Algoritmos Genéticos y Evolutivos

Práctica 1: N Reinas.

Autor:

Daniel Fernández: daniel.f.rico@alumnos.uc3m.es

1 de junio de 2018

Índice

1. Contexto

1.1. Problema de las 8 reinas

El problema de las ocho reinas es un problema clásico en computación y matemáticas, el cual consiste en colocar ocho reinas en un tablero de ajedrez (de 8 casillas) sin que se amenacen entre sí (en ajedrez, las reinas pueden moverse tantas casillas como quieran en cualquier dirección).

El problema original data de 1848 y fue propuesto por el ajedrecista Max Bezzel. Durante años, muchos matemáticos (incluyendo a Gauss), han trabajado en él y lo han generalizado a N-reinas. Las primeras soluciones se plantean en 1850.

Edsger Dijkstra usó este problema en 1972 para ilustrar el poder de la llamada programación estructurada. Publicó una descripción muy detallada del desarrollo del algoritmo de backtracking, "depth-first".

Para nuestro problema, tomaremos un enfoque más abstracto: dado un tablero de dimensiones NxN, queremos colocar N reinas sin que se ataquen entre ellas.

1.2. Problema planteado

Se plantea resolver el problema de las N-reinas para un valor de N muy inferior a 1000 sin necesidad de hallar todas las soluciones, mediante la utilización de un Algoritmo Genético.

Primero afrontaremos el problema mediante un algoritmo de fuerza bruta con *backtracking*, para estudiar cuánto tarda en hallar posibles soluciones dentro del espacio de búsqueda.

A continuación, se implementa un Algoritmo Genético con el fin de hallar las soluciones para valores de N computacionalmente intensos.

2. Solucionar las N-reinas empleando fuerza bruta

Para esta aproximación al problema implementamos una solución por fuerza bruta al problema de las N-reinas. Esta implementación funciona tanto para encontrar una solución al problema como para encontrar todas (con diferentes criterios de parada y modificando ligeramente el código).

La implementación empieza con un vector de N reinas colocadas en coordenadas nulas, e intenta progresivamente colocarlas sobre el tablero, comprobando para cada una de ellas si no es adyacente a otra de las reinas ya colocadas anteriormente. Si no es así, se coloca la reina. Si recorremos todas las filas y no hemos sido capaces de colocar la reina que falta, hacemos *backtrack* eliminando la última reina colocada sobre el tablero (vector de coordenadas) y vuelta a empezar. Si esta opción tampoco es válida, necesitamos hacer doble *backtracking* para volver a una rama anterior para seguir explorando el árbol de búsqueda. Esto se consigue eliminando una reina más.

Esta implementación es muy rápida ya que el código es simple, y para valores relativamente pequeños de N el espacio de búsqueda no es muy grande. Considerando cada intento de colocar una reina como un ciclo, obtenemos los siguientes resultados:

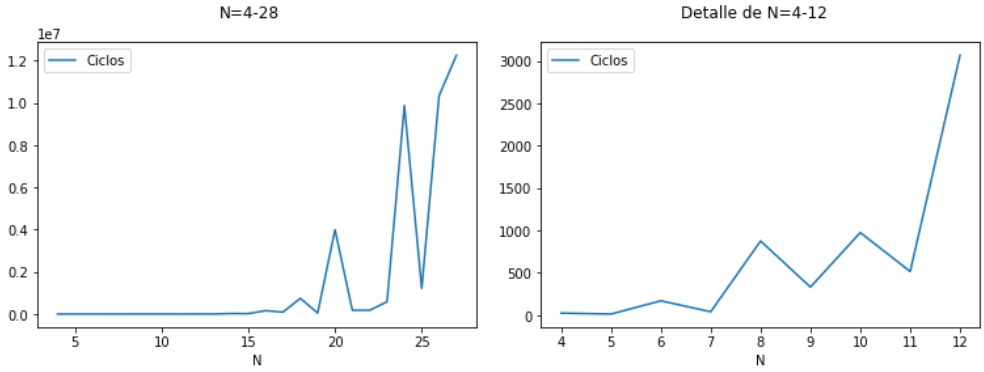

Figura 1: Número de iteraciones hasta encontrar 1 solución

3. Solucionar las N-reinas con Algoritmos Genéticos

3.1. Codificación

La codificación actual para nuestro problema trata de un vector de tamaño N, donde cada valor corresponde a la posición x de una reina sobre la fila y del tablero, siendo y el índice del elemento x en el vector de reinas.

3.1.1. ¿Qué otras alternativas se han barajado y descartado?

En primer lugar, se planteó una codificación del tablero como vector binario de tamaño NxN, de tal modo que cada individuo de la población fuese un vector de 0s y 1s, donde un 1 equivale a una reina en la posición equivalente del tablero, y un 0 la ausencia de ésta.

En segundo lugar, se planteó un vector de reinas de tamaño N, donde sólo se acumulasen las reinas que se encontraban en el tablero. Dicho vector estaría compuesto de tuplas (x, y) correspondientes a la posición de cada reina en el tablero.

Al darnos cuenta de que los números x e y han de ser excluyentes (restricción para evitar ataques en la misma fila y columna), se planteó la codificación actual que hace uso del índice de la lista como posición y de la reina (valor unívoco).

3.1.2. ¿Por qué la codificación actual es mejor que las otras?

Esta codificación es mucho más eficiente que las codificaciones anteriormente propuestas, ya que ocupa mucho menos espacio en memoria (un vector de enteros ocupa $K * N$ bits, siendo $K = 4$ para una arquitectura estándar de 32-bit o 64-bit, frente a la peor solución anteriormente propuesta, un vector $N * N$ binario, que ocuparía $N * N$ bits).

Además, esta codificación lleva implícitas las restricciones de fila y columna para cada reina. La restricción fila va implícita en la propia estructura de datos (no puede haber dos elementos en la misma posición de la lista), y la restricción columna va implícita en el hecho de que los valores del vector son permutaciones de N (no se repite ningún valor de x).

Esto nos elimina automáticamente muchas soluciones inválidas de la población, agilizando el algoritmo y permitiendo encontrar una solución mucho más rápidamente.

3.2. Función de *fitness*

Se plantean dos funciones de *fitness* para el problema a tratar:

1. En primer lugar, y considerando que no llegamos a posicionar N número de reinas, se considera la función $F = \frac{n}{N} - \frac{b}{n}$, siendo n el número total de reinas posicionadas sobre el tablero frente a N (número ideal de reinas y dimensión del lado del tablero), y b el número de reinas mal posicionadas. Esta función vale 1 cuando todas las reinas están bien posicionadas, y 0 cuando están todas mal posicionadas, o no hay reinas sobre el tablero.

2. Considerando que, debido a nuestra codificación, no existe la posibilidad de que n sea distinta de N (siempre hay N reinas sobre el tablero, aunque estén mal situadas), se propone la función $F = 1 - \frac{b}{N}$, que no es más que una sustitución de $n = N$ en la función propuesta anteriormente. Esta es la función implementada.

Nuestro algoritmo incorpora también un caché de *fitness* por individuo, con el objetivo de evitar evaluaciones sobre individuos idénticos. De esta forma, los accesos a caché no se cuentan como evaluaciones, sólo cuando es necesario calcular el *fitness* del individuo dado.

3.3. Métodos implementados

3.3.1. Selección

El método de selección aplicado para nuestro problema es una selección por torneo.

Para esta selección tomamos un parámetro K que se corresponde con el tamaño de la muestra de población que tomamos para nuestro torneo. A continuación, evaluamos la muestra y seleccionamos al mejor individuo un número de veces definido por el parámetro L (el número de hijos que queremos conseguir a partir de $L * 2$ padres).

Originalmente, se implementa el método de selección sugerido en el libro *An Introduction to Genetic Algorithms* (Melanie, M. 1996), en el cual la selección por torneo ordenaba la muestra de tamaño K por su grado de *fitness*, y seleccionaba los dos mejores como padres de la nueva población. Por razones de simplificación, se decide que el corte se realiza en L, quedando por defecto un tamaño de muestra K y un tamaño de ganadores del torneo de L.

Sin embargo, se llegó a la conclusión de que era mejor repetir el proceso de torneo L veces hasta conseguir el número deseado de padres.

El método de selección escogido va acompañado de la política de reemplazo correspondiente (siempre renovando una parte significativa de la población, igual al número de hijos que introducimos en ésta). El tamaño de la población siempre se mantiene constante.

3.3.2. Cruce

El cruce que se implementa es una recombinación discreta entre los dos padres con cuidado de no repetir ninguno de los anteriores valores de N.

En el código se implementan tres métodos de cruce:

- **Cruce secuencial constructivo aleatorio**: vamos construyendo el hijo progresivamente (de longitud 0 a N, donde el valor i-ésimo del hijo corresponderá con el bit i de uno de los padres, escogido al azar.

- **Cruce secuencial constructivo (SCX)**: parecido al método anterior, este método va construyendo progresivamente el hijo con una pseudo-evaluación a modo de heurística. Para cada par de bits $(p1, p2)$ se evalúa su coste (*fitness* del individuo $F'(I) = F(I)+p$) de ser añadido al hijo, y se coge el bit que mejora el *fitness* del individuo.

  Ejemplo:

  $p1 = [4, 3, 1, 2]$, $p2 = [1, 2, 3, 4]$

  $h = [1]$ (inicializado al azar) $\rightarrow$ se evalúa $h_1 = [1, 3]$ y $h_2 = [1, 2]$.

  Si se considera que $h_1 = [1, 3]$ tiene mejor *fitness* que $h_2 = [1, 2]$, $h = [1, 3]$, y así N veces hasta completar el hijo.

- **Cruce ordenado 1 (OC1)**: sean dos padres $p1$ y $p2$, uno de ellos se considera cadena de *corte* y otro cadena de *llenado*. Se selecciona una sub-cadena del primer padre y se inserta en el hijo respetando la posición original. Se llenan los huecos laterales con los valores del otro padre que no están ya en la sub-cadena seleccionada del primero (para no repetir valores).

  Ejemplo: $p1 = [A, B, C, D, E, F, G, H, I]$, $p2 = [a, b, c, d, e, f, g, h, i]$

  Sección de corte: $\_\_CDEF\_\_\_$. Valores de llenado: h, a, i, b, g

  $h = [h, a, C, D, E, F, i, b, g]$

  Posibles alternativas

  Cambiando el orden de inserción del bloque de llenado: $h_2 = [b, g, C, D, E, F, h, a, i]$

  Inviertiendo la selección *corte-llenado*: $h_3 = [H, A, c, d, e, f, I, B, G]$

3.3.3. Mutación

1. Primera aproximación: se mutan los dos hijos con probabilidad p_m. Aproximación clásica: por cada bit del individuo, se evalúa su probabilidad de mutación, y se cambia el bit (0-1). Al no ser codificación binaria, lo cambiamos por valores 0-N.

2. Segunda aproximación: con (1) obtenemos resultados repetidos, nuestra codificación es una permutación de columnas (valores 0-N). Por lo tanto, evaluamos la probabilidad del individuo entero de mutar, y si es positiva hacemos un reordenamiento de sus valores (*shuffle*).

3. Tercera aproximación: con (2) obtenemos valores demasiado aleatorios con una probabilidad de mutación elevada. Nuestra aproximación esta vez consiste de nuevo en evaluar la probabilidad de mutación bit a bit, y en caso de ser positiva, intercambiando dicho bit con otro bit del individuo de manera aleatoria. De esta forma, en caso de mutar 1 bit, el individuo sólo diferirá del original en 2 bits.

4. Cuarta aproximación: con (3) se estanca mucho. Queremos mutar el individuo más, introducimos un índice de diversidad. Este índice multiplica la probabilidad de mutación hasta 1, momento en el cual se resetea de nuevo (para evitar demasiada aleatoriedad). Cambiamos también el planteamiento de (3), ahora considera la probabilidad pm para todo el individuo, y en caso positivo hace un solo cambio de un par de bits (*swap*).

Nota: la probabilidad de mutar depende de N, ya que el número de bits a mutar depende también del tamaño del individuo, así que para hacerlo más constante tomamos la probabilidad de 1/N y la multiplicamos por la probabilidad por individuo.

3.3.4. Evaluación de la población: posibles soluciones

En primer lugar, recorremos toda la población evaluando el *fitness* de cada individuo. En caso de encontrar un individuo con *fitness* perfecto (1), es decir, una solución, se imprime por pantalla la solución, el número de evaluaciones hasta ese momento, así como el número de ciclos.

En caso de no encontrar solución, revisamos igualmente el *fitness* del mejor individuo de la población para ver si ha mejorado de una generación a otra. En caso contrario, modificamos el índice de diversidad descrito anteriormente.

3.4. Parametrización

En *An Introduction to Genetic Algorithms* (Melanie, M. 1996) se menciona una parametrización propuesta por De Jong (1975) que consiste en un tamaño de población de 50 a 100 individuos, una probabilidad de cruce (para un cruce unipunto) de 0.6, y una tasa de mutación de 0.001 por bit.

En el mismo libro, Grefenstette (1986) propone un tamaño de población de 20 a 30 individuos, probabilidad de cruce de 0.75 a 0.95, y tasa de mutación de 0.005 a 0.01.

Los parámetros usados finalmente son: un tamaño de población de 100 individuos, un tamaño de torneo de 4 individuos (con una probabilidad de 'suerte' [que no salga elegido el mejor] de 0.75), 80 hijos generados (con un reemplazo generacional del 80 %), probabilidad de cruce de 0.9, y tasa de mutación de 0.03.

A continuación se incluyen unas gráficas de la evolución de la función de *fitness* a lo largo de varias iteraciones, donde podemos ver que con los parámetros anteriores (mejor reemplazo generacional, torneos más grandes, menor probabilidad de cruce, etc.) se llegaba a mínimos locales y la población se estancaba muy pronto.

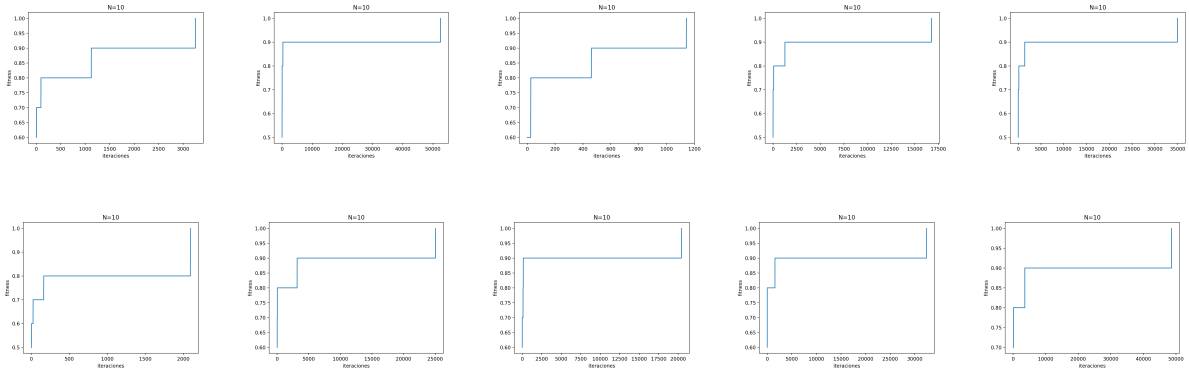

Figura 2: Parámetros De Jong para diferentes valores de N

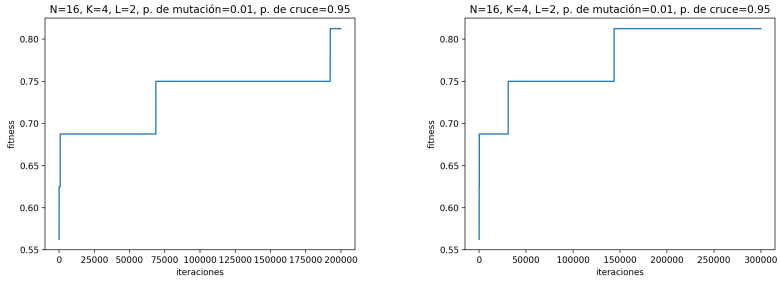

Figura 3: Parámetros Grefenstette

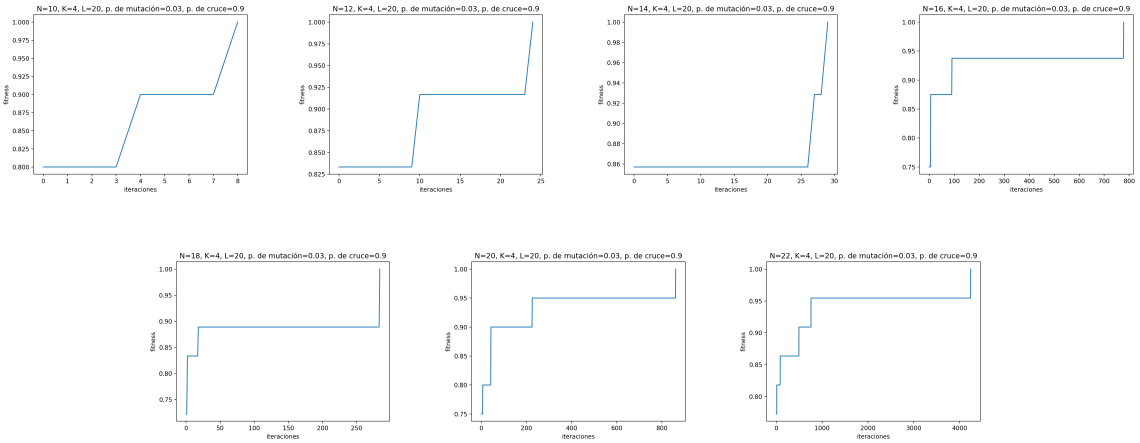

Figura 4: Parámetros propios

3.5. Resultados

En primer lugar podemos ver unas gráficas del número de ciclos y evaluaciones requeridas para hallar la primera solución para diversos valores de N con la codificación binaria (tablero como vector binario de longitud $N * N$). Esta codificación se evalúa con funciones canónicas (selección por torneo, cruce simple, mutación de *bit-flipping* sujeta a probabilidad...).

Para cada valor de N se ha ejecutado el código 10 veces con los mismos parámetros para evaluar con valores medios.

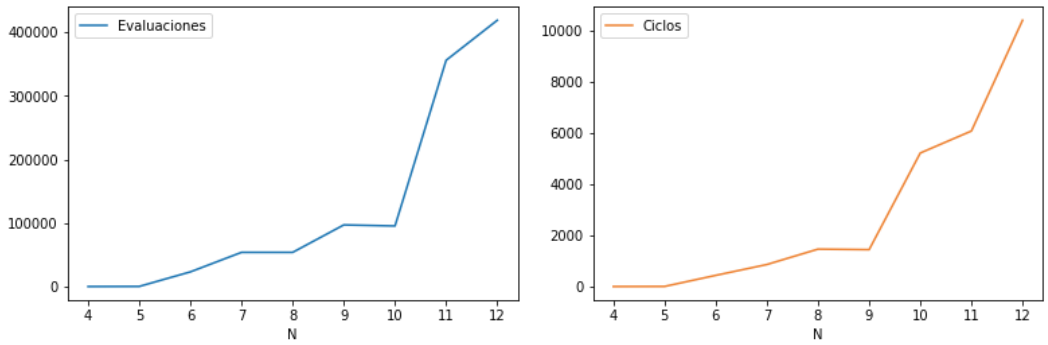

Figura 5: Codificación binaria

A pesar de cachear las evaluaciones, al trabajar con individuos mucho más grandes, la variabilidad de estos y el espacio de búsqueda son mucho mayores (no se restringen posiciones ni número de reinas).

A continuación podemos ver una gráfica del número de ciclos y evaluaciones requeridas para hallar una primera solución para diversos valores de N con las primeras versiones del algoritmo.

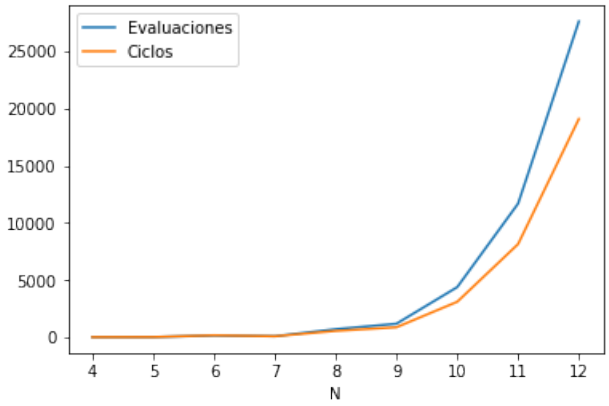

Figura 6: Primera versión con codificación ordenada

Podemos ver que el número de ciclos y evaluaciones se dispara por el crecimiento del espacio de búsqueda y por el aumento en la variedad de los individuos respectivamente.

Al implementar los cambios anteriormente descritos, vemos los siguientes resultados:

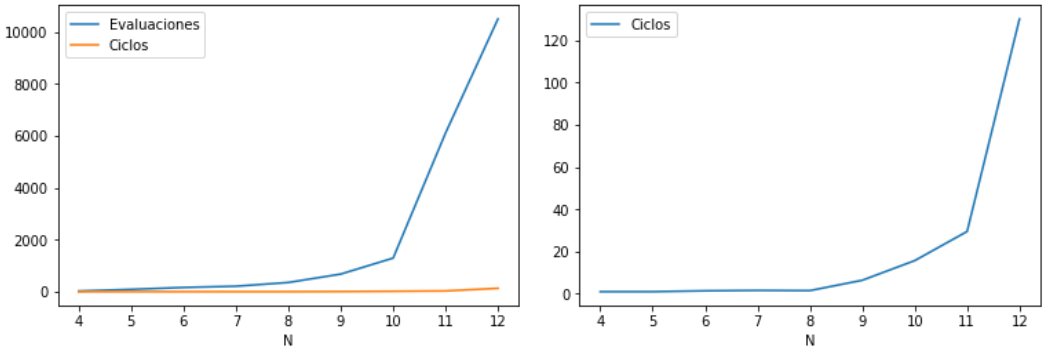

Figura 7: Versión final

En la versión final podemos ver que el número de evaluaciones tiene un crecimiento más suave y el número de ciclos se reduce significativamente.

4. Ampliación: todas las soluciones

Dado que el problema original de las 8 reinas se sabe que tiene 92 soluciones, se considera ese valor para nuestra N para poder evaluar las tres aproximaciones al problema anteriormente descritas, considerando como condición de parada que nuestro algoritmo encuentre 92 soluciones.

A continuación se detallan el número necesario de ciclos y evaluaciones para intentar encontrar las 92 soluciones para $N = 8$ con el algoritmo de fuerza bruta, el algoritmo genético con codificación binaria, y el algoritmo genético con codificación ordenada.

	Soluciones	Ciclos	Evaluaciones	Tiempo
Fuerza bruta	92	16456	-	0.151s
AG c. binaria	22	62490	3173586	33m23s
AG c. ordenada	92	2041	35819	19s

Figura 8: Resultados para diversas soluciones

Aunque los tiempos, el número de soluciones y ciclos varíen, se puede observar claramente que el algoritmo de fuerza bruta mejora muy sustancialmente los resultados de los otros dos (para un número pequeño de N como es 8). El genético con la codificación ordenada desempeña bien en número de evaluaciones y ciclos, a pesar de quedar peor en tiempo de ejecución con respecto al algoritmo de fuerza bruta.

El genético con la codificación binaria es el que peor resultado obtiene, consiguiendo sacar solo 22 soluciones en 62 490 ciclos y algo más de 30 minutos de ejecución.

Con la **primera versión** del algoritmo genético, obteníamos 22 soluciones en 1 628 883 ciclos (algo más de 2h de ejecución). Aquí podemos ver en detalle el número de ciclos necesarios para hallar las soluciones 1 a 22 en el algoritmo genético con codificación ordenada en esta primera versión.

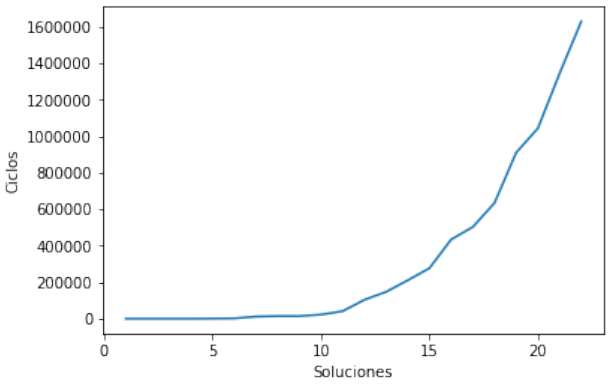

Figura 9: Ciclos necesarios para hallar varias soluciones

A partir de este conjunto de datos se intenta ajustar los valores a una ecuación polinómica que nos sirva para poder predecir el número de ciclos dado un cierto número de soluciones. Usamos esta técnica para intentar predecir el número de iteraciones que serían necesarias para conseguir 92 soluciones.

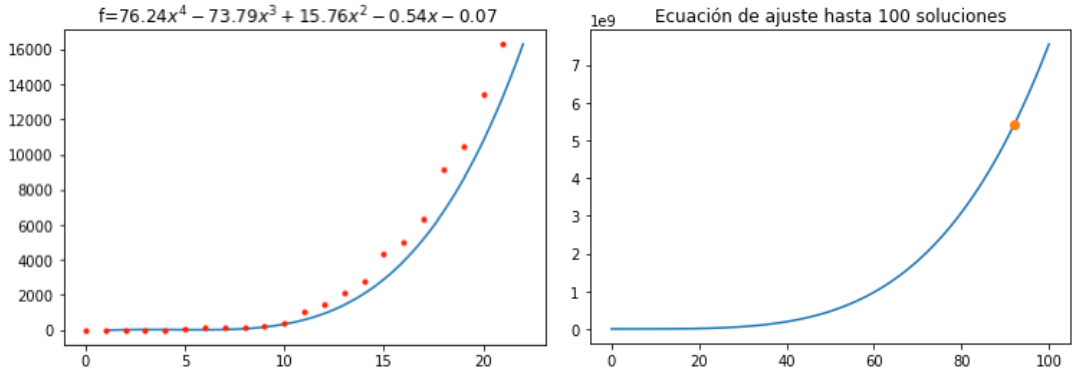

Figura 10: Ecuación de ajuste con los datos previamente representados

Si sustituimos el número de soluciones deseadas (92) en la ecuación anterior, obtenemos un valor de $5{,}40445 * 10^9$ ciclos. Teniendo en cuenta que el tiempo medio por ciclo es de 0,01075124740600586 segundos, el tiempo estimado para hallar 92 soluciones en un tablero de $8x8$ con nuestro algoritmo genético de codificación ordenada (en la primera versión) hubiera sido alrededor de 16 140 horas (672,5 días, o 1,84 años, según se prefiera).

Comparando estos resultados con la segunda versión de nuestro algoritmo genético de codificación ordenada (el cual nos da las 92 soluciones para $N = 8$ en menos de 20 segundos), podemos ver que la diferencia es más que considerable. La causa de esto era el estancamiento, ya que nuestro genético era muy dado a quedarse en mínimos o máximos locales (en función de la función de evaluación), y le costaba mucho avanzar hacia otras soluciones.

Nota: los valores de ciclos, tiempo y evaluaciones en el A.G. con codificación ordenada de la tabla se han obtenido ejecutando el algoritmo varias veces con el fin de reducir aleatoriedad en los tiempos de búsqueda. En el algoritmo de fuerza bruta no ha hecho falta, y en el binario ha sido imposible.

5. Conclusiones, Problemas encontrados y Opiniones personales

5.1. Conclusiones

Como hemos podido ver en los gráficos anteriores ilustrando las tres técnicas aplicadas a nuestro problema, la implementación final del algoritmo genético con los métodos y parámetros aplicados (selección repetida por torneo con muestra reducida, cruce ordenado simple, reemplazo general del 60-80 %, y mutación controlada con índice de diversidad) encuentra soluciones para diversos valores de N en un número reducido de iteraciones/ciclos/generaciones, quedando muy por encima de los resultados con la codificación binaria.

Desgraciadamente, al ser un método más complejo y requerir más tiempo de ejecución, para valores grandes de N, si bien encuentra solución en pocos ciclos, el tiempo de ejecución puede llegar a ser de horas. En esto gana la implementación por fuerza bruta, la cual encuentra soluciones para valores mucho mayores de N con un ordenador con unas especificaciones medias a día de hoy (Intel Core i5-7360U 2.3-3.6 GHz y 8 GB de RAM a 2133 MHz).

Esta aproximación al problema puede ser buena cuando la evaluación de la función de fitness tarde mucho tiempo (por el problema a tratar) y el cuello de botella no sea el tiempo de ejecución del código.

5.2. Problemas encontrados

A lo largo del desarrollo de la práctica, hemos encontrado los siguientes problemas:

- Estancamiento: con la mayoría de los métodos y mecanismos genéticos, la población era muy dada a estancarse muy rápidamente. El valor de *fitness* del mejor individuo de la población era muy dado a encontrarse por encima de $F = 0{,}9$ (siendo $F = f, 0 \leq f \leq 1$) antes de las 2000 iteraciones. La inclusión de un índice de diversidad solucionó parcialmente este problema.

  A continuación se listan los métodos de cruce aplicacados en la codificación ordenada, de menor a mayor estancamiento:

  - cruce SCX aleatorio
  - cruce SCX no ordenado
  - cruce SCX

- Lentitud de ejecución del código: la complejidad del código del Algoritmo Genético con codificación ordenada (en especial la selección L veces sobre muestras de tamaño K, y la evaluación de *fitness* para toda la población) conllevaba una lentitud en la ejecución considerable, en especial para tamaños grandes de P (tamaño de población) y L (tamaño de hijos a generar, relacionado con la selección, el cruce y el reemplazo generacional). La solución propuesta consiste en paralelizar de manera concurrente la ejecución de estos dos métodos (selección y evaluación) para intentar aprovechar los recursos de la máquina y agilizar la ejecución. Lamentablemente, los resultados no ven una mejora sustancial.

- Técnicas de cruce para codificaciones ordenadas: las técnicas de cruce aprendidas en clase (cruce multipunto, cruce simple), no sirven necesariamente para nuestra codificación ordenada, ya que había que aplicar restricciones en la generación del individuo o a posteriori (una vez generado el individuo). Por lo tanto, se aplican varias técnicas de cruce para codificaciones ordenadas o permutaciones, detalladas en la sección correspondiente de métodos de cruce.

5.3. Opiniones personales

Si bien la implementación de un algoritmo genético que solucione nuestro problema es relativamente trivial, el trabajo de investigación sobre las diferentes técnicas de Algoritmos Evolutivos y parámetros a utilizar, no lo es. Personalmente, aunque he disfrutado en la realización de esta práctica, considero que el volumen de trabajo necesario está a la altura de una práctica final. Quizás hubiera sido interesante ahorrarnos la codificación binaria, que ya sabíamos que no iba a dar buenos resultados, de cara a aligerar el tiempo de realización de esta práctica.

Creo que esta práctica nos ha servido para aprender mucho acerca de Algoritmos Genéticos y su aplicación, y en general estoy satisfecho con los resultados obtenidos.