

# Tarea 2

Algoritmos genéticos

Integrantes: Matías Rojas  
Profesor: Alexandre Bergel  
Auxiliares: Juan-Pablo Silva  
Ayudantes: Alonso Reyes  
Gabriel Chandia

Fecha de realización: 30 de noviembre de 2018

Fecha de entrega: 30 de noviembre de 2018

Santiago, Chile

# Índice de Contenidos

<b>1. Introducción a los algoritmos genéticos.</b>	<b>1</b>
<b>2. Parte 1. Creando un algoritmo genético</b>	<b>2</b>
<b>3. Resultados</b>	<b>3</b>
<b>4. Parte 2. N-Queen</b>	<b>7</b>
4.1. Solución . . . . .	7
4.2. Función fitness. . . . .	8
4.3. Ejemplo para tablero de 4x4. . . . .	10
4.4. Ejemplo para tablero 6x6 . . . . .	12
4.5. Ejemplo para tablero de 10x10 . . . . .	14
<b>5. Discusión.</b>	<b>16</b>

## Lista de Figuras

1. Ejemplo de bits 1. . . . .	3
2. Ejemplo de bits 2. . . . .	3
3. Ejemplo de bits 3. . . . .	4
4. Ejemplo de bits 4. . . . .	4
5. Ejemplo de String 1. . . . .	5
6. Ejemplo de String 2. . . . .	5
7. Ejemplo de String 3. . . . .	5
8. Ejemplo de String 4. . . . .	6
9. . . . .	8
10. Fitness para NQueen . . . . .	8
11. Tablero 4x4, Población: 10. . . . .	10
12. Gráfico poblacion 10, para 4x4. . . . .	10
13. Tablero 4x4, Población: 70. . . . .	11
14. Gráfico población 70, para 4x4 . . . . .	11
15. Tablero 6x6, Población: 20. . . . .	12
16. Gráfico poblacion 20, para 6x6. . . . .	12
17. Tablero 6x6, Población: 90. . . . .	13
18. Gráfico poblacion 90, para 6x6. . . . .	13
19. Tablero 10x10, Población: 50. . . . .	14
20. Gráfico poblacion 50, para 10x10. . . . .	14
21. Tablero 10x10, Población: 150. . . . .	15
22. Gráfico poblacion 150, para 10x10. . . . .	15

El link al repositorio de GITHUB es el siguiente: [REPOSITORIO](#)

# 1. Introducción a los algoritmos genéticos.

Un algoritmo es una serie de pasos organizados que describe el proceso que se debe seguir, para dar solución a un problema específico. En particular, un algoritmo genético se basa en la evolución biológica y su base genético-molecular. Estos algoritmos tienen como fin hacer evolucionar una población de individuos sometiéndola a acciones aleatorias semejantes a las que actúan en la evolución biológica (crossover y mutaciones), así como también a una selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados.

En particular, en este curso implementaremos un algoritmo genético simple. La presente tarea se dividirá en dos partes:

1. En primer lugar, se creará un algoritmo genético base, es decir, implementaremos las funciones básicas que debe tener un algoritmo genético y que posteriormente servirá para resolver una tarea más general. En particular, para probar esta parte se probará con los siguientes problemas: Encontrar una secuencia de bits igual a la esperada, o bien dado un alfabeto encontrar un String esperado.
2. En segundo lugar, se resolverá el problema de las N reinas que se explicará más adelante.

## 2. Parte 1. Creando un algoritmo genético

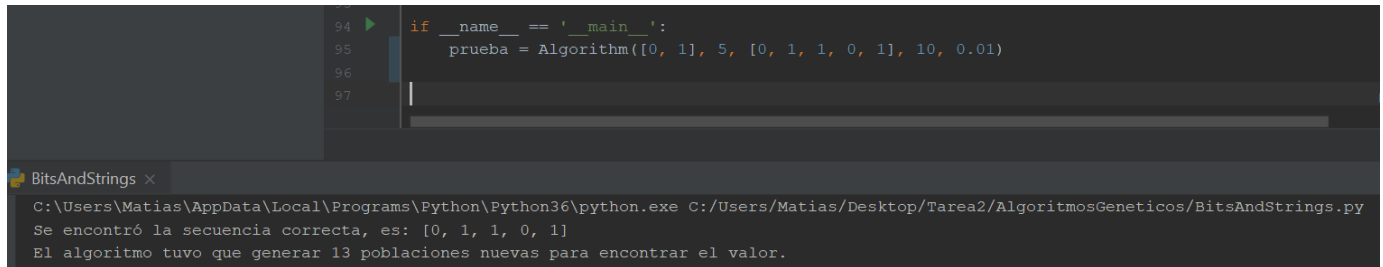
Para poder resolver esta parte de la tarea se utilizó la siguiente metodología:

1. Primero, se crea una clase denominada `Algorithm`, esta clase encapsula todo el comportamiento que posee el algoritmo, salvo la función `Fitness` ya que esta es la principal variante de problema en problema. Para poder crear un objeto de esta clase se necesitan los siguientes argumentos en el orden que viene ahora: Los genes, que son una especie de alfabeto con los posibles valores que tomará este problema, modelado correctamente, para que quede más claro, por ejemplo si sobre el alfabeto compuesto por 0 y 1 se busca la secuencia 00001111, entonces nuestros genes serían 0 y 1, ya que son los posibles símbolos que existen en nuestra búsqueda, lógicamente esto puede extenderse mucho más. Luego tenemos el número de genes que no es más que el largo de la lista anterior, luego el `expectedSequence` que corresponde al valor que estamos buscando. Para terminar, los últimos dos parámetros son `numberOfPopulation` que es la cantidad de secuencias que generaremos en cada generación, y finalmente el `MutationRate` que nos será útil al momento de realizar la mutación en la reproducción.
2. Segundo, tenemos el método `Fitness` que varía de problema en problema, en este caso se reduce a comparar si los caracteres/símbolos correspondientes de la secuencia esperada con la secuencia actual coinciden o no, esto entrega un número `fitness` que nos dice que tan acertada va la solución. Intuitivamente, en caso que este número sea igual al número de genes, el problema estará resuelto, y la palabra o secuencia habrá sido encontrada.
3. En tercer lugar, está el método auxiliar `fillFitnessList`. Su función es actualizar el `fitness` asociado a cada miembro de la población luego de cada reproducción y al comienzo también.
4. En cuarto lugar, nos encontramos con el método de `reproduction`, que es fundamental en el afán de encontrar la secuencia/palabra, ya que este se encarga de crear una población con el doble de integrantes en comparación a la anterior, lo que representará a los padres. Esto lo hace con el método del torneo explicado en cátedra. Luego se realiza un `crossover` y una mutación entre pares de padres para poder generar los nuevos miembros de la población y repetir el procedimiento.
5. Finalmente, encontramos el método `start` que se encarga, de iniciar la búsqueda del elemento deseado, para ello revisa si algún miembro de la población actual coincide con el elemento esperado, en tal caso el programa termina, de lo contrario, se inicia el proceso de reproducción y así hasta encontrar el elemento pedido.

### 3. Resultados

Para poder compilar el programa, hay que hacer lo siguiente: En el módulo llamado BitsAndStrings.py, ir hacia abajo hasta donde esté el main, en dicho lugar, hay que pasar los parametros y luego presionar run.

Los inputs con los resultados correspondientes para el problema de los bits son los siguientes:



```
94 if __name__ == '__main__':
95     prueba = Algorithm([0, 1], 5, [0, 1, 1, 0, 1], 10, 0.01)
96
97
```

BitsAndStrings ×

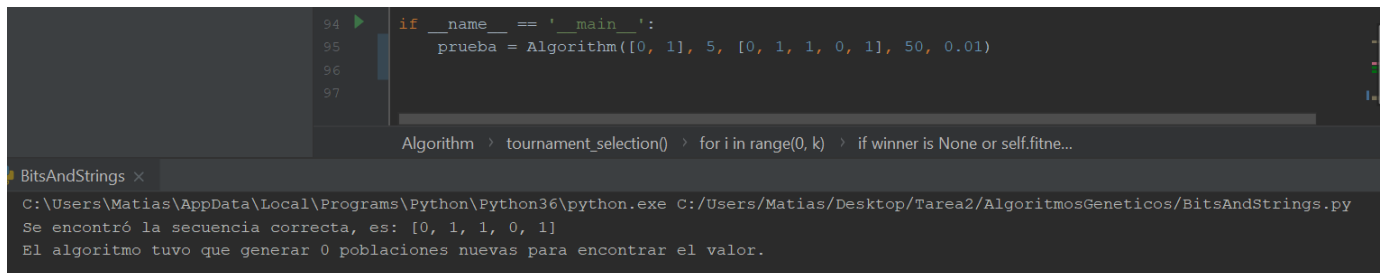
C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/AlgoritmosGeneticos/BitsAndStrings.py

Se encontró la secuencia correcta, es: [0, 1, 1, 0, 1]

El algoritmo tuvo que generar 13 poblaciones nuevas para encontrar el valor.

Figura 1: Ejemplo de bits 1.

Podemos apreciar que se encontró la secuencia luego de realizar 13 reproducciones, esto se debe a la poca cantidad de miembros de miembros en la población, si utilizamos los argumentos pero ahora con una población con 50 miembros el resultado es el siguiente:



```
94 if __name__ == '__main__':
95     prueba = Algorithm([0, 1], 5, [0, 1, 1, 0, 1], 50, 0.01)
96
97
```

Algorithm > tournament\_selection() > for i in range(0, k) > if winner is None or self.fitne...

BitsAndStrings ×

C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/AlgoritmosGeneticos/BitsAndStrings.py

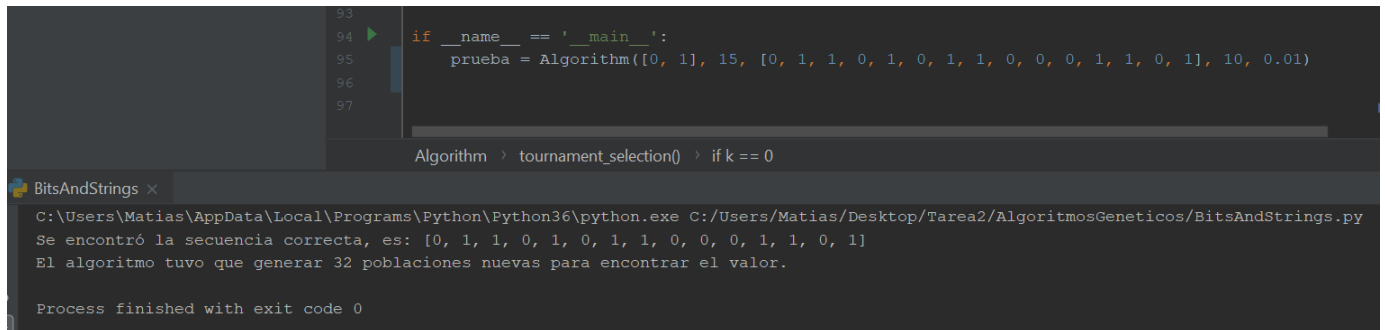
Se encontró la secuencia correcta, es: [0, 1, 1, 0, 1]

El algoritmo tuvo que generar 0 poblaciones nuevas para encontrar el valor.

Figura 2: Ejemplo de bits 2.

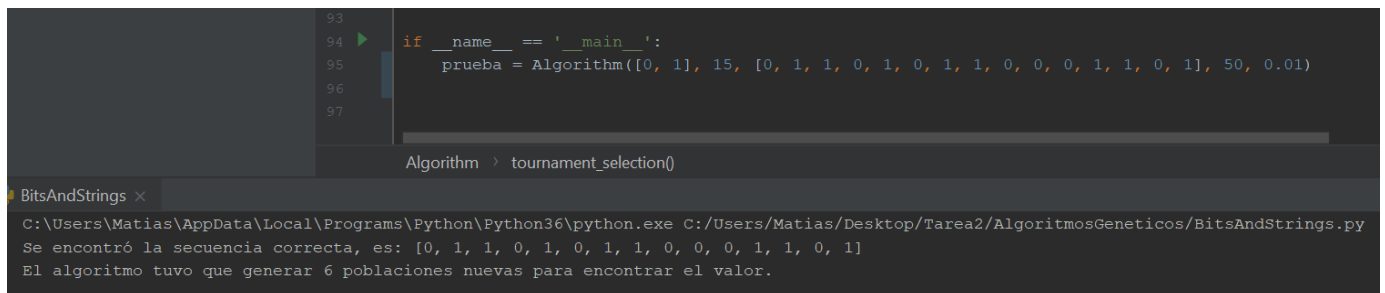
Ahora lo encuentra inmediatamente, ya que al generar 50 miembros aleatorios y con cada uno con un largo de 5, no es difícil encontrar a la primera la secuencia de 5 bits esperada.

Otros ejemplo más claro es el siguiente, para una población más grande y una secuencia esperada más grande:



```
93  
94 if __name__ == '__main__':  
95     prueba = Algorithm([0, 1], 15, [0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1], 10, 0.01)  
96  
97  
Algorithm > tournament_selection() > if k == 0  
BitsAndStrings x  
C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/AlgoritmosGeneticos/BitsAndStrings.py  
Se encontró la secuencia correcta, es: [0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1]  
El algoritmo tuvo que generar 32 poblaciones nuevas para encontrar el valor.  
Process finished with exit code 0
```

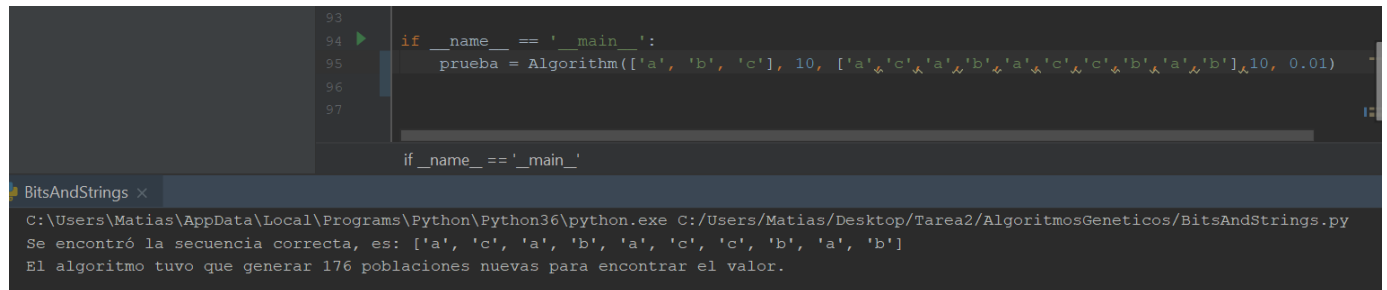
Figura 3: Ejemplo de bits 3.



```
93  
94 if __name__ == '__main__':  
95     prueba = Algorithm([0, 1], 15, [0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1], 50, 0.01)  
96  
97  
Algorithm > tournament_selection()  
BitsAndStrings x  
C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/AlgoritmosGeneticos/BitsAndStrings.py  
Se encontró la secuencia correcta, es: [0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1]  
El algoritmo tuvo que generar 6 poblaciones nuevas para encontrar el valor.
```

Figura 4: Ejemplo de bits 4.

Para el caso de encontrar un String, el problema es similar veamos los siguientes dos ejemplos en donde se muestra una palabra a encontrar sobre un alfabeto con distintas poblaciones y genes.

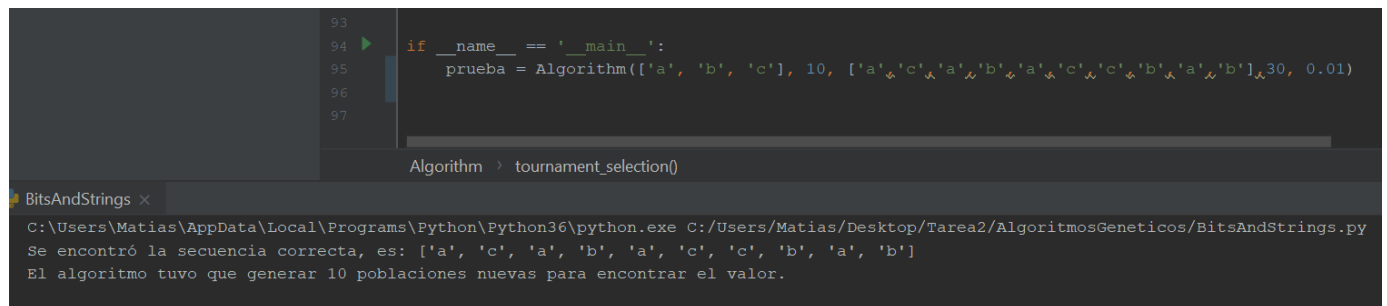


```
93
94 if __name__ == '__main__':
95     prueba = Algorithm(['a', 'b', 'c'], 10, ['a' * 10, 'a' * 9 + 'b', 'a' * 8 + 'b' + 'c', 'a' * 7 + 'b' + 'c' + 'b', 'a' * 6 + 'b' + 'c' + 'b' + 'c'], 10, 0.01)
96
97
if __name__ == '__main__':
```

BitsAndStrings x

C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/AlgoritmosGeneticos/BitsAndStrings.py  
Se encontró la secuencia correcta, es: ['a', 'c', 'a', 'b', 'a', 'c', 'c', 'b', 'a', 'b']  
El algoritmo tuvo que generar 176 poblaciones nuevas para encontrar el valor.

Figura 5: Ejemplo de String 1.

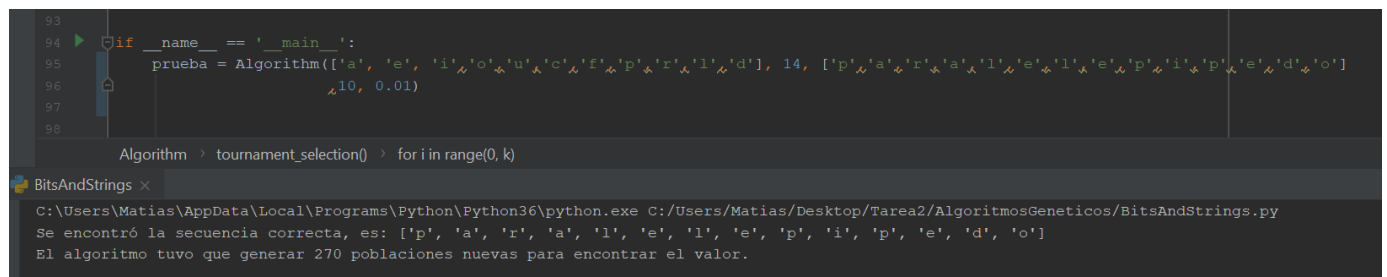


```
93
94 if __name__ == '__main__':
95     prueba = Algorithm(['a', 'b', 'c'], 10, ['a' * 10, 'a' * 9 + 'b', 'a' * 8 + 'b' + 'c', 'a' * 7 + 'b' + 'c' + 'b', 'a' * 6 + 'b' + 'c' + 'b' + 'c'], 30, 0.01)
96
97
Algorithm > tournament_selection()
```

BitsAndStrings x

C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/AlgoritmosGeneticos/BitsAndStrings.py  
Se encontró la secuencia correcta, es: ['a', 'c', 'a', 'b', 'a', 'c', 'c', 'b', 'a', 'b']  
El algoritmo tuvo que generar 10 poblaciones nuevas para encontrar el valor.

Figura 6: Ejemplo de String 2.

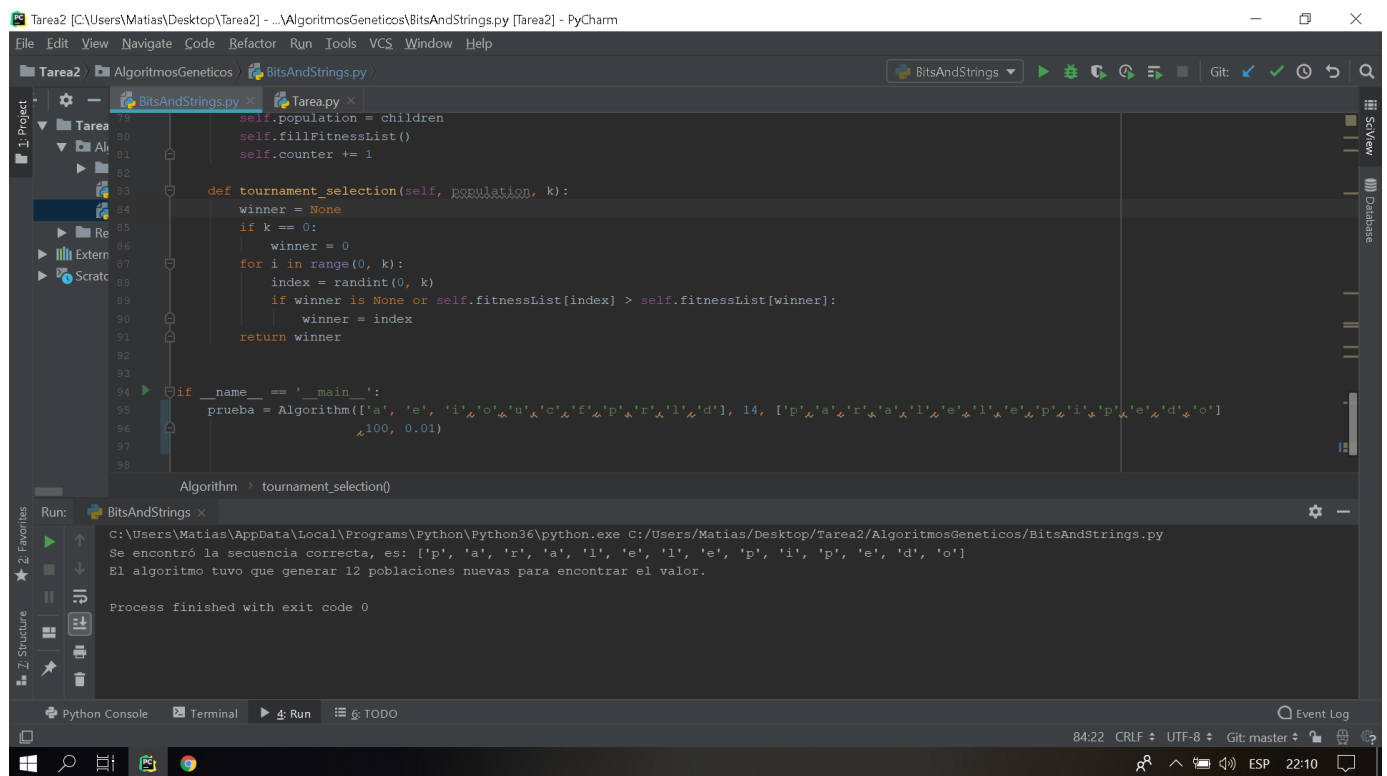


```
93
94 if __name__ == '__main__':
95     prueba = Algorithm(['a', 'e', 'i', 'o', 'u', 'c', 'f', 'p', 'r', 'l', 'd'], 14, ['p' * 14, 'p' * 13 + 'a', 'p' * 12 + 'a' + 'l', 'p' * 11 + 'a' + 'l' + 'e', 'p' * 10 + 'a' + 'l' + 'e' + 'p', 'p' * 9 + 'a' + 'l' + 'e' + 'p' + 'i', 'p' * 8 + 'a' + 'l' + 'e' + 'p' + 'i' + 'p', 'p' * 7 + 'a' + 'l' + 'e' + 'p' + 'i' + 'p' + 'e', 'p' * 6 + 'a' + 'l' + 'e' + 'p' + 'i' + 'p' + 'e' + 'd', 'p' * 5 + 'a' + 'l' + 'e' + 'p' + 'i' + 'p' + 'e' + 'd' + 'o'], 10, 0.01)
96
97
98
Algorithm > tournament_selection() > for i in range(0, k)
```

BitsAndStrings x

C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/AlgoritmosGeneticos/BitsAndStrings.py  
Se encontró la secuencia correcta, es: ['p', 'a', 'r', 'a', 'l', 'e', 'l', 'e', 'p', 'i', 'p', 'e', 'd', 'o']  
El algoritmo tuvo que generar 270 poblaciones nuevas para encontrar el valor.

Figura 7: Ejemplo de String 3.



```
Tarea2 [C:\Users\Matias\Desktop\Tarea2] - ...\AlgoritmosGeneticos\BitsAndStrings.py [Tarea2] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help

Tarea2 | AlgoritmosGeneticos | BitsAndStrings.py
1 self.population = children
2 self.fillFitnessList()
3 self.counter += 1
4
5 def tournament_selection(self, population, k):
6     winner = None
7     if k == 0:
8         winner = 0
9     for i in range(0, k):
10         index = randint(0, k)
11         if winner is None or self.fitnessList[index] > self.fitnessList[winner]:
12             winner = index
13     return winner
14
15 if __name__ == '__main__':
16     prueba = Algorithm(['a', 'e', 'i', 'o', 'u', 'c', 'f', 'p', 'r', 'l', 'd'], 14, ['p', 'a', 'r', 'a', 'l', 'l', 'e', 'l', 'p', 'i', 'p', 'e', 'd', 'o'])
17     prueba.run(100, 0.01)
18
Algorithm > tournament_selection()

Run: BitsAndStrings x
C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:\Users\Matias\Desktop\Tarea2\AlgoritmosGeneticos\BitsAndStrings.py
Se encontró la secuencia correcta, es: ['p', 'a', 'r', 'a', 'l', 'l', 'e', 'l', 'p', 'i', 'p', 'e', 'd', 'o']
El algoritmo tuvo que generar 12 poblaciones nuevas para encontrar el valor.

Process finished with exit code 0

Python Console Terminal Run TODO
84:22 CRLF UTF-8 Git: master ESP 22:10
```

Figura 8: Ejemplo de String 4.

Cabe mencionar en todos estos ejemplos, que la métrica utilizada para medir cuanto tarda el algoritmo en generar el resultado esperado es la cantidad de generaciones que se reproducen para llegar al resultado. Se puede apreciar comparando en las imágenes que es una buena métrica para dilucidar el efecto del número de población y cantidad de genes en el algoritmo.



## 4. Parte 2. N-Queen

Ahora que ya se implementó el algoritmo, hay que aplicarlo a algo más concreto que una búsqueda de símbolos y caracteres. Para ello, como tarea se elige resolver el problema N-Queen, este es un famoso problema algorítmico que consiste en lo siguiente:

Dado un tablero de ajedrez de N por N casillas, ubicar N reinas de forma ninguna pueda atacar a otra. Para ello, cabe mencionar que una reina puede atacar a cualquier otra ficha de ajedrez ( en particular, a una reina) si es que esta ficha se encuentra en la misma columna, en la misma fila, o en alguna de las dos diagonales que cruzan a la ficha.

### 4.1. Solución

Para implementar la solución al problema, cabe mencionar que se creará un nuevo módulo en Python llamado Tarea.py esto ya que hay que hacer unas modificaciones a la clase Algorithm presentada anteriormente. Pero en general, la lógica sigue siendo la misma. Lo que se hizo fue:

- a) En primer lugar, los únicos parámetros que tendrá el objeto de tipo Algorithm creado serán: El número de genes, la cantidad de población y el mutation rate. En particular eliminamos, los tipos de genes ya que en este caso no son necesarios ya que en la función init se creará aleatoriamente la población en función del número de genes. Y también se elimina el expectedSequence ya que ese es el trabajo que debe hacer el algoritmo, si supieramos todas las soluciones no habría caso realizar esto.
- b) Entonces, los miembros de la población, que serán las posiciones en que se deben ubicar las reinas se verán como una lista con una notación en particular, por ejemplo en un tablero de 2 por 2:

3, 0, 2, 1

representa que en la primera fila, en la última posición estará ubicada una reina, en la segunda fila estará en la primera posición, en la tercera fila en la tercera posición y en la última fila estará en la segunda posición. Notar que los índices parten de 0, pero esta notación simplificará la vida al momento de implementar las funciones. Entonces, al momento de crear el Algoritmo, lo que se hace es crear una lista de miembros de la población, donde cada uno será una lista con números al azar entre 0 y el número de genes menos 1. Para ello se utiliza la función choice. El resto de la clase queda exactamente igual, lo que cambia totalmente es la función fitness que se utilizará ahora y que describirá a continuación.

## 4.2. Función fitness.

En la siguiente imagen se puede apreciar la función:

```
# First we create de function that tells how many characters are the same.
def queenFitnessCalcule(currentSequence):
    lenght = len(currentSequence)
    matrix = np.zeros((lenght, lenght))
    for i in range(lenght):
        aux = currentSequence[i]
        matrix[i][aux] = 1
    result = lenght
    exit = 0
    # filas y columnas
    for i in range(lenght):
        for j in range(lenght):
            if matrix[i][j] == 1:
                for k in range(j + 1, lenght):
                    if matrix[i][k] == 1:
                        exit = 1
                    else:
                        continue
                for k in range(i + 1, lenght):
                    if matrix[k][j] == 1:
                        exit = 1
            else:
                continue
        if exit == 1:
            result -= 1
        exit = 0
```

Figura 9

```
for i in range(lenght):
    for j in range(i + 1, lenght):
        if abs(currentSequence[j] - currentSequence[i]) == abs(j - i):
            result -= 1
return result
```

Figura 10: Fitness para NQueen

En primer lugar, lo que se hace en el algoritmo es recibir como parámetro una lista que corresponde a la representación de un individuo de la población es decir, una lista que representa las posiciones en que esta ubicada la reina según la descripción dada anteriormente. Luego, se crea una matriz con puros 0s que representará al tablero, en ella llenamos con 1 en los espacios donde las reinas estén ubicadas y pasamos al paso más importante:

Como se dijo en la descripción del problema, primero veremos que no hayan dos o más 1s tanto en las filas como en las columnas correspondientes a un individuo. En caso que esto si ocurra, lo que se hace es restarle uno a la variable result y pasar al siguiente paso que es ver que no hayan 2 o más 1s en cada una de las diagonales que atraviesan a la reina. Para ello notemos que la condición encontrada para que esto no ocurra, es que en la lista entregada no puede pasar que la resta de dos elementos cualquiera de la lista sea igual a la resta de los índices en que están ubicados en la lista. Cabe destacar aca que la lista no es circular por lo que la distancia entre el primer y último elemento no es 1.

Finalmente, en el método start se agregaron dos elementos nuevos: En primer lugar se realiza un print de la matriz que representa al tablero con las reinas ubicadas correctamente, es decir la solución gráfica del problema. Y en segundo lugar, se realiza un gráfico donde se muestra el mejor fitness (número entero) en cada generación hasta encontrar al máximo correspondiente al largo total de los genes.

En las siguientes páginas se pueden ver ejemplos de ejecución del programa, junto con los respectivos gráficos. Para ejecutar el código es análogo al caso anterior solo que hay que pasar menos parámetros. Omitiremos el tablero de 1x1 ya que la solución es trivial, veremos casos más interesantes.

### 4.3. Ejemplo para tablero de 4x4.

a) Para una población de 10 individuos:

```
139 ▶ if __name__ == '__main__':  
140     Algorithm(4, 10, 0.01)  
141
```

Algorithm > start() > while True > for k in range(len(self.populat... > if self.fitnessList[k] == sel

Tarea x

C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/AlgoritmosGeneticos/Tarea.py  
Se encontró la secuencia correcta, es: [2, 0, 3, 1]  
El algoritmo tuvo que generar 139 poblaciones nuevas para encontrar el valor.  
La siguiente es la representación gráfica, los 1's marcan las posiciones de las reinas  
[[0. 0. 1. 0.]  
 [1. 0. 0. 0.]  
 [0. 0. 0. 1.]  
 [0. 1. 0. 0.]]

Figura 11: Tablero 4x4, Población: 10.

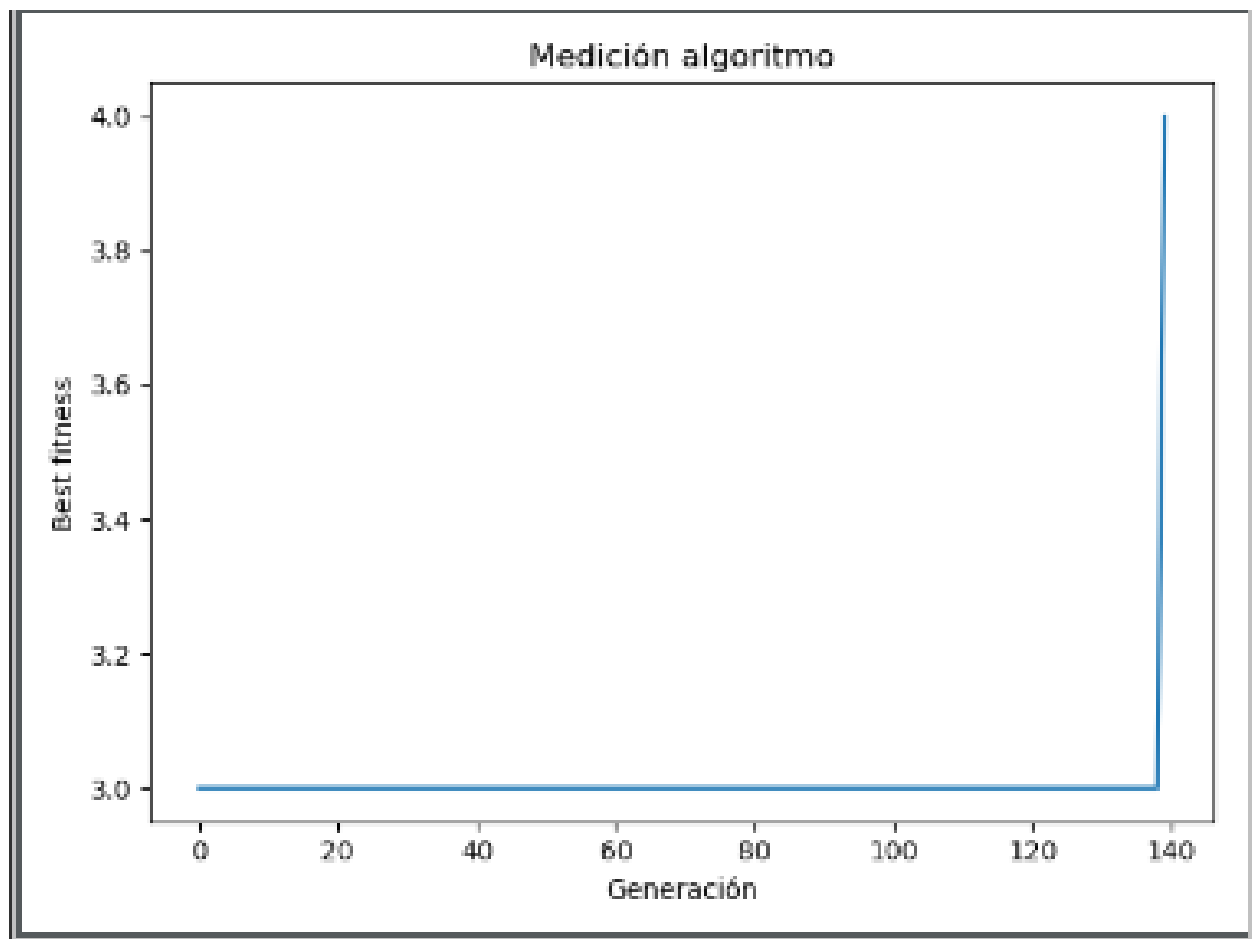


Figura 12: Gráfico poblacion 10, para 4x4.

b) Para una población de 70 individuos:

```

138
139
140
141
if __name__ == '__main__':
    Algorithm(4, 70, 0.01)

```

Algorithm > start() > while True > for k in range(len(self.populat... > if self.fitnessList[k] == se

Tarea x

C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/AlgoritmosGeneticos/Tarea.py

Se encontró la secuencia correcta, es: [2, 0, 3, 1]

El algoritmo tuvo que generar 1 poblaciones nuevas para encontrar el valor.

La siguiente es la representación gráfica, los 1's marcan las posiciones de las reinas

```

[[0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 1. 0. 0.]]

```

Figura 13: Tablero 4x4, Población: 70.

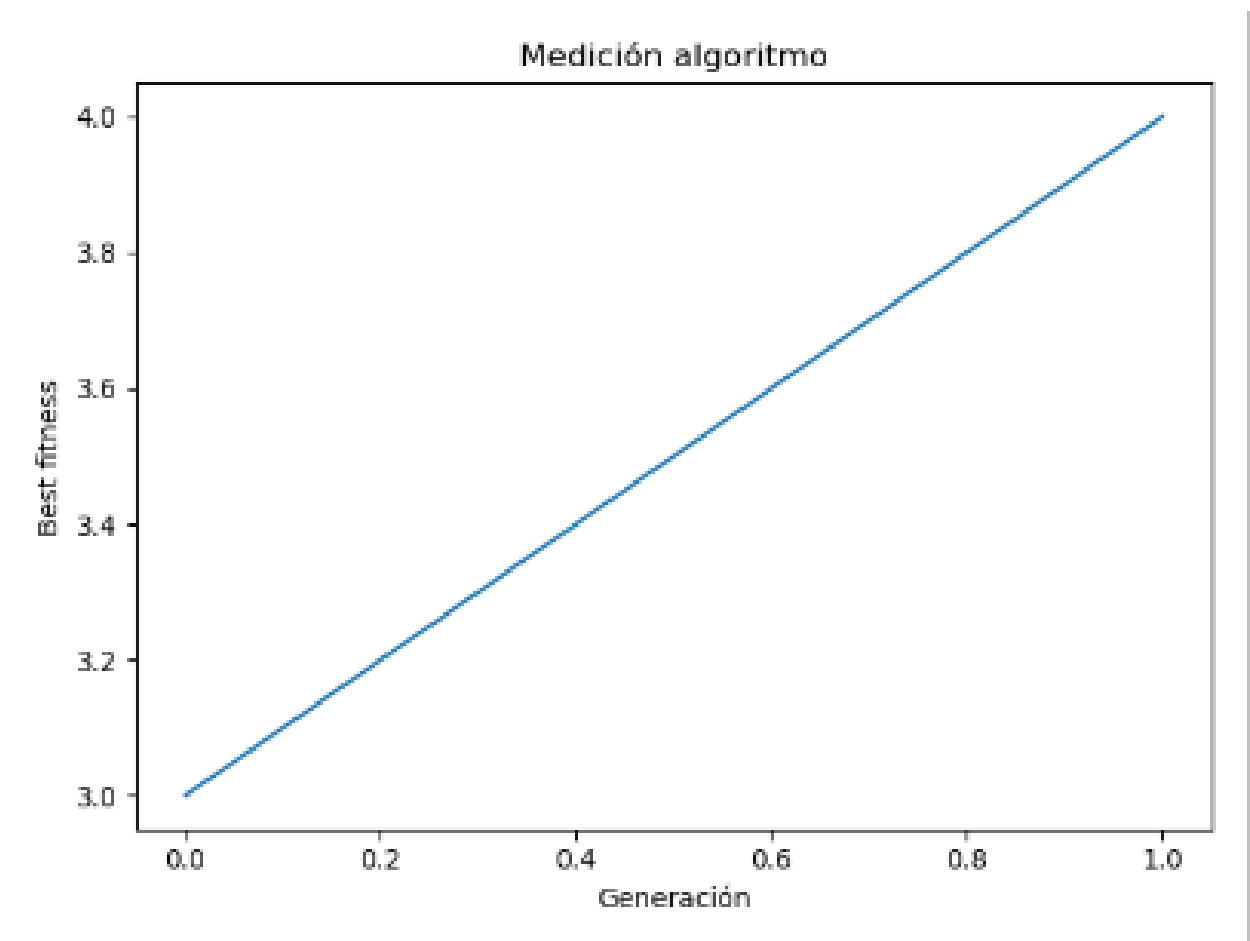
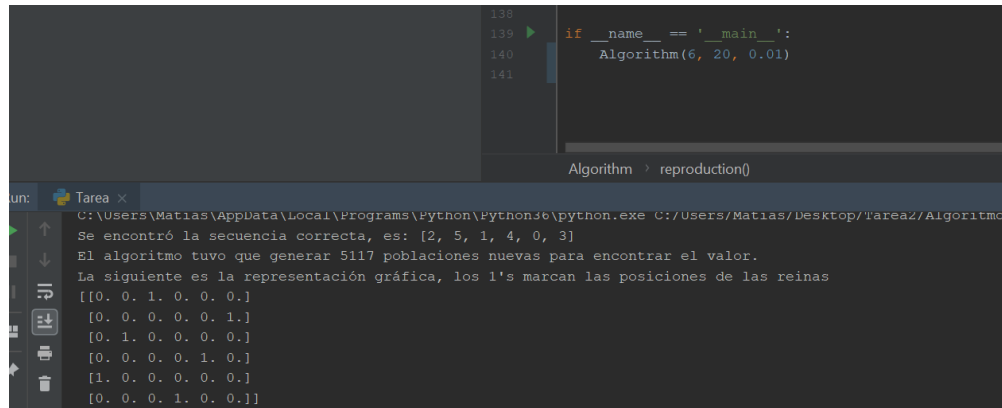


Figura 14: Gráfico población 70, para 4x4

## 4.4. Ejemplo para tablero 6x6

1) Para 20 individuos:



```

138
139
140
141
if __name__ == '__main__':
    Algorithm(6, 20, 0.01)

```

Algorithm > reproduction()

C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/Algoritmos

Se encontró la secuencia correcta, es: [2, 5, 1, 4, 0, 3]

El algoritmo tuvo que generar 5117 poblaciones nuevas para encontrar el valor.

La siguiente es la representación gráfica, los 1's marcan las posiciones de las reinas

```

[[0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]]

```

Figura 15: Tablero 6x6, Población: 20.

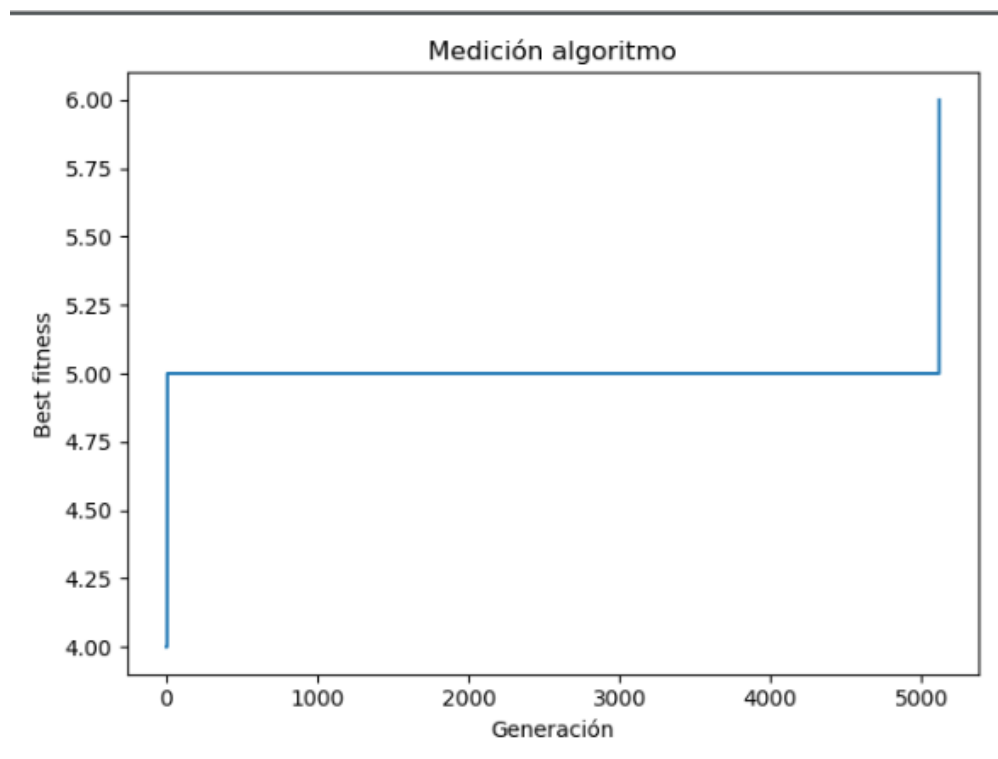


Figura 16: Gráfico poblacion 20, para 6x6.

2) Para 90 individuos:

```
atches and Consoles
139
140
141
Algorithm > tournament_selection() > for i in range(0, k)

Tarea x
Se encontró la secuencia correcta, es: [1, 3, 5, 0, 2, 4]
El algoritmo tuvo que generar 3 poblaciones nuevas para encontrar el valor.
La siguiente es la representación gráfica, los 1's marcan las posiciones de las reinas
[[0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]]
```

Figura 17: Tablero 6x6, Población: 90.

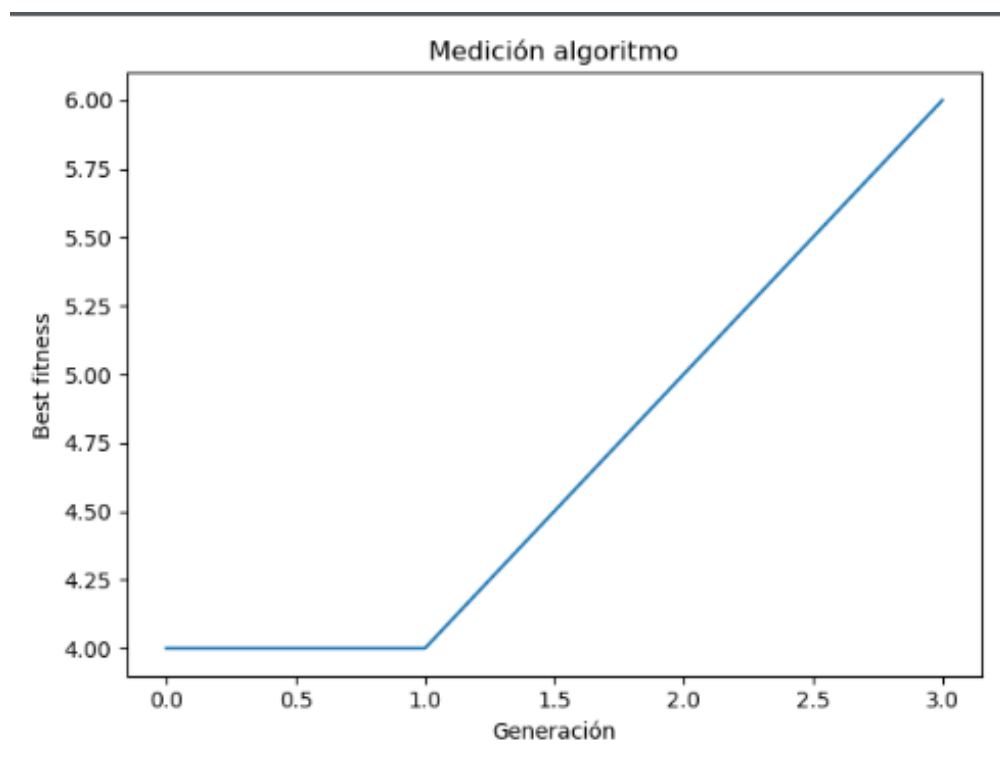
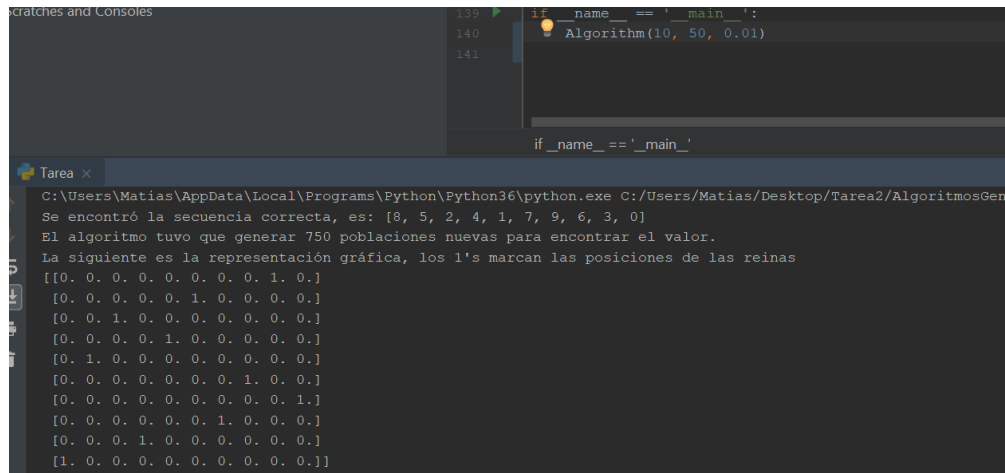


Figura 18: Gráfico poblacion 90, para 6x6.

## 4.5. Ejemplo para tablero de 10x10

1) Para 50 individuos:



```
139 if __name__ == '__main__':
140     Algorithm(10, 50, 0.01)
141
if __name__ == '__main__':

C:\Users\Matias\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Matias/Desktop/Tarea2/AlgoritmosGen
Se encontró la secuencia correcta, es: [8, 5, 2, 4, 1, 7, 9, 6, 3, 0]
El algoritmo tuvo que generar 750 poblaciones nuevas para encontrar el valor.
La siguiente es la representación gráfica, los 1's marcan las posiciones de las reinas
[[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Figura 19: Tablero 10x10, Población: 50.

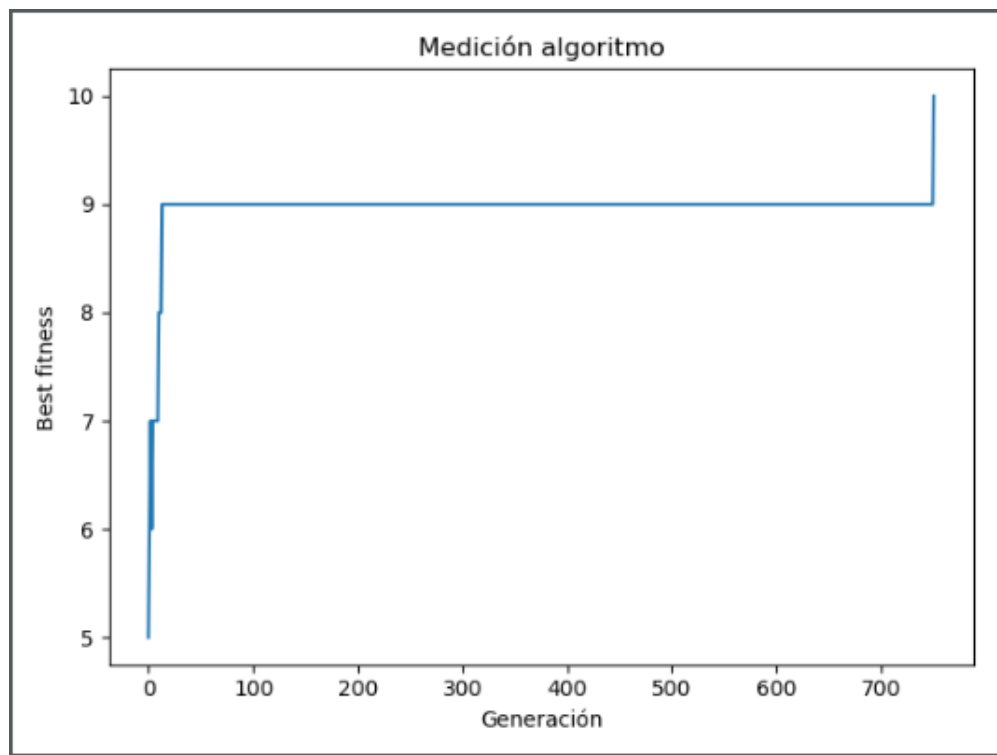


Figura 20: Gráfico poblacion 50, para 10x10.



2) Para una población de 150 individuos:

```
Red Neuronal
Internal Libraries
atches and Consoles

139  if __name__ == '__main__':
140      Algorithm(10, 150, 0.01)
141

Algorithm > fillFitnessList()

Tarea x
Se encontró la secuencia correcta, es: [7, 4, 1, 9, 2, 6, 8, 3, 5, 0]
El algoritmo tuvo que generar 95 poblaciones nuevas para encontrar el valor.
La siguiente es la representación gráfica, los 1's marcan las posiciones de las reinas
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Figura 21: Tablero 10x10, Población: 150.

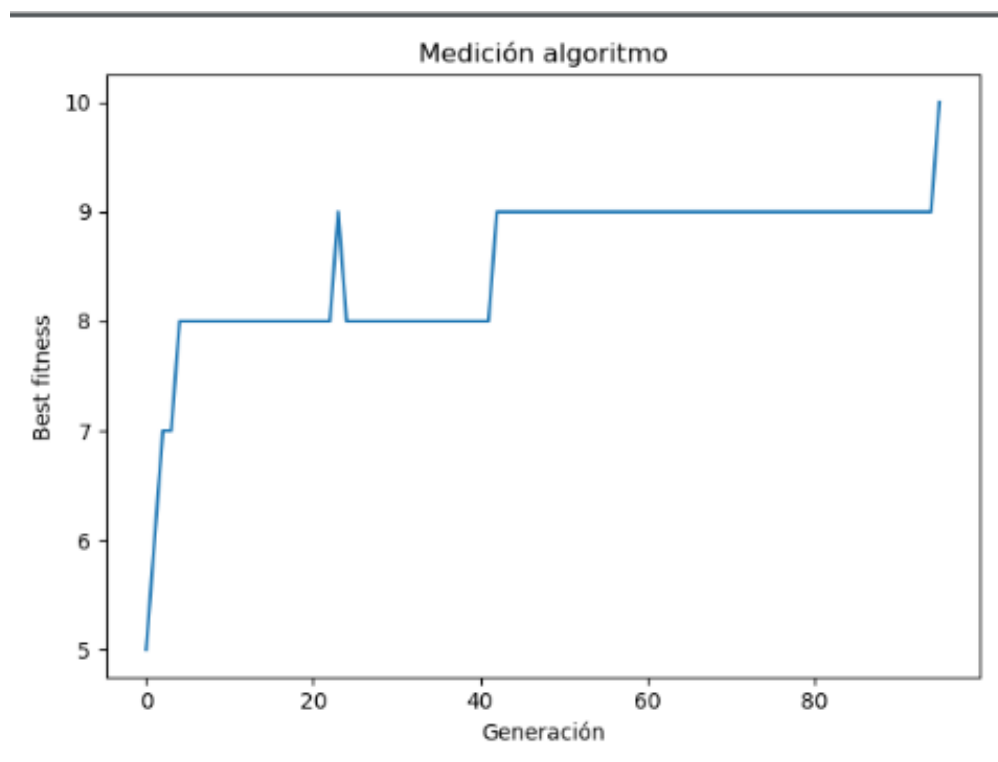


Figura 22: Gráfico poblacion 150, para 10x10.

## 5. Discusión.

Para concluir con este trabajo se pueden realizar las siguientes reflexiones:

Los algoritmos implementados tanto para los problemas básicos como la búsqueda de la secuencia o del string y la tarea realizada sobre el problema N-Queen funcionan correctamente y cumplen con el objetivo planteado.

Los factores que más influenciaron en el éxito del proyecto fueron en primer lugar, crear una especie de molde con el primer algoritmo para luego solo extenderlo en la tarea de las reinas, bastaba solo con modificar la función de fitness. En segundo lugar, el hecho de recibir pocos parámetros al momento de crear un objeto Algorithm da una simplicidad al programa y permite apreciar fácilmente las variaciones de las soluciones a medida que se cambian los parámetros.

En cuanto al algoritmo en sí del NQueen, se puede apreciar que en un tiempo no muy prolongado se puede dar solución a algo que tomaría horas o quizás días para un ser humano, lo que es muy interesante. En cuanto a cosas más técnicas, con los ejemplos mostrados se puede concluir que a medida que se aumenta la población, el tiempo de búsqueda exitosa del algoritmo es pronunciadamente menor. Como también, influye la cantidad de genes, ya que a medida que hay más genes más tiempo le cuesta al computador resolver el problema.

En general, se logró un buen trabajo y entendimiento sobre cómo funcionan los algoritmos genéticos y como simulan la biología en general. Como acotaciones, se puede decir que sería interesante extender este algoritmo y entregar más métricas, es decir, realizar más gráficos sobre el comportamiento del algoritmo, puede ser variar el MutationRate o comparar los fitness entre distintas generaciones, para generar otras medidas como varianza o bien una correlación entre los datos.