Inteligencia artificial
Práctica de búsqueda local

Jia Long Ji Qiu
Shuang Long Ji Qiu
You Wu

Curso 2021/2022 Q1

Índice

1. Descripción del problema

1.1. Introducción

El objetivo de esta práctica es resolver un problema de planificación de rutas de abastecimiento mediante el uso de algoritmos de búsqueda local.

En el problema que se nos plantea, una compañía de distribución de combustible desea planificar rutas diarias para poder abastecer ciertas gasolineras con tal de que estas sigan funcionando. Dicha planificación debe optimizar el beneficio obtenido bajo unas ciertas condiciones, las cuales detallaremos durante los siguientes apartados.

1.2. Análisis del problema

La compañía tiene *n* centros de distribución repartidos geográficamente dentro de un área de 100 x 100 km^2, donde también se encuentran las gasolineras.

Cada uno de estos centros tiene uno o más camiones para realizar la distribución de combustible, teniendo cada camión el doble de la capacidad de almacenamiento de cada gasolinera. Además, los camiones tendrán las siguientes restricciones:

- Los camiones pueden hacer un máximo de *v* viajes por día, definidos como un recorrido de ida y vuelta desde el centro al que está asignado.
- En cada viaje se pueden servir hasta 2 peticiones, debido a la limitación de almacenaje.
- La distancia que pueden recorrer los camiones queda limitada a *k* kilómetros diarios.
- Cada kilómetro que recorre un camión tiene un coste de 2.

Una gasolinera solicita diariamente una lista de peticiones las cuales tendremos que asignar a los camiones de los distintos centros, para que estos las sirvan haciendo un recorrido en concreto. Cada uno de los depósitos que tiene una gasolinera tiene un valor de 1000 de base.

El beneficio que obtendremos por cada petición atendida dependerá de los días que lleva la petición sin atender. Si la petición lleva 0 días, es decir, se ha solicitado el mismo día, la compañía cobrará el 102% del precio del depósito, si no, el porcentaje de reducción que se le aplica al precio se calcula con la siguiente fórmula:

$$\%precio = (100 - 2 * días)\%$$

La distancia entre dos puntos A y B, ya sea para determinar el recorrido desde un centro hasta una gasolinera, o de una gasolinera a otra, se calcula a partir de la fórmula:

$$d(A, B) = |A_x - B_x| + |A_y - B_y|$$

A partir de estos datos podemos extraer qué es lo que realmente afectará a nuestro objetivo de optimización: para todo un conjunto de peticiones, hemos de servir aquellas peticiones que nos aporte el máximo de ganancias, priorizando las que llevan menos tiempo pendiente, y reduciendo lo máximo posible el gasto producido por el recorrido que implica servirla. Además, hemos de procurar cumplir en todo momento todas las restricciones impuestas sobre los camiones: el límite diario de viajes y distancia recorrida por viaje, y la cantidad de peticiones que se pueden servir con un único viaje. Por último, también nos interesa tener en cuenta las pérdidas que supondría no atender una petición.

Como podemos observar, dada la complejidad del problema, sería complicado intentar aplicar un algoritmo cualquiera y que el resultado de este realmente sea bueno. Por este motivo, utilizar un algoritmo de búsqueda local es una buena idea, ya que, partiendo de una solución inicial arbitraria, podemos ir optimizándola mediante pequeños cambios para aumentar lo máximo posible el beneficio final.

2. Adaptación del problema a la búsqueda local

2.1. Representación del estado del problema

2.1.1. Implementación

A la hora de decidir cómo representar el estado del problema, es decir, cómo parametrizar la solución, primero hemos de identificar qué es lo relevante en una solución para este problema.

Está claro que lo que nos interesa saber es, para cada camión, qué viajes debe realizar y cómo. Esto implica que necesitamos analizar, desde un principio, cuál es la situación del mundo, es decir, dónde se sitúan los centros de distribución y las gasolineras, las distancias que hay entre estas, y qué peticiones hay pendientes a ser servidas.

Además, con tal de respetar todas las restricciones del problema, también debemos estar informados en todo momento sobre las circunstancias de cada camión, es decir, sus viajes y kilómetros restantes.

Para definir exactamente cómo es el conjunto de viajes que hace un camión, basta con añadir la información de qué peticiones ha servido, identificando la gasolinera que la solicitó.

Por lo tanto, forman parte de nuestro estado los siguientes elementos:

- *CentrosDistribucion centros*

Representa una *ArrayList* de centros de distribución. El único método que ofrece es la constructora, la cual genera un determinado número de centros de distribución con unas coordenadas aleatorias. Además, si se desea asignar más de un camión a un centro, basta con tener réplicas de este centro (asumimos que los camiones pertenecientes a centros con mismas coordenadas pertenecen en realidad a un mismo centro).

Este objeto nos permitirá saber dónde están ubicados los centros de distribución a partir de sus coordenadas, las cuales necesitaremos para calcular distancias.

- *Gasolineras gasolineras*

Representa una *ArrayList* de gasolineras. Con un comportamiento parecido a la clase *CentrosDistribucion*, la constructora de la clase *Gasolineras* genera un determinado número de gasolineras con coordenadas aleatorias. Cada gasolinera genera de manera aleatoria sus peticiones, con un cierto número de días que lleva pendiente, también al azar. También podemos obtener las coordenadas de estas y consultar su lista de peticiones, representada como una *ArrayList* de enteros.

Al igual que los centros, este objeto también nos permitirá saber las ubicaciones de las gasolineras con tal de calcular las distancias necesarias.

- *ArrayList<Peticion> peticionesPendientes*

Representa una *ArrayList* de las peticiones pendientes de todas las gasolineras.

Peticion es una clase auxiliar que hemos creado para poder guardar la información de una petición, es decir, para guardar cuántos días lleva pendiente a ser servida y a qué gasolinera pertenece. Ofrece una constructora la cual crea un objeto *Peticion* dada una gasolinera (identificada con un índice) y los días que lleva pendiente. También ofrece los métodos para obtener dichos atributos.

Esta estructura nos permitirá trabajar más cómodamente a la hora de asignar o desasignar peticiones, equivalente a eliminar y añadir a la lista, ya que añade todas las peticiones presentes en un día dentro de un mismo conjunto, en lugar de tener que acceder a cada instancia de gasolinera y obtener de allí sus peticiones. También facilita el cálculo del beneficio teórico, un concepto que explicaremos más adelante, necesario para satisfacer una de las condiciones de nuestro objetivo. Consultar cuántas peticiones quedan por ser servidas también se vuelve trivial.

- *ArrayList<Viaje>[] viajes*

Representa los viajes que hace el camión de un centro determinado. Cada fila de la matriz contiene una *ArrayList* de *Viaje* que hace un camión.

Viaje es una clase auxiliar que hemos creado para poder guardar la información de un viaje, concretamente guardamos una *ArrayList* de *Peticion* y los kilómetros que se hacen. Ofrece una constructora que crea un *Viaje* dada una petición; las consultoras para obtener la lista de peticiones de ese viaje y los kilómetros; y una modificadora que sirve para añadir una petición a la lista. Esta última nos permitirá asignar de manera estructurada una segunda petición, una vez ya se ha asignado la primera.

Esta estructura es el núcleo de nuestra representación. Representa de manera descriptiva e intuitiva cada viaje, indicando la ruta que debe seguir el camión al que se le ha asignado.

- *int[][] distanciaCenGas*

Esta matriz nos permite guardar la distancia entre los centros de distribución y las gasolineras, de esta manera, no será necesario repetir ningún cálculo de distancias, pues siempre tendremos acceso a estos en tiempo constante.

Para un escenario en el que solo se asigna un camión por centro, la estructura es óptima ya que todas las distancias son distintas y por lo tanto no se puede aprovechar ninguna simetría.

- *ArrayList<Integer>[] distanciaGasGas*

Esta matriz también nos permite guardar distancias, pero esta vez entre gasolineras. A diferencia de la matriz de distancias entre centros y gasolineras, en este caso sí que podemos aprovechar la

simetría de las distancias entre gasolineras y elegir una estructura que nos permita optimizar lo máximo posible su complejidad espacial.

La estructura más optima que se nos ha ocurrido ha sido un vector de *ArrayList*, de manera que, para cada gasolinera con índice g_i, se guardará la distancia entre esta gasolinera y las gasolineras con un índice mayor a g_i. Por lo tanto, si queremos acceder a la distancia entre dos gasolineras g_i y g_j, si los índices son iguales, entonces la distancia será cero (no se guarda en esta matriz), en caso contrario, el acceso será el siguiente:

$$distGasGas[\min(g_i, g_j)].get(\max(g_i, g_j) - \min(g_i, g_j) - 1)$$

- *int[] kmRestantes*

Guarda los kilómetros restantes que le queda por recorrer a un camión. Este vector nos permitirá hacer un seguimiento en todo momento de si realmente podemos asignar o no una cierta petición, lo cual hace que sea una estructura necesaria para cumplir siempre las restricciones impuestas.

- *double beneficioTeorico*

Indica el beneficio del problema en un mundo ideal, donde el coste por kilómetros recorridos sería cero, y podemos asignar todas las peticiones pendientes.

Todas las estructuras se han implementado para garantizar una máxima optimalidad espacial. Aquellas estructuras cuyos datos se mantienen igual a lo largo de la ejecución se han declarado como variables *static*.

2.1.2. Análisis del espacio de búsqueda

El espacio de búsqueda está formado por todas las soluciones posibles que hay para un problema y depende de la representación del estado. En nuestro caso, el espacio de búsqueda viene determinado por todas las permutaciones posibles entre el número total de peticiones, asignadas a cada centro. Por lo tanto, este espacio de búsqueda será de tamaño $O(p^c)$, donde p es el número total de peticiones y c el número de centros de distribución.

2.2. Estrategias de generación de la solución inicial

La generación de la solución inicial nos define el punto de partida a partir del cual se aplica el algoritmo de búsqueda local. Por este motivo, es importante definir distintas estrategias de generación para que, mediante la experimentación, podamos determinar cuál de estas nos permite hallar la mejor solución. Así pues, las distintas estrategias que hemos implementado han sido:

- Asignación mediante fuerza bruta

La asignación de peticiones mediante fuerza bruta consiste en asignar, de manera ordenada por cada camión, el máximo de peticiones posible.

El algoritmo consiste en iterar sobre cada camión una única vez de manera que, para cada camión, recorriendo la lista de peticiones pendientes, se vayan asignando nuevos viajes hasta que, o ya no tenga suficientes kilómetros restantes para coger más peticiones, o haya alcanzado el límite de viajes diarios, o ya no queden peticiones pendientes.

Estos viajes se crean a partir de la asignación de una primera petición, una vez se asegure que el camión puede hacer una ida y vuelta desde su centro hasta la gasolinera que la ha solicitado. Tras ser asignada, se recorre el resto de las peticiones para comprobar si en el mismo viaje se puede servir la segunda, calculando la distancia de la nueva ruta que tendría que recorrer, y asegurando que sea menor o igual a los kilómetros restantes que le quedan al camión.

Las peticiones que han podido ser asignadas se eliminan de la lista de peticiones pendientes, y tras cada viaje nuevo asignado, los kilómetros restantes del camión se actualizan según la distancia que se necesita recorrer en dicho viaje.

El coste de este algoritmo es $O(\#c * \#v * \#p) = O(\#c * \#p)$, donde *#c* es el número de centros de distribución, *#v* el límite de viajes diarios y *#p* el número de peticiones pendientes. Como que el límite de viajes diarios es una constante, no lo tendremos en cuenta en la notación.

- Asignación por amplitud

La asignación por amplitud consiste en una asignación ordenada de peticiones por niveles, es decir, para volver a comprobar si se puede asignar una petición más a un centro, antes se ha de haber intentado asignar una vez al resto de centros.

La implementación de este algoritmo es muy parecida a la de la asignación mediante fuerza bruta solo que, en este caso, una vez se consigue asignar un viaje nuevo a un camión, se procede a iterar sobre los siguientes camiones.

Un detalle relevante sobre este algoritmo es que, si en una iteración sobre un centro no se ha asignado ningún viaje nuevo, significa ya no se le podrá asignar ninguna petición más, así que no se vuelve a tener en cuenta dicho centro y de esta manera evita hacer recorridos innecesarios.

En este caso el coste también es $O(\#c * \#p)$, no hay ninguna diferencia notable respecto a la asignación mediante fuerza bruta.

- Asignación aleatoria por amplitud

Esta generación es exactamente igual a la asignación por amplitud, solo que se mezclan las peticiones previamente.

Mezclar una lista tiene un coste lineal, por lo que la complejidad temporal es también la misma.

- Asignación por amplitud y distancia mínima

La asignación por amplitud y distancia mínima también es muy parecida a la asignación únicamente por amplitud, pero esta vez se prioriza para cada centro aquellas peticiones cuya distancia necesaria para servirlas es mínima.

La diferencia de este algoritmo respecto al de la asignación únicamente por amplitud es que, a la hora de intentar crear un nuevo viaje, la petición candidata a formar parte de este es siempre aquella petición cuya distancia necesaria para ser servida es mínima para el centro en el que se esté asignando. Por lo tanto, es necesario hacer un recorrido lineal sobre todas las peticiones pendientes para hallar dicha petición. A la hora de intentar asignar la segunda petición también se aplica la misma estrategia.

Este cambio hace que la complejidad temporal de este algoritmo incremente un poco, pero bajo la notación asintótica sigue siendo $O(\#c * \#p)$.

En cuanto a la bondad de cada una de las estrategias, tanto la asignación mediante fuerza bruta como por amplitud dependerán mucho del orden en la que se generan las peticiones,

La asignación aleatoria por amplitud puede ser interesante debido a que permite ejecutar el algoritmo varias veces y, al tener siempre un punto de partida distinto, los resultados variarán y se podrá hacer una selección de aquellas soluciones que más beneficio dan.

Por último, la asignación por amplitud priorizando distancias mínimas permitirá acercarse mucho a un máximo local (si hablamos de beneficio neto), por lo que hallar una buena solución no requerirá mucho tiempo. Sin embargo, es posible que no podamos explorar otras soluciones que quizás dan un beneficio mayor.

2.3. Operadores

2.3.1. Análisis

La selección de operadores debe permitir explorar una gran parte del espacio de soluciones. Teniendo esto en cuenta, nos interesaría poder permutar todas las combinaciones posibles de la asignación de peticiones. Por lo tanto, hemos pensado en tres operadores; el intercambio entre dos asignaciones (ya sea entre peticiones ya asignadas o entre asignadas y pendientes), la asignación de peticiones pendientes y la cancelación de un viaje con peticiones ya asignadas.

Los intercambios entre asignaciones nos permitirán reducir la distancia recorrida. Además, con tal de complementar este operador, se utiliza el operador de asignación que, en realidad, no garantiza que se pueda asignar, sino que intenta asignar peticiones pendientes tras cada intercambio, ya que no se puede asegurar que realmente se reduzca la distancia recorrida. Por último, el operador de cancelar viajes permite comprobar si un viaje realmente aporta algún beneficio.

También se ha tenido en cuenta que se podría implementar otro operador que consistiese en mover una asignación de un centro a otro, pero el impacto que tendría sería prácticamente negligible, debido a que nos interesaría para aquellos casos en las que se terminan se asignar todas las peticiones, pero mayoritariamente trabajamos sobre estados donde siempre quedan peticiones pendientes.

2.3.2. Implementación

Así pues, la implementación de nuestros operadores es:

- *swapPeticionesAsignadas(int c1, int v1, int p1, int c2, int v2, int p2)*

Hace un intercambio entre dos peticiones *p1* y *p2* de ciertos viajes *v1* y *v2* que ya han sido asignadas entre los centros *c1* y *c2*, respectivamente. Es un operador que nos interesa utilizar para poder reducir la distancia recorrida en un viaje.

La condición de aplicabilidad es que, tras el intercambio, ninguno de los centros exceda el límite de kilómetros que se les ha impuesto. Su factor de ramificación es $O(\frac{\#p_a(\#p_a-1)}{2})$, donde $\#p_a$ es el número de peticiones asignadas, ya que prueba todas las combinaciones posibles sin repetir ninguna.

- *swapPeticionesAsignadaYNoAsignada(int c, int v, int p)*

Realiza un intercambio entre una petición que ya ha sido asignada a un camión y una petición que está en la lista de peticiones pendientes. Este operador, al igual que el anterior, también permite reducir la distancia recorrida en un viaje.

La condición de aplicabilidad es que, tras el intercambio, el camión no exceda el límite de kilómetros. El factor de ramificación en este caso es $O(\#p_a * \#p_{!a})$, donde $\#p_{!a}$ el número de peticiones sin asignar, ya que para cada posición asignada haríamos cambios con todas las peticiones sin asignar.

- *intentarAsignarPeticiones(int c)*

Utiliza fuerza bruta para intentar asignar el máximo de peticiones pendientes a un camión. Si el camión ya tiene viajes definidos, primero se comprueba si se les pueden acoplar una segunda petición. Si el camión aún no ha alcanzado el límite de viajes, entonces se le intentará también asignar viajes nuevos con una o dos peticiones.

La condición de aplicabilidad es que, tras asignar todas las peticiones posibles, el camión no exceda el límite de viajes y de kilómetros recorridos. Para este caso, el factor de ramificación es $O(\#centros)$, pues el escenario que tendríamos sería una en la que cada sucesor tiene un centro distinto con el máximo de asignaciones posibles para este.

- *quitarViaje(int c, int v)*

Se elimina el viaje *v* del centro *c*. Las peticiones que estaban asignadas en este viaje vuelven a la lista de peticiones pendientes.

A pesar de que, con las condiciones iniciales dadas en el problema, este operador no es de gran ayuda, si el coste por kilómetro recorrido pudiera aumentar, sí que tomaría relevancia ya que sí que se puede llegar a dar el caso en el que una asignación nos genere pérdidas. Veremos más detalles sobre su utilidad en el apartado de experimentación.

Este operador no tiene ninguna condición de aplicabilidad, ya que el estado nuevo que resulta de haber quitado un viaje sigue siendo una solución válida. En cuanto al factor de ramificación, este sería $O(\#viajes\ asignados)$ ya que, a partir de un estado, los sucesores que puede generar son como máximo *#viajes asignados* estados, donde en cada estado sucesor falta un viaje (distinto en cada sucesor).

Cabe mencionar que, cada operador, además de hacer su función principal, también debe encargarse de mantener actualizados los parámetros de la representación del estado: la lista de peticiones pendientes, los kilómetros y viajes restantes del centro que ha sufrido cambios.

2.3.3. Función sucesora

La estrategia que utilizaremos para generar los estados sucesores será iterar sobre cada petición *p* asignada a un viaje *v* de un centro *c*, aplicando los operadores de una manera determinada. Un primer conjunto de sucesores será el obtenido aplicando la operación de quitar un viaje. Los siguientes serán los resultantes de haber hecho intercambios de *p* con otras peticiones, tanto asignadas como pendientes. Finalmente, un último conjunto se obtiene mediante el operador de intentar asignar nuevas peticiones.

2.4 Elección de la función heurística

A partir del análisis del problema hemos considerado la calidad de nuestra solución depende de la ganancia y el coste de recorrido total. Por este motivo, las heurísticas que hemos implementado se basan en distintas maneras de combinar estos dos factores.

- Heurística de maximización de beneficio bruto (*GasolinaHeuristicFunction1*)

Maximiza la ganancia total generada por todos los camiones. Es el sumatorio de las ganancias generadas por cada camión, y no se tiene en cuenta el coste generado por la distancia recorrida.

- Heurística de minimización del coste del recorrido (*GasolinaHeuristicFunction2*)

Minimiza el recorrido total hecho por todos los camiones. Es el sumatorio de los kilómetros hechos por los camiones multiplicado por el coste.

- Heurística de maximización del beneficio neto (*GasolinaHeuristicFunction3*)

Maximiza el beneficio neto generado por todos los camiones. Se calcula como el sumatorio de la diferencia entre el beneficio bruto y el coste por kilómetro recorrido de cada camión.

- Heurística de maximización del beneficio neto respecto al beneficio ideal (*Gasolina HeuristicFunction4*)

Maximiza el beneficio neto respecto al beneficio ideal. El beneficio ideal es el beneficio máximo que podríamos obtener en un mundo donde se pueden asignar infinitas peticiones y no hay coste por kilómetro recorrido, este se calcula como el sumatorio del valor de todas las peticiones generadas. Es un coeficiente que tiene en cuenta el beneficio neto generado y la pérdida que supone no atender ciertas peticiones.

A priori, parece que las heurísticas que mejor definirán la calidad de los estados serán la tercera y la cuarta, debido a que lo que nos interesa maximizar en este problema es el beneficio neto. Ambas heurísticas permiten considerar si realmente vale la pena servir una petición, ya que estos tienen en cuenta el coste por kilómetro recorrido.

La heurística que maximiza el beneficio bruto lo más probable es que también dé buenos resultados, pero seguramente no tan buenos como con las dos anteriores porque ya no se tiene en cuenta el coste. El comportamiento que tendría el algoritmo sería priorizar las peticiones que más valor tienen (es decir, aquellas que llevan menos tiempo pendiente), sin tener en cuenta la distancia necesaria para servirla, lo cual implicaría que el beneficio que se cree haber generado, realmente no será tan alto una vez aplicado el coste.

En cuanto a la heurística de reducción del coste por kilómetro recorrido, pese a tener un mínimo de lógica, es en realidad una función que no nos aportará nada a la hora de hacer la búsqueda local. Después de todo, para reducir al máximo este coste bastaría con un estado donde ningún camión se mueve.

3. Experimentación

3.1. Influencia de los operadores (Experimento 1)

A la hora de pensar qué operadores utilizar de los propuestos para explorar el espacio de soluciones, pensamos que el modus operandi más adecuado sería ir añadiéndolas uno tras otro, para comprobar si realmente tienen algún impacto en la solución final. Por lo tanto, cada combinación a comprobar consiste en la inserción de uno de los operadores definidos:

- Combinación 1: hacer únicamente intercambios entre peticiones asignadas.
- Combinación 2: hacer intercambios entre peticiones asignadas y entre asignadas y no asignadas.
- Combinación 3: hacer intercambios entre peticiones e intentar asignar nuevas.

En este experimento no tendremos en cuenta la operación de quitar viajes ya que, bajo las condiciones de este, nunca será necesario quitar un viaje (el coste por kilómetro recorrido no supone ningún problema), pero de todas formas está presente en cada una de las combinaciones.

Para comprobar la eficacia de estos tres operadores, y que realmente son todos necesarios, hemos realizado un experimento bajo las siguientes condiciones:

- La solución inicial que hemos escogido ha sido la de asignar peticiones de manera voraz, es decir, asignando el máximo de peticiones posibles a cada centro con un solo recorrido sobre los centros.
- La función heurística de beneficio tiene en cuenta las pérdidas que supone no atender una petición.
- Tenemos un escenario en el que el número de centros es 10, hay un camión por centro y el número de gasolineras es 100.
- El algoritmo de búsqueda local utilizado ha sido *Hill Climbing*.

A la hora de proceder, hemos probado las distintas combinaciones de los tres operadores haciendo cambios en la función sucesora y lo ejecutaremos eligiendo 10 semillas aleatorias para nuestro mundo. Partimos de la hipótesis de que la combinación de los tres operadores es mejor que cualquier otra. Así pues, los resultados que hemos obtenido son los siguientes:

	Beneficio		
Muestra	Combinación 1	Combinación 2	Combinación 3
1	84968	87516	94816
2	77412	79496	95328
3	81628	83180	94672
4	80792	82528	95516
5	85596	88168	95176
6	81672	84124	94720
7	77708	80084	95376
8	76060	77688	94180
9	72900	75552	94896
10	74844	76284	95748
Promedio	79358	81462	95042,8

Tabla 1. Valor del beneficio de la solución, según la combinación de operadores que se ha utilizado.

Como podemos observar en la Tabla 1, la inserción de cada operador siempre nos está aportando una mejora en el beneficio que se obtiene. En promedio, la combinación 2 ha mejorado 2104 respecto la combinación 1, mientras que la combinación 3 ha mejorado 13580 respecto la combinación 2. La segunda combinación no tiene un impacto muy significativo, y el cambio se refleja sobre todo a la reducción del coste del recorrido total, debido a que el impacto que tiene intercambiar las peticiones es pequeño. Sin embargo, la tercera, al permitir hacer más asignaciones, sí que mejora mucho más el beneficio neto.

Con esto podemos confirmar que realmente todos los operadores son útiles, así que, a partir de ahora, en todos los experimentos utilizaremos la misma función sucesora, la cual tiene implementada la combinación 3.

3.2. Influencia del estado inicial (Experimento 2)

Una vez visto en el experimento 1 que la combinación que consiste en usar los operadores propuestos para este problema es el que da mejor resultados, vamos a determinar cuál de todos los estados iniciales que hemos definido es mejor que las otras. Recordemos que las estrategias utilizadas para generar el estado inicial son las siguientes:

- Estado inicial 1: Asigna tantas peticiones como pueda a los camiones (voraz).
- Estado inicial 2: Asigna peticiones por amplitud.
- Estado inicial 3: Asigna de forma aleatoria las peticiones por amplitud.
- Estado inicial 4: Asigna las peticiones más cercanas por amplitud.

El experimento parte de ver si uno de estos estados solución inicial produce mejor resultado que otros. Por lo tanto, esta será nuestra hipótesis alternativa y, por otro lado, la hipótesis nula será que todos los estados dan el mismo resultado.

Las condiciones y la metodología de este experimento son muy parecidas al anterior, solo que en esta ocasión tenemos fijada la función sucesora y se variará la estrategia para hallar la solución inicial. Probaremos cada estrategia sobre 10 semillas distintas bajo las siguientes condiciones:

- La función sucesora tiene implementada la combinación de todos los operadores propuestos.
- La función heurística de beneficio tiene en cuenta las pérdidas que supone no atender una petición.
- Tenemos un escenario en el que el número de centros es 10, hay un camión por centro y el número de gasolineras es 100.
- El algoritmo de búsqueda local utilizado ha sido *Hill Climbing*.

Observación: para calcular el resultado del estado inicial 3, debido a que se trata de una asignación aleatoria, se ha hecho la media de 10 ejecuciones para cada semilla.

	Estado inicial 1			Estado inicial 2		
Muestra	Beneficio	Nodos	Tiempo	Beneficio	Nodos	Tiempo
1	94344	95	2,911	94564	82	2,493
2	94492	89	2,929	94448	74	2,503
3	94644	89	2,236	94896	79	2,143
4	96596	85	2,714	96408	85	2,794
5	95380	86	2,231	95404	84	2,375
6	93964	74	2,109	94392	83	2,449
7	94676	85	2,64	94864	85	2,458
8	94936	88	2,86	94624	77	2,537
9	96212	80	2,479	96344	80	2,571
10	95364	93	3,002	94924	81	2,756
Promedio	95060,8	86,4	2,6111	95086,8	81	2,5079

Tabla 2. Diferencia de beneficio, nodos expandidos y tiempo entre los estados iniciales 1 y 2

	Estado inicial 3			Estado inicial 4		
Muestra	Beneficio	Nodos	Tiempo	Beneficio	Nodos	Tiempo
1	94728,4	89,1	2,703	95080	19	0,9371
2	94582	83,8	2,607	94556	25	1,192
3	95562,4	85	2,489	96332	24	1,071
4	96543,2	87	2,86	97172	20	1,117
5	95236	81,7	2,303	95812	15	0,738
6	94692	88,4	2,688	95120	22	1,097
7	95454,4	86,7	2,676	95696	20	1,01
8	94952,4	85	2,876	95240	25	1,26
9	96479,6	88,1	2,722	97036	24	1,134
10	95423,2	92,1	3,076	95860	18	1,053
Promedio	95365,36	86,69	2,700	95790,4	21,2	1,06091

Tabla 3. Diferencia de beneficio, nodos de expansión y tiempo entre los estados iniciales 3 y 4

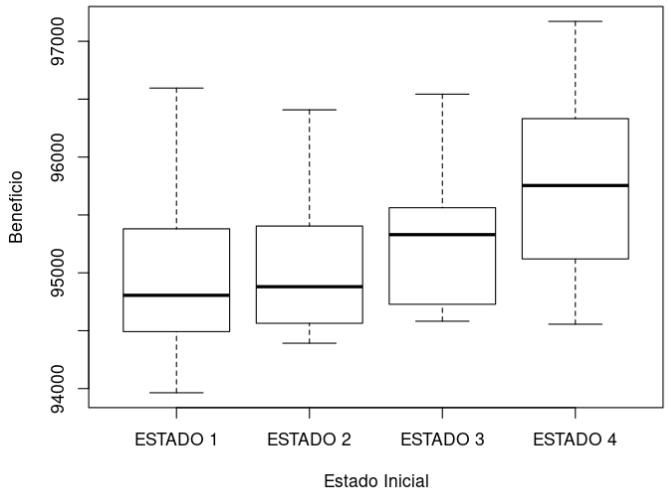

Figura 1. Distribución del beneficio para cada estado inicial

Analizando los resultados en la Figura 1, podemos observar que la diferencia entre las estrategias 1 y 2 no es notable. Tanto el beneficio obtenido como el tiempo de ejecución son muy parecidos, así que sería indistinto elegir entre una de estas dos. Ahora bien, ambos dan peores resultados que la tercera y cuarta estrategia, así que nos centraremos en estas.

La asignación aleatoria a veces nos permite encontrar máximos locales más altos que la de la asignación priorizando las peticiones más cercanas para cada centro. Sin embargo, la ejecución es más lenta ya que explora un espacio de soluciones más grande. A efectos prácticos, la elección entre estas dos estrategias dependerá de si se quiere priorizar el tiempo de ejecución o el beneficio neto, porque la cuarta estrategia ya nos da un resultado bastante bueno, y no debemos olvidar que, para hallar una solución mejor mediante aleatoriedad, es posible que se necesite hacer varias ejecuciones. En nuestro caso, fijaremos la cuarta estrategia para el resto de los experimentos.

3.3. Influencia de la función heurística

Ahora que ya tenemos claro los operadores que utilizaremos para nuestra función sucesora, además de la estrategia para generar la solución inicial, procederemos a estudiar las distintas heurísticas propuestas en un principio. Recordemos que las heurísticas a probar son:

- Heurística 1: maximización del beneficio bruto.
- Heurística 2: minimización del coste de recorrido.
- Heurística 3: maximización del beneficio neto.
- Heurística 4: maximización del coeficiente beneficio neto/beneficio teórico

Hasta ahora, la heurística que habíamos elegido para los experimentos anteriores había sido la heurística 4, ya que es la más adecuada si tenemos en cuenta el criterio de la solución impuesto en el problema. Además, tal y como mencionamos anteriormente, a priori parece que las heurísticas 1 y 2 no serán muy útiles comparados con las otras dos, así que esto es lo que querremos comprobar en este experimento. También estudiaremos el comportamiento de las heurísticas 3 y 4.

Una vez más, el procedimiento consistirá en probar las cuatro heurísticas para diez semillas distintas, bajo las siguientes condiciones:

- La función sucesora tiene implementada todos los operadores propuestos.
- La estrategia para generar la solución inicial es la asignación por amplitud priorizando la distancia mínima para cada centro.
- Tenemos un escenario en el que el número de centros es 10, hay un camión por centro y el número de gasolineras es 100.
- El algoritmo de búsqueda local utilizado ha sido *Hill Climbing*.

	Beneficio			
Muestra	Heurística 1	Heurística 2	Heurística 3	Heurística 4
1	92600	3980	94784	94784
2	92628	2040	96148	96148
3	93808	4040	96568	96568
4	92984	0	95636	95636
5	93056	2040	95680	95680
6	92364	1920	95972	95972
7	92604	0	95880	95880
8	91392	2040	94456	94456
9	91852	6020	95144	95144
10	90480	1900	93244	93244
Promedio	92376,8	2398	95351,2	95351,2

Tabla 4. Comparativa de beneficio según la heurística

Mirando la Tabla 4, a simple vista ya podemos ver que la heurística 2 no es nada útil porque, tal y como comentamos anteriormente, la búsqueda consistiría únicamente en quitar viajes asignados hasta que el recorrido sea 0. Debido a esto, dejaremos de tenerla en cuenta.

La heurística 1 tampoco nos interesa ya que no tiene en cuenta el coste por kilómetro recorrido, provocando de esta manera que el algoritmo se detenga cuando aún no ha hallado el máximo local con el beneficio neto más alto, que es el objetivo del problema.

Finalmente, en el caso de las heurísticas 3 y 4, no hay ninguna diferencia a la hora de comparar el beneficio neto con el coeficiente de beneficio neto respecto al beneficio teórico. De todas maneras, fijaremos la heurística 4 para el resto del experimento, ya que esta se ajusta al criterio de tener en cuenta las pérdidas que supone no atender ciertas peticiones. En la Figura 2 podemos ver de manera más clara la distribución del beneficio según la heurística.

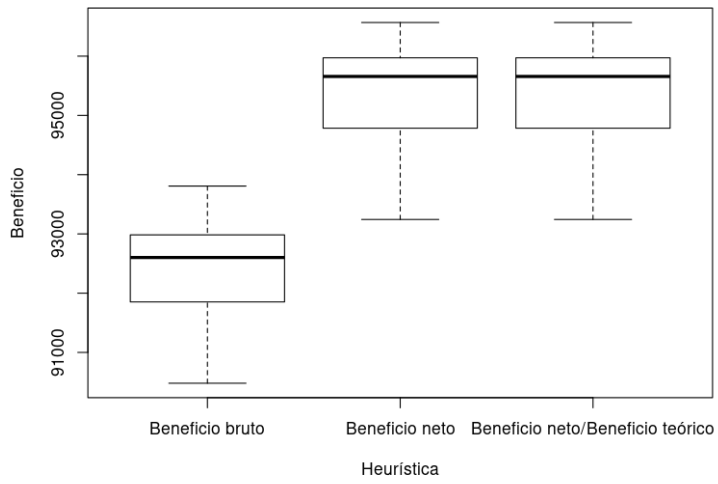

Figura 2. Distribución del beneficio según la heurística

3.4. Simulated Annealing (Experimento 3)

En este experimento averiguaremos cuales son los parámetros más adecuados para el algoritmo de Simulated Annealing. Observaremos si hay un conjunto determinado de parámetros que sea mejor que otros. Los parámetros que tendremos que determinar serán los siguientes:

- Número de iteraciones (*steps*).
- Número de iteraciones durante las cuales el algoritmo mantiene la misma temperatura (*stiter*).
- Los parámetros lambda y k que son variables que se utiliza para la función de aceptación de estados.

Nuestra hipótesis será que podemos encontrar una combinación de los parámetros del Simulated Annealing que nos dé resultados iguales o que dé mejores resultados.

El planteamiento para abordar esta parte del experimento es sencillo: iremos comprobando qué combinación de valores de los parámetros dan mejor resultado.

La metodología es la siguiente:

- Utilizar el estado inicial que previamente hemos determinado como la mejor.
- Utilizar el conjunto de operadores que previamente hemos determinado como la mejor.
- Utilizar la función heurística que cumpla el criterio de la solución del problema.
- Fijar el número de iteraciones en base al beneficio resultante.
- Establecer un rango para el parámetro lambda. En nuestro caso lambda tendrá los siguientes valores: {1, 0.1, 0.01, 0.001, 0.0001}.
- Establecer un rango para el parámetro k. En nuestro caso k tendrá los siguientes valores: {1, 5, 10, 25, 100, 125}.
- Establecer un rango para *stiter*: {1, 10, 50, 100, 200, 500}.
- Probar las combinaciones posibles e ir descartando los peores resultados.

El primer paso será calcular cuantas iteraciones son necesarias, para ellos hemos fijado lambda igual a uno, lambda a 0.1 y *stiter* a 1. La gráfica de la Figura 3 muestra la evolución del beneficio en función del número de iteraciones.

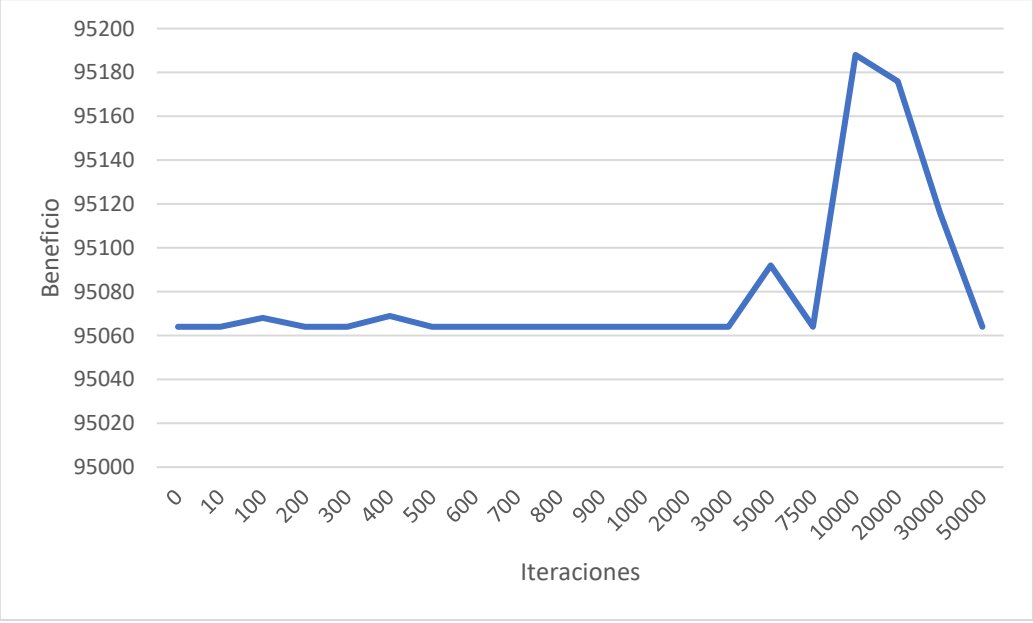

Figura 3. Variación del beneficio según el número de iteraciones

Observamos que en el rango que abarca desde 0 hasta 2000 iteraciones se mantiene a un valor constante de 95064 de beneficio aproximadamente y es a partir de este punto que notamos cambios. Vemos que con 10000 iteraciones el beneficio sube hasta 95186 aproximadamente y después, con iteraciones más altas que 10000 vemos que va bajando. Por lo tanto, tomamos 10000 como el número de iteraciones.

Ahora veremos con qué combinación k-lambda nos quedaremos con la combinación inicial pero ahora fijando 10000 iteraciones vamos a ver todas las combinaciones posibles entre los valores que hemos acotado sobre k y lambda.

En la Figura 4 muestra el beneficio para todas las combinaciones posibles entre k-lambda después de ejecutar el programa par a cada uno de sus valores. Se puede observar que para el mayor beneficio k y lambda toma, respectivamente, valores de 100 y 0.1, por lo tanto, podemos concluir que esta combinación es la mejor.

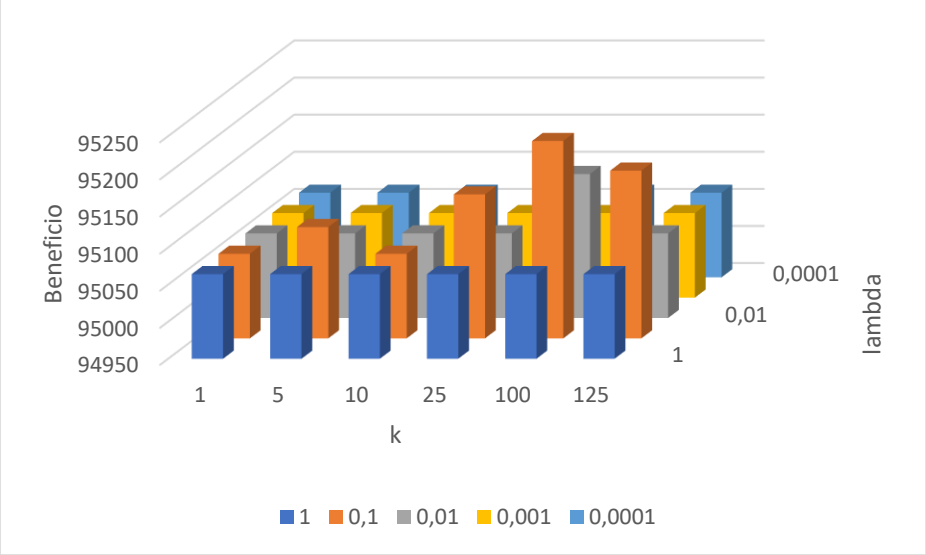

Figura 4. Variación del beneficio según diferentes valores de k y lambda

Visto ya qué número de iteraciones, k y lambda que mejor nos conviene, veremos qué valor debe de tomar *stiter*. Ejecutaremos el programa con cada uno de los valores definidos anteriormente. La gráfica resultante se muestra en la Figura 5. Podemos observar que cuando stiter toma valor 200, este proporciona el mayor beneficio, por lo tanto, podemos tomar 200 como valor para stiter.

Finalmente, obtenemos la siguiente configuración:

- Iteraciones: 10000.
- *Stiter* = 200.
- Lambda = 0,1.
- K = 100

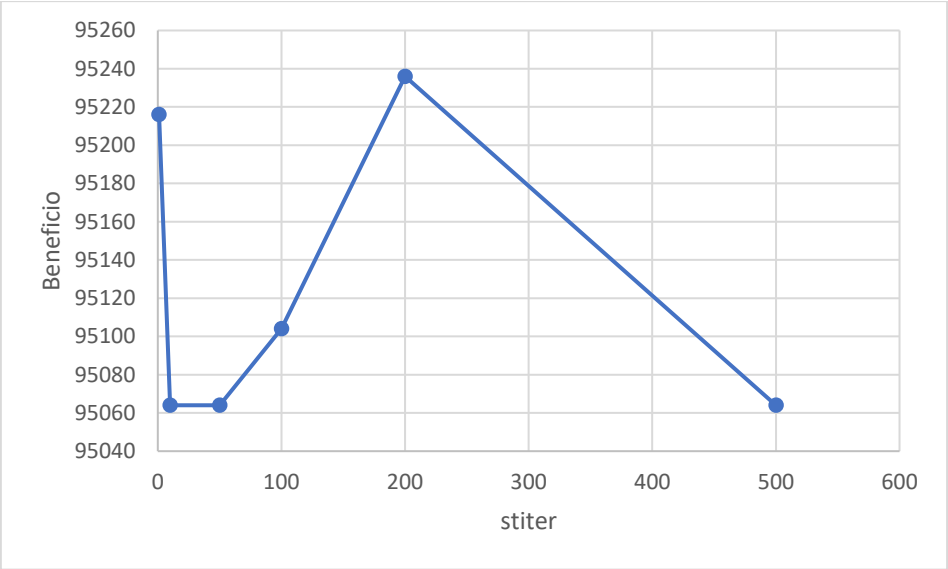

Figura 5. Variación del beneficio variando stiter fijando el número de iteraciones k y lambda

3.5. Comparativa entre Hill Climbing y Simulated Annealing (Experimento 4)

En el experimento 4 se nos pide estudiar la evolución del tiempo de ejecución del programa según el número de centros que definamos para el escenario ejecutándolo tanto con Hill Climbing como con Simulated Annealing. Además, veremos si los parámetros que hemos calculado en el experimento anterior dan buenos resultado aun aumentando el tamaño del problema.

Nuestra hipótesis será que, independientemente de la proporción que definamos par a los parámetros del problema número de centros no hay un cambio en el tiempo de ejecución o, por lo contrario, el tiempo de ejecución sí que se ve afectado, aumentando en proporción al número de centros que haya.

El planteamiento será, con la proporción 10:100, fijaremos el número de gasolineras a 100 e iremos incrementando el número de centros en 10, es decir, empezaremos con 10 centros, luego con 20, luego 30 y así hasta llegar a 100 centros.

La metodología será la siguiente:

- Utilizar el estado inicial que previamente hemos determinado como la mejor opción.
- Utilizar el conjunto de operadores que previamente hemos determinado como la mejor opción.
- Utilizar la función heurística que cumpla el criterio de la solución del problema.
- Elegir 10 semillas aleatorias para cada proporción.
- Elegir Hill Climbing como el algoritmo de búsqueda.
- Ejecutar el programa primero con 10 centros, fijando 100 gasolineras, e ir aumentando el número de centros en 10.
- Elegir 5 semillas aleatorias para cada proporción.

- Elegir Simulated Annealing con los parámetros hallados en el experimento 3.
- Ejecutar el programa primero con 10 centros, fijando 100 gasolineras, e ir aumentando el número de centros en 10.
- Calcular media de ambos casos.

Primero observaremos la evolución del tiempo utilizando Hill Climbing. En la gráfica de la Figura 6 podemos ver claramente que el tiempo de ejecución aumenta a medida que aumenta el número de centros. Esta relación aumenta aproximadamente de manera lineal como indica la línea de tendencia de la gráfica.

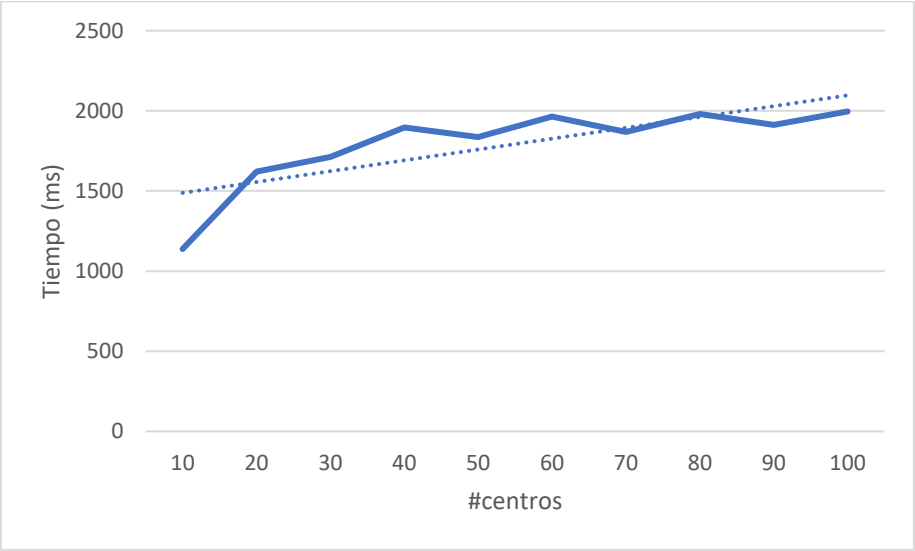

Figura 6. Variación del tiempo respecto al número de centros (ejecución HC)

Igual que antes, observaremos cómo evoluciona el tiempo de ejecución utilizando Simulated Annealing. En la gráfica de la Figura 7 también podemos observar que el tiempo aumenta a medida que aumenta el número de centros. Igual que en el caso del Hill Climbing, la función tiende a aumentar de forma lineal.

Finalmente, en la gráfica de la Figura 8 se puede observar que al aumentar el tamaño del problema no afecta de manera negativa al Simulated Annealing.

Figura 7. Variación del tiempo respecto al número de centros (ejecución SA)

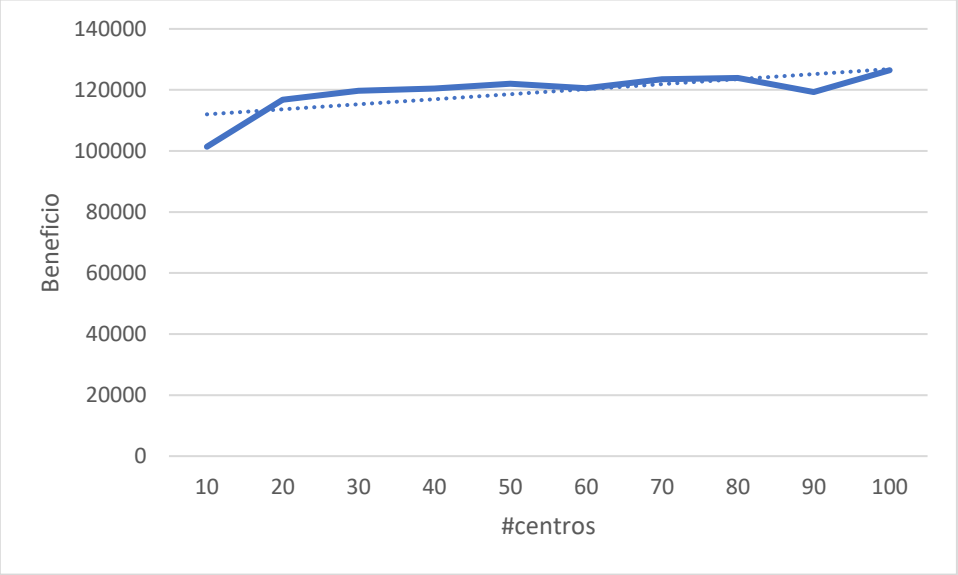

Figura 8. Variación del beneficio respecto al número de centros (ejecución SA)

Respecto a la influencia del tamaño de la entrada (número de centros) sobre el tiempo de ejecución, podemos concluir que es el comportamiento esperado, ya que, al aumentar el tamaño de la entrada, el computo de posibles estados soluciones también aumenta.

Nótese que la diferencia del tiempo de ejecución entre Hill Climbing y Simulated Annealing es notable. Mientras que uno tarda segundos en acabar, SA tarda minutos en terminar la ejecución, esto podría ser una señal de que puede que no hallamos escogido bien los valores para los parámetros a pesar de que estos dan un buen resultado.

3.6. Influencia del número de centros (Experimento 5)

En los experimentos anteriores nos hemos mantenido en un escenario donde teníamos 10 centros de distribución, 1 camión por centro y 100 gasolineras porque asumíamos que tener centros no nos cuesta nada. Por eso ahora vamos a ver qué pasaría si estos centros tuviesen un coste.

En este quinto experimento observaremos cómo influye en el beneficio y los kilómetros recorridos si solamente reducimos los 10 centros a la mitad, es decir, contemplar el escenario donde hay 5 centros, 2 camiones por centro y 100 gasolineras.

Nuestra hipótesis nula será que la acción de reducir a la mitad el número de centros no afectará en absoluto ni al beneficio ni a los kilómetros recorridos o alternativamente (H1), que sí que afectará.

El planteamiento para abordar esta parte será calcular tanto el beneficio como los kilómetros recorridos para el escenario original y el nuevo escenario y comparar uno con el otro.

La metodología es la siguiente:

- Utilizar el estado inicial que previamente hemos determinado como la mejor opción.
- Utilizar el conjunto de operadores que previamente hemos determinado como la mejor opción.
- Utilizar la función heurística que cumpla el criterio de la solución del problema.
- Elegir 10 semillas aleatorias para la ejecución.
- Elegir *Hill Climbing* como el algoritmo de búsqueda.
- Ejecutar el programa para el primer escenario y segundo escenario, una única vez por semilla.
- Calcular media de ambos casos para el beneficio y kilómetros.

En la Tabla 5. Variación del beneficio respecto al número de centros (ejecución SA) podemos apreciar que realmente sí que hay una diferencia entre una configuración y la otra.

Teniendo en cuenta que tenemos 100 gasolineras para los casos, observamos a simple vista que para el mundo donde hay 10 centros y un solo camión por centro da mayor beneficio que el escenario donde tenemos 5 centros y 2 camiones por centro.

Esto ocurre lo mismo en el caso de los kilómetros recorridos, en el primer caso se hacen menos kilómetros que en el segundo. En las gráficas de las figuras Figura 9 y Figura 10 se presencia mejor esta diferencia.

	10 centros, 1 camión por centro		5 centros, 2 camiones por centro	
	Beneficio	Kilómetros	Beneficio	Kilómetros
Media	95387,2	2436,4	94232,8	3002,6

Tabla 5. Variación del beneficio respecto al número de centros (ejecución SA)

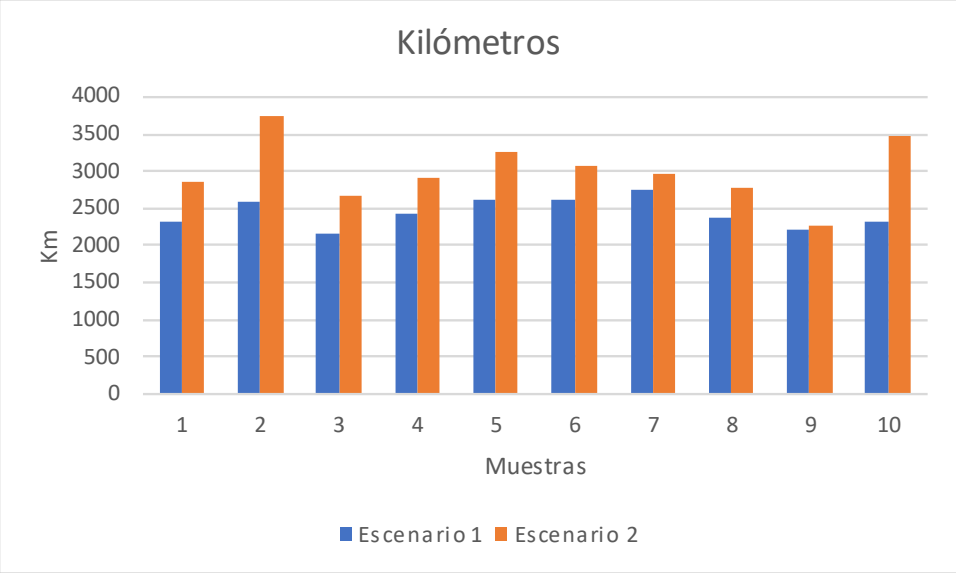

Figura 9. Diferencia de kilómetros recorridos entre el escenario 1 (10-1-100) y 2 (5-2-100)

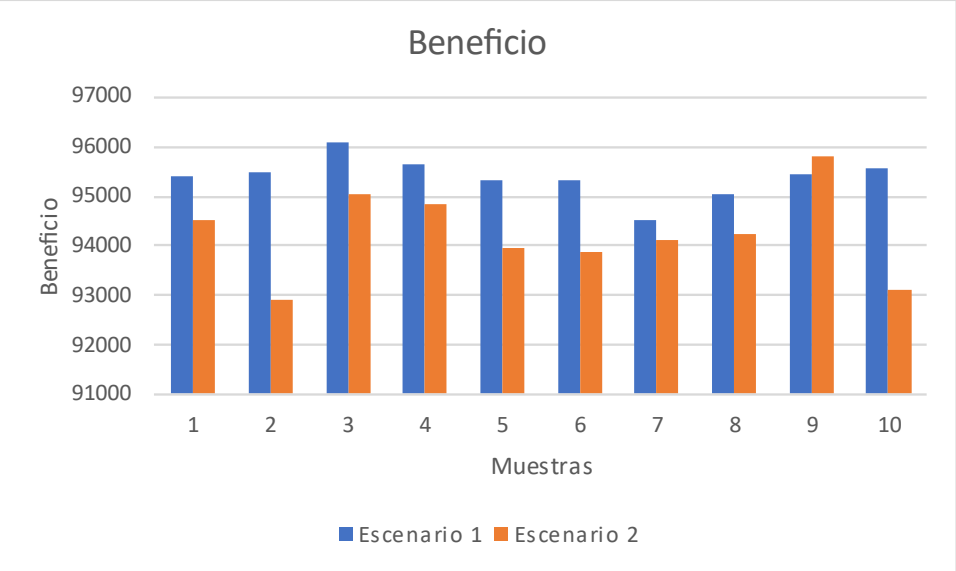

Figura 10. Diferencia de beneficio entre el escenario 1 (10-1-100) y 2 (5-2-100)

Viendo los resultados, podemos intuir que, aunque mantengamos la proporción centro – camión por centro, será más beneficioso si disponemos de más centros de distribución, ya que, si es así, habrá más dispersión en el mundo permitiendo así, tener más posibilidades de disponer centros alternativos que estén más cercanos a las gasolineras.

3.7. Influencia del coste por kilómetro recorrido (Experimento 6)

Análogamente a lo que hemos realizado en experimento 5 donde hemos visto que el cambio de números de centros tiene un efecto en la solución del problema, vamos a observar si tiene algún efecto en las peticiones servidas si el coste por kilómetro va aumentando exponencialmente. Recordemos que el coste que tenemos ahora es 2.

La hipótesis nula (H_0) será que el cambio del coste por kilómetro no tiene ningún efecto en las peticiones que vayamos a servir o, por lo contrario, sí que tiene algún afecte en esto último (H_1).

El planteamiento para este experimento será, para varias semillas, ver cuantas peticiones se hacen a medida que vamos doblando el coste por kilómetro comenzando por coste 2.

La metodología para este experimento es la siguiente:

- Utilizar el estado inicial que previamente hemos determinado como la mejor opción.
- Utilizar el conjunto de operadores que previamente hemos determinado como la mejor opción.
- Utilizar la función heurística que cumpla el criterio de la solución del problema.
- Elegir 10 semillas aleatorias para la ejecución.
- Elegir Hill Climbing como el algoritmo de búsqueda.
- Acotar el coste c por kilómetro $\forall n, n \in [0, 10] : c \in 2^n$.
- Ejecutar el programa tantas veces como semillas haya e ir calculando el número de peticiones que se sirven según los días pendientes.
- Calcular la totalidad de las peticiones que se sirven.

En la gráfica que muestra la Figura 11, podemos observar claramente que el número de peticiones que se sirven sí que se ve afectado por el aumento del coste.

El número de peticiones que se sirven en total disminuye drásticamente en un cierto punto, en este caso, a partir del punto donde hay coste 8, el número de peticiones que se hacen decrece de forma exponencial hasta cierto punto.

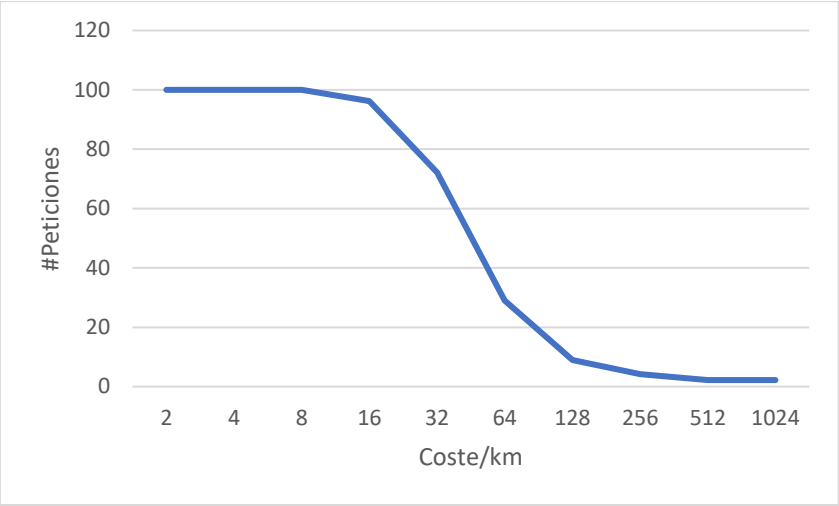

Figura 11. Evolución del número de peticiones según al aumento de coste por km

El resultado que hemos visto es el esperado, ya que al aumentar el coste por kilómetro afecta en nuestra heurística que determina si vale o no la pena servir una petición. Tal como hemos descrito nuestra heurística (ver sección 2.4), esta descartará servir una petición si el coste que se genera al servir dicha petición no sale rentable.

3.8. Influencia del límite de kilómetros (Experimento 7)

En este último experimento, tendremos que comprobar en qué afecta, si es que tiene repercusión, modificar las horas de trabajo de los camioneros sobre el beneficio. Recordemos que en el escenario original las horas máximas que puede hacer un camión al día es de 8h.

Nuestra hipótesis nula (H_0) será que la modificación de las horas de trabajo ya sea disminuyéndolo o aumentándolo no influenciará en el beneficio obtenido o alternativamente (H_1) que sí hay una influencia.

El planteamiento será ir modificando el límite de horas diarias e ir obteniendo los beneficios generados.

La metodología para este experimento es la siguiente:

- Utilizar el estado inicial que previamente hemos determinado como la mejor opción.
- Utilizar el conjunto de operadores que previamente hemos determinado como la mejor opción.
- Utilizar la función heurística que cumpla el criterio de la solución del problema.
- Elegir 10 semillas aleatorias para la ejecución.
- Elegir Hill Climbing como el algoritmo de búsqueda.
- Modificar el límite de horas diarias
- Ejecutar el programa tantas veces como semillas haya para 10 horas h ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Podemos pensar a priori que modificar el límite de horas diarias que puedes hacer sí que afecta en el beneficio ya que limita los kilómetros por día. En el caso de disminuir las horas implica disminuir el número de viajes que se pueda hacer, por ende, se servirán menos peticiones. Pero ahora, ¿qué pasa si aumentáramos las horas? Esto último lo discutiremos con el resultado se refleja en la gráfica de la Figura 12.

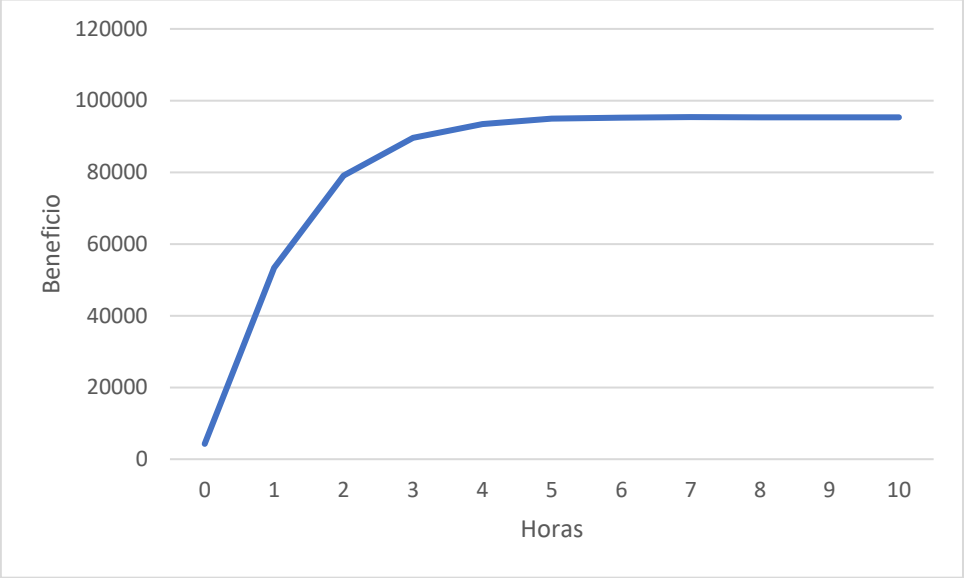

Figura 12. Beneficio según el límite de horas diarias

Observando el resultado, podemos concluir que, como ya esperábamos, si disminuyéramos las horas de trabajo esto podría afectar negativamente en el beneficio. En cambio, aumentar las horas de trabajo no tiene ninguna repercusión en el beneficio a partir de cierto punto, esto es debido a que, si con una cierta hora ya podemos conseguir el mayor beneficio, a partir de ese límite, el beneficio también será la máxima.

4. Conclusiones

A lo largo de esta práctica, hemos podido observar cómo los algoritmos de búsqueda local nos permiten facilitar hallar soluciones suficientemente buenas para problemas complejos, como es la organización de una ruta de distribución de bienes. Sin embargo, cabe destacar que, a la hora de representar un problema con estos algoritmos, es muy importante hacer un buen diseño para facilitar la ejecución de estos.

Con la parte experimental, hemos observado que las variaciones de algunos parámetros pueden influir a gran medida la solución del problema ya sea de forma negativa o positiva. Así mismo, también nos ha ayudado a determinar que tanto la representación que hemos dado al problema como los operadores y heurísticas han sido correctos.

Dicho esto, podemos decir que los algoritmos de búsqueda son una muy buena herramienta para problemas donde se requiera una solución lo suficientemente buena, pero no se busque específicamente la optimalidad.