Práctica 8

*

December 30, 2015

1 Introducción:

1.1 Exposición de la práctica:

En la práctica presentada se nos pide implementar los algoritmos de optimización denominados búsqueda local(para una sola solución) y algoritmos genéticos (varias soluciones) para el conjunto de datos dantzig42. Para ello hemos desarrollado nuestro código en el lenguaje de programación Java y después hemos creado nuestro ejecutador .jar. Además hemos desarrollado este documento para plasmar las partes más importantes de la práctica.

1.2 Objetivos:

Los objetivos son claros; conseguir un algoritmo eficiente y efectivo para optimizar una solución o un conjuntos de soluciones de un conjunto de datos. El problema planteado tiene nombre propio TSP. El código que hemos desarrollado no es dependiente a un solo conjunto de datos, ya que es posible utilizarlo en más de un conjunto de datos (menos la lectura).

Como hemos recibido el archivo dantzig42 para desarrollar la práctica, vamos a analizar el archivo para ver su contenido. En el archivo de datos podemos encontrar un conjunto de valores numéricos separados por ceros. Estos ceros indican un salto de linea. Si procesamos estos datos, conseguimos la mitad una matriz de una dimensión de 49x49. Como tenemos 49 ciudades, el punto en común de los pares de ciudades nos indica cual es el peso de una ciudad a otra.

Para optimizar nuestras ciudades vamos a aplicar distintos criterios en el proceso. Empezaremos utilizando el criterio Best First y luego utilizaremos el Greedy, para la búsqueda local. Una vez conseguidos estos procesos, haremos el calculo de el algoritmo genético.

2 Explicación y Pseudocódigo del algoritmo:

2.1 Búsqueda Local:

En el los algoritmos de búsqueda local, tanto en Greedy o BestFirst, se recibe la matriz de distancias con la cual aleatoriamente crea un camino el cual se intuye que no es el optimo y se procede a optimizar. En ambos dos, se crea una combinatoria de los parámetros de la solución aleatoria sacada (en nuestro caso utilizamos swap). Luego escogeremos la primera que optimize un poco el resultado aleatorio o la mejor de todas las combinatorias sacadas, dependiendo si estamos con el BestFirst o Greedy. Repetimos este proceso hasta que no se mejore en una iteración o nos quedemos sin tiempo o llamadas a la función de costo. Con estos métodos no se asegura la solución optima, pero son muy rápidos, aunque dependiendo de la primera solución aleatoria cambia mucho el resultado final.

Entrada: TSP
Salida : Recorrido
$Recorrido = sacarRecorridoAleatorio()$;
while $Valor(Recorrido) < Valor(RecorridoAnt)$ & $!TiempoComputacionalAcabado$ & $!Numerodeiteracionesmaximo$ **do**
  $RecorridoAnt = Recorrido\ Combinatoria = CrearCombinatoriaDeSwaps()$
  $Recorrido = BusquedaMejorRecorrido(Combinatoria)$
end
return *Recorrido*

Algorithm 1: Búsqueda Local

2.2 Algoritmo Genetico:

En el algoritmo genético también se genera una solución a partir de un modelo TSP. Pero en este caso, lo primero es generar una población aleatoria de soluciones, es decir, crear varias soluciones que luego se cruzarán y mutarán para ir mejorando la solución hasta que no podamos llamar más veces a la función de costo o el tiempo de ejecución de acabe.

*

Para hacer los cruces hemos utilizado un algoritmo llamado CrossOver. Este algoritmo recibe dos padres puede crear una o varias soluciones. En nuestro caso hemos creado 2. Primero se coge un segmento de uno de los padres y se copia tal cual al hijo, luego se va rellenando los huecos restantes con los datos del segundo padres empezando desde la posición del ultimo número metido, sin repetir los números.

Entrada: $Camino1, Camino2$
Salida : $CaminoHijo$
Elegir segmento de azar y copiarlo de $Camino1$ en $CaminoHijo$
for *Para cada elemento fuera del segmento, empezando desde el final del segmento* **do**
  if *el elemento seleccionado de* $Camino2$ *no esta en el segmento seleccionado de* $Camino1$ **then**
    colocarlo en orden después del segmento en $CaminoHijo$ (si lista está llena se empezará desde el principio hasta el principio del segmento
  end
end
return $CaminoHijo$

Algorithm 2: Order 1 CrossOver

Los hijos sufrirán mutaciones en el 30 % de los casos (criterio establecido por nosotros) y se hará un nuevo conjunto de datos entre los padres y los hijos mutados y no mutados. De esta forma se obtendrá una población del mismo número de la que teníamos de entrada.

El algoritmo puede converger en estos tres casos:

- Tiempo computacional.
- Numero de iteraciones.
- Población de rasgos muy similares.

entrada: TSP
salida : Camino
while $!TiempoComputacionalAcabado \ \& \ !NumeroDeIteracionesMaximo$ **do**
  $PoblacionOrdenada = OrdenarPoblacionListaCiudades()$
  $Cruces = CrearCruces(PoblacionOrdenada/2)$
  $mutaciones = CrearMutaciones(Cruces)$
  $PoblacionListaCiudadesMejoradas =$
   $ReemplazarMutacionesPorMuestrasDeBajoNivel(PoblacionOrdenada, Mutaciones)$
end
return $PoblacionListaCiudadesMejoradas$

Algorithm 3: Algoritmo Genético

3 Experimentación y Resultados:

Una vez que tenemos el archivo 'practica7.jar', lo único que hay que hacer es dejar al lado el fichero del TSP 'dantzig42'. Para ejecutarlo solo hace falta ejecutar la siguiente linea:

```
> java -jar practica7.jar
```

Aunque si queremos se puede ejecutar cada método independientemente, cambiar el número de vueltas (por defecto 10),cambiar la población del GeneticAlgorithm o especificar si queremos muestras aleatorias uniformes o sesgadas. Para ello hay que introducir los siguientes comandos:

```
-checkAleatorio : si se quiere comprobar los resultados del aleatorio normal y sesgado.
-greedy : si se quiere comprobar el algoritmo Greedy.
-bestfirst : si se quiere comprobar el algoritmo Best First.
-genetic : si se quiere comprobarf el algoritmo Genetico.
-l num  : para cambiar el número de vueltas (ej: -l 20)
-pobmax num : para cambiar la poblacion del algoritmo Genetic (ej: -pobmax 1000)
-aleatorioNormal : para que se ejecute con el aleatorio normal.
-aleatorioSesgado : para que se ejecute con el aleatorio sesgado.
```

Y un ejemplo sería:

```
> java -jar practica7.jar -greedy -bestfirst -l 50 -aleatorioNormal
> java -jar practica7.jar -genetic -l 20 -aleatorioSesgada -pobmax 300
```

Hemos ejecutado el programa de la manera simple y nos han salido estos resultados:

```
########################################
Comprobaremos si es mejor empezar con un aleatorio uniforme o sesgada:
Aleatorio uniforme:
        Distancia Media: 3139.0
        Tiempo de ejecucion Medio: 0
Aleatorio sesgado:
        Distancia Media: 2418.0
        Tiempo de ejecucion Medio: 26
Para la selección de tipo de aleatoriedad nos basaremos en la distancia.
        Elegimos aleatoriedad sesgada que tiene distancia más baja.
########################################
Analizaremos como funciona el Greedy: (10 vueltas)
        Se tarda una media de 35ms para ejecutarse cada greedy.
        La media de distancia: 1039.0
La mejor opcion salida:
[21,20,28,32,29,27,11,12,13,14,15,17,18,19,16,23,24,26,25,30,31,33,34,35,36,37,38,39,40,41,0,1,4,5,6,3,2,7,8,9,10,22]
        ->  864.0
########################################
Analizaremos como funciona el BestFirst: (10 vueltas)
        Se tarda una media de 37ms para ejecutarse cada Bestfirst.
        La media de distancia: 980.6
La mejor opcion salida:
[14,13,12,11,9,1,0,41,40,39,38,37,36,35,34,4,3,2,7,8,6,5,33,32,31,30,29,28,27,26,25,24,23,10,22,21,20,19,18,17,16,15]
        ->  777.0
########################################
Analizaremos como funciona el GeneticAlgorithm: (10 vueltas)
        La población será de: 2500
        Se tarda una media de 21733ms para ejecutarse cada GeneticAlgorithm.
        La media de distancia: 786.3
La mejor opcion salida:
[40,39,38,37,36,35,34,33,32,31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0,41]
        ->  699.0
########################################
```

Como vemos en los resultados obtenidos, los resultados son bastante similares en los tres algoritmos. En todos ellos utilizamos un algoritmo, de creación de vecinos o de una sola solución, que nos devuelve una solución ligeramente mejor que una aleatoria. Utilizamos este tipo de algoritmo ya que teniendo una solución mejor desde un principio conseguimos unos resultados un poco mejor que utilizando soluciones totalmente aleatorias, aunque tarde bastante más en sacarlo.

En el algoritmo genético hemos ido utilizando distintos valores para el parámetro que mide el número de vecinos que utilizamos durante todo el algoritmo. Después de varios intentos hemos decidido poner 2500 de número de población ya que ofrece buenos resultados en un tiempo de ejecución razonable. De todos modos, si se quiere, se puede incluir en la ejecución el numero de vecindario que se quiere.

Como en el algoritmo genético se empieza con una población aleatoria, el proceso de crearla mediante nuestro método puede tardar un poco ya que para crear una único camino se necesitan 26ms. Con lo que ejecutarlo 2500 veces llevará al rededor de un minuto cada vez. Mas lo que tarde en ejecutarse el propio algoritmo, que no es mucho. Pero cabe destacar que la diferencia es notable. Los resultados bajan 100 unidades aproximadamente en el caso del algoritmo genetico.

4 Conclusiones:

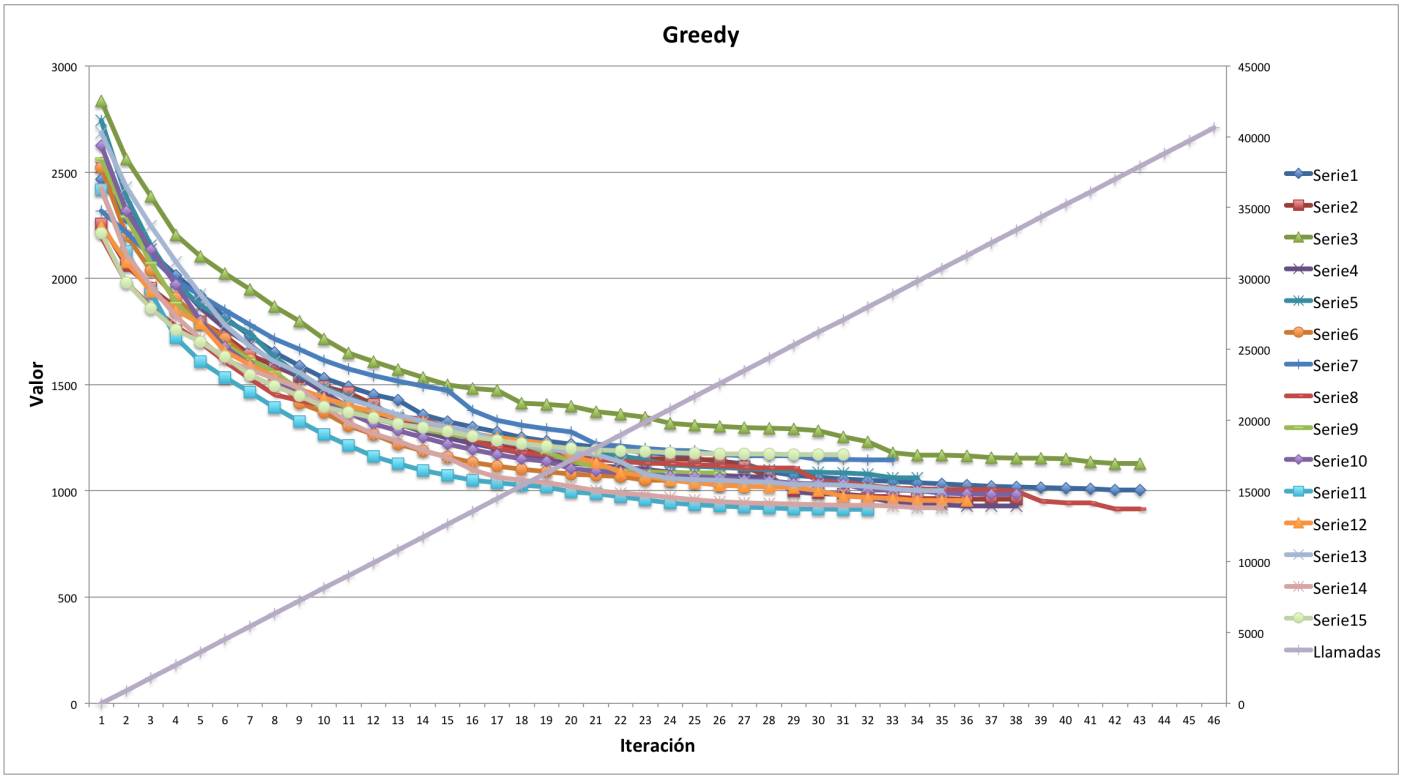
Tras estar probando con los algoritmos y jugar con sus parámentros en el caso del algoritmo genético, hemos sacado las siguientes conclusiones.

4.1 Greedy:

En el caso del greedy como ya intuíamos, en cada iteración se hacen el mismo número de llamadas a la función de costo y en las primeras iteraciones este costo disminuía mucho más que al final, lo que es normal ya que el margen de mejora es menor.

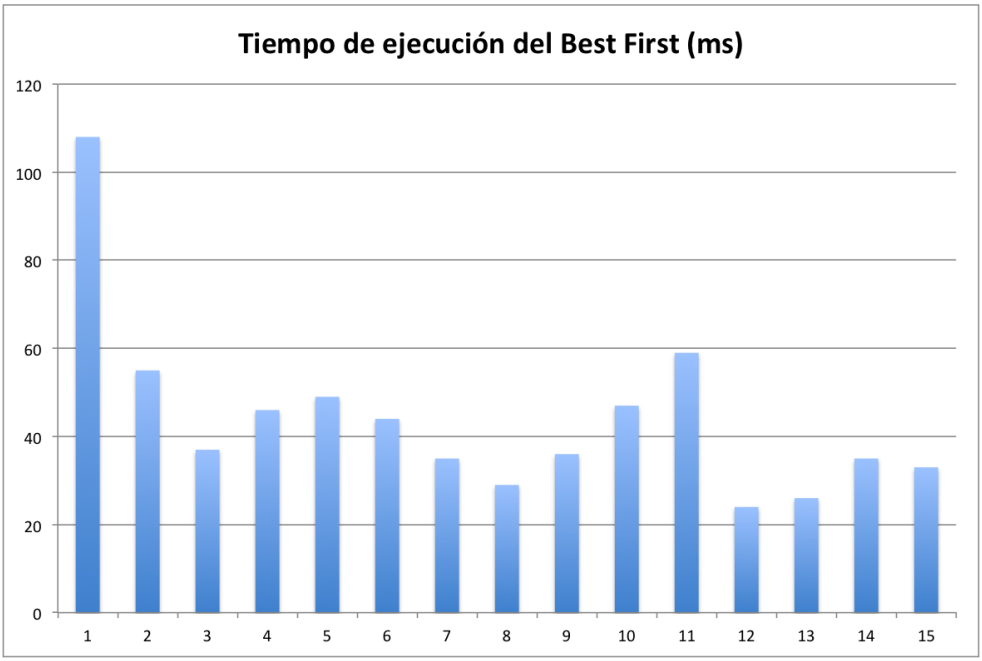
Se puede decir que este algoritmo siempre converge en una solución parecida aunque las iteraciones necesitadas y por lo que lleva su tiempo de ejecución sea muy distinto. A la vez de que no por tener más iteraciones sabemos que va a terner una solución mejor, como podemos ver en el gráfico de abajo, la serie 3, ha sido una de las que mas ha tardado y de las que peor resultado ha obtenido.

En este caso el algoritmo converge más rápido que el BestFirst.



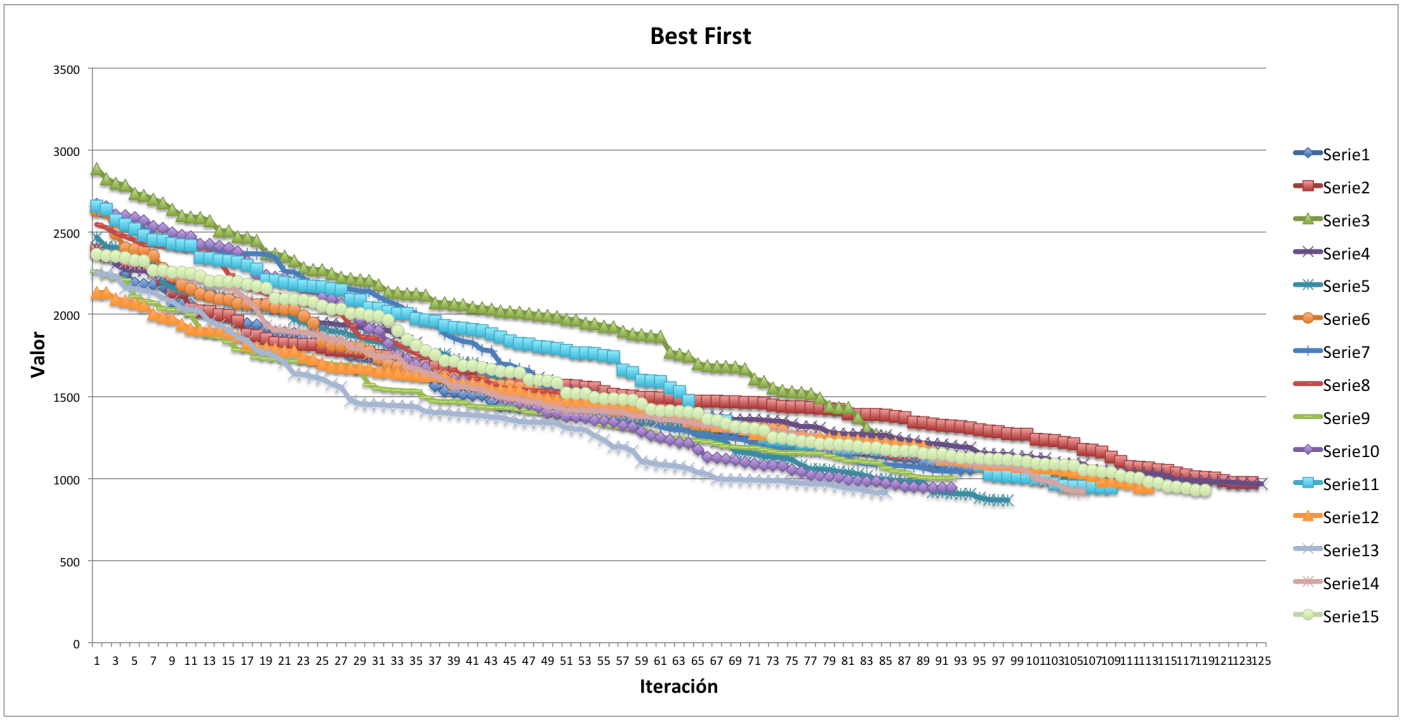
4.2 Best First:

Como podemos ver en este gráfico de tiempos, este algoritmo es muy rápido, pero su duración es muy diferente cuando cambiamos el camino aleatorio de inicio. Y aunque no lo hayamos probado, con la implementación podemos saber que también puede influir el tipo de combinación que se utilice.



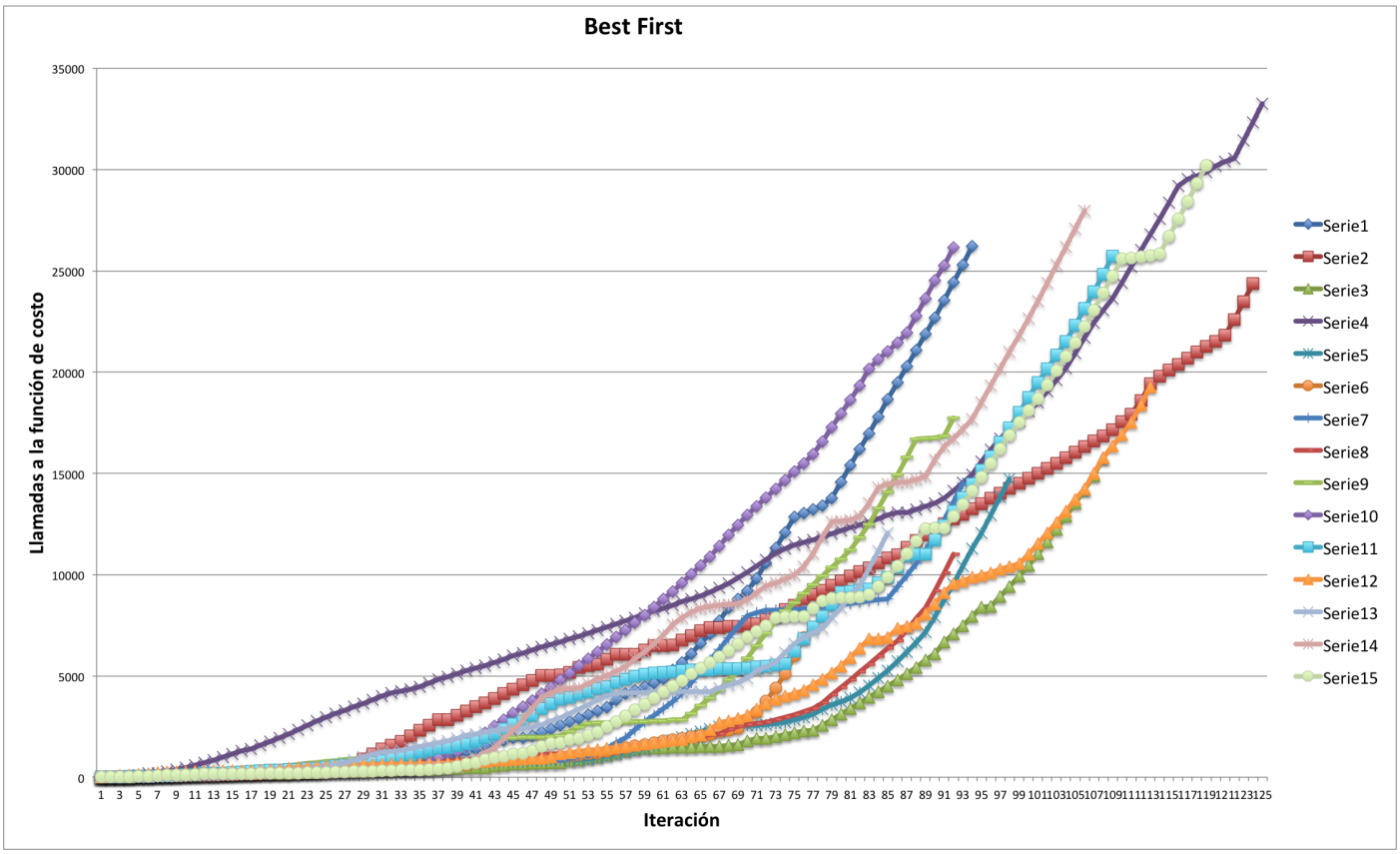
Para analizar este algoritmo hemos tenido que separar en dos gráficos como disminuye el valor de costo y como aumenta las llamadas a la función de costo en cada iteración.

En cuanto al valor de costo, con el Best First, nos hemos dado cuenta que baja de manera uniforme en cada iteración y que al igual que el greedy, converge en soluciones parecidas aunque las iteraciones necesitadas sean diferentes.



Si nos fijamos en las llamadas que ha utilizado en cada iteración y cada serie, podemos ver que no sigue una tendencia clara aunque parezca que en las primeras iteraciones casi no necesite ninguna llamada para mejorar. Pero esto es irregular porque busca el primer camino que sea algo mejor que el padre, y esto puede suceder en cualquiera de las opciones, aunque es verdad que en las primeras iteraciones hay mas opciones de mejora, con lo que la probabilidad de que la mejora esté entre las primeras es mayor. Y que en las últimas iteraciones hace más llamadas ya que hay pocas mejoras.

Por lo que se puede decir que tiene una tendencia exponencial.



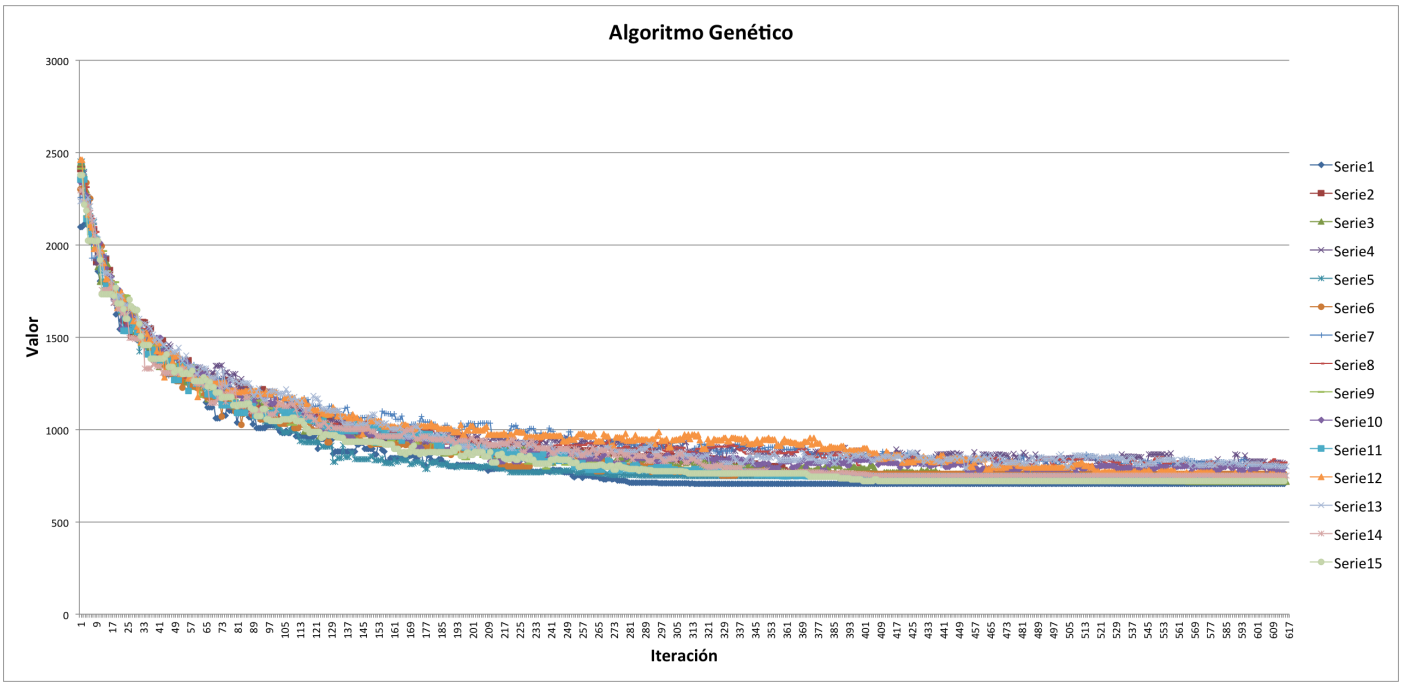
Y mirando los dos gráficos a la vez, concluimos que al igual que en el Greedy, el número de iteraciones que se hayan echo no implica que el resultado sea mejor, además de que en este caso tampoco sabemos si se han necesitado más o menos iteraciones ya que es bastante irregular.

4.3 Algoritmo Genético:

Se puede decir que este algoritmo es mas sofisticado que los anteriores y que saca resultados mejores y más estables. también es verdad que es necesario ajustar tanto los parámetros de población como de llamadas necesarias.

Como podemos ver en el gráfico de abajo, en las ultimas iteraciones apenas hay mejora, con lo que se podría reducir el número de llamadas y no habría mucho cambio, siempre que le demos margen para que llegue a la zona estable.

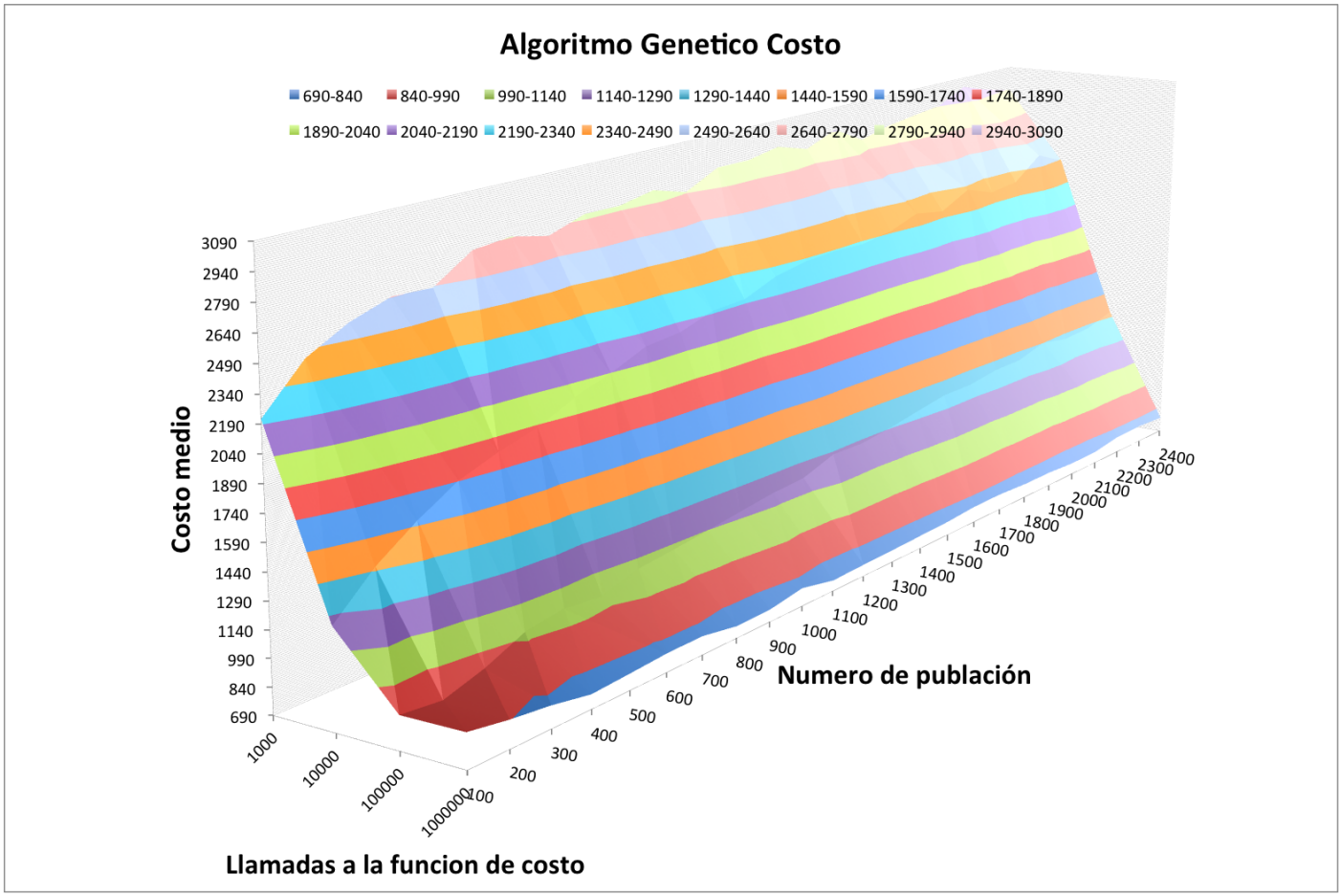
En las primeras iteraciones baja muy rápido, igual que el greedy, pero este necesita muchas más iteraciones para estabilizarse. Otra de las apreciaciones es que siempre da una solución parecida, ya que en todas mejora de una forma regular. Esto puede cambiar si le damos menos población, ya que nosotros le dimos una población bastante grande (2500).



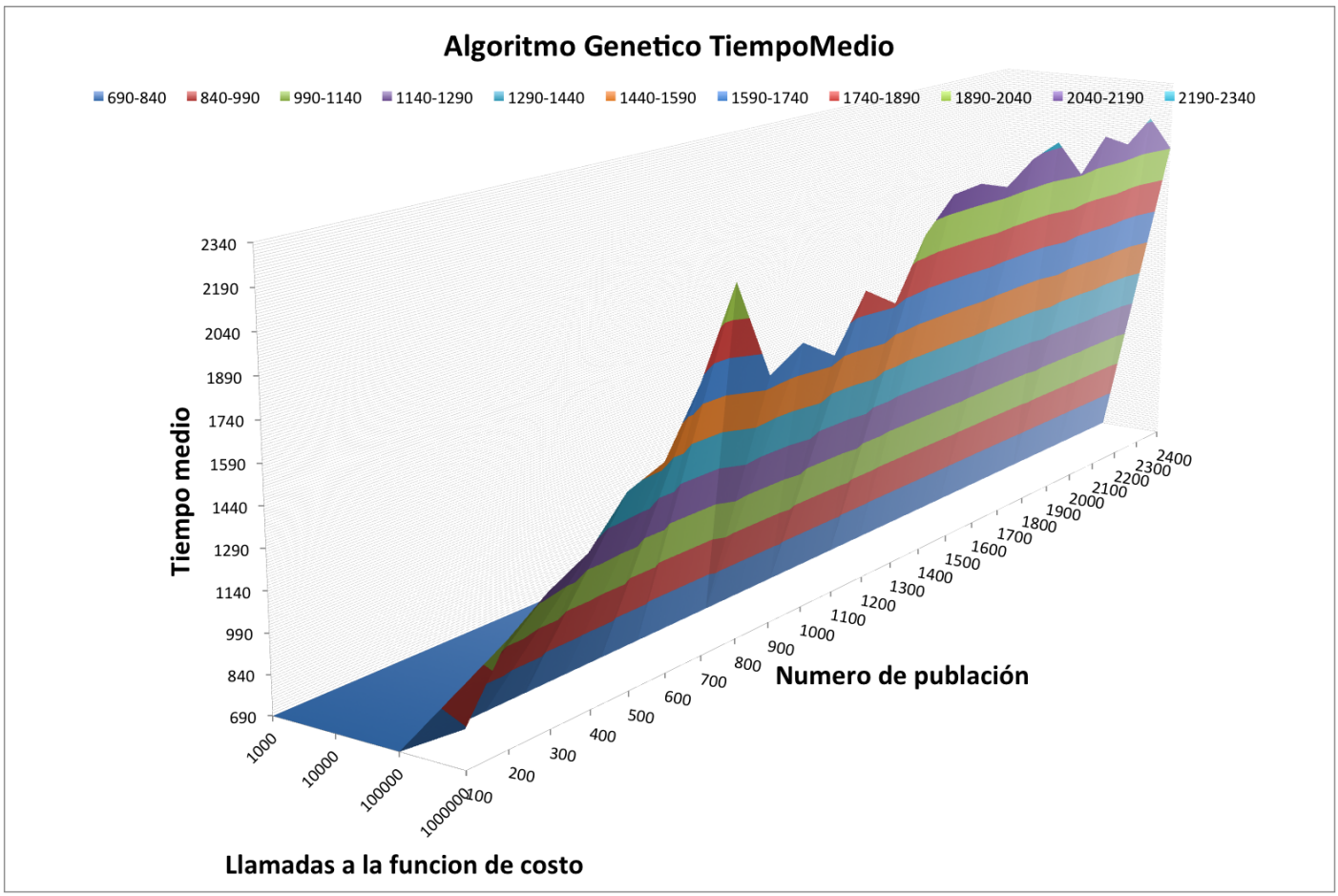
Como este algoritmo es más sofisticado, hemos entrado más a fondo y hemos sacado el gráfico de abajo que es la media de resultados que nos da con 10 secuencias para cada opción, mientras cambiamos la población de 100 a 2.400 y las llamadas a la función de costo de 1.000 a 1.000.000 en cuatro pasos.

Podemos apreciar que la población tampoco debe ser muy grande ya que no mejora casi a partir de una población de 400, aunque es importante mirar que cuantas más veces llamemos mejor, aunque en la última no se haya mejorado mucho para el cambio que ha habido (100.000 − 1.000.000).

Por lo que viendo este gráfico nos quedaríamos con una población de 400 y 1.000.000 de llamadas.



Como era de esperar, el tiempo de ejecución sube igual que suben las llamadas a la función de costo, pero podemos apreciar que cuando la población es baja no tarda mucho ya que hace menos combinaciones.



4.4 Camino Aleatorio:

Al sacar el camino aleatorio, vi que sería bueno si no escogiéramos caminos al azar puro y podríamos coger caminos mirando los valores que teníamos. Lo que hicimos fue dar importancia a los caminos con menor viaje con esta formula:

$$Peso = \frac{1}{distancia}$$

Así la probabilidad de que saquemos caminos mejores es mayor, con lo que podemos ver que en una ejecución de 500 iteraciones ($java\ -jar\ practica7.jar\ -checkAleatorio\ -l\ 500$) mejora 600 unidades aunque tarde un poco más en sacarlo, pero consideramos que es una mejora importante.

```
########################################
Comprobaremos si es mejor empezar con un aleatorio uniforme o sesgada:
Aleatorio uniforme:
         Distancia Media: 3111.03
         Tiempo de ejecucion Medio: 0
Aleatorio sesgado:
         Distancia Media: 2465.89
         Tiempo de ejecucion Medio: 9
Para la selección de tipo de aleatoriedad nos basaremos en la distancia.
         Elegimos aleatoriedad sesgada que tiene distancia más baja.
Se ejecutará con aleatoriedad sesgada.
########################################
```

Hay que mencionar que tanto para el Greedy como para el BestFirst, utilizar un camino aleatorio sesgado es favorable en todos los sentidos, ya que solo se usa una vez y el tiempo de ejecución es insignificante para este caso. Por otro lado, en el Algoritmo Genético, como se ejecutan tantas veces como el número de población, ya que se utiliza para iniciar la población, se nota en tiempo si se está utilizando un algoritmo u otro. Por eso, ya que con este algoritmo los resultados conseguidos son muy parecidos (774 sesgando frente a 779 con aleatoriedad uniforme (ejecutando 100 veces con una población de 500)), proponemos que este algoritmo se utilice con una aleatoriedad uniforme.

5 Valoración Subjetiva:

Ieltzu: Ha sido una práctica en la que no hemos invertido mucho tiempo. Los tres teníamos que haber estudiado los dos algoritmos, por lo tanto, no ha sido muy complicado implementarlo. Práctica interesante, pero en la época del año que nos ha pillado, para mi, sobraba.

Mikel: Es una práctica diferente a las demás y en la que me he llevado varios mosqueo con la función de aleatoriedad sesgada, y a la hora de implementar los cruces, ya que cambiamos de método ya que no conseguíamos implementarlo. Pero además de eso es interesante ver las distintas maneras de optimización y analizarlas como funcionan.

Maria: (No ha estado este periodo de tiempo, así que no lo ha echo.)

Bibliografia

- http://www.herrera.unt.edu.ar/gapia/Curso_AG/Curso_AG_08_Clase_5.pdf
- http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/t2geneticos.pdf