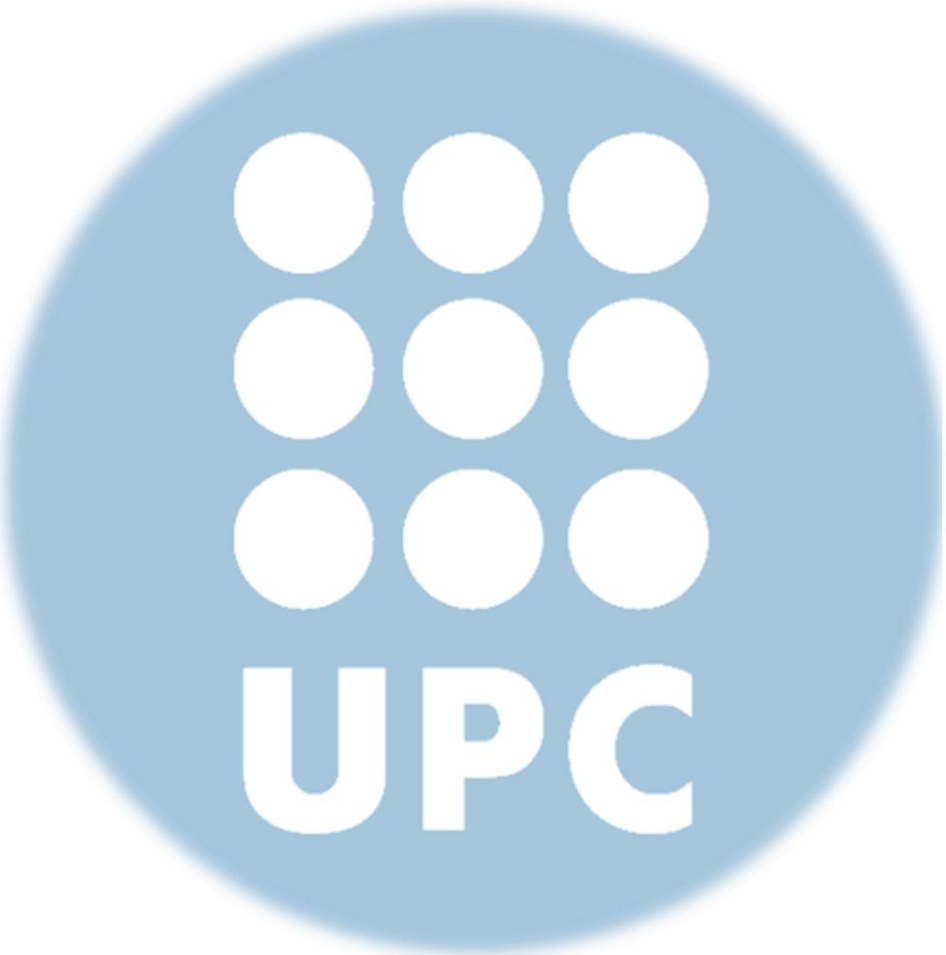
Q2 - 2011

Práctica de Búsqueda Local - IA



Carlos Vivas Abilahoud

Mireia Morales Bonet

Inteligencia Artificial

Q2 - 2011

Índice

1. Introducción

El objetivo de la práctica es resolver un problema mediante algoritmos de búsqueda local estudiados en clase.

Debemos demostrar que somos capaces de razonar sobre la naturaleza del problema y plantearlo como un problema de búsqueda local, solucionándolo con las librerías de algoritmos estudiadas en el laboratorio de la asignatura.

En nuestro caso se trata de resolver un problema de logística, al cual se enfrentan numerosas empresas en la actualidad, para hacer llegar mercancías a los centros de fabricación, venta o distribución desde los almacenes de materiales o productos.

Generalmente nos encontramos con el problema de que hay muy pocos almacenes y muchos centros de fabricación, venta o distribución. Éstos suelen realizar peticiones al almacén indicando qué productos y cantidades son necesarias y cuál es el plazo límite de entrega. Este plazo no tiene porque ser estricto, pero no cumplirlo puede suponer una pérdida o un aumento de costes.

1.1 Elementos del problema

Nosotros nos tenemos que poner en el papel de una compañía de transporte. La compañía que nos ha contratado tiene que gestionar las entregas desde un almacén a una serie de centros de producción.

El beneficio lo obtenemos por lo que nos pagan por hacer el transporte. Estos centros realizan un conjunto de peticiones de productos al almacén al final del día. Cada petición indica:

- el producto que se demanda
- la cantidad (en kilogramos)
- la hora límite de entrega.

Supondremos que en nuestro problema hay 6 centros de producción. La cantidad de las peticiones debemos asumir es un número entero de 100 a 500 kg. en múltiplos de 100.

El almacén tendrá preparadas las peticiones que se han de repartir en el día y nosotros tenemos que organizar el transporte. Para ello disponemos de un conjunto de camiones. Cada camión tiene una capacidad de carga. Supondremos que tenemos camiones capaces de transportar 500, 1000 y 2000 kg, a los que nosotros nos referiremos respectivamente como camiones **pequeños, medianos o grandes**.

Nosotros organizamos un horario para las entregas que supondremos que van desde las 8 de la mañana hasta las 5 de la tarde. La organización será de la siguiente manera:

- Para cada hora de entrega (de 8 a 17) y cada centro de producción (1 a 6) tendremos un camión.
- Este camión saldrá del almacén y llegará al centro de producción a la hora establecida.

Es decir, tenemos 60 transportes que programar. Asumiremos que tenemos camiones suficientes para realizar todos los transportes.

Para cada transporte deberemos decidir cual es la capacidad del camión que lo hace y qué peticiones transporta. Respecto a esas capacidades supondremos que tenemos un número específico de camiones de cada capacidad (en total sumarán los 60 camiones). De esta manera podremos experimentar la influencia del tipo de flota de camiones que tenemos en el problema.

Como hemos comentado, el beneficio lo obtendremos por el pago del transporte. La tabla de precios del envío de cada petición es la siguiente:

Peso	Precio
100 y 200 kg	peso euros
300 y 400 kg	1,5 × peso euros
500 kg	2 × peso euros

Es decir, si la carga es 200 o menor, el beneficio en precio corresponde al peso de la carga, es decir, 100€ o 200€.

En el caso de que una petición llegue con retraso con respecto a la hora indicada, el precio que cobraremos se reducirá en un 20% por cada hora de retraso (es decir, si el paquete se retrasa más de cinco horas nos tocará a nosotros pagar siguiendo la misma proporción).

Evidentemente el paquete puede llegar antes de hora, pero en este caso no nos pagan de más. Si una petición no se puede entregar en el día, el coste para nosotros será el 20% por cada hora de retraso hasta las 5 de la tarde, más el precio de entrega de la petición.

1.2 Definición del problema

Dada una lista de peticiones de los centros de producción, donde cada una indica el centro de entrega, la cantidad y la hora límite de entrega y dado que vamos a realizar 60 transportes (uno por cada combinación de horas y centros de producción):

- Asignar el número máximo de peticiones a los transportes que vamos a realizar
- Determinar la capacidad del camión que va a realizar cada transporte

1.3 Criterios de la solución

Para obtener y evaluar la solución usaremos los siguientes criterios y restricciones:

- Tenemos una cantidad establecida de camiones de cada capacidad que suma el número total de transportes a realizar.
- Tendremos dos criterios para evaluar la calidad de una solución:
- Maximizar la ganancia que obtenemos con los transportes
- Minimizar el valor absoluto de la diferencia entre la hora límite de entrega de la petición y la hora efectiva de entrega. Asumiremos que este valor para las peticiones no entregadas es la diferencia entre la hora límite de entrega y las 8 de la mañana del día siguiente.

1.4 Tareas a realizar

En definitiva, las tareas que debemos realizar para llevar a cabo la práctica son las que citamos a continuación:

- Implementar el problema de tal manera que se puedan generar problemas aleatorios. En este caso los elementos que varían son:

- El número total de camiones de cada capacidad (siempre habrá 60 camiones en total).

- El número de peticiones.

  - Los destinos, pesos y horas de llegada de las peticiones. Para generar los valores de los pesos y las horas supondremos que siguen distribuciones de probabilidad uniformes y que se puede indicar como parámetro las probabilidades para cada valor de cada variable. La probabilidad de los destinos la consideraremos equiprobable.

  - Definir e implementar la representación del estado del problema para poder ser resuelto utilizando las clases del AIMA. Pensad bien en la representación, ha de ser eficiente en espacio y en tiempo.

- Definir e implementar dos estrategias para generar la solución inicial.

- Definir e implementar la función generadora de estados sucesores. Esto implica decidir el conjunto de operadores para explorar el espacio de búsqueda. Deberéis pensar y evaluar diferentes alternativas de conjuntos de operadores y justificar la elección de uno de ellos para realizar los experimentos. Deberéis implementar la función generadora de manera diferente para Hill Climbing y Simulated Annealing para que se puedan comparar sus tiempos de ejecución tal como se explicó en clase de laboratorio.

  - Definir e implementar dos funciones heurísticas, cada una de ellas implementará uno de los criterios de calidad de la solución indicados en el apartado anterior.

2. Estructura

Para llevar a cabo la práctica hemos implementado clases en lenguaje Java para complementar las librerías AIMA proporcionadas en la clase de laboratorio de la asignatura.

A continuación definimos la estructura de estas clases, como se relacionan entre ellas y sus funcionalidades.

2.1 Clase Área

La clase que hemos definido como Área es la clase principal de nuestra estructura.
Las variables más importantes que contiene son las siguientes:

```
int numCamiones=60;
int numCentros=6;
int maxPeticiones = 1200; //6*10*(2000/100)
int camionesG, CamionesM, CamionesP;
Vector<Peticion> peticiones_t;
Estado est;
```

numCamiones: Número de camiones necesarios para implementar nuestro problema. Esta variable, como dice el enunciado, es un número constante cuyo valor es siempre 60.

numCentros: Número de centros a los que llevaremos las peticiones que forman nuestro problema. Del mismo modo que la variable anterior, el enunciado da su valor que será 6.

maxPeticiones: Es el número máximo de peticiones que puede generar el problema. Hemos decidido que su valor será 1200 ya que es el resultado de calcular su valor máximo.

CamionesG, Camiones M y CamionesP: Estas variables nos indican el número de camiones que hay para los tres tipos posibles, pequeños (capaces de cargar hasta 500 kg), medianos (capaces de transportar hasta 1000 kg) y grandes (pueden llegar a llevar hasta 2000kg de carga).

peticiones_t: Esta variable se corresponderá a un vector de peticiones. Más adelante se detalla qué es una Petición.

Estado: Esta es una clase muy importante de la que hablaremos más adelante.

Puesto que todas las variables están fijas o se tienen que generar aleatoriamente, solo necesitamos un parámetro en la creadora de esta clase:

```
public Area(int estrategia){

        this.generador(); //genera las peticiones y los camiones de forma aleatoria.
        this.asignar(estrategia);
        this.getEst().actualizarHeuristicos();
        this.getEst().calcularHeuristicos();
```

Esta clase además contiene un método llamado generador que nos permite iniciar las variables tomando valores aleatorios en los casos necesarios:

```
public void generador() {

  this.peticiones_t= new Vector<Peticion>();
  SecureRandom rnd = new SecureRandom();
  this.camionesG=rnd.nextInt(60);
  this.CamionesM=rnd.nextInt(60-this.camionesG);
  this.CamionesP=60-this.camionesG-this.CamionesM;

  for(int i = 0; i < this.getRandPeticiones(); ++i) {
 Peticion actual= new Peticion(this.getRandCentro(),
this.getRandCantidad(),this.getRandHoraLim());
            peticiones_t.add(actual);
  }
```

Como podemos observar, en esta función se generan aleatoriamente el número de camiones de cada tipo de carga teniendo en cuenta que el máximo siempre es 60. Por otro lado iniciamos el vector de Peticiones con un identificador de centro, una cantidad de carga y una hora límite aleatorias para cada posición de dicho vector.

El parámetro “estrategia”, que es pasado al llamar al método de la creadora, será una opción escogida por el usuario que será enviado desde la clase Main. Esta variable “estrategia" indica cómo se desea comenzar el problema.

Otro método muy importante de la clase Área, es la función *asignar(int estrategia)*, ésta es la encargada de crear nuestro estado inicial del problema. Este estado puede ser generado de dos maneras como nos indica el enunciado de la práctica, una manera simple y una compleja, es por esto que necesitamos pasarle por parámetro la variable “estrategia” comentada antes. Se profundiza más detalladamente en el apartado ***2.4 Estado***.

2.2 Petición

Como hemos comentado, una estructura importante de nuestra implementación son las *Peticiones*, que sería la estructura que define los pedidos de los centros de producción con los datos necesarios de las entregas. Esta clase contiene las siguientes variables:

```
private int centro;
private int cantidad;
private int horaLimit;
```

centro: Esta variable es un identificador del centro que ha realizado la petición.

centidad: Indica la cantidad de carga demandada.

horaLimit: Es la hora límite de entrega de la petición.

2.3 Asignacion

Siguiendo un orden jerárquico de nuestra estructura ahora tocaría explicar en qué consiste nuestra clase *Asignacion*.

Esta clase representa la asignación que se hace a un camión con una serie de peticiones. Las variables más importantes que la componen son las siguientes:

```
public  int camion;
public Vector<Peticion> cargas;
int beneficio;
int tiempoAbs;
int horaSalida;
int cargaTotal;
```

camion: Esta variable no es más que un identificador de tipo de camión, es decir, indica la carga que puede transportar como máximo.

cargaTotal: Es la carga resultante de sumar todas las cargas del vector de peticiones de esta asignación, es decir, cuanto de cargado en total se encuentra el camión.

cargas: Esta variable es un vector que contiene un listado de peticiones que se han asignado a un camión en concreto.

beneficio: Esta variable se utilizará para uno de los heurísticos donde buscaremos maximizar el beneficio total.

tiempoAbs: Esta variable será necesaria también para un heurístico donde tendremos que minimizar las horas de retraso.

horaSalida: Esta variable indica la hora en que realmente se realiza la entrega.

En esta clase además se recalculan los heurísticos para cada asignación siempre que nos movemos por el escenario de soluciones. Esta función se explica con detalle en el apartado ***2.4.3 Heurísticos***.

2.4 Estado

Esta clase representa los diferentes estados por los que puede pasar el problema. Es una de las partes más importantes de los algoritmos de búsqueda local ya que contiene información de todos los elementos que forman el problema en todo momento.

Cada estado contiene las siguientes variables:

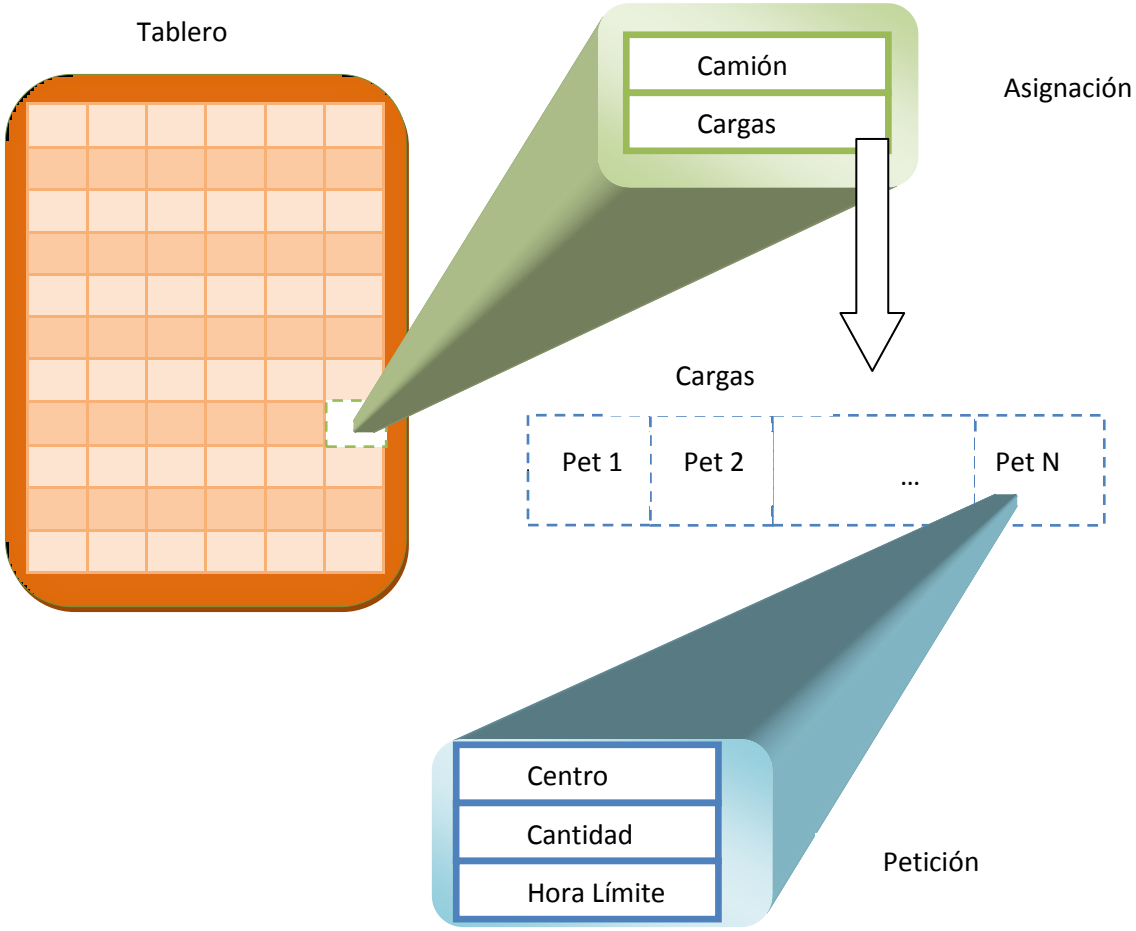
```
Asignacion [][] tablero= new Asignacion[11][6];//Horas x Centros
int beneficioTotal;//heuristico 1
int retrasos; // heuristico 2
```

beneficioTotal: Esta variable la utilizaremos para el primer heurístico, donde intentaremos maximizar el beneficio obtenido.

retrasos: De manera similar a la variable anterior, esta variable la utilizaremos para el segundo heurístico, donde el objetivo será minimizar el número de horas en las que se ha entregado un pedido después de su hora límite.

tablero: Cada estado tiene una matriz de asignaciones. Las filas representan las horas posibles entre las 8 de la mañana y las 5 de la tarde, hay una hora más donde se asignarán todos los pedidos que no hayan podido ser entregados en el día y deban ser entregados a las 8 de la mañana del día siguiente. Nosotros denominaremos a los camiones de esta hora ficticia "camiones infinitos" ya que no tienen restricción de carga. Y las columnas del tablero representarían los 6 centros de producción donde se tienen que realizar las entregas.

Llegamos a este punto, para facilitar la comprensión de la estructura general de nuestra implementación, mostramos un esquema donde se puede ver la relación de las diferentes clases que forman los estados.



2.4.1 Estado Inicial

Una parte muy importante del estado es como generamos la solución inicial. Esta solución se calcula en la clase Area mediante una de las dos estrategias escogidas por el usuario que obtenemos al iniciar el programa. Esta estrategia es básicamente un número entero que solo puede tomar dos valores.

Número	Estrategia
1	Estrategia simple
2	Estrategia compleja

Estrategia Simple

Una manera de generar el estado inicial sería asignar los pedidos a nuestros "camiones infinitos" solo teniendo en cuenta el centro de producción. A continuación mostramos la parte del código correspondiente a estas asignaciones.

```
while(numPetAsig < peticiones_t.size()){

        /*Mientras no hayamos asignado todas las peticiones
          Asigna las peticiones en el camion infinito*/

        Peticion pet= peticiones_t.get(numPetAsig);
        centro= peticiones_t.get(numPetAsig).getCentro();
        cantidad= peticiones_t.get(numPetAsig).getCantidad();
        this.est.tablero[10][centro].cargas.add(pet));
        this.est.tablero[10][centro].cargaTotal+=cantidad;
        this.est.tablero[10][centro].horaSalida=18;

        numPetAsig++;

}
```

Como podemos ver en el código adjunto, se asignan las peticiones siempre a la fila que corresponde a los camiones infinitos. Únicamente nos fijamos en que se corresponda a la columna del centro que corresponde a cada petición que queremos asignar.

Es un estado inicial muy ineficiente ya que siempre estaremos lejos de una solución donde maximicemos los beneficios y ocurran el mínimo número de retrasos.

Estrategia Compleja

La otra manera de crear un estado inicial que hemos implementado, es asignar los camiones aleatoriamente y después, una vez ordenada la lista de pedidos por horas, para cada pedido, ir asignándolos empezando por las primeras horas en caso de que quepan, si no caben intentar asignarlos a la hora siguiente y así hasta llegar al camión infinito si no puede asignarse a ninguna hora para el centro que le corresponde.

```
mergeSort(peticiones_t);
while(numPetAsig != peticiones_t.size()){

    asignaPeticion((Peticion)peticiones_t.get(numPetAsig).clone());
    numPetAsig++;
}
```

Básicamente recorremos el vector de las peticiones generadas y llamamos al método asignaPeticion, al cual le pasamos como parámetro un clon de la petición a asignar.

A continuación podemos ver el funcionamiento de este método.

```
public void asignaPeticion(Peticion p){

    boolean asignado=false;
    int i = 0;
    int cp = p.getCentro();
    int ct = p.getCantidad();

    while (!asignado){

            //Si cabe en la posicion, se asigna.

            if(this.est.tablero[i][cp].cargaTotal+ct<=this.est.getTablero()[i][cp].camion){

                    this.est.tablero[i][cp].cargas.add(p);
                    this.est.tablero[i][cp].cargaTotal+=ct;
                    this.est.tablero[i][cp].horaSalida=i+8;
                    asignado=true;

            }
    i++;
    }
}
```

Como podemos observar, cada petición se asigna a la primera posición de la matriz, correspondiente a la columna de su centro, lo antes posible. Miramos si hay capacidad suficiente para asignar la petición y vamos recorriendo la columna hasta que encontremos donde cabe. Finalmente en la posición donde quepa, actualizamos las variables de la asignación con los datos necesarios.

Esta solución inicial es muy buena ya que al tener el vector de peticiones ordenado por horas, siempre intentaremos asignar a las primeras horas los que antes deben ser entregados, reduciendo así el número de retrasos considerablemente.

2.4.2 Operadores

En esta clase *Estado* también tenemos definidos los operadores. Los operadores son aquellas funciones que hemos implementado para hacer los cambios de un estado a otro para movernos por el espacio de soluciones.

Los explicaremos con más detalle en el apartado *3. Operadores.*

2.4.3 Variables para los Heurísticos

Otra característica importante de la clase *Estado* es que realiza los cálculos necesarios para actualizar las variables que necesitaremos para calcular los heurísticos.

Cada vez que generamos un sucesor, en el posible estado nuevo tenemos que calcular los heurísticos de la nueva solución.

```
public void calcularHeuristicos(){

    this.beneficioTotal=0;
    this.retrasos=0;
    this.actualizarHeuristicos();
    for (int i = 0; i<11;i++){
            for(int j = 0;j<6;j++){
                    //obtiene los beneficios de las asignaciones
                     this.beneficioTotal+=this.getTablero()[i][j].beneficio;
                     this.retrasos+=this.getTablero()[i][j].tiempoAbs;
             }
    }
}
```

Esta función actualiza el valor de los heurísticos para el nuevo estado y actualiza las variables generales *(beneficioTotal* y *retrasos)*.

El trozo de código que se corresponde con *actualizarHeuristicoses* el siguiente:

```
public void actualizarHeuristicos(){
     for (int i =0;i<11;i++){
         for (int j = 0; j<6;j++){

             this.getTablero()[i][j].actualizaHeuristicoAsignacion();
        }
     }
}
```

Como podemos ver, esta función recorre toda la tabla de asignaciones y para cada asignación llama a su método de actualizar los heurísticos.

En el método *actualizarHeuristicosAsignación* se encuentra toda la parte densa de la actualización de estas variables.

A continuación la explicamos por partes:

```
if(this.camion<2001){
//calculamos primero el beneficio
    for(i=0;i<this.getCargas().size();i++){

            beneficio_temporal=this.getCargas().get(i).getCantidad();
            if(beneficio_temporal==500)
            beneficio_temporal=1000;

else if(beneficio_temporal>200)
{
    beneficio_temporal*=1.5;
}
```

En primer lugar comprobamos si el camión no es del tipo infinito mirando que su capacidad sea menor que 2001. En caso de no serlo, calculamos el beneficio que supone respecto a su carga como nos indica el enunciado.

```
hora_lim=this.cargas.get(i).getHoraLimit();
    //calculamos la diferencia horaria
    if(this.horaSalida>hora_lim){

            diff_tiempo= this.horaSalida-hora_lim;
            beneficio_temporal-= (beneficio_temporal*0.2*diff_tiempo);

    }
    else{
            diff_tiempo=hora_lim-this.horaSalida;
    }
this.beneficio+=beneficio_temporal;
this.tiempoAbs+=diff_tiempo;
```

Después, en caso de que la hora de salida sea mayor que la límite, es decir, en caso de que haya un retraso, calculamos la diferencia de horas entre la hora límite de la petición y la que realmente se hace la entrega y actualizamos el beneficio restando un 20% por hora de retraso. Por último suma al total de los beneficios y de retrasos los valores obtenidos.

```
else {
    for(i=0;i<this.getCargas().size();i++){

            beneficio_temporal=this.getCargas().get(i).getCantidad();
    if(beneficio_temporal==500)
                    beneficio_temporal=1000;
            else if(beneficio_temporal>201)
       {
                    beneficio_temporal*=1.5;
        }
```

Después miramos para los casos de los camiones infinitos. Calculamos nuevamente los beneficios del mismo modo que antes.

```
    hora_lim=this.cargas.get(i).getHoraLimit();
    //en este caso siempre está al final, en el camion infinito
    diff_tiempo= 17-hora_lim;
    beneficio_temporal-= (beneficio_temporal*0.2*diff_tiempo);
```

Sin embargo para calcular la diferencia de tiempo solo tendremos en cuenta la hora límite de la petición, ya que estamos en el caso de camiones infinitos, donde siempre entregaremos a las 8 de la mañana del día siguiente, y restamos esta hora a 17 que es la última hora que puede haber entregas. Por último restamos el beneficio debido al retraso, restando el 20% por hora.

```
    this.beneficio+=beneficio_temporal;
    this.tiempoAbs+=diff_tiempo;
```

Finalmente actualizamos las variables que contienen los valores de beneficio y retrasos absolutos del estado.

2.5 Heurístico1

Tanto esta clase como las dos siguientes, implementan la clase aima.search.framework.HeuristicFunction. Es uno de los heurísticos que hemos implementado para garantizar que se cumplen los criterios de búsqueda. Es explicado con detalle en el apartado ***5. Heurísticos.***

2.6 Heurístico2

Así como la clase anterior, esta clase implementa la clase aima.search.framework.HeuristicFunction. Es otro de los heurísticos que hemos implementado para garantizar que se cumplen los criterios de búsqueda. Para mayor detalle mirar apartado ***5. Heurísticos.***

2.7 Heurístico3

Este es el tercer heurístico que hemos implementado para garantizar los criterios de búsqueda de nuestro problema e implementa la clase aima.search.framework.HeuristicFunction. Para más detalle mirar el apartado ***5. Heurísticos.***

2.8 Goal

Esta clase implementa la clase de AIMA. Como nos indica el enunciado de la práctica tenemos que devolver siempre falso.

2.9 Sucesores

Esta clase es la encargada de generar los diferentes estados sucesores posibles desde el estado actual, que será el que le pasemos como parámetro. Esta clase implementa la clase aima.search.framework.SuccesorFunction de AIMA.

Es una clase muy importante de la práctica y por eso le hemos dedicado el apartado *4.Generación de sucesores* para explicarla con más detalle.

3. Operadores

Los operadores son los encargados de modificar un estado con la finalidad de convertirlo en otro y así poder movernos por el espacio de soluciones. Depende del problema, un operador puede hacer que un estado que no era solución pase a serlo, y uno que lo era, deje de serlo. En ocasiones, estos operadores generan un espacio de estados muy grande, y por muchos de los estados que pueden generar no vale la pena pasar.

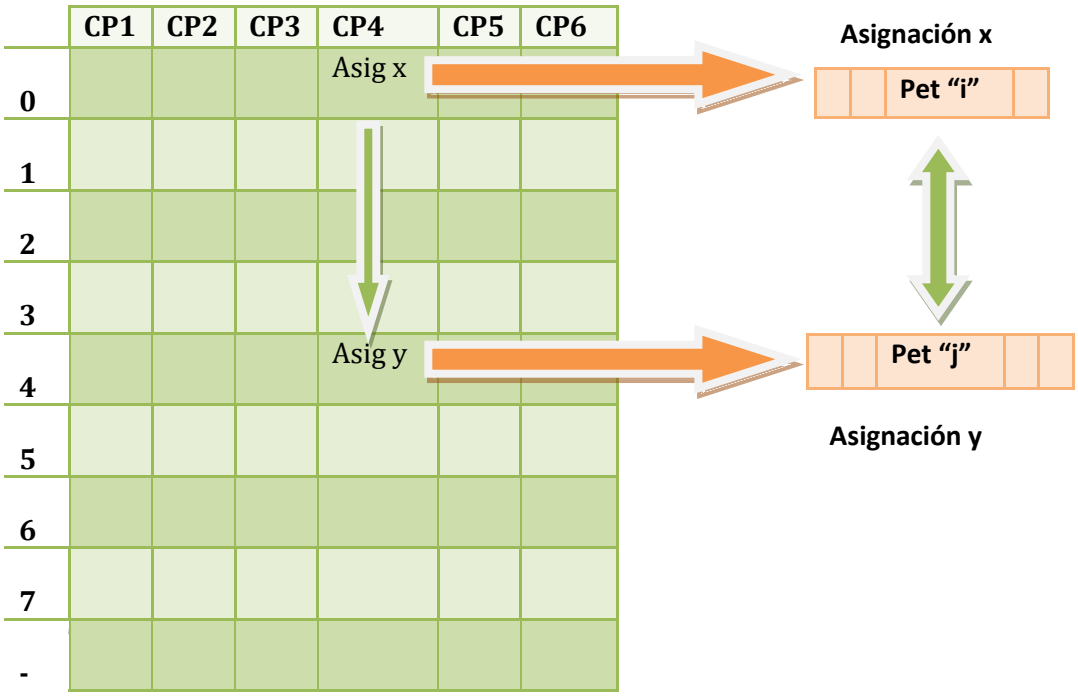
Por lo tanto los operadores tienen la responsabilidad de mover la búsqueda local por el espacio de estados, intentando encontrar las soluciones más óptimas. Es por esto que los operadores son un punto muy importante a la hora de obtener un sistema basado en la búsqueda local, que sea capaz de resolver nuestro problema.

En nuestro caso hemos decidido que el operador siempre se mueva por el espacio de estados en que todos los estados son solución, es decir, por el espacio de soluciones. Estos estados son aquellos en que se reparten siempre todos los paquetes, es decir, no queda ninguna petición por asignar a un camión y a una hora determinada. Por esta razón, nuestro estado inicial es ya una solución. Inicialmente habíamos implementado un operador que movía las peticiones de sitio, pero sin intercambio. Esto nos ocasionaba problemas ya que en las asignaciones donde teníamos los camiones llenos no se producían cambios por tanto decidimos modificarlo y sustituirlo por otro que no solo pudiera mover una carga de sitio si no que también planteara la posibilidad de intercambiarse con otra petición del mismo centro. El operador inicial se sigue encontrando en el código aunque no se utiliza.

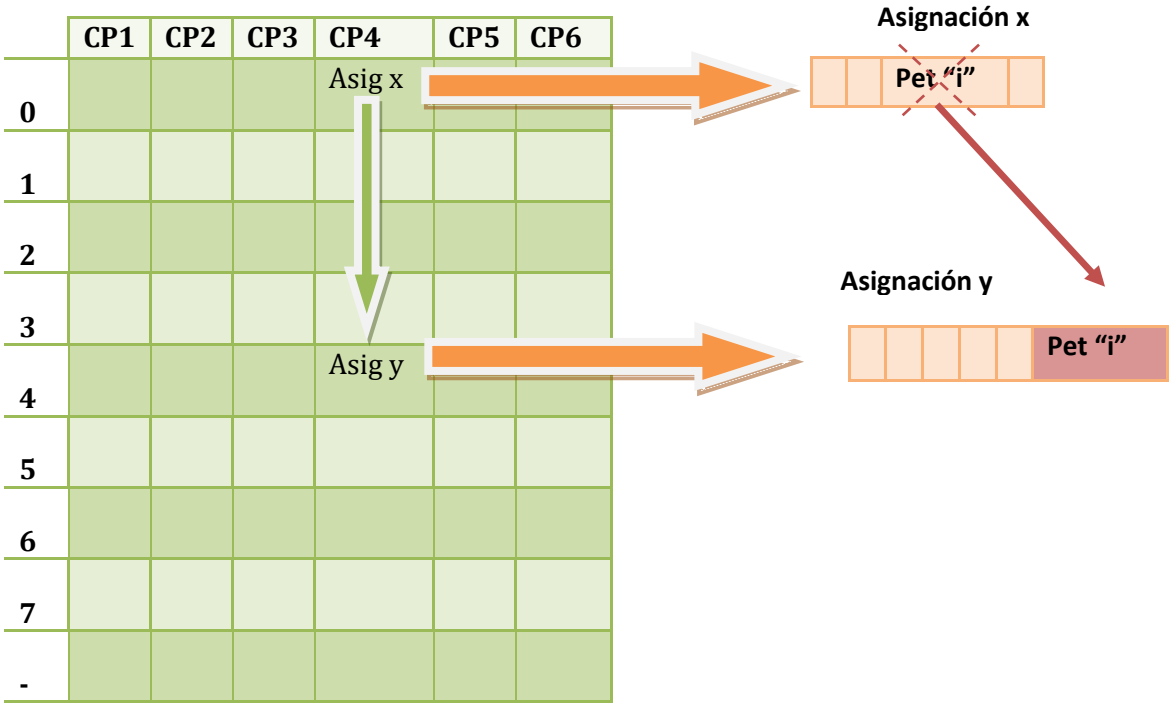
Los operadores definitivos que hemos implementado son:

- **Mover petición**: Intercambia de sitio dos peticiones de dos asignaciones con diferente hora de entrega pero el mismo centro de producción, o bien, cambia de sitio una petición de una asignación a otra del mismo centro aunque la asignación futura no tenga ninguna petición en el vector *cargas.* En conclusión, o lo intercambia o solo lo cambia de sitio.

La siguiente imagen mostraría un intercambio de peticiones de diferentes asignaciones para un mismo centro de producción pero diferentes horas.

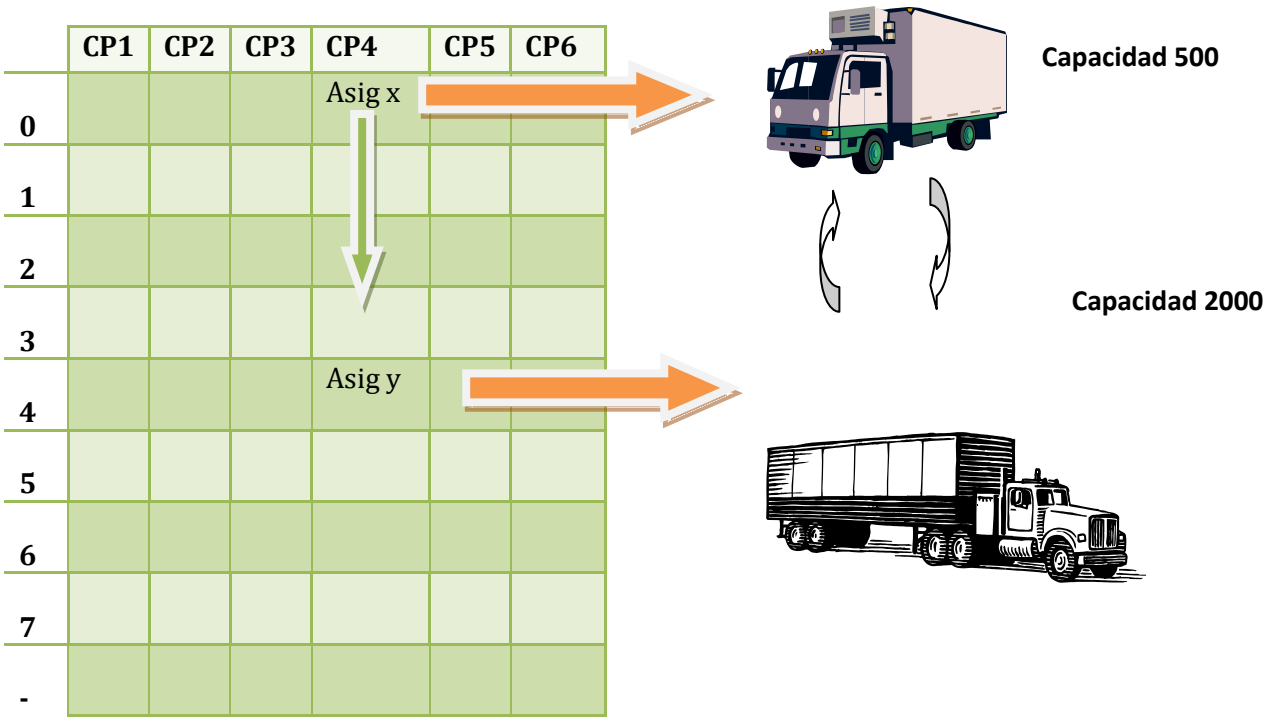


La siguiente imagen muestra como una petición se mueve a otra asignación sin intercambio. El hecho de no haber intercambio puede ser porque no quepa la petición en la otra asignación, o porque estuviera vacía.



- **Intercambiar Camión**: Intercambia peticiones y además camiones, es decir, las cargas como en el operador anterior y además la capacidad de carga de las asignaciones.

Gráficamente sería una mezcla del operador anterior y de la imagen siguiente donde se mostraría el intercambio de cargas:



Este intercambio de camiones, no es más que cambiar la capacidad de una asignación de un centro a una hora determinada, buscando así que las horas más demandadas puedan ser realizadas sus peticiones teniendo la menor pérdida posible por retrasos.

3.1 Operador intercambiarCamion

Ahora explicaremos como hemos implementado este operador.

```
public boolean intercambiarCamion(int x1, int y1, int x2, int y2){

        int camion1=this.tablero[x1][y1].camion;
        int camion2=this.tablero[x2][y2].camion;
        boolean b = false;
```

A este operador le pasaremos como parámetro las posiciones del tablero a intercambiar. Las variables *x1* y *y1* se corresponden, respectivamente, a la fila y columna de un camión1, y *x2* y *y2* se corresponden con la fila y columna del camión2.

```
if(camion1!= camion2){
    if(this.tablero[x1][y1].cargaTotal<=camion2 &&
    this.tablero[x2][y2].cargaTotal<=camion1 && x1<10 && x2<10 )
            {
```

```
                //Si  puede intercambiarse la capacidad y quede todo dentro
                        this.tablero[x1][y1].camion=camion2;
                        this.tablero[x2][y2].camion=camion1;
                        b=true;
                }
        }
```

A continuación comprobamos que los dos camiones a intercambiar no son de la misma carga, ya que sería un cambio improductivo y estaríamos en dos estados equivalentes. También debemos comprobar que la carga total de la asignación futura es menor que la capacidad de cada camión, ya que si no se podría producir el intercambio, y finalmente restringimos que los camiones a intercambiar no sean los infinitos, es decir, no sean los de la fila 10, ya que asignaríamos a una asignación una capacidad demasiado alta.

Estos operadores serán llamados desde la case sucesores, donde se quiere obtener una lista de todos los posibles estados que se pueden obtener a partir del actual aplicando los operadores. Para saber si un operador realmente ya llegado a producir un cambio en el estado actual, hemos añadido a estos operadores una variable booleana que será *true* en el caso de haber realizado el cambio.

```
this.calcularHeuristicos();
    return b;
```

Por último el operador llamará a calcularHeurísticos, ya explicado en el apartado *2.4.3 Cálculos para los heurísticos*, y retorna la variable booleana.

3.2 Operador moverPeticion

A continuación explicamos cómo hemos implementado este operador.

```
public boolean moverPeticion(int x1, int y, int x2, int peticion1, int peticion2){
                Peticion p=null;
                Peticion p2=null;
                int cargaTotal1;
                int cargaTotal2;

                boolean b = false;
```

A este operador le pasaremos como parámetro las filas de la posición antigua y nueva, la columna (debe ser la misma ya que tienen que pertenecer al mismo centro de producción) y el índice de las dos peticiones a intercambiar.

```
if (!this.tablero[x1][y].getCargas().isEmpty() && peticion1>=0){
                p=this.tablero[x1][y].getCargas().get(peticion1);
}

    if (!this.tablero[x2][y].getCargas().isEmpty() && peticion2>=0){
                p2=this.tablero[x2][y].getCargas().get(peticion2);
    }
```

Ahora hacemos dos comprobaciones importantes. Por un lado, el primer "if" mira si la asignación 1 no tiene el vector de cargas vacío, es decir, si no tiene ninguna petición asignada, y el índice es mayor o igual que cero, nos guardamos esta petición en una variable temporal del tipo Petición.

Del mismo modo comprobamos que el vector cargas de la asignación 2 no este vacío y el índice de petición sea positivo, y nos guardamos la petición en una variable local.

```
cargaTotal1=this.tablero[x1][y].cargaTotal;
cargaTotal2=this.tablero[x2][y].cargaTotal;
```

Seguidamente nos guardamos las cargas totales de las dos asignaciones en variables temporales.

El intercambio lo realizaremos como dos cambios haciendo las comprobaciones necesarias.

```
if(x1!=x2 && p!=null){
    if (cargaTotal2+p.getCantidad()<=this.tablero[x2][y].camion && p2==null){
    //caso en el que cabe la caja en destino y no hace falta tocar ninguna petición
                    this.tablero[x1][y].eliminarPeticion(peticion1);
                    this.tablero[x2][y].getCargas().add(p)
                    this.tablero[x2][y].cargaTotal+=p.getCantidad();
                    b=true;
    }
```

Miramos para cambiar la petición 1 a la asignación de la petición 2.

Ahora miramos que las filas no sean las mismas, ya que estaríamos intercambiando peticiones de la misma asignación y obtendríamos un estado equivalente al actual, lo cual sería ineficiente. También comprobamos que la petición 1 no sea null. Y finalmente comprobamos que la carga del destino más la carga de la petición 1, no superen la capacidad del camión de la asignación. Si cumple estas comprobaciones, borramos la petición de la asignación actual, la añadimos a la asignación 2 e incrementamos la carga total de la asignación 2.

```
else if(p2!=null && cargaTotal2+p.getCantidad()-
        p2.getCantidad()<=this.tablero[x2][y].camion && cargaTotal1-
         p.getCantidad()+p2.getCantidad()<=this.tablero[x1][y].camion ){
    //no cabe y hay que intercambiar
            this.tablero[x1][y].eliminarPeticion(peticion1);
            this.tablero[x2][y].eliminarPeticion(peticion2);

            this.tablero[x1][y].getCargas().add(p2);
            this.tablero[x1][y].cargaTotal+=p2.getCantidad();

            this.tablero[x2][y].getCargas().add(p)
            this.tablero[x2][y].cargaTotal+=p.getCantidad();

    b = true;
}
```

Ahora miramos el caso de que no quepa la petición 1 en la asignación futura, entonces probamos de intercambiarlos. Para ello debemos comprobar que la petición 2 no sea nula y que la suma de las cargas (restando la que quitamos) no supere a la capacidad del camión en

ninguna de las dos asignaciones. En caso de cumplir estas restricciones realizamos el intercambio.

```
this.calcularHeuristicos();
return b ;
```

Por último calculamos los Heurísticos para actualizar las variables del nuevo estado y devolvemos la variable booleana que indicará a la clase sucesores si se ha realizado un cambio de estado.

4. Generación de Sucesores

La generación de sucesores de un estado, está muy relacionada con los operadores que utilizamos. Como hemos visto, según el operador que utilizamos haremos un cambio u otro sobre una asignación. La generación de sucesores consiste básicamente en, dado un estado *e*, crear todos los posibles sucesores que puede tener aplicando los operadores definidos en el apartado anterior. Todos estos sucesores los guardamos en una lista.

Para calcular el factor de ramificación no tendremos en cuenta niel tamaño de las peticiones ni si caben en los camiones para las diferentes asignaciones que podemos obtener.

- Tenemos *i* Horas
- Para cada hora *j* centros de producción
- En cada uno tenemos un camión que podemos mover entre *k* horas y *l* centros objetivos
- En cada asignación (hora-centro de producción) tenemos *m* peticiones
- Cada petición puede moverse entre *n* horas objetivo
- Cada petición puede intercambiarse con *p* peticiones objetivo de la asignación de destino

Como resultado, tenemos que el coste es de O(*i*j*k*l*m*n*p)* pero como *i=k=n* y *j=l* tenemos al final un resultado de ***O(i³*j²*m*p).***

La implementación del algoritmo que genera los sucesores es el que explicamos a continuación:

```
public List<Successor> getSuccessors(Object e) {

    LinkedList<Successor> sucesores = new inkedList<Successor>();
    Estado copia = (Estado)e;
Successor s;
```

La clase Sucesores tiene un solo método, *getSuccesors*. Primero necesita hacer una copia del estado actual para no modificarlo al buscar los posibles cambios de estado que formarán la lista de los sucesores que debemos retornar.

Para generar los sucesores debemos llamar a los dos operadores con todas las combinaciones posibles. Para ellos lo dividimos en dos partes, por un lado genereamos todos los posibles estados que se crean la intercambiar dos camiones.

```
for (int i = 0; i<11; i++) {
    for (int j =0; j<6; j++){
            for(int k=0;k<10;k++){
                    for(int l=0;l<6;l++){

                            Estado copia1 = (Estado) copia.clone();
                            if(copia1.intercambiarCamion(i, j, k, l))
                            {
                            //si podemos intercambiar camión, lo hacemos
                                    s=new Successor(new String(), copia1);
                                    sucesores.add(s);
```

```
                    }
                }
            }
```

Como podemos ver tenemos varios bucles, los dos externos, con las variables *i* y *j*, se corresponden con las horas y centros, respectivamente, del tablero del estado actual. Los siguientes dos bucles sirven para volver a recorrer todas las posiciones de la matriz para intercambiar los camiones de cada asignación con todas las demás. Como ya hemos visto en las descripciones de los operadores, en realidad se hacen restricciones a la hora de intercambiar para evitar cambios innecesarios o prohibidos, así como intercambiar dos camiones de la misma posición de la matriz, o intercambiar camiones infinitos. Después de realizar el operador, si se ha producido algún cambio en el estado, éste devolverá *true*, por tanto añadimos este sucesor al vector de sucesores.

Ahora veamos la parte de intercambiar peticiones.

```
for(int m=0; m<copia.tablero[i][j].cargas.size();m++){
    for(int n=0;n<11;n++){
        for(int a = -1; a<copia.tablero[n][j].getCargas().size(); a++)
        {
            Estado copia2 = (Estado) copia.clone();
            if(copia2.moverPeticion(i, j, n, m, a)){
            // si podemos mover una peticion, lo hacemos
                    s=new Successor(new String(), copia2);
                    sucesores.add(s);

            }
      }
}
return sucesores;
```

Dentro de los dos bucles externos que recorren la matriz de asignaciones del estado actual, volvemos a tener dos bucles. Esta vez, el bucle de fuera recorre las peticiones del vector cargas de cada asignación, y para cada una de estas peticiones, para cada hora, recorre toda la columna correspondiente al centro de producción donde debe ser entregado. Y llegados a este punto, para cada centro, prueba de intercambiarlo con otra petición de otra asignación que también pertenezca a su centro, es decir, intenta cambiarse por otra petición que pertenezca a una asignación de la misma columna del tablero.

Una vez más, recordamos que las restricciones para evitar cambios innecesarios se realizan dentro de los operadores ya explicados en el apartado *3. Operadores*.

Nuevamente, el operador devolverá *true* en caso de haber realizado algún cambio en el estado. En este caso, guardamos este sucesor en la lista de sucesores.

Y finalmente retornamos la lista de los posibles sucesores que hemos encontrando aplicando las operaciones teniendo en cuenta las condiciones de aplicabilidad que que restringen los cambios de estado descritos en su implementación.

5. Heurísticos

Los heurísticos son los métodos por los cuales la búsqueda heurística analiza la bondad de un estado determinado. En función del algoritmo de búsqueda local que se esté utilizando, sirve para determinar cuáles de los estados sucesores es el más óptimo o puede permitir llegar a estados mejores.

El agente de búsqueda de la librería AIMA considera que un estado *a* es mejor que un estado *b*, si h(*a*) < h(*b*). El enunciado de nuestro problema nos pide que consideremos los siguientes criterios de búsqueda:

- Maximizar la ganancia que obtenemos con los transportes.
- Minimizar el valor absoluto de la diferencia entre la hora límite de entrega de la petición y la hora efectiva de entrega. (Asumiendo que este valor para las peticione son entregadas es la diferencia entre la hora límite y las 8 de la mañana del día siguiente)

Por tanto hemos hecho tres heurísticos que definimos a continuación.

Heurístico 1

Este heurístico será el encargado de buscar los estados maximizando la ganancia obtenida con los transportes.
A continuación mostramos su implementación.

```
public double getHeuristicValue(Object arg0)
{

        Estado estado = (Estado)arg0;
        return -estado.getBeneficioTotal();

}
```

Como vemos es muy sencillo. Simplemente retorna el beneficio máximo del estado que nos pasan por parámetro con signo negativo. El cambio de signo es debido a que nosotros queremos maximizar, mientras que la *HeuristicFunction* de AIMA busca el heurístico mínimo de entre los posibles estados sucesores.

Heurístico 2

Este otro heurístico se encargará de minimizar el número de entregas con hora superior a la hora límite de la petición, es decir, minimizar el número de retrasos.

```
public double getHeuristicValue(Object arg0)
{

        Estado estado = (Estado)arg0;
        return estado.getRetrasos();
    }
```

De manera muy similar al anterior, el heurístico dos lo único que hace es devolver la variable retrasos del estado pasado como parámetro. Esta vez no necesitamos cambiar el signo ya que lo que buscamos es minimizar este número de retrasos obtenidos.

Heurístico 3

Por último hemos hecho otro heurístico con el cual tenemos en cuenta los dos criterios de búsqueda.

```
public double getHeuristicValue(Object arg0){

Estado estado = (Estado)arg0;
  return (-estado.P*estado.getBeneficioTotal()+
            (1- estado.P)*estado.getRetrasos()*30);

}
```

Básicamente lo que hacemos es devolver una ponderación de la ganancia y de los retrasos del estado que nos pasan por parámetro. Esta ponderación la llevamos a cabo utilizando una constante "*P*" que es preguntada en el Main de la aplicación en caso de que el usuario escoja este heurístico. De esta manera el usuario puede indicar el porcentaje de importancia que tiene maximizar la ganancia por los transportes o de minimizar el número de retrasos. También multiplicamos por 30 los retrasos para que sean valores comparables. Este heurístico es más por temas de curiosidad que por razones de seguimiento del enunciado.

6. Algoritmos

Antes de comenzar los experimentos, explicaremos un resumen de los algoritmos que utilizamos en nuestra práctica. Los algoritmos propuestos por la asignatura son: Hill Climbing y Simulated Annealing.

Hill Climbing

Es un algoritmo de búsqueda local del que se pueden destacar las siguientes características:

- **informado**: utiliza la información del estado para elegir un nodo u otro.

- **No exhaustivo**: no explora todo el espacio de estados, y por tanto la solución obtenida siempre representa un mínimo local. Esto implica que encuentra buenas soluciones pero no siempre la mejor.

- **Eficiencia**: Evita la exploración de una parte del espacio de estados considerable. Cuanto más grande es el factor de ramificación, más grande es la eficiencia respecto a otros algoritmos.

- **Funcionamiento**: para cada nodo visitado, genera sus hijos y escoge el mejor según la función heurística. Si un nodo no tiene hijos, el algoritmo acaba.

Por otro lado Hill Climbing presenta un inconveniente importante que hace falta tener en cuenta. El algoritmo puede acabar en un mínimo local de la función heurística, y no en el mínimo global que sería la mejor solución, ya que no explora todas las posibilidades. Este problema es especialmente crítico, debido a que las decisiones tomadas son definitivas, es decir, no se puede volver atrás.

Simulated Annealing

Este algoritmo presenta una dificultad adicional ya que presenta cuatro parámetros que definen su funcionamiento. Hemos realizados alguna prueba para ver cuál era la mejor configuración de estos para adaptarlo al problema para que obtenga mejores resultados.

Los parámetros son:

- **Steps**: numero de pasos para cada temperatura.
- **Degree**: valor de la temperatura inicial.
- **Lambda**: criterio de parada.
- **k**: número de sucesores que se generarán.

7. Experimentos

El objetico de realizar los experimentos consiste en ver a partir de ejemplos prácticos que conjuntos de operadores generan una mejor solución, qué estrategia de generación de la solución inicial de las que tenemos elabora una primera solución suficientemente buena, o bien, qué heurístico de los definidos es el más óptimo. También podemos comparar si el algoritmo Hill Climbing supone una mejor solución respecto al Simmulated Anealing.

La metodología que hemos seguido consiste en, para cada experimento, definir las siguientes partes:

- Lo que se busca en el experimento
- El planteamiento del experimento
- Los resultados esperados
- Los análisis de los datos obtenidos
- Las conclusiones que hemos deducido

Junto con todos los archivos de entrega de la práctica, adjuntamos un Excel con todos los experimentos y las gráficas resultantes de los valores extraídos de las ejecuciones. En este Excel figura cada experimento en una diferente pestaña (Hoja de cálculo). Existen algunos experimentos que hemos realizado por curiosidad que no figuran en la documentación de la práctica ya que no los pedían en el enunciado, aunque sí que hemos incluido alguno que no figurada en el guión de la práctica porque ha producido resultados que nos parecían interesantes respecto a lo esperado, por ejemplo en el experimento 8, donde hemos contrastados los resultados también para el doblar la probabilidad de asignar las peticiones a las *últimas* cuatro horas.

7.1 Primer experimento: conjunto de operadores.

Este experimento se centra en determinar cuál de los conjuntos de operadores da mejores resultados para la primera función heurística, que optimice el primer criterio de calidad (maximizar los beneficios por carga), con un escenario de 250 peticiones y una distribución de las capacidades de los camiones, pesos de las peticiones y horarios de entrega, equiprobables. El algoritmo que debemos utilizar es Hill Climbing.

7.1.1 Planteamiento del experimento

Primero hemos generado 10 escenarios, con los 60 camiones, escogiendo sus capacidades aleatoriamente y 250 peticiones. Como hay dos maneras de generar el estado inicial y no se especifica cual debemos utilizar, hemos probado con las dos. Es decir, para cada escenario se ejecutará el programa para cada conjunto de operadores y para cada manera de generar el estado inicial. En total se realizaran 40 ejecuciones (10 escenarios*2 conjuntos de operadores*2 formas de generar el estado inicial).

En el análisis se compararán los datos obtenidos en los diferentes escenarios entre conjuntos de operadores diferentes utilizando el mismo generador inicial.

Debemos añadir que en cuanto a los operadores que tenemos, no tiene sentido utilizar por separado el de intercambiar camiones, por tanto los dos operadores que consideraremos para los experimentos será: mover peticiones, y el conjunto de operadores que forman mover

peticiones junto con cambiar camiones. Para referirnos a ellos los denominaremos, a partir de ahora, operador **simple** y operador **compuesto**, respectivamente.

7.1.2Resultados esperados

Los resultados esperados son los siguientes:

- Se espera que los resultados obtenidos con el operador compuesto sean mejor o iguales que los obtenidos por el operador simple. El motivo es que el operador compuesto incluye el operador simple y a demás otro operador.

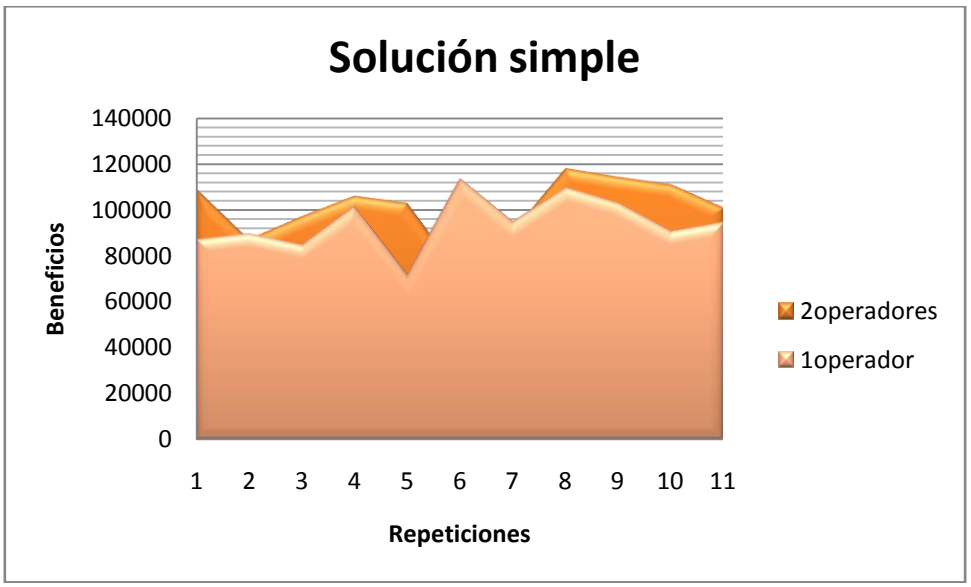
7.1.3 Análisis de los datos obtenidos

A continuación se encuentran los análisis más relevantes de los datos obtenidos.

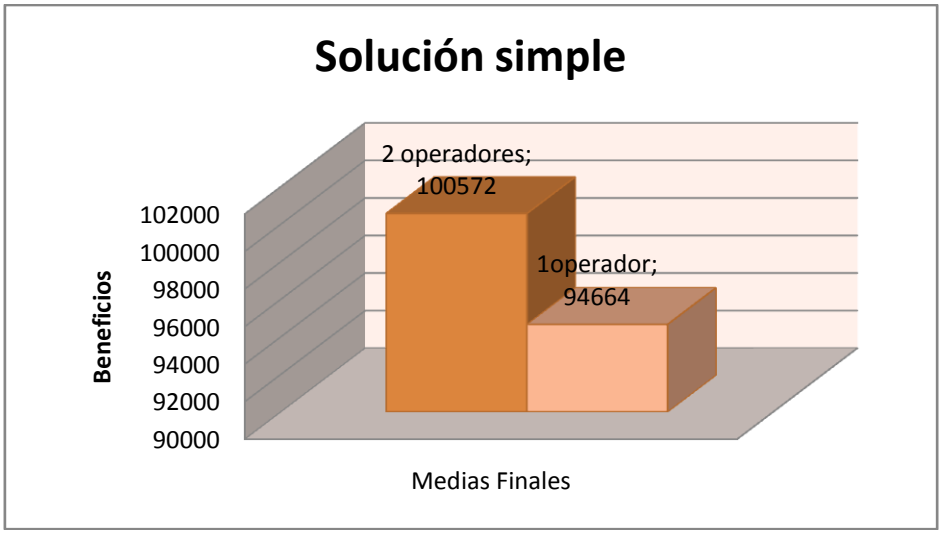
Análisis de los beneficios obtenidos (heurístico 1)

Analizamos los beneficios obtenidos para los diferentes operadores y para las diferentes estrategias de solución inicial.

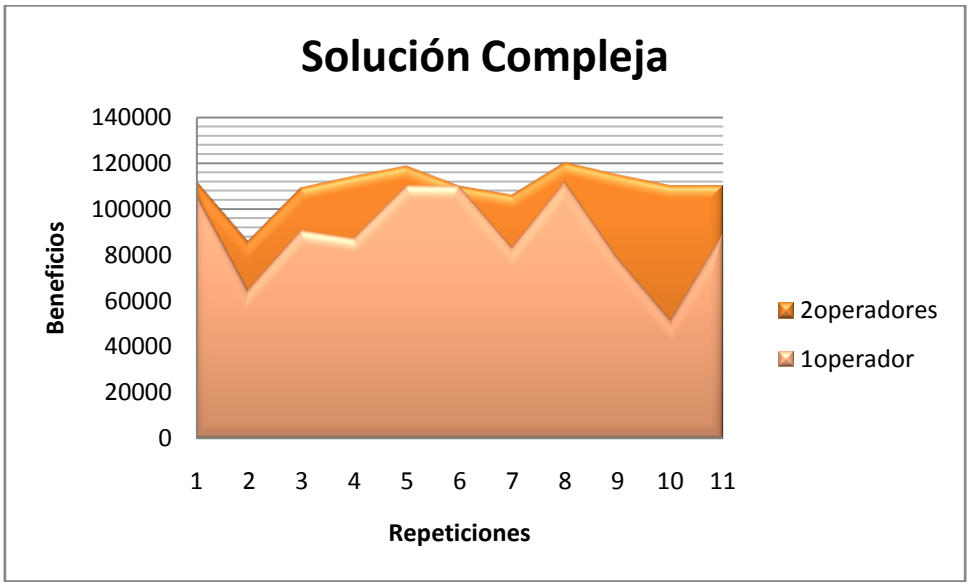
Para la ***estrategia simple*** hemos obtenido la siguiente gráfica:



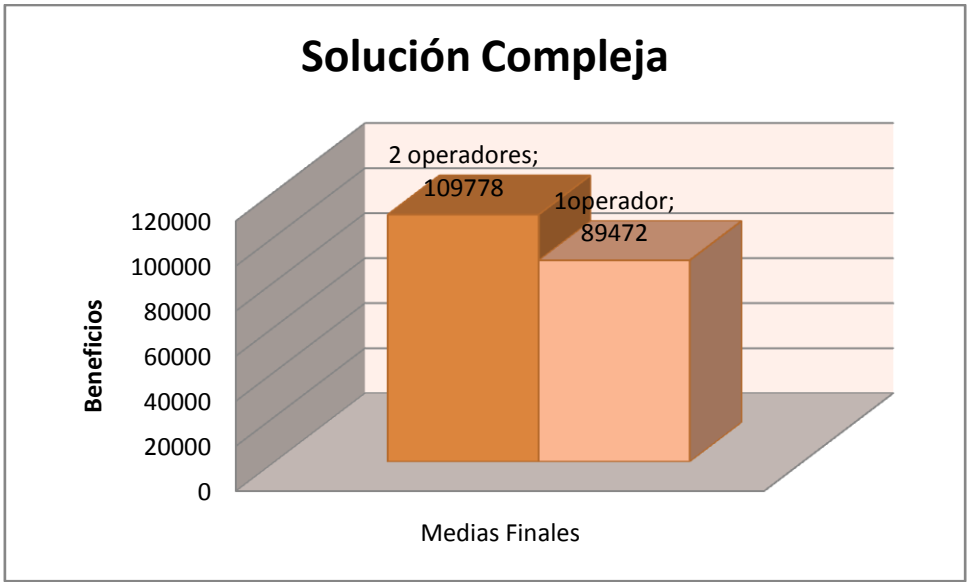
Como podemos ver cuando usamos el operador compuesto (2 operadores) tenemos un beneficio superior, suponiendo el heurístico donde buscamos maximizar el beneficio por cargas, en la mayoría de los escenarios. La última repetición que hemos añadido a la gráfica (repetición 11) se corresponde con la media de los 10 escenarios. En la siguiente gráfica podemos ver con más claridad el resultado de estas medias.



Para la ***estrategia compleja*** hemos obtenido la siguiente gráfica:



Nuevamente cuando utilizamos el operador compuesto respecto del operador simple tenemos mayor beneficio en la mayoría de los estados, esta vez más claramente que con la generación de estado inicial simple. La última repetición se corresponde con la media, lo veremos más claro en la siguiente gráfica.



7.1.4 Conclusiones

Este experimento intentaba ver cuál de los operadores era mejor a la hora de buscar el mayor beneficio respecto al heurístico 1. En el apartado anterior hemos visto que, tanto cuando generamos el estado inicial con estrategia simple como con la estrategia compleja, el operador compuesto obtiene como media un beneficio claramente mayor respecto al operador simple. Es por esto que el operador compuesto será escogido para los siguientes experimentos.

7.2 Segundo experimento: Estrategia de inicio

En este experimento debemos determinar qué estrategia de generación de solución inicial da mejores resultados usando la función heurística de maximizar los beneficios por cargas, con el escenario del apartado anterior y usando el algoritmo de Hill Climbing. A partir de las conclusiones obtenidas deberemos fijar la mejor estrategia de generar la solución inicial para los siguientes experimentos.

7.2.1 Planteamiento del experimento

Con los mismos 10 escenarios que el apartado anterior, y ya habiendo escogido el operador compuesto que utilizaremos a partir de ahora, el objetivo de este experimento será ver cuál de las dos estrategias de generación del estado inicial es mejor.

7.2.2Resultados esperados

Los resultados que esperamos en este experimento son los siguientes:

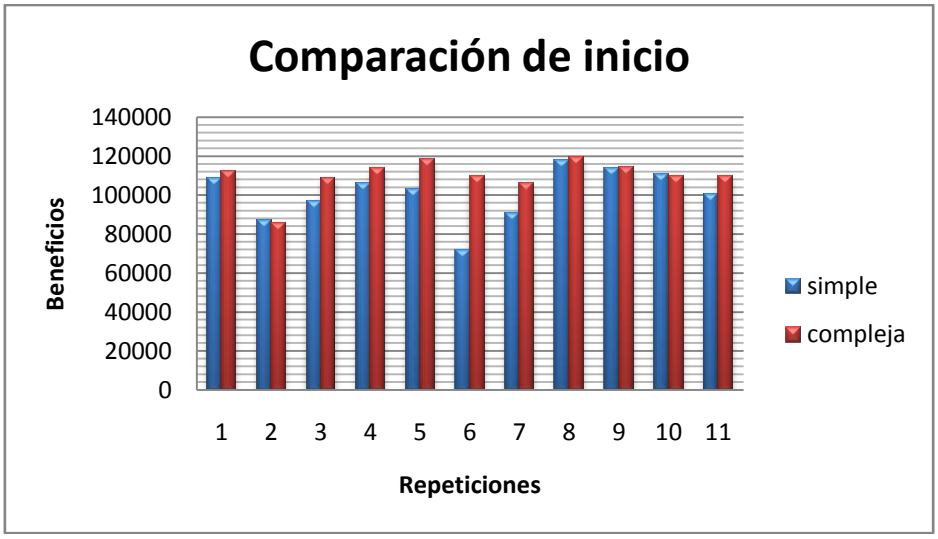
- Se espera que la estrategia de generación del estado inicial compleja sea más eficiente y que obtenga más beneficios por carga que la estrategia simple. Esto debería ser así ya que, en la estrategia simple, no nos basamos en ningún método para hacer más eficiente el estado inicial, simplemente asignamos a los camiones infinitos las peticiones y la aplicación tendrá que realizar muchos cambios hasta alcanzar un beneficio mejor. No obstante, la estrategia compleja, intenta llenar al máximo los camiones de las primeras horas para que haya el mínimo número de retrasos posibles.

7.2.3 Análisis de los datos obtenidos

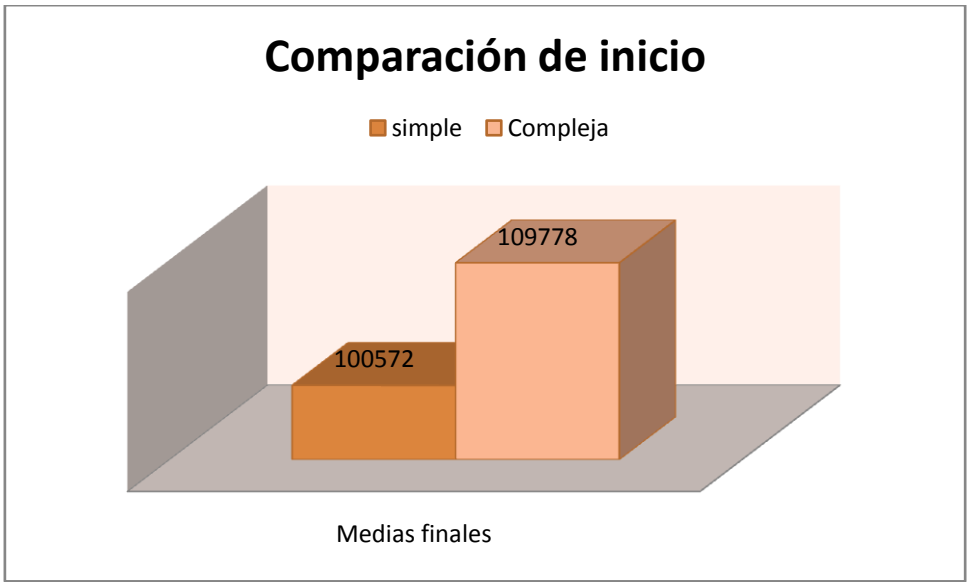
A continuación se encuentran los análisis más relevantes de los datos obtenidos.

Análisis de los beneficios obtenidos (heurístico 1)

Como podemos ver en la siguiente gráfica, en todos los estados hemos obtenido un beneficio mayor cuando utilizamos la estrategia compleja.



La última repetición se corresponde con la media de los 10 escenarios, podemos ver más claramente en la siguiente gráfica que esta media predice que la estrategia de generar el estado inicial compleja es mejor.



7.2.4 Conclusiones

En conclusión podemos determinar, después de ver los gráficos del apartado anterior, que la estrategia compleja, es mejor que la simple cuando buscamos maximizar el beneficio obtenido por las cargas de los camiones. A partir de ahora, utilizaremos siempre la estrategia compleja para los experimentos siguientes.

7.3 Tercer experimento: Determinación de parámetros Simulated Annealing.

En este experimento debemos determinar qué paramos para el Simulated Annealing dan mejor resultados para los mismos escenarios, usando la misma función heurística y los operadores y estrategia de generación de estado inicial escogidos en los experimentos anteriores.

7.3.1 Planteamiento del experimento

Dado los 10 escenarios definidos en los experimentos anteriores, probaremos para cada parámetro del Simulated Annealing, diferentes valores, los estudiaremos y nos quedaremos con el que más nos convenga para maximizar el beneficio teniendo en cuenta el tiempo de ejecución.

7.3.2Resultados esperados

Se espera encontrar unos buenos parámetros para el Simulated Annealing que permitan encontrar soluciones de calidad.

7.3.3 Análisis de los datos obtenidos

A continuación mostramos como hemos seleccionado los diferentes parámetros

Step:

Para realizar el estudio del *step* hemos fijado los siguientes valores:

- **Degree** = 50
- **K** = 50
- **Lambda** = 0.01

En la tabla vemos los diferentes tiempos de ejecución que hemos obtenido con diferentes *steps* que hemos probado.

Step	**10**	**100**	**150**	**200**	**250**	**400**	**500**	**600**	**1000**	**2000**
Tiempo ejecución	12	72	119	149	207	307	486	603	1144	2103

Podemos apreciar que a medida que crecer el valor *step* también aumenta el tiempo de ejecución. Además cuanto más grande sea también hemos notado que el beneficio aumenta.

El valor que hemos decido escoger para el parámetro *step* es 400, para tener un buen beneficio sin un tiempo de ejecución demasiado alto.

Degree

Manteniendo el valor del *step*=400, y los parámetros para *k* y *lambda* mencionados en el apartado anterior, hemos realizado los estudios convenientes para el parámetro *degree*.

degree	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**
tiempo	306	316	321	317	274	301	301	304	304	283
Beneficio	66370	93270	79780	72170	41330	93560	93040	70190	89930	39000

Como vemos en la tabla, los valores del tiempo de ejecución no varían demasiado con diferentes valores para el parámetro *degree*, por tanto cogemos el valor 50 ya que es el que mayor beneficio nos ha dado.

k

Para determinar el valor de este parámetro hemos fijado el step y degree que hemos escogido en los anteriores apartados, y la misma lambda que hemos estimado en el primer apartado.

valor K	**5**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**200**
Tiempo ejecución	306	303	311	289	274	292	301	273	289	308
Beneficio	77960	58170	86410	104990	62480	96480	77330	43360	47200	73860

En la tabla que hemos obtenido vemos que los tiempos de ejecución también son muy parecidos así que hemos decidido su valor por el máximo beneficio obtenido.

Lambda

Por último, para determinar el parámetro lambda, hemos fijado los otros parámetros ya escogidos y hemos obtenido los siguientes resultados.

valor lambda	**0,005**	**0,01**	**0,05**	**0,1**	**0,2**	**0,5**	**0,7**	**0,8**	**0,9**	**1**
Tiempo ejecución	311	292	319	313	308	302	310	312	300	293
Beneficio	59620	90280	78980	73400	62080	56870	82830	76290	74030	86410

Para *lambda*, hemos escogido el valor 0.01 ya que minimiza el tiempo de ejecución y además produce buenos resultados en cuanto a los beneficios.

7.3.4 Conclusiones

Los mejores valores que hemos determinado para el algoritmo Simulated Annealing, son los siguientes:

- **Step:** 400
- **Degree:** 50
- **K:** 30
- **Lambda:** 0,01

Estos valores serán los que emplearemos a continuación en los demás experimentos. Los tiempos de ejecución no han variado a penas para los valores de grado, k, y lambda pero sí para step.

7.4 Cuarto experimento: Tiempo de ejecución respecto al número de peticiones.

En el cuarto experimento debemos estudiar cómo evoluciona el tiempo de ejecución para hallar la solución en función del número de peticiones que tenemos. Para llevarlo a cabo, debemos, para los escenarios de los apartados anteriores, ejecutar el programa empezando con 200 peticiones, e ir aumentándolas en número de 50 en 50, hasta ver la tendencia. Debemos usar el algoritmo de Hill Climbing y la heurística que busca maximizar el beneficio obtenido con los transportes.

7.4.1 Planteamiento del experimento

El planteamiento del experimento es el siguiente:

- Se generan 10 escenarios con 200 peticiones. Después ejecutaremos el programa con estos escenarios y se recogerán los datos obtenidos.

- Una vez analizada la etapa de recogida de datos, se modificarán los escenarios anteriores sumando 50 al número de peticiones. Se ejecuta el programa con estos nuevos escenarios y se recogen los datos.

- A continuación analizamos si podemos determinar una tendencia con los datos ya almacenados. En caso afirmativo, se vuelve al punto 2. En caso contrario, se acaba la parte de experimentación y se determinan las conclusiones.

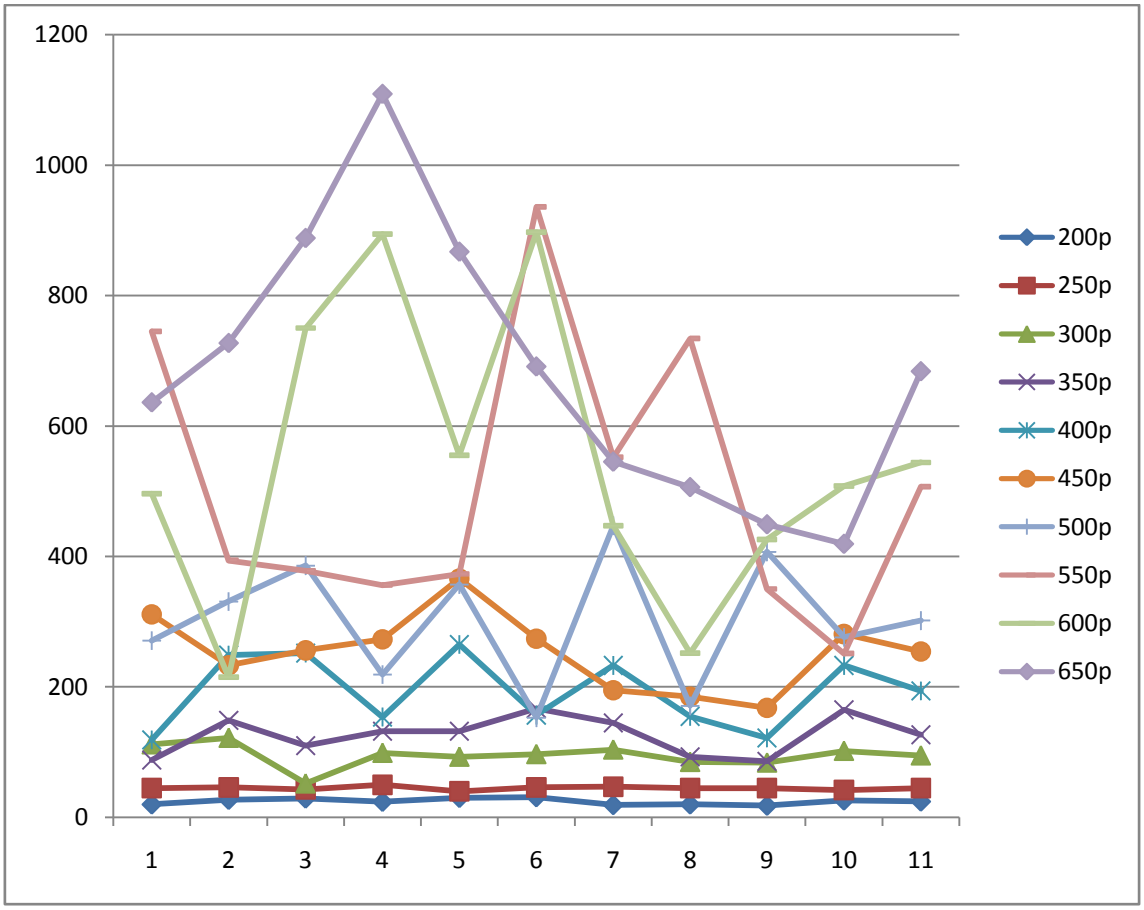
7.4.2Resultados esperados

Los resultados esperados son los siguientes:

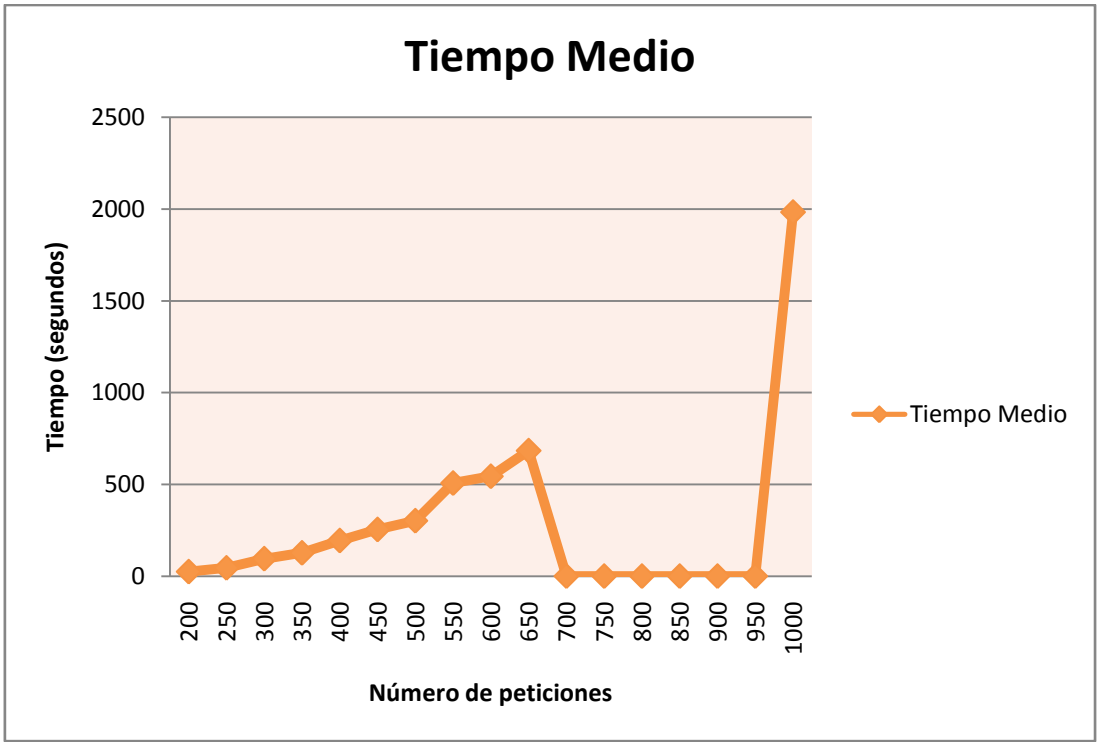
- Se espera que, a medida que se vaya incrementando el número de peticiones, el tiempo de ejecución sea cada vez mayor. Esto es debido a que cuantas más peticiones tengamos para repartir entre los distintos centros de producción, más grande es el espacio de soluciones por el que nos movemos.

7.4.3 Análisis de los datos obtenidos

En la siguiente gráfica, podemos ver las diferentes ejecuciones para los diferentes escenarios y para los diferentes números de peticiones, empezando desde 200 hasta 650.



Para más claridad de la tendencia miramos la siguiente gráfica con los tiempos medios de los diferentes escenarios. Esta tendencia es claramente creciente. Hemos realizado también la ejecución de los 10 escenarios para 1000 peticiones para comprobar que seguía creciendo a medida que seguíamos aumentando las peticiones. Debemos añadir, que no hemos realizado las ejecuciones para las peticiones entre 700 y 650, por tanto esta parte de la gráfica debemos abstraerla de los resultados que obtendríamos.



7.4.4 Conclusiones

Tal y como tenemos planteado nuestro problema, añadir más peticiones repercute directamente sobre nuestro tiempo de ejecución total. Esto tiene sentido, ya que cuantas más peticiones debemos asignar, más posibles estados sucesores tenemos en el rango de soluciones, y más veces deberemos aplicar los operadores; en definitiva, realizamos una combinación mucho mayor por el valor de entrada de número de peticiones, lo que afecta el tiempo de ejecución del programa.

7.5 Quinto experimento: Diferencias entre heurísticos.

Para este experimento debemos estimar la diferencia entre la ganancia obtenida y el tiempo de ejecución para hallar la solución, usando dos heurísticas, la usada en los experimentos anteriores y otro de los que hemos definido que optimice el segundo criterio de calidad propuesto en el enunciado de la práctica, es decir, que minimice los retrasos.

Para llevar a cabo este experimento debemos ejecutar el escenario del primer experimento con 200, 250 y 300 peticiones. El algoritmo que debemos utilizar es Hill Climbing.

7.5.1 Planteamiento del experimento

Para realizar el experimento realizamos el siguiente planteamiento:

- Utilizaremos los escenarios del experimento 1 con 200, 250 y 300 peticiones
- Obtendremos los datos de las ejecuciones para todos los escenarios utilizando el primer heurístico.
- Obtendremos los datos de las ejecuciones para todos los escenarios utilizando el heurístico 2.
- Se analizarán los datos obtenidos.
- Se determinarán las conclusiones.

7.5.2Resultados esperados

Los resultados esperados son los siguientes.

- En media, se espera que el beneficio total obtenido con el primer heurístico en los diferentes escenarios, sea mayor o igual que los obtenidos con el heurístico dos. Ya que el segundo heurístico no optimiza directamente la ganancia.

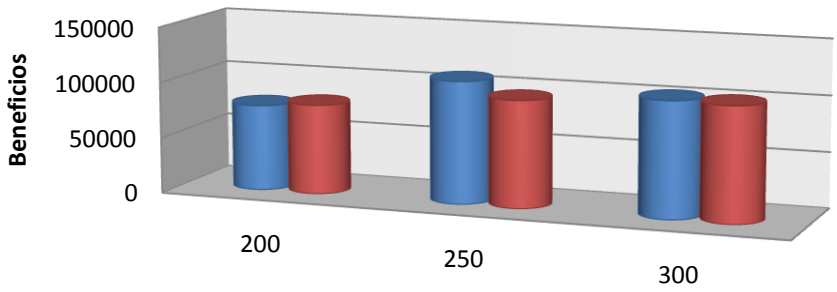
- En cuanto al tiempo de ejecución, se espera que el primer heurístico tarde menos que el segundo, ya que desde nuestro estado inicial parece que sea más rápido encontrar una solución que maximice los beneficios con un número menos de cambios de estado.

7.5.3 Análisis de los datos obtenidos

A continuación analizamos los datos más relevantes obtenidos.

En la siguiente tabla vemos en medios los beneficios obtenidos para cada escenario con los diferentes números de peticiones que debíamos realizar para el experimento, con los dos heurísticos.

Comparación de beneficios

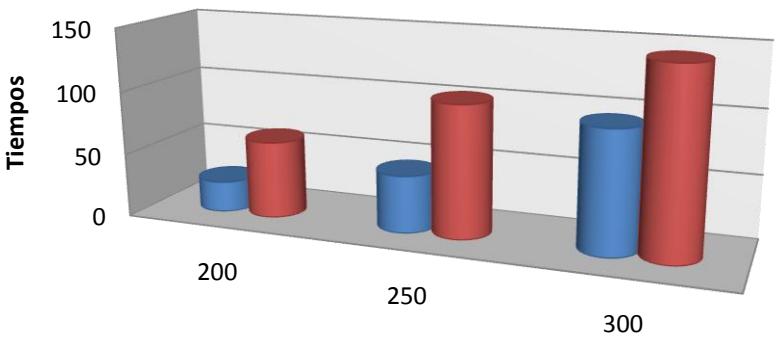


	200	250	300
Beneficios h1	77106	109778	103838
Beneficios h2	80633	96079	102635

Podemos observar que ambos heurísticos están muy igualados en cuanto a los beneficios obtenidos, pero que el primer heurístico en media es un poco mejor. Nos ha sorprendido que la diferencia entre estos dos heurísticos no haya sido mayor pues el primero afecta más directamente a las ganancias por transporte. Sin embargo tiene sentido, ya que indirectamente, al reducir el número de retrasos, estamos consiguiendo reducir la mayoría de las pérdidas de beneficio total.

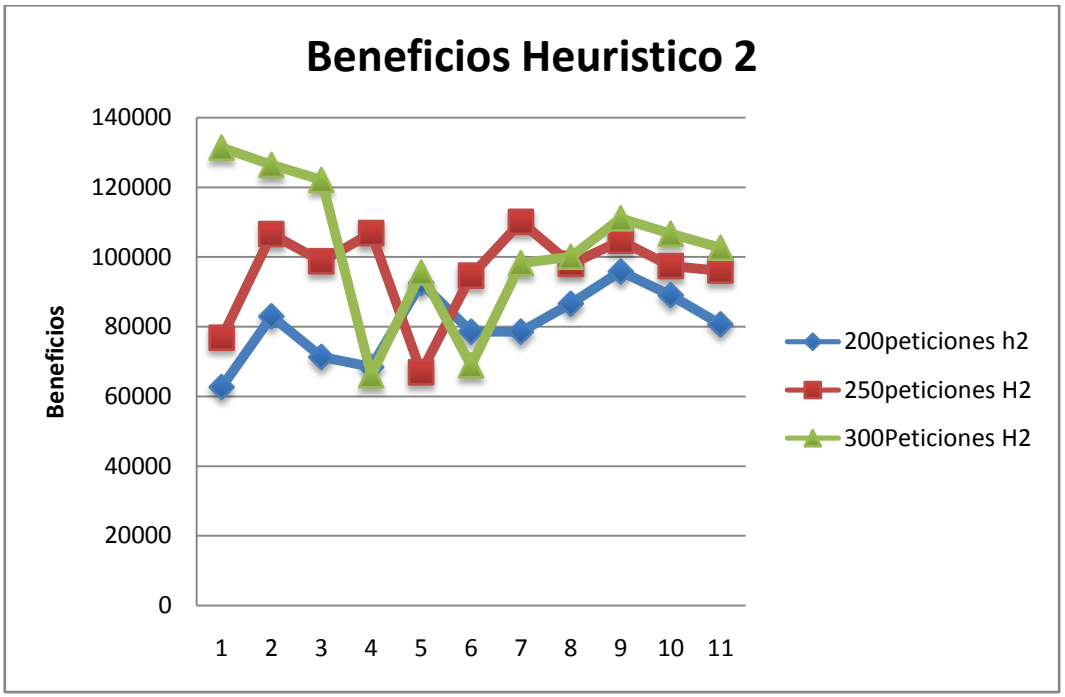
Comparación de tiempos

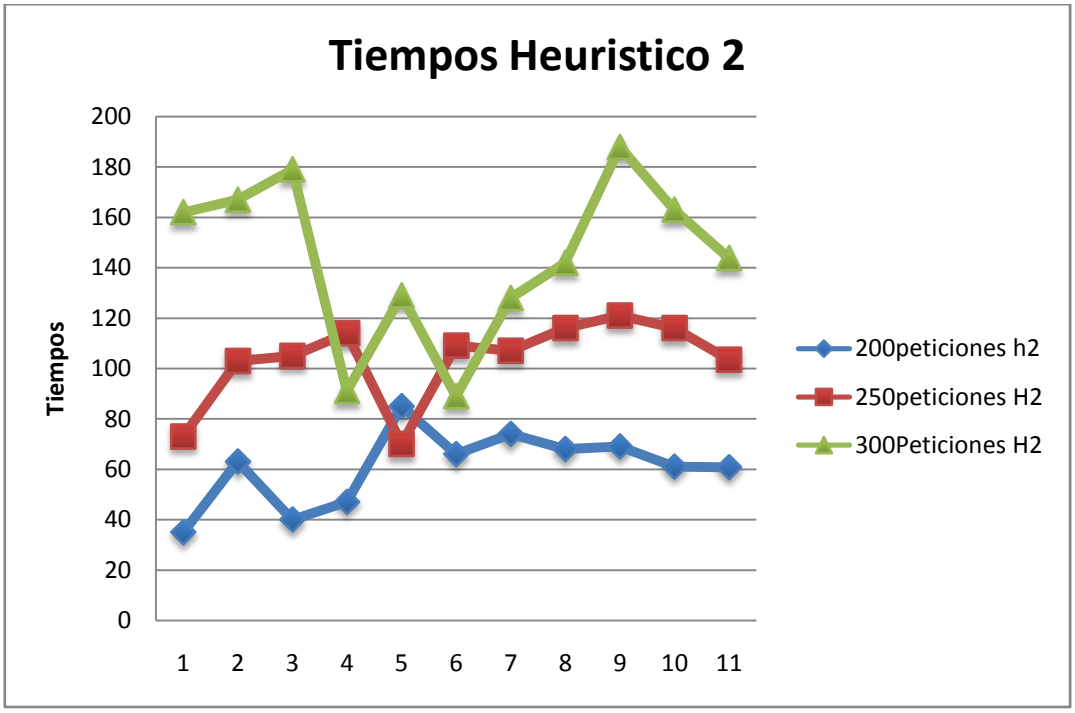
En esta otra gráfica vemos que los tiempos de ejecución tienen una diferencia mucho más relevante. El heurístico dos, en media, tarda mucho más en completar las ejecuciones.



	200	250	300
Tiempos h1	24,4	44,9	95
Tiempos h2	60,8	103,4	143,8

A continuación vemos las gráficas de beneficios y tiempos del heurístico 2 para los escenarios del experimento:

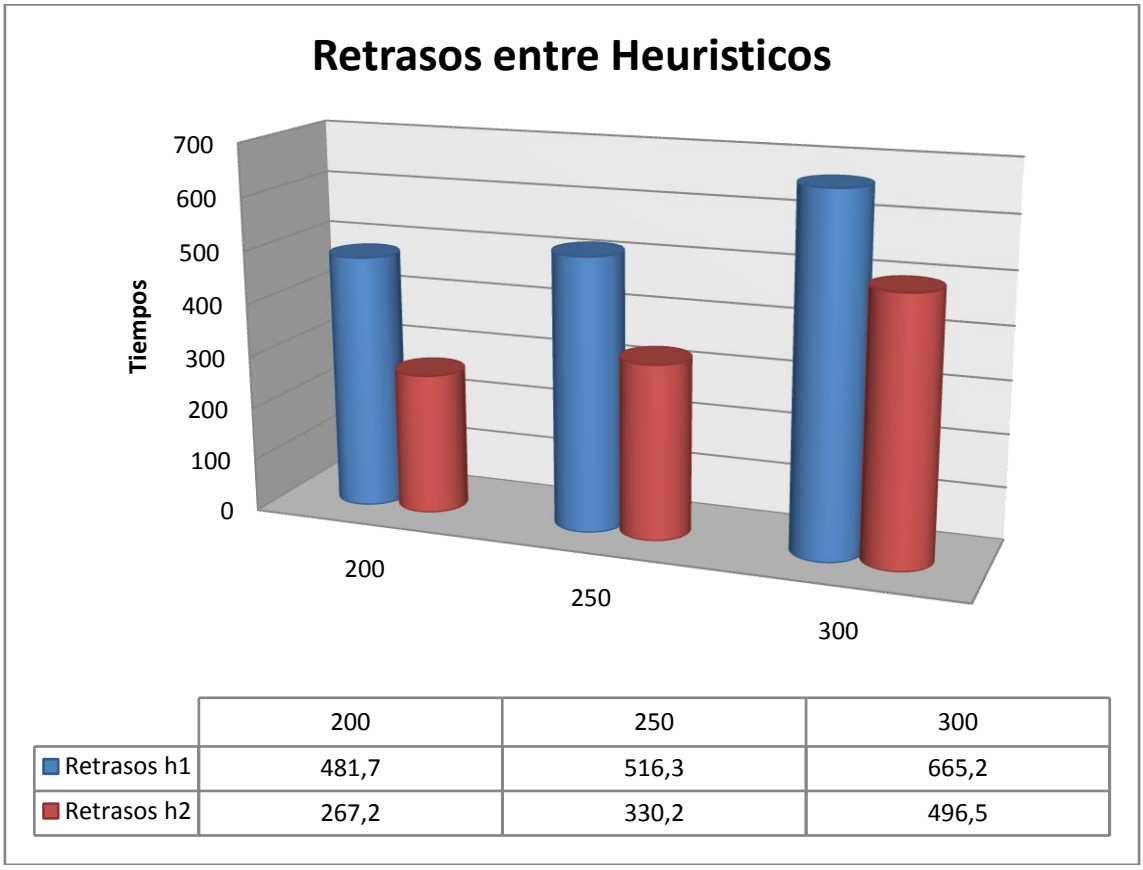




Vemos que son acordes con los resultados esperados que a mayor número de peticiones, mayor tiempo de ejecución y mayor beneficio.

Comparación de retrasos

Como podemos ver en la siguiente gráfica, el heurístico 2 obtiene mejores resultados en cuanto al objetivo de minimizar los retrasos. Tiene sentido, ya que este heurístico tiene como objetivo reducir este número de peticiones entregadas después de su hora límite, mientras que el primer heurístico, solo tiene en cuenta el beneficio y no considera los retrasos.

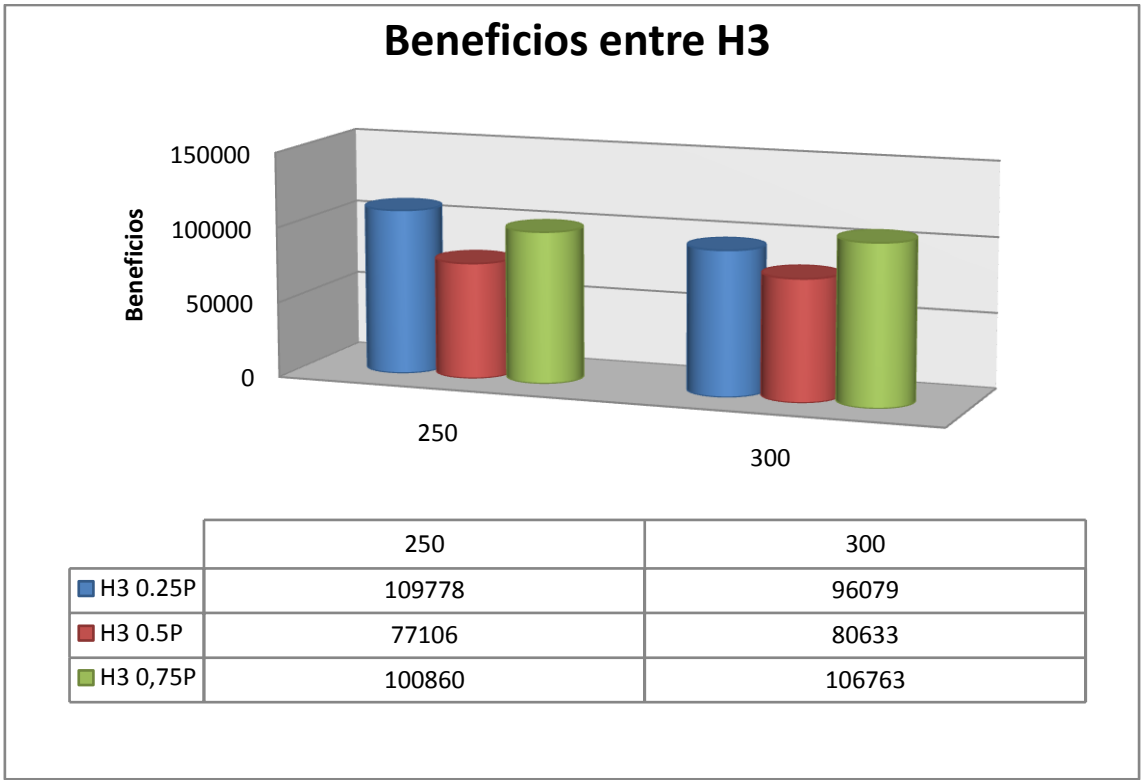


	200	250	300
Retrasos h1	481,7	516,3	665,2
Retrasos h2	267,2	330,2	496,5

Experimento adicional: Heurístico 3

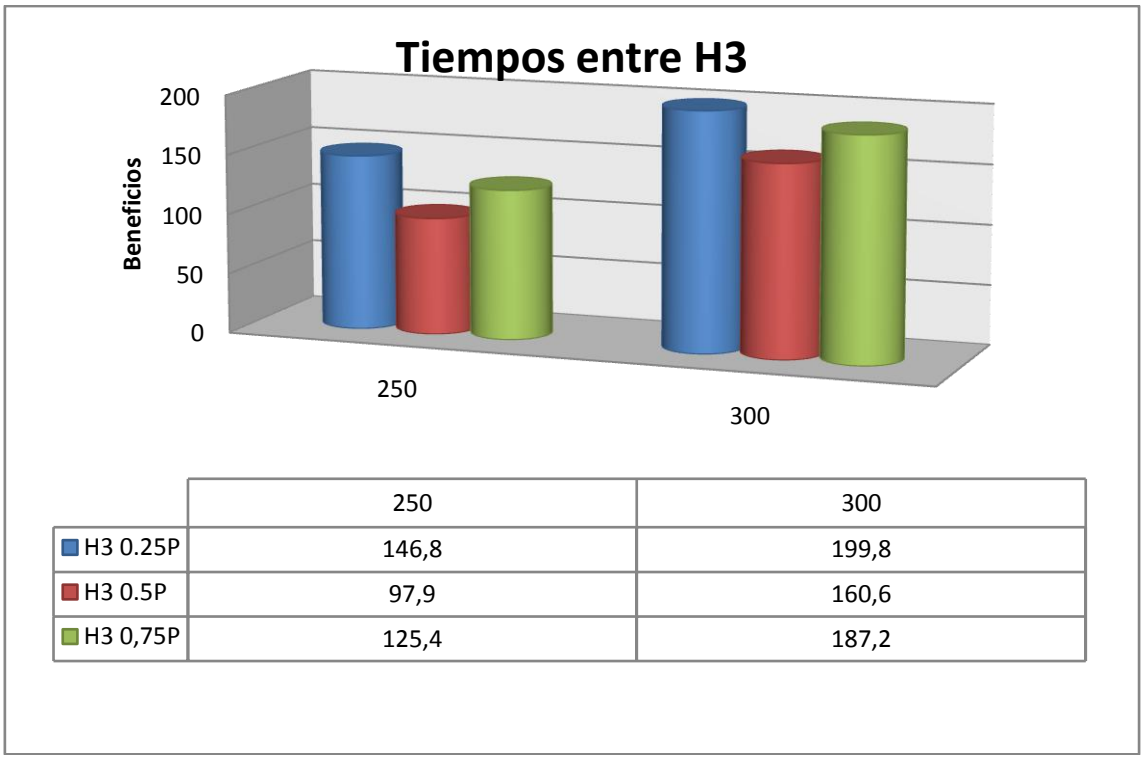
Como curiosidad nuestra hemos realizado también experimentos con el tercer heurístico que hemos implementado, para comprobar si realmente se obtenía en media mejores resultados que con los otros dos heurísticos que pedía el enunciado de la práctica. Recordamos que un valor de P pequeño ponderaría a favor del heurístico 2 y un valor de P grande ponderaría a favor del heurístico 1.

La siguiente gráfica muestra los resultados obtenidos para diferentes constantes de *P* que miden la ponderación a la hora de usar el heurístico, teniendo en cuenta el beneficio medio obtenido con los transportes. Como podemos ver, no por introducir una constante P muy alta obtendremos más beneficios, esto es debido a que indirectamente, como hemos visto en otros experimentos, el segundo heurístico también afecta en gran medida al beneficio total obtenido.



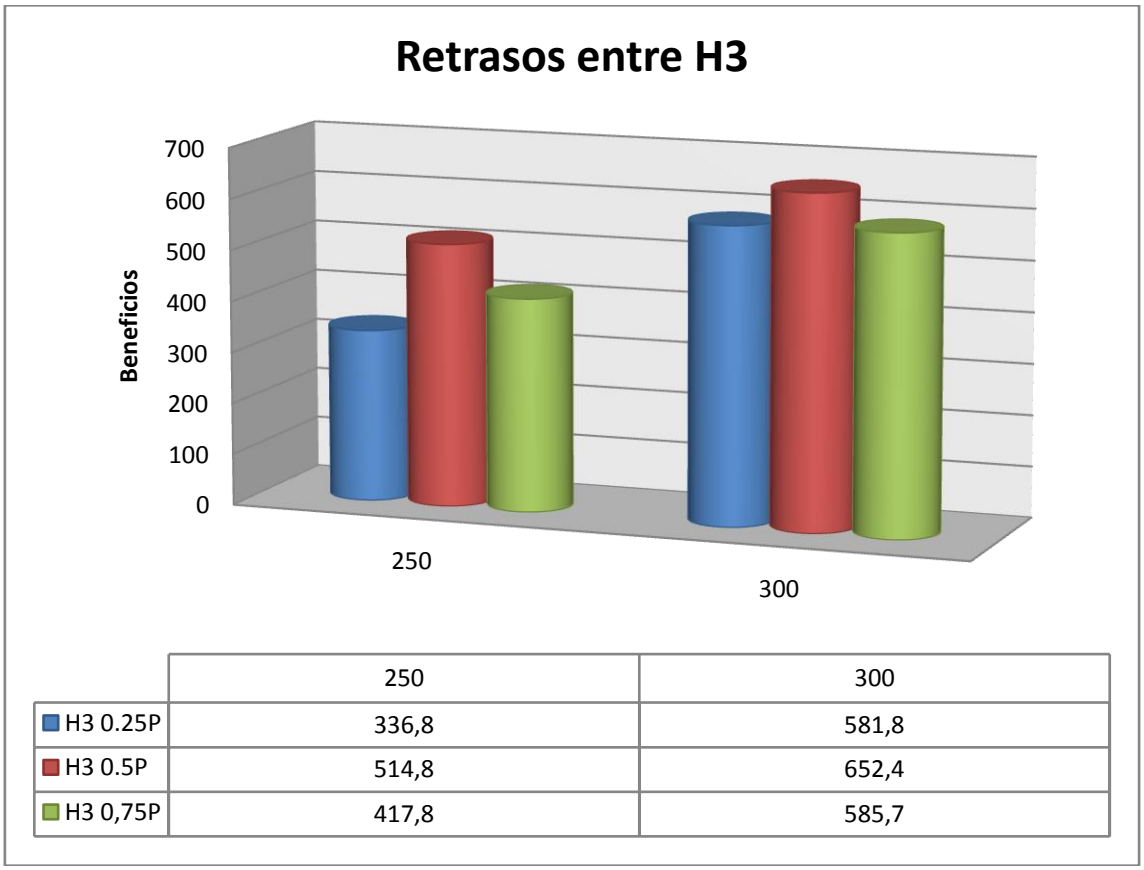
	250	300
H3 0.25P	109778	96079
H3 0.5P	77106	80633
H3 0,75P	100860	106763

En cuanto a los tiempos vemos que con una constante *P* muy pequeña más tarda en ejecutarse el programa en la media total de las ejecuciones realizadas. Sin embargo, con una constante muy alta, no obtenemos mejor resultados en cuanto al tiempo de ejecución. Los mejores resultados es con una constante intermedia de 0,5 donde tiene en cuenta un 50% de ambos criterios de calidad.



	250	300
H3 0.25P	146,8	199,8
H3 0.5P	97,9	160,6
H3 0,75P	125,4	187,2

Por último, para el segundo criterio de calidad, vemos que el tercer heurístico, cuanto mejores resultados tiene es cuando la constante P que introducimos es baja. Esto tiene sentido, ya que la constante mide el porcentaje de importancia para dar al primer criterio de calidad.



	250	300
H3 0.25P	336,8	581,8
H3 0.5P	514,8	652,4
H3 0,75P	417,8	585,7

7.5.4 Conclusiones

En conclusión tenemos que el heurístico 2 en media tarda más que el 1 para los diferentes estados en los que variamos el número de peticiones. Mientras que los beneficios obtenidos no se diferencian tanto comparando con los del heurístico 1.
Por otro lado, el heurístico 2 obtiene mejores resultados a la hora de intentar minimizar el número de entregas fuera de la hora límite de los pedidos, ya que este afecta directamente a este hecho.
Nos ha sorprendido en parte que el segundo heurístico obtenga unos resultados mucho mejores que el primero en sentido global, pues reduce retrasos y ofrece buenos beneficios. Suponemos que el hecho de su largo tiempo de ejecución no favorece su uso, y a niveles altos de número de peticiones (+1000) ambos obtendrán resultados parecidos en ambos aspectos pues los retrasos y las peticiones irán mucho mas ligadas al tener más peticiones.

7.6 Sexto experimento: Hill Climbing vs Simulated Annealing.

En este experimento, dados nuevamente los mismos escenarios que en el apartado anterior, debemos estimar la diferencia entre la ganancia obtenida con Hill Climbing y la obtenida con Simulated Annealing para las dos heurísticas usadas en el apartado anterior y el tiempo de ejecución para hallar la solución de ambos algoritmos.

7.6.1 Planteamiento del experimento

El planteamiento del experimento es el siguiente:

- Utilizaremos los escenarios creados en el experimento 5.
- Utilizaremos los datos obtenidos en el apartado anterior como datos del HillClimbing para ambos heurísticos.
- Ejecutaremos el programa utilizando el algoritmo Simulated Annealing en los escenarios del experimento anterior.
- Compararemos los datos obtenidos con el método Simulated Annealing, con los de Hill Climbing del apartado anterior.
- Determinaremos las conclusiones.

7.6.2Resultados esperados

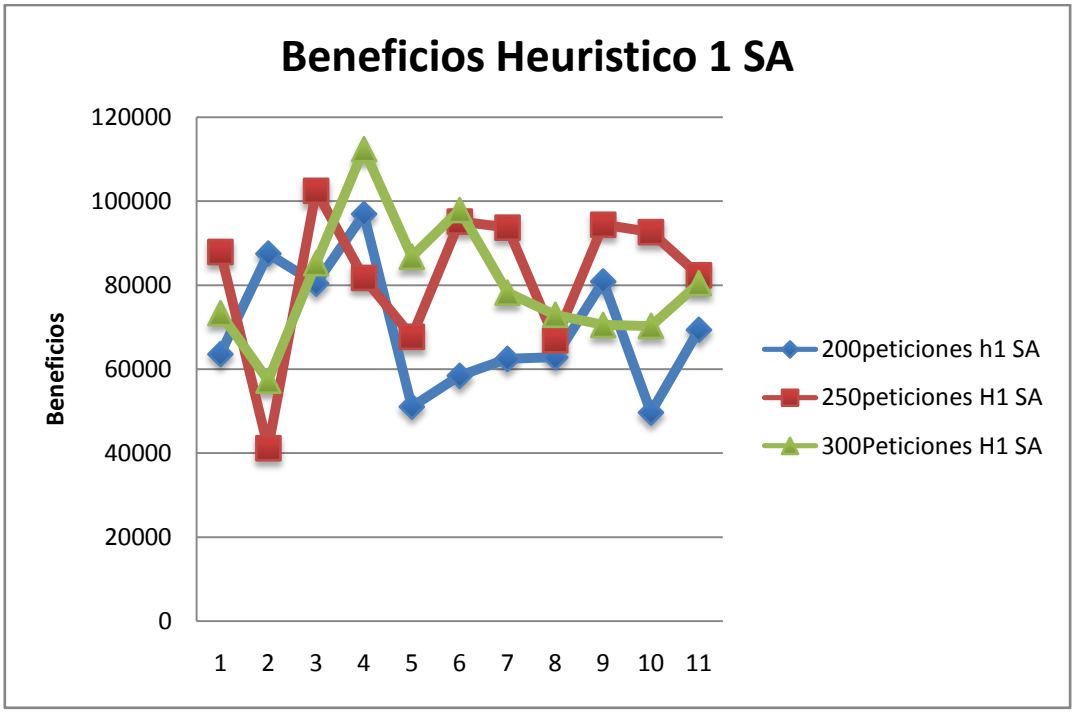
Se espera que tanto los beneficios totales, como el tiempo de ejecución, sea como media, mejor con el algoritmo de Hill Climbing que con el Simulated Annealing.

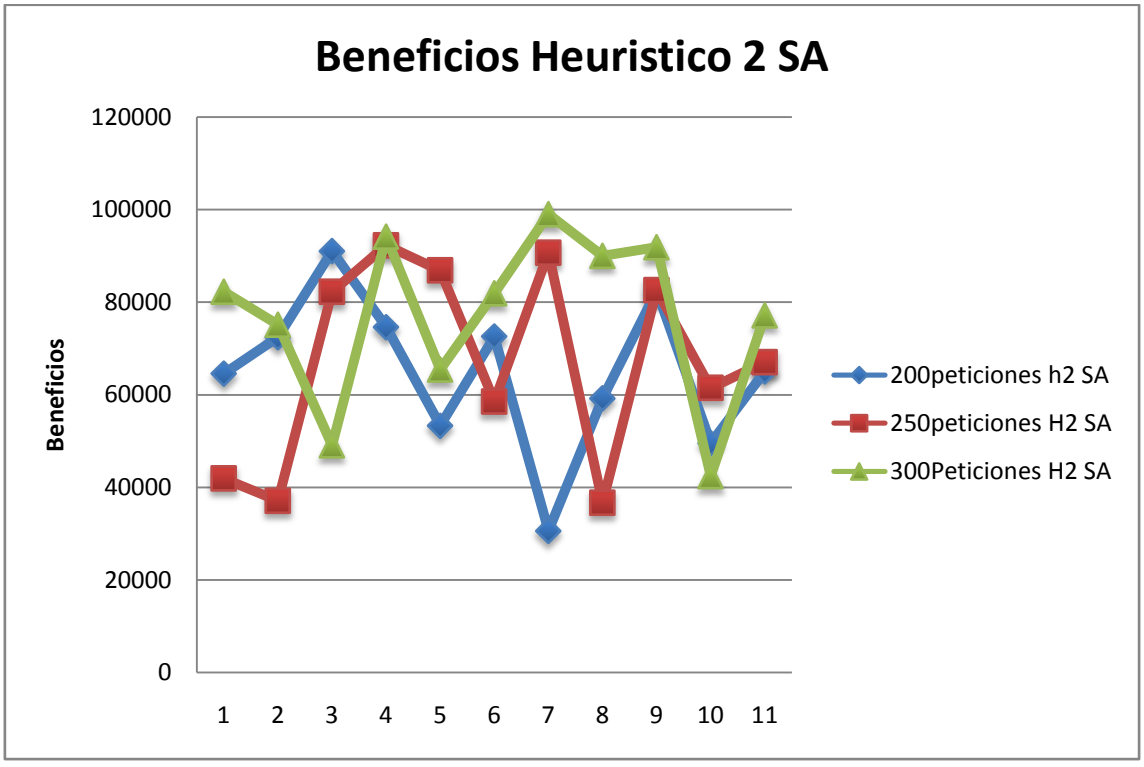
7.6.3 Análisis de los datos obtenidos

A continuación mostramos las gráficas que hemos obtenido con los datos de las ejecuciones de los diferentes escenarios con cada uno de los heurísticos, para los dos algoritmos.

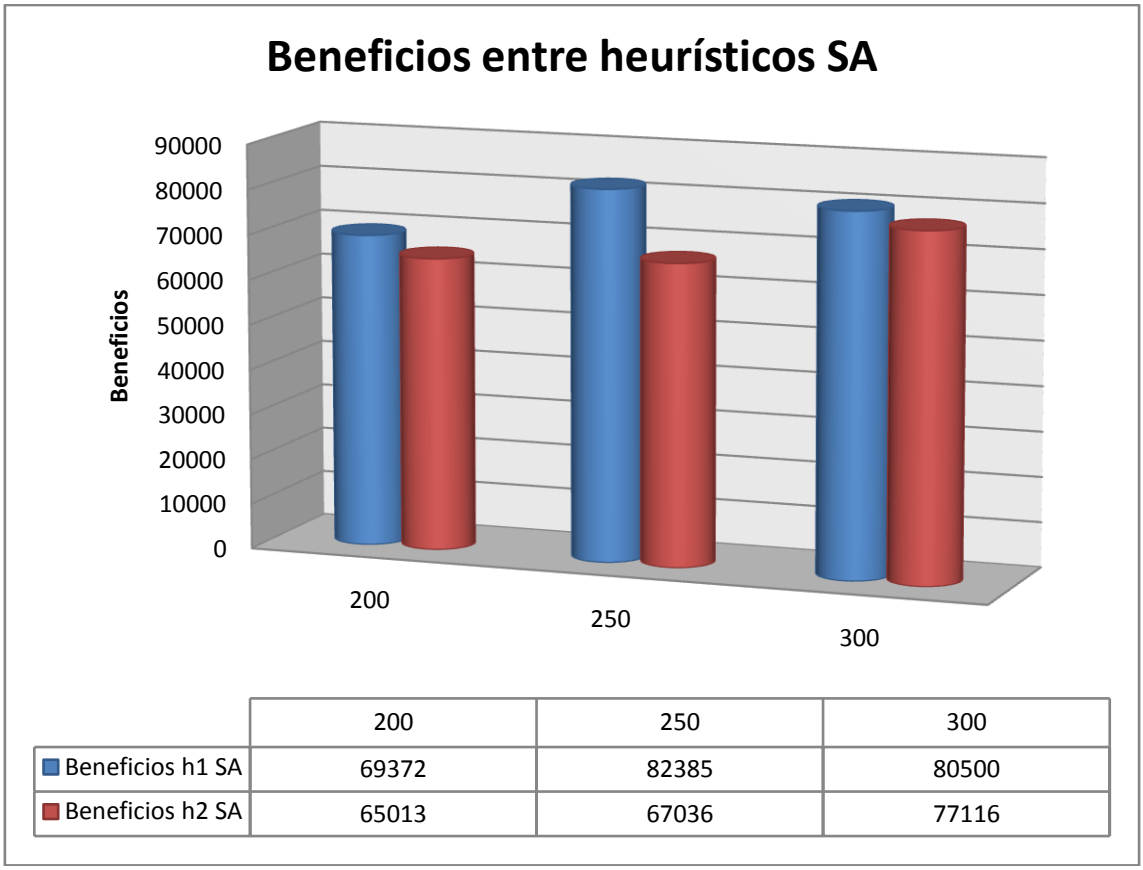
Comparación de beneficios con Simulated Annealing

En las siguientes gráficas podemos ver las comparaciones de los dos heurísticos usando el algoritmo Simulated Annealing, en cuanto al beneficio obtenido en las diferentes ejecuciones.





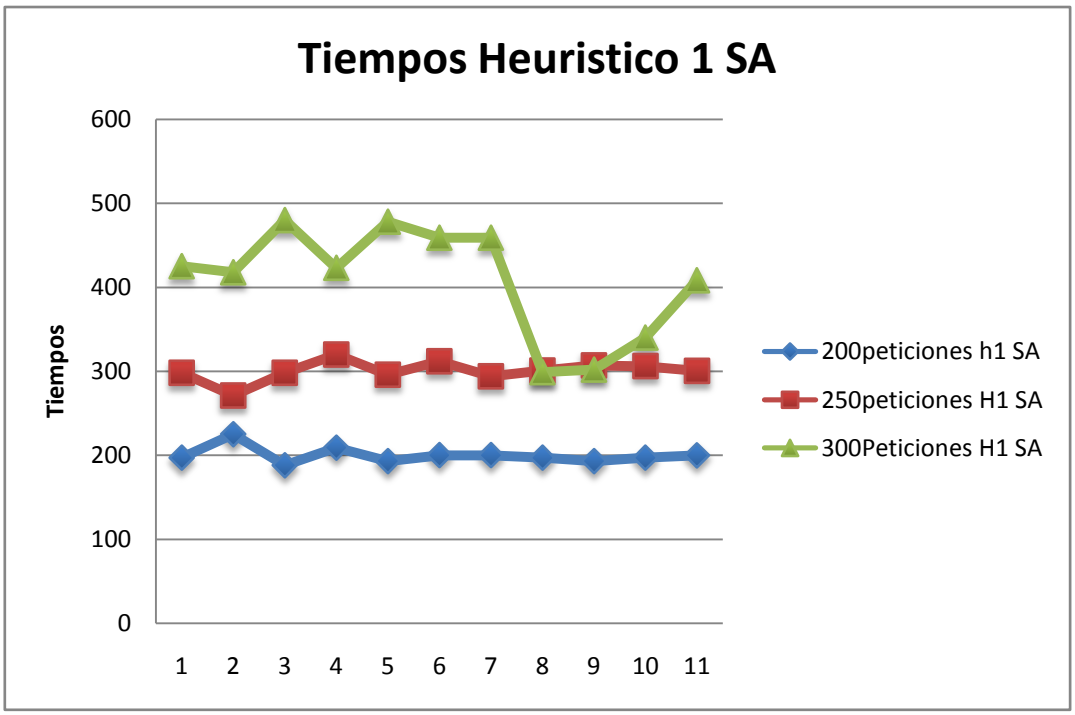
Del mismo modo que con Hill Climbing, el heurístico 1 obtiene, en media, mejores resultados en cuanto a beneficios obtenidos por transportes.

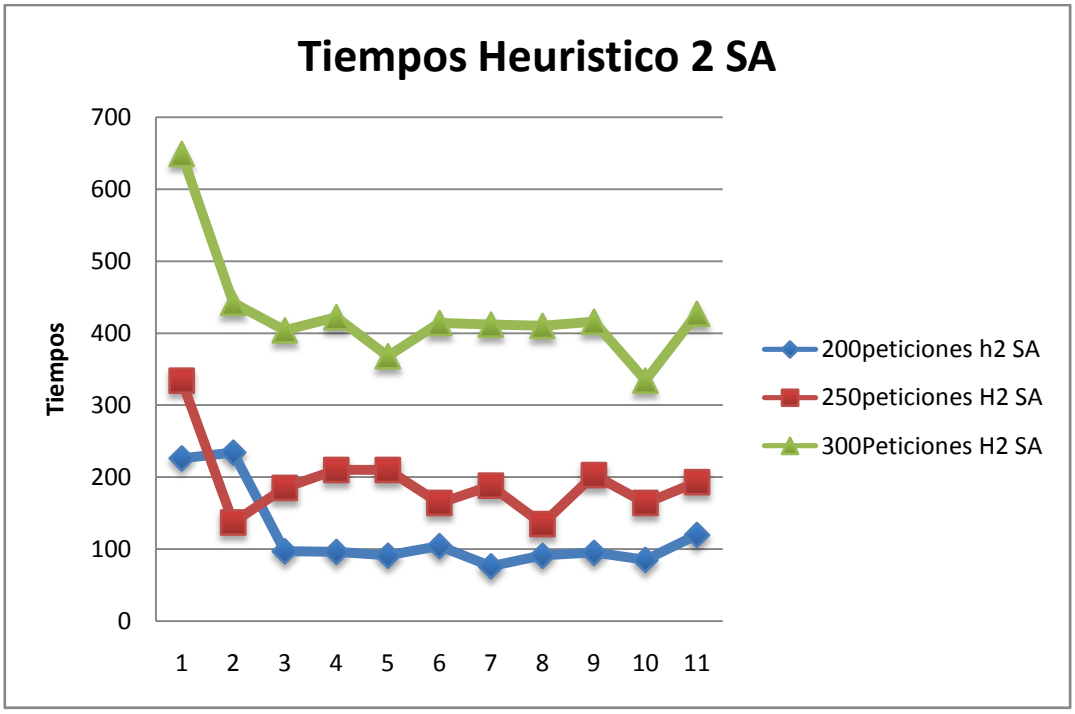


	200	250	300
Beneficios h1 SA	69372	82385	80500
Beneficios h2 SA	65013	67036	77116

Comparación de tiempos con Simulated Annealing

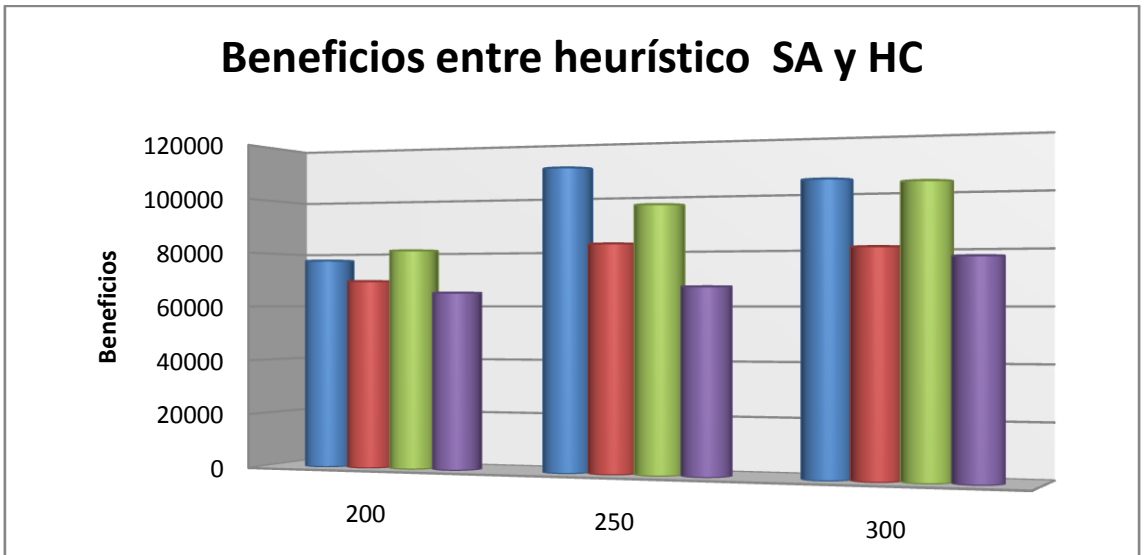
En las siguientes gráficas podemos ver las comparaciones de los dos heurísticos usando el algoritmo Simulated Annealing, en cuanto al beneficio obtenido en las diferentes ejecuciones.





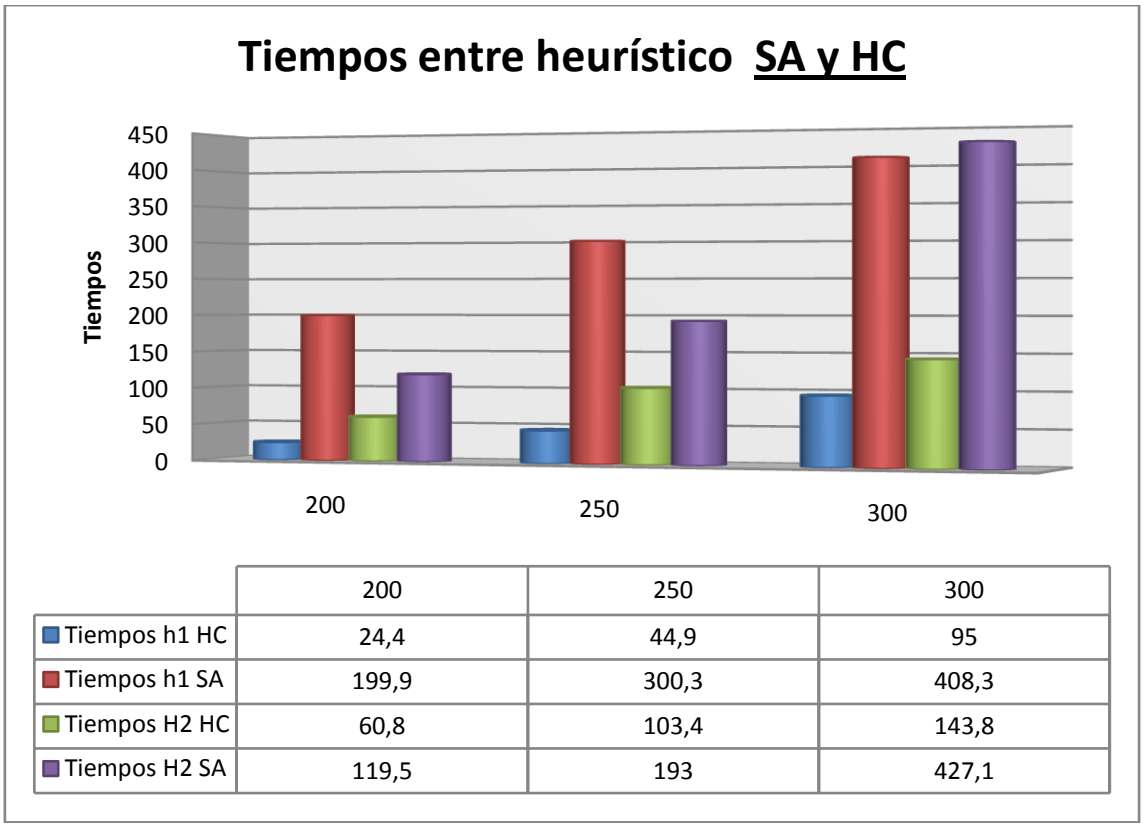
Comparación de los dos algoritmos en cuanto a los diferentes heurísticos

En la siguiente gráfica hemos juntado los datos finales medios, obtenidos para las 4 posibilidades de combinación de los algoritmos con los heurísticos. En conclusión tenemos que con Hill Climbing, para ambos heurísticos, en media obtiene mejores resultados en cuanto al beneficio total obtenido con los transportes.



	200	250	300
Beneficios h1 HC	77106	109778	103838
Beneficios h1 SA	69372	82385	80500
Beneficios H2 HC	80633	96079	102635
Beneficios H2 SA	65013	67036	77116

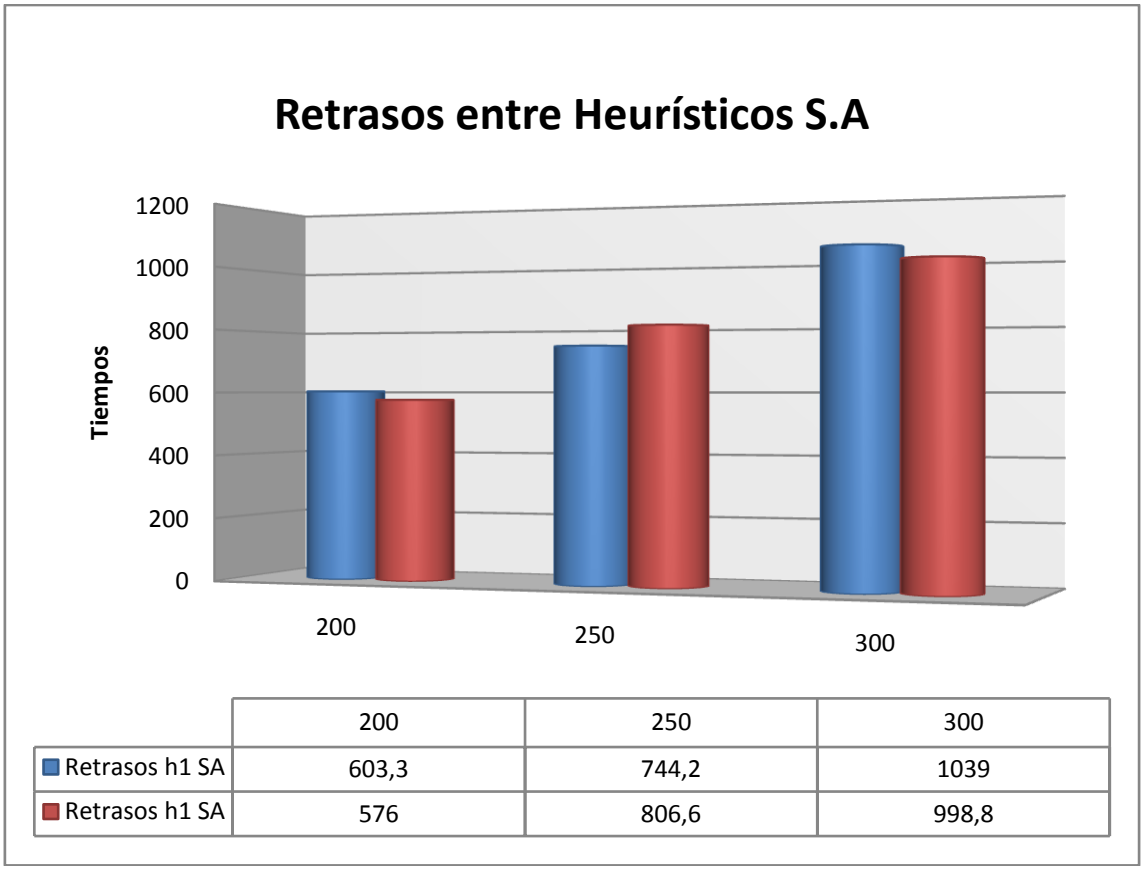
En la siguiente gráfica hemos juntado los datos finales medios, obtenidos para las 4 posibilidades de combinación de los algoritmos con los heurísticos. En conclusión tenemos que con Hill Climbing, para ambos heurísticos, en media el tiempo de ejecución es notablemente menos que con Simulated Annealing.



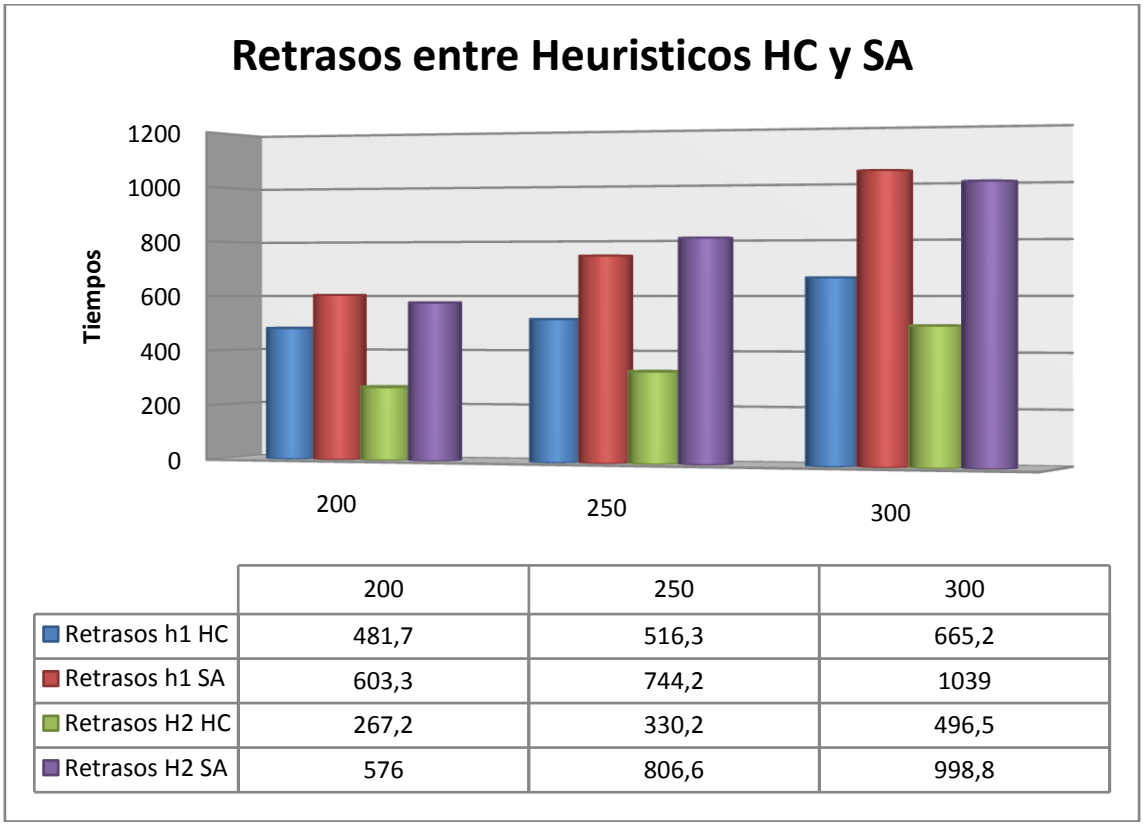
	200	250	300
Tiempos h1 HC	24,4	44,9	95
Tiempos h1 SA	199,9	300,3	408,3
Tiempos H2 HC	60,8	103,4	143,8
Tiempos H2 SA	119,5	193	427,1

Comparación de los retrasos

Para acabar el experimento, añadimos las gráficas con los datos obtenidos en realición con los retrasos que han resultado de las diferentes ejecuciones utilizando Simulated Annealing, y la comparación de estos resultados, con los de Hill Climbing ya obtenidos de otro experimento.



	200	250	300
Retrasos h1 SA	603,3	744,2	1039
Retrasos h1 SA	576	806,6	998,8



	200	250	300
Retrasos h1 HC	481,7	516,3	665,2
Retrasos h1 SA	603,3	744,2	1039
Retrasos H2 HC	267,2	330,2	496,5
Retrasos H2 SA	576	806,6	998,8

7.6.4 Conclusiones

Podemos concluir con los datos obtenidos, que el algoritmo Hill climbing en media obtiene mejores resultados para ambos heurísticos y en cuanto a tiempo de ejecución, que el Simulated Annealing.

7.7 Séptimo experimento: Repercusión de las flotas de camiones.

Para llevar a cabo este experimento podemos disponer de diferentes flotas de camiones. Dado el escenario del primer experimento, tenemos que probar tres variaciones en las que tengamos un 50% de camiones de un tamaño y 25% de los otros dos tamaños. El heurístico que debemos utilizar es el primero, donde buscamos maximizar la ganancia obtenida con los transportes, y el algoritmo será Hill Climbing. Una vez realizado el experimento debemos de explicar la diferencia entre las soluciones de los distintos escenarios.

7.7.1 Planteamiento del experimento

El planteamiento del experimento es el siguiente:

- Se generarán los escenarios necesarios con las tres variaciones de flotas (50% camiones de un tamaño y 25% de los otros tamaños) y los demás datos utilizados en el primer experimento.
- Obtendremos los datos para todas las ejecuciones.
- Compararemos los resultados obtenidos en los diferentes escenarios.

7.7.2 Resultados esperados

Los resultados que esperamos en este experimento son los siguientes:

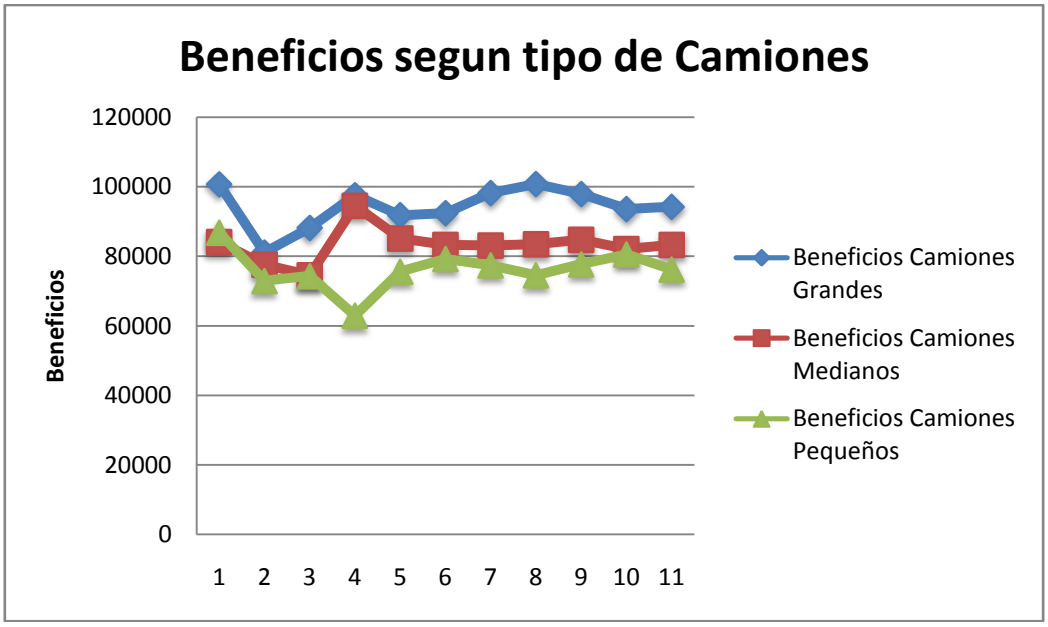
- En cuanto al beneficio, esperamos obtener mayor cuando el 50% de los camiones son grandes, ya que hay más probabilidad de que los camiones que reparten los pedidos a su hora sean grandes y por tanto tenemos más beneficio.
- De la misma forma, esperamos tener más beneficio con un 50% de camiones medianos respecto a los camiones pequeños.
- En cuanto al tiempo de ejecución, se espera que cuando hay un 50% de camiones grandes tarde más en media, ya que al tener más capacidad permiten más posibilidades a la hora de intercambiar peticiones.

7.7.3 Análisis de los datos obtenidos

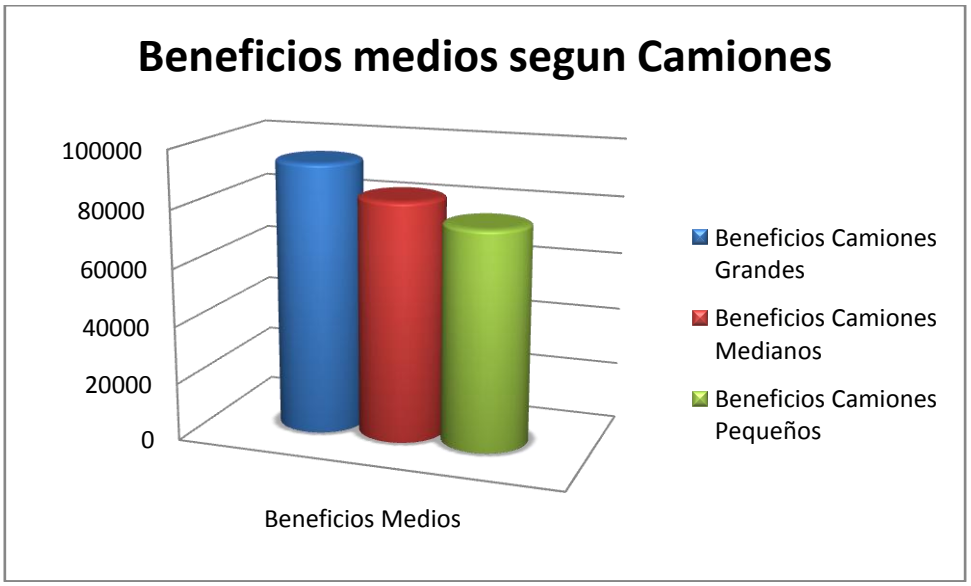
A continuación explicamos los datos más relevantes obtenidos al realizar las ejecuciones para los diferentes escenarios. Por un lado miraremos las repercusiones de las diferentes flotas respecto al beneficio obtenido y por otro lado compararemos el tiempo de ejecución.

Comparación de beneficios

Como podemos ver en la siguiente gráfica, se obtienen los datos esperados, donde el escenario con un 50% de camiones grandes producen en general mejores beneficios que los escenarios con 50% medianos o pequeños. Además en media, los escenarios con un 50% de camiones medianos también producen mejores beneficios que cuando hay un 50% de camiones pequeños.

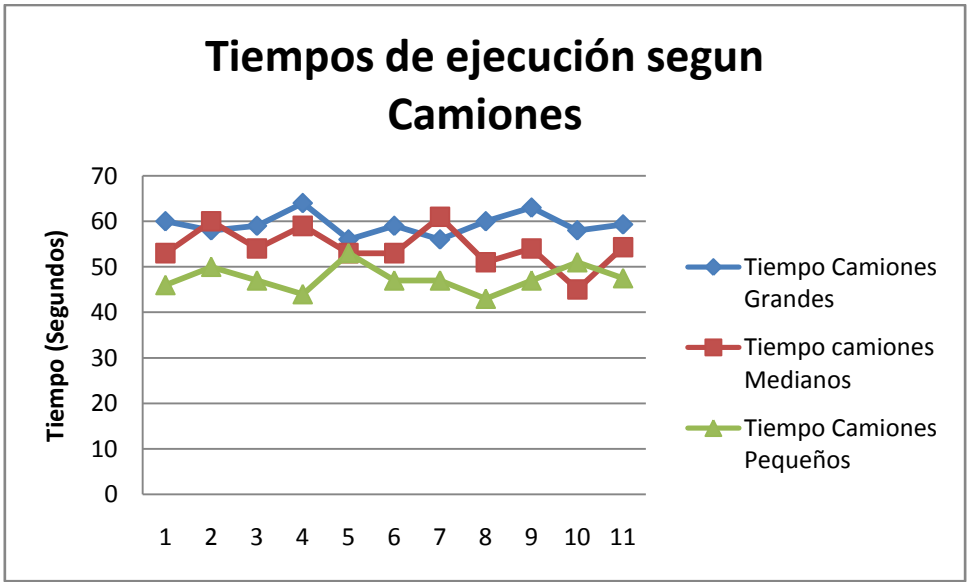


En la siguiente gráfica vemos los valores medios de estas ejecuciones para las diferentes flotas de camiones y vemos claramente los resultados obtenidos y explicados de la gráfica anterior.

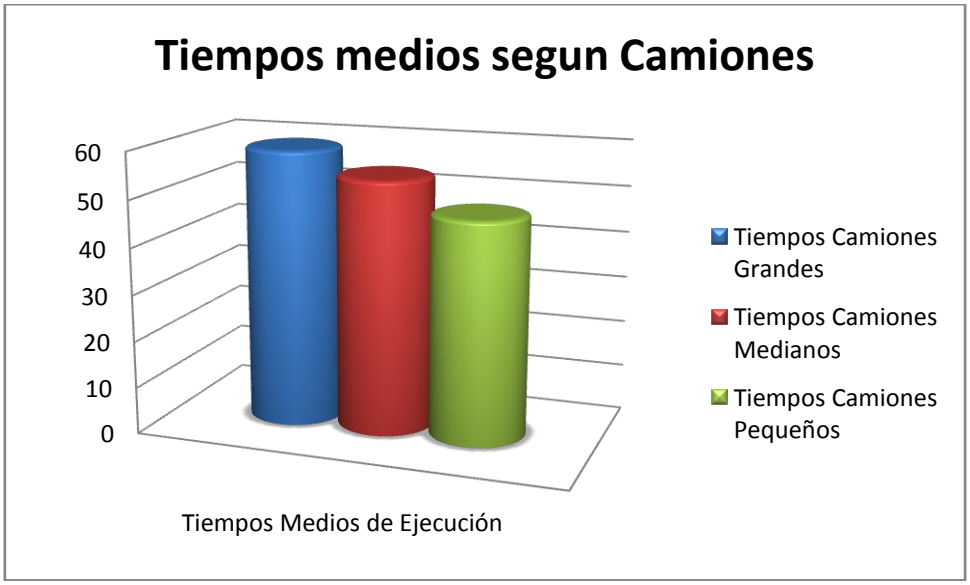


Comparación de tiempos de ejecución

Como podemos ver en la siguiente gráfica, en general, cuando el escenario tiene un 50% de camiones grandes el programa tarda más en acabar. Esto coincide con los resultados esperados ya que al haber más camiones grandes caben más peticiones en cada asignación por lo tanto permite más posibles sucesores de cada estado.



En la siguiente gráfica mostramos las medias para cada tipo de flota de camiones para ayudar visualmente a ver este resultado.



7.7.4 Conclusiones

Con los resultados obtenidos podemos concluir que cuantos más camiones grandes tengamos, mayores beneficios tendremos y a la vez, mayor será el tiempo de ejecución. Ello concuerda con los supuestos de más cantidad de camiones grandes, más capacidad, más beneficios, más posibilidades de intercambiar paquetes entre ellos por lo que obtenemos un mayor tiempo de ejecución.

7.8 Octavo experimento: Repercusión de probabilidades horarias.

En el último experimento que debemos realizar de la práctica, debemos tener algunas horas con más peticiones que otras. Debemos considerar el primer escenario, pero con la variación de que las 4 primeras horas tengan el doble de probabilidad de tener peticiones que el resto y comparar los resultados obtenidos con los del primer experimento. Para ello debemos usar el algoritmo Hill Climbing.

Nosotros para ver si era relevante hemos realizado el experimento también para las últimas 4 horas y las hemos comparado.

7.8.1 Planteamiento del experimento

El planteamiento del experimento es el siguiente:

- Para llevar a cabo el experimento primero debemos realizar una pequeña modificación en el código en la que las horas generadas ahora sean 14 y a la hora de asignarlas hacemos módulo 10 para que las 4 primeras tengan el doble de probabilidad.
- Después utilizando los escenarios del experimento 1 ejecutamos el programa.
- Realizamos la modificación en el código para poner más probabilidad a las 4 últimas horas.
- Ejecutamos el programa para los mismos escenarios.
- Comparamos los resultados obtenidos en las dos ejecuciones y con los resultados del primer experimento.

7.8.2Resultados esperados

Los resultados que esperamos con este experimento son los siguientes:

- Se espera que tanto con las 4 primeras horas con más probabilidad como las últimas 4 horas, obtengan más beneficios que con la solución equiprobable del primer experimento.

- En cuanto al tiempo de ejecución esperamos valores un poco menores que con los del primer experimento, ya que será más lógico cargar los camiones grandes en las horas donde más probabilidad de peticiones haya, y esto da lugar a menos cambios de estado.

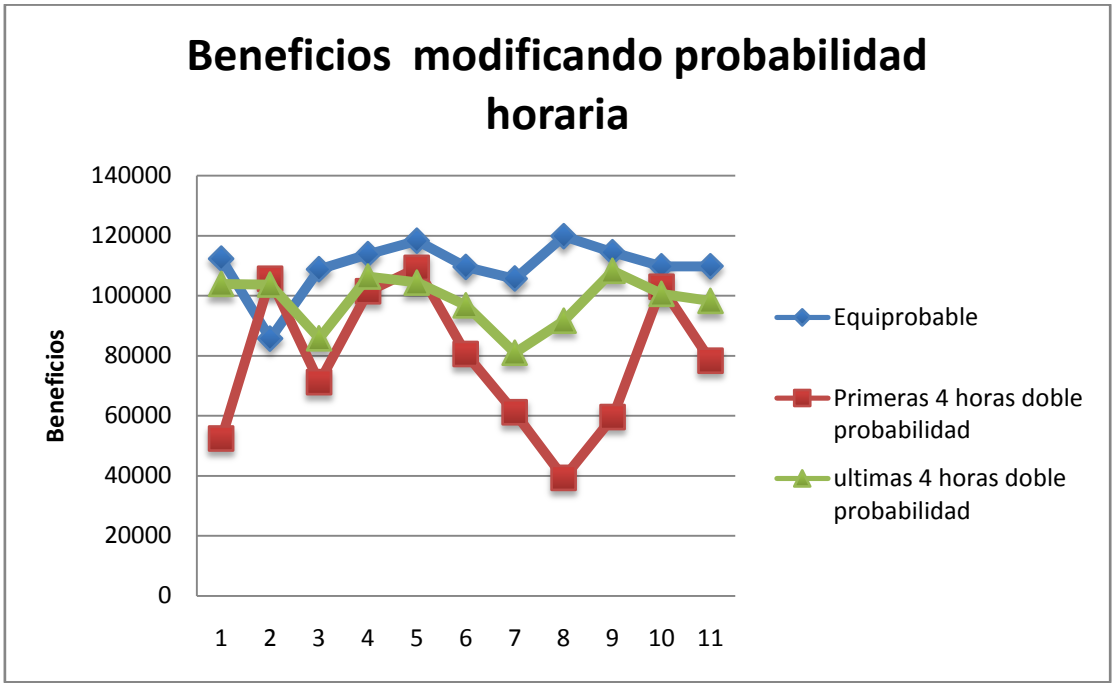
7.8.3 Análisis de los datos obtenidos

A continuación explicamos los análisis de los datos más relevantes obtenidos en el experimento.

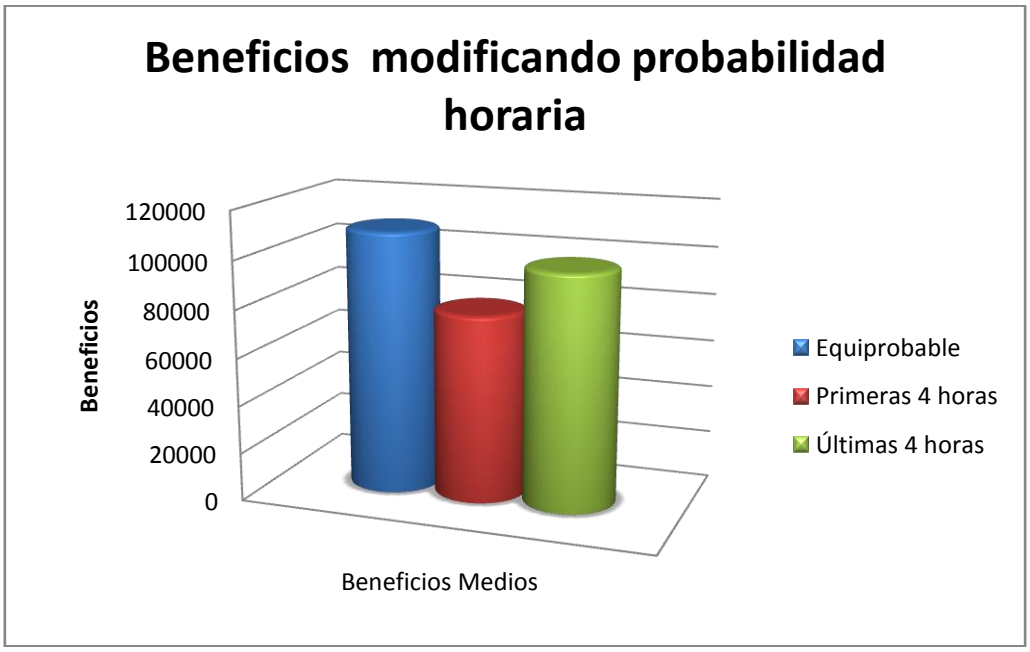
Comparación de Beneficios

En la siguiente gráfica vemos los resultados obtenidos para los 3 escenarios que queremos comparar. Pensábamos que obtendríamos más beneficios, pero pensándolo con más

detenimiento los datos obtenidos son más lógicos, ya que, al tener muchas peticiones que entregar en las 4 primeras horas del día, si tenemos pocos camiones grandes, no podremos cargar estos con todas las peticiones necesarias y obtendremos muchos retrasos. De manera parecida pasa con las 4 últimas horas con más probabilidades, obtenemos más beneficios que con las primeras 4 horas con más probabilidad pero aun seguimos teniendo menos beneficios que con los resultados del primer experimento.

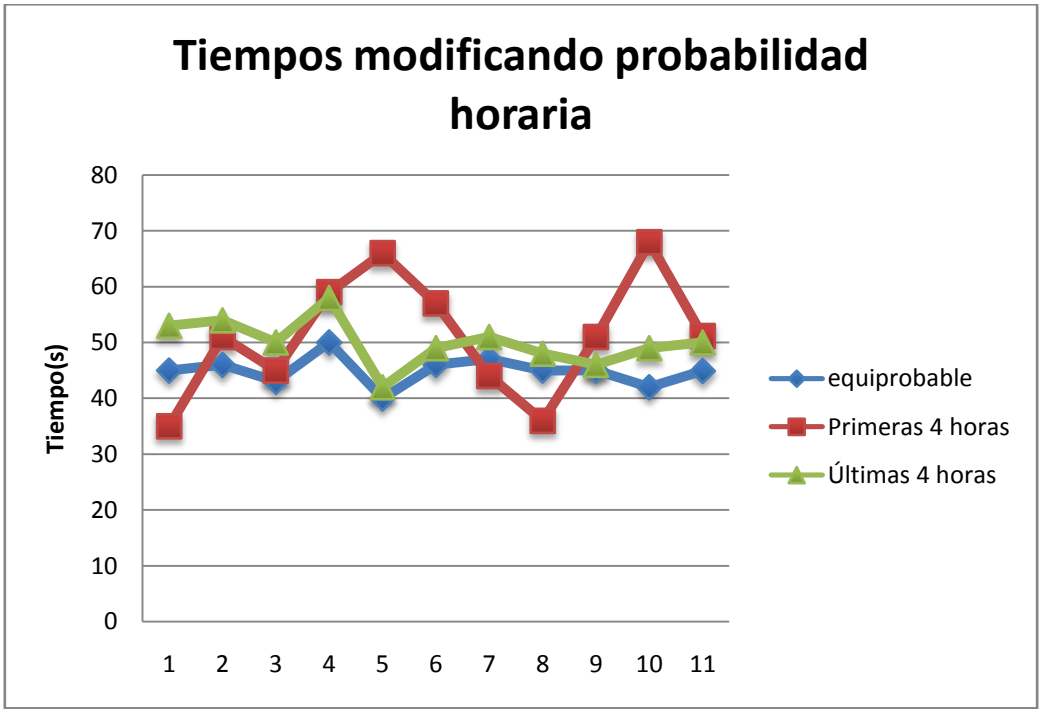


A continuación mostramos una gráfica con las medias de los resultados obtenidos donde vemos más claramente la influencia de los repartos de las peticiones.

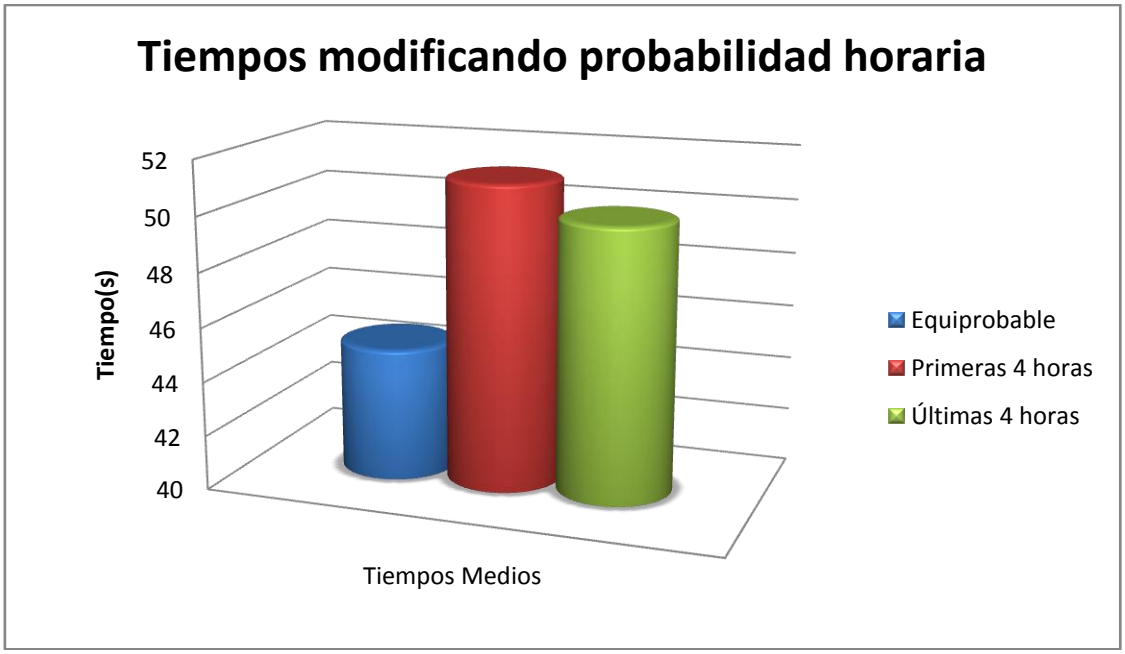


Comparación de tiempos de ejecución

A continuación mostramos la gráfica de los datos obtenidos al comparar los tiempos de ejecución comparando los diferentes escenarios. Como vemos, para el escenario con más probabilidad en las 4 primeras horas hay muchos picos ya que esto depende del número de camiones grandes que hay en el problema, sin embargo para las 4 últimas horas con más probabilidad tenemos un tiempo mayor que en el escenario equiprobable, pero mucho más relacionado.



A continuación vemos las medias de los tiempos de ejecución de los diferentes escenarios.



7.8.4 Conclusiones

Como conclusión podemos decir que aumentando la probabilidad de asignar peticiones a horas determinadas no produce más beneficios, al contrario, en media es un cambio que empeora el resultado. Además el tiempo de ejecución se ve aumentado.

8. Problemas que nos hemos encontrado

A la hora de realizar la práctica nos hemos encontrado con algunos problemas.

8.1 Problema 1

El problema principal ha sido, que al realizar algunos experimentos nos daba una excepción de que sobrepasábamos la memoria del sistema debido al gran número de peticiones a asignar a los camiones, y el gran número de estados sucesores que suponía. Para resolver este problema hemos encontrado una solución. Puesto que nosotros ejecutábamos la aplicación desde *eclipse*, buscamos en internet que podíamos hacer para evitar esta excepción. Encontramos un blog donde una persona explicaba cómo había resuelto su problema, con las librerías de AIMA también.

8.2 Solución 1

Uno de los motivos es que por defecto la memoria de la virtual machine dentro de eclipse, está limitada. Para poder aumentar la memoria podemos añadir un parámetro opcional dentro del menu-> RUN…

Disponemos de la pestaña *arguments*. Donde tendremos argumentos del programa y argumentos de la VM (Virtual machine).

Para indicarle que queremos más memoria RAM utilizamos le comando:

- Xmx1024m

Esto indica que queremos reservar 1024 MegaBytes de memoria para la VM de java dentro de eclipse. Podemos aumentar este número tanto como nos permita nuestro ordenador, dependiendo de la RAM que tengamos instalada. Miramos la foto, que se verá más claro dónde y cómo poner el parámetro para aumentar la memoria en eclipse, para aplicaciones java.

Además de este parámetro hemos utilizado el siguiente:

- Xms512m

Lo cual indica que queremos reservar 512 MegaBytes para el tamaño inicial del heap.

8.3 Problema 2

El segundo problema que encontramos fue con la función de clone de los diferentes elementos. En ciertos apartados, el sistema presentaba un comportamiento anómalo que fue muy difícil detectar. Tras emplear el debugger y ver qué es lo que estaba sucediendo realmente en el sistema, nos dimos cuenta que el sistema realizaba una "*Shallow Copy*" en lugar de una "*Deep Copy*". Pese a implementar un ".clone()" propio en cada clase y aplicarlo a todos los elementos, seguía existiendo un fallo a nivel de las referencias y en la función de sucesores: al realizar una copia local del estado, seguían modificándose los valores del estado no clonado.

8.4 Solución 2

La forma que encontramos para solucionar el problema de la *"shallow copy"* fue implementar una función dentro de nuestro propio clone que copiase elemento a elemento. Pensamos que quizás haya alguna manera de que sea más sencillo o mejor que nuestra implementación, pero no queríamos desviar el tiempo dedicado a ello. Nuestro objetivo primordial era que funcionase el sistema y luego optimizarlo. Creemos que en ciertos aspectos podría mejorar el rendimiento una copia del estado de forma más veloz. No llegamos a implementar una mejora de nuestra solución funcional principal pero corregimos el fallo implementando un sistema de *"deep copy".*

9. Conclusiones

Hemos realizado un análisis del rendimiento de los diferentes algoritmos (Hill climbing y simulated annealing) aplicado al problema concreto del enunciado. Por curiosidad hemos implementado un heurístico diferente mezclando ambos de los anteriores y los resultados han sido similares. Pensamos que en este caso el segundo heurístico está íntimamente relacionado con el primero por las condiciones relativas a su aplicación. Por ello, creemos que en parte es un heurístico "menos informado" que el primero en el mismo tema que el primero.

Nos ha extrañado no emplear en mayor medida el segundo heurístico, pese a sospechar que a altos niveles de peticiones se comportarían de forma muy parecida teniendo el segundo mayor demora.

En cuanto al desarrollo de la práctica, hemos podido ver la complejidad de éstos algoritmos, llegando algunos juegos de prueba a tardar más de 7 horas en pasar (en totalidad de las ejecuciones), inclusive en ordenadores que están "*al día*" a nivel de componentes. Hemos empleado exclusivamente los ordenadores de la facultad para ejecutar las pruebas para que se realizasen en situaciones similares. Sin embargo, las pruebas del tercer heurístico y de 1000 peticiones se realizaron en otros ordenadores (pero los datos no han sido tenido en cuenta a en detalle pues no calificarían para un diseño experimental, sino son de carácter anecdótico). Los ordenadores dónde se han realizado las pruebas son un macbook con cpu core2Duo (similar a los de la facultad, en el cual se observaba un rendimiento idéntico a los ordenadores de la facultad), y un ordenador con cpu core i7 y 6GB de Ram (de los cuales 3 asignados a la maquina virtual Java). En éste último se observaron mejores resultados a nivel de tiempos pero similares en resultado.

Si tuviésemos que emitir una valoración global sobre los algoritmos, el algoritmo de Hill Climbing es más de nuestro agrado que el algoritmo de Simulated Annealing, pues no solamente es fácil de evaluar, sino también más fácil de utilizar y más veloz / ofrece mejores resultados.