# Optimizando un sistema de compartición de vehículos mediante algoritmos de búsqueda local

Arnau Esteban, Fernando Gómez, Fernando Guirao

1 de mayo de 2023

Inteligencia Artificial 2022-2023 Q2

**Resumen**

Este informe está realizado sobre la práctica de algoritmos de búsqueda local de la asignatura de Inteligencia Artificial del grado en Ingeniería Informática de la Facultad de Informática de Barcelona (FIB) de la Universitat Politècnica de Catalunya (UPC).

Se propone un sistema de compartición de vehículos para personas que se desplazan en coche a su puesto de trabajo. Los usuarios apuntados al servicio podrán ser llevados en coches de otros usuarios, o deberán llevar en su vehículo a otros usuarios.

Se pide generar y optimizar mediante algoritmos de búsqueda local un plan de rutas para los vehículos disponibles en el servicio. Estas rutas deberán satisfacer una serie de restricciones.

Por último se realizan una serie de experimentos modificando y comparando aspectos de la implementación del problema y los algoritmos utilizados.

# Índice

# 1. Descripción del problema

Se plantea un sistema de compartición de vehículos donde cada usuario pone a disposición plazas de su vehículo para recoger a otros usuarios en su trayecto al trabajo. Cada mes, de manera rotativa, se definen $M$ conductores de los $N$ usuarios apuntados al servicio (naturalmente $M < N$).

El servicio se pone en marcha en una ciudad representada (por simplicidad) por una cuadrícula de 10×10 kilómetros formada por manzanas de 100×100 metros. Cada usuario tiene una dirección de origen (casa) y destino (trabajo). Las direcciones se corresponden con cruces de la cuadrícula.

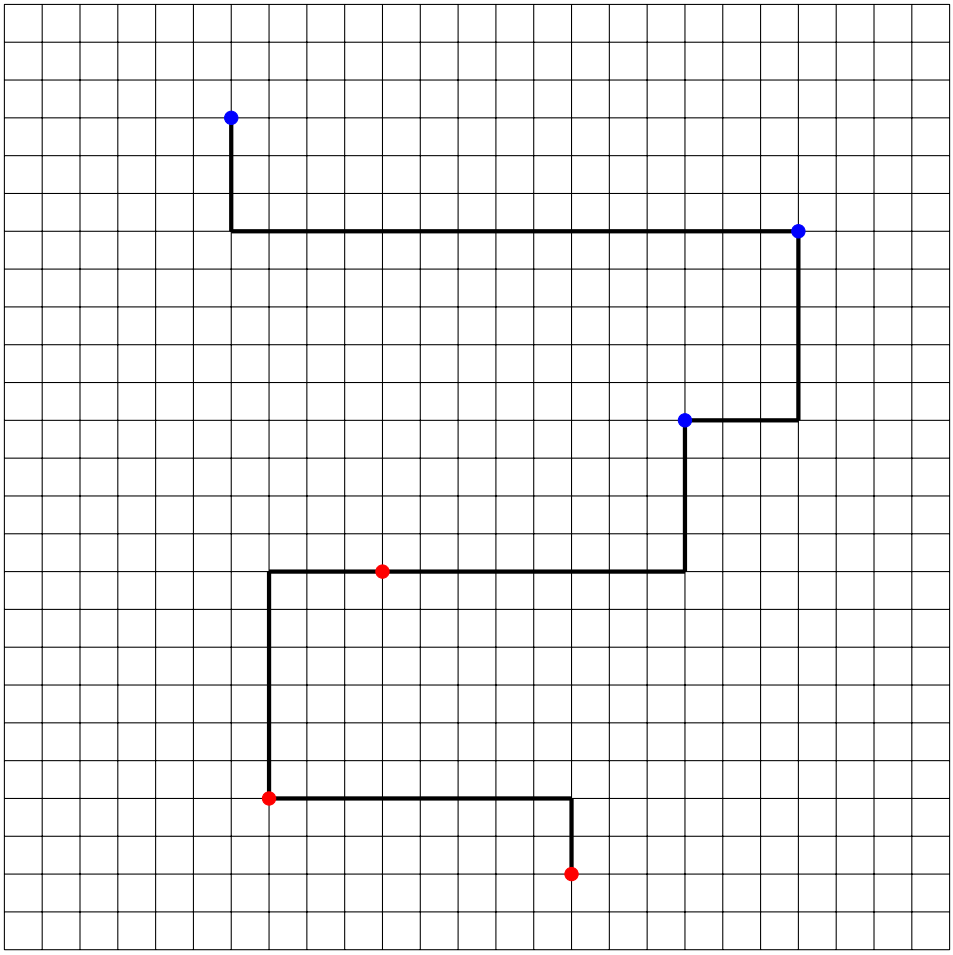

Figura 1: Representación simplificada de la ciudad con los puntos de origen y destino de 3 usuarios. El conductor los recoge y deja en sus trabajos.

La distancia entre 2 puntos $A$ y $B$ de la ciudad se calcula como la distancia Manhattan (o distancia del taxista):

$$d(A, B) = |a_x - b_x| + |a_y - b_y|$$

Nótese que el trayecto entre $A$ y $B$ puede realizarse de múltiples formas, no es relevante el camino exacto.

La solución a este problema que se pide es, para un mes concreto, generar un plan que indique a cada conductor la ruta que debe seguir cada día. Cada ruta incluye los puntos de recogida y destino de cada usuario que lleva cada conductor. El plan generado debe cumplir las siguientes condiciones:

- El conductor del vehículo debe llegar a tiempo a su trabajo. El vehículo inicia su recorrido a las 7 de la mañana y debe llegar al trabajo a las 8 de la mañana. Suponemos que la velocidad media del tráfico es de 30 km/h, por lo tanto **cada vehículo recorrerá, como mucho, 30 km**.
- Todos los usuarios apuntados al servicio serán recogidos y dejados en su trabajo.
- En un vehículo no puede haber más de 3 personas al mismo tiempo (1 conductor y 2 pasajeros).
- Sólo pueden conducir los $M$ conductores designados.

El objetivo es generar un plan eficiente, por ello se usarán 2 criterios para evaluar la calidad de la solución:

1. Minimizar la distancia recorrida total recorrida por los vehículos.
2. Minimizar la distancia recorrida total recorrida por los vehículos y el número de vehículos utilizados.

Nótese que, debido a los criterios de calidad, pueden darse soluciones en las que...

- Se utilicen menos vehículos, y, por lo tanto no todos los $M$ conductores conduzcan.
- La ruta de un vehículo incluya los puntos de origen y destino de más de 2 pasajeros. El vehículo puede dejar a un pasajero durante la ruta y ocupar nuevamente la plaza.

## 1.1. ¿Por qué usar algoritmos de búsqueda local?

La complejidad de este problema es alta (asumimos NP-Hard). Se trata de un problema de optimización combinatoria y, en particular, un problema de optimización de rutas (conocidos por ser especialmente difíciles). La razón principal de esta complejidad es que el número de posibles soluciones aumenta exponencialmente con el número de usuarios y conductores. En concreto, el número de posibles soluciones es del orden de $O(N^M)$, donde $N$ es el número de usuarios y $M$ es el número de conductores. Este número es exponencial en $N$ y $M$, lo que hace que la búsqueda exhaustiva de la solución óptima sea impracticable para instancias de tamaño moderado.

Además, el problema involucra múltiples restricciones que deben ser satisfechas, como la limitación del tiempo de llegada al trabajo, la limitación del número de pasajeros en un vehículo y la necesidad de que todos los usuarios sean recogidos y dejados en sus destinos. Estas restricciones hacen que el problema sea más complejo y requiera técnicas avanzadas de optimización y búsqueda.

Los algoritmos de búsqueda local son particularmente útiles cuando se trata de problemas como este, en los que no se puede evaluar rápidamente todas las posibles soluciones, ya que pueden encontrar soluciones cercanas a la óptima sin necesidad de evaluar todas las soluciones posibles. Estos algoritmos son altamente escalables y se pueden aplicar fácilmente a problemas grandes y complejos.

# 2. Implementación del problema

Para implementar y poder resolver este problema se usarán las clases Java del libro AIMA *(Artificial Intelligence: A Modern Approach)*. Estas clases son un conjunto de estructuras y algoritmos que representan los conceptos clave de la inteligencia artificial y se utilizan para implementar soluciones a problemas específicos. En la implementación de las clases Java del AIMA hay varias clases disponibles, nosotros usaremos las enfocadas a búsqueda, en particular, *Hill Climbing* y *Simulated Annealing*.

En el enunciado de esta práctica se indica que nos proporcionan de entrada 2 clases en la librería `IA.Comparticion`:

- `Usuario`, que tiene la estructura que representa un usuario, es decir, sus coordenadas de origen y destino, y si es conductor este mes o no. Las coordenadas de un usuario usan como unidades el centenar de metros.
- `Usuarios`, que genera una estructura con todos usuarios que están apuntados al servicio. El constructor recibe tres parámetros, el número de usuarios a generar, el número de posibles conductores del mes y la semilla para el generador de números aleatorios (direcciones).

## 2.1. Representación del estado

Lo primero que se hará es implementar una forma de representar un estado del problema. Debido a que utilizaremos algoritmos de búsqueda local, en este caso un estado representa una solución al problema, ya que la búsqueda se realiza en el espacio de soluciones.

Para representar un estado, la información mínima y necesaria es saber qué coches son utilizados y qué ruta sigue cada uno, es decir, a qué usuarios recoge y deja, y en qué orden. Otra información como la longitud de la ruta puede ser calculada bajo demanda, por lo que no se almacena en memoria, i.e. no forma parte del estado.

A continuación se comentan los aspectos más importantes de las clases que hemos se han implementado.

### 2.1.1. La clase `Car`

Se implementa primero la clase `Car`, que contiene un

```
ArrayList<Usuario> m_PassengersRoute
```

Este *ArrayList* contiene la ruta que sigue el coche. Cada usuario del *ArrayList* se corresponde con una acción que realiza el coche, ya sea recoger (*Pick-up*) o dejar (*Drop-off*). Por lo tanto si $n$ usuarios utilizan el coche (incluido el conductor), tendremos $2n$ elementos en el *ArrayList*.

Por ejemplo, una ruta en la que el conductor 1 recoge a los usuarios 2, 3, los deja en sus trabajos, y termina en su trabajo, se ve así:

$$[P_1, P_2, P_3, D_2, D_3, D_1]$$

Figura 2: Representación de la ruta de un coche, donde $P$ (*Pick-up*) indica la acción de recoger, y $D$ indica la acción de dejar (*Drop-off*).

El coche consulta las coordenadas de origen cuando detecta un Pick-up, y las de destino cuando se trata de un Drop-off. Naturalmente el primero en subirse al vehículo, y el último en bajarse, es el conductor y dueño del mismo.

Otro método importante de esta clase es

```
CheckRouteIntegrity()
```

que verifica que en ningún momento de la ruta se sobrepasa la capacidad del vehículo.

#### 2.1.2. La clase `State`

Esta clase cuenta con un conjunto de usuarios

```
Usuarios m_Users
```

que contiene todos los usuarios apuntados al sistema, y un

```
ArrayList<Car> m_Cars
```

que contiene todos los $M$ coches disponibles para ser usados. Para saber los coches que realmente se usan, usamos el método

```
GetNonEmptyCars()
```

### 2.2. Operadores

Una vez definido definido el estado como el conjunto de todas las rutas de los vehículos que se utilizan. Deben pensarse los operadores adecuados para explorar espacio de estados y poder así llegar a nuevas soluciones, es decir, realizar modificaciones a las rutas para tratar de mejorarlas.

Para poder explorar todas las soluciones correctamente, los operadores deben permitir:

- Modificar una ruta internamente sin afectar a las demás (estas modificaciones no afectan al tamaño de la ruta).
- Aumentar o disminuir las rutas (añadiendo o quitando pasajeros, y sus respectivas acciones), llegando incluso a eliminarlas para disminuir el número de vehículos (en este caso las modificaciones sí que repercuten en las demás rutas, pues los pasajeros deben ser reubicados).

Con la premisa de que los operadores tengan un factor de ramificación aceptable, se deciden implementar tres operadores.

#### 2.2.1. *Swap* interno

Este operador intercambia 2 acciones dentro de una ruta. La condición de aplicabilidad es que no se quiebre la integridad de la ruta al realizarse:

- Las acciones asociadas al conductor no pueden ser intercambiadas.

- No se pueden intercambiar las 2 acciones de un mismo usuario.
- El intercambio no puede provocar que se supere la capacidad máxima del vehículo en un momento de la ruta.

El operador se implementa como un método asociado a la clase `Car` (por ello no es necesario indicar en los parámetros la ruta de qué coche estamos cambiando). Se indica para cada acción el tipo de acción que es (*Pick-up* o *Drop-off*) y el usuario sobre el que se realiza.

```
swap(accion1, usuario1, accion2, usuario2)
```

Por ejemplo, si se tiene la ruta

$$[P_1, P_2, P_3, D_2, D_3, D_1]$$

y realizamos un `swap(pickup, 3, dropoff, 2)`, la ruta resultante es

$$[P_1, P_2, D_2, P_3, D_3, D_1]$$

Para determinar el factor de ramificación de este operador, se define $c$ como el número de coches utilizados y $p$ como el número de pasajeros en cada coche. Nótese que, para cada coche, cada acción $(2p)$ se puede intercambiar con el resto $(2p-1)$, luego el factor de ramificación es del orden de $O(c((2p)(2p-1))) \Rightarrow O(c(2p)^2) \Rightarrow O(cp^2)$.

### 2.2.2. *Shift*

Este operador nuevamente modifica una ruta sin alterar las demás. Permite seleccionar una acción de la ruta y posponerla tanto como se desee. En este caso la única condición de aplicabilidad es que el desplazamiento no se haga sobre alguna de las acciones asociadas al conductor. Si se desplaza una acción más posiciones que número de acciones hay en la ruta no importa, ya que la implementación interna del operador se comporta como un carrusel.

También se implementa como un método asociado a la clase `Car`, por lo que únicamente se debe indicar el tipo de acción, usuario y número de posiciones que se desea desplazar en la ruta.

```
shift(accion, usuario, steps)
```

Por ejemplo, si se tiene la ruta

$$[P_1, P_2, D_2, P_3, D_3, D_1]$$

y se realiza un `shift(dropoff, 2, 2)`, la ruta resultante es

$$[P_1, P_2, P_3, D_3, D_2, D_1]$$

Nuevamente, para calcular el factor ramificación se ve que, para cada ruta, cada acción se puede desplazar a tantas posiciones como acciones tiene la ruta (menos ella misma, claro). Por lo tanto el factor de ramificación vuelve a ser $O(c((2p)(2p-1))) \Rightarrow O(c(2p)^2) \Rightarrow O(cp^2)$.

### 2.2.3. *Move*

Este operador permite mover un usuario de un vehículo a otro, siempre y cuando no se quebrante la integridad de la ruta del vehículo que abandona el usuario, ni la del vehículo al que es llevado. Por eso se debe tener en cuenta que:

- Un conductor no puede ser movido a otro vehículo si quedan pasajeros en el suyo.
- Si un conductor es movido a otro vehículo, este queda vacío y por lo tanto **ya no cuenta como vehículo usado**.

Otra consideración importante es que se decide implementar este operador de manera que, a la hora de añadir el usuario al vehículo nuevo, sus acciones de *Pick-up* y *Drop-off* se añaden con un factor de "**aleatoriedad**"(no es aleatorio del todo que ya debe respetarse la integridad de la ruta al añadir las acciones).

Para utilizar este operador debe indicarse el usuario que se desea mover, y a qué vehículo lo se desea llevar.

```
move(usuario, coche)
```

Por ejemplo, si se tienen las rutas

$$\text{Coche 1 } [P_1, P_2, D_2, P_3, D_3, D_1] \text{ Coche 2 } [P_4, P_5, P_6, D_5, D_6, D_4]$$

y realizamos un `move(2, 2)`, las rutas resultantes son

$$\text{Coche 1 } [P_1, P_3, D_3, D_1] \text{ Coche 2 } [P_4, P_5, P_6, D_5, P_2, D_6, D_2, D_4]$$

aunque el resultado de la ruta 2 podría ser distinto debido al factor de aleatoriedad.

En este caso el factor de ramificación puede determinarse al verse que, si se define el conjunto de usuarios como $n$ y el conjunto de coches utilizados como $c$, cada usuario puede ser movido a todos los demás coches menos el suyo. Por lo tanto, el factor de ramificación es del orden de $O(n(c-1)) \Rightarrow O(nc)$

## 2.3. Funciones heurísticas

Tras definir los operadores, se puede explorar el espacio de soluciones. Pero para poder hacerlo correctamente se deben definir unas heurísticas adecuadas y capaces de guiar a los algoritmos de *Hill Climbing* y *Simulated Annealing* para lograr la mejor solución posible.

Para este problema, las heurísticas a implementar están directamente relacionadas con los criterios de calidad de las soluciones. Por lo tanto se implementan 2 funciones heurísticas, una por cada criterio de calidad.

### 2.3.1. Distancia total recorrida

Esta primera función heurística busca satisfacer el primer criterio de calidad de la solución: *Minimizar la distancia recorrida por los conductores.* En este caso, la calidad de la solución depende de un único factor, que es la distancia total recorrida por los vehículos. Por lo tanto, es

obvio que esta heurística es simplemente el valor total de la distancia recorrida. Específicamente, recoge la suma total de las distancias recorridas por cada vehículo. Es capaz de guiar a los algoritmos hacia soluciones mejores, y además es fácil de calcular y no requiere mucho tiempo de cómputo, lo que es una ventaja importante en un problema de búsqueda local donde se necesitan evaluar muchas soluciones en un corto periodo de tiempo.

Formalmente, la función heurística de la distancia total recorrida viene dada por el sumatorio

$$\sum_{i \in C} = d_i$$

- C es el conjunto de los vehículos que se conducen
- $d_i$ es la distancia recorrida por el coche $i$

### 2.3.2. Distancia total recorrida y número de vehículos utilizados

La segunda función heurística debe satisfacer el segundo criterio de calidad de la solución: *Minimizar la distancia recorrida por los conductores y minimizar el número de conductores*. En este caso, la heurística ya no depende de un único factor, sino que depende de dos: la distancia recorrida y el número de conductores.

Debido a que esta situación depende de varios factores, la mejor manera para representar la heurística es mediante una suma ponderada. En general, es más ventajoso eliminar un conductor que reducir un kilómetro de distancia, ya que la eliminación del conductor puede ahorrar más que reducir en 1 kilómetro la distancia total.

Para determinar entonces cuáles deben ser los factores de la suma ponderada, hay que plantearse antes la siguiente pregunta. Cuando se quita un vehículo de la solución, ¿de cuántos kilómetros se reduce la distancia total recorrida?

Para encontrar una respuesta a esta pregunta, hemos utilizado los resultados del experimento 1 (véase 3.1). De ese experimento, extrayendo los resultados numéricos de los 10 experimentos realizados con los mejores operadores (*Shift* + *Move*), la solución inicial *RansomSolutionState* y mediante el algoritmo de *Hill Climbing*, la media de kilómetros conducidos por cada conductor es de 15,5 km. Por lo tanto, utilizamos este resultados para definir la función heurística como la suma ponderada siguiente:

$$h(d, c) = 15,5c + d$$

- $c$ es el número de conductores
- $d$ es la distancia total recorrida

De esta forma, en cuanto a la calidad de la solución, se asegura que reducir 15,5 kilómetros tienen el mismo peso que reducir un conductor.

## 2.4. Generación de un estado inicial

Generar un estado inicial equivale en este caso a generar una solución a partir de la cual ejecutar los algoritmos de búsqueda. Construir esta solución inicial implica lograr un plan de rutas correctas, que cumplan las restricciones del problema, lo cual no es sencillo. Esto puede desincentivar la búsqueda de esta solución inicial y motivar alternativas como empezar la búsqueda desde el espacio de no soluciones.

Previamente a la implementación de las formas de generar una solución inicial que se muestran a continuación, se probó a ejecutar los algoritmos a partir de un estado inicial que no era solución. La generación de este estado inicial se implementó mediante dos estrategias distintas:

- Asignar los usuarios (no conductores) de forma secuencial a todos los vehículos disponibles, de manera equitativa. Los algoritmos lograban mejorar los criterios de calidad, pero el estado alcanzado no cumplía con las restricciones del problema i. e. no era solución. Ante esto se probó a penalizar fuertemente la heurística para aquellos estados que no eran solución, pero el resultado fue muy pobre. En este caso, esporádicamente se alcanzaba un estado solución pero, si se lograba, los criterios de calidad no eran aceptables.
- Asignar los usuarios (no conductores) de forma aleatoria a todos los vehículos disponibles. De esta manera ocurría algo similar a lo que sucedía con la estrategia secuencial.

Por lo tanto empezar desde el espacio de no soluciones no es una buena opción. Los operadores definidos deben aplicarse muchas veces para cambiar sustancialmente el estado, por lo que abandonar el espacio de no soluciones es muy difícil. Lo que terminaba ocurriendo en ambos casos era que los algoritmos se limitaban a optimizar la penalización.

Descartadas ambas estrategias, se proponen a continuación 2 nuevas estrategias algo más elaboradas con el objetivo de poder generar una solución inicial.

### 2.4.1. Generación de estado inicial, aproximación avariciosa

El algoritmo se basa en la selección aleatoria de vehículos y usuarios, y utiliza técnicas de optimización de rutas para minimizar la distancia total recorrida. Para ello, se crean dos listas de usuarios (que se corresponden con acciones) (*Pick-ups* y *Drop-offs*) que contienen las acciones asociadas a los usuarios que no son conductores. A continuación, se seleccionan aleatoriamente los vehículos disponibles y se les asignan acciones de estas listas, en función de su ubicación. Se da prioridad a las acciones más cercanas, para minimizar la distancia recorrida por los vehículos.

Si la distancia total recorrida por un vehículo supera la distancia máxima permitida, se eliminan los usuarios más recientes y sus acciones se vuelven a agregar a las listas. Este proceso se repite hasta que se ha asignado un vehículo a cada usuario y se ha calculado la distancia total recorrida.

Se ha visto que existe la probabilidad de que no se asignen todos los usuarios, pero por la naturaleza aleatoria del sistema de asignaciones, si repetimos el algoritmo varias veces, se llega a una solución rápidamente cuando $M \geq N/2$

### 2.4.2. Modificación de la aproximación avariciosa

El algoritmo es análogo al anterior, pero la selección de la acción más cercana es aleatoria. De la lista de acciones disponibles, se añade, de manera aleatoria, una distancia a cada acción y se escoge aquella que tenga la distancia mínima. Gracias a esto, se puede ejecutar el algoritmo anterior con más .ªleatoriedad".

# 3. Experimentos

En esta sección se plantean una serie de experimentos para observar el comportamiento de los algoritmos en diferentes escenarios ajustando los operadores, la heurística, la estrategia de generación de solución inicial, etc.

## 3.1. Experimento 1: Buscando el mejor conjunto de operadores

Para sacarle el máximo partido a los algoritmos de búsqueda es crucial usar el o los mejores operadores en la búsqueda. Durante las pruebas de ejecución en la fase de implementación de los operadores, se observó que hay algunos que pueden funcionar mejor que otros, y concretamente se observó a priori un buen rendimiento combinando los operadores de *Shift* + *Move*.

En este experimento se comparan los operadores para ver qué combinación da mejor resultado a la hora de optimizar el primer criterio de calidad, es decir, qué combinación devuelve la mejor heurística asociada a la distancia total recorrida. El número de usuarios ($N$) se fija en 200, y el número de vehículos disponibles ($M$) en 100. La estrategia para hallar la solución inicial utilizada es la especificada en la sección 2.4.1 y se utiliza el algoritmo de *Hill Climbing*. Para cada conjunto de operadores se ejecuta 10 veces el problema con una semilla aleatoria cada vez. Los conjuntos de operadores a comparar son los siguientes:

- *Shift* + *Move*
- *Swap* interno + *Move*
- *Swap* interno + *Shift* + *Move*
- *Move*

Cabe destacar que el operador de *Move* se utiliza siempre puesto que es el único que permite modificar el tamaño de las rutas, por lo tanto es imprescindible para explorar todo el espacio de soluciones. Además incluso por si solo puede hacerlo.

Las características del experimento se resumen en la tabla de la figura 3.

En la figura 4 se muestran los resultados obtenidos. Se observa que el conjunto de operadores con los que más distancia se recorre es el conjunto de *Swap* interno + *Move*, teniendo el rango intercuartílico más alto. Cuando se realiza la búsqueda usando los tres operadores se encuentran soluciones mejores, pero aparentemente la combinación *Shift* + *Move* mejora a las demás. El operador *Move* llega incluso a lograr la mejor solución en esta muestra tomada, pero de manera aislada.

Para reforzar la hipótesis de que la combinación *Shift* + *Move* es significativamente mejor, se realiza una prueba de hipótesis utilizando las hipótesis nula (H0) y alternativa (HA) que se muestran en la figura 3.

Se selecciona un nivel de significancia (alfa) para la prueba de 0.05, lo que significa que la probabilidad de cometer un error tipo I (rechazar la hipótesis nula cuando es verdadera) es del 5 %. Como prueba estadística se utiliza una t de Student para muestras independientes. Se calculan los valores $t$ (mide la diferencia entre las medias de las muestras) y $p$ (probabilidad de obtener un valor de prueba al menos tan extremo como el observado si la hipótesis nula fuera verdadera). Si el valor $p$ es menor que el nivel de significancia seleccionado, se rechaza la hipótesis nula y se concluye que hay evidencia suficiente para apoyar la hipótesis alternativa. Al realizar la prueba, se obtiene el siguiente resultado:

| | |
|---|---|
| Observación | Los operadores *Shift + Move* pueden obtener mejores soluciones. |
| Planteamiento | Se escogen diferentes conjuntos de operadores y realizamos varias ejecuciones con cada uno. |
| Hipótesis | ▪ Hipótesis nula (H0): La media de la distancia recorrida por los vehículos utilizando el conjunto de operadores *Shift + Move* es igual o peor que la media de la distancia recorrida por los vehículos utilizando los otros conjuntos de operadores.<br>▪ Hipótesis alternativa (HA): La media de la distancia recorrida por los vehículos utilizando el conjunto de operadores *Shift + Move* es significativamente mejor que la media de la distancia recorrida por los vehículos utilizando los otros conjuntos de operadores. |
| Método | Para cada conjunto de operadores se realizan 10 ejecuciones, cada una con una semilla aleatoria, y se mide la distancia total recorrida. |

Figura 3: Tabla que muestra las características del experimento 1.

```
t = -2.5947, df = 38, p-value = 0.00669
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
      -Inf -9.164107
sample estimates:
mean of x mean of y
 868.7000  894.8667
```

El valor de $p$ es 0.00669, lo que significa que hay suficiente evidencia estadística para rechazar la hipótesis nula a un nivel de significancia del 5 %. El valor de $t$ es -2.5947, lo que indica que la media de distancia recorrida para el conjunto de operadores *Shift + Move* es significativamente menor que la media de los otros tres conjuntos de operadores. Esto, dicho de otro modo, se puede afirmar con un nivel de confianza del 95 %.

Aunque no forme parte del experimento, se incluye también en la figura 5 los tiempos de ejecución para cada conjunto de operadores. Se aprecia como el conjunto de operadores *Shift + Move* no marca los mejores tiempos, y al mismo tiempo resaltan los tiempos del operador de *Move*. Se establece un *tradeoff* entre calidad de la solución y coste temporal, pero se prioriza la calidad de la solución.

**Conclusión:** se asume el coste temporal añadido, y se definen los operadores *Shift + Move* para ser usados en los siguientes experimentos.

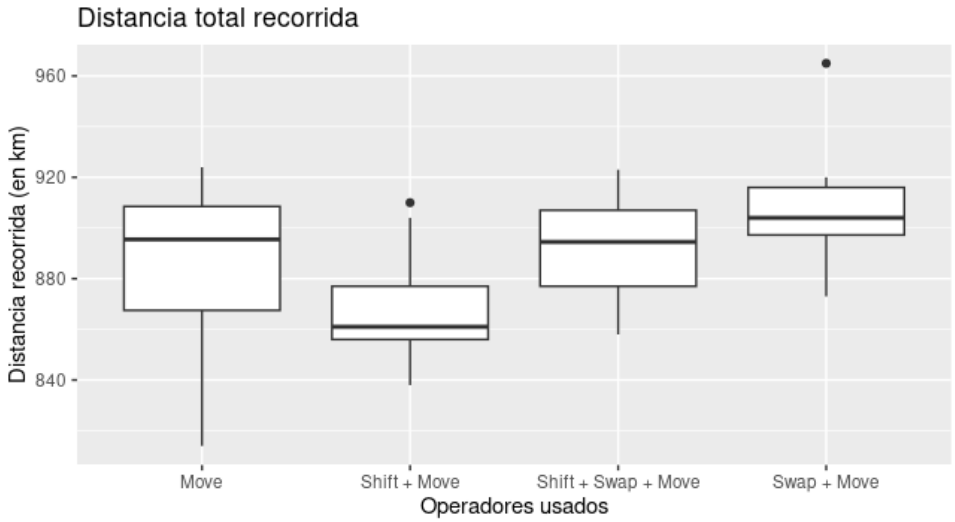


Figura 4: Distancia total recorrida en función del conjunto de operadores utilizado.


Figura 5: Tiempo de ejecución en función del conjunto de operadores utilizado.

## 3.2. Experimento 2: La mejor estrategia para generar la solución inicial

Cuando se usan algoritmos de búsqueda local para tratar de optimizar una solución, es importante partir de una solución adecuada. Partir de una solución no demasiado buena puede hacer que el algoritmo quede atrapado en un óptimo local. Esta solución inicial debe ser lo suficientemente buena como para estar cerca del óptimo global, pero al mismo tiempo debe tener cierto margen de mejora (por ejemplo, que no se encuentre en una meseta).

Para este experimento se utiliza el mismo escenario que en el anterior, pero en este caso se toma una muestra de 10 ejecuciones para cada estrategia de generación de la solución inicial (especificadas en la sección 2.4). Las características del experimento se resumen en la siguiente tabla:

| | |
|---|---|
| Observación | Una de las 2 estrategias de generación de la solución inicial conduce a un resultado mejor que la otra. |
| Planteamiento | Se toma una muestra de resultados ejecutando la búsqueda local sobre soluciones iniciales generadas por las estrategias a comparar. |
| Hipótesis | ▪ Hipótesis nula (H0): No hay una diferencia significativa entre las medias de la distancia total recorrida en las muestras para cada estrategia de generación de la solución inicial.<br>▪ Hipótesis alternativa (HA): Hay una diferencia significativa entre las medias de la distancia total recorrida en las muestras para cada estrategia de generación de la solución inicial. |
| Método | Para cada estrategia de generación de la solución inicial se realizan 10 ejecuciones, cada una con una semilla aleatoria, y se mide la distancia total recorrida. |

En la figura 6 se muestran los resultados obtenidos.

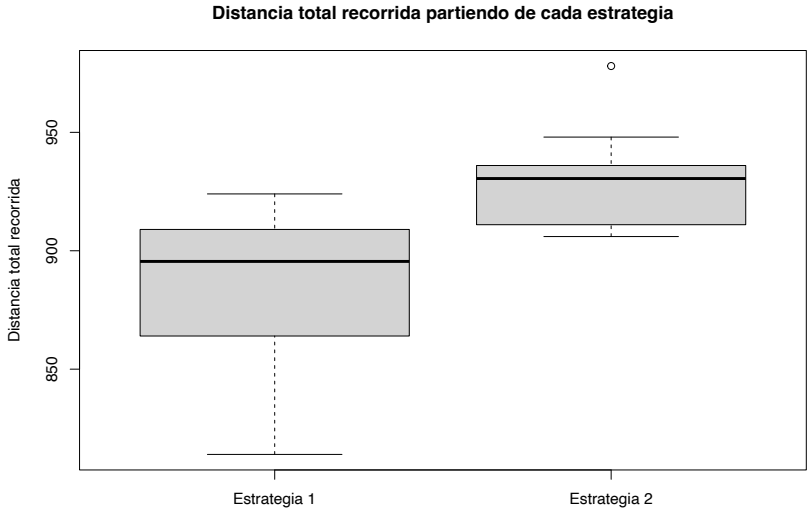


Figura 6: Distancia total recorrida en función de la estrategia de generación de la solución inicial

Se observa como la primera estrategia presenta una mayor varianza, pero los resultados en conjunto son mejores, lo que sugiere que el factor de aleatoriedad de la segunda estrategia no ayuda a lograr una solución de partida mejor.

Para reforzar la hipótesis de que la combinación *Shift* + *Move* es significativamente mejor, se realiza una prueba de hipótesis utilizando las hipótesis nula (H0) y alternativa (HA) que se muestran en la tabla.

Se utiliza la prueba t de student y se determina un valor de significancia $t$ de 0.05. Se compara si la diferencia media entre las dos estrategias es significativamente diferente de cero. La hipótesis nula es que no hay diferencia significativa entre las dos estrategias, mientras que la hipótesis alternativa es que hay una diferencia significativa. Si el valor p es menor que el nivel de significancia elegido, se puede rechazar la hipótesis nula y concluir que hay una diferencia significativa entre las dos estrategias.

```
t = -3.1987, df = 9, p-value = 0.01085
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -79.21506 -13.58494
sample estimates:
mean of x
    -46.4
```

**Conclusión:** En efecto, con un nivel de confianza del 95 % puede decirse que la diferencia entre estrategias es significativa. Nuevamente se proporcionan también los tiempos de ejecución, para ver cómo influye el uso de cada estrategia. Esto, como se observa en la figura 7, justifica más si cabe el uso de la primera estrategia.

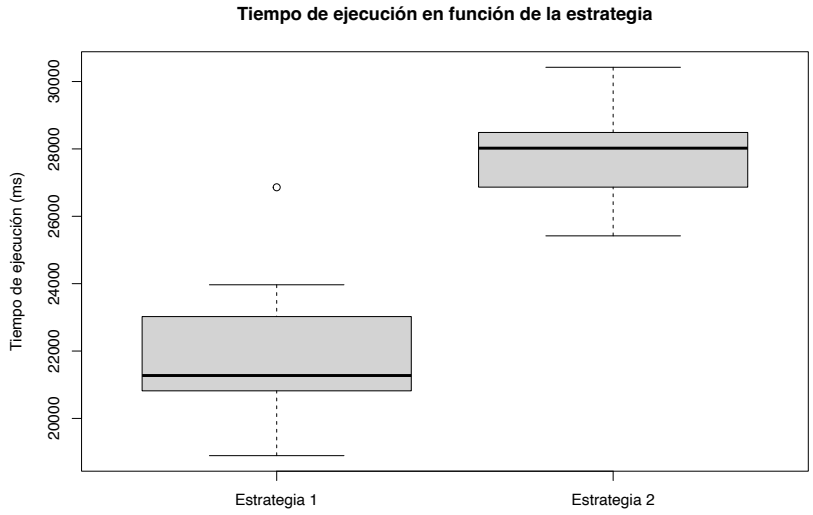


Figura 7: Tiempo de ejecución en función de la estrategia de generación de la solución inicial

### 3.3. Experimento 3: Fijando los parámetros para *Simulated Annealing*

A diferencia del algoritmo de *Hill Climbing*, donde en cada iteración se examinan todos los estados vecinos y se selecciona el mejor, en *Simulated Annealing* se genera un estado vecino al azar que, en caso de ser una solución peor, puede ser aceptado. Esto hace que el algoritmo pueda escapar de óptimos locales.

El *Simulated Annealing* se basa en el proceso real del recocido del acero y cerámicas, una técnica que consiste en calentar y luego enfriar lentamente el material para variar sus propiedades físicas, es decir, mejorarlo. Con este algoritmo se busca hacer lo mismo con la solución inicial. Se parte de una temperatura y se va disminuyendo En función de la temperatura, los estados subóptimos pueden aceptarse en mayor o menor medida esta función de aceptación viene dada por la siguiente expresión:

$$\mathcal{F}(T) = ke^{-\lambda T}$$

Para utilizar el algoritmo de *Simulated Annealing* se deben ajustar una serie de parámetros:

- Número total de iteraciones.
- Iteraciones por cada cambio de temperatura (ha de ser un divisor del anterior).
- Parámetro $k$ de la función de aceptación de estados subóptimos.
- Parámetro *lambda* de la función de aceptación de estados subóptimos.

El número total de iteraciones que ha de hacer el algoritmo antes de parar se ha de determinar experimentalmente y, aparte de depender del problema, depende de los valores $k$ y $\lambda$.

El número de pasos por cada cambio de temperatura divide las iteraciones en conjuntos de pasos en los que las condiciones de aceptación son iguales.

Para determinar cómo ajustar los parámetros $k$ y $\lambda$ debe tenerse en cuenta la función que determina la probabilidad de aceptación de un estado subóptimo:

$$P(estado) = e^{\frac{\Delta E}{\mathcal{F}(T)}}$$

El diferencial de energía ($\Delta E$) equivale al diferencial entre la calidad de la solución actual y la calidad de la solución subóptima que se está comprobando aceptar.

Vistas las 2 expresiones, las conclusiones sobre los parámetros $k$ y $\lambda$ son:

- A medida que aumenta $k$, aumenta el número de iteraciones en las que la probabilidad de aceptar soluciones subóptimas es más alta.
- Cuanto mayor es $\lambda$, aumenta la velocidad a la que desciende la probabilidad de aceptar estados subóptimos, y por lo tanto disminuye el número de iteraciones para llegar a probabilidad 0.

Ahora, primero debe determinarse el número total de iteraciones para el algoritmo. Para ello, se fijan arbitrariamente el resto de parámetros. Las características de este primer experimento se definen en la siguiente tabla:

| | |
|---|---|
| Observación | Aumentar el número de iteraciones no siempre mejora la calidad de la solución. |
| Planteamiento | Se aumenta el número de iteraciones hasta que deje de mejorar la calidad de la solución. |
| Hipótesis | A partir de un número de iteraciones, la calidad de la solución no mejora y sólo incrementa el coste temporal. |
| Método | Se determina el número inicial de iteraciones en 2500 y se incrementa cada vez en 2500 hasta 35000 iteraciones. Se fija el número de iteraciones por cada cambio de temperatura en 10, $k$ en 5 y $\lambda$ en 0.01. Se mide la calidad de la solución, es decir, la distancia total recorrida para cada número de iteraciones, y además se mide también el tiempo de ejecución. |

Los datos recogidos se muestran en la figuras figuras 8 y 9. Se observa que a partir de a partir de las 20000 iteraciones, la distancia total se establece entre 1000 y 1100 km. Los repuntes producidos pueden deberse a un ajuste inadecuado de los parámetros $k$ y $\lambda$. El tiempo de ejecución para ese número de iteraciones se considera admisible y por lo tanto se fija en 20000.

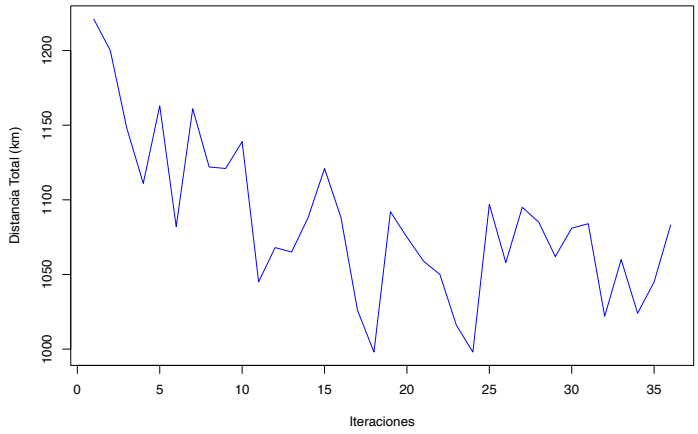


Figura 8: Distancia total recorrida en función del número de miles de iteraciones realizadas.

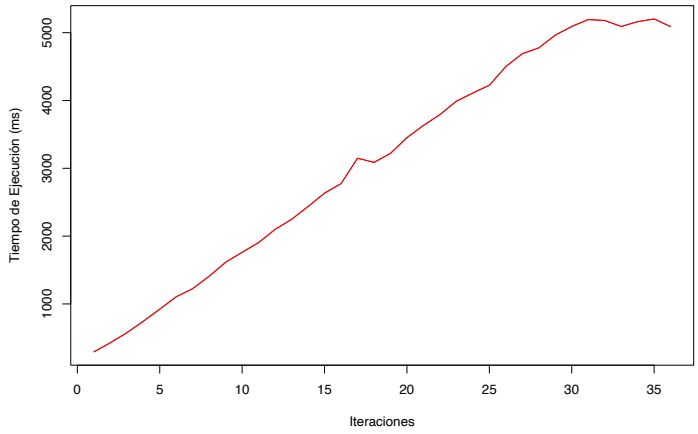


Figura 9: Tiempo de ejecución en función del número de miles de iteraciones realizadas.

Una vez fijado el número de iteraciones, se experimenta simultáneamente con los valores de $k = \{1, 10, 120\}$ y $\lambda = \{0.001, 0.001, 0.1\}$. Los resultados de esta experimentación se muestran en el gráfico de la figura 10:

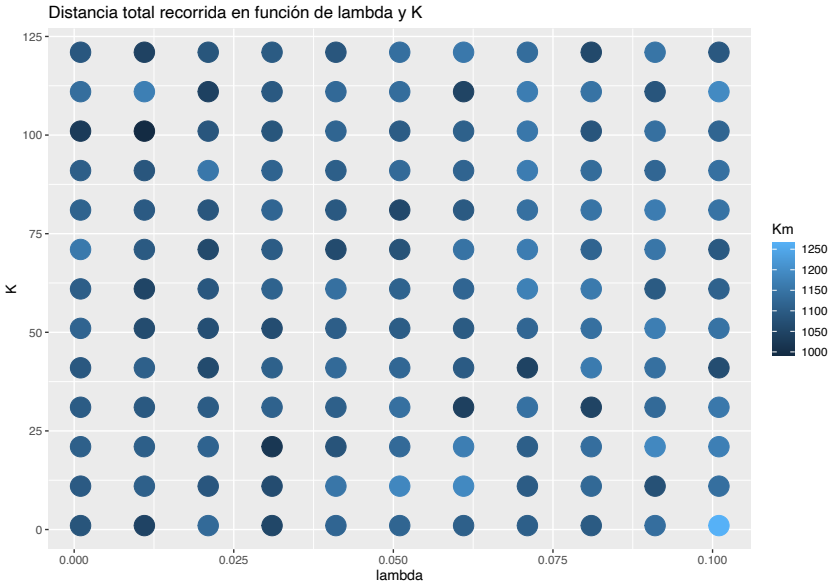


Figura 10: Distancia total recorrida en función de $\lambda$ y $k$ y .

Se observa ligeramente que los vehículos en conjunto recorren menos distancia para valores bajos de $\lambda$, lo que sugiere que el algoritmo funciona mejor cuando la probabilidad de aceptar soluciones subóptimas desciende lentamente. Por otro lado, el valor de $k$ no parece ser determinante en cuanto a la distancia total recorrida.

Como añadido, se deciden tomar datos sobre el número de vehículos utilizados para cada valor de $k$ y $\lambda$ con el fin de localizar el mejor rango de valores para $k$. En la figura 11 se muestran los resultados:

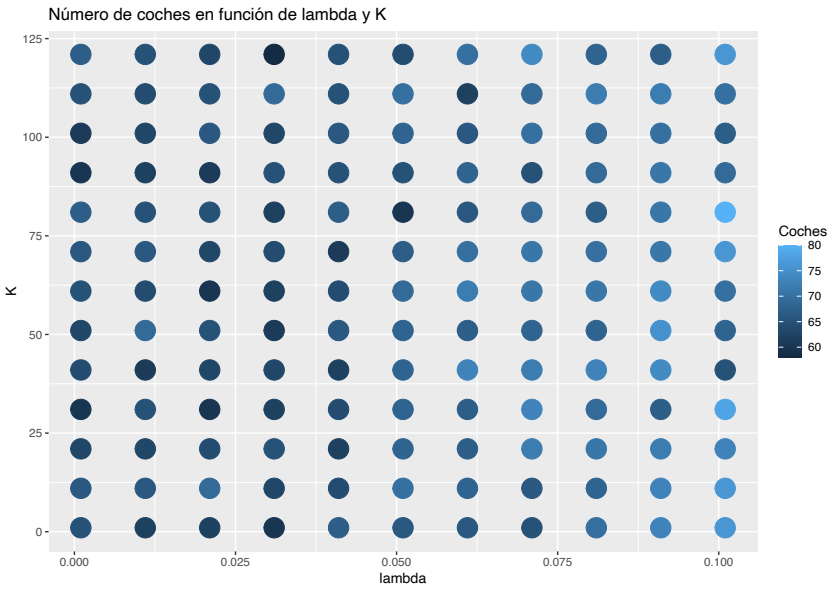


Figura 11: Número de vehículos utilizados en función de $\lambda$ y $k$ y .

En este caso se refuerza la idea de que los valores bajos para $\lambda$ funcionan bien. Se observa

muy tímidamente que las soluciones para valores de $k$ en torno a 100 obtienen un menor número de vehículos utilizados.

Finalmente se mide el tiempo de ejecución en función de $k$ y $\lambda$. En este caso el resultado es más predecible ya que cuantos más estados subóptimos se acepten, más trabaja el algoritmo. Esto se observa en la figura 12:

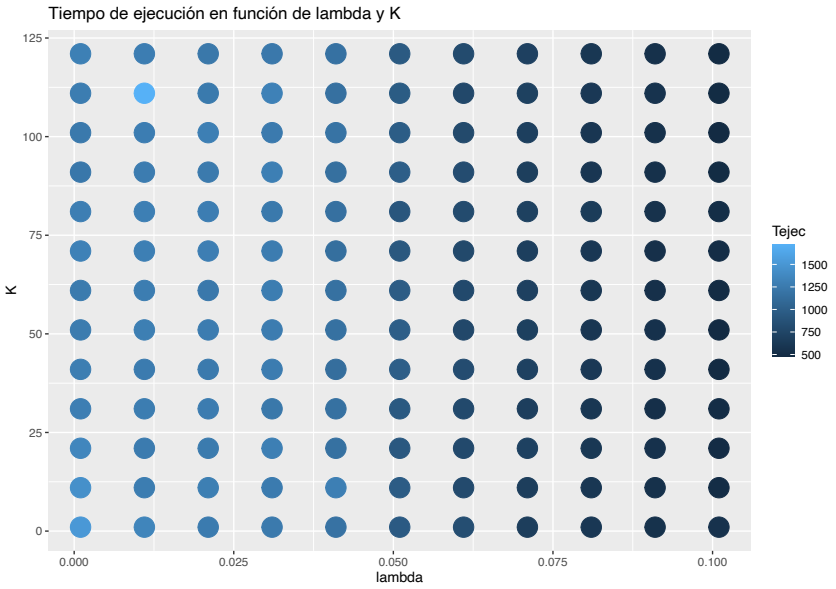


Figura 12: Tiempo de ejecución en función de $\lambda$ y $k$ y .

En definitiva se hace más evidente que $\lambda$ es más determinante, lo cual se puede observar en la propia función de aceptación de estados.

**Conclusión:** Pese a no encontrar valores claramente mejores para todos los parámetros, se deciden fijar los parámetros del *Simulated Annealing* de la siguiente forma:

- Número total de iteraciones = 20000
- Iteraciones por cada cambio de temperatura = 10 (fijado arbitrariamente)
- $k = 100$
- $\lambda = 0.01$

## 3.4. Experimento 4: Incrementando el tamaño del problema

En este experimento se pide incrementar el tamaño del problema incrementando el número de usuarios a la vez que se mantiene la proporción de conductores a $N/2$, para ver cómo crece el tiempo de ejecución. Por ejemplo: si tenemos seiscientos usuarios, tendremos trescientos conductores.

Al ser este un problema de optimización combinatoria y, presumiblemente pertenecer a la clase de complejidad NP-Hard, esto sugiere que el coste temporal crezca exponencialmente respecto al tamaño del problema.

El experimento se ha realizado para el algoritmo de *Hill Climbing*, con semilla aleatoria, utilizando los operadores de *Shift + Move* y la primera estrategia para generar la solución inicial. Las características del experimento se indican en la siguiente tabla:

| | |
|---|---|
| Observación | Los tiempos de ejecución escalan rápidamente en función del tamaño del problema. |
| Planteamiento | Se escogen diversos tamaños del problema y se mide el tiempo de ejecución. |
| Hipótesis | El coste temporal del problema es exponencial. |
| Método | Para cada tamaño escogido se mide el tiempo medio para 5 ejecuciones distintas con 5 semillas aleatorias. |

Durante las pruebas de implementación ya se observaron dificultades a la hora de computar soluciones para instancias grandes del problema, por lo que se espera que las soluciones sean impracticables en un tiempo admisible a partir de los 400-500 usuarios. Tras probarlo intencionadamente, los datos obtenidos muestran lo siguiente:

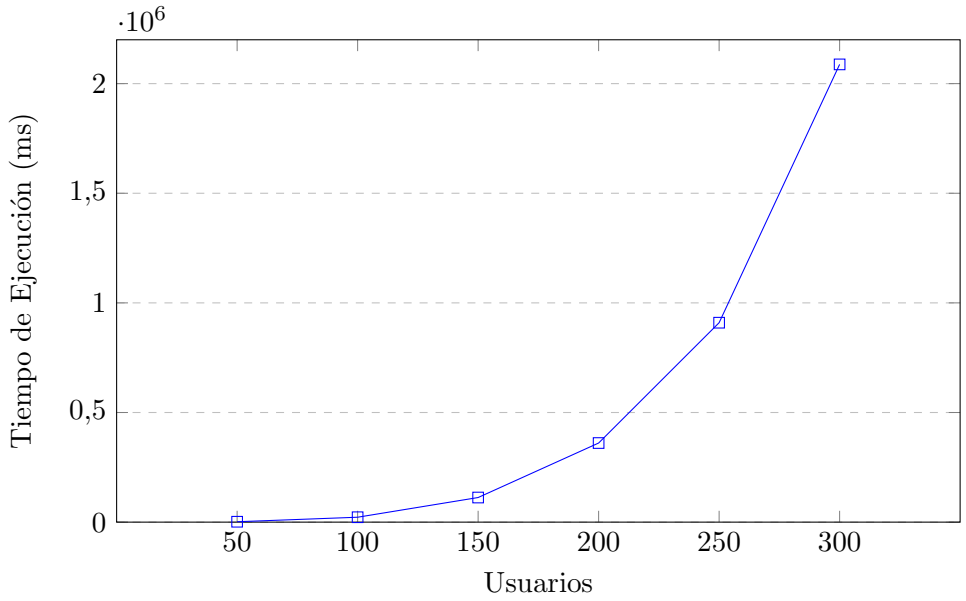


Figura 13: . Gráfico de tiempo de ejecución en función del tamaño del problema.

En el gráfico se observa un crecimiento que podría ser exponencial. Para reforzar esta hipótesis, se plantea una prueba de hipótesis, donde la hipótesis nula (H0) es que no hay ningún tipo de relación entre tiempo de ejecución y tamaño del problema, y la hipótesis alternativa (HA) es que sí que existe dicha relación, y además el tiempo crece de forma exponencial.

Para buscar esta relación exponencial se aplica un modelo de regresión lineal sobre los datos obtenidos utilizando el logaritmo natural del tiempo como variable dependiente, y el resultado es el siguiente:

```
Residuals:
       1        2        3        4        5        6
-0.81004  0.35418  0.60723  0.41547 -0.01912 -0.54771

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 6.942839   0.593057  11.707 0.000304 ***
Tamano      0.027188   0.003046   8.927 0.000871 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.637 on 4 degrees of freedom
Multiple R-squared:  0.9522,Adjusted R-squared:  0.9403
F-statistic: 79.69 on 1 and 4 DF,  p-value: 0.0008707
```

$p$ es 0.0008707, lo que significa que es muy poco probable que la relación observada entre el tamaño del problema y el logaritmo natural del tiempo de ejecución se deba al azar. Por lo tanto, se puede refutar la hipótesis nula de que no hay relación entre el tamaño del problema y el tiempo de ejecución. Además, el valor del coeficiente de determinación ajustado (*Adjusted R-squared*) es de 0.9403, lo que indica que el modelo explica el 94.03 % de la variación en el logaritmo natural del tiempo de ejecución.

**Conclusión:** Puede concluirse que existe una relación significativa entre el tamaño del problema y el tiempo de ejecución, y que el modelo de regresión lineal sugiere que el tiempo de ejecución crece exponencialmente en función del tamaño del problema.

## 3.5. Experimento 5: Comparando las heurísticas (*Hill Climbing*)

El objetivo de este experimento es determinar qué heurística funciona mejor para la distancia total recorrida y para el tiempo de ejecución. Hasta ahora se ha experimentado únicamente con un función heurística, pero en este experimento se va a usar también una segunda heurística (véase 2.3.2) que mida el segundo criterio de calidad de la solución: *Minimizar la distancia total recorrida y el número de vehículos utilizados.*

El escenario que se va a usar es el siguiente:

- Operadores: *Shift + Move*
- Solución inicial: *RandomSolutionState*
- Algoritmo: *Hill Climbing*
- 200 usuarios y 100 coches

**Nota** En adelante, la Heurística 1 (véase 2.3.1) se referirá a la heurística del primer criterio de calidad de la solución: *Minimizar la distancia total recorrida*; y la Heurística 2 (véase 2.3.2) se referirá a la heurística del segundo criterio de calidad de la solución: *Minimizar la distancia total recorrida y minimizar el número de conductores.*

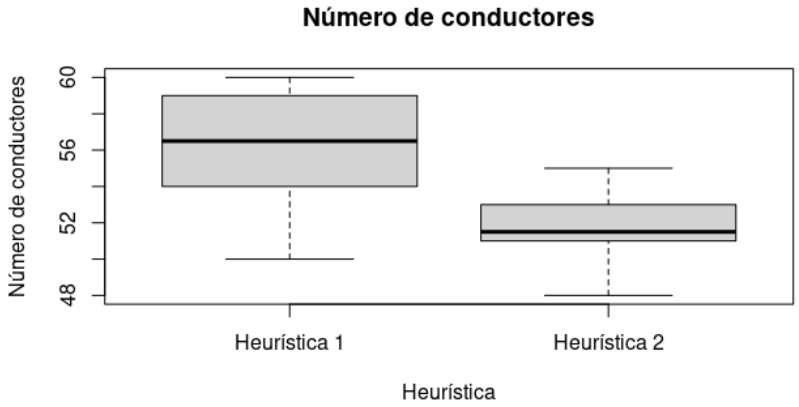


Figura 14: Número de conductores en función de la heurística utilizada

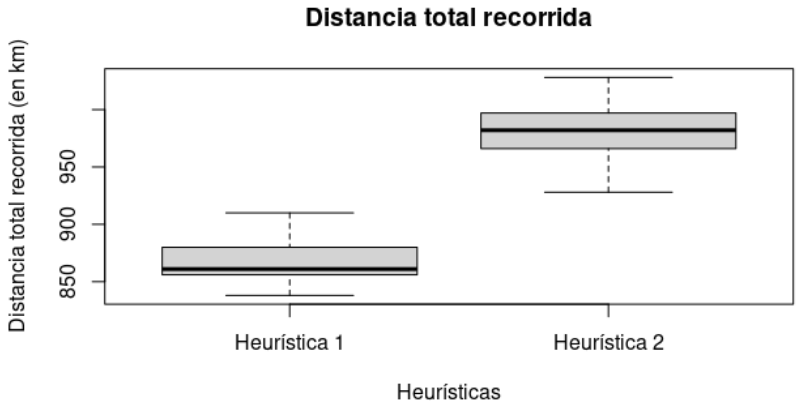


Figura 15: Distancia total recorrida en función de la heurística utilizada

Como se puede observar en la figura 15, existe una diferencia entre ambas heurísticas en lo que se refiere a la distancia total recorrida. Mientras que la mediana de la distancia recorrida usando la heurística 1 es cercana a 850km, con la segunda heurística es de aproximadamente 975km. Por lo que la distancia total recorrida es aproximadamente 125km menor con la heurística 1 en comparación con la segunda heurística.

De la figura 16 podemos abstraer que la diferencia entre las medianas de los tiempos de ejecución es notoria. Mientras que con la heurística 1 la mediana del tiempo de ejecución es cercana a los 27500ms, con la heurística 2 la mediana es de unos 15000ms.

La segunda heurística toma como un factor extra el número de conductores, por lo que trata de reducir el número de vehículos, y esto se observa en la figura 14. La mediana del número de coches es inferior en la heurística 2, no obstante esta disminución es bastante leve, pues la reducción es de 56 vehículos con la heurística 1 a 51 aproximadamente con la heurística 2.

**Conclusiones** Si bien la primera heurística lleva a soluciones con una distancia menor comparado con la segunda heurística, esta diferencia no es abismal. En cambio, a nivel de tiempo de ejecución, la segunda heurística muestra un importante mejora con respecto a la primera (prácticamente la mitad del tiempo). Por ende, si el tiempo de ejecución no es una prioridad,

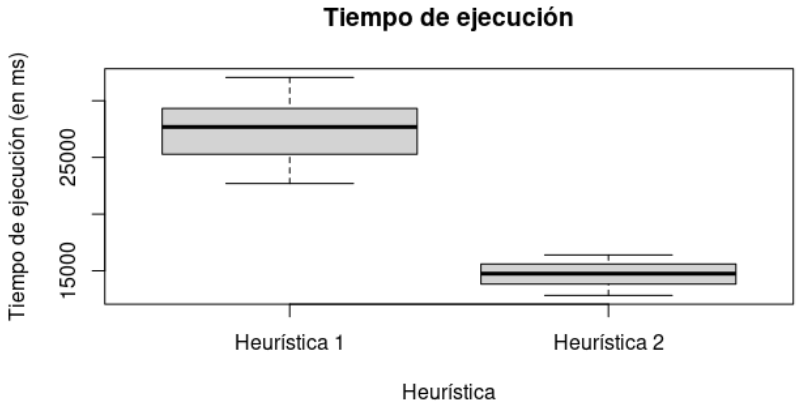


Figura 16: Tiempo de ejecución en función de la heurística utilizada

la primera heurística dará mejores resultados de distancia total recorrida.

## 3.6. Experimento 6: Comparando las heurísticas (*Simulated Annealing*)

Este experimento se plantea sobre el mismo escenario que el apartado anterior, pero en este caso usando el algoritmo de *Simulated Annealing*. Se busca comparar la distancia total recorrida, el número de vehículos usados, y el tiempo de ejecución en función de la heurística utilizada.

Se espera que los resultados se adecuen a los criterios de calidad propuestos. La heurística 1 debe conducir al algoritmo a priorizar el hecho de minimizar la distancia total recorrida por los conductores, mientras que la heurística 2 debe guiar la búsqueda hacia soluciones donde el número de vehículos se reduzca tanto como sea posible.

Los datos obtenidos se muestran en la figuras :

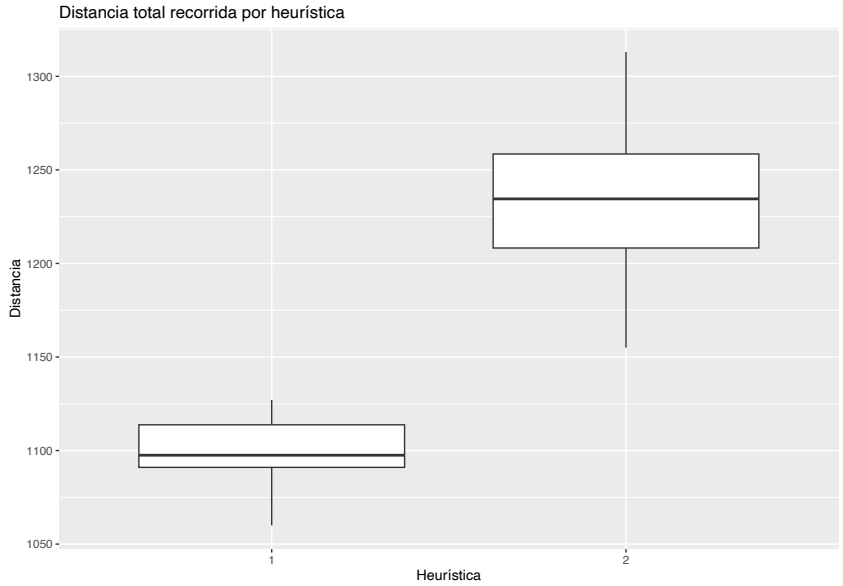


Figura 17: Distancia total recorrida (km) en función de la heurística utilizada

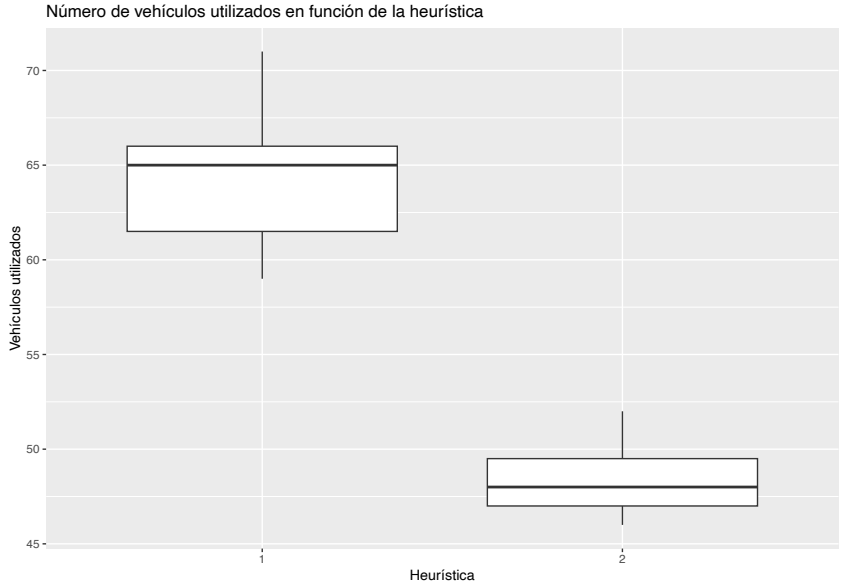


Figura 18: Número de vehículos utilizados en función de la heurística utilizada

En efecto, se observa claramente como cada heurística se centra en optimizar un criterio de

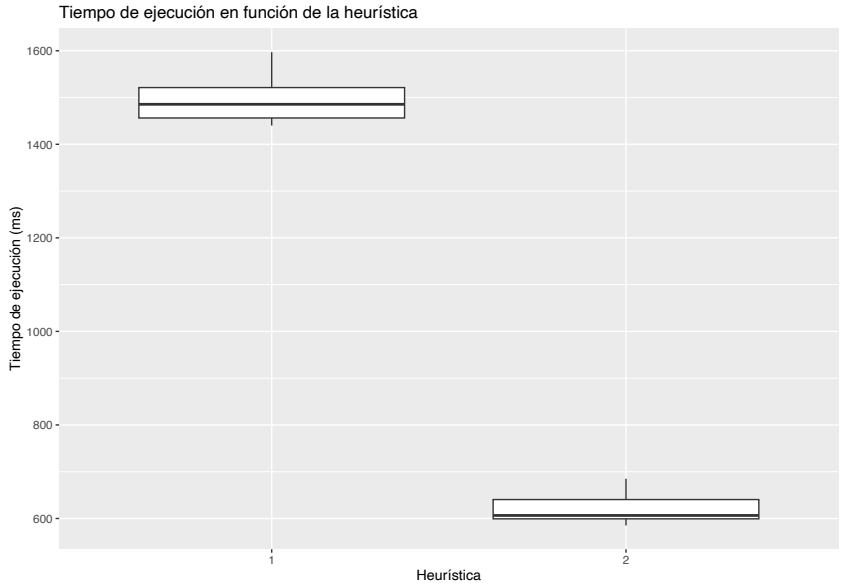


Figura 19: Tiempo de ejecución en función de la heurística utilizada

calidad.

La segunda heurística, para reducir el número de vehículos, fuerza al algoritmo a aumentar la distancia total recorrida respecto a la primera heurística. Esto se debe a que los vehículos deberán adoptar rutas en las que participen más usuarios.

Lo que es interesante observar es, que la primera heurística, pese a estar enfocada simplemente a reducir la distancia total, también conlleva una disminución significativa del número de vehículos. Esto se produce indirectamente por el hecho de que esta heurística provoca que no se conduzcan más kilómetros de los necesarios. Por ejemplo, si un conductor viaja sólo, y puede ser incluido en la ruta de otro vehículo, éste será reubicado. En otras palabras, la heurística implica que los coches se llenen.

Por último, cabe destacar los tiempos de ejecución que se muestran en la figura 19. La segunda heurística logra que el algoritmo se ejecute en menos de un segundo.

**Conclusión:** Si se asume el pequeño incremento en la distancia total recorrida y se desea computar los planes de rutas rápidamente a la par que descongestionar el tráfico, la segunda heurística resulta más interesante.

## 3.7. Experimento 7: Buscando la proporción real de conductores

El objetivo de este experimento es determinar que proporción de conductores suele haber al finalizar la ejecución con cualquiera de los algoritmos de búsqueda local propuestos. Se han realizado los experimentos para Hill Climbing:

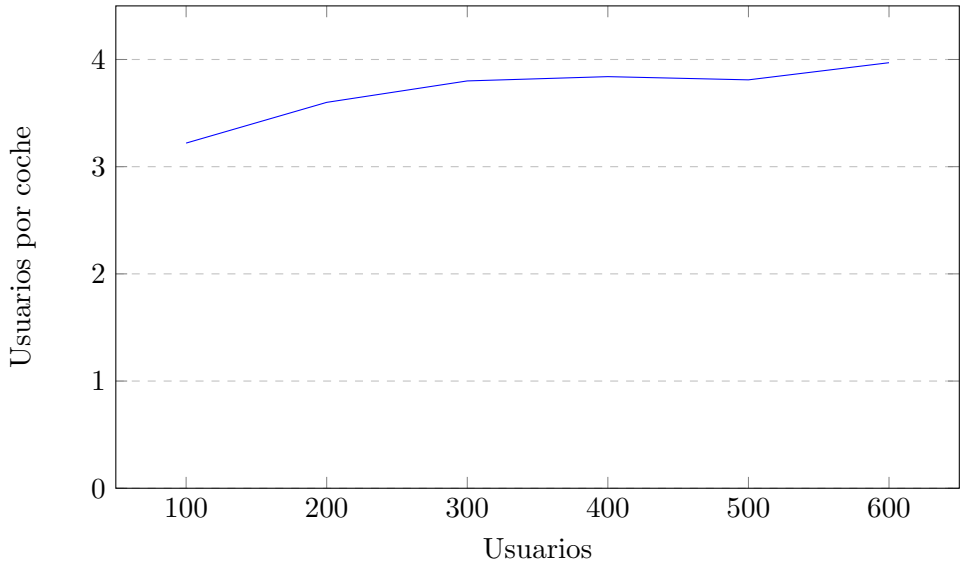


Figura 20: . Gráfico que indica cada tamaño del problema, cuantos coches se necesitaran por usuario

Del gráfico 20 se infiere que cada coche acaba recogiendo de tres a cuatro usuarios. Por lo menos para los tamaños de problema abarcados. Sin embargo no se ha podido experimentar con tamaños de problema superiores a seiscientos, por lo que no sabríamos como crece esta proporción. Sería deseable tratar con tamaños de problema mas grandes pero por limitaciones de memoria y tiempo no se ha podido experimentar más.

A la hora de experimentar con la generación de soluciones iniciales con los datos inferidos, no se han podido generar soluciones iniciales en un tiempo razonable. Esto se debe a que para pocos coches, encontrar una solución inicial no es una tarea trivial, se requiere de un algoritmo sofisticado para cumplir todas las restricciones de distancia ya que a medida que reducimos el numero de coches iniciales, la ruta debe ser mejor ya que:

- Una solución con pocos coches debe aproximarse mucho a un mínimo local, pues el conductor debe cumplir la restricción de llegar a tiempo al trabajo. Si se quiere llevar a muchas personas con pocos conductores, la solución debe ser cercana a la óptima.
- Se deduce que si reducimos el numero de coches, también reducimos la distancia de la ruta, o mejor dicho, todos los coches deben aprovechar al máximo la distancia máxima que pueden recorrer, generando rutas más óptimas. De esto debería encargarse un algoritmo de búsqueda local.