Documentación de la práctica de búsqueda local

Laboratorio de Inteligencia Artificial
Curso 2017/2018

Guillem Rodríguez Corominas
David Moreno Borràs
Marco Soldan

Índice general

1. El Problema

1.1 Descripción del problema.

El problema consiste en ayudar a una compañía de distribución de combustible a determinar, diariamente, a qué gasolineras de un área debe reabastecer para que éstas puedan seguir funcionando.

La compañía dispone de n **centros de distribución** repartidos dentro de ese área. En cada uno hay **uno (o más) camiones** que permiten llevar el combustible a las gasolineras que lo necesiten. Cada **camión** tiene capacidad para llenar **dos depósitos** completamente.

Las **gasolineras** tienen **varios depósitos** donde se almacena dicho combustible. Cuando un depósito se vacía, la gasolinera genera una **petición**. De esta manera, cada día se dispone de un listado de peticiones de gasolineras que han ido llegando y todavía no se han podido servir. Algunas peticiones pueden llevar varios días sin haber sido atendidas. Dependiendo del **número de días** que lleva una petición pendiente, la compañía rebaja el **precio** que cobra por el combustible.

Todas las mañanas, a una cisterna le ha de llegar un conjunto de peticiones y ésta las ha de servir haciendo unos viajes específicos.

OBJETIVO

El objetivo es encontrar, para un día, la **asignación de peticiones** a cisternas y cómo éstas se atenderán, **maximizando** el **beneficio** que obtenemos, pero **minimizando** lo que **perderemos** con las peticiones no atendidas y la **distancia** total recorrida por las cisternas.

DATOS

- **Número** de **gasolineras** a abastecer, las **coordenadas** de cada gasolinera, las **peticiones** que debemos abastecer de cada gasolinera y los **días** que llevan pendientes.
- **Número** de **centros de distribución** de los que disponemos y sus **coordenadas**.
- Número de **kilómetros** que puede recorrer un camión cisterna y cuantos **viajes** puede hacer **en un día**.

RESTRICCIONES

- Una cisterna **no** puede recorrer **más de k kilómetros diarios**. Las cisternas viajan a 80Km/h y trabajan durante 8 horas, así que podrán recorrer un **máximo de 640 Km diarios.**
- Una cisterna **no** puede hacer **más de v viajes diarios**. En este caso, el número **máximo** de **viajes** es **5**.

OBSERVACIONES

- El **precio** que la compañía cobra por el combustible sigue lo siguiente:
    - Si el número de días pendientes de una petición es **0** (se acaba de terminar el depósito), la compañía cobra el **102%** del precio.
    - A partir de ahí, el porcentaje de precio que se cobra sigue la **fórmula**:

      $$\%precio = (100 - 2^{dias})\%$$

      Obviamente, a la compañía no le interesa dejar esperar mucho las peticiones.
- La **distancia** entre los centros de distribución y las gasolineras se calcula de la siguiente forma:

  $$d(D,G) = |D_x - G_x| + |D_y - G_y|$$

  siendo D_x y D_y las coordenadas x e y del centro de distribución, G_x y G_y las coordenadas x e y de la gasolinera y $|\cdot|$ el valor absoluto.

ASUNCIONES

- El **área geográfica** es una **cuadrícula de 100x100** km^2 y disponemos de las coordenadas en las que están situados los centros de distribución.
- El **valor de un depósito** es de **1000**.
- El **coste por kilómetro** es **2**.
- Las gasolineras van usando un depósito hasta que se vacía y entonces pasan al siguiente.
- Puede haber **varias peticiones** por atender de la **misma gasolinera**, una por cada depósito que se le ha vaciado.
- Las **peticiones no atendidas** las atenderemos al **día siguiente**.
- Los tiempos de carga y descarga son **instantáneos**.

¿Por qué es un problema de búsqueda local?

1. Es **computacionalmente "imposible"** (muy costoso) encontrar la **solución óptima** para el problema (ya que tamaño del espacio de soluciones no nos lo permite), así que queremos encontrar la **mejor** de entre las soluciones **posibles alcanzable** en un **tiempo razonable**.
2. El **camino** para llegar a la solución **no** nos **importa**, buscamos en el espacio de soluciones.
3. Tenemos una forma de **aproximar** la **calidad** de una solución, que combina los elementos del problema y sus restricciones: la **heurística**. En este caso, la calidad se basa en las **ganancias** que obtenemos con una posible asignación/distribución de las peticiones.
4. Buscamos **optimizar** (maximizar o minimizar) dicha función heurística. .

1.2 Estado del problema y representación.

Para representar el estado del problema hemos probado varias implementaciones y al final hemos decidido quedarnos con la siguiente:

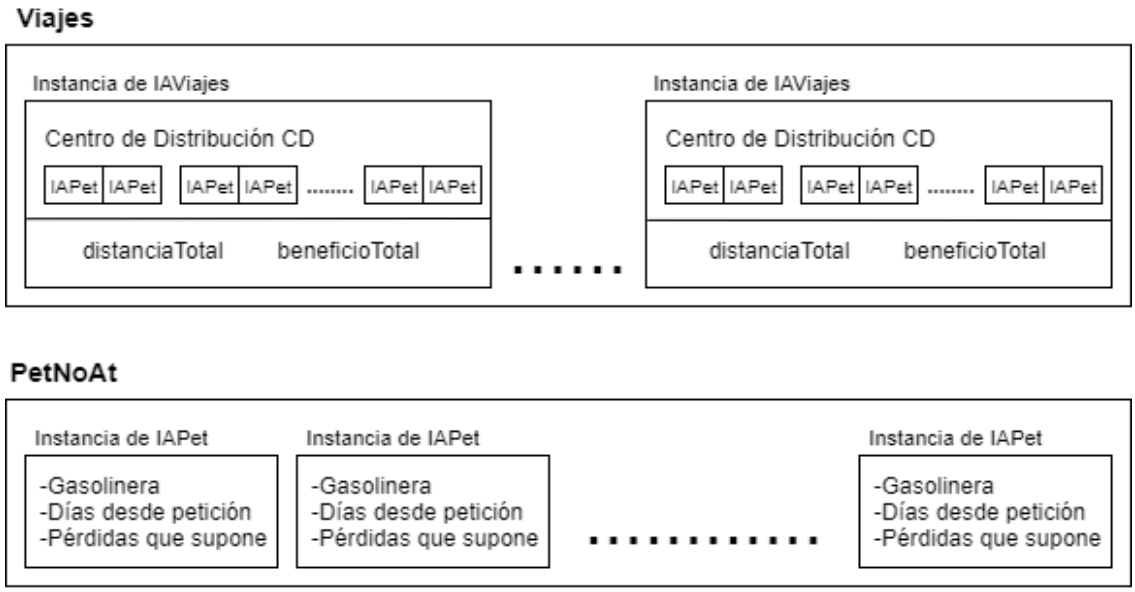


Figura 1.1 Representación visual del estado del problema

En primer lugar, en nuestra representación, una **petición** se representa como una instancia de una clase llamada ***IAPet***, la cual tiene como atributos:

- **g**: Gasolinera a la que pertenece.
- **dias**: Días que lleva pendiente la petición.
- **beneficio**: Beneficio que supondría atender dicha petición.
- **perdidas**: Pérdidas que supondría no atender dicha petición.

Hemos decidido implementar una **petición** como un objeto en sí para que fuera más fácil de manipular. Una petición se identifica por la gasolinera a la que pertenece y por los días que lleva sin ser atendida, datos esenciales que nos proporcionan. Además, decidimos añadir el beneficio y las pérdidas que supondría atender (o no) dicha petición para no tener que calcularlos cada vez que se trabaje con ésta, ya que los valores no varían durante la ejecución.

Por otra parte, también tenemos la clase ***IAViajes***. Cada instancia de la clase representa **todos los diferentes viajes que realiza un camión** de un cierto centro de distribución (y las respectivas peticiones atendidas). Para ello, consta de los siguientes elementos:

- **CD**: Centro de distribución al cual pertenece.
- **Peticiones**: ArrayList de peticiones. Consideramos un "viaje" cada par de peticiones.
- **distanciaTotal**: Distancia total recorrida por el camión en todos sus viajes.
- **beneficioTotal**: Beneficio que nos aportaría atender dichas peticiones.

Hemos decidido implementar los viajes así por las siguientes razones:

1. Es **fácil comprobar las restricciones**:

   - El tamaño máximo que puede tener *Peticiones* es 10, ya que cada camión puede, como máximo, hacer 5 viajes y atender 2 peticiones por cada uno.

   - distanciaTotal nunca puede ser mayor que el número máximo de kilometros que puede recorrer un camión por día (k). En este caso son 640Km.

2. Resulta **fácil añadir peticiones** y siempre se hace de forma **"eficiente"**, es decir, si hay un viaje con una sola petición, la nueva se "añadirá" a este viaje, si no, se "creará" uno nuevo. Simplemente la tenemos que añadir al final del ArrayList.
3. Nos **ahorra mucho cálculo** a la hora de tener que obtener el beneficio que nos aportaría atender las peticiones asignadas y los kilómetros recorridos por el camión, ya que siempre están guardados y actualizados.

Hay una instancia de ***IAViajes*** por cada camión.

El **estado** es una instancia de una clase llamada ***IAMap*** que consta de los siguientes elementos principales:

- **PetNoAt**: ArrayList de todas las **peticiones (*IAPet*)** del estado que **no** están **atendidas**.
- **Viajes**: ArrayList de los **viajes (*IAViajes*)** que realiza cada camión.
- **perdidas**: **Suma** de las **perdidas** de todas las **peticiones no atendidas**
- **petAtendidas**: Número **total** de **peticiones atendidas**.

El estado se podría representar simplemente con PetNoAt y Viajes, pero decidimos añadir el atributo *perdidas* para no tener que sumarlas todas cada vez. Por otra banda, tuvimos que añadir el atributo petAtendidas para que, cuando usamos *Simulated Annealing*, si no hemos atendido ninguna petición, siempre añada una.

A parte de esto, también tenemos un objeto tipo *CentrosDistribución* (*cd*) y otro tipo *Gasolineras* (*gas*), que son los **centros** y las **gasolineras** (respectivamente) que obtenemos como datos al inicio del problema. Como no tenemos que modificarlos, los hemos declarado como **estáticos,** para que así los compartan todos los estados sucesores y se use menos espacio (sería información duplicada innecesariamente).

REPRESENTACIÓN PREVIA

En un principio planteamos **otra representación,** que implementaba de forma distinta la clase ***IAMap*** e ***IAViajes*** (***IAPet*** no existía aún).

En esta implementación alternativa, una instancia de ***IAViajes*** tenía un centro de distribución **CD** y dos ArrayLists (de máximo 2 elementos), uno de *Integers* "**Pet**" y otro de *Gasolineras* "**Gas**". Así, el CD atendía la petición Pet[i] de la gasolinera Gas[i]. Es decir, una **instancia** de ***IAViajes*** guardaba **un único viaje** de un camión (que como sabemos puede atender 2 peticiones en un mismo viaje). ***IAMap*** (lo que representa el estado) tenía un ArrayList con los diferentes viajes ya programados.

Por otra banda, las **peticiones no atendidas** se guardaban en un **map**, donde las claves eran las diferentes gasolineras y el valor un ArrayList con los números de días que llevaban pendientes las peticiones de esa gasolinera.

Esta implementación presentaba varios problemas:

1. El **flujo de peticiones** de atendidas a no atendidas era **complicado** e **ineficiente**, por la forma en que estaban guardadas.
2. **Comprobar las restricciones** era bastante **costoso**. Por ejemplo, para no sobrepasar el número máximo de viajes por camión teníamos que recorrer todos los viajes para contar los que pertenecían dicho camión. Con el número de kilómetros pasaba una cosa similar.
3. Algunos **operadores** (como la acción de añadir una petición a un camión determinado) también resultaban ser algo **ineficientes**. Al añadir, por ejemplo, tenías que recorrer todos los viajes para ver si había alguno con solo una petición y, si no, crear uno nuevo.

1.3 Operadores

En un principio, los operadores que estuvimos considerando fueron **Add** y **Delete,** para añadir o borrar una petición de un viaje determinado. Al pensarlo mejor, nos dimos cuenta de que *Delete* era inútil, ya que, con el algoritmo de *Hill Climbing*, borrar una petición atendida y que dejara de estarlo hacía que la calidad de esa solución fuera peor (ya que perdíamos mucho beneficio) y, entonces, nunca se usaba.

Teniendo esto en cuenta, decidimos añadir un nuevo operador: **Swap,** para intercambiar dos peticiones (del mismo o distinto viaje).

ADD

El operador tiene la **forma** siguiente: **Add(V,P)**

donde V es un índice: 0 <= V < Viajes.size()
P es un índice: 0 <=P < PetNoAt.size()

Funcionamiento: Añade al viaje Viajes[V] la petición PetNoAt[P].

1. Coge la petición PetNoAt[P] y resta al atributo *pérdidas* las pérdidas de dicha petición (porque pasará a estar atendida).
2. Añade la petición a Viajes[v]. Si el camión ya ha llegado al máximo de viajes que puede atender, devuelve falso. En caso contrario, añade la petición a los viajes y modifica sus atributos (*distanciaTotal* y *benefcioTotal*). Si la distancia total supera el máximo permitido, también devuelve falso, aunque la petición queda añadida (será la función generadora de sucesores la encargada de no guardar el sucesor generado si dicha función devuelve falso).
3. Incrementa el número de peticiones atendidas (*petAtendidas*).

El operador en sí **no comprueba si los índices se encuentran fuera de rango**, ya que es una precondición que deben cumplir dichos índices. Las comprobaciones necesarias las hace la clase que genera los estados sucesores, llamando a la función siempre con el rango adecuado de valores.

Debido al rango de los índices, el **factor de ramificación = v*p**

donde v = Viajes.size() = número de camiones/cisternas.
p = PetNoAt.size() = número de peticiones no atendidas

Como v tiene un valor fijo (el tamaño de *Viajes* depende únicamente de los datos que nos proporciona el problema), el **factor de ramificación depende enormemente de p**: cuanto **más grande** sea el número de **peticiones no atendidas**, **mayor factor** de ramificación (y viceversa). Este operador, por lo tanto, es **especialmente útil** cuando casi **no se ha asignado ninguna petición** (los viajes están vacíos).

SWAP (1)

El operador tiene la **forma** siguiente: **SwapViaje(i1,j1,i2,j2)**

donde i1,i2 son índices: 0 <= i1 <= i2 < Viajes.size()
j1 es un índice: 0 <= j1 < Viajes[i1].size()
j2 es un índice: 0 <= j2 < Viajes[i2].size()

Funcionamiento: Intercambia la petición Viajes[i1][j1] con Viajes[i2][j2].

1. Intercambia la petición Viajes[i1][j1] con Viajes[i2][j2] y modifica los atributos *distanciaTotal* y *beneficioTotal* de ambos viajes. Si la distancia total de algún viaje supera el máximo permitido después del cambio, devuelve falso, aunque las peticiones quedan intercambiadas (será la función generadora de sucesores la encargada de no guardar el sucesor generado si dicha función devuelve falso).

El operador en sí **no comprueba si los índices se encuentran fuera de rango**, ya que es una precondición que deben cumplir dichos índices. Las comprobaciones necesarias las hace la clase que genera los estados sucesores, llamando a la función siempre con el rango adecuado de valores.

Debido al rango de los índices, el **factor de ramificación** es igual al **número de combinaciones sin repetición** (intercambiar con uno mismo es inútil) de **dos elementos** (peticiones) elegidos **entre el total** (n). Esto se denota como:

$$\mathbf{C_{(n,2)} = (n(n-1))/2}$$

donde n = número de peticiones atendidas

El factor, por lo tanto, es de orden **cuadrático**, y depende exclusivamente del número de peticiones atendidas. Cuanto más grande sea el número de peticiones atendidas, mayor factor (y viceversa). Este operador, por lo tanto, es **especialmente útil** cuando ya **hay** bastantes **peticiones asignadas**, ya que se generan una gran cantidad de posibles sucesores. El principal **problema** es que el **coste aumenta considerablemente** a medida que aumenta *n* y, por lo tanto, puede acabar siendo **contraproducente,** tanto temporal como espacialmente.

Posteriormente, cuando ya funcionaba HillClimbing, pensamos que se podría dar el caso en que, en un estado con muchas peticiones y todos los viajes llenos (porque no hay suficientes centros para atenderlas todas), muchas peticiones se quedarían en la lista *PetNoAt* para siempre, sin ser atendidas.

Teniendo esto en cuenta, planteamos un implementar un nuevo operador para poder intercambiar peticiones también con la lista de no atendidas, y no solo con las ya asignadas.

SWAP (2)

El operador tiene la **forma** siguiente: **SwapPets(i1,j1,P)**

donde i1 es un índice: 0 <= i1 < Viajes.size()
j1 es un índice: 0 <= j1 < Viajes[i1].size()
P es un índice: 0 <=P < PetNoAt.size()

Funcionamiento: Intercambia la petición Viajes[i1][j1] con PetNoAt[P].

1. Coge la petición PetNoAt[P] y resta al atributo *pérdidas* las pérdidas de dicha petición (porque pasará a estar atendida).
2. Coge la petición Viajes[i1][j1] y suma al atributo *pérdidas* las pérdidas de dicha petición (porque pasará a no estar atendida).
3. Intercambia la petición Viajes[i1][j1] con PetNoAt[P] y modifica los atributos *distanciaTotal* y *beneficioTotal* del viaje afectado. Si la distancia total de dicho viaje supera el máximo permitido después del cambio, devuelve falso, aunque las

peticiones quedan intercambiadas (será la función generadora de sucesores la encargada de no guardar el sucesor generado si dicha función devuelve falso).

El operador en sí **no comprueba si los índices se encuentran fuera de rango**, ya que es una precondición que deben cumplir dichos índices. Las comprobaciones necesarias las hace la clase que genera los estados sucesores, llamando a la función siempre con el rango adecuado de valores.

Debido al rango de los índices, el **factor de ramificación = n*p**

donde n = número de peticiones atendidas
p = PetNoAt.size() = número de peticiones no atendidas

Como *n = total de peticiones - p*, se puede comprobar que el **mayor factor de ramificación** se produce cuando la **mitad de las peticiones están atendidas** (y la otra mitad no), es decir, cuando *n = p = total de peticiones/2.* Este operador, por lo tanto, es **especialmente útil** cuando hay una **balanza entre peticiones atendidas y no atendidas**, ya que se generan una gran cantidad de posibles sucesores.

Como ya hemos visto, el factor de ramificación de los operadores depende de diversos factores, y es por eso que éstos son **útiles en situaciones distintas**. Por lo tanto, es lógico pensar que, a diferente estados iniciales, los operadores que funcionen mejor cambien. Es por eso que hemos decidido realizar los experimentos 1 y 2 juntos *(los resultados se muestran en el apartado de experimentos de la documentación)*.

1.4 Función heurística

Nuestra función heurística **optimiza** el **criterio de calidad del problema** descrito en el apartado 3.1 del enunciado y, por lo tanto, tenemos en cuenta 3 elementos: **beneficio**, **pérdidas** y **distancia recorrida**.

<u>BENEFICIO</u>

Hay que **maximizarlo**: es el sumatorio Σ del beneficio de todas las peticiones atendidas.

- Si días pendientes = 0 --> beneficio = 1020€ (un tanque = 1000€ y si se atiende la petición el mismo día cuesta 102%)
- Si días pendientes > 0 --> beneficio = 1000 - 10(2^días)

<u>PÉRDIDAS</u>

Hay que **minimizarlas**: es el sumatorio Σ del coste de todas las peticiones NO atendidas, es decir, la diferencia entre el beneficio que tendríamos hoy y si lo atendemos un día más tarde

$$\Sigma\ (beneficios(dia) - beneficios(dia + 1))$$

DISTANCIA RECORRIDA

Hay que **minimizarla**: es el sumatorio Σ de todos los kilómetros recorridos por los camiones multiplicado por 2 (ya que 1 km cuesta 2€)

Por lo tanto, el **heurístico** (que hay que minimizar) es:

H() = pérdidas + distancia recorrida - beneficios

ponderado 1:1:1,los tres factores tienen la misma importancia.

Hemos escogido estas ponderaciones porque, al estar todo "medido" en euros y al querer obtener el máximo beneficio general, creemos que esta es la **función que mejor evalúa la calidad de las soluciones**.

1.5 Estado inicial

Estuvimos planteando dos estados iniciales distintos: un **estado inicial vacío** (es decir, un instancia de IAViajes sin peticiones asignadas y todas las peticiones de las gasolineras del problema guardadas en PetNoAt) y un **estado inicial lleno** (atendiendo peticiones "aleatorias").

VACÍO

La **implementación** en este caso es **muy sencilla**, ya que lo único que debe hacer el constructor del estado del problema es crear *Viajes* (el ArrayList de ***IAViajes***) con tamaño n, donde n es el número de centros de distribución que tiene nuestro problema.

Pros:

- La **creación** de este estado inicial es **muy poco costosa**.
- Al no haber ninguna petición atendida, hay **mucho margen de mejora** para el algoritmo.

Contras:

- Al partir de una **solución relativamente "mala"** en cuanto a **calidad**, significa que el algoritmo necesitará abrir más "nodos" para mejorarla y, por lo tanto, realizar más iteraciones. Eso implica una **mayor coste**, tanto temporal como espacial, a la hora de **realizar la búsqueda**.
- Estamos **obligados a usar el operador Add**, ya que, si no, no podremos usar los demás, al no haber peticiones atendidas.

LLENO

La **implementación** de esta opción es **algo más complicada**. Lo que hacemos es, básicamente, ir añadiendo peticiones a los viajes conforme están en la lista PetNoAt, comprobando en todo momento que, al añadir una petición, ésta no viole ninguna

restricción (como, por ejemplo, que un camión haga más de 5 viajes o 640 km en un mismo día).

Pros:

- Partimos de una **solución de calidad aceptable**, teniendo en cuenta de que el **coste** de llegar a esta **no es tan alto** como si partiéramos del estado vacío y usáramos operadores.
- Podemos **usar cualquier operador**.

Contras:

- Las **peticiones aceptadas no** tienen por qué ser las **óptimas**, y puede ser que peticiones que fueran mejor se quedaran como no atendidas al inicio.
- Hay **poco margen de mejora**, así que los operadores pueden no ser muy eficientes en cuanto a mejora respecto al tiempo que tarden.

Las dos opciones han estado implementadas en todo momento, y es en los experimentos donde se ve cuál es más efectiva.

2. Experimentos

2.1 Experimentos 1 y 2

Decidimos hacer juntos los dos primeros experimentos, para determinar **qué conjunto de operadores** y **estrategia de generación de la solución inicial** nos da **mejores resultados**.

Lo decidimos así porque los operadores que son efectivos dependen mucho del estado inicial. Por ejemplo, un estado inicial vació usará principalmente Add, mientras que un estado inicial lleno usará Swaps para mejorar. Así que veremos qué **combinación estado inicial/operadores** es mejor.

Las combinaciones posibles son:

(1) *Vacío + Add*
(2) *Vacío + Add + Swap1*
(3) *Vacío + Add + Swap2*
(4) *Vacío + Add + Swap1 + Swap2*

(5) *Lleno + Add*
(6) *Lleno + Add + Swap1*
(7) *Lleno + Add + Swap2*
(8) *Lleno + Add + Swap1 + Swap2*
(9) *Lleno + Swap1*
(10) *Lleno + Swap2*
(11) *Lleno + Swap1 + Swap2*

**La combinación Vacío + Swap queda descartada porque no podría hacer swap al no haber ningún viaje asignado.*

Antes de hacer estos experimentos, esperábamos que la mejor combinación sería una con el estado inicial **vacío** y al menos ***Add***, ya que, así, es el propio algoritmo el que llena el estado mejorando la solución, y no aleatoriamente.

- **Observación:** Puede haber métodos de inicialización combinados con operadores determinados que obtienen mejores soluciones.
- **Planteamiento:** Probamos distintos métodos de inicialización combinados con distintos operadores y observamos la calidad de sus soluciones.
- **Hipótesis:** Todos los métodos de inicialización y operadores son iguales (H0) o hay algunos mejores que otros.

- **Método:**
    - Usaremos 20 semillas aleatorias, una para cada combinación.
    - Usaremos en todo momento un escenario en el que el número de centros de distribución es 10, hay un camión en cada centro y el número de gasolineras es 100.
    - Usaremos el algoritmo de Hill Climbing.
    - En cada ejecución registramos 3 valores: el **beneficio obtenido** (calculado a partir del beneficio que aporta atender las peticiones menos el coste de los kilómetros totales), el **tiempo de ejecución** y el **número de nodos** expandidos.

CONSIDERACIONES PREVIAS

El **tiempo de ejecución varía mucho según el ordenador** (por ejemplo, la ejecución del experimento especial, que a nosotros nos tardaba 2-3s en casa, en los Macs de la FIB tardaba medio segundo). Por eso, **los experimentos se realizan siempre en el mismo ordenador.**

Programamos una serie de métodos en el *Main* para poder realizar todos los experimentos con todas sus variaciones (manteniendo las mismas semillas) y guardamos los resultados en un .txt. El resultado es así:

Vacío + *Add + Swap1:*	*LLeno + Add + Swap1 + Swap2:*
93860.0 716.0 95	*94428.0 1352.0 114*
93376.0 726.0 102	*93944.0 1345.0 112*
[...]	*[...]*

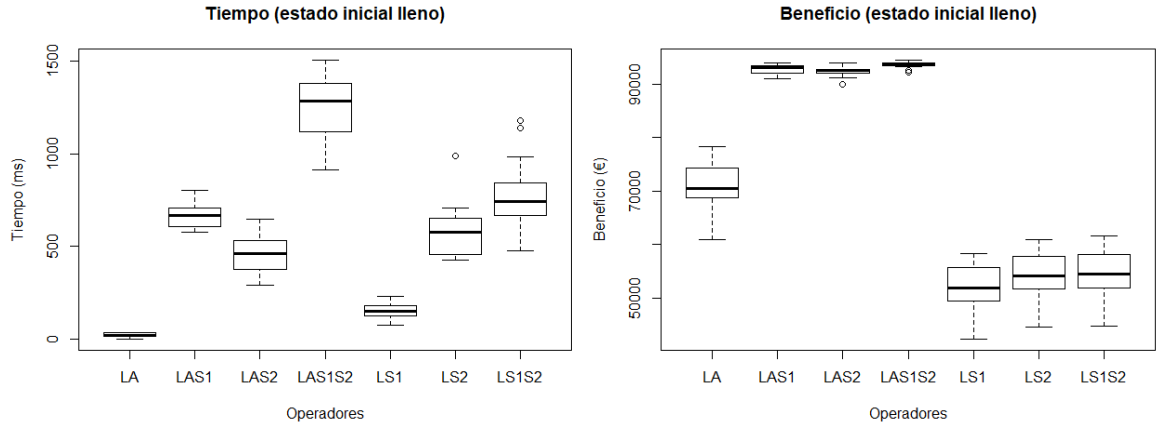
(Veinte pruebas para cada una de las combinaciones)

RESULTADOS

Una vez obtenidos todos estos datos, continuamos con el análisis de los mismos mediante **R**. Los resultados obtenidos son los siguientes:

*Nota: **vA** significa operador **Add** (con estado inicial **v**acío),*
***LAS1** operador **Add** + **Swap1** (con estado inicial **L**leno), etc*

ESTADO INICIAL LLENO



Medias	LA	LAS1	LAS2	LAS1S2	LS1	LS2	LS1S2
Beneficio (€)	70510	92787	92353	93688	51649	53883	54218
Tiempo (ms)	20.1	667.5	458.4	1248	149.2	578.2	774.8

Figura 2.1 Distribución del tiempo de ejecución y el beneficio (estado inicial lleno)

Como se puede apreciar en los boxplots anteriores, el **beneficio** para un estado inicial lleno es **considerablemente mejor si**, para empezar, usamos el **operador *Add***. Esto se debe a que, por las restricciones, hay peticiones que se siguen quedando fuera, pero que pueden ser añadidas igualmente en otro viaje. Por lo tanto, las tres combinaciones **LS1, LS2 y LS1S2 quedan descartadas**, por tener un **beneficio claramente inferior a todas las demás**, incluso necesitando más tiempo, como es el caso de LS1S2.

Por otra banda, **LA es muy rápido, pero no obtiene un beneficio tan grande** como en el caso de **LAS1**, **LAS2** o **LAS1S2**. Esto se puede deber a que muchas peticiones, aunque estén atendidas, no lo están de la forma más eficiente, provocando que se violen rápidamente las restricciones e impidiendo atender otras peticiones nuevas. Por lo tanto, **descartamos también LA**.

Así pues, aquí tenemos una comparación, ahora sin las opciones descartadas, para poder apreciar mejor la diferencia entre las demás:

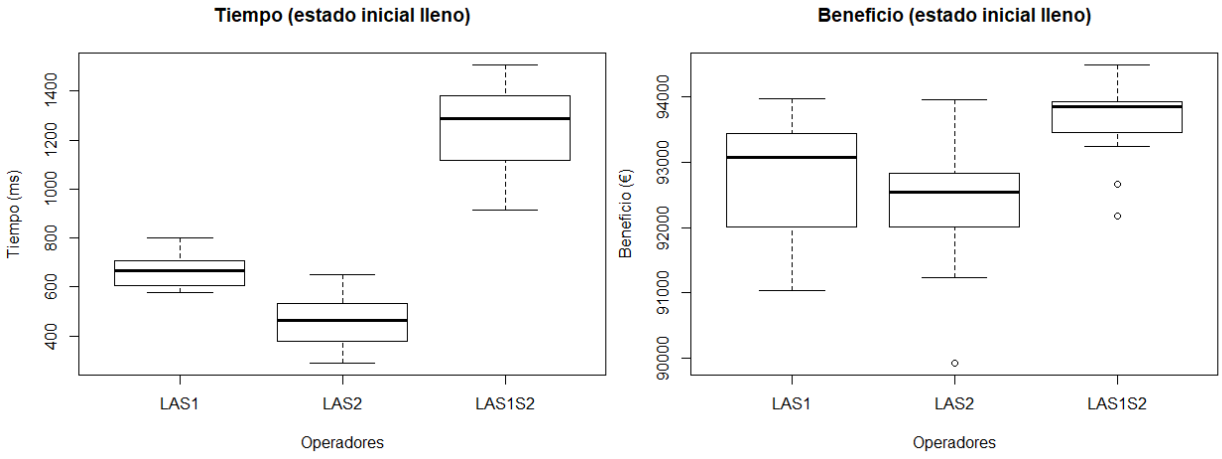
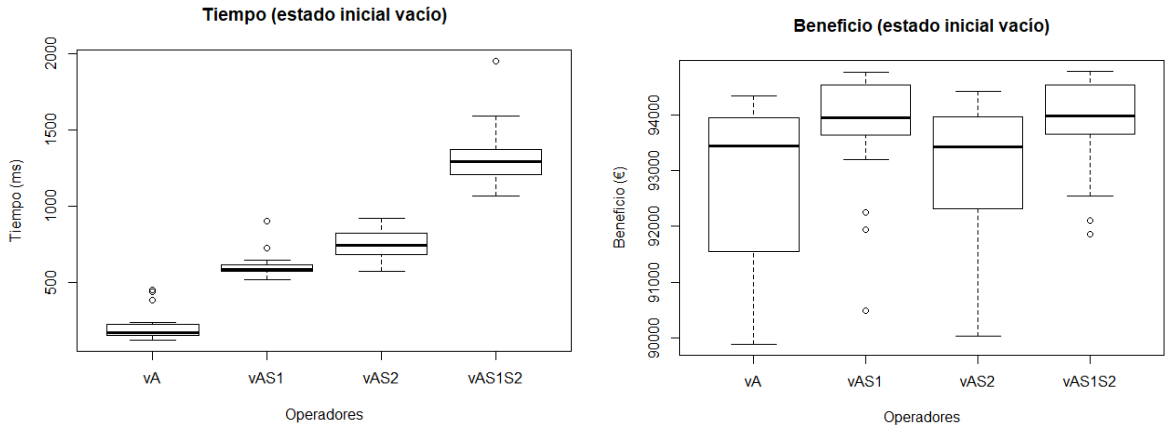


Figura 2.2 Distribución del tiempo de ejecución y el beneficio (mejores)

Estas tres combinaciones las tendremos en cuenta luego, al comparar los resultados con los del estado inicial vacío.

ESTADO INICIAL VACÍO



Medias	vA	vAS1	vAS2	vAS1S2
Beneficio (€)	9286	93743	93001	93848
Tiempo (ms)	212.4	608.8	750.9	1326

Figura 2.3 Distribución del tiempo de ejecución y el beneficio (estado inicial vacío)

Para el estado inicial vacío, obtenemos **beneficios muy similares** entre todas las combinaciones, aunque **la variación de vA y vAS2 es muy grande** y, por lo tanto, las descartamos.

Por otra banda, **vAS1 y vAS1S2 parecen tener un beneficio un poco más alto** y muy parecido, pero **AS1S2 tarda** un poco **más** en ejecutarse (casi un segundo más). Analizamos los resultados de ambas combinaciones para ver hasta qué punto aportan un beneficio parecido:

data: V_AS1$Benef and V_AS1S2$Benef
t = -0.34224, df = 36.118, p-value = 0.7342
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval: -728.5348 518.1348
sample estimates: mean of x mean of y
93742.8 93848.0

El p-valor nos indica que no hay suficiente evidencia para poder rechazar la hipotesi nula y que, por lo tanto se podría afirmar que **no hay una diferencia significativa entre ambos beneficios**. A pesar de esto, **vAS1 es considerablemente más rápido** (las medias son 608.8 ms y 1326.15 ms y, de hecho, si hacemos lo mismo pero con el tiempo, obtenemos un p-valor de 5.558e-14) y, por lo tanto, **descartamos** la opción **vAS1S2.**

RESULTADOS FINALES

Aquí tenemos una comparación para las combinaciones que hemos escogido como las mejores por su relación beneficio/tiempo de ejecución: **vAS1**, **LAS2** y **LAS1**.

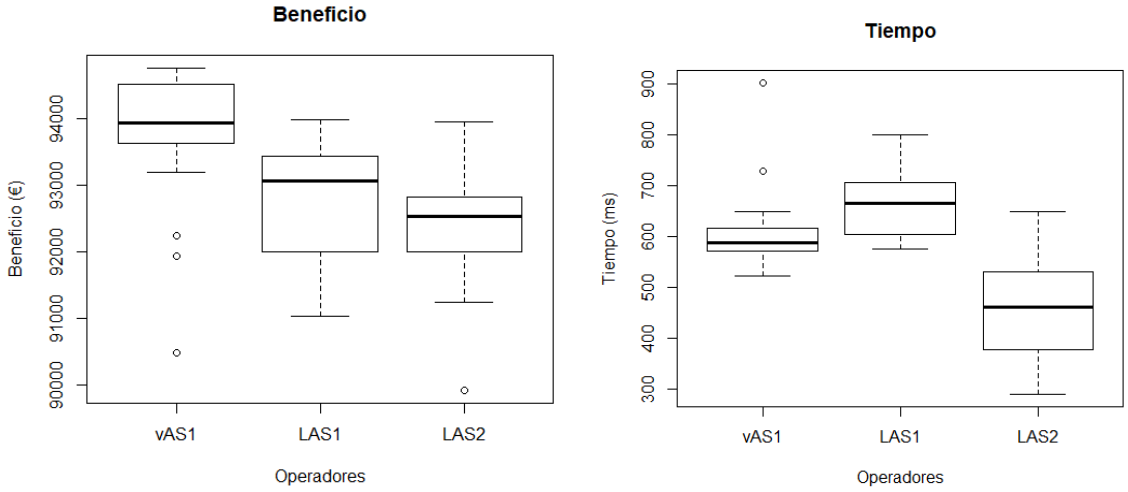


Figura 2.3 Distribución del tiempo de ejecución y el beneficio (mejores)

Finalmente, nos decidimos por **vAS1** (es decir, estado inicial **vacío** + **Add** + **Swap1**) ya que, en general, **da el mayor beneficio y es bastante rápido** teniendo en cuenta que LAS1 tarda más y aporta menos beneficios.

Así pues, a partir de ahora usaremos la siguiente combinación:

- **Estado inicial:** Vacío
- **Operadores:** Add y Swap1

Al final, ha habido mucha diferencia entre combinaciones (rechazamos la hipótesis nula) y la mejor combinación ha sido, como esperábamos antes de hacer estos experimentos, una con el estado inicial **vacío** y, al menos, ***Add***.

2.2 Experimento 3

En este experimento, vamos a determinar **qué parámetros dan mejor resultado para Simulated Annealing** en un escenario con It: 10.000, Stiter: 100, cd:10, gas:100.

RESULTADOS

La media de beneficio después de realizar 20 experimentos con seeds distintas para cada combinación **lambda/k** es la siguiente:

Media		k			
		1	**5**	**25**	**125**
Lambda	**1**	92232,4	91992	91987,2	92238,4
	0,01	95883,2	95940,8	95778,8	95973,6
	0,0001	95818	95768	95622,8	92518

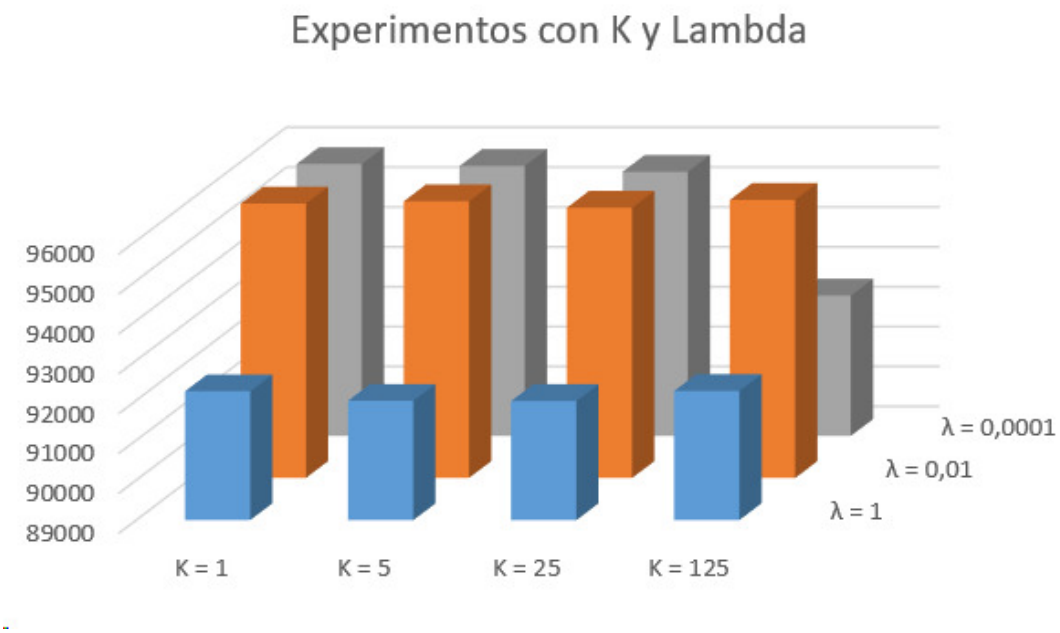


Figura 2.4 Beneficio según los parámetros del Simulated Annealing

Al final, decidimos usar para el resto de experimentos la combinación **lambda= 0,01 y k= 125**, ya que es la que en general da **mejores resultados**.

2.3 Experimento 4

En este experimento, estudiaremos cómo **evoluciona el tiempo de ejecución** para hallar la solución **para valores crecientes de los parámetros del problema** siguiendo la proporción 10:100 para centros y gasolineras. Comenzaremos con 10 centros de distribución e incrementaremos el número de 10 en 10 hasta que se vea la tendencia. Usaremos el algoritmo de *Hill Climbing* y *Simulated Annealing* y la misma función heurística que antes.

RESULTADOS

Primero, probamos con ***Hill Climbing*** y vimos que **a partir de 30 gasolineras el algoritmo se estaba muchisimo tiempo** (lo paramos después de 5 minutos):

n Centros	Tiempo (ms)
10	1,4
20	8,9
30	94,6
40	> 5 min

Con *Simulated Annealing,* hicimos veinte experimentos, en los que el número de centros iba aumentando de 10 en 10 hasta 500 centros (y manteniendo la proporción 10:100 con el número de gasolineras).

La media de tiempo de esos 20 experimentos en función del número de centros es la siguiente:

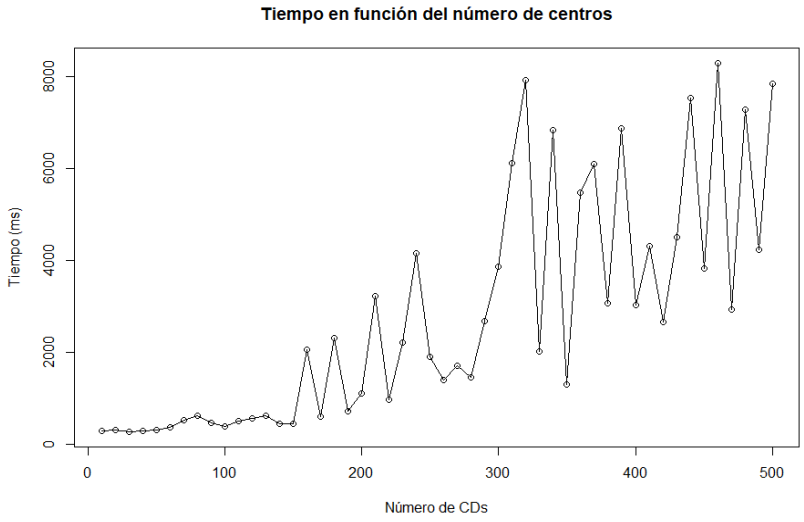


Figura 2.5 Tiempo en función del número de centros con Simulated Annealing

Como podemos apreciar, el **tiempo de ejecución tiende a aumentar,** pero hay **muchas excepciones**.

2.4 Experimento 5

En este experimento queremos ver **qué pasa si mantenemos el número de camiones, pero reducimos el número de centros a la mitad**. En un principio, creíamos que sería peor para el beneficio, porque los camiones estarían menos repartidos, así que habría gasolineras que se quedarían lejos, aunque no estábamos seguros.

- **Observación:** Puede haber un cambio en el beneficio y tiempo de ejecución si mantenemos los mismos camiones pero reducimos el número de centros a la mitad.
- **Planteamiento:** Probamos dos variantes del problema, en ambas el número de gasolineras es 100; en una tenemos 10 CDs y un camión en cada centro y en la otra 5 CDs y dos camiones en cada uno.

- **Hipótesis:** Las dos variantes tienen el mismo beneficio y kilómetros recorridos (H0) o hay diferencia.
- **Método:**
    - Usaremos 20 semillas aleatorias para el experimento.
    - Usaremos esas 20 semillas con un escenario en el que el número de centros de distribución es 10, hay un camión en cada centro y el número de gasolineras es 100.
    - Usaremos esas 20 semillas con el escenario alternativo en el que el número de centros de distribución es 5, hay dos camiones en cada centro y el número de gasolineras es 100.
    - Usaremos el algoritmo de Hill Climbing.
    - En cada ejecución registramos 2 valores: el **beneficio obtenido** (calculado a partir del beneficio que aporta atender las peticiones menos el coste de los kilómetros totales) y los **kilómetros recorridos**.

RESULTADOS

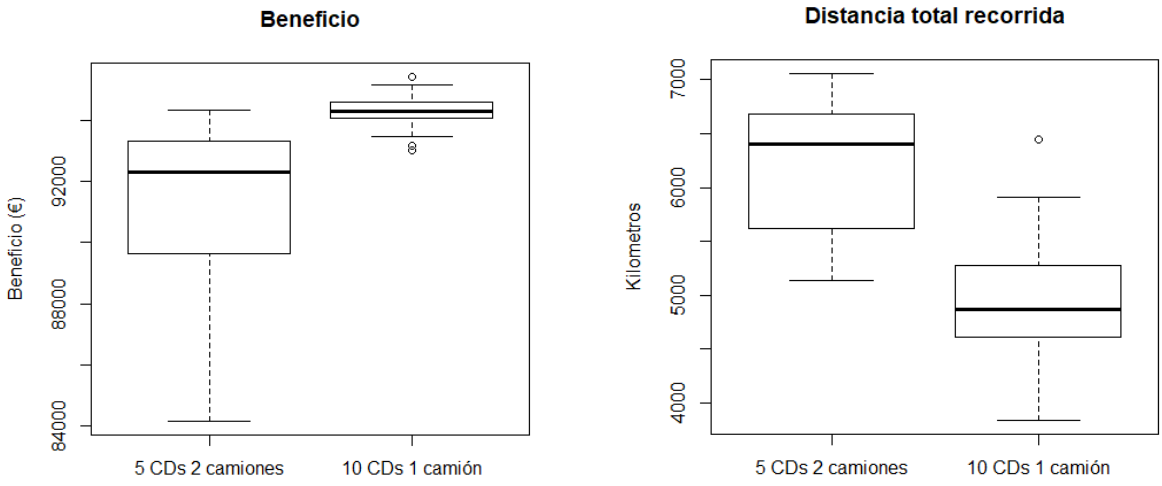


Figura 2.6 Distribución del beneficio y la distancia total recorrida

Medias	5 CDs 2 Camiones	10 CDs 1 Camiones
Beneficio (€)	91341	94266
Distancia Total (km)	6208	4933

Como se puede ver en los boxplots, el **beneficio**, en general, es **mayor** si tenemos **1 camión en cada centro** de distribución y, además, **realiza menos kilómetros**.

Así que, como esperábamos en un principio, sí ha habido una diferencia (rechazamos la hipótesis nula) y es mejor tener un camión en cada centro ya que se obtienen más beneficios haciendo menos kilómetros.

2.5 Experimento 6

En este experimento estudiaremos **cómo afecta al número de peticiones servidas el aumentar el coste por kilómetro**.

En un principio, creíamos que ir doblar el coste haría que se redujera el número de peticiones, ya que el número de peticiones que "salen a cuenta" atender se reduciría debido al alto precio de viajar.

- **Observación:** Puede haber un cambio en el número de peticiones servidas el aumentar el coste por kilómetro.
- **Planteamiento:** Vamos doblando el coste del kilómetro (inicialmente 2) para ver qué ocurre.
- **Hipótesis:** El número de peticiones no varía independientemente del coste del kilómetro (H0) o sí lo hace.
- **Método:**
    - Empezaremos con un coste por kilómetro de 2 euros y lo iremos doblando.
    - Usaremos el algoritmo de Hill Climbing y el escenario del primer experimento (10 CDs, 10 Gas)
    - En cada ejecución registramos el número de peticiones atendidas

RESULTADOS

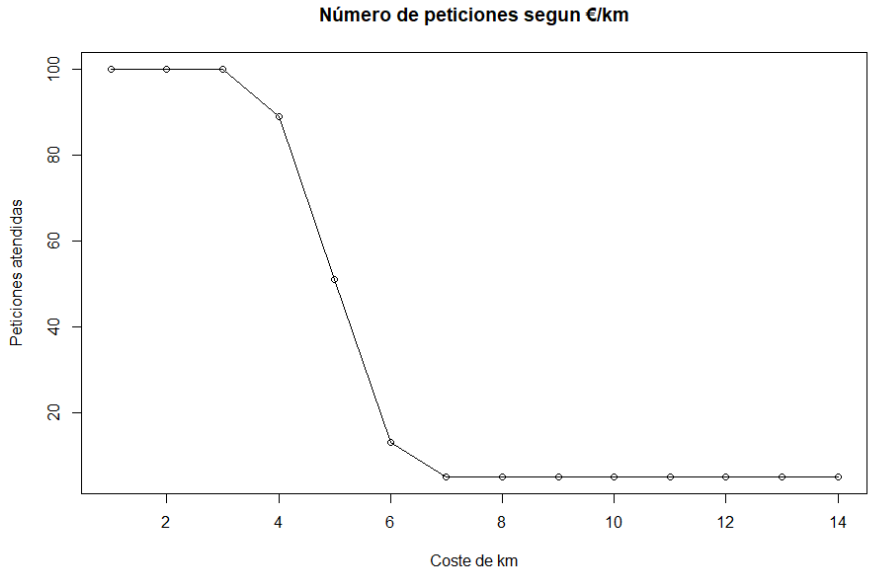


Figura 2.7 Número de peticiones atendidas en función del coste de viajar un km

Como podemos apreciar la figura anterior, el **número de peticiones atendidas disminuye al aumentar el coste por km**. Además, **a partir de cierto coste/km**, el número de peticiones atendidas **pasa a ser 0** (o no necesariamente 0, como veremos luego) así que, como esperábamos, el número de peticiones se ha ido reduciendo hasta llegar a 0 y, por lo tanto, rechazamos la hipótesis nula.

Nota: El gráfico està expresado con n = 2, 3, 4, 5… para poder verla bien.
En realidad, el coste del km es igual a 2^n

Aquí tenemos un ejemplo para otra prueba:

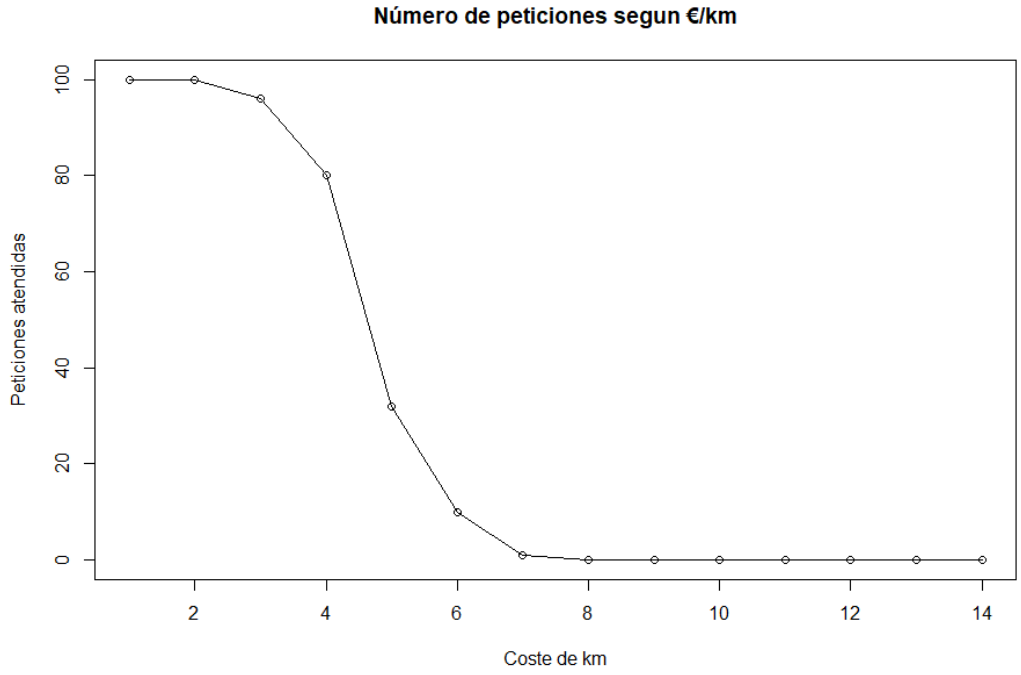


Figura 2.8 Número de peticiones atendidas en función del coste de viajar un km

Está gráfica es de uno de los primeros seeds que hemos usado: el 1234 (del experimento especial), y no entendíamos porque en lugar de quedarse en 0 se quedaba en 5. Al final, nos hemos dado cuenta de que éste seed se caracteriza por tener centros que están justo en el mismo lugar que gasolineras así que hay peticiones las cuales atiende sin ningún coste por km (en este caso, 5).

En conclusión, al **aumentar el coste del km**, **disminuye el número de peticiones atendidas**, hasta llegar al **número de peticiones** de las gasolineras que se encuentran a **distancia 0 de algún centro** de distribución.

2.6 Experimento 7

En este experimento veremos **qué efecto tiene el aumentar y disminuir los kilómetros que pueden recorrer las cisternas en un día** (aumentando/disminuyendo en 1 hora el tiempo de trabajo, es decir, en 80 km el límite de km).

En un principio, creíamos que **aumentar las horas supondría un aumento en el beneficio**, y al contrario al disminuir.

- **Observación:** Puede haber un cambio en el beneficio al aumentar y disminuir los kilómetros que pueden recorrer las cisternas en un día.
- **Planteamiento:** Realizaremos experimentos para ambos casos para ver qué sucede.
- **Hipótesis:** El beneficio no varía en función del número de horas de trabajo (H0) o sí lo hace.
- **Método:**
    - Usaremos el algoritmo de Hill Climbing.
    - Usaremos 20 semillas distintas
    - Para cada semilla veremos el beneficio con límite de 560, 640 y 720 km
    - En cada ejecución registramos el beneficio.

RESULTADOS

Medias	560	640	720
Beneficio (€)	93472	93586	93860

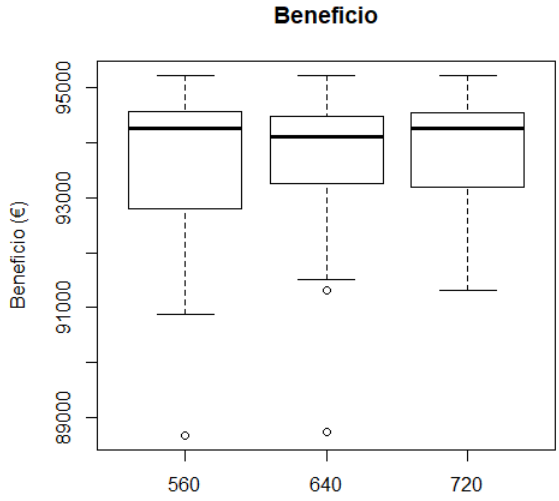


Figura 2.9 Beneficio según el límite de kilometros que puede realizar una cisterna en un día

Como podemos ver, tanto en la tabla como en el boxplot, la **variación es mínima**. De hecho, si realizamos el t-Student, nos sale un p-valor de 0,87, así que **no rechazamos la hipótesis nula**.

Los resultados han sido distintos a como esperábamos, porque pensábamos que, al variar el límite los camiones, éstos podrían atender más o menos peticiones. Al final, la

diferencia ha sido mínima, **seguramente debido a que el límite de viajes es un factor mucho más determinante que el límite de km**.

3. Trabajo de innovación

El tema que hemos escogido es Atlas, de Boston Dynamics.

3.1 Descripción

Atlas es uno de los últimos robot humanoides de Boston Dynamics. Su objetivo es obtener un robot humanoide capaz de sustituir a un humano en operaciones comunes, como por ejemplo caminar, abrir puertas, ascender cargas o conducir vehículos.

Un escenario de utilización es, por ejemplo, situaciones peligrosas para un ser humano, como en un incendio.

3.2 Distribución del trabajo

Los aspectos del robot que sobre los que queremos investigar son el algoritmo para el movimiento y de qué manera la I.A. puede ser utilizada para mejorar la interacción hombre-máquina. El aspecto económico del problema sería interesante también pero no hay mucho material al respecto.
Interacción hombre-máquina -> David Moreno Borràs
Algoritmo decisional para el movimiento -> Guillem Rodriguez Corominas / Marco Soldan

3.3 Lista de referencias

- https://www.bostondynamics.com/atlas [10/10/2017 17.30]
- https://www.researchgate.net/profile/Hongkai_Dai/publication/282477851_Optimization-based_locomotion_planning_estimation_and_control_design_for_the_atlas_humanoid_robot/links/5614501f08ae983c1b4073ac.pdf [21/10/2017, 18.37]
- http://ieeexplore.ieee.org/document/7063218/#full-text-section [20/10/2017 19.05]
- http://www.worldscientific.com/doi/pdf/10.1142/S0219843616500079 [20/10/2017 20.12]
- http://ai2-s2-pdfs.s3.amazonaws.com/5075/fb85bd1830b61d2aa41f65ea8ad7b31006f0.pdf [23/10/2017 14.20]
- https://s3.amazonaws.com/academia.edu.documents/41980899/Team_IHMCs_Lessons_Learned_from_the_DAR20160203-30232-1p7o2um.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1509322418&Signature=V0urVIbvyqeWFi3vGWdoeFizRfU%3D&response-content-disposition=inline%3B%20filename%3DTeam_IHMCs_Lessons_Learned_from_the_DARP.pdf [23/10/2017 14.50]