CS 7770: Project 1

N-Queens Using Evolutionary Algorithms

Nolan Rink

Fall 2024

09/26/2024

Table of Contents

Introduction

The purpose of this project is to investigate the effectiveness of Evolutionary Algorithms (EAs) in solving the N-Queens problem. In the game of chess the queen is the strongest piece and is allowed to move vertically, horizontally, and diagonally across the board as shown in figure 1.1. The N-Queens problem is a combinatorial puzzle that involves placing N queens on an N×N chessboard such that no two queens are attacking each other. This problem is interesting because it is NP-Complete, which makes it easy to find solutions computationally while N is small, but as N increases the number of possible configurations increases exponentially making it computationally challenging. I will examine whether EAs can efficiently solve this problem at different board sizes and examine how different population sizes and mutation rates effect the efficiency. For the graduate requirement, I will compare the EA's performance to a traditional backtracking algorithm that solves the N-Queens problem. I will focus on examining the time to find a solution and how each algorithm scales as N increases. This analysis will help us understand the trade-offs between heuristic and exact algorithms.

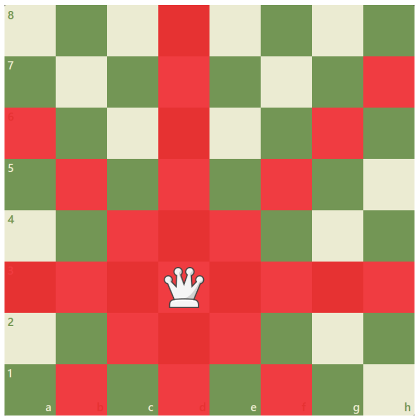


Figure 1.1 : How the queen moves

Technical Description

Evolutionary Algorithms are inspired by the idea of natural evolution and can be used to find solutions to numeric or combinatorial problems.Genetic algorithms (GA) are a subset of EAs that work by initializing a population of potential solutions encoded into chromosomes, where each member of the population is then evaluated based on the strength of the solution and strategies such as selection, crossover, and reproduction are performed to create a new population. This process repeats until a satisfactory solution is found.

For my implementation of the N-Queens problem using a GA I decided to represent each member of the population using permutation encoding. Each individual was stored in an array of size N where each index corresponds to the column in which the queen was placed on the board and the value corresponds to the row in which the queen was placed.

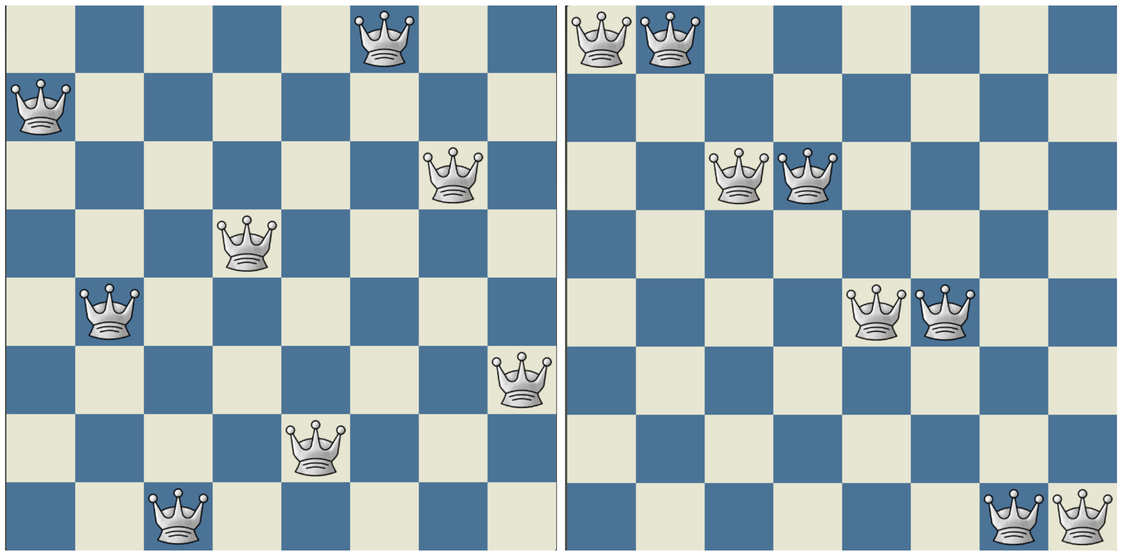


Figure 1.2 :

Left Encoding = [1,4,7,3,6,0,2,5] **Right Encoding = [0,0,2,2,5,5,7,7]**

My reasoning behind choosing this representation of the board is that, due to the nature of the problem two queens cannot be on the same row, column, or diagonal. By design this representation prevents two queens from being in the same column, and it allows for a quick check for whether or not a queen has already been placed in a certain row.

In order to evaluate each member of the population I needed a fitness function. For this particular problem each member of the population either satisfies the problem or does not satisfy the problem. To evaluate how close each member is to satisfying the problem I used a common technique for the N-Queens problem where the number of non-attacking queens is calculated. We can calculate the maximum amount of non-attacking queens using the equation $(N(N-1))/2$. If any member of the population is equal to this equation then we have found a solution. If we have not found a solution we can take the maximum amount of queens and subtract the amount of conflicting pairs of queens to produce a fitness score. The GA will then try to maximize these fitness scores.

To select individuals that will be parents for the next generation I used tournament selection. Tournament selection works by taking a subset of the population, and from this subset the individual with the best fitness score is returned. I chose this implementation of tournament selection because it is easy to implement, efficient, and because tournament selection has a good balance between exploitation and exploration.

For the crossover method I initially tried using two point crossover, but the GA performed very poorly and was unable to consistently find solutions in a reasonable amount of time. After researching better methods for crossover on the N-Queens problem I ended up choosing to use Partially Mapped Crossover (PMX). PMX is good for problems dealing with permutations because it allows crossover while still maintaining valid permutations without duplicates.

In order to keep from getting stuck in a local maximum of the search space I used mutation. For each individual I defined there to be a 5% chance that the individual would be mutated. If the individual was selected for mutation, two random indices would be selected and their values would be swapped.

Design of Algorithms

Initialization

To represent the individual permutation encoding was used. As discussed earlier the reason for this representation was so that no two queens would ever be in the same row or column. In Python the initialization for the N-Queens problem could look something like this:

```
    - population = [random.sample(range(N), n) for individual in range(size)]
```

This code just creates population of count size where each individual is randomly assigned a row and column on the board such that there is no conflicts from the row or column.

Fitness Function

Implementation of the of non-attacking queens fitness function:

Step 1: initialize N and conflict, and calculate the max of the fitness function

```
    - N = length(individual)
    - conflicts = 0
    - max = N(N-1)/2
```

Step 2: iterate through every pair of queens of the individual and find conflicts

```
    - for i from 0 to N-1
        - for j from i+1 to N-1
            - If absoluteValue(individual[i] - individual[j]) == absoluteValue(i - j)
                - conflicts += 1
```

Step 3: return the max number of pairs of queens minus the conflicts

```
    - return max - conflicts
```

Rationale: The fitness function measures the number of non-attacking pairs of queens which allows the EA to know which individuals are closer to a solution

Selection

Implementation of tournament selection:

Step 1: Calculate tournament size and select that size randomly from the population. I chose to make the tournament size 20% of the population size in order to allow the population to be diverse, but still have competitive individuals.

```
    - tournament_size = population_size / 5
    - selected = random_sample(population, tournament_size)
```

Step 2: Sort the sample population and return the best individual

```
    - best = individual with highest score
    - return best
```

Rationale: Tournament selection balances exploitation and diversity. Adjusting the tournament size to 20% of the population allows for a competitive yet diverse selection.

Crossover

Implementation of Partially Mapped Crossover (I needed help from chat GPT to get this implemented correctly)

<u>Step 1:</u> Create a copy of each parent

```
- size = len(parent1)
- child1 = copy(parent1)
- child2 = copy(parent2)
```

<u>Step 2:</u> Select crossover points

```
- cx_point1 = random_num(0, size-2)
- cx_point2 = random_num(cx_point1 + 1, size - 1)
```

<u>Step 3:</u> Exchange the segments and fix conflicts

```
- FOR i FROM cx_point1 TO cx_point2:
    - val1 = child1[i]
    - val2 = child2[i]
    - idx1 = index_of(val2 IN child1)
    - idx2 = index_of(val1 IN child2)
    - swap(child1[i], child1[idx1])
    - swap(child2[i], child2[idx2])
```

Rationale: PMX is suitable for permutation problems as it preserves the sequence and position of elements, maintaining valid permutations without duplicates.

Mutation

Implementation of the mutation function

<u>Step 1:</u> Determine whether or not to mutate

```
- If random_float < mutation_rate
    - swap(individual[random_index1], individual [random_index2]
- return individual
```

Rationale: this type of mutation introduces diversity while maintaining valid permutations by swapping two positions, avoiding duplicate values.

Genetic Algorithm

Implementation of the Genetic Algorithm

Step 1: Initialize Population

Step 2: Evaluate the fitness for each individual

Step 3: Check for a solution

Step 4: Select parents using tournament selection

Step 5: Apply Crossover to produce offspring

Step 6: Apply mutation to offspring

Step 7: Set population to offspring

Step 8: Loop

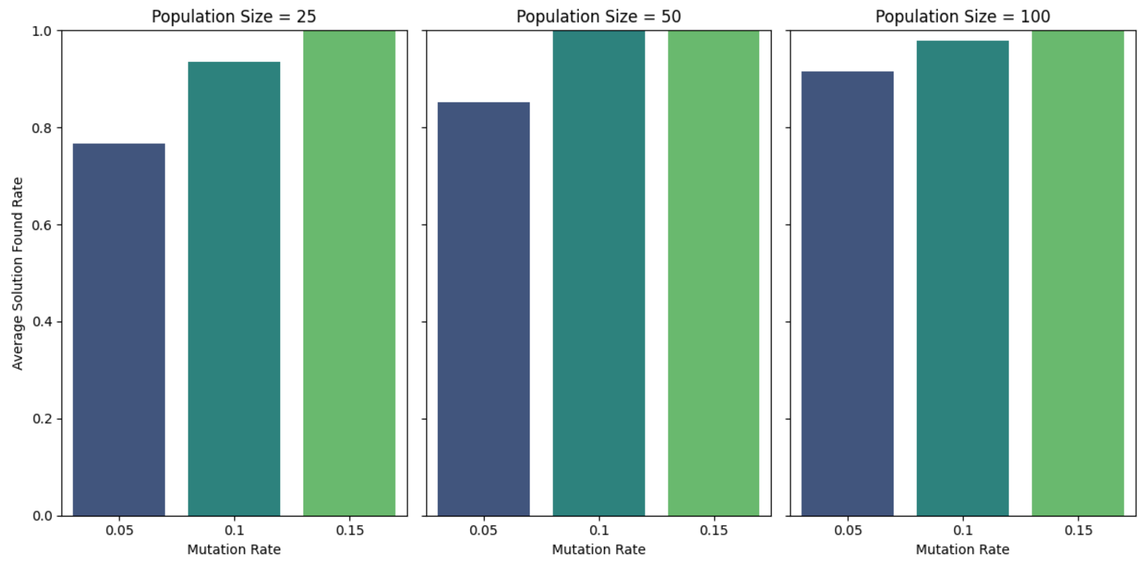
Experiment Description

The experiments conducted involved applying a genetic algorithm to solve the N-Queens problem for board sizes ranging from 4 to 40. For each board size, the algorithm was tested using different combinations of mutation rates (0.05, 0.1, and 0.15) and population sizes (25, 50, and 100). Each experiment ran the algorithm for a max of 5000 generations, recording whether a valid solution was found, the time taken, the best fitness score achieved, and the corresponding queen positions.

These experiments were performed to evaluate the effectiveness of the genetic algorithm in solving the N-Queens problem under varying conditions. By altering the mutation rates and population sizes, the study aimed to understand how these parameters impact the algorithm's ability to find valid solutions, its efficiency, and its scalability with increasing board sizes. The collected data enables analysis of optimal parameter settings and provides insights into the algorithm's performance trends, helping to inform improvements and assess its applicability to larger or more complex instances of the problem.

Experiment Analysis

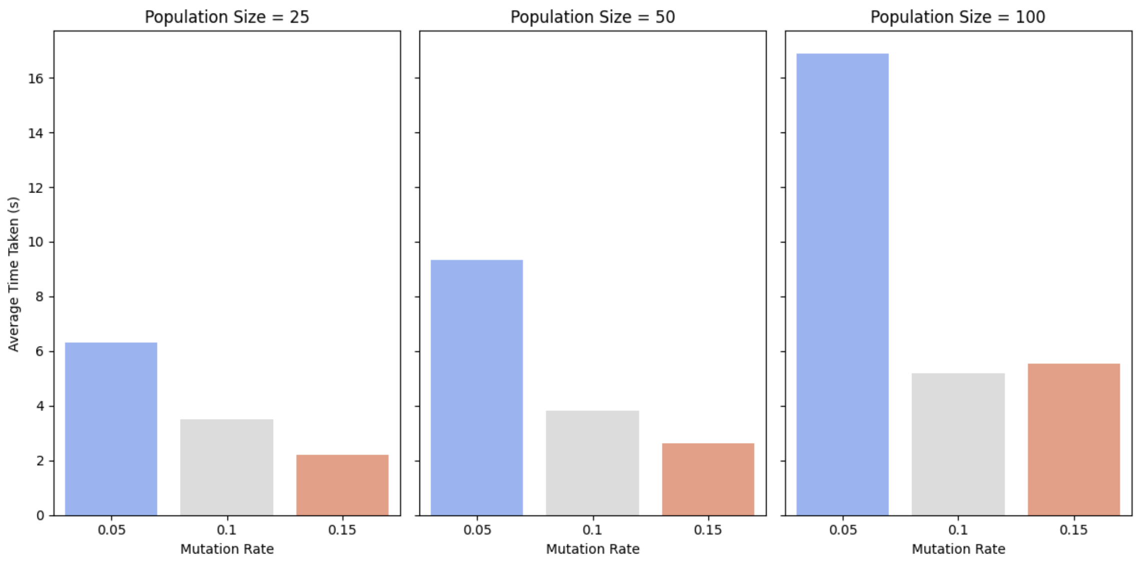
First lets analyze the effects of mutation rates and population size on the GAs ability to find a solution.



Observations:

- For smaller population sizes, a higher mutation rate leads to a better solution found rate. Lower mutation rates tend to perform worse, leading to incomplete search and solution discovery.
- As the population size increases, all mutation rates result in consistently high solution found rates, with the 0.15 mutation rate still showing a slight advantage. This suggests that population size plays a critical role in the rate at which the GA can find solutions.
- Overall, a higher mutation rate tends to ensure that the genetic algorithm doesn’t get stuck in local optima, leading to a more consistent ability to find solutions across different population sizes.

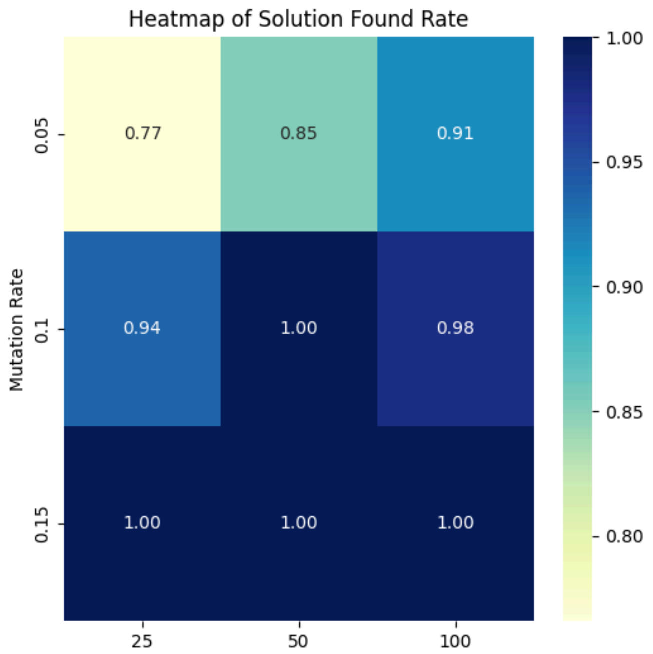
Now lets analyze the effects of mutation rates and population size on the GAs time to fins a solution.

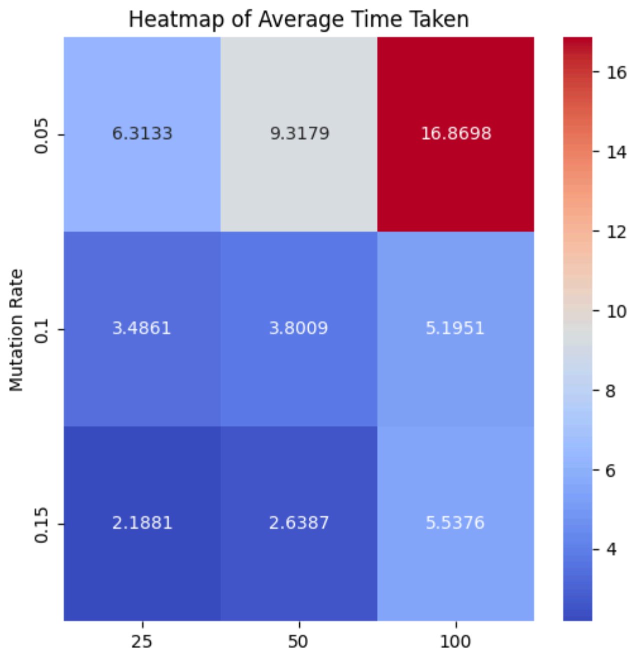


Observations:

- The mutation rate of 0.05 takes noticeably longer, particularly for larger population sizes. This indicates that a low mutation rate may lead to slower convergence and reinforces my thinking that a low mutation rate leads to getting stuck at a local optima more frequently.
- The mutation rate of 0.15 shows the lowest average time across population sizes, especially for smaller boards, demonstrating faster convergence due to more diverse exploration of the solution space.
- The Medium mutation rates seem to provide a middle ground, balancing exploration and time efficiency but don’t always outperform the higher mutation rates.

The data from both the previous bar charts can also be shown as heartmaps.

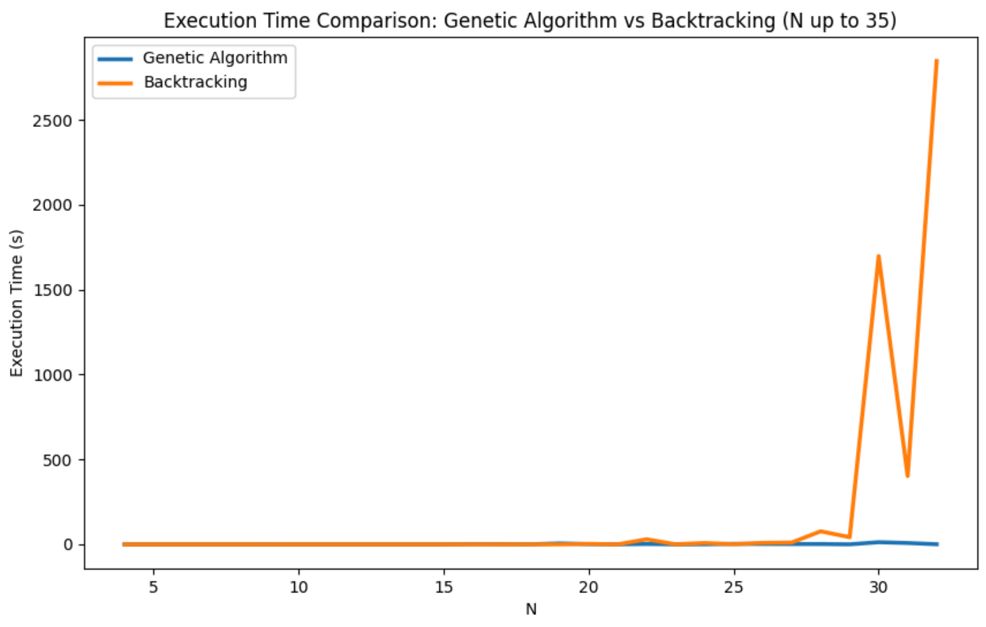




Observations:

- Again we see that a higher mutation rate leads to a higher success rate in finding solutions.
- Low mutation rates perform very poorly in both time to find a solution and ability to find a solution compared to higher mutation rates.
- As we also saw earlier the larger population sizes found a solution more consistently but took more time on average to find the solutions. This is likely because operations such as mutation and crossover had to be performed more times making the GA explore the solution space slower.

Now for the graduate requirement I will analyze compare the solution speed of the GA and a traditional backtracking approach.



Observation:

- The execution time for backtracking remains minimal and closely aligned with the GA for small board sizes (N ≤ 25). Both algorithms exhibit similar, very low execution times and can't really be differentiated
- For board sizes beyond N = 25, the execution time for backtracking increases drastically, with a significant spike seen around N = 30 to N = 35, where the execution time rises sharply to over 2500 seconds.
- The genetic algorithm (GA) maintains a consistently low execution time, even as the board size increases. There is only a slight increase in execution time for GA compared to the extreme spike observed in backtracking, indicating a much more stable performance across different values of N.

Conclusions on backtracking vs GA:

- Backtracking scales poorly with increasing board size, particularly for N ≥ 30, where it becomes highly inefficient in terms of time complexity. The sharp increase in execution time suggests that the backtracking approach struggles to manage the exponential growth in the search space as N grows, making it impractical for solving larger instances of the N-Queens problem.
- The GA demonstrates superior scalability, showing minimal changes in execution time even as N increases. This indicates that the GA is more resilient to larger board sizes due to its ability to search the solution space more efficiently using probabilistic methods, rather than exhaustively exploring all possibilities like backtracking.
- For larger board sizes (N > 30), the genetic algorithm becomes the more practical choice due to its ability to maintain low execution times, while backtracking becomes infeasible due to the rapid increase in execution time. This highlights the advantage of heuristic-based approaches like GA in solving complex, large-scale combinatorial problems like the N-Queens problem.

Conclusion

The optimal configuration for solving the N-Queens problem using a genetic algorithm appears to involve higher mutation rate that is probably slightly lower than 0.15 combined with medium to large population sizes. This configuration provides the best balance of exploration and exploitation, enabling the algorithm to efficiently converge to valid solutions without becoming stuck in local optima. Larger population sizes contribute to greater genetic diversity, and higher mutation rates ensure that the algorithm explores new areas of the solution space, which is especially crucial for larger board sizes ($N \geq 30$).

For smaller board sizes, the differences between mutation rates and population sizes are less pronounced, but as the problem size increases, the genetic algorithm's performance becomes more sensitive to these parameters. Hence, tuning mutation rates and population sizes is critical for ensuring that the algorithm scales effectively to solve larger instances of the N-Queens problem.

Citations

https://research.ijcaonline.org/volume102/number7/pxc3898667.pdf
Thada, Vikas & Dhaka, Shivali. (2014). Performance Analysis of N-Queen Problem using Backtracking and Genetic Algorithm Techniques. International Journal of Computer Applications. 102. 26-29. 10.5120/17829-8667.