Testing de sistemas restaurables empleando metaheurísticas

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA CURSO 2017-2018

Autores

Andrés Pascual Contreras

José Manuel Pérez Zamorano

Adrián Muñoz Gámez

Director

Pablo M. Rabanal Basalo

FACULTAD DE INFORMÁTICA UNIVERSIDAD COMPLUTENSE DE MADRID

ÍNDICE

1. RESUMEN

1.1. Español

Este trabajo consiste en la elaboración de una aplicación que permita realizar pruebas sobre máquinas de estados utilizando técnicas de inteligencia artificial, concretamente heurísticas y metaheurísticas, optimizando los costes necesarios para llevarlas a cabo.

En el entorno de la computación, cualquier programa de software, lógica electrónica o aplicación informática puede representarse como una máquina de estados que codifique todas las posibles combinaciones de uso que pueda realizar y todos los pasos que atraviesa en su proceso.

En sistemas pequeños resulta sencillo comprobar manualmente el correcto funcionamiento de estos, sin embargo, cuando manejamos sistemas de cierta envergadura la complejidad aumenta exponencialmente.

En estos casos es imprescindible el uso de mecanismos que automaticen las pruebas, pero en ocasiones resultan ineficientes debido a las múltiples posibilidades que se presentan.

Por ello, es necesario desarrollar procedimientos que sean capaces de abordar las pruebas de dichos sistemas en unas condiciones de tiempo aceptables.

El objetivo de este trabajo es precisamente tratar de resolver esta problemática utilizando varios elementos de computación que permitan reducir al máximo los costes y elaborar un estudio que muestre qué métodos se han mostrado más efectivos.

Estos métodos consisten en tres metaheurísticas ya definidas, que son Ant Colony Optimization, Genetic Algorithm y River Formation Dynamics, sobre las que se realizará un desarrollo para adaptarlas al problema que se trata de resolver. Tras haber implementado estas metaheurísticas, se realizará una investigación exhaustiva para examinar su comportamiento y descubrir la mejor solución entre las propuestas.

1.2. Inglés

This work consists in the development of an application that allows to perform tests on state machines using artificial intelligence techniques, specifically heuristics and meta-heuristics, optimizing the costs necessary to carry them out.

In the computing environment, any software program, electronic logic or computer application can be represented as a state machine that encodes all the possible combinations of use that can be made, and all the steps that it goes through in its process.

In small systems it is easy to manually check the correct operation of them, however, when we manage systems of a certain size, the complexity increases exponentially.

In these cases, it is essential to use mechanisms to automate the tests, but sometimes they are inefficient due to the multiple possibilities that arise.

Therefore, it is necessary to develop procedures that are capable of addressing the testing of such systems in acceptable time conditions.

The goal of our work is to try solving this problem using several computing elements that allow to reduce the time costs as much as possible and develop a study that shows which methods have been most effective.

These methods consist of three metaheuristics already defined, on which a development will be carried out to adapt them to the problem to be solved. After implementing these metaheuristics, a thorough research will be conducted to examine their behavior and discover the best solution among the proposals.

1.3. Palabras clave

Español

- ***Pruebas de software***
- ***Optimización***
- ***Metaheurísticas***
- ***Optimización por colonia de hormigas (ACO)***
- ***Formación dinámica de los ríos (RFD)***
- ***Algoritmo Genético (GA)***
- ***Algoritmos evolutivos***
- ***Mínima Secuencia de Carga (MLS)***
- ***Máquinas de estados***

Inglés

- ***Testing***
- ***Optimization***
- ***Metaheuristics***
- ***Ant Colony Optimization (ACO)***
- ***River Formation Dynamics (RFD)***
- ***Genetic Algorithm (GA)***
- ***Evolutionary Algorithms***
- ***Minimum Load Sequence (MLS)***
- ***State Machines***

2. INTRODUCCIÓN

2.1. Objetivo

El objetivo de este trabajo es solucionar el problema de la Mínima Secuencia de Carga [1] (Minimum Load Secuence, en adelante MLS) que es un problema de *testing* de sistemas restaurables (trabaja con máquinas de estados finitos FSM), que se describirá con detalle en la sección de *Descripción del Problema* de la presente memoria.

Para llevar a cabo dicho objetivo, la labor efectuada se divide en dos partes. Por un lado, la creación de una aplicación que implemente tres metaheurísticas para resolver el problema: Formación Dinámica de Ríos [10] (River Formation Dynamics, en adelante RFD), el algoritmo de Colonia de Hormigas [14] (Ant Colony Optimization, en adelante ACO) y el algoritmo Genético [18] (en adelante GA). Por otro lado, el proceso de estudio e investigación del mejor método para hallar la solución al problema.

La elección del algoritmo RFD está motivada porque forma parte del estudio fundamental del artículo desde el que parte este trabajo [1]. Respecto a ACO, ha sido seleccionado porque supone un planteamiento similar a RFD y se ha mostrado efectivo en la resolución de problemas que implican recorridos sobre grafos. Por último, la inclusión de GA en el trabajo es debida a su popularidad y versatilidad a la hora de abordar problemas similares.

Respecto a la primera tarea, se desarrollará una herramienta que genere las FSM de entrada y que sea capaz de implementar los algoritmos citados anteriormente.

Es importante señalar que, aunque el desarrollo de la aplicación es una parte con una carga de trabajo y una complejidad considerables, debe ser entendida como un medio para un fin. Es decir, se trata de conseguir una herramienta a través de la cual podamos llevar a cabo el objetivo principal, que es, como se ha señalado anteriormente, el ajuste y la comparativa entre los diferentes métodos de resolución del problema propuestos.

Nuestro objetivo fundamental es, por tanto, encontrar un método competitivo que sea capaz de ofrecer soluciones al problema planteado en un tiempo aceptable, de hecho, en el tiempo más óptimo posible. Debido a la naturaleza del problema no es trivial encontrar una respuesta válida a esta cuestión y es necesario realizar un profundo proceso de investigación para tratar de hallar una solución lo más aproximada posible.

Este proceso consiste en someter los tres métodos que vamos a utilizar a pruebas intensivas, variando sus distintas configuraciones hasta encontrar el mejor conjunto de parámetros, que nos permita conseguir la solución más competitiva en cada uno de los tres y después compararlos entre sí y ver cuál es el mejor método general para resolver el problema.

Por tanto, el proceso de investigación consistirá en elaborar un estudio exhaustivo cuyo cometido es abordar los factores más relevantes de las posibles formas de resolución del problema planteadas, tales como las distintas evoluciones en los resultados obtenidos variando las configuraciones, las diferencias encontradas respecto a la utilización del recurso de carga y la repercusión de su coste, o el comportamiento de los distintos métodos frente a sistemas de diferente envergadura. Todo ello con el propósito de satisfacer nuestro objetivo fundamental que no es otro que el de intentar dar una respuesta a cuál es la mejor forma posible de resolver este problema de las tres propuestas.

2.2. Antecedentes

Al hablar de antecedentes al problema, en primer lugar se tiene que nombrar el artículo de P. Rabanal, I. Rodríguez y F. Rubio [1] que es la base de este proyecto y es de donde se ha extraído la definición formal del problema. A parte de este artículo en el que se resuelve el mismo problema tratado en este trabajo, el problema objeto de estudio pertenece al campo de los problemas de optimización de *testing*.

En el campo del *testing* de software se aplican algoritmos basados en búsqueda en espacio de estados y metaheurísticas. En un artículo de R. Lefticaru y F. Ipate [2] se utilizan técnicas de búsqueda en espacio de estados aplicadas al campo del *testing* funcional. En [20] P. McMinn resuelve un problema de structural testing utilizando las técnicas de *hill-climbing* y una adaptación del algoritmo genético. A. Sharma, R. Patani y A. Aggarwal en [9] señalan aplicaciones del algoritmo genético en el campo del *testing* y en el de la resolución de problemas complejos (NP).

Otro de los campos donde se pueden encontrar multitud de aplicaciones de técnicas metaheurísticas es el campo del *clustering* de datos. En [5] S. Kashef y H. Nezamabadi-pour resuelven el problema de selección de variable utilizando distintos métodos basados en ACO y luego comparan los resultados con otras heurísticas como el Binary Genetic Algorithm (BGA) o el Binary Particle Swarm Optimization (BPSO), este artículo compara ACO con una gran cantidad de algoritmos evolutivos y otras técnicas de búsqueda metaheurística. Se han encontrado otros dos artículos [6-7] en los que se adapta ACO para problemas de *clustering* de datos y comparan los resultados con otras técnicas de *clustering*. También pueden encontrarse aplicaciones de ACO en redes de telecomunicaciones y *routing*, tal como puede verse en el artículo de Paul Sharkey [8].

El algoritmo genético se emplea en multitud de campos como pueden ser: resolución de caminos, *machine learning* y *testing*. J. Carlos Rincón aplicó GA para encontrar la forma más óptima de diseñar un sistema de abastecimiento de agua tomando en cuenta, principalmente, elementos económicos y costes de fabricación y distribución [13].

El algoritmo RFD, para ser un método joven, ha sido aplicado en numerosos campos como las redes de telecomunicación, problemas de *testing*, diseño de infraestructuras de comunicación y sistemas de navegación para robots [10]. Cabe destacar que el año pasado G. Redlarski, M. Dabkowski y A. Palkowski resolvieron un problema que consiste en encontrar el mejor camino para llegar de un punto a otro en un medio estático o dinámico con obstáculos [3] y además hicieron un estudio comparando RFD y ACO entre otros algoritmos. En [4] se combina el algoritmo RFD con uno de *clustering* con el objetivo de mejorar la eficiencia energética en redes WSN.

2.3. Plan de trabajo

Este proyecto ha constado de varias partes: estudio y comprensión de MLS [1], definición de la estructura de la aplicación a desarrollar, desarrollo de la aplicación, adaptación de las metaheurísticas a MLS, realización del estudio y redacción de la memoria.

Se comenzó por el estudio y comprensión de MLS, actividad que se llevó a cabo de forma independiente, con reuniones intermedias para comentar puntos entre los miembros del grupo. El plan inicial era dar por finalizado esta actividad para la mitad de octubre de 2017, y acabó alargándose hasta el final de ese mismo mes.

Tras haber comprendido el problema, se comenzó el desarrollo de la aplicación, que conllevaba definir la estructura de esta, y la adaptación de las metaheurísticas propuestas al problema tratado (MLS). El desarrollo comenzó en noviembre de 2017, y el plan inicial era terminarlo para mediados/finales de abril de 2018. Finalmente, el desarrollo tuvo distintas complicaciones, pues la adaptación de las metaheurísticas no fue sencilla y acabó terminando en agosto de 2018. Para la realización de esta tarea, en los meses de noviembre de 2017 a febrero de 2017 se trabajó conjuntamente mediante herramientas online, como puede ser Skype y Google Drive. En esos meses la estructura de la aplicación quedó definida y se comenzó el desarrollo de RFD, hasta el punto en que RFD no se conseguía adaptar correctamente y se encontró una primera barrera. Tras el acontecimiento anterior, se repartió el trabajo de desarrollo para avanzar con el resto de metaheurísticas y la parte visual, hasta el final del desarrollo en agosto de 2018.

La redacción de la memoria tenía planificado su inicio con el final del desarrollo de la aplicación, en mediados/finales de abril de 2018, pero al retrasarse el desarrollo, se retrasó el comienzo de la elaboración de la memoria hasta junio de 2018. En junio de 2018 se comenzó la redacción de la memoria mientras se seguía con el desarrollo de la aplicación.

El estudio y análisis de resultados, fue realizado desde mediados de agosto hasta septiembre de 2018, además de todas las conclusiones, ideas y pruebas previas extraídas de la adaptación de las metaheurísticas a lo largo de los meses del desarrollo.

2.4. Goal

The objective of this work is to solve the problem of the Minimum Load Sequence [1] (Minimum Load Sequence, hereinafter MLS) which is a problem of testing of restorable systems (works with finite-state machines FSM), which will be described detail in the section Description of the Problem.

To carry out this objective, the work carried out is divided into two parts. On the one hand, the creation of an application that implements three metaheuristics to solve the problem: Dynamic River Formation [10] (River Formation Dynamics, hereinafter RFD), the algorithm of Ant Colony Optimization [14] (in forward ACO) and the Genetic Algorithm [18] (hereinafter GA). On the other hand, the process of study and investigation of the best method to find the solution to the problem.

The choice of the RFD algorithm is motivated because it is part of the fundamental study of the article from which this work is based [1]. Regarding ACO, it has been selected because it assumes a similar approach to RFD and has proven effective in solving problems involving paths over graphs. Finally, the inclusion of GA in the work is due to its popularity and versatility when it comes to addressing similar problems.

Regarding the first task, a tool will be developed to generate the input FSMs and be able to implement the algorithms mentioned above.

It is important to note that, although the development of the application is a part with a considerable workload and complexity, it must be understood as a means to an end. That is to say, it is a question of obtaining a tool through which we can carry out the main objective, which is, as it has been indicated previously, the adjustment and the comparison between the different methods of resolution of the proposed problem.

Our fundamental objective is to find a competitive method that is capable of offering solutions to the problem posed in an acceptable time, in fact, in the most optimal time possible. Due to the nature of the problem, it is not trivial to find a valid answer to this question and it is necessary to conduct a thorough research process to try to find a solution as close as possible.

This process consists of submitting the three methods that we will use to intensive tests, varying their different configurations until finding the best set of parameters, which allows us to get the most competitive solution in each of the three and then compare them to each other and see which It is the best general method to solve the problem.

Therefore, the research process will consist of an exhaustive study whose purpose is to address the most relevant factors of the possible ways of solving the problem, such as the different evolutions in the results obtained by varying the configurations, the differences found with respect to the utilization of the resource of load and the repercussion of its cost, or the behavior of the different methods in front of systems of different magnitude. All with the purpose of satisfying our fundamental objective that

is none other than trying to give an answer to which is the best possible way to solve this problem of the three proposals.

2.5. Background

When discussing the background to the problem, first of all we must name the article by P. Rabanal, I. Rodríguez and F. Rubio [1] which is the basis of this project and from which the formal definition of the problem has been extracted. Apart from this article in which the same problem dealt with in this work is solved, the problem object of study belongs to the field of testing optimization problems.

In the field of software testing algorithms based on search in statespace and metaheuristics are applied. In an article by R. Lefticaru and F. Ipate [2] search techniques are used in the space of states applied to the field of functional testing. In [20], P. McMinn solves a problem of structural testing using hill-climbing techniques and an adaptation of the genetic algorithm. A. Sharma, R. Patani and A. Aggarwal in [9] point out applications of the genetic algorithm in the field of testing and complex problem solving (NP).

Another field where you can find many applications of metaheuristic techniques is the field of data clustering. In [5] S. Kashef and H. Nezamabadi-pour solve the problem of variable selection using different methods based on ACO and then compare the results with other heuristics such as the Binary Genetic Algorithm (BGA) or the Binary Particle Swarm Optimization (BPSO), this article compares ACO with a large number of evolutionary algorithms and other metaheuristic search techniques. Two other articles have been found [6-7] in which ACO is adapted for data clustering problems and compare the results with other clustering techniques. ACO applications can also be found in telecommunication and routing networks, as can be seen in Paul Sharkey's article [8].

The genetic algorithm is used in many fields such as: road solving, Machine Learning and testing. J. Carlos Rincon applied GA to find the most optimal way to design a water supply system taking into account, mainly, economic elements and manufacturing and distribution costs [13].

The RFD algorithm, to be a young method, has been applied in numerous fields such as telecommunication networks, testing problems, design of communication infrastructures and navigation systems for robots [10]. It should be noted that last year G. Redlarski, M. Dabkowski and A. Palkowski solved a problem that consists of finding the best way to get from one point to another in a static or dynamic environment with obstacles [3] and they also made a study also comparing RFD and ACO in addition to other algorithms. In [4], the RFD algorithm is combined with one of clustering with the aim of improving energy efficiency in WSN networks.

2.6. Workplan

This project has consisted of several parts: study and understanding of MLS [1], definition of the structure of the application to develop, application development, adaptation of metaheuristics to MLS, realization of the study and writing of the report.

It began with the study and understanding of MLS, an activity that was carried out independently, with intermediate meetings to discuss points among the members of the group. The initial plan was to end this activity for the middle of October 2017 and ended up extending until the end of that month.

After having understood the problem, the development of the application began, which entailed defining the structure of the application, and the adaptation of the proposed metaheuristics to the problem addressed (MLS). The development started in November 2017, and the initial plan was to finish it by mid / late April 2018. Finally, the development had different complications, since the adaptation of the metaheuristics was not easy and ended in August 2018. For the completion of this task, in the months of November 2017 to February 2017 we worked together using online tools, such as Skype and Google Drive. In those months the structure of the application was defined, and the development of RFD was started, to the point where RFD could not be adapted correctly and a first barrier was found. After the previous event, the development work was shared to advance with the rest of the metaheuristics and the visual part, until the end of the development in August 2018.

The writing of the report had planned its beginning with the end of the development of the application, in the middle / end of April 2018, but when the development was delayed, the beginning of the preparation of the report until June 2018 was delayed. of 2018 the writing of the memory began while it continued with the development of the application.

The study and analysis of results was carried out from mid-August to September 2018, in addition to all the conclusions, ideas and previous tests extracted from the adaptation of metaheuristics throughout the months of development.

3. DESCRIPCIÓN DEL PROBLEMA

3.1. Introducción

Dada una máquina de estados (la cual puede reestablecer estados previos) y una lista de estados por los que se requiere pasar, el problema que se trata en el presente trabajo consiste en encontrar la secuencia más óptima en coste y tiempo de recorrer dichos estados.

Cualquier sistema, como por ejemplo una aplicación informática o una máquina de refrescos, puede representarse como una máquina de estados que codifique todos los posibles estados y todas las transiciones factibles entre los mismos. Abstrayendo el problema y puesto que, a su vez, cualquier máquina de estados puede codificarse como un grafo, lo que se trata de conseguir es recorrer determinados nodos del grafo de la forma más eficiente posible en términos de tiempo y coste.

La particularidad fundamental del problema es la introducción de la posibilidad de restauración de estados anteriores, es decir, que para llegar hasta determinado estado se puede realizar volviendo a recorrer el camino necesario para llegar hasta él o cargar un estado anterior, por el que ya se pasó para ir a dicho estado. Esto implica que las soluciones que se pretenden obtener son representadas por caminos no lineales, en los cuales no tienen por qué recorrerse todos los estados y transiciones determinadas de principio a fin, sino que pueden ramificarse para cubrir diferentes secciones cuyas bifurcaciones implican un proceso de carga. Esta particularidad no es otra que el problema de la mínima secuencia de carga (Minimum Load Sequence, en adelante MLS) [1].

El objetivo es encontrar un método que permita realizar esta tarea para cualquier grafo, incorporando algunos requisitos que emulen el proceso de prueba del sistema tales como la posibilidad de cargar un estado anterior o la de elegir no solo que nodos del grafo se quiere pasar sino también caminos concretos por los que se deba pasar obligatoriamente. En conjunto, la funcionalidad exigida para hallar la solución aglutina todas las opciones mínimas deseables en un proceso de pruebas suficientemente completo.

Se trata, por tanto, de un problema que se asemeja al conocido Problema del viajero, y cuya complejidad se enmarca, al igual que este, en la de los problemas de clase NP. Dicha complejidad obliga a buscar soluciones en un proceso de investigación empírica, dado que el coste exponencial de este tipo de problema imposibilita (al menos hasta que se demuestre lo contrario) encontrar una solución determinista.

Para ello, se proponen tres formas para hallar la mejor solución posible: el método RFD, el método ACO y el método GA

Lo que se expone en la presente memoria es precisamente el proceso de investigación y experimentación llevado a cabo para examinar los diferentes métodos propuestos,

adaptarlos a resolver el problema y determinar cuál de ellos, y en qué circunstancias concretas, se muestra más eficiente para encontrar una solución válida con el menor coste posible.

3.2. Complejidad del problema: Problemas P y NP

Dado que el problema que se trata resolver es un problema de naturaleza $NP\ Completo$[1], es conveniente comenzar con una pequeña introducción acerca de las clases de complejidad de los problemas, para entender de forma precisa lo que supone la resolución del mismo respecto a dificultad y coste.

En términos de complejidad, cuando el tiempo de ejecución de un algoritmo que resuelve un problema, se puede calcular a partir del número de variables implicadas a través de una fórmula polinómica, entonces se dice que el problema es de clase P. Sin embargo, cuando el tiempo de ejecución no puede calcularse de esta manera, el problema es denominado NP.

Por ejemplo, si tenemos un algoritmo cuya complejidad es $3n^2$, sabemos que el coste de tiempo dependerá de n, y puede calcularse de forma polinómica. Por otro lado, si estamos ante un problema cuyo algoritmo de resolución sea -por ejemplo- 3^n, ya no se trata de un polinomio y se trataría de un problema de clase NP.

Actualmente no se ha podido demostrar si $P = NP$, y respecto a ello, P se considera un subconjunto de NP. Esto significa que ante la imposibilidad actual de demostrar la igualdad anterior, los problemas NP pueden ser P o no. Por lo que, dicho de una forma simple, los problemas NP son aquellos que no se ha podido demostrar que son P. Es decir, los problemas para los que no se ha encontrado una forma polinómica para calcular su coste de tiempo.

Esto es importante porque el problema que se aborda pertenece a la clase $NP\ Completo$ y por ello debe ser tratado de forma diferente a los problemas de tipo P, debido a que los costes exponenciales situarían su resolución en el terreno de los intratables, impidiendo hallar una solución en un tiempo mínimamente aceptable.

Por este motivo, se proponen las soluciones alternativas sobre las que versa el presente trabajo, para intentar dar una solución aceptable y lo más competente posible en términos de coste.

3.3. Definición del problema

Realizar pruebas que detecten fallos en el desarrollo y garanticen el correcto funcionamiento de un sistema es una necesidad fundamental. Existen técnicas para realizar baterías de pruebas de forma sistemática, las cuales facilitan la labor del probador y permiten obtener unos resultados exhaustivos sobre la funcionalidad del sistema, permitiendo tener la seguridad de que todo procede como debería. Sin embargo, en muchos casos realizar unas pruebas que evalúen de forma precisa y completa la funcionalidad de una implementación implica probar un número casi infinito de comportamientos disponibles, lo cual resulta inviable.

Se ha observado que asumir ciertas hipótesis sobre el sistema a probar, hace posible la existencia de conjuntos finitos de pruebas. Sin embargo, en sistemas grandes y/o de cierta complejidad, aún incluso asumiendo estás hipótesis, el conjunto de pruebas, aunque sea finito, tiene unas dimensiones inmensas que siguen haciéndolo inasumible.

Por este motivo, los evaluadores suelen renunciar a buscar la completitud y en su lugar deciden realizar una serie de pruebas que, siguiendo unos criterios de prueba de cobertura, permiten centrarse en algunas partes concretas de la implementación y examinar algunos comportamientos específicos de la misma.

Como se dijo anteriormente, cualquier implementación de un sistema puede representarse como una máquina finita de estados (Finite State Machine, FSM a partir de ahora). Dada la especificación de un sistema representada por una FSM, probar todos los estados y transiciones posibles es una tarea de una complejidad exponencial al número de estados. Abordar un conjunto de pruebas que pretenda algo parecido es del todo imposible para cualquier sistema mínimamente complejo.

El factor clave es el concepto de máquinas de estados restaurables (la posibilidad de cargar estados anteriores), lo que implica la incorporación del problema del MLS [1] a todo lo mostrado anteriormente. La posibilidad de cargar estados concretos alcanzados previamente mediante el recorrido de estados anteriores es una herramienta que puede ser bastante efectiva a la hora de reducir el coste temporal en las pruebas de un sistema representado mediante FSM.

Por ejemplo, imaginemos que desde un estado A, que no tiene porqué ser el estado inicial, debemos probar dos estados críticos B y C, a los que solo es posible llegar a través de él. Si vamos desde A a B y después queremos ir desde A a C, debemos evaluar si es más eficiente en términos de coste repetir los pasos necesarios para llegar a A desde el estado inicial y después ir a C o si desde B cargamos el estado A y después vamos C.

Desde el punto de vista concreto de la implementación puede verse como la capacidad de un sistema para salvar su configuración, o en el caso de una aplicación informática el guardado en caché de los datos necesarios para transitar desde un estado a otro.

El peso fundamental radica, por tanto, en estudiar la cuestión de cuándo se incorpora el recurso de carga a la solución y cuándo se muestra más efectivo evitarla, lo que depende básicamente del coste asociado al mecanismo de carga y del coste de las alternativas a él. Esto se suma a la problemática ya expuesta del recorrido óptimo de un grafo y constituye de forma indisoluble el problema que se trata de resolver.

Para tratar de enfrentarse a este problema, dado que una FSM se representa como un grafo, se han utilizado varias técnicas de teoría de grafos para encontrar secuencias de prueba con el objetivo de alcanzar estados o transiciones concretas.

Un famoso ejemplo a este problema (aunque sin el elemento distintivo de la carga) es el Problema del Viajero, el cual consiste en pasar, al menos una vez, por todos los nodos de un grafo recorriéndolos en el menor tiempo posible. Siguiendo esta línea de proceder, si se tiene una FSM que represente de forma completamente equivalente la implementación que se pretende probar, podemos asegurar que los estados y las transiciones alcanzadas por la FSM sobre la que probamos serán alcanzados exactamente de la misma manera en el sistema real.

Con las técnicas adecuadas, las probabilidades de alcanzar dichos estados/transiciones y de detectar fallos en el funcionamiento son mucho mayores que probando una secuencia de entradas aleatorias, ya que estas no se definen con el propósito de alcanzar una configuración determinada del sistema, sino que atraviesan partes al azar del sistema sin permitir probar secciones concretas que sean de especial interés para los evaluadores. Esta diferencia se acentúa especialmente cuando se trata con configuraciones a las que es más enrevesado llegar como, por ejemplo, estados a los que solo se puede llegar tras un largo transcurso de transiciones, los cuales probablemente nunca serán alcanzados con baterías aleatorias.

Por todo ello, resulta evidente que encontrar métodos que permitan llevar a cabo pruebas que, si bien no son completas, posibiliten llegar a estados y transiciones concretas por los caminos más cortos y detectar la mayor cantidad de fallos en el menor tiempo posible es una necesidad crucial para probar estos sistemas.

El objetivo del presente trabajo es precisamente proponer una serie de métodos que consigan dar solución a este problema. Para lograr este cometido se realizará una labor de investigación empírica con los métodos propuestos para encontrar qué método se muestra más eficiente y que configuraciones resultan más óptimas.

4. PROPUESTA DE SOLUCIÓN

Debido a las características particulares del problema vistas anteriormente, obtener la mejor secuencia de interacción entre los estados y las transiciones indicados es un problema de clase NP, por lo que no es un objetivo realista. Por tanto, en lugar de buscar la mejor solución exacta se van a desarrollar métodos heurísticos que permitan aproximarse lo más posible a la mejor solución.

La solución a este tipo de problema no es otra que un camino que defina los nodos y las transiciones por las que se debe pasar y en qué orden para abordar todas las configuraciones que se deseen probar en el menor tiempo posible. Teniendo en cuenta la posibilidad de cargar estados previos de la que se ha hablado anteriormente, la forma de representar el camino, en términos de implementación, será una estructura de datos de tipo árbol. Esto permitirá designar el camino hasta un estado, la carga de un estado anterior, es decir la vuelta atrás a un nodo anterior a aquel desde donde se ha realizado la carga, y la continuación por otro camino distinto.

Todas las posibles soluciones que se hallarán a través de la herramienta estarán representadas por árboles, los cuales tendrán un coste asociado que será el de ser recorridos (recuérdese que las distintas transiciones entre estados tienen un coste determinado). Aquel árbol que tenga el menor coste será la mejor aproximación a la solución óptima.

La metodología de trabajo, que consiste en la aplicación de las metaheurísticas, será la misma para las tres propuestas que se han llevado a cabo, con el objetivo de que compitan en las condiciones más igualitarias posibles. Las entradas consistirán en:

- Definir una FSM (con sus estados, sus transiciones y el coste asociado a cada transición).
- Un coste de carga (en el caso que se desee utilizar esta opción).
- Un conjunto de estados y de transiciones (llamados nodos y aristas críticos) que son aquellos que por los que se debe pasar obligatoriamente en el camino-solución y que representan aquellas configuraciones que el evaluador quiere someter a prueba.

Como ya se ha visto en el apartado anterior, encontrar una solución óptima para problemas NP requiere un tiempo exponencial y no es un objetivo asumible. Por ello, los tres métodos que se presentan en este trabajo son métodos heurísticos que generalmente se utilizan para lograr conseguir soluciones aceptables para este tipo de problemas.

En muchas ocasiones la naturaleza ha inspirado algoritmos que han demostrado ser muy útiles en la resolución de problemas que se mostraban intratables en un primer momento mediante otro tipo de intentos de solución. Su fuerza radica en que permite enfrentarnos a dichos problemas desde un enfoque alternativo y obtener unas soluciones aproximadas muy competitivas en tiempo polinómico.

Este es precisamente el caso de los tres algoritmos que utilizaremos en este trabajo y que serán explicados con detalle en próximos apartados, pero que se presentarán brevemente aquí con el objetivo de introducir al lector a los mismos y mostrar de qué forma abordan la problemática particular que se enfrenta.

- RFD: es un algoritmo que se basa en la formación de caminos por parte de las gotas de agua de lluvia que caen en las montañas y van trazando caminos por la tierra, hasta llegar a su destino, el mar. En su avance, las gotas se juntan con otras que han caído en distintos lugares o se separan para seguir diferentes caminos. Los factores que marcan la "decisión" de las gotas a la hora de elegir un camino u otro se basan en las pendientes creadas por la erosión que realizan al pasar por una ruta determinada. Basándose en esta dinámica se pretende utilizar la potencia de este mecanismo para que las "gotas" recorran las posibles soluciones de la FSM potenciando aquellos caminos que resulten más eficientes hasta acabar obteniendo una solución competente.

- ACO: los algoritmos basados en colonias de hormigas (ACO) se han mostrado bastante útiles a la hora de resolver problemas basados en recorridos. Es por ello por lo que ha sido elegido para investigar cómo se comporta en este problema concreto. Se inspira en el funcionamiento de la búsqueda de comida de las hormigas y su comunicación con otras hormigas, a través de feromonas, para indicarles el mejor camino a seguir.

- GA: los algoritmos genéticos se basan en la teoría de la evolución de Darwin. Cada uno de los individuos de una población representa una posible solución al problema. De la misma forma que un ser vivo real tiene mayor o menor capacidad de supervivencia debido al grado de adaptación al medio que posea, los individuos-solución del algoritmo genético tienen mayor o menor calidad, la cual viene determinada por su capacidad de ser mejor o peor solución al problema respecto a sus congéneres. Al igual que en la naturaleza, solo los mejores individuos, los que ofrezcan, en nuestro caso, una solución en menor tiempo, sobrevivirán y serán seleccionados para reproducirse. La mecánica del algoritmo consiste en fusionar las cualidades de los mejores individuos para crear otros con el objetivo de que se acerquen aún más a la solución óptima que sus padres y repetir este ciclo múltiples veces hasta conseguir una solución lo más óptima posible.

Estos algoritmos no solo toman el sujeto del problema, en este caso la FSM, y devuelven la solución, sino que tienen un número considerable de parámetros que determina su comportamiento y cuya variación modifica considerablemente la calidad de la solución.

La labor que realizar, por tanto, no consiste solo en desarrollar la aplicación que implemente la búsqueda de la solución al problema mediante estos tres métodos, sino también la labor de investigación consiste en hallar las mejores configuraciones para cada uno de ellos y mostrar cuál se muestra más efectivo, como se verá más adelante en el punto "*Análisis de resultados*".

4.1. Algoritmo de colonias hormigas

El algoritmo de colonia de hormigas [14, 22, 16] se basa en el mecanismo que usan las hormigas para comunicar que camino tienen que ir siguiendo las hormigas de la colonia para encontrar una fuente de comida y volver de nuevo a su colonia.

En los siguientes subapartados se tratarán el funcionamiento general y posterior adaptación del algoritmo de colonia de hormigas, primero veremos el comportamiento de las hormigas en la naturaleza, después el funcionamiento general del algoritmo extraído de esta idea y por último la adaptación.

Comportamiento de las hormigas y funcionamiento básico

El proceso que sigue una colonia de hormigas para encontrar el mejor camino hasta una fuente de alimentos consiste en dejar un rastro de feromonas por cada sitio que van pasando, entonces cuantas más hormigas pasan en el mismo periodo de tiempo por un punto concreto, más feromonas hay depositadas en este punto.

Con el tiempo las feromonas se evaporan y si un camino no es muy concurrido acabara quedando con un número pequeño de feromonas y las hormigas dejaran de pasar por este.

Al final acaban pasando más hormigas por el mejor camino y estas dejan más feromonas con lo que este camino se refuerza.

Con el objetivo de mostrar gráficamente este proceso, se va a ver una ilustración donde se observa de manera sencilla como se comporta una colonia de hormigas en la vida real para realizar la labor de encontrar el mejor camino hasta una fuente de alimento.

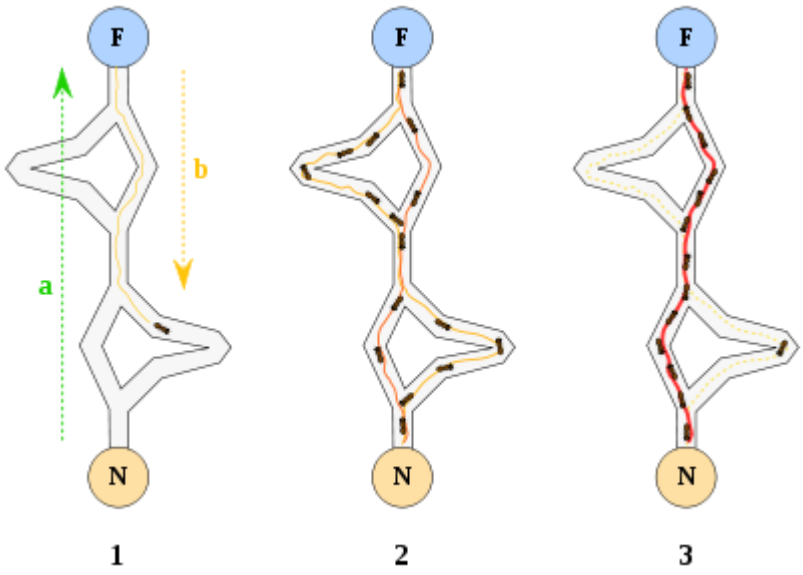

Figura 4.1 [15] – Comportamiento de hormigas en la naturaleza

En el primer dibujo empezando por la izquierda podemos observar que hay una hormiga solo que sale en busca de una fuente de alimento siguiendo un camino cualquiera.

En el segundo dibujo la primera hormiga ya ha avisado al resto de la colonia, y estas se dirigen desde la colonia de hormigas hasta la fuente de comida por diferentes caminos, pero como se puede observar el camino más corto se va reforzando poco a poco porque las hormigas que pasan por ese camino vuelven antes y depositan más número de feromonas por unidad de tiempo.

En el último dibujo casi todas toman el mejor camino porque en el resto de los caminos se siguen menos lo que hace que se vayan evaporando las feromonas y las hormigas dejen de seguirlos.

Una vez descrito el funcionamiento básico, vamos a explicar más en detalle como establecen las hormigas el camino más corto.

En la siguiente figura podemos observar como las hormigas llegan a un punto en el que tienen que decidir entre dos caminos

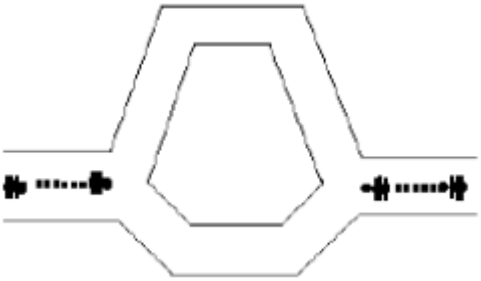

Figura 4.2 [16] – Estado inicial de las hormigas

Una vez llegan al punto de intersección las hormigas eligen al azar uno de los dos caminos hasta la fuente de alimentos.

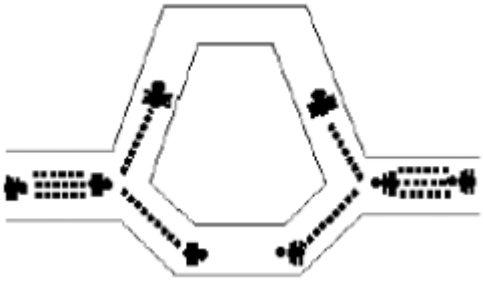

Figura 4.3[16] – Hormigas recorriendo caminos

Las hormigas acaban llegando a la fuente de alimentos por los dos caminos, pero al moverse a una velocidad constante, se depositan más feromonas por el camino de abajo, ya que en el mismo periodo de tiempo a las hormigas de abajo les da tiempo a volver más rápido de la fuente de comida, entonces las hormigas de abajo depositan más feromonas por unidad de longitud.

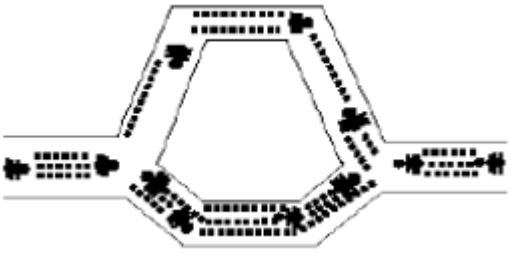

Figura 4.4 [16] – Comportamiento de las hormigas con ante los diferentes caminos

Al final, como el camino de abajo tiene más feromonas que el de arriba, las hormigas tienden a usar el camino de abajo y el sistema se retroalimenta, cuantas más hormigas más se refuerza el camino, lo que hace que más hormigas vayan por ese camino y se depositan más feromonas y así sucesivamente.

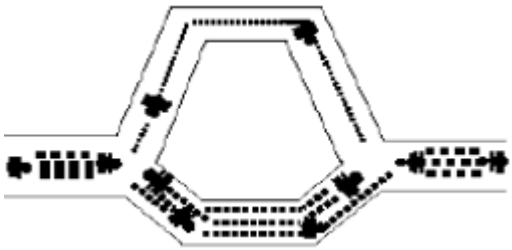

Figura 4.5 [16] – Refuerzo del mejor camino

Adaptación

En este punto trataremos la adaptación de la representación del problema y la implementación de este con el algoritmo de hormigas.

La representación del problema en este caso consiste un grafo en el que cada componente tiene el coste de ir de un nodo a otro y el número de feromonas que hay en esa transición (inicialmente todos tendrán el mismo número de feromonas), a parte tenemos una lista de nodos críticos, a partir de cada nodo critico se pondrán un número determinado de hormigas para qué se dirijan desde ese nodo hasta el nodo final. Las hormigas solo tienen el nodo actual y el camino que han seguido para llegar a el nodo actual.

Es una representación muy sencilla y funcional, las hormigas usan exclusivamente las feromonas para decidir cuál va a ser su siguiente destino, y para decidir cuantas feromonas depositar se utiliza el coste de la solución. Con lo que, teniendo coste y feromonas de un nodo al siguiente en el grafo, y el camino en la hormiga, el algoritmo

es completamente funcional y hace que las hormigas acaben transitando los mejores caminos para llegar a la solución final desde cada nodo crítico.

Una vez descrito como vamos a representar nuestro problema, vamos a explicar cómo funcionaría el algoritmo:

Inicialmente se pone un número fijo de hormigas en cada nodo crítico del problema, en cada iteración se calcula cual va a ser el siguiente nodo que va a visitar cada hormiga teniendo en cuenta el número de feromonas que hay en cada nodo, por ultimo las hormigas se van moviendo por el grafo de esta manera hasta llegar al nodo destino.

Cuando una hormiga llega al destino deposita feromonas por todo el camino que ha recorrido y vuelve a lanzarse en un nodo crítico aleatorio, en cada iteración cada hormiga se mueve un paso y se evapora un porcentaje de feromonas en todas las aristas del grafo, este es el funcionamiento básico.

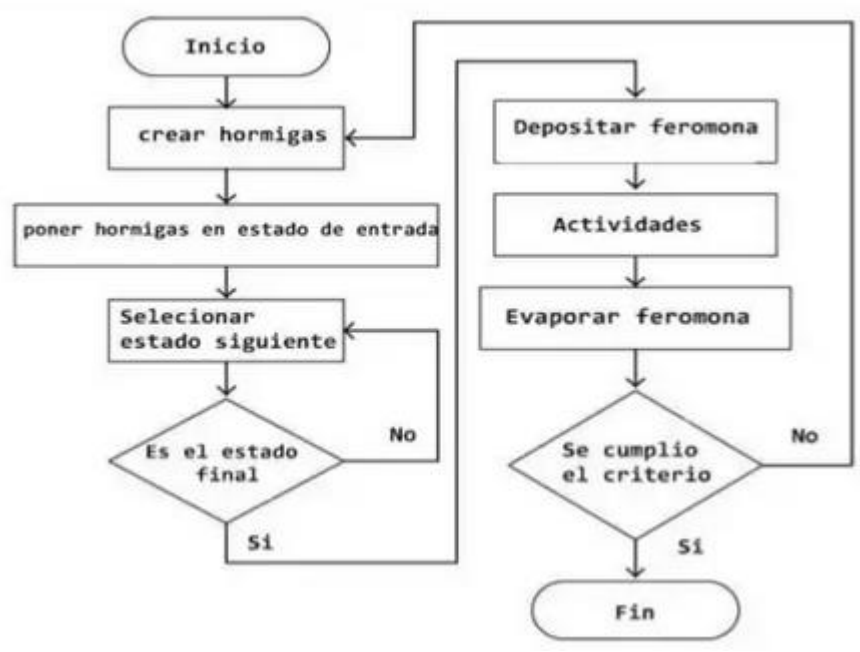

Figura 4.6 [15] – Esquema general ACO

A la hora de implementar el algoritmo nos dimos cuenta de que tenía que cubrir una serie de requisitos:

1. Necesitamos que el algoritmo refuerce las soluciones con mejor coste global.
2. Como va a decidir la hormiga que nodo escoger de un nodo al siguiente, si escogemos siempre el camino con más feromonas el algoritmo se estanca.
3. La evaporación tiene que funcionar de tal manera que ayude a que se creen nuevos caminos.

Se han solventado de la siguiente manera:

- Para reforzar las mejores soluciones la cantidad de feromonas depositadas depende del coste del camino recorrido por la hormiga, a mayor coste menos feromonas depositadas. Otra idea que se barajo es la de depositar las feromonas teniendo en cuenta el coste final del árbol.

- Para decidir qué camino deben seguir las hormigas utilizan el método de selección ruleta siendo más probable escoger el camino con más feromonas, con esto conseguimos que se muevan con más libertad al principio y a la vez conseguimos más variedad al evaporar las feromonas.
- La evaporación es un porcentaje de feromonas que se elimina en todos los puntos del grafo en cada vuelta del algoritmo, de esta manera evapora más cantidad de feromonas en las aristas con mayor número de feromonas. Con esto conseguimos aumentar la variedad de caminos que puede seguir una hormiga al salir de un nodo crítico.

Después de solventar todos estos puntos y empezar a hacer pruebas nos encontramos con otros dos inconvenientes en la adaptación, la evaporación sigue sin dar suficiente variedad al algoritmo y al ir depositando las feromonas según van llegando las hormigas el algoritmo escoge antes caminos más cortos que contengan solo un nodo critico antes de escoger caminos que cubren más nodos críticos y en el cómputo tienen un mejor coste.

Para solucionar el primer inconveniente decidimos aumentar el porcentaje de evaporación en cada iteración hasta un máximo de evaporación y una vez se llega a ese máximo el porcentaje de evaporación vuelve a estar como inicialmente.

El segundo inconveniente lo solucionamos esperando en cada iteración a que todas las hormigas lleguen al nodo destino, entonces depositamos todas las feromonas de todos los caminos a la vez sin importar el tamaño en número de nodos del camino, sino teniendo en cuenta solo el coste del camino y el número de nodos cubiertos por un camino.

Esta versión ya daba buenos resultados, pero fijándonos en como habíamos adaptado RFD, vimos que se podía hacer más. En la última iteración de la adaptación cada hormiga va formando un árbol solución, en cada iteración se va lanzando la hormiga por los nodos críticos que no haya visitado y cuando este árbol cubre todos los nodos críticos deposita feromonas según la calidad de la solución que forma el árbol.

4.2. Algoritmo genético

Los algoritmos genéticos [18, 21, 16] son algoritmos de búsqueda basados en la selección natural y la teoría de la evolución de Darwin. Combinan las técnicas de supervivencia de los individuos más adaptados a un medio, con la mutación de ADN y el cruce de cromosomas que se da cuando una especie animal se reproduce. Estos se utilizan con éxito en gran variedad de problemas donde las técnicas convencionales de búsqueda no otorgan buenos resultados.

Descripción y funcionamiento

Principios en los que está basado

Los principios básicos de los algoritmos genéticos se derivan de las teorías sobre la evolución de Darwin, de estas teorías se extraen los siguientes conceptos:

- Existe una población con individuos que son diferentes y tienen distintas habilidades.
- La naturaleza crea nuevos individuos siempre parecidos a los ya existentes.
- Los individuos mejor adaptados tienen más probabilidades de sobreviven y los peor adaptados menos, por lo que los nuevos individuos se seleccionan, en mayor medida, a partir de los individuos mejor adaptados.

A nivel biológico este proceso de selección tiene lugar gracias a la reproducción, donde se cruzan los genes de dos individuos de la población con distintas habilidades y se producen mutaciones, que finalmente crean nuevos individuos con nuevas habilidades. Las células que contienen el genoma de un individuo son los cromosomas.

Los cromosomas, en el caso de la biología, portan el código genético de cada individuo, este código contiene el conjunto de características de un individuo. En los GAs los cromosomas son codificaciones de una solución al problema. Normalmente se suelen utilizar representaciones binarias, enteras o reales. Aunque se pueden utilizar otro tipo de representaciones.

Una vez explicados los principios biológicos que sustentan estos algoritmos, es momento de describir en qué consisten a nivel de computación.

Definición

Los algoritmos genéticos ocupan un lugar central en la computación evolutiva. Esto es así porque reúnen las ideas fundamentales de la computación evolutiva, por su flexibilidad a la hora de combinarse con otros métodos y de adoptar nuevas técnicas, y porque el propio algoritmo no necesita tener ningún conocimiento sobre la aplicación.

Los algoritmos genéticos son métodos estocásticos de búsqueda ciega de soluciones cuasi-optimas. Mantienen una población de individuos, esta población es sometida a transformaciones y después de un proceso de selección se crea una nueva población

nacida de los individuos transformados de la anterior población escogiendo qué individuos sobreviven usando la selección.

Los GAs son métodos de búsqueda:

1. Ciega: no saben nada sobre el problema a resolver, su búsqueda se basa en los valores de la función objetivo.
2. Codificada: trabajan sobre una codificación de un conjunto de soluciones posibles al problema.
3. Múltiple: procesan un conjunto completo de soluciones.
4. Estocástica: referida tanto a las fases de selección como a las de transformación.

¿Qué es un individuo?

Un individuo es una solución del problema que se quiera resolver y un solo miembro de la población, el individuo está compuesto por:

- Genotipo: consiste en la codificación de los parámetros de una solución del problema a resolver.
- Fenotipo: es la decodificación del genotipo, es decir se convierte la codificación con la que se trata al individuo durante la ejecución del algoritmo a su valor real dentro de nuestro problema.

¿Qué es la población?

La población contiene un conjunto de individuos, normalmente tiene un tamaño fijo y puede haber varios individuos con el mismo genotipo. Los operadores genéticos habitualmente tienen en cuenta el tamaño de la población, es decir, las probabilidades son relativas a la generación actual de individuos.

Funcionamiento

El funcionamiento del algoritmo consiste en crear una población inicial de individuo. Sobre esta población se hacen transformaciones (cruce y mutación, por ejemplo), después de estas transformaciones se utiliza un método de selección para favorecer a los mejores individuos de la población, y con esto se genera una nueva población en la que solo se encuentran los individuos escogidos en la selección. A cada ciclo de transformación + selección se le llama generación.

Después de alcanzar el objetivo, o de pasar por un número n de generaciones. El mejor individuo de la población es la mejor solución que ha encontrado nuestro GA.

Aquí tenemos un esquema básico del funcionamiento de un GA y un pseudocódigo de ejemplo:

Figura 4.7 [16] – Esquema general GA

Pseudocódigo:

```
t = 0;
Generar población inicial(P(t));
Evaluar población(P(t));
Mientras (t < Num_max_gen) y no CondFin() {
     t++;
     Poblacion(t) = Seleccion(P(t-1));
     Reproduccion(P(t));
     Evaluar Poblacion(P(t));
}
```

¿Cómo definir el genotipo, los operadores y la selección?

Hay muchos factores determinantes para el buen funcionamiento de un AG, pero uno de los principales es conseguir una buena codificación del conjunto de soluciones con el que trabaja este, es decir, la representación del individuo. Una buena codificación puede marcar la diferencia, hay que fijarse bien en la dificultad que conllevara después implementar los operadores genéticos y que estos vayan a darle variedad a la población.

Una vez se tiene una codificación hay que trabajar en la función heurística que medirá la calidad de las soluciones, dentro de este apartado pones el foco sobre todo en reflejar lo más fielmente posible la calidad que tiene cada solución dentro de nuestro problema, comúnmente se utiliza directamente el valor que devuelve la función del problema a resolver introduciendo el fenotipo, para ello se utiliza una función de fitness, una función de fitness consiste en una función con la que a partir de una

heurística fijada por la persona que adapta el algoritmo genético valoramos la calidad de una solución, esta función debe de reflejar muy bien esta calidad

Con la estructura del individuo ya definido, se puede comenzar a trabajar en los diferentes operadores genéticos para darle variedad a nuestra población y en la selección para ir quedándonos con las mejores soluciones.

Los principales operadores son el cruce y la mutación. El cruce consiste en mezclar los genotipos de dos individuos de la población y la mutación en hacer una pequeña modificación al azar en un individuo concreto. Los operadores genéticos se ejecutan solo en un porcentaje de la población habitualmente, el orden de ejecución de estos depende del problema y a veces se ejecutan varias veces la mutación o el cruce o se elimina uno de ellos, también se pueden diseñar otros operadores que se adapten mejor al problema a resolver. Por eso a la hora de definir el algoritmo genético decimos que es un algoritmo muy flexible, puedes añadir o eliminar módulos según convenga.

La selección suele hacerse justo después de ejecutar los operadores genéticos y de volver a evaluar el valor de fitness que tienen todos los individuos de la población, hay muchas formas de seleccionar a los individuos de la población. Ahora se va a hacer un repaso de las más utilizadas:

- **La selección de ruleta**: esta selección se caracteriza por seleccionar a los individuos de la población de manera proporcional a su función de fitness, el proceso consiste en calcular una probabilidad de selección para cada individuo de la siguiente manera, fitness(i) / sumatorio de fitness de la población, y con esto hacer una tabla de probabilidades.

individuo	1	2	3	4	5	6	7	8	9	10
fitness	2.0	1.8	1.6	1.4	1.2	1.0	0.8	0.6	0.4	0.2
Prob. selección	0.18	0.16	0.15	0.13	0.11	0.09	0.07	0.06	0.03	0.02

Figura 4.8 [16] – Ejemplo de selección de ruleta

A partir de esta se crea una nueva tabla con las probabilidades acumuladas.

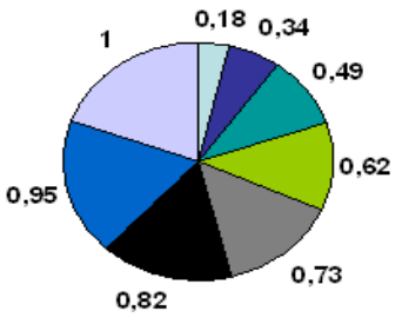

Figura 4.9 [16] – Repartición final de ruleta

Por último, se lanza un generador de números aleatorios, para cada número se mira en que parte de la tabla de probabilidades acumuladas cae, y según eso se decide que individuo se copia a la nueva población. Este proceso se repite hasta tener una nueva población entera.

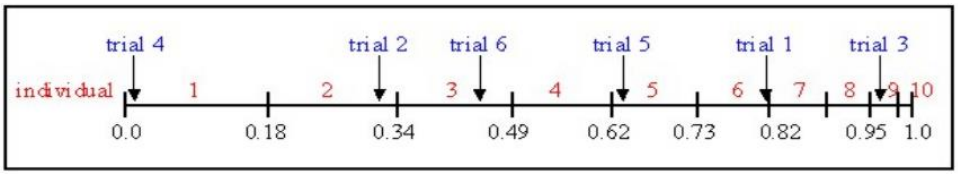

Figura 4.10 [16] – Representación final ruleta

- **La selección estocástica:** esta selección es muy parecida a la de ruleta, la única diferencia entre las dos es la generación de números aleatorios. En ruleta se genera un numero aleatorio para cada individuo a seleccionar de la tabla de probabilidad. En cambio, la selección estocástica genera un solo aleatorio, y a partir de este calcula el resto sumando 1/n, siendo n el número de individuos a seleccionar. La posición de la primera marca es siempre un valor entre 0 y 1/n y el resto están a una distancia 1/n de la siguiente marca. En el siguiente ejemplo podemos observar un caso en el que se quieren seleccionar 6 individuos, por lo tanto, la distancia entre las marcas es de 1/6 y la primera marca es un valor entre 1/6.

individuo	1	2	3	4	5	6	7	8	9	10
fitness	2.0	1.8	1.6	1.4	1.2	1.0	0.8	0.6	0.4	0.2
Prob. selección	0.18	0.16	0.15	0.13	0.11	0.09	0.07	0.06	0.03	0.02

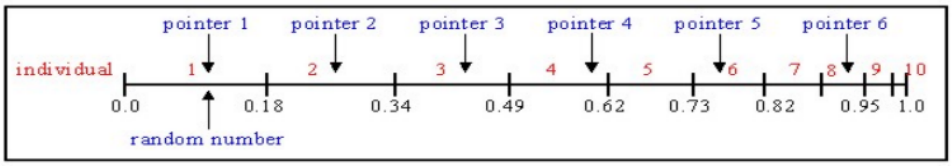

Figura 4.11 [16] – Probabilidades y representación método selección estocástico

- **Selección por torneo determinista:** esta selección escoge un grupo de n individuos al azar, y de esta selección se queda con los m mejores, siendo siempre m menor que n. Se repite este proceso hasta que tenemos un número suficiente de individuos. Aquí tenemos un ejemplo sencillo, en este caso n=6 y m=4.

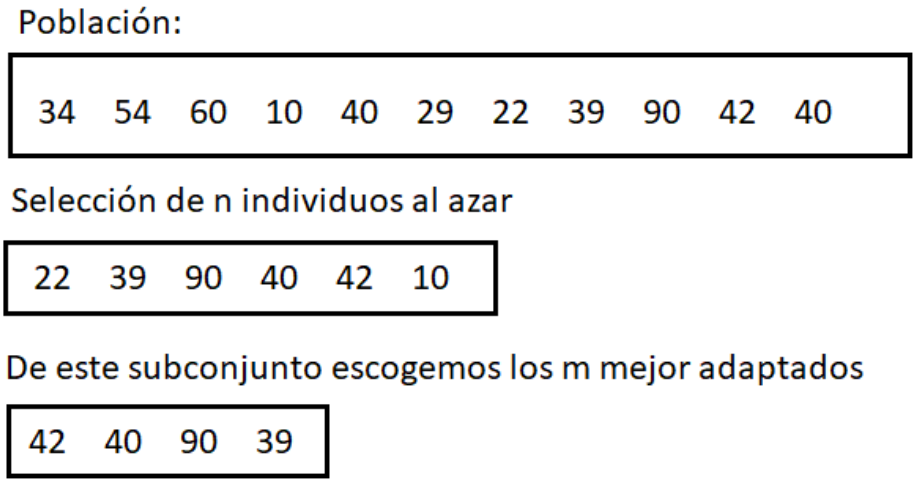

Figura 4.12 – Explicación del funcionamiento de torneo

- **Selección por torneo probabilista:** es igual que la determinista, pero a la hora de escoger los m elementos saca un valor aleatorio 0 o 1 para cada individuo, y según el valor de este se escoge un individuo con un buen valor de fitness o se escoge un individuo con un valor bajo de fitness del subconjunto.

A parte de la selección existe otro método para destacar las mejores soluciones, este es el elitismo, es un método muy sencillo que consiste en quedarte con los n mejores individuos de la población para que estos no se pierdan, luego vamos juntamos estas mejores soluciones con el resto de la población para ver si conseguimos a partir de una solución buena otra mejor, a veces lo que se hace es poner estas soluciones en los huecos donde van las peores soluciones de la población.

Adaptación

Para adaptar el algoritmo los factores que se han tenido en cuenta son la codificación de las soluciones, el diseño de los operadores genéticos y que tipo de selecciones íbamos a implementar.

El primer paso en la adaptación del algoritmo fue pensar cómo codificar las soluciones, es decir, cuál iba a ser el genotipo del individuo. Al principio se pensó en codificar los individuos como una cadena de enteros y que esta cadena representase la totalidad del árbol solución utilizando un carácter especial para dividir las ramas del árbol.

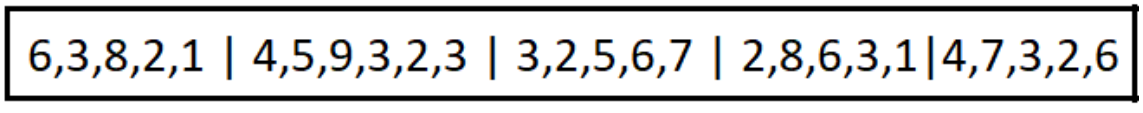

Figura 4.13 – Representación inicial del individuo

El problema de esta estructura era que no se podían representar árboles con una rama que colgase de cualquier nodo que no fuese el nodo 0, ya que teníamos un conjunto de ramas del árbol, una detrás de otra y no tenemos nada para representar la raíz de cada una de ellas.

La segunda estructura en la que pensamos fue utilizar un montículo, pero como nuestro árbol tiene múltiples hijos por cada nodo, no podíamos representar bien una solución con un montículo.

Al final, a pesar del aumento en coste que esto conlleva, utilizamos el propio valor de la solución como codificación de nuestro problema. Trabajar con árboles, a priori parecía que iba a darnos muchos problemas a la hora de diseñar los operadores genéticos y que los costes se iban a disparar. En realidad, los resultados han sido bastante mejores de los esperado; la adaptación ha sido compleja, pero los resultados están bastante bien. Todo esto teniendo en cuenta que en general los otros dos algoritmos se adaptan mejor a este problema y este es el que sale peor parado en los análisis de resultados, pero bueno ese tema lo trataremos en otro apartado.

Una vez tenemos nuestro individuo hay que pensar cómo se va a realizar el diseño de los operadores genéticos. En nuestro caso se ha implementado un cruce y dos mutaciones (mutación por transformación y mutación por eliminación), a partir de la interfaz de la aplicación se puede utilizar una o la otra.

El cruce consiste en cruzar una rama de un individuo al azar con otra rama de otro individuo, siempre se escogen dos ramas que contengan en su nodo hoja el mismo nodo crítico, y se cruza la rama entera. El primer paso es buscar un nodo candidato empezando por el nodo hoja de una rama del árbol de manera aleatoria, entonces subimos hasta el primer nodo que tenga un padre con más de un hijo. Este es el nodo seleccionado para cruzar en ese individuo, luego repetimos el mismo proceso con el otro individuo.

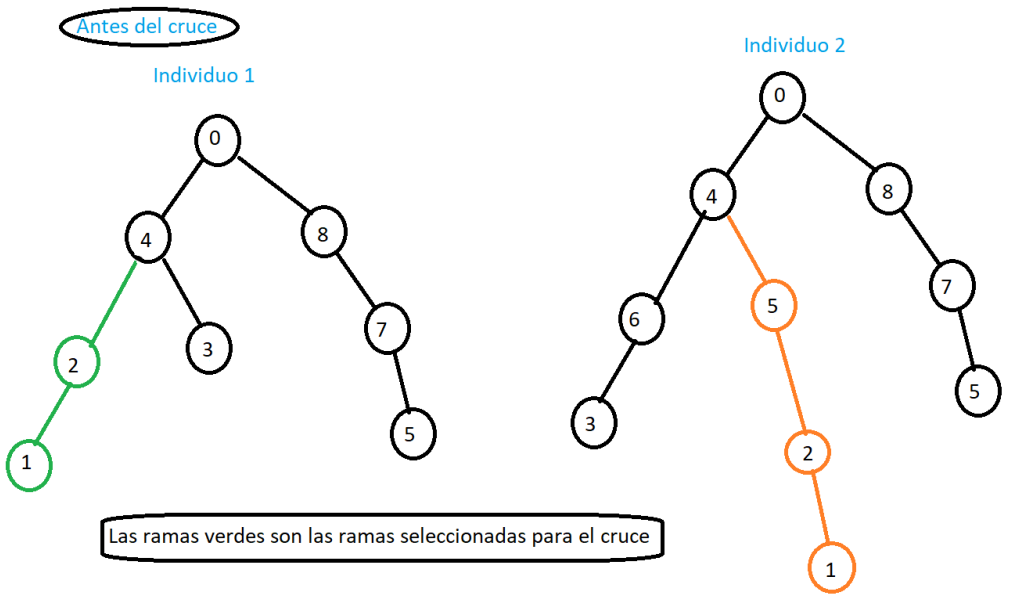

Figura 4.14 – Selección de las ramas para el cruce

Como se puede observar en el dibujo se escogen siempre ramas que no tengan ninguna bifurcación.

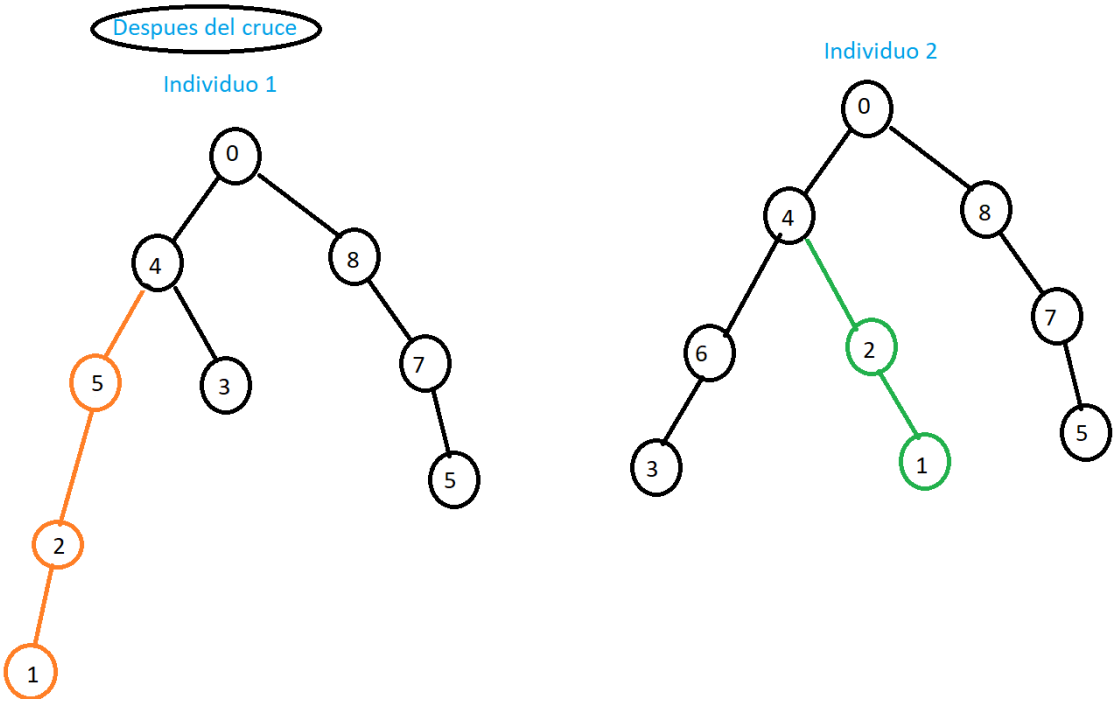

Figura 4.15 – Resultado del cruce

La mutación por transformación simplemente cambia el valor de un nodo del árbol al azar por otro nodo, no se puede cambiar el valor del nodo 0, ni darle el valor 0 a ningún otro nodo del árbol.

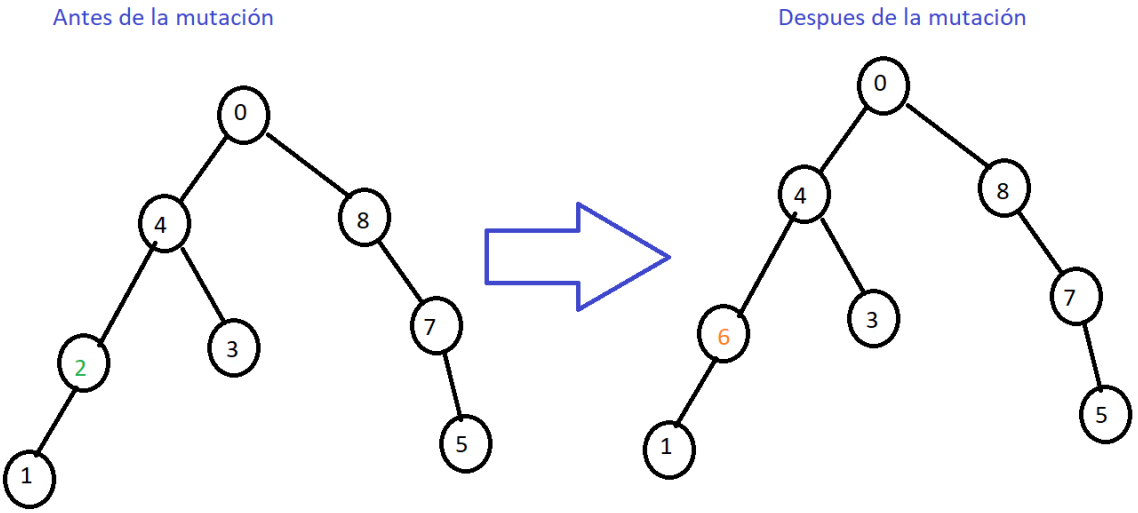

Figura 4.16 – Mutación por transformación

La mutación por eliminación elimina un nodo al azar del árbol y los hijos de este nodo pasan a ser hijos de su nodo padre.

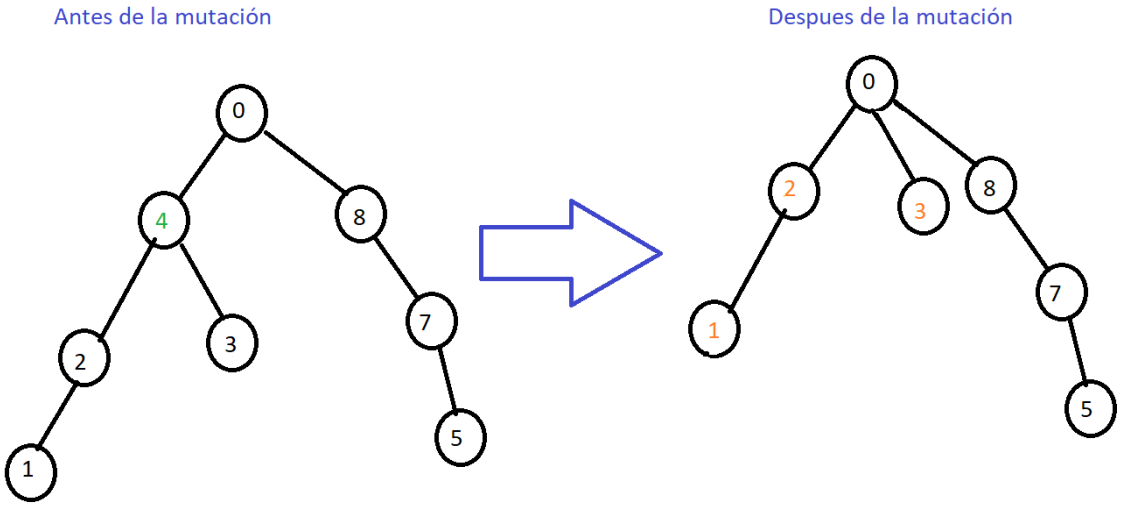

Figura 4.17 – Mutación por eliminación

Tanto la mutación como el cruce se aplican en un porcentaje de la población, este proceso funciona de la siguiente forma en cada caso:

- En el caso de la mutación, para cada individuo se genera un numero aleatorio entre 0 y 1, si cae entre 0 y el porcentaje de mutación el individuo se muta.
- En el caso del cruce se sacan dos individuos aleatorios de la población, y después se saca un numero aleatorio entre 0 y 1 y si al igual que en la mutación si este es menor que el porcentaje de cruce, los dos individuos se cruzan, este proceso se repite un número de veces concreta y siempre se sustituyen los individuos escogidos por los nuevos que surgen del proceso de cruce.

Los tipos de selección que hemos implementado son la selección por ruleta, la selección estocástica y la selección por torneo determinista. La implementación es tal y como se describe en el apartado “descripción y funcionamiento”.

Además, también hemos implementado un sistema de elitismo, que consiste en quedarse con un conjunto de los mejores individuos de todas las generaciones hasta la iteración que se está ejecutando, e introducir estos en la posición donde van los peores individuos de la población.

4.3. Algoritmo de formación de ríos

El algoritmo de formación de ríos (RFD) sigue la idea del evento que ocurre en la naturaleza cuando se forma un río:

1. Llueve sobre el terreno.
2. Las gotas de lluvia avanzan hasta el mar por las pendientes más favorables, erosionando el terreno por el que pasan, y depositando los sedimentos arrancados.

Este fenómeno favorece las pendientes más pronunciadas, ya que la velocidad de descenso de las gotas hace que la erosión producida en estas sea mayor, y abandona las pendientes menos eficientes, pues las gotas pierden velocidad, bien hasta llegar al mar con una velocidad despreciable (la erosión producida no es relevante), o incluso hasta quedarse paradas y no llegar al destino.

Descripción y funcionamiento

El algoritmo RFD, es un algoritmo heurístico propuesto por P. Rabanal en 2007 [1,10,19]. Esta heurística trata de imitar el comportamiento de las gotas de lluvia en la naturaleza cuando forman caminos que llegan hasta el mar.

En un principio, se presenta un terreno plano y un destino: el mar. Este terreno no tiene pendientes, estas se formarán cuando comience la lluvia y las gotas erosionen la superficie.

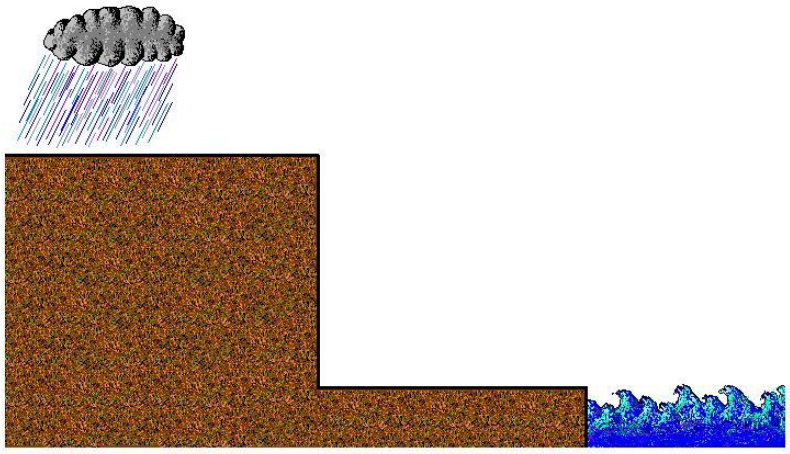

Figura 4.18 [17] – Terreno plano y mar

Al producirse la lluvia, las gotas comienzan a moverse por el terreno plano hasta llegar al mar, formando pendientes en el camino realizado. Cuando vuelva a llover, las gotas tratarán de seguir las pendientes más favorables, debido a que estas hacen que las gotas se muevan a mayor velocidad, arrancando más parte del terreno. Además de erosionar, las gotas depositan los sedimentos arrancados a lo largo del camino que han recorrido.

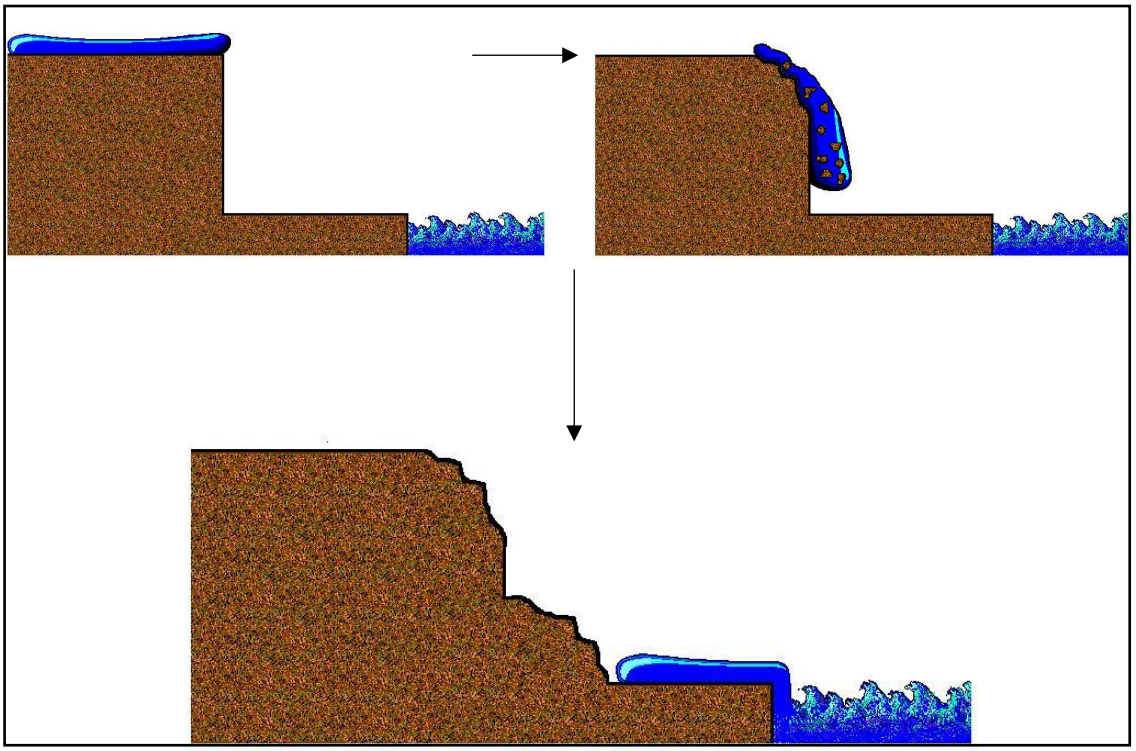

Figura 4.19 [17] – Proceso de formación de caminos

Al repetirse numerosas veces el anterior proceso, se terminan formando uno o varios caminos que permiten a las gotas llegar al mar de forma rápida (eficiente).

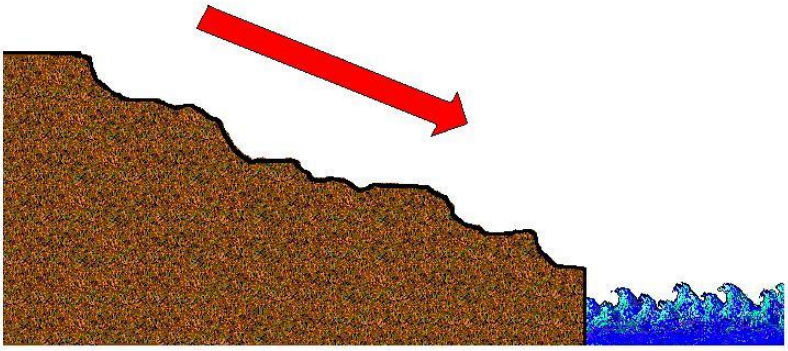

Figura 4.20 [17] – Terreno tras la formación de caminos

La traducción de este fenómeno de la naturaleza a algoritmo heurístico utiliza como terreno un grafo, cuyos vértices tienen altura, simulando así las pendientes, y también se guarda el coste de realizar una transición de un nodo a otro. Al comienzo del algoritmo, las alturas de todos los nodos -excepto el nodo 0- son inicializadas al mismo valor, formando así un terreno plano; el nodo 0 se considera como el destino o mar, y su altura siempre será 0, permitiendo así que las gotas que estén en los nodos adyacentes traten de ir ahí.

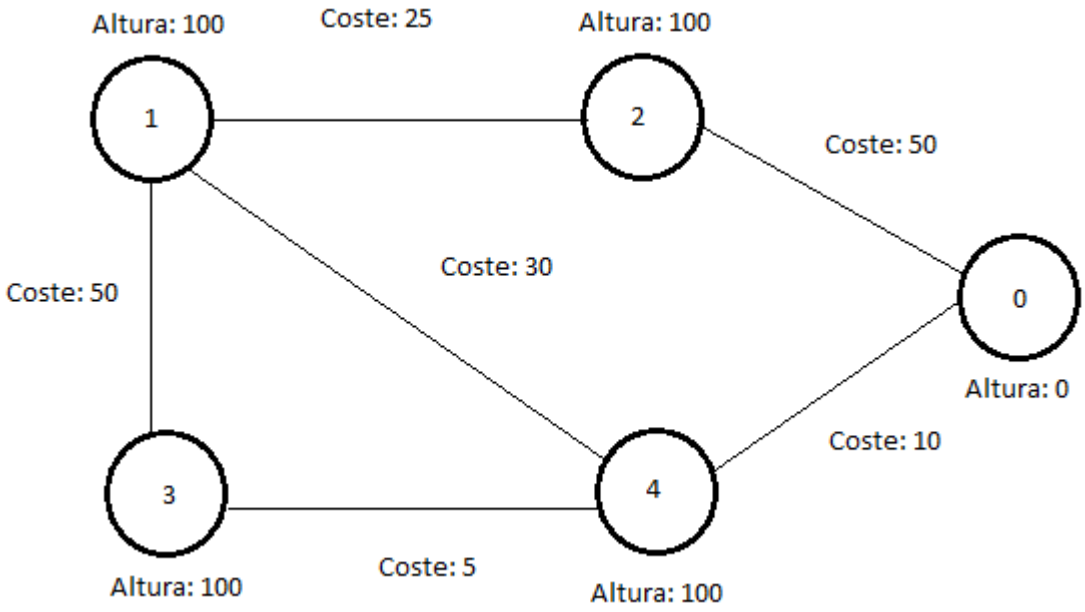

Figura 4.21 - Grafo que representa un terreno plano

Hay que introducir una mejora al terreno presentado, ya que, si se quiere forzar a que una gota pase por un cierto camino, por ejemplo $1 \rightarrow 3$, es posible y muy probable que tras erosionar el nodo 4, al realizar el camino $1 \rightarrow 3 \rightarrow 4$, este favorezca que las gotas tiendan a realizar el camino $1 \rightarrow 4$, evitando realizar el camino que se quería imponer. Debido a este motivo, se incluyen los *nodos barrera* o aristas con altura, que imposibilitan que ocurra la situación anterior.

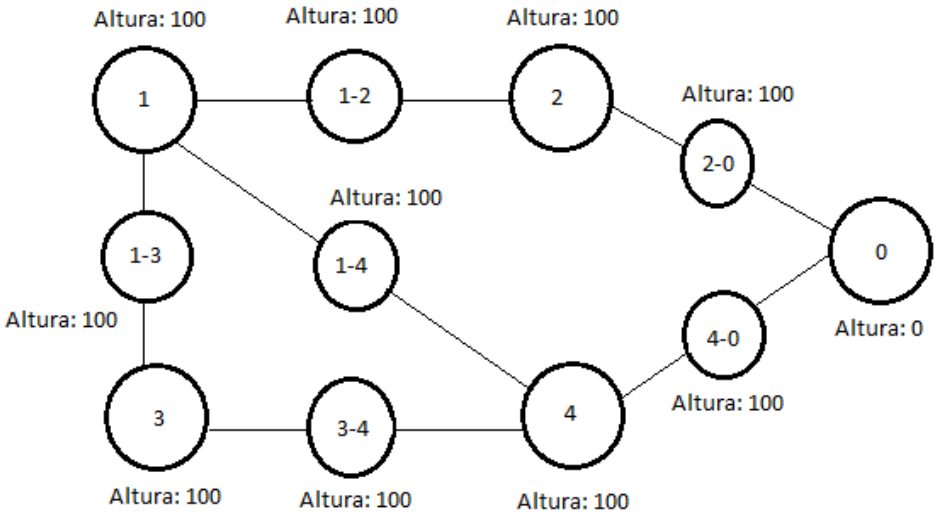

Figura 4.22 - Grafo que representa un terreno plano con nodos críticos

Una vez definido el terreno, el esquema general del algoritmo es el siguiente:

```
initializeDrops()
initializeNodes()
while (not allDropsFollowTheSamePath()) and
      (not otherEndingCondition())
   moveDrops()
   erodePaths()
   depositSediments()
   analyzePaths()
end while
```

Figura 4.23 - Esquema General RFD [10]

Se generan las gotas en los nodos que se desea empezar a resolver el problema, y se comienza un bucle donde en un inicio las gotas se mueven por el terreno plano hacia los distintos nodos alcanzables, erosionan los nodos y aristas con altura por los que han pasado, depositan sedimentos recogidos al erosionar (medida para que la altura de estos no llegue a 0, y evitar caminos despreciables), y por último se analizan las soluciones obtenidas. Las gotas al llegar al nodo destino se evaporan para volver a comenzar el bucle, esta vez no se moverán por un terreno plano, si no que seguirán las pendientes más favorables formadas por las gotas en las anteriores iteraciones, reforzando los mejores caminos.

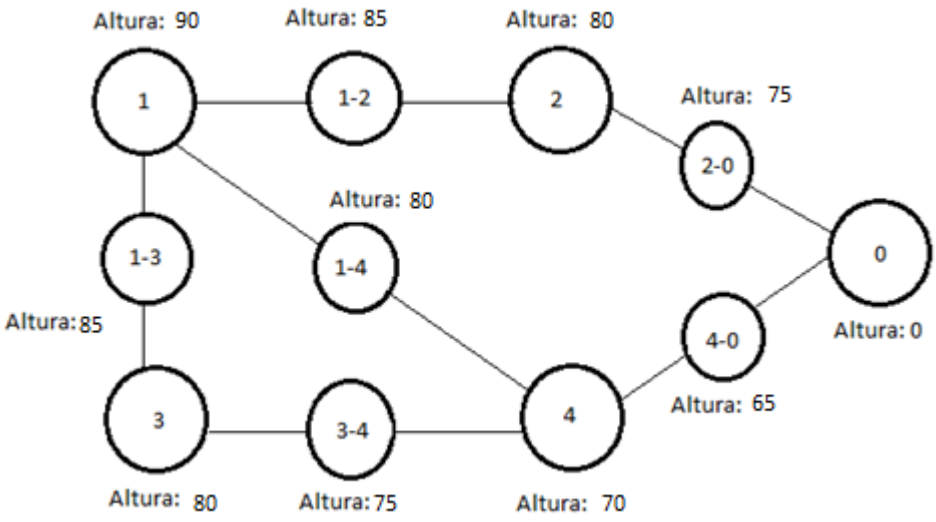

Figura 4.24 - Terreno tras la ejecución del esquema general

En el esquema general descrito anteriormente, se puede añadir un comportamiento que amplía la exploración de las gotas, permitiendo encontrar nuevos caminos. Esta mejora es llamada *Gota Trepadora*, -que como su propio nombre indica- permite que las gotas puedan ir por pendientes ascendentes. Esta mejora se presenta como un porcentaje activo desde el inicio del algoritmo, que al principio permite que muchas gotas puedan moverse hacia nodos con pendiente desfavorable, y se va decrementando conforme el tiempo o iteraciones van aumentando.

Adaptación

Para abordar el MLS, el terreno de RFD no se representa con el grafo original, se completa usando Dijkstra [11], con el que se rellena el grafo con las conexiones mínimas a todos los nodos. Esto permite que el algoritmo pueda explorar más fácilmente, y llegar a soluciones que sin esta premisa no se conseguirían.

Tras distintas implementaciones e iteraciones, el esquema general del RFD adaptado a este problema es el siguiente:

```
inicializaNodos()
inicializaGotas()
while(!condicionFin()) {
        while(!todasGotasCubrenNodos()){
                moverGotas()
                depositar()
                quitarGotasConCiclos()
                evaporar()
        }
        solucionParcial()
        inicializaGotas()
}
```

Figura 4.25 - Esquema de RFD adaptado a MLS

Como se puede observar, se comienza iniciando las alturas de los nodos y las gotas en los nodos críticos que se desea visitar, entonces comienza un bucle:

- Dentro de este bucle, se ejecuta otro, cuya condición de fin es que todas las gotas hayan cubierto los nodos críticos, ya que las gotas guardan las posibles soluciones. Para que una gota haya recorrido todos los nodos críticos, esta ha tenido que pasar por ellos al menos en una rama.
    - Se mueven las gotas por los distintos nodos: las gotas en un principio se mueven a todos los nodos por igual, y más adelante, cuando se hayan formado pendientes, se mueven siguiendo las pendientes óptimas.
    - Si hay gotas estancadas en un valle (todas las pendientes son desfavorables), estas depositan sedimentos en el nodo donde estén para aumentar la altura y así poder volver a tener pendientes favorables. Este evento solo ocurre cuando la probabilidad de *Gotas Trepadoras* ha decrementado, y las gotas no pueden subir por pendientes inversas.
    - Se quitan las gotas con ciclos, es decir, las gotas que en la misma rama han pasado más de una vez por el mismo nodo. Para dar una mayor variedad a los resultados, esto no se hace en cuanto una gota ha repetido un nodo, si no que se puede definir por parámetro que longitud tiene que tener la rama para que se elimine.
    - Al final de este bucle, se evaporan las gotas que hayan llegado al nodo destino:
        - Si la gota ha cubierto todos los nodos críticos, se marca para valorar.
        - Si la gota no ha cubierto todos los nodos críticos, se evapora en el nodo crítico más alto que aún no haya visitado. Antes de realizar la evaporación, se guarda en el árbol la rama que la gota ha conseguido, bien cargando (si el coste de carga es favorable) en un nodo ya visitado anteriormente o enganchándolo al nodo destino directamente.
- Al finalizar el bucle, se evalúan las soluciones obtenidas por las gotas (cada gota ha obtenido una solución):
    - Se erosionan las soluciones en función del coste obtenido (a menor coste, mayor erosión). La erosión se ha abordado de esta forma, ya que el problema necesita que se valoren las soluciones globalmente (por su coste total), y en iteraciones anteriores -donde se premiaba caminos localmente- no se obtuvieron buenos resultados.
    - Se depositan los sedimentos arrancados en la erosión, para evitar que los nodos lleguen a altura 0. Para obtener más variedad, se ha añadido un parámetro a la sedimentación, el cual decide a partir de qué altura se realiza esta, para así obtener una mayor exploración.

- Finalmente, se inicializan las gotas de nuevo en los nodos críticos a visitar, y se continua la ejecución del bucle principal hasta que se llega a la condición de finalización.

5. DESARROLLO DE LA APLICACIÓN

En este apartado se tratarán los detalles de la aplicación, poniendo el énfasis en la arquitectura de esta. Se describirá su estructura y sus componentes, así como las tareas principales que realiza cada uno de ellos y cómo se comunican entre sí.

Debido a la naturaleza del problema que se quiere abordar, el desarrollo de la aplicación se planificó pensando la funcionalidad. Se planteó como un software de carácter científico/técnico destinado a ejecutar los distintos métodos de resolución del problema para investigar las mejores configuraciones y formas de resolverlo satisfactoriamente, así como servir de herramienta para realizar dichos estudios y mostrar conclusiones en la presente memoria.

Por estos motivos se apostó por un diseño sencillo que primara la funcionalidad y la optimización por encima de otros factores que no eran prioritarios para el cumplimiento de los objetivos. A la vez, también se pretendía que la aplicación fuese lo más cómoda posible de cara a realizar múltiples pruebas. Dado que para el estudio era necesario repetir la ejecución de los problemas en un gran número de veces, era muy importante diseñar la aplicación de forma que incorporara algunas funcionalidades destinadas a facilitar lo más posible dicha labor, pero que, al mismo tiempo, no implicaran una pérdida de rendimiento.

Toda la adaptación de los algoritmos se encuentra en el siguiente enlace:

https://github.com/Josepe04/TFG [23].

5.1. Estructura

Desde el punto de vista estructural, la aplicación se divide en cuatro grandes bloques.

En primer lugar, tenemos los elementos comunes de la aplicación entre los que se encuentran aquellos que son utilizados por el resto de componentes de la aplicación (tales como el punto de entrada a la aplicación, las vistas, etc.) así como aquellos componentes que albergan las distintas funcionalidades adicionales que, si bien no son imprescindibles para el funcionamiento de aplicación, facilitan en gran medida el trabajo con la misma (salvado y carga de casos de prueba, guardado de resultados, etc.).

Los otros tres bloques de la aplicación son para cada uno de los tres algoritmos objeto del estudio: RFD, ACO y GA. En ellos se desarrollan las funciones únicas para cada uno de ellos, así como la especificación de algunas clases comunes para la que cada una de estas tres formas de resolución necesita añadir particularidades concretas.

5.2. Clases

Clases generales:

- **Principal**: es el punto de entrada de la aplicación. Se encarga de llamar a la vista, que será la que tome el control de la aplicación durante toda su ejecución.

- **Maquina**: esta clase codifica la máquina de estados. Representa la máquina como un grafo, el cual, a su vez, es representado a nivel de código como una matriz bidimensional NXN donde sus N filas representan cada uno de los estados de la máquina. Cada una de las posiciones de la matriz (ver clase Arista) simboliza el coste de llegar de un estado a otro (o la imposibilidad de alcanzarlo de forma directa). Por ejemplo, si el valor de AXB es 10, significa que el coste de ir del nodo A al nodo B es 10. También incluye, posterior al cálculo sobre el grafo original, el grafo de costes mínimos que se halla a través del algoritmo de Dijkstra (ver clase Dijkstra más adelante) y los nodos críticos, que son aquellos estados y transiciones que el usuario de la aplicación determina como obligatorios (las partes que se probarán forzosamente).

    - Algoritmos:

        - *genererarAleatorio*: algoritmo que se encarga de generar la representación de la máquina de estados con las condiciones que le asignemos (tales como tamaño o aleatoriedad).

        - *completar*: una vez generado, el grafo debe conexo. Esto es, que desde un nodo dado se debe poder llegar a cualquier otro, de tal forma que no queden "islas". Como los costes y la posibilidad de ser alcanzados son aleatorios, puede ocurrir que exista algún nodo que quede aislado. Para corregir esta circunstancia, una vez creada la máquina esta pasa por un algoritmo que revisa su corrección y lo completa en caso necesario.

        - *creaMatrizCostesMinimos*: utilizando Dijkstra, calcula el menor coste posible que hay de un nodo origen a otro destino y lo hace para todos los nodos del grafo.

- **Arista**: esta clase representa los caminos de la máquina de estados. Es decir, para ir de un estado X a un estado Y hay un "camino" que conlleva un coste determinado. Ese camino con su coste es lo que representa la Arista. En la clase anterior (Máquina) se vio que se compone de una matriz que contiene costes. Se trata por tanto de una matriz de Aristas. Juntos representan la máquina de estados que los diferentes problemas tratan de testear. La clase abstrae la

forma más básica y general de la arista, y servirá para que cada uno de los problemas la adapten mediante herencia a sus necesidades concretas.

- **Dijkstra**: llamada como el algoritmo que implementa, sirve de herramienta para hallar el camino más corto entre dos nodos de la máquina de estados. Como ya se abordó anteriormente, sabemos que todos los nodos son alcanzables. Sin embargo, para llegar hasta un nodo desde otro puede no haber un camino directo, si no varios que atravesando diversos nodos lleguen al destino. Esto es especialmente frecuente en cuanto que empezamos a tratar con máquinas con un número considerable de estados. Por tanto, se necesita poder calcular de forma óptima cuales son los caminos más baratos, en términos de coste, para llegar desde un nodo a otro y así poder usarlo para realizar el testing de todas las partes necesarias de la mejor forma posible.

    - Algoritmos: además de las funcionalidades periféricas necesarias para realizar el objetivo sobre nuestro grafo, el algoritmo principal de esta clase es la implementación del algoritmo de Dijkstra, también conocido como "algoritmo de los caminos mínimos", creado por Edsger Dijkstra en 1959 [11], que determina el camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista.

- **Arbol**: esta clase implementa una estructura de datos de tipo árbol. El objetivo es usar esta estructura para codificar el camino que representa la solución del problema. Es decir, cada una de las soluciones hallada por los distintos algoritmos que resuelven el testing de la máquina de estados es un camino que define los nodos por los que se pasa para cubrir todos los casos de prueba designados por el usuario al inicio de la ejecución.

    - Algoritmos: En esta clase, además de implementar los algoritmos básicos típicos de este tipo de estructuras de datos, tales como hallar la profundidad, u obtener la raíz del árbol, también se implementan algunos métodos específicos para nuestro problema:
        - *arbolTieneValor:* Obtiene el árbol que tenga el valor pasado por parámetro. Esta función es esencial para el proceso de carga ya que permite conseguir el subárbol concreto (tramo del recorrido del grafo) que va a participar la carga.
        - *getArista:* Dados dos valores enteros la función devuelve (si existe) un subárbol cuyo valor sea el primer argumento y cuyo padre tiene como valor el segundo argumento de la función.

- **DatosGrafica**: esta clase sirve como herramienta para comunicar los resultados obtenidos en la ejecución de los diferentes problemas en la lógica de la aplicación y poder mostrarlos al usuario en la vista. Incorpora varias estructuras de datos para albergar los distintos elementos que se calculan en la resolución de los problemas, tales como el coste total del camino solución, o los costes parciales que se van obteniendo en cada iteración de los algoritmos.

- **Vista**: esta clase contiene todos los elementos necesarios para constituir la interfaz gráfica de la aplicación, así como las funciones que se encargan de manejar la entrada de datos y su validación para enviarla a la lógica de la aplicación e iniciar los algoritmos de resolución. Así mismo, cuenta con los métodos necesarios para mostrar los resultados al usuario. Por último, cuenta con varias funcionalidades extra destinadas a gestionar con mayor comodidad y eficacia el manejo de la aplicación, la repetición de las pruebas y el estudio de los resultados, tales como la posibilidad de guardar máquinas de estados para poder cargarlas posteriormente y poder ejecutar los problemas a lo largo del tiempo sobre una máquina concreta que se considere especialmente significativa para las conclusiones del estudio. También permite guardar ejecuciones múltiples de un mismo algoritmo de resolución y guardar cada uno de los resultados en un log para establecer posteriormente comparativas entre todos.

Clases de River Formation Dynamics (RFD):

- **AlgoritmoRFD**: clase que implementa un algoritmo que se basa en la formación dinámica de ríos y que trata de aprovechar sus mecanismos para generar caminos eficientes para aplicarlo a hallar la mejor ruta posible para pasar por todos los estados que se requieran. En esta clase se aglutina toda la funcionalidad necesaria para la resolución del problema.

    - Algoritmos:

        - *Constructor*: recibe la máquina sobre la que se va a realizar el testing, los nodos críticos por los que el usuario determina que se debe pasar (estados y aristas concretos que se someterán siempre a prueba), y el resto de parámetros propios del algoritmo, tales como las gotas, la erosión, etc.

        - *especializarGrafo*, *inicializarAlturasNodos*, *inicializarGrafo* e *inicializaAristas*: tomando la máquina (el grafo) pasada por la vista, especifican las características que necesita el algoritmo para funcionar e inicializan los diversos componentes necesarios para este proceso, como la AristaRFD (ver la siguiente clase).

        - *ejecutaAlgoritmo*: es el método principal que realiza todas las tareas para la resolución del problema. En su interior se inicializan algunas estructuras y se ejecuta el bucle donde se realizan las iteraciones solicitadas por el usuario. Dentro de este bucle se repite el siguiente proceso:
            1. Bucle en el que hasta que las gotas no hayan recorrido los nodos críticos, se realizan las siguientes acciones:
                - Mover gotas.
                - Eliminación de gotas con ciclos.
                - Deposición para solucionar la formación de *valles* (nodos rodeados de caminos con pendientes muy pronunciadas desde los cuales costaría que la gota se moviera hacia otro lugar).
                - Evaporación de las gotas que no hayan cubierto todos los nodos necesarios.
            2. Se evalúan los caminos obtenidos por las gotas, erosionando según el coste de la solución obtenida.
            3. Se sedimenta la erosión realizada.
            4. Asignación de los resultados parciales.
            5. Reinicio de las gotas.
            6. Comparación de resultados.

- *moverGotas*: este algoritmo se encarga de trasladar las gotas desde el nodo en el que se encuentran actualmente a otro nodo (que sea accesible desde el origen). En cada nodo, se calculan las probabilidades que tienen las gotas en situadas ese nodo para moverse a los nodos adyacentes. La probabilidad de que una gota decida ir a un nodo es proporcional a las pendientes con los nodos colindantes.

- *depositar*: se encarga de incrementar la altura de los nodos que no puedan acceder a los nodos cercanos, debido a que se ha formado un valle.

- *quitarGotasConCiclos*: comprueba si las gotas han realizado ciclos, y si la longitud del camino es mayor a la indicada por parámetro, la gota es eliminada.

- *evaporar:* comprueba si las gotas han recorrido todos los nodos críticos:

    - Si han recorrido todos los nodos críticos, las gotas son marcadas para valorar.
    - Si faltan nodos críticos por visitar, las gotas son evaporadas al nodo crítico restante más alto, para continuar formando una solución.

- *solucionParcial*: para cada solución encontrada por las gotas, se realiza una erosión del camino en función del coste total y se guarda la mejor solución descubierta. El proceso de erosión consiste en el decremento de la altura de los nodos y aristas pertenecientes a ese camino, este decremento será mayor con costes de solución más bajos (mejores).

- *sedimentarErosion*: la erosión realizada se divide entre todos los nodos y aristas de la máquina de estados, evitando así que las alturas de estos lleguen a cero, además de eliminar pendientes despreciables.

- **AristaRFD**: clase que especifica Arista, añadiendo la propiedad *Altura*, que permite realizar las pendientes que formarán caminos.

- **Gota**: esta clase implementa el elemento gota, que es el motor fundamental para la resolución del problema por el método RFD. Las gotas se moverán a lo largo de la máquina de estados y para ello esta clase implementa todas las características necesarias para su funcionamiento, tales como la posición de las gotas, la gestión del movimiento, la erosión y la sedimentación.

Clases de Ant Colony Optimization (ACO):

- **AlgoritmoHormigas**: clase que implementa un algoritmo que se basa en el comportamiento de las colonias de hormigas en el trazo de caminos para buscar comida y volver a su hogar. Utilizando técnicas basadas en los rastros de feromonas, intenta hallar el mejor camino posible para obtener soluciones eficientes en el proceso de testing.

    - Algoritmos:

        - *Constructor*: inicializa los elementos necesarios permitiendo que el usuario elija parámetros específicos del problema.

        - *especializarGrafo*, *especializaAristas*, e inicializarAlgoritmo: crea, inicializa y especifica las estructuras necesarias y las adapta al problema de hormigas (ver clase AristaHormiga).

        - *ejecutaAlgoritmo*: algoritmo que realiza todas las funciones para resolver el problema. En su interior se encuentra el bucle que realiza tantas iteraciones del problema como el usuario seleccione en la interfaz. Su estructura es la siguiente:
            1. Llamada a la función moverHormigas (ver más adelante), la cual lleva casi todo el peso del algoritmo.
            2. Calcular y comparar la solución.

        - *moverHormigas*: desplaza a las hormigas por los nodos calculando para cada una de ellas su nodo de destino. Tras el movimiento, llama las funciones resetHormigas, evaporarFeromonas y resetHormigasEnDestino (ver a continuación) realizando así el ciclo completo del problema.

        - *evaporarFeromonas*: reduce la cantidad de feromonas que hay depositadas en las aristas del grafo en un porcentaje determinado que el usuario puede variar a su elección.

        - *resetHormigasEnDestino*: cuando una hormiga llega al nodo destino definido, si ya a cubierto todos los nodos críticos deposita feromonas, y se crea otra que cae aleatoriamente sobre alguno de los nodos críticos. En caso de no cubrir todos los nodos críticos se lanza por otro nodo critico de manera aleatoria.

        - *depositarFeromonas*: método que aumenta la cantidad de feromonas que posee una solución por la cual una hormiga acaba de

pasar. La cantidad de feromonas depositada es variable y dependiente de la relación entre el coste de la solución y su coste máximo.

- *crearSolucionParcial*: crea una solución parcial al problema utilizando el mapa de feromonas que crean las hormigas.

- **AristaHormigas**: clase que hereda de la clase Arista y especifica la misma para la resolución del problema con el método Hormigas, añadiendo la propiedad de las feromonas.

- **Hormiga**: esta clase implementa a una hormiga. Las hormigas son objetos que poseen una posición determinada (el nodo en el que se encuentran situadas actualmente) y una solución, que consiste en un árbol que se va formando a partir de los nodos que visita la hormiga para llegar a 0 desde cada uno de los nodos críticos.

Clases de Algoritmo Genético:

- **AlgoritmoGenetico**: basado en las teorías evolutivas de Darwin, esta clase implementa un algoritmo toma soluciones aleatorias como individuos de una población e implementa métodos que permiten reproducirse a estos individuos generando otros nuevos que combinen características de ambos progenitores, métodos que hacen posible simular mutaciones en estos individuos, métodos que evalúan a cada individuo en función de su calidad como solución (en este caso concreto, por su mejor coste) y métodos que permiten seleccionar a los mejores individuos para repetir el proceso en sucesivas generaciones.

    - Algoritmos:

        - *Constructor*: inicializa el algoritmo tomando los parámetros que el usuario le pasa desde la interfaz.
        - *calculoDeFitness:* recorre toda la población llamando a la función de calcular fitness de cada individuo.
        - *poblacionInicial*: inicializa la población inicial de la que se va a partir como primera generación de individuos.
        - *seleccionRuleta*: realiza una selección de ruleta sobre toda la población tal y como esta descrita en el punto 4.2.
        - *seleccionTorneo*: realiza una selección de torneo determinista sobre toda la población tal y como esta descrita en el punto 4.2.
        - *seleccionEstocastica*: realiza una selección estocástica sobre toda la población tal y como esta descrita en el punto 4.2.
        - *mutacionPorTransformacion*: transforma uno de los valores del árbol pasado por parámetro en otro valor aleatorio entre 1 y el tamaño del grafo.
        - *valorValidoTransformacion*: función utilizada por *mutacionPorTransformacion* para comprobar si el nuevo valor aleatorio es válido.
        - *mutacionPorEliminacion*: transforma el árbol que se le pasa por parámetro eliminando uno de sus nodos y posteriormente adjuntando los hijos del nodo eliminado al padre de este.
        - *crucePorNodoCritico*: cruza dos árboles que se le pasan por parámetro poniendo la rama en la que se encuentra el nodo critico pasado como argumento en uno de los árboles en el otro y viceversa.
        - *genraIndividuoAleatorio*: genera un árbol que cubre todos los nodos críticos de manera aleatoria.

- *ejecutaAlgoritmo*: gestiona todos los elementos implicados en la resolución del problema. Se estructura de la siguiente forma:
    1. Gestión del elitismo (en caso de que se haya seleccionado).
    2. Se realiza el cruce de los individuos.
    3. Se gestiona la mutación de los individuos.
    4. Cálculo del fitness.
    5. Actualización de la población (en caso de que haya).
    6. Selección de los mejores individuos.

- **Individuo**: implementa la estructura de datos necesaria para resolver el problema con el método Genético. En esta ocasión no se trata de objetos que avanzan por la máquina de estados para formar un camino, como ocurre con los métodos RFD y Hormigas, sino que representan el camino en sí. Los individuos poseen un genoma que es la representación de un camino, es decir, de una solución al problema y un fitness, que representa el valor de dicha solución.

    - Algoritmos:

        - *calcularFitness*: este método calcula la calidad del individuo en relación al coste que supone como solución para el problema. Analiza su genoma y devuelve su coste total.

6. ANÁLISIS DE RESULTADOS

En esta sección se va a estudiar el comportamiento de los algoritmos que se han adaptado al problema objeto de estudio, MLS. El estudio del comportamiento se va a dividir en dos partes: obtención de la configuración óptima de los algoritmos, y comparación de los algoritmos con la configuración óptima encontrada.

Para la obtención de la configuración óptima se utilizará un problema de entrenamiento igual para todos los algoritmos:

Problema de entrenamiento con alta densidad

La máquina de estados constará de 100 estados, y cada nodo tendrá un 70% de conexiones con el resto de nodos (será un grafo muy denso). Los costes de realizar una transición de un estado a otro se generan aleatoriamente entre 2 y 100. La generación del grafo se realiza de manera aleatoria con la densidad de aristas y el coste entre transiciones.

El número de nodos críticos se establece en 20. Los nodos que se quieren visitar (elegidos al azar) son:

1,3,7,9,12,14,18,24,27,32,35,46,48,55,69,75,77,83,92,94.

Para este problema de entrenamiento, todas las metaheurísticas se enfrentarán a él en dos escenarios distintos: un escenario que favorece las cargas, con coste de carga 1, y un escenario en el que se penalizarán las cargas, con un coste de carga de 500.

Problema de entrenamiento con baja densidad

El segundo problema de entrenamiento que se va a utilizar tendrá 100 estados, con unas conexiones entre nodos del 30%, convirtiéndolo en un grafo poco denso. Los costes de realizar una transición tendrán valores entre 2 y 500. La generación del grafo se realiza de forma aleatoria con la densidad de aristas y el coste entre transiciones.

La entrada será de un total de 20 nodos críticos a visitar, elegidos aleatoriamente:

1,7,9,14,18,24,28,30,35,40,47,53,58,61,62,75,83,89,93,98.

Para este problema problema, al igual que en el problema de entrenamiento denso, los algoritmos se enfrentarán a él favoreciendo las cargas, con coste de carga 1, y desfavoreciendo las cargas, coste de carga 500.

En la búsqueda de la configuración óptima de cada algoritmo, se realizarán diversas ejecuciones independientes con distintas configuraciones, de las que se extraerán conclusiones para llegar a una configuración competitiva.

6.1. Configuración de algoritmos

GA

Los parámetros a determinar algoritmo genético que tiene la aplicación desarrollada son los siguientes:

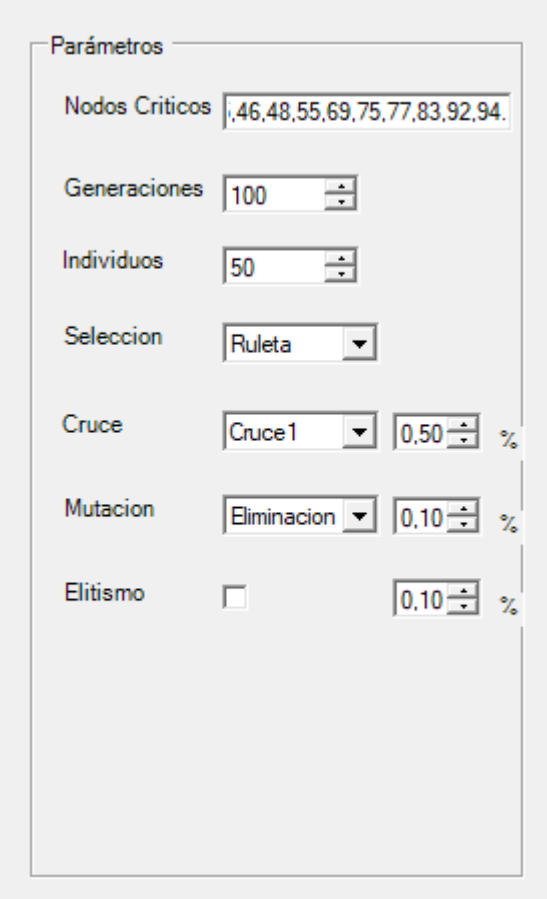

Figura 6.1 - Panel de parámetros para GA

Generaciones: Equivalentes al número de iteraciones en los otros algoritmos. Se irá aumentando en las distintas pruebas para intentar mejorar los resultados del algoritmo hasta hallar el valor aproximado de estancamiento. Las generaciones van directamente relacionadas con la calidad de la solución encontrada, ya que es lo que permite la variación de los individuos a través de los métodos de cruce y mutación.

Individuos: Número de individuos que generan una solución aleatoria, que irán mejorando a lo largo de las iteraciones. Influye en el espacio de búsqueda del algoritmo.

Selección: Forma en que los individuos son seleccionados para sobrevivir dependiendo de su calidad como solución al problema. Se probarán los tres métodos implementados (Ruleta, Estocástico y Torneo).

Cruce: Porcentaje de individuos que se van a reproducir en cada generación. Se utilizarán valores de uno, dos y tres cuartos de la población.

Mutación: Modificaciones pseudoaleatorias que sufren algunos individuos de la población. Se probará con los distintos métodos de mutación implementados (Eliminación y Transformación), así como con distintos valores de probabilidad de mutación.

Elitismo: Posibilidad de que los mejores individuos sobrevivan siempre. Se variará entre utilizar o no este recurso, así como con el tamaño de este grupo de individuos de élite.

Problema de entrenamiento con alta densidad – 70%:

El proceso de pruebas comenzará por investigar el comportamiento de algunos de los parámetros más básicos del algoritmo, tales como el número de generaciones, de individuos y el porcentaje de cruce. Se pretende hallar la combinación más óptima de estos elementos para pasar después a variar el resto de parámetros manteniendo la mejor configuración hallada para los primeros. Las baterías de pruebas se dividirán en dos apartados generales. Primero se probará el problema de entrenamiento en un ámbito donde el coste de carga será muy elevado (500). Después, aprovechando los datos obtenidos en las primeras fases del estudio para no repetir pruebas poco fructíferas, se volverán a lanzar pruebas para el problema de entrenamiento en un escenario donde el coste de carga sea muy pequeño (1).

Escenario con coste de carga 500

En primer lugar, como se comentó anteriormente, se comenzará por investigar el comportamiento de algunos de los parámetros más básicos del algoritmo, tales como el número de generaciones, de individuos y el porcentaje de cruce. Para todas las pruebas relatadas sucesivamente, hasta que se especifique lo contrario más adelante, el mecanismo de selección será Ruleta debido a que su carácter más aleatorio que los otros dos tipos de selección, que son más dirigidos, será utilizado como una forma de aislamiento de este parámetro en favor de estudiar, de forma más específica, los parámetros citados anteriormente.

Aislando el resto de parámetros (100 individuos, 100 generaciones y sin mutación ni elitismo) y modificando el cruce en porcentajes de 25%, 50%, 75% y 100% se observan los siguientes resultados:

- 25%: Mejor coste: 915; Media: 1103; Desviación Típica: 95
- 50%: Mejor coste: 802; Media: 1118; Desviación Típica: 100,8
- 75%: Mejor coste: 767; Media: 1126,14; Desviación Típica: 94,8
- 100%: Mejor coste: 846; Media: 1110,34; Desviación Típica: 106,1

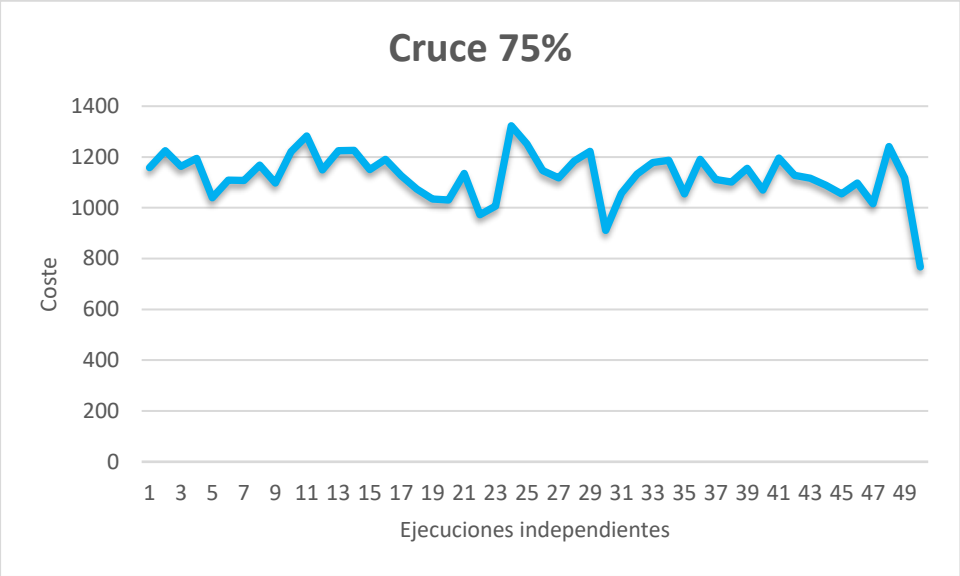

Figura 6.2 – 75%: Mejor coste: 767; Media: 1126,14; Desviación Típica: 94,8

La gráfica presentada es la de la configuración de cruce que mejor resultado ha obtenido en general, no solo por obtener el mejor coste, si no por tener la mejor relación media obtenida en 50 ejecuciones y su desviación.

Como se puede observar el progreso varía entre los distintos porcentajes de cruce, pero no de forma lineal. Aunque el mejor resultado obtenido varía considerablemente, atendiendo a la media de los resultados obtenidos en todas las ejecuciones así como en las desviaciones, no se puede aseverar una progresión determinante. Cómo los resultados observados no son determinantes, en las siguientes pruebas se van a modificar las generaciones y la población de individuos para ver si se obtienen datos más concluyentes.

En la siguiente batería de pruebas va a seguirse atacando el parámetro de cruce de la misma manera (25%, 50%, 75% y 100%), pero aumentando el número de individuos a 300 y aislando el resto de parámetros (300 individuos, 100 generaciones y sin mutación ni elitismo):

- 25%: Mejor coste: 848; Media: 1100,4; Desviación Típica: 115,1
- 50%: Mejor coste: 794; Media: 1051,16; Desviación Típica: 83,19
- 75%: Mejor coste: 836; Media: 1016,4; Desviación Típica: 85,6
- 100%: Mejor coste: 682; Media: 1018,1; Desviación Típica: 94,2

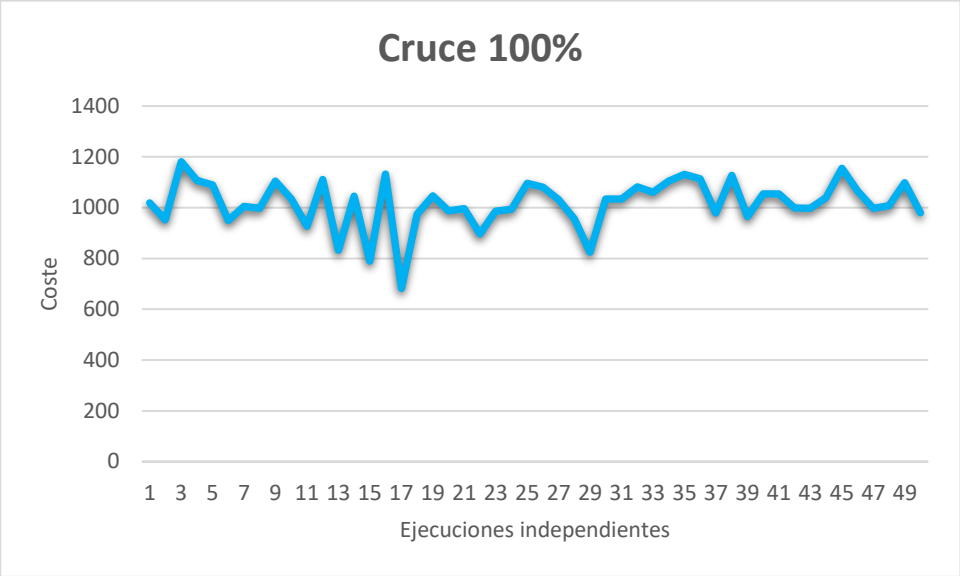

Figura 6.3 – 100%: Mejor coste: 682; Media: 1018,1; Desviación Típica: 94,2

Como la gráfica muestra, los valores obtenidos -en general- están por debajo de coste 1000, mientras que los resultados obtenidos en la anterior prueba, la mayoría se situaban por encima de este valor.

Después de ejecutar las mismas pruebas con un número mayor de individuos, se observa una mejora estable respecto a la anterior batería de pruebas, no solo respecto a la solución de mejor coste sino también una mayor robustez en cuanto a media de resultados y una menor desviación cuanto mayor porcentaje de cruce se aplica. Sin embargo, cabe señalar que cuando el cruce es el máximo (100%) se observa una pequeña regresión respecto a la Desviación Típica observada en los porcentajes 50% y 75% a la que se deberá estar atento en próximas pruebas para determinar si es casual o un comportamiento recurrente, con el objetivo de aportar información acerca del mejor valor de cruce posible.

Para el siguiente set de pruebas se va a seguir modificando, como hasta ahora, el porcentaje de cruce pero se va a aumentar aún más el número de individuos (500 individuos, 100 generaciones y sin mutación ni elitismo) con el objetivo de estudiar si la progresión hallada en el anterior estudio se mantiene aumentando este factor.

- 25%: Mejor coste: 798; Media: 1057,5; Desviación Típica: 119,19
- 50%: Mejor coste: 773; Media: 1021,2; Desviación Típica: 84,45
- 75%: Mejor coste: 780; Media: 1006,74; Desviación Típica: 68,69
- 100%: Mejor coste: 775; Media: 992,8; Desviación Típica: 77,28

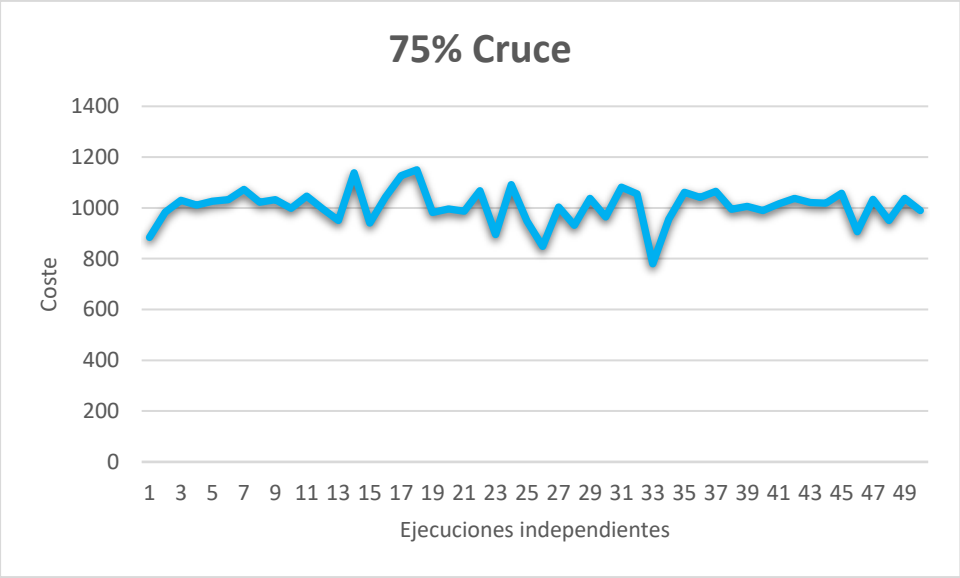

Figura 6.4 - 75%: Mejor coste: 780; Media: 1006,74; Desviación Típica: 68,69

La gráfica refleja el conjunto de ejecuciones más estable de las combinaciones de cruce realizadas en el set de pruebas. Cabe destacar que aunque los mejores resultados obtenidos son muy similares en las cuatro variaciones, la gráfica presenta una desviación considerablemente menor lo que la hace más relevante.

Tras aumentar el número de individuos considerablemente (hasta 500 individuos), pero dejando en un nivel bajo el número de generaciones, se continúan observando mejores resultados con cruce alto, entre 75 y 100. Sin embargo, en línea con lo postulado tras el set de pruebas anterior, se sigue observando cierta regresión en la desviación típica cuando el cruce asciende al 100%. En ocasiones puede reportar un mejor coste absoluto, pero con una menor robustez que se refleja en una media similar o, incluso, inferior a cruces ligeramente menores y una desviación típica algo más acusada. Por ello, teniendo en cuenta todo el conjunto de pruebas ejecutado hasta el momento, se establecerá el porcentaje de cruce en un valor entre 75% y 100%, pero que, considerando los datos obtenidos, no sea el máximo. El porcentaje se ajustará en un 85% a partir de ahora.

A continuación, se modificarán combinadamente los parámetros de generaciones e individuos. Como ya se señaló anteriormente, aumentar el número de individuos aportó mejoras interesantes. Sin embargo, para no descartar la posibilidad de que el mayor peso descanse de manera más vinculante con las generaciones en lugar de los individuos, se realizarán pruebas modificando ambos valores y aislando, como hasta ahora, el resto (250-500 Generaciones, 100-500 Individuos y sin Mutación ni Elitismo).

- 250 Generaciones - 100 Individuos: Mejor coste: 876; Media: 1052,14; Desviación Típica: 67,97
- 250 Generaciones - 300 Individuos: Mejor coste: 841; Media: 971,26; Desviación Típica: 63,74
- 250 Generaciones - 500 Individuos: Mejor coste: 690; Media: 937,52; Desviación Típica: 73,9

- 500 Generaciones - 500 Individuos: Mejor coste: 656; Media: 894,9; Desviación Típica: 71,19

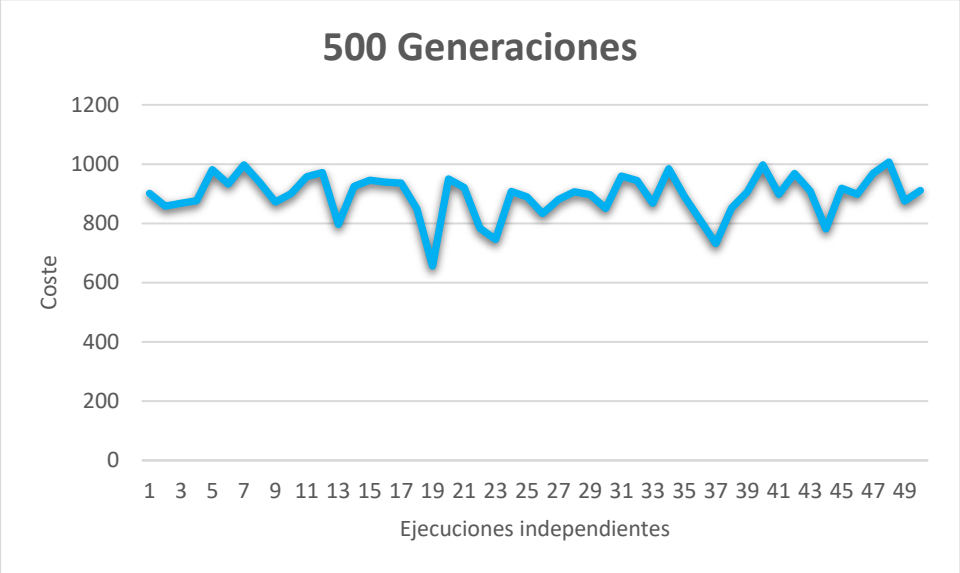

Figura 6.5 -500 Generaciones - 500 Individuos: Mejor coste: 656; Media: 894,9; Desviación Típica: 71,19

Los resultados que se presentan en la gráfica demuestran que la configuración de 500 generaciones – 500 Individuos es más consistente que el resto de configuraciones anteriores y las probadas en el mismo set.

Los cambios observados en los resultados mejoran respecto a las baterías de pruebas anteriores, especialmente atendiendo a la media y a la desviación típica, lo cual implica una mayor robustez en las presentes configuraciones.

Se observa una mejora importante, sobre todo aumentando los individuos dentro de un número de generaciones medio o alto (entre 250-500). Sin embargo, aunque se obtiene una mejora teniendo más generaciones, se observa una mejora más significativa cuando son los individuos los que aumentan. Debido a ello, se va a mantener un número de generaciones medio, aumentándolas ligeramente (a 300) pero manteniendo una tasa alta de individuos (500) para las siguientes pruebas, donde se va a experimentar con los distintos tipos de selección que no se han probado aún (Estocástico y Torneo) y aislando los parámetros restantes. Es decir: Generaciones 300, Individuos 500, Cruce 85%, y Mutación y Elitismo desactivados.

- Selección Estocástico: Mejor coste: 818; Media: 1286,3; Desviación Típica: 136,44
- Selección Torneo: Mejor coste: 705; Media: 1116,46; Desviación Típica: 131,17

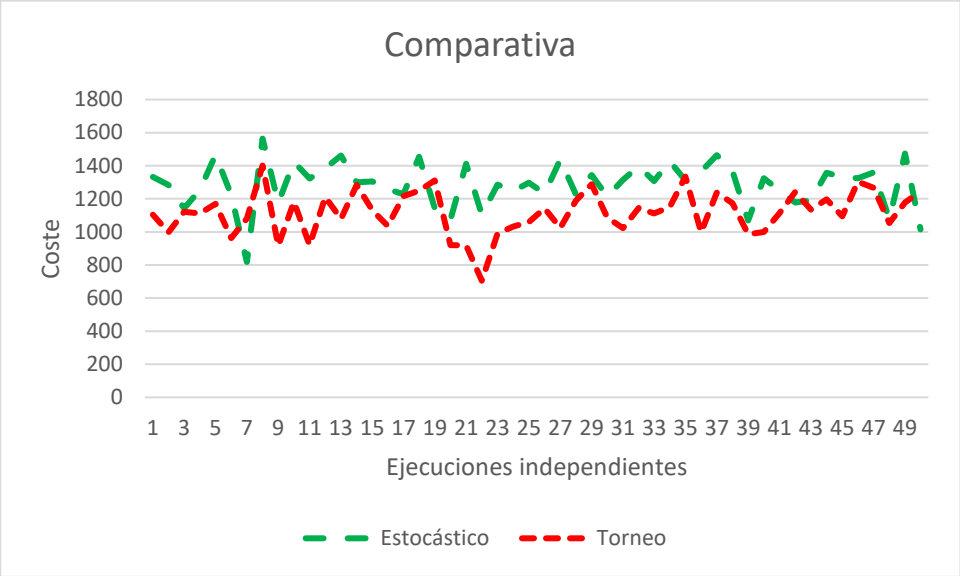

Figura 6.6 – Comparación entre Estocástico y Torneo

Tras las pruebas efectuadas, se observa que la selección de tipo Torneo se muestra más efectiva que la de tipo Estocástico. Sin embargo, los resultados de ninguna de las nuevas dos selecciones son suficientemente distintos de Ruleta como para descartar ninguna de ellas.

A continuación, se introduce el factor de mutación en las pruebas. Se llevará a cabo con un nivel bajo (10%) ya que es un elemento que, si bien puede introducir novedades en los individuos que los haga mejorar mucho y rompa posibles estancamientos, es un factor que en un nivel alto puede resultar desestabilizador. Se comenzará evaluando el comportamiento con la mutación de tipo eliminación. Debido a que el anterior set de pruebas no ha mostrado una diferencia clara entre las selecciones, se va a probar la mutación para cada uno de los tipos de Selección (300 Generaciones, 500 Individuos, Cruce 85%, sin Elitismo).

- Selección Ruleta: Mejor coste: 685; Media: 997,98; Desviación Típica: 106,71
- Selección Estocástica: Mejor coste: 201; Media: 454,28; Desviación Típica: 255,46
- Selección Torneo: Mejor coste: 194; Media: 235,98; Desviación Típica: 15,39

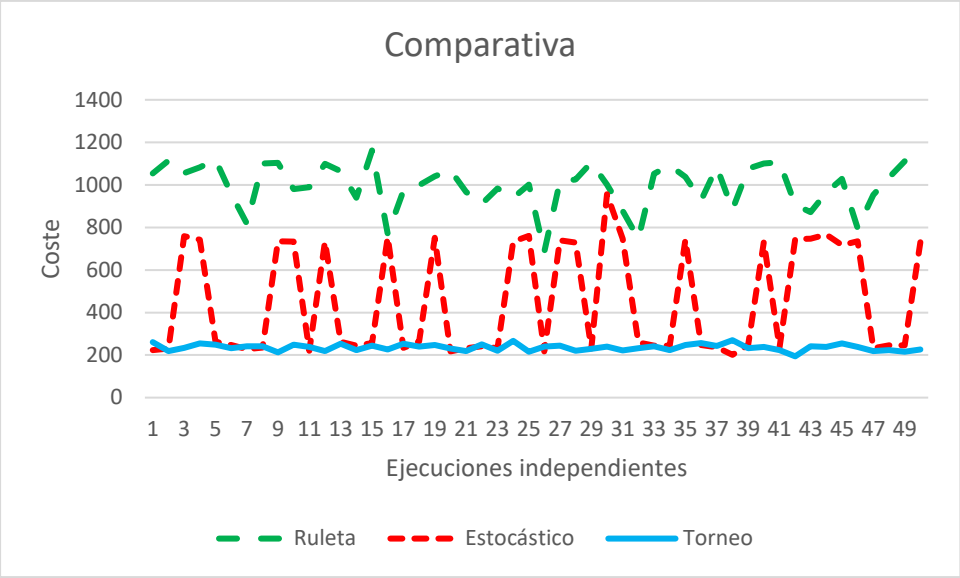

Figura 6.7 – Comparativa entre los métodos de selección al habilitar mutación por eliminación

Se observa una mejora más que considerable incorporando el factor de mutación. La mejora se da en todos los niveles: la mejor solución encontrada no solo es mucho mejor que todas las anteriores (mejores costes entre 200 y 300 frente a la horquilla de 700-800 obtenida en las mejores ejecuciones sin mutación) sino que además en todas las ejecuciones efectuadas se obtiene mayor robustez, con una media mucho más ajustada y una desviación ínfima. La otra conclusión clave que se puede observar es que dicha mejora se produce solo cuando la mutación se acompaña de los tipos de Selección Estocástica y Torneo. Cuando se utiliza Ruleta los resultados apenas varían respecto de las pruebas anteriores que, como ya se ha señalado anteriormente, son mucho peores. Por ello, se puede concluir no solo el factor de mutación se muestra fundamental para mejorar la configuración del algoritmo, sino que, además, en conjunción con este, las selecciones Estocástica y Torneo se muestran muy superiores a Ruleta, por lo que se va a descartar esta última en futuras pruebas. No obstante, la Selección de tipo Estocástico, a pesar de que es mucho más efectiva que la de tipo Ruleta, presenta una desviación muy acusada que la hace mucho menos robusta que la Selección de tipo Torneo.

Para el siguiente set de pruebas se va a probar la Mutación de tipo Transformación para analizar las diferencias que se obtienen respecto de las anteriores pruebas utilizando el tipo Eliminación. Para las próximas ejecuciones se va a utilizar la configuración más óptima de las anteriores, que fue utilizando la Selección de tipo Torneo (300 Generaciones, 500 Individuos, Selección Torneo, Cruce 85% sin Elitismo).

- Mutación Transformación: Mejor coste: 815; Media: 1120,7; Desviación Típica: 132,42

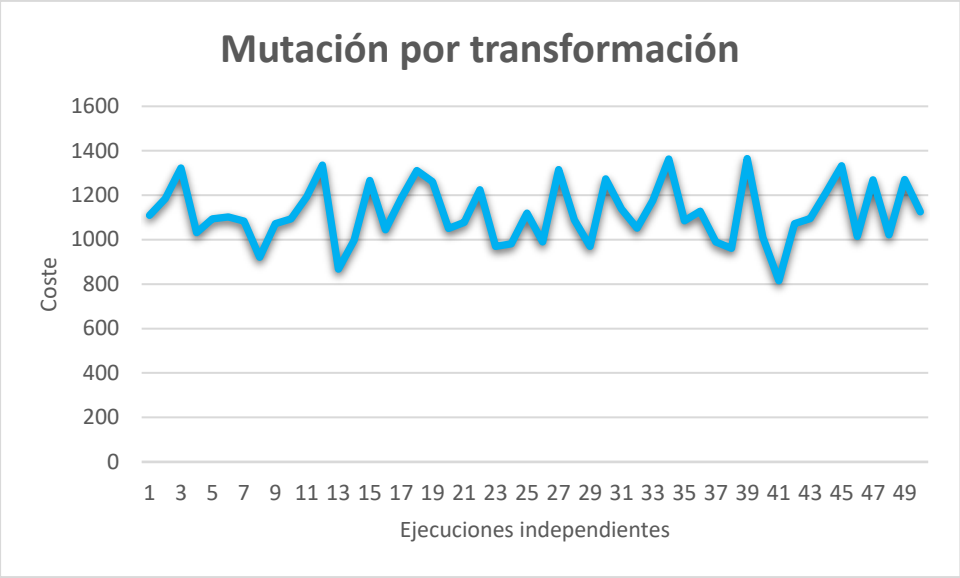

Figura 6.8 – Mutación Transformación: Mejor coste: 815; Media: 1120,7; Desviación Típica: 132,42

La mutación por transformación, sin embargo, muestra un desempeño muy deficiente en comparación con la mutación por eliminación, compitiendo en las mismas condiciones con la primera. De hecho, arroja unos resultados similares, e incluso peores a algunas de las pruebas realizadas anteriormente sin introducir el factor de mutación, por lo que será descartada de cara a futuras pruebas, manteniendo la mutación por eliminación al 10%, debido a la calidad de sus resultados.

A continuación, se va a incluir el factor elitismo. Esta característica asegura la supervivencia de los mejores individuos de cada generación. La cantidad depende del porcentaje seleccionado, el cual es recomendable que oscile en unos valores bajos, para no corromper el concepto de élite. Por ello, se designará un valor de un 10% de élite de la población, manteniendo el resto de parámetros con los valores que han mostrado ser más efectivos hasta ahora (300 Generaciones, 500 Individuos, Cruce 85%, Mutación Eliminación 10%).

- Elitismo: Mejor coste: 194; Media 239,58; Desviación Típica: 14,92

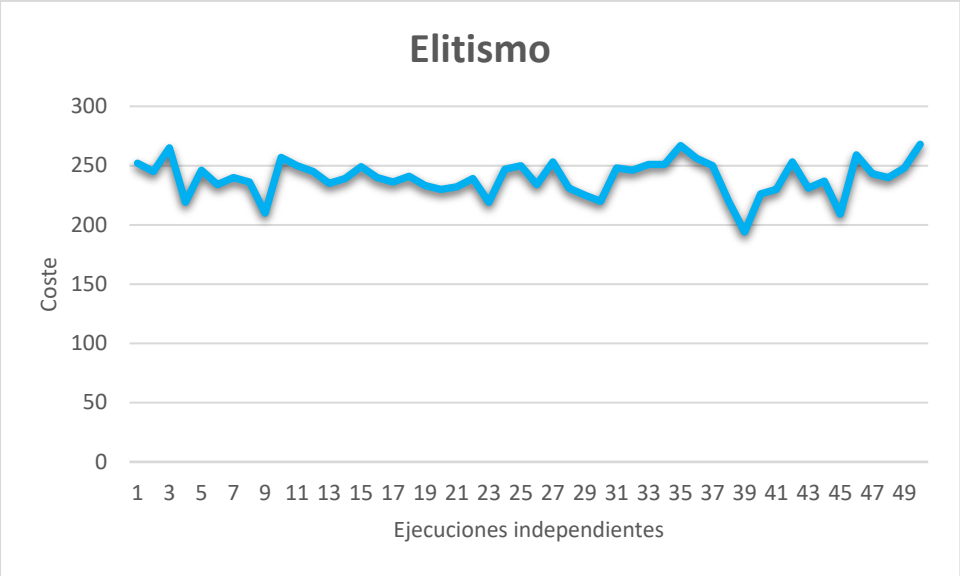

Figura 6.9 – Elitismo: Mejor coste: 194; Media 239,58; Desviación Típica: 14,92

L os resultados aplicando elitismo son muy similares a los obtenidos anteriormente con las mismas condiciones sin elitismo, pero con una ligera mejora en la desviación. En la siguiente sección de pruebas (pruebas en el problema de entrenamiento en un escenario donde siempre se carga) se volverá a ejecutar la batería de pruebas activando y desactivando este factor para comprobar si resulta más significativo que en las pruebas actuales.

Escenario con coste de carga 1

Se comenzarán las pruebas manteniendo aquellos parámetros que se han mostrado claramente más efectivos en las pruebas anteriores (escenario donde nunca se realizan cargas) tales como el número de individuos, el número de generaciones y la mutación por eliminación al 10%, y se modificarán aquellos en los que la variación no ha sido suficientemente significativa para ser descartados. En primer lugar, se realizarán pruebas con los dos métodos de selección que se mostraron más útiles en las anteriores pruebas (Estocástica y Torneo) y cuyas diferencias resultaron poco relevantes anteriormente con el objetivo de comprobar si en un entorno en el que se favorezca la carga se producen diferencias significativas (300 Generaciones, 500 Individuos, Selección Estocástica, Cruce 85%, Mutación Eliminación 10% sin Elitismo).

- Selección Estocástica: Mejor coste: 217; Media 243,38; Desviación Típica: 15,41
- Selección Torneo: Mejor coste: 212; Media 242,4; Desviación Típica: 16,11

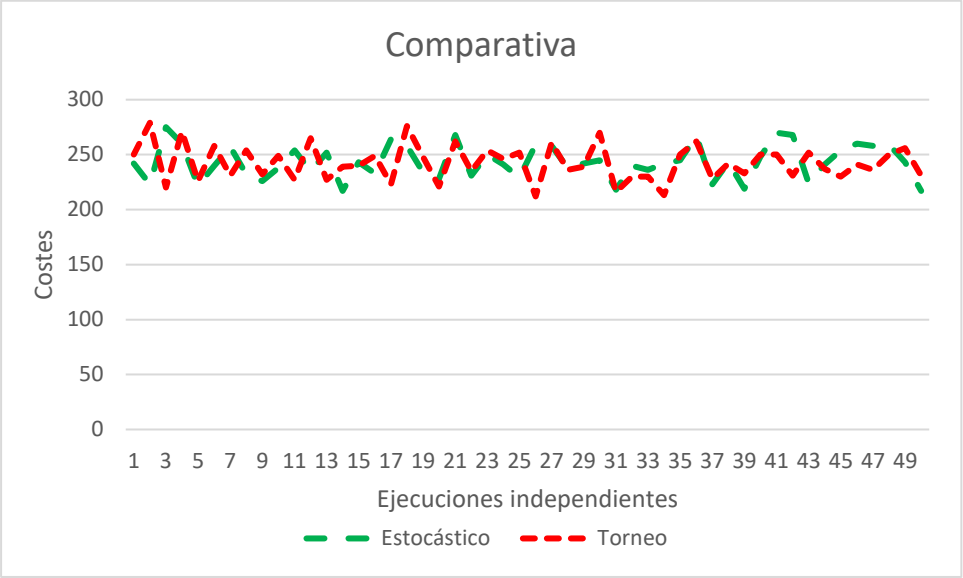

Figura 6.10 – Comparativa entre Estocástico y Torneo quitando elitismo

Ambos tipos de selección muestran un desempeño similar, por lo que se repetirán de nuevo las pruebas con ellos introduciendo el factor de elitismo, para comprobar cómo se comporta en un entorno en el que se favorece la carga (300 Generaciones, 500 Individuos, Selección Estocástica, Cruce 85%, Mutación Eliminación 10% con Elitismo).

- Selección Estocástica: Mejor coste: 205; Media 237,08; Desviación Típica: 13,99
- Selección Torneo: Mejor coste: 202; Media 236,34; Desviación Típica: 15,38

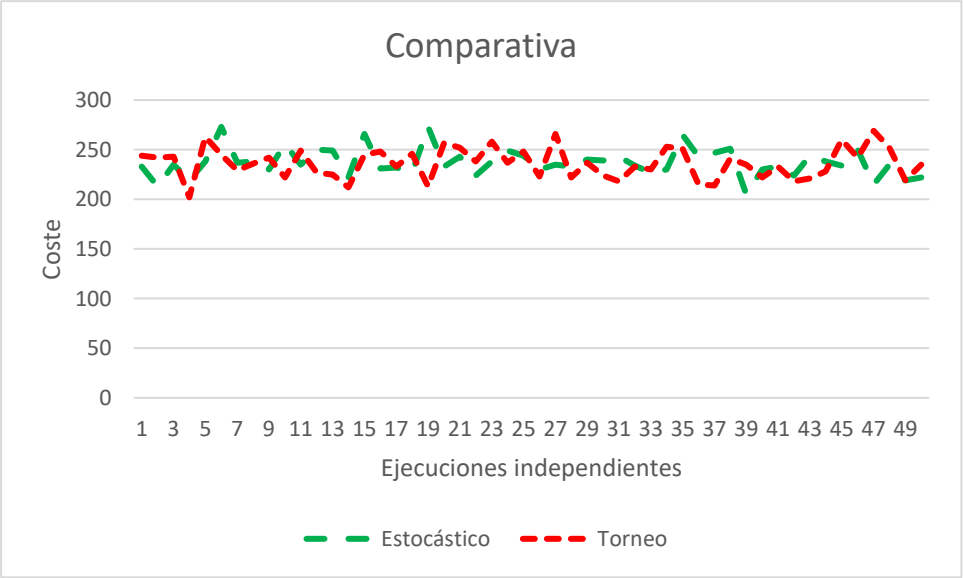

Figura 6.11 – Comparativa entre Estocástico y Torneo con elitismo activo (10%)

El elitismo presenta una mejora con ambos métodos de selección respecto a las mismas pruebas sin elitismo. La mejora no es muy sustancial pero sí lo suficiente como para tenerla en cuenta, ya que no solo encuentra mejores costes, sino que también presenta medias mejores en el conjunto de las ejecuciones y una menor desviación típica.

Respecto a los otros dos parámetros tenidos en cuenta en las últimas pruebas (la Selección de tipo Estocástico y Torneo), los resultados son altamente similares, pero Torneo se ha comportado ligeramente mejor a la hora de encontrar la mejor solución y obtener la mejor media, aunque con una Desviación Típica mínimamente inferior.

Tabla recopilatoria de resultados

A continuación, se presenta una tabla resumen de los resultados de las pruebas realizadas para este problema de entrenamiento:

Ejs	Generaciones	Individuos	Selección	Cruce	Mutación	Elitismo	Nodos	Densidad	C. Carga	Mejor	Media	Desviación
50	100	100	Ruleta	25%	No	No	100	70%	500	915	1103,00	95,00
50	100	100	Ruleta	50%	No	No	100	70%	500	802	1118,00	100,80
50	100	100	Ruleta	75%	No	No	100	70%	500	767	1126,14	94,80
50	100	100	Ruleta	100%	No	No	100	70%	500	846	1110,34	106,10
50	100	300	Ruleta	25%	No	No	100	70%	500	848	1100,40	115,10
50	100	300	Ruleta	50%	No	No	100	70%	500	794	1051,16	83,19
50	100	300	Ruleta	75%	No	No	100	70%	500	836	1016,40	85,60
50	100	300	Ruleta	100%	No	No	100	70%	500	682	1018,10	94,20
50	100	500	Ruleta	90%	No	No	100	70%	500	798	1057,50	119,19
50	100	500	Ruleta	70%	No	No	100	70%	500	773	1021,20	84,45
50	100	500	Ruleta	70%	No	No	100	70%	500	780	1006,74	68,69
50	100	500	Ruleta	70%	No	No	100	70%	500	775	992,80	77,28
50	250	100	Ruleta	85%	No	No	100	70%	500	876	1052,14	67,97
50	250	300	Ruleta	85%	No	No	100	70%	500	841	971,26	63,74
50	250	500	Ruleta	85%	No	No	100	70%	500	690	937,52	73,90
50	500	500	Ruleta	85%	No	No	100	70%	500	656	894,90	71,19
50	300	500	Estocástica	85%	No	No	100	70%	500	818	1286,30	136,44
50	300	500	Torneo	85%	No	No	100	70%	500	705	1116,46	131,17
50	300	500	Ruleta	85%	Eliminación 10%	No	100	70%	500	685	997,98	106,71
50	300	500	Estocástica	85%	Eliminación 10%	No	100	70%	500	201	454,28	255,46
50	300	500	Torneo	85%	Eliminación 10%	No	100	70%	500	194	235,98	15,39
50	300	500	Torneo	85%	Transformación 10%	No	100	70%	500	815	1120,70	132,42
50	300	500	Torneo	85%	Eliminación 10%	10%	100	70%	500	194	239,58	14,92
50	300	500	Estocástica	85%	Eliminación 10%	No	100	70%	1	217	243,38	15,41
50	300	500	Torneo	85%	Eliminación 10%	No	100	70%	1	212	235,98	16,11
50	300	500	Estocástica	85%	Eliminación 10%	10%	100	70%	1	205	237,08	13,99
50	300	500	Torneo	85%	Eliminación 10%	10%	100	70%	1	202	236,34	15,38

Figura 6.12 – Tabla resumen de las pruebas realizadas para el problema de entrenamiento con alta densidad

Conclusiones

Después de todas las baterías de pruebas efectuadas la mejor configuración hallada para el Algoritmo Genético, tanto en el escenario sin cargas, como en el ámbito con cargas es la siguiente:

- 300 generaciones
- 500 individuos
- Selección de tipo Torneo
- Cruce al 85%
- Mutación por Eliminación al 10%
- Elitismo al 10%

La mejor solución sobre el problema de entrenamiento en el escenario con coste de carga 500 es de 194, con una media en la batería de ejecuciones de 239,58 y una desviación típica de 14,92.

En el problema de entrenamiento con coste de carga 1, la mejor solución encontrada es de 202, con una Media de 236,34 y una Desviación Típica de 15,38.

Como se puede observar el algoritmo genético no aprovecha bien la funcionalidad de carga de nodos, esto es debido en gran medida a la estructura de cruce de individuos, que al no ser inteligente realiza cargas con caminos que, en la selección de los individuos para la siguiente generación no tienen una buena puntuación, puesto que no carga la subsecuencia más optima del camino, sino que carga el recorrido completo. Además de esto, la generación de individuos tampoco crea unas cargas favorables, dando lugar a soluciones que son menos rentables que las que realizan menos cargas.

Problema de entrenamiento con baja densidad - 30%:

Escenario con coste de carga 500:

Se volverá a iniciar el estudio con las pruebas penalizando la carga (Coste de Carga 1). Para la elaboración de este segundo estudio se tendrá en cuenta todo lo aprendido en el primero para ahorrar un número significativo de pruebas que no se mostraron nada concluyentes y que no mejoraron (o empeoraron) en absoluto los resultados. Sin embargo, no se descartarán otras que, si bien no supusieron una mejora suficientemente sustancial para mostrarse ganadoras con el anterior problema de entrenamiento, no puede descartarse su posible mejor efectividad en el presente problema de entrenamiento.

Uno de los factores clave que se descubrió en el anterior estudio es que la inclusión de la mutación es absolutamente fundamental para lograr unos resultados competentes. El enorme aumento de efectividad obtenido para las distintas configuraciones cuando se incluye el factor de mutación (concretamente la de tipo Eliminación, aunque no se descartará la de tipo Transformación más adelante) obliga a que sea un elemento fijo en la configuración de GA e imprescindible para todas las pruebas que se realizarán en adelante. Por tanto, será siempre utilizada en las distintas baterías que se ejecutarán sobre este segundo problema de entrenamiento.

El segundo factor clave que se va a provechar es el tipo de selección. Como se observó en el pasado, las selecciones de tipo Estocástico y Torneo son extremadamente superiores a Ruleta cuando se acompañan del factor mutación. La selección Ruleta es mucho peor que las citadas anteriormente tanto con mutación como sin ella, por lo que aunque utilizarla fue interesante para comenzar una primera investigación partiendo de cero, se concluye con total seguridad que, debido a su carácter netamente aleatorio, no supone una herramienta recomendable en la búsqueda de la mejor configuración. Por ello, en las sucesivas pruebas se utilizarán solo la selección Estocástica y Torneo.

En primer lugar, como se hizo anteriormente, se comenzará por investigar el comportamiento de algunos de los parámetros más básicos del algoritmo, tales como el número de generaciones, de individuos y el porcentaje de cruce. Aprovechando lo aprendido en el anterior estudio, se someterán a prueba dichos parámetros teniendo en cuenta las horquillas que mejores resultados arrojaron, con el objetivo de observar para este segundo problema de entrenamiento se encuentra algún cambio significativo en los mismos.

Se comenzará por el cruce, aislando el resto de parámetros. Sin embargo, se aprovecharán los datos obtenidos anteriormente para designar cantidades de individuos y generaciones más cercanas a valores que se han mostrado más efectivos. Es decir, se comenzará con cantidades ligeramente más altas que en el primer estudio y se irán variando más adelante (150 individuos, 200 generaciones, Selección Estocástica, Mutación Eliminación 10%, sin Elitismo) y se modificará el porcentaje de

cruce en 50%, 70% y 90%, cubriendo la mejor horquilla de resultados hallada en la misma fase del estudio anterior.

- 50%: Mejor coste: 388; Media: 459,52; Desviación Típica: 38,87
- 70%: Mejor coste: 365; Media: 424,74; Desviación Típica: 25,69
- 90%: Mejor coste: 369; Media: 461,66; Desviación Típica: 39,39

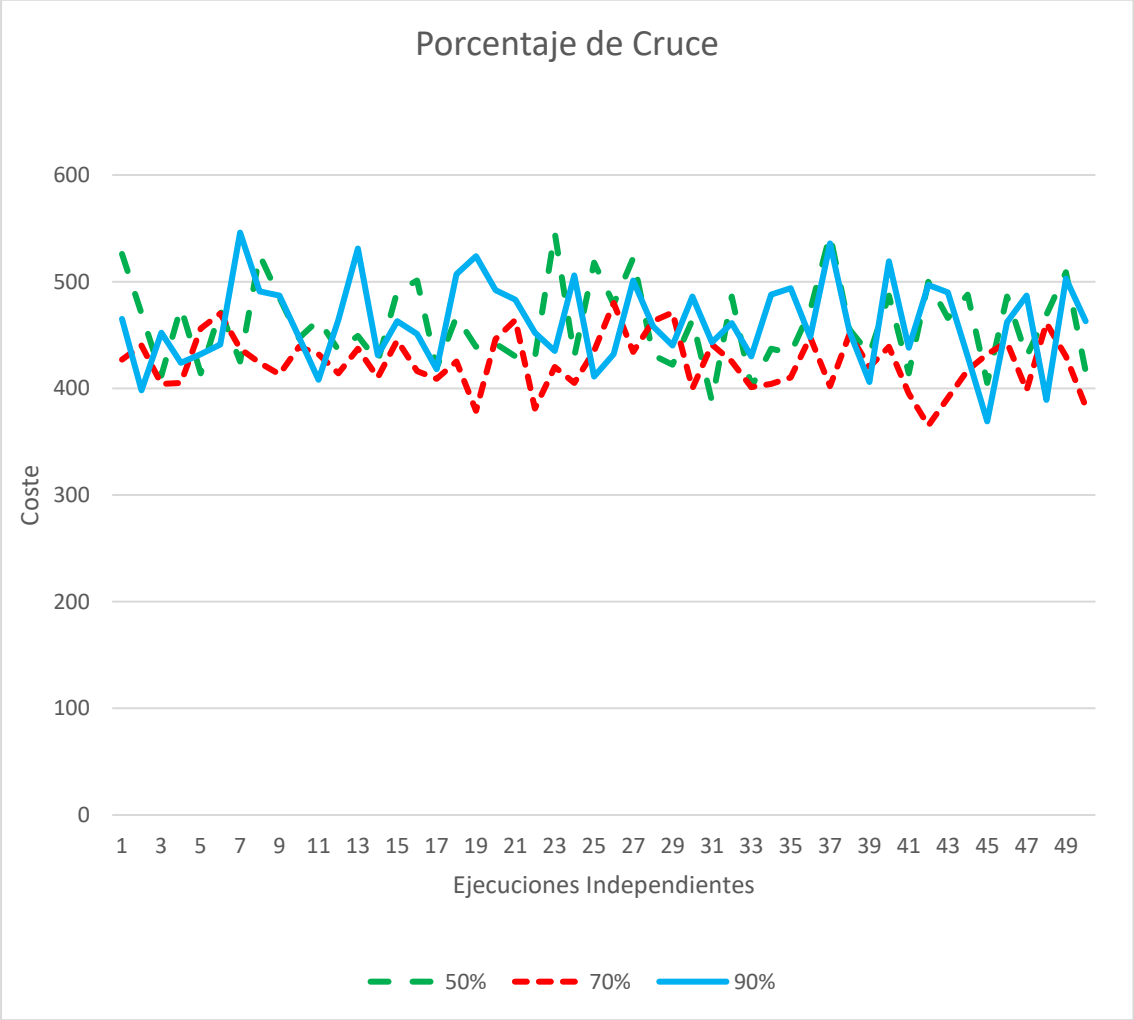

Figura 6.13 – Coste 50% vs Coste 70% vs Coste 90%

En línea con el primer estudio, puede observarse una mejora sustancial cuando el cruce aumenta. Sin embargo, una vez más, dicha mejora experimenta una regresión cuando los valores de cruce se aproximan demasiado al 100%. Anteriormente se estableció el valor óptimo en 85%, pero en esta ocasión, a la luz de los datos obtenidos, parece mostrarse más efectivo un valor ligeramente inferior. No obstante, se aprovecharán las próximas pruebas para observar si dichas conjeturas se mantienen.

En la siguiente batería de pruebas va a seguirse atacando el parámetro de cruce de la misma manera (50%, 70% y 90%), pero aumentando el número de individuos a 200 y las generaciones a 400, aislando el resto de parámetros (200 individuos, 400 generaciones, Selección Estocástica, Mutación Eliminación 10%, sin Elitismo):

- 50%: Mejor coste: 374; Media: 439,6; Desviación Típica: 27,32
- 70%: Mejor coste: 383; Media: 427,98; Desviación Típica: 25,07
- 90%: Mejor coste: 386; Media: 445,64; Desviación Típica: 25,22

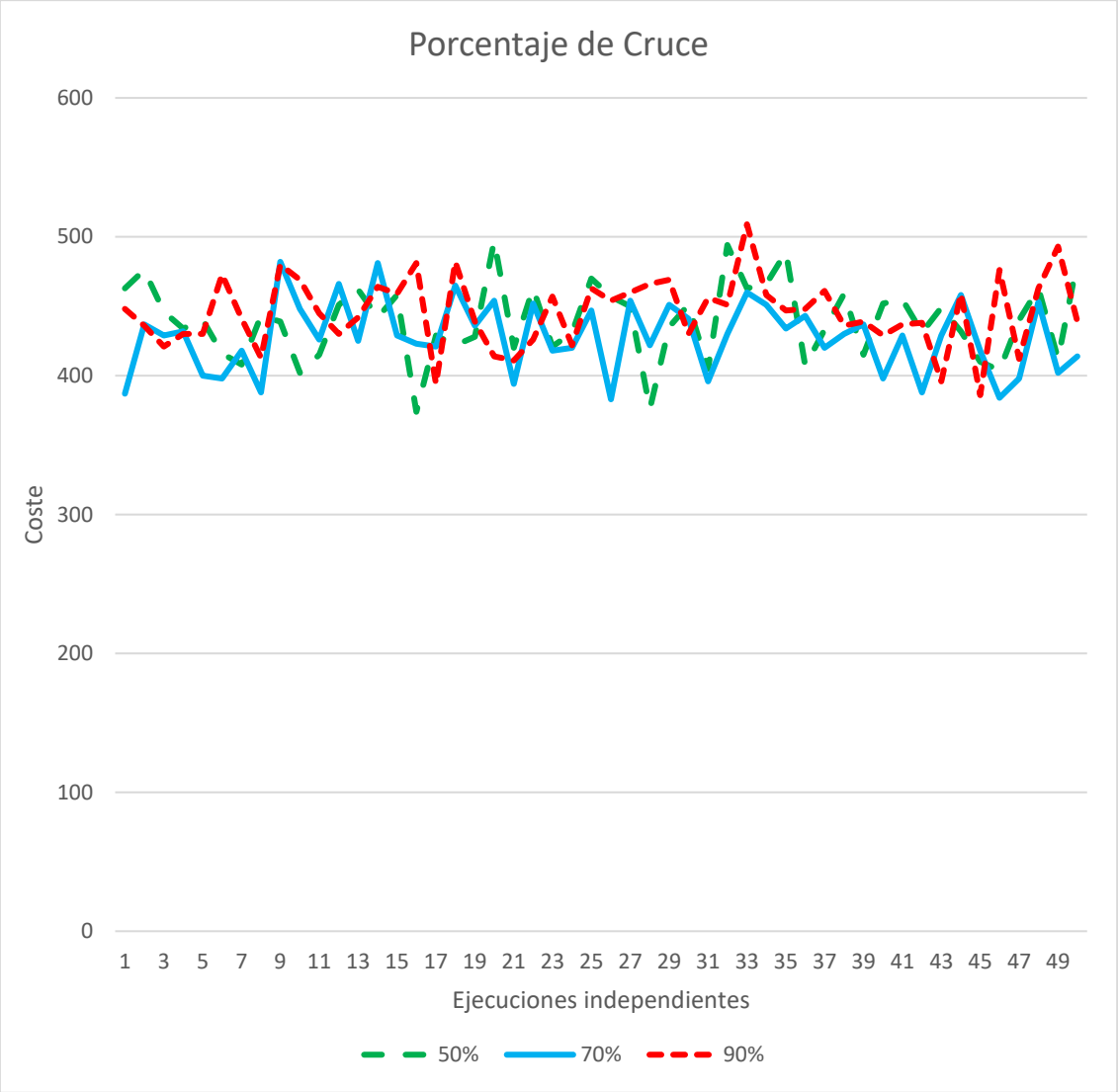

Figura 6.14 – Coste 50% vs Coste 70% vs Coste 90%

Tras las pruebas se observan unos resultados relativamente similares a los anteriores, si bien los mejores resultados obtenidos son prácticamente idénticos, hay una mejora en la desviación y en la media en la mayoría de pruebas que permiten afirmar una mayor estabilidad del algoritmo cuando tiene un mayor cantidad de población y de generaciones. Se sigue observando una mejor media en los resultados con un cruce de 70%, aunque esta vez el cruce de 90% arroja unos resultados más ajustados en media y desviación, por lo que se seguirá detenidamente en el siguiente set de pruebas.

En la siguiente batería de pruebas se va a seguir modificando, como hasta ahora, el porcentaje de cruce pero se va a aumentar aún más el número de individuos y generaciones, utilizando la configuración óptima de los mismos que se halló en el primer problema de entrenamiento (300 individuos, 500 generaciones, Selección

Estocástica, Mutación Eliminación 10%, sin Elitismo). El objetivo perseguido es doble: por un lado, observar si el aumento de los citados parámetros consigue una mejora suficientemente considerable en este problema como para mantenerlo y, por otro lado, tener más datos con los que decidir finalmente si el mejor porcentaje de cruce debería acercarse más al 70% o al 90%.

- 50%: Mejor coste: 392; Media: 436,74; Desviación Típica: 24,33
- 70%: Mejor coste: 355; Media: 413,34; Desviación Típica: 24,34
- 90%: Mejor coste: 365; Media: 426,26; Desviación Típica: 26,91

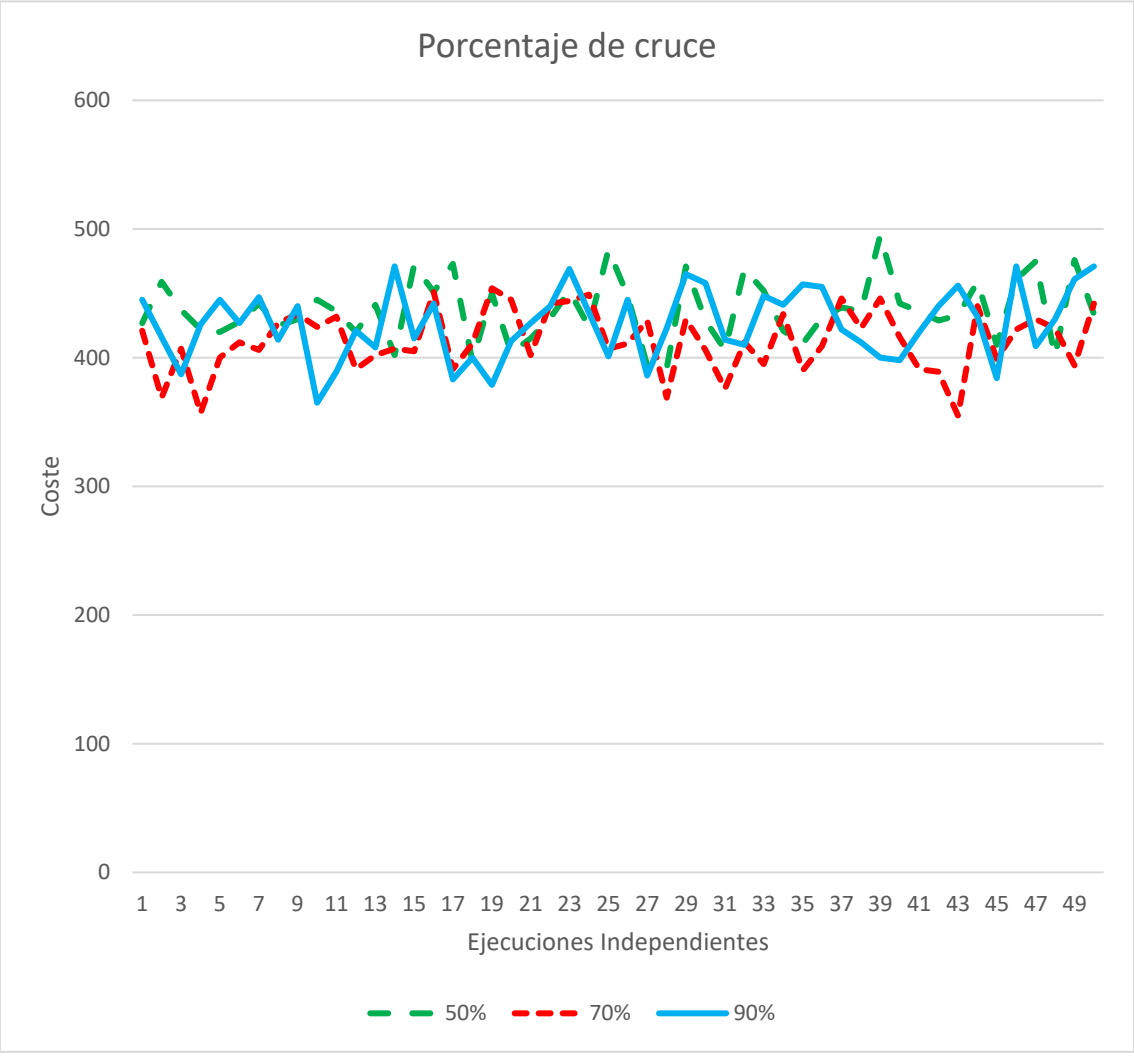

Figura 6.15 - Coste 50% vs Coste 70% vs Coste 90%

Tras el aumento del número de individuos y de generaciones se observa una mejora en los cruces de 70% y 90%. Especialmente en el de 70%, que mejora no solo en el mejor resultado hallado si no que se muestra más robusto al mejorar su media y su desviación respecto al planteamiento anterior con menos población y generaciones. Tal y como sucedía con el primer problema de entrenamiento, la mejor es suficiente como para tenerla en cuenta pero no como para aumentar más ya que el coste de tiempo de ejecución al aumentar estos dos parámetros aumenta linealmente mientras

que la mejora se va reduciendo paulatinamente (basta observar las pruebas realizadas hasta el momento). Por tanto, se mantendrán estos parámetros al mismo valor que en el primer problema de entrenamiento, sin aumentarlos más.
Respecto a la otra incógnita, vuelve a observarse que el valor de 70% de cruce se muestra más efectivo en general que el de 90%. Debido a que dicho comportamiento se mantiene en todas las pruebas realizadas hasta el momento, puede llegarse a la conclusión de que un valor de cruce más cercano a este es más efectivo. Por este motivo, se establecerá de forma fija el valor de cruce 70% para el resto de las pruebas.

En todas las anteriores pruebas se utilizó la selección Estocástica y la mutación por Eliminación. Sería interesante observar si hay algún tipo de sinergia entre los distintos tipos de selección y mutación, por lo que, con este objetivo, se van a realizar pruebas cruzadas entre ellos combinando ambos tipos entre sí.

- Selección Estocástica – Mutación Eliminación: ver resultados anteriores
- Selección Torneo – Mutación Transformación: Mejor coste: 1623; Media: 2169,2; Desviación Típica: 214,16
- Selección Estocástica – Mutación Transformación: Mejor coste: 1692; Media: 2335,02; Desviación Típica: 233,51
- Selección Torneo – Mutación Eliminación: Mejor coste: 379; Media: 442,64; Desviación Típica: 26,045

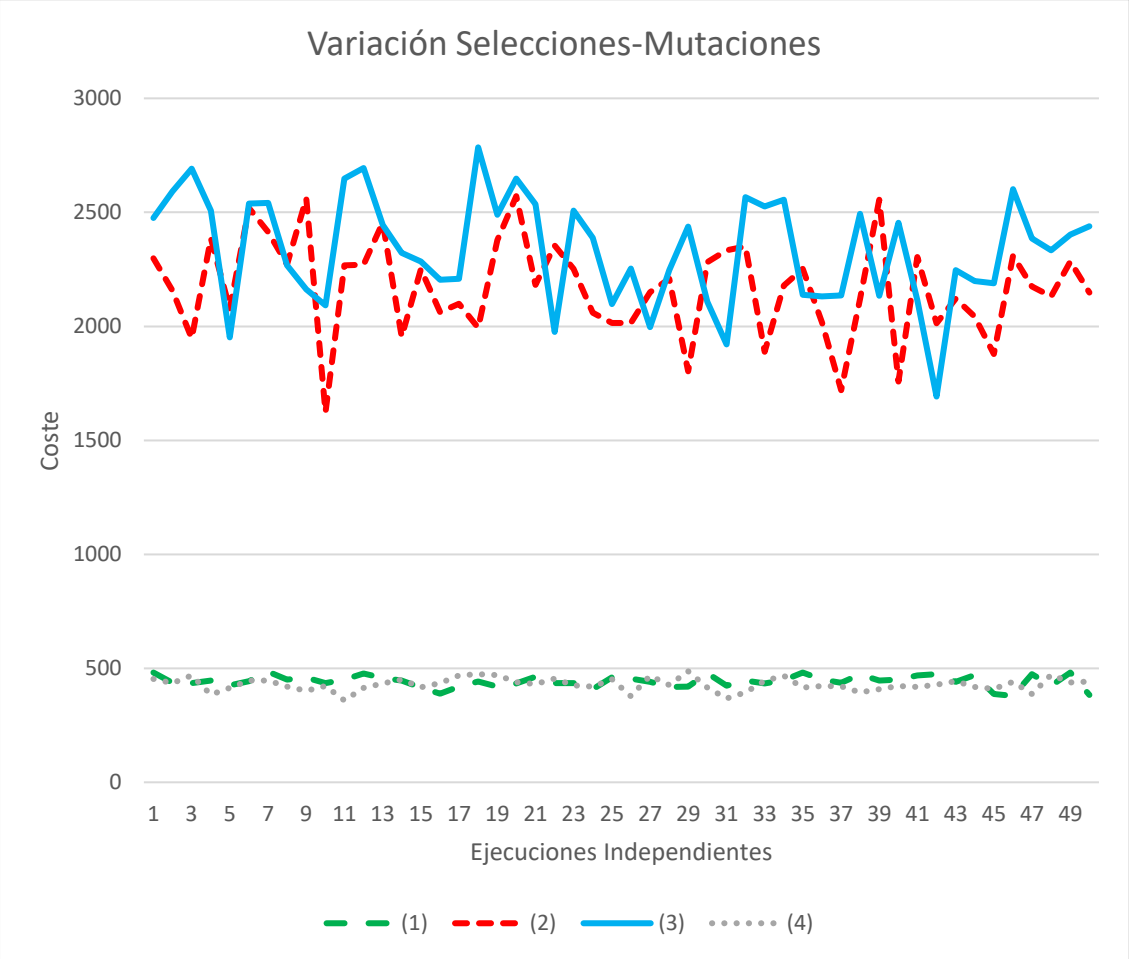

Figura 6.16 - Comparativa Selección Estocástica y Torneo - Mutación Transformación y Eliminación

Al margen del tipo de selección (Estocástica o Torneo), la mutación de tipo Transformación se muestra altamente ineficiente. Cabe destacar que los resultados son muy parecidos entre ambos tipos de selección cuando se usa la misma mutación para ambos. Respecto a la selección Torneo, arroja unos resultados prácticamente idénticos a Estocástica cuando se utiliza con mutación Eliminación, por lo que no se puede asegurar cual es mejor. Lo que sí puede asegurarse sin duda alguna es que la mutación más efectiva es la de tipo Eliminación, por lo que se mantendrá fija en el resto de las pruebas que se lleven a cabo.

En las siguientes pruebas se introducirá el factor Elitismo (al 10%) y se volverá a probar con ambos tipos de selección para investigar si, acompañadas de este recurso, muestran mayores diferencias entre ellas (300 individuos, 500 generaciones, Mutación Eliminación 10%, Elitismo 10%).

- Selección Estocástica: Mejor coste: 359; Media: 429,28; Desviación Típica: 28,11
- Selección Torneo: Mejor coste: 373; Media: 434; Desviación Típica: 25,64

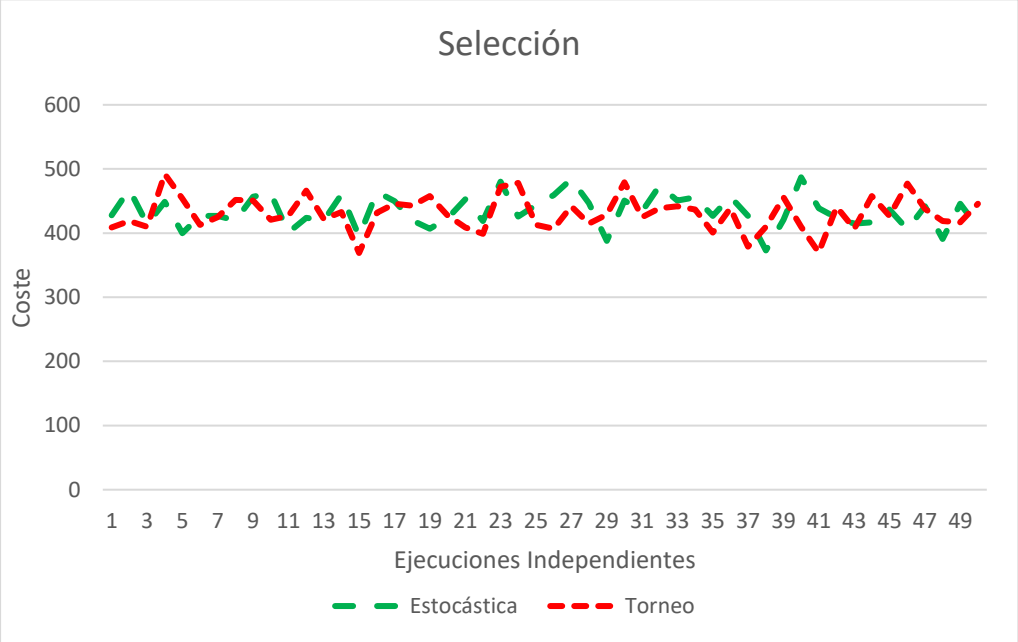

Figura 6.17- Selección Estocástica vs Selección Torneo

Como se puede observar, aunque ambos tipos de selección se han comportado de forma parecida en ambas baterías de pruebas (con y sin Elitismo), la selección Estocástica ha obtenido unos resultados ligeramente superiores en ambas modalidades, por lo que se muestra la mejor en este segundo problema de entrenamiento.

Para finalizar el estudio, se van a realizar unas pruebas que no se llevaron a cabo en el primer problema de entrenamiento pero que dados los resultados tan buenos que arroja la mutación por Eliminación pueden suponer una interesante línea de investigación. Se trata de aumentar ligeramente el porcentaje de la misma (15% y 20%) con el objetivo de ver si sigue progresando en la obtención de mejores resultados (300 individuos, 500 generaciones, Selección Estocástica, Elitismo 10%).

- Mutación Eliminación 15%: Mejor coste: 369; Media: 431,02; Desviación Típica: 26,52
- Mutación Eliminación 20%: Mejor coste: 370; Media: 431,62; Desviación Típica: 20,55

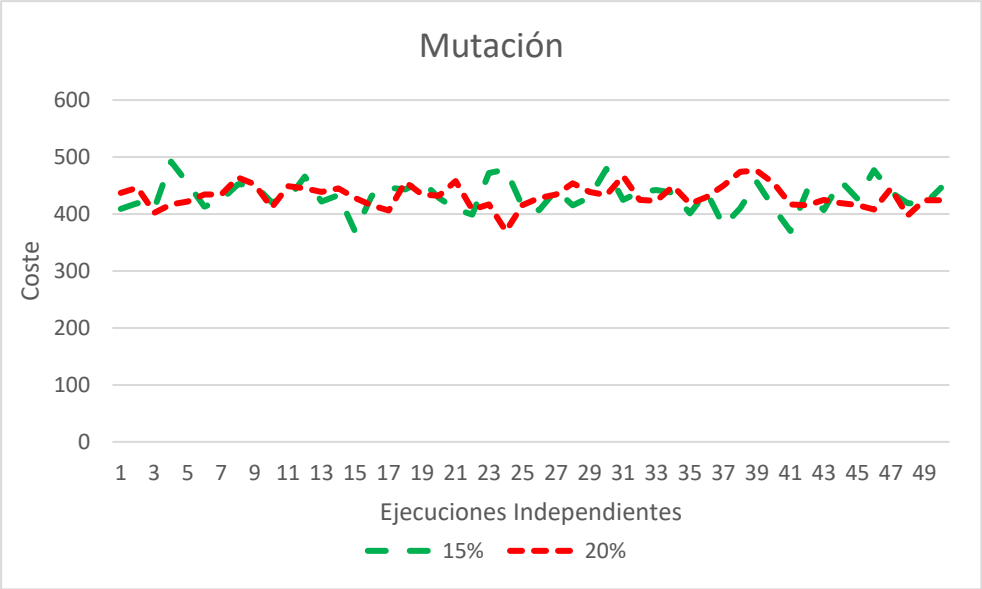

Figura 6.18 – Mutación 15% VS. Mutación 20%

Es interesante observar que, si bien por un lado los mejores costes hallados, así como la media son muy parecidos a los resultados obtenidos anteriormente, la Desviación Típica se reduce en varios puntos cuando se aumenta el porcentaje de mutación.

No obstante, si comparamos con los mejores resultados obtenidos anteriormente con mutación al 10% (ver gráfica inferior) puede observarse que, aunque la Desviación Típica obtenida con el 20% es mejor, tanto la media como los mejores costes se consiguen con la mutación al 10%, por lo que se mantendrá dicho porcentaje en futuras pruebas.

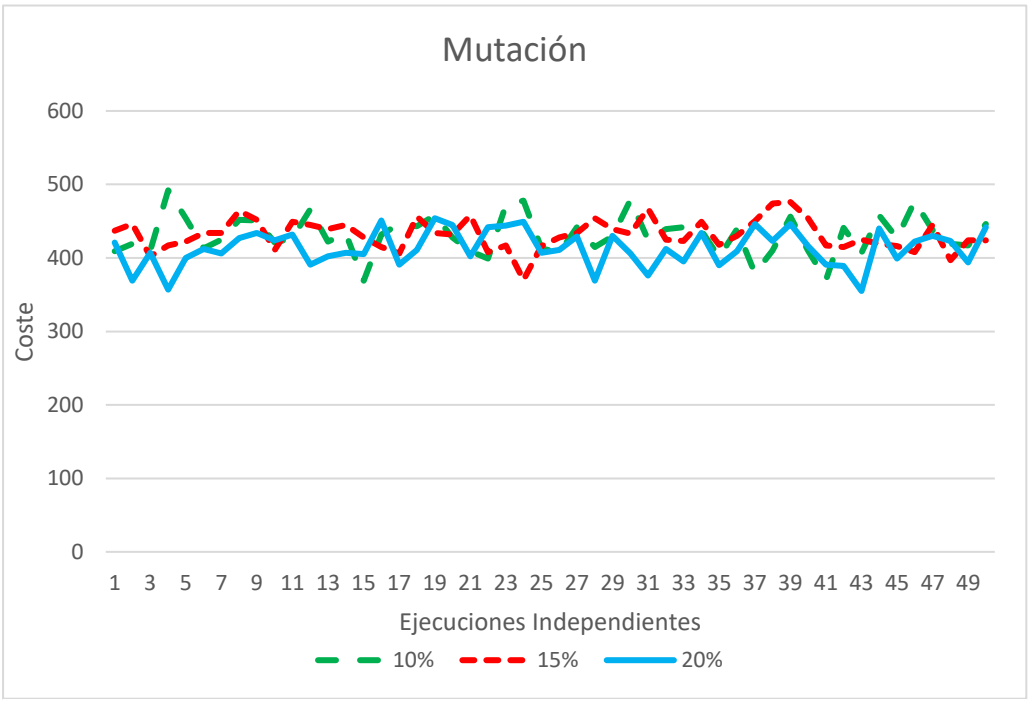

Figura 6.19 - Mutación 10% vs Mutación 15% vs Mutación 20%

Escenario con coste de carga 500:

Al igual que lo efectuado en el primer problema de entrenamiento se comenzarán las pruebas manteniendo aquellos parámetros que se han mostrado claramente más efectivos en las pruebas anteriores (penalizando carga) tales como el número de individuos, el número de generaciones y la mutación por eliminación al 10%, y se modificarán aquellos en los que la variación no ha sido suficientemente significativa para ser descartados. Básicamente, las variables que no se han mostrado claramente inferiores y que tendrían la posibilidad de mostrar alguna mejora en el escenario con carga son la selección Torneo y el Elitismo. Ambos consiguieron resultados muy cercanos a sus competidores (selección Estocástica y ausencia de Elitismo) en las pruebas penalizando la carga. Por tanto, situando el Coste de Carga en 1 para favorecerla, se ejecutarán diversas baterías de pruebas para investigar el comportamiento de las mejores configuraciones halladas anteriormente.

Dado que los parámetros que se van a variar suponen una combinatoria de cuatro posibilidades, se van comparar los cuatro grupos de resultados a la vez para observar de forma conjunta la mejor configuración general teniendo en cuenta todas las variables objeto del estudio. Es decir, aislando los demás parámetros (300 Generaciones, 500 Individuos, Cruce 70%, Mutación Eliminación 10%) se modificará el tipo de selección y la utilización de Elitismo (10%).

- Selección Estocástica – Sin Elitismo: Mejor coste: 393; Media 487,24; Desviación Típica: 56,91
- Selección Estocástica – Con Elitismo 10%: Mejor coste: 403; Media 491,78; Desviación Típica: 71,77
- Selección Torneo – Sin Elitismo: Mejor coste: 366; Media 430,12; Desviación Típica: 29,62
- Selección Torneo – Con Elitismo 10%: Mejor coste: 373; Media 430,58; Desviación Típica: 28,63

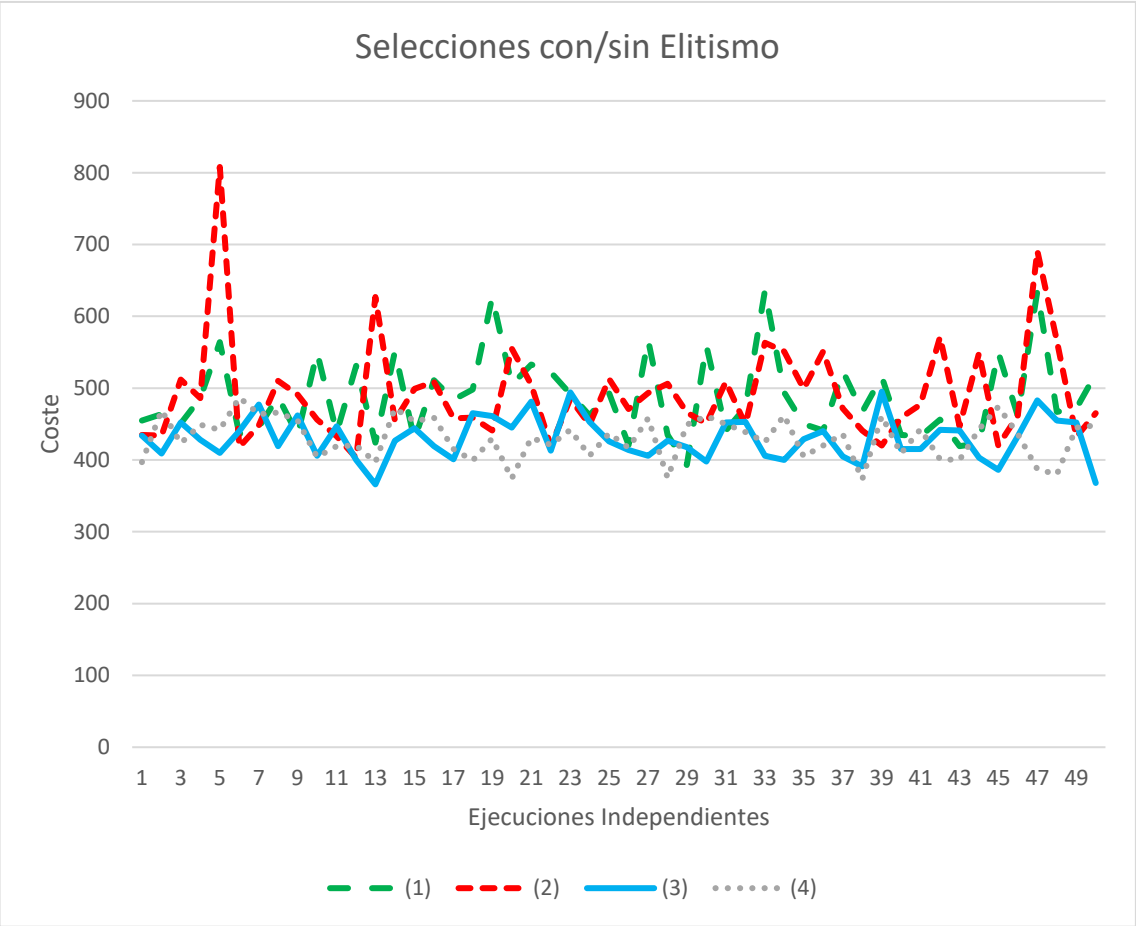

Figura 6.20 - Comparativa Selección Estocástica y Torneo – Sin/Con Elitismo

Tabla recopilatoria de resultados

A continuación, se presenta una tabla resumen de los resultados de las pruebas realizadas para este problema de entrenamiento:

Ejs	Generaciones	Individuos	Selección	Cruce	Mutación	Elitismo	Nodos	Densidad	C. Carga	Mejor	Media	Desviación
50	200	150	Estocástica	50%	Eliminación 10%	No	100	30%	500	388	459,52	38,87
50	200	150	Estocástica	70%	Eliminación 10%	No	100	30%	500	365	424,74	25,69
50	200	150	Estocástica	90%	Eliminación 10%	No	100	30%	500	369	369,00	39,39
50	400	200	Estocástica	50%	Eliminación 10%	No	100	30%	500	374	439,60	27,32
50	400	200	Estocástica	70%	Eliminación 10%	No	100	30%	500	383	427,98	25,07
50	400	200	Estocástica	90%	Eliminación 10%	No	100	30%	500	386	445,64	25,22
50	500	300	Estocástica	50%	Eliminación 10%	No	100	30%	500	392	436,74	24,33
50	500	300	Estocástica	70%	Eliminación 10%	No	100	30%	500	355	413,34	24,34
50	500	300	Estocástica	90%	Eliminación 10%	No	100	30%	500	365	426,26	26,91
50	500	300	Estocástica	70%	Eliminación 10%	No	100	30%	500	355	413,34	24,34
50	500	300	Torneo	70%	Transformación 10%	No	100	30%	500	1623	2169,20	214,16
50	500	300	Estocástica	70%	Transformación 10%	No	100	30%	500	1692	2335,02	233,51
50	500	300	Torneo	70%	Eliminación 10%	No	100	30%	500	379	442,64	26,05
50	500	300	Estocástica	70%	Eliminación 10%	10%	100	30%	500	359	429,28	28,11
50	500	300	Torneo	70%	Eliminación 10%	10%	100	30%	500	373	434,00	25,64
50	500	300	Estocástica	70%	Eliminación 15%	10%	100	30%	500	369	431,02	26,52
50	500	300	Estocástica	70%	Eliminación 20%	10%	100	30%	500	370	431,62	20,55
50	500	300	Estocástica	70%	Eliminación 10%	No	100	30%	1	393	487,24	56,91
50	500	300	Estocástica	70%	Eliminación 10%	10%	100	30%	1	403	403,00	71,77
50	500	300	Torneo	70%	Eliminación 10%	No	100	30%	1	366	430,12	29,62
50	500	300	Torneo	70%	Eliminación 10%	10%	100	30%	1	373	430,58	28,63

Figura 6.21 – Tabla resumen de las pruebas realizadas para el problema de entrenamiento con baja densidad

Conclusiones

Después de todas las baterías de pruebas efectuadas la mejor configuración hallada, a diferencia del primer problema de entrenamiento, varía para el entorno en el que se favorece la carga y el entorno en el que se penaliza la carga.

Mejor configuración de entorno penalizando carga:

- 300 generaciones
- 500 individuos
- Selección de tipo Estocástica
- Cruce al 70%
- Mutación por Eliminación al 10%
- Sin Elitismo

Mejor configuración de entorno favoreciendo la carga:

- 300 generaciones
- 500 individuos
- Selección de tipo Estocástica
- Cruce al 70%
- Mutación por Eliminación al 10%
- Sin Elitismo

La mejor solución hallada con penalización de carga fue de 355, con una Media en la batería de ejecuciones de: 413,34 y una Desviación Típica de 24,34.

Favoreciendo la carga, la mejor solución fue de 366, con una Media de 430,12 y una Desviación Típica de 29,62.

Como se puede observar el algoritmo genético no está aprovechando bien la funcionalidad de carga de nodos, esto hace que sus resultados no sean los esperados con coste de carga 1, como ya se ha visto en el anterior problema de entrenamiento.

ACO

Descripción de parámetros

Al igual que en el resto de las propuestas de solución, ACO tiene una gran cantidad de parámetros que se pueden modificar a la hora de ejecutar el algoritmo.

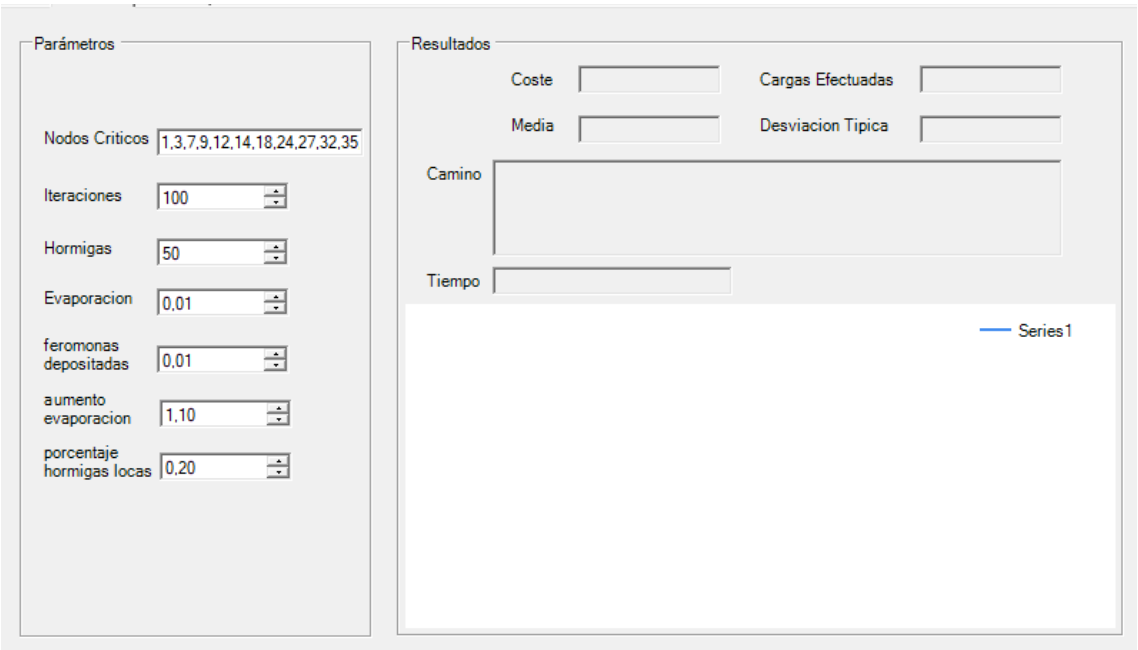

Figura 6.22 – Vista de ACO en la aplicación

A la izquierda de la imagen se pueden observar los parámetros disponibles:

Iteraciones: Número de veces que las hormigas van a ir desde un nodo critico hasta 0. Este factor es muy importante, ya que las hormigas forman un árbol y este árbol cubre todos los nodos críticos.

Hormigas: Número de hormigas que se van a lanzar desde cada nodo en la primera iteración del algoritmo.

Evaporación: Porcentaje de feromonas que se va a evaporar de cada nodo en cada iteración.

Feromonas depositadas: Es la cantidad de alicientes que van dejando las hormigas para que sus compañeras sigan su rastro. Este parámetro va muy ligado a la evaporación (lo lógico es que, si depositas más feromonas deberías evaporar también más).

Aumento evaporación: Indica cuanto va a aumentar el porcentaje de evaporación en cada iteración. En cada iteración se multiplica el porcentaje de evaporación por este parámetro (cuando el porcentaje de evaporación llega a 0.5 vuelve a su valor inicial).

Porcentaje hormigas locas: Porcentaje de hormigas que elegirán realizar movimientos aleatorios en lugar de seguir el camino formado por las feromonas, permitiendo aumentar la exploración del grafo.

Ajuste de parámetros

A la hora de ajustar los parámetros hay muchos factores a tener en cuenta, para ir acotando y obteniendo una configuración se va a empezar con los parámetros de iteraciones y numero de hormigas, que son los dos parámetros que más influyen en la calidad de los resultados.

El número de hormigas debe seleccionarse teniendo en cuenta la cantidad de nodos críticos que se quieren cubrir, ya que, si se quiere visitar 10 nodos críticos y se lanzan 10 hormigas por cada uno de estos, habrá 100 hormigas trabajando para buscar la solución. El número de iteraciones también es un parámetro que según la cantidad de nodos críticos puede llegar a aumentar mucho el coste en tiempo de resolución del problema. En cada iteración el algoritmo espera a que todas las hormigas lleguen al nodo destino, es decir, a mayor cantidad de nodos críticos, mayor será la espera para obtener los resultados del problema.

El porcentaje de evaporación y el porcentaje de feromonas depositadas se van a tratar en conjunto porque se complementan entre sí. Si se deposita una gran cantidad de feromonas y no se ajusta la evaporación, entonces se tendrá un grafo con una cantidad demasiado alta de feromonas en algunos nodos y otros nodos serán casi imposibles de visitar, lo que supondría una reducción del espacio de búsqueda. En el caso contrario el grafo resultante tendría unas diferencias despreciables entre unos caminos y otros, la convergencia en este caso sería más compleja.

Por último, están los parámetros de aumento de evaporación y porcentaje de hormigas locas. Estos parámetros sirven para aumentar el espacio de búsqueda y la variedad de soluciones. Aumentar mucho su valor produce ruido a la hora de conseguir una buena solución, en cambio si su valor es pequeño el espacio de búsqueda también se verá reducido.

A continuación, se van a realizar pruebas variando los parámetros comentados anteriormente en el problema de entrenamiento con el objetivo de encontrar una configuración óptima para más adelante realizar una comparación entre las metaheurísticas estudiadas en este trabajo.

Problema entrenamiento con alta densidad – 70%:

Tal como está contemplado en el problema de entrenamiento 1, las pruebas presentadas más adelante se van a dividir en dos secciones, pruebas en las que siempre se realizan cargas, y pruebas en las que nunca se realizaran cargas (los costes de carga están definidos en la introducción del punto 6, problema de entrenamiento).

En las pruebas expuestas en el caso de no indicarse lo contrario la evaporación es 0.01, feromonas depositadas 0.05 y aumento de evaporación de 1.10 (aumenta un 10% cada vez que se evaporan feromonas), se usan estos valores para los parámetros por que al

estar relacionados e ir incrementando la evaporación con el paso del tiempo parece recomendable el uso de esta configuración. El porcentaje de hormigas locas se ha dejado en 0.2 que es un valor lo suficientemente grande para ampliar el espacio de búsqueda, sin llegar a generar mucho ruido y hacer que las hormigas se comporten de forma errática.

Escenario con coste de carga 500

Las primera pruebas que vamos a realizar comparan los parámetros de iteraciones y numero de hormigas, empezando primero con las pruebas que han obtenido mejor resultado y mostrando posteriormente las pruebas con peores configuraciones y una comparativa de resultados entre las dos mejores configuraciones iteraciones–hormigas. Se han probado configuraciones de entre 200 y 600 iteraciones y 1 y 10 hormigas por nodo.

En primer lugar, el mejor coste obtenido en 50 ejecuciones ha sido de 169, coste conseguido por una configuración de 400 iteraciones - 1 hormiga y 600 iteraciones – 10 hormigas.

La configuración de 400 iteraciones - 1 hormiga ha obtenido una media de 181,8 y una desviación típica de 4,73:

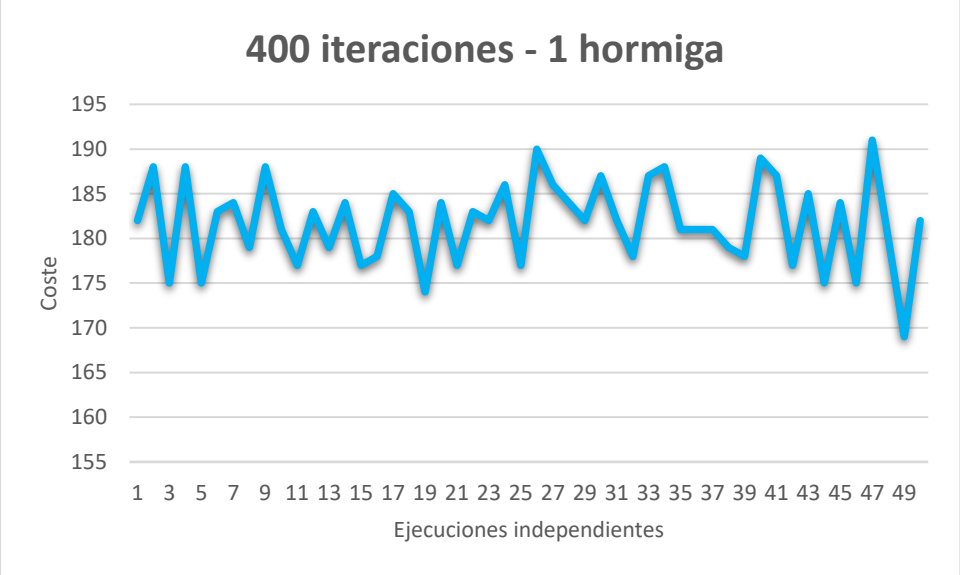

Figura 6.23 – Gráfica de 400 Iteraciones – 1 Hormiga

Como se puede observar en la gráfica, el valor de 169 solo ha sido obtenido una vez en 50 ejecuciones, pero gran parte de los valores obtenidos son menores de 180.

La configuración de 600 iteraciones - 10 hormiga también ha conseguido llegar al coste de 169 y ha obtenido una media de 185,56 en 50 ejecuciones con una desviación típica de 5,15.

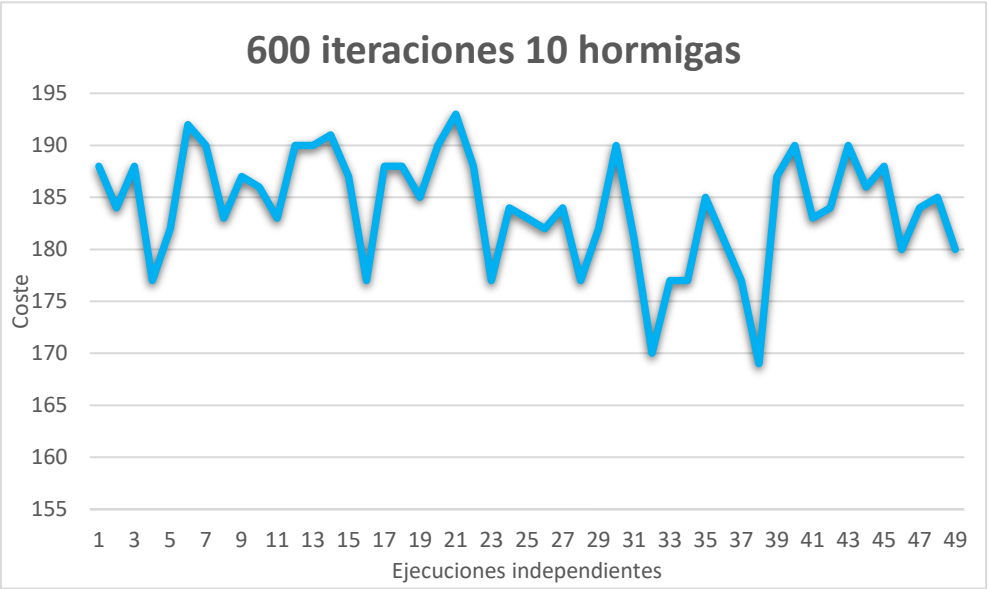

Figura 6.24 - Gráfica de 600 Iteraciones – 10 Hormigas

La grafica de esta configuración muestra que solo se ha llegado una vez al coste de 169 y que en general los resultados obtenidos suelen ser mayores de 180.

La otra configuración con resultados aceptables es de 400 iteraciones – 5 hormigas, ha obtenido un mejor resultado de 177 y una media de 185,6 en 50 ejecuciones con 5,15 de desviación típica.

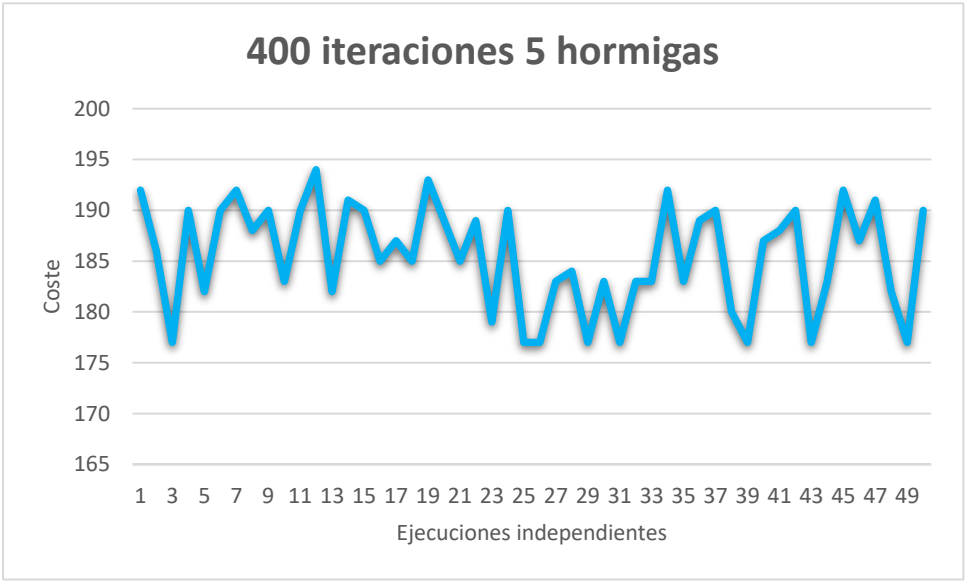

Figura 6.25 - Gráfica de 400 Iteraciones – 5 Hormigas

Esta configuración como podemos observar es peor que las dos anteriores, la gran mayoría de los resultados son mayores a 180 y algunos de ellos llegan a superar los 190.

En base a que una de las mejores configuraciones era de 400 iteraciones con una hormiga, se intentó probar la misma configuración con 200 iteraciones, pero los resultados no estaban a la altura, con un mejor coste de 175 y una media de 188,34 con una desviación típica de 6,09.

Tras descartar el resto de las pruebas realizadas por no llegar a la cota de calidad de las dos mejores configuraciones que mejores resultados han sacado, se va a ver una comparativa entre estas:

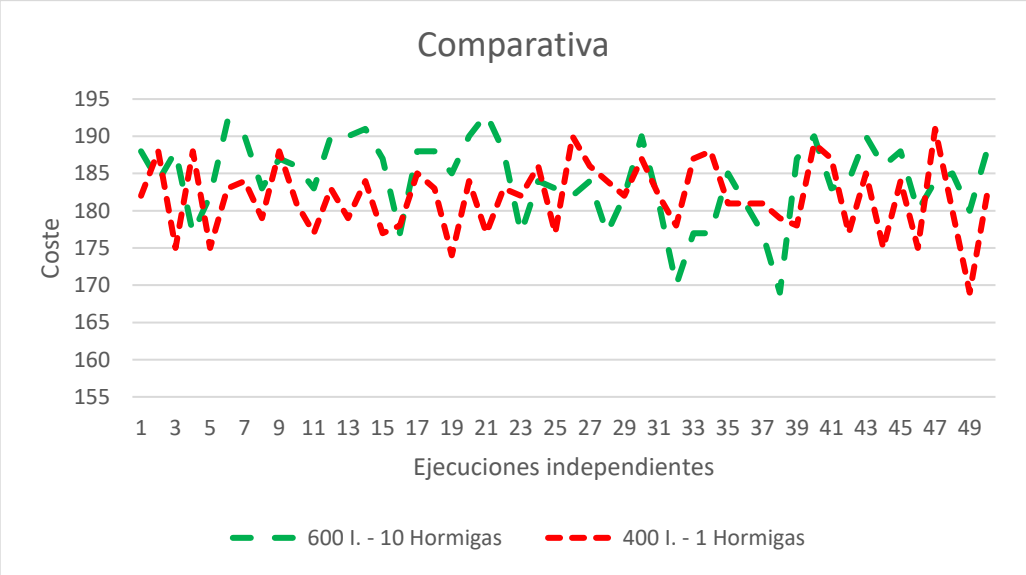

Figura 6.26 – 600 Iteraciones – 10 Hormigas vs 400 Iteraciones – 1 Hormiga

Como se puede observar en la gráfica, aunque la de 600 iteraciones ha conseguido llegar una vez a la mejor solución y otra se ha quedado muy cerca, en general la gráfica amarilla tiene unos costes inferiores, y por lo tanto la configuración de 400 iteraciones - 1 hormiga es la mejor opción iteraciones/hormigas en problemas con coste de carga 1.

Con la configuración obtenida anteriormente, se va a probar a modificar el valor del resto de parámetros que puedan llegar a tener un impacto positivo en la calidad de los resultados. Primero se van a mostrar diferentes variaciones del porcentaje de hormigas locas y luego se verán varias configuraciones de evaporación, aumento de evaporación y deposición de feromonas.

Si se varía el porcentaje de hormigas locas puede que el espacio de búsqueda se amplié llegando a soluciones que con la configuración que se ha establecido, se va a probar a aumentar este parámetro, para ver qué resultados se obtienen.

Realizando diferentes variaciones del parámetro, la mayoría de las pruebas han obtenido costes peores a lo obtenido por las pruebas realizadas previamente salvo la configuración de 0.4 que sin llegar a sacar mejores resultados o igualarlos sí que se ha acercado.

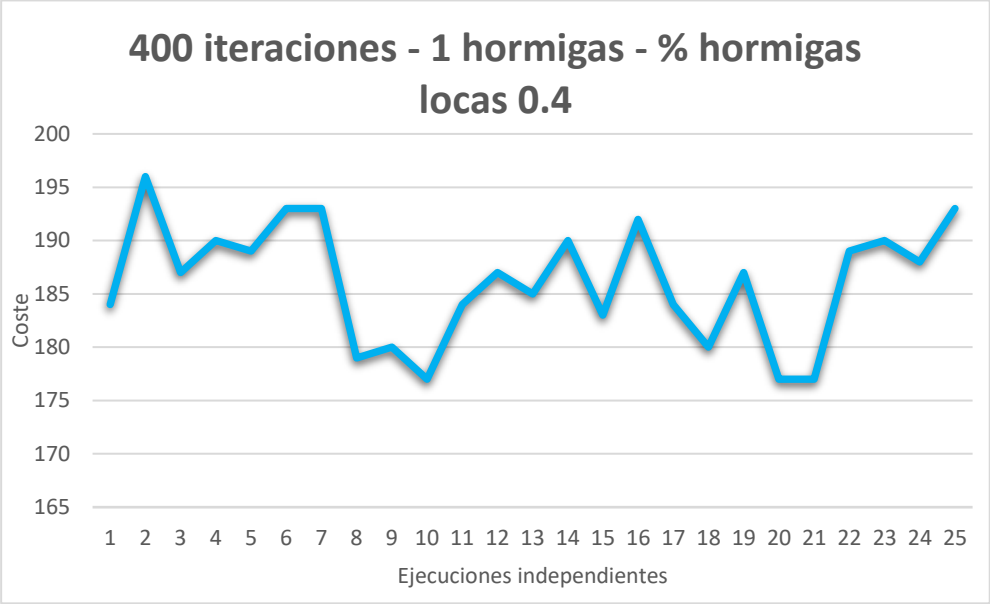

Figura 6.27 – Resultados de 400 Iteraciones – 1 Hormiga y H. Locas al 40%

Como demuestran los resultados de la gráfica, hay ocasiones en las que saca resultados por debajo de 180, que son más cercanos a los obtenidos por la mejor configuración de iteraciones-hormiga, pero su media en 25 iteraciones es de 186,16 con una desviación típica 5,44 y como se puede ver gran parte de las ejecuciones está por encima de 180, llegando incluso a picos de más de 195 de coste. Este resultado es peor que dos configuraciones vistas con anterioridad, por lo que según los datos obtenidos aumentar el porcentaje de hormigas locas no es una buena opción a la hora de mejorar los resultados.

Ahora vamos a ver una comparativa entre la mejor configuración iteraciones-hormigas y esta misma configuración con el parámetro de porcentaje de hormigas locas con valores del 40 y el 80 por ciento respectivamente.

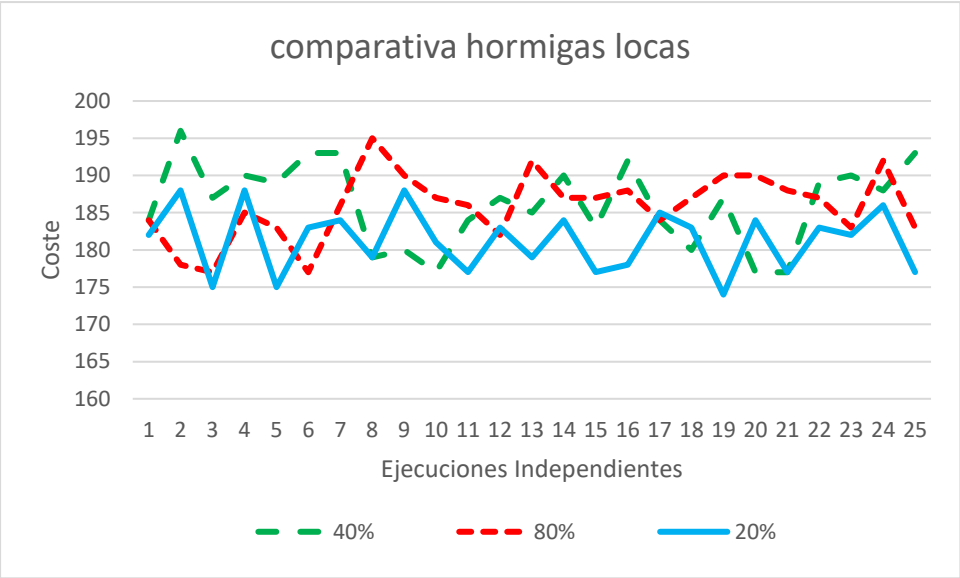

Figura 6.28 – Comparativa entre variaciones del porcentaje de H. Locas

Como se puede apreciar en la gráfica la configuración original de 20 por ciento es la mejor de las configuraciones estando en la gran mayoría de casos por debajo de las otras dos, las otras configuraciones también tienen unos resultados aceptables, pero gran parte de sus resultados se sitúan por encima de 185. Aquí tenemos una tabla con sus media, desviaciones típicas y mejor resultado en 25 ejecuciones.

	40%	80%	20%
Media	185,999561	185,81333	181,184274
Desviación típica	5,44558537	4,43549321	4,16192263
Mejor solución	177	177	174

En base a los resultados finalmente se puede concluir que la configuración de porcentaje de hormigas locas utilizada para las pruebas iteraciones – hormigas es la mejor de las tres opciones.

Otro parámetro cuya variación puede tener un impacto positivo a la hora de conseguir que se refuercen mejor los caminos es el aumento del número de feromonas depositadas. Para probar una variación de este parámetro se decidió aumentar -aunque no de manera proporcional- el porcentaje de evaporación, ya que si no siempre se queda con la primera solución que encuentra en la primera iteración.

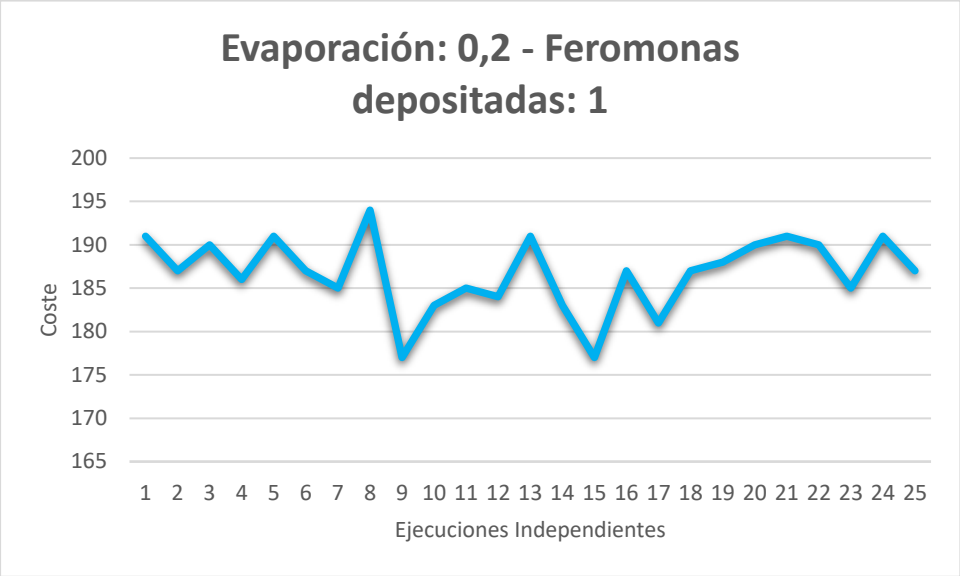

Figura 6.29 – Resultados Evaporación al 20% y Feromonas depositadas con valor 1

Los resultados que arroja la gráfica con media 186,62 con desviación típica de 4,21 y mejor coste de 177 en 25 ejecuciones independientes, demuestra que claramente esta configuración no obtiene mejor resultados que la mejor configuración encontrada anteriormente.

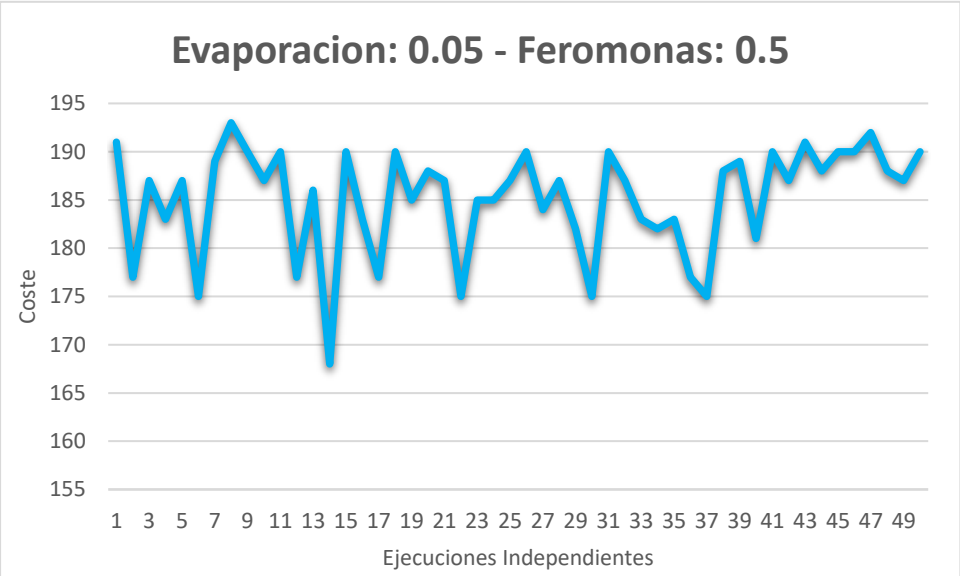

Figura 6.30 – Evaporación al 5% y Feromonas depositadas con valor 0,5

La siguiente grafica muestra que esta configuración parece tener menos robustez que la configuración de 400 iteraciones – 1hormiga vista anteriormente, con unos resultados muy dispares entre las diferentes ejecuciones. Esto a la vez hace que esta configuración haya tenido más suerte de encontrar la mejor solución hasta hora con un coste de 168, la media en 50 ejecuciones es de 185,16 y la desviación típica es de 5,54.

En la siguiente configuración vamos a intentar utilizar una configuración de 200 iteraciones con 10 hormigas aumentando el porcentaje de evaporación y el aumento de evaporación, viendo como esta relación de parámetros acepta en los resultados.

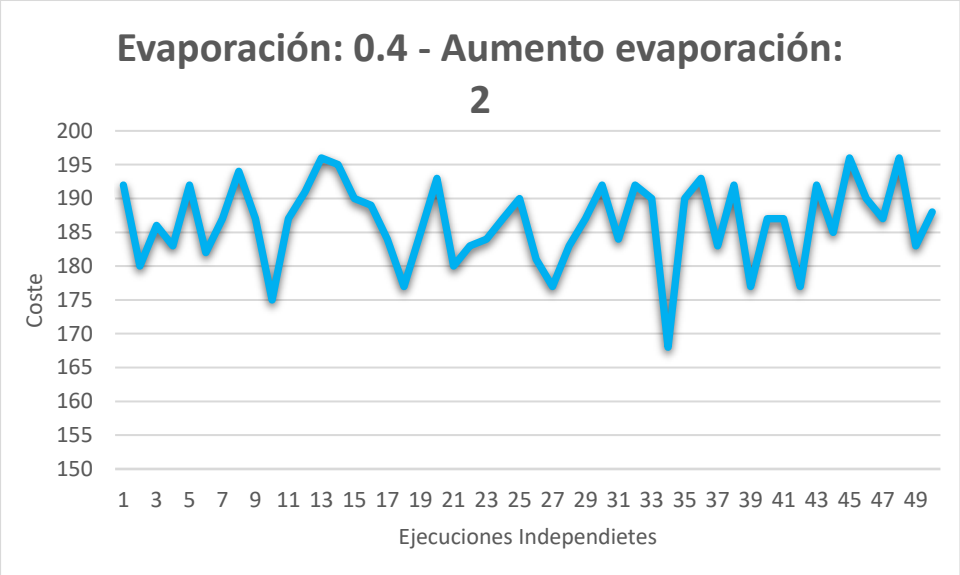

Figura 6.31 – Resultados Evaporación al 40% y el Aumento de Evaporación con valor 2

Esta configuración con una mejor solución de 168 igual que la anterior, pero en este caso la media es de 186,52 con una desviación típica de 6,02. Con una de las peores medias y una alta desviación típica parece que esta configuración no es muy recomendable.

En base a los resultados intentamos realizar otro ajuste con solo 0.2 de evaporación, un aumento de evaporación de 1,5, 400 iteraciones 5 hormigas y 0,05 feromonas depositadas.

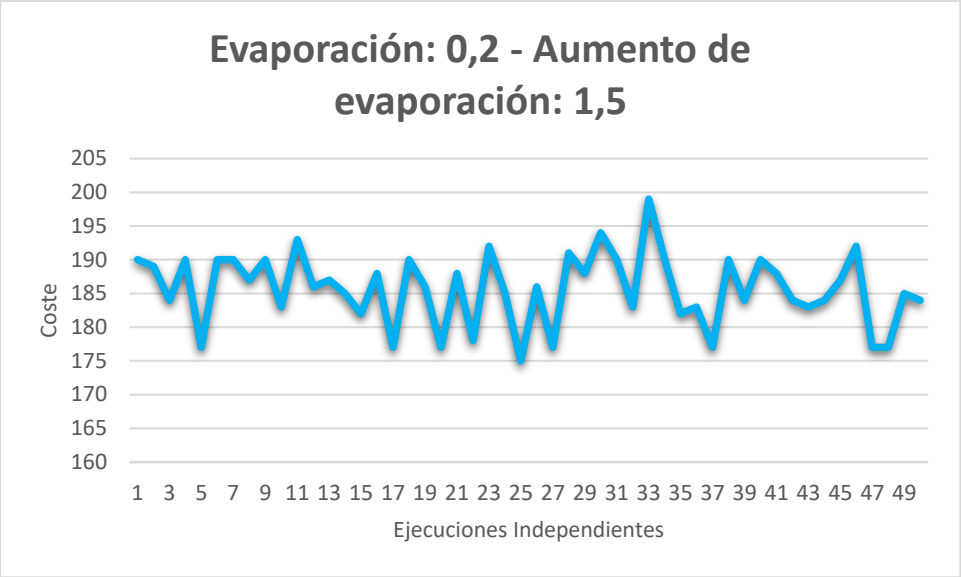

Figura 6.32 – Resultados Evaporación al 20% y Aumento de Evaporación con valor 1,5

Los resultados de esta configuración tampoco son muy buenos, con una mejor solución de 175 y una media de 185,68 con una desviación típica de 5,3.

Tabla recopilatoria de resultados

A continuación, se presenta una tabla resumen de los resultados de las pruebas realizadas para este problema de entrenamiento:

Ejs	Iteraciones	Hormigas	Evaporación	Feromonas	Aumento	% H.locas	Nodos	Densidad	Carga	Mejor	Media	Desviacion
50	400	1	0,01	0,05	1,1	0,2	100	70%	500	169	181,8	4,73
50	600	10	0,01	0,05	1,1	0,2	100	70%	500	169	185,56	5,15
50	400	5	0,01	0,05	1,1	0,2	100	70%	500	177	185,6	5,15
25	400	1	0,01	0,05	1,1	0,2	100	70%	500	174	181,18	4,16
25	400	1	0,01	0,05	1,1	0,4	100	70%	500	177	186,16	5,44
25	400	1	0,01	0,05	1,1	0,8	100	70%	500	177	185,81	4,43
25	400	1	0,2	1	1,1	0,2	100	70%	500	177	186,62	4,21
50	400	1	0,05	0,5	1,1	0,2	100	70%	500	168	185,16	5,54
50	400	10	0,4	0,05	2	0,2	100	70%	500	168	186,52	6,02
50	400	1	0,2	0,05	1,5	0,2	100	70%	500	175	185,68	5,3

Figura 6.33 – Tabla resumen de las pruebas realizadas para el problema de entrenamiento con alta densidad y coste de carga 500

Conclusiones:

La mejor configuración iteraciones – hormigas encontrada es la de 400 iteraciones - 1 hormiga, esta configuración además de haber obtenido el segundo mejor resultado, en la mayor parte de las ejecuciones que ha realizado ha encontrado valores cercanos al mejor.

El aumento del número de feromonas depositadas hace que el algoritmo explore más, pero también hace que en muchas de las ejecuciones el algoritmo no consiga encontrar una buena solución, obteniendo peores resultados que la configuración de parámetros de 400 iteraciones - 1 hormiga. En el caso de aumentar el porcentaje de evaporación y el aumento de evaporación el resultado también es peor que el de la

configuración de 400 iteraciones – 1 hormiga, y vemos que en este caso le cuesta aún más encontrar buenas soluciones.

En vista de los resultados obtenidos en las pruebas, la mejor configuración para el algoritmo de hormigas para el problema de entrenamiento 1 es la siguiente: 400 iteraciones, 1 hormiga por nodo crítico, 0,01 de porcentaje de evaporación, 0,05 feromonas depositadas y aumento de evaporación de 1,10. Con una mejor solución de 169 y una media de 181,8 con desviación típica de 4,73.

Escenario con coste de carga 1

Para las pruebas en el problema de entrenamiento en las que siempre se carga, se va a seguir un planteamiento parecido a la búsqueda de la configuración óptima sin carga, obteniendo en primer lugar la mejor configuración iteraciones – hormigas.

Mientras no se indique lo contrario mantendremos la siguiente configuración para el resto de parámetros: evaporación 0,01 – feromonas depositadas 0.05 – aumento de evaporación 1.10 – % hormigas locas 0,20.

El mejor resultado obtenido en estas pruebas es de 75 con la configuración de 600 iteraciones – 10 hormigas, con media de 90,91 y desviación típica de 5.9 en un total de 50 ejecuciones.

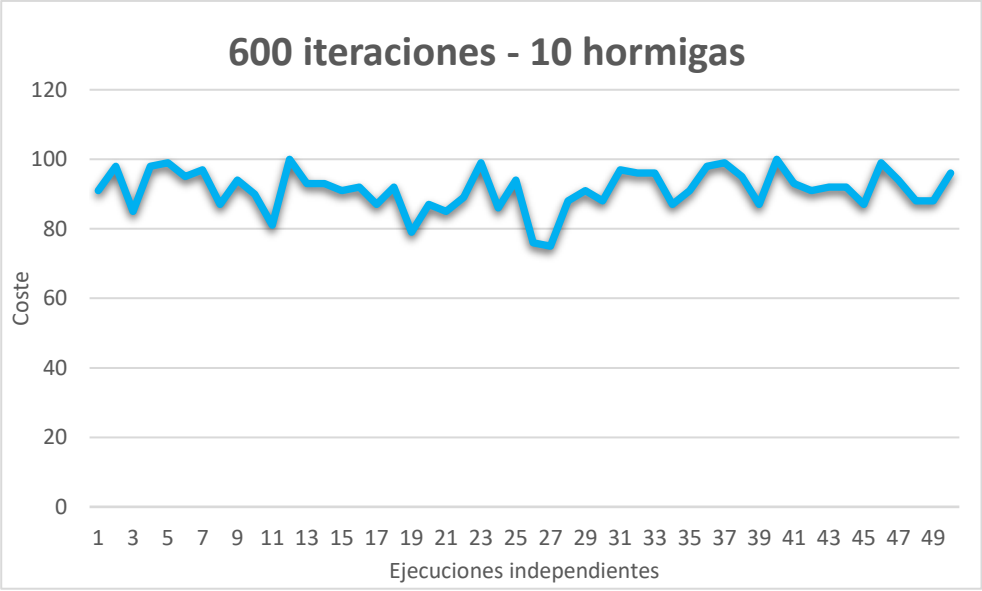

Figura 6.34 - Gráfica de 600 Iteraciones – 10 Hormigas

Los resultados de la gráfica indican que gran parte de los valores es cercana a 90, se obtienen varias veces resultados de 80 o menos y nunca se llega a superar el coste 100.

El segundo mejor resultado en la relación iteraciones-hormigas se obtiene con unos parámetros de 400 iteraciones – 5 hormigas, con una media de 101.36 y desviación típica 8.12. El mejor resultado de esta configuración es de 80.

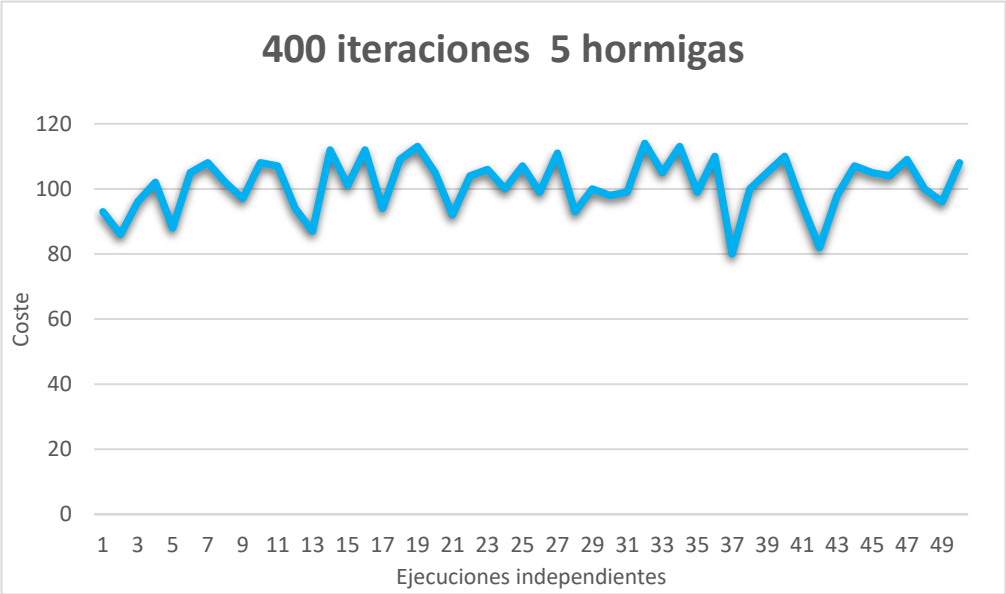

Figura 6.35 – Gráfica de 400 Iteraciones – 5 Hormigas

En la gráfica se observa que los resultados suelen estar por encima de 100, aunque varias veces ha bajado a coste inferiores a 85.

El resto de las pruebas realizadas, no llegan a las cotas alcanzadas por estas configuraciones, pero aun así caben destacar los resultados de las configuraciones de 400 iteraciones – 3 hormigas y 400 iteraciones - 1 hormigas.

La configuración de 400 iteraciones – 3 hormigas, se planteó intentando conseguir mejorar los resultados de la configuración de 400 iteraciones – 5 hormigas (configuración que ha obtenido los segundos mejores resultados), pero al final los resultados no fueron los esperados, con una mejor solución de 86, una media de 106,84 y una desviación típica de 6,73 en 50 ejecuciones.

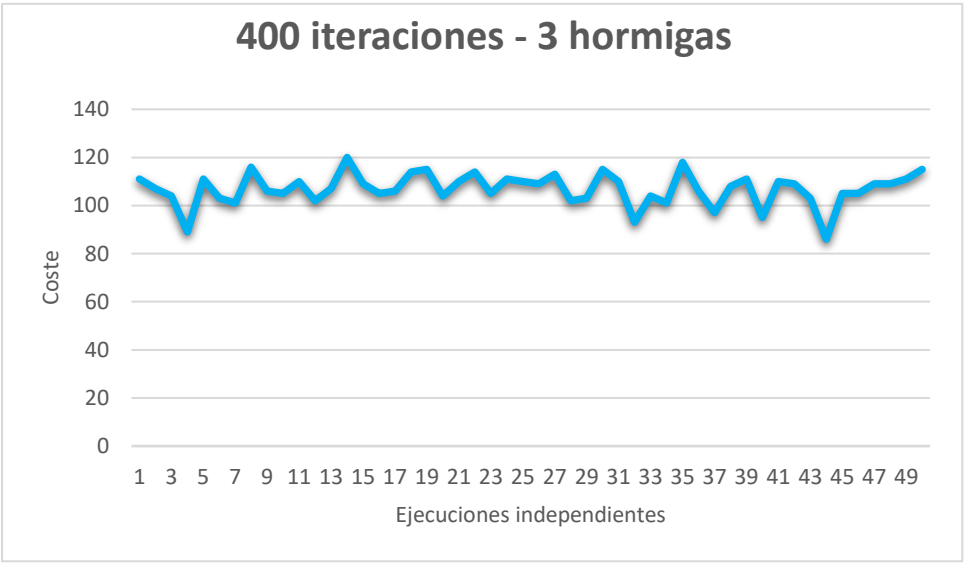

Figura 6.36 - Gráfica de 400 Iteraciones – 3 Hormigas

En base a los resultados obtenidos por la configuración de 400 iteraciones – 1 hormiga en el problema de entrenamiento con coste de carga 500, se decidió lanzar pruebas para ver si esta configuración también funcionaba para las pruebas en las pruebas con coste de carga 1.

El resultado no fue el esperado, dando a lugar a una gráfica con valores mayoritariamente por encima de 100, una mejor solución de 95, media de 118,38 y desviación típica de 8,03 en 50 ejecuciones.

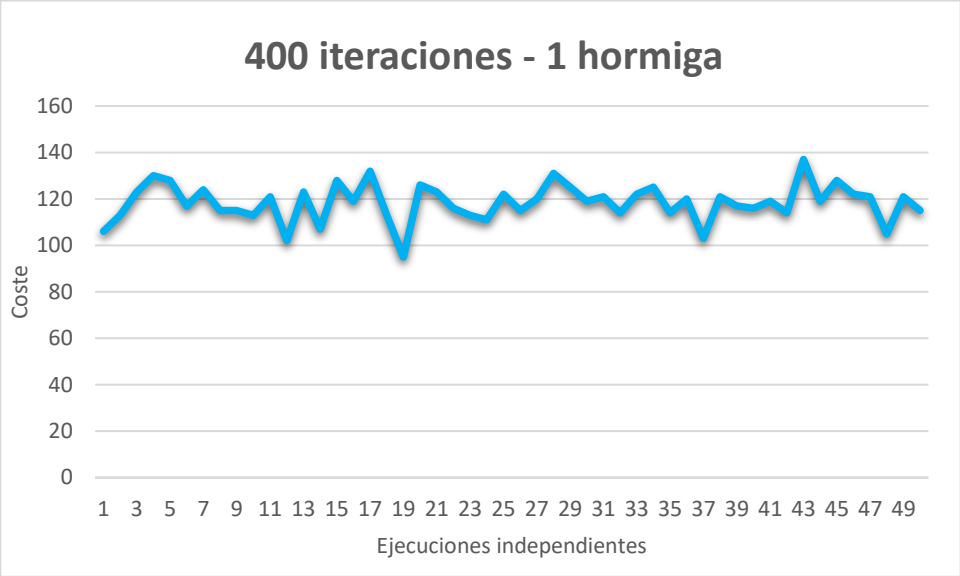

Figura 6.37 - Gráfica de 400 Iteraciones – 1 Hormiga

Tras haber conseguido la mejor configuración iteraciones – hormigas para el problema de entrenamiento con coste de carga 1, se va a proceder a modificar el resto de parámetros para comprobar si se puede llegar a una mejor configuración.

En las siguientes configuraciones se han probado los parámetros de evaporación y feromonas depositadas manteniendo siempre los valores de 600 iteraciones y 10 hormigas, aumentando la evaporación a 0.5 y con 0.01 de feromonas depositadas, los resultados son parejos a la configuración de 400 iteraciones – 1 hormiga, con una mejor solución de 92, una media de 101,1 y 5,26 de desviación típica.

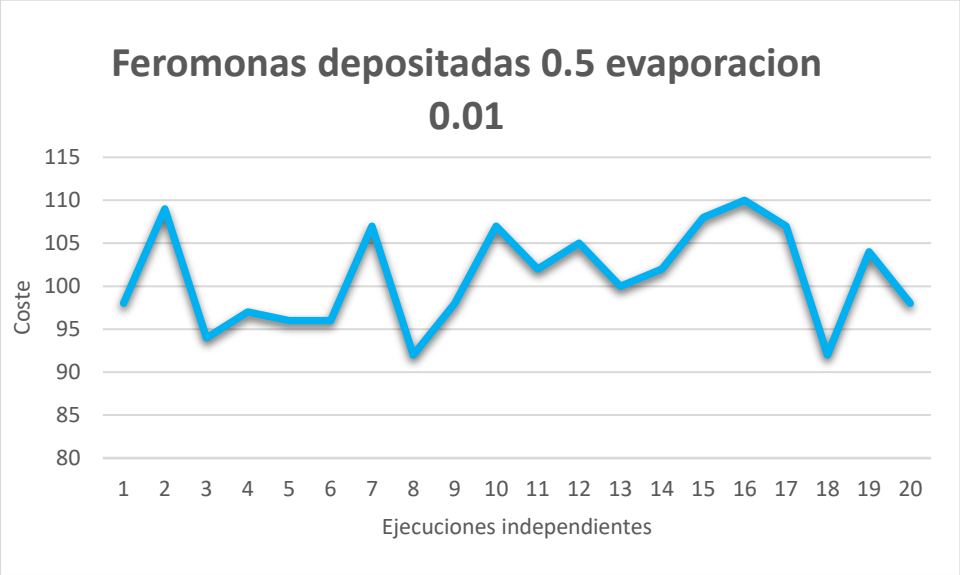

Figura 6.38 – Feromonas depositadas con valor 0,5 y Evaporación al 10%

Para comprobar si se podían mejorar los resultados aumentando el número de feromonas depositadas, se probó una configuración con evaporación 0.05 y feromonas depositadas 1.0.

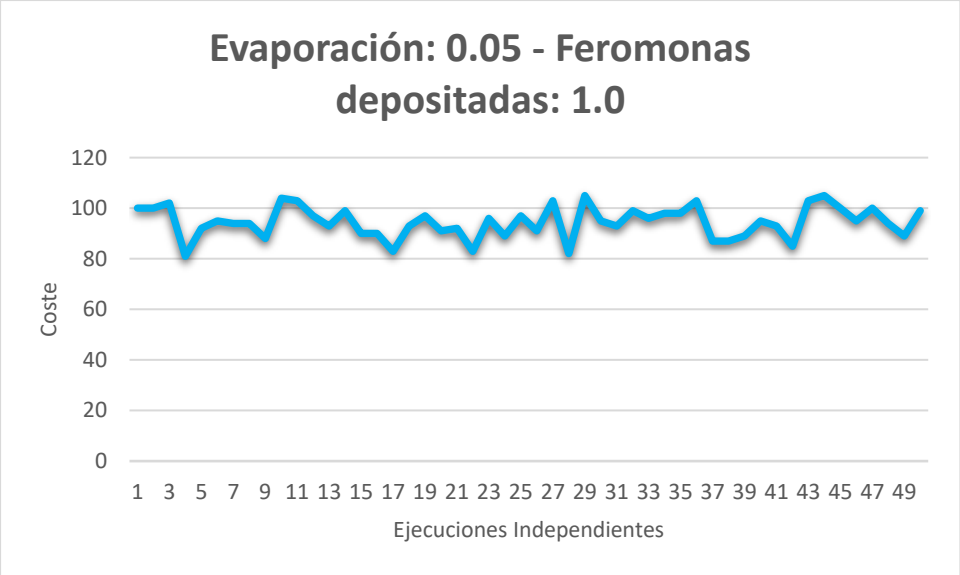

Figura 6.39 – Evaporación al 5% y Feromonas Depositadas con valor 1

En la gráfica se puede observar que los resultados se sitúan entre 100 y 80 sin ningún resultado superior a 110, esta solución con un mejor resultado de 81 y una media de 94,54 con desviación típica de 6,18 es una de las mejores encontradas hasta la fecha.

En cuanto al porcentaje de hormigas locas, se intentó aumentar su porcentaje a 0.8 y los resultados no han sido buenos, con una mejor solución de 99, una media de 116,8 y una desviación típica de 7.

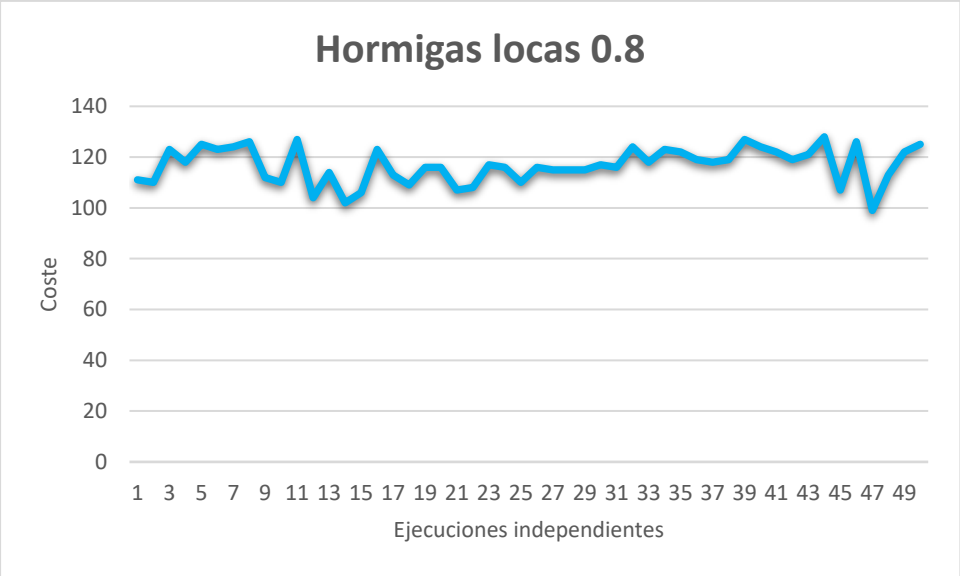

Figura 6.40 – Resultados de H. Locas al 80%

La gráfica nos indica que los resultados son en su mayoría por encima de 100 y algunos están por encima de 120, con lo que la opción de aumentar el porcentaje de hormigas locas no es una buena opción.

La siguiente grafica es una comparativa entre tres configuraciones con 20, 40 y 80 porciento de hormigas locas y el resto de parámetros iguales que los de la mejor configuración iteraciones-hormigas para el problema de entrenamiento 1 con coste de carga 1.

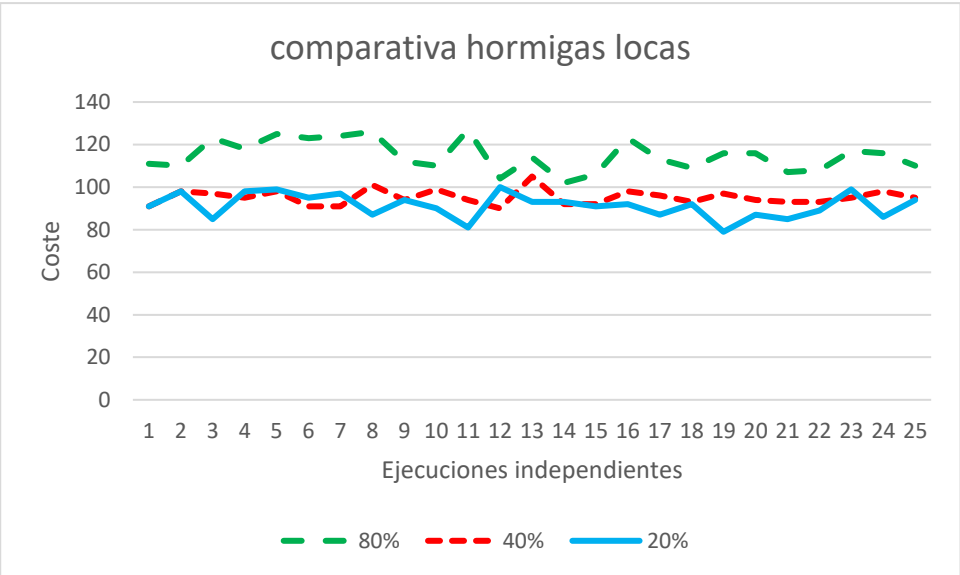

Figura 6.41 – Comparativa de diferentes porcentajes de H. Locas

Como se puede observar en la gráfica, aumentar el porcentaje de hormigas locas es contraproducente y se consiguen mejores resultados con la configuración de un 20 por ciento. Ahora vamos a ver una tabla con las medias, desviaciones típicas y mejor solución.

	80%	40%	20%
Media	114,357143	95,0763605	90,9302686
Desviación típica	7,14982517	3,47562944	5,58225761
Mejor solución	102	90	81

Los resultados finalmente muestran que en este caso el aumento del porcentaje de hormigas locas es contraproducente y la mejor opción es mantener este porcentaje al 20 por ciento.

Por último, vamos a comparar las dos mejores configuraciones para coste de carga 1, la primera es la de 600 iteraciones – 10 hormigas con el resto de parámetros sin modificar y esta misma relación de iteraciones -hormigas con una evaporación de 0,05 y 1 de feromonas depositadas.

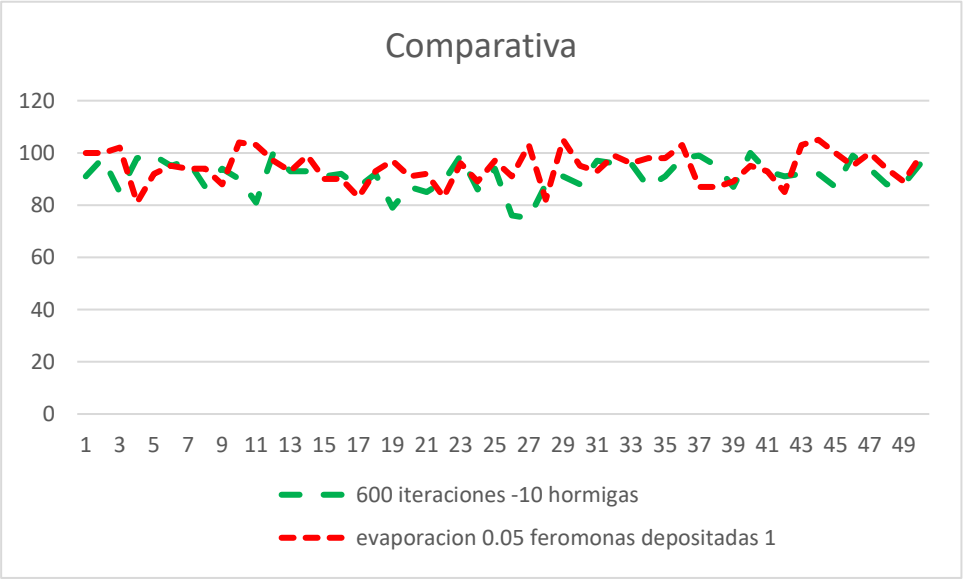

Figura 6.42 – Comparativa entre las mejores configuraciones encontradas

La grafica indica que los resultados en este caso están muy ajustados, si se aprecia que en general la configuración de 600 iteraciones y 10 hormigas tiene mejores resultados, la media de esta configuración es de 90,91 con una desviación típica de 5.9, estos resultados son mucho mejores que los de la segunda configuración con una media de 94,54 con desviación típica de 6,18.

Tabla recopilatoria de resultados

A continuación, se presenta una tabla resumen de los resultados de las pruebas realizadas para este problema de entrenamiento:

Ejs	Iteraciones	Hormigas	Evaporación	Feromonas	Aumento	% H.locas	Nodos	Densidad	Carga	Mejor	Media	Desviacion
50	600	10	0,01	0,05	1,1	0,2	100	70%	1	75	90,91	5,9
50	400	5	0,01	0,05	1,1	0,2	100	70%	1	80	101,36	8,12
50	400	3	0,01	0,05	1,1	0,2	100	70%	1	86	106,84	6,73
50	400	1	0,01	0,05	1,1	0,2	100	70%	1	95	118,38	8,03
20	600	10	0,01	0,5	1,1	0,2	100	70%	1	92	101,1	5,26
50	600	10	0,05	1.0	1,1	0,2	100	70%	1	81	94,54	6,18
25	600	10	0,01	0,05	1,1	0,2	100	70%	1	81	90,93	5,58
25	600	10	0,01	0,05	1,1	0,4	100	70%	1	90	95,07	3,47
25	600	10	0,01	0,05	1,1	0,8	100	70%	1	102	114,35	7,14

Figura 6.43 – Tabla resumen de las pruebas realizadas para el problema de entrenamiento con alta densidad y coste de carga 1

Conclusiones:

La mejor configuración para ACO con el problema de entrenamiento con coste de carga 1 es de 600 iteraciones, 10 hormigas, 0,05 evaporación, 0,01 feromonas depositadas, 1,1 aumento de evaporación y 0,2 porcentaje de hormigas locas, esta configuración ha obtenido una mejor solución de 75 y una media 90,91 con desviación típica de 5,9. Esta misma configuración con una evaporación de 0,05 y 1 de feromonas depositadas se queda en el segundo puesto con una mejor solución de 81 y una media de 84,54 con desviación típica 6,18.

Un aumento del porcentaje de evaporación no nos otorga unos buenos resultados quedándose muy por detrás de estas configuraciones, y el resto de las configuraciones con diferente relación iteraciones – hormigas no llegan a las cotas de estas dos. En cuanto al porcentaje de hormigas locas se evidencia que las configuraciones con un 40 y un 80 por ciento de hormigas locas impactan negativamente en la calidad de los resultados y la mejor opción de las tres probadas es la de mantener este parámetro en un 20 por ciento.

Problema de entrenamiento con baja densidad – 30%:

Con las configuraciones optimas iteraciones – hormigas obtenidas en las pruebas del problema anterior, se van a realizar nuevas pruebas variando el resto de parámetros en este problema, de esta manera se verá cómo afecta el número de vecinos al ajuste de parámetros.

Escenario con coste de carga 500

La configuración de la que se va a partir para todas las pruebas es la siguiente: 400 iteraciones, 1 hormiga, 0,01 evaporación, 0,05 feromonas depositadas y aumento de evaporación de 1,10. En todas las pruebas se indicarán los parámetros modificados y el resto de los parámetros se mantendrán de acuerdo con esta configuración.

El primer parámetro que vamos a probar en este apartado es el porcentaje de hormigas locas realizando una comparativa entre poner este valor al 20, 40 y 80 por ciento.

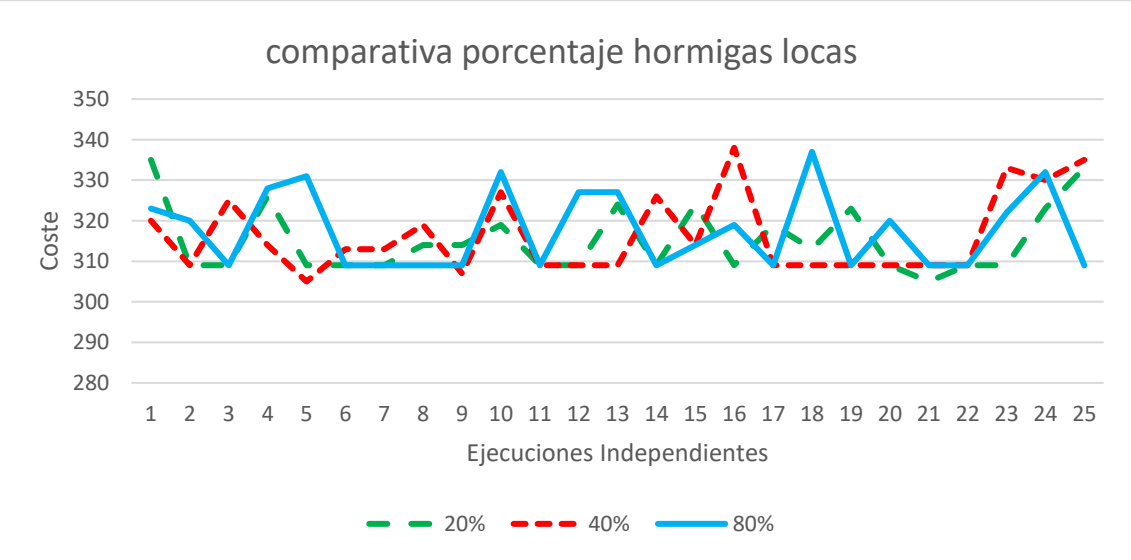

Figura 6.44 – Comparativa de distintos porcentajes de H. Locas

Como se puede observar en la gráfica la diferencia de resultados entre las tres configuraciones es más reducida en este problema. Los resultados en cuanto a media, desviación típica y mejor resultado se muestran en la siguiente tabla.

	20 %	40 %	80 %
Media	314,988807	316,067818	317,324852
Desviación típica	8,25590698	9,74014374	9,41912947
Mejor solución	309	305	305

En base a los resultados, la mejor opción por su media y desviación sigue siendo la de un 20 por ciento de hormigas locas como en el problema anterior, pero en este caso la

mejor solución la sacan las otras dos configuraciones con una diferencia de 4 puntos de coste por debajo de la configuración del 20 por ciento.

Los parámetros de evaporación, aumento de evaporación y feromonas depositadas los tratamos en conjunto por que tienen una relación entre ellos. Las configuraciones que se van a probar tratan los siguientes escenarios:

1. La evaporación es baja y se depositan más feromonas de lo que se evapora. La configuración en este caso es: evaporación 0,05 – depositar feromonas 0,5.
2. La evaporación es media con un aumento de evaporación más alta y una deposición de feromonas baja en relación con estos parámetros. La configuración en este caso es: evaporación 0,20 - aumento de evaporación 1,50.
3. La evaporación es media y la deposición de feromonas alta. La configuración en este caso es: evaporación 0,20 – feromonas depositadas 1.

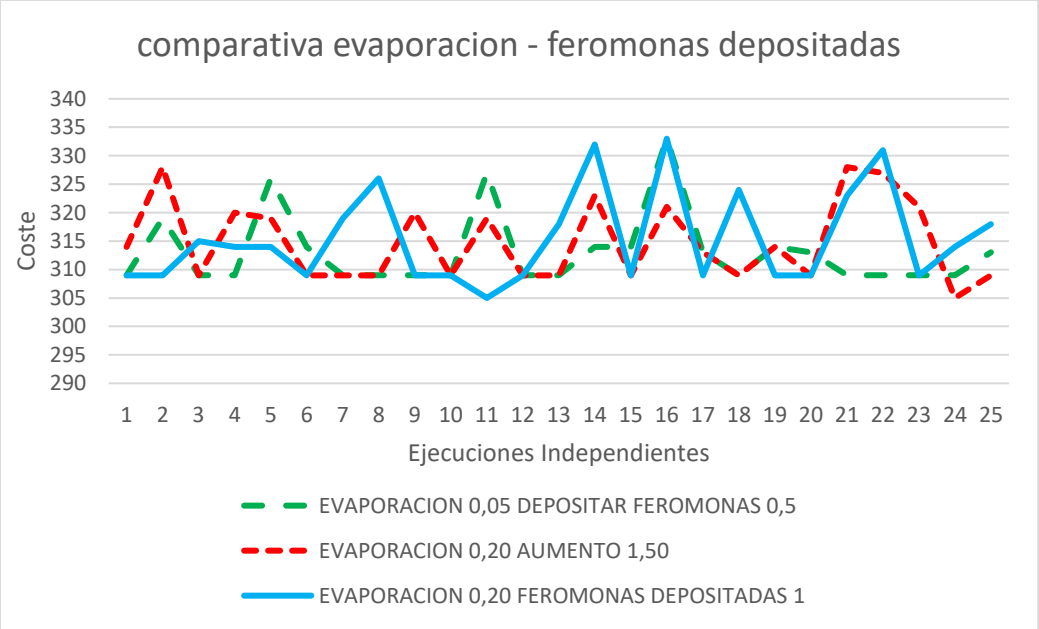

Figura 6.45 – Comparativa entre distintos valores de Evaporación/Feromonas Depositadas

Los resultados de la gráfica son muy parecidos en los tres casos, en general la gráfica azul es un poco más robusta, pero sin grandes diferencias. En la siguiente tabla se van a ver las medias, desviaciones y mejor solución de las tres configuraciones.

	Evaporación 0,05 Depositar feromonas 0,5	Evaporación 0,20 Aumento de evaporación 1,50	Evaporación 0,20 Feromonas depositadas 1
Media	312,913244	314,68917	315,193642
Desviación	6,41548128	6,9321281	8,15843122
Mejor solución	309	305	305

Finalmente se puede concluir que de los tres escenarios el primero de todos es el mejor para el problema de entrenamiento con baja densidad de vecinos, por su media

y desviación típica, a pesar de que los otros dos escenarios han conseguido una mejor solución en 25 ejecuciones.

Tabla recopilatoria de resultados

A continuación, se presenta una tabla resumen de los resultados de las pruebas realizadas para este problema de entrenamiento:

Ejs	Iteraciones	Hormigas	Evaporación	Feromonas	Aumento	% H. locas	Nodos	Densidad	Carga	Mejor	Media	Desviacion
50	400	1	0,01	0,05	1,1	0,2	100	30%	500	309	314,98	8,25
50	400	1	0,01	0,05	1,1	0,4	100	30%	500	305	316,06	9,74
50	400	1	0,01	0,05	1,1	0,8	100	30%	500	305	317,32	9,41
25	400	1	0,05	0,5	1,1	0,2	100	30%	500	309	312,91	6,41
25	400	1	0,2	0,05	1,5	0,2	100	30%	500	305	314,68	6,93
25	400	1	0,2	1	1,1	0,2	100	30%	500	305	315,19	8,15

Figura 6.46 – Tabla resumen de las pruebas realizadas para el problema de entrenamiento con baja densidad y coste de carga 500

Conclusiones:

En vista de todas las pruebas realizadas en las pruebas con coste de carga 500, podemos concluir que el mejor valor para el porcentaje de hormigas locas es del 20 por ciento. La mejor configuración en la relación de parámetros de evaporación, aumento de evaporación y deposición de feromonas es la de evaporación 0,05 y deposición de feromonas de 0,5.

Escenario con coste de carga 1

La configuración de la que se va a partir para todas las pruebas es la siguiente: 600 iteraciones, 10 hormiga, 0,01 evaporación, 0,01 feromonas depositadas y aumento de evaporación de 1,10. En todas las pruebas se indicarán los parámetros modificados y el resto de los parámetros se mantendrán de acuerdo con esta configuración.

En primer lugar, se realizará una comparativa entre tres configuraciones con distinto porcentaje de hormigas locas, probando con un 20, 40 y 80 porciento.

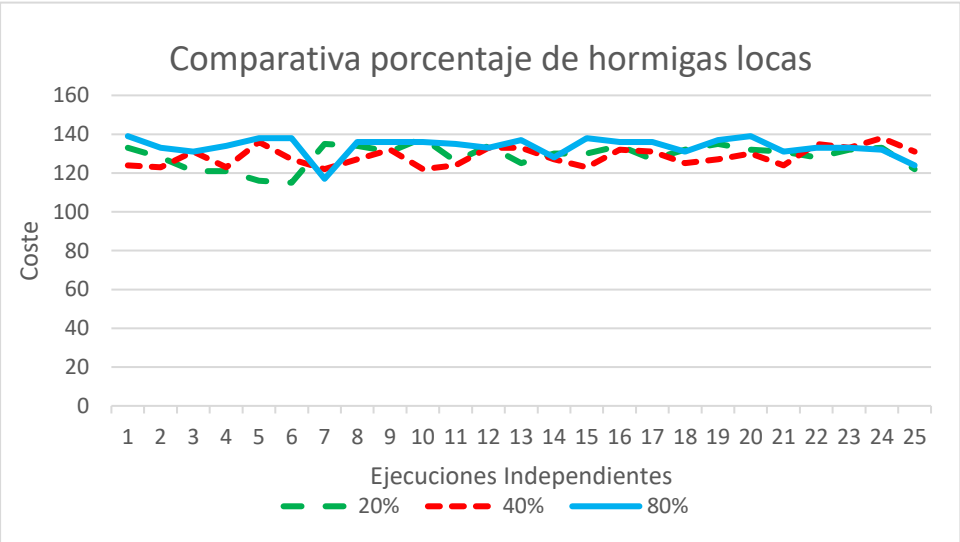

Figura 6.47 – Comparativa de distintos porcentajes de H. Locas

En la gráfica se puede apreciar que con este problema con menos densidad de vecinos que la opción del 80 por ciento es la peor igual que en el problema de más densidad, pero las otras dos configuraciones tienen unos resultados parejos. En la siguiente tabla se verán la media, desviación típica y mejor solución de las tres configuraciones.

	20 %	40 %	80 %
Media	128,92	128,52	133,64
Desviación típica	5,898	4,69	4,87
Mejor solución	115	122	117

La mejor de las tres configuraciones es la del 40 por ciento, pero le sigue muy de cerca la del 20 por ciento, diferenciándose únicamente por 1,2 puntos en la desviación típica y 0,4 puntos en la media.

Los parámetros de evaporación, aumento de evaporación y feromonas depositadas trataremos los mismos escenarios que se probaron con coste de carga 500:

1. La evaporación es baja y se depositan más feromonas de lo que se evapora. La configuración en este caso es: evaporación 0,05 – depositar feromonas 0,5.
2. La evaporación es media con un aumento de evaporación más alta y una deposición de feromonas baja en relación con estos parámetros. La configuración en este caso es: evaporación 0,20 - aumento de evaporación 1,50.
3. La evaporación es media y la deposición de feromonas alta. La configuración en este caso es: evaporación 0,20 – feromonas depositadas 1.Los resultados de la gráfica muestran que la mejor configuración es la de evaporación 0,2 con aumento de evaporación de 1,5. En la siguiente tabla se verán la media, desviación típicas y mejor solución.

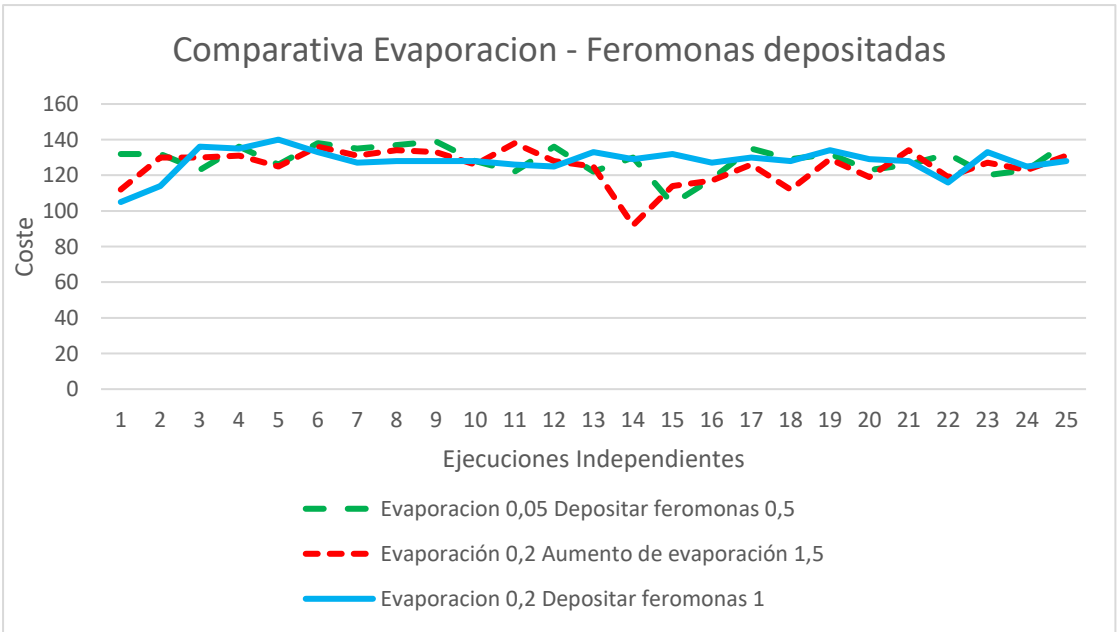

Figura 6.48 – Comparativa con distintos valores para Evaporación/Feromonas

	Evaporación 0,05 Depositar feromonas 0,5	Evaporación 0,2 Aumento de evaporación 1,5	Evaporación 0,2 Depositar feromonas 1
Media	128,108726	123,993921	127,44036
Desviación típica	7,95426929	9,7829239	7,16279275
Mejor solución	104	92	136

La mejor en media, desviación típica y mejor solución es la configuración de evaporación 0,2 y aumento de evaporación de 1,5.

Tabla recopilatoria de resultados

A continuación, se presenta una tabla resumen de los resultados de las pruebas realizadas para este problema de entrenamiento:

Ejs	Iteraciones	Hormigas	evaporación	feromonas	aumento	% locas	Nodos	Densidad	Carga	Mejor	Media	Desviacion
50	600	10	0,01	0,05	1,1	0,2	100	30%	1	115	128,92	5,89
50	600	10	0,01	0,05	1,1	0,4	100	30%	1	122	128,52	4,69
50	600	10	0,01	0,05	1,1	0,8	100	30%	1	117	133,64	4,87
25	600	10	0,05	0,5	1,1	0,2	100	30%	1	104	128,1	7,95
25	600	10	0,2	0,05	1,5	0,2	100	30%	1	92	123,99	9,78
25	600	10	0,2	1	1,1	0,2	100	30%	1	136	127,44	7,16

Figura 6.49 – Tabla resumen de las pruebas realizadas para el problema de entrenamiento con baja densidad y coste de carga 1

Conclusiones:

La comparativa de hormigas locas ha sacado unos resultados muy ajustados, con una diferencia muy pequeña entre la configuración del 20 por ciento y la del 40 por ciento; la configuración de 0,8 vuelve a quedar la última con unos resultados inferiores.

En la comparativa de evaporación, aumento de evaporación y feromonas depositadas se puede observar que la opción de aumentar la evaporación a 0,2 con un aumento de evaporación del 1,5 los resultados son más de un 10 por ciento superiores a las otras dos configuraciones probadas, quedando como la mejor configuración para este problema con coste de carga 1.

RFD

Introducción de la configuración en la aplicación

El algoritmo RFD muestra la siguiente pantalla de configuración:

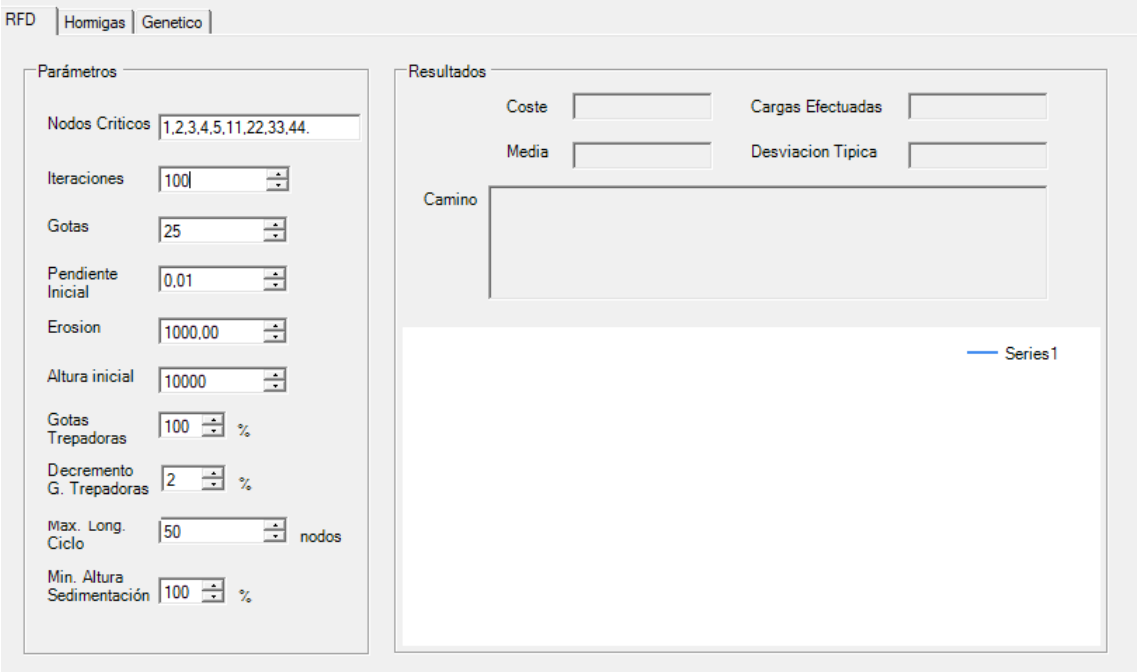

Figura 6.50 – Vista de RFD en la aplicación

Como se puede observar, los parámetros que se pueden modificar son los siguientes:

Iteraciones: Es el número de veces que el algoritmo va a realizar el bucle principal, cada iteración incluye la ejecución de la función de cálculo de coste.

Gotas: Número de gotas que se inicializan en cada nodo crítico. Es decir, si se indican 5 estados y 25 gotas, el número total de gotas sería $5x25 = 125$.

Pendiente inicial: Se utiliza cuando el terreno es plano, y no hay pendientes formadas, por lo que se usa el valor indicado aquí.

Erosión: Parámetro utilizado en la fórmula que calcula la erosión. En esa fórmula actúa como numerador que se divide por el producto de la pendiente calculada por el tamaño de la máquina y el número de nodos críticos [1,10], el resultado de esta fórmula es la erosión provocada. En otras implementaciones de RFD es usada como constante [1].

Altura inicial: Valor de altura con el que se inicializan todos los estados o nodos que no sean el destino. En otras implementaciones de RFD es usada como constante [1].

Gotas Trepadoras: Es el porcentaje inicial de gotas trepadoras. Según este porcentaje, se permitirá que las gotas puedan escalar pendientes.

Decremento de Gotas Trepadoras: En cada iteración el porcentaje de gotas trepadoras será decrementado con el valor indicado aquí.

Máxima longitud ciclo: Las gotas, al ser libres pueden llegar a realizar ciclos, pero pueden acabar llegando al nodo destino, por lo que es interesante poder indicar con que tamaño de camino que hayan realizado las gotas se eliminan las gotas que tengan ciclos. Por ejemplo, establecer este parámetro a 50 permitirá que las gotas exploren más estados que poniendo el parámetro a 0, que eliminaría las gotas en cuanto repitiesen un nodo.

Mínima Altura Sedimentación: Con este parámetro se puede indicar a partir de qué porcentaje de altura -respecto a la altura inicial- indicada se sedimenta la erosión provocada, para favorecer la formación de soluciones cuya erosión no es muy alta, es decir, si este parámetro se sitúa en 100% cada vez que se produzca una mínima erosión, se sedimentará en todos los nodos que no tengan la altura inicial (eliminando caminos despreciables), mientras que si se pone al 90%, solo se sedimentará la erosión en los nodos que tengan una altura menor al 90% de la altura inicial.

Estudio de configuración óptima

En primer lugar, para comenzar a buscar la configuración óptima de RFD entre su multitud de parámetros, se ha empezado por buscar la mejor configuración iteraciones/gotas para intentar acotar los resultados, ya que la variación de estos parámetros puede llegar a ser muy notable, llegando a afectar a la calidad de las soluciones encontradas y al tiempo de ejecución del algoritmo.

Para este primer acercamiento a encontrar la configuración óptima, se han ejecutado distintas pruebas, en las que únicamente se ha modificado el número de iteraciones y gotas, dejando el resto de parámetros fijos. El resto de los parámetros se han dejado de la siguiente forma:

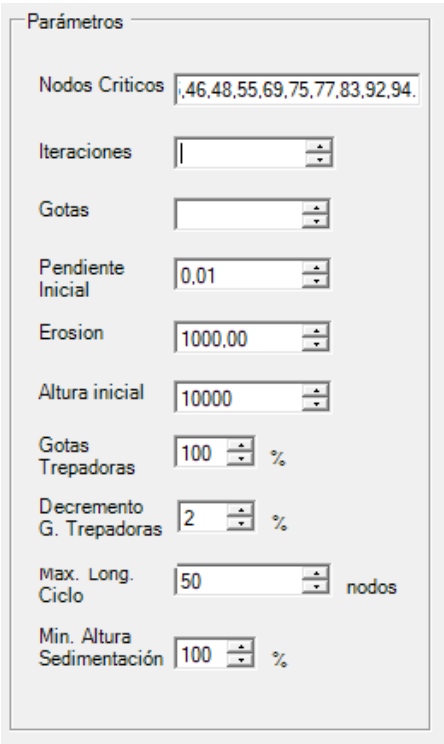

Figura 6.51 – Parámetros estándar para las pruebas

Se ha elegido esa configuración (a partir de ahora estándar), ya que tras el desarrollo de la aplicación se ha llegado a las siguientes conclusiones:

- Pendiente inicial no es muy grande (ni sería recomendable que lo fuese), ya que es la pendiente usada por las gotas al moverse por terreno plano.
- Erosión y Altura inicial van muy relacionados, y por la forma en que se calculan las pendientes y erosiones, parece recomendable que Altura inicial sea mayor que Erosión (en otras implementaciones de RFD, se usan como constantes [1]).
- Gotas trepadoras empieza con un porcentaje alto, y se decrementa un 2% en cada iteración, ya que al principio es muy necesario que las gotas puedan realizar exploraciones más amplias.
- La longitud máxima del ciclo se ha puesto en 50 nodos (mitad de tamaño de la máquina de estados), que permite que una rama o camino pueda contener ciclos (estados repetidos) hasta llegar al tamaño de 50 nodos y así pueda explorar más ampliamente.
- Por último, el tamaño mínimo en el cual se empieza a sedimentar la erosión realizada se ha dejado en 100%, evitando que se llegue a altura de nodos igual a 0 y evita reforzar caminos más despreciables.

Problema de entrenamiento con alta densidad – 70%:

Escenario con coste de carga 500

Clasificación de estas primeras pruebas:

- Más iteraciones que gotas.
- Mismas iteraciones que gotas.
- Más gotas que iteraciones.

Al realizar las distintas pruebas, se ha observado:

El mejor coste obtenido ha sido de 128, y se ha conseguido con una configuración de 100 Iteraciones - 100 Gotas, la media de resultados encontrados en 50 ejecuciones es de 166,66 con una desviación típica de 16,84.

Figura 6.52 - 50 ejecuciones de 100 Iteraciones -100 Gotas

Como se puede observar en la gráfica, se ha obtenido varias veces resultados inferiores a 150, que podrían ser considerados buenos resultados.

El siguiente mejor resultado ha sido 131, obtenido por una configuración con más iteraciones que gotas en 10 ejecuciones independientes, 1000 Iteraciones – 25 Gotas. Esta configuración se ha ejecutado menos veces debido al tiempo que tarda en completarse.

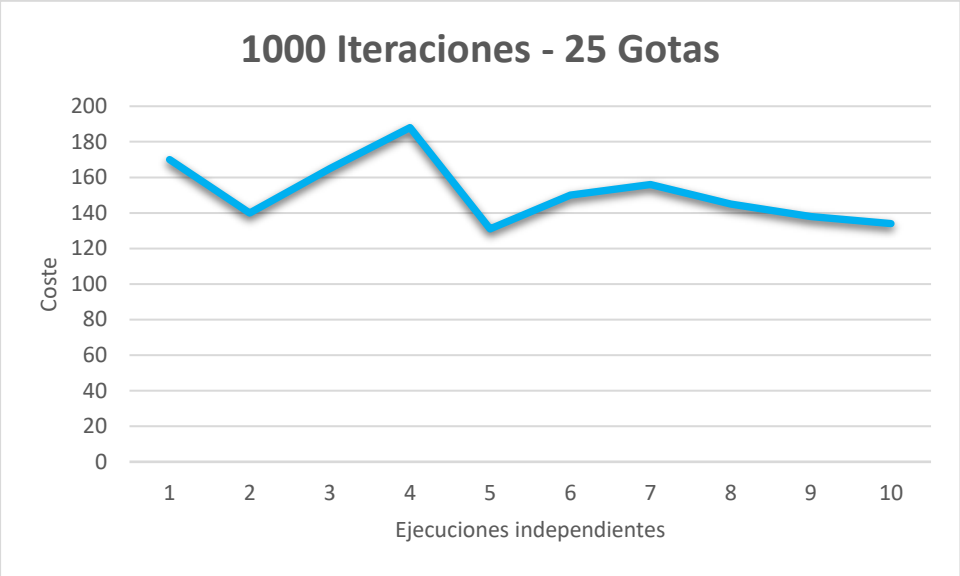

Figura 6.53 - 10 ejecuciones de 1000 Iteraciones -25 Gotas

Como la gráfica muestra, en 10 ejecuciones ha obtenido varias veces resultados cercanos a 130, y la media en estas iteraciones es de 151,7 (media más baja de todas las pruebas) con una desviación de 17,21.

El tercer mejor resultado, ha sido obtenido también por una configuración con más iteraciones que gotas, 100 Iteraciones - 25 Gotas, cuyo mejor coste encontrado en 50 Iteraciones es de 147, con media de 170 y desviación típica de 13,84.

Figura 6.54 - 50 ejecuciones de 100 Iteraciones -25 Gotas

En la gráfica se puede ver que esta configuración también ha conseguido bajar en distintas ocasiones del coste de 150.

El resto de las pruebas realizadas (ver figura 6.61), mayoritariamente con más gotas que iteraciones, no han sacado resultados menores a 147. Con una configuración de 25 Iteraciones-100 Gotas, el resultado obtenido ha sido de 199 en 50 ejecuciones, con una media de 199 y una desviación de 0.

Figura 6.55- 50 ejecuciones de 25 Iteraciones -100 Gotas

Con una configuración similar, 50 Iteraciones – 100 Gotas, el mejor resultado obtenido ha sido de 148, pero su media en 50 ejecuciones es de 178, 28 y una desviación típica de 12,73.

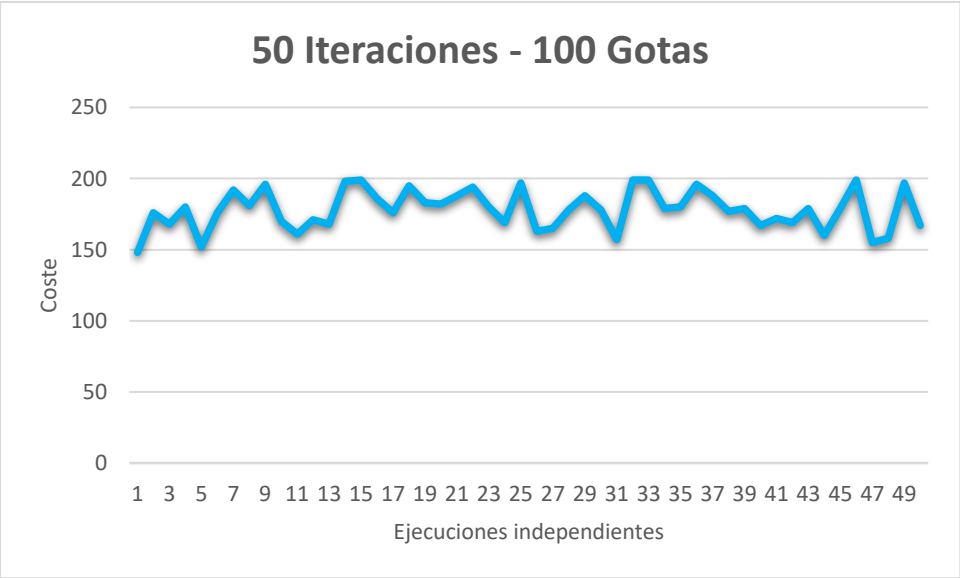

Figura 6.56 - 50 ejecuciones de 50 Iteraciones -100 Gotas

En la gráfica destaca que los resultados obtenidos en 50 ejecuciones suelen estar por encima de 150.

Tras ver los resultados anteriores, se ha decidido realizar una comparativa entre las 3 mejores configuraciones en 10 ejecuciones, para sacar una comparativa real con la configuración de 1000 Iteraciones – 25 Gotas.

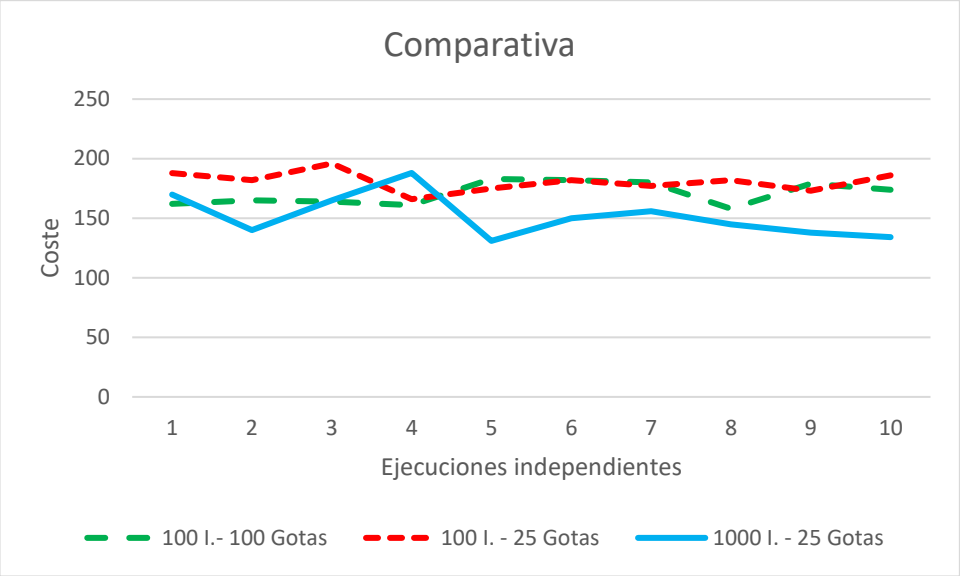

Figura 6.57 – Comparativa de las mejores configuraciones iteraciones/gotas en 10 ejecuciones

La gráfica muestra que la configuración que mejores resultados saca -en general- es 1000 iteraciones – 25 gotas. Podemos concluir que esta configuración es la más óptima de las pruebas realizadas aunque no llegue a obtener el mejor coste de todas las pruebas (coste obtenido una vez en 50 ejecuciones por la configuración 100 iteraciones – 100 gotas), ya que es más robusto obteniendo costes cercanos al mejor en la mayoría de las ejecuciones.

De las anteriores pruebas también podemos extraer que el algoritmo necesita realizar más iteraciones o estar más tiempo ejecutándose, hecho que le permite explorar más y encontrar mejores soluciones.

Hay que destacar que el aumento de iteraciones implica un menor impacto en el tiempo que tarda en realizar una ejecución, que el aumento de gotas. El aumento de gotas afecta mucho más al tiempo de ejecución, debido a que en cada iteración el algoritmo espera que todas las gotas que superen la criba de ciclos y longitud, cubran todos los nodos críticos. Es necesario recordar que el número de gotas total no es el que se indica en la entrada, ya que se deposita el número de gotas indicado en todos los nodos críticos, quedando:

$$Total\ Gotas = param.gotas * num.nodos\ críticos$$

Tras las pruebas orientadas a iteraciones-gotas, se va a probar a cambiar los parámetros definidos como estándar (en la configuración obtenida anteriormente) para comprobar si se pueden mejorar los resultados.

Las pruebas realizadas consisten en:

- Modificación de porcentaje de gotas trepadoras y decremento de este.
- Modificación de longitud permitida para ramas con nodos repetidos.
- Modificación de altura mínima a partir de la que se sedimenta.

La modificación del porcentaje de gotas trepadoras puede llegar a dar una mayor exploración del grafo, pero puede romper los caminos encontrados, o no llegar a encontrar alguno decente. Las pruebas realizadas que modifican este parámetro no han obtenido mejores resultados que los conseguidos por la configuración estudiada:

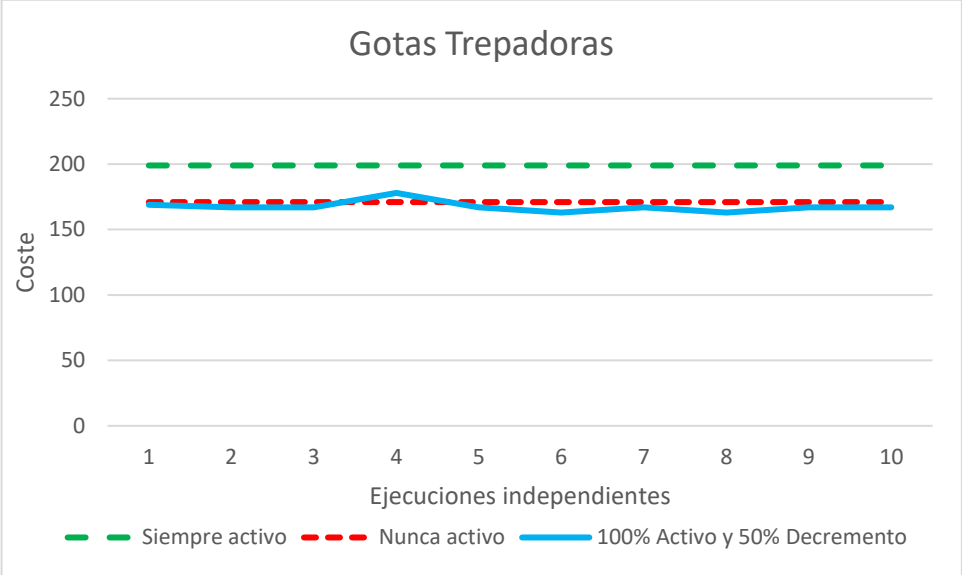

Figura 6.58- Comparativa porcentaje de gotas trepadoras

Como se ve en la figura 6.58, ninguna de las configuraciones mostradas tiene mejores resultados a los conseguidos anteriormente, con un mejor coste de 163 (configuración con 100% Activo y 50% Decremento), y medias superiores a 167 (ver figura 6.61).

Cambiar de la longitud del ciclo puede tener gran impacto en la eficiencia del algoritmo, ya que si la rama no permite nodos repetidos el algoritmo elimina esas gotas, haciendo que las iteraciones sean más rápidas. Tener una rama con nodos repetidos más larga, favorece a la exploración. Los resultados han sido:

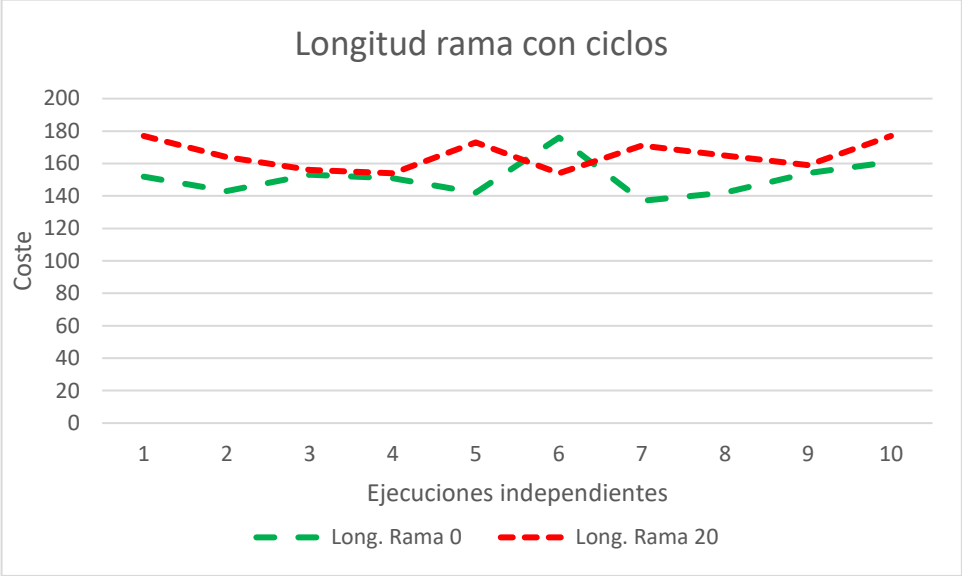

Figura 6.59 – Comparativa variación longitud de ciclo

Tal como se puede observar en la gráfica, dar valor 0 a la longitud de rama con ciclos obtiene unos resultados muy parecidos a los mejores conseguidos por la mejor configuración obtenida hasta este momento, con un mejor coste de 137, una media de 151,1 y una desviación típica de 10,77. Además, como se ha comentado antes, tener la longitud de rama que permite nodos repetidos a 0 puede ser muy interesante, ya que hace que el algoritmo sea más eficiente y pueda realizar ejecuciones más rápido.

Alterar el valor establecido como estándar para la altura mínima a partir de la que se sedimenta, permite que se exploren caminos menos favorables (con la sedimentación a partir de la altura completa, no llegarían a contemplarse):

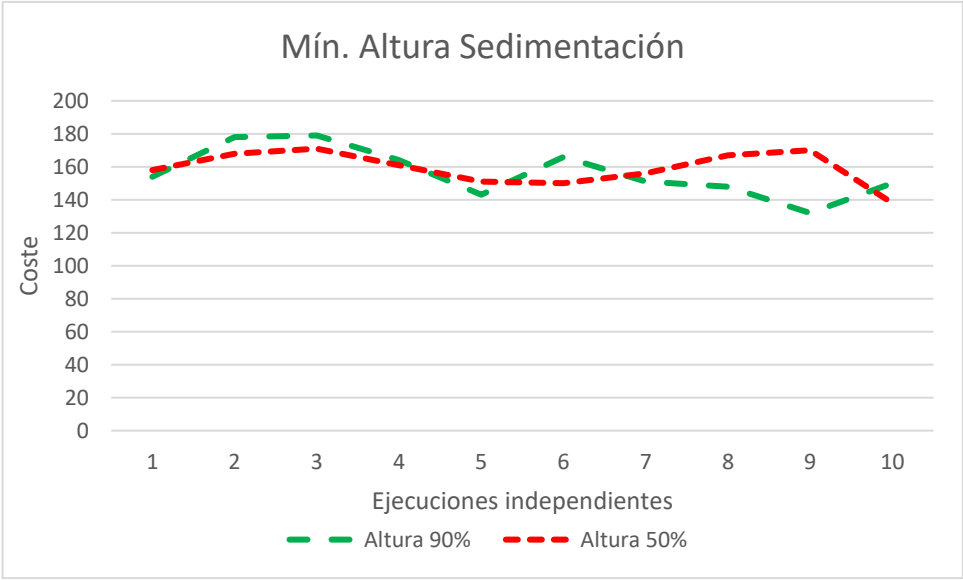

Figura 6.60 – Comparativa modificación Mín. Altura Sedimentación

Ambas configuraciones llegan a conseguir resultados de menos de 140, bastante cercanas a los mejores costes obtenidos por la mejor configuración encontrada, pero la media (ambas superiores a 156) y los resultados en general, no son tan buenos como los de la configuración encontrada con los parámetros estándar (ver figura 6.61).

Tabla recopilatorio de resultados

A continuación, se presenta una tabla resumen de los resultados de las pruebas realizadas para este problema de entrenamiento con coste de carga 500 (los parámetros que no aparecen en la tabla se han mantenido invariables de los estándares definidos al inicio de esta sección):

Ejs.	Iteraciones	Gotas	% G. Trepadora	% Decremento G. Trepadora	Max. Long. Ciclo	% Min. Altura Sedimentación	Nodos	Densidad	C. carga	Mejor	Media	Desviación
50	100	100	100	2	50	100	100	70%	500	128	166,66	16,84
50	100	50	100	2	50	100	100	70%	500	148	176,84	12,49
50	100	25	100	2	50	100	100	70%	500	147	170,98	13,84
50	25	100	100	2	50	100	100	70%	500	199	199,00	0,00
50	50	100	100	2	50	100	100	70%	500	148	178,28	13,73
50	25	25	100	2	50	100	100	70%	500	194	198,50	1,50
50	50	50	100	2	50	100	100	70%	500	155	183,80	11,48
10	1000	25	100	2	50	100	100	70%	500	131	151,70	17,21
10	100	1000	100	2	50	100	100	70%	500	140	167,40	11,53
10	1000	25	100	0	50	100	100	70%	500	199	199,00	0,00
10	1000	25	0	0	50	100	100	70%	500	171	171,00	0,00
10	1000	25	100	50	50	100	100	70%	500	163	167,50	3,93
10	1000	25	100	2	0	100	100	70%	500	137	151,10	10,77
10	1000	25	100	2	20	100	100	70%	500	154	165,00	8,65
10	1000	25	100	2	100	100	100	70%	500	142	161,20	13,32
10	1000	25	100	2	200	100	100	70%	500	128	161,00	18,59
10	1000	25	100	2	50	90	100	70%	500	132	156,50	14,31
10	1000	25	100	2	50	50	100	70%	500	138	159,00	10,05

Figura 6.61 – Tabla resumen de las pruebas realizadas para el problema de entrenamiento con alta densidad y coste de carga 500

Conclusión:

La mejor configuración encontrada para este problema es 1000 Iteraciones – 25 gotas, con los parámetros estándar definidos al comienzo de este punto, ya que los resultados que encuentra son bastante cercanos al mejor encontrado en la realización de todas las pruebas, y en su conjunto los resultados obtenidos en todas las ejecuciones se podrían considerar como buenas soluciones. Tal como se comentaba antes, también se podría considerar la configuración 1000 Iteraciones – 25 gotas, con longitud de rama con ciclo 0 y el resto de parámetros estándar, ya que saca resultados muy cercanos a la configuración óptima en menor tiempo.

Escenario con coste de carga 1:

Clasificación de estas primeras pruebas:

- Más iteraciones que gotas.
- Mismas iteraciones que gotas.
- Más gotas que iteraciones.

La mejor solución obtenida en las pruebas realizadas tiene un coste de 60, y es obtenida por la configuración 100 Iteraciones – 100 Gotas, cuya media en 50 ejecuciones es 74,28 y una desviación típica de 5,5.

Figura 6.62 - 50 ejecuciones de 100 Iteraciones -100 Gotas

La gráfica muestra que obtiene la mayoría de las ejecuciones valores inferiores a 80, por lo que asegura conseguir resultados decentes en casi todas las ejecuciones.

El siguiente mejor coste 66, ha sido obtenido por la mayoría de las pruebas realizadas, entre las que se encuentran:

- 100 iteraciones – 25 Gotas, con una media en 50 ejecuciones de 82,28 y una desviación típica de 7,31.
- 100 iteraciones – 50 Gotas, con una media en 50 ejecuciones de 80,48 y una desviación típica de 8,45.

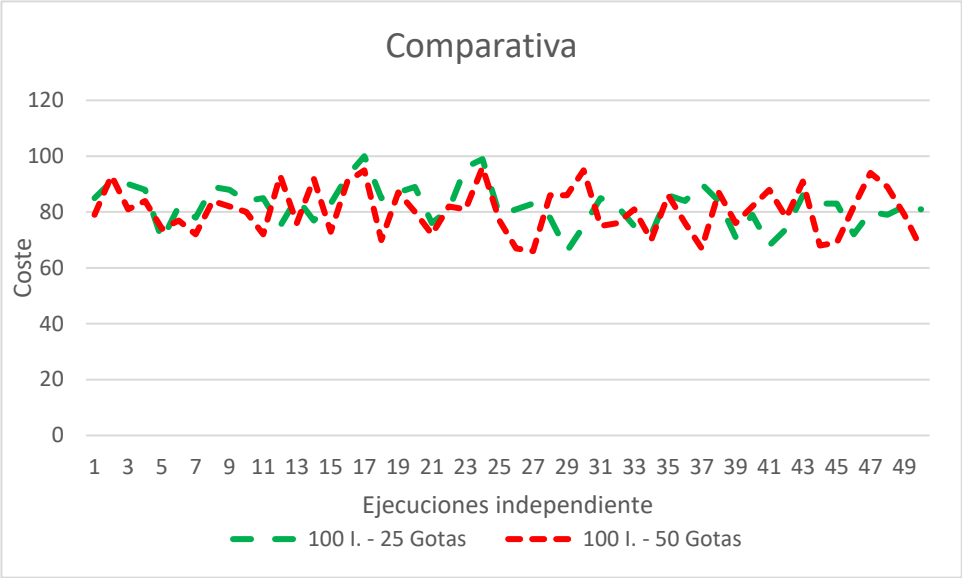

Figura 6.63 – Comparativa 100 I. – 25 Gotas con 100 I. – 50 Gotas

Como se puede ver en la gráfica, ambas configuraciones sacan resultados muy parejos a lo largo de las ejecuciones, y su media es bastante parecida.

Otra configuración que obtiene el coste de 66 es 1000 Iteraciones – 25 gotas, en un total de 10 ejecuciones independientes, ha obtenido una media de 74,2 y una desviación típica entre los resultados de 3,54.

Figura 6.64 – 10 ejecuciones 1000 Iteraciones – 25 Gotas

La gráfica de los mejores resultados de las ejecuciones no sube en ninguna de ellas a 80, por lo que en las 10 ejecuciones ha obtenido resultados a bastante buenos.

La configuración que peor se comporta -al igual que en el estudio sin cargas- es la configuración con 25 iteraciones – 100 gotas, que obtiene un mejor coste de 84, con una media de 93,28.

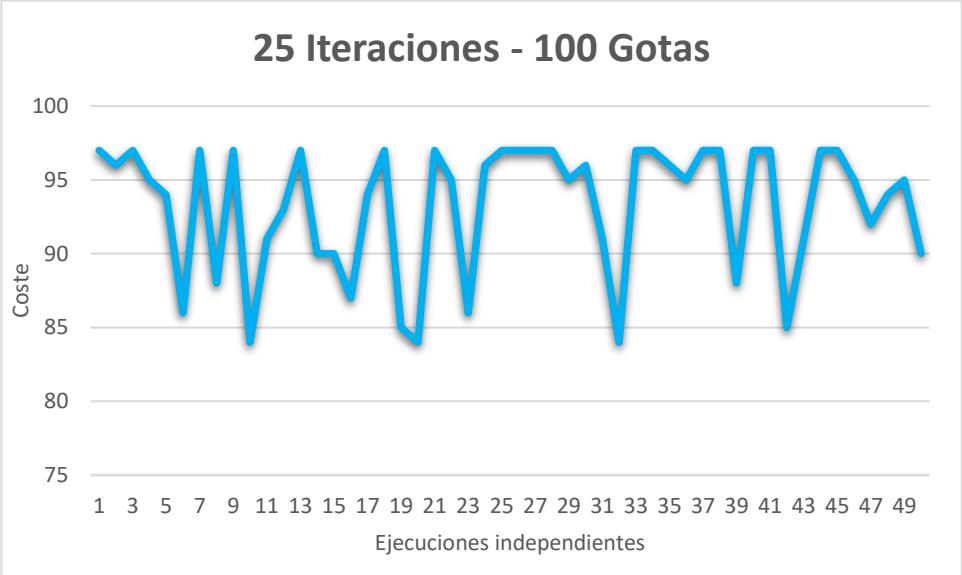

Figura 6.65 – 50 ejecuciones de 25 Iteraciones -100 Gotas

La gráfica de esta configuración muestra que el coste de todas las mejores soluciones encontradas en las 50 ejecuciones independientes realizadas es siempre superior a 80, mostrándose claramente peor que el resto de las configuraciones probadas.

Con los resultados obtenidos, se va a comparar las mejores configuraciones en igualdad de condiciones: 100 Iteraciones – 100 Gotas y 1000 Iteraciones – 25 Gotas en 10 ejecuciones independientes.

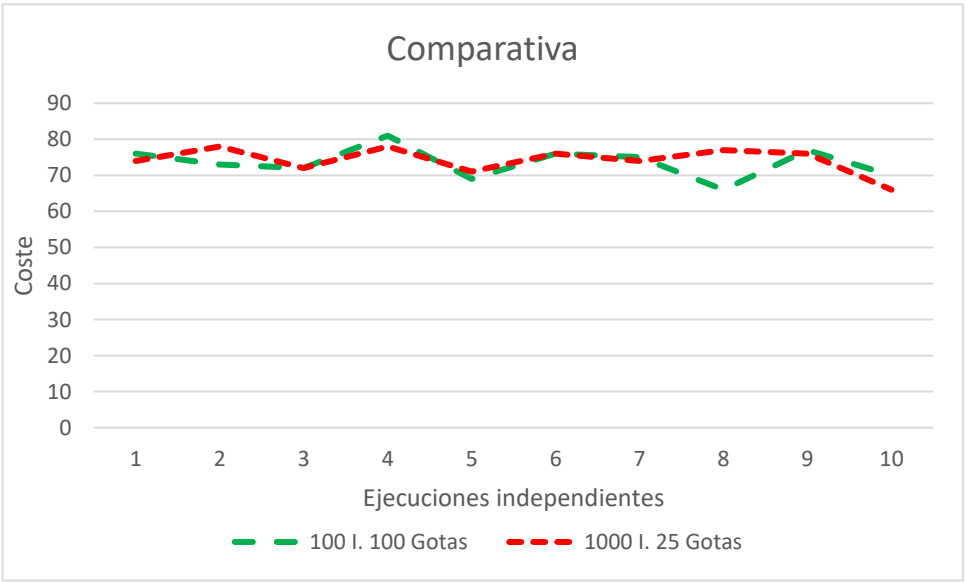

Figura 6.66 – Comparativa entre las mejores configuraciones iteraciones/gotas

La configuración de 100 Iteraciones – 100 Gotas tiene menor media 73,5 y desviación típica de 4,17, frente a una media de 74,2 y una desviación de 3,54 conseguidos por la configuración 1000 Iteraciones – 25 Gotas. Se tomará como mejor configuración 100 Iteraciones – 100 Gotas para el problema con cargas, ya que además de tener mejor media, en la prueba de 50 ejecuciones ha llegado a obtener el mejor resultado de todas las pruebas realizadas y además el tiempo que tarda en realizar una ejecución completa es menor.

Al igual que en el apartado anterior donde se estudia la mejor configuración para el problema sin cargas, tras estudiar la mejor configuración iteraciones-gotas, se va a probar a realizar variaciones de los parámetros definidos como estándar para comprobar si se puede obtener un mejor resultado.

Las pruebas realizadas consisten en:

- Modificación de porcentaje de gotas trepadoras y decremento de este.
- Modificación de longitud permitida para ramas con nodos repetidos.
- Modificación de altura mínima a partir de la que se sedimenta.

La modificación del porcentaje de gotas trepadoras puede llegar a dar una mayor exploración del grafo, pero puede romper los caminos encontrados, o no llegar a encontrar alguno decente:

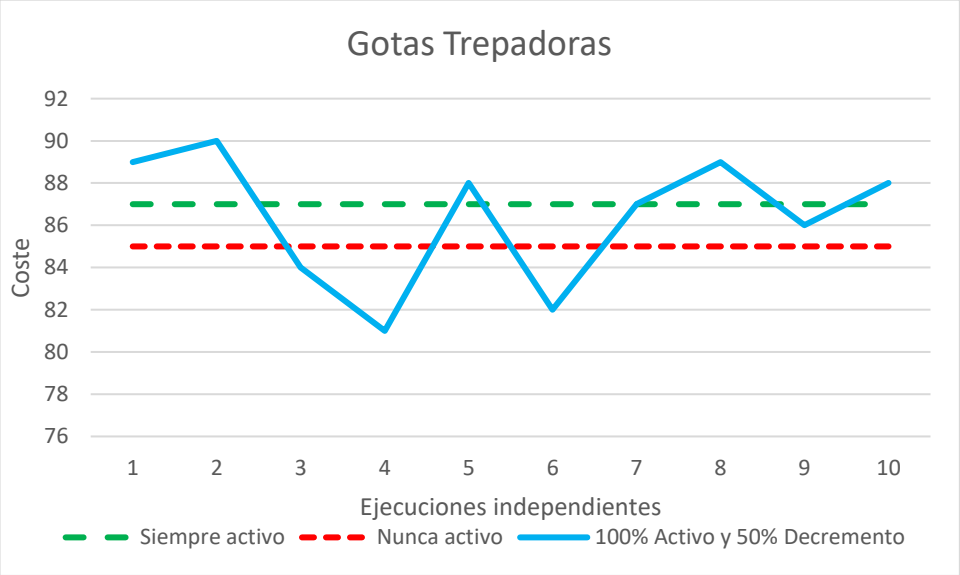

Figura 6.67 – Comparativa variaciones de porcentaje de gotas trepadoras

La variación de la configuración estándar de los parámetros asociados a gotas trepadoras no consigue mejores resultados que la configuración óptima obtenida en el estudio de iteraciones-gotas. El mejor resultado obtenido por estas configuraciones es de 81, y medias superiores a 85 (ver figura 6.70).

Modificar de la longitud del ciclo puede tener gran impacto en la eficiencia del algoritmo, ya que si la rama no admite nodos repetidos el algoritmo elimina esas gotas, haciendo que las iteraciones sean más rápidas. Tener una rama con nodos repetidos más larga, favorece a la exploración:

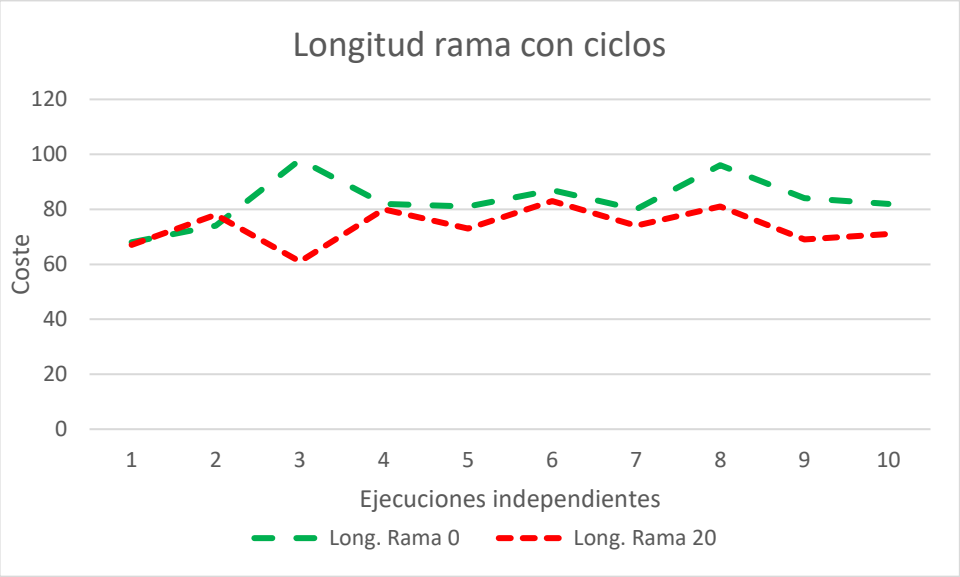

Figura 6.68 – Comparativa variación longitud de ciclo

Como se puede ver en la gráfica, la longitud de la rama 20, siendo el problema sobre el que se trabaja una máquina de estados de 100 con 20 nodos críticos, es una configuración que saca resultados muy parecidos a la configuración óptima con los parámetros estándar, pero sacando en 10 ejecuciones un coste de 61, que es casi parejo al mejor resultado encontrado (coste 60 en 50 ejecuciones) en las pruebas realizadas, además de tener la menor media de 73,09 y una desviación típica de 6,59.

Cambiar el valor para la altura mínima a partir de la que se sedimenta, incentiva la exploración de caminos menos favorables (con la sedimentación a partir de la altura completa, no llegarían a contemplarse):

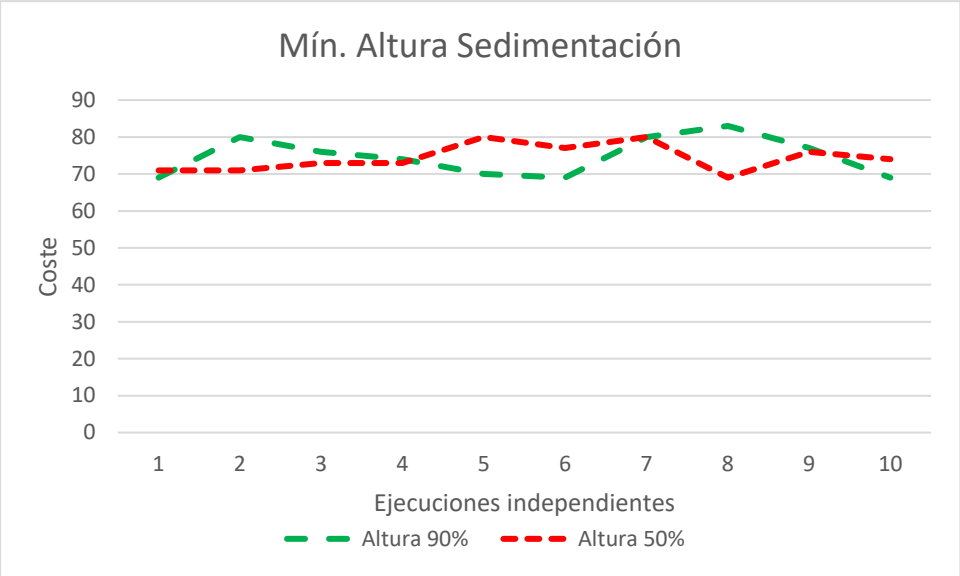

Figura 6.69 – Comparativa modificación Mín. Altura Sedimentación

Los resultados que se observan en la gráfica no son mejores que los de la configuración óptima con parámetros estándar o que esta con longitud de rama que permite ciclos de 20 y el resto de los parámetros estándar. Las configuraciones presentadas en la figura 6.67 obtienen una mejor solución de 69, la configuración con Mín. Altura 90% tiene una media de 74,37 y una desviación de 5,05, y la configuración con 50% como altura mínima a partir de la que se sedimenta tiene una media de 74,23 y una desviación típica de 3,75.

Tabla recopilatorio de resultados

A continuación, se presenta una tabla con los resultados obtenidos en las pruebas realizadas para este problema de entrenamiento en el escenario con coste de carga 1 (los parámetros que no aparecen en la tabla se han mantenido invariables de los estándares definidos al inicio de esta sección):

Ejs.	Iteraciones	Gotas	% G. Trepadora	% Decremento G. Trepadora	Max. Long. Ciclo	% Min. Altura Sedimentación	Nodos	Densidad	C. carga	Mejor	Media	Desviación
50	100	100	100	2	50	100	100	70%	1	60	74,28	5,56
50	100	50	100	2	50	100	100	70%	1	66	80,48	8,45
50	100	25	100	2	50	100	100	70%	1	66	82,28	7,31
50	25	100	100	2	50	100	100	70%	1	84	93,28	4,37
50	50	100	100	2	50	100	100	70%	1	69	80,44	5,97
50	25	25	100	2	50	100	100	70%	1	85	98,74	4,96
50	50	50	100	2	50	100	100	70%	1	68	81,80	8,17
10	1000	25	100	2	50	100	100	70%	1	66	74,20	3,54
10	25	1000	100	2	50	100	100	70%	1	78	85,40	3,88
10	100	1000	100	2	50	100	100	70%	1	71	75,88	3,00
10	100	100	100	0	50	100	100	70%	1	87	87,00	0,00
10	100	100	0	0	50	100	100	70%	1	85	85,00	0,00
10	100	100	100	50	50	100	100	70%	1	81	86,40	2,94
10	100	100	100	2	0	100	100	70%	1	68	83,20	8,55
10	100	100	100	2	20	100	100	70%	1	61	73,09	6,59
10	100	100	100	2	100	100	100	70%	1	72	76,80	2,74
10	100	100	100	2	200	100	100	70%	1	74	77,50	2,69
10	100	100	100	2	20	90	100	70%	1	69	74,37	5,06
10	100	100	100	2	20	50	100	70%	1	69	74,23	3,75
10	100	100	100	2	20	25	100	70%	1	67	78,80	6,34
10	100	100	100	2	20	0	100	70%	1	74	78,60	3,38

Figura 6.70 – Tabla resumen de las pruebas realizadas para el problema de entrenamiento con alta densidad y coste de carga 1

Conclusión:

La mejor configuración obtenida para el problema abordado con cargas es 100 Iteraciones – 100 Gotas, con el parámetro que controla la longitud de la rama que permite ciclos a 20 y el resto de los parámetros estándar. Es mejor solución que la obtenida en el punto iteraciones-gotas ya que en 10 ejecuciones consigue llegar a valores muy cercanos al valor mínimo encontrado (coste 60 en 50 ejecuciones) y el tiempo en el que se termina en completar una ejecución es menor.

Problema de entrenamiento con baja densidad – 30%

Con las configuraciones optimas obtenida en el anterior problema de entrenamiento, se va a enfrentar a un problema con una densidad de conexiones notablemente menor, para comprobar cómo se comporta y estudiar si se pueden conseguir configuraciones más competentes para un grafo con densidad baja.

Escenario con coste de carga 500:

La mejor configuración obtenida para el escenario que desfavorecía las cargas es 1000 Iteraciones – 25 Gotas, con parámetros estándar. Se va a partir de esa configuración, realizando modificaciones en los parámetros que mayor impacto tuvieron en las pruebas previas.

El parámetro que mayores cambios mostró, y que mejores resultados obtuvo, incluso consiguiendo una solución bastante competitiva con un coste en tiempo de ejecución menor, es la longitud de ciclo:

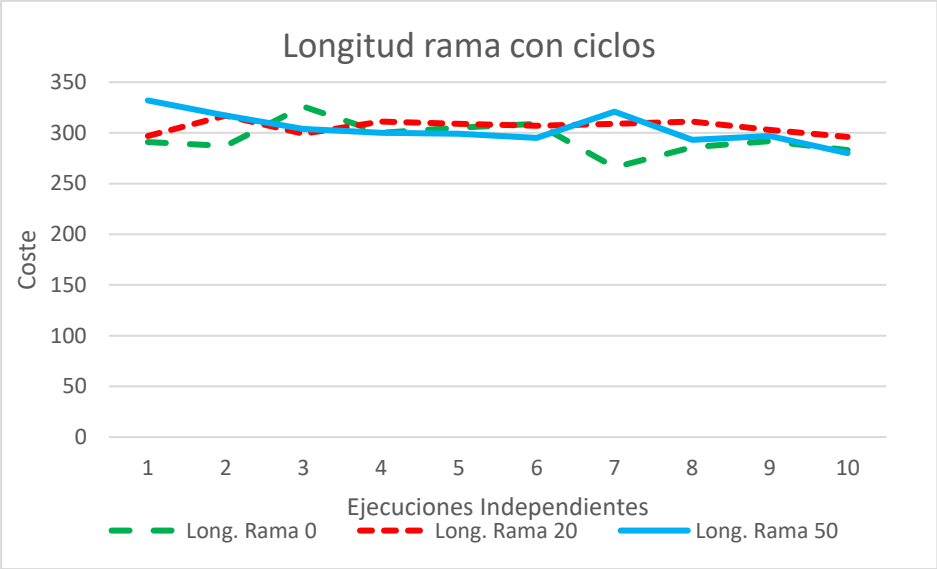

Figura 6.71 – Comparativa de las variaciones de longitud de ciclo más significativas

En la figura 6.71, se han mostrado los valores que mejores resultados han obtenido (el resto de valores probados aparecen al final del apartado en la figura 6.73) entre los probados: 0, 20, 50, 75, 100, y 200. En esta prueba, se ha observado que a menor valor de longitud de rama que permite nodos repetidos, se consiguen mejores resultados. El mejor resultado obtenido por la configuración con longitud de camino que contiene ciclos 0 ha sido de 266, con una media de 293,68 en 10 ejecuciones y una desviación típica de 15,60. El siguiente mejor resultado lo ha conseguido la configuración con el parámetro que controla la longitud de ciclos a 50 (mejor configuración obtenida en el anterior problema), con una media de 303,11 en 10 ejecuciones y una desviación de 14,52. El tercer mejor coste hallado es de 296, con una media de 305,75 y una desviación de 6,54, que ha sido encontrado por la configuración de longitud de ciclo

20, configuración que presenta la relación media/desviación más baja, pero como se observa en la gráfica, en general consigue los peores resultados.

El resto de variaciones del parámetro longitud de ciclo probadas han obtenido peores resultados, con medias superiores a 312 -10 puntos superior a las medias de los mejores anteriormente vistos (ver figura 6.75).

Tras esta prueba, por ahora la configuración óptima obtenida para un escenario que desfavorece las cargas en un grafo con baja densidad, es diferente a la configuración óptima obtenida en el problema previo.

Las siguientes pruebas se realizarán con la configuración 1000 Iteraciones – 25 Gotas, longitud de ciclos 0, y el resto de parámetros estándar, salvo el parámetro objeto de prueba.

Como se comentó anteriormente, un parámetro que puede variar la exploración del grafo es el porcentaje de gotas trepadoras, permitiendo que en el inicio del problema se puedan explorar más caminos, pero si se mantiene activo puede llegar a ser contraproducente:

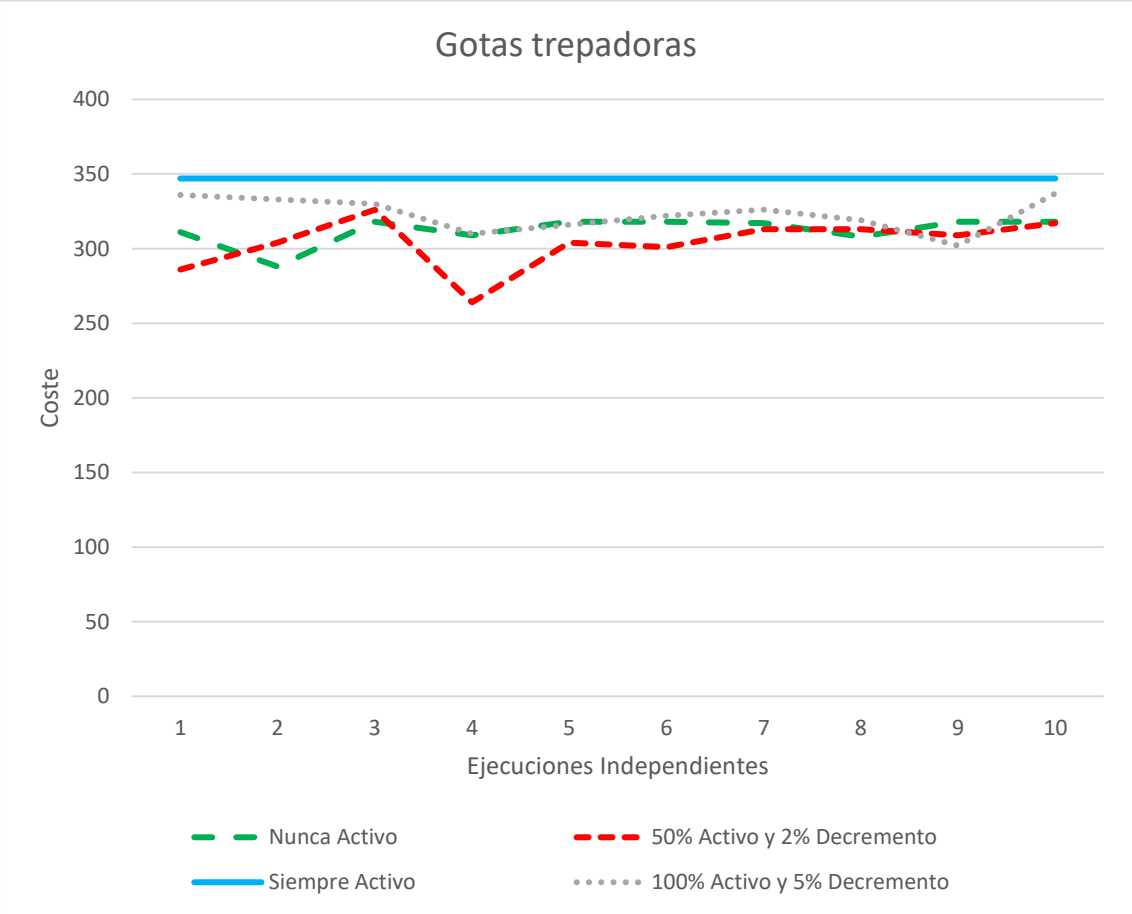

Figura 6.72 – Comparativa variaciones de gotas trepadoras

El mejor resultado de las variaciones ha sido de 264 (mejor resultado para este escenario), por la configuración que tiene en un inicio el porcentaje de gotas

trepadoras al 50% y decrementa 2% cada iteración, con una media de 302,72 y una desviación de 16,64. El resto de combinaciones ha obtenido una media superior a 312, que supera en casi 10 puntos a la media del resultado anterior y quedando a la altura de los peores resultados de este escenario (ver figura 6.75).

Se va a comparar la configuración conseguida al principio del escenario (1000 Iteraciones – 25 Gotas, long. ciclo 0 y resto de parámetros estándar), con la obtenida en la anterior prueba (1000 Iteraciones – 25 Gotas, long. ciclo 0, porcentaje de gotas trepadoras 50% y 2% de decremento, el resto de parámetros estándar), pues parece que pueden llegar a competir, aunque la media de este último sea superior en 5 puntos:

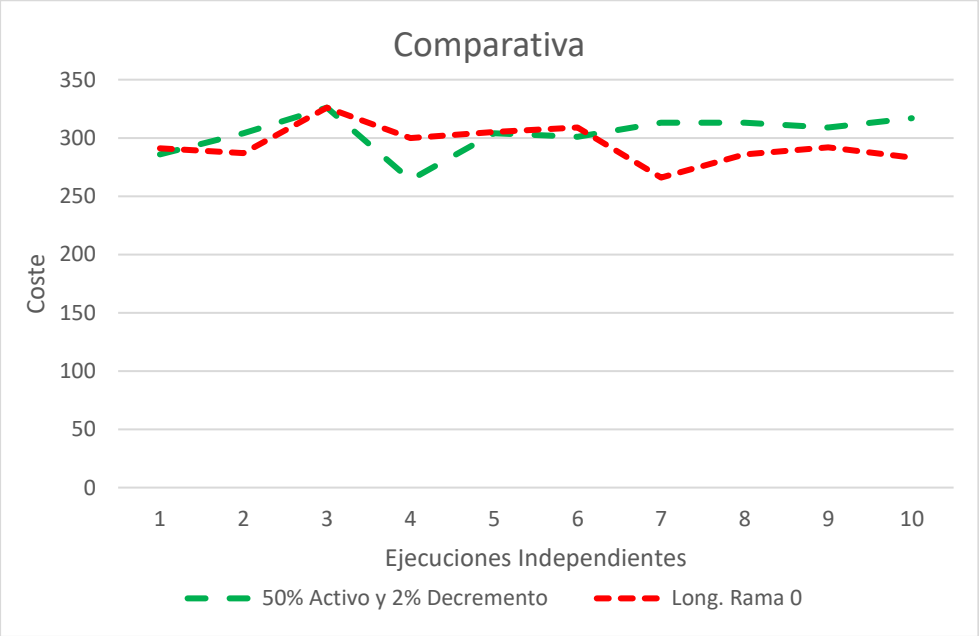

Figura 6.73 – Comparativa entre las variaciones más destacables

Aunque la configuración con porcentaje de gotas trepadoras al 50% y decremento de 2% obtenga el mejor valor, suele conseguir resultados peores a la configuración que tiene longitud de rama 0, y el resto de parámetros estándar, por lo que es más robusta esta última.

Para las siguientes variaciones de parámetros se seguirá utilizando la configuración 1000 Iteraciones – 25 Gotas, long. ciclo 0 y resto de parámetros estándar, que es la que -en general- mejores resultados ha obtenido a lo largo de este problema penalizando las cargas.

Para concluir con este escenario, se va a modificar el parámetro de altura mínima, que puede afectar positivamente a la exploración, favoreciendo la búsqueda en caminos más despreciables:

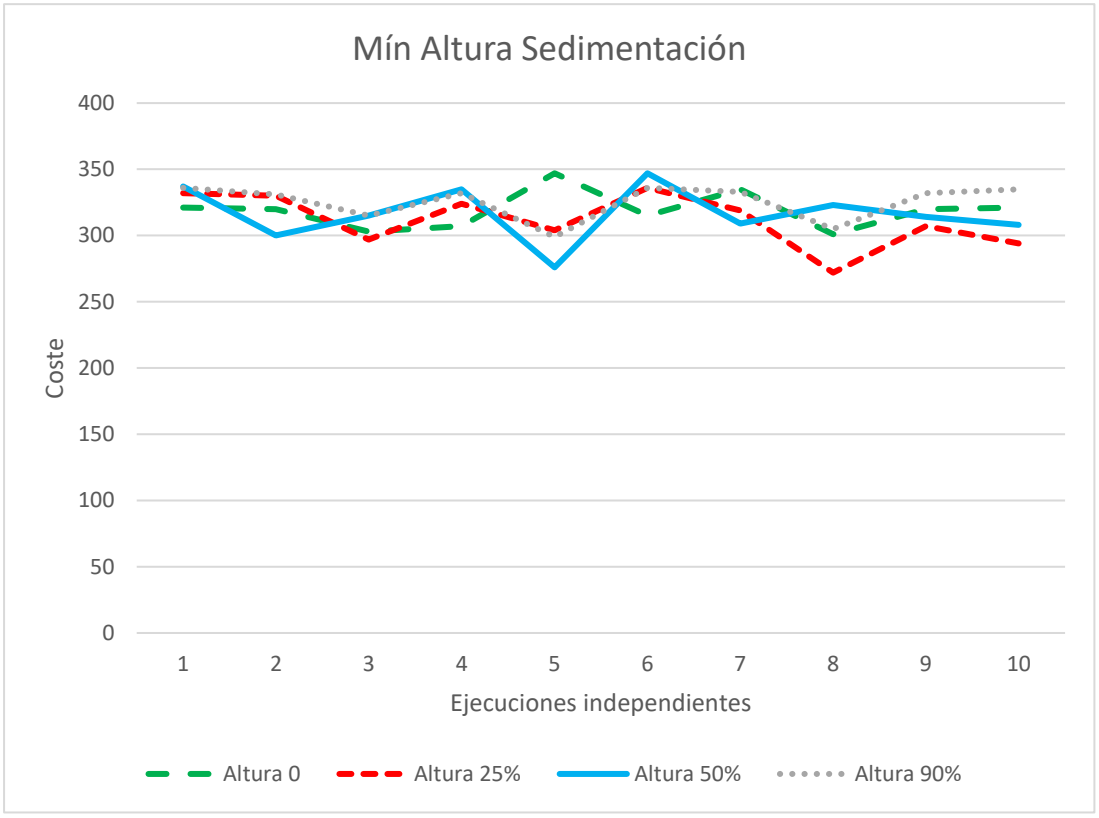

Figura 6.74 – Comparativa de modificaciones de Mín. Altura Sedimentación

Las configuraciones con mínima altura de sedimentación 25% y 50% han conseguido en una ocasión valores de 272 y 276 respectivamente, que son relativamente cercanos al mejor obtenido en este escenario (coste de 264). Pero las medias del conjunto de pruebas que se observan en la gráfica, son todas superiores a 310 llegando incluso a superar una media 324 en 10 ejecuciones (ver figura 6.75).

Tabla recopilatoria de resultados

A continuación, se presenta una tabla con los resultados de las pruebas ejecutadas para este problema de entrenamiento con coste de carga 500 (los parámetros que no aparecen en la tabla se han mantenido invariables de los estándares definidos al inicio de esta sección):

Ejs.	Iteraciones	Gotas	% G. Trepadora	% Decremento G. Trepadora	Max. Long. Ciclo	% Min. Altura Sedimentación	Nodos	Densidad	C. carga	Mejor	Media	Desviación
10	1000	25	100	2	0	100	100	30%	500	266	293,68	15,60
10	1000	25	100	2	20	100	100	30%	500	296	305,76	6,55
10	1000	25	100	2	50	100	100	30%	500	280	303,12	14,52
10	1000	25	100	2	75	100	100	30%	500	301	316,22	13,75
10	1000	25	100	2	100	100	100	30%	500	277	312,38	18,26
10	1000	25	100	2	200	100	100	30%	500	290	313,12	15,56
10	1000	25	0	0	0	100	100	30%	500	288	312,03	8,98
10	1000	25	100	0	0	100	100	30%	500	264	302,72	16,65
10	1000	25	50	2	0	100	100	30%	500	347	347,00	0,00
10	1000	25	100	5	0	100	100	30%	500	302	322,72	10,95
10	1000	25	100	2	0	0	100	30%	500	301	318,45	13,38
10	1000	25	100	2	0	25	100	30%	500	272	310,26	19,31
10	1000	25	100	2	0	50	100	30%	500	276	315,17	19,45
10	1000	25	100	2	0	90	100	30%	500	299	324,85	13,09

Figura 6.75 – Tabla resumen de las pruebas realizadas para el problema de entrenamiento con baja densidad y coste de carga 500

Conclusión:

La mejor configuración obtenida en este problema de entrenamiento, penalizando las cargas es 1000 Iteraciones – 25 Hormigas, longitud de ciclo a 0, y el resto de parámetros estándar. Esta configuración varía de la óptima obtenida en el problema de entrenamiento previo en la longitud de ciclo, por lo que en grafos más densos se utilizará la configuración anterior, y en grafos menos poblados se utilizará esta (escenarios donde se penalicen las cargas). Por último, como ya se ha comentado anteriormente, un punto a destacar sobre la configuración hallada es que gracias a la limitación de rama que contiene nodos repetidos es 0, el algoritmo es más eficiente y puede realizar ejecuciones completas más rápido.

Escenario con coste de carga 1:

Para este escenario, se va a usar como base la configuración óptima obtenida en el problema de entrenamiento previo con coste de carga 1, que es de 100 Iteraciones – 100 Gotas, con el parámetro que controla la longitud de ciclo a 20, y el resto de parámetros estándar.

Como en el escenario anterior, se va a comenzar por modificar el parámetro que controla el tamaño de las ramas que contienen ciclos, pues suele realizar cambios notorios y suelen ser positivos:

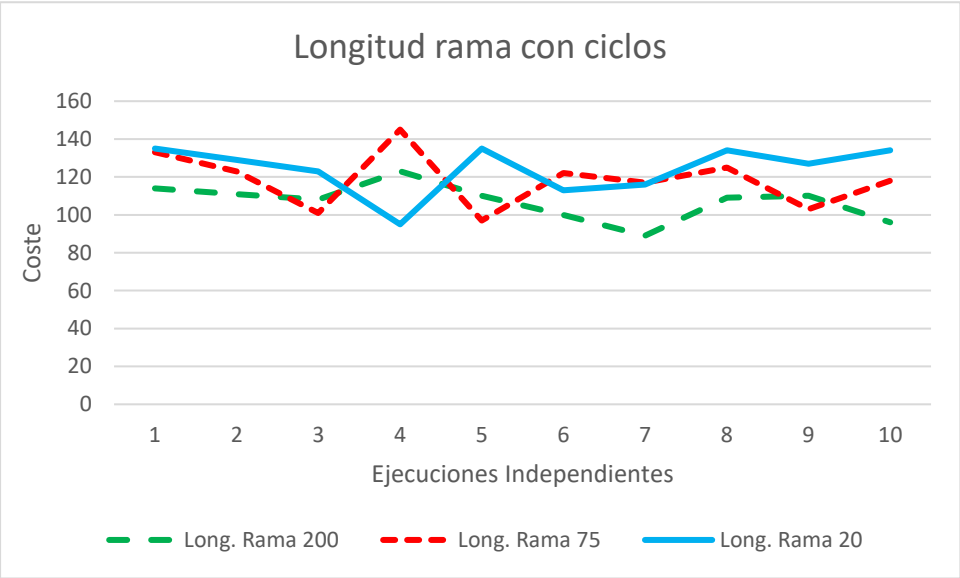

Figura 6.76 – Comparativa de las variaciones de longitud de ciclo más relevantes

En la gráfica se muestran las configuraciones que han resultado los costes más bajos: coste de 89 por la configuración con Long. Ciclo 200, coste de 95 por la configuración con Long. Ciclo 20 y coste de 97 por la configuración de Long. Ciclo 75. Como se puede observar, la variación que mejores resultados obtiene es la que tiene como límite de longitud de caminos que contienen nodos repetidos con valor 200, con una media de 106,18 y desviación 9,15. La configuración con límite 75 para los caminos que contienen ciclos, ha llegado a conseguir el valor de 95, pero como se puede comprobar en la gráfica, ha sido una vez en 10 ejecuciones, y el resto de valores son notablemente superiores a 100, su media en 10 ejecuciones es de 122,71 y la desviación típica de 12,25. La configuración con valor 20 en el parámetro que controla la longitud de los caminos que tienen nodos repetidos -además de obtener el tercer mejor coste- tiene una media de 116,70 y una desviación de 14,13.

Otra configuración que destacar, que no aparece en la figura 6.74 es la configuración con máxima longitud de rama con nodos repetidos a 100, que obtiene un mejor coste de 108, una media de 114,24 y una desviación de 4,27 en 10 ejecuciones, con una media superior a las configuraciones que han conseguido el segundo y tercer puesto respecto al mejor coste, pero sus estadísticas no superan a la configuración que ha obtenido el menor coste:

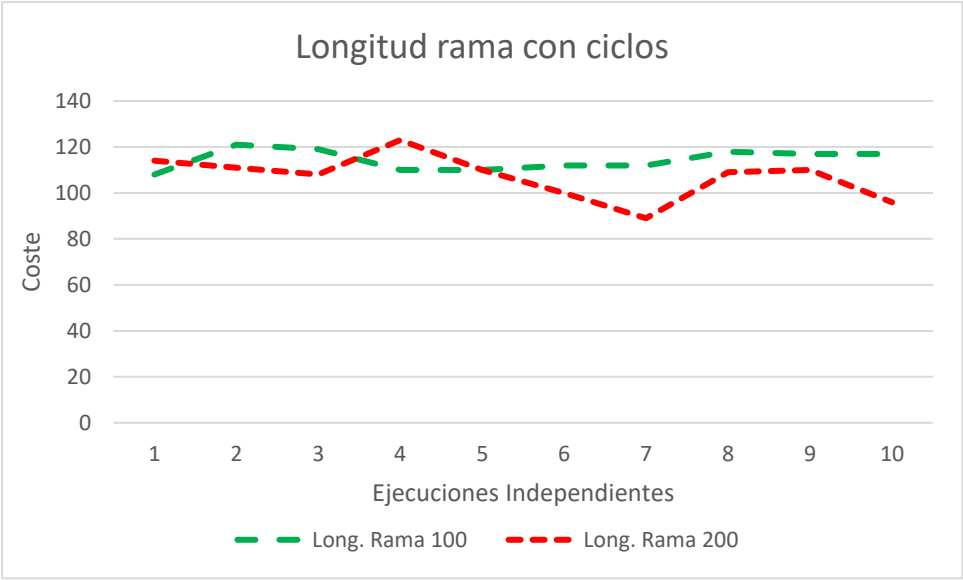

Figura 6.77 – Comparativa Long. Rama 100 y Long. Rama 200

Como se puede ver, la configuración con *"Long. Rama 100"* es muy estable, pero en 10 ejecuciones no consigue bajar de 108, mientras que la configuración con *"Long. Rama 200"*, además de tener una mejor media (106,18), obtiene mejores valores la mayoría de las veces.

El resto de variaciones del parámetro que controla la longitud del camino que contiene nodos repetidos han obtenido resultados peores a los descritos en los párrafos previos con mejores resultados y medias peores, por lo que no son reseñables (ver Figura 6.80).

El siguiente parámetro que se va a comprobar si su variación tiene un efecto positivo respecto a los resultados obtenidos, es el porcentaje de gotas trepadoras:

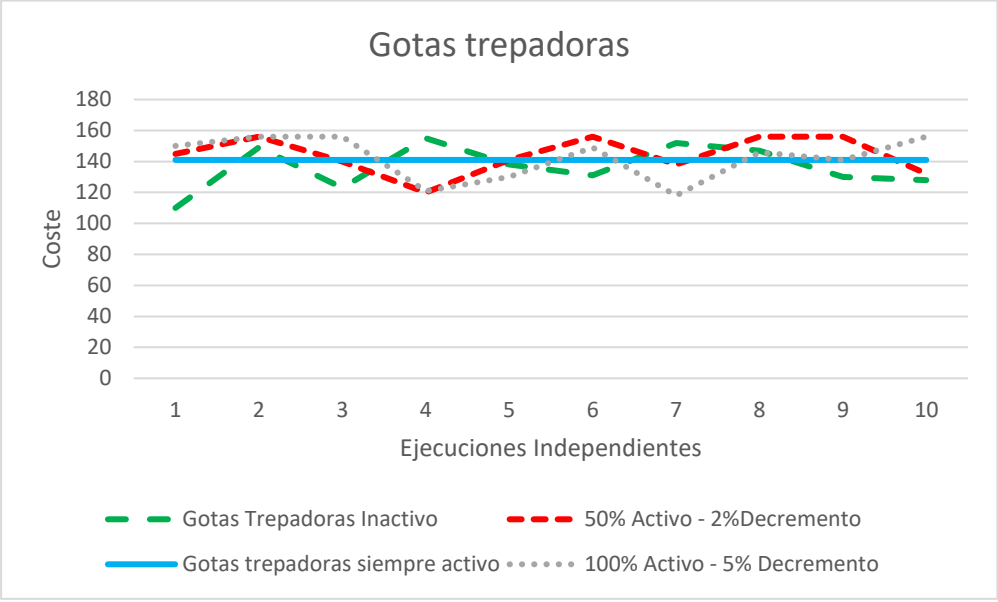

Figura 6.78 – Comparativa de las variaciones de porcentaje de gotas trepadoras

Tal como se puede comprobar, la mayoría de los costes obtenidos por las distintas modificaciones superan un coste de 120, y todas las medias son superiores a 134, una diferencia de más de un 20% de coste respecto a la mejor media obtenida (106,18). La mejor media de estas pruebas, conseguido por la configuración con porcentaje de gotas trepadoras inactivo en 10 ejecuciones es de 134,86 y una desviación típica de 13,71, el mejor coste que ha obtenido es de 110.

Por último, el parámetro que se va a estudiar si tiene puede disminuir el coste obtenido, es el porcentaje de altura mínima a partir de la que se realiza la sedimentación. Como se ha explicado anteriormente, este parámetro permite que se exploren soluciones que en un principio son más desfavorables, ampliando el rango de búsqueda:

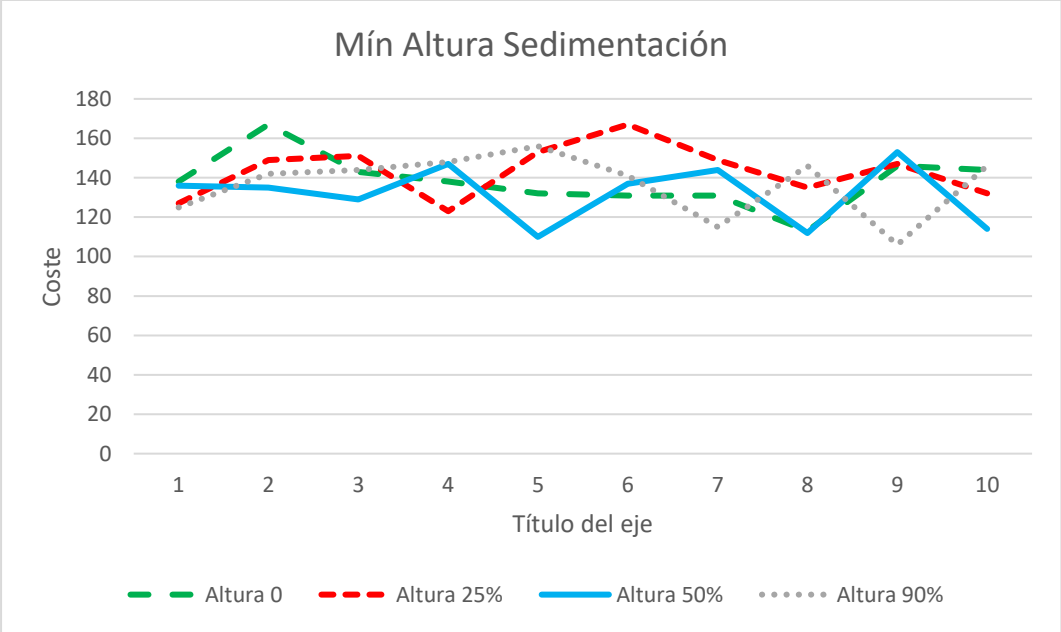

Figura 6.79 – Comparativa de las modificaciones de Mín. Altura Sedimentación

Al igual que en las pruebas anteriores (variación de porcentaje de gotas trepadoras), en general, los resultados están por encima de 120, aunque si que se han dado más casos que han llegado a bajar de este coste, pero la media de todas las pruebas es superior a 130. El mejor coste obtenido es de 106, por la configuración de altura 90%, con una media de 134,99 en 10 ejecuciones y una desviación de 15,24.

Tabla recopilatoria de resultados

A continuación, se presenta una tabla resumen de las pruebas realizadas para este problema de entrenamiento en el escenario con coste de carga 1 (los parámetros que no aparecen en la tabla se han mantenido invariables de los estándares definidos al inicio de esta sección):

Ejs.	Iteraciones	Gotas	% G. Trepadora	% Decremento G. Trepadora	Max. Long. Ciclo	% Min. Altura Sedimentación	Nodos	Densidad	C. carga	Mejor	Media	Desviación
10	100	100	100	2	0	100	100	30%	1	123	138,95	10,91
10	100	100	100	2	20	100	100	30%	1	95	122,71	12,26
10	100	100	100	2	50	100	100	30%	1	103	116,39	7,71
10	100	100	100	2	75	100	100	30%	1	97	116,71	14,14
10	100	100	100	2	100	100	100	30%	1	108	114,24	4,27
10	100	100	100	2	200	100	100	30%	1	89	106,18	9,15
10	100	100	0	0	200	100	100	30%	1	110	134,86	13,71
10	100	100	100	0	200	100	100	30%	1	120	143,00	11,65
10	100	100	50	2	200	100	100	30%	1	141	141,00	0,00
10	100	100	100	5	200	100	100	30%	1	118	140,88	13,70
10	100	100	100	2	200	0	100	30%	1	113	137,06	13,13
10	100	100	100	2	200	25	100	30%	1	123	142,13	12,92
10	100	100	100	2	200	50	100	30%	1	110	130,06	14,41
10	100	100	100	2	200	90	100	30%	1	106	135,00	15,24

Figura 6.80 – Tabla resumen de las pruebas realizadas para el problema de entrenamiento con baja densidad y coste de carga 1

Conclusión

La configuración óptima en este problema es de 100 Iteraciones – 100 Gotas, valor 200 para el valor que controla el tamaño de las ramas con ciclos, y el resto de parámetros estándar, con un mejor coste de 95, una media de 106,18 y una desviación típica de 9,15 en 10 ejecuciones. Al igual que en el escenario que penaliza las cargas de este problema, la configuración varía respecto a la obtenida en el problema de entrenamiento con alta densidad de conexiones entre nodos, por lo que se usará esta configuración para los problemas que tengan menor densidad de conexiones, y el otro para grafos cuyos nodos tengan acceso a un número considerable de nodos vecinos.

6.2. Comparación de algoritmos

Tras encontrar las configuraciones óptimas en el apartado anterior para cada uno de los algoritmos, a continuación, se compararán estos en diferentes problemas que se describirán más adelante. Para ver que los algoritmos obtienen mejor coste que un método que genera soluciones aleatoriamente, también se unirá un generador pseudo-aleatorio a la competición:

- El funcionamiento del método Random es el mismo que el de la generación de individuos aleatorios para la población inicial del algoritmo genético, su funcionamiento consiste en generar una rama para cada nodo crítico, y (en el caso de poder y que el coste de carga sea rentable) se añadirá el final de la rama generada con uno de los nodos de otra rama, sino se añadirá la rama al árbol solución por separado.

Problema 1

La máquina de estados será de 50 estados y cada nodo, estará conectado al 50% de los nodos vecinos. Los costes de realizar una transición de un estado a otro se generan aleatoriamente entre 10 y 100.

La entrada para este problema será de 10 nodos críticos elegidos al azar a visitar:

$$4,7,9,25,11,22,45,37,11,17$$

Para realizar la comparativa, se realizarán 20 ejecuciones completas e independientes de los algoritmos con sus mejores configuraciones. A partir de esas 20 ejecuciones, se sacarán conclusiones del comportamiento de las distintas metaheurísticas.

Problema 2

La máquina de estados tendrá tamaño 100, y las conexiones entre sus nodos serán del 50%. Las transiciones entre distintos estados tendrán un coste definido aleatoriamente entre 10 y 100.

La entrada que se usará es 20 nodos críticos elegidos de forma aleatoria a visitar:

$$5,6,8,15,22,23,26,35,39,45,48,57,59,61,62,74,78,83,84,91$$

Para realizar la comparativa, se ejecutarán 20 veces e independientemente los algoritmos con sus configuraciones óptimas. Con esas 20 ejecuciones, se procederá a observar el comportamiento de los algoritmos.

Problema 3

Para este problema, la máquina de estados se compondrá de 200 estados, y las interconexiones entre los estados serán del 50% para cada nodo. Los costes de ir de un estado a otro estado vecino se situarán entre 10 y 100.

Se va a pasar por 40 estados elegidos arbitrariamente:

9,7,5,2,11,15,16,14,23,22,29,27,35,36,38,34,42,44,48,45,51

59,54,52,62,67,63,69,70,72,73,75,85,82,86,89,90,91,95,99

Para la comparativa entre las metaheurísticas, se realizarán 20 ejecuciones independientes con las mejores configuraciones obtenidas. Con los datos obtenidos en las 20 ejecuciones independientes, se sacarán conclusiones respecto a las actuaciones de los algoritmos en este problema.

Problema 4

El cuarto problema que tratarán de solventar los algoritmos tiene una máquina de 50 estados, con una densidad muy baja (10% de conexiones entre sus nodos). Ir de un estado a otro tendrá un coste entre 10 y 100.

Se visitarán 10 nodos críticos elegidos al azar:

1,8,13,22,31,40,49,7,15,10.

Problema 5

Este problema tendrá una máquina de 100 estados, con interconexiones entre sus nodos del 10%. La transición de un estado a otro tendrá un coste asociado, generado aleatoriamente entre 10 y 100.

Se van a recorrer 20 nodos críticos de la máquina de estados propuesta, todos generados de forma arbitraria:

2,56,8,34,12,86,5,54,34,94,85,78,98,32,11,33,90,58,87,76.

Problema 6

El último problema en el que se va a realizar una comparativa de los tres métodos, tiene una máquina de estados con tamaño 200 y sus nodos están relacionados con el 10% de nodos vecinos. El movimiento de un estado a otro tiene asignado un coste (generado aleatoriamente) entre 10 y 100.

Los algoritmos pasarán por 40 nodos críticos elegidos al azar:

2,26,56,8,62,34,12,120,86,5,175,54,100,94,85,78,98,32,90,58,87,76,164,

166,198,110,168,183,122,199,150,145,108,101,177,158,43,80,190,144.

Para los tres problemas propuestos se realizarán las pruebas con costes de carga: 1, 20, y 500. Las comparaciones con distintos costes, permitirán ver en qué circunstancias se comporta mejor cada algoritmo.

Los métodos realizarán 20 ejecuciones independientes con sus mejores configuraciones para cada valor de carga. Con esas 20 ejecuciones (para cada valor de carga), se extraerán conclusiones del comportamiento de las metaheurísticas.

Problema 1:

Las pruebas realizadas en el problema 1, como se ha explicado anteriormente, proponen la misma máquina para los algoritmos, cambiando el coste de cargar un estado previo.

Para un coste de carga 1 (es menor que el valor mínimo de las aristas, por lo que se favorecerán las cargas) los resultados han sido:

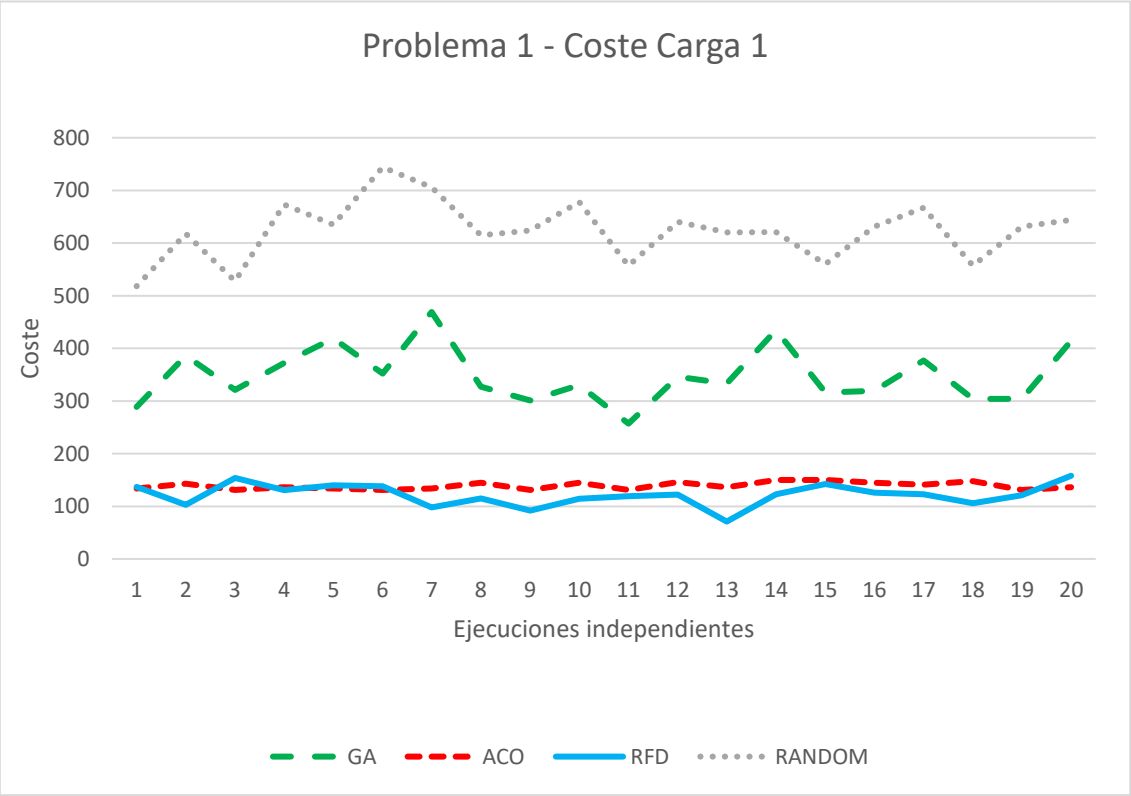

Figura 6.81 – Problema 1 - Coste de Carga 1

Como se puede observar, RFD es el que mejores resultados ha obtenido en las 20 ejecuciones, con un mejor coste de 71, una media de 121,65 y una desviación típica de 20,54. Por detrás de RFD, le sigue ACO con una mejor solución de 131, que siendo la más cercana, se aleja 60 de coste de la mejor solución obtenida por RFD y 10 de coste por encima de su media obtenida. GA es el algoritmo que peor se comporta en el problema, consiguiendo un mejor coste de 257, resultado que se queda bastante lejos de los otros dos algoritmos comentados. El método aleatorio -como era de esperar- no obtiene resultados si quiera cercanos a GA, con un mejor coste obtenido de 518.

El siguiente escenario del problema plantea un coste de carga 20, que al ser 10 el tamaño mínimo de la transición entre aristas hará que en ciertos casos los algoritmos decidan cargar, y en otros continuar con las transiciones planificadas:

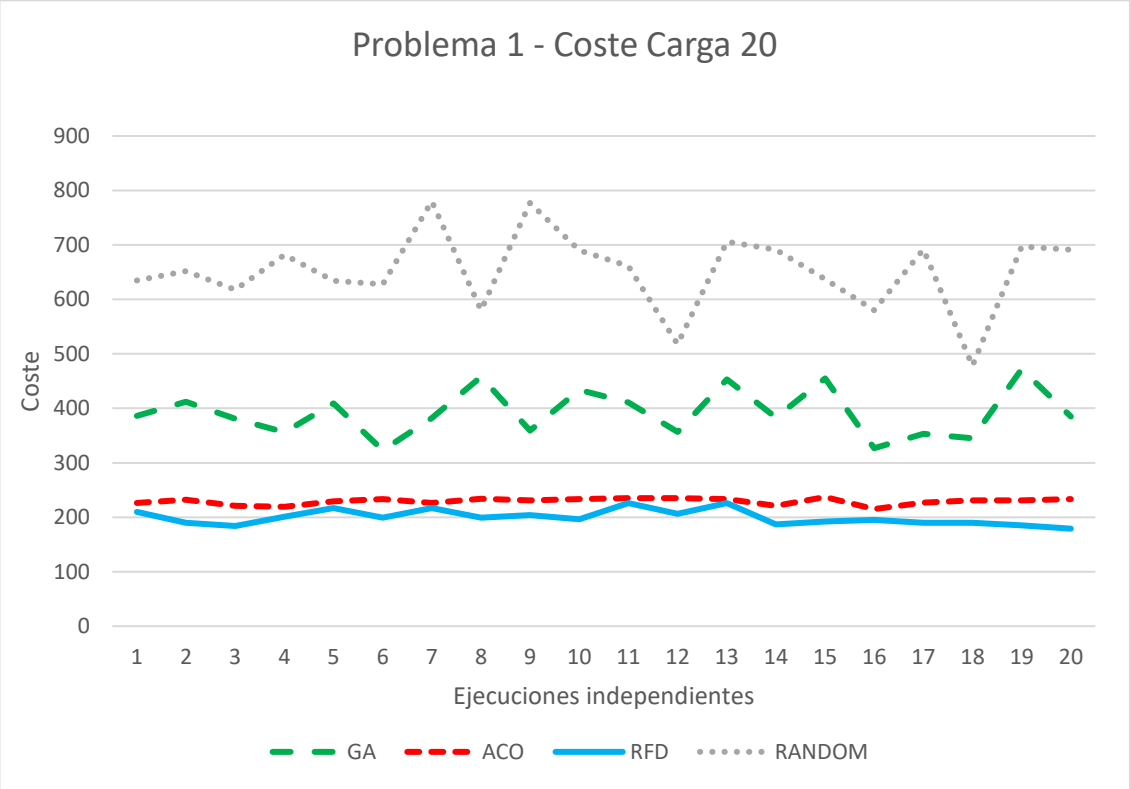

Figura 6.82 – Problema 1 - Coste de Carga 20

Al igual que en la competición anterior, RFD sale ganador con un mejor resultado de 179, con una media de 199,65 y una desviación de 13,37 entre los valores de la muestra. El mejor coste obtenido por ACO es de 215, que es peor que el mejor coste de RFD como su media. GA llega a unos valores parecidos a la prueba anterior, con un mínimo de 323, un coste un 80% peor que el coste de 179 de RFD. Random ha vuelto a obtener los peores resultados, de nuevo, con 518 como mejor resultado.

Para finalizar con las pruebas del problema 1, el último ámbito donde los algoritmos deben competir tiene coste de carga 500, penalizando en gran medida realizar cargas:

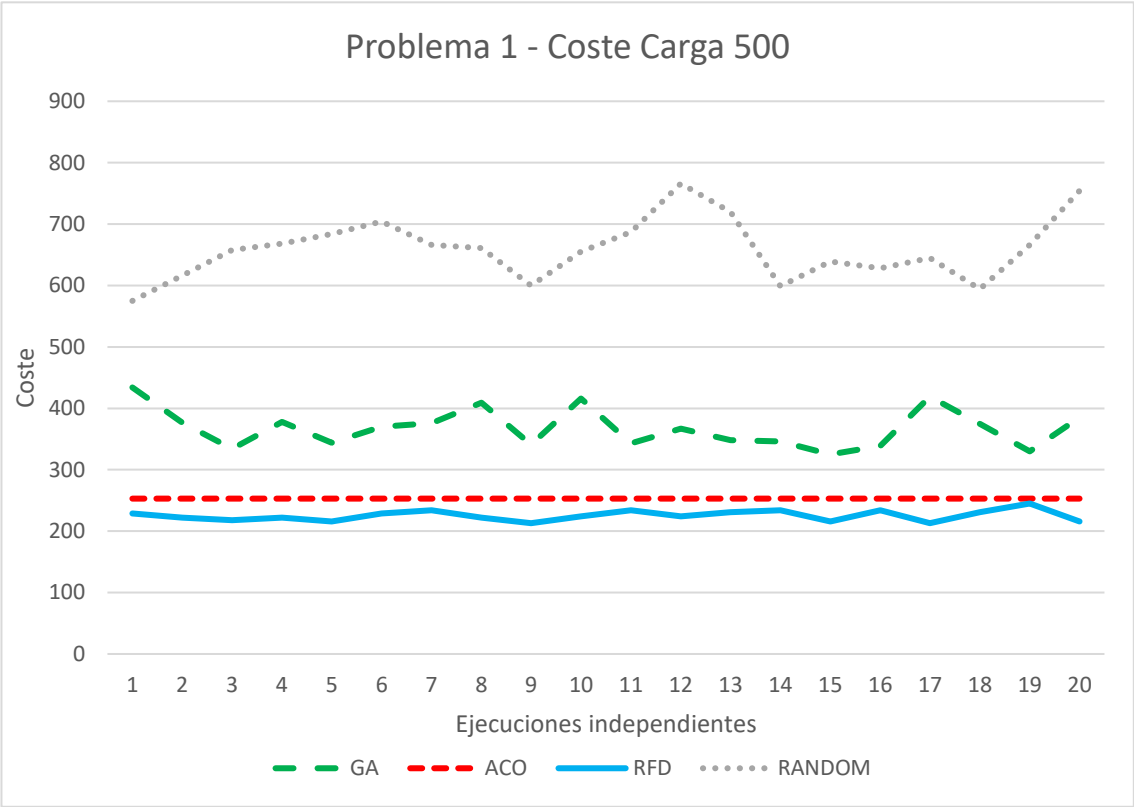

Figura 6.83 – Problema 1 - Coste de Carga 500

Para concluir con las pruebas del problema 1, RFD ha vuelvo a obtener los mejores resultados, con un mejor coste de 213, una media de 225,35 y una desviación típica de 8,46. En este caso ACO parece quedarse estancado en todas las ejecuciones que realiza, seguramente debido al número reducido de vecinos, por ello obtiene un mejor coste de 253 en todas las ejecuciones realizadas. GA realiza una actuación parecida a las anteriores, con una mejor solución de 323. Random continua en la misma tónica que en las anteriores pruebas, quedándose lejos de las metaheurísticas que son objeto de estudio.

Como primeras conclusiones a extraer, se puede ver que RFD se comporta bastante bien, siendo el mejor en las 3 pruebas realizadas. ACO le sigue, aunque sus resultados no lleguen a igualarle. GA parece no reaccionar ni aprovechar bien a las cargas, que aunque alguna vez saque algún resultado más cercano a ACO y RFD, normalmente les separa una distancia bastante notable, aunque como es obvio, obtiene mejores resultados que el método aleatorio.

Problema 2:

Este nuevo problema al que se tienen que enfrentar los algoritmos tiene un mayor tamaño que el problema probado anteriormente, con un tamaño de 100 estados.

En primer lugar, se procederá a realizar la comparativa con valor de coste de carga 1, que permite que se puedan realizar todas las cargas posibles:

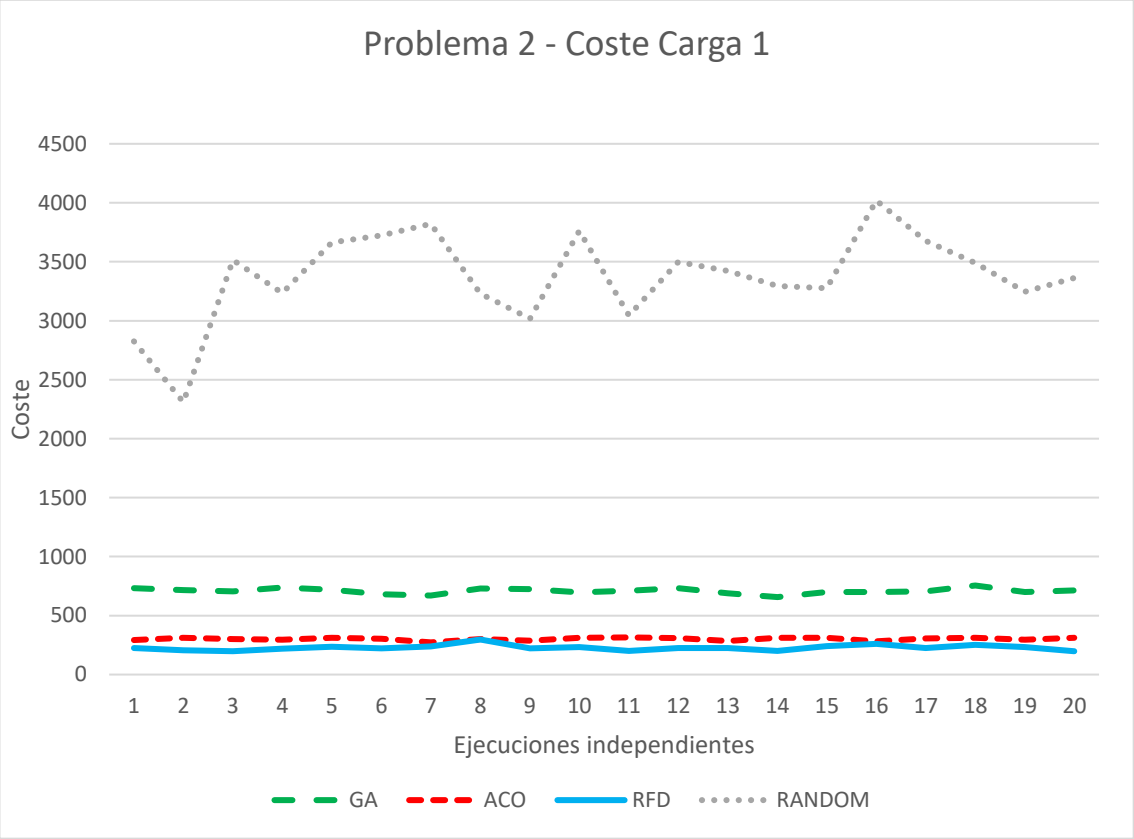

Figura 6.84 - Problema 2 - Coste de Carga 1

Como se puede observar la gráfica, RFD saca los mejores resultados en general, con un mejor coste de 198, una media de 228,1 y una desviación de 23,22. ACO -al igual que en el problema anterior- es el algoritmo que más de cerca le sigue con un mejor coste de 273, pero no consigue llegar a la calidad de soluciones de RFD. GA se queda muy por detrás de los anteriores algoritmos mencionados, con un coste mínimo de 657, que supera con creces lo obtenido por Random, cuyo mejor coste es 2310.

Para continuar con la evaluación de los algoritmos en el problema 2, la siguiente prueba que realizarán las distintas metaheurísticas tiene como coste de carga 20:

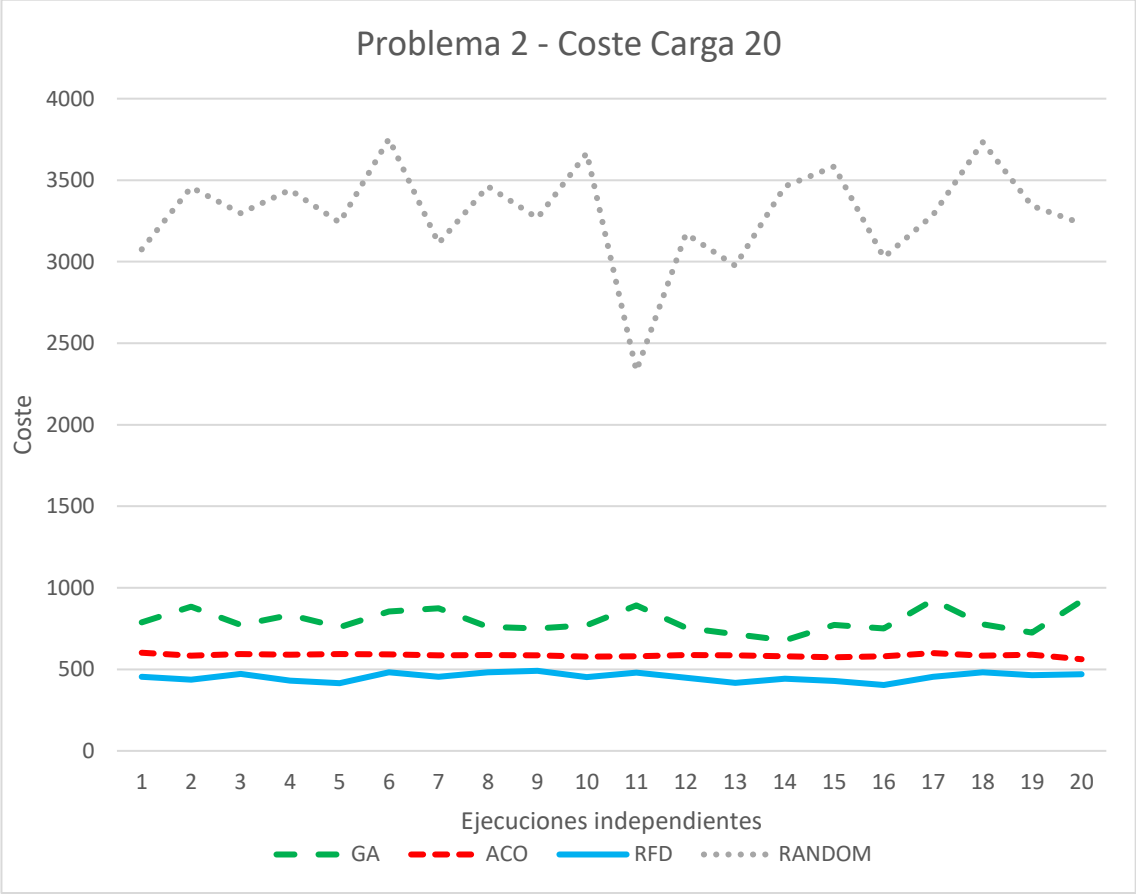

Figura 6.85 - Problema 2 - Coste de Carga 20

En esta prueba RFD -como es costumbre- obtiene el mejor coste de los 3 algoritmos de estudio, con un mejor coste de 404, una media de 453,35 y una desviación de 24,5. El coste mínimo conseguido por ACO -562- en esta prueba dista bastante de la media de RFD como del mejor coste. GA saca un coste similar al de la prueba anterior, con 678 de mejor coste. Random consigue costes por encima de 2000, una distancia abismal con los costes obtenidos por las metaheurísticas estudiadas.

La última prueba que realizar con este problema es la que asigna a su máquina de estados un coste de 500 para realizar cargas de un estado alcanzado previamente, imposibilitando esta tarea:

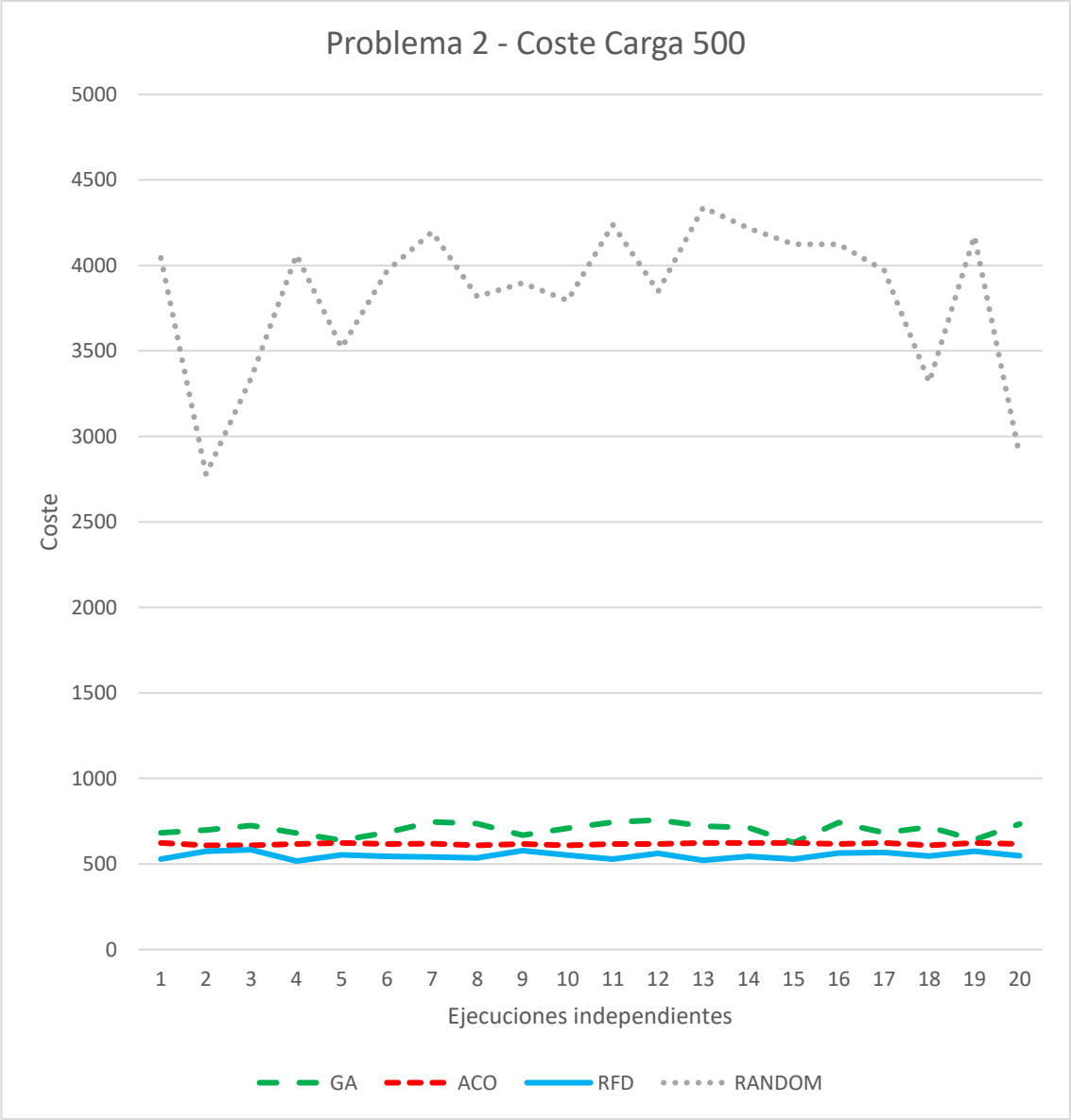

Figura 6.86 - Problema 2 - Coste de Carga 500

Al igual que en todas las pruebas ejecutadas, RFD ha obtenido el mejor coste de todos con un mínimo de 517 con una media de 549,75 y una desviación de 19,25. Como en la prueba anterior, ACO tiene una gran diferencia con el mejor coste obtenido por RFD, pues el mínimo coste encontrado por ACO es de 609, con casi 100 de coste de diferencia. GA se queda con un menor coste de 624, bastante lejos de los costes de RFD y ACO.

En las pruebas realizadas para este problema 2, comienzan a aparecer diferencias notorias como se puede comprobar en las figuras 6.47 y 6.48, donde RFD sigue sacando el mejor coste, y aunque ACO le siga, la diferencia es cercana a 100 de coste.

Tras las comparaciones de este apartado, se reafirma la teoría de que GA no reacciona bien a las cargas.

Problema 3:

El siguiente problema que las metaheurísticas van a tratar de resolver es de 200 estados, y se tratarán de cubrir un total de 40 nodos críticos.

En primer lugar, se va a realizar la comparación para coste de carga 1, que permite que se cargue siempre que se pueda:

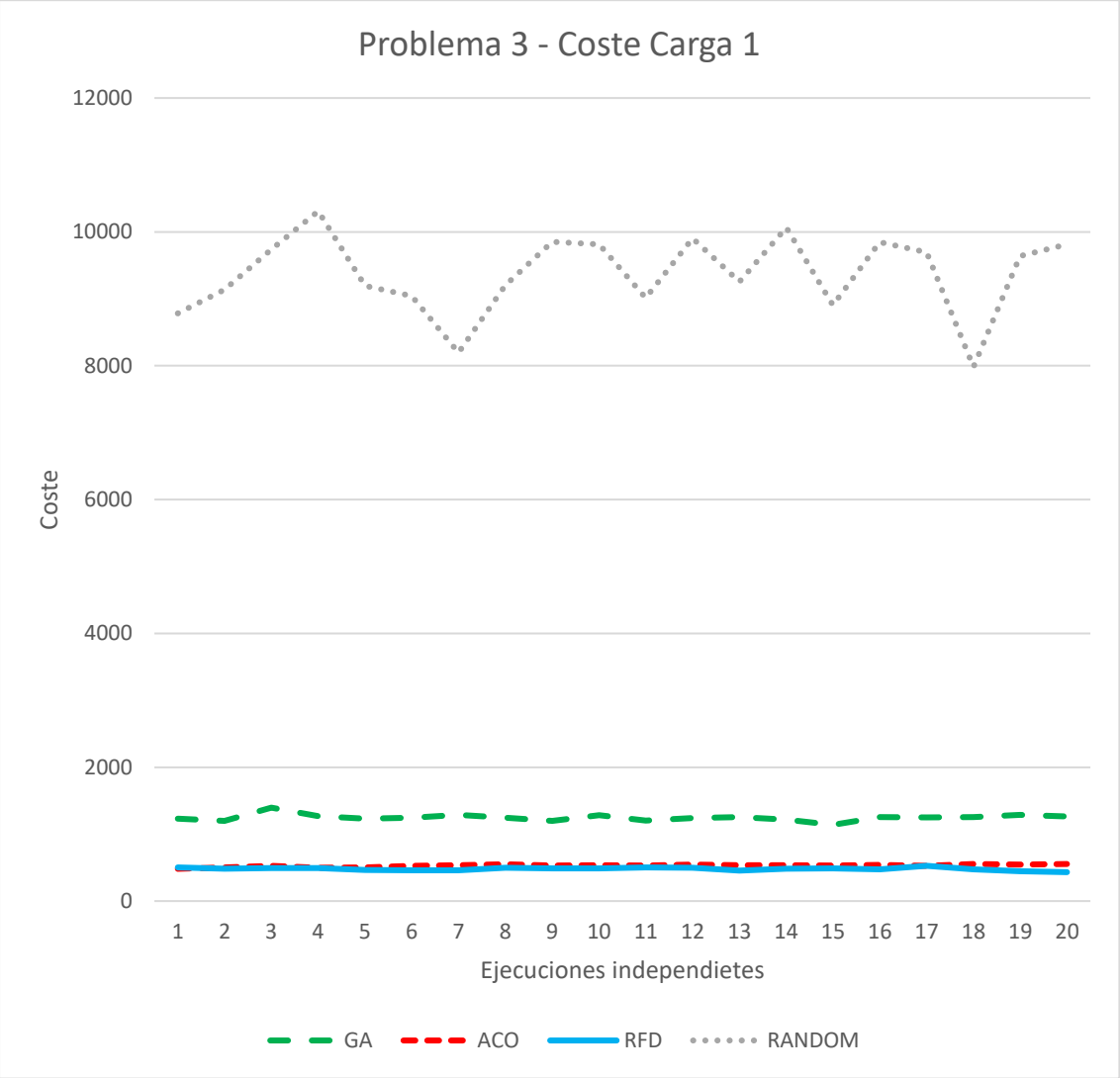

Figura 6.87 - Problema 3 - Coste de Carga 1

Debido a la diferencia que tienen los algoritmos con Random, no se puede apreciar las diferencias reales entre los métodos estudiados en el trabajo, por lo que se va a proceder a mostrar la misma sin el método Random para poder ver claramente el comportamiento de RFD, ACO y GA.

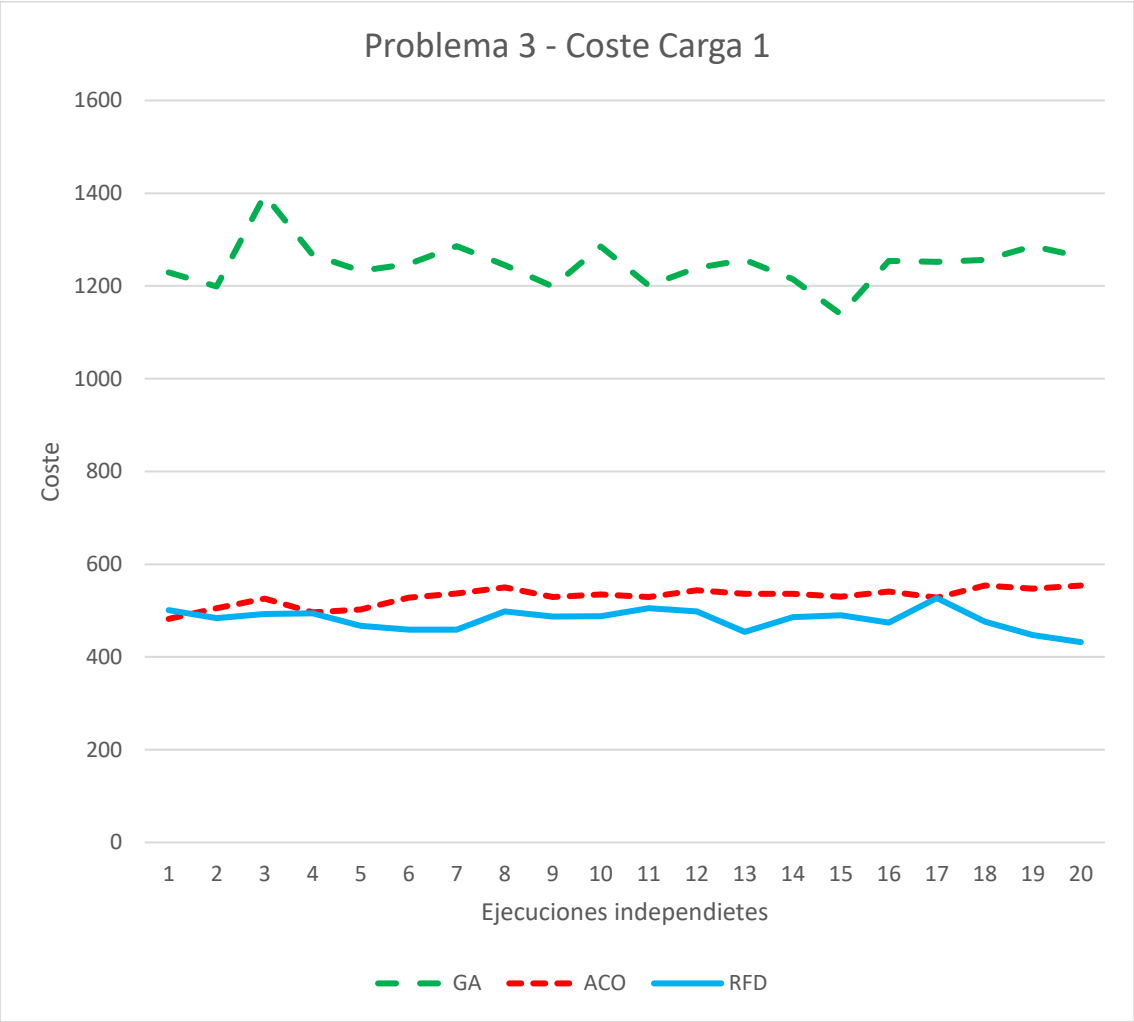

Figura 6.88 - Problema 3 - Coste de Carga 1 – Sin Random

GA obtiene en todas las ejecuciones realizadas costes superiores a 1000, resultados que quedan muy lejos por los que arrojan RFD y ACO. RFD tiene el pico de menor coste en 432, con una media de 480,95 y una desviación típica de 22,03. Hay resultados de ejecuciones de ACO que llegan a la calidad de algunas soluciones encontradas por RFD, sin embargo, el pico lo tiene en 482, superior a la media de 480,95 de RFD.

La siguiente prueba -siguiendo la metodología de las anteriores- se ha realizado con el mismo problema y un coste de carga 20 (se ha omitido los resultados del generador aleatorio para poder comparar los resultados más fácilmente):

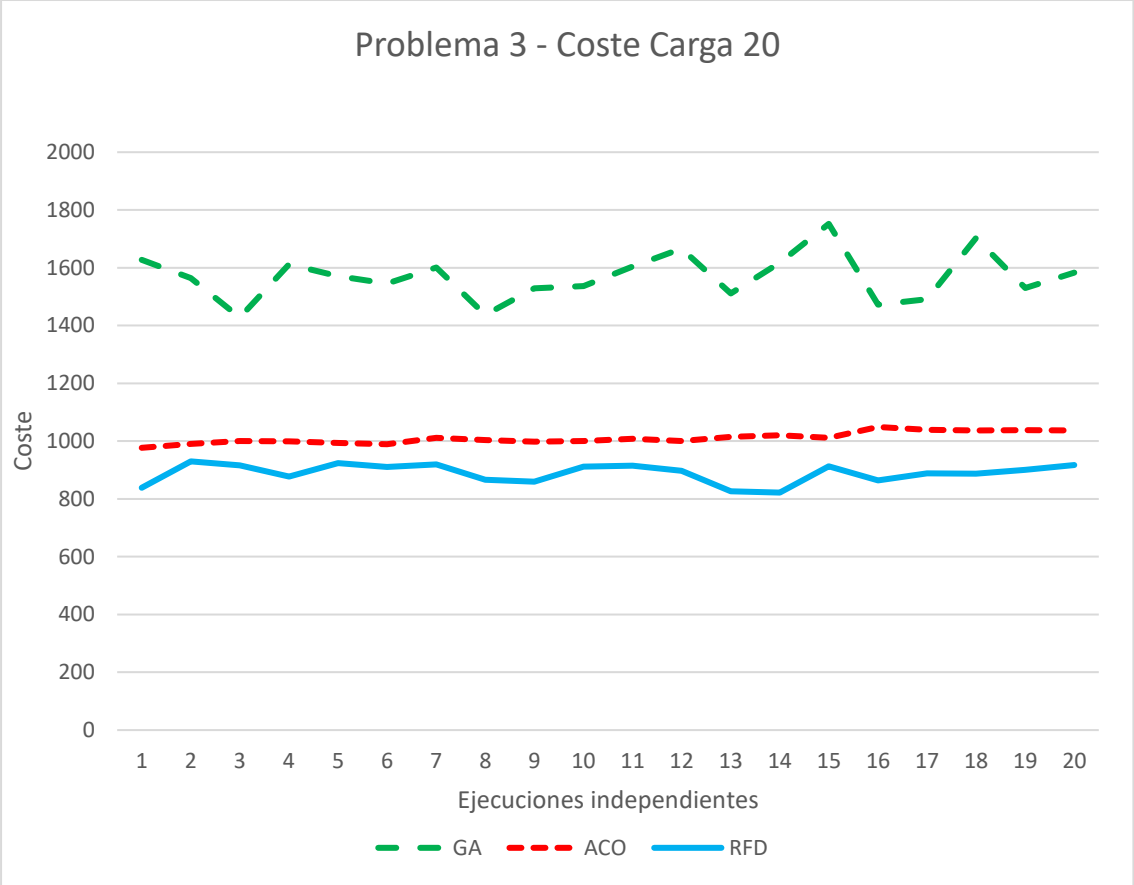

Figura 6.89 - Problema 3 - Coste de Carga 20

En esta prueba, RFD no se ve alcanzado por ninguno de los otros dos métodos, con un mejor coste de 822, media de 889,35 y desviación típica de 32,26, ACO -en comparación- saca un mínimo de 977, con una diferencia de 155 de coste respecto al mínimo de RFD. GA obtiene peores resultados que en la anterior prueba, pero sigue quedando muy lejos de los otros dos algoritmos.

Por último y para finalizar las comparativas, se va a comprobar el rendimiento de los algoritmos en el mismo problema con un coste 500 -o lo que es lo mismo- un problema en el que nunca llegarán a plantear una carga:

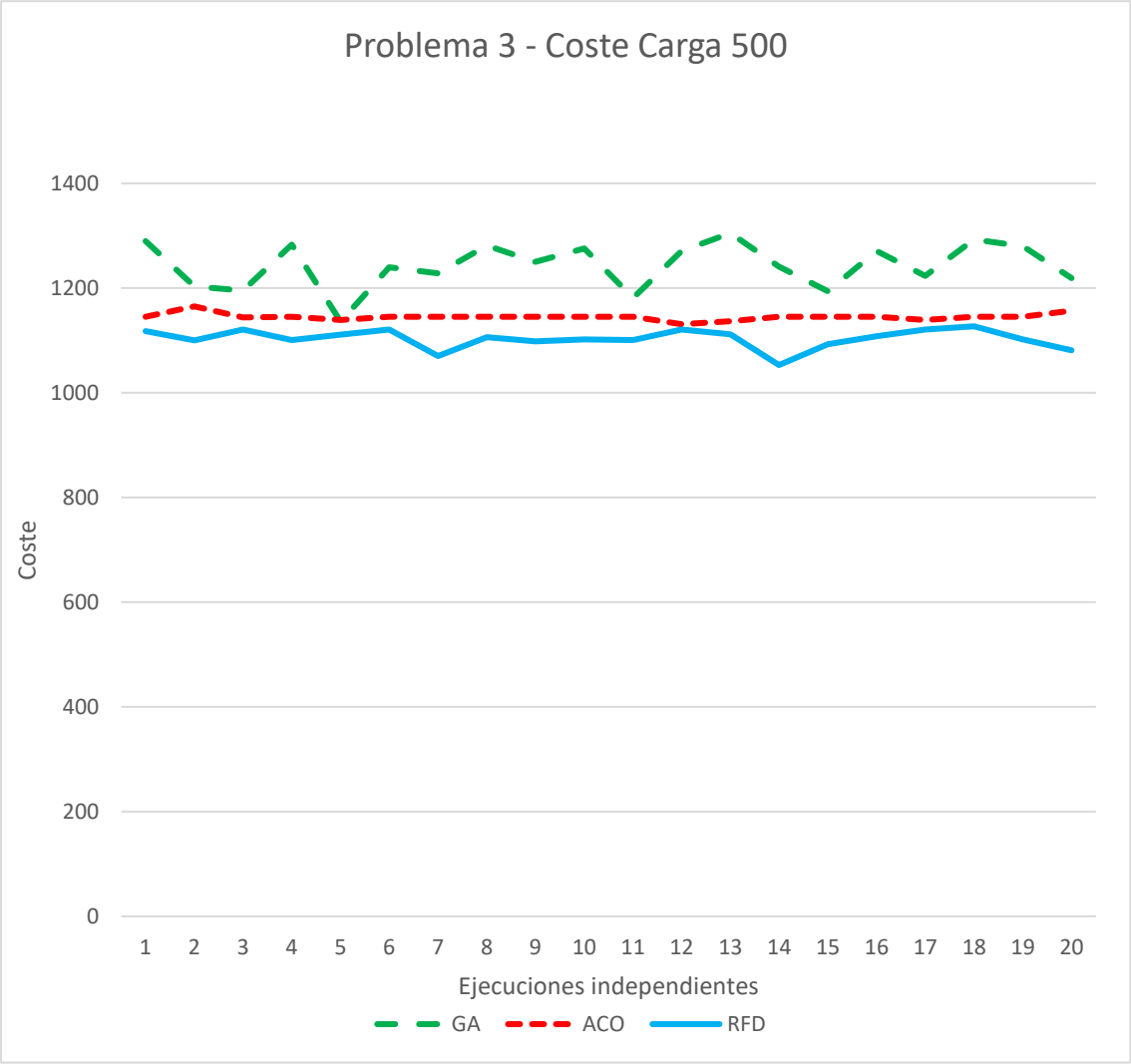

Figura 6.90 – Problema 3 - Coste de Carga 500

En esta prueba los 3 algoritmos llegan a resultados superiores a 1000, los mejores resultados siguen siendo los de RFD, con un mejor resultado de 1053, una media de 1103,35 y desviación de 18,01. ACO se acerca más a RFD en esta prueba que en la previa, pero les sigue separando una distancia notable, puesto que su mejor coste es de 1131. GA en esta comparativa es en el que menos perjudicado sale, ya que saca resultados relativamente "cercanos" en comparación con las pruebas anteriores.

Al igual que en el resto de comparativas, RFD es el que mejor se ha comportado siendo el método que mejores resultados saca en todas las pruebas, y además siendo la mayoría de ellos superiores a los de sus competidores. La distancia entre RFD y ACO es parecida a la del problema anterior o incluso un poco más notable con pruebas cuya distancia es superior a 100 de coste. GA termina demostrando que no reacciona bien a las cargas, pero en un ámbito donde ninguno de los otros algoritmos carga, sí que se

acerca más a la calidad de resultados de estos, aunque quedando la mayoría de las veces por detrás. El método de aleatorio dejó de mostrarse debido a la descomunal diferencia con el resto de algoritmos, ya que hacía más difícil la visualización de las comparativas de RFD, GA y ACO entre sí.

Problema 4:

En la siguientes comparativas, los algoritmos van a tratar de resolver un problema con pocas conexiones entre sus nodos y así ver como se comportan.

En este problema, el tamaño del grafo es de 50 estados, y los nodos a visitar son 10.

En primer lugar, se ha realizado las comparativas con coste de carga 1, que favorece que se realicen cargas siempre que se puedan (el coste de la arista es menor que el coste mínimo):

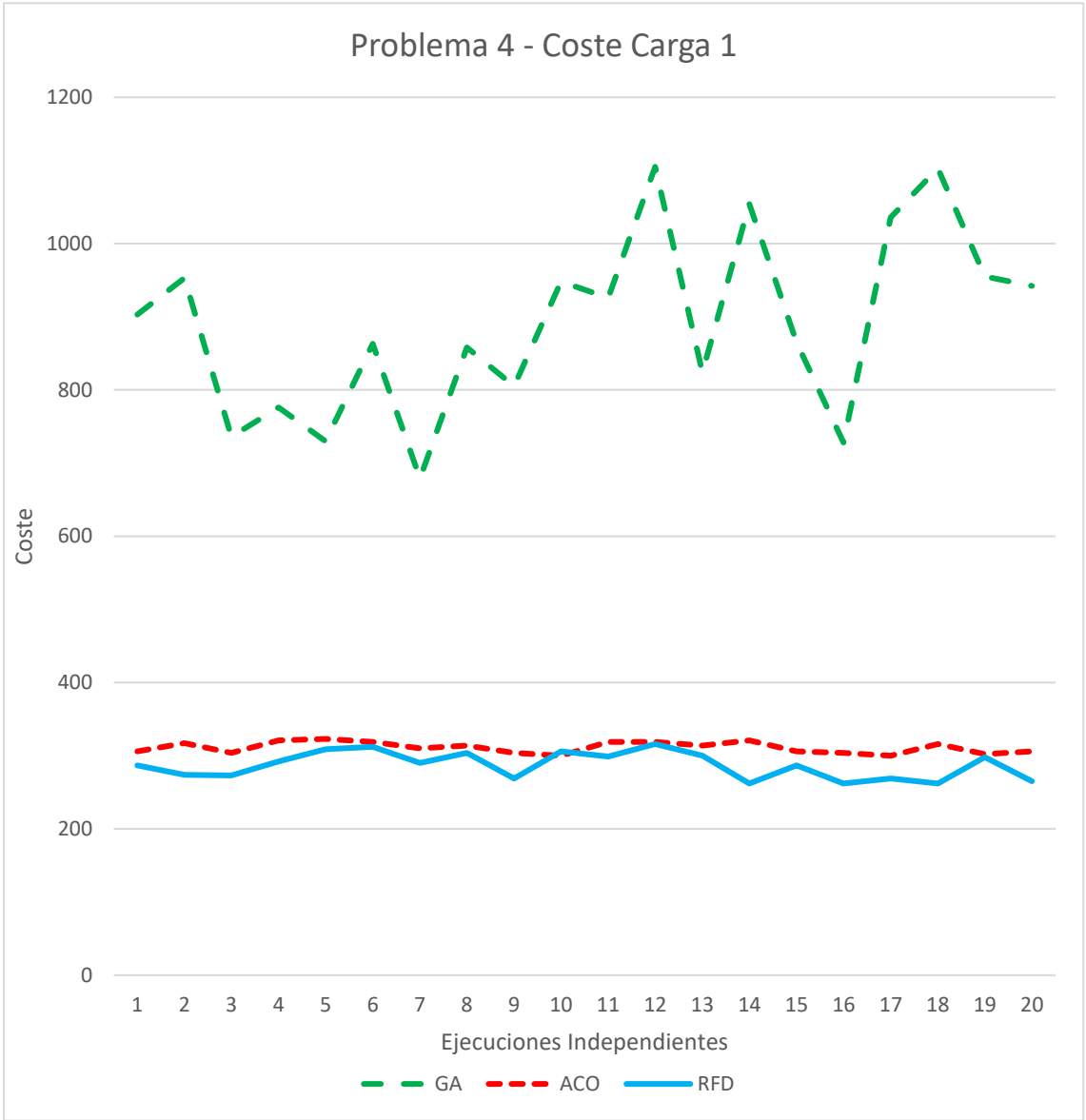

Figura 6.91 – Problema 4 - Coste de Carga 1

En la gráfica no se ha añadido Random ya que dificultaría la visualización de esta debido a los altos costes que obtiene, con un mejor coste de 1675, una media de 2063,98 en 10 ejecuciones y una desviación típica de 151,27. El algoritmo que mejor se comporta en esta prueba es RFD, con un mejor coste de 262, media de 285,67 y desviación de 17,94. ACO se mantiene en una segunda posición, con un mejor coste de 300, una media de 311,06 y una desviación de 7,61, obteniendo un mejor resultado que está casi 15 puntos por encima de la media de RFD. GA se comporta mal con cargas, tal como se ha ido viendo en los anteriores problemas, con menor coste encontrado de 679, una media de 872,39 y desviación de 122,99, creando una notable diferencia entre los otros dos algoritmos, de más de 500 puntos con la media ACO y de casi 600 con la de RFD.

La siguiente prueba se realiza con coste de carga 20, que hará que en algunos momentos se cargue, y en otros no:

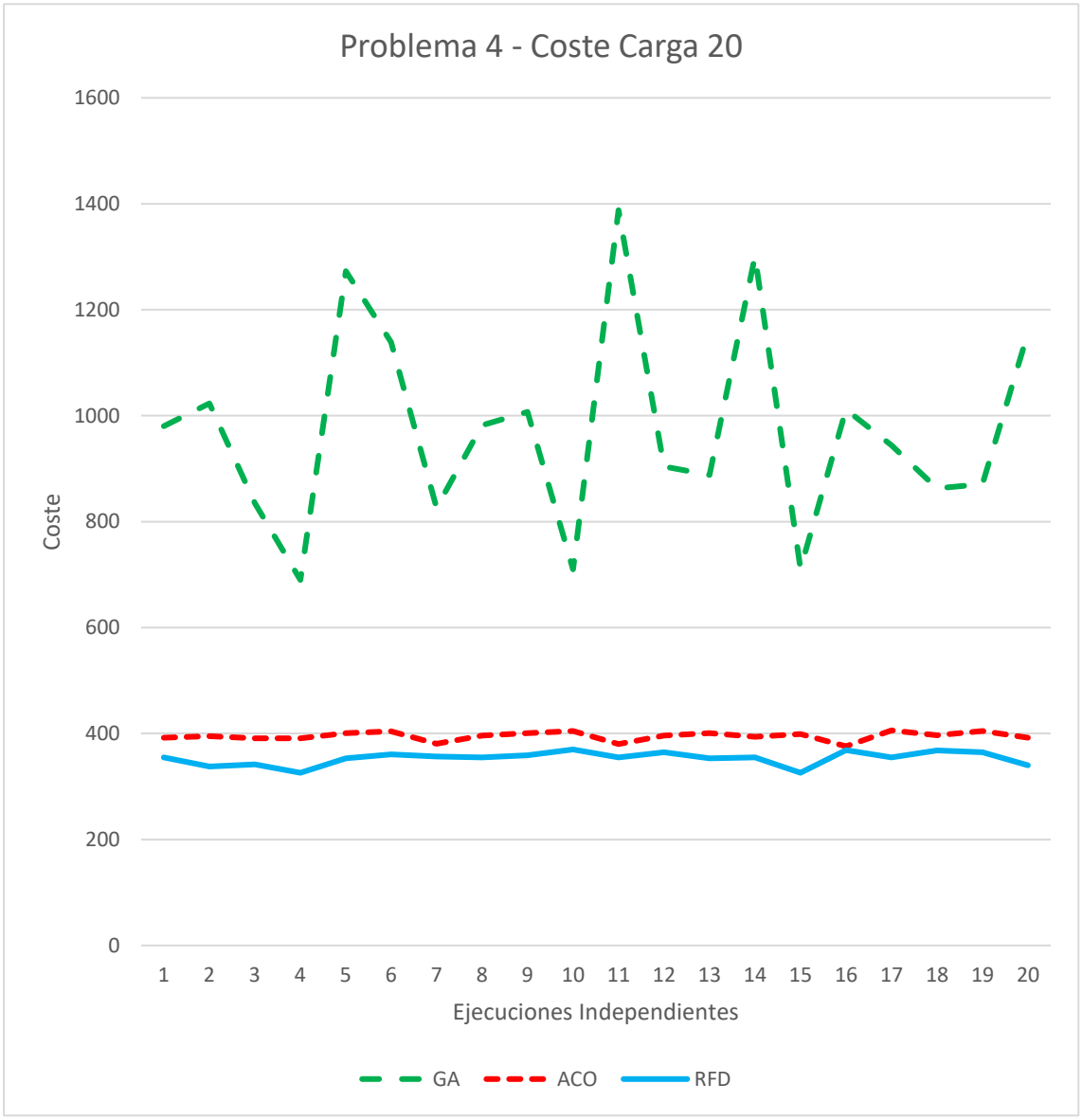

Figura 6.92 – Problema 4 - Coste de Carga 20

El menor coste obtenido es de 326, por RFD con una media de 352,89 y una desviación típica de 12,58 en 20 ejecuciones. ACO obtiene una mejor solución con coste de 376, media de 394,97 y desviación típica de 8,26. GA en esta prueba obtiene un menor coste de 690, una media de 939,58 (superior a la de la prueba anterior) y una desviación de 191,04. Random (no mostrado en la gráfica, por cuestiones visuales) obtiene un mejor coste de 1769, con una media de 2333,85 y una desviación de 216,68.

El último enfrentamiento de este problema se realiza con coste de carga 500, que penalizará las cargas en todo momento, llegando a impedir en la mayoría de casos que estas se produzcan:

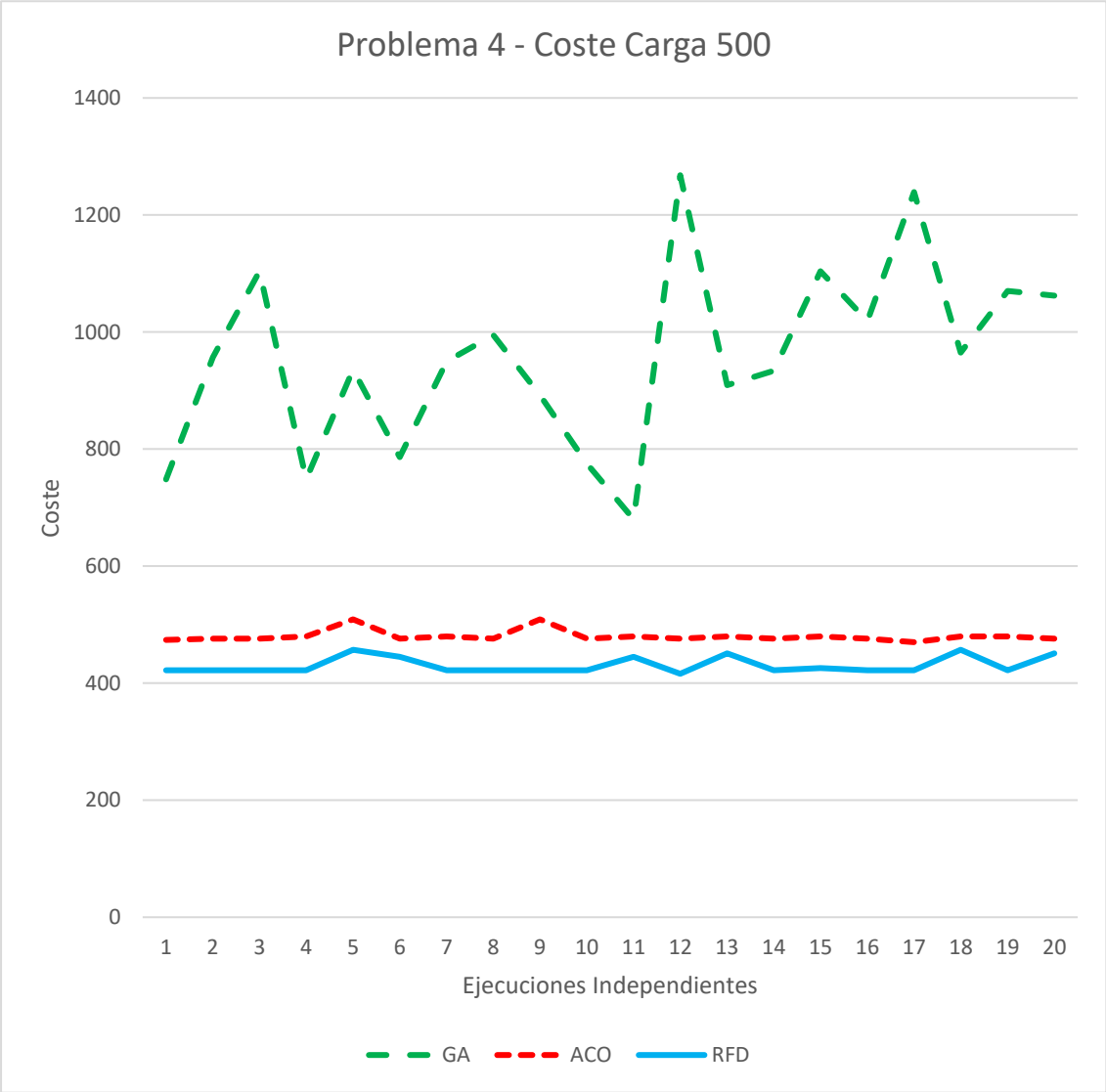

Figura 6.93 - Problema 4 - Coste de Carga 500

GA obtiene resultados similares a los de las anteriores pruebas, con un menor coste de 680, una media de 931,49 y una desviación típica de 155,34. ACO consigue un mejor coste de 470, media de 480,11 y desviación típica 9,9. RFD halla la mejor solución de la

prueba, con una mejor solución de coste 416, media de 430,17 y una desviación típica de 13,72. Random obtiene un mejor resultado de 2132, con media superior a 2756.

En este problema GA muestra ser bastante inestable con desviaciones muy altas en las tres pruebas realizadas. ACO y RFD se muestran relativamente cercanos, aunque como en el resto de pruebas, RFD es el que mejor se comporta, aunque ambos son notablemente estables respecto a los resultados que obtienen a lo largo de las distintas ejecuciones.

Problema 5:

El siguiente problema, objeto de comparativa entre las distintas metaheurísticas, es muy poco denso, con un tamaño de 100 y 20 nodos críticos a visitar.

La primera prueba de este problema -al igual que en los anteriores- se realizará con coste de carga 1, que favorece en gran medida la realización de cargas:

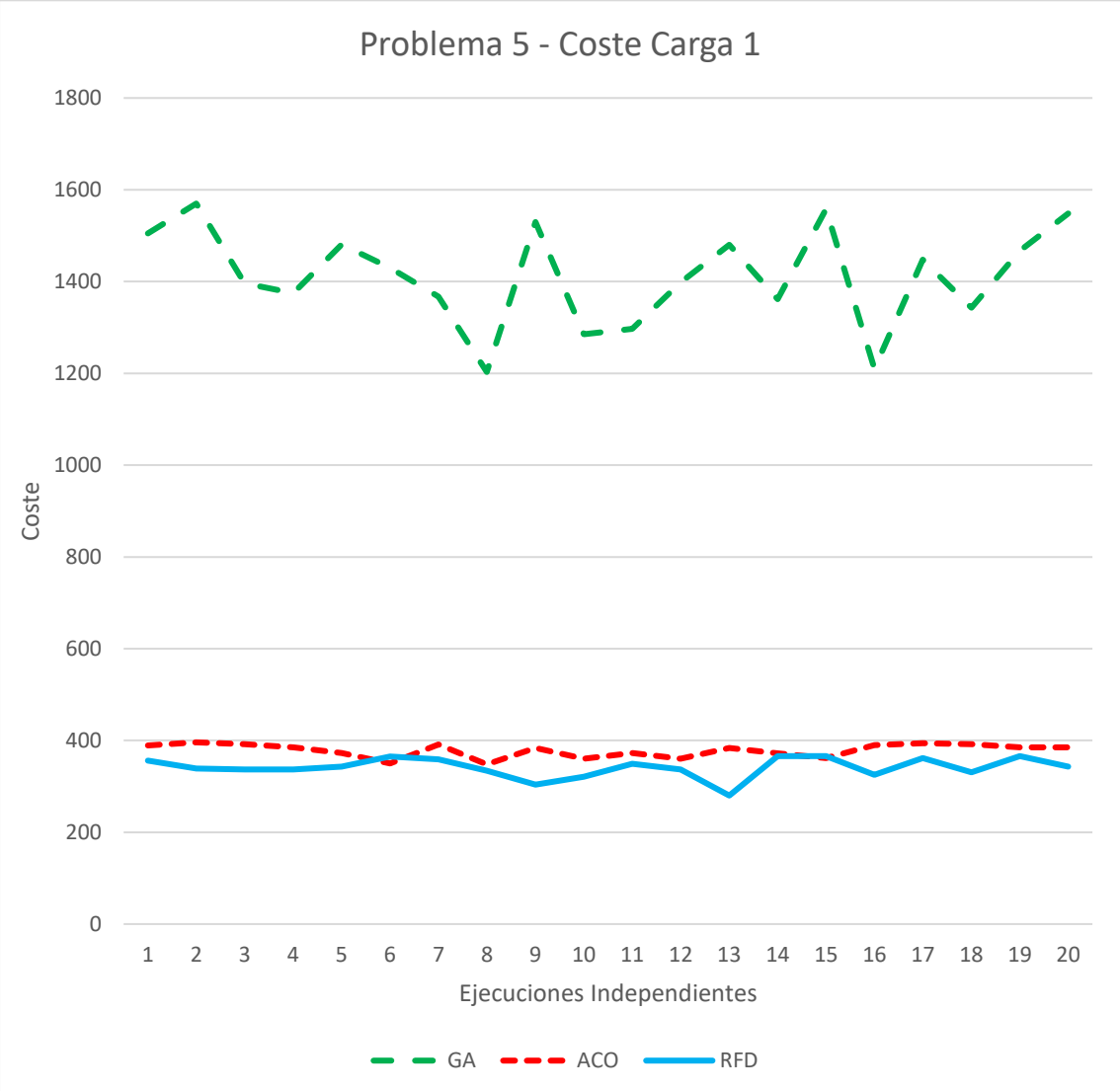

Figura 6.94 – Problema 5 - Coste de Carga 1

El mejor resultado de esta prueba es de 280, alcanzado por RFD, con una media de 339,50 y desviación típica de 21,75. El segundo mejor coste es de 348, obtenido por ACO, cuya media es de 377,66 y una desviación de 14,66. GA consigue unos resultados señaladamente peores, con un mejor coste de 1203, con media de 1404,25 y desviación de 106,38, datos con una diferencia de una cifra de casi tres dígitos respecto a las otras dos metaheurísticas de estudio. Random llega a un coste de 5154, coste 4,28 veces superior al mejor coste de GA, que ya era el que peores resultados mostraba.

La prueba que se presenta a continuación se ejecutará con coste de carga 20, que permite cargar en distintas situaciones, pero no asegura esta acción:

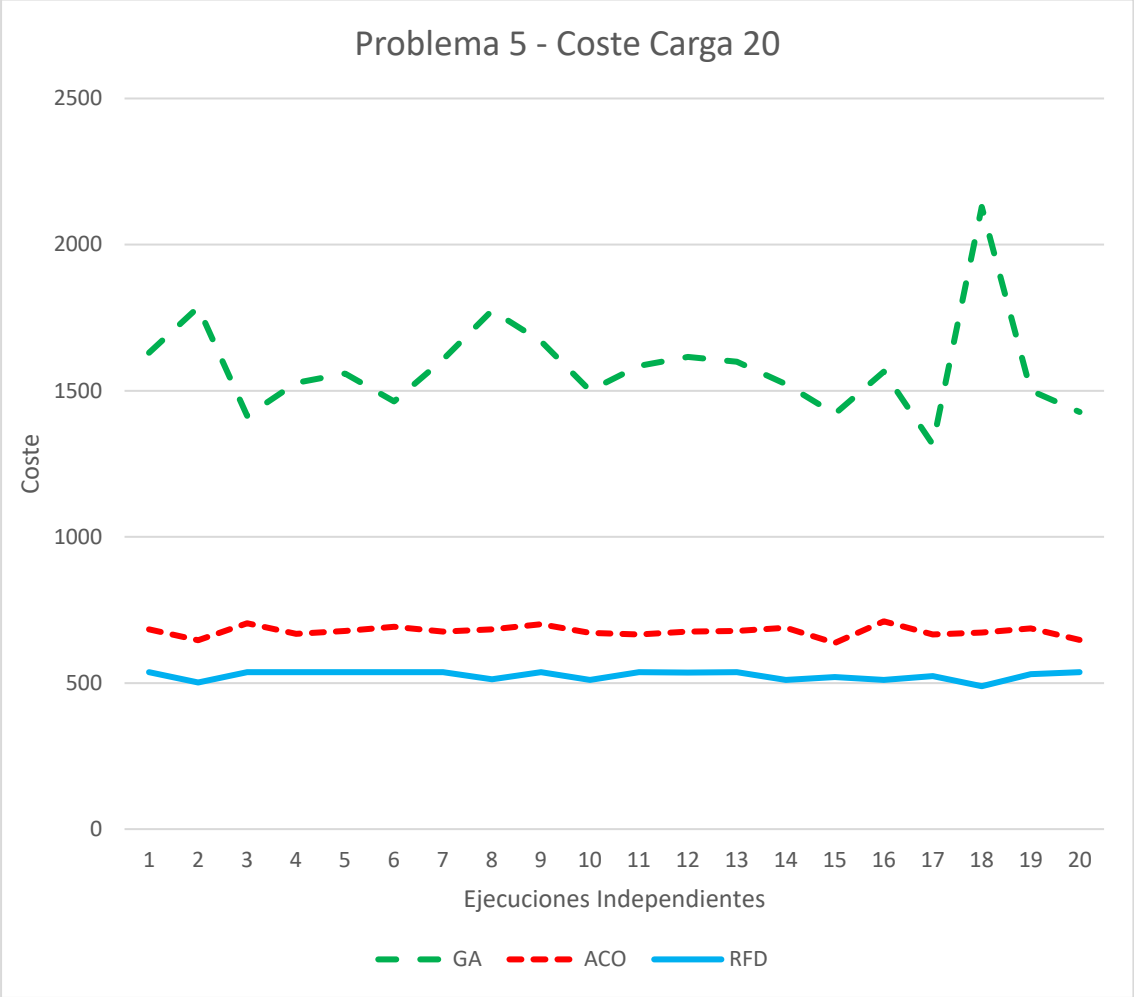

Figura 6.95 – Problema 5 - Coste de Carga 20

El mejor coste -con una marcada diferencia- es de 489, perteneciente a RFD, con una media de 525,44 y desviación de 14,41. ACO obtiene el siguiente mejor resultado de la comparativa es de 637, con media de 676,14 y una desviación de 18,51, datos notoriamente diferentes a los obtenidos por RFD, con diferencia de más de 100 puntos entre sus medias. GA queda totalmente fuera de la competición, con un menor coste de 1316, una media de 1564,56 y una desviación de 169,22. Random se comporta de

una forma parecida a la anterior prueba, con un mejor coste de 5102, una media de 6086,85 y desviación de 358,65.

La última prueba de este problema se llevará a cabo con un valor de carga de 500, que dificultará en gran medida la carga:

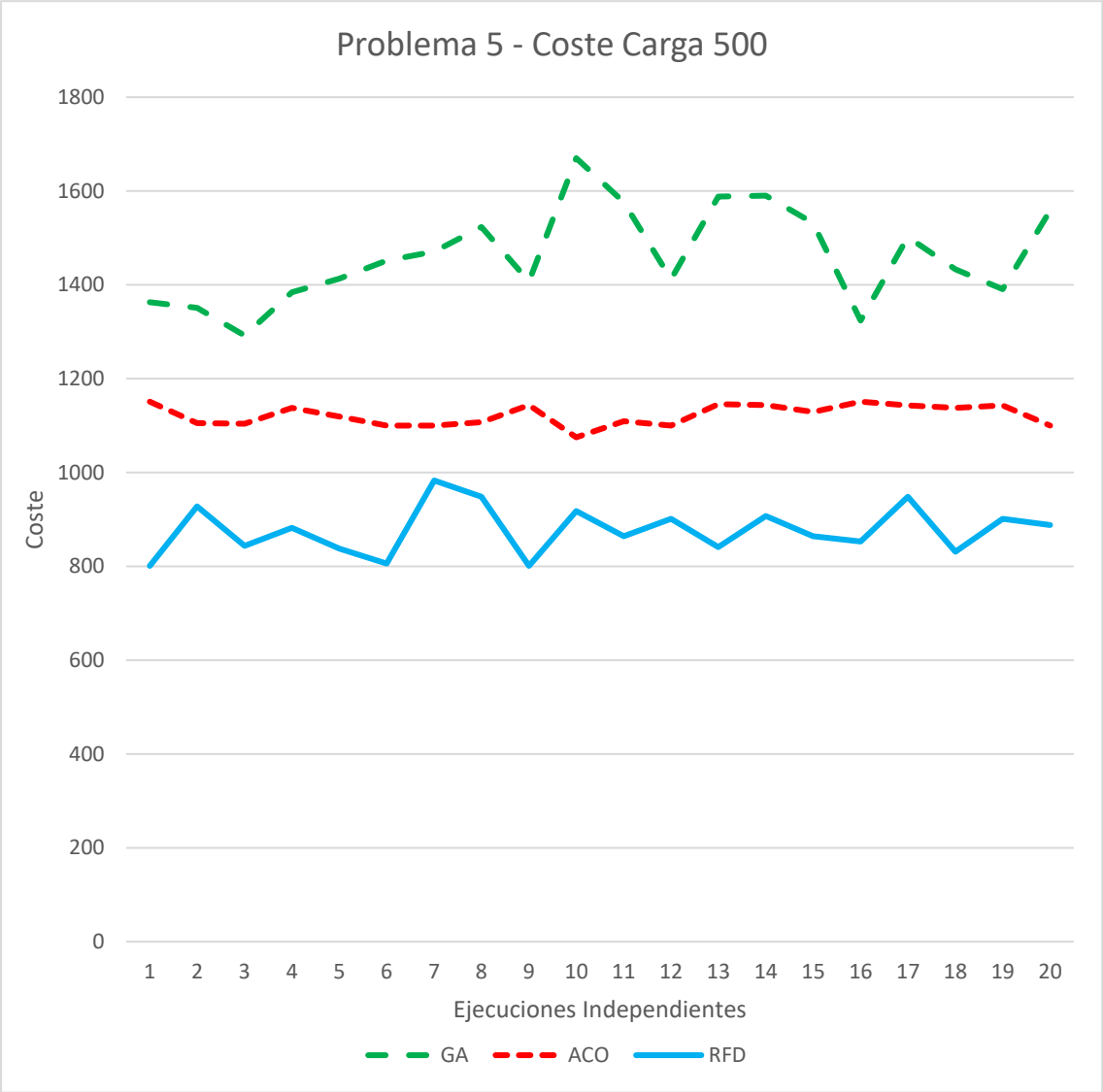

Figura 6.96 – Problema 5 - Coste de Carga 500

La gráfica muestra grandes diferencias entre los 3 métodos, colocando por delante -como es usual- a RFD con un mejor coste de 801, media de 874,45 y desviación de 50,65; seguido de ACO, con un mejor coste de más de 200 puntos por encima del anterior, 1075, una media de 1121,86 y una desviación típica de 22,16; por último GA consigue resultados parecidos a los de las anteriores ejecuciones, con un mejor coste de 1292, una media de 1454,91 y una desviación de 99,52 que es relativamente menor a la de las anteriores pruebas.

En este problema, GA ha mostrado mayores diferencias en comparación al resto de algoritmos, mostrándose bastante inestable (al igual que con el problema 4),

denotando que en los problemas con una densidad muy baja no es muy robusto. ACO muestra las mayores diferencias de todas las pruebas realizadas respecto a RFD, aunque si que es bastante estable en cuanto a los resultados obtenidos en las ejecuciones independientes. RFD obtiene los mejores resultados de una forma robusta en la mayoría de sus ejecuciones, aunque menos robusta -en comparación- que ACO, pero las diferencias de coste hacen que se sitúe en la cabecera con una clara diferencia.

Problema 6:

El último problema en el que se va a comparar los algoritmos estudiados, tiene una máquina de 200 estados, 40 nodos críticos a visitar y una densidad marcadamente baja.

El primer escenario en el que se va a comparar a los métodos, tiene coste de carga 1, que beneficia la carga en la mayoría de los casos:

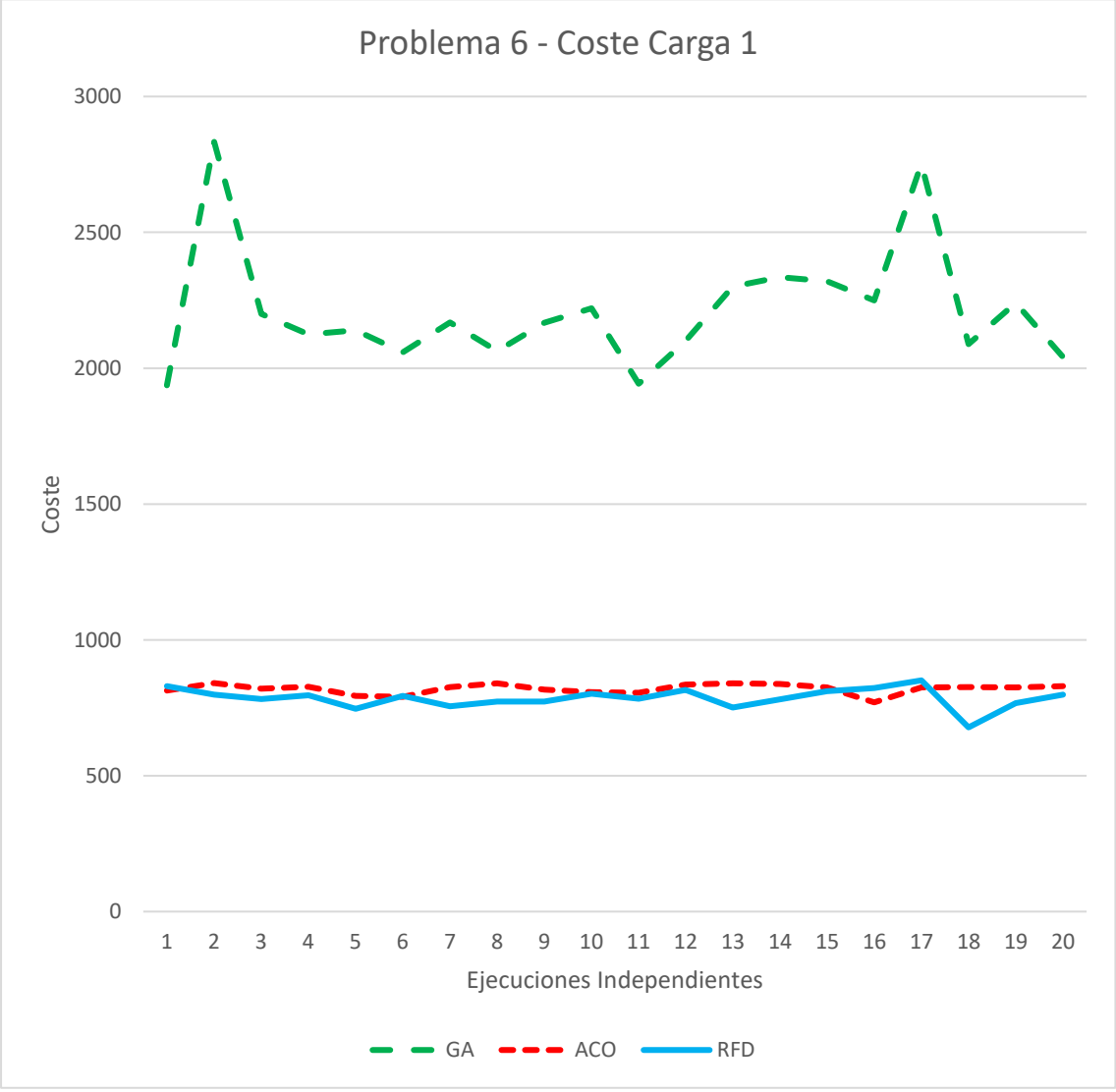

Figura 6.97 – Problema 6 - Coste de Carga 1

GA -como en la mayoría de pruebas- queda fuera de la comparativa, pero esta vez con diferencias mucho más notables que en el resto de pruebas, con un mejor coste obtenido de 1937, media de 2194,49 y desviación típica de 221,54. ACO obtiene claramente mejores resultados que GA, con un mejor coste de 770, una media de 819,44 y una desviación de 18,09. RFD llega al mejor resultado de los 3 algoritmos, con un coste de 678, una media de 783,76 y una desviación de 36. Random se sale de la escala de la comparativa -con una diferencia abismal- mostrando un mejor coste de 14004 y una media de 14832,42.

La prueba presentada a continuación, tiene asociado un coste de carga de 20, que favorece la carga, pero no la asegura en todos los casos:

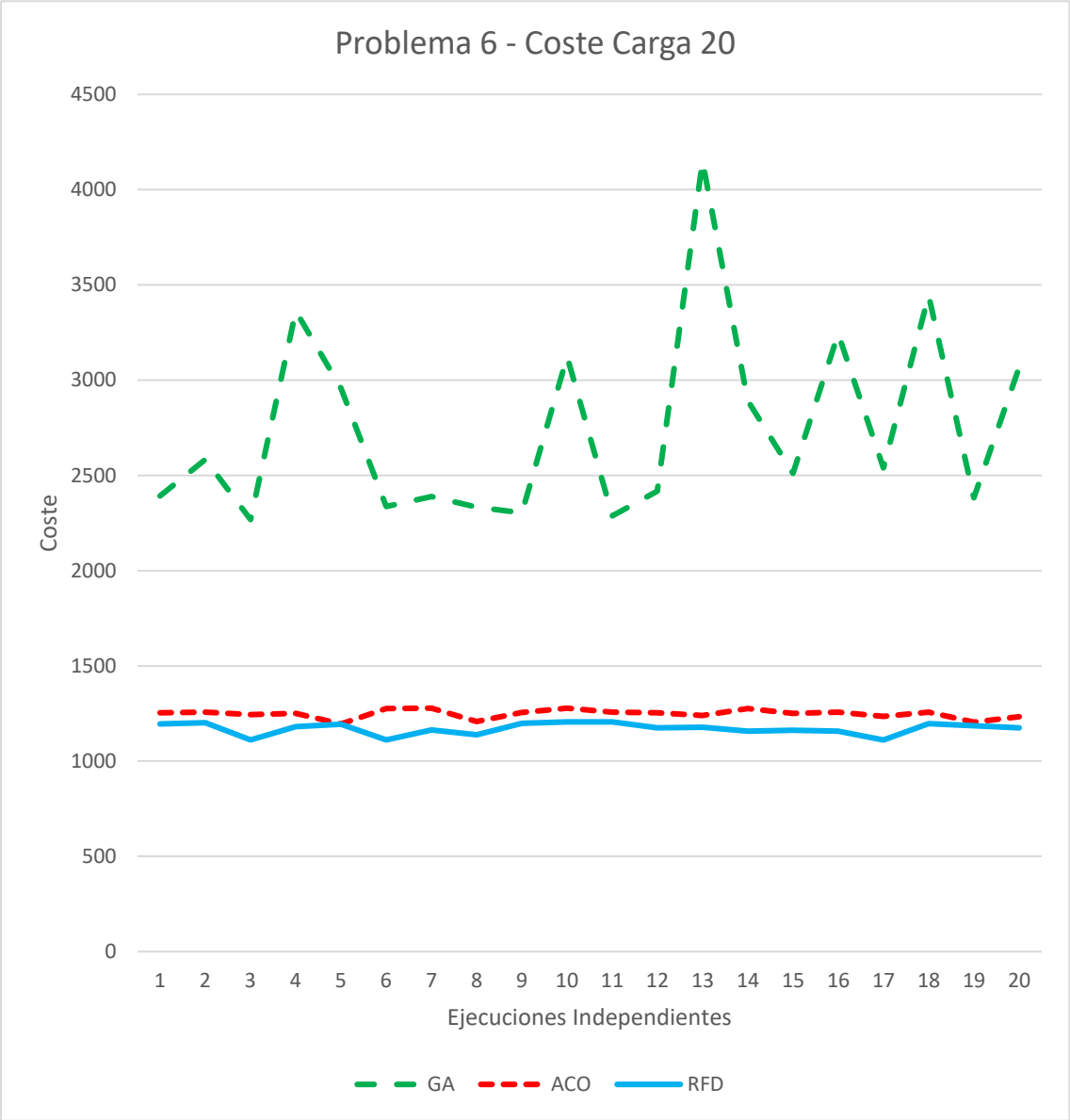

Figura 6.98 – Problema 6 - Coste de Carga 20

RFD obtiene los mejores resultados en la mayoría de las ejecuciones, con un mejor coste de 1111, una media de 1169,45 y una desviación de 30,30. ACO halla el segundo

mejor coste con 1195, media de 1247,52 y una desviación de 22,97. GA obtiene peores resultados que en la anterior prueba, con un mejor coste de 2268, media de 2671,91 y desviación de 499,50. Random saca una media de 15306,63, media que es 5,73 veces mayor a la de GA (peor de las medias).

La última prueba por realizar tiene coste de carga 500, que impedirá realizar las cargas en la mayoría de los casos:

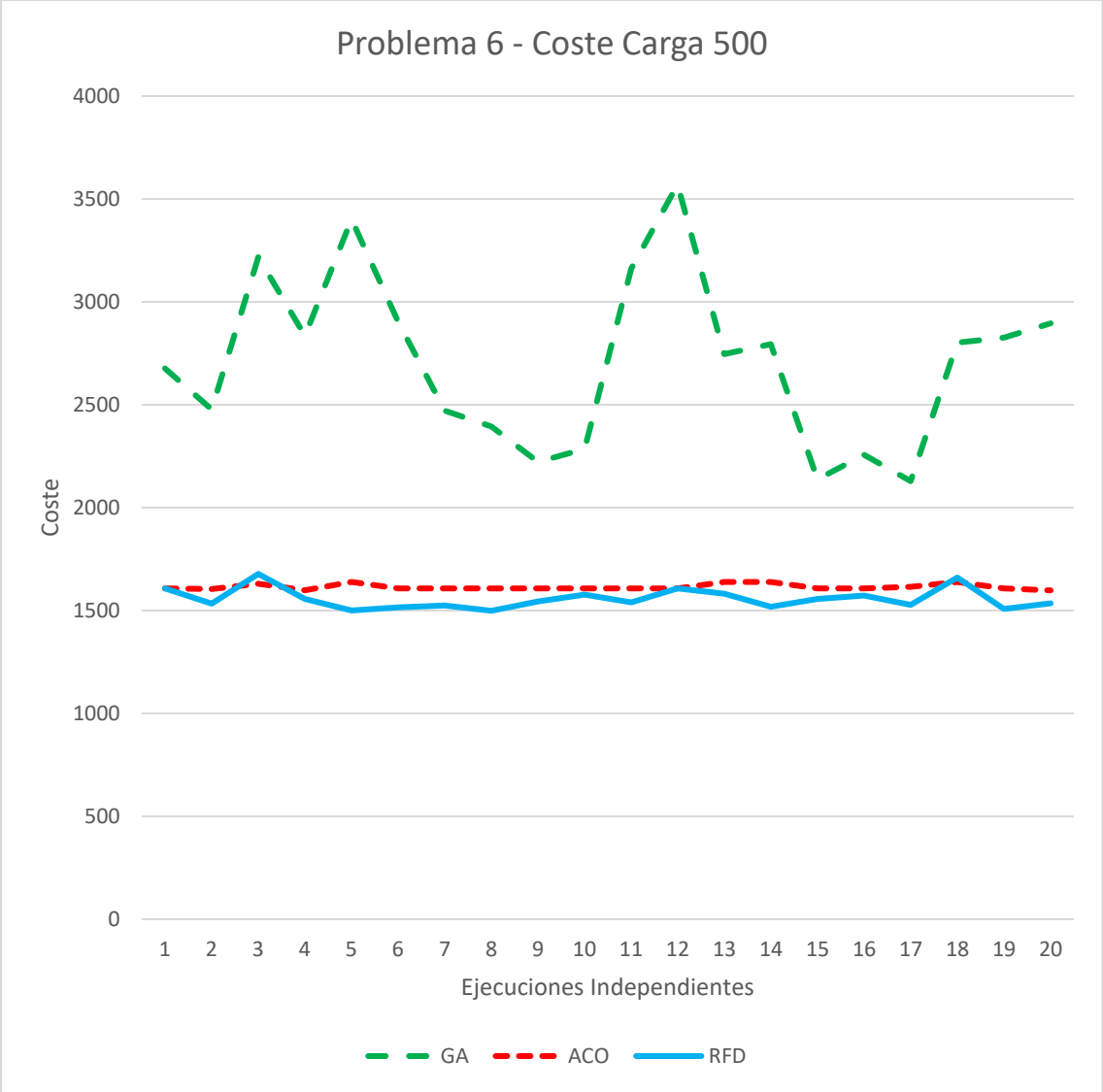

Figura 6.99 - Problema 6 - Coste de Carga 500

RFD obtiene el menor coste de todos de 1499, media de 1555,63 y una desviación típica de 48,59. ACO tiene el segundo puesto, con una mejor solución de 1598 y una desviación típica de 13,92, resultados que presentan unas diferencias relativamente menores a las encontradas en el problema previo, acercándose ligeramente a los resultados de RFD. GA es el que peor funciona de los tres métodos en esta prueba, con un mejor coste de 2129, media de 2650,96 y una desviación de 406. Random llega a conseguir un menor coste de 14002 y una media de 15465,83.

Como se ha podido observar en las tres pruebas (y en los problemas previos) GA se comporta claramente peor en máquinas de estados con baja densidad, mientras que se acerca más al coste de ACO y RFD (sin cargas) conforme aumenta la densidad. ACO en este problema se muestra más cercano a RFD, aunque la diferencia sigue siendo acusada. RFD se ha comportado de una forma excelente en todos los problemas realizados, siendo imbatible en todos ellos.

6.3. Conclusiones

A continuación, se presenta una tabla con todos los resultados obtenidos por las mejores configuraciones de los algoritmos, tanto en el problema de entrenamiento como en los problemas de comparación:

Método	Problema	Tamaño del problema	Densidad de conexiones	Coste de carga	Mejor obtenido	Media	Desviación típica
RFD	Problema 1	50 Nodos	50%	1	71	121,65	20,54
ACO	Problema 1	50 Nodos	50%	1	131	138,57	6,96
GA	Problema 1	50 Nodos	50%	1	257	348,55	52,87
RANDOM	Problema 1	50 Nodos	50%	1	518	618,21	56,04
RFD	Problema 1	50 Nodos	50%	20	179	199,65	13,37
ACO	Problema 1	50 Nodos	50%	20	215	229,1	5,88
GA	Problema 1	50 Nodos	50%	20	323	392,05	43,57
RANDOM	Problema 1	50 Nodos	50%	20	478	642,56	72,51
RFD	Problema 1	50 Nodos	50%	500	213	225,35	8,46
ACO	Problema 1	50 Nodos	50%	500	253	253	0
GA	Problema 1	50 Nodos	50%	500	325	367,8	31,33
RANDOM	Problema 1	50 Nodos	50%	500	575	655,56	49,79
RFD	Problema 2	100 Nodos	50%	1	198	228,1	23,22
ACO	Problema 2	100 Nodos	50%	1	273	301,55	11,95
GA	Problema 2	100 Nodos	50%	1	657	708,9	23,21
RANDOM	Problema 2	100 Nodos	50%	1	2310	3323,08	377,95
RFD	Problema 2	100 Nodos	50%	20	404	453,35	24,5
ACO	Problema 2	100 Nodos	50%	20	562	585,65	8,6
GA	Problema 2	100 Nodos	50%	20	678	797,95	69,29
RANDOM	Problema 2	100 Nodos	50%	20	2331	3261,83	311,12
RFD	Problema 2	100 Nodos	50%	500	517	549,75	19,25
ACO	Problema 2	100 Nodos	50%	500	609	617,2	5,36
GA	Problema 2	100 Nodos	50%	500	624	702,2	37,61
RANDOM	Problema 2	100 Nodos	50%	500	2778	3775,53	432,34
RFD	Problema 3	200 Nodos	50%	1	432	480,95	22,03
ACO	Problema 3	200 Nodos	50%	1	482	528,74	18,96
GA	Problema 3	200 Nodos	50%	1	1139	1247,35	48,77
RANDOM	Problema 3	200 Nodos	50%	1	7981	9329,89	593,11
RFD	Problema 3	200 Nodos	50%	20	822	889,35	32,26
ACO	Problema 3	200 Nodos	50%	20	977	1010,53	19,32
GA	Problema 3	200 Nodos	50%	20	1428	1569,05	81,19
RANDOM	Problema 3	200 Nodos	50%	20	8526	9468,93	535,94
RFD	Problema 3	200 Nodos	50%	500	1053	1103,35	18,01
ACO	Problema 3	200 Nodos	50%	500	1131	1144,85	6,62
GA	Problema 3	200 Nodos	50%	500	1136	1243	43,86
RANDOM	Problema 3	200 Nodos	50%	500	9067	10542,86	670,12
RFD	Problema 4	50 Nodos	10%	1	262	285,67	17,94
ACO	Problema 4	50 Nodos	10%	1	300	311,06	7,61
GA	Problema 4	50 Nodos	10%	1	679	872,39	123,00
RANDOM	Problema 4	50 Nodos	10%	1	1675	2063,98	151,27
RFD	Problema 4	50 Nodos	10%	20	326	352,89	12,58
ACO	Problema 4	50 Nodos	10%	20	376	394,97	8,26
GA	Problema 4	50 Nodos	10%	20	690	939,58	191,04
RANDOM	Problema 4	50 Nodos	10%	20	1769	2333,85	216,68
RFD	Problema 4	50 Nodos	10%	500	416	430,18	13,72
ACO	Problema 4	50 Nodos	10%	500	470	480,11	9,90
GA	Problema 4	50 Nodos	10%	500	680	931,49	155,34
RANDOM	Problema 4	50 Nodos	10%	500	2132	2756,08	324,18
RFD	Problema 5	100 Nodos	10%	1	280	339,50	21,75
ACO	Problema 5	100 Nodos	10%	1	348	377,66	14,66
GA	Problema 5	100 Nodos	10%	1	1203	1404,35	106,38
RANDOM	Problema 5	100 Nodos	10%	1	5154	6175,78	439,63
RFD	Problema 5	100 Nodos	10%	20	489	525,44	14,41
ACO	Problema 5	100 Nodos	10%	20	637	676,14	18,51
GA	Problema 5	100 Nodos	10%	20	1316	1564,56	169,22
RANDOM	Problema 5	100 Nodos	10%	20	5102	6086,85	358,65
RFD	Problema 5	100 Nodos	10%	500	801	874,45	50,66
ACO	Problema 5	100 Nodos	10%	500	1075	1121,86	22,17
GA	Problema 5	100 Nodos	10%	500	1292	1454,91	99,52
RANDOM	Problema 5	100 Nodos	10%	500	5415	6340,66	422,88
RFD	Problema 6	200 Nodos	10%	1	678	783,76	36,00
ACO	Problema 6	200 Nodos	10%	1	770	819,44	18,09
GA	Problema 6	200 Nodos	10%	1	1937	2194,49	221,54
RANDOM	Problema 6	200 Nodos	10%	1	14004	14832,43	543,87
RFD	Problema 6	200 Nodos	10%	20	1111	1169,45	30,30
ACO	Problema 6	200 Nodos	10%	20	1195	1247,52	22,97
GA	Problema 6	200 Nodos	10%	20	2268	2671,91	499,50
RANDOM	Problema 6	200 Nodos	10%	20	13467	15306,64	811,31
RFD	Problema 6	200 Nodos	10%	500	1499	1555,63	48,60
ACO	Problema 6	200 Nodos	10%	500	1598	1613,83	13,92
GA	Problema 6	200 Nodos	10%	500	2129	2650,96	406,02
RANDOM	Problema 6	200 Nodos	10%	500	14002	15465,84	994,13

Figura 6.100 – Tabla comparativa de la actuación de los algoritmos en los problemas definidos

Tras ver los resultados obtenidos en los puntos previos, es hora de sacar conclusiones:

1) RFD es el algoritmo que mejor se ha comportado en la mayoría de las pruebas llevadas a cabo. Además de obtener unos resultados bastante buenos en casi todas las ejecuciones realizadas, también suele sacar los mejores resultados en general comparando con el resto de los métodos estudiados. Desde un principio, se esperaba que RFD fuese uno de los mejores algoritmos en tratar de resolver MLS [1], por lo que ha cumplido con lo esperado.
2) ACO se lleva el segundo puesto, ya que siempre va a la cola de RFD sacando unos resultados bastante "cercanos". Debido a la adaptación de ACO en este problema, se notan las pérdidas de eficiencia en tiempo en comparación con GA y RFD. También ha cumplido las expectativas, pues desde un principio se suponía que la pelea iba a estar entre ACO y RFD, aunque es cierto que se esperaba un mejor comportamiento de ACO superando a RFD en algunas pruebas. La diferencia de resultados entre ACO y RFD oscila entre un 3% y un 24%, con una mayor diferencia en los escenarios con coste de carga 20 y una menor en los escenarios con coste de carga 500.
3) GA se ha comportado de una forma sorprendente, pues en un planteamiento inicial no se pensaba que fuese a llegar a sacar resultados si quiera comparables a RFD y ACO, pero se ha mostrado realmente competitivo en algunas pruebas y sobre todo en el problema de entrenamiento, cuyos valores son mucho más cercanos a los que obtienen RFD y ACO. En términos de eficiencia en tiempo a la hora de sacar resultados, es mucho más rápido que los dos algoritmos mencionados. Un cruce más inteligente podría ayudarle a acercarse mucho más a los resultados de ACO y RFD, sobre todo en el momento de realizar cargas, aunque la operación de este cruce podría ser muy costosa.
4) RANDOM se ha comportado tal y como se esperaba, sacando unos resultados que no son comparables con los otros algoritmos probados, y demostrando que las heurísticas se están comportando bien.

7. CONCLUSIONES DEL TRABAJO

7.1. Español

Una vez finalizado tanto el desarrollo de la aplicación, como la realización del estudio del comportamiento de las metaheurísticas en el problema de la secuencia de carga mínima (Minimum Load Sequence – MLS), se va a realizar una reflexión sobre el trabajo realizado.

En primer lugar, se ha visto que el desarrollo y adaptación de las metaheurísticas que se tratan en este trabajo (RFD, GA y ACO) ha sido más complejo de lo que se esperaba en un principio debido a las características propias del problema MLS. En todos los algoritmos que se pretendían adaptar se ha tenido que variar el esquema general, no solo para que cumpliese las reglas del problema estudiado, si no para que su comportamiento con este fuese mínimamente decente.

El algoritmo que más ha costado adaptar ha sido RFD, pues su esquema general trata de premiar cada movimiento de gota que se realiza, obteniendo una mejora local y no una visión global de toda la solución. Pero antes de llegar a la conclusión previa, se realizaron distintas modificaciones en el esquema general como compensación de alturas o el uso de una erosión fija por gota, que no obtuvieron buenos resultados. Al llegar a la conclusión de que se deberían premiar los caminos encontrados de forma global -respecto al coste total obtenido- se implementaron gotas que cubriesen una solución entera, y la erosión se produciría respecto a ellas en función del coste de cada solución encontrada.

ACO también ha tenido que sufrir modificaciones, puesto que el esquema general propone que la deposición se haga cuando se llega al destino, sin embargo, en MLS no es recomendable -y no obtiene buenos resultados- incentivar el uso de un camino desde el nodo inicio al nodo destino, pues no tienen por qué cubrirse todos los nodos críticos especificados. En la adaptación realizada (tras el aprendizaje del desarrollo de RFD) se llegó a la conclusión en la que depositar alicientes cuando se ha realizado una solución completa (un árbol) conseguía mejores resultados.

GA ha sido el esquema más fácil de adaptar al problema -se esperaba que fuese el más complejo de los tres- aunque si se observan los resultados del estudio, estos están alejados de los que obtienen RFD o ACO.

Tras tener las adaptaciones listas para poder realizar el estudio del comportamiento de estas en MLS, se han extraído las siguientes conclusiones de los resultados obtenidos:

En cuanto a coste computacional, debido a la adaptación de ACO y RFD a este problema, tienen un coste mayor que GA, cuyo tiempo para terminar una ejecución es más rápido, pero no llega a obtener la calidad de soluciones que los otros dos métodos, aunque supera con creces las expectativas iniciales, en las que se pensaba que en ningún problema podría llegar a acercarse a RFD y ACO.

Los mejores resultados se obtienen por RFD en todos los casos probados, seguido por ACO, aunque, con unas notables diferencias, más de las esperadas en el inicio del proyecto.

Se ha conseguido adaptar los 3 algoritmos al problema, aunque GA sea el que peor comportamiento tiene, se piensa que si se combinase con aprendizaje incremental [12], junto con un cruce y generación inicial más inteligentes, podría llegar a tener una cierta mejora.

Como se ha comentado antes, ACO y RFD son las dos metaheurísticas que tienen un coste computacional más elevado, por lo que sería interesante optimizar las estructuras y tratar de primar la eficiencia en algunos puntos, como puede ser la introducción de concurrencia, sobre todo en ACO pues es el algoritmo que más tarda en terminar sus ejecuciones.

Además de lo explicado previamente, otra de las cosas que hubiese sido de ayuda, sería la modificación de la interfaz gráfica de la aplicación para que la representación de la máquina de estados y de la solución obtenida fuesen más visuales, facilitando la comprensión de resultados. Junto con esta mejora, también sería interesante que el fichero generado fuese un archivo Excel para poder post-procesar o estudiar los datos de una forma más sencilla, o incluso la implementación de un script que tratase los datos.

Para finalizar, se ha llegado a cumplir todos los objetivos propuestos: se ha adaptado las metaheurísticas para MLS y se ha realizado un estudio comparativo de estos para comprobar como se comportan con el problema. Es cierto que en los párrafos anteriores se han comentado algunos puntos de mejoras, pero en general, los objetivos propuestos se han superado, con ciertas dificultades debido al complejo desarrollo de las adaptaciones de las metaheurísticas, pero con un resultado final correcto. Además, se está considerando la posibilidad de continuar la línea de trabajo seguida hasta ahora con el propósito de publicar un artículo de investigación que recoja los resultados y las conclusiones más importantes halladas. Para ello, el objetivo es profundizar en los estudios ya realizados así como realizar más pruebas que permitan obtener nuevos resultados aún más precisos.

7.2. Inglés

Once finished the development of the application and the study of the behaviour of the three metaheuristics proposed in this project on the problem of the Minimum Load Sequence – MLS, a reflection will be made about the work done.

On the first hand, the development and adaptation of the three metaheuristics studied in this work (RFD, GA y ACO) has been more difficult than expected at the beginning, due to own peculiarities of MLS. In all the algorithms, has been done changes in the general schema, not only to pass the specific rules of the problem, besides to improve the behaviour of them in it:

RFD has been the more complex algorithm to adapt, due it's own general schema erodes every movement of a drop, getting a local improvement and not got a global view of the full solution. Before discovering the previous idea, had been implemented a variety of ideas, as compensation of altitude of the nodes after erosion or the use of a fixed value for the erosion that drops produces, that finally didn't get good results. When the previous idea of eroding the solutions globally -due total cost obtained-, developed drops that found a full solution, and the erosion will be done in respect of the solution found in function of the cost.

ACO had been modified too, because the general schema shows that an ant has to drop pheromones when it arrives the destiny, however, in MLS is not recommendable -and the doesn't get good solutions- to do this action, because the path found doesn't have to fill all the nodes specified in the problem. In the adaptation done (after learning in the development of RFD) a conclusion has achieved, drop pheromones when a complete solution has done (a tree) get better results.

GA had the easiest adaptation -the initial think was it will be the hardest of the three algorithms-, however, if take a fast look to the results of the study, the results got by GA are worst than RFD or ACO.

After the adaptations of the three metaheuristics were ready to study the behaviour of them in MLS, the next conclusions are reached:

In terms of computational cost, due to ACO and RFD adaptation to this problem, they have higher cost than GA, that the time of it to finish an execution is faster, but it cannot reach the quality of the solutions of the other two, although GA beats the initial thinks.

The best results have been reached by RFD, followed by ACO with substantial differences, more than expected.

Finally, the adaptation of the three algorithms was reached, although GA behaviour is the worst, it will be better if mix GA with Incremental Learning [12], in addition with more intelligent method of reproduction or generation.

As commented before, RFD and ACO are the methods with higher computational cost, it could be interesting improve the data structures and try to improve the efficiency in some points, like introducing parallelism, especially in ACO because is the slowest algorithm finishing the executions.

Other thing that can help, is the modification of the graphical user interface (GUI) of the application to show the state machine or the solution in more visual way, helping

to understand the results obtained. In addition to the previous improvement, could be nice to generate an Excel file better than a txt, to improve the post-process of the data, or even develop a script to process the data.

To conclude, all the objectives has been achieved: has been adapted the three metaheuristics to MLS and done the study of its behaviour in the problem. It is right the fact that in previous paragraphs have been proposed improvements over the work -but in general- the objectives had been completed with difficulties due to hard development of the metaheuristics adaptations, but with a correct result. In addition, the possibility of continuing the line of work followed so far with the purpose of publishing a research article that includes the most important results and conclusions found is being considered. For this, the objective is to deepen the studies already carried out as well as to carry out more tests that allow obtaining even more precise results.

8. CONTRIBUCIONES PERSONALES

8.1. Adrián Muñoz Gámez

En el inicio del proyecto -junto con mis compañeros- empezamos diseñando la estructura de clases y organizando el trabajo juntos, mediante herramientas de comunicación online. Decidimos en qué lenguaje realizar el proyecto, y la forma de trabajo, que al principio fue la mayoría del tiempo en sesiones conjuntas para no perder detalle del comienzo proyecto, y más adelante -cuando ya conocíamos el proyecto- se decidió paralelizar las tareas.

El lenguaje que decidimos utilizar fue C#, ya que no conocíamos mucho como era y nos apetecía aprenderlo debido a su uso en distintos ámbitos, como la programación de videojuegos o el *Framework* de desarrollo .NET. Para no tener una visión superficial y sacar el máximo potencial al lenguaje, comencé a buscar información de sus características más destacadas, y pedir consejo de las prácticas usadas a compañeros del trabajo que usan diariamente este lenguaje. Tras esta búsqueda, una de mis contribuciones personales que más hemos usado en el desarrollo de la aplicación, es el aprendizaje y uso de *Linq*, que permite tener un código más legible y obtener de una forma muy ágil y sencilla subconjuntos de datos, con una sintaxis muy parecida a SQL. Otra de mis contribuciones de cara al desarrollo (y característica destacada de C#) fue la abstracción de la clase Arista, permitiendo especificarla a las características de cada problema, puesto que RFD usa aristas con Altura y ACO deposita feromonas en estas.

Al tener las primeras clases desarrolladas, y empezar a paralelizar el trabajo, vimos que íbamos a necesitar versionar los cambios que realizábamos. Para el versionado del proyecto, aprovechamos un servidor NAS del que dispongo, permitiéndonos asegurar nuestros avances en todo momento de forma privada.

Conforme avanzamos en el desarrollo del proyecto, vimos que el esquema general del RFD no se comportaba bien con MLS. Decidimos repartirnos el trabajo, para no quedarnos estancados hasta que RFD fuese funcional.

Tras este reparto, la adaptación de RFD quedó de mi parte, y tuve que realizar distintas implementaciones hasta llegar a la actual. Entre las implementaciones realizadas, probé con una que compensase la altura de los nodos, para que no hubiese en ningún momento pendientes negativas -solución que tenía pésimos resultados-; más adelante probé a cambiar el método de selección del esquema general del RFD, por otros como Ruleta, para comprobar si se obtenían mejores resultados o se producían más variaciones en estos -solución que tampoco terminó de funcionar-; también hubo un momento en que utilizábamos erosiones estáticas para que fuese más controlada, y que la pendiente no influyese tanto debido a las pendientes con el nodo 0… Finalmente llegué a la conclusión de que las soluciones deberían valorarse globalmente -puesto que las implementaciones previas premiaban mucho las transiciones entre estados, pero el conjunto no era el mejor posible-, además también

llegué a la idea de que se deberían generar muchas soluciones parciales en cada iteración, para que la evaluación global se diese en la mayor variedad de casos posibles. Este cambio de planteamiento, junto con la inclusión de gotas trepadoras (soluciona el problema en el inicio del algoritmo de las pendientes inversas) llegué a la implementación actual, donde cada gota es una solución completa que se evalúa y erosiona globalmente.

Al llegar a una solución competente, observé que RFD necesita todos los mecanismos que implementa, y es mejor no tratar de realizar pruebas de resultados hasta que todos funcionen correctamente, pues las conclusiones que se pueden llegar a extraer pueden dar lugar a frustración y pensar que las funcionalidades implementadas no son correctas: RFD no es la suma de sus partes, si no el conjunto de estas.

Una vez RFD comenzó a sacar buenos resultados, al realizar unas comparaciones previas, vimos que ACO no llegaba a acercarse a estas soluciones y se decidió aplicar un razonamiento parecido a RFD en ACO, haciendo que las hormigas recorriesen un árbol entero, y considerando esto para la deposición de feromonas.

Una vez comenzamos a tener listos apartados de la memoria, unifiqué estos y apliqué el estilo para la memoria final. Los apartados que he realizado son los referentes a RFD, debido a que era el que mejor lo conocía, puesto que -tal como comenté antes- realicé el estudio del esquema general y el desarrollo del algoritmo final.

Además de lo anterior, también decidí post-procesar los datos que obteníamos de las aplicaciones (texto plano), para obtener gráficas que facilitasen el estudio de los algoritmos y la comparación de estos, además de mejorar la comprensión lectora de las explicaciones asociadas gracias el estímulo visual.

Por último, quiero comentar que tras realizar el estudio de los algoritmos y la comparativa entre ellos, todos hemos participado en la obtención de conclusiones y en el apartado de trabajo futuro, donde cada uno hemos expuesto y consensuado las cosas que se han quedado por el camino, y las cosas que podría llegar a ser interesantes mejorar o implementar.

8.2. Andrés Pascual Contreras

Al inicio del Trabajo de Fin de Grado (en adelante TFG) comenzamos a trabajar de forma habitual los tres juntos. Normalmente utilizábamos la herramienta de video comunicación Skype, ya que nos permitía compartir el escritorio. Gracias a ello, podíamos mostrarnos en tiempo real el trabajo que estábamos realizando cada uno. Además, podíamos centrar la atención de todos los miembros del equipo sobre uno para aportar ideas en diferentes puntos de la programación de la aplicación, así como en las labores de planificación, diseño e investigación.

Decidimos empezar a trabajar de esta manera para que todos los miembros del equipo conociéramos las bases fundamentales del trabajo que estábamos realizando y pudiéramos conocer y participar todos en las decisiones más importantes que sentarían las bases de todo el TFG.

Posteriormente, cuando llevábamos parte del trabajo hecho decidimos repartirnos las tareas para poder progresar de forma más eficiente, sobre todo en términos de tiempo, ya que por nuestra situación (los tres miembros hemos estado trabajando) debíamos optimizar lo más posible el tiempo del que disponíamos. Sin embargo, a pesar de la necesaria repartición de tareas, nos hemos mantenido en comunicación constante durante la elaboración de estas y ha sido habitual el intercambio de información y ayuda entre las partes realizadas por todos.

Respecto a las aportaciones netamente personales, al comienzo del trabajo me dedique a diseñar la arquitectura de la aplicación. La primera decisión consistió en elegir las tecnologías a utilizar. En un primer momento valoramos utilizar Java con el IDE Eclipse, ya que estábamos familiarizados con él debido a varias asignaturas cursadas en la carrera. Sin embargo, yo había empezado hace unos meses a trabajar en una consultoría donde desarrollaban en C#. Plantee la opción de utilizar C# con el *Framework* de desarrollo .NET, utilizando el IDE Visual Studio. Aunque el IDE sí lo habíamos utilizado, mis compañeros nunca habían programado en C# y yo estaba aprendiendo, por lo que nos pareció buena idea para aprovechar la elaboración del TFG para conocer y utilizar nuevas tecnologías.

El desarrollo de la aplicación, aunque es fundamental para nuestro trabajo, no es sino un medio para un fin que es la investigación y el estudio de los algoritmos para resolver el problema, tal y como ya se ha mencionado en la presente memoria. Por ello, decidí esbozar una arquitectura lo más sencilla, utilizando lo estrictamente necesario para que la herramienta cumpliera su función. Además, otra de las cosas que más me importaba es que se pudiese compartimentar fácilmente de cara a que pudiéramos trabajar cada uno en ella por separado y luego unir las distintas partes de forma sencilla, tan solo modificando las clases concretas que cada uno había tocado teniendo las mínimas dependencias posibles.

Cuando terminé de construir el esqueleto y programar una mínima funcionalidad, comenzamos a trabajar juntos por un largo período de tiempo. Nos turnábamos para

programar compartiendo escritorio mediante Skype mientras que los otros dos miembros aportaban ideas o correcciones. Después nos dividimos el trabajo que quedaba en la aplicación. A partir de ese momento me centré en diseñar la interfaz de usuario, desarrollar las vistas y la lógica aplicada a los procesos de recogida y muestra de información y de validación de los datos.

Dividí la aplicación en dos partes fundamentales: por un lado, las herramientas para generar los casos de prueba y por otro lado para manipular cada una de las tres implementaciones de los algoritmos. Mi objetivo principal era que resultara una herramienta cómoda, fácil y rápida de usar para que nos ayudara lo más posible en el proceso de investigación.

Después, añadí algunas funcionalidades extra que no estaba pensadas al principio para facilitar la gestión de múltiples casos de prueba, como la posibilidad de repetir múltiples ejecuciones completas de un algoritmo, o la de guardar archivos con los resultados de cada prueba.

Me resultó particularmente útil mi experiencia en el trabajo donde había utilizado ya esta tecnología en proyectos reales en la parte *Front*.

En lo relativo al trabajo de investigación en sí y en la elaboración de la memoria, mi contribución ha consistido en la realización de los siguientes apartados que expondré a continuación. En primer lugar, realicé el resumen y su traducción al inglés. Posteriormente abordé el apartado de Descripción del Problema, para lo cual investigué sobre teoría de la complejidad computacional de cara a poder explicar la magnitud de nuestro problema, así como en el artículo que sirvió de base a la elaboración de nuestro proyecto [1]. Después elaboré la introducción del apartado Propuesta de Solución, donde se resumen los métodos que se detallarán más adelante para resolver el problema. A continuación, redacté el Desarrollo de la Aplicación hablando de algunas de las cosas que cité anteriormente en las contribuciones, tales como las decisiones iniciales y la estructura de la aplicación, para terminar, detallando más a fondo el funcionamiento de las distintas clases que conforman el código, resaltando el funcionamiento de los algoritmos más importantes intentando explicarlos de manera comprensible para facilitar su lectura y entendimiento.

Por último, en la labor de investigación, me encargué de estudiar el Algoritmo Genético, para tratar de hallar su configuración óptima de parámetros a la hora de resolver el problema, con el objetivo de obtener los mejores resultados para compararlos con los de mis compañeros. Para esta tarea me resultó especialmente valiosa mi experiencia previa manejando este tipo de algoritmo, obtenida gracias a la asignatura optativa de Programación Evolutiva. Este proceso puede observarse en el subapartado de la memoria de Algoritmo Genético (GA), dentro del apartado de Análisis de Resultados.

8.3. José Manuel Pérez Zamorano

En el desarrollo de la aplicación

En la parte en la que desarrollamos las estructuras principales con las que representamos el problema trabaje junto a mis compañeros y durante los tres primeros meses estuve investigando y aportando ideas para adaptar el algoritmo RFD al problema de MLS, la estructura que representa una solución al problema tome un papel importante sobre todo a la hora de explicarle a mis compañeros cómo funcionaba el algoritmo de Dijkstra. También definí la clase arista, en un principio pensamos en la idea de utilizar dos matrices (una para representar las alturas o las feromonas según el problema y otra para los costes).

Durante la primera fase de adaptación de RFD trabajamos todos juntos, ya que era el único algoritmo que no conocíamos ninguno de los tres y preferimos inicialmente aportar todos ideas. Debido al uso de la pendiente y Dijkstra yo desarrolle una nueva versión de la erosión que, aunque no conseguía grandes resultados era funcional. También hice una gran cantidad de pruebas y resolví el problema de los pozos, en el que una gota se quedaba en un pozo y no podía ir hacia ninguna dirección. Mi solución consistió en evitar la formación de estos pozos fijándome en la erosión de la gota en la iteración anterior y más modificaciones en las funciones de erosión. También empecé a barajar la idea que posteriormente se implementó de guardar el camino que hace la gota y erosionar al final, para que así no se pisen los caminos que forma una gota con los que forma otra durante su travesía hasta el nodo final, aunque en un principio descarte esa idea finalmente se guarda no solo eso sino un árbol solución completo.

Después de estas aportaciones me puse a trabajar en la implementación del algoritmo de colonia de hormigas, mi primera aportación fue la creación de la clase hormiga y en definir el comportamiento de las hormigas, inicialmente era un algoritmo sencillo en el que las hormigas iban desde un nodo critico hasta el final y durante el proceso iban dejando un rastro de feromonas, después de la realización de muchas pruebas decidí que esta solución no era correcta, ya que a nivel local valoraba muy bien los mejores caminos en coste, pero a nivel global no se valoraba el camino completo de manera global. Durante este periodo de pruebas también ajuste el número de feromonas depositadas y la evaporación de feromonas.

Hormigas dio un gran paso hacia delante después de tomar la decisión de dejar de depositar de manera local y pasar a depositar de manera global, este proceso fue lo que marcó la diferencia, en ese momento las hormigas pasaron de depositar según iban pasando por cada nodo a depositar al llegar al nodo final, el número de feromonas depositadas por cada hormiga antes era siempre el mismo. Pero después de tomar la decisión de guardar el camino decidí depositar según la calidad del camino seguido por la hormiga para llegar al destino, este fue el punto de inflexión que hizo que el algoritmo pasara a tener mejores resultados y que posteriormente le sugiriese a Adrián Muñoz hacer algo parecido con el algoritmo RFD.

Adrián aplico un montón de cambios en RFD y al final yo tuve que aplicar muchas de sus ideas en ACO, principalmente cambie el comportamiento de las hormigas, las cuales en la versión final también van guardando un árbol solución y depositan feromonas según la calidad de la solución. La otra gran modificación fue introducir el concepto de hormigas locas que es parecido al concepto de gota trepadora, aumentando el espacio de búsqueda del algoritmo.

El desarrollo de algoritmo genético también lo fui desarrollando paralelamente con hormigas. Primero pensé como iba a representar el individuo, lo cual me llevo casi una semana pensando formas de hacerlo y como repercutiría esto en la cantidad de información que almacenaba este y también en la dificultad que tendría la implementación de los operadores genéticos.

Después de tomar la decisión de utilizar el propio árbol solución como genoma para nuestro individuo. Me puse manos a la obra y diseñé varios tipos de cruce y de mutación, en la versión final únicamente se quedaron un cruce y dos mutaciones. El uso de árboles me dio un montón de problemas, pero a pesar de ello después de muchas pruebas a lo largo de unas 3 semanas conseguí que funcionasen correctamente.

En la última etapa del desarrollo probé varias técnicas de cruce y al final dejé solo ruleta, estocástica y torneo. Estos cruces yo noté que no estaban dando buenos resultados, pero después de dejar parado el proyecto de genético un mes descubrí que las selecciones a veces daban un problema al copiar los individuos y por eso no funcionaban bien. Arregle ese problema y los resultados de genético aun siendo peores que los del resto de algoritmos, son más que aceptables sobre todo para maquinas grandes.

En la parte final del desarrollo implemente un generador aleatorio de soluciones para poder comparar los resultados con el resto de los algoritmos.

En la parte de la vista exclusivamente enlacé la lógica de los algoritmos genético y hormigas con la vista siguiendo las indicaciones de Andrés Pascual, introduje un panel para poder limitar el tiempo de ejecución y hacer pruebas en un tiempo concreto e introduje nuevos parámetros en la parte del algoritmo de colonias de hormigas para poder hacer pruebas con los parámetros introducidos en la última iteración de la adaptación.

En el desarrollo de la memoria

Realice un gran trabajo de búsqueda de antecedentes a nuestros problemas y a las propuestas de solución, también redacte parte de ese apartado en el que explico varios usos de RFD y del algoritmo de hormigas.

En cuanto a redacción de propuestas de solución, he redactado las partes de algoritmo de colonias de hormigas y algoritmo genético. En su realización primero busque cómo funcionaban los conceptos base del algoritmo en la naturaleza y posteriormente como

se suelen implementar de manera más general, por último, explico cómo se han adaptado concretamente en este problema.

En cuanto al análisis de resultado, mi cometido en este apartado fue realizar todas las pruebas y conclusiones del algoritmo ACO.

9. REFERENCIAS

[1] P. Rabanal, I. Rodríguez and F. Rubio, "Testing restorable systems: formal definition and heuristic solution based on river formation dynamics", *Formal Aspects of Computing*, vol. 25, no. 5, pp. 743-768, 2012.

[2] R. Lefticaru and F. Ipate, "Functional Search-based Testing from State Machines", *2008 International Conference on Software Testing, Verification, and Validation*, 2008.

[3] G. Redlarski, M. Dabkowski and A. Palkowski, "Generating optimal paths in dynamic environments using River Formation Dynamics algorithm", *Journal of Computational Science*, vol. 20, pp. 8-16, 2017.

[4] K. Guravaiah and R. Leela Velusamy, "Energy Efficient Clustering Algorithm Using RFD Based Multi-hop Communication in Wireless Sensor Networks", *Wireless Personal Communications*, vol. 95, no. 4, pp. 3557-3584, 2017.

[5] S. Kashef and H. Nezamabadi-pour, "An advanced ACO algorithm for feature subset selection", *Neurocomputing*, vol. 147, pp. 271-279, 2015.

[6] S. Khan and A. Baig, "Ant colony optimization based hierarchical multi-label classification algorithm", *Applied Soft Computing*, vol. 55, pp. 462-479, 2017.

[7] T. İnkaya, S. Kayalıgil and N. Özdemirel, "Ant Colony Optimization based clustering methodology", *Applied Soft Computing*, vol. 28, pp. 301-311, 2015.

[8] P. Sharkey, *Ant Colony Optimisation: Algorithms and Applications*. 2014.

[9] A. Sharma, R. Patani and A. Aggarwal, "Software Testing Using Genetic Algorithms", *International Journal of Computer Science & Engineering Survey*, vol. 7, no. 2, pp. 21-33, 2016.

[10]P. Rabanal, I. Rodríguez and F. Rubio, "Applications of river formation dynamics", *Journal of Computational Science*, vol. 22, pp. 26-35, 2017.

[11] T. Cormen and C. Leiserson, *Introduction to algorithms, 3rd edition*, 3rd ed. pp. 658-664.

[12] M. Verleysen, *24th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. pp. 357-368.

[13] Rincón JC, *Aplicación de algoritmo genético en la optimización del sistema de abastecimiento de agua de Barquisimeto-Cabudare, Avances en Recursos Hidráulicos, Nº14, 10/2006*, pp. 25-38

[14] C. Blum, *"Ant colony optimization: Introduction and recent trends", Physics of Life Reviews, vol. 2, no. 4*, pp. 353-373, 2005.

[15] F. Caparrini and W. Work, *"Algoritmos de hormigas y el problema del viajante - Fernando Sancho Caparrini", Cs.us.es, 2018. [Online]. Available: http://www.cs.us.es/~fsancho/?e=71.* [Accessed: 10- Ago- 2018].

[16] C. Cervigón and L. Araujo, *Transparencias Programación Evolutiva [PEV] - Facultad de Informática Universidad Complutense de Madrid. 2015.*

[17] P. Rabanal, *Algoritmos heurísticos y aplicaciones a métodos formales, Tesis Doctoral, 2010.*

[18] C. Anderson-Cook, "Practical Genetic Algorithms", Journal of the American Statistical Association, vol. 100, no. 471, pp. 1099-1099, 2005.

[19] P. Rabanal, I. Rodríguez and F. Rubio, "Using River Formation Dynamics to Design Heuristic Algorithms", *Lecture Notes in Computer Science*, pp. 163-177.

[20] P. McMinn, "Search-based software test data generation: a survey", *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105-156, 2004.

[21] L. Davis, *Handbook of genetic algorithms*. New York: Van Nostrand Reinhold, 1991.

[22] M. Dorigo and T. Stützle, *Ant colony optimization*. Cambridge, Mass.: MIT Press, 2004.

[23] https://github.com/Josepe04/TFG.