UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería en Organización Industrial

La metaheurística de Búsqueda Tabú aplicada al problema de Enrutamiento de Vehículos

Autor:

Bodas López, Raquel

Tutor:

Pérez Rodríguez, María Teresa
Dpto. Matemática Aplicada

Valladolid, Julio de 2017

Agradecimientos

A mi tutora María Teresa, por hacer que este trabajo haya sido posible, por su paciencia y por el tiempo dedicado. Pero sobre todo por sus ganas de enseñar.

A mi familia y amigos por haberme apoyado en todo momento y creer en mí cuando más lo necesitaba.

A mis profesores y toda la comunidad de la Universidad de Valladolid que han hecho que este momento se lograse.

Resumen:

En el presente trabajo se lleva a cabo el estudio del problema de enrutamiento de vehículos (conocido como VRP) repasando sus variantes más significativas. Este problema de optimización combinatoria, de gran interés en la Industria, es difícil de tratar computacionalmente. Por ello para su resolución se han venido empleado muy diferentes técnicas, siendo las metaheurísticas, y en particular la búsqueda tabú, de las más exitosas. Por ello, se incluye el estudio de los algoritmos utilizados en la resolución del VRP, con especial hincapié en los de búsqueda tabú. Se ha seleccionado la búsqueda tabú granular, junto con la heurística de Clark y Wright, para la presentación de una explicación pormenorizada detallando su estructura paso a paso aplicada al VRP clásico y la resolución de un ejemplo, para su mejor comprensión. Además, la búsqueda tabú granular se ha implementado en Python y, con el programa creado, se han resuelto varios casos.

Palabras clave:

Enrutamiento de vehículos, técnicas heurísticas, método de Clark & Wright, búsqueda tabú granular, Python

ÍNDICE

INTRODUCCIÓN 9

Capítulo 1 11

Optimización 11

1.1. Introducción 11
1.2. Historia de la optimización 14
1.3. Tipos de optimización 16
1.4. Optimización combinatoria y métodos de resolución 18

Capítulo 2 21

El problema de enrutamiento de vehículos (VRP) 21

2.1. Generalidades 21
2.2. Problema VRP clásico 22
2.2.1. Formulación matemática 23
2.2.2. Problema VRP con flota heterogénea 25
2.2.3. Otras variantes del problema VRP clásico 26
2.3. Problema VRP con ventanas de tiempo 27
2.4. Problema de rutas de vehículos con inventario 29
2.4.1. Problema IRP determinístico 29
2.4.2. Problema IRP estocástico 30

Capítulo 3 31

Métodos utilizados para resolver VRP 31

3.1. Métodos de resolución para el VRP clásico 31
3.1.1. Algoritmos exactos para el VRP 31
3.1.2. Heurísticas clásicas para el VRP 32
3.1.3. Metaheurísticas 40
3.2. Métodos de resolución para el VRPTW 45
3.3. Métodos de resolución para el problema IRP 45

Capítulo 4 47

Búsqueda tabú 47

4.1. La búsqueda local 47
4.2. Generalidades 48
4.3. Algoritmo de la búsqueda tabú 49
4.4. Atributos de los movimientos y restricciones tabú 51
4.5. Tamaño de la lista tabú 51
4.6. Memoria a medio plazo (Intensificación de la búsqueda) 52
4.7. Memoria a largo plazo (Diversificación de la búsqueda) 52
4.8. La búsqueda tabú aplicada al problema VRP 53
4.8.1. Implementación de Willard 54
4.8.2. Algoritmo de Osman 55
4.8.3. Algoritmo Ruta Tabú 55
4.8.4. Algoritmo de Taillard 61
4.8.5. Procedimiento de memoria adaptativa 61
4.8.6. Algoritmo de Xu y Kelly 62
4.8.7. Algoritmo Cadena Tabú 63

4.8.8. Algoritmo Flor ... 63
4.8.9. Algoritmo de la búsqueda tabú granular ... 64
4.8.10. Algoritmo de búsqueda tabú unificada ... 64

Capítulo 5 ... 67

Búsqueda tabú granular aplicada al VRP ... 67

5.1. Introducción ... 67
5.2. El problema ... 68
5.3. Movimientos de *k*-Intercambio ... 68
5.4. El método ... 70
5.5. Resultados computacionales ... 74

Capítulo 6 ... 79

Aplicación de métodos al problema VRP ... 79

6.1. Búsqueda tabú granular ... 79
6.2. Método de Clarke & Wright ... 98

Capítulo 7 ... 107

Programación de la búsqueda tabú granular aplicada al problema VRP ... 107

7.1. El lenguaje Python ... 107
7.2. Búsqueda tabú granular en Python ... 109
7.3. Módulo de usuario BTGranular ... 113
7.4. Resultados computacionales ... 117

Conclusiones y futuras líneas de continuación ... 125

Bibliografía ... 127

Anexo ... 133

INTRODUCCIÓN

En el presente trabajo se estudia el problema de enrutamiento de vehículos (o Vehicule Routing Problem en inglés), sus variantes y diferentes métodos de resolución para este problema. El problema de enrutamiento de vehículos es difícil de resolver numéricamente ya que es de tipo NP-Duro, por lo que en el trabajo nos fijamos en los métodos de búsqueda tabú. En concreto nos centramos en la búsqueda tabú granular que se estudia en detalle y se programa en Python. La memoria del trabajo se divide en siete capítulos, a través de los cuales se guía al lector para que llegue a entender completamente el problema del enrutamiento de vehículos y la aplicación de la búsqueda tabú granular a este problema.

La memoria se estructura de la siguiente manera: en el primer capítulo se introducen ciertos conceptos básicos relacionados con el campo de la optimización. Se comienza por explicar la finalidad de la optimización y su historia. Además, se hace una clasificación de los problemas de optimización y se introducen los problemas de optimización combinatoria.

En el segundo capítulo se expone el problema del enrutamiento de vehículos y las variantes más estudiadas de este problema. Para cada una de estas variantes se aporta su formulación matemática y las características más importantes. En el siguiente capítulo se hace una revisión de los distintos métodos disponibles para la resolución de las diferentes variantes del problema de enrutamiento de vehículos. Esta revisión incluye técnicas exactas, heurísticas clásicas y metaheurísticas.

En el Capítulo 4 se presenta la heurística de búsqueda local. A continuación se expone la búsqueda tabú y las variantes más importantes de la búsqueda tabú aplicada al problema de enrutamiento de vehículos. En concreto en el Capítulo 5 se explica detalladamente el funcionamiento de la búsqueda tabú granular aplicada al problema de enrutamiento de vehículos. Se expone el método de intercambios de arcos mediante el cual se generan los entornos, la metodología en sí de la búsqueda tabú granular y los resultados que ofrece esta variante.

En el Capítulo 6 se ilustra mediante un ejemplo realizado a mano el funcionamiento de la búsqueda tabú granular para un problema propuesto. Así mismo se ilustra como generar un solución inicial mediante el método de Clarke & Wright. En el último capítulo se expone cómo se ha realizado la programación de la búsqueda tabú granular en Python y los resultados de la experimentación obtenidos con dicho programa.

MOTIVACIÓN Y OBJETIVOS

En un mundo tan globalizado como en el que vivimos hoy día se hace necesario para las empresas reducir sus costes de distribución. En este contexto, el problema de enrutamiento de vehículos cobra una gran importancia, ya que miles de empresas se enfrentan a este problema diariamente. Una buena planificación de la distribución de los bienes que produce una empresa puede llevar a una reducción considerable de costes.

El problema de enrutamiento de vehículos es un problema muy complejo, y su resolución mediante métodos exactos no es apropiada. Por ello, se han propuesto diferentes técnicas heurísticas y metaheurísticas para su resolución. Este tipo de técnicas conducen generalmente a buenas soluciones pero no aseguran que se alcance la solución óptima.

El objetivo del presente trabajo es hacer una revisión de los diferentes métodos disponibles para la resolución del problema de enrutamiento de vehículos. El estudio se centra en los métodos de búsqueda tabú y en concreto en la variante de la búsqueda tabú granular, así como en el algoritmo de Clarke & Wright que se utiliza como algoritmo auxiliar para obtener una solución inicial para la búsqueda.

Además, se plantea otro objetivo que consiste en la implementación de la búsqueda tabú granular en el potente lenguaje de programación Python. Utilizando el programa elaborado para resolver varios problemas, se pretende evidenciar las ventajas que supone utilizar una técnica de este tipo en la planificación de la distribución.

Capítulo 1

Optimización

1.1. Introducción

Según la Real Academia de la Lengua Española, optimizar consiste en “buscar la mejor manera de realizar una actividad”. Atendiendo a esta definición, la optimización se puede entender como una técnica o instrumento muy potente y versátil, que se puede aplicar a multitud de problemas (Cavazzuti, 2013). Se puede citar, entre otros, la ingeniería o las finanzas como ámbitos donde la optimización es realmente una herramienta útil.

Pero sin ir tan lejos, la optimización también forma parte de nuestro día a día aunque no nos demos cuenta de ello. Esto se debe a que, al plantearnos un objetivo, lo pretendemos conseguir sacando el máximo rendimiento del tiempo invertido y de los recursos empleados. Por ejemplo, para hacer la compra diaria en el barrio, intentamos que el tiempo que invertimos sea el menor posible, y esto lo conseguimos reduciendo al máximo la distancia recorrida entre los distintos comercios.

Así mismo, las empresas, mediante el desarrollo de su actividad, tratan de maximizar sus beneficios, sin olvidarse de minimizar los costes asociados. Es en este contexto donde cobra sentido la optimización aplicada al mundo de la Industria.

Una formulación matemática genérica para un problema de optimización es la siguiente:

optimizar $f(s)$
sujeto a s $\in F$,

donde $f: S \rightarrow R$ es la función objetivo o función de coste y $F \subset S$ se denomina conjunto factible y queda definido por las restricciones del problema.

A continuación se muestra un ejemplo sencillo de problema de optimización con el fin de aclarar determinados conceptos. Supongamos que queremos construir una caja de cartón con la mayor capacidad posible pero con un límite de material utilizado. Las variables que intervienen en este problema son las dimensiones de la caja, es decir, el ancho (x), el alto (y) y el largo (z). Por tanto, la función que define el volumen de la caja será:

$$V\ (x, y, z) =\ x\ y\ z$$

Dicha función es la función objetivo, es decir, la función que en este caso se quiere maximizar.

Consideremos que el área de cartón del que se dispone es A. El área lateral de la caja de cartón es $A(x, y, z) = 2 * (xy + yz + zx)$. Esta limitación es una restricción del problema. Otra restricción consiste en imponer que las variables x, y, z no sean negativas, ya que no tiene sentido.

Por tanto, el problema se formularía de la siguiente manera:

$$\begin{aligned} &\text{Maximizar} \quad x\ y\ z \\ &\text{sujeto a:} \\ &\qquad 2 * (xy + yz + zx) = A \\ &\qquad x, y, z\ \geq 0 \end{aligned}$$

Siguiendo con la formulación matemática propuesta anteriormente, en este problema el subconjunto F se compone de las distintas soluciones que cumplen las restricciones impuestas. La solución óptima del problema será aquella que maximiza la función de coste $V(x, y, z)$. Por tanto, para resolver el problema hay que identificar los valores de las variables x, y, z que maximizan la función objetivo.

Otro aspecto importante relacionado con la optimización es la diferencia entre extremos (máximos o mínimos) locales y globales. Está claro que en las aplicaciones siempre se quiere la mejor solución de entre todas las que verifican las restricciones del problema pero, en la práctica este objetivo no es siempre alcanzable y en muchos casos hay que conformarse con una "buena solución" factible. Un problema que surge a la hora de resolver los problemas de optimización es la aparición de lo que se conoce como extremos locales, que son soluciones óptimas comparadas con los puntos que están en su entorno pero que pueden no ser óptimos globales.

Como vemos, el concepto de óptimo local conlleva otro asociado que es el de entorno. Normalmente la definición de los entornos depende del problema que se esté tratando. Por ejemplo tomemos un conjunto A cualquiera de R^n, y consideremos una función f de A en R y x^* un punto del conjunto A.

Se dice que f tiene un máximo local en el punto x^* si existe una bola $B(x^*, r)$, que en este caso sería el entorno, tal que para todo $x \in B(x^*, r) \cap A,\ f(x) \leq f(x^*)$. Análogamente, se dice que f tiene un mínimo local en el punto x^* si existe una bola $B(x^*, r)$, tal que para todo $x \in B(x^*, r) \cap A,\ f(x) \geq f(x^*)$.

En el caso de los extremos globales la comparación se lleva a cabo con todos los elementos del conjunto. Así la función f tiene un máximo global en x^* si $f(x) \leq f(x^*)$ para todo $x \in A$. Si para todo $x \in A,\ f(x) \geq f(x^*)$ entonces f tiene un mínimo global en x^*.

En la Figura 1.1. se ilustra la diferencia entre extremos globales y locales para una función cualquiera. Se puede observar que el máximo local es un extremo local debido a que es el punto máximo dentro de la bola B(x*,r), marcada en color gris, centrada en el máximo local y de radio r. Lo mismo ocurriría con el mínimo local. Los extremos globales, tanto máximo como mínimo, es evidente que son el mayor punto y el menor punto de la función en todo el conjunto considerado.

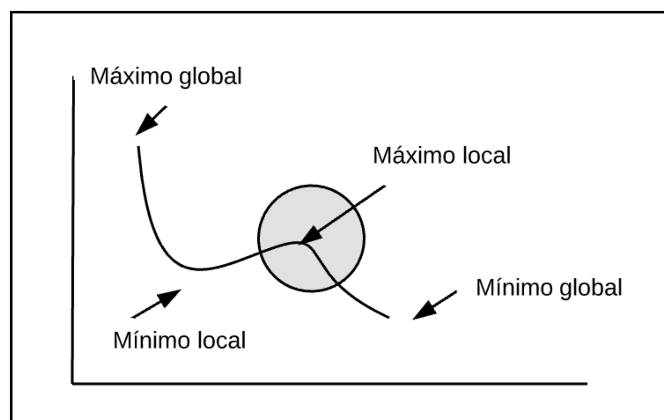


Figura 1.1. Extremos globales y locales

En los problemas de optimización interesa encontrar los extremos globales existentes en el conjunto factible, ya que cuando se pretende maximizar o minimizar algo, los extremos locales no son suficientes. Como se verá más adelante, las técnicas de optimización disponen de mecanismos que evitan caer en extremos locales, de forma que se facilite encontrar los extremos globales buscados.

1.2. Historia de la optimización

En el presente apartado se da una idea general de la evolución de la optimización a lo largo de la historia, siguiendo a (Yang, 2010).

Los problemas de optimización son tan antiguos como la ciencia. De hecho, algunos matemáticos griegos ya se plantearon distintos problemas de optimización. Por ejemplo, hacia el año 300 a.C. Euclides probó que un cuadrado encierra el mayor área posible comparando con los distintos rectángulos que se pueden formar con el mismo perímetro.

Hasta el siglo XX son múltiples los casos en los que se estudiaron la optimización y problemas relacionados. Cabe destacar el planteamiento de Isaac Newton publicado en sus *Principios Matemáticos* en 1687 para minimizar la resistencia de un cuerpo de revolución al movimiento en el seno de un fluido.

Otro caso fue el que propuso Bernoulli en 1696, que consiste en encontrar la curva que conecta dos puntos a distinta altura de forma que un cuerpo cayese siguiendo dicha trayectoria en el menor tiempo posible debido al efecto de la gravedad. También Gauss, alrededor del año 1801, utilizó el método de los mínimos cuadrados para predecir la posición de las órbitas del asteroide Ceres.

Es en el año 1906 cuando el matemático J. Jensen introduce el concepto de convexidad. Dicho concepto gana relevancia asociado a la optimización convexa y al ser aplico a diversas áreas, como por ejemplo, la economía. La optimización convexa es un tipo importante de optimización matemática ya que cualquier óptimo que se encuentre se puede garantizar que es un óptimo global. Muchos de los problemas de optimización planteados hasta el momento se reformularon en términos de la optimización convexa.

En 1917 H. Hancock publicó el primer libro de optimización que lleva por título *"Theory of Minima and Maxima"* y en 1939 L. Kantorovich desarrolló el primer algoritmo de programación lineal.

Otro hito importante en la historia de la optimización fue la invención en 1947 por parte de George Dantzig del método simplex para la programación lineal. Un ejemplo clásico al que se le aplicó por primera vez el método simplex fue el problema de la dieta óptima (Gill, 2007) que consiste en determinar la cantidad de distintos alimentos que satisfagan ciertos requerimientos nutricionales reduciendo el coste de los alimentos utilizados. Dicho problema consta de 9 restricciones y 77 variables y se resolvió con calculadoras manuales.

Durante la Segunda Guerra Mundial se trató de encontrar la solución óptima o al menos cerca del óptimo a problemas muy diversos en cuanto a complejidad. La temática de estos problemas también fue variada, desde redes de comunicación, planificación de proyectos, programación de talleres o planificación del transporte, hasta temas relacionados con gestión. A partir de los años 60 son muchos los estudios que se llevaron a cabo relacionados con la optimización, por lo que no es posible recoger todos los algoritmos a los que han dado lugar. Por ello, a continuación se procede a hacer una breve descripción de algunas de las heurísticas y metaheurísticas más relevantes desarrollas a partir de entonces. Es importante indicar que este tipo de métodos ha sido posible gracias a los avances que han tenido lugar en la informática y en la construcción de ordenadores con una gran potencia computacional.

Probablemente fue Alan Turing el primero en utilizar un algoritmo heurístico durante la Segunda Guerra Mundial para descifrar mensajes codificados por la máquina Enigma. Para ello, un mecanismo electromagnético de criptoanálisis empleaba un algoritmo heurístico para buscar entre 10^{22} posibles combinaciones y descifrar el mensaje cifrado por la máquina Enigma. Turing llamó "*heurística"* al método de búsqueda, ya que se esperaba que funcionase en la mayoría de los casos, y definitivamente fue un éxito. Fue en 1948 cuando Turing presentó en un estudio las nuevas ideas sobre la inteligencia de las máquinas y los métodos basados en aprendizaje, las redes neuronales, así como una primera versión de los algoritmos genéticos.

El siguiente paso fue la aparición de los algoritmos evolutivos entre los años 60 y los 70. En primer lugar John Holland y sus colegas desarrollaron los algoritmos genéticos basados en la teoría evolutiva de Darwin. Las principales ideas de los algoritmos genéticos se presentaron en un libro publicado en 1975. Desde entonces, los algoritmos genéticos han tenido éxito resolviendo distintos tipos de problemas de optimización y debido a esto se han llevado a cabo miles de investigaciones sobre dichos algoritmos. Además, una gran cantidad de empresas usan, de manera rutinaria, este tipo de algoritmos para resolver problemas de optimización combinatoria, aplicados a la planificación o la programación de tareas.

Las décadas de los 80 y los 90 fue el período de mayor auge de las técnicas metaheurísticas. El recocido simulado (o Simulating Annealing) fue desarrollado por S. Kirkpatrick, C.D. Gellat y M.P. Vecchi, y es un método de optimización inspirado en el proceso de recocido de los metales que consiste en calentar un sólido a altas temperatura y enfriarlo lentamente permitiendo que alcance el estado de mínima energía para que se solidifique con una estructura cristalizada. Dicho algoritmo incorpora un parámetro que hace las

veces de temperatura y acepta, con una determinada probabilidad, pasos que empeoran la solución, lo que le permite escapar de los óptimos locales.

El primer uso real de una técnica metaheurística se debe probablemente a Fred Glover que propuso la búsqueda tabú en 1986. Esta técnica se describe en detalle en el Capítulo 4.

En 1992 Marco Dorigo describe en su tesis un proceso de optimización basado en colonias de hormigas. Este método se inspira en el comportamiento de las hormigas guiadas por las feromonas que dejan tras su camino.

Con la llegada del siglo XXI la investigación de las técnicas metaheurísticas se acentúa. Debido a esto surgen distintos algoritmos, como la búsqueda armónica propuesta por Zong Woo Geem en 2001, diversos métodos basados en el comportamiento de insectos, como los algoritmos de enjambre de abejas y de la luciérnaga, o de los bancos de peces, como el de optimización de enjambre de partículas.

Como es evidente, son muchos los algoritmos ya desarrollados. De la misma manera, son muchos los algoritmos que se siguen estudiando, por lo que con el paso del tiempo se conseguirán grandes avances en el campo de la optimización.

1.3. Tipos de optimización

Según (Yang, 2010) se pueden clasificar los distintos tipos de problemas de optimización atendiendo a distintos criterios tal y como puede verse en la Figura 1.2.

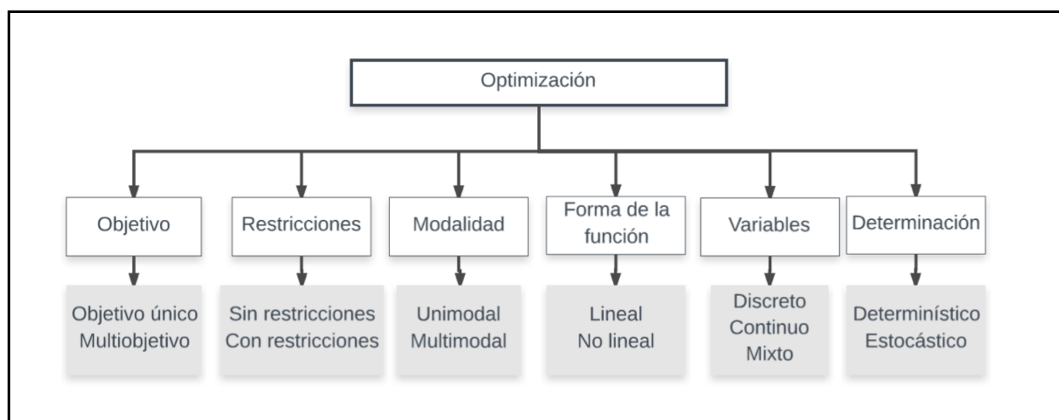


Figura 1.2. Clasificación de los problemas de optimización

Uno de los criterios de clasificación es el número de funciones que se desean optimizar. Siguiendo este criterio los problemas se clasifican en multiobjetivo u objetivo único. Los problemas de objetivo único tienen en cuenta en la optimización a un solo factor. Los problemas multiobjetivo son los que persiguen optimizar más de un factor y son los más comunes en el mundo real. Por ejemplo, a la hora de diseñar un motor para un vehículo se intenta maximizar la eficiencia en el consumo de combustible y minimizar las emisiones y los niveles de ruido. En muchas de las situaciones en las que los problemas son multiobjetivo pueden presentarse conflictos entre ellos.

En términos del número de restricciones existentes, el problema puede ser sin restricciones o con restricciones.

Atendiendo a la forma de la función objetivo, así como a la forma que toman las distintas restricciones, el problema puede ser lineal o no lineal. En caso de que tanto la función objetivo como las restricciones sean lineales, el problema es lineal. Pero el caso más común es aquel en el que la función objetivo o las restricciones no son lineales, y el problema es, por tanto, no lineal.

Otra clasificación divide los problemas en discretos, enteros o mixtos. Los problemas discretos son aquellos en los que todas las variables toman valores discretos, mientras que los problemas enteros son aquellos en los que todas sus variables toman valores enteros. En el caso en el que aparecen variables discretas y enteras, el problema pasa a denominarse mixto. Los problemas de optimización discreta, también son conocidos como problemas de optimización combinatoria, y son el tipo más popular de optimización con problemas tan conocidos como el problema de la mochila, el problema del viajante o el enrutamiento de vehículos.

Otra clasificación hace referencia a los problemas de optimización unimodales y a los problemas multimodales. Para una función objetivo en la que se busca optimizar un solo factor y si solo existe un valle o un pico siendo este el único óptimo global, entonces el problema es unimodal. En este caso el óptimo local, coincide con el óptimo global. Un tipo especial de optimización unimodal es la optimización convexa, ya que la solución óptima global está garantizada. Sin embargo, la mayoría de los problemas de optimización son multimodales y son mucho más difíciles de resolver.

Por último, los problemas se pueden dividir en determinísticos y en estocásticos. Son estocásticos aquellos problemas en los que intervienen variables de las que no se puede tener un valor exacto. Por ejemplo, si en el problema interviene una propiedad medida de un objeto físico, dicha medida no es exacta, sino que tiene cierto ruido producido por la incertidumbre del

proceso de medición. En caso de que esto no ocurra, el problema es determinístico.

1.4. Optimización combinatoria y métodos de resolución

La optimización combinatoria es una rama de la optimización, en la cual existe un conjunto finito de posibles soluciones (Papadimitriou & Steiglitz, 1998). En la mayoría de los casos, los problemas del mundo real involucran tantas variables que, aunque tengan un número finito de posibles soluciones, evaluar todas ellas es prácticamente imposible. Esto se debe a que, al aplicar un método exacto de resolución, el tiempo requerido para evaluar estas posibles soluciones sería excesivo.

Por tanto, un problema de optimización combinatoria, persigue encontrar una solución (puede ser óptima o no serlo) para la función de coste. Para resolver estos problemas se sigue un procedimiento con un número de pasos finito, conocido como algoritmo. Los algoritmos se pueden clasificar en función de su complejidad computacional, que es el tiempo que tarda un algoritmo en resolver un problema determinado. La complejidad se expresa en forma de función y para ello se utiliza la notación $O(f(n))$.

Cada problema tiene un tamaño determinado que se mide mediante el parámetro n. Por ejemplo, en un algoritmo de ordenación, n será el número de elementos que hay que ordenar. Supongamos que la complejidad de un determinado algoritmo de ordenación es $O(n)$, es decir, su complejidad es lineal. Entonces, si el número de elementos a ordenar se duplica, entonces el tiempo necesario de ejecución también será el doble. En cambio, si la complejidad del algoritmo fuese $O(n^2)$ y el número de elementos a ordenar se duplicase, entonces el tiempo de ejecución aumentaría de forma cuadrática.

Volviendo a la clasificación de los algoritmos según su complejidad los algoritmos pueden ser exponenciales y polinómicos. Los algoritmos exponenciales se caracterizan porque su complejidad no puede estar limitada por funciones polinómicas, sino que su complejidad viene determinada por $O(c^n)$ donde c es una constante real mayor que 1. En el caso de los algoritmos polinómicos, la complejidad temporal es $O(p(n))$, donde n es el tamaño del problema y $p(n)$ es una función polinómica de n.

Los problemas de optimización combinatoria se clasifican en problemas tipo P, problemas tipo NP y problemas NP-Duros. Esta clasificación se realiza en función de su complejidad computacional y dichos problemas tienen las siguientes características:

- Problemas tipo P o polinómicos: son problemas que se pueden resolver mediante algoritmos polinómicos, cuya complejidad temporal es O(p(n)).

- Problemas tipo NP o polinómicos no determinísticos: no se conoce ningún algoritmo polinómico para resolver este tipo de problemas. Sin embargo, los algoritmos polinómicos son útiles para este tipo de problema porque sirven para determinar si una solución es factible o no lo es.

- Problemas NP-Duros (o NP-Hard): son problemas muy complejos para los que, a día de hoy, no se ha encontrado ningún algoritmo polinómico que sirva para su resolución. De hecho, algunos investigadores piensan que no existe ningún algoritmo polinómico capaz de resolver este tipo de problema. Desafortunadamente, los problemas NP-Duros son muy comunes en campos muy diversos.

Nos centraremos en los problemas tipo NP-Duros, ya que el problema de enrutamiento de vehículos, que se estudia en el presente trabajo, es de tipo NP-Duro. Como se ha indicado, los algoritmos exactos, tales como la programación lineal o la programación dinámica, son insuficientes cuando se pretende resolver un problema NP-Duro. Por tanto, para enfrentarnos a estos problemas, existen técnicas heurísticas y metaheurísticas de resolución que sí resultan ser eficaces. En general, estos métodos funcionan de forma eficiente y práctica, proporcionando soluciones de calidad, aunque no hay garantía de que un algoritmo funcione para un problema en concreto.

Este tipo de técnicas se basan en la estrategia de prueba y error y el objetivo es encontrar una solución factible que sea suficientemente buena en una escala de tiempo aceptable. A pesar de que estas técnicas llegan a una solución que garantiza que se cumplan unas determinadas restricciones, no aseguran que se llegue a la solución óptima. El ahorro de tiempo conseguido con estas técnicas con respecto a los algoritmos exactos, es posible debido a que buscan la solución final en un subconjunto de las posibles soluciones.

Ahora que ya se ha descrito en qué consiste la optimización y se ha detallado el por qué de la necesidad de las técnicas metaheurísticas para cierto tipo de problemas, se procede en el siguiente capítulo a presentar en detalle el problema del enrutamiento de vehículos.

Capítulo 2

El problema de enrutamiento de vehículos (VRP)

En el presente capítulo se expone el problema del enrutamiento de vehículos comenzando con su versión clásica. Para esta versión clásica se presentan una serie de formulaciones matemáticas del problema y las variantes más estudiadas. A continuación se describe el problema de enrutamiento de vehículos con ventanas de tiempo, así como el problema de enrutamiento de vehículos con inventario. Debido a que son muchos los métodos aplicados para la resolución de las distintas variantes se dedica el capítulo siguiente a describir dichos métodos.

2.1. Generalidades

El problema de enrutamiento de vehículos, también conocido como VRP (Golden B. , 2008), por sus siglas en inglés (Vehicule Routing Problem), es uno de los problemas de optimización combinatoria más estudiados desde que fue propuesto en 1959 por Dantzig y Ramser (Coelho, 2016). Desde entonces, ha tenido lugar una evolución en el diseño de metodologías de resolución, tanto exactas como aproximadas para este problema.

Debido a que es un problema de tipo NP-Duro, actualmente no se conoce ningún algoritmo exacto (Cordeau & Laporte, 2002) capaz de dar una solución óptima cuando se aplica a situaciones reales en las que existen múltiples restricciones, o en caso de haber más de 50 clientes a los que servir. Es por esta razón, que las técnicas heurísticas son las más empleadas a la hora de enfrentarse a un problema de este tipo. Aunque este problema se lleva estudiando más de cinco décadas, en la actualidad sigue siendo un gran desafío.

El problema VRP es un problema real al que se enfrentan las empresas en su día a día ya que, generalmente, un bien consumible no se fabrica en el mismo lugar donde se va a consumir. Esto hace necesario una planificación de la

logística y distribución, de manera que el coste asociado a estas actividades sea el mínimo posible y el beneficio obtenido sea mayor. De hecho, algunos métodos implementados en computador han conseguido un ahorro entre un 5% y un 20% (Cordeau & Laporte, 2002) en costes de distribución.

A continuación, se expone el problema clásico VRP y sus variantes, el problema VRP con ventanas de tiempo y el problema del enrutamiento de vehículos con inventario.

2.2. Problema VRP clásico

El problema VRP clásico es una generalización del problema del viajante, en el cual se busca una ruta con coste mínimo que conecte un número determinado de ciudades. El problema VRP consta de un conjunto de clientes con una determinada demanda, un número de almacenes y una flota de vehículos. El problema consiste en encontrar un conjunto de rutas que minimicen el coste. Estas rutas deben comenzar y terminar en el almacén y cada cliente debe ser visitado exactamente una sola vez por único vehículo para cubrir su demanda.

A menudo el problema VRP se define bajo restricciones de capacidad y de longitud de las rutas. Cuando solo está presente la restricción de capacidad el problema se denomina VRP de capacidad o capacited – VRP en inglés (CVRP). En el caso de que el problema contenga restricciones relativas a la longitud máxima de las rutas, se habla de un problema VRP con restricción en la distancia o DVRP por sus siglas en inglés (Distance contrained VRP)

El problema VRP se define en un grafo completo G = (V,E) donde V es el conjunto de vértices V = $\{v_0, \dots, v_n\}$. Cada vértice $v_i \in V \setminus \{0\}$ representa a un cliente que tiene asociada una demanda no negativa d_i. El vértice v_0 se corresponde con el almacén. Se asocia un coste c_{ij} (no negativo) a cada arco (i, j). Cuando $c_{ij} = c_{ji}$ el problema se denomina simétrico. Se dispone de una flota con un número K de vehículos idénticos, con una capacidad C, que inicialmente se encuentran en el almacén.

Un caso particular del caso simétrico es el problema VRP Euclídeo, en el que cada coste c_{ij} se define como la distancia Euclídea entre dos puntos correspondientes a los vértices v_i y v_j. Recordemos que la distancia Euclidea existente entre dos puntos P_1 y P_2 situados en el plano y definidos por sus coordenadas (x, y) se calcula como:

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2}$$

El problema VRP asimétrico se define también sobre un grafo, pero a diferencia del problema simétrico, el grafo en el problema asimétrico es dirigido. Cuando el problema es asimétrico $c_{ij} \neq c_{ji}$. En este caso las rutas serán dirigidas.

El problema VRP, tanto simétrico como asimétrico, busca un conjunto de K rutas cuyo coste total sea mínimo y cumpla las siguientes condiciones:

a. Cada cliente es visitado exactamente una vez y en una sola ruta.
b. Cada ruta empieza y termina en el almacén.
c. El total de las demandas de los clientes de una ruta no excede la capacidad C del vehículo.
d. La longitud de cada ruta no excede un límite L prefijado. Normalmente se considera velocidad constante y por ello las distancias, el tiempo y el coste son sinónimos.

Por tanto, la solución se puede ver como un conjunto de K rutas que comparten un vértice, que se corresponde con el almacén. Se dice que una ruta es factible si la cantidad total demandada por los consumidores visitados no supera la capacidad del vehículo C y si la longitud de las rutas no supera el límite L prefijado. El coste de una ruta se puede calcular como la suma de los costes de los arcos que componen la ruta.

Por otro lado, cabe destacar que la mayoría de autores que han estudiado este problema, ha empleado un número fijo de vehículos, pero otros autores han considerado el número de vehículos como una variable de decisión. Esto es así porque, las mejores soluciones encontradas, en términos de distancias, no siempre utilizan el menor número de vehículos posible.

2.2.1. Formulación matemática

El problema CVRP se puede formular mediante programación lineal entera de forma que cada arco $e \in E$ tiene asociada una variable x_e que indica el número de veces que el arco e es atravesado por un vehículo en la solución. $r(S)$ representa el número mínimo de vehículos necesarios para servir a los clientes de un subconjunto S. El valor de $r(S)$ se puede determinar resolviendo un problema de carga de contenedores o BPP por sus siglas en inglés (Bin Packing Problem). En este problema de carga de contenedores, se deben almacenar objetos de diferentes volúmenes en un número finito de contenedores, cada uno de ellos con un volumen V y se pretende minimizar el número de contenedores usados. Por tanto, para determinar $r(S)$ se considera que los objetos son los clientes de S, el volumen de los objetos se identifica con la demanda de los clientes y los contenedores son de capacidad C. Finalmente,

para $S \subset V$, $\delta(S) = \{(i,j): i \in S,\ j \notin S \text{ ó } i \notin S,\ j \in S\}$. Si $S = \{i\}$ se denota $\delta(i)$ en lugar de $\delta\{(i)\}$.

Por tanto, la formulación propuesta en (Laporte & Al., 1985) quedaría como sigue:

$$\text{Minimizar} \quad \sum_{e \in E} c_e \, x_e$$

Sujeto a:

$$\sum_{e \in \delta(i)} x_e = 2, \qquad i \in V \ \{0\} \qquad (1)$$

$$\sum_{e \in \delta(0)} x_e = 2k \qquad (2)$$

$$\sum_{e \in \delta(S)} x_e \geq 2\, r(S), \qquad S \subseteq V \setminus \{0\}, \qquad S \neq 0 \qquad (3)$$

$$x_e \in \{0,1\}, \qquad e \notin \delta(0) \qquad (4)$$

$$x_e \in \{0,1,2\}, \qquad e \in \delta(0) \qquad (5)$$

La restricción (1) asegura que cada cliente sea visitado exactamente una sola vez. La restricción (2) indica que hay K rutas formadas. La fórmula (3) impone la restricción de capacidad. Por último, las condiciones (4) y (5) imponen que cada arco entre dos clientes sea atravesado como mucho una vez y cada arco incidente en el almacén sea atravesado a lo sumo dos veces. En este último caso, el vehículo lleva a cabo una ruta que visita a un solo cliente.

Otra formulación consiste en denotar la colección de rutas factibles como $R = \{R_1, \ldots, R_s\}$ donde s = | R |. Cada ruta R_j tiene asociado un coste γ_j y a_{ij} es un coeficiente binario que es igual a 1 si y solo si el vértice v_i es visitado por la ruta R_j. Una variable binaria x_j con j = 1, ..., s es igual a 1 si y solo si la ruta R_j forma parte de la solución.

El modelo propuesto por (Balinski & Quandt, 1964) queda como sigue:

$$\text{Minimizar} \quad \sum_{j=1}^{s} \gamma_j \, x_j$$

Sujeto a:

$$\sum_{j=1}^{s} a_{ij} x_j = 1, \qquad i \in V \ \{0\} \qquad (6)$$

$$\sum_{j=1}^{s} x_j = k \qquad (7)$$

$$x_j \in \{0,1\}, \quad j = 1 \dots s \qquad (8)$$

De este modo la restricción (6) impone que cada cliente es visitado exactamente por un vehículo y una sola vez. La restricción (7) asegura que K rutas forman parte de la solución. Debido a que las rutas consideradas dentro del conjunto *R* son factibles, este modelo se puede modificar fácilmente para tener en cuenta otras restricciones.

2.2.2. Problema VRP con flota heterogénea

Una variante muy estudiada del problema clásico es el problema VRP con flota heterogénea o flota mixta, que se caracteriza por disponer de vehículos con distinta capacidad máxima.

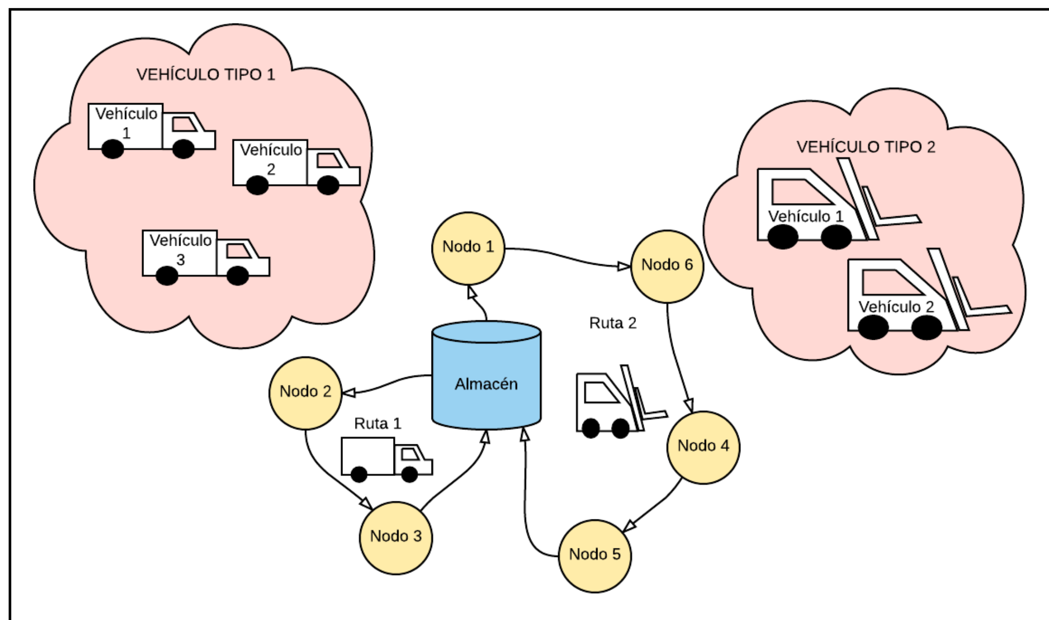


Figura 2.1. Ejemplo gráfico de un problema VRP con flota heterogénea

En la Figura 2.1. aparece un ejemplo concreto del problema VRP con flota heterogénea. En este ejemplo, la flota de vehículos está compuesta por dos

tipos de vehículos (camiones y carretillas), cada tipo de vehículo con una capacidad distinta. La solución del supuesto problema consta de dos rutas:

- Ruta 1: determinada por el par (R, m) donde $= \{0,2,3,0\}$, es decir, comienza en el almacén, suministra a los consumidores de los nodos 2 y 3 (en ese orden) y termina en el almacén. Además, m representa el vehículo que cubre la ruta, que en este caso, es un camión.

- Ruta 2: determinada por el par (R, m) donde $R = \{0,1,6,4,5,0\}$, es decir, comienza en el almacén, suministra a los consumidores de los nodos 1, 6, 4 y 5 (en ese orden) y termina en el almacén. Además, m representa el vehículo que cubre la ruta, que en este caso, es una carretilla.

2.2.3. Otras variantes del problema VRP clásico

Existen otras variantes del problema VRP en función de la flota de vehículos disponible, así como de los costes que intervienen en el problema. Las características que más comúnmente se modifican son el número de vehículos disponibles de cada m tipo, ya que puede ser ilimitado. Además se pueden considerar unos costes fijos F_j, que representan el coste del alquiler del vehículo o el coste de amortización del vehículo. Otro factor que se puede modificar es la dependencia o independencia de los costes asociados a cada arco en función del tipo de vehículo.

Otra variante de este problema (Toth & Vigo, 2003) , consiste en asociar a cada consumidor un tiempo de servicio s_i (no negativo) que debe ser respetado. Así mismo, es posible imponer otras restricciones relacionadas con el tiempo, como puede ser que la duración total de cada una de las rutas no supere un límite superior T (o que no lo haga la longitud total L en caso de que se formule en términos de distancia). A esta variante del problema de enrutamiento se conoce como problema VRP con restricción en la distancia, ya mencionada anteriormente.

Las variantes más estudiadas del problema VRP, así como sus principales características se recogen en la Figura 2.2. Se consideran dos tipos distintos de problema, HVRP y FSM, este último con una flota de vehículos ilimitada. En dicha tabla, los acrónimos incorporan las letras H para indicar que la flota es heterogénea, F cuando se tienen en cuenta los costes fijos y D si el coste asociado a cada arco depende del vehículo que recorra dicho arco. Un caso especial es el problema SDVRP, conocido como VRP dependiente del lugar o VRP Site-Dependent (Zare-Reisabadi & Hamid Mirmohammadi, 2015), el cual se caracteriza porque cada consumidor puede incluir restricciones respecto al

tipo de vehículo que le suministra. Este tipo de problema es aplicable cuando, por ejemplo, una zona urbana está restringida a un tipo determinado de vehículo por cuestión del tamaño del trazado urbano.

Variante del problema	Tamaño de la flota	Costes fijos	Costes de ruta
HVRPFD	Limitado	Considerados	Dependiente
HVRPD	Limitado	No considerados	Dependiente
FSMFD	Ilimitado	Considerados	Dependiente
FSMD	Ilimitado	No considerados	Dependiente
FSMF	Ilimitado	Considerados	Independiente
SDVRP	Limitado	No considerados	SDVRP

Figura 2.2. Variantes del problema VRP y sus características

2.3. Problema VRP con ventanas de tiempo

El problema VRP con ventanas de tiempo o VRPTW por sus siglas en inglés (VRP with Time Windows) (Cordeau J. F., 2007) es una importante generalización del problema VRP clásico en el cual el momento en el que se sirve a un cliente i debe estar en un intervalo de tiempo $[a_i, b_i]$. Un vehículo puede llegar a servir a un cliente antes de a_i y esperar hasta que el cliente se encuentre disponible, pero el vehículo no puede llegar al cliente después de b_i. Se pueden citar como ejemplos dentro de esta variante, el reparto de comida y bebida, el reparto de prensa o la recogida de residuos industriales.

El problema VRPTW es de tipo NP-Duro ya que es una generalización del problema CVRP el cual se obtiene con $a_i = 0$ y $b_i = \infty$ para cada cliente i. Incluso cuando el tamaño de la flota de vehículos es fijo, encontrar una solución factible para un problema VRPTW es un problema del tipo NP – completo. Por esta razón, la investigación del problema VRPTW se ha centrado en las técnicas heurísticas. Sin embargo cuando el problema está suficientemente restringido, como por ejemplo cuando el intervalo $[a_i, b_i]$ es suficientemente estrecho, es posible resolver el problema de forma óptima a través de técnicas de programación matemática.

El problema VRPTW se define en un grafo dirigido, $G = (V, A)$ donde $|V| = n + 2$ y el depósito queda representado por los vértices v_0 y v_{n+1}. Las rutas factibles se corresponden con aquellas rutas que empiezan en el vértice v_0 y terminan en el vértice v_{n+1}. El conjunto de los vehículos disponibles se denota por K, con $|K| = k$. Además se considera un tiempo de servicio s_i para cada vértice v_i (con

$s_0 = s_{n+1} = 0$) y t_{ij} el tiempo que se tarda en llegar del cliente del vértice v_i al cliente situado en v_j. A cada vértice $v_i \in N = V \setminus \{0, n+1\}$ se le asocia una ventana de tiempo $[a_i, b_i]$, y de la misma manera se pueden considerar las siguientes ventanas de tiempo $[a_0, b_0]$ y $[a_{n+1}, b_{n+1}]$ asociadas al vértice del almacén. Si no se aplican restricciones particulares a la disponibilidad de los vehículos, entonces $a_0 = min_{i \in N}\{a_i - t_{0i}\}$, $b_0 = max_{i \in N}\{b_i - t_{0i}\}$, $a_{n+1} = min_{i \in N}\{a_i + s_i + t_{i,n+1}\}$ y $b_{n+1} = max_{i \in N}\{b_i + s_i + t_{i,n+1}\}$. Al igual que en el problema CVRP, d_i es la demanda de cada cliente y C la capacidad máxima de los vehículos.

En la formulación matemática del VRPTW intervienen dos tipos de variables. Se emplean variables binarias $x_{ij}^k, (i,j) \in A$ con $k \in K$, que son igual a 1 si el arco (i, j) lo recorre un vehículo k. Por otro lado, se utilizan variables continuas w_i^k, con $i \in N$ y $k \in K$, para indicar el instante de tiempo en el que cada vehículo k empieza a servir al cliente del vértice v_i. Por último se toma $\delta^+(i) = \{j: (i,j) \in A\}$ y $\delta^-(j) = \{i: (i,j) \in A\}$. Según se propone en (Desrochers & Soumis, 1988) la formulación matemática quedaría como sigue:

$$\text{Minimizar} \sum_{k \in K} \sum_{(i,j) \in A} c_{ij} x_{ij}^k$$

Sujeto a:

$$\sum_{k \in K} \sum_{j \in \delta^+(i)} x_{ij}^k = 1, \qquad i \in N \tag{9}$$

$$\sum_{j \in \delta^+(0)} x_{0j}^k = 1, \qquad k \in K \tag{10}$$

$$\sum_{i \in \delta^-(j)} x_{ij}^k - \sum_{i \in \delta^+(j)} x_{ij}^k = 0, \qquad k \in K, \qquad j \in N \tag{11}$$

$$\sum_{i \in \delta^-(n+1)} x_{i,n+1}^k = 1, \qquad k \in K \tag{12}$$

$$x_{ij}^k \left(w_i^k + s_i + t_{ij} - w_j^k \right) \leq 0, \qquad k \in K, \qquad (i,j) \in A \tag{13}$$

$$a_i \leq w_i^k \leq b_i, \qquad k \in K, \qquad i \in V \tag{14}$$

$$\sum_{i \in N} d_i \sum_{j \in \delta^+(i)} x_{ij}^k \leq C, \qquad k \in K \tag{15}$$

$$x_{ij}^k \in \{0,1\}, \qquad k \in K, \qquad (i,j) \in A \tag{16}$$

La restricción (9) impone que cada cliente se visite una sola vez, mientras que las restricciones (10), (11) y (12) aseguran que cada vehículo se utilice exactamente una vez. La restricción (13) asegura la consistencia de las

variables de tiempo w_i^k y las ventanas de tiempo se imponen en la restricción (14). Por último, la restricción (15) limita la cantidad transportada de cada vehículo a la capacidad máxima C.

2.4. Problema de rutas de vehículos con inventario

El problema de rutas de vehículos con inventario o IRP por sus siglas en inglés (Inventory Routing Problem) es una importante variante del problema VRP, que integra decisiones de rutas y de control del inventario (Cordeau J. F., 2007). El problema aparece en entornos en los que se siguen políticas en las que el proveedor tiene el poder de planificar el volumen y frecuencia de los repartos. Por el contrario, el proveedor acepta asegurar a su cliente que no se quedará sin stock. En la relación tradicional proveedor-cliente, en la que el cliente hace los pedidos según sea necesario, pueden existir ineficiencias que resultan en altos niveles de inventario así como altos costes de distribución.

A pesar de las ventajas que presenta que el proveedor planifique los repartos, esta tarea no es fácil y en concreto resulta más difícil cuando se tiene un gran número de clientes. Por tanto, el problema de vehículos con inventario busca determinar una estrategia de distribución que minimice, a largo plazo, los costes de distribución.

2.4.1. Problema IRP determinístico

El problema IRP determinístico cobra sentido cuando se lleva a cabo una distribución de un solo producto y desde una sola instalación, a un conjunto de n clientes en un horizonte temporal T (puede ser infinito). Cada cliente i consume el producto con una tasa u_i (cantidad consumida al día) y puede mantener un inventario del producto hasta una cantidad máxima Q_i. El inventario del cliente i es I_i^0 en el instante de tiempo 0. Se dispone de una flota de K vehículos iguales con capacidad C. Si se reparte una cantidad q_i al cliente i el proveedor gana una cantidad $r_i q_i$. Además, se considera que un vehículo tarda t_{ij} en recorrer un arco (i, j) y recorrer dicho arco tiene un coste asociado c_{ij}. El objetivo es maximizar el beneficio en el horizonte de planificación sin causar ruptura de stock a los clientes.

Para simplificar la expresión del coste, tomemos u como la tasa de consumo, Q la cantidad que puede almacenar el cliente, I^0 el inventario inicial, c el coste de suministrar a un cliente y C la capacidad de los vehículos. Considerando un periodo de planificación T el coste para dicho periodo es v_T y se calcula como sigue:

$$v_T = \max\left\{0, \left[\frac{Tu - I^0}{\min\{Q, C\}}\right]\right\} c$$

2.4.2. Problema IRP estocástico

En el problema IRP estocástico las demandas de los clientes se definen en instantes discretos de tiempo t por medio de variables aleatorias. Para ello se define un vector $U_t = (U_{1t}, \dots, U_{nt})$ que contiene las demandas para el tiempo t. En esta variante estocástica del IRP, debido a que la demanda es incierta, existen mayores posibilidades de que el cliente sufra una ruptura de stocks, y esta situación en muchas ocasiones es imposible de evitar. En consecuencia, se aplica una penalización $p_i z_i$ si la demanda insatisfecha en el día t del cliente i es z_i. Al igual que en el IRP determinístico, el objetivo es determinar una política de distribución que maximice los beneficios en el periodo temporal considerado.

Otras variantes del problema IRP estocástico consideran que cada cliente i estará presente con una determinada probabilidad. Además es posible considerar aleatorios los tiempos de servicio s_i de un cliente i y el tiempo t_{ij} en recorrer un arco (i, j).

Capítulo 3

Métodos utilizados para resolver VRP

En este capítulo se hace una revisión de los distintos métodos disponibles para la resolución de las diferentes variantes del problema de enrutamiento de vehículos. Se comienza por el problema VRP clásico y se describen, en primer lugar, los métodos exactos, a continuación, diferentes técnicas heurísticas y, por último, las técnicas metaheurísticas. Para los problemas VRP con ventanas de tiempo y el problema de enrutamiento de vehículos con inventario se describen brevemente algunos métodos disponibles.

3.1. Métodos de resolución para el VRP clásico

En este apartado se describen algunos métodos exactos y heurísticos disponibles para la resolución del VRP clásico, aunque la mayoría de estas técnicas se han adaptado a otras de las variantes descritas.

En general, los algoritmos exactos se han desarrollado teniendo en cuenta la restricción de capacidad y, haciendo algunas modificaciones, se pueden aplicar a problemas con restricción de longitud en las rutas. En cambio, la mayoría de las técnicas heurísticas consideran ambos tipos de restricción.

3.1.1. Algoritmos exactos para el VRP

Los algoritmos exactos más utilizados para la resolución del problema VRP son los algoritmos de ramificación y poda (branch and bound). Este tipo de métodos consisten en un proceso de enumeración implícita de las soluciones mediante la división del espacio de búsqueda en subconjuntos de soluciones (cada uno de los cuales da lugar a un subproblema) a modo de árbol con la raíz en el problema inicial. Mediante el uso de cotas de los problemas, el algoritmo explora las ramas del árbol que representan a dichos subconjuntos del conjunto de soluciones, descartando aquellas ramas que no puedan producir mejores soluciones que la mejor solución encontrada hasta el momento. Las

ramas que superan la poda se vuelven a dividir en subconjuntos y se repite el procedimiento de nuevo.

Los algoritmos de ramificación y poda se desarrollan adaptados a una formulación concreta del problema VRP, por lo que existen múltiples variantes del algoritmo de ramificación y cota. En (Cordeau J. F., 2007) se hace una revisión de las propuestas más relevantes.

Otros métodos exactos para la resolución del problema VRP son los algoritmos de ramificación y corte que actualmente son la mejor opción entre los métodos exactos disponibles para el problema VRP. La descripción de algunos métodos de ramificación y corte se puede encontrar en (Naddef & Rinaldi, 2002) y en (Letchford, Lysgaard, & Eglese, 2007).

3.1.2. Heurísticas clásicas para el VRP

Hasta ahora se han propuesto un gran número de heurísticas para la resolución del VRP. Inicialmente se aplicaron sobre todo algoritmos constructivos, mientras que recientemente se han desarrollado técnicas metaheurísticas mucho más potentes. A continuación, se presenta una revisión de ambas familias de algoritmos. La mayoría de ellos se han desarrollado para el problema VRP simétrico. Además, debido a que encontrar una solución factible con exactamente K vehículos es un problema NP-completo, casi todos los métodos asumen que se dispone de un número ilimitado de vehículos. Muchos de los métodos propuestos se pueden adaptar fácilmente para, en la práctica, tener en cuenta otras restricciones. Dentro de este apartado se clasifican las heurísticas en métodos constructivos, métodos de dos fases y métodos de mejora de rutas.

a. Heurísticas constructivas de rutas

Estas heurísticas fueron las primeras que se aplicaron al problema CVRP y aún forman parte de implementaciones de software para distintas aplicaciones de rutas. Estos algoritmos generalmente empiezan con una solución vacía y se van añadiendo uno o más clientes en cada iteración, hasta que todos los clientes formen parte de una ruta. Este tipo de algoritmos se subdividen en secuenciales y paralelos en función del número de rutas elegibles en la inserción de un cliente. Los métodos secuenciales solo tienen en cuenta una ruta cada vez, mientras que los métodos paralelos consideran más de una ruta simultáneamente.

Los algoritmos constructivos quedan totalmente especificados por tres aspectos:

a. Criterio de inicialización.
b. Selección del criterio que especifica qué clientes se eligen para insertar en la iteración actual.
c. Criterio de inserción para decidir dónde situar los clientes elegidos dentro de las rutas actuales.

La primera y una de las más famosas heurísticas dentro de este grupo es la que fue propuesta por (Clarke & Wright, 1964), que se basa en el concepto de los ahorros. Este algoritmo es, en principio, paralelo ya que más de una ruta está activa cada vez, aunque también puede ser implementado en forma secuencial. El algoritmo resultante es bastante rápido pero puede no dar buenos resultados. Las propuestas de (Golden, Magnanti, & Nguyen, 1977), (Paessens, 1988) y (Nelson, 1985) implementan algunos cambios en el método de Clarke & Wright que llevan a mejores soluciones y son más eficientes en términos de computación. Dicho método de Clarke & Wright se describe en detalle en un subapartado de esta sección, ya que en el Capítulo 6 se ilustra el funcionamiento de este método para generar una solución inicial para el método de la búsqueda tabú granular aplicada al problema VRP.

Otra heurística clásica constructiva es el algoritmo de inserción secuencial (Mole & Jameson, 1976). El algoritmo utiliza como criterio de selección y de inserción la evaluación de la distancia extra que resulta de insertar un cliente x que no pertenece a ninguna ruta entre dos clientes consecutivos i y j de la ruta actual. Esta diferencia se calcula como $\alpha(i, x , j) = c_{jx} + c_{xj} - \lambda c_{ij}$ donde λ es un parámetro a especificar. Algunas variantes consideran como distancia extra la distancia existente entre un cliente y el almacén. Después de cada inserción, la ruta actual se puede mejorar mediante un procedimiento 3-opt. Los métodos k-opt (Leizhen Cai, 2013) consisten en la optimización de una ruta eliminando k arcos y reconectando de nuevo los vértices de todas la maneras posibles para a continuación, evaluar dichas alternativas y elegir la mejor entre todas ellas. En concreto, el procedimiento 3-opt elimina tres arcos.

La heurística de inserción propuesta por (Mingozzi, Christofides, & Toth, 1979) es un método en dos pasos que representa un buen equilibrio entre efectividad y eficiencia. En el primer paso se utiliza el algoritmo de inserción secuencial para determinar un conjunto de rutas factibles. El segundo paso consiste en una inserción en paralelo, de forma que para cada ruta determinada en el primer paso, se selecciona un cliente representativo y se construye para cada uno de ellos una ruta que solo visita a dicho cliente. El resto de clientes que

no pertenecen a ninguna ruta se insertan utilizando un criterio de arrepentimiento, en el que se tiene en cuenta la diferencia en el coste entre la mejor inserción y la segunda mejor inserción. Posteriormente las rutas se mejoran mediante el procedimiento 3-opt.

❖ Método de Clarke & Wright

Formulación del método

Se dispone de un número ilimitado de vehículos, cada uno con una determinada capacidad, situados en un punto P_0 (vértice v_0), que es el almacén. Se quiere satisfacer la demanda d_i con $i = 1, \dots, n$ de n clientes, situados cada uno de ellos en un punto P_i (vértice v_i) con $i = 1, \dots, n$. Se dan las distancias l entre los distintos puntos y se busca minimizar la distancia total recorrida por los vehículos.

Cada cliente i situado en un punto P_i, estará unido a otros dos vértices, de forma que uno de ellos o ambos podría ser P_0. Es decir, un cliente puede estar unido exclusivamente al almacén, formando una ruta en la que dicho cliente es el único visitado por ese vehículo. Otra posibilidad consiste en que el cliente sea el primero o el último al que visita un vehículo dentro de una ruta en la que visita a más clientes. Por último, si el cliente no está unido a P_0, significa que es un cliente intermedio en la ruta que sigue un vehículo determinado.

Se consideran dos puntos P_y y P_z, unidos respectivamente a un punto anterior y siguiente denotados como P_{y-1} y P_{y+1} en el caso de P_y y P_{z-1} y P_{z+1} para el caso de P_z. Esta situación inicial se muestra en la Figura 3.1.

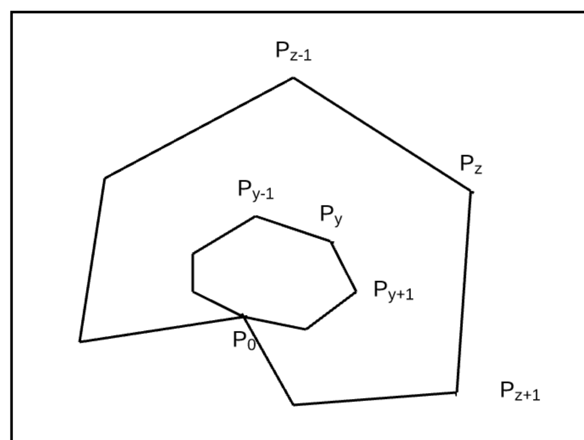


Figura 3.1. Situación inicial de P_y y P_z

En primer lugar se calcula el efecto de unir los puntos P_y y P_z, evaluando las cuatro posibles combinaciones mostradas en la Figura 3.2. Estas posibilidades se consiguen variando la existencia o no de las uniones entre $P_{y-1} - P_y$; $P_y - P_{y+1}$, a la vez que permanecen o no las uniones entre $P_{z-1} - P_z$; $P_z - P_{z+1}$.

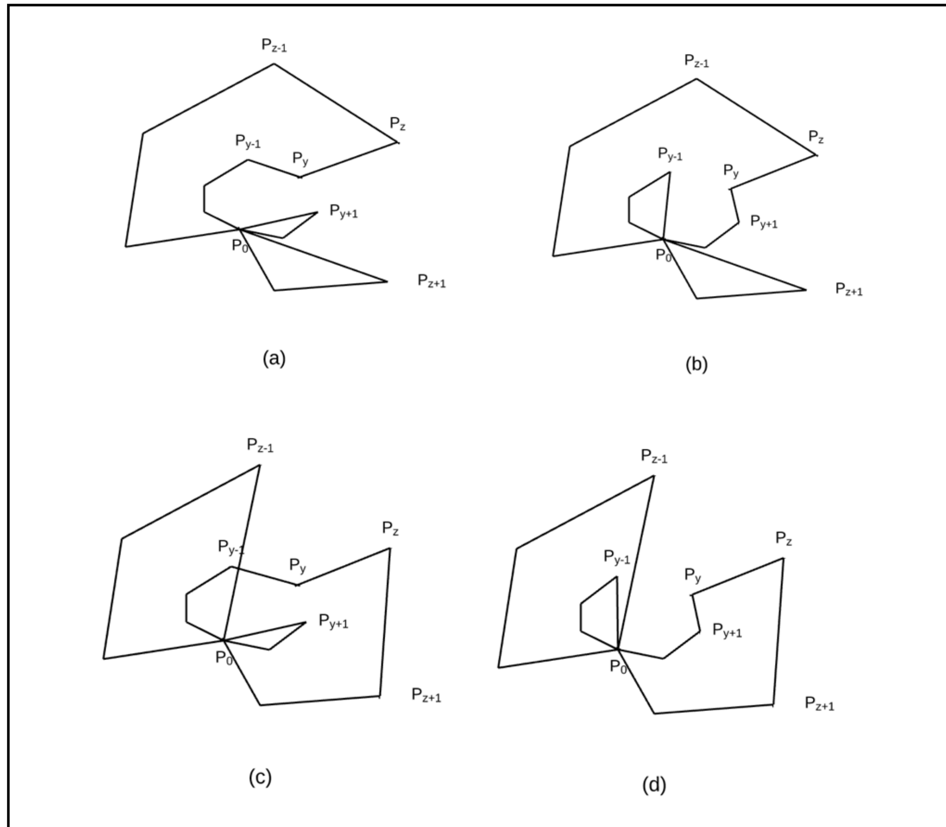


Figura 3.2. Posibles combinaciones de unir P_y y P_z

Estas combinaciones son posibles siempre y cuando se cumpla que P_y y P_z se encuentran en distintas rutas. Si los puntos P_y y P_z considerados se encuentran en la misma ruta, como se muestra en la Figura 3.3. una de las cuatro opciones no es posible.

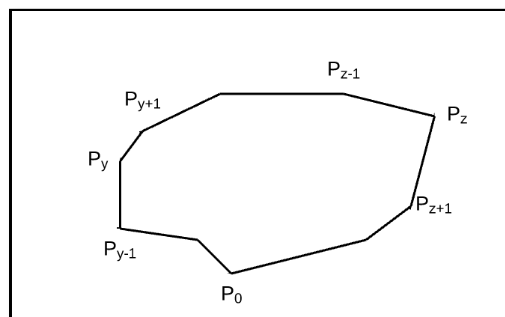


Figura 3.3. P_y y P_z en la misma ruta

En concreto, como se puede observar en la Figura 3.4., es la opción (b) la que no es posible, ya que la ruta que contiene a los clientes P_y y P_z, señalada en color rojo, no está conectada con el almacén.

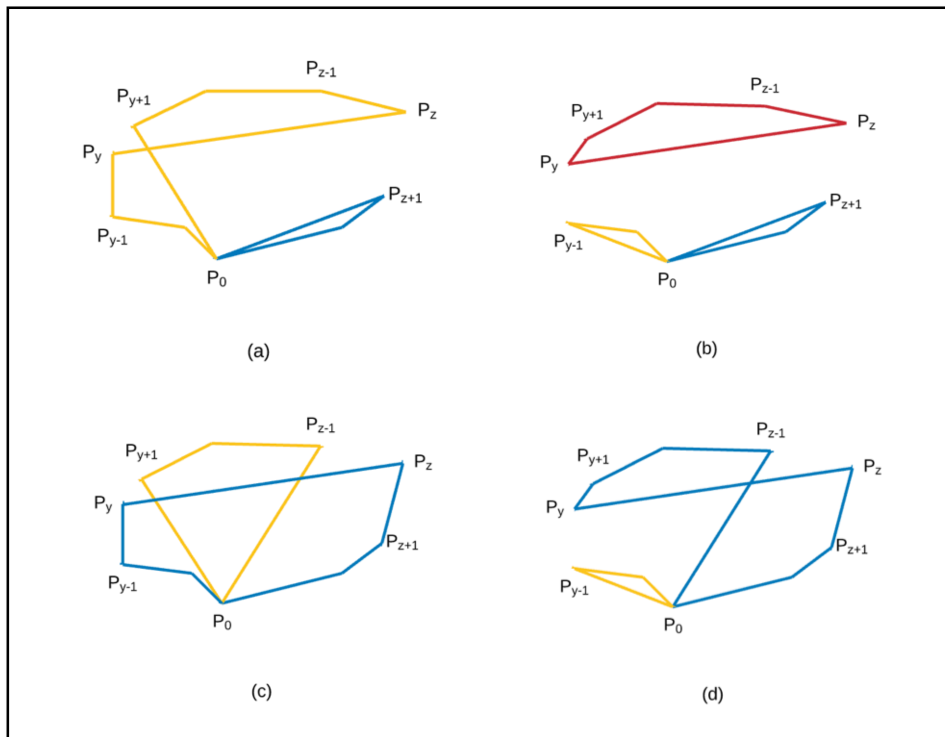


Figura 3.4. Posibles combinaciones de unir P_y y P_z si están en la misma ruta

Para cada una de las posibilidades señaladas, se calcula el "ahorro" que se consigue con respecto a la distancia recorrida, el cual viene dado por las siguientes fórmulas:

Opción (a): $l_{y,y+1} - l_{0,y+1} + l_{z,z+1} - l_{0,z+1} - l_{y,z}$

Opción (b): $l_{y-1,y} - l_{0,y-1} + l_{z,z+1} - l_{0,z+1} - l_{y,z}$

Opción (c): $l_{y,y+1} - l_{0,y+1} + l_{z,z-1} - l_{0,z-1} - l_{y,z}$

Opción (d): $l_{y-1,y} - l_{0,y-1} + l_{z,z-1} - l_{0,z-1} - l_{y,z}$

Estos ahorros se calculan para cada par de clientes y se selecciona la opción factible, en cuanto al número de vehículos y su capacidad, que proporciona el máximo ahorro. Los dos clientes que cumplan dichas condiciones se unen y a continuación se recalculan los ahorros.

Este método es equivalente a asignar dos “precios sombra” a cada cliente. Por ejemplo, para el cliente P_y, los precios sombra serán $l_{y-1,y} - l_{0,y-1}$ y $l_{y,y+1} - l_{0,y+1}$. Cuando un enlace se corta, el precio sombra se reduce en la cantidad que contiene la celda que une a los clientes entre los cuales se produce la desunión. Debido a que este valor de la celda es máximo, cuando un cliente está unido a otros dos clientes (que no sean el almacén) todos los valores de celda asociados a ese cliente se convierten en negativos, por lo que dicho punto no volverá a considerarse para formar parte de un enlace.

Como resultado de esto, las únicas uniones que serán eliminadas son aquellas que unen los puntos de los clientes al punto almacén (P_0), y por tanto, los ahorros se calcularán según la siguiente expresión:

$$l_{0,y} + l_{o,z} - l_{y,z}$$

Procedimiento

En primer lugar, dados los puntos en los que se sitúan los *n* clientes, se calculan las distancias para cada par de clientes. Una vez calculadas las distancias, se procede a calcular los ahorros conseguidos para cada par de clientes según la expresión anterior.

El procedimiento comienza asignando a cada vehículo la visita a un solo cliente. En caso de que la demanda d_i de un cliente sea mayor que la capacidad C del vehículo, la carga se dividirá entre tantos vehículos como sean necesarios y se realizará el procedimiento con la carga restante. Por ejemplo, si la capacidad del vehículo es 1200 unidades y un cliente demanda 1800 unidades, las primeras 1200 unidades se enviarán en un vehículo que solo visitará a dicho cliente, y el resto, es decir, las otras 600 unidades son las que se consideran como dato de entrada para resolver el problema.

Para aplicar este método, se suele utilizar una media matriz como la que aparece en la Figura 3.5. En los “peldaños superiores” se indica el cliente situado en el punto P_i con $i = 1, \dots, n$. En la columna izquierda se anotan las demandas de los respectivos clientes una vez que la carga ha sido divida como se detalló anteriormente. En cada una de las celdas (y, z), se indica en la parte inferior derecha la distancia que une a los clientes P_y y P_z, y en la parte inferior izquierda, el máximo ahorro calculado para ese par de clientes.

D	P_0				
d_1	$t_{0,1}$ s $l_{0,1}$	P_1			
d_2	$t_{0,2}$ s $l_{0,2}$	$t_{1,2}$ s $l_{1,2}$	P_2		
d_i	$t_{0,i}$ s $l_{0,i}$	$t_{1,i}$ s $l_{1,i}$	$t_{2,i}$ s $l_{2,i}$	P_i	
d_n	$t_{0,n}$ s $l_{0,n}$	$t_{1,n}$ s $l_{1,n}$	$t_{2,n}$ s $l_{2,n}$	$t_{i,n}$ s $l_{i,n}$	P_n

Figura 3.5. Estructura matriz método Clarke & Wright

Además, en el centro de las celdas se colocará una cifra $t_{y,z}$, que puede ser 0, 1 ó 2 (en caso de ser 0 se omite para simplificar). Cuando $t_{y,z}$ = 1, los clientes P_y y P_z están unidos; en caso contrario $t_{y,z}$ = 0. Si un cliente es servido exclusivamente por un vehículo, $t_{y,0}$ = 2. En cualquier caso, se debe cumplir que la suma de los números que aparecen en la fila y la columna asociadas a un cliente debe ser igual a 2, tal y como aparece en la siguiente expresión.

$$\sum_{z=0}^{y-1} t_{y,z} + \sum_{z=y+1}^{z=n} t_{y,z} = 2$$

Por tanto, se debe encontrar el máximo ahorro por filas y columnas, sujeto a las siguientes condiciones:

I. $t_{y,0}$ y $t_{z,0}$ deben ser mayores que 0.
II. P_y y P_z no deben estar en la misma ruta.
III. Para que se puedan unir dos clientes, debe ser posible satisfacer la demanda con un mismo vehículo, es decir, la suma de la demanda d_y y d_z de los clientes situados en los puntos P_y y P_z, y de los clientes ya incluidos en esa ruta, no debe exceder la capacidad del vehículo.

Si estas condiciones se cumplen, $t_{y,z}$ toma valor 1 y los demás valores de $t_{i,j}$ son modificados según la expresión anterior. Así mismo, la columna izquierda que contiene las demandas se actualiza, de forma que la demanda de un

cliente será 0 si $t_{j,0}$= 0 y se cambiará el valor de la demanda d_j por la suma de las demandas de los clientes que forman parte de esa ruta.

b. Heurísticas en dos fases

Estas heurísticas se basan en la descomposición del proceso de resolución del VRP en dos fases. La primera fase conocida como agrupamiento o "*clustering*" consiste en determinar una división de los clientes y agruparlos en subconjuntos, de forma que cada subconjunto corresponderá a una ruta. La segunda fase de enrutamiento o "*routing*" determina la secuencia de los clientes en cada ruta. Se han propuesto diferentes técnicas para llevar a cabo la fase de agrupamiento, mientras que la fase de enrutamiento equivale a resolver un problema del viajante.

Un ejemplo de este tipo de heurística es el algoritmo de barrido o algoritmo "sweep" propuesto por (Wren, 1971), (Wren & Holliday, 1972) y (Gillet & Miller, 1974) que se aplica a problemas VRP planos. El algoritmo comienza con un cliente elegido al azar y luego asigna los restantes clientes al vehículo actual considerándolos en orden creciente del ángulo polar que forma cada uno de ellos con respecto al almacén y el cliente inicial. Cuando no es factible añadir un cliente al vehículo actual por exceso de carga, se crea una nueva ruta con dicho cliente. Una vez que todos los clientes han sido asignados a un vehículo, se determina el orden de los clientes dentro de cada ruta resolviendo un problema del viajante para cada una de las rutas.

Otra heurística en dos fases es el método de ramificación y cota truncado propuesto en (Christofides & al., The vehicule routing problem, 1979). En este método el árbol de decisión contiene tantos niveles como número de vehículos disponibles, y en cada nivel del árbol de decisión un nodo dado corresponde a una solución parcial compuesta por un conjunto de rutas completas. Los nodos inferiores corresponden a todas las posibles rutas incluyendo un subconjunto de clientes que no están asociados a ninguna ruta.

El algoritmo de (Fisher & Jaikumar, 1981) lleva a cabo el paso de agrupamiento resolviendo un problema de asignación. Cada vehículo se asigna a un cliente representativo, y el coste de un cliente asignado a un vehículo es igual a su distancia a dicho cliente representativo. Por tanto, se resuelve el problema de asignación y las rutas definitivas se determinan resolviendo un problema del viajante para cada grupo de clientes.

Los métodos en dos fases permiten tener un control directo sobre el número de rutas que forman parte de la solución final, mientras que el algoritmo de barrido no permite este control. El rendimiento de estos algoritmos es

generalmente comparable a los algoritmos constructivos en términos de efectividad.

Una familia diferente de los métodos de dos fases son los algoritmos de pétalos. Estos algoritmos generan un gran conjunto de rutas factibles, denominadas pétalos, y selecciona el subconjunto final resolviendo un problema de partición (set partitioning problem). (Foster & Ryan, 1976) han propuesto reglas heurísticas para determinar el conjunto de rutas seleccionadas, mientras que (Renaud, Boctor, & Laporte, 1996) han descrito una extensión que considera configuraciones 2-pétalos, que consisten en dos rutas cruzadas o incrustadas. El rendimiento general de estos algoritmos es normalmente superior al algoritmo de barrido.

Finalmente, dentro de los métodos de dos fases se propone una alternativa en la que se crea en la primera fase una ruta con todos los clientes para depués subdividirles en rutas factibles. El rendimiento de esta propuesta es generalmente peor que el de las anteriores.

c. Heurísticas de mejora de rutas

Los algoritmos de búsqueda local a menudo se usan para mejorar una solución inicial generada mediante otras heurísticas. Comenzando desde una solución dada, el método de búsqueda local aplica modificaciones simples, como intercambios de arcos o movimientos de clientes, para obtener soluciones en el entorno con mejores costes. Si se encuentra una solución mejor dicha solución se convierte en la solución actual y el proceso se repite. En caso contrario, se identifica como un mínimo local.

Existe una gran variedad de entornos, que se pueden subdividir en entornos intra-ruta si operan en una sola ruta cada vez, o entornos inter-ruta si consideran más de una ruta simultáneamente. El entorno más común se genera mediante k -opt propuesto en (Lin, 1965), ya descrito anteriormente. El tiempo de computación requerido para examinar todos los entornos de una solución es proporcional a nk. Por eso, en la práctica solamente se utilizan valores de k igual a 2 o 3. Como alternativa se pueden utilizar entornos limitados con movimientos asociados a valores más altos de k, considerando solo un subconjunto de todos los intercambios.

3.1.3. Metaheurísticas

Se han aplicado varias metaheurísticas al problema VRP. Con respecto a las heurísticas clásicas, las metaheurísticas llevan a cabo una búsqueda más minuciosa en el espacio de soluciones y es menos probable que terminen en

un óptimo local. Las metaheurísticas pueden ser dividas en tres grandes grupos:

i. Búsqueda local: recocido simulado, búsqueda tabú.
ii. Búsqueda basada en poblaciones: algoritmos genéticos y procedimientos de memoria adaptativa.
iii. Mecanismos de aprendizaje: redes neuronales y colonia de hormigas.

Los algoritmos de búsqueda local exploran el espacio de soluciones realizando iterativamente movimientos desde una solución x_t en la iteración t hasta una solución x_{t+1} en el entorno $N(x_t)$ de x_t hasta que se satisfaga un criterio de parada. Si $f(x)$ denota el coste de la solución x, entonces $f(x_{t+1})$ no es necesariamente menor que $f(x_t)$. Debido a esto se implementan mecanismos para evitar caer en un proceso cíclico.

En el recocido simulado se elige al azar una solución x en el entorno $N(x_t)$. Si $f(x) \leq f(x_t)$, entonces $x_{t+1} = x$. En el otro caso se toma $x_{t+1} = x$ con una determinada probabilidad p y $x_{t+1} = x_t$ con probabilidad $1 - p_t$, donde p_t es una función decreciente de t y de $f(x) - f(x_t)$. Esta probabilidad a menudo es igual a:

$$p_t = \exp\left(-\frac{f(x)-f(xt)}{\theta_t}\right)$$

donde θ_t es la temperatura en la iteración, definido en una función no creciente de t. El recocido determinístico es similar y existen dos versiones principales. En la primera versión $x_{t+1} = x$ si $f(x) < f(x_t) + \theta_1$ donde θ_1 es un parámetro controlado por el usuario. En la otra versión, $x_{t+1} = x$ si $f(x_{t+1} < \theta_2 f(x^*)$ donde θ_2 es un parámetro controlado por el usuario y x^* es la mejor solución encontrada.

En la búsqueda tabú para evitar caer en un proceso cíclico, cualquier solución que posea un cierto atributo de x_{t+1} se declara tabú por un número de iteraciones. En la iteración t, la búsqueda se mueve hacia la mejor solución x no tabú en el entorno $N(x_t)$.

Estos algoritmos de búsqueda local raramente se implementan en su versión básica y su éxito depende de la implementación de ciertos mecanismos. La regla empleada para definir los entornos es crítica para la mayoría de las heurísticas de búsqueda local. En el recocido simulado se utilizan distintas reglas para definir θ_t (Osman, 1993), mientras que en la búsqueda tabú se

emplean varias estrategias para implementar la permanencia tabú, la diversificación y la intensificación en la búsqueda.

Los algoritmos basados en poblaciones operan sobre varias generaciones de soluciones. En los algoritmos genéticos es común repetir *k* veces las siguientes operaciones: se extraen dos padres de la población para crear dos descendientes usando un proceso de cruce y aplicando un mecanismo de mutación a cada descendiente. Después, se eliminan los 2*k* peores elementos de la población y se reemplazan con 2*k* descendientes. Existen distintas reglas de cruce para los problemas secuenciales propuestas en (Bean, 1994), (Potvin, 1996), (Drezner, 2003) y (Prins, 2004). En los procedimientos de memoria adaptativa un descendiente se crea extrayendo y recombinando elementos de varios padres. En la versión original propuesta por (Taillard Y. R., 1995) para el problema VRP, se crea una solución inicial formada por rutas extraídas de varios padres y posteriormente esta solución se va completando y se optimiza mediante la búsqueda tabú.

Las redes neuronales son modelos compuestos por unidades interconectadas mediante enlaces ponderados, y se inspira en la naturaleza de las conexiones de las neuronas en el cerebro. Estos algoritmos construyen gradualmente una solución a través de un mecanismo de retroalimentación que modifica los pesos de los enlaces. En el campo del VRP los modelos de redes neuronales son redes elásticas que se ajustan a los vértices para generar soluciones factibles.

Los algoritmos de colonias de hormigas también utilizan un mecanismo de aprendizaje y se derivan de una analogía con las hormigas que dejan feromonas en su camino cuando buscan comida. Con el tiempo se depositan más feromonas en los caminos más frecuentados. Cuando se construye una solución VRP, a un movimiento se le puede asignar una mayor probabilidad de ser seleccionado si ha llevado a soluciones mejores en las iteraciones previas.

a. Heurísticas de búsqueda local

A principios de los años 90 se propusieron varias heurísticas de recocido simulado para el problema CVRP. La implementación de (Osman, 1993) define los entornos mediante 2-intercambios y aplica diferentes reglas para los cambios de temperatura. En lugar de utilizar una función no creciente como hacen otros autores, Osman reduce θ_t continuamente si la solución mejora, y en caso de que no mejore, θ_t se reduce a la mitad o es reemplazada por la temperatura de la mejor solución encontrada. Este algoritmo produce buenas soluciones pero no es competitivo con la mejor implementación de búsqueda

tabú del mismo período. El recocido determinístico fue aplicado por primera vez al problema VRP por (Golden & Al., 1998) y posteriormente el método fue sofisticado por (Li & Al., 2005).

Son múltiples los algoritmos de búsqueda tabú que se han propuesto para el problema VRP. Estas variantes se exponen en el apartado 4.8. ya que está dedicado a las distintas implementaciones de la búsqueda tabú aplicadas al problema VRP.

b. Heurísticas basadas en poblaciones

El proceso de memoria adaptativa propuesto por (Taillard Y. R., 1995) resultó ser una gran contribución al campo de la metaheurística. Inicialmente fue desarrollado en el contexto del problema VRP, pero es de aplicación general. La memoria adaptativa es un conjunto de buenas soluciones que se actualiza reemplazando los peores elementos por otros mejores. Para generar una nueva solución, se seleccionan varias soluciones dentro del conjunto y se recombinan. En el problema VRP las rutas se extraen de estas soluciones y se usan como base de una nueva solución. El proceso de extracción se aplica si es posible identificar rutas que no se solapen con las rutas seleccionadas previamente. Cuando esto ya no es posible, el proceso de búsqueda se inicializa desde una solución parcial formada por las rutas seleccionadas y algunos clientes que no pertenecen a ninguna ruta. La solución construida de este modo reemplaza a la peor de las soluciones del conjunto si tiene un coste mejor.

Una variante propuesta por (Tarantilis & Kiranoudis, 2002) obtiene en una primera fase una solución inicial mediante el procedimiento constructivo propuesto por (Paessens, 1988), que consiste en aplicar el método de Clarke y Wright y después movimientos 2-opt, intercambios de vértices entre rutas y reinserciones de vértices. Con el fin de generar nuevas soluciones a partir del conjunto de soluciones de la memoria adaptativa, Tarantilis and Kiranoudis extraen segmentos de rutas.

(Prins, 2004) ha desarrollado un algoritmo que combina dos características de la búsqueda evolutiva, que son los cruces y las mutaciones. Los cruces consisten en crear soluciones descendientes a partir de los padres, mientras que las mutaciones se llevan a cabo aplicando algoritmos de búsqueda local a un descendiente. Esta combinación lleva a un método que a menudo se conoce como el “algoritmo memético” propuesto en (Moscato & Cotta, 2003). En este algoritmo para crear un descendiente a partir de dos padres, una cadena $(i, \dots, j)$ se selecciona de uno de los padres y los vértices del segundo padre se exploran desde la posición $j + 1$ saltándose aquellos de la cadena $(i, \dots, j)$. Un

segundo descendiente se genera de manera similar pero invirtiendo el papel que juegan cada uno de los padres. Los descendientes se mejoran aplicando una combinación de inserción de vértices y arcos, intercambio de vértices e intercambios de vértices y arcos.

Otro algoritmo dentro de este tipo es el propuesto por (Berger & Barkaoui, 2004), que trabaja con poblaciones de tamaño constante, en las que se reemplaza a los padres por los descendientes creados y también hay migraciones entre dos poblaciones. En este algoritmo los descendientes se obtienen combinando rutas de dos padres si esto puede hacerse sin solapar e insertando clientes que no tienen ruta asignada en función del criterio de proximidad. A continuación, se aplica una heurística VLNS (Shaw, 1998) combinando tres mecanismos de inserción, y luego se eliminan vértices y se reinsertan por medio del procedimiento propuesto por (Solomon, 1987).

El método de (Mester & Bräysy, 2005) se desarrolló en un principio para el problema VRP con ventanas de tiempo, y después se aplicó al problema VRP clásico. Este método combina la búsqueda local con una estrategia evolutiva propuesta por (Rechenberg, 1973). Esto da lugar a un proceso iterativo de dos etapas. La estrategia evolutiva utiliza una regla determinística para seleccionar a la solución padre y crear un único descendiente a partir de este único padre. Los descendientes reemplazan al padre si le mejoran y son mejorados a través de un proceso complejo de búsqueda que combina, entre otros, la búsqueda tabú granular, un mecanismo de diversificación continua, intercambios de vértices y movimientos 2-opt.

c. Mecanismos de aprendizaje

Se han propuesto varias heurísticas basadas en mecanismos de aprendizaje para el problema VRP, pero ninguna red neuronal ni el algoritmo de colonia de hormigas pueden competir con las mejores propuestas de métodos de resolución del problema VRP. Sin embargo, recientemente, (Reimann, Rubio, & Wein, 1999) propusieron una heurística llamada *D-ant* que consigue buenos resultados, la cual aplica iterativamente dos fases hasta que se alcanza un criterio de parada. En la primera fase, se construye una primera generación de buenas soluciones a través de la heurística de Clarke and Wright y luego se aplica un procedimiento 2-opt de mejora a cada solución. A continuación, se crean nuevas generaciones de soluciones beneficiándose del conocimiento ganado al producir las generaciones anteriores. De este modo, en lugar de utilizar los ahorros estándar $s_i = l_{i0} + l_{0j} - l_{ij}$, se utiliza un valor $x_{ij} = \tau_{ij}^{\alpha} s_{ij}^{\beta}$ donde α y β son parámetros controlados por el usuario y τ_{ij}^{α} contiene información de lo bueno que ha resultado realizar la unión de los vértices v_i y

v_j en las generaciones anteriores. En la segunda fase se identifica la mejor solución de la primera fase y se descompone en subproblemas que se optimizan siguiendo el procedimiento descrito para la primera fase.

3.2. Métodos de resolución para el VRPTW

Al igual que para el problema VRP, para el problema VRPTW también existen múltiples métodos de resolución tanto exactos como técnicas heurísticas y metaheurísticas. Entre las metaheurísticas cabe destacar la búsqueda tabú propuesta en (Semet & Taillard, 1993) para el VRPTW y su sofisticación propuesta por (Taillard E. , 1997). También se han desarrollado distintos algoritmos genéticos aplicados al problema VRPTW como el que se describe en (Homberger & Gehring, 1999) o en (Berger, Barkaoui, & Bräysy, 2003). Para más información acerca de métodos de resolución del problema VRPTW se puede consultar (Cordeau J. F., 2007).

3.3. Métodos de resolución para el problema IRP

Debido a que el problema IRP es más complejo que los anteriores, los métodos que se aplican para resolver este problema difieren de los ya citados para el problema VRP o el VRPTW.

Una primera variante de los métodos disponibles para la resolución del problema IRP consiste en utilizar modelos de programación entera discretizada en el tiempo para determinar el conjunto de clientes que se debe visitar a corto plazo, así como la cantidad que se reparte a cada uno de ellos. (Fisher & Al., 1982) y (Bell & Al., 1983) propusieron los primeros modelos dentro de esta variante.

Otra segunda variante se basa en el análisis del IRP con un solo cliente y existen varios métodos relacionados propuestos en (Dror & Al., 1985) o en (Dror & Ball, 1987).

Otra variante se basa en el análisis asintótico de la política de reparto. En la propuesta de (Anily & Al., 1990) los clientes se dividen en regiones con el fin de que la demanda de cada zona sea igual a la carga que transportan los vehículos. Otras propuestas se desarrollan en (Gallego & Simchi-Levi, 1990) o en (Bramel & Simchi-Levi, 1995). Para más información acerca de los métodos existentes se recomienda consultar (Cordeau J. F., 2007).

Capítulo 4

Búsqueda tabú

En el presente capítulo se describe la metaheurística de la búsqueda tabú. Para ello, en primer lugar, se exponen ciertos conceptos básicos sobre la búsqueda local, debido a que la búsqueda tabú es una variante de la búsqueda local. Posteriormente se describen distintas variantes de la búsqueda tabú aplicadas al problema VRP.

4.1. La búsqueda local

Siguiendo a (Youssef, 1999) la búsqueda local es una de las técnicas heurísticas más antiguas y uno de los métodos más sencillos de optimización. Aunque el algoritmo es simple funciona bien con una amplia variedad de problemas tipo NP-Duro. El algoritmo empieza con una solución factible y utiliza una subrutina de mejora para encontrar una solución mejor en el entorno. Si se encuentra una solución mejor en el entorno, la búsqueda continúa entre las soluciones del entorno de esta mejor solución. El algoritmo termina cuando se topa con un óptimo local.

Este tipo de heurísticas deben tener en cuenta distintos factores:

1. Solución inicial: es posible comenzar con una solución obtenida mediante un algoritmo constructivo o de una solución aleatoria. Otra posibilidad es ejecutar varias veces la búsqueda local partiendo de diferentes soluciones iniciales y seleccionar entre ellas la mejor para utilizarla como solución inicial.

2. Elección del entorno: es determinante elegir correctamente el entorno a explorar debido a que, si esta zona es grande, requiere mayor tiempo de búsqueda y ofrece una buena solución, mientras que si la zona vecina a explorar es pequeña, se ejecuta más rápido pero puede llevar a una convergencia rápida a un óptimo local. Asociado a este punto, cabe señalar que si se decide trabajar con entornos pequeños, es importante partir de una solución inicial buena. Por el contrario, si se elige un entorno grande,

no es significativa la calidad de la solución inicial en la calidad de la solución final.

3. La subrutina de mejora: puede seguir dos estrategias distintas. Una de ellas consiste en aceptar como válida la primera solución encontrada que mejore la solución inicial. La otra opción consiste en explorar todo el entorno y seleccionar la solución final como la mejor solución entre las soluciones exploradas.

4.2. Generalidades

La búsqueda tabú es un método reciente de optimización basado en la Inteligencia Artificial. Este método fue introducido por Fred Glover en 1986 como una heurística iterativa para la resolución de problemas de optimización combinatoria y ha demostrado ser una técnica versátil y efectiva aplicada a este tipo de problemas.

Como ya se había indicado, la búsqueda tabú es un método simple desarrollado como una variante de la búsqueda local. Este método basado en un algoritmo iterativo, parte de una solución inicial la cual se modifica para llegar a una solución que mejora la función de coste. En algunas ocasiones se parte de una solución inicial aleatoria, mientras que otras veces se genera una solución mediante alguna técnica heurística para alimentar al método de la búsqueda tabú.

Esta técnica puede ser utilizada como un algoritmo determinístico o como un algoritmo estocástico. Recordemos que los algoritmos determinísticos son aquellos que llegan a la solución final a través de decisiones determinísticas por lo que, en distintas ejecuciones con los mismos datos para un mismo problema siempre llevan a la misma solución. Por el contrario, los algoritmos estocásticos construyen la solución a base de decisiones aleatorias, que dan lugar a soluciones distintas si se corre varias veces. Por tanto, la búsqueda será determinística o estocástica según se determinen distintos aspectos, como la permanencia en la lista tabú, que puede ser fija o elegida al azar en cada iteración.

A diferencia de la búsqueda local, técnica que termina cuando no se encuentra una solución mejor en el entorno, la búsqueda tabú continúa la búsqueda incluso si la solución encontrada en una iteración es peor que la solución actual. Para evitar hacer una búsqueda cíclica, se almacena en una "lista tabú" la información acerca de las últimas soluciones visitadas, de forma que los movimientos a estas "zonas tabú" no están permitidos.

El “estado tabú” de una solución se mantiene durante un número de iteraciones o hasta que se satisface una cierta condición, conocida como criterio de aspiración. Los criterios de aspiración se explican en detalle más adelante.

La principal característica que diferencia a la búsqueda local de otros métodos es su capacidad de memoria. Mientras que métodos como los algoritmos genéticos o recocido simulado carecen de memoria, la búsqueda tabú tiene una memoria adaptativa que se basa en:

I. Componente de memoria a corto plazo: es la componente básica de la búsqueda tabú.
II. Componente de memoria a medio plazo: utilizado para la intensificación de zonas en la búsqueda, es decir, para volver a visitar zonas que contenían buenas soluciones.
III. Componente de memoria a largo plazo: empleada para la diversificación de la búsqueda, que lleva a zonas aún no exploradas.

Utilizando la memoria a corto plazo se elabora un historial H que se mantiene con el objeto de guiar el proceso de búsqueda. El entorno a explorar es remplazado por otro entorno, el cual es función del historial H. El historial determina qué soluciones pueden ser alcanzadas por movimientos desde la solución actual. La componente de memoria a corto plazo se implementa a través de una serie de condiciones tabú y los criterios de aspiración asociados.

La idea principal de la memoria a corto plazo es clasificar ciertas direcciones de búsqueda como tabú o prohibidas. Esta memoria contiene atributos de las soluciones exploradas recientemente en la “lista tabú”, cuyo tamaño es el número de iteraciones en las cuales un movimiento se mantiene como prohibido.

4.3. Algoritmo de la búsqueda tabú

El proceso comienza a partir de una solución factible R del conjunto de soluciones factibles. Cada solución R tiene un entorno $N(R)$ asociado que está determinado por la propia solución R. Se genera la muestra de soluciones V^* contenidas en el entorno $N(R)$ de la solución R, es decir, se genera un entorno reducido, ya que la búsqueda tabú no explora el entorno completo. De este conjunto de soluciones V^* se elige la mejor de ellas (R^*) y se toma como la nueva solución actual. El movimiento hasta R^* se lleva a cabo aunque R^* sea peor que la solución actual R, pudiendo de esta manera escapar de óptimos

locales. A pesar de esto, cabe alguna posibilidad de caer en un proceso cíclico que conduzca a un óptimo local.

La lista tabú tiene la función de evitar volver a soluciones visitadas previamente y contiene información acerca de los movimientos realizados. Esto es así, porque almacenar información de las soluciones visitadas supone un gasto enorme en términos de memoria y tiempo de computación.

Las restricciones que impone la lista tabú también pueden evitar que la búsqueda se mueva hacia soluciones atractivas que aún no han sido visitadas. Por ello, se hace necesario relajar los efectos de la lista tabú y anular el estado tabú del movimiento en ciertas situaciones, lo que se lleva a cabo mediante los criterios de aspiración. Los criterios de aspiración son mecanismos usados para anular el estado tabú de los movimientos cuando sea apropiado hacerlo, es decir, cuando sean suficientemente buenos. El criterio de aspiración debe asegurar que volver a un movimiento reciente (es decir, presente en la lista tabú) dirija la búsqueda hacia soluciones que no se hayan visitado todavía y que generalmente son mejores. Un ejemplo de criterio de aspiración, consiste en aceptar una solución tabú, si dicha solución es mejor que la mejor solución encontrada hasta el momento. Este criterio de aspiración es el más utilizado pero existen otros criterios diferentes.

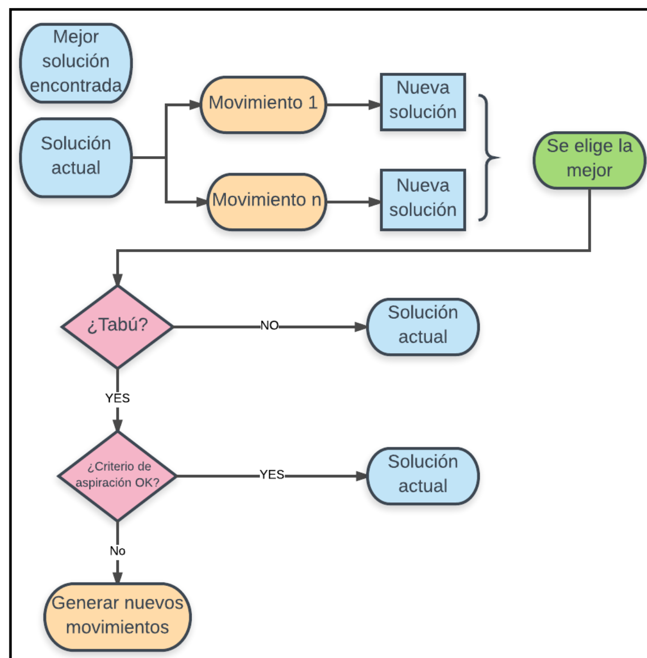


Figura 4.1. Diagrama de flujo del algoritmo de la búsqueda tabú

El algoritmo continúa evaluando los distintos movimientos, y se va actualizando la lista tabú, el número de iteración y la mejor solución encontrada hasta el

momento. El procedimiento sigue hasta un número fijado de iteraciones o hasta que se satisfaga un criterio de parada prefijado. Este proceso iterativo se ve reflejado en la Figura 4.1.

4.4. Atributos de los movimientos y restricciones tabú

Con el fin de evitar movimientos inversos a los ya realizados, se puede almacenar información acerca del propio movimiento, del movimiento inverso o ciertos atributos del movimiento. Cualquier aspecto que cambie como resultado de un movimiento desde la solución actual hasta la siguiente solución a evaluar, se considera como un atributo del movimiento.

Si se almacena el movimiento inverso, este será evitado, pero si se almacenan los atributos del movimiento, todos los movimientos con dichos atributos serán evitados mientras permanezca el estado tabú.

Supongamos que la solución a un problema se puede codificar con 0 y 1, en función de si las variables que la componen están presentes o no. Por ejemplo, si el problema se compone de las variables x_1, x_2, x_3, y x_4, una posible solución será [0,1,1,0], es decir, las variables x_2 y x_3 forman parte de la solución. Por tanto, el cambio de las variables de 0 a 1 y a la inversa, se pueden considerar atributos de un movimiento. Otro posible atributo a considerar, sería el cambio en la función coste de una solución a otra.

Continuando con el ejemplo propuesto, consideremos un movimiento que implica activar x_1 (x_1 = 1) y desactivar x_3 (x_3 = 0). Una secuencia de permutaciones de estos valores puede llevar a caer en un ciclo. Para evitar esto, se crean dos categorías de atributos: Desde y Hasta. En el ejemplo, el atributo del movimiento sería *Desde*[x_3] = 0 y como consecuencia, la restricción tabú necesaria es *Hasta*[x_3] = 0. De esta manera, se evita realizar movimientos cíclicos mediante las restricciones tabú, ya que las soluciones que implican que x_3 sea igual a 0 es tabú.

4.5. Tamaño de la lista tabú

Uno de los parámetros a determinar a la hora de implementar el algoritmo de la búsqueda tabú es el tamaño de la lista tabú, que puede ser fijo o variable. Generalmente el tamaño de la lista tabú es pequeño, para reflejar la componente de memoria a corto plazo. Algunos experimentos proponen utilizar un tamaño entre 5 y 12 iteraciones (Youssef, 1999), en función del espacio de búsqueda y del tipo de restricciones tabú empleadas. Para las listas tabú de

tamaño fijo, se vienen utilizando valores como 7, n o $\sqrt{n}$, donde n es un parámetro relacionado con el tamaño del problema. Por ejemplo, en el problema del enrutamiento de vehículos, se puede determinar el tamaño del problema mediante el número de clientes a visitar.

El tamaño puede ser determinado mediante la ejecución del programa y observar si se llega a situaciones cíclicas en caso de ser un tamaño demasiado pequeño, o si la calidad de las soluciones no es suficientemente buena debido a que el tamaño de la lista tabú es demasiado grande. También cabe la posibilidad de utilizar más de una lista tabú, asociada cada una de ellas a un atributo diferente, e incluso tener un tamaño distinto.

Las listas tabú dinámicas, es decir, con tamaño variable, nacen por la necesidad de adaptar el tamaño de la lista al momento de búsqueda, dando lugar a procesos más robustos que con listas de tamaño fijo. Si estamos próximos a un óptimo local es preferible emplear un tamaño de lista pequeño, mientras que un tamaño grande es útil para escapar de entornos con mínimos locales. Cuando se opta por usar listas dinámicas, el tamaño varía entre unos límites t_{min} y t_{max} y se actualiza durante la búsqueda.

4.6. Memoria a medio plazo (Intensificación de la búsqueda)

La componente de memoria a medio plazo mejora la calidad de la solución alcanzada mediante la búsqueda tabú. Gracias a este mecanismo la búsqueda se vuelve más agresiva. Se elige un número número m >> T (tamaño de la lista tabú) de mejores soluciones generadas durante un período particular de la búsqueda y sus características se almacenan y se comparan. Las características que más se repiten se buscan entre las nuevas soluciones. Para conseguir esto se penalizan los movimientos que eliminan dichas características.

Esta estrategia de intensificación es útil para resolver grandes problemas debido a que la búsqueda se centra en generar soluciones que son buenas.

4.7. Memoria a largo plazo (Diversificación de la búsqueda)

Los principios involucrados en este mecanismo son precisamente los opuestos a los de la componente de memoria a medio plazo. En lugar de intensificar la búsqueda en regiones ya visitadas que contienen buenas soluciones, la finalidad de la componente de memoria a largo plazo es conducir la búsqueda hacia nuevas regiones aún no exploradas. Sin esta herramienta la búsqueda

se podría localizar en una zona pequeña del espacio de soluciones, eliminando la posibilidad de encontrar el óptimo global.

La mayoría de las técnicas metaheurísticas incorporan este mecanismo mediante la aleatorización. Por ejemplo, en los algoritmos genéticos, se logra a través de los cruces, las mutaciones y la selección.

En concreto, la mayoría de las técnicas basadas en la búsqueda tabú penalizan aquellos atributos que son frecuentes entre las soluciones intermedias y aquellos movimientos que se llevan a cabo con cierta frecuencia. Este mecanismo puede implementarse por medio de un término de penalización que se calcula como el producto de tres factores:

a. Factor que mide la frecuencia del movimiento.
b. Factor que mide el tamaño del problema (por ejemplo, $\sqrt{n}$).
c. Factor de escala controlado por el usuario.

Implementar este mecanismo es efectivo y barato en términos de computación, por lo que en muchas ocasiones se tiene en cuenta durante la búsqueda.

4.8. La búsqueda tabú aplicada al problema VRP

En este apartado se procede a hacer una revisión de las principales variantes de la búsqueda tabú aplicadas al problema VRP siguiendo a (Cordeau & Laporte, 2002). Dicha exposición se realizará en orden cronológico y para cada una de ellas se cita el autor que lo propuso así como las principales características del método.

Pero antes de describir dichas variantes hay que hacer un apunte sobre los principales métodos para realizar movimientos en el problema VRP y de esta manera generar los entornos. Dichos métodos son el *k*-intercambio y las cadenas de eyección.

Los k-intercambios consisten en intercambiar hasta *k* clientes entre dos rutas, y el valor de *k* a menudo se limita a 1 ó 2 para reducir el número de posibilidades. Estos intercambios se pueden definir mediante parejas (k_1, k_2) con $k_1 < k$ y $k_2 < k$, donde k_1 clientes son eliminados de la ruta 1 e insertados en la ruta 2 y k_2 clientes son eliminados de la ruta 2 e insertados en la ruta 1. Por tanto, si $k = 2$ se pueden definir los siguientes movimientos (2,2), (2,1), (2,0), (1,1) y (1,0). En principio los intercambios k-intercambios no abarcan los

movimientos entre nodos de una misma ruta, pero se puede generalizar para permitir esta posibilidad.

Por otro lado, las *cadenas de eyección* (Rego, 2000) consisten en primer lugar en identificar un conjunto de rutas $R_1, R_2, \ldots, R_m$ y después en mover vértices de manera cíclica de la ruta R_1 a la ruta R_2, de la ruta R_2 a la ruta R_3, etc. Este mecanismo permite llevar a cabo movimientos intra-ruta e inter-ruta.

Las cadenas de eyección se definen como un conjunto de tripletes T ={ (v_{t-1}, v_t, v_{t+1}), … , (v_{i-1}, v_i, v_{i+1}), … , (v_{b-1}, v_b, v_{b+1}) }, y para simplificar se puede definir como un conjunto de vértices centrales $v_t, \ldots, v_i, \ldots, v_b$. El proceso de eyección resulta al mover un vértice a una nueva posición ocupada por otro vértice, desconectando este vértice de su posición.

En la Figura 4.2. se muestran dos mecanismos distintos de mover los vértices entre los distintos tripletes. En la opción (a) el número de nodos en las rutas se mantiene constante mientras que en la opción (b) el número de nodos en las rutas es variable.

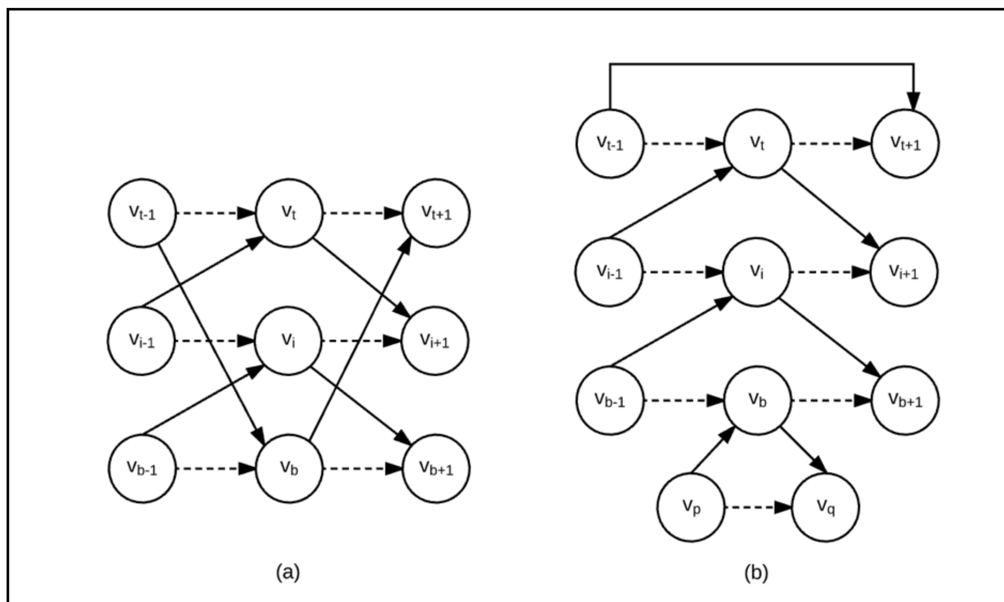


Figura 4.2. Dos mecanismos distintos de aplicar cadenas de eyección

4.8.1. Implementación de Willard

La primera implementación de la búsqueda tabú aplicada al problema VRP se debe a (Willard, 1989). Tal y como expone (Cordeau J. F., 2007), una solución para el problema CVRP se representa mediante una ruta que contiene varias

copias del almacén y las distintas cadenas se corresponden con rutas factibles asociadas a cada uno de los vehículos. Los entornos se definen por medio de intercambios 3-opt (ya descritos en el apartado 3.1.2.) y la solución actual se determina mediante el mejor movimiento que no sea tabú. Este método fue rápidamente superado por otros algoritmos más potentes como los que se describen en las secciones que siguen.

4.8.2. Algoritmo de Osman

Este algoritmo propuesto en (Osman, 1993) se basa en el concepto de los *k*-intercambios. En su implementación, Osman utilizó un valor de $k = 1$ para generar una solución inicial factible y un valor de $k = 2$ en la siguiente fase del método. Un valor de $k = 2$ permite realizar movimientos de un vértice o de dos, así como intercambios de uno o dos vértices entre distintas rutas.

Osman probó dos estrategias para seleccionar una solución del entorno. En la primera estrategia, denominada "Mejor Admisible", se selecciona la mejor solución que no sea tabú. En la otra estrategia, conocida como "Primera Mejor Admisible", se selecciona, si es que existe, la primera solución admisible que mejora a la solución anterior. En caso de que no exista se selecciona la mejor solución admisible.

Osman observó a través de pruebas experimentales que utilizando el mismo criterio de parada, la estrategia "Primera Mejor Admisible" produce soluciones que son ligeramente mejores que con la estrategia "Mejor Admisible". Por el contrario, la estrategia "Primera Mejor Admisible" es mucho más lenta que la estrategia "Mejor Admisible".

Esta implementación de la búsqueda tabú de Osman utiliza una permanencia en la lista tabú que es fija. Además, este método no tiene en cuenta ningún mecanismo de memoria a largo plazo ni tampoco lleva a cabo ningún proceso de intensificación durante la búsqueda.

4.8.3. Algoritmo Ruta Tabú

Tal y como se expone en (Cordeau, Gendreau, & Laporte, 1994), en el algoritmo Ruta Tabú (o Taburoute) las soluciones del entorno se consiguen moviendo un vértice desde su ruta actual R a otra ruta R' que contiene uno de sus vecinos más próximos. Esta inserción en la ruta R' se lleva a cabo en paralelo con una re-optimización local mediante el mecanismo GENI (Gendreau, Hertz, &

Laporte, 1992), lo que puede resultar en la creación de una nueva ruta o en la eliminación de una ruta ya existente. El mecanismo GENI se describe al final de este apartado.

Con el fin de limitar el tamaño del entorno, el autor solo tiene en cuenta en cada iteración un subconjunto formado por vértices elegidos al azar que se considerarán para reinsertarlos en otras rutas.

En este algoritmo se introdujo el concepto de penalización de la función objetivo, en caso de que no se cumplan las restricciones de capacidad o de duración total de las rutas. Dicha penalización se calcula según la expresión que aparece a continuación. En dicha expresión $c(x)$ es el coste de la solución x, $C(x)$ y $T(x)$ miden el exceso de capacidad y de duración respectivamente, α_c y α_d son unos parámetros que se van ajustando durante la búsqueda.

$$c'(x) = c(x) + \alpha_c C(x) + \alpha_d T(x)$$

En concreto en este algoritmo los parámetros α_c y α_d son actualizados cada diez iteraciones. De esta forma si las diez soluciones anteriores son factibles con respecto a la capacidad, entonces α_c se divide entre 2; en caso de que todas ellas fuesen no factibles, entonces α_c es multiplicado por 2. Si no se cumple ninguna de las dos condiciones anteriores, el valor de α_c no se modifica. De manera análoga se actualiza el valor de α_d con respecto a la restricción de duración de las rutas.

Un vértice que se elimina de una ruta R en la iteración t, no puede ser reinsertado en dicha ruta hasta la iteración $\theta + t$, donde θ toma un valor aleatorio en el intervalo [5, 10]. Esta condición se debe cumplir excepto si el movimiento produce una solución mejor que la mejor solución encontrada hasta el momento.

En este algoritmo se implementa un mecanismo que lleva a cabo un proceso continuo de diversificación. Las rutas se actualizan periódicamente usando la heurística US de post-optimización propuesta por (Gendreau, Hertz, & Laporte, 1992) para el problema del viajante. La heurística US de post-optimización se describe al final de este apartado.

La intensificación durante la búsqueda se implementa a través de dos mecanismos. En primer lugar se lleva a cabo una búsqueda limitada que comienza desde varias soluciones iniciales y, a continuación, se ejecuta la búsqueda tabú comenzando con la mejor solución obtenida. Además, si tras un número dado de iteraciones no hay mejora se intensifica la búsqueda

comenzando desde la mejor solución obtenida, pero permitiendo, en este caso, que un número mayor de vértices se consideren para cambiar de ruta.

4.8.3.1. Mecanismo GENI

El procedimiento GENI (Gendreau, Hertz, & Laporte, 1992) es un método de inserción, así como de post-optimización de rutas. Este método es aplicable tanto a problemas simétricos como a problemas asimétricos. La principal característica consiste en que la inserción de un vértice v no tiene que ser necesariamente entre dos vértices consecutivos. Sin embargo, después de la inserción, estos dos vértices quedan adyacentes al vértice v que se inserta.

Supongamos que queremos insertar un vértice v entre dos vértices v_i y v_j. Tomando uno de los dos posibles sentidos de la ruta, v_k es un vértice en el camino de v_j a v_i y v_l es un vértice de v_i a v_j. Para cada vértice v_h en la ruta, v_{h-1} es su vértice anterior y v_{h+1} su sucesor. La inserción de v entre v_i y v_j se puede hacer mediante dos tipos de inserción.

- Inserción tipo I: para esta inserción es necesario que $v_k \neq v_i$ y $v_k \neq v_j$. Insertar el vértice v en la ruta implica eliminar los arcos $(v_i,\ v_{i+1})$, $(v_j,\ v_{j+1})$, $(v,\ v_{k+1})$ y en consecuencia aparecen los arcos (v_i, v), (v, v_j), (v_{i+1}, v_k) y (v_{j+1}, v_{k+1}). Además los arcos (v_{i+1}, v_j) y (v_{j+1}, v_k) invierten su sentido. Este tipo de inserción se ilustra en la Figura 4.3.

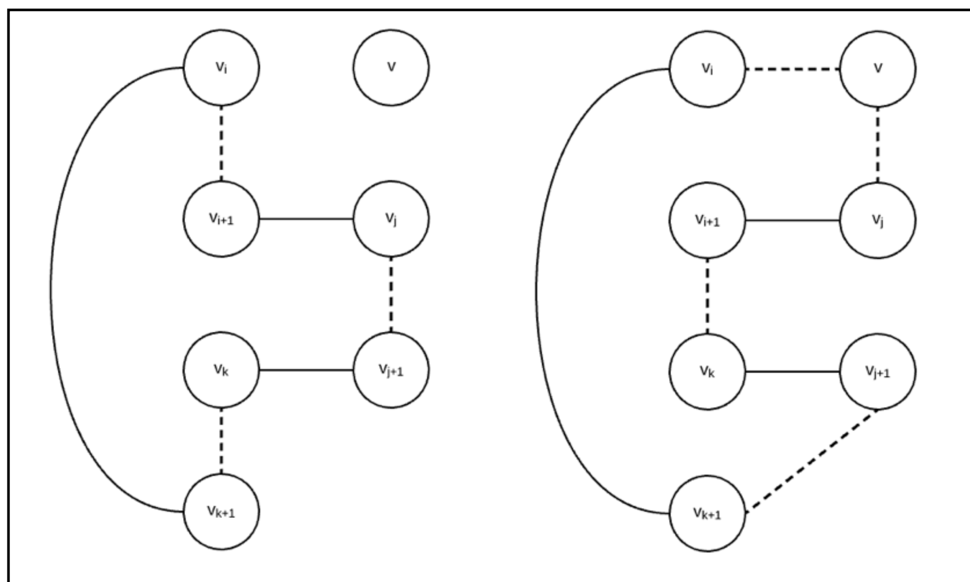


Figura 4.3. Tipo I de inserción.

- Inserción tipo II: aquí $v_k \neq v_j$, $v_k \neq v_{j+1}$, $v_l \neq v_i$, $v_l \neq v_{i+1}$. Insertar el vértice v en la ruta implica eliminar los arcos (v_j, v_{j+1}), (v_{k-1}, v_k), (v_i, v_{i+1}) y (v_{l-1}, v_l) y en consecuencia aparecen los arcos (v, v_j), (v_l, v_{j+1}), (v_{k-1}, v_{l-1}), (v_{i+1}, v_k) y (v_i, v). Los arcos (v_j, v_l) y (v_{i-1}, v_{l-1}) invierten su sentido. Este segundo tipo de inserción se ilustra en la Figura 4.4.

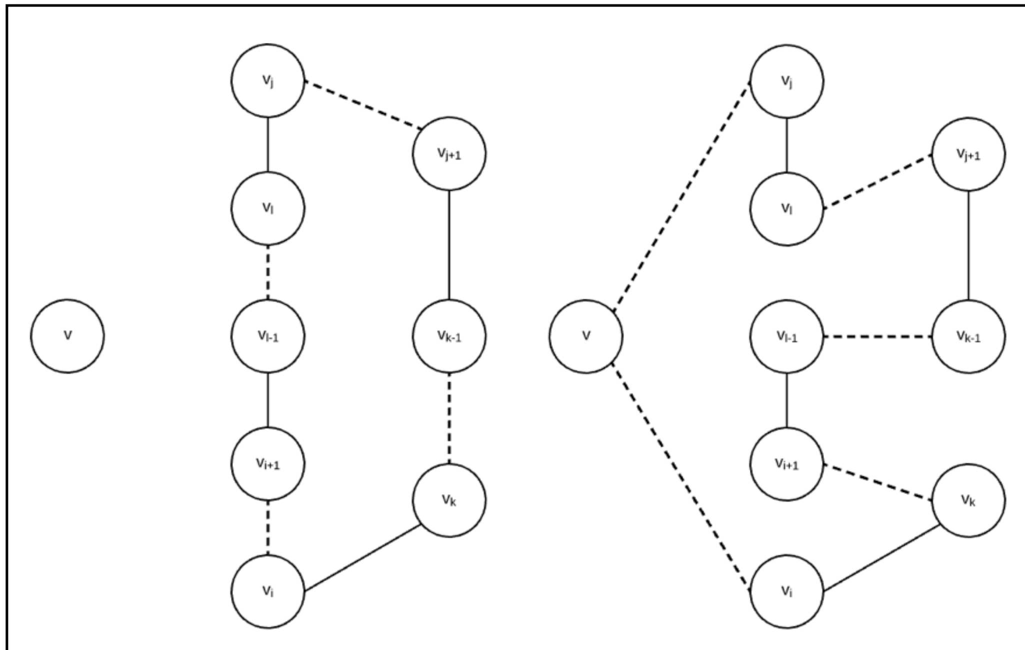


Figura 4.4. Tipo II de inserción

El algoritmo considera los dos posibles sentidos de la ruta para cada inserción. Ya que el número de opciones para elegir v_i, v_j, v_k y v_l es del orden n^4, la búsqueda se limita de manera que para cada vértice $v \in$ V se define su p-entorno $N_p(v)$ como un conjunto de p vértices cercanos a v (en términos de c_{ij}). Si v tiene menos de p vecinos, entonces todos los vértices pertenecen a $N_p(v)$. Por tanto para un parámetro dado p, primero se selecciona v_i, $v_j \in N_p(v)$, $v_k \in N_p(v_{i+1})$ y $v_l \in N_p(v_{j+1})$. También se consideran todas las inserciones del vértice v entre dos vértices consecutivos v_i y v_{i+1} si $v_i \in N_p(v)$. En la práctica p es un número relativamente pequeño.

El algoritmo GENI consta de los siguientes pasos:

- Paso 1: se crea una ruta inicial seleccionando arbitrariamente un subconjunto de tres vértices y se inicializan los p - entornos de todos los vértices.

– Paso 2: aleatoriamente se elige un vértice v que aún no está en la ruta. Se implementa la inserción de v de menor coste considerando los dos posibles sentidos de la ruta y los dos tipos de inserción. Se actualizan los p - entornos de todos los vértices para tener en cuenta el hecho de que v ya forma parte de la ruta.

– Paso 3: si todos los vértices forman parte de la ruta se para y en caso contrario se vuelve al Paso 2.

4.8.3.2. Algoritmo US – post-optimización

El algoritmo US (Gendreau, Hertz, & Laporte, 1992) es un algoritmo de post-optimización que consiste en eliminar un vértice de una ruta factible y en volverlo a insertar. El proceso de inserción es igual que el Paso 2 del algoritmo GENI y el proceso de eliminación del vértice de la ruta se puede llevar a cabo de dos maneras distintas que se describen a continuación.

- Tipo I de eliminación de cliente de la ruta: $v_j \in N_p(v_{i+1})$ y para una orientación dada de la ruta $v_k \in N_p(v_{i-1})$ es un vértice en el camino $(v_{i+1}, \dots, v_{j-1})$. Por tanto, se eliminan los arcos (v_{i-1}, v_i), (v_i, v_{i+1}), (v_k, v_{k+1}) y (v_j, v_{j+1}) y aparecen en su lugar los arcos (v_{i-1}, v_k), (v_{i+1}, v_j) y (v_{k+1}, v_{j+1}). Los caminos $(v_{i+1}, \dots, v_k)$ y $(v_{k+1}, \dots, v_{j+1})$ cambian su sentido. Todos estos cambios se ven reflejados en la Figura 4.5.

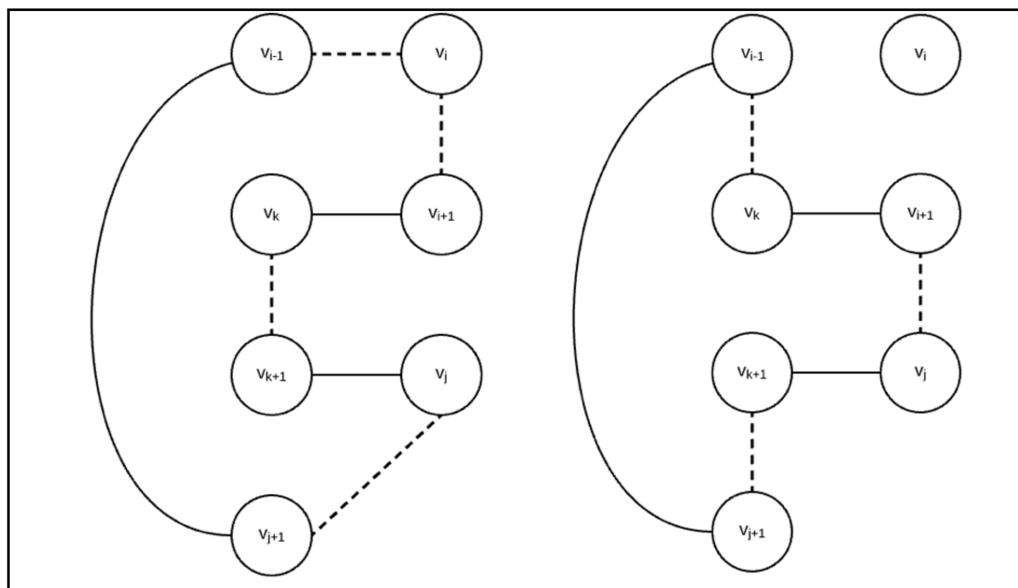


Figura 4.5. Tipo II de eliminación de cliente de una ruta.

- Tipo II de eliminación de cliente de la ruta: $v_j \in N_p(v_{i+1})$. Para una orientación dada de la ruta $v_k \in N_p(v_{i-1})$ es un vértice en el camino $(v_{j+1}, \dots, v_{i-2})$; $v_l \in N_p(v_{k+1})$ es un vértice en el camino $(v_j, \dots, v_{k-1})$. Por tanto, en este tipo de eliminación desaparecen los arcos (v_{i-1}, v_i), (v_i, v_{i+1}), (v_{j-1}, v_j) y (v_l, v_{l+1}). En su lugar, se insertan los arcos (v_{i-1}, v_k), (v_{i+1}, v_{j-1}), (v_{i+1}, v_j). Los caminos $(v_{i+1}, \dots, v_{j-1})$ y $(v_{l+1}, \dots, v_k)$ cambian su sentido. Se ilustra el cambio de arcos en la Figura 4.6.

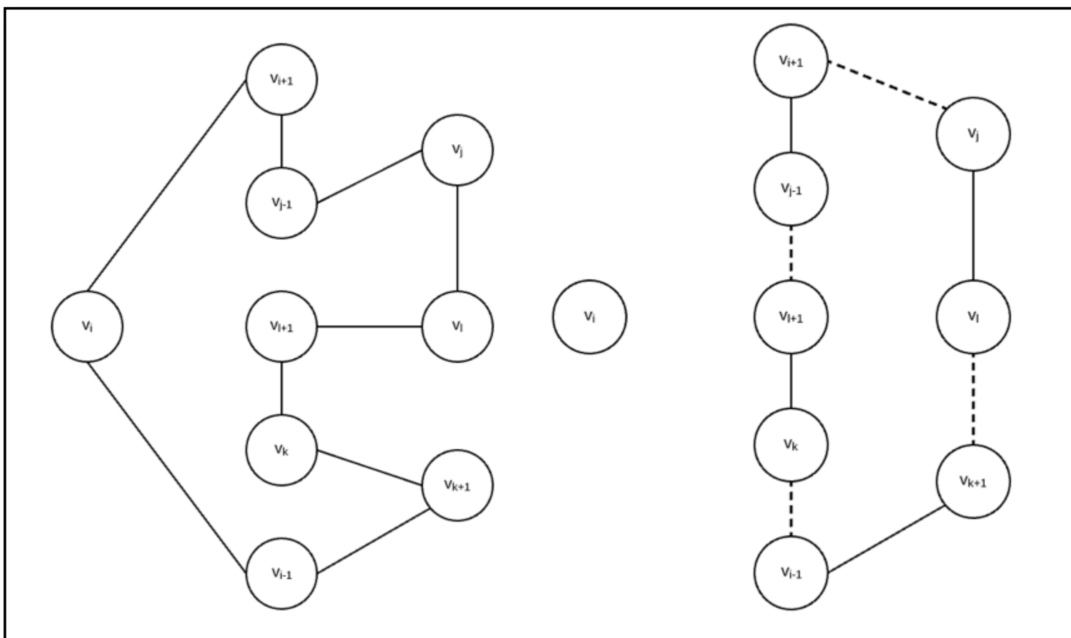


Figura 4.6. Tipo II de eliminación de cliente de una ruta

El algoritmo consta de los siguientes pasos:

– Paso 1: se considera una ruta inicial R con coste c. Se establece $R^* = R$, $c^* = c$ y t = 1.

– Paso 2: partiendo de la ruta R se aplican los métodos de eliminación e inserción al vértice v_t, considerando en cada caso las dos posibles variantes y los dos sentidos de la ruta. Entonces R' es la ruta obtenida y c'su coste. Ahora $R = R'$ y $c = c'$.

  - Si $c < c^*$: $R^* = R$, $c' = c$ y t = 1, repetir el Paso 2.

- Si $c \geq c^*$: $t = t + 1$.
- Si $t = n + 1$ se para (la mejor ruta obtenida es R^* y su coste c^*). En caso contrario se vuelve al Paso 2.

4.8.4. Algoritmo de Taillard

Aunque el algoritmo (Taillard E. , 1993) fue publicado en 1993, es decir, un año antes que el algoritmo Ruta Tabú, ambos métodos fueron desarrollados en el mismo periodo. Este algoritmo utiliza el mecanismo 1 – Intercambio con el fin de definir las soluciones que forman parte del entorno. Esto se lleva a cabo sin reoptimización local y sin permitir soluciones no factibles. Esta estructura es mucho más simple que la utilizada en el algoritmo Ruta Tabú y por tanto permite llevar a cabo más iteraciones en un mismo tiempo de ejecución.

El mecanismo tabú y la componente de memoria a largo plazo se implementan de la misma manera que en el algoritmo Ruta Tabú. Periódicamente las rutas son reoptimizadas usando el algoritmo exacto de (Volgenant & Jonker, 1983) para el problema del viajante.

El proceso de búsqueda desarrollado por Taillard emplea un esquema de descomposición que se utiliza para la programación en paralelo. En un problema VRP plano, el conjunto de clientes se divide en sectores, con centro en el almacén, o también en círculos concéntricos. De esta manera la búsqueda se realiza en cada subregión por un procesador distinto y se consigue la paralelización del problema. Los límites de las subregiones se actualizan periódicamente para producir un efecto de diversificación.

El algoritmo de Taillard es una de las mejores variantes de la búsqueda tabú aplicada al problema VRP encontradas hasta el momento.

4.8.5. Procedimiento de memoria adaptativa

El procedimiento de memoria adaptativa fue propuesto por (Taillard Y. R., 1995) y en su momento se presentó bajo el título “Diversificación e intensificación probabilística”. Este método no debería verse como una heurística VRP como tal, sino como un procedimiento general aplicable en distintos contextos y a distintas heurísticas. Aplicado al VRP, este procedimiento ha ayudado a mejorar algunas soluciones generadas mediante el algoritmo de Taillard.

4.8.6. Algoritmo de Xu y Kelly

El algoritmo propuesto en (Xu & Kelly, 1996) define los entornos utilizando cadenas de eyección e intercambios de vértices entre dos rutas. Las cadenas de eyección se determinan minimizando el flujo en una red como la mostrada en la Figura 4.7. Los arcos del nivel 2 y del nivel 3 tienen un flujo 1 ó 0 para indicar si un cliente es eliminado de una ruta (nivel 2) o reinsertado en una ruta (nivel 3).

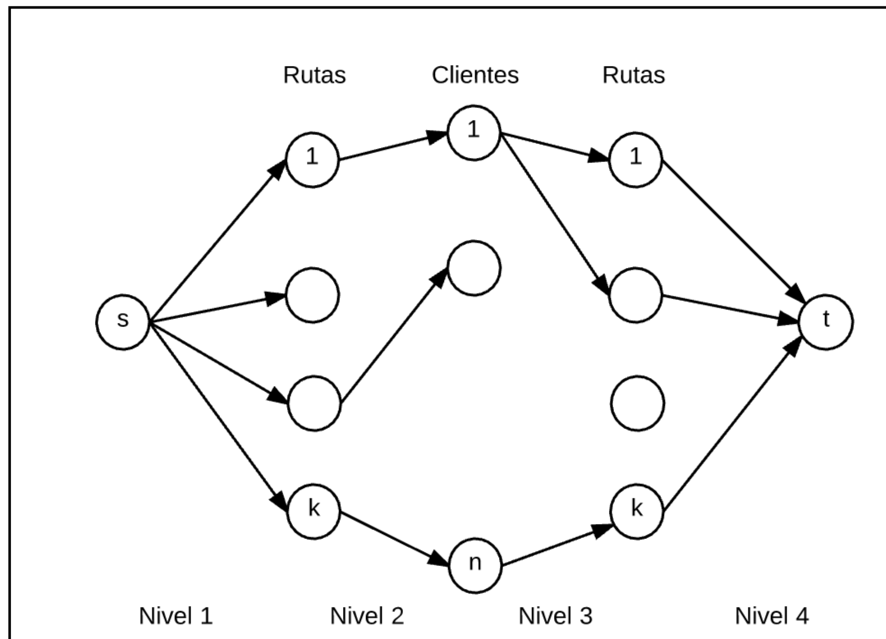


Figura 4.7. Flujo en redes para el algoritmo Xu y Kelly.

Debido a que varios clientes se pueden eliminar de una misma ruta o reinsertar en una misma ruta, los costes de los arcos son aproximaciones. Al igual que en algunos de los algoritmos descritos anteriormente, las rutas individuales son reoptimizadas periódicamente mediante operaciones 3-opt y 2-opt.

Este algoritmo permite seleccionar soluciones intermedias no factibles con respecto a la restricción de capacidad, utiliza una permanencia tabú fija y aplica una componente de memoria a largo plazo.

Durante la búsqueda se mantiene un conjunto de mejores soluciones y, el proceso de búsqueda se aplica periódicamente a este conjunto de mejores soluciones con el fin de ver si es posible generar mejores soluciones que las ya encontradas. En general, este algoritmo produce buenas soluciones para el problema VRP con restricción de capacidad, pero requiere bastante tiempo y es complejo porque es necesario dar valores a un conjunto de parámetros.

4.8.7. Algoritmo Cadena Tabú

El algoritmo Cadena Tabú o Tabuchain (Rego & Roucairol, 1996) utiliza cadenas de eyección involucrando l niveles o rutas para definir los entornos. Básicamente el proceso de eyección empuja un vértice de un nivel al nivel siguiente, empezando en el nivel 1 y terminando en el nivel l – 1, tal y como se muestra en la Figura 4.8.

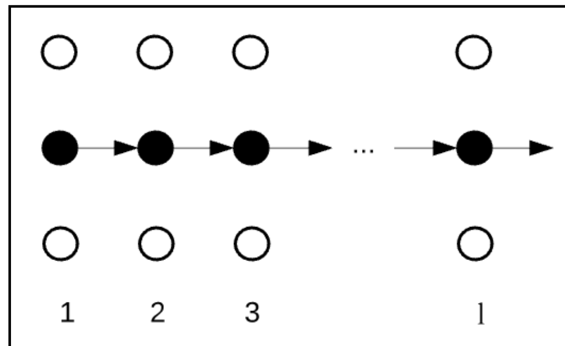


Figura 4.8. Cadena de eyección

En cualquier nivel de la cadenas de eyección el último vértice empujado se puede recolocar en la posición del vértice eliminado del nivel 1, o puede ser recolocado en cualquier posición, tal y como se ilustró en la introducción a este apartado. El proceso asegura que ningún arco sea considerado más de una vez en una solución; en cambio, se aceptan las rutas que violan la restricción de capacidad o duración. Este algoritmo fue implementado por sus autores tanto en versión secuencial como en versión paralela.

4.8.8. Algoritmo Flor

El algoritmo Flor o Flower (Rego, 1998) aplica un proceso de eyección para moverse desde una solución VRP compuesta por varias flores (blossoms) o rutas a otra solución mediante la eliminación e introducción de arcos. Aplicando estas operaciones se pueden crear estructuras intermedias que consisten en un camino (stem), que parte del depósito y varias flores unidas a él. Así, una flor se puede transformar en un camino o se puede dividir en una flor y un camino.

Durante este proceso el número de rutas activas puede variar. Las soluciones candidatas que consisten solo en flores se obtienen llevando a cabo movimientos de eyección que mantienen la estructura de flor. Esto se consigue eliminando unos arcos y añadiendo otros.

Este proceso incluye un mecanismo tabú que no permite reintroducir un arco que acaba de ser eliminado. Como en alguno de los algoritmos descritos anteriormente, la permanencia tabú es variable y se utiliza una estrategia de diversificación basada en la frecuencia de los movimientos. Además, la penalización es aleatoria y se genera en un intervalo cuya amplitud está relacionada con $\sqrt{n}$, donde n es el tamaño del problema.

4.8.9. Algoritmo de la búsqueda tabú granular

Este algoritmo (Toth & Vigo, 2003) no es aplicable solamente al problema VRP o a los algoritmos de búsqueda tabú, sino que es aplicable a todo el campo de la optimización combinatoria. El algoritmo de la búsqueda tabú granular se describe en detalle en el Capítulo 5 ya que es el método que se ha programado.

4.8.10. Algoritmo de búsqueda tabú unificada

Este algoritmo fue desarrollado en un principio para el problema VRP periódico (en el cual se considera el problema VRP a lo largo de varios días) y para el VRP con varios almacenes (Cordeau, Gendreau, & Laporte, 1997). Aplicado a estas variantes, el algoritmo de búsqueda tabú unificada produce buenos resultados. Posteriormente dicho método fue aplicado al VRP dependiente del lugar con ventanas de tiempo (Cordeau & Laporte, 2001), al VRP periódico con ventanas de tiempo, al VRP con varios almacenes con ventanas de tiempo (Cordeau & Laporte, 2001) y finalmente al problema VRP clásico (Cordeau, Gendreau, Laporte, Potvin, & Semet, 2002) con restricciones de capacidad y longitud máxima en las rutas.

Este método utiliza varias características descritas en el algoritmo Ruta Tabú, como la estructura de los entornos, inserciones GENI y la componente a largo plazo basada en penalizaciones.

Por el contrario hay aspectos en los que este algoritmo es diferente del algoritmo Ruta Tabú ya que en este método, la búsqueda se aplica a una única solución inicial, la permanencia tabú es fija y no se emplean mecanismos de intensificación de la búsqueda. Por tanto, este método gana en eficiencia sacrificando la sofisticación del algoritmo Ruta Tabú.

El algoritmo permite seleccionar soluciones intermedias no factibles, pero divide o multiplica α_c y α_d por un factor $1 + \delta$ (con δ un número pequeño y positivo) en cada iteración en función de si la solución previa fue factible o no con respecto a las restricciones de capacidad y duración de las rutas.

El mecanismo tabú opera sobre un atributo $B(x) = \{(i,k)$: v_i es visitado por un vehículo k en la solución $x\}$. Las soluciones del entorno se obtienen eliminando (i,k) de $B(x)$ y reemplazándolo por (i,k') donde $k' \neq k$. El atributo (i,k) se declara tabú por un número de iteraciones y el criterio de aspiración se define en relación a dicho atributo (i,k).

Cabe resaltar dos características del algoritmo de la búsqueda tabú unificada: su simplicidad, ya que opera con pocos parámetros y su flexibilidad. Este algoritmo ha arrojado buenos resultados aplicado a una gran cantidad de variantes del VRP manteniendo sus parámetros al mismo valor.

Es evidente que son muchos los métodos de búsqueda tabú disponibles para la resolución del problema de enrutamiento de vehículos y por ello el trabajo se centra en la búsqueda tabú granular, que se explica al detalle en el capítulo que sigue.

Capítulo 5

Búsqueda tabú granular aplicada al VRP

En este capítulo se describe en detalle la búsqueda tabú granular aplicada al problema VRP. En primer lugar, se hace una introducción a los conceptos en los que se basa el método y, luego, se describen los diferentes movimientos utilizados para generar los entornos granulares. En último lugar, se explica el procedimiento que sigue la búsqueda durante las iteraciones del algoritmo y los resultados que este método ofrece.

5.1. Introducción

Como ya se ha indicado anteriormente, el algoritmo de la búsqueda tabú se encuentra entre uno de los mejores métodos aplicados a problemas de optimización de tipo NP-Duro. Sin embargo, el tiempo de computación necesario para encontrar buenas soluciones es generalmente mayor que con otros métodos tradicionales, como por ejemplo las heurísticas constructivas que requieren un tiempo de computación menor.

Por ello, en el presente capítulo se expone una herramienta de intensificación en la búsqueda propuesta por (Toth & Vigo, 2003) que puede ser usada en la búsqueda tabú, así como en otras técnicas heurísticas y metaheurísticas. Esta herramienta es posible aplicarla a múltiples problemas de optimización, así como a distintas clases de grafos. Entre estos problemas se pueden citar el problema del viajante y otros problemas de rutas y secuenciación en los cuales el coste de la solución viene determinado por la suma de los costes de los elementos que pertenecen a la propia solución.

El método se basa en el uso de entornos granulares, que son entornos que se obtienen eliminando movimientos que es probable que no produzcan soluciones factibles de buena calidad. Por tanto, los entornos granulares hacen que la búsqueda evalúe en cada iteración solo aquellos movimientos que son prometedores. De esta manera, el tiempo requerido para alcanzar soluciones de buena calidad, es normalmente mucho menor que el requerido cuando se emplean entornos completos.

A continuación se describe la herramienta de intensificación mediante entornos granulares para la búsqueda tabú aplicada al problema VRP. En el Capítulo 6 se muestra su aplicación a un problema concreto para ilustrar en detalle el funcionamiento de dicho método.

5.2. El problema

En la exposición del método se trabajará con un problema VRP con restricción de capacidad y de longitud máxima en las rutas, que se describió anteriormente en el Capítulo 2. Se considera un grafo en el que todas las posibilidades de unión entre los distintos vértices son posibles, es decir, un grafo completo $G = (V, E)$, donde $V = \{v_0, \dots, v_n\}$ es el conjunto de vértices y E es el conjunto de arcos. Los vértices $v_i = v_1, \dots, v_n$ corresponden a los n clientes a los que hay que servir, cada uno de ellos con una demanda d_i, no negativa. El vértice v_0 representa el almacén y por tanto se le asocia una demanda $d_0 = 0$.

A cada arco $(i, j) \in E$ se le asocia un coste c_{ij}. Estos costes se pueden almacenar en una matriz de costes, y se considera que $c_{ik} + c_{kj} \geq c_{ij}$, es decir, el camino más corto entre los vértices v_i y v_j es más corto que si se pasa por el vértice intermedio v_k. Se dispone de K vehículos idénticos, cada uno con una capacidad C.

Para la búsqueda tabú granular también se puede considerar el problema VRP con restricción en la distancia de las rutas. En ese caso cada vértice tiene asociado un tiempo de servicio s_i, de forma que para cada ruta la suma de la longitud total de los arcos más los tiempos de servicio de los clientes servidos no exceda un límite L. La longitud de cada arco es equivalente a su coste, y en general, los tiempos de servicios son todos 0 o todos igual a un valor común ($s_i = s$ para $i = 1, \dots, n$).

5.3. Movimientos de *k*-Intercambio

Los algoritmos de búsqueda local aplicados al problema VRP generalmente utilizan entornos basados en intercambios de arcos o movimientos de clientes. En particular, dada una solución actual *R*, un entorno basado en *k-Intercambios* se construye con todos los movimientos obtenidos eliminando un número *k* de arcos usados en *R* y reemplazándolos por otros *k* arcos, de forma que se consigue una nueva solución. Ya que la cardinalidad de un entorno generado mediante un *k*-intercambio es O(n^k), para limitar el tiempo de computación

requerido para evaluar el entorno, en la práctica se toman valores pequeños de *k* como *k* = 2, 3, 4.

En las siguientes figuras se muestran los movimientos para generar los entornos granulares. Denotamos por π_i el vértice anterior y por σ_i el vértice siguiente a un vértice dado $v_i \in V$ de la solución actual. Además, σ_{σ_i} es el segundo vértice que sigue al vértice v_i. En las figuras se representa con líneas discontinuas los arcos que son eliminados, y en línea continua los arcos que forman parte de la solución tras el intercambio. En concreto, en la Figura 5.1. se ilustra el 2-intercambio considerado, en la Figura 5.2. aparecen las dos variantes de 3-intercambio utilizadas, y por último en la Figura 5.3. se muestra el 4-intercambio elegido, en el cual dos pares de arcos consecutivos son eliminados.

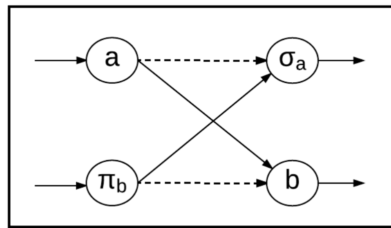


Figura 5.1. 2-Intercambio.

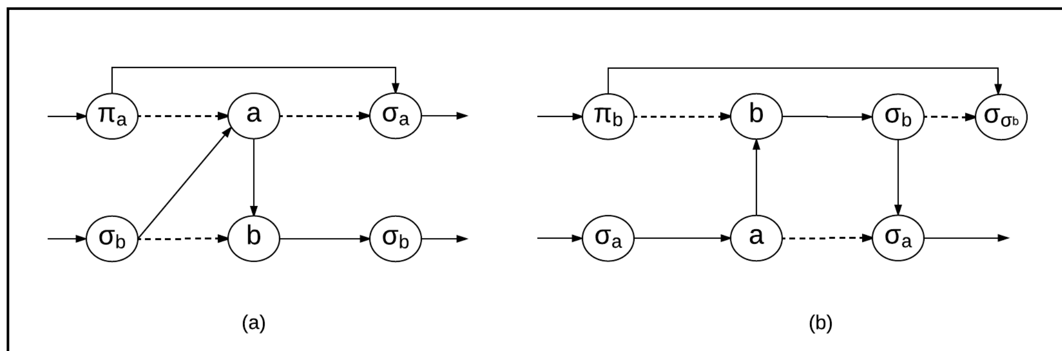


Figura 5.2. 3- Intercambios.

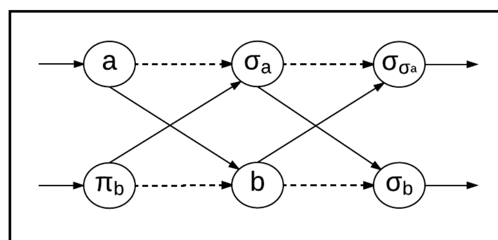


Figura 5.3. 4-Intercambio

Como se puede observar, la cardinalidad de cualquiera de los entornos generados por cada uno de los movimientos es O(n^2), ya que aunque se pudieran ver involucrados dos o más arcos a la vez, uno o dos de estos arcos quedan implícitamente determinados después de que los dos primeros arcos a eliminar hayan sido elegidos.

5.4. El método

La búsqueda tabú granular es una estrategia que persigue reducir, en cada iteración, el tiempo de cómputo a la hora de explorar los entornos. Este objetivo se consigue utilizando entornos que pueden ser examinados en mucho menos tiempo en comparación con el método tradicional y sin afectar a la calidad de las soluciones encontradas. Con este fin, se descarta una gran cantidad de movimientos que son poco prometedores y se explora un pequeño subconjunto que contiene aquellos movimientos de los que se espera obtener buenos resultados. La idea general de la búsqueda tabú granular consiste en la intuición de que los “arcos largos”, es decir, aquellos que tienen asociado un coste alto, tienen baja probabilidad de formar parte de la solución.

La búsqueda tabú granular comienza con una solución elegida al azar u obtenida mediante alguna técnica heurística. Uno de los métodos para generar la solución inicial es el propuesto por (Clarke & Wright, 1964), descrito en el apartado 3.1.2. Se propone utilizar este método para generar una solución inicial porque es una heurística constructiva que genera con poco esfuerzo una solución a partir de los clientes que forman parte del problema. Esta heurística utilizada no garantiza una solución que cumpla las restricciones en cuanto al número de vehículos empleados. En caso de que esta solución supere el número K de vehículos, las rutas menos “cargadas” se eliminan y los clientes se reinsertan en una de las K rutas en la mejor posición posible. Por tanto, la solución inicial puede ser no factible, en lo referente a las restricciones de capacidad de los vehículos o la restricción de longitud máxima de la ruta. Se verá un ejemplo a modo de ilustración de esta situación en el Capítulo 6.

Partiendo del grafo original $G = (V, E)$ descrito anteriormente, se define un nuevo grafo disperso $G' = (V, E')$, con $|E'| << n^2$, es decir, que el número de arcos sea mucho menor que el número de arcos considerados inicialmente. Este grafo disperso incluye aquellos arcos que deben ser considerados para ser incluidos en la solución actual. Entre estos arcos se encuentran los “arcos cortos” y un subconjunto de arcos que pueden llegar a ser importantes, como aquellos arcos que unen los vértices con el almacén y otros arcos que aparecen en las mejores soluciones encontradas hasta el momento.

Para determinar qué arcos son “cortos” y cuáles son “largos” se define una regla, por la cual, un arco es “corto” si su coste no es mayor que el valor del umbral de granularidad ϑ. Dicho umbral se calcula según la fórmula que aparece a continuación. En dicha fórmula β es un parámetro positivo relacionado con la dispersión, c' es el valor de una solución, n es el número de clientes y K el número de vehículos.

$$\vartheta = \beta \cdot \frac{c'}{(n+K)'}$$

Por tanto, en el subconjunto de arcos E', se incluyen los arcos cuyo coste se encuentren por debajo del valor del umbral de granularidad ϑ, los arcos incidentes en el almacén, los arcos pertenecientes a la mejor solución encontrada hasta el momento y los arcos de la solución actual.

Los arcos pertenecientes al subconjunto E' del grafo disperso, se utilizan como generadores de movimientos y determinan los arcos que se ven involucrados en un movimiento particular, ya que al tomar un arco (a, b) los otros arcos que participan en el movimiento quedan implícitamente determinados. De esta manera, el número total de movimientos evaluados en la búsqueda granular en un entorno es $O(|E'|)$.

Otro aspecto importante de la búsqueda tabú granular es la modificación dinámica del grafo disperso para incluir la diversificación y la intensificación durante la búsqueda. Por ejemplo, variando el valor del parámetro de dispersión β el valor del umbral de granularidad se modifica, en consecuencia los arcos que forman el grafo disperso varían, y de esta forma es posible explorar diferentes entornos durante la búsqueda. Con este fin, el algoritmo de la búsqueda tabú granular alterna pasos de intensificación con valores bajos de β, y pasos de diversificación que se consiguen aumentando el valor de β. Estos últimos pasos de diversificación hacen posible la inclusión en la solución de nuevos arcos, que serán más largos cuanto mayor sea el valor de β, ya que aumenta el valor umbral de granularidad ϑ. Por tanto, la búsqueda tabú granular aplica los movimientos generados por k-intercambios, ya descritos anteriormente, y el grafo disperso se reconstruye totalmente cada $2n$ iteraciones.

Durante la búsqueda es posible visitar soluciones no factibles con respecto a la restricción de capacidad o de longitud, lo que se refleja añadiendo al coste de la ruta una penalización al igual que en el algoritmo Ruta Tabú descrito en el apartado 4.8.3. Recordemos que consistía en sumar el exceso de capacidad

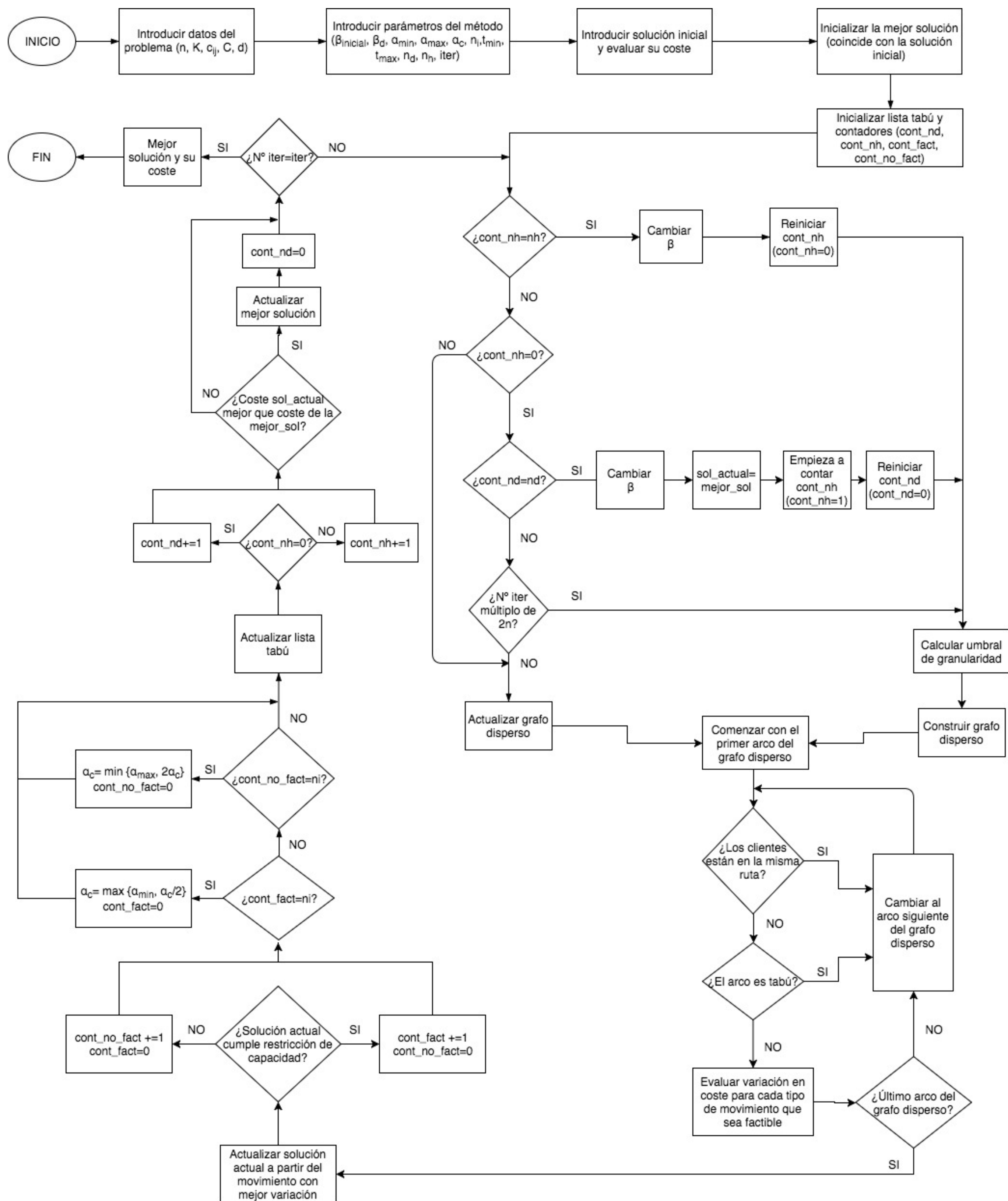
multiplicado por el parámetro de penalización $\propto_C$ más el exceso en la distancia por el parámetro de penalización $\propto_D$.

Los valores de los parámetros de penalización $\propto_C$ y $\propto_D$ se actualizan durante la búsqueda tomando valores dentro de un rango $[\propto_{min}, \propto_{max}]$. En concreto, cada n_i iteraciones, si todas las n_i soluciones visitadas anteriormente son factibles con respecto a la restricción de capacidad, entonces el valor de $\propto_c$ se actualiza al máximo entre $\{\propto_{min},\ \propto_C/2\}$. Por el contrario, si no son factibles se establece $\propto_C$ al valor mínimo de $\{\propto_{max},\ 2\propto_C\}$. De manera análoga se actualiza el valor del parámetro $\propto_D$.

Un movimiento se considera tabú si intenta reinsertar un arco que se ha eliminado en los movimientos previos. La permanencia tabú t para cada movimiento es un número entero aleatorio y uniformemente distribuido en el intervalo $[t_{max},\ t_{min}]$.

Cuando la mejor solución encontrada hasta el momento no mejora después de n_d iteraciones, se aumenta el valor del factor de diversificación hasta β_d, en consecuencia se actualiza el grafo disperso, y se realizan n_h $(n_h = n)$ iteraciones comenzando desde la mejor solución encontrada. Posteriormente el parámetro de diversificación vuelve a su valor original y la búsqueda continúa.

A continuación se presenta un diagrama de flujo para la búsqueda tabú granular, con el fin de aclarar el funcionamiento del proceso de búsqueda. En el diagrama se omite la penalización por exceso de distancia en las rutas, ya que el funcionamiento es exactamente igual que la penalización por exceso de carga en las rutas.



INICIO
Introducir datos del problema (n, K, c_{ij}, C, d)
Introducir parámetros del método ($\beta_{inicial}$, β_d, α_{min}, α_{max}, α_c, n_i, t_{min}, t_{max}, n_d, n_h, iter)
Introducir solución inicial y evaluar su coste
Inicializar la mejor solución (coincide con la solución inicial)
Inicializar lista tabú y contadores (cont_nd, cont_nh, cont_fact, cont_no_fact)
¿Nº iter=iter?
SI
NO
Mejor solución y su coste
FIN
¿cont_nh=nh?
Cambiar β
Reiniciar cont_nh (cont_nh=0)
¿cont_nh=0?
¿cont_nd=nd?
Cambiar β
sol_actual= mejor_sol
Empieza a contar cont_nh (cont_nh=1)
Reiniciar cont_nd (cont_nd=0)
¿Nº iter múltiplo de 2n?
Calcular umbral de granularidad
Construir grafo disperso
Actualizar grafo disperso
Comenzar con el primer arco del grafo disperso
¿Los clientes están en la misma ruta?
Cambiar al arco siguiente del grafo disperso
¿El arco es tabú?
Evaluar variación en coste para cada tipo de movimiento que sea factible
¿Último arco del grafo disperso?
Actualizar solución actual a partir del movimiento con mejor variación
¿Solución actual cumple restricción de capacidad?
cont_no_fact +=1 cont_fact=0
cont_fact +=1 cont_no_fact=0
¿cont_fact=ni?
α_c= max {α_{min}, α_c/2} cont_fact=0
¿cont_no_fact=ni?
α_c= min {α_{max}, 2α_c} cont_no_fact=0
Actualizar lista tabú
¿cont_nh=0?
cont_nd+=1
cont_nh+=1
¿Coste sol_actual mejor que coste de la mejor_sol?
Actualizar mejor solución
cont_nd=0

5.5. Resultados computacionales

En la presente sección se exponen los resultados computacionales del estudio llevado a cabo sobre el algoritmo de búsqueda tabú granular de (Toth & Vigo, 2003). En dicho estudio el algoritmo fue codificado en Fortran 77 y probado con un procesador Pentium 200 MHz. El conjunto de pruebas computaciones consideran varios problemas Euclídeos VRP y DVRP con hasta 500 clientes que son usados como punto de referencia para la comparación de los distintos algoritmos aplicados al problema VRP ((Golden & al, 1998).

Para denotar a cada problema, los autores siguieron una codificación del tipo: $tnnn - kkp$ donde:

- t: tipo de problema. Toma valor E para los problemas Euclídeos VRP y valor D para los problemas Euclídeos DVRP.
- nnn: número de vértices incluidos el almacén.
- kk: número de vehículos disponibles.
- p: identifica el artículo de origen en el que el problema fue propuesto.

Los problemas considerados en las pruebas experimentales son:

- Los catorce problemas clásicos VRP y DVRP descritos en (Christofides & Eilon, 1969) y (Christofides & al., 1979).
- Tres problemas Euclídeos propuestos por (Fisher, 1994).
- Un conjunto de veinte problemas Euclídeos VRP y DVRP propuestos por (Golden & al, 1998).

En la Figura 5.4. se muestran los principales datos de algunos de los problemas considerados. Para cada problema, la tabla recoge el número de clientes (n), el número de vehículos (K), la capacidad de los vehículos (C) y para los problemas DVRP también se aporta la longitud máxima de las rutas (L) y el tiempo de servicio para cada cliente (s).

La tabla recoge además para cada problema el artículo en el que se propuso, el coste de la mejor solución conocida hasta entonces y el artículo en el que esa solución fue reportada. La última columna de la tabla contiene, en caso de que existan, los valores de la nueva mejor solución encontrada mediante el método de búsqueda tabú granular en las pruebas llevadas a cabo.

Notación del problema	n	K	C	L	s	Fuente de los datos del problema	Mejor solución encontrada hasta entonces	Artículo de la solución	Nueva mejor solución
E045-04f	44	4	2010	-	-	Fisher (1994)	723,54	Fisher (1994)	
E051-05e	50	5	160	-	-	Christofides and Eilon (1969)	524,61	Gendreau et al (1994)	
E241-22k	240	22	200	-	-	Golden et al. (1998)	720,44	Golden et al (1998)	709,90
E484-19k	483	19	1000	-	-	Golden et al. (1998)	1.137,18	Golden et al (1998)	1.136,05
D051-06c	50	6	160	200	10	Christofides et al. (1979)	555,43	Taillard (1993)	
D151-14c	150	14	200	200	10	Christofides et al. (1979)	1.162,55	Taillard (1993)	
D361-09k	360	9	900	1300	0	Golden et al. (1998)	11.047,69	Golden et al (1998)	10.515,33
D481-12k	480	12	1000	1600	0	Golden et al. (1998)	14.639,32	Golden et al (1998)	14.336,36

Figura 5.4. Algunos problemas considerados en las pruebas computaciones.

El objetivo del test computacional consiste en evaluar el comportamiento del algoritmo de la búsqueda tabú granular teniendo en cuenta el equilibrio entre la calidad de la solución y el tiempo de computación requerido. Con este fin, para cada problema se comparan los resultados del algoritmo de búsqueda tabú granular con los algoritmos de búsqueda tabú que se probaron en los problemas anteriormente, de los cuales se dispone del tiempo de computación requerido para su ejecución.

Por esta razón se aportan en la Figura 5.5. los tiempos medios de ejecución para los distintos algoritmos estudiados y un factor ρ que permite calcular una aproximación de la equivalencia de dichos tiempos a tiempos de un Pentium 200MHz.

Referencia	Tiempo Max (min)	Tiempo Medio (min)	ρ^5
Taillard (1993)	65	22	0,3
Rochat y Taillard (1995)	45	16	1
Gendreau et al. (1994)	100	45	0,4
Rego y Roucairol (1996)	52	15	0,25
Xu y Kelly (1996)	368	130	2

Figura 5.5. Tabla para transformación de tiempos a un Pentium 200MHz.

En la Figura 5.6. se comparan los resultados obtenidos para algunos de los catorce problemas clásicos VRP y DVRP mediante el algoritmo de la búsqueda tabú granular con los resultados obtenidos por (Cordeau, Gendreau, & Laporte, 1994), (Xu & Kelly, 1996) y (Rego & Roucairol, 1996) (representados por las siglas GHL, XK y RR respectivamente). Para cada problema, la tabla recoge la variación, expresada en porcentaje, del valor de la solución con respecto al valor de la mejor solución encontrada, así como el tiempo de computación necesario expresado en minutos. Además aparece un valor medio para las variaciones y los tiempos, para algunos de los catorce problemas clásicos. Para consultar la tabla completa diríjase a (Toth & Vigo, 2003).

		Búsqueda tabú granular			GHL		XK		RR	
Problema	Mejor solución	Solución	%	Tiempo	%	Tiempo	%	Tiempo	%	Tiempo
E051-05e	524,61	524,61	100	0,81	100	6	100	29,92	100	0,85
E076-10e	835,26	838,6	100,4	2,21	100,06	53,8	100	48,8	100,27	16,8
E200-17c	1291,45	1318,25	102,08	7,5	102,42	90,9	100,55	272,52	103,64	16,25
Media VRP			100,47	3,1	100,95	38,01	100,09	102,99	100,96	14,65
D051-06c	555,43	555,43	100	0,86	100	13,5	100	30,67	100	3,17
D101-09c	865,94	869,48	100,41	2,9	100	25,6	101,78	98,15	100,27	8,6
D200-18c	1395,85	1435,74	102,86	9,11	101,62	99,8	103,11	368,37	101,79	52,02
Media DVRP			100,81	4,58	100,78	55,63	103,62	160,3	100,58	16,26

Figura 5.6. Comparación resultados obtenidos para los problemas clásicos.

En la Figura 5.7. se consideran los problemas Euclídeos VRP propuestos por (Fisher, 1994) y se comparan los resultados obtenidos mediante el algoritmo de búsqueda tabú granular con los resultados obtenidos mediante el algoritmo de (Xu & Kelly, 1996).

			Búsqueda tabú granular		XK	
Problema	Mejor solución	Solución	%	Tiempo	%	Tiempo
E045-04f	723,54	727,75	100,58	0,67	100	1,43
E072-04f	241,97	241,97	100	0,6	101,06	86,65
E135-07f	1162,96	1165,88	100,25	10,64	101,18	191,46
Media			100,28	3,97	100,75	93,18

Figura 5.7. Comparación resultados obtenidos para los problemas propuestos por Fisher.

Como se puede observar en las tablas anteriores, el algoritmo de la búsqueda tabú granular es capaz de determinar, en un tiempo de computación relativamente corto, buenas soluciones de calidad comparable a la obtenida mediante otros métodos propuestos de búsqueda tabú. En concreto, tal y como se expone en (Toth & Vigo, 2003) el algoritmo de la búsqueda tabú granular es peor que el algoritmo GHL en tan solo uno de los siete problemas clásicos VRP. Para los problemas DVRP, el algoritmo GHL es mejor que el algoritmo de la búsqueda tabú granular en tres de los siete casos estudiados, y el porcentaje medio de actuación es similar en dichos algoritmos. Los tiempos de computación del algoritmo de la búsqueda tabú granular es aproximadamente un quinto del tiempo de ejecución requerido en el algoritmo GHL.

En la comparación entre la búsqueda tabú granular y el algoritmo XK en los siete problemas clásicos VRP, el algoritmo XK generalmente arroja mejores soluciones. En cambio, la búsqueda tabú granular resulta más efectiva que el algoritmo XK para los problemas clásicos DVRP así como para los problemas VRP propuestos por Fisher. Además el tiempo de computación del algoritmo de

la búsqueda tabú granular es entre 50 y 70 veces menor que el tiempo de ejecución requerido para el algoritmo XK.

Finalmente, el algoritmo de búsqueda tabú granular es similar al algoritmo RR en términos de calidad de la solución obtenida y del tiempo de computación. En concreto, la búsqueda tabú granular es algo mejor para los problemas VRP y algo peor para los DVRP. Los tiempos de computación son, en promedio, similares en ambos algoritmos.

En la Figura 5.8. se hace una comparación de algunos problemas VRP y DVRP propuestos por (Golden & al, 1998). Los resultados completos se pueden consultar en (Toth & Vigo, 2003). En este caso los resultados obtenidos son comparados con el algoritmo XK y con el algoritmo de recocido simulado (RTR) descrito en (Golden & al, 1998). Para estos problemas, la búsqueda tabú granular determina trece nuevas soluciones mejores en un promedio de un 1% respecto a las soluciones del RTR. En cuanto a los tiempos de computación, la búsqueda tabú granular requiere tiempos similares a los que requiere el algoritmo RTR y son unas 300 veces menores que los del algoritmo XK.

		Búsqueda tabú granular			XK		RTR	
Problema	Mejor solución	Solución	%	Tiempo	%	Tiempo	%	Tiempo
E241-22k	720,44	711,07	98,7	14,29	103,72	2314	100	5,69
E253-27k	881,04	868,8	98,61	11,43	100	1465,77	100	6,01
E481-38k	1656,66	1652,32	99,74	23,07	100	8943,45	100,08	47,55
Media			100,05	23,32	101,32	3512,13	100,05	32,51
D201-05k	6702,73	6697,53	99,92	2,38	----	591,4	100	11,24
D361-09k	11047,69	10547,44	95,47	11,66	----	1062,73	101,5	22,55
D481-12k	14639,32	14910,62	101,85	15,13	----	2432,42	100	122,61
Media			98,87	8,89	----	1254,64	101,34	44,12

Figura 5.8. Comparación resultados obtenidos para algunos problemas propuestos en (Golden & al, 1998)

Se puede concluir que cuando se ejecuta el algoritmo de búsqueda tabú con los entornos completos (no granulares) se obtienen soluciones algo mejores que los que se obtienen mediante el algoritmo de búsqueda tabú granular, pero el tiempo requerido de computación es del mismo orden de magnitud que los que requieren los algoritmos XK o GHL.

Con esto se demuestra claramente que el uso de entornos granulares tiene un efecto positivo aplicado a métodos de búsqueda local. Por un lado, los entornos granulares se pueden introducir fácilmente en cualquier algoritmo de búsqueda local, y su implementación lleva a una importante reducción en el esfuerzo computacional. Por otro lado, la consideración de entornos granulares no afecta a la eficacia general del método, ya que la calidad de las soluciones

obtenidas mediante la búsqueda tabú granular es comparable a la de los mejores algoritmos planteados.

Además, la búsqueda tabú granular mantiene el comportamiento típico de los algoritmos de búsqueda tabú en los cuales al inicio hay una mejora rápida de la calidad de la solución con respecto a la solución inicial.

Debido a que el algoritmo de la búsqueda tabú granular puede ser un poco difícil de comprender de manera teórica, se procede en el siguiente capítulo a realizar un ejemplo numérico para explicar cómo funciona.

Capítulo 6

Aplicación de métodos al problema VRP

En este capítulo se propone un problema al que se le aplica la búsqueda tabú granular (capítulo 5) y el método de Clarke & Wright (capítulo 3). Gracias a la resolución manual del problema el lector será capaz de entender al detalle el funcionamiento de ambos métodos.

6.1. Búsqueda tabú granular

Para la ilustración de la búsqueda tabú granular se propone un problema para el que se elige al azar la solución inicial y se realizarán dos iteraciones del algoritmo para entender su funcionamiento. Se considera el problema con 15 clientes y 3 vehículos. La capacidad de cada vehículo es C = 1500 y la demanda de los clientes viene especificada por el siguiente vector $d = (d_i)_{i=1,\dots,15}$.

d=(230, 20, 376, 45, 750, 320, 120, 500, 210, 180, 965, 143, 67, 34, 247)

Se supone que el cliente i, con i = 1,…,15, se encuentra situado en el punto $p_i = (x_i, y_i)$ del plano. El coste de ir desde p_i a p_j viene dado por la distancia generada por la norma 1 en R^2, es decir:

$$c_{ij} = d_1(p_i, p_j) = |x_i - x_j| + |y_i - y_j|$$

Se ha optado por utilizar esta distancia y no la distancia Euclídea descrita en el capítulo 1 con el fin de simplificar los cálculos ya que las distancias resultarán ser números enteros.

Las posiciones de los clientes se muestran en la Figura 6.1.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	0	1	7	6	10	4	2	5	7	8	1	2	3	10	2	5
Y	0	1	3	0	7	9	0	8	6	9	4	9	2	2	7	5

Figura 6.1. Posición de los clientes.

Suponemos que el almacén se encuentra situado en el punto P_0= (0,0) y tiene una demanda d_0 = 0.

En la Figura 6.2. se muestra la posición de los clientes en el plano.

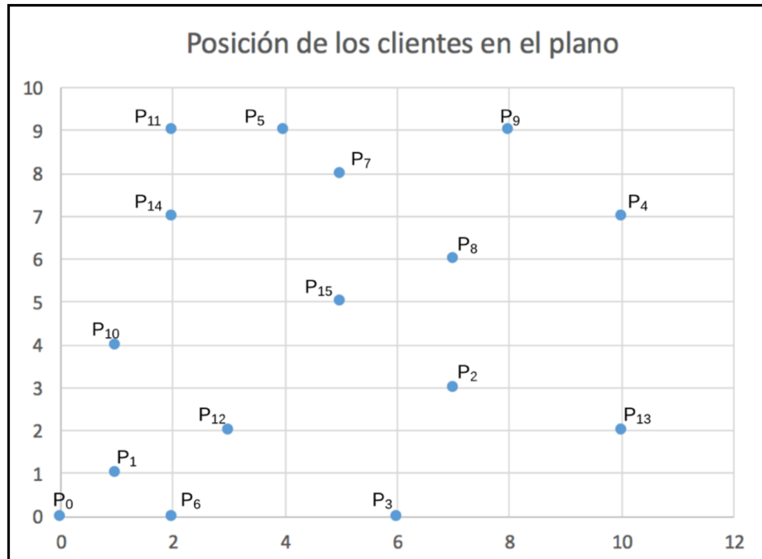


Figura 6.2. Posición de los clientes en el plano.

La matriz de costes calculada como la distancia generada por la norma 1 es la que se muestra en la Figura 6.3. Dicha matriz es simétrica porque el coste de ir de un vértice a otro no depende del sentido en el que se recorra.

Cij		DESTINO															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ORIGEN	0	0	2	10	6	17	13	2	13	13	17	5	11	5	12	9	10
	1	2	0	8	6	15	11	2	11	11	15	3	9	3	10	7	8
	2	10	8	0	4	7	9	8	7	3	7	7	11	5	4	9	4
	3	6	6	4	0	11	11	4	9	7	11	9	13	5	6	11	6
	4	17	15	7	11	0	8	15	6	4	4	12	10	12	5	8	7
	5	13	11	9	11	8	0	11	2	6	4	8	2	8	13	4	5
	6	2	2	8	4	15	11	0	11	11	15	5	9	3	10	7	8
	7	13	11	7	9	6	2	11	0	4	4	8	4	8	11	4	3
	8	13	11	3	7	4	6	11	4	0	4	8	8	8	7	6	3
	9	17	15	7	11	4	4	15	4	4	0	12	6	12	9	8	7
	10	5	3	7	9	12	8	5	8	8	12	0	6	4	11	4	5
	11	11	9	11	13	10	2	9	4	8	6	6	0	8	15	2	7
	12	5	3	5	5	12	8	3	8	8	12	4	8	0	7	6	5
	13	12	10	4	6	5	13	10	11	7	9	11	15	7	0	13	8
	14	9	7	9	11	8	4	7	4	6	8	4	2	6	13	0	5
	15	10	8	4	6	7	5	8	3	3	7	5	7	5	8	5	0

Figura 6.3. Matriz de costes.

Se parte de la siguiente solución inicial aleatoria:

Vehículo 1 : $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
Demanda satisfecha en la ruta (DSR) = 1421
Coste ruta = (2+8+4+11+8+13) = 46

Vehículo 2: $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$
DSR= 1330
Coste ruta = (2+11+4+4+12+5) = 38

Vehículo 3: $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_0$
DSR = 1456
Coste ruta = (11+8+7+13+5+10) = 54

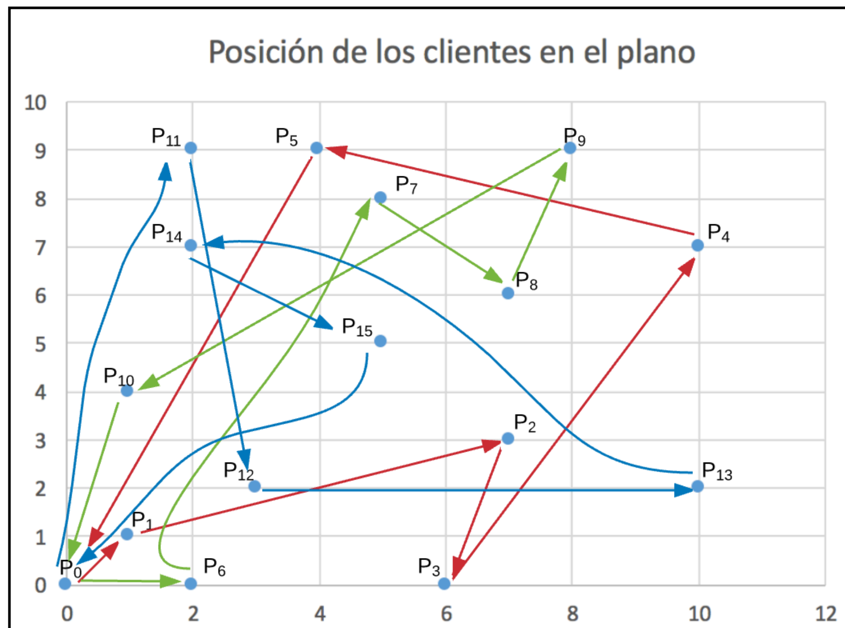


Figura 6.4. Rutas de la solución inicial.

Como se puede observar en la figura 6.4. esta solución inicial es factible, ya que se sirve la demanda de todos los clientes, los vehículos no exceden su capacidad y cada una de las rutas empieza y termina en el almacén. El coste total de esta solución es 138. Aún así, se puede apreciar que los vehículos siguen unas rutas un tanto "extrañas", por lo que la búsqueda tabú intentará reducir las distancias recorridas y con ello el coste asociado.

PRIMERA ITERACIÓN

Se comienza por definir el grafo disperso G' que contiene los arcos cortos y los arcos relevantes. Se consideran arcos relevantes aquellos que son incidentes en el almacén (punto P_0), y los que pertenecen a la mejor solución encontrada hasta el momento. Consideraremos en esta primera iteración, que la mejor solución encontrada hasta el momento es la propia solución inicial.

Calculamos el valor del umbral de granularidad ϑ. Utilizamos un valor de $\beta = 1$ como se propone en (Toth & Vigo, 2003). El número de clientes n es igual a 15, K es el número de vehículos que es $K = 3$. Por último, c' es el valor de una solución inicial que es $c' = 138$.

$$\vartheta = \beta \cdot \frac{c'}{(n+K)'}$$

Por tanto, sustituyendo los valores en la fórmula del umbral de granularidad se obtiene un valor de $\vartheta = 7{,}67$.

En la matriz de la Figura 6.5. se muestran coloreados los arcos que forman parte del grafo G'. Aparecen en color verde aquellos arcos “cortos”, cuyos costes se encuentran por debajo del umbral de granularidad. En color naranja se muestran los arcos incidentes en el almacén, es decir, los arcos que unen a los clientes con el punto P_0. En color azul se muestran los arcos pertenecientes a la mejor solución encontrada (solución inicial). Por último, en color blanco se muestran los arcos que no forman parte del grafo G', debido a que no cumplen ninguna de las condiciones anteriores.

Cij		DESTINO															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ORIGEN	0	0	2	10	6	17	13	2	13	13	17	5	11	5	12	9	10
	1	2	0	8	6	15	11	2	11	11	15	3	9	3	10	7	8
	2	10	8	0	4	7	9	8	7	3	7	7	11	5	4	9	4
	3	6	6	4	0	11	11	4	9	7	11	9	13	5	6	11	6
	4	17	15	7	11	0	8	15	6	4	4	12	10	12	5	8	7
	5	13	11	9	11	8	0	11	2	6	4	8	2	8	13	4	5
	6	2	2	8	4	15	11	0	11	11	15	5	9	3	10	7	8
	7	13	11	7	9	6	2	11	0	4	4	8	4	8	11	4	3
	8	13	11	3	7	4	6	11	4	0	4	8	8	8	7	6	3
	9	17	15	7	11	4	4	15	4	4	0	12	6	12	9	8	7
	10	5	3	7	9	12	8	5	8	8	12	0	6	4	11	4	5
	11	11	9	11	13	10	2	9	4	8	6	6	0	8	15	2	7
	12	5	3	5	5	12	8	3	8	8	12	4	8	0	7	6	5
	13	12	10	4	6	5	13	10	11	7	9	11	15	7	0	13	8
	14	9	7	9	11	8	4	7	4	6	8	4	2	6	13	0	5
	15	10	8	4	6	7	5	8	3	3	7	5	7	5	8	5	0

Figura 6.5. Arcos que forman parte del grafo disperso G’

Una vez construido el grafo disperso G', se procede a realizar los movimientos generados por los *k*-intercambios. La búsqueda tabú granular se lleva a cabo con entornos granulares múltiples, basados en los cuatro movimientos ya descritos en el apartado 5.3. Por tanto, se generan todas las posibles soluciones con k = 2, 3, 4 y se evalúa el coste de cada una de ellas para encontrar la mejor solución entre todas las soluciones generadas en cada iteración. En este ejemplo, por simplificar la exposición, se utilizará k = 2, es decir, intercambios de dos arcos. Al final del problema, se ilustrará un ejemplo de *k*-intercambios con k = 3, 4 para uno de los arcos a evaluar.

Gracias al método de la búsqueda tabú granular, el número de movimientos en cada iteración es mucho menor que en el método clásico de búsqueda tabú. Esto se debe a que el número de arcos considerados para generar movimientos es mucho menor en el grafo disperso, que en el grafo original. Por esto, el tiempo de computación es menor en la búsqueda tabú granular.

Es importante destacar que para los k-intercambios (con cualquier valor de k), los arcos presentes en la solución actual no generan un movimiento. Esto se debe a que los movimientos pretenden insertar el arco considerado, y si este arco se encuentra en la solución actual, el movimiento pierde sentido.

Además, al estudiar los *k*-intercambios con k = 2, entre los movimientos posibles se encuentran bastantes de ellos que no serán factibles, ya que las rutas no pasarán por el depósito o dejarán algún cliente aislado, es decir, no será visitado. Esto se da en los casos en los que los arcos unen clientes que se encuentran en la misma ruta. Por tanto, estos arcos no merece la pena evaluarlos en cuanto al coste total de la solución que se obtiene. Cabe señalar que, como se verá más adelante, para algunos intercambios con $k \neq 2$ es posible generar soluciones factibles a partir de arcos con clientes que se encuentran en la misma ruta.

Consideremos una ruta cualquiera como la que aparece en la Figura 6.6.

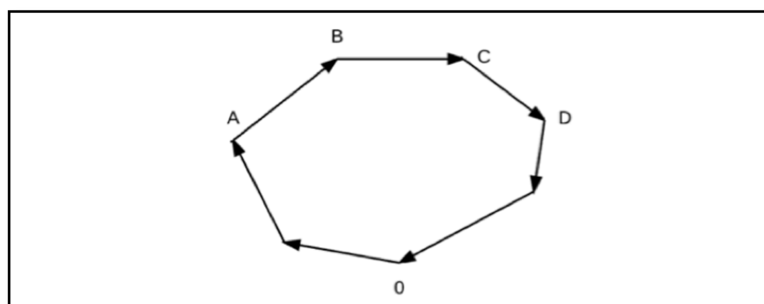


Figura 6.6. Ruta genérica tomada para las demostraciones.

Tomando el arco (A,D) como generador de un 2-intercambio, se llega a una solución, que como se señalaba anteriormente no es factible, ya que una ruta se desdobla en dos, lo que supone que necesitamos un vehículo a mayores y además, una de las rutas no queda conectada con el almacén. Esto se debe a que los clientes A y D se encuentran en la misma ruta. En consecuencia, cualquier movimiento que une a dos clientes cualesquiera que se encuentren en una misma ruta, llevará a una solución no factible. Este caso se muestra en la Figura 6.7.

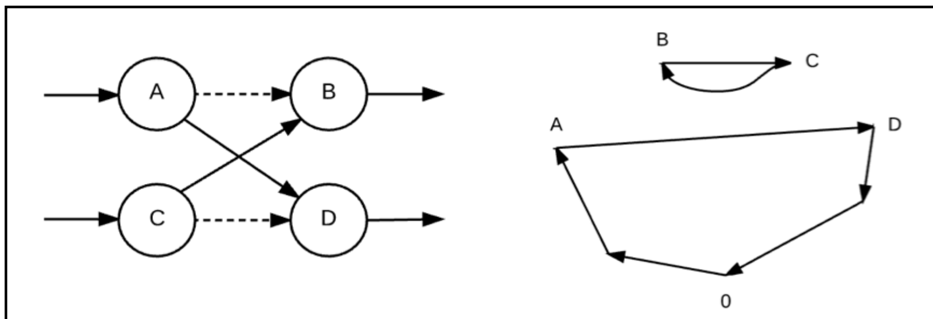


Figura 6.7. Ejemplo movimiento con clientes en la misma ruta.

Un caso particular de un movimiento generado por intercambio de clientes que se encuentran en la misma ruta, es el que se produce cuando, el arco conecta a un cliente X y a σ_{σ_X}, es decir, al segundo cliente que sigue a este cliente X. La solución obtenida con este tipo de movimientos no es factible, ya que deja a un cliente aislado. Si se toma el arco (A,C) del ejemplo anterior, se puede ver en la Figura 6.8. que el cliente B no es visitado por ningún vehículo. En caso de que el arco sea (0, σ_{σ_0}), esta situación se puede evitar considerando el almacén dentro de otra ruta tal y como se demostrará más adelante.

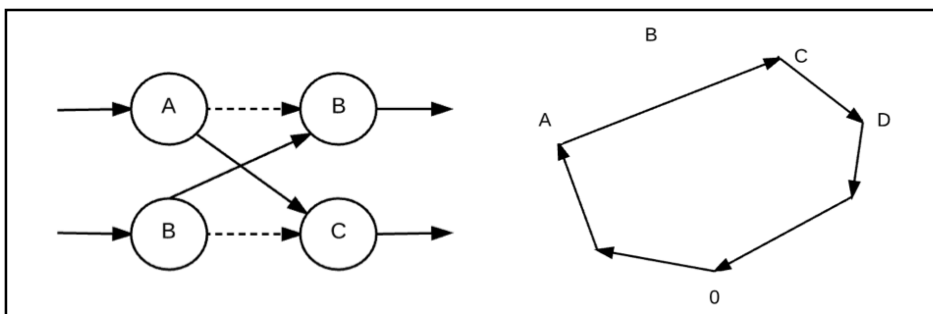


Figura 6.8. Ejemplo movimiento con clientes en la misma ruta con arco (X, σ_{X_X}).

Un caso especial a tener en cuenta, son los movimientos generados a partir de los arcos incidentes en el depósito. Si tomamos como ejemplo el arco (0,2), nos daremos cuenta de que se debe considerar el punto P_0 como un punto que

pertenece a una ruta que no sea la que incluye al cliente situado en P_2. Como ya se ha mencionado, si se consideran puntos que se encuentran en una misma ruta se generan movimientos que llevan a soluciones no factibles. A continuación se muestra para el arco (0,2) los tres posibles movimientos con un 2-intercambio.

Posibilidad 1: considerando el punto P_0 dentro de la ruta $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$. En la Figura 6.9. se muestra el intercambio y se puede observar que la solución generada no es factible porque el punto P_1 queda aislado, es decir, no es visitado por ningún vehículo para satisfacer su demanda.

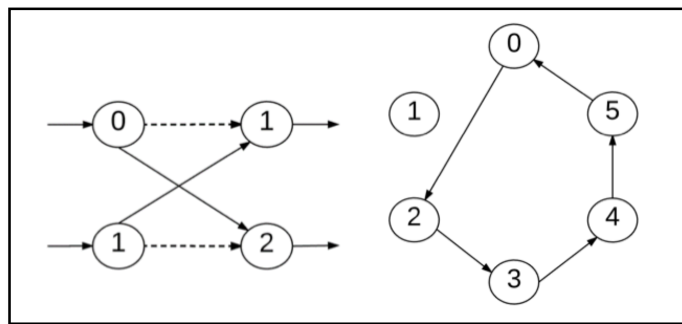


Figura 6.9. 2-Intercambio con arco (0,2). Misma ruta.

Posibilidad 2: considerando el punto P_0 dentro de la ruta $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$. Como se observa en la Figura 6.10. este movimiento genera una solución que es factible en cuanto a que a todos los clientes son visitados una sola vez y todas las rutas pasan por el almacén (P_0). Se procede por tanto a evaluar dicha solución.

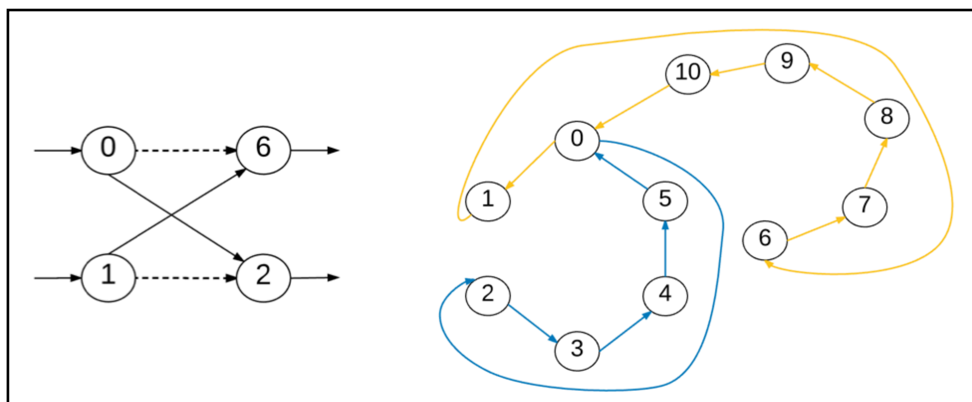


Figura 6.10. 2-Intercambio con arco (0,2). Distinta ruta (a)

Ruta 1: $P_0 - P_1 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$
DSR = 1560 Coste=(2+2+11+4+4+12+5)+(1560-1500)x100=6040
Ruta 2: $P_0 - P_2 - P_3 - P_4 - P_5 - P_0$
DSR = 1191 Coste = (10+4+11+8+13) = 46
Ruta 3: $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_0$
DSR = 1456 Coste = (11+8+7+13+5+10) = 54
Coste total de solución = 6140

Esta solución no es factible respecto a la restricción de capacidad de los vehículos. En este caso, se penaliza el coste de la solución multiplicando el exceso de capacidad por un parámetro de penalización α_C = 100, ya que es el valor inicial recomendado en (Toth & Vigo, 2003) para el parámetro de penalización.

Posibilidad 3: considerando el punto P_0 dentro de la ruta $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_0$. Se ilustra gráficamente en la Figura 6.11.

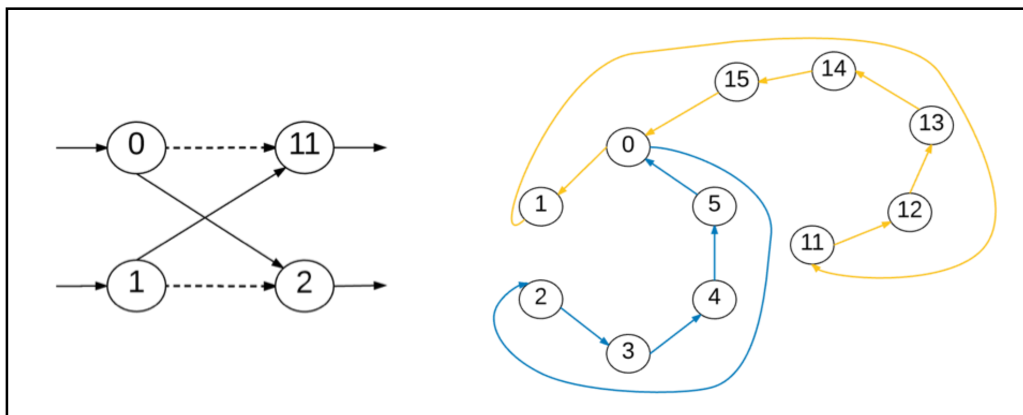


Figura 6.11. 2-Intercambio con arco (0,2). Distinta ruta (b)

Ruta 1: $P_0 - P_1 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_0$
DSR = 1686 Coste=(2+9+8+7+13+5+10)+(1686-1500)x100=18654
Ruta 2: $P_0 - P_2 - P_3 - P_4 - P_5 - P_0$
DSR = 1191 Coste = (10+4+11+8+13) = 46
Ruta 3: $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$
DSR = 1330 Coste = (2+11+4+4+12+5) = 38
Coste total de solución = 18738

De nuevo, la solución no es factible debido al exceso de carga del vehículo que circula por la ruta 1.

A continuación se muestran las soluciones obtenidas aunque para simplificar la exposición del problema solo se describirán y evaluarán las soluciones factibles.

- Arco (2,7). Coste total = 134

Ruta 1: P_0 – P_1 – P_2 – P_7 – P_8 – P_9 – P_{10} – P_0
DSR = 1260 Coste = (2+8+7+4+4+12+5) = 42
Ruta 2: P_0 – P_6 – P_3 – P_4 – 9_5 –P_0
DSR = 1491 Coste = (2+4+11+8+13) = 38
Ruta 3: P_0 – P_{11} – P_{12} – P_{13} - P_{14} – P_{15} – P_0
DSR = 1456 Coste = (11+8+7+13+5+10) = 54

- Arco (6,3). Coste total = 134

Ruta 1: P_0 – P_6 – P_3 – P_4 – P_5 – P_0
DSR = 1491 Coste = (2+4+11+8+13) = 38
Ruta 2: P_0 – P_1 – P_2 – P_7 – P_8 – P_9 – P_{10} – P_0
DSR = 1260 Coste = (2+8+7+4+4+12+5) = 42
Ruta 3: P_0 – P_{11} – P_{12} – P_{13} – P_{14} – P_{15} – P_0
DSR = 1456 Coste = (11+8+7+13+5+10) = 54

- Arco (8,13). Coste total = 146

Ruta 1: P_0 – P_1 – P_2 – P_3 – P_4 – P_5 – P_0
DSR = 1421 Coste = (2+8+4+11+8+13) = 46
Ruta 2: P_0 – P_6 – P_7 – P_8 – 9_{13} – P_{14} – P_{15} – P_0
DSR = 1288 Coste = (2+11+4+7+13+5+10) = 52
Ruta 3: P_0 – P_{11} – P_{12} – P_9 – P_{10} – P_0
DSR = 1498 Coste = (11+8+12+12+5) = 48

- Arco (9,15). Coste total = 132

Ruta 1: P_0 – P_1 – P_2 – P_3 – P_4 – P_5 – P_0
DSR = 1421 Coste = (2+8+4+11+8+13) = 46
Ruta 2: P_0 – P_6 – P_7 – P_8 – 9_9 – P_{15} – P_0
DSR = 1397 Coste = (2+11+4+4+7+10) = 38
Ruta 3: P_0 – P_{11} – P_{12} – P_{13} – P_{14} – P_{10}– P_0
DSR = 1389 Coste = (11+8+7+13+4+5) = 48

- Arco (11,9). Coste total = 140

Ruta 1: $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
DSR = 1421 Coste = (2+8+4+11+8+13) = 46
Ruta 2: $P_0 - P_6 - P_7 - P_8 - 9_{12} - P_{13} - P_{14} - P_{15} - P_0$
DSR = 1431 Coste = (2+11+4+8+7+13+5+10) = 60
Ruta 3: $P_0 - P_{11} - P_9 - P_{10} - P_0$
DSR = 1355 Coste = (11+6+12+5) = 34

- Arco (12,10). Coste total = 132

Ruta 1: $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
DSR = 1421 Coste = (2+8+4+11+8+13) = 46
Ruta 2: $P_0 - P_6 - P_7 - P_8 - 9_9 - P_{13} - P_{14} - P_{15} - P_0$
DSR = 1498 Coste = (2+11+4+4+9+13+5+10) = 58
Ruta 3: $P_0 - P_{11} - P_{12} - P_{10} - P_0$
DSR = 1288 Coste = (11+8+4+5) = 28

- Arco (14,10). Coste total = 132

Ruta 1: $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
DSR = 1421 Coste = (2+8+4+11+8+13) = 46
Ruta 2: $P_0 - P_6 - P_7 - P_8 - 9_9 - P_{15} - P_0$
DSR = 1397 Coste = (2+11+4+4+7+10) = 38
Ruta 3: $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{10} - P_0$
DSR = 1389 Coste = (11+8+7+13+4+5) = 48

Hasta ahora se han mostrado los movimientos generados a partir de los arcos pertenecientes al grafo disperso G'. En esta iteración se ha conseguido una solución que mejora a la mejor solución encontrada hasta el momento (que coincide con la solución inicial). Por tanto, el mejor coste encontrado hasta el momento era de 138, y se ve mejorado con la solución encontrada en esta iteración, ya que tiene un coste de 132. En esta iteración se han encontrado varias soluciones que tienen coste 132, por lo que se elige al azar una de ellas para continuar con la segunda iteración.

Por tanto, consideremos la mejor solución encontrada hasta ahora con un coste de 132, la compuesta por las siguientes rutas:

Ruta 1: $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
Ruta 2: $P_0 - P_6 - P_7 - P_8 - P_9 - P_{15} - P_0$
Ruta 3: $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{10} - P_0$

En las posteriores iteraciones hay que tener en cuenta qué movimientos son tabú y cuáles no. Se considerarán movimientos tabú, aquellos que intenten

añadir arcos que se han eliminado en los movimientos anteriores. La permanencia en la lista tabú t para cada movimiento, es una variable entera aleatoria y uniformemente distribuida en el intervalo (t_{min}, $t_{máx}$). En este problema se utiliza t_{min}=5 y $t_{máx}$=10 según recomiendan (Toth & Vigo, 2003).

En este momento, se debe hacer una revisión de cuáles han sido los arcos eliminados en el movimiento realizado para llegar a la solución actual. Los arcos eliminados se almacenan en la siguiente lista tabú, con su permanencia en dicha lista tabú:

- Arco (9,10) t = 7
- Arco (14,15) t = 7

La permanencia en la lista tabú determina que estos arcos no pueden ser reinsertados hasta que no hayan trascurrido 7 iteraciones.

SEGUNDA ITERACIÓN

En esta segunda iteración el grafo disperso G' no se reconstruye de la nada, ya que esto se realiza cada $2n$ iteraciones, donde n es el número de clientes. Es decir, el grafo G' no se modifica hasta que no se hayan realizado 2 x 15 = 30 iteraciones. Pero en esta iteración G' sí debe ser ligeramente modificado para añadir los arcos de la mejor solución encontrada hasta el momento que, de nuevo, coincide con la solución actual.

Por tanto, habría que actualizar el grafo disperso añadiendo los arcos insertados en la iteración anterior, es decir, los arcos (9,15) y (14,10). Debido a que estos arcos ya formaban parte del grafo disperso, dicho grafo se mantiene igual que en la primera iteración.

A continuación se detallan los movimientos factibles, con el coste asociado:

- Arco (6,3). Coste total = 132

Ruta 1: $P_0 – P_1 – P_2 – P_7 – P_8 – P_9 – P_{15} – P_0$
DSR = 1327 Coste = (2+8+7+4+4+7+10) = 42
Ruta 2: $P_0 – P_6 – P_3 – P_4 – 9_5 – P_0$
DSR = 1491 Coste = (2+4+11+8+13) = 42
Ruta 3: $P_0 – P_{11} – P_{12} – P_{13} – P_{14} – P_{10} – P_0$
DSR = 1389 Coste = (11+8+7+13+4+5) = 48

- Arco (9,10). Coste total = 138

Ruta 1 : $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
DSR = 1421 Coste = (2+8+4+11+8+13) = 46
Ruta 2: $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$
DSR= 1330 Coste = (2+11+4+4+12+5) = 38
Ruta 3: $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_0$
DSR = 1456 Coste = (11+8+7+13+5+10) = 54

- Arco (11,9). Coste total = 134

Ruta 1 : $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
DSR = 1421 Coste = (2+8+4+11+8+13) = 46
Ruta 2: $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$
DSR= 1364 Coste = (2+11+4+8+7+13+4+5) = 54
Ruta 3: $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_0$
DSR = 1422 Coste = (11+6+7+10) = 34

- Arco (12,15). Coste total = 132

Ruta 1 : $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
DSR = 1421 Coste = (2+8+4+11+8+13) = 46
Ruta 2: $P_0 - P_6 - P_7 - P_8 - P_9 - P_{13} - P_{14} - P_{10} - P_0$
DSR= 1431 Coste = (2+11+4+4+9+13+4+5) = 52
Ruta 3: $P_0 - P_{11} - P_{12} - P_{15} - P_0$
DSR = 1355 Coste = (11+8+5+10) = 34

- Arco (14,15). Coste total = 138

Ruta 1 : $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
DSR = 1421 Coste = (2+8+4+11+8+13) = 46
Ruta 2: $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$
DSR= 1330 Coste = (2+11+4+4+12+5) = 38
Ruta 3: $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_0$
DSR = 1456 Coste = (11+8+7+13+5+10) = 54

De estos movimientos, cabe destacar que las soluciones generadas por los arcos (9,10) y (14,15) son movimientos tabú, ya que insertarían los arcos eliminados en la anterior iteración. El método descrito en (Toth & Vigo, 2003) no considera ningún criterio de aspiración por lo que cualquier movimiento tabú se descarta directamente.

De nuevo, en esta iteración, encontramos dos soluciones con el menor coste. Al igual que antes, se elige una de ellas al azar para continuar iterando. En concreto, el coste es 132, por lo que la mejor solución no se ve mejorada.

Por tanto, la solución actual está formada por las siguientes rutas:

Ruta 1 : $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
Ruta 2: $P_0 - P_6 - P_7 - P_8 - P_9 - P_{13} - P_{14} - P_{10} - P_0$
Ruta 3: $P_0 - P_{11} - P_{12} - P_{15} - P_0$

Ahora es momento de actualizar la lista tabú, de manera que la permanencia de los arcos añadidos en la iteración anterior se reduce en una unidad. Se añaden también los arcos eliminados en la presente iteración, y se les acompaña de la permanencia en la lista tabú generada de manera aleatoria.

- Arco (9,10) $t = 6$
- Arco (14,15) $t = 6$
- Arco (12,13) $t = 9$
- Arco (9,15) $t = 9$

Para las iteraciones siguientes se procedería de la misma manera, actualizando el grafo disperso y evaluando los distintos movimientos. Tras realizar un número elevado de iteraciones se espera que el coste total se reduzca y se obtenga una solución de una calidad razonable.

EJEMPLO DE MOVIMIENTOS CON k = 3, 4

En primer lugar se procede a ilustrar para los movimientos del tipo 3-Intercambio y 4-Intercambio la generalización de los casos en los que dichos movimientos no son factibles.

k-Intercambio con k = 3. Opción(a)

En este tipo de intercambio, a diferencia de lo que ocurre en los 2-intercambios, es posible generar soluciones factibles con arcos que incluyen clientes que se encuentran en la misma ruta. Supongamos una ruta cualquiera como la que aparece en la Figura 6.12.

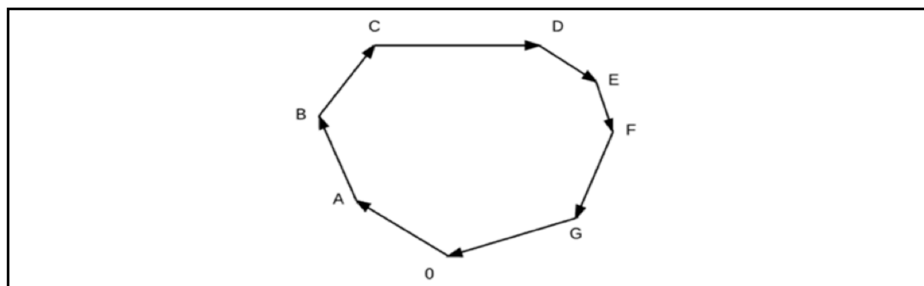


Figura 6.12. Ruta genérica

Si tomamos un arco cualquiera generador de un movimiento, por ejemplo el arco (C,G), se puede observar en la Figura 6.13. que la solución obtenida es factible desde el punto de vista de la forma de las rutas. Nótese que la ruta resultante tiene los mismos clientes pero se visitan en distinto orden, lo que hará que varíe el coste. Este caso particular visto en el ejemplo del arco (C,G) se puede extender para un par cualquiera de clientes que se encuentren dentro de la misma ruta. Además, esta misma situación se da cuando uno de los puntos que forman parte del arco generador del movimiento es el almacén.

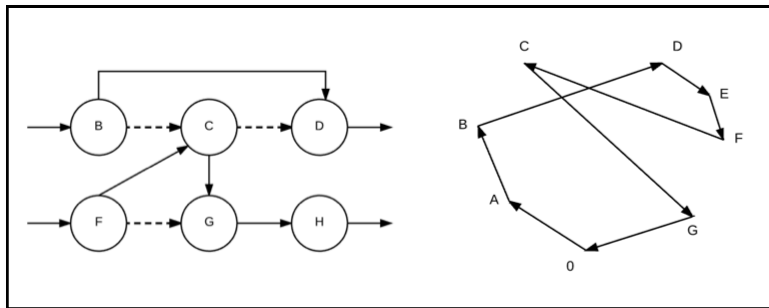


Figura 6.13. Movimiento arco (C,G). Clientes en la misma ruta.

En el caso de que se considere el almacén (P_0) en la misma ruta que el cliente que forma parte del arco, el movimiento generará una solución factible. Esto se debe a que es una particularización del caso general de puntos en la misma ruta, ya ilustrado en la Figura 6.13.

En cambio, los arcos del tipo (0,X) no generan soluciones factibles si se considera el punto P_0 (almacén) en distinta ruta que el cliente situado en el punto P_X. Veamos un ejemplo en la Figura 6.14. tomando, por ejemplo, el arco (0,D), donde se puede observar que una de las rutas no queda unida con el punto 0 (almacén), y por lo tanto el movimiento no es factible.

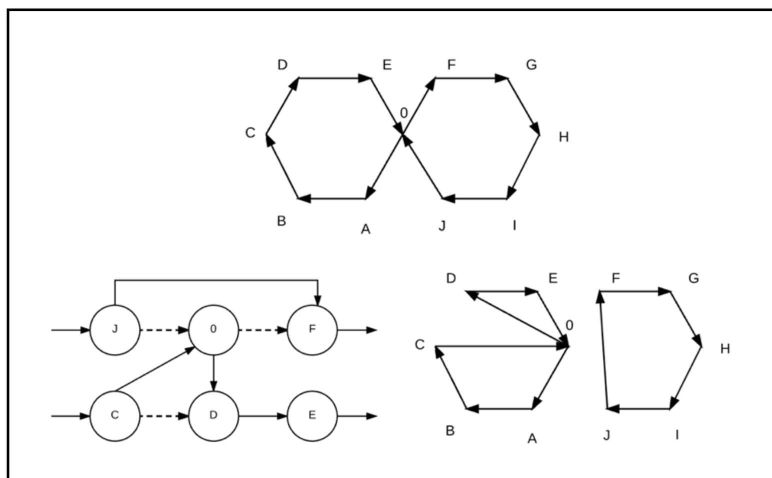


Figura 6.14. Movimiento arco (0,D). Almacén en distinta ruta.

k-Intercambios con k = 3. Opción(b)

Al igual que en la opción (a) del 3-intercambio, con este tipo de movimiento es posible generar soluciones factibles partiendo de un arco (a, b) con los clientes a y b situados en la misma ruta, salvo algunas excepciones.

En la opción (b) del 3-intercambio es posible generar soluciones factibles con los arcos (0,X) tanto si se considera el punto P_0 (almacén) dentro de la misma ruta que el cliente X, como si se considera en distinta ruta, excepto en algunos casos concretos que se muestran a continuación.

- Arco (0,X) con los nodos 0 y X en distinta ruta: la solución obtenida no es factible si el cliente X es el último cliente visitado en su ruta. Este caso se puede observar en la Figura 6.15. aplicado al arco (0,J) donde J es el último cliente visitado en su ruta. Esto mismo ocurre para los arcos (X,Y) si el cliente Y es el último visitado en su ruta. La razón por la que esto ocurre es que en el 3-Intercambio, como los clientes b y σ_b cambian de ruta, si uno de ellos es el almacén la ruta de la que se quita queda sin almacén.

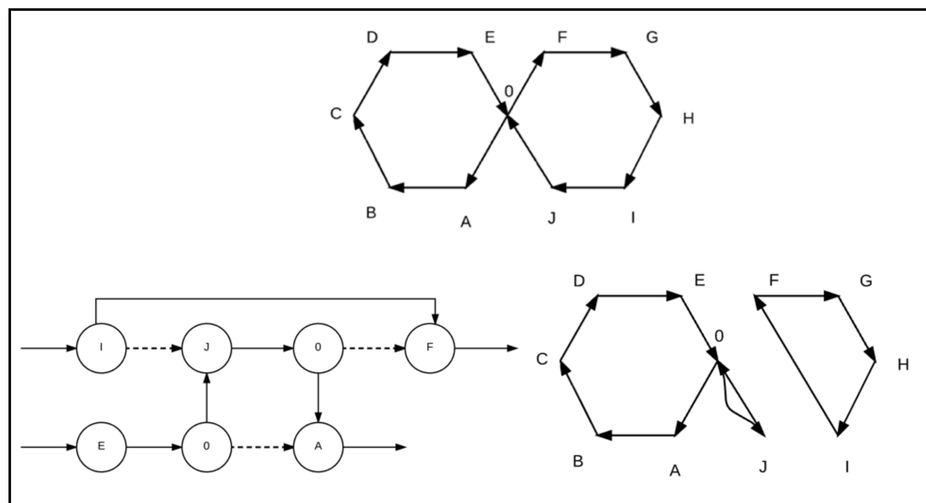


Figura 6.15. Movimiento arco (0,J). Almacén en distinta ruta.

- Arco (0,X) con los nodos 0 y X en la misma ruta: las soluciones generadas no son factibles si el cliente X es el último visitado en su ruta, al igual que ocurre con el arco (0,X) si se consideran al almacén y el cliente X en distinta ruta tal y como se acaba de ilustrar.

k-Intercambios con k = 4

Con este tipo de movimiento es posible obtener soluciones factibles a partir del arco (0,X) si el cliente X se considera en distinta ruta que el almacén. Por el contrario, si el cliente X se considera en la misma ruta que el almacén, el movimiento no será factible si el cliente X es el segundo cliente visitado dentro de su ruta. Este caso se ilustra en la Figura 6.16. para el arco (0,B) del ejemplo genérico ya utilizado anteriormente.

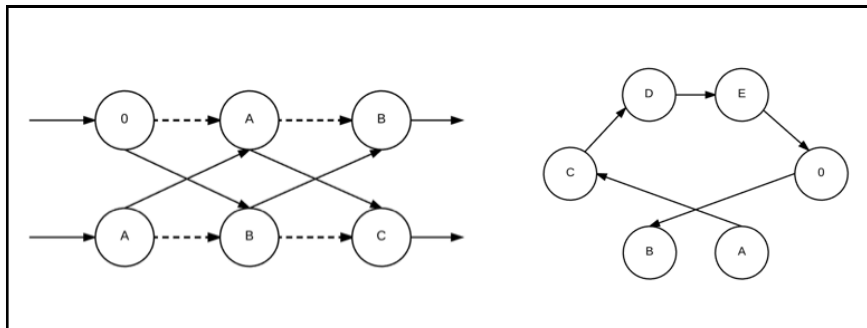


Figura 6.16. Movimiento arco (0,B). Almacén en misma ruta.

Además para un arco cualquiera (X,Y) si X es el último cliente visitado en su ruta, el 4-intercambio no es factible para ningún cliente Y. Este caso se muestra en la figura 6.17. donde se observa que una ruta no pasa por el almacén. Lo mismo ocurre si Y es 0, es decir, si el cliente b es el almacén. Estos casos particulares en los que las soluciones no son factibles cuando se considera un arco (a, b) con los clientes a y b en distintas rutas, se han tenido en cuenta en la implementación de la búsqueda tabú granular descrita en el Capítulo 7.

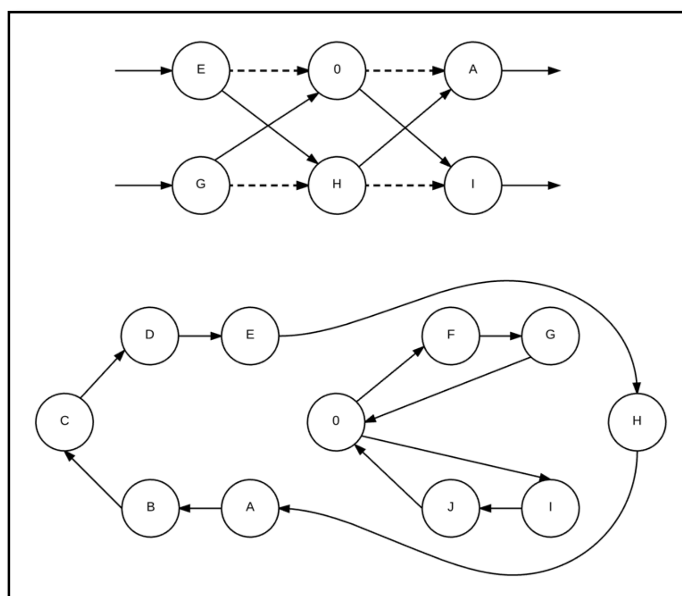


Figura 6.17. Movimiento arco (0,H)

A continuación, se ilustra la generación de movimientos mediante *k*-Intercambios con k = 3, 4 para el arco (0,2). En concreto, se llevará a cabo considerando el punto P_0 dentro de la ruta $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$ y dentro de la ruta $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$. Si se considerase dentro de la ruta $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_0$, se procedería de manera análoga a como se realiza para la ruta $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$. No se evaluarán los costes de las soluciones, pero sí se muestran las distintas rutas que se formarían para entender el funcionamiento de este tipo de intercambios.

k-Intercambios con k = 3. Opción (a)

Posibilidad 1: considerando el punto P_0 dentro de la ruta $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$. Como se puede ver en la Figura 6.18. la solución generada mediante este intercambio de arcos es factible, debido a que todos los clientes son visitados una vez, la ruta pasa por el almacén y además, no se excede la capacidad de los vehículos. Respecto a la solución inicial, esta solución solo modifica el orden en el que los clientes son visitados en la ruta 1, por lo que los costes serán diferentes.

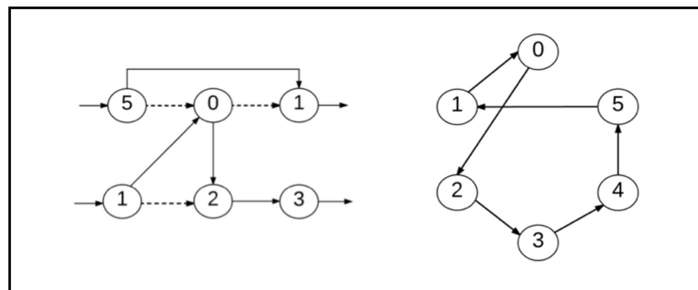


Figura 6.18. 3-Intercambio (a) arco (0,2). Misma ruta.

Posibilidad 2: considerando el punto P_0 dentro de la ruta $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$. En este caso, tal y como se muestra en la Figura 6.19. la solución generada no es factible, debido a que los clientes situados en los puntos P_6, P_7, P_8, P_9, P_{10} no están unidos con el almacén.

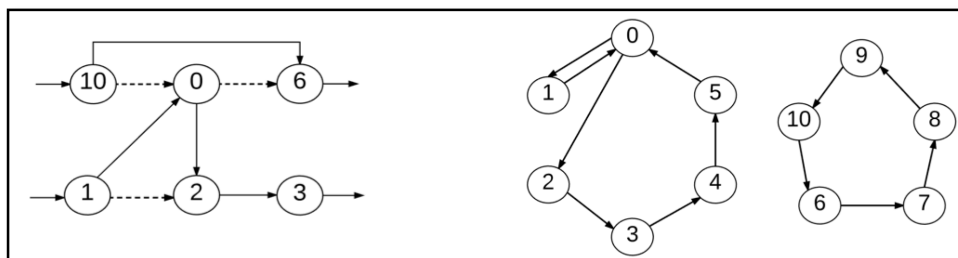


Figura 6.19. 3-Intercambio (a) arco (0,2). Distinta ruta.

k-Intercambio con k = 3. Opción (b)

Posibilidad 1: considerando el punto P_0 dentro de la ruta $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$. La solución obtenida ilustrada en la Figura 6.20. es factible en cuanto a la forma de las rutas ya que solo se ha visto alterado el orden en el que se visita a los clientes dentro de la ruta 1.

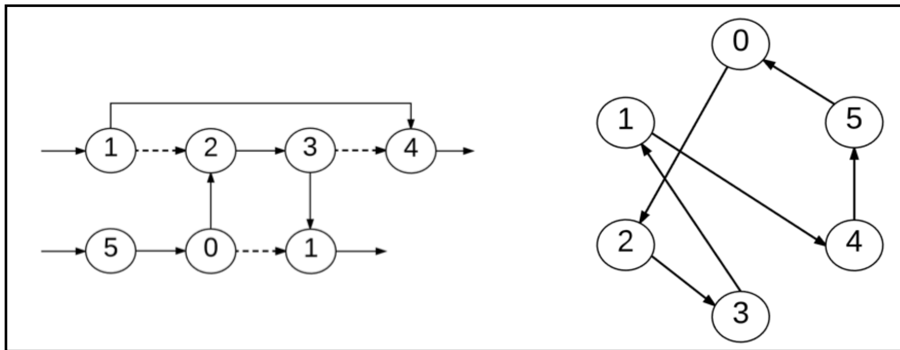


Figura 6.20. 3- Intercambio (b) arco (0,2). Misma ruta.

Posibilidad 2: considerando el punto P_0 dentro de la ruta $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$. De nuevo, la solución obtenida mostrada en la Figura 6.21. es factible en cuanto a la forma que toman las rutas, pero seguramente, si se evaluase, se llegaría a la conclusión de que no cumple la restricción de carga máxima de los vehículos, debido a que la ruta $P_0 - P_2 - P_3 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$, está muy saturada.

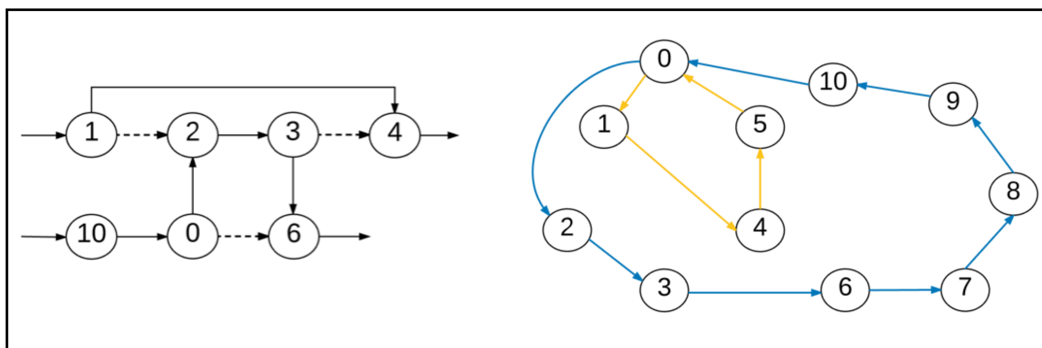


Figura 6.21. 3-Intercambio (b) arco (0,2). Distinta ruta.

k-Intercambio con k = 4

Posibilidad 1: considerando el punto P_0 dentro de la ruta $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$. Para este movimiento, la solución obtenida, no es factible, tal y como se puede ver en la Figura 6.22. La ruta pasa por el almacén, pero por el contrario no queda cerrada ya que del cliente situado en el punto P_1 no hay ningún arco entrante y de la misma manera para el cliente situado en el punto P_2 no hay ningún arco saliente.

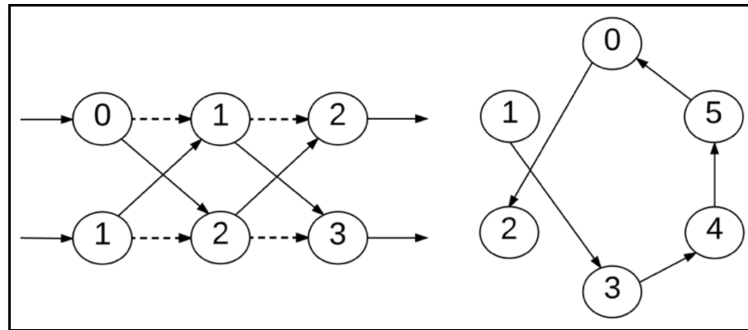


Figura 6.22. 4-Intercambio arco (0,2). Misma ruta.

Posibilidad 2: considerando el punto P_0 dentro de la ruta $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$. Para este movimiento se obtiene una solución formada por rutas cerradas. En concreto el cambio ha consistido en el intercambio de los clientes 2 y 6 entre la ruta 1 y la ruta 2, tal y como puede observarse en la Figura 6.23. Por tanto esta ruta es factible en cuanto a la forma que toman las rutas.

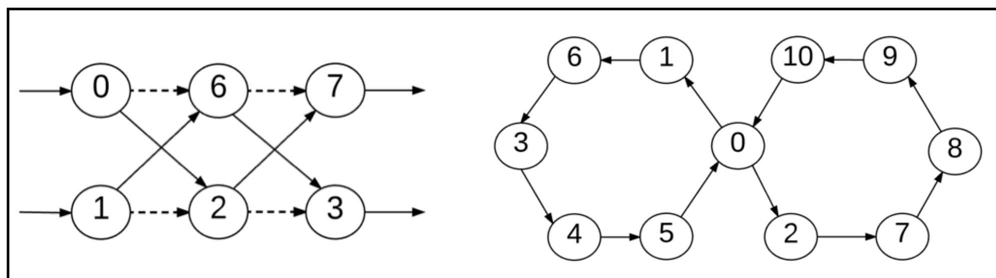


Figura 6.23. 4-Intercambio arco (0,2). Distinta ruta.

6.2. Método de Clarke & Wright

En este apartado, se ilustrará mediante un ejemplo el funcionamiento del método de Clarke & Wright descrito en el apartado 3.1.2. Para ello, se tomarán como datos los mismos que se utilizaron en el problema para la exposición del método de la búsqueda tabú granular aplicada al VRP. Recordemos los datos más importantes:

Se tienen 15 clientes y 3 vehículos. La capacidad de cada vehículo es C = 1500 y la demanda de los clientes viene especificada por el siguiente vector $d = (d_i)_{i=1,\ldots,15}$.

d=(230, 20, 376, 45, 750, 320, 120, 500, 210, 180, 965, 143, 67, 34, 247)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X	0	1	7	6	10	4	2	5	7	8	1	2	3	10	2	5
Y	0	1	3	0	7	9	0	8	6	9	4	9	2	2	7	5

Figura 6.24. Posición de los clientes

Las posiciones de los clientes se muestran en la Figura 6.24. y se calculaban las distancias entre ellos como la norma 1 en R^2. Dichas distancias se almacenan en una media matriz, que aparece en la Figura 6.25, ya que el problema es simétrico.

Cij	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2														
2	10	8													
3	6	6	4												
4	17	15	7	11											
5	13	11	9	11	8										
6	2	2	8	4	15	11									
7	13	11	7	9	6	2	11								
8	13	11	3	7	4	6	11	4							
9	17	15	7	11	4	4	15	4	4						
10	5	3	7	9	12	8	5	8	8	12					
11	11	9	11	13	10	2	9	4	8	6	6				
12	5	3	5	5	12	8	3	8	8	12	4	8			
13	12	10	4	6	5	13	10	11	7	9	11	15	7		
14	9	7	9	11	8	4	7	4	6	8	4	2	6	13	
15	10	8	4	6	7	5	8	3	3	7	5	7	5	8	5

Figura 6.25. Matriz de distancias

A continuación, se calculan los ahorros para cada par de clientes (sin tener en cuenta el almacén) y se incluyen en la media matriz de costes. Así mismo, se añade a la media matriz la condición inicial de que cada cliente es visitado por un solo vehículo, es decir, para la columna del almacén (P_0) el valor de $t_{y,z}$ = 2, tal y como se muestra en la Figura 6.26. Para que esto sea posible, inicialmente se supone que se dispone de un número ilimitado de vehículos.

Q	P0															
230	2 2	P1														
20	2 10	4 8	P2													
376	2 6	2 6	12 4	P3												
45	2 17	4 15	20 7	12 11	P4											
750	2 13	4 11	14 9	8 11	22 8	P5										
320	2 2	2 2	4 8	4 4	4 15	4 11	P6									
120	2 13	4 11	16 7	10 9	24 6	24 2	4 11	P7								
500	2 13	4 11	20 3	12 7	26 4	20 6	4 11	22 4	P8							
210	2 17	4 15	20 7	12 11	30 4	26 4	4 15	26 4	26 4	P9						
180	2 5	4 3	8 7	2 9	10 12	10 8	2 5	10 8	10 8	10 12	P10					
965	2 11	4 9	10 11	4 13	18 10	22 2	4 9	20 4	16 8	22 6	10 6	P11				
143	2 5	4 3	10 5	6 5	10 12	10 8	4 3	10 8	10 8	10 12	6 4	8 8	P12			
67	2 12	4 10	18 4	12 6	24 5	12 13	4 10	14 11	18 7	20 9	6 11	8 15	10 7	P13		
34	2 9	4 7	10 9	4 11	18 8	18 4	4 7	18 4	16 6	18 8	10 4	18 2	8 6	8 13	P14	
247	2 10	4 8	16 4	10 6	20 7	18 5	4 8	20 3	20 3	20 7	10 5	14 7	10 5	14 8	14 5	P15

Figura 6.26. Matriz para desarrollar el método de Clarke & Wright

El procedimiento comienza por localizar la celda (o celdas) con un mayor ahorro. En caso de haber varias celdas con el mismo ahorro, se elige al azar una de ellas. En este ejemplo, por facilitar el procedimiento, en caso de empate de ahorros se ha elegido la celda recorriendo las filas de arriba abajo y de izquierda a derecha.

El siguiente paso consiste en evaluar si es posible unir los clientes asociados a dicha celda, es decir, si la suma de las demandas de los clientes a unir no

excede la capacidad del vehículo. Además, se debe comprobar que los valores $t_{y,z}$ sean mayores que uno. En la primera iteración esto siempre se cumple, ya que se establece al comienzo del procedimiento que cada cliente es visitado por un vehículo y por tanto $t_{y,z}$ = 2. Por último, se debe verificar que los clientes que se están considerando en la unión no formen parte de una misma ruta, ya que no tendría sentido unirles de nuevo.

En caso de cumplir las condiciones anteriores, los clientes se pueden enlazar, de forma que se coloca en la celda correspondiente el valor de $t_{y,z}$= 1 y se actualizan los valores de $t_{y,0}$ y $t_{z,0}$, reduciendo cada uno de ellos su valor en una unidad.

Q	P0															
230	2 2	P1														
20	2 10	4 8	P2													
376	2 6	2 6	12 4	P3												
255	1 17	4 15	20 7	12 11	P4											
750	2 13	4 11	14 9	8 11	22 8	P5										
320	2 2	2 2	4 8	4 4	4 15	4 11	P6									
120	2 13	4 11	16 7	10 9	24 6	24 2	4 11	P7								
500	2 13	4 11	20 3	12 7	26 4	20 6	4 11	22 4	P8							
	1 17	4 15	20 7	12 11	1 30 4	26 4	4 15	26 4	26 4	P9						
180	2 5	4 3	8 7	2 9	10 12	10 8	2 5	10 8	10 8	10 12	P10					
965	2 11	4 9	10 11	4 13	18 10	22 2	4 9	20 4	16 8	22 6	10 6	P11				
143	2 5	4 3	10 5	6 5	10 12	10 8	4 3	10 8	10 8	10 12	6 4	8 8	P12			
67	2 12	4 10	18 4	12 6	24 5	12 13	4 10	14 11	18 7	20 9	6 11	8 15	10 7	P13		
34	2 9	4 7	10 9	4 11	18 8	18 4	4 7	18 4	16 6	18 8	10 4	18 2	8 6	8 13	P14	
247	2 10	4 8	16 4	10 6	20 7	18 5	4 8	20 3	20 3	20 7	10 5	14 7	10 5	14 8	14 5	P15

Figura 6.27. Primera iteración

En la Figura 6.27. se muestra la primera iteración. La celda con la que se consigue un mayor ahorro (30), es la que une a los clientes P_4 y P_9. Observamos que d_4 + d_9 = 45 + 210 = 255, no excede la capacidad C = 1500 de los vehículos. Además, los clientes situados en P_4 y P_9 no forman parte de la misma

ruta (de hecho, no hay rutas formadas en esta primera iteración). La última condición a evaluar es que los valores de $t_{4,0}$ y $t_{9,0}$, sean ambos mayores que 0, y en este caso si lo eran. Por tanto, la unión es posible entre P_4 y P_9, se modifica el valor de $t_{4,9}$ = 1, y se actualizan los valores de $t_{4,0}$ y $t_{9,0}$, reduciéndolos en una unidad. También se deben modificar los valores de la columna correspondiente a la demanda, colocando el valor de la demanda total de P_4 y P_9 a uno de los dos clientes.

Q	P0															
230	2 2	P1														
20	2 10	4 8	P2													
376	2 6	2 6	12 4	P3												
	0 17	4 15	20 7	12 11	P4											
750	2 13	4 11	14 9	8 11	22 8	P5										
320	2 2	2 2	4 8	4 4	4 15	4 11	P6									
120	2 13	4 11	16 7	10 9	24 6	24 2	4 11	P7								
755	1 13	4 11	20 3	12 7	1 26 4	20 6	4 11	22 4	P8							
	1 17	4 15	20 7	12 11	1 30 4	26 4	4 15	26 4	26 4	P9						
180	2 5	4 3	8 7	2 9	10 12	10 8	2 5	10 8	10 8	10 12	P10					
965	2 11	4 9	10 11	4 13	18 10	22 2	4 9	20 4	16 8	22 6	10 6	P11				
143	2 5	4 3	10 5	6 5	10 12	10 8	4 3	10 8	10 8	10 12	6 4	8 8	P12			
67	2 12	4 10	18 4	12 6	24 5	12 13	4 10	14 11	18 7	20 9	6 11	8 15	10 7	P13		
34	2 9	4 7	10 9	4 11	18 8	18 4	4 7	18 4	16 6	18 8	10 4	18 2	8 6	8 13	P14	
247	2 10	4 8	16 4	10 6	20 7	18 5	4 8	20 3	20 3	20 7	10 5	14 7	10 5	14 8	14 5	P15

Figura 6.28. Segunda iteración

En la Figura 6.28. se muestra una segunda iteración. Observamos que el ahorro inmediatamente inferior al anterior (30) es 26. Dicho ahorro es igual para varias casillas, por lo que se podría elegir al azar cualquiera de ellas, pero como ya se indicó anteriormente, en este ejemplo se recorren las filas de arriba abajo y de izquierda a derecha.

Por tanto, se considera la unión de los clientes P_4 y P_8. En primer lugar hay que fijarse en si los clientes ya están unidos, y en si $t_{4,0}$ y $t_{8,0}$ son mayores que 0, que en este caso sí lo son. También se debe evaluar si unir dichos clientes es factible en cuanto a la restricción de capacidad de los vehículos. Recordemos que el cliente P_4 ya se encontraba unido al cliente P_9, por lo que se tiene que verificar que $(d_4 + \ d_9) + d_8 \leq C$. En este caso, esta condición se cumple ya que 255 + 500 ≤ 1500.

Las siguientes iteraciones se llevan a cabo de la misma manera, recorriendo las celdas en orden decreciente del valor del ahorro asociado. En la Figura 6.29. se muestra la solución final del método de Clarke & Wright aplicado a este problema.

Q	P0															
1194	1	P1														
			P2													
696	1			P3												
					P4											
						P5										
	1			1			P6									
								P7								
			1		1				P8							
					1			1		P9						
	1					1					P10					
965	2											P11				
1352	1												P12			
			1										1	P13		
		1				1									P14	
	1							1								P15

Figura 6.29. Solución inicial generada mediante el método de Clarke & Wright

Las rutas que forman la solución generada son las siguientes:

Ruta 1: $P_0 - P_{12} - P_{13} - P_2 - P_8 - P_4 - P_9 - P_7 - P_{15} - P_0$
DSR = 1352 Coste = (5+7+4+3+4+4+4+3+10) = 44

Ruta 2: $P_0 - P_{10} - P_5 - P_{14} - P_1 - P_0$
DSR = 1194 Coste = (5+8+4+7+2) = 26

Ruta 3: $P_0 - P_3 - P_6 - P_0$
DSR = 696 Coste = (6+4+2) = 12

Ruta 4: $P_0 - P_{11} - P_0$
DSR = 965 Coste = (11+11) = 22

Si se quisiera utilizar esta solución generada como solución de partida del método de búsqueda tabú granular sería necesario realizar algunos ajustes. Esto se debe a que se han obtenido cuatro rutas en la solución mientras que el problema solo consideraba tres vehículos.

Para solucionar este problema se debe eliminar la ruta menos cargada, es decir, aquella ruta cuya DSR sea menor y reinsertar los clientes de dicha ruta en las rutas restantes de la mejor manera posible. En este caso, la ruta menos cargada es la Ruta 3 que visita a los clientes situados en P_3 y P_6.

A la hora de reinsertar a los clientes P_3 y P_6 se debe optar por incluirlos de forma que la capacidad de los vehículos (C) se exceda en la menor cantidad posible. Es evidente que la mejor manera de reinsertarlos consiste en incluir a P_3 en la Ruta 2 y a P_6 en la Ruta 4. Así, la DSR en las rutas 1, 2 y 4 sería 1352, 1570 y 1341, respectivamente.

A continuación surge la duda de dónde colocar a dichos clientes dentro de la ruta, porque en función del lugar en el que se inserten, el coste de la ruta se va a ver afectado en mayor o menor medida.

En el caso del cliente P_6 cuando se inserta en la Ruta 4, no es necesario evaluar cuál es la mejor posición debido a que la ruta solo contiene un cliente. La ruta quedaría como sigue: $P_0 - P_6 - P_{11} - P_0$.

Para el cliente P_3 la situación es distinta porque hay varias posibilidades al insertarlo en la Ruta 2. Por tanto hay que evaluar la posición en la que insertar a dicho cliente, de forma que el coste de la ruta sea el menor posible.

En la Figura 6.30. se muestra una ruta inicial con los clientes P_A y P_B y la misma ruta a la que se ha añadido un cliente P_X. El coste de la ruta inicial es $d_{0A} + d_{AB} + d_{B0}$. El coste de la ruta en la que se inserta el cliente P_X es $d_{0A} + d_{AX} + d_{XB} + d_{B0}$. Al insertar el cliente en una posición cualquiera se rompe un enlace y se añaden dos nuevos, y el incremento en el coste de la ruta se calcula como la diferencia entre los costes de ambas ruta. Por tanto, la diferencia en el coste se obtiene mediante la siguiente expresión:

$$(d_{0A} + d_{AX} + d_{XB} + d_{B0}) - (d_{0A} + d_{AB} + d_{B0}) = d_{AX} + d_{XB} - d_{AB}$$

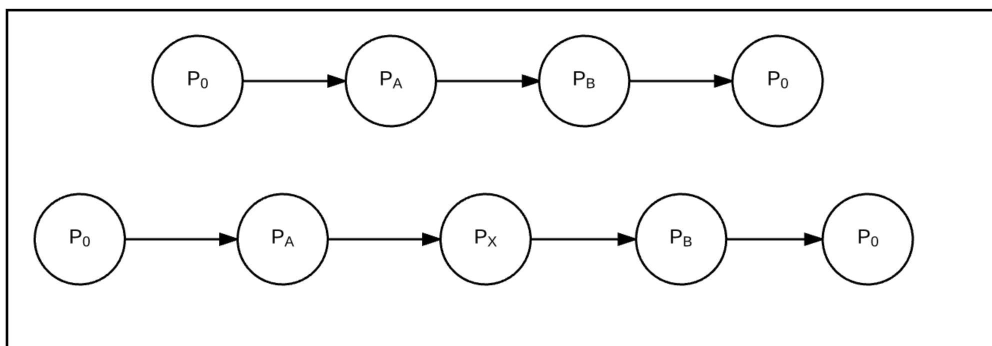


Figura 6.30. Efecto en el coste de insertar un cliente en una ruta.

A continuación se estudian las distintas posibilidades de insertar al cliente P_3 en la Ruta 2 ($P_0 - P_{10} - P_5 - P_{14} - P_1 - P_0$), calculando la diferencia en el coste de la ruta.

- Entre P_0 y P_{10}: $d_{0,3} + d_{3,10} - d_{0,10}$ = 6 + 9 - 5 = 10
- Entre P_{10} y P_5: $d_{10,3} + d_{3,5} - d_{10,5}$ = 9 + 11 – 8 = 12
- Entre P_5 y P_{14}: $d_{5,3} + d_{3,14} - d_{5,14}$ = 11+ 11 – 4 = 18
- Entre P_{14} y P_1: $d_{14,3} + d_{3,1} - d_{14,1}$ = 7 + 6 – 7 = 6
- Entre P_1 y P_0 : $d_{1,3} + d_{3,0} - d_{1,0}$ = 6 + 6 – 2 = 10

La mejor opción es insertar al cliente P_3 entre los clientes P_{14} y P_1, ya que el incremento en el coste es el menor entre todas las posibilidades.

Finalmente el conjunto de las rutas quedaría como sigue:

Ruta 1: $P_0 - P_{12} - P_{13} - P_2 - P_8 - P_4 - P_9 - P_7 - P_{15} - P_0$
DSR = 1352 Coste = (5+7+4+3+4+4+4+3+10) = 44

Ruta 2: $P_0 - P_{10} - P_5 - P_{14} - P_3 - P_1 - P_0$
DSR = 1570 Coste=(5+8+4+11+6+2)+(1570-1500)x100=7036

Ruta 3: $P_0 - P_6 - P_{11} - P_0$
DSR = 1341 Coste = (2+9+11) = 22

El coste total de la solución es 7102 y cabe señalar que una de las rutas excede la capacidad C de los vehículos, por lo que dicho coste total incluye la penalización por exceso de capacidad.

Capítulo 7

Programación de la búsqueda tabú granular aplicada al problema VRP

El presente capítulo está dedicado a describir la programación que se ha hecho de la búsqueda tabú granular para el problema del enrutamiento de vehículos. Para la implementación del método se ha utilizado Python, un lenguaje de programación relativamente reciente. Los motivos de esta elección se exponen en la primera sección de este capítulo. Además se ven los detalles de la implementación y las explicaciones necesarias para entender el código del programa, que se incluye en el anexo. En concreto se definen ciertas estructuras de datos empleadas en el programa y se detalla la tarea que realizan cada una de las funciones programadas. Por último, se muestran los resultados experimentales conseguidos por el programa sobre distintos problemas.

7.1. El lenguaje Python

Python es un lenguaje de programación creado en los años 90 por el holandés Guido van Rossum en el instituto de investigación Centrum Wiskunde & Informatica (centro para investigación en Matemáticas e Informática) de Amsterdam. Debido a su simplicidad, con el paso del tiempo, fue ganando popularidad entre los programadores y fue adoptado por grandes empresas.

A diferencia de otro software de programación, Python es de distribución libre y accesible para cualquier persona, y esto resulta ser una gran ventaja. De hecho, Python se puede descargar desde varias páginas web, como por ejemplo desde Anaconda: https://anaconda.org/anaconda/python. Además, Python es multiplataforma, lo cual quiere decir que puede ser interpretado por diferentes sistemas operativos como Windows, Linux o MacOS. También cabe señalar que Python cuenta con una gran comunidad trabajando en su desarrollo.

Al ser un lenguaje de alto nivel, es decir, con una estructura sintáctica y semántica adecuada a la capacidad cognitiva humana, es muy utilizado como herramienta de aprendizaje de programación informática.

En comparación con otros lenguajes de programación como C++ o Java, Python tiene un nivel de abstracción más elevado, por lo que resulta más fácil de aprender. Esta ventaja lleva asociada un inconveniente y es que los programas implementados en Python son generalmente más lentos en ejecutarse. En términos de funcionalidad, Python es tan potente como C++ o Java.

Otra ventaja de Python es que es un lenguaje interpretado, es decir, no es necesario un compilador para ejecutar un programa. Por el contrario, necesita un intérprete que funciona de manera similar a un compilador pero sin necesidad de generar un ejecutable. También cabe destacar que Python es un lenguaje de programación que admite la programación orientada a objetos.

Un punto más a favor de Python es su librería estándar, a la que se puede acceder una vez que se descarga el paquete de instalación. Dicha librería estándar está formada por diferentes módulos, que incorporan distintas funcionalidades para facilitar la programación. Uno de los módulos más utilizados en Python es el módulo de *Numpy*, acrónimo de "Numeric Python" o "Numerical Python". Este módulo dota a Python de funciones rápidas precompiladas para rutinas matemáticas y númericas. Por ejemplo, para sumar matrices no es necesario programar esta suma componente a componente, sino que el módulo *Numpy* tiene un comando que realiza dicha operación. Este módulo enriquece Python con estructuras de datos potentes que permiten cálculos eficientes con matrices y arrays multidimensionales. Además contiene una extensa librería de funciones matemáticas de alto nivel que trabajan con matrices y vectores. Otro módulo utilizado frecuentemente en Python es el módulo *Random*, el cual permite generar números aleatorios. Otros módulos de gran interés son ScyPy, que extiende las capacidades de NumPy con funciones para el cálculo científico, y Matplotlib, la librería de dibujo que permite hacer representaciones gráficas de funciones de manera sencilla.

Aparte de los módulos de la librería estándar, el usuario puede crear sus propios módulos, integrados por distintas funciones que, programadas de forma genérica, simplifiquen la programación de otras aplicaciones.

Por último, se debe hacer un apunte sobre Jupyter, que es una interfaz que permite programar en Python. El entorno de programación Jupyter se muestra en la Figura 7.1. donde se ve que es una interfaz muy sencilla de manejar y en la que el usuario puede programar de manera interactiva en un ejecutable. Actualmente la versión disponible incorpora pocas opciones en la barra de

herramientas, aunque se está desarrollando una nueva versión que amplíe las distintas opciones.

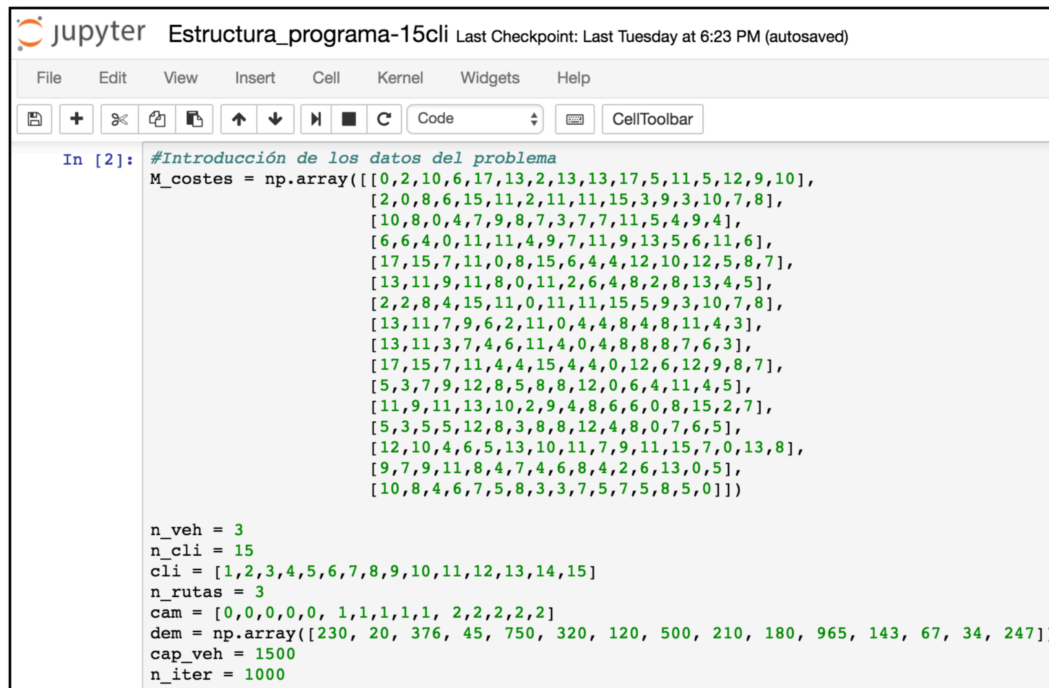


Figura 7.1. Entorno de programación Jupyter

Como conclusión, Python resulta ser un lenguaje adecuado para la programación científica. Por esto y por todas las ventajas citadas anteriormente, se ha elegido Python para programar la búsqueda tabú granular en el presente estudio. Si desea información relativa a cómo programar en Python puede dirigirse a https://docs.python.org/3/ donde encontrará un tutorial y otra documentación relacionada.

7.2. Búsqueda tabú granular en Python

En esta sección se procede a describir las claves de la implementación llevada a cabo de la búsqueda tabú granular en Python aplicada al problema de enrutamiento de vehículos. De esta manera el lector será capaz de entender el código del programa que se encuentra en el anexo.

El problema del viajante, en el que se persigue determinar el orden de los clientes a los que se visita dentro de una misma ruta, resulta fácil de programar con una estructura de datos sencilla, como por ejemplo, un vector. Esto se debe a que el tamaño del vector es fijo, ya que el número de clientes en la ruta no varía y solo se busca determinar el orden en el que se les suministra, con el fin de reducir el coste total.

En cambio, el problema VRP consiste en encontrar un conjunto de rutas que minimicen el coste total. En la ejecución de las distintas iteraciones de la búsqueda tabú granular, la longitud de las rutas que forman la solución actual varía, ya que se eliminan e insertan clientes en dichas rutas mediante los movimientos de k-intercambio. Debido a esto, los vectores no resultan ser una estructura de datos adecuada y pueden llegar a dar problemas, ya que su longitud varía a lo largo de la búsqueda en función del número de clientes que forman cada ruta. Por tanto, se hace necesario encontrar una estructura de datos más apropiada para este problema. Además, surge también la necesidad de buscar una estructura de datos adecuada para implementar el grafo disperso. Tal y como proponen (Toth & Vigo, 2003) las listas enlazadas son una buena opción para implementar el grafo disperso. Aunque Toth & Vigo utilizan las listas enlazadas para el grafo disperso, en la programación desarrollada en este trabajo también se han empleado las listas enlazadas para implementar las rutas que componen una solución.

Una lista enlazada es una estructura de datos que almacena de forma lineal una colección de elementos. A estos elementos se les denomina nodos y cada uno de ellos apunta al siguiente nodo por medio de un puntero, en el caso de una lista simple, y al nodo siguiente y al anterior en el caso de una lista doble.

En la programación de la búsqueda tabú granular para el problema VRP se han utilizado listas doblemente enlazadas de forma que cada nodo se corresponde con un cliente, y cada cliente apunta al cliente inmediatamente anterior y posterior. Un ejemplo de lista enlazada para una ruta es el siguiente:

0, 6, 3, 1, 2, 8, 0

Como se puede observar, el almacén (0) se incluye al inicio y al final de la ruta para asegurar que la ruta empieza y termina en el almacén.

Para el caso de la implementación del grafo disperso los nodos que componen las listas almacenan los arcos que forman dicho grafo disperso. En el ejemplo que se muestra a continuación se puede observar que cada nodo representa un arco:

Grafo disperso:
(0,1),
(0,2),
(1,2),
...

Esta estructura de datos se ha programado por medio de objetos, de manera que se han creado dos clases: ListaEnlazada y Nodo. Con la primera se consiguen listas doblemente enlazadas para representar tanto las rutas como el grafo disperso. La segunda, la clase Nodo, permite crear los nodos que forman las listas enlazadas. Los atributos de la clase ListaEnlazada son la cabeza y la cola que hacen referencia al nodo inicial y final, respectivamente, de la ListaEnlazada. En el caso de la clase Nodo los atributos son el nodo anterior y el nodo siguiente, así como el dato que contiene el número del cliente al que representa dicho nodo en el caso de las rutas y el arco en el caso del grafo.

Gracias a esta estructura de datos, resulta sencillo realizar los distintos movimientos, ya que las rutas se pueden modificar simplemente indicando cuál es el cliente anterior y sucesivo de cada uno de los clientes. Además, la clase ListaEnlazada tiene programados una serie de métodos que facilitan la programación de la búsqueda tabú granular. Entre estos métodos se encuentra el método "buscar" que busca un determinado cliente dentro de una ruta, el método "mostrar" para imprimir los datos de los nodos que componen la ListaEnlazada, el método "enlazar_lista" para añadir un nodo al final o el método "copiar" para crear una copia de una ListaEnlazada.

Además, para estructurar el programa y para facilitar su lectura, se ha creado un módulo de usuario (BTGranular) que contiene las funciones utilizadas a lo largo de la programación de la búsqueda.

Con el fin de situar al lector y facilitar la comprensión de la estructura que tiene el programa se incluye a continuación un seudocódigo, en el que se indican, de manera genérica, los distintos pasos que sigue el programa para alcanzar la mejor solución.

1. Introducir datos del problema (n, d, c_{ij}, k, C)
2. Introducir parámetros del método: ($\beta_{inicial}$, β_d, α_{min}, α_{max}, α_c, n_i, t_{min}, t_{max}, n_d, n_h, iter)
3. Introducir solución inicial y evaluar su coste
4. Iniciar mejor solución y su coste (coincide con la solución inicial)
5. Inicializar lista tabú
6. Inicializar contadores (Cont_nh, Cont_nd, Cont_fac, Cont_no_fact)
7.
8. Solución actual = solución inicial
9. Para un número de iteraciones dado (iter):
10.
11. Si la iteración es múltiplo de 2n:

12. Construir grafo disperso con $\beta_{inicial}$
13. Si el cont_nd = n_{d}:
14. Solución actual = mejor solución
15. Cambiar umbral a β_d y recalcular el grafo disperso
16. Comienza a contar el cont_nh y cont_nd = 0
17. Si cont_nh = n_h +1:
18. Cambiar umbral a $\beta_{inicial}$ y recalcular el grafo disperso
19. Reiniciar cont_nh = 0
20. En otro caso:
21. Actualizar grafo disperso (insertar arcos añadidos en iteración anterior)
22.
23. Para cada arco presente en el grafo disperso:
24. Si los clientes que forman el arco no están en la misma ruta y el arco no es tabú:
25. Para cada tipo de movimiento (2k, 3ak , 3bk y 4k):
26. Si el movimiento es factible:
27. Evaluar variación en el coste con respecto al arco anterior
28. Seleccionar movimiento con mejor variación
29. Actualizar solución actual mediante movimiento almacenado
30. Almacenar arcos introducidos y eliminados por el movimiento
31. Actualizar coste solución actual
32. Actualizar contadores de penalización de carga:
33. Si la solución es factible respecto a la restricción de capacidad:
34. cont_fact se incrementa
35. cont_no_fact = 0
36. Si la solución no es factible respecto a la restricción de capacidad:
37. cont_no_fact se incrementa
38. cont_fact = 0
39. Actualizar parámetro de penalización (α_c) si es necesario:
40. Si cont_fact=ni → α_c=max{α_{min}, $\alpha_c/2$} y reiniciar cont_fact
41. Si cont_no_fact=ni → α_c=min{$\alpha_{máx}$, $2\alpha_c$} y reiniciar cont_no_fact
42. Actualizar lista tabú (arcos eliminados)
43. Incrementar cont_nh si está comprendido entre 1 y n_h
44. Si cont_nh = 0:
45. Actualizar cont_nd
46. Si el coste de la solución actual es menor que el de la mejor solución calculada:
47. Actualizar mejor solución y su coste
48. cont_nd = 0:
49. Mostrar mejor solución y su coste

7.3. Módulo de usuario BTGranular

Ahora que ya se presentado la estructura básica del programa, se procede a mostrar un índice de los distintos archivos que componen el módulo de usuario BTGranular ordenados por orden alfabético. Además, para cada archivo se indica las funciones que contiene y una breve descripción de la tarea que realizan dichas funciones cuando son llamadas.

1. **arcos_sol**:

   – arcos_soluc: función que crea una ListaEnlazada con los arcos que forman parte de una solución.

2. **clases**:

   – Nodo: clase para definir los nodos que forman las ListaEnlazada. La clase Nodo almacena el número del cliente, el cliente inmediatamente anterior y el inmediatamente posterior.

   – ListaEnlazada: clase para definir las listas enlazadas que contienen la información (a través de los Nodo) de las distintas rutas que forman una solución. La clase ListaEnlazada tiene programados los siguientes métodos:
     - buscar: determina si un cliente está presente en una ruta.
     - mostrar: imprime los clientes que forman parte de una ruta.
     - enlazar_lista: añade un cliente al final de una ruta.
     - copiar: crea una copia de una ListaEnlazada.

3. **coste:**

   – penaliz: función que calcula la penalización por exceso de capacidad para una solución.

   – calc_coste: función que realiza el cálculo del coste de una solución sin tener en cuenta el coste por exceso de capacidad, es decir, suma los costes asociados a los arcos de las rutas que forman una solución.

4. **demsatisf**:

   – dem_satisf: función que calcula la demanda satisfecha en cada una de las rutas que forman parte de una solución.

5. **factible**:

   – cli_misma_ruta: función que determina si dos clientes cualesquiera forman parte de una misma ruta.

   – fact_3ak: función que determina si el movimiento 3-intercambio con la opción (a) es factible para un arco dado.

   – fact_3bk: función que determina si el movimiento 3-intercambio con la opción (b) es factible para un arco dado.

   – fact_4k: función que determina si el movimiento 4-intercambio es factible para un arco dado.

6. **grafodisperso**:

   – constr_gd: función que crea una ListaEnlazada con los arcos que forman el grafo disperso. Para ello inserta los arcos incidentes con el almacén, los arcos cuyo coste es menor que el umbral de granularidad, los arcos de la mejor solución encontrada y los arcos de la solución actual.

   – act_gd: función que actualiza el grafo disperso añadiendo a la ListaEnlazada los arcos que se han insertado en las rutas en la iteración anterior en consecuencia al movimiento realizado.

7. **interc_arcos**:

   – mov_2k: función que realiza el movimiento mediante 2-intercambio completo para un arco dado, teniendo en cuenta que el movimiento sea factible para ese arco y considerando las distintas combinaciones posibles si uno de los clientes que forman parte del arco es el almacén.

   – mov_3ak: función que realiza el movimiento 3a-intercambio completo para un arco dado, teniendo en cuenta que el movimiento sea factible para ese arco y considerando las distintas combinaciones posibles si uno de los clientes que forman parte del arco es el almacén.

   – mov_3bk: función que realiza el movimiento 3b-intercambio completo para un arco dado, teniendo en cuenta que el movimiento

sea factible para ese arco y considerando las distintas combinaciones posibles si uno de los clientes que forman parte del arco es el almacén.

- mov_4k: función que realiza el movimiento 4-intercambio completo para un arco dado, teniendo en cuenta que el movimiento sea factible para ese arco y considerando las distintas combinaciones posibles si uno de los clientes que forman parte del arco es el almacén.

- mov_arco: función que realiza los intercambios (2, 3(a), 3(b) y 4) para un arco dado y determina cuál es el movimiento que produce una mejor variación.

8. **mejorsol**:

- act_mejor: función que actualiza la mejor solución si la solución encontrada en una iteración es mejor o igual que la mejor solución encontrada hasta el momento.

- in_mejor_sol: función que inicializa la mejor solución al comienzo del algoritmo creando una copia de la solución inicial.

9. **mov_iteracion:**

- mov_grafo: función que realiza los 4 movimientos para cada arco del grafo disperso, en caso de que el arco no sea tabú, el arco no forme parte de la solución actual ni que los clientes que forman el arco estén en la misma ruta.

10. **movimientos:**

- dosk: función que realiza un intercambio 2k para un arco dado una vez que dicho movimiento se ha elegido como el mejor en una iteración y determina los arcos que son eliminados e insertados en dicho movimiento.

- tresAk: función que realiza un intercambio 3ak para un arco dado una vez que dicho movimiento se ha elegido como el mejor en una iteración y determina los arcos que son eliminados e insertados en dicho movimiento.

- tresBk: función que realiza un intercambio 3bk para un arco dado una vez que dicho movimiento se ha elegido como el mejor en una iteración y determina los arcos que son eliminados e insertados en dicho movimiento.

- cuatrok: función que realiza un intercambio 4k para un arco dado una vez que dicho movimiento se ha elegido como el mejor en una iteración y determina los arcos que son eliminados e insertados en dicho movimiento.

- dos_interc: función que calcula la variación en el coste que supone realizar un intercambio 2k para un arco. No lleva a cabo el intercambio de arcos, ya que es la función que se utiliza dentro de la iteración para identificar qué arco y qué movimiento es el que produce mejor variación.

- tresA_interc: función que calcula la variación en el coste que supone realizar un intercambio 3ak para un arco. No lleva a cabo el intercambio de arcos, ya que es la función que se utiliza dentro de la iteración para identificar qué arco y qué movimiento es el que produce mejor variación.

- tresB_interc: función que calcula la variación en el coste que supone realizar un intercambio 3bk para un arco. No lleva a cabo el intercambio de arcos, ya que es la función que se utiliza dentro de la iteración para identificar qué arco y qué movimiento es el que produce mejor variación.

- cuatro_interc: función que calcula la variación en el coste que supone realizar un intercambio 4k para un arco. No lleva a cabo el intercambio de arcos, ya que es la función que se utiliza dentro de la iteración para identificar qué arco y qué movimiento es el que produce mejor variación.

11. **reconstruir:**

- reconstruir_sol: función que actualiza la solución actual reconstruyéndola a partir de los datos almacenados del arco y movimiento que mejor variación producen en una iteración.

12. **rutacliente**:

- ruta_cli: función que determina en qué ruta de una solución se encuentra un cliente.

- n_cli_ruta: función que calcula el número de clientes que se encuentran en una ruta determinada sin tener en cuenta el almacén.

13. **tabu**:

- act_ltabu: función que actualiza la lista tabú asignando una permanencia tabú aleatoria para cada arco eliminado en la iteración y disminuyendo la permanencia del resto de arcos en la lista.

Gracias al seudocódigo y a la descripción de las diferentes funciones que componen el módulo BTGranular, el lector se puede hacer una idea general de cómo funciona el programa. Los detalles de la programación se encuentra en el anexo que contiene el código completo del programa.

7.4. Resultados computacionales

Una vez programado el método de la búsqueda tabú granular para el problema VRP se ha realizado una batería de pruebas sobre varios problemas para evaluar el funcionamiento del programa.

PROBLEMA 1

Uno de los problemas que se ha probado es el que se ha tratado en el Capítulo 6 para la ilustración del método de la búsqueda tabú granular y el método de Clarke & Wright. En este problema, se ha partido de la misma solución inicial que la que se tomó en el ejemplo del Capítulo 6. Dicha solución estaba formada por las rutas:

Ruta 1: $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_0$
Ruta 2: $P_0 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$
Ruta 3: $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_0$

Recordemos que esta solución tenía un coste de 138 y en el ejemplo manual en la primera iteración se obtenía una solución con coste 132, que no lograba mejorar en la segunda iteración.

Al ejecutar el programa de la búsqueda tabú granular se obtiene una solución formada por las rutas:

Ruta 1: $P_0 - P_6 - P_3 - P_2 - P_8 - P_{15} - P_0$
Ruta 2: $P_0 - P_{12} - P_{13} - P_4 - P_9 - P_7 - P_5 - P_0$
Ruta 3: $P_0 - P_{10} - P_{11} - P_{14} - P_1 - P_0$

Esta solución tiene un coste de 88 y se consigue realizando un número relativamente pequeño de iteraciones, ya que a partir de 50 iteraciones se llega a esta solución. Este número de iteraciones puede variar en función de la permanencia tabú que se asigne a los arcos eliminados en cada iteración, ya que dicha permanencia tabú recordemos que se generaba de manera aleatoria. Por tanto, gracias al programa se consigue una reducción en el coste de la solución con respecto a la solución inicial de un 63,76%.

Además, el programa resulta ser bastante rápido en la resolución de este problema, aunque el tiempo de ejecución varía dependiendo del ordenador en el que se ejecute el programa. Cabe señalar que en la implementación del método no se ha utilizado ningún criterio de parada adicional al número máximo de iteraciones, siguiendo lo expuesto en el artículo de Toth & Vigo, por lo que el programa continúa con la búsqueda aunque lleve un número grande de iteraciones sin mejorar el coste de la solución.

Es evidente que la ventaja que presenta este método es enorme, ya que se consigue una solución que mejora con creces a la solución inicial en un tiempo relativamente corto.

En cambio, si ejecutamos el programa partiendo de la solución inicial calculada mediante el método de Clarke & Wright en la sección 6.2. del presente trabajo, la búsqueda de la mejor solución se complica, ya que dicha solución inicial es peor que la que se eligió de manera aleatoria, ya que recordemos que estaba penalizada por exceso de capacidad en una de las rutas. En consecuencia se requiere un poco más de tiempo para llegar a la solución de coste 88.

PROBLEMA 2

En este segundo caso se ha recurrido a un problema propuesto en la literatura disponible para el problema VRP. En concreto, se ha tomado un problema propuesto en (Christofides & Eilon, 1969), que es del tipo VRP Euclídeo (descritos en la sección 2.2.) con doce clientes y cuatro vehículos. La capacidad máxima de los vehículos es $C = 6000$. La demanda de los clientes viene dada por el siguiente vector:

d=[1200,1700,1500,1400,1700,1400,1200,1900,1800,1600,1700,1100]

La matriz de costes (c_{ij}) se consigue mediante el cálculo de la distancia Euclídea para cada par de clientes. A continuación se muestran las posiciones de los clientes:

```
C[0,1]=9.0
C[0,2]=14.0
C[1,2]=21.0
C[0,3]=23.0
C[1,3]=22.0
C[2,3]=25.0
C[0,4]=32.0
C[1,4]=36.0
C[2,4]=38.0
C[3,4]=42.0
C[0,5]=50.0
C[1,5]=52.0
C[2,5]=5.0
C[3,5]=12.0
C[4,5]=22.0
C[9,12]=8.0
C[10,12]=10.0
C[11,12]=10.0
C[0,6]=21.0
C[1,6]=24.0
C[2,6]=31.0
C[3,6]=35.0
C[4,6]=37.0
C[5,6]=41.0
C[0,7]=49.0
C[1,7]=51.0
C[2,7]=7.0
C[3,7]=17.0
C[4,7]=16.0
C[5,7]=23.0
C[6,7]=26.0
C[0,8]=30.0
C[1,8]=36.0
C[2,8]=36.0
C[3,8]=44.0
C[4,8]=46.0
C[5,8]=10.0
C[6,8]=21.0
C[7,8]=30.0
C[0,9]=27.0
C[1,9]=37.0
C[2,9]=43.0
C[3,9]=31.0
C[4,9]=37.0
C[5,9]=39.0
C[6,9]=19.0
C[7,9]=28.0
C[8,9]=25.0
C[0,10]=35.0
C[1,10]=41.0
C[2,10]=29.0
C[3,10]=31.0
C[4,10]=29.0
C[5,10]=9.0
C[6,10]=10.0
C[7,10]=16.0
C[8,10]=22.0
C[9,10]=20.0
C[0,11]=28.0
C[1,11]=30.0
C[2,11]=7.0
C[3,11]=11.0
C[4,11]=13.0
C[5,11]=17.0
C[6,11]=25.0
C[7,11]=27.0
C[8,11]=10.0
C[9,11]=16.0
C[10,11]=10.0
C[0,12]=18.0
C[1,12]=20.0
C[2,12]=6.0
C[3,12]=6.0
C[4,12]=14.0
C[5,12]=16.0
C[6,12]=12.0
C[7,12]=12.0
C[8,12]=20.0
```

La matriz de costes calculada quedaría como sigue:

```
[[  0.   9.  14.  23.  32.  50.  21.  49.  30.  27.  35.  28.  18.]
 [  9.   0.  21.  22.  36.  52.  24.  51.  36.  37.  41.  30.  20.]
 [ 14.  21.   0.  25.  38.   5.  31.   7.  36.  43.  29.   7.   6.]
 [ 23.  22.  25.   0.  42.  12.  35.  17.  44.  31.  31.  11.   6.]
 [ 32.  36.  38.  42.   0.  22.  37.  16.  46.  37.  29.  13.  14.]
 [ 50.  52.   5.  12.  22.   0.  41.  23.  10.  39.   9.  17.  16.]
 [ 21.  24.  31.  35.  37.  41.   0.  26.  21.  19.  10.  25.  12.]
 [ 49.  51.   7.  17.  16.  23.  26.   0.  30.  28.  16.  27.  12.]
 [ 30.  36.  36.  44.  46.  10.  21.  30.   0.  25.  22.  10.  20.]
 [ 27.  37.  43.  31.  37.  39.  19.  28.  25.   0.  20.  16.   8.]
 [ 35.  41.  29.  31.  29.   9.  10.  16.  22.  20.   0.  10.  10.]
 [ 28.  30.   7.  11.  13.  17.  25.  27.  10.  16.  10.   0.  10.]
 [ 18.  20.   6.   6.  14.  16.  12.  12.  20.   8.  10.  10.   0.]]
```

La solución inicial está formada por las siguientes rutas:

Ruta 1: $P_0 - P_1 - P_2 - P_3 - P_0$
Ruta 2: $P_0 - P_4 - P_5 - P_6 - P_0$
Ruta 3: $P_0 - P_7 - P_8 - P_9 - P_0$
Ruta 4: $P_0 - P_{10} - P_{11} - P_{12} - P_0$

El coste de esta solución inicial es de 398 y no tiene penalización por exceso de capacidad en ninguna ruta. Mediante la ejecución de la búsqueda tabú granular, la mejor solución conseguida tiene un coste de 247 y el menor

número de iteraciones necesario para alcanzar dicho coste está alrededor de 200. Dicha solución está formada por las siguientes rutas:

Ruta 1: $P_0 - P_{11} - P_4 - P_7 - P_2 - P_0$
Ruta 2: $P_0 - P_3 - P_5 - P_8 - P_0$
Ruta 3: $P_0 - P_1 - P_0$
Ruta 4: $P_0 - P_6 - P_{10} - P_{12} - P_9 - P_0$

Al igual que en el problema estudiado en el caso anterior, la solución obtenida es mucho mejor que la inicial y debido a que el número de clientes presentes en el problema no es grande el programa no requiere demasiado tiempo para llegar a esta solución, aunque no se ha medido exactamente el tiempo de ejecución que ha requerido alcanzar esta solución.

PROBLEMA 3

De nuevo en este tercer caso se ha recurrido a experimentar con un problema típico en los estudios del problema VRP Euclídeo propuesto en (Christofides & Eilon, 1969). Este problema tiene un tamaño más grande ya que consta de 50 clientes y 5 vehículos. En este caso la capacidad de los vehículos es $C = 160$. El vector que contiene las demandas de los clientes se muestra a continuación:

d=[7,30,16,9,21,15,19,23,11,5,19,29,23,21,10,15,3,41,9,28,8,8,16,10,28, 7,15,14,6,19,11,12,23,26,17,6,9,15,14,7,27,13,11,16,10,5,25,17,18,10]

La matriz de costes de este problema es una matriz 51 x 51 por lo que para simplificar la exposición se muestran a continuación la posición de los clientes. La matriz de costes se ha construido realizando el cálculo de la distancia Euclídea para cada par de clientes.

```
pos[0]=[30.0,40.0]
```

```
pos[1]=[37.0,52.0]
pos[2]=[49.0,49.0]
pos[3]=[52.0,64.0]
pos[4]=[20.0,26.0]
pos[5]=[40.0,30.0]
pos[6]=[21.0,47.0]
pos[7]=[17.0,63.0]
pos[8]=[31.0,62.0]
pos[9]=[52.0,33.0]
pos[10]=[51.0,21.0]
pos[11]=[42.0,41.0]
pos[12]=[31.0,32.0]
pos[13]=[5.0,25.0]
pos[14]=[12.0,42.0]
pos[15]=[36.0,16.0]
pos[16]=[52.0,41.0]
pos[17]=[27.0,23.0]
pos[18]=[17.0,33.0]
pos[19]=[13.0,13.0]
pos[20]=[57.0,58.0]
pos[21]=[62.0,42.0]
pos[22]=[42.0,57.0]
pos[23]=[16.0,57.0]
pos[24]=[8.0,52.0]
pos[25]=[7.0,38.0]
pos[26]=[27.0,68.0]
pos[27]=[30.0,48.0]
pos[28]=[43.0,67.0]
pos[29]=[58.0,48.0]
pos[30]=[58.0,27.0]
```

```
pos[31]=[37.0,69.0]    pos[41]=[10.0,17.0]
pos[32]=[38.0,46.0]    pos[42]=[21.0,10.0]
pos[33]=[46.0,10.0]    pos[43]=[5.0,64.0]
pos[34]=[61.0,33.0]    pos[44]=[30.0,15.0]
pos[35]=[62.0,63.0]    pos[45]=[39.0,10.0]
pos[36]=[63.0,69.0]    pos[46]=[32.0,39.0]
pos[37]=[32.0,22.0]    pos[47]=[25.0,32.0]
pos[38]=[45.0,35.0]    pos[48]=[25.0,55.0]
pos[39]=[59.9,15.0]    pos[49]=[48.0,28.0]
pos[40]=[5.0,6.0]      pos[50]=[56.0,37.0]
```

La solución inicial considerada para este problema es la formada por las siguientes rutas:

Ruta 1: $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_6 - P_7 - P_8 - P_9 - P_{10} - P_0$
Ruta 2: $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_{16} - P_{17} - P_{18} - P_{19} - P_{20} - P_0$
Ruta 3: $P_0 - P_{21} - P_{22} - P_{23} - P_{24} - P_{25} - P_{26} - P_{27} - P_{28} - P_{29} - P_{30} - P_0$
Ruta 4: $P_0 - P_{31} - P_{32} - P_{33} - P_{34} - P_{35} - P_{36} - P_{37} - P_{38} - P_{39} - P_{40} - P_0$
Ruta 5: $P_0 - P_{41} - P_{42} - P_{43} - P_{44} - P_{45} - P_{46} - P_{47} - P_{48} - P_{49} - P_{50} - P_0$

Esta solución tiene un coste de 5255,4 que incluye una penalización de 3800 por exceso de capacidad. Tras ejecutar la búsqueda tabú granular se obtiene una solución con coste 557,88, por lo que al igual que en los problemas anteriores se consigue una gran mejora con respecto a la solución inicial. Dicha solución se alcanza al realizar alrededor de 2000 iteraciones y está formada por las siguientes rutas:

Ruta 1: $P_0 - P_{32} - P_1 - P_{22} - P_{20} - P_{36} - P_{35} - P_3 - P_{28} - P_{31} - P_{26} - P_8 - P_0$
Ruta 2: $P_0 - P_{47} - P_4 - P_{41} - P_{13} - P_{19} - P_{40} - P_{42} - P_{17} - P_{18} - P_0$
Ruta 3: $P_0 - P_{12} - P_{37} - P_{15} - P_{44} - P_{45} - P_{33} - P_{39} - P_{10} - P_{49} - P_5 - P_{46} - P_0$
Ruta 4: $P_0 - P_{38} - P_9 - P_{34} - P_{30} - P_{50} - P_{16} - P_{21} - P_{29} - P_2 - P_{11} - P_0$
Ruta 5: $P_0 - P_6 - P_{25} - P_{14} - P_{24} - P_{43} - P_{23} - P_7 - P_{48} - P_{27} - P_0$

La solución obtenida mediante el programa la podemos comparar con la mejor solución conseguida mediante otros métodos e incluso con la propia búsqueda tabú granular programada en (Toth & Vigo, 2003). Los autores consiguen un coste 524,61 para este problema.

Por tanto, la solución obtenida con el programa desarrollado en este trabajo es peor que la que la que obtienen Toth & Vigo para este problema. Esto se puede deber a que el número de iteraciones que ellos realizaron fue mayor, o debido a que el método implementado en este trabajo difiera en algún aspecto, ya que determinados mecanismos no se describen con detalle en el artículo de referencia y se han podido interpretar de distinta manera a la que los autores proponen.

PROBLEMA 4

El último problema con el que se ha experimentado es un problema de nuevo propuesto en (Christofides & Eilon, 1969). El problema es Euclídeo y consta de 75 clientes y 10 vehículos. La capacidad de los vehículos es $C = 140$. El vector que contiene las demandas de los clientes se muestra a continuación:

d=[18,26,11,30,21,19,15,16,29,26,37,16,12,31,8,19,20,13,15,22,28,12,6,27,14,18,17,29,13,22,25,28,27,19,10,12,14,24,16,33,15,11,18,17,21,27,19,20,5,22,12,19,22,16,7,26,14,21,24,13,15,18,11,28,9,37,30,10,8,11,3,1,6,10,20]

La matriz de costes de este problema es una matriz 76 x 76, por lo que como en el problema anterior se muestran las posiciones de los clientes y se calcula la matriz de costes como la distancia Euclídea entre cada par de clientes.

```
pos=np.zeros((76,2))

pos[0]=[40.0,40.0]
pos[1]=[22.0,22.0]
pos[2]=[36.0,26.0]
pos[3]=[21.0,45.0]
pos[4]=[45.0,35.0]
pos[5]=[55.0,20.0]
pos[6]=[33.0,34.0]
pos[7]=[50.0,50.0]
pos[8]=[55.0,45.0]
pos[9]=[26.0,59.0]
#
pos[10]=[40.0,66.0]
pos[11]=[55.0,65.0]
pos[12]=[35.0,51.0]
pos[13]=[62.0,35.0]
pos[14]=[62.0,57.0]
pos[15]=[62.0,24.0]
pos[16]=[21.0,36.0]
pos[17]=[33.0,44.0]
pos[18]=[9.0,56.0]
pos[19]=[62.0,48.0]
#
pos[20]=[66.0,14.0]
pos[21]=[44.0,13.0]
pos[22]=[26.0,13.0]
pos[23]=[11.0,28.0]
pos[24]=[7.0,43.0]
pos[25]=[17.0,64.0]
pos[26]=[27.0,68.0]
pos[27]=[55.0,34.0]
pos[28]=[35.0,16.0]
pos[29]=[52.0,26.0]

pos[30]=[43.0,26.0]
pos[31]=[31.0,76.0]
pos[32]=[22.0,53.0]
pos[33]=[26.0,29.0]
pos[34]=[50.0,40.0]
pos[35]=[55.0,50.0]
pos[36]=[54.0,10.0]
pos[37]=[60.0,15.0]
pos[38]=[47.0,66.0]
pos[39]=[30.0,60.0]
#
pos[40]=[30.0,50.0]
pos[41]=[12.0,17.0]
pos[42]=[15.0,14.0]
pos[43]=[16.0,19.0]
pos[44]=[21.0,48.0]
pos[45]=[50.0,30.0]
pos[46]=[51.0,42.0]
pos[47]=[50.0,15.0]
pos[48]=[48.0,21.0]
pos[49]=[12.0,38.0]
#
pos[50]=[15.0,56.0]
pos[51]=[29.0,39.0]
pos[52]=[54.0,38.0]
pos[53]=[55.0,57.0]
pos[54]=[67.0,41.0]
pos[55]=[10.0,70.0]
pos[56]=[6.0,25.0]
pos[57]=[65.0,27.0]
pos[58]=[40.0,60.0]
pos[59]=[70.0,64.0]

pos[60]=[64.0,4.0]
pos[61]=[36.0,6.0]
pos[62]=[30.0,20.0]
pos[63]=[20.0,30.0]
pos[64]=[15.0,5.0]
pos[65]=[50.0,70.0]
pos[66]=[57.0,72.0]
pos[67]=[45.0,42.0]
pos[68]=[38.0,33.0]
pos[69]=[50.0,4.0]
#
pos[70]=[66.0,8.0]
pos[71]=[59.0,5.0]
pos[72]=[35.0,60.0]
pos[73]=[27.0,24.0]
pos[74]=[40.0,20.0]
pos[75]=[40.0,37.0]
```

Las rutas que forman la solución inicial son las siguientes:

Ruta 1: $P_0 - P_1 - P_2 - P_3 - P_4 - P_5 - P_6 - P_7 - P_8 - P_0$
Ruta 2: $P_0 - P_{11} - P_{12} - P_{13} - P_{14} - P_{15} - P_{16} - P_{17} - P_{18} - P_0$
Ruta 3: $P_0 - P_{21} - P_{22} - P_{23} - P_{24} - P_{25} - P_{26} - P_{27} - P_{28} - P_0$
Ruta 4: $P_0 - P_{31} - P_{32} - P_{33} - P_{34} - P_{35} - P_{36} - P_{37} - P_{38} - P_0$
Ruta 5: $P_0 - P_{41} - P_{42} - P_{43} - P_{44} - P_{45} - P_{46} - P_{47} - P_{48} - P_0$
Ruta 6: $P_0 - P_9 - P_{10} - P_{19} - P_{20} - P_{29} - P_{30} - P_{39} - P_{40} - P_0$
Ruta 7: $P_0 - P_{49} - P_{50} - P_{51} - P_{52} - P_{53} - P_{54} - P_{55} - P_{56} - P_0$
Ruta 8: $P_0 - P_{56} - P_{57} - P_{58} - P_{59} - P_{60} - P_{61} - P_{62} - P_0$
Ruta 9: $P_0 - P_{63} - P_{64} - P_{65} - P_{66} - P_{67} - P_{68} - P_{69} - P_0$
Ruta 10: $P_0 - P_{70} - P_{71} - P_{72} - P_{73} - P_{74} - P_{75} - P_0$

Esta solución tiene un coste de 12820 que incluye una penalización de 10600 por exceso de capacidad. Al igual que el problema 3 con 50 clientes, este problema también presenta alguna dificultad a la hora de realizar la búsqueda ya que tarda más tiempo al tener más clientes. La mejor solución conseguida tiene un coste de 880,51 y está formada por las siguientes rutas:

Ruta 1: $P_0 - P_{30} - P_{48} - P_{21} - P_{36} - P_{47} - P_5 - P_{29} - P_0$
Ruta 2: $P_0 - P_{38} - P_{65} - P_{66} - P_{11} - P_{53} - P_7 - P_0$
Ruta 3: $P_0 - P_{12} - P_{72} - P_{39} - P_{26} - P_{31} - P_{10} - P_{58} - P_0$
Ruta 4: $P_0 - P_{67} - P_{34} - P_{46} - P_{52} - P_4 - P_{75} - P_0$
Ruta 5: $P_0 - P_6 - P_1 - P_{41} - P_{43} - P_{23} - P_{56} - P_{63} - P_{33} - P_0$
Ruta 6: $P_0 - P_{17} - P_{32} - P_{50} - P_{18} - P_{55} - P_{25} - P_9 - P_0$
Ruta 7: $P_0 - P_{51} - P_{24} - P_{49} - P_{16} - P_3 - P_{44} - P_{40} - P_0$
Ruta 8: $P_0 - P_{73} - P_{42} - P_{64} - P_{22} - P_{28} - P_{62} - P_2 - P_{68} - P_0$
Ruta 9: $P_0 - P_8 - P_{35} - P_{14} - P_{59} - P_{19} - P_{54} - P_{13} - P_{27} - P_0$
Ruta 10: $P_0 - P_{45} - P_{15} - P_{57} - P_{37} - P_{20} - P_{70} - P_{60} - P_{71} - P_{69} - P_{61} - P_{74} - P_0$

Al igual que en el anterior problema, este problema con 70 clientes no alcanza la solución encontrada con la búsqueda tabú granular propuesta por (Toth & Vigo, 2003) que tiene un coste de 835,26.

Conclusiones y futuras líneas de continuación

En el presente trabajo se ha descrito el problema de enrutamiento de vehículos y sus variantes. Para cada una de estas variantes se ha hecho una revisión de los diferentes métodos de resolución disponibles. Gracias a esta exposición el lector ha podido hacerse una idea de lo complejo que es el problema en sí y también del gran trabajo de investigación que hay detrás de este problema.

Además, en el trabajo se ha realizado una exposición de la técnica de búsqueda tabú para, a continuación, hacer una revisión de los métodos de búsqueda tabú disponibles para el problema de enrutamiento de vehículos. Debido a que son muchos los métodos de búsqueda tabú desarrollados para este problema, en el trabajo nos hemos centrado en la búsqueda tabú granular.

Mediante la exposición de la búsqueda tabú granular y a través de su ilustración en la resolución manual de un problema hemos sido capaces de evidenciar las múltiples ventajas que aporta este método frente a otras variantes de la búsqueda tabú. Como futura línea de continuación se propone describir en detalle e implementar cualquier otro método de búsqueda tabú aplicado al problema de enrutamiento de vehículos y comparar las ventajas y desventajas que presenta con respecto a la búsqueda tabú granular.

Respecto al programa desarrollado se puede concluir que funciona de manera adecuada y requiere poco tiempo de ejecución para problemas con pocos clientes. En cambio, si el número de clientes es grande el programa se ralentiza y, aunque se consiguen buenas soluciones, no se alcanza la mejor solución encontrada con el método de la búsqueda tabú granular propuesto en (Toth & Vigo, 2003).

Por ello se propone como línea futura de continuación optimizar la implementación del problema para conseguir que los tiempos de ejecución sean más cortos, tanto para problemas con pocos clientes como para aquellos en los que el número de clientes es mayor.

También se podrían realizar pruebas experimentales modificando algunos aspectos del código de la programación hasta que se consigan obtener los mismos resultados que los que consiguen Toth & Vigo en su implementación para los problemas de referencia.

Además para completar el método de la búsqueda tabú granular se podría implementar algún método para generar la solución inicial con la que se comienza a iterar, como por el ejemplo el método de Clarke & Wright o cualquier otra heurística constructiva disponible para el problema de enrutamiento de vehículos.

Es evidente que aún queda mucho por investigar en el mundo de la optimización y en concreto en la optimización de problemas de tipo NP-Duro, ya que, como ya se señaló, a día de hoy no se ha encontrado ningún algoritmo polinómico capaz de conseguir una solución óptima para estos problemas. Por ello, para concluir el presente trabajo se anima al lector a continuar con las futuras líneas de investigación propuestas, así como en cualquier otro ámbito del campo de la optimización.

Bibliografía

- Anily , S., & Al. (1990). One warehouse multiple retailer systems with vehicle routing costs. *Management Science , 36*, 92-114.

- Balinski, M., & Quandt, R. (1964). On an integer program for a delivery problem. *Operations Research , 12*, 300–304.

- Bean, J. (1994). Genetic algorithms and random keys for the sequencing and optimization. *Journal on Computing , 6*, 154-160.

- Bell, W., & Al. (1983). Improving the distribution of industrial gases with an on-line computerized routing and scheduling optimizer. *Interfaces , 13*, 4-23.

- Berger, J., & Barkaoui, M. (2004). A new hybrid genetic algorithm for the capacitated vehicle routing problem. *Journal of the Operational Research Society , 54*, 1254–1262.

- Berger, J., Barkaoui, M., & Bräysy, O. (2003). A route-directed hybrid genetic approach for the vehicle routing problem with time windows. *INFOR , 41*, 171-194.

- Bramel, J., & Simchi-Levi, D. (1995). A location based heuristic for general routing problems. *Operations Research , 43*, 649–660.

- Cavazzuti, M. (2013). *Optimization methods: from theory to design.* Springer.

- Christofides, N., & al. (1979). The vehicule routing problem. *Combinatorial Optimization* , 315-338.

- Christofides, N., & Eilon, S. (1969). An algorithm for the vehicule dispatching problem. *Oper. Res. Quart , 20*, 309-318.

- Clarke, G., & Wright, J. (1964). Scheduling of vehicules form a central depot to a number of delivery points. *Operations Research , 12* (4), 568-581.

- Coelho, V. (2016). An ILS-based algorithm to solve a large-scale real heterogeneous fleet VRP with multi-trips and docking constraints. *European Journal of Operational Research* , 367-376.

- Cordeau, J. F. (2007). Vehicule Routing. *Handbook in OR & MS , 14*.

- Cordeau, J., & Laporte, G. (2001). A Tabu Search Algorithm for the Site Dependent Vehicle Routing Problem with Time Windows. *INFOR , 39*, 292-298.

- Cordeau, J., Gendreau, M., & Laporte, G. (1997). A Tabu Search Heuristic for Periodic and Multi-Depot Vehicle Routing Problems. *Networks , 30*, 105–119.

- Cordeau, J., Gendreau, M., & Laporte, G. (1994). A Tabu Search Heuristic for the Vehicle Routing Problem. *Management Science , 40*, 1276–1290.

- Cordeau, J., Gendreau, M., Laporte, G., Potvin, J., & Semet, F. (2002). A Guide to Vehicle Routing Heuristics. *Journal of the Operational Research Society , 53*, 512–522.

- Cordeau, J., Laporte, G., & Mercier, A. (2001). A Unified Tabu Search Heuris- tic for Vehicle Routing Problems with Time Windows. *Journal of the Operational Research Society , 52*, 928–936.

- Cordeau, J.-F., & Laporte, G. (2002). Tabu search heuristics for the vehicule routing problem.

- Desrochers, M., & Soumis, F. (1988). A generalized permanent labeling algorithm for the shortest path problem with time windows. *INFOR , 26*, 191-212.

- Drezner, Z. (2003). A new genetic algorithm for the quadratic assignment problem. *Journal on Computing , 15*, 320–330.

- Dror, M., & Al. (1985). A computational comparison of algorithms for the inventory routing problem. *Annals of Operations Research , 4*, 3-23.

- Dror, M., & Ball, M. (1987). Inventory/routing: Reduction from an annual to a short period problem. *Naval Research Logistics Quarterly , 34*, 891-905.

- Fisher, M. (1994). Optimal Solution of vehicule routing problems using minimum k-trees. *Oper. Res. , 42*, 626-642.

- Fisher, M., & Al. (1982). Real-time scheduling of a bulk delivery fleet: Practical application of Lagrangean relaxation. *Technical report, The Wharton School, Uni- versity of Pennsylvania.*

- Fisher, M., & Jaikumar, R. (1981). A generalized assignment heuristic for the vehicle routing problem. *Networks , 11*, 109-124.

- Foster, B., & Ryan, D. (1976). An integer programming approach to the vehicle scheduling problem. *Operations Research , 27*, 367-384.

- Gallego, G., & Simchi-Levi, D. (1990). On the effectiveness of direct shipping strategy for the one- warehouse multi-retailer r-systems. *Management Science , 36*, 240–243.
- Gendreau, M., Hertz, A., & Laporte, G. (1992). New insertion and postoptimization procedures for the travelling salesman problem. *Operations Research , 40*, 1083–1094.

- Gill, P. E. (2007). George B. Dantzig and system optimization. *Science Direct*.

- Gillet, B., & Miller, L. (1974). A heuristic algorithm for the vehicle-dispatch problem. *Operations Research , 21*, 340–349.

- Golden, & al. (1998). The impact of metaheuristics on solving the vehicule routing problem: algorithm, poblem sets and computacional results. *Fleet Management and Logistics* , 33-56.

- Golden, B. (2008). *The vehicule routing problem.* New York: Springer.

- Golden, B., & Al. (1998). Metaheuristics in vehicle routing. *Fleet Management and Logistics* , 33-56.

- Golden, B., Magnanti, T., & Nguyen, H. (1977). Implementing vehicle routing algorithms. *Networks , 7*, 113-148.

- Homberger, J., & Gehring, H. (1999). Two evolutionary metaheuristics for the vehicle routing problem with time windows. *INFOR , 37*, 297-318.

- Laporte, G., & Al. (1985). Optimal routing under capacity and distance restric- tions. *Operations Research , 33*, 1050–1073.

- Leizhen Cai, S.-W. C.-W. (2013). *Algorithms and Computation.* Hong Kong, China: ISAAC.

- Letchford, A., Lysgaard, J., & Eglese, R. (2007). A branch-and-cut algorithm for the capacitated open vehicle routing problem. *Journal of the Operational Research Society , 58*, 1642–1651.

- Li, F., & Al. (2005). Very large-scale vehicle routing: New test problems, algorithms and results. *Computers & Operations Research , 32*, 1165–1179.

- Lin, S. (1965). Computer solutions of the travelling salesman problem. *Bell System Technical Journal , 44*, 2245–2269.

- Mester, D., & Bräysy, O. (2005). Active guided evolution strategies for large scale vehicle routing problem with time windows. *Computers & Operations Research , 32*, 1593–1614.

- Mingozzi, A., Christofides, N., & Toth, P. (1979). The vehicle routing problem. *Combinatorial Optimization* , 315-338.

- Mole, R., & Jameson, S. (1976). A sequential route-building algorithm employing a generalized sav- ings criterion. *Operational Research Quarterly , 27*, 503–511.

- Moscato, P., & Cotta, C. (2003). A gentle introduction to memetic algorithms. *Handbook of Metaheuristics* , 105-144.

- Naddef, D., & Rinaldi, G. (2002). Branch-and-cut algorithms for the capacitated VRP. *SIAM Monographs on Discrete Mathematics and Applications* , 53-84.

- Nelson, M. (1985). Implementation techniques for the vehicle routing problem. *Computers & Operations Research , 12*, 273–283.

- Osman, I. (1993). Metastrategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routing Problem. *Annals of Operations Research , 41*, 421–451.

- Paessens, H. (1988). The savings algorithm for the vehicle routing problem. *European Journal of Oper- ational Research , 34*, 336-344.

- Papadimitriou, C. H., & Steiglitz, K. (1998). *Combinatorial Optimization: Algorithms and Complexity.* Mineola, New York: Dover Publications.

- Potvin, J. (1996). Genetic algorithms for the traveling salesman problem. *Annals of Operations Re- search , 63*, 339–370.

- Prins, C. (2004). A simple and effective evolutionary algorithm for the vehicle routing problem. . *Computers & Operations Research , 31*, 1985–2002.

- Rechenberg, I. (1973). *Evolutionsstrategie.* Stuttgart, Germany.

– Rego, C. (1998). A Subpath Ejection Method for the Vehicle Routing Problem. *Management Science , 44*, 1447–1459.

– Rego, C. (2000). Node Ejection Chains for the Vehicle Routing Problem: Sequential and Parallel Algorithms.

– Rego, C., & Roucairol, C. (1996). A Parallel Tabu Search Algorithm Using Ejection Chains for the Vehicle Routing Problem. *Meta-Heuristics: Theory and Applications* , 661–675.

– Reimann, M., Rubio, R., & Wein, L. (1999). Heavy traffic analysis of the dynamic stochastic inventory- routing problem. *Transportation Science , 33*, 361–380.

– Renaud, J., Boctor, F., & Laporte, G. (1996). An improved petal heuristic for the vehicle routing prob- lem. *Journal of the Operational Research Society , 47*, 329-336.

– Semet, F., & Taillard, E. (1993). Solving real-life vehicle routing problems efficiently using tabu search. *Annals of Operations Research , 41*, 469–488.

– Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. *Principles and Practice of Constraint Programming* , 417–431.

– Solomon, M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research , 35*, 254–265.

– Taillard, E. (1997). A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation Science , 31*, 170-186.

– Taillard, E. (1993). Parallel Iterative Search Methods for Vehicle Routing Problem. *Networks , 23*, 661–673.

– Taillard, Y. R. (1995). Probabilistic Diversification and Intensi- fication in Local Search for Vehicle Routing. *Journal of Heuristics , 1*, 147–167.

– Tarantilis, C., & Kiranoudis, C. (2002). Bone route: Adaptive memory method for effective fleet management. *Annals of Operations Research , 115*, 227–241.

– Toth, P., & Vigo, D. (2003). The granular tabu search and its application to the vehicule-routing problem. *Journal on Computing , 15*, 333-346.

- Volgenant, A., & Jonker, R. (1983). The Symmetric Traveling Salesman Prob- lem and Edge Exchanges in Minimal 1-Tree. *European Journal of Op- erational Research , 12*, 394–403.

- Willard, J. (1989). Vehicle routing using r-optimal tabu search.

- Wren, A. (1971). Computers in Transport Planning and Operation.

- Wren, A., & Holliday, A. (1972). Computer scheduling of vehicles from one or more depots to a number of delivery points. *Operational Research Quarterly , 23*, 333–344.

- Xu, J., & Kelly, J. (1996). A Network Flow-Based Tabu Search Heuristic for the Vehicle Routing Problem. *Transportation Science , 30*, 379–393.

- Yang, X.-S. (2010). Engineering optimization. An introduction with metaheristic applications. New Jersey: Wiley.

- Youssef, S. M. (1999). *Iterative Computer Algorithms with applications in engineering.* EEUU: Los Alamitos.

- Zare-Reisabadi, E., & Hamid Mirmohammadi, S. (2015). Site dependent vehicle routing problem with soft time window: Modeling and solution approach. *Computers & Industrial Engineering , 90*, 177-185.

Anexo

Código de la programación

En el presente anexo se recopilan las distintas funciones que componen el código de la implementación de la búsqueda tabú granular aplicada al problema de enrutamiento de vehículos. Para una explicación genérica del código acuda al Capítulo 7.

En primer lugar, se encuentra el código de la función principal que llama a las funciones que componen el módulo de usuario BTGranular y, luego, se encuentran dichas funciones que componen el módulo BTGranular.

A continuación se muestra un índice paginado de la librería que contiene el código del programa:

1. btg: función principal 134
2. arcos_sol 137
3. clases 137
4. coste 139
5. demsatisf 140
6. factible 140
7. grafodisperso 142
8. interc_arcos 144
9. mejorsol 152
10. mov_iteracion 153
11. movimientos 154
12. reconstruir 165
13. rutacliente 167
14. tabu 168

1. btg: función principal

```
import numpy as np
from BTGranular import demsatisf as ds
from BTGranular import clases as cl
from BTGranular import rutacliente as rc
from BTGranular import coste as ct
from BTGranular import arcos_sol as ar
from BTGranular import movimientos as mv
from BTGranular import factible as fc
from BTGranular import interc_arcos as ia
from BTGranular import reconstruir as re
from BTGranular import tabu as tb
from BTGranular import grafodisperso as gd
from BTGranular import mejorsol as ms
from BTGranular import mov_iteracion as mi



def btg(M_costes, n_veh, n_cli, cli, n_rutas, cam, dem, cap_veh, sol_actual, 
n_iter):

    #Introducimos los parámetros del problema
    beta_in = 1
    beta_d = 1.75
    alfa_min = 1
    alfa_max = 6400
    alfa = 100
    ni = 10
    t_min= 5
    t_max= 10
    nd = 15 * n_cli
    nh = n_cli

#--------------------------------------------------------------------------------

    #Inicializamos los contadores
    cont_fact = 0
    cont_no_fact = 0
    cont_nd = 0
    cont_nh = 0

#--------------------------------------------------------------------------------

    #Determinamos los arcos que forman la solución inicial
    arcos_sol = ar.arcos_soluc(sol_actual)

    #Inicializamos la mejor solución
    mejor_sol = ms.in_mejor_sol(sol_actual)

    #Determinamos los arcos que forman la mejor solución (son los mismos que
        #los de la solución inicial)
    arcos_mejor_sol = ar.arcos_soluc(mejor_sol)

    #Calculamos el coste de los arcos de la solución inicial
    coste_arcos = ct.calc_coste(M_costes, arcos_sol)

    #Calculamos la demanda satisfecha en las rutas de la solución inicial
    DSR = ds.dem_satisf(n_rutas, dem, n_cli, cam)
```

```
    #Calculamos el coste por exceso de capacidad en la ruta inicial
    coste_cap = ct.penaliz(n_rutas, DSR, cap_veh, alfa)

    #Determinamos el coste de la solución inicial como la suma de los arcos y
         #la penalizacion por exceso de capacidad
    coste = coste_arcos + coste_cap

    #El coste de la mejor solución coincide con el coste de la solución inicial
    coste_mejor = coste

    #Inicialización de la lista tabú como una matriz de ceros
    l_tabu = np.zeros((n_cli+1, n_cli+1))

    #Calculamos el umbral de granularidad para cada valor de beta (beta_in y
    #beta_d)
    umbral_in = beta_in * (coste / (n_cli + n_veh))
    umbral_d = beta_d * (coste / (n_cli + n_veh))

#--------------------------------------------------------------------------------

    for i in range(n_iter):

    #Actualizamos o reconstruimos el grafo disperso según corresponda

        if i % (2*n_cli) == 0 and cont_nd != nd and cont_nh != nh+1:
            grafo_disp = gd.constr_gd(arcos_sol, arcos_mejor_sol, M_costes,
                                       umbral_in, 0)

        elif cont_nd == nd:
            grafo_disp = gd.constr_gd(arcos_sol, arcos_mejor_sol, M_costes,
               umbral_d, 1)
            for i in range (len(sol_actual)):
                sol_actual[i] = mejor_sol[i].copiar()
            cont_nd = 0
            cont_nh = 1

        elif cont_nh == nh+1:
            grafo_disp = gd.constr_gd(arcos_sol, arcos_mejor_sol, M_costes,
               umbral_in, 0)
            cont_nd = 0
            cont_nh = 0

        else:
            gd.act_gd(grafo_disp, arcos_insert)


    #Inicializamos una lista que permite construir la solución final en cada
        #iteracion
    # posición 0: arco que produce la mejor variación
    # posición 1: tipo de movimiento (1 para 2K, 2 para 3aK, 3 para 3bK y
        # 4 para 4K)
    # posición 2: variación conseguida con dicho arco y movimiento
    # posición 3: ruta a la que pertenece el cliente a del arco (a,b)
    # posición 4: ruta a la que pertenece el cliente b del arco (a,b)
        mejor_it = [[-1,-1], -1, 0, -1, -1]

    #Realizamos los distintos movimientos sobre los arcos del grafo disperso
        mi.mov_grafo(grafo_disp, arcos_sol, mejor_it, l_tabu, coste_cap,
               n_rutas, sol_actual, n_cli, cli, cam, M_costes, dem, cap_veh,
               alfa)
```

```
#Reconstruimos la solución actual a partir de los datos almacenados en
    #mejor_it
    coste,arcos_elim,arcos_insert,exceso = re.reconstruir_sol(mejor_it,
           sol_actual, coste, cam, M_costes, n_rutas, n_cli, dem,
           cap_veh, alfa)

#Actualizamos la penalización por exceso de la capacidad
    coste_cap=exceso

#Actualizamos los arcos que pertenecen a la solución actual
    arcos_sol = ar.arcos_soluc(sol_actual)

#Actualizamos la lista tabú
    tb.act_ltabu(l_tabu, arcos_elim, t_min, t_max)

#Actualizamos la mejor solución y los arcos que pertenecen a la
    #mejor solución
    coste_mejor, actualiza = ms.act_mejor(coste, coste_mejor, sol_actual,
           mejor_sol, mejor_it)
    arcos_mejor_sol = ar.arcos_soluc(mejor_sol)

#Actualizamos el contador de la mejor solución
    if cont_nh == 0:
        if actualiza==1:
            cont_nd = 0
        elif actualiza==0:
            cont_nd += 1

#Actualizamos contador de alfa
    if exceso==0:
        cont_fact += 1
        cont_no_fact = 0
    else:
        cont_no_fact += 1
        cont_fact = 0

#Actualizamos el valor de alfa en caso de que sea necesario
    if cont_fact == ni:
        cont_fact = 0
        if alfa_min > alfa/2:
            alfa = alfa_min
        else:
            alfa = alfa/2

    elif cont_no_fact == ni:
        cont_no_fact = 0
        if alfa_max < 2*alfa:
            alfa = alfa_max
        else:
            alfa = 2*alfa

#Actualizamos el valor del contador nh si es necesario
    if cont_nh != 0:
        cont_nh += 1

return mejor_sol, coste_mejor, mejor_it
```

2. arcos_sol

```
from BTGranular import clases as cl

def arcos_soluc(sol):

    #Función que crea una ListaEnlazada con los arcos que forman parte de
        #una solución

    #INPUTS:
    #sol: lista que contiene las ListasEnlazada de las rutas que forman
         #una solución

    #OUTPUTS:
    #arcos: ListaEnlazada con los arcos que forman una solución

    arcos=cl.ListaEnlazada()
    for i in range (len(sol)):
        p=sol[i].head
        if p != None:
            while p.next != None:
                arcos.enlazar_lista([p.dato,p.next.dato])
                p=p.next

    return arcos
```

3. clases

```
class Nodo(object):

    # la clase Nodo permite definir objetos con 3 atributos:
    #       dato: donde guardamos lo que denota el nodo
    #       next: guarda el nodo (tipo Nodo) que le sigue en la lista enlazada
    #       prev: guarda el nodo (tipo Nodo) que le antecede en la lista enlazada

    def __init__(self,dato,camino):
        # esta función inicializa el Nodo con los valores dados
        self.dato = dato
        self.next = None
        self.prev = None


class ListaEnlazada(object):

    # la clase ListaEnlazada permite definir listas doblemente enlazadas a
        #partir de objetos de la clase Nodo anteriormente definida
    # tiene dos atributos:
    #       head nodo (tipo Nodo) inicial de la lista con head.prev=None
    #       tail nodo (tipo Nodo) final de la lista con tail.next=None

    def __init__( self ) :
        # esta función inicializa la ListaEnlazada
        self.head = None
        self.tail = None
```

```python
def buscar(self,k):

    # busca un Nodo en la lista enlazada con dato=k
    # devuelve el Nodo cuyo dato es k, si exite y si no existe devuelve None

    p=self.head
    if p != None:
        while p.next != None:
            if(p.dato==k):
                return p
            p=p.next
        if(p.dato==k):
            return p
    return None



def mostrar(self):

    # imprime en orden los datos de los Nodos que están en la ListaEnlazada

    print("La lista enlazada es:")
    n_actual=self.head
    while n_actual is not None:
        print(n_actual.dato,",",)
        n_actual=n_actual.next



def enlazar_lista(self,k):

    # se añade el nodo cuyo dato es k al final de la lista

    nodo=Nodo(k,1)
    if self.head is None:
        self.head=self.tail=nodo
    else:
        nodo.prev=self.tail
        nodo.next=None
        self.tail.next=nodo

    self.tail=nodo



def copiar(self):

    #crea una copia de una ListaEnlazada

    p=self.head
    copy=ListaEnlazada()
    while p!= None:
        copy.enlazar_lista(p.dato)
        p=p.next

    return copy
```

4. coste

```
from BTGranular import clases as cl
from BTGranular import demsatisf as ds


def penaliz( n_rutas, DSR, cap_veh, alfa):

# Cálculo de la penalización por exceso de carga en los vehículos

    #INPUTS:
    # n_rutas: entero que indica el número de rutas que componen una solución
    # DSR: array que contiene en cada posición [i] la demanda satisfecha en
        #la ruta i
    # cap_veh: float que indica la capacidad máxima que pueden transportar
        #los vehículos
    # alfa: parámetro tipo float que penaliza el exceso de capacidad

    #OUTPUTS:
    # penalizacion: número (float) con la penalización por exceso de capacidad
        #para una solución

    penalizacion = 0

    for i in range(n_rutas):
        if (DSR[i] > cap_veh):
            penalizacion += DSR[i] - cap_veh

    penalizacion *= alfa

    return penalización



def calc_coste(M_costes, arcos):

# Cálculo del coste de una solución

    #INPUTS:
    # M_costes: matriz de numpy cuya componente i,j es el coste del arco (i,j)
    # arcos: ListaEnlazada de los arcos que forman parte de la solución cuyo
        #coste se quiere calcular

    #OUTPUTS:
    #coste: float que indica el coste de una solución sin tener en cuenta la
        #penalización por exceso de capacidad


    coste = 0

    p=arcos.head

    while p != None:
        #print(p.dato)
        coste += M_costes[p.dato[0],p.dato[1]]
        p = p.next

    return coste
```

5. demsatisf

```
def dem_satisf( n_rutas, dem_cli, n_cli, cam):

# Cálculo de la demanda satisfecha en las rutas

    #INPUTS:
    # n_rutas: entero que indica el número de rutas que componen una solución
    # dem_cli: array que contiene en la posición [i] la demanda del cliente i-1
    # n_cli: entero que indica el número total de clientes en el problema
    # cam: array que contiene en cada posición [i] la ruta a la que pertenece
        #cada cliente i-1

    #OUTPUTS
    # dem_ruta: array que contiene en cada posición [i] la demanda satisfecha
        #en la ruta i


    dem_ruta = np.zeros(n_rutas)

    for i in range(n_rutas):
        for j in range(n_cli):
            if cam[j]==i:
                dem_ruta[i] += dem_cli[j]

    return dem_ruta
```

6. factible

```
import numpy as np
from BTGranular import rutacliente as rc

def cli_misma_ruta (a, b, sol_actual, n_rutas):

#Función que determina si dos clientes están en la misma ruta

    #INPUTS:
    # a y b: tipo Nodo.dato son los clientes que se quiere saber si están en
        #la misma ruta
    # sol_actual: array formado por ListasEnlazadas que contiene las rutas de
        #la solución
    # n_rutas: int con el número de rutas que forman parte de la solución

    #OUTPUTS:
    # misma: int igual a 0 si los clientes están en distinta ruta e igual a 1
        #si están en la misma ruta

    misma=0

    if (a!=0 and b!=0):

        cam_a = rc.ruta_cli(a, sol_actual, n_rutas)
        cam_b = rc.ruta_cli(b, sol_actual, n_rutas)

        if cam_a == cam_b:
            misma = 1

    return misma
```

```python
def fact_3ak(a, b, sol_actual, n_rutas):

#Determinamos si el arco formado por los clientes a,b es factible o no para
        #el movimiento 3-intercambio con la opción(a)

    #INPUTS:
    # a y b: tipo Nodo.dato y son los clientes que forman parte del arco
    # sol_actual: array formado por ListasEnlazadas que contiene las rutas
        #de la solución
    # n_rutas: int con el número de rutas que forman parte de la solución

    #OUTPUTS:
    # fact: 1 si el arco es factible y 0 si no es factible

    fact = 1

    if (a==0):
        fact = 0

    cam_a = rc.ruta_cli( a, sol_actual, n_rutas )
    cli_cam_a = rc.n_cli_ruta(sol_actual[cam_a])
    if cli_cam_a <= 1:
        fact = 0

    return fact

def fact_3bk(a, b, sol_actual, n_rutas):

#Determinamos si el arco formado por los clientes a,b es factible o no para
        #el movimiento 3-intercambio con la opción (b)

    #INPUTS:
    # a y b: tipo Nodo.dato y son los clientes que forman parte del arco
    # sol_actual: array formado por ListasEnlazadas que contiene las rutas
        #de la solución
    # n_rutas: int con el número de rutas que forman parte de la solución

    #OUTPUTS:
    # fact: 1 si el arco es factible y 0 si no es factible


    fact = 1

    if (b==0):
        fact = 0

    else:
        cam_b = rc.ruta_cli(b, sol_actual, n_rutas)
        b_nodo = sol_actual[cam_b].buscar(b)
        if b_nodo.next.dato==0:
            fact = 0

    cam_b = rc.ruta_cli( b, sol_actual, n_rutas )
    cli_cam_b = rc.n_cli_ruta(sol_actual[cam_b])
    if cli_cam_b <= 2:
        fact = 0

    return fact
```

```
def fact_4k(a, b, sol_actual, n_rutas):

#Determinamos si el arco formado por los clientes a,b es factible o no para
        #el movimiento 4k

    #INPUTS:
    # a y b: tipo Nodo.dato y son los clientes que forman parte del arco
    # sol_actual: array formado por ListasEnlazadas que contiene las rutas
        #de la solución
    # n_rutas: int con el número de rutas que forman parte de la solución

    #OUTPUTS:
    #fact: 1 si el arco es factible y 0 si no es factible

    fact = 1

    if(b==0):
        fact = 0

    else:
        cam_a = rc.ruta_cli(a, sol_actual, n_rutas)
        p = sol_actual[cam_a].head
        while p.next.next != None:
            p=p.next
        if p.dato==a:
            fact = 0

    cam_a = rc.ruta_cli( a, sol_actual, n_rutas )
    cli_cam_a = rc.n_cli_ruta(sol_actual[cam_a])
    if cli_cam_a <= 2:
        fact = 0

    cam_b = rc.ruta_cli( b, sol_actual, n_rutas )
    cli_cam_b = rc.n_cli_ruta(sol_actual[cam_b])
    if cli_cam_b < 1:
        fact = 0


    return fact
```

7. grafodisperso

```
import numpy as np
from BTGranular import clases as cl


def constr_gd(l_arcos, l_arcos_mejor, M_costes, umbral, codigo):

#Función que construye el grafo disperso de la nada

    #INPUTS:
    # l_arcos: ListaEnlazada que contiene los arcos de la solución actual
    # l_arcos_mejor: ListaEnlazada que contiene los arcos de la mejor solución
        #encontrada hasta el momento
    # M_costes: matriz cuya componente i,j es el coste del arco (i,j)
    # umbral: float que representa el umbral de granularidad para construir
        #el grafo disperso
```

```
# codigo: int igual a 0 si se quieren añadir al grafo disperso los arcos
    #de la solución actual y de la mejor solución; igual a 1 si solo se
    #quieren añadir al grafo disperso los arcos de la mejor solución.

#OUTPUTS:
# gd: ListaEnlazada con los arcos que forman el grafo disperso


gd = cl.ListaEnlazada()

#Introducimos los arcos incidentes con el almacén
for i in range (M_costes.shape[0]):
    if i!=0:
        gd.enlazar_lista([0,i])

for j in range (M_costes.shape[1]):
    if j!=0:
        gd.enlazar_lista([j,0])

#Introducimos los arcos con coste menor que el umbral de granularidad
for k in range (M_costes.shape[0]):
    for l in range (M_costes.shape[1]):
        if k!=l and k!=0 and l!=0:
            if M_costes[k][l]<umbral:
                gd.enlazar_lista([k,l])

if codigo == 0: #introducimos los arcos de la sol_actual y de la mejor_sol

#Introducimos los arcos que forman parte de la solución actual
    p = l_arcos.head
    while p != None:

        if (p.dato[0] != 0 and p.dato[1] != 0 and
                   M_costes[p.dato[0],p.dato[1]]>umbral):
            gd.enlazar_lista(p.dato)
            p = p.next

        else:
            p = p.next

#Introducimos los arcos de la mejor solución
    q = l_arcos_mejor.head
    if q!= None:

        while q.next!=None:

            if (M_costes[q.dato[0],q.dato[1]]>umbral):

                if gd.buscar(q.dato)==None:
                    gd.enlazar_lista(q.dato)
                    q = q.next

                else:
                    q=q.next

            else:
                q = q.next
```

```
    elif codigo == 1: #solo introducimos los arcos de la mejor solución

    #Introducimos los arcos de la mejor solución
        q = l_arcos_mejor.head
        if q!= None:

            while q.next!=None:

                if (M_costes[q.dato[0],q.dato[1]]>umbral):

                    if gd.buscar(q.dato)==None:
                        gd.enlazar_lista(q.dato)
                        q = q.next

                    else:
                        q=q.next

                else:
                    q = q.next


    return gd

def act_gd(grafo, arcos_insert):

#Función que actualiza el grafo disperso tras cada iteración añadiendo los
        #arcos insertados en la última iteración

    #INPUTS:
    # grafo: ListaEnlazada que contiene los arcos que forman el grafo disperso
        #Este argumento se modifica al ejecutar la función de forma
        #que se obtiene el grafo actualizado
    # arcos_insert: lista que contiene los arcos que han sido insertados en la
        #última iteración

    for i in range (len(arcos_insert)):
        if grafo.buscar(arcos_insert[i])==None:
            grafo.enlazar_lista(arcos_insert[i])
```

8. interc_arcos

```
import numpy as np
from BTGranular import clases as cl
from BTGranular import factible as fc
from BTGranular import movimientos as mv
from BTGranular import rutacliente as rc


def mov_2k(a, b, n_rutas, mejor_it, coste_cap, sol_actual, n_cli, cli, cam,
M_costes, dem, cap_veh, alfa):

#Función que realiza el movimiento 2-intercambio completo para un arco de
        #clientes a y b

    #INPUTS:
    # a y b: tipo Nodo.dato y son los clientes que forman parte del arco
    # n_rutas: entero que indica el número de rutas que componen una solución
```

```
# mejor_it: lista con las características del movimiento con mejor variación
    # [0]: arco que produce la mejor variación (lista)
    # [1]: tipo de movimiento (1 si 2k, 2 si 3ak, 3 si 3bk y 4 si 4k) (int)
    # [2]: mejor variación (float)
    # [3]: número del camino del cliente a del arco (a,b)
    # [4]: número del camino del cliente b del arco (a,b)
# coste_cap: float con la penalización de la solución anterior
# sol_actual: array que con las ListaEnlazadas que forman la solución actual
# n_cli: int con el número de clientes que forman parte del problema
# cli: lista con la numeración de los clientes
# cam: lista que contiene en cada posición [i] el camino al que pertenece
    #el cliente [i+1]
# M_costes: matriz de numpy cuya componente i,j es el coste del arco (i,j)
# dem: array de numpy que contiene en cada posición [i] la demanda del
    #cliente [i+1]
# cap_veh: float con la capacidad máxima de los vehículos
# alfa: float parámetro de penalización por exceso de carga

#OUTPUTS:
# var: float con la variación calculada para un arco con el movimiento
    #2-intercambio

var=0

if(a!=0 and b!=0):
    sol_it = np.ndarray(shape=(n_rutas), dtype=(cl.ListaEnlazada))
    for i in range (n_rutas):
        sol_it[i] = sol_actual[i].copiar()

    cam_a = rc.ruta_cli(a, sol_actual, n_rutas)
    cam_b = rc.ruta_cli(b, sol_actual, n_rutas)

    var = mv.dos_interc(sol_it[cam_a], sol_it[cam_b], a, b, cam, M_costes,
     coste_cap, n_rutas, dem, n_cli, cap_veh, alfa)

    if var < mejor_it[2] or mejor_it==[[-1,-1], -1, 0, -1, -1]:
        mejor_it[0] = [a,b]
        mejor_it[1] = 1
        mejor_it[2] = var
        mejor_it[3] = cam_a
        mejor_it[4] = cam_b


    #Añadimos una segunda opción si a = 0 para considerar todas las
           #posibles combinaciones con el almacén en las demás rutas.
elif(a==0):

    cam_b = rc.ruta_cli(b, sol_actual, n_rutas)
    vector_aux = np.zeros(n_rutas)
    vector_aux[cam_b] = 1

    for i in range (n_rutas):
        if (vector_aux[i]==0):
            sol_it = np.ndarray(shape = (n_rutas),dtype= (cl.ListaEnlazada))

            for j in range (n_rutas):
                sol_it[j] = sol_actual[j].copiar()
```

```
                var = 0
                var = mv.dos_interc(sol_it[i], sol_it[cam_b], a, b, cam,
                       M_costes, coste_cap, n_rutas, dem, n_cli, cap_veh, alfa)

                if var < mejor_it[2] or mejor_it==[[-1,-1], -1, 0, -1, -1]:
                    mejor_it[0] = [a,b]
                    mejor_it[1] = 1
                    mejor_it[2] = var
                    mejor_it[3] = i
                    mejor_it[4] = cam_b



        #Añadimos una tercera opción si b = 0 para considerar todas las
        #posibles combinaciones con el almacén en las demás rutas.

    elif(b==0):

        cam_a = rc.ruta_cli(a, sol_actual, n_rutas)
        vector_aux = np.zeros(n_rutas)
        vector_aux[cam_a] = 1

        for i in range (n_rutas):
            if (vector_aux[i]==0):
                sol_it = np.ndarray(shape = (n_rutas),dtype= (cl.ListaEnlazada))

                for j in range (n_rutas):
                    sol_it[j] = sol_actual[j].copiar()

                var = 0
                var = mv.dos_interc(sol_it[cam_a], sol_it[i], a, b, cam,
                       M_costes, coste_cap, n_rutas, dem, n_cli, cap_veh, alfa)

                if var < mejor_it[2] or mejor_it==[[-1,-1], -1, 0, -1, -1]:
                    mejor_it[0] = [a,b]
                    mejor_it[1] = 2
                    mejor_it[2] = var
                    mejor_it[3] = cam_a
                    mejor_it[4] = i


    return var

def mov_3ak(a, b, n_rutas, mejor_it, coste_cap, sol_actual, n_cli, cli, cam,
M_costes, dem, cap_veh, alfa):

#Función que realiza el movimiento 3-intercambio con la opción (a) completo
        #para un arco de clientes a y b

    #INPUTS:
    #a y b: tipo Nodo.dato y son los clientes que forman parte del arco
    #n_rutas: entero que indica el número de rutas que componen una solución
    #mejor_it: lista con las características del movimiento con mejor variación
        # [0]: arco que produce la mejor variación (lista)
        # [1]: tipo de movimiento (1 si 2k, 2 si 3ak, 3 si 3bk y 4 si 4k) (int)
        # [2]: mejor variación (float)
        # [3]: número del camino del cliente a del arco (a,b)
        # [4]: número del camino del cliente b del arco (a,b)
    # coste_cap: float con la penalización de la solución anterior
    # sol_actual: array con las ListaEnlazadas que forman la solución actual
```

```
# n_cli: int con el número de clientes que forman parte del problema
# cli: lista con la numeración de los clientes
# cam: lista que contiene en cada posición [i] el camino al que pertenece
    #el cliente [i+1]
# M_costes: matriz de numpy cuya componente i,j es el coste del arco (i,j)
# dem: array de numpy que contiene en cada posición [i] la demanda del
    #cliente [i+1]
# cap_veh: float con la capacidad máxima de los vehículos
# alfa: float parámetro de penalización por exceso de carga

#OUTPUTS:
#var: float con la variación calculada para un arco con el movimiento
    #3-intercambio con la opción (b)

var=0

if (fc.fact_3ak(a, b, sol_actual, n_rutas)==1):

    if(b!=0):
        sol_it = np.ndarray(shape = (n_rutas), dtype = (cl.ListaEnlazada))

        for i in range (n_rutas):
            sol_it[i] = sol_actual[i].copiar()

        cam_a = rc.ruta_cli(a, sol_actual, n_rutas)
        cam_b = rc.ruta_cli(b, sol_actual, n_rutas)

        var = mv.tresA_interc(sol_it[cam_a], sol_it[cam_b], a, b, cam,
           M_costes, coste_cap, n_rutas, dem, n_cli, cap_veh, alfa)

        if var < mejor_it[2] or mejor_it==[[-1,-1], -1, 0, -1, -1]:
            mejor_it[0] = [a,b]
            mejor_it[1] = 2
            mejor_it[2] = var
            mejor_it[3] = cam_a
            mejor_it[4] = cam_b



    #Añadimos una segunda opción si b = 0 para considerar todas las
           #posibles combinaciones con el almacén en las demás rutas.
    else:

        cam_a = rc.ruta_cli(a, sol_actual, n_rutas)
        vector_aux = np.zeros(n_rutas)
        vector_aux[cam_a] = 1

        for i in range (n_rutas):
            if (vector_aux[i]==0):
                sol_it =np.ndarray(shape=(n_rutas),dtype=(cl.ListaEnlazada))

                for j in range (n_rutas):
                    sol_it[j] = sol_actual[j].copiar()

                var = 0
                var = mv.tresA_interc(sol_it[cam_a], sol_it[i], a, b, cam,
                   M_costes, coste_cap, n_rutas, dem, n_cli, cap_veh, alfa)
```

```
                if var < mejor_it[2] or mejor_it==[[-1,-1], -1, 0, -1, -1]:
                    mejor_it[0] = [a,b]
                    mejor_it[1] = 2
                    mejor_it[2] = var
                    mejor_it[3] = cam_a
                    mejor_it[4] = i


    return var



def mov_3bk(a, b, n_rutas, mejor_it, coste_cap, sol_actual, n_cli, cli, cam, M_c
ostes, dem, cap_veh, alfa):

#Función que realiza el movimiento 3-intercambio con la opción (b) completo
        #para un arco de clientes a y b

    #INPUTS:
    #a y b: tipo Nodo.dato y son los clientes que forman parte del arco
    #n_rutas: entero que indica el número de rutas que componen una solución
    #mejor_it: lista con las características del movimiento con mejor variación
        # [0]: arco que produce la mejor variación (lista)
        # [1]: tipo de movimiento (1 si 2k, 2 si 3ak, 3 si 3bk y 4 si 4k) (int)
        # [2]: mejor variación (float)
        # [3]: número del camino del cliente a del arco (a,b)
        # [4]: número del camino del cliente b del arco
    # coste_cap: float con la penalización de la solución anterior
    # sol_actual: array con las ListaEnlazadas que forman la solución actual
    # n_cli: int con el número de clientes que forman parte del problema
    # cli: lista con la numeración de los clientes
    # cam: lista que contiene en cada posición [i] el camino al que pertenece
        #el cliente [i+1]
    # M_costes: matriz de numpy cuya componente i,j es el coste del arco (i,j)
    # dem: array de numpy que contiene en cada posición [i] la demanda del
        #cliente [i+1]
    # cap_veh: float con la capacidad máxima de los vehí
    # alfa: float parámetro de penalización por exceso de carga

    #OUTPUTS:
    #var: float con la variación calculada para un arco con el movimiento
        #3-intercambio con la opción (b)

    var=0

    if (fc.fact_3bk(a, b, sol_actual, n_rutas)==1):

        if(a!=0):
            sol_it = np.ndarray(shape = (n_rutas), dtype = (cl.ListaEnlazada))
            for i in range (n_rutas):
                sol_it[i] = sol_actual[i].copiar()

            cam_a = rc.ruta_cli(a, sol_actual, n_rutas)
            cam_b = rc.ruta_cli(b, sol_actual, n_rutas)

            var = mv.tresB_interc(sol_it[cam_a], sol_it[cam_b], a, b, cam,
               M_costes, coste_cap,n_rutas, dem, n_cli, cap_veh, alfa)


            if var < mejor_it[2] or mejor_it==[[-1,-1], -1, 0, -1, -1]:
```

```
            mejor_it[0] = [a,b]
            mejor_it[1] = 3
            mejor_it[2] = var
            mejor_it[3] = cam_a
            mejor_it[4] = cam_b


    #Añadimos una segunda opción si a = 0 para considerar todas las
            #posibles combinaciones con el almacén en las demás rutas.
    else:
        cam_b = rc.ruta_cli(b, sol_actual, n_rutas)
        vector_aux = np.zeros(n_rutas)
        vector_aux[cam_b] = 1

        for i in range (n_rutas):

            if (vector_aux[i]==0):
                sol_it =np.ndarray(shape=(n_rutas),dtype=(cl.ListaEnlazada))

                for j in range (n_rutas):
                    sol_it[j] = sol_actual[j].copiar()

                var=0
                var = mv.tresB_interc(sol_it[i], sol_it[cam_b], a, b, cam,
                   M_costes, coste_cap, n_rutas, dem, n_cli, cap_veh, alfa)

                if var < mejor_it[2] or mejor_it==[[-1,-1], -1, 0, -1, -1]:
                    mejor_it[0] = [a,b]
                    mejor_it[1] = 3
                    mejor_it[2] = var
                    mejor_it[3] = i
                    mejor_it[4] = cam_b


    return var


def mov_4k(a, b, n_rutas, mejor_it, coste_cap, sol_actual, n_cli, cli, cam,
M_costes, dem, cap_veh, alfa):

#Función que realiza el movimiento 4-intercambio completo para un arco de
        #clientes a y b

    #INPUTS:
    #a y b: tipo Nodo.dato y son los clientes que forman parte del arco
    #n_rutas: entero que indica el número de rutas que componen una solución
    #mejor_it: lista con las características del movimiento con mejor variación
        # [0]: arco que produce la mejor variación (lista)
        # [1]: tipo de movimiento (1 si 2k, 2 si 3ak, 3 si 3bk y 4 si 4k) (int)
        # [2]: mejor variación (float)
        # [3]: número del camino del cliente a del arco (a,b)
        # [4]: número del camino del cliente b del arco
    # coste_cap: float con la penalización de la solución anterior
    # sol_actual: array con las ListaEnlazadas que forman la solución actual
    # n_cli: int con el número de clientes que forman parte del problema
    # cli: lista con la numeración de los clientes
    # cam: lista que contiene en cada posición [i] el camino al que pertenece
        #el cliente [i+1]
    # M_costes: matriz de numpy cuya componente i,j es el coste del arco (i,j)
```

```python
# dem: array de numpy que contiene en cada posición [i] la demanda del
    #cliente [i+1]
# cap_veh: float con la capacidad máxima de los vehículos
# alfa: float parámetro de penalización por exceso de carga

#OUTPUTS:
#var: float con la variación calculada para un arco con el movimiento
    #4-intercambio

var = 0

if (fc.fact_4k(a, b, sol_actual, n_rutas)==1):

    if(a!=0):
        sol_it = np.ndarray(shape = (n_rutas), dtype = (cl.ListaEnlazada))

        for i in range (n_rutas):
            sol_it[i] = sol_actual[i].copiar()

        cam_a = rc.ruta_cli(a, sol_actual, n_rutas)
        cam_b = rc.ruta_cli(b, sol_actual, n_rutas)

        var = mv.cuatro_interc(sol_it[cam_a], sol_it[cam_b], a, b, cam,
           M_costes, coste_cap, n_rutas, dem, n_cli, cap_veh, alfa)

        if (var < mejor_it[2] or mejor_it==[[-1,-1], -1, 0, -1, -1]):
            mejor_it[0] = [a,b]
            mejor_it[1] = 4
            mejor_it[2] = var
            mejor_it[3] = cam_a
            mejor_it[4] = cam_b


    #Añadimos una segunda opción si a = 0 para considerar todas las
           #posibles combinaciones con el almacén en las demás rutas.
    else:
        cam_b = rc.ruta_cli(b, sol_actual, n_rutas)
        vector_aux = np.zeros(n_rutas)
        vector_aux[cam_b] = 1

        for i in range (n_rutas):
            if (vector_aux[i]==0):
                sol_it =np.ndarray(shape=(n_rutas),dtype=(cl.ListaEnlazada))
                for j in range (n_rutas):
                    sol_it[j] = sol_actual[j].copiar()

                var = 0
                var = mv.cuatro_interc(sol_it[i],sol_it[cam_b], a, b, cam,
                   M_costes, coste_cap, n_rutas, dem, n_cli, cap_veh, alfa)

                if var < mejor_it[2] or mejor_it==[[-1,-1], -1, 0, -1, -1]:
                    mejor_it[0] = [a,b]
                    mejor_it[1] = 4
                    mejor_it[2] = var
                    mejor_it[3] = i
                    mejor_it[4] = cam_b


return var
```

```python
def mov_arco(a, b, mejor_it, coste_cap, n_rutas, sol_actual, n_cli, cli, cam,
M_costes, dem, cap_veh, alfa):

#Función que realiza todos los movimientos posibles para un arco (a,b)

    #INPUTS:
    # a y b: tipo Nodo.dato y son los clientes que forman parte del arco
    # mejor_it: lista con las características del movimiento con mejor variación
        # [0]: arco que produce la mejor variación (lista)
        # [1]: tipo de movimiento (1 si 2k, 2 si 3ak, 3 si 3bk y 4 si 4k) (int)
        # [2]: mejor variación (float)
        # [3]: número del camino del cliente a del arco (a,b)
        # [4]: número del camino del cliente b del arco
        #Este argumento se modifica y es el Output de la función
    # coste_cap: float con la penalización de la solución anterior
    # n_rutas: int con el número de rutas que forman la solución
    # sol_actual: array con las ListaEnlazadas que forman la solución actual
    # n_cli: int con el número de clientes que forman parte del problema
    # cli: lista con la numeración de los clientes
    # cam: lista que contiene en cada posición [i] el camino al que pertenece
        #el cliente [i+1]
    # M_costes: matriz de numpy cuya componente i,j es el coste del arco (i,j)
    # dem: array de numpy que contiene en cada posición [i] la demanda del
        #cliente [i+1]
    # cap_veh: float con la capacidad máxima de los vehículos
    # alfa: float parámetro de penalización por exceso de carga


    #OUTPUTS:
    #mejor_it: lista con las características del movimiento con mejor variación
        # [0]: arco que produce la mejor variación (lista)
        # [1]: tipo de movimiento (1 si 2k, 2 si 3ak, 3 si 3bk y 4 si 4k) (int)
        # [2]: mejor variación (float)
        # [3]: número del camino del cliente a del arco (a,b)
        # [4]: número del camino del cliente b del arco


    for i in range (4):

        if (i==0):
            mov_2k(a, b, n_rutas, mejor_it, coste_cap, sol_actual, n_cli, cli,
                cam, M_costes, dem, cap_veh, alfa)

        elif (i==1):
            mov_3ak(a, b, n_rutas, mejor_it, coste_cap, sol_actual, n_cli, cli,
                cam, M_costes, dem, cap_veh, alfa)

        elif (i==2):
            mov_3bk(a, b, n_rutas, mejor_it, coste_cap, sol_actual, n_cli, cli,
                cam, M_costes, dem, cap_veh, alfa)

        elif (i==3):
            mov_4k(a, b, n_rutas, mejor_it, coste_cap, sol_actual, n_cli, cli,
                cam, M_costes, dem, cap_veh, alfa)

    return mejor_it
```

9. mejorsol

```
def act_mejor(coste, coste_mejor, sol_actual, mejor_sol, mejor_it):

#Función que actualiza la mejor solución tras cada iteración

    #INPUTS:
    # coste: float que contiene el coste de la solución obtenida en la iteración
    # coste_mejor: float que contiene el coste de la mejor solución encontrada
        #hasta el momento
    # sol_actual: lista que contiene las ListasEnlazadas de las rutas que forman
        #la solución obtenida en la iteración
    # mejor_sol: lista que contiene las ListasEnlazadas de las rutas que forman
        #la solución obtenida en la iteración
        #Este argumento se modifica al ejecutar la función si la solución de la
        #iteración es mejor
    # mejor_it: lista con las características del movimiento con mejor variación
        # [0]: arco que produce la mejor variación (lista)
        # [1]: tipo de movimiento (1 si 2k, 2 si 3ak, 3 si 3bk y 4 si 4k) (int)
        # [2]: mejor variación (float)
        # [3]: número del camino del cliente a del arco (a,b)
        # [4]: número del camino del cliente b del arco

    #OUTPUTS:
    # coste_mejor: float con el coste de la mejor solución
    # actualiza: int igual a 0 si no se actualiza la mejor solución en la
        #iteración e igual a 1 en caso de que se actualice

    actualiza = 0

    if (coste < coste_mejor):
        actualiza = 1
        coste_mejor = coste
        for i in range (len(sol_actual)):
            mejor_sol[i] = sol_actual[i].copiar()
    else:
        coste_mejor

    return coste_mejor, actualiza

def in_mejor_sol(sol_actual):

    #Función que inicializa la mejor solución creando una copia de la solución
        #inicial

    #INPUTS:
    # sol_actual: lista que contiene las ListaEnlazada que forman las rutas de
        #la solución inicial

    #OUTPUTS:
    # mejor_sol: lista que contiene las ListaEnlazada que forman la ruta de la
        #mejor solución (solución inicial)

    mejor_sol = []

    for i in range (len(sol_actual)):

        mejor_sol.append(sol_actual[i].copiar())

    return mejor_sol
```

10. mov_iteracion

```
import numpy as np
from BTGranular import clases as cl
from BTGranular import interc_arcos as ia
from BTGranular import factible as fc

def mov_grafo(gd, arcos_sol_actual, mejor_it, lista_tabu, coste_cap, n_rutas,
sol_actual, n_cli, cli, cam, M_costes, dem, cap_veh, alfa):

    #Función que realiza los 4 movimientos para cada arco del grafo disperso,
        #en caso de que el arco no sea tabú ni forme parte de la solución actual

    #INPUTS:
    # gd: ListaEnlazada con los arcos que forman parte del grafo disperso
    # arcos_sol_actual: ListaEnlazada con los arcos que forman la sol_actual
    # mejor_it: lista con las características del movimiento con mejor variación
        #encontrada para una iteración
        # [0]: arco que produce la mejor variación (lista)
        # [1]: tipo de movimiento (1 si 2k, 2 si 3ak, 3 si 3bk y 4 si 4k) (int)
        # [2]: mejor variación (float)
        # [3]: número del camino del cliente a del arco (a,b)
        # [4]: número del camino del cliente b del arco
        # Este argumento se modifica al ejecutar la función
    # lista_tabu: array de numpy que contiene en cada componente [i,j] la
        #permanencia tabú del arco (i,j)
    # coste_cap: float con la penalización por exceso de carga en la iteración
        #anterior
    # n_rutas: int con el número de rutas que forman la solución
    # sol_actual: lista que contiene las ListaEnlazada con las rutas que forman
        #la solución
    # n_cli: int número de clientes que tiene el problema
    # cli: lista con la numeración de los clientes
    # cam: lista que contiene en cada posición i el camino al que pertenece el
        #cliente i+1
    # M_costes: matriz de numpy cuya componente i,j es el coste del arco (i,j)
    # dem: array que contiene en la posición i la demanda del cliente i+1
    # cap_veh: float con la capacidad máxima de los vehículos
    # alfa: parámetro de penalización por exceso de capacidad

    p = gd.head
    while p != None:
        if (p.dato[0]!=0 and p.dato[1]!=0):

            if fc.cli_misma_ruta(p.dato[0], p.dato[1], sol_actual, n_rutas)==0:

                if arcos_sol_actual.buscar(p.dato)==None :

                    if lista_tabu[p.dato[0], p.dato[1]]==0:

                        mejor_it = ia.mov_arco(p.dato[0],p.dato[1],mejor_it,
                                coste_cap, n_rutas, sol_actual, n_cli, cli, cam,
                                M_costes, dem, cap_veh, alfa)
                        p = p.next

                    else:
                        p = p.next

                else:
                    p = p.next
```

```
        else:
            p = p.next

    elif (p.dato[0]==0 or p.dato[1]==0):

        if arcos_sol_actual.buscar(p.dato)==None:

            if lista_tabu[p.dato[0], p.dato[1]]==0:

                mejor_it = ia.mov_arco(p.dato[0],p.dato[1],mejor_it,
                           coste_cap, n_rutas, sol_actual, n_cli, cli, cam,
                           M_costes, dem, cap_veh, alfa)
                p = p.next

            else:
                p = p.next

        else:
            p = p.next
```

11. movimientos

```
import numpy as np
from BTGranular import clases as cl
from BTGranular import demsatisf as ds
from BTGranular import coste as ct

def dosk(l1, l2, a, b, cam):

#Función que realiza un 2-intercambio (solo el movimiento) y retorna los arcos
#eliminados en el movimiento

    #INPUTS:
    # l1,l2: ListaEnlazada en la que se encuentra el cliente a y b
        #respectivamente
    # a,b: tipo nodo.dato que representa el arco (a,b) al que se quiere
        #aplicar el intercambio
    # cam: lista que contiene en cada posición i la ruta a la que pertenece el
        #cliente i+1

    #OUTPUTS:
    # arcos_elim: lista con los arcos que han sido eliminados en el movimiento
    # arcos_insert: lista con los arcos que han sido insertados en el movimiento

        #Determinamos el número de camino inicial de cada ruta
        cam1 = cam[l1.head.next.dato-1]
        cam2 = cam[l2.head.next.dato-1]

        #Buscamos en las rutas los nodos que tienen a y b como dato
        a_nodo = l1.buscar(a)
        b_nodo = l2.buscar(b)
        if(b==0):
            p = l2.head
            while p.next != None:
                p = p.next
            b_nodo = p

        sigma_a = a_nodo.next
```

```
        pi_b = b_nodo.prev

        #Determinamos los nodos a partir de los cuales se produce un cambio
               #de ruta
        nodo_cambio_l1 = a_nodo.next
        nodo_cambio_l2 = b_nodo

        #recomponemos el camino l1
        a_nodo.next = b_nodo
        b_nodo.prev = a_nodo

        #recomponemos el camino l2
        pi_b.next = sigma_a
        sigma_a.prev = pi_b

        #Cambiamos el número de la ruta a la que pertenecen los clientes

        if (nodo_cambio_l1 != None):
            while (nodo_cambio_l1.next != None):
                cam[nodo_cambio_l1.dato-1] = cam2
                nodo_cambio_l1 = nodo_cambio_l1.next

        if (nodo_cambio_l2 != None):
            while (nodo_cambio_l2.next != None):
                cam[nodo_cambio_l2.dato-1] = cam1
                nodo_cambio_l2 = nodo_cambio_l2.next

        c = sigma_a.dato
        d = pi_b.dato

        arcos_elim = [[a,b],[d,c],[a,c],[d,b]]
        arcos_insert = [[a,b],[d,c]]

    return arcos_elim, arcos_insert


def tresAk(l1, l2, a, b, cam):

#Función que realiza un 3-intercambio con la opción (a) (solo el movimiento) y
#retorna los arcos eliminados en el movimiento

    #INPUTS:
    # l1,l2: ListaEnlazada en la que se encuentra el cliente a y b
       #respectivamente
    # a,b: tipo nodo.dato que representa el arco (a,b) al que se quiere
       #aplicar el intercambio
    # cam: lista que contiene en cada posición i la ruta a la que pertenece el
       #cliente i+1

    #OUTPUTS:
    # arcos_elim: lista con los arcos que han sido eliminados en el movimiento
    # arcos_insert: lista con los arcos que han sido insertados en el movimiento

    #Determinamos el número de camino inicial de cada ruta
        cam1 = cam[l1.head.next.dato-1]
        cam2 = cam[l2.head.next.dato-1]

     #Buscamos en las rutas los nodos que tienen a y b como dato respectivamente
        a_nodo = l1.buscar(a)
        b_nodo = l2.buscar(b)
```

```
        if(b==0):
            p = l2.head
            while p.next != None:
                p = p.next
            b_nodo = p


        pi_a = a_nodo.prev
        pi_b = b_nodo.prev
        sigma_a = a_nodo.next
        if (b!=0):
            sigma_b = b_nodo.next
        else:
            sigma_b = l2.head.next

        #Determinamos los nodos a partir de los cuales se produce un cambio
               #de ruta
        nodo_cambio_l1 = a_nodo
        nodo_cambio_l2 = a_nodo.next

        #recomponemos el camino l1
        pi_a.next = sigma_a
        sigma_a.prev = pi_a

        #recomponemos el camino l2
        pi_b.next = a_nodo
        a_nodo.next = b_nodo
        a_nodo.prev = pi_b
        b_nodo.prev = a_nodo

        #Cambiamos el número de la ruta a la que pertenecen los clientes

        if (nodo_cambio_l1 != None):
            while (nodo_cambio_l1.next != None):
                cam[nodo_cambio_l1.dato-1] = cam2
                nodo_cambio_l1 = nodo_cambio_l1.next

        if (nodo_cambio_l2 != None):
            while (nodo_cambio_l2.next != None):
                cam[nodo_cambio_l2.dato-1] = cam1
                nodo_cambio_l2 = nodo_cambio_l2.next

        c = pi_a.dato
        d = sigma_a.dato
        e = pi_b.dato
        f = sigma_b.dato

        arcos_elim = [[c,a],[a,d],[e,b]]
        arcos_insert = [[a,b],[c,d],[e,a]]

        return arcos_elim, arcos_insert


def tresBk(l1, l2, a, b, cam):

#Función que realiza un 3-intercambio con la opción (b) (solo el movimiento) y
#retorna los arcos eliminados en el movimiento
```

```
#INPUTS:
# l1,l2: ListaEnlazada en la que se encuentra el cliente a y b
    #respectivamente
# a,b: tipo nodo.dato que representa el arco (a,b) al que se quiere
    #aplicar el intercambio
# cam: lista que contiene en cada posición i la ruta a la que pertenece el
    #cliente i+1

#OUTPUTS:
# arcos_elim: lista con los arcos que han sido eliminados en el movimiento
# arcos_insert: lista con los arcos que han sido insertados en el movimiento

    #Determinamos el número de camino inicial de cada ruta
    cam1 = cam[l1.head.next.dato-1]
    cam2 = cam[l2.head.next.dato-1]

    #Buscamos en las rutas los nodos que tienen a y b como dato
    a_nodo = l1.buscar(a)
    b_nodo = l2.buscar(b)

    if(a==0):
        p = l2.head
        while p.next.next != None:
            p = p.next
        pi_a = p
    else:
        pi_a = a_nodo.prev

    pi_b = b_nodo.prev
    sigma_a = a_nodo.next
    sigma_b = b_nodo.next
    sigma_sigma_b = sigma_b.next

    #Determinamos los nodos a partir de los cuales se produce un cambio
            #de ruta
    nodo_cambio_l1 = sigma_sigma_b
    nodo_cambio_l2 = b_nodo

    #recomponemos el camino l1
    pi_b.next = sigma_sigma_b
    sigma_sigma_b.prev = pi_b

    #recomponemos el camino l2
    a_nodo.next = b_nodo
    b_nodo.prev = a_nodo
    sigma_b.next = sigma_a
    sigma_a.prev = sigma_b

    #Cambiamos el número de la ruta a la que pertenecen los clientes

    if (nodo_cambio_l1 != None):
        while (nodo_cambio_l1.next != None):
            cam[nodo_cambio_l1.dato-1] = cam2
            nodo_cambio_l1 = nodo_cambio_l1.next

    if (nodo_cambio_l2 != None):
        while (nodo_cambio_l2.next != None):
            cam[nodo_cambio_l2.dato-1] = cam1
            nodo_cambio_l2 = nodo_cambio_l2.next
```

```
        c = pi_b.dato
        d = sigma_b.dato
        e = sigma_sigma_b.dato
        f = pi_a.dato
        g = sigma_a.dato

        arcos_elim = [[c,b],[d,e],[a,g]]
        arcos_insert = [[a,b],[c,e],[d,g]]

        return arcos_elim, arcos_insert


def cuatrok(l1, l2, a, b, cam):

#Función que realiza un 4-intercambio (solo el movimiento) y retorna los arcos
#eliminados en el movimiento

    #INPUTS:
    # l1,l2: ListaEnlazada en la que se encuentra el cliente a y b
        #respectivamente
    # a,b: tipo nodo.dato que representa el arco (a,b) al que se quiere
        #aplicar el intercambio
    # cam: lista que contiene en cada posición i la ruta a la que pertenece el
        #cliente i+1

    #OUTPUTS:
    # arcos_elim: lista con los arcos que han sido eliminados en el movimiento
    # arcos_insert: lista con los arcos que han sido insertados en el movimiento

        #Determinamos el número de camino inicial de cada ruta
        cam1 = cam[l1.head.next.dato-1]
        cam2 = cam[l2.head.next.dato-1]

        #Buscamos en las rutas los nodos que tienen a y b como dato
        a_nodo = l1.buscar(a)
        b_nodo = l2.buscar(b)

        sigma_a = a_nodo.next
        if sigma_a.next==None:
            sigma_sigma_a=l1.head.next
        else:
            sigma_sigma_a = sigma_a.next
        pi_b = b_nodo.prev
        sigma_b = b_nodo.next


        #Determinamos los nodos a partir de los cuales se produce un cambio
                #de ruta
        nodo_cambio_l1 = a_nodo.next
        nodo_cambio_l2 = b_nodo

        #recomponemos el camino l1
        a_nodo.next = b_nodo
        b_nodo.prev = a_nodo
        b_nodo.next = sigma_sigma_a
        sigma_sigma_a.prev = b_nodo

        #recomponemos el camino l2
        pi_b.next = sigma_a
        sigma_a.prev = pi_b
```

```
        sigma_a.next = sigma_b
        sigma_b.prev = sigma_a

        #Cambiamos el número de la ruta a la que pertenecen los clientes

        if (nodo_cambio_l1 != None):
            while (nodo_cambio_l1.next != None):
                cam[nodo_cambio_l1.dato-1] = cam2
                nodo_cambio_l1 = nodo_cambio_l1.next

        if (nodo_cambio_l2 != None):
            while (nodo_cambio_l2.next != None):
                cam[nodo_cambio_l2.dato-1] = cam1
                nodo_cambio_l2 = nodo_cambio_l2.next

        c = sigma_a.dato
        d = sigma_sigma_a.dato
        e = pi_b.dato
        f = sigma_b.dato

        arcos_elim=[[a,c],[c,d],[e,b],[b,f]]
        arcos_insert = [[a,b],[e,c],[c,f],[b,d]]


        return arcos_elim, arcos_insert

def dos_interc( l1, l2, a, b, cam, M_costes, penaliz_ant, n_rutas, dem, n_cli,
cap_veh, alfa):

# Función que realiza el 2-intercambio para dos clientes a y b

    #INPUTS:
    # l1 y l2 son del tipo ListaEnlazada y son la lista en la que están los
        #clientes a y b respectivamente
    # a y b: son del tipo Nodo.dato y son los clientes que forman parte del
        #arco al que se aplica el intercambio
    # cam: lista que contiene en cada posición [i] la ruta a la que pertenece
        #cada cliente i-1
    # M_costes: matriz cuya componente i,j es el coste del arco (i,j)
    # penaliz_ant: float que contiene la penalización por exceso de capacidad
        #de la solución de la iteración anterior
    # n_rutas: int con el número de rutas que forman la solución
    # dem: array que contiene en cada posición i la demanda del cliente i+1
    # n_cli: int número de clientes que tiene el problema
    # cap_veh: float que indica la capacidad máxima de los vehículos
    # alfa: float parámetro de penalización del exceso de carga en las rutas

    #OUTPUTS:
    # var: float con la variación total del coste de la solución

        #Creamos copia del vector camino para que no se modifique el original
        cam_aux = np.copy(cam)

        #Determinamos el número de camino inicial de cada ruta
        cam1 = cam_aux[l1.head.next.dato-1]
        cam2 = cam_aux[l2.head.next.dato-1]

        #Buscamos en las rutas los nodos que tienen a y b como dato
        a_nodo = l1.buscar(a)
        b_nodo = l2.buscar(b)
```

```
        if(b==0):
            p = l2.head
            while p.next != None:
                p = p.next
            b_nodo = p

        sigma_a = a_nodo.next
        pi_b = b_nodo.prev


        #Determinamos los nodos a partir de los cuales se produce un cambio
        #de ruta
        nodo_cambio_l1 = a_nodo.next
        nodo_cambio_l2 = b_nodo

        #recomponemos el camino l1
        a_nodo.next = b_nodo
        b_nodo.prev = a_nodo

        #recomponemos el camino l2
        pi_b.next = sigma_a
        sigma_a.prev = pi_b


        #Cambiamos el número de la ruta a la que pertenecen los clientes

        if (nodo_cambio_l1 != None):
            while (nodo_cambio_l1.next != None):
                cam_aux[nodo_cambio_l1.dato-1] = cam2
                nodo_cambio_l1 = nodo_cambio_l1.next

        if (nodo_cambio_l2 != None):
            while (nodo_cambio_l2.next != None):
                cam_aux[nodo_cambio_l2.dato-1] = cam1
                nodo_cambio_l2 = nodo_cambio_l2.next

        #Cálculo de la DSR
        DSR_mov = ds.dem_satisf(n_rutas, dem, n_cli, cam_aux)

        #Cálculo de la penalizacion por exceso de capacidad
        coste_exceso = ct.penaliz(n_rutas, DSR_mov, cap_veh, alfa)

        #Cálculo de la variación en el coste debido al intercambio de arcos
        c = sigma_a.dato
        d = pi_b.dato

        coste_arcos = M_costes[a,b] + M_costes[d,c] - M_costes[a,c] –
               M_costes[d,b]

        #Cálculo de la variación total con respecto a la solución actual
        var = coste_arcos + coste_exceso - penaliz_ant

        return var

def tresA_interc(l1, l2, a, b, cam, M_costes, penaliz_ant, n_rutas, dem, n_cli,
cap_veh, alfa):

# Función que realiza el 3-intercambio con la opción (a) para dos clientes a y b
```

```
#INPUTS:
# l1 y l2 son del tipo ListaEnlazada y son la lista en la que están los
    #clientes a y b respectivamente
# a y b: son del tipo Nodo.dato y son los clientes que forman parte del
    #arco al que se aplica el intercambio
# cam: lista que contiene en cada posición [i] la ruta a la que pertenece
    #cada cliente i-1
# M_costes: matriz cuya componente i,j es el coste del arco (i,j)
# penaliz_ant: float que contiene la penalización por exceso de capacidad
    #de la solución de la iteración anterior
# n_rutas: int con el número de rutas que forman la solución
# dem: array que contiene en cada posición i la demanda del cliente i+1
# n_cli: int número de clientes que tiene el problema
# cap_veh: float que indica la capacidad máxima de los vehículos
# alfa: float parámetro de penalización del exceso de carga en las rutas

#OUTPUTS:
# var: float con la variación total del coste de la solución

    #Creamos copia del vector camino para que no se modifique el original
    cam_aux = np.copy(cam)

    #Determinamos el número de camino inicial de cada ruta
    cam1 = cam[l1.head.next.dato-1]
    cam2 = cam[l2.head.next.dato-1]

    a_nodo = l1.buscar(a)
    b_nodo = l2.buscar(b)
    if(b==0):
        p = l2.head
        while p.next != None:
            p = p.next
        b_nodo = p

    pi_a = a_nodo.prev
    pi_b = b_nodo.prev
    sigma_a = a_nodo.next
    if (b!=0):
        sigma_b = b_nodo.next
    else:
        sigma_b = l2.head.next

    #Determinamos los nodos a partir de los cuales se produce un cambio
           #de ruta
    nodo_cambio_l1 = a_nodo
    nodo_cambio_l2 = a_nodo.next

    #recomponemos el camino l1
    pi_a.next = sigma_a
    sigma_a.prev = pi_a

    #recomponemos el camino l2

    pi_b.next = a_nodo
    a_nodo.next = b_nodo
    a_nodo.prev = pi_b
    b_nodo.prev = a_nodo
```

```
        #Cambiamos el número de la ruta a la que pertenecen los clientes

        if (nodo_cambio_l1 != None):
            while (nodo_cambio_l1.next != None):
                cam_aux[nodo_cambio_l1.dato-1] = cam2
                nodo_cambio_l1 = nodo_cambio_l1.next

        if (nodo_cambio_l2 != None):
            while (nodo_cambio_l2.next != None):
                cam_aux[nodo_cambio_l2.dato-1] = cam1
                nodo_cambio_l2 = nodo_cambio_l2.next

        #Cálculo de la DSR

        DSR_mov = ds.dem_satisf(n_rutas, dem, n_cli, cam_aux)

        #Cálculo de la penalizacion por exceso de capacidad
        coste_exceso = ct.penaliz(n_rutas, DSR_mov, cap_veh, alfa)

        #Cálculo de la variación en el coste debido al intercambio de arcos
        c = pi_a.dato
        d = sigma_a.dato
        e = pi_b.dato
        f = sigma_b.dato

        coste_arcos = M_costes[a,b] + M_costes[c,d] + M_costes[e,a] –
               M_costes[c,a] - M_costes[a,d] - M_costes[e,b]

        #Cálculo de la variación total con respecto a la solución actual
        var = coste_arcos + coste_exceso - penaliz_ant

        return var

def tresB_interc(l1, l2, a, b, cam, M_costes, penaliz_ant, n_rutas, dem, n_cli,
cap_veh, alfa):

# Función que realiza el 3-intercambio con la opción (b) para dos clientes a y b

    #INPUTS:
    # l1 y l2 son del tipo ListaEnlazada y son la lista en la que están los
        #clientes a y b respectivamente
    # a y b: son del tipo Nodo.dato y son los clientes que forman parte del
        #arco al que se aplica el intercambio
    # cam: lista que contiene en cada posición [i] la ruta a la que pertenece
        #cada cliente i-1
    # M_costes: matriz cuya componente i,j es el coste del arco (i,j)
    # penaliz_ant: float que contiene la penalización por exceso de capacidad
        #de la solución de la iteración anterior
    # n_rutas: int con el número de rutas que forman la solución
    # dem: array que contiene en cada posición i la demanda del cliente i+1
    # n_cli: int número de clientes que tiene el problema
    # cap_veh: float que indica la capacidad máxima de los vehículos
    # alfa: float parámetro de penalización del exceso de carga en las rutas

    #OUTPUTS:
    # var: float con la variación total del coste de la solución

        #Creamos copia del vector camino para que no se modifique el original
        cam_aux = np.copy(cam)
```

```python
#Determinamos el número de camino inicial de cada ruta
cam1 = cam[l1.head.next.dato-1]
cam2 = cam[l2.head.next.dato-1]

a_nodo = l1.buscar(a)
b_nodo = l2.buscar(b)

if(a==0):
    p = l2.head
    while p.next.next != None:
        p = p.next
    pi_a = p
else:
    pi_a = a_nodo.prev

pi_b = b_nodo.prev
sigma_a = a_nodo.next
sigma_b = b_nodo.next
sigma_sigma_b = sigma_b.next

#Determinamos los nodos a partir de los cuales se produce un cambio
#de ruta
nodo_cambio_l1 = sigma_sigma_b
nodo_cambio_l2 = b_nodo

#recomponemos el camino l1
pi_b.next = sigma_sigma_b
sigma_sigma_b.prev = pi_b

#recomponemos el camino l2
a_nodo.next = b_nodo
b_nodo.prev = a_nodo
sigma_b.next = sigma_a
sigma_a.prev = sigma_b


#Cambiamos el número de la ruta a la que pertenecen los clientes

if (nodo_cambio_l1 != None):
    while (nodo_cambio_l1.next != None):
        cam_aux[nodo_cambio_l1.dato-1] = cam2
        nodo_cambio_l1 = nodo_cambio_l1.next

if (nodo_cambio_l2 != None):
    while (nodo_cambio_l2.next != None):
        cam_aux[nodo_cambio_l2.dato-1] = cam1
        nodo_cambio_l2 = nodo_cambio_l2.next

#Cálculo de la DSR
DSR_mov = ds.dem_satisf(n_rutas, dem, n_cli, cam_aux)

#Cálculo de la penalizacion por exceso de capacidad
coste_exceso = ct.penaliz(n_rutas, DSR_mov, cap_veh, alfa)

#Cálculo de la variación en el coste debido al intercambio de arcos
c = pi_b.dato
d = sigma_b.dato
e = sigma_sigma_b.dato
f = pi_a.dato
g = sigma_a.dato
```

```
        coste_arcos = M_costes[a,b] + M_costes[c,e] + M_costes[d,g] —
               M_costes[c,b] - M_costes[d,e] - M_costes[a,g]

        #Cálculo de la variación total con respecto a la solución actual
        var = coste_arcos + coste_exceso - penaliz_ant

        return var


def cuatro_interc(l1, l2, a, b, cam, M_costes, penaliz_ant, n_rutas, dem, n_cli,
cap_veh, alfa):

# Función que realiza el 4-intercambio para dos clientes a y b

    #INPUTS:
    # l1 y l2 son del tipo ListaEnlazada y son la lista en la que están los
        #clientes a y b respectivamente
    # a y b: son del tipo Nodo.dato y son los clientes que forman parte del
        #arco al que se aplica el intercambio
    # cam: lista que contiene en cada posición [i] la ruta a la que pertenece
        #cada cliente i-1
    # M_costes: matriz cuya componente i,j es el coste del arco (i,j)
    # penaliz_ant: float que contiene la penalización por exceso de capacidad
        #de la solución de la iteración anterior
    # n_rutas: int con el número de rutas que forman la solución
    # dem: array que contiene en cada posición i la demanda del cliente i+1
    # n_cli: int número de clientes que tiene el problema
    # cap_veh: float que indica la capacidad máxima de los vehículos
    # alfa: float parámetro de penalización del exceso de carga en las rutas

    #OUTPUTS:
    # var: float con la variación total del coste de la solución

        #Creamos copia del vector camino para que no se modifique el original
        cam_aux = np.copy(cam)

        #Determinamos el número de camino inicial de cada ruta
        cam1 = cam[l1.head.next.dato-1]
        cam2 = cam[l2.head.next.dato-1]

        a_nodo = l1.buscar(a)
        b_nodo = l2.buscar(b)

        sigma_a = a_nodo.next
        if sigma_a.next==None:
            sigma_sigma_a=l1.head.next
        else:
            sigma_sigma_a = sigma_a.next
        pi_b = b_nodo.prev
        sigma_b = b_nodo.next

        #Determinamos los nodos a partir de los cuales se produce un cambio
               #de ruta
        nodo_cambio_l1 = a_nodo.next
        nodo_cambio_l2 = b_nodo

        #recomponemos el camino l1
        a_nodo.next = b_nodo
        b_nodo.prev = a_nodo
```

```
        b_nodo.next = sigma_sigma_a
        sigma_sigma_a.prev = b_nodo

        #recomponemos el camino l2
        pi_b.next = sigma_a
        sigma_a.prev = pi_b
        sigma_a.next = sigma_b
        sigma_b.prev = sigma_a

        #Cambiamos el número de la ruta a la que pertenecen los clientes

        if (nodo_cambio_l1 != None):
            while (nodo_cambio_l1.next != None):
                cam_aux[nodo_cambio_l1.dato-1] = cam2
                nodo_cambio_l1 = nodo_cambio_l1.next

        if (nodo_cambio_l2 != None):
            while (nodo_cambio_l2.next != None):
                cam_aux[nodo_cambio_l2.dato-1] = cam1
                nodo_cambio_l2 = nodo_cambio_l2.next

        #Cálculo de la DSR
        DSR_mov = ds.dem_satisf(n_rutas, dem, n_cli, cam_aux)

        #Cálculo de la penalizacion por exceso de capacidad
        coste_exceso = ct.penaliz(n_rutas, DSR_mov, cap_veh, alfa)

        #Cálculo de la variación en el coste debido al intercambio de arcos
        c = sigma_a.dato
        d = sigma_sigma_a.dato
        e = pi_b.dato
        f = sigma_b.dato

        coste_arcos = M_costes[a,b] + M_costes[e,c] + M_costes[c,f] +
               M_costes[b,d] - M_costes[a,c] - M_costes[c,d] –
               M_costes[e,b] - M_costes[b,f]


        #Cálculo de la variación total con respecto a la solución actual
        var = coste_arcos + coste_exceso - penaliz_ant

        return var
```

12. reconstruir

```
from BTGranular import movimientos as mv
from BTGranular import coste as ct
from BTGranular import arcos_sol as ar
from BTGranular import demsatisf as ds

def reconstruir_sol(sol_it, sol_actual, coste, cam, M_costes, n_rutas, n_cli,
dem_cli, cap_veh, alfa):

#Función que reconstruye la solución actual tras cada iteración
```

```
#INPUTS:
# sol_it: lista que contiene las características del movimiento con mejor
    #variación encontrada para una iteración
    # [0]: arco que produce la mejor variación (lista)
    # [1]: tipo de movimiento (1 si 2k, 2 si 3ak, 3 si 3bk y 4 si 4k) (int)
    # [2]: mejor variación (float)
    # [3]: número del camino del cliente a del arco (a,b)
    # [4]: número del camino del cliente b del arco
# sol_actual: lista que contiene las ListaEnlazadas con las rutas que
    #forman la solución actual
   # Este argumento se modifica y toma las rutas de la solución obtenida
# coste: float con el coste de la solución actual
# cam: lista que contiene en cada posición [i] la ruta a la que pertenece
    #cada cliente i-1
# M_costes: matriz cuya componente i,j es el coste del arco (i,j)
# n_rutas: int con el número de rutas que forman la solución
# n_cli: int número de clientes que tiene el problema
# dem_cli: array que contiene en cada posición i la demanda del cliente i+1
# cap_veh: float que indica la capacidad máxima de los vehículos
# alfa: float parámetro de penalización del exceso de carga en las rutas


#OUTPUTS:
# coste: float con el nuevo coste de la solución obtenida en la iteración
# arcos_elim: lista con los arcos que han sido eliminados en el movimiento
# arcos_elim: lista con los arcos que han sido insertados en el movimiento

if sol_it[1]==1:
    arcos_eliminados,arcos_insertados = mv.dosk(sol_actual[sol_it[3]],
           sol_actual[sol_it[4]],sol_it[0][0],sol_it[0][1],cam)

elif sol_it[1]==2:
    arcos_eliminados,arcos_insertados = mv.tresAk(sol_actual[sol_it[3]],
           sol_actual[sol_it[4]],sol_it[0][0],sol_it[0][1],cam)

elif sol_it[1]==3:
    arcos_eliminados,arcos_insertados = mv.tresBk(sol_actual[sol_it[3]],
           sol_actual[sol_it[4]],sol_it[0][0],sol_it[0][1],cam)

elif sol_it[1]==4:
    arcos_eliminados,arcos_insertados = mv.cuatrok(sol_actual[sol_it[3]],
           sol_actual[sol_it[4]],sol_it[0][0],sol_it[0][1],cam)


arcos_sol_actual = ar.arcos_soluc(sol_actual)
coste_arcos = ct.calc_coste(M_costes, arcos_sol_actual)
demanda = ds.dem_satisf(n_rutas, dem_cli, n_cli, cam)
exceso = ct.penaliz( n_rutas, demanda, cap_veh, alfa)


coste =  coste_arcos + exceso



return coste, arcos_eliminados, arcos_insertados, exceso
```

13. rutacliente

```
from BTGranular import clases as cl


def ruta_cli( k, rutas, n_rutas ):

# Determinar la ruta a la que pertenece un cliente

    #INPUTS:
    # k: número entero del cliente del que se quiere encontrar la ruta a la
        #que pertenece
    # rutas: array que contiene las listas enlazadas que forman la solución en
        #la que busca la ruta del cliente
    # n_rutas: entero que indica el número de rutas que componen la solución

    #OUTPUTS:
    # i: número del camino al que pertenece el cliente k

    for i in range (n_rutas):

        if rutas[i].buscar(k) != None:
            return i

    return i



def n_cli_ruta(l1):

# Determinar el número de clientes que hay en una ruta, sin tener en cuenta
#los nodos almacén

    #INPUTS:
    # l1: ListaEnlazada de la que se quiere calcular el número de clientes
        #que contiene

    #OUTPUTS:
    # n_cli: entero que indica el número de clientes que pertenecen a l1

    n_cli = 0

    p = l1.head
    while p != None:

        n_cli += 1
        p = p.next

    n_cli -= 2

    return n_cli
```

14. tabu

```
import numpy as np
import random


def act_ltabu(l_tabu, arcos_elim, tmin, tmax):

#Función que actualiza la lista tabú

    #INPUTS:
    # l_tabu: matriz de numpy que contiene en cada componente [i][j] la
        #permanencia tabú del arco (i,j)
        #Este argumento se modifica al ejecutar la función
    # arcos_elim: lista que contiene los arcos que se han eliminado en una
        #iteración
    # tmin,tmax: int que definen el rango entre el cual se genera aleatoriamente
       #la permanencia tabú (int)

    for i in range (l_tabu.shape[0]):
        for j in range (l_tabu.shape[1]):
            if (l_tabu[i][j] !=0):
                l_tabu[i][j]-=1

    for k in range (len(arcos_elim)):
        l_tabu[arcos_elim[k][0],arcos_elim[k][1]] = random.randint(tmin,tmax)
```