Práctica 4

Inteligencia Artificial
Grupo 05

Borja Aday Guadalupe Luis

Eduardo Bryan de Renovales Alvarado

Índice

1. Algoritmo genético del AIMA.

1.1. Funcionamiento del algoritmo genético en AIMA.

1.1.1 Representaciones de los problemas.

En el algoritmo genético implementado por AIMA, un estado está representado como un individuo dentro de la población. Este individuo se define en la clase Individual (aima.core.search.local) y está compuesto por dos atributos:

- List<A> representation : Es una lista que usa un parámetro genérico que nos aporta flexibilidad a la hora de representar el problema, pues esta lista puede ser de Integer, Boolean, String, o incluso clases propias que nosotros hayamos definido.

- int descendants: Es un número que nos indica a cada momento la cantidad de descendientes que tiene el individuo.

Por tanto, para poder hacer uso del algoritmo genético, la representación del estado debe ser una lista de un tipo concreto. Para aportar más posibilidades de representación del estado, AIMA nos ofrece un parámetro genérico <A> para que podamos convertir la representación del estado en una lista de un objeto cualquiera, aumentando así los distintos tipos de representación del problema.

1.1.2. Métodos de selección.

El método de selección utilizado por AIMA es el método de ruleta o Montecarlo, este se implementa en la función randomSelection() de la clase GeneticAlgorithm (aima.core.search.local). Esta función recoge los valores de fitness de toda la población en un array y los normaliza (deja los valores de fitness entre 0 y 1 dividiendo cada uno de ellos por la suma todos los valores). Una vez tiene el vector de los valores de fitness normalizados, genera un número aleatorio entre 0 y 1 y va sumando todos los valores de fitness normalizados hasta encontrar el primero que supere el número aleatorio generado.

Por tanto, el método de selección de ruleta escoge el primer individuo de la población en el que sumando todos los valores de fitness anteriores (normalizados), supere una probabilidad generada al azar. En caso de que no encuentre ninguno que cumpla estas condiciones, simplemente selecciona el último individuo de la población.

1.1.3. El cruce de dos individuos.

El cruce de dos individuos X e Y se realiza mediante la combinación de estos. La cantidad de “genes” que comparte con X e Y se asigna aleatoriamente, se genera un número aleatorio y se rellena los “genes” del hijo hasta esa posición con los “genes” de X y se completa con los de Y. De tal forma que queda un individuo nuevo que comparte información con sus predecesores (aunque no tiene por que ser al 50% con cada uno de ellos, el porcentaje lo decide el random).

1.1.4. La mutación de un individuo.

Cada vez que se realiza un cruce se escoge de manera aleatoria un número, si este es menor a la probabilidad de mutación (en el caso de NReinas 15%) entonces se llama a la función mute() encargada de realizar la mutación del individuo.

Para hacer la mutación, se selecciona aleatoriamente un "gen" que mutar y se le asigna un valor aleatorio dentro de los posibles, de esta forma, obtenemos un nuevo individuo con genes de los padres y que además presenta una mutación con respecto a ellos, para dar variedad a la población y así encontrarla de forma más rápida y efectiva.

1.1.5. Valores configurables en el algoritmo genético.

El algoritmo de AIMA es configurable en su mayoría, ya que la clase en su constructora recibe por parámetro el tamaño máximo de un individuo, el alfabeto posible para cada "gen", la probabilidad de mutación y puede recibir el componente aleatorio del algoritmo.

Por otra parte, la función que genera todos los procesos dentro de esta clase recibe como argumentos la población inicial, la función de evaluación de cada individuo, la función de estado objetivo y el tiempo máximo de ejecución.

Gracias a todos estos parámetros podemos modificar el algoritmo prácticamente de forma libre, aunque es cierto que la representación está limitada a una lista de elementos de un cierto tipo que dificulta en parte a la representación del estado, pero que se salva en gran medida gracias al parámetro genérico <A> que nos permite un mayor juego en la representación del estado en los individuos.

1.2. El problema de las NReinas.

1.2.1. Función de Fitness en el NReinas.

La función de fitness está en la clase NQueensFitnessFuction que se encuentra definida dentro de la clase NQueensGenAlgoUtil (aima.core.environment.nqueens). Dicha función esta implementada por medio de un contador que aumenta en 1 cuando una reina no es atacada por ninguna otra reina en un anillo de radio n alrededor de ella, y a cada iteración ese anillo aumenta en 1 su radio. Para comprobar que no es atacada, para cada reina se comprueba que, primero no haya ninguna reina en la posición de la derecha, después en la de la diagonal derecha arriba, y por último en la diagonal derecha abajo.

1.2.2. Función objetivo en el NReinas.

La función objetivo está en la clase NQueensGoalTest (aima.core.environment.nqueens). Dicha función analiza si están todas las reinas colocadas en el tablero (hay tantas reinas como el tamaño del lado del tablero) y si ninguna es atacada por otra. Esto último se comprueba en la función getNumberOfAttackingPairs(), que está definida en la clase NQueensBoard (aima.core.environment.nqueens). Si se cumple estas condiciones, habremos encontrado un estado objetivo y nuestro problema estará solucionado.

1.2.3. Generación de población inicial en NReinas.

La población inicial en la demo se genera de manera aleatoria teniendo en cuenta el tamaño del tablero. Este proceso se realiza en la función nQueensGeneticAlgorithmSearch(), por medio de la llamada a la función generateRandomIndividual() definida en la claseNQueenGenAlgoUtil que es la encargada de generar realmente el nuevo individuo aleatorio. Dado que la población inicial debe ser de 50 individuos, llama a ésta última función en un bucle for 50 veces, para así tener completa la población inicial y poder continuar en la búsqueda de la solución.

1.2.4. Análisis del problema de las NReinas con algoritmo genético.

Para poder analizar correctamente este algoritmo para este problema en concreto, hemos realizado 4 pruebas diferentes, de tal forma que quede claro que nuestros resultados son coherentes y no están condicionados por factores externos al código (temperatura del procesador, batería del ordenador, etc.).

La primera prueba ha consistido en ejecutar 100 veces el algoritmo genético con un límite de 1 segundo por búsqueda y hemos obtenido los siguientes datos:

Número de soluciones encontradas: 37
Mínimo fitness: 24.0
Máximo fitness: 28.0
Media fitness: 26.7
Mínimo tiempo de las soluciones encontradas: 46
Máximo tiempo de las soluciones encontradas: 968
Media de tiempo de las soluciones encontradas: 482

Dado que 100 veces no son suficientes para cerciorarse de que los datos obtenidos son correctos y representan fielmente el algoritmo de solución del problema, hemos decuplicado la cantidad de veces que ejecutamos el código y hemos comprobado que tanto las soluciones encontradas como las medias son muy similares y por tanto, podemos confiar en que los factores externos al código no afectan de forma directa al algoritmo de solución. Por tanto los valores obtenidos tras 1000 ejecuciones y de los cuales podemos fiarnos más que tras solo 100 ejecuciones son:

Número de soluciones encontradas: 351
Mínimo fitness: 23.0
Máximo fitness: 28.0
Media fitness: 26.586
Mínimo tiempo de las soluciones encontradas: 15
Máximo tiempo de las soluciones encontradas: 998
Media de tiempo de las soluciones encontradas: 515

De estos datos podemos extraer varios datos interesantes:

- La probabilidad de encontrar la solución en menos de un segundo es del 35%.
- Si encuentra la solución, es probable que tarde alrededor de 515 milisegundos en encontrarla.
- Podemos comprobar que el factor suerte está muy presente en el algoritmo, pues puede tardar 15ms en encontrarla o 998 ms, esto se debe a la población aleatoria generada al inicio y las mutaciones aleatorias que se den durante el proceso.
- En cuanto al valor de fitness podemos comprobar que hay un rango bastante corto donde 28.0 indica que se ha alcanzado un estado objetivo.

Tras obtener estos datos hemos realizado 50 ejecuciones sin límite de tiempo para comprobar el tiempo medio que puede tardar en encontrar una solución, y la varianza entre el mínimo y máximo tiempo que tarda en encontrarlas. Hemos obtenido la siguiente información:

Mínimo tiempo de las soluciones encontradas: 156
Máximo tiempo de las soluciones encontradas: 10890
Media de tiempo de las soluciones encontradas: 2373

Tras las 50 ejecuciones podemos comprobar que el algoritmo ha sido capaz de encontrar siempre la solución aunque los tiempos varían muchísimo, desde 156ms hasta 10.890ms, una diferencia de 10.7 segundos entre la más rápida y la más lenta. A pesar de ello, no todas las soluciones se llevan esa diferencia entre ellas y es que lo normal es que el algoritmo tarde alrededor de 2.373ms en encontrar la solución.

Tipo de ejecución	Número soluciones	Mínimo fitness	Máximo fitness	Media fitness	Mínimo tiempo	Máximo tiempo	Media tiempo
100 ejecuciones límite 1”	37	24.0	28.0	26.7	46	968	482
1000 ejecuciones límite 1”	351	23.0	28.0	26.586	15	998	515
50 ejecuciones sin límite					156	10890	2372

Todos estos datos nos llevan a la conclusión de que el algoritmo genético es una buena solución frente a este problema pues es capaz de darnos una respuesta “rápida” al problema. Dependiendo del factor suerte puede tardar prácticamente nada o estar un par de segundos buscando la solución, pero siempre estará en un rango de tiempo más que aceptable.

2. Problema de expresión aritmética con algoritmo genético.

2.1. Análisis del algoritmo original.

Para el análisis de este problema con algoritmo genético hemos realizado distintas pruebas. En el inicio nos encargamos de ejecutar el código para todos los números posibles (desde el 101 hasta el 999) varias veces modificando el valor de la probabilidad de mutación para ver si afecta de manera directa a la solución o no.

2.1.1. Con probabilidad de mutación del 15%.

Al ejecutar el código con una probabilidad de mutación de 0.15 obtuvimos los siguientes resultados:

Número de soluciones encontradas: 450
Mínimo fitness: 5.5063683E7
Máximo fitness: 2.147483647E9
Media fitness: 1.3838017501679645E9
Mínimo tiempo de las soluciones encontradas: 0
Máximo tiempo de las soluciones encontradas: 967
Media de tiempo de las soluciones encontradas: 50

De estos datos podemos extraer varios factores:

- Encuentra la solución en un 50,05% de los casos en menos de un segundo de búsqueda.
- Dependiendo del factor random del algoritmo podemos encontrar la solución instantáneamente o llegar a tardar hasta 967ms en encontrarla.
- Podemos ver que si el algoritmo es capaz de encontrar la solución, la encuentra de forma casi instantánea, tardando alrededor de 50ms en encontrarla.

2.1.2. Con probabilidad de mutación del 30%.

Número de soluciones encontradas: 500
Mínimo fitness: 7.9536431E7
Máximo fitness: 2.147483647E9
Media fitness: 1.4814762396006675E9
Mínimo tiempo de las soluciones encontradas: 0
Máximo tiempo de las soluciones encontradas: 878
Media de tiempo de las soluciones encontradas: 55

Con una probabilidad del 30% de mutación, podemos comprobar como la probabilidad de encontrar la solución sube a un 55,61% sin subir demasiado el tiempo medio en encontrar la solución, tan solo una diferencia de 5ms con respecto a una mutación del 15%.

2.1.3. Con probabilidad de mutación del 70%.

Número de soluciones encontradas: 616
Mínimo fitness: 8.2595524E7
Máximo fitness: 2.147483647E9
Media fitness: 1.7043886185906563E9
Mínimo tiempo de las soluciones encontradas: 1
Máximo tiempo de las soluciones encontradas: 995
Media de tiempo de las soluciones encontradas: 66

Con una probabilidad del 70% de mutación, podemos comprobar como la probabilidad de encontrar la solución sube a un 68,52% aumentando así con creces la eficiencia del código que es capaz de encontrar la solución muchas más veces que con el algoritmo original.

2.1.4. Con una probabilidad de mutación del 99%.

Número de soluciones encontradas: 827
Mínimo fitness: 1.26322567E8
Máximo fitness: 2.147483647E9
Media fitness: 2.0289668621724138E9
Mínimo tiempo de las soluciones encontradas: 0
Máximo tiempo de las soluciones encontradas: 884
Media de tiempo de las soluciones encontradas: 39

Con esta solución la probabilidad de encontrar la solución sube hasta un 91,99%. Es el mejor dato que obtenemos con este algoritmo. Esto nos hace indicar que para resolver este ejercicio con el algoritmo propuesto, suele mejorar con una mutación mayor dado que si se producen

pocas mutaciones el código se quedará estancado analizando formulas muy parecidas con las que nunca llegará a encontrar la solución o tardará demasiado.

2.1.5. Conclusión del análisis del algoritmo original.

Tipo de ejecución	Número soluciones	Mínimo fitness	Máximo fitness	Media fitness	Mínimo tiempo	Máximo tiempo	Media tiempo
P(15%)	450	5.5063E7	2.14746E9	1.383801E9	0	967	50
P(30%)	500	7.9536E7	2.14746E9	1.481476E9	0	878	55
P(70%)	616	8.2595E7	2.14746E9	1.704388E9	1	995	66
P(90%)	842	2.1477E9	2.14746E9	2.058702E9	0	990	34

Tras ejecutar varias veces el algoritmo original con distintas probabilidades de mutación, hemos comprobado que ésta es directamente proporcional al número de soluciones encontradas; y es que a mayor probabilidad de mutación, el algoritmo ha sido capaz de encontrar soluciones para más números diferentes. Por tanto, para resolver este ejercicio con el algoritmo original propuesto recomendamos usarlo con una probabilidad de mutación de entre un 90 y un 99%. Si se ejecuta el algoritmo original con un 15% para todos los números posibles tardará unos 12 minutos en hallar los resultados de 453 números, sin embargo, con una probabilidad del 99% tarda alrededor de 3 minutos y te ofrece resultados para 842 números diferentes. Es más que notable la diferencia de eficiencia del código.

2.2. Análisis del algoritmo con probabilidad de cruce.

Esta es una modificación muy simple del algoritmo anterior donde añadimos una probabilidad de cruce entre 2 individuos, de tal forma que si el número generado está por debajo de esta probabilidad, los cruzamos y generamos un nuevo individuo. Si esto no ocurre, se devuelve uno de los padres al azar.

2.2.1. Algoritmo con probabilidad de cruce del 70%.

Tras ejecutar el algoritmo con una probabilidad de cruce del 70% y una probabilidad de mutación del 15% hemos obtenido los siguientes resultados:

Número de soluciones encontradas: 467
Mínimo fitness: 3.1122951E7
Máximo fitness: 2.147483647E9
Media fitness: 1.4109870748965516E9
Mínimo tiempo de las soluciones encontradas: 0
Máximo tiempo de las soluciones encontradas: 980
Media de tiempo de las soluciones encontradas: 57

Añadiendo la probabilidad de cruce y sin modificar la probabilidad de mutación obtenemos datos mejores. Pues pasa de encontrar un 50,38% de las soluciones a encontrarlas en un 51,94% de los casos. Podemos comprobar como el hecho de no cruzar siempre a los individuos no mejora en gran medida el algoritmo. A pesar del aumento de tiempo medio, merece la pena este aumento de tiempo para la cantidad de soluciones que es capaz de aportar.

2.2.2. Algoritmo con probabilidad de cruce del 80%.

Esta ejecución se ha realizado con una probabilidad de cruce del 80% y una probabilidad de mutación del 15% para poder así comparar los resultados obtenidos con la ejecución anterior.

Número de soluciones encontradas: 425
Mínimo fitness: 5.8040098E7
Máximo fitness: 2.147483647E9
Media fitness: 1.3388092317474973E9
Mínimo tiempo de las soluciones encontradas: 1
Máximo tiempo de las soluciones encontradas: 998
Media de tiempo de las soluciones encontradas: 59

Esta nueva probabilidad de cruce empeora el algoritmo original con 15% de probabilidad de mutación pues encuentra la solución en el 47,27% de los casos. Aumenta el tiempo en medio en encontrarlas y además encuentra muchas menos, lo que hace que el tiempo que está en ejecución buscando soluciones para 899 números aumente y además encuentre menos soluciones que el algoritmo original.

2.2.3. Conclusión del algoritmo con probabilidad de cruce.

Tipo de ejecución	Número soluciones	Mínimo fitness	Máximo fitness	Media fitness	Mínimo tiempo	Máximo tiempo	Media tiempo
P(70%)	467	3.1122E7	2.14746E9	1.410987E9	0	980	57
P(80%)	425	5.8040E7	2.1474E9	1.338809E9	1	998	59

Este algoritmo que añade una probabilidad de cruce no mejora considerablemente el algoritmo inicial. Podemos observar que no hay una diferencia enorme entre el 70% de probabilidad de cruce y el 80% pues nos dan resultados muy parecidos. Incluso en algunas ejecuciones realizadas se han turnado y uno ha sido mejor que el otro y viceversa. Aunque en los datos dados anteriormente indica que se encontró más resultados con el 70% que con el 80%. Ambas son buenas soluciones al problema pero no suponen una gran mejora con respecto al algoritmo original por lo que su implementación no es necesaria para nuestro problema.

2.3. Análisis del algoritmo con generación de dos hijos.

Esta es una modificación muy simple del algoritmo original donde en la función nextGeneration() de la case GeneticAlgorithm en vez de generar un solo hijo, generamos dos y los añadimos a la nueva población. De esta forma, generaremos dos individuos a partir de dos individuos padres y reducimos así las variaciones entre individuos por lo que previsiblemente conseguiremos peores resultados pues hará que busque siempre la solución en una rama con la que es muy complicado llegar.

Para asegurarnos de que los datos son fiables, hemos ejecutado este algoritmo varias veces y hemos comprobado que los resultados son muy similares, por lo que nos quedaremos con dos de las soluciones más representativas y compararemos con el algoritmo inicial.

Ejecución 1:
Número de soluciones encontradas: 471
Mínimo fitness: 6.1356675E7
Máximo fitness: 2.147483647E9
Media fitness: 1.4088272614538376E9
Mínimo tiempo de las soluciones encontradas: 0
Máximo tiempo de las soluciones encontradas: 1000
Media de tiempo de las soluciones encontradas: 56

Ejecución 2:
Número de soluciones encontradas: 442
Mínimo fitness: 4.5691141E7
Máximo fitness: 2.147483647E9
Media fitness: 1.370884372008987E9
Mínimo tiempo de las soluciones encontradas: 0
Máximo tiempo de las soluciones encontradas: 920
Media de tiempo de las soluciones encontradas: 41

Media de las ejecuciones:
Número de soluciones encontradas: 456.5
Mínimo fitness: 5.3523908E7
Máximo fitness: 2.147483647E9
Media fitness: 1.389855817E9
Mínimo tiempo de las soluciones encontradas: 0
Máximo tiempo de las soluciones encontradas: 960
Media de tiempo de las soluciones encontradas: 48.5

Tipo de ejecución	Número soluciones	Mínimo fitness	Máximo fitness	Media fitness	Mínimo tiempo	Máximo tiempo	Media tiempo
Ejecución 1	471	6.1356E7	2.1474E9	1.408822E9	0	1000	56
Ejecución 2	442	4.5691E7	2.1474E9	1.370884E9	0	920	41
Media	456.5	5.3523E7	2.1474E9	1.389858E9	0	960	48.5

Una vez obtenidos estos datos ya podemos comenzar a comparar con los del algoritmo original, como podemos ver, encuentra más veces la solución. Mientras que este algoritmo encuentra la solución en el 50,77% de los casos, el algoritmo original las encuentra en el 50,05% de los casos, suponiendo una mejora del código de un 0,77% que es prácticamente invisible. En cuanto a las medias de fitness también se asemejan muchísimo al del original, lo que indica que además se a la misma distancia de la solución que el algoritmo original. Y por último, destacable el tiempo medio de encontrar la solución, mientras que el algoritmo original es capaz de encontrar la solución en 50ms, este necesita de una media de 48,5ms, por tanto, consigue de media más soluciones que el original y en menos tiempo.

2.4. Análisis del algoritmo con estrategia no destructiva.

Esta es una modificación muy simple del algoritmo original donde en la función nextGeneration() de la case GeneticAlgorithm generamos un hijo y si este tiene un mayor fitness que sus padres, lo añadimos a la población, si esto no ocurre, añadimos el padre con mayor fitness. De esta forma conseguimos que el fitness nunca disminuya, pero eso no quiere decir que vayamos a encontrar mayor cantidad de soluciones pues posiblemente se quede atascado en una mala rama intentando buscar una solución por un camino muy complicado.

Para asegurarnos de que los datos son fiables, hemos ejecutado este algoritmo varias veces y hemos comprobado que los resultados son muy similares, por lo que nos quedaremos con dos de las soluciones más representativas y compararemos con el algoritmo inicial.

Ejecución 1:
Número de soluciones encontradas: 387
Mínimo fitness: 2.3091222E7
Máximo fitness: 2.147483647E9
Media fitness: 1.273346422355951E9
Mínimo tiempo de las soluciones encontradas: 0
Máximo tiempo de las soluciones encontradas: 99
Media de tiempo de las soluciones encontradas: 12

Ejecución 2:
Número de soluciones encontradas: 392
Mínimo fitness: 2.4403223E7
Máximo fitness: 2.147483647E9
Media fitness: 1.2596873494382648E9
Mínimo tiempo de las soluciones encontradas: 0
Máximo tiempo de las soluciones encontradas: 125
Media de tiempo de las soluciones encontradas: 9

Media de las ejecuciones:
Número de soluciones encontradas: 389.5
Mínimo fitness: 2.3747222E7
Máximo fitness: 2.147483647E9
Media fitness: 1.266516886E9
Mínimo tiempo de las soluciones encontradas: 0
Máximo tiempo de las soluciones encontradas: 112
Media de tiempo de las soluciones encontradas: 10.5

Tipo de ejecución	Número soluciones	Mínimo fitness	Máximo fitness	Media fitness	Mínimo tiempo	Máximo tiempo	Media tiempo
Ejecución 1	387	2.3091E7	2.1474E9	1.2733E9	0	99	12
Ejecución 2	392	2.4403E7	2.1474E9	1.2596E9	0	125	9
Media	389.5	2.3747E7	2.1474E9	1.2665E9	0	112	10.5

Con estos datos podemos comparar con este algoritmo con el original, como podemos ver, encuentra menos veces la solución. Mientras que este algoritmo encuentra la solución en el 43,32% de los casos, el algoritmo original las encuentra en el 50,05% de los casos, suponiendo un empeoramiento del código de un 6,73%. En cuanto a las medias de fitness también disminuyen, lo que indica que además de no encontrarlas, se queda más lejos que el algoritmo original de la solución final. Y por último, muy destacable el tiempo medio de encontrar la solución, mientras que el algoritmo original suele encontrar la solución en 50ms, este necesita de una media de 10.5ms, esto nos indica que si el algoritmo es capaz de encontrar la solución, lo va a encontrar casi de forma instantánea, pero no al ver que no encuentra sino la mitad de las soluciones podemos afirmar que este algoritmo se atasca demasiado en caminos muy difíciles para hallar la solución. Esto es provocado porque muchas veces los nuevos hijos no serán mejores que los padres y entonces nos quedamos con los propios padres para seguir probando, lo que genera una nueva población muy poco variada con respecto a la anterior y así disminuyendo la probabilidad de encontrar una respuesta.

2.5. Análisis final de las soluciones propuestas.

Tras el análisis de todos los algoritmos propuestos. Podemos concluir varias cosas:

- El algoritmo que mejor resultado ha dado con una mutación del 15% es el que añade una probabilidad de cruce del 70%.
- Todos los algoritmos propuestos son muy similares en cuanto a rendimiento.
- Lo que hace variar verdaderamente la eficiencia del algoritmo es la probabilidad de mutación de un individuo. Esto se debe a que al haber pocas mutaciones la población generada siempre será muy similar y nos quedaremos buscando por una rama en la que quizás es muy complicado encontrar una solución. Al añadir muchas mutaciones pero sin dejar de lado la influencia de los padres obtenemos diversas ramas para conseguir la solución final de forma más rápida.

Por tanto, para resolver este problema recomendamos usar una mutación de entre el 90% y el 99%, de esta forma se conseguirá una tasa de acierto superior al 90% en la mayoría de los casos.

2.6. Comparativa entre releases.

Durante la recogida de datos de las distintas ejecuciones, hemos usado dos ordenadores distintos que poseían en el Eclipse distintos reléase del AIMA y nos hemos dado cuenta de que existe una variación más que notable entre ambos.

A pesar de que todos los datos que puedes observar en este documento están obtenidos mediante el último release que se ha publicado en el campus, no podemos dejar de comentar la diferencia tan grande que supone usar el release anterior en el caso de usar el algoritmo con probabilidad de cruce tanto al 70% como el 80%.

Mejora considerablemente la cantidad de soluciones obtenidas, usando el mismo código pero con el release anterior. Los datos que hemos obtenido usando el release anterior con el algoritmo que incluye una probabilidad de cruce son:

Tipo de ejecución	Número soluciones	Mínimo fitness	Máximo fitness	Media fitness	Mínimo tiempo	Máximo tiempo	Media tiempo
P(70%)	897	18722661	2.14746E9	2.142710E9	0	937	85
P(80%)	895	1852876	2.14746E9	2.137938E9	0	999	72

Como podemos observar, consigue hallar la solución en el 99,66% de los casos en menos de un segundo, lo que lo hace convertirse en el rey de los algoritmos propuestos en este PDF para encontrar la solución a este problema, dejando atrás a todos los demás.

Hemos investigado para ver si podíamos comprender de donde viene esta diferencia y el único cambio notable está en la clase nextGeneration(), pues mientras en el anterior release devuelve el mejor Indivual<A> de la población generada, el nuevo release devuelve una lista List<Individual<A>> con todos los nuevos individuos generados, suponemos que esto supone un coste mucho mayor y por tanto en un segundo genera menos hijos en los que podría encontrar la solución como hace su release predecesor.