Facultad
de
Ciencias

UN ALGORITMO MEMÉTICO PARA EL PROBLEMA DE RUTAS DE VEHÍCULOS CON RESTRICCIÓN DE CAPACIDAD

(A memetic algorithm for the capacitated vehicle routing problem)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Daniel Sebastián San Martín

Directora: Inés González Rodríguez

Junio - 2019

Resumen

Un problema central en el ámbito del transporte y la logística es encontrar rutas de mínimo coste bajo ciertas restricciones. En los últimos años, este tipo de problemas ha captado la atención de los investigadores en ciencias de la computación por dos razones. La primera de ellas es la gran cantidad de aplicaciones en la vida real que tienen este tipo de problemas, que van desde la distribución de puntos de recarga para vehículos eléctricos hasta el reparto de paquetes usando camiones y drones. La segunda razón es que este tipo de problemas pertenecen a la clase de complejidad `NP-Hard`. Dado que únicamente es posible resolver este tipo de problemas de forma exacta en tiempo exponencial, nuestro objetivo es encontrar un algoritmo aproximado que obtenga soluciones cercanas al óptimo en tiempo razonable.

En este trabajo, desarrollaremos un algoritmo memético competitivo para el "Problema de rutas de vehículos con capacidad" (`CVRP`). Este algoritmo combina un algoritmo genético con una búsqueda local, denominada búsqueda tabú granular, e incorpora dos heurísticos voraces para proporcionar semillas a la población inicial. Además, utiliza varios mecanismos de control de diversidad.

Se ha evaluado la calidad del algoritmo desarrollado a través de la ejecución de instancias de uso común en la literatura, para las cuales se han obtenido soluciones cercanas al óptimo. Por otro lado, se ha realizado un estudio de sinergia entre las distintas componentes que constituyen el algoritmo memético para comprobar si su rendimiento es mayor de forma conjunta que por separado.

Palabras clave: Problema de rutas de vehículos con capacidad, Algoritmo genético, Búsqueda tabú granular, Algoritmo memético, Búsqueda metaheurística, Optimización combinatoria

Abstract

One of the central problems in the area of transportation and logistics is to find minimum cost routes under certain restrictions. In recent years this kind of problems has caught the attention of computer science researchers for two reasons. The first one is the huge number of real life applications that have this kind of problems, ranging from the distribution of charging points for electric vehicles to the delivery of packages using trucks and drones. The second reason is that this kind of problems belongs to the complexity class `NP-Hard`. Since solving this kind of problems using exact methods is only possible in exponential time, our goal is to obtain an approximate algorithm that yields solutions close to the optimum within reasonable time.

In this work, we develop a competitive memetic algorithm for the "Capacitated vehicle routing problem" (`CVRP`). This algorithm combines a genetic algorithm with a local search, called granular tabu search, and uses two greedy heuristic algorithms to seed the initial population. In addition, it incorporates several diversity-control mechanisms.

The quality of the developed algorithm has been evaluated using commonly used benchmarks, obtaining in most of the cases solutions close to the optimum. Besides, a synergy studied has been carried out to discover the interactions among the different components of the memetic algorithm.

Keywords: Capacitated vehicle routing problem, Genetic algorithm, Granular tabu search, Memetic algorithm, Metaheuristic search, Combinatorial optimization

Índice

CAPÍTULO 1

Introducción

> The salesman knows nothing of what he is selling save that he is charging a great deal too much for it. [1]
>
> **Oscar Wilde**

Índice

El objetivo de este primer capítulo del Trabajo de Fin de Grado (TFG) es mostrar a los lectores la motivación (1.1) para su elaboración así como las metas que se pretenden alcanzar (1.2). Al final del capítulo, se incluye una sucinta descripción del contenido de los capítulos y las secciones que constituyen esta memoria (1.3).

1.1. Motivación

En la década de 1940, el matemático estadounidense Merrill M. Flood [**F, 56**] planteó por primera vez, cuando se disponía a estudiar la organización del sistema de transporte escolar de New Jersey, un problema que en la actualidad desempeña un papel central en los ámbitos de la logística, la optimización combinatoria y la programación entera, el "Travelling Salesman Problem" (`TSP`).

La generalización natural del `TSP` se denomina "Vehicle Routing Problem" (`VRP`), y fue introducida por Dantzig & Ramser [**DR, 59**] en 1959. Desde entonces, ha existido un gran interés en la resolución de problemas de enrutamiento de vehículos. El `VRP` presenta muchas versiones, en función del tipo de restricciones que se imponen. Un trabajo que expone las distintas versiones del `VRP` y los últimos avances producidos en cada una de ellas es [**ITV, 14**].

En este trabajo, vamos a tratar el "Capacitated Vehicle Routing Problem" (`CVRP`), que impone restricciones de capacidad a los vehículos. El `CVRP` puede definirse mediante un grafo completo no dirigido $G := (V, E)$, donde $V := \{0, 1, \ldots, n\}$ es el conjunto de nodos y E el conjunto de aristas, en este caso con $|E| := \frac{n(n+1)}{2}$. El nodo 0 representa al depósito, que cuenta al principio del problema con k vehículos idénticos de capacidad Q. Para este trabajo, el número k de vehículos no es fijo, es una variable más. El resto de nodos, $0 < i \leq n$, representan a los clientes, y cada uno de ellos cuenta con una demanda no negativa $q(i)$. Cada una de las aristas $(i, j) \in E$ tiene asociado un coste de transporte no negativo $c(i, j) := c(j, i)$. El objetivo del `CVRP` es determinar un conjunto de k rutas que minimice el coste de forma que cada ruta comience y termine en el depósito, cada cliente sea visitado una única vez y que la demanda que satisface cada

[1] El vendedor no sabe nada sobre lo que está vendiendo, salvo que está cobrando demasiado por ello.

vehículo no exceda Q. En la Figura 1.1 se muestra un ejemplo de un grafo completo para un VRP con 15 clientes, donde el depósito aparece en rojo.

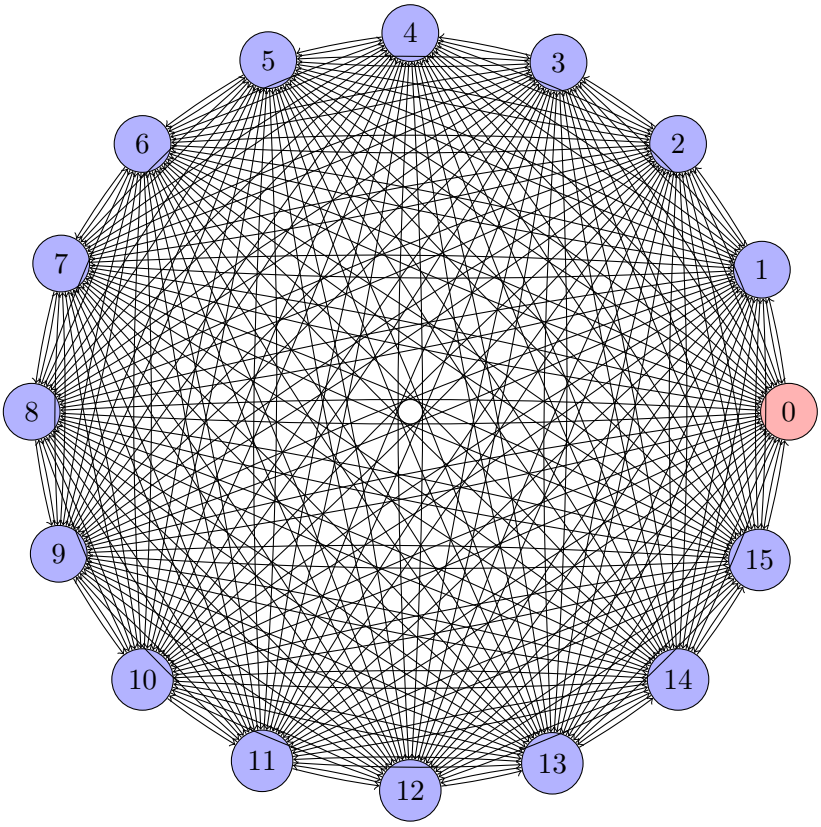

FIGURA 1.1. Grafo completo con 16 nodos y 120 aristas.

A continuación, pasamos a exponer por qué este tipo de problemas son interesantes. En primer lugar, cabe destacar que el VRP se encuentra en la clase de complejidad NP-Hard, lo que indica que este problema por su propia naturaleza constituye un gran reto para las matemáticas y la informática moderna. Es importante destacar, que en la actualidad hay instancias con 75 clientes que no han sido resueltas de forma óptima.

En segundo lugar, distintas versiones del VRP permiten modelar una gran cantidad de problemas que aparecen en la práctica [**CAGRA, 15**], que van desde la organización de itinerarios de empresas de reparto que utilizan drones [**DHMM, 16**] hasta la distribución de puntos de recarga para vehículos eléctricos [**LCHCL, 14**]. Durante los últimos años, se han publicado multitud de artículos acerca de este tema en importantes revistas de investigación operativa, ciencias del transporte y búsqueda metaheurística. Entre las revistas en las que se publican artículos relacionados con la temática de este trabajo podemos destacar: "Transportation Science", "ACM Computing Surveys", "Expert Systems with Applications", "Computers & Operations Research", "European Journal of Operational Research", "EURO Journal on Transportation and Logistics", "Applied Soft Computing", "Journal of the Operational Research Society", "Transportation Research", "Networks" y "Journal of Heuristic". Por otro lado, cabe mencionar, que en la práctica totalidad de los congresos sobre investigación operativa, ciencias del transporte o búsqueda metaheurística siempre hay una conferencia acerca de los últimos avances en problemas relacionados con el VRP.

1.2. Objetivos

El principal objetivo de este TFG es desarrollar un algoritmo competitivo para el CVRP. Cabe destacar que existen dos tipos de enfoques para resolver este tipo de problemas, los algoritmos exactos y los algoritmos aproximados.

Los algoritmos exactos son aquellos algoritmos que resuelven los problemas de optimización de forma óptima. El principal inconveniente que presentan los algoritmos exactos es que, a menos que la conjetura de Cook sea cierta y, por lo tanto, P = NP, no va a ser posible obtener algoritmos de este tipo con complejidad polinomial. La investigación en el ámbito de los algoritmos exactos se centra en encontrar algoritmos de este tipo cuya complejidad sea exponencial pero con base pequeña. Actualmente los algoritmos exactos presentan dificultades al resolver instancias con más de 75 clientes. En [**STV, 14**] aparece una recopilación de los métodos exactos que se emplean para resolver el CVRP. A día de hoy, el trabajo donde se obtienen los mejores resultados para el CVRP usando un algoritmo exacto es el elaborado por Pecin, Pessoa, Poggi & Uchoa [**PPPU, 17**], donde los autores combinan las técnicas *Cut separation* y *Column generation*.

Los algoritmos aproximados aparecen debido al gran coste computacional que presentan los métodos exactos y su objetivo es obtener soluciones cercanas al óptimo usando tiempos de cómputo razonables. La mayor parte de los algoritmos aproximados emplean técnicas de la inteligencia artificial, heurísticas y metaheurísticas. En cuanto al CVRP, podemos agrupar las técnicas empleadas en tres clases:

- **Búsquedas locales:** incluyendo simulated annealing, deterministic annealing y búsquedas tabú.
- **Algoritmos genéticos.**
- **Algoritmos de aprendizaje:** redes neuronales, clustering, etcétera.

En [**LPPMS, 16**] encontramos una recopilación de los métodos aproximados más utilizados para el CVRP, donde se explican las fortalezas y los problemas de cada uno de ellos.

En este trabajo, vamos a desarrollar un algoritmo aproximado, en el que combinaremos un **algorítmo genético** con una **búsqueda local**. Esta idea está inspirada en el trabajo de Prins [**P, 04**], donde el autor elabora un algoritmo genético en el que la fase de mutación consiste en una búsqueda local. Siguiendo el trabajo de Prins, nuestro algoritmo va a adoptar un enfoque "ordena primero, agrupa después" y va a incorporar heurísticos voraces para generar soluciones de calidad en la población inicial. En cuanto a las diferencias con el trabajo de Prins, el algoritmo desarrollado para este TFG utiliza mecanismos de control de diversidad para el algoritmo genético distintos de los empleados por Prins, usa la búsqueda local como un proceso de intensificación de las soluciones obtenidas por el algoritmo genético y no como un operador de mutación, emplea una versión de la búsqueda tabú granular de Toth & Vigo [**TV, 03**] como búsqueda local y elimina las restricciones en el número de vehículos y en las longitudes de las rutas que aparecen en el algoritmo de Prins.

La combinación de algoritmos evolutivos con búsquedas locales no es nueva, fue introducida en 1989 por Moscato [**M, 89**] y recibe el nombre de algoritmo memético. Por lo tanto, podemos decir que el objetivo de este TFG es el desarrollo de un **algoritmo memético** competitivo para el CVRP.

Una vez construido el algoritmo memético, realizaremos un estudio experimental que consta de dos partes. En la primera parte se llevará a cabo un estudio de sinergia

entre las distintas componentes del algoritmo memético. Por otro lado, en la segunda parte comprobaremos la calidad del algoritmo desarrollado a través de la ejecución de instancias de uso común en la literatura.

1.3. Estructura del documento

Este TFG se organiza en 6 capítulos como sigue:

El primer capítulo (1), que es en el que nos encontramos, está formado por tres secciones, que tratan respectivamente sobre la motivación (1.1), los objetivos (1.2) y la estructura del documento (1.3).

El segundo capítulo (2) contiene el marco teórico del trabajo y consta de 5 secciones. En la primera sección (2.1) se realiza una revisión bibliográfica de los métodos aproximados competitivos existentes para el `CVRP`. La segunda sección (2.2) se ocupa de los heurísticos voraces que generan semillas para los algoritmos de búsqueda. En la tercera sección (2.3) se detalla la configuración del algoritmo genético que vamos a emplear. La cuarta sección (2.4) se centra en las vecindades escogidas para las búsquedas locales y en la búsqueda tabú granular. En la quinta y última sección (2.5) se describe el algoritmo memético elaborado para este trabajo.

En el tercer capítulo (3) se recoge el análisis de requisitos (3.1, 3.2) del proyecto software realizado para este trabajo así como la metodología empleada (3.3). Además, se describen las herramientas y tecnologías usadas en el proyecto (3.4).

El cuarto capítulo (4) se ocupa del diseño y de la implementación del software construido en este trabajo. Consta de dos secciones, en la primera (4.1) se expone el diseño arquitectónico empleado mientras que en la segunda (4.2) se relaciona el diseño arquitectónico con el desarrollo del software.

El quinto capítulo (5) contiene los resultados de los experimentos que se han llevado a cabo para este trabajo y consta de tres secciones. En la primera se discute la elección de los parámetros (5.1) que se van a emplear en los experimentos. En la segunda sección, se realiza un estudio de sinergia (5.2) entre los métodos voraces de inicialización heurística, el algoritmo genético y la búsqueda tabú granular. Por último, en la tercera sección se exponen y analizan los resultados obtenidos al aplicar el algoritmo memético que hemos desarrollado a una colección de instancias de uso común el la literatura (5.3).

El sexto y último capítulo (6) contiene las conclusiones del TFG (6.1) y algunas posibles líneas de trabajo futuro (6.2).

CAPÍTULO 2

Algoritmos para el CVRP

> Une théorie, un système, une hypothèse, l'Évolution?... Non point: mais, bien plus que cela, une condition générale à laquelle doivent se plier et satisfaire désormais, pour être pensables et vrais, toutes les théories, toutes les hypothèses, tous les systèmes. Une lumiere eclairant tous les faits, une courbure que doivent épouser tous les traits: voilà ce qu'est l'Évolution. [1]
>
> **Pierre Teilhard de Chardin**

Índice

Este capítulo constituye la parte teórica del trabajo y está formado por 5 secciones. En la primera sección (2.1) se realiza una revisión de la literatura sobre los métodos de resolución aproximados competitivos para el `CVRP`. La segunda sección (2.2) se ocupa de los heurísticos voraces que generan semillas para los algoritmos de búsqueda. En la tercera sección (2.3) se expone la configuración del algoritmo genético que emplearemos en este trabajo, haciendo hincapié en el sistema de codificación/decodificación (2.3.1), en los operadores genéticos (2.3.2) y en los mecanismos de control de la diversidad (2.3.3). La cuarta sección (2.4) se centra en las búsquedas locales y consta de dos subsecciones, en la primera de ellas se describe la estructura de las vecindades que vamos a emplear (2.4.1) mientras que en la segunda se expone el esquema de la búsqueda tabú granular (2.4.2). En la última sección del capítulo (2.5) se describe el algoritmo memético elaborado en este trabajo.

2.1. Estado del Arte

Como ya se ha indicado en el Capítulo 1, en este trabajo nos vamos a centrar en las estrategias referentes a algoritmos aproximados. Respecto a este tipo de algoritmos, existe una extensa literatura y actualmente se utiliza un amplio abanico técnicas para enfrentarse al `CVRP`. En [**CLSV, 07**], [**LPPMS, 16**] y [**ZBB, 10**] podemos encontrar

[1] ¿ Es la evolución una teoría, un sistema o una hipótesis ? Es mucho más: es una condición general ante la cual todas las teorías, todas las hipótesis, todos los sistemas deben inclinarse y satisfacer a partir de ahora si pretenden ser razonables y verdaderas. La evolución es la luz que ilumina todas las verdades, una curva que todas las líneas deben seguir.

una recopilación de los métodos de inteligencia artificial y las técnicas heurísticas y metaheurísticas que se emplean para resolver el CVRP de forma aproximada.

El objetivo de este trabajo es desarrollar un algoritmo memético para el CVRP, que surge de la combinación de un algoritmo genético con una búsqueda local. En relación a los algoritmos meméticos, su esquema principal se presenta en [**M, 89**] y en este trabajo seguiremos las líneas propuestas en [**P, 04**]. A continuación, pasamos a analizar la literatura correspondiente a cada una de las dos componentes que constituyen el algoritmo memético.

Los algoritmos genéticos aparecen por primera vez en [**H, 75**]. Este tipo de algoritmos evolutivos se basan en la acción de una serie operadores (selección, cruce, mutación y reemplazo) sobre una población de posibles soluciones que van evolucionando. Los algoritmos genéticos, como la mayoría de los métodos basados en poblaciones, priman la diversificación sobre la intensificación, por lo que su rendimiento puede resentirse en espacios de búsqueda de gran amplitud como el del CVRP. Podemos encontrar más información acerca de este tipo de algoritmos en [**B, 03**]. Cabe destacar que en este trabajo se adopta un enfoque "ordena primero, agrupa después". Aunque durante muchos años se consideró poco atractivo, fue precisamente Prins quien lo integró por primera vez en una búsqueda metaheurística demostrando su efectividad. Desde entonces, se ha utilizado con éxito en múltiples ocasiones, como se demuestra en [**PLP, 14**]. Una ventaja de este enfoque es que el método metaheurístico explora un espacio de búsqueda considerablemente más pequeño, el de las "grandes rutas" que resuelven el TSP que resulta de relajar el problema original eliminando la restricción de capacidad de los vehículos. Es a la hora de evaluar estas grandes rutas cuando se realiza la partición o asignación a distintos vehículos incorporando las restricciones de capacidad, pero de tal forma que, dentro de cada vehículo, se respeten los órdenes relativos entre clientes fijados por la gran ruta. Aquí reside otra ventaja de este enfoque, ya que se puede realizar la partición garantizando que es óptima para dicha gran ruta y, viceversa, para una solución óptima del CVRP original existe al menos una gran ruta cuya partición coincide con la solución óptima. En resumen, este enfoque permite reducir considerablemente el espacio de búsqueda sin por ello perder información.

A continuación, pasamos a las búsquedas locales. Este tipo de búsquedas trazan una trayectoria, moviéndose de una solución a otra a partir de una solución inicial. En cada paso, se hallan soluciones cercanas a la solución actual S, según una estructura de vecindad, y se selecciona una de ellas. En este trabajo, vamos a emplear dos tipos de vecindades, las definidas por los movimientos `2-opt` y `2-opt*`. Las estructuras anteriores son casos particulares de los movimientos `k-opt` descritos en [**LK, 73**]. En la literatura, podemos encontrar otros tipos de movimientos como el `Or-opt` [**O, 76**] o el λ-intercambio de Osman [**O, 93**]. Uno de los esquemas de búsqueda más utilizados en la actualidad es la búsqueda tabú, que fue introducida por Glover en 1986 [**G, 86**]. En este trabajo, vamos a emplear como esquema de búsqueda una versión de la búsqueda tabú creada por Toth & Vigo en 2003 [**TV, 03**], denominada búsqueda tabú granular.

Finalizamos esta sección haciendo referencia a los heurísticos para el CVRP que hemos empleado en este trabajo con el objetivo de obtener individuos de calidad (semillas) para nuestra población inicial. En este caso, hemos utilizado los heurísticos Nearest Neighbor [**LPPMS, 16**] y de Clarke & Wright [**CW, 64**]. Una recopilación de este tipo de heurísticos para el CVRP puede encontrarse en [**LS, 02**].

2.2. Heurísticos Constructivos

El objetivo de esta sección es describir los dos heurísticos voraces que vamos a emplear en este trabajo.

Comenzamos con el heurístico Nearest Neighbor. Este heurístico es un algoritmo voraz que comienza en el depósito y construye una ruta de forma progresiva añadiendo el cliente más cercano que no se encuentra en ninguna otra ruta y que no viola las restricciones de capacidad del vehículo. Cuando no se pueden añadir más clientes a una ruta, el vehículo vuelve al depósito y se inicia una nueva ruta. La complejidad temporal de este algoritmo es $\mathcal{O}(n^2)$. El Algoritmo 1 contiene el pseudocódigo del heurístico Nearest Neighbor.

Algoritmo 1 Heurístico Nearest Neighbor

```
1: rutaActual := Iniciar ruta
2: nodosSinVisitar := nodos(Grafo)
3: mientras nodosSinVisitar es no vacío hacer
4:     nodoActual := últimoNodo(rutaActual)
5:     siguienteNodo := ∅; distanciaMínima := ∞
6:     para cada nodo ∈ nodosSinVisitar hacer
7:         si c(nodoActual, nodo) < distanciaMínima y cargaRutaActual+q(nodo) ≤
   Q entonces
8:             siguienteNodo := nodo
9:         fin si
10:    fin para cada
11:    si siguienteNodo = ∅ entonces
12:        Almacenar rutaAcutal
13:        rutaActual := Iniciar ruta
14:    si no
15:        rutaActual := rutaActual + siguienteNodo
16:        cargaRutaActual := cargaRutaActual + q(siguienteNodo)
17:        Eliminar siguienteNodo de nodosSinVisitar
18:    fin si
19: fin mientras
```

Por otro lado, el heurístico de Clarke & Wright comienza con una solución inicial formada por n rutas, una por cada cliente. En cada iteración, se evalúan las posibles concatenaciones de dos rutas y se lleva a cabo aquella concatenación que proporcione un mayor ahorro. Obviamente, solamente pueden concatenarse rutas siempre y cuando la suma de las respectivas demandas sea menor que Q. El proceso termina cuando se obtiene una sola ruta, después de $n-1$ concatenaciones, o cuando al unir cualesquiera de las rutas restantes, se produce una violación de las restricciones de capacidad. Cabe destacar, que en este contexto no se suele hablar de ahorro, y que es frecuente utilizar el término "coste variacional". Si tenemos dos rutas $R := \{r_0 := 0, r_1, r_2, \ldots, r_{|R|} := 0\}$ y $S := \{s_0 := 0, s_1, s_2, \ldots, s_{|S|} := 0\}$ y unimos la ruta R con la ruta S a través del cliente $r_{|R|-1}$, el coste variacional se define como $c(r_{|R|-1}, s_1) - c(r_{|R|-1}, 0) - c(0, s_1)$. Si la instancia es euclídea, los costes variacionales son siempre negativos. El heurístico contempla las 4 posibles formas distintas de unir dos rutas; usando las notaciones anteriores, las aristas que se pueden añadir son $(r_{|R|-1}, s_1)$, $(r_{|R|-1}, s_{|S|-1})$, (r_1, s_1) y $(r_1, s_{|S|-1})$.

Como veremos en la Sección 5.2, el heurístico de Clarke & Wright da buenos resultados en la práctica y es considerablemente mejor que el heurístico Nearest Neighbor. Nótese

que el heurístico de Clarke & Wright reduce en cada iteración el número de vehículos empleados en una unidad, por lo que podemos decir que este algoritmo tiende a minimizar tanto el coste como el número de vehículos. Una evaluación rápida de todas las posibles uniones en cada iteración nos lleva a obtener una complejidad de $\mathcal{O}(n^3)$. Sin embargo, podemos mejorar el valor anterior usando las estructuras de datos adecuadas, y teniendo en cuenta que cada concatenación puede codificarse mediante una arista. Las $\frac{n(n-1)}{2}$ aristas del grafo G que no tienen un extremo en el depósito pueden ordenarse al principio del algoritmo en una lista L, en orden creciente según su coste variacional asociado. Usando un algoritmo de ordenación por montículos, L puede obtenerse en $\mathcal{O}(n^2 log_2\, n)$. Una vez que tenemos L, el algoritmo comprueba cada una de las aristas (i, j) almacenadas. Si los clientes i y j se encuentran en los extremos de dos rutas distintas y la suma de las cargas de las respectivas rutas no viola las restricciones de capacidad, entonces se lleva a cabo la concatenación de las rutas. Por lo tanto, cada arista de L puede procesarse en $\mathcal{O}(1)$ y la complejidad del algoritmo viene determinada por la fase de ordenación, que en este caso es $\mathcal{O}(n^2 log_2\, n)$. El Algoritmo 2 contiene el pseudocódigo del heurístico de Clarke & Wright.

Algoritmo 2 Heurístico de Clarke & Wright

1: Crear n rutas, una para cada cliente
2: Calcular los costes variacionales
3: $L :=$ Lista de aristas del grafo ordenadas crecientemente según su coste variacional asociado
4: **mientras** L es no vacía **hacer**
5: $\quad e := (e_1, e_2) :=$ Obtener cima de L
6: $\quad r_1 :=$ Ruta que contiene a e_1
7: $\quad r_2 :=$ Ruta que contiene a e_2
8: $\quad$ **si** $r_1 \neq r_2$, e_1 es extremo de r_1, e_2 es extremo de r_2 y $q(r_1)+q(r_2) \leq$ Q **entonces**
9: $\quad\quad r :=$ Unir(r_1, r_2, e)
10: $\quad\quad$ Añadir la ruta r y eliminar las rutas r_1 y r_2
11: $\quad$ **fin si**
12: **fin mientras**

2.3. Algoritmo Genético

En esta sección, expondremos las características del algoritmo genético que sirve de base para el algoritmo memético objeto de este trabajo.

Los algoritmos genéticos fueron inventados en 1975 por John Holland [**H, 75**], investigador de la Universidad de Michigan. Este tipo de algoritmos se engloban dentro de la categoría de algoritmos de optimización, ya que su objetivo es encontrar la mejor solución de un problema entre un conjunto de soluciones posibles. El nombre de "genéticos" aparece porque los mecanismos de los que se valen estos algoritmos pueden verse como una metáfora de los procesos de evolución biológica. Este tipo de algoritmos comienza con una población inicial, que va evolucionando a lo largo de múltiples generaciones de acuerdo a los principios de selección natural y supervivencia del más fuerte postulados por Charles Darwin en 1859 en su famoso libro "El origen de las especies". Un recorrido histórico sobre este tipo de algoritmos puede encontrarse en [**G, 06**].

Las ideas de Holland fueron formalizadas por su alumno Goldberg en [**G, 89**]. Cuando trasladamos las ideas anteriores a un lenguaje más matemático nos encontramos con varios retos. En primer lugar, hemos de definir las estructuras de datos que vamos a utilizar para manejar los individuos de la población. Por otro lado, hemos de especificar

una función que nos permita evaluar la calidad de los distintos individuos de nuestra población. Por último, necesitamos detallar los operadores genéticos fundamentales, que son los operadores de inicialización, selección, recombinación (cruce y mutación) y reemplazo. De forma esquemática, se presenta la estructura de un algoritmo genético en el Algoritmo 3.

Algoritmo 3 Algoritmo Genético

1: $P :=$ Población inicial
2: Evaluar P
3: **mientras** el criterio de parada no se satisface **hacer**
4: $P^* :=$ Recombinar(Seleccionar(P))
5: Evaluar(P^*)
6: $P :=$ Reemplazar($P \cup P^*$)
7: **fin mientras**

El esquema general expuesto en el Algoritmo 3 ha de adaptarse al problema particular que se intenta resolver, concretando cada una de sus componentes. En nuestro caso, el algoritmo se detiene si se supera el número máximo de iteraciones o si no se mejora la calidad del mejor individuo de la población durante un número prefijado de iteraciones.

Organizaremos esta sección referente al algoritmo genético en varias subsecciones. La parte relativa a sistemas de codificación/decodificación se tratará en la Subsección 2.3.1 mientras que los operadores de inicialización, selección, cruce, mutación y reemplazo se detallarán en la Subsección 2.3.2. En la última subsección (2.3.3), presentaremos los mecanismos de control de diversidad que hemos empleado.

2.3.1. Codificación/Decodificación. En esta subsección se expondrá la estructura de datos escogida para representar a los individuos de la población así como los algoritmos que se emplean para transformar soluciones del `CVRP` (fenotipo) en grandes rutas (cromosoma, genotipo) y viceversa.

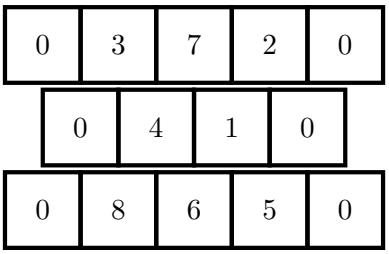

FIGURA 2.1. Ejemplo: representación de una solución de una instancia de `CVRP` con 8 clientes que necesita 3 vehículos.

Para definir la estructura de datos que utilizaremos para representar la población hemos de tener el cuenta el problema objeto de este trabajo, el `CVRP`. En este caso, las soluciones son conjuntos de rutas disjuntas que satisfacen las restricciones de capacidad. Podemos representar cada ruta mediante una lista de números, donde el primer y el último número son 0 y los números intermedios representan los nodos del grafo G que visita la ruta correspondiente, los cuales aparecen en el mismo orden en el que los recorre dicha ruta. El 0 representa al depósito, y su posición en la lista anterior nos indica que todas las rutas comienzan y terminan en este lugar. Por lo tanto, si queremos representar una solución del `CVRP` basta crear una lista que contenga todas las listas que representan las distintas rutas que constituyen la solución. Un ejemplo de solución

así representada (fenotipo) aparece en la Figura 2.1.

En el algoritmo genético, no emplearemos una codificación "directa", sino que vamos a seguir el esquema de Prins [**P, 04**], quien realiza el algoritmo genético sobre grandes rutas, las cuales son posteriormente transformadas en soluciones del CVRP. La codificación que usaremos para las grandes rutas es idéntica a la del CVRP salvo en dos aspectos. En primer lugar, en este caso disponemos de un vehículo, y por lo tanto, de una única lista. En segundo lugar, en esta codificación no incluimos los 0's que representan al depósito. Un ejemplo de una solución representada de esta forma (cromosoma, genotipo) aparece en la Figura 2.2.

3	7	2	4	1	8	6	5

FIGURA 2.2. Ejemplo: codificación de la solución de la Figura 2.1.

Como ya hemos mencionado previamente, el algoritmo genético trabajará con una codificación "indirecta" en forma de grandes rutas que hará necesario decodificar a soluciones del CVRP para la fase de evaluación. En los párrafos siguientes, se expondrán los algoritmos de codificación y decodificación que hemos empelado en este trabajo.

Comenzaremos con el algoritmo de codificación, que nos permite transformar soluciones del CVRP en grandes rutas. Dada una solución del CVRP, como lista que contiene listas que representan a las distintas rutas, el algoritmo de codificación obtiene un cromosoma concatenando las distintas rutas del CVRP y eliminando los 0's. El resultado es una gran ruta que resulta de relajar la restricción de capacidad, codificada mediante una lista de números. Un ejemplo de codificación es el paso de la Figura 2.1 a la Figura 2.2.

El algoritmo de decodificación que usaremos para transformar grandes rutas en soluciones del CVRP es bastante más complejo. Este algoritmo se basa en el algoritmo SPLIT, propuesto por Beasley en 1983 [**B, 83**]. El algoritmo SPLIT recibe como entrada una gran ruta y proporciona como salida una partición óptima de la gran ruta en rutas para el CVRP que tienen en cuenta la restricción de capacidad. Cuando decimos óptima, nos referimos a que SPLIT proporciona la partición de menor coste que respeta las restricciones de carga del CVRP.

El algoritmo SPLIT propuesto por Beasley se basa en la obtención de caminos mínimos en un grafo auxiliar. Dada una gran ruta $T := \{t_1, t_2, \ldots, t_n\}$, se construye un grafo auxiliar dirigido $H := (V, A, Z)$ que modela todas las rutas factibles que se pueden extraer de T. El conjunto de vértices de H contiene un nodo inicial 0 y n nodos enumerados de 1 a n, que representan a $t_1, \ldots, t_n$ respectivamente. El conjunto de aristas A incluye un arco $(i-1, j)$ por cada subsecuencia de clientes $(t_i, t_{i+1}, \ldots, t_j)$ que representa una ruta factible para el CVRP, es decir, aquellas que verifican $\sum_{k=i}^{j} q(t_k) \leq Q$ donde $i, j \in \{1, \ldots, n\}$. Por otro lado, Z representa los costes asociados, que vienen dados por $z(i-1, j) = c(0, t_i) + \sum_{k=i}^{j-1} c(t_i, t_{i+1}) + c(t_j, 0)$. Cabe destacar, que el grafo resultante es un grafo dirigido acíclico. La partición óptima para el CVRP coincide con el camino más corto del nodo 0 al nodo n, que puede obtenerse en $\mathcal{O}(|A|)$ usando el algoritmo de Bellman para grafos dirigidos acíclicos.

En nuestro artículo de referencia, Prins [**P, 04**] realiza una implementación del algoritmo SPLIT que no requiere calcular de forma explícita el grafo auxiliar. Prins emplea un algoritmo de etiquetado, cuyo pseudocódigo se describe en el Algoritmo 4. En este

algoritmo, los bucles **para** y **repetir** se utilizan para inspeccionar cada posible ruta $(t_i, t_{i+1}, \ldots, t_j)$, calculando su coste C y su carga L. Cuando incrementamos j, el coste C y la carga L se actualizan en $\mathcal{O}(1)$ ya que no es necesario volver a escanear toda la subsecuencia. Por otro lado, si la ruta modelada por el arco $(i-1, j)$ en el grafo auxiliar satisface la restricción de capacidad y el camino obtenido al añadir el arco $(i-1, j)$ al camino más corto que finaliza en el nodo $i-1$ mejora el coste del camino más corto conocido al nodo j, la etiqueta V_j se actualiza con el nuevo coste y el predecesor de j en ese camino se almacena en P_j. Los predecesores son necesarios para reconstruir el camino cuando finaliza el algoritmo.

Algoritmo 4 Algoritmo `SPLIT`

Entrada: $T := \{t_1, t_2, \ldots, t_n\}$ ▷ Cromosoma
$V_0 = 0$. Inicializar el resto de etiquetas $V_1, \ldots, V_n$ a $+\infty$.
para $i := 1$ hasta n **hacer**
    $L := 0$; $C := 0$; $j := i$
    repetir
        $L := L + q(t_j)$
        si $i = j$ **entonces**
            $C := c(0, t_j) + c(t_j, 0)$
        si no
            $C := C - c(t_{j-1}, 0) + c(t_{j-1}, t_j) + c(t_j, 0)$
        fin si
        si $L \leq Q$ **entonces**
            si $V_{i-1} + C < V_j$ **entonces**
                $V_j := V_{i-1} + C$
                $P_j := i - 1$
            fin si
            $j := j + 1$
        fin si
    hasta que $j > n$ o $L > Q$
fin para

La complejidad del Algoritmo 4 puede obtenerse de forma sencilla. Sea b el número medio de clientes en cada subsecuencia factible. El valor b coincide con el número medio de arcos salientes de cada nodo del grafo auxiliar H, de lo que se deduce que este grafo contiene nb arcos y por lo tanto, el algoritmo `SPLIT` tiene una complejidad de $\mathcal{O}(nb)$.

Actualmente, el método `SPLIT` es uno de los métodos más empleados en el ámbito del enrutamiento de vehículos. Un amplio estudio de este algoritmo puede encontrarse en [**PLR, 09**].

Como hemos indicado en la introducción de esta sección, es necesario definir una función de evaluación que nos permita cuantificar la calidad de los individuos de la población. En este trabajo, nuestra función de evaluación va a recibir como entrada un cromosoma y a devolver como salida el coste asociado a la solución del `CVRP` obtenida por el algoritmo `SPLIT`.

2.3.2. Operadores. El objetivo de esta sección es describir los operadores de inicialización, selección, cruce, mutación y reemplazo que se han empleado en este trabajo. Una de las ventajas de emplear el enfoque "ordena primero, agrupa después" es que en el algoritmo genético se pueden utilizar operadores de cruce y mutación basados en órdenes (i.e., permutaciones o grandes rutas), una de las representaciones más usadas para el `TSP` para la que existe abundante literatura. En [**LKMID, 99**] podemos

encontrar una recopilación de los operadores de cruce y mutación más empleados con representaciones basadas en órdenes.

Comenzamos la sección con el operador de inicialización. Recordemos que los individuos de nuestra población están codificados de forma "indirecta" en grandes rutas. Los individuos de la población inicial serán generados de forma aleatoria permutando una lista que contenga los números desde 1 hasta n. Por otro lado, en este trabajo, se van a introducir dos individuos en la población inicial generados por los heurísticos Nearest Neighbor y de Clarke & Wright, descritos en la Sección 2.2. Cabe destacar, que ambos heurísticos obtienen soluciones para el `CVRP`, las cuales serán codificadas como cromosomas según lo visto en la Subsección 2.3.1.

Pasamos ahora a describir el operador de selección. El objetivo de este operador genético es extraer de la población las grandes rutas que empleará el operador de cruce para generar nuevos individuos. En la literatura podemos encontrar varias propuestas para operadores de selección, siendo los más populares: la selección por ruleta, la selección por torneo y la selección aleatoria. En la selección por ruleta los individuos son escogidos de forma inversamente proporcional a su coste, esto es, a menor coste mayor probabilidad de ser seleccionado. La selección por torneo consiste en obtener un subconjunto de la población de forma aleatoria y seleccionar a los individuos de menor coste. Por último, la selección aleatoria extrae un individuo de la población de acuerdo a una distribución de probabilidad uniforme. En este trabajo se ha optado finalmente por la selección aleatoria como se indica en la Sección 5.1.

Algoritmo 5 Operador de Cruce: `OX1`

Entrada: $padre, madre$ ▷ Cromosomas de longitud L
$L :=$ longitud($padre$)
$pos_1 :=$ númeroAleatorio$(0, L-1)$
$pos_2 :=$ númeroAleatorio$(pos1, L-1)$
$hijo_1 :=$ ceros(L); $hijo_1 := padre[pos_1 : pos_2]$
$hijo_2 :=$ ceros(L); $hijo_2 := madre[pos_1 : pos_2]$
$i, j, k := pos_2$
repetir
    si $madre[i]$ no está en $padre[pos_1 : pos_2]$ **entonces**
        $hijo_1[j] := madre[i]$
        $j = (j+1) \bmod L$
    fin si
    si $padre[i]$ no está en $madre[pos_1 : pos_2]$ **entonces**
        $hijo_2[k] := padre[i]$
        $k = (k+1) \bmod L$
    fin si
    $i = (i+1) \bmod L$
hasta que $i = pos2$
devolver $hijo1, hijo2$

El siguiente operador que vamos a describir es el de cruce, que combina parejas de soluciones obtenidas en el paso de selección para generar nuevas soluciones. La idea del cruce es la siguiente: si combinamos las partes "adecuadas" de dos soluciones buenas para generar una nueva solución, la calidad de esta nueva solución debe ser mayor que la de los padres. En este trabajo, se ha escogido como operador de cruce, el operador `OX1`, cuyo pseudocódigo se describe en el Algoritmo 5, donde $lista[pos_1 : pos_2]$ representa la sublista que comienza en la posición pos_1 y termina en la posición $pos_2 - 1$, longitud($lista$) es un método que devuelve la longitud de $lista$, númeroAleario(a, b) es

un método que genera un número aleatorio en el rango $[a, b]$ y ceros(L) es un método que crea una lista de ceros de longitud L.

Continuamos con los operadores de mutación. El objetivo de los operadores de este tipo es modificar soluciones de forma aleatoria para ampliar el rango de búsqueda y poder de este forma abandonar óptimos locales. Cabe destacar, que no se aplica el operador de mutación a todas las instancias, solamente se aplica a un pequeño porcentaje de ellas, según una probabilidad de mutación p_m. En este trabajo, para aumentar la diversidad, hemos empleado 3 operadores de mutación: mutación por inversión simple (Algoritmo 6), mutación por inversión de sublista (Algoritmo 7) y mutación por mezcla de sublista (Algoritmo 8).

El primero de los operadores de mutación que hemos utilizado (Algoritmo 6) es bastante sencillo, y consiste en intercambiar dos posiciones obtenidas de forma aleatoria.

Algoritmo 6 Mutación: Inversión Simple

1: **Entrada:** $ruta$ ▷ Cromosoma
2: $L :=$ longitud($ruta$)
3: $pos_1, pos_2 :=$ númeroAleatorio$(0, L-1)$ ▷ Con $pos_1 \neq pos_2$
4: Intercambiar los valores de $ruta[pos_1]$ y $ruta[pos_2]$
5: **devolver** $ruta$

El segundo operador de mutación (Algoritmo 7) realiza una inversión de una sublista de la gran ruta proporcionada como entrada.

Algoritmo 7 Mutación: Inversión Sublista

1: **Entrada:** $ruta$ ▷ Cromosoma
2: $L :=$ longitud($ruta$)
3: $pos_1, pos_2 :=$ númeroAleatorio$(0, L-1)$ ▷ Con $pos_1 < pos_2$
4: $subruta := ruta[pos_1 : pos_2]$
5: InvertirOrden($subruta$)
6: **devolver** $ruta[0 : pos_1] + subruta + ruta[pos_2 : L]$

El tercer y último operador de mutación que hemos utilizado (Algoritmo 8) es similar al anterior, la diferencia es que en este caso, en lugar de invertir la sublista, esta se mezcla de forma aleatoria.

Algoritmo 8 Mutación: Mezcla Sublista

1: **Entrada:** $ruta$ ▷ Cromosoma
2: $L :=$ longitud($ruta$)
3: $pos_1, pos_2 :=$ númeroAleatorio$(0, L-1)$ ▷ Con $pos_1 < pos_2$
4: $subruta := ruta[pos_1 : pos_2]$
5: Mezclar($subruta$)
6: **devolver** $ruta[0 : pos_1] + subruta + ruta[pos_2 : L]$

Finalizamos esta subsección describiendo el mecanismo de reemplazo que hemos utilizado. El operador de reemplazo determina qué individuos van a "sobrevivir", y por lo tanto, formarán parte de la siguiente generación. En este caso, el operador de reemplazo va a trabajar sobre una estructura que hemos denominado "familia". Una familia está formada por 4 cromosomas, dos padres, que se han obtenido mediante el operador de selección, y dos hijos, obtenidos al mutar el resultado del operador de cruce `OX1` aplicado a los padres anteriores.

El operador de reemplazo comienza aplicando elitismo para obtener la mejor solución de la generación actual. Este primera fase del operador de reemplazo es necesaria ya que el operador de selección es probabilista y no podemos asegurar que el mejor individuo de la generación actual pase a la siguiente. Una vez almacenada la mejor solución de la generación actual, el operador de reemplazo itera sobre las "familias" realizando una selección por torneo para obtener los 2 individuos de mayor calidad de cada una de ellas y añadirlos a la población de la siguiente generación. Uno de los problemas que presenta este método es que, como consecuencia de la selección aleatoria, puede aparecer un mismo padre en varias familias. Si este padre es de buena calidad, varias copias del mismo pasarán a la siguiente generación. El fenómeno anterior es indeseable, ya que propiciará la convergencia a óptimos locales, y vamos a evitarlo empleando un mecanismo para preservar la diversidad basado en un archivo de soluciones.

2.3.3. Mecanismos de control de diversidad. Los algoritmos de búsqueda metaheurística han de lograr un equilibrio entre la tendencia a converger a buenas zonas del espacio de búsqueda y la capacidad de exploración de dicho espacio de búsqueda. Este equilibrio es especialmente delicado en el `CVRP`, con un espacio de búsqueda inmenso con muchos óptimos locales. El objetivo del algoritmo memético es encontrar dicho equilibro, donde la parte genética se utiliza para preservar y favorecer la diversidad mientras que las búsquedas locales se emplean para el proceso de intensificación.

El estudio en detalle de la diversidad viene motivado por los resultados experimentales que se obtuvieron durante las primeras pruebas. En estas pruebas, se percibía que la población no evolucionaba y se quedaba atascada en óptimos locales.

El algoritmo inicial presentaba únicamente un operador de mutación, el Algoritmo 6, con el que se obtenían resultados bastante pobres, ya que se trata de un operador que realiza modificaciones mínimas. Para aumentar la capacidad de exploración, añadimos los operadores de mutación que aparecen en los Algoritmos 7 y 8, que son más agresivos y, por lo tanto, ayudan a realizar una exploración más amplia de espacio de búsqueda, ayudando así a evitar los óptimos locales.

Otro de los problemas a los que nos enfrentamos en las primeras versiones del algoritmo fue la aparición de cromosomas idénticos en la población a medida que esta iba evolucionando. Para solucionar lo anterior, implementamos un archivo de soluciones. La función del archivo es almacenar un porcentaje de los mejores individuos que se han añadido a la nueva generación. Este archivo se va modificando durante la fase de reemplazo a medida que se procesan las familias y aparecen individuos de mayor calidad. Cuando se realiza la selección por torneo sobre una familia y se obtienen los 2 cromosomas de mayor calidad, se ha de comprobar si estos están en el archivo. Si alguno de los cromosomas seleccionados está en el archivo, este se elimina de la familia y se realiza de nuevo la selección por torneo sobre los elementos restantes. En el caso de que queden solamente dos individuos en la familia, estos pasan automáticamente a la siguiente generación.

Al principio, en la población inicial había únicamente una solución heurística, la proporcionada por el heurístico de Clarke & Wright. Esta solución, al ser de una calidad muy superior a las obtenidas aleatoriamente, hacía que el resto de soluciones convergieran hacia ella prematuramente. Esto producía que las soluciones de mayor calidad de la población fuesen modificaciones sencillas de la solución heurística. Para evitar lo anterior, se introdujo en la población inicial otra solución heurística, la obtenida a través del heurístico Nearest Neighbor. De esta forma, la población inicial dispone de dos soluciones con una calidad significativamente mayor que el resto, evitando así que la población evolucione hacia una única solución. Al inspeccionar las soluciones obtenidas

tras unas cuantas iteraciones en el caso de emplear dos heurísticos, se pudo apreciar que la población no tiende únicamente hacia una única solución, sino que aparecen soluciones intermedias, explorando así una parte del espacio de búsqueda que antes se ignoraba.

El último mecanismo de control de diversidad que hemos empleado es un reinicio, tal y como se sugiere en [**CLV, 01**]. En este caso, vamos a reemplazar la mitad de los individuos de la población, aquellos de menor calidad, con individuos generados de forma aleatoria cuando se alcance la mitad del número máximo de iteraciones permitidas sin mejora. Además, para favorecer la diversidad, duplicaremos tras el reinicio la probabilidad de mutación hasta que se mejore la calidad del mejor individuo de la población.

2.4. Búsqueda Local

El objetivo de esta sección es exponer las características de las búsquedas locales que vamos a emplear en el algoritmo memético.

Una búsqueda local parte de una solución inicial S, obtenida normalmente a través de algún heurístico, y considera un subconjunto $\mathcal{N}(S)$ de soluciones cercanas a S en términos de su estructura, llamado vecindad de S. En la práctica, $\mathcal{N}(S)$ está definido por una aplicación $S \rightarrow S'$ llamada movimiento, evitando de esta forma dar una definición extensiva del subconjunto. El algoritmo de búsqueda local inspecciona esta vecindad en busca de una solución de mayor calidad S'. Si S' existe, esta pasa a ser la solución actual del algoritmo de búsqueda y se repite el proceso. De esta forma, la solución se mejora de forma progresiva hasta que se alcanza un óptimo local de la vecindad. El esquema general de este tipo de búsquedas aparece en el Algoritmo 9. En este algoritmo, $c(S)$ representa el coste de la solución S. El bucle **repetir** se encarga de ir explorando las sucesivas vecindades mientras que el bucle **para cada** inspecciona cada elemento $vecino \in \mathcal{N}(S)$ para una solución S, y calcula la diferencia de costes Δ. Si $\Delta = 0$, S es un óptimo local de la vecindad y el procedimiento se detiene. Cabe destacar, que es necesario almacenar las características del mejor movimiento. En este caso, se obtiene el mejor movimiento de la vecindad, otra opción, es moverse nada más encontrar un vecino con menor coste.

Algoritmo 9 Esquema general para búsquedas locales

```
1:  Entrada: S                                          ▷ Solución inicial
2:  repetir
3:      Δ := 0
4:      para cada vecino ∈ N(S) hacer
5:          si c(vecino) − c(S) < Δ entonces
6:              Δ := c(vecino) − c(S)
7:              S' := vecino
8:              Almacenar el movimiento S ↦ S'
9:          fin si
10:     fin para cada
11:     si Δ < 0 entonces
12:         Realizar el movimiento S ↦ S'; S := S'
13:     fin si
14: hasta que Δ = 0
```

2.4.1. Vecindades. En esta subsección, vamos a describir los movimientos que hemos empleado en este trabajo, que son el `2-opt` y el `2-opt*` y que definen estructuras de vecindad para el `CVRP`.

El movimiento `2-opt` es un movimiento en el que está involucrada una única ruta y consiste en eliminar dos aristas para a continuación generar otras dos conectando los vértices de las aristas eliminadas de forma distinta. Por ejemplo, dada una ruta R, un movimiento `2-opt` puede ser el descrito en la Figura 2.3: si se eliminan las aristas (u, x) y (v, y) de R, se añaden las aristas (u, v) y (x, y) a R. Nótese que este movimiento invierte el camino que había de x a v en la solución original. Por otro lado, este tipo de movimiento sólo tiene sentido si la diferencia de costes entre las soluciones es negativa.

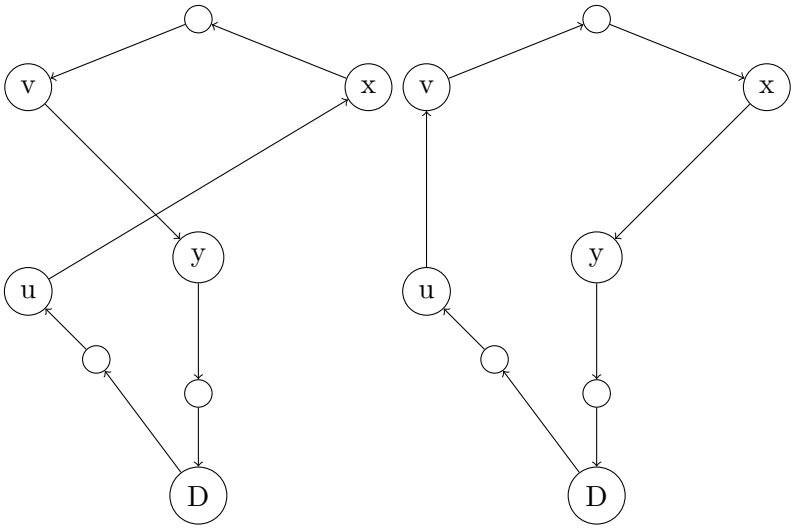

FIGURA 2.3. Ejemplo de movimiento `2-opt`: si se eliminan las aristas (u, x) y (v, y), se añaden las aristas (u, v) y (x, y). En este caso, D representa el depósito.

El Algoritmo 10 contiene el pseudocódigo del movimiento `2-opt`. En este caso, el bucle **repetir** se encarga de ir explorando las sucesivas vecindades mientras que el bucle **para cada** inspecciona cada una de las rutas $R = \{r_1 = 0, r_2, \ldots, r_{|R|} = 0\}$ presentes en la solución S del `CVRP`. El tamaño de la vencindad $\mathcal{N}(S)$ en este algoritmo es $\mathcal{O}(n^2)$ para n clientes [**LK, 73**].

El movimiento `2-opt*` puede verse como una generalización del movimiento `2-opt` a dos rutas. En este caso, al estar involucradas dos rutas, hemos de tener en cuenta las restricciones de capacidad de los vehículos. Por ejemplo, si tenemos dos rutas R y T, un movimiento `2-opt*` puede ser el descrito en la Figura 2.4: si se eliminan las aristas $(u, x) \in R$ y $(v, y) \in T$, se añaden las aristas (u, y) y (v, x), obteniéndose dos nuevas rutas. En este caso, las vecindades tienen un tamaño $\mathcal{O}(n^2)$ donde n es el número de clientes.

El Algoritmo 11 contiene el pseudocódigo del movimiento `2-opt*`. En este algoritmo, para cada pareja de rutas distintas $\{R, T\}$, los índices u y v se emplean para inspeccionar cada pareja de nodos (r_u, t_v) en tiempo constante. Para cada pareja de nodos (r_u, t_v) se actualiza la demanda acumulada y si se verifican las restricciones de capacidad, se calculan los costes variacionales y se actualiza, si es posible, el mejor movimiento. Por otro lado, Q_R y Q_T representan la demanda acumulada hasta el momento mientras que $q(R)$ y $q(T)$ hacen referencia a la demanda total de R y T respectivamente. Destacar

que r_u y t_v también pueden ser el depósito, para ello es necesario asumir que $q(0) = 0$. En particular, si r_u es el último cliente de R y t_v es el depósito al principio de T, las dos rutas pueden unirse a través del movimiento `2-opt*`.

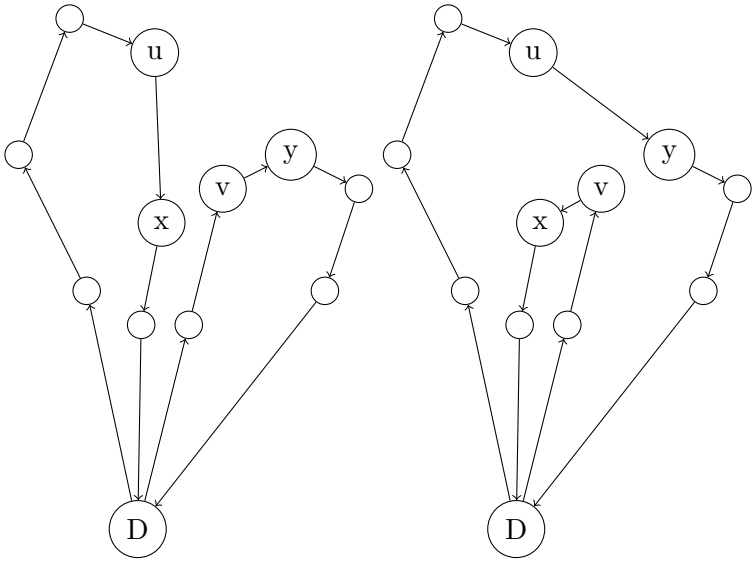

FIGURA 2.4. Ejemplo de movimiento `2-opt*`: si se eliminan las aristas (u, x) y (v, y), se añaden las aristas (u, y) y (v, x). En este caso, D representa el depósito.

En el Algoritmo 11 las demandas acumuladas se han calculado de forma incremental. Sin embargo, en la versión definitiva de este trabajo, estas cantidades han sido precomputadas. Este enfoque proviene del artículo [**VCGP, 15**] de Vidal et al., donde los autores se dan cuenta de que la mayoría de los movimientos de las búsquedas locales pueden expresarse como concatenaciones de secuencias de nodos. La principal ventaja de este enfoque es la mejora de la legibilidad y la extensibilidad del código. Para cada cantidad útil Z empleada en la búsqueda local, los autores proponen precalcular $Z(\sigma)$ para cualquier secuencia de nodos σ contenida en las rutas de la solución inicial. Estos cálculos se realizan mediante dos operadores:

- **Inicialización:** para calcular el valor de $Z(\sigma)$ cuando σ está formado por un único nodo.
- **Concatenación:** para calcular el valor de la concatenación de σ y τ, esto es, $Z(\sigma \oplus \tau)$, cuando conocemos los valores de $Z(\sigma)$ y $Z(\tau)$.

En la práctica, Z es una matriz, donde Z_{ij} almacena el valor asociado a la secuencia de nodos delimitada por i y j. El operador de inicialización nos permite calcular Z_{ii} mientras que a través del operador de concatenación podemos obtener, usando el valor anterior, la cantidad Z_{ij} para cada j.

En el caso del `CVRP`, emplearemos esta técnica para obtener la demanda acumulada de cada secuencia de nodos σ. Estas secuencias incluirán el depósito al principio y al final, por lo que las filas y las columnas de la matriz $\mathcal{Q}$, que contiene las demandas acumuladas, se indexarán de 0 hasta $n + 1$. Los operadores en este caso vienen dados por:

- **Inicialización:** $\mathcal{Q}(i, i) := q(i)$ con $i \in V$, y teniendo en cuenta que $q(0) = q(n + 1) = 0$ donde n es el número de clientes.

- **Concatenación:** $\mathcal{Q}(\sigma \oplus \tau) := \mathcal{Q}(\sigma) + \mathcal{Q}(\tau)$ donde σ y τ son secuencias de. nodos

Algoritmo 10 Movimiento `2-opt`

Entrada: S ▷ Solución del `CVRP`
repetir
  $\Delta^* := 0$
  para cada ruta $R \in S$ **hacer**
    para $x := 2$ hasta $|R| - 2$ **hacer**
      para $v := x + 1$ hasta $|R| - 1$ **hacer**
        $\Delta := c(r_{x-1}, r_v) + c(r_x, r_{v+1}) - c(r_{x-1}, r_x) - c(r_v, r_{v+1})$
        si $\Delta < \Delta^*$ **entonces**
          $\Delta^* := \Delta$; $x^* := x$; $v^* := v$
        fin si
      fin para
    fin para
  fin para cada
  si $\Delta^* < 0$ **entonces**
    $i := x^*$; $j := v^*$
    mientras $i < j$ **hacer**
      Intercambiar r_i y r_j; $i := i + 1$; $j := j - 1$
    fin mientras
  fin si
hasta que $\Delta = 0$

Algoritmo 11 Movimiento `2-opt*`

Entrada: S ▷ Solución del `CVRP`
repetir
  $\Delta^* := 0$
  para cada par de rutas distintas $\{R, T\}$ en S **hacer**
    $Q_R := 0$; $Q_T := 0$
    para $u := 0$ hasta $|R| - 1$ **hacer**
      $Q_R := Q_R + q(r_u)$
      para $v := 0$ hasta $|T| - 1$ **hacer**
        $Q_T := Q_T + q(t_v)$
        si $Q_R + q(T) - Q_T \leq Q$ y $Q_T + q(R) - Q_R \leq Q$ **entonces**
          $\Delta := c(r_u, t_{v+1}) + c(t_v, r_{u+1}) - c(r_u, r_{u+1}) - c(t_v, t_{v+1})$
          si $\Delta < \Delta^*$ **entonces**
            $\Delta^* := \Delta$; $u^* := u$; $v^* = v$
          fin si
        fin si
      fin para
    fin para
  fin para cada
  si $\Delta^* < 0$ **entonces**
    $R := (r_1, \ldots, r_{u^*}, t_{v^*+1}, \ldots, t_{|T|})$
    $T := (t_1, \ldots, t_{v^*}, r_{u^*+1}, \ldots, r_{|R|})$
  fin si
hasta que $\Delta = 0$

En la versión final del algoritmo correspondiente a la búsqueda local hemos combinado los movimientos `2-opt` y `2-opt*`, esto ha sido posible ya que hemos considerado el movimiento `2-opt` como el caso particular del movimiento `2-opt*` en el que las rutas son iguales.

2.4.2. Búsqueda Tabú Granular. La búsqueda tabú es uno de los métodos más empleados y más efectivos en el ámbito de la optimización. Este método fue propuesto por primera vez por el informático estadounidense Fred Glover en 1986 [**G, 86**]. En [**GP, 19**] podemos encontrar una recopilación de las aplicaciones actuales de la búsqueda tabú.

La idea de la búsqueda tabú es procesar completamente la vecindad de la solución actual y hacer el mejor movimiento posible, incluso si esto deteriora la calidad de la solución. Este algoritmo evita volver a una solución ya visitada a través del uso de una memoria, llamada lista tabú, que almacena la historia reciente de la búsqueda. Por otro lado, la mejor solución obtenida durante la ejecución del algoritmo es almacenada para ser retornada al final, cuando se alcanza el criterio de parada. Este tipo de búsqueda nos permite realizar movimientos "a peor" para escapar de óptimos locales.

El principal problema de la búsqueda tabú es que las vecindades son bastante grandes, y su procesamiento requiere mucho tiempo de cómputo. En este trabajo, vamos a utilizar el método de búsqueda tabú desarrollado por Toth & Vigo en [**TV, 03**], denominado búsqueda tabú granular, el cual reduce los tiempos de cómputo considerablemente. La idea de Toth & Vigo es prohibir los movimientos que involucran aristas que tienen pocas posibilidades de aparecer en soluciones de calidad. Esta reducción del espacio de búsqueda nos permitirá procesar las vecindades en mucho menos tiempo, y generalmente, sin una pérdida significativa de la calidad de la solución obtenida.

En su artículo, Toth & Vigo proponen eliminar las aristas del grafo de mayor coste, ya que es poco probable que estas aparezcan en una solución de calidad. Al eliminar estas aristas, obtenemos un grafo disperso $G' := (V, E')$ que contiene los n vértices de G, pero con $|E'| \ll n^2$ aristas. E' incluye todos los arcos con coste bajo y aquellos que unen a los clientes con el depósito.

Por lo tanto, las vecindades van a estar formadas por soluciones generadas por aristas pertenecientes a G', es decir, movimientos que involucren al menos una arista del conjunto E'. Lo anterior no significa que la solución actual no tenga ninguna arista de coste elevado. Nuestra solución puede contener aristas de coste alto, es más, estas pueden ser insertadas a través de un movimiento, ya que la prohibición nos dice únicamente que no se consideran aquellos movimientos donde todas las aristas involucradas presentan un coste alto.

El siguiente paso es proporcionar criterios concretos de selección de aristas. En este trabajo vamos a seguir el criterio que Toth & Vigo plantean en su artículo. El conjunto E' contiene todas las aristas (i, j) que conectan a los clientes con el depósito, y aquellas que conectan dos clientes y tienen un coste menor que βd, donde β es una constante denominada granularidad y d es el coste medio de las aristas que aparecen en la solución inicial. Sin embargo, dado que excluimos la mayoría de las aristas, la reducción anterior debe ser temporal para no perder soluciones. Si no se alcanza una mejora tras un número específico de iteraciones durante una búsqueda tabú granular, la granularidad se aumenta a un valor $\beta' > \beta$ para agrandar el espacio de búsqueda, añadiendo a G' todas las aristas con un coste menor que $\beta' d$. Este ajuste dinámico de granularidad nos va a ayudar a diversificar la búsqueda. Si n es el número de clientes, cada $2n$ iteraciones, E' es reconstruido usando el d de la mejor solución actual.

Por otro lado, para manejar de forma eficaz los movimientos permitidos, debemos adaptar las técnicas empleadas en los Algoritmos 10 y 11 a la búsqueda tabú. En este caso, la mejor solución es iterar sobre las aristas comprobando si son tabú o no, como se indica en el Algoritmo 12. Dentro de cada iteración, este algoritmo calcula la diferencia de costes de `2-opt` o de `2-opt*` dependiendo si la arista tiene los extremos en la misma ruta o en rutas distintas. En este punto, es fundamental destacar la importancia de $listaIndices$, una lista que contiene tantas posiciones como clientes e indica en cada posición la ruta donde está el cliente y su posición dentro de la misma. Esta lista permite que el procesamiento de cada arista se lleve a cabo en $\mathcal{O}(1)$. Cabe destacar, que los métodos $ruta(nodo)$ y $pos(nodo)$ hacen uso de $listaIndices$ para obtener la ruta donde está $nodo$ y su posición en dicha ruta. Tras recorrer todas las aristas, el algoritmo actualiza los tiempos de la $listaTabu$. Si $\Delta^* < 0$, se ejecuta el movimiento almacenado, se añaden las nuevas aristas a la $listaTabu$, y se actualizan la $listaIndices$ y la matriz de cargas $\mathcal{Q}$. Cuando se elimina una arista, se etiqueta como tabú durante las siguientes t iteraciones, donde t es un número aleatorio entre t_{min} y t_{max}. Esto se corresponde a tener una lista tabú dinámica. A diferencia de lo que hacen Toth & Vigo, nosotros no vamos a permitir soluciones intermedias que no satisfacen las restricciones de capacidad.

Acabamos esta sección presentando el algoritmo de la búsqueda tabú granular que empleamos en este trabajo, cuyo psuedocódigo aparece en el Algoritmo 13. La búsqueda comienza calculando el valor d y obteniendo la matriz de cargas $\mathcal{Q}$ siguiendo el enfoque de [**VCGP, 15**]. Para agilizar la búsqueda, se construye $listaIndices$, estructura de datos fundamental para el Algoritmo 12. Posteriormente, se obtiene el grafo reducido G'. En it se almacena el número de iteraciones sin mejora, en $itdiv$ el número de iteraciones que se han producido con el valor de granularidad actual, $maxIterIni$ es el número máximo de iteraciones con β, $maxIterDiv$ es el número máximo de iteraciones con β', div indica el tipo de granularidad y $maxIt$ es el número máximo de iteraciones del algoritmo. Dentro del **para** se realiza la búsqueda local que aparece en el Algoritmo 12, se comprueba si la solución obtenida en la búsqueda local tiene mayor calidad que la actual y se actualizan los parámetros del algoritmo. Cabe destacar que este algoritmo dispone de dos mecanismos de parada. El algoritmo se detiene cuando se supera $maxIt$ o cuando estando con granularidad β' la lista tabú es vacía, lo que indica que se ha alcanzado un óptimo de la vecindad.

Algoritmo 12 Búsqueda local: Combinación de los movimientos `2-opt` y `2-opt*`

1: $\Delta^* := 0$
2: **para cada** arista en E' **hacer**
3: **si** arista no está en $listaTabu$ **entonces**
4: $(a_1, a_2) := arista$ ▷ Vértices de la arista
5: $ruta_1 := ruta(a_1)$; $ruta_2 := ruta(a_2)$
6: **si** $ruta_1 = ruta_2$ **entonces** ▷ Caso: `2-opt`
7: **si** $pos(a_1) < pos(a_2)$ **entonces**
8: $x := pos(a_1)$; $v := pos(a_2) - 1$
9: **si no**
10: $x := pos(a_2)$; $v := pos(a_1) - 1$
11: **fin si**
12: $\Delta := c(ruta_1(x-1), ruta_1(v)) + c(ruta_1(x), ruta_1(v+1)) - c(ruta_1(x-1), ruta_1(x)) - c(ruta_1(v), ruta_1(v+1))$
13: **si** $\Delta < \Delta^*$ **entonces**
14: $\Delta^* := \Delta$
15: $x^* := x$; $v^* := v$; $ruta_1^* := ruta_1$; $tipo := 1$
16: **fin si**
17: **si no** ▷ Caso: `2-opt*`
18: $x := pos(a_1)$; $v := pos(a_2)$
19: **si** $\mathcal{Q}(0, ruta_1(x-1)) + \mathcal{Q}(v+1, n+1) \leq Q$ y $\mathcal{Q}(0, a_2) + \mathcal{Q}(a_1, n+1) \leq Q$ **entonces**
20: $\Delta := c(ruta_1(x-1), ruta_2(v+1)) + c(a_2, a_1) - c(ruta_1(x-1), ruta_1(x)) - c(ruta_2(v), ruta_2(v+1))$
21: **si** $\Delta < \Delta^*$ **entonces**
22: $\Delta^* := \Delta$
23: $x^* := x$; $v^* := v$; $ruta_1^* := ruta_1$; $ruta_2^* := ruta_2$; $tipo := 2$
24: **fin si**
25: **fin si**
26: **fin si**
27: **fin si**
28: **fin para cada**
29: Actualizar los tiempos de $listaTabu$
30: **si** $\Delta^* < 0$ **entonces**
31: Generar números aleatorios entre t_{min} y t_{max} para los tiempos de $listaTabu$
32: **si** $tipo = 1$ **entonces**
33: Aplicar el movimiento a $ruta_1$
34: Añadir las aristas a $listaTabu$
35: Actualizar $listaIndices$ y $\mathcal{Q}$
36: **si no si** $tipo = 2$ **entonces**
37: Generar y almacenar las nuevas rutas
38: Añadir las aristas a $listaTabu$
39: Actualizar $listaIndices$ y $\mathcal{Q}$
40: **fin si**
41: **fin si**

Algoritmo 13 Búsqueda Tabú Granular

Calcular d
Calcular matriz $\mathcal{Q}$ que contiene la carga de las distintas subsecuencias
$listaIndices$:= Lista que indica para cada cliente, en qué ruta está y su posición dentro de la misma
G' := GrafoReducido(β, d)
$S^{'}$:= S
$listaTabu := \emptyset$
$itdiv := 0$; $it := 0$; $div := False$
para $k := 1$ hasta $maxIt$ **hacer**
    BúsquedaLocal($S, G', listaTabu, listaIndices, \mathcal{Q}$)
    si $k \, mod \, 2n = 0$ **entonces** ▷ donde n es el número de clientes
        Actualizar d con $S^{'}$
    fin si
    $itdiv := itdiv + 1$
    si $c(S) < c(S^{'})$ **entonces**
        $S^{'} := S$
    si no
        $it := it + 1$
        si ($it > maxIterIni$ o ($it > 1$ y $listaTabu = \emptyset$)) y $div = False$ **entonces**
            G' := GrafoReducido(β', d)
            $itdiv := 0$
            $div := True$
        fin si
        si $itdiv > maxIterDiv$ y $div = True$ **entonces**
            G' := GrafoReducido(β, d)
            $it := 0$
            $div := False$
        fin si
        si $div = True$ y $listaTabu = \emptyset$ **entonces**
            devolver $S^{'}$
        fin si
    fin si
fin para
devolver $S^{'}$

2.5. Algoritmo Memético

En este trabajo proponemos combinar el algoritmo genético (incluyendo las semillas heurísticas) con la búsqueda tabú granular descritos en las secciones anteriores para dar lugar a un algoritmo memético. Elaboramos este algoritmo inspirados en el trabjo de Prins [**P, 04**], quien desarrolla un algoritmo genético con un operador de mutación que consiste en una búsqueda local. A diferencia de Prins, nuestro algoritmo emplea un operador de mutación "tradicional" cuyo objetivo es diversificar mientras que la búsqueda local se utiliza en la parte de intensificación.

La finalidad de utilizar un algoritmo memético para resolver el `CVRP` es encontrar un equilibrio entre la diversidad que proporciona el algoritmo genético y la intensificación que ofrece la búsqueda tabú granular. Como ya se ha mencionado previamente, los algoritmos genéticos no son suficientemente agresivos y evolucionan a un ritmo bastante lento mientras que las búsquedas locales suelen converger a óptimos locales. Por lo tanto, el algoritmo memético funcionará de forma adecuada si existe un fenómeno de sinergia entre el algoritmo genético y la búsqueda tabú granular. Este fenómeno será estudiado experimentalmente en la Sección 5.2.

Algoritmo 14 Algoritmo Memético

$P :=$ Población inicial
Añadir a la población los individuos obtenidos con los heurísticos Nearest Neighbor y de Clarke & Wright
Intensificar P con la búsqueda tabú granular
Evaluar P
para $i := 0$ hasta $iterMax$ **hacer**
    $familias :=$ BúsquedaTabúGranular(Mutar(Cruzar(Seleccionar(P))))
    $P :=$ Reemplazar($familias$)
    si ha mejorado el coste del mejor individuo **entonces**
        $aleatoriedad := False$
        $ratioMutacion := ratioMutacionInicial$
        $iterNoMejora := 0$
    si no
        $iterNoMejora := iterNoMejora + 1$
        si $iterNoMejora > numMaxSinMejora$ **entonces**
            devolver P
        si no si $iterNoMejora > numMaxSinMejora/2$ y no $aleatoriedad$ **entonces**
            $ratioMutacion := 2 \cdot ratioMutacionInicial$; $aleatoriedad := True$
            Reiniciar población: se sustituye la mitad peor de los individuos por individuos aleatorios
        fin si
    fin si
fin para

El pseudocódigo del algoritmo memético que hemos implementado aparece en el Algoritmo 14. El algoritmo comienza creando e inicializando la población inicial, en la que se incluyen las soluciones obtenidas mediante los heurísticos Nearest Neighbor y de Clarke & Wright. A continuación, la población inicial se decodifica mediante el algoritmo `SPLIT` y se aplica la búsqueda tabú granular a todas las soluciones del `CVRP` obtenidas. Tras finalizar la búsqueda anterior, las soluciones se codifican y se reemplaza la población inicial anterior con su versión mejorada. En cada generación, tal y como ocurría en el algoritmo genético, se aplican los operadores de selección, cruce, mutación

y reemplazo. En este caso, además de los operadores anteriores, se va a aplicar la búsqueda tabú granular a los nuevos individuos. Normalmente, la intensificación no se aplica a todos los nuevos individuos y se establece una probabilidad de intensificación. Cabe destacar que, cuando se realiza la intensificación, tiene lugar un proceso de decodificiación/codificación. Este algoritmo también incluye, al igual que sucedía con el algoritmo genético, un mecanismo de diversidad basado en el reinicio y en el aumento de la probabilidad de mutación. Por último, mencionar que este algoritmo se detiene si se supera el número máximo de iteraciones permitidas $iterMax$ o si la solución del mejor individuo de la población no ha mejorado durante $numMaxSinMejora$ iteraciones.

CAPÍTULO 3

Análisis de Requisitos, Metodología y Herramientas

> If debugging is the process of removing bugs, then programming must be the process of putting them in. [1]
>
> **Edsger W. Dijkstra**

Índice

Este capítulo comienza realizando un análisis de requisitos del proyecto software que se va a elaborar para este trabajo, especificando los requisitos funcionales (3.1) y no funcionales (3.2). Posteriormente, se expone la metodología (3.3) empleada. Por último, se describen las herramientas y tecnologías utilizadas en el proyecto (3.4).

3.1. Requisitos Funcionales

En esta sección, se especifican los requisitos funcionales que debe satisfacer el sistema. Cabe destacar, que los requisitos funcionales son aquellos que definen una función del software que el sistema debe cumplir, estableciendo así el comportamiento del mismo.

A continuación, se detallan los requisitos funcionales del sistema:

ID	Descripción
RF1	El sistema aceptará ficheros en formato `VRP-REP`.
RF2	El sistema empleará técnicas heurísticas para obtener soluciones iniciales de calidad.
RF3	El sistema implementará un algoritmo genético.
RF4	El sistema implementará un algoritmo de búsqueda tabú granular.
RF5	El sistema implementará un algoritmo memético.
RF6	El sistema permitirá a los usuarios escoger si introducir heurísticos en las poblaciones iniciales de los algoritmos genético y memético.
RF7	El sistema contará con un mecanismo de escritura que permite pasar los resultados obtenidos de la ejecución de los algoritmos a ficheros de texto.

Tabla 3.1. Requisitos funcionales.

[1] Si la depuración es el proceso de eliminar errores, entonces la programación debe ser el proceso de introducirlos.

3.2. Requisitos No Funcionales

En esta sección, se especifican los requisitos no funcionales que debe satisfacer el sistema. En este caso, nos centraremos en la eficiencia y la usabilidad.

- **Eficiencia:** los usuarios deben ser capaces de modificar los parámetros de los algoritmos para adaptar las ejecuciones a las instancias del `CVRP` que quieren resolver.
- **Usabilidad:** la aplicación debe resolver cualquier instancia del `CVRP` dado su fichero en formato `VRP-REP` [2]. Además, debe generar un archivo de texto, fácilmente interpretable, con las soluciones obtenidas.

3.3. Metodología

Cualquier proyecto software debe seguir una metodología que permita controlar la evolución del mismo. La metodología es el marco de trabajo empleado para estructurar, planificar y controlar el proceso de desarrollo del proyecto. Para este proyecto, hemos escogido seguir una metodología iterativa-incremental [**LB, 03**]. Esta metodología se caracteriza por descomponer el proyecto en pequeños módulos, donde cada uno de ellos aporta una funcionalidad específica al sistema. La principal ventaja que presenta esta metodología es que permite disponer de una versión operativa del proyecto desde una etapa temprana de su desarrollo, a la cual se le van añadiendo nuevas funcionalidades de forma progresiva.

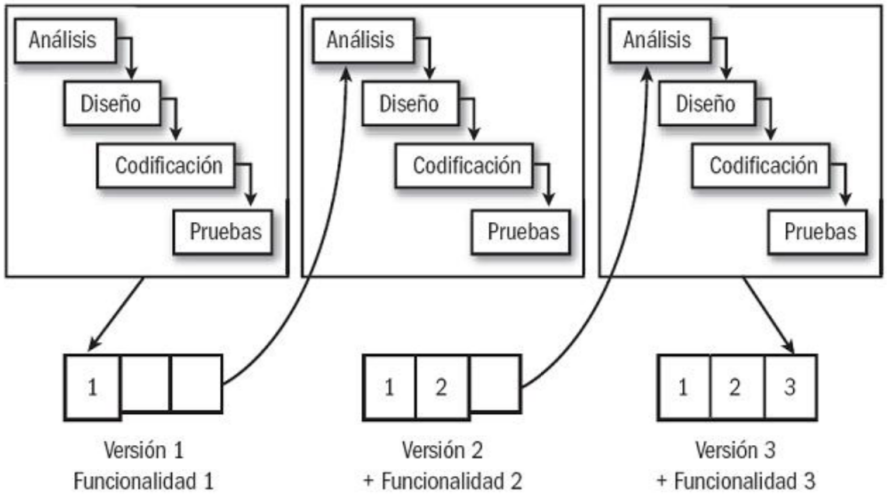

FIGURA 3.1. Modelo de desarrollo iterativo-incremental.

Para este proyecto, han sido necesarias 6 iteraciones. La primera iteración se corresponde con el diseño e implementación de un lector para ficheros en formato `VRP-REP` [RF1]. Este lector debe transformar los fichero de entrada en las estructuras de datos necesarias para trabajar con los grafos asociados a las instancias del `CVRP`.

La segunda iteración consiste en la codificación de los heurísticos voraces Nearest Neighbor y de Clarke & Wright [RF2].

[2] http://www.vrp-rep.org/resources.html

En la tercera iteración se realiza la implementación del algoritmo genético [RF3]. Se debe programar una opción para que los usuarios puedan decidir si incluir o no en la población inicial los resultados de los heurísticos [RF6].

La cuarta iteración consiste en el diseño y codificación de la búsqueda tabú granular [RF4].

En la quinta iteración, se combinan el algoritmo genético y la búsqueda tabú granular para obtener el algoritmo memético [RF5].

La sexta y última iteración consiste en el diseño e implementación de un mecanismo de escritura que permita pasar los resultados obtenidos de la ejecución de los algoritmos a ficheros de texto [RF7].

Iteración	Incorporación	Requisito
1	Lector de ficheros	RF1
2	Heurísticos constructivos	RF2
3	Algoritmo genético	RF3, RF6
4	Búsqueda tabú granular	RF4
5	Algoritmo memético	RF5
6	Mecanismo de escritura	RF7

TABLA 3.2. Iteraciones del desarrollo iterativo-incremental de este proyecto software.

3.4. Herramientas y Tecnologías

En esta sección, se describen las herramientas y tecnologías que se han empleado en el proyecto. En este caso, se ha utilizado el lenguaje de programación `Python` (3.4.1) y el entorno de desarrollo `Jupyter` (3.4.2).

3.4.1. Python. Se trata de un lenguaje de programación interpretado. `Python` fue creado por Guido van Rossum a finales de la década de los 80 y su principal ventaja es que su sintaxis favorece la legibilidad del código, permitiendo crear código que sea sencillo y potente al mismo tiempo [**VD, 95**]. Por otro lado, `Python` es un lenguaje de programación multiparadigma, ya que soporta programación orientada a objetos, programación imperativa y, en menor media, programación funcional.

Al ser un lenguaje interpretado, `Python` permite el desarrollo rápido de prototipos, algo muy deseable en software de investigación. En la actualidad, `Python` es uno de los lenguajes de programación más usados en el campo de la inteligencia artificial [**L, 19**], y dispone de un gran abanico de librerías. Una de las librerías que hemos empleado en este trabajo es `joblib` [3], que nos ha permitido paralelizar gran parte del código de forma sencilla.

El objetivo de este trabajo no es obtener una aplicación comercial. Nuestra meta es crear un prototipo para la investigación, por lo que `Python` es el lenguaje de programación más adecuado. Si queremos crear una versión más eficiente, hemos de abandonar `Python` y pasarnos a un lenguaje de programación más potente. Una opción es utilizar `C++`, un lenguaje de programación con una sintaxis mucho más compleja, pero

[3] https://joblib.readthedocs.io/en/latest/

100 veces más rápido que `Python` según la web The Computer Language Benchmark Game [4].

3.4.2. Jupyter. Se trata de un entorno de desarrollo al que se puede acceder desde un navegador web [5] o desde un servidor local. La principal característica de `Jupyter` es que su unidad básica de trabajo son las "celdas". En una celda, podemos escribir una o varias líneas de código, y ejecutarla. También, podemos escribir texto plano, código `HTML` o LaTeX.

En la actualidad, `Jupyter` es una de las herramientas más utilizadas para programar en `Python`, y esto se debe principalmente a dos razones. La primera de ellas es que `Jupyter` dispone de una interfaz dinámica basada en celdas que permite a los usuarios inspeccionar, modificar y corregir el código de forma rápida y sencilla. La segunda razón es que permite exportar fácilmente los resultados obtenidos a muchos formatos, como `HTML`, `PDF` o LaTeX. Por lo dicho anteriormente, `Jupyter` es una herramienta perfecta para el prototipado.

Por otro lado, `Jupyter` también presenta algunas desventajas. Desde mi punto de vista, los principales problemas de este entorno de desarrollo son su incompatibilidad con el sistema de control de versiones `Git`, las dificultades que surgen al realizar proyectos colaborativos o al manejar varios archivos y la ausencia de algunas de las herramientas típicas de los entornos de desarrollo como un depurador, un generador automático de código o un refactorizador. Los problemas anteriores hacen que `Jupyter` no sea una herramienta adecuada para realizar proyectos software de grandes dimensiones. Cabe destacar que al tratarse este trabajo de un prototipo y no de una gran proyecto colaborativo, las desventajas mencionadas anteriormente no han supuesto una dificultad adicional.

Finalizamos esta subsección mencionando algunas alternativas a `Jupyter`. Una de las opciones más habituales para programar en `Python` es emplear un entorno de desarrollo de escritorio como `PyCharm` [6] o `Rodeo` [7]. Estas herramientas son más completas que `Jupyter`, sin embargo, han sido descartadas ya que `Jupyter` nos ofrece una interfaz específicamente diseñada para construir prototipos de forma rápida y sencilla. Por otro lado, en la actualidad, la herramienta más parecida a `Jupyter` es `Apache Zeppelin` [8], basada también en celdas. Se ha descartado esta última herramienta ya que se encuentra en una fase inicial, por lo que es algo inestable y presenta algunos errores. Además, la comunidad de usuarios es bastante reducida y prácticamente no hay extensiones disponibles.

[4] https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/gpp-python3.html

[5] https://jupyter.org/

[6] https://www.jetbrains.com/pycharm/

[7] https://rodeo.yhat.com

[8] https://zeppelin.apache.org

CAPÍTULO 4

Diseño e implementación

> The function of good software is to make the complex appear to be simple. [1]
>
> **Grady Booch**

Índice

El objetivo de este capítulo es exponer el diseño del software especificado en el Capítulo 3 así como algunos detalles de su implementación. Este capítulo se divide en dos secciones, en la primera (4.1) se expone el diseño arquitectónico empleado mientras que en la segunda (4.2) se relaciona el diseño arquitectónico con el desarrollo del proyecto.

4.1. Diseño arquitectónico

La arquitectura en la que se basa nuestro software se conoce como arquitectura `Pipe & Filter` [**B, 17**]. Se trata de una arquitectura sencilla, que consta de una serie de componentes (filtros), que procesan/transforman datos, conectados entre sí a través de tuberías (pipelines). Esta arquitectura es típica de los programas de `Unix` y de los compiladores.

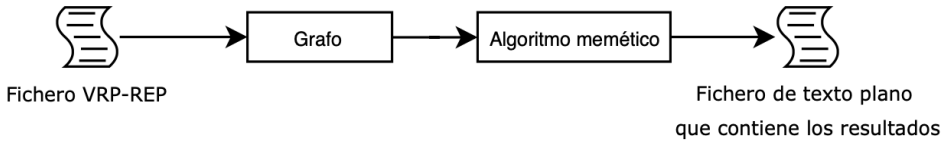

FIGURA 4.1. Arquitectura `Pipe & Filter` empleada en este trabajo.

En la Figura 4.1 se expone la arquitectura de nuestro software, donde las imágenes y recuadros de texto representan los filtros que procesan/transforman los datos y las flechas denotan las tuberías que conectan los filtros. En nuestro caso, el software comienza con un fichero de texto en formato `VRP-ERP`, que contiene información relativa a una instancia del `CVRP`. Este fichero es transformado por nuestro programa en un grafo, que representa la instancia del `CVRP` que contenía el fichero. A continuación, este grafo es procesado por el algoritmo memético. La salida de este algoritmo contiene el conjunto de rutas de la mejor solución que el algoritmo ha encontrado para el `CVRP` que

[1] La función de un buen software es hacer que lo complejo aparente ser simple.

representa el grafo así como algunas estadísticas e información acerca de su rendimiento. Finalmente, se generará un fichero de texto plano que contenga la salida del algoritmo memético.

4.2. Diseño e implementación de la aplicación

En esta sección describiremos cada una de las componentes (filtros) del software desarrollado. Una de las características de nuestro software es que cada método se encuentra en una "celda" distinta de Jupyter. Este tipo de construcción ha permitido la realización de pruebas tanto unitarias como de integración de forma progresiva.

4.2.1. Fichero VRP-REP. Nuestro software trabaja con ficheros en formato `VRP-REP`, que representan instancias del `VRP`. Este tipo de ficheros se caracterizan por tener una estructura bien definida que contiene los siguientes elementos: nombre de la instancia, comentario (autores de la instancia y el coste del óptimo o mejor solución conocida), versión del `VRP`, número de clientes, si la instancia es euclídea o no, la capacidad de los vehículos, las aristas (definidas de forma implícita si la instancia no es euclídea; coordenadas de los clientes si la instancia es euclídea) y las demandas de los clientes.

4.2.2. Grafo. Dado que no es cómodo ni eficiente trabajar directamente con ficheros de texto, vamos a procesar el fichero `VRP-REP` de entrada para obtener estructuras de datos que se adecúen mejor a nuestras necesidades. El método que se va a encargar de procesar el fichero anterior se denomina `cargarGrafo`. Este método comienza procesando las especificaciones de la instancia y verificando si se trata de una instancia del `CVRP`. A continuación, comprueba si la instancia es euclídea o no y, según el resultado obtenido, aplica el algoritmo correspondiente para procesar las aristas. Finalmente, recorre y almacena las demandas de los clientes. Como salida, el método `cargarGrafo` proporciona la matriz de adyacencia que representa el grafo de la instancia del `CVRP`, la lista que contiene las demandas de los clientes, la capacidad de los vehículos y el coste del óptimo o del mejor valor conocido. Cabe destacar que cualquier error en el formato del fichero provoca que el método `cargarGrafo` se detenga e informe al usuario del error producido.

4.2.3. Algoritmo Memético. Las estructuras de datos definidas en la subsección anterior constituyen la entrada del algoritmo memético. Cabe mencionar que normalmente este algoritmo se ejecuta varias veces y se devuelve como solución final la mejor obtenida de entre todas las ejecuciones.

En nuestro programa, este filtro se corresponde con el método `algoritmoMemetico`, que implementa el pseudocódigo que aparece en el Algoritmo 14. Se trata del algoritmo central de nuestro trabajo y, como ya hemos mencionado previamente, resulta de la combinación del algoritmo genético y la búsqueda tabú granular. El método `algoritmoMemetico` usa como base el código desarrollado para el algoritmo genético (`algoritmoGenetico`), añadiendo la búsqueda tabú granular (`busquedaTabuGranular`) al principio, para intensificar toda la población inicial, y tras el operador de mutación. Además, `algoritmoMemetico` contiene rutinas para obtener las semillas proporcionadas por los heurísticos voraces Nearest Neighbor (`heuristicoNearestNeighbor`) y de Clarke & Wright (`heuristicoClarkeWright`).

El método `algoritmoGenetico` está constituido por varios procedimientos que hacen referencia a los operadores descritos en la Subsección 2.3.2 (`creaPoblacion`, `seleccionAleatoria`, `cruceOX1`, `mutacionInversionSimple`, `mutacionInversionSublista`, `mutacionMezclaSublista` y `reemplazo`). Este método también hace uso del algoritmo `SPLIT` para evaluar a los individuos de su población.

Por otro lado, el método `busquedaTabuGranular` hace uso de la rutina `busquedaLocal`, que implementa el Algoritmo 9, y de un procedimiento que calcula el grafo dispeso (`grafoDisperso`). Además, se cuenta con el método `codificador` que permite transformar soluciones del `CVRP` en grandes rutas.

Como salida, el método `algoritmoMemetico` proporciona la población final, el coste medio de los individuos de esta población, la mejor solución obtenida junto a su coste, el número de vehículos empleados en la mejor solución y el tiempo de ejecución.

4.2.4. Fichero de resultados. A partir de la salida del algoritmo memético se genera un fichero de texto plano a través de la rutina `generarFichero`. Además del conjunto de rutas que constituyen la mejor solución y su coste, este fichero contiene el coste medio de los individuos de la población, el tiempo de ejecución y los parámetros usados en el algoritmo memético.

Si el algoritmo se ejecuta varias veces, el fichero también contendrá para cada ejecución: el coste de la mejor solución, el número de vehículos usados en la mejor solución, el coste medio de los individuos de la población y el tiempo de ejecución.

CAPÍTULO 5

Resultados Experimentales

> No amount of experimentation can ever prove me right; a single experiment can prove me wrong. [1]
>
> **Albert Einstein**

Índice

El objetivo de este capítulo es realizar una exposición de los experimentos que se han llevado a cabo para este trabajo así como de los resultados obtenidos. Este capítulo se divide en tres secciones. En la primera (5.1) se exponen los parámetros que se van a usar en los experimentos de las siguientes secciones así como el procedimiento que se ha seguido para su obtención. En la segunda sección, se realiza un estudio de sinergia (5.2) entre los métodos voraces de inicialización heurística, el algoritmo genético y la búsqueda tabú granular, mientras que en la tercera se exponen y analizan los resultados obtenidos al aplicar el algoritmo memético que hemos desarrollado a una colección de instancias (5.3).

En la actualidad, existen muchos conjuntos de instancias de referencia, también llamados *benchmarks*, para el `CVRP`. La mayoría de ellos pueden descargarse, junto con las mejores soluciones conocidas hasta el momento, en `VRP-REP` [2]. Otros conjuntos de instancias pueden encontrarse en la página web del profesor Bernabé Dorronsoro [3] y en `CVRPLIB` [4]. Una de las ventajas de disponer de la solución óptima o de la mejor conocida hasta el momento es que podemos compararla con las soluciones que hemos obtenido con nuestros algoritmos, y así evaluar la calidad de nuestro trabajo.

Tanto en el estudio de sinergia como en los experimentos realizados con el algoritmo memético, cada algoritmo se ha ejecutado 20 veces. Esto se debe a que nuestro objetivo no es sólo obtener la mejor solución que encuentra nuestro algoritmo, sino que además, queremos extraer conclusiones sobre el comportamiento medio del algoritmo, y por lo tanto, necesitamos realizar un número estadísticamente significativo de ejecuciones. Para cada instancia y cada algoritmo, recogemos el coste de la mejor y la peor solución obtenidas, el error relativo de la mejor solución obtenida con respecto a la mejor solución conocida, la media y la desviación típica del coste de las soluciones, el tiempo medio y el número de vehículos empleados en la mejor solución.

[1] Ninguna cantidad de experimentos pueden probar que tengo razón; un solo experimento puede probar que me he equivocado.

[2] `http://www.vrp-rep.org/`

[3] `http://www.bernabe.dorronsoro.es/vrp/`

[4] `http://vrp.atd-lab.inf.puc-rio.br/index.php/en/`

Los experimentos han sido realizados en un ordenador con procesador AMD A10-67000, 16 GB de RAM, 4 núcleos y sistema operativo Ubuntu 18.04.2 LTS.

5.1. Ajuste paramétrico

El objetivo de esta subsección es exponer los parámetros que se van a emplear en el estudio experimental objeto de este capítulo. Cabe destacar que los parámetros han sido obtenidos a través de una fase experimental previa, cuya finalidad ha sido encontrar un conjunto de parámetros que proporcionen soluciones de calidad en tiempo razonable, siguiendo una estrategia secuencial de optimización [**T, 09**]. En la mayoría de los casos, los parámetros son dependientes del tamaño de la instancia que se va a ejecutar.

Comenzamos con la elección del tamaño de la población. En primer lugar, nótese que una población pequeña proporciona poca diversidad y nos va a conducir a obtener óptimos locales mientras que una población grande va a requerir mucho tiempo de cómputo para su procesamiento. En nuestro caso, tras probar con diferentes tamaños, hemos elegido tomar como tamaño de población el resultado obtenido al multiplicar 0.75 por el número de clientes de la instancia.

El siguiente factor al que vamos a hacer referencia es el operador de selección. Durante la experimentación previa realizamos pruebas usando los tres operadores descritos en la Subsección 2.3.2: la selección por ruleta, la selección por torneo y la selección aleatoria. En los experimentos se obtuvieron resultados similares con todos los métodos, por lo que finalmente nos decantamos por el método más simple, la selección aleatoria. Por otro lado, cabe destacar que tanto la selección por ruleta como la selección por torneo otorgan una mayor probabilidad de reproducción a los individuos de mayor calidad, hecho que reduce la diversidad.

En cuanto a la probabilidad de mutación, vamos a emplear un valor de $p_m = 0.15$. Se trata de un valor bastante elevado si le comparamos con el 0.05 que normalmente aparece en la literatura. La elección de este valor tan elevado se debe a que durante la fase de experimentación previa percibimos una falta de diversidad si usábamos valores pequeños.

El siguiente parámetro que ajustamos fue el tamaño del archivo que se emplea en la fase de reemplazo. En este caso, su tamaño va a ser un 25% de la población total. En la experimentación previa pudimos observar que aumentar el tamaño archivo producía pequeñas mejoras en los resultados pero incrementaba excesivamente los tiempos de cómputo.

Los parámetros empleados para la búsqueda tabú granular son $\beta := 1.75$, $\beta' := 2.5$, $maxIt := 500$, $maxIterIni := n$, $maxIterDiv := n$, $t_{min} := 5$ y $t_{max} := 10$, donde n es el número de clientes. Los valores para los parámetros anteriores siguen las recomendaciones que Toth & Vigo dan en su artículo [**TV, 03**]. Los únicos parámetros que difieren de las recomendaciones anteriores son β y β', que toman un valor algo superior para aumentar de esta forma el tamaño del grafo disperso y ampliar el espacio de búsqueda.

El último parámetro es la probabilidad de intensificación del algoritmo memético, para la cual se ha escogido un valor del 75%. Se trata de un valor bastante alto, sin embargo, estas intensificaciones son clave para obtener soluciones de alta calidad.

5.2. Estudio de Sinergia

El objetivo de esta sección es exponer los resultados y las conclusiones obtenidas en el estudio de sinergia. Este estudio consiste en realizar un análisis de las componentes que constituyen el algoritmo memético que hemos desarrollado, el cual combina un algoritmo genético, que proporciona mecanismos diversidad, con una búsqueda tabú granular, que aporta técnicas de intensificación. Para ello, evaluaremos cómo se comporta cada una de las componentes por separado, en comparación con el algoritmo memético, y comprobaremos si al combinar ambas componentes se obtienen resultados de mayor calidad que cada una por separado. Además de estudiar la sinergia, estudiaremos la influencia de aportar soluciones heurísticas en las poblaciones iniciales de los algoritmos memético y genético.

Este análisis se va a realizar sobre 5 instancias del conjunto de `Augerat`, conocido en la literatura como conjunto `P` y elaborado en el año 1995. Las instancias escogidas son `P-n50-k10`, `P-n51-k10`, `P-n55-k10`, `P-n76-k5` y `P-n101-k4`, en las que el número que aparece tras n indica el número de clientes, incluyendo al depósito, y el número que se exhibe tras k representa la cantidad de vehículos empleados. Cabe destacar, que en la mayoría de instancias el número de vehículos es un valor prefijado, sin embargo, para nuestro algoritmo es una variable más. Para el estudio de sinergia, siempre hemos obtenido soluciones con el mismo número de vehículos que indica el problema. Dada la carga computacional de los experimentos, las instancias escogidas no son excesivamente grandes, y se han elegido tres instancias pequeñas (`P-n50-k10`, `P-n51-k10`, `P-n55-k10`), una de tamaño intermedio (`P-n76-k5`) y una algo más grande (`P-n101-k4`).

Vamos a obtener la solución para cada instancia a través de 5 algoritmos distintos, y ejecutaremos cada uno de ellos 20 veces. Los algoritmos que vamos a emplear son: algoritmo memético con heurístico (`AMH`), algoritmo memético sin heurístico (`AM`), algoritmo genético con heurístico (`AGH`), algoritmo genético sin heurístico (`AG`) y búsqueda tabú granular (`BTG`).

Para obtener resultados comparables, el experimento comienza lanzando los algoritmos meméticos. Cuandos estos han terminado, se calcula el tiempo medio empleado y el número medio de búsquedas tabú granulares (intensificaciones) que se han realizado. A continuación, se ejecutan los algoritmos genéticos usando como criterio de parada el tiempo medio obtenido. Por último, se lleva a cabo la búsqueda tabú granular, donde cada "ejecución" se corresponde con realizar tantas búsquedas tabú granulares desde individuos aleatorios como el número medio de búsquedas realizadas en los algoritmos meméticos.

Respecto a los resultados obtenidos para cada instancia, estos se han organizado de la siguiente forma. En primer lugar, se indica el número de ciudades (clientes), el coste de la solución óptima, el coste de los heurísticos Nearest Neighbor y de Clarke & Wright sin intensificar y tras aplicarles la búsqueda tabú granular, el tiempo medio y el número medio de intensificaciones. A continuación, se presenta una tabla que incluye para cada uno de los algoritmos empleados, el coste de la mejor y la peor solución obtenidas, el error relativo de la mejor solución obtenida con respecto al óptimo, la media y la desviación típica del coste de las soluciones, el tiempo medio y el número de vehículos empleados en la mejor solución. Cabe destacar que para estas instancias la mejor solución conocida es óptima. Por último, se ha construido un diagrama de cajas que permite visualizar los resultandos de forma más clara.

Finalizamos esta sección presentando las conclusiones extraídas de los experimentos. En primer lugar, mencionar que en líneas generales, los resultados obtenidos mediante

los distintos algoritmos han presentado un comportamiento similar en todas las instancias. Los mejores resultados han sido obtenidos por `AM` y `AMH`, y no han distado mucho del óptimo. En el caso de los algoritmos meméticos, no se han apreciado diferencias significativas entre el uso o no de soluciones heurísticas en la población inicial. Con `BTG` hemos obtenido resultados de buena calidad, sin embargo, se ha percibido una cierta distancia con los resultados obtenidos para el memético (en media, se produce un error relativo del 3.81% con respecto a los resultados de `AMH`). Lo anterior indica que los individuos proporcionados por el algoritmo genético constituyen mejores puntos de partida para las búsquedas que individuos generados aleatoriamente. Respecto a `AG` y `AGH`, los resultados obtenidos distan bastante del óptimo y de las soluciones aportadas por el memético y la búsqueda local. Por otro lado, se han apreciado diferencias sustanciales entre el uso o no de soluciones heurísticas. Cuando no se han usado soluciones heurísticas, el algoritmo genético (`AG`) presenta resultados pobres, indicando que no ha dispuesto de tiempo suficiente. En cambio, cuando se han incluido las soluciones heurísticas (`AGH`) los resultados han mejorado; sin embargo, la mejor solución coincide con la del heurístico de Clarke & Wright, lo que indica que la mejor solución no ha evolucionado y es la misma que en la primera generación.

Teniendo en cuenta lo dicho en el párrafo previo, podemos decir que existe un fenómeno de sinergia entre el algoritmo genético y la búsqueda tabú granular, hecho que justifica el uso del algoritmo memético.

P-n50-k10 - Las características y los resultados experimentales obtenidos para la instancia P-n50-k10 son:

- **Número de ciudades:** 49.
- **Óptimo:** 696 con 10 vehículos.
- **Heurístico Nearest Neighbor:** 923.
- **Heurístico Nearest Neighbor (con intensificación):** 810.
- **Heurístico de Clarke & Wright:** 743.
- **Heurístico de Clarke & Wright (con intensificación):** 741.
- **Tiempo medio:** 60 minutos.
- **Número medio de intensificaciones:** 3311.

	Mejor valor	Error relativo (%)	Media mejor valor	Desviación típica	Peor valor	Vehículos
AMH	697	0.14	709.30	4.90	717	10
AM	703	1.00	711.00	5.54	723	10
AGH	741	6.46	742.10	0.99	743	10
AG	868	24.71	893.50	20.40	931	10
BTG	728	4.60	744.25	6.69	754	10

TABLA 5.1. Estadísticas obtenidas de los resultados experimentales realizados para la instancia P-n50-k10.

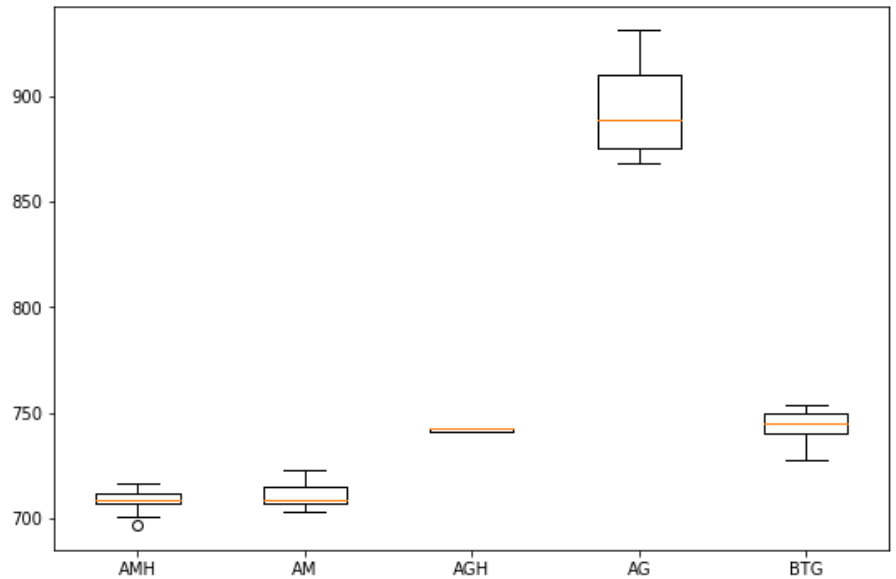

FIGURA 5.1. Diagrama de cajas de los resultados obtenidos para la instancia P-n50-k10.

P-n51-k10 - Las características y los resultados experimentales obtenidos para la instancia P-n51-k10 son:

- **Número de ciudades:** 50.
- **Óptimo:** 741 con 10 vehículos.
- **Heurístico Nearest Neighbor:** 1078.
- **Heurístico Nearest Neighbor (con intensificación):** 890.
- **Heurístico de Clarke & Wright:** 782.
- **Heurístico de Clarke & Wright (con intensificación):** 782.
- **Tiempo medio:** 76 minutos.
- **Número medio de intensificaciones:** 4086.

	Mejor valor	Error relativo (%)	Media mejor valor	Desviación típica	Peor valor	Vehículos
AMH	746	0.67	750.50	3.74	760	10
AM	743	0.27	749.90	3.42	757	10
AGH	782	5.53	782.00	0.00	782	10
AG	909	22.67	931.80	17.77	966	10
BTG	774	4.45	786.25	7.93	801	10

TABLA 5.2. Estadísticas obtenidas de los resultados experimentales realizados para la instancia P-n51-k10.

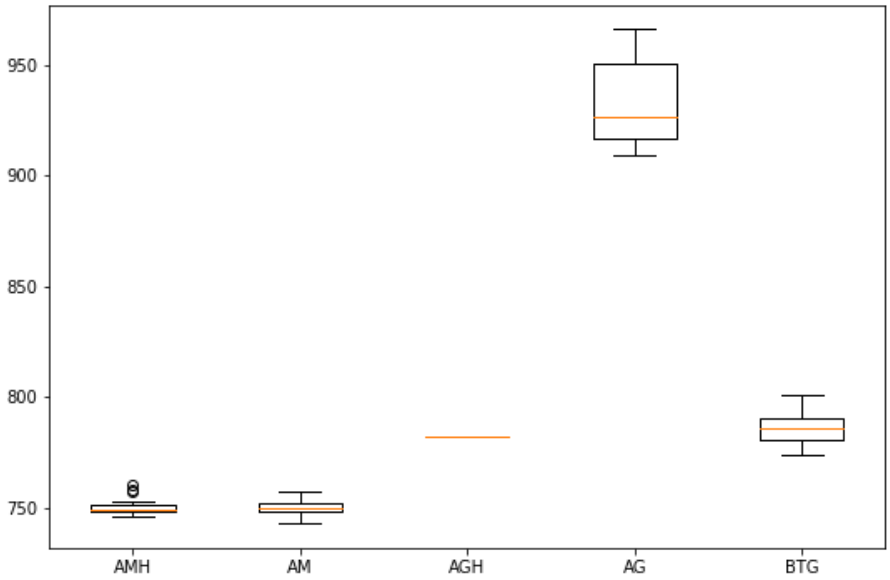

FIGURA 5.2. Diagrama de cajas de los resultados obtenidos para la instancia P-n51-k10.

P-n55-k10 - Las características y los resultados experimentales obtenidos para la instancia P-n55-k10 son:

- **Número de ciudades:** 54.
- **Óptimo:** 694 con 10 vehículos.
- **Heurístico Nearest Neighbor:** 1009.
- **Heurístico Nearest Neighbor (con intensificación):** 805.
- **Heurístico de Clarke & Wright:** 732.
- **Heurístico de Clarke & Wright (con intensificación):** 724.
- **Tiempo medio:** 100 minutos.
- **Número medio de intensificaciones:** 4255.

	Mejor valor	Error relativo (%)	Media mejor valor	Desviación típica	Peor valor	Vehículos
AMH	696	0.28	705.00	4.60	714	10
AM	701	1.00	708.05	4.21	717	10
AGH	732	5.48	732.00	0.00	732	10
AG	877	26.36	922.95	25.77	956	10
BTG	721	3.89	732.30	6.33	745	10

TABLA 5.3. Estadísticas obtenidas de los resultados experimentales realizados para la instancia P-n55-k10.

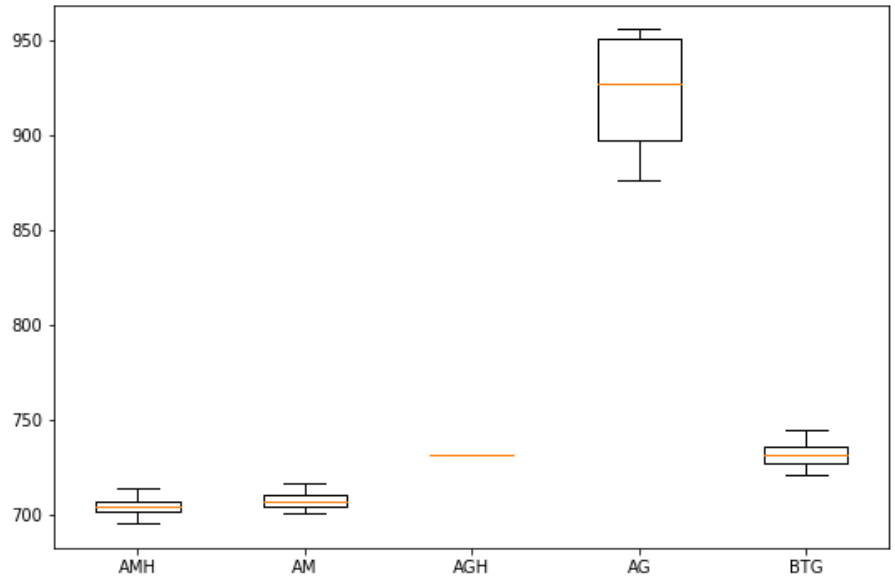

FIGURA 5.3. Diagrama de cajas de los resultados obtenidos para la instancia P-n55-k10.

P-n76-k5 - Las características y los resultados experimentales obtenidos para la instancia `P-n76-k5` son:

- **Número de ciudades:** 75.
- **Óptimo:** 627 con 5 vehículos.
- **Heurístico Nearest Neighbor:** 826.
- **Heurístico Nearest Neighbor (con intensificación):** 807.
- **Heurístico de Clarke & Wright:** 687.
- **Heurístico de Clarke & Wright (con intensificación):** 682.
- **Tiempo medio:** 148 minutos.
- **Número medio de intensificaciones:** 6327.

	Mejor valor	Error relativo (%)	Media mejor valor	Desviación típica	Peor valor	Vehículos
AMH	634	1.11	643.30	7.20	663	5
AM	629	0.31	642.90	6.51	657	5
AGH	687	9.57	687.00	0.00	687	5
AG	1128	79.90	1217.35	53.40	1312	5
BTG	654	4.30	669.65	5.76	678	5

TABLA 5.4. Estadísticas obtenidas de los resultados experimentales realizados para la instancia `P-n76-k5`.

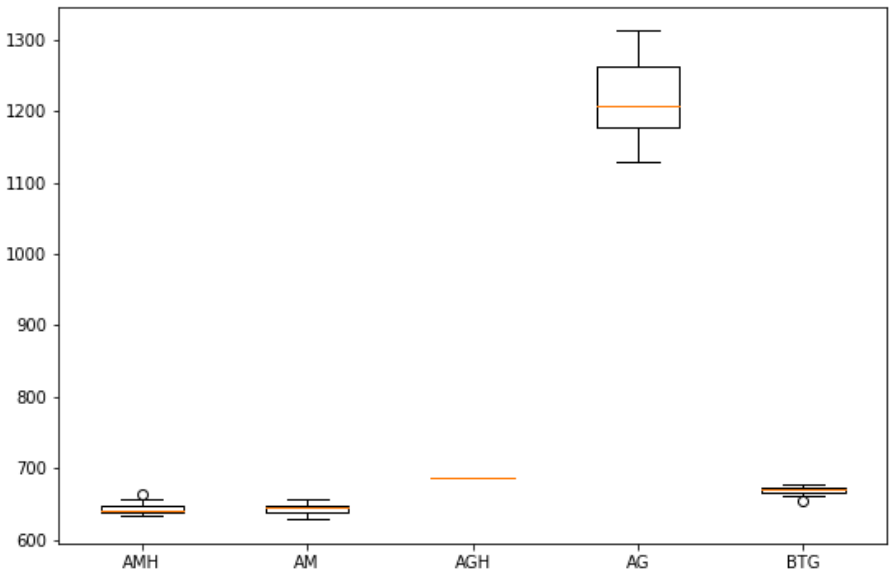

FIGURA 5.4. Diagrama de cajas de los resultados obtenidos para la instancia `P-n76-k5`.

P-n101-k4 - Las características y los resultados experimentales obtenidos para la instancia P-n101-k4 son:

- **Número de ciudades:** 100.
- **Óptimo:** 681 con 4 vehículos.
- **Heurístico Nearest Neighbor:** 1084.
- **Heurístico Nearest Neighbor (con intensificación):** 901.
- **Heurístico de Clarke & Wright:** 752.
- **Heurístico de Clarke & Wright (con intensificación):** 752.
- **Tiempo medio:** 396 minutos.
- **Número medio de intensificaciones:** 8557.

	Mejor valor	Error relativo (%)	Media mejor valor	Desviación típica	Peor valor	Vehículos
AMH	684	0.40	692.70	5.34	705	4
AM	685	0.59	694.70	5.06	702	4
AGH	751	10.28	753.85	0.65	754	4
AG	1540	126.14	1696.15	74.98	1828	4
BTG	712	4.55	718.67	5.31	725	4

TABLA 5.5. Estadísticas obtenidas de los resultados experimentales realizados para la instancia P-n101-k4.

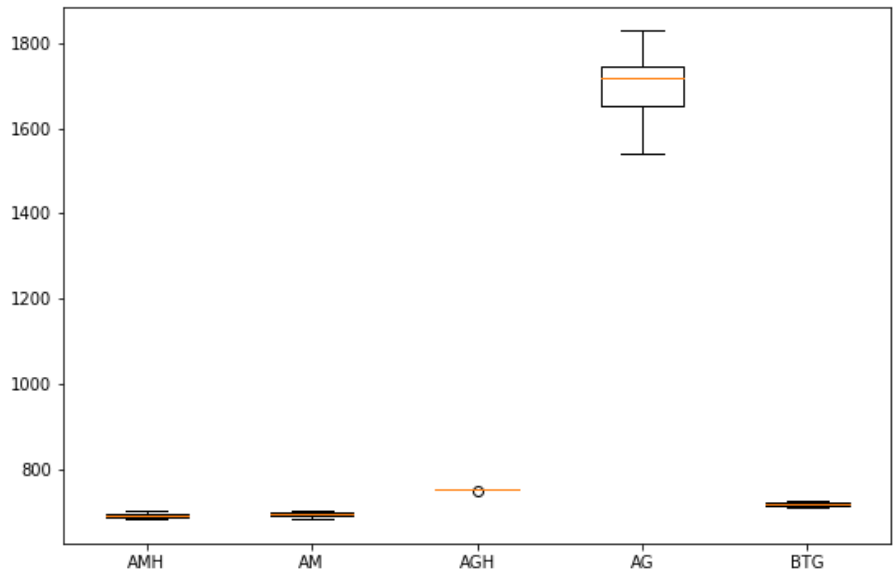

FIGURA 5.5. Diagrama de cajas de los resultados obtenidos para la instancia P-n101-k4.

5.3. Instancias de Augerat, Christofides, Mingozzi & Toth

El objetivo de esta sección es estudiar la calidad del algoritmo memético que hemos desarrollado. Para ello, se analizarán los resultados experimentales obtenidos al aplicar el algoritmo memético a ciertos conjuntos de instancias de uso común en la literatura.

En este caso, hemos empleado dos conjuntos de instancias. El primero de ellos es el de `Augerat`. Este conjunto contiene una amplia variedad de instancias, con un rango de clientes que va desde 15 hasta 100, lo que nos permite ver la evolución de las estadísticas que estamos extrayendo a medida que aumenta el tamaño de los problemas. El segundo conjunto de instancias que vamos a utilizar es el de `Christofides, Mingozzi & Toth`, conocido en la literatura como conjunto `CMT` y elaborado en el año 1979. La elección de este conjunto se debe a que lo emplea Prins [**P, 04**] en sus experimentos para analizar su algoritmo genético que utiliza búsquedas locales como operador de mutación, que es el método de la literatura que más se asemeja al desarrollado en este trabajo. Cabe destacar, que únicamente hemos realizado experimentos con las instancias `CMT1`, `CMT2`, `CMT3`, `CMT4`, `CMT5` y `CMT11` del conjunto de `Christofides, Mingozzi & Toth`. Esto se debe a que las instancias `CMT6`, `CMT7`, `CMT8`, `CMT9`, `CMT10` y `CMT13` son versiones de las instancias `CMT1`, `CMT2`, `CMT3`, `CMT4`, `CMT5` y `CMT11` respectivamente, donde se impone una longitud máxima de ruta, algo que no contempla el algoritmo desarrollado. Por otro lado, en las instancias `CMT12` y `CMT14`, además de imponer una longitud máxima de ruta, se impone un número máximo de vehículos, cantidad que es variable en nuestro algoritmo.

En total, se han obtenido resultados para 29 instancias, los cuales se recogen en la Tabla 5.6. Para cada una de las instancias, se incluye la solución óptima, el número de clientes, la capacidad máxima de los vehículos (Q), el número de vehículos empleados en la solución óptima, el coste de la mejor y la peor solución obtenidas, el error relativo de la mejor solución obtenida con respecto al óptimo, la media y la desviación típica del coste de las soluciones, el tiempo medio y el número de vehículos empleados en la mejor solución obtenida. Señalar que para todas las instancias se ha encontrado en la literatura el valor óptimo para un número fijo de vehículos. En nuestro caso, salvo para las instancias `P-n22-k8`, `P-n50-k8` y `P-n55-k15`, el algoritmo memético ha obtenido una solución con el mismo número de vehículos que el óptimo encontrado en la lietratura. Para los tres casos anteriores, se han obtenido soluciones con más vehículos pero con un coste menor (marcados en la Tabla 5.6 con un asterisco *).

En base a los resultados obtenidos, podemos decir que el algoritmo memético que hemos desarrollado es competitivo. Me gustaría destacar que en 12 (marcadas en negrita en la Tabla 5.6) de las 29 instancias se ha alcanzado el óptimo y que, además, en el resto de instancias el mejor valor obtenido se queda cerca del óptimo. Únicamente en 3 de las 29 instancias se obtiene un error relativo superior al 1%. Por otro lado, los resultados obtenidos para la media y la desviación típica indican que no existen grandes diferencias entre las distintas ejecuciones. Finalmente, comentar que los tiempos de ejecución han sido en general elevados; esto se debe a las limitaciones físicas del hardware y, sobre todo, al lenguaje de programación empleado, que como ya se ha comentado resulta adecuado para un desarrollo ágil de prototipos, sacrificando la rapidez.

Instancia	Óptimo	Clientes	Q	Vehículos óptimo	Mejor valor	Error relativo (%)	Media mejor valor	Desviación típica	Peor valor	Tiempo medio	Vehículos
P-n16-k8	450	15	35	8	**450**	0.00	450.00	0.00	450	1.05	8
P-n19-k2	212	18	160	2	**212**	0.00	212.00	0.00	212	3.27	2
P-n20-k2	216	19	160	2	**216**	0.00	216.00	0.00	216	4.13	2
P-21-k2	211	20	160	2	**211**	0.00	211.00	0.00	211	4.78	2
P-n22-k2	216	21	160	2	**216**	0.00	216.00	0.00	216	5.46	2
P-n22-k8	603	21	3000	8	590*	-	590.00	0.00	590	2.20	9
P-n23-k8	529	22	40	8	**529**	0.00	529.00	0.00	529	3.28	8
P-n40-k5	458	39	140	5	**458**	0.00	458.00	0.00	458	32.78	5
P-n45-k5	510	44	150	5	**510**	0.00	510.00	0.00	510	46.40	5
P-n50-k7	554	49	150	7	**554**	0.00	558.31	2.34	562	73.80	7
P-n50-k8	631	49	120	8	629*	-	635.25	3.09	640	72.31	9
P-n50-k10	696	49	100	10	697	0.10	705.85	4.19	717	68.75	10
P-n51-k10	741	50	80	10	746	0.67	750.50	3.75	760	78.36	10
P-n55-k7	568	54	170	7	570	0.30	575.15	2.91	580	100.40	7
P-n55-k10	694	54	115	10	696	0.28	701.42	2.46	706	105.01	10
P-n55-k15	989	54	70	15	945*	-	948.80	2.21	955	73.60	16
P-n60-k10	744	59	120	10	745	0.10	755.05	6.99	773	131.92	10
P-n60-k15	968	59	80	15	971	0.30	974.30	2.91	981	108.61	15
P-n65-k10	792	64	130	10	**792**	0.00	800.75	7.71	814	174.57	10
P-n70-k10	827	69	135	10	828	0.10	844.05	9.01	855	223.40	10
P-n76-k4	593	75	350	4	**593**	0.00	601.88	4.18	608	334.72	4
P-n76-k5	627	75	280	5	634	1.11	643.30	7.20	663	148.47	5
P-101-k4	681	100	400	4	684	0.40	692.70	5.34	705	396.02	4
CMT1	524.62	50	160	5	**524.62**	0.00	524.61	0.00	524.61	23.85	5
CMT2	835.36	75	140	10	839.98	0.50	851.90	6.22	867.07	100.36	11
CMT3	826.14	100	200	8	831.89	0.70	836.54	2.76	841.98	249.85	8
CMT4	1028.42	150	200	12	1049.41	2.04	1063.68	6.45	1076.76	458.47	12
CMT5	1291.29	199	200	17	1348.66	4.44	1374.08	7.01	1378.21	1171.05	17
CMT11	1042.12	120	200	11	1046.03	0.38	1046.83	0.27	1046.93	331.35	7

TABLA 5.6. Resultados obtenidos con el algoritmo memético.

CAPÍTULO 6

Conclusiones y Trabajo Futuro

> The question of whether a computer can think is no more interesting than the question of whether a submarine can swim. [1]
>
> **Edsger W. Dijkstra**

Índice

El objetivo de este último capítulo es exponer las conclusiones (6.1) obtenidas en este TFG y describir algunas de las posibles líneas de trabajo futuro (6.2).

6.1. Conclusiones

Para poder extraer conclusiones acerca del trabajo, conviene recordar los objetivos planteados al comienzo. En este caso, teníamos dos objetivos principales. El primero de ellos era elaborar un algoritmo memético competitivo para el `CVRP` y el segundo era comprobar la existencia de un efecto de sinergia entre el algoritmo genético y las búsquedas locales, en este caso, búsquedas tabú granulares.

En cuanto al primer objetivo, la construcción del algoritmo memético se realizó en varias etapas. En primer lugar, se diseñó y se implementó el algoritmo genético (2.3), el cual ha sido ampliamente probado y modificado para favorecer la diversidad. En segundo lugar, se desarrolló la búsqueda local granular (2.4.2), que emplea vecindades `2-opt` y `2-opt*`. Además, se llevó a cabo una amplia experimentación para ajustar los parámetros de la búsqueda. Por último, se combinaron los dos procedimientos anteriores para generar el algoritmo memético. Podemos decir que, a la vista de los resultados obtenidos, se trata de un algoritmo competitivo, ya que en la mayoría de los casos el mejor valor producido por nuestro algoritmo se encuentra cerca del óptimo. Cabe destacar que el algoritmo desarrollado no es más que un prototipo, por lo que los tiempos de ejecución mejorarán sustancialmente si lo implementamos usando otro lenguaje de programación más potente.

En relación al segundo de los objetivos, los resultados indican que, al menos para las instancias que hemos empleado, existe un efecto de sinergia entre el algoritmo genético y la búsqueda tabú granular. Es importante señalar que la existencia de esta sinergia justifica el uso del algoritmo memético frente al algoritmo genético o a la búsqueda tabú granular por separado.

Un subobjetivo que también mencionábamos al principio era conocer la influencia de introducir soluciones heurísticas en las poblaciones iniciales de los algoritmos genético y memético. En el caso del algoritmo genético, pudimos comprobar que los heurísticos

[1] La pregunta de si un computador puede pensar no es más interesante que la pregunta de si un submarino puede nadar.

condicionan en gran medida la mejor solución proporcionada por el algoritmo y, por otra parte, que es necesario proporcionar más de una solución heurística para evitar convergencias prematuras. Respecto al algoritmo memético, no se pudieron apreciar diferencias significativas durante los experimentos.

Para finalizar, destacar que para la elaboración de este trabajo ha sido necesaria una comprensión global de la disciplina tratada. Por otra parte, mencionar que este trabajo consiste en un proyecto original de las tecnologías específicas de la Ingeniería en Informática. En particular, se han sintetizado e integrado competencias de la Mención de Computación, tal y como se indica en la competencia específica del grado CE25.

6.2. Trabajo Futuro

Dado que actualmente solo se dispone de un prototipo, el siguiente paso será crear un software más eficiente escrito en un lenguaje de programación más potente como `C++`, para poder lanzar experimentaciones más extensas.

Uno de los puntos que habría que estudiar con más detalle es el fenómeno de la diversidad, ya que como hemos visto en la Subsección 2.3.3 resulta clave a la hora de diseñar algoritmos que emplean poblaciones.

Por otro lado, sería conveniente emplear otro tipo de movimientos, como el `Or-opt` o el λ-intercambio de Osman, en la búsqueda tabú granular y comparar los resultados con los obtenidos para `2-opt` y `2-opt*`. Respecto a las búsquedas locales, sería interesante investigar si estas pueden adaptarse a la topología de los problemas para mejorar su rendimiento.

Otra posible línea de trabajo sería adaptar y aplicar el algoritmo memético a otras variantes del `CVRP`. Desde mi punto de vista, las variantes más difíciles pero a la vez más fascinantes son las estocásticas, en las que algunos de los parámetros, como los costes de las aristas o las demandas de los clientes, son variables aleatorias. Resultaría de gran interés poder ver los resultados que produce el algoritmo memético ante instancias estocásticas.

Bibliografía

[B, 17] Balachandran, A. (2017). "Software Architecture with Python". Packt Publishing.

[B, 83] Beasley, J. E. (1983). "Route first-cluster second methods for vehicle routing". *Omega*, *11*(4), 403-408.

[B, 03] Bodenhofer U. (2003). "Genetic algorithms: theory and applications". 2^a Edición. Linz, Austria, Johannes Kepler Universität.

[CAGRA, 15] Caceres-Cruz, J., Arias, P., Guimarans, D., Riera, D., & Juan, A. A. (2015). "Rich vehicle routing problem: Survey". *ACM Computing Surveys (CSUR)*, *47*(2), artículo 32.

[CLSV, 07] Cordeau, J. F., Laporte, G., Savelsbergh, M. W., & Vigo, D. (2007). "Chapter 6: Vehicle routing". En Barnhart, C., Laporte, G. (Eds.), *Handsbooks in Operations Research and Management Science*, *14*, 367-428.

[CLV, 01] Cheung, B. S., Langevin, A., & Villeneuve, B. (2001). "High performing evolutionary techniques for solving complex location problems in industrial system design". *Journal of Intelligent Manufacturing*, *12*(5-6), 455-466.

[CW, 64] Clarke, G., & Wright, J. W. (1964). "Scheduling of vehicles from a central depot to a number of delivery points". *Operations Research*, *12*(4), 568-581.

[DHMM, 16] Dorling, K., Heinrichs, J., Messier, G. G., & Magierowski, S. (2016). "Vehicle routing problems for drone delivery". *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *47*(1), 70-85.

[DR, 59] Dantzig, G. B., & Ramser, J. H. (1959). "The truck dispatching problem". *Management Science*, *6*(1), 80-91.

[F, 56] Flood, M. M. (1956). "The traveling-salesman problem". *Operations Research*, *4*(1), 61-75.

[G, 06] Gil, N. (2006). "Algoritmos genéticos". Apuntes del curso Algoritmos Genéticos de la Universidad Nacional Medellin, Colombia. Recuperado de: http://www.monografias.com/trabajos-pdf/algoritmos-geneticos/algoritmos-geneticos.pdf

[G, 86] Glover, F. (1986). "Future paths for integer programming and links to artificial intelligence". *Computers & Operations Research*, *13*(5), 533-549.

[G, 89] Goldberg, D. E. (1989). "Genetic Algorithms in Search. Optimization and Machine Learning". Addison-Wesley. Reading, MA.

[GP, 19] Gendreau, M. & Potvin, J. Y. (2019). "Tabu Search". En Gendreau, M. & Potvin, J. Y. (Eds.), *Handbook of Metaheuristics. International Series in Operations Research & Management Sciece*, *272*. Springer, Cham.

[H, 75] Holland, J. H. (1975). "Adaptation in natural and artificial systems: an introductory analysis with application to biology, control and artificial intelligence". Ann Arbor, University of Michigan Press.
metodologías y ciclos de vida". Documento de trabajo del Laboratorio Nacional de Calidad del Software.

[ITV, 14] Irnich, S., Toth, P., & Vigo, D. (2014). "Chapter 1: The family of vehicle routing problems". En Toth P., & Vigo D. (Eds.), *Vehicle Routing: Problems, Methods, and Applications*, 2^a Edición, 1-33. Society for Industrial and Applied Mathematics.

[L, 19] Luashchuk, A. (2019). "8 Reasons Why Python is Good for Artificial Intelligence and Machine Learning". Recuperado de: https://djangostars.com/blog/why-python-is-good-for-artificial\-intelligence-and-machine-learning/I

[LB, 03] Larman, C., & Basili, V. R. (2003). "Iterative and incremental developments. a brief history". *Computer*, *36*(6), 47-56.

[LCHCL, 14] Lin, C., Choy, K. L., Ho, G. T., Chung, S. H., & Lam, H. Y. (2014). "Survey of green vehicle routing problem: past and future trends". *Expert Systems with Applications*, *41*(4), 1118-1138.

[LK, 73] Lin, S., & Kernighan, B. W. (1973). "An effective heuristic algorithm for the traveling-salesman problem". *Operations Research*, *21*(2), 498-516.

[LKMID, 99] Larrañaga, P., Kuijpers, C. M. H., Murga, R. H., Inza, I., & Dizdarevic, S. (1999). "Genetic algorithms for the travelling salesman problem: A review of representations and operators". *Artificial Intelligence Review*, *13*(2), 129-170.

[LS, 02] Laporte, G., & Semet, F. (2002). "Classical heuristics for the capacitated VRP". En Toth P., & Vigo D. (Eds.), *The Vehicle Routing Problem*. Society for Industrial and Applied Mathematics.

[LPPMS, 16] Labadie, N., Prins, C., Prodhon, C., Monmarché, N., & Siarry, P. (2016). "Metaheuristics for Vehicle Routing Problems". ISTE Limited.

[M, 89] Moscato, P. (1989). "On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms". *Technical Report Caltech Concurrent Computation Program, Report 826*.

[MF, 04] Michalewicz, Z., & Fogel, D. B. (204). "How to solve it: modern heuristics". 2^a Edición. Springer Science & Business Media.

[O, 76] Or, I. (1976). "Traveling salesman-type combinatorial problems and their relation to the logistics of blood banking". PhD thesis (Department of Industrial Engineering and Management Science, Northwestern University).

[O, 93] Osman, I. H. (1993). "Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem". *Annals of Operations Research*, *41*(4), 421-451.

[P, 04] Prins, C. (2004). "A simple and effective evolutionary algorithm for the vehicle routing problem". *Computers & Operations Research*, *31*(12), 1985-2002.

[PLP, 14] Prins, C., Lacomme, P., & Prodhon, C. (2014). "Order-first split-second methods for vehicle routing problems: A review". *Transportation Research Part C: Emerging Technologies*, *40*, 179-200.

[PLR, 09] Prins, C., Labadi, N., & Reghioui, M. (2009). "Tour splitting algorithms for vehicle routing problems". *International Journal of Production Research*, *47*(2), 507-535.

[PPPU, 17] Pecin, D., Pessoa, A., Poggi, M., & Uchoa, E. (2017)." Improved branch-cut-and-price for capacitated vehicle routing". *Mathematical Programming Computation*, *9*(1), 61-100.

[STV, 14] Semet, F., Toth, P., & Vigo, D. (2014). "Chapter 2: Classical exact algorithms for the capacitated vehicle routing problem". En Toth P., & Vigo D. (Eds.), *Vehicle Routing: Problems, Methods, and Applications*, 2^a Edición, 37-57. Society for Industrial and Applied Mathematics.

[T, 09] Talbi, E. G. (2009). "Metaheuristics: from design to implementation". John Wiley & Sons.

[TV, 03] Toth, P., & Vigo, D. (2003). "The granular tabu search and its application to the vehicle-routing problem". *Informs Journal on Computing*, *15*(4), 333-346.

[VD, 95] Van Rossum, G., & Drake Jr, F. L. (1995). "Python tutorial". Amsterdam, Netherlands: Centrum voor Wiskunde & Informatica.

[VCGP, 15] Vidal, T., Crainic, T. G., Gendreau, M., & Prins, C. (2015). "Timing problems and algorithms: Time decisions for sequences of activities". *Networks*, *65*(2), 102-128.

[ZBB, 10] Zäpfel, G., Braune, R., & Bögl, M. (2010). "Metaheuristic search concepts: A tutorial with applications to production and logistics". Springer-Verlag, Berl. Heidelb.