CC5114 - T2

Algoritmos Genéticos

Nombre: Ariel Suil A.
Repositorio: asuil/CC5114-Neural
Profesor: Alexandre Bergel
Auxiliar: Ignacio Slater M.
Entrega: 01/12/20

1. Motor del Algoritmo Genético

El motor del algoritmo es el encargado de, a partir de un modelo de individuo y función de fitness, manejar generaciones de individuos en busca de el ideal para la resolución de un problema.

1.1. Estructura General

El algoritmo en términos generales comienza creando una población aleatoria, luego dentro de un loop, obtiene sus resultados de fitness, checkea si ya terminó, elige los individuos a ser padres, crea la siguiente generación, y repite el loop hasta terminar.

El algoritmo puede terminar por dos razones: El resultado de fitness del mejor individuo cumple con una condición dada, o se cumplió el máximo de generaciones.

1.2. Decisiones Particulares

La implementación del algoritmo es bastante predecible en la mayoría de áreas excepto al momento de crear una nueva generación a partir de la generación anterior. En esta parte se tuvo bastante libertad por lo cual a continuación se detallará diversas decisiones tomadas en su desarrollo.

1.2.1. Selección

Para obtener una población final de N (pop_size) individuos en la nueva generación se decidió seleccionar $\sqrt{N}$ padres y cruzar todos con todos. Por otro lado también se decidió no cruzar a un individuo consigo mismo para aumentar la variación en los hijos resultantes, por esto y por el hecho de que no todo N debía ser una raíz perfecta se decidió a la siguiente fórmula: `n_parents = math.ceil(math.sqrt(self.__pop_size) + 1)`

Una vez decidido el número de padres, debíamos elegir quiénes serían estos, para ello se empleó una estrategia de Tournament, obteniendo una muestra de la población total, y guardando el de mejor rendimiento. El tamaño de la muestra era inicialmente de 0.1 veces la población, pero luego se modificó a `1/n_parents` veces la población para tener un número más constante de individuos a analizar. Esto se repite `n_parents` veces para obtener los padres necesarios y se continúa a la etapa de crossover.

1.2.2. Crossover

Para la realización del crossover se toma el valor `mut_rate` y se multiplica por el número de genes en un individuo para así obtener el número de genes a modificar NG. Luego se eligen NG índices al azar y se reemplazan los genes de un individuo 1 por los de un individuo 2. Cabe recordar que todo individuo se cruza con los demás, es decir el cruce se realiza en ambos sentidos ($A \times B \rightarrow ab$ y $B \times A \rightarrow ba$).

Debido a que en la selección no se evita elegir al mismo individuo dos veces, es posible que un individuo haga crossover consigo mismo manteniéndose igual, pero esto se permite debido a que para ser elegido dos veces debe ser un buen individuo, además de que de todas maneras será alterado en la etapa de mutación.

1.2.3. Mutación

Finalmente en la etapa de mutación nuevamente se obtiene el valor NG de genes a modificar, y todo individuo es sometido a un reemplazo de NG genes por valores generados aleatoriamente por la fábrica de genes suministrada.

2. Word Problem

El problema consiste en adivinar una palabra con la única información de cuántos aciertos se llevan.

2.1. Modelación del Problema

Un individuo se define como una lista de letras en minúscula y su fitness está determinada por cuántos aciertos tiene respecto a la palabra deseada. Si el número de aciertos es igual al largo de la palabra, el algoritmo termina.

2.2. Resultados

A continuación se muestran resultados obtenidos para las palabras "donas" y "estornudo" con los parámetros `mutation_rate = 0.3` y `population_size = 200`:

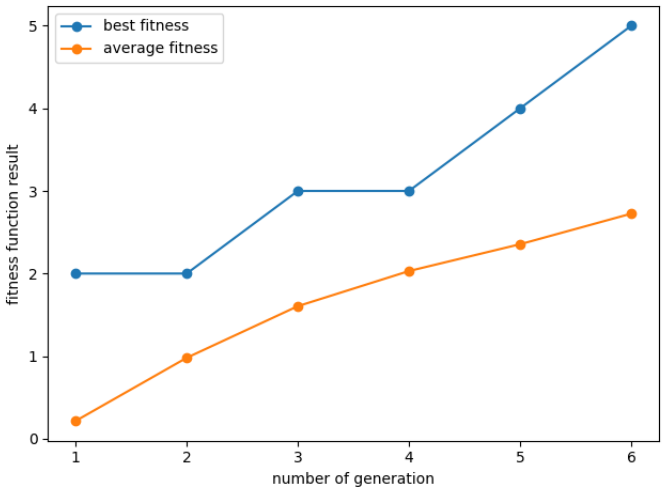

Figura 1: Rendimiento para "donas"

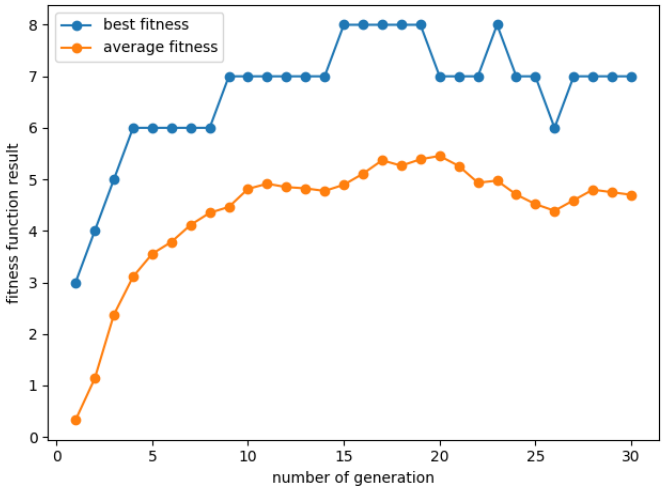

Figura 2: Rendimiento para "estornudo"

2.3. Discusión

La palabra "donas" al ser corta no tiene problema en ser adivinada por el algoritmo, pero debido al crecimiento de posibles individuos con palabras largas (cada caracter adicional aumenta en 23 veces las opciones) el algoritmo no es capaz de adivinar la palabra "estornudo" y luego de 30 generaciones nos retorna "estownmdo".

3. Binary Problem

El problema consiste en obtener la representación binaria de un número utilizando un algoritmo genético.

3.1. Modelación del Problema

Un individuo se define como una lista de enteros (0 o 1) y su fitness está determinada por la distancia entre el valor que representa y el valor deseado. Por comodidad de cálculos los individuos poseen una representación invertida del binario, es decir: 100 se representa como [0,0,1]. Los individuos son invertidos antes de ser presentados al usuario para mayor legibilidad. El algoritmo termina cuando la diferencia de valores es 0.

3.2. Resultados

A continuación se muestran resultados obtenidos para los números 528 con individuos de largo 10 y 2068 con individuos de largo 20 (seed=0):

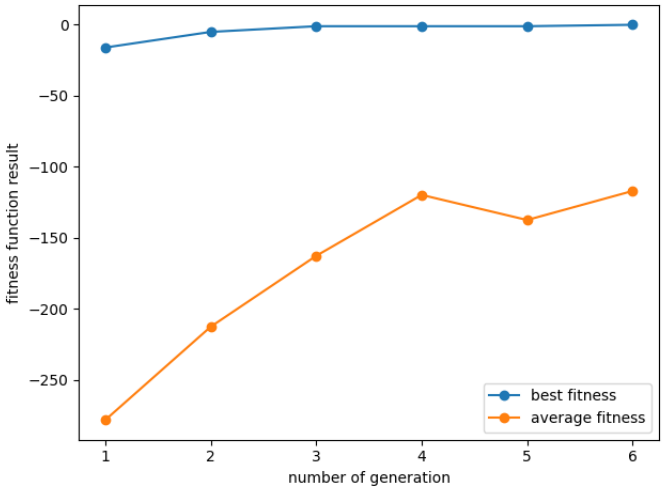

Figura 3: Rendimiento para 528, mr=0.3, ps=100

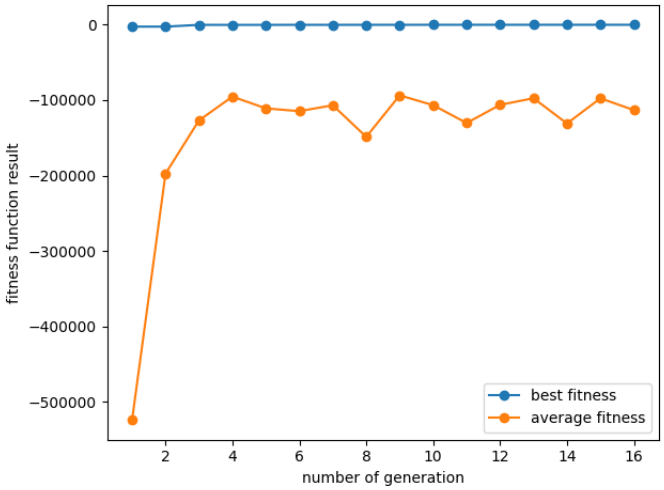

Figura 4: Rendimiento para 2068, mr=0.2, ps=150

3.3. Discusión

Vemos que con un re-ajuste de los parámetros `mutation_rate y population_size` el algoritmo es capaz de resolver un problema con individuos de incluso el doble de largo, esto gracias a que a diferencia de las palabras solo se tienen 2 posibilidades por gen, es decir, cada gen que añademos a un individuo solo duplica las opciones.

4. N-Queen Problem

El problema seleccionado consiste en encontrar una distribución de reinas sobre un tablero de ajedrez de modo que ninguna pueda atacar a otra. Las condiciones son que para un tablero de dimensiones $N \times N$ se deben ubicar N reinas para cualquier $N > 3$.

4.1. Modelación del Problema

La modelación del individuo consiste en ingresar las coordenadas de cada reina una después de la otra en un mismo arreglo de largo $2N$ como se muestra a continuación:

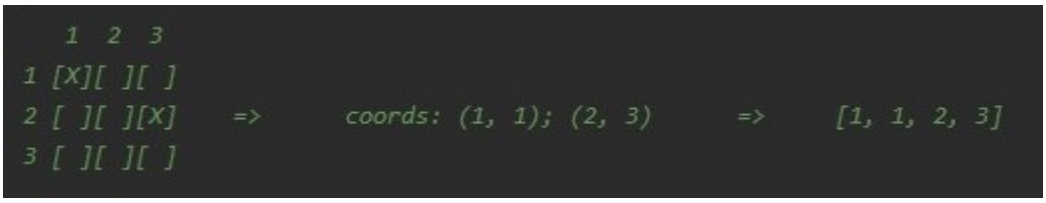

Figura 5: Modelo de un individuo

Por otro lado la función de fitness se describe como: `fitness=-n_of_attacks/2` donde el número de ataques se determina contando, por cada reina, las reinas que ataca (contando así cada ataque dos veces), este número se divide en 2 por legibilidad de resultados y se multiplica por -1 para seguir el objetivo de maximización de fitness.

La función de creación de genes simplemente retorna un número entre 1 y N, y la creación de individuo llama a la creación de genes $2N$ veces para completar un arreglo.

El algoritmo termina cuando la fitness alcanza el valor 0, es decir, no existen reinas atacándose entre sí.

4.2. Resultados

El algoritmo incluye un método `graph()` para mostrar el desarrollo del mejor rendimiento y el rendimiento promedio a través de las generaciones, y por otro lado el programa en `main.py` incluye una visualización del mejor individuo al finalizar el experimento.

A continuación se muestran resultados obtenidos para $N = \{4, 5, 6\}$ con los parámetros `mutation_rate = 0.2 y population_size = 100`:

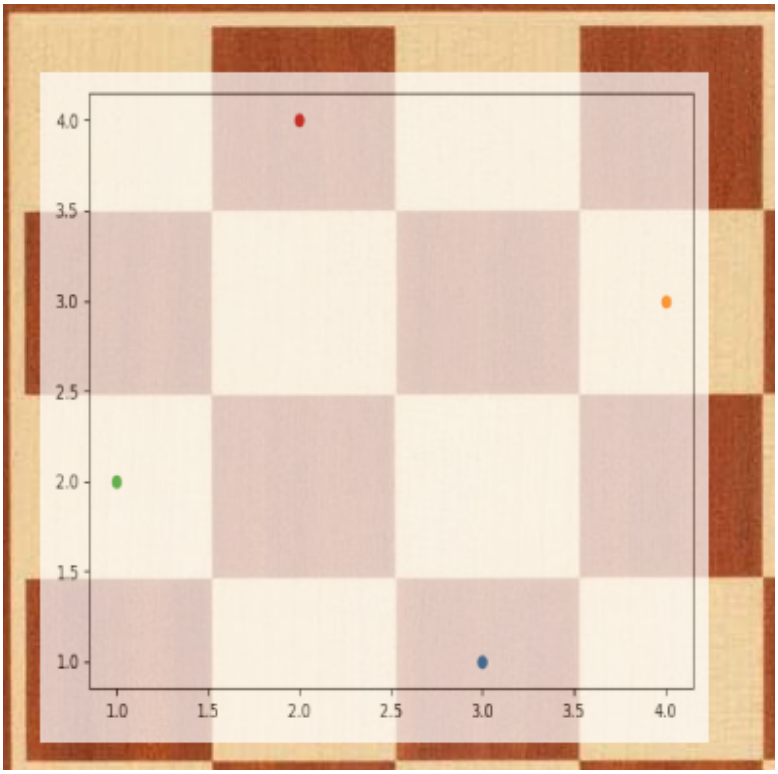

Figura 6: Resultado correcto para N=4, seed=43

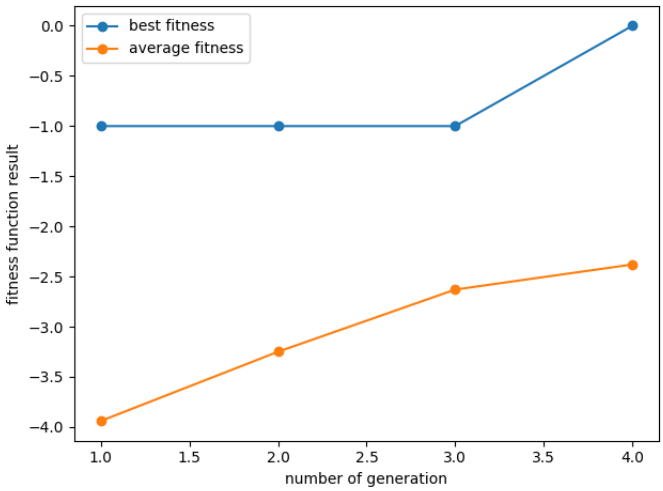

Figura 7: Desempeño para N=4, seed=43

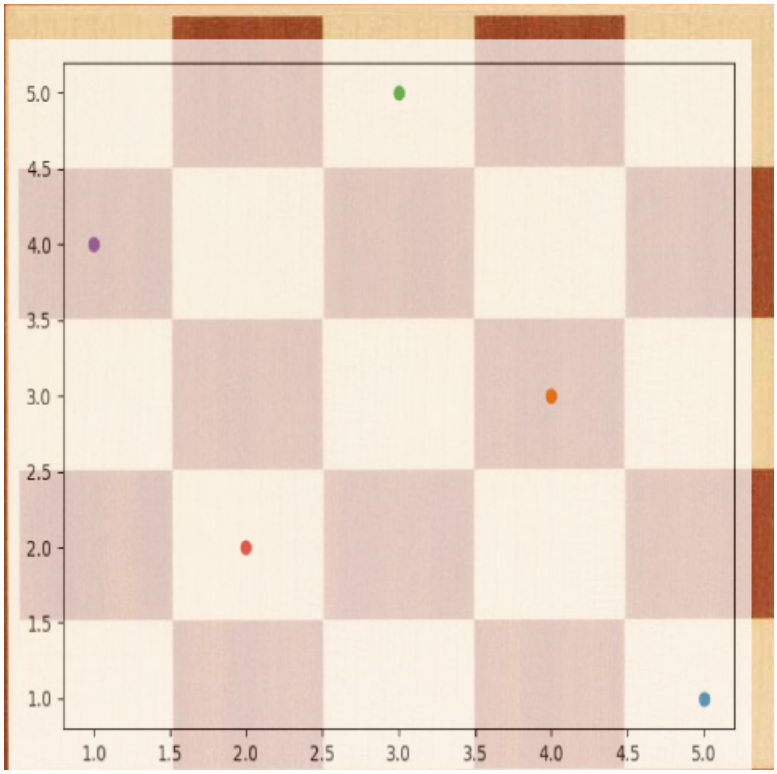

Figura 8: Resultado correcto para N=5, seed=43

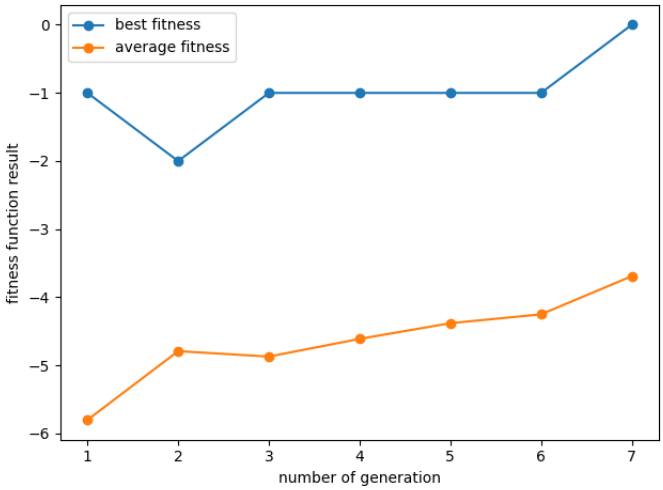

Figura 9: Desempeño para N=5, seed=43

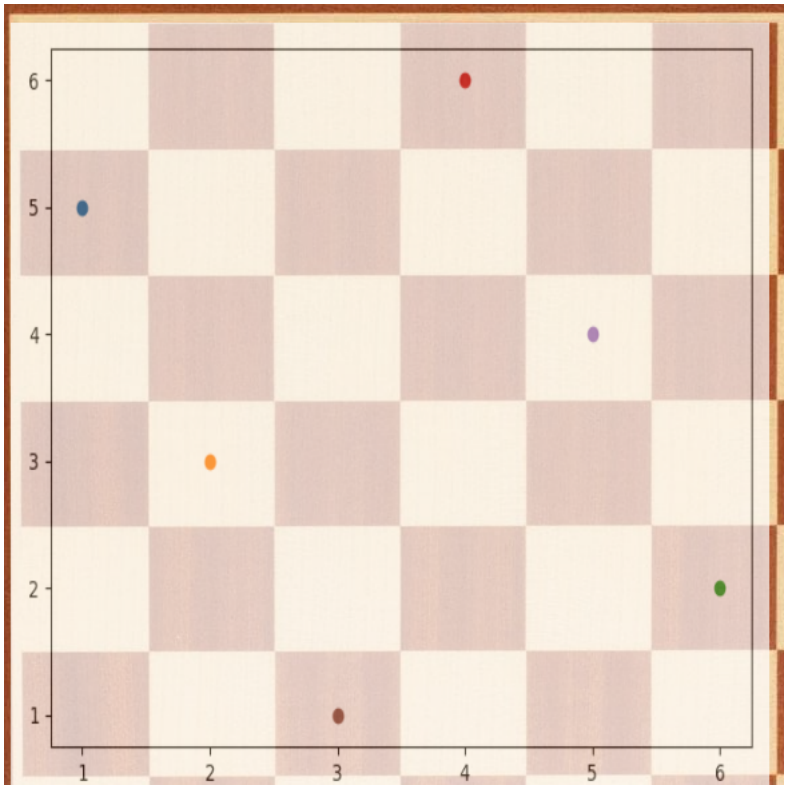

Figura 10: Resultado correcto para N=6, seed=528

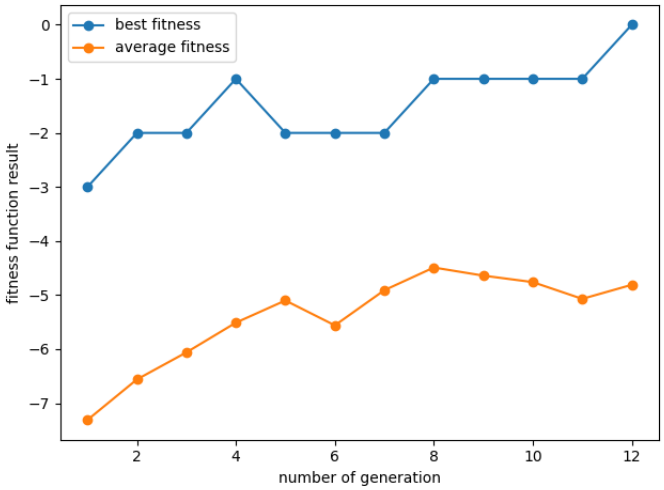

Figura 11: Desempeño para N=6, seed=528

Luego de estos experimentos se decidió aumentar drásticamente el valor N para obtener resultados más descriptivos y menos centrados en la suerte de la seed inicial, a continuación se muestra el resultado para $N = 30$:

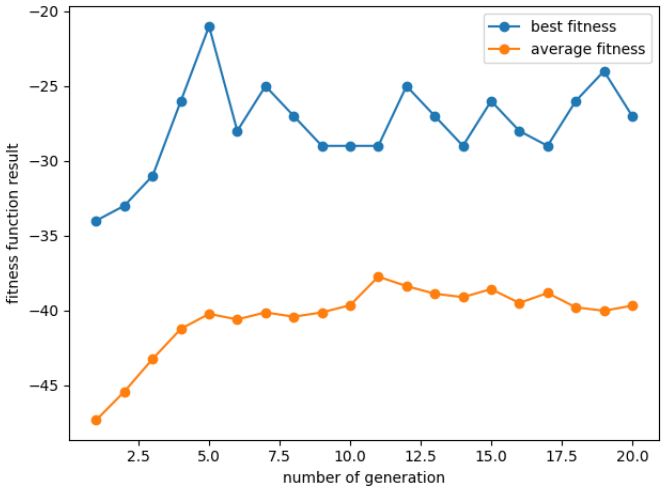

Figura 12: Desempeño para N=30, seed=10

Visto que el resultado no cumplía con el objetivo ni parecía tender a este, se decidió modificar los parámetros a `mutation_rate = 0.05` y `population_size = 500`, obteniendo el siguiente resultado:

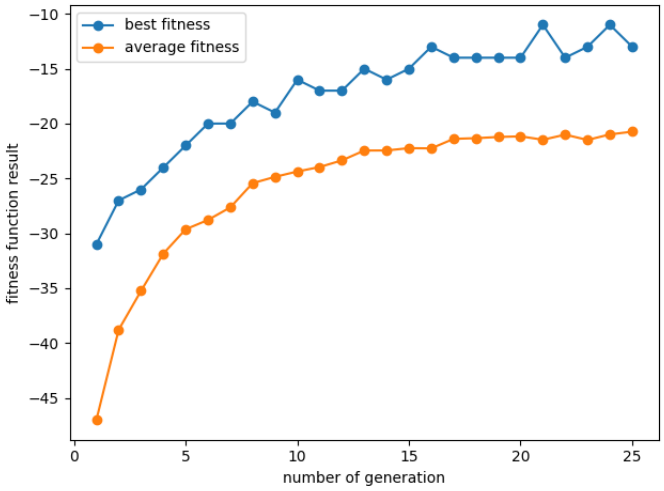

Figura 13: Desempeño para N=30, seed=10, mr=0.05, ps=500

Con la mejora se decidió modificar nuevamente los valores a `mutation_rate = 0.02` y `population_size = 700`, produciendo el siguiente gráfico:

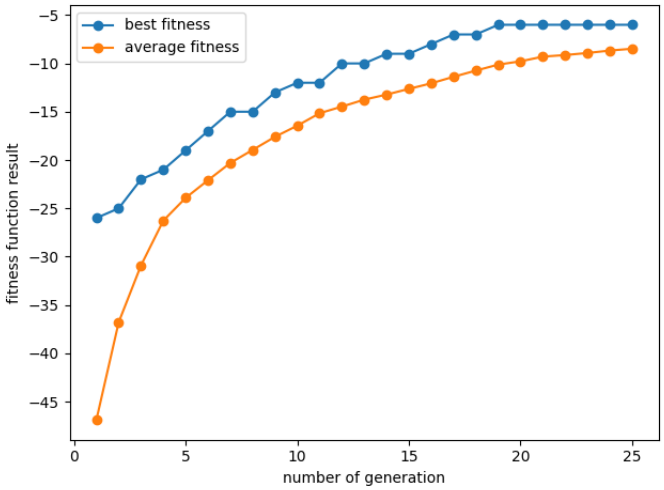

Figura 14: Desempeño para N=30, seed=10, mr=0.02, ps=700

Finalmente se aumentó el número de generaciones para tener una mejor idea de la convergencia del resultado, obteniendo:

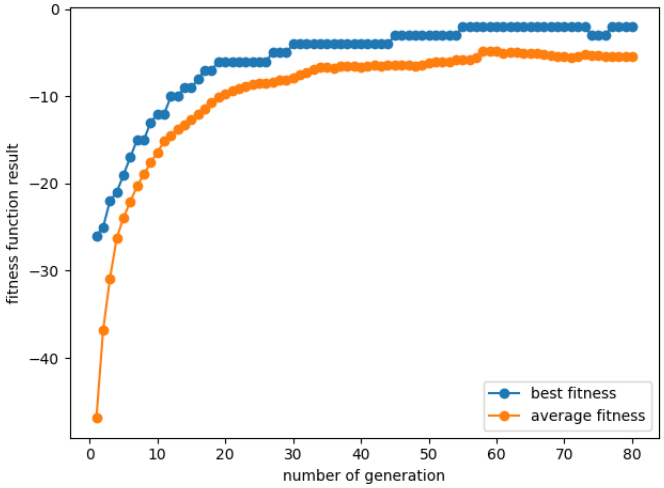

Figura 15: Desempeño para N=30, seed=10, mr=0.02, ps=700, gen=80

4.3. Discusión

Observamos que para valores pequeños de N el algoritmo tiene un buen desempeño obteniendo la respuesta correcta sin necesidad de modificar parámetros, pero a medida que aumentamos el valor de N se vuelve necesario tener más precaución sobre los parámetros globales utilizados.

Se observa claramente que aumentar la población y disminuir el ratio de mutación los resultados son más consistentes y la curva tiende a solo subir o mantenerse en cada generación como se ve en la figura 10, en comparación a por ejemplo la figura 8. Esto probablemente se debe a que teniendo un menor ritmo de mutación, las mejoras obtenidas en previas generaciones tienden a mantenerse y no sobre-escribirse. Por otro lado, es importante recalcar que no se puede seguir disminuyendo infinitamente el ritmo de mutación, pues si este es muy bajo, los individuos dejarán de mutar y tenderán a quedarse atrapados en un máximo local.

Para automatizar la obtención de un individuo perfecto se tendría que ejecutar el algoritmo repetidas veces con distintos valores en los parámetros mencionados. Ingresar valores aleatorios eventualmente nos daría una respuesta correcta, pero para una mayor eficiencia también se podría emplear otro algoritmo genético el cual tiene como individuo un arreglo con los valores a modificar y como fitness function la ejecucción de nuestro algoritmo (sería importante mantener una misma seed para mantener la eficiencia de individuos respecto a la ejecución de generaciones anteriores).

5. Posibles mejoras

La zona con más posibles mejoras es nuevamente al momento de seleccionar cómo reproducir una nueva generación. Se podría intentar con otro método de selección para así elegir siempre los mejores individuos; o se podría disminuir el número de padres seleccionados, y con ellos el número de hijos, para luego aumentarlo en la etapa de mutación, guardando varias copias mutadas de cada hijo. Obtener una generación mejor que la anterior es vital para un mejor rendimiento de un algoritmo genético.

Por otro lado, una mejora particular y más abarcable que realizaría, sería dejar un porcentaje modificable al momento de realizar el torneo, pues al mantenerlo asociado al tamaño de la población (`1/n_parents`) tenemos un grado de control menos para manejar el rendimiento del algoritmo, pues al modificar la población se alteran ambos valores.