Práctica de algoritmos evolutivos

Adrián de la Torre Rodríguez

1. Índice

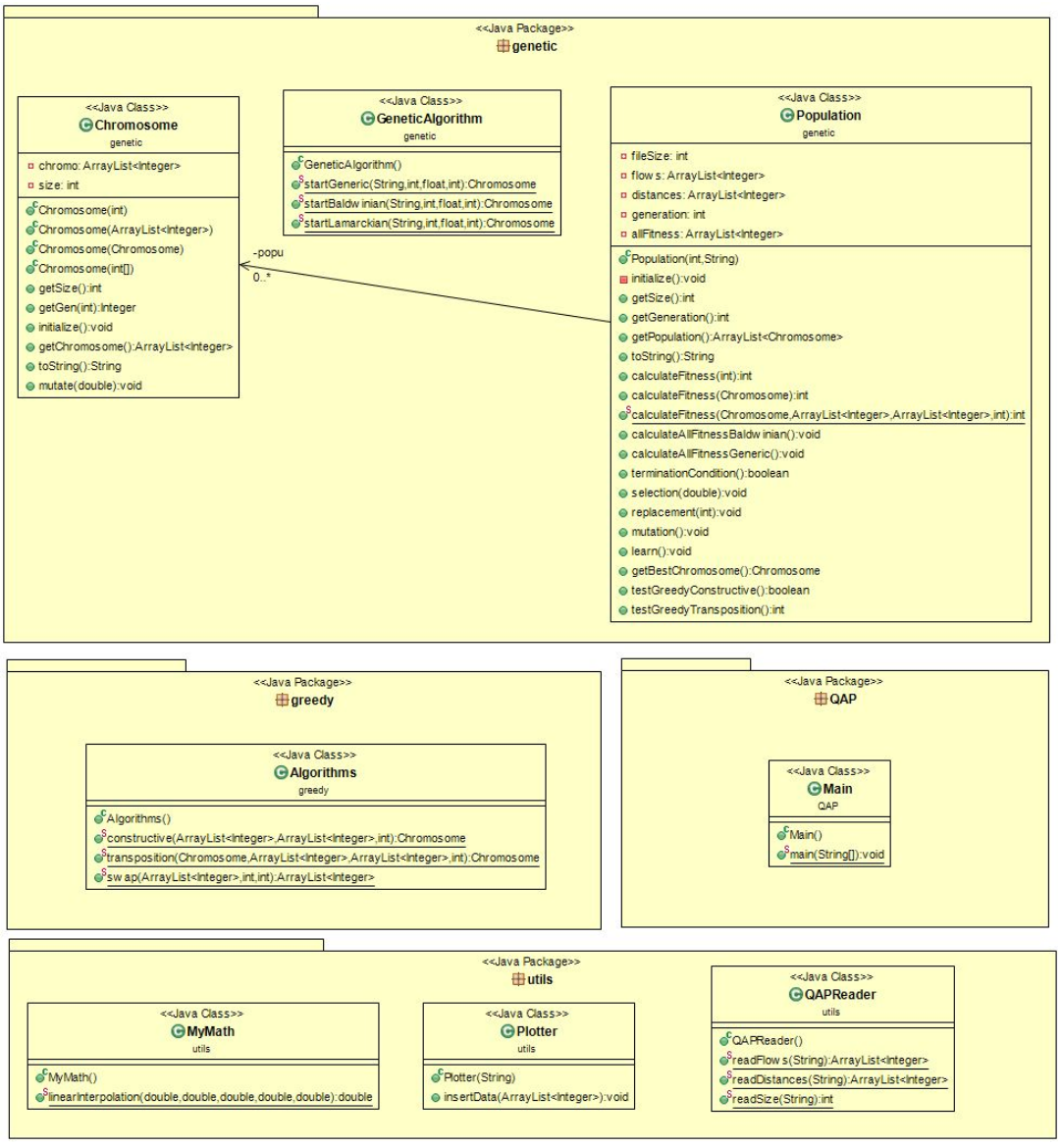
2. Introducción

Esta práctica consiste en solucionar el problema de asignación cuadrática (QAP) haciendo uso de algoritmos genéticos para encontrar una solución no óptima pero bastante buena en un tiempo reducido.

Para ello se ha creado un proyecto en java el cual realiza todo el proceso de selección, mutación y reemplazo del algoritmo genético. Además se han realizado dos variantes, una lamarckiana y otra baldwiniana y se han realizado comparaciones entre las dos variantes con la genérica y con un algoritmo greedy.

3. Proyecto en Java

La siguiente imagen muestra el diagrama de clases del proyecto.



3.1. Paquete genetic

En este paquete se encuentran todas las clases necesarias para representar los elementos de un algoritmo genético.

- **Chromosome:** Representa el cromosoma en el algoritmo genérico. Consiste en un ArrayList de enteros donde cada elemento representa un gen.
- **Population:** Representa la población del algoritmo genético. Contiene un ArrrayList de Chromosome. Esta clase contiene los métodos necesarios para evolucionarla como son selección, mutación y reemplazo.
- **GeneticAlgotihm:** Esta clase es la encargada de crear e inicializar una población y realizar todos los pasos del algoritmo genético mientras no se de la condición de terminación de la población.

3.2. Paquete greedy

Este paquete contiene los algoritmos greedy usados por el algoritmo genético para encontrar óptimos locales.

- **Algorithms:** Esta clase tiene dos métodos estáticos que son los algoritmos greedy constructivo y de transposición.

3.3. Paquete QAP

Paquete simple que contiene el main que se lanzará para probar el algoritmo genético.

3.4 Paquete utils

En este paquete se encuentran las clases necesarias para realizar acciones dentro del algoritmo genético pero que teóricamente no debería entrar dentro de él ya que no están relacionadas.

- **MyMath:** Solo se encuentra una función para realizar una interpolación lineal ya que las de otras librerías no se adaptaban muy bien a lo que yo quería hacer.
- **Plotter:** Clase encargada de representar gráficamente una función dados los puntos del eje x y los del eje y.
- **QAP:** Esta clase ha sido desarrollado para leer los archivos de datos que se aportan con la documentación de la práctica.

4. Algoritmos greedy

Antes de empezar a explicar el algoritmo genérico es necesario comprender qué es lo que hacen los algoritmos greedy y cómo funcionan. Los algoritmos greedy sirven para encontrar una solución aceptable en un tiempo muy bajo. Para esta práctica se han desarrollado dos algoritmos greedy para diferentes metas.

4.1. Algoritmo constructivo

Este algoritmo se encarga generar una cromosoma desde cero a partir de la matriz de distancias y la matriz de flujos. Es una variante del vecino más cercano que se usa típicamente para el Travelling Salesman Problem (TSP). El proceso es sencillo:

1. Se crea un array vacío con tantos elementos como el tamaño de las matrices.
2. Se genera una posición y una fábrica aleatoria y se introducen en el array el número de esa fábrica en la posición generada.
3. Se eliminan de las fábricas y posiciones restantes los elementos generados.
4. Para la última fábrica introducida en una posición, encontrar la posición más cercana a esa posición y la fábrica con la que hay más flujo entre los arrays de elementos restantes.
5. Introducir en la posición más cercana la fábrica con más flujo en el array.
6. Actualizar la última fábrica introducida y su posición y eliminarlas de los arrays de fábricas y posiciones restantes.
7. Repetir 4, 5 y 6 mientras queden elementos en los arrays de fábricas y posiciones restantes.

Con este algoritmo se obtiene una buena solución en muy poco tiempo pero tiene una pega. He comprobado que generando muchos cromosomas aleatoriamente siempre uno de ellos tiene una solución mejor que la encontrada por el algoritmo constructivo y en un tiempo menor. Debido a este motivo se ha decidido no usarlo para inicializar la población y se ha optado por inicializar los cromosomas aleatoriamente. Aunque no se use, este algoritmo se encuentra dentro del paquete greedy.

4.2. Algoritmo de transposición

Este algoritmo se diferencia del anterior en que parte de un cromosoma ya inicializado y se pretende encontrar el intercambio de genes que maximice el fitness del cromosoma. En este caso se ha usado un algoritmo 2-opt, es decir, solo se consideran 2 transposiciones. Me he basado en el pseudocódigo de la práctica para realizarlo y su proceso es el siguiente.

```
S = candidato inicial con coste c(S)

do {

  mejor = S

  for i=1..n
      for j=i+1..n
          T = S tras intercambiar i con j
          if c(T) < c(S)
             S = T

} while (S != mejor)
```

5. Algoritmo genético

En este apartado se va a explicar el funcionamiento del algoritmo genético desarrollado, así como sus variantes y los operadores usados. Para este problema se ha representado la disposición de unas fábricas en un array que es el cromosoma. Por ejemplo si tenemos 5 fábricas y 5 posiciones disponibles, el array [2,0,1,4,5] quiere decir que la fábrica 2 está en la posición 0, la fábrica 0 está en la posición 1 y así sucesivamente.

El fitness del cromosoma se evalúa usando la siguiente función donde *w* es la matriz de flujos, *d* es la matriz de distancias y *p* es la permutación de las fábricas.

$$\sum_{i,j} w(i,j)d(p(i),p(j))$$

Esta función devuelve un número, cuanto mayor sea ese número quiere decir que la solución tiene más coste. El objetivo es minimizar ese número y, para que se adapte con el objetivo de los algoritmos genéticos que es maximizar el fitness se va a usar la anterior función pero multiplicada por -1. De esta manera cuanto menor sea el número devuelto, al estar multiplicado por -1 será mayor y así cumplimos el objetivo de maximizar el fitness.

5.1. Operadores genéticos

Los operadores usados normalmente en un algoritmo genético son: inicialización, evaluación, selección, reproducción, cruce y mutación. En este apartado se explican cuáles se han usado y por qué.

- **Inicialización:** Se ha decidido inicializar los cromosomas aleatoriamente porque como se ha explicado, el algoritmo greedy constructivo no conseguía tan buenos resultados como una inicialización aleatoria y por lo tanto consumía un tiempo que podíamos ahorrarnos.

- **Evaluación:** La evaluación consiste en determinar todos los fitness de los cromosomas, quedarnos con el mejor cromosoma (mayor fitness) y si se da la condición de parada (en nuestro caso que la generación sea la 1000) se para el algoritmo y devolvemos el mejor cromosoma.

- **Selección:** Para este operador se ha optado por usar una interpolación lineal entre el peor y mejor fitness de los cromosomas y calcular la probabilidad de ser seleccionado. De esta manera, el peor cromosoma tiene una probabilidad del 10% de ser seleccionado y el mejor cromosoma tiene una probabilidad del 100% de ser seleccionado. Gracias a que el mejor cromosoma es siempre seleccionado usamos la técnica del **elitismo**, la cual consiste en quedarse siempre con el mejor cromosoma. Esta técnica se ha demostrado que es muy efectiva.

- **Reproducción:** De igual manera que en la selección, se calcula la probabilidad de ser copiado en la población en base al fitness del cromosoma. Esta vez la probabilidad de ser copiado del mejor cromosoma es del 70% ya que en la selección nos aseguramos de que está al menos una vez el mejor cromosoma.

- **Cruce:** Este operador no se ha usado en la práctica porque no he averiguado una manera de cruzar dos padres sin que haya errores tales como dos fábricas que se repiten en un cromosoma. Por otro lado un algoritmo genético tiene éxito con solo mutación, sin necesidad del cruce (aunque sea un operador muy útil).

- **Mutación:** La mutación de un cromosoma consiste en intercambiar de sitio pares de genes. La cantidad de intercambios que se realiza en el cromosoma es un número aleatorio entre 0 y la quinta parte del tamaño del cromosoma. Esos intercambios se realizarán según una probabilidad establecida según el fitness del cromosoma.

5.2. Variantes

Se han usado tres variantes para el algoritmo genético, cada una con sus ventajas e inconvenientes. Como el algoritmo genético es diferente para cada variante se van a explicar por separado tanto su objetivo principal como el procedimiento del algoritmo.

5.2.1. Algoritmo genético genérico

Este algoritmo sigue los pasos típicos de los algoritmos genéricos sin hacer uso de algoritmos greedy para encontrar óptimos locales. A continuación se muestra el código en java que realiza este algoritmo.

```
//Initial population
Population p = new Population(population_size, data );
//While not termination
while(!p.terminationCondition()) {
    //Calculate all the Fitness of the population
    p.calculateAllFitnessGeneric();
    System.out.println("GENERATION " + p.getGeneration() + "/1000  --------------------
    //Population = Select population
    p.selection(selection_percentage);
    //Population = Mutate population
    p.mutation();
    //Calculate again all the fitness for the mutated chromosomes
    p.calculateAllFitnessGeneric();
    //Population = replace population
    p.replacement(replacement_size);
    //Evaluate population
    best = p.getBestChromosome();
    //Add the best fitness to the array
    generatedFitness.add(-p.calculateFitness(best));
    System.out.println("Best fitness = " + p.calculateFitness(best) );

}
```

El procedimiento es el siguiente:

- Se crea una población inicial de tamaño dado por el parámetro *population_size* e inicializada por los datos indicados en el parámetro *data*.
- Mientras no se de la condición de terminación (1000 generaciones)...
- Calculamos todos los fitness de los cromosomas y los guardamos en un array.
- Hacemos una selección para quedarnos con el porcentaje de cromosomas indicado en el parámetro *selection_percentage*.
- Mutamos los cromosomas en base a su fitness.
- Volvemos a calcular otra vez el fitness ya que los cromosomas han mutado.
- En base a los nuevos fitness realizamos el reemplazo de cromosomas hasta alcanzar el número de cromosomas indicado en *replacement_size*.
- Actualizamos el mejor cromosoma de esta generación.
- Almacenamos su fitnes para posteriormente ver cómo han evolucionado

5.2.2. Algoritmo genético Lamarckiano.

Este algoritmo es igual que el anterior pero sigue la filosofía de Lamarck que defendía que lo que aprendían los padres era transmitido a los hijos. El aprendizaje de los cromosomas se hace mediante el algoritmo greedy de transposición. El código es muy parecido al anterior:

```
//Initial population
Population p = new Population(population_size, data );

//While not termination
while(!p.terminationCondition()) {
    //Use a greedy algorithm to improve the genes
    p.learn();
    //Calculate all the fitness of the population
    p.calculateAllFitnessGeneric();
    System.out.println("GENERATION " + p.getGeneration() + "/1000  -----------
    //Population = Select population
    p.selection(selection_percentage);
    //Population = Mutate population
    p.mutation();
    //Calculate again all the fitness of the population
    p.calculateAllFitnessGeneric();
    //Population = replace population
    p.replacement(replacement_size);
    //Evaluate population
    best = p.getBestChromosome();
    //Add the best fitness to the array
    generatedFitness.add(-p.calculateFitness(best));
    System.out.println("Best fitness = " + p.calculateFitness(best) );

}
```

La única diferencia reside en la línea *p.learn();* donde se llama al método *learn* el cual mediante el algoritmo greedy de transposición altera los cromosomas a una versión mejorada de ellos donde se ha encontrado un óptimo local. El procedimiento prosigue con normalidad ya que se pretende transmitir lo aprendido a los hijos.

5.2.3. Algoritmo genético Baldwiniano

En esta variante, el fitness de los individuos se calcula en base a “lo que podrían llegar a ser”, es decir, se calcula el fitness al cromosoma una vez se le ha aplicado el algoritmo greedy de transposición. La diferencia con el Lamarckiano es que si bien usamos el fitness del cromosoma “mejorado”, el cromosoma permanece igual. A continuación se muestra el código de esta variante:

```
//Initial population
Population p = new Population(population_size, data );

//While not termination
while(!p.terminationCondition()) {
    //Calculate all the fitness of the population using the local optimum of the genes
    p.calculateAllFitnessBaldwinian();
    System.out.println("GENERATION " + p.getGeneration() + "/1000  ----------------------------
    //Population = Select population
    p.selection(selection_percentage);
    //Population = Mutate population
    p.mutation();
    //Calculate again all the fitness of the population using the local optimum of the genes
    p.calculateAllFitnessBaldwinian();
    //Population = replace population
    p.replacement(replacement_size);
    //Evaluate population
    best = p.getBestChromosome();
    //Add the best fitness to the array
    generatedFitness.add(-p.calculateFitness(best));
    System.out.println("Best fitness = " + p.calculateFitness(best) );

}
```

Ahora no hay aprendizaje y se usa otro método para calcular el fitness de los cromosomas el cual usa el algoritmo de transposición pero deja los cromosomas originales intactos.

5.3. Parámetros seleccionados

En este apartado se van a explicar los parámetros usados en el algoritmo genético.

- **Probabilidad de ser mutado:** Interpolación lineal entre el menor y mayor fitness siendo para el menor probabilidad 1 y para el mayor probabilidad 0.1.
- **Probabilidad de ser seleccionado:** Interpolación lineal entre el menor y mayor fitness siendo para el menor probabilidad 0.1 y para el mayor probabilidad 1. De esta manera nos aseguramos el elitismo.
- **Probabilidad de ser copiado:** Interpolación lineal entre el menor y mayor fitness siendo para el menor probabilidad 0.1 y para el mayor probabilidad 0.7.
- **Población inicial:** 100 cromosomas.
- **Población tras selección:** el 50% de 100, es decir 50 cromosomas.
- **Población tras reemplazo:** 100 cromosomas.
- **Condición de parada:** 1000 generaciones.

6. Resultados

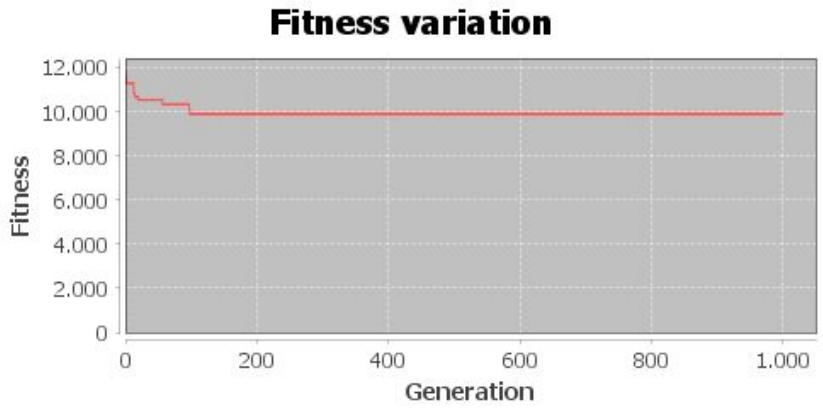
En este apartado se va a mostrar una comparación de resultados de los algoritmos desarrollados para algunos problemas. Hay algunos algoritmos que eran demasiado lentos como para medirlos por lo que se han usado solo para tamaños de problema pequeños.

En la siguiente tabla se muestra una comparación de los pesos obtenidos para cada problema. Hay algunas casillas que están en blanco debido a que duraba demasiado el algoritmo.

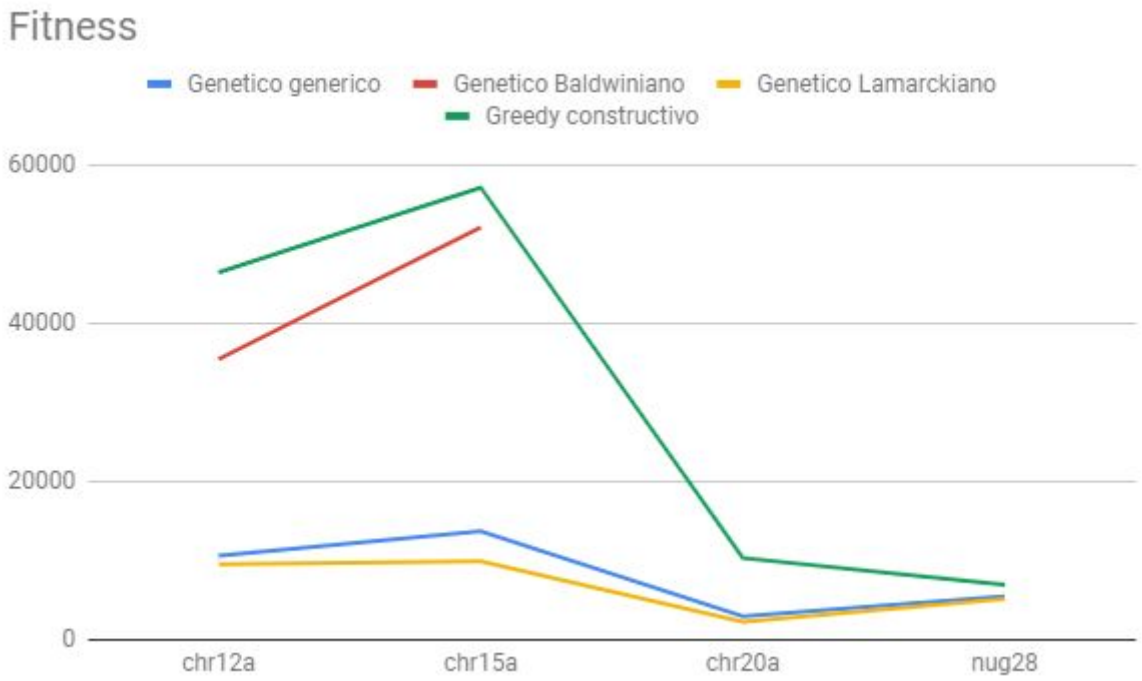
PESO	Genetico generico	Genetico Baldwiniano	Genetico Lamarckiano	Greedy constructivo
chr12a	10624	35450	9552	46392
chr15a	13724	52098	9896	57100
chr20a	2976		2244	10316
nug28	5482		5166	6934
tai100a	23091980			24133164
tai256c	46665220			57480638

Se puede observar en la tabla que los mejores resultados obtenidos han sido por el algoritmo Lamarckiano. Por otro lado el algoritmo greedy encuentra soluciones peores ya que como he comentado, en una simple inicialización aleatoria de cromosomas aparecía siempre una solución mejor.

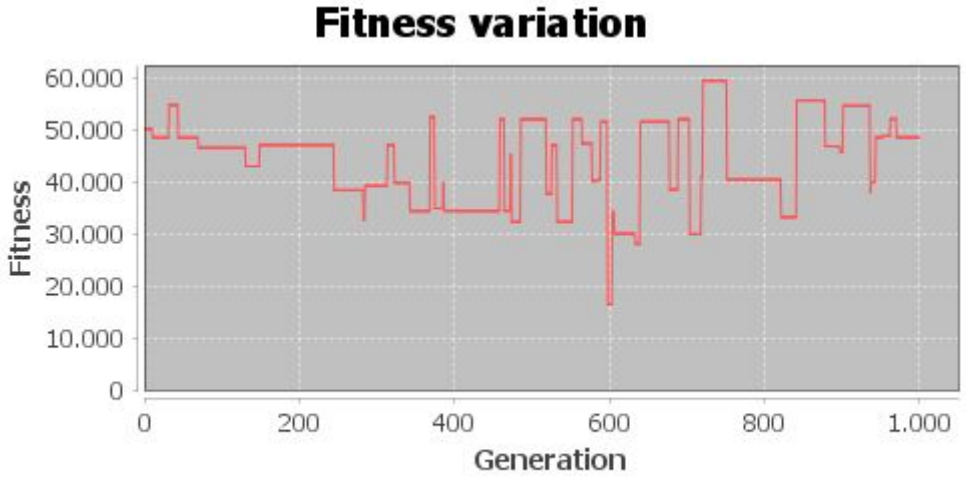
He observado que el algoritmo Lamarckiano encontraba un óptimo en generaciones tempranas y luego no mejoraba más. En la siguiente gráfica se puede ver la evolución del peso para el problema de chr15a resuelto por el algoritmo Lamarckiano.



La siguiente gráfica muestra una comparación de los pesos obtenidos por los algoritmos donde se vuelve a apreciar que el Lamarckiano es el que obtiene mejores resultados.



He llegado a la conclusión de que el algoritmo Baldwiniano encontraba soluciones muy malas ya que se usaba un fitness que no tenía que ver con el cromosoma original. Esto se debe a que se calcula el fitness en base al algoritmo de transposición pero luego si el cromosoma mutaba lo hacía a una permutación totalmente distinta a la calculada por el algoritmo de transposición. En la siguiente gráfica se muestra la variación del peso según la generación para el problema chr12a resuelto por el algoritmo Baldwiniano.

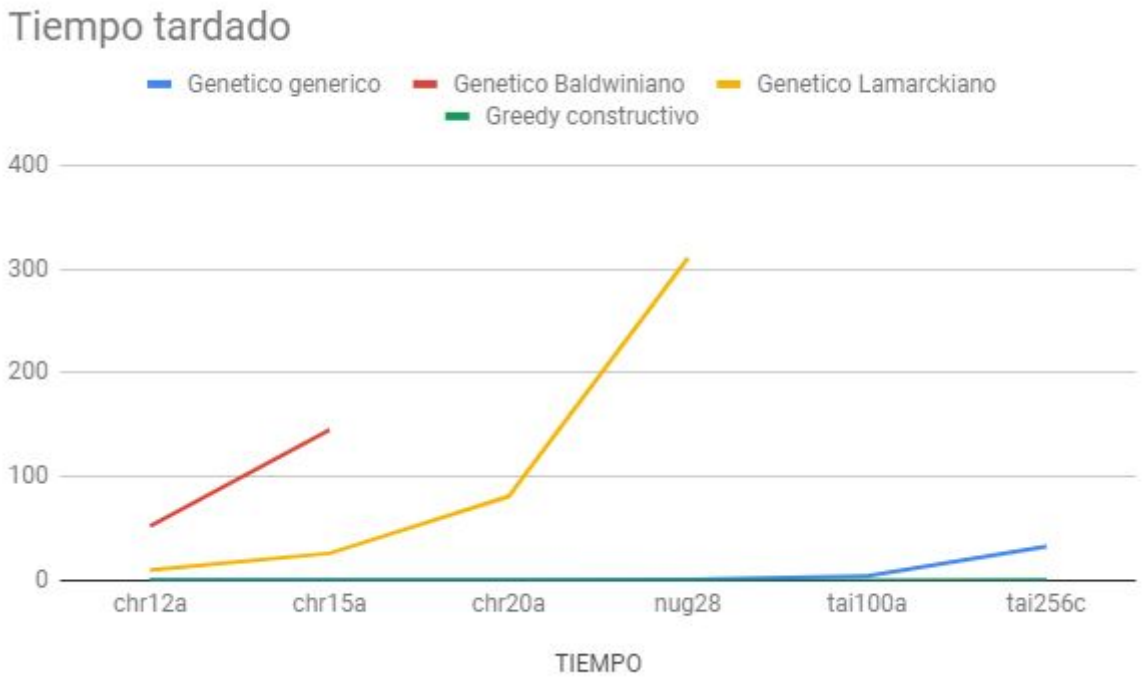


La oscilación del peso se debe a lo que he comentado antes de que el fitness no se corresponde con el genoma real.

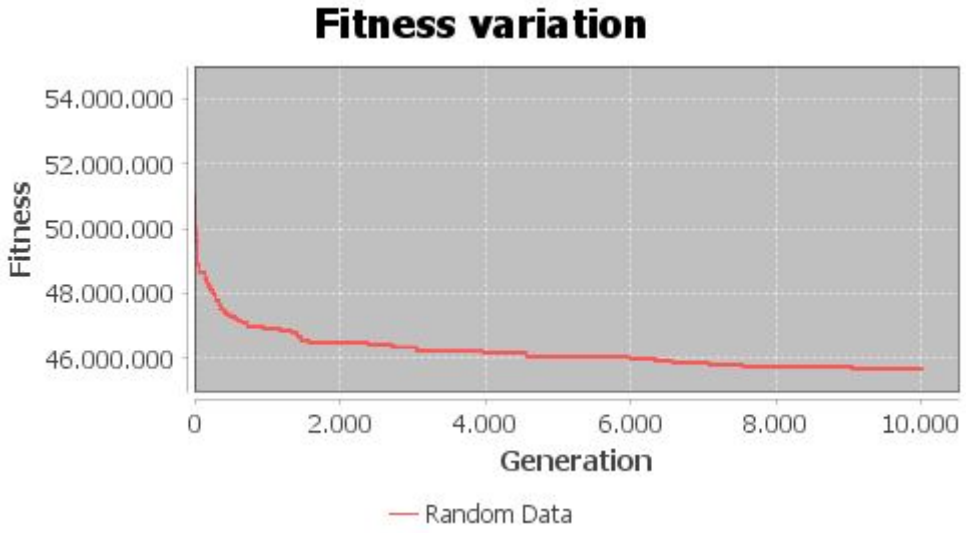
En cuanto al tiempo, en la siguiente tabla se muestra el tiempo tardado (en segundos) por cada algoritmo en resolver distintos tamaños de problemas. De nuevo, las casillas en blanco significa que el problema tardaba demasiado en resolverse.

TIEMPO	Genetico generico	Genetico Baldwiniano	Genetico Lamarckiano	Greedy constructivo
chr12a	0,302942165	52	10,21005068	0,009876429
chr15a	0,353916761	144,9734526	26,21778871	0,010332769
chr20a	0,462224306		80,92989666	0,011930745
nug28	0,793609118		310,6203656	0,029360063
tai100a	4,435922033			0,108183592
tai256c	33			0,318260321

Los mejores tiempos han sido obtenidos por el algoritmo greedy mientras que los peores se han obtenido con el algoritmo baldwiniano. En la siguiente gráfica se aprecia mejor la diferencia.



Para acabar el apartado de resultados se muestra la evolución del peso encontrado por el algoritmo genético genérico para el problema de mayor tamaño (tai256c).



7. Conclusión

En vista a los resultados obtenidos se puede decir que el algoritmo genético genérico funciona correctamente y es viable para problemas de gran tamaño debido a que tiene un tiempo de cómputo aceptable y encuentra mejores soluciones que el algoritmo greedy.

La variante Lamarckiana ha demostrado ser la mejor encontrando soluciones pero para problemas de gran tamaño tarda mucho tiempo. Por otro lado la variante Baldwiniana es la peor tanto en tiempo de ejecución como en calidad de la solución encontrada.