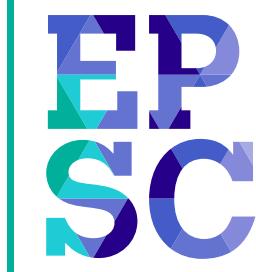




**ESCUELA POLITÉCNICA  
SUPERIOR DE CÓRDOBA**  
*Universidad de Córdoba*



**TRABAJO FIN DE GRADO**  
*Grado en Ingeniería Informática*  
**Estudio y comparación de implementaciones de  
metaheurísticas con ChatGPT**

**Autor:** Valentín Gabriel Avram Aenachioei  
**Directores:** Carlos García Martínez

Córdoba, Enero 2024



UNIVERSIDAD DE CÓRDOBA



# Resumen

Dentro de las ciencias de la computación, los problemas de optimización representan un desafío conocido por los altos costes relacionados a la búsqueda de soluciones efectivas. Para ello, las metaheurísticas han surgido como aproximaciones eficaces a la hora de abordar esta complejidad, ofreciendo enfoques flexibles para encontrar soluciones aproximadas en problemas difíciles de resolver de manera exacta.

En paralelo, la última gran revolución dentro de las ciencias de la computación han sido los modelos largos de lenguaje. Estos son capaces de simular una interpretación y comprensión del lenguaje natural.

Con estas capacidades, los modelos de lenguaje más avanzados, como ChatGPT [1] o LLaMA [2], son capaces de generar código de programación, tanto a partir código ya existente como generado de cero.

A partir de estos puntos, podemos combinar la capacidad de generación de código de los modelos largos de lenguaje con la capacidad de resolución de problemas por parte de las metaheurísticas. El objetivo principal del proyecto es estudiar la capacidad que tienen los modelos de lenguajes de reconocer, interpretar y generar implementaciones de algoritmos correctamente. Además, estas implementaciones se compararán entre sí, estudiando su rendimiento y analizando los resultados obtenidos por los algoritmos sobre problemas de prueba.

Para ello, se han seleccionado 24 algoritmos metaheurísticos distintos que han sido implementados por el modelo ChatGPT, usando el propio modelo largo de lenguaje para corregir el código generado. Estas implementaciones se han analizado, estudiando la viabilidad de estas comparadas con implementaciones teóricas del algoritmo. Las implementaciones obtenidas, sean correctas o no, se evalúan y comparan a partir de los resultados obtenidos sobre un conjunto de problemas de prueba.

Finalmente, los resultados de nuestro análisis experimental han demostrado la capacidad del modelo seleccionado, ChatGPT-3.5, de reconocer teóricamente algoritmos metaheurísticos concretos, además de sus características. También ha quedado patente la escasa capacidad de generación de código por parte del modelo, aún consiguiendo resultados óptimos por varios de los algoritmos estudiados.



# Abstract

In the field of computer science, optimisation problems represent a challenge known for the high costs associated with finding effective solutions. To this end, metaheuristics have emerged as effective approaches to deal with this complexity, offering flexible approaches to find approximate solutions to problems that are difficult to solve exactly.

In parallel, the latest major revolution in computer science has been large language models. These are capable of simulating both natural language interpretation and text understanding. With these capabilities, the most advanced language models, such as ChatGPT [1] or LLaMA [2], are able to generate programming code, both existing and completely new.

From these points, we can combine the code generation capability of large language models with the problem solving capability of metaheuristics. The main objective of the project is to study the ability of language models to recognise, interpret and generate implementations of algorithms correctly. Furthermore, these implementations will be compared with each other, studying their performance and analysing the results obtained by the algorithms on test problems.

For this purpose, 24 different metaheuristic algorithms have been selected and implemented using ChatGPT, using the language model itself to correct the generated code. These implementations have been analysed, studying their feasibility compared to theoretical implementations of the algorithm. The implementations obtained, whether correct or not, are evaluated and compared on the basis of the results obtained on a set of test problems.

At the end, the results of our experimental analysis have demonstrated the ability of the selected model, ChatGPT-3.5, to theoretically recognize specific metaheuristic algorithms, as well as their characteristics. It has also been demonstrated the low code generation ability of the model, even though it achieves optimal results for several of the algorithms studied.



# Índice general

Índice de tablas	III
Índice de figuras	V
Índice de pseudocódigos	VII
<b>1. Introducción</b>	<b>3</b>
1.1. Introducción al proyecto . . . . .	3
1.2. Definición del problema . . . . .	6
1.3. Trabajos y estudios relacionados . . . . .	8
1.4. Restricciones . . . . .	11
1.5. Recursos . . . . .	13
1.5.1. Recursos humanos . . . . .	13
1.5.2. Recursos de hardware . . . . .	13
1.5.3. Recursos de software . . . . .	14
<b>2. Objetivos</b>	<b>15</b>
2.1. Objetivos teóricos . . . . .	15
2.2. Objetivos prácticos . . . . .	16
<b>3. Antecedentes</b>	<b>19</b>
<b>4. Análisis</b>	<b>25</b>
4.1. Pruebas y evaluación . . . . .	25
4.2. Evaluación de resultados prácticos y tests estadísticos . . . . .	28
<b>5. Diseño</b>	<b>33</b>
<b>6. Resultados</b>	<b>37</b>
6.1. Implementaciones generadas . . . . .	41
6.1.1. Búsqueda aleatoria . . . . .	41

6.1.2.	Búsqueda local por primer vecino . . . . .	43
6.1.3.	Búsqueda local por mejor vecino . . . . .	46
6.1.4.	Búsqueda tabú . . . . .	48
6.1.5.	Enfriamiento simulado . . . . .	52
6.1.6.	Algoritmo de colonia de hormigas . . . . .	55
6.1.7.	Algoritmo de colonia de abejas . . . . .	59
6.1.8.	Optimización por enjambre de partículas . . . . .	63
6.1.9.	Búsqueda gravitatoria . . . . .	67
6.1.10.	Algoritmo de agujero negro . . . . .	70
6.1.11.	Búsqueda dispersa . . . . .	73
6.1.12.	Algoritmo cultural . . . . .	76
6.1.13.	Búsqueda con vecindario variable . . . . .	79
6.1.14.	Búsqueda armónica . . . . .	82
6.1.15.	Algoritmo imperialista competitivo . . . . .	85
6.1.16.	Algoritmo de optimización caótica . . . . .	88
6.1.17.	Sistema inmune artificial . . . . .	90
6.1.18.	Dinámica de formación de ríos . . . . .	93
6.1.19.	Algoritmo de luciérnagas . . . . .	95
6.1.20.	Algoritmo genético . . . . .	98
6.1.21.	Búsqueda local guiada . . . . .	101
6.1.22.	Algoritmo búsqueda de cuervo . . . . .	103
6.1.23.	Algoritmo de maleza invasora . . . . .	105
6.1.24.	Algoritmo del murciélagos . . . . .	107
6.1.25.	Evolución diferencial . . . . .	110
6.2.	Calidad de las implementaciones . . . . .	113
6.3.	Resultados experimentales . . . . .	118
<b>7.</b>	<b>Conclusiones</b>	<b>145</b>
7.1.	Posibles mejoras y cambios a futuro . . . . .	147
<b>8.</b>	<b>Manual de uso</b>	<b>149</b>
8.1.	Versión ejecutable . . . . .	150
8.2.	Proyecto completo . . . . .	152
<b>Bibliografía</b>		<b>153</b>



# Índice de tablas

6.1.	Tabla de implementaciones . . . . .	114
6.2.	Tabla de los rankings obtenidos tras 100 problemas aleatorios . . . . .	119
6.3.	Resultados tests de Wilcoxon . . . . .	136
6.4.	Resultados test de Friedman . . . . .	137
6.5.	Resultados test de Friedman manual . . . . .	138





# Índice de figuras

3.1.	Número posibles soluciones por tamaño del problema . . . . .	21
3.2.	Representación de ejemplo de Instancia TSP . . . . .	22
5.1.	Diagrama del proceso realizado . . . . .	33
5.2.	Ejemplo de uso de ChatGPT . . . . .	35
6.1.	Ejemplo rendimiento Búsqueda aleatoria . . . . .	43
6.2.	Ejemplo rendimiento Búsqueda local primer vecino . . . . .	45
6.3.	Ejemplo rendimiento Búsqueda local mejor vecino . . . . .	48
6.4.	Ejemplo rendimiento Búsqueda tabú . . . . .	51
6.5.	Ejemplo rendimiento Enfriamiento simulado . . . . .	55
6.6.	Ejemplo rendimiento Algoritmo colonia hormigas . . . . .	58
6.7.	Ejemplo rendimiento Algoritmo colonia abejas . . . . .	63
6.8.	Ejemplo rendimiento Algoritmo enjambre partículas . . . . .	66
6.9.	Ejemplo rendimiento Búsqueda gravitacional . . . . .	69
6.10.	Ejemplo rendimiento Algoritmo agujero negro . . . . .	72
6.11.	Ejemplo rendimiento Búsqueda dispersa . . . . .	75
6.12.	Ejemplo rendimiento Algoritmo cultural . . . . .	78
6.13.	Ejemplo rendimiento Búsqueda vecindario variable . . . . .	81
6.14.	Ejemplo rendimiento Búsqueda armónica . . . . .	84
6.15.	Ejemplo rendimiento Algoritmo imperialista competitivo . . . . .	87
6.16.	Ejemplo rendimiento Algoritmo optimización caótica . . . . .	89
6.17.	Ejemplo rendimiento Algoritmo sistema inmune artificial . . . . .	92
6.18.	Ejemplo rendimiento dinámica formación ríos . . . . .	95
6.19.	Ejemplo rendimiento Algoritmo luciérnaga . . . . .	97
6.20.	Ejemplo rendimiento Algoritmo genético . . . . .	100
6.21.	Ejemplo rendimiento Búsqueda local guiada . . . . .	102
6.22.	Ejemplo rendimiento búsqueda cuervo . . . . .	104
6.23.	Ejemplo rendimiento Algoritmo maleza invasora . . . . .	107
6.24.	Ejemplo rendimiento Búsqueda murciélagos . . . . .	110
6.25.	Ejemplo rendimiento Evolución diferencial . . . . .	113

6.26. Ranking rendimiento de los 25 algoritmos . . . . .	123
6.27. Gráfica convergencia de los 25 algoritmos . . . . .	124
6.28. Ranking rendimiento algoritmos naturales . . . . .	125
6.29. Gráfica convergencia algoritmos naturales . . . . .	126
6.30. Ranking rendimiento algoritmos poblaciones . . . . .	127
6.31. Gráfica convergencia algoritmos poblaciones . . . . .	128
6.32. Ranking rendimiento algoritmos búsqueda local . . . . .	129
6.33. Gráfica convergencia algoritmos búsqueda local . . . . .	130
6.34. Ranking rendimiento algoritmos evolutivos . . . . .	131
6.35. Gráfica convergencia algoritmos evolutivos . . . . .	132
6.36. Gráfica convergencia 5 peores algoritmos . . . . .	133
6.37. Ranking rendimiento 5 mejores algoritmos . . . . .	134
6.38. Gráfica convergencia 5 mejores algoritmos . . . . .	135
6.39. Resultados Nemenyi algoritmos naturales . . . . .	139
6.40. Resultados Nemenyi algoritmos poblacionales . . . . .	140
6.41. Resultados Nemenyi algoritmos trayectorias . . . . .	141
6.42. Resultados Nemenyi algoritmos evolutivos . . . . .	142
6.43. Resultados Nemenyi peores . . . . .	143
6.44. Resultados Nemenyi mejores algoritmos . . . . .	143
8.1. Ejemplo de archivo de configuración . . . . .	151



# Índice de pseudocódigos

1.	Búsqueda aleatoria . . . . .	42
2.	Búsqueda local por primer vecino . . . . .	44
3.	Búsqueda local por mejor vecino . . . . .	46
4.	Búsqueda tabú . . . . .	50
5.	Enfriamiento simulado . . . . .	53
6.	Colonia de hormigas . . . . .	57
7.	Colonia de abejas . . . . .	61
8.	Optimización por enjambre de partículas . . . . .	64
9.	Búsqueda gravitacional . . . . .	67
10.	Algoritmo agujero negro . . . . .	71
11.	Búsqueda dispersa . . . . .	73
12.	Algoritmo cultural . . . . .	77
13.	Búsqueda con vecindario variable . . . . .	80
14.	Búsqueda armónica . . . . .	83
15.	Algoritmo imperialista competitivo . . . . .	86
16.	Algoritmo de optimización caos . . . . .	88
17.	Sistema inmune artificial . . . . .	91
18.	Dinámica de formación de ríos . . . . .	94
19.	Algoritmo de luciérnagas . . . . .	96
20.	Algoritmo genético . . . . .	99
21.	Búsqueda local guiada . . . . .	101
22.	Algoritmo búsqueda de cuervo . . . . .	103
23.	Algoritmo de maleza invasora . . . . .	105
24.	Algoritmo de murciélagos . . . . .	108
25.	Algoritmo Evolución diferencial . . . . .	111



UNIVERSIDAD DE CÓRDOBA

---

## ÍNDICE DE PSEUDCÓDIGOS

---





# Capítulo 1

## Introducción

### 1.1. Introducción al proyecto

Las ciencias de la computación desempeñan un papel crucial en la resolución de una amplia gama de problemas complejos, los cuales rozan la imposibilidad a la hora de ser abordados manualmente. Estos problemas van desde la optimización de rutas logísticas hasta la asignación óptima de recursos en entornos dinámicos.

Desde la resolución de problemas de programación combinatoria hasta la creación de algoritmos de búsqueda avanzados, las ciencias de la computación, respaldadas por metaheurísticas, abordan problemas que son inherentemente difíciles de resolver mediante métodos tradicionales, ofreciendo enfoques innovadores y soluciones adaptativas ante escenarios cambiantes y complejos.

Para afrontar estos problemas, las metaheurísticas surgen como una respuesta inteligente a la complejidad de resolución de problemas computacionales difíciles. Estos algoritmos, inspirados en procesos naturales y estrategias heurísticas, buscan encontrar soluciones aproximadas para problemas complejos que podrían tener múltiples soluciones óptimas.

Funcionan explorando y explotando el espacio de búsqueda de posibles soluciones, variando entre diversificación e intensificación de las áreas exploradas del espacio de búsqueda [3]. Estas técnicas, al operar con un enfoque de “aprendizaje adaptativo”, ajustan sus estrategias según la información obtenida durante la búsqueda, ofreciendo flexibilidad y capacidad para encontrar soluciones aceptables en tiempos razonables.



Las ventajas de las metaheurísticas son notables: su capacidad para encontrar soluciones cercanas a óptimas en problemas complejos, su adaptabilidad a diferentes tipos de problemas y su eficiencia en términos de tiempo computacional. Sin embargo, presentan ciertos inconvenientes, ya que no garantizan la convergencia a la solución óptima en todos los casos, son sensibles a la configuración de hiperparámetros y se necesitan adaptar los operadores y estrategias para cada problema específico [4].

A pesar de estas limitaciones, el potencial de las metaheurísticas en la resolución de problemas complejos ha llevado a su aplicación en diversas áreas, desde la optimización de procesos industriales hasta la toma de decisiones en contextos logísticos y financieros.

Por otro lado, los modelos largos de lenguaje han representado una innovación extraordinaria en el campo de la inteligencia artificial y la computación. Estos modelos han revolucionado la comprensión del lenguaje natural al aprender patrones complejos y estructuras lingüísticas a partir de enormes conjuntos de datos textuales.

Su capacidad para generar texto coherente y contextualmente relevante ha trascendido la mera comprensión, permitiendo incluso la generación de código. Los modelos de lenguaje pueden interpretar solicitudes en lenguaje natural y traducirlas en código de programación funcional, ofreciendo un potencial revolucionario en el desarrollo de software y la automatización de tareas de programación, aunque con desafíos en la garantía de la precisión y seguridad en el código producido.

Así, podemos agrupar estos dos conceptos para realizar nuestro estudio. La convergencia entre el potencial de generación de código de los modelos largos de lenguaje y la implementación de metaheurísticas presenta un campo de estudio prometedor.

Este enfoque innovador busca explorar y evaluar la capacidad de los modelos largos de lenguaje para comprender, interpretar y generar implementaciones de metaheurísticas.

El objetivo principal de esta investigación es analizar exhaustivamente la habilidad de estos modelos para capturar la lógica intrínseca de las metaheurísticas y transformarla en código funcional.



La evaluación se centrará en la precisión, la coherencia y la eficacia de las implementaciones generadas, permitiendo comprender hasta qué punto los modelos largos de lenguaje pueden no solo entender sino también aplicar conceptos algorítmicos complejos, como estrategias de búsqueda global, optimización de soluciones y adaptabilidad a diferentes contextos de problemas.

Por último, la comparación de implementaciones sobre problemas clásicos como el Problema del Viajante de Comercio [5] (TSP por sus siglas en inglés) nos ofrece una última perspectiva para evaluar la eficiencia y la efectividad de las implementaciones generadas.

Al confrontar las implementaciones, se puede analizar la calidad de las soluciones, el tiempo computacional requerido para llegar a ellas y la capacidad de adaptación a diferentes instancias del problema.

Esta comparativa permitirá discernir el desempeño relativo de todas las aproximaciones, brindando información crucial sobre el potencial de los modelos de lenguaje en la resolución de problemas complejos de optimización combinatoria como el TSP.

Este estudio podría revelar no sólo la capacidad de los modelos de lenguaje para interpretar conceptos algorítmicos avanzados, sino también su potencial para ser utilizados como herramientas de generación y optimización de código en dominios complejos y especializados como las metaheurísticas.



## 1.2. Definición del problema

El enfoque central de este proyecto de Fin de Grado radica en la evaluación minuciosa de los algoritmos metaheurísticos implementados por modelos largos de lenguaje desde dos perspectivas esenciales, su accesibilidad en términos de programación y su capacidad para ofrecer soluciones competitivas en problemas complejos de optimización.

Las metaheurísticas plantean el interrogante sobre si su implementación es realmente asequible en entornos de programación y si sus resultados son equiparables o superiores a los obtenidos por enfoques tradicionales. Este estudio se adentra en esta indagación mediante la utilización de modelos largos de lenguaje, en particular el modelo ChatGPT-3.5, como instrumento fundamental para comprobar esta hipótesis. El modelo ChatGPT-3.5, el modelo largo de lenguaje gratuito de OpenAI [6], será empleado para generar implementaciones de metaheurísticas, evaluando posteriormente su rendimiento en la resolución de problemas complejos de optimización.

En este contexto, se plantea la cuestión crucial sobre si las metaheurísticas, conocidas por su potencial en la resolución de problemas complejos, son realmente accesibles desde la perspectiva de la programación, es decir, si su implementación es sencilla y práctica para desarrolladores y programadores.

Además, se busca indagar si estas implementaciones, ofrecen resultados competitivos en la resolución de problemas específicos de optimización. La evaluación exhaustiva de este interrogante se llevará a cabo mediante la generación y análisis de implementaciones de metaheurísticas a través del ChatGPT-3.5, evaluándolas y obteniendo conclusiones significativas sobre la viabilidad y eficacia de las metaheurísticas en la resolución de problemas desafiantes.



La implementación de metaheurísticas conlleva un conjunto diverso de desafíos y complejidades inherentes. Estas técnicas algorítmicas, aunque flexibles y poderosas en la resolución de problemas complejos, requieren un profundo entendimiento de los principios subyacentes de cada metaheurística específica, así como de la capacidad de adaptar y ajustar sus parámetros y operadores para abordar eficazmente diferentes problemas.

La configuración de una metaheurística para un problema particular puede ser un proceso no trivial, donde la selección adecuada de estrategias de búsqueda, la optimización de funciones objetivo y la gestión del equilibrio entre exploración y explotación del espacio de soluciones son aspectos críticos [7]. Esta complejidad se ve agravada por la necesidad de comprender la dinámica de cada problema en particular y adaptar las estrategias de las metaheurísticas para obtener soluciones efectivas y eficientes.

El estudio se enfocará en la generación de implementaciones de metaheurísticas específicamente diseñadas para abordar el Problema del viajante de comercio (TSP), un problema emblemático en la literatura de optimización combinatoria [8]. El TSP, dada su complejidad y relevancia en la optimización de rutas y logística, proporciona un escenario idóneo para evaluar la capacidad de las metaheurísticas generadas por el modelo ChatGPT-3.5 en la búsqueda de soluciones óptimas.

A través de esta evaluación se buscará analizar el rendimiento, la eficiencia y la calidad de las soluciones proporcionadas por las metaheurísticas generadas usando modelos largos de lenguaje en la resolución de instancias específicas del TSP.

El Problema del viajante de comercio se establece como el problema sobre el cual se evaluarán las implementaciones generadas. Ampliamente estudiado en la literatura de optimización combinatoria, consiste en encontrar la ruta más corta que visite un conjunto de ciudades exactamente una vez y regrese al punto de partida. Dada su naturaleza NP-compleja, el TSP ha sido un desafío significativo para los investigadores en áreas de optimización, algoritmos y ciencias de la computación.

La amplia literatura [9] disponible ofrece diversas estrategias y enfoques para abordar este problema, desde algoritmos exactos hasta métodos heurísticos y metaheurísticas, evidenciando su complejidad y relevancia en el campo de la optimización combinatoria. Esta abundante información será fundamental para contextualizar y comparar las implementaciones generadas por el modelo ChatGPT-3.5 en este estudio.



### 1.3. Trabajos y estudios relacionados

Para un estudio de estas características, es necesario trabajar y estudiar una amplia bibliografía previa. Para estudiar nuestras hipótesis se necesita comprender los modelos largos de lenguaje, y en específico, el cómo se interpretan las entradas, cómo se generan respuestas, y sobre todo, cómo es capaz de generar código de programación funcional.

Por otro lado, para analizar las implementaciones generadas por el modelo es necesario entender qué se busca en cada implementación. Por ello, para cada metaheurística distinta se debe de explorar la bibliografía respectiva a cada algoritmo, comprendiendo su funcionamiento y elementos característicos.

Aunque los transformadores, tecnología que sirve de base para los modelos de GPT [10], se origina en los años 90, el primer gran Modelo Largo de Lenguaje se data a mediados del 2018 con GPT-1, de OpenAI [11][12]. A partir de aquí, distintos modelos y aplicaciones se han desarrollado hasta llegar al GPT-4 actual, o el GPT-3.5 usado en este estudio, pero seguimos enfrentándonos a una tecnología relativamente nueva. Aunque es una tecnología de moda, y se está estudiando en prácticamente cada área, sigue habiendo mucho campo por explorar [13].

En primer lugar, un estudio interesante sobre la “capacidad de comprensión” del modelo GPT-3.5 sería el estudio[14], realizado por investigadores españoles, en el que se demuestra que los modelos de GPT son muy dependientes del conjunto de datos de entrenamiento, pues el modelo evaluado no es capaz de comprender tecnicismos, jerga específica, o responder correctamente ante conocimiento desconocido, dando como correcta información que desconoce.

Entrando en el área de la programación, tenemos algunos estudios sobre el uso de estos modelos para la generación de código. En primer lugar el estudio [15] muestra unos cortos ejemplos de generación de pequeñas porciones de código, sobre todo enfocado al lenguaje HTML. Para una evaluación más compleja con respecto al código, el estudio [16] trata varios modelos pre-entrenados de GPT, haciendo un énfasis en las entradas, distintos lenguajes de programación y distintos modelos.

Una de las varias conclusiones de este estudio es que, dependiendo de las condiciones del modelo y el código a generar, estos modelos siguen teniendo una gran tasa de fallos a la hora de generar código funcional, aún tratando con modelos pre-entrenados con grandes repositorios de código.



Por último, lo más cercano a nuestro estudio podría ser [17], en la que se intenta generar el código en R para una sola metaheurística, la búsqueda Tabú. Este pequeño artículo, que no puede considerarse un estudio en profundidad, muestra que GPT es capaz de generar una implementación prácticamente correcta, con fallos menores. Aún así, solo se genera un solo algoritmo bastante conocido, definiendo previamente el problema y el algoritmo en profundidad.

Podemos afirmar que nuestro estudio es pionero al evaluar la capacidad de un Modelo Largo de Lenguaje de generar implementaciones de algoritmos complejos y menos conocidos.

En cuanto a metaheurísticas, el panorama es el contrario. Representan un campo de estudio sólidamente establecido y ampliamente investigado en la comunidad científica.

Esta disciplina se ha convertido en un pilar casi fundamental en la resolución de problemas complejos de optimización, ofreciendo una gama diversa de técnicas algorítmicas que permiten abordar problemas difíciles en diversas áreas.

Desde algoritmos genéticos hasta optimización por enjambre de partículas, enfriamiento simulado y búsqueda tabú, el campo de las metaheurísticas ha experimentado un desarrollo continuo, respaldado por una vasta bibliografía que incluye investigaciones teóricas, estudios empíricos y comparativos y aplicaciones prácticas en diversos dominios.

Para nuestro estudio, podemos partir de los artículos [9][18], los cuales ofrecen un exhaustivo panorama sobre una amplia gama de metaheurísticas. Estos estudios recopilan una extensa cantidad de técnicas, explorando tanto las más consolidadas como aquellas relativamente nuevas, abordando diferentes problemas de optimización.

En estos trabajos, se evidencia una clara preponderancia de algunas metaheurísticas ampliamente estudiadas, como algoritmos genéticos, recocido simulado y optimización por enjambre de partículas, las cuales han sido objeto de numerosos análisis y aplicaciones en diversos campos. Además, se destacan otras familias de metaheurísticas que, aunque menos frecuentemente estudiadas en ciertos contextos, muestran potencial y relevancia en el abordaje de problemas específicos.



Estos artículos también proporcionan una clasificación detallada, categorizando las metaheurísticas en grupos según sus enfoques fundamentales, desde algoritmos bio-inspirados hasta técnicas de optimización basadas en la física, permitiendo una comprensión profunda de las distintas corrientes y corredores de investigación en el campo de las metaheurísticas.

A partir de estos dos artículos, podemos desarrollar un amplio listado de metaheurísticas distintas y variadas para poder implementar con el modelo largo de lenguaje. Una vez seleccionada una serie de algoritmos a desarrollar, podemos explorar la bibliografía de cada algoritmo específico, estudiando su desarrollo y funcionamiento específico.

Para cada algoritmo, se estudiará bibliografía previa necesaria para comprender el funcionamiento y las implementaciones esperables.



## 1.4. Restricciones

Al tratarse de un trabajo de fin de grado, este proyecto estará limitado, tanto en el análisis teórico como en las pruebas y desarrollo práctico.

En cuanto al desarrollo teórico y estrategias a seguir a la hora de decidir cómo plantear la problemática y cómo tratarla, hay varios aspectos a tratar:

- **Número de algoritmos:** Para este estudio, se han usado exactamente 25 algoritmos, 24 metaheurísticas y un algoritmo sencillo. Este número es suficiente para tener una variedad de algoritmos, con distintos enfoques y distintos planteamientos de funcionamiento. Aunque este número podría extenderse se limita por tiempo, tanto tiempo de desarrollo como tiempo de pruebas.
- **Limitaciones humanas:** Este proyecto está limitado por la normativa de la Universidad de Córdoba respecto a Trabajos de Fin de Grado. Se estipula que el trabajo debe ser realizado por un único estudiante, con ayuda y guía de su tutor. Esto estipula que el trabajo realizado debe ser original, citando todo el contenido de terceros.
- **Limitaciones económicas:** Al no ser necesario una inversión económica para realizar un estudio de estas características, se limita el software y hardware a elegir versiones y opciones de uso gratuito.
- **Limitaciones de software:** Por limitaciones económicas y de tiempo, solo se usarán implementaciones generadas por ChatGPT-3.5, versión gratuita. No se usarán los modelos de lenguaje de Bing [19], Bard [20] o versiones de pago de ChatGPT [21].



Además de los aspectos teóricos, el proyecto se limita también en su aspecto práctico:

- **Límite de tiempo:** Para poder cumplir con los límites de tiempos en el desarrollo de un Trabajo de Fin de Grado, el número de pruebas y el tiempo de estas estará limitado.
- **Límite en los problemas:** Para poder cumplir con los límites de tiempo anteriormente impuestos, los propios problemas sobre los que evaluar los algoritmos estarán limitados en tamaño, de forma que se puedan obtener soluciones óptimas en los tiempos establecidos.
- **Límite en la evaluación:** Por la naturaleza de los algoritmos y de los problemas sobre los que se evalúan, tanto en número de problemas como en tamaño, la posterior evaluación de los algoritmos está limitada. Por las características de los problemas, no se pueden usar test paramétricos para evaluar el rendimiento. Esto limita la evaluación a tests no paramétricos y a la comparación por el rendimiento en los resultados.



## 1.5. Recursos

### 1.5.1. Recursos humanos

Autor del proyecto:

- Valentín Gabriel Avram Aenachioei, estudiante de Ingeniería Informática, mención en Computación en la Escuela Politécnica Superior de Córdoba

Director del proyecto:

- Carlos García Martínez, profesor titular del departamento de Informática y Análisis Numérico en la Escuela Politécnica Superior de Córdoba

### 1.5.2. Recursos de hardware

Todas las pruebas se ejecutan en una máquina del departamento de Informática y Análisis Numérico. Los recursos de hardware usados para el desarrollo del código y las implementaciones no se indicarán, pues no es influyente. Los recursos de la máquina usada para las pruebas son:

- Procesador: Intel Core i7, 8 núcleos a 2.8 GHz
- Memoria RAM: 21 GB
- Sistema Operativo: Ubuntu 22.04



### 1.5.3. Recursos de software

Para el entorno de desarrollo y programación se ha usado:

- Python 3.12 [22]
- PyCharm Community Edition 2023.1.2 [23]

Dentro del entorno de desarrollo en Python, se han usado varias librerías específicas:

- Numpy 1.26.3 [24]
- Pandas 2.1.4 [25]
- Scipy 1.11.4 [26]
- Matplotlib 3.8.2 [27]
- Orange3 3.30 [28]

Como lenguaje de modelo usado para desarrollar las implementaciones de los algoritmos se ha usado:

- ChatGPT-3.5 [29]

Por último, para el desarrollo del documento se ha usado:

- Overleaf, editor de LaTeX en línea [30]

# Capítulo 2

## Objetivos

Los objetivos de este proyecto o estudio se pueden dividir en dos subgrupos relacionados.

Por un lado tenemos evaluaciones teóricas, estudiando las implementaciones metaheurísticas y las capacidades del modelo largo de lenguaje para implementar estas. Por otro lado se busca un análisis más práctico, comparando y estudiando el rendimiento de los algoritmos obtenidos.

Los objetivos, con sus respectivos sub-objetivos, se pueden listar como:

### 2.1. Objetivos teóricos

- Estudio previo de cada metaheurística a desarrollar.
- Generación de implementaciones usando ChatGPT, sin instrucciones específicas.
- Corrección de errores usando ChatGPT. Se pretenden conseguir algoritmos funcionales sin interacción humana.
- Evaluación teórica de las implementaciones. Se busca comprobar que cumplan los elementos y operaciones característicos del algoritmo.



## 2.2. Objetivos prácticos

- Evaluación de cada algoritmo sobre el conjunto de problemas.
- Comparativa de los resultados de los algoritmos sobre el conjunto de problemas
- Comparativa mediante tests no paramétricos

En primer lugar, el proyecto debe comenzar con un estudio en profundidad de cada metaheurística a implementar. Partiendo de estudios y artículos previos, se busca comprender el funcionamiento y operaciones característicos del algoritmo a desarrollar. A partir de este estudio previo se propone una exploración exhaustiva del entendimiento de ChatGPT sobre algoritmos metaheurísticos, y por ende, estudiar la complejidad de las implementaciones de dichos algoritmos. Para lograrlo, se realizará un análisis detallado de las capacidades de ChatGPT para discernir y comprender las complejidades de cada algoritmo.

Además, se pretende evaluar la profundidad de su conocimiento, examinando su capacidad para identificar los principios fundamentales, las estrategias de optimización y las limitaciones de cada enfoque metaheurístico.

En este caso, no se pretende evaluar la capacidad de generación de respuestas fuera de los datos de entrenamiento del modelo, pues para generar nuestras implementaciones no se darán detalles de las implementaciones a generar. Todo el código generado será a partir del propio conocimiento previo del modelo.

Por otro lado, se busca llevar a cabo una investigación exhaustiva sobre la habilidad de ChatGPT para generar implementaciones prácticas de estos algoritmos. Una vez reconocido cada algoritmo y sus elementos característicos, queremos comprobar si el modelo es capaz generar su implementación, codificar sus elementos característicos, generar soluciones válidas e ir mejorando estas soluciones a lo largo de las iteraciones.

La clave de esto es poder conseguir implementaciones de código totalmente funcionales sin entrar en detalles técnicos, solamente indicando el algoritmo que queremos desarrollar y el tipo de soluciones a generar.

Por último en cuanto a código, se pretende evaluar la capacidad de ChatGPT para identificar y rectificar errores inherentes a la implementación de algoritmos de metaheurísticas sin una indicación explícita de la ubicación o naturaleza de los errores presentes en el código.



Se busca entender si el modelo largo de lenguaje puede detectar anomalías en el código, diagnosticar posibles puntos problemáticos y ofrecer soluciones correctivas de manera autónoma. Esto implica una evaluación minuciosa de la habilidad de ChatGPT para interpretar contextos algorítmicos, inferir posibles fallas y proponer correcciones precisas sin una guía específica sobre la ubicación o naturaleza de los errores presentes en la implementación.

Este enfoque no solo busca determinar el nivel de conocimiento teórico de ChatGPT sobre las metaheurísticas, sino también explorar su utilidad práctica como asistente en el desarrollo efectivo de algoritmos metaheurísticos.

Una vez evaluada la calidad de las implementaciones, se pretende llevar a cabo pruebas exhaustivas con las diversas metaheurísticas aplicadas a un tipo de problema específico. Se busca evaluar la eficacia y el rendimiento de estas metaheurísticas en la resolución de problemas particulares, analizando su capacidad para encontrar soluciones óptimas o cercanas a óptimas y la capacidad de mejorar sus soluciones con el paso de las iteraciones.

Estas pruebas permitirán no solo comparar el desempeño relativo de cada técnica metaheurística, sino también analizar cómo influye el impacto de las implementaciones, con sus mínimos cambios y posibles errores, en el rendimiento, comprobando si esta forma de generar código es realmente útil y eficiente.

Queremos comprobar si estos algoritmos son funcionales, si generan soluciones validas, si estas mejoran con el paso de las iteraciones, y si estas llegan a ser óptimas, o se acercan a la solución óptima. Además, se quiere comprobar lo competitivos que son los algoritmos comparados entre sí y comprobar si realmente es mejor usar un algoritmo sub-óptimo bien optimizado, o el algoritmo ideal mal implementado.

Una pregunta que buscamos resolver con este estudio es:  
¿Puede un usuario sin conocimiento previo resolver problemas complejos con código generado por un modelo largo de lenguaje?



UNIVERSIDAD DE CÓRDOBA

## Objetivos

---



# Capítulo 3

## Antecedentes

La comparación de varias metaheurísticas y diferentes implementaciones en el contexto del No Free Lunch Theorem (NFL) representa un desafío significativo y esencial en la investigación en optimización algorítmica.

El teorema No Free Lunch [31] sostiene que no existe un algoritmo universalmente superior para todos los problemas, puesto que para un número ilimitado de problemas, cada algoritmo tendrá un rendimiento único para cada problema. Así, los algoritmos superiores en algunos problemas, serán peores en muchos otros.

Esto implica que cualquier mejora en el rendimiento de un algoritmo en un problema específico podría acompañarse de un empeoramiento en otro. En este marco, la evaluación comparativa de diversas metaheurísticas se convierte en un instrumento crucial para entender cómo estas técnicas se desempeñan en diferentes tipos de problemas y contextos.

Es fundamental destacar que la elección y parametrización de una metaheurística pueden influir significativamente en su rendimiento. Por lo tanto, la comparación debe abordar no sólo la diversidad de problemas, sino que en nuestro estudio se centrará en la variabilidad de las implementaciones y el cómo estas afectan al rendimiento. Además, la variabilidad en las implementaciones, incluyendo la selección de operadores y estrategias específicas, permite analizar cómo ajustes particulares pueden afectar el desempeño relativo de cada metaheurística.



La relación con el teorema No Free Lunch se manifiesta en la necesidad de reconocer que no hay un enfoque único y universalmente superior. Al comparar diversas metaheurísticas, se evidencia la importancia de entender las características del problema en cuestión y adaptar estratégicamente las técnicas algorítmicas para maximizar su eficacia. Este estudio puede contribuir a la comprensión más profunda de cómo diferentes metaheurísticas se especializan y destacan en ciertos problemas, arrojando luz sobre las complejidades que rodean la optimización algorítmica y su relación con las limitaciones teóricas impuestas por el NFL.

Por otro lado, para entender cómo se van a evaluar los algoritmos, se debe entender el problema sobre el que serán probados, el problema del **Viajante de Comercio**.

El problema del Viajante de comercio [5] o TSP por sus siglas en inglés (Travelling Salesperson Problem), es uno de los problemas de optimización más conocidos en el campo de las ciencias de la computación. Se puede definir de la siguiente manera:

Dado un conjunto de  $N$  ciudades y conociendo la distancia entre los pares de ciudades, el objetivo es encontrar el camino que recorra todas las ciudades, sin pasar más de una vez por una misma ciudad y volviendo a la ciudad de origen.

Con  $N$  ciudades, tenemos un total de  $N!$  caminos posibles. Sabiendo que debemos volver al punto de inicio, los posibles caminos se reducen a  $(N-1)!$  posibles caminos.

A la hora de tratar el TSP existen distintos enfoques [32][33], siendo los más clásicos los algoritmos exactos, que garantizan una solución óptima a un alto coste de ejecución en términos de tiempo. Un ejemplo pueden ser la fuerza bruta[34] o algoritmos de ramificación y poda [35]. El problema de estos algoritmos es que son inasequibles a partir de un número moderado de ciudades.

Por ejemplo, trabajando sobre  $N = 10$  ciudades, el algoritmo de fuerza bruta garantiza la solución óptima tras evaluar un total de  $(N - 1)! = 362,880$ . Elevando el número de ciudades a  $N = 14$ , el número de evaluaciones sube a  $(N - 1)! = 6,227021 \cdot 10^9$ . La magnitud de este problema puede apreciarse en la figura 3.1.

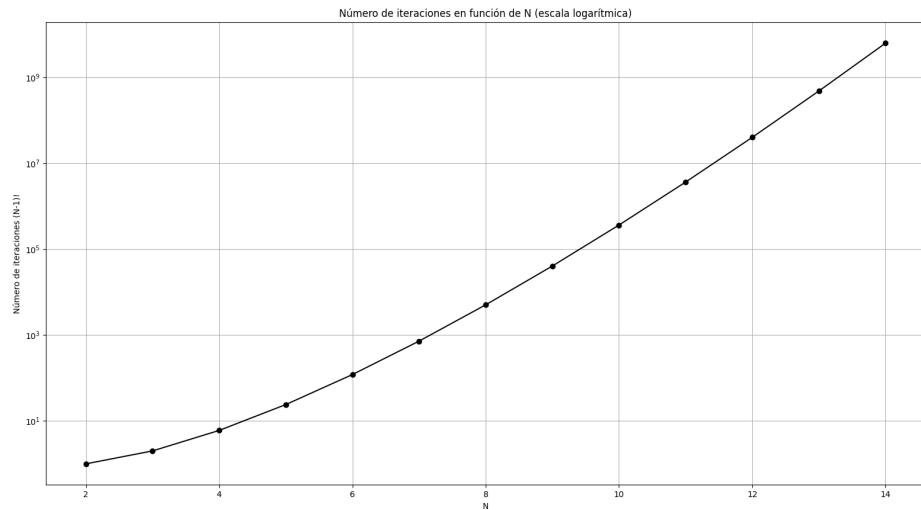


Figura 3.1: Número posibles soluciones por tamaño del problema

Teniendo en cuenta que estos problemas suelen evaluarse sobre valores de cientos o miles de ciudades, estos algoritmos se hacen inviables.

Otro enfoque muy usado para tratar este problema son los algoritmos heurísticos, como el algoritmo del Vecino más próximo [36], u otros algoritmos aproximados [37][38] garantizan buenas soluciones en un tiempo razonable, pero no garantizan obtener la mejor solución posible. A partir de los algoritmos heurísticos podemos desarrollar los enfoques metaheurísticos que estudiaremos en este proyecto.

El problema del viajante de comercio es “ícono” por ser un problema con amplios usos en diferentes aplicaciones prácticas. Además, es un problema NP-Hard [39], lo que significa que aún no se conoce un algoritmo eficiente, es decir, que devuelva la solución óptima en tiempo polinómico. Siendo un problema NP-Hard, es idóneo para poder estudiar algoritmos sobre este, y comprobar cuán útil puede ser un algoritmo sobre un número arbitrario de problemas.

En nuestro proyecto, para poder evaluar los algoritmos sobre ejemplos de estos problemas se han usado dos elementos principales, las Instancias de TSP y el Generador de Experimentos. Se debe recalcar que ambas porciones de código son adaptaciones del código original escrito y desarrollado por Carlos García Martínez [40], director de este proyecto. Partiendo de su código, este se ha adaptado y modificado para las necesidades de este estudio, creando un entorno de pruebas original a partir de este.

El elemento principal es la **Instancia de TSP**, que se encarga de generar un problema TSP. Esta clase de Python desarrolla los elementos clave del propio problema partiendo de un número exacto de ciudades.

Se generan las ciudades del problema, cada una con unas coordenadas aleatorias y se desarrolla la función de evaluación, la cual a partir de una permutación de números enteros devuelve el coste de ese recorrido, partiendo de las posiciones de esas ciudades. Un ejemplo de problema, con sus respectivas ciudades generadas aleatoriamente y un recorrido que representa una solución se puede apreciar en la figura 3.2.

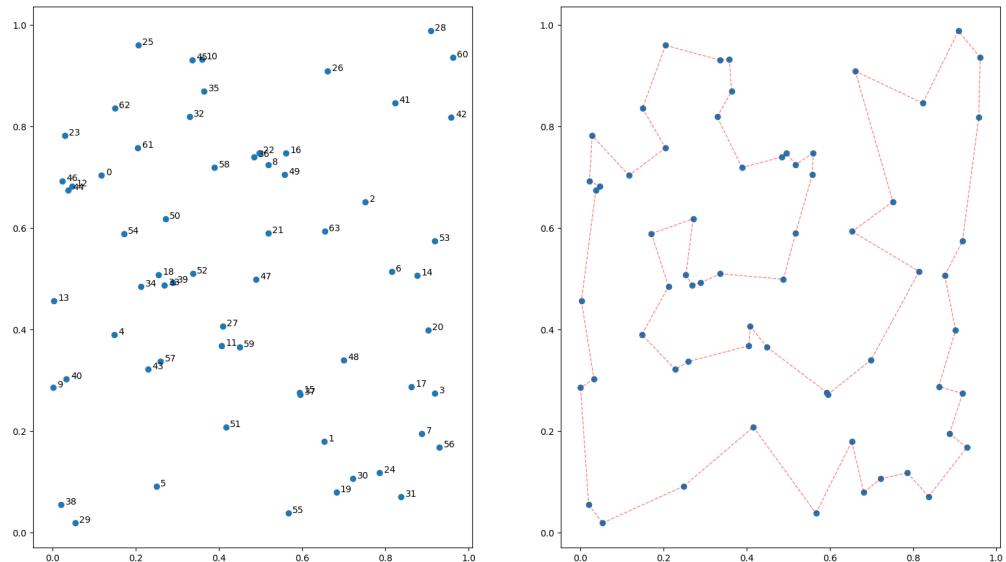


Figura 3.2: Representación de ejemplo de Instancia TSP



Además, se tienen funciones para representar gráficamente los “mapas” que representan cada problema generado y el recorrido que se ha obtenido como solución.

A partir de esas Instancias de TSP se trabaja con el **Generador de Experimentos**. Con todos los problemas ya instanciados se genera una elemento de Experimento con los problemas aleatoriamente generados. Este almacena todos los problemas y resultados obtenidos por cada algoritmo en cada iteración de cada problema.

Así, una vez evaluada cada solución, esta se almacena en el propio elemento del Experimento, lo que nos permite representar tanto gráficas de ranking como de convergencia, no sólo para un problema, si no para todo el conjunto de problemas. Además, al estar los datos de cada iteración almacenados, estos pueden ser usados para generar rankings y calcular tests estadísticos como Wilcoxon o Friedman a partir de esos datos.

Por último, se podría hacer mención al “estado del arte” en materia de algoritmos metaheurísticos, pero en el posterior análisis teórico de las implementaciones generadas por el modelo largo de lenguaje se hará hincapié en el funcionamiento de cada metaheurística, referenciando estudios previos en cada caso.



UNIVERSIDAD DE CÓRDOBA

## Antecedentes

---

# Capítulo 4

## Análisis

### 4.1. Pruebas y evaluación

Para evaluar los algoritmos se ha optado por la opción más sencilla, ejecutar los algoritmos sobre problemas TSP y a partir de los resultados obtenidos, evaluar su rendimiento. Con los resultados obtenidos se realizará una evaluación teórica analizando si los resultados evolucionan, aunque no se llegue a una solución óptima.

Para estos experimentos, se ha generado un número de instancias de problemas TSP, concretamente 100 problemas diferentes, cada uno con un número aleatorio de ciudades, entre 25 y 100 ciudades. Sobre cada uno de estos problemas se ejecutarán los algoritmos en igualdad de condiciones, consiguiendo al final 25 soluciones finales sobre cada problema, una por algoritmo.

Para conseguir esta igualdad de condiciones, cada algoritmo se ejecuta durante 10 minutos por cada problema. Así, aunque cada algoritmo se ejecute durante un número distinto de evaluaciones, todos se ejecutarán por un mismo tiempo, lo que nos puede dar una aproximación de su funcionamiento en problemas reales. Equiparando las ejecuciones por número de tiempo evitamos diferencias de rendimiento entre los algoritmos sencillos, complejos y mal implementados. Con 10 minutos de ejecución para cada algoritmo conseguimos equiparar un alto número de iteraciones de los algoritmos sencillos con un menor número de iteraciones de los algoritmos mas lentos y complejos.



Una vez ejecutados los algoritmos, el primer paso será realizar una comparativa de resultados, comparando los costes de todas las ejecuciones y realizando un ranking según resultados obtenidos. Así, conseguimos estudiar qué algoritmos funcionan mejor sobre el tipo de problema. Además podemos realizar los mismos rankings para comparar los resultados medios obtenidos, los mejores resultados y los peores.

Los datos de cada ejecución se almacenan en un archivo CSV, almacenando el mejor coste obtenido, el peor coste obtenido y la media de costes. También se almacenan el número de evaluaciones o iteraciones realizadas por cada algoritmo y la mejor permutación obtenida para poder representarla gráficamente. Con los datos de costes y evaluaciones, se clasifican los algoritmos en base a su coste y sus evaluaciones.

Para el conjunto de datos, se clasifican los algoritmos en función a su rendimiento sobre cada problema. A partir del ranking de resultados de cada problema se realiza un ranking final de algoritmos.

Por otro lado, todas las soluciones obtenidas en cada iteración de cada problema se almacenan en tiempo de ejecución como un elemento de la instancia del experimento. A partir de estos datos, se pueden representar gráficas de ranking en cuanto a resultados y gráficas de convergencia, tanto para un único problema como para el conjunto de problemas.

Partiendo de los mejores costes de cada problema en cada algoritmo, podemos realizar comparaciones a través de tests no paramétricos, como el test de Wilcoxon, el test de Friedman y el test de Nemenyi como método Post-Hoc.

El test de Wilcoxon se aplica en nuestro caso sobre los dos mejores algoritmos y los dos peores, según el ranking de resultados sobre el conjunto de todos los problemas. Con este test podemos afirmar dentro un cierto valor de confianza si hay diferencias significativas en cuanto al rendimiento de los algoritmos.

El test de Friedman es similar al de Wilcoxon, pero analizando grupos de algoritmos en lugar de solo dos. De la misma forma, este test nos indica dentro un cierto valor de confianza si hay diferencias significativas en cuanto al rendimiento de los algoritmos. Este test será de utilidad al comparar algoritmos dentro de una misma sub-familia, pues siendo algoritmos basados en principios similares podemos suponer de base que, dentro de ciertos subgrupos y divisiones, las diferencias no serán significativas, al tratar con algoritmos similares.



Por último, la prueba post-hoc propuesta ha sido el test de Nemenyi. Las pruebas post-hoc se utilizan para determinar, en caso de haberlas, las diferencias determinadas por el test de Friedman. El problema es que aún subdividiendo los algoritmos en subgrupos por similitudes en sus conceptos teóricos, seguimos tratando con algoritmos muy diferentes entre sí. Además, mezclando en esos subgrupos algoritmos bien implementados con otros mal implementados, estas diferencias se acentúan.

Como último detalle, tanto las gráficas de ranking y de convergencia como los tests no paramétricos anteriormente mencionados se han dividido en distintos subgrupos para poder hacer un análisis por grupos y por sub-familias de algoritmos. Las divisiones hechas, además de un estudio completo de todos los algoritmos sobre el conjunto de problemas han sido:

- 3 mejores algoritmos
- 3 peores algoritmos
- 5 mejores algoritmos
- 10 mejores algoritmos
- Metaheurísticas híbridas
- Metaheurísticas evolutivas
- Metaheurísticas evolutivas
- Metaheurísticas de búsqueda local
- Metaheurísticas de enjambre
- Metaheurísticas basadas en poblaciones
- Metaheurísticas inspiradas en procesos naturales

## 4.2. Evaluación de resultados prácticos y tests estadísticos

Para evaluar el rendimiento práctico de los algoritmos, se han tenido en cuenta varias métricas.

En primer lugar, los algoritmos se comparan por sus métricas básicas de rendimiento, como lo son la mejor y peor solución obtenida, el número de evaluaciones realizadas o el coste medio de las soluciones obtenidas a lo largo del proceso. Para cuantificar estas métricas sobre todo el conjunto de problemas, se han “rankeado” los algoritmos en función de sus mejores costes obtenidos, media de costes obtenidos, media de evaluaciones realizadas, y por último, métricas de evaluaciones y coste en función del tiempo.

Al tratar los algoritmos sobre tantos problemas distintos, la única forma de unificar estos resultados es a partir de rankings.

Por último, una comparación entre algoritmos nos debe llevar a una conclusión: ver si hay diferencias significativas entre los algoritmos, y en caso de haberlas, determinar que algoritmo puede ser mejor.

Para esto, se han aplicado los test no paramétricos de Wilcoxon para comparar parejas de algoritmos, y el test de Friedman para comparar grupos de algoritmos. Al tratar una gran variedad de problemas y algoritmos, no podemos suponer que nuestros datos sigan una distribución normal, por lo que no podemos aplicar test paramétricos para evaluar nuestras hipótesis.

Partiendo de la hipótesis de que hay diferencias significativas en el rendimiento de los algoritmos, aplicamos los test no paramétricos para evaluar nuestra hipótesis. Primero, se comparan los dos mejores algoritmos a través de la prueba de Wilcoxon, y luego, a través de la prueba de Friedman se comparan los distintos subgrupos, por familias y por rankings. Además, como prueba post-hoc, en caso de confirmar la hipótesis, se usa el test de Nemenyi, determinando la diferencia crítica entre algoritmos y representándola gráficamente.



El primero de los test a realizar será el test de Wilcoxon [41], utilizado para determinar si existe una diferencia significativa entre dos conjuntos de datos relacionados, pero no necesariamente distribuidos de manera normal. Se emplea en situaciones donde se buscan establecer diferencias entre dos muestras relacionadas, es decir, cuando dos condiciones se aplican a los mismos sujetos. Al ser una prueba que evalúa pares, es perfecto para determinar si hay diferencias significativas entre un par de algoritmos, en nuestro caso, entre los dos mejores o los dos peores.

La razón principal para utilizar el test de Wilcoxon es su capacidad para evaluar diferencias entre dos muestras sin asumir una distribución específica de los datos. Esto lo hace particularmente útil en casos donde los datos no siguen una distribución normal o cuando la muestra es pequeña. En contraste con pruebas paramétricas como la t de Student, el test de Wilcoxon se basa en los rangos de los datos observados, lo que lo hace menos sensible a valores atípicos y más robusto frente a supuestos de normalidad.

El funcionamiento del test consiste en calcular la diferencia entre pares de observaciones, asignarles un rango según su magnitud absoluta y determinar si estos rangos difieren significativamente.

Se emplean los rangos para calcular una estadística de prueba que permite establecer si hay suficiente evidencia para rechazar la hipótesis nula, que afirma que no hay diferencia entre las muestras. Este test ofrece información sobre si existe una diferencia significativa entre las dos muestras emparejadas, sin especificar la dirección de esta diferencia, es decir, no indica cuál es mayor o menor, sino simplemente que hay una disparidad estadísticamente significativa entre ambas. Esto lo convierte en una herramienta valiosa para comparar la efectividad de dos algoritmos sin prejuicios sobre cuál podría ser mejor, sino simplemente si hay suficiente evidencia para afirmar que uno supera al otro en términos de desempeño.

Una vez evaluados los dos mejores algoritmos, se evaluarán todos los algoritmos por grupos usando el test de Friedman [42]. Esta es una prueba no paramétrica que se utiliza para comparar múltiples conjuntos de datos relacionados. Es especialmente útil cuando se busca determinar si hay diferencias significativas en el rendimiento entre varios algoritmos o métodos aplicados a un mismo conjunto de problemas. A diferencia del test de Wilcoxon, que compara solo dos muestras, el test de Friedman permite evaluar simultáneamente varios algoritmos.



Se utilizará el test de Friedman por las mismas razones que el test de Wilcoxon, por su naturaleza no paramétrica, lo que significa que no asume una distribución particular de los datos ni requiere que estos sigan una distribución normal.

El funcionamiento del test de Friedman pasa por evaluar el rendimiento de cada algoritmo en cada problema, clasificándolos de mejor a peor resultado, calculando un valor de clasificación medio para el algoritmo. Luego se calcula una estadística de prueba basada en las diferencias entre los rangos promedio de los algoritmos, lo que permite determinar si existe una diferencia significativa en el rendimiento general entre ellos.

Al obtener un resultado significativo en el test de Friedman, se puede concluir que al menos uno de los algoritmos evaluados difiere en rendimiento. Al igual que el test de Wilcoxon, esta prueba no identifica específicamente cuál o cuáles algoritmos son diferentes entre sí.

Una vez evaluadas las diferencias significativas entre algoritmos y en caso de existir dichas diferencias, el objetivo sería comprobar donde se dan esas diferencias, es decir, que algoritmos superan en rendimiento a los demás. Para ello podemos realizar pruebas post-hoc, como el test de Nemenyi [43].

El test de Nemenyi es una prueba empleada después de un test global, como el test de Friedman, para determinar qué pares de algoritmos presentan diferencias significativas en su rendimiento. Es una herramienta valiosa cuando se busca identificar específicamente qué algoritmos difieren entre sí después de haber obtenido un resultado significativo en una prueba global que indica que al menos uno de los algoritmos es diferente.

Esta prueba considera que el rendimiento de dos clasificadores cualesquiera se considera significativamente diferente si sus rangos medios difieren al menos en la diferencia crítica:

$$CD = q_\alpha \cdot \sqrt{\frac{k(k+1)}{6n}}$$

Donde  $k$  es el número de algoritmos evaluados,  $n$  el número de problemas sobre los que se evalúan los algoritmos, y  $q_\alpha$  es el valor crítico de la distribución de referencia para un nivel de significancia  $\alpha$ .



Se calcula una estadística de prueba que se compara con un valor crítico obtenido de tablas estadísticas específicas para determinar si hay diferencias significativas entre pares de algoritmos. Si la distancia entre los rangos promedio de dos algoritmos es mayor que un valor crítico específico, se concluye que estos algoritmos tienen un rendimiento significativamente diferente en comparación con los demás.

Como se ha indicado anteriormente, la elección de pruebas estadísticas no paramétricas sobre las paramétricas se basa en la flexibilidad y la robustez que ofrecen frente a los supuestos de distribución de los datos. Los test no paramétricos no requieren que los datos sigan una distribución específica, como la normal, lo que los hace ideales para analizar conjuntos de datos que pueden presentar desviaciones de la normalidad o para muestras pequeñas.

Además, estos test son menos sensibles a valores atípicos y ofrecen una mayor generalización, siendo una opción más confiable cuando los supuestos de las pruebas paramétricas no se cumplen o no pueden verificarse con certeza en el conjunto de datos analizado.



# Capítulo 5

## Diseño

Al igual que cualquier otro proyecto, es necesario comprender la metodología de trabajo seguida. Aunque en nuestro caso no se pueda hablar de un diseño propio de los desarrollos de software convencionales, si estamos ante una metodología de experimentación clara a explicar. En primer lugar, podemos representar la metodología seguida en la figura 5.1.

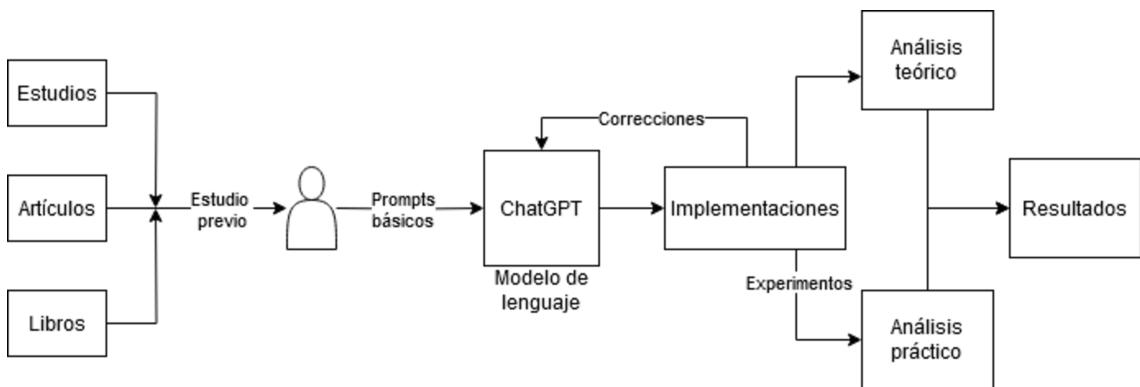


Figura 5.1: Diagrama del proceso realizado

En la figura podemos ver que el primer paso de nuestro proyecto ha sido el estudio previo, tanto de las tecnologías a usar como de los algoritmos a implementar. En este estudio previo se ha hecho especial énfasis en el estudio de los algoritmos a desarrollar.



Los primeros estudios explorados [9][18] han sido de utilidad a la hora de desarrollar un listado de algoritmos metaheurísticos, buscando mezclar algoritmos muy estudiados con otros menos estudiados, partiendo de distintas bases de funcionamiento y distintos conceptos teóricos. En definitiva, se buscaba un listado de metaheurísticas lo más variado y diverso posible.

Para cada algoritmo se han buscado artículos, estudios y publicaciones en libros que desarrollen en detalle el funcionamiento de cada uno. Sabiendo que cada autor pueda dar una interpretación propia al desarrollo y funcionamiento del algoritmo, sobre todo en la implementación, se han buscado publicaciones lo más generales posibles, sin buscar implementaciones concretas de una metaheurística en específico.

Además, cada uno de estas publicaciones parten de una misma premisa, explicar a través del lenguaje natural el funcionamiento y desarrollo de las operaciones características del algoritmo. Partiendo de estas descripciones en lenguaje natural, podemos estudiar el funcionamiento del algoritmo sin analizar la implementación dependiente del autor, lenguaje de programación usado o del problema sobre el cual trabaja.

Una vez estudiados los algoritmos, también se debe estudiar los dos siguientes elementos importantes, el problema sobre el que se trabaja, la forma de evaluar y los tests paramétricos a usar. Los tres elementos se han explicado en detalle en los apartados de Antecedentes 3, Pruebas y evaluación 4.1 y Evaluación de resultados prácticos y tests estadísticos 4.2 respectivamente.

Con la base teórica ya estudiada, podemos empezar a generar las implementaciones usando el modelo largo de lenguaje, en nuestro caso, ChatGPT. Usando la versión web [29], estamos ante una herramienta sencilla de usar. El modelo, en forma de chat o conversación, recibe una entrada de texto, el llamado *prompt*, devuelve una respuesta en lenguaje natural. En nuestro caso el prompt, que no es más que una simple cadena de texto, busca ser lo más sencilla y poco específica posible. La respuesta generada por el modelo largo de lenguaje no es más que otra cadena de texto, en nuestro caso acompañada de código de programación. Un ejemplo de uso sería el representado en la figura 5.2. Los prompts serán explicados con más detalle en el apartado de Resultados 6.



The screenshot shows a conversation between a user and ChatGPT. The user asks for a Python function to implement bubble sort. ChatGPT responds with the following code:

```
python
def ordenacion_burbuja(lista):
    """
    Ordena una lista utilizando el algoritmo de ordenación burbuja.

    :param lista: Lista a ser ordenada.
    :return: Lista ordenada.
    """

    n = len(lista)

    # Recorrer todos los elementos de la lista
    for i in range(n):
        # Últimos i elementos ya están en su lugar correcto
        for j in range(0, n - i - 1):
            # Intercambiar si el elemento encontrado es mayor que el si
            if lista[j] > lista[j + 1]:
```

Figura 5.2: Ejemplo de uso de ChatGPT

Las implementaciones generadas se probarán brevemente para comprobar que el código sea funcional y genere soluciones válidas. Se debe recalcar que en estas revisiones no se busca comprobar que la implementación sea la esperable ni que implemente las operaciones características del algoritmo, solo que sea código funcional.

En caso de obtener código que no funcione o que no genere soluciones válidas, la propia implementación generada será reevaluada y corregida por el modelo de lenguaje, generando las modificaciones que este considere oportunas.



Con todos los algoritmos ya implementados y funcionales, se pasa a una doble fase de evaluación. En primer lugar, una evaluación teórica, evaluando el código generado por ChatGPT y comparandola con una implementación esperable según el estudio previo. Posteriormente, las implementaciones pasarán por una fase de experimentación de la que se obtendrán los resultados de estas ejecuciones para su posterior estudio y análisis. Estas pruebas están detalladas en el apartado Pruebas y evaluación 4.1.

Por último, los resultados obtenidos serán estudiados y evaluados, obteniendo distintas ordenaciones de los algoritmos en función de estos resultados y pudiendo sacar conclusiones del rendimiento a través de tests no paramétricos. Tanto la evaluación de los resultados como los tests estadísticos están detallados en el apartado Evaluación de resultados prácticos y tests estadísticos 4.2.

Con estos pasos anteriormente descritos podemos resumir el proceso seguido para realizar nuestro estudio.



# Capítulo 6

## Resultados

A continuación, analizaremos de forma teórica las implementaciones generadas por el modelo largo de lenguaje.

Queremos estudiar si, para cada algoritmo, ChatGPT es capaz de dar una implementación a partir de una entrada genérica. Al buscar implementaciones genéricas de los algoritmos, es decir, adaptables para cualquier tipo de problema, es importante tener una entrada o petición descriptiva, que indique con claridad el algoritmo a obtener y la clase de soluciones a tratar.

A la hora de tratar peticiones y sobre todo al esperar respuestas específicas de modelos largos de lenguaje, lo más importante es la petición a realizar (en inglés *prompt*), es decir, el mensaje que se envía al modelo de lenguaje, esperando un resultado. Mínimas variaciones en estos prompts, como pueden ser el idioma, tecnicismos, o palabras clave pueden afectar al resultado obtenido.

Aunque distintas pruebas propias dan a entender que los prompts escritos en inglés suelen dar mejores resultados, se ha optado por usar entradas en español, al estar todo el proyecto en este idioma. Se ha optado por una entrada que busca ser específica en cuanto a funciones, parámetros, variables y retornos. También se especifican restricciones en cuanto a las soluciones y su formato, ya que se tiende a generar soluciones no válidas para el problema específico.

Además, se ha priorizado el no especificar nada del algoritmo a desarrollar, salvo el nombre. Como se busca analizar hasta qué punto el modelo de lenguaje conoce el algoritmo, no se especificará ningún detalle sobre el funcionamiento del algoritmo, su flujo de ejecución, ni ningún otro detalle que no pueda ser aplicable a cualquier otra metaheurística.



El prompt usado como base para obtener una primera implementación de todos los algoritmos ha sido:

Crea una función en Python que realice la metaheurística:  
"Nombre de la metaheurística"

La función tendrá como parámetros un número de ciudades y una función que evalúa el coste de las soluciones generadas.

Debe trabajar con permutaciones de los primeros num\_ciudades números enteros. Cada permutación no puede tener repeticiones de números dentro de sí misma.

Para calcular el coste, usa la función evaluar(permutación). Como retorno, deberá devolver la mejor solución

En todas las ocasiones, al especificar que se trabaja con metaheurísticas y con permutaciones sin repetición de un tamaño *número de ciudades*, reconoce que está trabajando sobre un problema del Viajante de comercio.

Una vez obtenida una implementación, esta rara vez suele ser útil. En la gran mayoría de casos, las implementaciones no suelen ser funcionales, suelen tener errores de programación o fallan a la hora de generar soluciones válidas, entre otros errores. En estos casos, se evita una modificación por parte del programador humano, especificándole al modelo de lenguaje que cambio implementar. Un claro ejemplo de esto es la generación de soluciones válidas, pues se tiende a generar permutaciones con repeticiones. En estos casos, se le indica al modelo de lenguaje que modifique su propia implementación, de la forma que considere oportuna.

En estos casos, no se especifica la causa del error ni dónde puede estar causado. Para solucionarlo, nos limitamos a un prompt genérico, indicando el tipo de solución que debe generar la implementación. Un ejemplo de estos prompts son:

Las soluciones generadas no son correctas.

Revisa la implementación, generando como soluciones permutaciones de los primeros num\_ciudades números enteros, sin repetición.



El otro gran problema de las implementaciones generadas es el desconocimiento a primeras instancias del algoritmo o su funcionamiento. En estos casos, se tiende a generar una metaheurística genérica, basada en población con un reemplazo aleatorio, lo que se puede resumir como una búsqueda aleatoria.

En estos casos, se le indica al modelo de lenguaje que la implementación dada es genérica, con prompts como:

**La implementación generada es una metaheurística genérica.**

**No se aplican las operaciones propias del algoritmo solicitado.**

**Revisa la implementación, aplicando las operaciones propias del algoritmo.**

Es importante recalcar que en ningún momento se especifican ni aclaran cuales son las operaciones propias del algoritmo ni lo que se espera. En el caso específico de ChatGPT, tiende a responder con algoritmos genéricos en el caso de no reconocer realmente el algoritmo. Aplica un cierto “sesgo de confirmación”, afirmando que si conoce la metaheurística solicitada, y dando una implementación que nada tiene que ver, cambiando el nombre de las variables.

En los casos en los que la implementación no se acerca en se intenta mejorar la implementación original durante varios intentos, variando el prompt indicado anteriormente, hasta que llega a variar la implementación (sin necesidad de que sea correcta o aproximada), o hasta que entre en un bucle en el que no llegue a variar la implementación.

Por último, se ha evitado realizar cambios, por mínimos que sean, en las implementaciones generadas. Los únicos cambios realizados han sido el almacenar los resultados de cada iteración en listas, para su posterior tratamiento y análisis de los resultados prácticos, además de cambios menores que no afectan al funcionamiento del algoritmo.

A continuación, se analizarán las implementaciones de cada algoritmo, haciendo énfasis en los aspectos teóricos de estas. No se comentará el propio código generado, pues el impacto en el funcionamiento es mínimo. El énfasis estará en el funcionamiento del algoritmo y su flujo de trabajo, representado por el pseudocódigo de la implementación.

Para seleccionar los algoritmos a implementar se ha intentado buscar una lista variada de metaheurísticas, mezclando las metaheurísticas más estudiadas según



el artículo [9] con otras menos estudiadas. Además, basándonos en el artículo [18] podemos seleccionar metaheurísticas basadas en distintos conceptos y bases de funcionamiento, pudiendo evaluar distintos principios de funcionamiento sobre el mismo tipo de problema. Se ha buscado generar una lista de metaheurísticas variada tanto en funcionamiento como en bibliografía previa para ver como influye este conocimiento previo en la generación de código por parte del modelo largo de lenguaje y a la vez se ha buscado que la lista de algoritmos sea variada en cuanto a bases de funcionamiento para ver cómo esto influye en el rendimiento sobre el problema a tratar.



## 6.1. Implementaciones generadas

### 6.1.1. Búsqueda aleatoria

El primer algoritmo implementado ha sido la búsqueda aleatoria.

Aunque es un enfoque simple y no siempre eficiente, la Búsqueda aleatoria se suele utilizar como punto de partida o enfoque inicial en problemas de optimización o búsqueda, especialmente cuando no se dispone de información previa sobre el problema.

La búsqueda aleatoria se basa en un principio básico, generar una solución aleatoria y evaluarla [44]. Generalmente, la gran mayoría de metaheurísticas tienen un componente de búsqueda aleatoria, por ejemplo, al generar las poblaciones iniciales.

Si bien es cierto que la búsqueda aleatoria no se puede considerar una metaheurística en el sentido tradicional, ya que no sigue una estrategia específica para explorar el espacio de búsqueda, sirve como un elemento de comparación.

Basándonos en la implementación obtenida para la búsqueda aleatoria podemos comparar la calidad de las demás implementaciones, ver hasta qué punto es capaz de completar la implementación, y aún más, comparar lo útil que es el algoritmo generado.

Por poco eficaz que sea la búsqueda aleatoria, con el paso del tiempo y de las iteraciones la solución evoluciona de una forma totalmente aleatoria. Nos interesa que todas las metaheurísticas implementadas sean, como mínimo, mejores que una búsqueda aleatoria.

Podemos afirmar que cualquier metaheurística con una estrategia específica para explorar el espacio de búsqueda y mejorar las soluciones obtenidas debe resultar en mejores soluciones que cualquier búsqueda aleatoria. Podemos aproximar el número de muestras necesarias para que una búsqueda aleatoria obtenga la solución óptima con cierta probabilidad, a partir de la ecuación:

$$N > \frac{\log(1 - p)}{\log(q)}$$



Donde  $N$  representa el número mínimo de iteraciones o muestras necesarias para obtener la solución óptima,  $p$  la probabilidad de encontrar la solución óptima en al menos una de las muestras y  $q$  la probabilidad de obtener la solución óptima en cada intento.

En el caso en el que una metaheurística tenga unos resultados peores que una búsqueda aleatoria podemos afirmar que estará mal implementado. Cualquier metaheurística con una clara estrategia de búsqueda, con elementos de intensificación y diversificación de soluciones debe obtener mejores soluciones, aunque sea a un mayor coste de tiempo y recursos.

La implementación obtenida para este algoritmo es la representada en el pseudocódigo 1.

---

**Algoritmo 1** Búsqueda aleatoria

---

```
1: mejor_coste  $\leftarrow \infty$ 
2: mientras tiempo < tiempo límite hacer
3:   solucion  $\leftarrow$  generar_permutacion()
4:   coste  $\leftarrow$  evaluar_solucion(solucion)
5:   si coste < mejor_coste entonces
6:     mejor_solucion  $\leftarrow$  solucion
7:     mejor_coste  $\leftarrow$  coste
8: devolver mejor_solucion
```

---

En este caso, al ser un algoritmo tan sencillo y reconocido, la implementación ha sido correcta, sin ningún error ni fallo en el concepto.

Dentro de un bucle, en este caso por límite de tiempo, se genera una solución completamente aleatoria, se evalúa y se almacena sólo si es mejor que la última mejor solución. El proceso acaba retornando la mejor solución obtenida.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.1.

## Resultados

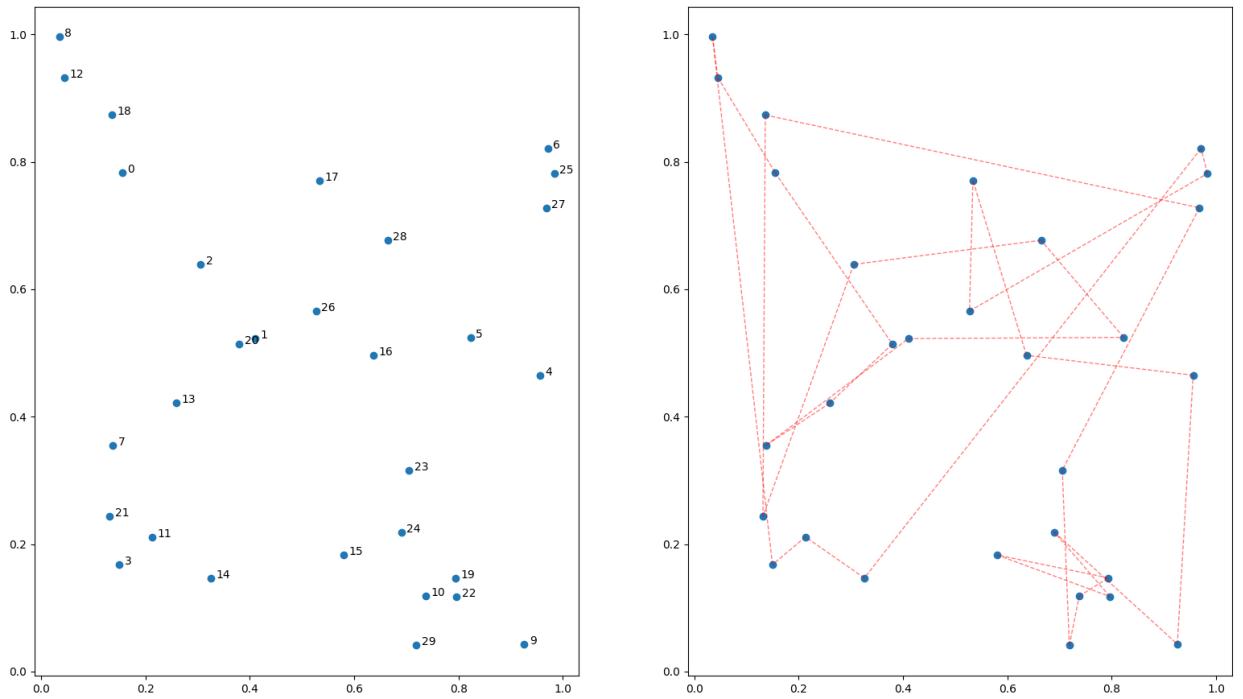


Figura 6.1: Ejemplo rendimiento Búsqueda aleatoria

### 6.1.2. Búsqueda local por primer vecino

El siguiente algoritmo se trata de una variación de la simple búsqueda local, la búsqueda local por primer vecino [45]. A partir de este algoritmo, ya estamos tratando metaheurísticas propiamente dichas, con sus claras estrategias de exploración del espacio de búsqueda, intensificación y diversificación.

La búsqueda local por primer vecino se basa en la búsqueda local clásica, en la que se parte de una solución aleatoria. De esta solución se obtienen las soluciones vecinas, escogiendo como nueva solución el primer vecino que mejore la solución previa.

El proceso se repite hasta no poder mejorar la solución. Esta variación de la búsqueda local prioriza la diversificación de soluciones a la intensificación, al no buscar el mejor vecino posible.



La implementación obtenida es la reflejada en el pseudocódigo 2.

---

**Algoritmo 2** Búsqueda local por primer vecino

---

```
1: mejor_solucion ← generar_permutacion()
2: mejor_coste ← evaluar_solucion(mejor_solucion)

3: mientras tiempo < tiempo límite hacer
4:   vecinos ← explorar_soluciones_vecinas(mejor_solucion)
5:   para todos vecino en vecinos hacer
6:     coste ← evaluar_solucion(vecino)
7:     si coste < mejor_coste entonces
8:       mejor_solucion ← vecino
9:       mejor_coste ← coste

10: devolver mejor_solucion
```

---

La implementación refleja una búsqueda local correcta. El flujo de trabajo del algoritmo es el esperado, partiendo de una solución aleatoria y evaluando sus vecinos.

La generación de vecinos es también correcta, aunque no esté reflejada en el pseudocódigo. La generación de vecinos es completa, generando todos los posibles vecinos partiendo de dos operadores de vecindario muy comunes a la hora de trabajar con metaheurísticas, “2-opt” [46], que invierte el orden de los elementos de la solución dentro de un rango, y “2-swap” [47][48] que intercambia elementos de la solución. En este caso, la generación de vecinos es considerablemente completa, pues para soluciones de 100 elementos, se generan un total de 4950 soluciones vecinas distintas por cada operador.

$$\text{Nº vecinos} = C(n, 2) = \frac{n!}{(2! * (n - 2)!)}$$

Una vez generados los vecinos estos se evalúan, y es aquí donde la implementación es errónea.

Al evaluar las soluciones del vecindario, no hay ningún criterio de parada, es decir, el algoritmo evaluará todos los posibles vecinos, seleccionando el mejor.

## Resultados

Aunque sea una implementación correcta, no se estaría implementando una búsqueda local por primer vecino que pare al encontrar la primera solución vecina que mejore el coste, sino que se está implementando una búsqueda local por mejor vecino, en el que se evalúan todos los posibles vecinos, seleccionando el mejor. Esta interpretación de la búsqueda local será estudiada posteriormente.

La implementación de la búsqueda local por primer vecino es errónea, al fallar en el elemento clave que la diferencia de otros métodos. Aún así, es destacable la calidad de la implementación, en especial al generar vecinos.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.2.

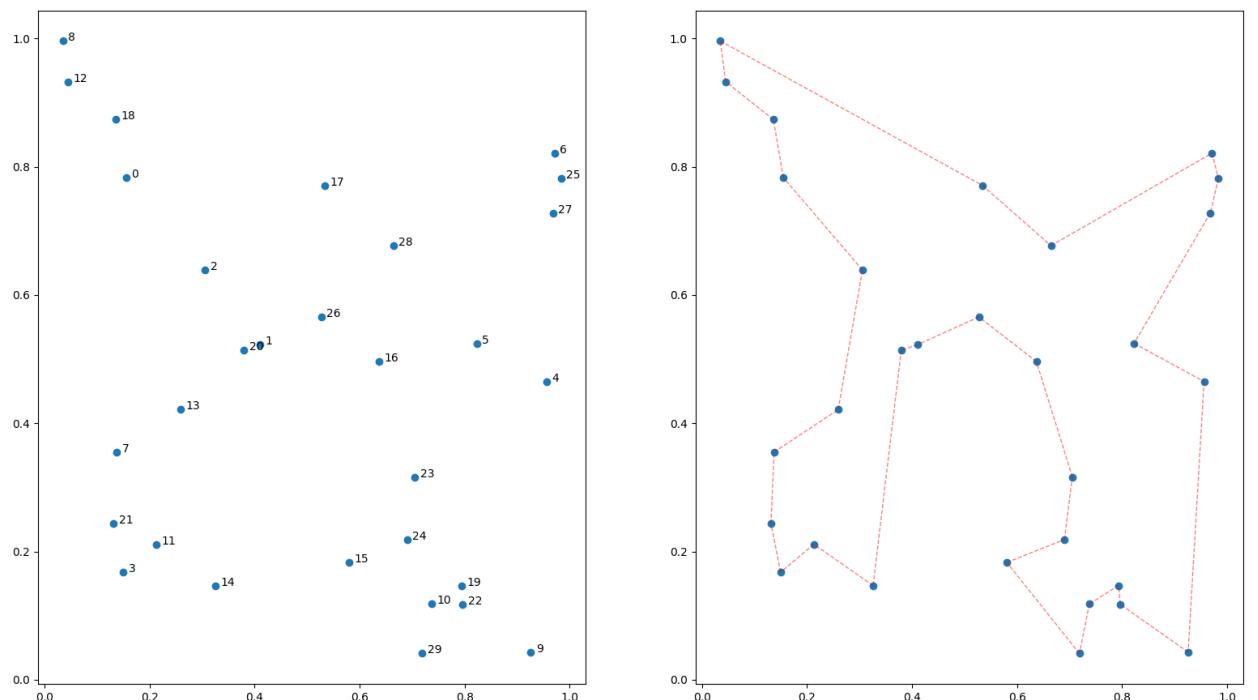


Figura 6.2: Ejemplo rendimiento Busqueda local primer vecino



### 6.1.3. Búsqueda local por mejor vecino

Este algoritmo es la segunda variación de la búsqueda local clásica. La búsqueda local por mejor vecino es la versión “opuesta” [49] a la búsqueda local por primer vecino vista anteriormente.

El implementar dos interpretaciones similares, basadas en un mismo algoritmo nos sirve para estudiar cuan bien llega el modelo de lenguaje a entender la entrada, lo que realmente se le está solicitando y estudiar si es capaz de entender e implementar los elementos diferenciadores por menores que sean.

De nuevo, partimos de la búsqueda local clásica, generando una solución aleatoria, y explorando los posibles vecinos. La gran diferencia llega a la hora de explorar ese vecindario.

En lugar de explorar el vecindario hasta encontrar una mejor solución, evaluamos una por una todas las soluciones de este vecindario, almacenando el mejor de los vecinos y partiendo de este mejor vecino para la siguiente iteración. Aunque esta implementación sea más costosa en cuanto a tiempo de ejecución por el aumento en el número de evaluaciones, está centrada en la intensificación de las soluciones.

Realmente este es el algoritmo implementado anteriormente en el intento de búsqueda local por primer vecino. La implementación obtenida para este algoritmo está representada en el pseudocódigo 3.

---

#### Algoritmo 3 Búsqueda local por mejor vecino

---

```
1: mejor_solucion  $\leftarrow$  generar_permutacion()
2: mejor_coste  $\leftarrow$  evaluar_solucion(mejor_solucion)
3: mientras tiempo < tiempo límite hacer
4:   vecinos  $\leftarrow$  explorar_soluciones_vecinas(mejor_solucion)
5:   para todos vecino en vecinos hacer
6:     coste  $\leftarrow$  evaluar_solucion(vecino)
7:     si coste < mejor_coste entonces
8:       mejor_solucion  $\leftarrow$  vecino
9:       mejor_coste  $\leftarrow$  coste
10: devolver mejor_solucion
```

---



En este caso, la implementación obtenida es correcta. En primer lugar, es destacable que la implementación obtenida es exactamente igual a la de la búsqueda local por primer vecino, salvo pequeñas modificaciones que no afectan al flujo de trabajo ni funcionamiento del algoritmo.

La generación inicial de soluciones es correcta, y la generación de soluciones vecinas es completa y correcta. Para generar las soluciones del vecindario, se usan las mismas funciones implementadas en el algoritmo anterior, usando los dos mismos operadores de vecindario, “2-opt” y “2-swap”. De nuevo, se generan todos los posibles vecinos.

Para esta implementación, la evaluación del vecindario si es la esperada en este algoritmo en concreto, pues se evalúan todos los vecinos sin ninguna condición de parada, almacenando la mejor solución y usando esta nueva solución para la siguiente iteración del algoritmo. Podemos afirmar que la implementación es completa y correcta.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.3.

## Resultados

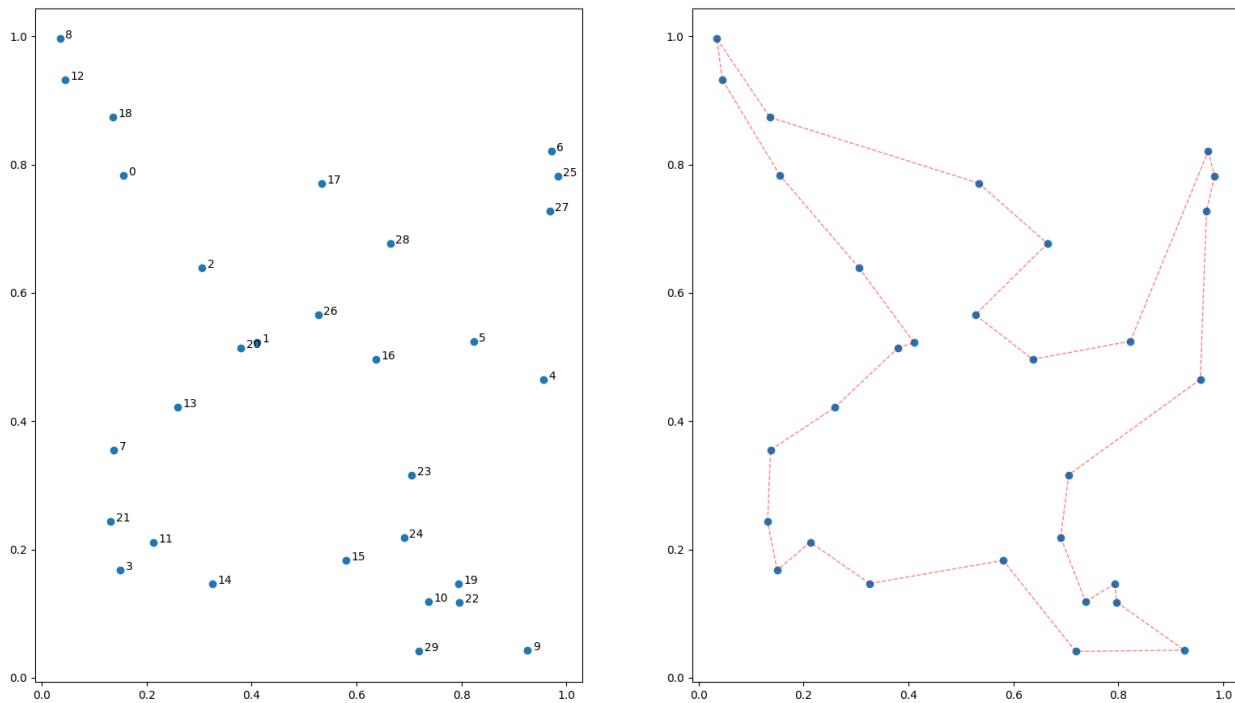


Figura 6.3: Ejemplo rendimiento Búsqueda local mejor vecino

### 6.1.4. Búsqueda tabú

La búsqueda tabú es una de las metaheurísticas basadas en trayectoria [50][51] más conocidas. Este algoritmo está orientado a evitar la región de atracción de un óptimo local.

Las metaheurísticas basadas en trayectoria tienden a converger rápidamente a soluciones sub-óptimas, sin poder “escapar” de esa región del espacio de búsqueda, al orientar la trayectoria que se siguen en el espacio de búsqueda únicamente hacia mejores soluciones.

Para escapar de estas cuencas de atracción de las soluciones óptimas, este algoritmo aplica el concepto de **lista tabú**. Esta lista, con un tamaño y tenencia prefijados, almacenará soluciones previamente “visitadas”, evitando que el algoritmo vuelva a iterar sobre estas.

Así, el algoritmo irá explorando el espacio de búsqueda a partir de una búsqueda local, explorando las soluciones vecinas y seleccionando la mejor posible.



Cada una de estas soluciones seleccionadas será almacenada en la lista tabú, y permanecerá en la lista por un número de iteraciones fijado por la tenencia de la lista.

Al llegar a una solución que sea una óptima local, es decir, que no tenga ningún vecino mejor, el algoritmo pasará a evaluar el mejor vecino que no esté almacenado en la lista tabú. Así, el algoritmo trabajará sobre una solución en primera instancia peor, pero abre la posibilidad de explorar un nuevo área del espacio de búsqueda. Además, al existir la lista tabú, nos aseguramos que no vuelva a trabajar sobre la solución óptima anteriormente evaluada.

Así se mantiene un equilibrio entre intensificación, trabajando sobre las mejores soluciones de cada vecindario, y diversificación, explorando nuevas áreas del espacio de búsqueda al toparnos con una óptima local.



La implementación obtenida para el algoritmo se representa en el pseudocódigo 4.

---

**Algoritmo 4** Búsqueda tabú

```
1: solucion_actual ← generar_permutacion()
2: mejor_solucion ← solucion_actual
3: mejor_coste ← evaluar_solucion(solucion_actual)
4: lista_tabu ← lista_vacia

5: mientras tiempo < tiempo límite hacer
6:   vecinos ← explorar_soluciones_vecinas(solucion_actual)
7:   vecinos ← eliminar_vecinos_tabu(vecinos, lista_tabu)
8:   para todos vecino en vecinos hacer
9:     coste ← evaluar_solucion(vecino)
10:    si coste < mejor_coste entonces
11:      mejor_coste ← coste
12:      mejor_solucion ← vecino
13:    solucion_actual ← mejor_solucion(lista_tabu)
14:    lista_tabu ← mejor_solucion

15: devolver mejor_solucion
```

---

En este caso, el algoritmo está bien implementado, respetando todos los elementos específicos y diferenciadores propios de la metaheurística.

La generación de vecinos es similar a los algoritmos de búsqueda local implementados previamente, siendo esta correcta y completa. Para generar los vecinos se aplican los dos operadores de vecindario vistos en los algoritmos anteriores, “2-opt” y “2-swap”. La evaluación de vecinos es correcta, trabajando con la mejor solución del vecindario.

Una vez evaluado todo el vecindario, se almacena el mejor vecino en la lista tabú, y en la siguiente iteración se trabaja con esta solución, generando las soluciones vecinas de estas. En este caso, se trabaja en base a la mejor solución entre los posibles vecinos.

Si bien se podría trabajar con el primer vecino que mejore la solución inicial, no se han especificado detalles específicos para la implementación en el prompt, así que podemos afirmar que la implementación generada es correcta.

## Resultados

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.4.

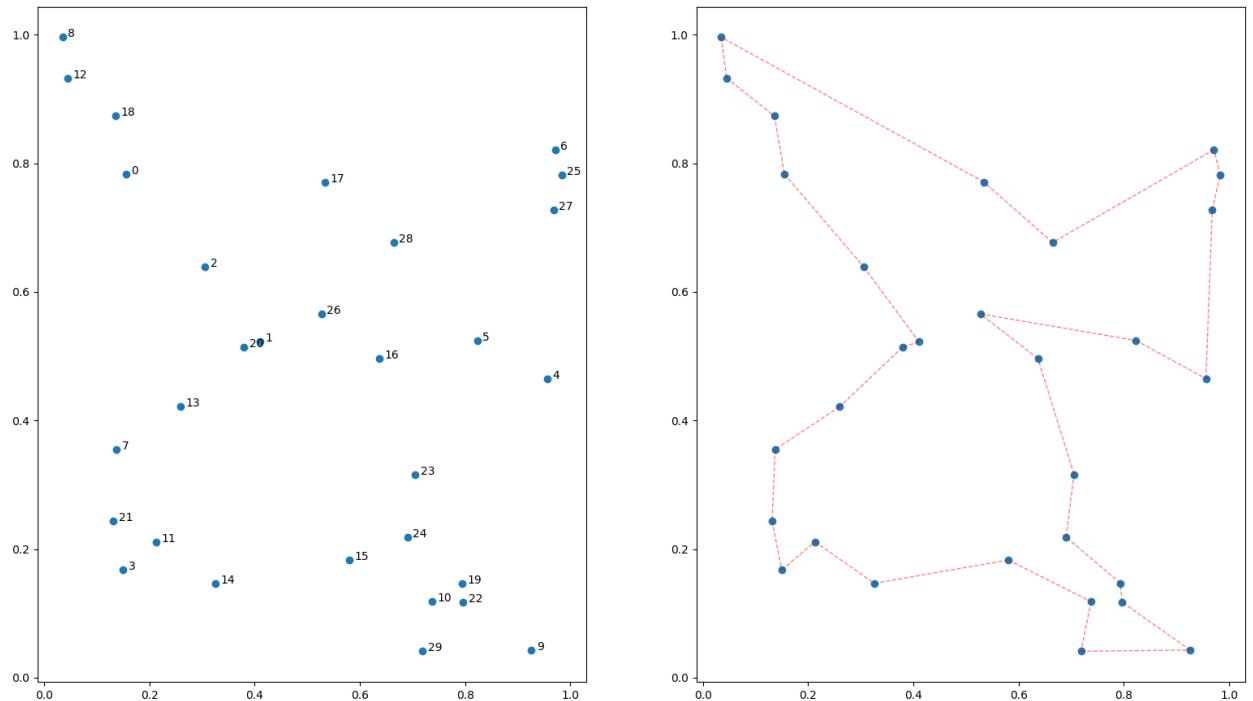


Figura 6.4: Ejemplo rendimiento Búsqueda tabú



### 6.1.5. Enfriamiento simulado

Dentro de las metaheurísticas basadas en trayectorias, la gran problemática es poder escapar de las óptimas locales. Junto a la búsqueda tabú, uno de los algoritmos más conocidos es el enfriamiento simulado.

El enfriamiento simulado funciona en torno al concepto de temperatura [52]. A lo largo del proceso existe un valor de temperatura que irá disminuyendo, es decir se irá enfriando, conforme pasen las iteraciones. Este enfriamiento puede variar, siendo esta fórmula de enfriamiento un factor determinante en el funcionamiento del algoritmo. Por ejemplo, el método de enfriamiento más común es reducir la temperatura en un porcentaje por iteración:

$$T = \alpha T, \text{ donde } (\alpha = 0,99)$$

El algoritmo comienza con una solución aleatoria, y a partir de esta, se genera una solución en el vecindario. Esta nueva solución se evalúa, y en caso de ser mejor que la solución inicial, se mantiene como nueva solución sobre la que trabajar.

En caso en que el vecino generado sea peor, se podrá mantener como solución o no según una función de aceptación. Esta función de aceptación será la que determine la probabilidad con la que se considera una solución como válida para seguir trabajando con ella o no, basándose en el valor de temperatura en ese punto de la ejecución.

A medida que la temperatura sea mayor, la función de aceptación aceptará peores soluciones con una mayor probabilidad. Así, mientras las iteraciones avanzan y la temperatura disminuye, menor será la probabilidad de aceptar una solución peor.

El algoritmo asegura una diversificación de las soluciones, sobre todo al principio de la ejecución, al permitir soluciones a priori peores. También se asegura la intensificación, pues una solución mejor siempre será aceptada.



La implementación generada para este algoritmo es la representada en el pseudocódigo 5.

---

**Algoritmo 5** Enfriamiento simulado

---

```
1: mejor_solucion ← generar_permutacion()
2: mejor_coste ← evaluar_solucion(mejor_solucion)

3: mientras tiempo < tiempo limite hacer
4:    $T_i \leftarrow T_0 * (\text{tasa enfriamiento}^{\text{limite tiempo}})$ 
5:   probabilidad ← numero entre 0 y 1
6:   si probabilidad < 0.5 entonces
7:     vecino ← generar_vecino_2_otp(mejor_solucion)
8:   si probabilidad > 0.5 entonces
9:     vecino ← generar_vecino_2_swap(mejor_solucion)
10:  coste ← evaluar_solucion(vecino)
11:  dif ← (coste - mejor_coste)
12:  si coste < mejor_coste o numero_aleatorio < difTi entonces
13:    mejor_coste ← coste
14:    mejor_solucion ← vecino

15: devolver mejor_solucion
```

---

La implementación tiene ciertos puntos correctos, como la generación de vecinos. En este caso, genera vecinos a partir de dos operadores de vecindario previamente vistos, “2-opt” y “2-swap”. Además de la propia diversificación que proporciona el crear vecinos aleatorios, es destacable el que se generan vecinos aleatorios a través de dos operadores completamente distintos, lo que añade aún más variedad en las soluciones.

La nueva solución se evalúa y se acepta en caso de ser mejor, o de ser aceptada por la función de aceptación. En este caso la función de aceptación decide basándose en la temperatura, y a la vez teniendo en cuenta un factor de aleatoriedad. La función de aceptación, que opera a partir de la diferencia de costes entre la solución original y el vecino generado ( $\Delta E$ ) es:

$$\text{aceptacion}(\Delta E) = \begin{cases} \text{acepta} & \text{si n\'umero aleatorio} < \exp\left(-\frac{\Delta E}{T}\right) \\ \text{rechaza} & \text{resto de casos} \end{cases}$$



La implementación parece conocer los requisitos y elementos claves de la metaheurística indicada, pero falla al reducir el valor de la temperatura. Aunque bien intenta actualizar este valor en cada iteración, esta actualización es realmente inútil, pues en cada iteración le acaba dando el mismo valor. La función de enfriamiento desarrollada es:

$$T = T_0 \cdot \text{tasa\_enfriamiento}^{\text{limite\_tiempo}}$$

Se puede observar que la temperatura se actualiza en función de una tasa de enfriamiento y del límite del tiempo de ejecución, lo que, aunque poco eficaz, puede considerarse correcto. El fallo se da al calcular la temperatura a partir de la  $T_0$ , la temperatura inicial. Haciendo esto, nunca se considera el valor de temperatura en la iteración previa, por lo cual no se llega a actualizar.

Podemos afirmar que el modelo de lenguaje llega a interpretar correctamente los elementos claves de la metaheurística e intenta implementarlos, pero falla a la hora realizar cálculos específicos, en este caso, el enfriamiento de la temperatura.

Aún con este fallo, la metaheurística sigue siendo “funcional”, y sigue conservando los elementos de intensificación y diversificación de resultados propios del algoritmo. El gran problema de esta implementación es que se pierde la evolución, desde la diversificación al inicio con valores altos de temperatura a la intensificación dada con valores bajos de temperatura.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.5.

## Resultados

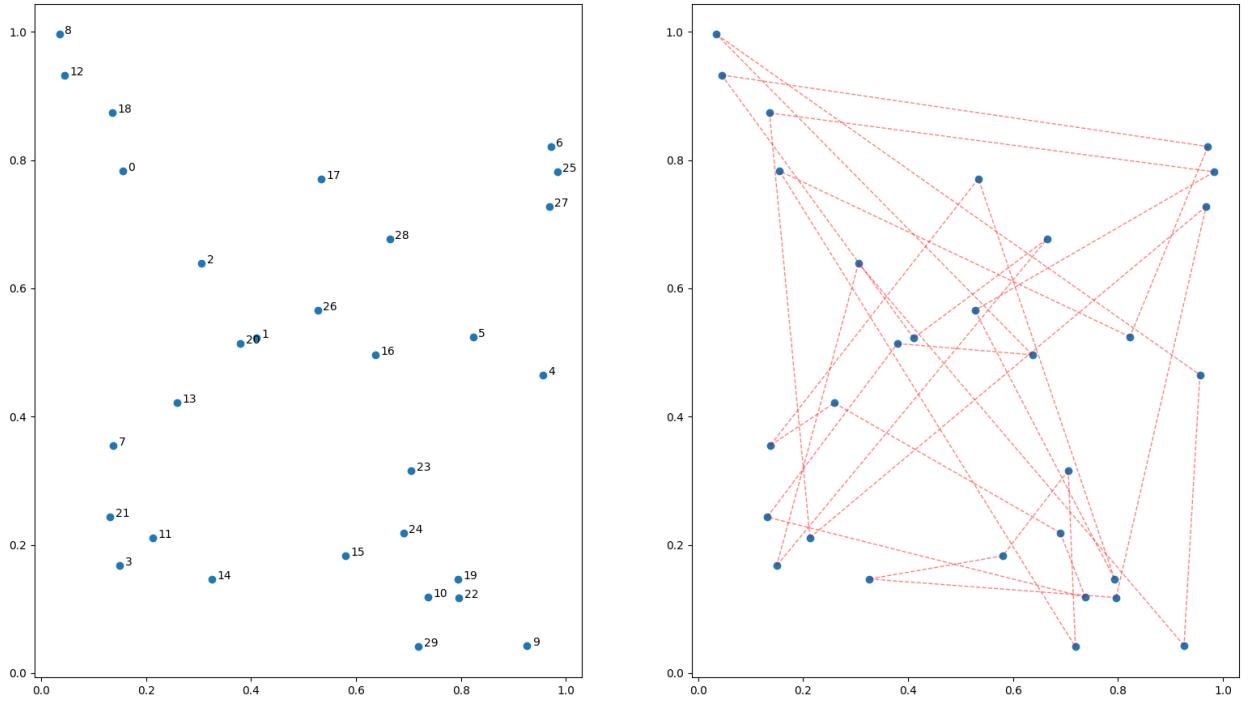


Figura 6.5: Ejemplo rendimiento Enfriamiento simulado

### 6.1.6. Algoritmo de colonia de hormigas

La metaheurística de colonia de hormigas es uno de los algoritmos basados en procesos naturales más conocidos y estudiados. Su funcionamiento está inspirado en cómo las hormigas, trabajando como población, consiguen explorar las proximidades de su hormiguero para encontrar alimento [53].

Lo curioso de este comportamiento no es sólo que se consigue encontrar un camino entre el hormiguero y una fuente de alimento a priori desconocida, sino que todas las hormigas son capaces de seguir un mismo camino “óptimo”, evitando así obstáculos y aunando un esfuerzo colectivo a un mismo objetivo.

En la naturaleza, las hormigas exploran el área cercana a su hormiguero en busca de alimento. En esta búsqueda, las hormigas sueltan un rastro de feromonas [54], en primer lugar, para poder encontrar el camino de vuelta a su hormiguero. Al encontrar alimento, se necesita la cooperación de más hormigas para acercar la comida al hormiguero.



Así, al volver al hormiguero, las hormigas dejan un rastro aún mayor de feromonas por dos motivos. En primer lugar, para poder encontrar con facilidad la fuente de alimento, y poder atraer a más hormigas a través de esas feromonas. A través del rastro de feromonas, estas hormigas pueden encontrar el camino directo entre el hormiguero y su objetivo, y atraer a más hormigas para trabajar conjuntamente.

En nuestro algoritmo, podemos emular este proceso natural a través de una matriz de feromonas, que emula el mapa de nuestro problema. Así, partiendo de una población de hormigas, cada una representando una posible solución, cada hormiga dejará una cantidad de feromonas en el “camino” que representa, proporcional a la calidad de ese camino. A mejor sea la solución, mayor cantidad de feromonas se dejará en ese camino.

Así, se consigue un equilibrio entre diversificación, representada por la propia población de soluciones aleatorias, e intensificación, representada por la cantidad de feromonas en cada elemento de cada solución.

Una vez evaluadas las soluciones y actualizada la matriz de feromonas, las hormigas que componen la población crearán nuevas soluciones, eligiendo de forma probabilística cada nuevo elemento de la solución. Esta elección probabilística se realiza en función a la cantidad de feromonas.

Al basarse en un proceso natural que encuentra caminos óptimos evitando obstáculos, este algoritmo es idóneo y ampliamente reconocido a la hora de tratar el problema TSP.



En nuestro caso, la implementación obtenida del modelo de lenguaje es el representado en el pseudocódigo 6.

---

**Algoritmo 6** Colonia de hormigas

---

```
1: mejor_solucion ← generar_permutacion()
2: mejor_coste ← evaluar_solucion(mejor_solucion)
3: matriz_feromonas ← matriz_vacia

4: mientras tiempo < tiempo limite hacer
5:   para todos numero de ciudades hacer
6:     ciudades_no_visitadas ← lista de ciudades
7:     solucion ← lista vacia
8:     mientras ciudades_no_visitadas no este vacío hacer
9:       ciudad_actual ← eleccion_probabilistica()
10:      solucion ← agregar(ciudad_actual)
11:      ciudades_no_visitadas ← eliminar(ciudad_actual)
12:      coste ← evaluar_solucion(solucion)
13:      si coste < mejor_coste entonces
14:        mejor_coste ← coste
15:        actualizar_matriz_feromonas()

16: devolver mejor_solucion
```

---

La primera característica es el elemento poblacional. Se genera una población de hormigas, una para cada posible ciudad en el problema, donde cada hormiga va creando una solución aleatoria elemento a elemento, seleccionado cuál será la siguiente ciudad a visitar dentro de la lista de no visitadas a partir de una función probabilística.

Esta función asigna a cada posible ciudad a elegir una probabilidad en función de la cantidad de feromonas asignadas a esa solución y la distancia entre la última ciudad visitada y las posibles ciudades. Así, las opciones más probables serán aquellas ciudades más cercanas a la última visitada y que tengan una mayor cantidad de feromonas.

Así, se mantiene la intensificación, seleccionando con mayor probabilidad las mejores soluciones conocidas, y aún se mantiene un componente de diversificación al poder seleccionar elementos de la solución de forma aleatoria, permitiendo la exploración de nuevos áreas en el espacio de búsqueda.

## Resultados

La solución generada se evalúa y almacena si mejora la mejor solución hasta el momento y se actualiza la matriz de feromonas, inicializada a cero en un principio. A la hora de actualizar dicha matriz, la cantidad de feromonas depositadas en cada solución dependerá de la calidad de la solución.

Así, dentro de cada camino de la solución se le añade un valor de feromonas equivalente a  $coste^{-1}$ . Así, a menor coste, es decir, mejor solución, mayor cantidad de feromonas.

Tras analizar la implementación generada por el modelo de lenguaje, podemos afirmar que esta es correcta. Desde la generación de una población de soluciones al uso de una matriz de feromonas y función de selección probabilista, la implementación es correcta y luego veremos que además es la mejor de todos los algoritmos probados.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.6.

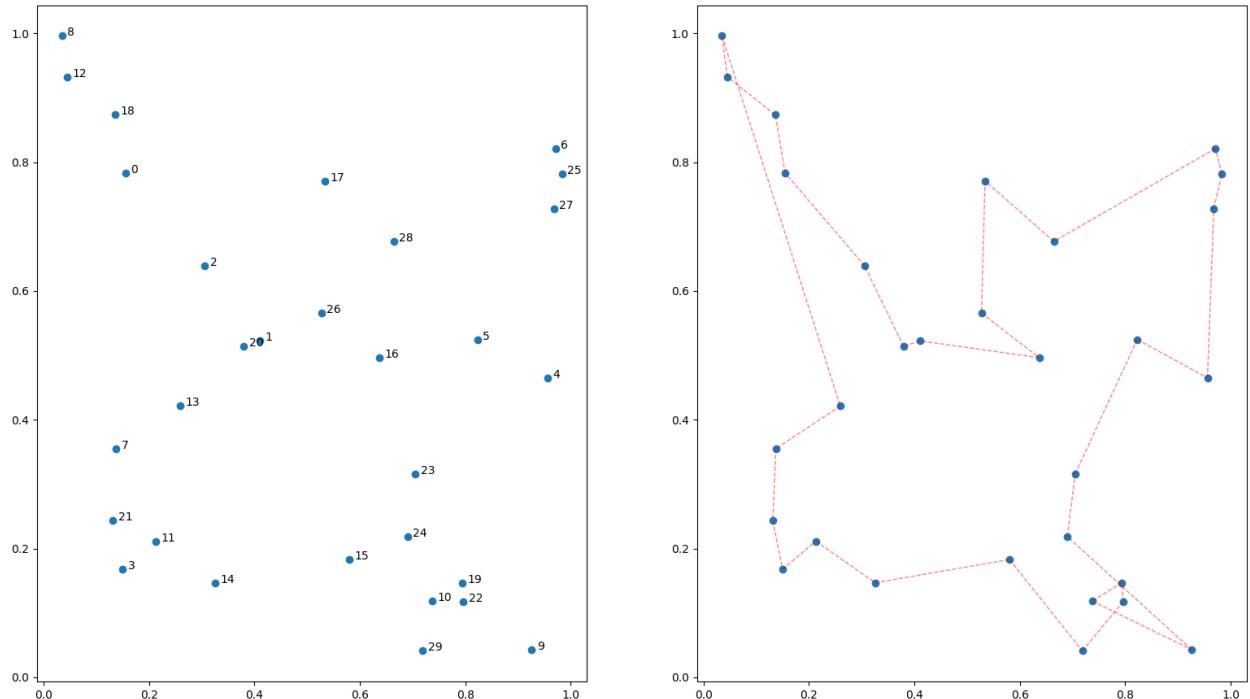


Figura 6.6: Ejemplo rendimiento Algoritmo colonia hormigas



### 6.1.7. Algoritmo de colonia de abejas

Este algoritmo, como su nombre indica, se inspira en el comportamiento de las abejas reales y como las colmenas, con sus distintos roles, funcionan para encontrar alimento. En específico, la base del algoritmo son los diferentes roles que las abejas pueden ejercer en este proceso de búsqueda.

El primer rol que participa en el proceso son las abejas exploradoras, que buscan en las proximidades de la colmena fuentes de alimento. Al encontrar una fuente de alimento, vuelven a la colmena, realizando una danza para atraer a otras abejas a la fuente de alimento.

Ahora, las abejas empleadas se centran en explotar esa fuente de alimento para la colmena. A medida que la fuente de alimento se degrada o se agota, estas abejas empleadas pasan a ser exploradoras, pasando a buscar soluciones nuevas o mejores que las ya encontradas.

Según la implementación teórica de la Colonia de Abejas Artificial, ideada originalmente por Dervis Karaboga en 2005 [55] este proceso natural se adapta, añadiendo nuevos roles y expandiendo los ya existentes.

Primeramente, las abejas empleadas no sólo se limitan a explotar a una fuente de alimento, es decir, una solución, si no que además explora una solución cercana, evaluando si es mejor que la ya obtenida. La abeja empleada mantiene como información la mejor solución evaluada y vuelve a la colmena. El rol de la abeja exploradora ahora se combina con el de la propia empleada.

Al volver a la colmena, la abeja danza esta vez para un nuevo rol, las abejas observadoras, quienes evalúan la calidad de cada solución representada en la danza y escogen cual explotar.



El proceso originalmente planteado por el autor sería:

1. Inicializar soluciones aleatorias (abejas exploradoras)
2. Evaluar las soluciones (abejas empleadas)
3. Selección de soluciones a explotar (abejas observadoras)
4. Explotación de soluciones
5. Detener el proceso de explotación de las fuentes agotadas por las abejas
6. Reemplazar las soluciones agotadas por nuevas aleatorias (abejas exploradoras)
7. Devolver la mejor solución encontrada

La implementación de este algoritmo obtenida del modelo de lenguaje se representa en el pseudocódigo 7.



---

**Algoritmo 7** Colonia de abejas

---

```
1: mejor_solucion ← generar_permutacion()
2: mejor_coste ← evaluar_solucion(mejor_solucion)

3: mientras tiempo < tiempo limite hacer
4:   abeja_empleada ← generar_vecino_two_otp(mejor_solucion)
5:   coste ← evaluar_solucion(abeja_empleada)
6:   si coste < mejor_coste entonces
7:     mejor_coste ← coste
8:     mejor_solucion ← abeja_empleada
9:   para todos abejas observadoras hacer
10:    abeja_observadorai ← generar_vecino_two_otp(mejor_solucion)
11:    coste ← evaluar_solucion(abeja_observadorai)
12:    si coste < mejor_coste entonces
13:      mejor_coste ← coste
14:      mejor_solucion ← abeja_observadora
15:    si probabilidad entonces
16:      abeja_exploradorai ← generar_vecino_two_otp(mejor_solucion)
17:      coste ← evaluar_solucion(abeja_exploradorai)
18:      si coste < mejor_coste entonces
19:        mejor_coste ← coste
20:        mejor_solucion ← abeja_exploradora

21: devolver mejor_solucion
```

---

En cuanto a la implementación obtenida, parece relacionar los conceptos teóricos propios de la implementación, pero falla en su implementación.

Primeramente, no se trabaja sobre una población inicial de abejas exploradoras, se parte de una única solución. Aunque se pueda considerar correcto, sería una implementación muy ineficiente, partiendo el algoritmo de una falta de diversificación. A partir de la solución inicial, la abeja empleada realiza su función correctamente, generando una solución vecina por “2-otp” y evaluándola.

Esta mejor solución obtenida por la abeja empleada es tratada por las abejas observadoras, generando cada abeja una nueva solución a partir de esta mejor solución. De todas las soluciones generadas, se almacena la mejor.



Ahora a partir de la mejor solución, una única abeja exploradora genera una nueva solución a partir una probabilidad. En este caso no se genera una nueva solución aleatoria, si no que se genera un vecino de la mejor solución. Esta solución se evalúa, almacenándose si es mejor.

Esta implementación es peculiar, pues parece reconocer los elementos característicos del algoritmo, implementa correctamente el funcionamiento de la abeja empleada, y reconoce la existencia de los demás roles. Ahora, empieza a trabajar sobre una única solución en lugar de una población, lo que rompe parte de la diversificación del algoritmo.

Además, al no existir una población de soluciones donde elegir, el funcionamiento teórico de las abejas observadoras queda inutilizado, al no tener solución donde elegir por probabilidad.

Aunque técnicamente sea incorrecto, el enfoque es interesante, pues realmente implementa un algoritmo de colonia de abejas artificial aplicado a una solución, no a una población.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.7.

## Resultados

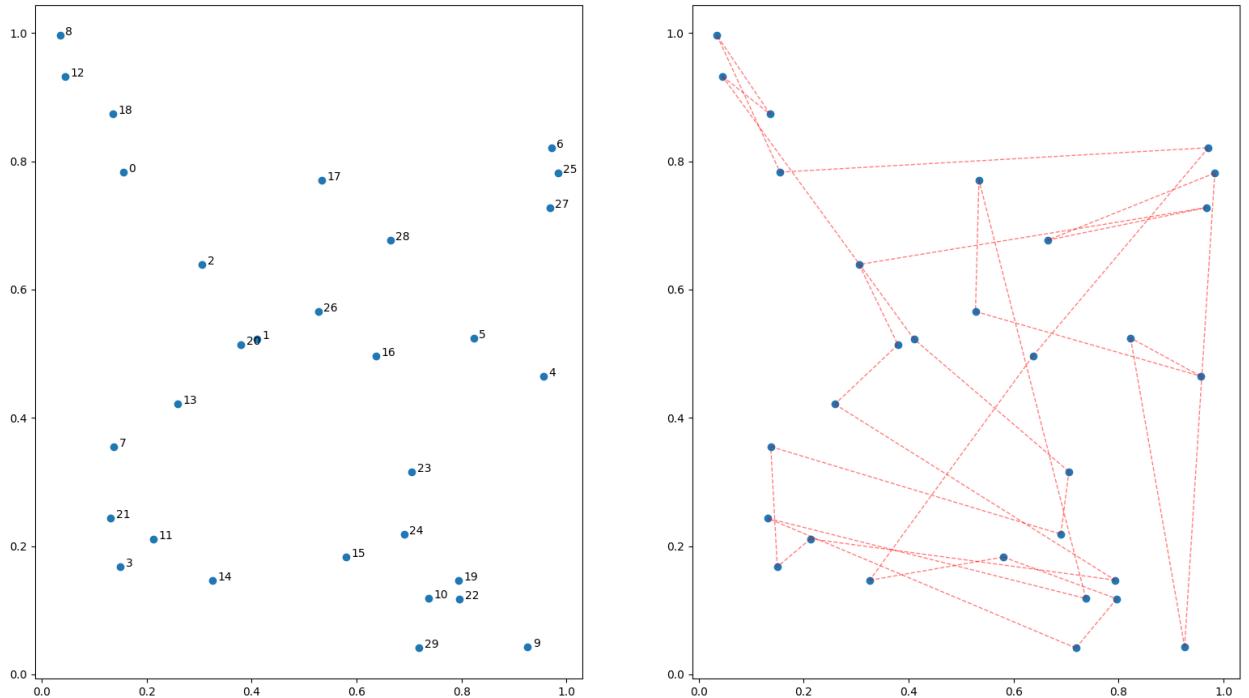


Figura 6.7: Ejemplo rendimiento Algoritmo colonia abejas

### 6.1.8. Optimización por enjambre de partículas

El algoritmo de optimización por enjambre de partículas (PSO por sus siglas en inglés) se basa, nuevamente, en las poblaciones animales, como lo pueden ser las colmenas de abejas o las bandadas de pájaros.

En este caso, el enfoque es distinto, ya que la solución se optimiza teniendo en cuenta dos enfoques a la misma vez, la propia solución obtenida por cada individuo de la población, y cómo éste se ve influido por el mejor elemento de la población. Así, cada individuo tiene un cierto grado de independencia, mientras que se ve influido por el comportamiento del conjunto de la población.

A la hora de tratar el algoritmo, según el autor original [56], cada elemento (partícula) de la población se rige por una velocidad, que dirige su “movimiento” por el espacio de búsqueda. Por cada iteración del algoritmo, cada una de las partículas recalculará una nueva velocidad, bien a partir de su mejor solución local, siendo este un elemento de diversificación, o bien a partir de la mejor solución global de entre todas las partículas, siendo este el elemento de intensificación.



Estos nuevos valores de velocidad llevan al cálculo de las nuevas posiciones de cada partícula, con las cuales se actualizan los valores de mejores soluciones locales y mejor solución global.

La implementación dada por el modelo de lenguaje se representa en el pseudocódigo 8.

---

**Algoritmo 8** Optimización por enjambre de partículas

---

```
1: mejor_solucion ← Nula
2: mejor_coste ← ∞
3: particulas ← generar_soluciones()
4: velocidades ← generar_velocidades()

5: mientras tiempo < tiempo límite hacer
6:   para todos particulasi hacer
7:     solucion ← particulai
8:     coste ← evaluar_solucion(solucioni)
9:     si coste < mejor_coste entonces
10:      mejor_coste ← coste
11:      mejor_solucion ← particulai
12:      actualizacion_velocidad(particulai)
13:      actualizacion_posicion(particulai)
14:      particulai ← actualizacion_particula(velocidadi, posicioni)
```

---

15: **devolver** *mejor\_solucion*

---

En primer lugar, este código genera una población de partículas, siendo cada una una permutación que representa una solución.

Además, se genera un vector de velocidades, que representa la velocidad de cada partícula. Esta velocidad no es más que un número aleatorio generado en un rango:  $\pm \text{velocidad maxima}$ , donde el valor máximo de velocidad un hiperparámetro.

Una vez generada la población, en cada iteración, se evalúa cada una de las soluciones. Para cada partícula, se actualiza su valor de velocidad. Para ello, se parte de un concepto de actualización cognitiva, basado en la mejor solución obtenida por la partícula, y el concepto de actualización social, basado en la mejor solución obtenida por la población.



Estos valores se representan por:

$$act\ cognitive = peso\ cognitivo \cdot rand() \cdot (mejor_i - pos\_actual_i)$$

Donde el peso cognitivo es un hiperparámetro, la función “rand()” representa un número aleatorio y “ $mejor_i$ ” representa el mejor valor obtenido por esa partícula en concreto. Así, se obtiene un valor numérico que cuantifica la mejora de la propia partícula con respecto a sí misma, en un ámbito local.

Y por:

$$act\ social = peso\ social \cdot rand() \cdot (mejor\ global - pos\_actual_i)$$

Donde el peso social es un hiperparámetro, la función “rand()” representa un número aleatorio y “mejor global” representa el mejor valor obtenido por esa la población, la mejor solución obtenida por la mejor partícula.

Así, se obtiene un valor numérico que cuantifica la mejora del conjunto de la población en un ámbito global.

Con estos valores, se actualiza el valor de velocidad de cada partícula, siguiendo la fórmula:

$$vel_i = (inercia \cdot vel_i) + act\ social + act\ cognitive$$

Donde la inercia es un hiperparámetro.

Tras esto, el valor de velocidad pasa por una proceso donde se restringen los valores en el rango  $\pm velocidad\ maxima$ , aunque técnicamente no se normalizan. Este nuevo valor de velocidad se le suma a la permutación actual que representa la partícula, generando así una nueva solución. Para asegurar que la solución es válida, la permutación pasa por el mismo proceso de restricción, esta vez en el rango del número de ciudades, y pasando los valores a números enteros.

Así, este proceso de actualización se repite para cada partícula de la población, durante un número de iteraciones, en nuestro caso, limitadas por tiempo.

## Resultados

Podemos afirmar que la implementación es correcta, pues se acerca a los conceptos de funcionamiento propios del algoritmo. La solución se actualiza a partir de dos componentes principales, uno global, representado por la mejor solución obtenida por la población, y otro local, representado por la mejor solución obtenida por el propio elemento en particular.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.8.

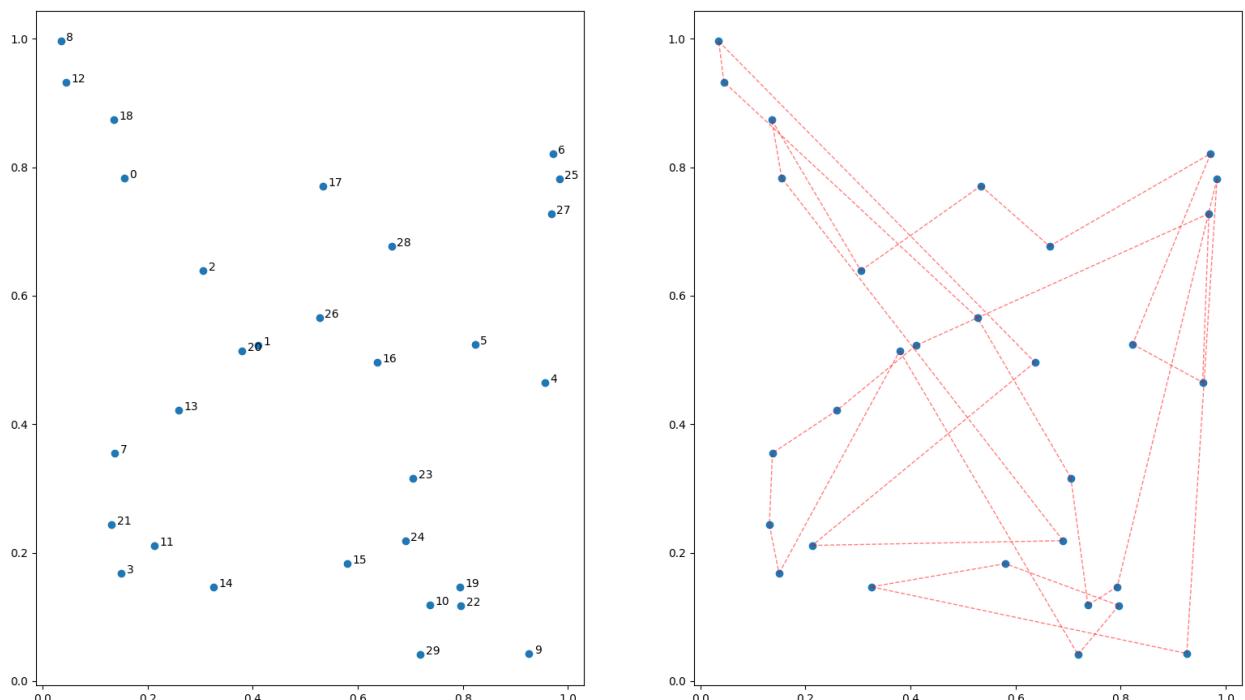


Figura 6.8: Ejemplo rendimiento Algoritmo enjambre partículas



### 6.1.9. Búsqueda gravitatoria

Este es otro algoritmo basado en procesos naturales, en este caso, basa su funcionamiento en la Ley de la gravedad y como la masa influye en esta [57].

Este algoritmo parte de una población de partículas o agentes. Cada agente se componen de distintos elementos: posición, masa inercial, masa gravitatoria activa y masa gravitatoria pasiva, siendo la posición lo que representa la solución. La posición de cada agente se actualiza en función de sus masas.

En este algoritmo, las mejores soluciones tienen más masa, y las peores, menos masa. Así, los agentes con mayor masa atraen a los demás agentes, siendo este el componente de intensificación. Por otro lado, los agentes de menor masa tienden a moverse libremente siempre que puedan, explorando nuevas áreas del espacio de búsqueda, siendo este el componente de diversificación.

La implementación obtenida por el modelo de lenguaje se representa en el pseudocódigo 9.

---

#### Algoritmo 9 Búsqueda gravitacional

---

```
1: mejor_solucion ← Nula
2: mejor_coste ← ∞
3: agentes ← generar_poblacion_agentes()

4: mientras tiempo < tiempo limite hacer
5:   para todos agente i hacer
6:     coste ← evaluar_solucion(agentei)
7:     si coste < mejor_coste entonces
8:       mejor_coste ← coste
9:       mejor_solucion ← agentei)
10:    para todos agente j hacer
11:      si agentei ≠ agentej entonces
12:        fuerzai ← fuerzas_ejercidas(agentei)
13:        agentei ← actualizar_agente(agentei, fuerzai)

14:    para todos agentes hacer
15:      agentei ← generar_vecino(agentei)

16: devolver mejor_solucion
```

---



Esta implementación comienza generando una población de agentes. En este caso, además de una permutación representada en la posición, cada agente tiene un valor de masa predeterminado para todos los agentes y un vector de velocidad con respecto a cada posible elemento de la solución, inicializado a cero.

A lo largo de las iteraciones, cada agente se evalúa. Tras esa evaluación, se actualiza el agente, a partir de las fuerzas ejercidas por el resto de agentes de la población.

La fuerza de atracción se calcula a partir de la fórmula  $F = \frac{G \cdot m_1 \cdot m_2}{r^2}$  donde  $m_i$  representa la masa de cada agente,  $r$  la distancia euclídea entre ambos agentes y la constante  $G$  es  $G = 6,67430 \cdot 10^{-11}$ .

Con esta fuerza, se actualiza la posición del agente. Se calcula la aceleración, que es  $a = \frac{\text{fuerza}}{\text{masa}}$ . Esta aceleración se usa para actualizar el vector de velocidad asociado al agente, y ese vector se le suma a la posición, ajustando los valores de posición a los límites del problema.

Para un único agente, este proceso de cálculo de fuerzas y actualización de posición se repite para cada agente restante de la población. Así, la posición se actualiza en base al efecto de cada uno de los agentes de la población sobre el agente original.

Este proceso se repite por cada agente y por cada iteración. Hasta este punto, el algoritmo está correctamente implementado, reconoce los elementos teóricos correctamente, aún simplificando el concepto de masa y añadiendo una velocidad. Además, estos conceptos se implementan correctamente.

Por último, durante cada iteración cada agente de la población pasa por un operador de vecindario. A partir de una probabilidad aleatoria, el agente genera una solución vecina bien por *2-opt* o *2-swap*. Este último proceso modifica todo el proceso del algoritmo, pues todo los cálculos realizados en función de la población, las masas y todo lo especificado anteriormente se ve desperdiciado por este cambio, al generar una solución vecina completamente distinta a la generada por el proceso.

## Resultados

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.9.

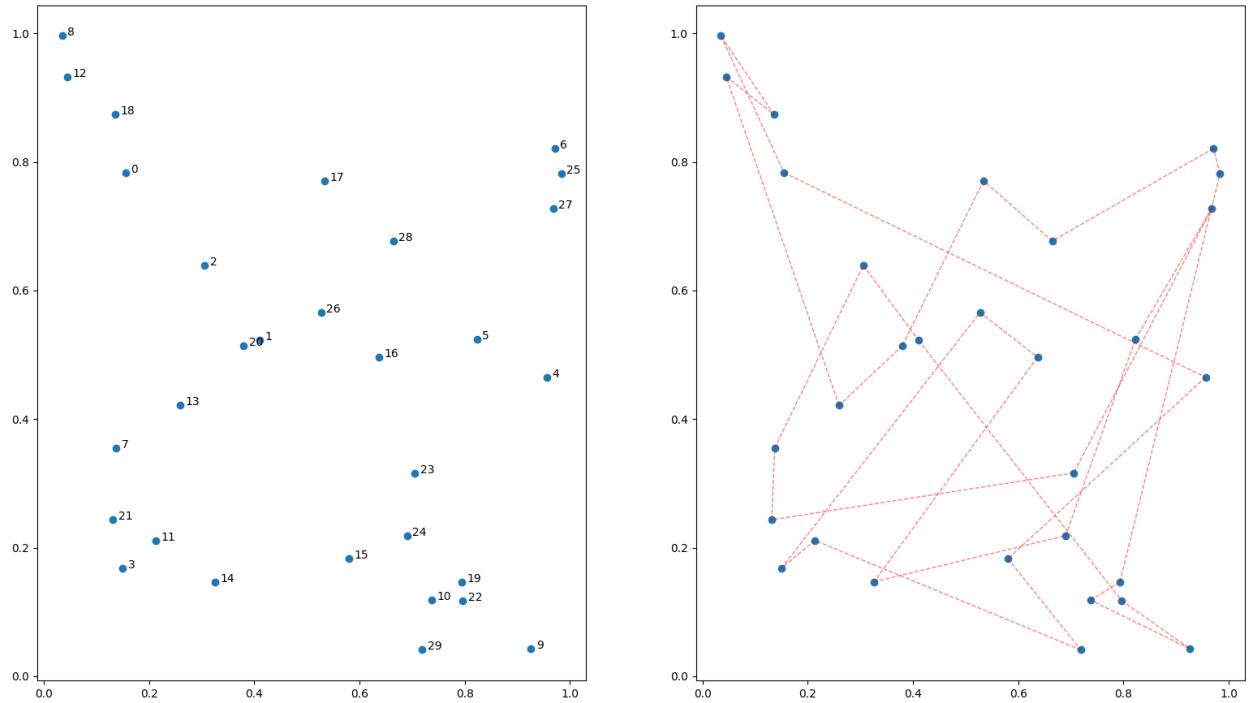


Figura 6.9: Ejemplo rendimiento Búsqueda gravitacional



### 6.1.10. Algoritmo de agujero negro

Este algoritmo se inspira en el funcionamiento natural de los agujeros negros [58] y en cómo su gran masa atrae a todo lo que se le acerque, sin posibilidad de escapatoria.

Así, este algoritmo es poblacional, donde cada solución de la población es llamada estrella. De esas estrellas, la solución de mejor coste será seleccionado como agujero negro. A su vez, todas las soluciones tienen un componente de movimiento a través del cual se mueven hacia el agujero negro de forma probabilística.

En cada iteración, las estrellas se van moviendo y acercándose al agujero negro. Las estrellas que más se acercan son absorbidas por el agujero negro, desapareciendo. Por cada estrella que desaparece, una nueva estrella aparece, representando una nueva solución completamente aleatoria.

Así, se consigue el efecto de intensificación, representado en el agujero negro, y pudiendo centrar todo el esfuerzo de diversificación en el espacio de búsqueda. Las estrellas que exploran las proximidades del agujero negro lo hacen por un tiempo limitado hasta ser absorbidas por este. Al generarse nuevas estrellas, estas exploran una porción completamente aleatoria del espacio de búsqueda, lo que permite un gran componente de diversificación.

La implementación generada por el modelo de lenguaje es la representada en el pseudocódigo 10.



---

**Algoritmo 10** Algoritmo agujero negro

---

```
1: soluciones ← lista de soluciones aleatorias
2: mejor_coste ← evaluar_soluciones(soluciones)
3: mientras tiempo < tiempo límite hacer
4:   agujero ← mejor_solucion(soluciones)
5:   para todos soluciones i hacer
6:     atraccion ← calcular_atraccion(agujeroi)
7:     coste ← evaluar_solucion(solucioni)
8:     si numero_aleatorio < atraccion entonces
9:       solucion ← generar_vecino(solucioni)
10:      si coste < mejor_coste entonces
11:        mejor_coste ← coste
12:        mejor_solucion ← solucioni
13: devolver mejor_solucion
```

---

Esta implementación comienza generando una población de soluciones aleatorias. Se evalúa el conjunto de soluciones y se selecciona la mejor de las soluciones, lo que podemos interpretar como la selección del agujero negro.

Para todas las demás soluciones, se calcula la atracción hacia el agujero negro seleccionado, y si este valor de atracción supera cierto umbral aleatorio, esta solución se modifica a través de un operador de vecindario seleccionado aleatoriamente entre *2-opt* y *2-swap*. Al acabar el proceso se selecciona un nuevo agujero negro.

Esta implementación genera correctamente ciertos conceptos principales del algoritmo, con una diferencia notable. El agujero negro ya no absorbe estrellas para generar otras completamente nuevas y aleatorias, este proceso se ve sustituido por la sustitución por una solución vecina.

Así, esta implementación pierde un importante componente de diversificación, al no realizar grandes movimientos por el espacio de búsqueda. Además, no se tiene ninguna noción de movimiento por parte de las soluciones. Aún así, la sustitución por solución vecina puede considerarse como un movimiento dentro del vecindario.

En términos generales, la implementación es correcta, pero muy falta en diversificación.

## Resultados

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.10.

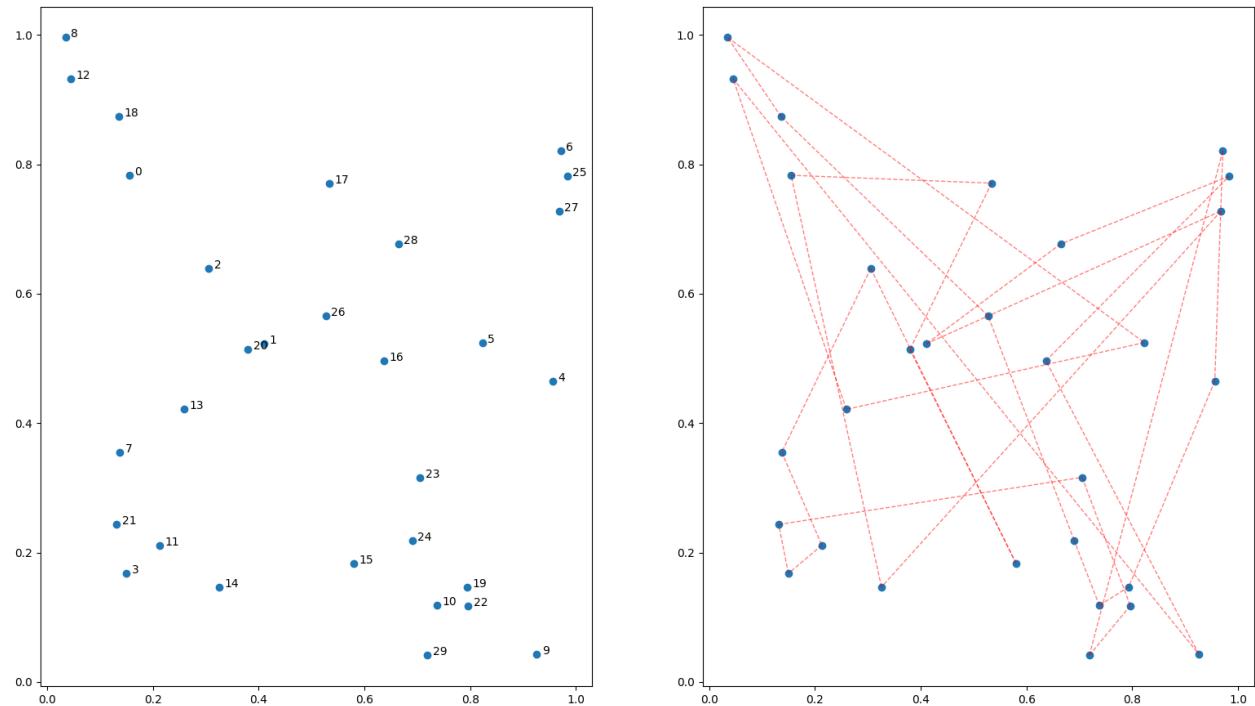


Figura 6.10: Ejemplo rendimiento Algoritmo agujero negro



### 6.1.11. Búsqueda dispersa

El algoritmo de búsqueda dispersa [59] es una metaheurística enfocada en encontrar soluciones para problemas de optimización combinatoria.

En primer lugar, se inicia con la generación de un conjunto inicial de soluciones diversas que en principio no tienen que ser necesariamente óptimas.

A partir de ese conjunto o población inicial se obtiene un conjunto de referencia. La mitad de soluciones de ese conjunto serán las mejores soluciones de la población, y la otra mitad del conjunto se compondrá de soluciones que disten de las ya seleccionadas, siguiendo un criterio de diversidad.

Una vez se tiene el conjunto de referencia, estas soluciones se combinan entre sí, manteniendo un componente de intensificación al operar con las mejores soluciones a priori, y diversificación, combinándolas con soluciones completamente distintas.

Estos nuevos conjuntos, junto con el conjunto de referencia, se intentan mejorar a partir de un proceso generalmente heurístico. Esta mejora se aplica tanto al conjunto de referencia original como a los nuevos subconjuntos generados por combinación. El método de mejora más común suele ser la búsqueda local.

Así, el algoritmo itera devolviendo la mejor solución obtenida.

La implementación generada por el modelo de lenguaje es el representado en el pseudocódigo 11.

---

#### Algoritmo 11 Búsqueda dispersa

---

```
1: mejor_solucion ← generar_permutacion()
2: mejor_coste ← evaluar_solucion(solucion)

3: mientras tiempo < tiempo limite hacer
4:   perturbacion ← pertubar_solucion(mejor_solucion)
5:   nueva_solucion ← busqueda_local(perturbacion)
6:   coste ← evaluar_solucion(nueva_solucion)
7:   si coste < mejor_coste entonces
8:     mejor_coste ← coste
9:     mejor_solucion ← nueva_solucion

10: devolver mejor_solucion
```

---



En el caso de la implementación dada por el modelo de lenguaje, parece no reconocer los elementos clave del algoritmo ni es capaz de implementarlos correctamente.

En primer lugar, no reconoce el punto de partida, siendo este una población de soluciones. Tampoco se llega a reconocer la formación de conjuntos de referencia. Al no tener un conjunto de soluciones de referencia, tampoco se pueden generar combinaciones de soluciones, y por lo tanto, se pierde toda la base de intensificación y diversificación.

En su lugar, se parte de una única solución. En este caso, en lugar de un proceso de combinación, se ha generado un proceso de perturbación de la solución. El método usado ha sido el previamente visto “2-swap”, que selecciona dos índices aleatorios en la solución y luego intercambia los elementos ubicados en esos índices.

Esta solución perturbada se mejora a través de un proceso de búsqueda local. El proceso de mejora aplicado está correctamente implementado.

Visto esto, podemos afirmar que la implementación obtenida es incorrecta, ya que de los elementos que caracterizan el algoritmo sólo consigue implementar la mejora de las solución a través de una búsqueda local.

## Resultados

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.11.

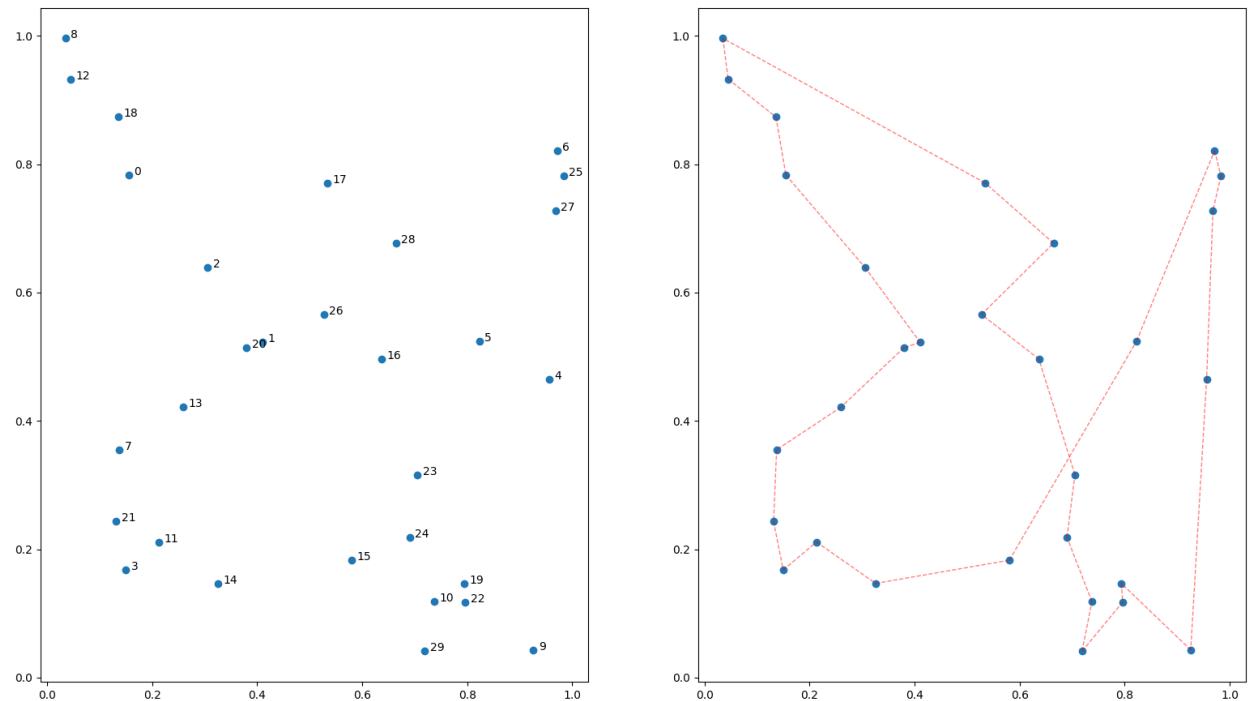


Figura 6.11: Ejemplo rendimiento Búsqueda dispersa



### 6.1.12. Algoritmo cultural

Los algoritmos culturales son una familia de algoritmos poblacionales, basados en fenómenos sociológicos, ideados originalmente en 1994 [60].

Basándose en la evolución de las poblaciones comunes a estos tipos de algoritmos, diferencia dos evoluciones simultáneas: la evolución “genética”, heredada de padres a hijos, y una evolución cultural, que es el conocimiento adquirido por los individuos a través de las generaciones.

Este conocimiento cultural es compartido, recabado y accesible por todos los miembros de la población. A su vez, cada individuo interpreta y asimila este conocimiento cultural a su propia manera, influyendo cada individuo en el conocimiento común.

Estos algoritmos funcionan con dos *espacios* a la misma vez, el espacio de población, que representa a la población y como esta evoluciona de padres a hijos, y el espacio de creencias, que es el conocimiento común al conjunto de la población.

La implementación más simplificada de estos tipos de algoritmos debe generar una población inicial y un espacio de creencias. Al evaluar la población, se genera un grupo selecto de individuos de la población, que se usan para actualizar el espacio de creencias.

A partir de este espacio de creencias se generan nuevos elementos de la población, que son variaciones de los individuos originales modificados por la **función de influencia**. Esta función ejerce presión sobre los nuevos individuos, para que estos se acerquen a las creencias almacenadas por la población. Por último, los mejores nuevos individuos de la población reemplazan a los peores.



La implementación dada por el modelo de lenguaje se representa en el pseudo-código 12.

---

**Algoritmo 12** Algoritmo cultural

---

```
1: poblacion  $\leftarrow$  generar_soluciones()
2: mejor_coste  $\leftarrow \infty$ 

3: mientras tiempo < tiempo límite hacer
4:   para todos soluciones de la población hacer
5:     costei  $\leftarrow$  evaluar_solucioni
6:     si costei < mejor_coste entonces
7:       mejor_coste  $\leftarrow$  costei
8:       mejor_solucion  $\leftarrow$  solucioni
9:   poblacion_elite  $\leftarrow$  mejores_soluciones(poblacion)
10:  para todos soluciones elite hacer
11:    poblacion  $\leftarrow$  aprendizaje_individual(elitei)
12:    poblacion  $\leftarrow$  aprendizaje_social(poblacion_elite)
13:    poblacion  $\leftarrow$  llenar_poblacion(poblacion)
14: devolver mejor_solucion
```

---

Esta implementación empieza por generar una población de individuos válidos. Esta población se evalúa y se selecciona un conjunto con los mejores individuos de la población.

A partir de esta selección elitista se genera una población completamente nueva. Esta nueva población, que reemplazará a la original, se compone en primer lugar de los elementos de la selección elitista. Ahora, para cada uno de esos individuos se genera un nuevo hijo por un proceso llamado *aprendizaje individual*, que no es más que generar una solución vecina a través de una *2-swap*.

Luego, se aplica un proceso de *aprendizaje social*. Cada elemento de la selección elitista se compara con otro aleatorio del conjunto, reemplazando este por un vecino generado por aprendizaje individual siempre el individuo original sea peor que con el cual se compara.

Al final, esta nueva población se rellena con soluciones completamente aleatorias para después reemplazar completamente a la población original.

## Resultados

La implementación es correcta en términos generales, pues consigue reconocer e implementar ambos tipos de evolución basadas en las creencias comunes, sin perder la propia herencia a través de generaciones, en este caso representada por el elitismo.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.12.

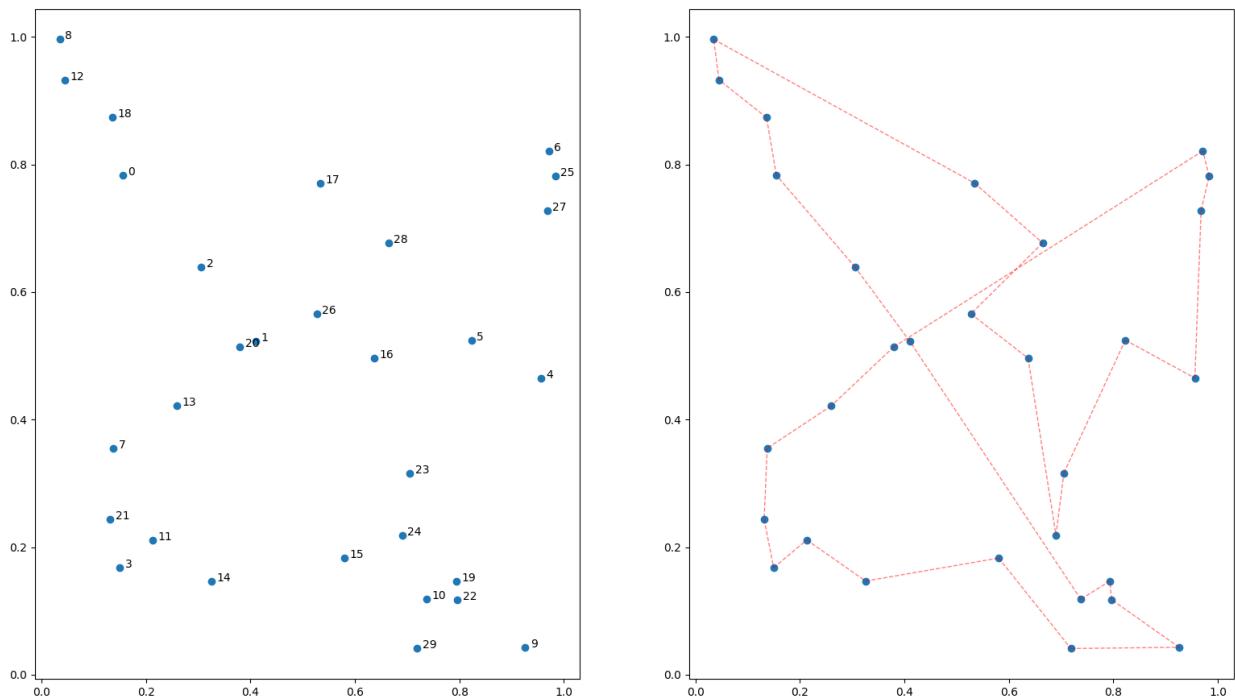


Figura 6.12: Ejemplo rendimiento Algoritmo cultural



### 6.1.13. Búsqueda con vecindario variable

Este algoritmo es otra adaptación del algoritmo básico de búsqueda local.

Como se ha explicado anteriormente, el algoritmo de búsqueda local es muy limitado, pues al no tener ningún mecanismo de diversificación más allá de un inicio aleatorio y la posibilidad de reinicios, la búsqueda local tiende converger hacia óptimos locales, sin ninguna posibilidad de evadirlos.

El óptimo local con respecto a un vecindario no tiene por qué serlo con respecto a otra estructura de vecindario, pero la solución óptima global lo será para cualquier estructura de vecindario. A partir de esta premisa, nace el algoritmo de la búsqueda con vecindario variable.

La búsqueda con vecindario variable no es más que un algoritmo de búsqueda local que sistemáticamente varía la estructura del vecindario, generalmente al toparse con un óptimo local [61]. Así, los vecindarios pueden ser representados no como un conjunto de posibles soluciones vecinas, si no como las operaciones que puedan generar dicho conjunto de soluciones vecinas.

La estrategia del algoritmo es sencilla, ir realizando una búsqueda local, ya sea por primer o mejor vecino, y al encontrar óptimo local, cambiar el operador de vecindario. Al cambiar el operador, se genera un nuevo conjunto de soluciones vecinas, lo que abre la posibilidad a escapar de este óptimo local.

La implementación obtenida por el modelo de lenguaje es la representada en el pseudocódigo 13.

**Algoritmo 13** Búsqueda con vecindario variable

---

```
1: solucion  $\leftarrow$  generar_permutacion()
2: mejor_coste  $\leftarrow$  evaluar_solucion(solucion)  
  
3: mientras tiempo < tiempo límite hacer
4:   operador  $\leftarrow$  seleccion_operador_vecindario()
5:   mientras operador de vecindario hacer
6:     para todos numero_ciudadesi hacer
7:       para todos numero_ciudadesj hacer
8:         solucion  $\leftarrow$  generar_vecino(solucion)
9:         coste  $\leftarrow$  evaluar_solucion(solucion)
10:        si coste < mejor_coste entonces
11:          mejor_coste  $\leftarrow$  coste
12:          mejor_solucion  $\leftarrow$  solucion
13:        sino operador  $\leftarrow$  actualizar_operador_vecindario()  
  
14: devolver mejor_solucion
```

---

El algoritmo generado parte correctamente de una solución aleatoria, que es evaluada y de la cual se genera un vecindario, partiendo de un operador de vecindario concreto, en este caso *2-opt*. Sobre esta solución se aplica una búsqueda local por mejor vecino, evaluando la totalidad del vecindario generado e iterando a continuación sobre la mejor solución encontrada hasta el momento.

En el momento en que ninguna solución del vecindario mejora la solución de la cual se parte, se cambia el operador de vecindario, en este caso, a un anteriormente explicado *2-swap*.

Partiendo de la mejor solución obtenida en el anterior proceso de búsqueda se repite el proceso, en este caso con el nuevo operador de vecindario. Al volver a llegar a un óptimo, el proceso finaliza, devolviendo la mejor solución obtenida al tratar con este último operador. Así, este proceso de trabajo con dos operadores de vecindarios se reinicia y repite hasta que se llegue a un tiempo límite.

En la implementación se reconocen los elementos claves del algoritmo, el cómo y el cuándo variar de vecindarios, cambiando de operador de vecindario al no mejorar la solución. Además, al cambiar de operador de vecindario, la búsqueda se reinicia partiendo de la última solución explorada con el operador de vecindario previo.

## Resultados

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.13.

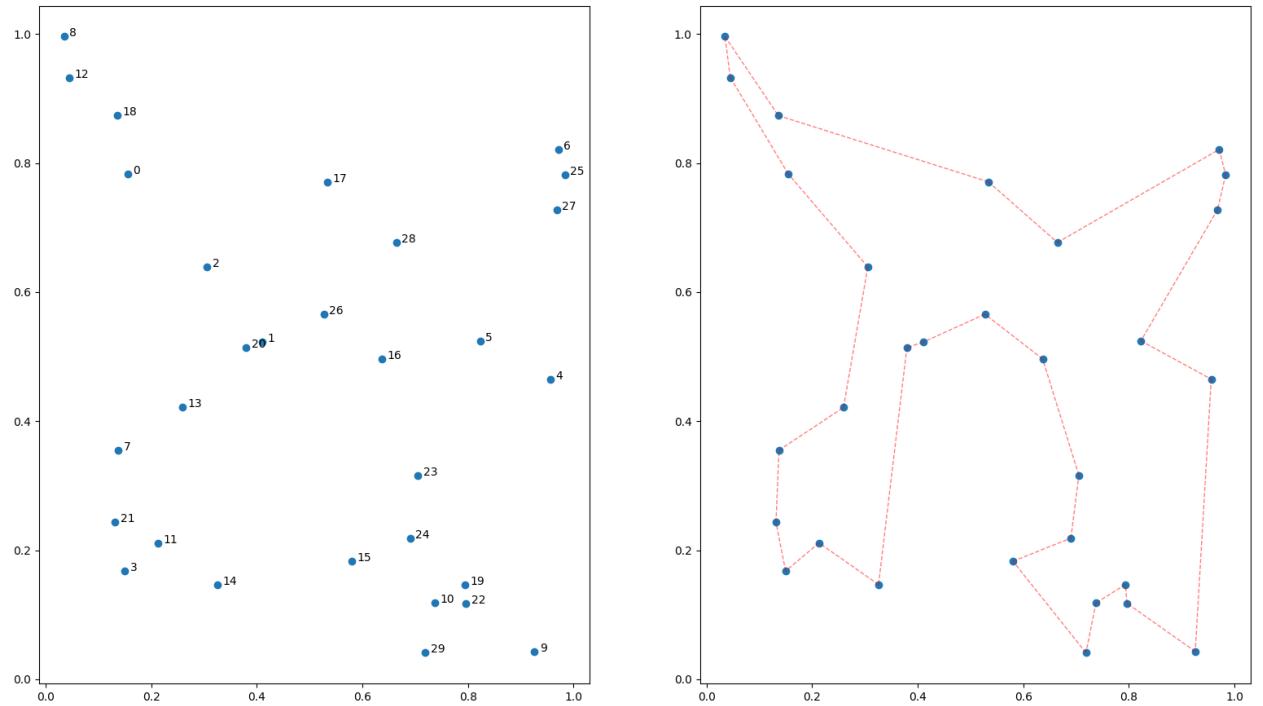


Figura 6.13: Ejemplo rendimiento Búsqueda vecindario variable



### 6.1.14. Búsqueda armónica

El algoritmo de búsqueda armónica es otro algoritmo poblacional, inspirado en el proceso de improvisación musical.

Propuesto inicialmente por Geem en 2001 [62], este algoritmo simula el proceso por el cual los músicos y compositores buscan producir armonías agradables, estando estas determinadas por los estándares estéticos y auditivos que se contemplen.

A la hora de componer música, un músico realizará una de las siguientes acciones:

- Interpretar una melodía ya conocida con anterioridad
- Componer una melodía completamente nueva, basándose en sus conciertos previos
- Componer e interpretar melodías parecidas a las ya conocidas, adaptándolas a los estándares deseados

En el desarrollo teórico propuesto por los autores, estos elementos se denominan como uso de memoria armónica, aleatoriedad y ajuste de tono, respectivamente.

Al final, estos procesos pueden interpretarse como los elementos propios de un algoritmo metaheurístico, representando los elementos anteriores procesos de intensificación, diversificación y generación de soluciones vecinas respectivamente. Estos conceptos se añan en uno sólo para generar nuevas armonías.

Partiendo de una población de soluciones aleatorias, la conocida como memoria armónica, se van improvisando nuevas armonías, es decir, nuevas soluciones. Estas se generan por el proceso de ajuste de tono, donde a partir de una solución ya existente en la memoria armónica, esta se modifica en base a hiperparámetros y ciertos elementos de aleatoriedad. Un ejemplo de ajuste tono básico podría ser:

$$Armonia_i = Armonia_i + r \cdot bw$$

Donde la armonía original se modifica añadiéndole un valor aleatorio  $r$  y un valor de ancho de banda arbitrario. Estas nuevas armonías se mantienen siempre que mejoren la solución de la que parten, reemplazando a la peor solución existente en la memoria.



La implementación dada por el modelo de lenguaje se representa en el pseudo-código 14.

---

**Algoritmo 14** Búsqueda armónica

---

```
1: armonias ← generar_poblacion()
2: mejor_solucion ← seleccionar_solucion(armonias)
3: mejor_coste ← evaluar_solucion(solucion)

4: mientras tiempo < tiempo limite hacer
5:   solucion ← generar_permutacion
6:   coste ← evaluar_solucion(solucion)
7:   si coste < mejor_coste entonces
8:     mejor_coste ← coste
9:     poblacion ← actualizar_solucion(solucion)

10: devolver mejor_solucion
```

---

En este caso, el modelo de lenguaje no es capaz de reconocer los elementos propios del algoritmo. Esta implementación es una genérica, que se repite cuando el modelo no es capaz de reconocer que debe de hacer este algoritmo.

Así, esta implementación genera una población de soluciones aleatorias. Por cada iteración, se genera una nueva solución completamente aleatoria, se evalúa y en caso de mejorar la mejor solución hasta el momento, reemplaza a la solución previa. Además, aunque se genera una población de soluciones no se trabaja con ella, sólo se trabaja con una única solución de esta población.

A efectos prácticos, este algoritmo no es más que una búsqueda aleatoria con un elemento poblacional. La población no evoluciona por ningún proceso concreto, sólo por un proceso de búsqueda aleatoria.

Obviamente, esta implementación es completamente errónea con respecto al algoritmo solicitado.

## Resultados

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.14.

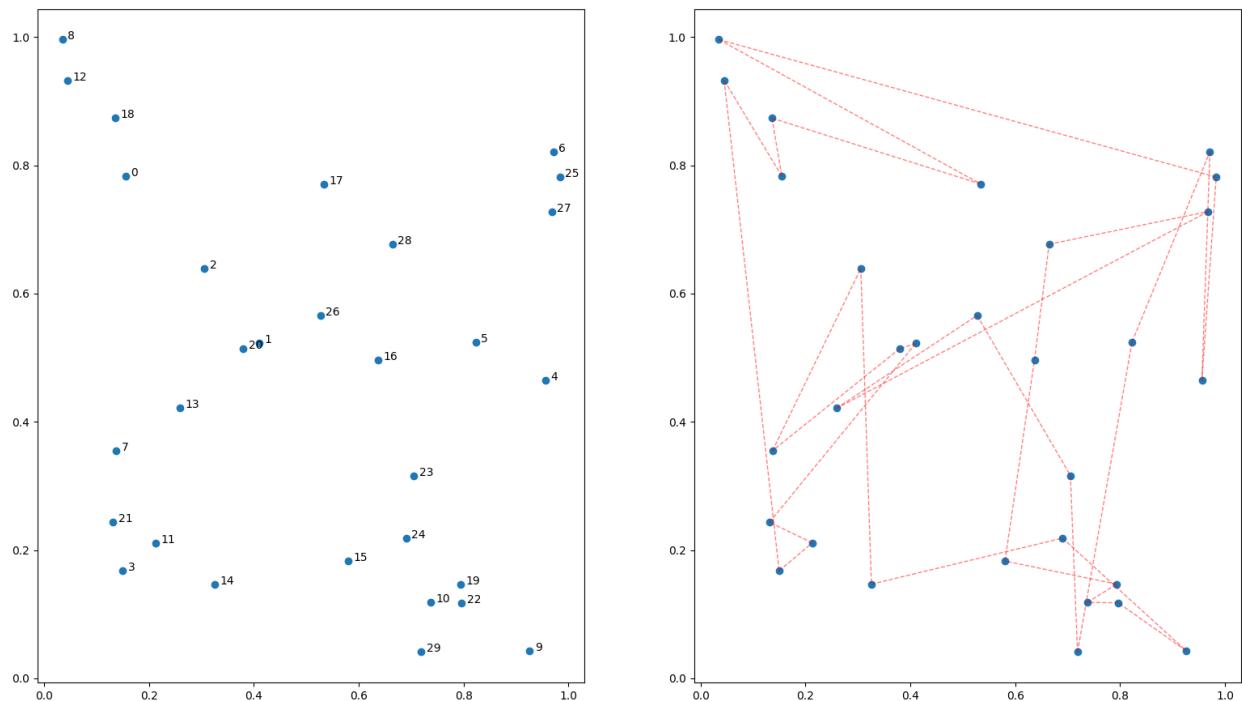


Figura 6.14: Ejemplo rendimiento Búsqueda armónica



### 6.1.15. Algoritmo imperialista competitivo

Este algoritmo se basa en los comportamientos sociales y humanos, reflejados en los comportamiento de las propias naciones y cómo estas interactúan entre sí, en este caso, a través del imperialismo [63].

Es un algoritmo poblacional, partiendo de un conjunto de soluciones aleatorias que representan países. Esta población es evaluada, y se divide en dos grupos: los mejores países de la población son considerados imperialistas y los demás, países a colonizar.

En cada iteración se pasa por distintos procesos. Se empieza por la *asimilación*, donde las peores soluciones (los países a colonizar) se mueven en dirección a los imperialistas. Tras esto, se pasa por un proceso de *revolución*, donde algunos de los países colonizados serán reemplazados por nuevos países completamente nuevos y aleatorios.

Al tener nuevos países, aquellos que sean mejores tienen la posibilidad de *reemplazar* a un imperialista, pasando este a ser un país a colonizar.

Una vez establecidos los nuevos imperios, estos competirán por su supervivencia, tomando posesión de las colonias de otros imperios. El poder de los imperios se calcula por la fórmula:

$$Poder_i = \text{coste}(\text{imperialista}_i) + \xi \text{ media}(\text{coste}(\text{colonias}_i))$$

Donde  $\xi$  es hiperparámetro que representa la influencia de las colonias sobre el imperio.

Basándose en este coste, se da el proceso de *competición imperialista*. Generalmente, en este proceso se selecciona la peor colonia del imperio más débil para que se compita por ella. Aunque la elección de a qué imperio pertenecerá la colonia sea probabilística, está influenciada por el poder de cada imperio, siendo más probable que acabe perteneciendo al mejor imperio.

Así, este proceso se va repitiendo, donde los imperios más débiles colapsan al quedarse sin colonias, pasando a ser una colonia más a repartir entre los imperios restantes. El algoritmo acabará al quedar un único imperio al que pertenezcan todas las posibles colonias.



La implementación dada por el modelo de lenguaje es el representado en el pseudocódigo 15.

---

**Algoritmo 15** Algoritmo imperialista competitivo

---

```
1: paises ← lista de permutaciones
2: mejor_solucion ← seleccionar_mejor_pais(paises)
3: mejor_coste ← evaluar_solucion(mejor_solucion)  
  
4: mientras tiempo < tiempo límite hacer
5:   lista_imperialistas ← seleccion_mejores(paises)
6:   lista_colonizados ← seleccion_peores(paises)
7:   para todos colonizados hacer
8:     imperialista ← seleccion_aleatoria(lista_imperialistas)
9:     pais ← mutacion(imperialista)
10:    coste ← evaluar_solucion(pais)
11:    si coste < mejor_coste entonces
12:      mejor_coste ← coste
13:      mejor_solucion ← pais
14:    si coste < coste(imperialista) entonces
15:      imperialistai ← pais  
  
16: devolver mejor_solucion
```

---

El algoritmo comienza generando una población de soluciones aleatorias. Esta población se divide en dos, un grupo de países imperialistas formado por las mejores soluciones, y un conjunto de países colonizados formado por las peores soluciones.

Para cada país a colonizar, se selecciona al azar uno de los posibles países imperialistas. A partir de este imperialista, se genera un nuevo país a partir de una mutación de la solución que representa el imperialista. En este caso, la única mutación realizada es un *2-opt*, donde se intercambian dos elementos de la solución. Este nuevo país reemplazará al imperialista en caso de ser mejor que este. El proceso se repite por cada posible país de la lista de colonizados, acabando así una iteración.

En cada iteración, se vuelve a generar la lista de países imperialistas y colonizados, repitiendo el proceso de generación de nuevas soluciones.

## Resultados

En términos generales, la implementación es errónea, pues no realiza las acciones propias del algoritmo. Este algoritmo se puede considerar una buen algoritmo imperialista, pero en ningún momento se realiza una competición imperialista. Los imperios no compiten ni se acaban destruyendo unos a otros. De la misma manera, los países colonizados no son parte de un imperio liderado por un país imperialista.

Aún así, la implementación mantiene los conceptos de imperialismo, generando nuevas soluciones a partir de un número establecido de países imperialistas. Se parte de un número de imperialistas, aunando diversificación e intensificación. Además, esta diversificación aumenta al generar nuevas soluciones cercanas al espacio de búsqueda ya conocido por los imperialistas.

Podemos afirmar que la implementación reconoce los elementos básicos imperialistas, pero falla al generar una competición imperialista, por lo cual es una implementación incorrecta.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.15.

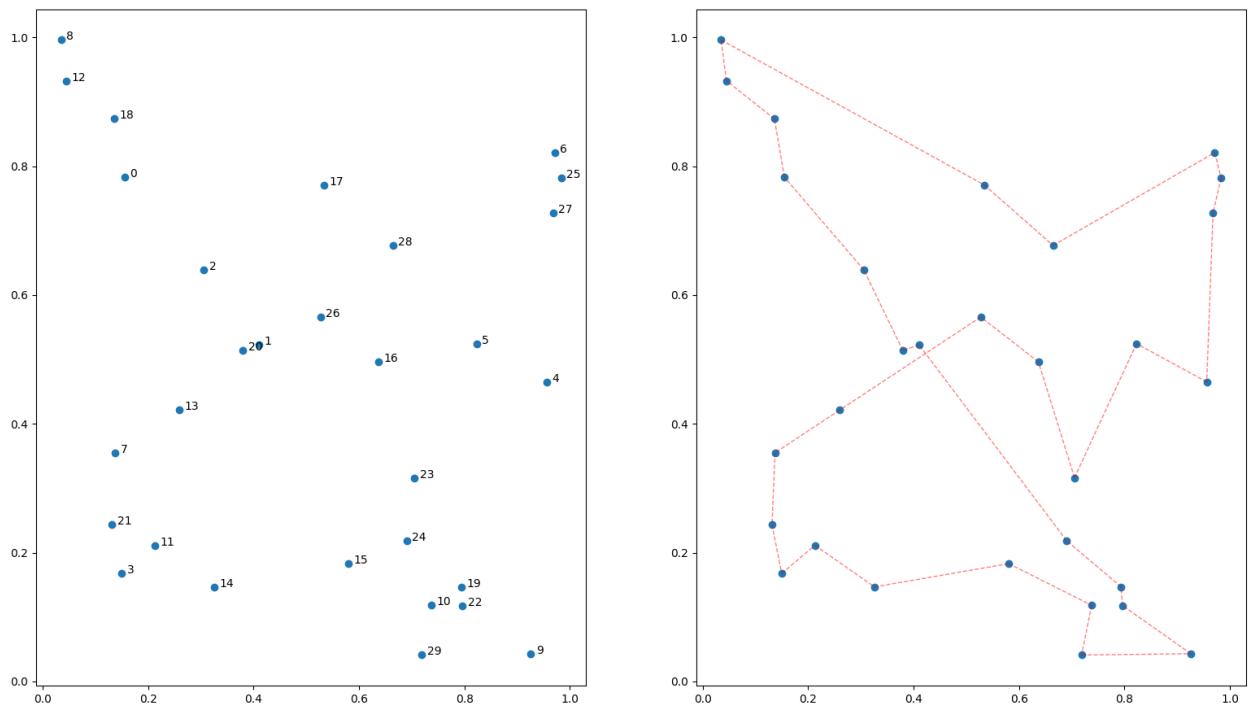


Figura 6.15: Ejemplo rendimiento Algoritmo imperialista competitivo



### 6.1.16. Algoritmo de optimización caótica

Los algoritmos de optimización caótica constituyen una extensa familia que se fundamenta en componentes caóticos, abarcando desde algoritmos propios hasta variantes de metaheurísticas ya establecidas a las cuales se les incorporan elementos caóticos.

Se inspiran en los sistemas dinámicos caóticos, que exhiben comportamientos impredecibles pero deterministas, generando secuencias caóticas de números en lugar de números aleatorios.

El uso de secuencias caóticas en lugar de número semi-aleatorios parece ser una estrategia óptima para mejorar algoritmos metaheurísticos convencionales y el como escapan estos de los óptimos locales [64].

La implementación más básica del algoritmo consiste en la generación de una población de soluciones candidatas. Las soluciones se generan mediante un vector de secuencias caóticas  $\gamma_i$ . Para ello, cada elemento de  $\gamma_i$  se mapea linealmente a un intervalo específico dentro de la región factible.

En cada iteración, se genera un nuevo vector de secuencias caóticas usando un mapa caótico, donde cada componente se genera usando una fórmula específica. El óptimo local se va actualizando mientras se converge.

La implementación generada por el modelo de lenguaje es la representada en el pseudocódigo 16.

---

#### Algoritmo 16 Algoritmo de optimización caos

---

```
1: solucion ← generar_permutacion()
2: mejor_coste ← evaluar_solucion(solucion)

3: mientras tiempo < tiempo limite hacer
4:   solucion ← perturbacion(solucion)
5:   coste ← evaluar_solucion(solucion)
6:   si coste < mejor_coste o criterio de aceptación entonces
7:     mejor_coste ← coste
8:     mejor_solucion ← solucion

9: devolver mejor_solucion
```

---

## Resultados

En este caso, el modelo de lenguaje no es capaz de reconocer ninguno de los elementos característicos. En este caso, se trata de otra búsqueda aleatoria , seleccionando las mejores nuevas soluciones o una peor según un criterio probabilístico. Eso puede llevar a devolver una solución peor a la mejor obtenida hasta el momento.

Las soluciones de la población se generan de forma aleatoria, sin ningún componente caótico, ni por vectores de secuencia ni por la generación de un mapa caótico.

La implementación es incorrecta, tanto como algoritmo caótico como algoritmo en general, pues ni siquiera garantiza devolver la mejor solución obtenida durante todo el proceso.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.16.

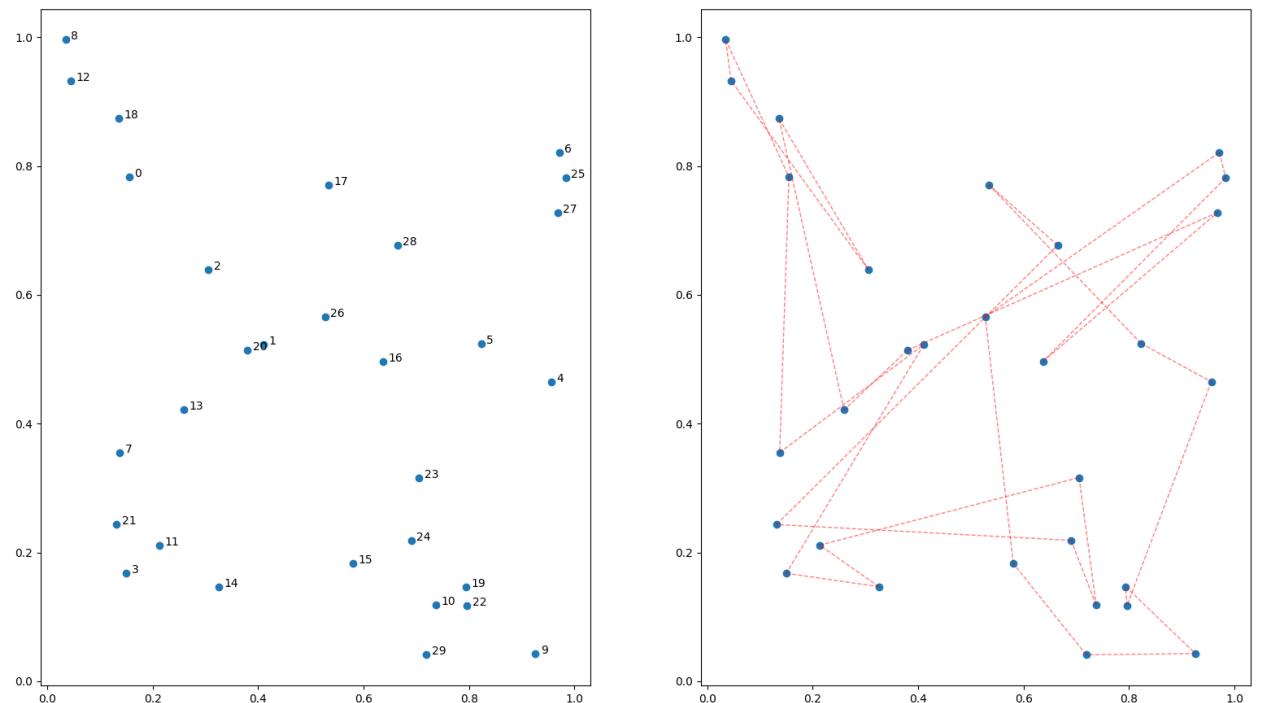


Figura 6.16: Ejemplo rendimiento Algoritmo optimización caótica



### 6.1.17. Sistema inmune artificial

Este algoritmo es otro inspirado en procesos naturales, en este caso, en el funcionamiento del sistema inmune propio de animales y humanos [65]. Dentro de la familia de metaheurísticas de sistema inmune artificial, buscaremos la implementación de un algoritmo de Selección Clonal [66].

El algoritmo se compone de tres elementos principales. En primer lugar, las células del sistema inmune (anticuerpos), que no son más que representaciones de soluciones dentro del espacio de búsqueda. Por otro lado, están los antígenos, elementos que se buscan compensar a través de los anticuerpos propios del sistema inmune. Esto se puede entender como los resultados que se buscan obtener. Por último existe el concepto de afinidad, que no es más que la distancia euclídea entre anticuerpos y antígenos.

El algoritmo comienza generando una población de soluciones aleatorias, los anticuerpos. Por cada iteración, se evalúa la población, y se seleccionan los mejores anticuerpos según su afinidad. Para cada célula del subconjunto de mejores anticuerpos, se crean  $N_c$  clones de la propia célula. El número clones  $N_c$  de cada anticuerpo será proporcional a su afinidad.

Una vez generados los clones, estos se mutarán de forma proporcionalmente inversa. Esto significa que las copias de los mejores anticuerpos serán aquellos que menores modificaciones sufran. Los mejores clones sustituirán a los peores anticuerpos del sistema inmune original.



La implementación obtenida por el modelo de lenguaje es la representada en el pseudocódigo 17.

---

**Algoritmo 17** Sistema inmune artificial

---

```
1: anticuerpos  $\leftarrow$  lista de permutaciones
2: clones  $\leftarrow$  lista_clones(anticuerpos)
3: mejor_coste  $\leftarrow \infty$ 

4: mientras tiempo < tiempo límite hacer
5:   para todos clones hacer
6:     cloni  $\leftarrow$  mutar_solucion(clon)
7:     mejor_clon  $\leftarrow$  mejor_solucion(clones)
8:     coste  $\leftarrow$  mejor_solucion(clones)
9:     si coste < mejor_coste entonces
10:      mejor_coste  $\leftarrow$  coste
11:      mejor_solucion  $\leftarrow$  mejor_clon
12:    clones  $\leftarrow$  anticuerpos

13: devolver mejor_solucion
```

---

Esta implementación comienza generando una población de soluciones aleatorias denominadas anticuerpos. Además, se inicializan una tasa de mutación y un número de clones como hiperparámetros, por lo que parece que el modelo de lenguaje ha interpretado los elementos básicos del algoritmo correctamente.

A lo largo de las iteraciones, se crea una lista de clones, de un tamaño aleatorio, que se rellena con los  $N_c$  primero anticuerpos de la población. Ahora, esta población de clones se muta en función de una probabilidad a través de un operador de vecindario, en este caso, un  $2\text{-opt}$  explicado anteriormente. La lista de clones pasa a ser la nueva lista de anticuerpos a usar en la siguiente iteración. Este proceso se repite hasta llegar a un límite de tiempo, donde se devuelve la mejor solución obtenida.

Aunque al principio del algoritmo parece inicializar hiperparámetros correctos, el desarrollo del algoritmo es erróneo. A lo largo del procedimiento no existe ningún elemento parecido a la afinidad. Otro detalle a considerar es la mutación. En lugar de una mutación desarrollada, basada en el coste de la célula original y su afinidad, se limita a un simple operador de vecindario.

Al no existir el concepto de afinidad, la actualización de la población con el paso de las iteraciones es un simple reemplazo, no hay ninguna operación heurística.

## Resultados

Con todas estas diferencias, la implementación generada es errónea, no interpreta las operaciones características de la metaheurística solicitada, más bien se limita a una búsqueda local con elementos poblacionales.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.17.

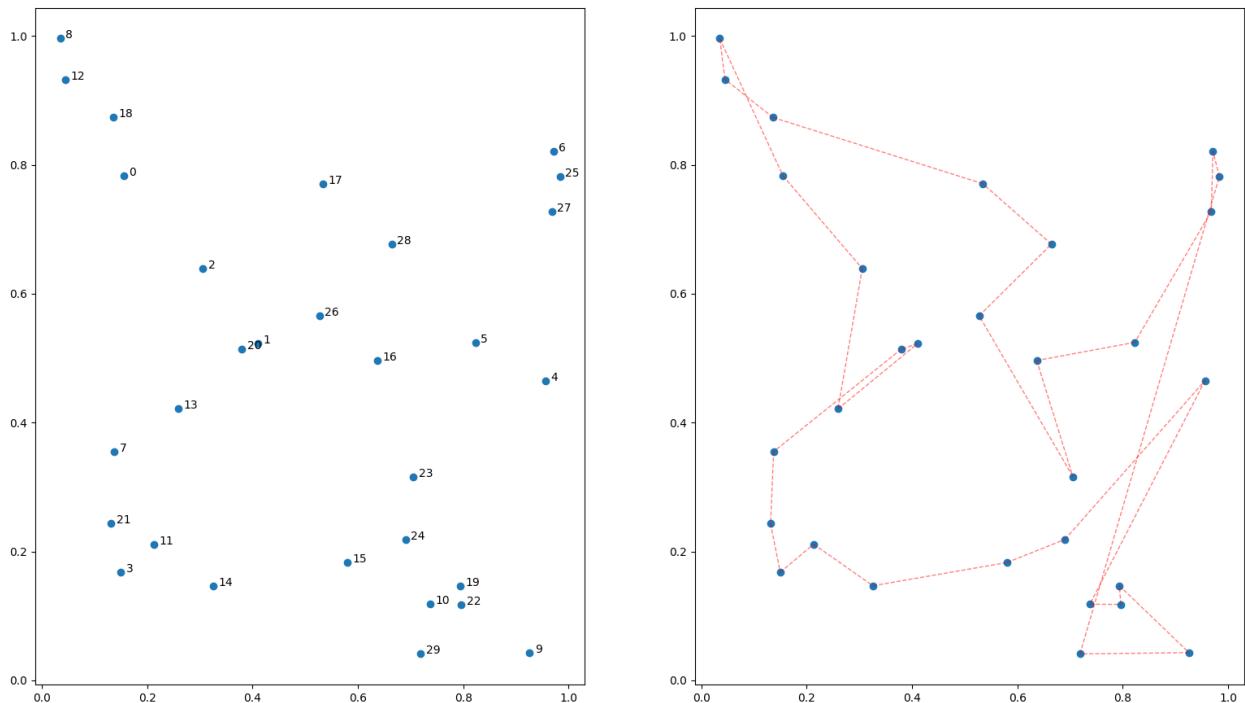


Figura 6.17: Ejemplo rendimiento Algoritmo sistema inmune artificial



### 6.1.18. Dinámica de formación de ríos

En este caso estamos ante otro caso de algoritmo inspirado en procesos naturales, en este caso, la formación de ríos, y como estos erosionan y modifican el terreno que atraviesan [67]. Partiendo de fuentes de aguas, estas se van desarrollando por un camino hasta llegar a su final, el mar.

Cada fuente de agua va a ir desarrollando su camino, y las fuentes cercanas se acabarán desviando a otros caminos en caso de ser mejores. Además, todo el terreno tiene una altura, donde las fuentes de agua tenderán a desarrollar sus caminos por las zonas de menor altura.

Las fuentes de agua se inicializan, representando soluciones únicas. Cada fuente de agua tiene una altura, de la que avanza hasta llegar a su posición final. El objetivo ideal del algoritmo sería que todas las fuentes lleguen al objetivo final a través del mismo camino y partiendo del mismo punto de inicio.

Durante el proceso, cada fuente de agua va desarrollando su camino, moviendo su flujo de agua de forma parcialmente aleatoria, tendiendo a las zonas de menor altura. Por cada movimiento que hace, se erosionan las zonas ya visitadas en función a la calidad de la solución que representa. Así, los caminos más erosionados, es decir, los de menor altura, serán las mejores soluciones. Además, se pasa por un proceso de depósito de sedimentos. Mientras se erosionan los mejores caminos, se depositan sedimentos en las demás posiciones, aumentando su altura.

Este proceso de erosión se basa en los conceptos de altura que representa cada solución y el gradiente de cada solución, siendo ésta la relación entre las alturas y distancias de la fuente de agua y el fin.

Por cada iteración, los caminos se evalúan y cuando vuelvan a moverse de forma parcialmente aleatoria este movimiento tenderá a las zonas más erosionadas, es decir, de menor altura.

La implementación dada por el modelo de lenguaje es la representada en el pseudocódigo 18.

**Algoritmo 18** Dinámica de formación de ríos

---

```
1: solucion  $\leftarrow$  generar_permutacion()
2: mejor_coste  $\leftarrow \infty$ 

3: mientras tiempo < tiempo límite hacer
4:   solucion  $\leftarrow$  generar_vecino(solucion)
5:   coste  $\leftarrow$  evaluar_solucion(solucion)
6:   si coste < mejor_coste entonces
7:     mejor_coste  $\leftarrow$  coste
8:     mejor_solucion  $\leftarrow$  solucion

9: devolver mejor_solucion
```

---

En este caso, la implementación generada por el modelo de lenguaje es completamente errónea, pues estamos ante una metaheurística genérica. Lo único que hace este algoritmo es generar soluciones aleatorias, generar un vecindario y evaluarlo. En ningún momento no se llega a intentar recrear los elementos propios del algoritmo.

Pareciera que el modelo de lenguaje desconoce o es incapaz de relacionar el nombre del algoritmo a su concepto aunque al pedirle una explicación meramente teórica del algoritmo, esta sí se acerca al desarrollo teórico propio del algoritmo. Por mucho que el modelo de lenguaje parezca relacionar el algoritmo a su funcionamiento teórico, es incapaz de generar una implementación válida.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.18.

## Resultados

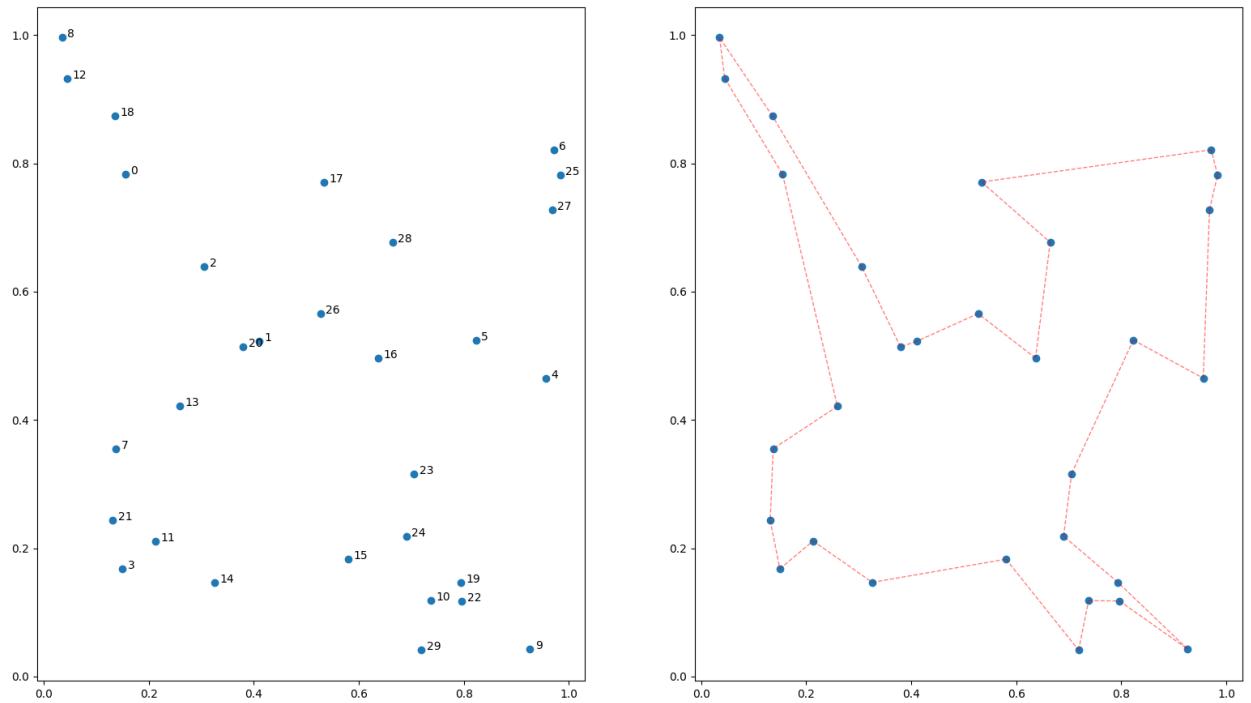


Figura 6.18: Ejemplo rendimiento dinámica formación ríos

### 6.1.19. Algoritmo de luciérnagas

Estamos ante otra metaheurística inspirada en procesos naturales, en este caso, en como las luciérnagas se ven atraídas a fuentes de luz.

La implementación básica, según lo desarrollado en el artículo [68], debería de tener los siguientes componentes:

Se empieza generando una población de luciérnagas, donde cada luciérnaga representa una solución. Cada luciérnaga tiene un valor de intensidad lumínica, proporcional a la calidad de la solución representada por la luciérnaga.

En cada iteración, se evalúa cada luciérnaga. Durante esta evaluación, la primera luciérnaga  $L_i$  se compara con todas las demás luciérnagas  $L_j$  de la población. Se calculan sus intensidades lumínicas y si la intensidad de  $L_j$  es mayor que la intensidad de  $L_i$ , la primera luciérnaga se moverá hacia la segunda luciérnaga, moviéndose en función de la distancia, un componente de atracción basado en un hiperparámetro, el coeficiente de absorción de luz y un elemento de aleatoriedad.



Una vez realizado el movimiento, se evalúan las nuevas soluciones y se actualizan los valores de intensidad lumínica.

Este proceso se realiza para todas las luciérnagas, comparándolas a todas las demás de la población, salvo un caso especial, la mejor luciérnaga de la población. Esta, al no verse atraída por ninguna otra luciérnaga, se moverá a posiciones completamente aleatorias, añadiendo así un componente de diversificación al algoritmo.

La implementación dada por el modelo de lenguaje es la representada en el pseudocódigo 19.

---

**Algoritmo 19** Algoritmo de luciérnagas

---

```
1: poblacion ← generar_poblacion()
2: costes ← evaluar_poblacion(poblacion)

3: mientras tiempo < tiempo limite hacer
4:   para todos luciérnagas en la población hacer
5:     solucion ← luciernagai
6:     coste ← evaluar_solucion(solucion)
7:     si costes de luciernagai < coste entonces
8:       distancia ← distancia_euclidea(solucion, luciernagai)
9:       atraccion ← calcular_atraccion(distancia)
10:      direccion ← calcular_direccion(distancia)
11:      solucion ← actualizar_posicion(solucion, distancia, atraccion, direccion)
12:      coste ← evaluar_solucion(solucion)
13:      si coste < mejor_coste entonces
14:        mejor_coste ← coste
15:        mejor_solucion ← solucion
16:      poblacion ← actualizar_poblacion(solucion)

17: devolver mejor_solucion
```

---

Esta implementación comienza generando una población de soluciones aleatorias, y para cada solución, se compara su coste, con su propio coste. En caso en el que el coste de la solución es mejor que el coste de la solución con la que se compara, la solución original se modifica en función a la distancia entre las luciérnagas, un componente de atracción y un componente de aleatoriedad, así que esa parte de la implementación es correcta.

## Resultados

El problema está en que en ningún momento se llega a realizar tal operación, pues al compararse consigo misma, nunca se cumple la condición de esta. Además, tampoco se tiene en cuenta un trato especial para la mejor luciérnaga de la población.

Aunque el cálculo de las nuevas posiciones, con su respectivo cálculo de la atracción entre luciérnagas y posición parezca correcta, nunca se llega a usar, por lo que es inútil.

Aunque un ligero cambio en la comparación de soluciones arreglaría la implementación, o por lo menos la mejoraría, el código dado por el modelo de lenguaje es incorrecto. No solamente es incorrecto, como algoritmo es inútil, pues salvo la primera iteración, no se obtienen nuevos resultados. Lo único que hace este algoritmo es generar una población de soluciones aleatorias que no varía de ningún modo. Como algoritmo es completamente inútil.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.19.

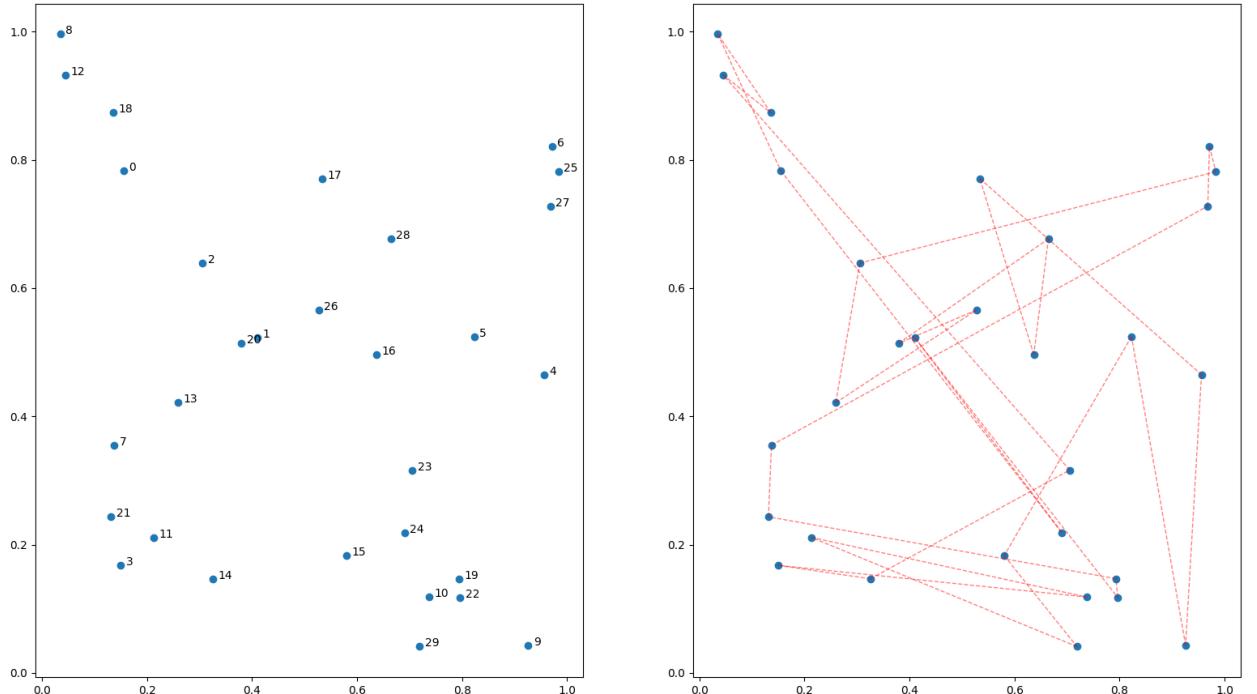


Figura 6.19: Ejemplo rendimiento Algoritmo luciérnaga



### 6.1.20. Algoritmo genético

Los algoritmos genéticos son la familia de algoritmos más estudiados y comentados dentro de la familia de las metaheurísticas.

A su vez, son un subgrupo dentro de los algoritmos evolutivos, por lo que tienen un funcionamiento y un flujo de ejecución muy característico.

Estos algoritmos se inspiran en la genética, en como la información genética se almacena en cromosomas, compuestos de genes individuales. Estos cromosomas pasan por un proceso natural de mutación, donde por probabilidad un gen puede cambiar su valor aleatoriamente [69].

Además, en una implementación básica de estos algoritmos, se pasará por un proceso de cruce, donde dos individuos de la población se cruzarán, dando lugar a uno o varios individuos que heredan ciertas características de los originales.

Dentro de la familia de algoritmos genéticos, existe una gran variedad de implementaciones por toda la posible variedad a la hora de desarrollar el proceso de mutación, de cruce, el proceso de selección de soluciones “padre” o incluso el uso de elitismo para asegurar la intensificación.

Por toda la variedad de posibilidades, estos algoritmos son de los más estudiados y diversos [9].

Una implementación básica de algoritmo genético debería comenzar generando una solución de soluciones aleatorias y evaluándola.

Para cada iteración, se seleccionarán un par de soluciones de la población según su calidad, es decir, seleccionar las dos mejores soluciones. Estas dos soluciones “padres” se cruzan según una probabilidad a través de un operador de cruce, dando lugar a una o varias nuevas soluciones “hijas”, la población descendiente.

Cada elemento de esta nueva población descendiente sufrirá una mutación aleatoria según una probabilidad en uno de sus genes.

Una vez creada y mutada la nueva población, esta sustituirá a la original, ya sea reemplazando a los  $N$  peores individuos de la población o a toda la población si se ha generado una completa.



El algoritmo acabará devolviendo la mejor solución encontrada.

La implementación dada por el modelo de lenguaje es la representada en el pseudocódigo 20.

---

**Algoritmo 20** Algoritmo genético

---

```
1: poblacion ← generar_poblacion()
2: mejor_coste ← ∞

3: mientras tiempo < tiempo limite hacer
4:   solucion ← seleccionar_mejor(poblacion)
5:   coste ← evaluar_solucion(solucion)
6:   si coste < mejor_coste entonces
7:     mejor_coste ← coste
8:     mejor_solucion ← solucion
9:   nueva_poblacion ← lista vacia
10:  para todos soluciones en la población hacer
11:    padre1 ← seleccion_aleatoria(poblacion)
12:    padre2 ← seleccion_aleatoria(poblacion)
13:    mejor_padre ← seleccion_torneo(padre1, padre2)
14:    hijo ← mutacion(mejor_padre)
15:    nueva_poblacion ← incluir(hijo)

16: devolver mejor_solucion
```

---

La implementación comienza correctamente generando una población de soluciones aleatorias.

A lo largo de las iteraciones, se evalúa la población y se seleccionan dos padres. Estos padres se seleccionan aleatoriamente de la población, y de estos dos padres, se selecciona el mejor como parental.

A partir de este parental, se genera una nueva solución hija, que no es más que una copia de este parental seleccionado. Este hijo se muta, variando aleatoriamente entre los operadores *2-opt* y *2-swap*.

Este proceso se repite hasta generar suficientes hijos para generar una población completa. Una vez se tenga una nueva población, esta reemplazará a la original, y se repetirá el proceso. El proceso acaba devolviendo la mejor solución obtenida.

## Resultados

Estamos ante otro caso en el que el modelo de lenguaje reconoce el algoritmo y sus elementos característicos, pero falla al implementarlos.

En primer lugar, la selección de soluciones padres, aunque no sea completamente incorrecta, es muy impráctica, pues al seleccionar padres aleatorios en lugar de soluciones buenas, se pierde el principal componente de intensificación del algoritmo.

Por otro lado, no existe una operación de cruce entre los padres, ya que el hijo es una copia exacta del mejor de los padres.

Con la impráctica selección de padres y la completa falta de operación de cruce, podemos afirmar que la implementación es incorrecta.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.20.

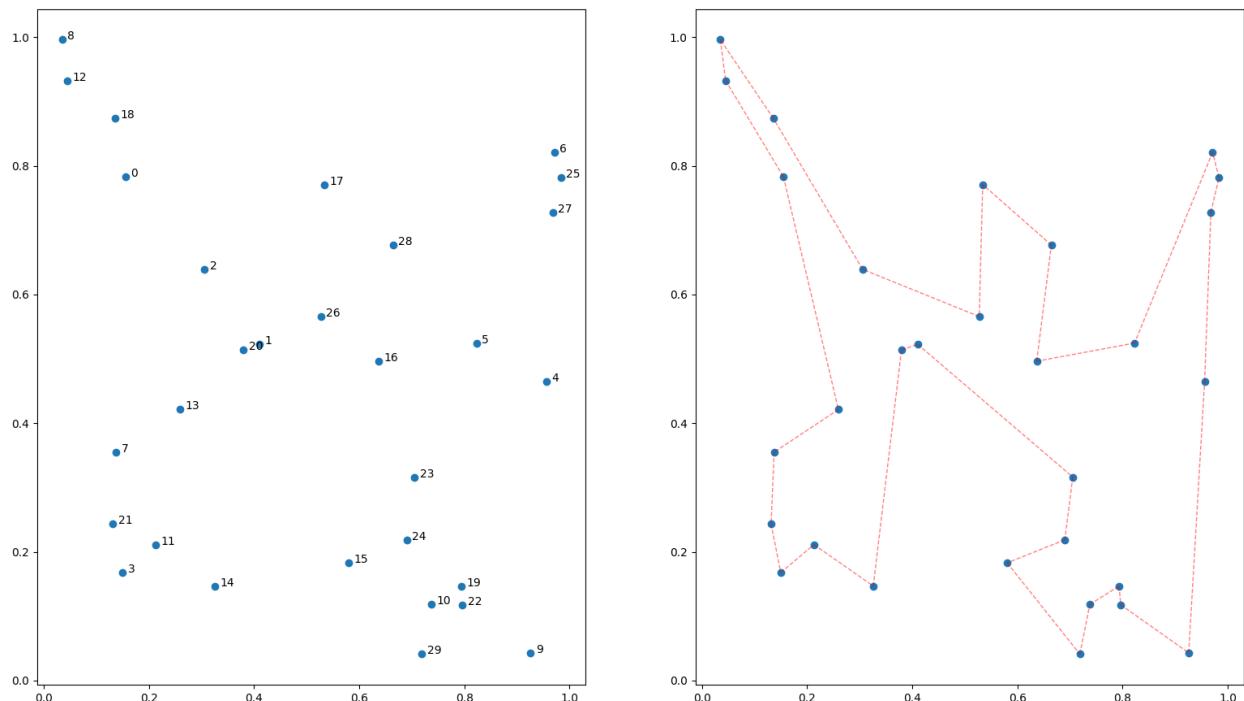


Figura 6.20: Ejemplo rendimiento Algoritmo genético



### 6.1.21. Búsqueda local guiada

Después de una serie de algoritmos inspirados en procesos naturales, volvemos a un algoritmo basado en trayectorias. En este caso, la búsqueda local guiada es una extensión de la búsqueda local simple.

Durante esta búsqueda, se imponen restricciones para poder escapar de las zonas del espacio de búsqueda de las óptimas locales [70]. Esto se consigue modificando la función objetivo al llegar a un óptimo local, escapando de la zona del óptimo. A esta nueva función se la llama función de coste aumentada.

El algoritmo se basa en la información de las búsquedas previas para guiar las futuras búsquedas. Esta información se representa en restricciones que modifican la función objetivo.

Se comienza realizando una búsqueda local sin ninguna restricción, buscando un óptimo local de la función de coste. Una vez encontrado un óptimo, se modifica la función objetivo, añadiendo restricciones y se reinicia la búsqueda, partiendo del óptimo ya encontrado.

La implementación dada por el modelo de lenguaje está representada en el pseudocódigo 21.

---

#### Algoritmo 21 Búsqueda local guiada

---

```
1: solucion  $\leftarrow$  generar_permutacion()
2: mejor_coste  $\leftarrow \infty$ 

3: mientras tiempo  $<$  tiempo limite hacer
4:   mejor_vecino  $\leftarrow$  seleccionar_mejor_vecino(solucion)
5:   coste  $\leftarrow$  evaluar_solucion(mejor_vecino)
6:   si coste  $<$  mejor_coste entonces
7:     mejor_coste  $\leftarrow$  coste
8:     mejor_solucion  $\leftarrow$  solucion

9: devolver mejor_solucion
```

---

Esta implementación parte de una solución aleatoria, sobre la cual se realiza una búsqueda local. Se analiza las soluciones vecinas a la solución original, generando estas por el operador de 2-opt. Se analiza el coste de todos los vecinos y se selecciona aquella que más mejore la solución original, es decir, el mejor vecino. Sobre esta

## Resultados

nueva solución se repite el proceso de búsqueda local.

El proceso se repite hasta no encontrar un vecino mejor, es decir, hasta caer en un óptimo local ya conocido.

Con todo lo comentado, claramente estamos ante una búsqueda local por mejor vecino. No se tiene en cuenta nada de la información obtenida en las distintas búsquedas. No se plantea ningún concepto parecido a las restricciones, ni modifica la función objetivo para guiar la búsqueda y escapar de los óptimos locales.

En definitiva, la implementación es completamente errónea.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.21.

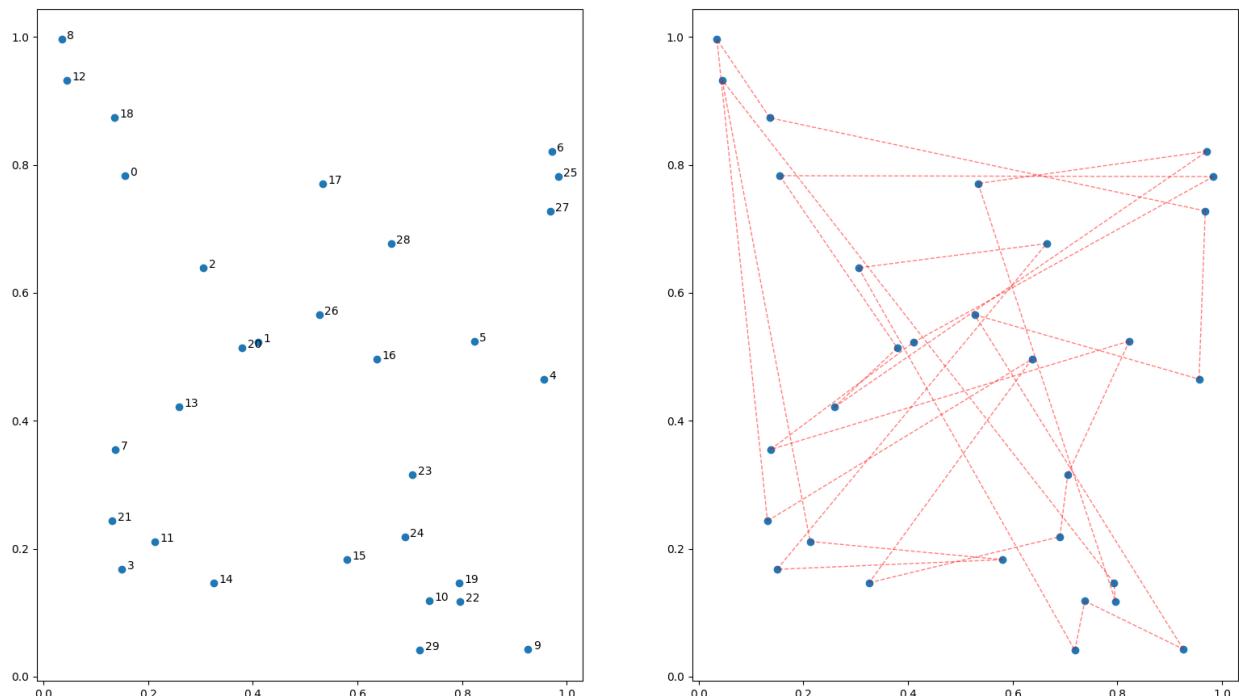


Figura 6.21: Ejemplo rendimiento Búsqueda local guiada



### 6.1.22. Algoritmo búsqueda de cuervo

Volviendo a los algoritmos basados en procesos naturales, la búsqueda de cuervo es un algoritmo de inteligencia de enjambre que simula el comportamiento de los cuervos, que almacenan el exceso de comida y la recuperan cuando la necesitan [71].

El algoritmo se inicializa generando una población de cuervos, representando cada uno una solución inicialmente aleatoria y teniendo cada uno una memoria propia.

A lo largo de las iteraciones, la solución representada por cada cuervo se evalúa, y su costo o valor se almacena en su memoria como valor inicial.

Para cada cuervo  $C_i$ , este seleccionará aleatoriamente a otro cuervo  $C_j$ , de la población y genera un valor aleatorio. Si este valor aleatorio supera un umbral, la posición del cuervo  $C_i$  se modifica, moviéndose en dirección al valor almacenado en la memoria de  $C_j$ .

El proceso se repite para todos los cuervos de la población, acabando cada iteración actualizando el valor de memoria de cada cuervo.

La implementación dada por el modelo de lenguaje es la representada en el pseudocódigo 22.

---

#### Algoritmo 22 Algoritmo búsqueda de cuervo

---

```
1: poblacion  $\leftarrow$  generar_poblacion()
2: mejor_coste  $\leftarrow \infty$ 

3: mientras tiempo  $<$  tiempo límite hacer
4:   para todos cuervos en la población hacer
5:     coste  $\leftarrow$  evaluar_solucion(cuervoi)
6:     cuervo_vecino  $\leftarrow$  busqueda_local(cuervoi)
7:     nuevo_coste  $\leftarrow$  evaluar_solucion(cuervo_vecino)
8:     si nuevo_coste  $<$  coste entonces
9:       poblacion  $\leftarrow$  actualizar_poblacion(cuervo_vecino)
10:      mejor_solucion  $\leftarrow$  nuevo_coste

11: devolver mejor_solucion
```

---

## Resultados

El proceso inicia generando una población de soluciones aleatorias. A lo largo de las iteraciones, para cada cuervo se evalúa su solución y se selecciona un cuervo vecino a través de un proceso de búsqueda local. Los vecinos se generan por el operador de vecindario *2-opt*, devolviendo el mejor de los posibles cuervos. Al final, es un proceso de búsqueda local por mejor vecino.

Si este nuevo cuervo vecino es mejor, se reemplaza al cuervo original por su mejor vecino.

Como podemos ver, este algoritmo es un proceso de búsqueda local por mejor vecino, con la característica de ser poblacional y no iterar sobre una única solución. No se aplica ninguno de los elementos propios de la búsqueda de cuervo, ni memorias, ni movimiento en dirección a mejores cuervos. La implementación es completamente errónea.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.22.

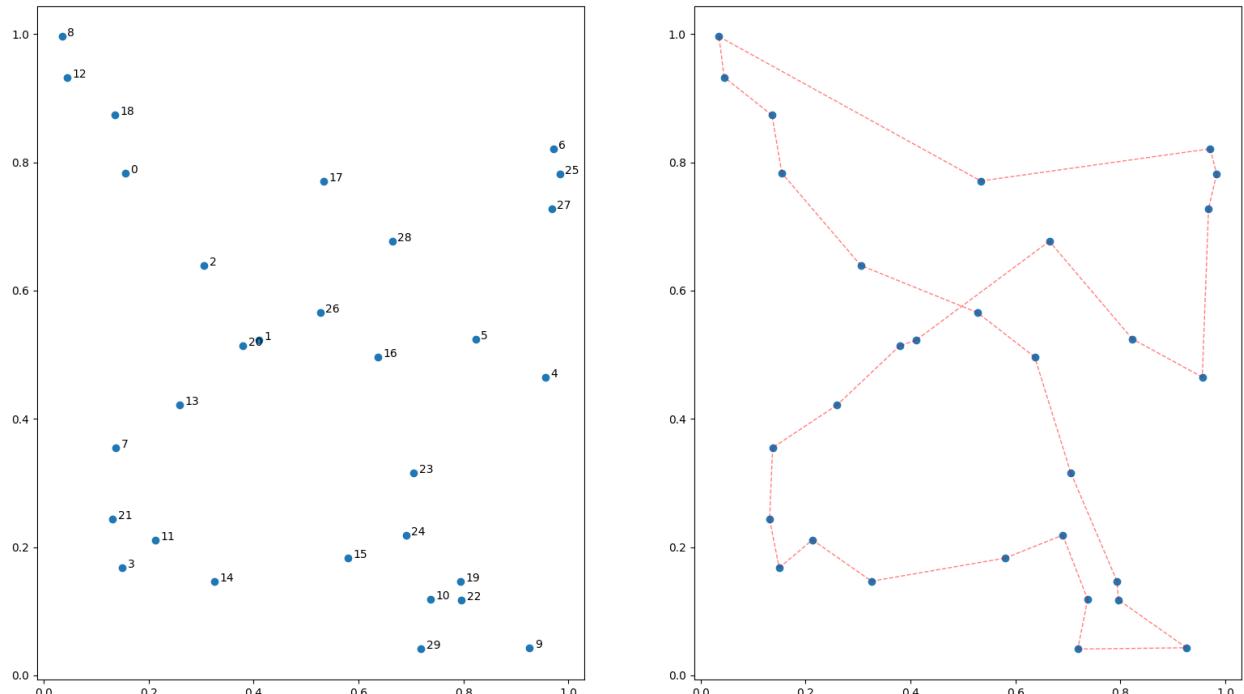


Figura 6.22: Ejemplo rendimiento búsqueda cuervo



### 6.1.23. Algoritmo de maleza invasora

El algoritmo de maleza invasora, nuevamente, está inspirado en procesos naturales. En este caso, en cómo las malezas o malas hierbas se reproducen y se adaptan, expandiéndose a todo tipo de terrenos y zonas [72].

El algoritmo comienza generando una población de malezas, representando cada una una solución inicialmente aleatoria. La población se evalúa, y para cada maleza individual se genera una cantidad de semillas.

Esta cantidad de semillas será proporcional a la calidad de las soluciones representadas por la maleza. Estas semillas se distribuyen por el espacio de búsqueda mediante una distribución normal de media cero pero varianza variable, reduciéndose esta a medida que se acerca al óptimo global. Así se asegura que las semillas se distribuyan aleatoriamente por el espacio, pero manteniéndose cerca del padre.

Se evalúa la nueva población, agrupando las malezas originales y las nuevas malezas generadas por las semillas. Si se supera un límite de tamaño en la población, se pasa por un proceso de eliminación competitiva, eliminándose aquellas malezas que generen pocas o ninguna semilla, es decir, permaneciendo las mejores soluciones de la población.

La implementación dada por el modelo de lenguaje es la representada en el pseudocódigo 23.

---

#### Algoritmo 23 Algoritmo de maleza invasora

---

```
1: poblacion ← generar_poblacion()
2: mejor_coste ← ∞

3: mientras tiempo < tiempo limite hacer
4:   para todos cuervos en la población hacer
5:     costes ←
6:     adecuacion ← calcular_adecuacion(costes)
7:     semillas ← distribuir_semillas(poblacion, adecuacion)
8:     poblacion ← nueva_poblacion_según_semillas(semillas)

9:   mejor_solucion ← mejor_solucion(poblacion)
10:  mejor_coste ← evaluar_solucion(mejor_solucion)
11:  devolver mejor_solucion
```

---



Esta implementación comienza generando una población de soluciones iniciales aleatorias. Para cada iteración, se evalúa la población y basado en los costes de la población, se calcula un valor de adecuación, siendo  $adec_i = \frac{1}{coste_i}$ .

A partir de este valor de adecuación, se establece el número de semillas a generar y se generan las semillas, que no son más que copias de la solución que representa la maleza padre. Cada semilla se copia en la población un número estipulado de veces, en función de su calidad. Todas las semillas se añaden a la población, y al final de cada iteración, la población se limita a los  $N$  mejores individuos.

Aunque simplificado, la idea se aproxima al algoritmo ideal. Se genera una población de semillas hijas a partir de la calidad de las soluciones, y reemplazan a las peores soluciones de la antigua población. El gran fallo llega a la hora de distribuir las semillas, pues al ser estas copias de la maleza original, no hay ningún tipo de diversificación. No se exploran nuevas áreas del espacio de búsqueda, si no que se tiende a explorar el primer óptimo local descubierto.

La implementación es aproximada, pero no llega a ser correcta.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.23.

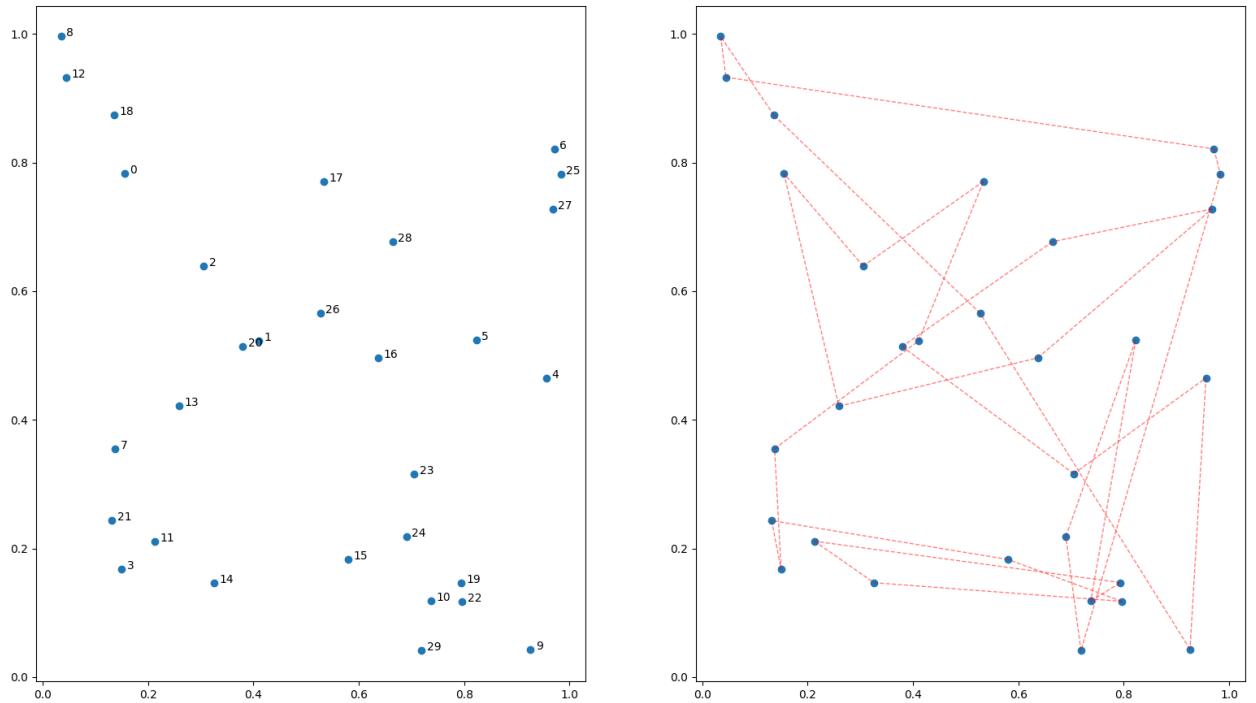


Figura 6.23: Ejemplo rendimiento Algoritmo maleza invasora

#### 6.1.24. Algoritmo del murciélagos

El algoritmo del murciélagos es otro algoritmo inspirado en procesos naturales. En este caso, el proceso de optimización gira en torno a la capacidad de ecolocalizar presas, diferenciar distintos tipos de insectos e incluso el entorno a su alrededor, hasta en completa oscuridad. Esta ecolocalización puede ser interpretada como una función objetiva a ser optimizada por un algoritmo [73].

En este algoritmo, cada murciélagos usa la ecolocalización para percibir distancias y poder diferenciar entre comida y presas de obstáculos del terreno. Además, los murciélagos vuelan aleatoriamente con una velocidad y tasa de emisión, ajustando la intensidad de sus pulsos para buscar sus presas. También pueden ajustar la tasa de emisión de los pulsos, dependiendo de la proximidad a la presa.

Una implementación básica del algoritmo deberá comenzar generando una población de murciélagos, cada uno con un valor de velocidad, frecuencia y representando cada murciélagos una solución.



A lo largo de las iteraciones, por cada murciélagos se debe generar una nueva solución. Si de forma probabilista se supera la tasa de pulso, se genera una nueva solución cercana a la original, es decir, una solución vecina. Además, se genera una segunda solución, completamente aleatoria.

Una vez generadas las soluciones, estas se aceptan si mejoran la calidad e intensidad de la solución original. Tras esto, se aumenta la tasa de pulso y se reduce la intensidad asociada a la solución.

La implementación dada por el modelo de lenguaje es la representada en el pseudocódigo 24.

---

**Algoritmo 24** Algoritmo de murciélagos

---

```
1: poblacion ← generar_poblacion()
2: mejor_coste ← ∞
3: frecuencia ← hiperparametro

4: mientras tiempo < tiempo limite hacer
5:   para todos murciélagos en la población hacer
6:     solucion ← generar_permutacion()
7:     solucion ← perturbacion(solucion, frecuencia)
8:     coste ← evaluar_solucion(solucion)
9:     si coste < mejor_coste entonces
10:      mejor_coste ← coste
11:      mejor_solucion ← solucion
12:      frecuencia ← actualizar_frecuencia(frecuencia)
13: mejor_solucion ← mejor_solucion(poblacion)
14: devolver mejor_solucion
```

---

La implementación comienza generando una población de murciélagos, e inicializando hiperparámetros de intensidad, tasa de pulso mínima y máxima. En este caso, se usan valores globales de estos hiperparámetros, y no un valor único para cada individuo de la población.



Para cada individuo de la población, se genera una solución aleatoria y se genera una perturbación de la misma. Esta perturbación se genera probabilísticamente al superar un umbral de tasa mínima de pulsos. Esta permutación, a su vez, se genera de forma aleatoria para cada posible elemento de la solución representada. Esta nueva solución se acepta si mejora la solución original, o bien se acepta según una probabilidad.

El modelo de lenguaje parece entender elementos teóricos, ya que genera correctamente la población, hiperparámetros, y parece entender la función de las tasas de pulsos o intensidad. La generación de soluciones vecinas es correcta, pero por lo demás, el algoritmo falla en implementar el procedimiento y dar uso a los hiperparámetros generados.

Sólo se genera una solución vecina, y no se añade diversificación generando soluciones aleatorias. Además, al usar hiperparámetros globales, y no valores para cada individuo de la población, hace que el progreso del algoritmo, tanto en diversificación como en intensificación sean inútiles.

Tras esto, podemos afirmar que la implementación es incorrecta y dista en muchos aspectos de la implementación teórica esperable.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.24.

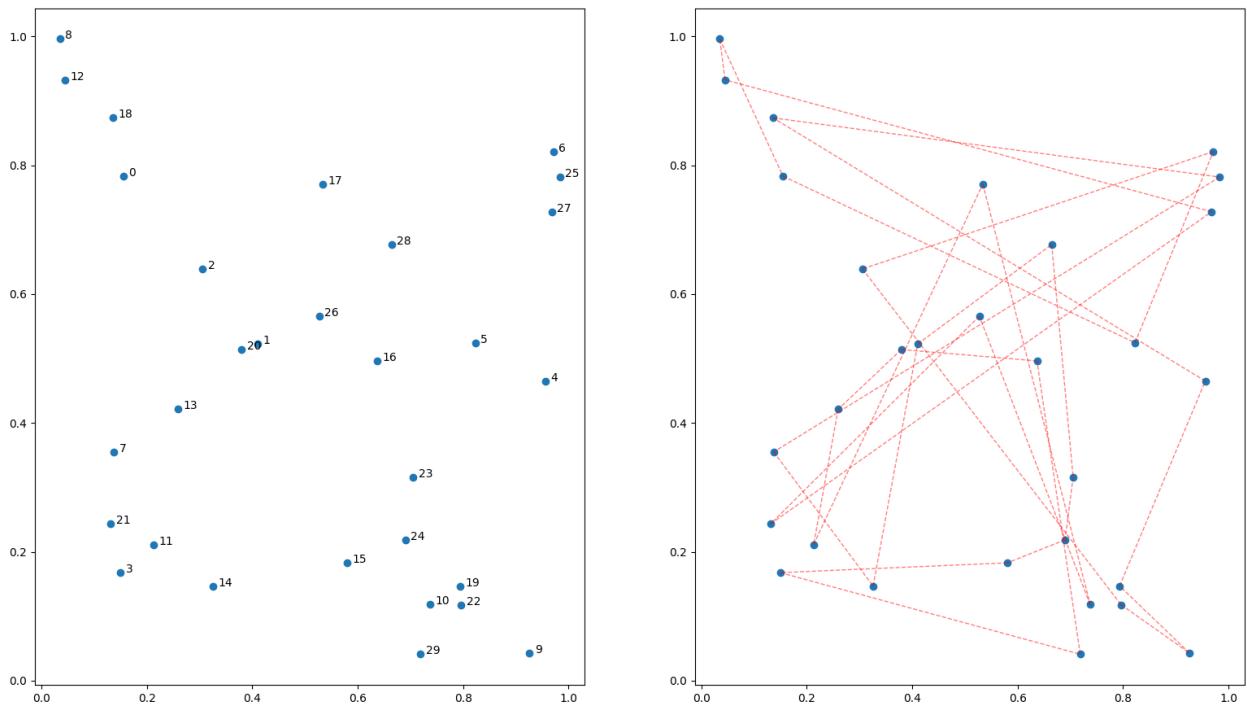


Figura 6.24: Ejemplo rendimiento Búsqueda murciélagos

### 6.1.25. Evolución diferencial

La evolución diferencial es un bien conocido algoritmo evolutivo, parecido a los algoritmos genéticos en funcionamiento. Es un algoritmo basado en poblaciones, con una población inicial elegida aleatoriamente, cuyos parámetros se encuentran dentro de unos límites preestablecidos.

El funcionamiento es similar al de los algoritmos evolutivos comunes, siendo la gran diferencia el uso de vectores de parámetros en el proceso de mutación para explorar el espacio de búsqueda [74].

El algoritmo muta y combina la población para generar nuevos individuos de la población. Partiendo de tres individuos distintos, se crea uno nuevo, combinándolos y añadiendo un elemento de aleatoriedad, siguiendo la fórmula:

$$V_{nuevo} = V_0 + A \cdot (V_1 - V_2)$$

Donde A es un número aleatorio entre 0 y 1.



Tras este proceso de generación de nuevas soluciones la población de individuos se recomienda a través de operadores de cruce comunes. Específicamente, el proceso consiste en combinar individuos originales con alguno de los nuevos individuos generados por las mutaciones si se cumple una condición probabilística.

Así, se consigue un doble componente de diversificación, tanto por la combinación como por la mutación de los individuos.

Por último, se pasa por un proceso de selección. Si el nuevo individuo  $N_i$  mejora al individuo  $V_i$ , lo reemplazará en la siguiente generación.

La implementación dada por el modelo de lenguaje es la representada en el pseudocódigo 25.

---

**Algoritmo 25** Algoritmo Evolución diferencial

---

```
1: poblacion ← generar_poblacion()
2: mejor_coste ← ∞

3: mientras tiempo < tiempo limite hacer
4:   para todos soluciones en población hacer
5:     solucion1 ← muestra_aleatoria(poblacion)
6:     solucion2 ← muestra_aleatoria(poblacion)
7:     solucion3 ← muestra_aleatoria(poblacion)
8:     si Condición de mutación entonces
9:       solucion ← mutacion(solucion1, solucion2, solucion3)
10:    si Condición de cruce entonces
11:      solucion ← cruce(solucion, solucion1, solucion2, solucion3)
12:      coste ← evaluar_solucion(solucion)
13:      si coste < mejor_solucion(poblacion) entonces
14:        poblacion ← actualizar_poblacion(solucion)

15: mejor_solucion ← mejor_solucion(poblacion)
16: devolver mejor_solucion
```

---



La implementación comienza generando una población de soluciones aleatorias de  $N$  elementos. Para cada iteración, se generan  $N$  nuevos individuos a partir de mutaciones. Cada uno de estos nuevos individuos se genera seleccionando 3 individuos aleatorios de la población original y se mutan, siguiendo la fórmula teóricamente esperable:

$$V_{nuevo} = V_0 + M \cdot (V_1 - V_2)$$

Donde  $M$  es la tasa de mutación, un hiperparámetro. Una vez generado el nuevo individuo, se pasa por un proceso de cruce. Esta operación está mal implementada, pues en lugar de cruzar un individuo original con uno mutado, solamente se genera una copia del nuevo individuo mutado.

Por último, este nuevo individuo  $N_i$  formará parte de la nueva generación si mejora la solución previa  $V_i$ .

El modelo de lenguaje parece entender y reconocer los elementos característicos del algoritmo, como el proceso de mutación a partir de 3 individuos, la fórmula de mutación o la construcción de la nueva generación. En cambio, falla en elementos clave como lo es el cruce.

Aunque el algoritmo es funcional y puede llegar a generar soluciones válidas y hasta sub-óptimas, es una implementación muy falta tanto en diversificación como en intensificación.

Para estimar el rendimiento esperable del algoritmo tenemos un ejemplo de resultado, evaluado tras un minuto de ejecución sobre un problema aleatorio de 30 ciudades, reflejado en la figura 6.25.

## Resultados

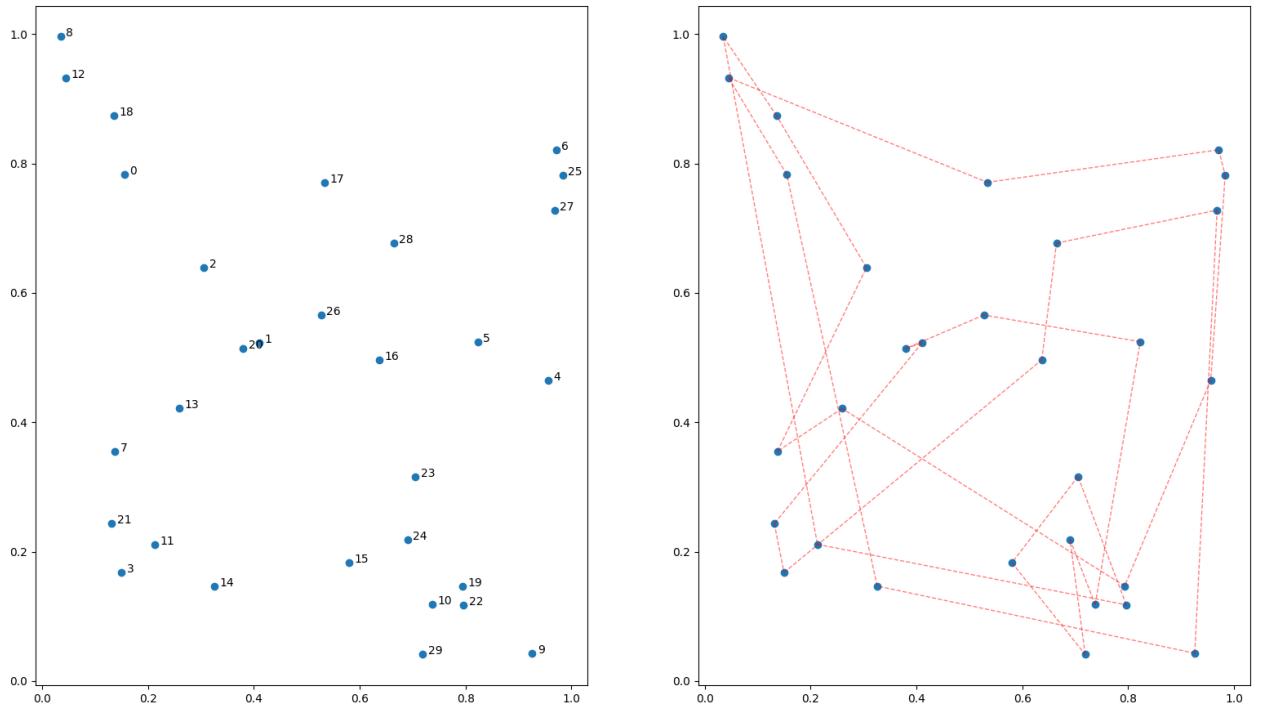


Figura 6.25: Ejemplo rendimiento Evolución diferencial

## 6.2. Calidad de las implementaciones

Después de todo el análisis teórico de las implementaciones y las comparaciones del funcionamiento teórico del algoritmo con las implementaciones obtenidas por el modelo de lenguaje, podemos resumir esta información, representada en la tabla 6.1.

Esta tabla muestra un resumen de las implementaciones generadas por el modelo de lenguaje. Para cada algoritmo destacamos si se reconoce el algoritmo, es decir, si el modelo parece tener un conocimiento teórico del algoritmo, ya sea a través de la explicación del funcionamiento del mismo o por la propia implementación.

Destacamos también si el funcionamiento es correcto con el esperable teóricamente del propio algoritmo. Es decir, un algoritmo reconocido puede estar mal implementado, o su funcionamiento puede ser el correcto aún teniendo errores en la implementación.

A modo de conclusión, se añade un breve comentario característico a dicha implementación.



Tabla 6.1: Tabla de implementaciones

Algoritmo	Reconoce algoritmo	Funcionamiento	Conclusión
Búsqueda aleatoria	Sí	Correcto	Implementación correcta
Búsqueda local primer vecino	Sí	Correcto	Implementación errónea
Búsqueda local mejor vecino	Sí	Correcto	Implementación correcta
Búsqueda tabú	Sí	Correcto	Implementación correcta
Enfriamiento simulado	Sí	Erroneo	Implementación errónea
Colonia de hormigas	Sí	Correcto	Implementación correcta
Colonia de abejas	No	Erroneo	Implementación errónea
Enjambre de partículas	Sí	Correcto	Implementación correcta
Búsqueda gravitacional	Sí	Erronea	Implementación errónea
Algoritmo agujeros negros	Sí	Correcto	Implementación correcta
Búsqueda dispersa	No	Erroneo	No se reconoce el algoritmo
Algoritmo cultural	Sí	Correcto	Implementación correcta
Búsqueda vecindario variable	Sí	Correcto	Implementación correcta
Búsqueda armónica	No	Erroneo	No se reconoce el algoritmo
Algoritmo imperialista competitivo	Si	Correcto	Implementación errónea
Algoritmo optimización del caos	No	Erroneo	No se reconoce el algoritmo
Sistema Inmune artificial	No	Erroneo	No se reconoce el algoritmo
Dinámica formación ríos	No	Erroneo	No se reconoce el algoritmo
Algoritmo luciérnagas	Sí	Erroneo	Implementación errónea
Algoritmo genético	Sí	Correcto	Implementación errónea
Búsqueda local guiada	No	Erroneo	No se reconoce el algoritmo
Búsqueda de cuervos	No	Erroneo	No se reconoce el algoritmo
Algoritmo maleza invasora	Si	Correcto	Implementación correcta
Algoritmo murciélagos	No	Erroneo	No se reconoce el algoritmo
Evolución diferencial	Si	Correcto	Implementación errónea



A partir de este resumen, podemos diferenciar tres tipos de implementaciones:

- Algoritmos reconocidos y bien implementados.
- Algoritmos reconocidos pero mal implementados
- Algoritmos no reconocidos

En cuanto al reconocimiento teórico de los algoritmos, podemos ver que la mayoría de algoritmos son reconocidos. De los 25 algoritmos implementados, en 16 reconoce su funcionamiento esperado y las características esperables. En los 9 algoritmos restantes, la implementación no muestra ninguno de los elementos esperables de cada algoritmo. En estos casos, la implementación obtenida suele ser bien una búsqueda aleatoria o una búsqueda local simple, lo que el modelo parece reconocer como implementaciones genéricas de metaheurísticas.

Esto se puede explicar por el proceso de entrenamiento y generación de respuestas del modelo de lenguaje [75]. Un modelo de lenguaje de estas características no es capaz de generar respuestas completamente fuera de los datos de entrenamiento. Por esto mismo, los algoritmos más estudiados y trabajados, con mayor bibliografía son aquellos en los que se reconocen todos los elementos característicos sin errores.

Aún sin conocer públicamente los datos de entrenamiento del modelo de lenguaje usado, podemos suponer que los algoritmos no reconocidos pueden serlos por falta de datos de entrenamiento, al ser algoritmos menos trabajados o estudiados en comparación, además de por falta de ejemplos prácticos de sus implementaciones dentro de los datos de entrenamiento del modelo.

Aún así cabe resaltar que en todos los casos el modelo de lenguaje afirma conocer el algoritmo. Al solicitar una implementación para un algoritmo concreto, no indica desconocimiento sino que generar directamente código, dando incluso una breve explicación teórica. Cuando se le pregunta al modelo de lenguaje por una lista de metaheurísticas que sea capaz de implementar, los 25 algoritmos evaluados se incluyen en dicha lista.



Así, el modelo cae en contradicción. Al solicitar las implementaciones, no parece reconocer ningún elemento del funcionamiento esperable, pero al preguntar por separado por información teórica sí parece reconocer elementos propios de los algoritmos. Esto ocurre en casos específicos, como en la metaheurística del Sistema Inmune Artificial, mientras que en otros casos no reconoce ni las características teóricas del algoritmo.

En cuanto al funcionamiento práctico, nos encontramos con lo mismo. De los algoritmos que no se llegan a reconocer teóricamente se obtienen búsquedas locales o aleatorias, por lo cual nos son inútiles a efectos prácticos.

En las implementaciones que sí se acercan a lo esperado teóricamente, encontramos diferentes casuísticas. De los 16 algoritmos reconocidos, tenemos:

- Implementaciones correctas (9 / 16)
- Implementaciones con errores menores (3 / 16)
- Fallos de concepto (4 / 16)

De los 16 algoritmos que se reconocen, solamente 9 se han implementado sin ningún tipo de error ni falta. Estos 9 algoritmos han sido aquellos con más bibliografía previa, como lo son la búsqueda local, búsqueda tabú y colonia de hormigas entre otros.

Este se explica por los mismos motivos anteriormente nombrados. Mientras que los conceptos teóricos de un algoritmo no varían, las implementaciones pueden variar según el lenguaje de programación, autor o problema a resolver, dejándonos con una gran variedad de implementaciones distintas. Esto, sumado a la limitación del conjunto de datos con el que se entrena el modelo, puede llevar a confusiones y malas interpretaciones a la hora de generar código.

Con los algoritmos más estudiados y documentados, podemos ver que la cantidad de información y bibliografía se traduce en implementaciones precisas.



Al revisar el resto de implementaciones, vemos que el modelo de lenguaje comete fallos destacables, aunque no sean completamente inútiles. En 3 de los algoritmos, se cometén errores menores, como confundir la búsqueda local por primer vecino con la búsqueda local por mejor vecino. Al tratarse de variaciones del mismo algoritmo con cambios mínimos, un error de estas características es esperable, hasta pudiéndose considerar como errores de programación a la hora de implementar. Otro ejemplo de error menor se da en el Enfriamiento Simulado, que calcula erróneamente la tasa de enfriamiento por iteración.

Un caso destacable sería el del algoritmo luciérnaga. En este caso, el modelo parece reconocer el algoritmo, tanto teórica como prácticamente. A lo largo de la implementación, va generando las operaciones características de este, pero falla al implementar una operación específica que compara nuevas soluciones con la población original. Al implementarse mal la comparación, no se da evolución, por lo que el algoritmo es inútil. De nuevo, es un error menor que podría modificarse manualmente, pero que inhabilita el funcionamiento del algoritmo.

Otro casos concretos son implementaciones que se reconocen teóricamente y en los que se implementan todas las características del algoritmo, pero con errores conceptuales. En estos casos estamos frente a implementaciones correctas, reconocibles como el algoritmo que buscamos, pero bien faltando un concepto característico del algoritmo o bien cambiando estos. Aunque estas modificaciones no inutilizan el algoritmo por completo ni lo hacen irreconocible, si que lo hacen sub-óptimo. Ejemplos claros son la falta de componentes de diversificación, trabajando con individuos en lugar de poblaciones o generando copias en lugar de mutaciones. Ejemplos pueden ser el algoritmo competición imperialista con su falta de componente competitivo o el algoritmo genético con sus errores a la hora de cruzar padres y generar hijos.

De nuevo, estos cambios hacen de las implementaciones versiones sub-óptimas de los algoritmos, pero aún reconocibles y correctas.

A modo de conclusión, vemos que el modelo de lenguaje es capaz de reconocer la mayoría de algoritmos, 16 de los 25 evaluados. Al generar las implementaciones, el modelo empeora, donde de los 16 algoritmos reconocidos solo 9 se generan sin ningún tipo de error. Entre los 7 restantes, tres de ellos se generan con errores menores, mientras que los cuatro restantes se generan con cambios que afectan negativamente al rendimiento.



### 6.3. Resultados experimentales

A modo de anexo, se pueden consultar los resultados de las cien pruebas realizadas, resultados finales y gráficas generadas en el enlace[76].

En la tabla 6.2 se tiene un resumen de las clasificaciones de cada algoritmo en distintas métricas. Este “ranking” se ha realizado a partir de las clasificaciones de estos algoritmos sobre cada problema individualmente.

Así, podemos clasificar los algoritmos en función de su mejor coste obtenido (primera columna), de sus costes medios a lo largo de cada problema (segunda columna), de su número de evaluaciones (tercera columna) y de la relación de esas evaluaciones sobre el tiempo de ejecución (cuarta columna).

En cuanto a los mejores costes obtenidos, podemos obtener distintas conclusiones. En primer lugar, al igual que al estudiar la calidad de las implementaciones obtenidas, tenemos 9 algoritmos que tras la ejecución obtienen peores resultados que una búsqueda aleatoria simple.

Esto de por si ya sería motivo para ignorar estas implementaciones, pues siendo más complejas y costosas no logran superar resultados completamente aleatorios. Aún así, nos encontramos diferentes casuísticas.



Tabla 6.2: Tabla de los rankings obtenidos tras 100 problemas aleatorios

Algoritmo	Ranking Mejor Coste	Ranking Coste Medio	Ranking Evaluaciones	Ranking Evaluaciones vs Tiempo
Algoritmo Colonia Hormigas	<b>2.38</b>	8.61	<b>1.48</b>	<b>1.39</b>
Búsqueda Tabú	4.32	4.25	19.9	18.56
Busqueda local primer vecino	4.46	7.44	10.2	20.67
Busqueda local mejor vecino	4.96	4.72	17.4	21.05
Algoritmo genético	5.58	5.24	14.31	13.27
Búsqueda vecindario variable	5.725	<b>2.54</b>	5.14	4.83
Dinámica formación ríos	6.11	2.82	14.88	13.83
Enfriamiento simulado	6.175	2.73	16.38	15.17
Búsqueda dispersa	6.435	6.92	6.32	5.97
Búsqueda de cuervos	10.77	11.59	8.9	8.34
Sistema Inmune artificial	11.315	13.0	12.82	11.97
Algoritmo imperialista competitivo	12.39	10.51	8.77	8.3
Algoritmo cultural	12.41	10.63	10.12	9.57
Algoritmo optimización caos	12.88	14.01	21.0	19.83
Colonia de abejas	14.09	17.57	18.08	16.83
<b>Búsqueda aleatoria</b>	<b>17.4</b>	<b>22.16</b>	<b>24.42</b>	<b>24.42</b>
Algoritmo murcielago	18.46	19.14	24.33	24.3
Busqueda armonica	18.49	15.61	20.53	19.68
Algoritmo maleza invasora	18.67	15.84	3.13	3.02
Algoritmo agujero negro	19.99	24.28	17.42	16.17
Evolución diferencial	20.84	16.97	3.17	2.86
Busqueda gravitatoria	20.98	22.06	10.99	10.36
Enjambre de partículas	21.64	21.92	8.69	8.18
Algoritmo luciernagas	23.53	22.15	23.2	23.2
Búsqueda local guiada	25.0	22.29	3.42	3.23



Los algoritmos de maleza invasora, agujero negro y evolución diferencial, aún estando medianamente bien implementados, tienen ciertas modificaciones que, sin considerarse completamente erróneas, hacen que la implementación este falta en diversificación. Por ello, aunque el algoritmo sea funcional y sus soluciones evoluciones, están carentes de un componente de diversificación que les permita escapar de óptimos locales.

Un caso que destaca es el de la búsqueda local guiada, que obtiene los peores resultados en todos y cada uno de los problemas. Al ser una implementación errónea, generando una búsqueda local sin ningún método para escapar del óptimo local, en cuanto converge al primer óptimo local encontrado el algoritmo deja de avanzar.

Otro caso destacable es de la búsqueda gravitacional, la cual tiene un error menor que selecciona vecinos aleatorios en lugar de mejorar una propia solución óptima. Por ello, el algoritmo pierde gran parte del componente de intensificación necesario para converger a una solución óptima.

Por otro lado, en los 15 algoritmos mejores que la búsqueda aleatoria vemos diferentes casuísticas. Nos encontramos con algoritmos que sin ser la implementación esperable, como el sistema inmune artificial, la colonia de abejas o la dinámica de formación de ríos, tienen un rendimiento mejor de lo esperable.

El caso más destacable es el de la dinámica de formación de ríos, que aún siendo nada más que una búsqueda aleatoria que evalúa sus vecinos, consigue resultados bastante positivos. Esta búsqueda aleatoria, sobrada en diversificación, añade un punto de intensificación al evaluar el vecindario de cada nueva solución aleatoria que equilibra el rendimiento del mismo.

Además, vemos que los algoritmos que han sido perfectamente implementados, como la colonia de hormigas, búsqueda local o la búsqueda tabú obtienen los mejores resultados.

El más destacable es sin duda, el algoritmo de colonia de hormigas, con un rendimiento muy superior a los demás algoritmos. Partiendo de que es uno de los algoritmos implementado sin ningún tipo de error, es una metaheurística idónea para tratar problemas de TSP. Al tratar problemas de distinto tamaño, desde 50 a 100 ciudades en problemas TSP, vemos que el algoritmo es idóneo para este problema sin importar la dimensión del problema.



La consistencia y eficacia del algoritmo de colonia de hormigas puede atribuirse a su capacidad de aprovechar principios inspirados en el comportamiento de las colonias de hormigas. Al inspirarse en un proceso natural basado en la obtención de caminos óptimos esquivando obstáculos, resulta idóneo para tratar nuestro problema.

En cuanto a los costes medios obtenidos por problemas, vemos que únicamente 2 de los 25 problemas obtienen peores costes medios que una búsqueda aleatoria. Esto significa que a lo largo del proceso, las soluciones no evolucionan, o al menos, no más que una simple búsqueda aleatoria. Por lo tanto, podemos afirmar que las implementaciones obtenidas para los algoritmos de búsqueda local guiada y agujero negro son completamente inútiles, pues se estancan en el primer óptimo obtenido, sin posibilidad de diversificar su solución.

Por otro lado, vemos que los algoritmos basados en trayectorias, como lo son las búsquedas locales por mejor vecino, búsqueda por vecindario variable o enfriamiento simulado convergen rápidamente a soluciones sub-óptimas.

Aún sin obtener la mejor solución, son algoritmos a tener en cuenta, pues obtienen soluciones de cierta calidad en poco tiempo. En cambio, el algoritmo de colonia de hormigas tarda más en converger a una solución óptima, pero cuando lo hace, converge hasta la mejor solución encontrada.

En cuanto a evaluaciones, los algoritmos se clasifican en función al número de evaluaciones realizadas. Los algoritmos mejor clasificados en este aspecto serán aquellos que realicen menos evaluaciones. Por esto, la búsqueda aleatoria es de los peores algoritmos, siendo un algoritmo sencillo que realiza muchas más evaluaciones que los demás.

Por otro lado, vemos que el algoritmo de colonia de hormigas es el más lento en cuanto a evaluaciones, ya que es el algoritmo que menos evaluaciones consigue realizar durante el tiempo de ejecución. Esto se puede explicar a través de la implementación, al ser un algoritmo relativamente complejo en funcionamiento, y estando este perfectamente implementado, será de los más lentos. Por lo demás, vemos variedad en cuanto al número de evaluaciones para cada algoritmo.



Por último, podemos evaluar el número de evaluaciones sobre el tiempo. Estos se clasifican por la cantidad de evaluaciones que logran hacer por segundo, clasificándose como mejores aquellos algoritmos más lentos. Teniendo en cuenta que todos los algoritmos tienen el mismo tiempo de ejecución, 10 minutos, es un reflejo del número de evaluaciones.

Aún así, podemos ver dos casos excepcionales. Más allá de los algoritmos mal implementados, desatacan dos algoritmos por su velocidad, los algoritmos de búsqueda por primer y mejor vecino. Mientras tienen un “alto” número de evaluaciones por segundo, las propias implementaciones tiene un límite de evaluaciones.

Así, estos algoritmos se detienen automáticamente al llegar a  $N$  evaluaciones sin mejorar su resultados. Además, al no tener ningún elemento de diversificación, los algoritmos se estancan en el primer óptimo local obtenido. Por esto, ninguno de estos algoritmos llega a los 10 minutos de ejecución. Por lo demás, la relación de evaluaciones sobre el tiempo es la esperable.

Aún así, podemos visualizar el rendimiento de las implementaciones sobre el conjunto de problemas de forma gráfica. Podemos ver la clasificación según su mejor valor obtenido a lo largo de las evaluaciones en la figura 6.26.

## Resultados

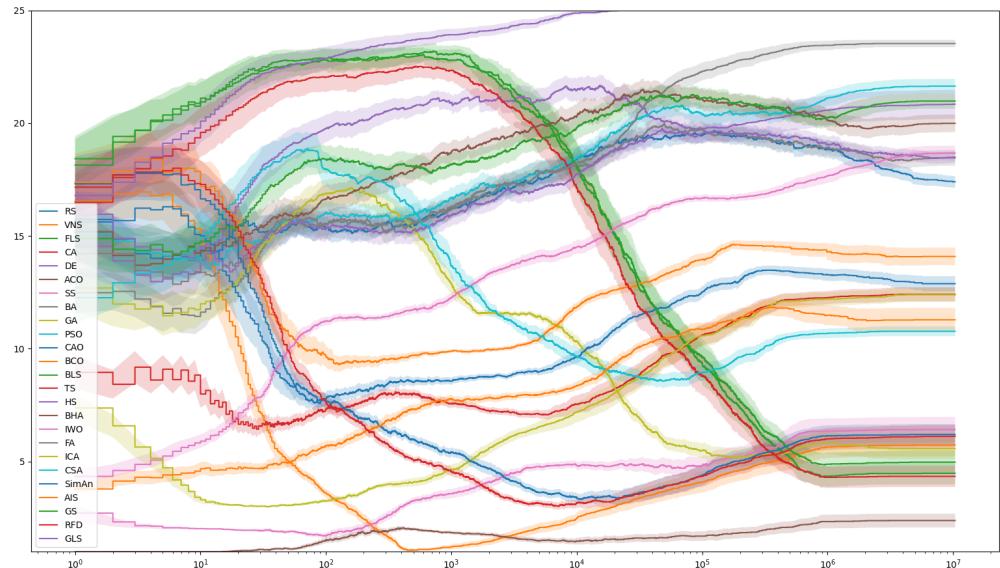


Figura 6.26: Ranking rendimiento de los 25 algoritmos

Además, podemos ver como cada algoritmo converge a un resultados finales en la figura 6.27.

## Resultados

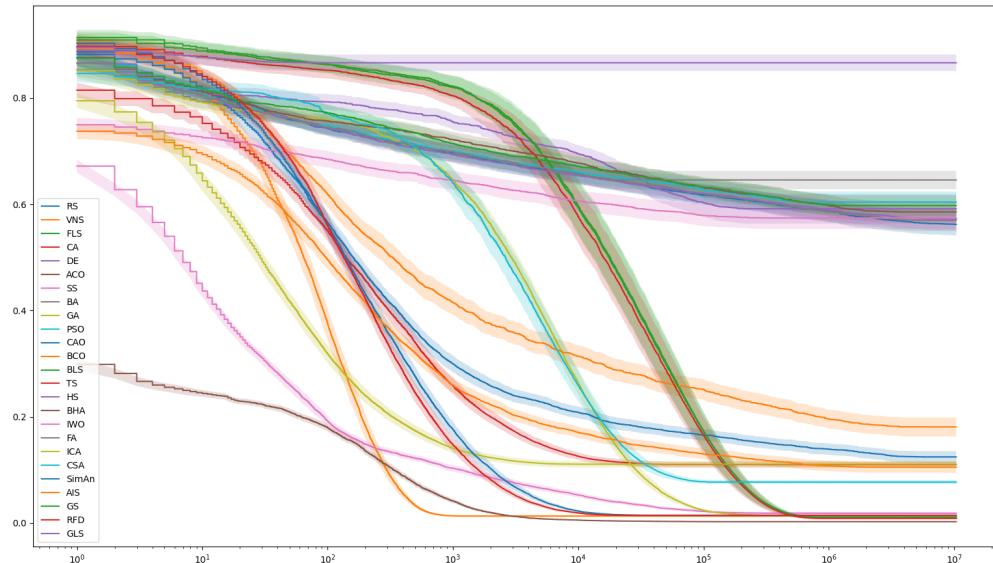


Figura 6.27: Gráfica convergencia de los 25 algoritmos

Obviamente, al tener información de 25 algoritmos en una misma gráfica la hace ilegible por la sobrecarga de información. Así, podemos dividir esta información en familias, agrupando algoritmos con mismas características y bases de funcionamiento similares.

## Resultados

En primer lugar, podemos visualizar el rendimiento de los algoritmos inspirados en procesos naturales, tanto en clasificación por rendimiento en la figura 6.28 como por convergencia en la figura 6.29.

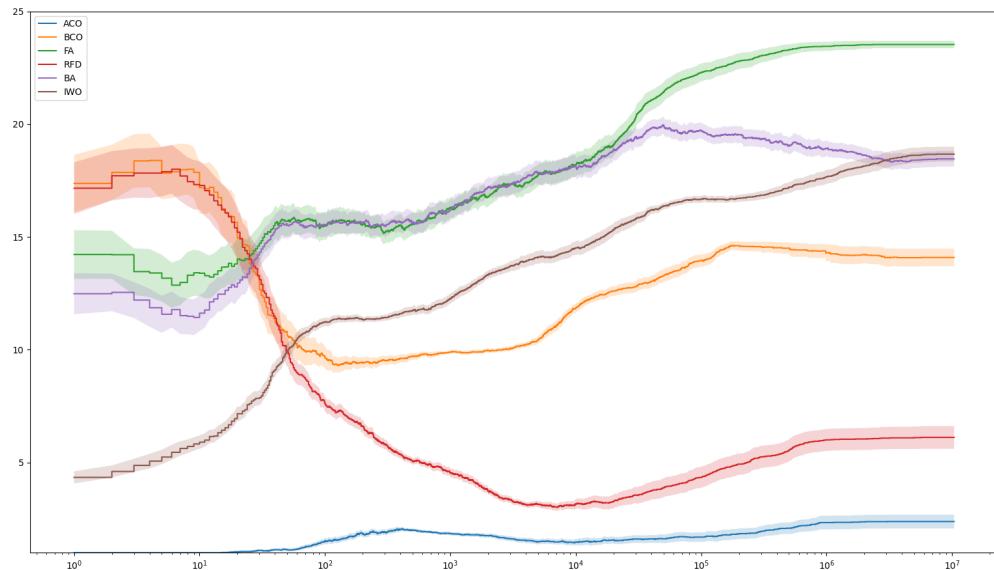


Figura 6.28: Ranking rendimiento algoritmos naturales

En cuanto a clasificación, vemos lo esperable. Los algoritmos mal implementados, como el algoritmo de luciérnagas (FA), el algoritmo de maleza invasora (IWO) o el algoritmo murciélagos (BA) van empeorando al compararse con los algoritmos bien implementados.

Los casos destacables son el del algoritmo de dinámica de formación de ríos (RFD), que aún sin implementar el algoritmo esperado tiene un rendimiento muy superior a los demás, y el algoritmo de la colonia de hormigas, que es mejor que cualquier otro algoritmo de esta categoría. Además, este algoritmo destaca por ser el mejor de entre los 25 algoritmos en las iteraciones iniciales, sin ningún tipo de competencia hasta superar las 50 evaluaciones.

La gráfica de convergencia 6.29 nos muestra la importancia de mantener un equilibrio entre diversificación e intensificación. Los algoritmos mal implementados tardan más en converger, y en cuanto lo convergen, lo hacen a una solución subóptima. En cambio, los algoritmos bien implementados si logran converger a una solución óptima y en menos evaluaciones.

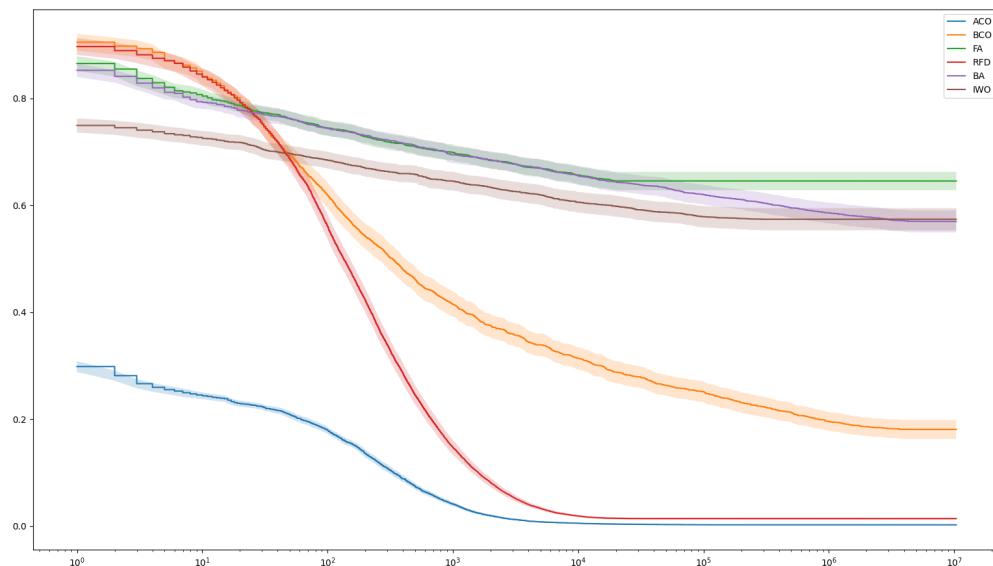


Figura 6.29: Gráfica convergencia algoritmos naturales

El algoritmo de dinámica de formación de ríos consigue converger a una solución cercana a la óptima en  $10^4$  evaluaciones, mientras que el algoritmo de la colonia de hormigas, empezando por una solución inicial mucho mejor que los demás, consigue converger a la solución óptima en menos de  $10^4$  evaluaciones.

Al final, lo más destacable es la capacidad del algoritmo de colonia de hormigas de empezar a partir de soluciones inicialmente buenas.

## Resultados

En los algoritmos poblaciones, o basados en poblaciones, también se incluyen varios de los algoritmos basados en procesos naturales. Podemos ver las clasificación por rendimiento en la figura 6.30 y la gráfica de convergencia en la figura 6.31.

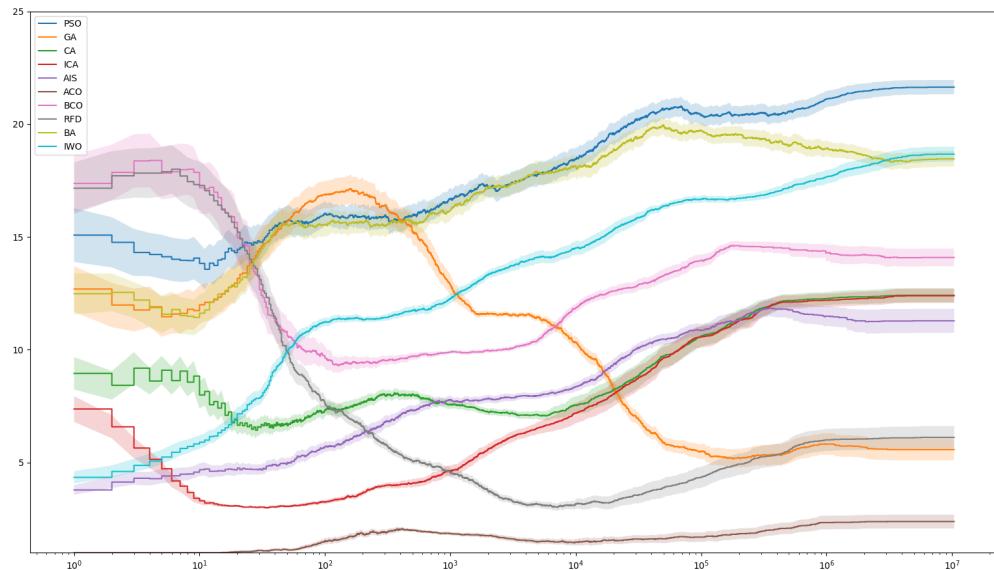


Figura 6.30: Ranking rendimiento algoritmos poblaciones

En cuanto a rendimiento, podemos ver que con el paso de las iteraciones solo los algoritmos de colonia de hormigas, genético y dinámica de formación de ríos son los únicos que acaban siendo mejores que la mayoría. Mientras que por debajo de mil iteraciones el rendimiento del algoritmo genético es de los peores, el algoritmo imperialista competitivo solo se ve superado por el algoritmo de colonia de hormigas.

Con la gráfica de convergencia 6.31 podemos obtener varias conclusiones.

Algoritmos mal implementados, como el algoritmo cultural, el algoritmo imperialista competitivo o el algoritmo de sistema inmune artificial consiguen converger a soluciones sub-óptimas similares, mostrando un aparente equilibrio entre diversificación e intensificación.

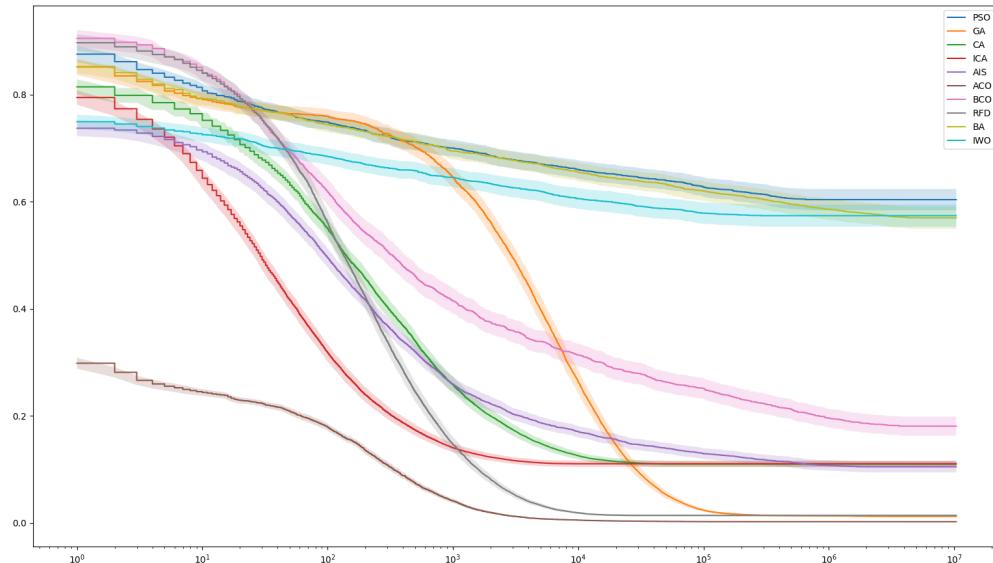


Figura 6.31: Gráfica convergencia algoritmos poblaciones

En cambio, otros algoritmos mal implementados, como el algoritmo de maleza invasora, algoritmo del enjambre de partículas o el algoritmo de murciélagos convergen a soluciones distantes de la óptima, mostrando una gran diversificación sin capacidad de converger a soluciones de calidad.

Por otro lado, el algoritmo genético, que comienza con un rendimiento malo, siendo el peor alrededor de las  $10^2$  evaluaciones, consigue un equilibrio entre diversificación e intensificación, consigue obtener resultados muy cercanos a los óptimos.

## Resultados

Sobre los algoritmos basados en trayectorias, podemos ver la clasificación por rendimiento en la figura 6.32.

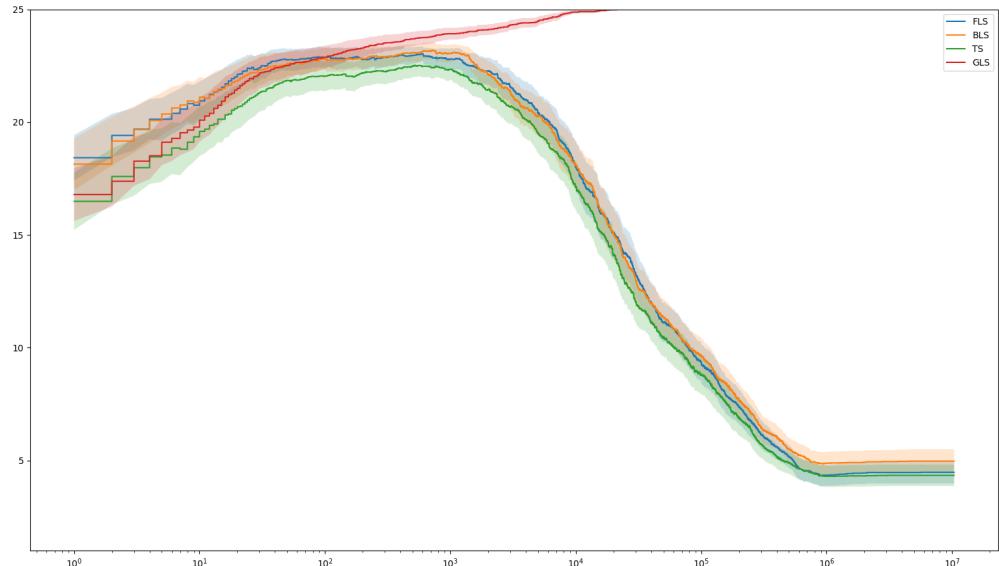


Figura 6.32: Ranking rendimiento algoritmos búsqueda local

El algoritmo de búsqueda local guiada (GLS) es el peor con diferencia, pues rápidamente se estanca en el primer óptimo local que encuentra. Esto se ve reflejado aún más el gráfica de convergencia 6.33.

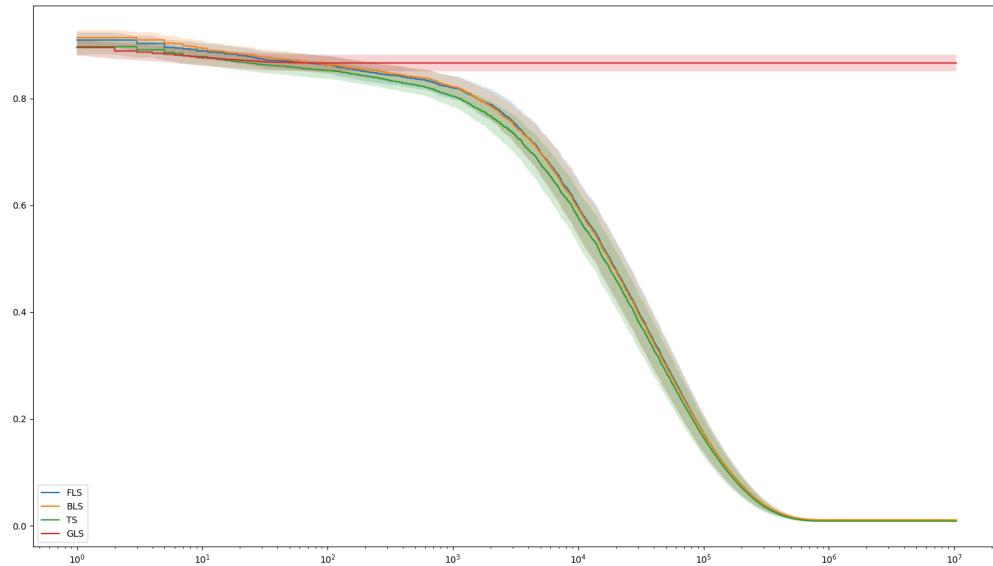


Figura 6.33: Gráfica convergencia algoritmos busqueda local

Mientras que el algoritmo de búsqueda local guiada converge a una solución sub-óptima en apenas  $10^2$  iteraciones donde deja de mejorar, las demás implementaciones tienen un rendimiento similar, pues los dos algoritmos de búsqueda local y el algoritmo de búsqueda tabú diversifican por más de  $10^5$  evaluaciones, donde empiezan a converger hacia soluciones cercanas a la óptima.

En los algoritmos basados en procesos evolutivos vemos distintos comportamientos. Podemos ver la clasificación por rendimiento en la gráfica 6.34 y la gráfica de convergencia en la figura 6.35.

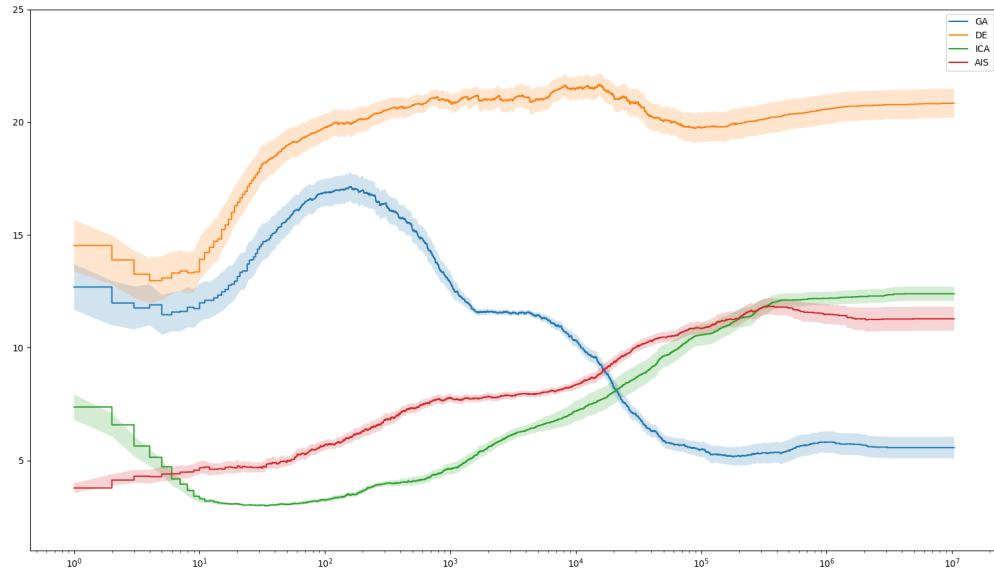


Figura 6.34: Ranking rendimiento algoritmos evolutivos

Con respecto a los algoritmos genético (GA) y de evolución diferencial (DE) vemos que ambos comienzan siendo mediocres, obteniendo peores resultados comparados a otros algoritmos hasta alrededor de las  $10^2$  evaluaciones. A partir de este punto, el algoritmo de evolución diferencial, con su evaluación completamente errónea, se ve superado por más algoritmos con el paso de las evaluaciones, mientras que el algoritmo genético sigue mejorando su rendimiento con el paso de las evaluaciones.

En cambio, el algoritmo de sistema inmune artificial (AIS) y el algoritmo imperialista competitivo, ambas implementaciones erróneas, comienzan con un rendimiento muy superior al resto de algoritmos, pero rápidamente empeoran, viéndose ambos superados por gran cantidad de algoritmos.

Con respecto a como convergen a soluciones óptimas, el algoritmo genético es que más tarda en converger a una solución, pero converge a una solución cercana a la óptima.

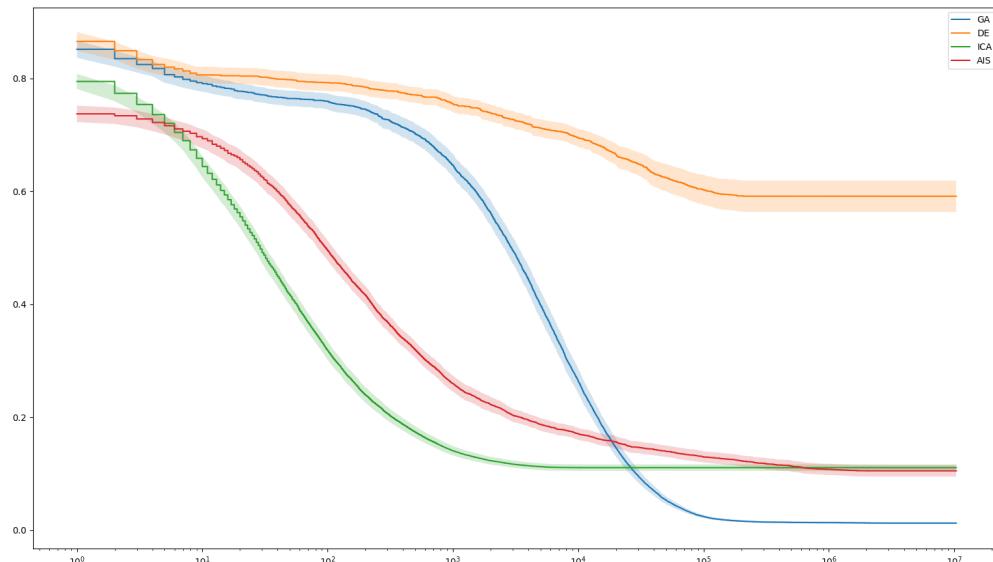


Figura 6.35: Gráfica convergencia algoritmos evolutivos

En cambio, el algoritmo de sistema inmune artificial y el algoritmo imperialista competitivo convergen ambos a soluciones similares, ambas sub-óptimas, donde destaca el algoritmo imperialista competitivo por converger en muchas menos evaluaciones a esas soluciones.

## Resultados

Por último, se puede comparar el rendimiento de los peores algoritmos. Podemos ver su gráfica de convergencia en la figura 6.36.

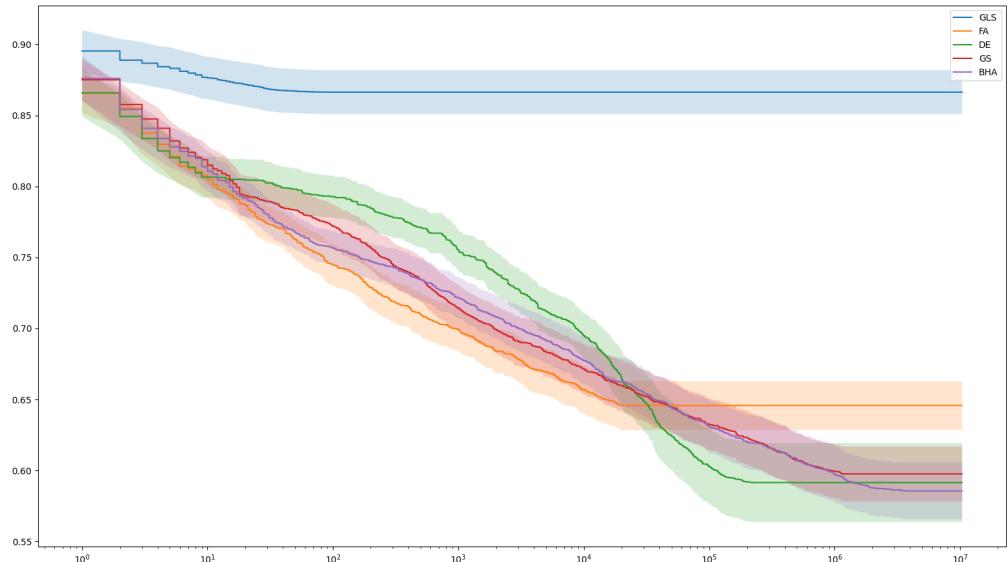


Figura 6.36: Gráfica convergencia 5 peores algoritmos

Todos estos algoritmos, erróneos en implementación, tienen un rendimiento similar, convergiendo a soluciones similares en costo y alejadas de la solución óptima.

Además, es destacable que todos estos algoritmos convergen a dichas soluciones en un número de evaluaciones similar. La única excepción es el algoritmo de búsqueda local guiada (GLS), que como se ha visto anteriormente, es el peor en rendimiento, convergiendo al primer óptimo encontrado.

## Resultados

De la misma forma, se puede evaluar el rendimiento de los mejores algoritmos. En la figura 6.37 se puede ver su clasificación en función del rendimiento.

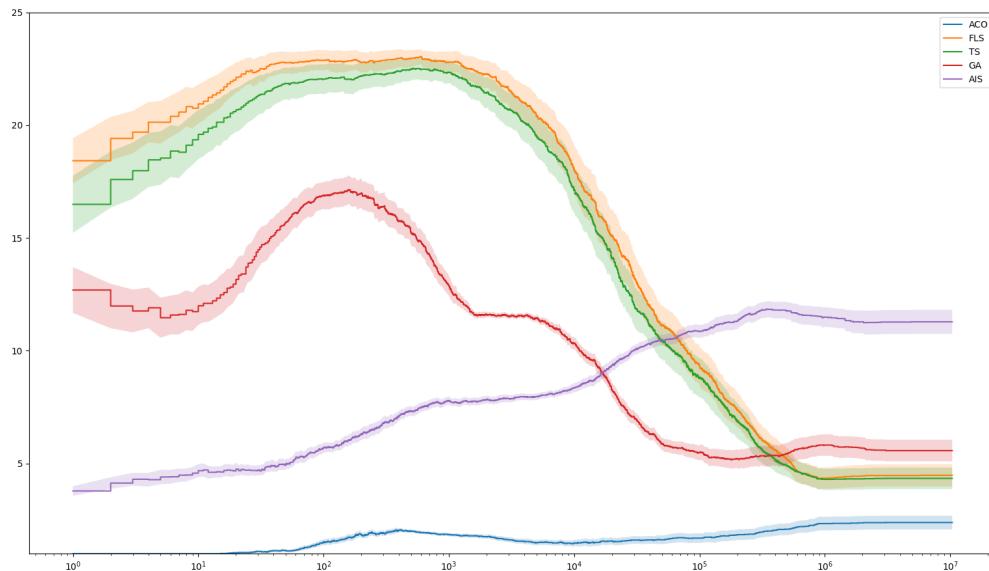


Figura 6.37: Ranking rendimiento 5 mejores algoritmos

Todos estos algoritmos son individualmente destacables. Mientras que el algoritmo de colonia de hormigas (ACO) y el algoritmo del sistema inmune artificial (AIS) parten con un rendimiento inicial muy superior a los demás, sólo la colonia de hormigas es capaz de mantener los buenos resultados con el paso de las iteraciones mientras que el algoritmo del sistema inmune artificial se ve superando con el paso de las evaluaciones, siendo el “peor entre los mejores”.

Con la gráfica de convergencia, podemos apreciar que el algoritmo de colonia de hormigas es el mejor con diferencia, no solamente por ser el algoritmo que mejores soluciones obtiene, sino por converger rápidamente a esta solución óptima.

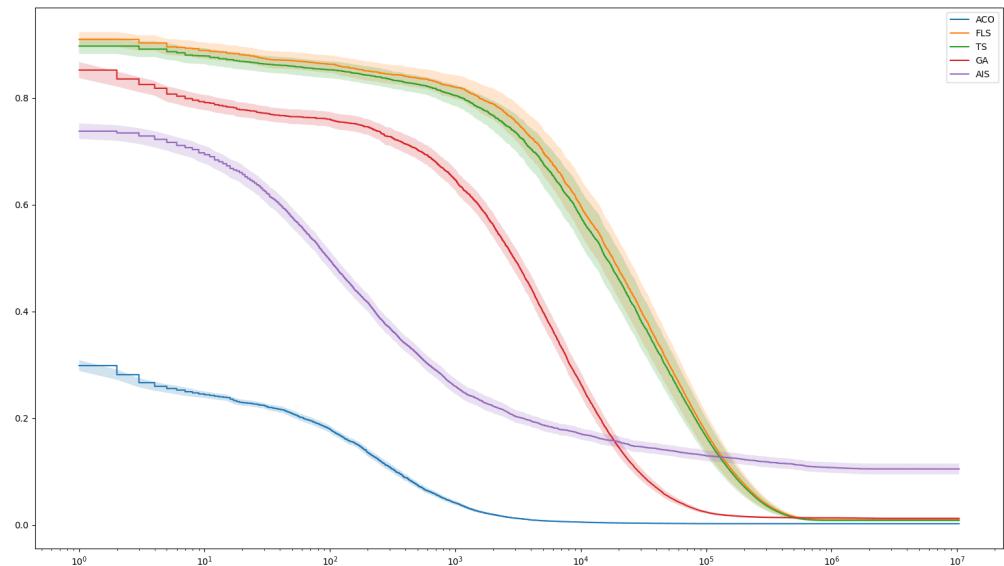


Figura 6.38: Gráfica convergencia 5 mejores algoritmos

Es destacable que las soluciones obtenidas por el algoritmo de sistema inmune artificial son las peores, mientras que los algoritmos de búsqueda tabú y búsqueda local por primer vecino tienen un rendimiento prácticamente igual, convergiendo a soluciones similares en el mismo número de evaluaciones. En cambio, el algoritmo genético converge a soluciones ligeramente peores en menos evaluaciones.



Una vez evaluado el rendimiento real sobre los problemas, podemos evaluar las implementaciones a través de test no paramétricos. Partiremos de la hipótesis nula por la que los algoritmos son iguales en rendimiento.

En cuanto al test de Wilcoxon, se han evaluado dos parejas de algoritmos, los mejores algoritmos, Colonia de hormigas (ACO) y Búsqueda local por primer vecino (FLS), y por otro lado los peores, Búsqueda local guiada (GLS) y Algoritmo luciérnaga (FA).

Esta selección se ha basado en los rankings de rendimiento vistos anteriormente, buscando comprobar si realmente existen diferencias significativas entre los dos mejores algoritmos y entre los dos peores. Para ello, se compara el rendimiento dentro de cada par, es decir, comparando el rendimiento entre los dos mejores algoritmos y repitiendo el proceso con los dos peores.

Los resultados se reflejan en la tabla 6.3.

Tabla 6.3: Resultados tests de Wilcoxon

Algoritmos	Valor T	Valor P	Valor $\alpha$	Conclusión
Mejores Algoritmos	874,5	$2,32 \cdot 10^{-8}$	0,05	Diferencias significativas
Peores Algoritmos	0,0	$3,89 \cdot 10^{-18}$	0,05	Diferencias significativas

Con respecto a los mejores algoritmos, el valor de T nos indica la magnitud de la diferencia entre los algoritmos. Un valor tan alto, de 874.5 nos sugiere diferencias en el rendimiento. Además, el valor de P  $2,32 \cdot 10^{-8}$  es mucho menor que el nivel de significancia  $\alpha$  (0.05), podemos afirmar que hay una diferencia significativa en el rendimiento de los algoritmos.

Algo similar pasa con los peores algoritmos. En este caso el valor de T es 0, lo que nos podría sugerir que no hay diferencias significativas. A pesar de esto, un valor de P tan pequeño, en este caso  $3,89 \cdot 10^{-18}$ , nos niega la hipótesis nula.

Estos resultados concuerdan con los resultados de rendimiento real vistos anteriormente, viendo como el algoritmo de colonia de hormigas era muy superior en rendimiento mientras que el algoritmo de búsqueda local guiada es el peor algoritmo en todos los problemas.



En cuanto a los test de Friedman, los resultados obtenidos por la prueba de Friedman propia de la librería Scipy[26] se reflejan en la tabla 6.4.

Tabla 6.4: Resultados test de Friedman

Algoritmos	Estadístico	Valor P	Valor $\alpha$	Conclusión
Mejores algoritmos	212,42	$8,006 \cdot 10^{-45}$	0,05	Diferencias significativas
Peores algoritmos	279,92	$2,31 \cdot 10^{-59}$	0,05	Diferencias significativas
Algoritmos naturales	475,78	$1,34 \cdot 10^{-100}$	0,05	Diferencias significativas
Algoritmos poblacionales	819,29	$1,49 \cdot 10^{-170}$	0,05	Diferencias significativas
Algoritmos trayectorias	182,84	$2,14 \cdot 10^{-39}$	0,05	Diferencias significativas
Algoritmos evolutivos	261,25	$2,4 \cdot 10^{-56}$	0,05	Diferencias significativas
Todos los algoritmos	2207,77	0,0	0,05	Diferencias significativas

De estos resultados, podemos obtener las mismas conclusiones que del test de Wilcoxon. En todos los tests tenemos valores estadísticos muy altos, lo que nos sugiere una posible diferencia en el rendimiento. Además, los valores de P cercanos a cero nos confirman que sí hay diferencias significativas en cuanto a rendimiento.

Viendo los resultados reales reflejados en la tabla 6.2, los resultados de los test no paramétricos son los esperables, mostrando una clara diferencia entre los algoritmos.



Aún así, los valores de P resultan extraordinariamente pequeños. Ante la posibilidad de estar obteniendo resultados erróneos por el proceso propio de la librería Scipy, los test de Friedman se han repetido “manualmente”.

Los resultados se reflejan en la tabla 6.5.

Tabla 6.5: Resultados test de Friedman manual

Algoritmos	Estadístico	Valor P	Valor $\alpha$	Conclusión
Mejores algoritmos	8739,02	0,0	0,05	Diferencias significativas
Peores algoritmos	127790,11	0,0	0,05	Diferencias significativas
Algoritmos naturales	52727,89	0,0	0,05	Diferencias significativas
Algoritmos poblacionales	20540,94	0,0	0,05	Diferencias significativas
Algoritmos trayectorias	62336,23	0,0	0,05	Diferencias significativas
Algoritmos evolutivos	46509,61	0,0	0,05	Diferencias significativas
Todos los algoritmos	4978,47	0,0	0,05	Diferencias significativas

Estos resultados nos confirman los obtenidos automáticamente con la función propia de la librería Scipy. Viendo que en todos los casos el valor de p es menor que el valor de significancia  $\alpha$ , podemos afirmar que hay diferencias significativas en el rendimiento de los algoritmos, tanto en el conjunto de 25 algoritmos como en los diferentes subgrupos evaluados.



Una vez confirmada una diferencia de rendimientos entre los algoritmos, podemos aplicar un test post-hoc para determinar dónde se encuentran dichas diferencias. El planteamiento es aplicar un test de Nemenyi para cada uno de los test de Friedman. Dentro de estos test de Nemenyi podemos obtener varias conclusiones.

En primer lugar, podemos analizar las diferencias entre los algoritmos basados en procesos naturales, representadas en la figura 6.39.

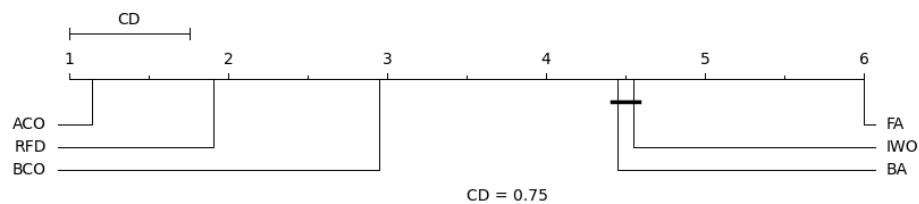


Figura 6.39: Resultados Nemenyi algoritmos naturales

De la misma forma que se veía en la figura 6.28, podemos ver disparidad en el rendimiento de los algoritmos, destacándose el algoritmo de colonia de hormigas (ACO) por ser el mejor en la mayoría de casos y por otro lado, los algoritmo de búsqueda de murciélago (BA) y de maleza invasora (IWO) por su rendimiento similar, siendo en los únicos donde no se aprecian diferencias significativas.

## Resultados

En cuanto a los algoritmos basados en poblaciones, los resultados del test de Nemenyi pueden verse en la figura 6.40.

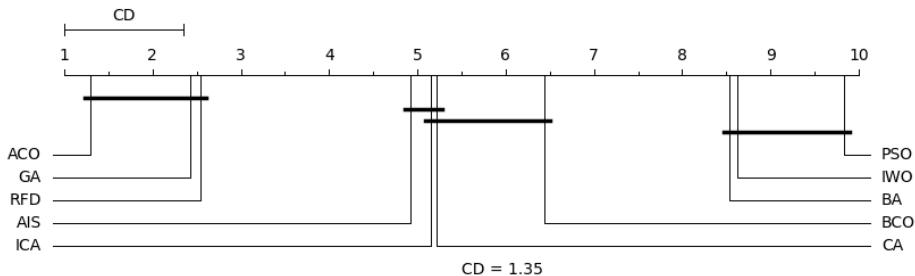


Figura 6.40: Resultados Nemenyi algoritmos poblacionales

Al tratar con un mayor número de algoritmos vemos una mayor disparidad, así como una mayor diferencia crítica.

Al estar comparando algunos de los mejores algoritmos con algunos de los peores, vemos pequeños subgrupos. Comparados con los demás algoritmos, los mejores 3 mejores algoritmos parecen similares, de la misma forma que los 3 peores no muestran diferencias significativas. Esto se analizará posteriormente.

Además, los casos más significativos son los de los algoritmos de sistema inmune artificial (AIS), el imperialista competitivo (ICA) y el cultural (CA), donde muestran un rendimiento muy similar, sin mostrar diferencias significativas.



Sobre los algoritmos basados en trayectorias o búsquedas locales, podemos ver los resultados del test de Nemenyi en la figura 6.41.

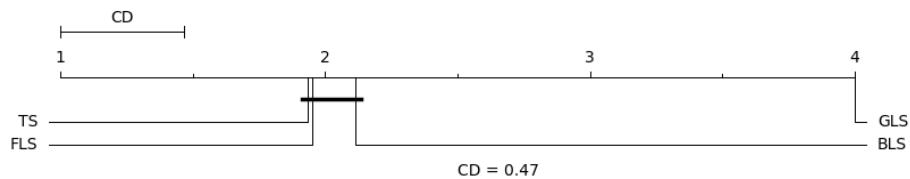


Figura 6.41: Resultados Nemenyi algoritmos trayectorias

Destaca la falta de diferencias significativas entre los algoritmos de búsqueda tabú (TS), por primer vecino (FLS) y por mejor vecino (BLS).

También destaca que el algoritmo de búsqueda tabú tenga un rendimiento tan similar a la búsqueda local por primer vecino, no solo porque las búsquedas locales sean muy similares, si no porque la búsqueda local por primer vecino está mal implementada, siendo de hecho una búsqueda local por mejor vecino.

Por último, el rendimiento de la búsqueda local guiada (GLS) no sorprende, pues ha sido el peor algoritmo de nuestro estudio en todos y cada uno de los problemas.



Los resultados del test de Nemenyi sobre los algoritmos basados en procesos evolutivos pueden verse en la figura 6.42.

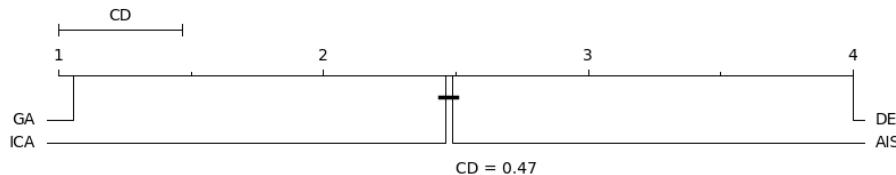


Figura 6.42: Resultados Nemenyi algoritmos evolutivos

Se demuestran las diferencias significativas en los algoritmos, tanto por el algoritmo genético (GA), como del algoritmo de evolución diferencial (DE).

Destaca la falta de diferencias significativas entre los algoritmos de sistema inmune artificial e imperialista competitivo, también visible en el test de Nemenyi de los algoritmos basados en poblaciones.



## Resultados

Por ultimo, podemos ver los resultados de los test de Nemenyi de los peores y mejores algoritmos de nuestro estudio, representados en las figura 6.43 y 6.44 respectivamente.

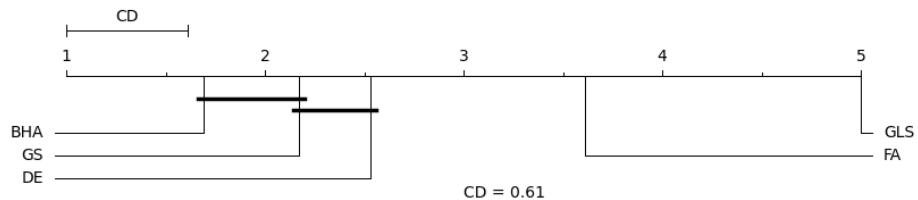


Figura 6.43: Resultados Nemenyi peores

En cuanto a los peores algoritmos, vemos similitudes entre los algoritmos de búsqueda de agujero negro y búsqueda gravitacional, y entre los algoritmos de búsqueda gravitacional y evolución diferencial.

Por otro lado destacan los algoritmos de búsqueda local guiada (GLS) y luciérnaga (FA), por ser los peores con diferencia.

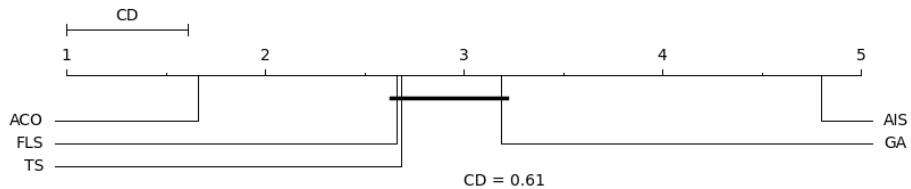


Figura 6.44: Resultados Nemenyi mejores algoritmos

En cuanto a los mejores algoritmos, como se veía en la 6.37, vemos que el algoritmo de colonia de hormigas (ACO) es el mejor con diferencia, mientras que el algoritmo de sistema inmune artificial es el “peor entre los mejores”.

Destacan los algoritmos de búsqueda por primer vecino (FLS), búsqueda tabú (TS) y genético (GA) por su falta de diferencias significativas en su rendimiento, sobre todo en los casos de la búsqueda tabú y la búsqueda por primer vecino.

A modo de resumen, se puede apreciar una gran disparidad en el rendimiento de los algoritmos implementados. De los 25 algoritmos, solo 4 se acercan a soluciones óptimas o sub-óptimas, destacándose un solo algoritmo, la colonia de hormigas, por ser el único que llega a soluciones óptimas en la mayoría de problemas.



UNIVERSIDAD DE CÓRDOBA

## Resultados

---



# Capítulo 7

## Conclusiones

Como conclusión principal, se han logrado todos los objetivos estipulados para este estudio.

Como primer objetivo, hemos conseguido realizar un estudio en profundidad de las implementaciones generadas por un modelo largo de lenguaje, en este caso ChatGPT. Estudiando distintos algoritmos, con diferentes bases y principios de funcionamiento, hemos podido estudiar la capacidad de reconocimiento teórico en cuanto a algoritmos y su implementación.

Hemos concluido que ChatGPT tiene una capacidad teórica destacable, reconociendo los elementos y características propias de 16 de los 25 algoritmos implementados. A la hora de implementar código, el modelo de lenguaje empeora, siendo capaz de implementar correctamente solamente 9 de los algoritmos. Además, de los otros 7 algoritmos reconocidos, los 7 han sido implementaciones aproximadas qué, aún conteniendo errores y fallos de concepto, no inhabilitan su funcionamiento.

Las implantaciones generadas por el modelo de lenguaje han sido evaluadas sobre problemas generados aleatoriamente, mostrando una gran variedad de rendimientos. Desde algoritmos mal implementados obteniendo un rendimiento notable a algoritmos rápidos y eficaces, hemos demostrado que algunas de las implementaciones, por mucha variedad que haya, son eficientes y eficaces a la hora de resolver los problemas sobre los que se han evaluado.

Con respecto a la pregunta formulada en los objetivos de este estudio: “¿Puede un usuario sin conocimiento previo resolver problemas complejos con código generado por un modelo largo de lenguaje?”, la respuesta es afirmativa.



Con lo explorado en este estudio, hemos visto que un usuario sin necesario conocimiento teórico sobre metaheurísticas, su funcionamiento o su implementación puede usar modelos largos de lenguaje para generar implementaciones útiles y óptimas, capaces de resolver problemas inabordables manualmente, como el problema del viajero de comercio.

Como conclusión final, podemos afirmar que el modelo ChatGPT tiene un notable conocimiento teórico con respecto a algoritmos metaheurísticos. Esto, sumado a una capacidad de generar implementaciones correctas para ciertos algoritmos, puede concluir en algoritmos óptimos en rendimiento y resultados, implementados rápidamente y sin necesidad de conocimientos previos.



## 7.1. Posibles mejoras y cambios a futuro

El estudio realizado es solo una corta introducción al problema, pero es más que representativo. Aún así, este estudio puede expandirse en gran medida.

En primer lugar, además de usar el modelo ChatGPT-3.5 para generar implementaciones, se pueden usar otros, como pueden ser ChatGPT-4, el modelo de pago de OpenAI[1], Bard[20], Microsoft Copilot[19], entre muchos otros.

Así, el estudio puede enfocarse en la capacidad de generación de código por parte de distintos modelos, con distintas estructuras, número de parámetros y datos de entrenamiento.

Por otro lado, en cuanto a algoritmos, podemos expandir el estudio aún más. Sobre metaheurísticas, se pueden implementar y comparar una gran cantidad de algoritmos metaheurísticos, subiendo la cantidad de algoritmos evaluados de 25 a cientos. Además, una expansión aún mayor puede ser la evaluación y generación de distintos tipos de algoritmos. Desde algoritmos de búsqueda, de ordenamiento, hasta clasificadores o modelos sencillos de redes neuronales. El estudio se puede expandir a evaluar la generación de código de cualquier tipo.

Otro concepto interesante sería evaluar la capacidad del modelo de generar código a partir de lenguaje natural. En lugar de indicar el algoritmo a implementar, se puede usar como entrada al modelo texto en lenguaje natural, explicando el funcionamiento y elementos característicos del algoritmo mediante descripciones en lenguaje natural. A partir de las implementaciones generadas podemos evaluar la capacidad del modelo de interpretar el lenguaje natural y su posterior traducción a código de programación funcional.

A la hora de comparar y evaluar el rendimiento de los algoritmos, se pueden añadir más métricas, como gasto de memoria, robustez, o evaluar los algoritmos sobre distintos problemas.

Por último, el código del proyecto puede mejorarse en gran manera, añadiendo la capacidad de incluir nuevos algoritmos de forma sencilla e inmediata, o reformulando el código entero, usando técnicas de código limpio y manteniendo una estructura modular y flexible para futuras expansiones.





UNIVERSIDAD DE CÓRDOBA

# Capítulo 8

## Manual de uso



El manual de uso pretende servir como guía para cualquiera que quiera replicar las pruebas de comparación y evaluación del rendimiento de los algoritmos, así como para quien quiera expandir el proyecto.

Para esto, el proyecto se podrá usar en dos modos distintos.

En primer lugar, se tendrá un modo ejecutable, que permita replicar los experimentos realizados, pudiendo modificar únicamente los parámetros de las pruebas y los hiperparámetros característicos de cada algoritmo.

Además, si se quiere modificar el proyecto en profundidad o expandirlo, se puede trabajar con el proyecto completo en Python.

Por último, todo el código del proyecto puede visualizarse y revisarse desde el repositorio en GitHub[77].

## 8.1. Versión ejecutable

La versión ejecutable, que se puede descargar a partir del enlace[78], es sencilla de usar. Solamente se necesitan dos ficheros, el ejecutable, y el archivo de configuración. Además, para la generación de gráficas y resultados, se debe generar una carpeta en el mismo directorio del ejecutable, la carpeta *data*.

En cuanto al archivo de configuración **config.json** se pueden indicar el número de problemas sobre los que probar, el tiempo de ejecución de cada algoritmo sobre cada problema, y para cada uno de los problemas a generar, el rango mínimo y máximo de ciudades con los que se generarán los problemas.

En este archivo también están especificados todos los hiperparámetros de los algoritmos, para poder evaluar también el impacto de estos sobre el rendimiento.



Un ejemplo del archivo **config.json** puede verse la imagen 8.1.

```
1  {
2      "number_problems": 2,
3      "max_runtime": 2,
4      "min_cities": 99,
5      "max_cities": 100,
6      "initial_pheromone": 1.0,
7      "num_empires": 5,
8      "num_colonies": 10,
9      "revolution_rate": 0.1,
10     "assimilation_rate": 0.1,
11     "num_clones": 100,
12     "mutation_rate_AIS": 0.2,
13     "population_size_BA" : 10,
14 }
```

Figura 8.1: Ejemplo de archivo de configuración

Se debe recalcar que en el archivo de configuración no están incluidos los hiperparámetros de todos los algoritmos, algunos por ser una implementación errónea y otros por no ser de utilidad.

Una vez ajustadas todas las configuraciones requeridas, basta con ejecutar el programa de pruebas, con:

`./programa_pruebas.exe`

Las gráficas y resultados se generarán en la carpeta *data*.



## 8.2. Proyecto completo

Si se quiere trabajar con el proyecto completo, pudiendo modificar cualquier parte de este, se puede descargar desde el enlace[79].

En primer lugar, se debe crear un entorno virtual para poder ejecutar el proyecto. Esto se puede hacer, en entornos Linux, con:

```
python -m venv entorno_virtual
```

Una vez creado el entorno y activado el entorno, se deben importar e instalar las librerías necesarias para el proyecto. Estas están en un documento del propio proyecto, *requirements.txt* . Las librerías requeridas se pueden instalar con:

```
pip install -r requirements.txt
```

Una vez instaladas las librerías necesarias, el proyecto ya puede ejecutarse. El proyecto se divide en 3 directorios:

- **test\_frame**: donde se encuentran dos archivos, **test\_frame.py**, que codifica el entorno de pruebas, y **numerical\_test.py**, donde se codifican los test no-paramétricos.
- **TSP\_Instance\_Experiment**: donde se encuentran dos archivos distintos, **tsp\_instance.py** y **tsp\_experiment.py**, que codifican las instancias de problemas de TSP y los problemas sobre los que experimentar, respectivamente.
- **Algorithms**: donde cada algoritmo se codifica en un archivo diferente.

Además, en la raíz del proyecto se encuentran el fichero de configuración **config.json** y el archivo **main.py**, base de la ejecución.



# Bibliografía

- [1] OpenAI and Josh Achiam et al. Gpt-4 technical report. 2023. doi: 10.48550/arXiv.2303.08774. URL <https://arxiv.org/abs/2303.08774>.
- [2] Hugo Touvron, Thibaut Lavril, and Gautier Izacard et al. Llama: Open and efficient foundation language models. 2023. doi: 10.48550/arXiv.2302.13971. URL <https://arxiv.org/abs/2302.13971>.
- [3] Abdulaziz Alorf. A survey of recently developed metaheuristics and their comparative analysis. *Engineering Applications of Artificial Intelligence*, 2023. ISSN 0952-1976. doi: <https://doi.org/10.1016/j.engappai.2022.105622>. URL <https://www.sciencedirect.com/science/article/pii/S0952197622006121>.
- [4] Saman Almufti, Awaz Shaban, Rasan Ali, and Jayson Fuente. Overview of metaheuristic algorithms. *Polaris Global Journal of Scholarly Research and Trends*, pages 10–32, 2023. doi: 10.58429/pgjsrt.v2n2a144. URL [https://www.researchgate.net/publication/370422753\\_Overview\\_of\\_Metaheuristic\\_Algorithms](https://www.researchgate.net/publication/370422753_Overview_of_Metaheuristic_Algorithms).
- [5] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2): 231–247, 1992. ISSN 0377-2217. doi: [https://doi.org/10.1016/0377-2217\(92\)90138-Y](https://doi.org/10.1016/0377-2217(92)90138-Y). URL <https://www.sciencedirect.com/science/article/pii/037722179290138Y>.
- [6] Chatgpt-3.5 models, . URL <https://platform.openai.com/docs/models/gpt-3-5>.
- [7] Junqin Xu and Jihui Zhang. Exploration-exploitation tradeoffs in metaheuristics: Survey and analysis. pages 8633–8638, 2014. doi: 10.1109/ChiCC.2014.6896450. URL <https://ieeexplore.ieee.org/document/6896450>.
- [8] Eneko Osaba, Xin-She Yang, and Javier Del Ser. Traveling salesman problem: a perspective review of recent research and new results with bio-inspired metaheuristics. pages 135–164, 05 2020. doi: 10.1016/B978-0-12-819714-1.00020-8. URL



[https://www.researchgate.net/publication/341123222\\_Traveling\\_salesman\\_problem\\_a\\_perspective\\_review\\_of\\_recent\\_research\\_and\\_new\\_results\\_with\\_bio-inspired\\_metaheuristics.](https://www.researchgate.net/publication/341123222_Traveling_salesman_problem_a_perspective_review_of_recent_research_and_new_results_with_bio-inspired_metaheuristics)

- [9] Zhongqiang Ma, Guohua Wu, Ponnuthurai Nagaratnam Suganthan, Aijuan Song, and Qizhang Luo. Performance assessment and exhaustive listing of 500+ nature-inspired metaheuristic algorithms. *Swarm and Evolutionary Computation*, 77, 2023. ISSN 2210-6502. doi: <https://doi.org/10.1016/j.swevo.2023.101248>. URL <https://www.sciencedirect.com/science/article/pii/S2210650223000226>.
- [10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2023.
- [11] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018. URL [https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language-understanding\\_paper.pdf](https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language-understanding_paper.pdf).
- [12] Repositorio código gpt-1, . URL <https://github.com/openai/fine-tune-transformer-lm>.
- [13] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, Zihao Wu, Lin Zhao, Dajiang Zhu, Xiang Li, Ning Qiang, Dingang Shen, Tianming Liu, and Bao Ge. Summary of chatgpt-related research and perspective towards the future of large language models. *Meta-Radiology*, 1(2), 2023. ISSN 2950-1628. doi: <https://doi.org/10.1016/j.metrad.2023.100017>. URL <https://www.sciencedirect.com/science/article/pii/S2950162823000176>.
- [14] Gonzalo Martínez, Javier Conde, Pedro Reviriego, Elena Merino Gómez, José Hernández, and Fabrizio Lombardi. How many words does chatgpt know? the answer is chatwords. 09 2023. doi: 10.48550/arXiv.2309.16777. URL [https://www.researchgate.net/publication/374544616\\_How\\_Many\\_Words\\_Does\\_Chatgpt\\_Know\\_The\\_Answer\\_Is\\_Chatwords](https://www.researchgate.net/publication/374544616_How_Many_Words_Does_Chatgpt_Know_The_Answer_Is_Chatwords).
- [15] Eric Sarrion. *Using ChatGPT for Code Generation in Computer Programs*. 2023. ISBN 978-1-4842-9529-8. doi: 10.1007/978-1-4842-9529-8\_15. URL [https://doi.org/10.1007/978-1-4842-9529-8\\_15](https://doi.org/10.1007/978-1-4842-9529-8_15).
- [16] Chao Liu, Bao Xuanlin, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. Improving chatgpt prompt for code generation. 05 2023. URL <https://arxiv.org/pdf/2305.08360.pdf>.



- [17] Prompting chatgpt to build a tabu search algorithm. URL <https://jmsallan.netlify.app/blog/2023-02-03-prompting-chatgpt-to-build-a-tabu-search-algorithm/>.
- [18] Hojatollah Rajabi Moshtaghi, Abbas Toloie Eshlaghy, and Mohammad Reza Motadel. A comprehensive review on meta-heuristic algorithms and their classification with novel approach. *Journal of Applied Research on Industrial Engineering*, 8(1):63–89, 2021. ISSN 2538-5100. doi: 10.22105/jarie.2021.238926.1180. URL [https://www.journal-aprie.com/article\\_126005.html](https://www.journal-aprie.com/article_126005.html).
- [19] Microsoft copilot. URL <https://copilot.microsoft.com/>.
- [20] Google bard. URL <https://bard.google.com/>.
- [21] Chatgpt. URL <https://openai.com/research/gpt-4>.
- [22] Python. URL <https://docs.python.org/3.12/>.
- [23] Pycharm. URL <https://www.jetbrains.com/help/pycharm/2023.1/getting-started.html>.
- [24] Numpy. URL <https://numpy.org/doc/1.26/>.
- [25] Pandas. URL [https://pandas.pydata.org/docs/user\\_guide/index.html](https://pandas.pydata.org/docs/user_guide/index.html).
- [26] Documentación scipy, 2024. URL <https://docs.scipy.org/doc/scipy/>.
- [27] Matplotlib. URL <https://matplotlib.org/stable/users/index>.
- [28] Orange3. URL <https://orange3.readthedocs.io/projects/orange-data-mining-library/en/latest/index.html>.
- [29] Chatgpt-3.5. URL <https://chat.openai.com/>.
- [30] Overleaf. URL <https://www.overleaf.com/learn>.
- [31] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. doi: 10.1109/4235.585893.
- [32] Amanur Rahman Saiyed. The traveling salesman problem. 2012. URL <http://cs.indstate.edu/~zeeshan/aman.pdf>.



- [33] JianChen Zhang. Comparison of various algorithms based on tsp solving. *Journal of Physics: Conference Series*, 2083(3), nov 2021. doi: 10.1088/1742-6596/2083/3/032007. URL <https://dx.doi.org/10.1088/1742-6596/2083/3/032007>.
- [34] Sriyani Violina. Analysis of brute force and branch & bound algorithms to solve the traveling salesperson problem (tsp). *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, 12:1226–1229, 2021. URL <https://turcomat.org/index.php/turkbilmat/article/view/3031>.
- [35] Szymon Jagielo Radoslaw Grymin. *Fast Branch and Bound Algorithm for the Travelling Salesman Problem*. 2016. doi: 10.1007/978-3-319-45378-1\_19. URL <https://inria.hal.science/hal-01637523/document>.
- [36] Bisan Alsalibi, Marzieh Babaeianjelodar, and Ibrahim Venkat. A comparative study between the nearest neighbor and genetic algorithms: A revisit to the traveling salesman problem. *International Journal of Computer Science and Electronics Engineering*, 1:34–38, 12 2012.
- [37] Anna Karlin, Nathan Klein, and Shayan Oveis Gharan. An improved approximation algorithm for tsp in the half integral case. 2019.
- [38] Anna Adamaszek, Matthias Mnich, and Katarzyna Paluch. *New Approximation Algorithms for (1,2)-TSP*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*. 2018. ISBN 978-3-95977-076-7. doi: 10.4230/LIPIcs.ICALP.2018.9. URL <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2018.9>.
- [39] *The Traveling Salesman Problem*. 2008. ISBN 978-3-540-71844-4. doi: 10.1007/978-3-540-71844-4\_21. URL [https://doi.org/10.1007/978-3-540-71844-4\\_21](https://doi.org/10.1007/978-3-540-71844-4_21).
- [40] Carlos García Martínez. Comparador original de metaheurísticas. URL [https://github.com/cgarcia-UCO/TSP\\_MH\\_Comparator](https://github.com/cgarcia-UCO/TSP_MH_Comparator).
- [41] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7(1):1–30, 2006. URL <https://www.jmlr.org/papers/volume7/demsar06a/demsar06a.pdf>.
- [42] D. Pereira, Anabela Afonso, and Fátima Medeiros. Overview of friedman's test and post-hoc analysis. *Communications in Statistics - Simulation and Computation*, 44:2636–2653, 2015. doi: 10.1080/03610918.2014.931971.



- [43] AYRNA. Test de nemenyi. URL <http://www.uco.es/ayrna/QRBF/node5.html#Friedman>.
- [44] Zelda B. Zabinsky. *Random Search Algorithms*. John Wiley Sons, Ltd, 2011. ISBN 9780470400531. doi: <https://doi.org/10.1002/9780470400531.eorms0704>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470400531.eorms0704>.
- [45] Pierre Hansen and Nenad Mladenović. First vs. best improvement: An empirical study. *Discrete Applied Mathematics*, 154(5):802–817, 2006. doi: <https://doi.org/10.1016/j.dam.2005.05.020>. URL <https://www.sciencedirect.com/science/article/pii/S0166218X05003070>.
- [46] Paulo R d O da Costa, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay. *Learning 2-opt Heuristics for the Traveling Salesman Problem via Deep Reinforcement Learning*, pages 465–480. 2020. URL <https://proceedings.mlr.press/v129/costa20a.html>.
- [47] Duncan Adamson, Nathan Flaherty, Igor Potapov, and Paul G. Spirakis. On the structural and combinatorial properties in 2-swap word permutation graphs. 2023.
- [48] G. Gutin and D. Karapetyan. A memetic algorithm for the generalized traveling salesman problem. *Natural Computing*, 9:47–60, 2010. doi: 10.1007/s11047-009-9111-6.
- [49] H. F. Amaral, S. Urrutia, and L. M. Hvattum. Delayed improvement local search. *Journal of Heuristics*, 27:923–950, 2021. doi: 10.1007/s10732-021-09479-9.
- [50] Fred Glover, Manuel Laguna, and Rafael Marti. Tabu search. *Tabu Search*, 16, 2008. doi: 10.1007/978-1-4615-6089-0.
- [51] Vishnu Kumar Prajapati, Mayank Jain, and Lokesh Chouhan. *Tabu Search Algorithm (TSA): A Comprehensive Survey*. 2020. doi: 10.1109/ICETCE48199.2020.9091743.
- [52] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. doi: 10.1126/science.220.4598.671. URL [http://wexler.free.fr/library/files/kirkpatrick%20\(1983\)%20optimization%20by%20simulated%20annealing.pdf](http://wexler.free.fr/library/files/kirkpatrick%20(1983)%20optimization%20by%20simulated%20annealing.pdf).

BIBLIOGRAFÍA

---

- [53] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics—Part B*, 26:29–41, 1996. doi: 10.1109/3477.484436.
- [54] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1:28–39, 12 2006. doi: 10.1109/MCI.2006.329691.
- [55] Dervis Karaboga. An idea based on honey bee swarm for numerical optimization, technical report - tr06. *Technical Report, Erciyes University*, 01 2005.
- [56] J. Kennedy and R. Eberhart. *Particle swarm optimization*. 1995. doi: 10.1109/ICNN.1995.488968.
- [57] Esmat Rashedi, Hossein Nezamabadi-pour, and Saeid Saryazdi. Gsa: A gravitational search algorithm. *Information Sciences*, 179(13):2232–2248, 2009. ISSN 0020-0255. doi: <https://doi.org/10.1016/j.ins.2009.03.004>. URL <https://www.sciencedirect.com/science/article/pii/S0020025509001200>.
- [58] Santosh Kumar, Deepanwita Datta, and Sanjay Singh. Black hole algorithm and its applications. *Studies in Computational Intelligence*, 575:147–170, 12 2015. doi: 10.1007/978-3-319-11017-2\_7.
- [59] Silvia Casado and Rafael Martí. Principios de la búsqueda dispersa. URL <https://www.uv.es/~rmarti/paper/docs/ss11.pdf>.
- [60] Robert G. Reynolds. An introduction to cultural algorithms. pages 131–139, 1994. URL [https://www.researchgate.net/publication/201976967\\_An\\_Introduction\\_to\\_Cultural\\_Algorithms](https://www.researchgate.net/publication/201976967_An_Introduction_to_Cultural_Algorithms).
- [61] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers Operations Research*, 24(11):1097–1100, 1997. doi: [https://doi.org/10.1016/S0305-0548\(97\)00031-2](https://doi.org/10.1016/S0305-0548(97)00031-2). URL <https://www.sciencedirect.com/science/article/pii/S0305054897000312>.
- [62] Zong Woo Geem, Joong Hoon Kim, and G.V. Loganathan. A new heuristic optimization algorithm: Harmony search. *SIMULATION*, 76(2):60–68, 2001. URL <https://doi.org/10.1177/003754970107600201>.
- [63] Esmaeil Atashpaz-Gargari and Caro Lucas. Imperialist competitive algorithm: An algorithm for optimization inspired by imperialistic competition. *2007 IEEE Congress on Evolutionary Computation, CEC 2007*, 7:4661 – 4667, 10 2007. doi: 10.1109/CEC.2007.4425083.



- [64] Juan Velásquez. Una introducción a los algoritmos basados en caos para optimización numérica. *Avances en Sistemas e Informática*, 8:51–60, 03 2011.
- [65] Jon Timmis, T Knight, Leandro De Castro, and Emma Hart. An overview of artificial immune systems. *An Overview of Artificial Immune Systems*, 2004. doi: 10.1007/978-3-662-06369-9\_4.
- [66] J. Timmis, P. Andrews, N. Owens, et al. An interdisciplinary perspective on artificial immune systems. *Evolutionary Intelligence*, 1:5–26, 2008. doi: 10.1007/s12065-007-0004-2.
- [67] Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio. *Using River Formation Dynamics to Design Heuristic Algorithms*. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73554-0.
- [68] Xin-She Yang. *Firefly Algorithms for Multimodal Optimization*. Berlin, Heidelberg, 2009. ISBN 978-3-642-04944-6.
- [69] John H. Holland. *Genetic Algorithms and Adaptation*, pages 317–333. Springer US, Boston, MA, 1984. doi: 10.1007/978-1-4684-8941-5\_21. URL [https://doi.org/10.1007/978-1-4684-8941-5\\_21](https://doi.org/10.1007/978-1-4684-8941-5_21).
- [70] Christos Voudouris, Edward P.K. Tsang, and Abdullah Alshedy. *Guided Local Search*, pages 321–361. 2010. doi: 10.1007/978-1-4419-1665-5\_11. URL [https://doi.org/10.1007/978-1-4419-1665-5\\_11](https://doi.org/10.1007/978-1-4419-1665-5_11).
- [71] Alireza Askarzadeh. A novel metaheuristic method for solving constrained engineering optimization problems: Crow search algorithm. *Computers & Structures*, 169:1–12, 2016. doi: <https://doi.org/10.1016/j.compstruc.2016.03.001>. URL <https://www.sciencedirect.com/science/article/pii/S0045794916300475>.
- [72] A.R. Mehrabian and C. Lucas. A novel numerical optimization algorithm inspired from weed colonization. *Ecological Informatics*, 1(4):355–366, 2006. ISSN 1574-9541. doi: <https://doi.org/10.1016/j.ecoinf.2006.07.003>. URL <https://www.sciencedirect.com/science/article/pii/S1574954106000665>.
- [73] Xin-She Yang. *A New Metaheuristic Bat-Inspired Algorithm*, pages 65–74. 2010. doi: 10.1007/978-3-642-12538-6\_6. URL [https://doi.org/10.1007/978-3-642-12538-6\\_6](https://doi.org/10.1007/978-3-642-12538-6_6).
- [74] R. Storn and K. Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359, 1997. doi: 10.1023/A:1008202821328.

BIBLIOGRAFÍA

---

- [75] Konstantinos I. Roumeliotis and Nikolaos D. Tselikas. Chatgpt and open-ai models: A preliminary review. *Future Internet*, 15(6), 2023. ISSN 1999-5903. doi: 10.3390/fi15060192. URL <https://www.mdpi.com/1999-5903/15/6/192>.
- [76] Valentín G. Avram Aenachioei. Datos experimentales, 2024. URL <https://github.com/ValentinAvram/Trabajo-Fin-Grado/tree/main/datos>.
- [77] Valentín G. Avram Aenachioei. Proyecto completo, 2024. URL <https://github.com/ValentinAvram/Trabajo-Fin-Grado>.
- [78] Valentín G. Avram Aenachioei. Ejecutable del proyecto, 2024. URL <https://github.com/ValentinAvram/Trabajo-Fin-Grado/tree/main/ejecutable>.
- [79] Valentín G. Avram Aenachioei. Código fuente del proyecto, 2024. URL <https://github.com/ValentinAvram/Trabajo-Fin-Grado/tree/main/proyecto>.