Vehicle Routing Problem con objetivo de distancia total recorrida

Cristian Abrante Dorta
Ángel Igareta Herráiz
Daute Rodríguez Rodríguez
Carlos Domínguez García
Alberto González Álvarez

Universidad de La Laguna
Departamento de Ingeniería Informática y de Sistemas

Preprint submitted to Elsevier *2 de junio de 2018*

Índice

1. Introducción

La optimización es un campo relativamente joven dentro de las matemáticas, pero que hoy en día ha cobrado una especial relevancia, puesto que nos permite obtener soluciones a problemas muy complejos.

Las raíces históricas de la optimización subyacen en ciertos problemas económicos: la planificación y gestión de operaciones y el uso eficiente de los recursos. A partir de estos objetivos fundamentales, comenzaron a desarrollarse técnicas mucho más complejas en diversas áreas como informática, gestión logística, telecomunicaciones, secuenciación de ADN, étc [10]. La gran trascendencia e importancia que suelen tener este tipo de problemas se debe a que: *Contiene los dos elementos que hacen atractivo un problema a los matemáticos: planteamiento sencillo y dificultad de resolución* [8].

De manera natural, los problemas de optimización pueden dividirse en dos grandes categorías. La primera categoría se refiere a los problemas que involucran variables continuas, como buscar el mínimo de la función:

$$f : \mathbb{R} \to \mathbb{R} \quad f(x) = x^2$$

Nosotros no nos centraremos en la resolución de este tipo de problemas sino que exploraremos la segunda categoría, en las cuales las variables son discretas, también conocida como **optimización combinatoria**.

La optimización combinatoria tratará de minimizar o maximizar una determinada función objetivo satisfaciendo un conjunto dado de restricciones. Formalmente un problema (P) de optimización combinatoria será una *t-upla*

$$P = (S, f, \Omega)$$

Donde:

- S: Es el conjunto de todas las soluciones posibles que puede tener nuestro problema. Se trata de un conjunto finito y numerable, aunque la gran cantidad de combinaciones posibles hacen que sea muy costoso computacionalmente explorarlo de manera secuencial.
- f: Es la función objetivo. Para cada solución de S proporciona un valor de la bondad de la solución.

- Ω: Es el conjunto de las restricciones que deben cumplir las soluciones para ser factibles.

A partir de estos elementos podemos definir el concepto de óptimo global (s^*). Es decir, el elemento del conjunto de soluciones cuyo valor de la función objetivo es mejor que el valor de la función objetivo de cualquier otra solución.

$$\forall s \in S \quad f(s^*) \geq f(s)$$

Siempre teniendo en cuenta que las soluciones deben verificar las restricciones presentes en Ω.

Una posible solución a un problema de optimización combinatoria podría consistir en explorar todos los elementos del conjunto de soluciones (S) y evaluar sus correspondientes valores objetivo. Ahora bien, esto suele ser computacionalmente inviable incluso para problemas de un tamaño relativamente pequeño.

Por ello, necesitaremos hacer una *exploración inteligente* del espacio de soluciones, utilizando diferentes técnicas denominadas **metaheurísticas**. En nuestro caso, trataremos de resolver uno de los problemas clásicos en el terreno de la optimización combinatoria: el problema del enrutamiento de vehículos [1].

El problema del enrutamiento de vehículos (*Vehicle Routing Problem*) tiene interés en muchas áreas diferentes, y es crucial en la actividad diaria de muchas empresas [7] que se enfrentan a problemas de reparto de mercancía para un conjunto de clientes.

2. Descripción del Problema

2.1. Descripción informal

Tal y como comentamos, el problema de enrutamiento de vehículos (*Vehicle Routing Problem*) trata de dar solución a un problema muy común en la actividad empresarial. Informalmente se describe de la siguiente forma: tenemos un conjunto de clientes a los cuales tendremos que abastecer, cada uno con una determinada demanda. Para poder abastecerlos, contaremos con una

serie de coches que tendrán una cierta capacidad de carga. El problema consiste en determinar las rutas que deberán llevar a cabo cada uno de los coches de tal forma que se minimice la distancia total recorrida por los vehículos. De esta forma proporcionaremos un enfoque de capacidad al problema (*Capacity Vehicle Routing Problem*)

En general, este problema de optimización puede llegar a tener una gran cantidad de restricciones. Por ejemplo, restricciones relacionadas con ventanas de tiempo, con diferentes depósitos de los que pueden salir los vehículos o con el tráfico de las rutas que se tomarán.

Nuestro enfoque será más clásico, ya que las restricciones que deberemos cumplir serán; por un lado que la demanda de todos los clientes esté satisfecha, y por otro que las rutas tomadas por cada vehículo no excedan la capacidad del mismo. Además supondremos que cada uno de los vehículos tendrá que salir y llegar al depósito.

2.2. Descripción formal

Tras haber dado una descripción informal, estaremos en disposición de formalizar el *Capacity Vehicle Routing Problem.*

En primer lugar contaremos con un depósito (D), el cual estará situado en una posición en un plano bidimensional:

$$(D_x, D_y) \in \mathbb{R}^2$$

También, contaremos con un conjunto finito de n clientes (V):

$$V = \{v_1, v_2, \ldots, v_n\} \quad |V| = n$$

Donde cada $v_1, v_2, \ldots, v_n$ posee una demanda, que se considera como un número entero.

$$\forall v_i \in V \quad \exists d_i \in \mathbb{N}$$

Y los cuales también tendrán asociada una posición en un plano bidimensional, que representará su localización geográfica:

$$\forall v_i \in V \quad \exists (v_{ix}, v_{iy}) \in \mathbb{R}^2$$

Es necesario también contar con una función (δ) que nos mida la distancia entre dos clientes o entre un cliente y el depósito. Por simplicidad hemos considerado la distancia euclídea como la medida utilizada, la cual se define de esta forma:

$$\delta : \{V \cup D\}^2 \to \mathbb{R}$$

$$\delta(v_i, v_j) = \sqrt{(v_{jx} - v_{ix})^2 + (v_{jy} - v_{iy})^2}$$

Cabe destacar que esta función debe estar definida tanto entre clientes como entre un cliente y el depósito.

Por otra parte, contaremos con un conjunto de m vehículos (K) para satisfacer las demandas:

$$K = \{k_1, k_2, \ldots, k_m\} \quad |K| = m$$

Cada uno de los cuales tendrá una capacidad determinada que también se tratará de un número entero:

$$\forall k_i \in K \quad \exists c_i \in \mathbb{N}$$

En este punto cabe destacar, que las demandas de cada uno de los clientes deben poder ser satisfechas por un solo coche, es decir, que no sean necesarios dos o más coches para satisfacer una demanda.

3. Representación y codificación de soluciones

3.1. Descripción formal de la solución

Tal y como comentamos previamente, todo problema de optimización combinatoria posee un espacio de soluciones. En nuestro caso, cada una de las soluciones estará definida de la siguiente forma:

$$\forall s \in S, \quad s = \{R_1, R_2, \ldots, R_r\} \quad |s| = r$$

Cada una de las soluciones (s) es un conjunto de r rutas, donde el número de rutas que tenemos no sobrepasará el número de coches del que disponemos ($r \leq m$). A su vez cada una de las rutas será un conjunto de clientes donde el orden de visita de dicho cliente está determinado por su posición dentro de dicha ruta:

$$\forall R_i \in s \quad R_i =< v_{it}, \ldots, v_{it'} > / R_i \subseteq V$$

Cada una de las r rutas se llevará a cabo por un vehículo diferente. De esta forma, la ruta *i-ésima* estará realizada por el vehículo *i-ésimo*.

3.2. Codificación de la solución

La codificación de las soluciones debe satisfacer diferentes criterios. En concreto, la codificación diseñada debe ser eficiente, es decir que utilice el mínimo de recursos posibles, y también que permita que se pueda obtener de manera sencilla una solución a partir de otra.

Por ello, nos decantamos por utilizar un vector de números enteros. De esta forma, una ruta queda especificada por una secuencia de enteros, cada uno de los cuales representa el identificador único de un cliente, y cuyo orden de visita queda especificado por el orden en el que aparecen en la secuencia. Para diferenciar entre diferentes rutas hemos usado un separador, es decir, un valor que no forma parte del conjunto de identificadores de los clientes, que en nuestro caso será el valor **-1**.

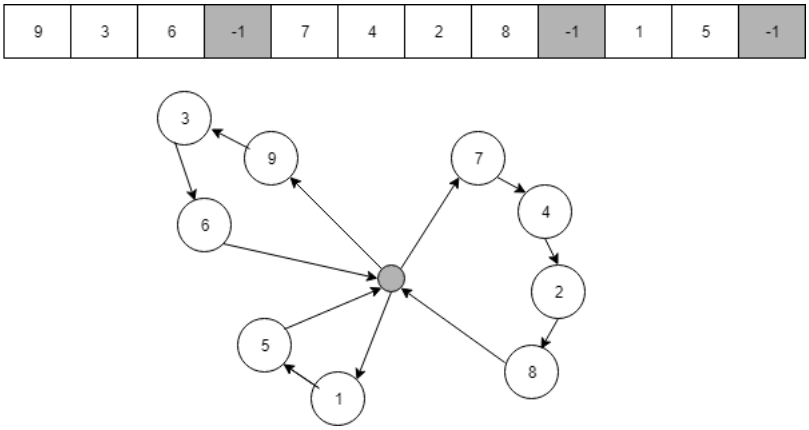


Figura 1: Codificación de una solución para un problema particular

El tamaño de nuestro vector de rutas vendrá determinado por el tamaño de cada una de las rutas que almacene, teniendo en cuenta que al final de cada una, habrá una posición adicional para almacenar el separador de rutas.

$$size = \sum_{i=1}^{r}(|R_i| + 1)$$

Cabe destacar que en nuestra codificación hemos omitido las distancias a los depósitos. Las cuales se situarían justo en la primera y en la última posición dentro de cada ruta (justo antes del separador), aunque si que hay que tenerlas en cuenta a la hora de evaluar la función objetivo.

4. Evaluación de la función objetivo

4.1. Descripción de la función objetivo

La función objetivo (f) es otro de los elementos fundamentales en los problemas de optimización. Su objetivo es dar una medida numérica de la bondad de la solución. En nuestro caso se corresponderá con la distancia total que han recorrido todos los vehículos, la cual deberemos minimizar.

$$f : S \to \mathbb{R}^+$$

$$f(s) = \sum_{i=1}^{r}[\delta(D, v_{i1}) + \sum_{j=2}^{|R_i|-1} \delta(v_{ij}, v_{ij+1}) + \delta(v_{i|R_i|}, D)]$$

La evaluación de nuestra función objetivo recorrerá todos los clientes presentes en cada una de las rutas y calculará la distancia de los clientes situados en la posición j de la ruta i con el cliente situado en la posición $j + 1$ de la misma ruta i. Esta suma se considerará para todas las rutas. Además, es importante considerar las distancias al depósito tanto del primer como del último cliente de la ruta.

4.2. Restricciones o criterios de factibilidad

Los criterios de factibilidad del problema (Ω) es el conjunto de restricciones que debe cumplir una solución para que sea válida para nuestro problema. De esta forma, no todas las soluciones presentes en el espacio de soluciones serán factibles.

El primero de los criterios se corresponde con que todos los clientes de nuestra solución tengan satisfecha su demanda.

$$s \in S \quad es \quad factible \leftrightarrow \bigcup_{i=1}^{r} R_i = V$$

Además, la demanda de las rutas no deber sobrepasar la capacidad del vehículo que la tiene asignada:

$$s \in S \quad es \quad factible \leftrightarrow \forall R_i \in s, \sum_{i=1}^{|R_i|} d_i \leq c_i$$

Finalmente el tercer criterio sería que las soluciones factibles siempre comenzaran y finalizaran en el depósito. Aunque en nuestro caso esta restricción siempre se cumplirá de manera implícita, debido a la codificación que hemos elegido.

5. Definición de las estructuras de entorno y evaluación de los movimientos

Tal y como comentamos anteriormente, no es viable explorar todo el espacio de soluciones de nuestro problema, con lo cual deberemos hacer exploraciones inteligentes para descubrir nuevas soluciones a partir de otras ya dadas. Para hacer esta exploración de vecindad, utilizaremos estructuras de entornos.

Una estructura de entorno (N) es una función que para una solución determinada, asocia un subconjunto de soluciones del problema que sean cercanas en algún sentido:

$$N : S \to Y \subseteq S$$

$$\forall s \in S, \exists s' \in Y \mid N(s) = s'$$

Dado nuestro espacio de soluciones (S) y una solución determinada (s), existirá otra solución (s') perteneciente al espacio de soluciones que sea resultado de aplicar la función definida por la estructura de entorno. Diremos entonces que s' es *vecina* de s y por tanto el entorno de una solución, estará formado por:

$$s' \in N(s) \subseteq S$$

Nuestra labor será entonces definir cuáles serán los movimientos que aplicaremos a la solución s y que nos permitirán generar su entorno, o conjunto de soluciones *vecinas*.

Una característica deseable de las estructuras de entorno es que nos permitan evaluar cuál es el valor objetivo de la solución vecina sin tener que recalcularla completamente, sino solo evaluando el impacto del movimiento en el coste de la solución. Este impacto lo denominaremos *coste del movimiento* ($\Delta\delta$) y nos permitirá generar la función objetivo de la solución vecina de esta forma:

$$f(s') = f(s) + \Delta\delta$$

Para nuestro problema, hemos definido cuatro estructuras de entorno diferentes, cuyo movimiento y cálculo del coste de la solución vecina explicaremos por separado.

5.1. Intercambio intrarruta

5.1.1. Definición

Esta estructura de entorno viene definida por el movimiento de seleccionar un cliente de una ruta e intercambiarlo con otro cliente de la misma ruta.

A partir de una solución inicial, el número de soluciones vecinas viene dado por el número de posibles intercambios intrarruta que se puedan hacer en la solución. Esto es, para cada ruta, el número de posibles combinaciones de los clientes cogidos de dos en dos.

$$\sum_{i=1}^{r} \binom{|R_i|}{2} = \sum_{i=1}^{r} \frac{|R_i|(|R_i|-1)}{2}$$

Para todas las iteraciones que realizará esta estructura de entorno, la función seleccionará una ruta determinada (R_i) y dentro de ella iterará por los diferentes pares de clientes posibles (v_j, v_k) intercambiándolos.

5.1.2. Evaluación de la función objetivo

El impacto de aplicar este movimiento en la solución inicial (s) hará que se deban hacer algunos cambios en su valor objetivo, para computar el valor de la solución vecina (s').

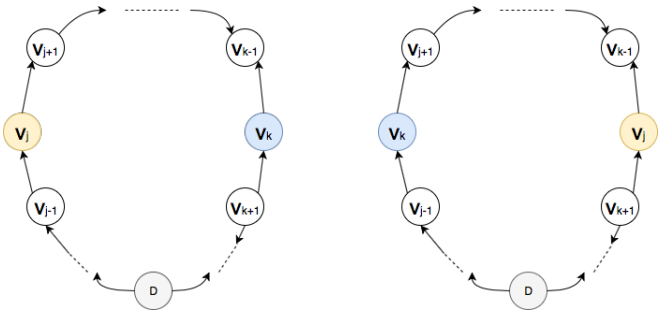


Figura 2: Visualización de un intercambio intrarruta

Si lo pensamos gráficamente, deberemos eliminar los arcos que van desde el par de los vecinos seleccionados (v_j, v_k) hasta sus predecesores y antecesores respectivamente. Esto se correspondería con el siguiente valor de distancias:

$$\delta_1 = \delta(v_{j-1}, v_j) + \delta(v_j, v_{j+1}) + \delta(v_{k-1}, v_k) + \delta(v_k, v_{k+1})$$

Luego deberemos definir un segundo valor de distancias, es decir, aquel que se corresponde con los nuevos enlaces que crearemos debido al movimiento.

$$\delta_2 = \delta(v_{j-1}, v_k) + \delta(v_k, v_{j+1}) + \delta(v_{k-1}, v_j) + \delta(v_j, v_{k+1})$$

Estos dos valores de distancias deben ser restados para poder así conocer la diferencia en la distancia total que obtendremos con el movimiento, lo que será el *coste del movimiento*.

$$\Delta\delta = \delta_2 - \delta_1$$

Finalmente para conocer el coste de cualquier solución vecina a nuestra solución inicial tan solo deberemos sumar el coste del movimiento al coste inicial de esta:

$$f(s') = f(s) + \Delta\delta$$

En cuanto a la factibilidad de las soluciones vecina generadas mediante intercambio intraruta, podemos afirmar que el movimiento no afectará a la factibilidad de la solución. Esto se debe a que el orden de visita de los clientes dentro de una ruta no afecta ni a la capacidad del vehículo que las visita ni a si se visitan todos los clientes. De esta forma si la solución inicial es factible,

las vecinas generadas a partir de ella por el intercambio intraruta también lo serán.

5.2. Intercambio entre rutas

5.2.1. Definición

La idea básica de este movimiento consiste en elegir dos nodos de dos rutas diferentes e intercambiarlos entre sí. Para ello debemos de tener en cuenta algunas limitaciones que impiden el movimiento.

Para poder intercambiar elementos de dos rutas, es necesario que estas rutas no estén vacías, por lo que debemos de contemplar el saltar rutas que no contengan elementos, tanto de la ruta que partimos hacia la que queremos intercambiar.

También debemos de tener en cuenta no repetir intercambios. Para ello, consideramos que la ruta destino (a la que queremos intercambiar) siempre está en la siguiente ruta no vacía a la ruta origen (de la que queremos intercambiar). De esta manera evitaríamos repetir intercambios pues en esta estructura de entorno no importa el orden de los movimientos, es lo mismo intercambiar $v_i \leftrightarrow v_j$ y $v_j \leftrightarrow v_i$

El número posible de soluciones vecinas o movimiento que realizará está estructura de entorno viene determinado por:

$$\sum_{i=1}^{r-1}(|R_i| * \sum_{j=i+1}^{r} |R_j|)$$

Por último, podemos ver un ejemplo del movimiento interruta en la figura 3.

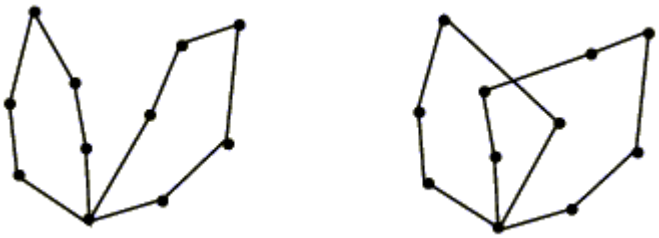


Figura 3: Visualización de un intercambio interruta

5.2.2. Evaluación de la función objetivo

Para poder evaluar el impacto del movimiento en la solución inicial sin tener que recalcularla completamente consideraremos estos 2 elementos:

- v_{ip}: Se trata del cliente situado en la posición p de la ruta i, donde $1 \leq p \leq |R_i|$. Consideramos que la ruta i es el origen del movimiento.
- v_{jq}: Se trata del cliente situado en la posición q de la ruta j, donde $1 \leq q \leq |R_j|$. Consideramos que la ruta j es el destino del movimiento.

Estos elementos nos servirán para evitar generar la solución cada vez que cambiemos de movimiento, si no que sólo la generaremos cuando nos lo pidan explícitamente, si el coste ha sido mejor.

Para poder determinar el coste de la solución resultante del movimiento, sin tener que generar una solución, llevaremos a cabo los siguientes cálculos. Antes de nada recordemos que para que un cliente se pueda intercambiar, la ruta no puede estar vacía, además, con que en la ruta haya sólo un cliente, ya existen 3 nodos en la ruta, pues parte y llega al depósito.

Dado el cliente de la ruta origen que hemos intercambiado, debemos de quitar de esa la función objetivo el coste del nodo previo y posterior a éste cliente. Esto también lo haremos con el cliente de la ruta destino. Esta distancia la denominaremos δ_1.

$$\delta_1 = \delta(v_{ip-1}, v_{ip}) + \delta(v_{ip}, v_{ip+1}) + \delta(v_{jq-1}, v_{jq}) + \delta(v_{jq}, v_{jq+1})$$

A su vez deberemos sumar el coste del que era predecesor del cliente origen al cliente destino y del cliente destino al que era sucesor del cliente origen. Esto se hará recíprocamente en la ruta destino. A este coste lo denominaremos δ_2.

$$\delta_2 = \delta(v_{ip-1}, v_{jq}) + \delta(v_{jq}, v_{ip+1}) + \delta(v_{jq-1}, v_{ip}) + \delta(v_{ip}, v_{jq+1})$$

A partir de estas distancias, calcularemos el coste del movimiento tal y como hicimos anteriormente.

$$\Delta\delta = \delta_2 - \delta_1$$

Lo cual nos permitirá computar de forma eficiente el coste de las soluciones vecinas.

$$f(s') = f(s) + \Delta\delta$$

En cuanto a la factibilidad de la solución, debemos tener en cuenta que este movimiento puede generar soluciones que no sean factibles a partir de otras que si lo sean, puesto que el cambio de un cliente de una ruta a otra puede sobrepasar la capacidad del vehículo que la atenderá. Con lo cual en cada movimiento, para comprobar la factibilidad deberemos asegurarnos que la solución vecina cumple las restricciones definidas en Ω.

Esta comprobación también se puede hacer de manera eficiente para ambas rutas, sin tener que recalcular la suma de las demandas. De esta forma, tan solo tendremos que restar la demanda del cliente que extraemos y sumar la del cliente que añadimos en cada una de las dos rutas.

$$(\sum_{k=1}^{|R_i|} d_k) - d_p + d_q \leq c_i$$

$$(\sum_{k=1}^{|R_j|} d_k) - d_q + d_p \leq c_j$$

Debemos destacar hay que comprobar si esta nueva demanda no sobrepasa la capacidad del vehículo i o j respectivamente, que será el cual la atenderá.

5.3. Reinserción

5.3.1. Definición

La reinserción o *relocate* es otro de los movimientos que consideramos para intentar mejorar nuestra solución base. La idea básica consiste en extraer un cliente de una ruta e insertar ese cliente en otra ruta diferente. No obstante existen algunas limitaciones a la hora de su implementación.

Para poder mover clientes de una ruta a otra, como es lógico, necesitamos que esa ruta base tenga clientes, al menos uno. Sin embargo, a la hora de elegir la ruta de destino, no hace falta que cumpla el criterio de contener clientes, pues estamos insertando en ella, por lo que podemos trabajar con

rutas vacías.

Además de ésto, a diferencia del intercambio entre rutas, no sólo debemos considerar las rutas siguientes a la ruta base, si no que, **por cada ruta no vacía** debemos tratar de insertar cada elemento en cada posible lugar de todas las demás rutas, lógicamente sin incluir la ruta base, pues recordemos que éste movimiento es entre rutas.

Si analizamos esta complejidad, por cada cliente podemos encontrar un vecino si insertamos ese cliente en cualquier otra posición de cualquier otra ruta. Un detalle importante es que debemos tener en cuenta insertarla antes, entre y después de cada posible elemento. En el caso de haber una ruta vacía, lo insertaríamos en ella sólo 1 vez, pues no hay mas posiciones.

El numero de movimientos que aplicará la estructura de entrono vendrá determinado entonces por:

$$\sum_{i=1}^{r}\sum_{j=1}^{|R_i|}[\sum_{k=1}^{j-1}(|R_k|+1)+\sum_{k=j+1}^{r}(|R_k|+1)]$$

Cabe añadir que, si comparamos éste movimiento con los vistos anteriormente, podemos afirmar que es el que más diversificación nos aporta, pues es capaz de insertar elementos en rutas sin elementos y vaciar rutas, cambiando por completo la solución. Tal y como podemos observar en el ejemplo de la figura 4.

5.3.2. Evaluación de la función objetivo

Al igual que en el movimiento del intercambio entre rutas, para calcular los movimientos de una manera eficiente, utilizaremos estos dos elementos:

- v_{ip}: Se trata del cliente situado en la posición p de la ruta i, donde $1 \leq p \leq |R_i|$. Consideramos que la ruta i es el origen del movimiento.
- v_{jq}: Se tratará de la posición q de la ruta j, donde $1 \geq q \geq |R_j|$. Este cliente se desplazará y será el cual pase a ser el posterior al cliente que queremos insertar, considerando a j la ruta destino.

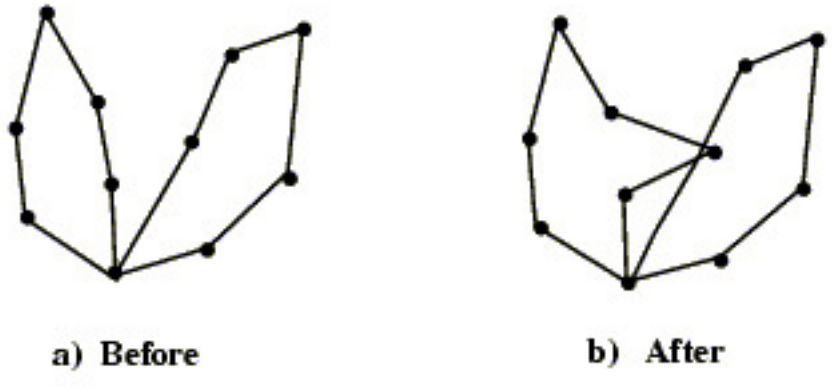


Figura 4: Visualización de un intercambio por reinserción

Estos elementos nos servirán para evitar generar la solución cada vez que cambiemos de movimiento, si no que sólo la generaremos cuando nos lo pidan explícitamente, si el coste ha sido mejor.

Antes de pasar a ver los cálculos necesarios para hallar el coste del movimiento sin generar la solución, recordemos que un cliente se puede insertar en una ruta vacía pero no podemos extraer un cliente de una ruta vacía.

Dado el cliente que hemos extraído de la ruta origen (v_{ip}), debemos quitar el coste de ir de su predecesor al cliente, así como del cliente al sucesor. Y sumar la nueva distancia entre el predecesor y el sucesor respectivamente. Esto nos servirá si en la ruta sólo quedaba un elemento, pues quitaríamos los costes al depósito. A esta operación la denominaremos δ_1.

$$\delta_1 = \delta(v_{ip-1}, v_{ip+1}) - \delta(v_{ip-1}, v_{ip}) - \delta(v_{ip}, v_{ip+1})$$

Ahora bien, en cuanto a la ruta destino podremos tener dos situaciones diferentes. En primer lugar podemos tener una ruta vacía, en la cual consideraremos la siguiente distancia como δ_2:

$$\delta_2 = \delta(D, v_{ip}) + \delta(v_{ip}, D) = 2\delta(v_{ip}, D)$$

Como en todos los casos, deberemos tener en cuenta las distancias a los depósitos. En las rutas con un solo elemento, serán los únicos costes.

La segunda situación que se puede presentar es que la ruta destino j posea más de un elemento, en este caso, consideraremos δ_2 como:

$$\delta_2 = \delta(v_{jq-1}, v_{ip}) + \delta(v_{ip}, v_{jq}) - \delta(v_{jq-1}, v_{jq})$$

Suponemos que la posición q es donde queremos insertar el elemento en la ruta destino (j) y que v_{jq} es el elemento que se encuentra en dicha posición. Deberemos restar el coste de ir del predecesor de v_{jq} al propio elemento, y sumar las distancias al elemento v_{ip} tanto del predecesor como del sucesor. De esta forma, nos quedaría así el coste del movimiento:

$$\Delta\delta = \delta_1 + \delta_2$$

Y como en otros movimientos, podremos calcular el coste a la solución vecina:

$$f(s') = f(s) + \Delta\delta$$

Tal y como comentamos previamente, el movimiento de reinserción es el que más diversidad genera, así que tendremos que tener en cuenta que las soluciones vecinas de una solución factible, podrían ser no factibles. La factibilidad solo podría estar afectada en la ruta destino, puesto que la demanda del cliente que insertamos en dicha ruta podría sobrepasar la capacidad del vehículo que la satisface. Por ello deberemos comprobar en la ruta j:

$$(\sum_{k=1}^{|R_j|} d_k) + d_p \leq c_j$$

5.4. 2-Opt

5.4.1. Definición

La última estructura de entorno que utilizaremos es la definida por el movimiento 2-opt. Esta fue propuesta por G. A. Croes en 1958, en el contexto de la resolución del problema del viajante de comercio [6]. La idea conceptual de este movimiento es reorganizar una ruta que se cruza en si misma para evitar dicho cruce.

Se trata de un movimiento intrarruta, y consiste en elegir dos clientes de una determinada ruta $v_i \in R_k$ y otro cliente $v_j \in R_k$ donde $R_k \in S$, a continuación, eliminamos las dos aristas. Del primer nodo elegido, eliminaremos la

arista que llega hacia él, y del segundo eliminaremos la arista que sale de él. Esa exploración la haremos hasta que se hayan efectuado todas las posibles combinaciones de dos elementos en todas las rutas que posee nuestra solución inicial.

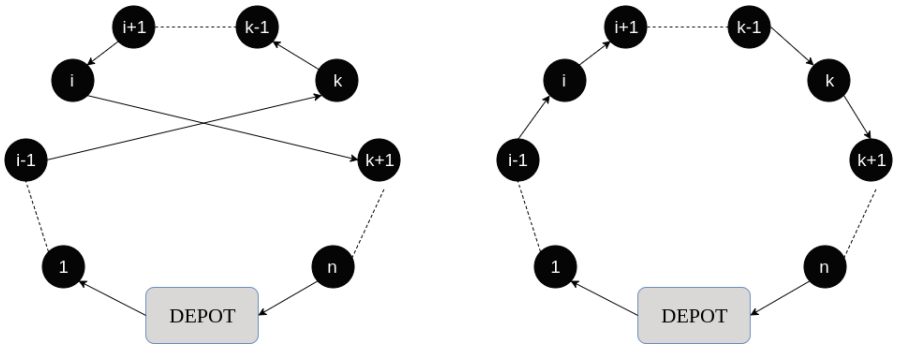


Figura 5: Visualización de un intercambio 2-opt

Dada una solución inicial, el movimiento de la estructura de entorno explorará cada una de las rutas $R \in S$ donde el número de clientes sea mayor que dos. Dentro de dicha ruta considerará cada cliente que se encuentre en las posiciones $i \in \{1, 2, \ldots, |R| - 1\}$ y $j \in \{2, \ldots, |R|\}$.

Si tenemos un conjunto de R_i rutas donde cada una de las rutas tiene $|R_i| \geq 2$, entonces el número de iteraciones que realizará el movimiento en la estructura de entorno es:

$$\sum_{i=1}^{r} \binom{|R_i|}{2} = \sum_{i=1}^{r} \frac{|R_i|(|R_i| - 1)}{2}$$

Dada nuestra codificación de la solución, y considerando una iteración en la que seleccionamos un cliente v_i y v_j, el algoritmo 2-opt realizará los siguientes cambios en el vector de las soluciones:

- Para todos los clientes $v_i \in R_k$ de la ruta actual, introducirlos desde la posición 0 hasta la $|R_k| - 1$, de manera ascendente.
- Para todos los clientes de la ruta actual R_k desde $v_i \in R_k$ hasta $v_j \in R_k$ introducirlos en orden inverso.

- Para todos los clientes de la ruta R_k desde $v_{j+1} \in R_k$ hasta $|R_k|$ introducirlos de manera ascendente.

Este procedimiento se llevará a cabo por cada una de las iteraciones de búsqueda de vecinos de la solución actual. Y además, para cada iteración calcularemos la variación que se llevará a cabo en la función objetivo tras hacer el movimiento. Este nuevo coste (c'), se calculará de esta forma:

$$c' = c - d_{i-1,i} - d_{k,k+1} + d_{i,k+1} + d_{i-1,k}$$

Este nuevo coste corresponderá con el coste que teníamos previamente menos las dos nuevas aristas que eliminamos. Sin embargo, debemos sumar las distancias de las nuevas aristas que debemos crear.

6. Búsquedas por entornos

En general, una búsqueda por entorno consiste en, partiendo de una solución inicial, ir actualizando la solución actual a una de su entorno hasta que se cumpla un criterio de parada.

En el caso más concreto de una búsqueda local, ésta consiste en, partiendo de una solución inicial, ir actualizando la solución actual a una mejor de su entorno hasta que la solución actual sea la mejor de su entorno. Es decir, hasta que la solución actual sea un óptimo local.

Para explicarlo de una manera gráfica, representaremos la solución actual como un punto en un plano y su estructura de entorno como una circunferencia con la solución que genera el entorno como centro. Se puede ver el funcionamiento de una búsqueda local como se muestra en la figura 6:

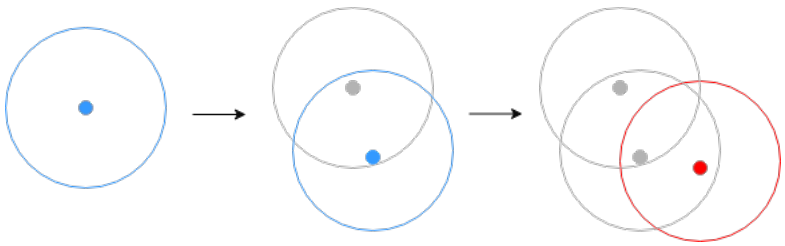


Figura 6: Visualización de una búsqueda local

Donde el punto azul es la solución actual, los puntos grises los previos y el punto rojo la solución óptima local.

A continuación definiremos los tipos de búsqueda que hemos implementado.

6.1. Búsqueda greedy

La búsqueda *greedy* es una búsqueda local en la que, a la hora de actualizar la solución actual, se elige al mejor vecino de la solución actual.

6.2. Búsqueda ansiosa

La búsqueda ansiosa es una búsqueda local en la que, a la hora de actualizar la solución actual, se elige al primer vecino que sea mejor que el actual.

6.3. Variable Neighborhood Descent

La búsqueda variable descendente (o VND del inglés *Variable Neighborhood Descent*) es una búsqueda local que se basa en el siguiente principio:

- Un óptimo local bajo una estructura de entorno no tiene que serlo necesariamente bajo otra.

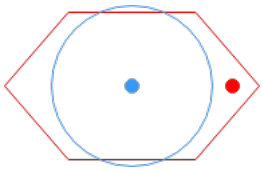


Figura 7: Visualización de un óptimo local sobre un entorno y no sobre otro

En la figura 7 el punto azul representa la solución actual, la cual es un óptimo local respecto a la estructura de entorno representada por la circunferencia azul. El punto rojo representa una solución vecina respecto de la estructura de entorno representada por el hexágono rojo. Se puede ver que, como la estructura de entorno roja contiene soluciones vecinas que no tiene la azul, puede ocurrir que la solución roja sea mejor que la azul. Por tanto, el punto azul no sería un óptimo local respecto a la estructura de entorno roja.

Por esto el VND no usa una sola estructura de entorno, sino que usa una lista de estructuras de entornos. Su funcionamiento se basa en ir actualizando la solución actual a la mejor solución vecina del primer entorno, cuando la solución actual sea un óptimo local, se busca la mejor solución respecto del siguiente entorno de la lista. En caso de no encontrar ninguna mejor entonces la solución actual también es óptimo local respecto del segundo entorno y se repetiría el procedimiento con el siguiente entorno de la lista. Cuando la solución actual sea un óptimo local respecto de todos los entornos se para la búsqueda y se devuelve esa solución. En caso de que se encuentre una mejor solución en cualquier estructura de entorno, se actualiza la solución actual a esa mejor solución y se continúa la búsqueda VND con el primer entorno.

6.4. Notas de implementación

Siguiendo un desarrollo orientado a objetos, las búsquedas locales reciben como parámetros tanto la solución inicial como los movimientos a usar. De esta manera se pueden probar todas las combinaciones de movimientos para cada búsqueda sin tener que repetir el código de los movimientos para cada búsqueda local. Por la misma razón, las propias búsquedas locales también se han implementado como objetos para poder pasarlas como parámetros a los algoritmos que necesiten usarlas.

7. Algoritmos implementados

7.1. Constructivo determinista

Un algoritmo determinista es un algoritmo que es totalmente predictivo, entendiéndose esto como: si se conocen la entrada del algoritmo, se conocerá la salida que generará, además, se diferencian de los no deterministas en que, estos, si son ejecutados **N** veces con la misma entrada, las **N** veces producirá exactamente la misma salida, en cambio los no deterministas tienen un factor aleatorio y no siempre producen la misma salida teniendo la misma entrada.

- **Fase de construcción:** Si tenemos en cuenta que nuestro problema tiene una serie de vehículos (en principio desconocidos), la fase de construcción tratará de asignar a un vehículo determinado los clientes a los que puede abastecer, en función de la capacidad de abastecimiento máxima que tenga nuestro vehículo (en nuestro caso hemos optado porque todos los vehículos tengan las mismas capacidades). Si un vehículo se queda sin capacidad para abastecer a cualquier otro cliente se deberá

añadir otro nuevo vehículo y, evidentemente, otra nueva ruta para ese nuevo vehículo, que tendrá que abastecer a los clientes restantes que pueda. Esto lo haremos hasta que todos los clientes sean abastecidos.

7.1.1. Implementación propuesta

El algoritmo ha de decidir qué cliente añadir en cada momento, para ello hacemos uso de la distancia del nodo actual hasta el próximo nodo, de manera que se introducirán en las rutas **el cliente más próximo que se pueda abastecer** teniendo en cuenta la posición del nodo actual y la capacidad restante del vehículo.

7.2. Greedy Randomized Adaptative Search Procedure

GRASP es un **algoritmo metaheurístico** ampliamente utilizado en problemas de optimización combinatoria [3]. Normalmente consta de dos fases que se ejecutan de forma iterativa. Una fase de construcción y una fase de mejora. La fase de construcción se encarga de generar una solución base, y la fase de mejora busca un óptimo local a partir de dicha solución base.

- **Fase de construcción:** en esta fase se genera una solución base con un algoritmo greedy al que se le añade un pequeño componente aleatorio. Dicha aleatoriedad consigue que la generación de la solución base en la fase de creación no sea determinista, otorgando diferentes soluciones para la misma entrada en distintas ejecuciones.

  La aleatoriedad se consigue haciendo uso de una lista **restringida de candidatos**. En dicha lista se almacenan los N mejores candidatos a introducir en la solución que se está construyendo. De esos N mejores candidatos se escoge uno de manera aleatoria y se añade a la solución. Tras haberlo añadido se ha de adaptar la lista restringida de candidatos eliminando aquellos que han dejado de ser candidatos factibles (candidatos que al haber añadido el anterior no pueden ser añadidos pues se pierde la factibilidad de la solución que se está construyendo) y añadiendo los nuevos posibles candidatos (si es que los hay) hasta alcanzar el tamaño máximo de la lista restringida de candidatos.

- **Fase de mejora:** tras la fase de construcción comienza la fase de mejora. En esta fase se le aplica a la solución base generada en la fase previa una estrategia de búsqueda local, obteniendo así un óptimo local.

Durante la ejecución del GRASP vamos almacenando la mejor solución obtenida para devolverla cuando se cumpla algún criterio de parada.

7.2.1. Implementación propuesta

En nuestro caso en particular, hemos decidido que el criterio a seguir en el algoritmo de la fase constructiva sea escoger **el nodo más cercano al nodo actual**. Además, la función en la que se realiza el GRASP recibe los siguientes parámetros:

- **Número de iteraciones:** Especifica el número máximos de iteraciones que se ejecutarán. Entendemos por iteración una fase de construcción y una fase de mejora.
- **Número de iteraciones sin mejora:** Especifica el número máximo de iteraciones sin mejora que se ejecutarán. Si se alcanza esta cantidad de iteraciones sin haber mejorado la mejor solución obtenida hasta el momento se alcanza la condición de parada y el algoritmo devuelve la mejor solución obtenida hasta el momento.
- **Tamaño de la lista de candidatos:** Determina el tamaño que tendrá la lista restringida de candidatos de la fase constructiva del algoritmo.
- **Estrategia de búsqueda local:** Simboliza la estrategia de búsqueda local que se utilizará durante la ejecución de la fase de mejora de GRASP. Este parámetro contiene el tipo de movimiento que se aplicará para generar las soluciones vecinas.

7.3. Método Multiarranque

Mediante la aplicación de una búsqueda local a una sola solución se puede garantizar que se obtiene un óptimo local. Pero no se puede estar seguro de que sea la mejor solución posible (óptimo global). Los métodos multiarranque permiten explorar más zonas del espacio de soluciones mediante la construcción de varias soluciones iniciales distintas y la aplicación de una búsqueda local sobre cada una. Esto se puede hacer de manera secuencial o paralela y se devuelve como resultado la mejor solución obtenida.

7.3.1. Implementación propuesta

A continuación se describen los detalles del método multiarranque implementado para el presente trabajo.

- **Fase de construcción:** Las soluciones iniciales han de ser diferentes para poder explorar distintos puntos del espacio de soluciones. En el presente trabajo se ha decidido que la construcción de las soluciones iniciales sea de manera aleatoria pero de manera que se garantice la factibilidad de la solución.

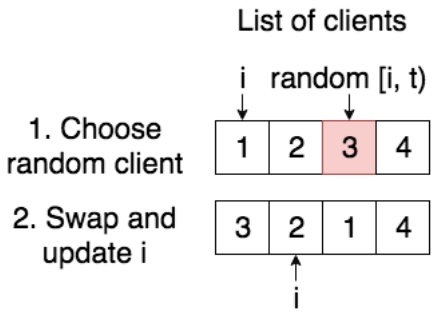


Figura 8: Visualización de la selección aleatoria de clientes

Partiendo de una lista de clientes de tamaño t se va insertando en la i-ésima posición de la solución un cliente tomado aleatoriamente de la lista de clientes en el rango $[i, t)$. Si la demanda del cliente no puede ser satisfecha en esa ruta primero se inserta un separador (-1) y luego el cliente. Finalmente se intercambian las posiciones del i-ésimo cliente con el tomado aleatoriamente. De esta manera se tienen en las posiciones de 0 a $i-1$ de la lista de clientes los clientes que ya han sido introducidos en la solución y no se consideran para las restantes inserciones.

- **Búsqueda local:** La búsqueda local a usar es un parámetro que se le pasa al método. De esta manera se pueden hacer experimentos con búsquedas locales distintas sin necesidad de modificar el código del multiarranque.

- **Iteraciones:** Se considera como una iteración la ejecución de la fase de construcción seguida por la búsqueda local sobre la solución obtenida en la fase construcción. Además se actualiza el valor de la mejor solución encontrada en caso de que el nuevo óptimo local sea mejor que el mejor óptimo local registrado hasta el momento. Como criterio de parada se usa un número de iteraciones sin mejora, parámetro que se le pasa al método como parámetro.

7.4. Variable Neighborhood Search

La búsqueda por entornos variable (o VNS del inglés *Variable Neighborhood Search*) [5] es un método basado en una idea simple que ha sido aplicado exitosamente a una gran variedad de problemas.

En el VNS se parte de un óptimo local inicial como mejor solución y, para tratar salir del óptimo local de la mejor solución encontrada hasta el momento, se aplica una agitación/perturbación. Esto consiste en obtener una solución del entorno distinta, pudiendo ser una solución vecina obtenida de manera aleatoria o siguiendo algún criterio. A continuación, a esta solución alejada del óptimo local se le aplica una búsqueda local. Puede ocurrir que se llegue a un óptimo local mejor, en este caso se actualiza la mejor solución a ésta y se repite el proceso con la primera estructura de entorno. En caso de que no se llegue a un óptimo local mejor, se repite el proceso con la siguiente estructura de entorno. Así hasta que no se consiga salir del mejor óptimo local agitando y mejorando la solución en todas las estructuras de entornos a usar.

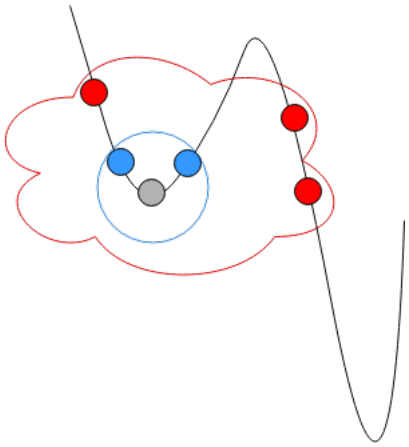


Figura 9: Visualización de la fase de agitación del VNS

En la figura 9 el punto gris representa el mejor óptimo local encontrado hasta el momento. La circunferencia azul representa la primera estructura de entorno a usar y la nube roja la segunda estructura de entorno. Los puntos azules y rojos son soluciones vecinas de la solución gris. La gráfica representa

la función objetivo.

Se puede observar que tras agitar la solución en el primer entorno y aplicar una búsqueda local **usando el mismo entorno**, no se lograría llegar al óptimo global. Pero note que si se usara VND o una búsqueda local con una estructura de entorno mayor se podría llegar al óptimo global porque está relativamente cerca.

En un ejemplo real un mejor óptimo local se podría encontrar mucho más lejos y rodeado de soluciones peores por lo que una agitación en una estructura de entorno pequeña acompañada de una búsqueda local no serviría para acercarse al mejor óptimo local. Por otro lado las estructuras de entornos pequeñas son computacionalmente más eficientes, note que en el caso más extremo una estructura de entorno puede estar compuesta por el espacio completo de posibles soluciones factibles y la búsqueda local sería una búsqueda exhaustiva lo cual es lo que se pretende evitar con los métodos heurísticos. Por estos motivos en el VNS se usan varias estructuras de entorno que, idealmente, estarían ordenadas de menor a mayor tamaño.

7.4.1. Implementación propuesta

- **Agitación:** Para el presente trabajo se implementado la agitación de manera aleatoria debido a la sencillez de este método. Un posible criterio para no implementar la agitación de manera totalmente aleatoria podría ser coger la solución del entorno más alejada. Sin embargo, para ello habría que, además, definir una medida de distancia entre soluciones.

  Debido a la manera de implementar los movimientos no es posible obtener directamente un i-ésimo vecino. En lugar de eso se tiene que actualizar el movimiento para que apunte a los nuevos vecinos de manera secuencial, uno tras otro (es preciso recordar que solo se actualiza el estado del objeto movimiento, la solución vecina sólo se genera cuando se pide explícitamente). Entonces, para poder implementar la agitación de manera aleatoria no se puede hacer uso de un número aleatorio de 1 a k y coger ese vecino. En lugar de eso, se va actualizando el estado del movimiento para que apunte a nuevos vecinos y en cada iteración se genera un número aleatorio de 0 a 1, si dicho número es menor o

igual que una probabilidad se toma al vecino que apunta el movimiento como resultado de la agitación. Dicha probabilidad, aunque podría ser un parámetro del algoritmo, se ha decidido implementar como una constante (0.001) debido al gran número de posible combinaciones de parámetros que ya tiene el VNS, especialmente si se usa como búsqueda local el VND.

- **Búsqueda local:** Si la búsqueda local usada es el VND al VNS se le denomina VNS general (GVNS), mientras que si se trata de otra búsqueda local se le denomina VNS básico (BNVS). En el presente trabajo, como se ha indicado previamente en la sección de búsquedas por entornos y en otros algoritmos, el método recibe como parámetro la búsqueda local que va a usar. Por tanto, para realizar los experimentos se ha podido tener en cuenta tanto el BNVS como el GVNS sin necesidad de cambiar el código del VNS.

- **Iteraciones:** Como se mencionó previamente, la ejecución del VNS termina cuando no ocurre ninguna mejora de la mejor solución encontrada tras hacer una agitación usando todas las estructuras de entorno dadas y aplicar la búsqueda local. Como a la hora de implementarlo se vio que el método se ejecutaba muy rápido, se decidió implementar el VNS en un bucle. Por tanto, **el VNS implementado en el presente trabajo es en realidad un bucle de VNS** donde los VNS siguientes al primero reciben como solución inicial la salida del VNS anterior. Este bucle itera mientras la solución no mejore un número de veces que se recibe como parámetro. El beneficio que aporta este bucle es que, al haber implementado la agitación de manera aleatoria, puede ocurrir que al ejecutar un siguiente VNS la agitación genere una solución lo suficientemente perturbada como para alcanzar un mejor óptimo local.

7.5. Tabu search

La Búsqueda Tabú es un algoritmo metaheurístico que se usa para resolver problemas de optimización combinatoria. Se trata de una búsqueda que aumenta el rendimiento del método de búsqueda local gracias a utilizar estructuras de memoria como la lista de clientes con Tabu Tenure. Cabe añadir que la Búsqueda Tabú fue introducida en 1986 por **Fred Glover** [4].

El algoritmo consiste en realizar una búsqueda local que utiliza una estructura de memoria para penalizar ciertos movimientos. Esta estructura se

denomina **Tabu Tenure**. Recordemos que una búsqueda local se trata de un algoritmo para explorar un entorno y, tras haber recorrido el mismo, si se encuentra una solución mejor a la inicial, seguir aplicando la búsqueda desde esa solución hasta que no haya opción de mejora. En el caso de la búsqueda tabú cuando se elije una solución mejor a la actual, se establece un criterio de penalización, introduciendo ese movimiento en el Tabu Tenure durante un número de iteraciones. Si, a la hora de coger una nueva solución, esta está en la lista de Tabu Tenure, se seguirá explorando, pues las soluciones de dentro de la lista están penalizadas. Esto evita que se produzcan bucles, de ahí el destacado éxito de la búsqueda tabú a la hora de resolver problemas de optimización duros.

Cabe añadir que existe una excepción en la que podemos coger una solución a pesar de que este en la Tabu Tenure. Esto es cuando la solución que podemos obtener es mejor que cualquiera de las obtenidas anteriormente. A esto se le denomina **criterio de aspiración.**

Respecto a las estructuras de memoria a utilizar en la búsqueda tabú, se pueden clasificar según cuatro características principales, la propiedad de ser reciente, de su frecuencia, de su calidad y en la influencia.

7.5.1. Implementación propuesta

A la hora de penalizar un movimiento e introducirlo en la Tabu Tenure, debemos establecer la manera de penalizarlo. En el caso del VRP, todos los movimientos excepto el movimiento de reinserción consisten en intercambiar dos clientes, ya sea de la misma o de rutas distintas. Por ello, para penalizar un movimiento, penalizaremos que esos dos clientes se vuelvan a intercambiar durante un número de iteraciones. Para poder usar la búsqueda tabú con todos los movimientos, en el caso del movimiento de reinserción, penalizaremos que ese cliente se vuelva a mover durante un número de iteraciones.

El método de la búsqueda tabú recibe los siguientes parámetros:

- **Porcentaje de Tabu Tenure:** Indica el tamaño de la lista de movimientos penalizados, que será un porcentaje de la lista de clientes total.
- **Lista de Movimientos para la búsqueda local:** A la hora de la

implementación, hemos decidido añadir la posibilidad de utilizar varios movimientos para realizar la búsqueda local, en vez de sólo guiarnos por uno. La manera en la que utilizamos éstos movimientos es aleatoria, es decir, cada vez que vamos a explorar el entorno de una solución, elegimos un movimiento al azar entre la lista que nos pasen. Esto nos ha traído mucho mejores resultados que sólo utilizando uno.

- **Número de iteraciones sin mejora:** Cuando buscamos una solución mejor que la inicial, es raro encontrarla la primera vez que realizemos la búsqueda local. Por ello, hemos decidido añadir un número de iteraciones sin mejora para darle una *segunda oportunidad* al algoritmo y así poder realizar de nuevo la búsqueda local, pero esta vez partiendo de un vecino aleatorio de la anterior búsqueda.

7.6. Large neighborhood search

LNS o búsqueda de vecinos en entornos grandes es una búsqueda local propuesta por Paul Shaw en 1998 [9]. Inicialmente esta búsqueda fue pensada para solucionar problemas de similares al VRP, es por ello que hemos obtenido unos buenos resultados aplicándola. La búsqueda se basa en que, dada una solución base inicial, se destruye un porcentaje de esa solución y luego se vuelve a reconstruir de nuevo, asegurándonos que la nueva solución obtenida a partir de la reconstrucción es factible. Hay diferentes maneras para reconstruir la solución y para destruirla, sin embargo, optaremos por una destrucción aleatoria y una reconstrucción greedy.

- **Fase de destrucción:** En la fase de destrucción se destruye un porcentaje de la solución de forma aleatoria y descartando aquellos clientes que estén en una ruta con un solo cliente. Estos clientes serán introducidos en un vector de clientes eliminados V_R, para, posteriormente ser reintroducidos en la solución en la fase de reconstrucción.

- **Fase de construcción:** Para la fase de construcción se tienen en cuenta los $|V_R|$ clientes eliminados de las rutas para ser insertados de nuevo siguiendo un algoritmo greedy. De esta manera, conseguimos insertar los clientes eliminados en las rutas que hagan que la nueva distancia total de esta nueva solución sea mínima.

Si esta nueva solución construida es mejor que la que teníamos anteriormente, entonces, en la siguiente iteración se aplicará el algoritmo sobre esa solución, descartando la anterior.

7.6.1. Implementación propuesta

- **Numero de iteraciones:** Es un parámetro que indica el número de veces que se reconstruirá una solución en busca de otra mejor solución. Cuando se lleguen a ejecutar todas las iteraciones especificadas se devolverá la mejor solución obtenida hasta el momento.

- **Diferencia entre distancia total:** Siendo s_i la mejor solución encontrada hasta el momento, si, después de aplicar la fase de destrucción y construcción se obtiene una nueva solución s_{i+1} cuya distancia total es mayor a un determinado valor, entonces se aplicará una búsqueda local sobre esa solución, siendo la mejor solución encontrada hasta el momento el resultado de aplicar la búsqueda local.

- **Fase de mejora:** Si la diferencia entre la distancia total de la nueva solución s_{i+1} y la anterior solución s_i es mayor a un determinado valor se aplicará una búsqueda local para mejorar aún más la solución, esta búsqueda local será un parámetro de nuestro algoritmo de búsqueda.

8. Experiencia Computacional

8.1. Objetivos de la experiencia computacional

En este apartado se compararán todos los algoritmos anteriormente expuestos. Dado que cada uno de estos algoritmos tiene partes variables, primero se elegirá que combinación de parámetros resulta mejor respecto a la relación calidad-tiempo de la solución para cada uno de ellos. Tras ésto, asegurándonos que tenemos la mejor combinación para cada algoritmo, podremos hacer comparaciones entre algoritmos. Concretamente analizaremos tres aspectos:

- **Mejor algoritmo para obtener la solución inicial.** Comparando el algoritmo GRASP con el método Multiarranque.

- **Mejor algoritmo para obtener la mejor solución en relación calidad-tiempo.** Comparando los algoritmos VNS, Tabu Search y LNS.

- **Mejor algoritmo para obtener la mejor solución lo mas rápido posible.** Comparando los algoritmos VNS, Tabu Search y LNS.

8.2. Descripción de las instancias utilizadas

Las instancias se han sacado de la página web de la Universidad de Málaga [2]. Encontramos más de 60 instancias diferentes, sin embargo, elegimos un conjunto de ellas en las que el número de clientes variara para ver cómo se comportaban los algoritmos implementados con un número mayor o menor de clientes.

En total elegimos 6 instancias, y empezamos eligiendo la primera instancia con 32 clientes hasta llegar a 80 clientes. El formato que tenían las instancias encontradas era el siguiente.

```
NAME : Nombre_de_la_instancia
COMMENT : (Augerat et al, Min no of trucks: x, Optimal value:
   Mejor_valor_obtenido)
TYPE : CVRP
DIMENSION : Numero_de_clientes
EDGE_WEIGHT_TYPE : Calculo_distancia_entre_clientes
CAPACITY : Capacidad_de_vehiculos

NODE_COORD_SECTION
Numero_nodo Coordenada_x Coordenada_y
.
.
.
DEMAND_SECTION
Numero_nodo Demanda_nodo
.
.
.
DEPOT_SECTION
1
-1
EOF
```

Listing 1: Ejemplo de instancia

Como mencionamos, el número de instancias eran 6, y decidimos ejecutar cada una dellas un total de 5 veces, hallando luego la media de los valores

obtenidos para cada uno de los tests.

8.3. Entorno de ejecución

Cabe destacar que, hemos realizado todos los test en la siguiente máquina:

- **Procesador:** AMD Ryzen 7 1700
- **Memoria RAM:** G.Skill 2x8GB 3200mhz CL14
- **Disco duro:** Samsung 960 EVO M.2 256GB

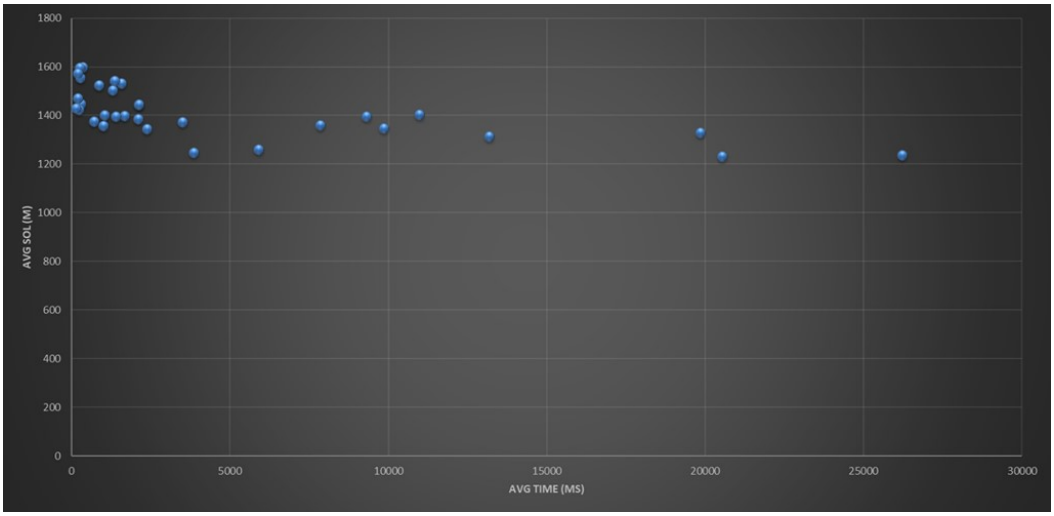
8.4. Comparativa entre parámetros de algoritmos

8.4.1. Greedy Randomized Adaptative Search Procedure

Los parámetros que podíamos pasarle al algoritmo GRASP y las combinaciones que hemos estudiado son:

- Tamaño de la lista restringida de candidatos: {3, 5}
- Máximo número de iteraciones sin mejora: {10, 50}
- Búsqueda Local: {Búsqueda Greedy, Búsqueda ansiosa}
- Movimiento por búsqueda: {Reinserción, Intercambio interruta, Intercambio intraruta, Two Opt}

Las combinaciones posibles si mezclamos éstos parámetros es de **32**. A continuación representamos gráficamente estas combinaciones, donde en el **eje x** está la media del tiempo medio que se ha tardado en resolver todas las instancias con esa combinación de parámetros y en el **eje y** la solución obtenida.



A raíz de estudiar la gráfica, decidimos elegir las mejores soluciones en relación calidad-tiempo. Estas eran las siguientes, donde R.C.L se refiere al tamaño de la lista restringida de candidatos, I.W.I a las iteraciones sin mejora y L.S a la búsqueda local:

1. Grasp con R.C.L = 3, I.W.I = 10, L.S: BN, Move: Relocation
2. Grasp con R.C.L = 3, I.W.I = 10, L.S: FBN, Move: Relocation
3. Grasp con R.C.L = 5, I.W.I = 10, L.S: BN, Move: Relocation
4. Grasp con R.C.L = 5, I.W.I = 10, L.S: FBN, Move: Relocation

Tras ésto decidimos compararlos utilizando el test de **Friedman** con $\alpha = 0{,}005$, lo que nos resultó con un p-value de 0.0009.

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
1º	6	777.247	2005.956	1246.688	530.220
2º	6	809.120	2150.139	1344.737	581.746
3º	6	778.363	2042.540	1260.599	543.166
4º	6	815.017	2225.116	1374.103	600.934

Cuadro 1: Test Friedman

Por lo tanto, debemos rechazar la hipótesis nula de que las muestras vienen de la misma población y aceptar que vienen de poblaciones distintas. Dado que el p-value es muy bajo (0.0009) vemos que hay una diferencia

significativa entre estas cuatro muestras, por lo que decidimos compararlas utilizando el test de **Wilcoxon** a pares.

Finalmente comparamos las dos mejores utilizando el test de Wilcoxon:

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
1º	6	777.247	2005.956	1246.688	530.220
3º	6	778.363	2042.540	1260.599	543.166

Cuadro 2: Test Wilcoxon

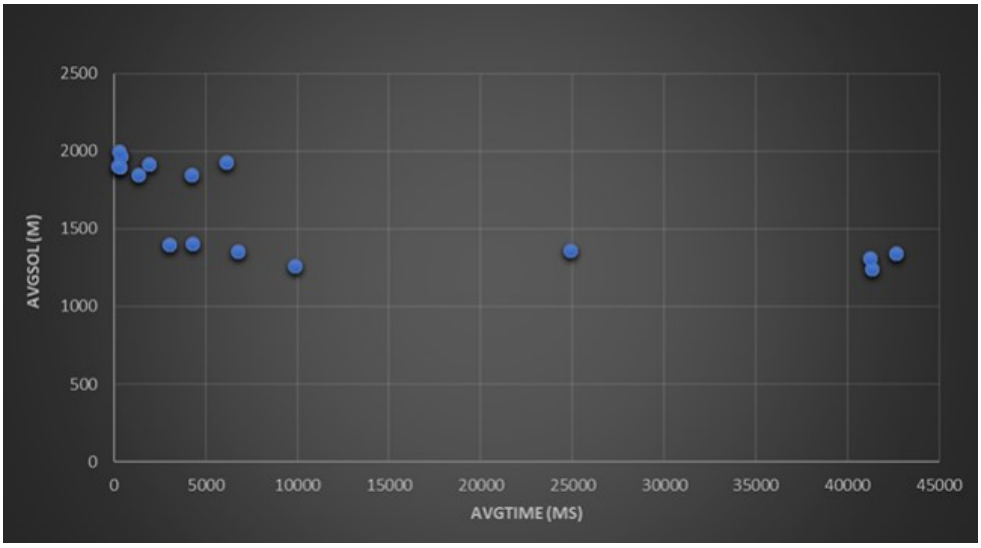
Tras aplicar el test de **Wilcoxon** nos daba un p-value de 0.688, por lo que podemos elegir cualquiera de ellas con una probabilidad de acertar que vienen de la misma población de un 68.75 %. No obstante, decidimos considerar la primera como la mejor por la diferencia de tiempo con la segunda, pues tener una lista restringida de candidatos de tamaño 5 es computacionalmente mas costoso que de tamaño 3.

8.4.2. Método multiarranque

Los parámetros de entrada posibles para el método multiarranque y las combinaciones de los mismos que hemos probado son:

- Máximo número de iteraciones sin mejora: {10, 50}
- Búsqueda Local: {Búsqueda Greedy, Búsqueda ansiosa}
- Movimiento por búsqueda: {Reinserción, Intercambio interruta, Intercambio intraruta, Two Opt}

Las combinaciones posibles si mezclamos estos parámetros son de 16. A continuación, representaremos gráficamente estas combinaciones, donde en el **eje x** está la media del tiempo medio que se ha tardado en resolver todas las instancias con esa combinación de parámetros y en el **eje y** la solución obtenida.



Tras observar detenidamente la gráfica podemos ver que se han formado 3 clusters, cogeremos el que mas cerca está del origen de coordenadas, pues significará una relación calidad-tiempo mejor. Las combinaciones pertenecientes a ese cluster son:

1. Multiarranque con I.W.I = 10, L.S: BN, Move: Relocation
2. Multiarranque con I.W.I = 10, L.S: BN, Move: Interroute Swap
3. Multiarranque con I.W.I = 10, L.S: FBN, Move: Relocation
4. Multiarranque con I.W.I = 10, L.S: FBN, Move: Interroute Swap

Para determinar que solución es mejor, aplicamos el test de **Friedman**, resultando la siguiente tabla, con un p-value de 0.004.

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
1º	6	762,082	2016,178	1263,961	543,747
2º	6	866,020	2224,006	1404,127	548,293
3º	6	803,991	2165,863	1355,148	588,425
4º	6	866,737	2200,517	1397,668	551,964

Cuadro 3: Test Friedman

Por lo tanto, debemos **rechazar la hipótesis nula H0** y aceptar la alternativa Ha, que dice que las muestras no vienen de la misma población. En

este caso, hicimos el **test de Wilcoxon** por cada par de combinaciones hasta resultar con éste par, que eran las dos soluciones que mejores resultados daban:

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
1º	6	762,082	2016,178	1263,961	543,747
2º	6	866,020	2224,006	1404,127	548,293

Cuadro 4: Test Wilcoxon

El resultado del test de Wilcoxon fue a**ceptar la hipótesis alternativa Ha con un p-value de 0,031**, por lo que sigue habiendo una diferencia significativa. Dado que la media de la 1º combinación es mucho mejor y su desviación típica menor, elegimos ésta como mejor solución.

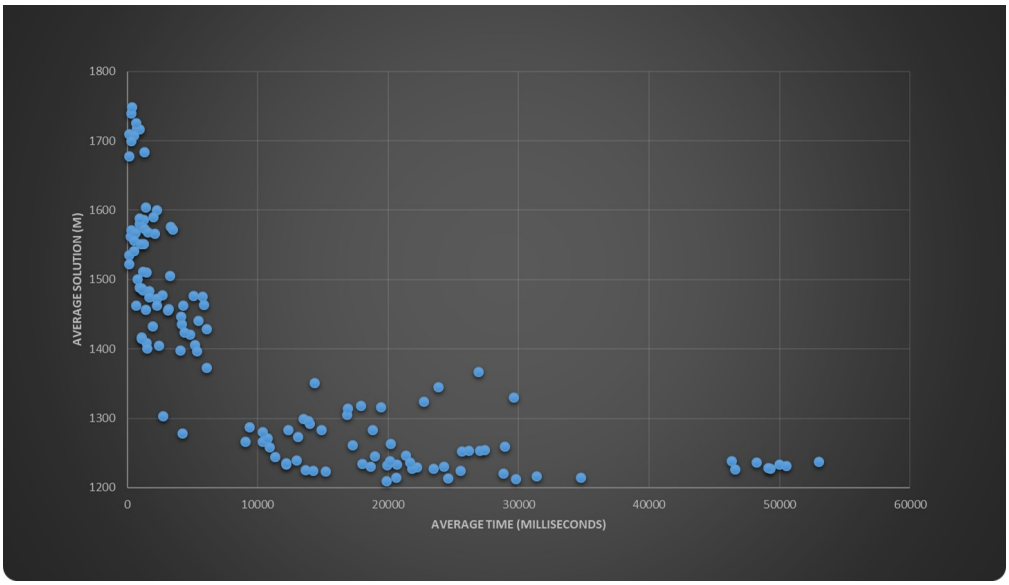
8.4.3. Variable Neighborhood Search con GRASP fase constructiva

El siguiente algoritmo a comparar es el VNS, no obstante vamos a dividirlo en dos apartados. El primero será VNS utilizando de solución inicial el **GRASP en fase constructiva** y, por tanto, probando que combinaciones obtienen una mejor solución. El segundo será el VNS utilizando el **GRASP con la mejor configuración** que obtuvimos en su comparativa.

En éste primer apartado, las combinaciones posibles para el VNS son:

- Tamaño lista restringida de candidatos: {3, 5}
- Máximo número de iteraciones sin mejora: {10, 50}
- Búsqueda local: {Búsqueda Greedy, Búsqueda ansiosa, VND}
- Movimientos para búsqueda local: {Reinserción, Intercambio interruta, Intercambio intraruta, Two Opt}
- Movimientos para shaking VNS: {Intercambio intraruta + Two Opt, Intercambio intraruta + Intercambio interruta + Relocation + Two Opt, Intercambio interruta + Relocation + Intercambio intraruta + Two Opt}

El número posible de combinaciones es de **132**. A continuación las representamos gráficamente, donde en el **eje x** está la media del tiempo medio que se ha tardado en resolver todas las instancias con esa combinación de parámetros y en el **eje y** la solución obtenida.



Tras observar el gráfico detenidamente podemos ver tres puntos con muy buena solución en poco tiempo. Las combinaciones a las que atienden éstos puntos son:

1. VNS con R.C.L: 5, I.W.I: 10, LS Moves: BN + Relocation, Shaking Moves: Interroute + Relocation + Intraroute + TwoOpt
2. VNS con R.C.L: 3, I.W.I: 5, LS Moves: BN + Relocation, Shaking Moves: Intraroute + TwoOpt
3. VNS con R.C.L: 5, I.W.I: 10, LS Moves: BN + Relocation, Shaking Moves: Intraroute + TwoOpt

Para ver qué combinación es estadísticamente mejor hacemos el test de Friedman con esas tres combinaciones de parámetros y nos da un p-valor de 0.115.

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
1º	6	771.467	1990.099	1266.602	524.115
2º	6	772.968	2005.865	1278.894	524.107
3º	6	811.236	2057.984	1303.825	545.829

Cuadro 5: Test Friedman

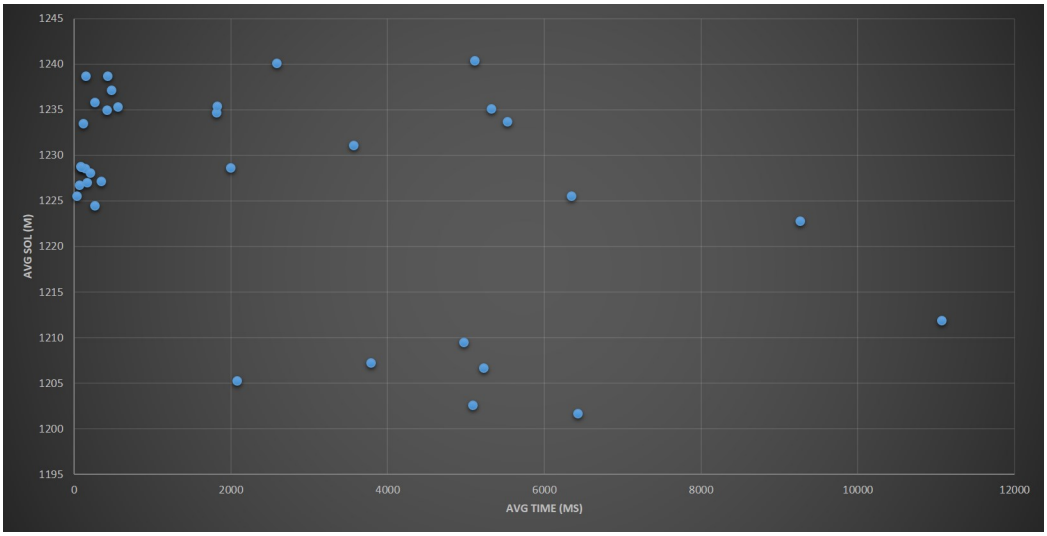
Por lo tanto, no podemos rechazar la hipótesis nula, así que las muestras vienen de la misma población, da igual cuál consideremos mejor pues tienen el mismo comportamiento a nivel estadístico. No obstante, podemos concluir que el p-value no es muy alto porque la 3º combinación se aleja del resto, es de peor calidad frente a las dos primeras.

8.4.4. Variable Neighborhood Search con mejor configuración GRASP

Como habíamos comentado, el segundo apartado del VNS que analizaremos será utilizando **como solución inicial la mejor configuración del GRAS**P que obtuvimos durante su comparativa. Los parámetros y combinaciones posibles del VNS son:

- Búsqueda local: {Búsqueda Greedy, Búsqueda ansiosa, VND}
- Movimientos para búsqueda local: {Reinserción, Intercambio interruta, Intercambio intraruta, Two Opt}
- Movimientos para shaking VNS: {Intercambio intraruta + Two Opt, Intercambio intraruta + Intercambio interruta + Relocation + Two Opt, Intercambio interruta + Relocation + Intercambio intraruta + Two Opt}

El número posible de combinaciones es de **36**. A continuación las representamos gráficamente, donde en el **eje x** está la media del tiempo medio que se ha tardado en resolver todas las instancias con esa combinación de parámetros y en el **eje y** la solución obtenida.



Si observamos detenidamente la gráfica, podemos ver que existen varios puntos que consiguen muy buenos resultados en poco tiempo. Concretamente, se puede observar que hay **5**. Estos se corresponden con la siguiente combinación de parámetros:

1. VNS con LS: VND, LS Moves: Intraroute + Interroute + Relocation + TwoOpt, Shaking Moves: Intraroute + TwoOpt
2. VNS con LS: VND, LS Moves: Interroute + Relocation + Intraroute + TwoOpt, Shaking Moves: Intraroute + TwoOpt
3. VNS con LS: VND, LS Moves: Interroute + Relocation + Intraroute + TwoOpt, Shaking Moves: Intraroute + Interroute + Relocation + TwoOpt
4. VNS con LS: VND, LS Moves: Intraroute + Interroute + Relocation + TwoOpt, Shaking Moves: Interroute + Relocation + Intraroute + TwoOpt
5. VNS con LS: VND, LS Moves: Interroute + Relocation + Intraroute + TwoOpt, Shaking Moves: Interroute + Relocation + Intraroute + TwoOpt

Para ver qué combinación es estadísticamente mejor haremos el test de Friedman con esas cinco combinaciones.

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
1º	6	752,460	1912,737	1205,297	502,928
2º	6	746,302	1897,270	1207,294	497,771
3º	6	757,693	1884,073	1206,692	489,458
4º	6	749,373	1872,780	1202,620	494,757
5º	6	752,954	1916,262	1209,497	506,294

Cuadro 6: Test Friedman

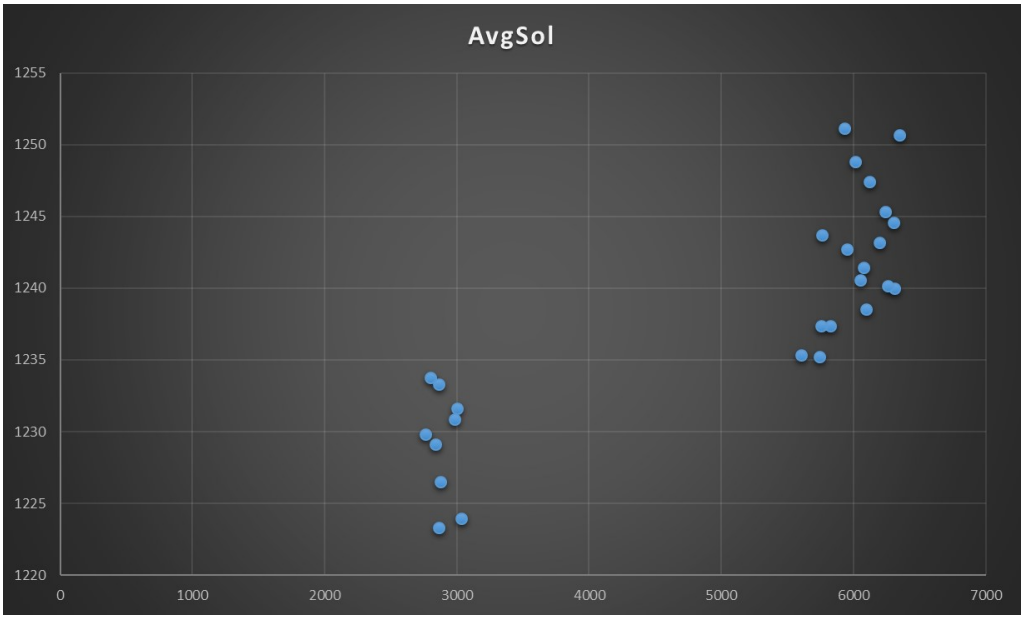
El test de Friedman nos da como resultado un **p-value de 0.785**, por lo que no podemos rechazar la hipótesis nula y las muestras son estadísticamente similares con un 78.47 % de probabilidad.

8.4.5. Tabu Search

El siguiente algoritmo a analizar es la Búsqueda Tabú. Cabe destacar que, para este algoritmo, partimos de la **solución inicial con la mejor configuración encontrada para el GRASP** explicada anteriormente. Los parámetros que se le pueden pasar y los valores que probamos son:

- Máximo tiempo para mejorar (M.T.I): {500ms, 1000ms, 2000ms}
- Porcentaje Tabu Tenure (T.T.P): {15 %, 20 %, 25 %}
- Lista Movimientos: {InterrouteSwap + Two Opt, IntrarouteSwap + InterrouteSwap + Relocation + Two Opt, InterrouteSwap + Relocation + IntrarouteSwap + Two Opt}

Las combinaciones resultantes son un total de 27. A continuación las representamos gráficamente, donde en el **eje x** está la media del tiempo medio que se ha tardado en resolver todas las instancias con esa combinación de parámetros y en el **eje y** la solución obtenida.



Como se puede observar en la gráfica se forman dos clústers muy bien divididos. Para analizar qué combinación es estadísticamente mejor seleccionamos el primer clúster, pues sus soluciones son mucho mejores respecto al segundo. Las combinaciones de este clúster son:

1. Tabu Search con TTP: 15, M.T.I: 500, MOVES: Intraroute + TwoOpt
2. Tabu Search con TTP: 15, M.T.I: 1000, MOVES: Intraroute + TwoOpt
3. Tabu Search con TTP: 15, M.T.I: 2000, MOVES: Intraroute + TwoOpt
4. Tabu Search con TTP: 20, M.T.I: 500, MOVES: Intraroute + TwoOpt
5. Tabu Search con TTP: 20, M.T.I: 1000, MOVES: Intraroute + TwoOpt
6. Tabu Search con TTP: 20, M.T.I: 2000, MOVES: Intraroute + TwoOpt
7. Tabu Search con TTP: 25, M.T.I: 500, MOVES: Intraroute + TwoOpt
8. Tabu Search con TTP: 25, M.T.I: 1000, MOVES: Intraroute + TwoOpt
9. Tabu Search con TTP: 25, M.T.I: 2000, MOVES: Intraroute + TwoOpt

Para comparar las muestras estadísticamente haremos el test de Friedman con las 9 muestras:

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
1º	6	761,325	1943,015	1229,813	518,569
2º	6	751,662	1940,135	1223,321	516,325
3º	6	755,531	1958,905	1231,632	521,155
4º	6	754,945	1962,141	1233,762	521,722
5º	6	755,333	1963,277	1229,110	521,304
6º	6	760,218	1960,577	1226,495	521,218
7º	6	758,333	1973,045	1233,298	526,285
8º	6	755,226	1949,328	1223,948	516,205
9º	6	751,846	1946,877	1230,855	515,643

Cuadro 7: Test Friedman

El p-value resultante del test de **Friedman** es de 0.198. Por lo que no se puede rechazar la hipótesis nula, lo que significa que, estadísticamente, no hay una diferencia significativa entre las 9 muestras. No obstante, dado que el p-value no es demasiado grande, decidimos hacer el test de **Wilcoxon** a pares hasta finalmente quedarnos con el par mas prometedor:

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
2º	6	751,662	1940,135	1223,321	516,325
8º	6	755,226	1949,328	1223,948	516,205

Cuadro 8: Test Wilcoxon

Tras hacer el test de **Wilcoxon** con ese par, nos da un p-value de 0.219. Por lo que las muestras tienen más probabilidad de pertenecer a la misma distribución. No obstante, para ayudar a decidirnos entre una combinación mejor, decidimos usar el factor tiempo y hacer un test de Wilcoxon. La tabla resultante fue:

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
2º	6	1561,000	6017,000	2863,333	1769,234
8º	6	1636,000	6583,000	3032,667	1923,528

Cuadro 9: Test Wilcoxon

El test de Wilcoxon nos da un p-value de 0.688, por lo que, de nuevo, no se puede rechazar la hipótesis nula H0 de que las muestras vienen de la misma

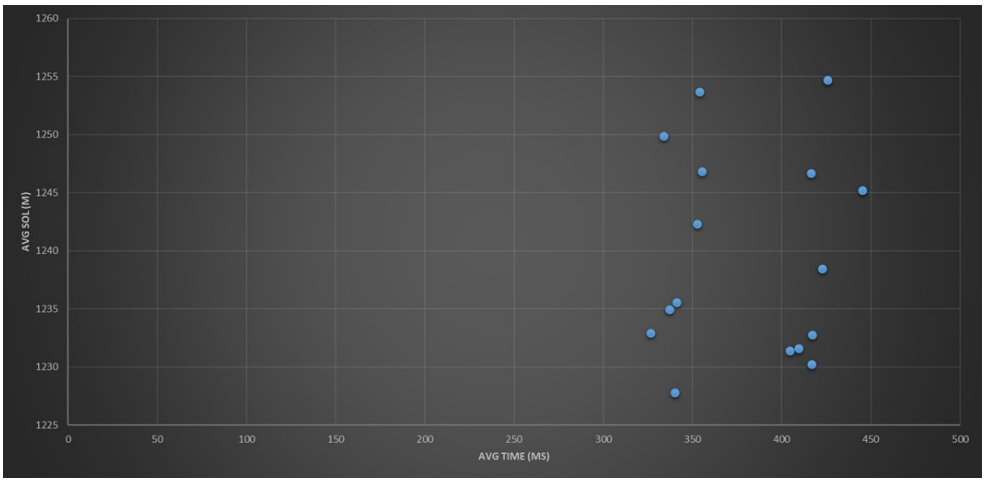
población. No obstante sí vemos una diferencia significativa en la media de tiempo, así que consideramos como mejor combinación de la Tabu Search la 2º variable.

8.4.6. Large Neighborhood Search

Por último, en lo que a comparativa de parámetros de algoritmos se refiere, analizaremos qué combinación para el LNS es mejor. Cabe añadir que para este algoritmo, **partimos de la solución inicial que nos daba la mejor configuración encontrada para el GRASP**, explicada anteriormente. Los parámetros que el LNS podía recibir y las combinaciones que usamos fueron:

- Porcentaje de destrucción (D.P): {20 %, 25 %}
- Diferencia mínima para destruir (M.D): {5}
- Reconstrucciones (M.R): {100}
- Búsqueda local: {Búsqueda Greedy, Búsqueda ansiosa, VND}
- Movimiento por búsqueda: {Reinserción, Intercambio interruta, Intercambio intraruta, Two Opt}

El total de las combinaciones elegidas para el LNS es de **16**. A continuación las representamos gráficamente, donde en el **eje x** está la media del tiempo medio que se ha tardado en resolver todas las instancias con esa combinación de parámetros y en el **eje y** la solución obtenida.



Como podemos observar se han formado 4 clusters. Sin embargo, existe uno compuesto por 4 combinaciones que supera a los demás considerablemente tanto en calidad de solución como en tiempo empleado en encontrarla. Estas combinaciones de parámetros del LNS son:

1. LNS con DP: 0.2, MD: 5, MR: 100, LS: BN + Intraroute
2. LNS con DP: 0.2, MD: 5, MR: 100, LS: BN + Two Opt
3. LNS con DP: 0.2, MD: 5, MR: 100, LS: FBN + Intraroute
4. LNS con DP: 0.2, MD: 5, MR: 100, LS: FBN + TwoOpt

Para averiguar qué combinación resulta estadísticamente mejor hicimos el test de Friedman. Lo que nos dio como resultado la siguiente tabla.

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
1º	6	762.226	1971.066	1234.941	520.279
2º	6	759.501	1942.358	1235.549	517.093
3º	6	767.340	1951.985	1227.784	516.403
4º	6	759.763	1984.877	1232.916	526.567

Cuadro 10: Test Friedman

El resultado del test de Friedman fue de un p-value de 0.659, por lo que hay mucha probabilidad de que las muestras vengan de la misma población y, por lo tanto, no podemos rechazar la hipótesis nula H0. Como conclusión podemos elegir cualquiera de las 4 combinaciones en relación de calidad pues se comportan estadísticamente de manera similar.

No obstante, si nos fijamos en la media que consiguen vemos que la 3º combinación consigue las mejores soluciones, por lo que se podría considerar la mejor combinación de parámetros para el LNS.

8.5. Comparativa entre algoritmos

En este apartado, compararemos la mejor combinación de parámetros de los algoritmos obtenidos, esto se hará atendiendo a los siguientes 3 criterios:

- Mejor algoritmo para obtener la solución inicial.
- Mejor algoritmo para obtener la mejor solución en relación calidad-tiempo.
- Mejor algoritmo para obtener la mejor solución lo mas rápido posible.

8.5.1. Mejor algoritmo para obtener la solución inicial

El primer criterio que utilizaremos será qué algoritmo es mejor para calcular la solución inicial, respecto tanto a calidad como a tiempo. En este apartado sólo entran dos en juego: GRASP y método multiarranque.

El primer paso será coger las mejores combinaciones de cada uno que habíamos visto en la experiencia computacional, éstas son:

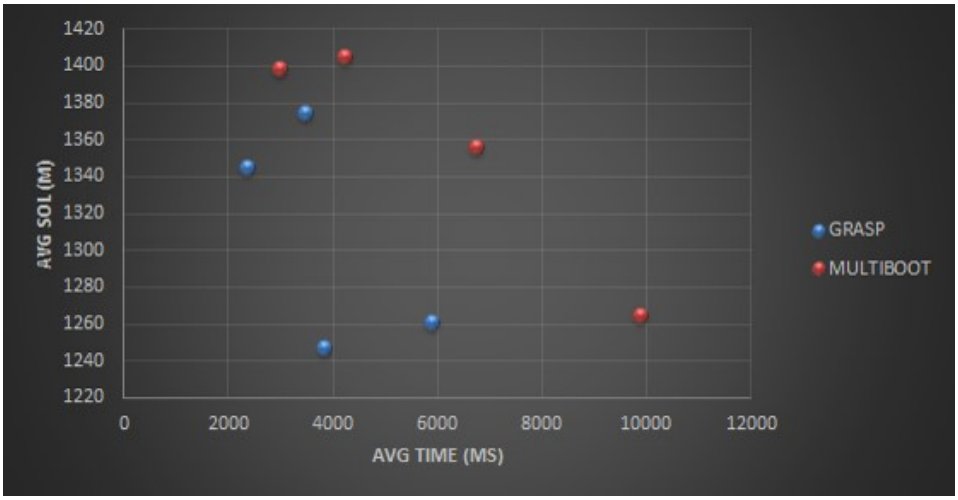
Variable	Observations	Minimum	Maximum	Mean	Std. deviation
1º	6	777.247	2005.956	1246.688	530.220
2º	6	809.120	2150.139	1344.737	581.746
3º	6	778.363	2042.540	1260.599	543.166
4º	6	815.017	2225.116	1374.103	600.934

Cuadro 11: Algoritmo GRASP

Variable	Observations	Minimum	Maximum	Mean	Std. deviation
1º	6	762,082	2016,178	1263,961	543,747
2º	6	866,020	2224,006	1404,127	548,293
3º	6	803,991	2165,863	1355,148	588,425
4º	6	866,737	2200,517	1397,668	551,964

Cuadro 12: Algoritmo Multiarranque

A continuación las representamos gráficamente, donde en el **eje x** está la media del tiempo medio que se ha tardado en resolver todas las instancias con esa combinación de parámetros y en el **eje y** la solución obtenida.

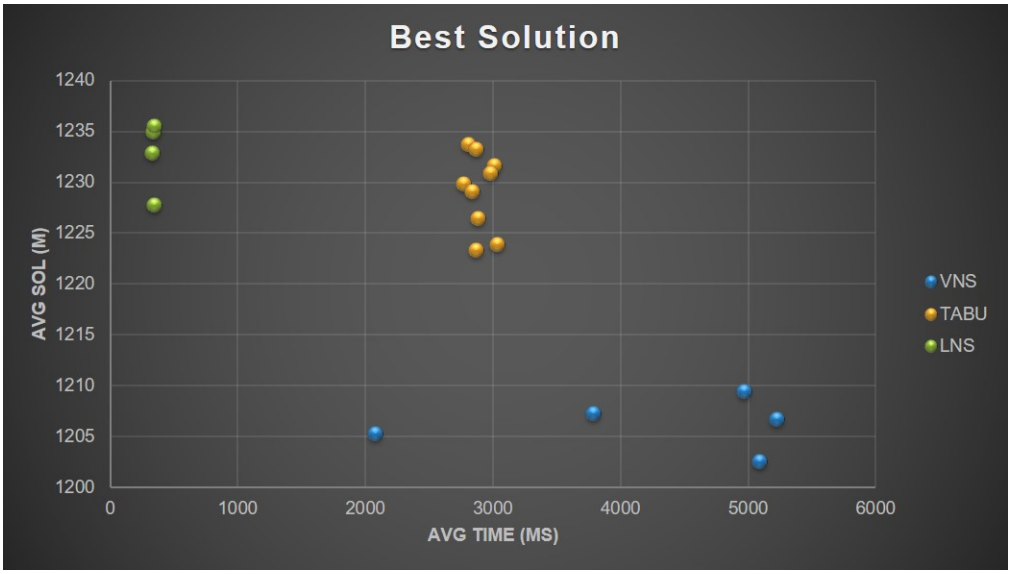


Como podemos apreciar, **las mejores soluciones obtenidas para GRASP fueron sensiblemente mejores que las que se obtuvieron al usar el método Multiarranque**, tanto en aspectos de calidad como de tiempo empleado para obtener la solución. Por lo tanto, para obtener la solución inicial nos decantaremos por el algoritmo GRASP.

8.5.2. Mejor algoritmo para obtener la mejor solución en relación calidad-tiempo

El siguiente criterio será determinar qué algoritmo resulta mejor para mejorar la solución inicial que obtuvimos del GRASP. En éste apartado nos decantaremos por el mejor algoritmo en relación calidad-tiempo. **Intervienen los algoritmos VNS, LNS y Tabu Search.**

Si representamos gráficamente las mejores combinaciones obtenidas de cada algoritmo, como habíamos visto anteriormente, obtenemos el siguiente gráfico. Cabe añadir que en el **eje x** está la media del tiempo medio que se ha tardado en resolver todas las instancias con esa combinación de parámetros y en el **eje y** la solución obtenida.



La mejor solución que hemos obtenido fue la que construimos con VNS, así que **en el aspecto de relación calidad el VNS es el claro ganador**. Sin embargo esta no fue la que mejor tiempo nos proporcionó, que sin duda fue LNS y, aunque no se obtuvieron soluciones tan buenas como con VNS, sí que aportaba muy buenas soluciones en poco tiempo.

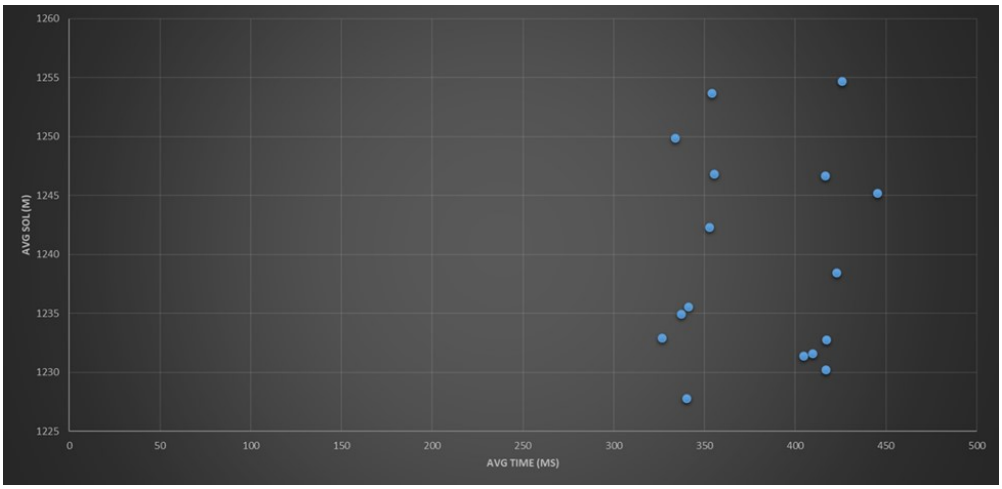
En cuanto a la Tabu Search cabe mencionar que para las pruebas que realizamos, el tiempo que tomó en hallar una solución se mantuvo mas o menos estable, superando en la mayoría de las ocasiones al algoritmo VNS, sin embargo, en cuanto a la calidad de las soluciones no pudo superarlo.

Por tanto, si tuviéramos que clasificar a los algoritmos según la relación calidad-tiempo, tendríamos problemas para decidir si colocar en primer lugar a VNS o LNS, pues la búsqueda tabú obtuvo soluciones parecidas a la LNS, pero tomaron más tiempo para ser calculadas, así que el ranking respecto a calidad de la solución quedaría de la siguiente forma:

1. **LNS:** Tiempo medio: 350 ms, soluciones medias: [1227 - 1236]
2. **VNS:** Tiempo medio: 3.500 ms, soluciones medias: [1205 - 1209]
3. **Tabú Search:** Tiempo medio: 2.800 ms, soluciones medias: [1222 - 1229]

8.5.3. Mejor algoritmo para obtener la mejor solución lo mas rápido posible

Como mencionamos anteriormente, para este caso **el mejor algoritmo sin lugar a dudas resultó ser el LNS**, pues obtuvo tiempos inferiores a medio segundo para obtener la mejor solución, algo que puede resultar especialmente útil si lo que buscamos es obtener una solución inmediata en un determinado conflicto sin descuidar tampoco, en exceso, la calidad de la solución.



9. Conclusiones generales

Una de las conclusiones extraídas de la experimentación es que importa mucho la solución inicial que se genere, pues determinará en parte, qué tan buena solución pueda encontrar una determinada búsqueda local. Si se parte de una solución inicial con una función objetivo alta, en la mayoría de casos, la búsqueda local es capaz de mejorarla, pero no se acerca tanto a la óptima. Sin embargo, cuando se genera una buena solución inicial, la búsqueda es capaz de mejorarla y acercarse bastante a la óptima.

A la hora de calcular la nueva distancia resultante de efectuar un movimiento en nuestra solución inicial, hemos visto que el calcularla de forma parcial reduce considerablemente el tiempo de ejecución, pues en un principio puede parecer que realizar unas cuantas operaciones más no influirá demasiado en el tiempo de ejecución. Sin embargo, cuando se ejecutan muchas veces, puede resultar un lastre. Es por ello que decidimos calcularla de forma

parcial, teniendo en cuenta la diferencia que había entre la distancia total de la solución anterior y la nueva solución.

Cabe destacar que la codificación de la solución ha influido notablemente a la hora de calcular las posibles soluciones vecinas, pues el haber decidido codificarla así **influyó positivamente en el rendimiento**, al ser un simple vector de rutas separadas por el separador '-1'.

Sin embargo, el hecho de hacer un diseño modular utilizando varias clases, generará un mayor desperdicio de recursos. No obstante, hemos priorizado hacer un diseño que facilitara incluir nuevos algoritmos de búsqueda local y movimientos, sin tener que tener en cuenta el trabajo realizado anteriormente por otras personas, siguiendo de esta manera **el principio Open-Closed** de los principios SOLID.

10. Posibles líneas futuras

En primer lugar y a rasgos generales, como posibles líneas futuras, se debería seguir probando diferentes combinaciones de parámetros para cada algoritmo, con el fin de obtener mejores soluciones en un tiempo más reducido. En algunos de los algoritmos utilizados se han combinado pocos parámetros por falta de tiempo en el proyecto.

A su vez se podría investigar utilizar un **pool de soluciones para la Tabu Search** y combinar estas soluciones entre sí. Esto se haría si no hemos encontrado una solución mejor explorando el entorno. Si al combinar las soluciones del pool conseguimos obtener una solución de mejor calidad, se podría utilizar como siguiente solución para la búsqueda local, en vez de elegir una solución aleatoria como hacíamos. De esta manera, se mejorarían mucho los resultados obtenidos y tardaría menos tiempo.

Por último, un resultado interesante de la experiencia computacional fue el comportamiento del **algoritmo LNS**, pues como vimos fue el que mejores soluciones nos dio en un tiempo muy reducido. Podríamos permitir que tardase un tiempo mayor, probar un número mayor de combinaciones entre sus parámetros y así poder obtener soluciones de mayor calidad, similares a las obtenidas usando el **algoritmo VNS** pero en un tiempo mucho menor.

Referencias

Bianchi, L. (2000). Notes on dynamic vehicle routing: The state of the art.

de Malaga, U. (2018). Capacitated vrp instances. http://neo.lcc.uma.es/vrp/vrp-instances/capacitated-vrp-instances/.

Feo, T. A. and Resende, M. G. C. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133.

Glover, F. and Laguna, M. (1997). *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA.

Hansen, P. and Mladenović, N. (2001). Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449 – 467.

Johnson, D. S. and McGeoch, L. A. (1995). The traveling salesman problem: A case study in local optimization.

Li, F., Golden, B., and Wasil, E. (2005). Very large-scale vehicle routing: new test problems, algorithms, and results. *Computers and Operations Research*, 32(5):1165 – 1179.

Reinelt, G. (1994). *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag, Berlin, Heidelberg.

Shaw, P. (1998). *Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Yepes, V. (2002). *Optimización heurística económica aplicada a las redes de transporte del tipo VRPTW*.