Algoritmos Genéticos y Evolutivos

Práctica 1: N Reinas.

Autor:

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

22 de octubre de 2017

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.

2.1. ¿Cuánto se tarda en obtener la primera solución?

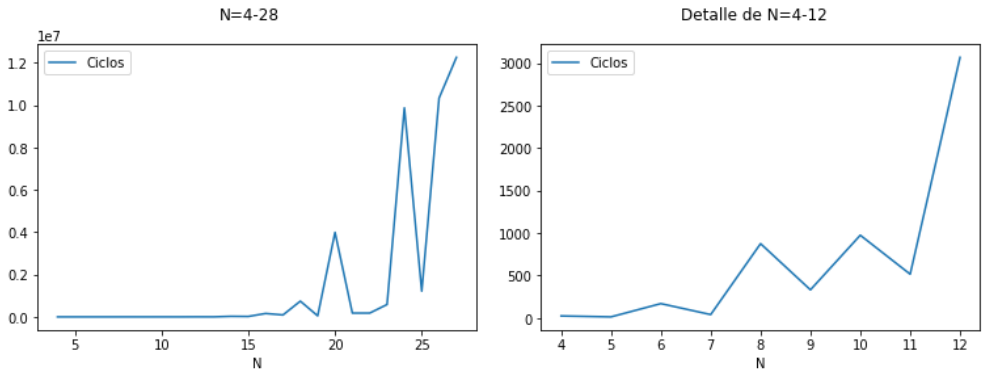

Figura 1: Número de iteraciones hasta encontrar 1 solución

2.2. ¿Cuánto se tardan en obtener soluciones subsecuentes?

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 ints 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 substitución 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 "suerteque 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 locales y la población se estancaba muy pronto.

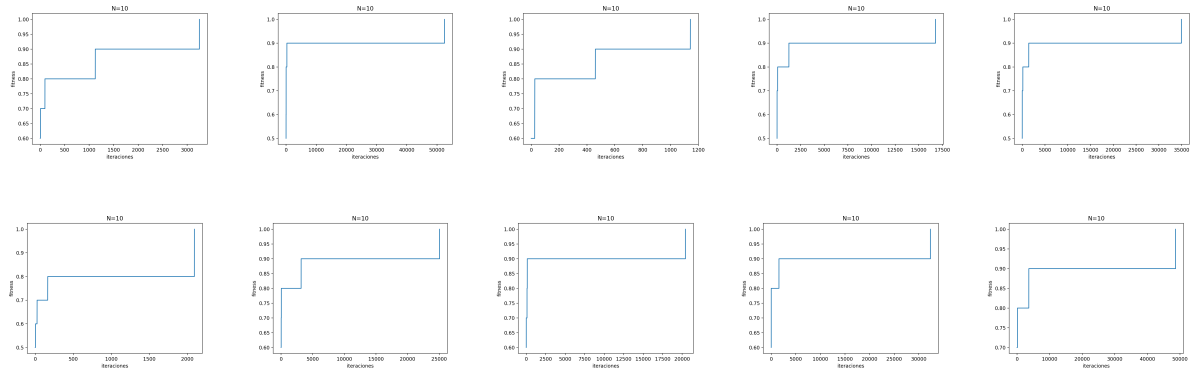

Figura 2: Parámetros De Jong para diferentes valores de N

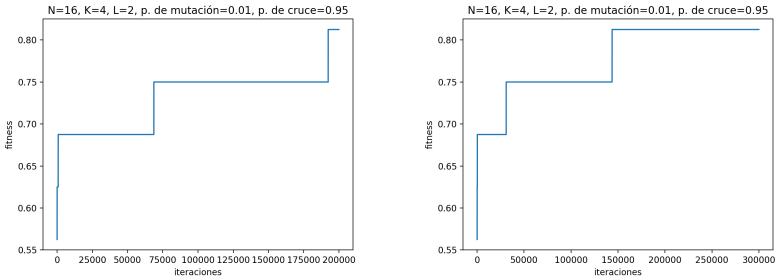

Figura 3: Parámetros Grefenstette

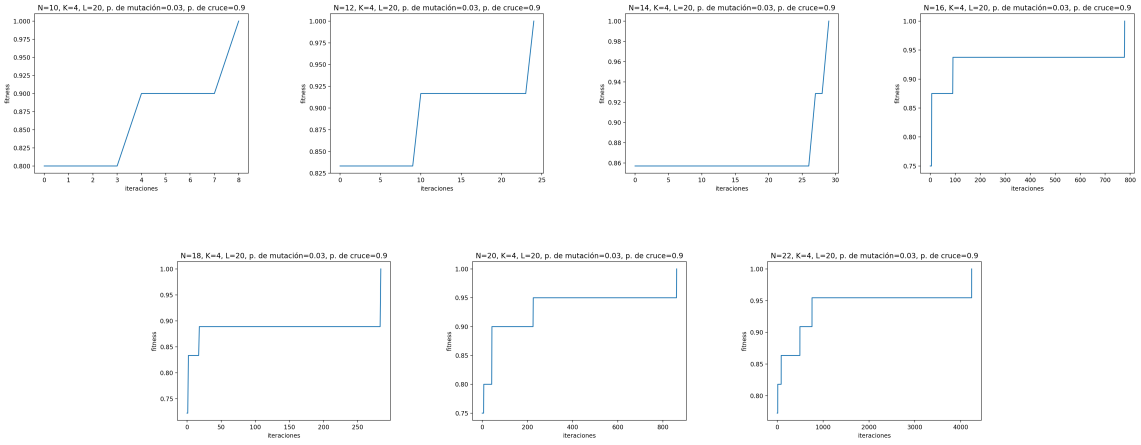

Figura 4: Parámetros propios

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.

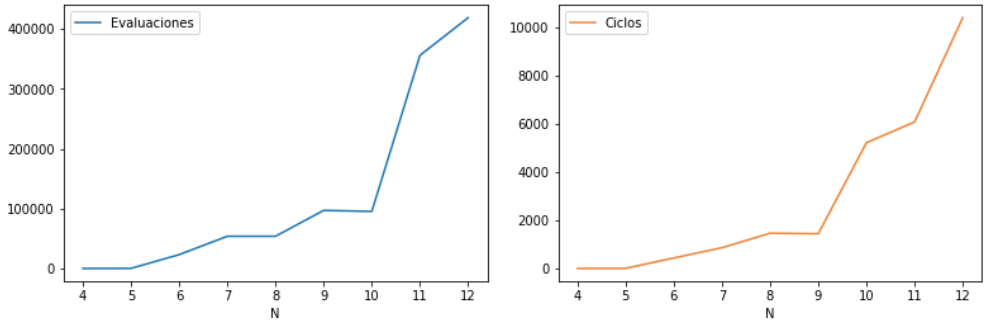

Figura 5: Codificación binaria

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.

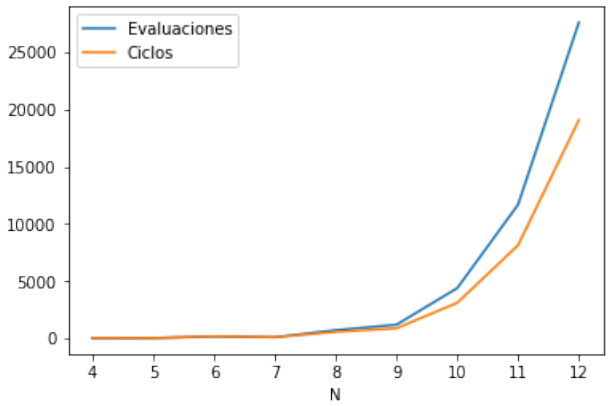

Figura 6: Primera versión con codificación ordenada

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:

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

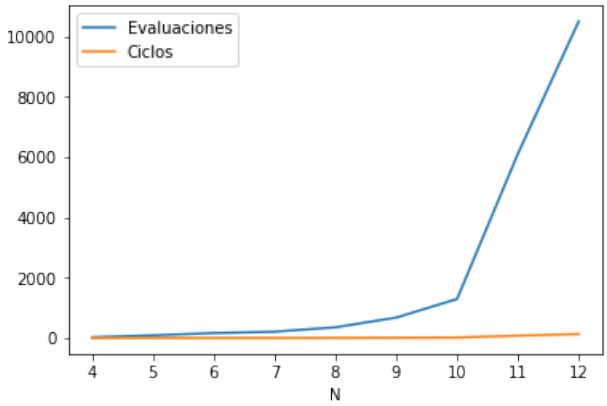

Figura 7: Versión final

4. Ampliación

4.1. Todas las soluciones

Probar con los tres métodos.

5. Conclusiones, Problemas Encontrados y Opiniones Personales

5.1. Conclusiones

La memoria debe finalizar con unas conclusiones en las que resumáis los resultados de vuestro análisis: ¿qué técnica funciona mejor?, ¿cómo afecta a los resultados vuestra codificación mejorada?, ¿cómo escala cada técnica a valores elevados de N?, etc.

5.2. Problemas encontrados

- Estancamiento. De menor a mayor estancamiento en cruce:
  - cruce SCX aleatorio
  - cruce SCX no ordenado
  - cruce SCX
- Lentitud de ejecución del código. Solución propuesta: paralelizar. Resultados? no.

5.3. Opiniones personales