**COMPUTER PROGRAMMING-I**

**“SEMESTER PROJECT”**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



By

**2022-MC-61**

**AHSAN ABDULLAH**

To

**MISS QURUTUL AIN MASUD**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Mechatronics and Control Engineering Department

**University of Engineering and Technology, Lahore**

***TABLE OF CONTENTS***

|  |  |  |
| --- | --- | --- |
| **SR**  **NO** | **CONTENT** | **PAGE**  **NO** |
| **1** | **INTRODUCTION** | **1** |
| **2** | **METHODOLOGY** | **1** |
| **3** | **PSEUDO CODE** | **6** |
| **4** | **FLOWCHART OF GENETIC ALGORITHM** | **11** |
| **5** | **RESULT** | **11** |
| **6** | **PROBLEM FACED** | **12** |
| **7** | **REFERENCES** | **12** |

***INTRODUCTION***

* **Objective**: Determine the minimum number of chess knights needed to attack every square on an 8x8 chessboard.
* **Approach:** Implement a genetic algorithm for an optimized solution to this chess optimization problem.
* **Significance:** This project combines chess strategy with computational techniques, showcasing the practical application of genetic algorithms in real-world problem-solving.
* **Challenges:** Chessboard dynamics and computational intricacies present unique challenges in achieving an efficient solution.
* **Documentation:** This report serves as a comprehensive record of the project's journey, including theoretical foundations, methodology, challenges faced, and achieved results.
* **Practical Application:** Highlights the adaptability of genetic algorithms, demonstrating their role in solving complex problems within the context of course of Computer Programming (CP-1).

***METHODOLOGY***

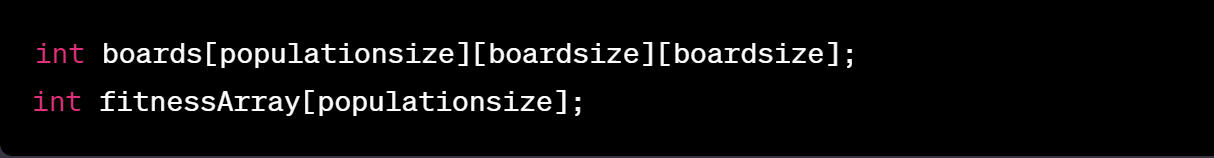
Below is the detailed methodology and explanation of the code:

A screen shot of a computer

Description automatically generated

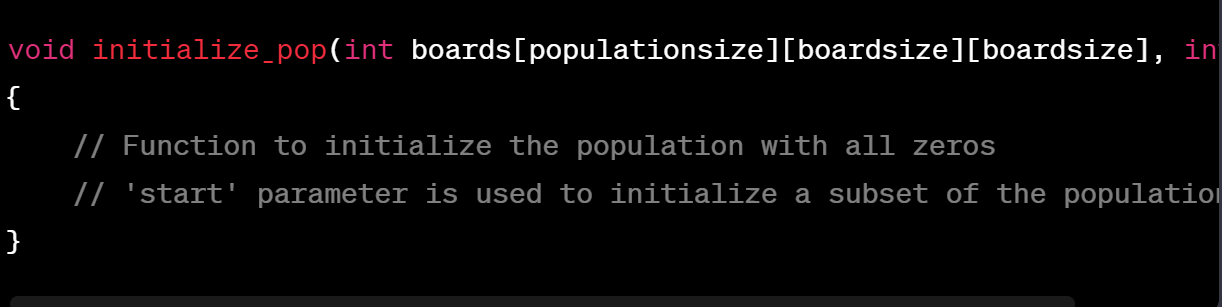
**Header Files and Definitions:**

* Includes necessary header files **(‘stdio.h’, ‘time.h’, ‘stdlib.h’).**
* Defines constants for the chessboard size **(‘boardsize’)**, the number of knights **(‘knights’)**, the population size **(‘populationsize’)**, and the number of parents **(‘parents’).**



**Global Variables:**

* Declares a 3D array ‘**boards’** to represent the population of chessboards.
* Declares an array ‘**fitnessArray’** to store fitness values for each individual in the population.



**Population Initialization Function (initialize\_pop):**

* Function to initialize the chessboard population with all elements set to zero.
* The **‘start’** parameter is used to initialize a subset of the population.

A black screen with white text

Description automatically generated

**Random Knight Placement Function (placeRandomKnights):**

* Function to randomly place a specified number of knights on each chessboard, avoiding duplicate placements.

A screen shot of a computer

Description automatically generated

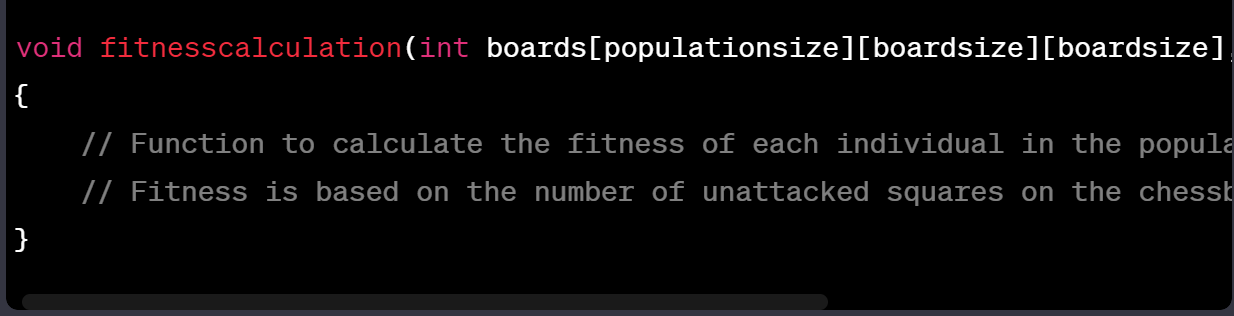
**Chessboard Placement Function (Placement):**

* Function to print the current placement of knights on a single chessboard.



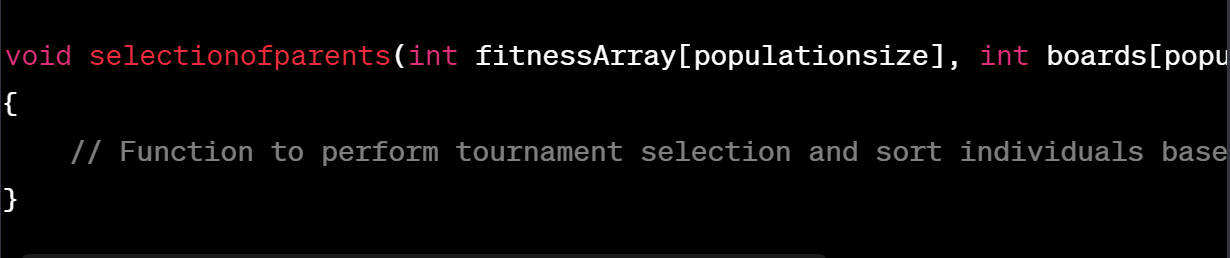
**Applying Attacks Function (applyingattacks):**

* Function to mark squares attacked by knights on all chessboards based on their possible moves.



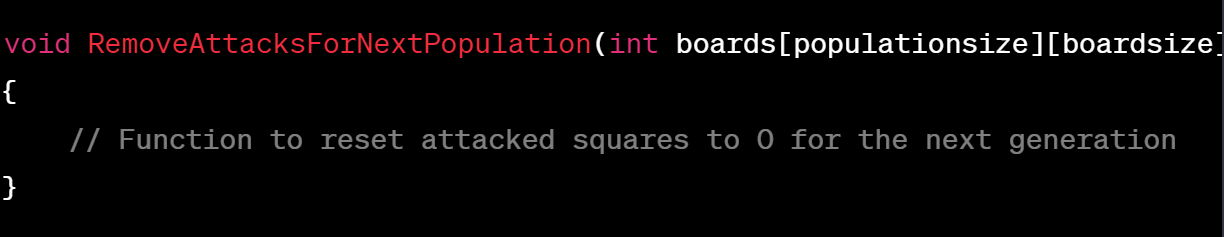
**Fitness Calculation Function (fitnesscalculation):**

* Function to calculate the fitness of each individual in the population based on the number of unattacked squares on the chessboard.

******

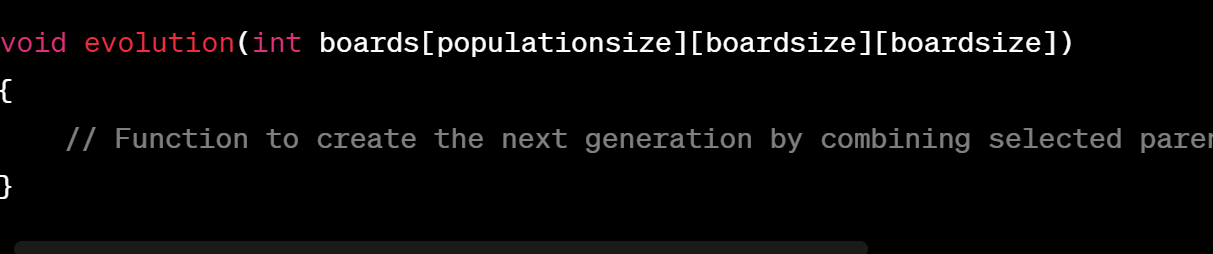
**Parent Selection Function (selectionofparents):**

* Function to perform tournament selection, sorting individuals based on fitness, and swapping their positions in the population accordingly.



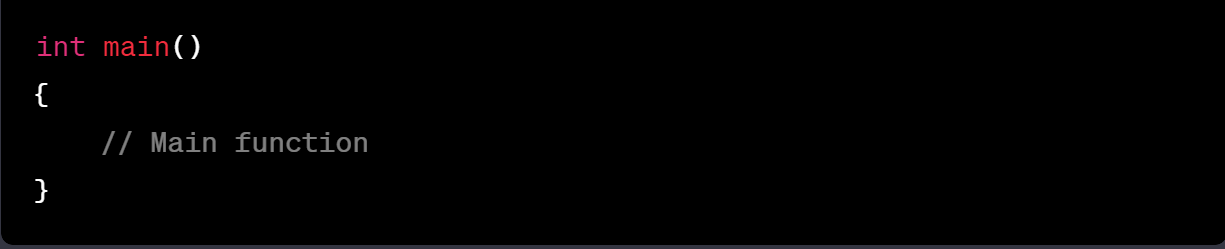
**Reset Attacks Function (RemoveAttacksForNextPopulation):**

* Function to reset attacked squares to 0 on all chessboards in preparation for the next generation.



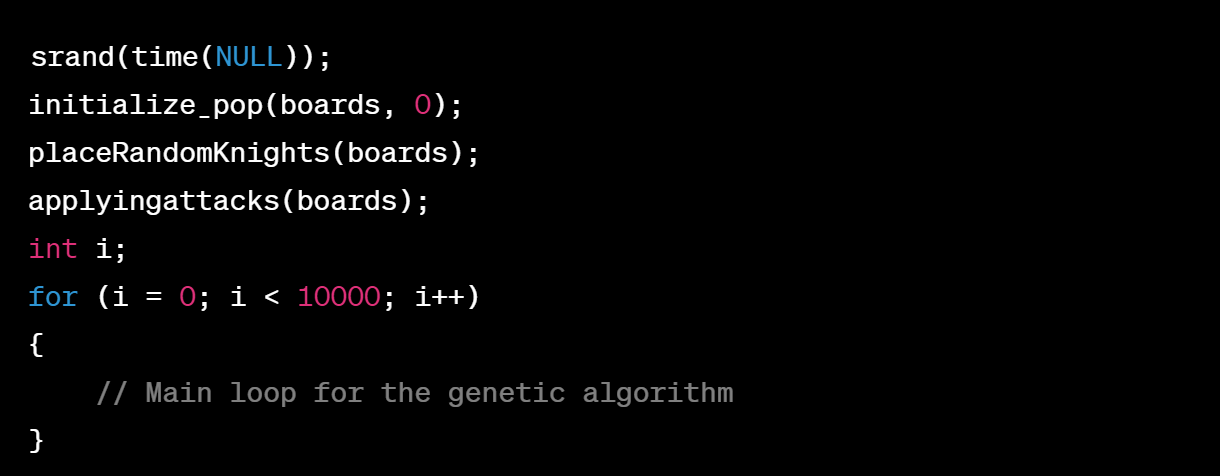
**Evolution Function (evolution):**

* Function to create the next generation by combining selected parents, preserving the fittest individuals, and introducing mutations to enhance genetic diversity.



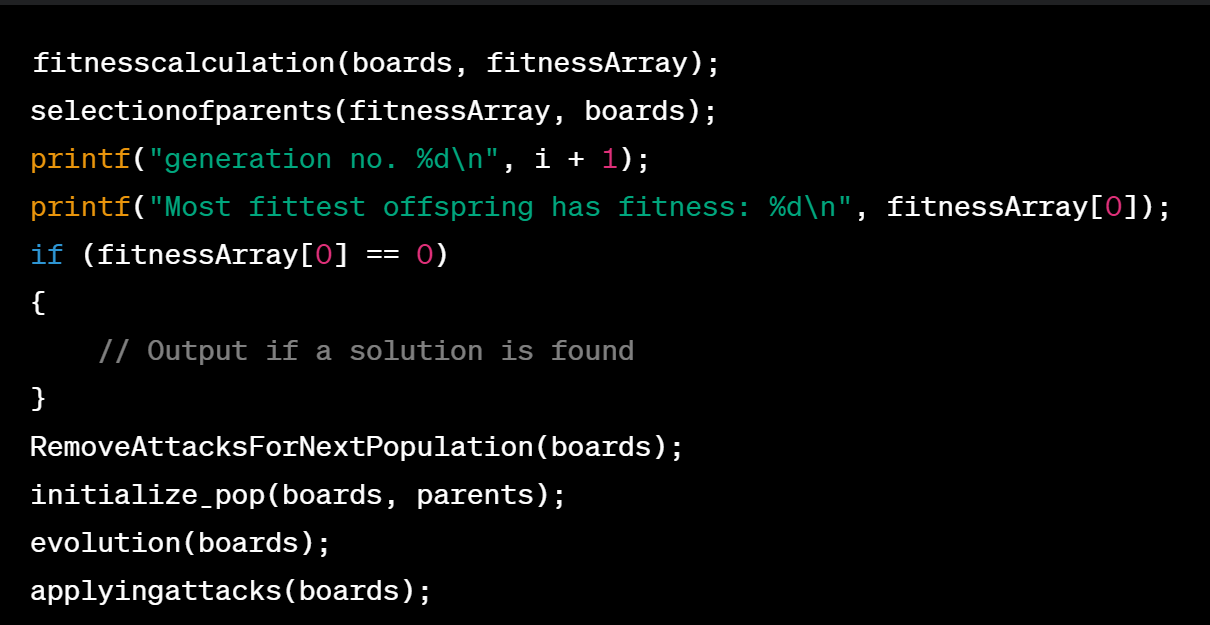
**Main Function (main):**

* Entry point of the program.
* Initializes the random number generator.
* Calls functions to initialize the population, place knights, apply attacks, and perform the genetic algorithm loop.



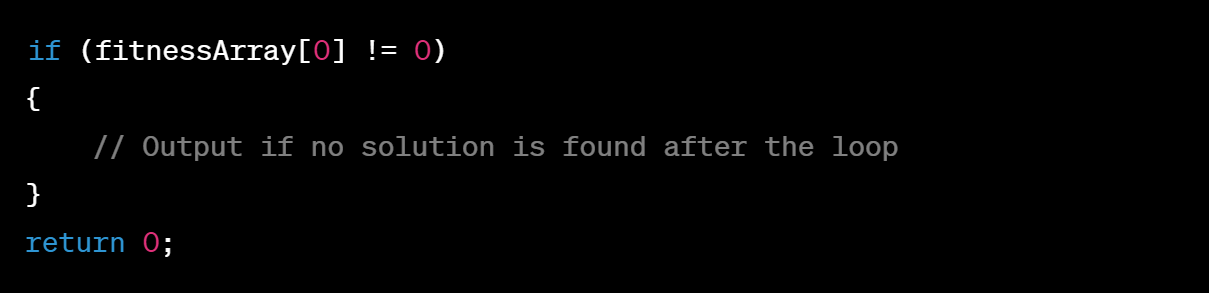
**Initialization in Main:**

* Seeds the random number generator with the current time.
* Initializes the population, places knights randomly, and applies attacks to the chessboards.
* Starts a loop for the genetic algorithm, iterating up to 10,000 generations or until a solution is found.



**Genetic Algorithm Steps in Main Loop:**

* Calculates fitness, performs parent selection, and prints information about the current generation.
* Checks for a solution and outputs information if found.
* Resets attacked squares, initializes the second half of the population, performs evolution, and applies attacks for the next generation.



**Solution Output or Termination:**

* If no solution is found after the loop, outputs the most fit individual of the last generation.
* Returns 0, indicating successful program execution.

***PSEUDO CODE***

// Constants

boardsize = 8

knights = 16

populationsize = 500

parents = populationsize / 2

// Global Variables

boards[populationsize][boardsize][boardsize]

fitnessArray[populationsize]

// Function to initialize the population

initialize\_pop(boards, start)

for z from start to populationsize

for x from 0 to boardsize

for y from 0 to boardsize

boards[z][x][y] = 0

// Function to randomly place knights on the chessboards

placeRandomKnights(boards)

for z from 0 to populationsize

for k from 0 to knights

x = random number from 0 to boardsize

y = random number from 0 to boardsize

if boards[z][x][y] != 1

boards[z][x][y] = 1

else

k = k - 1

// Function to print the placement of knights on a chessboard

Placement(board)

k = 0

print “ a b c d e f g h"

for i from 0 to boardsize

print boardsize - i, "|"

for j from 0 to boardsize

if board[i][j] == 1

print "k"

k = k + 1

else if board[i][j] == 2

print "X"

else

print " "

print "|"

print "\n +-+-+-+-+-+-+-+-+"

print "no. of knights = ", k

// Function to mark squares attacked by knights on all chessboards

applyingattacks(boards)

knightMoves = [[2, 1], [2, -1], [-2, 1], [-2, -1], [1, 2], [1, -2], [-1, 2], [-1, -2]]

for z from 0 to populationsize

for x from 0 to boardsize

for y from 0 to boardsize

if boards[z][x][y] == 1

for attack from 0 to 7

Rx = x + knightMoves[attack][0]

Ry = y + knightMoves[attack][1]

if Rx >= 0 and Rx < boardsize and Ry >= 0 and Ry < boardsize and boards[z][Rx][Ry] != 1

boards[z][Rx][Ry] = 2

// Function to calculate the fitness of each individual in the population

fitnesscalculation(boards, fitnessArray)

for P from 0 to populationsize

fitness = 0

for i from 0 to boardsize

for j from 0 to boardsize

if boards[P][i][j] == 0

fitness = fitness + 1

fitnessArray[P] = fitness

// Function to perform tournament selection and sort individuals based on fitness

selectionofparents(fitnessArray, boards)

for i from 0 to populationsize

for j from i + 1 to populationsize

if fitnessArray[j] < fitnessArray[i]

swap fitnessArray[i] and fitnessArray[j]

swap boards[i] and boards[j]

// Function to reset attacked squares to 0 for the next generation

RemoveAttacksForNextPopulation(boards)

for P from 0 to populationsize

for R from 0 to boardsize

for C from 0 to boardsize

if boards[P][R][C] == 2

boards[P][R][C] = 0

// Function to create the next generation by combining selected parents and introducing mutations

evolution(boards)

final\_position[populationsize][2]

k = 0

for p from 0 to parents

for r from 0 to boardsize

for c from 0 to boardsize

if boards[p][r][c] == 1

boards [p + parents][r][c] = 1

k = k + 1

if k == knights / 2

break

if k == knights / 2

break

final\_position[p][0] = r

final\_position[p][1] = c

k = 0

for p from 0 to parents - 1

for r from 0 to boardsize

for c from 0 to boardsize

if boards[p][r][c] == 1 and r > final\_position[p][0] and c > final\_position[p][1]

boards [p + 1 + parents][r][c] = 1

k = k + 1

if k == knights

break

if k == knights

break

k = 0

// mutation

for i from 0 to populationsize

k = 0

for r from 0 to boardsize

for c from 0 to boardsize

if boards[i][r][c] == 1

k = k + 1

if k < knights

while k < knights

row = random number from 0 to boardsize

col = random number from 0 to boardsize

if boards[i][row][col]! = 1

boards[i][row][col] = 1

k = k + 1

else if k > knights

while k > knights

row = random number from 0 to boardsize

col = random number from 0 to boardsize

if boards[i][row][col] == 1

boards[i][row][col] = 0

k = k - 1

// Main Function

main ()

srand(time (NULL))

Initialize\_pop(boards, 0)

placeRandomKnights(boards)

applyingattacks(boards)

for i from 0 to 10000

fitnesscalculation(boards, fitnessArray)

selectionofparents(fitnessArray, boards)

print "generation no. ", i + 1

print "Most fittest offspring has fitness: ", fitnessArray[0]

if fitnessArray[0] == 0

print "Solution found after ", i, " generations”.

Placement(boards [0])

Break

RemoveAttacksForNextPopulation(boards)

initialize\_pop(boards, parents)

evolution(boards)

applyingattacks(boards)

if fitnessArray[0] != 0

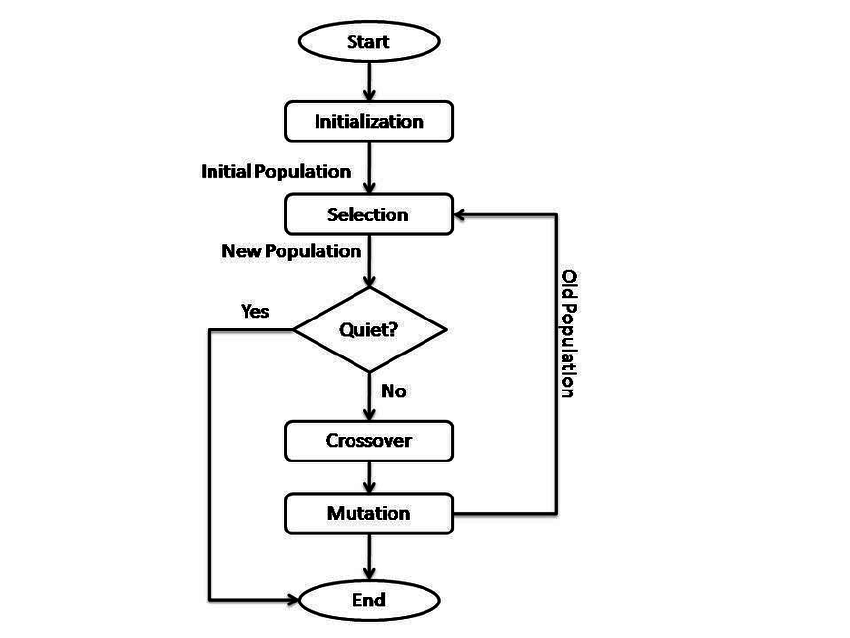
print "No solution found after ", i, " generations”.

print "Here's the fittest offspring:"

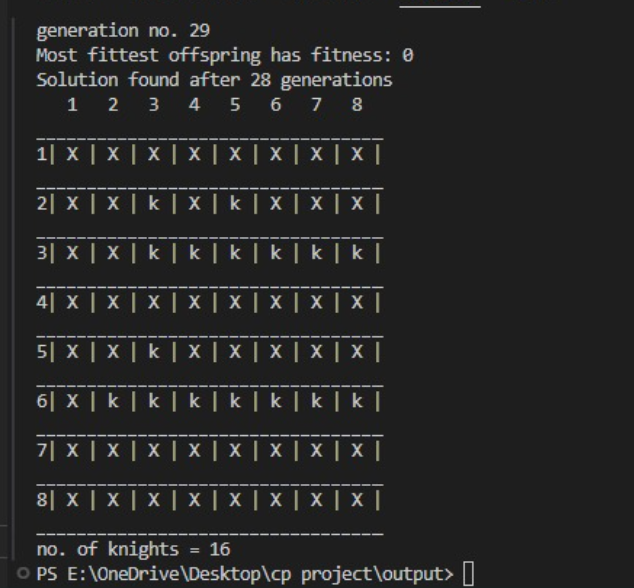
Placement (boards [0])

return 0

***FLOWCHART OF GENETIC ALGORITHM***



***RESULT***

******

***PROBLEMS FACED***

**New Algorithm Concept:** Understanding the genetic algorithm concept, especially for a problem different from the familiar N-Queens, posed a learning curve.

**Transition to 3D Array:** Adapting from a 2D array (N-Queens) to a 3D array for the knight's problem introduced new complexity in representing solutions.

**Fitness Function Design:** Designing an effective fitness function for the knight's problem presented challenges in determining what constitutes a 'fit' solution.

**Crossover and Mutation Strategies:** Implementing appropriate crossover and mutation operations for knights required experimentation and tuning.

**Implementing Genetic Operators:** Implementing selection, crossover, and mutation operations demanded a deep understanding of their impact.

**Understanding Output**: Interpreting the algorithm's output, especially in cases where no solution was found, required careful analysis.

Overcoming these challenges involved a mix of theoretical understanding, experimentation, and debugging to fine-tune the genetic algorithm for the knight's problem.

***REFERENCES***

* <https://youtu.be/bbkdiUbou74?feature=shared>
* <https://en.wikipedia.org/wiki/Mathematical_chess_problem>
* https://www.geeksforgeeks.org/the-knights-tour-problem/