Inteligencia Artificial:

Práctica de Búsqueda Local

Especialidad en computación

Abella Nieto, Javier
Chen, He
Gálvez Alcántara, David

Otoño 2022-2023

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Descripción

Descripción del problema

Tenemos la tarea de asignar un conjunto de centrales generadoras de electricidad a clientes. Hay clientes, a los que llamaremos garantizados, que deben recibir forzosamente la electricidad que están demandando, otros clientes, los no garantizados, pueden o no ser servidos.

Hay que tener en cuenta que la distancia entre un cliente y la central que le sirve puede hacer que haya pérdidas de electricidad por el camino que provocan que la central haya de generar más energía de la que el cliente acabará consumiendo (y pagando) para compensar la pérdida.

También, es importante recalcar que las centrales sólo pueden o estar paradas (teniendo un coste base) o estar produciendo energía a su máxima capacidad (con un coste mucho más elevado), además los clientes solo pueden ser servidos en la totalidad de su contrato o no serlo de ninguna manera.

El objetivo del problema es encontrar la configuración que nos dé el máximo beneficio.

Vemos desde el primer momento, que es un problema en el que debido a la gran cantidad de elementos y la gran cantidad de combinaciones posibles, estados posibles en los que se podría estar, sería inabordable con los métodos más convencionales. Es conocido que con estás técnicas es probable que no lleguemos a la solución realmente óptima.

Dado el problema, nos hacemos una série de preguntas, que nos ayudan a plantear las estructuras de datos y los métodos de resolución para abordar el problema.

Preguntas de interés

1. ¿Qué elementos intervienen en el problema?

   - Tipos de centrales (A,B,C)
   - Tipos de grandes consumidores (G, MG, XG)
   - Pérdida de electricidad según la distancia recorrida

2. ¿Cuál es el espacio de soluciones

   - Una asignación de centrales con clientes que satisfaga la restricción de servicio garantizado y tenga en cuenta la pérdida de electricidad por la distancia.

3. ¿Qué tamaño tiene el espacio de búsqueda?

   - #centrales * [1] #clientes

4. ¿Qué es un estado inicial?

   - Una solución del problema que sin ser óptima cumpla las condiciones de servicio garantizado y distancia.

5. ¿Qué condiciones cumple un estado final?
   - Todos los clientes tienen una central asignada. Todas las centrales son capaces de servir como mínimo el servicio garantizado de los clientes a los que proveen de electricidad teniendo en cuenta la distancia a la que se encuentran.

6. ¿Qué operadores permiten modificar los estados y cuál es su ramificación?
   - Aunque los operadores han ido variando a través del tiempo estos son todos los que se nos ocurrieron en un principio y los que se fueron sumando a medida que íbamos implementando

     * Poner en marcha central (CE) -> O(CE)

* Apagar central (CE) -> O(CE)

* Quitar cliente(CL) -> O(CL)

* Asignar cliente (CL, CE) -> O(CE)

* Mover cliente (CL, CE) -> O(CL*CE)

* MueveBulk(vector<CL>) -> O ($2^{\#clientes}$ * #centrales)

* Swap centrales (CE1,CE2) -> O(CE1*CE2)

* Swap cliente(CL1,CE1, CL2,CE2)-> O(CL1*CE1+CL2*CE2) -> O(CL* CE)

* ApagarCentral(Central1, Central2) -> O(CL^2)

* asignarClienteNoAsignado(CL, CE) -> O(CL, CE)

* reasignarCliente(Cliente, Central) -> O(CL*CE)

7. ¿Qué estructura de datos hemos implementado para representar el estado?

   - Un vector de listas donde cada posición del vector es una central y en las listas se apunta (con un entero) a que clientes tiene que satisfacer con su demanda.

   - Una clase centrales y una de clientes que contenga información sobre estos. Dentro de un vector para almacenar todas las centrales y clientes.

   - Una clase relaciones que indique la relación establecida entre las centrales y sus clientes, además de contener información adicional que nos permite hacer cálculos de forma más rápida.

Problema búsqueda local

La respuesta al problema planteado no es encontrar los pasos que nos llevarían a una solución, sino que es tratar de optimizar los beneficios, tratar de encontrar la mejor solución posible al problema, sin importarnos qué pasos hemos tomado, o en qué orden se han hecho las diferentes asignaciones que representan la configuración final.

Además, podemos obtener una solución válida de forma rápida, como trataremos en un apartado posterior y podemos crear una función que cuyo valor esté ligado con la calidad de la solución.

Es por ello, que en este caso usaremos búsqueda local, en el espacio de soluciones, para tratar de obtener una solución lo más cercana a la óptima que podamos.

Representación del problema

La representación del problema ha sufrido varias modificaciones a lo largo del proyecto. La primera configuración guardaba una serie de clientes y centrales y las relaciones entre ellos, en esas mismas clases. Cosa que era impracticable en el momento de generar los sucesores debido a la gran cantidad de información que se debía copiar. Más adelante, esas relaciones entre clientes y centrales se representaron en una clase externa. Este modelo, pese a ser funcional, era quizás aún excesivo en memoria. El tercer y último modelo, que se explica a continuación es una simplificación de este último, que nos permite evitar hacer recorridos gracias a que guardamos cierta información y ocupa menos memoria.

Hemos creado una clase Cliente y otra clase Central. La clase Cliente guarda la información del cliente, y contiene algunas funciones que son útiles y se usan en otras clases. En la clase cliente no hay ninguna información sobre si este tiene una central asignada o cúal tiene en específico. Análogamente la clase central contiene información sobre cada central y funciones para poder acceder a sus atributos.

Cada Cliente y Central tiene asignado un identificador (un entero) que nos sirve para poder referirnos a ellos durante la ejecución. El espacio extra de este identificador es pequeño y hace que podamos hacer accesos inmediatos a los Clientes y Centrales como se explica a continuación.

Para facilitar la programación, hemos creado dos clases Clientes y Centrales, que son los contenedores donde guardamos los Clientes y las Centrales. En concreto, los Clientes se guardan en un map<Integer,Cliente> y las Centrales en un map<Integer,Central>. Esto se ha decidido así porque el espacio adicional en memoria es relativamente pequeño comparado con la inmensidad que se puede llegar a reservar durante el algoritmo de búsqueda y facilita el trabajo de programación así como los cambios que se deban hacer en el código.

Las relaciones entre centrales y clientes se representan en una clase llamada Relaciones que contiene un ArrayList<Integer> que nos dice para cada cliente (cada posición del arrayList indexado según el id del cliente) que central tiene asignada. Además, para acelerar la función heurística, la cual se puede ejecutar potencialmente cientos o miles de veces por cada iteración del algoritmo de búsqueda, se mantienen dos variables, brutoTotal y costeTotal, que sirven para calcular los beneficios además de un ArrayList<double> con los MW Usados por cada central, útil para poder decidir en tiempo constante si una central

puede aceptar un cliente o no. Estas variables se actualizan convenientemente cada vez que hacemos o deshacemos una asignación de un cliente a una central.

Con esta representación del estado, al no tener representada las relaciones entre las instancias en las clases Clientes y Centrales, no hemos de copiar estas instancias cada vez que generamos un estado nuevo. En cambio, únicamente hemos de crear una nueva clase Relaciones, ahorrando espacio.

Análisis espacio de búsqueda

El espacio de búsqueda es de tamaño #clientes * #centrales, aunque como estamos buscando dentro del espacio de soluciones, realmente el tamaño de búsqueda será algo inferior ya que es seguro que no exploramos aquellos estados que no tienen una configuración válida.

Descripción operadores

Tenemos diversos operadores que han ido cambiando a lo largo del desarrollo y se han descartado con la experimentación. Para el cálculo del factor de ramificación, usaremos c, para delimitar el número de clientes y p, para el número de centrales. En general, aplicaremos los operadores comprobando antes no solo que su aplicación no nos haga pasar a una solución no válida, sino también, esto último por una cuestión de eficiencia, que tenga cierto sentido aplicarlo, pues comprobar esto, es mucho menos costoso que generar un nuevo estado, aplicar el operador y calcular la heurística.

La condición de aplicabilidad de todos los operadores que hemos creado es que no salgamos del espacio de soluciones, es decir, que el estado que queda después de aplicar el operador siga siendo una solución válida. Además de esta condición general, cada operador puede tener otras condiciones de aplicabilidad propias.

Los operadores que hemos planteado son los siguientes:

- asignarCliente(Cliente, Central)
    - Asigna el cliente a la central.
    - O(c*p)
    - La condición de aplicabilidad es que la central a asignar el cliente tenga capacidad disponible.
    - Si el cliente ya estaba asignado a una central se le desasigna.
    - Se puede derivar las operadores asignarClienteNoAsignado(Cliente, Central) y reasignarCliente(Cliente, Central). Estos dos operadores solo se han usado en Simulated Annealing. Sus implementaciones están en la clase de la función

sucesora de SA. Básicamente se pasa un cliente con las condiciones requeridas y se aplica el operador asignarCliente.

- asignarClienteNoAsignado(Cliente, Central)
    - Asigna un cliente sin central a una central dada.
    - O(c*p)
    - La condición de aplicabilidad es que el cliente no está asignado.

- reasignarCliente(Cliente, Central)
    - Asigna un cliente con central a una central dada.
    - O(c*p)
    - La condición de aplicabilidad es que el cliente está asignado.

- quitarCliente(Cliente)
    - Quita un cliente de una central
    - O(c*p)
    - La condición de aplicabilidad es que el cliente esté asignado a una central

- swap(Central1, Central2)
    - Intercambia los clientes de dos centrales.
    - $O(p^2)$

- swapCliente(Cliente1, Cliente2)
    - Intercambia la central de dos clientes
    - $O(c^2)$

- ApagarCentral(Central1, Central2)
    - Desasigna los clientes de la central 1 y asigna los garantizados a la central 2, los no garantizados pasan a estar sin asignar.
    - $O(c^2)$

Se pensaron otros operadores a lo largo del desarrollo del proyecto, pero finalmente se descartaron (incluso antes de hacer las pruebas) por ser impracticables o poco prácticos. Por ejemplo, se planteó un operador llamado mueveBulk, con factor de ramificación $O(2^c*p)$ que permitiría mover un subconjunto de clientes a una central en específico, quizás después de detectar clientes que podían ser asignados de forma muy eficiente (por cercanía) a una central en concreto.

La condición de aplicabilidad de los operadores es que nos lleven a una solución que sea válida, es decir, que no quitemos a ningún cliente garantizado o que un cliente garantizado deje de ser servido por falta de capacidad de la central que le sirve. Esto, en casos particulares, quiere decir que hay algunos operadores que solo se pueden aplicar sobre un tipo específico de clientes, por ejemplo, el operador quitarCliente(Cliente), solo se puede aplicar sobre aquellos que son no garantizados.

Cabe destacar que únicamente con los operadores de asignar cliente y quitar cliente, podemos recorrer todo el espacio de soluciones. Al añadir operadores extra, lo hemos hecho con el objetivo de tratar de darle más “facilidad de movimiento” al algoritmo por el espacio de soluciones de manera que encontremos mejores soluciones.

Dependiendo de la heurística, puede ser que operaciones como quitarCliente no tengan sentido, pues en general queremos maximizar los beneficios y quitar un cliente nos aleja de ello, aunque evidentemente en algunos casos, para llegar a una solución óptima hayamos de pasar por antes quitar un cliente. Es por ello que probaremos varias heurísticas para ver cuales dan mejores resultados.

Vemos que puede haber un solapamiento entre los operadores de asignarClienteNoAsignado y reasignarCliente con el de AsignarCliente. Esto se ha hecho así pues aunque parece que su aplicación podría tener los mismos efectos, preferimos confirmarlo en la parte de experimentación, especialmente para el algoritmo Simulated Annealing.

Análisis función heurística

Nuestra heurística actual es el resultado de muchos cambios desde la idea inicial que tuvimos.

En un principio probamos en hacer una heurística bastante simple teniendo en cuenta sólo los beneficios, aunque pronto nos dimos cuenta de que no era la solución adecuada ya que producía unos beneficios bastante pobres, debido a que la búsqueda básicamente asignaba clientes hasta no tener espacio en ninguna central, sin llegar a mover un cliente de una central a otra en ningún momento para tratar de minimizar las pérdidas por distancia y acabar sirviendo a más clientes.

Nuestro primer enfoque fue restar los mw perdidos a los beneficios. Esto resultó ser mejor que el caso anterior, pero parecía no funcionar tan bien como habíamos esperado. Sí que se usaban otros operadores que no fuese el asignar cliente, pero muy al final de la ejecución y de manera escasa.

Para solucionar lo anterior, ponderamos los Mw perdidos en el heurístico. Tras varias pruebas, acabamos decidiendo que un mw perdido equivaliera a 300€, y es que ese es aproximadamente el precio medio de venta de un mw (aunque esto último depende evidentemente de la distribución de los clientes). Esto fue realmente efectivo pues el algoritmo de búsqueda pasó a utilizar muchos más operadores que el de asignar cliente y llegamos a tener mejores beneficios. También se planteó el penalizar por los clientes no garantizados no asignados, aunque se descartó rápidamente pues en parte ya se está penalizando al no ser sumados en el beneficio y además, es posible que la solución óptima tenga clientes sin asignar ya que no compensa asignarlos debido a las indemnizaciones.

También hemos implementado una heurística basado en la fórmula de la entropía vista en clase de teoría:

$$p * In(p)$$

Donde $p = \frac{beneficio\ neto}{MwTotal*600}$. Es decir, dividimos el beneficio neto total entre el total de Mw producido por las centrales multiplicado por el precio máximo que puede vender un Mw de energía (600 euros/Mw).

Descripción solución inicial

Para iniciar la búsqueda, hemos implementado diversas funciones para calcular una solución inicial que debido a la complejidad del problema, es subóptima. Esta solución inicial, debe ser una configuración válida del problema. Distinguiremos entre las diferentes maneras de obtener una solución inicial por qué tan *greedy* son. De hecho, en función de los operadores y el algoritmo de búsqueda, quizás nos interese una solución menos *greedy* y probablemente peor, pero que nos permita movernos más fácilmente por el espacio solución y que el algoritmo de búsqueda no pare en un máximo local rápidamente.

Hemos implementado varios algoritmos para generar el estado inicial según diversas ideas, de manera que podamos comparar entre ellos y ver cuales producen los mejores resultados, algunos de ellos han sido descartados después de la experimentación.

En todas las funciones, empezamos siempre asignando a los clientes garantizados, pues es solamente si están todos estos asignados (y servidos) cuando nuestra solución es válida. Es entonces, cuando los clientes no garantizados se asignan o no en función de un parámetro de entrada a la función. En general, hemos decidido que era mejor asignar solamente los

garantizados y dejar al algoritmo de búsqueda asignar los garantizados. Pues además deja hueco en las centrales para poder mover clientes y hacer swaps entre ellas.

- **Función 1:**
    - Asignamos clientes a centrales de manera aleatoria, primero los garantizados y en caso de que sobre espacio y dependiendo de si un parámetro de entrada de la función así lo indica los no garantizados. Debido a que generalmente habrán muchos más clientes que centrales, es altamente probable que todas las centrales queden con algún cliente asignado y por consiguiente encendidas.

    - Ya que se usan todas las centrales en la asignación, el coste de la solución inicial es generalmente el máximo (todas las centrales encendidas y por tanto produciendo al 100% siempre).

    - El coste es lineal con el número de clientes. No tenemos ninguna garantía de la calidad de la solución inicial, más allá de que será válida.

- **Función 2:**
    - Distribuimos los clientes de manera que minimicemos la distancia entre los clientes y las centrales. El algoritmo de minimización que hemos implementado es greedy, y generarla la configuración mínima de distancias es un problema complejo, así que realmente pueden existir (y existen en la mayoría de los casos) configuraciones en la que la suma de distancias entre cada central y sus clientes pueda ser más pequeña. Pero el cálculo de la óptima sería más costoso de hacer y no debería necesariamente producir mejores resultados.

    - El coste es como mucho el número de clientes por el número de centrales, pues por cada cliente hemos de recorrer todas las centrales buscando la más cercana que tenga capacidad para el cliente.

    - Este algoritmo al ser greedy tampoco nos garantiza que la solución inicial tenga la combinación óptima de distancias, aunque sí que

debería dejarnos en una mejor configuración para iniciar la búsqueda que otros algoritmos.

- **Función 3:**
    - Distribuimos a los clientes de manera que minimicemos el número de centrales encendidas. Análogamente a las funciones anteriores, el algoritmo que genera la solución inicial será greedy, pues realmente encontrar la configuración que tenga el menor número de centrales encendidas sería un algoritmo más costoso de lo que queremos hacer.

    - El coste es lineal con el número de clientes + número de centrales * log(número de centrales). En general, vamos llenando las centrales por capacidad (es por eso que en el coste hemos de tener en cuenta que las hemos de ordenar al principio de la ejecución) y vamos recorriendo los clientes hasta que la central seleccionada se queda sin capacidad, momento en el que seleccionamos otra. Este algoritmo tampoco nos da garantías más allá de que el coste será relativamente cercano al mínimo para los clientes asignados, pese a que claro está, el beneficio no vaya a ser el óptimo.

- **Función 4:**
    - Distribuimos los clientes de manera que cada central tiene un número parecido de clientes. Este problema también es complejo, así que usaremos un algoritmo greedy que puede o no darnos la solución que buscamos o una parecida.

    - El coste es clientes + centrales. Recorremos los clientes cambiando cada vez de central para que aproximadamente cada central tenga el mismo número de clientes.

- **Función 5:**
    - Distribuimos los clientes de manera que clientes de tipo 0 van a centrales de tipo 0. Si no hay centrales de un tipo disponibles,

intentamos asignarlo en el siguiente tipo. Primero se asignan los garantizados.

- El algoritmo tiene coste centrales*clientes, y es que hemos de buscar por cada cliente que centrales de su tipo están libres para asignarle una. El algoritmo tampoco nos garantiza nada sobre la calidad de la solución inicial, más allá de que será válida.

Implementacion Simulated Annealing

Hemos cogido como solución inicial asignar todos los clientes garantizados a centrales aleatorias. Los operadores son swapCliente, quitarCliente, reasignarCliente y asignarClienteNoAsignado.

Para escoger el operador a usar en una iteración, para cada operador se le asigna un intervalo de enteros. Se suma todos los intervalos y cogemos y número aleatorio entre 0 y el resultado de la suma de todos los intervalos.

En principio, el tamaño del intervalo asignado a un operador es el número total de casos diferentes donde se puede aplicar dicho operador. Por ejemplo, si el número de cliente es 5, el tamaño del intervalo swapCliente seria 4+3+2+1 = 10. No obstante, en los experimentos, seguir las proporciones matemáticas, puede dar el caso que un operador tenga una probabilidad mucho mayor que el resto, y el algoritmo solo usa ese operador. Esto lo tendremos en cuenta a hora de hacer experimentos.

Experimentos

Experimentación

En esta sección, trataremos los diversos experimentos que hemos realizado en el desarrollo del programa para poder averiguar qué conjunto de operadores, funciones para la obtención de la solución inicial y heurística producen los mejores resultados.

Para tratar de hacer los experimentos de la manera más rigurosa posible, hemos creado un pequeño script en bash que se encarga de ejecutarlo de forma automática usando seeds previamente seleccionadas por nosotros. De esta manera, no solo se es más consistentes pues todas las ejecuciones se hacen seguidas (hay un tiempo entre estas, para dar tiempo al ordenador a vaciar la memoria ocupada por la ejecución previa y volver a un estado lo más

estable posible) si no que además es más rápido y fácil ejecutar el programa 20, 30 o 50 veces al ser automático.

Experimento 1

Encontrando el mejor conjunto de operadores

Este experimento tiene como objetivo descubrir el mejor conjunto de operadores posibles, que maximicen la función heurística. Para poder extraer conclusiones del experimento, usaremos un mismo número de centrales y clientes y elegiremos un conjunto de semillas para el generador de números aleatorios para poder probar los diferentes operadores en el mismo escenario. Variar la semilla no cambiará mucho, pues el número de centrales y clientes es fijo, pero nos ayudarán a poder realizar varias ejecuciones, para reducir la influencia de una ejecución buena o mala por casualidad y no por la calidad de los operadores.

Tendremos: 5 centrales de tipo A, 10 centrales de tipo B y 25 centrales de tipo C. 1000 clientes, con una proporción de 25% de extra grandes, 30% de muy grandes y 45% de grandes, además, el 75% de los clientes (750 de ellos), tendrán el suministro garantizado.

Para el desarrollo de los experimentos únicamente usaremos la primera función generadora del estado inicial, que asigna de forma aleatoria.

La heurística que vamos a usar es el beneficio menos los mw de electricidad perdidos debidos a la distancia entre clientes y centrales multiplicado por 300 (que consideramos es el precio medio aproximado de venta de un kw).

Los conjuntos de operadores que probaremos son:

- Conjunto 1: Operadores básicos
    - AsignarCliente
    - QuitarCliente
- Conjunto 2: Operadores básicos más intercambiar clientes:
    - AsignarCliente
    - QuitarCliente
    - SwapCliente
- Conjunto 3: Operadores básicos y swap centrales (del mismo tipo)

    - AsignarCliente
    - QuitarCliente
    - SwapCentrales
- Conjunto 4: Operadores básicos y de centrales
    - AsignarCliente
    - QuitarCliente
    - SwapCentrales
    - ApagarCentral
- Conjunto 5: Operadores básicos y swap de centrales y clientes
    - AsignarCliente
    - QuitarCliente
    - SwapCentral
    - SwapCliente

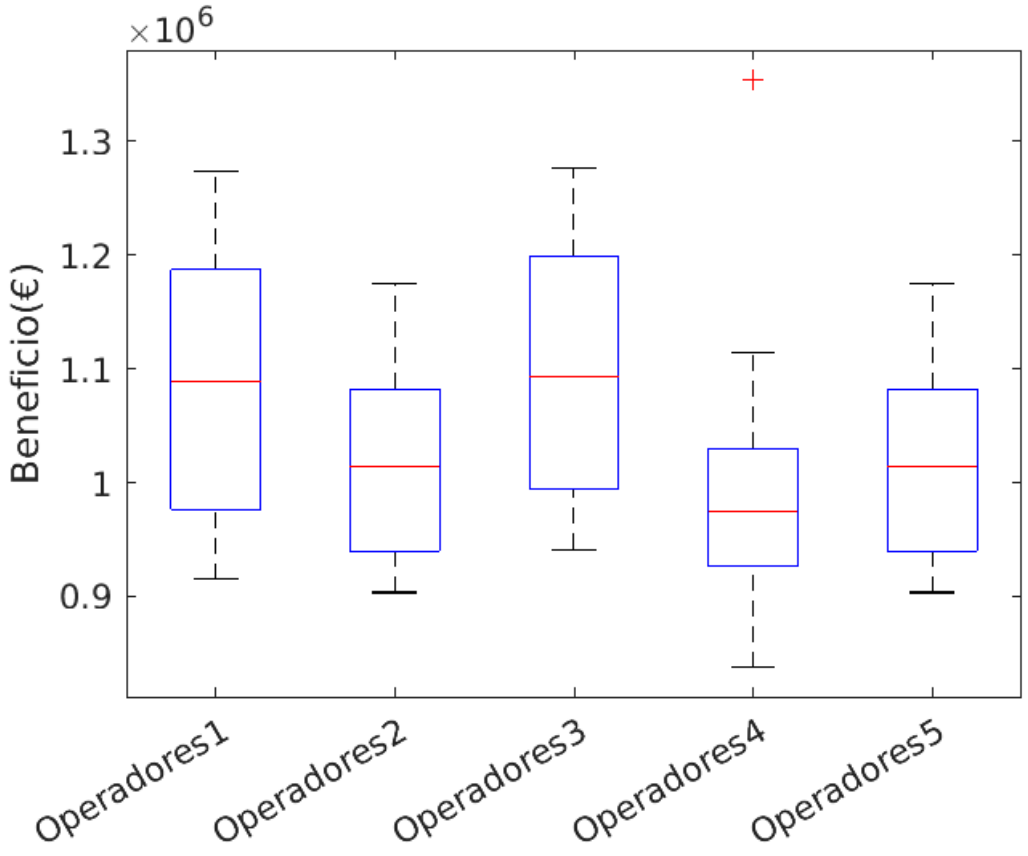
El objetivo de estos experimentos es determinar si añadir más operadores a los de asignar y quitar cliente (que podríamos considerar los más básicos), realmente nos ayuda a no estancarnos en un máximo local y por tanto encontrar mejores soluciones y en qué medida lo hace, además de si el tiempo que se añade por ellos es asumible o no.

Los resultados calculando la media, después de 30 ejecuciones por cada conjunto de operadores son:

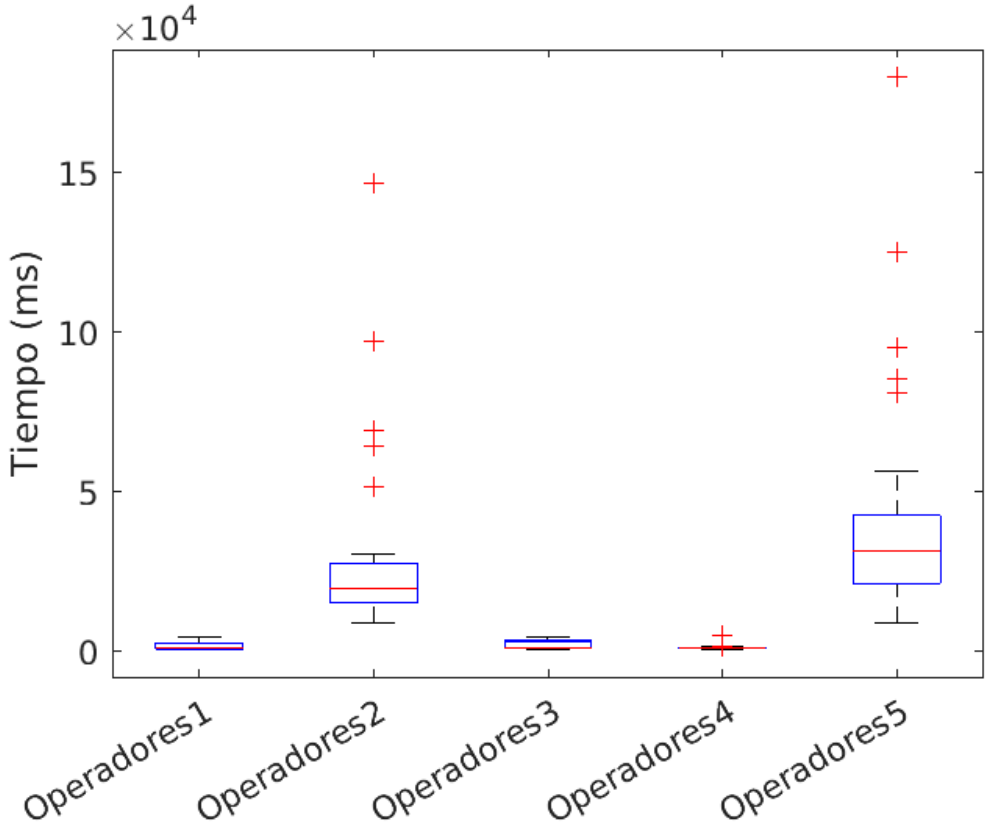
Operadores en uso	Beneficio final (€)	Incremento beneficio	σ del incremento porcentual Beneficio (%)	T_{exe} (ms)	Sucesores generados
1	1.088.410	1.43	0.19	1519	744.178

2	1.017.390	1.33	0.15	29577	14.369.957
3	1.109.244	1.45	0.18	1873	822.498
4	979.308	1.28	0.12	1096	212.093
5	1.017.455	1.33	0.15	42948	14.619.803

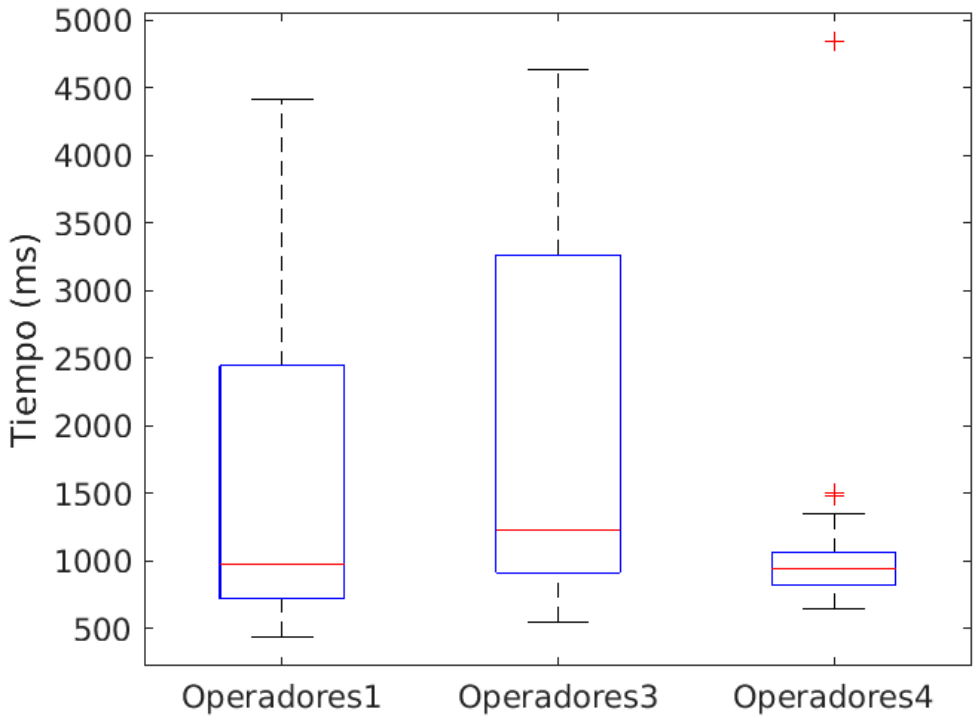
Estos son los resultados de los beneficios por cada operador en un intervalo de confianza del 95%.



Y estos los de los tiempos:



Lo mismo pero sin los operadores 2 y 5 para que la escala se adapte a las otros:



Comentarios respecto a lo que hemos observado sobre cada conjunto:

- Conjunto 1:

- Este conjunto, pese a ser muy básico, ya nos permite movernos por el espacio de soluciones. Por lo que se ve en la tabla, y aunque pueda sorprender debido a la simpleza de este conjunto de operadores, los resultados en beneficio son muy cercanos a los de otros conjuntos y con un tiempo de cálculo pequeño.

- Conjunto 2:
  - En este experimento, el alto factor de ramificación del operador de swap cliente (clientes^2) hizo que el tiempo de cálculo explotase, especialmente al probar con un número de clientes mayor al que se propone en el experimento). Para paliar ese efecto, hubo un gran *refactoring* del código, además de limitar el uso del operador a situaciones concretas.

  - El operador de swapCliente, suele usarse más al final de la ejecución que hacia el principio de esta. Esto es debido a que en la heurística que usamos, pese a que tiene en cuenta los MWperdidos por distancia, el beneficio es el factor más importante. Así que en general, añadir un cliente hace que suba más la heurística que mover un cliente a otro para reducir la energía perdida.

  - Se pueden ver al final de la ejecución largas cadenas de swapCliente, seguidas de un asignarCliente, debido a que se han ido moviendo clientes hasta que ha quedado hueco en alguna central para añadir a uno que no estaba añadido. Al principio de la ejecución, la mayoría de centrales deberían tener espacio para añadir clientes directamente.

  - El gran coste que tiene y el hecho de que obtenga los mismos beneficios que el conjunto 1, hace que lo descartemos.

- Conjunto 3:
  - La única situación en que tiene sentido un operador de swap entre centrales es cuando las dos centrales a ser intercambiadas tienen una capacidad de producción de energía similar. Si una de las centrales

puede producir mucha más energía que otra, el operador no se puede aplicar.

- En los experimentos hemos visto que en general es más probable que el operador se aplique al principio de la ejecución del algoritmo (cuando el número de nodos expandidos no es muy alto) que al final.

- Este conjunto de operadores da resultados ligeramente superiores que el conjunto 1 y tiene un coste parecido (pese a que el factor de ramificación es $O(c^2)$ limitamos el operador solo para centrales del mismo tipo así que podemos generar menos sucesores).

- Conjunto 4:
  - Este conjunto usa los operadores básicos y aquellos que modifican las centrales. Tiene el propósito de dotar a la búsqueda con más maneras de moverse por el espacio de soluciones.

  - El operador de apagarCentral, tiene como propósito detectar en qué momentos nos compensa dejar a una serie de clientes sin asignar pues pagar su compensación es mejor que tener la central encendida.

  - Como vemos por la tabla, no funciona mal, el problema está en que el operador de apagarCentral parece usarse únicamente al principio de la ejecución, cuando precisamente nos interesa usarlo en medio de la ejecución o al final, tratando de apagar una central con pocos clientes, cosa que hace que su efecto sea reducido, que la búsqueda sea más lenta y que no lleguemos a conseguir mejores beneficios, además de que probablemente deberíamos refinar la heurística únicamente para este conjunto de manera que el operador se use más a lo largo de la ejecución.

- Conjunto 5:

- Este conjunto combina la gran cantidad de estados generados y por tanto el elevado coste de computación con el operador de swap con el operador que tan buenos resultados ha dado del conjunto anterior. Los tiempos de cómputo han aumentado debido a que generamos más estados.

- Debido a que estamos usando el operador de swapCliente, en general, hay menos pérdidas por energía y por tanto es menos probable que se acabe usando el operador de swapCentral, así que hay ejecuciones en las que no se usa ni una sola vez y otras en las que si se usa, es únicamente al principio de la ejecución.

- Podríamos pensar que el hecho de aplicar el operador de swapCentral en este conjunto nos ahorra ejecuciones respecto a si tuviésemos que cambiar entre clientes con el operador de swapCliente. Aunque esto no es realmente cierto ya que aplicando el swapCentral y no el swapCliente podemos llegar a otras configuraciones que conlleven aplicar más operadores después.

Descartando los conjuntos 2 y 5 debido al gran coste computacional y nula mejora respecto a otros conjuntos, queda decidir cúal escoger entre el 1,3 y 4. Descartamos el 4 porque los beneficios que genera son los más pequeños, y aunque es el que menos tiempo tarda, preferimos priorizar la ganancia. Entre el 1 y el 3, el argumento es similar, el 3 tarda ligeramente más pero se obtienen con él mejores beneficios, así que es el que escogemos.

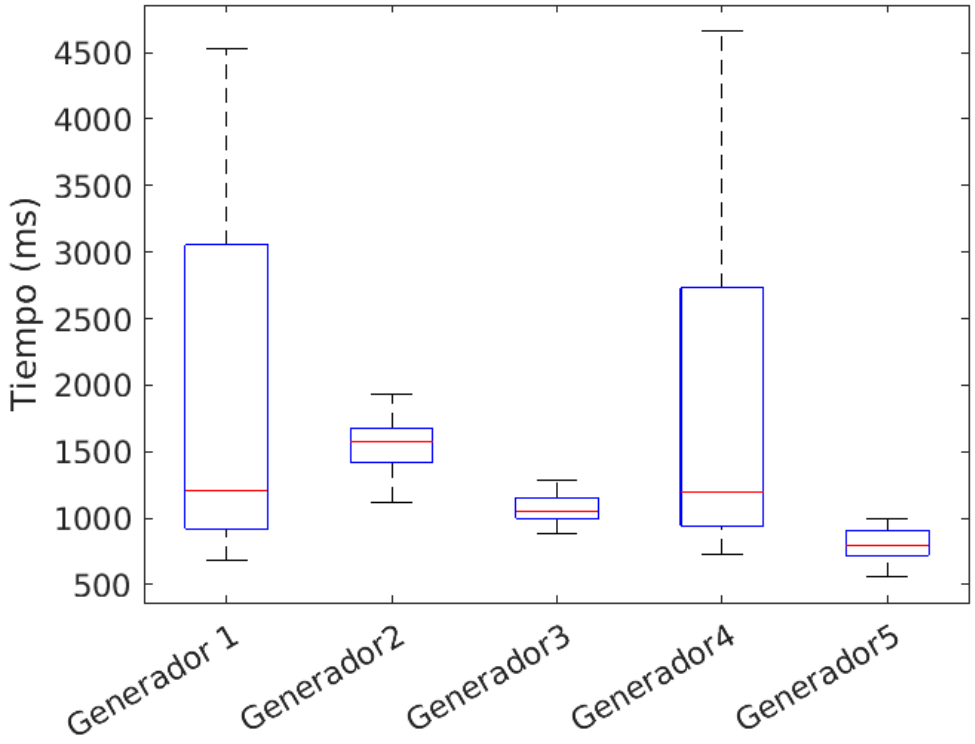
Experimento 2

Encontrando la mejor función de generación inicial

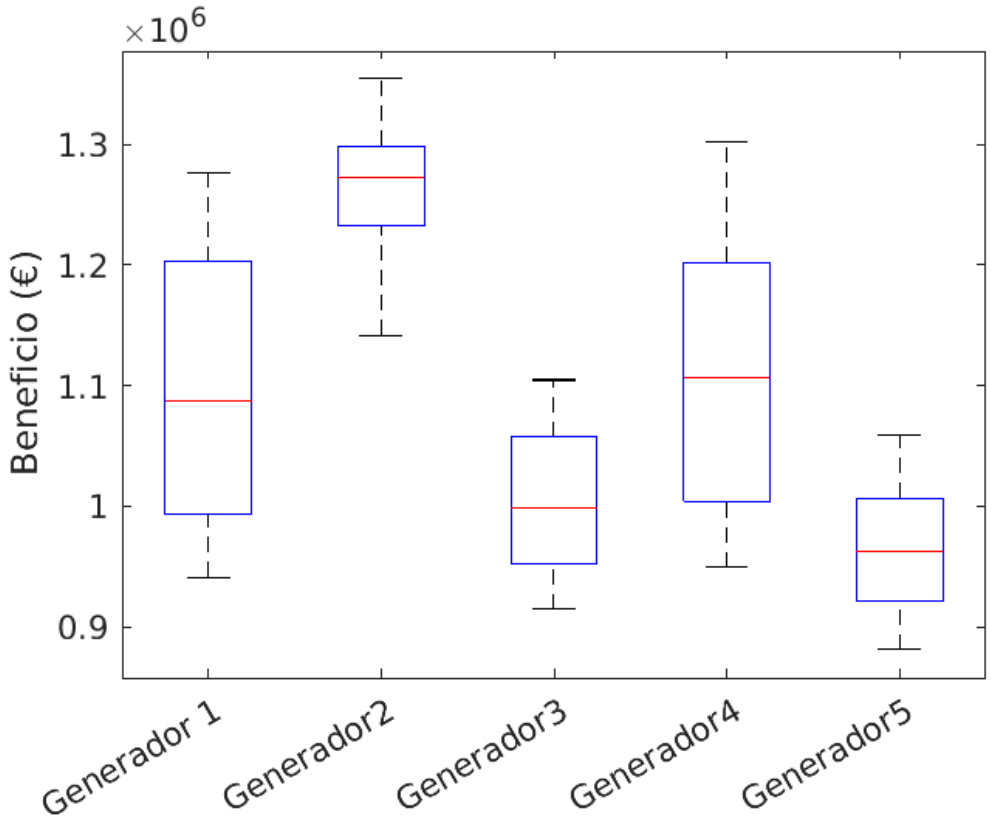
El objetivo de este experimento es encontrar la mejor función de generación del estado inicial posible para el conjunto de operadores escogido en el apartado anterior. Las funciones que ejecutaremos, están especificadas en el apartado correspondiente.

Los resultados calculando la media, después de 30 ejecuciones por cada conjunto de operadores son:

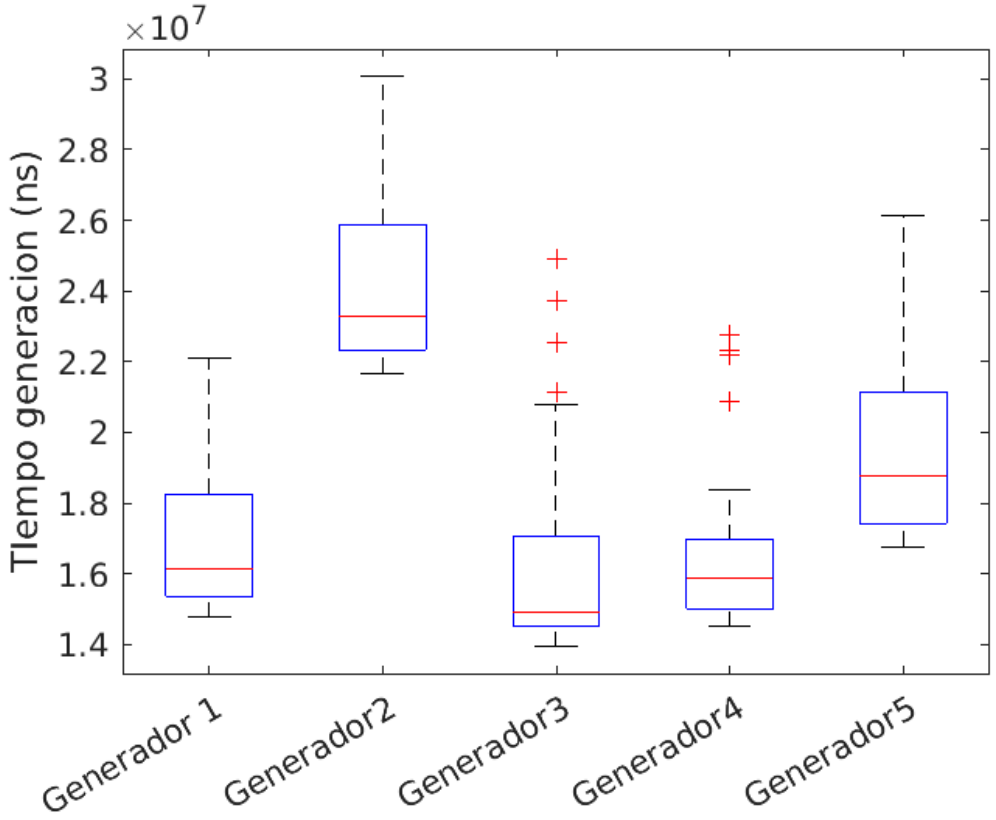
Función en uso	Beneficio Inicial (€)	Beneficio final (€)	Incremento beneficio (%)	σ del incremento porcentual Beneficio	t_{exe} (s)	Sucesores generados	t_{exe} generación inicial (ms)
1	741389	1109350	1.45	0.19	1890	822766	17.27
2	756118	1265965	1.62	0.09	1527	480007	24.12
3	926055	1005346	1.08	0.02	1070	342116	16.37
4	765379	1110253	1.45	0.18	1877	809362	16.60
5	913972	962451	1.05	0.02	803	168371	19.55



Vemos que la 1 y la 4 tienen un rango de tiempos muy amplios mientras que el resto se mueven entre valores más cercanos.



Vemos que la función de generación 2 está claramente por encima de las otras, saliendo del intervalo del 95% para dos de ellas y teniendo la media por encima de todas ellas.



Vemos que respecto al tiempo de generación, la función 2 es la que más tarda, seguida de la 5. También vemos que la 1, 3 y 4 están muy parejas.

Comentarios respecto a lo que hemos observado sobre cada función:

- Función 1: Tal y como esperábamos, esta función no añade un excesivo tiempo de cálculo al programa. Además, parece dejarnos siempre (pese al hecho de que es aleatorio) en una posición que nos permite aplicar operadores para llegar hasta el máximo local en el que pararemos.

- Función 2: Sorprendentemente, este algoritmo de generación de una solución inicial genera en muchos casos una solución con un beneficio inicial similar al de los otros acercamientos, pero con una configuración mucho más óptima, que hace que tardemos menos en acabar la ejecución y con un beneficio mejor o igual al de los otros acercamientos. Tal y como esperábamos por el coste extra de generar esta solución (numeroDeClientes*numeroDeCentrales), es la que más tiempo tarda en media, pero parece compensar.

- Función 3: Esta función, como se ve en los experimentos, no funciona dada nuestra heurística. En este caso, la búsqueda nunca enciende una central, y es que al poder

asignarle solamente un cliente de golpe, la heurística baja (el beneficio cae en picado pues hemos de pagar toda la producción de energía cuando solo tenemos un cliente). El programa se dedica a mover clientes de una central a otra o asignar clientes a alguna de las centrales encendidas y acaba rápidamente. No funciona bien.

- Función 4: Esta función tampoco parece funcionar bien, de hecho, tiene unos resultados muy similares a la función random, como se puede ver tanto en la tabla como en los gráficos, siendo en todos los aspectos muy parecida.

- Función 5: Esta función es la que genera soluciones iniciales más cercanas a la solución final del algoritmo, como se puede ver por ser la más rápida. El problema es que el final del algoritmo es un máximo local, muy lejos de los beneficios que quisiéramos obtener. Parecería que estuviésemos siendo excesivamente greedy. Este comportamiento no iba alineado con lo que estábamos esperando antes de la ejecución de los experimentos. Esta función parece ser especialmente buena si queremos encontrar una solución buena rápido, pero si queremos encontrar las mejores soluciones que nos puede proporcionar el algoritmo de búsqueda, parecen mejores otros acercamientos.

Por los bajos beneficios, las funciones 3, 4 y 5 quedan descartadas. Entre las funciones 1 y 2, vemos que la 2 genera soluciones sustancialmente mejores y pese a que la generación de la solución inicial es más lenta en el caso de la segunda, el tiempo de ejecución total es mejor, pues estamos más cerca del final del algoritmo de búsqueda.

En conclusión, escogemos la función de generación 2.

Experimento 3: Simulated Annealing

Encontrando la mejor configuración de SA

Este experimento tiene como objetivo encontrar los parámetros más adecuados para el algoritmo Simulated Annealing. Los parámetros a experimentar son:

- Número total de iteraciones (steps)

- Iteraciones por cada cambio de temperatura (ha de ser un divisor del anterior) (stiter)

- Parámetro k de la función de aceptación de estados

- Parámetro lambda de la función de aceptación de estados

Se irá probando con diferentes valores de estos parámetros para determinar qué combinación da mejor resultado.

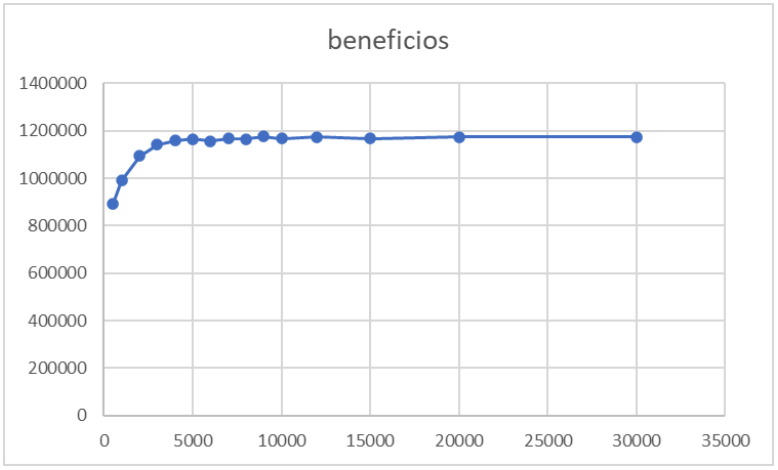
Después de seleccionar una instancia particular del problema (seed = 1123), hemos cogido como función de estado inicial la que asigna aleatoriamente los clientes garantizados a centrales (no se asigna los no garantizados). La función heurística que sea usado está basado en la fórmula de entropía.

Hemos cogido los siguientes rangos para los parámetros:

- Lambda: {1, 0.1, 0.01, 0.001, 0.0001}
- K: {1, 5, 10, 25, 100, 125}
- Stiter: {1, 10, 50, 100, 200, 500, 750, 1000}

Probaremos las diferentes combinaciones posibles e iremos descartando las peores soluciones.

Primero empezaremos averiguando cuántas iteraciones son necesarias. Para ello fijamos la k a 5, la lambda a 0.01 y stiter a 50

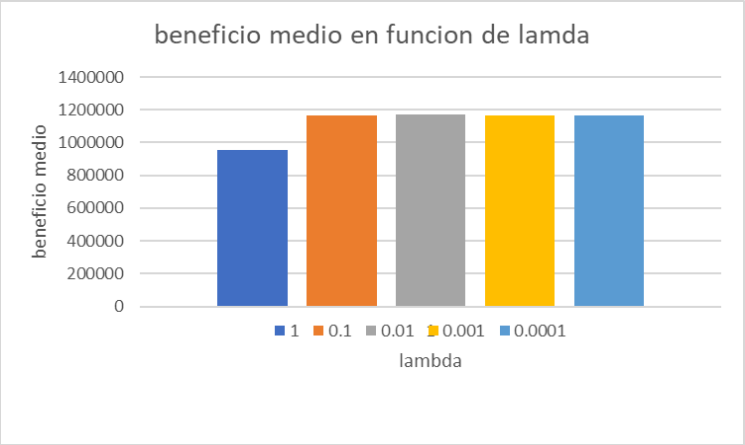


Evolución de beneficios: X: steps, y: beneficios

Vemos que el beneficio se estabiliza después de 5000 iteraciones, y con 9000 iteraciones alcanza un máximo local, por lo tanto tomamos 9000 con el número de iteraciones.

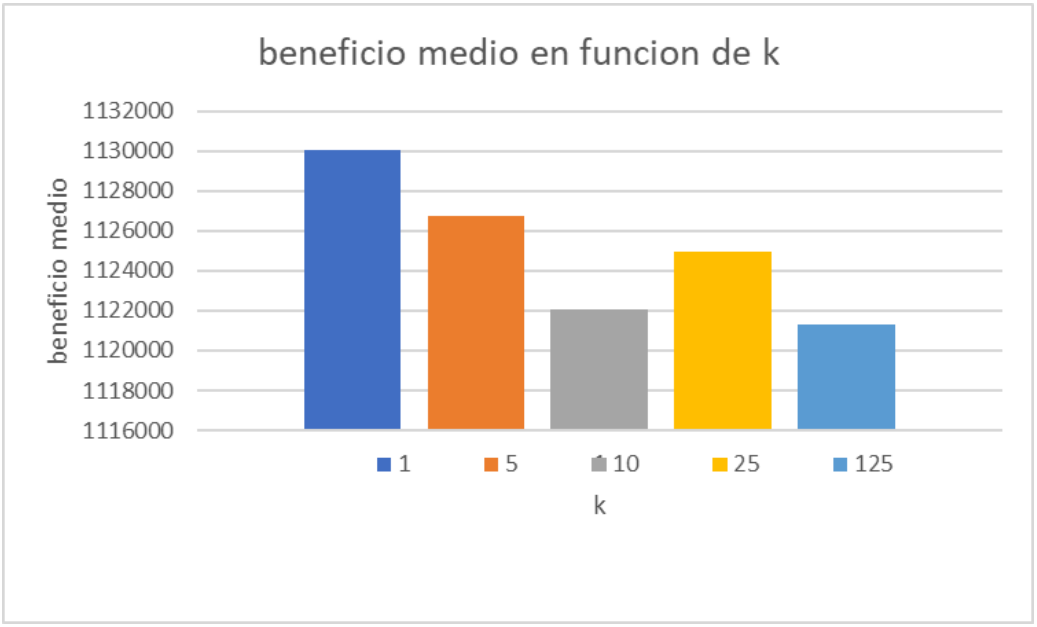
A continuación procedemos a experimentar con qué combinación de lambda y k nos quedaremos, fijando steps a 9000 y stiter a 50.

La siguiente gráfica, obtenido calculando la media de beneficios de K: {1, 5, 10, 25, 100, 125} para diferente valor de lambda:

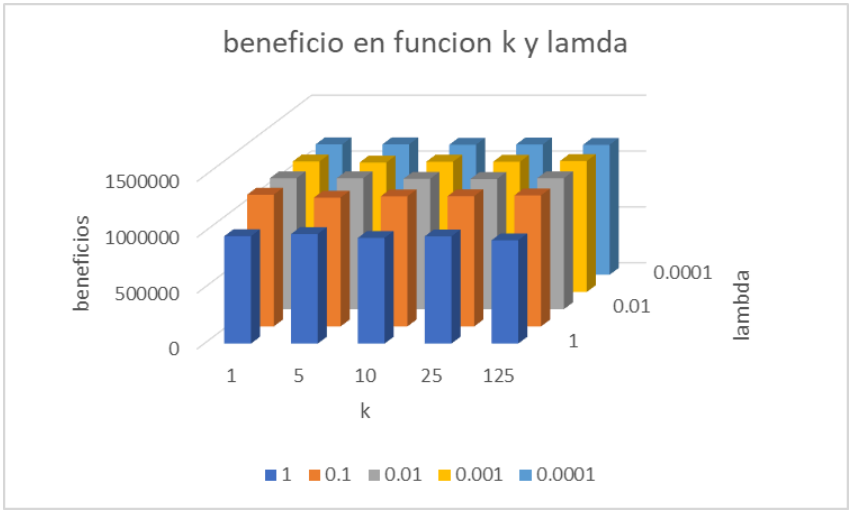


Podemos ver, no hay mucha diferencia en el beneficio medio para lambda :{0.1, 0.01, 0.001, 0.0001}. Hemos cogido 0.01 como valor de la lambda, ya que es el que tiene el valor numérico más grande.

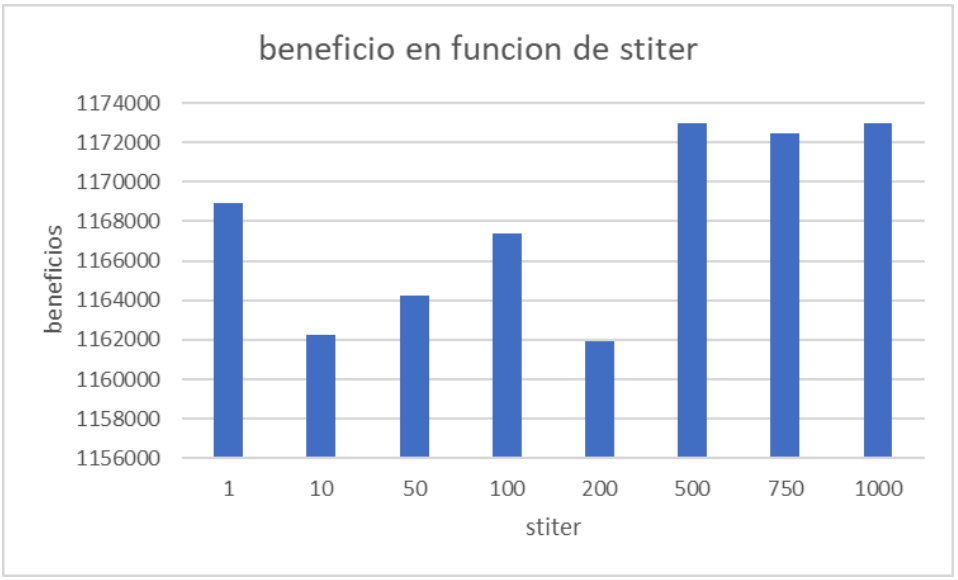
La siguiente gráfica, obtenido calculando la media de beneficios de Lambda: {1, 0.1, 0.01, 0.001, 0.0001}para diferente valor de k:



Se puede ver que el valor de k con mayor beneficio medio es k = 1.
Por lo tanto fijamos el valor de lambda a 0.01 y k a 1.
La siguiente gráfica 3D se puede ver los beneficios de todas las combinaciones posibles de k y lambda:



Finalmente, nos falta por fijar un valor para stiter. Ejecutamos el algoritmo para los valores de stiter anteriores, en la siguiente gráfica podemos ver como evoluciona (steps = 9000, k=1, lambda = 0.01):

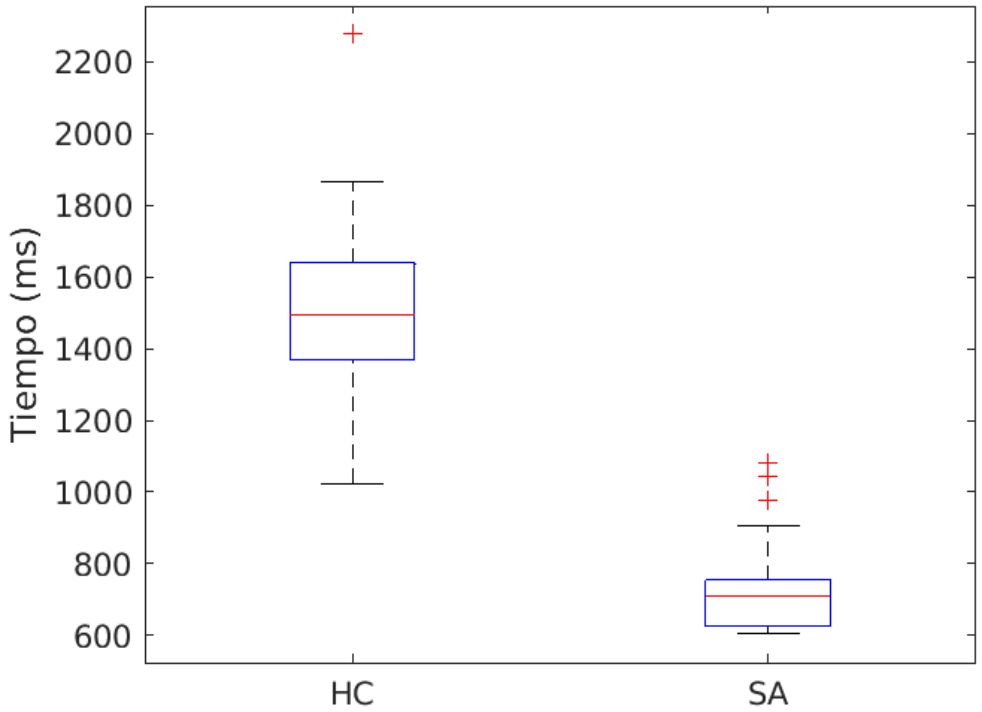


Después de stiter = 500, el valor de los beneficios ya no varía mucho, así que hemos cogido 500 como valor para stiter.
Finalmente obtenemos la siguiente configuración:

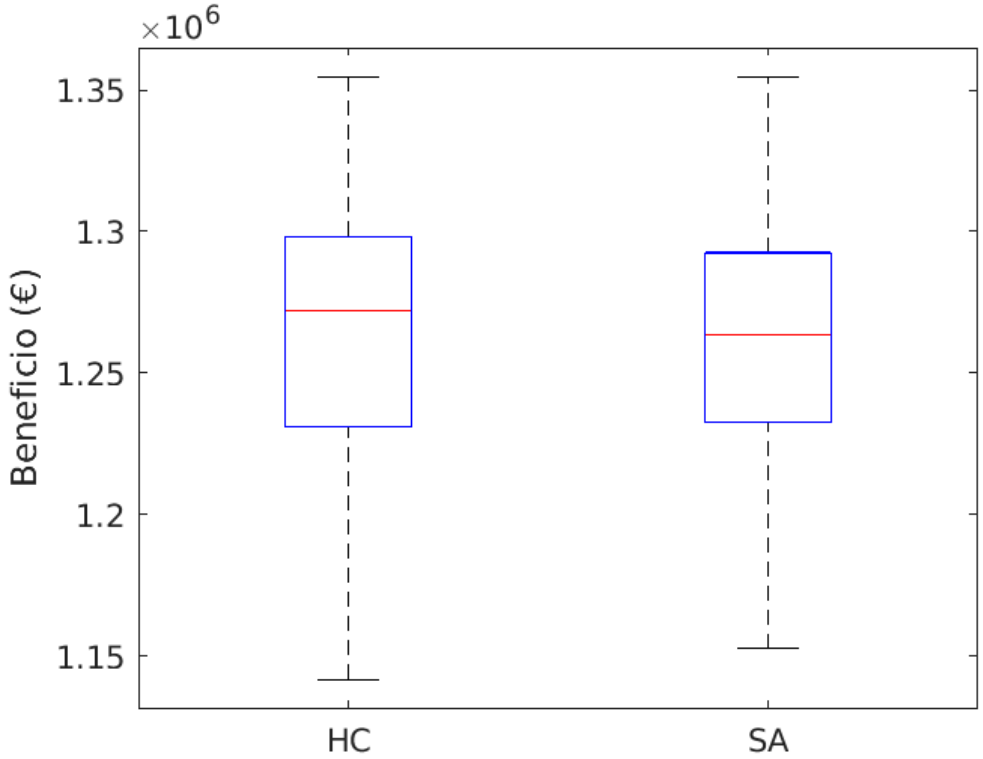
- Steps: 9000
- Stiter: 500
- Lambda: 0.01
- K: 1

Comparando Hill Climbing y Simulated annealing

Usaremos los parámetros óptimos de Simulated annealing que hemos encontrado en este apartado y los operadores y heurística escogidos para ver cuál de los dos produce mejores resultados. Probaremos con 30 seeds diferentes para que los resultados sean más fiables.



Vemos en el gráfico que claramente Simulated Annealing funciona de forma mucho más rápida que hill climbing, con la configuración óptima de simulated annealing que hemos encontrado.



Vemos que ambos nos dan aproximadamente el mismo beneficio en media.

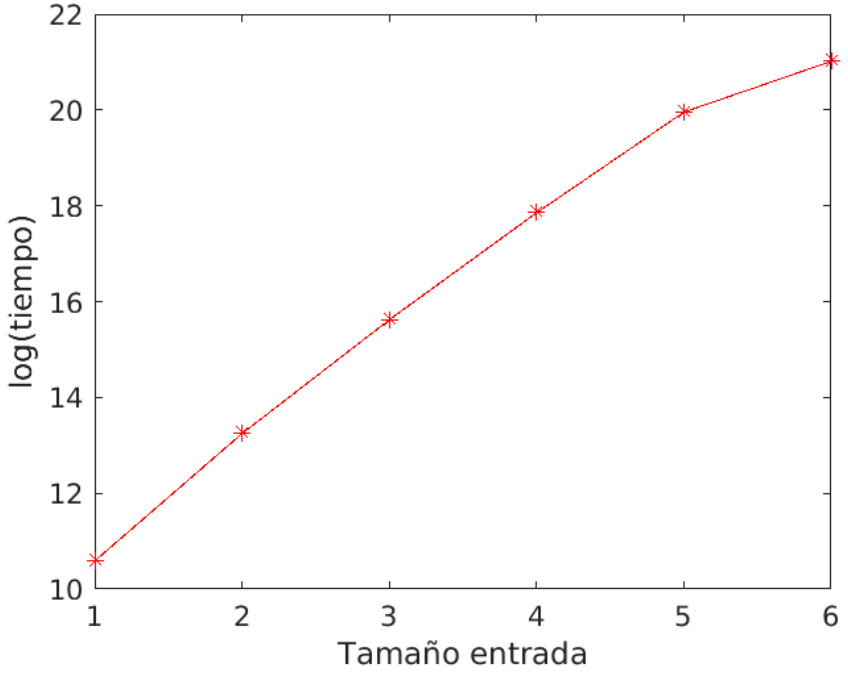
De estos resultados no podemos concluir que Hill Climbing sea mejor que Simulated Annealing, pero sí que con nuestra heurística, operadores y función de generación inicial, Hill Climbing funciona mejor.

Experimento 4

Aumentando número de centrales y clientes

Este experimento tiene como propósito encontrar qué tanto varía el tiempo de ejecución con el tamaño de la entrada con el conjunto de operadores y función de generación escogida. Sabemos que el problema propuesto es difícil de resolver y que en general, son intratables para una entrada muy grande, en este apartado veremos qué tan rápido aumenta.

Iremos aumentando el tamaño de la entrada, sumando 1000 clientes, 5 centrales A, 10 centrales B y 25 centrales C, hasta que el tiempo de cálculo sea inasumible o hasta que tengamos datos suficientes como para ver qué relación hay.
Esperaríamos que asintóticamente la función tiempo creciese según $O(c^2)$ dado el conjunto de operadores que estamos usando. Graficamos tiempo y número de sucesores. Para poder visualizar los datos mejor, tomaremos los logaritmos de los datos (en base 2) cuando sea necesario.

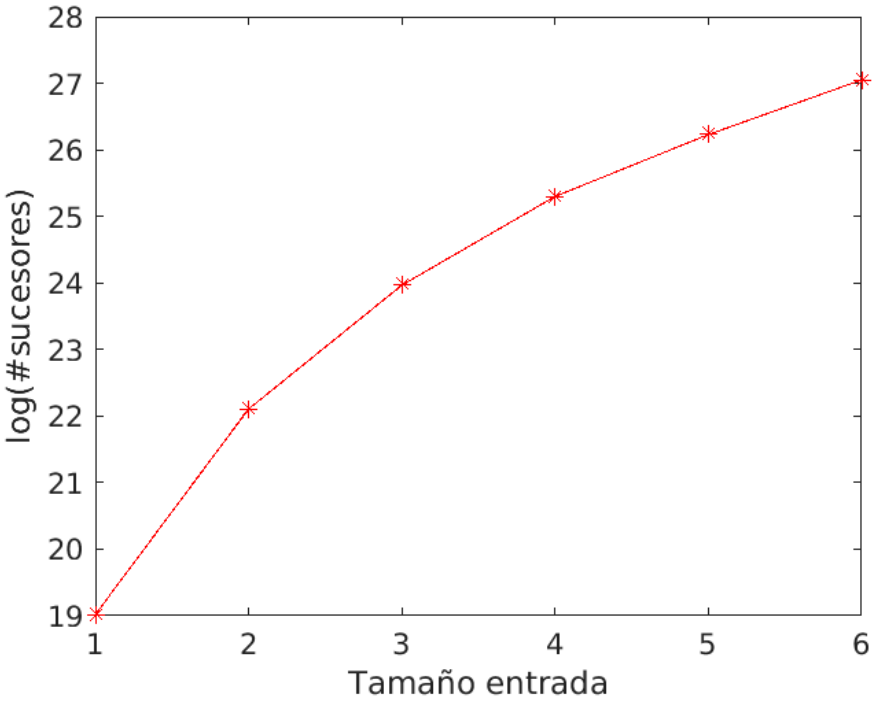


De este gráfico vemos como crece el logaritmo del tiempo de ejecución según el tamaño de la entrada. En particular vemos que el logaritmo del tiempo crece linealmente con la entrada, lo que nos quiere decir que el tiempo (sin calcular el logaritmo) crece de forma exponencial. En concreto, se puede ver en la gráfica que multiplicar el tamaño de la entrada por 2 supone multiplicar el tiempo por 16, aproximadamente (es lo que resulta de restar el

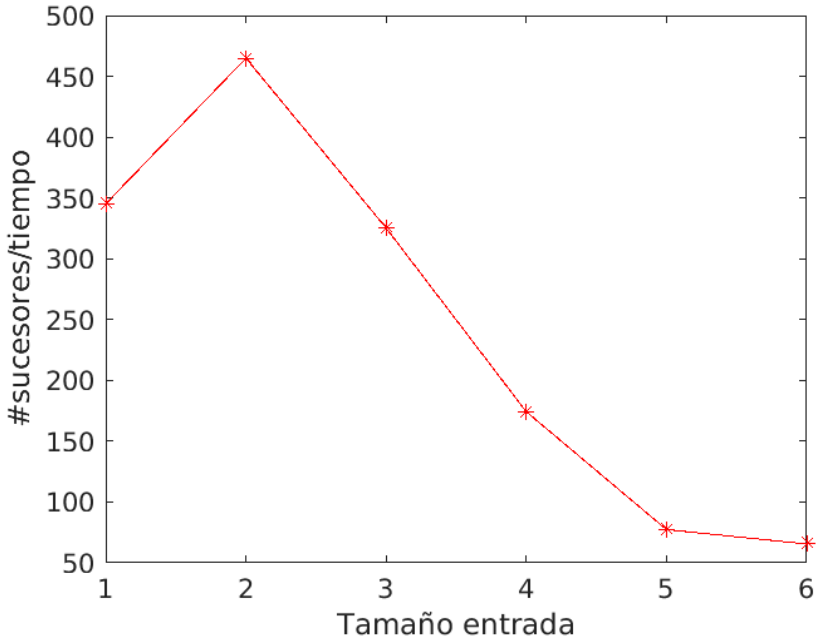
tiempo de dos ejecuciones que tienen una el doble de tamaño que la otra y elevarlo a 2 para hacer el inverso del logaritmo).

Parecería por la gráfica que el coste es proporcional a $2^{10} * 16^{log(x)} = 1024 * (2^4)^{log(x)} = 1024 * (2^{log(x)})^4 = 1024 * x^4$.

En cualquier caso, por la gráfica también podría ser que el coste fuese exponencial, según la función $4^{(5+x)} = 2^{(10+2x)}$ aproximadamente.
Viendo que para tamaño de la entrada 6 el aumento del tiempo se frena, cabe pensar que el coste es proporcional a la primera función, pero nos faltan puntos para poder saberlo del todo y cada vez es más costoso obtenerlos.



Vemos que el número de sucesores parece seguir una forma parecida a la gráfica anterior, creciendo muy rápido y frenando ligeramente conforme aumenta el tamaño de la entrada.

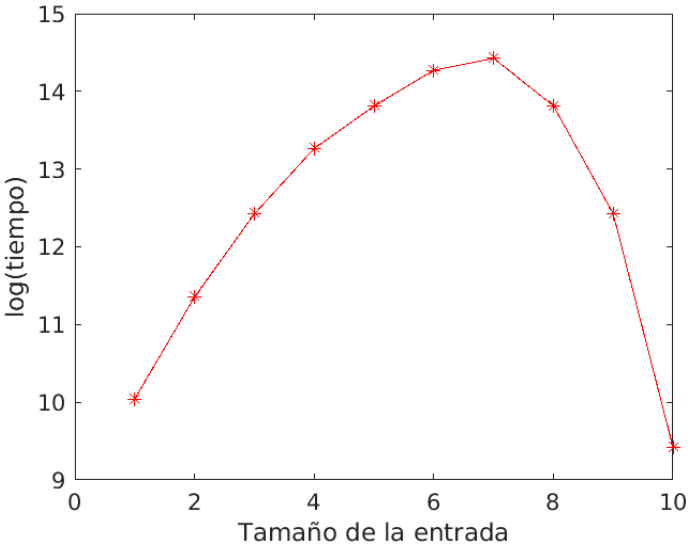


Vemos que si analizamos cuánto tardamos en crear un sucesor según el tamaño de la entrada, el tiempo que tardamos en hacerlo se reduce. Esto, aunque pueda parecer algo contradictorio con que el tiempo aumenta conforme aumenta el tamaño de la entrada, es coherente cuando se considera que el número de sucesores aumenta muy rápidamente como hemos visto en la anterior gráfica y también cuando sabemos que los sucesores más rápidos de crear son los de los clientes. La proporción entre los clientes y las centrales aumenta conforme aumenta la entrada, por lo que cada vez hay más sucesores que vienen de clientes en relación con los sucesores que vienen de operar por las centrales y por tanto el tiempo medio por sucesor se reduce.

Este análisis lo hemos hecho aumentando el número de centrales y el número de clientes a la misma vez. Para acabar de ver la tendencia y ver cómo afecta cada uno de los parámetros separamos el análisis, aumentando el número de clientes y el número de centrales por separado.

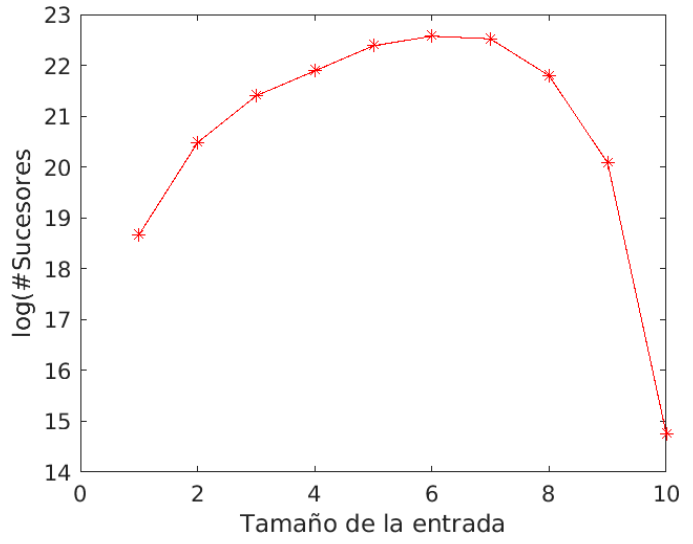
Fijando centrales y aumentando clientes

Empezaremos con 40 centrales y 500 clientes. Aumentaremos el número de clientes de 500 en 500 hasta que tengamos suficientes datos como para hacernos una idea de la función tiempo según el tamaño de la entrada. Lo que hemos hecho en el test para tener espacio suficiente para todos los clientes es usar 40 centrales de tipo A, si no, la función que genera la solución inicial no acaba nunca porque no puede asignar a todos los clientes garantizados.



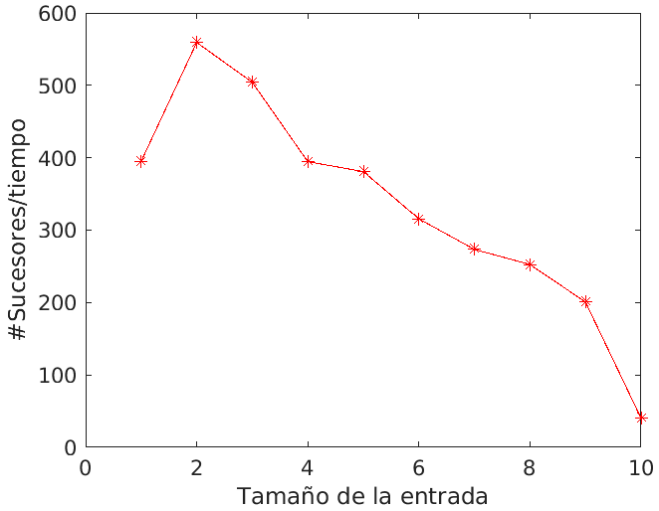
Vemos que esta gráfica es notablemente diferente a la del anterior experimento. Al principio, el tiempo de ejecución crece de forma muy rápida, probablemente de forma exponencial, pero llega un punto en el que deja de crecer y cae muy rápidamente. Esto último es debido a que para un número muy grande de clientes en relación con el número de centrales, una vez hemos creado la solución inicial, todas las centrales se encuentran muy llenas y con una distribución de clientes y distancias relativamente buena (la función de generación inicial trata de minimizarlo). El operador de swapCentral no se usa casi nunca pues las centrales están tan llenas que es poco probable que se saque algún beneficio o que incluso usarlo nos deje en una solución válida. Así que el programa únicamente expande unas pocas decenas de nodos, asignando a clientes no garantizados y cambiado a algunos clientes de una central a otra y acaba.

Queda claro que no estamos encontrando una solución óptima de esta manera. El problema se encuentra en que para solucionarlo deberíamos probablemente crear un operador que hiciese swap de clientes, operador que ya probamos en el primer experimento y que resultó en tiempos de ejecución muy muy altos.



Vemos que el número de sucesores generados es coherente con la gráfica anterior, y es que llega un punto en el que el número de nodos expandidos decrece y aunque el número de

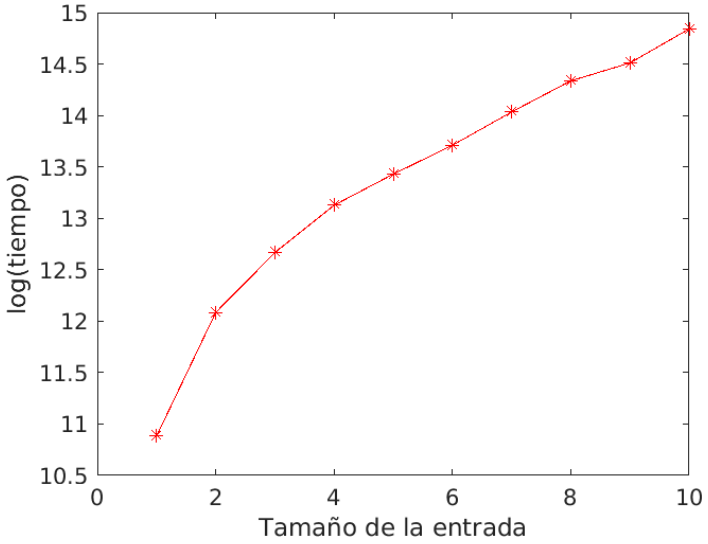
sucesores por iteración aumenta (debido a que hay más clientes sobre los que aplicar los operadore), si el número de nodos expandidos decrece más rápido, el número de sucesores acaba decreciendo también.



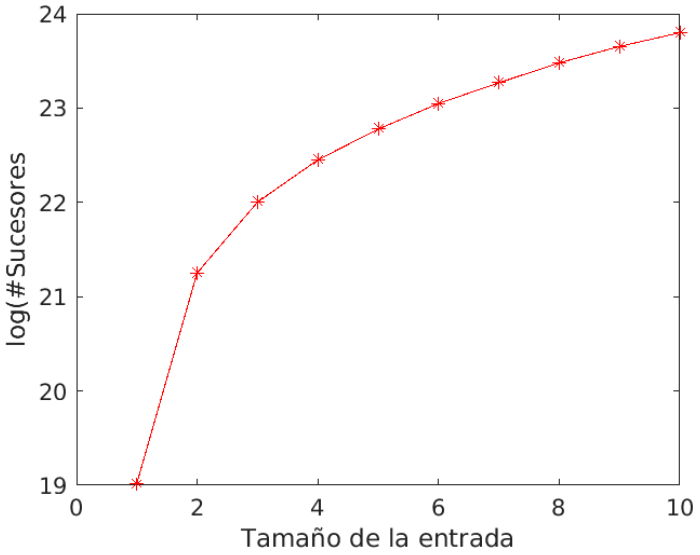
Vemos que tal y como en el experimento pasado, y por las mismas razones, cada vez tardamos en media menos tiempo en generar un sucesor.

Fijando clientes y aumentando centrales

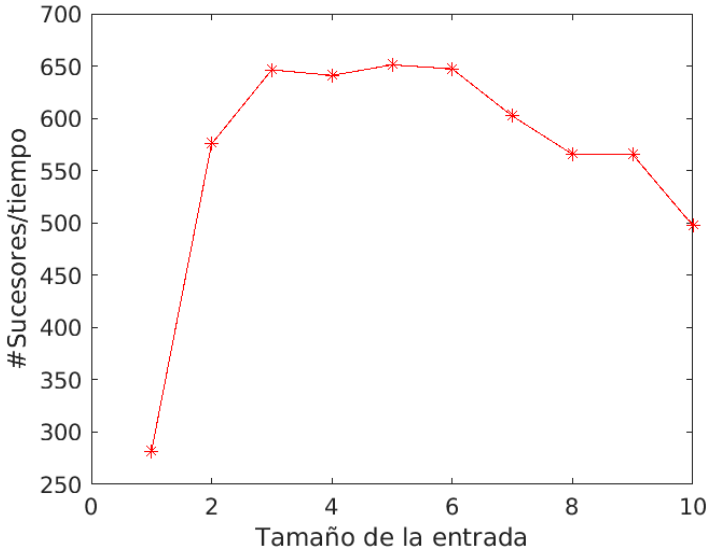
En este experimento fijaremos el número de clientes en 1000 e incrementaremos el número de centrales (de forma proporcional a 5 de tipo A, 10 de tipo B y 25 de tipo C) hasta que nos podamos hacer una idea del coste.



Vemos que el tiempo aumenta de forma rápida con la entrada aunque de forma más lenta que en el primer caso primero. En general, el trabajo que hacemos es el de asignar todos los clientes en la función de generación inicial y después asignar los clientes no garantizados restantes.



Vemos que el número de sucesores se comporta de forma similar aunque con un crecimiento aún más lento.



En general vemos que nuestra algoritmo funciona suficientemente rápido para pequeñas entradas y que fijar el número de los centrales o los clientes mientras se aumenta el otro hace que el tiempo aumente de forma más lenta. También vemos que el algoritmo parece funcionar mucho mejor (de ahí que tarde más y es que estamos llegando a algo más óptimo) cuando el número de centrales y clientes se encuentra de forma tal que hay espacio suficiente en las centrales como para asignar y desasignar clientes suficientes y hay clientes suficientes no garantizados como para tener que encontrar una configuración óptima en las centrales para asignarlos todos. Dada nuestra heurística, en general, no estamos llevando al algoritmo a que apague centrales (ya vimos en el experimento 1 que no parecía funcionar) pero en algunos de los casos parecería que falta algo más para poder encontrar una solución mejor, pues en estos últimos casos no es que acabemos pronto porque ya estamos en un óptimo sino que acabamos pronto porque dada la heurística que tenemos no podemos encontrar una solución mejor, encendida una central casi nunca la estamos apagando y nuestra función de generación inicial casi siempre enciende todas las centrales. Aunque esto

debería probarse en algún otro experimento y quizás cambiar operadores, heurística y generación inicial en función de la distribución de centrales y clientes que nos encontramos.

Experimento 5

Hill Climbing

En este experimento veremos cómo se comporta el algoritmo de búsqueda cuando en vez de comprobar que los sucesores que se generan son válidos, añadimos una penalización a la función heurística. Haremos que la función de generación del estado inicial sea vacía. La única manera que tendremos de salirnos de una solución válida será no asignando a clientes garantizados, pero mantendremos todas las otras. El único cambio que haremos de hacer en los operadores es con el operador de quitar, que ahora sí puede aplicarse sobre clientes garantizados. La función de sucesores se quedará igual a excepción de ese cambio. La heurística la hemos modificando restando el número de clientes no garantizados de manera que aplicamos una penalización a las no soluciones (para poder obtener esto de forma eficiente hicimos un pequeño cambio añadiendo una variable a la representación) y restando el número de centrales sin clientes.

Esto último es necesario pues partiendo de que las centrales están apagadas, dejando la heurística del beneficio es muy difícil que lleguemos a encender una central, y si se enciende son solamente las más pequeñas, es por ello que ese factor, que ponderamos, tiene como objetivo hacer que se enciendan más centrales. Otra solución sería aumentar el factor que penaliza a los clientes garantizados hasta que veamos que todos son asignados porque se pueden encender todas las centrales. Simulated annealing no sufre tanto este problema.

Si nuestra heurística es de la forma:
beneficios - totalDesperdiciado*300 - garantizadosNoAsignados*k - centralesApagadas*r, hemos de garantizar que todos los clientes garantizados quedarán asignados, que siempre salga a cuenta añadir un cliente no garantizado, incluso si para ello hemos de encender una central que estaba apagada.
Cuando asignamos al cliente, estamos sumando a la heurística (dejando de restar) k+r, pero restando de los beneficios el coste de encender a la central menos el beneficio por el cliente y sumando al totalDesperdiciado las potenciales pérdidas de energía.

Por ello, para garantizar que la heurística siempre aumenta al asignar un cliente no asignado, k + r + venta cliente ≥ perdida_energia*300 + coste_encender_Central.
El coste máximo de encender una central es, si hemos de encender una de tipo (A), 750*50+(20000-15000) = 42.500€.
Asumiendo que venta_cliente ≥ perdida_energia, nos queda que: K+r ≥ 42500€

Dado que en general tendremos muchos más clientes que centrales y dado que no solo hemos de asignar los garantizado sino también los no garantizados, para que la heurística quede equilibrada, es interesante que r≥k. Hemos hecho las siguientes pruebas:

k	r	Beneficio medio	Tiempo
0	42500	1145561.9	19.608
8500	34000	990383.4	18.492
17000	25500	759658.9	12.199
25500	17000	728770.9	9.05
34000	8500	718129.4	7.512
42500	0	718129.4	8.138

Vemos que el óptimo se debe encontrar entre k = 0, r = 42500 y k = 8500 y r = 34000. Repetiremos los experimentos centrándonos en esa zona:

k	r	Beneficio medio (€)	Tiempo (s)
0	42500	1145561.9	19.608
1700	41800	1145561.9	19.550
3400	39100	1171740.7	19.431
5100	37400	1031421.7	19.194
6800	35700	1013454.2	18.791
8500	34000	990383.4	18.492

Por lo tanto, vemos que los parámetros ideales son aproximadamente k = 3400 y r = 39100 pues son los que arrojan mejores beneficios. Podemos encontrar también otras configuraciones que acaban sustancialmente antes pero que obtienen menos beneficios.

El problema con los tiempos en este experimento es que pese a que hemos cambiado la representación del problema para poder hacer los cálculos de la nueva heurística en tiempo constante, dado que el estado inicial es vacío, hemos de dar muchos más pasos para encontrar la solución inicial, lo que deriva en que acabamos generando muchos más sucesores, que es donde se acaba yendo el coste.

Simulated Annealing

k	r	Beneficio medio	Tiempo (s)
0	42500	1075202.90	1.075
8500	34000	972478.40	1.122
17000	25500	928760.40	1.159
25500	17000	954056.90	1.549
34000	8500	946592.90	1.269
42500	0	965580.40	1.218

Igual que Hill Climbing, el óptimo se encuentra entre k =0, r = 42500 y k = 8500, r = 34000. Repetiremos los experimentos centrándonos en esa zona:

k	r	Beneficio medio (€)	Tiempo (s)
0	42500	1075202.90	1.075
1700	41800	975272.40	1.385
3400	39100	941145.39	1.210
5100	37400	954972.89	1.090
6800	35700	945724.39	1.193
8500	34000	968474.39	1.187

Vemos que para k = 0, r = 42500 obtenemos mejor beneficio.

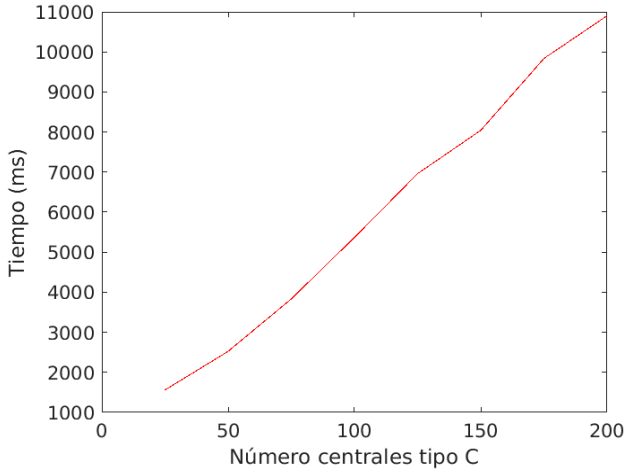
A diferencia del Hill Climbing, aquí las iteraciones ejecutadas están determinadas(9000) y para cada iteración aplica una operación, por eso, observamos que Simulated Annealing es mucho más eficiente.
Otro factor importante que observamos es que los beneficios de Hill Climbing son más elevados.

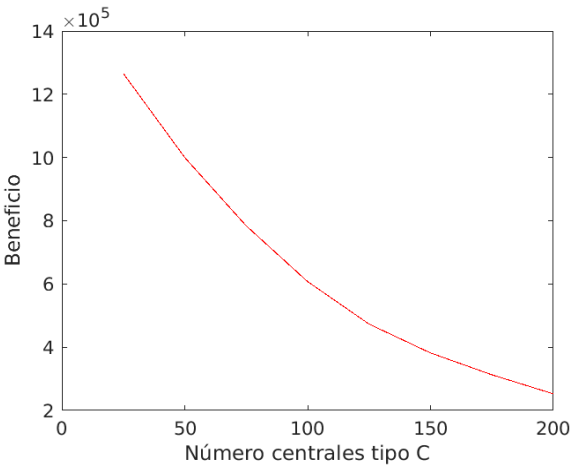
Experimento 6

Hill Climbing

Este experimento tiene como propósito ver cómo cambian los beneficios y los tiempos en función de las centrales de tipo C, que son las más pequeñas. Teóricamente, si tenemos más centrales de tipo C, deberíamos ser capaces de encontrar una asignación que use más estas centrales y menos las centrales de tipo A y B, más grandes y por tanto más caras si no las aprovechamos al completo.
Empezando con una configuración de 5 centrales de tipo A, 10 de tipo B y 25 de tipo C, aumentaremos las de tipo C de 25 en 25 para ver cómo afecta al beneficio y a los tiempos, anotaremos también el número de centrales usadas.

Centrales tipo C	Beneficio	Tiempo	Centrales usadas
25	1265965.94	1561	39
50	1000187.07	2525	59
75	784094.31	3834	74
100	607605.81	5368	86
125	473296.81	6961	94
150	381988.34	8043	99
175	312868.66	9836	102
200	252676.06	10903	103

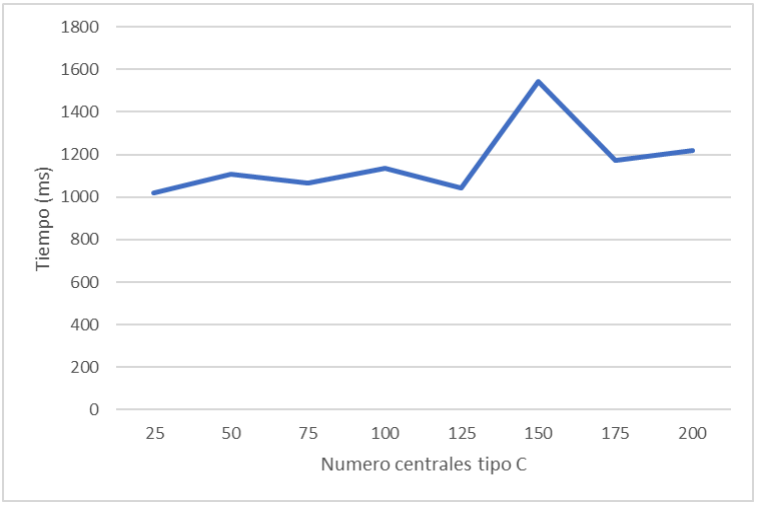




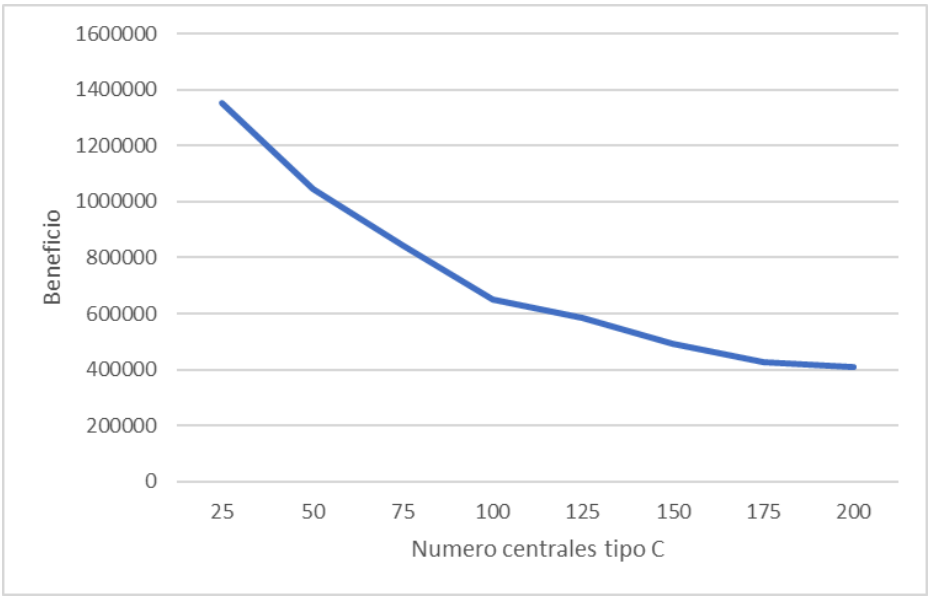
Vemos que con la heurística y operadores escogidos en anteriores experimentos el programa no aprovecha las nuevas centrales. Está tratando de minimizar los MwPerdidos por distancia (además de evidentemente tratar de tener mayores beneficios). También vemos que el incremento de centrales usadas va frenando. Esto es simplemente consecuencia de los operadores que hemos escogido y la heurística, que funcionan bien en unos casos y mal en otros. Como hemos explicado anteriormente, nuestro conjunto de operadores y heurística trata de minimizar los MwPerdidos, encontrando la distribución óptima de clientes para ello, pero tiene grandes dificultades para apagar una central, puesto que sólo puede desasignar un cliente a la vez. Ya probamos un operador en la sección 1 tratando justamente de evitar esto que no dió de igual manera buenos resultados.

Simulated Annealing

Centrales tipo C	Beneficio	Tiempo	Centrales usadas
25	1351299.9	1022	38
50	1044670.40	1108	59
75	841693.40	1064	76
100	651991.40	1134	83
125	585477.40	1041	92
150	491635.40	1543	96
175	426116.90	1175	102
200	407898.90	1218	104



Igual que el experimento anterior, el tiempo de ejecución no cambia mucho para los diferentes números de centrales.



La evolución de los beneficios en función de los números de centrales tipo C es igual que en el caso de Hill Climbing. Los beneficios son en general mejores que el caso anterior.

Vemos que el número de centrales encendidas es similar a HillClimbing, con la diferencia de que los beneficios son algo superiores. Esto es debido a que estamos usando más las centrales de tipo A y B.