Alex Pendell

CIS 421 – Artificial Intelligence

Assignment 2 – NQueens

September 27th, 2019

**The N-Queens Problem**

**I. BASIC PARAMETERS**

For this assignment, we used the parameters given via the assignment writeup by Dr. Grabowski. The population size was not changed from N \* 10, the mutation rate was kept at 10%, and the size of the mating pool remained at N (that is, if N was 8, there would be a population of 80, and 8 individuals would be chosen for generating offspring.)

The program works by first creating a population in which to generate solutions. The population is an array of objects called Workers, where the genotype of each worker is a potential solution to the N-Queens problem. When a Worker is inserted into the population array, the index at which it is inserted is its ID number. Because the method used for replacement is Age-Biased, keeping track of where each Worker is in the array make it easier to replace them when they get replaced by their children.

Due to a possible oversight when interpreting the instructions, the only liberty that was taken regarding the parameters is that the program runs for 1,500 generations instead of the given 1,000. This choice will be explained in Section VI. Conclusion & Closing Thoughts.

For the sake of simplicity, much of the algorithms implemented are dependent on what N was chosen to be. This enhances scalability should N be chosen to be large or small.

**II. PARENT SELECTION & PAIRING**

Parent selection was done by first initializing an array of size N (or N + 1 if N is odd) to be our parent pool. Inside this array is where we keep the IDs of all the Workers who have been selected to become parents. Afterwards we must start selecting candidates to fill the parent pool. This is done by selecting three applicants at random, then choosing the applicant with the highest fitness. Something worth noting is that when an applicant is selected, that Worker’s ID is added to a Set. When randomly more applicants, we first check to see if the randomly selected ID is in the Set. If it exists within the set, that means that we have already selected this applicant once before, and we will select another. This means that parent selection was done without replacement.

Once we have filled our parent pool, we pair parents based on subsequent positions within the array. That is, the Worker at position 0 in the array will create two children with the Worker at position 1, the Worker at position 2 will pair with the Worker at position 3, and so on. Two parents will spawn two children.

**III. CREATING CHILDREN & SUVIVOR SELECTION**

The creation of children is done via Crossover. A breakpoint is chosen at random from the set [0, N). Two children arrays are initialized to serve later as the genotypes for the children when they are created near the end of the process.

As stated in the previous section, two parents will spawn two children. The first child’s genotype will contain a copy of the first parent’s genotype up to the breakpoint. It is similar for the second child and the second parent. After the child arrays have been filled up to the breakpoint index, we iterate through the unused parent’s array to fill in the rest of the child’s array with the missing values. That is, the first child’s array will get filled in with the missing values in the order that the appear in the second parent. The process is repeated for the second child and the first parent.

It is at this point that it is decided whether mutation will occur. A random integer is generated from the set [0, 10). If a 0 is chosen mutation will occur. A number is chosen for the first and second child individually so mutation can occur on an individual basis.

Once we have the genotypes and mutation, we construct a worker using the children’s genotypes. A Boolean is passed into the Worker constructor that let’s the constructor know to mutate this genotype. When a mutation occurs, two indices are selected at random and the value at their positions are swapped.

For this assignment, we used Age-Bias as a form of survivor selection. This means that after the child Workers have been created, they replace their parent’s positions in the population array. This is done by assigning the child Workers IDs to be the same as their parents, and then replacing them within their parent’s positions in the population array.

**IV. CONFLICT CHECKING**

This section was saved for last as it will most likely require the most explanation. There are two main assertions that went into designing the algorithm:

1. We encoded our solutions via an array that contains the values 1 through N where the index determines the column, and the value at that index determines which row the queen sits. This means that since we can only have one value in any index, we will never have more than one queen in a column. And since we do not allow duplicate values in the solution array, we will never encounter a row with more than one queen. Given this information, the first assertion is that *if a collision occurs, it will be along a diagonal.*
2. We count a collision between queens as one collision. So, if Queen 1 collides with Queen 2, we still only count this as a single collision even though it can be said that Queen 2 collides with Queen 1 as well. Since we check for collisions from left to right in any given solution array, we make our second assertion that *we can disregard any collisions that come from the left*, as we would have detected those already in previous iterations of the collision checking. Any new collisions must come from the right direction.

With these in mind, the collision detection was simple. After analyzing what it means to collide along a diagonal, a straightforward algorithm can be utilized. Since we can only collide diagonally: given a value at any index in the array, if any subsequent values in the array are equal to the original value plus (or minus since diagonals go both up and down) the indexed distance within the array, there is a collision. For example: Given a potential solution array:

[1, 2, 6, 7, 3, 5, 8, 4]

We can detect first that the first queen collides with the second: The value in the first position (1) plus/minus the index distance to the second position (1) equals the value in the second position (2). This means that these queens are along a diagonal. Likewise, the queen in the fourth position of the array collides with the queen in the sixth position. The value at the fourth position (7) plus/minus the index distance to the sixth position (2) equals the value in the sixth position (5). These queens are also along a diagonal.

After we detect the collisions for a queen for an index, we never have to visit that queen again thanks to the second assertion that if this queen collides with the other queen, the other queen also collides this queen. Put simply, once we reach a position in the array, we don’t have to check to see if a queen collides with any queens who inhabit a previous index since they will already have been counted.

**V. DATA COLLECTION**

When executing the program, it will prompt the user how many queens to give it, as well as how many experiments to run. After the program has completed, there will be an output file in the /src/ directory titled ‘results.txt’. This file will contain the results of the experiments completed for the given N and number of experiments chosen.

For our experiments, we chose N to be 12 and ran 25 experiments with this value. The results are as follows:

**VI. CONCLUSION & CLOSING THOUGHTS**

In conclusion, there a few design choices that perhaps were a detriment to the performance of the algorithms. The first being that the population can contain duplicate values for its Worker’s genotypes. It’s perhaps a small variance in time but requiring duplicate genotypes to be recreated before adding them back to the population would no doubt decrease its convergence time as more genotypes would be exposed to the population, thus increasing the probability that a solution will be found in a reasonable amount of generations. As N grows, however, this becomes less of an issue as the probability of stumbling across duplicate genotypes substantially decreases once N reaches a high enough value.

Nonetheless, this is the reason that the program runs for 1,500 generations instead of 1,000. The extra 500 generations, we believe, allows decent scrutiny of the implemented algorithms.

It does still feel slightly slow compared to the expected output the 50% solution production rate. Since we are using Age-Bias, the convergence rate is already slowed compared to finding and replacing the lowest fitness’ individuals in the population pool. Couple this with how duplicates are allowed within the population, and it’s certain that convergence times will be noticeably slowed.