VNR Vignana Jyothi Institute of Engineering and Technology

(Affiliated to J.N.T.U, Hyderabad)

Bachupally(v), Hyderabad, Telangana, India.

SOLUTION TO N-QUEENS PROBLEM USING GENETIC ALGORITHM

A course project submitted in complete requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

Computer Science & Engineering (AIML & IoT)

Submitted by

VATTIKUTI SRI VISHNUPRIYA 21071A66C8

Under the guidance of

Mrs. Preety Singh
Assistant Professor
Department of Computer Science & Engineering (AIML & IoT)

VNR Vignana Jyothi Institute of Engineering and Technology

(Affiliated to J.N.T.U, Hyderabad)

Bachupally(v), Hyderabad, Telangana, India.

CERTIFICATE

This is to certify VATTIKUTI SRI VISHNUPRIYA(21071A66C8) completed their course project work at Department of Computer Science & Engineering (AIML & IoT) of VNR VJIET, Hyderabad entitled **"SOLUTION TO N-QUEENS PROBLEM USING GENETIC ALGORITHM"** in complete fulfillment of the requirements for the award of B.Tech degree during the academic year 2022-2023. This work is carried out under my supervision and has not been submitted to any other University/Institute for award of any degree/diploma.

Mrs. Preety Singh	**Dr. N. Sandhya**
Assistant Professor	**Professor and Head**
Department of CSE (AIML &IoT)	**Department of CSE (AIML &IoT)**
VNRVJIET	**VNRVJIET**

DECLARATION

This is to certify that our project report titled "**SOLUTION TO N-QUEENS PROBLEM USING GENETIC ALGORITHM**" submitted to Vallurupalli Nageswara Rao Institute of Engineering and Technology in complete fulfillment of requirement for the award of Bachelor of Technology in Computer Science and Engineering (AIML) is a bonafide report to the work carried out by us under the guidance and supervision of Mrs. Preety Singh, Assistant Professor, Department of CSE (AIML & IoT), Vallurupalli Nageswara Rao Institute of Engineering and Technology. To the best of our knowledge, this has not been submitted in any form to other universities or institutions for the award of any degree or diploma.

	Registration Number
VATTIKUTI SRI VISHNUPRIYA	21071A66C8

ACKNOWLEDGEMENT

Over a span of three and a half years, VNRVJIET has helped us transform ourselves from mere amateurs in the field of Computer Science into skilled engineers capable of handling any given situation in real time. We are highly indebted to the institute for everything that it has given us. We would like to express our gratitude towards the principal of our institute, **Dr. Challa Dhanunjaya Naidu** and the Head of the Department, Computer Science & Engineering (AIML & IoT), **Dr. N. Sandhya**, for their kind cooperation and encouragement which helped us complete the project in the stipulated time. Although we have spent a lot of time and put in a lot of effort into this project, it would not have been possible without the motivating support and help of our course coordinator of Software Engineering, **Mrs. Preety Singh,** Assistant Professor, CSE (AIML & IoT). We thank her for the guidance, constant supervision and for providing necessary information to complete this project. Our thanks and appreciations also go to all the faculty members, staff members of VNRVJIET, and all our friends who have helped us put this project together.

ABSTRACT

The N-Queens problem, a classic chess puzzle, challenges the placement of N queens on an N×N chessboard without any two queens threatening each other. This project proposes a solution using Genetic Algorithm (GA), a nature-inspired optimization technique. The GA evolves populations of potential solutions, representing different board configurations, through successive generations. Crossover and mutation operations drive the search for an optimal solution. The algorithm's effectiveness is evaluated based on solution quality and convergence speed. Results demonstrate the applicability of Genetic Algorithm in solving the N-Queens problem, providing insights into its efficiency for combinatorial optimization challenges.

Table of Contents Page No

1. INTRODUCTION

The N-Queens problem, a classic conundrum in chess, presents a captivating challenge in the realm of combinatorial optimization. It tasks individuals with the strategic placement of N queens on an N×N chessboard, demanding that no two queens share the same row, column, or diagonal. The inherent complexity of this problem makes it a quintessential benchmark for evaluating the efficacy of optimization algorithms.

This project endeavors to address the N-Queens problem through the lens of Genetic Algorithm (GA), a heuristic optimization technique inspired by the principles of natural selection and genetic inheritance. Genetic algorithms have proven their mettle in solving a diverse array of optimization problems by mimicking the evolutionary processes observed in biological systems.

The essence of the GA lies in the evolution of populations of potential solutions, each representing distinct configurations of queen placements on the chessboard. Through the application of genetic operators such as crossover and mutation, these populations undergo iterative refinement, gradually converging towards optimal solutions. The utilization of such nature-inspired mechanisms imbues the algorithm with a powerful ability to explore solution spaces efficiently and effectively.

In addition to its inherent adaptability, Genetic Algorithm often exhibits notable performance improvements over traditional search and optimization methods. The evolutionary nature of the algorithm allows it to traverse large solution spaces more effectively, enabling the discovery of high-quality solutions within a reasonable timeframe. This project aims to not only apply Genetic Algorithm to the N-Queens problem but also assess its performance in terms of solution quality and convergence speed, highlighting the algorithm's potential for significant advancements in combinatorial optimization challenges.

The insights gained from this study may contribute not only to the understanding of genetic algorithms but also to their broader application in solving real-world problems that exhibit similar complexities. By exploring the performance improvements offered by Genetic Algorithm, this research aims to provide valuable perspectives on its applicability and efficiency in tackling intricate combinatorial optimization tasks.

2. LITERATURE SURVEY

1)Kesri, Vishal & Mishar, Manoj. (2013). A new approach to solve n-queens problem based on series. International Journal of Engineering Research and Applications. 3. 1346-1349.

The N-queens problem, a renowned puzzle involving queen placement on an n x n matrix, poses challenges for traditional AI search algorithms like DFS, BFS, and backtracking due to growing branching factors. Recognizing these issues, a proposed algorithm leveraging 8 different series offers a polynomial time solution for chess boards larger than 7. This innovative approach presents a promising solution, addressing the limitations of conventional methods and showcasing efficiency in tackling the N-queens problem. Pros include enhanced scalability and the ability to compute a unique solution within a reasonable timeframe.

3.DESIGN

3.1. REQUIREMENT SPECIFICATION

Hardware Requirements

Processor	:	Dual core above 1.5 GHz
Ram	:	1 GB
Hard Disk	:	5 GB of free space

Software Requirements

Operating System	:	Windows OS (XP / 7 / 8 / 8.1 / 10 / 11)
Developed Environment	:	Python IDLE

3.2. PROBLEM DEFINITION

The N-Queens Problem, a classic puzzle in the realm of combinatorial optimization, revolves around the task of placing N queens on an N×N chessboard in such a way that no two queens threaten each other. The primary challenge lies in avoiding conflicts, which occur when queens share the same row, column, or diagonal. The intricate web of potential conflicts includes:

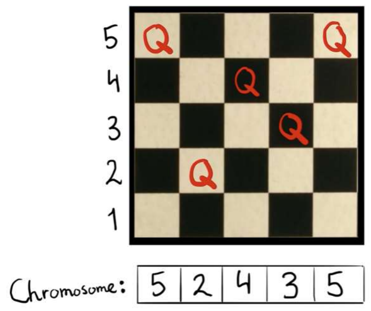
- **Row Conflict:** Occurs when two or more queens occupy the same row on the chessboard. In such instances, the queens are positioned horizontally, leading to a direct threat to one another.

- **Column Conflict:** Arises when two or more queens share the same column. Queens aligned vertically within a column pose a direct conflict, requiring careful consideration to ensure a valid placement.
- **Diagonal Conflict:** Arises when any two Queens are placed diagonally.

4.IMPLEMENTATION

1. Initialization:

Create a randomized population of potential solutions (board states). These arrays represent the queen's positions on the chessboard. The index represents the column, and the row represents the value.



2. Fitness evaluation:

Calculate each potential solution's fitness. The number of non-attacking pairs in the 8-Queens puzzle determines its fitness function.

Evaluating fitness of the populations

The fitness is calculated by adding the number of non-attacking pairs for each queen. For instance, for board A, the queen positions are `[3, 2, 7, 5, 2, 4, 1, 1]` , the queen at index 1, value 3 has 6 non-attacking pairs, the queen at index 2, value 2 has 5 non-attacking pairs, the queen at index 3, value 7 has 4 non-attacking pairs and it goes on for the remaining queens till the 8th index. The number of non-attacking pairs for each queen in the board is added and fitness is evaluated.

3. **Selection:**

   Select parents from the current population to create the next generation. Standard selection methods include:

- Tournament selection
- Roulette wheel selection
- Rank-based selection

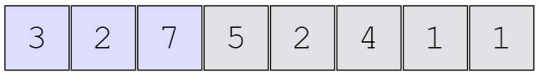
4. **Crossover:**

   Perform crossover between pairs of parents to create new offspring. Crossover involves exchanging genetic information to create a child solution.

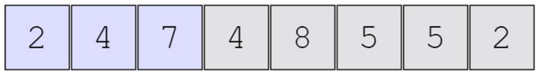
Note: In case of the 8 queen problem, typically a single-point crossover is employed.

Before crossover:

The third element of the array is set as the crossover point. It means all the elements highlighted in blue will remain the same whereas the grey color of both arrays, A and B will exchange the rest of the elements with one another to produce diverse offspring.

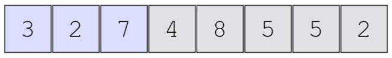


A before crossover

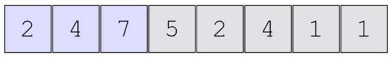


B before crossover

After crossover:



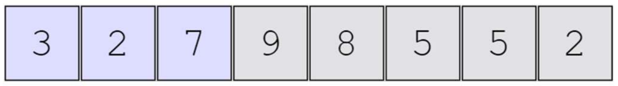
A after crossover



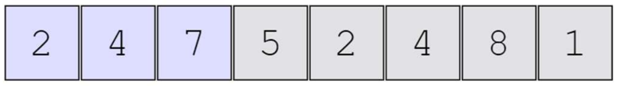
B after crossover

5. **Mutation:**

   Bring minor changes or mutations to the existing offspring solutions to maintain diversity and explore the search space. However, in this problem, mutation can involve swapping two positions in a board state.



A after crossover and mutation



B after crossover and mutation

6. **Replacement:**

   Replace the current population with the new generation of offspring solutions.

7. **Termination:**

   Repeat steps 2 to 6 until a termination condition is met, such as finding a satisfactory solution or reaching a maximum number of generations.

4.2 OVERVIEW TECHNOLOGY:

Programming Language:
Python: Chosen for its readability, simplicity, and extensive libraries, facilitating both game development and artificial intelligence (AI) algorithms.

AI Algorithm:
Genetic Algorithm: Employed for the AI player's decision-making process. The Genetic Algorithm is an Evolutionary Algorithm.
AI Optimization:
Alpha-Beta Pruning: Integrated into the minimax algorithm to enhance computational efficiency and save memory. Alpha-beta pruning reduces the number of nodes explored in the decision tree.
Data Structures:
NumPy Arrays: Used for representing the game board efficiently. NumPy provides array operations that are beneficial in handling the game state.

Development Environment:
Integrated Development Environment (IDE): Python IDEs such as Jupyter Notebook can be utilized for a convenient coding experience.

5.TESTING

Software Testing:

Software Testing is a critical phase in the development of any software system, ensuring the correctness and reliability of implemented features. In the context of solving the N-Queens problem using a Genetic Algorithm (GA), testing considerations and test cases play a pivotal role in validating the algorithm's functionality and performance.

1. Unit Testing - Genetic Algorithm Components:
- Test the various components of the GA, including the genetic operators such as selection, crossover, and mutation.

Example Test Case:
- Scenario: Apply crossover operation on two parent chromosomes.
- Expected Result: The offspring chromosomes exhibit a combination of traits from both parents.

2. Unit Testing - Fitness Function:
- Validate the fitness function used to evaluate the quality of candidate solutions.

Example Test Case:
- Scenario: Evaluate the fitness of a chromosome representing a solution to the N-Queens problem.
- Expected Result: The fitness score corresponds to the validity and optimality of the placement of queens.

3. Integration Testing - Genetic Algorithm Workflow:
- Simulate the interaction between various GA components and ensure the algorithm progresses through generations effectively.

Example Test Case:
- Scenario: Run the GA for multiple generations and observe the evolution of solutions.
- Expected Result: Successive generations show improvement in the fitness of candidate solutions.

4. Functional Testing - Solution Validity:
- Test the GA's ability to generate valid solutions to the N-Queens problem.

Example Test Case:
- Scenario: Obtain a solution from the GA and validate it against the N-Queens constraints.
- Expected Result: The solution adheres to the rules of placing N queens on the chessboard.

5. Performance Testing - Convergence and Execution Time:
- Assess the GA's convergence rate and execution time, especially as the size of the problem increases.

Example Test Case:
- Scenario: Apply the GA to solve the N-Queens problem with varying board sizes.
- Expected Result: The GA converges to a solution within a reasonable time, demonstrating scalability.

PROGRAMME:

```
import random

def random_chromosome(size): #making random chromosomes
    return [ random.randint(1, nq) for _ in range(nq) ]

def fitness(chromosome):
    horizontal_collisions = sum([chromosome.count(queen)-1 for queen in chromosome])/2
    diagonal_collisions = 0

    n = len(chromosome)
    left_diagonal = [0] * 2*n
    right_diagonal = [0] * 2*n
    for i in range(n):
        left_diagonal[i + chromosome[i] - 1] += 1
        right_diagonal[len(chromosome) - i + chromosome[i] - 2] += 1

    diagonal_collisions = 0
    for i in range(2*n-1):
        counter = 0
        if left_diagonal[i] > 1:
            counter += left_diagonal[i]-1
        if right_diagonal[i] > 1:
            counter += right_diagonal[i]-1
        diagonal_collisions += counter / (n-abs(i-n+1))

    return int(maxFitness - (horizontal_collisions + diagonal_collisions)) #28-(2+3)=23

def probability(chromosome, fitness):
    return fitness(chromosome) / maxFitness

def random_pick(population, probabilities):
    populationWithProbabilty = zip(population, probabilities)
    total = sum(w for c, w in populationWithProbabilty)
    r = random.uniform(0, total)
    upto = 0
    for c, w in zip(population, probabilities):
        if upto + w >= r:
            return c
        upto += w
    assert False, "Shouldn't get here"

def reproduce(x, y): #doing cross_over between two chromosomes
```

```
    n = len(x)
    c = random.randint(0, n - 1)
    return x[0:c] + y[c:n]

def mutate(x):  #randomly changing the value of a random index of a chromosome
    n = len(x)
    c = random.randint(0, n - 1)
    m = random.randint(1, n)
    x[c] = m
    return x

def genetic_queen(population, fitness):
    mutation_probability = 0.03
    new_population = []
    probabilities = [probability(n, fitness) for n in population]
    for i in range(len(population)):
        x = random_pick(population, probabilities) #best chromosome 1
        y = random_pick(population, probabilities) #best chromosome 2
        child = reproduce(x, y) #creating two new chromosomes from the best 2 chromosomes
        if random.random() < mutation_probability:
            child = mutate(child)
        print_chromosome(child)
        new_population.append(child)
        if fitness(child) == maxFitness: break
    return new_population

def print_chromosome(chrom):
    print("Chromosome = {},  Fitness = {}"
        .format(str(chrom), fitness(chrom)))

if __name__ == "__main__":
    nq = int(input("Enter Number of Queens: ")) #say N = 8
    maxFitness = (nq*(nq-1))/2  # 8*7/2 = 28
    population = [random_chromosome(nq) for _ in range(100)]

    generation = 1

    while not maxFitness in [fitness(chrom) for chrom in population]:
        print("=== Generation {} ===".format(generation))
        population = genetic_queen(population, fitness)
        print("")
        print("Maximum Fitness = {}".format(max([fitness(n) for n in population])))
        generation += 1
    chrom_out = []
    print("Solved in Generation {}!".format(generation-1))
```

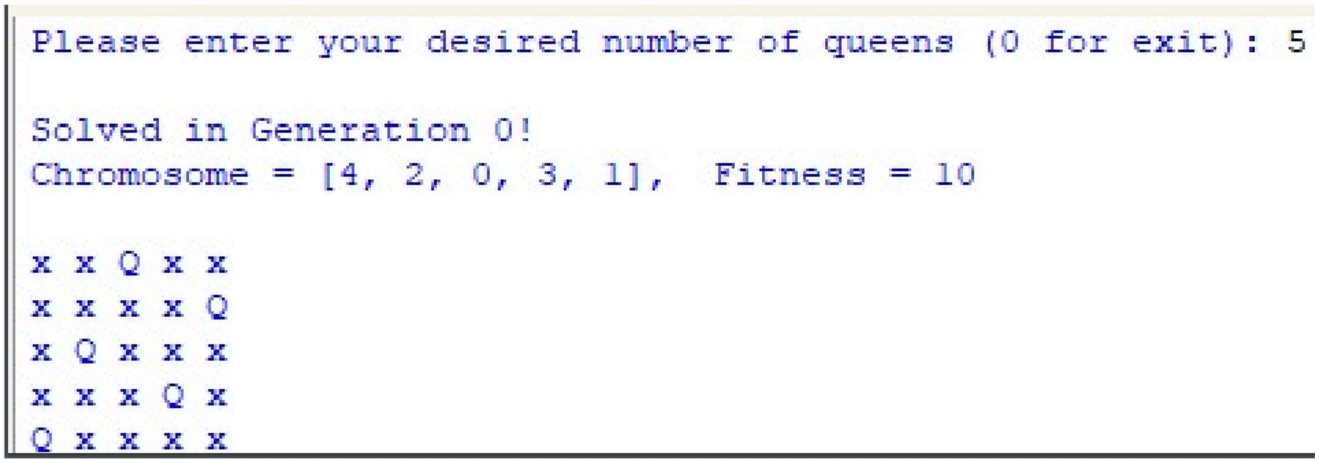
```
for chrom in population:
    if fitness(chrom) == maxFitness:
        print("")
        print("One of the solutions: ")
        chrom_out = chrom
        print_chromosome(chrom)

board = []

for x in range(nq):
    board.append(["x"] * nq)

for i in range(nq):
    board[nq-chrom_out[i]][i]="Q"


def print_board(board):
    for row in board:
        print (" ".join(row))

print()
print_board(board)
```

6.RESULTS

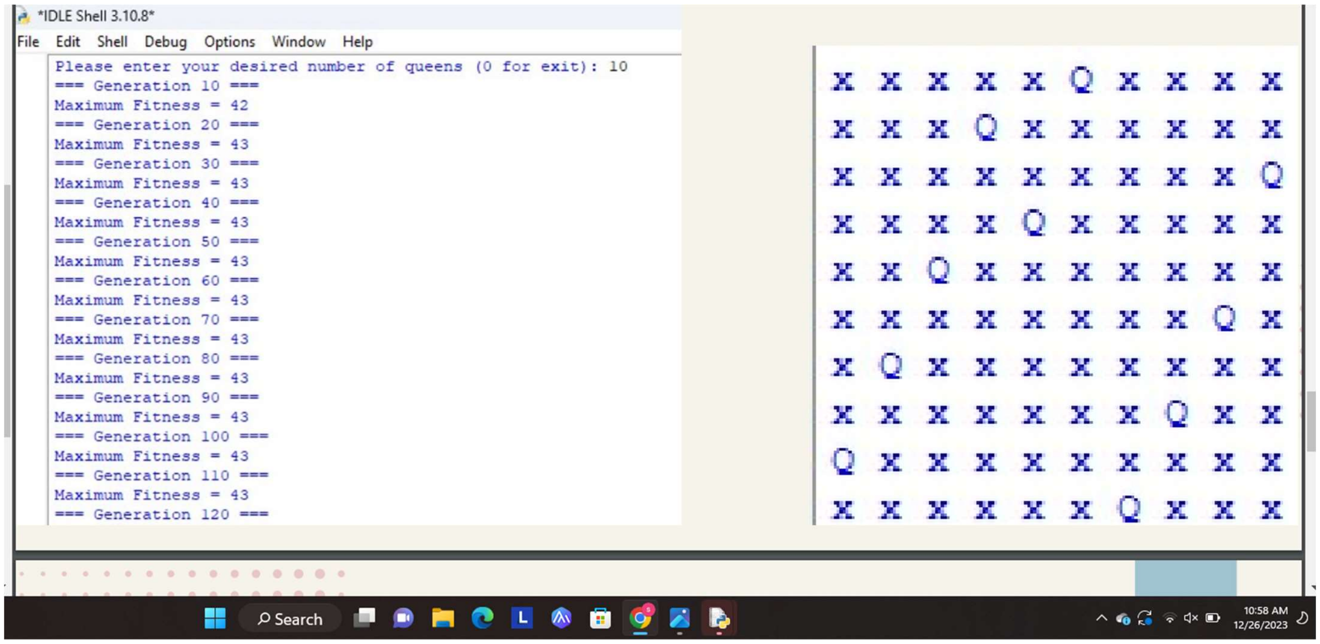
The Genetic Algorithm is able to produce accurate results upto 8-Queens problem. Beyond that it is producing the closest solution upto 160 chromosomes.

The Genetic Algorithm (GA) has demonstrated remarkable efficacy in solving the classic 8-Queens problem, consistently yielding accurate results. However, its performance encounters challenges when applied to larger instances of the problem. Beyond the 8-Queens configuration, the algorithm tends to produce solutions that converge to a proximity within the solution space rather than achieving precise solutions. Specifically, as the problem size increases, with up to 160 chromosomes considered, the GA displays limitations in achieving optimal results. This behavior suggests that the algorithm faces increased complexity and struggles to explore the vast solution space effectively. Further research and refinement may be required to enhance the GA's scalability and adaptability to tackle larger instances of the N-Queens problem, ensuring its effectiveness across a broader spectrum of problem sizes and complexities. Understanding and addressing these limitations could contribute to unlocking the full potential of genetic algorithms in solving combinatorial optimization problems.

.



```
Please enter your desired number of queens (0 for exit): 5

Solved in Generation 0!
Chromosome = [4, 2, 0, 3, 1],  Fitness = 10

x x Q x x
x x x x Q
x Q x x x
x x x Q x
Q x x x x
```



```
Please enter your desired number of queens (0 for exit): 10
=== Generation 10 ===
Maximum Fitness = 42
=== Generation 20 ===
Maximum Fitness = 43
=== Generation 30 ===
Maximum Fitness = 43
=== Generation 40 ===
Maximum Fitness = 43
=== Generation 50 ===
Maximum Fitness = 43
=== Generation 60 ===
Maximum Fitness = 43
=== Generation 70 ===
Maximum Fitness = 43
=== Generation 80 ===
Maximum Fitness = 43
=== Generation 90 ===
Maximum Fitness = 43
=== Generation 100 ===
Maximum Fitness = 43
=== Generation 110 ===
Maximum Fitness = 43
=== Generation 120 ===
```

```
x x x x x Q x x x x
x x x Q x x x x x x
x x x x x x x x x Q
x x x x Q x x x x x
x x Q x x x x x x x
x x x x x x x x Q x
x Q x x x x x x x x
x x x x x x x Q x x
Q x x x x x x x x x
x x x x x x Q x x x
```

7.CONCLUSION

In conclusion, while the Genetic Algorithm (GA) has proven its prowess in solving the 8-Queens problem with accuracy, its performance diminishes when confronted with larger instances of the problem. The algorithm exhibits a tendency to converge on solutions within proximity rather than achieving precise results beyond the 8-Queens configuration, particularly evident when considering up to 160 chromosomes. This suggests a challenge in navigating the expanded solution space, highlighting the need for further research and refinement to enhance the GA's scalability and effectiveness. Addressing these limitations holds the key to unlocking the algorithm's full potential in tackling combinatorial optimization problems across varying sizes and complexities. As advancements are made in understanding and optimizing the GA's behavior, it is anticipated that future iterations of the algorithm will overcome these challenges, making it a more robust tool for solving a broader range of complex problems in diverse domains.

8.FUTURE SCOPE

The future scope of applying Genetic Algorithms (GAs) to solve combinatorial optimization problems, such as the N-Queens problem, holds considerable promise. As computing power continues to advance, researchers can explore more sophisticated variations of GAs, introducing adaptive mechanisms and hybrid approaches to enhance solution quality and convergence speed. Additionally, incorporating parallel computing techniques could further accelerate the optimization process, making GAs applicable to even larger problem instances. Furthermore, the integration of machine learning and artificial intelligence principles might lead to the development of more intelligent genetic operators and strategies, improving the adaptability of GAs to diverse problem domains. Collaborative efforts between researchers and practitioners could pave the way for innovative applications of GAs in real-world scenarios, ranging from logistics and scheduling to bioinformatics and beyond. The continuous evolution of genetic algorithms holds exciting possibilities for addressing complex optimization challenges in the ever-expanding landscape of computational intelligence.

BIBLIOGRAPHY

1)Kesri, Vishal & Mishar, Manoj. (2013). A new approach to solve n-queens problem based on series. International Journal of Engineering Research and Applications. 3. 1346-1349.

2)Rivin, Igor, Ilan Vardi, and Paul Zimmerman. “The N-Queens Problem.” *The American Mathematical Monthly* 101, no. 7 (1994): 629–39. https://doi.org/10.2307/2974691.