Universitat Politècnica de Catalunya

Inteligencia Artificial

Práctica de búsqueda local IA - Bicing

Jordi Muñoz, Jianing Xu y Yiqi Zheng

30/10/2023

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Índice

1. Introducción

El propósito principal de esta primera experiencia en el laboratorio era adentrarse en las técnicas de solución de problemas basadas en la búsqueda local. En particular, exploramos y empleamos los enfoques de Hill Climbing y Simulated Annealing, dos estrategias ampliamente reconocidas en el ámbito de la inteligencia artificial en la rama de la optimización matemática.

Para llevar a cabo esta tarea, hicimos uso de la librería AIMA en Java, que ya cuenta con la implementación de estas dos técnicas. Como veremos más adelante, la utilización de esta librería nos facilitó mucho la tarea, ya que únicamente tuvimos que proporcionar a cada una de ellas nuestro estado actual del problema, la función generadora de sucesores, un verificador de estado final y una función heurística.

Una vez leída y releída la práctica, analizamos todos los matices y complicaciones que podríamos encontrarnos. A continuación, presentamos los resultados de nuestra solución al problema y de nuestros experimentos.

2. Descripción del problema

El problema que se nos plantea consiste en optimizar la distribución de bicicletas en las estaciones de Bicing, de manera que la mayor cantidad de usuarios pueda disponer de una bicicleta en una hora concreta cuando la necesite.

Identificamos que es un problema de búsqueda local porque se desea optimizar unos criterios, que son maximizar la demanda suplida y minimizar el coste de transporte, sin necesidad de llegar a un óptimo global. Además, si quisieramos conseguir el óptimo global, sería muy difícil hacerlo con un algoritmo de búsqueda informada como A* por ejemplo debido a que no es trivial encontrar una heurística para ello, y en caso de encontrarlo, el coste temporal y espacial sería excesivo.

Para poder resolver este problema, el servicio de Bicing nos proporciona la siguiente información:

- GetNumBicicletasNoUsadas: Una previsión de cuántas bicicletas no serán utilizadas en una estación durante una hora específica y que podrían ser reubicadas en otra estación.
- GetNumBicicletasNext: Una previsión de cuántas bicicletas habrá en una estación la hora siguiente a la actual. Esta información tiene en cuenta solo los cambios en la cantidad de bicicletas en una estación debidos a los usuarios, es decir, solo si no se transportan bicicletas entre estaciones por otros medios.
- GetDemanda: Una previsión de cuántas bicicletas debería haber en una estación a la hora siguiente a la actual para cubrir la demanda prevista (la suma de la demanda puede ser mayor que el número total de bicicletas).

Para simplificar el problema, supondremos que la ciudad es un cuadrado de 10x10 kilómetros y que las calles forman una cuadrícula donde cada manzana tiene 100×100 metros y en donde las estaciones de Bicing se encuentran en los cruces entre las calles.

El cálculo de la distancia en metros entre dos puntos de la ciudad se hará mediante la función:

$$d(i, j) = |i_x - j_x| + |i_y - j_y|$$

Donde i_x y i_y son las coordenadas x e y en metros del punto i en la cuadrícula.

Nos fijamos que si hay una estación en una esquina del mapa que le sobran 30 bicicletas y otra estación que necesita 30 bicicletas para suplir su demanda en la otra esquina del mapa, el coste de transporte para una furgoneta será de 78 euros, pero el beneficio será solo de 30. Por lo tanto, el factor coste de transporte es importante y se tendrá que evaluar desde qué estaciones a que estaciones es más optimo transportar bicis, disminuyendo lo posible este coste.

La cantidad de estaciones de bicicletas en la ciudad es E, y el número total de bicicletas es B, las cuales están distribuidas entre todas las estaciones. El número total de bicicletas es, evidentemente, constante, por lo que no se pueden quitar ni añadir bicicletas al sistema. Asumiremos que el número de bicicletas que cabe en una estación es ilimitado. Bicing ha observado que existen dos escenarios diferentes de demanda: el escenario de EQUILIBRIUM, en el cual la demanda de bicicletas en cada estación es más o menos similar, y el de RUSH_HOUR, en el cual algunas estaciones experimentan una demanda mayor que otras.

También contamos con la variable F, que representa la cantidad de furgonetas disponibles que podemos utilizar para trasladar bicicletas de una estación a otra. Es importante tener en cuenta que el número máximo de bicicletas que podemos transportar es de 30, y cada estación sólo puede ser el punto de origen de una furgoneta una vez, pero puede ser el destino cuantas veces sea necesario. Esto puede ser un problema, ya que al poder solo usar una vez una estación como origen, la estación en caso de sobrarle más de 30 bicicletas, estaría perdiendo potencial.

Lo que habrá que optimizar del problema será el beneficio, que aumenta a medida que añadimos bicicletas a una estación y se acerca a su demanda, y disminuye en caso contrario, que se aleje de la demanda. También habrá que tener en cuenta el coste de transporte, que tendrá la forma de $((nb + 9)\ div\ 10)€/km$, siendo nb el número de bicicletas que se transportan en una furgoneta y div 10 la división entera. Con esta fórmula de coste, uno podría plantearse llevar las bicicletas en packs de 10 en 10 ya que eso no supondría ningún coste adicional respecto al precio por kilómetro.

Al final, la solución tendrá que indicar:

- Cuál ha de ser el origen de la furgoneta, los destinos que tendrá (dos como máximo) y en qué orden los recorrerá.
- Cuántas bicicletas coge en su origen y cuántas deja en cada destino cada furgoneta.

3. Implementación del estado

El objetivo principal es que el estado contenga los atributos mínimos necesarios para identificar entre un estado del problema y otro, y permita operar con eficiencia en espacio y tiempo para la búsqueda local. Además, debe poder representar una solución al problema. Guardamos solo la información que varía entre estados, mientras que la información invariable se almacena de manera estática para evitar duplicados.

A continuación, se explica de manera detallada nuestra implementación:

Clase State

Esta clase representa completamente el estado del problema, en el cuál podemos identificar qué rutas hacen las furgonetas, las bicis que contienen las estaciones, entre muchos elementos más.

Los atributos que contiene la clase State son:

- **fleet:** matriz con F filas y 6 columnas. Cada fila representa una furgoneta y las columnas representan en orden: el id de la estación origen, el número de bicicletas cogidas de la estación origen, el id de la estación destino 1, el número de bicicletas dejadas en el destino 1, el id de la estación destino 2 y el número de bicicletas dejadas en el destino 2.
  Este atributo es imprescindible para representar la ruta de las furgonetas y las bicis que lleva y deja.
- **isStationVisited:** vector de E enteros, que nos indica el número de furgoneta que tiene por origen la estación i. En caso de no tener, el valor es -1. Este atributo es útil para la comprobación de la restricción de que una estación no puede ser origen de dos furgonetas, operación realizada constantemente.
- **bikesNeeded:** vector de E enteros que nos indica en cada posición i, el número de bicis que necesita la estación i para suplir la demanda de la hora siguiente. En caso de que el valor sea negativo, representa el número de bicis que le sobran. Es imprescindible ya que nos facilitará el cálculo de las rutas de las furgonetas para saber a qué estaciones se pueden coger y a cuáles dejar para que nos proporcionen beneficio.
- **transportCost:** double que representa el coste de transporte de las bicis con la configuración de las rutas de las furgonetas actual.
- **benefit:** double que representa el beneficio obtenido, calculado con suppliedDemand - transportCost.
- **suppliedDemand:** double que representa el número de bicis transportadas y dejadas a una estación que contribuyen beneficio. Es útil para no tener que ir recalculando desde cero (recorrido de la matriz fleet) la demanda suplida cuando se aplica un operador que lo altera.
- **stations:** objecto *estático* de la clase Estaciones que representa las estaciones de Bicing en su inicialización.
- **E:** entero *estático* que representa el número de estaciones.
- **F:** entero *estático* que representa el número de furgonetas.

El tamaño del espacio de búsqueda será de los posibles valores que puede tomar la matriz fleet que es de $O(F \cdot 30^3 \cdot E^3)$.

- Como para una furgoneta, las posibles estaciones origen es E, posibles destino 1 seria E-1, y posibles destino 2 seria E-2, con un total de $O(E^3)$.
- Además, para cada estación hay que añadirle la posibilidad de coger o dejar entre 0 y 30 bicis para cada estación asignada, contribuyen al coste a $O(30^3 \cdot E^3)$.
- Finalmente, como tenemos F furgonetas, obtendremos $O(F \cdot 30^3 \cdot E^3)$.

4. Operadores elegidos

La elección de los operadores ha estado influenciada por el tipo de algoritmo utilizado, que es la búsqueda local. En la búsqueda local, se parte de una solución inicial y se explora el espacio de soluciones hasta encontrar una mejor solución. Por lo tanto, los operadores elegidos deben ser capaces de generar nuevas soluciones que sean válidas, es decir, que cumplan con las restricciones del problema.

Hemos elegido 3 operadores, pese a que hemos llegado a tener un total de 5 operadores pero que acabamos descartando porque solo aumentaban el factor de ramificación y no aportaban mucho a la solución final. Éstos son:

- **ChangeDestination1**: dada la *vanId* de una furgoneta y la *newDestId* de una estación, se intercambia el destino 1 al que iba la furgoneta por la nueva dada. Factor de ramificación: O(F·E).
    - Condiciones de aplicabilidad:
        - La furgoneta *vanId* debe tener un origen.
        - La estación *newDestId* del nuevo destino no puede ser la id estación del origen de la furgoneta ni la de dest1.
        - La estación *newDestId* debe necesitar bicicletas.
- **ChangeDestination2**: dada la *vanId* de una furgoneta y la *newDestId* de una estación, se intercambia el destino 2 al que iba la furgoneta por la nueva dada. Factor de ramificación: O(F·E).
    - Condiciones de aplicabilidad:
        - La furgoneta *vanId* debe tener un origen y un destino 1.
        - La estación *newDestId* del nuevo destino no puede ser la id estación del origen de la furgoneta ni tampoco el de su destino 1.
        - La estación *newDestId* debe necesitar bicicletas.
- **ChangeOrigin**: dada la *vanId* de una furgoneta y la *newOriginId* de una estación, se intercambia el origen por el cual se parte. Si otra furgoneta ya tenía esa estación como origen, las dos furgonetas de intercambian estaciones origen. Factor de ramificación: O(F·E).
    - Condiciones de aplicabilidad:
        - La estación *newOriginId* no puede ser ni el destino 1 ni el destino 2 de la furgoneta *vanId*.
- **SubstractVan**: dada la *vanId* de una furgoneta y un entero $i \mid 0 \leq i \leq 9$, se quitan y bicicletas de las cogidas en la estación origen de la furgoneta dada. Factor de ramificación: O(F).
    - Condiciones de aplicabilidad:
        - La furgoneta *vanId* tiene estación origen.
        - Las bicicletas que coge la furgoneta *vanId* es mayor o igual a *i*.

ChangeDestination1 y ChangeDestination2

Estos operadores son muy buenos cuando partimos de una solución inicial en la que el número de bicicletas que se cogen de un origen es bajo, ya que al final iremos intercambiando esas estaciones por otras que tengan muchas más.

Como el beneficio va en función de la cantidad de bicicletas que se le dan a una estación, cuando una estación necesita pocas bicicletas, la podemos intercambiar por otra que necesite muchas más, lo que aumentará la cantidad posible de beneficio que podemos alcanzar.

Además, a veces nos interesa cambiar el destino actual por otro si el nuevo destino está más cerca de la furgoneta, ya que así se reduce el coste de transporte.

ChangeOrigin

Este operador funciona de manera inversa a los ChangeDestination, ya que va muy bien cuando partimos de una solución en la que la estación de origen tiene una cantidad baja de bicicletas que le sobran que al final resultará siendo una que tenga más que la inicial si es lo más óptimo.

Con los mismos argumentos que en ChangeDestination, tener este operador aumenta la cantidad de bicicletas que nos sobran, para a su vez aumentar la cantidad de beneficio a la que podemos llegar. Además, nos viene bien para disminuir el coste de transporte en caso de que la estación origen de una furgoneta esté muy apartada de sus destinos haciendo que el coste de transporte sea elevado.

SubstractVan

Este operador no lo hemos acabado usando pese a que está implementado, porque realizando los experimentos, vimos que su efecto era nulo. Al principio pensábamos que era buena idea ya que si sobraba una pequeña cantidad de bicicletas y la distancia de recorrido de la furgoneta era elevada, el coste podría aumentar bastante pero hemos comprobado que eso no sucede ya que la demanda suplida son números naturales y el coste de transporte puede ser menor a 1.

5. Estrategias para la generación de la solución inicial

En búsqueda local, siempre es deseable empezar desde un estado inicial que sea una solución, para realizar una búsqueda únicamente dentro del espacio de soluciones.

Para ello, cada furgoneta en el estado inicial debe tener una ruta planificada, que se traduce en tener un origen, destinos y las bicis cogidas y dejadas.

Primero, las furgonetas al inicio no tienen un origen determinado. Para su asignación, pensamos que debían tener como origen una estación, dado que una furgoneta no podía estar inicialmente en un punto aleatorio de la ciudad, ya que para ser útil debía pasar sí o sí por una estación a coger bicis. Además, tuvimos en cuenta que dos furgonetas diferentes no podían tener la misma estación como origen, pero una estación sí podía recibir bicis de más de una furgoneta.

5.1. Solución inicial aleatoria o EASY

Esta solución inicial inicializa el origen de las furgonetas a estaciones de manera aleatoria, cogiendo todas las bicicletas no usadas. Asimismo, asigna como primer destino de estas furgonetas, otras estaciones, también, de forma aleatoria. Es decir, en ningún caso nos preocupamos de si a una estación le faltaran o le sobraran bicicletas en la hora siguiente. Todo ello comprobando que cumplen la condición de ser un estado solución. Coste: $O(F + E)$

5.2. Solución inicial MEDIUM

Esta solución inicial recorre las estaciones linealmente sin ningún tipo de ordenación previa y va asignando la primera estación con bicis sobrantes en la siguiente hora, a la primera estación que vea que le falten bicis en la siguiente hora, teniendo en cuenta también que si las bicis que ha dejado en este primer destino es menor que el número de bicis que ha cogido de la estación origen, puede y debe ir también a un segundo destino que será la siguiente estación que nos encontremos en la estructura de las estaciones con bicis sobrantes.
Coste: $O(F \cdot E^3)$

5.3. Solución inicial ordenada voraz o HARD

Esta solución inicial hace una ordenación según las bicis que le sobran a cada estación de forma decreciente, de mas sobrante a menos sobrante, y seguidamente, asigna una furgoneta a las F primera estaciones, que serán las que les sobren más en la siguiente hora, y las va asignando a las F últimas estaciones, que serán las que les falten más en la siguiente hora, teniendo en cuenta también la posibilidad de que vaya a un segundo destino en el caso de que las bicis que haya cogido de la estación origen menos las bicis que le faltaban al primer destino sea positivo, de esta manera nos aseguramos que transporte todas las bicis cogidas, obteniendo la mayor demanda suplida posible.
Coste: $O(E^2 + F \cdot E)$, donde E^2 proviene del coste de la ordenación implementado con selection sort por su simplicidad (se podría substituir por un merge sort con coste $E \cdot \log E$ pero es más complejo de implementar) y $F \cdot E$ de la asignación de las furgos a estaciones.

6. Funciones heurísticas

Para realizar un problema de búsqueda local, es imprescindible utilizar una función heurística que guiará al algoritmo a la hora de elegir qué sucesor es mejor. Esta función debe ser admisible e informativa ya que representará la calidad de los distintos sucesores que genera el algoritmo de búsqueda.

Dado que el enunciado nos pedía dos heurísticas, una que tiene en cuenta el coste de transporte y la otra que los discrimina para escoger una heurística o otra simplemente utilizamos una variable, que puede ser 0 o 1, en las funciones que calculan coste de transporte, la cual multiplica el coste para o bien tenerlo en cuenta (cuando la variable es 1) o discriminarlo (cuando la variables es 0).

Para calcular el coste del transporte utilizamos la fórmula: $((nb + 9)\ div\ 10)/1000 * d(i,j)$, donde $d(i, j) = |i_x - j_x| + |i_y - j_y|$, que previamente ha sido comentada en la descripción del problema.

7. Generación de estados sucesores

Una vez realizada la generación de soluciones iniciales e implementados los operadores, implementamos las funciones encargadas de generar sucesores.

Para ello, utilizamos dos algoritmos de búsqueda local, Hill Climbing y Simulated Annealing, a continuación vemos cómo lo hicimos.

7.1. Hill Climbing

La generación de estados sucesores para Hill Climbing está compuesta por 1 bucle for que recorre las F furgonetas que tenemos disponibles. Este bucle for se compone de 3 bucles for adicionales que cada uno servirá para aplicar individualmente 1 de los 3 operadores.

7.2. Simulated Annealing

En el caso de Simulated Annealing, se escoge todo de manera aleatoria, tanto el operador que vamos a aplicar como los elementos del estado que vendría a ser la id de la furgoneta a utilizar, el origen, el destino 1 y el destino 2. Todo esto se implementa con la clase Random.

8. Experimentación

8.1. Influencia de los operadores utilizados

Tenemos que determinar cuáles son los operadores más utilizados y que mejoran la solución para descartar los que son innecesarios. Nos hemos fijado que en este experimento, al utilizar la función heurística 1 (coste de transporte es gratis), nuestro operador substractVan que resta el número de bicis que lleva una furgoneta, nunca mejora la heurística y, por lo tanto, inútil en este experimento y no lo consideraremos.

Para los otros operadores, changeDestination1, changeDestination2 y changeOrigin no sabemos de primeras qué combinación va a ser mejor, así que, vamos a experimentar con estos.

- cjt1 = changeOrigin + changeDest1
- cjt2 = changeOrigin + changeDest2
- cjt3 = changechangeOrigin + changeDest1 + changeDest2

Experimento a realizar:

Observación	Hay operadores que no se utilizan, ya sea porque no mejoran la solución o porque otros operadores son mejores y, por lo tanto, son innecesarios (y sólo aumentan la memoria y tiempo)
Planteamiento	Escogemos diferentes combinaciones de operadores que aplicaremos para los estados sucesores
Hipótesis	La combinación que utiliza los tres operadores es la mejor
Método	- Fijaremos el escenario con número de estaciones a 25, núm. total de bicis a 1250, núm. de furgonetas a 5, demanda equilibrada y heurística que optimice el primer criterio (transporte gratis). - Usaremos el algoritmo de Hill Climbing con inicialización medium (asigna el origen de las furgonetas a las primeras estaciones que les sobran, y lo mismo con las destinaciones), ya que la inicialización hard ya nos obtiene el máximo beneficio sin utilizar ningún operador. - Ejecutaremos 10 semillas diferentes para cada combinación de operadores. - Mediremos el **beneficio obtenido** y el **tiempo de ejecución** para realizar la comparación.

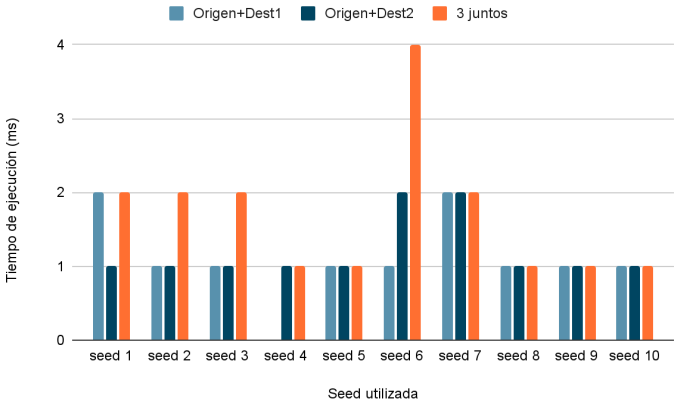

Figura 1: Tiempo de ejecución con cada conjunto de operadores en milisegundos (ms)

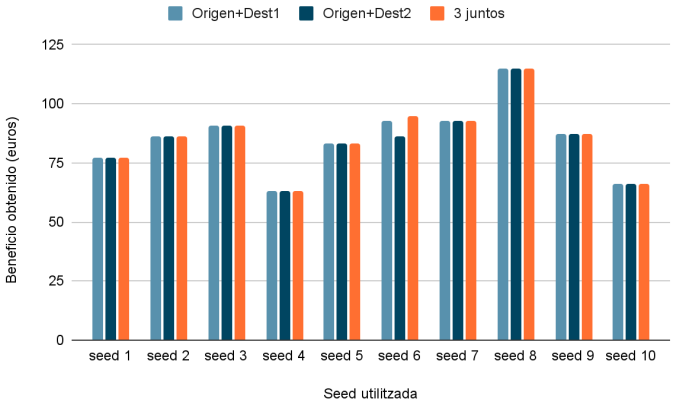

Figura 2: Beneficio obtenido con cada conjunto de operadores

En un principio, esperábamos que la combinación que utiliza los tres operadores sería la mejor. Sin embargo, podemos observar en los resultados que esto podría ser falso, ya que todas las combinaciones han obtenido el mismo beneficio en todas las ejecuciones, con la excepción de una.

Aun así, es importante destacar que nuestra generación de la solución inicial ya llegaba a un óptimo local en cuanto a beneficio, con muy poca exploración, cuando se utiliza la heurística de coste de transporte gratuito. Por lo tanto, creemos que los resultados no reflejan la verdadera ventaja de la aplicación de los tres operadores, que nos permitiría explorar más.

Por este motivo, utilizaremos la combinación de los tres operadores juntos, ya que creemos que funcionará mejor en situaciones donde se tenga en cuenta el coste de transporte en la heurística. En la mayoría de esos casos, nuestra generación de la solución inicial no llegaría a un óptimo local.

8.2. Influencia de la solución inicial

Tenemos que determinar si hay métodos de generación de la solución inicial mejores que otros. Para ello, consideraremos tres métodos distintos de inicialización: random (easy), medium, greedy (hard), explicados en la sección de generación de solución inicial del documento.

Experimento a realizar:

Observación	Pueden haber métodos de inicialización que nos permiten obtener mejores soluciones
Planteamiento	Escogemos diferentes métodos de inicialización y observamos sus resultados
Hipótesis	La inicialización de solución inicial greedy es mejor que las otras
Método	- Fijaremos el escenario con número de estaciones a 25, núm. total de bicis a 1250, núm. de furgonetas a 5, demanda equilibrada y heurística que optimice el primer criterio (transporte gratis). - Utilizaremos los operadores elegidos en la 1a experimentación. - Usaremos el algoritmo de Hill Climbing. - Ejecutaremos 10 semillas diferentes para las diferentes

	generación de la solución inicial. - Mediremos el **beneficio obtenido** y el **tiempo de ejecución** para realizar la comparación.

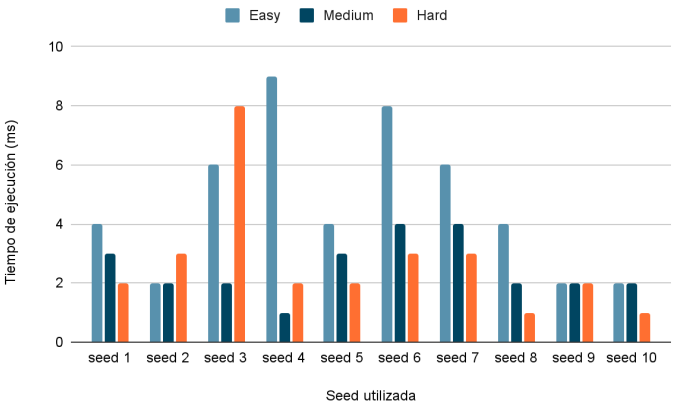

Figura 3: Tiempo de ejecución para cada tipo de inicialización

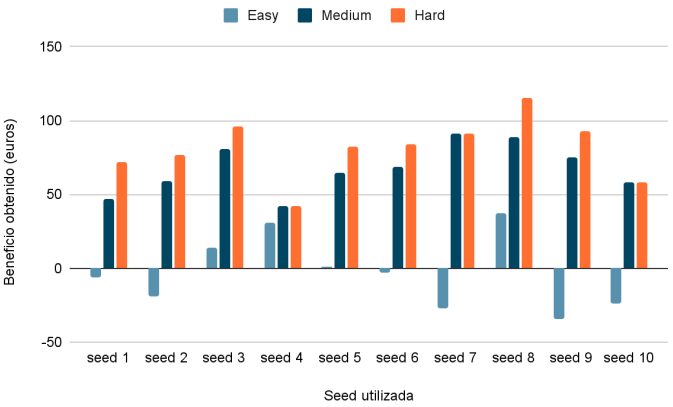

Figura 4: Beneficio obtenido para cada tipo de inicialización utilizando HC

Antes de llevar a cabo el experimento, esperábamos que la solución inicial "hard" nos proporcionaría mejores resultados.

Tras realizar el experimento, podemos observar que nuestra hipótesis parece ser cierta. Hemos notado que la generación "greedy" se ejecuta más rápido en la mayoría de los casos en comparación con otras inicializaciones, y además, obtiene el resultado máximo en todas las instancias experimentadas. La “greedy” nos acerca lo suficiente a un óptimo local para ahorrarnos hacer una búsqueda excesiva.

Por lo tanto, hemos decidido utilizar la inicialización "hard" para el resto de los experimentos, ya que creemos que nos permitirá obtener soluciones de mayor calidad en menos tiempo.

8.3. Parámetros que dan mejor resultado para el Simulated Annealing

Tenemos que determinar qué valores para los parámetros del algoritmo Simulated Annealing nos proporcionan una solución mejor, que es, el mayor beneficio. Para ello, fijaremos el número de iteraciones a 100.000 para asegurarnos que es suficientemente grande para que el algoritmo llegue a converge a una solución para los diferentes valores de λ y K.

Experimento a realizar:

Observación	Pueden haber combinaciones de parámetros que son significativamente mejores que los otros
Planteamiento	Vamos a elegir 5 valores de λ y 5 valores de K y vamos a observar los resultados
Hipótesis	Todos las combinaciones de parámetros λ y k del Simulated Annealing nos darán los mismos resultados
Método	- Fijaremos el escenario con número de estaciones a 25, núm. total de bicis a 1250, núm. de furgonetas a 5, demanda equilibrada y heurística que optimice el primer criterio (transporte gratis). - Utilizaremos los operadores elegidos en la 1a experimentación y inicialización elegida en la 2a experimentación. - Usaremos el algoritmo Simulated Annealing con 100.000 iteraciones, step = 20, probando para cada combinación de k y λ con estos valores K = {1, 5, 25, 50, 125} y para λ = {1, 0.1, 0.01, 0.001, 0.0001}. - Para cada combinación, ejecutaremos con 10 semillas diferentes. - Mediremos el beneficio obtenido para la comparación.

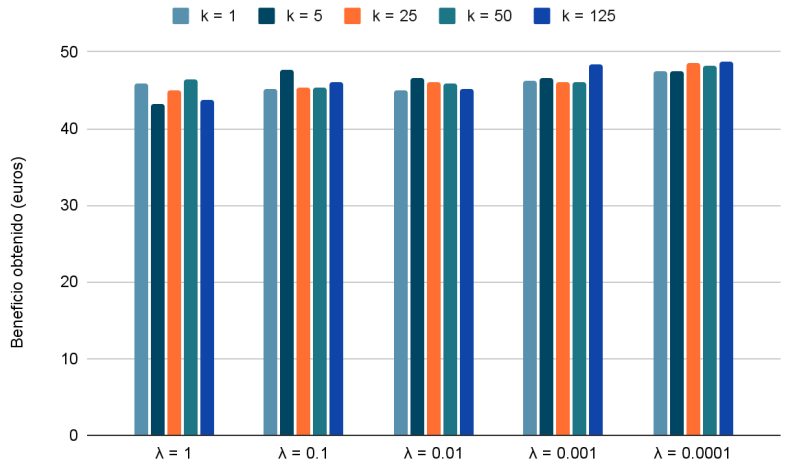

Figura 5: Valores k / λ de Simulated Annealing

A partir de los resultados, hemos observado que en el algoritmo Simulated Annealing, la combinación del valor más pequeño de λ (=0.0001) junto con el valor más grande de K (=125) nos proporciona la mejor solución y por lo tanto, la hipótesis de que todas la combinaciones nos darían un resultado idéntico debería ser rechazada.
Por lo tanto, vamos a asumir que la combinación λ=0.0001 y K=125 es buena y la emplearemos en los experimentos siguientes.

8.4. Evolución del tiempo de ejecución con HC

Tenemos que estudiar la evolución del tiempo de ejecución en función del número de estaciones, furgonetas y bicicletas. Nos dan el dato de que la proporción entre estaciones y bicicletas es de 1/50, y la que hay entre furgonetas y estaciones es de 1/5.

Experimento a realizar:

Observación	El tiempo de ejecución es posible que siga una función creciente respecto al tamaño del problema
Planteamiento	Escogemos diferentes tamaños de problema y observamos sus tiempos de ejecución
Hipótesis	La función del tiempo de ejecución será creciente respecto al tamaño del problema
Método	- Experimentaremos con problemas de E=25, 50, 75, …, hasta 250 estaciones, con bicis siendo B=E*50, furgos F=E/5 y utilizando la heurística 1. - Utilizaremos los operadores elegidos en la 1a experimentación y inicialización elegida en la 2a experimentación. - Ejecutaremos usando el algoritmo de Hill Climbing, 10 ejecuciones con semillas aleatorias para cada tamaño y haremos medias de los resultados. - Mediremos el tiempo de cómputo para realizar la comparación.

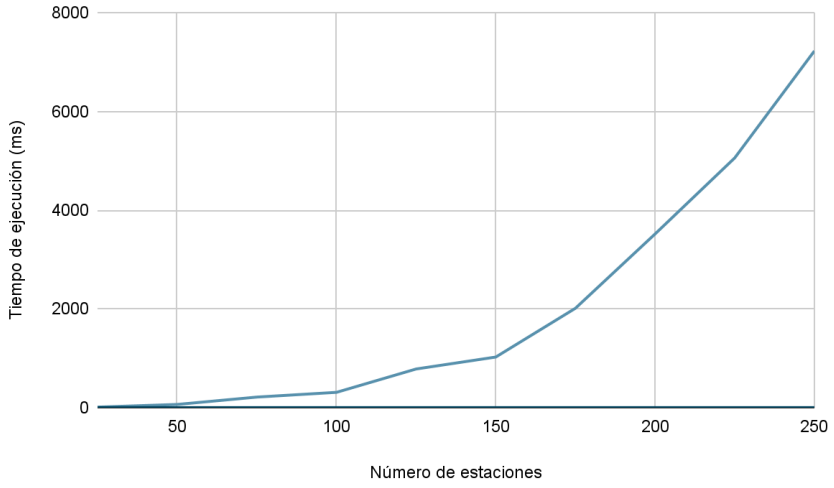

Figura 6: Tiempo empleado para la ejecución de un problema variando el número de estaciones, bicis y furgonetas

Los resultados obtenidos indican que el algoritmo de Hill Climbing requiere más tiempo de ejecución a medida que el tamaño del problema aumenta. Observamos que este aumento no es lineal, sino que se asemeja más a una curva exponencial. Estos resultados nos dan a entender que nuestra hipótesis inicial puede ser cierta.

8.5. Comparación de resultados de HC con SA

Queremos comparar los dos algoritmos de búsqueda local para las dos heurísticas implementadas (con coste de transporte y sin coste) para ver cuál obtiene soluciones con mejor calidad. Es necesario estimar las diferencias en cuanto al beneficio obtenido, la distancia total recorrida y el tiempo de ejecución.

Experimento a realizar:

Observación	Habrá un algoritmo mejor que otro
Planteamiento	Vamos a ejecutar el programa con los 2 algoritmos para las 2 heurísticas y sacar resultados
Hipótesis	Simulated Annealing es mejor que Hill Climbing
Método	- Fijamos el escenario con número de estaciones a 25, núm. total de bicis a 1250, núm. de furgonetas a 5, demanda equilibrada. - Utilizaremos las 2 heurísticas implementadas y los operadores elegidos en la 1a experimentación. - Los algoritmos serán Hill Climbing y Simulated Annealing con una K = 25 y una λ = 125. - Ejecutaremos 10 semillas diferentes para las diferentes generaciones de la solución inicial. - Mediremos el **beneficio obtenido**, el **tiempo de ejecución** y la **distancia total recorrida** para realizar la comparación.

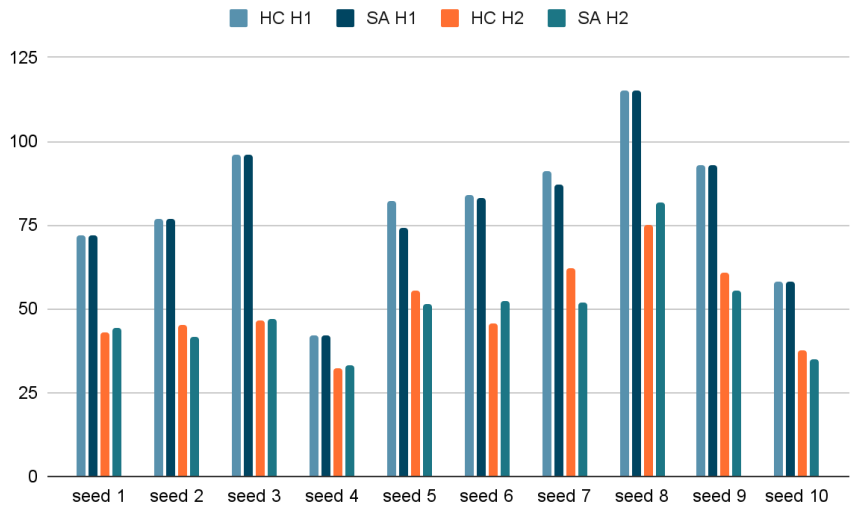

Figura 7: Beneficios obtenido de los algoritmos para cada heurística

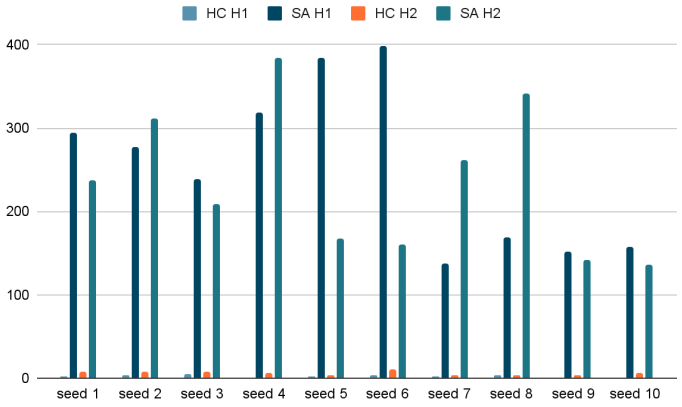

Figura 8: Tiempo de ejecución de los algoritmos para cada heurística

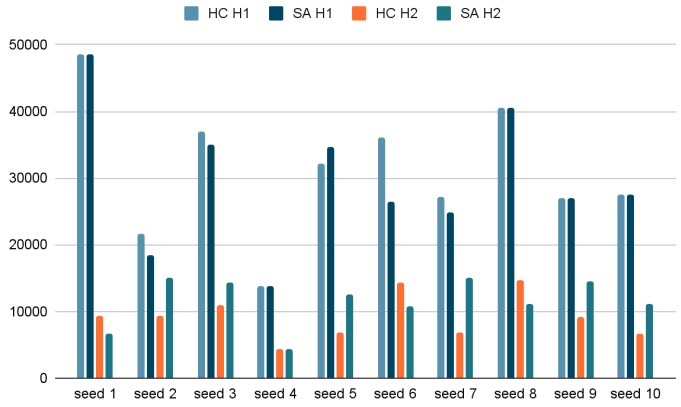

Figura 9: Distancias total recorrida por cada algoritmo con cada heurística

Basándonos en los resultados, Hill Climbing parece ser una opción más efectiva que Simulated Annealing, lo cual contradice nuestra hipótesis inicial. Inicialmente, creíamos que la capacidad de Simulated Annealing para explorar soluciones subóptimas antes de converger a mejores óptimos locales le daría una ventaja, pero en estos casos, esto no parece ser cierto.

A través de nuestras tablas, hemos observado que Hill Climbing ofrece un menor tiempo de ejecución y, en general, un mayor beneficio. Aunque no podemos afirmar con certeza que uno sea superior al otro, dado este experimento, optaremos por utilizar Hill Climbing en futuros experimentos.

Por otro lado, si queremos comparar las distintas funciones heurísticas y cómo influencia en el beneficio obtenido y el tiempo de ejecución, observamos que con la heurística 1, el beneficio obtenido es más alto que la heurística 2 por su naturaleza, no tenemos en cuenta el coste de transporte. En relación al tiempo de ejecución, también es menor ya que nuestra generación inicial greedy ya se acerca a un mínimo local sin necesidad de expandirse más allá de 3 nodos con la heurística 1.

8.6. Tiempo de ejecución según tipo de demanda

Tenemos que estudiar y determinar si existe alguna diferencia en el tiempo de ejecución para resolver el problema entre una demanda en hora punta y una demanda equilibrada.

Experimento a realizar:

Observación	Es posible que los tiempos de ejecución sean diferentes dependiendo del tipo de demanda del problema
Planteamiento	Ejecutar los programas con los 2 tipos de demanda y obtener los tiempos
Hipótesis	La demanda en hora punta tiene un tiempo de ejecución más alto
Método	- Fijaremos el escenario con número de estaciones a 25, núm. total de bicis a 1250, núm. de furgonetas 5, demanda a determinar y heurística que optimice el primer criterio (transporte gratis). - Utilizaremos los operadores elegidos en la 1a experimentación, inicialización elegida en la 2a experimentación, y el algoritmo

	elegido en la 3a experimentación. - Para cada tipo de demanda, ejecutaremos 10 veces con semillas diferentes. - Mediremos el **tiempo de ejecución** para realizar la comparación.

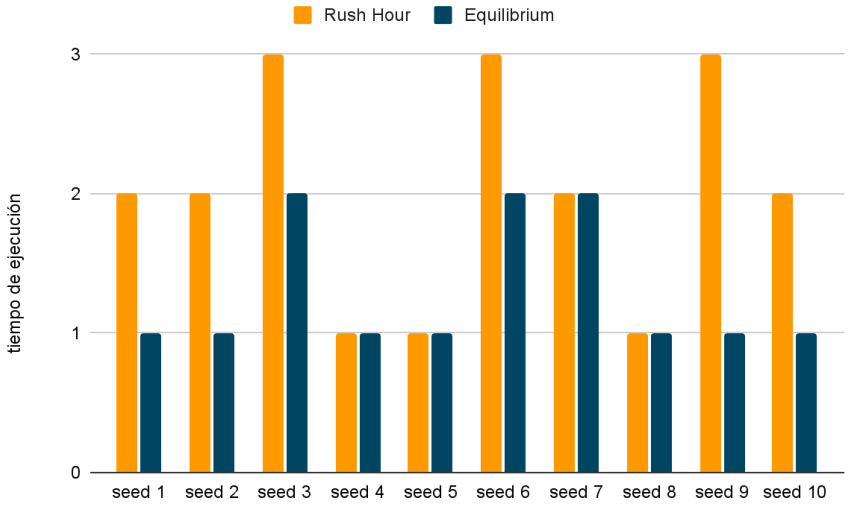

Figura 10: Tiempo de ejecución de HC con cada tipo de demanda

Los resultados indican claramente que el tiempo de ejecución es, en la mayoría de los casos, significativamente mayor durante la hora punta, como pensábamos. Suponemos que se debe a la presencia de situaciones extremas en las que algunas estaciones tienen un exceso notable de bicicletas, mientras que otras tienen una escasez significativa. Esto conlleva a un mayor número de expansiones del algoritmo en su búsqueda de nuevas soluciones y que con nuestros operadores, se satisfacen las condiciones de aplicabilidad con más probabilidad.

8.7. Número de furgonetas óptimo para cada tipo de demanda

Tenemos que determinar cuál es aproximadamente el número de furgonetas que son necesarias para obtener la mejor solución.

Experimento a realizar:

Observación	Hay un valor del número de furgonetas que nos permite obtener la solución óptima
Planteamiento	Ejecutar los programas con los 2 tipos de demanda e ir augmentando las furgonetas disponibles y obtener el beneficio obtenido
Hipótesis	El número de furgonetas que nos permite obtener la solución óptima es menor o igual que el número de estaciones El número de furgonetas que nos permite obtener la solución óptima en demanda punta es menor que en demanda equilibrada
Método	- Fijaremos el escenario con número de estaciones a 25, núm. total de bicis a 1250, núm. de furgonetas de 5 en 5 hasta llegar a 25

	número de estaciones, demanda a determinar y heurística que optimice el primer criterio (transporte gratis). - Utilizaremos los operadores elegidos en la 1a experimentación, inicialización elegida en la 2a experimentación, y el algoritmo elegido en la 3a experimentación. - La semilla será la misma durante todo el experimento.

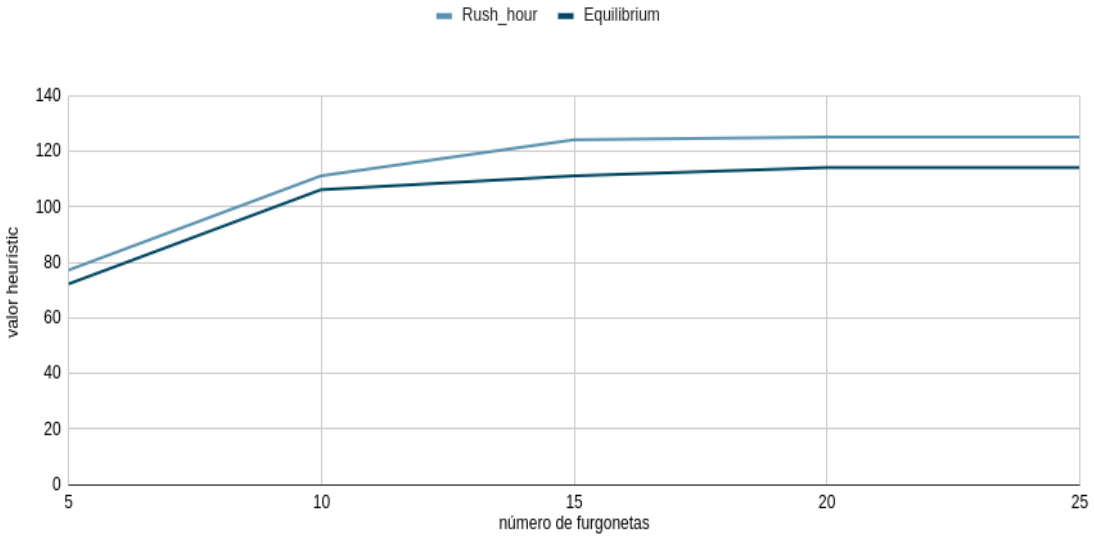

Figura 11: Beneficio para cada número de furgonetas con los tipos de demanda

El número de furgonetas que son necesarias para alcanzar el óptimo local parece que cumple con nuestra hipótesis. Como una estación solo puede tener una furgoneta como origen, como máximo podremos tener el número de estaciones como el número de furgonetas que nos sean útiles. E nnuestros experimentos, para demanda equilibrada este valor es de 20 y para la demanda de hora punta es de 15 en este escenario concreto.

Por otro lado, observamos que el número de furgonetas es diferente para cada tipo de demanda, siendo menor en hora punta como esperábamos en la hipótesis. Esto puede ser debido a que al tener las bicicletas concentradas en puntos concretos, habrá pocas estaciones que acumulen todas las bicis sobrantes, requiriendo de menos furgonetas para cubrirlas recordando que una estación solo puede tener una furgoneta como origen.

9. Conclusiones

Antes de embarcarnos en este proyecto, ninguno de los miembros de nuestro equipo tenía experiencia previa con el lenguaje de programación Java ni con las clases de AIMA. A lo largo de nuestra investigación, hemos aprendido mucho sobre este lenguaje, que es ampliamente utilizado en el campo de la inteligencia artificial y la programación orientada a objetos, y que también es relevante en otras asignaturas.

Después de completar nuestros experimentos y comparar los resultados con nuestras hipótesis, hemos confirmado que nuestras expectativas se han cumplido en general. Nuestra investigación ha arrojado una comprensión más profunda del funcionamiento de los algoritmos de búsqueda local, destacando la importancia de la elección de sus componentes.

En particular, hemos verificado la relevancia de la elección de la representación del estado, la generación del estado solución inicial, el conjunto de operadores utilizados y cómo cambios en la función heurística pueden influir significativamente en la eficacia de ciertos operadores. Además, hemos subrayado la importancia de definir adecuadamente la función heurística para abordar el problema de acuerdo con los criterios de optimización pertinentes.

Como posible futura experimentación, quedaría abierta la opción de trabajar con una generación de solución inicial que no se acercase tan rápidamente a un mínimo (o máximo) local, ya que nos tememos que por esta causa, el algoritmo de Hill Climbing nos haya dado resultados muy parecidos aunque un poco por encima del Simulated Annealing. En las clases de teoría, Hill Climbing siempre avanza hasta encontrarse con un mínimo local y por otra parte el Simulated Annealing va saltando de mínimos locales en mínimos locales y se queda con el que tenga un mejor resultado.

Incluso se podría considerar una generación estocástica/indeterminista para permitir reejecuciones del algoritmo Hill Climbing para llegar a diferentes mínimos locales y quedarnos con el mejor.

En resumen, nuestro trabajo nos ha proporcionado valiosas lecciones sobre la optimización de algoritmos de búsqueda local y sus componentes clave, y abre nuevas puertas para futuras investigaciones en este emocionante campo de la inteligencia artificial.

10. Trabajo de innovación

Para nuestro trabajo de innovación estábamos bastante indecisos entre dos temas, NVIDIA DLSS o WaveNet de Google DeepMind, ya que pese a que cada uno trata un campo distinto, ambos nos parecían muy interesantes, finalmente nos decantamos por NVIDIA DLSS ya que era un proyecto más reciente y nos llamó mucho mas la atención nada más empezar a investigar en él ya que está muy relacionado con una de las asignaturas que estamos cursando, Gràfics.

El DLSS (Deep learning super sampling) es una tecnología que se centra en el escalado de imágenes desarrollada por Nvidia y exclusiva de las tarjetas gráficas de Nvidia para uso en tiempo real en videojuegos, que utiliza aprendizaje profundo para escalar imágenes de menor resolución a una resolución más alta para visualizarlas en monitores de grandes resoluciones, utilizando redes neuronales entrenadas con miles de fotogramas de entrada con aliasing y su salida se juzga en comparación con los fotogramas acumulados "perfectos".

Para empezar con la búsqueda de información para realizar el trabajo nos dividimos un poco los 6 temas a consultar y entre todos fuimos buscando y encontrando distintos links y referencias en relación a:

- Descripción de las técnicas de IA que se han utilizado
- Descripción de cómo han sido utilizadas las técnicas
- Explicación de porqué es un producto/servicio innovador y la naturaleza de la innovación (innovación en la técnica/método de IA, uso innovador de técnicas ya existentes)
- Impacto de la innovación hecha al producto en la empresa (beneficios, riesgos, posición en el mercado)
- Impacto de la innovación hecha al producto en el usuario o en la sociedad (beneficios y riesgos)

Para el primer y el segundo punto encontramos un Programming_Guide_Release para desarrolladores en el repositorio de github de NVIDIA que pensamos que nos podría ser de gran ayuda:

https://github.com/NVIDIAGameWorks/NVIDIAImageScaling/blob/main/docs/RTX%20UI%20Developer%20Guidelines.pdf 23 de septiembre

Para estos apartados también encontramos un Technical Blog de NVIDIA en el que los desarrolladores responden a preguntas muy interesantes que han hecho distintos usuarios sobre el funcionamiento de las técnicas de IA que se han utilizado en el producto:

https://developer.nvidia.com/blog/dlss-what-does-it-mean-for-game-developers/ 23 de septiembre

También encontramos un documento con muchos aspectos técnicos de la versión 2.0 de DLSS que pensamos que nos sería útil para dar datos de las implementaciones del producto:

http://behindthepixels.io/assets/files/DLSS2.0.pdf 23 de septiembre

De cara a los tres últimos puntos hemos encontrado estos tres puntos que nos pueden ser muy útiles, los tres explican muy bien en qué consiste la innovadora tecnología que han implementado con ayuda

de la IA para mejorar la resolución de imágenes con redes neuronales entrenadas para evitar el aliasing.

https://developer.nvidia.com/rtx/dlss/get-started 21 de octubre
https://resources.nvidia.com/en-us-game-dev-dlss/deep-learning-super 21 de octubre
https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/ 21 de octubre

Y finalmente, este último link nos habla de las múltiples empresas que han empezado a utilizar esta nueva tecnología y por lo tanto nos podría ser útil para explicar los múltiples beneficios tanto económicos, como de visión e imagen de marca.

https://www.nvidia.com/en-us/geforce/news/june-2021-rtx-dlss-game-update/ 21 de octubre