

Exploring Permutation-Based Mutation Operators with the N-Queens Problem

*Scott H. Douglas¹
Rochester Institute of Technology, Rochester, NY
February 7, 2005*

Abstract

Mutation is a key component for a working genetic algorithm (GA). In the realm of genetic algorithms, encoding of the problem can take the form of bit strings, permutation arrays, or even parse trees. Mutation for bit strings is elementary, but mutation for permutation based problems is not so straight forward due to the restriction of preserving proper permutations. This paper examines a few methods of mutation for a permutation based GA that solves the N-Queens problem.

1. Introduction

The N-Queens problem attempts to place N queens on an $N \times N$ chess board, where no queen is attacking any other queen. The N-Queens problem has a solution for all boards of size $N \geq 4$. Other researchers have found a constant time algorithm for placing queens on the board. This solution lacks variability and trivializes the N-Queens problem if only one solution for a given board size is required. GAs can find multiple solutions if they are run multiple times on the same problem. Solving the N-Queens problem is largely an exercise of the brain and lacks much useful application.

Mutation is one major method that GAs use to converge on a solution. Mutation for permutation based GAs is the process of randomly exchanging elements of the permutation. Some methods of mutation completely change the permutation. Others try to preserve relative orders of the elements.

Since fitness evaluations are costly, they are a good metric of how a GA is performing. By running the N-Queens problem with different mutation operators, we hope to find sweet spots for mutation operators that minimize fitness evaluation calls and to provide empirical data for comparing operators.

2. Software Design

2.1. Functional Description

The GA system must evolve a solution to the N-Queens problem in the form of a permutation.

2.2. System Overview

The system implemented is similar to the system implemented in "Crossover in Function Optimization" by Scott Douglas. Generic GA classes were created for Chromosomes and Populations.

Chromosomes were permutation based instead of bit string based. Instead of an array of bits, the chromosomes consists of an array of integers. This makes crossover and mutation easier. In this system, it is impossible to morph a permutation element into another element. You can only shift elements. This ensures that the permutation is always valid.

Interfaces were created for defining custom crossover, fitness, and mutation functions. To use the system to solve a problem, simply provide a crossover function, a fitness function that measures the fitness of the permutation, and a mutation operator. Pass these parameters into a Population instance, which randomly generates the initial population. At this point, statistics about the population can be queried.

To advance to the next generation, call `nextGeneration()`. This function copies a certain percentage of individuals to the next generation based on fitness. For the remaining individuals, tournaments of two individuals are held to find two parents. Crossover is performed based on the crossover rate. This is done until the new generation is full. Mutation is employed based on the mutation rate parameter. The next generation process is repeated until the solution is found or a maximum number of fitness evaluations has been exceeded.

When gathering statistics, such as the average number of fitness evaluations required, the system runs the GA multiple times based on the repeat rate.

¹Send correspondence to shd0326@cs.rit.edu.

2.3. Initial Parameters

- Crossover Rate: 0.7
- Crossover Type: PMX
- Mutation Rate: Varied
- Max. Fitness Calls: 300,000
- Population Size: 50
- Chromosome Length: 32 (A 32×32 board)
- Copy Rate: 0.1
- Repeat Rate: 5
- Measurements: Average

2.4. Crossover

The system used PMX crossover. A random index is generated between 0 and $N - 1$. The permutation elements at that index are recorded from both parents. Those elements are swapped among both chromosomes. Refer to Figure 1 for a visual explanation. This process was repeated N times.

2.5. Encoding

Permutations of the range $[0, N - 1]$ are encoded as a series of N integers in a chromosome. Each integer represents the row a queen should be placed on. The column is derived from the elements index in the chromosome integer array. Due to this method of encoding, no queen can ever attack another queen by moving on a row or column since element indices and permutation elements will both be unique sets. Due to this, the fitness function only needs to check diagonals for possible queen attacks.

2.6. Fitness Evaluation

The GA system attempts to minimize the number of queens being attacked. It is calculated using the method shown in Figure 2. Due to the encoding mentioned in Section 2.5, the function needs to only check diagonals.

3. Testing

In order to ensure the system is working properly before beginning experimentation, testing was performed to make sure the GA converged to solutions using a small 12×12 board and the Toggle mutation operator. a known mutation operator. The system was able to converge on correct solutions to the N-Queens problem. Other mutation operators will be compared to the the known mutation operator. The initial parameters were used.

4. Experimentation

Using the N-Queens problem with a 32×32 board, the mutation rate was varied for different mutation operators to try and determine sweet spots for the operators. The results provide empirical evidence for comparing operators with each other. The mutation operators studied are included below:

- Toggle - This operator creates a random variable on the range $[0.0, 1.0]$. If the value it picks is less than the mutation rate, the operator picks two random unique indices between 0 and $N - 1$ and exchanges the permutation elements at those indices. The process is repeated N times.
- Toggle After - This operator loops over the permutation elements. At each element, the operator checks whether to mutate this element based on the mutation rate. If the operator needs to mutate the element at index J , it swaps the element with the element at index $(J + 1) \bmod N$.
- Shift - This operator creates N random numbers on the range $[0.0, 1.0]$. For each number below the mutation rate, the permutation is circularly shifted to the right once.
- Shift Half - This operator creates N random numbers on the range $[0.0, 1.0]$. If any of these numbers are below the mutation rate, it swaps the two sides of the board. Rows 0 through $\frac{N}{2} - 1$ get swapped with rows $\frac{N}{2}$ through $N - 1$.
- Invert Selection - This operator picks a random section of the chromosome ranging from size 1 to N and inverts it.

5. Results

Only the Toggle operator was able to evolve a solution. The results are summed up in the list below:

- Toggle - The global minimum of fitness evaluations occurred when the mutation rate was 0.036. On average, it required only 21,897 fitness evaluations to solve the problem. Almost any value from the range $[0.012, 0.067]$ provides a minimal number of fitness evaluations.
- Toggle After - The operator was able to find a solution in one case, which is not statistically significant.
- Shift - The operator was unable to evolve a solution in any case.

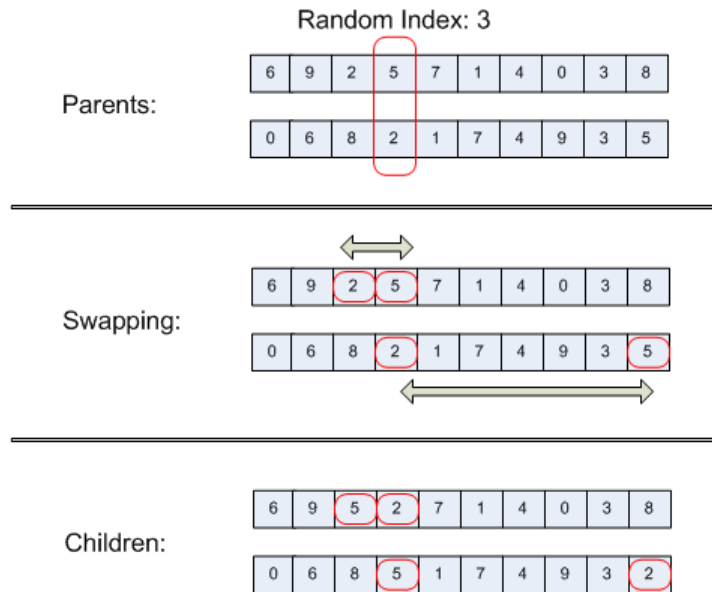


Figure 1: In PMX Crossover, a random index is selected between 0 and $N - 1$. The permutation elements at that index from both parents are recorded. Those elements are then swapped in both chromosomes individually.

```

int NQueensFitness::getFitness( PermChromosome* aChromosome ) {
    totalCalls++;
    int num = aChromosome->getLength();
    int collisions = 0;

    // Check for collisions
    // abs( deltaX ) == abs( deltaY ) -> collision
    for ( int column = 0; column < num; column++ ) {
        int row = aChromosome->getPermAt( column );
        // Check against all columns after ourselves
        for ( int columnCheck = column + 1; columnCheck < num; columnCheck++ ) {
            int rowCheck = aChromosome->getPermAt( columnCheck );

            if ( abs( row - rowCheck ) == abs( column - columnCheck ) ) {
                collisions++;
            }
        }
    }

    return collisions;
}

```

Figure 2: The code used to determine the fitness of a chromosome.

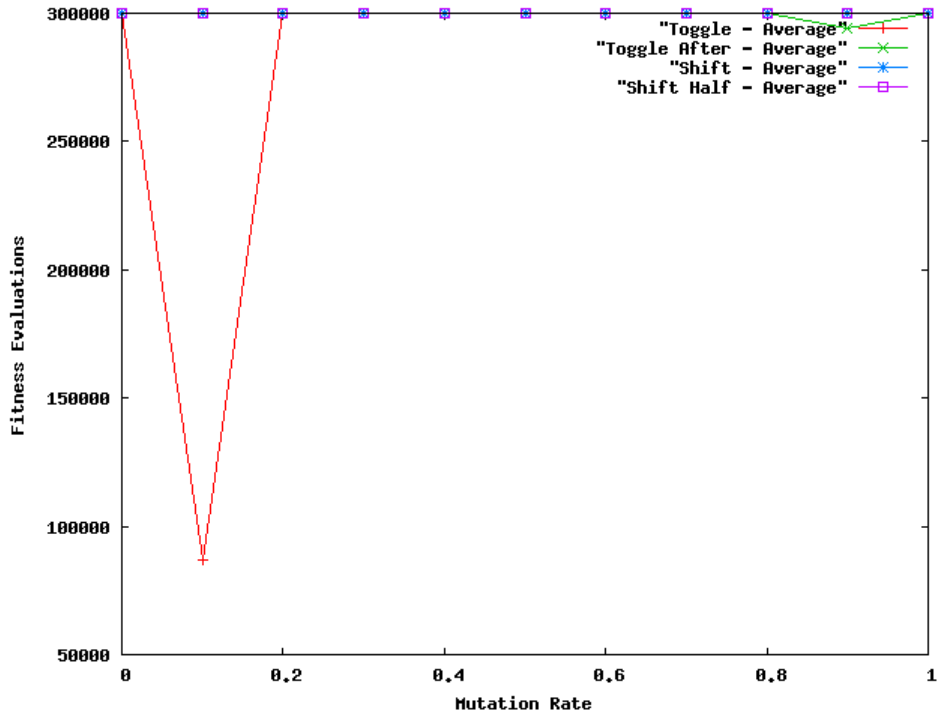


Figure 3: Mutation rates for various mutation operators were tested over the range of [0.0, 1.0] with an increment of 0.1.

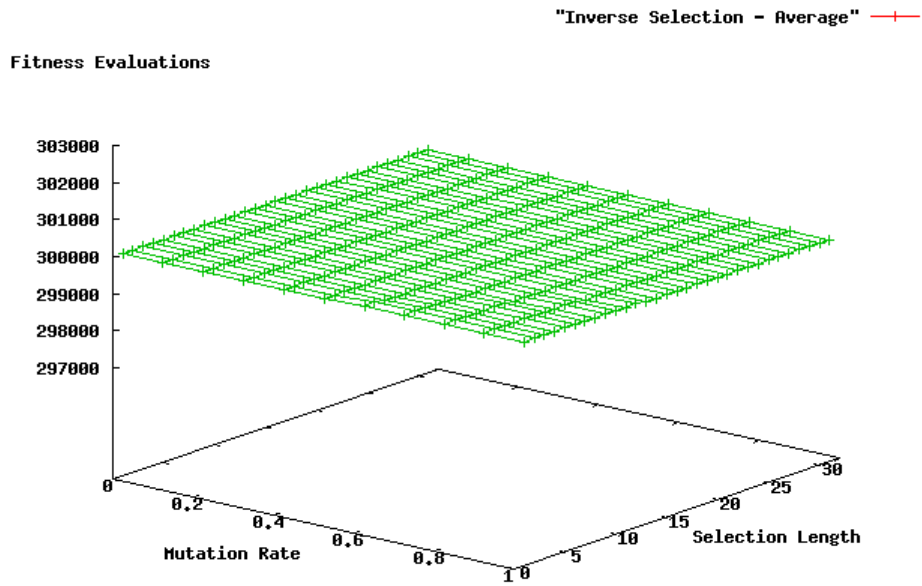


Figure 4: Mutation rate and selection size were varied when using the Invert Selection operator to determine the sweet spot for the operator.

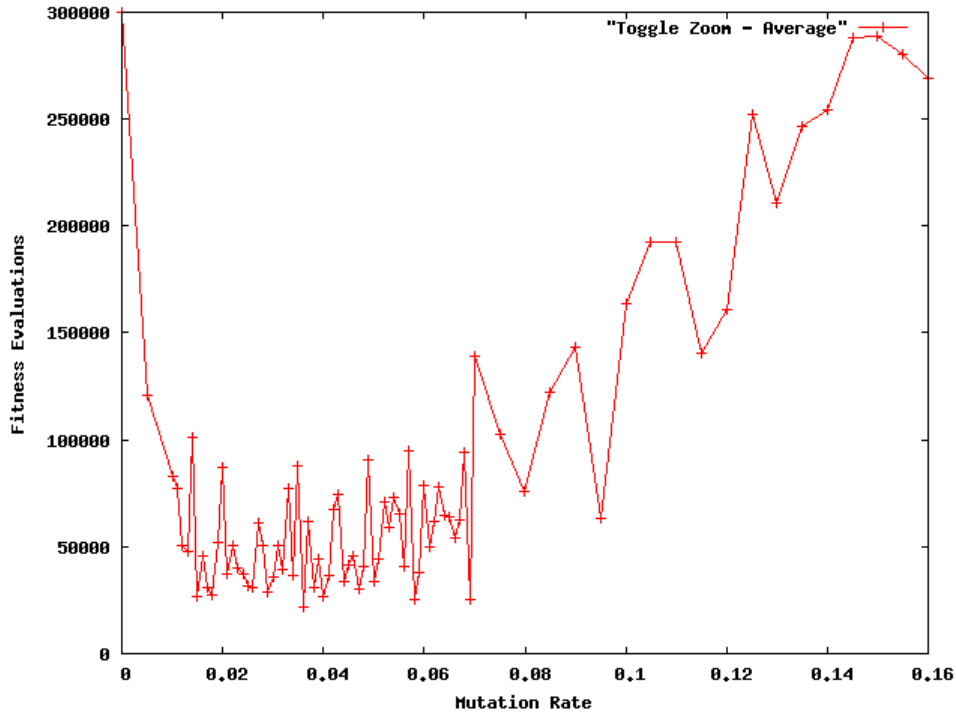


Figure 5: Mutation rates for the Toggle Method were tested over the range of $[0.00, 0.16]$ to find the sweet spot of the operator.

- Shift Half - The operator was unable to evolve a solution in any case.
- Invert Selection - The operator was unable to evolve a solution in any case.

Refer to Figure 3 for a graph of all the mutation operators over the range $[0.0, 1.0]$ except the Invert Selection operator. Refer to Figure 4 for a graph of the Invert Selection operator over the range of $[0.0, 1.0]$ while also varying the selection size over the range $[1, N]$. These graphs were used to narrow down the range to search to find each operators sweet spot. Refer to Figure 5 for a graph of the Toggle operator over the range $[0.00, 0.16]$.

6. Conclusion

Surprisingly, the Toggle mutation operator was the only method of mutation that allowed the system to evolve a solution to the problem. The Toggle operator doesn't preserve any relative positions. For this reason, the shift operator was tested. The Shift operator would preserve a large number of relative positions while still mutating the permutation. No form of the shift operator evolved a solution. The Toggle After operator, a variant of the Toggle operator, was also unable to evolve a solution. The Invert Selection operator preserves absolute positions of the elements

not picked and relative—albeit inverted—positions of the selection that was inverted. The most destructive mutation operator was the only one to correctly evolve a solution.

The goal of the paper was partially realized. Due to most operators hitting the maximum number of fitness evaluations, we have no conclusive way to compare all the operators to each other. We do have enough information to say that the Toggle operator is best, but we're not sure how much better it is. It is at least fifteen times more effective.

The sweet spot for the Toggle mutation operator was 0.036. This agrees with the generally accepted best mutation rate of 0.001 for bit string GAs. With a bit string representation thirty two bits generally represent an integer. Each integer is one value the fitness evaluation looks at. Each element in our permutation representation is equivalent to one section of thirty two bits from the bit string representation. To obtain the same mutation success for our permutation based GA, we would need to apply thirty two times the amount of mutation we applied for the bit string representation. The derived value of 0.032 is very close to the sweet spot we found for the Toggle operator of 0.036.

Future work could repeat these experiments with a finer resolution for mutation rates to prevent missing small spikes. A GA could even tune itself to evolve the proper crossover rate, mutation rate, and copy rate all at once.