27 OCTUBRE 2022

PRÁCTICA DE BÚSQUEDA LOCAL

ALGORITMOS BÁSICOS PARA LA INTELIGENCIA ARTIFICIAL

Gesiarz, Victor
Planell Bosch, Noel Nathan
Ni, Huilin

Facultad de Informática
de Barcelona

ÍNDICE

INTRODUCCIÓN

En esta primera práctica de la asignatura vamos a adaptar un problema propuesto por los profesores a un programa con tal de poder ejecutar los algoritmos *Hill Climbing* y *Simulated Annealing*.

Descripción del problema

Con las nuevas regulaciones europeas para el mercado de la energía, la relación entre productores y consumidores de energía es más directa y surgen más oportunidades para optimizar oferta y demanda. Somos una empresa que gestiona un parque de centrales eléctricas de diferentes tipos que permiten producir diariamente cierta cantidad de megavatios.

Los diferentes tipos de centrales eléctricas que tenemos son los siguientes, cada una tiene un cierto rango de producción de megavatios, un coste de mantenerse activa por día y un coste de cerrar la central:

TIPO	PRODUCCIÓN	COSTE MARCHA	COSTE PARADA
A	250 a 750 Mw	*Prod* · 5 + 2000	1500€
B	100 a 250 Mw	*Prod* · 8 + 1000	500€
C	10 a 100 Mw	*Prod* · 15 + 500	150€

Si no servimos toda la electricidad generada, esta se perderá, pero asumiremos el coste de producirla, por lo tanto nos interesa producir lo justo y necesario para satisfacer la demanda de los clientes y así obtener el máximo beneficio posible.

Por otra parte, tenemos también los clientes a los que suministramos nuestra electricidad. Estos están clasificados en tres categorías: *extra grandes (XG), muy grandes (MG)* y *grandes (G).* Cada categoría tiene un consumo dentro de un rango y además el cliente especifica la prioridad con la que quiere ser servido, puede elegir entre ser *garantizado* o *no garantizado*, donde los *garantizados* nos tenemos que asegurar de servirlos siempre (por lo tanto tienen un precio de contrato más elevado) y los *no garantizados* pueden sufrir de cortes de energía. En la siguiente tabla podemos ver los precios de los diferentes contratos de nuestros clientes:

		PRIORIDAD		
CLIENTE	CONSUMO	GARANTIZADO	NO GARANTIZADO	INDEMNIZACIÓN
XG	5 a 20 Mw	40 euros / Mw	30 euros / Mw	5 euros / Mw
MG	2 a 5 Mw	50 euros / Mw	40 euros / Mw	5 euros / Mw
G	1 a 2 Mw	60 euros / Mw	50 euros / Mw	5 euros / Mw

Además, podemos ver que hay un coste de indemnización, este es cuando no podemos servir a un cliente *no garantizado*, entonces tenemos que devolverle ese coste de indemnización (el cliente no para de pagar su tarifa, si no que nosotros devolvemos parte como indemnización).

En el problema estamos ubicados en un espacio 2D, en un área cuadrada de 100 km por 100 km. Cada cliente y central tiene unas coordenadas (x, y) enteras en Km, que indican donde están ubicados. Por esto hay que tener en cuenta que el transporte de electricidad no es perfecto. Por el camino se producen pérdidas por la disipación de la electricidad en forma de calor y otros factores. Entonces cuanto más lejos se encuentre un cliente más tendremos que producir para poder suministrar su demanda de electricidad, pero seguiremos cobrandole solamente lo que pide, no lo que tenemos que producir para satisfacerle.

DISTANCIA	PÉRDIDA
Hasta 10 Km	0
Hasta 25 Km	10%
Hasta 50 Km	20%
Hasta 75 Km	40%
Más de 75 Km	60%

El problema

Queremos resolver el problema para un día en concreto. Para ello dispondremos de una lista de las *Centrales* que tenemos con la información necesaria de ellas, es decir: el tipo, la producción y sus coordenadas; por otra parte tendremos la lista de *Clientes* que contiene: el tipo, el consumo, el tipo de contrato y la localización.

Al ejecutar el programa la solución nos tendrá que mostrar en qué forma deberíamos asignar a los clientes para poder obtener el máximo beneficio.

Objetivo

Crear un programa que sea capaz de resolver el problema descrito de la forma más eficiente implementando dos algoritmos: *Hill Climbing* y *Simulated Annealing.* Posteriormente se deberán realizar unos experimentos para comprobar la calidad y la utilidad de los diferentes parámetros y funciones.

Para la creación del programa disponemos de un archivo creado por los profesores llamado *abia_energia.py* que contiene las clases de *Cliente, Clientes, Central, Centrales* y *Energía*, con las cuales tenemos que trabajar y adaptarnos a ellas para ejecutar el código usandolas.

Búsqueda local

Nos encontramos ante un problema de búsqueda local porque no tratamos de encontrar la solución más óptima, es decir, el máximo beneficio, ya que encontrar este requeriría demasiado costo computacional y dependiendo el tamaño del problema, podría ser computacionalmente imposible. En este caso, ganar 100 euros más o menos no es importante, lo que sí es importante sería poder calcular el problema muy rápido para poder organizar el suministro eléctrico a tiempo real. Es por ello que lo que buscamos es una solución cercana al máximo beneficio, que se ejecute en el mínimo tiempo posible y que la podamos parar cuando necesitemos, entonces los algoritmos que mejor cumplen estos requisitos son los de busqueda local.

PARÁMETROS DEL PROBLEMA

Para poder realizar la ejecución del programa, tal y como hemos visto en clase, creamos una clase que guarde los parámetros del problema, en este caso:

- *clients_vector*: Una lista con los diferentes clientes generados.
- *power_plants_vector*: Una lista con las diferentes centrales generadas.

```
class ProblemParameters(object):
    def __init__(self, clients_vector: Clientes, power_plants_vector: Centrales):
        self.clients_vector = clients_vector
        self.power_plants_vector = power_plants_vector

    def __repr__(self):
        return f"clients_vector={self.clients_vector}\n \
                \npower_plants_vector={self.power_plants_vector})"
```

REPRESENTACIÓN DEL ESTADO

Para representar el estado del problema, en la clase *StateRepresentation,* tenemos 3 principales variables:

- *params*: La cual es la que contiene los parámetros del problema, es decir, la lista de Clientes y la lista de Centrales.

- *c_pp*: La cual es una lista que asigna los clientes a una central. Esta representación está hecha mediante los índices de las centrales y clientes, es decir, que la posición 0 de la lista se referirá al cliente de posición 0 en la lista de clientes, y el valor de esta posición, que es un índice de una central, indicará la central a la que está asignado el cliente.

- *gain*: Está es una simple variable tipo *float* que nos sirve para guardar el valor de las ganancias.

```
class StateRepresentation(object):

    def __init__(self, params: ProblemParameters,
                 c_pp: List[int],
                 remain: List[float],
                 consum: List[List[float]],
                 gain: float,
                 prices: List[List[float]],
                 last_action = None,
                 count_actions = [0,0,0]) -> None:

        self.params = params
        self.c_pp   = c_pp
        self.remain = remain
        self.consum = consum
        self.prices = prices
        self.gain   = gain

        self.last_action = last_action
        self.count_actions = count_actions
        self.misplaced_clients = misplaced_clients
```

Además, hemos creado otras variables más que nos son útiles para optimizar el código y para hacer los experimentos:

- *remain*: Es una lista en la cual almacenamos la energía que falta por asignar de las centrales, en la posición 0 estará el sobrante de energía de la central de posición 0. De esta forma no tenemos que calcularlo cada vez que lo necesitemos.

- *consum*: Esta variable es una matriz constante de dos dimensiones. La hemos añadido para ahorrar cálculos, ya que en nuestros operadores frecuentemente calculamos cual es la energía que consume en bruto un X cliente en una central Y. Por lo tanto, como este valor solamente depende del consumo neto del cliente y de la distancia entre central y cliente, lo cual son dos valores constantes, hemos decidido calcular estos valores en el generador inicial y guardarlos en esta variable. Entonces tenemos una matriz donde sus filas son las centrales (la fila 0 corresponde a la central de posición 0 en la lista de centrales) y las columnas son los clientes.
  Por ejemplo, si se quiere acceder al consumo bruto del tercer cliente en la cuarta central, habrá que poner *consum[4][3]*.

- *prices*: Esta es otra matriz en la cual guardamos todos los posibles precios. Tiene filas columnas:
  - En la primera fila guardamos el coste de tener apagada una central.
  - En la segunda fila guardamos el coste de encender una central.
  - En la tercera fila guardamos el coste de indemnización de cada cliente.
  - No nos guardamos lo que paga cada cliente ya que esto solo lo necesitamos al principio para calcular las ganancias, luego, tal y como está planteado el problema, no nos sirve ya que no lo volvemos a usar porque cada cliente si o si tiene que pagar ese precio (esto lo modificamos en el último experimento).

- *last_action* y *count_actions*: Estas variables las usamos para los experimentos que veremos más adelante y nos sirven para saber cuánto usamos cada uno de los operadores y ver que combinación es la mejor.

- *misplaced_clients*: En el último experimento empezamos en un estado inicial vacío, y por tanto, no es válido para la definición de este problema, ya que los clientes con suministro garantizado no están asignados a ninguna central (tampoco lo están los no garantizados) . Es por ello, que hemos tenido que modificar el heurístico agregando esta nueva variable para que nos conduzca a una solución válida.

- Analizar el tamaño del espacio de búsqueda.

ESTADO INICIAL

Para el estado inicial hemos decidido crear una clase estática para poder organizar el código mejor. Esta clase estática se llama *InitialState* y en ella podemos crear varios estados iniciales. También guardamos en ella las funciones para calcular los parámetros necesarios para ejecutar correctamente el programa.

- *simple_state*: Como constructores de estado inicial hemos hecho dos relativamente sencillos y bastante similares, pero como veremos más adelante en los experimentos, brindan resultados considerablemente diferentes.

  Los dos constructores funcionan de la misma forma. Empiezan seleccionando una central a la que tratarán de añadir todos los clientes con suministro garantizado que puedan, a la que haya un cliente que no puedan añadir, seleccionan la siguiente central, en el caso de que ya hayan mirado toda la lista de centrales, vuelven a la primera central.

  Si hay un cliente que ha mirado si puede ser suministrado por todas las centrales y no queda suficiente suministro en ningúna, el constructor del estado inicial se detiene dando un error: "Not enough power plants to generate initial state". Para este error en algunos casos si se optimizará mejor el espacio entre centrales dentro del constructo entrarían los clientes, pero sería un generador de estado inicial muy greedy que haría perder el sentido al hill climbing y al simulated annealing.

  La diferencia entre los dos constructores es el orden en como van seleccionando las centrales. Aunque parezca que está diferencia no cambia nada, analizando cómo se genera el problema podemos ver que si lo hace. Las centrales y los clientes se generan aleatoriamente pero se guardan por su tipo. En otras palabras, en la lista *Centrales,* primero van las centrales del tipo A que son las que más energía suministran, luego las del tipo B cuyo suministro de energía es medio, y por último, las del tipo C que su suministro de energía es el más bajo.

  En cierta forma podemos decir que las centrales están bastante ordenadas por su suministro, y es por ello, que el orden en como recorramos la lista de centrales si afecta a cómo se genera el estado inicial (Si el número de clientes es muy elevado, el efecto se disminuye).

  Para indicar con qué orden se quiere que el simple_state seleccione las centrales está el parámetro *asc* dentro de la función*,* si el valor de asc es True, lo hará cogiendo las centrales más grandes primero, si es False, cogerá las más pequeñas primero.

- *remain_consum*: Esta es una simple función que utilizamos para calcular las variables ya mencionadas anteriormente de *remain* y *consum*.

- *calculate_gain*: Aquí calculamos el beneficio inicial para poder guardarlo como una variable y también calculamos la variable prices que nos ahorra cálculos durante la ejecución.

Cabe destacar también que para las listas de mayor tamaño que usamos en el programa intentamos aplicar las listas de *numpy*, con las cuales obtenemos un mejor rendimiento y mejor tiempo de ejecución dado a que son más rápidas que las de Python.

Hemos escogido estos generadores de estado inicial principalmente porque el funcionamiento de estos es muy sencillo y comparando ambos, podemos analizar si es más efectivo llenar con prioridad o bien las centrales más grandes o bien las más pequeñas, como podremos ver en la realización de los experimentos más adelante.

El coste de hallar un estado inicial en ambos casos el caso peor es el número de clientes por número de centrales, aunque como en el estado inicial también generamos constantes como remain, gain, el coste de la función generate initial state es bastante más elevado.

OPERADORES

Inicialmente pensamos en todos los posibles operadores que puede tener el problema y fuimos descartando en función de sus características y nuestras necesidades.

Operadores seleccionados

Los operadores que hemos usado para el problema son los siguientes:

- *MoveClient*: Este operador permite mover un cliente a una central. Este puede estar ya asignado a otra central y, por tanto, lo estaríamos cambiando de central, o puede que aún no estuviera asignado. Es un operador completamente necesario porque es la forma principal de cómo asignaremos una central a un cliente. Su ramificación es multiplicar el número de centrales abiertas por el número de clientes. En este operador está incluido implícitamente el operador de cerrar una central, cuando un cliente en una central es el último y es movido a otra central, la central se cierra reduciendo su coste de mantenimiento.

```
class MoveClient(Operator):
    def __init__(self, id_client, id_destination_PwP):
        self.id_client = id_client
        self.id_destination_PwP = id_destination_PwP

    def __repr__(self) -> str:
    return f"Client {self.id_client} has been moved to \
             power plant {self.id_destination_PwP}"
```

- *SwapClients*: Intercambiamos dos clientes de dos centrales distintas. Ambos clientes deben estar asignados a una central para que funcione el operador. Podría parecer que teniendo el operador *MoveClient* no sería necesario este operador. No obstante, puede darse la situación de que dos clientes para llegar al caso óptimo tengan que intercambiarse pero que sus centrales estén tan llenas que no puedan hacerlo con el operador *MoveClient*. Su factor de ramificación es el número de clientes al cuadrado.

```
class SwapClients(Operator):
    def __init__(self, id_client1, id_client2):
        self.id_client1 = id_client1
        self.id_client2 = id_client2

    def __repr__(self) -> str:
        return f"Swap between Client {self.id_client1} and \
                 Client {self.id_client2}"
```

- *TurnOnPowerPlant*: Este operador lo usaremos para poder abrir una central cerrada y así empezar a mover clientes a esta central. Se puede utilizar mientras exista alguna central cerrada. Su factor de ramificación es el número de centrales cerradas.

```
class TurnOnPowerPlant(Operator):
    def __init__(self, id_pp: int):
        self.id_pp = id_pp

    def __repr__(self) -> str:
        return f"Power Plant {self.id_pp} has been turned on"
```

Seleccionar estos tres operadores nos permitiría dependiendo del estado inicial movernos por todos los posibles espacios de soluciones. La condición para que esto se dé es que en el estado inicial los clientes no garantizados no estén asignados a ninguna central.

Operadores descartados

- *MergePowerPlants*: El operador fusionar centrales se podría formularse de muchas formas, la que nosotros pensamos es que esté operador seleccionará dos centrales y si en la que quedará mayor espacio entraban los clientes de la otra central, los moviera todos ha está, cerrando así una central. Las principales razones por las cuales descartamos este operador es que era bastante complejo de hacer. Además, aumentaría considerablemente el coste de ejecución ya que tendría una ramificación equivalente al número de centrales al cuadrado y su coste de aplicación sería muy elevado por los bucles que tendría. Por último, tampoco conseguimos idear una una implementación que a priori ayudará a resolver el problema, ya que con los tres operadores que seleccionamos, ya podemos abarcar todo el espacio, en cierta forma este operador sería una combinación de muchos operadores de *MoveClient*.

- *RemoveNGClients:* El operador quitar un cliente no garantizado nos permitiría alcanzar todos los posibles espacio solución si partieramos de un estado inicial con clientes no garantizados asignados a alguna central. No obstante, decidimos que nuestro generador de estados inicial solamente asignará clientes garantizados. Por lo tanto, este operador ya no es necesario, ya que de llegar a utilizarlo no nos interesaría que nuestro heurístico lo seleccionará porque eso implicaría que nos hemos equivocado asignando a una central un cliente no garantizado que no debíamos y este operador se ha tenido que ocupar de revertir este mal movimiento. Su factor de ramificación sería igual al número de clientes no garantizados.

- TurnOffPowerPlant: El operador de cerrar una central hemos decidido añadirlo implícitamente en el operador *MoveClient*, cuando se mueva un cliente se comprueba si la central ha quedado vacía, y si lo ha hecho, cierra esa central, lo que quiere decir, que aumenta los beneficios ya que reduce su coste de mantenimiento y bloquea que otros clientes puedan acceder a esa central, a no ser de que se vuelva a abrir con el operador *TurnOnPowerPlant*. Implementarlo dentro del operado *MoveClient* hace que sea mucho más eficiente porque evitas generar ramificaciones de este.

HEURÍSTICOS

Hemos creado 3 heurísticos para el problema, los dos primeros porque cada uno analiza en lo que falla el otro, y el tercero, porque queríamos que fuera la combinación ideal de los dos primeros, para que así potenciará todos los aspectos importantes del problema.

Nuestro objetivo en el problema es maximizar los beneficios. Entonces, tal y como se define el problema, hemos deducido que se logra un estado bastante cercano al óptimo si se añaden todas los clientes en el mínimo número de centrales y la energía que falta por suministrar suministrar a clientes no garantizados (explicaremos nuestro razonamiento en los experimentos).

Es por ello que los factores que hemos de potenciar es que persiga añadir clientes buscando el beneficio máximo, pero a la vez reordene el suministramiento de las centrales para que suministren toda la energía posible. También se podría medir que los clientes consumieran de centrales más cercanas para evitar las pérdidas energéticas por distancia. No obstante, consideramos que los dos primeros factores eran más importantes, y por eso, nos centramos en tratar de potenciarlos.

- *gain_heuristic*: El primer heurístico que realizamos es el más obvio cuando pensamos en maximizar las ganancias, y es precisamente usar las ganancias como heurístico. Para ello, como ya hemos explicado anteriormente, tenemos una variable que se ocupa de guardar los beneficios, la variable *gain*. Entonces, en el heurístico devolvemos su valor. Con este heurístico potenciamos que se asignen clientes siempre que se pueda.

```
def gain_heuristic(self) -> float:
    return self.gain
```

- *entropy_heuristic*: Este heurístico se basa en el uso de la entropía, con este tratamos de maximizar el uso de las centrales, es decir, con este heurístico potenciamos que las centrales o bien queden vacías o bien queden completamente llenas, en otras palabras, este heurístico potencia que se suministre toda la energía posible. Para lograr este propósito, nuestra función aplica la fórmula de la entropía sobre las centrales, definiendo la occupancy como 1 - la energia sin consumir de una central entre la producción máxima de está central.

```
def entropy_heuristic(self) -> float:
    total_entropy = 0
    for id, remain in enumerate(self.remain):
        max_prod = self.params.power_plants_vector[id].Produccion
        occupancy = 1 - (remain/max_prod)
        if occupancy > 0:
            total_entropy +=  - (occupancy * log(occupancy))
    return -total_entropy
```

- *heurístico combinado*: Este combina los dos anteriores heurísticos con el objetivo de sacar las ventajas de cada uno de ellos. Nuestro objetivo es que asigne el máximo número de clientes llenando al máximo las centrales hasta que ya no puedan suministrar nada más de energía.
  Para lograrlo, sumamos los dos heurísticos, el gain y el entropy. Como tienen unidades diferentes y no los estamos normalizando, tendrá prioridad el que produzca valores más grandes, que en este caso es el heurístico de ganancias. Es decir, con este heurístico primero priorizaremos que asigne todos los clientes posibles y luego reordene los clientes entre las centrales para tratar de añadir más clientes.

```
def combined_heuristic(self) -> float:
    return self.gain_heuristic() + self.entropy_heuristic()
```

EXPERIMENTOS

Ahora llegamos al apartado de los experimentos del problema. En esta parte hemos creado diferentes escenarios y hemos puesto a prueba nuestro programa jugando con los diferentes parámetros y funciones que hemos creado con tal de intentar encontrar la mejor forma de ejecutarlo y poder extraer conclusiones de nuestra implementación de los diversos algoritmos.

Todos los datos obtenidos están también guardados en el excel adjunto.

Experimento 1

Objetivo:
El objetivo de este experimento es seleccionar qué operadores usaremos para realizar los próximos experimentos.

Parametros:
Tal y como se indica en el enunciado de la práctica, usaremos como parámetros del problema:

- 1000 clientes
- Proporción de clientes: 25 % (XG), 30 % (MG) y 45 % (G)
- Proporción de clientes garantizados: 75%
- Número de centrales: 5 (A), 10 (B) y 25 (C)
- La técnica que usaremos es el *Hill Climbing*

OBSERVACIONES	Hay operadores que aportan más a la solución final que otros.
PLANTEAMIENTO	Combinaremos los diferentes operadores que tenemos (usando siempre el move client ya que sin este no podríamos encontrar una solución más óptima) para medir el efecto de cada operador o combinación de operadores en la solución final.
HIPÓTESIS	Los mejores operadores serán el de *MoveClient* y *TurnOnPowerPlant*.
MÉTODO	· Elegiremos 10 semillas aleatorias, una para cada réplica. · Usaremos el generador de estado inicial 1. · Ejecutaremos un experimento con cada posible combinación de operadores que contenga el operador *MoveClient*. · Usaremos el algoritmo de Hill Climbing. · Mediremos cuántas veces se ejecuta cada uno de los operadores y los resultados de las ejecuciones para realizar la comparación.

Para hacer este primer experimento ejecutaremos el código del archivo *experiment_1.ipynb* en el cual generamos 10 semillas random y podemos ejecutar el programa para cada una, cambiando los posibles operadores y heurísticos.

Resultados de los experimentos

Primero haremos una prueba para ver qué acciones estamos utilizando. Para esto tenemos la variable *count_actions* en la representación del estado, en la cual guardamos cuantas veces aplicamos cada uno de los operadores. Haremos una ejecución con 10 semillas utilizando todos los operadores para cada heurístico disponible.

- *gain_heuristic*:

	Move client	Swap Client	Turn On Power Plant
Media	**22,7**	**0**	**0**

Podemos ver que estamos utilizando solamente el operador *MoveClient* y los demás no se utilizan en ningún momento. Esto se debe a que el heurístico de las ganancias que estamos utilizando no está teniendo en cuenta estos operadores dado a que no generan ningún beneficio directo.

Por lo tanto podemos concluir en este caso que para el heurístico *gain_heuristic* el único operador que necesita es el de *MoveClient*.

- *entropy_heuristic*:

	Move client	Swap Client	Turn On Power Plant
Media	**9,1**	**12,4**	**0**

En esta ejecución podemos ver que este heurístico las aprovecha más, intentando buscar nuevas combinaciones para buscar un estado final óptimo. Sin embargo, sigue sin utilizar el operador para encender centrales, debido a que tampoco lo encuentra útil en ninguna situación. Posteriormente tendremos que ver si merece la pena utilizar ambos, solo uno o no utilizar este heurístico.

- *combined_heuristic*:

	Move client	Swap Client	Turn On Power Plant
Media	**20,86**	**4,14**	**0**

Por último, utilizando este heurístico nos damos cuenta de que nuevamente no utiliza el operador de encender central y por ende, ninguno de los tres heurísticos considera que ese operador debería ser utilizado.

Ahora miraremos los beneficios, clientes servidos y tiempo ejecución de los casos que vemos oportunos comprobar, ninguno tiene el de encender una central ya que ninguno lo utiliza:

Gain_heuristic (MoveClient):

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**2,545**	*Clientes servidos*	*Beneficio*
754,6	**121856,6**		**776,9**	**122667,6**

Entropy_heuristic (MoveClient, SwapClients)

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**116,36**	*Clientes servidos*	*Beneficio*
754,6	**121856,6**		**762,30**	**122295,60**

Entropy_heuristic (MoveClient)

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**6,44**	*Clientes servidos*	*Beneficio*
754,6	**121856,6**		**773,5**	**122489,1**

Combined_heuristic (MoveClient, SwapClients)

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**110,83**	*Clientes servidos*	*Beneficio*
754,6	**121856,6**		**776,4**	**122647,6**

Combined_heuristic (MoveClient)

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**6,26**	*Clientes servidos*	*Beneficio*
754,6	**121856,6**		**776,4**	**122647,6**

Podemos ver que las mejores ejecuciones son aquellas en las que hemos utilizado solo el MoveClient, lo cual no extraña ya que el programa tiene que realizar muchísimas menos ejecuciones que utilizando también el SwapClients. También, los beneficios alcanzados son mejores o iguales con solo el primer operador.

Conclusiones

Primero hemos visto que el operador de *TurnOnPowerPlant* no está funcionando como esperábamos ya que ninguno de los heurísticos lo considera útil. Esto lo podríamos solucionar creando un nuevo heurístico que sí que considerara este operador o por otra parte podríamos combinar el operador de *MoveClient* y *TurnOnPowerPlant* en uno.

En segundo lugar, los heurísticos *entropy_heuristic* y *combined_heuristic* proporcionan resultados similares en cuanto a las ganancias y los clientes servidos, pero el tiempo de ejecución es mucho mayor, por lo que lo mejor es usar el *gain_heuristic*.

Al contrario de nuestra hipótesis, hemos podido determinar que el *MoveClient* es nuestro operador base, ya que como hemos visto en los experimentos no merece la pena utilizar los otros. Los dos heurísticos descartados intentan jugar más con los operadores disponibles para hallar una mejor solución, pero lamentablemente esta tarda mucho más y no consigue superar a las otras.

Por lo tanto nos quedamos con el heurístico ***gain_heuristic*** y el operador ***MoveClient.***

Experimento 2

Objetivo: Determinar qué estrategia de generación de la solución inicial da mejores resultados para la función heurística usada en el experimento 1, con el escenario de este y usando el algoritmo de Hill Climbing. A partir de estos resultados, fijar también la estrategia de generación de la solución inicial para el resto de experimentos.

OBSERVACIONES	Pueden haber estados iniciales que favorezcan a ciertos operadores
PLANTEAMIENTO	Emplearemos el algoritmo de Hill Climbing con nuestros dos estados iniciales utilizando en cada caso un heurístico diferente de los 3 que tenemos (*gain*, *entropy* y la fusión de estos, *entropy + gain*).
HIPÓTESIS	*Initial state 1* es **mejor** que *Initial state 2* en el sentido que nos aportará más beneficios y menos tiempo de ejecución.
MÉTODO	· Elegiremos 10 semillas aleatorias, una para cada réplica. · Usaremos el algoritmo de Hill Climbing. · Utilizaremos en cada caso el operador elegido anteriormente *MoveClient*. · Adicionalmente del heurístico elegido en el primer experimento, utilizaremos los otros 2 heurísticos para cada uno de nuestros 2 estados iniciales para analizar sus resultados. · Anotaremos los beneficios y los clientes servidos tanto del estado inicial como del final, junto con sus tiempos de ejecución para, más adelante, calcular sus medias y comparar las diferentes ejecuciones.

Para hacer este experimento emplearemos el código del archivo *experiment_2.ipynb,* en el cual generamos también 10 semillas random y podemos ejecutar el programa para cada una, cambiando los posibles operadores y heurísticos.

Resultados de los experimentos

Primero ejecutaremos el *Initial State 1* con cada uno de los heurísticos. A continuación podemos observar las medias obtenidas de cada uno de los datos registrados.

Initial State 1 - Gain (H1)

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**3,09**	*Clientes servidos*	*Beneficio*
750,70	**118113,80**		**775,50**	**119013,30**

Initial State 1 - Entropy (H2)

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**8,70**	*Clientes servidos*	*Beneficio*
750,70	**118113,80**		**777,20**	**118767,80**

Initial State 1 - Entropy + Gain (H3)

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**6,70**	*Clientes servidos*	*Beneficio*
750,70	**118113,80**		**774,40**	**118998,80**

Como bien podemos observar, de los 3 heurísticos aplicados, el que mejor rendimiento nos da es el caso del heurístico 1 (H1) *gain*. A pesar de no tener la mejor media en la cantidad de clientes servidos, sí que nos da el mejor tiempo y beneficio de los tres casos, por lo que podríamos decir que compensa esos 1,7 clientes de diferencia que tiene con el caso que más clientes servidos hay.

Ahora procederemos a comparar dichos resultados del primer *Initial State* con los obtenidos con el segundo.

Initial State 2 - Gain (H1)

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**7,83**	*Clientes servidos*	*Beneficio*
750,70	**108886,80**		**793,90**	**111008,30**

Initial State 2 - Entropy (H2)

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**14,15**	*Clientes servidos*	*Beneficio*
750,70	**108886,80**		**772,50**	**109753,30**

Initial State 2 - Entropy + Gain (H3)

ESTADO INICIAL		TIEMPO (s)	ESTADO FINAL	
Clientes servidos	*Beneficio*	**18,23**	*Clientes servidos*	*Beneficio*
750,70	**108886,80**		**792,80**	**111002,80**

Con el *Initial State 2* podemos ver claramente que la ejecución realizada con el heurístico (H1) *gain* es el mejor de los 3, ya que en este caso, no solo nos da el mayor beneficio y tiempo, sino que también es el que más clientes servidos acaba teniendo.

Conclusiones

Después de estudiar todas las posibles combinaciones de nuestros heurísticos con los estados iniciales de los cuales disponíamos, hemos llegado a la conclusión que el mejor estado inicial es el **primero** que, junto al heurístico del primer experimento, nos da la combinación con mejor rendimiento y beneficios.

Es verdad que el número de clientes servidos no es el mayor, pero como lo que nos interesa es maximizar el beneficio generado, el número de clientes es negligible en este caso.

También, hemos comparado los resultados con los otros dos heurísticos y efectivamente podemos observar como el segundo generador de estado inicial, pese a tener muy pocas diferencias del primero influye mucho en el tiempo de ejecución y los beneficios generados.

Experimento 3

Objetivo: Determinar los parámetros que dan mejor resultado para el *Simulated Annealing* con el mismo escenario, usando la misma función heurística y los operadores y la estrategia de generación de la solución inicial escogidos en los experimentos anteriores.

Tal y como se comenta en el enunciado de la práctica, la mayor dificultad para realizar este experimento es ajustar los valores del *Simulated Annealing* ya que la única indicación que tenemos son los resultados que podemos obtener experimentalmente, eso nos obligará a hacer pruebas exhaustivas para ver cómo se comporta el problema.

PLANTEAMIENTO	Experimentaremos con los parámetros del Simulated Annealing hasta obtener un resultado más óptimo.
HIPÓTESIS	El algoritmo de Simulated Annealing es mejor que el de Hill Climbing, por lo que nos proporcionará un mejor rendimiento y resultados que este último.
MÉTODO	· Usaremos las mismas 10 semillas aleatorias generadas para el experimento 2. · Ejecutaremos un experimento con cada posible combinación entre estados iniciales y heurísticos. · Usaremos el algoritmo de Simulated Annealing. · Mediremos diferentes parámetros para realizar la comparación.

Para hacer este experimento emplearemos el código del archivo *experiment_3.ipynb,* en el cual utilizamos las mismas 10 semillas aleatorias generadas para el experimento anterior, y podemos ejecutar el programa para cada una, cambiando los posibles operadores y heurísticos.

Tal y como hemos planteado el programa, para este experimento nos surge un problema ya que hemos acabado escogiendo solo un operador. El *Simulated Annealing* al escoger las acciones lo hace aleatoriamente, pero como en nuestro caso solo tiene una acción, siempre escoge la misma. Además, hemos tenido un problema de falta de tiempo para realizar algunos experimentos

Resultados de los experimentos

Experimentando con varios valores en el *Simulated Annealing*, hemos podido ver que cambiando los valores no obtenemos un cambio significativo de los resultados. El único valor que influye más es el de el límite.

	k = 20, lam = 0.005, limit = 100		
	TIEMPO	BENEFICIO (final - inicial)	CLIENTES
Media	**2,67**	**380,00**	**23,40**

	k = 1, lam = 0.005, limit = 100		
	TIEMPO	BENEFICIO (final - inicial)	CLIENTES
Media	**2,74**	**405,50**	**25,00**

	k = 50, lam = 0.005, limit = 100		
	TIEMPO	BENEFICIO (final - inicial)	CLIENTES
Media	**3,23**	**352,50**	**25,20**

	k = 20, lam = 0.9, limit = 100		
	TIEMPO	BENEFICIO (final - inicial)	CLIENTES
Media	**2,79**	**388,00**	**24,20**

	k = 20, lam = 0.005, limit = 2		
	TIEMPO	BENEFICIO (final - inicial)	CLIENTES
Media	**0,30**	**30,50**	**2,30**

	k = 1, lam = 0.005, limit = 200		
	TIEMPO	BENEFICIO (final - inicial)	CLIENTES
Media	**5,29**	**635,50**	**44,60**

Simulated Annealing		
Time	Gain	Clients
5,29	**635,50**	**44,60**

Hill Climbing		
Time	Gain	Clients
3,09	**899,50**	**24,80**

Conclusiones

Lo que podemos ver es que aumentando el límite del algoritmo conseguimos aumentar los beneficios más que en los otros casos. Pero al solo tener una acción, se queda sin estas rápidamente y si aumentamos el límite demasiado nos sale un error.

También comparando los resultados del mejor caso de *Simulated Annealing* con los del *Hill Climbing*, vemos que el segundo tarda menos y obtiene un mejor beneficio, pero de media asigna menos clientes.

Experimento 4

Objetivo: En este escenario, para 1000 clientes, con los parámetros y la misma función heurística utilizados anteriormente y usando el algoritmo *Hill Climbing,* evaluaremos el tiempo de ejecución para hallar la solución aumentando de 40 en 40 el número de centrales con el objetivo de encontrar la tendencia de este.

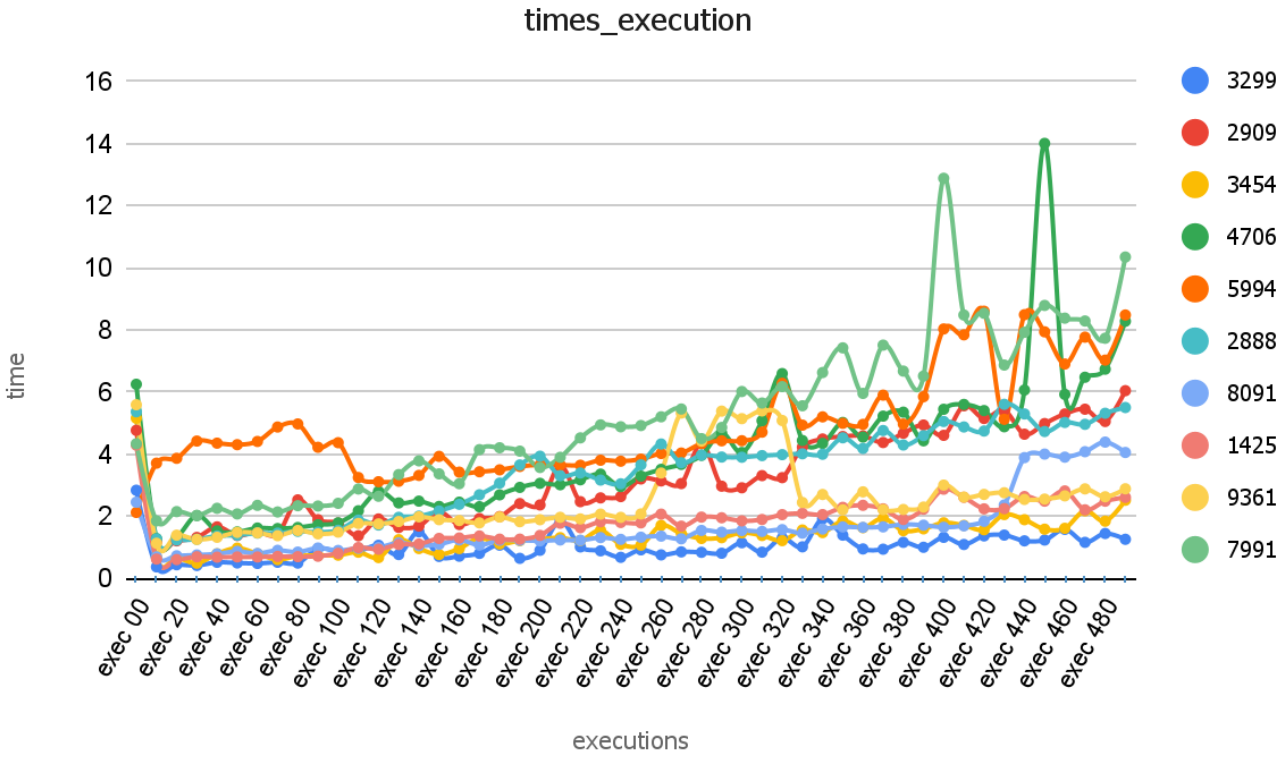
PLANTEAMIENTO	Ejecutaremos dicho programa aumentando el número de centrales de forma progresiva hasta hallar la tendencia de los tiempos de ejecución.
HIPÓTESIS	El tiempo de ejecución irá aumentando a medida que aumentemos el número de centrales.
MÉTODO	· Elegiremos 10 semillas aleatorias. · Ejecutaremos el algoritmo tantas veces hasta que empecemos a ver una tendencia. · Por cada ejecución del algoritmo aumentaremos el número de centrales proporcionalmente por 40. · Usaremos el algoritmo de Hill Climbing, usando solo el operador *MoveClient* y el heurístico *gain_heuristic*. · Mediremos el tiempo de ejecución del algoritmo y del generador de estado inicial.

Para hacer este experimento emplearemos el código del archivo *experiment_4.ipynb,* en el cual generamos 10 semillas random y podemos ejecutar el programa para cada una, cambiando los posibles operadores y heurísticos.

Resultados de los experimentos

Ejecutando el algoritmo varias veces hemos visto que para que se empiece a ver una tendencia más clara necesitamos hacer un mínimo de 500 ejecuciones, pero como esto tardaría demasiado, hemos decidido hacerlas de 10 en 10 y por tanto ejecutamos el código 50 veces. El tiempo de ejecución, como ahora podremos ver, no aumenta drásticamente, pero el tiempo de creación del estado inicial sí, porque cuantas más centrales pongamos, más tarda este código.

Como podemos ver en los siguientes gráficos de las ejecuciones y los tiempos de ejecución, este último va aumentando a medida que aumentamos el número de centrales, lo cual concuerda con nuestra hipótesis.

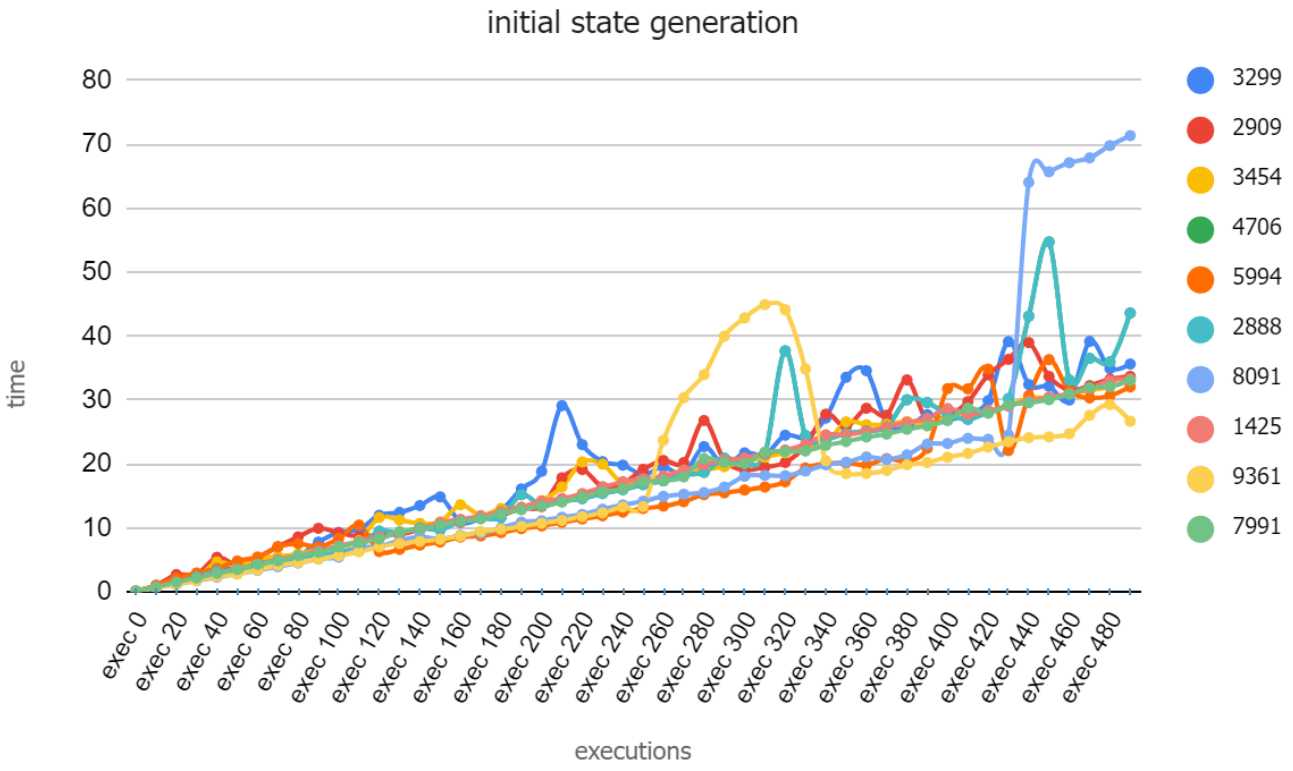


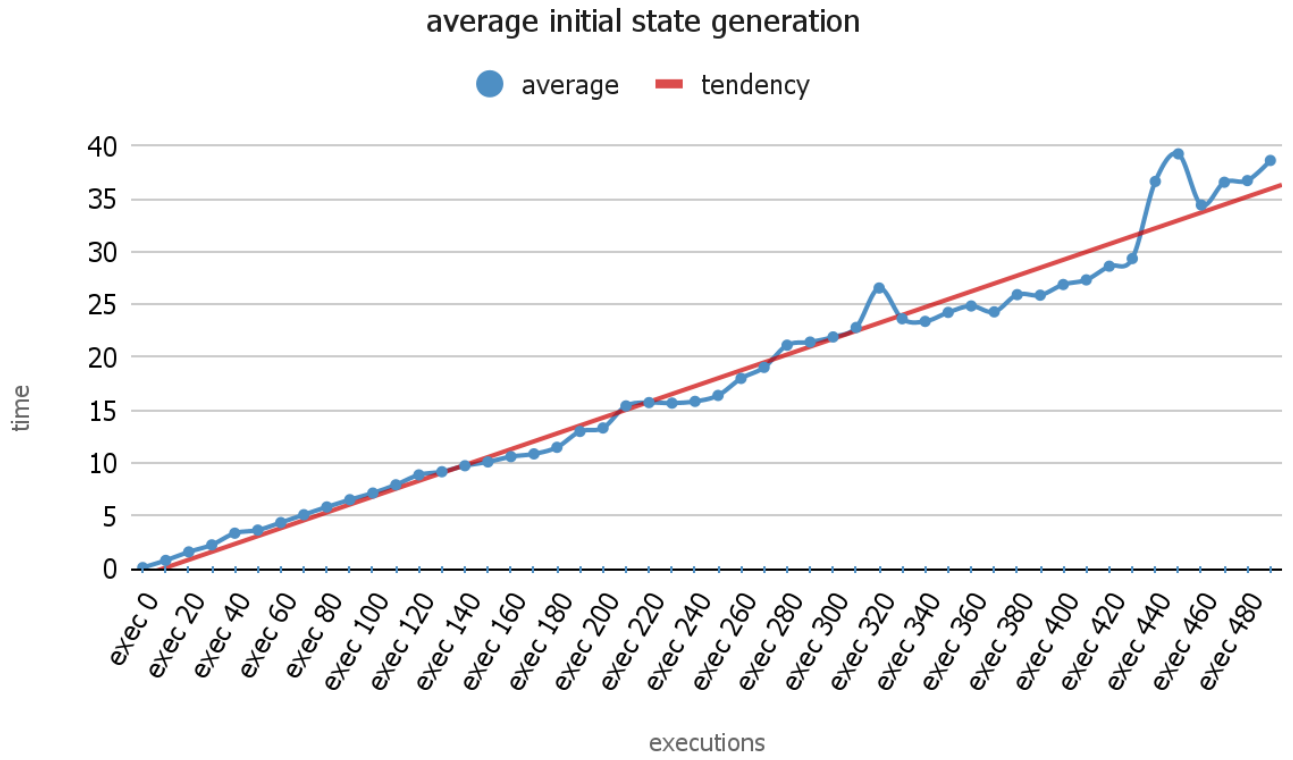
Si miramos los tiempos de ejecución medios entre todas las ejecuciones realizadas obtenemos el siguiente gráfico:

Podemos ver que el tiempo de ejecución tiene una tendencia a aumentar linealmente.

Un elemento que destaca mucho en la gráfica es que en todas las ejecuciones, la primera de todas siempre tarda bastante más que la mayoría de las demás. Creemos que esto se debe a...

Por otra parte, si medimos el tiempo de ejecución de la generación del estado inicial podemos observar como este también aumenta mucho, pero más drásticamente.





Conclusiones

Tras la realización de los experimentos, hemos podido comprobar que la tendencia del tiempo de ejecución es aumentar de forma lineal. Además, podemos ver que lo que más afecta al tiempo de ejecución no es el algoritmo *Hill Climbing,* sino la generación del estado inicial, que ocupa mucho más tiempo y crece mucho más rápido que no el de la ejecución.

Experimento 5

Objetivo: El objetivo de este experimento es determinar qué deberíamos añadir a nuestro heurístico para lograr que cuando se termine de ejecutar el hill climbing o el simulated annealing, devuelvan siempre un estado válido. Para poner esto a prueba, este experimento se realizará empezando en un estado vacío donde ningún cliente está asignado a ninguna central, y por tanto, es un estado no válido porque los clientes garantizados no tienen central.
El verdadero objetivo del experimento es encontrar dla mínima k que garantiza que el estado final es válido.

Para poder conseguir esto hemos tenido que modificar un poco la estructura de nuestro programa, concretamente hemos añadido:

- Un nuevo generador de estado inicial el cual genera un estado vacío donde ninguna cliente está asignado a ninguna central.
- Hemos creado un nuevo heurístico llamado *fix_state_heuristic*, el cual es igual al gain_heuristico, pero con la diferencia de que por cada cliente garantizado no suministrado se resta un valor K al beneficio.
- Un nuevo operador llamado *SuperMoveClient*. Este operador literalmente es hacer *TurnOnPowerPlant y MoveClient* a la vez. Nos hemos visto obligados a crear este operador porque nunca ejecutaba el operador *TurnOnPowerPlant* ya que nuestro heurístico se basa en beneficio y el añadir una central siempre reduce el beneficio y como no reduce el número de clientes garantizados no suministrados, añadir una central no mejora el heurístico. Es por ello, que hemos añadido este operador, que al hacer a la vez el abrir una central y añadir un cliente, el coste de añadir una central se compensa por la reducción de clientes garantizados no suministrados.

PLANTEAMIENTO	Estableceremos dos K, una elevado donde el estado final sea válido, una pequeña donde el estado no sea válido.
HIPÓTESIS	El tiempo de ejecución irá aumentando a medida que aumentemos el número de centrales.
MÉTODO	· Elegiremos 10 semillas aleatorias. · Estableceremos dos K, una elevada donde el estado final sea válido (K1), una pequeña donde el estado no sea válido(K2). · Cogeremos sumaremos las dos K y su suma la dividiremos entre dos, calcularemos de nuevo el experimento con el resultado de está operación. (K3 = (K1 + K2) /2) · Repetiremos el anterior paso pero está vez las dos K que tendremos son: Si el resultado de K3 es un espacio válido: K3 y K1 Si el resultado de K3 es un espacio no válido: K3 y K2 Repetiremos tantas veces este paso hasta que agotemos lo suficiente el rango mínimo de K · Para cada ejecución. mediremos ...

Resultados de los experimentos

Hill Climbing

SIMPLE STATE + GAIN

TIEMPO (s)	BENEFICIO	CLIENTES MAL PUESTOS	CENTRALES ABIERTAS
3,50	119316,5	0	22

EMPTY STATE + FIX STATE + K = 10000

TIEMPO (s)	BENEFICIO	CLIENTES MAL PUESTOS	CENTRALES ABIERTAS
168,94	82599	0	38

EMPTY STATE + FIX STATE + K = 1000

TIEMPO (s)	BENEFICIO	CLIENTES MAL PUESTOS	CENTRALES ABIERTAS
36,48	4.635	709,6	38,3

EMPTY STATE + FIX STATE + K = 5000

TIEMPO (s)	BENEFICIO	CLIENTES MAL PUESTOS	CENTRALES ABIERTAS
106,99	48.062	467,3	35,6

EMPTY STATE + FIX STATE + K = 7500

TIEMPO (s)	BENEFICIO	CLIENTES MAL PUESTOS	CENTRALES ABIERTAS
152,4044	82.264	0	38,1

EMPTY STATE + FIX STATE + K = 6250

TIEMPO (s)	BENEFICIO	CLIENTES MAL PUESTOS	CENTRALES ABIERTAS
142,01	79.063	73,2	38,5

EMPTY STATE + FIX STATE + K = 6875

TIEMPO (s)	BENEFICIO	CLIENTES MAL PUESTOS	CENTRALES ABIERTAS
149,01	82.244	0	38,5

Conclusiones

Hemos tenido problemas con el simulated annealing y no hemos podido obtener datos válidos para estudiarlo.

En el caso del hill climbing, vemos que cuanto la k mínima está cerca de 6875, el cual es aproximadamente el coste máximo de abrir una central, lo cual tiene todo el sentido del mundo, ya que el coste de una central es lo que bloquea abrirla.

CONCLUSIONES

Las conclusiones a las que hemos llegado es que tal y como tenemos construido nuestro código, obtenemos el mejor rendimiento del resultado ejecutando el hill climbing con el estado inicial ordenadas de mayor a menor, con el heurístico de beneficios y solamente usando el operador de MoveClient. También hemos deducido que pese a que el heurístico de ganancias y el heurístico de entropía funcionan como queríamos, no hemos logrado combinarlos correctamente en el un tercer heurístico y es por eso que solamente usamos el operador MoveClient.