Cera Oh
CS541 AI
Program 2
Fall 2021
11/18/2021

Program 2 Report

Introduction and instruction how to run the program

I am turning in one C code, GA.c. *PopulationSize* and *NumIterations* are hardcoded and can be changed in the *main*() function. *MutationPct* was set to 5% and was hard coded as an integer 5, and can be changed in the *Genetic_Algorithm*() function. The code should compile and once compiled should run and output the results automatically. *PopulationSize* is set to 200 and *NumIterations* is set to 1000. The code contains *printf*() statements that tells which members are reproducing, what is the crossover point, whether mutation occurred and where for the child, and all members of the population and their fitness per generation. They made it easier for me to follow what the code was doing. Feel free to comment any of them out. Once run, the program should automatically give you an output.

Description of parameters/Techniques

Initial population was created using *rand*() function in C library and included *PopulationSize* number of members. Each individual member was represented by struct *Member*, which contains an 8-integer array variable *position* that represents its "genes" and an integer variable *fitness* that holds the fitness of the member. Each index 0 to 7 of *position* represents the columns 1 to 8, respectively, on the 8x8 board where the Queens can be located. Each number stored in each index of the array represents the row that Queen in the particular column is in. For example, if *position* array is {1, 2, 3, 4, 4, 5, 6, 8}, then column 1 Queen is at row 1, column 2 Queen is at row 2, column 3 Queen is at row 4, column 4 Queen is at row 4, column 5 Queen is at row 4, column 6 Queen is at row 5, column 7 Queen is at row 6, and column 8 Queen is at row 8. This method was shown in the textbook pg. 116.

Fitness of each member was calculated using *calc_fitness*() function and stored in the member's *fitness* variable. The fitness function I used was the number of pairs of non-attacking Queens. Queens are attacking if more than one Queen is in the same row, column, or diagonal (p.112 of our textbook) and there are known to be 28 non-attacking pair of Queens (p.116 of the textbook). Because it was easier for me to count attacking pair of Queens, the fitness function starts with the highest fitness value of 28 and subtracts 1 fitness value every time an attacking Queen was found on the right side of the current Queen that is evaluated, whether the attacking Queen is in the same row, positively slopped diagonal, or negatively sloped diagonal. For example, if column 1 was attacking column 2 and column 3 Queens, I only considered column 3 Queen when evaluating column 2 Queen since the column 1 Queen and column 2 Queen pair was already counted when we evaluated column 1 Queen previously. No Queens were ever on the same column due to the way we set up our *position* "gene" string.

Member array *population* contained all members of each generation. Pointers to *population* was used to pass and update *population* as the algorithm proceeded. *PopultionSize* variable represented the population size and stayed constant throughout the algorithm as all populations were replaced each generation. A member could be chosen as a parent to reproduce multiple times in one generation. Each reproduction produced two children with crossover point chosen using *rand*() function in the range of index 1 to 7, so no child was an exact copy of one of its parents. Any extra child that was created once *PopulationSize* has been met was thrown out. Any member that did not reproduce died off. Therefore, each generation had all new members.

Selection of parents was performed for each pair of parents using the *parent_selector*() function, which passed back the indices of the chosen two parents from *population*. The probability of selecting a parent from the population was determined using the normalized fitness calculation shown in our assignment sheet: $s_i = \frac{f_i}{\sum_{j=1}^{popsize} f_j}$ where s_i is the selection probability of *i*-th element of the population and f_i is the fitness of the *i*-th element. The sum of the fitness of the entire population was performed first. Then, I found the s_i for each member and then divided up the sum of the fitness into ranges between 0 to 99 that is proportional to the selection probabilty. Once again, I used *rand*() function to select each parent depending on which range the number fell. For example, if there was a population of 4 members with s_0 = 20, s_1 = 30, s_2 = 10, and s_3 = 40, then the sum of fitness of the population is 20 + 30 + 10 + 40 = 100. The range was divided into 0-19 representing s_0, 20-49 representing s_1, 50-59 representing s_2, and 60-99 representing s_3. If rand() chose 51 as the number, then member at index 2 (the 2nd element) would have been chosen as a parent. This method seemed to work pretty well in choosing parents for reproduction. No member could reproduce with itself.

MutationPct was an integer variable representing the mutation rate. The chance for mutation was checked for each child that was produced. Mutation chance was chosen by *rand*(). *MutationPct* was set to 5. If the number chosen by *rand*() was in the range 0-4, then mutation occurred. If it was out of the range, mutation did not occur. Mutation point was also chosen at random using *rand*() with range 0 to 7, so any of the index position in *position* can be mutated. r*and*() was used again to pick a random number between 1 to 8 to replace the value in the chosen index. I excluded the original row value stored in the chosen index from the possible choice, so mutation produced something different than what we started with. For example, if child with {1, 2, 3, 4, 5, 6, 7, 8} is to be mutated at the 1st index, which has value 2, 2 was excluded so the mutated child looked different after the mutation.

Summary of Findings and Data

MutationPct, the mutation probability, was kept at 5 while looking for possible population size, *PopulationSize,* and iteration size, *NumIterations*. Randomly generated initial populations seem to have average fitness between 19 and 21. I tried various population size and iterations. Population size of 10 did not produce good results when I tried 10, 50, 100, 200, 250, 500, and 1000 iterations. Because the population size was so small, sometimes the average fitness dropped by the end of the run and sometimes it increased by a point or two. Highest average fitness at the last generation I saw was 22 when I did 10 iterations. This population has started with average of 19.9 fitness, so it was not a good improvement. Worst performance I saw was when I did 1000 iterations and ended up with 15.5 fitness average from 18.9 initial average fitness. I noticed that the population quickly turned uniform and only changes that were being introduced as iterations went on are by mutations. I decided 10 was a bad population size. Population size of 50 produced better results. 10 iterations produced negligible differences in initial and final fitness averages. 50, 100, 200, 250, 500, and 1000 iterations all produced improvements in average fitness values between 3-6 points, which was promising. I also had a special stopping condition whenever a best fitness child is found (i.e. a solution) and this occurred for the first time when I ran 1000 iterations on population of 50. 45th generation produced one child with fitness 28. Population size of 100 produced negligible improvement with 10 and 50 iterations. 100, 200, 250, 500, and 1000 iterations all had fitness increase of about 5 points. Generation 14 in one run of 50 iterations and one run of 100 iterations produced solutions. Population of 500 produced decent average fitness increase of about 5 to 6 points in 100, 200, 250, 500, and 1000 iterations. 10 iterations also produced one perfect child in two separate runs, once in 10th generation and once in 8th generation. Overall, 500 population size seems to be doing as well as population size of 100 but taking longer computation time. 1000 population size did not perform much better and I decided it took too long without producing good results.

Here are some sample run outputs of average finesses:

* indicates solution found in at least one run out of <5 runs			
Population size	Number of Iterations	Sample Average Initial Fitness	Sample Final Generation Average Fitness
10	10	19.9	22.0
	50	21.1	21.1
	100	20.1	24.0
	200	20.8	19.6
	250	20.2	21.2
	500	19.9	17.0
	1000	18.9	15.5
50	10	20.5	20.6
	50	20.4	24.5
	100	19.9	26.3
	200	19.8	23.6
	250	20.1	23.6
	500	20.0	23.7
	1000*	20.4	24.9
100	10	20.2	20.9
	50*	20.0	22.7
	100*	17.8	24.9
	200	20.0	24.9
	250	20.3	25.4
	500	20.3	26.5
	1000	20.2	24.8
500	10*	20.0	20.2
	50	20.1	23.1
	100	20.2	25.7
	200	20.0	25.7
	250	20.1	25.4
	500	20.3	25.7
	1000	20.1	25.5
1000	10*	20.1	20.4
	50	20.2	21.6

	100	20.1	23.3
	200*	20.1	23.8
	250*	20.1	25.3
	500	20.1	26.0
	1000	20.1	25.1

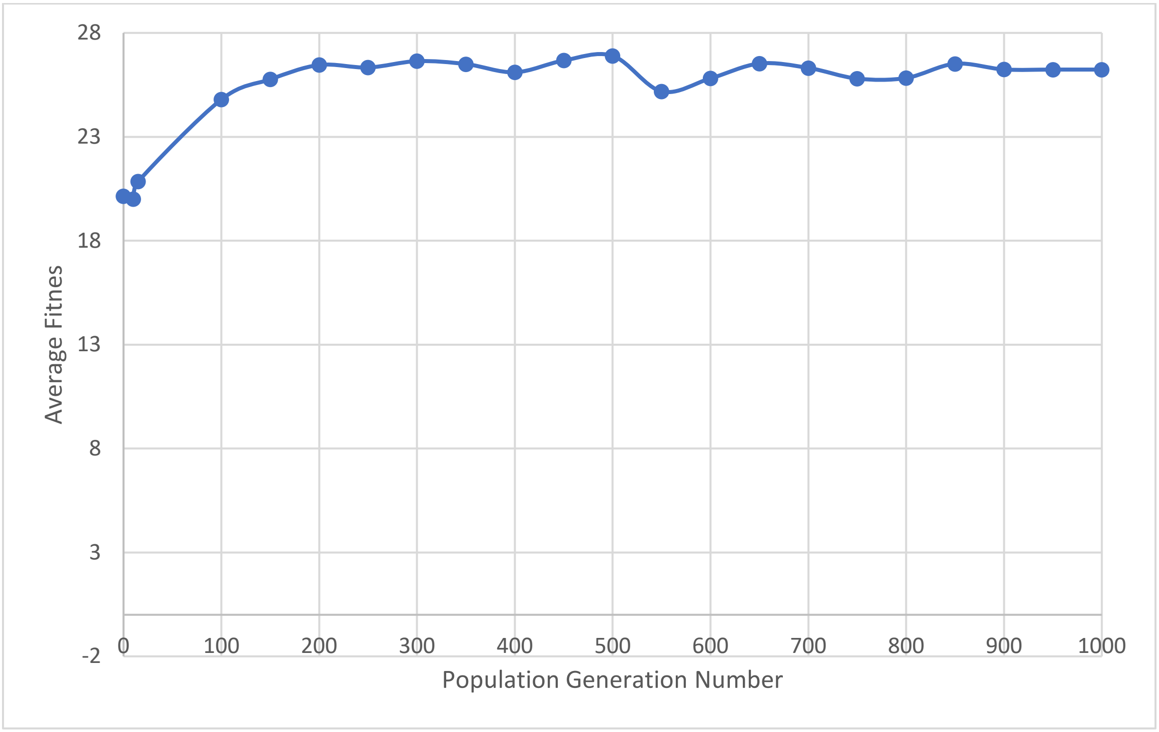
Therefore, I chose population size of 200 as my population size and for 1000 iterations as my iteration size. My runs showed that a solution can be found in 50 or even 100 iterations of population size 100 from running the algorithm a few times, so the chosen size and the iteration number seemed good. I then tried adjusting the mutation size to see if the algorithm behaved better. I increased the mutation probability to 10% and realized it added too much randomness and average fitness dropped. Mutation probability of 8% was okay, but still did not perform as well as 5%. Probability of 3% seems to not work as well as 5% as not enough randomness was introduced. I decided 5% mutation probability gave us good enough randomness to get Genetic Algorithm to close to the goal as possible.

My algorithm stops once a solution is found or if 1000 iterations have been reached. Some runs of the algorithm found a solution fast in few generations, while others converged around average fitness value between 24 and 27 and never was able to find a solution. Here is a data chart of 50 runs using above parameters:

Run #	Initial Population Average Fitness	Final Generation Average Fitness	Solution Found?
1	20.139999	26.235001	No
2	20.340000	25.004999	No
3	20.230000	Stopped at Gen 37: 22.645000	Yes: {6, 8, 2, 4, 1, 7, 5, 3}
4	20.445000	26.430000	No
5	19.760000	25.450010	No
6	19.975000	26.275000	No
7	20.170000	24.100000	No
8	20.100000	25.950010	No
9	20.180000	25.780001	No
10	20.235001	Stopped at Gen 16: 22.315001	Yes: {3, 7, 2, 8, 5, 1, 4, 6}
11	20.129999	Stopped at Gen 14: 21.388999	Yes: {4, 7, 3, 8, 2, 5, 1, 6}
12	20.200001	Stopped at Gen 10: 21.120001	Yes: {6, 8, 2, 4, 1, 7, 5, 3}
13	20.410000	26.070000	No
14	20.245001	24.635000	No
15	20.135000	26.129999	No
16	20.129999	24.295000	No
17	19.795000	25.959999	No
18	20.254999	26.385000	No

19	20.320000	24.530001	No
20	19.934999	Stopped at Gen 40: 21.815001	Yes: {6, 3, 7, 4, 1, 8, 2, 5}
21	20.030001	24.459999	No
22	20.030001	Stopped at Gen 17: 21.309999	Yes: {5, 3, 8, 4, 7, 1, 6, 2}
23	19.959999	Stopped at Gen 22: 21.225000	Yes: {5, 8, 4, 1, 3, 6, 2, 7}
24	20.090000	25.905001	No
25	20.150000	24.809999	No
26	20.174999	24.299999	No
27	19.965000	25.420000	No
28	20.165001	26.110001	No
29	20.035100	Stopped at Gen 37: 22.430000	Yes: {4, 7, 5, 2, 6, 1, 3, 8}
30	20.030001	23.985001	No
31	20.070000	24.099999	No
32	20.334999	25.750000	No
33	20.459999	24.965000	No
34	20.350000	25.450001	No
35	20.105000	24.395000	No
36	20.205000	25.715000	No
37	19.920000	Stopped at Gen 31: 21.655001	Yes: {1, 7, 4, 6, 8, 2, 5, 3}
38	20.360010	25.700001	No
39	20.305000	25.020000	No
40	20.084999	25.924999	No
41	19.870001	26.860001	No
42	20.145000	26.129999	No
43	19.855000	25.040001	No
44	20.295000	26.459999	No
45	20.290001	24.495001	No
46	20.059999	25.405001	No
47	20.135000	25.370001	No
48	20.115000	24.815001	No
49	20.325001	25.009999	No
50	20.045000	Stopped at Gen 590: 26.705000	Yes: {3, 6, 8, 1, 5, 7, 2, 4}

I used the first run that ran to 1000 iterations (i.e. no solution found) for graphing and data collecting purposes. Here is a graph of Population Generation Number v. Average Fitness graph for a sample run # 1 with 1000 iteration of Genetic Algorithm with 200 initial population and mutation probability of 5%:



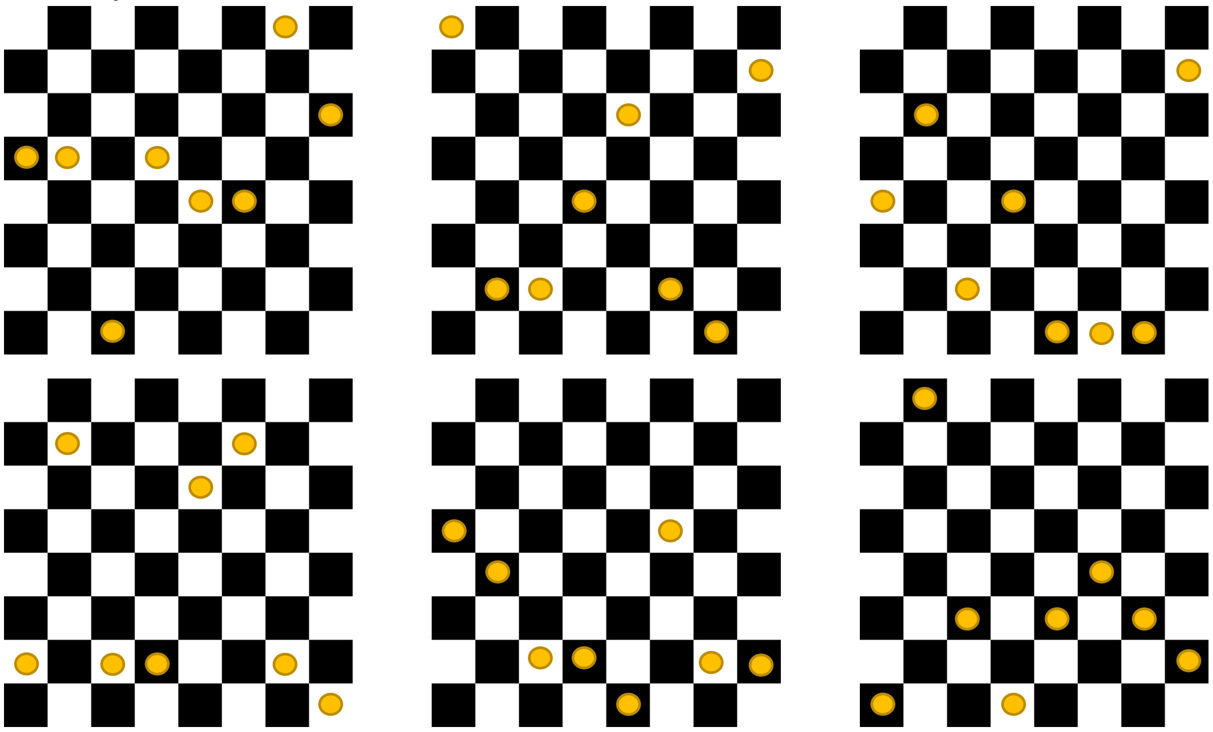
The data chart for the graph:

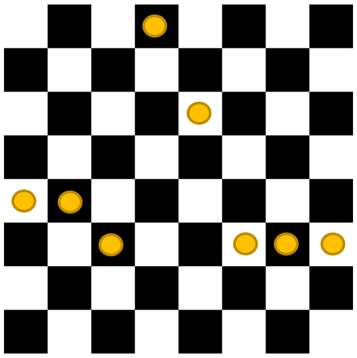
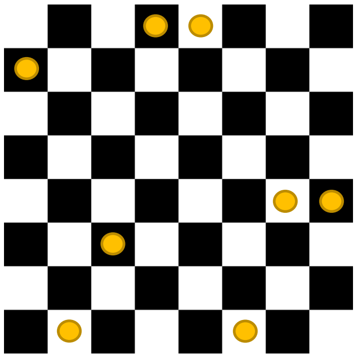
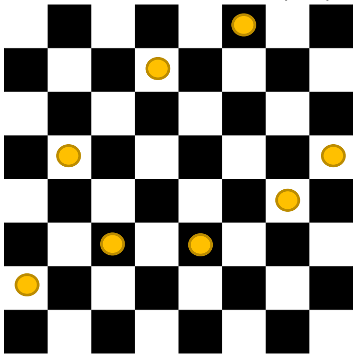
Population Generation Number	Average Fitness
0	20.180000
10	20.000000
15	20.850000
100	24.795000
150	25.764999
200	26.455000
250	26.334999
300	26.639999
350	26.500000
400	26.105000
450	26.670000
500	26.895000
550	25.184999
600	25.809999
650	26.525000
700	26.315001
750	25.799999

800	25.830000
850	26.514999
900	26.240000
950	26.235001
1000	26.235001

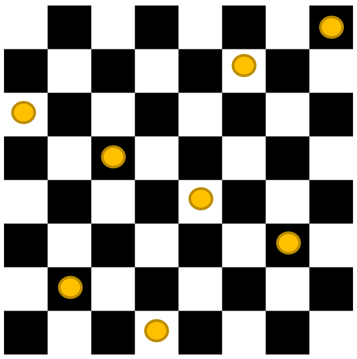
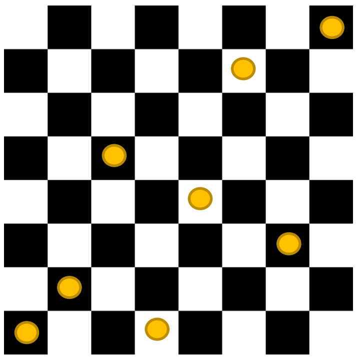
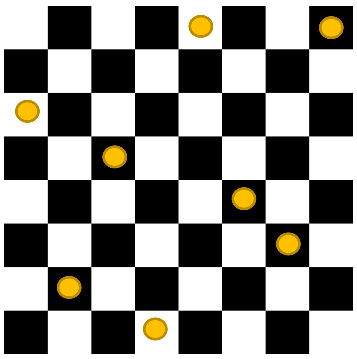
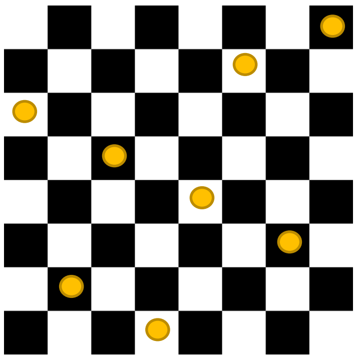
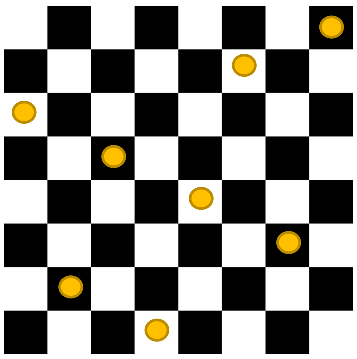
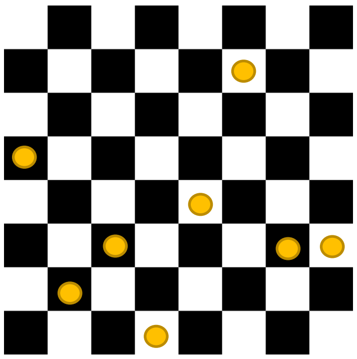
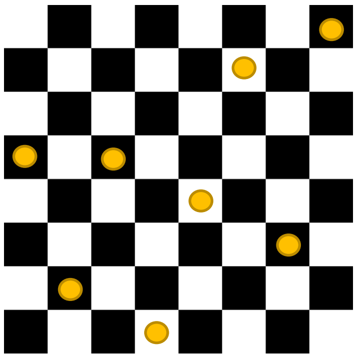
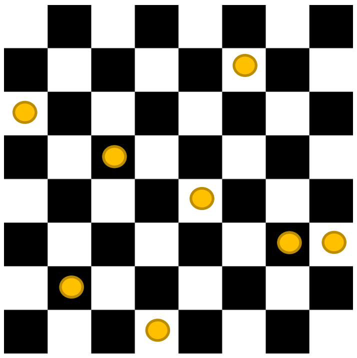
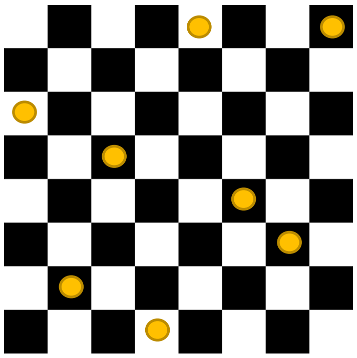
As you can see, the population starts with average fitness of 20.18 and steadily increases so its average fitness is at 24.795 by the time it reaches Generation 100. The average fitness still increases slowly until it reaches around 26.455 at Generation 200. Then it plateaus and fluctuates a little between 25.5 and 26.5 for the rest of the Generations. This population was never able to find a solution. Once a population reaches its maximum fitness through reproduction and crossovers, it plateaus as most members of the population now has same genetic makeup. Only mutation introduces enough randomness to sometimes push the population's average fitness slightly down or up. Basically, a right mutation must occur at this point in order to find the solution, which does not occur too often. Sample population of generations of this run is shown below. I have grabbed random 9 samples from the population to show for the initial population, generation 150, generation 200, and final population. Yellow dots represent the Queens:

Initial Population

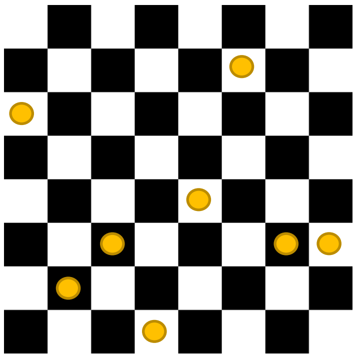
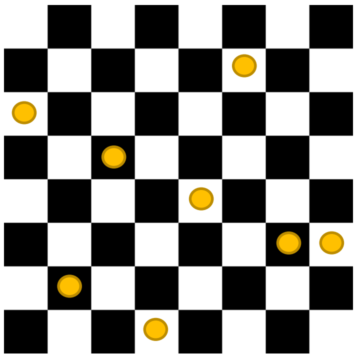
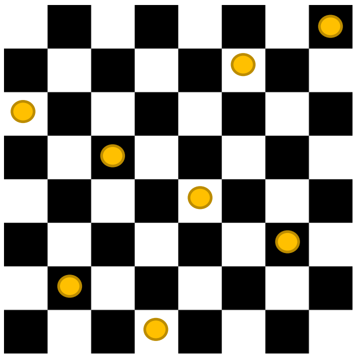
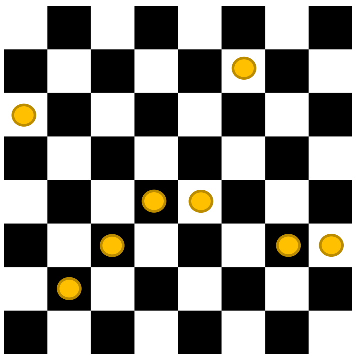
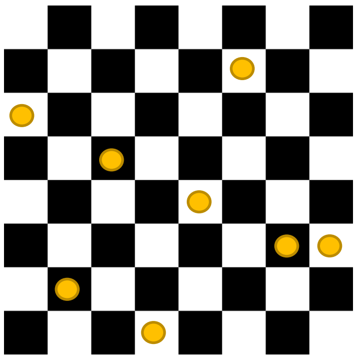
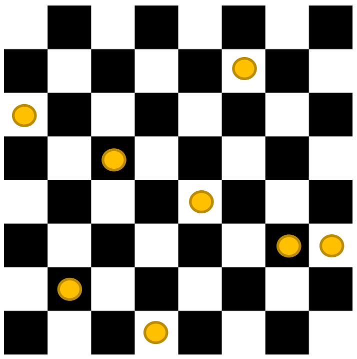
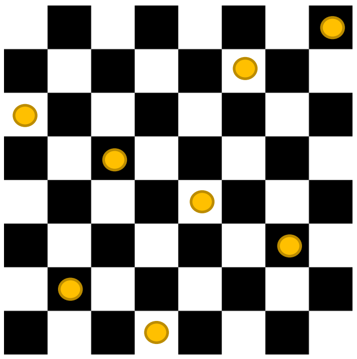
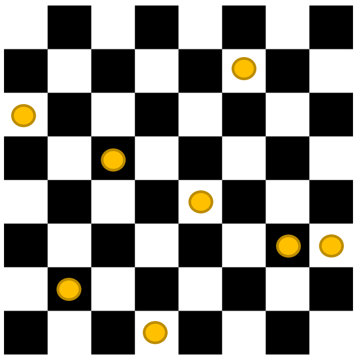
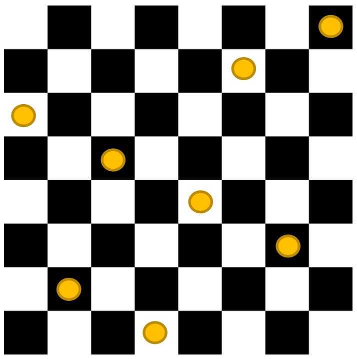


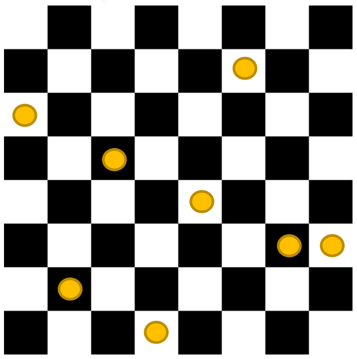
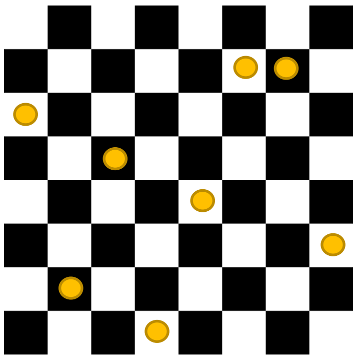
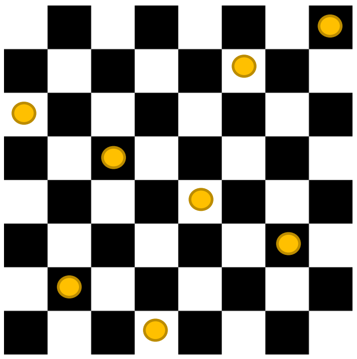
Generation 150

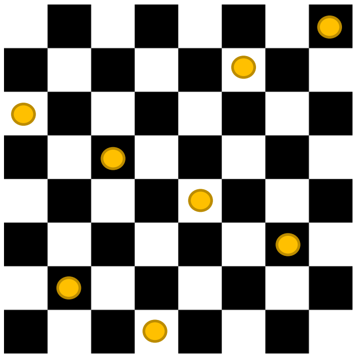
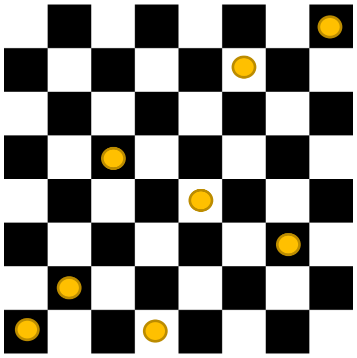
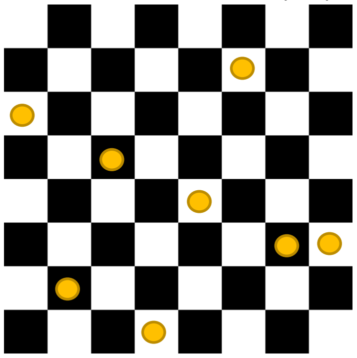
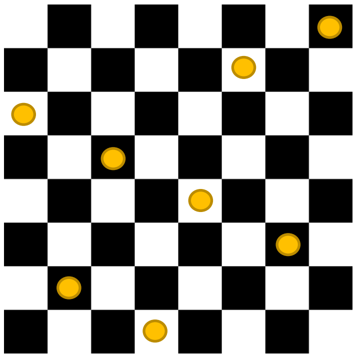
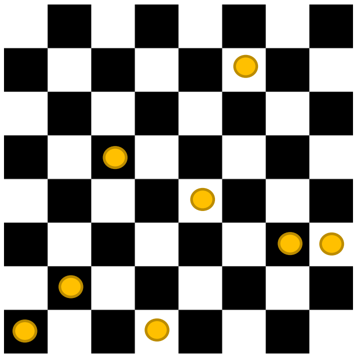
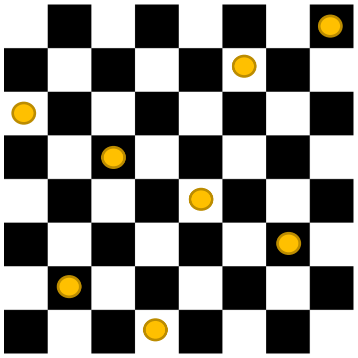










Generation 200

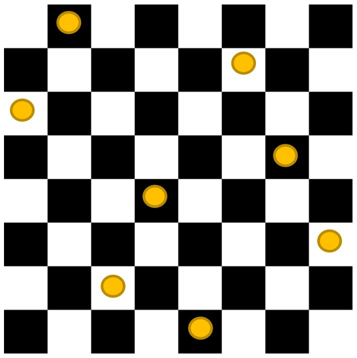
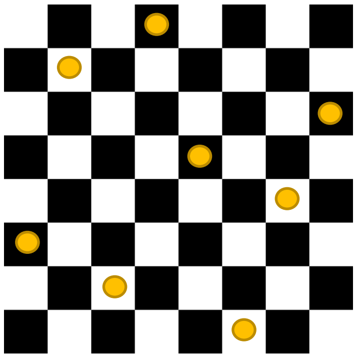
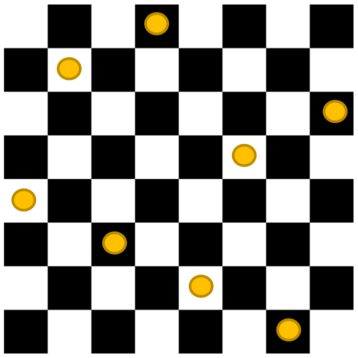
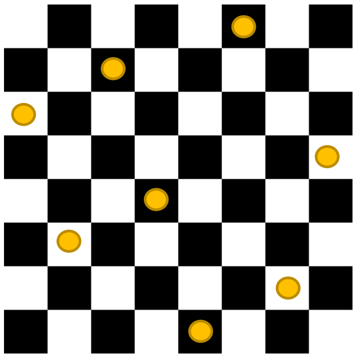
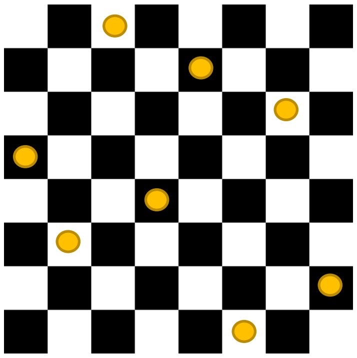
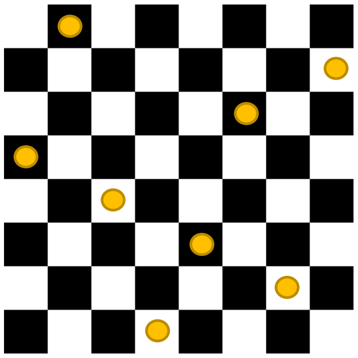










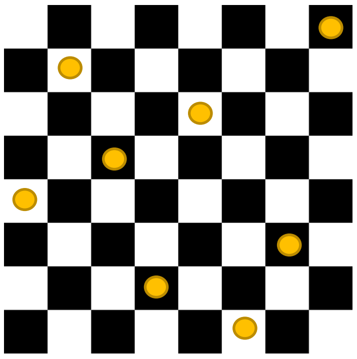
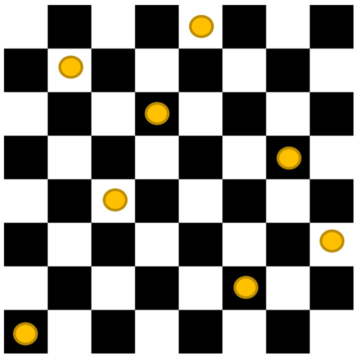
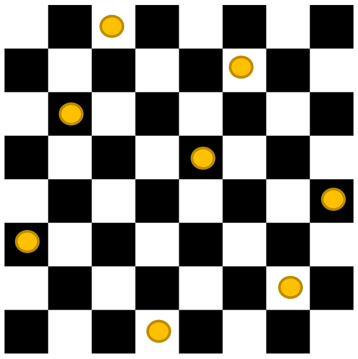
Final Population

Here are some of the solutions found using the GA algorithm using the same parameters:

Conclusion

I believe my choice of 200 population size and 1000 iteration with 5% mutation probability was good. Genetic Algorithm with random selection of parents depending on fitness probability does a good job raising the average fitness of the population to near optimal. Since chance of reproducing increases with fitness, individuals with higher fitness end up reproducing more and leaving more children, thereby taking over the population. Most of time, reproduction and crossover will produce children who are near the solution, even sometimes producing children who have fitness of 27 and just need one or two gene changes to reach the solution. Average fitness increases steadily when there are more genetic variations in the population, so in the beginning we see a steady increase in the average fitness. The average fitness for the population plateaus once the population has been mostly taken over by the progenies of the most fit individuals, in which point crossover no longer does much to change the genetic makeup of the children. Once the population plateaus, only change that can be introduced are occasional mutations, which occur at a low level at random location of genes to change the chosen gene to a random value. I have found that if random mutation or crossover is to find the solution, it usually occurs early in the generations. By the time the genetic makeup of the population has turned mostly uniform, waiting for random mutation to produce the solution takes a long time and usually does not occur within 1000 generations due to the randomness of the mutation. Once in a lucky while right mutation will show up and a solution will be found.