Práctica de

Búsqueda Local

Intel·ligència artificial

2023/2024 Q1

Grau en Enginyeria Informàtica

Autores:

Sergi Pérez Escalante

Diego Velilla Recio

Pablo Buxó Hernando

Índice general

1. Implementación del estado

Para la implementación del estado, debemos cerciorarnos de que podemos representar cualquier posible solución y prestar atención al tamaño que puede llegar a ocupar en memoria cada instancia de la clase en la que lo representamos para no quedarnos sin memoria en las ejecuciones.

Hemos creado una clase con el nombre *PracticaBiciCiutat*. Cualquier posible solución que generemos será una instancia de esta clase. En primer lugar, hay que diferenciar entre la información que permanecerá igual en todas las soluciones de un mismo problema y las que varían a lo largo del espacio de soluciones.

En nuestro caso, hemos considerado que debemos mantener cuántas furgonetas hay en total y una instancia de la clase *Estacions*, que tiene información de cada una de las estaciones de un problema: su demanda, las bicicletas que habría y las que se pueden sacar si no se transportara ninguna en furgoneta. Los dos últimos valores podríamos irlos actualizando conforme vayamos añadiendo viajes de furgonetas, pero eso implicaría mantener esta estructura de datos para cada instancia y sería ineficiente porque no todos se actualizan. Además, valores como las bicicletas que habría en cada estación, si no hubiera viajes en furgoneta, los usaremos para la heurística, así que cambiarlos generaría más problemas.

Es por ello, que hemos decidido que tanto el número de furgonetas como la instancia de *Estacions* debe ser información estática en todas las instancias. Gracias a esto, podemos ahorrar una gran cantidad de memoria declarando estas variables como *static*.

En segundo lugar, tenemos que considerar aquella información que diferencia a una posible solución de otra y sí que hay que mantener para cada instancia. Lo que nos pide el problema son los viajes en furgoneta que transportan bicicletas. Cada viaje será identificado por su estación de origen, estación de primer destino, estación de segundo destino (a la que pondremos -1 si no existe), bicicletas dejadas en el primer destino y bicicletas dejadas en el segundo (serán 0 si no hay segundo destino). Esta información la podemos guardar en un

Array de enteros y tener un *ArrayList* que guarde para todas las furgonetas un *Array* como este.

Para no tener que guardar viajes de furgonetas que no hacen ninguno, solo se mantendrá en la lista los *Arrays* de furgonetas que realicen desplazamientos. Esto nos hará ahorrar memoria.

También hemos determinado que es importante mantener ciertos valores actualizados de cada estación, porque aunque se puedan deducir de los viajes en furgoneta, calcularlos cada vez que se necesiten sería demasiado costoso. Tenemos un *Array* en el que cada posición es una estación en la que hay un *Array* que contiene las bicicletas que habrá en esa estación, las que se podrán sacar y si sale alguna furgoneta de esa estación. Son valores que nos serán de gran utilidad cuando comprobemos si un operador se puede ejecutar o no según las restricciones y no tendremos la necesidad de recorrer todos los viajes para consultarlos.

2. Operadores de estado

Los operadores de estado son los que nos permiten modificar una solución de forma que lleguemos a otra que siga en el espacio de soluciones.

En concreto, para cada operador, explicaremos funcionalidad, sus condiciones de aplicabilidad y su factor de ramificación. Finalmente, se detallará por qué la combinación de estos operadores nos permite viajar por todo el espacio de soluciones. Es decir, se puede generar cualquier solución.

2.1 Añadir desplazamiento con 1 destino

Funcionalidad

A este operador se le pasan como parámetros dos estaciones (que llamaremos *e0* y *e1*) y un número de bicis, que llamaremos *b1*. Este operador añade un viaje a una furgoneta que no hacía ningún desplazamiento. Este viaje tendrá la estación *e0* como origen, la estación e1 como destino y deja *b1* bicis en esta última. Además, no tiene segundo destino y, por lo tanto, no deja bicis en ninguna segunda estación.

En la implementación del estado realizada, se añade un elemento al *ArrayList ViajeFurgo*. Este elemento será un *Array* que contiene la información del viaje: su origen (*e0*), primer destino (*e1*), segundo destino (será igual -1, al no tener ninguno), bicis dejadas en *e1* (*b1*), y bicis dejadas en segundo destino (en este caso, ninguna).

Además, se actualizarán diversos valores en el *Array ActEstacions*, que mantiene actualizados para cada estación las bicicletas que habrá, las que se podrán sacar y si sale alguna furgoneta de allí. En concreto, se actualiza para la estación *e0* las bicicletas que habrá y las que podrá sacar, a ambos valores se les restará *b1*, que son el número de bicicletas que se transportarán. Además, se pone a 1 el valor que indica si sale alguna furgoneta de esta. Por último, también se modifica el valor que indica la cantidad de bicicletas que habrá en *e1*, al que se le sumará *b1*.

Condiciones de aplicabilidad

Por otro lado, este operador tiene unas determinadas condiciones de aplicabilidad que una función distinta se encarga de comprobar: no puede salir ninguna furgoneta de *e0* (porque no pueden salir dos furgonetas de la misma estación), hay como mínimo b1 bicicletas que se pueden sacar de *e0*, hay alguna furgoneta libre, *b1* es menor a 30 (dado que la capacidad máxima de la furgoneta es 30) y *e0* es una estación distinta a *e1*.

Factor de ramificación

En caso peor, el factor de ramificación de este operador es E^2*31, teniendo en cuenta que *e0* y *e1* pueden ser cualquiera de las *E* estaciones existentes y que *b1* es un valor que va de 0 a 30. Aun así, la mayoría de los posibles sucesores que se generarían con este operador no se acabarán generando debido a las condiciones de aplicabilidad que lo restringen. Además, en la función de sucesores, *b1* solo tiene un rango de 1 a 10, porque existen operadores que añaden bicicletas y no es necesario añadir todas con un solo operador. Esto reduce el factor de ramificación.

Como aclaración, es importante que este operador pueda añadir bicicletas y no cree un desplazamiento que no transporte bicicletas (aunque luego se puedan añadir) porque *Hill Climbing* nunca escogería si no fuera así, al no mejorar su situación. El coste de los viajes es los kilómetros recorridos multiplicados por el número de bicicletas transportadas más 9 y esto dividido entre 10. Por lo tanto, un rango de 1 a 10 es suficiente para que, si un viaje puede llegar a generar beneficio, el algoritmo lo escoja y no se quede en un máximo local sin avanzar a una situación que sería mejor si la furgoneta transportara más bicicletas.

2.2 Añadir segundo destino a un desplazamiento

Funcionalidad

Este operador recibe como parámetro una furgoneta *f1*, una estación *e2* y un número de bicis *b2*. El operador añade un segundo destino a la furgoneta *f1*, que hacía un viaje sin segundo destino. Este segundo destino es la estación *e2* y a ella llegan *b2* bicicletas.

En la implementación realizada, se actualizan los valores de segundo destino y bicicletas dejadas en el segundo destino del *Array* de *ViajeFurgo* en la posición *f1*. Además, se

actualizan diversos valores de *ActEstacions*. Se le resta *b2* a las bicicletas que habrá y que se podrán mover del origen del desplazamiento que hace *f1* y se suma *b2* a las bicicletas que habrá en la estación *e2*.

Condiciones de aplicabilidad

Como condiciones de aplicabilidad, debemos asegurar nos que la furgoneta *f1* existe y hace un viaje, pero no tiene segundo destino. Además, la estación *e2* no puede ser igual al origen de este viaje ni a su primer destino, las bicicletas dejadas en el primer destino sumadas a *b2* no pueden ser más de 30 (que es el límite de capacidad de cada furgoneta). Por último, el número de bicicletas que se puedan sacar del origen del viaje debe ser mayor o igual a *b2*.

Factor de ramificación

El factor de ramificación es *F***E**31. Siendo *F* el número de furgonetas que hacen viajes y *E* el número de estaciones. No obstante, la mayoría de los posibles sucesores serán descartados por las otras condiciones de aplicabilidad y, además, solo escogemos valores de *b2* en un rango de 1 a 10, por motivos análogos a los del anterior operador explicado.

2.3 Modificar primer destino

Funcionalidad

Este operador recibe una furgoneta *f1* y una estación *e1*. La furgoneta *f1* existe y hace un viaje. Con este operador, cambiamos el primer destino del viaje de *f1* a la estación *e1*.

En la implementación realizada, cambiamos el primer destino del *Array* en la posición *f1* de del *ArrayList ViajeFurgo*. Además, se actualizan los siguientes valores de *ActEstacions*: restamos el número de bicicletas que se dejaban en el anterior primer destino de las bicicletas que habrá en esa estación (dado que ya no se viajará allí) y, simétricamente, añadimos ese número de bicicletas a la estación *e1* (porque ahora se trasladarán allí).

Condiciones de aplicabilidad

Hay diversas condiciones de aplicabilidad para este operador: la furgoneta *f1* hace un viaje (con esto también aseguramos que tendrá un primer destino) y *e1* es una estación distinta al origen del viaje de *f1*, al anterior primer destino y al segundo destino (si es que hay alguno).

Factor de ramificación

El factor de ramificación de este operador es *F***E*, siendo *F* el número de furgonetas que hacen viajes y *E* el número de estaciones. Sin embargo, algunos posibles sucesores serán descartados por las condiciones de aplicabilidad.

2.4 Modificar segundo destino

Funcionalidad

Este operador es muy similar al anterior porque recibe también una furgoneta *f1* y una estación *e2*. La furgoneta *f1* hace un viaje que tiene dos destinos y se modifica el segundo destino a *e2*.

En la implementación realizada, cambiamos el segundo destino del *Array* en la posición f1 del *ArrayList ViajeFurgo*. Además, actualizamos *ActEstacions*, restando las bicicletas que se dejaban en la segunda estación a las que habrá en esa estación y sumándoselas a *e2*, que pasará a ser el segundo destino.

Condiciones de aplicabilidad

Hay diversas condiciones de aplicabilidad para este operador: la furgoneta *f1* hace un viaje, ese viaje tiene un segundo destino y *e2* es una estación distinta al origen del viaje de *f1*, al primer destino y al anterior segundo destino.

Factor de ramificación

El factor de ramificación de este operador es *F***E*, siendo *F* el número de furgonetas que hacen viajes y *E* el número de estaciones. Sin embargo, muchos posibles sucesores serán descartados por las condiciones de aplicabilidad. Por ejemplo, todos aquellos que surjan de una furgoneta que hace un viaje sin segundo destino.

2.5 Añadir bicis del primer destino

Funcionalidad

Este operador recibe por parámetro el identificador de la furgoneta *f1* y una cantidad de bicis *b1*. En caso de poder aplicarse, la furgoneta *f1* dejará *b1* bicis más en su primer destino *e1*.

En la implementación realizada, aumentamos en *b1* tanto las bicis transportadas por *f1* del ArrayList ViajeFurgo como las bicis que habrá en la estación destino *e1* del vector ActEstacions. Aunque también tenemos que restar, en este último vector, *b1* a las bicis que habrá y las que se pueden sacar de la estación origen *e0*.

Condiciones de aplicabilidad

Hay diversas condiciones de aplicabilidad para este operador: la furgoneta *f1* hace un viaje, en la estación de origen *e0* hay por lo menos *b1* bicis disponibles y la suma de las bicis dejadas entre las dos estaciones de destino *e1* y *e2* tiene que ser menor o igual a 30.

Factor de ramificación

El factor de ramificación de este operador es *F*29*, siendo *F* el número de furgonetas que hacen viajes y *29* el número máximo de bicicletas que podemos añadir a un viaje ya existente. Sin embargo, muchos posibles sucesores serán descartados por las condiciones de aplicabilidad. Por ejemplo, todos aquellos que se pasen del límite de 30 bicis por viaje o los que intenten coger más bicis de las que hay disponibles en el origen.

2.6 Añadir bicis del segundo destino

Funcionalidad

Este operador recibe por parámetro el identificador de la furgoneta *f1* y una cantidad de bicis *b2*. En caso de poder aplicarse, la furgoneta *f1* dejará *b2* bicis más en su segundo destino *e2*.

En la implementación realizada, aumentamos en *b2* tanto las bicis transportadas por *f1* del *ArrayList ViajeFurgo* como las bicis que habrá en la estación destino *e2* del vector

ActEstacions. Aunque también tenemos que restar, en este último vector, *b2* a las bicis que habrá y las que se pueden sacar de la estación origen *e0*.

Condiciones de aplicabilidad

Hay diversas condiciones de aplicabilidad para este operador: la furgoneta *f1* hace un viaje, este viaje tiene un segundo destino e2, en la estación de origen *e0* hay por lo menos *b2* bicis disponibles y la suma de las bicis dejadas entre las dos estaciones de destino *e1* y *e2* tiene que ser menor o igual a 30.

Factor de ramificación

El factor de ramificación de este operador es *F*29*, siendo *F* el número de furgonetas que hacen viajes y *29* el número máximo de bicicletas que podemos añadir a un viaje ya existente. Sin embargo, muchos posibles sucesores serán descartados por las condiciones de aplicabilidad. Por ejemplo, todos aquellos que no hagan tengan segundo destino.

2.7 Quitar bicis del primer destino

Funcionalidad

Este operador recibe por parámetro el identificador de la furgoneta *f1* y una cantidad de bicis *b1*. En caso de poder aplicarse, la furgoneta *f1* dejará *b1* bicis menos en su primer destino *e1*.

En la implementación realizada, disminuimos en *b1* tanto las bicis transportadas por *f1* del *ArrayList ViajeFurgo* como las bicis que habrá en la estación destino *e1* del vector *ActEstacions*. Aunque también tenemos que sumar, en este último vector, *b1* a las bicis que habrá y las que se pueden sacar de la estación origen *e0*.

Condiciones de aplicabilidad

Hay diversas condiciones de aplicabilidad para este operador: la furgoneta *f1* hace un viaje y *b1* no puede ser mayor al número de bicis que dejabas anteriormente en el primer destino *e1*.

Factor de ramificación

El factor de ramificación de este operador es *F*29*, siendo *F* el número de furgonetas que hacen viajes y *29* el número máximo de bicicletas que podemos quitar a un viaje ya existente. Sin embargo, muchos posibles sucesores serán descartados por las condiciones de aplicabilidad. Por ejemplo, todos aquellos que intenten dejar menos bicis de las que tenían en un principio.

2.8 Quitar bicis del segundo destino

Funcionalidad

Este operador recibe por parámetro el identificador de la furgoneta *f1* y una cantidad de bicis *b2*. En caso de poder aplicarse, la furgoneta *f1* dejará *b2* bicis menos en su segundo destino *e2*.

En la implementación realizada, disminuimos en *b2* tanto las bicis transportadas por *f1* del *ArrayList ViajeFurgo* como las bicis que habrá en la estación destino *e2* del vector *ActEstacions*. Aunque también tenemos que sumar, en este último vector, *b2* a las bicis que habrá y las que se pueden sacar de la estación origen *e0*.

Condiciones de aplicabilidad

Hay diversas condiciones de aplicabilidad para este operador: la furgoneta *f1* hace un viaje, este viaje tiene un segundo destino *e2* y *b2* no puede ser mayor al número de bicis que dejabas anteriormente en el primer destino *e1*.

Factor de ramificación

El factor de ramificación de este operador es *F*29*, siendo *F* el número de furgonetas que hacen viajes y *29* el número máximo de bicicletas que podemos quitar a un viaje ya existente. Sin embargo, muchos posibles sucesores serán descartados por las condiciones de aplicabilidad. Por ejemplo, todos aquellos que intenten dejar menos bicis de las que tenían en un principio o los que no tengan un segundo destino *e2*.

2.9 Quitar desplazamiento

Funcionalidad

Este operador nos permite quitar alguno de los desplazamientos que se realicen. Se le pasa por parámetro una furgoneta *f1* y se elimina del viaje que efectuaba, de forma que todas las bicicletas que transportaba se quedan en la estación de origen.

En la implementación realizada, esto supone hacer un *remove* de la posición *f1* del *ArrayList* donde se guardan todos los viajes en furgoneta. Este operador, a diferencia de todos los demás, que tienen coste constante, tiene un coste O (n).

Condiciones de aplicabilidad

Para poder aplicarlo, debemos asegurarnos de que la furgoneta *f1* realmente hace un viaje.

Factor de ramificación

El factor de ramificación de este operador es F, siendo F el número de furgonetas que hacen viajes, ya que se puede quitar el desplazamiento de cualquiera de ellas.

2.10 Añadir primer destino

Funcionalidad

Existe el caso en el que se use un generador que asigna furgonetas a estaciones, haciendo que estas se conviertan en el origen de sus viajes, pero no efectúa ningún viaje. Es decir, estas furgonetas solo tienen origen, pero no poseen ni primer ni segundo destino.

Dado que no existía ningún operador para añadir un primer destino en el caso en el que el viaje en furgoneta exista, pero no tenga primer destino, se ha añadido este operador.

En la implementación realizada, este operador recibe una furgoneta f1, una estación e1 y un número de bicicletas b1. Posteriormente, asigna e1 al destino del *ArrayList ViajeFurgo* en la posición f1 y se actualiza *ActEstaciones* restando a las bicicletas que habrá en el origen y las que se podrán sacar b1 y sumando b1 a las bicicletas que habrá en la estación e1.

Condiciones de aplicabilidad

Para poder aplicarse, debe existir el viaje con furgoneta f1, no puede tener primer destino, b1 ha de ser menor a 30 (por la capacidad de la furgoneta) y en la estación de origen debe haber como mínimo b1 bicicletas que se puedan mover.

Factor de ramificación

El factor de ramificación es *F***E**31. Siendo *F* el número de furgonetas que hacen viajes y *E* el número de estaciones. No obstante, la mayoría de los posibles sucesores serán descartados por las otras condiciones de aplicabilidad y, además, solo escogemos valores de *b1* en un rango de 1 a 10, por motivos análogos a los del operador que añade segundo desplazamiento.

2.11 Espacio de soluciones

Gracias a estos operadores, se puede generar cualquier solución del espacio de soluciones. Dado que para cada furgoneta podemos tener un origen, dos destinos y dejar entre 0 y 30 en cada uno (sin sumar más de 30 entre las dos), el espacio de soluciones tiene tamaño $O[(E^3)*(30^2)]^F$. La parte más importante y que identifica a una solución son los viajes que hacen las furgonetas, puesto que las otras estructuras de datos utilizadas, son solo para tener la información actualizada y disponible de forma rápida.

Para ello, se pueden añadir desplazamientos que tendrán cualquier origen y primer destino. También se pueden añadir segundos destinos a estos desplazamientos. Además, a estos desplazamientos se les podrán añadir bicicletas para poder cubrir todas las posibilidades, teniendo en cuenta que al añadir desplazamientos, solo se sumarán hasta 10 bicicletas. En caso de comenzar con una estrategia de solución inicial que ya tenga viajes, se podrán eliminar esos viajes con el operador de quitar desplazamientos o reducir las bicicletas que transporta con el operador de quitar bicicletas. Los operadores que modifican destinos no aumentan el espacio de posibles soluciones por el que se puede viajar con nuestra implementación, pero sí ayuda a viajar de forma más rápida al realizar cambios que, de otra forma, requerirían muchos más pasos.

En el caso en que las furgonetas ya tengan asignada una estación, el operador de añadir primer destino, desbloquea la situación, permitiendo que haya primer y segundo destino, que se puedan modificar y añadir o quitar bicis de los mismos.

3. Estrategias para general la solución inicial

Para empezar a aplicar los operadores necesitaremos una primera solución inicial. Nosotros hemos creado dos tipos de soluciones iniciales para este problema. En ambos, generamos un *ArrayList* de las estaciones y un vector donde guardar los viajes que hará cada furgoneta. La primera será una solución muy simple para poder empezar a ejecutar el problema desde algún punto y para la segunda haremos cambios que permitan mejorar el rendimiento de la ejecución, ya que empezaremos desde un punto potencialmente mejor respecto a la primera estrategia.

3.2 Estrategia de generación simple

Como primera estrategia para generar la solución inicial hemos cogido una solución a la que no le aplicamos ningún cambio respecto a los datos de entrada. Es decir, no habrá viajes iniciales ni cambios en las estaciones. Tal y como nos lleguen las estaciones con sus números de bicis en cada una, así se quedarán en esta solución inicial. Es la solución inicial más simple posible, ya que no estás cambiando nada para acomodar esta solución inicial y por eso es la que potencialmente dará peores resultados. A cambio, como no hará falta hacer nada, no tendrá un coste añadido. Su coste es O(1), es constante.

3.2 Estrategia de generación compleja

Nuestra segunda estrategia para generar la solución inicial será más compleja y, previsiblemente, situará el estado inicial en una solución mejor. Esta consistirá en ordenar descendentemente todas las estaciones por las bicicletas que sobran en esa estación. Es decir, las bicicletas que habrá menos las que se demandan. Para cada estación de la primera mitad, comprobaremos si se puede mover alguna bicicleta y, si es el caso, moveremos un número aleatorio de bicicletas (sin pasarse de las que se pueden mover) a alguna de las estaciones de la segunda mitad, que también será escogida de manera aleatoria.

La lógica detrás de esta solución es que las estaciones en las que sobran bicicletas son las que deberían ser el origen de un desplazamiento hacia una estación en la que faltan para poder adecuar el número de bicicletas en cada estación a la demanda.

El coste de esta solución es ordenar todas las estaciones en una estructura de datos distinta (un *ArrayList*) y, posteriormente, añadir los nuevos desplazamientos a medida que se recorre esta estructura. Añadir desplazamientos y generar los números aleatorios se hace en tiempo constante, así que el coste es el siguiente: O (*E**log(*E*) + *E*) = **O (*E**log(*E*))**, siendo *E* el número de estaciones.

3.3 Estrategia de generación alternativa

La tercera y última estrategia de generación de solución inicial trata asignar desde el inicio cada furgoneta a una estación y no hacerlo mientras el algoritmo de Hill Climbing o Simulated Annealing se ejecuta. Para ello, se asignan las furgonetas a aquellas estaciones en las que sobran más bicicletas. Es decir, el resultado de las bicicletas que habrá menos las que se demandan es mayor. De esta forma, no solo se asignarán las furgonetas de forma aleatoria, sino que, además, se hará siguiendo un criterio que promoverá que aquellas estaciones con más bicicletas sobrantes sean el origen de un viaje que, potencialmente, acabará yendo a una estación más carente de bicicletas.

El coste de esta solución es ordenar todas las estaciones en una estructura de datos distinta (un *ArrayList*) y, posteriormente, añadir desplazamientos sin primer destino y sin bicicletas a mover a las primeras estaciones haciendo un recorrido por todas ellas. Por lo tanto, el coste es **O(E*log(E)**, al igual que para la anterior estrategia.

4. Funciones heurísticas

Para poder seleccionar el mejor sucesor a nuestra solución actual necesitaremos algún criterio a partir del cual puntuarlos. Para eso hemos creado dos heurísticos, el primero más simple para poder empezar a hacer ejecuciones y el segundo, generado a partir del primero, pero mejorándolo.

4.1 Primer heurístico

En este primer heurístico, solo tendremos en cuenta la cantidad de bicis en cada estación respecto a su demanda. Para ello simplemente calcularemos la diferencia entre las bicis que habrá en cada estación con su previsión. Solo hay dos casos en los que este cálculo cambia, si la demanda es menor a las bicis que habrá y si la previsión es mayor a la demanda, en ambos casos el problema se resuelve cambiando o las bicis que habrá o la previsión por la demanda. Luego sumamos el valor de cada estación y le cambiamos el signo para que de esta manera al minimizar la función estemos escogiendo la que tenga mayor puntuación.

4.2 Segundo heurístico

Con el segundo heurístico el único cambio que hemos realizado es contabilizar y tener en cuenta también los kilómetros recorridos en cada desplazamiento acorde a como nos lo presentan en el enunciado. Siguiendo la fórmula que nos indican calculamos el coste de cada recorrido, lo sumamos y encontramos el coste total. Luego simplemente calculamos el beneficio restando los ingresos que calculamos en el primer heurístico y el coste. Por el mismo motivo que antes el resultado tenemos que retornarlo cambiado de signo.

4.3 Tercer heurístico

En este heurístico, se intenta minimizar la distancia recorrida, de forma que se reduzca el coste que suponen los transportes. Dado que una aproximación tan sencilla implicaría que nunca se moverían las furgonetas porque, al hacerlo, se estará recorriendo distancia, se añadirá también un factor que dé beneficio cuantos más viajes se den. De esta forma, el código intentará recorrer poca distancia, pero teniendo en cuenta que dar un viaje aporta beneficio (contando que una furgoneta con dos destinos hace dos viajes).

En concreto, se suma la distancia recorrida en metros por la solución y se le resta 1000 multiplicado por los viajes que realiza. De esta forma, buscará hacer viajes en los que no se recorra más de 1 kilómetro. Cuanto menor sea este resultado, mejor será según el heurístico. Se hace de esta forma porque se minimiza.

5. Experimentación

5.1 Escoger conjunto de operadores

Planteamiento

Este experimento trata de determinar qué conjunto de operadores da mejores resultados. Estos fijarán los operadores para el resto de experimentos.

Hipótesis

Tenemos la certeza de que debemos poder añadir desplazamientos y añadir segundo destino (y añadir primer destino), porque, si no, el programa no avanzaría por el espacio de soluciones para el primer generador de solución inicial. Por otro lado, creemos que serán importantes los operadores de añadir bicis, para poder cubrir los 30 espacios que tenemos por furgoneta y los operadores de modificar destino, para redirigir viajes que abran paso a una mejor combinación, aunque den algo más de pasos.

Respecto a los operadores de quitar bicis, tenemos dudas de que sean determinantes, porque no deberían añadirse bicicletas que permitan que más adelante deban quitarse para obtener una mejor solución.

Por último, pensamos que el operador de quitar desplazamiento no será necesario, puesto que serán prácticamente nulas las ocasiones en que se añada un desplazamiento que resulte no dar beneficio, ni siquiera cambiando su destino y bicicletas desplazadas.

Método

Para realizar el experimento, usaremos el algoritmo de *Hill Climbing*, el primer heurístico, la estrategia para generar la solución inicial compleja (segunda) y un escenario con 25 estaciones, 1250 bicicletas, 5 furgonetas y la demanda equilibrada. Empezaremos con los dos operadores de añadir desplazamiento y añadir destino, que serán inevitablemente necesarios. Seguidamente, iremos pasando por los operadores de modificar destinos, añadir bicis, quitar bicis y quitar desplazamiento. En cada iteración, miraremos si añadirlos mejora los resultados notablemente. Y, si lo hace, los incluiremos para las siguientes iteraciones. Los operadores que consigan pasar por este proceso, serán los que mantendremos para los

siguientes experimentos. Para cada iteración, ejecutaremos el programa 10 *seeds* diferentes, para asegurarnos de que los resultados son significativos. Además, serán el mismo conjunto de *seeds* para cada iteración, con tal de preservar comparaciones que no dependan de la complejidad de estas. Para cada *seed*, se harán 3 ejecuciones. En cada ejecución se medirá tanto el beneficio como los pasos dados.

Resultados

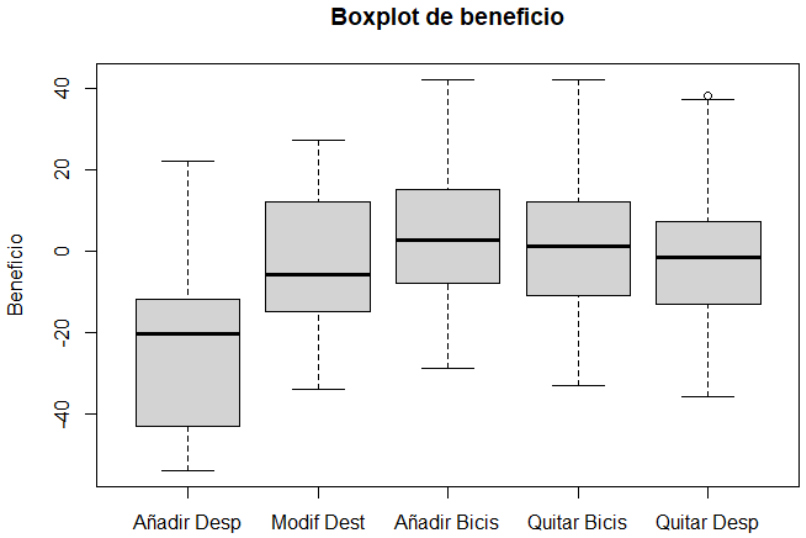
Para comprobar que los datos son significativos, usaremos el estimador *t-value*, que es un indicador que nos retorna el ratio entre la señal y el ruido. Cuanto más alto sea, más significativo será la variable sobre la que lo realicemos. Se considera que a partir de 5, la variable empieza a ser significativa, esto nos diría que la señal es 5 veces mayor que el ruido.

Para cada iteración, este es el *t-value* que nos ha dado. Se han utilizado 10 seeds distintas. Cabe mencionar, que se muestra el valor que recoge las ejecuciones de todas las *seeds*.

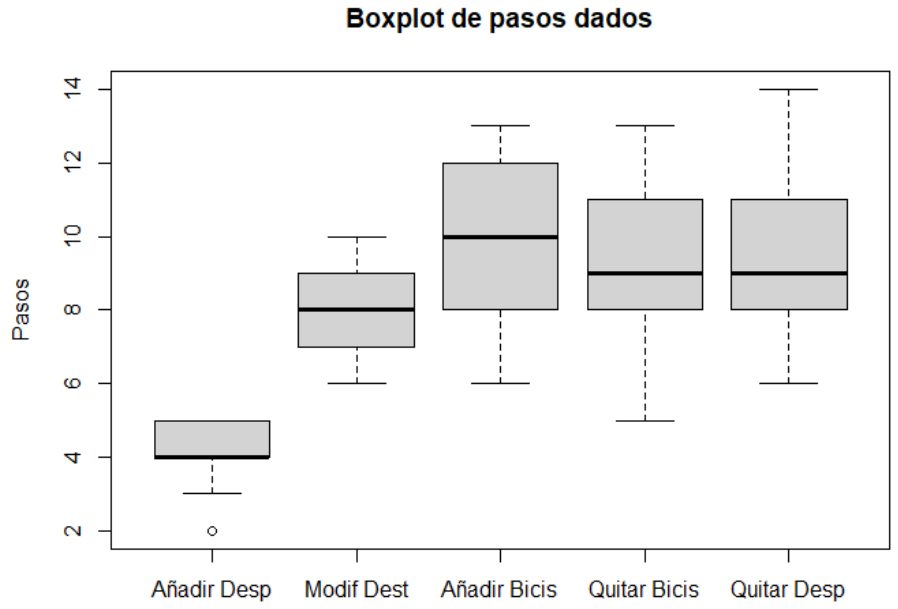
T-Value	Añadir Desp.	Modif. Dest.	Añadir Bicis	Quitar Bicis	Quitar Desp.
Beneficio	7.97	9.3755	8.320	10.074	9.319
Pasos	30.224	34.005	25.274	25.987	26.293

Como podemos ver, para cada iteración en la que comprobamos nuevos operadores, el *t-value* es claramente superior a 5, por lo que los resultados son significativos.

La tabla de los resultados para el beneficio y los pasos está en el **anexo**. A continuación, podemos observar el beneficio obtenido para cada iteración con sus respectivos *boxplot*. Aunque la función heurística minimiza, hemos preferido mostrar los resultados con valores positivos para mejorar la comprensión:



También hemos generado el *boxplot* para los pasos dados:



A partir de la segunda iteración, debíamos decidir si los operadores añadidos iban a mantenerse o no para las siguientes iteraciones, dependiendo de si mejoraban los resultados. Consideramos que el beneficio es el valor que más importante y los pasos dados pueden servir para desempatar o si existen grandes diferencias.

Para el caso de modificar desplazamiento, podemos observar claramente que los resultados mejoran. Concretamente, para la diferencia de beneficio entre tener los operadores de modificar desplazamiento y la primera iteración, tenemos un intervalo de confianza del 95% de (11.9181, 25.8819), por lo que es prácticamente seguro que el beneficio es mayor. Añadir desplazamiento tiene una media de -22.366 y modificar destino de-3.466 en beneficio. Para los pasos dados, añadir desplazamiento tiene una media de 4.2 y modificar desplazamientos de 7.83. Pese a que este último da más pasos, lo compensa con la mejora en el beneficio. Mantendremos estos operadores.

Para los operadores de añadir bicicletas, el beneficio también aumenta. Para la diferencia de beneficio entre esta tercera iteración y la anterior, tenemos un intervalo de confianza del 95% de (0.4842, 13.1828). También se puede asegurar con un alto grado de confianza que el beneficio es mayor. Añadir bicis tiene en beneficio una media de 3.36, superior a la anterior y una media de 9.8. Pese a que da algunos pasos más, lo compensa con una clara mejora en el beneficio. Por lo tanto, mantendremos estos operadores.

Para los operadores de quitar bicicletas, los resultados son más ajustados. Para la diferencia de beneficio entre esta iteración y la anterior, tenemos un intervalo de confianza del 95% de (-8.256, 4.123). En este caso, tendría sentido tanto considerar que mejora como que se mantiene estable. No obstante, el intervalo tiene una mayor parte negativa, por lo que hemos considerado que estos operadores no mejoran el beneficio. Con quitar bicicletas, se obtiene una media de 1.3. Para los pasos dados, se obtiene una media de 9.13, por lo que da algo menos de pasos en media respecto a la anterior iteración. Sin embargo, empeora el beneficio y no acaba compensando. No añadimos estos operadores.

Por último, para el operador de quitar desplazamiento, tenemos un intervalo de confianza del 95% de la diferencia entre el beneficio de esta iteración y la anterior de (-8.817,1.684). Este es un caso parecido al anterior, pero observamos que el intervalo tiene mayor parte negativa, así que lo más probable es que este operador no aporte beneficio. La media para al beneficio con este último operador es de 1.3. Para los pasos dados, la media es de 9.133. Por lo que es prácticamente igual a la anterior iteración sin que esto repercuta en una mejora en el beneficio. Debido a esto, tampoco se añaden este operador.

Conclusiones

En conclusión, para los 10 operadores que se consideran existen una gran cantidad de combinaciones posibles, pero teniendo en cuenta que, marginalmente, el operador de quitar desplazamiento y el de quitar bicis no ofrece resultados que nos induzcan a pensar que aporta beneficio, hemos optado por excluirlos para los experimentos posteriores. Además, hemos tenido en cuenta que quitar desplazamiento es el único operador con un coste lineal (y no constante).

Respecto a las hipótesis, hemos estado bastante acertados, porque los operadores de añadir desplazamientos, añadir bicicletas y modificar desplazamientos son realmente importantes. Finalmente, el operador de quitar bicis ha sido descartado (nuestras dudas sobre el mismo eran razonables) y el de quitar desplazamientos ha acabado siendo descartado, tal como razonamos. También pronosticamos un posible aumento en los pasos dados para los operadores de modificar destino y añadir bicis que ha sido confirmada, aunque no determinante.

5.2 Determinar mejor método de generación

Planteamiento

Este experimento trata de encontrar el mejor de los tres métodos de generación de solución inicial. El primero de ellos es vacío, es decir, empieza sin viajes iniciales. El segundo ordena las estaciones descendentemente por cantidad de bicis restantes y envía un número aleatorio de bicis desde la mitad de las estaciones con más bicis restantes hacia la mitad con menos. El tercero y último asigna a las furgonetas una estación de origen. El mejor será el utilizado en el resto de experimentos. Usaremos los operadores escogidos en el apartado anterior, misma heurística, escenario y *Hill Climbing*.

Hipótesis

Nuestra hipótesis es que la segunda generación será la más beneficiosa y que también será la más costosa de hacer.

Método

Para llevar a cabo el experimento lo que haremos será ejecutar 10 *seeds* distintas y aleatorias para cada generación y tomaremos el beneficio obtenido y su tiempo de ejecución. Luego comprobaremos la representatividad de los datos utilizando el *t-value* y los compararemos en gráficas para ver cuál es el mejor de los tres.

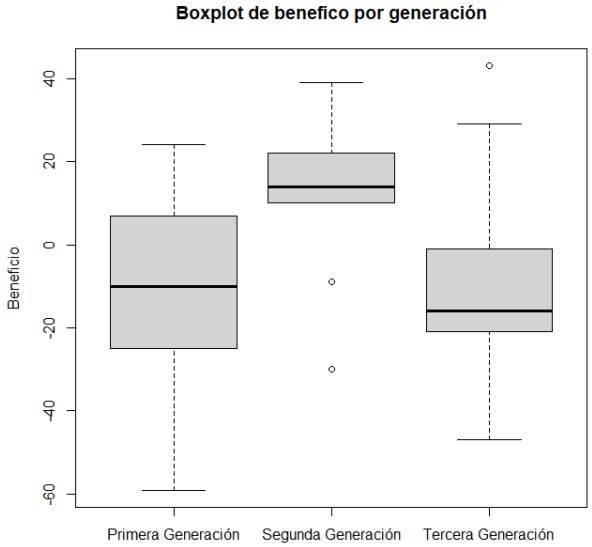
Resultados

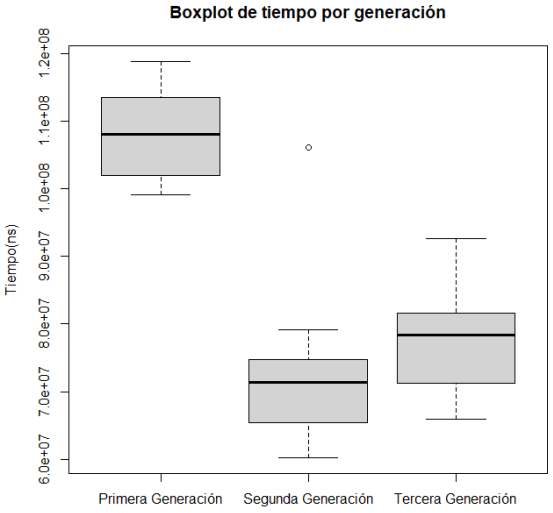
Primero, como hemos comentado anteriormente, comprobaremos que los datos obtenidos son significativos con el *t-value*.

	Beneficio	Tiempo
1ª generación	24.694	47.224
2ª generación	6.8202	18.522
3ª generación	9.6073	29.572

Como podemos observar, el *t-value* para todos los datos es grande, por lo tanto, podemos tomar esta muestra, ya que representa bien la realidad.

A continuación veremos las gráficas obtenidas de conseguir el beneficio y los pasos necesarios de cada una de las tres generaciones distintas.





Como podemos ver en el gráfico de la izquierda, el del beneficio, el segundo método de inicialización es el claro ganador. No solo tiene la media más alta, sino que es el que tiene la varianza más baja. En cuanto a beneficio, el segundo es el mejor de los tres. Mirando ahora el gráfico de la derecha podemos ver que tanto la segunda como la tercera generación son las que tienen el tiempo de ejecución más bajo, dejando la primera generación muy atrás.

Conclusión

Como hemos podido observar, nuestra primera hipótesis era cierta ya que la segunda generación es la que agiliza más el problema y la que obtiene un beneficio mayor. En cuanto a la segunda hipótesis, era normal pensar que el tiempo de ejecución iba a ser mayor o por lo menos iba a estar más cercano ya que esta solución requiere ordenar todas las estaciones y hacer viajes iniciales. El problema es que este tiempo que tarda en ordenar luego se lo ahorra en la ejecución de lo que queda de programa y por lo tanto es la más rápida de todas. Para el resto de experimentos utilizaremos la segunda generación de solución inicial.

5.2.5 Determinar la mejor función heurística (extra)

Planteamiento

Este es un experimento extra que realizamos para poder determinar la función heurística que se usará en los próximos experimentos. Deberemos decidir entre las 3 que se han explicado en su respectivo apartado.

Hipótesis

Creemos que la mejor heurística, en cuanto a beneficio, será la segunda, que intenta obtener el máximo beneficio posible. Es decir, considera tanto el beneficio por suplir cierta demanda como el coste de los traslados. En segundo lugar, pensamos que se situará la tercera heurística que, pese a no tener en cuenta el beneficio que aporta si intenta recorrer la menor distancia posible, algo que puede aumentar drásticamente el coste. En tercer lugar, consideramos que se posicionará la primera heurística, la cual considera los transportes gratuitos. Esto puede ser muy problemático, ya que puede elevar el coste de los traslados sin límite.

En relación con los pasos dados, no creemos que vayan a existir grandes diferencias entre las diferentes funciones heurísticas.

Método

Para realizar este experimento, se usará el escenario con 1250 bicicletas, 25 estaciones, 5 furgonetas, demanda equilibrada y segundo generador de solución inicial. Emplearemos el algoritmo de *Hill Climbing* y, se generarán 10 *seeds* aleatorias. Para cada una de ellas, se hará una ejecución con cada función heurística, para poder observar las diferencias entre ellas. En cada ejecución se extraerán los datos del beneficio obtenido y los pasos dados.

Resultados

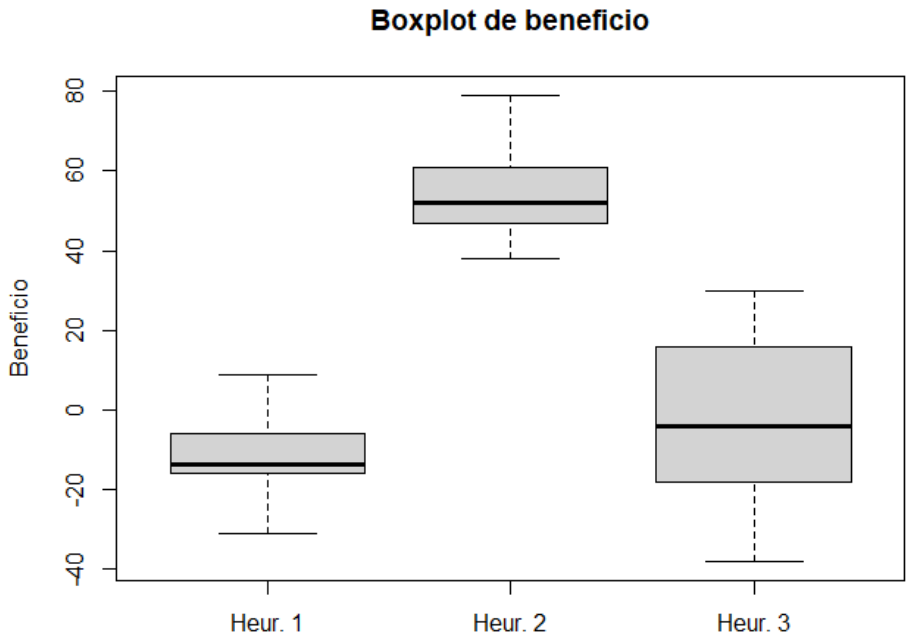
Para empezar, se corroborará que los resultados encontrados son significativos. A continuación se muestra la tabla que contiene los valores del estimador *t-value* en cada uno de los casos estudiados:

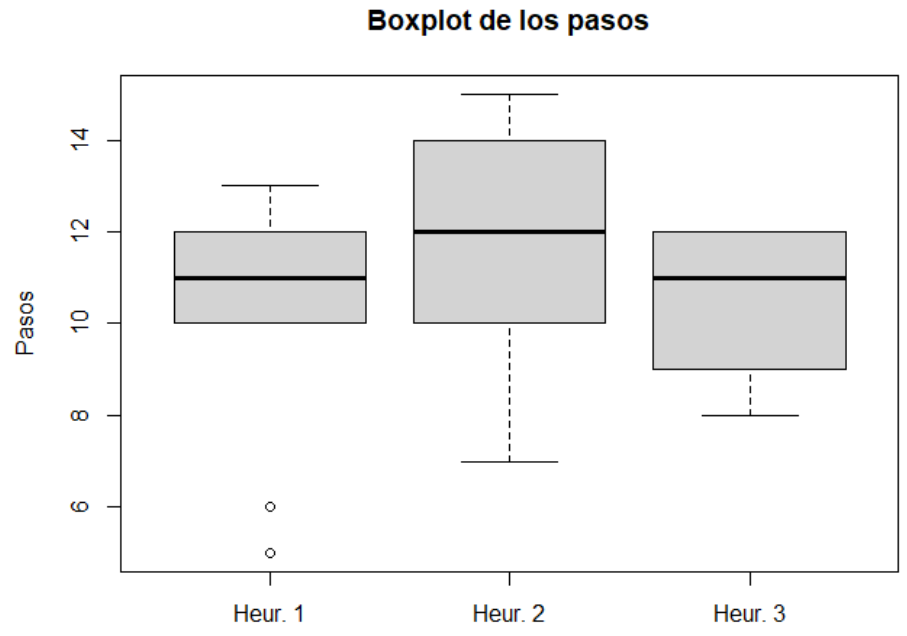
T-value	Heurística 1	Heurística 2	Heurística 3
Beneficio	5.295	15.03	5.284
Pasos dados	12.134	14.819	23.159

Como se puede observar, todos los valores se sitúan por encima del 5. Por lo tanto, son significativos. Seguidamente, se muestra una tabla con todos los valores recogidos en este experimento.

Heurística 1		Heurística 2		Heurística 3	
Beneficio	Pasos	Beneficio	Pasos	Beneficio	Pasos
-31	12	61	12	-38	12
-15	5	38	14	-15	10
-16	11	52	12	16	12
7	11	61	14	18	9
9	13	79	11	30	11
-13	12	62	15	-21	12
-14	10	50	10	5	11
-20	10	52	13	1	11
-11	12	45	9	-9	9
-6	6	47	7	-18	8

Seguidamente también se mostrará el boxplot del beneficio y los pasos dados, para visualizar los datos de forma más clara:





En relación con el beneficio, se puede observar que la segunda heurística es la que mejor funciona. En concreto, la media del beneficio de las heurísticas es la siguiente -11, 54.7 y -3.1 para la primera, la segunda y la tercera heurística respectivamente. Además, el intervalo de confianza del 95% de la diferencia de la segunda heurística y la tercera (la que más se le acerca) es de (43.88, 71.71), por lo que se puede asegurar con muy alta confianza que la segunda función heurística tiene más beneficio.

En relación con los pasos dados, obtenemos una media de 10.2, 11.7 y 10.5 para la primera, la segunda y la tercera heurística respectivamente. Son diferencias escasas y, si observamos el intervalo de confianza del 95% de la diferencia de la segunda heurística y la tercera, nos da el siguiente resultado: (-0.337, 2.737). Por lo que, pese a que la media de pasos para la segunda heurística es mayor, ni siquiera se puede asegurar que la esperanza lo sea con rotundidad, ya que el intervalo cubre valores positivos y negativos.

Conclusiones

En relación con el beneficio, se obtiene una aplastante ventaja para la segunda heurística. Con esto debería ser suficiente para decantarnos por esta, porque es el valor que consideramos más relevante. No obstante, además de esto, nos encontramos que las diferencias en los pasos dados no son muy grandes. Consecuentemente, el pequeño aumento de pasos que se produce al usar la segunda heurística es compensado con creces por el beneficio que aporta.

En cuanto a las hipótesis, hemos acertado en el orden que mantendrían para el beneficio y que las diferencias en los pasos dados no serían muy grandes.

En conclusión, la segunda heurística es la que se empleará siempre que se nos pida la mejor heurística.

5.3 Escoger parámetros para el *Simulated Annealing*

Planteamiento

Este experimento trata de encontrar los mejores parámetros para el *Simulated Annealing*, que serán usados en los experimentos posteriores.

Hipótesis

Como hipótesis, creemos que el mejor *stiter* será 10, ya que será importante no estar demasiadas iteraciones por cada paso. Para la *k*, creemos que el mejor valor será 10. No queremos una temperatura excesivamente alta, pero sí suficiente como para no estancarnos en máximos locales. Para *lambda*, suponemos que el valor que aportará mayor beneficio será 0.1, porque es importante que se reduzca gradualmente para no comportarse de manera similar al *Hill Climbing* y perder aleatoriedad.

En cuanto al tiempo de ejecución, no pensamos que vaya a ser determinante en ningún caso, no habrá grandes diferencias.

Método

Para el algoritmo de *Simulated Annealing*, tenemos 4 parámetros. El primero de ellos es *steps*, la cantidad de pasos que va a dar el algoritmo hasta pararse y arrojar el resultado que haya alcanzado en ese momento. Consideramos que cuantos más pasos se den, mejor serán los resultados a los que llegue el algoritmo. Por lo tanto, no necesitaremos observar cómo evoluciona este parámetro, porque siempre tenderá a dar mayor beneficio cuantos más pasos dé. Es por ello que se ha escogido un número grande de pasos (4000) y serán los que se usen para este experimento.

Para el resto de parámetros, sí debemos considerar diversas posibilidades, puesto que no hay una noción clara de qué valores resultarán en mejores resultados. Si, por ejemplo, consideráramos 20 posibles valores para cada variable, tendríamos 8000 combinaciones posibles, que resultarían en un aproximado de 80.000 ejecuciones (10 *seeds* por cada una de ellas). Dado que este es un número muy elevado que, además, dificultaría el proceso a seguir para mostrar los resultados, se ha considerado un método alternativo.

En este método, tendremos un escenario con 25 estaciones, 1250 bicicletas, 5 furgonetas, demanda equilibrada, estrategia de generación de solución inicial compleja y el segundo heurístico. Partiremos de unos parámetros iniciales *k* y *lambda* que son lógicos: 10 y 0.05 respectivamente. Para empezar, miraremos cuál es el *stiter* que da mejores resultados teniendo esa *k* y *lambda*. Una vez lo hayamos determinado, mantendremos ese *stiter* y comprobaremos diversos valores de *k* hasta hallar cuál es el que da mejores resultados, manteniendo la *lambda*. Por último, una vez tengamos el *stiter* y la *k*, repetiremos el proceso con diversos valores para *lambda* hasta quedarnos con el mejor.

Con este método, conseguiremos mejorar los parámetros iniciales y encontrar 3 valores nuevos que sabremos que funcionarán correctamente para *Simulated Annealing*.

Para el *stiter*, se comprobarán valores de 10 en 10 desde 10 hasta 100. Para la *k*, se calculará con valores de 5 en 5 hasta desde 5 hasta 50. Para lambda, se experimentará con valores de 0.1 en 0.1 desde 0.1 hasta 1. Para cada uno de los valores comprobados de cada parámetro, se comprobará el resultado en 10 *seeds* generadas aleatoriamente. En cada parámetro, las 10 *seeds* serán las mismas para que la variabilidad del escenario que den, no afecte a los resultados. Para cada ejecución se medirá tanto el beneficio como el tiempo de ejecución.

Resultados

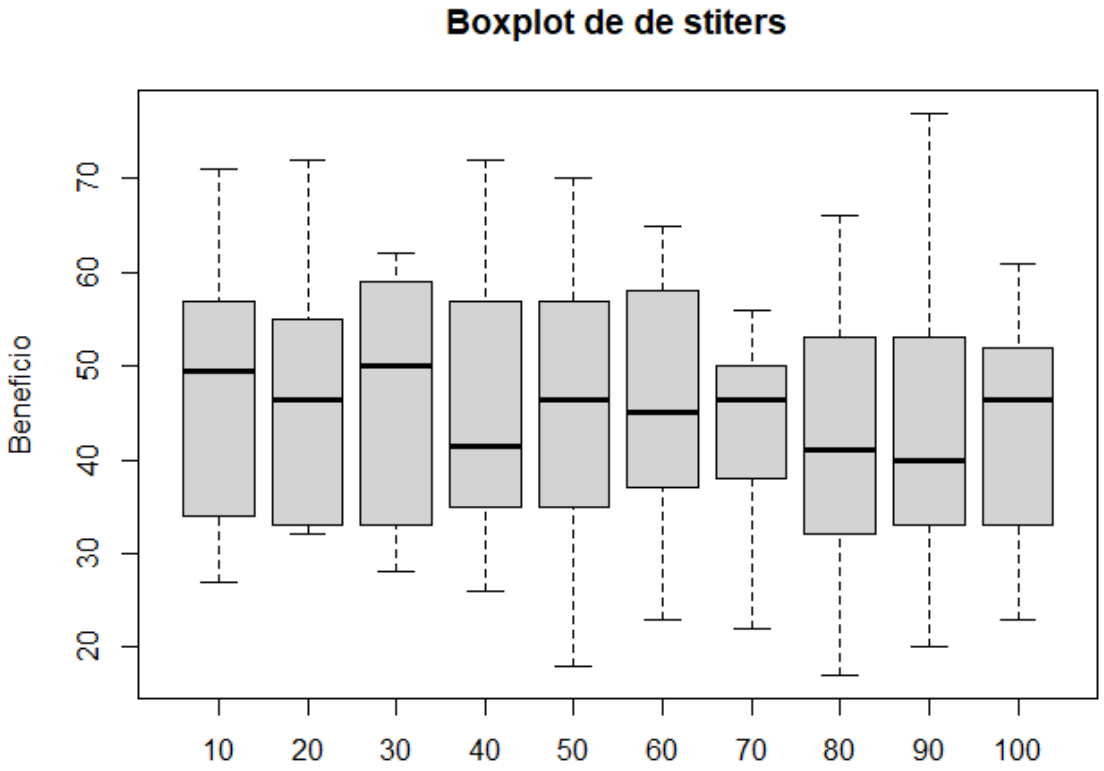
Para empezar, es importante aclarar que los resultados son significativos. En la siguiente tabla se pueden observar los *t-value* de cada combinación comprobada. Cabe aclarar que "B" y "T" indican si se trata de la medición de tiempo o beneficio y recordar que para cada variable se miden los resultados con 10 parámetros, indicados ya en el método.

	Parám. 1	Parám. 2	Parám. 3	Parám. 4	Parám. 5	Parám. 6	Parám. 7	Parám. 8	Parám. 9	Parám. 10
stiterB	10.37	11.20	11.375	9.10	9.49	10.129	12.962	9.281	8.23	10.286
stiterT	1405.6	336.99	333.93	515.42	477.05	201.66	196.76	269.11	219.99	315.2
kB	9.76	9.96	10.836	9.063	9.575	11.21	12.039	12.611	11.64	9.484
kT	617.96	284.18	197.15	443.88	452.19	523.83	264.94	333.03	293.82	404.66
lamdaB	11.76	10.09	12.51	9.75	9.12	9.45	10.76	10.123	10.526	10.381
lambdT	471.08	316.9	858.53	294.97	417.46	215.6	363.68	618.76	340.55	566.77

Como podemos ver, todos los *t-value* se sitúan por encima de 5. Esto nos indica que son significativos.

Dada la cantidad de datos que se manejan en este experimento, hemos preferido añadir un archivo a la carpeta del proyecto llamado “Resultados del experimento 3” en la que figuran todos los resultados.

A continuación, podemos ver el *boxplot* que incluye el beneficio obtenido en las diferentes ejecuciones, separados por cada valor de *stiter*. De esta forma, se puede analizar cuál es el que ofrece mejores resultados:

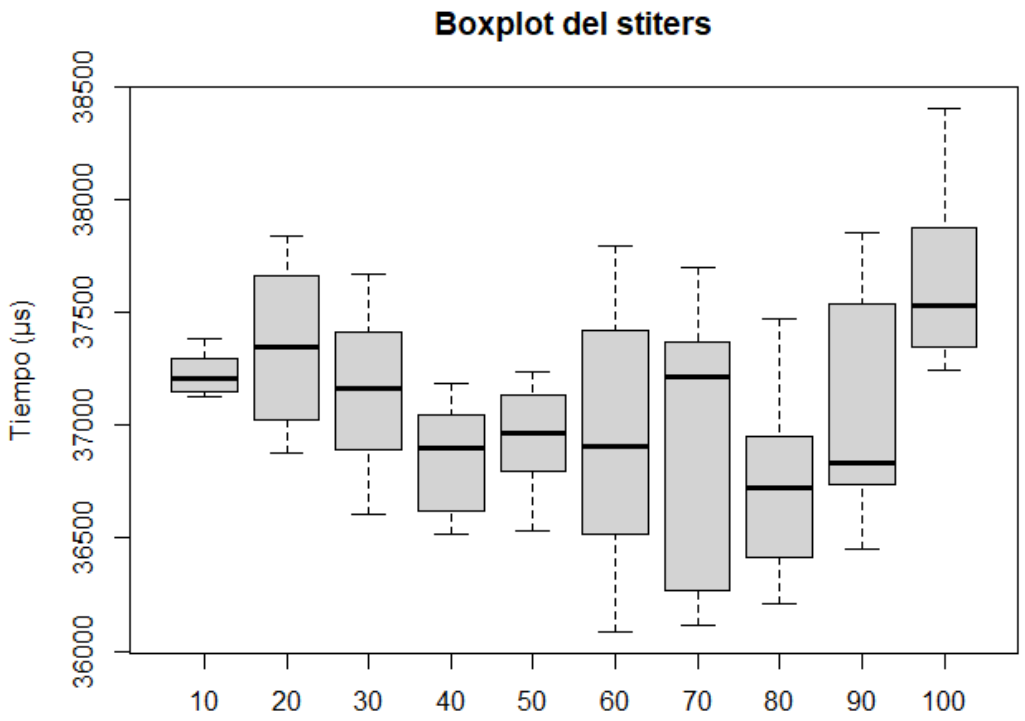


Aunque no existen grandes diferencias entre los valores probados, sí es apreciable y ha sido calculado que el valor 30 es el que mayor media tiene.

Las medias son las siguientes:

	Parám. 1	Parám. 2	Parám. 3	Parám. 4	Parám. 5	Parám. 6	Parám. 7	Parám. 8	Parám. 9	Parám. 10
stiterB	46.6	46.4	47.2	44.8	45.5	45.7	43.8	41.8	43.3	43.2

A continuación, mostramos el *boxplot* para el tiempo ejecutado con los diferentes parámetros probados para *stiters* en (µs).

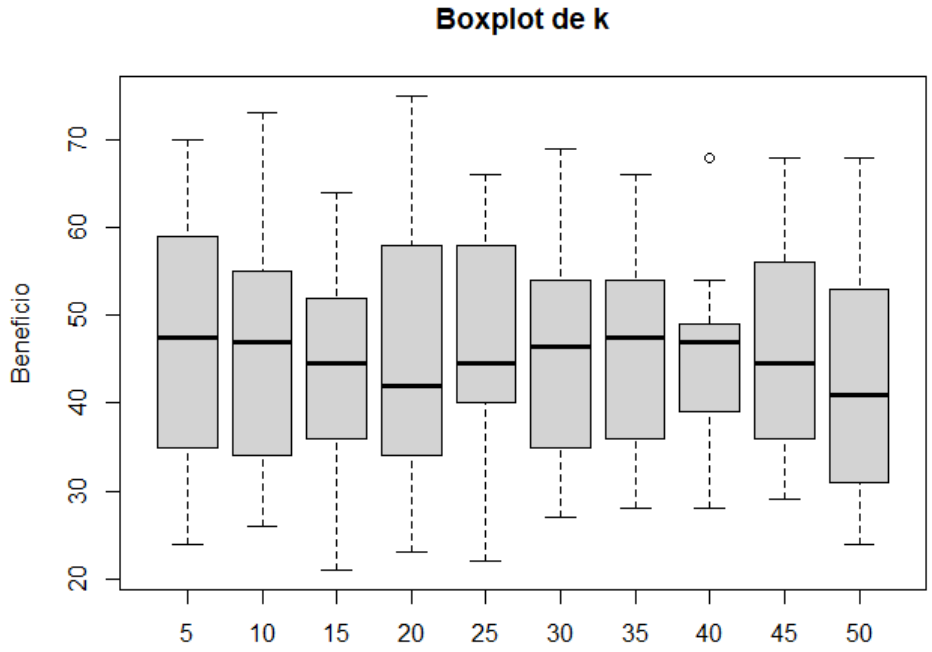


Además, las medias del tiempo ejecutado son las siguientes:

	Parám. 1	Parám. 2	Parám. 3	Parám. 4	Parám. 5	Parám. 6	Parám. 7	Parám. 8	Parám. 9	Parám. 10
stiterT	37226.8	37333.9	37147.9	36862.6	36927.1	36968.6	36988.5	36776.4	37072.2	37642.2

El parámetro 3 corresponde al valor igual a 30. Podemos ver que es el 4 que más tiempo tarda en ejecutarse en media. Además, dado que las diferencias entre todos son escasas (pues están comprendidos en un rango pequeño respecto a su valor), el beneficio que nos aporta compensa el tiempo que tarda. Así que *stiter* = 30 es el escogido de ahora en adelante.

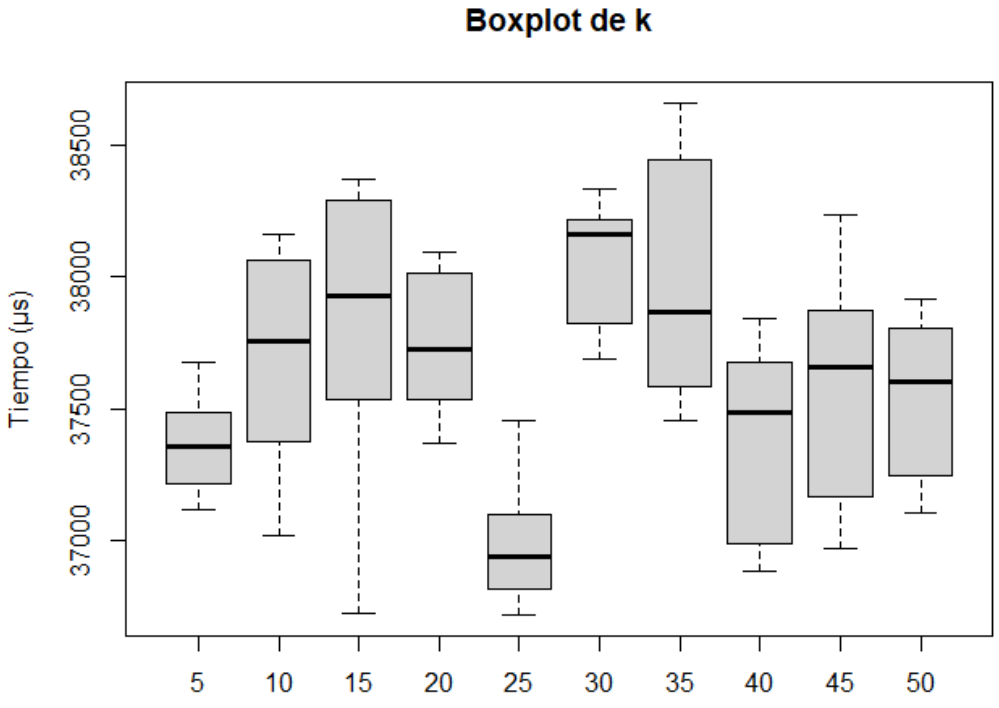
Seguidamente, podemos ver el *boxplot* del parámetro *k* con el beneficio obtenido en las diferentes ejecuciones, separados por cada valor probado:



Aunque no existen grandes diferencias entre los valores con los que se ha experimentado, el que tiene una mejor esperanza estimada es 5, con una esperanza estimada de 47.5. A continuación, mostramos la tabla con la media de los resultados obtenidos.

	Parám. 1	Parám. 2	Parám. 3	Parám. 4	Parám. 5	Parám. 6	Parám. 7	Parám. 8	Parám. 9	Parám. 10
kB	47.5	46	44.3	44.3	45.3	45.2	45.9	45.4	45.9	43

En relación con el tiempo de ejecución en µs, el *boxplot* que nos ofrecen los resultados recopilados es el siguiente:

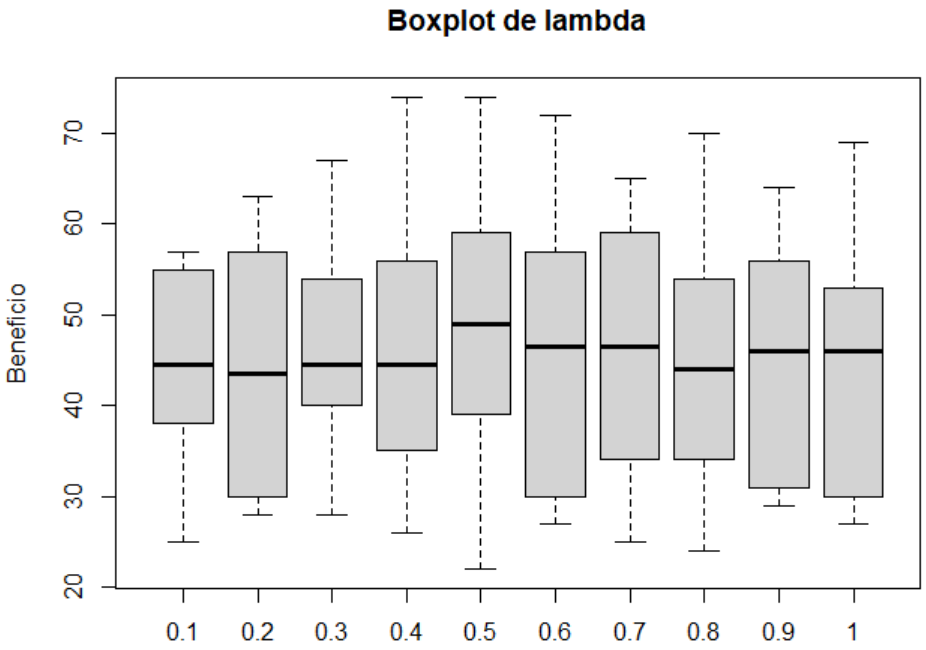


En este caso, podemos observar valores algo más dispares entre sí, pero siguen manteniéndose dentro de un rango pequeño respecto a los valores. Las medias de los parámetros del tiempo de ejecución en µs son las siguientes.

	Parám. 1	Parám. 2	Parám. 3	Parám. 4	Parám. 5	Parám. 6	Parám. 7	Parám. 8	Parám. 9	Parám. 10
kT	37369.4	37672	37790.9	37761.5	37008.5	38069	37983.3	37388.6	37589.5	37565.5

El valor que mayor beneficio aportaba en media era el 5. Este corresponde al parámetro 1 y es el segundo que menos tiempo tarda en ejecutarse. Esto nos indica que funciona bien en ambas variables. Por lo tanto, el valor escogido es $k = 5$

Por último, analizamos el parámetro *lambda*. A continuación, se puede observar el *boxplot* con los 10 valores distintos con los que se ha experimentado para el beneficio:

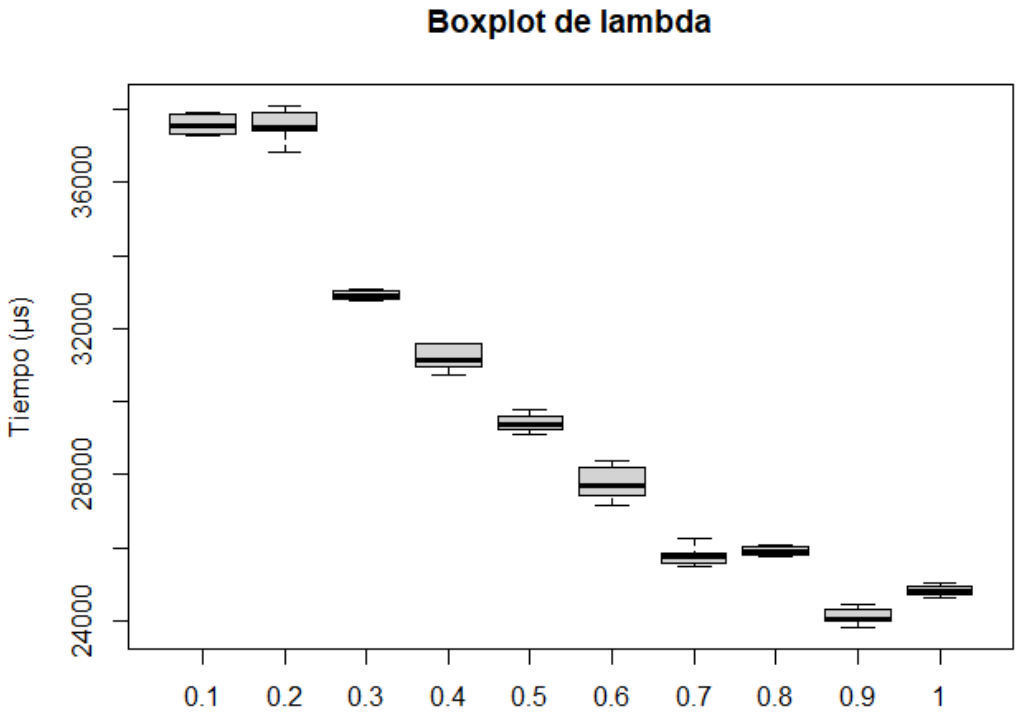


No se observan grandes diferencias. Seguidamente, enseñamos la tabla con la media de los resultados obtenidos.

	Parám. 1	Parám. 2	Parám. 3	Parám. 4	Parám. 5	Parám. 6	Parám. 7	Parám. 8	Parám. 9	Parám. 10
lambdaB	43.4	43.5	46.7	46	47.4	46	45.5	45.7	44.5	44.4

Esto nos indica que el valor con mayor media es el parámetro 5, que corresponde a *lambda* igual a 0.5.

Seguidamente, hemos generado el *boxplot* del tiempo de ejecución con los valores estudiados en µs.



Esta tabla muestra unos resultados con una clara tendencia a disminuir conforme la *lambda* aumenta. Sabemos que *lambda* controla cómo de rápido desciende la probabilidad de aceptar peores soluciones. Si esta lambda es grande, la probabilidad desciende tanto que es será improbable que se cojan sucesores peores y, en consecuencia, el algoritmo será más rápido, puesto que ese caso es más complejo. A continuación, se muestran las medias obtenidas del tiempo de ejecución en µs:

	Parám. 1	Parám. 2	Parám. 3	Parám. 4	Parám. 5	Parám. 6	Parám. 7	Parám. 8	Parám. 9	Parám. 10
lambdaT	37555.4	37582.8	32909.7	31204.3	29418.9	27788.6	25769	25910.1	24121.6	24834.3

El parámetro con valor 0.5 es el 5 que más tarda. Dado que el beneficio es considerada la variable más importante y que no tiene un tiempo de ejecución que se sitúe entre los más elevados, esta es el valor escogido. El valor escogido es *lambda* = 0.5

Conclusiones

En conclusión, después de comprobar que los resultados eran significativos y un análisis de los mismos, hemos visto que los parámetros que mejor funcionan, siguiendo el método explicado, son los siguientes: *stiter* = 30, *k* = 5 y *lambda* = 0.5.

Respecto a las hipótesis de *stiter* = 10, *k* = 10 y *lambda* = 0.1. Tenemos una diferencia en el beneficio entre el valor escogido y la hipótesis para *stiter* de (-5.094968, 6.294968), con un intervalo de confianza del 95%. No es descartable al 100% que *stiter* = 10 sea mejor, pero el intervalo es mayormente positivo.

Para *k*, tenemos una diferencia entre el valor escogido y la hipótesis de (-6.898465, 9.898465) con un intervalo de confianza del 95% para el beneficio. Similarmente, no es descartable al 100% que *k* = 10 sea mejor, pero es improbable, porque el intervalo tiene mayor parte positiva.

Para *lambda*, tenemos una diferencia entre el valor escogido y la hipótesis de (-1.213333, 9.213333) con un intervalo de confianza del 95% del beneficio. De forma análoga, no es descartable al 100% que *lambda* = 0.1 sea mejor, pero es altamente improbable.

En relación con los tiempos de ejecución, nuestra hipótesis ha sido correcta para el *stiter* y la *k*, pero completamente errónea para *lambda*, variable con la cual el tiempo de ejecución cambiaba notablemente.

5.4 Evolución del tiempo en función de las estaciones

Planteamiento

En este experimento se trata de estudiar cómo evoluciona la duración del programa, tanto en pasos, como en tiempo, en función del número de estaciones.

Hipótesis

Como hipótesis, creemos que tanto el tiempo como el número de pasos crecerán exponencialmente en función del número de estaciones.

Método

Para realizar este experimento, generamos 10 seeds aleatorias y ejecutamos 1 vez cada una para cada cantidad de estaciones. Las estaciones empezaran siendo 25 y aumentaran de 25 en 25. Primero veremos como evolucionan los pasos y después el tiempo de ejecución. En el caso de los pasos, hemos ido aumentando el número de estaciones hasta ver claramente cómo evoluciona. Para el tiempo de ejecución hemos seguido el mismo procedimiento. Hemos usado el método de Hill Climbing, la segunda generación de viajes y el segundo heurístico.

Una vez tengamos los datos de todas las ejecuciones, comprobaremos que los datos recopilados sean representativos usando el t-value y graficamos estos mismos para ver fácilmente qué evolución presentan.

Para medir el tiempo, hemos visto que si medimos el tiempo de varios problemas en una misma ejecución, el primer problema tardaba mucho más que el resto, por eso mismo, y con la finalidad de poder llevar a cabo más ejecuciones para este experimento, generamos 11 problemas con 11 *seeds* distintas para cada ejecución y descartamos el primer resultado. Por lo tanto puede haber discrepancia entre el tiempo resultante de este experimento y otros, pero como lo que nos interesa es el crecimiento de este tiempo, no importa esto.

Resultados

En primer lugar, cabe remarcar que los resultados obtenidos son significativos. Para ello, se muestra a continuación una tabla en la que aparece el *t-value* de cada uno de los casos

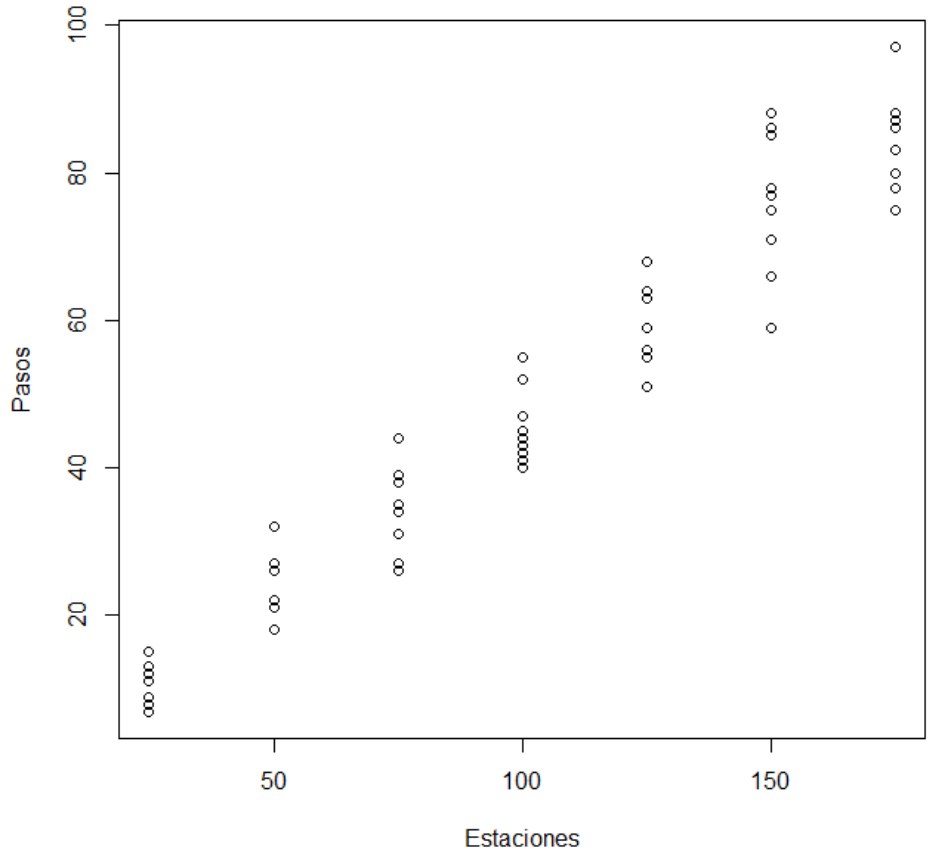
estudiados. Esta tabla también puede ser útil para visualizar con qué casos estamos experimentando:

	T-value
Pasos	15.303
Tiempo de ejecución	5.4907

Como se puede observar, todos los valores se sitúan por encima de 5. Por lo tanto, son valores significativos.

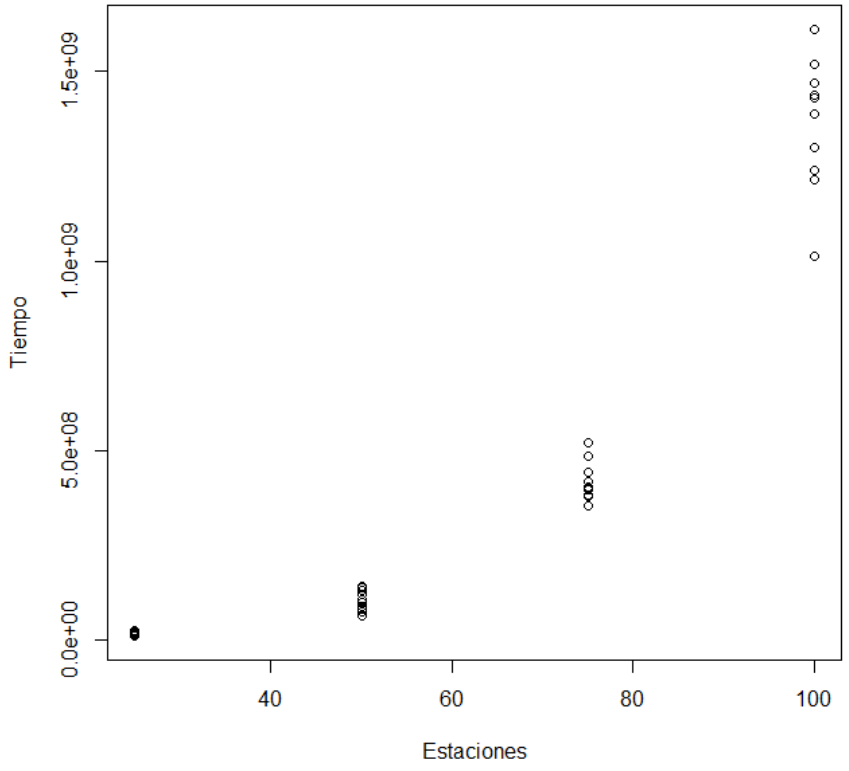
La tabla con todos los resultados obtenidos se encuentra en el anexo. En este apartado se observarán los gráficos.

A continuación veremos el gráfico para la cantidad de pasos en función del número de estaciones.

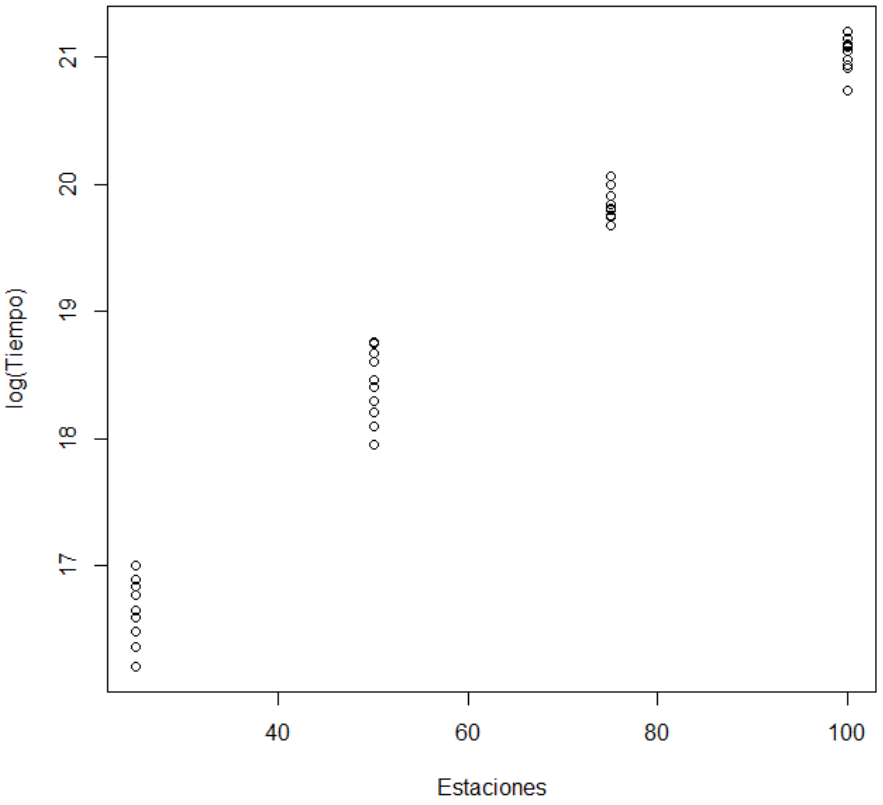


Como podemos observar en el gráfico, a diferencia de como pensábamos, el crecimiento de los pasos en función del número de estaciones es claramente lineal. Más concretamente, esta recta tiene un pendiente de 0.50629, lo que significa que, para cada 25 bicicletas que aumentemos, deberíamos ver un aumento de unos 12,5 pasos. También podemos ver que cuanto mayor es el número de estaciones, mayor es la varianza.

A continuación, veremos el gráfico para el crecimiento del tiempo en nanosegundos en función de la cantidad de estaciones.



Aquí en ya podemos ver claramente la forma exponencial que esperábamos del crecimiento. Además, es mucho más claro, ya que con este han hecho falta menos ejecuciones, puesto que a las 100 estaciones se intuía una clara tendencia exponencial. Aun así para comprobar este resultado, también podemos graficar el logaritmo del tiempo que, al ser exponencial, pasará a ser lineal.



Efectivamente, podemos comprobar que el tiempo es exponencial, ya que al aplicar el logaritmo vemos como ahora el gráfico sí que coge una forma de recta.

Conclusiones

Respecto a las hipótesis planteadas anteriormente, hemos podido comprobar que efectivamente para el tiempo el crecimiento era exponencial, pero no era cierto que también lo era para los pasos. Viendo los resultados hemos llegado a la conclusión de que este resultado, a pesar de que pueda ser poco intuitivo, tiene una explicación lógica. Como estamos usando el método de *Hill Climbing* para este experimento, al aumentar el número de estaciones lo que estaremos aumentando será el número de sucesores posibles para cada estado de forma exponencial. Por lo tanto, aumentará el tiempo que tardamos en ver cuál es el mejor de ellos también exponencialmente. Pero el número de pasos que tengamos que hacer hasta encontrar un máximo local no dependerá de estas estaciones exponencialmente.

5.5 Comparación entre *Simulated Annealing* y *Hill Climbing*

Planteamiento

En este experimento se hace una comparación entre los algoritmos de *Hill Climbing* y *Simulated Annealing* para nuestro problema. Para ello se estudiará la diferencia entre el beneficio, la distancia recorrida y el tiempo de ejecución en las dos funciones heurísticas explicadas.

Hipótesis

Como hipótesis, creemos que *Simulated Annealing* va a funcionar mejor para las 3 variables estudiadas. Al no quedarse parado en máximos locales dará buenos resultados para el beneficio y la distancia recorrida. Además, al no tener que generar todos los sucesores posibles para cada operador, sino uno solo aleatorio, tardará menos en ejecutarse.

Método

Para realizar este experimento, generaremos 5 *seeds* aleatoriamente. Para cada una de las variables a comparar (beneficio, distancia recorrida y tiempo de ejecución) se obtendrán datos para el primer y el segundo heurístico y para cada algoritmo (*Hill Climbing* y *Simulated Annealing*). Esto significa que para cada variable, tendremos 4 casos a estudiar: *Simulated Annealing* con primer heurístico, *Simulated Annealing* con segundo heurístico, *Hill Climbing* con primer heurístico y *Hill Climbing* con segundo heurístico.

Cada uno de los casos, se ejecutará en 10 *seeds*. También hay que tener en cuenta que las *seeds* serán las mismas en todas las variables estudiadas, para que la variabilidad de estas no desvirtúe los resultados. En cada ejecución se medirá el beneficio, la distancia recorrida y el tiempo de ejecución.

Una vez recopilemos los resultados, se procederá a analizar la diferencia entre los dos algoritmos para cada variable y caso.

También es preciso mencionar que el escenario de este experimento tiene 25 estaciones, 1250 bicicletas, 5 furgonetas, demanda equilibrada y estrategia de generación de solución inicial compleja.

Resultados

En primer lugar, cabe remarcar que los resultados obtenidos son significativos. Para ello, se muestra a continuación una tabla en la que aparece el *t-value* de cada uno de los casos estudiados. Esta tabla también puede ser útil para visualizar con qué casos estamos experimentando:

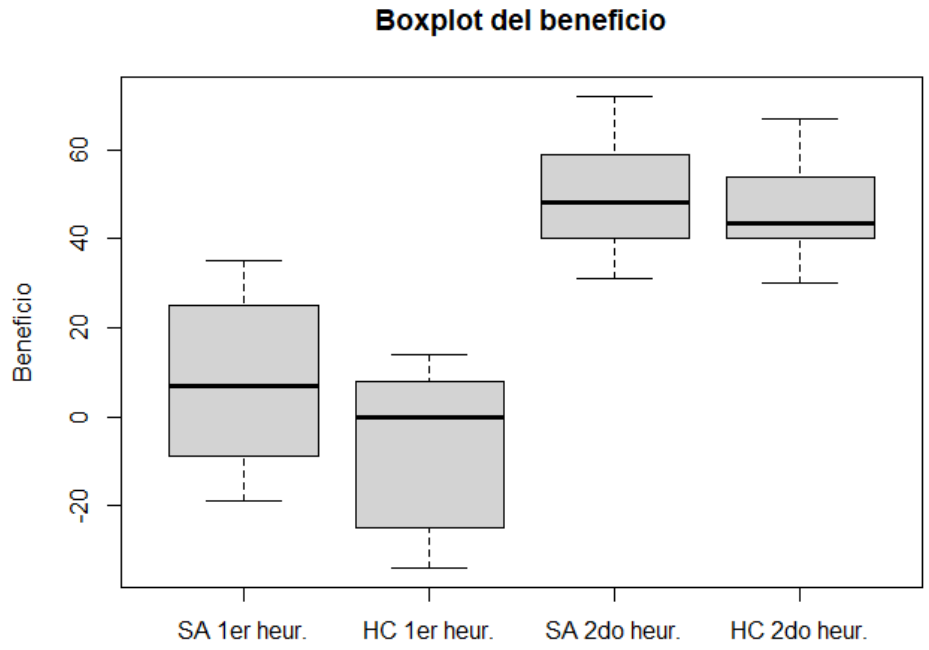
T-value	SA y 1er heur.	SA y 2do heur.	HC y 1er heur.	HC y 2do heur.
Beneficio	6.4178	11.326	6.0146	12.111
Distancia rec.	13.342	7.5187	12.346	10.519
Tiempo ejec.	298.71	246.54	85.22	86.061

SA: *Simulated Annealing*, HC: *Hill Climbing*

Como se puede observar, todos los valores se sitúan por encima de 5. Por lo tanto, son valores significativos.

La tabla con todos los resultados obtenidos se encuentra en el anexo. En este apartado se observarán los gráficos.

A continuación, se mostrará el *boxplot* de los resultados al analizar la variable de beneficio para sus 4 casos.

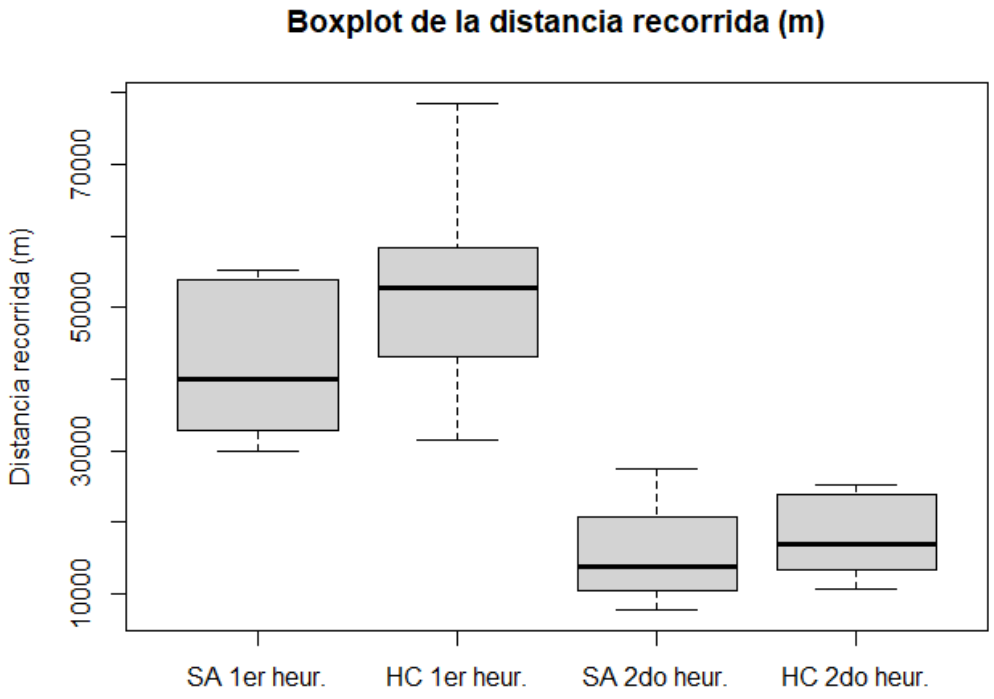


Lo primero que se puede apreciar es que hay notables diferencias entre el beneficio del primer heurístico y el segundo. Esto es debido a que en el primer heurístico el transporte es considerado gratuito y, consecuentemente, se recorre mucha más distancia, lo que incurre en un aumento del coste.

Analizando las diferencias entre *Simulated Annealing* y *Hill Climbing* en su beneficio con el primer heurístico, obtenemos un intervalo de confianza de (2.945, 23.854). Esto significa que, con un alto grado de confianza, podemos asegurar que, *Simulated Annealing* aporta un mayor beneficio. Además, sus medias para el primer heurístico son 7.4 y -6, para *SA* i *HC* respectivamente.

Analizando las diferencias entre *Simulated Annealing* y *Hill Climbing* en su beneficio con el segundo heurístico, obtenemos un intervalo de confianza de (-0.5218, 4.7218) del 95%. En este caso, Simulated Annealing da mejores resultados, pero no se puede aseverar con total confianza que sea mejor, porque el intervalo tiene parte negativa. Además, Simulated Annealing también da mayor beneficio de media con 48.7 frente a 46.6.

Seguidamente, se mostrará el *boxplot* de los resultados al analizar la variable de distancia recorrida para sus 4 casos, que se mide en metros.

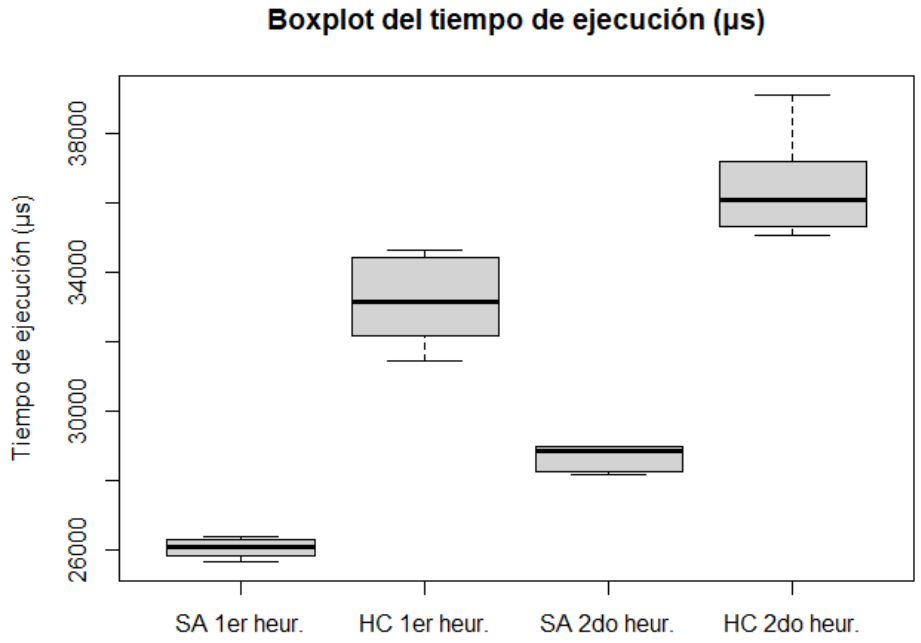


En este gráfico podemos observar que para la segunda función heurística, la distancia recorrida es muy inferior. Esto es lógico, ya que la segunda heurística no trata los transportes como gratuitos y, por lo tanto, forzará al algoritmo a buscar soluciones teniendo en cuenta que la distancia recorrida afecta al beneficio.

Analizando las diferencias entre *Simulated Annealing* y *Hill Climbing* en la distancia recorrida con el primer heurístico, obtenemos un intervalo de confianza de (-20843.91, -716.08) de 95% en metros. Esto significa que, se puede asegurar con alta confianza que *Simulated Annealing* recorre menos distancia. Sus medias para *SA* y *HC* son 45536m y 52380m respectivamente.

Analizando las diferencias entre *Simulated Annealing* y *Hill Climbing* en la distancia recorrida con el segundo heurístico, obtenemos un intervalo de confianza del 95% de (-5353.22, 293.22) metros. Por lo tanto, pese a que el intervalo tiene una parte positiva, es muy probable que SA recorra menos distancia. Además, tienen una media de 15170m y 17700m para *SA* y *HC* respectivamente.

Por último, se mostrará el *boxplot* de los resultados al analizar la variable de tiempo de ejecución para sus 4 casos, que se mide en µsegundos.



En un primer vistazo, se puede apreciar que el tiempo de ejecución con la segunda heurística es mayor para los dos algoritmos.

Analizando las diferencias entre *Simulated Annealing* y *Hill Climbing* en el tiempo de ejecución con el primer heurístico, obtenemos un intervalo de confianza de (-7961.484, -6307.516) µs. Esto supone que con casi total seguridad, podemos afirmar que el *Simulated Annealing* tarda menos en ejecutarse con el primer heurístico, en el caso estudiado. Además, tienen una media de 26041.5µs y 33176µs para *HC* y *SA* respectivamente.

Analizando las diferencias entre *Simulated Annealing* y *Hill Climbing* en el tiempo de ejecución con el segundo heurístico, obtenemos un intervalo de confianza de (-8839.332, -6728.268)ms. Es una situación parecida a la del primer heurístico, puesto que observamos que no hay valores positivos y, por lo tanto, el *Simulated Annealing* tarda menos en ejecutarse. Además, tienen una media de 28642.5µs y 36426.3µs para *HC* y *SA* respectivamente.

Conclusiones

Respecto a la hipótesis planteada, ha sido confirmada para todas las variables y ambas heurísticos, puesto que *Simulated Annealing* ha ofrecido mejores resultados continuamente. Solo en el beneficio y la distancia recorrida para la segunda heurística ha sido igualado entre ambos algoritmos, pero *Simulated Annealing* seguía teniendo ventaja.

En conclusión, *Simulated Annealing* que, al no quedarse atrapado en máximos locales y no tener que generar todos los sucesores posibles, da mejores resultados, en los casos estudiados. Por lo tanto, es el algoritmo escogido.

5.6 Comparación para distintas demandas

Planteamiento

En este experimento llevaremos a cabo una comparación entre la demanda en hora punta y equilibrada.

Hipótesis

Nuestra hipótesis es que, dado que en hora punta habrá algunas estaciones que tengan más demanda que las demás, el problema se simplificará porque es más probable que haya que centrarse en esas estaciones para maximizar el beneficio y no tener que atender tantas estaciones como con la demanda equilibrada. Por lo tanto, creemos que el tiempo de ejecución será menor para el escenario en hora punta.

Método

El método para realizar este experimento consistirá en tener un escenario con 25 estaciones, 1250 bicicletas, 5 furgonetas, generador de solución complejo y segundo heurístico. Para cada una de las posibilidades de la demanda (equilibrada y hora punta), se efectuarán ejecuciones en 10 *seeds*, que serán las mismas para ambas opciones con tal de que la variabilidad de estas no afecte a los resultados.

Por lo tanto, para cada opción se harán 10 ejecuciones en las que se comprobará el tiempo de ejecución, utilizando el algoritmo de *Simulated Annealing*.

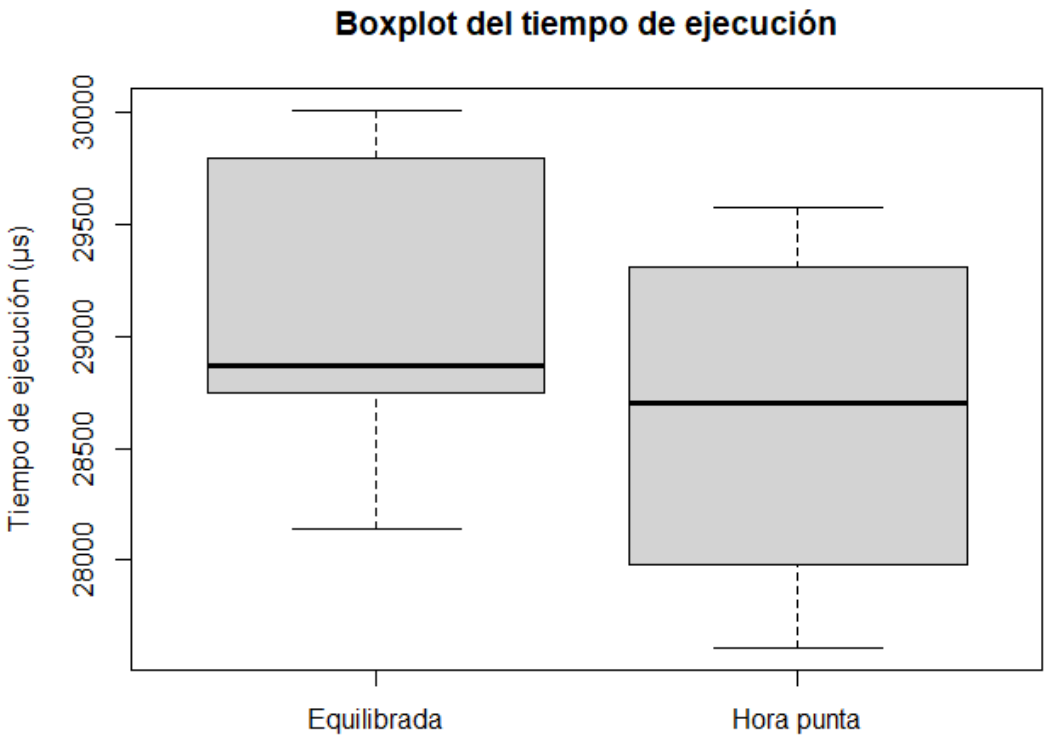
Resultados

Antes de mostrar los datos recopilados para efectuar la comparación, cabe aclarar que los resultados alcanzados son significativos. Para ello, mostramos el *t-value* de los resultados con la demanda equilibrada: 149.93 y con la demanda en hora punta: 121.38. Como se puede observar, ambos valores están claramente por encima del umbral de 5, por lo que son valores significativos.

A continuación, enseñamos la tabla con todos los valores recogidos.

	Seed 1	*Seed* 2	*Seed* 3	*Seed* 4	*Seed* 5	*Seed* 6	*Seed* 7	*Seed* 8	*Seed* 9	*Seed* 10
Equilibrada	28981	30006	28141	29789	28816	28557	29841	28744	28914	28814
Hora punta	27626	29574	29455	29311	27610	28208	28423	28986	27982	29021

Seguidamente, se muestra el resultado con un gráfico *boxplot* en µs.



Se puede observar que en hora punta se tarda algo menos de tiempo en ejecutar. En concreto, la media en hora punta es de 28619.6µs y en equilibrada, 29060.3µs. Además, hemos calculado el intervalo de confianza del 95% de la diferencia entre el tiempo de ejecución entre la demanda equilibrada y hora punta (-171.54, 1052.94)µs. Al tener mayor parte positiva, significa que la demanda equilibrada es probablemente la que tarda más, aunque no se puede asegurar con total confianza al haber parte negativa.

Conclusiones

Respecto a la hipótesis, ha sido acertada y la razón que encontramos a que en hora punta se tarde menos en ejecutar es que es más fácil satisfacer la demanda de unas pocas estaciones, que distribuir las bicis entre todas las estaciones, que tiene mayor complejidad.

5.7 Comparación para distinta cantidad de furgonetas

Planteamiento

En este experimento veremos el número óptimo de furgonetas para maximizar el beneficio obtenido y minimizar el tiempo de ejecución. Este experimento se ejecutará dos veces, una en hora punta y otra no.

Hipótesis:

El beneficio aumentará siempre que aumentes el número de furgonetas, para los dos casos, hasta que alcance el número de estaciones, ya que no puede salir más de una furgoneta de una misma estación.

Método:

Para realizar el experimento, usaremos el algoritmo de *Hill Climbing*, el segundo heurístico, la estrategia para generar la solución inicial compleja y el escenario que nos presentaban en el primer experimento. Este es: 25 estaciones, 1250 bicicletas y 5 furgonetas que irán aumentando de 5 en 5. Ejecutaremos una vez por cada una de las 10 *seeds* diferentes y aleatorias para cada cantidad de furgonetas. Comprobaremos que los resultados sean significativos con el *t-value* y mediremos el beneficio y el tiempo de ejecución para razonar cuál es el número óptimo de furgonetas.

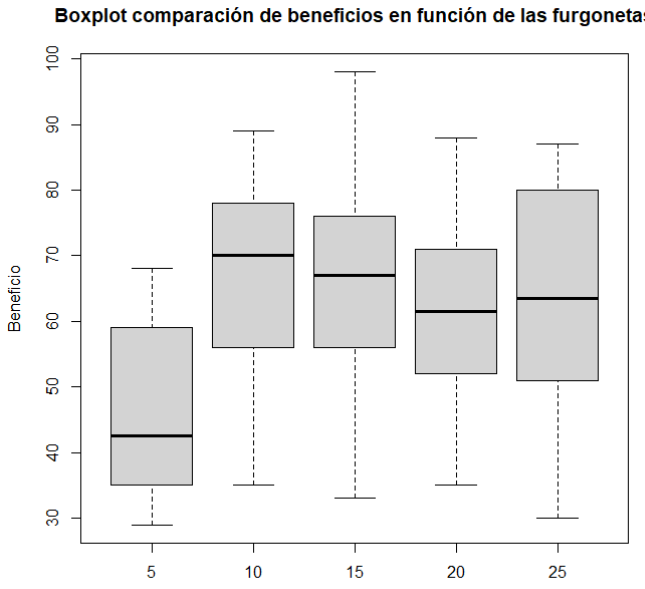
Resultados:

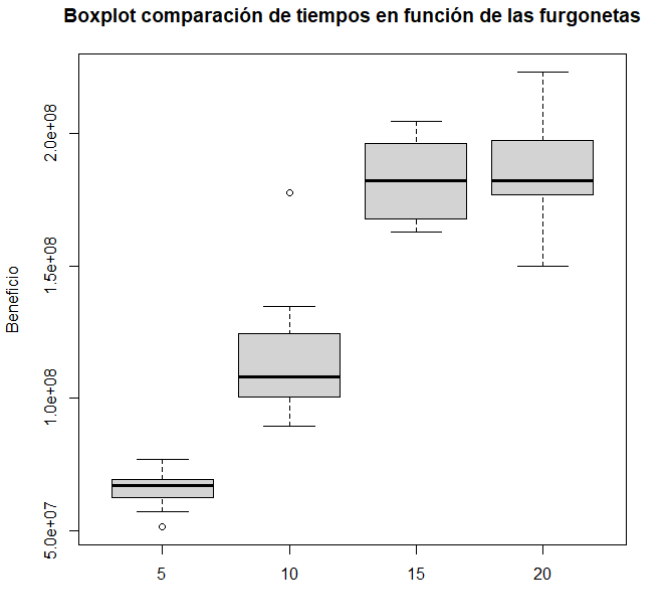
Para empezar, como siempre, comprobamos que el *t-value* es mayor a 5 para ver lo significativos que son los datos recogidos.

	Benef. sin HP	Benef. con HP	Tiempo sin HP	Tiempo con HP
5 furgonetas	10.556	14.462	26.814	15.489
10 furgonetas	12.111	14.127	14.426	15.975
15 furgonetas	11.611	18.214	35.732	38.508
20 furgonetas	13.882	17.63	27.904	25.521
25 furgonetas	11.227	16.998	30.95	23.959

Como podemos ver, los valores del *t-value* son grandes, por lo tanto, podemos afirmar que los valores que ahora veremos son representativos.

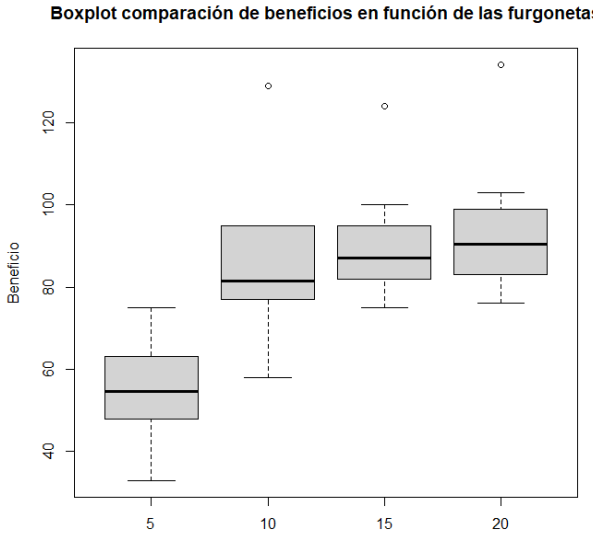
En primer lugar, veremos la evolución del beneficio y tiempo en función de las furgonetas en el caso en el que no tenemos hora punta. Los datos de estas gráficas están en el anexo.

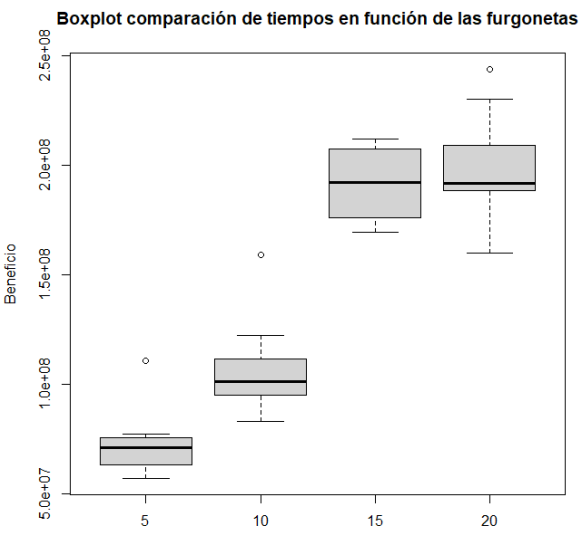




Como se puede ver, el beneficio al aumentar hasta 10 furgonetas es mucho más grande que en el resto de aumentos. También vemos que el salto de tiempo entre 10 y 15 furgonetas es muy grande, por lo que nos podría sugerir que si el salto en cuanto a beneficio no es muy importante, hacer este aumento puede ser poco óptimo. En resumen, el beneficio se estanca a partir de las 10 furgonetas y es también a partir de este punto donde el tiempo casi se duplica.

Vemos ahora el mismo experimento pero activando la hora punta.





Podemos ver como, aunque activemos la hora punta, se comporta de igual manera. El beneficio pega un gran salto entre 5 y 10 furgonetas a partir de donde luego se estanca. También volvemos a ver el gran salto temporal entre 10 y 15 furgonetas.

Conclusiones

Viendo los resultados obtenidos podemos apreciar que esté o no activada la hora punta, todo parece apuntar a que el número de furgonetas que puede llegar a interesar más es 10. Esto se debe a que, como comentamos, el salto temporal entre 10 y 15 furgonetas es demasiado grande y no conlleva un salto de igual magnitud en cuanto a beneficio. Además, hemos podido observar que el hecho de activar o no la hora punto lo único que hace es aumentar el beneficio, aunque no cambié como crece este. La hora punta tampoco afecta ni al valor ni al crecimiento del tiempo de ejecución. Por lo tanto, la hipótesis era falsa, ya que el valor óptimo son 10 furgonetas.

6. Comparación entre *Hill Climbing* y *Simulated Annealing*

A continuación, explicaremos los resultados del experimento que compara los resultados de *Hill Climbing* y *Simulated Annealing* para la primera y la segunda heurística. Aunque este experimento ya fue explicado con detalle en su respectivo apartado, incluyendo medias, tablas, gráficos e intervalos de confianza, trataremos de nuevo la comparación de forma más breve.

En cuanto al beneficio, para la primera heurística Simulated Annealing funciona mejor con un alto nivel de confianza. Para la segunda heurística, Simulated Annealing también tiene una media superior, pero no se puede garantizar con alta confianza que la esperanza sea superior.

Analizando las diferencias entre *Simulated Annealing* y *Hill Climbing* en la distancia recorrida con el primer heurístico, se puede asegurar con alta confianza que *Simulated Annealing* recorre menos distancia que *Hill Climbing*. De forma similar al beneficio, para la segunda heurística, *Simulated Annealing* tiene una media inferior (recorre menos), pero no se puede aseverar con alta confianza que la esperanza también lo sea.

Analizando las diferencias entre *Simulated Annealing* y *Hill Climbing* en el tiempo de ejecución con el primer y el segundo heurístico, obtenemos grandes diferencias en las que se refleja que *Simulated Annealing* tarda mucho menos en ejecutarse con alta confianza. Esto es debido a que *Simulated Annealing* no genera todos los sucesores, sino que genera uno aleatorio, por lo que tarda menos tiempo en realizar cada paso.

En conclusión, *Simulated Annealing,* al no quedarse atrapado en máximos locales y no tener que generar todos los sucesores posibles, da mejores resultados para las 3 variables, en los casos estudiados. Por lo tanto, ha sido el algoritmo escogido como el mejor.

7. Conclusiones

Por último, como hemos podido ver durante toda la documentación de la práctica, hemos conseguido solucionar el problema del Bicing dando una primera solución muy amplia y luego eliminando partes que empeoraban o incluso añadiendo nuevas que mejoraban el rendimiento de este mediante experimentos.

Gracias a estos experimentos hemos concluido que la mejor forma de solucionar este problema es con el conjunto de operadores descrito anteriormente en el primer experimento. Acompañado por la segunda generación, aleatoria para así asegurarnos de que el algoritmo de *Hill Climbing* explore distintas soluciones. Y por último hemos visto que la mejor era la segunda heurística y el mejor algoritmo *Simulated Annealing*.

Después de experimentar con parámetros que dependen de nosotros como los operadores o heurísticos para así mejorarlos, hemos pasado a comprobar para qué valores de estaciones y furgonetas funcionaba mejor nuestro programa. O si la hora punta afectaba de forma positiva o negativa a nuestra propuesta de solución.

Nos hemos encontrado con varios baches durante la práctica, como por ejemplo el tener que repetir en varias ocasiones experimentos por errores sin solucionar en el código o porque no estábamos siendo suficientemente rigurosos.

Pese a ello, hemos conseguido llegar a una solución correcta, que, respaldada por la cantidad de experimentos que hemos realizado y sus resultados, podemos afirmar con suficiente seguridad que es óptima.

8. Anexo

Experimento 1: Escoger conjunto de operadores

Beneficio para diferentes conjuntos de operadores (10 *seeds*, 3 ejecuciones para cada una)

Seed	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5
Añadir Desp	-17	-12	-28	-2	-21	-11	-29	-50	-47	-43	-20	-54	-44	-32	-15
Modif. Desp	-9	20	23	13	7	-6	-16	-8	12	-25	-29	-6	-34	-32	-30
Añadir bicis	4	12	3	15	15	1	23	15	-4	8	2	-4	-23	-19	-18
Quitar bicis	0	16	10	13	10	9	-11	-8	-16	0	7	10	10	-33	-24
Quitar Desp.	0	16	38	-1	7	-25	-8	-13	-26	3	-8	-12	-28	-36	-25

Seed	6	6	6	7	7	7	8	8	8	9	9	9	10	10	10
Añadir Desp	-21	-5	-44	-20	-20	-15	-9	-28	-13	-26	-43	-43	22	21	-2
Modif. Desp	-21	-15	7	-2	-9	-6	15	-1	10	-6	-13	-14	27	25	19
Añadir bicis	-1	-5	-4	7	-10	9	-8	15	-10	29	-23	-29	27	32	42
Quitar bicis	-13	-11	12	-9	2	-9	-9	18	13	-22	-17	-25	38	36	42
Quitar Desp.	9	8	1	-6	-1	-2	-15	-12	2	-30	-31	-6	37	27	37

Pasos para diferentes conjuntos de operadores

Seed	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5
Añadir Desp	5	3	5	4	4	5	4	4	4	2	4	4	4	5	4
Modif. Desp	9	7	9	8	7	9	7	6	6	8	10	9	6	6	6
Añadir bicis	12	13	12	10	7	7	10	7	10	10	11	13	6	6	8
Quitar bicis	11	10	12	8	11	8	10	9	8	9	13	11	11	6	5
Quitar Desp.	11	11	14	9	10	7	8	9	11	12	11	7	7	7	6

Seed	6	6	6	7	7	7	8	8	8	9	9	9	10	10	10
Añadir Desp	4	4	3	5	5	4	4	4	4	5	5	5	5	3	5
Modif. Desp	10	8	9	8	7	6	9	9	9	7	7	8	8	8	9
Añadir bicis	12	9	9	7	9	11	11	12	12	9	9	8	11	13	10
Quitar bicis	8	8	9	7	11	8	9	13	9	10	9	7	7	9	8
Quitar Desp.	11	7	9	8	8	9	12	11	12	8	9	9	10	8	11

Experimento 2: Determinar mejor método de generación

Resultados del beneficio obtenido al ejecutar 10 seeds aleatorias con los tres tipos de generación

	seed 1	seed 2	seed 3	seed 4	seed 5	seed 6	seed 7	seed 8	seed 9	seed 10
1ª	-7	-27	-14	-25	23	-5	24	-59	7	-13
2ª	12	28	-30	10	39	10	22	16	18	-9
3ª	-1	-4	-16	-21	29	-16	43	-47	-20	-31

Resultados del tiempo en µs obtenido al ejecutar 10 seeds aleatorias con los tres tipos de generación

	seed 1	seed 2	seed 3	seed 4	seed 5	seed 6	seed 7	seed 8	seed 9	seed 10
1ª	106762	101954	112702	109207	100015	118764	113447	103835	118449	99124
2ª	106092	65460	74726	60303	70917	74612	64553	71781	70956	79204
3ª	65908	73795	85133	79570	66832	80282	77205	71225	81592	92615

Experimento 4: Evolución del tiempo en función de las estaciones

Resultados obtenidos con cada una de las 10 *seeds* aleatorias para todos los conjuntos de estaciones hasta 175. El resultado está expresado en pasos dados.

	seed 1	*seed* 2	*seed* 3	*seed* 4	*seed* 5	*seed* 6	*seed* 7	*seed* 8	*seed* 9	*seed* 10
25	8	12	13	12	8	7	9	15	13	11
50	26	21	22	22	21	32	22	22	18	27
75	34	38	44	35	27	35	39	31	38	26
100	43	55	52	41	47	42	45	43	44	40
125	68	64	55	63	56	68	56	63	59	51
150	85	88	77	86	59	71	85	66	78	75
175	86	97	75	88	80	87	83	97	75	78

Número de estaciones por filas y seed por columnas.

Resultados obtenidos con cada una de las 10 *seeds* aleatorias para todos los conjuntos de estaciones hasta 100. El resultado está expresado en tiempo de ejecución en µs.

	seed 1	*seed* 2	*seed* 3	*seed* 4	*seed* 5	*seed* 6	*seed* 7	*seed* 8	*seed* 9	*seed* 10
25	20656	12770	11009	17060	14491	19147	21650	24399	16168	16963
50	141174	120520	138340	80536	104432	72270	98388	87995	62453	129200
75	520433	352807	484406	377515	402353	416714	397421	442375	400449	380829
100	1468421	1520449	1438753	1215243	1610649	1012155	1239163	1431718	1388603	1300727

Experimento 5: Comparación entre *Simulated Annealing* y *Hill Climbing*

Resultados obtenidos con la primera y la segunda función heurística, para el beneficio, la distancia recorrida en metros y el tiempo de ejecución en µs.

	Seed 1	*Seed* 2	*Seed* 3	*Seed* 4	*Seed* 5	*Seed* 6	*Seed* 7	*Seed* 8	*Seed* 9	*Seed* 10
SA 1r heur. B	-1	10	-19	31	5	25	-12	35	9	-9
SA 1r heur. D	40200	32900	55200	30000	43000	39800	54400	29900	36800	53800
SA 1r heur. T	25911	26269	26362	25863	26305	26266	25720	25651	25805	26263
SA 2n heur. B	32	40	51	59	40	72	53	64	31	45
SA 2n heur. D	27500	13500	20800	1100	14200	10500	9800	7800	14000	22600
SA 2n heur. T	28155	28310	28803	28246	28975	2891	28180	28867	28980	28948
HC 1r heur. B	-34	0	-29	14	-13	0	-25	8	11	8
HC 1r heur. D	78500	43900	66400	42400	54500	58300	54000	51300	31400	43100
HC 1r heur. T	31558	34604	33075	34117	34417	34638	33183	32154	32569	31445
HC 2n heur. B	32	43	41	54	40	57	53	62	30	44
HC 2n heur. D	25200	15500	23900	19700	14500	13400	18400	11700	10800	23900
HC 2n heur. T	39117	35339	377806	36615	35060	35563	35507	36891	37175	35190

Experimento 7: Comparación para distinta cantidad de furgonetas

Evolución del beneficio en función de las furgonetas.

	seed 1	*seed 2*	*seed 3*	*seed 4*	*seed 5*	*seed 6*	*seed 7*	*seed 8*	*seed 9*	*seed 10*
5	40	30	61	35	29	43	59	42	58	68
10	57	35	72	56	46	68	78	73	89	85
15	69	33	73	56	48	62	80	65	76	98
20	62	35	71	52	52	62	71	61	61	88
25	45	42	2	49	42	0	23	11	24	9

Evolución del tiempo en función de las furgonetas en µs.

	seed 1	*seed 2*	*seed 3*	*seed 4*	*seed 5*	*seed 6*	*seed 7*	*seed 8*	*seed 9*	*seed 10*
5	63681	66533	69442	67866	57245	51663	62489	67602	76761	76540
10	114595	106105	102616	97876	100300	177474	110317	89335	124241	134755
15	200896	175895	162948	188293	169318	194407	196180	167517	204465	164118
20	191700	197324	200371	149857	159555	176728	177541	223049	184955	178988
25	209165	160112	191804	183480	171875	167728	184684	153112	206473	177336

Evolución del beneficio en función de las furgonetas en hora punta.

	seed 1	*seed 2*	*seed 3*	*seed 4*	*seed 5*	*seed 6*	*seed 7*	*seed 8*	*seed 9*	*seed 10*
5	33	59	63	66	48	47	50	48	60	75
10	1	5	78	95	69	82	92	7	95	129
15	92	75	77	100	82	87	84	95	99	124
20	99	0	83	96	103	76	85	84	98	134
25	96	70	78	105	88	80	91	80	103	133

Evolución del tiempo en función de las furgonetas en µs en hora punta.

	seed 1	*seed 2*	*seed 3*	*seed 4*	*seed 5*	*seed 6*	*seed 7*	*seed 8*	*seed 9*	*seed 10*
5	110823	75564	64533	62955	63144	57342	69349	72853	73357	77496
10	103703	95923	83078	159052	99234	95313	103340	95117	122533	111785
15	194178	189963	175961	169494	207668	172677	207397	212071	185188	204556
20	230143	160119	188591	208584	243787	209323	189035	191566	177249	191834
25	222244	181332	155313	196295	173323	204169	210773	218619	200808	190841

9. Competencia de trabajo en equipo: Trabajo de innovación

9.1 Tema elegido

El tema que hemos elegido ha sido la IA de Google Vision, un servicio de API de Google Cloud que proporciona capacidades avanzadas de detección de objetos y reconocimiento de contenido visual mediante el uso de modelos de aprendizaje automático entrenados por Google. Permite a los desarrolladores y empresas integrar capacidades de visión por computadora en sus aplicaciones y flujos de trabajo, lo que facilita el análisis, la comprensión de imágenes/videos y la extracción de información valiosa de su contenido.

9.2 Reparto del trabajo entre los miembros del grupo

1. Introducción, portada, formato, índice -> Diego Velilla Recio
2. Descripción del servicio y de la innovación -> Sergi Pérez Escalante
- 2.1 ¿Qué es Google Vision?
  - Características de Google Vision
- 2.2 Innovación de Google Vision

3. Descripción de las técnicas de IA que se han utilizado
- 3.1. Detección de rostros -> Pablo Buxó Hernando
  - Cascada de clasificadores en tiempo real
  - Eigenfaces
  - Otras aportaciones
  - Como ha sido utilizado en Google Vision
- 3.2. Reconocimiento óptico de carácteres (OCR) -> Diego Velilla Recio
  - Kurzweil Reading Machine
  - Tesseract OCR
  - Como ha sido utilizado en Google Vision
- 3.3. Detección de contenido inapropiado -> Sergi Pérez Escalante
- 3.4. Detección de objetos y etiquetas

5. Impacto de la innovación
- 5.1. Impacto de la innovación en la empresa (beneficios, riesgos, posición en el mercado)
- 5.2. Impacto de la innovación hecha al producto en el usuario o en la sociedad (beneficios y riesgos)

6. Bibliografía (formato APA) -> Sergi Pérez Escalante

9.3 Dificultades encontradas

Inicialmente, tuvimos problemas en elegir el tema, dado que al haber muchas IA era complicado saber con cuál quedarse y tener en cuenta la información disponible que hay en internet para hacer un buen trabajo, que sea una IA atractiva y que nos motive aprenderla, que haya tenido un impacto positivo en la empresa y en la sociedad…

La dificultad más grande con las que nos hemos topado es el hecho de que Google Vision no es que utilice una técnica IA en concreto, sinó que aglutina muchas como podrían ser reconocimiento de rostros faciales, reconocimiento óptico de caracteres, detección de contenido inapropiado… de manera que seguramente no las podamos definir todas, y tendremos que ver cuáles serán las más relevantes y explicar brevemente algunas para poder dar una visión general de la innovación de Google Vision.

Otro problema que hemos encontrado es que dichas técnicas fueron creadas hace mucho, y han recibido diversas aportaciones de varios investigadores, de manera que dependiendo de la empresa se utiliza una versión o se utiliza otra. Además, ha habido aportaciones más significativas y otras menos, de manera que es importante hacer una evaluación con el equipo y decidir cuáles incluimos y cuáles no, ponernos de acuerdo, valorarlo…

En tema de búsqueda de la información, no es difícil de encontrar, puesto que hay mucho material y muchos artículos que sobre tecnología de análisis de fotografías, además de que Google es una empresa muy famosa, lo que hace que sus proyectos sean bastante conocidos

Por último, falta decidir cómo planificar el reparto del último punto (el del impacto de IA), ya que en parte depende de como gestionemos el apartado anterior, sobre cuantas técnicas describimos.

9.4 Lista de referencias

[1] Google. (n.d.-b). *Vision AI | Google Cloud*. Google. https://cloud.google.com/vision (Accedido el 25 de setiembre de 2023)

[2] *Google cloud vision API: Análisis de Imágenes en la nube*. ACKstorm. (2023, July 17). https://www.ackstorm.com/google-cloud-vision-api-imagenes-en-la-nube/#:~:text=Google%20Cloud%20Vision%20API%20es,informaci%C3%B3n%20valiosa%20de%20su%20contenido. (Accedido el 25 de setiembre de 2023)

[3] Özay, M. (2023, June 25). *What is google vision api & how does it work? - dopinger*. Dopinger Blog. https://www.dopinger.com/blog/what-is-google-vision-api (Accedido el 26 de setiembre de 2023)

[4] *Features*. What is Google Cloud Vision? (n.d.). https://www.resourcespace.com/blog/what-is-google-vision#:~:text=How%20does%20the%2 (Accedido el 26 de setiembre de 2023)

[5] Google. (n.d.-a). *Vertex AI vision | google cloud*. Google. https://cloud.google.com/vertex-ai-vision?hl=es (Accedido el 1 de octubre de 2023)

[6] Frąckiewicz, M. (2023a, August 26). *Api de Google Vision en el sector de la Salud: Mejora del Diagnóstico y el tratamiento Médicos*. TS2 SPACE. https://ts2.space/es/api-de-google-vision-en-el-sector-de-la-salud-mejora-del-diagnostico-y-el-tratamiento-medicos/ (Accedido el 1 de octubre de 2023)

[7] Camacho, L. F. Úbeda. (n.d.). *Reconocimiento facial con inteligencia artificial*. LinkedIn. https://www.linkedin.com/pulse/reconocimiento-facial-con-inteligencia-artificial-%C3%BAbeda-camacho/?originalSubdomain=es (Accedido el 3 de octubre de 2023)

[8] Viola, P., & Jones, M. J. (2004). Robust real-time face detection. *International Journal of Computer Vision*, *57*(2), 137–154. https://doi.org/10.1023/b:visi.0000013087.49260.fb (Accedido el 7 de octubre de 2023)

[9] Turk, M., & Pentland, A. (1991). Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, *3*(1), 71–86. https://doi.org/10.1162/jocn.1991.3.1.71 (Accedido el 8 de octubre de 2023)

[10] Zhao, W., Chellappa, R., Phillips, P. J., & Rosenfeld, A. (2003). Face recognition. *ACM Computing Surveys*, *35*(4), 399–458. https://doi.org/10.1145/954339.954342 (Accedido el 11 de octubre de 2023)

[11] Yan, J., Zhang, X., Lei, Z., & Li, S. Z. (2014). Face detection by structural models. Image and Vision Computing, 32(10), 790-799. https://link.springer.com/referenceworkentry/10.1007/978-3-030-63416-2_798 (Accedido el 12 de octubre de 2023)

[12] Frąckiewicz, M. (2023b, August 26). *Aplicación de la Api de Google Vision a la detección y el seguimiento de objetos*. TS2 SPACE. https://ts2.space/es/aplicacion-de-la-api-de-google-vision-a-la-deteccion-y-el-seguimiento-de-objetos/ (Accedido el 12 de octubre de 2023)

[13 Media. (2021, November 18). *OCR: ¿Qué es el reconocimiento óptico de Caracteres?*. Welcome to the Signaturit blog. https://blog.signaturit.com/es/ocr-que-es-el-reconocimiento-optico-de-caracteres (Accedido el 13 de octubre de 2023)

[14] Kurzweil, R. (1974). "The Kurzweil Reading Machine for the Blind". En IEEE Transactions on Man-Machine Systems, 15(1), 13-23. doi: https://doi.org/10.1109/tmms.1974.299848 (Accedido el 18 de octubre de 2023)

[15] Smith, R. (2007). An overview of the tesseract OCR engine. *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) Vol 2*. https://doi.org/10.1109/icdar.2007.4376991 (Accedido el 21 de octubre de 2023)

[16] *Announcing tesseract OCR*. Announcing Tesseract OCR. (n.d.). https://googlecode.blogspot.com/2006/08/announcing-tesseract-ocr.html (Accedido el 21 de octubre de 2023)

[17] https://programminghistorian.org/en/lessons/ocr-with-google-vision-and-tesseract

(Accedido el 22 de octubre de 2023)