Trabajo Fin de Grado
Grado en Ingeniería en Electrónica, Robótica y Mecatrónica.

Desarrollo y calibrado de un algoritmo evolutivo para la resolución de problemas de optimización.

Autor: Ángel Rivero Cides
Tutor: Alejandro Escudero Santana

Dpto. Organización Industrial y Gestión de Empresas II
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Desarrollo y calibrado de un algoritmo evolutivo para la resolución de problemas de optimización.

Autor:
Angel Rivero Cides

Tutor:
Alejandro Escudero Santana
Profesor contratado doctor

Dpto. Organización Industrial y Gestión de Empresas II
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

A mi familia

A mis maestros

Agradecimientos

Mi primer agradecimiento es para mi novia, Bea, la cual es la que más ha tenido que aguantar durante estos duros 4 años y no estaría aquí sin su apoyo, junto a mi familia la cual ha hecho un gran esfuerzo para costearme los estudios, como a mis amigos, David, Carlos, Fernando, Isidro, Samuel y Jorge con los que he compartido todo este tiempo a la ETSI haciendo los malos momentos más llevaderos, hablando de amigos no podían faltar mis "mahtuersos" Andrés, Carlos, Carlos, Jesús, Álvaro, Manu y Marcos a los que he tenido que dejar de ver tanto como me gustaría por exigencias de la carrera y aun así siempre han estado ahí. Y, por último, pero no menos importante a mi tutor por su ayuda en este proyecto y a todo el profesorado que me han enseñado.

Resumen

En el presente Trabajo Fin de Grado el principio objetivo se trata de desarrollar y calibrar un algoritmo evolutivo, el algoritmo NEMO Search, para la resolución de problemas de optimización. Este algoritmo está inspirado en el comportamiento de los peces payasos. Los problemas que se han escogidos para la prueba del algoritmo son el VRPTW y el problema de Steiner. Primeramente, antes de realizar las pruebas pertinentes en los problemas, se han realizado una serie de experimentos para la calibración del algoritmo determinando la mejor opción para los parámetros de diseño. Tras la calibración se han resuelto 4 baterías de problemas, 3 para el VRPTW y 1 para el problema de Steiner. Con el objetivo de poder realizar una comparación de la bondad del algoritmo, dichas baterías han sido resueltas adicionalmente, usando un algoritmo genético simple. Después de llevar a cabo las pruebas se obtiene que el algoritmo NEMO Search resuelve mucho más rápido los problemas del VRPTW, pero la calidad de las soluciones es muy pobre en comparación con las obtenidas utilizando el algoritmo genético simple, en cambio para el problema de Steiner se obtienen resultados similares tanto en tiempo de computación como en la calidad de las soluciones, por lo que se puede concluir que el algoritmo NEMO Search puede ser una buena opción para la resolución del problema de Steiner y en el caso del VRPTW es capaz de encontrar una solución muy rápido pero sin que la misma sea de mucha calidad.

Abstract

In this Final Degree Project, the aim is to develop and calibrate an evolutionary algorithm, the NEMO Search algorithm, for solving optimization problems. This algorithm is inspired by the behavior of clownfish. The problems that have been chosen for the algorithm test are the VRPTW and the Steiner problem. First, before performing the relevant tests on the problems, a series of experiments have been carried out for the calibration of the algorithm determining the best option for the design parameters. After calibration, 4 problem batteries have been solved, 3 for the VRPTW and 1 for the Steiner problem. In order to be able to make a comparison of the goodness of the algorithm, said batteries have been additionally solved, using a simple genetic algorithm. After carrying out the tests it is obtained that the NEMO Search algorithm solves the problems of the VRPTW much faster, but the quality of the solutions is very poor compared to those obtained using the simple genetic algorithm, instead for the Steiner problem Similar results are obtained both in computing time and in the quality of the solutions, so it can be concluded that the NEMO Search algorithm can be a good option for the resolution of the Steiner problem and in the case of VRPTW it is able to find a very fast solution but without it being of high quality.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN Y OBJETIVOS DEL TRABAJO

No es el más fuerte de las especies el que sobrevive, tampoco es el más inteligente el que sobrevive. Es aquel que es más adaptable al cambio.

- Charles Darwin -

En el año 1859, el naturalista Charles Darwin (1809-1882) publicó su obra más famosa, "*El origen de las especies"*, en ella recoge sus investigaciones realizadas en sus viajes a las islas Galápagos. Darwin observó que dependiendo de la isla en la que se encontrase, distintos individuos de una misma especie de pinzones presentaban diferencias en sus picos, estos picos habían ido cambiando debido a las condiciones de su entorno. Darwin concluyo que los pinzones que mejor se habían adaptado a su entorno eran los que habían prosperado y habían transmitido sus características diferenciantes a su descendencia, y con el tiempo situándose como los únicos individuos de la especie en cada isla respectivamente.

Inspirado en este fundamento biológico, John Henry Holland desarrolló los algoritmos genéticos, lo que fue el inicio de los algoritmos bio-inspirados para la resolución de problemas de optimización.

En los algoritmos genéticos se parte de una población de individuos, al igual que sucede en la naturaleza, este conjunto de individuos representa posibles soluciones del problema, la población es sometida a una serie de pruebas para comprobar su grado de adaptación al medio, es decir, se evalúan las soluciones para medir la bondad de estas. Los individuos que mejor estén adaptados tendrán una mayor probabilidad de reproducirse transmitiendo sus cualidades a su descendencia, descendencia que podrá sufrir mutaciones y cambiar su estado. Todo este proceso se repite iterativamente hasta que se alcanza una solución óptima.

Desde entonces son muchos los algoritmos bio-inspirados, por ejemplo, uno de los más conocidos es el algoritmo de la colonia de hormigas, algoritmo que se basa en el comportamiento de las hormigas para encontrar comida, a la misma escala se encuentra los algoritmos PSO, algoritmos de optimización por enjambre de partículas, que toma el comportamiento de las abejas en la búsqueda de la región con mayor densidad de flores. No solo en los insectos se encuentra inspiración, el comportamiento de los pájaros Cuco con su particular modo de reproducción en el que ocupan los nidos de otras especies de aves con sus huevos para que los polluelos sean alimentados por estas, también ha terminado inspirando un algoritmo.

Este trabajo se centra en un nuevo algoritmo bio-inspirado, la NEMO Search, inspirado en el comportamiento de los peces payaso, el cual difiere de lo mostrado en la película de Disney "Buscando a Nemo". Los peces payaso, a diferencia de lo mostrado en la ficción, no crían a su descendencia, al salir del huevo en la anémona los hijos son dispersados por las corrientes marinas hasta que finalmente llegan a otra anémona en la que son adoptados por otra pareja de peces payaso. Pero no es solo este aspecto en el que se basa en el algoritmo, en cada anémona existe una pareja dominante que hace de padre y madre, en el caso de que se produzca el fallecimiento de la madre, el padre cambiará de sexo y pasará a ser una hembra y el hijo más fuerte a ser el nuevo padre, la adaptación de este comportamiento al algoritmo se describirá a lo largo del capítulo 3.

A lo largo del trabajo se presentará el estado del arte de la resolución de problemas de optimización y se mostrará el desarrollo y calibrado del algoritmo juntamente con las pruebas realizadas para determinar la bondad y utilidad de este.

1.1 Objetivos

Se trata de un trabajo fundamentalmente metodológico, en el cual se pretende desarrollar y calibrar el algoritmo NEMO Search para la resolución de problemas de optimización y compararlo con otros algoritmos tradicionales, para comprobar la bondad de este nuevo algoritmo. Concretamente los problemas que se usarán para la prueba del algoritmo son el problema de Steiner y el VRPTW (*Vehicle Routing Problem with Time Windows*).

2 ESTADO DEL ARTE

Un algoritmo es un conjunto ordenado de operaciones sistemáticas que permiten hacer un cálculo y hallar la solución a todo tipo de problemas. Los algoritmos han estado presentes desde la antigüedad, cuando alrededor del 300 a. C. Euclides publicase su tratado *Elementos* en el que se recoge el algoritmo de Euclides, un método robusto para calcular el máximo común divisor. En el campo de la computación habría que esperar a Ada Lovalece, adelantada a su tiempo creo el que se considera el primer programa de ordenador con un algoritmo codificado recogido en *Notas*. El campo de los algoritmos es amplio y diverso, el presente capítulo se centrará en presentar el estado del arte de los algoritmos diseñados para la resolución de problemas de optimización.

2.1 Tipos de algoritmos

El campo de los algoritmos es amplio y diverso, de los cuales existen multitud de tipos, en este capítulo se van a tratar los siguientes tipos.

2.1.1 Algoritmos exactos

Los algoritmos exactos o de búsqueda exhaustiva se crean con el objetivo de resolver problemas, que valga la redundancia, requieran respuestas exactas, como podría ser encontrar el número de caminos totales entre dos puntos o encontrar todos los números primos menores que un determinado valor. Para resolver estos problemas es necesario recorrer todos los elementos posibles, esta casuística da lugar a que estos algoritmos arrojen pobres resultados y sean poco prácticos para conjuntos grandes.

Destacan la vuelta atrás o *back tracking* y su opuesto la criba, mientras que en el backtracking a cada iteración del proceso se trata de encontrar otra solución valida y en caso de que no sea así, retroceder en la búsqueda. Por otro lado, el planteamiento de la criba consiste en la eliminación de las soluciones que no sean válidas, este método fue originalmente desarrollado por Eratóstones para encontrar todos los números primos menores que un número natural dado *n*.

2.1.2 Algoritmos heurísticos

Los algoritmos heurísticos se utilizan para resolver problemas en los cuales el uso de algoritmos exactos es imposible, debido a que el coste computacional es demasiado alto para afrontar la búsqueda de la solución.

La heurística es el arte de inventar, es un término que viene de la antigua Grecia, pero fue popularizado por George Polya en su libro "How to solve it", libro en el cuál explica el método heurístico desde el punto de vista de las matemáticas. En el caso de la informática los algoritmos heurísticos son aquellos en los que se llega a una solución mediante prueba y error y no de forma directa, esta definición abarca a muchos posibles métodos distintos entre los más reconocibles se encuentran:

- Métodos de Reducción, este método consiste en restringir el espacio de soluciones, obligando a que estas cumplan una serie de propiedades observadas en la mayoría de las soluciones buenas.
- Métodos Constructivos, son métodos deterministas en los que en cada iteración se construye una solución factible del problema.
- Métodos de Búsqueda, en estos métodos se parte de una solución del problema y se busca mejorarla paso a paso hasta que no se encuentra una solución mejor para el problema.
- Métodos de Descomposición, se basa en la descomposición del problema en múltiples subproblemas que son más simples de resolver individualmente, para luego una vez resueltos todos los subproblemas

unir las soluciones en la solución del problema general.

El aspecto negativo de los algoritmos heurísticos es que no tienen por qué llegar a una solución óptima, pueden quedar atrapados en un máximo local y tomar esa solución como la mejor, para resolver este problema se crean los algoritmos metaheurísticos los cuales se describirán a continuación.

2.1.3 Algoritmos metaheurísticos

Los algoritmos metaheurísticos definidos por F. Glover son procesos iterativos que guían a una heurística para explorar adecuadamente el espacio de búsqueda atendiendo a distintos conceptos.

Según los conceptos utilizados para la guía de la metaheurística se pueden clasificar en:

- Trayectoria: la heurística subordinada se trata de una heurística de búsqueda local que sigue una trayectoria en el espacio de búsqueda.
- Poblaciones: durante la ejecución del algoritmo se trabaja con distintas poblaciones que evolucionan en paralelo.
- Uso de memoria: dependiendo de si se basan exclusivamente en el estado anterior, sin memoria, o si al contrario se utilizan estructuras de memoria para recordar la historia pasada, con memoria.
- Inspiración: se clasifican en función de si están inspirados en la naturaleza o en cambio se basan puramente en propiedades matemáticas.

Los algoritmos metaheurísticos no garantizan que se encuentre una solución óptima del proceso, pero sí que se obtendrá una solución buena lo que los hace útiles para procesos en los que una solución buena sea suficiente o procesos para los cuáles no exista un algoritmo específico creado para resolverlos o los que existan sean muy costosos e ineficientes.

2.2 Taxonomía

En este apartado se va a realizar una taxionomía de los algoritmos más utilizados para la resolución de problemas de optimización, se dividirán en dos grupos, algoritmos de búsqueda y algoritmos evolutivos.

2.2.1 Algoritmos de búsqueda

Los algoritmos de búsqueda son algoritmos que tratan de encontrar un elemento que cumpla una serie de propiedades, dentro de este amplio grupo destacan:

2.2.1.1 Local Search

Los algoritmos de búsqueda local son aquellos que dados un conjunto de estados posibles y una función que los evalue, parte de un estado inicial y tratan de mejorar la calidad de dicho estado a cada iteración del proceso. Para esta mejora se usan diversos procesos como puede ser la escalada en el cual se avanza al siguiente valor siempre que el valor del siguiente sea mayor que el actual y en caso contrario se permanece en el estado actual. Este proceso puede llevar a estancamientos en mínimos locales por lo que no es tan bueno como los siguientes métodos que se discutirán posteriormente.

2.2.1.2 Taboo Search

Este algoritmo fue creado por F. Glover y la idea general de este método es marcar como prohibidos los movimientos previamente realizados, es decir una vez que se ha hallado una solución en un determinado lugar del espacio de búsqueda, no se puede volver a recorrer el trayecto que ha llevado a esa solución.

De esta forma el algoritmo evita que se empobrezcan las soluciones halladas, estos movimientos previos se almacenarán en memoria en forma de una lista tabú en la que se almacena la información de las soluciones, conforme avance el desarrollo del algoritmo la memoria se adaptará para almacenar información vital para la guía del proceso tras haber utilizado la lista tabú varias veces, esta información suele estar compuesta de ratios

como podrían ser la cantidad de veces que un atributo cambia en las soluciones de una trayectoria. Esto hace que la "Taboo Search" pueda adaptarse dinámicamente a las soluciones encontradas.

2.2.1.3 Simulated annealing

La naturaleza es una fuente inabarcable de inspiración y no solo para los algoritmos evolutivos, este algoritmo encuentra su inspiración en el proceso de recocido del acero y cerámicas. El recocido consiste en el calentamiento del material para posteriormente enfriarlo lentamente, provocando así una variación de las propiedades físicas del mismo. La adición de calor causa un aumento de energía de los átomos que componen el material resultando en una reconfiguración del estado de los átomos en otro de menor energía.

De forma independiente este método fue desarrollado por Scott Kirpatrick, C. Daniel Gelatt y Mario P. Vechhi en 1983, en el cual, haciendo un símil con los estados de los átomos, en cada iteración del algoritmo se considerará mantenerse en el estado actual o pasar a un nuevo estado vecino cuya energía sea menor que la actual. Esta transición se modela mediante una función que depende de la diferencia de energía entre ambos estados y una variable denominada T, de temperatura por analogía. Podrá existir la posibilidad de cambiar a un estado de mayor energía, pero esta probabilidad disminuirá conforme disminuya la temperatura, descenso que se adapta en función del problema que se desea resolver hasta que cuando esta sea nula solo se podrá transicionar a un estado de menor energía, una vez que no se encuentren estados de menor energía terminará el algoritmo.

2.2.1.4 VNS

Este algoritmo propuesto Mladenovic, Hansen en 1997 se basa en los siguientes puntos:

- Un mínimo local con respecto a una vecindad no tiene por qué ser un mínimo para otra vecindad.
- Un mínimo global es un mínimo local con respecto a todas las vecindades.
- Para muchos problemas los mínimos con respecto a una o varias vecindades se encuentran relativamente cerca entre sí.

Es en la simpleza de estos preceptos donde reside la fortaleza del método, ya que se trata de un algoritmo muy simple de implementar pero que proporciona buenas soluciones, y es gracias a la simpleza de sus ideas que es fácil de añadir variaciones o incluso de usarlo juntamente con otros algoritmos.

2.2.2 Algoritmos evolutivos

Los algoritmos evolutivos son aquellos que se basan en postulados de la naturaleza consiguen resolver problemas con espacios de búsqueda extensos y no lineales para los que los otros métodos existentes no son capaces de llegar a una solución dentro de un tiempo razonable.

Destacan:

2.2.2.1 Algoritmos genéticos

Los algoritmos genéticos establecidos por Holland y desarrollados por otros autores como Goldberg, Davis, Michalewicz y Reeves, están basados en la selección natural.

El proceso que siguen es el siguiente:

- Se parte de una población inicial de individuos, cada individuo representa una solución factible del problema codificada.
- La población es evaluada mediante una función de evaluación que determinará su *fitness*, es decir lo buena o mala que es dicha solución.
- Una vez la población se ha evaluado se pasa al proceso de reproducción y mutación, en el cual los individuos mejor adaptados, es decir los individuos cuyo *fitness* es mayor, tendrán más posibilidades de reproducirse pudiendo así transmitir sus cualidades a la siguiente generación. El proceso de mutación se lleva a cabo de forma aleatoria independientemente del *fitness* del individuo.
- En el caso de que se llegue a converger, el algoritmo finalizará, en caso contrario seguirá el proceso,

conforme pasen las generaciones la población tenderá a mejorar hasta que la media de la población se encuentre al mismo nivel que el mejor individuo, esta tendencia se puede observar en la siguiente figura:

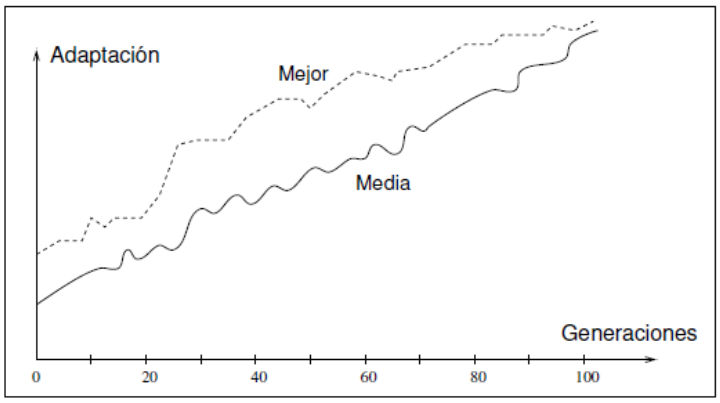


Figura 1: Evolución de la adaptación con respecto a las generaciones. (Fuente: http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf)

Todo este proceso se puede expresar en forma de pseudocódigo:

```
BEGIN /* Algoritmo Genético simple */
      Generar una población inicial.
      Computar la función de evaluación de cada individuo.
      WHILE NOT Fin DO
      BEGIN /* Producir nueva generación */
          FOR Tamaño población/2 DO
          BEGIN /* Ciclo Reproductivo */
            Seleccionar dos individuos de la anterior generación para el
            para el cruce (probabilidad proporcional al fitness).
            Cruzar con cierta probabilidad los individuos seleccionados
            obteniendo dos descendientes.
            Mutar los descendientes con cierta probabilidad.
            Computar la función de evaluación de los descendientes
            mutados.
            Insertar los dos descendientes mutados en la nueva
            generación.
          END
          IF Convergencia THEN
             Fin:= TRUE
      END
END
```

Figura 2: Pseudocódigo algoritmo genético (Fuente: http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf)

Para las operaciones de cruce y mutación existen operadores que actúan directamente sobre los individuos codificados, dicha codificación se elige libremente, aunque la teoría original codifica los genes como unos y ceros. El operador más simple se trata del operador de cruce en un punto el cual se observa en la siguiente imagen:

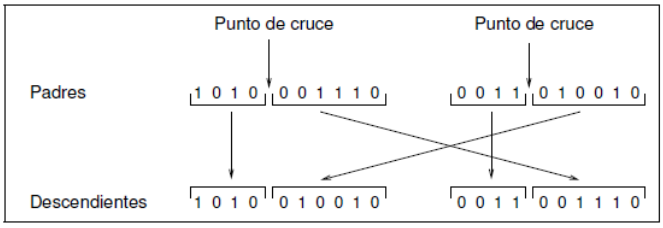


Figura 3: Operador de cruce en un punto. (Fuente: http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf)

Dicho operador es aplicado a los individuos seleccionados con una cierta probabilidad, la operación consiste en dividir a los individuos padres por un punto lo cual producirá cuatro ristras de genes, dichas divisiones se forman los descendientes. Existen diversos operadores de este tipo los cuáles se crean añadiendo más puntos de corte, aunque la adición de muchos puntos de corte no produce una mejora sustancial, De Jong (1975) determinó que la utilización de más de dos puntos de corte no presentaba una mejora sustancial en el comportamiento del algoritmo. El operador de cruce en dos puntos se puede ver en la siguiente figura:

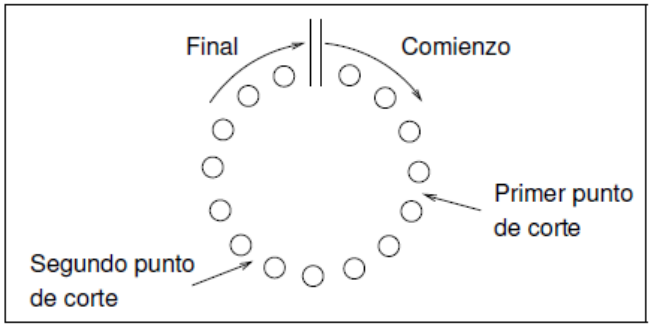


Figura 4: Operador de cruce en dos puntos (Fuente: http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf)

La mutación en cambio se aplica individualmente, este operador altera un gen del individuo, pero con una pequeña probabilidad, autores como Davis (1985) o Schaffer (1989) recalcaron la importancia de este operador ya que provoca que la calidad de las soluciones sea mayor que en el caso de que solo existiese un proceso que únicamente utilizase los operadores de cruce. El operador de mutación se observa en la siguiente figura:

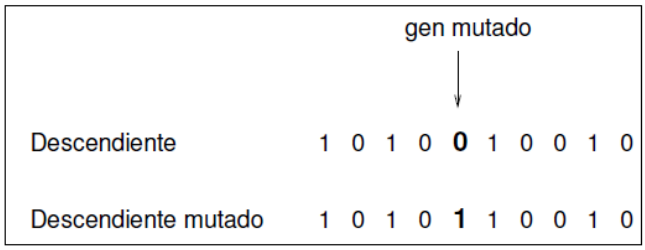


Figura 5: Operador de mutación (Fuente: http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf)

2.2.2.2 ACO

La optimización por colonia de hormigas fue propuesta por Marco Dorigo en 1992, es un algoritmo ideado para la solución de problemas de búsquedas de mejores caminos o rutas, su idea está basada en el comportamiento de las hormigas. Las hormigas deambulan de forma aleatoria hasta que consiguen encontrar comida, tras esto la hormiga regresa al hormiguero dejando en el camino un rastro de feromonas, cuando otras hormigas detecten este rastro es probable que dejen de deambular aleatoriamente y comiencen a seguir el rastro encontrado, una vez estas hormigas recorran el camino lo reforzarán a su vez con sus propias feromonas haciendo más probable que otras hormigas detecten el rastro. Dichas hormonas desaparecen con el tiempo lo que provoca que las rutas más largas al objetivo tengan menos atracción que las rutas más cortas ya que estas últimas al ser recorridas más

frecuentemente se ven más reforzadas. Gracias a esta desaparición se evita la convergencia en un óptimo local, sino sucediese los primeros caminos atraerían en demasía al resto de hormigas por lo que se reducirían las posibles soluciones.

Este proceso se puede observar en la siguiente figura:

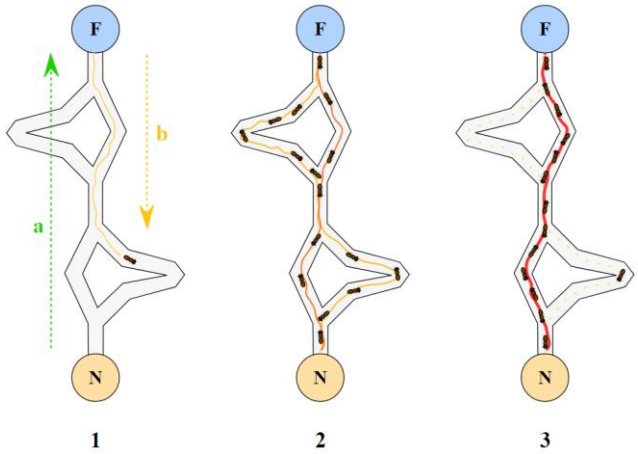


Figura 6: ACO (Fuente: Wikipedia)

1. Primero una hormiga encuentra un camino hacia la comida de forma aleatoria y vuelve al hormiguero dejando las feromonas a su paso.
2. Las hormigas recorren varios caminos para el mismo objetivo, pero el camino más corto es el que se verá más reforzado.
3. Finalmente, el camino más corto prevalece mientras que el resto de los caminos desaparecen gradualmente.

La retroalimentación positiva y negativa por parte del resto de hormigas evita que se produzcan bloqueos que se podrían producir en el caso de que existiese la misma cantidad de hormonas en todos los caminos. La mayor fortaleza de este algoritmo reside en la posibilidad de adaptarse en tiempo real a cambios en el grafo que se esté resolviendo.

2.2.2.3 PSO

La "Particle Swarm Optimization" fue propuesta por Russ C. Eberhart y James Kennedy, y se basa principalmente en los comportamientos sociales de animales como las aves, los peces y las abejas al desplazarse conjuntamente como medio para sobrevivir. El proceso parte inicialmente de una población compuesta por varias partículas, el tamaño de esta población tendrá una gran influencia en el coste computacional ya que a más partículas mayor número de operaciones y por lo tanto mayor coste, cada una de estas partículas representaría a un animal del grupo que deambulan por el espacio de búsqueda. El movimiento de las partículas estará afectado por el del resto, cada partícula se define por su posición, su velocidad, y su *fitness* (al igual que en los algoritmos genéticos) y, además, cada partícula tiene un registro de la mejor posición encontrada por sí misma y de la mejor posición encontrada por el enjambre de forma global. La posición de cada partícula variará en función de la velocidad, la velocidad es regulada por la inercia de la partícula, lo que hace que sean reacias a cambiar de dirección, y por la aceleración provocada por la fuerza de atracción ejercida por la mejor localización encontrada por el enjambre y por la mejor localización encontrada por sí misma de forma que cuanto mayor sea la distancia a ellas mayor será la fuerza de atracción que sufra la partícula, gráficamente se puede ver en la siguiente figura:

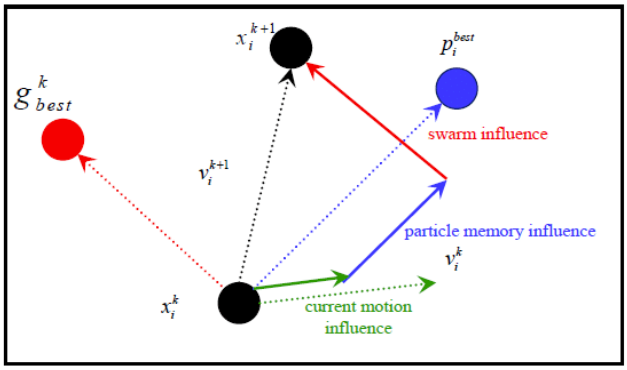


Figura 7: Actualización de la posición de una partícula. (Fuente: History Matching Using Hybrid Parameterisation and Optimisation Methods, Al-Shamma)

Las partículas continuarán su desplazamiento por el entorno hasta que finalmente el algoritmo converja. La principal fortaleza de este método recae en la simpleza de las operaciones realizadas por las partículas solo utilizan operaciones vectoriales básicas, por lo que este método es especialmente adecuado para problemas en los que se tenga una función de evaluación muy costosa computacionalmente de evaluar. Por otra parte, al igual que en el caso de los algoritmos genéticos, el resultado obtenido puede que no sea el valor óptimo, lo que se gana en términos de adaptabilidad y simplicidad de las operaciones se pierde en eficiencia en comparación con un determinado problema en comparación con un método específico.

2.2.2.4 Cuckoo Search

El algoritmo Cuckoo Search propuesto por Yang y Deb (2009) se inspira en el comportamiento reproductivo de los pájaros cuco, los cucos son parásitos de puesta. Los parásitos de puesta son parásitos que se valen de otros animales, ya sea de la misma o distinta especie, para la cría de su descendencia. Para el desarrollo del algoritmo se siguen los siguientes principios:

1. Cada cuco pone un solo huevo y los deposita en un nido elegido aleatoriamente.
2. El nido con mejores huevos permanecerá para la siguiente generación.
3. Los nidos disponibles son fijos y la probabilidad de que el anfitrión del nido descubra el huevo dejado por el cuco tiene una probabilidad $p_a \in [0,1]$

Cada cuco pondrá los huevos en torno a un área denominada ELR (Egg Laying Radius), la representación de dicho espacio se observa en la siguiente figura:

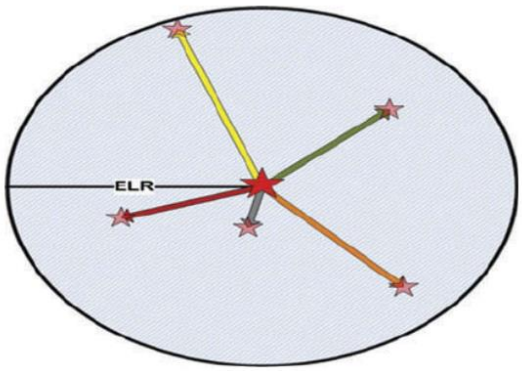


Figura 8: ELR (Fuente: Cuckoo Search Algorithm for Optimization Problems—A Literature Review and its Applications, Azizah Binti Mohamad)

Una vez se han puesto los huevos, el ave dueña del nido detectará los huevos que difieran en demasía con los suyos y los tirará del nido, esto adaptado al algoritmo significa que las soluciones nuevas que fuesen peores que las ya existentes se desechan, el resto de huevos existirán hasta que el huevo del cuco eclosione y serán

eliminados por este, una vez los cucos sean adultos emigrarán a nuevas áreas más ventajosas, es decir los cucos buscarán zonas en el que la calidad de las soluciones sean mayores, el algoritmo convergerá una vez que todos los cucos hayan emigrado a un área y los huevos sean similares. La migración se observa en la siguiente figura:

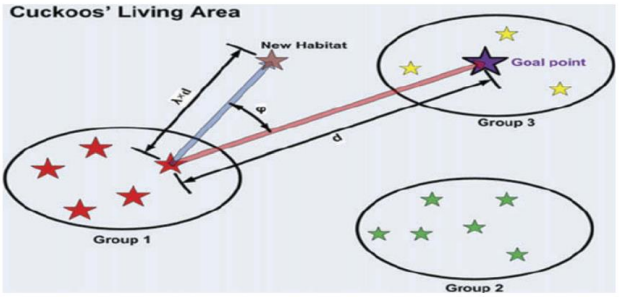


Figura 9: Migración de los cucos (Fuente: Cuckoo Search Algorithm for Optimization Problems—A Literature Review and its Applications, Azizah Binti Mohamad)

2.2.2.5 Algoritmos Meméticos

Los algoritmos meméticos fueron creados por P. Moscato para sobrepasar los límites de los algoritmos evolutivos, los cuales se ven limitados por el teorema NFL (No Free Lunch) el cual estipula que el rendimiento un algoritmo de búsqueda es directamente proporcional a la cantidad del conocimiento del problema que incorporen, para superar esto se combinan algoritmos de búsqueda local con algoritmos evolutivos. Los algoritmos meméticos se basan en el concepto de meme introducido por Richard Dawkins:

"Ejemplos de memes son melodías, ideas, frases hechas, modas en la vestimenta, formas de hacer vasijas, o de construir bóvedas. Del mismo modo que los genes se propagan en el acervo genético a través de gametos, los "memes" se propagan en el acervo memético saltando de cerebro a cerebro en un proceso que, en un amplio sentido, puede denominarse imitación"

A raíz de esto en todo algoritmo memético se parte de una población de agentes los cuáles pasarán por distintas etapas:

- Mejora individual: En esta etapa los agentes buscarán mejorar mediante algoritmos de búsqueda local como los descritos anteriormente en la sección Algoritmos de búsqueda.
- Cooperación: Etapa en la que se crean nuevos agentes mediante recombinación y mutación.
    - Recombinación: Se crean nuevos agentes mediante operaciones de cruce como en los algoritmos genéticos.
    - Mutación: Al igual que en los algoritmos genéticos se introduce este operador permite alterar los agentes recombinados ampliando así el espacio de búsqueda.
- Competición: Finalmente tras haber buscado una mejora tanto de forma individual como mediante la cooperación de los agentes solo los más aptos permanecerán para la siguiente generación del algoritmo.

De esta forma se consigue aprovechar las fortalezas de cada tipo de algoritmo, con las búsquedas locales conseguimos mejorar a los agentes en su entorno de forma más efectiva que si solo empleásemos un algoritmo evolutivo y a su vez, se consigue una mejora global gracias a la utilización de los algoritmos evolutivos que desempeñan mejor esta función.

2.3. Aplicaciones

El campo de la optimización tiene aplicación directa en muchos sectores y los algoritmos que se han presentado pueden resultar de gran ayuda. Entre las diversas aplicaciones se destacan:

- Transporte: los problemas de rutado son mundialmente conocidos, la búsqueda de la ruta más corta

supone a la empresa de transporte un gran ahorro, entre las últimas novedades se han desarrollado aplicaciones para el rutado en tiempo de real de vehículos mediante el uso de ACO y algoritmos genéticos, descrito en el artículo " Using the metaheuristic methods for real-time optimisation of dynamic school bus routing problem and an application" de Tuncay Yigit.

- Comunicaciones: la elección de las rutas con menor coste, al igual que en el transporte es fundamental en este campo, en el último estudio de Jing Li, se describe el uso del algoritmo ACO aplicado a la comunicación de satélites, dicho estudio ha conseguido mediante el uso del algoritmo ACO aumentar el ancho de banda de transmisión de datos y reducir los recursos de mando necesarios.
- Software: en el diseño de software una de las aplicaciones de los algoritmos es su uso para detectar "code smell" que es un indicador de la bondad del código fuente desarrollado, esto se ha descrito en el artículo "Hybrid particle swarm optimisation with mutation for code smell detection" publicado por G. Saranya.
- Control: el diseño robusto de controladores es otro campo que se presta al uso de estas técnicas, en el artículo "A memetic algorithm for power system damping controllers design" publicado por Wesley Peres, en el que se ha utilizado un algoritmo memético para el diseño de un controlador con el fin de amortiguar las oscilaciones de un sistema de potencia, también podemos encontrar otros estudios en el que se utiliza el algoritmo Cuckoo Search con el mismo fin publicado por Anil Swarnkar.
- Diseño electrónico: los algoritmos también se pueden utilizar para calcular los valores de los distintos componentes de un circuito, en el estudio "Cuckoo search design optimization and analysis of matching circuits composed of CRLH transmission cells" se utiliza el algoritmo Cuckoo Search para calcular los valores para el ajuste de impedancias.
- Energía: las principales aplicaciones desarrolladas en este campo están orientadas al control y planificación de distintas fuentes de energía, por ejemplo en el estudio "A discrete particle swarm optimisation algorithm to operate distributed energy generation networks efficiently" publicado por Pablo Cortés se describe el uso del algoritmo PSO para la optimización de la distribución de energía desde distintas fuentes a un solo cliente, o en otros estudios como "A distributed combinatorial optimisation heuristic for the scheduling of energy resources represented by self-interested agents" por Christian Hinrichs, realizan la programación de los recursos de energía a desarrollar.

2.4. Librerías existentes

Entre las librerías y framework existentes para la programación de algoritmos evolutivos destacan:

- JGAP (Java Genetic Algorithm Package) se trata de un framework diseñado para Java el cual ofrece clases e interfaces para representar genes, cromosomas, individuos, la población, operadores de ajuste y operadores genéticos, está orientada principalmente a algoritmos genéticos y es ahí donde reside su principal limitación, ya que para la implementación de otros algoritmos no ofrece tantas soluciones como otros frameworks existentes para otros lenguajes.

- EO (Evolving objects) este framework esta desarrollado para C++ y está formado principalmente de componentes que se pueden insertar en la resolución del problema según se requiera, de forma que la implementación de un algoritmo para un problema clásico para el que ya existe código definido la inclusión de los componentes de EO es sencilla e intuitiva. Pero para la resolución de algunos problemas en los que los componentes no contengan las suficientes funcionalidades será necesario crear la clase que describa el comportamiento de los individuos.

Figura 10: Logo EO

- DEAP (Distributed Evolutionary Algorithms in Python), este será el framework escogido para la implementación del algoritmo Nemo Search, está desarrollado para Python y entre sus principales ventajas destacan:
    - No hay limitación de tipos, el framework incorpora un módulo *creator* que permite construir el tipo de los individuos según convenga para el problema sin estar limitado por los tipos predefinidos del framework
    - Permite crear clases que contengan los operadores necesarios para el funcionamiento del algoritmo como por ejemplo los operadores de cruce y mutación, que ya están definidos dentro de funciones de la librería, o las funciones para la inicialización de los individuos que se llamarán automáticamente cada vez que se cree un individuo nuevo.
    - Para la construcción de algoritmos DEAP ofrece una serie de módulos que pueden insertarse en el que se esté diseñando, además también ofrece la posibilidad de usar cuatro algoritmos ya implementados dentro de la librería.

Figura 11: Logo DEAP

3 NEMO SEARCH

En el presente capítulo se procede a explicar el algoritmo NEMO Search, para ello se expondrá la inspiración natural del mismo para posteriormente pasar a la explicación del pseudocódigo y el método seguido por el algoritmo para encontrar la solución al problema.

3.1 Inspiración natural

La inspiración de este algoritmo surgió de la investigación dirigida por Laura Casas sobre los mecanismos del cambio de sexo en los peces payasos, más conocidos por la película de Disney "Buscando a NEMO", un trabajo conjunto del Red Sea Research Center de la King Abdullah University of Science and Technology (Arabia Saudí) y el Instituto de Investigaciones Marinas (CSIC).

El equipo estudió el genoma de esta especie que se agrupa en familias de dos, tres o hasta siete individuos, dependiendo del tamaño de la anémona en la que habiten. Si son dos siempre serán un macho y una hembra, siendo la hembra más grande y dominante que el macho, si son tres o más, la familia estará compuesta por una hembra y macho adultos y una serie de peces juveniles masculinos que en realidad albergan gónadas de ambos sexos. Estos peces más jóvenes no son hijos de la pareja existente en la anémona, una vez eclosionan los huevos puestos por la madre, los hijos son arrastrados y dispersados por la corriente para evitar la endogamia y el deterioro de la especie.

Los jóvenes peces van creciendo lentamente, siempre por debajo de la pareja dominante, manteniendo las gónadas de ambos sexos. En el caso de que muera la hembra, el macho pasará a perder las gónadas masculinas a cambio de convertirse en hembra, mientras que el mejor preparado de los jóvenes pasará a ser el nuevo macho de la anémona, aunque en el caso de que solo exista una pareja en la anémona el macho quedará viudo hasta que por aleatoriedad apareciese un nuevo macho en la anémona y pasaría a ser la hembra.

Este es el fundamento natural del algoritmo NEMO Search inspirado por este comportamiento de los peces payasos, en el algoritmo se hace un símil en el que los peces representan las soluciones del problema.

3.2 Algoritmo

Una vez que se ha dado el contexto desde donde parte el algoritmo, se procederá a explicar las distintas partes de este.

3.2.1 Generación de anémonas

Justo al comenzar el algoritmo se generarán las anémonas que contendrán a los peces, las anémonas actuarán como contenedores, cada anémona inicialmente contendrá entre dos individuos siendo estos la pareja principal de la anémona un macho y una hembra, y siete individuos, una pareja junto con los cadetes, machos jóvenes. Esta generación se realizará aleatoriamente creando tantas anémonas como indique el parámetro *Na*.

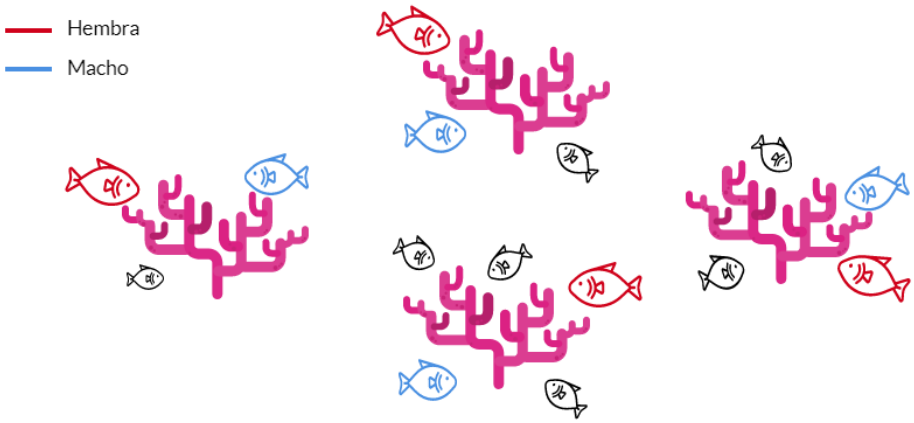


Figura 12: Generación de anémonas

3.2.2 Evaluación

La función de evaluación es una parte vital del algoritmo ya que esta función será la que permita medir la calidad de las soluciones obtenidas por el algoritmo y por ende la bondad de este, la función de evaluación cambia en función del problema que se desea resolver. Las funciones de evaluación que se han realizado para el VRPTW y el problema de Steiner se describirán en las subsecciones 4.1.1 y 4.1.2 del capítulo 4 respectivamente.

3.2.3 Bucle principal

Tras la evaluación se comienza el bucle de las generaciones y lo primero es realizar la elección entre reproducción o excursión, esta elección se realizará de forma aleatoria.

3.2.3.1 Reproducción

En el caso de que se elija reproducirse el proceso constará de varias partes:

1. Se elige aleatoriamente la anémona desde la que se generará la descendencia a partir de la pareja existente.
2. Se genera la descendencia, para ello los padres serán la pareja de la anémona escogida, y se aplicarán los operadores de cruce y mutación descritos anteriormente en la subsección de Algoritmos genéticos en el capítulo dos.
3. Tras la generación de la descendencia, esta será repartida por el resto de las anémonas existentes en la población.

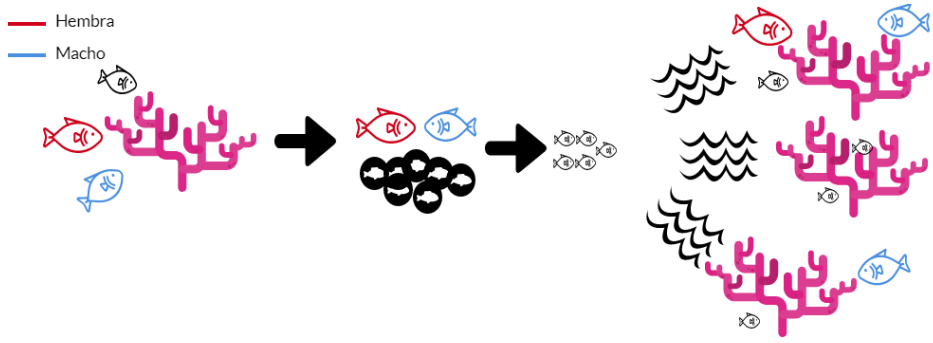


Figura 13: Reproducción

3.2.3.2 Excursión

En este caso el macho y la hembra que forman la pareja de la anémona realizarán excursiones fuera de la anemona, la excursión será el momento de aplicar los mecanismos de búsqueda local, este proceso se modelará según una exponencial decreciente de la forma:

$$y = x^{-a}$$

Dependiendo del valor de esta las funciones que se aplicarán son:

- 2-OPT dado un individuo rompe dos arcos y los reconecta creando dos nuevos, lo que se puede apreciar en la siguiente figura.

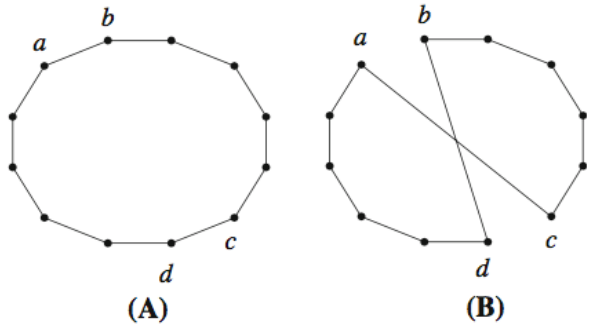


Figura 14: 2-OPT (Fuente:(Ouaarab et al 2014))

- DOUBLE-BRIDGE en esta operación se rompen cuatro arcos y se reconectan como se aprecia en la figura:

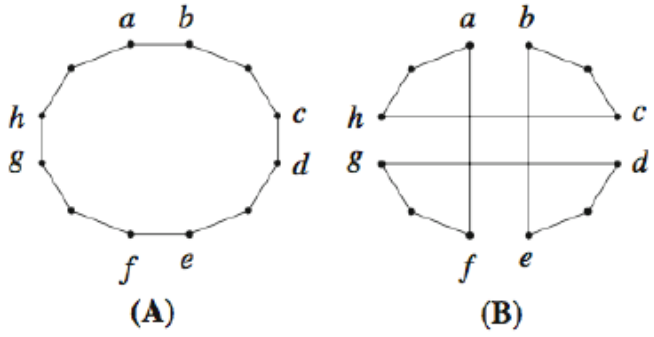


Figura 15: DOUBLE-BRIDGE (Fuente:(Ouaarab et al 2014))

- Recombinación, a partir del pez original se construyen nuevos individuos seleccionando aleatoriamente de este y se toma el mejor resultado.

Pero este proceso no está exento de riesgos ya que podría producirse la muerte de un pez, la cual se explicará en detalle en la siguiente subsección.

3.2.3.3 Muerte

La muerte es una situación que puede darse durante las excursiones del macho y la hembra, esta puede suceder por dos causas:

1. *Nmov* movimientos consecutivos sin mejora.
2. *Niter* iteraciones sin encontrar una solución mejor que la encontrada por el pez.

La muerte podrá afectar indistintamente tanto al macho como a la hembra, pero el proceso de readaptación de la anémona de procedencia del pez será distinto en función de si muere el macho o la hembra.

En el caso de que muera la hembra se distinguen dos situaciones, una en la que existan cadetes en la anémona y otra en la que solo existiese la pareja, en este último caso el macho quedará viudo hasta que llegue a la anémona algún cadete. En el caso de que existan cadetes previamente, el macho perderá las gónadas masculinas y pasará

a ser una hembra y el mejor de los cadetes tomará el puesto de macho. A su vez si el que muere es el macho este será reemplazado por uno de los cadetes. Este proceso se puede apreciar en la siguiente infografía donde los peces de color azul representan al macho de la anémona y el color rojo a la hembra.

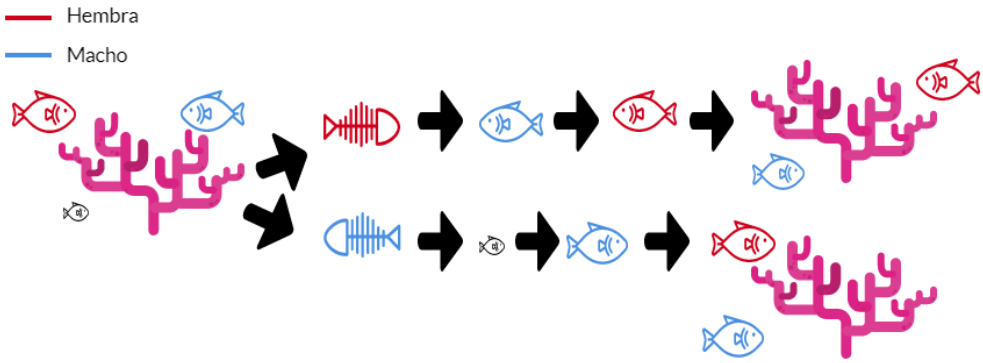


Figura 16: Muerte de un pez

3.2.4 Pseudocódigo

Una vez se han presentado todos los procesos ocurrentes en algoritmo el pseudocódigo de este es:

```
BEGIN /* NEMO Search */
      Generar anémonas.
      Computar la función de evaluación de cada individuo.
      FOR Número de generaciones DO
      BEGIN /* Nueva iteración */
          ¿Excursión o reproducción?
          IF Excursión THEN
            Elección aleatoria de anémona y pez
            Elección aleatoria de la distancia a recorrer
            Realizar desplazamientos
            IF Muerte THEN
              Transformación
          IF Reproducción THEN
            Elección aleatoria de anémona
            Generar descendencia
          Repartir descendencia
      END
END
```

Figura 17: Pseudocódigo NEMO Search

Al comenzar el algoritmo se generarán las anémonas y serán evaluadas, conforme entramos en el bucle principal se elegirá de forma aleatoria si se realiza una excursión o en cambio se opta por la reproducción, tras la realización de dichos procesos se evalúa de nuevo la población nueva resultante y se sigue con el proceso.

3.3 Parámetros de diseño

Para el diseño del algoritmo es necesario definir una serie de parámetros, muchos de ellos se han mencionado durante la explicación del algoritmo, recapitulando los parámetros a definir son:

- *Na*, número total de anémonas
- *Nmov*, número de movimientos consecutivos sin mejora.
- *Niter*, número de iteraciones sin encontrar una solución mejor a la mejor solución global.
- *Ngen*, número de iteraciones del algoritmo.

Con el fin de determinar el valor de los parámetros se han realizado una serie de experimentos que serán descritos en el capítulo 4.

4 PRUEBAS Y RESULTADOS

En este capítulo se procede a describir y presentar las distintas pruebas realizadas tanto para la calibración del algoritmo NEMO Search como para la comprobación de la bondad del algoritmo, adicionalmente, se realizará una breve descripción de los problemas a resolver.

4.1 Parámetros de diseño

4.1.1 VRPTW

El problema VRPTW consiste en encontrar una serie de rutas, que permitan a una serie de vehículos repartir a una serie de clientes de forma que se minimice el coste de las rutas. Además de las distancias de cada ruta para buscar el mínimo coste, también se deberá tener en cuenta las ventanas temporales. Estas ventanas temporales son un plazo de tiempo en el que el cliente tiene que recibir el envío, en el caso de que la llegada al cliente sea anterior al comienzo de la ventana, se deberá esperar hasta que sea el tiempo de inicio. Esta espera llevará asociado un coste, al igual que si se produce un retraso y se entrega fuera de la ventana temporal también se incurrirá en un coste adicional. Además de lo descrito también se tienen en cuenta otros factores como la capacidad de los vehículos de reparto o el número de ellos. Todas las rutas deben partir desde el almacén y acabar en este antes de que se supere el tiempo de vencimiento de la ventana temporal del almacén. Una resolución posible del problema se aprecia en la siguiente figura:

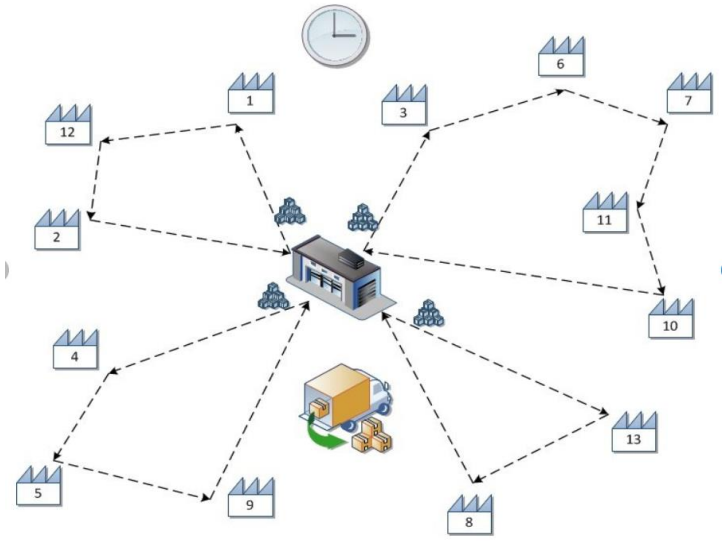


Figura 18: Resolución VRPTW (Fuente: Models and Algorithms for the Pollution-Routing Problem and Its Variations, Emrah Demir)

Para este problema la función de evaluación calculará el *fitness* como la inversa del coste total de la ruta representada por el individuo, dicho coste total incluirá el coste por distancia además del coste asociado al tiempo ya sea por retrasos o por tiempo de espera, para el cálculo del coste temporal se supone que la velocidad es de un metro por segundo para facilitar los cálculos.

4.1.2 Problema de Steiner

El problema de Steiner es un problema de rutado de grafos, en el cual se parte de un grafo con nodos conectados entre sí. Los nodos se dividen en dos grupos, nodos terminales y nodos de Steiner. El objetivo es encontrar el árbol que una a todos los nodos terminales entre sí de forma que el coste sea mínimo, un árbol dentro del grafo se define como el conjunto de nodos y los arcos que los conectan entre sí. Este problema fue propuesto en el

siglo XIX por el matemático Jacbo Steiner y su relevancia se extiende hasta nuestros días por su importancia en campos como el de las comunicaciones y rutado de redes.

En la siguiente figura se puede ver un ejemplo de un árbol de Steiner para un grafo dado, en los nodos verdes representan los nodos de Steiner y los nodos rojos los nodos terminales.

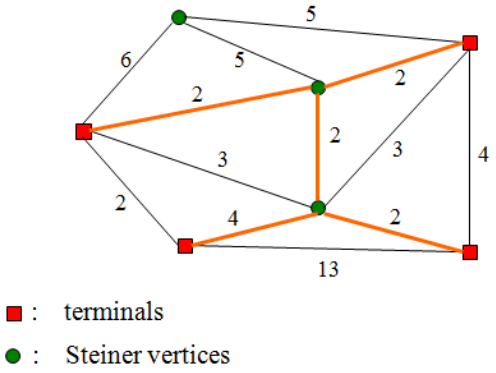


Figura 19: Árbol de Steiner(Fuente: An Approach for Single Object Detection in Images, Nileshsingh V. Thakur)

La función de evaluación en este caso es más simple que en el VRPTW, se tomará el coste original del árbol y se calculará el coste del árbol de Steiner propuesto por el individuo, el *fitness* será la diferencia de ambos.

4.2 Calibración del algoritmo

La calibración del algoritmo se lleva a cabo con el objetivo de encontrar el valor adecuado para la resolución de los problemas de los parámetros descritos en la sección 3.3 del capítulo 3. Para la calibración de cada problema se han tomado los parámetros *Na, Niter, Nmov* y *Ngen* y se han variado sus valores con el objetivo de seleccionar los valores que mejor resultados proporcionen, estos valores se han probado independientemente para el VRPTW y el problema de Steiner.

Para el VRPTW *Na* variará entre 25 y 100 anémonas con un paso de 25, *Niter* y *Nmov* variarán entre 1 y 15, con un paso de uno, iteraciones, y *Ngen* entre 10000 y 35000 generaciones con un paso de 5000 entre ellas, para el problema de Steiner *Na* variará entre 15 y 60 anémonas con un paso de 15, manteniendo el resto de los parámetros las mismas variaciones. Cada experimento se realizará 5 veces y el resultado obtenido será la media de los 5 experimentos con el fin de obtener un resultado que represente mejor a la realidad. El número de clientes escogido para la calibración del VRPTW es de 30 clientes.

En la siguiente figura se observan los resultados obtenidos para el VRPTW, en ella se representan la evolución del coste con respecto al número de iteraciones, cada color de la gráfica representa a un valor del parámetro *Na*, siendo el color rojo los resultados obtenidos para $Na = 25$, el color azul para $Na = 50$, el color negro para $Na = 75$ y el color verde para $Na = 100$. En la gráfica solamente se mostrarán los mejores resultados obtenidos.

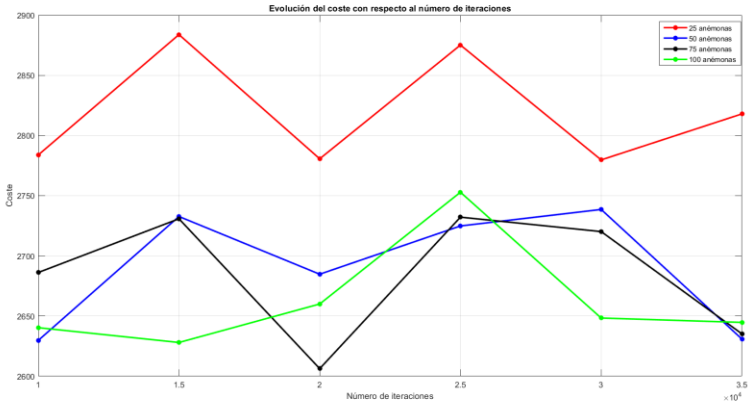


Figura 20: Evolución del coste con respecto al número de iteraciones (VRPTW)

Pero dado que existen otros factores para la elección de los parámetros también se ha calculado el tiempo de computación de cada conjunto de valores, esta comparativa se puede apreciar en la siguiente figura:

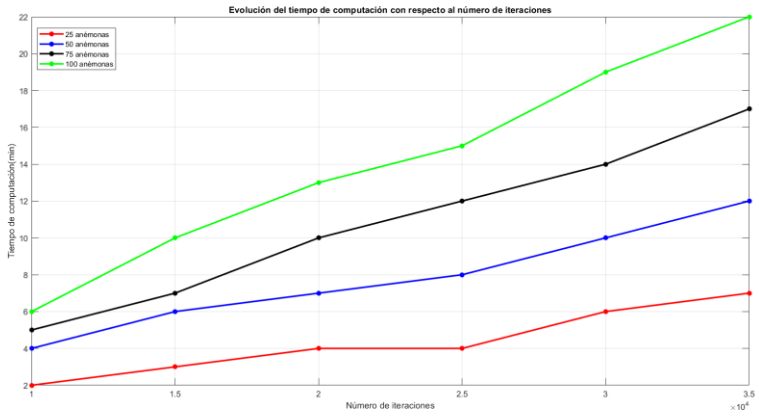


Figura 21: Tiempo de computación con respecto al número de iteraciones (VRPTW)

Tras el análisis de los resultados obtenidos, el conjunto de parámetros que presentan el mejor balance entre coste y tiempo de ejecución para el VRPTW son:

$$Na = 75$$

$$Niter = 7$$

$$Nmov = 5$$

$$Ngen = 20000$$

Este conjunto de parámetros proporciona el menor coste de todos los experimentos realizados, juntamente con un tiempo de computación de 10 minutos, el cuál es un tiempo más que aceptable para los resultados que proporciona.

Para la calibración de los parámetros para el problema de Steiner se ha realizado el mismo proceso, en la siguiente figura se presenta el coste con respecto al número de iteraciones para cada valor de *Na*, la gráfica roja corresponde a $Na = 15$, la gráfica azul a $Na = 30$, la gráfica negra a $Na = 45$ y finalmente la gráfica representa a $Na = 60$.

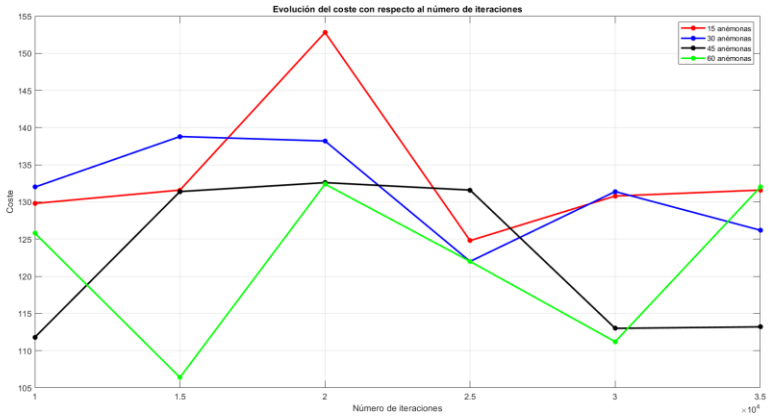


Figura 22: Evolución del coste con respecto al número de iteraciones (Steiner)

Al igual que para el VRPTW, el tiempo de computación será un factor para tener en cuenta para la elección de los parámetros, dicho tiempo de computación aparece representado en la siguiente figura:

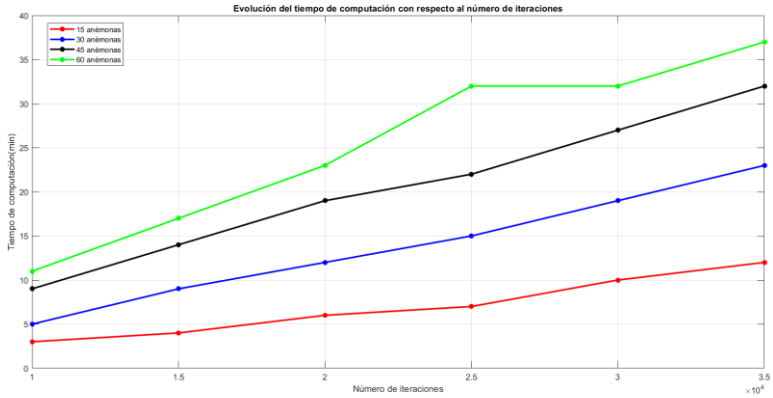


Figura 23: Tiempo de computación con respecto al número de iteraciones (Steiner)

El conjunto de parámetros elegidos para el problema de Steiner es:

$$Na = 60$$

$$Niter = 7$$

$$Nmov = 4$$

$$Ngen = 15000$$

Aunque el tiempo de computación para este valor a priori pueda parecer alto, 17 minutos, con respecto al resto de valores con el mismo *Ngen*, la reducción del coste es la mayor de todas las obtenidas y para obtener resultados similares con otro conjunto de parámetros se obtienen incluso tiempos de computación mayores, por lo que se ha optado por esta selección.

Una vez calibrado al algoritmo se realizarán un conjunto de pruebas para comprobar la bondad de este al someterlo a distintos escenarios.

4.3 Batería de pruebas

Para la prueba del algoritmo se han elegido una serie de problemas, para el VRPTW se han utilizado el conjunto de problemas propuesto por Solomon, dichos problemas incluyen todos los factores que pueden afectar a la resolución del problema, esta serie de problemas son de los más utilizados para la validación de nuevas técnicas

y métodos. Para el problema de Steiner se han utilizado los problemas de SteinLib Testdata Library un conjunto de problemas destinados a la validación de algoritmos desarrollados por el Zuse Institute Berlin, ZIB, uno de los centros más importantes en el campo de la matemática aplicada y la computación. En el caso del VRPTW los problemas se han separado por número de clientes, las distintas baterías de pruebas son:

- Batería 1: **25 clientes**, 12 escenarios desde r101 a r112.
- Batería 2: **50 clientes**, 5 escenarios r201, r202, r203, r204 y r205
- Batería 3: **100 clientes**, 5 escenarios r206, r207, r208, r209 y r210

Para el problema de Steiner se ha empleado una única batería compuesta por 6 escenarios abarcando los problemas b01, b02, b03, b04, b05 y b06.

Cada escenario se ha ejecutado 5 veces y se presentarán los resultados en las siguientes tablas en las que se recogen el valor promedio del coste, el mejor valor, la desviación típica y el tiempo de ejecución.

4.3.1 Batería 1

	Anémonas:75	*Niter*:7	*Nmov*:5	Iteraciones:20000
Problema	Promedio	Mejor valor	Desviación	Tiempo de ejecución (min)
r101	2196.042026709192	2136.772236982626	3’09417224411e-5	7’8349
r102	1777.513122856435	1697.5843125419995	3’9047554501e-05	7’785
r103	1398.6241679847612	1358.278172154105	4’8719432607e-05	7’912
r104	1126.4065232811479	933.9409741931761	6’3813797875e-05	7’614
r105	1838.8846856326127	1772.6627979219109	3’7008778897e-05	7’489
r106	1416.516109427712	1335.7215183471908	4’9583770789e-05	7’741
r107	1258.4294834712723	1182.5114644755374	5’5139367357e-05	7’368
r108	1029.385631176164	970.3234707470455	7’1933017703e-05	7’812
r109	1514.0862457838205	1476.6000926219454	4’7866849305e-05	7’419
r110	1272.4787326230594	1158.6555544196708	5’8230999708e-05	7’297
r111	1280.3194064263953	1205.5026696805135	5’3714237441e-05	8’012
r112	952.7112251318719	924.4664246153632	6’6741702497e-05	7’672

Tabla 1: Batería 1

4.3.2 Batería 2

	Anémonas:75	*Niter*:7	*Nmov*:5	Iteraciones:20000
Problema	Promedio	Mejor valor	Desviación	Tiempo de ejecución (min)
r201	14610.47319674252	13213.403204457489	4’4924619813e-06	20’569
r202	10781.862246997902	9963.656858284907	6’6935620132e-06	17’345
r203	7096.507507510012	6452.581612753732	1’0962271117e-05	18’468
r204	3004.3461722300735	2420.11251506949	3’0651937197e-05	21’247
r205	11264.54334433682	10766.981326635927	6’3724626200e-06	17’941

Tabla 2: Batería 2

4.3.3 Batería 3

	Anémonas:75	*Niter*:7	*Nmov*:5	Iteraciones:20000
Problema	Promedio	Mejor valor	Desviación	Tiempo de ejecución (min)
r206	19157.36919366217	18419.150269824106	3’4356949815e-06	57’861
r207	12627.025833300773	12303.610127773032	5’5582379332e-06	55’395
r208	7246.020002888338	6773.6910523343395	1’1317944616e-05	51’587
r209	19018.88669411923	17765.279796012812	3’2558987422e-06	56’427
r210	18653.606695899733	17561.4439103213	3’2124193662e-06	53’682

Tabla 3: Batería 3

4.3.4 Batería 4, problema de Steiner.

	Anémonas:60	*Niter*:7	*Nmov*:4	Iteraciones:15000
Problema	Promedio	Mejor valor	Desviación	Tiempo de ejecución (min)
b01	159.7	147	19’63553095701053	21’343
b02	106.4	86	17’99278709299092	17’681
b03	175.6	164	16’48925925273845	20’738
b04	135.8	120	21’39300751599302	18’245
b05	142.8	127	20’26149674080860	19’361

Tabla 4: Batería 4

4.4 Batería de pruebas algoritmo genético

Con el objetivo de comparar los resultados obtenidos por el algoritmo NEMO se han realizado los mismos experimentos anteriores pero esta vez utilizando un algoritmo genético simple. Los resultados obtenidos son:

4.4.1 Batería 1

	Población: 525	Iteraciones: 20000		
Problema	Promedio	Mejor valor	Desviación	Tiempo de ejecución (min)
r101	1202.4965670406907	1173.6380880334245	3.8259020448e-05	43’643
r102	964.8119756116764	901.8745832927467	4.1270285669e-05	37’127
r103	785.4751124954873	766.1015422635185	4.5917523772e-05	36’574
r104	697.4321402568712	657.0909904034863	5.0914023120e-05	36’214
r105	1003.2954053024863	895.4467682756074	4.8741201126e-05	37’429
r106	851.7259323987589	757.3649582036345	4.9085445630e-05	36’843
r107	742.9578113927197	708.0080804246086	5.0166685902e-05	36’173
r108	657.0958634212157	630.7662635537869	5.0383119023e-05	39’321
r109	802.5153256620387	709.588722622386	5.1686565186e-05	36’437
r110	715.3987961790938	617.5708597268856	5.4687163743e-05	35’891
r111	718.898641156386	718.898641156386	5.6588695354e-05	36’397
r112	618.2085442751732	543.9324771313294	5.6022860895e-05	36’592

Tabla 5: Batería 1 Algoritmo Genético

4.4.2 Batería 2

	Población: 525	Iteraciones: 20000		
Problema	Promedio	Mejor valor	Desviación	Tiempo de ejecución (min)
r201	3978.8207070705516	3104.305035557668	1.7581579895e-05	69’124
r202	2531.704168006129	2365.1291366570395	2.1761925465e-05	68’793
r203	1836.6405429376084	1783.1951859747337	2.6772605260e-05	68’257
r204	1615.7064758979257	1523.8232398797736	2.8727551575e-05	68’043
r205	2170.6966229150544	1972.67873648239	2.9944260636e-05	69’271

Tabla 6: Batería 2 Algoritmo Genético

4.4.3 Batería 3

	Población: 525	Iteraciones: 20000		
Problema	Promedio	Mejor valor	Desviación	Tiempo de ejecución (min)
r206	6145.134232158714	6115.170619540618	8.4182450959e-06	208’615
r207	4242.90298776602	4213.829602270944	1.1791037548e-05	209’862
r208	3237.891169707907	3223.257138513839	1.2660179914e-05	156’789
r209	4930.96662011356	4899.716345566052	1.3342506011e-05	157’682
r210	5237.932370827928	5215.402266312565	1.2150089541e-05	155’391

Tabla 7: Batería 3 Algoritmo Genético

4.4.4 Batería 4

	Población: 420	Iteraciones: 15000		
Problema	Promedio	Mejor valor	Desviación	Tiempo de ejecución (min)
b01	158	102	21.34885848244	19’821
b02	152.76	98	20.17342608932	15’358
b03	168.14	115	17.24189579256	16’364
b04	172.85	96	21.71018797488	18’973
b05	156.27	93	22.49502892931	19’364

Tabla 8: Batería 4 Algoritmo Genético

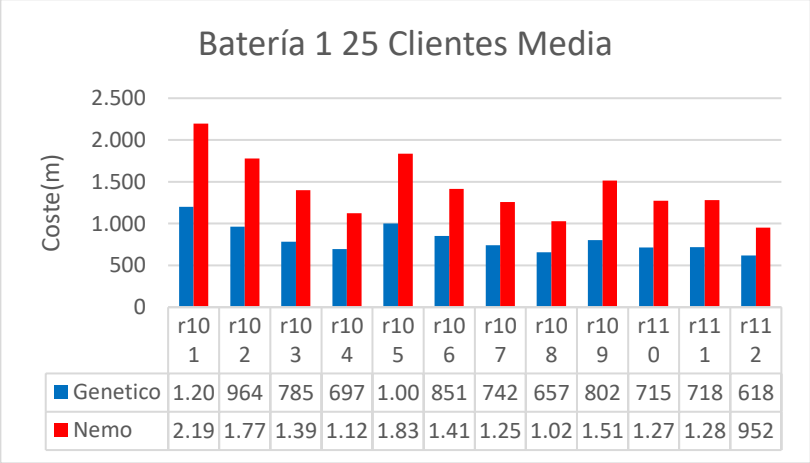
A continuación, se detalla el equipo con el que se han realizado las pruebas, ya que dependiendo de los componentes de este el tiempo de computación variará:

- Ordenador: Asus ROG.
- Sistema Operativo: Windows 10.
- Procesador: Intel Core i7-6700HQ 2.60 GHz.
- Memoria RAM: 16 GB DDR3.
- Tarjeta Gráfica: Nvidia Geforce GTX 960M.

5 ANÁLISIS DE LOS RESULTADOS

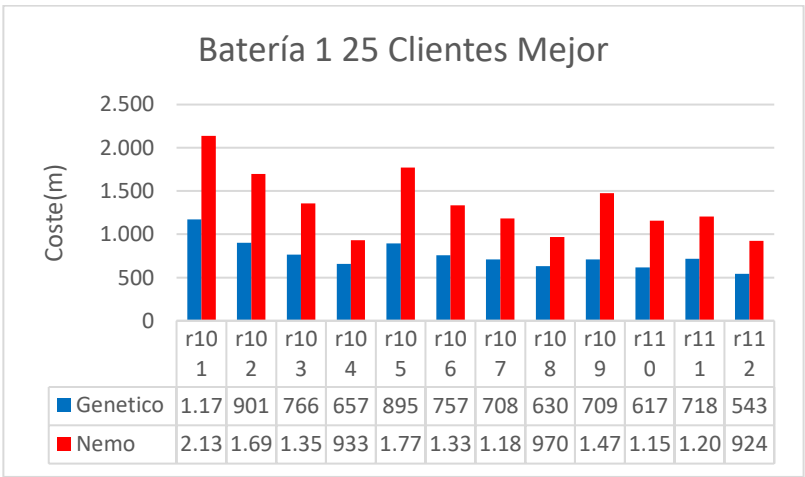
En este capítulo se procede a analizar el conjunto de resultados obtenidos en el capítulo anterior tras la ejecución del algoritmo usando las distintas baterías de pruebas. Por cada batería de pruebas se realizarán tres gráficas, comparando tanto el mejor resultado, como el promedio de la población y el tiempo de computación empleado en la resolución del problema.

5.1 Batería de pruebas 1



	r101	r102	r103	r104	r105	r106	r107	r108	r109	r110	r111	r112
Genetico	1.20	964	785	697	1.00	851	742	657	802	715	718	618
Nemo	2.19	1.77	1.39	1.12	1.83	1.41	1.25	1.02	1.51	1.27	1.28	952

Figura 24: Comparación de resultados media, batería 1



	r101	r102	r103	r104	r105	r106	r107	r108	r109	r110	r111	r112
Genetico	1.17	901	766	657	895	757	708	630	709	617	718	543
Nemo	2.13	1.69	1.35	933	1.77	1.33	1.18	970	1.47	1.15	1.20	924

Figura 25: Comparación de resultados mejor, batería 1

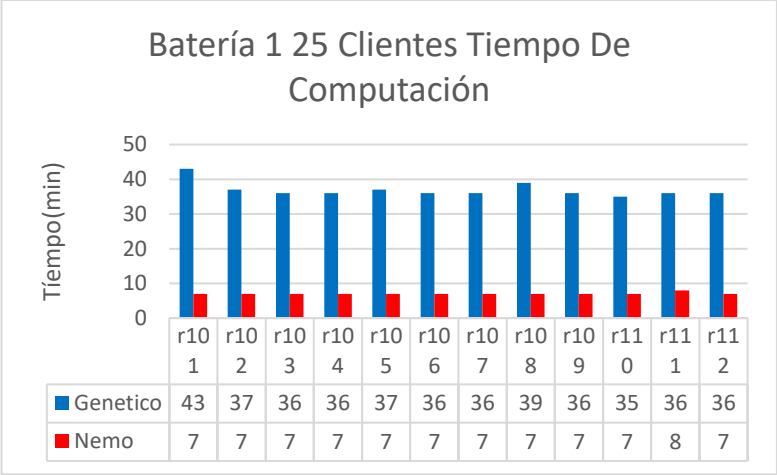


Figura 26: Comparación de resultados tiempo de computación, batería 1

Para esta primera batería se observa una gran diferencia en cuanto a la calidad de las soluciones obtenidas. Los resultados arrojados por el algoritmo genético son mejores tanto como la media general de la población como los mejores individuos que se han obtenido. En cambio, en cuanto a tiempo de computación el algoritmo genético ha sido mucho más lento que el algoritmo NEMO el cual se mantiene en torno a unos 7 minutos de tiempo de computación frente a la media de 37 minutos del algoritmo genético.

5.2 Batería de pruebas 2

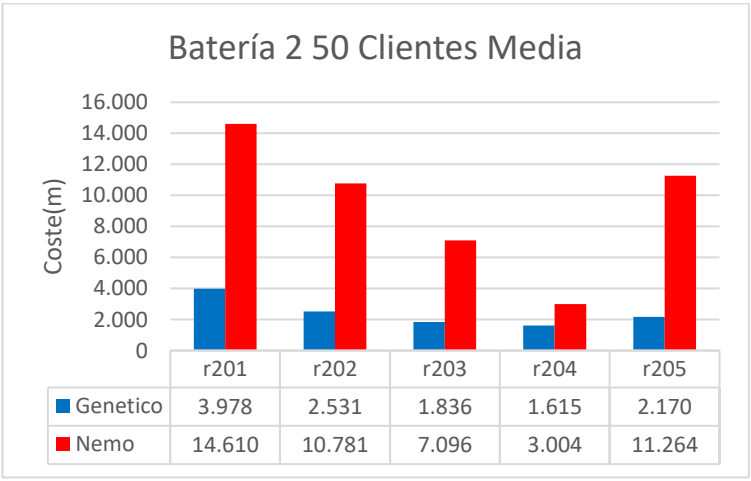


Figura 27: Comparación de resultados media, batería 2

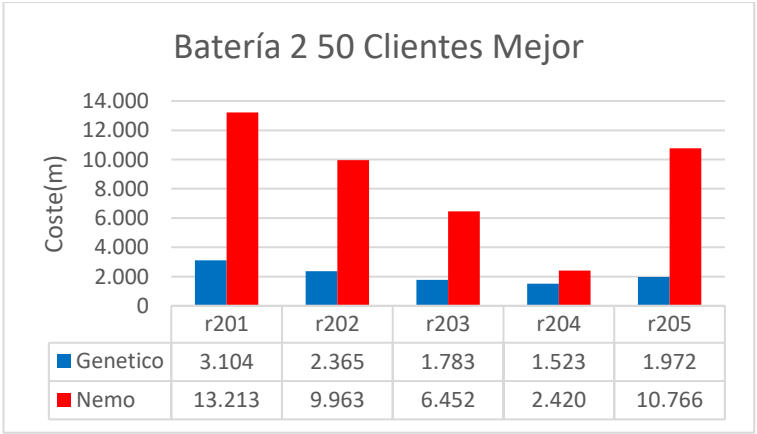


Figura 28: Comparación de resultados mejor, batería 2

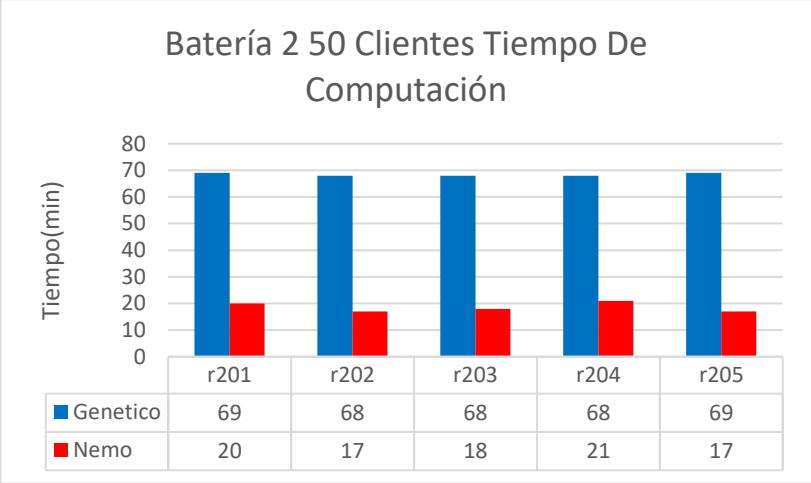


Figura 29: Comparación de resultados tiempo de computación, batería 2

En los problemas de la segunda batería los resultados obtenidos son similares a los que se han obtenido para la primera batería de problemas. De nuevo el algoritmo genético arroja mejores resultados en cuanto al coste de los individuos tanto en la media de la población como en los mejores individuos obtenidos, pero en cuanto a tiempo de computación el algoritmo NEMO es claramente superior con un tiempo de computación aproximadamente 3'5 veces menor que el tiempo de computación del algoritmo Genético.

5.3 Batería de pruebas 3

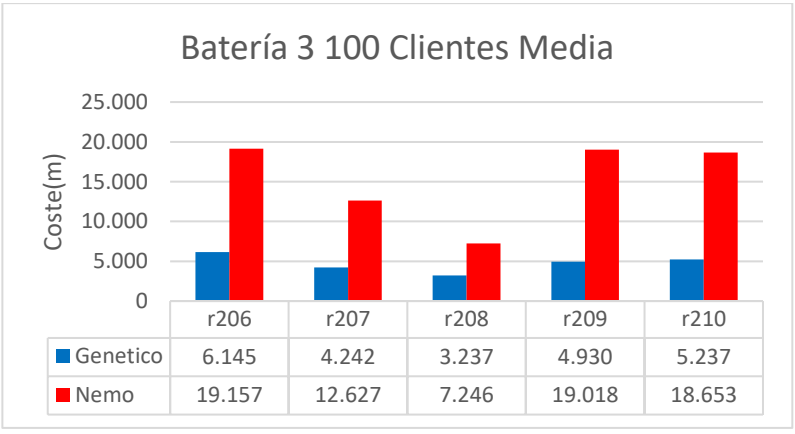


Figura 30: Comparación de resultados media, batería 3

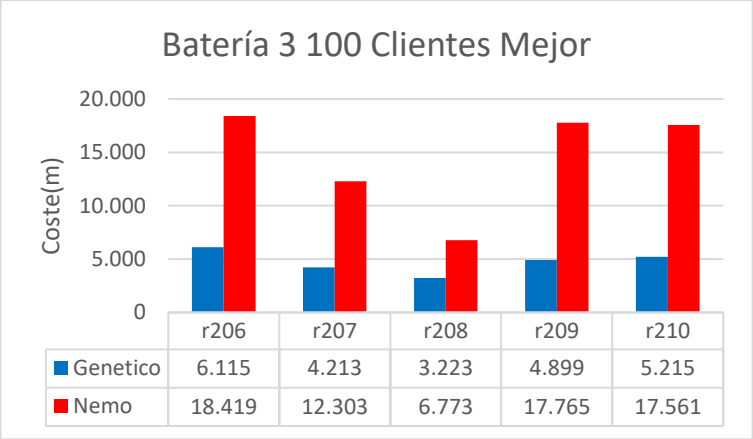


Figura 31: Comprobación de resultados mejor, batería 3

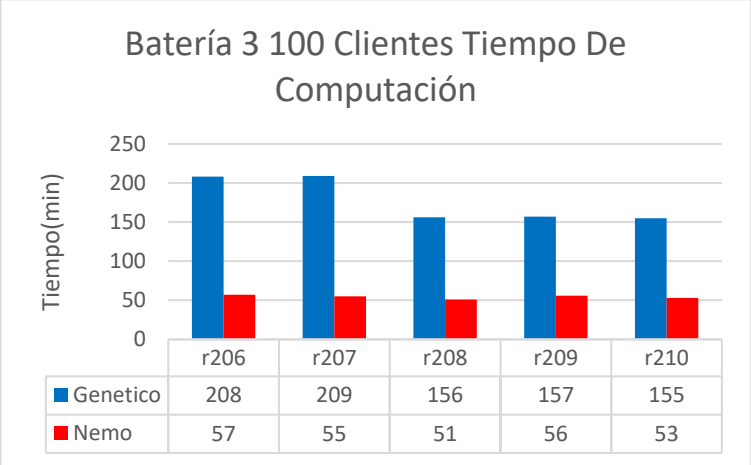


Figura 32: Comparación de resultados tiempo de computación, batería 3

De nuevo en la última batería de problemas dedicadas a el VRPTW se vuelven a obtener resultados parejos a los anteriores, manteniendo el algoritmo Genético el liderazgo en cuanto al coste total de las soluciones obtenidas, mientras que el algoritmo NEMO mantiene un tiempo de computación en torno a una hora, el algoritmo Genético tarda más del doble en los mismos problemas.

En la siguiente figura se observa la evolución del tiempo de computación con respecto al número de clientes

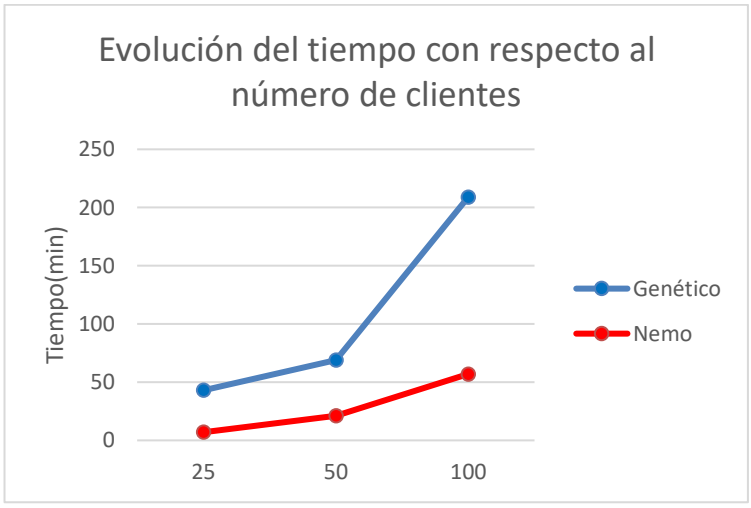


Figura 33: Evolución del tiempo con respecto al número de clientes (VRPTW)

El tiempo de computación en el caso del algoritmo NEMO ha evolucionado de forma mucho más lineal que en

el caso del algoritmo Genético el cual al pasar de 50 clientes a 100 experimenta una fuerte variación que en el caso del algoritmo NEMO no se produce.

5.4 Batería de pruebas 4

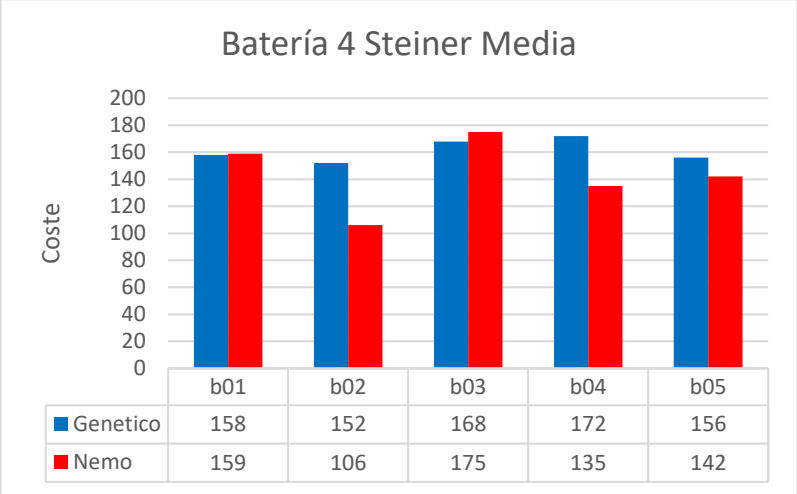


Figura 34: Comparación de resultados media, batería 4

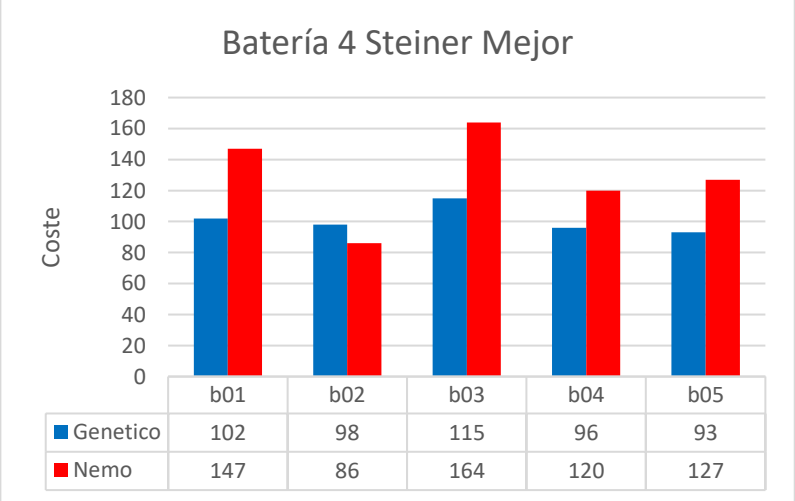


Figura 35: Comparación de resultados mejor, batería 4

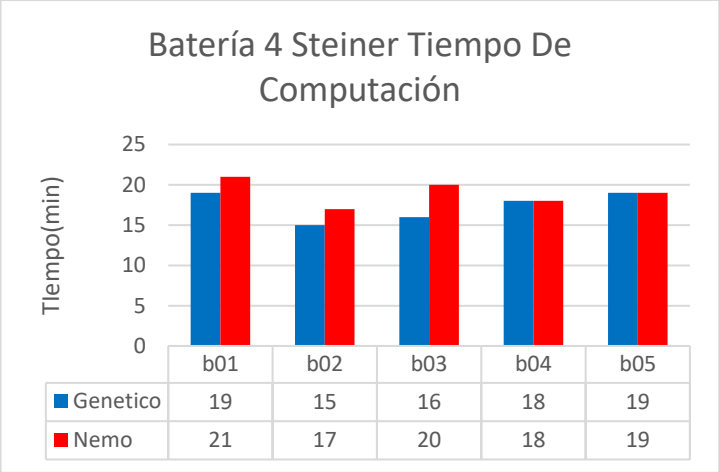


Figura 36: Comparación de resultados tiempo de computación, batería 4

En esta última batería de problemas en el que en vez del VRPTW se resuelve el problema de Steiner los resultados obtenidos difieren de los anteriores en las cuales, el algoritmo Genético era claramente superior en los costes obtenidos por sus individuos mientras que el algoritmo NEMO destacaba por su tiempo de computación. En este caso ambos algoritmos obtienen unos resultados más parejos, para el algoritmo NEMO el coste de la media de la población es un poco menor en media, aunque los mejores individuos se han obtenido mediante el algoritmo Genético. A su vez el tiempo de computación se ha igualado siendo en algunos casos menor el del algoritmo Genético.

6 CONCLUSIONES

El objetivo de este capítulo es remarcar las partes más importantes del trabajo llevado a cabo. Además, se extraen una serie de conclusiones tras la obtención y el análisis de los resultados de las distintas baterías de pruebas. A modo de resumen se comentarán los pasos que se han seguido durante el desarrollo del trabajo:

- Primero se realizó una breve introducción a los algoritmos bio-inspirados y se comentaron los objetivos que se tratarían de alcanzar con este trabajo.
- Posteriormente, se realizó una taxionomía con los distintos tipos de algoritmos que se iban a tratar y se presentaron distintas aplicaciones de los mismo juntamente con una serie de librerías existentes para su implementación.
- A continuación, se presentó el algoritmo NEMO tanto su inspiración natural como las distintas partes que componen el algoritmo.
- En el siguiente capítulo se realizaron distintas pruebas con el fin de calibrar el algoritmo para obtener los mejores resultados posibles al aplicar el algoritmo a la batería de problemas.
- Por último, se ha aplicado el algoritmo NEMO una vez calibrado a una batería de problemas comparando los resultados obtenidos con los proporcionados por un algoritmo genético al ser usado para la misma batería de problemas.

A la vista de los resultados obtenidos se pueden obtener las siguientes conclusiones:

1. Se ha desarrollado y calibrado con éxito el algoritmo NEMO, aunque, los costes de las soluciones obtenidas por este para el VRPTW son muy pobres en comparación con las obtenidas por el algoritmo genético en igualdad de condiciones.
2. Por otra parte, el algoritmo NEMO se ha ejecutado en unos tiempos de computación muy inferiores en comparación con el algoritmo genético.

Por lo que se puede concluir que el algoritmo NEMO puede ser una buena opción para el problema de Steiner, pero para el VRPTW, aunque se haya ejecutado en unos tiempos muy inferiores al algoritmo genético la gran diferencia en el coste de las soluciones no lo hace una buena opción para la resolución de este problema. A pesar de que no se han obtenido resultados positivos, se han cumplido los objetivos planteados al inicio de este trabajo, en los cuales se postulaba que el objetivo era el desarrollo y la calibración del algoritmo para posteriormente comprobar la bondad de este. Lo cual es justamente lo que se ha realizado a lo largo del capítulo 3 y el capítulo 4 aunque, desafortunadamente se ha comprobado que el algoritmo no es una buena opción para el VRPTW y para el problema de Steiner arroja resultados similares, por lo que no podemos asegurar que el algoritmo NEMO sea válido para problemas del tipo VRPTW, pero si puede ser una opción para la resolución de problemas de Steiner.

REFERENCIAS

http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf

Al-Shamma, Basil & Gosselin, Olivier & King, Peter. (2018). History Matching Using Hybrid Parameterisation and Optimisation Methods. 10.2118/190776-MS.

Yatim, Jamaludin & Zain, Azlan & Bazin, Nor Erne Nazira. (2014). Cuckoo Search Algorithm for Optimization Problems—A Literature Review and its Applications. Applied Artificial Intelligence. 28. 10.1080/08839514.2014.904599.

Yigit, Tuncay & Unsal, Ozkan & Deperlioglu, Omer. (2018). Using the metaheuristic methods for real-time optimisation of dynamic school bus routing problem and an application. International Journal of Bio-Inspired Computation. 11. 123. 10.1504/IJBIC.2018.091236.

Li, Jing & Ke, Liangjun & Ye, Gangqiang & Zhang, Tianjiao. (2017). Ant colony optimisation for the routing problem in the constellation network with node satellite constraint. International Journal of Bio-Inspired Computation. 10. 267. 10.1504/IJBIC.2017.087919.

Saranya, G & Nehemiah, Khanna & Arputharaj, Kannan. (2018). Hybrid particle swarm optimisation with mutation for code smell detection. International Journal of Bio-Inspired Computation. 12. 186. 10.1504/IJBIC.2018.094624.

Peres, Wesley & V.R. Silva, Valceres & Coelho, Francisco & Junior, Ivo & Passos Filho, João Alberto. (2018). A memetic algorithm for power system damping controllers design. International Journal of Bio-Inspired Computation. 11. 110. 10.1504/IJBIC.2018.091235.

D. Chitara, K. R. Niazi, A. Swarnkar and N. Gupta, "Cuckoo Search Optimization Algorithm for Designing of a Multimachine Power System Stabilizer," in *IEEE Transactions on Industry Applications*, vol. 54, no. 4, pp. 3056-3065, July-Aug. 2018. doi: 10.1109/TIA.2018.2811725

T. Karataev, F. Güneş and S. Demirel, "Cuckoo search design optimization and analysis of matching circuits composed of CRLH transmission cells," *2015 Twelve International Conference on Electronics Computer and Computation (ICECCO)*, Almaty, 2015, pp. 1-4. doi: 10.1109/ICECCO.2015.7416883

Guadix, José & Cortés, Pablo & Muñuzuri, Jesús & Onieva, Luis. (2018). A discrete particle swarm optimisation algorithm to operate distributed energy generation networks efficiently. International Journal of Bio-Inspired Computation. 12. 226. 10.1504/IJBIC.2018.10017840.

Hinrichs, Christian & Sonnenschein, Michael. (2017). A distributed combinatorial optimisation heuristic for the scheduling of energy resources represented by self-interested agents. International Journal of Bio-Inspired Computation. 10. 69. 10.1504/IJBIC.2017.085895.

Ouaarab, Aziz & Ahiod, Belaïd & Yang, Xin-She. (2014). Improved and Discrete Cuckoo Search for Solving the Travelling Salesman Problem. 10.1007/978-3-319-02141-6_4.

Demir, Emrah. (2012). Models and Algorithms for the Pollution-Routing Problem and Its Variations. 10.13140/2.1.3577.1680.

Thakur, Nileshsingh & Sharma, Kartik. (2014). An Approach for Single Object Detection in Images. International Journal of Image Processing (IJIP). 8. 278-293.

ANEXO A: EXPLICACIÓN DEL CÓDIGO

1 Algoritmo genético

Se procede a realizar la explicación del código realizado para la resolución del problema VRPTW mediante el algoritmo genético simple. Varias de estas funciones se reutilizan posteriormente en el algoritmo NEMO.

1.1 Leedatos

Esta función se crea con el objetivo de resolver el problema de la obtención de los distintos datos del problema, las coordenadas de cada cliente, la demanda y la ventana temporal para la entrega, y además tener toda esta información ordenada y fácilmente accesible. Se ha optado por utilizar un diccionario, que son un tipo de dato de Python que permite asociar cada dato con una clave, esto se ajusta perfectamente a los datos de nuestro problema ya que la realización clave-valor es perfectamente observable. La estructura del diccionario es la siguiente:

```
datos = {
    'max_veh': int,
    'capacidad': float,
    'cliente_%d': {
        'coordenadas': {
            'x': float,
            'y': float
        }
        'demanda': float,
        'time_start': float,
        'time_limit': float,
        'tiempo_servicio': float
    }
}
```

En el apartado cliente existirá un campo por cliente, siendo el cliente 0 el almacén.

Pasemos al código en sí

```
global matriz_distancia
with open(nombrefichero) as f:
```

Lo primero es la utilización de *global* esta palabra clave permitirá modificar la variable global *matriz_distancia* dentro de la función, sin *global* solo podríamos leer su contenido, esta variable se trata de una matriz que tendrá tantas filas y columnas como clientes haya. En ella cada fila y cada columna representa un cliente y los elementos son la distancia que hay entre ellos. Esta variable se inicializa aquí y se usará posteriormente en otras funciones que explicaremos más adelante.

Posteriormente abrimos el archivo utilizando *with*, esto nos permite realizar las operaciones pertinentes que necesitemos con el archivo y asegura que cuando termine su uso el archivo se cerrará, este método se introdujo en la versión de Python 2.6 para sustituir el uso de un bloque *try/finally* para asegurar que se cierren los recursos asociados.

```
        for numLin,linea in enumerate(f,start=1):
            if numLin in [1,2,3,4,6,7,8]:
```

```
                pass
            elif numLin == 5:
                # Número maximo de vehículos y capacidad de carga
                aux=linea.strip().split()
                datosfichero['max_veh'] = int(aux[0])
                datosfichero['capacidad'] = float(aux[1])

            elif (numLin>=9) and (numLin<=9+numClientes):
                # Resto de datos de los clientes
                aux = linea.strip().split()
                datosfichero['cliente_%d' % int(aux[0])] = {
                    'coordenadas':{
                        'x': float(aux[1]),
                        'y': float(aux[2])
                    },
                    'demanda': float(aux[3]),
                    'time_start': float(aux[4]),
                    'time_limit': float(aux[5]),
                    'tiempo_servicio': float(aux[6]),
```

Tras abrir el archivo pasamos a la lectura para esto se utiliza la función *enumerate* que permite iterar a través de los distintos elementos de un iterable y además obtener la posición de dicho elemento, gracias a esto podemos ir recorriendo el archivo del cuál sabemos su estructura que es la siguiente

```
<Instance name>
<empty                                                              line>
VEHICLE
NUMBER                                                          CAPACITY
                                   K                                   Q
<empty                                                              line>
CUSTOMER
CUST  NO.   XCOORD.    YCOORD.     DEMAND    READY  TIME   DUE  DATE    SERVICE  TIME
 <empty                                                             line>
     0        x0         y1          q0          e0           l0             s0
     1        x1         y2          q1          e1           l1             s1
   ...      ...        ...         ...         ...          ...            ...
  100     x100       y100        q100        e100         l100           s100
```

Los primeros datos de interés se encuentran en la línea 5 y posteriormente desde la línea 9 encontraremos los datos de los clientes, se obtendrán tantos datos adicionales como indique la variable de entrada *numClientes*.

```
for i in range(0, numClientes+1):
            for j in range(0, numClientes+1):
                matriz_distancia[i][j]=((datosfichero['cliente_%d' %i]['coordenadas']['x']
                                        -datosfichero['cliente_%d'%j]['coordenadas']['x'])**2
                                        + (datosfichero['cliente_%d' %i]['coordenadas']['y']
                                    - datosfichero['cliente_%d'%j]['coordenadas']['y'])**2)**0.5
```

Una vez obtenidos todos los datos procedemos a inicializar la variable *matriz_distancia* usando las coordenadas de los distintos clientes que ya hemos guardado.

1.2.decodifica

El objetivo de esta función es pasar de la codificación del individuo, en el cual se incluye la ruta completa de todos los vehículos, a un conjunto de las subrutas que seguirá cada vehículo.

Pasemos al código:

```
def decodifica(individuo,datos):
    ruta = []
    subruta = []
    cargaVeh = 0
    tiempoTrans = 0
    ultimoCliente = 0
    for cliente in individuo:
```

Primero inicializamos las variables ruta será una lista de listas en el cual se incluirán las distintas subrutas de los vehículos, siendo *subruta* otra lista en la que se irán añadiendo los clientes para formarla. Las variables *cargaVeh* y *tiempoTrans* se usarán para determinar cuando la subruta ha llegado a su fin porque no se cumplen las restricciones pertinentes, *ultimoCliente* almacenará el último cliente visitado que se usará para calcular la distancia que tiene que recorrer el vehículo. Tras esto entramos en el bucle for en el que iteramos a través de *individuo* recorriendo cada cliente de la lista.

```
sigCarga=cargaVeh+datos['cliente_%d' %cliente]['demanda']
        sigTime=(tiempoTrans+matriz_distancia[ultimoCliente][cliente]
         + datos['cliente_%d' %cliente]['tiempo_servicio']
         + matriz_distancia[cliente][0])
```

Calculamos la carga del vehículo para el siguiente cliente y el tiempo que pasaría si fuese al cliente, le sirviese y retornase al almacén, para esto se usa *matriz_distancia* calculada anteriormente que permite obtener los datos necesarios.

```
        if(sigCarga<=datos['capacidad']) and
sigTime<=datos['cliente_0']['time_limit']):
            subruta.append(cliente)
            cargaVeh = sigCarga
            tiempoTrans = sigTime-matriz_distancia[cliente][0]

        else:
            ruta.append(subruta)
            subruta = [cliente]
            cargaVeh = datos['cliente_%d' %cliente]['demanda']
            tiempoTrans = matriz_distancia[0][cliente]
+datos['cliente_%d' %cliente]['tiempo_servicio']
        ultimoCliente = cliente
```

Tras el cálculo comprobamos si se cumplen las restricciones, si la carga del vehículo es menor que el límite y si el tiempo para ir al siguiente cliente, servirlo y volver es menor que el tiempo límite para volver al almacén, significa que podemos seguir en la ruta, añadimos el cliente actual a la subruta, actualizamos las cargas del vehículo, y actualizamos el tiempo transcurrido sin tener en cuenta el tiempo que tardaría en volver al almacén. Para añadir el cliente a la subruta utilizamos el método *append* que añade el elemento que reciba a la lista. Si no se cumplen las restricciones añadimos la subruta a la ruta, e inicializamos la nueva subruta empezando en el cliente nuevo y actualizamos la carga del vehículo y el tiempo transcurrido, finalmente actualizamos el último cliente con el cliente actual.

```
if subruta!= []:
        ruta.append(subruta)
    return ruta
```

Por último, antes de devolver *ruta* comprobamos si hay alguna última subruta, para en el caso de que la haya añadirla también a la ruta.

1.3.Evaluación

Esta función es la encargada de evaluar a los distintos individuos de nuestra población y devolverá su *fitness*, la propiedad que define como de buenos son los individuos.

```
ruta = decodifica(ind,datos)
costeTotal = 0
for subRuta in ruta:
```

Primero utilizamos la función *decodifica*, descrita previamente, para pasar de la codificación de los individuos a una lista que contiene las distintas rutas que seguirán los vehículos, inicializamos el coste total y pasamos a recorrer las diferentes rutas de nuestro individuo.

```
        srTimeCost = 0
        srDistancia = 0
        tiempoTrans = 0
        ultimo_cliente = 0
        for cliente in subRuta:
```

Para cada ruta inicializamos a 0 las distintas variables representadas, el coste temporal, la distancia, el tiempo transcurrido y el último cliente visitado, tras esto empezamos a recorrer los clientes.

```
            distancia = matriz_distancia[ultimo_cliente][cliente]
            srDistancia = srDistancia+distancia
```

Actualizamos la distancia recorrida, de forma inmediata gracias a la matriz calculada anteriormente en la función *leedatos*.

```
tiempo_llegada = tiempoTrans+distancia
timeCost = wait_cost*max(datos['cliente_%d'%cliente]['time_start']-tiempo_llegada,0)\
   + delayCost * max(tiempo_llegada - datos['cliente_%d' % cliente]['time_limit'], 0)
srTimeCost = srTimeCost+timeCost
```

Tras actualizar la distancia, actualizamos el coste temporal, para ello calculamos el tiempo de llegada al próximo cliente y comprobamos si tenemos alguna penalización por haber llegado antes de tiempo o por haber llegado fuera del tiempo límite del cliente, si cumplimos ambos tiempos el coste temporal será 0.

```
     tiempoTrans                                                              =
tiempo_llegada+datos['cliente_%d'%cliente]['tiempo_servicio']
     ultimo_cliente = cliente
```

Por último, actualizamos el tiempo transcurrido y el último cliente visitado y pasamos al siguiente cliente de la subruta.

```
        # Calculamos el coste de transporte
        srDistancia = srDistancia+matriz_distancia[ultimo_cliente][0]
        srTransCost = coste_ini+coste_uni*srDistancia
        # Calculamos el coste total
        costeTotal = srTimeCost+srTransCost+costeTotal
```

Después de haber recorrido todos los clientes de una subruta, actualizamos la distancia de la subruta teniendo en cuento la distancia desde el último cliente al almacén, y calculamos el coste del transporte, por último, se actualiza el coste total de la ruta con el coste temporal y el coste de transporte de la subruta.

```
    fitness = 1.0/costeTotal
    return fitness,
```

Una vez efectuados todos los cálculos, el *fitness* se toma como la inversa del coste total, el objetivo del algoritmo será maximizar este *fitness*. Cabe denotar que *fitness* se devuelve como una tupla, condición necesaria para el buen funcionamiento del algoritmo incluido en los módulos DEAP, este es el motivo de la coma añadida en la última instrucción, para que *fitness* sea una tupla y no una variable de tipo flotante.

1.4. varAnd

Esta función se encuentra dentro de los módulos DEAP, se trata de una función simple en la cual se crea la descendencia.

Primero se clona la población de entrada dada por la variable *population*, usando el método *clone* de la clase *toolbox*, que es una clase de DEAP en la cual están incluidos los operadores necesarios para realizar la operación.

```
offspring = [toolbox.clone(ind) for ind in population]
```

Tras esto iteramos recorriendo *offspring* y realizando la operación de cruce y la operación de mutación según la probabilidad de cruce, *cxpb*, y la probabilidad de mutación, *mutpb*, respectivamente.

```
    # Apply crossover and mutation on the offspring
for i in range(1, len(offspring), 2):
 if random.random() < cxpb:
  offspring[i - 1], offspring[i] = toolbox.mate(offspring[i - 1],
                                                offspring[i])
  del offspring[i - 1].fitness.values, offspring[i].fitness.values

for i in range(len(offspring)):
 if random.random() < mutpb:
  offspring[i], = toolbox.mutate(offspring[i])
  del offspring[i].fitness.values
return offspring
```

Dado que se hace una copia de la población previamente, tras realizar el cruce y la mutación se elimina el *fitness* de *offspring*, ya que este deberá volver a ser evaluado porque no será igual que el de sus padres.

1.5.eaSimple

Esta función también se encuentra dentro de los módulos DEAP, pero se le han realizado una serie de modificaciones para evitar problemas que daban soluciones incorrectas, como la visita de clientes que ya habían sido visitados.

Este algoritmo se trata del algoritmo evolutivo más simple tal y como se presenta en [Back, 2000]

Lo primero que se hace al entrar en esta función es crear un objeto *Logbook* en el cual se registrarán las distintas estadísticas recogidas, además de evaluar los individuos de la población, y dependiendo del valor de las entradas *halloffame* y *verbose*, se decidirá si se guarda el mejor individuo de la población y si se muestra por consola el valor de las estadísticas en cada iteración.

```
logbook = tools.Logbook()
logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])

# Evaluate the individuals with an invalid fitness
invalid_ind = [ind for ind in population if not ind.fitness.valid]
```

```
fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)

for ind, fit in zip(invalid_ind, fitnesses):
 ind.fitness.values = fit

if halloffame is not None:
 halloffame.update(population)

record = stats.compile(population) if stats else {}
logbook.record(gen=0, nevals=len(invalid_ind), **record)

if verbose:
 print(logbook.stream)
```

Una vez se han realizado las inicializaciones necesarias se empieza el proceso evolutivo

```
for gen in range(1, ngen + 1):

# Select the next generation individuals
 offspring = toolbox.select(population, len(population))

# Vary the pool of individuals
 offspring = varAnd(offspring, toolbox, cxpb, mutpb)
```

Este proceso se repetirá tantas veces como indique la variable de entrada *ngen*, primero se selecciona la próxima generación de individuos usando el método *select* del objeto *toolbox*, tras esta selección los individuos se cruzan y mutan en la función *varAnd*, explicada previamente.

```
for offs_ind,individuo in enumerate(offspring):
 for indice,cliente in enumerate(individuo):
  if (cliente>num_clientes) or (cliente==0):
   individuo[indice]=random.randint(1,num_clientes)

  if individuo.count(cliente)>1 :
    repetido=True
    while(repetido):
     individuo[indice] = random.randint(1, num_clientes)
     repetido=False
     for j,x in enumerate(individuo):
      if individuo[indice]==x and j!=indice:
```

```
        repetido=True
 offspring[offs_ind]=individuo
```

Este fragmento de código recoge las modificaciones que han sido introducidas, tras realizar las operaciones de cruce y mutación, recorremos cada individuo de esta y vamos comprobando si algún elemento del individuo es mayor que el límite permitido, por ejemplo, si el problema consiste en 25 clientes no puede haber ningún elemento mayor que 25. También comprobamos que no tengamos ningún elemento repetido, utilizando el método *count* que devuelve cuantos elementos que indiquemos, hay en una lista, si es mayor que 1 habrá algún elemento repetido, esto lo indicamos con la variable *repetido*, mientras *repetido* sea *True* cambiaremos el valor del elemento repetido por otro aleatorio y volveremos a comprobar si se ha vuelto a repetir, dado que este reemplazo se realiza de manera aleatoria, por último, se asigna el individuo a la descendencia tras haber comprobado que cumple todos los requisitos.

```
# Evaluate the individuals with an invalid fitness
 invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
 fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)

 for ind, fit in zip(invalid_ind, fitnesses):
  ind.fitness.values = fit

  # Update the hall of fame with the generated individuals
 if halloffame is not None:
  halloffame.update(offspring)

 # Replace the current population by the offspring
 population[:] = offspring

 # Append the current generation statistics to the logbook
 record = stats.compile(population) if stats else {}
 logbook.record(gen=gen, nevals=len(invalid_ind), **record)

 if verbose:
  print(logbook.stream)

return population, logbook
```

Tras producir la descendencia la evaluamos y actualizamos el *hall of fame* si corresponde, después de la evaluación sustituimos la población por la nueva descendencia y añadimos el nuevo conjunto de estadística al objeto *logbook*, además, si procede se muestran por pantalla.

1.6.main

En esta función principalmente servirá para llamar a las funciones descritas previamente e inicializar las clases y métodos de DEAP.

```
    leedatos(datos, 'prueba.txt', indTam)
    creator.create('FitnessMax', base.Fitness, weights=(1.0,))
    creator.create('Individual', list, fitness=creator.FitnessMax)
    toolbox = base.Toolbox()
```

Primero llamamos a la función *leedatos* para tener los datos del problema definidos, posteriormente usando la clase *creator* y su método *créate* creamos la clase *FitnessMax* que hereda de la clase *Fitness*, que recibe el atributo *wieghts* que este al ser un valor positivo indica que el objetivo es maximizar el *fitness*, posteriormente creamos la clase *Individual* que hereda la clase *list* y además contiene la clase *FitnessMax* creada previamente, también declaramos una instancia de la clase *Toolbox*.

```
toolbox.register('indexes', random.sample, range(1, indTam+1), indTam)

# Iniciamos las estructuras
toolbox.register('individual',       tools.initIterate,       creator.Individual,
toolbox.indexes)

toolbox.register('population', tools.initRepeat, list, toolbox.individual)

toolbox.register('evaluate',    evaluacion,    datos=datos,    coste_uni=coste_uni,
coste_ini=coste_ini, wait_cost=wait_cost, delayCost=delayCost)

toolbox.register("mate", tools.cxTwoPoint)

toolbox.register("mutate", tools.mutUniformInt, low=0, up=100,indpb=0.2)
# Independent probability  : for each attribute to be mutated

toolbox.register("select", tools.selTournament, tournsize=3)
pop = toolbox.population(n=pobTam)
```

En *toolbox* registramos todas las funciones a utilizar, este registro asocia un alias a una función ya existente, cuando queramos utilizar una de estas funciones utilizaremos dicho alias asignado en vez del nombre de la función, además permite fijar ciertos valores a la función que no necesitaremos especificar posteriormente al llamar a la función.

```
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
```

```
    stats.register("avg", np.mean)
    stats.register("std", np.std)
    stats.register("min", np.min)
    stats.register("max", np.max)
    # Llamamos al algoritmo
    pop, log = eaSimple(pop, toolbox, cxpb=0.85, mutpb=0.01,
ngen=100, stats=stats, halloffame=hof,
    verbose=True,num_clientes=indTam)
    bestInd = tools.selBest(pop, 1)[0]

    print(bestInd)
```

Tras el registro de las funciones en *toolbox*, definimos el *hall of fame*, y las estadísticas a calcular y llamamos al algoritmo implementado en *eaSimple*, una vez que este termine se selecciona el mejor individuo de la población y se muestra en consola.

2 Algoritmo Nemo VRPTW

Se procede a explicar el código del algoritmo NEMO. Por simplicidad solo se explicarán las funciones que no apareciesen en la implementación del algoritmo genético simple.

2.1. crea_anemonas

Función encargada de crear las distintas anemonas de la población, se crearán tantas anemonas como indique la variable de entrada *Na*, cada anemona contendrá una población de entre 2 y 7 individuos inicialmente. Para la creación se utilizará el objeto *toolbox*, que se tiene como variable de entrada, en el que estarán incluidas las funciones necesarias para el desarrollo del algoritmo.

```
def crea_anemonas(toolbox,Na):
    anemonas=[]
    for i in range(0,Na):
        p=toolbox.population(n=random.randint(2,7))
        anemonas.append(p)
    return anemonas
```

2.2. best

El objetivo de esta función es dado una lista de *fitnesses* y una lista de individuos, devuelve el individuo con mayor *fitness*.

```
def best(fitness,lista):
    maximo=0.0
    indice=0
    for i in range(0,len(fitness)):
        string_fit='%s' %fitness[i]
        if float(string_fit)>maximo:
            maximo=float(string_fit)
            indice=i
    return lista[indice]
```

2.3. actualiza_mejor

Tal como su nombre indica esta función se encarga de devolver siempre el individuo con mejor *fitness* de toda la población de anemonas. Para ello recibe como entrada una lista con los mejores individuos de cada iteración, el *toolbox* que contiene las funciones a emplear, el *fitness* del mejor individuo, y el mejor individuo hasta ahora. Se realizan las comprobaciones y si no hay ningún individuo que supere al mejor individuo que había, volverá a devolver el mejor hasta ahora.

```
def actualiza_mejor(lista,toolbox,fit_mejor,mejor_ant):
    fitnesses=toolbox.map(toolbox.evaluate,lista)
    mejor=best(list(fitnesses),lista)
    string_aux='%s' % mejor.fitness.values
    if(float(string_aux)<fit_mejor):
        mejor=mejor_ant
    return mejor
```

2.4. reproducción

Esta función aplica los operadores de cruce entre el macho y la hembra de la anemona indicada, el tamaño de la descendencia viene dado por la variable de entrada *TamDesc*, también recibirá como entrada el *toolbox* y la probabilidad de reproducción *cxpb*.

```
def reproduccion(anemona,toolbox,TamDesc,cxpb):
    descendencia=[]
    for i in range(1,TamDesc+1):
        if random.random() < cxpb:
            indv0,indv1=toolbox.mate(anemona[0],anemona[1])
            if i%2==0:
                descendencia.append(indv0)
            else:
                descendencia.append(indv1)
    for i in range(0,len(descendencia)):
        descendencia[i]=elimina_repetidos(descendencia[i])
    fitnesses = toolbox.map(toolbox.evaluate, descendencia)
    for ind, fit in zip(descendencia, fitnesses):
        ind.fitness.values = fit
    return descendencia
```

No solo realizará el cruce, sino que además se asegura de que en esta descendencia no haya individuos que contengan clientes repetidos llamando a la función *elimina_repetidos*, que se explicará posteriormente, y evaluará la descendencia antes de devolverla.

2.5. elimina_repetidos

Como hemos anticipado anteriormente dado un individuo, en esta función comprobamos que no haya ningún cliente repetido recorriendo el individuo y en el caso de que lo hubiese se eliminará y se sustituirá por otro cliente aleatorio.

```
def elimina_repetidos(individuo):
    for indice, cliente in enumerate(individuo):
        if (cliente > numClientes) or (cliente == 0):
            individuo[indice] = random.randint(1, numClientes)
        if individuo.count(cliente) > 1:
            repetido = True
            while (repetido):
                individuo[indice] = random.randint(1, numClientes)
                repetido = False
                for j, x in enumerate(individuo):
                    if individuo[indice] == x and j != indice:
                        repetido = True
    return individuo
```

2.6. repartir

El objetivo es dado la descendencia distribuirla por el resto de las anemonas de forma aleatoria, como variables de entrada además de la descendencia y la lista de anemonas, recibe la anemona en la cual se ha creado la descendencia y el número de anemonas totales.

```
def repartir(descendencia,anemonas,anemona_elegida,Na):
    for i in descendencia:
        ind=random.randint(0, Na-1)
        if(anemonas[ind]!=anemona_elegida) and len(anemonas[ind])<7:
            anemonas[ind].append(i)
    return anemonas
```

2.7. rand_U

Esta función se crea por comodidad, devuelve un número entre 0.001 y 0.999 llamando a la función *random.uniform.*

```
def rand_U():
    return random.uniform(0.001, 0.999)
```

2.8. elige_desp

Función cuyo objetivo es elegir el desplazamiento a aplicar a la población, para ello recibe un número entre 0.001 y 0.999 y calcula el valor de la exponencial decreciente, para ello se llama a la función *pow* de la librería *math*, dependiendo de ese valor se devolverá un 1, un 2 o un 3.

```
def elige_desp(a):
    resul=pow(a,-1.0/3.0)
    if resul >2:
        desp=2
    elif resul>1:
        desp=1
    else:
        desp=3
    return desp
```

2.9. desplazamiento

Esta función es la encargada de aplicar los operadores de búsqueda local descritos en la explicación del algoritmo, al pez seleccionado. Dependiendo de la variable de entrada *desp* se aplicará uno u otro.

```
def desplazamiento(desp,pez,toolbox):
    pez_mov=toolbox.clone(pez)
    busqueda_local=[]
    if desp==3:
        for i in range(0,numClientes):

busqueda_local.append(creator.Individual(random.sample(pez_mov,len(pez_mov))))
            busqueda_local[i]=elimina_repetidos(busqueda_local[i])
        fitnesses=toolbox.map(toolbox.evaluate, busqueda_local)
        pez_mov=best(list(fitnesses),busqueda_local)
        fit=toolbox.evaluate(pez_mov)
        pez_mov.fitness.values=fit
    elif desp==2:
        for cliente in pez_mov[0:int(len(pez_mov)/2)]:
            busqueda_local.append(cliente)
        for cliente in pez_mov[-1:int(len(pez_mov)/2-1):-1]:
            busqueda_local.append(cliente)
        pez_mov=creator.Individual(busqueda_local)
        fit=toolbox.evaluate(pez_mov)
        pez_mov.fitness.values=fit
    else:
        for cliente in pez_mov[0:int(len(pez_mov)/4)]:
            busqueda_local.append(cliente)
        for cliente in pez_mov[int(len(pez_mov)/2):int(len(pez_mov)/4)-1:-1]:
            busqueda_local.append(cliente)
```

```
        for cliente in pez_mov[:int(len(pez_mov)/2):-1]:
            busqueda_local.append(cliente)
        pez_mov=creator.Individual(busqueda_local)
        fit=toolbox.evaluate(pez_mov)
        pez_mov.fitness.values=fit
    return pez_mov
```

2.10. transformación

En el proceso de desplazamiento puede ser que el macho o la hembra mueran, en el caso de que esto suceda, si la que muere es la hembra el macho se transformará en hembra y en el caso de que haya cadetes el más apto pasará a ser el macho, si no hay cadetes el macho quedará viudo a la espera de que llegue un cadete, en el caso de que muera el macho el cadete más apto pasará a ser el macho.

```
def transformacion(anemonas,anemona,pez,toolbox):
    for i in range(0,len(anemonas)):
        if anemonas[i]==anemona:
            for j in range(0,len(i)):
                #Si hay una pareja más cadetes en la anémona
                if len(i)>2:
                    #Si muere la hembra
                    if j==0 and anemona[j]==pez:
                        anemona[j]=anemona[j+1]

fitnesses=toolbox.map(toolbox.evaluate,anemona[j+1:])
                        anemona[j+1]=best(fitnesses,anemona[j+1:])
                        for aux in range(j+2,len(anemona)):
                            if anemona[aux]==anemona[1]:
                                anemona.pop(aux)
                                break
                    #Si muere el macho
                    elif j==1 and anemona[j]==pez:

fitnesses=toolbox.map(toolbox.evaluate,anemona[j:])
                        anemona[j]=best(fitnesses,anemona[j:])
                        for aux in range(j+1,len(anemona)):
                            if anemona[aux]==anemona[1]:
                                anemona.pop(aux)
                                break
                #Si solo hay una pareja
                else:
```

```
                    anemona.remove(pez)
            anemonas[i]=anemona
    return anemonas
```

2.11. sustituye

En el caso de que el desplazamiento sea exitoso y el pez no muera, será necesario sustituir al individuo que era antes del desplazamiento, para eso se crea esta función, que sustituye en la anemona correspondiente al anterior pez con el nuevo.

```
def sustituye(anemonas,anemona,pez,indv):
    for i in range(0,len(anemonas)):
        if anemonas[i]==anemona:
            for j in range(0,len(anemonas[i])):
                if anemona[j]==pez:
                    anemona[j]=indv
                    anemonas[i]=anemona
    return anemonas
```

2.12. comprueba_muerte

Función encargada de comprobar si se ha producido la muerte de un pez.

```
def comprueba_muerte(bestFitness,FitnessAct,Nmov,Niter,toolbox):
    global fallos_globales
    global fallos_locales
    global FitAnt
    muerte = False
    string_best='%s' % toolbox.evaluate(bestFitness)
    mejorFitness=float(string_best)
    if(FitnessAct<mejorFitness):
        fallos_globales+=1
        if(fallos_globales>=Niter):
            muerte=True
            fallos_globales=0
    if(FitAnt>FitnessAct):
        FitAnt = FitnessAct
        fallos_locales+=1
        if(fallos_locales>=Nmov):
            muerte=True
            fallos_locales=0

    return muerte
```

2.13. algoritmo

Función encargada de implementar el algoritmo NEMO, tras su ejecución devuelve la lista que contiene a las anemonas, una lista que contiene a los mejores individuos y el mejor individuo de todos.

```
def algoritmo(toolbox, Na, Nmov, Niter,cxpb,numGen,verbose):
    anemonas=crea_anemonas(toolbox,Na)
    mejores=[]
    for j in anemonas:
        invalid_ind = [ind for ind in j if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        mejores.append(best(list(fitnesses),j))
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit
    invalid_ind=[ind for ind in mejores if not ind.fitness.valid]
    fitnesses=toolbox.map(toolbox.evaluate, invalid_ind)
    for ind,fit in zip(invalid_ind,fitnesses):
        ind.fitness.values=fit
    mejor=actualiza_mejor(mejores,toolbox,0.0,[])
    for gen in range(1,numGen+1):
        val=random.random()
        if(val<cxpb):
            elegida=random.choice(anemonas)
            while(len(elegida)<2):
                elegida = random.choice(anemonas)
            descendencia=reproduccion(elegida,toolbox,5,cxpb)
            anemonas=repartir(descendencia,anemonas,elegida,Na)

        else:
            anemona=random.choice(anemonas)
            if len(anemona) >= 2:
                for i in range(0,2):
                    pez=anemona[i]
                    desp=elige_desp(rand_U())
                    indv=desplazamiento(desp,pez,toolbox)
                    TupAux=toolbox.evaluate(indv)
                    string_aux = '%s' % TupAux

muerte=comprueba_muerte(mejor,float(string_aux),Nmov,Niter,toolbox)
                    if muerte:
                        transformacion(anemonas,anemona,pez,toolbox)
                    else:
                        sustituye(anemonas,anemona,pez,indv)
        mejores=[]
        for j in anemonas:
            for indice in range(0,len(j)):
                j[indice] = elimina_repetidos(j[indice])
        for j in anemonas:
            invalid_ind = [ind for ind in j if not ind.fitness.valid]
            fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
            for ind, fit in zip(invalid_ind, fitnesses):
                ind.fitness.values = fit
            mejores.append(best(list(fitnesses), j))
        invalid_ind = [ind for ind in mejores if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit
        string_aux='%s' % mejor.fitness.values
```

```
        mejor = actualiza_mejor(mejores, toolbox,float(string_aux),mejor)
        # Append the current generation statistics to the logbook
        if verbose:
            print(mejor)
    return anemonas,mejores,mejor
```

2.14. main

En esta función se llaman al resto de funciones y se realizan los preparativos necesarios para el correcto funcionamiento del algoritmo, como registrar las distintas funciones en el *toolbox*, y tras la ejecución se calculan las estadísticas de la población y se pasan los resultados a un fichero de texto.

```
def main(datos, coste_uni, coste_ini , wait_cost, delayCost, indTam, Na,Nmov,Niter,numGen):
    leedatos(datos, 'prueba.txt', indTam)
    creator.create('FitnessMax', base.Fitness, weights=(1.0,))
    creator.create('Individual', list, fitness=creator.FitnessMax)
    toolbox = base.Toolbox()
    toolbox.register('indexes', random.sample, range(1, indTam + 1), indTam)
    # Iniciamos las estructuras
    toolbox.register('individual', tools.initIterate, creator.Individual, toolbox.indexes)
    toolbox.register('population', tools.initRepeat, list, toolbox.individual)
    toolbox.register('evaluate', evaluacion, datos=datos, coste_uni=coste_uni, coste_ini=coste_ini, wait_cost=wait_cost,
                     delayCost=delayCost)
    toolbox.register("mate", tools.cxTwoPoint)
    toolbox.register("select", tools.selTournament, tournsize=4)
    anemonas,mejores,mejor=algoritmo(toolbox=toolbox, Na=Na, Nmov=Nmov, Niter=Niter, cxpb=0.6,numGen=numGen,verbose=True)
    fitness=[]
    for anemona in anemonas:
        for individuo in anemona:
            string_aux = '%s' % individuo.fitness.values
            fitness.append(float(string_aux))
    media=np.mean(fitness)
    std=np.std(fitness)
    min=np.min(fitness)
    max=np.max(fitness)
    print("Estadisticas:"+'\n'+ "Media: %f" %media+'\n'+"Desviación estandar: %f" % std+'\n'+"Mínimo: %f" % min+'\n'+"Máximo: %f" % max)
```

```
    print(1/max)
    print(mejor)
    f=open('ResultadosNEMO.txt', "w")
    f.write("******RESULTADOS******\n")
    f.write("Ruta = ")
    subruta=decodifica(mejor,datos)
    for ruta in subruta:
        f.write(str(ruta)+"\n")
    f.write("Coste = " + str(1/max)+"\n")
    f.write("Media coste = " + str(1/media)+"\n")
    f.write("Desviación estandar = "+str(std))
    f.close()
```

3 Algoritmo génetico aplicado al problema de Steiner

En este apartado se explicará las funciones utilizadas para la resolución del problema de Steiner utilizando un algoritmo génetico simple.

3.1. crea_nuevos_pesos

Función encargada de generar nuevos costes aleatorios para las aristas, y así poder generar nuevas soluciones válidas usando el algoritmo de Kruskal. Para ello recorremos la lista que contiene las aristas, en la que cada elemento es una lista en la que el primer elemento es el peso de la arista y los otros dos componentes son los nodos que forman la arista, y cambiamos el primer elemento por un número aleatorio que irá desde 1 hasta el máximo de coste de los pesos.

```
def crea_nuevos_pesos(datosleidos):
    for edge in datosleidos['edges']:
        edge[0] = random.randint(1,max(costes))
```

3.2. genera_soluciones

En esta función se llamará a la función *kruskal* que proporcionará soluciones del problema de Steiner, cuando obtengamos una solución comprobaremos si la solución es válida usando la función *comprueba_solucion_kruskal*, si la solución es válida se añade a la lista de soluciones y se generan nuevos pesos para las aristas, este proceso se realizará hasta que el tamaño de la lista de soluciones sea igual a la variable de entrada *size*, finalmente se devuelve la lista de soluciones.

```
def genera_soluciones(datosleidos,size):
    soluciones = []
    nuevos_datos = datosleidos
    while(len(soluciones)<size):
        solucion = kruskal(nuevos_datos)
        if comprueba_solucion_kruskal(solucion,nuevos_datos):
            soluciones.append(solucion)
            crea_nuevos_pesos(nuevos_datos)
```

```
    return soluciones
```

3.3. comprueba_soluciones

El objetivo de esta función es comprobar que la nueva solución que proponen los individuos sea válida, esto se realiza creando una lista vacía a la que se le añaden las aristas de la solución dependiendo de si el elemento del individuo es 1, una vez se tiene esa lista completa se recorre y se comprueba que aparezcan todos los nodos terminales en ella, si es así el valor de retorno será *True* y en caso contrario *False*.

```
def comprueba_solucion(solucion,individuo,datosfichero):
    nueva_solucion = []
    nodos = [];
    for i in range(0,len(individuo)):
        if individuo[i] == 1:
            nueva_solucion.append(solucion[i])
    for i in nueva_solucion:
        nodos.append(i[1])
        nodos.append(i[2])
    for term in datosfichero['terminales']:
        if term not in nodos:
            return False
    return True
```

3.4. comprueba_soluciones_kruskal

El objetivo de esta función es similar al anterior, pero en vez de comprobar la solución propuesta por el individuo, se asegura que la solución obtenida usando el algoritmo de Kruskal contenga todos los terminales, y devolverá *True* si así es o *False* si no se cumple esto.

```
def comprueba_solucion_kruskal(solucion,datosfichero):
    TerminalesInSol = 0
    for i in solucion:
        for term in datosfichero['terminales']:
            if term == i[1] or term == i[2]:
                TerminalesInSol += 1
    if TerminalesInSol >= numTerminals:
        return True
    else:
        return False
```

3.5. imprime_solucion

Esta función devuelve una lista que contiene las aristas de la solución final según la codificación del individuo que recibe.

```
def imprime_sol(solucion, indv):
    solucion_final = []
    for i in range(0, len(indv)):
        if indv[i] == 1:
            solucion_final.append(solucion[i])
    return solucion_final
```

3.6. lee_datos

Función destinada a leer los datos del fichero y almacenarlos en un diccionario, el estilo es similar a las versiones mostradas anteriormente.

```
def leedatos(datosfichero,nombrefichero):
    leido = 0
    terminales = 0
    nodos = 0
    global numTerminals
    global matriz_costes
    global costes
    with open(nombrefichero) as f:
        for numLin,linea in enumerate(f,start=1):
            if numLin in [1,2,3,4,5,6,7,8]:
                pass
            else:
                aux=linea.strip().split()
                try:
                    if aux[0] == "Edges":
                        empiezaLeer = numLin+1
                        aux=linea.strip().split()
                        numEdges = int(aux[1])
                        leido = 1
                    elif aux[0] == "Terminals":
                        empiezaLeer = numLin+1
                        aux = linea.strip().split()
                        numTerminals = int(aux[1])
                        terminales = 1
                        leido = 0
                    elif aux[0] == "Nodes":
                        numNodos = int(aux[1])
```

```
                        if nodos == 0:
                            for i in range(1,numNodos+1):
                             datosfichero["vertices"].append(str(i))
                             nodos = 1


                except:
                    pass
                if leido == 1:
                    if numLin >= empiezaLeer and numLin < empiezaLeer+numEdges:
                        aux = linea.strip().split()
                        Laux = [int(aux[3]),aux[1],aux[2]]
                        costes.append(int(aux[3]))
                        datosfichero['edges'].append(Laux)
                if terminales == 1:
                    if numLin >= empiezaLeer and numLin < empiezaLeer+numTerminals:
                        aux = linea.strip().split()
                        datosfichero['terminales'].append(aux[1])
    for i in graph['edges']:
        index1 = int(i[1])
        index2 = int(i[2])
        matriz_costes[index1][index2] = i[0]
```

3.7. evaluacion

La función de evaluación, el fitness será la diferencia entre el coste total del árbol y el coste de la solución original dada por el algoritmo de Kruskal.

```
def evaluacion(indv, solucion):
    coste_orig = sum(costes)
    coste_sol = 0
    for edge, elemento in zip(solucion, indv):
        index1 = int(edge[1])
        index2 = int(edge[2])
        coste_sol += elemento*matriz_costes[index1][index2]
    fitness = coste_orig-coste_sol
    return fitness,
```

3.8. eaSimple

Se trata del algoritmo implementado en DEAP, al que se le ha añadido unas modificaciones para garantizar la validez de las soluciones.

```
if comprueba_solucion(soluciones[i],datos,offspring[i]) == False:
                nuevo_individuo=crea_individuos(len(offspring[i]))

while(comprueba_solucion(soluciones[i],datos,nuevo_individuo)        ==
False):
                    nuevo_individuo                                    =
crea_individuos(len(offspring[i]))
                offspring[i]=creator.Individual(nuevo_individuo)
```

3.9. main

Función donde se configuran los objetos de DEAP y se llama las funciones que generan las soluciones y ejecutan el algoritmo, posteriormente se graban los resultados en un fichero de texto.

```
def main(datos, sizePob, nombrefichero):
    leedatos(datos, nombrefichero)
    creator.create('FitnessMax', base.Fitness, weights=(1.0,))
    creator.create('Individual', list, fitness=creator.FitnessMax)
    toolbox = base.Toolbox()
    # Generador de atributos
    toolbox.register('indexes', random.randint, 0, 1)
    # Iniciamos las estructuras
    soluciones = genera_soluciones(datos, sizePob)
    toolbox.register('individual',                    tools.initRepeat,
creator.Individual, toolbox.indexes,len(soluciones[0]))
    toolbox.register('population',        tools.initRepeat,        list,
toolbox.individual)
    toolbox.register('evaluate', evaluacion)
    toolbox.register("mate", tools.cxTwoPoint)
    toolbox.register("mutate", tools.mutUniformInt, low=0, up=100,
                     indpb=0.2)   # Independent probability  : for each
attribute to be mutated.# low~up rondom int
    toolbox.register("select", tools.selTournament, tournsize=3)
    pop = toolbox.population(n=sizePob)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", np.mean)
    stats.register("std", np.std)
    stats.register("min", np.min)
```

```
    stats.register("max", np.max)
    pop, log = eaSimple(pop, toolbox, cxpb=0.85, mutpb=0.01, ngen=100,
stats=stats, halloffame=hof,
                        verbose=True,
soluciones=soluciones,datos=datos)


    fitness_mejor = 0
    for indv in pop:
        for solucion in soluciones:
            if comprueba_solucion(solucion,datos,indv) == True:
                fit = "%s"%toolbox.evaluate(indv,solucion)
                if float(fit) > fitness_mejor:
                    mejor = indv
                    fitness_mejor = float(fit)
                    sol_final = solucion



    sol = imprime_sol(sol_final,mejor)
    print(sol)
    f = open('ResultadosSteiner.txt', "w")
    f.write("******RESULTADOS******\n")
    f.write("Solucion=")
    aux = []
    for elemento in sol:
        ind1 = int(elemento[1])
        ind2 = int(elemento[2])
        aux.append(matriz_costes[ind1][ind2])
        straux = "%s" % matriz_costes[ind1][ind2]
        f.write(straux + " " + elemento[1] + " " + elemento[2])
        f.write("\n")
    coste = sum(costes)-fitness_mejor
    if coste == sum(aux):
        print("Flaama")
    f.write("Coste = " + str(coste) + "\n")
    media = log.select('avg')
    estd = log.select("std")
    f.write("Media coste = " + str(sum(costes)-media[-1]) + "\n")
    f.write("Desviación estandar = " + str(estd[-1]) + "\n")
    f.close()
```

4 Algoritmo NERO aplicado al problema de Steiner

Se detallarán las funciones que difieran o se hayan modificado para la resolución del problema de Steiner con

respecto al VRPTW.

4.1. desplazamiento

Se ha cambiado la comprobación de la validez de las soluciones.

```
def desplazamiento(desp,pez,toolbox, anemona_pez, solcuiones,datos):
    pez_mov = toolbox.clone(pez)
    busqueda_local = []
    fitness = []
    if desp == 3:
        for i in range(0,len(pez)):
            aux = creator.Individual(random.sample(pez_mov,len(pez_mov)))
            buena = False
            for solucion in solcuiones:
                if comprueba_solucion(solucion, aux, datos) == True:
                    buena = True
                    break
            if buena:
                busqueda_local.append(aux)
        for solucion,indv in zip(solcuiones,busqueda_local):
            if comprueba_solucion(solucion, indv, datos) == True:
                fitness.append(toolbox.evaluate(indv,solucion))
                break
        pez_mov = best(fitness,busqueda_local)
        pez_mov.fitness.values = max(fitness)
    elif desp == 2:
        for cliente in pez_mov[0:int(len(pez_mov)/2)]:
            busqueda_local.append(cliente)
        for cliente in pez_mov[-1:int(len(pez_mov)/2-1):-1]:
            busqueda_local.append(cliente)
        pez_mov=creator.Individual(busqueda_local)
        for solucion in solcuiones:
            if comprueba_solucion(solucion, pez_mov, datos) == True:
                fit = toolbox.evaluate(pez_mov,solucion)
                break
            else:
                fit = 0,
        pez_mov.fitness.values = fit
    else:
        for cliente in pez_mov[0:int(len(pez_mov)/4)]:
            busqueda_local.append(cliente)
        for cliente in pez_mov[int(len(pez_mov)/2):int(len(pez_mov)/4)-1:-
1]:
            busqueda_local.append(cliente)
        for cliente in pez_mov[:int(len(pez_mov)/2):-1]:
            busqueda_local.append(cliente)
        pez_mov = creator.Individual(busqueda_local)
        for solucion in solcuiones:
            if comprueba_solucion(solucion, pez_mov, datos) == True:
                fit = toolbox.evaluate(pez_mov,solucion)
                break
            else:
                fit = 0,
        pez_mov.fitness.values = fit
    return pez_mov
```

4.2. reproduccion

Similar a la versión utilizada para el VRPTW, pero antes de devolver a la descendencia se comprueba si forman una solución valida y en caso contrario se eliminan de esta.

```
def reproduccion(anemona,toolbox,TamDesc,cxpb,soluciones,datos):
    descendencia = []
    for i in range(1,TamDesc+1):
        if random.random() < cxpb:
            indv0, indv1 = toolbox.mate(anemona[0],anemona[1])
            if i%2 == 0:
                descendencia.append(indv0)
            else:
                descendencia.append(indv1)
    for hijo in descendencia:
        buena = False
        for solucion in soluciones:
            if comprueba_solucion(solucion, hijo, datos) == True:
                buena = True
                break
        if buena == False:
            descendencia.remove(hijo)

    return descendencia
```

4.3. algoritmo

El cuerpo del algoritmo sigue siendo el mismo, pero se le añaden ciertos cambios destinados a la asignación del fitness y la comprobación de la validez de las soluciones generadas por los individuos.

```
def algoritmo(toolbox, Na, Nmov, Niter,cxpb,numGen,verbose,soluciones,
datos):
    anemonas = crea_anemonas(toolbox,Na)
    lista_eval = []
    mejores = []
    for anemona in anemonas:
        for individuo in anemona:
            lista_eval.append(individuo)
    fitnesses = list(toolbox.map(toolbox.evaluate, lista_eval, soluciones))
    j = 0
    for anemona in anemonas:
        i = 0
        for individuo in anemona:
            individuo.fitness.values=fitnesses[i+j]
            i += 1
        j += len(anemona)

    for gen in range(1,numGen+1):
        val=random.random()
        if(val<cxpb):
            elegida=random.choice(anemonas)
            while(len(elegida)<2):
                elegida = random.choice(anemonas)
            descendencia=reproduccion(elegida,toolbox,5,cxpb, soluciones,
datos)
            anemonas=repartir(descendencia,anemonas,elegida,Na)

        else:
            anemona=random.choice(anemonas)
            if len(anemona) >=2:
```

```
                for i in range(0,2):
                    try:
                        pez=anemona[i]
                    except:
                        pass
                    desp=elige_desp(rand_U())
                    indv=desplazamiento(desp,pez,toolbox,
anemona,soluciones,datos)
                    for solucion in soluciones:
                        if comprueba_solucion(solucion,indv,datos) == True:
                            TupAux=toolbox.evaluate(indv,solucion)
                            string_aux = '%s' % TupAux
                            buena = True
                            break
                        else:
                            buena = False
                    if buena:
                        muerte =
comprueba_muerte(float(string_aux),Nmov,Niter)
                    else:
                        muerte = True
                    if muerte:

transformacion(anemonas,anemona,pez,toolbox,soluciones)
                    else:
                        sustituye(anemonas,anemona,pez,indv)
        lista_eval = []
        for anemona in anemonas:
            for individuo in anemona:
                lista_eval.append(individuo)
        fitnesses = list(toolbox.map(toolbox.evaluate, lista_eval,
soluciones))
        j = 0
        for anemona in anemonas:
            i = 0
            for individuo in anemona:
                individuo.fitness.values = fitnesses[i + j]
                i += 1
            j += len(anemona)
        actualiza_mejor(lista_eval, toolbox,soluciones,datos)
        mejores.append(mejor)
        # Append the current generation statistics to the logbook
        if verbose:
            print(mejor)
    return anemonas,mejores,mejor
```