Práctica 1
Documentación

Carlos Casar Morejon
Víctor González Prieto
Gerard Otero Martín

PARTE DESCRIPTIVA

Descripción del problema

El problema se basa en un sistema de servidores que contienen archivos y las peticiones que los usuarios hacen a un servidor principal, el cual redireccionará esas peticiones a otros ciertos servidores.

El objetivo del problema es hacer redirecciones de manera que el tiempo total de respuesta de los servidores a los usuarios sea el menor posible, y también que la repartición de la carga sea lo más equitativa posible.

Se nos dan dos objetos para obtener la representación del escenario: los servidores (Servers) y las peticiones (Requests). Servers se crea a partir de un número de servidores, un número máximo de replicaciones por archivo (siempre menor que la mitad del número de servidores), es decir, en cuantos servidores distintos aparecerá el mismo, y una semilla de aleatoriedad.

Es importante saber que los servers que contienen x archivo se obtienen mediante una función que implementa Servers, y que los devuelve en un Set, por lo que el acceso aleatorio queda anulado. También nos permite saber cuánto tiempo tardará un usuario x en recibir una respuesta de un servidor y (que están entre el rango de 100 y 5000 ms).

Requests se genera a partir de un número de usuarios, un número máximo de peticiones que realizará cada usuario, y una semilla. En este caso, podemos usar una función para obtener el número de usuario que hace una request x.

Mientras que los servidores y las request están ordenados de 0 a número de elementos – 1, los números de usuario y los números de archivo son números aleatorios.

Como es fácil deducir, no existe una respuesta correcta para resolver este problema, ya que no tenemos ningún estado concreto al que queremos llegar, sino que queremos explorar diferentes estados, todos posibles soluciones del problema, hasta llegar a alguno el cual nuestras funciones heurísticas no puedan mejorar y lo podamos considerar estado final. Es por esto que este problema es de búsqueda local.

Estado del problema y representación

Ya que lo que nos interesa para obtener una solución es saber a qué servidor va a ir dirigida cada request, lo más importante será tener una estructura de datos que nos lo indique, y esto lo resolvemos con un ArrayList de asignaciones request/servidor, en el cual cada posición equivale a una request, y el contenido de esa posición será el número de servidor al cual va dirigida esa request. Es decir, asignaciones[23] contendrá el número de servidor al cual va dirigida la request 23.

Luego tenemos otras estructuras y variables que usamos para calcular el valor de los heurísticos: un ArrayList con el tiempo acumulado de cada servidor en cada posición, el

tiempo total de transmisión, es decir, la suma de los tiempos acumulados, en tiempo de transmisión al cuadrado, que es el cuadrado del tiempo total de transmisión, usado para calcular la desviación estándar, el máximo de los tiempos acumulados, y el número de servidores que tienen este máximo.

Dado todo esto, podemos deducir que el tamaño del espacio de búsqueda deberá ser todas las posibles combinaciones de request/servidor. Es decir, que será o(cantidad de requests * número mínimo de replicaciones) y O(cantidad de requests * servers) cuando el número mínimo de replicaciones esté en su límite superior (servers/2) y sabiendo que la cantidad de requests será un número aleatorio de requests entre 0 y el máximo definido para cada uno de los usuarios hasta el número de usuarios también previamente definido.

Representación y análisis de los operadores

En un principio, el primer operador el cual implementamos fue un move(req,serv); el cual movía la request req al servidor serv. El factor de ramificación con éste operador es o(número de requests * replicaciones mínimas).

Otro operador en el cual pensamos fue un swap(req1,req2); el cual intercambia la request req1 con la request req2. El factor de ramificación con éste operador es también o(número de requests * replicaciones mínimas).

Problema: swap solo puede intercambiar dos requests, lo cual significa que, en un caso sencillo como por ejemplo en uno en que haya algún servidor sin ninguna request, swap ignorará ese servidor y este es un claro ejemplo de que este operador no explora todo el árbol de posibles estados sucesores. Esto sí que lo hace move, ya que no tiene problema a la hora de mover una request a otro servidor, ya esté vacío o lleno, pero este operador también tiene un problema que, aunque infrecuente, provoca que tampoco explore todo el árbol de posibles estados sucesores. Imaginemos que tenemos dos servers distintos: a y b. Tenemos las requests repartidas entre ellos tal que los tiempos acumulados de a y b son el mismo. Tenemos una ultima posibilidad en cada uno de los dos servidores: a los dos les ha sido asignada una request que en el otro tendría un menor tiempo de transmisión. ¿Qué ocurre si hacemos move del de b a a? Que el tiempo total de a aumentaría. ¿Y si lo hacemos al revés? Que aumentaría el de b. Nuestros heurísticos determinarían que se tratan de estados peores ¿Significa eso entonces que hemos llegado a un estado final? No, porque si hacemos swap de la request de a con la de b, los dos tiempos acumulados disminuirían, y por tanto tendríamos la opción de continuar expandiendo nodos.

Dicho esto, determinamos que un operador move+swap obtiene el máximo número de posibles estados sucesores, y tras haberlo pensado mucho, hemos llegado a la conclusión de que esta es la única combinación de operadores de la cual obtenemos todos los posibles sucesores del estado. El factor de ramificación de este operador es, por tanto, O(número de usuarios * número máximo de request por usuario * número de replicaciones).

Análisis de la función heurística

Según el criterio en que nos estemos fijando a la hora de ejecutar el programa, hemos desarrollado diversas funciones heurísticas.

Para el primer criterio:

Dado que este criterio se enfoca sobretodo en el máximo tiempo acumulado en uno de los servidores (el que mayor acumulado tenga), tenemos dos funciones heurísticas que tienen ese valor como máxima prioridad.

En la función A, nuestro heurístico devolverá el resultado de maxServerTime + (totalTransmissionTime/maxTotalTransmissionTime); donde maxServerTime es el tiempo máximo en un servidor, totalTransmissionTime es el tiempo total de transmisión acumulado entre todos los servidores, y maxTotalTransmissionTime es el tiempo máximo de transmisión que podríamos tener, sabiendo que el máximo tiempo por request y servidor es 5000 ms y que tenemos n requests multiplicados por estos 5000 ms. El problema de esta función es que hace trampas, ya que el criterio 1 no se debe regir en ningún caso por el total transmission time.

Es por esto que tenemos una función B, en la que implementamos algo parecido, pero sin usar el total transmission time. Seguimos considerando maxServerTime como máxima prioridad, pero en este caso, además comprobamos que no haya más servidores con ese tiempo máximo, usando maxServerTime + (maxTimeServers/nServers); donde maxTimeServers es el número de servidores que comparten éste tiempo máximo y nServers es el número de servidores. Así, cuando uno de los que comparten maxServerTime reduzca su tiempo, maxServerTime seguirá igual pero maxTimeServers/nServers hará que se reconozca este sucesor como un estado mejor.

Para el segundo criterio:

Este criterio se enfoca en que el tiempo total acumulado entre todos los servidores sea lo más bajo posible, intentando también que todos los servidores tengan un tiempo total de transmisión lo más similar posible.

Para esto tenemos una función heurística que el valor que nos devolverá será mean + stdev, donde mean es la media de tiempos totales acumulados, y stdev será la desviación estandar. Así, si la media baja sabremos que el tiempo total es cada vez más pequeño, y si la desviación también baja sabremos que los tiempos totales de los servidores son cada vez más similares entre ellos.

Luego tenemos un segundo criterio que lo único en que se fija es en el tiempo total de transmisión, por lo que siempre encontrará el tiempo total de transmisión más bajo, pero no repartirá los tiempos equitativamente entre los demás servidores.

Elección y generación del estado inicial

Hemos implementado 4 generadores del estado inicial.

El primero es muy simple: pregunta a Servers que servidores tienen el archivo de la request, y la coloca en el primero del Set que devuelve.

El segundo es algo más complicado: intenta colocar la request en el servidor al que menos requests se le hayan asignado ya.

El tercero también hace una repartición equitativa, pero este generador utiliza un Map para ello.

El cuarto reparte aleatoriamente cada request en uno de los servidores que contienen el archivo.

Tras ejecutar varias veces cada generador de solución inicial con diferentes semillas, hemos comprobado que todas tardan en general 0 ms en ejecutarse, y que en general dejan tiempos totales acumulados bastante altos y permiten la expansión de muchos nodos. Esto significa que cualquiera de los cuatro generadores es bastante bondadoso, y basándonos pues en la simplicidad, el tiempo de ejecución de la generación del estado inicial, y en el tiempo total de transmisión que resulta, hemos decidido que el mejor generador a usar será el tercero.

PARTE EXPERIMENTAL

Experimento 1 - Set de operadores

En este experimento compararemos los operadores que hemos programado. Los dos operadores son *move* y *swap* y, además de juntos, los consideramos individualmente como sets de operadores.

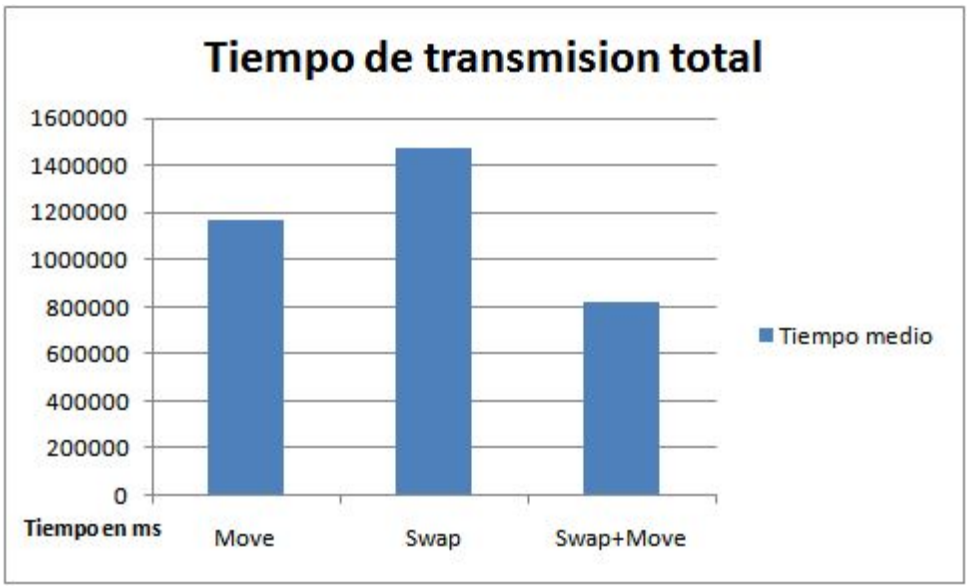
Para hacer este experimento hemos fijado el escenario habitual que se define en el enunciado (200 usuarios, máximo de 5 peticiones por usuario, 50 servidores, mínimo de 5 replicaciones por archivo). Respecto a los módulos de AIMA, hemos utilizado el generador de solución inicial #3, que reparte un poco la carga sin hacer una asignación random (no apta para hacer comparaciones, pues genera condiciones de entrada distintas para los operadores.)

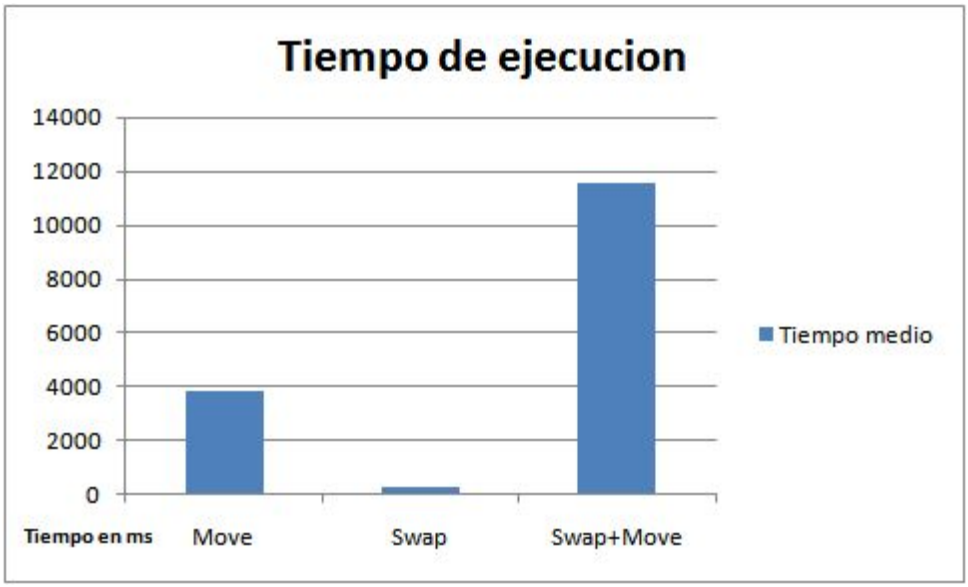
Hemos repetido el experimento 10 veces, cada repetición con una *seed* diferente para las clases de creación del dominio (del 1 al 10.)

En lo que se refiere a las expectativas: pensamos que el *swap* debería ejecutarse en menos tiempo y expandiendo menos nodos que el *move*. Ambas van de la mano, y se deben a que el *swap* tiene menos margen de aplicación que el *move*, con lo que genera menos sucesores posibles por cada estado expandido. También a causa de esto, se acerca a máximos locales más rápidamente (en perjuicio de la calidad de la solución).

No tenemos idea de cómo pueden responder las variables de interés respecto al set que junta los dos operadores. Los sucesores generados en cada estado van a ser la suma de los que generaría cada sucesor por separado, pero no sabemos cómo puede afectar esto al total de nodos expandidos.

Los resultados son los que siguen (todas las gráficas representan medias):





El experimento queda desvirtuado por el triste rendimiento de *swap*, que no funciona bien con esta solución. La distribución de peticiones que hace esta solución inicial no es lo suficientemente aleatoria con los servidores, por lo que el *swap*, que ya tiene un margen de aplicabilidad limitado de por sí, acaba expandiendo muy pocos nodos.

De hecho, tras la ejecución del experimento nos temimos que el *swap* estuviese mal programado. Al probarlo con un generador de soluciones aleatorio vimos que expande bien, mucho más que en este experimento. Otro indicador de que funciona es el hecho de que el set de los dos operadores combinados arroja soluciones sensiblemente mejores que las del *move* aplicado individualmente.

Los datos indican que, al menos para las 10 primeras *seeds*, el TTT que devuelve la búsqueda con el set combinado es del orden de ⅔ el que devuelve el la búsqueda con *move*. Respecto al tiempo de ejecución, la búsqueda combinada tarda 3 veces más en ejecutarse que la búsqueda con *move*.

Viendo que los resultados eran sensiblemente mejores, decidimos fijar el operador combinado para los siguientes experimentos. Es una decisión que se probó desacertada, la diferencia en tiempo de ejecución con el operador *move* es manejable en este experimento, pero escala muy mal. Al aumentar el tamaño del problema, ver experimento 4, el programa era incapaz de terminar tras horas de ejecución. Comparando experiencias con algunos compañeros que fijaron *move*, sus tiempos de ejecución eran mucho más manejables. En nuestro caso, la avaricia rompió el saco.

Experimento 2 - Generador de solución inicial

En este experimento debemos determinar el mejor generador de solución inicial para el problema. Para ello hemos hecho las 10 repeticiones de rigor, con distinta *seed* cada repetición y sobre el escenario habitual que habíamos comentado. Hemos pasado esos

escenarios por el filtro de las 4 soluciones iniciales que programamos. El operador fijado es el que comentamos, *move + swap*.

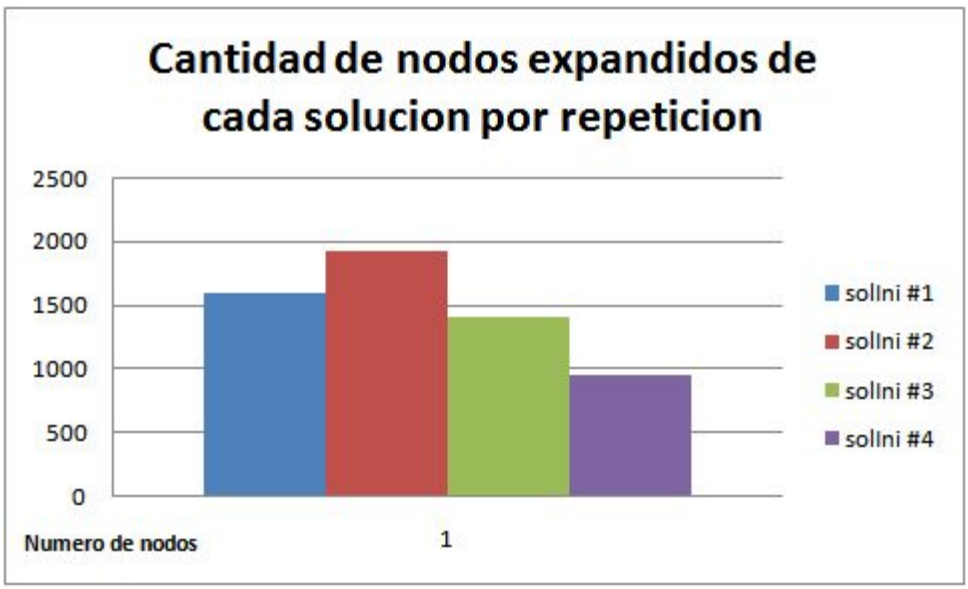
Las expectativas son que la solución #1 tardará muy poco pero su calidad será mínima, a consecuencia de esto la búsqueda tendrá que expandir más nodos hasta llegar a un máximo parcial y tardará mucho más. Cómo comentamos, su coste es lineal sobre el número de peticiones.

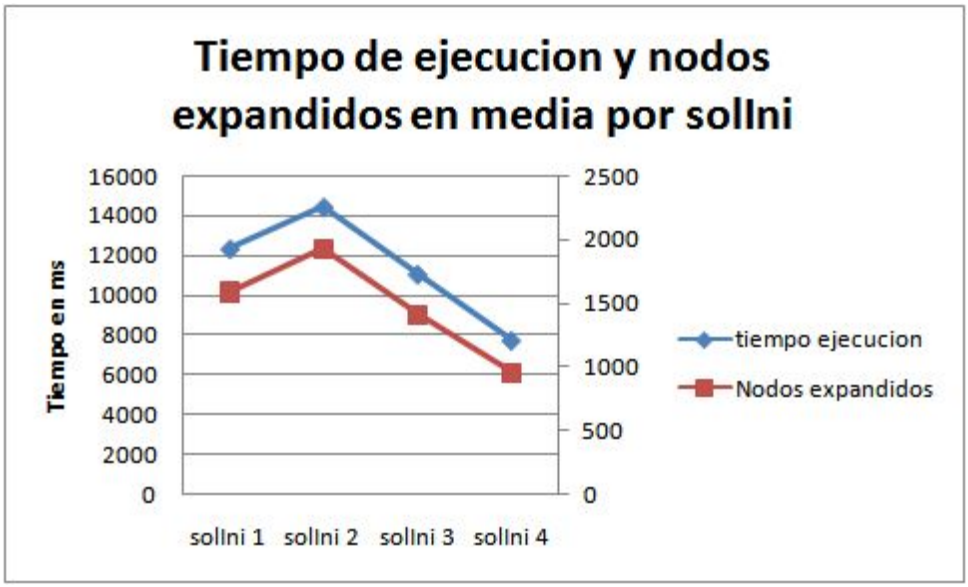
La solución #3 hace una distribución algo más inteligente, en el mismo coste temporal que #1. Eso hará que el tiempo de ejecución de la búsqueda sea menor, está un poco más cerca del máximo parcial. Pero también la calidad de la solución se resentirá.

El coste de la solución #4 ya no es tan lineal como las anteriores, pues itera aleatoriamente sobre el conjunto de servidores que lo contienen, lineal sobre el número de replicaciones. Dicho esto, si ese conjunto es pequeño (máximo de replicaciones bajo, como es el caso en este experimento) el factor añadido es casi constante respecto al del número de peticiones.

La solución #2 es algo más elaborada. Debería arrojar soluciones iniciales de mejor calidad que las de #1 y #3, pero a costa de tardar más. No me atrevería a decir que las soluciones serán mejores que en el caso del generador #4, el random debería nivelar la carga mejor que ninguno.

El experimento arrojó resultados interesantes:





Para empezar, los tiempos de ejecución de los cuatro generadores son despreciables, al menos para esta instancia de las variables del dominio/problema. Podemos comparar los generadores en función de variables como la calidad de la solución inicial, los nodos que expanden, el tiempo de ejecución de la búsqueda o la calidad de la solución final.

Las diferencias de calidad de soluciones (inicial y final) para cada generador son mínimas. Todos generan soluciones con una calidad sorprendentemente parecida; las búsquedas convergen a soluciones parecidas desde cualquier solución inicial. ¿Problema de heurístico?

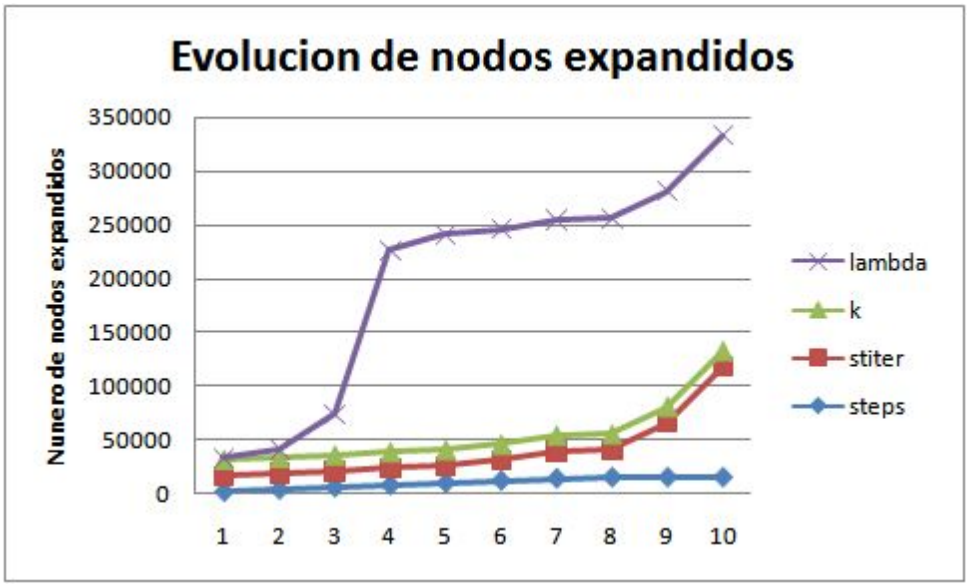
El único factor de diferenciación es el tiempo que tarda la búsqueda según la solución inicial. Aquí vemos que las que parten de una asignación aleatoria de las peticiones acaban bastante antes que las demás. Curiosamente la #2 arroja los peores resultados (pese a repartir la carga más equitativamente).

La naturaleza aleatoria del generador #4 desaconseja su uso en la experimentación, por lo que utilizaremos en adelante el #3, primero de los perdedores.

Experimento 3 - Parámetros del Simulated Annealing

Para este experimento debemos escoger los mejores parámetros para Simulated Annealing. Para ello, haremos 10 repeticiones. En cada repetición, variamos los diferentes parámetros (steps, stiter, k, lambda). Para determinar el mejor de cada uno de ellos, escogeremos el mínimo tiempo de ejecución y el mínimo tiempo de transmisión, dando prioridad a este último, debido a que suelen ser de crecimiento opuesto, nos tenemos que fijar el momento en que ambos se estancan y mejoran poco o en el momento en que el tiempo de ejecución se dispara y el de transmisión apenas mejora. Un parámetro que nos ayuda bastante con esto es el número de nodos expandidos, que cuando llega al máximo y se estanca es que ya ha abarcado el máximo posible, por tanto debes parar ahí.

Teniendo en cuenta los parámetros de decisión y con la tabla, estos son los valores de los parámetros que hemos decidido:



Steps: 1000000

Stiter: 10000

K = 25

Lambda = 0.0008

Experimento 4 - Variando los números de usuarios y servidores

En este experimento tenemos que observar cómo evolucionan las variables de interés (tiempo de ejecución y tiempo total de transmisión) cuando variamos el número de usuarios y cuando variamos el número de servidores.

Este experimento resultó fallido por la poca escalabilidad del operador combinado. Una sola repetición (una seed, todo el rango de nusers o nservidores) tardaba horas en ejecutar. Sólo hemos podido ejecutar dos repeticiones. Estos son los resultados:

En el primer caso, variación del número de usuarios, vemos que aumenta el tiempo de ejecución. Se diría que en una proporción exponencial, aunque es dificil de asegurarlo con 2 repeticiones.

El aumento del número de usuarios implica el aumento de peticiones en un factor proporcional al número de máximo de peticiones por usuario (que no cambia.) Ese aumento en el número de peticiones (lineal sobre el número de usuarios) supone que el factor de ramificación en cada paso de la búsqueda aumente. Eso explica porqué el tiempo de ejecución se dispara.

El tiempo total de transmisión también aumenta, como es lógico. Los servidores, que no han aumentado, tienen que repartirse una número de peticiones mayor, acaban más cargados.

En el segundo caso, variación del número de servidores, vemos que el tiempo de ejecución disminuye conforme aumenta dicho número. Esto se debe a que, con el número de peticiones constante, los servidores están mucho menos cargados; se reparten el mismo

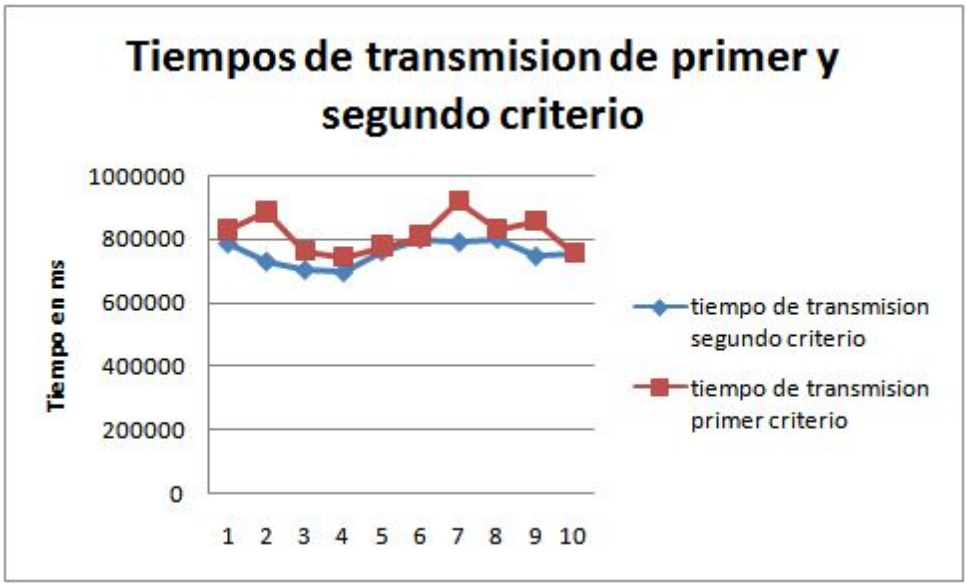
tráfico entre muchos más. Cuantas menos peticiones por servidos hayan asignadas, más dificil será mejorar una solución: el *move* sólo empeorará el heurístico y el *swap* sigue igual de acotado, pues el número máximo de replicaciones no ha variado.

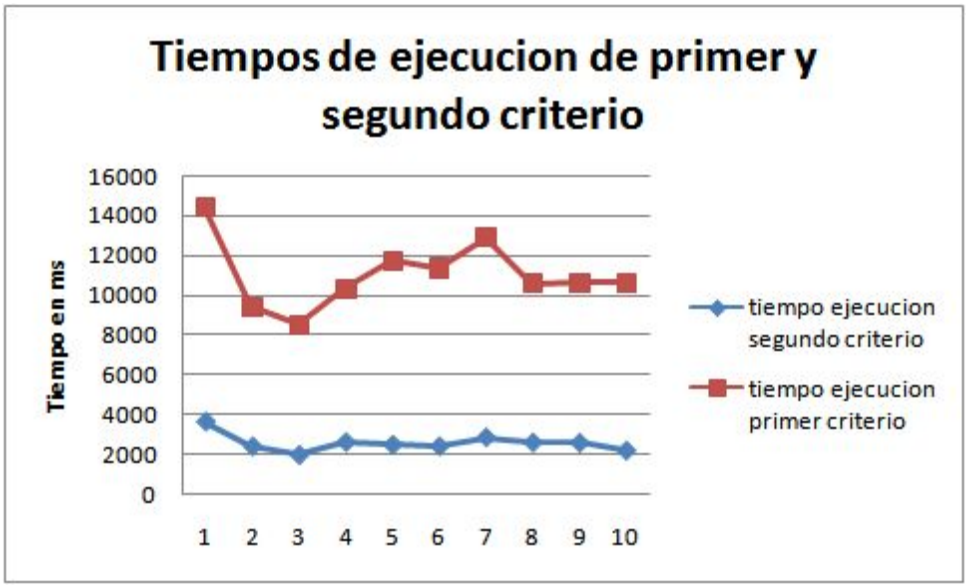
Experimento 5 - Comparando heurísticos (HC)

El objetivo aquí era ejecutar una búsqueda por Hill Climbing usando cada uno de los heurísticos y comprobar las diferencias entre ellos.

No partíamos con ninguna expectativa, hemos comprobado que el primer criterio puede estancarse en una meseta bajo ciertas condiciones algo particulares (mala distribución de peticiones en la solución inicial unido *swap* como operador único). Pero el segundo criterio lo testamos por primera vez.

Los resultados fueron los siguientes:





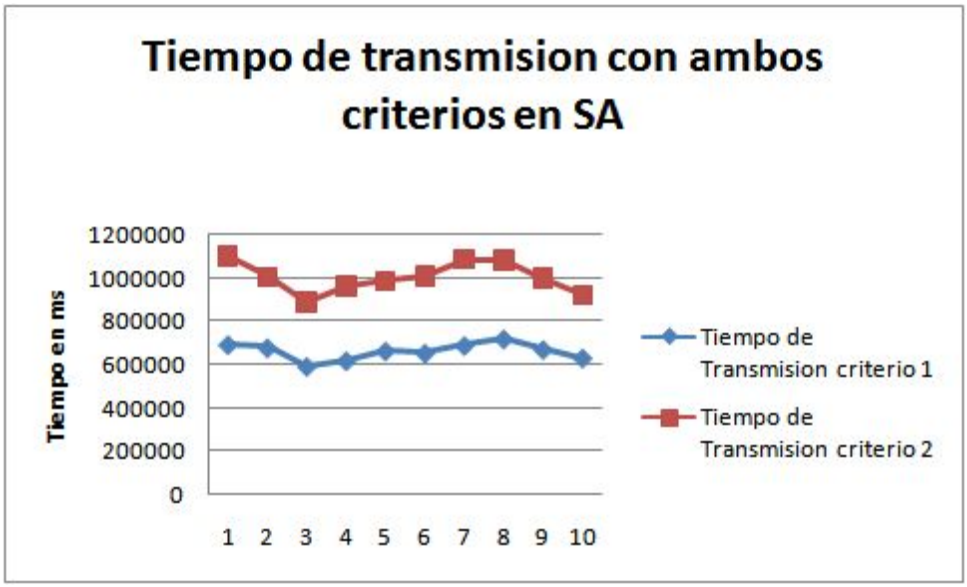
Podemos ver que en el apartado de calidad de solución los resultados son muy parecidos. Parece que ambos heurísticos hacen converger la búsqueda hacia soluciones que son bastante buenas. Supongo que son bastante buenas porqué hemos llegado a ellas por dos vías bastante distintas.

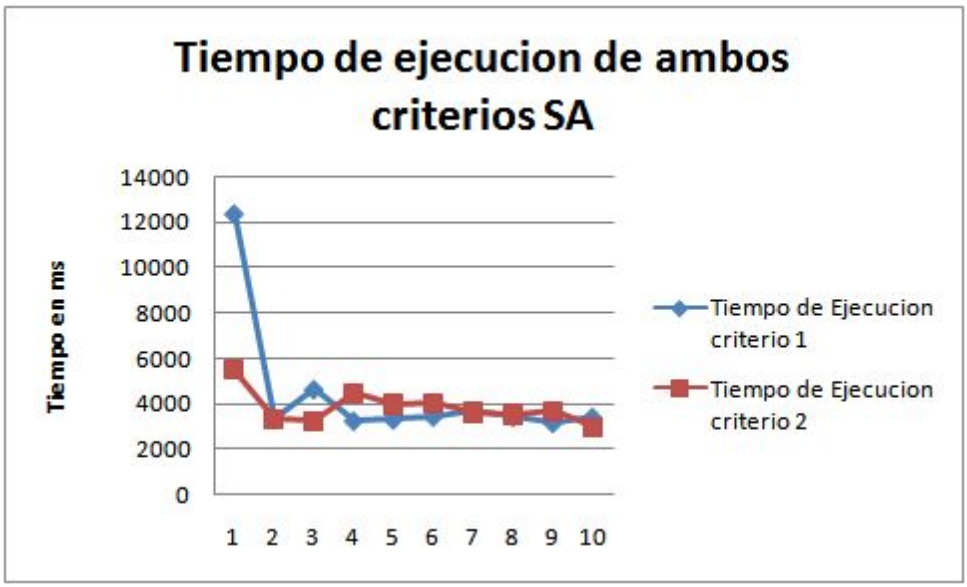
Pero en el departamento del tiempo de ejecución parece que el segundo heurístico es mucho más eficiente. Fijarse en el tiempo total de transmisión (y en que los tiempos de cada servidor estén bien balanceados) parece que hace converger la búsqueda mucho antes.

Experimento 6 - Simulated Annealing vs Hill Climbing

En este experimento teníamos que ejecutar la búsqueda por Simulated Annealing, con el escenario habitual ya comentado y para los heurísticos que implementan los dos criterios exigidos. El objeto era comparar los dos métodos de búsqueda local.

Por la naturaleza del SA, los resultados deberían ser mejores en todos los departamentos. En lo que respecta al tiempo de ejecución, SA no genera todos los sucesores de un estado como lo hace HC, genera uno aleatoriamente; por lo que, además, sólo tiene un heurístico que evaluar. Respecto a la calidad de la solución,





Cómo era de prever, el Simulated Annealing arroja unos tiempos de ejecución despreciables (excepto por un outlier en la ejecución para seed 1, heurístico 1.) Pero no sólo mejora a Hill Climbing en eso, sino que además encuentra soluciones mejores que las de Hill Climbing, confirmando lo que hemos visto en clase de teoría, Simulated Annealing encuentra mejores soluciones.

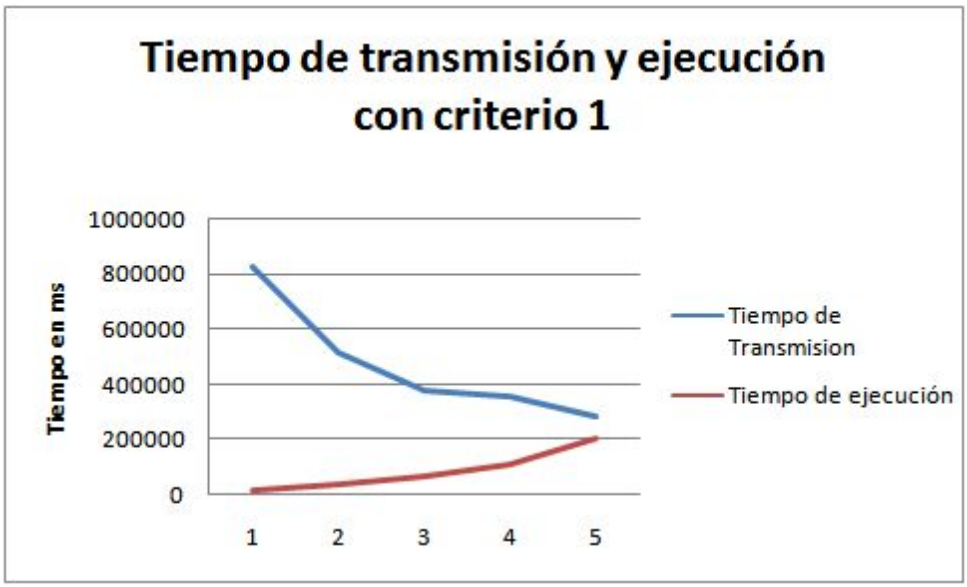
Experimento 7 - Variación en el número de replicaciones

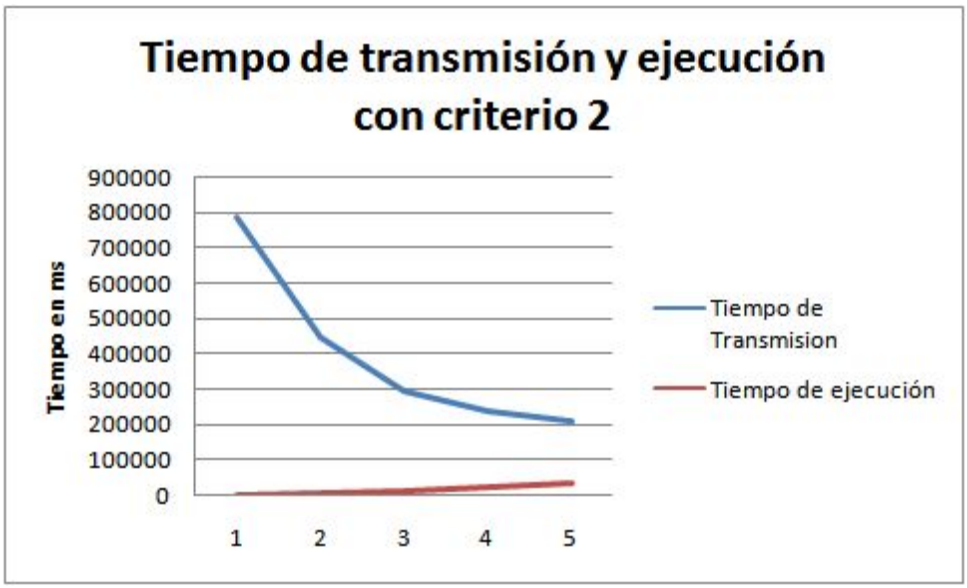
El objeto de este experimento era comprobar cómo afecta el número de replicaciones a las variables de interés. Las variables del problema vienen definidas por el escenario habitual que venimos utilizando.

A priori parece que cuantas más replicaciones, más opciones para repartir cada petición entre los servidores. Eso mejorará la distribución de la carga y por lo tanto el tiempo total de

transmisión bajará. Pero el factor de ramificación es muy sensible a esta variable, y cuanto más aumente el número mínimo de replicaciones, mayor debería ser el tiempo de ejecución.

Estos son los resultados:





Vemos que, en efecto, el tiempo total de transmisión desciende, de forma pronunciada, hasta que empieza a estancarse en los 200.000 ms. El tiempo de ejecución aumenta, pero muy tímidamente, para nada en la proporción que esperábamos.

Experimento 8 - Conclusiones

Con los resultados obtenidos, podemos concluir, que el que da mejor resultado, tanto el tiempo total de transmisión como en tiempo de ejecución es el SA con criterio 1 y con los parámetros definidos en el experimento 3. El segundo criterio no le funcionaba igual de bien, ya que en tiempo total de transmisión era bastante más lento. En el caso de HC, con el

segundo criterio, estaba parejo en cuanto a tiempo de ejecución, pero un poco más lejos que SA en tiempo total de transmisión.