N Queens Problem with Genetic Algorithm

Kerem Yıldırır, Orhun Barış, Berkant Deniz Aktaş, Çağrı Uluç Yıldırımoğlu

May 19, 2018

1 Problem Description

N-Queens is a puzzle where the player can place N queens in a N x N chessboard such that no 2 queens be able to attack each other. In chess, queen is the piece which is able attack all the directions. For example, a solution of N-Queens problem with N = 4 is as following :

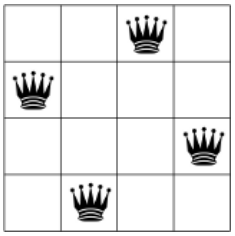

Figure 1: N-Queens solution with N = 4.

N-Queens problem is famous in Computer Science community since it is simple, but nontrivial. There are a lot of different approaches for solving the problem such as backtracking,local search algorithms, and genetic algorithms. The problem is first created by Max Bezzel in 1848 as 8-Queens puzzle. It's first solutions were published in 1850 by Franz Nauck, who extended the problem as N-Queens. Ever since, mathematicians worked on both N-Queen and 8-Queen problem, trying different approaches.

2 Algorithm Description

Genetic algorithms(GA) are optimization algorithms which are inspired by Darwin's theory of evolution. Algorithm starts with a randomly created set of initial states(populations). After creating each state, their success is evaluated by a fitness function. This functions determines how fit that population is, and sorts them regarding their fitness level. Then a fraction of the population's best candidates are selected for mating(natural selection). Fitter populations have a higher chance of being selected than the ones with low fitness. Though, some bad populations are also selected for mating in order to avoid getting stuck at local optimas. After selecting parents, a crossover operation is applied on parents and non-selected members of the population are replaced by the children of selected parents, which are produced by combining the parent populations between each other. After mating, a random change, mutation, is applied on the populations with a predetermined probability. The application of mutation is crucial since it adds diversity to the population, hence, preventing the algorithm from getting stuck at a local minimum or maximum. After mutation, a new generation emerges and algorithm evaluates the fitness of each population. If the fitness is at desired level, the algorithm converges. Else, it keeps generating new generations until convergence. Convergence may never happen depending on population size and mutation rate.

2.1 Initializing Populations

In the application of the N-Queens problem with GA, each population is represented as a state of the chess board. Hence, while creating initial populations we created initial board states as shown below.

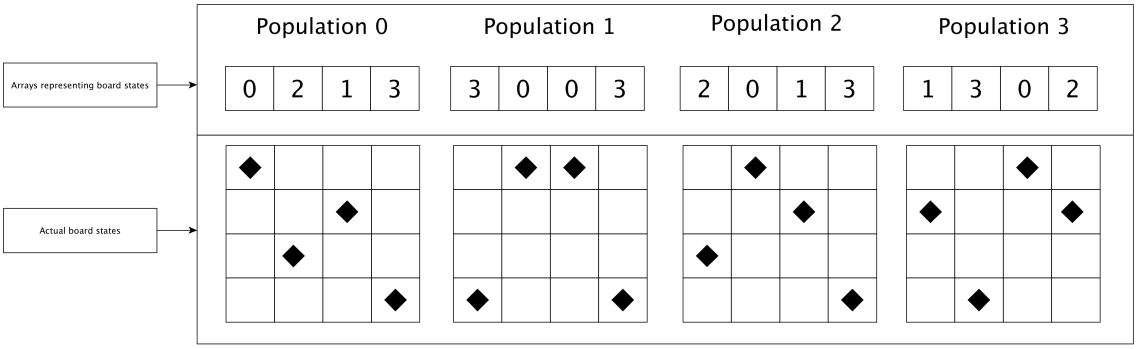

Figure 2: 4 sample populations initialized

2.2 Fitness Evaluation

After creating each member of the population, we need to determine their fitness level. In this application of the GA, fitness is determined by calculating the total number of conflicts in the chess board(number of queens attacking each other).

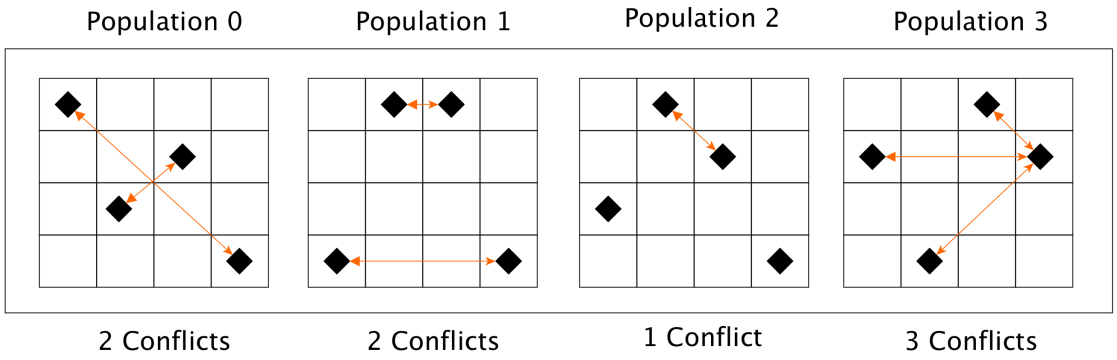

Figure 3: Fitness level of each sample

2.3 Creation of a New Generation

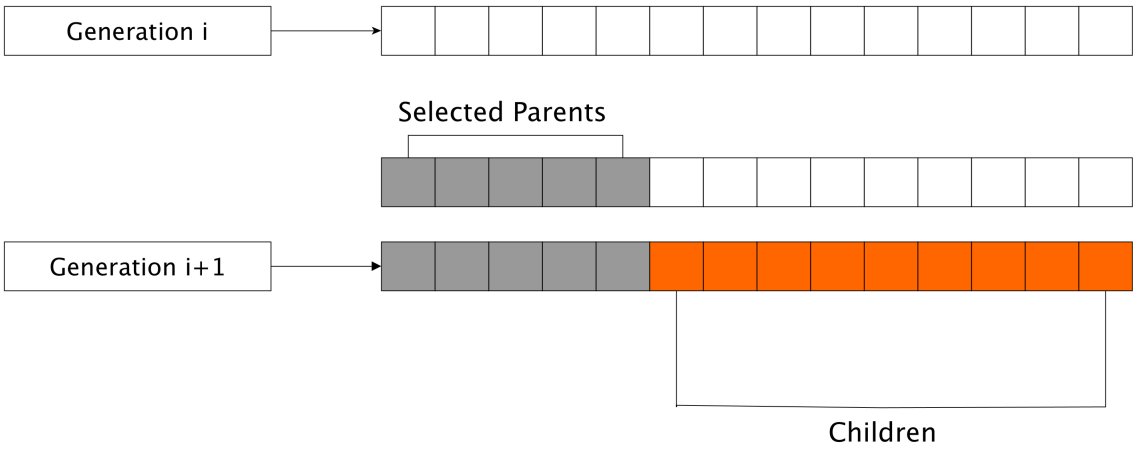

Figure 4: Emergance of the new generation

After determining how fit each population is, we sort them regarding their fitness levels and select $PopSize * SurvivalRate$ of them to be parents. Then, reproduce the rest of the population by mating selected parents. Number of children created by each selected parent is calculated by :

$$numberOfKids = \frac{1}{SurvivalRate} - 1$$

2.4 Mating of Parents

In GA, in order to produce the next generation, we need to produce offsprings from our selected parents. There are various cross-over functions in GA. The one we used is single point cross-over. It specifies a random position r then takes $[0, r]$ of the first parent,$[r, N]$ of the second parent.

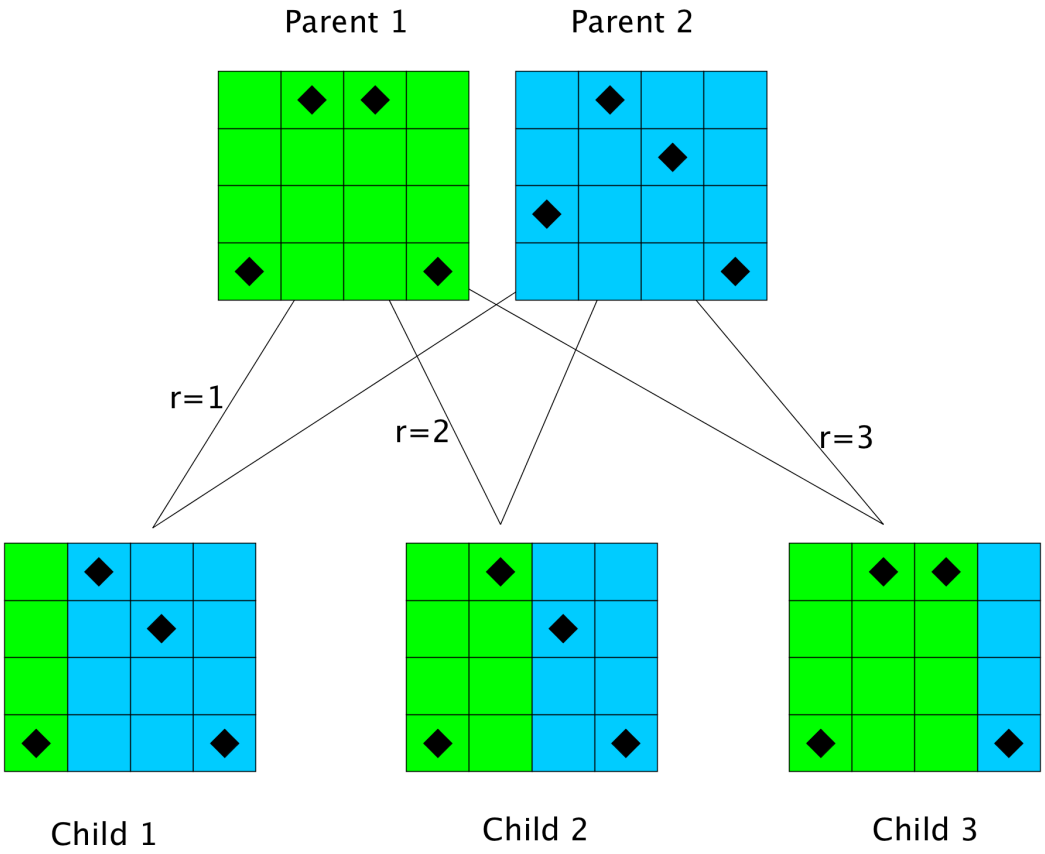

Figure 5: Mating

3 Algorithm Analysis

Algorithm has 4 parameters that contribute to the complexity. Initialization takes $O(PopSize*N)$, since we randomly fill a array of size PopSize with arrays of size N. In the main algorithm, in order to measure fitness,we need to evaluate the populations fitness level by looking their number of conflicts. This is done by a Quicksort algorithm which sorts an array of size PopSize and for comparison evaluates the number of conflicts in $O(N)$ time. Worst case for sorting is $N^2 * PopSize^2$. Mutation takes constant time and applied for each population. Hence, it takes $O(PopSize)$ time. Dominating part of the algorithm is sorting. This complexity is applies to all generations. Hence, total complexity is $O(N^2 * PopSize^2 * (\#ofgenerations))$.

4 Experimental Analysis

N	Mean Time (s)
4	0.00679612
8	0.22841875
10	0.22841875
15	0.62138629
20	0.99475187
25	1.71172667
30	3.181633
40	7.40939903
50	17.78323555

Figure 6: Test results.

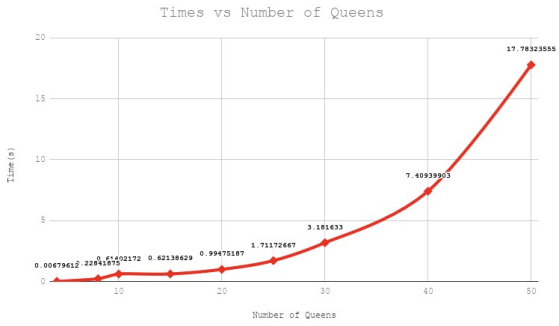

Figure 7: Statistics

5 Testing

For testing, we decided to use Black Box technique to determine cases for program's inputs and variables. In this part, we will try to explain equivalence partitioning and boundary value analysis.

5.1 Equivalence Partitioning

From defination of n-queens problem, N values which are smaller than 4 are invalid for program. This is because of NxN board size, creating a board which is smaller than size 4 will cause problem to be unsatisfiable. Thus, valid N values are the ones which are bigger than 4. In addition, validation of genetic algorithm parameters are required. In order to find valid values for *PopSize*, *MutationRate* and *SurvivalRate*; we changed parameters and tested our program with different combination of these corresponding parameters.

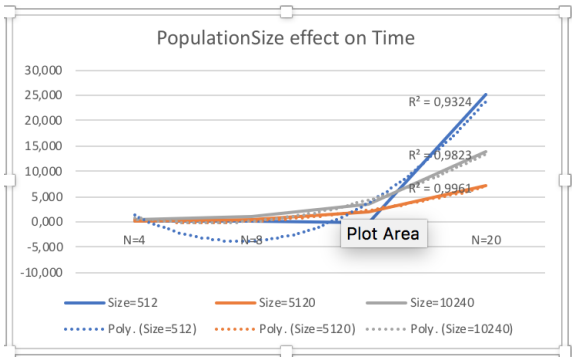

Figure 8: PopSize Effect On Time

Figure 8 is an instance of our testing data which shows change on running time for different *PopSize*s and constant *MutationRate* and *SurvivalRate*.

As can be seen in the graph, too low and too high population sizes increases the running time. Too low *PopSize* provides little diversity in the population, resulting in creation of too many generations. In fact for N larger than 20, time was measured too high to graph. Too high *PopSize* provides more diversity but it does not compansate for the amplification of calculations related to *PopSize*. For our tests, we used a formula to find a reasonable *PopSize* which is stated in section 5.2.

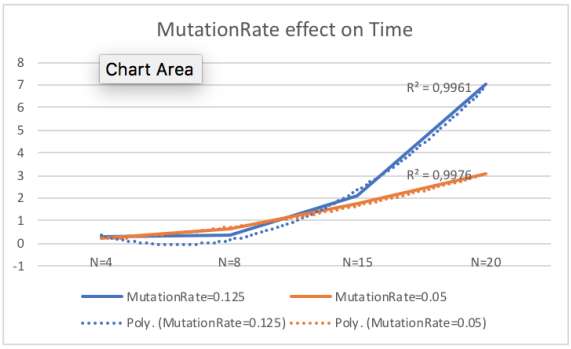

Figure 9: MutationRate Effect On Time

Figure 9 shows the relation between $MutationRate$ and time. These rates are constant for each N and as can be seen, high mutation rate increases the run time. Since the algorithm depends on children of fittest parents being even fitter, too many mutations cause unwanted randomness. This results in a nearly random search[1]. But as a reasonable $MutationRate$ is needed so that populations do not get stuck at local minima, considering the gradient based heuristic is used as in our algorithm, a formula that guarantees a minimal number of mutations is given in section 5.2.

5.2 Boundary Value Analysis

Depending on the result in 5.1 and as it is stated in various papers [1][2], we choose to use two functions to determine the values of $PopSize$ and $MutationRate$.

$$PopSize = max(min(N * N, 10240), 512);$$
$$MutationRate = 1.0/(double)N$$

Depending on these formulas, we created bounds for parameters as :

$$SurvivalRate = max(min(SurvivalRate, 1.0), 0.0)$$
$$MutationRate = max(min(MutationRate, 1.0), 0.0)$$
$$PopSize = max(min(PopSize, 10240), 512)$$

For instance, when program is computing a population for N = 10 queens, even thought $N*N = 100$, $PopSize$ will be 512. It is an example for a case which is lower than lower bound of $PopSize$. With similar reasoning; when N $>$ 100, $N * N$ will be greater than upper bound so it will be reduced to 10240.

These bounds are important because solution speed strongly depends on the parameters of genetic algorithm.

5.3 Correctness of Genetic Algorithm

In this section, we tested our program to see if it can solve all possible combinations on the board. Since there are lots of random values in the definition of Genetic Algorithm and exact different solution count depends on $Mutations$ and $CrossOvers$, we created so many cases that the number is bigger than the maximum number of different scenarios. After choosing $N = 4$ and running the program 10.000 times. Genetic Algorithm successfully computes all possible solutions.

Statistics of 10.000 runs for 4 Queens						
Mean	Std Dev	Std Err	CL90	CL95	CL99	CL99.9
0.00929150s	0.00385317	0.00038532	0.00865573- 0.00992727	0.00853628- 0.01004672	0.00829738- 0.01028562	0.00802342- 0.01055958

6 Conclusions

Genetic algorithm approach to the n-queen problem works with n larger than 30, which makes it a better approach than backtracking approach for finding single solutions to the problem. Backtracking approach finds all possible solutions but becomes infeasible when N>30 [2].

References

[1] Grefenstette, John J. "Optimization of control parameters for genetic algorithms." IEEE Transactions on systems, man, and cybernetics 16.1 (1986): 122-128.

[2] Thada, Vikas, and Shivali Dhaka. "Performance Analysis of N-Queen Problem using Backtracking and Genetic Algorithm Techniques." International Journal of Computer Applications 102.7 (2014).