
**Algorismes Bàsics per la Intel·ligència Artificial**

# BÚSQUEDA LOCAL

## Práctica 1

**GIA UPC**
**Curso 2023/2024**

Anna Casanovas
Marta Juncarol
Abril Risso

# ÍNDICE DE CONTENIDOS

Las siguientes páginas describen el proceso de creación de un programa diseñado para ayudar a la empresa Bicing a gestionar la distribución de las bicicletas en sus estaciones, que están repartidas por toda la ciudad. El objetivo es optimizar esta distribución para que los usuarios puedan acceder a una bicicleta cuando la necesiten.

En este informe se presentan los objetivos y motivaciones primordiales del proyecto, y se realiza una exposición detallada de aspectos como la implementación del estado, la selección de operadores, las estrategias para encontrar la solución inicial y la aplicación de funciones heurísticas. También se detalla la ejecución de los experimentos, incluyendo la comparativa entre los algoritmos Hill Climbing y Simulated Annealing.

A lo largo del informe se describen y justifican todas las decisiones tomadas, y se someten a una evaluación crítica. Al finalizar podremos encontrar las discrepancias entre las expectativas iniciales y los resultados obtenidos, las conclusiones extraídas, el aprendizaje adquirido y las dificultades encontradas a lo largo del desarrollo del proyecto.

# 1. Part descriptiva

## 1.1. Identificación del problema

La empresa Bicing enfrenta el desafío de que las bicicletas se acumulan en unas pocas estaciones según la hora del día. Queremos asistir a Bicing en la optimización de esta distribución para facilitar que los usuarios puedan encontrar una bicicleta cuando la necesiten.

Para ello, Bicing ha recopilado datos sobre la demanda y traslado de bicicletas en cada estación por hora. Esta información la podemos encontrar también definida en el fichero *abia_bicing.py* (que se nos da al empezar la práctica) dónde a través de ella se realizan cálculos como el excedente de bicicletas en cada estación o el número total de bicicletas necesarias para suplir la demanda en la ciudad. Toda esta información la resumimos en la siguiente tabla:

| | nomenclatura en *abia_bicing.py* | Descripción |
|---|---|---|
| Bicicletas no usadas | num_bicicletas_no_usadas | Previsión del número de bicicletas de la estación que no se usarán en la siguiente hora |
| Bicicletas estación | num_bicicletas_next | Previsión del número total de bicicletas que habrá en la estación al final de la hora (teniendo en cuenta el transporte de bicicletas causado por los clientes) |
| Demanda | demanda | Previsión del número de bicicletas que debería haber a la siguiente hora en la estación para cumplir con la demanda |
| Diferencia | diferencia | Número total de bicicletas en la estación restando la demanda. Si la diferencia es negativa faltan bicicletas en la estación, si es positiva sobran bicicletas |
| Excedente | excedente | número de bicicletas sobrantes en la estación |
| Acumulación bicicletas | acum_bicicletas | número total de bicicletas en la ciudad |
| Acumulación demanda | acum_demanda | demanda total en todas las estaciones de la ciudad |
| Acumulación bicicletas disponibles | acum_disponibles | bicicletas disponibles o sobrantes en toda la ciudad (excedente acumulado) |
| Acumulación bicicletas necesarias | acum_necesarias | bicicletas necesarias o faltantes en toda la ciudad (diferencia negativa acumulada) |

Para ayudar a Bicing a resolver este problema disponemos de furgonetas que nos ayudarán a transportar las bicicletas de una estación a otra. Nuestro objetivo es hacerlo de forma muy óptima para que al finalizar la hora de duración del problema la distribución de bicicletas sea la mejor posible.

## a. Los elementos del problema

Para simplificar el problema tenemos la siguiente información de la ciudad:

| elemento | tamaño |
|---|---|
| ciudad | **10 x 10 km** |
| manzanas | **100 x 100 m** |

También sabemos que en la ciudad hay un número E de estaciones, colocadas aleatoriamente (la colocación aleatoria se hace en el código que se nos proporcionaba en el *abia_bicing.py*) en las esquinas de las calles de la ciudad.

En la ciudad encontramos un número B de bicicletas repartidas entre las estaciones. El número de bicicletas debe cumplir una proporción de B = 50E, es decir, habrá 50 bicicletas por cada estación. La estación donde está localizada cada bicicleta también se decide aleatoriamente en el *abia_bicing.py*.

Por último tenemos F, el número de furgonetas. F debe cumplir una proporción máxima de F/5 = E, es decir, cada 5 estaciones habrá como máximo una furgoneta (pueden haber menos). Además, el problema se centrará en una hora de reloj durante la cual las furgonetas sólo podrán hacer un viaje, que consistirá en recoger un máximo de 30 bicicletas en una estación y dejarlas en una o dos estaciones cómo máximo. Dos furgonetas no pueden recoger bicicletas en la misma estación.

Para poder visualizar los elementos del problema hemos hecho un gráfico que representa la ciudad mediante una cuadrícula. Para esta visualización usaremos la semilla predeterminada (42), aunque como sabemos, el objetivo es poder solucionar el problema con cualquier semilla, es decir, para cualquier configuración del problema dentro de las reglas establecidas. Además de la configuración de la ciudad, también podemos ver representada la diferencia de cada estación. Los colores más cálidos (amarillo, naranja y rojo) són para las estaciones con una diferencia positiva, es decir, en las que sobran bicicletas, y los colores más fríos (rosa, lila y azul) para las que faltan bicicletas.

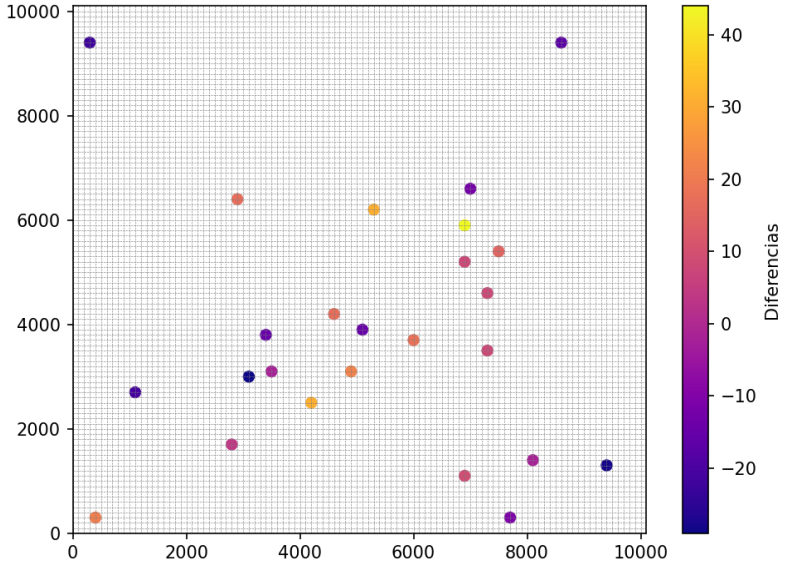


**Figura 1:** Visualización del escenario inicial del problema con semilla 42

## b. Problema de búsqueda local

La búsqueda local se puede entender como un proceso iterativo que empieza en una solución y la mejora realizando modificaciones locales. Esto sucede cuando a veces el camino para llegar a la solución no es importante sinó obtener la mejor solución entre las posibles dentro de un tiempo lógico. En este problema podemos encontrar claramente una similitud con la descripción de búsqueda local, ya que tal y como se describe en el enunciado de la práctica, este problema tiene como objetivo (es decir, solución final) obtener el máximo beneficio, y no nos preguntamos en ningún momento cómo se ha llegado a ello. Es decir, no interesa cómo se llega a la solución sinó que se trata de conseguir algoritmos útiles para que posteriormente, mediante la función heurística, se evalúe la calidad de solución en un tiempo limitado.

La búsqueda local tiene la desventaja de quedar atrapada fácilmente en mínimos locales y su solución final depende fuertemente de la solución inicial. Por este motivo, debemos crear como mínimo dos estrategias para obtener una solución inicial con un coste computacional reducido, pero que permitan observar qué pasa si se utiliza una inicialización simple o una más elaborada. Por otro lado, la búsqueda local tiene la ventaja de encontrar soluciones rápidamente moviéndose a través de soluciones vecinas mediante los operadores hasta encontrar una de mejor, y así sucesivamente hasta que no se pueda mejorar la solución actual. Para lograrlo habrá que implementar un conjunto de operadores eficaces, probando entre diferentes alternativas.

Por último, para implementar la búsqueda local debemos utilizar los algoritmos Hill Climbing y Simulated Annealing. Para cada algoritmo será necesario crear un método generador de estados distinto dado que la manera de funcionar de cada algoritmo es diferente,

## c. La solución y su evaluación

Para evaluar la calidad de una solución debemos fijarnos en dos criterios:

1. Maximización de lo que obtenemos por los traslados de las bicicletas
2. Minimización de los costes de transporte de las bicicletas

Para cuantificar el primer criterio utilizaremos una función de cálculo que para cada estado posible deberá calcular las ganancias. Sumará un euro por cada bicicleta que sea colocada correctamente y restará un euro por cada bicicleta mal colocada. Por otro lado, para cuantificar el segundo criterio debemos calcular la distancia recorrida por las furgonetas. Se nos pide hacerlo con la distancia de Manhattan $d(i, j) = |ix - jx| + |iy - jy|$ y teniendo en cuenta que el coste en euros por kilómetro recorrido es ((nb + 9) div 10), dónde nb equivale al número de bicicletas.

El objetivo de la práctica es también hacer una implementación del estado que tenga eficiencia espacio temporal. Recorrer el espacio del problema para la búsqueda local implica generar múltiples estados y sucesores. Como consecuencia, las implementaciones deben ser óptimas para que el programa tenga un tiempo de ejecución reducido.

## 1.2. Representación del estado

Nuestro trabajo consiste en 7 documentos. A continuación se explicarán sus usos, significado y funcionamiento.

### a. abia_bicing.py

Al inicio de la práctica se nos proporcionaba este documento con dos clases. Éstas se llaman Estacion y Estaciones, y a partir de ellas realizamos la representación de nuestro estado.

ESTACIÓN: la primera clase que se nos proporcionó representa una estación de bicing. En ella tenemos las coordenadas de dicha estación, el número de bicicletas que no se van a usar en la siguiente hora (*self.num_bicicletas_no_usadas*) y una aproximación del número de bicicletas que habrá en la estación en la siguiente hora (*self.num_bicicletas_next*).

ESTACIONES: esta clase representa el conjunto entero de estaciones. Se trata de una lista ordenada que contiene instancias de la clase Estacion. En esta clase se genera lo que tiene cada estación. Dadas un número de estaciones, un número de bicicletas y una semilla, se encarga de crear todas las estaciones y decidir su ubicación, y también reparte las bicicletas. Finalmente se calcula una demanda para cada estación.

### b. clase_furgoneta.py

Para seguir un poco con el formato que nos proporcionó la práctica, para representar el estado decidimos hacer una clase Furgoneta, que está en su propio documento. Esta clase nos servirá para almacenar el recorrido de las furgonetas y las bicicletas en cada estado.

FURGONETA: cada furgoneta la representamos con el número de la estación de origen (de donde parte la furgoneta, *self.est_inicial*), el número de la primera estación en la que deja bicis (*self.est1*) y el número de la segunda estación (*self.est2*) donde las deja. Puede ser que la furgoneta solo tenga una destinación y en ese caso el número de la segunda estación será None. Adicionalmente, en esta clase se almacena el número de la furgoneta (*self.id_furgo*), cuantas bicis coge de la primera estación (*self.bicis_cogidas*) y las bicicletas que deja en la primera estación (*self.bicis_dejadas_est1*). En el caso de que exista una segunda estación en la que dejamos bicicletas, se dejará la resta entre bicis cogidas y bicis dejadas en la estación 1.

### c. clase_parametros.py

En un documento aparte, hemos creado una clase llamada Parámetros.

PARÁMETROS: en la siguiente clase se almacenan todos los elementos de nuestro estado, los que usaremos para encontrar una solución. La clase recibe como parámetro una instancia de Estaciones, un número de furgonetas y la semilla para la primera función generadora del estado inicial. A partir de esta información se crea una lista con las instancias de Estacion (*self.estaciones*) y otra con todas las furgonetas disponibles (*self.furgonetas*). Las furgonetas de momento se crean vacías: no tienen ni estación de origen(None), ni estaciones destino(None) ni bicicletas(0).

### d. clase_operadores.py

Un problema de búsqueda local requiere tener una solución inicial. Esta solución la va modificando hasta conseguir una solución más óptima. Para hacer estas modificaciones se utilizan operadores. En un cuarto documento, hemos creado la clase *OperadorProblemasBicing*, con seis operadores que explicaremos posteriormente. Hemos creado una subclase de *OperadorProblemasBicis* para cada uno de estos operadores. Sus funciones se explicarán en el siguiente apartado (implementación de los operadores).

### e. clase_estado.py

Como hemos comentado anteriormente los problemas de búsqueda local necesitan una solución inicial de la que partir y buscar soluciones mejores. Al final de este documento se hallan 3 funciones generadoras del estado inicial distintas: *generar_estado_inicial1*, *generar_estado_inicial2* y *generar_estado_inicial3*. En el apartado 1.5 se hace una explicación más a fondo de estas funciones. Pero en este documento no podemos encontrar solamente la generación del estado inicial, sinó que también encontramos la clase Estado.

ESTADO: Esta clase es la más importante, dado que es la encargada de generar y aplicar los posibles operadores. Como parámetros le proporcionamos la lista de estaciones y la lista de furgonetas (*self.furgonetas, self.estaciones*). Al inicio tenemos un método *copy*, para poder hacer modificaciones a las furgonetas y estaciones de manera correcta, y el método *__repr__* para poder representar el estado. Hemos decidido representar el estado con la información de las furgonetas (dado que así vemos los cambios implementados por nuestro programa respecto al escenario inicial), las ganancias obtenidas según cada heurístico, y la distancia total recorrida por las furgonetas. Para verlo más claramente, en el fichero *main.py* hemos implementado una función para visualizar el estado inicial gráficamente. Por ejemplo, para el heurístico 2, el algoritmo Hill Climbing y un escenario de 25 estaciones, 1250 bicicletas y 5 furgonetas generado con una semilla de 42, el resultado es el siguiente:

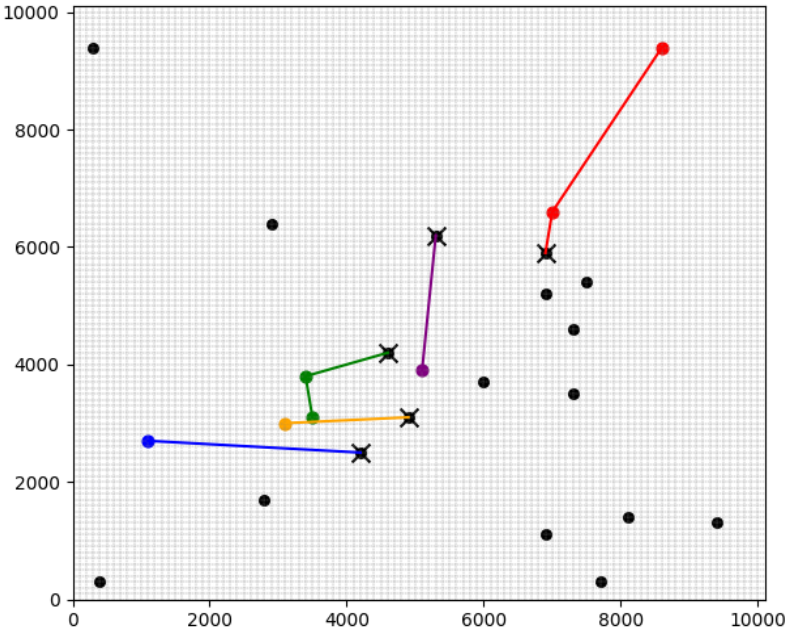


**Figura 2:** Visualización del estado final con semilla 42, Hill Climbing y heurístico 2

A continuación explicamos los demás métodos:

Generar acciones: En la práctica usamos los algoritmos de búsqueda local Hill Climbing y Simulated Annealing. El funcionamiento del primero, Hill Climbing, es generar todas las acciones posibles con el estado actual, que es lo que se hace en este método. Éste devuelve un generador. Los generadores generan elementos bajo demanda, a medida que los necesitas, en vez de almacenar todos los elementos en la memoria a la vez. Es muy eficiente y con un rendimiento alto.

Generar una acción: En cambio, cuando queremos usar el algoritmo Simulated Annealing, éste solo necesita un sucesor aleatorio. Lo que hacemos en este método es en vez de realizar un *yield* de cada acción que encontrábamos, la guardamos en un set. Más tarde, tendremos un set con todas las operaciones que se pueden hacer y eso para cada operador. Aleatóriamente se elegirá una acción de todas estas posibles y será la que evaluará Simulated Annealing.

Aplicar acciones: independientemente de qué método hayas usado para generar las acciones, cuando hayamos elegido cual queremos, la tenemos que aplicar. Este método recibe como parámetro una acción, mira de qué clase es instancia la acción dada y a partir de eso realiza la acción y devuelve un estado con esa acción aplicada.

Heurísticos: Para poder elegir qué acciones són mejores y aplicarlas, los algoritmos de búsqueda local requieren una función heurística. Como nos pide en el enunciado, hemos hecho dos funciones diferentes que responden a diferentes objetivos. Para poder hacer los heurísticos se han necesitado otros métodos adicionales que serán explicados con más profundidad en el apartado 1.4.

### f. clase_problema.py

Dentro de este documento tenemos una clase muy importante, que la hemos llamado *BicingProblem*. Es una subclase de Problem, una clase de AIMA. Uno de los parámetros es nuestro estado inicial, que es una instancia de la clase Estado. Los otros dos parámetros son booleanos que indican qué algoritmo de búsqueda local y qué función heurística queremos usar. En el método *actions,* si usamos el Hill Climbing ejecutará el método *generar_acciones* de la clase Estado, y sinó ejecutará el método *generar_una_accion*. Esto genera posibles acciones a tomar. Después, en el método *result*, le damos una acción que eligirá el algoritmo, y luego le aplica la acción a el estado actual. El resultado de este método es el estado resultante de aplicar una acción. El método value calcula como de buena es nuestra solución actual (el estado), así que retornará uno de los heurísticos que hemos implementado.

### g. main.py

Finalmente, el único documento que hemos añadido es main. Aquí, al elegir qué experimento queremos realizar, se crearán los parámetros correspondientes, se generarán las instancias de las clases y se llamará al Hill Climbing o Simulated Annealing. Adicionalmente, hemos creado una función para representar la solución de nuestro problema. Esta función se llama al final de cada ejecución del programa y muestra el mapa de nuestra ciudad, ubicando las estaciones y con líneas de colores se muestran los caminos elegidos de las furgonetas.

## 1.3. Implementación de los operadores

En la búsqueda local, se parte de una solución inicial, y los algoritmos utilizan operadores para generar estados sucesores y encontrar una mejor solución. Nosotras después de un tiempo de reflexión, implementamos seis operadores para mejorar nuestra solución inicial. Debemos tener en cuenta que hemos implementado las debidas comprobaciones en los métodos de generar y aplicar acciones de la clase Estado para que ningún operador nos saque nunca del espacio de soluciones. A continuación explicaremos los operadores, y también calcularemos su factor de ramificación ayudándonos del código del método *generar_acciones* de la clase Estado. Estudiaremos si son elevados o no, contrastándolos con el escenario de 25 estaciones, 1250 bicicletas y 5 furgonetas.

### a. Intercambiar destino furgonetas

La función de este operador es intercambiar el destino de dos furgonetas. Al operador le damos las 2 furgonetas de las cuales queremos cambiar sus destinos, y las dos estaciones que queremos que sean intercambiadas. Esto quiere decir que podemos intercambiar el primer destino de la furgoneta 1 con el primer o el segundo destino de la furgoneta 2, y también al revés. Para aplicar el operador, al crear una copia del estado, a las dos furgonetas les asignamos un identificador diferente que queremos intercambiar.
Este operador, en el peor de los casos (que sería que todas las bicicletas tuvieran dos estaciones destino), tiene un factor de ramificación de $4 \cdot (F-1) + (F-1)!$ siendo F el número de furgonetas. Con el escenario predeterminado el factor de ramificación tiene un valor de 40, lo que no es un número demasiado elevado por ser un operador.

### b. Reasignar furgoneta a estación

La función de este operador es parecida a la del anterior pero solo se hace con una furgoneta. Éste se basa en cambiar uno de los destinos. Podemos coger cualquier estación que no sea la inicial o las dos de origen de esa furgoneta y cambiarlas por una estación nueva.
Para calcular su factor de ramificación suponemos el peor de los casos (que sería que todas las bicicletas tuvieran dos estaciones destino). Si cada furgoneta (F) tiene dos estaciones (2) para cada una de ellas tendremos la posibilidad de que el operador cambie esa estación por alguna de las demás estaciones. Teniendo en cuenta que la nueva estación no puede ser ni la actual estación origen ni las actuales estaciones destinos (E-3), podemos concluir que el operador tiene un factor de ramificación de $2 \cdot (E-3) \cdot F$. Con el escenario predeterminado el factor de ramificación es de 220, bastante elevado.

### c. Reorganización de las entregas

Este operador consiste en cambiar el orden en el que la furgoneta entrega las bicis en las estaciones destino. La furgoneta que recibe cómo parámetro debe ser una furgoneta con dos estaciones de destino. El operador intercambia el orden de las estaciones destino de forma que en lugar de pasar primero por la que antes era la estación 1 y después por la 2, ahora el recorrido es al revés. El número de bicicletas que se deja en cada estación es el mismo, debido a que este operador está solamente pensado para reducir la distancia que recorre la furgoneta.
El factor de ramificación del operador es el número de furgonetas con dos estaciones, es decir, en el peor de los casos F. Es por tanto, un factor de ramificación muy pequeño.

### d. Ajustar cantidad de bicis del origen

Este operador nos permite cambiar el número de bicicletas que cogemos con una furgoneta de una estación. Con este nuevo número de bicicletas escogido, si solo hay una estación de destino las dejamos todas en esa estación. Sinó, siempre se prioriza satisfacer la demanda de la estación de destino uno antes que la segunda, eso hace que a veces este operador haga que se elimine el segundo destino de la furgoneta a causa de la falta de bicicletas.
Dado el caso de que todas las furgonetas estuvieran en uso este operador generaría (n-1)·F estados sucesores, siendo n el número máximo de bicicletas que puede cargar una furgoneta (en nuestro problema es 30). El factor de ramificación 29·F es elevado pero no especialmente.

### e. Reasignación de bicicletas

Este operador recibe cómo parámetro una furgoneta que tiene estación origen y dos estaciones destino, y su función es cambiar el número de bicicletas que se deja en cada estación destino. Esto significa que genera todas las posibles combinaciones de distribución de bicicletas entre las dos estaciones destino sin cambiar el número de bicicletas cogidas en la estación origen.
Este operador tiene un factor de ramificación de 28·F porque si cada furgoneta (F) cogiera el máximo de bicicletas (n, que en nuestro problema es 30), estas pueden ser distribuidas entre las estaciones destino de n-1 formas diferentes (teniendo en cuenta que ninguna estación destino queda sin recibir bicicletas), y queremos generar un estado distinto al presente así que multiplicaremos por (n-2).

### f. Añadir segunda estación

Con añadir segunda estación, le proporcionamos un número de bicis y una estación, y el operador pasará de dejar todas sus bicis en una sola estación estación destino, a solo dejar el número de bicis proporcionado, y luego el resto de bicicletas dejarlo en la segunda estación escogida.
Si todas las furgonetas tuvieran solamente una estación destino (F) y hubieran cogido el número máximo de furgonetas en la estación origen (n), podríamos asignarles otra estación destino de las E-2 disponibles, dado que la nueva estación no puede ser la misma que las otras dos estaciones por las que pasa. Podemos redistribuir el número de bicicletas en cada estación de n-1 formas diferentes. Por estos motivos el factor de ramificación de este operador es F·(E-2)·29. Con el escenario predeterminado esto equivale a 3335, es decir, es un factor de ramificación muy elevado.

## 1.4. Descripción de la función heurística

Para realizar la búsqueda local, es necesaria la creación de una función heurística para poder evaluar la calidad de nuestra solución. Teniendo en cuenta que las estrategias de generación de la solución inicial (que explicaremos en el siguiente apartado) generan siempre una solución dentro del espacio de soluciones, y que los operadores nos sacan en ningún momento de él, no es necesario penalizar las no soluciones en el heurístico, dado que nunca ocurrirán.

En el algoritmo Hill Climbing la función heurística es necesaria para la generación de nuevas soluciones (soluciones vecinas), donde se hacen pequeñas modificaciones en la solución actual

aplicando las acciones generadas con los operadores. De esta manera, este algoritmo selecciona la solución vecina con el valor heurístico más alto. Cuando el algoritmo llega a un máximo local, se detiene.
Para el algoritmo de Simulated Annealing, la función heurística se utiliza para lo mismo. En cambio, a la hora de generar nuevas soluciones es un poco diferente. En lugar de seleccionar la mejor solución vecina, utiliza la función heurística para determinar si se acepta o no la solución. Esta nueva solución podrá ser aceptada incluso si el heurístico es menor a la anterior (si cumple unas condiciones de probabilidad específicas). A medida que avanzan las iteraciones, la probabilidad de aceptar soluciones con un heurístico peor disminuye.

Para resolver la situación que se nos plantea en la práctica hemos decidido crear dos heurísticos que se calculan cada vez que se aplica una acción para así evaluar las siguientes soluciones generadas:

### a. Heurístico 1

Con este heurístico buscamos maximizar las ganancias obtenidas por el traslado de bicicletas, teniendo en cuenta que para cada bicicleta que colocamos correctamente ganamos un euro y para cada error perdemos uno. Es decir, no tenemos en cuenta la distancia recorrida por las furgonetas y solamente tendremos en cuenta la mejora de la demanda al haber aplicado nuestros operadores. Para ello tenemos un método llamado *actualizar_diferencias*, donde a partir de las diferencias iniciales modificamos la lista quitando el número de bicicletas cogidas de las estaciones iniciales y añadiendo este número a las estaciones 1 o 2 según las hayan dejado nuestras furgonetas. Después de actualizar la lista de diferencias, calculamos si hemos mejorado la demanda con la función *calculo_coste_diferencias*, donde compara la lista de diferencias anterior con la actualizada. Con un *for* recorremos cada elemento de la lista de diferencias y lo comparamos con la inicial (la anterior a aplicar el nuevo operador). Si la diferencia inicial es mayor a 0 (sobraban bicis) y ahora es menor a 0, restamos al resultado del coste el número de bicicletas que nos hemos pasado de coger ya que ahora faltan bicis (en nuestra función sumamos el número negativo). Si sigue siendo mayor a 0 no restamos nada ya que no hemos empeorado el estado de la estación, no faltaban bicicletas y ahora tampoco faltan. Si la diferencia inicial es menor a 0 (faltaban bicicletas) y la diferencia actualizada es mayor a 0, sumamos el número de la diferencia inicial ya que ahora ya no faltan bicicletas en la estación. En cambio si el valor de la diferencia actualizada sigue siendo menor, sumamos al resultado el número de bicicletas que hemos conseguido transportar *(abs(self.diferencias_inicial[i])-abs(self.diferencias[i]))*.

### b. Heurístico 2

En cambio en el segundo heurístico si tenemos en cuenta la distancia recorrida por las furgonetas. Este heurístico se calcula de la misma manera que el primero pero a este le hemos añadido la función llamada *calculo_coste_kilometros*, donde calculará la distancia recorrida. Es decir, en esta función también actualizaremos la lista de diferencias y compararemos las dos listas de diferencias (la anterior en aplicar el operador y la actualizada después de aplicarlo) y calcularemos si hemos mejorado la demanda, y por último aplicaremos el método *calculo_coste_kilometros* y restaremos al cálculo de las diferencias el coste de los kilómetros. En este método, para calcular las distancias utilizamos la distancia de Manhattan, que la calculamos a partir de las coordenadas de cada estación. Si la furgoneta tiene asignada una primera estación, calculamos la distancia entre la estación inicial y esta (con Manhattan) y calculamos el coste que esta distancia nos supone con la fórmula que nos

proporcionaban en el enunciado del problema: *costo1 = ((furgoneta.bicis_cogidas + 9) // 10) * distancia.* Si la furgoneta también tiene asignada una segunda estación, hacemos otra vez este cálculo pero en vez de con la estación inicial y la primera estación, con la primera estación asignada y la segunda y seguidamente añadimos el coste al resultado. Finalmente, este coste calculado lo dividimos entre 1000 ya que todo este cálculo es en metros y nosotras lo queremos en kilómetros.

## 1.5. Estrategias para hallar la solución inicial

Para llevar a cabo la búsqueda local se necesita una función de estado inicial a partir de la cuál empezar a recorrer el espacio de búsqueda y encontrar mejores soluciones contiguas. Después de reflexionar sobre que definía una solución inicial, decidimos implementar dos soluciones iniciales, una más sencilla y una más compleja. Pero a medida que avanzábamos en la práctica nos dimos cuenta de que necesitaríamos otra solución inicial con una complejidad intermedia respecto a las que ya teníamos. Por este motivo, implementamos una tercera solución inicial. Debemos tener en cuenta que estas soluciones iniciales son todas soluciones posibles del problema, así que es correcto afirmar que se empieza la búsqueda local dentro del espacio de soluciones.

### a. Función 1

La primera función, aún siendo la más sencilla, no quisimos hacerla completamente aleatoria. Esta función de generación del estado inicial busca colocar las furgonetas en estaciones con diferencia positiva (sobran bicicletas) y llevarlas a otras estaciones aleatoriamente.
La función, llamada *generar_estado_inicial1* consta de un *for* que recorre la lista de estaciones (uno de los parámetros creados en nuestra implementación de estado) y va asignando furgonetas de recogida de bicicletas a medida que encuentra estaciones con una diferencia positiva. Recoge todas las bicicletas que sobran y asigna la primera estación destino de la furgoneta aleatoriamente, con la semilla dada en los parámetros. A continuación, también de forma aleatoria y con la misma semilla, se elige si la furgoneta tendrá o no estación destino 2, y en caso de tenerla, se elige (también aleatoriamente) cuál será y cuántas bicicletas se dejarán en cada una de las dos estaciones.
La semilla que se utiliza en esta función la hemos determinado con valor 12 así como podría ser cualquier otro valor, porque queríamos que el *rng* fuera aleatorio. Asimismo en el segundo experimento probaremos con distintas semillas y determinaremos un valor óptimo.
Dado que, tal y como hemos explicado, la función consta de un *for* que recorre la lista de estaciones, podemos decir que esta función tiene un coste O(E) siendo E el número de estaciones.

### b. Función 2

La segunda función de generación del estado inicial que hicimos es la más eficaz ya que su objetivo es encontrar las mejores estaciones de recogida y de dejada de bicicletas para todas las furgonetas. En el código la función consta como *generar_estado_inicial3* pero la explicaremos en segundo lugar dado que nos parece más lógico explicar las soluciones en el orden en que fueron pensadas.
Al inicio de la función encontramos un bucle *for* que recorre la lista de estaciones calculando sus excedentes (y en el caso de ser 0 calcula sus diferencias) y metiendo esta información en dos listas. La primera, *est_y_ex_ordenadas* es una lista de tuplas que contienen en la primera posición el

identificador de la estación, y en la segunda su excedente o diferencia. Esta lista la ordenamos después con un *sort* en orden decreciente según el excedente o diferencia de las estaciones. La segunda lista, *recuento_exc_demanda,* tan solo guarda los excedentes o diferencias siguiendo el orden de los identificadores de las estaciones.
Ahora se asignan como estaciones de origen de las furgonetas las primeras estaciones que se encuentran en la lista *est_y_ex_ordenadas,* de modo que las furgonetas recogerán bicicletas en las estaciones con más excedente. A continuación se asignan como estaciones destino de las furgonetas las últimas estaciones de la lista, es decir, las que faltan más bicis. En el caso que estas estaciones no necesiten todas las bicicletas que lleva la furgoneta, se le asignará a la furgoneta una segunda estación destino donde dejarlas. La información de bicicletas dejadas en cada estación se va guardando en la lista *recuento_exc_demanda*, modificándola, de modo que no haya ningún posible fallo.

La función tiene un coste de O(E·F) porque al inicio se recorre la lista de estaciones para calcular las diferencias y excedentes, y luego se recorre la lista de furgonetas. Con esta función *greedy*, se consigue que las furgonetas coloquen en su sitio el mayor número de bicicletas posibles, consiguiendo de este modo las máximas ganancias según el primer heurístico. Es decir, al ejecutar la búsqueda de estados mejores con el heurístico que maximiza las ganancias obtenidas por el traslado de bicicleta, nunca se encuentran estados mejores ya que el estado inicial ya da las ganancias más elevadas. Esto supone un problema dado que partiendo de este estado no se utiliza ningún operador ni se viaja a través del espacio de estados, y en los experimentos no podremos, por ejemplo, determinar los operadores más eficientes. Por este motivo hemos implementado una tercera función de generación del estado inicial.

### c. Función 3

Esta función, llamada *generar_estado_inicial2* en el código, se sitúa en un punto intermedio de eficacia respecto a las dos anteriores. El código es muy similar al de la función anterior ya que crea las mismas dos listas al inicio, y asigna las estaciones de origen de las furgonetas con el mismo criterio. Sin embargo, en esta función se utiliza un for que recorre la lista de estaciones y asigna estaciones destino a las furgonetas a medida que encuentra estaciones con una diferencia negativa (les faltan bicicletas). Todas las furgonetas tienen una sola estación destino en la que dejan todas las bicicletas que llevaban. La diferencia entre esta función y la anterior es que las estaciones destino no son asignadas de la mejor forma posible. De esta manera, aún siendo un estado inicial muy bueno, tiene aún margen de mejora para el primer heurístico. Tiene el mismo coste que la función anterior, O(E·F).

# 2. Parte experimental

En esta sección se realizarán una serie de experimentos relacionados con la calidad de nuestra implementación.

## 2.1. Experimento 1

### Descripción:

Podemos observar que hay diferentes operadores que podríamos implementar para nuestra solución. Las opciones són infinitas, pero en nuestro caso hemos implementado nuestro estado con los seis operadores descritos anteriormente en la documentación. En este experimento analizaremos el efecto de estos operadores y sus combinaciones para encontrar la combinación óptima (el conjunto de operadores que da mejores resultados).

Condiciones del experimento:

- Usaremos la función heurística 1, que maximiza las ganancias obtenidas por el traslado de las bicicletas y no tiene en cuenta los kilómetros recorridos.
- El escenario será de 25 estaciones, 1250 bicicletas y 5 furgonetas.
- Usaremos el algoritmo Hill Climbing.
- Usaremos la generación de estado inicial número 1. En próximos experimentos podremos observar que esta no es la mejor solución, pero como empieza desde un punto aleatorio, así podremos observar mejor si los operadores están haciendo bien su efecto.

Nuestra hipótesis nula es que todos los operadores nos dan las mismas ganancias y sus combinaciones también. Veremos durante el experimento si esta afirmación es verdadera.

### Ejecución y resultados:

Hemos elegido 10 semillas aleatoriamente: [42,30,17,24,29,33,9,57,61,76]. Con cada una de estas semillas se realizarán 3 réplicas con diferentes semillas para la función generadora de estado inicial y se harán las medias de los resultados. Nos hubiera gustado probar todas las combinaciones posibles entre las 6 operadores, pero son demasiadas combinaciones y solo con algunas ya hemos podido ver un patrón y nos permitirá encontrar un resultado de nuestro experimento. Muchas de las combinaciones ya no las hacemos porque no nos dan más información.

Para nuestro análisis hemos partido de las mejoras de cada operador por su cuenta.

En la gráfica, cada experimento corresponde a una de las 10 semillas que hemos mencionado antes:

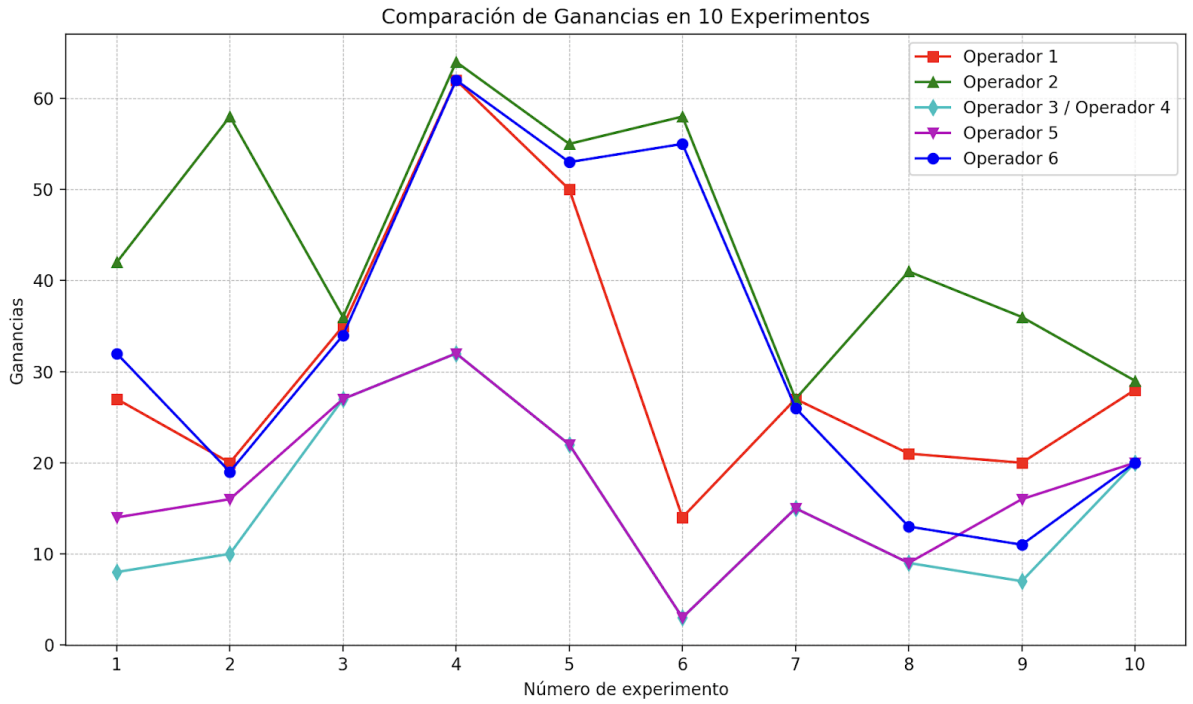


**Figura 3:** Gráfico de la comparación de las ganancias de los 6 operadores en 10 experimentos

De primeras observando el gráfico ya se ve cuales operadores són eficaces y cuáles no. En el caso del experimento con los operadores 3 y 4, se observa que hay unas ganancias muy bajas. De hecho, esos són los mismos valores de las ganancias que da la solución inicial. El operador 5 también tiene valores muy bajos, y en casi todos los casos iguales a los de la solución inicial.
En cambio, los operadores 1,2 y 6, vemos que són eficaces y aportan muchas ganancias, en la mayoría de experimentos, en el caso de la semilla número 3,4,5,6 y 7, los tres por su cuenta llegan prácticamente a la misma solución.

## Conclusiones:

Definitivamente, después de observar los resultados, podemos rechazar nuestra hipótesis nula y decir que depende del operador o combinación de operadores que uses para el problema, tendrás unas ganancias muy diversas.

Podemos concluir que los operadores 3, 4 y 5 no están haciendo nada para mejorar nuestra solución. También realizamos el experimento usando los 6 operadores, y nos dimos cuenta que eran los mismos valores que usando el operador 2 por su cuenta.

La mejor solución se da cuando aparece el operador 2. Si, por ejemplo, combinamos los tres en los que sí que hemos visto beneficios por su cuenta, también nos dan los resultados de ganancias máximas però tarda más en ejecutar-se. El operador dos por su cuenta tarda una media de 44,9 milisegundos y usando los operadores 1, 2, 6 tarda una media de 168,1545 ms.

Aun así, creemos que con esos 3 operadores juntos se pueden explorar más estados que solo con el operador 2,y por eso nos quedaremos con los operadores 1, 2, 6. Esta combinación tiene las ganancias máximas y solo usa operadores que han mostrado un beneficio.

## 2.2. Experimento 2

### Descripción:

Este experimento pide determinar qué estrategia de generación de la solución inicial da mejores resultados para la primera función heurística, con el escenario del apartado anterior, con los operadores seleccionados en el apartado anterior y usando el algoritmo de Hill Climbing.

Tal y como se ha explicado en la parte descriptiva, tenemos tres funciones generadoras de estados iniciales. La primera coloca las furgonetas en estaciones con bicicletas sobrantes (excedente) y las reparte aleatoriamente en otras estaciones. La segunda coloca las furgonetas en las estaciones con más excedente y asigna como única estación destino de las furgonetas alguna estación dónde falten bicicletas. La última función, coloca las furgonetas en las estaciones con más excedente y las lleva hacía las estaciones con más diferencia negativa (las que faltan más bicicletas).
Nuestra hipótesis nula para este experimento es que la tercera función, *generar_estado_inicial3,* será la estrategia de generación de la solución inicial que dará mejores resultados.

Utilizaremos la siguiente nomenclatura para nombrar las diferentes funciones generadoras del estado inicial:

- *generar_estado_inicial1* → función 1 o aleatoria
- *generar_estado_inicial2* → función 2 o media
- *generar_estado_inicial3* → función 3 o greedy

### Ejecución y resultados:

Ejecutaremos 10 experimentos, cada uno con una semilla distinta para la generación del espacio del problema (colocación de las 25 estaciones y las 1250 bicicletas en la ciudad y generación de las demandas), con el objetivo de poder comparar diferentes escenarios, y no siempre partir de la misma base. En el caso de la primera función, que utiliza una semilla para hacer parte de la inicialización aleatoriamente, haremos 5 ejecuciones de cada experimento con semillas diferentes y haremos la media de los resultados.

| | GANANCIAS INICIALES | | | GANANCIAS FINALES | | | TIEMPO EJECUCIÓN (ms) | | |
|---|---|---|---|---|---|---|---|---|---|
| | aleatoria | media | greedy | aleatoria | media | greedy | aleatoria | media | greedy |
| **1** | 8 | 42 | 95 | 42 | 95 | 95 | 332.10 | 529.46 | 179.09 |
| **2** | 10 | 68 | 77 | 58 | 77 | 77 | 332.58 | 386.11 | 160.09 |
| **3** | 27 | 76 | 87 | 36 | 87 | 87 | 203.01 | 441.09 | 170.35 |
| **4** | 32 | 49 | 79 | 64 | 79 | 79 | 315.09 | 486.15 | 115.23 |
| **5** | 22 | 46 | 94 | 55 | 94 | 94 | 387.09 | 604.08 | 134.44 |
| **6** | 3 | 62 | 90 | 58 | 90 | 90 | 311.08 | 662.00 | 106.10 |
| **7** | 15 | 36 | 92 | 27 | 92 | 92 | 95.09 | 540.00 | 189.40 |
| **8** | 9 | 32 | 86 | 41 | 86 | 86 | 264.10 | 794.08 | 164.07 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **9** | 7 | 41 | 81 | 36 | 81 | 81 | 220.08 | 512.00 | 171.08 |
| **10** | 20 | 44 | 59 | 29 | 59 | 59 | 122.04 | 327.08 | 119.10 |

**Figura 4:** Tabla comparando las ganancias iniciales y finales y el tiempo de ejecución de cada función generadora de la solución inicial

Con los resultados de los experimentos hemos elaborado las tablas anteriores donde podemos ver tres aspectos representados. Las ganancias iniciales son las ganancias que aporta la solución o estado inicial, sin la influencia de ningún operador, y las ganancias finales són las ganancias que se obtienen después de aplicar el Hill Climbing y encontrar una mejor solución. También vemos el tiempo de ejecución medio de cada generador de estado inicial en milisegundos.

Vemos como las ganancias finales de la solución media y la greedy son iguales, y equivalen al máximo de ganancias posibles teniendo en cuenta el primer heurístico. Las ganancias finales de la primera función en cambio son mucho más bajas, teniendo algunos resultados que se acercan bastante a las ganancias finales de las otras soluciones (casillas verdes) y otros muy inferiores (casillas rojas). Esto es debido a que la primera solución resulta ser muy mala, y además los operadores implementados no són especialmente eficaces para esta solución.

Para estudiar en más profundidad las ganancias haremos la media de los experimentos y calcularemos la diferencia de ganancias, antes y después de ejecutar el Hill Climbing.

| | ganancias iniciales | ganancias finales | diferencia |
|---|---|---|---|
| aleatoria | 15.3 | 44.6 | 29.3 |
| media | 49.6 | 84 | 34.4 |
| greedy | 84 | 84 | 0 |

**Figura 5:** Tabla comparando la media de las ganancias iniciales y finales y su diferencia de las funciones generadoras del estado inicial

Podemos ver como tanto la solución aleatoria como la media incrementan sus ganancias después de aplicar el algoritmo de búsqueda local, con una diferencia de aproximadamente 30€. La diferencia está en que la primera solución parte de unas ganancias iniciales mucho más pequeñas que la segunda y, por lo tanto, aún incrementando con la misma diferencia las ganancias, la segunda solución da unas ganancias finales mucho más grandes que la primera.
La tercera función en cambio tiene una diferencia de cero, dado que las ganancias iniciales que obtiene ya son las máximas ganancias posibles (motivo por el que se llama función avariciosa). Es decir, no tiene margen de mejora ya que al dar las máximas ganancias posibles no encontrará ningún estado mejor.

Una vez analizadas las ganancias vemos que la segunda y la tercera són las funciones generadoras del estado inicial a partir de las cuáles el algoritmo de búsqueda local funciona mejor y puede llegar a las ganancias más elevadas. A continuación analizaremos el tiempo de ejecución para comparar principalmente la función media y la greedy.

| aleatoria | media | greedy |
|---|---|---|
| 258.23 | 528.21 | 150.90 |

**Figura 6:** Tabla comparando las medias del tiempo de ejecución de las tres funciones

Es evidente que el método avaricioso es mucho más rápido que los otros dos. La explicación más plausible es que la solución inicial es directamente la solución final si usamos este método. Los otros comienzan en una solución más lejana y tienen que recorrer un camino más largo hasta un estado óptimo. Podemos ver cómo la función media es especialmente lenta, sobre todo en algunos casos. Cabe remarcar la importancia de la elección de la semilla generadora del problema porque en la tabla general del tiempo de ejecución podemos ver algunos valores (marcados en rojo) dónde el tiempo de ejecución con la función 2 es especialmente malo comparado con la función 3, mientras que en otros casos (marcados en verde) la diferencia es mucho menor. Aún así, el tiempo de ejecución de la segunda solución inicial no es lo suficientemente alto como para que sea significativo para descartar la función generadora de estado inicial.

## Conclusiones:

Una vez que hemos hecho los experimentos y los hemos analizado podemos sacar conclusiones para elegir un generador del estado inicial. Aunque con la segunda y la tercera función se obtienen las mismas ganancias finales, podemos concluir que se cumple la hipótesis nula y la función avariciosa es claramente la mejor debido al tiempo de ejecución. De todas formas, como ya hemos comentado, esta solución inicial no recorre el espacio de estados en la búsqueda local, ya que inicialmente ya tiene las máximas ganancias. Como consecuencia, si la seleccionamos para realizar los experimentos no podremos analizar ni evaluar los operadores ni otros factores que intervengan en el proceso de búsqueda local. Por este motivo, y dado que lo único que impide que la solución 2 sea la elegida es una diferencia de 377 milisegundos de media (ya que las ganancias de la segunda y tercera solución son equivalentes), la segunda estrategia de generación de la solución inicial será la elegida para realizar el resto de experimentos.

Con este experimento nos hemos dado cuenta de que los operadores que tenemos no son suficientes para cubrir todo el espacio de soluciones, dado que no existe un operador que permita cambiar la ubicación de la estación de origen de las furgonetas. Es por este motivo que, como la primera solución inicial no asigna a las furgonetas las mejores estaciones de origen posibles, nunca se podrá llegar a las máximas ganancias con esta solución inicial. En cambio, con las otras dos soluciones iniciales, sí que se pueden obtener las ganancias máximas, dado que no les afecta la falta de operadores para reasignar la estación de origen de las furgonetas. Este error podría haber supuesto un grave problema si no hubiéramos tenido estas funciones generadoras del estado inicial que colocan las furgonetas inicialmente en las mejores estaciones posibles.

## 2.3. Experimento 3

### Descripción:

Este experimento pide determinar los parámetros que dan mejor resultado para el Simulated Annealing con el mismo escenario, usando la misma función heurística y los operadores y la estrategia de generación de la solución inicial escogidos en los experimentos anteriores. Es decir, usaremos los operadores 1,2 y 6, la segunda función de generación de una solución inicial, y la función heurística 1 (donde solo se maximizan las ganancias).

Nuestra hipótesis nula para este experimento es que los valores de los hiper parámetros influyen en el resultado de las ganancias. Es decir, algunos parámetros harán que el algoritmo consiga maximizar el resultado al óptimo y otros parámetros hagan que este no consiga llegar al resultado óptimo.

### Ejecución y resultados:

El Simulated Annealing tiene los siguientes elementos:

- Una temperatura que será nuestro parámetro de control.
- Una energía, que nos da la calidad de la solución
- Una función de probabilidad de aceptación, que se usa para escoger a sus sucesores. Esta función tiene la siguiente forma:

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

En función de la temperatura y la diferencia de la calidad entre la solución actual y la solución candidata escogerá si cambia de solución. La temperatura es la que se encarga de determinar si la función de aceptación elegirá un sucesor peor o no. Si hay menos temperatura habrá menos probabilidad de elegir sucesores peores.

E no depende del problema y la hemos fijado a un valor -1. Lo que sí que importa en nuestro problema es F, la función de probabilidad de aceptación. Ésta depende del cálculo de la temperatura del sistema, que es el siguiente:

$$F(T) = k \cdot e^{-\lambda \cdot T}$$

Vemos que en esta función se hay unos parámetros $k$ y $\lambda$, que van a ser los que ajustemos en este experimento.

Nuestro objetivo en el experimento es maximizar nuestras ganancias, y para hacerlo queremos llegar a hacer un número de iteraciones mayor a el número de iteraciones donde la probabilidad se hace 0. Queremos que supere este número porque cuando la probabilidad se hace 0, significa que ya no aceptaremos soluciones peores entonces el algoritmo solo mejorará.

El número de iteraciones que tarda para llegar a este punto depende mucho de los dos parámetros comentados anteriormente. El parámetro $k$ cuanto más aumenta, el número de iteraciones donde es probable que coja soluciones peores aumenta. El parámetro $\lambda$, es lo rápido que desciende la probabilidad de escoger una solución peor.

Finalmente, contamos con una estrategia de enfriamiento, que con el número de iteraciones a realizar, nos dirá cómo bajar la temperatura y cuántos sucesores explorar a cada paso de temperatura.

Con estos conocimientos, hemos cogido un número grande de iteraciones (10000) y hemos probado diferentes combinaciones de λ (0.99, 0.1, 0.01, 0.001) y de *k* (5, 50, 100, 125).

En este experimento usaremos la semilla 42, y para cada combinación de lambda, k, y el número de iteraciones realizamos 10 repeticiones. En esta tabla podemos ver la media hacha entre las 10 repeticiones del tiempo y las ganancias para cada combinación de k y lambda, con 10000 iteraciones.

| **λ →** | **0.99** | | **0.1** | | **0.01** | | **0.001** | |
|---|---|---|---|---|---|---|---|---|
| **k** | tiempo (ms) | ganancias (€) | tiempo (ms) | ganancias (€) | tiempo (ms) | ganancias (€) | tiempo (ms) | ganancias (€) |
| 5 | 296.22 | 94.5 | 2759.52 | 95 | 2585.61 | 95 | 2719.54 | 95 |
| 50 | 297.43 | 94.2 | 2847.56 | 95 | 2637.35 | 95 | 2758.17 | 92 |
| 100 | 293.82 | 92.2 | 2799.32 | 95 | 2636.23 | 94,6 | 3017.06 | 85,4 |
| 125 | 297.54 | 94.8 | 2818.13 | 95 | 2703.11 | 92,6 | 3105.05 | 86,3 |

**Figura 7:** Tabla comparando los tiempos y ganancias medios para diferentes valores de lambda y k, con 10000 iteraciones

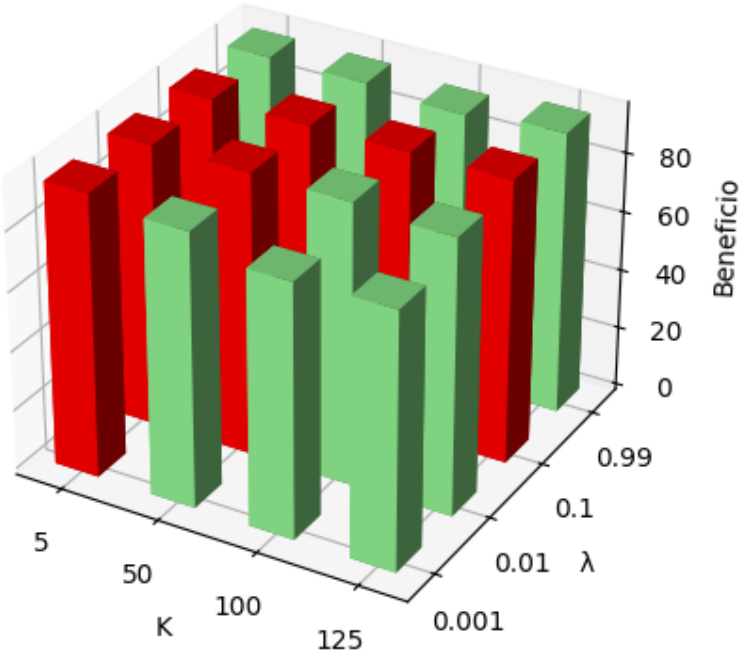


**Figura 8:** Gráfico de las ganancias medias para diferentes valores de lambda y k, con 10000 iteraciones

Este gráfico nos muestra la media del beneficio de 10 repeticiones según los hiper parámetros escogidos. Podemos observar que prácticamente la mitad de los beneficios llegan a 95, el máximo

beneficio que se puede obtener (está indicado con las barras que son de color rojo).

También podemos ver que ninguno de los valores con lambda igual a 0.99 llega a 95 de beneficio, todas las ganancias son superiores a 92 pero inferiores de 95. En cambio, todos los beneficios con lambda igual a 0.1, dan 95.

## Conclusiones:

Para determinar los parámetros que dan mejor resultado para el Simulated Annealing, nos hemos fijado en aquellos con los que hemos obtenido un 95 de beneficio. A partir de aquí, hemos decidido escoger los parámetros con los que la ejecución ha sido más rápida. Con lambda 0.01 y k igual a 5 el tiempo medio de ejecución ha sido de 2,585 segundos (la ejecución más rápida), por lo tanto, estos han sido los hiper parámetros escogidos.

Además, podemos afirmar nuestra hipótesis nula, ya que hemos podido observar que algunos parámetros hacen que el algoritmo consiga no estancarse en los máximos locales, en cambio hay otros que no lo consiguen.

## 2.4. Experimento 4

### Descripción:

Este experimento consiste en estudiar cómo evoluciona el tiempo de ejecución para hallar la solución en función del número de estaciones, furgonetas y bicicletas.

Como en los apartados anteriores, usaremos la función heurística 1 y el algoritmo Hill Climbing, y usaremos los operadores y la función generadora del estado inicial elegidos en el experimento 1 y 2 respectivamente. Para realizar el análisis que nos pide el experimento, iremos cambiando el número de estaciones, y a partir de este número se cambiarán el número de bicicletas y de furgonetas según las proporciones establecidas: 50 bicicletas por cada estación y 1 furgoneta cada 5 estaciones.

Nuestra hipótesis nula es que a medida que aumentamos las estaciones, furgonetas y bicicletas, aumenta el tiempo de ejecución de manera lineal.

### Ejecución y resultados:

Este experimento nos pide ver la evolución del tiempo de ejecución modificando el número de estaciones, furgonetas y bicicletas, asumiendo las proporciones adecuadas en cada caso. Para ello, hemos realizado experimentos con 25, 50, 75, 100, 125 y 150 estaciones. Además, para cada número de estaciones hemos realizado 10 repeticiones de cada experimento, de esta manera hemos podido observar si existe una variación del tiempo por cada repetición y la tardanza de cada experimento.

Hemos considerado que para este experimento necesitamos mirar el tiempo de ejecución medio por cada repetición no por cada réplica ya que queremos ver cómo evoluciona el tiempo en 1 escenario en concreto.

| Nº ESTACIONES | 25 | 50 | 75 | 100 | 125 | 150 |
|---|---|---|---|---|---|---|
| media tiempo (s) | 0.54 | 5.94 | 38.41 | 129.11 | 289.85 | 833.86 |

**Figura 9:** Tabla con el tiempo medio para cada cantidad de estaciones

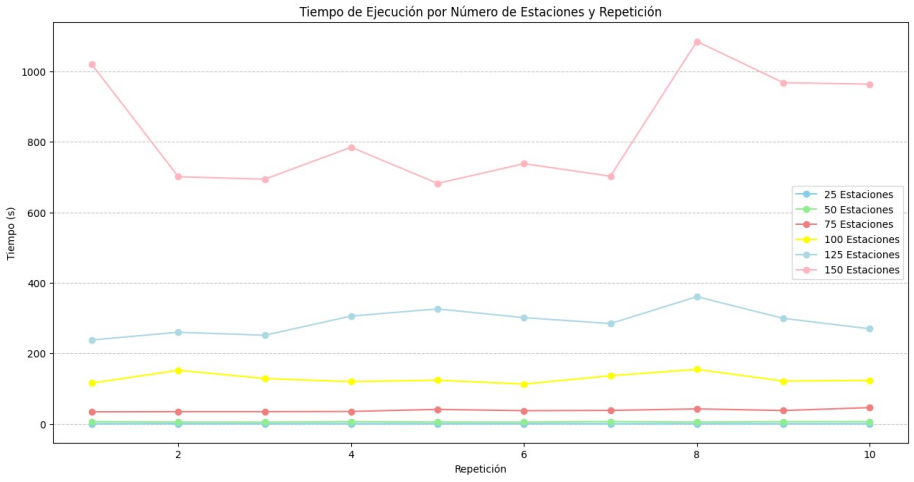


**Figura 10:** Gráfico de los tiempos de ejecución por cada cantidad de estaciones

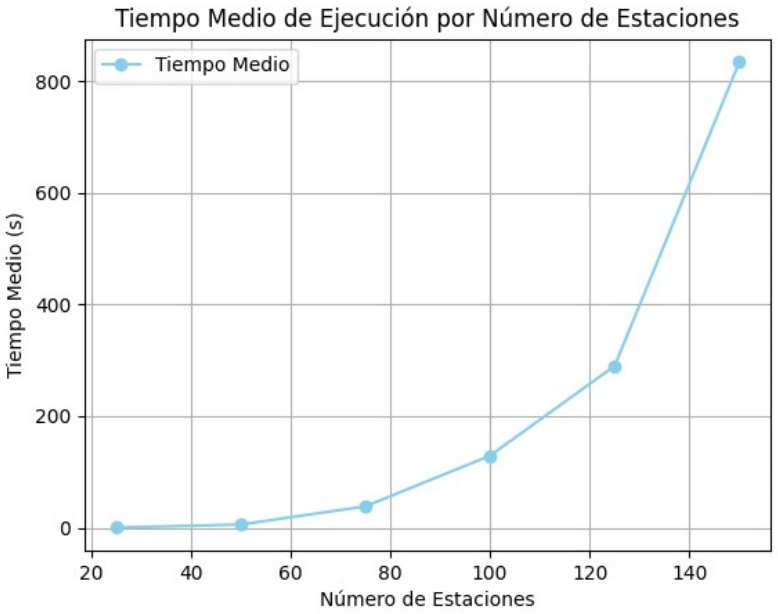


**Figura 11:** Gráfico del tiempo medio de ejecución por cada cantidad de estaciones

Como podemos observar en los gráficos, el aumento del número de estaciones, hace que el tiempo de ejecución de cada experimento también crezca. Podemos observar una tendencia similar al crecimiento exponencial.

## Conclusiones:

Definitivamente, después de observar los resultados de este experimento, podemos rechazar la hipótesis nula y concluir que el aumento de estaciones, furgonetas y bicicletas hace que el tiempo de ejecución tenga una tendencia que sigue un crecimiento exponencial. Lo que sugiere que la complejidad del problema aumenta con la cantidad de estaciones, furgonetas y bicicletas.

## 2.5. Experimento 5

### Descripción:

El objetivo de este experimento es comparar los resultados obtenidos de nuestro experimento para los algoritmos Hill Climbing y Simulated Annealing. Lo haremos con la generación de solución inicial número 2, y compararemos los algoritmos para nuestras dos funciones heurísticas (donde una sólo maximiza las ganancias, y la otra, además, minimiza los kilómetros recorridos por las furgonetas). Se da el escenario del primer apartado.

### Ejecución y resultados:

Para realizar esta comparación hemos escogido los operadores, la solución inicial y los parámetros del Simulated Annealing que ya hemos determinado con los experimentos anteriores. Para ello, hemos realizado 10 experimentos con semillas diferentes y 5 repeticiones por cada réplica y haremos la media de lo que queramos comparar (tiempos de ejecución, beneficios obtenidos o distancias recorridas). Así podremos observar el efecto del cambio de semilla en los dos algoritmos y podremos comparar cada uno cuando están afectados por diferentes escenarios.

Nuestra hipótesis nula es que según los experimentos realizados, el algoritmo Hill Climbing será más rápido que el Simulated Annealing en los tiempos de ejecución, aunque en las ganancias obtenidas y la distancia total recorrida puede que no haya una gran diferencia entre los dos algoritmos.

**HEURÍSTICO 1:**

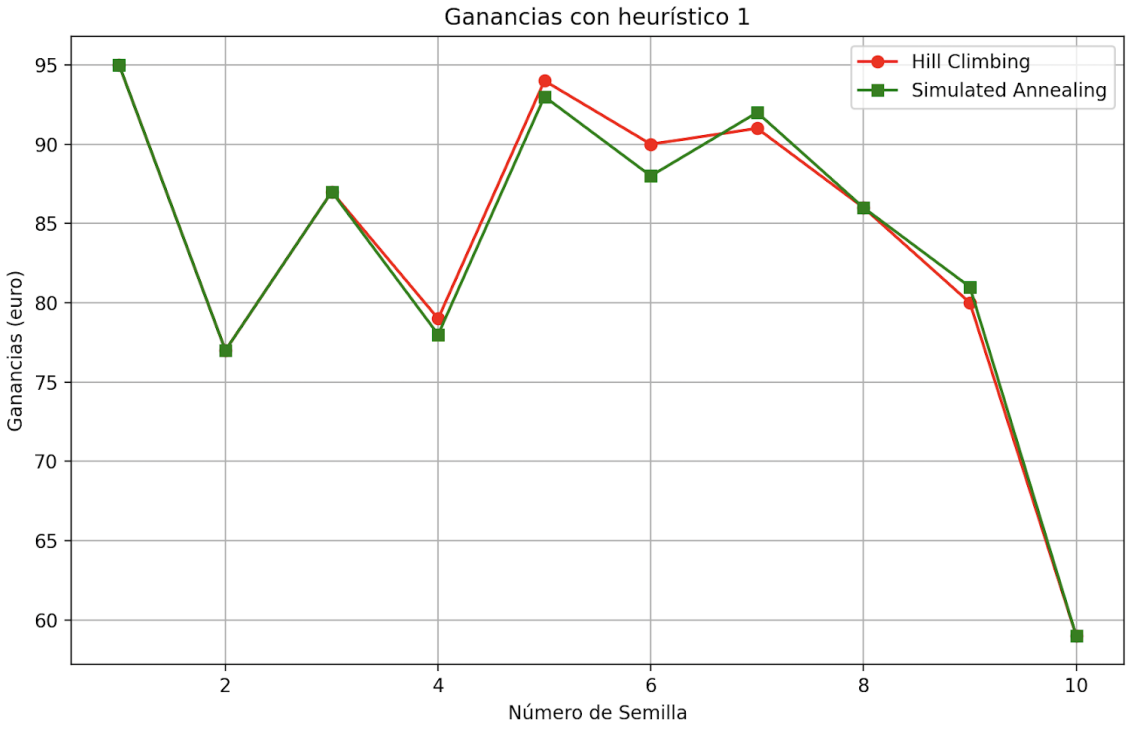


**Figura 12:** Gráfico comparando las ganancias medias de los dos algoritmos con el heurístico 1

En este primer gráfico estamos comparando los beneficios obtenidos por cada uno de los algoritmos. Como podemos ver, en la mayoría de ocasiones los dos algoritmos llegan a los mismos resultados. Aunque hay 3 casos donde el Hill Climbing supera los beneficios del Simulated Annealing y en otros dos casos es lo contrario. No podríamos decir que el algoritmo Hill Climbing supera de una manera significativa al algoritmo Simulated Annealing, aunque este llegue a mejores soluciones (por media) ya que esta diferencia no es para nada notable, supera los beneficios obtenidos por un valor muy pequeño.

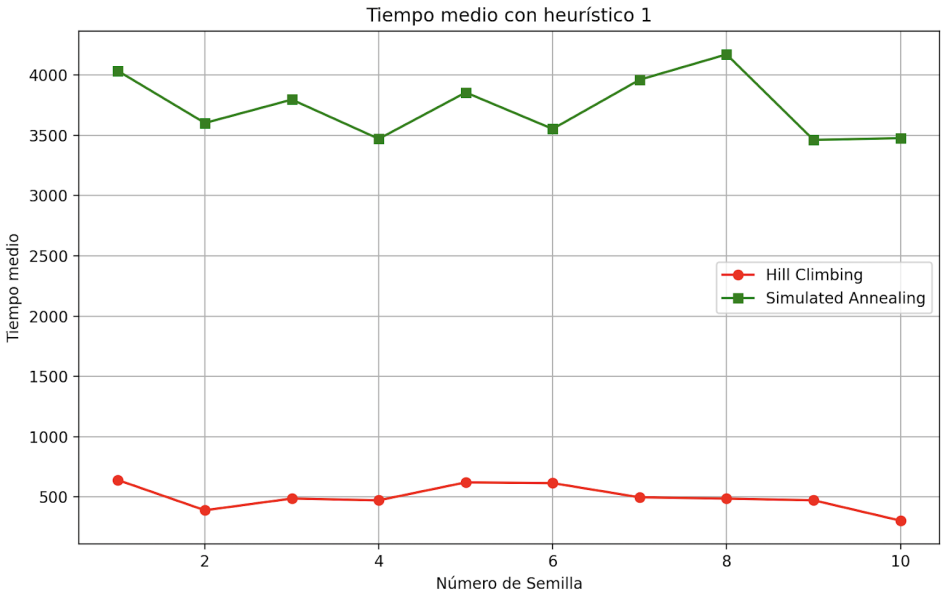


**Figura 13:** Gráfico comparando los tiempos medios de los dos algoritmos con el heurístico 1

Como podemos observar en este gráfico, los tiempos medios de ejecución de cada algoritmo son bastante diferentes. Una de las diferencias es que en el algoritmo Hill Climbing no hay una gran variación en los tiempos medios de diferentes semillas, en cambio, en el Simulated Annealing sí que podemos observar tiempos medios un poco más diversos. Pero sobre todo, lo más destacado de este gráfico es la diferencia de velocidades de los dos algoritmos, como podemos observar, el algoritmo Hill Climbing es mucho más rápido que el Simulated Annealing en todas las diferentes semillas. Esto es debido a que el Simulated Annealing también explora otras soluciones no tan buenas y de esta manera no quedarse estancado en un máximo local. Peró como hemos visto en la gráfica anterior, los dos algoritmos llegan a las mismas ganancias.

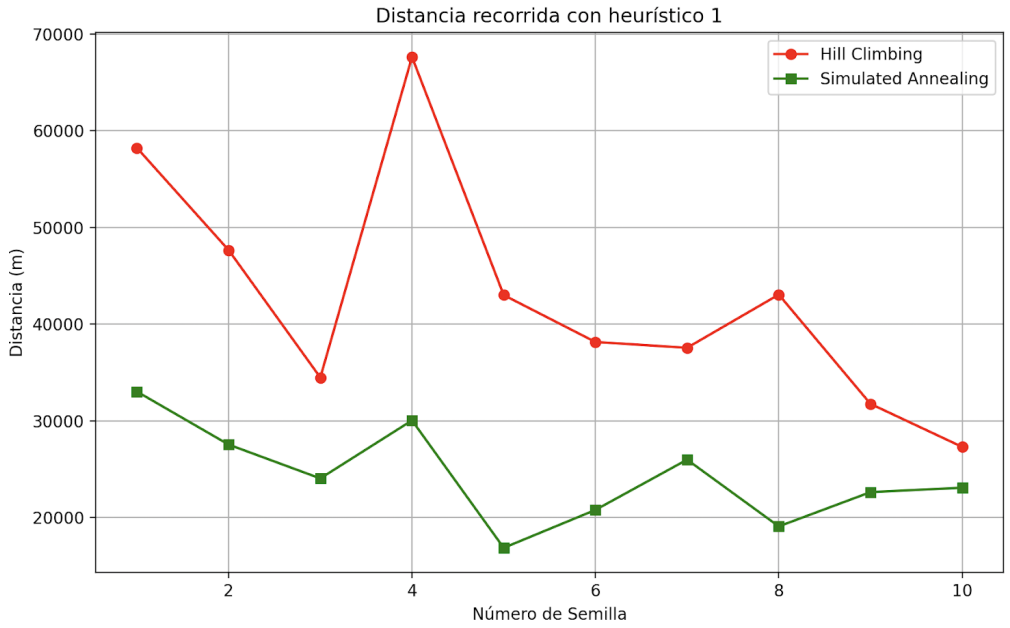


**Figura 14:** Gráfico comparando la distancia recorrida de los dos algoritmos con el heurístico 1

Como podemos observar, el algoritmo Hill Climbing en el primer heurístico hace un recorrido mucho mayor que el Simulated Annealing. Creemos que puede ser debido a que el Simulated Annealing al hacer muchos más pasos, modifica más la ruta de la solución inicial y en cambio el Hill Climbing, como parte de una solución que da bastantes beneficios, como este heurístico no se fija en las distancias recorridas, prácticamente no cambiará las distancias iniciales o las cambiará muy poco aunque estas no sean muy buenas.

**HEURÍSTICO 2:**

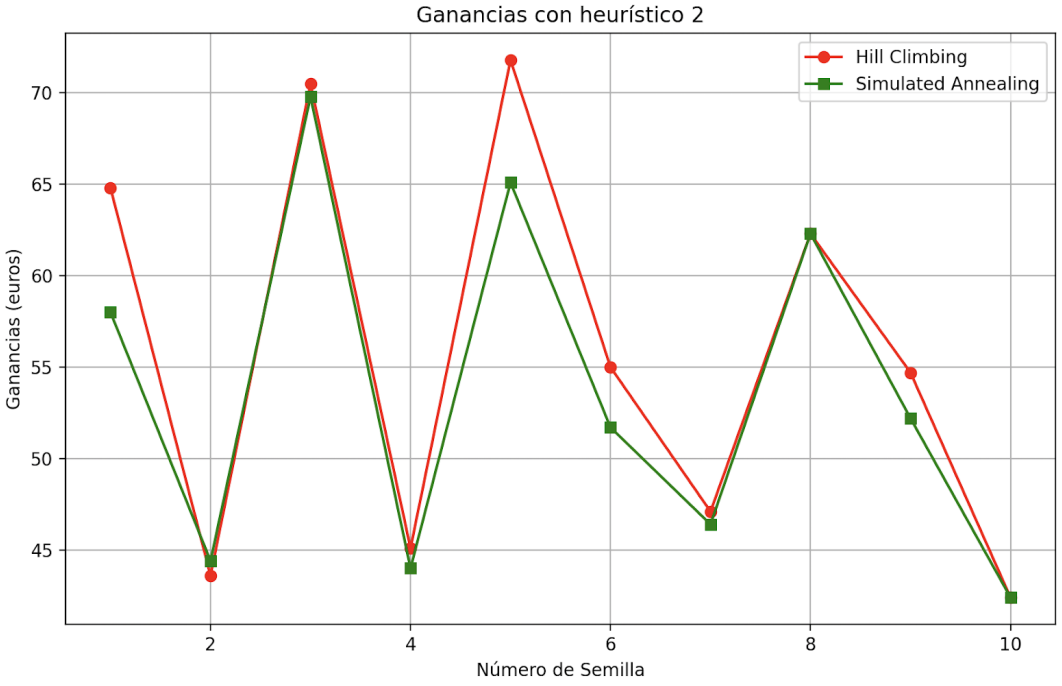


**Figura 15:** Gráfico comparando las ganancias medias de los dos algoritmos con el heurístico 2

En este gráfico estamos comparando la media de los beneficios obtenidos por cada réplica en el segundo heurístico. Como podemos observar, a diferencia del primer heurístico, si que hay una diferencia entre los beneficios obtenidos de los dos algoritmos en algunas réplicas. Podemos ver que el algoritmo Hill Climbing llega a mejores soluciones que el Simulated Annealing con las semillas 1,5, 6 y 9, aunque esto no quiere decir que el Hill Climbing llegue a la solución más òptima (seguramente se ha quedado en un máximo local).

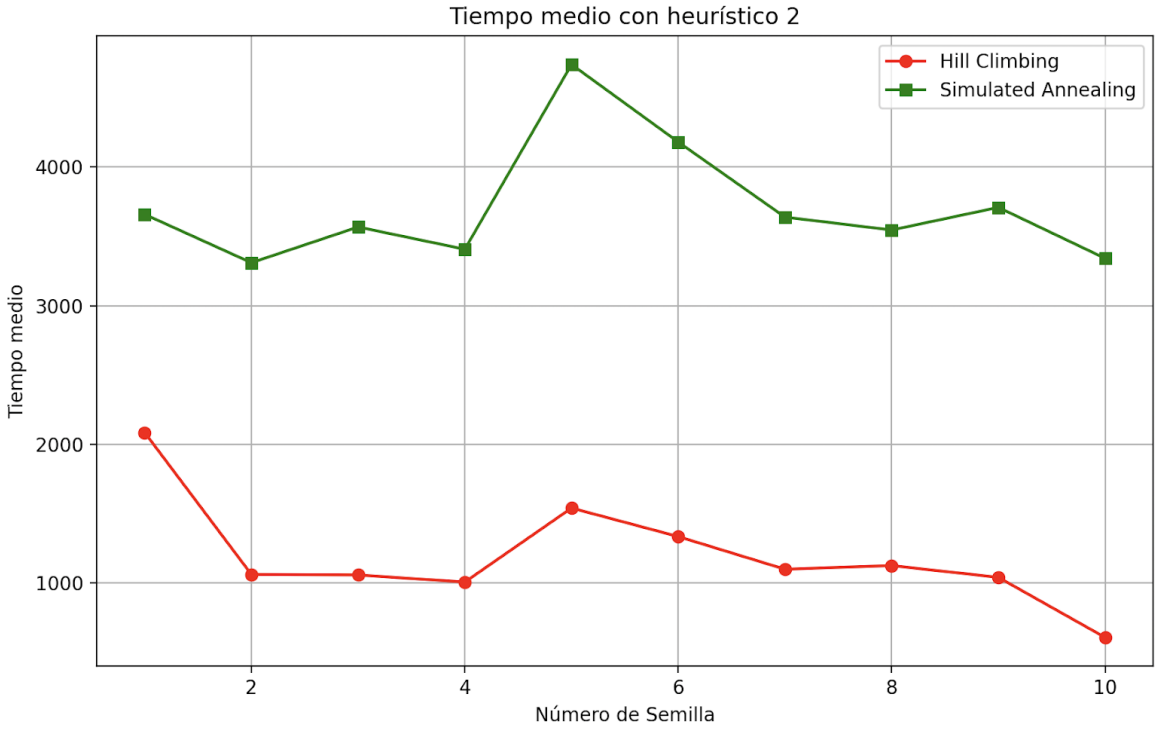


**Figura 16:** Gráfico comparando los tiempos medios de los dos algoritmos con el heurístico 2

Si observamos el tiempo medio que tardan con el segundo heurístico, vemos que tienen la misma tendencia que en el caso del heurístico 1. Aún así, vemos que hay un poco menos de diferencia entre los tiempos entre algoritmos ya que ahora, el Hill Climbing al tener en cuenta las distancias, tarda un poco más en encontrar la mejor solución.

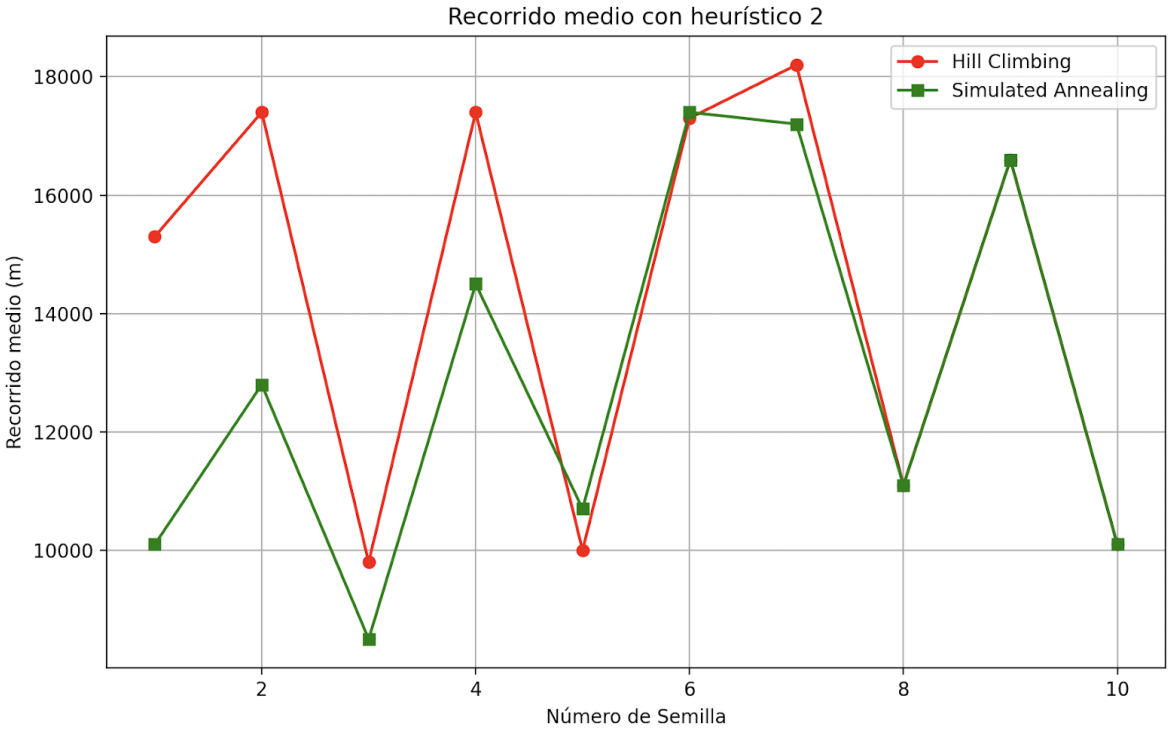


**Figura 17:** Gráfico comparando la distancia recorrida de los dos algoritmos con el heurístico 2

Ahora veamos las diferencias entre las distancias en el heurístico 2. En este caso vemos que en la mayoría de las semillas, el Hill Climbing recorre más kilómetros que el el Simulated Annealing. Como el objetivo con el heurístico dos es reducir los kilómetros recorridos, en este caso el Simulated Annealing está haciendo un mejor trabajo. Aun así, suponemos que como el Hill Climbing tiene ganancias mejores, eso es que está haciendo la repartición de bicis mejor aunque recorre más kilómetros.

## Conclusiones:

Nuestra hipótesis nula decía que el Hill Climbing superaría a Simulated Annealing en el tiempo de ejecución. Hemos podido observar que así ha sido ya que Hill Climbing no explora tantos estados y consecuentemente va más rápido. Nuestra hipótesis también decía que las ganancias y las distancias que se iban a obtener no tendrían diferencias significativas entre algoritmos. Podemos refutar esta hipótesis con el heurístico dos ya que se observa que Hill Climbing da mejores resultados que Simulated Annealing.

En el caso del heurístico 1, donde solo se maximizan las ganancias, los dos algoritmos pueden llegar a una buena solución ya que la generación de solución inicial ya es muy buena y no debe explorar muchos estados para llegar a un resultado muy bueno, por lo tanto a parte de encontrar una buena solución, la encuentra rápido. Aunque el Simulated Annealing si que explora más soluciones que en el Hill Climbing, este no llega a mejores resultados.

En cambio, para el heurístico dos, donde se tienen en cuenta las distancias, la generación de solución inicial no es tan buena, y para llegar a unas buenas ganancias, se tiene que explorar más estados y aquí se ponen a prueba las capacidades del algoritmo. El Simulated Annealing nos da peores resultados, tanto en tiempo como en ganancias.

## 2.6. Experimento 6

### Descripción:

Este experimento consiste en determinar el número de furgonetas necesario para obtener la mejor solución. Para ello empezaremos con 5 furgonetas e iremos aumentándolas de 5 en 5, hasta que no haya una mejora significativa.

Las condiciones para realizar el experimento serán las siguientes:

- Usaremos 25 estaciones y 1250 bicicletas
- Función heurística 1: maximizar las ganancias obtenidas con la distribución de bicicletas
- Algoritmo Hill Climbing
- Operadores y función generadora del estado inicial elegidos en los experimentos 1 y 2.

Ejecutaremos 10 experimentos, cada uno con una semilla distinta para la generación del espacio del problema, con el objetivo de poder comparar diferentes escenarios, y no siempre partir de la misma base.

Nuestra hipótesis nula es que las ganancias aumentarán según aumentemos el número de furgonetas, ya que podremos desplazar más bicis para ajustarse a la demanda.

### Ejecución y resultados:

En la siguiente tabla podemos ver recogidos los resultados después de haber realizado los 10 experimentos. Aparecen las medias del tiempo de ejecución en milisegundos y de las ganancias (en euros) por cada cantidad de furgonetas elegida.

| nº de furgonetas → | 5 | 10 | 15 | 20 | 25 | 30 |
|---|---|---|---|---|---|---|
| ganancias (€) | 84 | 115.6 | 125 | 150.9 | 189.6 | 189.6 |
| tiempo (ms) | 438.4 | 1152.7 | 1457.9 | 1463.0 | 594.2 | 664.9 |

**Figura 18:** Tabla con los tiempos medios y las ganancias para cada cantidad de furgonetas

Podemos ver como las ganancias dejan de subir a partir de las 25 furgonetas. También vemos cómo se produce una disminución brusca del tiempo de ejecución en el mismo punto. Por este motivo, hemos decidido profundizar en el intervalo de 20 a 25 furgonetas, estudiando los resultados para 21, 22, 23 y 24 furgonetas.

| nº de furgonetas → | 5 | 10 | 15 | 20 | 21 | 22 | 23 | 24 | 25 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|
| ganancias (€) | 84 | 115.6 | 125 | 150.9 | 159.3 | 161.7 | 168.3 | 182.5 | 189.6 | 189.6 |
| tiempo (ms) | 438.4 | 1152.7 | 1457.9 | 1463.0 | 1276.8 | 985.3 | 750.4 | 571.0 | 594.2 | 664.9 |

**Figura 19:** Tabla ampliada con los tiempos medios y las ganancias para cada cantidad de furgonetas

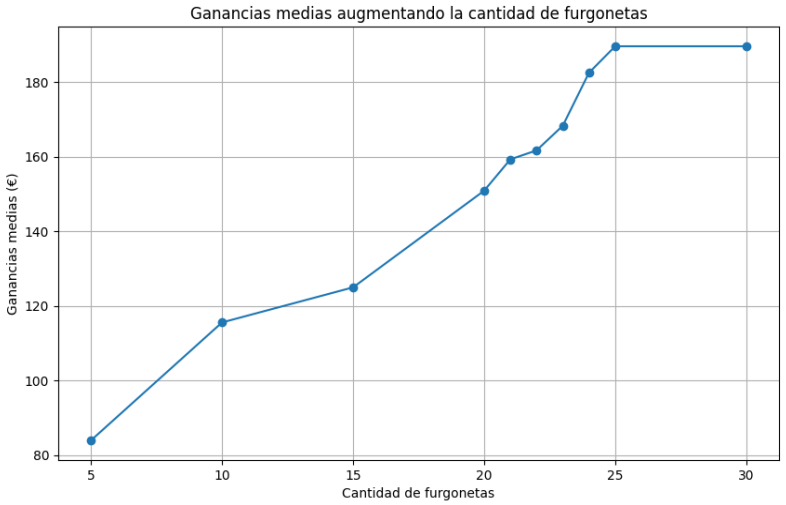


**Figura 20:** Gráfico de las ganancias medias para cada cantidad de furgonetas

Podemos observar como las ganancias aumentan con cada furgoneta que añadimos, pero cuándo el número de furgonetas llega a 25 las ganancias quedan estancadas. Esto sucede porque si vamos aumentando el número de furgonetas llega un momento en el que las bicicletas quedan correctamente distribuidas y ya no queda ninguna estación dónde falten bicicletas para suplir la demanda. Debemos recordar que en este experimento hemos usado la función heurística para maximizar las ganancias obtenidas con la distribución de bicicletas, y por este motivo, cuando ya se han distribuido correctamente todas las bicicletas, las ganancias quedan estancadas.

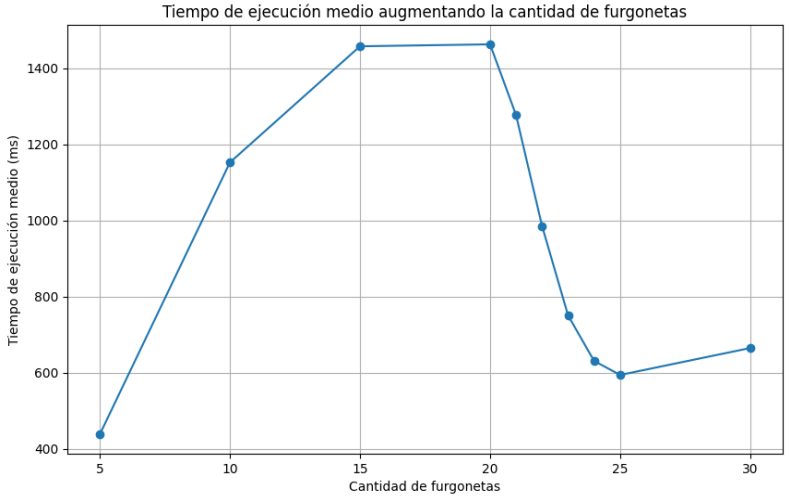


**Figura 21:** Gráfico del tiempo de ejecución medio para cada cantidad de furgonetas

En esta gráfica del tiempo de ejecución podemos ver como este va incrementando progresivamente hasta las 20 furgonetas, y después disminuye bruscamente. Esto es debido a que al principio, como más furgonetas tengamos más aumentará el factor de ramificación de los operadores y más sucesores tendrá que recorrer el Hill Climbing. Pero cuándo se llega a las 20 furgonetas, con las asignaciones de furgonetas en estaciones que se producen en la generación de la solución inicial, ya quedan la mayoría de demandas suplidas, y no hace falta recorrer tan profundamente el espacio de estados para conseguir

una solución óptima. En las 25 furgonetas la solución inicial ya es directamente la óptima, y a partir de allí, el aumento del tiempo de ejecución es debido a el tiempo que se tarda en asignar todas las furgonetas inicialmente.

## Conclusiones:

Con los resultados obtenidos podemos concluir que el número de furgonetas necesario para obtener la mejor solución es 25, ya que de las posibilidades donde se consiguen las máximas ganancias, es la que tiene un tiempo de ejecución más bajo.

No refutamos nuestra hipótesis nula ya que efectivamente hemos observado mejoras en las ganancias al ir aumentando el número de bicicletas. Aún así, como se ha comentado antes, es importante destacar que llega un punto que si tienes el mismo número de furgonetas que de estaciones, la demanda se cubre ya y si se aumentan aún más las furgonetas ya no se verán más mejoras.

## 2.7. Experimento Especial

### Descripción:

Este experimento requiere obtener el resultado del beneficio total obtenido, la longitud total del recorrido de las furgonetas y cuánto tiempo (aproximadamente) se tarda en hallar la solución en milisegundos.

Las condiciones del experimento deben ser:

- Escenario con 25 estaciones, 1250 bicicletas y 5 furgonetas
- La semilla de la generación del espacio del problema (colocación de las 25 estaciones y las 1250 bicicletas en la ciudad y generación de las demandas) será 42
- Función heurística 1: maximiza las ganancias obtenidas por el traslado de las bicicletas y no tiene en cuenta los kilómetros recorridos.
- Algoritmo Hill Climbing.
- Operadores y función generadora del estado inicial elegidos en los experimentos 1 y 2.

### Ejecución y resultados:

Ejecutamos el experimento 10 veces, y en cada repetición obtenemos un tiempo de ejecución diferente, pero parecido. En el siguiente gráfico lo podemos ver representado.

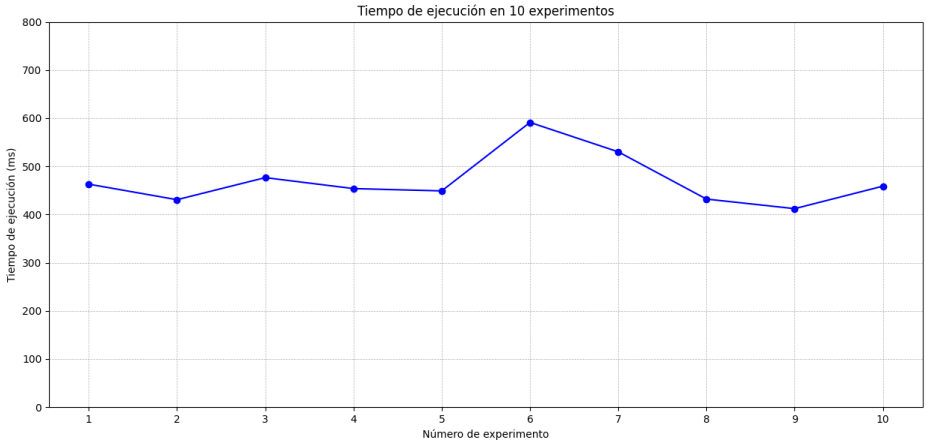


**Figura 22:** Gráfico de los tiempos de ejecución para 10 experimentos

A continuación, vemos los resultados promedio del experimento:

| ganancias obtenidas (€) | distancia total (m) | tiempo de ejecución (ms) |
|---|---|---|
| 95 | 62500 | 574.7 |

**Figura 23:** Tabla con las medias de ganancias, distancia y tiempo

## Conclusiones:

Las ganancias obtenidas en este experimento son de 95€, la distancia total recorrida por las furgonetas es de 62,5 km y el tiempo de ejecución es aproximadamente 574,7 ms. En el gráfico podemos observar cómo el tiempo de ejecución varía en las 10 ejecuciones entre valores de 400 y 600 milisegundos. Los otros parámetros que queremos calcular (las ganancias y la distancia total) no varían con cada ejecución, ya que siempre utilizamos la misma semilla. Podemos ver que la distancia total es bastante alta, ya que al utilizar la heurística de maximizar las ganancias obtenidas por los traslados de las bicicletas no tenemos en cuenta los kilómetros recorridos por las furgonetas y, por lo tanto, no se busca reducir este número.

Anteriormente habíamos realizado este experimento con la función *generar_estado_inicial3* porque era la que habíamos elegido en el experimento 2, ya que aún no habíamos creado la función que se ha seleccionado finalmente (*generar_estado_inicial2*). Por este motivo, cuándo este experimento fue enviado a los profesores el 22 de octubre, el tiempo de ejecución y la distancia total eran inferiores.

# 3. Conclusiones

Para finalizar la documentación de esta práctica, hemos realizado unas conclusiones que sintetizan la valoración, positiva o negativa, del trabajo realizado.

Para empezar, queremos evaluar nuestra implementación del programa y hacer una autocrítica sobre las decisiones tomadas. Al inicio tuvimos dudas sobre las estructuras de datos más adecuadas para representar los elementos del problema. Pero en seguida nos pareció buena idea implementar la clase Furgoneta y almacenar instancias de esta clase en una lista, dado que para nosotras era una representación del estado muy entendible y sencilla de manejar. Esta estructura de datos no ha presentado ningún problema significante para la eficiencia temporal, pero aún así, pensamos que otra implementación del estado quizás podría haber sido aún más eficaz.

En relación a las funciones generadoras de la solución inicial, creemos que las tres que hemos implementado són eficaces. Sin embargo, después de ver su influencia en la solución final, nos hemos dado cuenta que la función más buena, *generar_estado_inicial3*, no era demasiado útil para realizar la práctica porque al ser una solución tan buena, apenas se encontraban soluciones mejores. Es decir, el poco recorrido que se hace por el espacio de soluciones con esta solución inicial, hace que no la hayamos considerado una buena estrategia de generación del estado inicial. Las otras dos funciones sí que las hemos considerado útiles para la práctica.

Respecto a los operadores, creemos que hemos hecho una buena implementación de los 6 operadores creados, aunque los algoritmos no los utilicen todos para encontrar la mejor solución.

Con algunos experimentos nos hemos dado cuenta que harían falta más operadores para llegar a la solución óptima. Por ejemplo, no tenemos ningún operador que permita cambiar la ubicación de la estación de origen de las furgonetas, esto hace que no se pueda llegar a la mejor solución ya que como la primera solución inicial no asigna a las furgonetas las mejores estaciones de origen posibles, nunca se podrá llegar a las máximas ganancias. En cambio, con las otras dos soluciones iniciales, sí que se pueden obtener las ganancias máximas, dado que no les afecta la falta de operadores para reasignar la estación de origen de las furgonetas.

Además, puede que tampoco llegue a la solución óptima en el segundo heurístico debido a la eliminación de operadores del primer experimento, ya que estos fueron escogidos en función del primer heurístico que no tiene en cuenta la distancia.

En cuanto a los dos heurísticos que hemos implementado en nuestra práctica, creemos que aunque al principio nos costó entender su función, conseguimos implementar unas buenas funciones heurísticas a tiempo. Cumplen con los dos objetivos de la práctica: la maximización de las ganancias (a partir del cálculo de los euros ganados y perdidos según si acercamos o alejamos furgonetas a su destino), y la minimización de los kilómetros (restando los euros que tenemos que pagar por cada kilómetro que realiza cada furgoneta).

Se obtiene una buena representación de si una solución es buena o mala y por tanto cumple con las condiciones necesarias para ser un buen heurístico. Aunque otras partes de nuestra implementación, visto ahora con perspectiva, las hubiésemos implementado diferente, creemos que las funciones heurísticas no necesitan ningún cambio.

Con referencia a los algoritmos, los experimentos realizados nos han permitido saber que en nuestra implementación del programa, el algoritmo Hill Climbing, en general, supera al algoritmo Simulated Annealing. Sobre todo, esta diferencia se observa en el segundo heurístico. Ya que el Hill Climbing es mejor en términos de tiempo y ganancias. Como hemos podido ver anteriormente en los experimentos, el Simulated Annealing es mucho más lento. Estos resultados respecto los tiempos de ejecución eran los que nos esperábamos ya que como bien sabemos, el Simulated Annealing explora muchas más soluciones que el Hill Climbing, aunque los resultados de los beneficios nos han sorprendido un poco ya que pensábamos que el Simulated Annealing sería capaz de encontrar mejores soluciones.

En cambio, respecto a la distancia, el Simulated Annealing es mejor ya que hace que las furgonetas hagan un menor recorrido, peró igualmente en los beneficios finales sigue ganando el Hill Climbing.

Creemos que estos resultados pueden ser debidos a la creación de los operadores ya que los desarrollamos pensando en el algoritmo Hill Climbing y como este podría mejorar la generación inicial y llegar a una mejor solución. Además, cabe destacar que los operadores escogidos para hacer los experimentos fueron los que funcionaban mejor con el Hill Climbing y el primer heurístico.

También puede ser que estos resultados sean debidos a los hiper parámetros escogidos en el tercer experimento, ya que nosotras probamos unos valores en concreto y de entre estos escogimos los mejores, peró el rango de los parámetros es mucho mayor y puede que haya otros que hagan que el algoritmo Simulated Annealing llegue a resultados mejores y más rápido.

Aún así los dos algoritmos llegan a soluciones muy buenas en tiempos bastante reducidos.

En cuanto a la realización de los experimentos, hemos conseguido realizar los 7 experimentos propuestos en la práctica. En nuestro documento main, se puede elegir qué experimento quieres realizar y automáticamente se fijan los parámetros necesarios para realizarlos. Hay una explicación más extendida en el README.txt.

Hemos podido automatizar la mayoría de experimentos aunque algunos se deben ejecutar repetidas veces. El experimento 1 es el único para el que no hemos automatizado la elección de operadores así que para ejecutarlo se deben cambiar el código manualmente.

Además se han hecho manualmente los gráficos para representar los datos. Nos habría gustado haberlo automatizado también, haciendo que el programa guardase los resultados de los experimentos y él mismo a partir de ahí realizase los resultados (los gráficos para el análisis).

Creemos que con estos cambios nuestra realización de experimentos se hubiera hecho más amena.

Esta práctica nos ha permitido comprender y tener una visión mucho más amplia y específica de cómo funcionan los algoritmos de búsqueda local. Hemos ampliado mucho nuestro conocimiento sobre cómo plantear un problema y cómo implementar sus partes básicas (la función heurística, la función de generación inicial, la generación y la aplicación de acciones).

Consideramos que nos hemos organizado de manera bastante eficaz. Hemos conseguido seguir la planificación propuesta, aunque nos hubiera gustado dedicarle más tiempo a la comprensión del

Simulated Annealing. Para conseguir llevar el trabajo al día, conseguimos dividirnos el trabajo eficientemente. Al principio, para plantear el problema y sus partes, sí que dedicamos más tiempo las tres juntas para decidir una manera de implementar el problema. Pero la parte experimental hemos sabido paralelizarla para que cada una realizara dos experimentos sin que ninguna dependiera de las otras. Obviamente nos hemos comunicado de manera extensa para que todas tuviéramos total comprensión del estado de nuestro trabajo.

Como prueba de nuestro aprendizaje continuo, está el experimento especial que realizamos dos semanas antes de la entrega. Lo conseguimos hacer a tiempo y nos ayudó a comprender posibles problemas de nuestra implementación que grácias a eso tuvimos tiempo a rectificar.

Queremos mencionar que hemos visto la importancia de plantear bien el problema y tener muy bien estudiados todos los elementos del problema, así como el espacio de soluciones. En nuestro caso, al principio había conceptos que no habían quedado claros e hicieron que nos fuera más difícil mantener la práctica al día. En un futuro, antes de empezar con la práctica, dedicaremos más tiempo a entender los conceptos teóricos de la práctica antes de empezar con nuestra implementación. Aún así, hemos podido hacer la práctica siguiendo la planificación semanal que se nos proporcionaba.