Informe de la Práctica 1

Roger Baiges Trilla, Pau Prat Moreno i Cai Selvas Sala

30 de octubre de 2023

Universidad Politécnica de Cataluña
Grado en Inteligencia Artificial
Algoritmos Básicos para la Inteligencia Artificial

Abstract

Este documento corresponde al informe de la primera práctica la asignatura Algoritmos Básicos de la Inteligencia Artificial del Grado en Inteligencia Artificial de la Universidad Politécnica de Cataluña (UPC).

A continuación se comenta el planteamiento realizado para resolver el problema, los pasos seguidos, la estructura e implementación del programa en Python, los resultados obtenidos para el conjunto de ejercicios requeridos y las conclusiones de la práctica.

Contents

1 Introducción **5**
1.1 Descripción del problema . . . 5
1.2 Descripción de los elementos del estado del problema . . . 5
1.3 Por qué es un problema de búsqueda local . . . 6

2 Preguntas iniciales **7**
2.1 ¿Qué elementos intervienen en el problema? . . . 7
2.2 ¿Cuál es el espacio de búsqueda? . . . 7
2.3 ¿Qué tamaño tiene el espacio de búsqueda? . . . 7
2.4 ¿Qué es un estado inicial? . . . 7
2.5 ¿Qué condiciones cumple un estado final? . . . 8
2.6 ¿Qué operadores permiten modificar los estados? . . . 8
2.7 ¿Qué factor de ramificación tienen los operadores de cambio de estado? . . . 8

3 Estructura e implementación del programa **10**
3.1 Interpretación y análisis detallado de las características del problema . . . 10
3.2 Estructura general . . . 11
3.2.1 *main.py* . . . 11
3.2.2 *state_bicing.py* . . . 11
3.2.3 *generate_initial_state_bicing.py* . . . 13
3.2.4 *parameters_bicing.py* . . . 13
3.2.5 *furgoneta_bicing.py* . . . 13
3.2.6 *operators_bicing.py* . . . 14
3.2.7 *functions_bicing.py* . . . 14
3.2.8 *problem_bicing.py* . . . 14
3.3 Estado . . . 15
3.3.1 Variables del estado . . . 15
3.3.2 Restricciones . . . 15
3.4 Operadores . . . 16
3.4.1 Descripción de cada operador . . . 16
3.4.2 Condiciones de aplicabilidad y sus efectos . . . 16
3.4.3 Análisis del factor de ramificación de los operadores . . . 17
3.4.4 Explicación de la elección de los operadores . . . 18
3.5 Generación de la solución inicial . . . 18
3.6 Funciones heurísticas . . . 19
3.6.1 Análisis de los factores que intervienen en la heurística . . . 19
3.6.2 Justificación de las funciones heurísticas escogidas . . . 20
3.6.3 Explicación de los efectos de las funciones heurísticas en la búsqueda . . . 20
3.6.4 Ponderaciones que aparecen entre los elementos de las heurísticas . . . 20

4 Experimentos **21**
4.1 Primer experimento . . . 21
4.2 Segundo experimento . . . 23
4.3 Tercer experimento . . . 24
4.4 Cuarto experimento . . . 27
4.5 Quinto experimento . . . 29

4.6 Sexto experimento 34
4.7 Experimento especial 35

5 Hill Climbing vs Simulated Annealing **36**

6 Conclusiones **37**
6.1 Valoración del aprendizaje adquirido 37

1 Introducción

1.1 Descripción del problema

El problema empieza con una distribución de E estaciones situadas aleatoriamente en los cruces de una cuadrícula de $10km \times 10km$ que representa una ciudad con calles de $100m$ de longitud. También hay un número de B bicicletas que son distribuidas aleatoriamente entre el conjunto de estaciones y un número de F furgonetas que representa la cantidad de furgonetas que se disponen para realizar cargas/descargas de bicicletas.

Una vez determinada la distribución de estaciones y bicicletas, se generan aleatoriamente unas demandas y predicciones de bicicletas para cada estación.

El problema trata sobre conseguir satisfacer las demandas de bicicletas mediante movimientos de furgonetas cargadas de bicicletas (máximo 30 por furgoneta). Para cada bicicleta transportada que contribuya a satisfacer la demanda de una estación, se gana 1€; mientras que si esta bicicleta contribuye a alejarse del objetivo de demanda, se pierde 1€. Además, el transporte de bicicletas tiene un coste variable en función de la cantidad de bicicletas transportadas en cada trayecto y de los kilómetros realizados. No obstante, este coste solo se tiene en cuenta con una de las funciones heurísticas que se requieren en el enunciado de esta práctica.

Así pues, el objetivo del problema es maximizar el beneficio obtenido mediante modificaciones de la solución inicial generada.

Ciertos aspectos del problema pueden estar sujetos a interpretación, por lo que en el apartado 3.1 se exponen las interpretaciones, asunciones y características que se han tenido en cuenta para realizar esta práctica.

1.2 Descripción de los elementos del estado del problema

El estado del problema cuenta con los siguientes elementos:

- Operadores activos: Dependiendo de qué operadores estén activos y cuales no, el estado del problema será uno u otro, ya que los operadores nos permites movernos entre soluciones vecinas y, sin éstos, nunca podríamos modificar un estado actual. Éstos los explicamos más detalladamente en el apartado 3.4.
- Estaciones: Cada estación cuenta con los siguientes atributos: Id, coordenadas, número de bicicletas que habrá en la siguiente hora, número de bicicletas que no se moverán en la hora actual. Estas siempre se representan mediante la lista de diccionarios llamada *info_estaciones*, donde aparece el índice de la estación en cuestión, la diferencia de bicicletas y la disponibilidad (es decir, cuantas bicicletas no se usarán).
- Furgonetas: Estas tienen una estación de carga y dos de descarga (aunque pueden ser las mismas), también disponen del número de bicicletas a cargar. Por lo tanto, cada furgoneta cuenta con los siguientes atributos: Id de la furgoneta, Id de la estación de carga, destino 1 y destino 2, bicicletas cargadas, bicicletas descargadas en el primer destino, descargadas en el segundo destino y el número de bicicletas que se reducirán (relacionado con el operador *ReducirNumeroBicicletasCarga*, implementado en la función *__asingnar_bicicletas_carga_descarga*).

- Métodos para calcular el beneficio: Estos métodos son llamados cada vez que se llama a la función heurística para calcular el balance total. Para hacerlo calculan las distancias, el coste de transporte y las ganancias de dejar bicicletas en estaciones necesitadas.
- Métodos para la aplicación de acciones: Las funciones para aplicar las acciones creados por los operadores se encuentran también en el mismo estado, estas generalmente solo cambian la información de cada furgoneta y es el propio estado el que luego, al calcular el beneficio, hará las operaciones para saber si es un mejor estado o uno peor comparandolo con el que ya estamos.

1.3 Por qué es un problema de búsqueda local

Se trata de un problema de búsqueda local debido a que se empieza directamente desde una solución válida (mediante la estrategia de generación de la solución inicial). Además, a partir de esta solución inicial, se busca maximizar el valor de la función heurística (maximizar beneficios y minimizar costes) mediante modificaciones de los estados que permiten moverse a soluciones vecinas.

No hay un estado definido como la solución óptima en el que el algoritmo deba detenerse, sino que es el propio algoritmo el que se detiene cuando no puede mejorar más la solución (pudiendo tratarse de un máximo local) o cuando ha superado un límite de pasos.

2 Preguntas iniciales

A continuación se responden las preguntas iniciales referentes al problema a resolver y planteadas en el enunciado de la práctica. Estas preguntas son esenciales para entender las bases del problema y poder resolverlo adecuadamente.

2.1 ¿Qué elementos intervienen en el problema?

En este problema intervienen los siguientes elementos:

- Estaciones: Entendemos a estas como unas coordenadas y un número de bicicletas no usadas, bicicletas que se necesitan la siguiente hora y las bicicletas previstas en esa estación en la siguiente hora sin tener en cuenta el posible movimiento de las furgonetas.
- Furgonetas: Entendiendo a estas como un elemento que se sitúan en una estación para cargar bicicletas, carga un número específico de bicicletas (si es posible) y las reparte en como máximo dos estaciones.
- Bicicletas: Estos objetos son movidos entre estaciones gracias a las furgonetas.

2.2 ¿Cuál es el espacio de búsqueda?

En el espacio de búsqueda es influenciada por la estación de carga de las furgonetas, el número de bicis que se cargan en esa estación y las estaciones de destino. En nuestro espacio de búsqueda no influyen otros elementos como las bicicletas que se descargan en cada estación ya que siempre descargará **todas** las que pueda en la primera estación y las restantes en la segunda. Esto ya ha sido demostrado en la introducción ya que si no lo hiciéramos así perderíamos dinero por el transporte. Además, el número de bicicletas cargadas tampoco pasa por todos los posibles valores *(de 0 a 30 bicis por furgoneta)* sino que se calculan las bicis posibles que podemos cargar y también la suma de las bicis que faltan en las estaciones de descarga y usamos ese valor. Aún así, esto no es suficiente, ya que como también se ha explicado previamente puede influenciar el hecho de que para calcular el coste de la gasolina los valores múltiplos de 10 son más óptimos, ya que al hacer la división entera en la fórmula de coste de la gasolina son los valores que cuestan menos.

2.3 ¿Qué tamaño tiene el espacio de búsqueda?

El tamaño del espacio de búsqueda se puede calcular mediante operaciones con combinatoria. Para simplificar los cálculos vamos a referirnos al número de furgonetas como f y al número de estaciones como e. El tamaño total equivale a todas las combinaciones distintas de estaciones de carga para todas las furgonetas multiplicado por todas las estaciones de descarga posibles y finalmente multiplicarlo por dos ya que en nuestar implementación tan solo se consideran dos estados distintos en cuanto a número de bicis cargadas. La fórmula resultante es la siguiente:

$$\text{Tamaño} = \binom{e}{f} \cdot 4f \cdot (e-1) \tag{1}$$

2.4 ¿Qué es un estado inicial?

En nuestro problema, un estado inicial es una solución del problema generada aleatoriamente, la cual incluye la distribución de las bicicletas en las diferentes estaciones y la ubicación de las furgonetas.

A partir de esta solución inicial (o estado inicial), los diferentes algoritmos irán accediendo a soluciones vecinas con tal de aumentar el coste heurístico (beneficio), ya que al haberse generado aleatoriamente, el estado inicial es muy probable que sea muy poco óptimo. En nuestro caso, el estado inicial tendrá un beneficio significativamente inferior al beneficio obtenido por el estado inicial.
Así pues, nuestro estado inicial tendrá un beneficio (negativo o positivo), y habrá asignado a cada furgoneta una estación de carga y dos de descarga, pudiendo ser estas la misma estación, además de la distribución de carga y descarga de las bicicletas en cada estación.

2.5 ¿Qué condiciones cumple un estado final?

Aunque consideramos que un estado final podría ser cualquier solución por muy malos valores que obtenga, realmente entendemos como estado final como una solución candidata que se considera suficientemente buena según nuestros criterios, en este caso el beneficio obtenido, o que ya no puede mejorarse más según la estructura de búsqueda del algoritmo. En el caso del hill climbing llegaremos a un estado final en el momento en el que entremos en un máximo local, respectivamente, en el simulated annealing llegaremos solo en el momento en que alcance el límite de iteraciones puesto por nosotros previamente o cuando se le termine la temperatura devolviendo así el último estado visitado.

2.6 ¿Qué operadores permiten modificar los estados?

Tenemos un total de siete operadores, todos heredados de la clase *BicingOperator*. Los siete pueden modificar un estado, ya que generan nuevas soluciones vecinas a partir de una solución actual. Inicialmente creamos 3 operadores capaces de acceder a todo nuestro espacio de búsqueda, los cuales son:

- CambiarEstacionCarga
- CambiarEstacionDescarga
- ReducirNumeroBicicletasCarga

Por lo tanto, el resto de operadores solo nos ayudan para poder obtener una solución más fácilmente, aunque teóricamente son prescindibles. Los operadores son los siguientes:

- CambiarOrdenDescarga
- IntercambiarEstacionCarga
- IntercambiarEstacionDescarga
- ReasignarFurgonetaInformado

2.7 ¿Qué factor de ramificación tienen los operadores de cambio de estado?

Para calcular el factor de ramificación de todos los operadores debemos dar una fita superior de cuantos diferentes estados se crearan por cada uno. En nuestro proyecto hay mucos operadores que se puede calcular el número exacto usando combinatoria.

CambiarEstacionCarga: Para calcular el número de estaciones de carga disponibles debemos tener en cuenta que para cada furgoneta podemos asignarla a todas las estaciones menos aquellas donde hay ya una estación de carga. El resultado es el siguiente:

$$\text{Factor de ramificación} = O(f(e-f)) \tag{2}$$

IntercambiarEstacionCarga: Para calcular el número de nuevos estados generados con este operador tenemos que calcular todas las combinaciones de grupos de dos furgonetas que hay. Se consigue con la siguiente fórmula:

$$\text{Factor de ramificación} = O(\binom{f}{2}) \tag{3}$$

CambiarOrdenDescarga: Este operador genera exactamente f nuevos estados ya que por cada furgoneta solo creo uno nuevo.

$$\text{Factor de ramificación} = O(f) \tag{4}$$

CambiarEstacionDescarga: Para calcular el número de nuevos estados que se generan con este operador debemos multiplcar por cada furgoneta todas las estaciones de descarga distintas a si misma por lo tanto *e-1* multiplicado por dos ya que tenemos dos estaciones de descarga.

$$\text{Factor de ramificación} = O(2f(e-1)) \tag{5}$$

IntercambiarEstacionDescarga: El número de estados generados por este operador es todas las combinaciones de dos grupos de furgonetas por 4 ya que se pueden intercambiar la estación de descarga 1 de la primera furgoneta con la estación de descarga 2 de la segunda furgoneta pero también se puede hacer combinaciones de la estación 2 con la 2 y así sucesivamente.

$$\text{Factor de ramificación} = O(4\binom{f}{2}) \tag{6}$$

ReasignarFurgoneta: El factor de ramificación de este operador equivale a f ya que solo asigna una nueva combinación de valores por cada furgoneta.

$$\text{Factor de ramificación} = O(f) \tag{7}$$

ReducirNumeroBicicletas: El factor de ramificación de este operador equivale también a f ya que para cada furgoneta se crea un nuevo estado cambiando el número de bicicletas a escoger al múltiplo de 10 máx menor que el número de bicicletas actual más propero.

$$\text{Factor de ramificación} = O(f) \tag{8}$$

3 Estructura e implementación del programa

En este apartado se describen y justifican las implementaciones utilizadas en cada uno de los archivos y clases del programa.

3.1 Interpretación y análisis detallado de las características del problema

Como se ha mencionado en la introducción, hay ciertos aspectos del problema que pueden ser interpretados subjetivamente, llevando a una implementación u otra.

Para nuestra implementación, hemos tenido en cuenta las siguientes consideraciones:

- El número de bicicletas que carga y descarga una furgoneta en sus estaciones no puede ser cualquiera. Es decir, no haremos uso de un operador que modifique el número de bicicletas cargadas des de 1 a 30 y lo mismo para las estaciones de descarga, ya que consideramos que no tiene sentido. En nuestra implementación se utiliza una función que determina coherentemente el número de bicicletas a cargar y descargar.

  Por ejemplo, si en las estaciones de destino solo hace falte descargar un total de 14 bicicletas, esta función no permitirá que la furgoneta cargue más de 14 bicicletas en la estación de origen, ya que no existe un solo caso en el que cargar más bicicletas en esa ruta pudiera generar un beneficio. Además, esta función se encarga de que en la primera estación se descarguen todas las bicicletas necesarias hasta suplir la demanda de la estación o hasta no poder descargar más (debido a que la furgoneta no tiene más bicicletas para descargar). Esto se debe a que no existe ningún caso en el que se pueda generar un beneficio con el hecho de no descargar una bicicleta en la primera estación de destino para hacerlo posteriormente en la segunda.

  El único operador que tenemos relacionado con la carga/descarga de bicicletas es *ReducirNumeroBicicletasCarga*, que reduce el número de bicicletas de carga al primer múltiplo de 10 menor que el número de bicicletas de carga actual, ya que en la operación del cálculo del coste del transporte se utiliza una división entera entre 10 ya en ciertos casos esta reducción de bicicletas cargadas puede generar beneficios.

- En nuestra implementación, en ningún momento se puede dar que una furgoneta tenga como estación de descarga la misma estación en la que ha cargado bicicletas, ya que esto en ningún caso puede generar un beneficio y además puede conllevar problemas en el cálculo de beneficios.

  Además, una furgoneta siempre tiene dos estaciones de descarga asignadas (pudiendo ser la misma, tal y como se explicará en el siguiente punto).

  Una consecuencia de esta consideración es que en nuestro problema no existe la solución en la que las furgonetas no se mueven a ninguna estación de descarga. La solución válida más cercana a esta sería el caso en que las furgonetas cargan 0 bicicletas en todas las estaciones de carga, de manera que no importan las estaciones de descarga a las que vayan porque el coste de transporte será gratuito.

- Como se acaba de mencionar, en nuestras soluciones se permite que las dos estaciones de descarga sean la misma. En este caso, la función que asigna el número de bicicletas cargadas y descargadas se ocupa de que no se considere dos veces la misma estación (llevando a cargar el doble de bicicletas). Es decir, en este caso el resultado es como si la furgoneta solo tuviese una estación de destino, a pesar de que internamente tenga dos estaciones de descarga asignadas (la misma en ambas).

3.2 Estructura general

El programa entregado consta de 8 archivos Python para separar correctamente las distintas partes del código.

3.2.1 *main.py*

En este documento se encuentran todos los diferentes experimentos, cada uno representado por una función distinta. Cada una de estas funciones recibe unos ciertos parámetros y llama a otra función que se encuentra en el fichero functions_bicing.py (3.2.7). Al ejecutar este archivo, se crea un generador de semillas (*RNG*), desde el cual se generará una lista de 10 semillas, 9 generadas a partir de este generador, más la semilla 42, que es la que se nos proporciona en el enunciado de la práctica y hemos decidido incluirla. Así pues, este documento nos sirve para decidir qué experimento queremos ejecutar.

3.2.2 *state_bicing.py*

En este documento se encuentran la mayoría de funciones que permiten modificar y navegar por los diferentes estados. Tiene dos funciones estáticas: *__get_coords_est* y *__distancia_manhattan* que calculan, respectivamente, las coordenadas de una estación en concreto y la distancia de Manhattan entre dos coordenadas. Además, este documento cuenta con ciertas funciones privadas, que son las siguientes:

- *__restaurar_estaciones*: Restaura todas las estaciones a sus valores iniciales para poder recalcular las nuevas rutas sin que afecten los cambios anteriores. Dado que nuestro método *__realizar_ruta()* se llama en *heuristic()*, ha sido de mucha importancia implementar este método porque, al contrario de lo que pensbamos, AIMA frecuentemente hace más de una llamada a *heuristic()* por un mismo estado, de manera que es imprescindible que se restauren las estaciones en cada llamada a *heuristic()* para evitar rutas duplicadas y errores.

- *__realizar_ruta*: Llama a *__asignar_bicicletas_carga_descarga* y *__actualizar_valores_estaciones*

- *__asignar_bicicletas_carga_descarga*: Calcula y asigna el número de bicicletas que una furgoneta debe cargar de una estación de origen, así como las que tiene que descargar en dos estaciones de destino. Después actualiza estos valores en el objeto furgoneta para su uso posterior.

- *__actualizar_valores_estaciones*: Actualiza la información de la *diferencia* y *disponibilidad* de bicicletas de todas las estaciones de la furgoneta que se le pasa como argumento. Este método es muy importante para que todas las furgonetas tengan en cuenta las rutas de las otras furgonetas, evitando repeticiones o errores en la obtención del beneficio.

- *__calcular_balance_ruta_furgoneta*: Para una furgoneta, calcula el balance de la ruta que ha realizado basandose en las distancias que ha recorrido y el número de bicicletas que llevaba en cada trayecto.
- *__calcular_balance_rutas*: Calcula el balance de todas las rutas (el transporte, solo contabilizado en el heurístico 2).
- *__calcular_balance_estaciones*: Calcula el balance de todas las estaciones.
- *__calcular_balance_total*: Calcula el balance total (suma del balance de las rutas más el de las estaciones).

Hemos decidido hacer estas funciones privadas ya que sólo accedemos a ellas desde este mismo documento, y de esta manera nos aseguramos que no se puedan modificar desde otro fichero. Además, muchas las hemos creado para organizar el documento, que el código sea más entendible y esté más ordenado, ya que funciones como *__calcular_balance_rutas* y *__calcular_balance_estaciones* podrían estar dentro de *__calcular_balance_total*, pero de esta manera está mucho más claro qué función hace qué y aporta una mayor comprensibilidad a éste fichero, que es el más extenso de todos. Por otro lado, también tenemos distintas funciones públicas:

- *get_distancias_furgoneta*: Calcula las distancias de cada trayecto de una furgoneta en concreto y su suma total
- *heuristic*: Restaura las estaciones a sus valores por defecto y llama a los métodos encargados de calcular las rutas con las estaciones que han sido asignadas a cada furgoneta en el estado actual. Posteriormente llama a los métodos que, teniendo en cuenta el balance de rutas y el balance de estaciones, devuelven el balance obtenido para evaluar el coste heurístico de estado actual (En caso de que el heurístico sea 1, solo tiene en cuenta el balance de las estaciones).
- *generate_actions*: Genera todas las acciones posibles que se pueden tomar desde el estado actual del problema, devolviendo generador de todas estas acciones posibles. Este método actúa distinto en función de si se esta ejecutando el problema en Hill Climbing (hace todos los *yield*) o se esta ejecutando en Simulated Annealing (almacena todos los *yield* en una lista y escoge uno al azar).
- *apply_action*: Recibe un una acción como argumento y devuelve el nuevo estado (*EstadoBicing*) resultante al aplicar dicha acción al estado actual.
- *print_state*: Imprime el estado actual. Dependiendo del argumento inicial, la función mostrará si se trata del estado inicial o de la solución final. También imprime el balance de las rutas, el balance de estaciones y el balance total, calculados por las funciones internas *__calcular_balance_rutas*, __calcular_balance_estaciones y *__calcular_balance_total*, respectivamente.
- *visualize_state*: Crea una visualización del estado final utilizando el módulo *Pygame*. Este método es muy útil para ver si las rutas se estan realizando con coherencia o si hay errores evidentes. En la interfaz que se ejecuta, las estaciones verdes representan estaciones que solo son origen de alguna furgoneta; las estaciones amarillas representan esas que son tanto origen como destino de alguna furgoneta; y las estaciones rojas representan las que son solamente destino de una o más furgonetas.

3.2.3 *generate_initial_state_bicing.py*

Este fichero consta de la función *generate_initial_state*, la cual recibe como argumentos un entero *opt*, una semilla y un diccionario con los operadores que están activos (valores booleanos), y retorna un estado de *EstadoBicing*. La lista de operadores activos servirá para crear el estado inicial sólo teniendo en cuenta los operadores que tengan valor *True* i aquellos que no sean modificables. La semilla nos sirve para comparar los resultados de diferentes estrategias de optimización en las mismas condiciones, es decir, las diferencias en el creación del estado inicial solo serán atribuibles a las estrategias de optimización (opt), y no a la aleatoriedad al asignar furgonetas y estaciones.
En primer lugar, se crea el generador de números aleatorios a partir de la semilla proporcionada. Creamos también una lista con la información de cada estación: índice, diferencia y disponibilidad. Además, creamos una lista con todas las furgonetas, así como un *set()* que representa las estaciones que ya tienen una furgoneta asignada, con el fin de evitar que más de una furgoneta cargue en la misma estación.
Dependiendo del valor de *opt*, se asignan las estaciones de origen y destino a cada furgoneta siguiendo la estrategia correspondiente (0, 1 o 2).
Si $opt = 0$, se creará un estado inicial completamente aleatorio. Si $opt = 1$, se realizará la segunda estrategia, que tiene en cuenta la diferencia de las estaciones (con un cierto grado de aleatoriedad) y si $opt = 2$, se creará una distribución sin aleatoriedad, pero más optimizada que las dos anteriores.

3.2.4 *parameters_bicing.py*

Este documento sirve para poder ajustar los diferentes parámetros del proyecto como por ejemplo el número de estaciones, furgonetas, bicicletas o incluso la semilla utilizada para distribuir estas por el mapa. Este documento se carga en el archivo *main.py* para poder cambiar desde allí todos los valores. También se escoge allí el algoritmo utilizado para resolver el problema o los operadores que se deben usar en una ejecución.

3.2.5 *furgoneta_bicing.py*

En este documento se almacena la clase *Furgoneta* la cual tiene la siguiente información:

1. **id_estacion_origen:** Almacena un entero que referencia al id de la estación de carga de esa furgoneta.
2. **id_estacion_destino1:** Almacena un entero que referencia al id de la estación de descarga 1.
3. **id_estacion_destino2:** Almacene un valor entero que referencia al id de la estación de descarga 2.
4. **bicicletas_cargadas:** Este valor representa el número de bicicletas que se cargan en la estación de origen.
5. **bicicletas_descargadas1:** Este número equivale a las bicicletas descargadas en la primera estación de descarga (coincide con el máximo de bicicletas que se deben dejar en esa estación).
6. **bicicletas_descargadas2:** Este número equivale a las bicicletas descargadas en la segunda estación de descarga (coincide con *bicicletas_cargadas* - *bicicletas_descargadas1*).
7. **reducir_bicicletas_carga:** Este entero generalmente es 0 a no ser que el operador *ReducirNumeroBicicletasCarga* sea aplicado.

3.2.6 *operators_bicing.py*

Este documento incluye todos los distintos operadores que se han creado, aunque muchos de ellos no son usados debido a que en el experimento 1 se escogen solo los necesarios. Para empezar, se crea una clase abstracta llamada *BicingOperator*, la cual no incluye nada pero todos los operadores serán subclases de esta. El hecho de tener esta clase base en común es útil en las funciones de *generate_actions* y *apply_action*, que necesitan trabajar con diferentes tipos de operadores. Tenemos un total de 7 operadores donde cada uno almacena los valores necesarios para realizar su propósito en el métedo *apply_actions* del archivo *bicing_state* (detallados en el apartado 3.4.1).

3.2.7 *functions_bicing.py*

En este documento podemos encontrar las distintas funciones que se usan a lo largo del proyecto, sobretodo en los experimentos. Tenemos las funciones en otro documento para tener el código más limpio y que sea más fácil entender y comprender los experimentos que se encuentran en el *main.py*. Las funciones son las siguientes:

1. **ejecucion_individual_HC:** Esta función se utilitza para ejectuar una única vez el *hill_climbing* y ver las soluciones que ha comprobado y el tiempo total de ejecución. A la vez se muestra por pantalla el *visualizer* para ver el recorrido de las furgonetas.
2. **comparar_all_operadores**: Esta función es utilizada en el experimento 1 [4.1] para escoger los mejores operadores. Se comparan todas las opciones de operadores que no sean imprescindibles para escoger la mejor combinación.
3. **mejor_initial_state**: Esta función se utilitza en el experimento 2 [4.2] para comparar los diferentes estados iniciales que hemos creado.
4. **comparar_resultados_HC_SA**: Esta función se utilitza en diferentes experimentos pero sobretodo en el cinco [4.5]. Se utiliza para comparar el algoritmo Hill Climbing y el Simulated Annealing en igualdad de condiciones, es decir, utilizando las misma semillas por cada réplica y usando los mismos operadores en cada iteración.
5. **encontrar_parametros_SA**: Esta función se utiliza en el experimento 3 [4.3], su función es probar todas las combinaciones de diferentes hiperparámetros del Simulated Annealing y luego realizar un plot 3d para poder optimizar las ganancias.
6. **tiempos_experimento_con_aumento**: Esta función es utilizada en el experimento cuatro [4.4] para poder observar la tendéncia del tiempo respecto al incremento del problema.
7. **augmento_furgonetas**: Realiza múltiples ejecuciones de Hill Climbing con un incremento progresivo de furgonetas, para comprobar si el número de furgonetas es proporcional al beneficio obtenido. Es usada en el experimento 6 [4.6].

3.2.8 *problem_bicing.py*

En este fichero se especifica que *ProblemaBicing* es una subclase de *Problem*, y a continuación se especifican los métodos *actions*, *result*, *path_cost*, *value* i *goal_test*, que hacen lo siguiente:

- **actions:** Llama a la función *generate_actions* del archivo *state_bicing*.

- **results:** Llama a la función *apply_actions* del archivo *state_bicing*.
- **path_cost:** Devuelve c + 1 que es lo que devuelve siempre por defecto ya que en este problema no nos afecta.
- **value:** Devuelve el resultado de llamar a la función heurística para ese estado.
- **goal_test:** Devuelve siempre **False** ya que en una búsqueda local no nos interesa tener un *goal test* sino llegar a la mejor solución posible que se pueda encontrar.

3.3 Estado

Para representar el estado en este proyecto primero decidimos como se tenía que representar de la forma más eficaz computacionalmente. La información que necesita un estado se puede definir como el conjunto de información que tenemos sobre las estaciones del mismo estado y toda la información que disponemos de las furgonetas.

Para representar de forma eficiente todo lo mencionado, decidimos usar una lista llamada *lista_furgonetas* donde se guarda cada furgoneta; usamos una lista y no un *set()* ya que nos referimos a las furgonetas usando su posición en la lista. La información que difiere de estado a estado sobre las estaciones equivale a la diferencia, bicicletas que faltan o sobran en esa estación, y las bicicletas disponibles, inicialmente equivale a las bicicletas no usadas pero va variando dependiendo de las cargas y descargas.

Para representar esta información de manera que sea fácil acceder a ella usaremos un diccionario, *info_estaciones*, en vez de una lista ya que así accedemos la información con una llave, el id de la estación, y con una palabra nos referimos a si queremos la diferencia o las bicis disponibles.

Por último como en este proyecto jugamos mucho con los distintos operadores tenemos otro diccionario llamado *operadores_activos* que su función es en, un experimento, activar o desactivar los distintos operadores. Esto ha sido muy útil a la hora de decidir con que operadores quedarnos o cuales son muy poco eficientes.

3.3.1 Variables del estado

- **operadores_activos:** Diccionario con todos los operadores que se quieren usar en un experimento.
- **info_estaciones:** Lista de diccionarios con toda la información para cada estación (índice, diferencia y disponibilidad).
- **lista_furgonetas:** Lista con todas las instancias de furgonetas.

3.3.2 Restricciones

- Dos furgonetas no pueden cargar bicicletas en la misma estación.
- El máximo de bicicletas que puede cargar una furgoneta son 30.
- Una furgoneta no puede visitar más de dos estaciones.
- Una furgoneta solo puede cargar bicicletas en la estación de origen.

Con tal de no copiar información de más al generar nuevos estados, toda la información estática de las estaciones nos disponemos a obtenerla del archivo original para así no tener que copiarla, cosa que generaría posibles errores posteriormente y que, además, supone un coste muy elevado.

3.4 Operadores

3.4.1 Descripción de cada operador

1. **CambiarEstacionCarga:** Este operador se encarga de dar una nueva estación de carga para cada furgoneta. La única limitación es que no escoja una estación donde que ya es estación de origen de una furgoneta incluyendo la misma.

2. **CambiarOrdenDescarga:** Este operador cambia el orden de las estaciones de destino para cada furgoneta.

3. **IntercambiarEstacionCarga:** Este operador cambia la estación de carga entre dos furgonetas.

4. **CambiarEstacionDescarga:** Este operador genera una nueva estación de descarga para cada furgoneta. En total genera dos por furgoneta ya que cada furgoneta tiene dos estaciones de descarga.

5. **IntercambiarEstacionCarga:** Este operador se encarga de cambiar una estación de descarga de una furgoneta con otra estación de descarga de otra furgoneta.

6. **ReasignarFurgonetaInformado:** Este operador lo que hace es cambiar todos los valores de una furgoneta. Esto se hace para tratar de evitar máximos locales en el *hill_climbing* enviando una furgoneta que puede estar en un lugar sin sentido a la mejor estación de carga disponible (la que le sobran más bicicletas) y como destino las estaciones a la cuales les faltan más bicis.

7. **ReducirNumeroBicicletasCarga:** Este operador prueba de escoger un número distintos de bicis, en concreto el múltiplo de 10 más próximo al número actual de bicicletas a cargar. Esto se hace así ya que la fórmula para calcular el coste de transporte es ***((número_bicis + 9)//10) * distancia***, de aquí se puede deducir que como hay división entera de 10 debemos evitar llegar a un múltiplo de 10 y quedarnos en un múltiplo de 10 menos uno. Estos valores serían por ejemplo: 9, 19, 29, 39, 49... Pero como dentro del número de bicis ya se añaden 9 lo que debemos hacer es probar con múltiplos de 10; por eso nuestro número de bicicletas cargadas solo puede ser o un múltiplo de 10 o el valor que calculamos ya que cualquier otra opción daría menos dinero y así nos aseguramos no generar muchísimos estado de más que si tuviésemos un operador que probara todos los valores de bicicletas posibles a cargar.

3.4.2 Condiciones de aplicabilidad y sus efectos

- CambiarEstacionCarga

- Condiciones de aplicabilidad: Este operador es aplicable cuando una furgoneta necesita una nueva estación de carga y hay al menos una estación disponible que no sea la estación de origen de ninguna furgoneta, incluida la misma.

- Efectos: Cambia la estación de carga de la furgoneta seleccionada.

- CambiarOrdenDescarga

- Condiciones de aplicabilidad: Este operador es aplicable cuando una furgoneta tiene más de una estación de destino.
- Efectos: Reordena las estaciones de destino para la furgoneta seleccionada.

- IntercambiarEstacionCarga

- Condiciones de aplicabilidad: Este operador es aplicable cuando hay al menos dos furgonetas con estaciones de carga diferentes.
- Efectos: Intercambia las estaciones de carga entre dos furgonetas seleccionadas.

- CambiarEstacionDescarga

- Condiciones de aplicabilidad: Este operador es aplicable cuando una furgoneta necesita una nueva estación de descarga.
- Efectos: Genera una nueva estación de descarga para la furgoneta seleccionada.

- IntercambiarEstacionDescarga

- Condiciones de aplicabilidad: Este operador es aplicable cuando hay al menos dos furgonetas con estaciones de descarga diferentes.
- Efectos: Intercambia las estaciones de descarga entre dos furgonetas seleccionadas.

- ReasignarFurgonetaInformado

- Condiciones de aplicabilidad: Este operador es aplicable cuando una furgoneta está en una posición que no es óptima.
- Efectos: Reasigna todos los valores de la furgoneta seleccionada.

- ReducirNumeroBicicletasCarga

- Condiciones de aplicabilidad: Este operador es aplicable cuando el número de bicicletas a cargar no es un múltiplo de 10.
- Efectos: Ajusta el número de bicicletas a cargar al múltiplo de 10 más cercano al número actual.

3.4.3 Análisis del factor de ramificación de los operadores

$$\textbf{CambiarEstacionCarga} = \mathbf{O}(\mathbf{f} \times (\mathbf{e} - \mathbf{f})) \tag{9}$$

Este operador tiene un factor de ramificación que depende tanto del número de furgonetas f como del número de estaciones e. Si f es cercano a e, entonces $e - f$ será pequeño, reduciendo el factor de ramificación. Sin embargo, si f es mucho menor que e, el factor de ramificación será más grande. Por lo tanto, podría ser costoso en términos de tiempo y espacio computacional si hay muchas estaciones y furgonetas.

$$\textbf{IntercambiarEstacionCarga} = \mathbf{O}\left(\binom{\mathbf{f}}{\mathbf{2}}\right) \tag{10}$$

Este operador tiene un factor de ramificación que crece cuadráticamente con el número de furgonetas f. Específicamente, $\binom{f}{2} = \frac{f(f-1)}{2}$. Este operador será más costoso a medida que aumente el número de furgonetas.

$$\textbf{CambiarOrdenDescarga} = \mathbf{O(f)} \tag{11}$$

Este operador tiene un factor de ramificación lineal en función del número de furgonetas f. Es uno de los menos costosos en términos de ramificación, ya que solo genera un nuevo estado por cada furgoneta.

$$\textbf{CambiarEstacionDescarga} = \mathbf{O(2f \times (e-1))} \tag{12}$$

Este operador tiene un factor de ramificación que depende tanto de f como de e, similar a "CambiarEstacionCarga". Sin embargo, el factor adicional de 2 indica que este operador podría generar más estados que "CambiarEstacionCarga" si $e-1$ es significativo.

$$\textbf{IntercambiarEstacionDescarga} = \mathbf{O}\left(\mathbf{4} \times \binom{\mathbf{f}}{\mathbf{2}}\right) \tag{13}$$

Este operador tiene un factor de ramificación que es cuatro veces el de "IntercambiarEstacionCarga". Esto significa que podría ser uno de los operadores más costosos en términos de tiempo y espacio computacional, especialmente si el número de furgonetas f es grande.

$$\textbf{ReasignarFurgoneta} = \mathbf{O(f)} \tag{14}$$

Al igual que "CambiarOrdenDescarga", este operador tiene un factor de ramificación lineal en función de f. Es relativamente menos costoso en comparación con otros operadores.

$$\textbf{ReducirNumeroBicicletas} = \mathbf{O(f)} \tag{15}$$

Este operador también tiene un factor de ramificación lineal en función de f, similar a "ReasignarFurgoneta" y "CambiarOrdenDescarga", siendo éstos los menos costosos en términos de ramificación.

3.4.4 Explicación de la elección de los operadores

Los operadores que hemos escogido al final en nuestro proyecto han sido seleccionados a medida que íbamos avanzando la práctica. Primero iniciamos con los operadores *elementales* y luego fuimos añadiendo distintos operadores que creíamos que nos podrían resultar útiles para mejor el rendimiento de los algoritmos. Posteriormente comparábamos los resultados de usar ese operador o no y, si era suficientemente bueno, lo añadíamos a nuestros estado.

3.5 Generación de la solución inicial

Como se ha explicado en el apartado 3.2.3, cada estrategia de generación de la solución inicial representa un diferente grado de optimización de la misma. Si la semilla es la misma, se pueden

comprobar los respectivos cambios entre las diferentes estrategias, lo que más adelante no servirá de mucho con tal de escoger la mejor de las tres estrategias. Los tres diferentes grados de optimización son los siguientes:

Estrategia 0:
Se crea una distribución totalmente aleatoria, en la que, para cada furgoneta, su estación de origen es escogida aleatoriamente de entre las posibles estaciones que no tienen furgonetas asignadas, y las dos estaciones de destino son generadas aleatoriamente de entre todas las estaciones posibles.

Estrategia 1:
Esta estrategia sigue siendo aleatoria pero tiene ciertas mejoras respecto a la anterior. Creamos dos listas, una para las estaciones con diferencia positiva y la otra para las que tienen diferencia negativa. Para cada furgoneta, asignaremos una estación de origen perteneciente a la lista de las que tienen diferencia positiva (y tienen bicicletas disponibles), y las estaciones de destino solo podrán ser de las que tienen diferencia negativa. De esta manera, mejoramos en cierta parte la calidad de la solución inicial, ya que en todo momento estaremos quitando bicis de estaciones a las que les van a sobrar durante la siguiente hora, y las llevamos a estaciones que les faltan, por lo que nos aseguramos una cierta calidad, aunque el factor de la aleatoriedad sigue presente.

Estrategia 2:
Por último, la estrategia 2 es la única que no tiene nada de aleatoriedad, y en principio es la más optimizada de las tres estrategias diferentes. Como en la estrategia anterior, creamos una lista para las estaciones que tienen diferencia positiva y una para las que tienen diferencia negativa. Seguidamente, ordenamos la primera lista en orden descendente según la diferencia positiva entre bicicletas disponibles y demanda. Al hacerlo, las estaciones con el mayor exceso de bicicletas se colocan al principio de la lista y, de esta manera, las primeras furgonetas se asignarán a las estaciones con el mayor excedente, ayudando a redistribuir más eficazmente las bicicletas sobrantes. Por otro lado, la segunda lista (estaciones con diferencia negativa) se ordena en orden ascendente según la diferencia. Al hacerlo, las estaciones con la mayor falta de bicicletas (las que tienen diferencias negativas más grandes en valor absoluto) se colocan al principio de la lista, lo que provoca que las primeras furgonetas se asignen a las estaciones con la mayor necesidad de bicicletas.
Así pues, la asignación de furgonetas se realiza en función de estos ordenamientos para maximizar la eficiencia en el traslado de bicicletas desde las estaciones con excedente hacia las estaciones con necesidad.

3.6 Funciones heurísticas

3.6.1 Análisis de los factores que intervienen en la heurística

Nuestra heurística es una combinación de los siguientes factores:

- Estado de las estaciones: El estado de cada estación de bicicletas se mantiene en *self.info_estaciones*, que es una lista de diccionarios. Cada diccionario contiene el índice de la estación, la diferencia entre el número de bicicletas y la demanda ('dif'), y el número de bicicletas no utilizadas ('disp'). Este estado es muy importante para poder determinar dónde se necesitan más bicicletas y dónde hay un exceso.

- Estado de las furgonetas: Cada furgoneta tiene un estado que incluye su estación de origen, las estaciones de destino y el número de bicicletas que está transportando. Este estado es

dinámico y cambia con cada operación que realiza la furgoneta. Así pues, tener registrado el estado de cada furgoneta permite optimizar las rutas y minimizar el tiempo de transporte, lo cual influye directamente a la heurística.

- Coste de transporte: Se calcula en función de la distancia que recorre cada furgoneta y el número de bicicletas que está transportando. Para que éste tenga efecto en la heurística, el parámetro *coste_transporte* tiene que estar habilitado.
- Balance de Rutas y Estaciones: El método *__calcular_balance_total()* combina el balance de todas las rutas de las furgonetas y el balance de todas las estaciones para obtener un único valor que sirve como heurística. Este balance tiene en cuenta tanto el coste de transporte como la eficiencia en la asignación de bicicletas a las estaciones
- Operadores activos: Los operadores que están activos permiten modificar el estado actual para explorar nuevos estados, lo cual es esencial para una búsqueda de soluciones más óptimas.

3.6.2 Justificación de las funciones heurísticas escogidas

Las funciones heurísticas que hemos escogido se basan fundamentalmente en el beneficio obtenido para cada estado. El primer heurístico calcula el balance de dinero que se obtiene al sumar el dinero ganado por dejar las bicicletas en una estación necesitada. El segundo heurístico tiene en cuenta también el dinero que te quitan debido a las rutas de las furgonetas. Hemos escogido estos heurísticos ya que era lo más conveniente para realizar la práctica y para poder controlar, sobretodo, el dinero ganado y perdido que es realmente nuestro objetivo principal.

3.6.3 Explicación de los efectos de las funciones heurísticas en la búsqueda

El efecto de usar el primer heurístico respecto el segundo es muy grande, ya que si simplemente comparamos el coste computacional o temporal notamos una gran diferencia, donde para llegar a la solución óptima en el heurístico que no tiene en cuenta las rutas, no se necesitan casi recursos en comparación con el segundo.

3.6.4 Ponderaciones que aparecen entre los elementos de las heurísticas

En este problema, como hemos definido que el coste heurístico es la suma del balance de las estaciones más el balance de las rutas, no hace falta asignar diferentes ponderaciones, ya que estamos sumando beneficio en euros más beneficio en euros. Por lo tanto, en la función heurística los dos costes tienen el mismo peso.

4 Experimentos

A continuación se exponen los experimentos realizados, sus metodologías y sus resultados. En la mayoría de experimentos se usarán 10 semillas para la generación de diferentes réplicas, tal y como se ha explicado en el apartado 3.2.1.

4.1 Primer experimento

Condiciones:
Determinar qué conjunto de operadores da mejores resultados para una función heurística que optimice el primer criterio (el transporte es gratis) con un escenario en el que el número de estaciones es 25, el número total de bicicletas es 1250 y el número de furgonetas es 5, usando el algoritmo de Hill Climbing. Escoger una de las estrategias de inicialización de entre las propuestas. A partir de estos resultados, fijar los operadores para el resto de experimentos.

Características:

Observación	Pueden haber conjuntos de operadores que den mejores resultados que otros.
Planteamiento	Escogemos los cuatro operadores que podemos modificar (IntercambiarEstacionCarga, IntercambiarEstacionDescarga, CambiarOrdenDescarga y ReasignarFurgonetaInformado; explicados en el apartado 3.4.1) y observamos el beneficio medio obtenido con cada combinación de operadores posible.
Hipótesis	La mejor combinación es con todos los operadores activos.
Método	Elegiremos 10 semillas aleatorias[1], una para cada réplica. Para cada posible combinación de operadores activos, ejecutaremos 10 experimento para cada semilla. Siempre usaremos la estrategia de generación del estado inicial 1 (explicada en el apartado 3.2.3) porque tiene un cierto grado de aleatoriedad, obteniendo así resultados más fiables debido a que los operadores se encontraran en un estado inicial diferente para cada una de las 10 semillas.
Experimento	Experimentaremos con problemas de 5 furgonetas, 25 estaciones y 1250 bicicletas. Usaremos el algoritmo de Hill Climbing. Mediremos el beneficio medio obtenido (cociente de la suma de beneficios entre el número de iteraciones) y el tiempo de ejecución para realizar la comparación.

Table 1: Resumen de las características del Experimento 1.

Resultados:
Para el primer experimento hemos creado una función (*comparar_all_operadores*) que prueba todas las posibles combinaciones de operadores (solo los que se pueden activar/desactivar) y compara los resultados. Los operadores *CambiarEstacionCarga, CambiarEstacionDescarga* y *ReducirNumeroBicicletasCarga* deben mantenerse siempre activados, ya que son los operadores mínimos que garantizan que se pueda explorar todo el espacio de soluciones, así que la esta función solo evalúa

[1]Semillas: 42, 989, 796, 451, 119, 7, 92, 932, 965, 961. La generación de estas semillas ha sido explicada previamente en el apartado 3.2.1

los casos en que estos operadores están activados.

En la tabla [2] podemos ver los resultados del primer experimento tras 10 ejecuciones y con el estado inicial escogido ($opt = 1$). Solo se muestran las 5 mejores combinaciones de operadores (ordenadas de mayor a menor beneficio, y en caso de tener el mismo beneficio se ordenan de menor a mayor según el tiempo de ejecución) para evitar una tabla demasiado extensa.
Nos referimos a cada uno de los operadores de la siguiente forma:

- IntercambiarEstacionCarga: IEC.
- IntercambiarEstacionDescarga: IED.
- CambiarOrdenDescarga: COD.
- ReasignarFurgonetaInformado: RFI.

BENEFICIO MEDIO (€)	TIEMPO MEDIO DE EJECUCIÓN (ms)	OPERADORES
85.88	44.73	IEC, COD, RFI
85.86	45.30	RFI
85.84	51.47	IED, COD, RFI
85.82	53.44	IEC, IED, RFI
85.8	45.5	IEC, RFI

Table 2: Tabla con las 5 mejores combinaciones de operadores con heurístico 1, generación de estado inicial 1 y tras 10 ejecuciones para cada una de las 10 semillas.

Como podemos observar en la tabla [2], hay ciertas combinaciones que dan como resultado un beneficio de 85.8 pero hay una que supera a las demás por solo 2 céntimos. Además, es también la combinación más rápida. Por lo tanto, para los futuros experimentos nos quedaremos con los operadores *IntercambiarOrdenCarga*, *CambiarOrdenDescarga* y *ReasignarFurgonetaInformado*.

Observaciones:
Es importante mencionar que en este primer experimento no se está teniendo en cuenta el heurístico 2, de manera que la elección de estos operadores es muy poco probable que sea la óptima para esta función heurística. Hemos realizado este mismo experimento con el segundo heurístico para ver cuales serían los mejores operadores para el escenario en cuestión.

BENEFICIO MEDIO (€)	TIEMPO MEDIO DE EJECUCIÓN (ms)	OPERADORES
56.63	136.83	TODOS
56.51	136.70	IED, RFI, IEC
56.22	118.41	RFI, COD, IEC
56.17	124.04	RFI, IEC
56.08	129.45	IED, RFI

Table 3: Tabla con las 5 mejores combinaciones de operadores con heurístico 2, generación de estado inicial 1 y tras 10 ejecuciones para cada una de las 10 semillas.

Como podemos ver el la tabla [3], la mejor combinación de operadores para el heurístico 2 es la que usa todos los operadores disponibles, y la segunda y tercera los usan todos los que pueden menos uno. Por lo tanto, debido a que en el los siguientes experimentos tenemos que usar los operadores

seleccionados en este experimento con el heurístico 1, debemos ser conscientes que los resultados que se obtendrán en futuros experimentos con el heurístico 2 no serán óptimos.

4.2 Segundo experimento

Condiciones:
Determinar qué estrategia de generación de la solución inicial da mejores resultados para la función heurística usada en el apartado anterior, con el escenario del apartado anterior y usando el algoritmo de Hill Climbing. A partir de estos resultados fijaremos también la estrategia de generación de la solución inicial para el resto de experimentos.

Características:

Observación	Pueden haber métodos de inicialización que obtienen mejores soluciones.
Planteamiento	Escogemos tres métodos de inicialización diferentes (0, 1 y 2; explicados en el apartado 3.2.3) y observamos el beneficio medio obtenido con cada uno.
Hipótesis	El método de inicialización 2 es el que da mejor beneficio de los tres, ya que es el que crea una mejor solución inicial (como se ha visto en el apartado 3.2.3).
Método	Elegiremos 10 semillas aleatorias (las mismas que hemos usado en el primer experimento), una para cada réplica. Ejecutaremos 5 experimentos para cada semilla para la inicialización aleatoria y haremos medias de los resultados. Ejecutaremos 1 experimento para cada semilla para la inicialización informada.
Experimento	Experimentaremos con problemas de 5 furgonetas, 25 estaciones y 1250 bicicletas. Usaremos el algoritmo de Hill Climbing. Mediremos el beneficio medio obtenido (cociente de la suma de beneficios entre el número de iteraciones) para realizar la comparación.

Table 4: Resumen de las características del Experimento 2.

Resultados:
El experimento ha dado como resultado que las mejor estrategia de generación de la solución inicial es la 2; es decir, nuestra hipótesis parece ser cierta. Con 10 iteraciones para cada una de las 10 semillas, hemos obtenido los siguientes beneficios medios para cada estrategia de generación:

- Opt: 0 → Beneficio medio: 85.6€
- Opt: 1 → Beneficio medio: 85.82€
- Opt: 2 → Beneficio medio: 85.9€

Debido a que el experimento se ejecuta muy rápidamente, lo hemos ejecutado más veces. En alguna de estas veces el mejor resultado no ha sido el de la estrategia de generación de la solución inicial 2. Para estar seguros de que la mejor estrategia es la 2, hemos decidido ampliar el número de réplicas de 10 a 100.

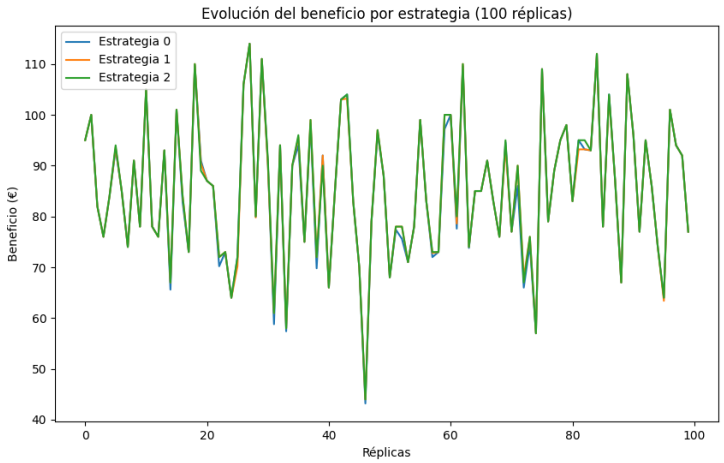

Figure 1: Evolución del beneficio con 100 réplicas para cada una de las estrategias de generación de la solución inicial.

Las nuevas medias obtenidas son:

- Opt: 0 → Beneficio medio: 85.102€
- Opt: 1 → Beneficio medio: 85.324€
- Opt: 2 → Beneficio medio: 85.4€

Como podemos observar en la figura [1] y en las nuevas medias obtenidas, una vez hemos ejecutado 100 réplicas distintas con 10 iteraciones cada una, la estrategia 2 sigue siendo la mejor, pero la diferencia es prácticamente imperceptible. Aún así, dado que la estrategia 2 es la única que no usa aleatoriedad, consideramos que la mejor estrategia sigue siendo esta.

Por lo tanto, para el resto de experimentos fijamos $opt = 2$ como nuestra estrategia de generación de la solución inicial.

4.3 Tercer experimento

Condiciones:
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.

Características:

Observación	Pueden haber parámetros del Simulated Annealing que proporcionen mejores resultados que otros.
Planteamiento	Escogemos una lista de valores para el parámetro k, una para el parámetro λ, determinamos un límite de pasos y observamos el beneficio medio obtenido con cada uno.
Hipótesis	Los parámetros $k = 1$ y $\lambda = 0.0001$ darán los mejores resultados.
Método	Generaremos todas las posibles combinaciones de parámetros. Realizaremos 5 ejecuciones en cada una de las 10 semillas para cada combinación de parámetros, con el objetivo de que los resultados del Simulated Annealing sean más fiables (menos susceptibles a su aleatoriedad). Usamos la estrategia de generación de la solución inicial 2, seleccionada en el segundo experimento. Usaremos el conjunto de operadores determinado en el primer experimento. Usaremos el heurístico 1, ya que así lo pide el enunciado. Los valores de k que probaremos serán 0.001, 0.01, 0.1, 1, 10 y 100. Los valores de λ que probaremos serán 0.0001, 0.001, 0.01, 0.1, 0.99. Inicialmente probaremos el experimento con límite de 1000 pasos.
Experimento	Experimentaremos con problemas de 5 furgonetas, 25 estaciones y 1250 bicicletas. Usaremos el algoritmo de Simulated Annealing. Mediremos el beneficio medio obtenido para realizar la comparación y determinar cual ha sido el mejor conjunto de parámetros.

Table 5: Resumen de las características del Experimento 3.

Resultados:
Inicialmente hemos realizado el experimento con un límite de pasos de 1000. Los resultados se pueden ver en la figura [2]. Además, en el output de la función que se llama para ejecutar este experimento (*encontrar_parametros_SA()*) se nos indica que hay tres combinaciones de valores que comparten el valor máximo de beneficio, tal y como podemos ver en la tabla [6].

K	λ	Beneficio (€)
1	0.1	85.9
0.01	0.01	85.9
0.01	0.001	85.9

Table 6: Mejores combinaciones de valores para Simulated Annealing con límite de 1000 pasos.

Con estos resultados podemos concluir que nuestra hipótesis no era correcta, ya que la combinación $k = 1$ y $\lambda = 0.0001$ no se encuentra entre ninguna de las que proporcionan mejores resultados.

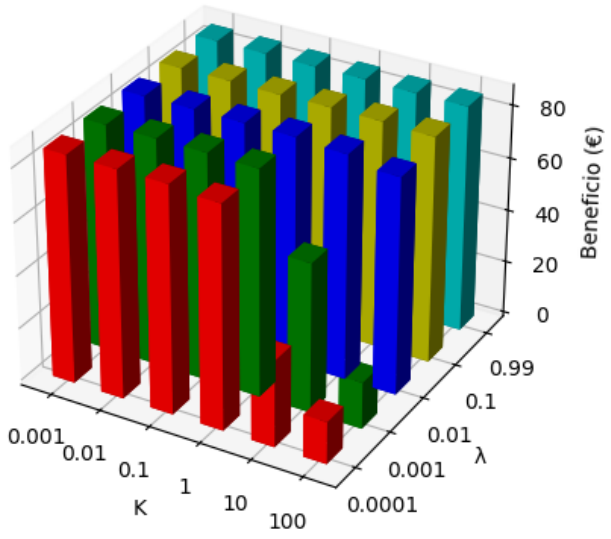

Figure 2: Beneficio en función de los valores de k y λ. Limite de 1000 pasos y se hacen 5 ejecuciones por cada una de las 10 semillas en cada combinación de valores.

Dado que los resultados con mayor beneficio llegan a un valor de 89.9€ (muy parecido al valor máximo que nos generaba Hill Climbing en el segundo experimento y con la misma generación de solución inicial), creemos que se trata de el máximo valor que se puede obtener con las semillas proporcionadas. Además, el hecho de que haya tres combinaciones de parámetros que nos hayan proporcionado el mismo beneficio máximo nos indica que con alta probabilidad hay alguna limitación que impide seguir mejorando el beneficio.

Para poder asegurar que realmente el valor no pueda ser mejorado por el algoritmo Simulated Annealing con un límite superior, probaremos de realizar el experimento de nuevo, pero con un límite de 25000 pasos y únicamente con las posibles combinaciones de valores k y λ que nos generaban mejores beneficios con el límite en 1000. Realizaremos una sola iteración por cada una de las semillas en cada combinación de parámetros para evitar tiempos de ejecución excesivos.

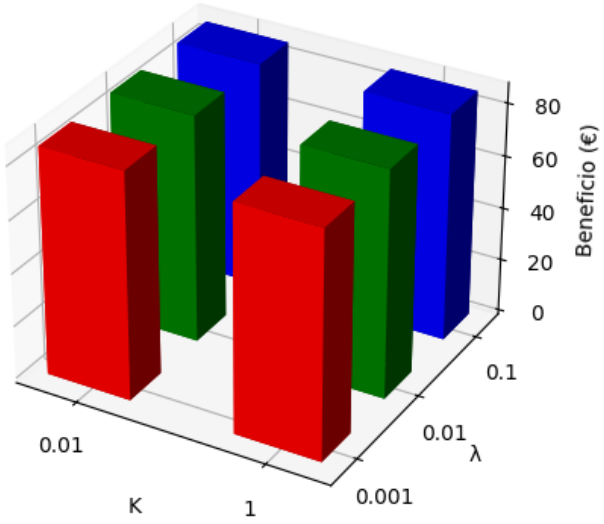

Figure 3: Beneficio en función de los valores de k y λ. Limite de 25000 pasos y se hace una sola ejecución por cada una de las 10 semillas en cada combinación de valores.

En la figura [3] podemos ver que no se superan los 89.9€ con el límite de 25000 pasos. Por lo tanto, entre los valores que comparten los mejores resultados en 1000 iteraciones, hemos decidido quedarnos arbitrariamente con la combinación $k = 1$ y $\lambda = 0.1$. Así pues, usaremos estos valores para futuros experimentos con Simulated Annealing. Además, dado que los resultados con límite 25000 no han mejorado los de límite 1000, elegimos quedarnos con el límite de 1000 pasos para una mayor eficiencia.

Observaciones:
Al estar usando solamente el heurístico 1 para este experimento, los parámetros escogidos pueden no ser los mejores para el heurístico 2. Especialmente el límite, que solo es de 1000 pasos, lo que puede ser suficiente para el primer heurístico, pero no para el segundo debido a que se generarán muchos más estados. Por lo tanto, en próximos experimentos se ha de tener en cuenta que los resultados obtenidos con Simulated Annealing y el heurístico 2 pueden no ser los óptimos.

4.4 Cuarto experimento

Condiciones:
Dado el escenario de los apartados anteriores, estudiad como evoluciona el tiempo de ejecución para hallar la solución en función del número de estaciones, furgonetas y bicicletas asumiendo una proporción 1 a 50 entre estaciones y bicicletas y 1 a 5 entre furgonetas y estaciones. Para ello empezad con 25 estaciones e id aumentándolas de 25 en 25 hasta que veáis la tendencia. Usad el

algoritmo de Hill Climbing y la misma heurística.

Características:

Observación	El tiempo de ejecución varia en función de la complejidad del problema (número de estaciones, furgonetas y bicicletas).
Planteamiento	Queremos comprobar como afecta la complejidad del problema al tiempo de ejecución del programa.
Hipótesis	El tiempo de ejecución incrementa exponencialmente cuando se aumenta el número de estaciones, furgonetas e bicicletas utilizadas para el problema.
Método	Realizaremos tandas de 10 ejecuciones (una para cada semilla de las 10 determinadas en el primer experimento) para la complejidad inicial del problema y mediremos el tiempo de ejecución medio. Posteriormente iremos incrementando el número de estaciones (y proporcionalmente el de furgonetas y bicicletas) y realizando 10 nuevas ejecuciones (con las mismas semillas) para cada configuración nueva, midiendo también el tiempo medio de ejecución. Probaremos incrementando la complejidad del problema 10 veces. En caso de que la tendencia no sea clara realizaremos más incrementos. Usaremos el conjunto de operadores y estrategia de generación de la solución inicial sacados de los resultados de los dos primeros experimentos. En este experimento no realizaremos réplicas con diferentes semillas, ya que esto no afecta a la tendencia que sigue el tiempo de ejecución al incrementar la complejidad del problema. Usaremos el heurístico 1, ya que así lo pide el enunciado.
Experimento	La primera tanda de 10 ejecuciones se hará con 25 estaciones, 5 furgonetas y 1250 bicicletas. Para cada nueva tanda incrementaremos en 25 en número de estaciones (y proporcionalmente el número de furgonetas y de bicicletas; es decir, sumando 5 furgonetas y 1250 bicicletas en cada incremento). Usaremos el algoritmo de Hill Climbing. Mediremos el tiempo medio de cada tanda de 15 ejecuciones y analizaremos su evolución para en función de la complejidad del problema.

Table 7: Resumen de las características del Experimento 4.

Resultados:
Al realizar todas las ejecuciones, podemos ver los resultados obtenidos en la figura [4]. En este gráfico se ve claramente que la evolución del tiempo de ejecución crece exponencialmente a medida que aumentamos linealmente la complejidad del problema. Por lo tanto, nuestra hipótesis es correcta.

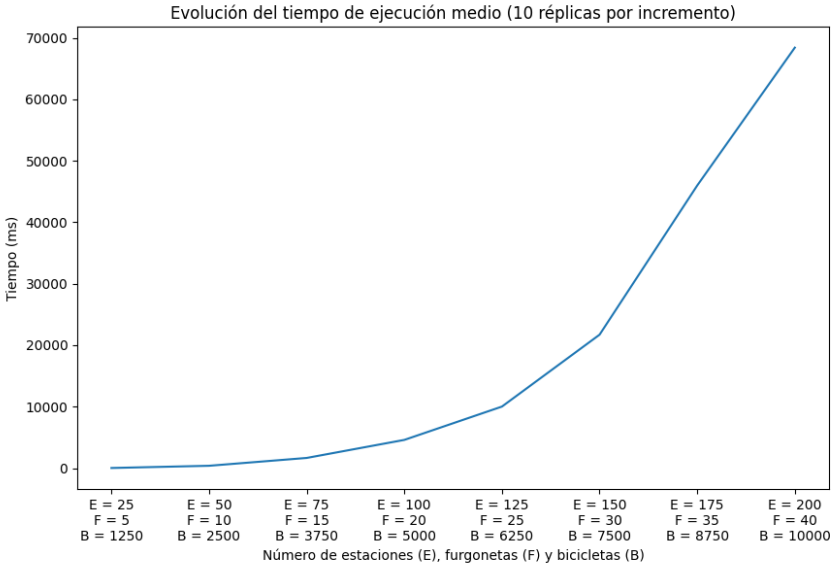

Figure 4: Evolución del tiempo medio de ejecución en función de la complejidad del problema (número de estaciones, furgonetas y bicicletas).

Observaciones:
Cabe destacar que era un resultado esperable, ya que como habíamos predicho en el apartado de operadores (3.4.3), al aumentar el número de furgonetas y estaciones, el factor de ramificación de algunos operadores sería mucho mayor, lo que provocaría que el tiempo de ejecución también aumentase considerablemente.
Por lo tanto, este gráfico permite reafirmar nuestro análisis anterior, así como nuestra hipótesis en este mismo experimento.

4.5 Quinto experimento

Condiciones:
Dado el escenario del primer apartado, estimad la diferencia entre el beneficio obtenido, la distancia total recorrida y el tiempo de ejecución para hallar la solución con el Hill Climbing y el Simulated Annealing para las dos heurísticas que habéis implementado (los experimentos con la primera ya los tenéis).

Características:

Observación	Los dos algoritmos pueden dar beneficios, distancias y tiempos de ejecución diferentes.
Planteamiento	Queremos estudiar cómo varía el beneficio obtenido, la distancia total recorrida y el tiempo de ejecución entre el algoritmo de Hill Climbing y Simulated Annealing.
Hipótesis	Los dos algoritmos proporcionarán resultados similares en términos de beneficio, distancia y tiempo.
Método	Usaremos el conjunto de operadores y estrategia de generación de la solución inicial sacados de los resultados de los dos primeros experimentos. Utilizaremos la semilla 42 para la generación de estados iniciales en ambos algoritmos.
Experimento	Experimentaremos con problemas de 5 furgonetas, 25 estaciones y 1250 bicicletas. Usaremos los algoritmos de Hill Climbing y Simulated Anealing. Mediremos el beneficio medio obtenido, la distancia total recorrida y el tiempo de ejecución para realizar la comparación y determinar qué algoritmo da mejores resultados

Table 8: Resumen de las características del Experimento 5.

Resultados:

- Beneficio obtenido

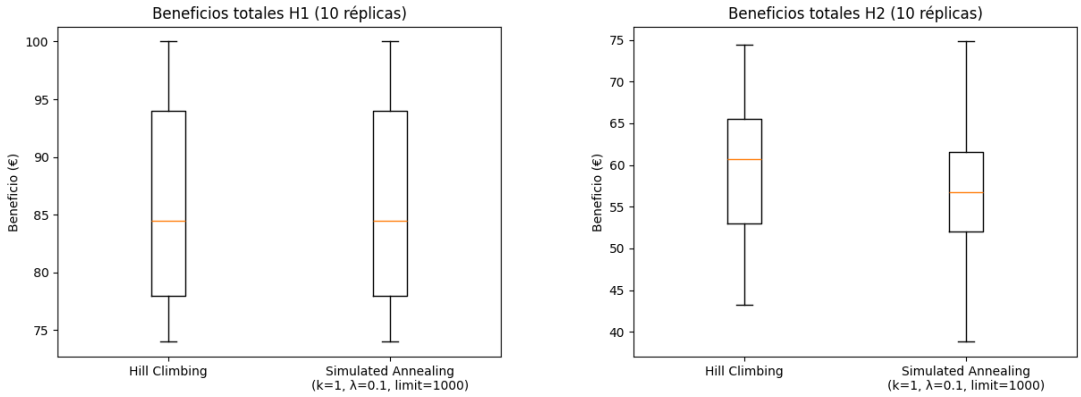

Figure 5: Comparación de los beneficios obtenidos en Hill Climbing y Simulated Annealing para ambos heurísticos. Simulated Annealing con un límite de 1000 pasos.

Como podemos observar en la figura [5], para la heurística 1, los resultados entre Hill Climbing y Simulated Annealing con 1000 pasos de límite son muy similares, mientras que para el heurístico 2 sí que vemos una pequeña mejora del HC, donde la mediana supera los 60 (€) de beneficio, mientras que para el SA está entre 55 y 60.

- Tiempo de ejecución

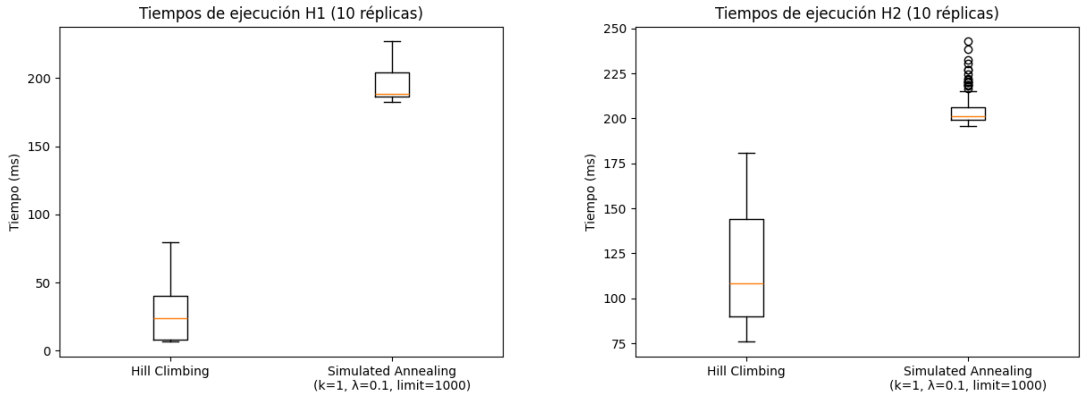

Figure 6: Comparación de los tiempos de ejecución en Hill Climbing y Simulated Annealing para ambos heurísticos. Simulated Annealing con un límite de 1000 pasos.

Para el tiempo de ejecución, podemos observar (figura [6]) cómo hay una mejora muy significativa entre ambos algoritmo, ya que claramente HC tiene un tiempo de ejecución muy por debajo del que se obtiene con SA. En heurística 2, la diferencia sigue siendo muy considerable, aunque en relación a la heurística 1, ha sido el algoritmo de HC el que ha sufrido un mayor aumento del tiempo de ejecución, manteniéndose aún por debajo del tiempo de SA.

- Distancias totales

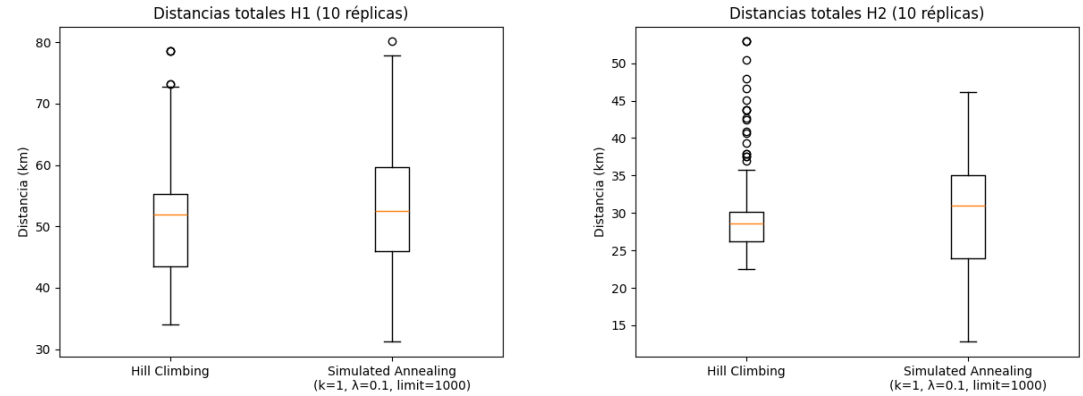

Figure 7: Comparación de las distancias totales en Hill Climbing y Simulated Annealing para ambos heurísticos. Simulated Annealing con un límite de 1000 pasos.

Por último, para las distancias totales recorridas (figura [7]), vemos que en heurística 1 las medianas de los dos algoritmos son muy similares, pero lo que diferencia los resultados obtenidos en ambos es la varianza, ya que podemos observar cómo SA muestra una mayor variabilidad en las distancias recorridas en comparación con HC, lo que nos puede indicar que, en este sentido, HC es un poco más consistente en sus resultados que SA.

Por otro lado, para la heurística 2, podemos ver como Hill Climbing presenta muchos outliers en el extremo superior, lo que indica que en algunas ejecuciones, HC puede terminar con soluciones significativamente peores que la mediana, hecho que puede deberse a que HC se puede quedar atascado en óptimos locales.
Simulated Annealing, por otro lado, proporciona soluciones más consistentes, aunque su mediana es más alta (lo que indica que en general encuentra soluciones que son más largas en distancia). Esto se debe a que SA tiene la capacidad de escapar de óptimos locales aceptando soluciones peores con una cierta probabilidad.

Una vez realizado el experimento, cómo hemos visto que en ciertos casos los resultados obtenidos son bastante similares en ambos algoritmos, hemos decidido aumentar el número de pasos en Simulated Annealing de 1000 a 25000, y así observar si aparece alguna diferencia significativa o los resultados obtenidos son los mismos. Así pues, volvemos a estudiar los gráficos de benficio, tiempo de ejecución y distancia total recorrida:

- Beneficio (SA con 25000 pasos)

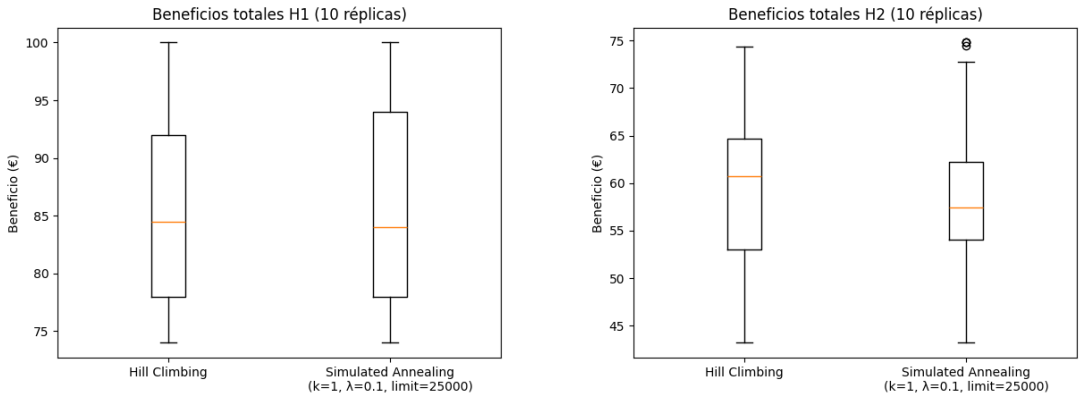

Figure 8: Comparación de los beneficios obtenidos en Hill Climbing y Simulated Annealing para ambos heurísticos. Simulated Annealing con un límite de 25000 pasos.

Podemos ver (figura [8]) que los resultados en cuanto a beneficios siguen siendo muy similares. En heurística 2 volvemos a tener una mejora en HC, tal y como hemos obtenido en el mismo gráfico pero con 1000 pasos de límite en SA. Por lo tanto, podemos concluir que en el primer heurístico prácticamente no hay diferencias entre los dos algoritmos, mientras que en el segundo heurístico, HC obtiene unos beneficios mejores que SA.
- Tiempo de ejecución (SA con 25000 pasos)

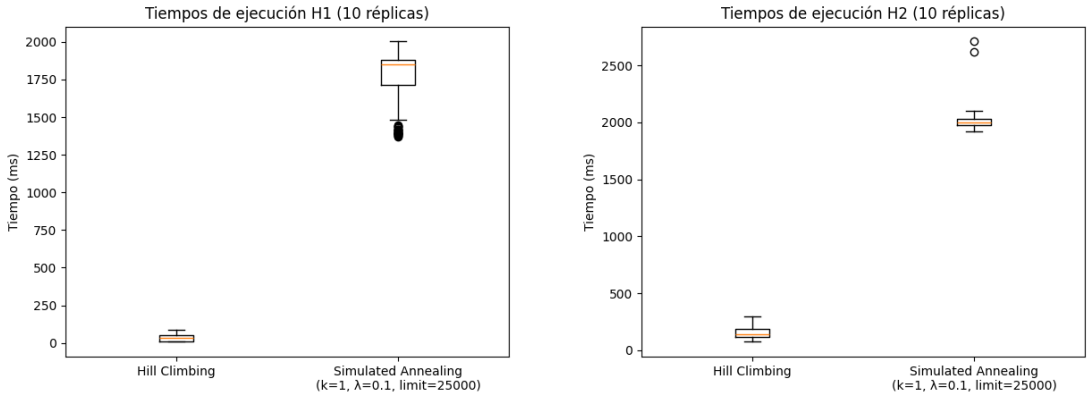

Figure 9: Comparación de los tiempos de ejecución en Hill Climbing y Simulated Annealing para ambos heurísticos. Simulated Annealing con un límite de 25000 pasos.

En el tiempo de ejecución, vemos (figura [10] cómo en los dos heurísticos hay una diferencia incluso más grande que anteriormente, ya que ahora SA tarda mucho más en ejecutarse, lo cual era de esperar ya que tiene un límite de pasos muy superior.

- Distancias totales (SA con 25000 pasos)

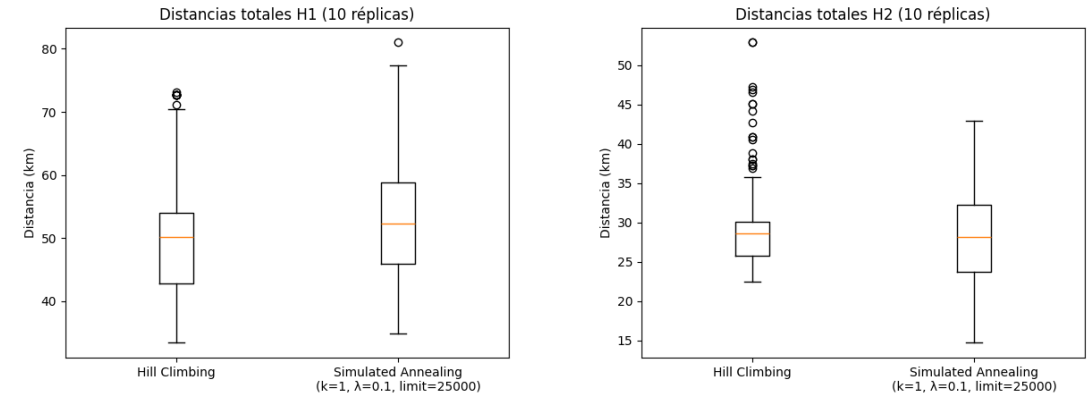

Figure 10: Comparación de las distancias totales en Hill Climbing y Simulated Annealing para ambos heurísticos. Simulated Annealing con un límite de 25000 pasos.

Finalmente, los algoritmos dan los mismos resultados en cuanto a distancias totales que anteriormente, donde en el primer heurístico, HC tiene una mediana por debajo que SA, pero en el segundo heurístico es SA el más consistente de los dos, ya que HC presenta mucha variabilidad en los resultados.

4.6 Sexto experimento

Condiciones:
Dado el escenario del primer apartado, estimad cual es aproximadamente el número de furgonetas que son necesarias para obtener la mejor solución. Para ello empezad con 5 furgonetas e id aumentándolas de 5 en 5 hasta que no haya una mejora significativa.

Características:

Observación	El número de furgonetas afecta al beneficio total obtenido.
Planteamiento	Queremos comprobar si existe un punto en el que incrementar más el número de furgonetas no afecta al beneficio total obtenido.
Hipótesis	El tiempo de ejecución incrementa exponencialmente cuando se aumenta el número de estaciones, furgonetas e bicicletas utilizadas para el problema.
Método	Realizaremos tandas de 10 ejecuciones (una para cada semilla de las 10 determinadas en el primer experimento) para la cantidad inicial de furgonetas y mediremos el beneficio medio obtenido. Posteriormente iremos incrementando solamente el número de furgonetas y realizando 10 nuevas ejecuciones (con las mismas semillas) para cada configuración nueva, midiendo también el tiempo medio de ejecución. Probaremos incrementando el número de furgonetas 5 veces. En caso de que la tendencia no sea clara realizaremos más incrementos. Usaremos el conjunto de operadores y estrategia de generación de la solución inicial sacados de los resultados de los dos primeros experimentos. Usaremos el heurístico 1, ya que así lo pide el enunciado.
Experimento	La primera tanda de 10 ejecuciones se hará con 25 estaciones, 5 furgonetas y 1250 bicicletas. Para cada nueva tanda incrementaremos en 5 el número de furgonetas). Usaremos el algoritmo de Hill Climbing. Mediremos el tiempo medio de cada tanda de 10 ejecuciones y analizaremos su evolución para en función del número de furgonetas.

Table 9: Resumen de las características del Experimento 6.

Resultados:
Como se puede observar en la figura [11], el resultado es como lo imaginábamos, es decir, el número de furgonetas afecta al beneficio hasta cierto punto. Concretamente al sobrepasar las 15 furgonetas ya no se aprecia crecimiento. Es muy coherente que si solo tenemos 25 estaciones no necesitamos más furgonetas que esa cantidad debido a que, por ejemplo, si tenemos 10 furgonetas entre las 10 estaciones de carga y las 20 de descarga ya suman más que el total de estaciones en el mapa. Por lo tanto el resultado es tal cual lo esperábamos.

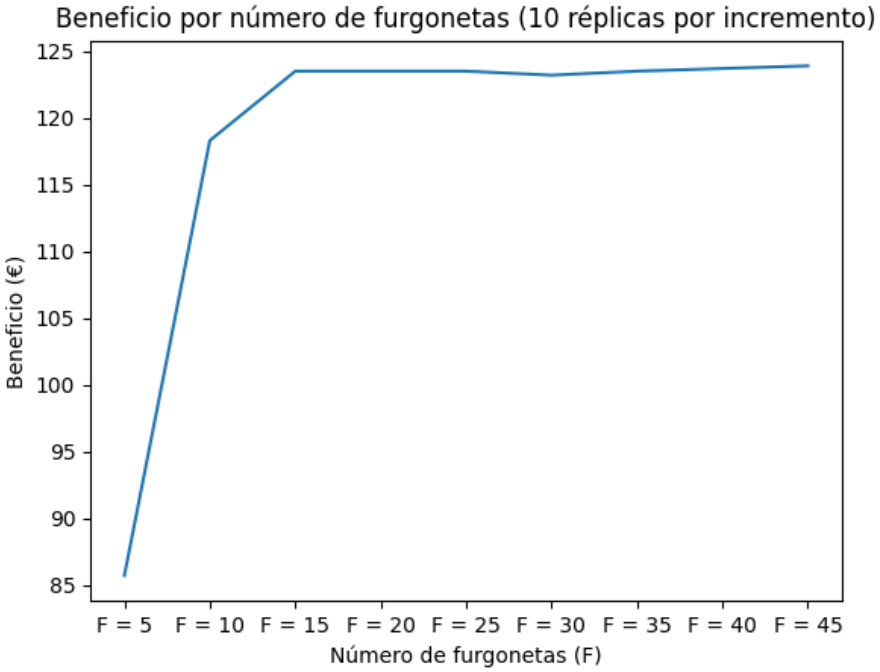

Figure 11: Evolución del beneficio medio en función del número de furgonetas.

4.7 Experimento especial

Este experimento ya ha sido previamente enviado a los profesores de la asignatura mediante correo electrónico, ya que debía entregarse en una fecha anterior a la entrega del informe final.

5 Hill Climbing vs Simulated Annealing

Como se ha podido observar en el quinto experimento, los resultados del Simulated Annealing no son, para nada, mejores que los del Hill Climbing. De hecho si los comparamos proporcionalmente por el tiempo se puede ver claramente que el Simulated Annealing es más de 10 veces más lento. Eso se podría deber a que no estamos dejando suficientemente libertad al algoritmo debido al límite de iteraciones que hemos escogido.

Sin embargo, también hace falta mencionar que el Hill Climbing es muy propenso a encontrar máximos locales y por lo tanto, aunque computacionalmente no sea el mejor algoritmo Simulated Annealing probablemente con los valores de hiperparámetros adecuados y con un número de iteraciones suficientes, sería capaz de ganar cualquier valor obtenido por el Hill Climbing y llegar a considerables máximos absolutos.

6 Conclusiones

Una vez finalizado el trabajo, podemos afirmar que hemos podido comprobar, gracias a los experimentos realizados, las principales diferencias de entre usar los algoritmos de Hill Climbing (HC) y Simulated Annealing (SA).

En el caso del algoritmo de Hill Climbing, hemos podido confirmar que el tiempo de ejecución aumenta de manera exponencial conforme crece la complejidad del problema. Esta observación está relacionada con lo que se esperaría dadas las características del algoritmo, particularmente cuando el espacio de búsqueda se expande al incrementar el número de estaciones, furgonetas y bicicletas. Cabe destacar que, aunque en nuestro problema sea una muy buena opción con tal de encontrar soluciones óptimas, este comportamiento entre el tiempo de ejecución y la complejidad del problema podría hacer poco práctico el uso de HC para problemas mayores, ya que el tiempo de ejecución podría llegar a ser demasiado elevado, haciendo que fuese muy poco óptimo realizarlo con este algoritmo.

Por otro lado, nuestros experimentos comparativos entre Hill Climbing y Simulated Annealing nos han permitido observar ciertas diferencias sutiles pero importantes en términos de beneficios, tiempo de ejecución y distancias totales. Aunque ambos algoritmos han mostrado un rendimiento similar en cuanto a los beneficios obtenidos, Hill Climbing tiene un tiempo de ejecución mucho menor, especialmente cuando incrementamos el número de pasos en Simulated Annealing de 1000 a 25000. Esto nos indica claramente que en nuestro problema, si queremos buscar una solución óptima con 5 furgonetas, 25 estaciones y 1250 bicicletas (parámetros con los que se hizo el experimento pertinente), la mejor opción es realizarlo con Hill Climbing, ya que obtiene el mismo beneficio y se ejecuta más rápidamente.

Además, la experimentación con diferentes heurísticas demostró que la elección de la heurística puede afectar no solo el rendimiento del algoritmo en términos de calidad de la solución sino también en su eficiencia computacional, ya que Hill Climbing, por ejemplo, tuvo un rendimiento ligeramente mejor con la segunda heurística.

6.1 Valoración del aprendizaje adquirido

Este trabajo nos ha permitido entender en profundidad qué es un problema de búsqueda local, así como el cómo implementarlo.
Por ejemplo, inicialmente tuvimos ciertas confusiones a la hora de entender el problema, ya que no todos teníamos la misma visión de cómo realizar el código del problema y nos costaba entender conceptos como el espacio de soluciones o qué era un estado inicial.

Una vez finalizado el trabajo, podemos afirmar que satisfactoriamente hemos comprendido y aplicado los elementos que involucran un problema de búsqueda local, así como la comprensión de cómo funcionan los algoritmos de Hill Climbing y Simulated Annealing detalladamente.

En cuanto al código, la parte que nos presentó más problemas fue la función *generate_actions*, y fue donde tuvimos que dedicar más tiempo, ya que muchas veces nos aparecían errores que no identificábamos de donde provenían. Por lo tanto dedicamos mucho tiempo a, primero identificar un error, y después solucionarlo. Cabe destacar que nos ayudó mucho el uso del *debuger*, ya que

nos permitió en muchos casos ver qué linea era la errónea.

Así pues, llegados a este punto, podemos concluir que nuestros conocimientos sobre código Python y sobre algoritmos en IA han aumentado en gran parte, y hemos aprendido a cómo implementar correctamente un problema de búsqueda local con diferentes algoritmos, así como ver las diferencias entre éstos.