Cómputo Evolutivo

Tarea 04 : Algoritmo Genético, problema de las N-Reinas (N-Queens)

Celeste Lorenzo Guerrero : 316162027
Antonio Sebastián Dromundo Escobedo : 419004327
Diego Dozal Magnani : 316032708

April 25, 2023

Descripción de problema

Dado un tablero de ajedrez de $n * n$ encontrar la configuración tal que al colocar n reinas, dos reinas no se ataquen la una a la otra. Es decir, que ninguna de las reinas compartan fila, columna o diagonal.

Representación de soluciones

`Implementación : src/queen_rep.py -> class QueenSolution`

Para representar un ejemplar del problema vamos a considerar una permutación de valores enteros S=[x_1,. . .,x_n] de tamaño n descrita en [1], cuyos valores van de 0 a $n-1$ (en caso de que los índices de la permutación se cuenten a partir de 0). La permutación S tiene las siguientes características :

Para cada x_i en S, el índice i representa la fila del tablero en la que la i-ésima reina está colocada.

Para cada x_i en S, el valor x_i representa la columna del tablero en la que la i-ésima reina está colocada.

Espacio de búsqueda

Como anteriormente ya se comentó, el espacio de busqueda serían todas las permutación de n y el tamaño del espacio de búsqueda es n.
Consideremaos importante resaltar que, como ocurre con el *problema de las 8 reinas* [1], las soluciones pueden compartir un patrón por lo que el número de soluciones puede ser reducido aplicando rotaciones y reflexiones del tablero. Para este trabajo no se aplican ni rotaciones ni reflexiones, por lo que el tamaño del espacio de búsqueda continúa siendo $n!$.

Función Evaluación

`Implementación : src/queen_rep.py -> class QueenSolution.evaluate`

Definimos un **conflicto** cuando una reina está en la misma fila, columna o diagonal que otra reina. De este modo, nos van a interesar las configuraciones con menores conflictos.
Notemos que, usando nuestro esquema de permutaciones en un arreglo ya no hay necesidad de considerar los conflictos de filas y columnas para las reinas, por lo que nos enfocaremos en reducir los conflictos en diagonales.
Considerando el vector $\mathbf{S} = [\ x_1, \ldots, x_i, \ldots, x_j, \ldots]$, las reinas x_i y x_j compartirán la diagonal si :

$$i - x_j == j - xi$$

ó

$$i + x_i == j + x_j$$

que se reduce a[1]:

$$||q_i - q_j|| = ||i - j||$$

Ahora bien, en un tablero de $n * n$ donde acomodamos a n reinas cada una en la misma diagonal, obtenemos la configuración con la **máxima cantidad de conflictos** (considerando nuestra representación). Para esta combinación la cantidad de conflictos son $n * (n - 1)$, por que cada reina genera un conflicto con cada **otra** reina (excluyéndose a sí misma).
Sean M la *máxima cantidad de conflictos* para un ejemplar del problema de las n reinas, S una solución (una configuración del tablero para el ejemplar) y $C(S) : S \to \mathbb{N}$ la cantidad de conflictos de S, la función de evaluación para S está dada por

$$f(S) = M - C(S) = n * (n - 1) - C(S)$$

Mientras más conflictos tenga S menor será $f(S)$ y mientras menos conflictos mayor será la función objetivo.Por lo que la solución óptima tendrá valor M. De este modo buscamos **maximizar** $f(S)$.

`Nota :` Para evitar repeticiones al momento de contar los conflictos simplemente dividimos entre dos el número total de conflictos (tanto para la máxima cantidad de conflictos) como para el conteo de conflictos correspondiente a una solución.

Función objetivo

Dado el valor de *fitness* de un individuo, nuestra función objetivo es minimizar la cantidad de conflictos (valor objetivo) de ese individuo a través de maximizar la función de evaluación. Para cada ejemplar de problema de las $n-$reinas, la configuración óptima es aquella en donde ninguna reina se ataca, es decir donde su valor objetivo es igual a $n * (n - 1)$ como ya se explicó anteriormente en la función de evaluación.

Algoritmo Genético

```
Implementación : src/ga.py -> class GeneticAlg.selection_rl
```

`Nota :` Para los métodos de selcción (tanto ruleta como elitismmo) utilizamos una variable `sel_proportion`, la cual es un floatante entre (0,1) que determina cuál será el porcentaje de la población que será seleccionado y cuantos seleccionados por la estrategia de elitismo. En general el porcentaje de proporción de la población para ser seleccionado por ruleta es mayor (rondando entre 0.8 y 0.9) al de selección por elitismo, el cual es 1-`sel_proportion`.

Selección por ruleta

`Implementación : src/ga.py -> class GeneticAlg.selection_rl`

También conocida como *Fitness Proportionate Selection* (FPS), la probabilidad de seleccionar un individuo es directamente proporcional a su valor objetivo. Esto es comparativo a usar una ruleta de casino y asignarle a cada individuo una porción de la rueda proporcional a su *fitness*.[2]
Para determinar la probabilidad del individuo i con *fitness* f_i hacemos

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_i}$$

Estrategia de Elisitmo

La estrategia de elitismo es considerar a los mejores de cada generación, para hacer esto ordenamos a los individuos respecto a su valor objetivo de forma descendente y utilizando la proporción de selección 1-`sel_proportion` (mencionada al inicio de esta sección), consideramos a los mejores individuos de cada generación. De esta forma garantizamos que el mejor de cada iteración siempre es seleccionado.

Operador de cruza

`Implementación : src/ga.py -> class GeneticAlg.crossover`

El operador de cruza implementado es el que fue explicado en clase para permutaciones. Consideramos los dos padres S_1=[x_1,...,x_n] y S_2=[y_1,...,y_n]. Vamos a tomar dos puntos de corte, para la implementación consideramos *Punto de corte inicial* $p_i = \lfloor \frac{n}{4} \rfloor$ y *Punto de corte final* $p_f = \lfloor \frac{3n}{4} \rfloor$. Una vez que tenemos los puntos de corte, Sea S_{12} el hijo entre S_1 y S_2, la generación del cromosoma es la siguiente :

1. Para agregar la información del padre S_2 se hace S_{12}[p_i,...,p_f] = S_2[p_i,...,p_f].

2. Para agregar la información del padre S_1, comenzando en el índice $k = p_f + 1$, hacemos $S_{12}[k] = S_1$[k] e iteramos k desde $k = p_f + 1$ hasta $k = p_i - 1$, es decir, vamos a realizar una operación de módulo para que el cromosoma del padre S_1 pueda ser iterado de forma circular.
   Adicionalmente buscamos generar cromosomas válidos, es decir, que sigan siendo permutaciones de n, por lo que cada vez que un elemento x_k del padre S_1 vaya a ser *heredado* a S_{12} tenemos que verificar si x_k no se encuentra ya dentro de la información que S_{12} heredó de S_2 (paso 1). Si x_k ya está en el cromosoma de S_{12} entonces ignoramos ese valor y continuamos iterando hasta que encontremos un elemento que podamos agregar.

El proceso es análogo para S_{21}, cuya información *intermedia* es la de S_1.

Operador de mutación

`Implementación : src/ga.py -> class GeneticAlg.mutate_individual`

El operador de mutación es el visto en clase. Para cada individuo de la nueva generación hay una probabilidad de ser seleccionado para ser mutado (en la implementación el valor está entre 0.1 y 0.2), si es seleccionado entonces se toman dos índices aleatorios del cromosoma y se intercambian esos valores.

Experimentación

Tabla de parámetros

No.Reinas	Tamaño Poblacion	P.Cruza	P.Mutación	Tiempo límite (s)
$n \in \{8, .., 20\}$	100	0.8	0.1	1
$n \in \{8, .., 20\}$	100	0.8	0.1	5

Resultados

La siguiente tabla fue generada con los resultados onbtenidos al realizar cada ejecución con un máximo de tiempo de **2 segundos** y 10 repeticiones por cada uno. **La única condición de paro fue el límite de tiempo.** Adicionalmente

No.Reinas	AVG. Tiempo (s)	Best.Known	AVG. Fitness
8	0.0040	28	26.9
9	0.0050	36	34.8
10	0.0062	45	43.3
11	0.0090	55	52.8
12	0.0086	66	63.5
13	0.0089	78	75.4
14	0.0119	91	87.7
15	0.0127	105	100.9
16	0.0129	120	115.7
17	0.0139	136	131.4
18	0.0160	153	147.5
20	0.0217	190	183.8

Table 1: Ejecuciones con condición de paro **límite de tiempo.**

es importante mencionar que según los datos arrojados, parec que el promedio de iteraciones para cada una de las ejecuciones (sin importat el número de reinas) se mantuvo en **0.0**.
Como podemos observar, todos los ejemplares están muy cerca del valor objetivo óptimo. Sin embargo comforme el *No.Reinas* incrementa más allá de 15, la diferencia con el óptimo es de 5 o incluso 7 puntos. Por otro lado, es importante resaltar que incluso para ejemplares mayores a 15, el algoritmo no tomo ni un segundo en llegar a resultados relativamente cercanos al óptimo.

La siguiente tabla (Table 2) fue considerando como **única condición de paro que el algoritmo encuentre el óptimo**. Es importante comentar que en todas las ejecuciones mostradas, el óptimo fue alcanzado para cada ejemplar. Como podemos observar, tanto el promedio de tiempo como de generaciones varía en gran medida para cada ejemplar. Lo que es evidente es que paraa los ejemplares mayores o iguales a 17, el tiempo promedio es bastante alto y las generaciones también.
Finalmente, la siguiente tabla (Table 3)es considerando tanto un **límite de tiempo de 5 segundos** o que el **algoritmo encuentre el óptimo global.**
Una primera observación es notar que efectivmente, el tiempo promedio crece respecto a la entrada, al número de reinas.
En segundo lugar es interesante notar que el número de iteraciones independientemente del ejemplar se parecen entre ellas, el promedio de generaciones no suele pasar de 89.0666 (que es el promedio de las generaciones de 12 ejemplares).

No.Reinas	AVG. Tiempo (s)	AVG. Generaciones
8	0.0301	2.7
9	0.0822	6.4
10	2.24	172.9
11	7.4482	360.8
12	3.22745	143.9
13	6.4116	240.3
14	8.6823	306.2
15	10.0063	275.1
16	12.2801	325.5
17	31.087	608.2
18	88.5792	1590.0
20	172.5752	2572.0

Table 2: Ejecuciones con condición de paro **encontrar el óptimo.**

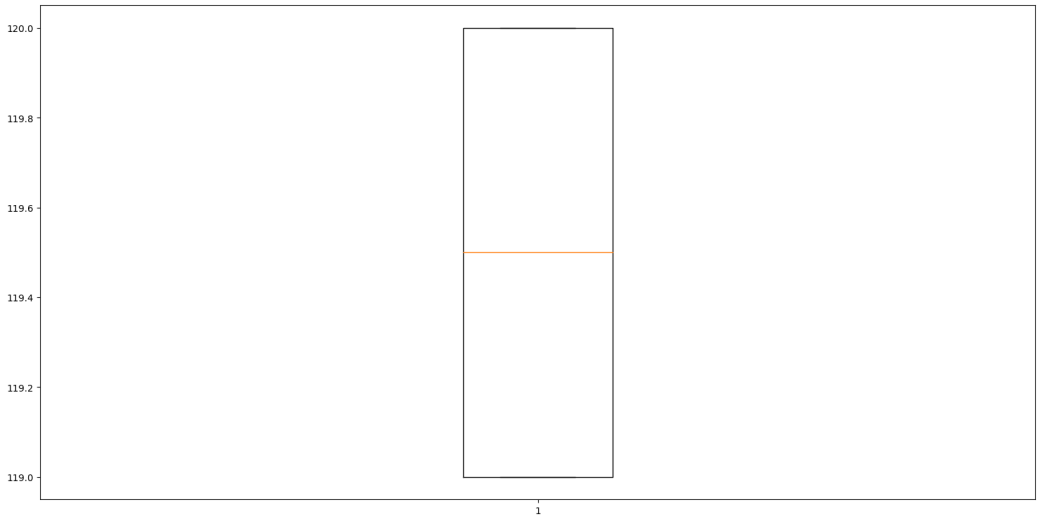
No.Reinas	AVG. Tiempo (s)	AVG. Generaciones	Best.Known	AVG. Fitness	% Succes
8	0.0828	6.9	28	28	100%
9	0.4825	43.7	36	36	100%
10	1.4444	98.5	45	44.8	80%
11	2.3620	139.0	55	54.8	80%
12	2.2735	104.5	66.0	65.7	70%
13	2.4653	99.9	78.0	77.8	80%
14	2.5155	89.3	91.0	90.7	70%
15	3.8174	105.5	105.0	104.6	60%
16	3.6051	102.9	120.0	119.5	50%
17	4.6481	103.0	136.0	135.1	80%
18	4.2742	81.9	153.0	152.3	30%
19	4.5459	77.9	171.0	169.8	20%
20	4.9891	77.1	190.0	188.7	10%

Table 3: Ejecuciones con condiciones de paro **encontrar el óptimo y límite de tiempo.**

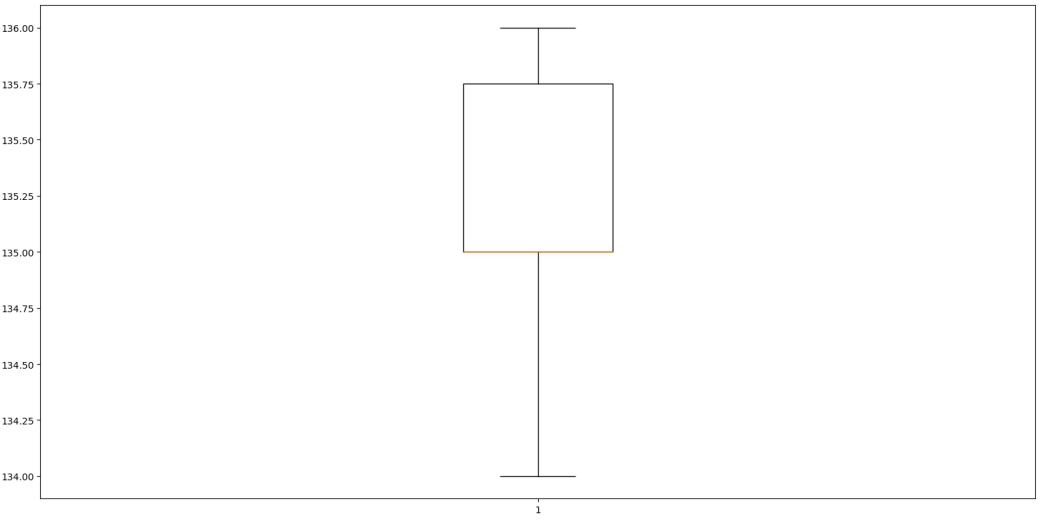
Como podemos observar, las ejecuciones en donde el número de reinas es menor o igual a 15, el porcentaje de éxito es mayor al 50 % , mientras que a partir de ejemplares con 16 o más reinas, el porcentaje de éxito disminuye considerablemente hasta el caso extremo del ejemplar de 20 reinas con 10% de éxito. Sin embargo, la diferencia entre el **mejor valor conocido** y el **promedio** en los ejemplares mayores a 16 es muy pequeña , incluso está muy cercana a el mejor valor conocido. Por lo que a pesar de que no alcance el óptimo global, sí alcanza valores muy cercanos a este.

Gráficas

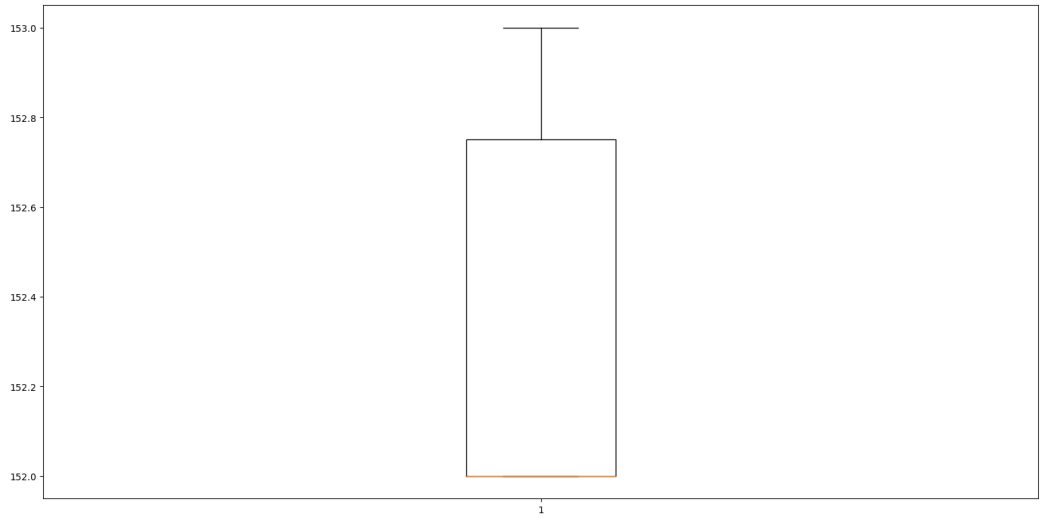
Se decidió utilizar gráficas **boxplot** para visualizar resultados.



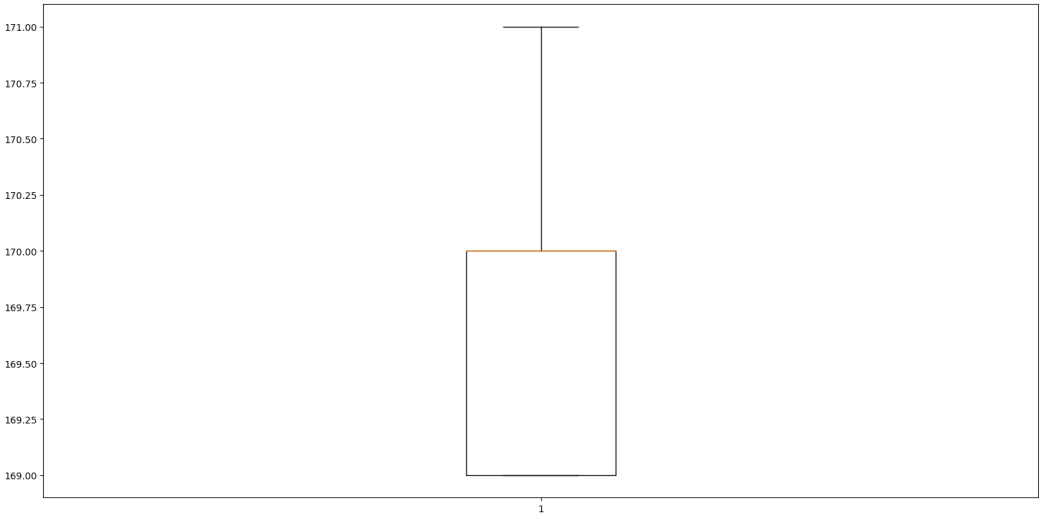
Notemos que la gráfica para el ejemplar con 16 reinas tiene una media de 119.4 , por lo que está muy cerca del óptimo, como habíamos analizado en las tablas anteriores, incluso el peor valor (119) sólo se aleja una unidad.



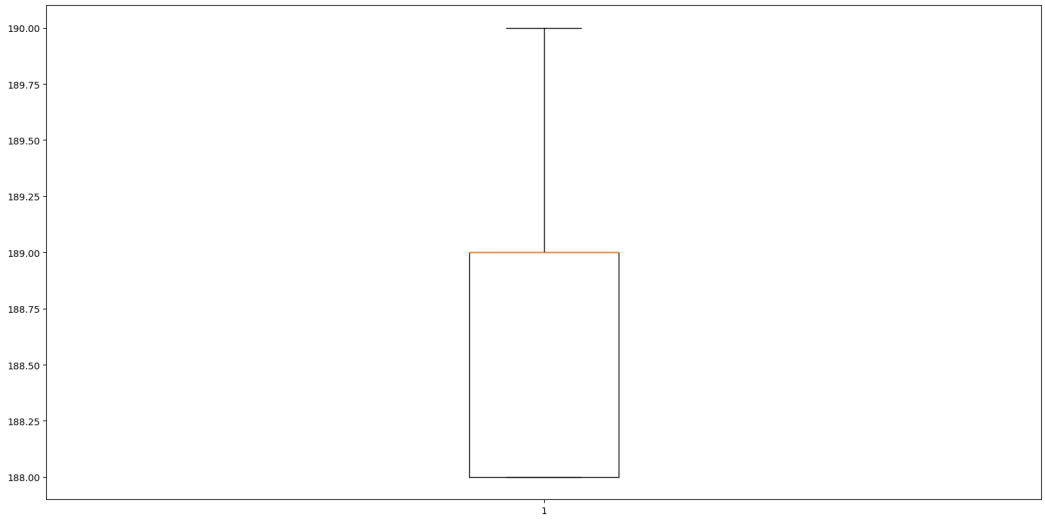
La gráfica para el ejemplar con 17 reinas muestra una mejora en los resultados, ya que la media coincide con los valores medios, los cuales se encuentran una fracción de la unidad menos cerca del óptimo. En este caso, el rendimiento del algoritmo es muy bueno.



Para el ejemplar con 18 reinas se muestra una disminución mínimo en encontrar el óptimo, ya que el mínimo valor coincide con la mediana, la cual se encuentra a una unidad de diferencia.



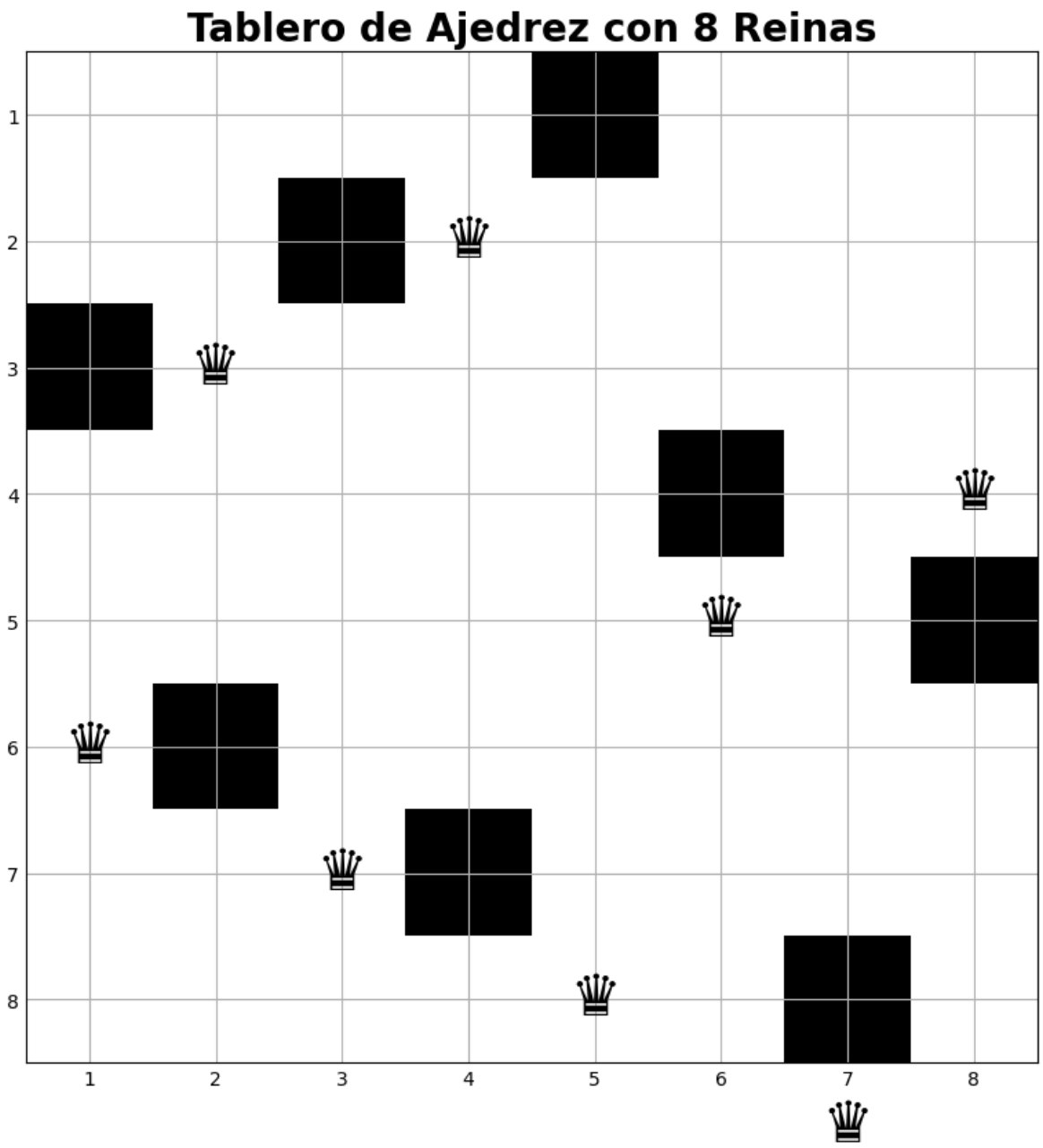
En el caso del ejemplar de las 19 reinas, la media se encuentra igualmente a una unidad de igual manera respecto al óptimo. El desempeño continúa siendo bueno.

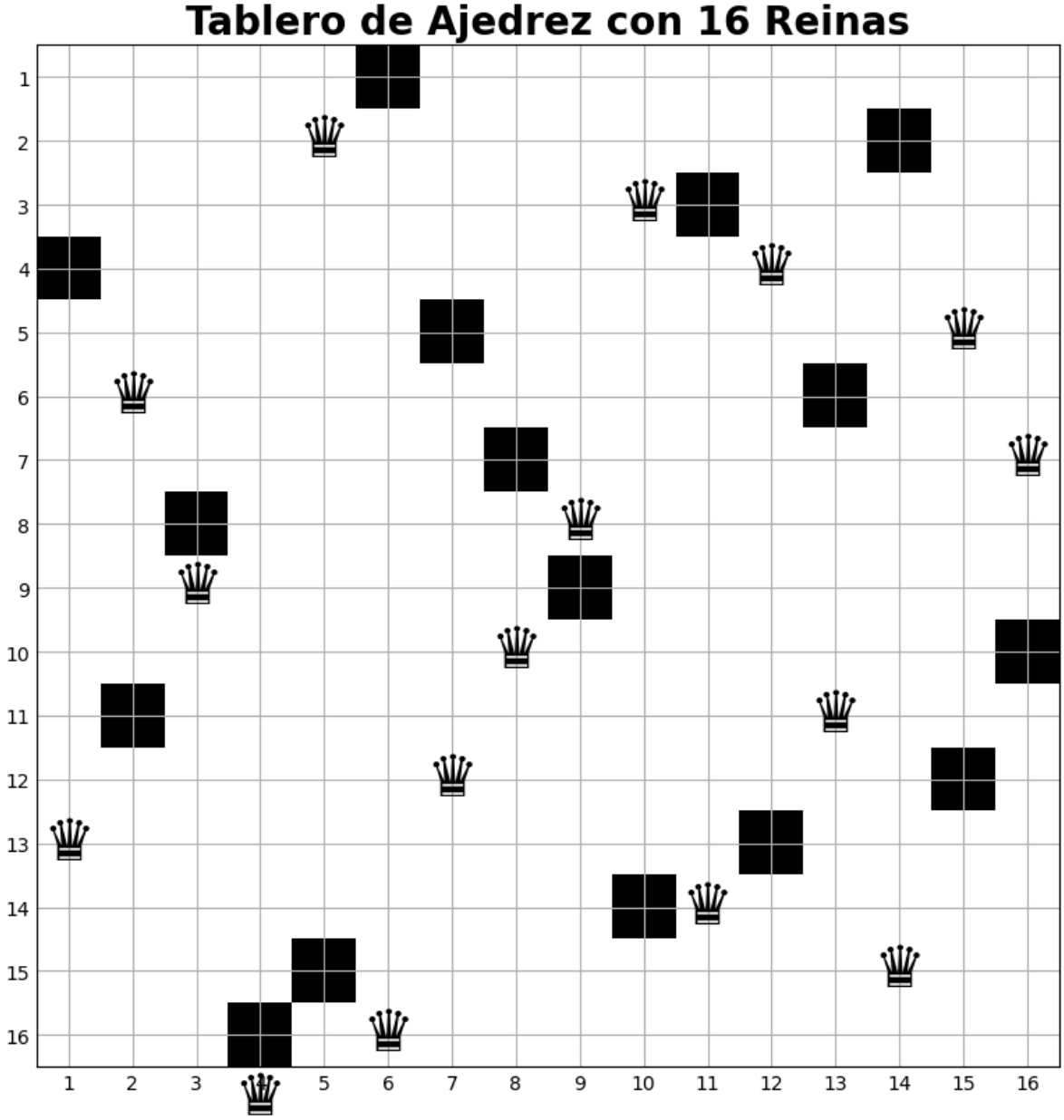


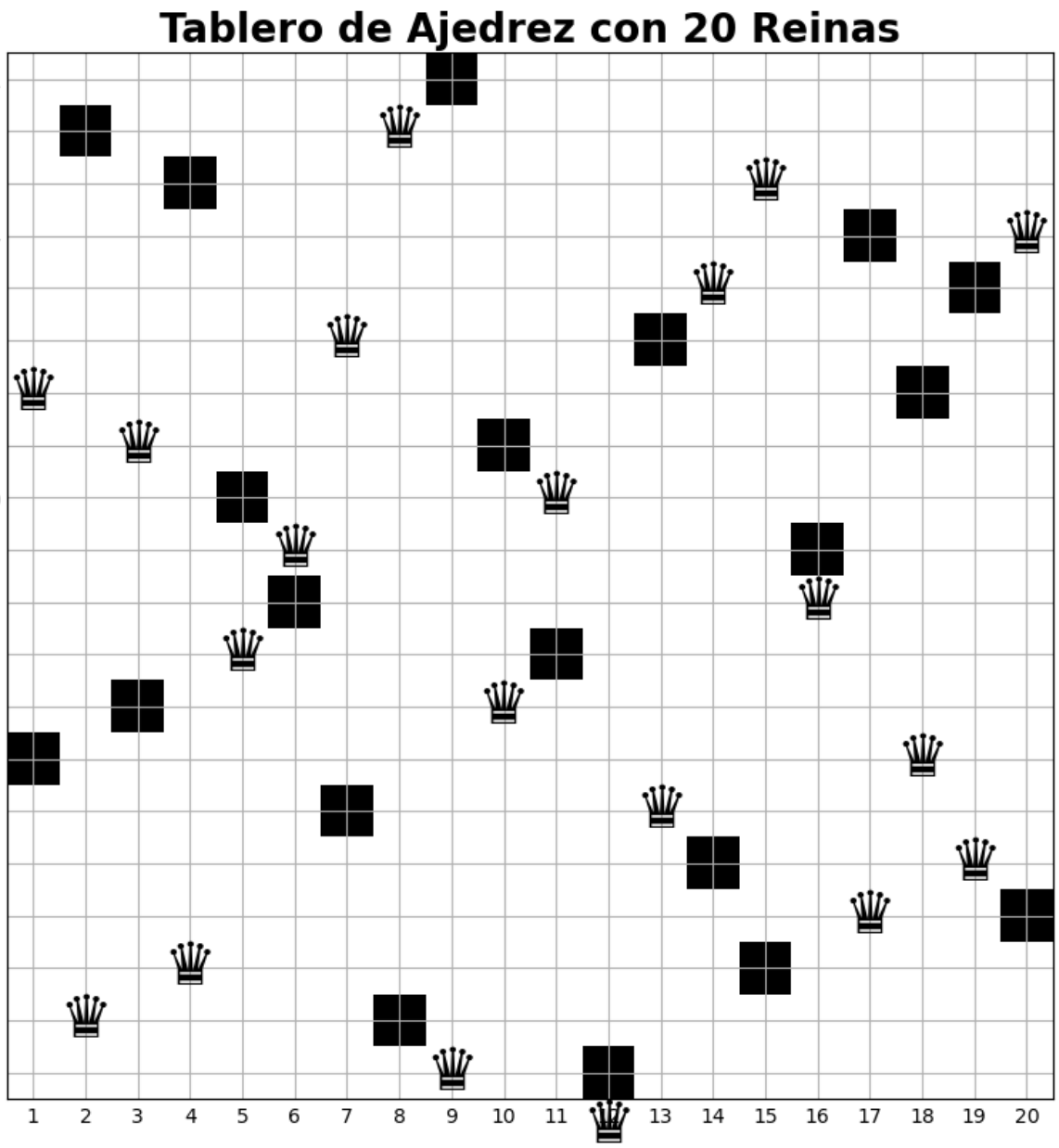
Finalmente el ejemplar con 20 reinas, de igual forma podemos notar que la media está a una unidad del óptimo, el óptimo se alcanza en alguna de las ejecuciones pero en general el promedio es estar a una unidad de diferencias.

Con las gráficas boxplot podemos afirmar que el desempeño del algoritmo con los ejemplares con mayor cantidad de reinas es bueno, incluso considerando los 5 segundos. Ya que en la mayoría de ellos el valor promedio se encuentra a una unidad del valor óptimo.

Ejemplos Gráficos




Tablero de Ajedrez con 16 Reinas
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
1
2
3
5
6
7
8
9
10
11
12
13
14
15
16


Tablero de Ajedrez con 20 Reinas
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Conclusiones

Consideramos que el contraste entre los dos criterios de paro (tiempo máximo y valor óptimo) era importante para analizar el rendimiento del algoritmo. Como ya antes se mencionó, el algoritmo logra encontrar buenos resultados cercanos al óptimo en un tiempo menor a 1 segundo, mientras que si buscamos encontrar el óptimo con tiempo libre, el algoritmo termina encontrándolo en todas sus ejecuciones más sin embargo el tiempo en que lo encuentra crece conforme crece la entrada.
Por otro lado , considerando ambos criterios de paro con un tiempo de **5 segundos** nos devolvió buenos resultados para cualquier ejemplar. Incluso para los ejemplares con mayor cantidad de reinas, dentro de los 5 segundos pudo encontrar en promedio configuraciones para el tablero con un valor objetivo que sólo distaban en una unidad respecto el valor óptimo.

Considerando los resultados obtenidos y reflejados en las tablas de secciones anteriores junto con lo representado en las gráficas boxplot, añadiendo el análisis previo a las conclusiones, afirmamos que nuestro algoritmo tiene buen rendimiento al encontrar el valor óptimo con los parámetros que establecimos.

References

[1] Nieves Vázquez Vázquez Juan Carlos Pozas Bustos. Algoritmos genéticos. aplicación al juego de las n reinas.

[2] Eyal Wirsansky. *Hands-On Genetic Algorithms with Python.* Packt Publishing Ltd., Livery Place, 35 Livery Street, Birmingham, B3 2PB, UK, 1 edition, 2020.