# Group 8 Assignment Report

**An Evolutionary Algorithm for the N-Queens problem**

Aswathy Gopalakrishnan Yonge Li
Cheng Yang Yi Yang Tingjun Yuan

October 3, 2024

## 1 Introduction

Evolutionary Algorithm is an optimization algorithm which is inspired by Charles Darwin's theory of biological evolution and natural selection. According to this theory, species evolve over time through the process, of "Survival of the fittest". This means, the ones with better fitness have a higher chance of survival and hence, reproducing a fitter generation. In an evolutionary algorithm, this principle is converted to a computational framework for optimization problems. It mimics the same evolutionary processes such as reproduction, mutation, recombination, and selection. The process continues across generations to refine solutions until a desired criteria is met. This method is well suited for solving problems in large, combinatorial search spaces.[1].

Evolutionary Algorithms are classified under heuristic algorithms. They are designed to find good enough solutions to complex problems in less time compared to traditional approaches. They do not guarantee to find the optimal solution. However, they aim to find the best solution within a reasonable timeframe.

In this assignment, our goal is to explore the N-Queen problem using Genetic Algorithm. The N-Queen problem based on this evolutionary algorithm has become a widespread platform for AI researchers. The desired solution to this problem is to place N queens on an N x N board in such a way that no queens attack each other[2]. N-queen is a permutation problem. Each solution can be thought of as a permutation of the values from 1 to N, without any duplicates. Considering the solution space is vast and complex, the evolutionary approach is a good choice to solve the N-Queen problem as it has a variety of techniques suitable for the same.

## 2 N-Queens Problem

The N-Queen problem came into existence after the generalization of the 8-Queen problem. The problem was introduced in 1848 by a German chess player, Max Bazze. In 1850, Franz Nauck gave the first solution to this problem and generalized the problem as the N-Queen problem with N not attacking queens on an N x N chess board.[3].

### 2.1 Inspiring Academic works

During the last 3 decades, the problem has served as a benchmark for the exploration of genetic algorithms. One of the first attempts in using the genetic algorithm to solve the N-Queens problem was by Turner and his colleagues to solve the limitation of memory for a large number of N.[4] Ismail A Humied has explored the problem using subproblems based on genetic algorithms. This has been an inspiration in our algorithm to explore various crossover and mutation operators. Xiao Hui Hu and his team explored a method called swam intelligence for permutation optimization using N-Queens.[5] However, unlike GA, this has no evolution operators such as crossover and mutation. Sachin Sharma and Vinod Jain have done a study on a novel mutation operator for the N-Queen problem which resulted in a better performance of the algorithm. This is an interesting concept where the queen with the maximum number of conflicts is mutated such that it has minimum conflicts, thereby creating a solution closer to the desired one. They call the operator as greedy mutation operator.[3]

### 2.2 Motivation behind the Algorithm

Evolutionary Algorithms offer a rich variety of techniques that can be used to explore complex optimization techniques, like the N-Queen problem. In the algorithm we have developed, multiple techniques are explored. By this, it is possible to systematically evaluate their performance and the quality of the generated outputs. The comparative analysis allows a deep understanding of the strengths and weaknesses of each approach, thereby facilitating in identification of the most effective approach. Engaging in the exploration enhances the understanding of all the mechanisms of evolutionary algorithms. This comprehensive evaluation strategy to identify optimal strategies is the motivation behind the algorithm developed.

## 2.3 Constraints and Mathematical formulas

N- Queen is a permutation problem. A permutation problem is a constraint satisfaction problem with the same number of variables as values and each variable has a unique value. When the constraints are satisfied, we can call it a feasible solution. There can be more than one feasible solution.

N-Queen involves locating N queens such that there are no conflicts. Hence, the problem has 3 constraints:

- 1. No queens can be in the same row.
- 2. No queens can be in the same column.
- 3. No queen can be on the same diagonal.

Since each queen must be placed on a different row and column, we can assume that the Queen Qi is placed in the ith column. All the solutions can be represented as an n-tuple (Q1, Q2, .....Qn) that are permutations of n-tuple (1, 2, ....., N). The position of the value represents the column in which the queen is placed and the value represents the row.[6].

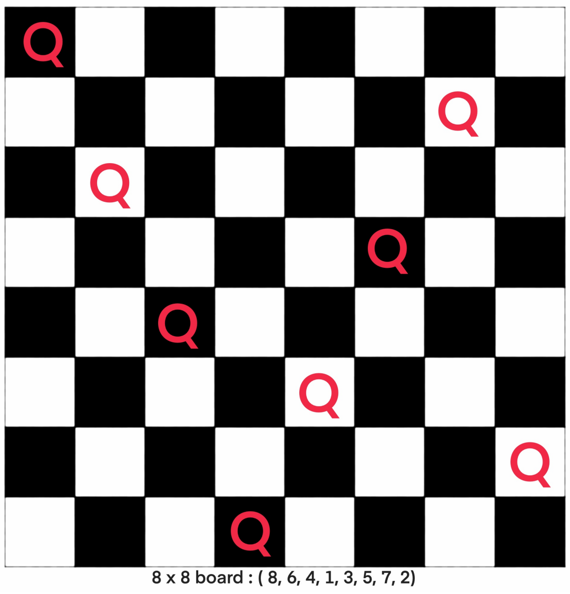


Figure 1: 8-queen problem

The mathematical representation of the three constraints can be written as below:

$$A = \{(Q_1, Q_2, \ldots, Q_n) \mid Q_i \in \{1, 2, \ldots, N\}\}$$

Where A is the sample space and Qi is the ith cell corresponding to the ith column of the chessboard.[6].

$$\forall i, j \in \{1, 2, \ldots, N\},\ Q_i \neq Q_j$$

This means for all $i$ and $j$ in the set $\{1, 2, \ldots, N\}$, where $N$ is the total number of queens or objects. $Q_i \neq Q_j$ means that the positions (or values) of the $i$th and $j$th queens must not be equal.[6].

$$\forall i, j \in \{1, 2, \ldots, N\},\ |Q_i - Q_j| \neq |i - j|$$

The diagonal conflict occurs if the difference in row indices $|i - j|$ equals the difference in column indices $|Q_i - Q_j|$. To prevent a diagonal attack, the condition $|Q_i - Q_j| \neq |i - j|$ must hold.[6].

## 2.4 Fitness Function

A good fitness function determines how close a candidate solution is to being a valid N-Queen solution. For N-Queens, the fitness is determined by the number of non-attacking pairs of queens. When you place a queen on an $N \times N$ board, there are multiple ways for two queens to attack each other. The number of possible pairs of queens that can attack each other can be calculated by the combinatorial formula $\binom{n}{2}$. This can be simplified as,

$$\textbf{Maximum number of non-attacking pairs} = \frac{N(N-1)}{2}$$

For instance, if N=8, the maximum number of non-attacking pairs is 28. This means, there are 28 possible ways for queens to attack each other.

The fitness of a solution is based on minimizing the number of attacks. Hence, the maximum fitness value can be considered as N(N-1)/2 which is the total number of non-attacking pairs. Now, if we consider diagonal conflicts, we can find the non-conflicting pairs as,

$$\textbf{Fitness} = \textbf{maximum No: of non-attacking pairs} - \textbf{No: of diagonal conflicts}$$

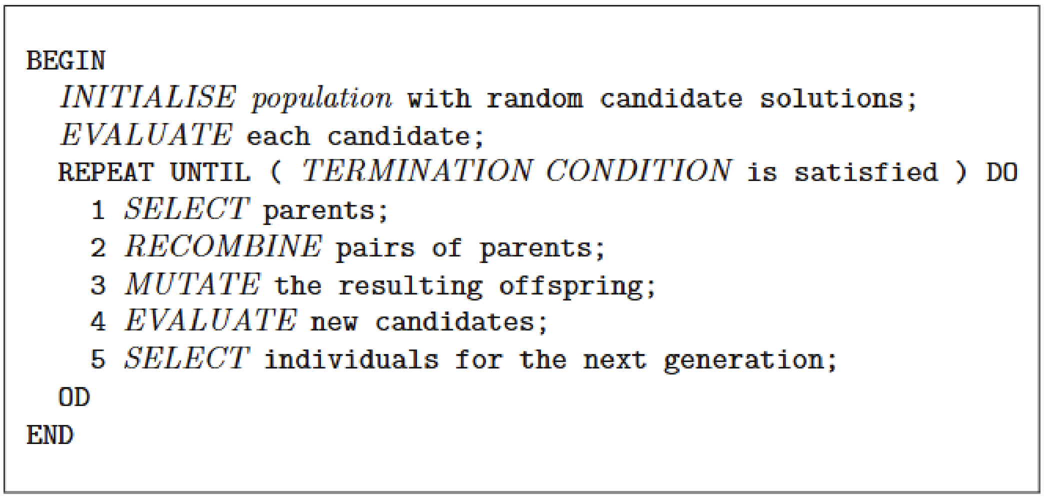
If there are no conflicts, then,

**Maximum Fitness = maximum number of non-attacking pairs**

## 3 Algorithm

The Algorithm developed is an adaptation of various techniques of a Genetic Algorithm to solve the N-Queens problem. The goal is to place N queens on an N x N chessboard without any queens attacking each other. All the techniques used are for permutational optimization. In the algorithm, multiple crossover and mutation operators are implemented, tested and compared to get the desired solution. This combined approach offers flexibility and modularity to test different techniques dynamically.

Algorithm Components



```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Figure 2: The general scheme of an evolutionary algorithm in pseudocode

### 3.1 Initialization

The initialization is the first process of the genetic algorithm where the initial population of possible solutions are generated. For permutation representation, it is not possible to consider each gene individually. Hence, we have initialized the population following a permutation representation of length N thereby avoiding duplicates. This ensures that all the queens are in the different columns. This method is called **Random Permutation**[?]**.**

**Algorithm 1** pseudocode for initialization

```
  $Input : population\_size, N$
  $Output : population$
     $population = []$                    ▷ initialize the population with an empty list

  BEGIN
       set population = [ ]
  for each individual i in range of population_size do
             Generate a randomly permutation of numbers 1 to N
             Append the generated permutation to the population
  end for
  Return population
```

## 3.2 Fitness Function

As discussed in the mathematical formula section, the fitness function determines how close a solution is to the optimal one. For the N-Queens problem, the maximum fitness is when there are no conflicting pairs of queens. The fewer the conflicts, the better the fitness. As we have initialized the population in such a way that no column conflicts exist, the fitness function is designed to minimize the row and diagonal conflicts.

Hence, the formula for fitness function = n(n-1)/2 -diagonal conflicts

## 3.3 Selection Operator

In situations where the population size is very large, obtaining knowledge of the probability distribution of the entire population can be hard. In these cases, **Tournament Selectio**n is a useful operator. It does not require any global knowledge of the population. The reason that it does not rely solely on fitness proportion makes it an ideal choice for N-Queens problem over the Roulette wheel or rank-based selection methods.

In Tournament selection, a subset of individuals is selected randomly and the best one in that subset is chosen as the parent. This method gives a balance between choosing fit individuals and random exploration. The concept is simple and easy to implement. Also, the probability that any individual can be selected during the selection improves the diversity and avoids premature convergence. The selection pressure is also easy to control by varying the tournament size.[7].

**Algorithm 2** pseudocode for fitness function

```
Input: candidate (list of queen positions)
Output: fitness score
BEGIN
    Set n = length of candidate solution
    Calculate maximum non-attacking pairs using the formula ( n * (n - 1) / 2 )
    $diagonal\_conflicts = 0$
for $i = 0$ to $n - 1$ do

    for $j = i + 1$ to $n - 1$ do

        if $|candidate[i] - candidate[j]| = |i - j|$ then
            diagonal_conflicts ← diagonal_conflicts + 1
        end if
    end for
end for
    fitness ← maximum_pairs - diagonal_conflicts
Return population
```

**Algorithm 3** pseudocode for selection

```
Input: population (list of candidates), tournament_size (integer)
Output:selected_candidate (the candidate with the highest fitness)
BEGIN
    Randomly select tournament_size candidates from the population
    tournament ← random.sample(population, tournament_size)
    $max\_fitness = 0$
    $selected\_candidate = []$
for each individual $i$ in tournament do
        fitness ← fitness_function(tournament[i])

    if $fitness > max\_fitness$ then
            max_fitness ← fitness
            selected_candidate ← tournament[i]
    end if
end for
Return selected_candidate
```

## 3.4 Crossover

In permutation-based representation, it is not possible to simply exchange substrings between parents as it creates duplicates and breaks the permutation property. The ideal crossover operators for these are partially mapped, edge, order and cycle crossover operators. In this algorithm, we have explored partially mapped, order and cycle operators to understand their performance and its effect on the algorithm. The crossover type can be set by giving the strings ‘pmx’, ‘ox’ or ‘cx as user input at the runtime.

### 3.4.1 Partially mapped crossover

Partially mapped crossover (PMX) is one of the most widely used operators for adjacent type problems. It works by swapping sections of the parents. Then, to ensure that valid permutations are maintained, it maps corresponding sections from the parent to the child while respecting the permutation constraint.

### 3.4.2 Order Crossover

In order crossover (**OX**), a segment is taken from one parent and the remaining elements are filled in order, respecting the sequence of the other parent. This ensures that the order of the solution is preserved. In the process, OX starts similar to PMX. However, it proceeds differently as the aim is to transfer genes in a relative order from the second parent.

### 3.4.3 Cycle crossover

Cycle crossover (**CX**) identifies cycles of corresponding elements between the two parents and swaps them to form the child. It preserves the order and positions of genes. But, during the exploration, we could find that in cases where the cycle goes all the way including all elements except the cases with the same values in the same columns in parents, the crossover reproduces to form children exactly the same as parents.[7].

**Algorithm 4** pseudocode for crossover(PMX/OX/CX)

```
Input:parent1, parent2, Type of crossover (PMX/OX/CX, crossover rate)
Output:child1, child2
BEGIN
      Set N = length of parent
if crossover rate<=0.9 then
          Initialize child1 and child2 as lists of length $N$ with None values
    if crossover_type == "pmx" OR crossover_type == "ox" then
        Select two random crossover points, point1 and point2 such that:
        $1 \leq$ point1 $<$ point2 $\leq N - 1$
        Interchange the segment between point1 and point2 between parents
        if crossover_type == "pmx" then
            for $i = 0$ to $N - 1$ do
                if $i <$ point1 OR $i >$ point2 then
                    Set the parent value as a temporary variable temp
                    while temp is in the copied segment of child do
                        Find the index of temp in the copied segment
                        Set temp = parent[mapped_index] for child
                    end while
                    Set child's gene as temp
                end if
            end for
            return child
        end if
        if crossover_type == "ox" then
            Set fill_point as the next gene after point2 [(point2 + 1)%$N$]
            for each gene in parent do
                if gene is not already in child then
                    Place the gene at child[fill_point]
                    Increment fill_point to the next position (cycling if necessary)
                end if
            end for
            return child
        end if
    else if crossover_type == "cx" then
        Initialize cycle_values as an empty list
        Set starting point as 0 and point current_point to this
        while current_point is not in cycle_values do
            Add current_point to cycle_values
            Set current_value = parent1[current_point]
            Set current_point = index of current_value in parent2
        end while
        for each $i$ in cycle_values do
            Copy parent to child
        end for
        for $i = 0$ to $N - 1$ do
            if child[i] is None then
                Copy parent[i] to child
            end if
        end for
        return child
    end if
```

## 3.5 Mutation

We have to consider the permutation factor when it comes to mutation operators too. In this Algorithm, Swap and Inversion mutation operators are used which are ideal for maintaining the permutation representation and thereby avoiding duplication. The mutation type can be set by giving the strings 'swap' or 'inversion' as user input at the runtime.

### 3.5.1 Swap Mutation

Swap mutation operator selects two random genes and swaps their values. This ensures the permutation property is maintained while exploring new values.

### 3.5.2 Inversion Mutation

In Inversion mutation, a subsequence of the chromosome is selected and reversed. This mutation is particularly introduced into the algorithm to ensure more diversity to the children reproduced through cycle crossover.

**Algorithm 5** Pseudocode for Mutation

```
Input: child, mutation_rate, mutation_type
Output: mutated_child

BEGIN
if randomly generated value < mutation_rate then
    if mutation_type == "swap" then
        Randomly select two indices $i$ and $j$
        Swap the values of child[$i$] and child[$j$]
    else if mutation_type == "inversion" then
        Set child as mutated_child
        Randomly select start and end positions where start < end
        Reverse the segment of mutated_child from start to end points
    end if
end if
Return mutated_child
END
```

## 3.6 Replacement

The method used for replacement in this algorithm is **Generational Replacement**. In this, the entire population is replaced by a new generation without preserving any individuals from the previous generation. This method encourages more exploration of the space, as every new generation introduces a completely new set of candidate solutions. Since no elite individuals are preserved, there is a risk of losing highly fit individuals.

## 3.7 Termination

If a candidate in the population reaches the maximum fitness level before reaching the generation limit, the algorithm terminates early and returns the optimal solution. If not, the algorithm works until the generation limit is met and returns the best solution found so far.

## 3.8 User Input function

The values of inputs to the main genetic algorithm keep changing based on value of N and desired crossover and mutation operators. For the same reasons, it's ideal for the user to input the function at the runtime. To help this a get_user_input function has been written which takes inputs given and converts them into required datatypes before passing them to the main genetic algorithm function.

## 3.9 Text output function

The final output is written into a text file listing all the necessary values and the final best solution.

A sample of the expected output is:

```
N = 8
Population size = 50
Generations = 100
Tournament size = 3
Mutation rate = 0.05
Mutation type = swap
Type of crossover = pmx
Solution : [7, 3, 8, 2, 5, 1, 6, 4]
```

**Algorithm 6** Genetic Algorithm for N-Queens Problem

```
Input: $N$, population_size, generations, tournament_size, crossover_type,
crossover_rate,mutation_rate, mutation_type
Output: best_candidate (the solution)

BEGIN
Initialize population = population_initialization(population_size, N)
for each generation do
    Initialize new_population = []
    for $i = 0$ to $(population\_size//2) - 1$ do
        Select two parents using tournament_selection
        Perform crossover between parents using the crossover_type
        Mutate child using mutation_rate and mutation_type
        Add children to new_population
    end for
    Replace population with new_population
    Find the best_candidate in the current generation using fitness function
    if best_candidate has maximum fitness then
        Print the solution
        return best_candidate
    end if
end for
if no solution is found after all generations then
    Output the best_candidate in the final generation
    return best_candidate
end if
END
```

**Algorithm 7** Pseudocode for user input function

```
Input: $N$ (number of Queens), population_size, generations, tournament_size,
crossover_type, crossover_rate,mutation_rate, mutation_type
BEGIN
Prompt user to input $N$, population_size, generations, tournament_size,
crossover_type, mutation_rate, and mutation_type
Typecast each of the inputs into respective variables
Return the inputs
END
```

**Algorithm 8** Pseudocode for user output function

```
Output: $N$ (number of Queens), population_size, generations, tournament_size,
crossover_type, crossover_rate, mutation_rate, mutation_type
BEGIN
Open the file "final solution.txt" in write mode
Write the required information and final solution to the file
Close the file
END
```

In conclusion, the genetic algorithm developed is a combination of all the techniques mentioned above. It practices the key evolutionary operations such as tournament selection, crossover, and mutation to efficiently explore the solution space. Also, the comparison of multiple techniques allows us to understand their efficiency and quality. The algorithm's iterative nature combined with generational replacement ensures a dynamic search process. This leads to efficiently finding optimal or near-optimal solutions.

## 4 Experimental part

### 4.1 Hardware and Software details

For the experimental set up, the following hardware and software configurations are used.

Hardware

- **System:** LENOVO IdeaPad Gaming 3 15ACH6
- **Processor:** AMD Ryzen 5 5600H with Radeon Graphics,NVIDIA GeForce RTX 3050
- **RAM:** 16 GB
- **Operating System:** Microsoft Windows 11

Software

- **Programming Environment:** Python 3.12
- **IDE:** Visual Studio Code (VSC)
- **Key Libraries:** random for stochastic operations, and time for measuring performance

## 4.2 Experimental Procedure

The experiments on the developed algorithm focused on exploring and evaluating various crossover techniques and their impact on the performance of the genetic algorithm to solve the N-Queen problem. Below is a structured overview of the steps followed along the development of the algorithm.

1. **Understand the problem**
   The knowledge about the problem was limited in the beginning. The initial was used in understanding the various processes involved in Genetic Algorithms. Also, identified the required input data, constraints, and expected output for the N-Queen problem.

2. **Initial set up**
   During the initial phase of the development, the algorithm was handled process by process. The initialization, fitness function and tournament selection were the first parts coded.

3. **Choosing crossover types**
   To understand the technique, single point crossover was initially used. However, later on it was observed that this is not apt for permutation problems. The duplicate values had to be replaced by the missing genes in the candidate using the pop method. This diverted the algorithm to explore more about the permutation-related crossover operators. The partially mapped crossover was the first one explored. PMX preserves the relative order of genes and maintains a valid permutation. Hence, developed the PMX function, ensuring that it efficiently handled duplicates.

4. **Completing the Genetic Algorithm**
   Incorporated Swap Mutation: After establishing PMX, it was integrated with the swap mutation operator which is again ideal for permutation problems.Generational Replacement: Implemented a generational replacement strategy where the entire population is replaced with new offspring after each generation.

5. **Exploration of Parameter values**
   Conducted experiments with different population sizes, generation counts, and mutation factors across various values of N. Started with small values and then increased them if the solution is not found within the expected time. This experiment helped in understanding how the values must be changed with the value of N and how it affects the performance of the algorithm.

6. **Introduction of more crossovers**
   The order crossover was introduced first. This is another permutation-related

crossover operator. OX maintains the order of genes while filling in the missing values. This offered a different mechanism for exploring crossover. Exploration of parameter values was done with OX crossover as well. The cycle crossover, yet another permutation supportive crossover mechanism, was introduced next. The same process was followed here while experimenting with various values. It was observed that some parents, when the cycle goes all the way, produces children same as the parents involved in the process.

7. **Incorporation of Inversion mutation**
   While exploring more mutation operators, an experiment was done using inversion operators to introduce more diversity in the children generated through CX.

8. **Text file Output**
   The assignment requires a text output file to represent the expected output representation. This was implemented using a write to file function which outs a text file with the input parameters and output solution.

9. **Performance measurement**
   A timing function was introduced to evaluate the time to run the algorithm when various crossover and mutation operators are used. This allowed for a direct comparison between the crossover and mutation operators. Experiments were conducted by mixing various crossovers with different mutation operators to asses their collective impact on the algorithm.

**Conclusion of experimental set up** This systematic approach provided valuable insights into how different configurations affect the algorithm and the out of the N-Queen problem.

## 4.3 Data used for the experiment

### Data collection method

Data was collected during the runtime of the program. The parameters such as the value of **N, population size, number of generations, tournament size, crossover type, mutation factor, and mutation type**are provided as user input.

**N, the number of Queens**
N is the primary variable determining the size of the chessboard and the complexity of the problem. Different values of N were tested starting from N=4 to N=64 to explore various complexities of the algorithm.

**Population size**
The population size determines how many candidate solutions are evaluated per generation. A larger population size provides more diversity. According to the value of N, the population size provided also changed.
For small values of N (N<=20): population size is 50 to 100
For medium values of N(20<=N<=50): population size is 100 to 200
For large values of N(N>50): population size is 200-500

**Data collection method**
The number of generations decides how long the GA will run. More generations allow the convergence toward an optimal solution.
For small values of N (N<=20): Number of generations is 500 to 1000
For medium values of N(20<=N<=50): Number of generations is 1000 to 2000
For large values of N(N>50): Number of generations is 2000-5000

**Tournament size**
This Parameter was varied to test its effect on the selection pressure. For small to medium values of N, 3 to 5 is used and for larger values of N 5 to 10 can be used.

**Crossover operators**
The values pmx, ox, or cx are given as input to choose the desired crossover method.Crossover rate is given as 0.9 which means 90 per cent of the population undergoes crossover.

**Mutation factor**
The mutation factor controls how often random changes are introduced to a candidate solution.
For small N: A mutation probability of 0.01 to 0.05 is used
For medium and larger values of N: A higher mutation probability of 0.05- 0.1 is used

**Mutation type**
The values swap or inversion are provided as input to choose the desired mutation type

# 5 Results and Analysis

In the analysis, the performance of different crossover and mutation operators in solving the n-queens problem is evaluated. Experiments were run by varying configurations of population size, number of generations, tournament size, and mutation rates for different values of N. The goal is to assess execution time, optimality of solution, and generation at which the convergence happens.

## 5.1 Basic Analysis - Fixed Parameters

The values for the input parameters are kept fixed for all values of N. The values are set as Population = 50, Generations = 500, Tournament size = 3, Crossover type = pmx, Mutation factor = 0.05, Mutation = Swap.

Table 1: Results of Fixed-Parameter Analysis

| Value of N | Runtime | Optimal | Generation of solution |
|---|---|---|---|
| 4 | 0.002044 s | yes | 0 |
| 8 | 0.002568 s | yes | 0 |
| 12 | 0.491578 s | yes | 351 |
| 16 | 1.011509 s | yes | 411 |
| 20 | 1.76977 s | yes | 180 |
| 24 | 2.454198 s | yes | 316 |
| 28 | 3.41216 s | no | 500 |
| 32 | 4.04993 s | no | 500 |
| 36 | 5.42044 s | no | 500 |
| 40 | 6.65575 s | no | 500 |

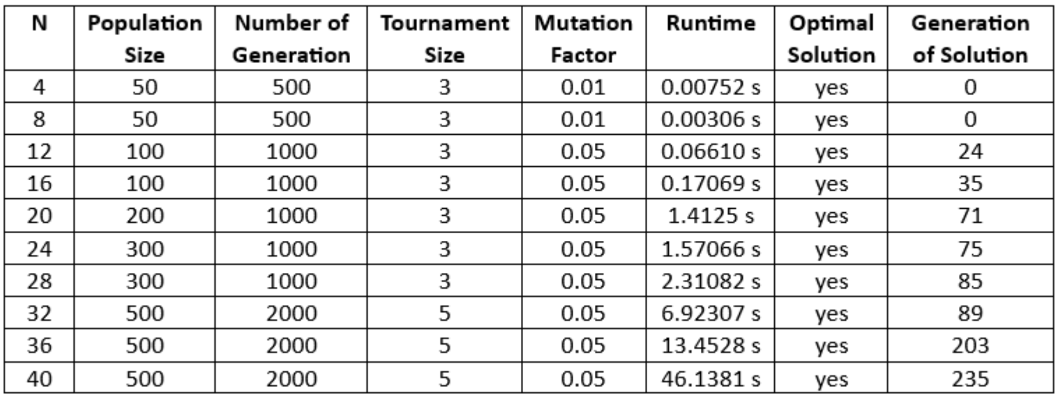
PMX crossover with swap mutation successfully found the optimal solution for values of N = 4 to N = 24 but failed for N = 28 and above within 500 generations. Also, the runtime increased as N increased. For instance; For N = 4, runtime was only 0.002044 seconds. On the other hand, for the highest value tested, N = 40, runtime increased to 6.65575 seconds and still failed to find the optimal solution.

## 5.2 Intermediate Analysis - Adjusted Parameters

The population size, number of generations, tournament size, and mutation factor are adjusted as the value of N increases. However, we are still using the same crossover operator, pmx and mutation operator, swap for this analysis.

When population size and generations were increased (up to 500 population and 2000 generations for larger N), the results improved, with optimal solutions being found for all N values up to 40. For N = 40, with population = 500 and 2000 generations, the optimal solution was found in 46.1381 seconds. If we increase the

**Table 2. Results of Adjusted Parameter Analysis**



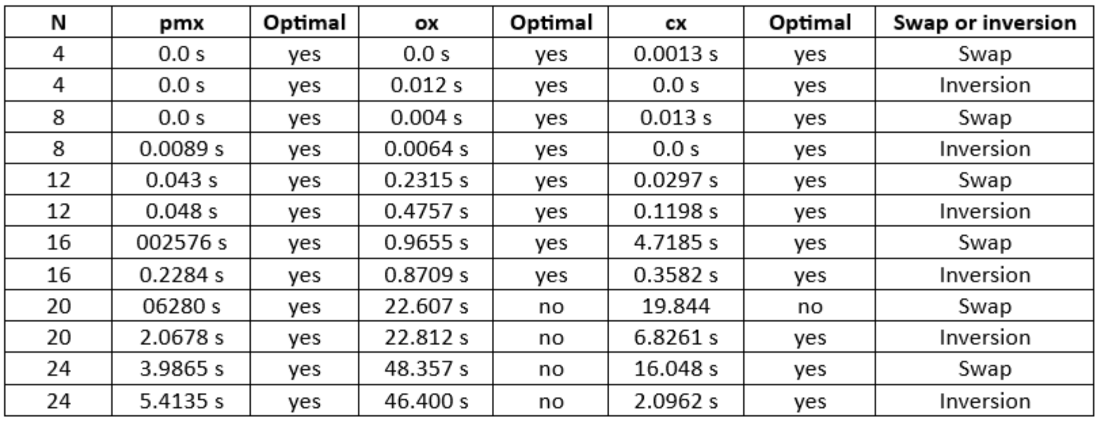
| N | Population Size | Number of Generation | Tournament Size | Mutation Factor | Runtime | Optimal Solution | Generation of Solution |
|---|---|---|---|---|---|---|---|
| 4 | 50 | 500 | 3 | 0.01 | 0.00752 s | yes | 0 |
| 8 | 50 | 500 | 3 | 0.01 | 0.00306 s | yes | 0 |
| 12 | 100 | 1000 | 3 | 0.05 | 0.06610 s | yes | 24 |
| 16 | 100 | 1000 | 3 | 0.05 | 0.17069 s | yes | 35 |
| 20 | 200 | 1000 | 3 | 0.05 | 1.4125 s | yes | 71 |
| 24 | 300 | 1000 | 3 | 0.05 | 1.57066 s | yes | 75 |
| 28 | 300 | 1000 | 3 | 0.05 | 2.31082 s | yes | 85 |
| 32 | 500 | 2000 | 5 | 0.05 | 6.92307 s | yes | 89 |
| 36 | 500 | 2000 | 5 | 0.05 | 13.4528 s | yes | 203 |
| 40 | 500 | 2000 | 5 | 0.05 | 46.1381 s | yes | 235 |

value of population size further, the runtime can be reduced further as it gives a wider space to explore.

## 5.3 Advanced Analysis - Crossover and Mutation Technique Comparison

Ordered crossover and cycle crossover operators along with inversion mutation operators are introduced to the algorithm and analysed. The performance of PMX, OX, and CX crossovers was evaluated along with swap and inversion mutations. Here the other parameter values are taken from the previous analysis table.

**Table 3. Results of Crossover and Mutation Technique Comparison Analysis**



| N | pmx | Optimal | ox | Optimal | cx | Optimal | Swap or inversion |
|---|---|---|---|---|---|---|---|
| 4 | 0.0 s | yes | 0.0 s | yes | 0.0013 s | yes | Swap |
| 4 | 0.0 s | yes | 0.012 s | yes | 0.0 s | yes | Inversion |
| 8 | 0.0 s | yes | 0.004 s | yes | 0.013 s | yes | Swap |
| 8 | 0.0089 s | yes | 0.0064 s | yes | 0.0 s | yes | Inversion |
| 12 | 0.043 s | yes | 0.2315 s | yes | 0.0297 s | yes | Swap |
| 12 | 0.048 s | yes | 0.4757 s | yes | 0.1198 s | yes | Inversion |
| 16 | 002576 s | yes | 0.9655 s | yes | 4.7185 s | yes | Swap |
| 16 | 0.2284 s | yes | 0.8709 s | yes | 0.3582 s | yes | Inversion |
| 20 | 06280 s | yes | 22.607 s | no | 19.844 | no | Swap |
| 20 | 2.0678 s | yes | 22.812 s | no | 6.8261 s | yes | Inversion |
| 24 | 3.9865 s | yes | 48.357 s | no | 16.048 s | yes | Swap |
| 24 | 5.4135 s | yes | 46.400 s | no | 2.0962 s | yes | Inversion |

PMX with **swap mutation** performed better for most N values, especially for small to medium N. In most cases, swap mutation is outperforming inversion. For example; For N = 24, swap mutation took **3.9865 seconds**, whereas inversion took **5.4135 seconds**, showing that swap was faster in this case. OX performed well for smaller N values but was significantly slower for larger N values. For example, at N = 24, OX took **48.357 seconds** with the swap mutation and failed to find the optimal solution. CX showed inconsistent performance, with slower runtimes and occasional failure to find optimal solutions. However, it is observed that cycle crossover with inversion does yield faster results in most cases in comparison with swap. From this, it is clear that inversion mutation helps in bringing in more diversity among the children in cycle crossover.

**Summary**

Based on the analysis, **PMX with swap mutation** outperforms other combinations in terms of runtime and solution optimality. It works well for both small and large N values, making it the best choice overall. OX and CX crossover techniques are less effective, particularly for larger N values, where they either take significantly longer time or fail to find optimal solutions.

## 6 Conclusions

In this assignment, we applied various genetic algorithm techniques, specifically PMX, OX, and CX crossovers, alongside swap and inversion mutations, to solve the n-queens problem. The goal was to compare these methods based on their runtime efficiency, ability to find optimal solutions, and the generation in which solutions were discovered. We conducted experiments across different values of N with adjustments to population size, the number of generations, tournament size, and mutation factors to evaluate performance.

**Recommendations for Future Work Parameter Optimization:** A simple next step could be to Implement self-tuning mechanisms to adjust its parameters dynamically based on performance metrics during its execution. Another step is to explore the greedy mutation operator to improve the performance of the algorithm.[3].Additionally, incorporating elitism to retain the best solutions across generations could further enhance the algorithm's ability to converge to the optimum.

**Explore other variants of the N-Queen problem:** N queen problem can be divided into three variant subproblems: Variant I (get one valid solution), Variant II (get a family of valid solutions), Variant III (get all valid solutions). In this assignment, we have explored Variant I. In advanced works, we can explore the other variants.[8]

**Attempt Practical Applications:**The n-queens problem is a classic example of constraint satisfaction and has practical implications in several fields. For instance, the n-queens problem is analogous to the scheduling problem where tasks are assigned to employees in a way to avoid overlap or conflict in schedules.

In conclusion, this assignment demonstrates the effectiveness of genetic algorithms in solving the n-queens problem and it also opens up numerous opportunities for further exploration and application in both theoretical and practical domains.

## References

[1] A. Eiben and J. Smith, "Evolutionary computing: The origins," in *Introduction to Evolutionary Computing*, Natural Computing Series, pp. 13–14, Springer, Berlin, Heidelberg, 2015.

[2] A. Eiben and J. Smith, "What is an evolutionary algorithm?," in *Introduction to Evolutionary Computing*, Natural Computing Series, pp. 25–48, Springer, Berlin, Heidelberg, 2015.

[3] S. Sharma and V. Jain, "Greedy mutation operator to improve performance of algorithm," in *IOP Conference Series: Materials Science and Engineering*, vol. 1116, p. 012195, 2021.

[4] A. A. Homaifar, I. Tumer, and S. Ali, "The n-queens problem and gmelic algorithm," in *Proceedings of the IEEE Southeast Conference*, pp. 262–261, IEEE, 1992.

[5] X. Hu, R. C. Eberhart, and Y. Shi, "Swarm intelligence for permutation optimization: A case study of n-queens problem," in *Proceedings of the 2003 IEEE Swarm Intelligence Symposium (SIS'03)*, (Indianapolis, IN, USA), pp. 243–246, IEEE, 2003.

[6] A. I. Humied, "Solving n-queens problem using subproblems based on genetic algorithm," *IAES International Journal of Artificial Intelligence (IJ-AI)*, vol. 7, pp. 130–137, September 2018.

[7] A. E. Eiben and J. E. Smith, *Representation, Mutation, and Recombination.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

[8] C. Jianli, C. Zhikui, W. Yuxin, and G. He, "Parallel genetic algorithm for n-queens problem based on message passing interface-compute unified device architecture," *Computational Intelligence*, vol. 36, pp. 1621–1637, 2020.