Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Algoritmo Genético

N-Queens

Alumno: Daniel Soto
Profesor: Alexandre Bergel
Auxiliar: Juan-Pablo Silva
Ayudantes: Alonso Reyes
Gabriel Chandia

Fecha de entrega: 2 de diciembre de 2018
Santiago, Chile

Código Fuente

Todo el código fuente de este algoritmo se encuentra en este repositorio.

Problema

El problema elegido para aplicar el algoritmo genético fue el de N-Queens, donde se pide encontrar una configuración de N reinas en un tablero de $N \times N$ casillas, donde no haya ninguna reina atacando a otra, según las reglas del ajedrez. Cada individuo del modelo creado representa a una posible solución del problema, donde el individuo mantiene las N posiciones distintas donde irían las reinas. Como funcioń de fitness se elegió una bastante intuitiva, que cuenta el número de ataques posibles entre las reinas y lo retorna negado. Esta función puede tener valores entre $-(N^2 - N)$ y 0, donde un individuo con fitness 0 representa una solución correcta al problema.

Descripción del Programa

Para ejecutar el algoritmo genético, se debe correr con `Python 3` el archivo `nqueen_guesser.py` en la terminal. El programa pedirá un valor para el número de reinas que se desean ubicar en el tablero adecuado. Debido a que el problema no tiene solución para 2 o 3 reinas, el programa se cae al recibir uno de esos números. La implementación sólo usa librerías nativas de la instalación de `Python 3`. Los hiperparámetros de problema se pueden cambiar en las siguientes ubicaciones:

- *Mutation Rate*: Basta con cambiar el valor de la variable `MUTATION_RATE` en el archivo `nqueen_individual.py`.
- *Population Size*: Se debe cambiar el valor de la variable `POPULATION_SIZE` en el archivo `nqueen_guesser.py`.
- *Fitness Function*: Se debe cambiar el cuerpo la función *fitness(an_individual)*, definida en `nqueen_guesser.py`.

El hiperparámetro del número de genes por individuo no se puede cambiar directamente en el código, pues este se define en base al número de reinas que se quieren ubicar, por lo que modificarlo no tiene mucho sentido en este contexto.

El algoritmo se implementó usando una clase `Population`, que permite resolver cualquier problema con un algoritmo genético, dadas un par de funciones para poder realizar algunos pasos intrínsecos al problema. Esta clase espera el número de individuos en la población, una función sin argumentos que genere un nuevo individuo, la función de fitness[1] para usar en el modelo, y una función para filtrar la población entre cada iteración[2]. Esta última no es estrictamente relacionada a cada problema en específico, pero debido a que existen distintas maneras de filtrar una población, se implementaron en el archivo `filtering.py` tres distintas maneras de filtrar la población, las que son compatibles con la clase `Population`.

Con esta implementación base, resolver un problema se reduce a:

[1] De un único argumento, el individuo.
[2] De un único pseudo-argumento, self.

1. Hacer una clase que represente a un individuo, con un método para reproducir dos individuos en uno nuevo.
2. Hacer una función que genere un individuo nuevo.
3. Programar una función de fitness para el problema.
4. Instanciar la población con el número de individuos deseados, la función generadora, la de fitness, y alguna de filtro.
5. Repetir hasta encontrar una solución:
   a) `population.compute_fitness()`
   b) Revisar si hay solución entre los individuos.
   c) `population.filter_population()`
   d) `population.reproduce()`

Evaluación

Para evaluar el rendimiento del algoritmo y encontrar unos hiperparámetros adecuados utilizaremos cuantas generaciones demora en encontrar una solución, y cuanto tiempo se demora en aquello. Haremos repetidos experimentos variando un hiperparámetro a la vez, para acercarnos a algún máximo local de rendimiento. Los experimentos los realizaremos para el problema clásico, con 8 reinas en un tablero de 8×8, a través de 10 ejecuciones.

Para los hiperparámetros iniciales, se usó *Mutation Rate*: 0,2, *Population Size*: 100 y la función de fitness señalada al comienzo. Además, se utilizará la función de filtro de el cuarto de mayor fitness.

Mutation Rate	Generaciones	Tiempo
0,1	1447,5	19,51
0,15	414,1	4,11
0,2	321,5	3,16
0,25	534,5	5,70
0,3	743,7	7,93

Figura 1: Generaciones y tiempo promedio, variando el mutation rate del algoritmo.

Es claro que el valor óptimo que buscamos es un *Mutation Rate* de cerca del 2 %. A continuación probamos cambiando el tamaño de la población.

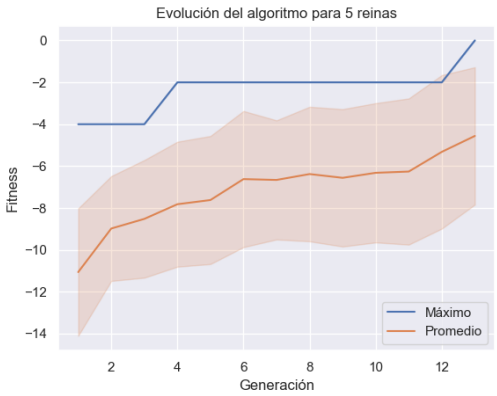
Population Size	Generaciones	Tiempo
50	883,3	4,78
100	321,5	3,16
150	696,9	12,91

Figura 2: Generaciones y tiempo promedio, variando el tamaño de la población del algoritmo.

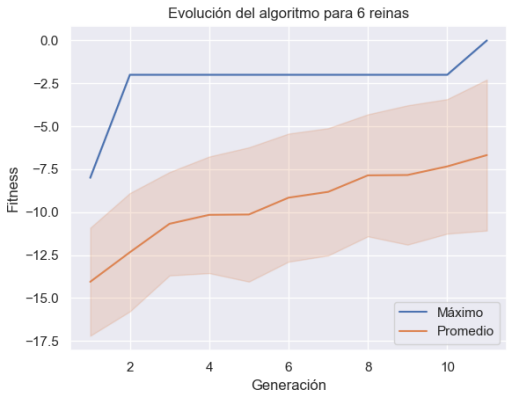
Usando el mismo tamaño de población, se intentó hacer mediciones variando el método de filtración. Los métodos alternativos, como filtro probabilístico y de torneo, no lograron obtener muy buenos resultados, y en la mayoría de los casos no encontraban una solución en un tiempo razonable. Pensamos que esto puede ser porque al ser un espacio de posibles soluciones muy grande, donde sólo unas pocas son correctas, se hace muy probable eliminar la gran mayoría de los elementos en una población dada, por lo que quedan muy pocos elementos con información útil sobre el problema, lo que implica que en cada generación se hace muy poco progreso hacia la solución. Comparando estos método con el que mantiene el cuarto de mejor fitness de la población, es claro que este conseguiría un progreso más rápido hacia la solución.

Cabe notar que el hecho de que estos métodos no funcionen bien, es probablemente algo relacionado con la naturaleza del problema, pues con un intento de solución dado, es relativamente fácil hacer cambios pequeños hasta que se encuentre una solución correcta, por lo que conviene quedarse con los individuos de mejor fitness. En problemas más complejos en cambio, podría ser útil quedarse con individuos de menor fitness, pues estos podrían traer nuevas 'estrategias' para la población, si es que el grupo de mejor fitness no puede encontrar la solución debido a que se necesitan cambios muy grandes en la solución de cada individuo. En otras palabras, en este problema hay pocos máximos locales de la función fitness que no son soluciones, pero en otros problemas podrían haber muchos máximos locales que no son soluciones, donde cambios pequeños en los genes de los individuos no logran sacarlos de esos máximos.

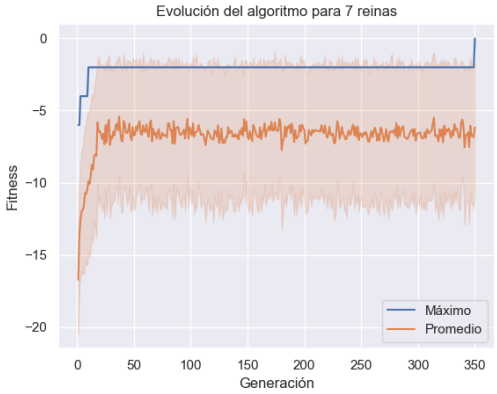
Con la configuración encontrada, se hicieron medidas más a fondo sobre la solución del problema, para distintos números de reinas. Estas medidas se presentan para ejecuciones arbitrarias del algoritmo, pues más que ver la tendencia en los resultados, queremos ver cómo se llega a éstos, es decir, cómo evoluciona la población.



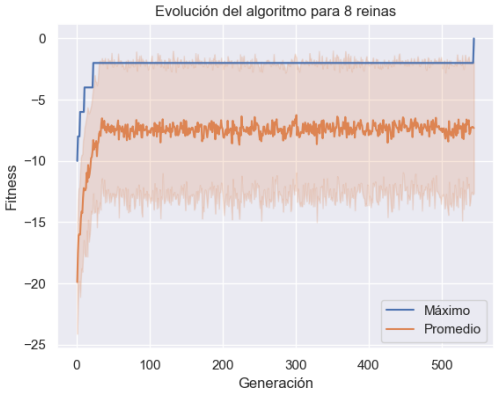
(a) 5 Reinas.



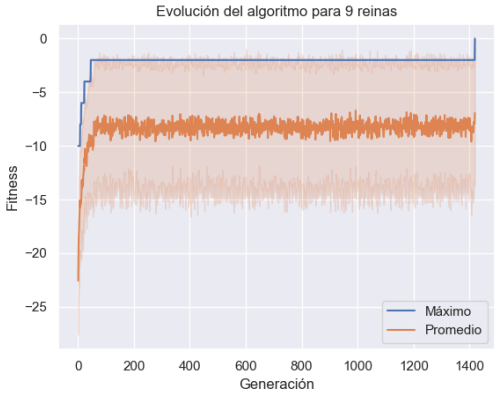
(b) 6 Reinas.



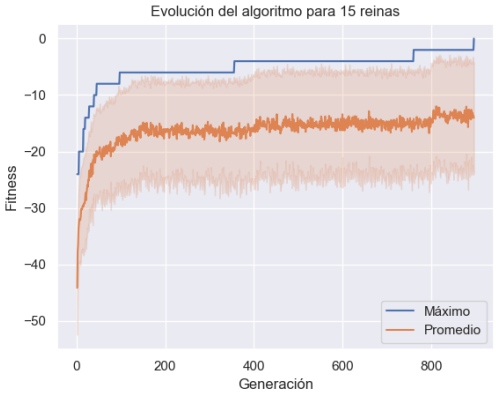
(c) 7 Reinas.



(d) 8 Reinas.



(e) 9 Reinas.



(f) 15 Reinas.

Evolución del algoritmo para distintos números de reinas

Discusión

Es claro que el algoritmo fue capaz de resolver el problema. Podemos atribuir esto a que el problema es fácil de modelar, por lo que lo individuos tienen relativamente pocos genes, los cuales son fáciles de mezclar para generar uno nuevo. Además, es fácil ir acercándose a una solución del problema cambiando las posiciones de pocas reinas a la vez, por lo que el algorimo se adapta bastante bien a lo buscado.

Un punto algo inesperado de los resultados fue que la población llega a una media rápidamente, de una varianza muy grande, donde se suele estancar hasta que un único individuo logra encontrar la solución. El resultado esperado hubiese sido que el fitness promedio de la población fuese aumentando constantemente, disminuyendo su varianza, y consiguiendo mejores intentos de soluciones en general. Una posible razón para esto es que la función de filtro, luego de cierto punto, no permitía mucha diferencia entre los miembros restantes de la población después de cada filtro. Esto, junto a que la función de reproducción es bastante básica, podría causar que los hijos de cada generación no logran tener una mejor o igual fitness que sus padres.

De todos modos, se logarron obtener resultados satisfactorios para el algoritmo, pero cabe notar que la varianza en los tiempos de ejecución es muy alta. Esto es inherente a la naturaleza aleatoria del algoritmo, así que una de las pocas mejoras posibles sería detener el algoritmo si es que demora demasiado y ejecutarlo nuevamente, con la esperanza de tener mejor suerte con la evolución de la población.