

**CSC 180-01 Intelligent Systems**

**Mini-Project 4: Solving N-Queens Problem using Genetic Algorithms**

**Due at 2:00 pm, Friday, October 30, 2020**

Tran Ngoc Bao Huynh (Student ID: 219763298)

Ong Thao (Student ID: 219467431)

**Problem Statement:**

Our goal for this project is to apply the genetic algorithm utilizing the Distributed Evolutionary Algorithms in Python (DEAP) library to solve the “n-queens” problem. The “n-queens” problem is a fun chess puzzle invented in the mid-1800s that involves placing “n” number of queens on an “nxn” board in such a way that no queens are attacking each other (thus, n cannot be 2 or 3). Our project focuses on 8-queens on an 8x8 board. Additionally, we will also try with larger queen numbers and board sizes, analyzing how the setup, initialization, and representation of the board affects the fitness and result quality. The board representations that we will be analyzing are: position-indexed based and row-indexed-based.

**Methodology:**

We are generally evaluating the boards by counting the number of distinct queen-pair conflicts and adding a “penalty” to the fitness for each conflict. A conflict can be caused by duplicates (where two queens are in the same position) or happen when a pair of queens are placed in such a way that enables them to attack each other (same row, column, or diagonal). Our goal with adding a penalty to each conflict is to evaluate the boards based on “minimum fitness” where if a board has the least amount of conflicts, they have a lower fitness meaning the board’s placement of the queens is more accurate than another board with a higher fitness.

Following the template provided, we first started with implementing the position-index-based representation of the board. For this board representation, the potential conflicts that may arise includes: row, column, diagonal, and duplicates. For duplicates, we added a penalty of  $n(n-1)/2$  (n is the number of row/column)

while for row, column, and diagonal conflicts (that arises from two queens being able to attack each other) we added a penalty of 1 in our fitness evaluation function. Then we started experimenting with different numbers of individuals in each generation and the number of generations that we wanted to go through for each run in an attempt to see which combination produces the best results.

After implementing the position-index-based representation, we then moved onto implementing the row-index-based. Everything is really similar to the position-index-based except we no longer have to worry about the row and duplicate conflict because each queen now is placed in their own row and each row only has one queen which gets rid of these conflicts. This difference is implemented in the new `evaFitness` function and then we moved onto testing out different combinations of the number of individuals in each generation and the number of generations.

### **Experimental Results and Analysis:**

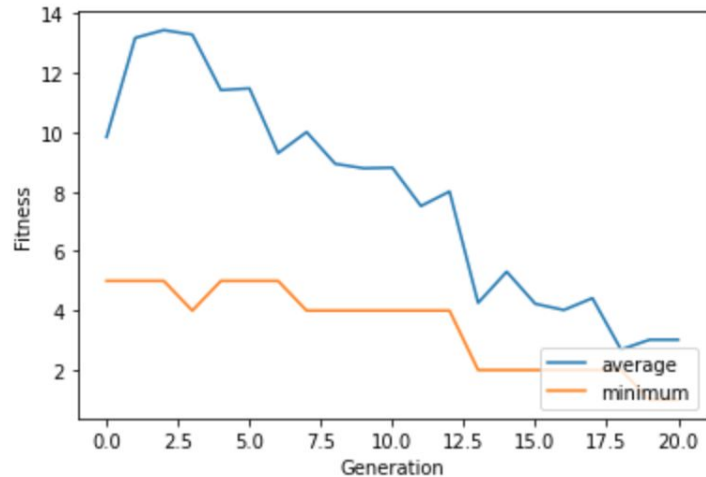
As expected, the row-index-based representation results did better than the position-index-based representation most likely because there are less conflicting factors meaning lower probability of a board having conflicts that will result in higher fitness values. This is because the total amount of possible conflicts caused by row-index-based is less than the total amount of possible conflicts in position-index-based. Besides that, row-index based is more efficient in time complexity compared to position-index-based. The algorithm that we use to detect conflict (2 or more queens attack each other) has time complexity:  $O(n)$ . However, the complexity when detect duplicate is  $O(n^2)$ . We can change this algorithm to have the complexity  $O(n)$  but this will take an extra  $O(n^2)$  space. Even if we use a hashmap to reduce both time and space complexity, the position-index-based will

still require more steps, memory, and time (to train later) compared to the row-index-based. Therefore, later on when applying the algorithm for larger queen and board size, we will use row-index-based.

As seen reflected in the min below, the position-index-based representation had min fitness values of 2-4 while row-index-based had min fitness values of 0-2 with most of them being 0s. Additionally, the averages for each generation in the row-index board were way lower than the averages for the position-index which all further shows that the row-index-based board did better.

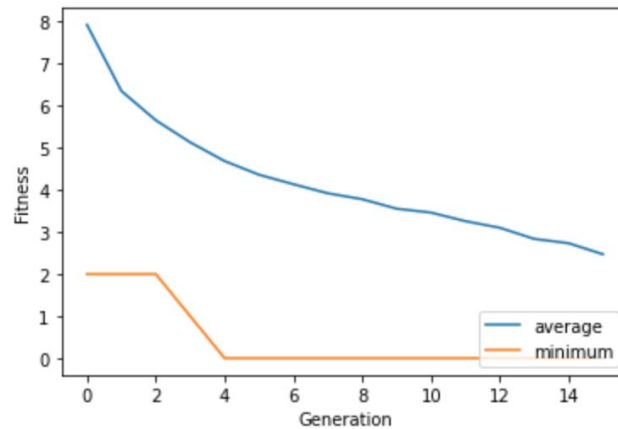
#### Results for “Position-Index-Based” Board Representation

gen	nevals	avg	min
0	100	9.84	5
1	48	13.18	5
2	55	13.44	5
3	63	13.29	4
4	58	11.42	5
5	49	11.48	5
6	69	9.3	5
7	62	10.01	4
8	63	8.94	4
9	63	8.79	4
10	63	8.81	4
11	57	7.52	4
12	69	8.01	4
13	63	4.26	2
14	61	5.31	2
15	67	4.23	2
16	58	4.02	2
17	61	4.42	2
18	56	2.69	2
19	71	3.02	1
20	69	3.02	1



## Results for “Row-Index-Based” Board Representation

gen	nevals	avg	min
0	2000	7.929	2
1	1204	6.3555	2
2	1257	5.659	2
3	1235	5.134	1
4	1172	4.6885	0
5	1195	4.3635	0
6	1217	4.137	0
7	1218	3.9225	0
8	1212	3.781	0
9	1206	3.5555	0
10	1227	3.463	0
11	1234	3.259	0
12	1237	3.102	0
13	1185	2.8375	0
14	1238	2.7345	0
15	1200	2.472	0



Tuning and changing the number of individuals of each generation and the number of generations will improve the correctness of the final output. These changes have a significant effect on the large number of queens and board size. Here are some improvement after we tuning:

```

pop = toolbox.population(n=10)
hof = tools.HallOfFame(maxsize=10)
pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=10,
                              stats=stats, halloffame=hof, verbose=True)

```

gen	nevals	avg	min
0	10	11.4	9
1	5	13.5	8
2	9	8.5	7
3	4	7.8	4
4	6	5.9	4
5	8	4.8	4
6	7	4.5	4
7	4	4.1	4
8	9	4.7	4
9	7	4.6	4
10	7	4	4

```
pop = toolbox.population(n=100)
hof = tools.HallOfFame(maxsize=10)

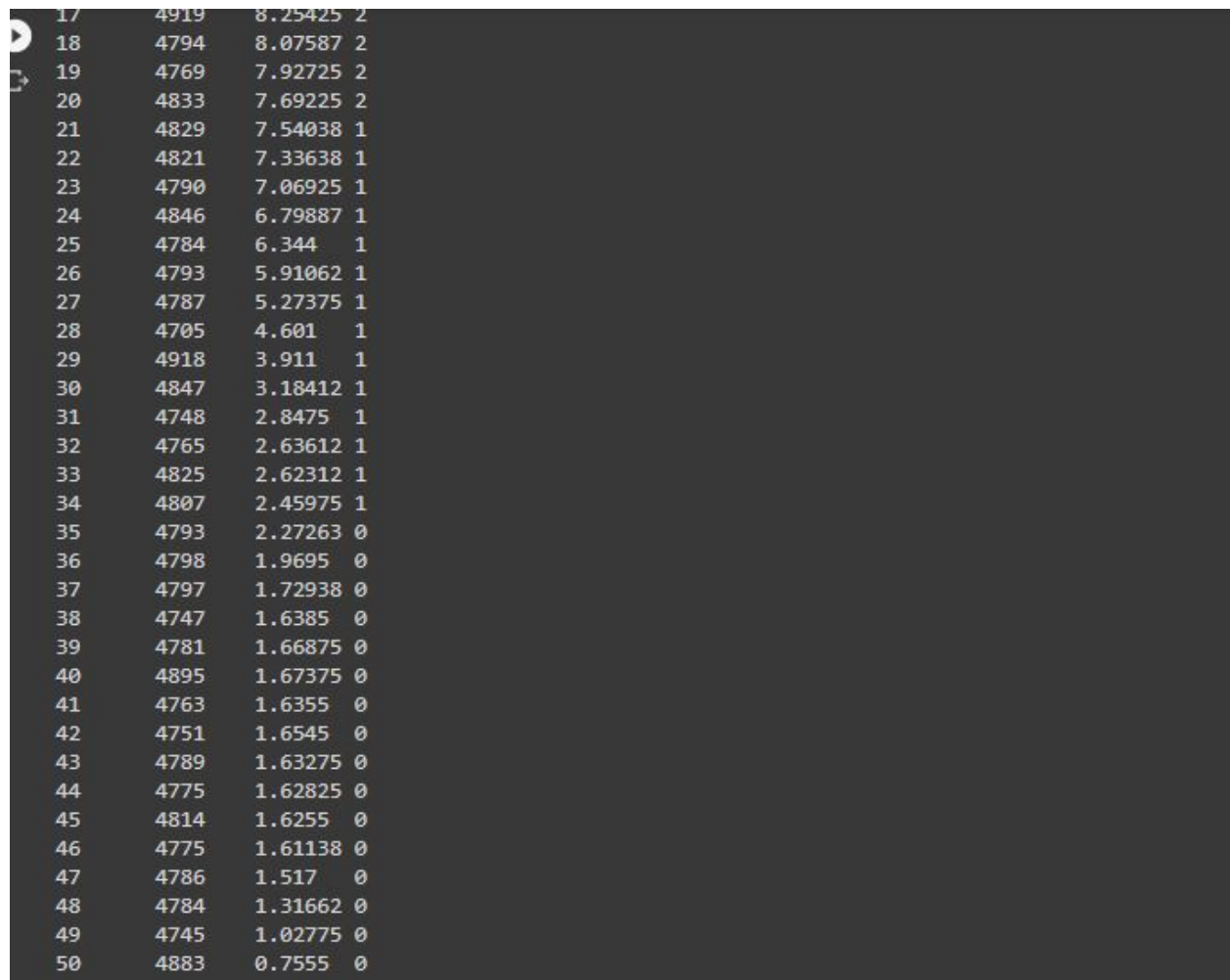
pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=10,
                              stats=stats, halloffame=hof, verbose=True)
```

gen	nevals	avg	min
0	100	9.97	6
1	56	11.72	5
2	66	12.45	5
3	70	12.86	4
4	65	13.12	4
5	56	12.81	4
6	63	12.64	4
7	71	8.27	4
8	42	7.76	3
9	64	8.01	2
10	55	7.86	2

With larger number of queens and board:

```
pop = toolbox.population(n=8000)
hof = tools.HallOfFame(maxsize=1)

pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=50,
                              stats=stats, halloffame=hof, verbose=True)
```



17	4919	8.25425	2
18	4794	8.07587	2
19	4769	7.92725	2
20	4833	7.69225	2
21	4829	7.54038	1
22	4821	7.33638	1
23	4790	7.06925	1
24	4846	6.79887	1
25	4784	6.344	1
26	4793	5.91062	1
27	4787	5.27375	1
28	4705	4.601	1
29	4918	3.911	1
30	4847	3.18412	1
31	4748	2.8475	1
32	4765	2.63612	1
33	4825	2.62312	1
34	4807	2.45975	1
35	4793	2.27263	0
36	4798	1.9695	0
37	4797	1.72938	0
38	4747	1.6385	0
39	4781	1.66875	0
40	4895	1.67375	0
41	4763	1.6355	0
42	4751	1.6545	0
43	4789	1.63275	0
44	4775	1.62825	0
45	4814	1.6255	0
46	4775	1.61138	0
47	4786	1.517	0
48	4784	1.31662	0
49	4745	1.02775	0
50	4883	0.7555	0

### **Task Division and Project Reflection:**

Our group splitted up the two parts of the project and each worked individually on them. After attempting our individual parts, we both came together to combine our work on Google Collab and discuss ways in which we can improve the implementation of our evaluation functions so it can yield the best results in the most efficient manner.

For the most part, the project was very straightforward, but figuring out how to implement the evaluation function and optimizing it so that we are effectively and

efficiently using our hardware resources was challenging. This project was very interesting as it provided us with the opportunity to deep-dive and learn more about the genetic algorithm, the DEAP library and have a better understanding of how search algorithms generally worked to help us in our future endeavors.