Práctica 3

Programación Evolutiva

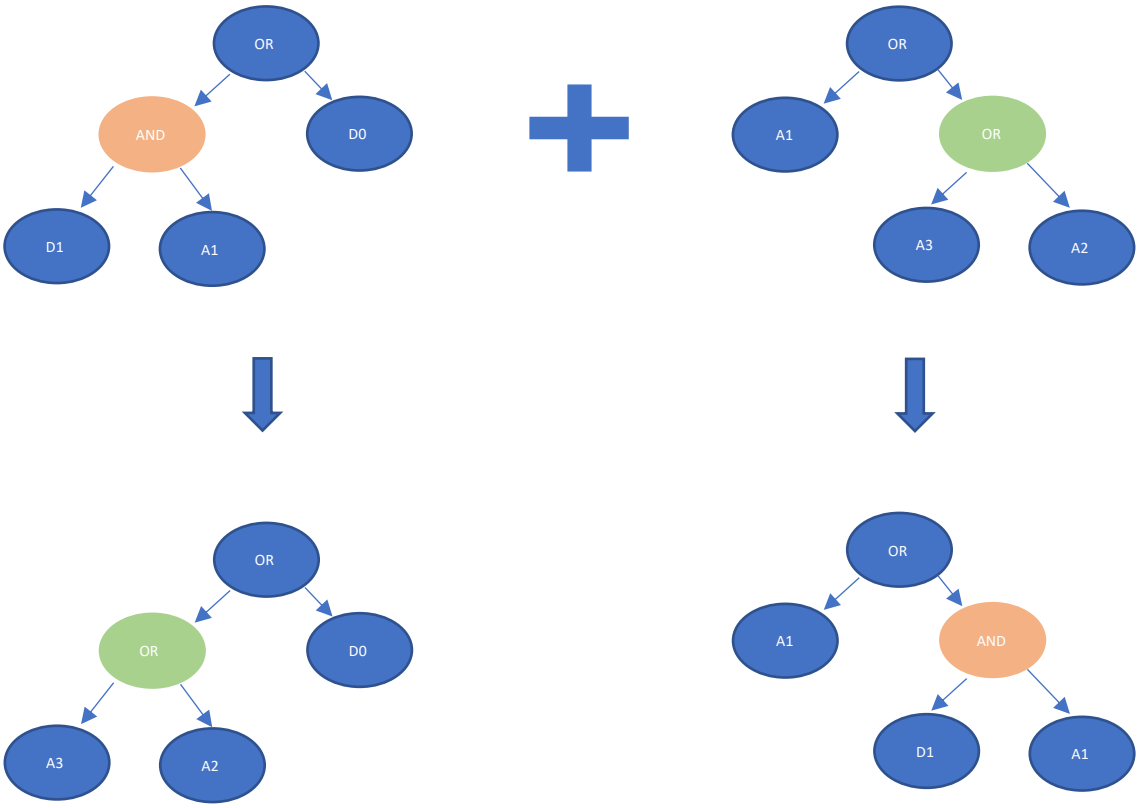
Borja Aday Guadalupe Luis
Valerio Moroni
Grupo 19

Índice

1. Método de cruce.

1.1. Cruce intercambio de subárboles.

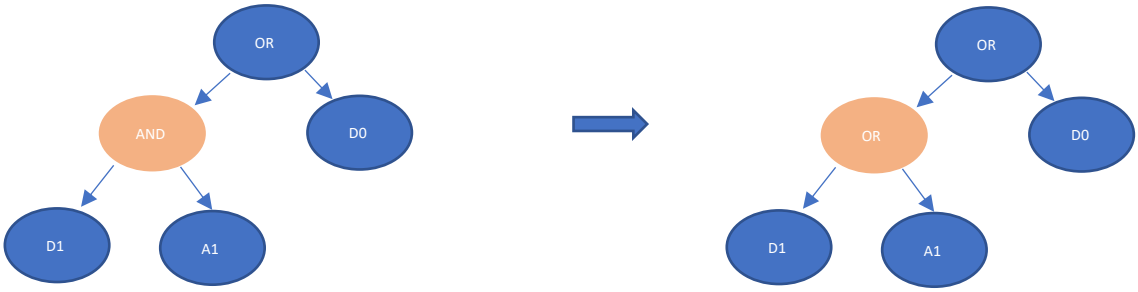
Este es el único cruce implementado en el algoritmo puesto que es el único que aparece en la teoría y además los árboles no presentan una estructura cómoda para implementar diferentes tipos de cruce. Esto se debe a que deben mantener una estructura concreta en la que los nodos funciones deben tener un número concreto de hijos. De este modo, si realizamos un cruce que transforme el árbol sin un control específico, llevaría a tener que corregir el árbol entero, y eso supone un aumento de tiempo más que considerable para cada ejecución del algoritmo.



2. Métodos de mutación.

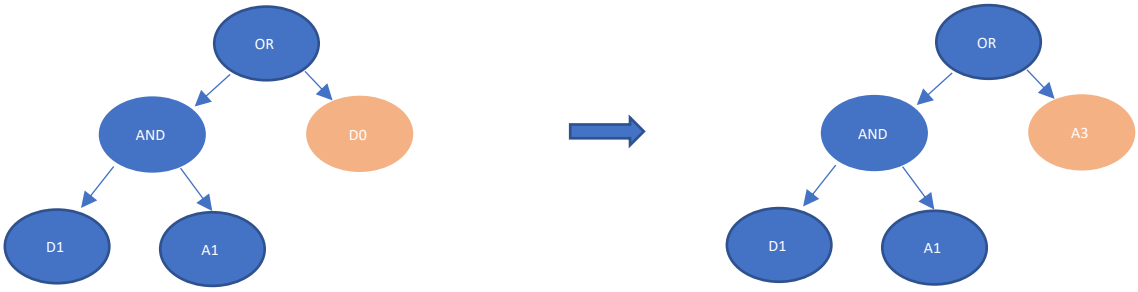
2.1. Mutación funcional.

La mutación funcional consiste en cambiar un nodo función por otro nodo función de la misma cardinalidad. En este caso las dos únicas funciones que tienen la misma cardinalidad es una AND o una OR.



2.2. Mutación terminal.

La mutación terminal consiste en intercambiar un nodo terminal por otro. Sin ninguna restricción.

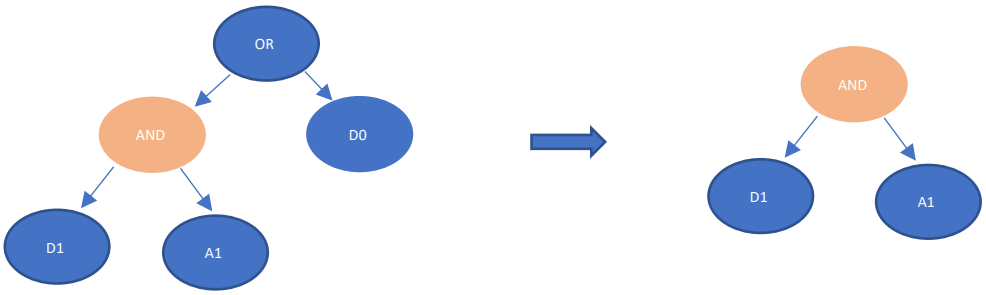


2.3. Mutación por permutación.

La mutación por permutación consiste en intercambiar dos nodos hijos. Esto solo se puede realizar en los nodos función puesto que los terminales no tienen hijos. Además, solo se realiza en los nodos cuyo valor es IF, ya que intercambiar nodos terminales en una AND o una OR resulta en un árbol equivalente al inicial.

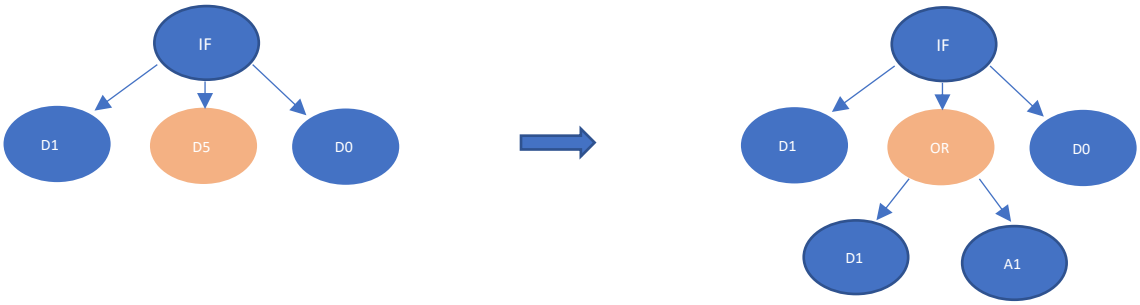
2.4. Mutación Hoist.

La mutación Hoist se basa en mutar un árbol a cualquier subárbol que lo compone. Dado que nos estamos quedando con una parte concreta del árbol inicial, se reduce el número de nodos. Esto ayuda al control del bloating, que se nota en la media de nodos alcanzada al final de la ejecución y en el tiempo de ejecución.



2.5. Mutación por expansión.

La mutación por expansión se basa en coger un nodo terminal y expandirlo convirtiéndolo en un nodo función. De esta forma aumenta el tamaño del árbol. Cabe destacar que con esta mutación se obtienen individuos bastante grandes y el tiempo de ejecución aumenta considerablemente. Para poder ejecutar el algoritmo usando esta mutación, se recomienda aplicar un control de bloating para que el tiempo de ejecución no sea excesivo.

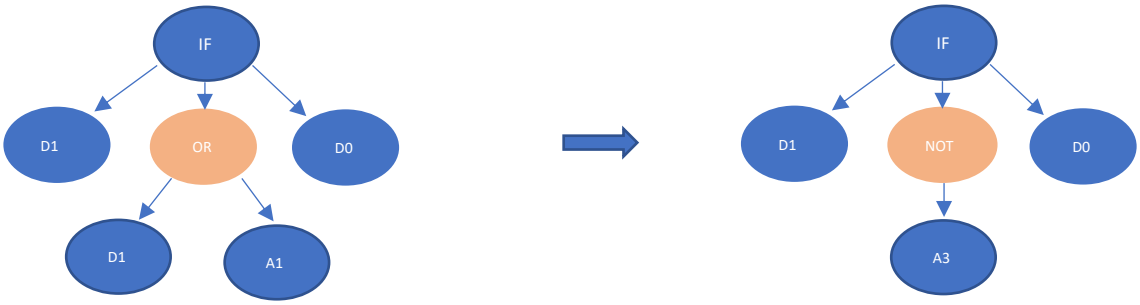


2.6. Mutación por contracción.

La mutación por contracción se basa en coger un nodo funcional y convertirlo en un nodo terminal.

2.7. Mutación por regeneración de subárbol.

Esta mutación es muy simple y se basa en regenerar un subárbol del árbol general. Esto ayuda bastante a la convergencia pues aporta mucha variedad a la población.



3. Corrección de intrones.

3.1. Lógica de la corrección de intrones.

La corrección de intrones es un método añadido a la práctica para ayudar al control de bloating y evitar que los individuos sean excesivamente grandes y que no se pueda comprender el resultado final. Esta corrección se lleva a cabo gracias a 3 funciones diferentes:

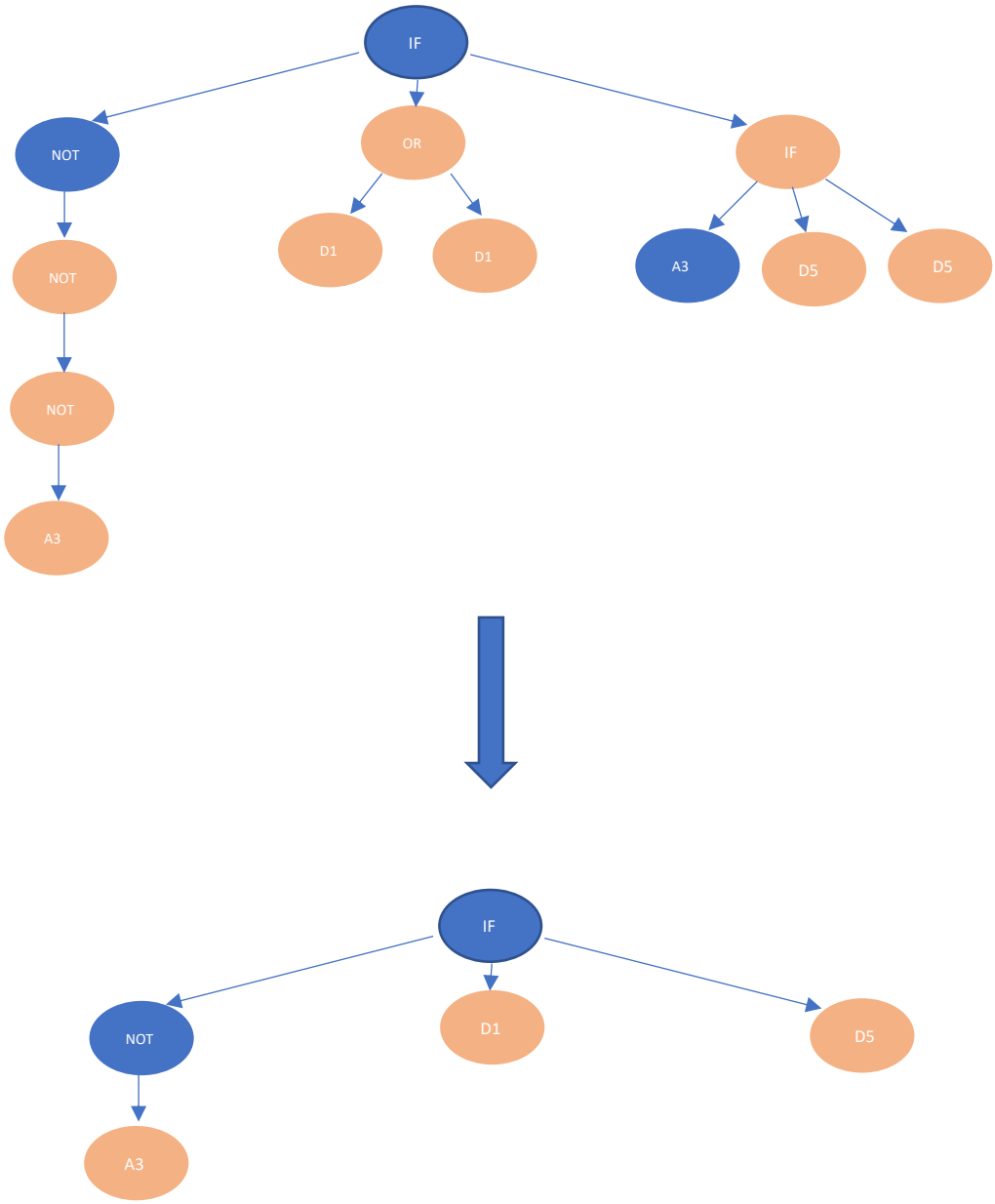
- **Función equals**: Te comprueba si un árbol es igual a otro. Este método se usa para comprobar qué subárboles componen un intrón. Es totalmente necesaria porque un intrón no solo se da cuando sus hijos son dos terminales iguales ((AND(D1 D1)) = D1), sino también cuando son dos nodos funciones iguales ((AND(AND(D1 D1) AND(D1 D1)) = D1).

- **Función corrige intrones**: Va recorriendo el árbol comprobando los intrones existentes utilizando la función equals. Los intrones que corrige en concreto son:

    - Las dobles negaciones.
    - Los nodos AND u OR con dos hijos iguales.
    - Los nodos IF cuyas dos opciones son iguales.

- **Función corrigeNumNodos**: Esta función se ejecuta tras corregir los intrones, pues el número de nodos de cada subárbol cambia, de este modo terminamos con un árbol completamente válido y equivalente al inicial para seguir ejecutando el algoritmo.

3.2. Ejemplo corrección de intrones.



4. Método de evaluación.

4.1. Forma de evaluar.

La parte de la evaluación de los individuos es prácticamente la parte más costosa en tiempo y recursos de todo el programa. Para intentar evitar estos costes hemos implementado dos cosas diferentes:

- Para conseguir que la evaluación de un individuo tuviera un coste O(n), era necesario disponer primero de la matriz de casos junto al array de soluciones. De forma que evaluar un individuo consistiera en evaluar el árbol con un caso dado concreto y si la solución era la esperada, se sumaba un acierto, que finalmente sería el fitness.

  Para alcanzar este objetivo, a la hora de lanzar el algoritmo, se genera el array de casos y soluciones para el problema solicitado. De este modo se calcula una sola vez y se le pasa una referencia de estas estructuras a cada individuo que creamos. Con esto conseguimos una mejora considerable en tiempo pues no tenemos que calcular el caso ni la solución esperada para cada uno de los individuos ya que está previamente calculado.

- Además, hemos implementado una opción de evaluación mediante hebras que es muy útil para el multiplexor de 3 entradas de control. Se basa en crear la cantidad de hebras que deseamos crear para la evaluación y la hora de evaluar un individuo, el trabajo de evaluar los 2048 casos posible se divide en esa cantidad de hebras. Con esto conseguimos un tiempo de ejecución más estable.

  No se recomienda utilizar esta opción en caso de:

  - Disponer de un ordenador con un procesador sin núcleos, incapaz de gestionar las hebras.
  - Tener un ordenador saturado en el que el porcentaje de uso de la CPU sea elevado.
  - Cuando se selecciona el problema del multiplexor de dos entradas o si se aplica algún método de control de bloating. Al aplicar un método de control de bloating, comprobaremos que el número medio de nodos no es excesivo, por lo que el coste de crear las hebras y lanzarlas al procesador supone un mayor tiempo a la ejecución sin concurrencia.

Estas son las mejoras que hemos aplicado a la evaluación de cada individuo.
Dicho esto, cabe destacar que el fitness se basa únicamente en la cantidad de aciertos que tiene el individuo concreto en la tabla de verdad del multiplexor pedido.

4.2. Control de bloating.

El hecho de utilizar únicamente directamente la cantidad de aciertos en la tabla de verdad como fitness nos lleva a un problema grave y es el descontrol en el tamaño de los árboles generados tras cada generación. Dado que los árboles se cruzan y mutan en cada generación, cada vez tendremos árboles más grandes que no necesariamente mejoran la cantidad de aciertos de los pequeños.

Para evitar este grave problema hemos aplicado dos métodos de control de bloating basados en la penalización de los individuos grandes:

- Método de Tarpeian: Se basa en penalizar a los árboles cuya cantidad de nodos supere la media de nodos general. De este modo permitimos que los árboles crezcan, pero hacemos presión para que esto no ocurra. En caso de que un árbol con más nodos que la media mejore considerablemente la cantidad de aciertos, tanto que aun penalizado sea mejor que el resto, este será seleccionado y la media de nodos de la población general aumentará.

  En este caso, tras realizar distintas pruebas hemos penalizado de una forma muy concreta, sigue la siguiente formula:

$$Fitness = aciertos - (\alpha * \ln(\frac{\max(1,\ numNodosExtra)}{3}))$$

  Donde α define la limitación que se decide aplicar. Este, además, es un parámetro modificable desde la interfaz, que permite ver cómo cambia la media de nodos en función de la limitación que se aplique.

  Hemos decido aplicar esta fórmula precisamente por seguir un progreso logarítmico, penaliza más a quien más nodos tenga, pero cada vez menos. De forma que los árboles con bastantes nodos más que la media, no se vean afectados hasta el punto de tener 0 posibilidades de pasar de generación.

  Lo hemos dividido entre 3 porque hemos comprobado que sin la división era demasiada penalización y nuestros árboles a penas crecían quedándose en una media entre 7 y 9 nodos.

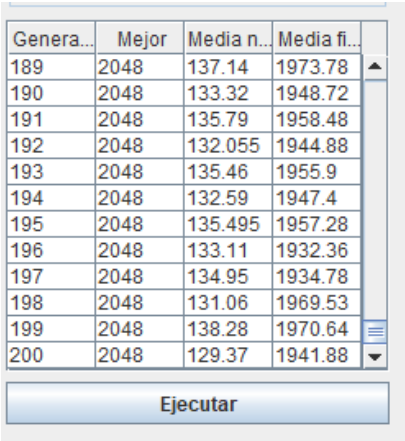
  Se recomienda utilizar un α = 5%, mejora considerablemente el número de nodos medio y con ello el tiempo de ejecución, pero dejando crecer lo justo para alcanzar las soluciones y permitiendo que éstas sean legibles.

- Método de penalización bien fundamentada: En este caso, se sigue otra fórmula que no depende de ningún α por lo que no se puede comprobar su efectividad en función de un parámetro, pero se puede comparar con la media de nodos al quitar el límite y se ve claramente la reducción de la media de nodos. En este caso sigue la fórmula:

$$Fitness = aciertos - \left(\frac{Cov(aciertos, numNodos)}{Varianza(numNodos)} * numNodos\right)$$

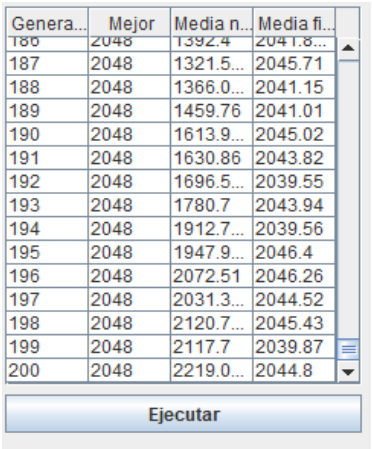
En este caso, la penalización se va calculando en cada generación, por lo que se calcula de forma dinámica y se ajusta mejor a las necesidades de la población en cada generación concreta.

Ahora veremos un ejemplo claro del control de bloating, en ambos casos se ha utilizado los mismos parámetros en el algoritmo y se ha alcanzado el 2048 en aciertos, pero se diferencian claramente en la media de nodos como podemos observar en las tablas (mientras que aplicando el control de bloating no supera los 129 nodos de media, sin control de bloating alcanza los 2219 nodos de media).



Genera...	Mejor	Media n...	Media fi...
189	2048	137.14	1973.78
190	2048	133.32	1948.72
191	2048	135.79	1958.48
192	2048	132.055	1944.88
193	2048	135.46	1955.9
194	2048	132.59	1947.4
195	2048	135.495	1957.28
196	2048	133.11	1932.36
197	2048	134.95	1934.78
198	2048	131.06	1969.53
199	2048	138.28	1970.64
200	2048	129.37	1941.88

Ejecutar



Genera...	Mejor	Media n...	Media fi...
186	2048	1392.4	2041.8...
187	2048	1321.5...	2045.71
188	2048	1366.0...	2041.15
189	2048	1459.76	2041.01
190	2048	1613.9...	2045.02
191	2048	1630.86	2043.82
192	2048	1696.5...	2039.55
193	2048	1780.7	2043.94
194	2048	1912.7...	2039.56
195	2048	1947.9...	2046.4
196	2048	2072.51	2046.26
197	2048	2031.3...	2044.52
198	2048	2120.7...	2045.43
199	2048	2117.7	2039.87
200	2048	2219.0...	2044.8

Ejecutar

5. Resultados y comparaciones con Heuristic Lab.

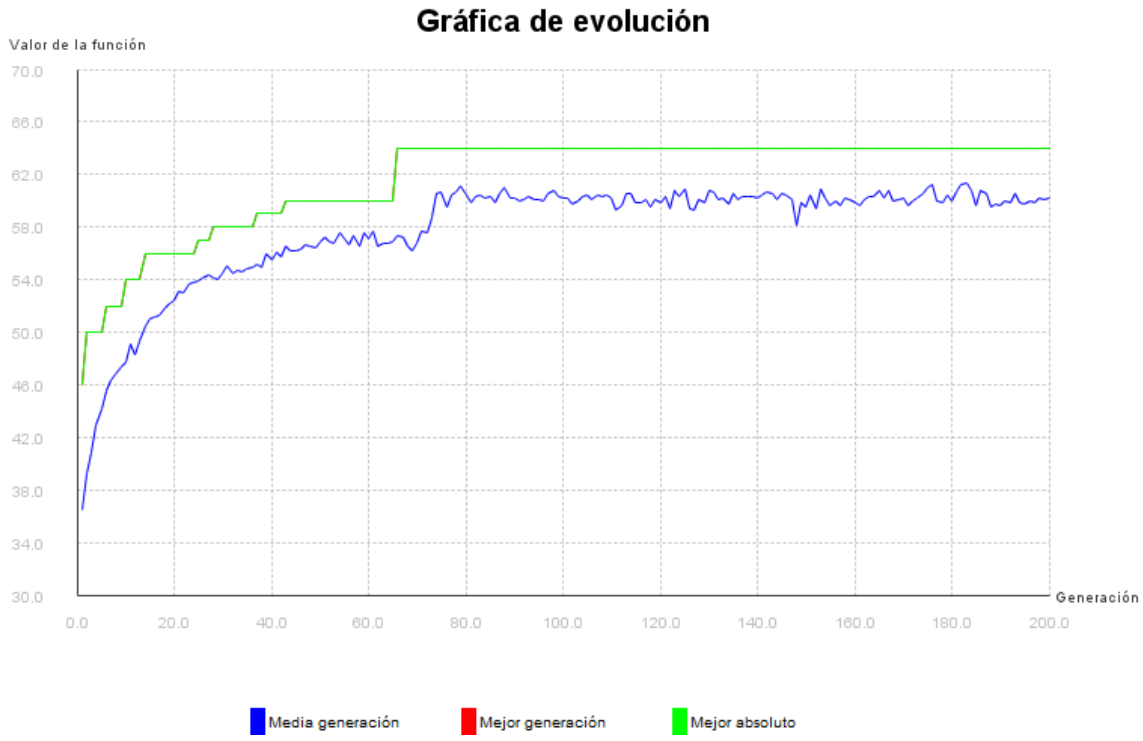
En cuanto a los resultados, cabe destacar que no todos los métodos de mutaciones, selecciones y parámetros modificables aportan los mismos resultados. En nuestro caso hemos identificado que lo que mejor funciona es aplicar la selección de torneo determinista con un 1% de élite y aplicando Tarpeian. Suele dar resultados bastante buenos, en el caso del multiplexor de 2 entradas encuentra la solución una media de 6-7 veces de cada 10. En el caso del multiplexor de 3 entradas, no es tan fácil encontrar la solución, pero es capaz de encontrarla tras varias ejecuciones.

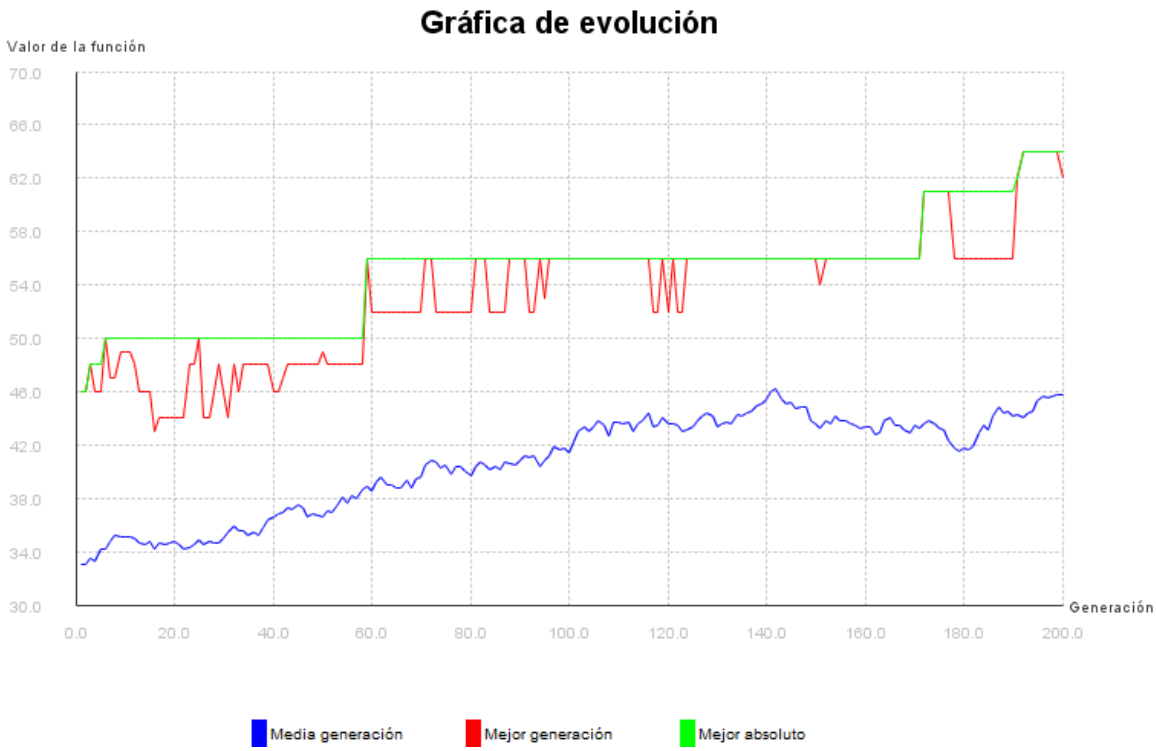
Otro aspecto importante es la mejora que se produce en el algoritmo aplicando elitismo, es bastante impresionante como acelera la convergencia, haciéndose muy difícil encontrar la solución sin elitismo, y con ella resultando bastante fácil o al menos quedándose siempre cerca de la solución perfecta. Las siguientes gráficas obtenidas son las mejores ejecuciones que hemos conseguido.

5.1. Problema del multiplexor de 2 entradas.

Este problema es bastante más fácil de resolver que el problema del multiplexor de 3 entradas, esto se debe a que la tabla de verdad tiene un tamaño inferior (tan solo 64 casos, frente a los 2048 del otro problema).

Dado que este problema es más sencillo, solo aportaremos dos gráficas en las que se alcanza la solución perfecta, la primera obtenida con la selección de torneo determinista aplicando elitismo y la segunda con la selección por ruleta sin elitismo. Ambas con el control de bloating de Tarpeian.

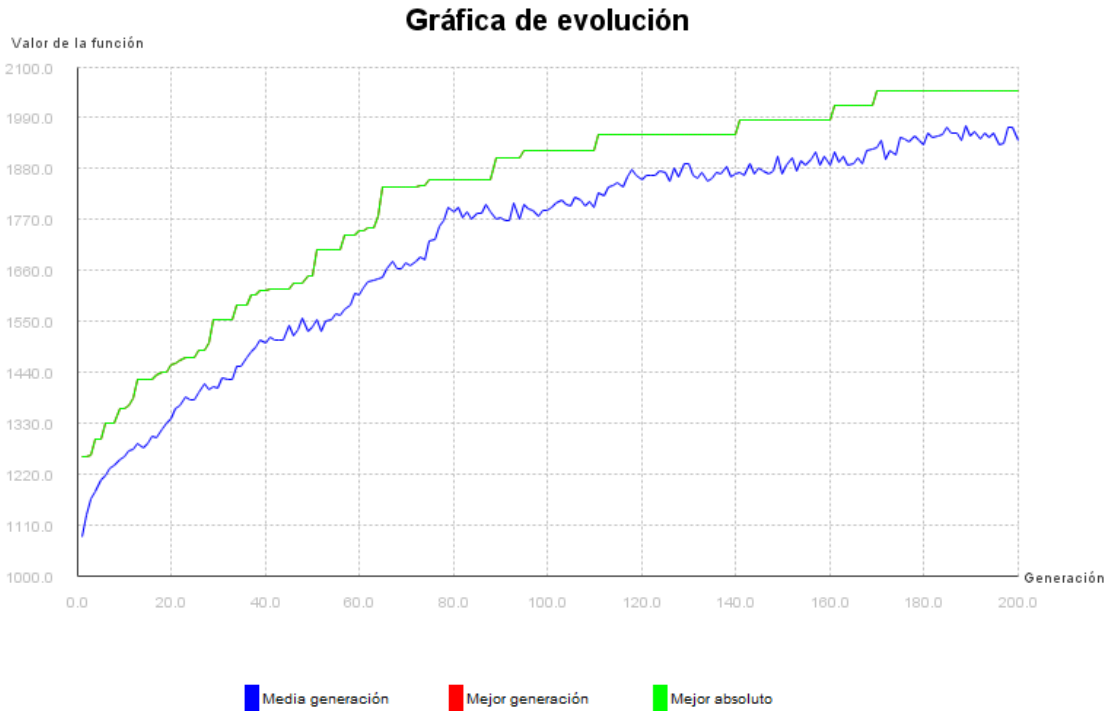


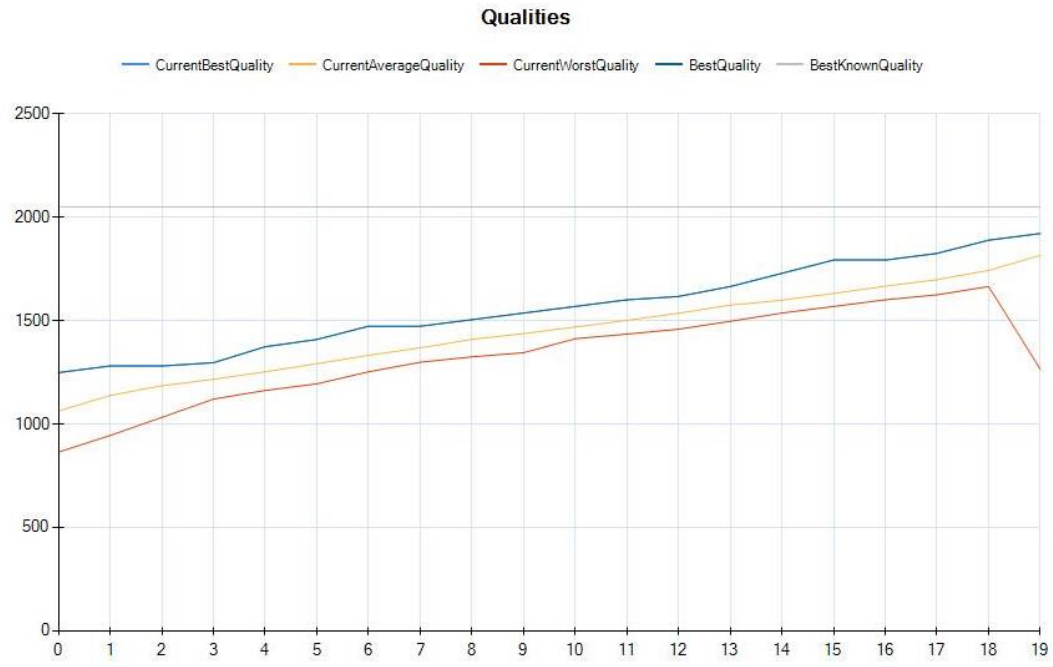


5.2. Problema del multiplexor de 3 entradas.

Este problema se hace mucho más interesante debido a su complejidad y comienza a ser un problema digno de la programación evolutiva. Encontrar la fórmula del multiplexor de 2 entradas, entra más en la parte de algoritmia de solución voraz pues su coste en tiempo no es muy elevado, cuando aumentamos el número de entradas, la programación evolutiva entra en juego para encontrar una solución o algo cercano a ella en un tiempo mucho mas asequible que si nos planteáramos una solución algorítmica polinomial.

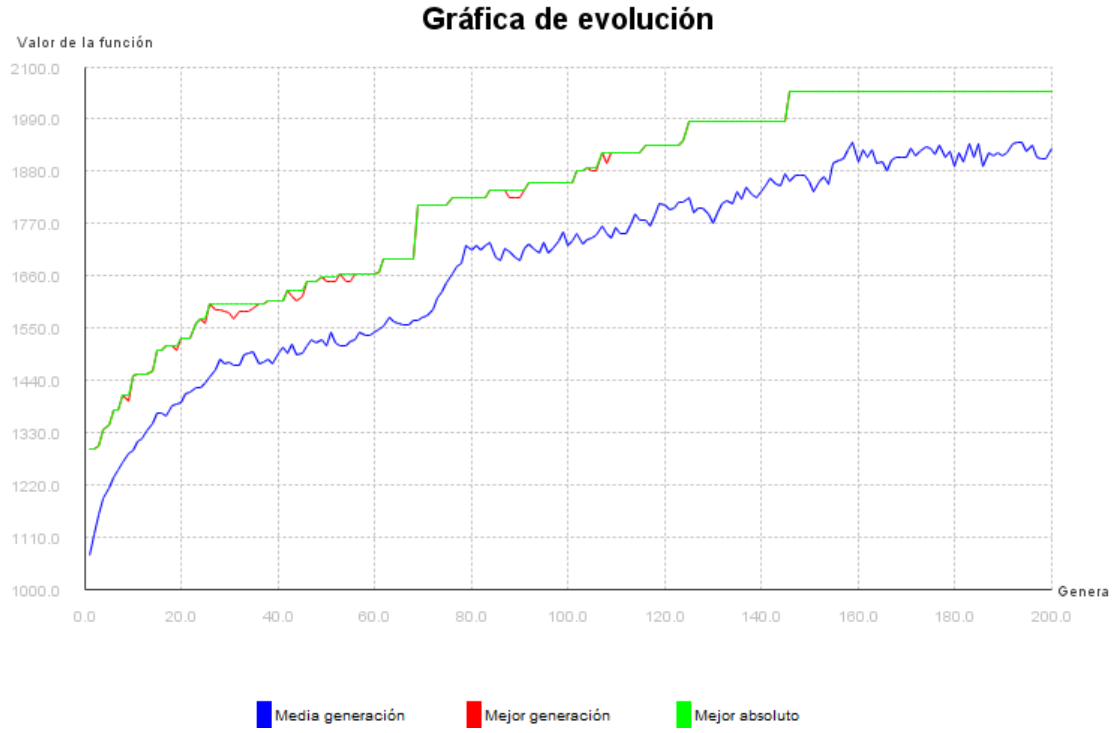
Las ejecuciones que mostramos a continuación son ejecuciones realizadas de la misma forma en nuestro programa y en Heuristic Lab. Se ha aplicado selección por torneo con un elitismo del 1%. En ambos casos se ha usado una población de 200 individuos y un máximo de generaciones de 200. En nuestro programa se ha utilizado el método de Tarpeian para el control del número de nodos.

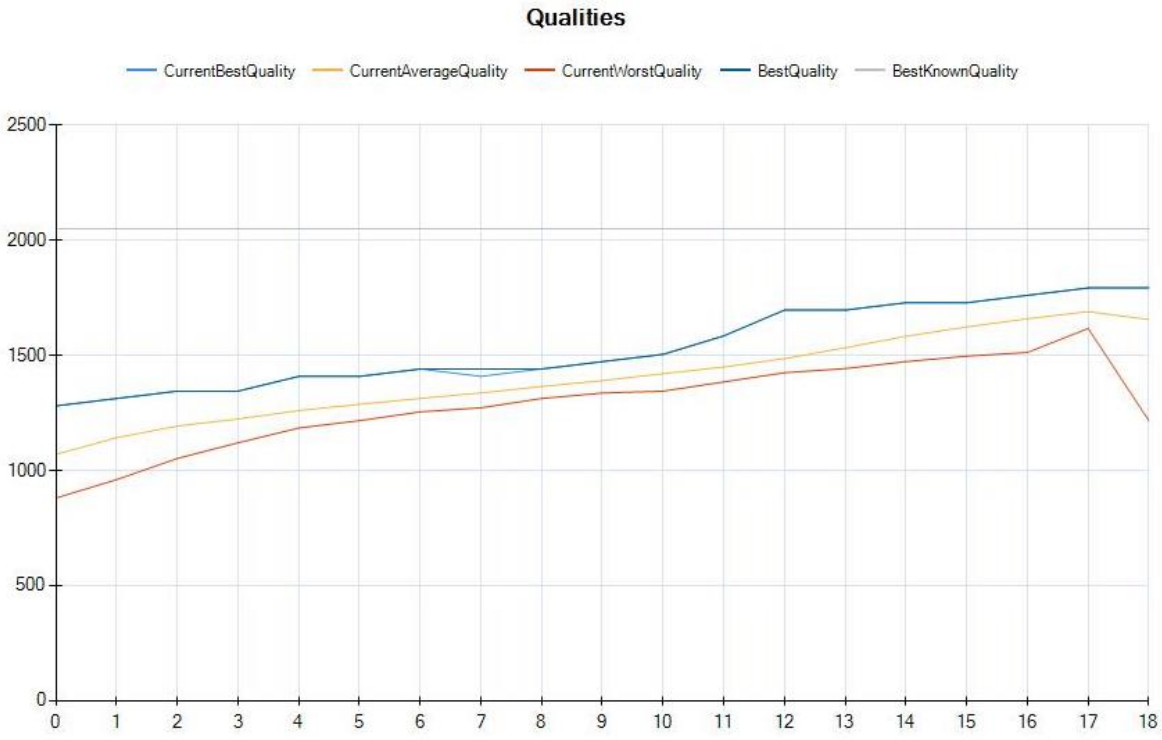




En esta ejecución cabe destacar una diferencia notable en las gráficas y es que mientras nuestra ejecución sigue una evolución cercana a la gráfica del logaritmo, la suya sigue una evolución prácticamente lineal. En nuestro caso se ha encontrado la solución y en su caso se ha quedado muy cerca de hallarla.

En este caso hemos quitado el elitismo en ambas ejecuciones y hemos obtenido estos resultados tanto en nuestro programa como en Heuristic Lab. Es por ello por lo que en nuestra ejecución se aprecia una línea roja y en la suya una azul más clarita en un momento concreto.

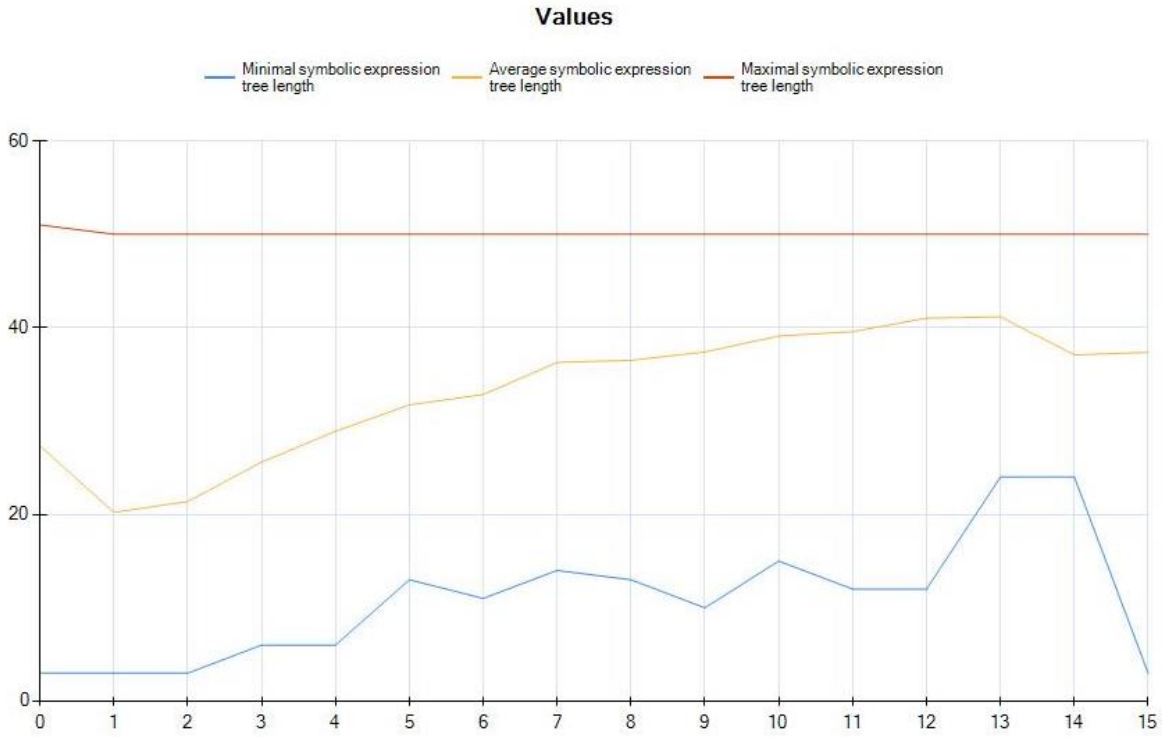




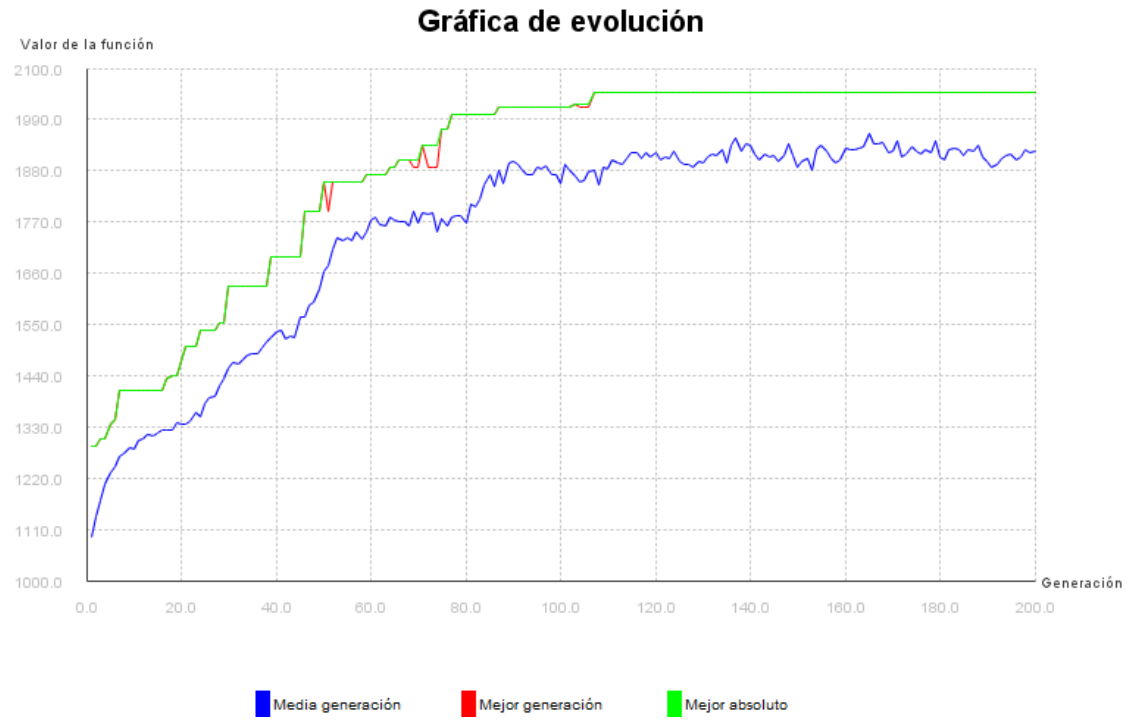
Vemos como en ambos casos obtenemos resultados peores, en nuestro caso se sigue encontrando la solución, pero la evolución es más inestable, con altibajos.

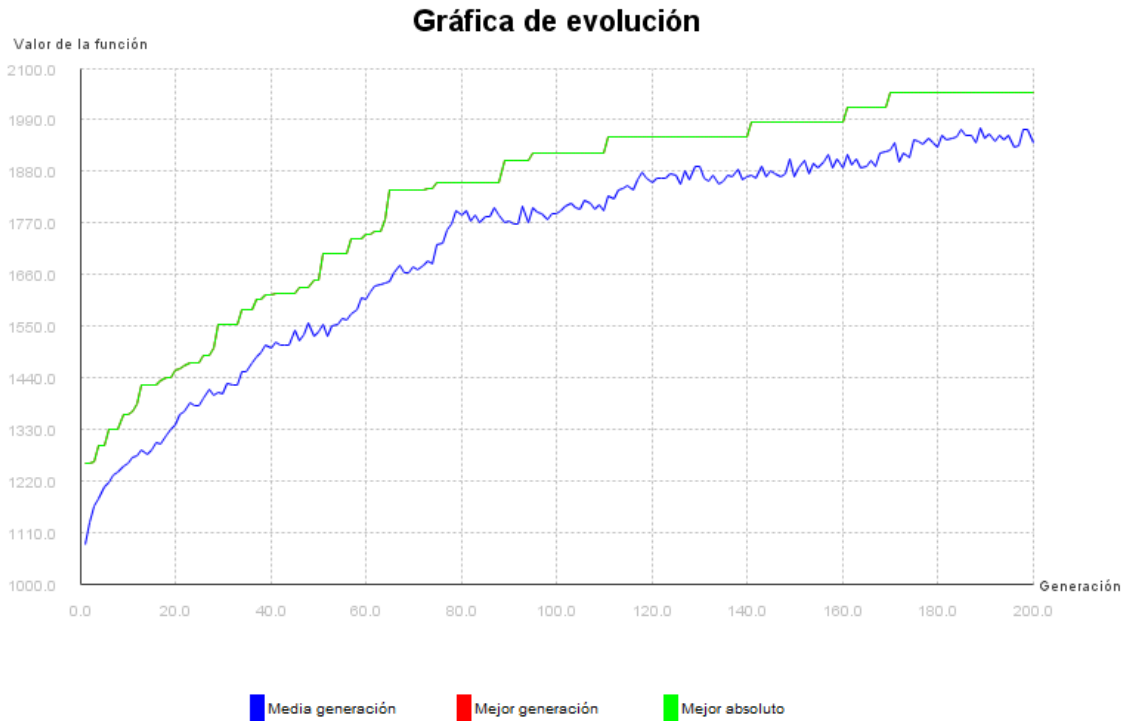
Lo cierto es que Heuristic Lab funciona muy bien frente al problema, pues aun jugando con sus valores suele encontrar resultados bastante buenos. Nos gustaría tener un sitio donde poder ver toda la información en una tabla, pero el entorno nos ofrece mucha información en gráficas separadas por lo que se hace complicado analizar a fondo una comparación entre sus resultados y los nuestros.

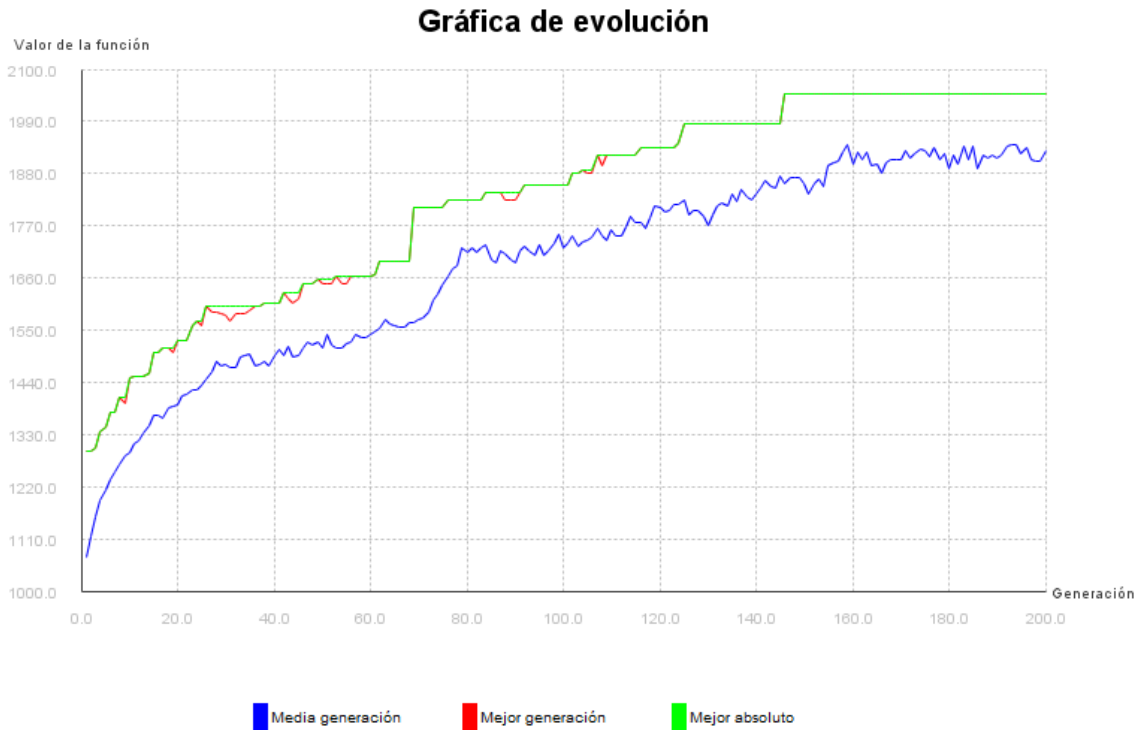
Un dato que cabe destacar es el número de nodos que alcanza el algoritmo, mientras que ellos consiguen unos resultados muy buenos alcanzando un pico de 60 nodos de media, nosotros alcanzamos los 100 nodos aun aplicando el control de bloating, es cierto que puede corregirse aplicando mayor limitación, pero esto normalmente empeora los resultados. A pesar de ello, alcanzar los 100 nodos de media, no significa que la solución sea muy grande, pues puede haberse obtenido en una generación temprana y que el algoritmo haya seguido buscando la solución y aumentado el tamaño de los árboles a pesar de tener el resultado perfecto.

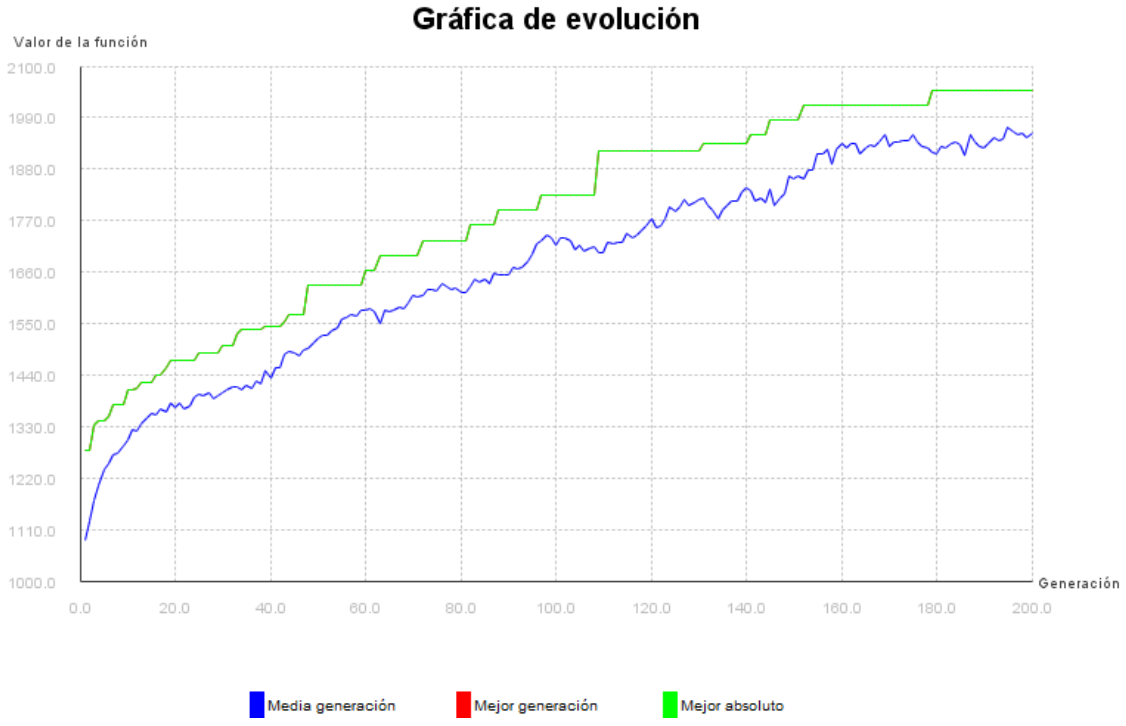


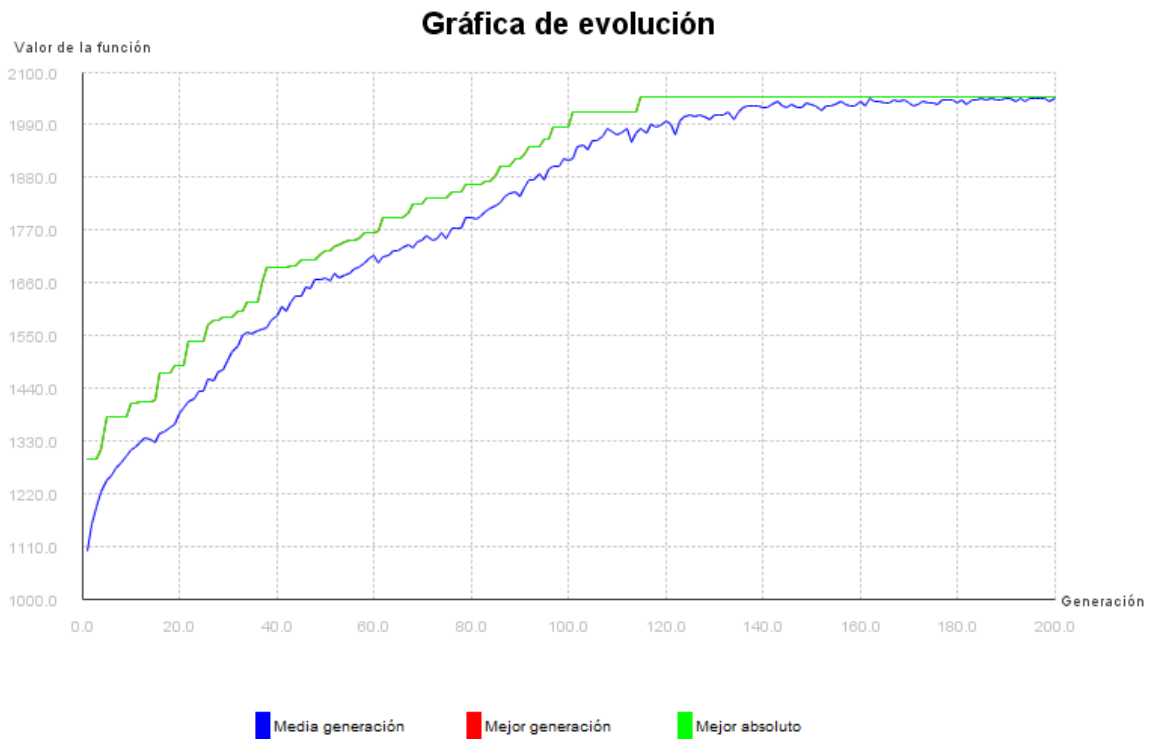
Por último, dejamos otras ejecuciones obtenidas para este problema, en las que se alcanza siempre la solución, podemos observar distintas evoluciones, unas más estables que otras, pero presentando siempre una evolución que es lo que importa, en todos los casos se hizo con una población de 200 individuos y 200 generaciones. Aunque se usaron distintos métodos de control de bloating y selecciones y mutaciones.






Gráfica de evolución
Valor de la función
2100.0
1990.0
1880.0
1770.0
1660.0
1550.0
1440.0
1330.0
1220.0
1110.0
1000.0
0.0
20.0
40.0
60.0
80.0
100.0
120.0
140.0
160.0
180.0
200.0
Generación
Media generación
Mejor generación
Mejor absoluto


Gráfica de evolución
Valor de la función
2100.0
1990.0
1880.0
1770.0
1660.0
1550.0
1440.0
1330.0
1220.0
1110.0
1000.0
0.0
20.0
40.0
60.0
80.0
100.0
120.0
140.0
160.0
180.0
200.0
Generación
Media generación
Mejor generación
Mejor absoluto


Gráfica de evolución
Valor de la función
2100.0
1990.0
1880.0
1770.0
1660.0
1550.0
1440.0
1330.0
1220.0
1110.0
1000.0
0.0
20.0
40.0
60.0
80.0
100.0
120.0
140.0
160.0
180.0
200.0
Generación
Media generación
Mejor generación
Mejor absoluto