué es y cuáles son sus principales características. Podemos definirlo como una metodología para resolver problemas que toma como modelo los principios de la evolución biológica. Esta técnica, clasificada dentro de los métodos basados en poblaciones, trabaja con un conjunto de soluciones posibles como entrada para un problema dado. Estas soluciones se representan mediante una codificación específica y se evalúan a través de una función del fitness, que mide de manera cuantitativa la calidad de cada solución candidata.

De forma general, el algoritmo puede describirse en los siguientes pasos:

1. La *Inicialización*; donde se crea una población (que es un conjunto de individuos o soluciones potenciales), ya sea de manera aleatoria o siguiendo algún criterio. En nuestro caso, la población inicial se conseguirá aplicando la heurística *Cheapest Insertion* que explicaremos en el apartado *4.2*.

2. La *Evaluación*; donde se determina el Fitness o la aptitud de cada individuo de la población. El fitness lo podemos definir como la clasificación de la idoneidad de cada individuo para resolver el problema.

3. La *Selección*; en la que se eligen los individuos que serán los padres para la siguiente generación. Es decir, los individuos que serán la base desde la que se cree la siguiente población.

4. El *Cruce (Crossover)*; con el objetivo de generar nuevas soluciones a partir de dos individuos (padres) se generen otros dos (hijos) con la intención de que estos ofrezcan una mejor solución que los padres, creando así una nueva generación.

5. La *Mutación*; se introducen cambios de forma aleatoria en algunos individuos de la población para añadir diversidad genética evitando caer en mínimos locales.

6. La *Parada*; se establece un criterio de parada para finalizar el algoritmo, en nuestro caso será el número de generaciones.

La selección de los individuos que deben utilizarse para formar la siguiente generación se puede realizar de diversas formas, mezclando la influencia de la evaluación de los individuos con cierto grado de aleatoriedad para permitir explorar un mayor número de alternativas evitando así que el algoritmo se estanque en un mínimo local.

Cabe destacar la posibilidad de emplear varias de las técnicas de selección, que vamos a ver a continuación, en un mismo algoritmo, tratando de aprovechar las ventajas de varios métodos y sus sinergias. Algunas de las más comunes son:

- La *selección por rueda de ruleta*; se eligen los individuos para la próxima generación asignándoles probabilidades proporcionales a su fitness. Si imaginamos una ruleta dividida en sectores donde los más aptos ocupan porciones más grandes. Se simula un giro generando un número aleatorio, y el individuo correspondiente es seleccionado. Este método balancea la selección favoreciendo a los mejores individuos, aunque permite que los menos aptos también tengan una oportunidad, promoviendo diversidad genética.
- La *selección por torneo*; los individuos se dividen en subgrupos seleccionados al azar, y compiten entre sí. Solo se elige a un individuo de cada subgrupo, y el más apto pasa a formar parte de la reproducción, también se puede añadir un carácter probabilístico en la competencia.
- La *selección por rango;* se organiza a los individuos de la población en un ranking basado en su fitness en lugar de usar los valores absolutos de aptitud. Una vez clasificados, la probabilidad de selección de cada individuo se asigna de acuerdo con su posición en el ranking, no directamente por su fitness.
- La *selección escalada*; para aplicar esta técnica se ajusta dinámicamente la presión selectiva para amplificar las diferencias entre individuos a medida que la aptitud promedio de la población aumenta. Transforma la función de fitness original, usando métodos como escalado lineal, sigma o logarítmico, para hacer más discriminatorias las pequeñas diferencias en aptitud entre los mejores individuos. Esto es útil en etapas avanzadas del algoritmo, evitando que la selección se vuelva ineficaz cuando las diferencias de fitness son pequeñas, y favoreciendo una evolución más eficiente sin sacrificar diversidad.
- La *selección elitista*; donde los mejores individuos de la población, según el fitness, pasan directamente a la siguiente generación, lo que ofrece una estrategia distinta a la evolución genética. Por lo general, se establece un porcentaje de la población total como tamaño de elite para asegurar el buen desempeño obtenido hasta el momento. Es de los métodos más empleados, aunque por lo general se utiliza en conjunto con otro de los mencionados.

Una vez hemos seleccionado los individuos que ejercerán de padres para la siguiente generación, debemos mezclarlos para crear los hijos. Este proceso que como hemos visto denominamos *cruce* podemos llevarlo a cabo de tres diferentes formas:

- El *cruce de un punto*; se selecciona un punto aleatorio en el genoma de los padres y se intercambian los genes en esa posición para crear dos descendientes. Los genes antes del punto provienen de un padre y los genes después del punto provienen del otro.

- El *cruce en dos puntos*; de forma muy similar al criterio anterior, se seleccionan dos puntos aleatorios en las secuencias de los padres, y los genes entre esos puntos se intercambian para generar

descendientes. En el contexto del Problema del Viajante, esto implica seleccionar dos puntos en las rutas de las listas de ciudades visitadas y cambiar el orden de las ciudades en ese segmento.

- El *cruce uniforme*; donde cada gen, o ciudad en nuestro caso, de la descendencia se selecciona al azar de uno de los dos padres, por lo que cada elemento del individuo tiene un 50% de posibilidades de provenir de uno de sus progenitores. En este enfoque, no se trabaja con segmentos contiguos, sino que la herencia de cada posición se decide individualmente.

El *Algoritmo Genético* es una herramienta que ha demostrado ser muy eficaz, por lo que es ampliamente empleada para la resolución de problemas complejos, especialmente aquellos que involucran espacios de búsqueda grandes y no lineales.

Una de sus principales fortalezas es la capacidad de encontrar soluciones cercanas al óptimo global; gracias a su exploración diversificada del espacio de soluciones, lo cual los hace menos propensos a quedar atrapados en mínimos locales. Además de que, son altamente adaptables por lo que se pueden aplicar fácilmente en una amplia variedad de problemas. Gracias a la naturaleza paralela del algoritmo se nos permite su implementación en equipos con hardware avanzado, mejorando significativamente el tiempo de ejecución en problemas de alta complejidad.

A pesar de sus ventajas, este algoritmo también presenta ciertas limitaciones. Una de las más notables es su dependencia en la parametrización adecuada (como tamaño de población, tasa de mutación, y número de generaciones), la cual puede requerir experimentación y ajuste manual considerable, punto clave que será analizado en mayor profundidad en el siguiente capítulo *Resultados obtenidos y conclusión.*

Además, la necesidad de definir una función de aptitud clara y adecuada puede llegar a ser compleja, especialmente en problemas con objetivos múltiples o poco definidos. Por último, existe el riesgo de convergencia prematura, donde la población se enfoca en soluciones subóptimas si no se asegura suficiente diversidad genética durante la ejecución del algoritmo, la cual se puede evitar gracias principalmente a la existencia y el ajuste de la tasa de mutación, aunque también influyen otras variables.

4.2 Cheapest Insertion

Como hemos visto, uno de los factores diferenciales al aplicar *Algoritmo Genético* es el iniciar el proceso con una buena población inicial, puesto que influye directamente en la eficiencia y calidad de las soluciones que obtendremos. Al poseer los individuos iniciales un resultado aceptable, reducimos el número de generaciones que el algoritmo necesitará para mejorar las soluciones, lo que reduce considerablemente el tiempo de ejecución. Esto facilita que se oriente el estudio hacia regiones del espacio con mejor fitness.

También, los parámetros como la tasa de mutación o la función de cruce actúan como una barrera para evitar el estancamiento en un mínimo local, ya que buscan aumentar la diversidad en la población, lo cual facilita el análisis de diversas soluciones. Todo ello ocasiona una buena sinergia que nos permite aplicar el algoritmo de forma eficiente maximizando el desempeño de este.

En nuestro caso, se ha optado por emplear la heurística del *Cheapest Insertion* [34] para generar la población inicial del *Algoritmo Genético*. Esta elección se basa en su facilidad de aplicación computacional en comparación con heurísticas más complejas, lo que permite reducir significativamente los tiempos de cálculo. Y además, ofrece soluciones iniciales que, a pesar de no ser óptimas, suelen ser razonablemente buenas. Todo esto facilita el proceso de convergencia al proporcionar rutas iniciales válidas y de costes moderados, los cuales pueden ser mejorados fácilmente con la aplicación de las operaciones de cruce y mutación propias del *Algoritmo Genético*.

El procedimiento de esta heurística se basa en construir un camino de forma iterativa, añadiendo nodos al recorrido existente con el menor incremento posible en el coste total. Este proceso comienza seleccionando un par inicial de nodos que forman un camino básico, por lo general la elección de estos dos nodos se realiza de forma aleatoria.

A continuación, se evalúan iterativamente los nodos que aún no forman parte del camino. Para cada nodo, se calcula el aumento en el coste total que implicaría insertarlo en cada posible posición del camino actual. Este incremento se mide comparando la distancia total antes y después de realizar la inserción. De entre todas las combinaciones evaluadas, se selecciona el nodo y la posición que generan el menor aumento en el coste. Este proceso se repite hasta que todos los nodos han sido incorporados al camino.

Una vez que todos los nodos están incluidos, el algoritmo conecta el último nodo añadido con el primero, formando un circuito cerrado. Este método garantiza una construcción progresiva del recorrido con un aumento mínimo en el costo en cada paso, aunque no asegura encontrar la solución óptima global.

El principal inconveniente de este método es su carácter determinista, ya que tiende a generar soluciones similares en cada ejecución. Además, puede ocasionar la formación de ciclos internos o patrones ineficientes en algunas instancias del problema. Sin embargo, estas desventajas se mitigan en nuestro caso gracias a dos factores: la restricción de no permitir pasar por la misma ciudad más de una vez, y el uso del *Cheapest Insertion* únicamente como generador de la población inicial del *Algoritmo Genético*. Esto evita que se produzcan los posibles bucles o ineficiencias característicos de esta heurística.

5 PROGRAMACIÓN DEL ALGORITMO

Tras haber estudiado en profundidad los algoritmos empleados para abordar el *Travelling Salesman Problem* desde un enfoque más teórico, en este capítulo nos centraremos en el análisis práctico realizado utilizando el lenguaje de programación Python 3.11 en el entorno de desarrollo Spyder.

Se presentarán las distintas funciones implementadas, junto con sus posibles variantes, con el objetivo de profundizar en el Problema del Viajante y su complejidad, aprovechando para programar y simular las diferencias que existen entre las posibilidades que nos ofrece este problema. En el próximo capítulo expondremos como afectan estas a los resultados obtenidos.

El funcionamiento de nuestro código de forma esquematizada puede apreciarse en el siguiente pseudocódigo:

```
Crear clase Ciudad
Crear clase Fitness
    Calcular distancia tota de la ruta
    Calcular fitness de la ruta
Crear ruta inicial
Crear población inicial
MIENTRAS no se cumpla el criterio de terminación
    Ordenar rutas según Fitness
    Seleccionar rutas para reproducción
    Crear población a partir de rutas seleccionadas
    Cruzar población
    Mutar población
    Calcular resultado de generación actual
    SI Resultado > Mejor resultado hasta ahora
        ContadorNoMejora = 0
    SINO
        Aumentar ContadorNoMejora
        SI ContadorNoMejora >= LimiteNoMejora
            Aumentar TasaMutacion
            ContadorNoMejora = 0
    Crear nueva generación
    Aumentar contador generaciones
    SI TasaMutacion > TasaMutacionMinima
        Adaptar TasaMutacion según NoGeneraciones
    SINO
         TasaMutacion = TasaMutacionMinima
    Establecer población actual como nueva población inicial
FIN MIENTRAS
```

Imprimir por pantalla mejor ruta calculada

Imprimir por pantalla mejor resultado calculado

Tras comprender el esquema general del algoritmo, profundizaremos ahora los conceptos y funciones creados para realizar este estudio. Todas las funciones empleadas en el código han sido nombradas intuitivamente para facilitar su identificación y compresión. Se comenzará por definir como se ha implementado la clase Fitness.

5.1 Fitness

Partiendo de que el Fitness está directamente relacionado con la distancia total recorrida: a menor distancia, mayor será el fitness. En la implementación, el fitness se calcula como la inversa de la distancia total de la ruta, permitiendo que las soluciones más cortas obtengan valores de fitness más altos.

Para ello, he definido una clase en Python llamada *Fitness*. Las clases son un componente esencial de la programación orientada a objetos, permitiendo agrupar datos y comportamientos relacionados dentro de una misma entidad. En este caso, nuestra clase organiza información sobre una ruta y proporciona métodos para calcular su distancia total y su fitness, usando funciones para el cálculo de la distancia total y el fitness de la ruta. Esta organización mejora la claridad y estructura del programa.

Con la función *DistanciaRuta* de la clase, calculamos la distancia total de cada ruta sumando las distancias entre ciudades consecutivas y cerrando el recorrido al retornar a la ciudad de origen. Este valor se almacena para evitar recalcularlo innecesariamente en futuras operaciones.

Por otro lado, la función *EvaluacionRuta,* utiliza esta distancia para calcular el fitness como la inversa de la distancia total.

5.2 Población inicial

Ya hemos visto la importancia de comenzar nuestro algoritmo con una población inicial adecuada. Para ello, hemos implementado en nuestro código dos versiones para obtener la primera población.

Con la primera de ellas, aplicando la función *CreaRuta,* se lee el tamaño de la población que introducimos como dato de partida y nos devuelve una única ruta ordenada aleatoriamente. Que, al pasar por la función *PoblacionInicial,* crea el conjunto de rutas aleatorias que forman la población inicial.

Como alternativa al método aleatorio, y para evaluar la influencia de la población inicial, se utiliza la función *CreaRutaCheapestInsertion,* donde se aplica la heurística que da nombre a la función para el cálculo de la ruta de un individuo. Siguiendo el mismo proceso del método anterior, con la aplicación de la función *PoblacionInicial* se genera la población inicial.

5.3 Selección y reproducción

Una vez disponemos del conjunto de rutas que forman nuestra población, se deben seleccionar cuáles se utilizarán para crear la siguiente población. Para ello, se utilizarán tres métodos diferentes: la selección elitista, que trabaja en conjunto con uno de los otros dos métodos estudiados, la selección por torneo y la selección por ruleta suavizada.

Antes de pasar a seleccionar las rutas, son ordenadas de forma descendente según su Fitness, de modo que la primera ruta de la población sea la que mayor aptitud tenga. Este paso se realiza aplicando la función *RankingRutas* que devuelve la población ordenada y actúa como dato de entrada para las funciones de selección

Por un lado, la selección por torneo se realiza con la función del código *SeleccionTorneo*, la cual también lee como entrada el tamaño del torneo, *TamTorneo*, que establece el número de individuos que

competirán entre sí. Las rutas ganadoras se almacenan, y tras pasar por la función *Reproduccion* crea la población con las rutas elegidas para reproducirse.

Por el otro, la selección por rueda de ruleta se aplica con *SeleccionRuleta*, donde se eligen los individuos proporcionalmente a su fitness. Aquí, se ha desarrollado una mejora para el afinado de la selección añadiendo el parámetro *Suavizacion*, el cual nos permite aumentar la ventaja de las rutas con mayor aptitud, con valores mayores a uno, u otorgar más posibilidades de selección a las rutas con menor fitness, con valores menores a la unidad. Al igual que la selección por torneo, se emplea *Reproducción* para generar la población.

La selección elitista está integrada en ambas funciones, actuando de forma conjunta a sendas técnicas de selección. Donde se añaden los mejores individuos al global de rutas escogidas en las funciones vistas en este apartado. Aplicando el parámetro de entrada *TamElite,* que establece el número de rutas de elite del algoritmo.

5.4 Cruce

Para efectuar el cruce y de forma análoga a lo que ocurre en la generación de la población inicial o la elección tenemos una primera función *CruceRuta*, donde se establece el proceso para un único individuo, y una segunda función *Cruce*, en la que se crea la población ejecutando *CruceRuta* para cada individuo.

En nuestro caso, el cruce es de dos puntos, concretamente de *Cruce de Orden* u *Order Crossover* en inglés, donde se selecciona un segmento al azar del primer padre y se añade directamente al hijo en la misma posición de la que se extrajo, y se rellena la ruta con las ciudades restantes del segundo padre siguiendo el orden en que aparecen evitando los duplicados.

El primer padre es escogido aleatoriamente, y el segundo será el de su posición inversa un 25% de las veces. El resto de las ocasiones será el de su posición inversa suavizada, siendo esta una ruta más cercana al primer padre, promoviendo así el cruce entre rutas de calidad similar. Puesto que, el uso exclusivo del opuesto podría influir negativamente en la convergencia de nuestro algoritmo.

Cabe destacar que las rutas escogidas como elites no pasan por este proceso, y directamente formarán parte de la siguiente población. Aunque, sí que forman parte del conjunto del que se obtienen las rutas empleadas en la fase de cruce.

5.5 Mutación

Para finalizar la aplicación del algoritmo genético, solo faltaría efectuar el último paso, la mutación. Para ello, se emplean las funciones *MutaRuta* y *MutaPoblacion*. Las cuales, siguiendo la misma línea del resto del código, ejecutan la mutación primero para una ruta únicamente y después se traslada a la dimensión de población.

El proceso consiste en iterar sobre cada ciudad de cada individuo, generando un número aleatorio por cada una de ellas. Si este número es menor que la tasa de mutación dada, *TasaMutacion*, se realiza un intercambio entre la ciudad actual y una aleatoria dentro de la misma ruta.

Una vez obtenemos la ruta final, aplicamos *MutaPoblacion* para conseguir la población final que será utilizada como población inicial del algoritmo hasta que lleguemos al criterio de parada.

5.6 Ejecución

Una vez definidos todos los pasos para calcular una nueva generación, creamos una función específica, llamada *NuevaGeneracion*, que reúne las llamadas a las funciones vistas anteriormente para obtener la nueva población.

Además, se empleará la función *LeerCoordenadasInstancia* para obtener los valores de la instancia que se usará en el código y asignarlos a cada ciudad en la clase *Ciudad*.

Por último, se encuentra la función *AlgGen*, que actúa como eje central del algoritmo. Esta función se

encarga de: efectúa la llamada para crear nuevas generaciones, calcula el resultado obtenido en cada generación, verifica el cumplimiento del criterio de parada y modula las modificaciones que podemos aplicarle a la tasa de mutación. Para ejecutar el algoritmo, únicamente se deberían modificar los valores de los parámetros que actúan como entrada de la función. Esta nos ofrecerá el resultado final y la ruta óptima calculada.

5.7 Parametrización

Antes de analizar los resultados, es conveniente entrar en detalle de algunos de los parámetros que afectan al desempeño final del algoritmo. El diseño de los parámetros y la elección de sus valores se ha realizado de forma experimental, se detallará en el capítulo 6, en el apartado *Análisis comparativo*.

El tamaño de la población, *TamPoblacion*, afecta significativamente en la diversidad genética y el tiempo de ejecución del programa. Una población pequeña favorece la convergencia prematura al reducir la diversidad inicial, puesto que no se explora tanto el espacio de búsqueda. Por otro lado, una población grande evita estos problemas a cambio de retrasar la convergencia.

Para ello, se ha establecido una población de 150 individuos, puesto que valoramos en mayor medida el buen desempeño del algoritmo por delante del tiempo de proceso de este. Para paliar el posible efecto de una lenta convergencia se ha marcado el criterio de parada en 400 generaciones, parámetro *NoGeneraciones*.

Respecto a la elección tenemos varios parámetros modificables, para el porcentaje de rutas que serán de elite, *TamElite*. La literatura nos recomienda entre un 1% y 5% del tamaño de la población, pudiendo llegar hasta un 10%. Por lo que se ha escogido un total de 6 rutas de elite, lo que equivale a un 4% [35].

En la elección por torneo, se puede modificar el tamaño del torneo, *TamTorneo*, donde un tamaño pequeño favorece la diversidad, pero reduce la convergencia y, según aumentamos el tamaño, se produce el efecto contrario. En la elección por ruleta suavizada, actúa el factor *Suavizacion* que, como ya hemos visto, afecta a la elección de las rutas con mayor o menor fitness según sea mayor o menos a la unidad.

Dentro de la parametrización de la tasa de mutación, se encuentran diversas variables que se pueden modular. Destacan la *TasaMutacionInicial*, que establece el valor de la tasa de mutación al iniciar el algoritmo, y la *TasaMutacionMinima* marcada en un 2% que limita el mínimo de tasa de mutación del código. Esto se debe a que la tasa de mutación empleada no es una constante, sino que, se reduce según el paso de las generaciones de tres formas diferentes: lineal, cuadrática y logarítmicamente. Esta elección se realiza al aplicar el modelo con la variable *TipoAdaptacion*.

Por último, para evitar el posible estancamiento en óptimos locales, se ha incluido un contador de no mejora, llamado *ContadorNoMejora*. Este contabiliza el número de generaciones sin mejorar el resultado y si supera el *LimiteNoMejora* establecido aumenta la tasa de mutación en un 100%, siempre con el límite superior de la *TasaMutacionInicial*.

6 RESULTADOS OBTENIDOS Y CONCLUSIÓN

Antes de comenzar con el análisis de los resultados que hemos obtenidos al estudiar los diferentes parámetros que disponemos en el *Algoritmo Genético* [31][32]. Debemos presentar la instancia que vamos a analizar. En nuestro caso, hemos escogido una de las instancias de Solomon [36], concretamente la instancia R101.50. La cual se muestra en la tabla a continuación:

CIUDAD Nº	Coord. X	Coord. Y
1	35	35
2	41	49
3	35	17
4	55	45
5	55	20
6	15	30
7	25	30
8	20	50
9	10	43
10	55	60
11	30	60
12	20	65
13	50	35
14	30	25
15	15	10
16	30	5
17	10	20
18	5	30
19	20	40
20	15	60
21	45	65
22	45	20
23	45	10
24	55	5
25	65	35

CIUDAD Nº	Coord. X	Coord. Y
26	65	20
27	45	30
28	35	40
29	41	37
30	64	42
31	40	60
32	31	52
33	35	69
34	53	52
35	65	55
36	63	65
37	2	60
38	20	20
39	5	5
40	60	12
41	40	25
42	42	7
43	24	12
44	23	3
45	11	14
46	6	38
47	2	48
48	8	56
49	13	52
50	6	68

Tabla 1. Coordenadas de la instancia R101.50.

6.1 Resolución óptima con nuestro algoritmo

El objetivo principal de este trabajo es analizar la influencia de los parámetros del algoritmo en el resultado final. Para ello, debemos establecer primero un caso de ejemplo que sirva como marco para la comparación de las diferentes casuísticas que se verán a lo largo del capítulo.

A pesar de que, en la apartado 5.6 *Parametrización* dejamos definidos una parte de los valores que nuestro algoritmo debe tomar, algunos de ellos no quedaron fijados, debido a que estos serán los modificados en el estudio que se realizara a continuación. Para nuestro caso óptimo, se han establecido estos valores, de los cuales se argumentarán aquellos que no fueron justificados anteriormente:

- Tamaño de la población en 150 individuos.
- Tamaño de la población de elite en 6 individuos.
- Tasa de mutación inicial en un 25%.
- Tasa de mutación mínima en un 2%.
- Número de generaciones en 400 unidades.
- Límite de generaciones sin mejora en 25 unidades.
- Tamaño del torneo en 5 individuos.
- Tipo de adaptación es cuadrática.

La tasa de mutación inicial se ha establecido con un 25%, lo cual se podría considerar un valor considerablemente alto. El objetivo que se persigue con ello es aumentar la diversidad y la exploración del espacio de búsqueda para poder evaluar la mayor cantidad de rutas posibles.

Se debe tener en cuenta que la tasa de mutación en nuestro algoritmo es adaptativa por lo que según avanzan las generaciones se va reduciendo su valor propiciando así la convergencia. Esta curva que representa la tasa de mutación tiene tres posibles escenarios, para el caso marco se ha escogido la variación cuadrática.

Continuando en la misma línea que con la tasa de mutación inicial, la selección se realiza por torneo con tan solo cinco individuos compitiendo en cada uno de ello. Este, sin ser un valor extremo si se considera bajo, lo que propicia una mayor diversidad en los individuos analizados [37].

También, se ha establecido un límite de no mejora en 25 individuos. Esta elección sigue la misma filosofía de los parámetros relacionados con el torneo y la mutación inicial al tratar de incorporar cierta aleatoriedad para aumentar la exploración, aunque con un enfoque ligeramente diferente. Puesto que, este factor solo entra en acción en las fases más avanzadas del proceso, donde el sistema se está aproximando a la solución final y, ocasionalmente, transcurren decenas de iteraciones sin que se logre una mejora en el resultado obtenido.

Esto puede deberse a dos razones: que ya nos encontramos en el óptimo o, con mayor probabilidad, estemos atrapados en una ruta subóptima que no posee numerosas alternativas cercana que la mejoren. Al aumentar el tasa de mutación en estas circunstancias, se favorece el análisis de mayor cantidad de rutas próximas a la actual, lo que puede ayudar "afinar" la búsqueda. Esto permite garantizar que la solución es la mejor de al menos una gran región a su alrededor o encontrar alguna que mejore.

```
En el algoritmo actual:
El tamaño de la población es de: 150
El tamaño de la élite es de: 6
La tasa de mutación inicial es de: 0.25
La tasa de mutación mínima es de: 0.02
El número de generaciones es de: 400
El límite de generaciones sin mejora es de: 25
El tamaño del torneo es de: 5
El tipo de adaptación es: Cuadrática
```

Figura 3. Salida por pantalla de las variables del problema.

Por recordar el contexto, la función objetivo de nuestro algoritmo busca minimizar la distancia total que necesita un único camión de reparto para recorrer todas las ciudades recorriendo la menor distancia posible. Teniendo en cuenta que se mide en unidades de distancia genéricas, *u.d.* Al ejecutar el código en un total de veinte ocasiones obtenemos un resultado siempre acotado en el rango de las 470 a las 525 *u.d.*

Siendo el mejor resultado obtenido de 471.16 *u.d.*, el propio código es el que nos ofrece la información tanto del resultado final de la ruta seguida por el camión, como podemos ver en la figura 5, y la representación de la ruta optima, la cual observamos en la figura 4. En ella están representadas todas las ciudades como círculos azules, destacando en color verde el punto de inicio desde donde sale el camión y comienza a la ruta, y en color rojo la última ciudad que es visitada antes de volver al punto de salida, que podríamos considerar el almacén o centro logístico.

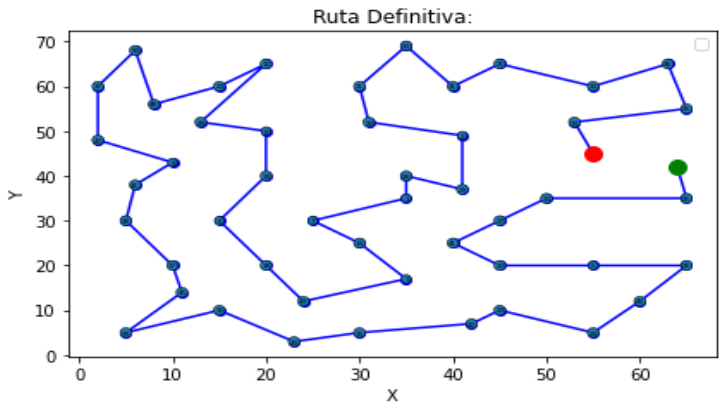

Figura 4. Representación de la mejor ruta calculada del TSP.

```
Mejor Ruta: [(30), (25), (13), (27), (41), (22), (5), (26), (40), (24), (23), (42), (16), (44), (15), (39), (45), (17), (18),
(46), (9), (47), (37), (50), (48), (20), (12), (49), (8), (19), (6), (38), (43), (3), (14), (7), (1), (28), (29), (2), (32),
(11), (33), (31), (21), (10), (36), (35), (34), (4)]
Distancia final: 471.16
```

Figura 5. Impresión por pantalla de la ruta y el valor final.

6.2 Análisis comparativo

Una vez tenemos la referencia de nuestro óptimo, podemos estudiar como influyen en el desempeño del algoritmo y en el resultado final las variables que tenemos a nuestra disposición. La mayor parte de los conceptos que veremos a continuación han sido desarrollados con anterioridad en este TFG, por lo tanto, únicamente se expondrán breves explicaciones sobre estos y su contexto. Con algunas excepciones donde sea necesario, entrar en mayor detalle.

6.2.1 Parámetros de selección

En los dos procesos de selección vistos hasta ahora, de ruleta suavizada y de torneo, tenemos una variable que modula y dirige su aplicación.

En el caso de la *selección por torneo*, el valor del número de individuos que competirán entre sí afecta directamente en la convergencia del problema. Para analizar su influencia hemos simulado tres situaciones diferentes. La primera es con un tamaño de torneo de tan solo dos individuos; la segunda, es con un tamaño mediano de ocho individuos; y la última será con cuarenta competidores por torneo, la cual es una cifra exagerada ya que no se suelen emplear tamaños de esas dimensiones.

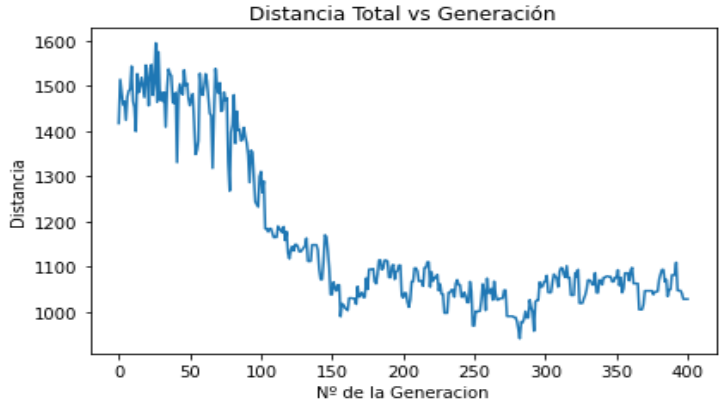

Figura 6. Representación de la solución para *TamTorneo=2*.

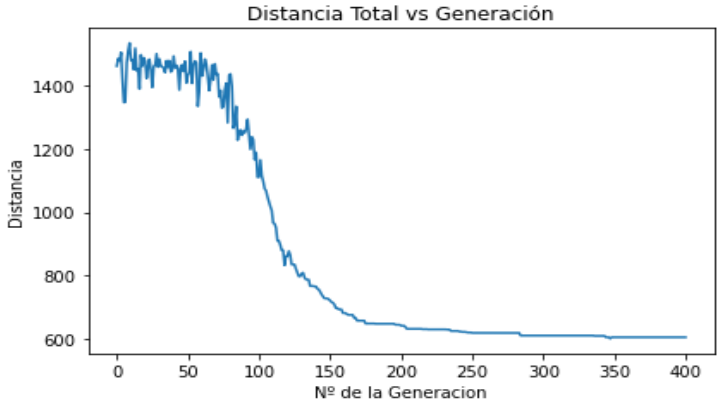

Figura 7. Representación de la solución para *TamTorneo=8.*

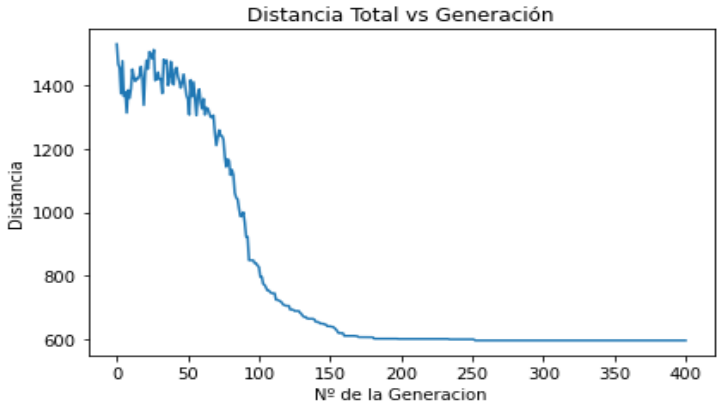

Figura 8. Representación de la solución para *TamTorneo=40.*

En las gráficas presentadas se muestra cómo evoluciona la función objetivo, distancia total recorrida en la ruta, en el eje respecto al paso de las generaciones de nuestro *Algoritmo Genético*. Esta estructura se repetirá a lo largo del capítulo.

Como podemos observar en las figuras 6, 7 y 8, en el primer caso donde únicamente dos individuos se enfrentaban, nuestro algoritmo no logra converger en una solución, a pesar de haber modificado la variable del tamaño de torneo exclusivamente, sin alterar ningún otro valor respecto a la versión óptima. El hecho de no converger es, aunque esperado, altamente llamativo, puesto que no solo no se converge a un valor cercano al obtenido en nuestro caso de referencia si no que a pesar del paso de 400 generaciones ni si quiera logra establecerse alrededor de un valor estable. Moviéndose en un entorno de las 900 a las 1100 *u.d.* más de la mitad del desarrollo del algoritmo.

Este resultado era lo esperado, puesto que la reducción del tamaño de los torneos, aunque aumenta la diversidad al dar relevancia a individuos menos beneficiosos, si llegamos a la situación extrema (dos individuos únicamente) se dificulta considerablemente la convergencia como hemos visto en la práctica.

Al analizar los dos casos restantes, se observa que, según aumentamos el número de individuos enfrentados, también incrementa la convergencia. Es cierto que, no se aprecian grandes diferencias a simple vista entre la segunda y la tercera gráfica, esto se debe a que nuestro código está diseñado para realizar una gran exploración inicial, empleando una alta tasa de mutación en las primeras etapas. No obstante, si analizamos el valor de la función objetivo para el instante en el que se alcanzan las 100 generaciones, en el segundo escenario se registra un valor superior a las 1000 *u.d.* mientras que en el tercero se tiene algo menos de 800 *u.d.*

A pesar de la similitud en la forma de las curvas entre ambos casos y su alto contraste con el primer escenario, lo cual nos invita a pensar que son muy parecidas, se puede corroborar la relación entre el aumento del tamaño del torneo y una convergencia más rápida.

Por último, se ha estudiado la influencia del tamaño del torneo en el tiempo de ejecución del algoritmo. Para el caso de un torneo de 8 individuos el tiempo de ejecución medio es de 8.68 segundos y en el caso de 40

individuos este valor asciende hasta los 14.73 segundos. Este aumento del 69.71% se debe al crecimiento del número de individuos que son comparados en cada torneo. Y puesto que se realiza un torneo por cada individuo, según crece el tamaño de los torneos se realizarán más operaciones de selección, lo que aumenta el total de operaciones a procesar computacionalmente retrasando la obtención del resultado final.

Una vez hemos analizado la influencia de la *selección por torneo* en el algoritmo corresponde lo propio con su alternativa la *selección por rueda de ruleta*. En esta técnica, el factor a parametrizar es la *Suavizacion,* nomenclatura empleada para definir el parámetro en Python y mencionada en el capítulo *Parametrización*. Siguiendo el mismo patrón del caso anterior vamos a simular tres valores del parámetro para asegurar un análisis completo. Se establecen los valores de *Suavizacion* en 0.1, 1 y 10.

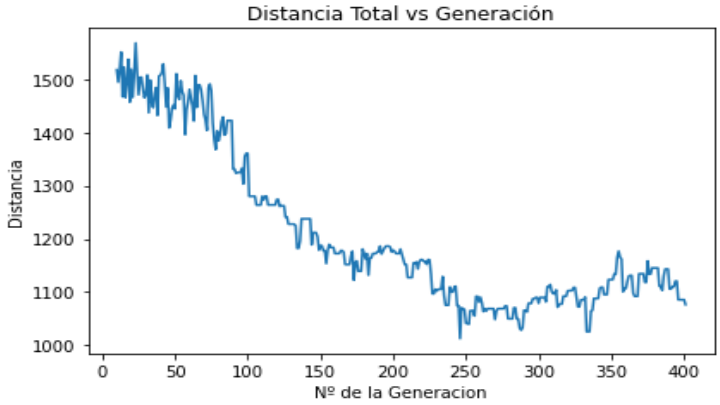

Figura 9. Representación de la solución para *Suavizacion = 0.1.*

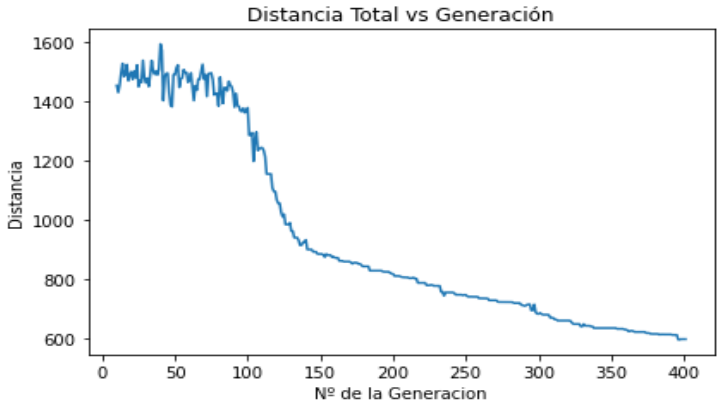

Figura 10. Representación de la solución para *Suavizacion = 1.*

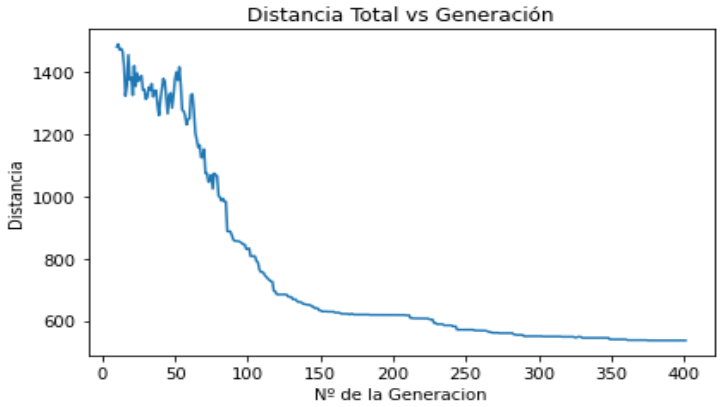

Figura 11. Representación de la solución para *Suavizacion = 10.*

De forma similar a como ocurría con la selección por torneo la primera simulación que correspondía al menor de los valores el algoritmo no converge para el número de generaciones simulado. Se ha realizado una cuarta simulación con este mismo valor de *Suavizacion* pero dejando el código calcular hasta las 1000 generaciones para comprobar si es una cuestión de tiempo la convergencia o del que sería un "mal diseño" de la variable.

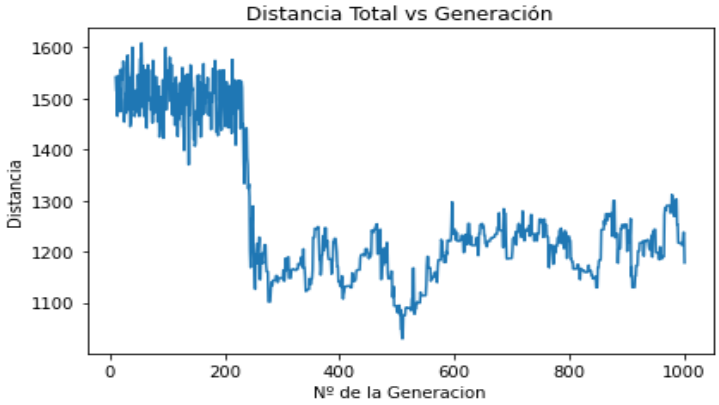

Figura 12. Representación de la solución para *Suavizacion = 0.1* con el transcurso de 1000 generaciones.

Como se puede observar, el problema de la falta de convergencia hacia un valor estable no se debe al número de generaciones simuladas, factor que esperábamos, puesto que 400 generaciones ya es un tamaño considerable. Por consiguiente, se puede concluir que, si trabajamos con valores que potencian en demasía la exploración del espacio de búsqueda por encima de la convergencia no se conseguirán resultados aceptables. Cabe destacar que, sí que se podría interpretar una cierta convergencia en este último escenario en torno a valores de 1150-1200 *u.d.* Sin embargo, además de encontrarse lejos del resultado de nuestro caso de referencia estos valores presentan un alto nivel de ruido, que podemos describir como perturbaciones, de forma análoga al uso del concepto en contextos de sistemas de control.

En la segunda y tercera simulación del factor de suavización si podemos observar una mayor diferencia entre sí de la que se extraía de las gráficas del método de selección por torneo. En este caso, el valor unitario de nuestro parámetro implica que se emplea directamente el valor de la aptitud de los individuos para su selección. Esto, pese a su convergencia final en un resultado aceptable contrasta con el desempeño que nos ofrece el escenario con mayor presión selectiva. Donde con una *Suavizacion* igual a 10, el algoritmo converge rápidamente hacia una solución con buen resultado final [38].

6.2.2 Tasa de mutación

La tasa de mutación es uno de los parámetros con mayor influencia en el desarrollo de nuestro algoritmo. Por ello, tiene multitud de variables asociadas como ya se detalló en el capítulo 5.6. Para realizar un estudio completo, vamos a ejecutar dos simulaciones diferentes.

En primer lugar, se pretende concretar el efecto de la tasa de mutación inicial, parámetro *TasaMutacionInicial* sobre el algoritmo. Para ello, se estudian tres valores.

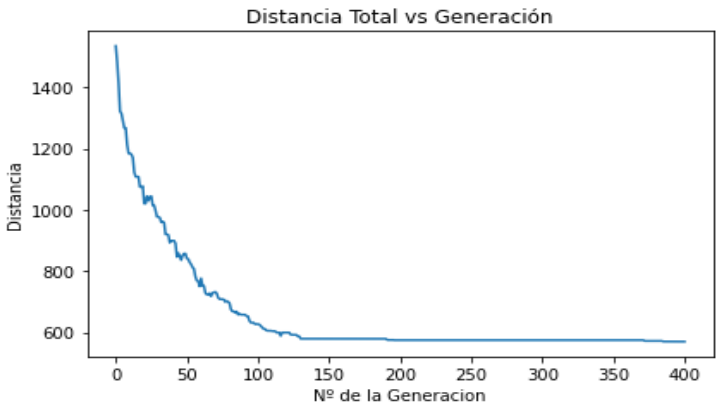

Figura 13. Representación de la solución para *TasaMutacionInicial = 0,02.*

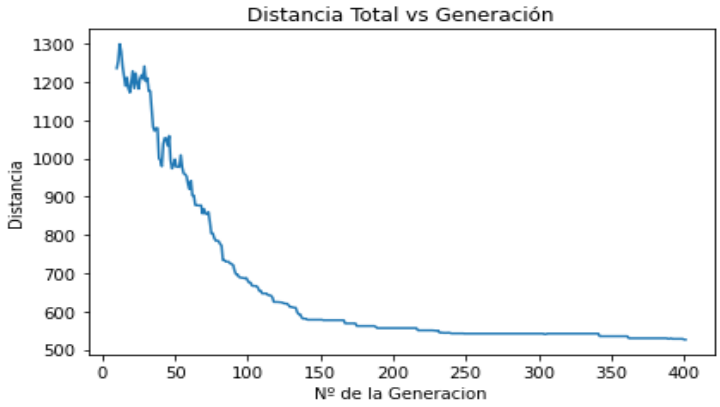

Figura 14. Representación de la solución para *TasaMutacionInicial = 0,25.*

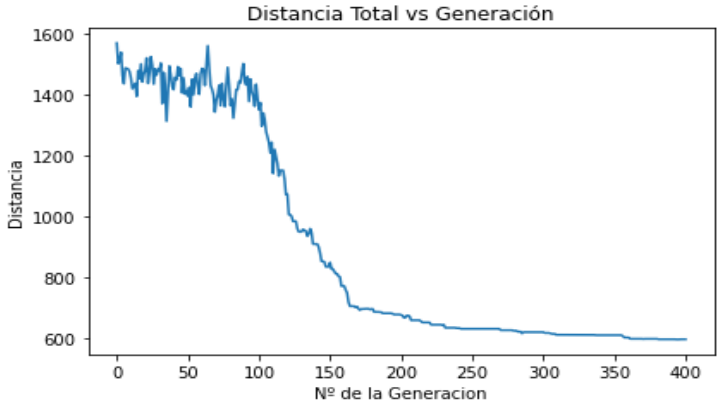

Figura 15. Representación de la solución para *TasaMutacionInicial = 0,5.*

Como se puede apreciar en las tres situaciones el algoritmo converge hacia un valor final muy cercano al óptimo, siendo el último tramo de las gráficas muy similar. Sin embargo, donde encontramos los contrastes es en las primeras fases del desarrollo. En el primer escenario, debido a que solo se permite un 2% de mutación no se aprecia a penas desviación en la mejora de las soluciones por lo que rápidamente alcanza un valor de 600 *u.d.* a la barrera de las 100 generaciones.

En la tercera prueba, encontramos el caso contrario. Donde ocasionado por la extremadamente alta mutación inicial, el código explora activamente un gran número de rutas, por lo que al paso de las 100 generaciones aún no se ha sobrepasado la barrera de las 1200 *u.d.* Esta exploración se ve compensada gracias a la reducción progresiva de la tasa de mutación que realizamos con el trascurso de las generaciones. Lo que acaba provocando que a pesar de no obtener inicialmente un buen grado de convergencia en las etapas finales del algoritmo este se consiga.

En una situación intermedia entre los dos escenarios analizados previamente, encontramos la gráfica con mejor desempeño. Puesto que, con una *TasaMutacionInicial* del 25%, aunque en algunos contextos se podría considerar algo elevada, se nos permite evaluar gran cantidad de alternativas inicialmente y aun así converger hacia una solución que ofrezca muy buen resultado. Al igual que en el tercer caso, nos podemos permitir trabajar con mutaciones un poco mayores a lo usual gracias a la adaptación de la mutación que realiza nuestro algoritmo.

Ya que hemos analizado la ventaja que nos ofrece la adaptación de la mutación con el paso de las generaciones, podemos entrar un poco más en detalle para conocer mejor su funcionamiento.

Como ya hemos visto, la disminución gradual de la mutación es un factor clave que nos permite encontrar el equilibrio entre un análisis inicial de amplias regiones del espacio de búsqueda y una posterior

convergencia en alguna de las mejores rutas valoradas.

Para realizar este proceso tenemos tres alternativas, a continuación, se explican brevemente y se muestra la evolución de la tasa de mutación que genera cada una:

- La *adaptación lineal*, la cual fue la primera versión que se implementó como función de control de la tasa de mutación. Tras el estudio en profundidad del problema se descubrió una mejor alternativa.
- La *adaptación cuadrática*, es la empleada en nuestro ejemplo de referencia. Puesto que es la que mejor resultados nos ha ofrecido. A diferencia de la adaptación original, esta garantiza mayor velocidad en la convergencia provocada ya que representa una curva no lineal
- La *adaptación logarítmica*, basándonos en la idea de aumentar la velocidad de convergencia surgió el planteamiento de emplear una variación logarítmica que nos permite una rápida reducción de la tasa de mutación.

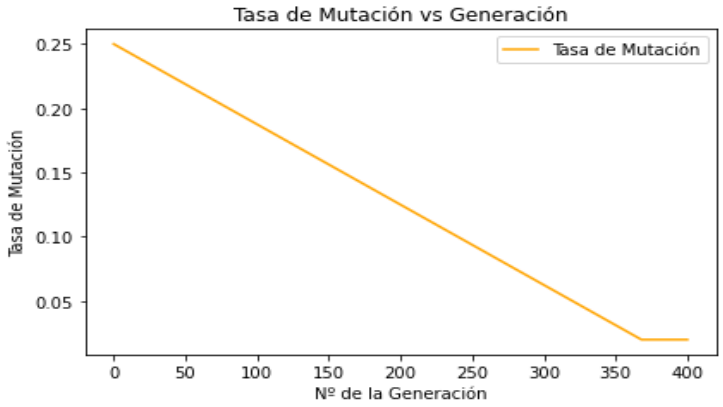

Figura 16. Representación de la evolución de la tasa de mutación con la adaptación lineal.

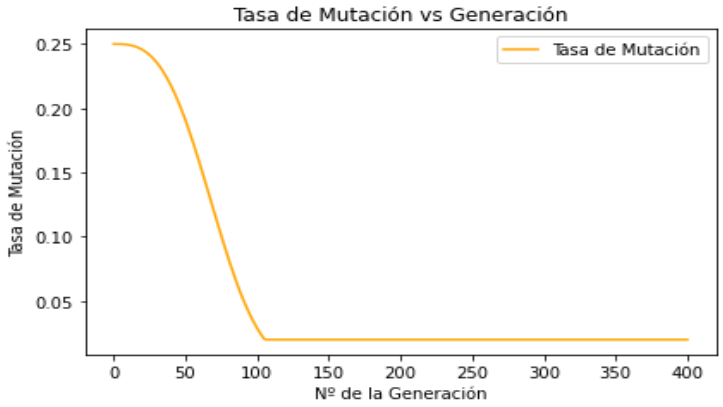

Figura 17. Representación de la evolución de la tasa de mutación con la adaptación cuadrática.

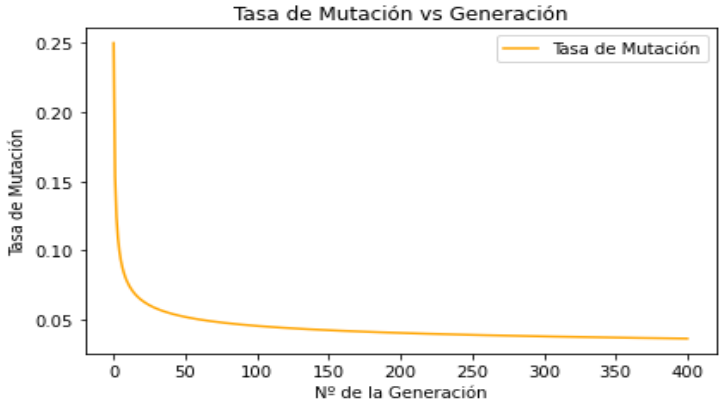

Figura 18. Representación de la evolución de la tasa de mutación con la adaptación logarítmica.

Analizaremos ahora como afecta a los resultados que nos ofrece el algoritmo cada una de ellas, para ello se muestran los resultados de su aplicación.

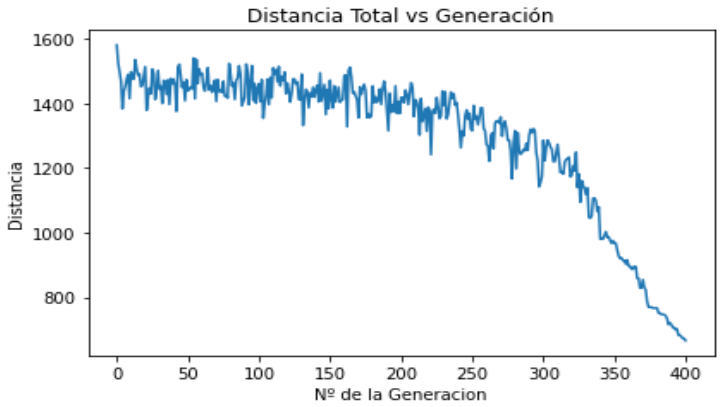

Figura 19. Representación de la solución con adaptación lineal de la tasa de mutación.

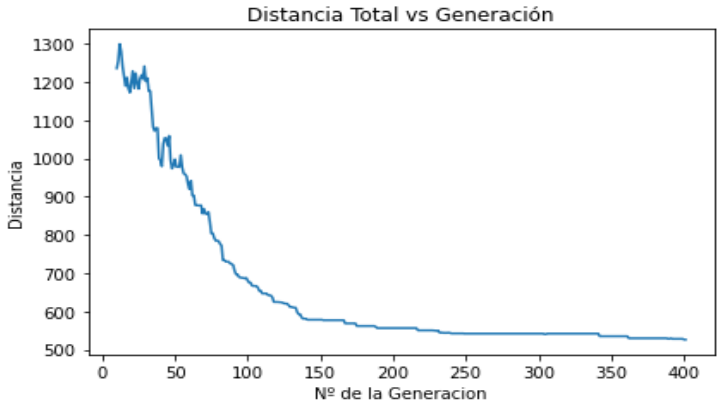

Figura 20. Representación de la solución con adaptación cuadrática de la tasa de mutación.

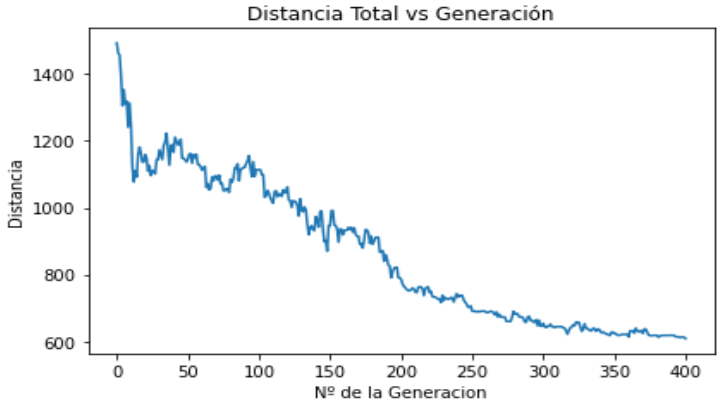

Figura 21. Representación de la solución con adaptación cuadrática de la tasa de mutación.

Como se puede comprobar, el método que peor resultados aporta es el *lineal*. Puesto que, como observamos en la figura 19 se tienen valores de mutación mayores al 5% hasta pasadas las 300 generaciones, lo que dificulta en gran medida la convergencia hasta que no se alcanzan valores menores durante el último tramo, cuando ya es tarde para conseguir un desempeño óptimo.

Tanto la *adaptación cuadrática* como la *logarítmica* comparten ciertos patrones en sus respectivas simulaciones con un grado de diversidad y exploración en las etapas iniciales, aunque difieren según avanzan las generaciones. La función logarítmica a pesar de reducir velozmente la tasa de mutación en el inicio, factor

que ya hemos concluido como positivo para el sistema, permite hasta etapas muy avanzadas valores de mutación en torno al 4-5% lo que provoca cierta inestabilidad en el algoritmo.

Por todo esto, el sistema de adaptación que mejor resultado asegura es la *adaptación cuadrática*, puesto que permite un amplio análisis de las soluciones iniciales y rápidamente converge garantizando un buen resultado final.

Para sintetizar la información presentada hasta ahora, se ha creado la tabla de *Parámetros Analizados*, donde se exponen las variables estudiadas, el rango de valores o alternativas evaluadas y el valor que ha mostrado un mejor desempeño.

Parámetro	Rango de Valores Estudiado	Valor Destacado
TamTorneo	[2 , 40]	8
Suavizacion	[0.1 , 10]	10
TasaMutacionInicial	[0.02 , 0.50]	0.25
TipoAdaptacion	Lineal; Cuadrática; Logarítmica.	Cuadrática

Tabla 2. Parámetros analizados.

Es importante destacar que este análisis se ha realizado con el propósito de comprender e indagar en cómo influyen las diversas variables utilizadas en el desempeño del *Algoritmo Genético*. Por esta razón, se ha centrado el estudio en analizar cómo varía el resultado obtenido al modificar los valores de cada parámetro. Esto nos ha permitido una mejor comprensión del comportamiento del código y su sensibilidad a la convergencia, especialmente al acercarnos a los valores límite de las factores estudiados.

También, es relevante mencionar que, aunque se ha estudiado el efecto de cada parámetro de forma individual, estos interactúan entre sí. En este sentido, debido al gran numerosas combinaciones posibles, se debe procurar siempre un equilibrio. Por ejemplo, si se selecciona un valor en una variable que prioriza la exploración del espacio de soluciones, deberá compensarse con otra variable cuyo valor promueva la convergencia tratando de encontrar el equilibrio entre ambas situaciones para optimizar el rendimiento del algoritmo.

6.2.3 Conclusiones

Para generar la primera versión del programa se utilizó el pseudocódigo del Algoritmo Genético presentado en la asignatura de *Programación de Operaciones*, que se adjunta en el anexo. Aunque en esta asignatura su implementación estaba orientada a la optimización de fabricación de lotes en una fábrica, sirvió como base para desarrollar nuestro algoritmo. Además, los conocimientos adquiridos en la asignatura han hecho más sencillo el trabajo con el lenguaje de programación Python, debido al gran número de horas empleadas en su aprendizaje.

Este estudio surge de la propia experiencia al tratar de programar el algoritmo. Puesto que, inicialmente, se intentó hallar el método que analizase el mayor número de soluciones, lo cual llevó a diferentes versiones del código que no conseguían converger en un resultado aceptable. Tras ello, comparé mi *Algoritmo Genético* con las numerosas versiones que existen por internet y pude observar que no existían grandes diferencias en la estructura o la forma de ejecutar el mismo. Por lo que el problema no debía estar en las funciones que forman el esqueleto del código.

Tras un periodo de cierta frustración, me dediqué a investigar otras alternativas. Fue entonces cuando descubrí algunos artículos ([35], [37], [38], [39]) donde se mencionaba y analizaba la importancia de la selección o la tasa de mutación, entre otros parámetros, como claves para la convergencia del algoritmo, lo que me abrió un camino a explorar. Este proceso generó la pregunta sobre qué y cómo se modula la convergencia de un Problema del Viajante resuelto con un *Algoritmo Genético*.

A partir de aquí me decidí a estudiar el efecto de estas variables en mi problema. Lo que propició en primer lugar un análisis de los parámetros que podía incluir y cómo modificarlos para simular la mayor cantidad de escenarios posibles, con el objetivo de entender cómo afectaba cada uno de ellos al problema. Todo esto, conllevó finalmente a un correcto funcionamiento de mi código.

Cabe destacar que una de las mayores dificultades que he encontrado a la hora de crear el código ha sido al ir añadiendo mayor diversidad y control en las variables del problema. Puesto que, aunque la base del algoritmo genético es siempre muy similar, según se añaden al algoritmo variables y parámetros, se requiere cada vez de un mayor nivel de adaptación y refinamiento del código para que todas las funciones funcionen de forma sincronizada. Especialmente

En resumen, los análisis realizados en este TFG nos permiten concluir que en la aplicación de un *Algoritmo Genético* es fundamental diseñar y ajustar adecuadamente los parámetros y variables que lo componen. Ya que, la selección errónea de alguno de estos indicadores puede afectar negativamente al funcionamiento, aunque sea un algoritmo correctamente diseñado. En particular, los valores que promueven la diversidad genética y la exploración del conjunto de soluciones se deben evaluar cuidadosamente para encontrar el equilibrio óptimo. Se debe valorar en cada circunstancia si se prioriza la necesidad de hallar la solución más cercana al óptimo, o la mayor convergencia del problema.

Finalmente, una de las posibles mejoras o ampliaciones que se podrían realizar en futuros trabajos sería el estudio de la parametrización de otros métodos heurísticos y metaheurísticos especialmente en los problemas con las variantes de mayor complejidad computacional y con 100 o más nodos.

REFERENCIAS

[1] Clay Mathematics Institute. (2022). “Millennium Prize Rules”. [En línea]. Disponible en: https://www.claymath.org/millennium-problems/rules/. [Consultado: 21 de septiembre de 2024].

[2] Larrañaga, P., Kuijpers, C., Murga, R., et al. (1999). "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators". Artificial Intelligence Review, vol. 13, pp. 129–170. [En línea]. Disponible en: https://doi.org/10.1023/A:1006529012972. [Consulta: 21 de septiembre de 2024].

[3] Barnett, J.H. (2009). "Early writings on graph theory: Hamiltonian circuits and the Icosian game". [En línea]. Disponible en: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=36b18e8664a2a77c334c445ebd7f2cf2682e045d. [Consulta: 22 de septiembre de 2024].

[4] RUBIN, Stuart H., et al. (2016). "Reusing the NP-hard traveling-salesman problem to demonstrate that P ≈ NP". IEEE 17th International Conference on Information Reuse and Integration (IRI). IEEE, pp. 574-581. [En línea]. Disponible en: https://ieeexplore.ieee.org/document/7785793. [Consulta: 12 de octubre de 2024].

[5] González Velarde, José Luis; Ríos Mercado, Roger Z. (1999). "Investigación de Operaciones en Acción: Aplicación del TSP en Problemas de Manufactura y Logística". Universidad de Colorado, Escuela de Graduados en Negocios y Administración; Texas A&M University, Departamento de Ingeniería Industrial. [En línea]. Disponible en: http://eprints.uanl.mx/9986/1/4_Jose_L_Gonzalez_investigacion_de_operaciones.pdf. [Consulta: 12 de octubre de 2024].

[6] ORMAN, A. J. y WILLIAMS, H. P. (2004). "A survey of different integer programming formulations of the traveling salesman problem". Operational Research Working Papers, LSEOR 04.67. Department of Operational Research, London School of Economics and Political Science, Londres. [En línea]. Disponible en: https://eprints.lse.ac.uk/9349/1/WP67_A_Survey_of_DifferentFormulationsoftheTSPJuly20051LSEROVERSION.pdf. [Consulta: 27 de diciembre de 2024].

[7] AKGÜN, Ibrahim. (2010). "Min-degree constrained minimum spanning tree problem: New formulation via Miller–Tucker–Zemlin constraints". Computers & Operations Research. [En línea]. Disponible en: https://www.sciencedirect.com/science/article/abs/pii/S0305054809000859. [Consulta: 14 de octubre de 2024].

[8] GUTIN, G. y PUNNEN, A. (2002). The Traveling Salesman Problem and Its Variations. Kluwer Academic Publishers. [En línea]. Disponible en: https://www.researchgate.net/profile/Matteo-Fischetti/publication/225979577_The_Generalized_Traveling_Salesman_and_Orienteering_Problems/links/5ccebcde92851c4eab83c09f/The-Generalized-Traveling-Salesman-and-Orienteering-Problems.pdf. [Consulta: 15 de octubre de 2024].

[9] BARVINOK, Alexander; GIMADI, Edward Kh; SERDYUKOV, Anatoliy I. (2007). "The maximum TSP". En: The Traveling Salesman Problem and Its Variations, pp. 585–607.

[10] BUITRAGO SUESCÚN, Oscar Yecid; BRITTO AGUDELO, Rodrigo Alberto; MEJÍA DELGADILLO, Gonzalo. (2007). "Análisis comparativo de colonia de hormigas vs. un enfoque combinado cuello de botella móvil/búsqueda tabú en la minimización de la tardanza ponderada total en sistemas de manufactura tipo taller". [En línea]. Disponible en: https://repository.unab.edu.co/handle/20.500.12749/8997. [Consulta: 15 de octubre de 2024].

[11] OCAMPO, Eliana Mirledy Toro; BOLAÑOS, Rubén Iván; ECHEVERRI, Mauricio Granada. (2014). "Solución del problema de múltiples agentes viajeros resuelto mediante técnicas heurísticas". Scientia et Technica, vol. 19, n.º 2, pp. 174–182. [En línea]. Disponible en: https://www.redalyc.org/pdf/849/84931680004.pdf. [Consulta: 15 de octubre de 2024].

[12] ANAYA-FUENTES, Gustavo. (2021). "Traveling Salesman Problem solved by clustering and genetic algorithms". [En línea]. Disponible en: https://repository.uaeh.edu.mx/revistas/index.php/icbi/article/download/7130/8045/. [Consulta: 15 de octubre de 2024].

[13] ZHAO, Xi y ZHU, Xiao-Ping. (2010). "Innovative genetic algorithm for solving GTSP". En: 2010 Second International Conference on Modeling, Simulation and Visualization Methods. IEEE, pp. 239–241. [En línea]. Disponible en: https://ieeexplore.ieee.org/abstract/document/5558311. [Consulta: 15 de octubre de 2024].

[14] CHEIKHROUHOU, Omar y KHOUFI, Ines. (2021). "A comprehensive survey on the Multiple Traveling Salesman Problem: Applications, approaches and taxonomy". Computer Science Review, vol. 40, p. 100369. [En línea]. Disponible en: https://www.sciencedirect.com/science/article/abs/pii/S1574013721000095. [Consulta: 15 de octubre de 2024].

[15] FOCACCI, Filippo; LODI, Andrea y MILANO, Michela. (2002). "A hybrid exact algorithm for the TSPTW". INFORMS Journal on Computing, vol. 14, n.º 4, pp. 403–417. [En línea]. Disponible en: https://pubsonline.informs.org/doi/abs/10.1287/ijoc.14.4.403.2827?casa_token=l4YwMwmPgiIAAAAA:jm-O6htnjVE5QTL8zsXs0qiOABk4XTErBO9fzgraMaYldXlc89dzTED80949Gzwfbp3r-j-uXEc. [Consulta: 15 de octubre de 2024].

[16] LOPEZ, Carlos Andres; MENDOZA, Jairo Alberto y CUARTAS, Ernesto. (2008). "Algoritmo para la exploración de todos los valores posibles en el problema del agente viajero (TPS)". Scientia et Technica, vol. 14, n.º 39, pp. 399–403. [En línea]. Disponible en: https://www.redalyc.org/pdf/849/84920503073.pdf. [Consulta: 1 de noviembre de 2024].

[17] BOYD, Stephen y MATTINGLEY, Jacob. (2007). "Branch and bound methods". Notes for EE364b, Stanford University, vol. 2006, p. 07. [En línea]. Disponible en: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2bf1aef9c02aedf869241f84d7c7250f1c3e5311. [Consulta: 1 de noviembre de 2024].

[18] PADBERG, Manfred y RINALDI, Giovanni. (1991). "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems". SIAM Review, vol. 33, n.º 1, pp. 60–100. [En línea]. Disponible en: https://www.jstor.org/stable/2030652. [Consulta: 1 de noviembre de 2024].

[19] ALEMAYEHU, Temesgen Seyoum y KIM, Jai-Hoon. (2017). "Efficient nearest neighbor heuristic TSP algorithms for reducing data acquisition latency of UAV relay WSN". Wireless Personal Communications, vol. 95, pp. 3271–3285. [En línea]. Disponible en: https://doi.org/10.1007/s11277-017-3994-9. [Consulta: 2 de noviembre de 2024].

[20] ROSENKRANTZ, Daniel J.; STEARNS, Richard E. y LEWIS, II, Philip M. (1977). "An analysis of several heuristics for the traveling salesman problem". SIAM Journal on Computing, vol. 6, n.º 3, pp. 563–581. [En línea]. Disponible en: https://disco.ethz.ch/courses/fs16/podc/readingAssignment/1.pdf. [Consulta: 3 de noviembre de 2024].

[21] GUTIN, Gregory y YEO, Anders. (2007). "The greedy algorithm for the symmetric TSP". Algorithmic Operations Research, vol. 2, n.º 1, pp. 33–36. [En línea]. Disponible en: https://www.erudit.org/en/journals/aor/2007-v2-n1-aor_2_1/aor2_1art04.pdf. [Consulta: 3 de noviembre de 2024].

[22] LEE, Sang-Un. (2012). "The Extended k-opt Algorithm for Traveling Salesman Problem". Journal of The Korea Society of Computer and Information, vol. 17, n.º 10, pp. 155–165. [En línea]. Disponible en: https://koreascience.kr/article/JAKO201233355899830.page. [Consulta: 7 de noviembre de 2024].

[23] BONROSTRO, Joaquín A. Pacheco y SERNA, Cristina R. Delgado. (2000). "Resultados de diferentes experiencias con búsqueda local aplicadas a problemas de rutas". Rect@: Revista Electrónica de Comunicaciones y Trabajos de ASEPUMA, vol. 2, n.º 1, pp. 53–81. [En línea]. Disponible en:

https://dialnet.unirioja.es/servlet/articulo?codigo=6473535. [Consulta: 23 de noviembre de 2024].

[24] AGUILAR, Lucero de Montserrat Ortiz, et al. (2015). "Comparativa de algoritmos bioinspirados aplicados al problema de calendarización de horarios". Research in Computing Science, (94), pp. 33–43. [En línea]. Disponible en: https://www.researchgate.net/profile/Claudia-Diaz-29/publication/323285737_Comparativa_de_algoritmos_bioinspirados_aplicados_al_problema_de_calendarizacion_de_horarios/links/5e98f057a6fdcca7892009d3/Comparativa-de-algoritmos-bioinspirados-aplicados-al-problema-de-calendarizacion-de-horarios.pdf. [Consulta: 27 de noviembre de 2024].

[25] YELMEWAD, Pramod y TALAWAR, Basavaraj. (2019). "Parallel iterative hill climbing algorithm to solve TSP on GPU". Concurrency and Computation: Practice and Experience, vol. 31, n.º 7, p. e4974. [En línea]. Disponible en: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4974?casa_token=QeMH_96aBbUAAAAA:t6ynA72782IpqfS1_WhJ-Tyfler2WnCOcRavd5w6rRogdd0mru2ygpUhtM-xJCqqH4IhyaW1nU8Z87mw. [Consulta: 23 de noviembre de 2024].

[26] GENG, Xiutang, et al. (2011). "Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search". Applied Soft Computing, vol. 11, n.º 4, pp. 3680–3689. [En línea]. Disponible en: https://www.sciencedirect.com/science/article/abs/pii/S1568494611000573. [Consulta: 24 de noviembre de 2024].

[27] LÓPEZ, Erasmo; SALAS, Óscar y MURILLO, Álex. (2014). "El problema del agente viajero: un algoritmo determinístico usando búsqueda tabú". Revista de Matemática: teoría y aplicaciones, vol. 21, n.º 1, pp. 127–144. [En línea]. Disponible en: https://www.redalyc.org/pdf/453/45331281008.pdf. [Consulta: 24 de noviembre de 2024].

[28] LIU, Yu-Hsin. (2007). "A hybrid scatter search for the probabilistic traveling salesman problem". Computers & Operations Research, vol. 34, n.º 10, pp. 2949–2963. [En línea]. Disponible en: https://www.sciencedirect.com/science/article/abs/pii/S0305054805003540. [Consulta: 24 de noviembre de 2024].

[29] MARINAKIS, Yannis y MARINAKI, Magdalene. (2010). "A hybrid multi-swarm particle swarm optimization algorithm for the probabilistic traveling salesman problem". Computers & Operations Research, vol. 37, n.º 3, pp. 432–442. [En línea]. Disponible en: https://www.sciencedirect.com/science/article/abs/pii/S0305054809000690. [Consulta: 27 de noviembre de 2024].

[30] FALCON HUALLPA, Elida. (2022). Mejora del rendimiento del algoritmo colonia de hormiga para resolver el problema de TSP. [En línea]. Disponible en: https://repositorio.unsaac.edu.pe/bitstream/handle/20.500.12918/7277/253T20221186.pdf?sequence=1&isAllowed=y. [Consulta: 27 de diciembre de 2024].

[31] MAGALLANES, Uriel Ervey Bernal, et al. (2016). Algoritmo genético y algoritmo de sistema de hormigas aplicados al problema del agente viajero. [En línea]. Disponible en: http://reaxion.utleon.edu.mx/Art_Algoritmo_genetico_y_algoritmo_de_sistema_de_hormigas_aplicados_al_problema_del_agente_viajero.html. [Consulta: 30 de noviembre de 2024].

[32] MEDINA, Josep R. y YEPES, Víctor. (2000). "Optimización de redes de distribución con algoritmos genéticos". En Actas del IV Congreso de Ingeniería del Transporte, pp. 205–213. [En línea]. Disponible en: https://personales.upv.es/vyepesp/00MYX07.pdf. [Consulta: 30 de noviembre de 2024].

[33] GESTAL POSE, Marcos. (s.f.). "Introducción a los Algoritmos Genéticos". Departamento de Tecnologías de la Información y las Comunicaciones, Universidade da Coruña. [En línea]. Disponible en: https://cursa.ihmc.us/rid=1KNKMJ4LN-11XXFSG-1KV5/Algoritmos%20de%20Terminos.pdf. [Consulta: 1 de diciembre de 2024].

[34] FARGIANA, Farid, et al. (2022). "Implementation of Cheapest Insertion Heuristic Algorithm in Determining Shortest Delivery Route". International Journal of Global Operations Research, vol. 3, n.º 2, pp. 37–45. [En línea]. Disponible en: https://iorajournal.org/index.php/ijgor/article/view/137. [Consulta: 30 de noviembre de 2024].

[35] MISHRA, A. y SHUKLA, A. (2017). "Analysis of the Effect of Elite Count on the Behavior of Genetic Algorithms: A Perspective". En 2017 IEEE 7th International Advance Computing Conference (IACC), Hyderabad, India, pp. 835–840. doi: 10.1109/IACC.2017.0172. [En línea]. Disponible en: https://ieeexplore.ieee.org/abstract/document/7976906?casa_token=gPbvenaag-oAAAAA:Pd7nyri3lBfBKPZBt7HbzTW44Sjy3UolM49EsXwk35AmxFT6l4rouXXfM44U7Ntdb_ZnGgNkM7o. [Consulta: 7 de diciembre de 2024].

[36] SOLOMON, M. M. (1984). "VRPTW Benchmark Problems and Solutions". [En línea]. Disponible en: http://web.cba.neu.edu/~msolomon/problems.htm. [Consulta: 11 de diciembre de 2024].

[37] PABICO, Jaderick P. y ALBACEA, Elizer A. (2015). "The Interactive Effects of Operators and Parameters to GA Performance Under Different Problem Sizes". [En línea]. Disponible en: https://arxiv.org/abs/1508.00097. [Consulta: 7 de diciembre de 2024].

[38] ARANGO, Jaime Antero, CASTRILLON, Omar Danilo y GIRALDO, Jaime Alberto. (2021). "Optimizing demand forecast parameters with an evolutionary algorithm". Universidad Nacional de Colombia, Facultad de Ingeniería y Arquitectura, Departamento de Ingeniería Industrial. [En línea]. Disponible en: https://www.laccei.org/LACCEI2021-VirtualEdition/full_papers/FP81.pdf. [Consulta: 14 de diciembre de 2024].

[39] BAKER, James Edward. (1989). An analysis of the effects of selection in genetic algorithms. Tesis doctoral. Vanderbilt University. ProQuest Dissertations & Theses, 8919677.

ANEXO

PSEUDOCÓDIGO PRIMITIVO ALGORITMO GENÉTICO:

```
Input: instance data, Psize, pmut
Output: pib
begin:
        Population:= Psize initial sequences
        pib:=Population[1]
        fitnessb=Fitness of pib
        Sb=Population[1]
        fit=[0 for j in range(psize)]
        for j=0 to Psize do
                fit[j]= Fitness of Population[j]
                if fit[j] <fitnessb then
                        pib=Population[j]
                        fitnessb=fit[j]
                        Sb=Population[j]
        Stop=0
        while stop<criterio parada do
                parent1=Sb
                parent2= random sequence selected from Population
                child= cruce parent1 y parent2
                if random(0;1) <= pmut
                        child2=mutate child
                        fitness= Fitness of child2
                else
                        fitness= Fitness of child
                        child2=child
                mal:=fitness
                cambio:=-1
                for j=0 to psize do
                        if fit[j]> mal
                                mal=fitj
                                cambio=j
                if cambio>=0
                        Population :=remove Population[cambio] and insert child2
                        fit[cambio]:=fitness
            if fitness < fitnessb
                    pib=child2
                    fitnessb =fitness
                    Sb=child2
            Stop=Stop+1
    return: pib
```