Project Number : **4**

Project Name : **Knight's Tour Problem Solver**

Project drive (Shortened URL) : https://rb.gy/n1eg1o

Team Members :

Member #	Member ID	Member Name	Member Department
1	20210565	عبدالواحد رجب عبدالواحد حماد	Dept. CS
2	20210531	عبدالرحمن مصطفي حامد محمود	Dept. CS
3	202102505	عبدالرحمن رمضان فتحي محمد	Dept. AI
4	20210261	جومانا محمد السيد طلبة	Dept. CS
5	20210166	الاء حسن عبدالرحمن عبدالرحمن	Dept. CS
6	20210172	الاء ممدوح احمد عثمان	Dept. CS

Project drive : https://drive.google.com/drive/u/0/mobile/folders/1lmGLXOc6Z-KDzV1ogpPCNRqLLzr6r2IX

1. Introduction and Overview

1.1 Project Idea and Overview :

The ”Knight’s Tour Problem Solver” is a classic chess puzzle that involves finding a sequence of moves for a knight on an n x n chessboard such that the knight visits every square exactly once. This problem has been studied for centuries and has connections to graph theory and algorithm design. There are various algorithms to solve the Knight's Tour problem, including Warnsdorff's rule, backtracking algorithms and genetic algorithms . The problem has applications in optimization, pathfinding, and puzzle-solving. Additionally, variations of the Knight's Tour problem exist, such as on different board sizes or with different constraints. Visualizing and analyzing the solutions to the Knight's Tour problem can provide insights into the nature of the problem and the effectiveness of different algorithms.

1.2 Problem Overview :

In the Knight’s Tour, the knight, a chess piece capable of unique L-shaped moves, must traverse the chessboard, covering each square precisely once. The problem is both combinatorial and algorithmic, requiring strategic exploration to determine a viable sequence of knight moves.

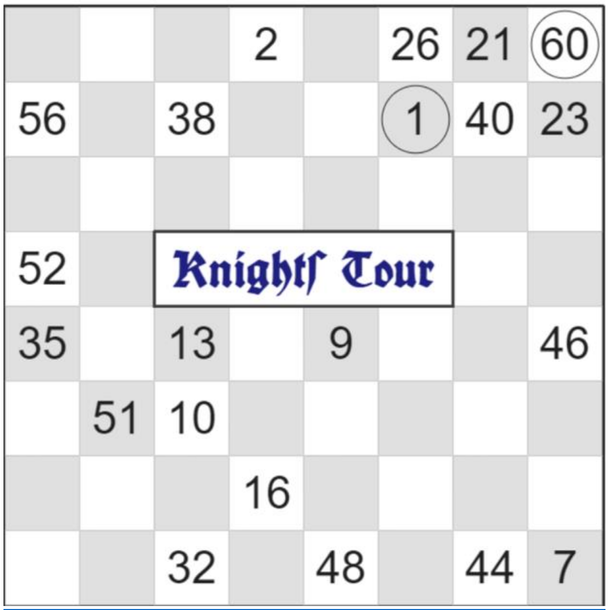
1.3 Approaches Used :

1. Backtracking Search Algorithm: Backtracking involves systematically exploring potential moves while intelligently abandoning paths that lead to dead-ends. This algorithm efficiently navigates the knight through the chessboard, optimizing exploration by discarding unfruitful paths.

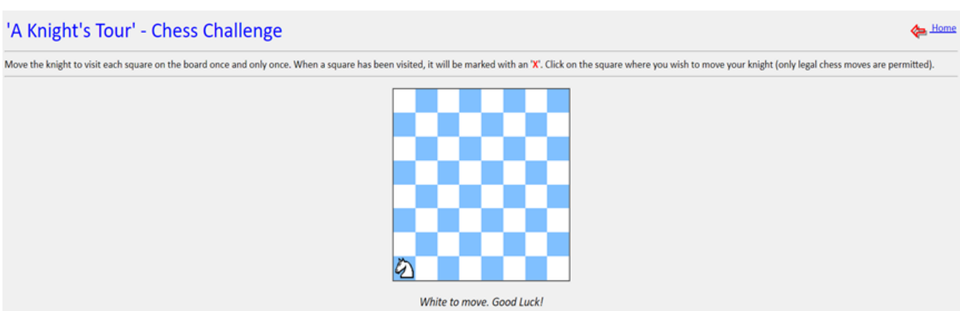
2. Genetic Algorithm: Genetic Algorithms emulate evolutionary processes, employing genetic operations like crossover and mutation to evolve a population of potential solutions. By exploring a diverse set of move sequences and evolving over generations, the algorithm aims to discover effective paths, bringing a stochastic and adaptive approach to solving the Knight’s Tour.

1.4 Similar Applications :

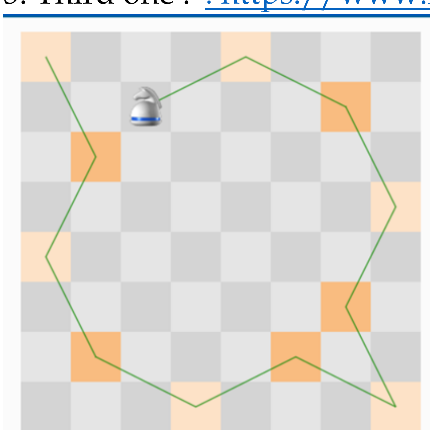
1. First one: https://www.knightstourpuzzle.com



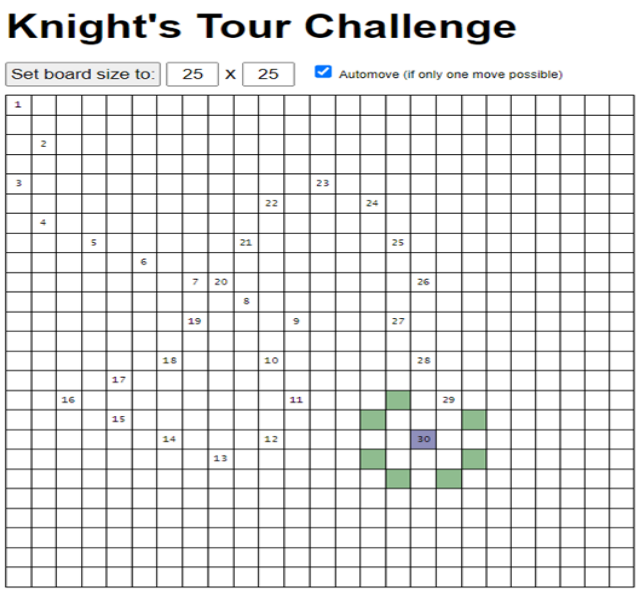
2. Second one: http://www.ee.unb.ca/cgi-bin/tervo/knight.pl?d6e4cc33e0e3b0434cb3



3. Third one : : https://www.flyordie.com/chess/knightstour/



4. Fourth one: https://randompearls.com/reference/tools/knight-solution/



1.5 Literature Review :

We used the papers :

1. First one: Genetic Algorithms with Heuristic Knight's Tour Problem

Link: https://www.researchgate.net/publication/220862449_Genetic_Algorithms_with_Heuristic Knight%27s_Tour_Problem

2. Second one: The Knight's Tour - Evolutionary vs. Depth-First Search

Link: https://ieeexplore.ieee.org/document/1331065

3.Third one: A simple recursive backtracking algorithm for knight's tours puzzle on standard 8×8 chessboard Link: https://ieeexplore.ieee.org/document/8126004

4.Fourth one : Knight's Tours by Ben Hill, Kevin Tostado
Link:http://www.geofhagopain.net/CS007A/assignments/Assignments_F15/ktpaper.pdf

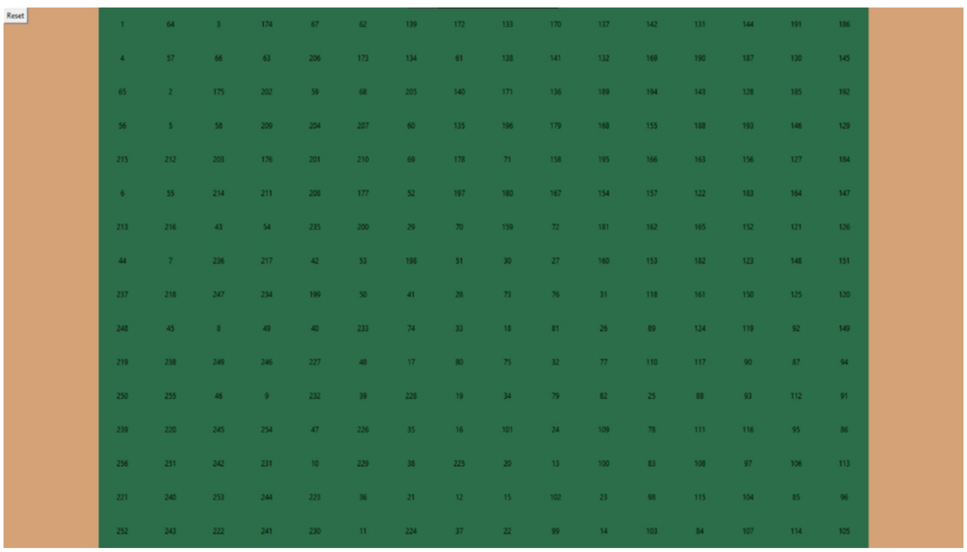
5. Fifth one : A Simple Recursive Backtracking Algorithm for Knight's Tours Puzzle on Standard 8×8 Chessboard , Debajyoti Ghosh Computer Science NIIT University Neemrana, India.

link:https://sites.science.oregonstate.edu/math_reu/proceedings/REU_Proceedings/Proceedings2004/2004Ganzfried.pdf

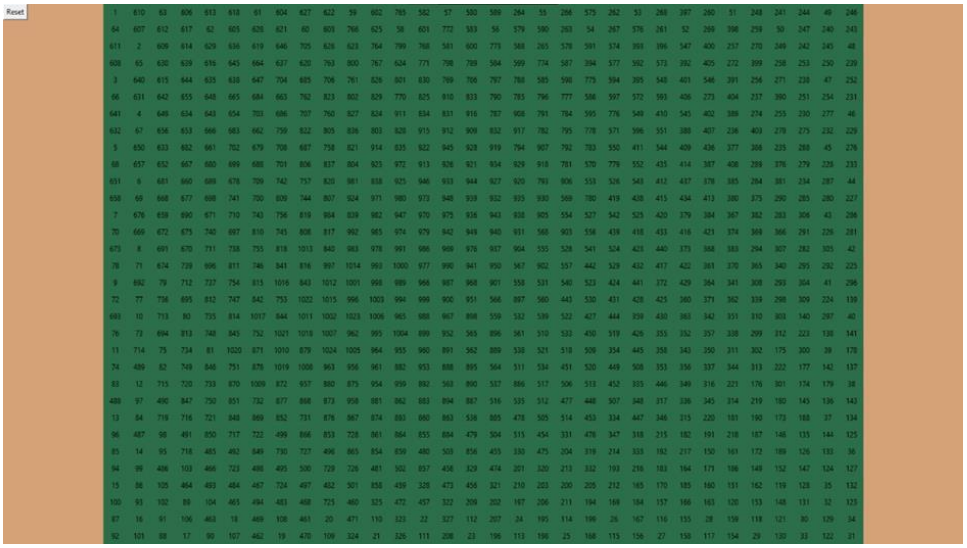
1.6 Proposed Solution :

The proposed solution for the Knight's Tour problem typically involves using algorithmic approaches such as backtracking and genetic algorithm to find a sequence of moves for the knight on the chessboard. These algorithms aim to systematically explore possible paths and choices to ensure that every square is visited exactly once. Additionally, heuristic methods and optimizations can be applied to enhance the efficiency of finding a solution. The chosen approach depends on factors such as the size of the chessboard, desired performance, and specific constraints or variations of the problem.

For example the user input n = 16 and select GA with heuristic :



For example, the user input n =32 and select Backtracking approach:



2. Implementation and applied Algorithms

2.1 Backtracking Approach :

1. Adaptive Heuristics (Warnsdorff's Rule heuristic):

Warnsdorff's Rule is adaptive because it changes its behavior based on the degree of the nodes in the graph.

It prioritizes moves that lead to squares with the fewest possible future moves, which changes as the search progresses.

However, it's important to note that while Warnsdorff's Rule is adaptive, it doesn't adapt in the same way as some other heuristics. For example,

it doesn't adjust its behavior based on the number of unvisited squares or the number of visited squares.

It simply adjusts its behavior based on the degree of the nodes in the graph

This can significantly reduce the time and computational resources required to find a solution, How can this happen ?

ANS: By prioritizing moves that lead to squares with the fewest possible future moves (in the code we get all accessible squares for every state , sort them in ascendingly and pass the next state that has fewer number of accessible squares first),the algorithm can avoid exploring branches of the search space that are unlikely to lead to a solution.

Backtracking Using Warndorff's rule implementation Analysis:

Initialization: The code initializes necessary variables, sets recursion limit, and defines a class KnightsTour to represent the Knight's Tour problem.

Class Methods:

- **`__init__(self, n)`**: The constructor initializes the **KnightsTour** object with the size of the chessboard **n**. It creates an empty chessboard (**self.board**) and defines possible knight moves (**self.moves**) as tuples , **self.solPath** is used to store the path of the solution.

```
print("Printing 1 solution for a starting state that is optimized by Warnsdorff's rule ")
class KnightsTour:
    def __init__(self, n): #initilize board , all squares to -1 as unvisited
        self.n = n
        self.board = [[-1]*n for _ in range(n)]
        self.moves = ((-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1))
        self.solPath = []
```

- **`is_valid_move(self, x, y)`**: Checks if the given coordinates **(x, y)** are within the chessboard boundaries and the corresponding square is unvisited.

```
    def is_valid_move(self, x, y):
        return 0 <= x < self.n and 0 <= y < self.n and self.board[x][y] == -1
```

- **`get_next_moves(self, x, y)`**: Returns a sorted list of tuples , (the list is sorted in ascending order based on **(count)** of possible next moves from a given position **(x, y), after checking it's a valid position.**

```
def get_next_moves(self, x, y):
    next_moves = []
    for move in self.moves:
        next_x, next_y = x + move[0], y + move[1]
        if self.is_valid_move(next_x, next_y):
            count = 0
            for m in self.moves:
                if self.is_valid_move(next_x + m[0], next_y + m[1]):
                    count += 1
            next_moves.append((next_x, next_y, count))
    return sorted(next_moves, key=lambda x: x[2])
```

- **`solve(self, row, col)`**: it called only one time when passing the initial state , it sets all board positions to 0 as visited , start passing the starting row and col to solve_util function , if it returns False means that there is no solution for this start state, else it prints the solution found.

```
def solve(self , row , col):
    self.board[row][col] = 0
    self.solPath.append((row ,col))
    if not self.solve_util(row, col, 1):
        print("No solution exists")
    else:
        self.print_solution()
```

- **`solve_util(self, x, y, move_num)`**: A recursive utility function , first it checks if passed move_num = n*n means that board if full and a solution is found so it returns True , else it attempts to find a solution by exploring possible knight moves of a state by getting next accessible moves of that state , if that path lead to a dead end it backtracks and checks rest of moves , if no of move can lead to a solution It backtracks and set initial state to –1 as unvisited and return False as no solution found

```python
    def solve_util(self, x, y, move_num):
        if move_num == self.n**2: #board is full
            return True

        for next_x, next_y, _ in self.get_next_moves(x, y):
            self.board[next_x][next_y] = move_num
            self.solPath.append((next_x , next_y))
            if self.solve_util(next_x, next_y, move_num+1):
                return True
            self.board[next_x][next_y] = -1
            self.solPath.pop()

        return False
    #end of solve_util function
```

- **`print_solution(self)`**: Prints the final chessboard configuration and the path taken to reach the solution

```python
    def print_solution(self):
        for row in self.board:
            print(row)

        print("/"*40)
        print(self.solPath)
        print("/"*40)
```

2. Randomized heuristic:

This heuristic involves using a depth-first search(backtracking approach) with a randomized order of moves.

The idea is to randomly shuffle the order of the moves for each step, which can help to avoid getting stuck in local optima and explore more of the search space(getting a different solution).

This heuristic can be particularly effective for larger boards where the number of possible moves becomes prohibitively large.

Backtracking Using Randomized heuristic implementation Analysis:

Initialization: The code initializes necessary variables, sets recursion limit, and defines a class KnightsTour to represent the Knight's Tour problem

Class methods:

- **`__init__(self, n)`**: Initializes the board by setting all squares to –1 as unvisited, defines possible moves for the knight, and initializes the solution path.

```python
class KnightsTour:
    def __init__(self, n):
        self.n = n
        self.board = [[-1] * n for _ in range(n)]
        self.moves = ((-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1))
        self.solPath = []
```

- **`is_valid_move`:** Checks if a move is valid within the chessboard boundaries.

```python
    def is_valid_move(self, x, y):
        return 0 <= x < self.n and 0 <= y < self.n and self.board[x][y] == -1
```

- **`get_next_moves:`** Generates the next possible moves for the passed state(x ,y), and prioritizing moves with the fewest possible next moves , sorting all next moves ascendingly after that sets the smallest count(number of accessible moves) from all next_moves of that state to min_next_moves and creates random_moves list that contain all moves from next_moves that has the smallest accessible moves for avoiding dead end paths and call create_random_list function to shuffle that moves for exploring different solutions each run.

```python
def get_next_moves(self, x, y):
    next_moves = []
    for move in self.moves:
        next_x, next_y = x + move[0], y + move[1]
        if self.is_valid_move(next_x, next_y):
            count = 0
            for m in self.moves:
                if self.is_valid_move(next_x + m[0], next_y + m[1]):
                    count += 1
            next_moves.append((next_x, next_y, count))

        # Sort the moves based on the number of next moves in ascending order
        next_moves.sort(key=lambda x: x[2])

    # Use Warnsdorff's rule to prioritize moves with the fewest possible next moves
    if len(next_moves) > 0:
        min_next_moves = next_moves[0][2] #contains only 1 element is the smallest cour
        random_moves = [move for move in next_moves if move[2] == min_next_moves] #take
        #random moves contain all states that has smallest accessible moves (smallest c
        if len(random_moves) > 0:
            random_l = self.Create_random_list(random_moves , len(random_moves))
            return random_l
    # If no moves have the fewest possible next moves, return all moves
    return next_moves
```


- **`Create_random_list`**: Creates a random list of tuples from the available next moves , because it allows the algorithm to explore different sequences of moves that lead to the same square.

```python
def Create_random_list(self , next_moves, num_tuples):
    random_list = []
    for _ in range(num_tuples):
        random_tuple = random.choice(next_moves)#No Duplicates
        random_list.append(random_tuple)

    return random_list
```

- **`Solve(self, x, y)`**: it called only one time when passing the initial state , it sets all board positions to 0 as visited , start passing the starting row and col to solve_util function , if it returns False means that there is no solution for this start state, else it prints the solution found.

```python
def solve(self , x , y):
    start_x, start_y = x, y
    move_num = 0
    self.board[start_x][start_y] = move_num
    self.solPath.append((start_x , start_y))
    if self.solve_util(start_x, start_y, move_num + 1):
        self.print_solution()
    else:
        print("No solution exists")
```

- **`solve_util (self, x, y, move_num)`**: A recursive utility function , first it checks if passed move_num = n*n means that board if full and a solution is found so it returns True , else it attempts to find a solution by exploring possible knight moves of a state by getting next accessible moves of that state , if that path lead to a dead end it backtracks and checks rest of moves , if no of move can lead to a solution It backtracks and set initial state to –1 as unvisited and return False as no solution found.

```python
def solve_util(self, x, y, move_num):
    if move_num == self.n ** 2:
        return True

    for next_x, next_y, _ in self.get_next_moves(x, y):
        self.board[next_x][next_y] = move_num
        self.solPath.append((next_x , next_y))
        if self.solve_util(next_x, next_y, move_num + 1):
            return True
        self.board[next_x][next_y] = -1
        self.solPath.pop()

    return False
```

- **`print_solution(self)`**: Prints the final board configuration and the solution path.

```python
def print_solution(self):
    for row in self.board:
        print(row)

    print("/"*40)
    print(self.solPath)
    print("/"*40)
```

3. Implementation for Getting all possible solutions for a state

Main functions:

- **`printKnightsTourSolution(chessBoard , n ,row ,col , upcomingMove , time_flag)`**:

1. The main function that solve the problem and finds all possible solutions for the initial state Recursively by exploring the Knight's Tour using backtracking.

2. Checking for valid moves and gets the next accessible moves for every state by applying Warndorff's optimization technique as next moves sorted ascendingly and

avoids all paths (moves) that leads to a dead end (for increasing optimization and decreasing time complexity).

3. When a complete tour solution is found , it displays the chessboard configuration , after that it sets the least state(last square visited) to 0 as unvisited to continue exploring its next moves for finding more solutions , this happens to all states for finding all solutions can be found from the start state.

4. Uses multithreading with a timeout to control the search.

```
def printKnightsTourSolutions(chessBoard, n, row, col, upcomingMove, timeout_flag):
    global TotalRuns
    TotalRuns += 1

    # Checking that passed row and column are valid
    if row < 0 or col < 0 or row >= n or col >= n or chessBoard[row][col] != 0:
        return

    if upcomingMove == n * n:
        chessBoard[row][col] = upcomingMove
        displayChessBoard(chessBoard)
        chessBoard[row][col] = 0
        return

    move_x = [2, 1, -1, -2, -2, -1, 1, 2]
    move_y = [1, 2, 2, 1, -1, -2, -2, -1]

    chessBoard[row][col] = upcomingMove

    next_moves = []
    for i in range(8):
        next_x = row + move_x[i]
        next_y = col + move_y[i]
        if next_x >= 0 and next_x < n and next_y >= 0 and next_y < n and chessBoard[next
```

```python
    chessBoard[row][col] = upcomingMove

    next_moves = []
    for i in range(8):
        next_x = row + move_x[i]
        next_y = col + move_y[i]
        if next_x >= 0 and next_x < n and next_y >= 0 and next_y < n and chessBoard[next_x][next_
            count = 0
            for j in range(8):
                if next_x + move_x[j] >= 0 and next_x + move_x[j] < n and next_y + move_y[j] >= 0
                    count += 1
            next_moves.append((next_x, next_y, count))

    # Sort the next moves based on the number of accessible squares(counts) for this move
    next_moves.sort(key=lambda x: x[2])

    for move in next_moves:
        if not timeout_flag.is_set():
            printKnightsTourSolutions(chessBoard, n, move[0], move[1], upcomingMove + 1, timeout_
        else:
            break

    chessBoard[row][col] = 0

# End of printKnightsTourSolutions function
```


- **displayChessBoard (*chessBoard*):**

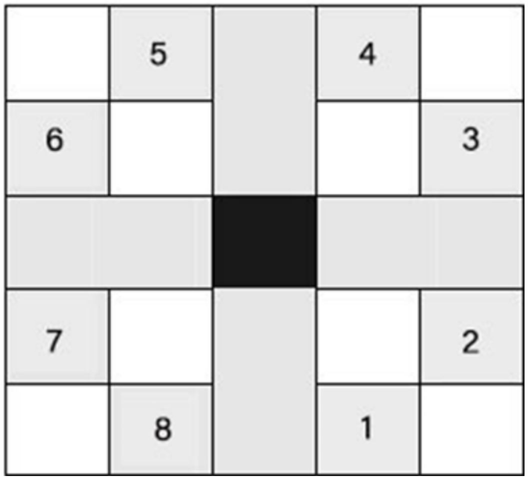
Prints the chessboard configuration as a solution is found, incrementing the number_of_solutions count.

```python
def displayChessBoard(chessBoard):
    print("/////////////////////////////")
    global number_of_solutions
    number_of_solutions += 1
    print("Solution number:", number_of_solutions)
    # Printing every possible solution
    for i in range(len(chessBoard)):
        for j in range(len(chessBoard)):
            print(chessBoard[i][j], end=" ")
        print()

    # End of displayChessBoard function
```

2.2 Genetic Approach implementation

1. Encoding



The encoding for the Knight's Tour problem involves representing the chessboard as a grid and the knight's moves as combinations of horizontal and vertical steps. Using a coordinate system, each square on the chessboard is assigned a unique identifier. The knight's moves can be encoded using numbers from 1 to 8, representing the eight possible directions the knight can move in. This encoding allows for the creation of algorithms that systematically explore possible move sequences. By representing the knight's moves in this way, it becomes possible to track and analyze the paths taken and ensure that every square is visited exactly once, which is the essence of solving the Knight's Tour problem.

2. Crossover and Mutation

In our approach we used a one-point crossover that is selected randomly, the crossover rate is between 85% and 95% with 0.5% mutation rate

```
4 usages
def crossover(self, parent1, parent2):
    if random.random() > self.crossover_rate:
        return random.choice([parent1, parent2])
    crossover_point = random.randint( a: 1, self.N * self.N - 1)
    offspring = parent1[:crossover_point] + parent2[crossover_point:]
    return offspring
```

The "crossover" function performs a genetic algorithm operation where two parent individuals are combined to create offspring. The crossover point is randomly selected, and the resulting offspring inherits genetic information from both parents. This process simulates genetic recombination and introduces diversity into the population.

```
4 usages
def mutation_flip(self, individual):
    for i in range(len(individual)):
        if random.random() < self.mutation_rate:
            individual[i] = random.randint( a: 1,  b: 8)
    return individual
```

The "mutation flip" function introduces random changes to an individual's genetic information. It iterates through the individual's genes and, with a certain probability, replaces each gene with a random value. This operation introduces variability and allows the genetic algorithm to explore a broader search space for potential solutions.

3. Selection

```
2 usages
def select_parents(self, population, use_heuristic=False):
    for i in range(self.population_size):
        self.fitness_values.append(self.evaluate_fitness_matrix(population[i], use_heuristic))
    total_fitness = sum(self.fitness_values)
    probabilities = [fitness / total_fitness for fitness in self.fitness_values]
    parents = random.choices(population, weights=probabilities, k=self.population_size)
    return parents
```

Chromosomes are selected using Russian Roulette wheel selection, where the better chromosomes have a greater chance of being selected. Using this selection method, a maximum of a quarter of the chromosomes (worst chromosomes in generation) are replaced by better chromosomes in the generation. This gives a greater chance for the best chromosomes to participate in the crossover to produce the next generation and reduces the chances of the worst chromosomes to produce the next generation.

4. Extending partial tours:

We tried to solve this problem using pure genetic algorithms, but this approach did not lead us to the proposed solution and take a lot of time to evaluate the fitness of population, so we determined to make some additional modifications on our approach and using Repair and heuristics.
Gordon and Slocum made a similar argument in their paper, evolutionary VS depth-first search.

```
3 usages
def evaluate_fitness_matrix(self, chromosome, use_heuristic=False):
    self.current_x = self.start_position[0]
    self.current_y = self.start_position[1]
    visited_board = [[False for _ in range(self.N)] for _ in range(self.N)]
    visited_board[self.current_x][self.current_y] = True
    num_legal_moves = 0
    for i in range(0, len(chromosome)):
        legal, chromosome[i] = self.move_matrix(chromosome[i], visited_board, use_heuristic)
        if not legal:
            return num_legal_moves
        else:
            num_legal_moves += 1
            visited_board[self.current_x][self.current_y] = True
    return num_legal_moves
```

This function evaluates a fitness matrix based on a given chromosome and optional heuristic. It initializes the current position and a visited board, then iterates through the chromosome to determine legal moves using a move matrix. If a move is illegal, it returns the number of legal moves encountered so far. Otherwise, it updates the number of legal moves and the visited board. Finally, it returns the total number of legal moves made.

The below function implements a genetic algorithm to find a solution. It initializes a population and iterates through a specified number of generations. During each generation, it selects parents based on fitness, performs crossover and mutation to create offspring, and updates the population with the new offspring. It then evaluates the fitness of each chromosome in the population and identifies the best fitness value and corresponding chromosome. If the best fitness reaches the maximum possible value, it prints the fittest chromosome, generation, best fitness, and the decoded best chromosome path. Finally, it returns the fittest chromosome, generation, best fitness, and the decoded best chromosome path when the maximum fitness is achieved.

```
7 usages
def run_genetic_algorithm(self, use_heuristic=False):
    # Run the Genetic Algorithm for finding a solution
    population = self.initialize_population()
    for generation in range(self.max_generations):
        self.fitness_values = []
        parents = self.select_parents(population, use_heuristic)
        offspring = []
        for i in range(0, self.population_size, 2):
            parent1 = parents[i]
            parent2 = parents[i + 1]
            child1 = self.crossover(parent1, parent2)
            child2 = self.crossover(parent2, parent1)
            offspring.extend([self.mutation_flip(child1), self.mutation_flip(child2)])
        population = offspring
        self.fitness_values = []

        for chromosome in population:
            value = self.evaluate_fitness_matrix(chromosome, use_heuristic)
            self.fitness_values.append(value)
        best_fitness = max(self.fitness_values)
        max_index = self.fitness_values.index(best_fitness)
        print(f'Fittest Chromosome: {population[max_index]}')
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")
        if best_fitness == self.N * self.N - 1:

            return (population[max_index], generation + 1, best_fitness, self.decode_best_chromosome(
                population[max_index]))
```

5. Algorithms Used with Genetic

1. With Repair :

For each partial solution (chromosome in a generation), when a knight jumps off the board or returns to a previously visited square, a modification is applied. We try to replace the move (gene) that represents the illegal move with another move that. allows the knight to proceed.

Before every move, we examine the squares that can be reached with legal moves from the current square. Then we try each possible move in an ascending order, then the night moves according to the first legal move of the possible move We repeat for each chromosome until a substitution cannot be made, and the evaluation of that chromosome then ends.

for example, the tour contained an illegal move when square 50 was visited twice.

3 – 5 – 6 – 1

(50 – 44 – 27 – 33 - 50)

Therefore, the right-most move (1) would be replaced with another legal move. According to our approach must find the legal moves from the current square (33 in this case) from the other seven possible moves. Which the legal moves are the squares, 43 (through move 1) and 18 (through move number 4) and choose the first legal move we found. Therefore, the move to square 18 is chosen (through move 4) and the gene (move 1) is substituted by move 4.

3 - 5 - 6 - 4

Each chromosome is evaluated when an illegal move is encountered in the same manner until the knight can no longer make a legal move.

2. With Heuristic:

We used the same heuristic approach as used in the first paper we mentioned above, but with different encoding.

For each partial solution (chromosome in a generation), when a knight jumps off the board or returns to a previously visited square, a modification with heuristic is applied. We try to replace the move that represents the illegal move with another move that allows the knight to proceed.

Before every move, we examine the squares that can be reached with legal moves from the current square. Then for each of those possible next squares, we count the number of legal moves at that square. The knight then moves to the square with the lowest number of new choices. We repeat for each chromosome until a substitution cannot be made, and the evaluation of that chromosome then ends. In the earlier example, the tour contained an illegal move when square 50 was visited twice

3 – 5 – 6 – 1 …

(50 – 44 – 27 – 33 - 50)

Therefore, the right-most move (1) would be replaced with another legal move. According to our approach heuristic must find the legal moves from the current square (33 in this case). Which are the squares, 43 (through move 1) and 18 (through move number 4) and choose the square with the lowest number of legal moves. Square 43 has 7 legal moves, while square 18 has 5 legal moves. Therefore, the move to square 18 is

chosen (move 4) and the (move 1) is substituted by the (move 4). (Moves codes were shown in
3 - 5 - 6 – 4 …
Each chromosome is evaluated when an illegal move is encountered in the same manner until the knight can no longer make a legal move.

3. Experiments & Results

3.1 For backtracking approach :

1. Samples of Applying Warndorff's rule

N = 5

```
Printing 1 solution for a starting state that is optimized by Warnsdorff's rule
Enter the size of the chessboard: 5
Enter the starting row: 0
Enter the starting column: 0
[0, 19, 8, 13, 2]
[9, 14, 1, 18, 23]
[15, 10, 5, 24, 17]
[6, 21, 16, 11, 4]
////////////////////////////////////////
[(0, 0), (1, 2), (0, 4), (2, 3), (4, 4), (3, 2), (4, 0), (2, 1), (0, 2), (1, 0), (3, 1), (4, 3), (2, 4), (0, 3), (1, 1), (3, 0), (4, 2), (3, 4), (1, 3), (0, 1), (2, 0), (4, 1), (2, 2), (1, 4), (3, 3)]
////////////////////////////////////////
total time :  0.061742305755615234
PS C:\Users\LENOVO12021\OneDrive\Documents\pythonTests\knightTour_Backtracking> & C:/Users/LENOVO12021/AppData/Loca
```

N = 8

```
l/Programs/Python/Python310/python.exe "c:/Users/LENOVO12021/OneDrive/Documents/pythonTests/knightTour_Backtracking/
knightsTourBackTracking_Warnsdorff's Rule heuristic.py"
Printing 1 solution for a starting state that is optimized by Warnsdorff's rule
Enter the size of the chessboard: 8
Enter the starting row: 2
Enter the starting column: 2
[22, 1, 18, 33, 40, 37, 16, 35]
[19, 32, 21, 50, 17, 34, 43, 38]
[2, 23, 0, 41, 58, 39, 36, 15]
[31, 20, 49, 28, 51, 42, 57, 44]
[24, 3, 30, 59, 48, 61, 14, 55]
[9, 6, 27, 52, 29, 56, 45, 62]
[4, 25, 8, 11, 60, 47, 54, 13]
[7, 10, 5, 26, 53, 12, 63, 46]
////////////////////////////////////////
[(2, 2), (0, 1), (2, 0), (4, 1), (6, 0), (7, 2), (5, 1), (7, 0), (6, 2), (5, 0), (7, 1), (6, 3), (7, 5), (6, 7), (4, 6), (2, 7), (0, 6), (1, 4), (0, 2), (1, 0), (3, 1), (1, 2), (0, 0), (2, 1), (4, 0), (6, 1), (7, 3), (5, 2), (3, 3), (5, 4), (4, 2), (3, 0), (1, 1), (0, 3), (1, 5), (0, 7), (2, 6), (0, 5), (1, 7), (2, 5), (0, 4), (2, 3), (3, 5), (1, 6), (3, 7), (5, 6), (7, 7), (6, 5), (4, 4), (3, 2), (1, 3), (3, 4), (5, 3), (7, 4), (6, 6), (4, 7), (5, 5), (3, 6), (2, 4), (4, 3), (6, 4), (4, 5), (5, 7), (7, 6)]
////////////////////////////////////////
total time :  0.023734569549560547
PS C:\Users\LENOVO12021\OneDrive\Documents\pythonTests\knightTour_Backtracking>
```

N = 16

```
knightsTourBackTracking_Warnsdorff's Rule heuristic.py"
Printing 1 solution for a starting state that is optimized by Warnsdorff's rule
Enter the size of the chessboard: 16
Enter the starting row: 0
Enter the starting column: 0
[0, 31, 66, 75, 2, 33, 62, 79, 4, 35, 58, 51, 6, 37, 40, 49]
[67, 74, 1, 32, 77, 80, 3, 34, 61, 86, 5, 36, 57, 50, 7, 38]
[30, 65, 76, 81, 108, 63, 78, 85, 100, 59, 92, 87, 52, 39, 48, 41]
[73, 68, 109, 64, 83, 106, 113, 60, 91, 96, 101, 56, 93, 88, 53, 8]
[110, 29, 82, 107, 112, 117, 84, 105, 114, 99, 90, 95, 102, 55, 42, 47]
[69, 72, 111, 118, 145, 142, 115, 160, 97, 104, 163, 186, 89, 94, 9, 54]
[28, 119, 70, 141, 116, 159, 146, 143, 162, 185, 98, 103, 164, 187, 46, 43]
[71, 140, 121, 156, 147, 144, 161, 184, 191, 176, 239, 188, 199, 44, 165, 10]
[120, 27, 148, 179, 158, 155, 192, 177, 240, 189, 198, 175, 244, 167, 200, 45]
[139, 122, 157, 154, 193, 178, 183, 190, 197, 250, 243, 238, 203, 174, 11, 166]
[26, 149, 180, 151, 182, 195, 216, 249, 226, 241, 204, 251, 172, 245, 168, 201]
[123, 138, 153, 194, 215, 220, 225, 196, 253, 248, 237, 242, 231, 202, 173, 12]
[136, 25, 150, 181, 152, 217, 254, 221, 236, 227, 252, 205, 246, 171, 232, 169]
[127, 124, 137, 214, 219, 212, 131, 224, 255, 222, 247, 230, 233, 206, 13, 16]
[24, 135, 126, 129, 22, 133, 218, 211, 20, 235, 228, 209, 18, 15, 170, 207]
[125, 128, 23, 134, 213, 130, 21, 132, 223, 210, 19, 234, 229, 208, 17, 14]
////////////////////////////////////////
[(0, 0), (1, 2), (0, 4), (1, 6), (0, 8), (1, 10), (0, 12), (1, 14), (3, 15), (5, 14),
```

```
////////////////////////////////////////
[(0, 0), (1, 2), (0, 4), (1, 6), (0, 8), (1, 10), (0, 12), (1, 14), (3, 15), (5, 14), (7, 15), (9, 14), (11, 15), (13
, 14), (15, 15), (14, 13), (13, 15), (15, 14), (14, 12), (15, 10), (14, 8), (15, 6), (14, 4), (15, 2), (14, 0), (12,
1), (10, 0), (8, 1), (6, 0), (4, 1), (2, 0), (0, 1), (1, 3), (0, 5), (1, 7), (0, 9), (1, 11), (0, 13), (1, 15), (2, 1
3), (0, 14), (2, 15), (4, 14), (6, 15), (7, 13), (8, 15), (6, 14), (4, 15), (2, 14), (0, 15), (1, 13), (0, 11), (2, 1
2), (3, 14), (5, 15), (4, 13), (3, 11), (1, 12), (0, 10), (2, 9), (3, 7), (1, 8), (0, 6), (2, 5), (3, 3), (2, 1), (0,
 2), (1, 0), (3, 1), (5, 0), (6, 2), (7, 0), (5, 1), (3, 0), (1, 1), (0, 3), (2, 2), (1, 4), (2, 6), (0, 7), (1, 5),
(2, 3), (4, 2), (3, 4), (4, 6), (2, 7), (1, 9), (2, 11), (3, 13), (5, 12), (4, 10), (3, 8), (2, 10), (3, 12), (5, 13)
, (4, 11), (3, 9), (5, 8), (6, 10), (4, 9), (2, 8), (3, 10), (4, 12), (6, 11), (5, 9), (4, 7), (3, 5), (4, 3), (2, 4)
, (3, 2), (4, 0), (5, 2), (4, 4), (3, 6), (4, 8), (5, 6), (6, 4), (4, 5), (5, 3), (6, 1), (8, 0), (7, 2), (9, 1), (11
, 0), (13, 1), (15, 0), (14, 2), (13, 0), (15, 1), (14, 3), (15, 5), (13, 6), (15, 7), (14, 5), (15, 3), (14, 1), (12
, 0), (13, 2), (11, 1), (9, 0), (7, 1), (6, 3), (5, 5), (6, 7), (7, 5), (5, 4), (6, 6), (7, 4), (8, 2), (10, 1), (12,
 2), (10, 3), (12, 4), (11, 2), (9, 3), (8, 5), (7, 3), (9, 2), (8, 4), (6, 5), (5, 7), (7, 6), (6, 8), (5, 10), (6,
12), (7, 14), (9, 15), (8, 13), (10, 14), (12, 15), (14, 14), (12, 13), (10, 12), (11, 14), (9, 13), (8, 11), (7, 9),
 (8, 7), (9, 5), (8, 3), (10, 2), (12, 3), (10, 4), (9, 6), (7, 7), (6, 9), (5, 11), (6, 13), (7, 11), (8, 9), (9, 7)
, (7, 8), (8, 6), (9, 4), (11, 3), (10, 5), (11, 7), (9, 8), (8, 10), (7, 12), (8, 14), (10, 15), (11, 13), (9, 12),
(10, 10), (12, 11), (13, 13), (14, 15), (15, 13), (14, 11), (15, 9), (14, 7), (13, 5), (15, 4), (13, 3), (11, 4), (10
, 6), (12, 5), (14, 6), (13, 4), (11, 5), (12, 7), (13, 9), (15, 8), (13, 7), (11, 6), (10, 8), (12, 9), (14, 10), (1
5, 12), (13, 11), (11, 12), (12, 14), (13, 12), (15, 11), (14, 9), (12, 8), (11, 10), (9, 11), (7, 10), (8, 8), (10,
9), (11, 11), (9, 10), (8, 12), (10, 13), (12, 12), (13, 10), (11, 9), (10, 7), (9, 9), (10, 11), (12, 10), (11, 8),
(12, 6), (13, 8)]
////////////////////////////////////////
total time :  0.04458260536193848
```

As illustrated in results , by applying Warndorff's rule it makes the code very efficient especially for large n boards like n = 16 , it takes 0.044 seconds to get the

optimial solution that decreasing search space and concentrating only on paths (accessible moves) that leads to a solution in shortest time , this achieved by ignoring all paths that leads to a dead end or path that has large accessible moves that takes many time to find optimal solution.

1. Samples of Applying Randomized heuristics (each run it gets another solution in no time):

N = 8 , start state (1 , 1)

The 1st run :

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS
N_Backtracking_RandomizedHeuristic.py
Printing 1 solution from same/different starting states (printing the different solution every run) using Randomized
heuristic.
Enter the size of the chessboard: 8
Enter the starting row: 1
Enter the starting column: 1
[18, 15, 32, 1, 20, 5, 40, 3]
[31, 0, 19, 16, 53, 2, 21, 6]
[14, 17, 52, 33, 22, 39, 4, 41]
[47, 30, 23, 54, 59, 56, 7, 38]
[24, 13, 46, 51, 34, 63, 42, 57]
[29, 48, 27, 60, 55, 58, 37, 8]
[12, 25, 50, 45, 10, 35, 62, 43]
[49, 28, 11, 26, 61, 44, 9, 36]
////////////////////////////////////////
[(1, 1), (0, 3), (1, 5), (0, 7), (2, 6), (0, 5), (1, 7), (3, 6), (5, 7), (7, 6), (6, 4), (7, 2), (6, 0), (4, 1), (2,
0), (0, 1), (1, 3), (2, 1), (0, 0), (1, 2), (0, 4), (1, 6), (2, 4), (3, 2), (4, 0), (6, 1), (7, 3), (5, 2), (7, 1), (
5, 0), (3, 1), (1, 0), (0, 2), (2, 3), (4, 4), (6, 5), (7, 7), (5, 6), (3, 7), (2, 5), (0, 6), (2, 7), (4, 6), (6, 7)
, (7, 5), (6, 3), (4, 2), (3, 0), (5, 1), (7, 0), (6, 2), (4, 3), (2, 2), (1, 4), (3, 3), (5, 4), (3, 5), (4, 7), (5,
 5), (3, 4), (5, 3), (7, 4), (6, 6), (4, 5)]
////////////////////////////////////////
total time :  0.10576248168945312
PS C:\Users\LEVNOVO12021\OneDrive\Documents\pythonTests\knightTour_Backtracking> 
```

The 2nd run :

```
N_Backtracking_RandomizedHeuristic.py
Printing 1 solution from same/different starting states (printing the different solution every run) using Randomized
heuristic.
Enter the size of the chessboard: 8
Enter the starting row: 1
Enter the starting column: 1
[28, 25, 14, 47, 18, 23, 12, 49]
[15, 0, 27, 24, 13, 48, 19, 22]
[26, 29, 52, 17, 46, 21, 50, 11]
[1, 16, 33, 62, 51, 58, 45, 20]
[32, 53, 30, 57, 34, 61, 10, 59]
[5, 2, 39, 54, 63, 56, 35, 44]
[40, 31, 4, 7, 42, 37, 60, 9]
[3, 6, 41, 38, 55, 8, 43, 36]
////////////////////////////////////////
[(1, 1), (3, 0), (5, 1), (7, 0), (6, 2), (5, 0), (7, 1), (6, 3), (7, 5), (6, 7), (4, 6), (2, 7), (0, 6), (1, 4), (0,
2), (1, 0), (3, 1), (2, 3), (0, 4), (1, 6), (3, 7), (2, 5), (1, 7), (0, 5), (1, 3), (0, 1), (2, 0), (1, 2), (0, 0), (
2, 1), (4, 2), (6, 1), (4, 0), (3, 2), (4, 4), (5, 6), (7, 7), (6, 5), (7, 3), (5, 2), (6, 0), (7, 2), (6, 4), (7, 6)
, (5, 7), (3, 6), (2, 4), (0, 3), (1, 5), (0, 7), (2, 6), (3, 4), (2, 2), (4, 1), (5, 3), (7, 4), (5, 5), (4, 3), (3,
 5), (4, 7), (6, 6), (4, 5), (3, 3), (5, 4)]
////////////////////////////////////////
total time :  0.05066847801208496
PS C:\Users\LEVNOVO12021\OneDrive\Documents\pythonTests\knightTour_Backtracking>
```

AS in the results , it gets two different solutions from the same start state , the both solutions found in no time because the randomized heuristic used is optimized by Warndorff's rule to decrease search space that leads to decreasing time and space complexity .

Now you will see the big difference in time taken for Randomized heuristic with and without optimized code :

With using Warndorrf's rule (optimization technique for prioritizing next moves and avoid dead end paths):

```
cuments/pythonTests/.venv/Scripts/python.exe
_Backtracking_RandomizedHeuristic.py
Printing 1 solution from same/different star
using Randomized heuristic.
Enter the size of the chessboard: 5
Enter the starting row: 1
Enter the starting column: 1
[20, 5, 16, 11, 18]
[15, 0, 19, 4, 9]
[6, 21, 10, 17, 12]
[1, 14, 23, 8, 3]
[22, 7, 2, 13, 24]
////////////////////////////////////////
[(1, 1), (3, 0), (4, 2), (3, 4), (1, 3), (0,
 4), (4, 3), (3, 1), (1, 0), (0, 2), (2, 3),
////////////////////////////////////////
total time :  0.04704928398132324
(.venv) PS C:\Users\LEVNOVO12021\OneDrive\D
```

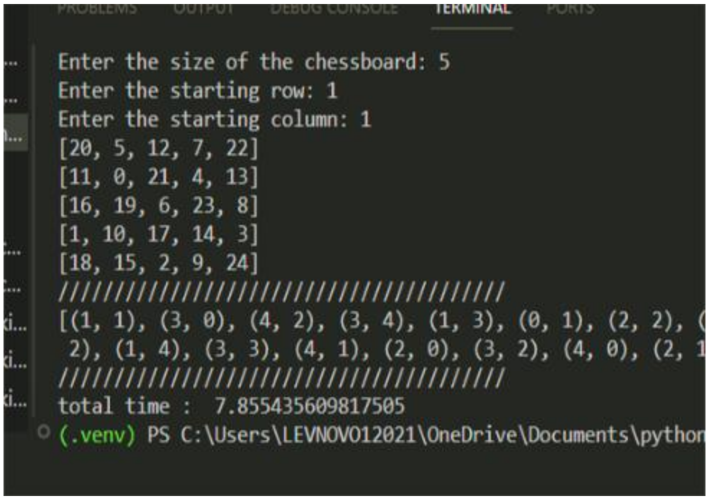
```
cuments/pythonTests/.venv/Scripts/python.exe c
_Backtracking_RandomizedHeuristic.py
Printing 1 solution from same/different starti
using Randomized heuristic.
Enter the size of the chessboard: 5
Enter the starting row: 1
Enter the starting column: 1
[24, 15, 6, 1, 18]
[5, 0, 17, 14, 7]
[16, 23, 10, 19, 2]
[11, 4, 21, 8, 13]
[22, 9, 12, 3, 20]
////////////////////////////////////////
[(1, 1), (0, 3), (2, 4), (4, 3), (3, 1), (1, 0
 2), (3, 4), (1, 3), (0, 1), (2, 0), (1, 2), (
////////////////////////////////////////
total time :  0.015612363815307617
(.venv) PS C:\Users\LEVNOVO12021\OneDrive\Docu
```

In both runs it gets two different solutions in no time

Without using Warndorff's rule or any optimization technique :

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    POR

using Randomized heuristic.
Enter the size of the chessboard: 5
Enter the starting row: 1
Enter the starting column: 1
[22, 5, 12, 7, 20]
[11, 0, 21, 4, 13]
[16, 23, 6, 19, 8]
[1, 10, 17, 14, 3]
[24, 15, 2, 9, 18]
////////////////////////////////////////
[(1, 1), (3, 0), (4, 2), (3, 4), (1, 3), (0, 1)
 2), (1, 4), (3, 3), (4, 1), (2, 0), (3, 2), (4
////////////////////////////////////////
total time :  5.203878402709961
(.venv) PS C:\Users\LEVNOVO12021\OneDrive\Docur
```

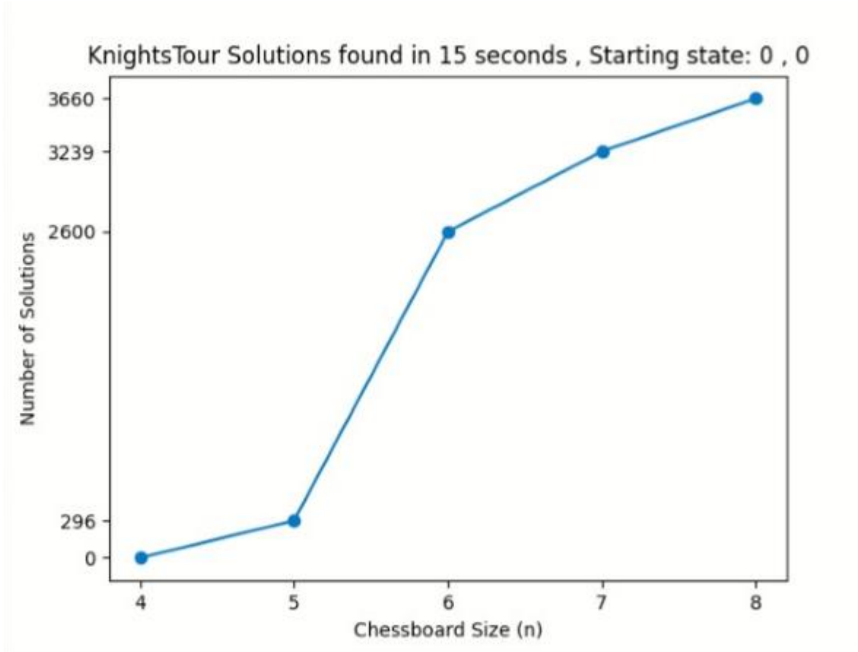


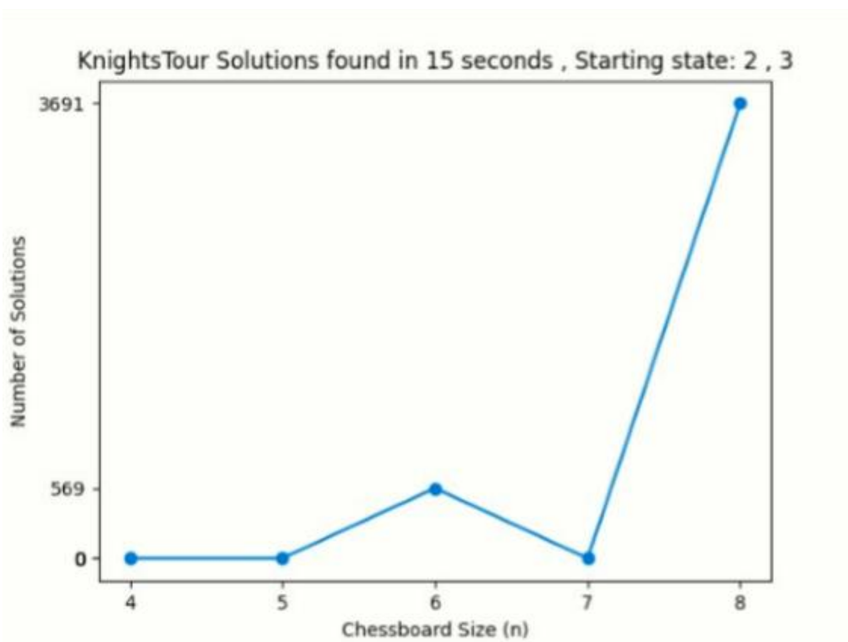
3. Plots analysis Using backtracking:

As we want to test the efficiency of algorithms used with backtracking based on (time takes to get the solution for different n sizes and different states and the number of tours (solutions) can get , number of runs for different n sizes and start states too)

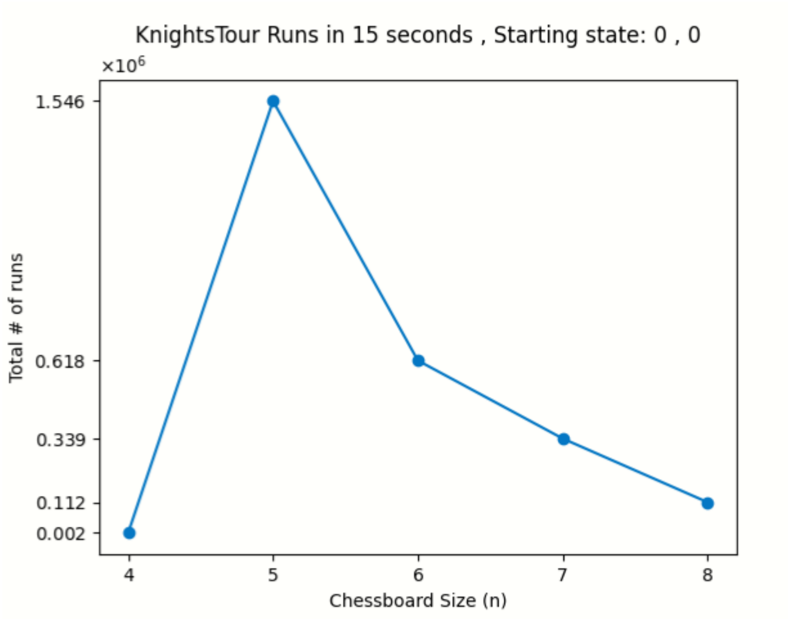
1. **Plot Analysis of finding all possible solutions for a specific start state for all n = {4 , 5 ,6 ,7 ,8}**

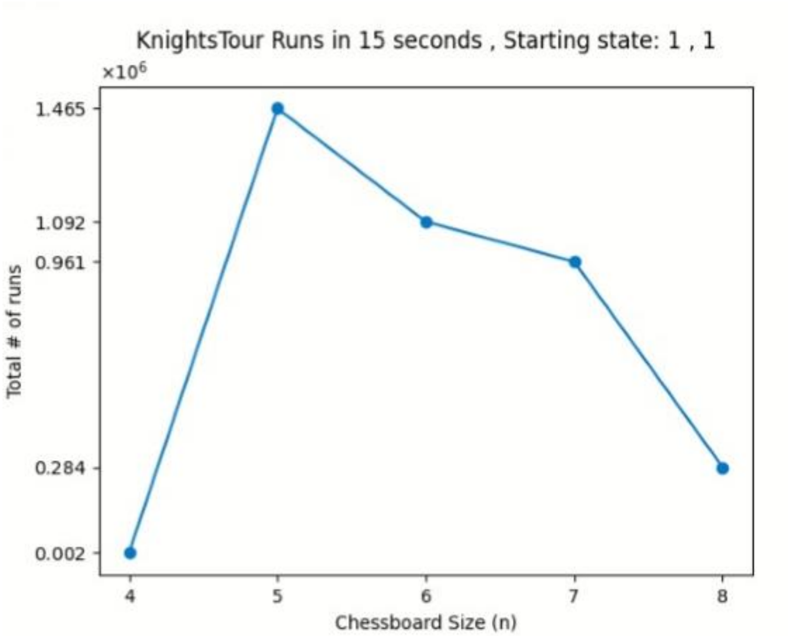
As the board size increases, the complexity of the search space also increases exponentially. The number of possible moves and paths to explore grows significantly with board size ,
Exploring more solutions increases.





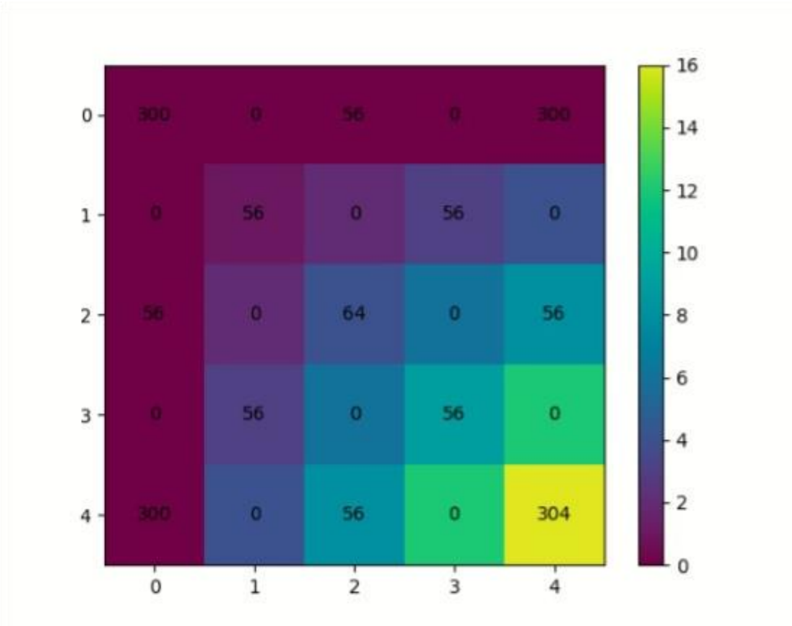
2. Analysis for total number of runs for a specific start state for all n



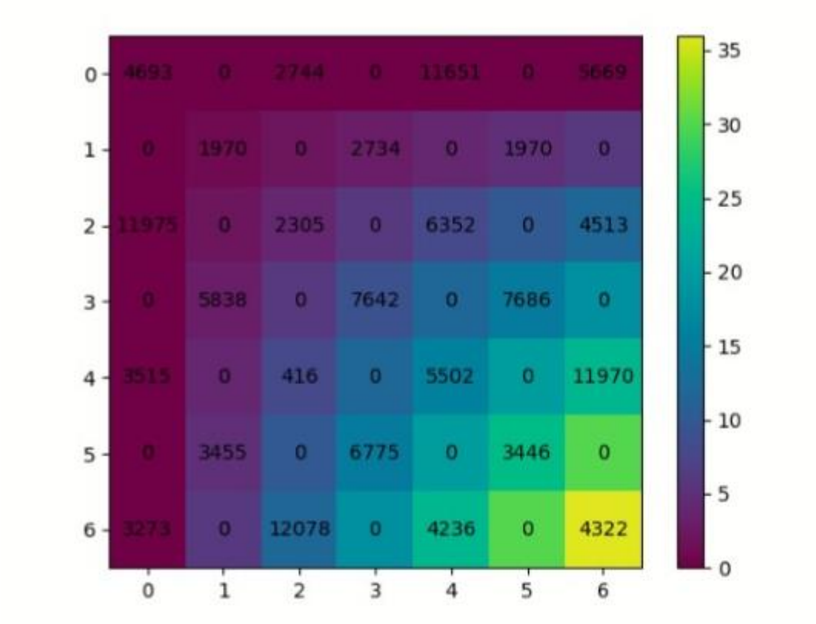


3. Analysis for possible number of solutions can get from (n) in the all states for a time period (15seconds)

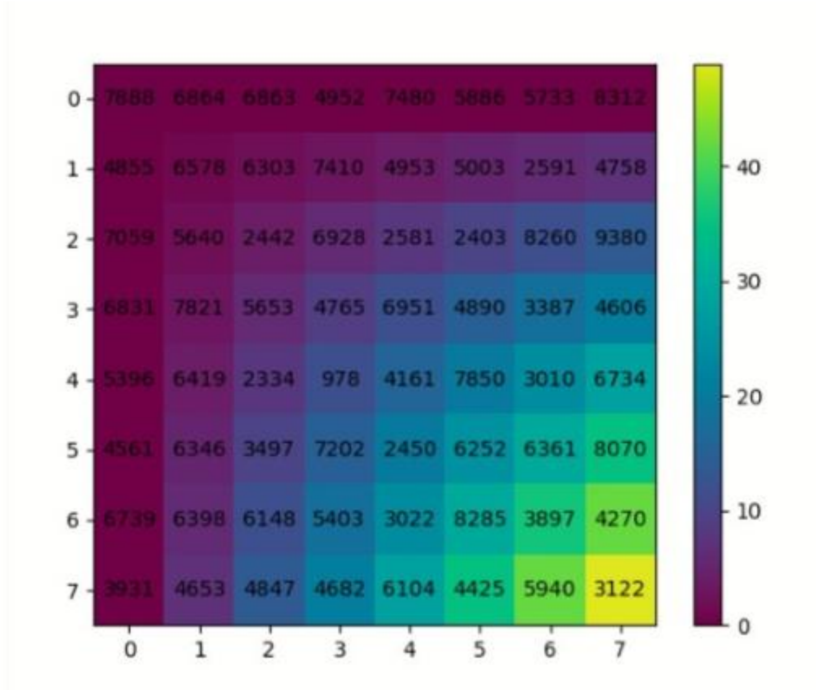
N =5



N = 7

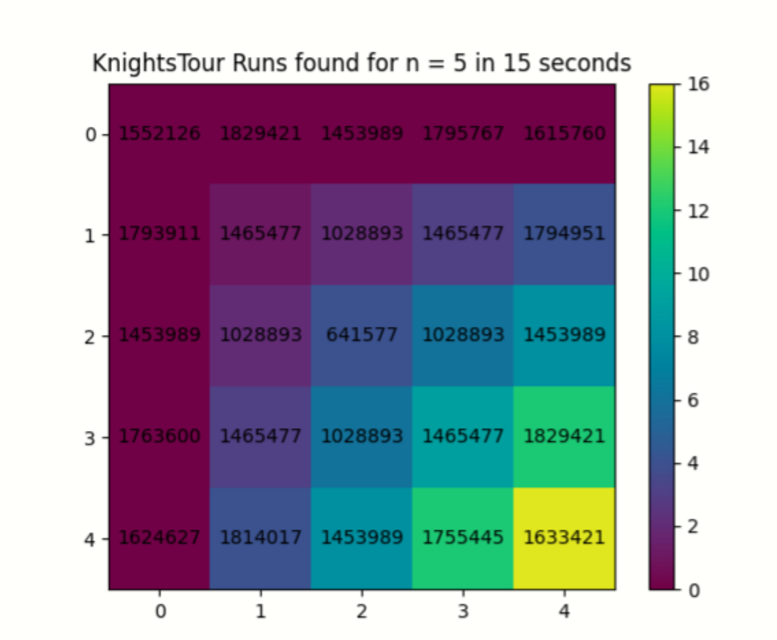


N = 8



4. **Analysis for possible number of Runs can get from (n) in the all states for a time period (15seconds)**

N = 5



From the two previous analysis of n = 5 for both plots (number of solutions and number of runs)you will notice that at state (3,4) the number of solutions = 0 , however the number of runs of the same state is the maximum number compared with all other states , The largest number of runs indicates that the backtracking algorithm attempted various paths or combinations, exploring different possibilities to find a solution, but none of these attempts led to a successful Knight's Tour from the given starting state on a 5x5 chessboard.

4 . Complexity of backtracking approach :

Backtracking is a general algorithmic technique that explores all potential solutions to a computational problem by trying out different choices and undoing those choices when they are found not to lead to a solution.

The time complexity of solving the Knight's Tour problem using a backtracking algorithm is typically high because the search space is quite large.

Let n represent the size of the chessboard (assuming it is an $n \times n$ board). The Knight's Tour problem has an exponential time complexity of approximately $O((n^2)!)$ due to the large number of possible move sequences that need to be explored.

But with using optimzation techniques as Warndorff's rule that minmized the search space , the complexity become (k^(n*n)) , as k is the number of accessible moves , And there is k number of moves for every state (square) and to find solution must visit all n*n squares , for that **O(k^(n*n))** is the complexity after optimzization.

3.2 For Genetic Approach :

For testing purposes, the GA parameters were set to common literature values, with a population size of 50, a crossover percentage ranging from 85% to 95%, and a low mutation rate of 0.5%. The GA was run five times for each square, totaling 320 runs for the full board, evaluating 800,000 chromosomes. Results were recorded for each run, and averages were calculated for each square based on the five runs, as well as for all squares based on the 320 runs.

The test results included the total number of distinct tours found and their ratio to the overall tested chromosomes, the maximum and minimum number of tours found in 5 runs of the squares, the average number of tours found per square.

1. GA with Repair

```
The run time was: 127.61244344711304 Sec
[(22, 10), (9, 2), (10, 15), (20, 22), (14, 7), (10, 3), (5, 4), (19, 8)]
[(6, 2), (32, 14), (7, 4), (7, 1), (11, 2), (0, 0), (6, 1), (11, 40)]
[(0, 0), (5, 14), (30, 16), (0, 0), (5, 10), (7, 5), (3, 1), (5, 26)]
[(7, 29), (10, 1), (15, 2), (31, 5), (6, 1), (0, 0), (0, 0), (23, 20)]
[(26, 40), (16, 6), (1, 7), (5, 1), (12, 4), (3, 2), (8, 10), (27, 26)]
[(11, 4), (9, 4), (0, 0), (0, 0), (1, 4), (21, 8), (8, 3), (4, 2)]
[(5, 3), (9, 1), (0, 0), (2, 3), (0, 4), (5, 4), (5, 5), (1, 2)]
[(20, 5), (6, 1), (4, 18), (0, 0), (13, 30), (6, 2), (22, 9), (12, 2)]
```

presents sample results of the GA with repair. Each tuple represents a square, The first value in each tuple represents the average number of generations needed to produce the first tour from that square (average for five runs), while the second value represents the total number of distinct solutions found in the five runs for that square.

```
The number_of_tours: 475
The number_of_evaluation: 800000
Hit Ratio (tours found/evaluated): 0.059375
average # tours found per square: 7
The maximum square: (1, 7) , the number of tours: 40
The minimum square: (1, 5) , the number of tours: 0
```

2 . GA with heuristic

```
The run time was: 224.31919145584106 Sec
[(1, 445), (1, 489), (1, 286), (4, 316), (2, 416), (1, 420), (2, 163), (2, 446)]
[(3, 237), (1, 116), (8, 167), (6, 197), (3, 191), (3, 89), (5, 174), (2, 163)]
[(1, 291), (2, 165), (1, 377), (4, 589), (6, 198), (3, 208), (9, 175), (1, 395)]
[(1, 158), (2, 197), (2, 303), (2, 208), (2, 270), (3, 354), (2, 244), (2, 381)]
[(4, 341), (1, 260), (6, 235), (1, 404), (1, 221), (3, 224), (8, 181), (2, 241)]
[(2, 347), (2, 179), (1, 316), (1, 259), (5, 204), (6, 134), (3, 285), (3, 191)]
[(2, 282), (8, 162), (5, 119), (3, 244), (1, 238), (4, 261), (7, 273), (2, 247)]
[(1, 420), (1, 366), (1, 276), (1, 343), (1, 282), (2, 431), (3, 197), (1, 783)]
```

presents sample results of the GA with heuristic. Each tuple represents a square, The first value in each tuple represents the average number of generations needed to produce the first tour from that square (average for five runs), while the second value represents the total number of distinct solutions found in the five runs for that square.

```
The number_of_tours: 17774
The number_of_evaluation: 800000
Hit Ratio (tours found/evaluated): 2.22175
average # tours found per square: 277
The maximum square: (7, 7) , the number of tours: 783
The minimum square: (1, 5) , the number of tours: 89
```

3. Analysis using N = 16 (Heuristic)

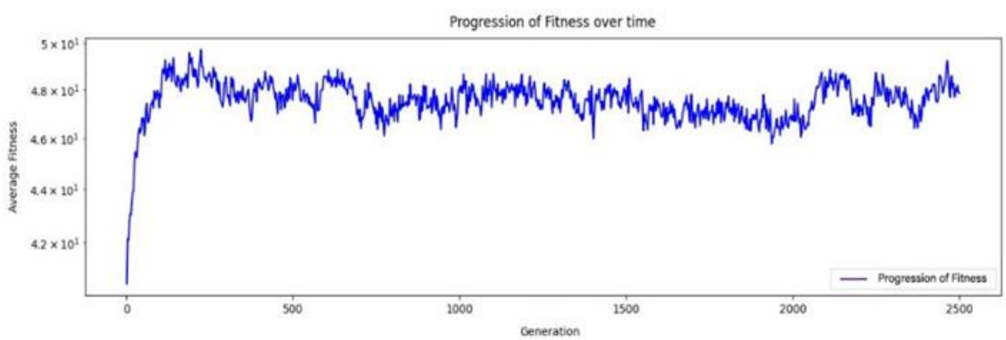
```
The run time was: 2744.0754420757294 Sec
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (5, 3), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (7, 1), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (9, 1), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(5, 1), (0, 0), (0, 0), (0, 0), (0, 0), (5, 1), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (2, 1), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
```

```
The number_of_tours: 8
The number_of_evaluation: 3200000
Hit Ratio (tours found/evaluated): 0.00025
average # tours found per square: 0
The maximum square: (1, 7) , the number of tours: 3
The minimum square: (0, 0) , the number of tours: 0
```

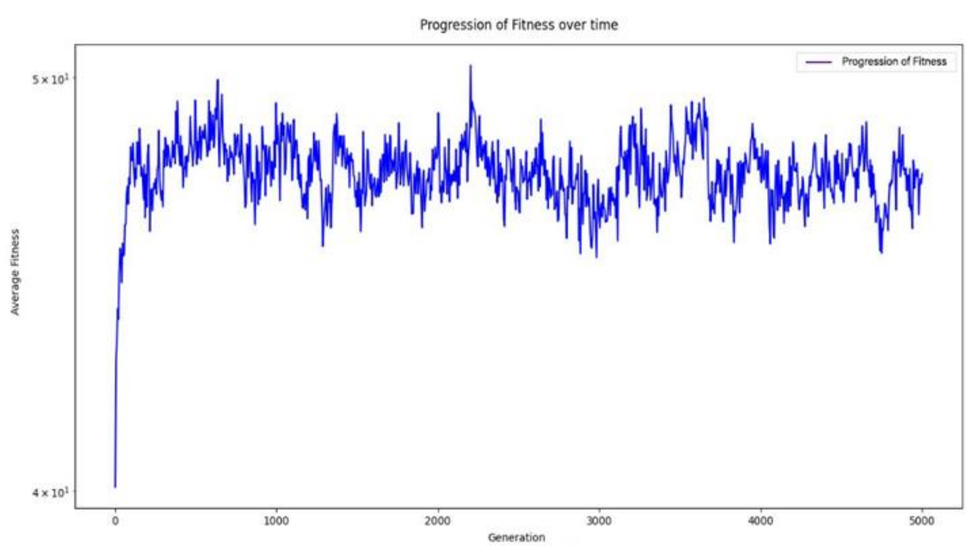
4. Plots for analysis of Genetic algorithm

We need to study the relationship between the fitness values and the number of generation over time , so we made the following plots :

GA with heuristic and N =8 and Generation = 2500 , population size = 1000



GA with heuristic and N =8 and Generation = 5000



Then we assumed that:

The graph displays progressive fitness over time, with a blue line showing the fitness level. The line starts at a lower point, indicating a lower fitness level, and gradually increases over time, indicating an improvement in fitness.

The graph shows a progressive increase in fitness over generations. The convergence of the plot indicates that the organisms are reaching a common optimal point due to natural selection. This suggests that the organisms with higher fitness levels are more likely to survive and reproduce, leading to an overall improvement in fitness levels.

5. Complexity of Genetic approach:

Genetic Algorithms are not chaotic; they are stochastic. The complexity is influenced by the genetic operators, their implementation (which can significantly impact overall complexity), the representation of individuals and the population, and, of course, the fitness function. Using common choices such as point mutation, one-point crossover, and roulette wheel selection, the complexity of a Genetic Algorithm can be expressed as O(g(nm + nm + n)), where g is the number of generations, n is the population size, and m is the size of the individuals. Therefore, the complexity is approximately O(gnm). This analysis does not account for the fitness function, which varies based on the specific application.

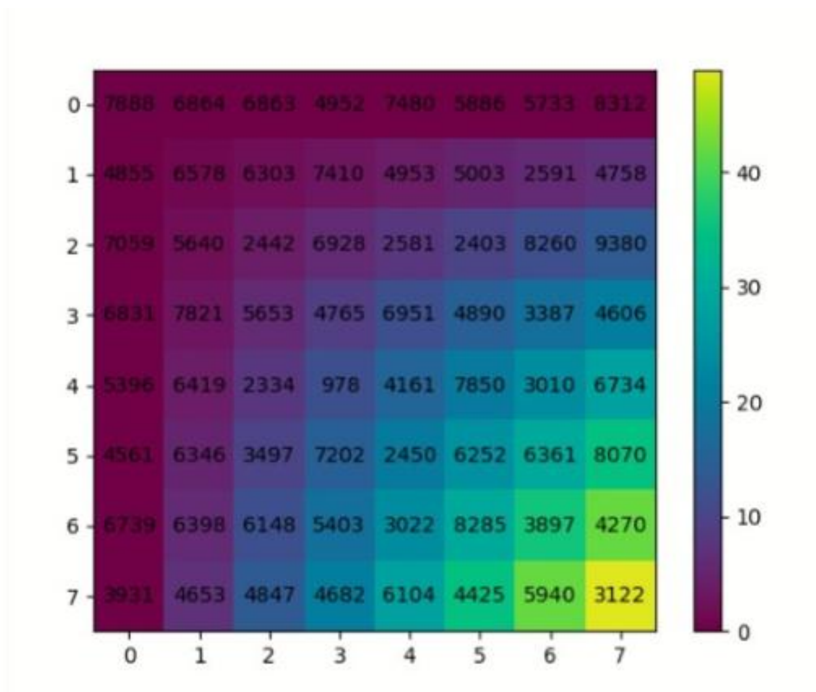
4. Comparison between two used approaches (Backtracking and Genetic)

- Backtracking is suitable for small or moderately sized Knight's Tour problems where complete exploration of the solution space is feasible, Genetic algorithms may be more appropriate for larger Knight's Tour instances.
- If a guaranteed optimal solution is required, backtracking is preferred for the Knight's Tour , Genetic algorithms are suitable when a good heuristic solution is acceptable, and optimality is not strictly necessary.
- Backtracking is effective for the well-defined structure of the Knight's Tour problem with clear decision points , Genetic algorithms are versatile and can handle the complexity of the Knight's Tour problem's solution space.
- Backtracking aims for optimality but may be computationally expensive for larger Knight's Tour instances , Genetic algorithms focus on efficiency and can quickly converge to good solutions for the Knight's Tour but may not guarantee optimality.
- **If we want to see real comparison for both approaches ?**

```
The run time was: 224.31919145584106 Sec
[(1, 445), (1, 489), (1, 286), (4, 316), (2, 416), (1, 420), (2, 163), (2, 446)]
[(3, 237), (1, 116), (8, 167), (6, 197), (3, 191), (3, 89), (5, 174), (2, 163)]
[(1, 291), (2, 165), (1, 377), (4, 589), (6, 198), (3, 208), (9, 175), (1, 395)]
[(1, 158), (2, 197), (2, 303), (2, 208), (2, 270), (3, 354), (2, 244), (2, 381)]
[(4, 341), (1, 260), (6, 235), (1, 404), (1, 221), (3, 224), (8, 181), (2, 241)]
[(2, 347), (2, 179), (1, 316), (1, 259), (5, 204), (6, 134), (3, 285), (3, 191)]
[(2, 282), (8, 162), (5, 119), (3, 244), (1, 238), (4, 261), (7, 273), (2, 247)]
[(1, 420), (1, 366), (1, 276), (1, 343), (1, 282), (2, 431), (3, 197), (1, 783)]
```

This is represents the average number of generations needed to produce the first tour from that square (average for five runs), while the second value represents the total number of distinct solutions found in the five runs for that square for 8*8 board

VS



The backtracking approach that represents the number of solutions found for every state(square) in 15 seconds.

Criteria	Genetic algorithm	Backtracking algorithm

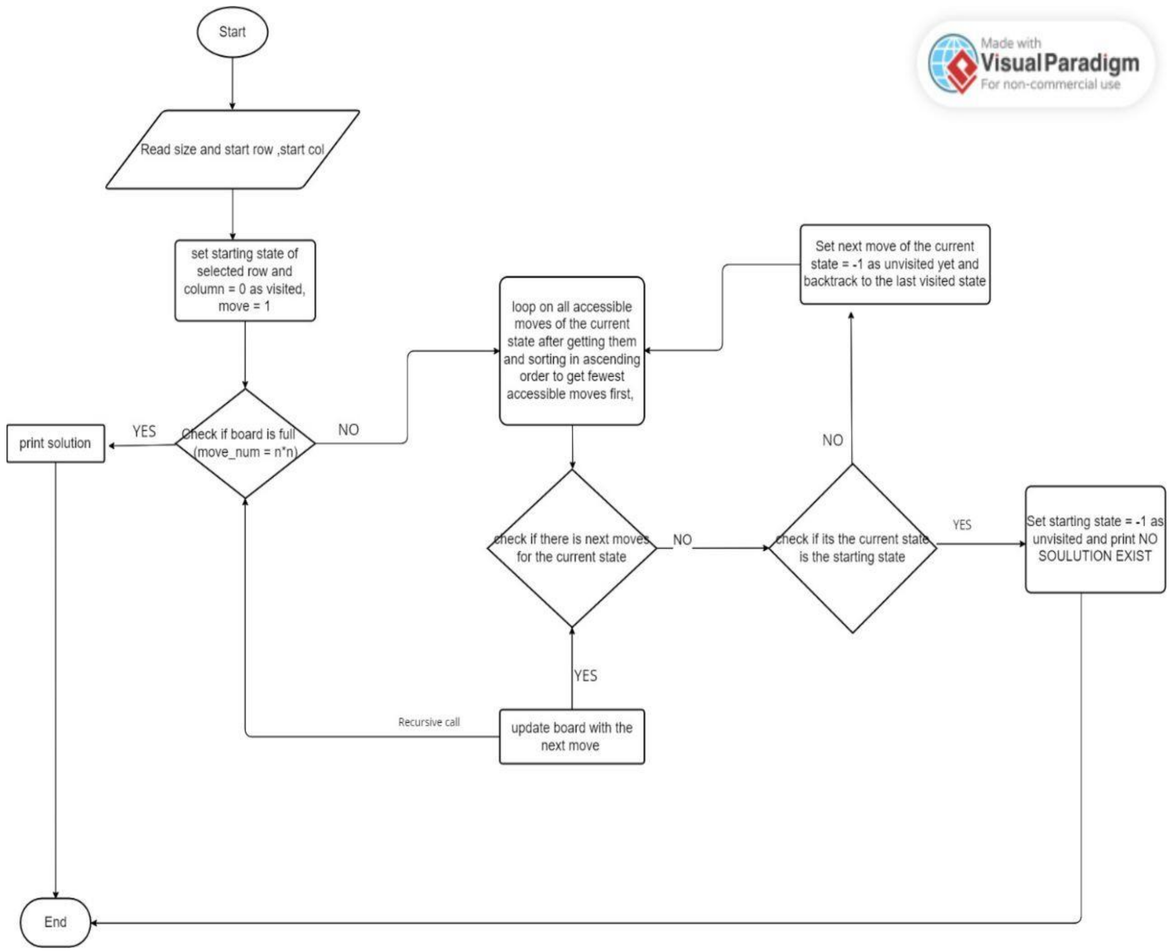
Quality of solution (n = 8)	**Good**	**Optimal**
Time to find first solution	**longer**	**shorter**
Number of solutions found	**multiple**	**single**

Conclusion : as illustrated how backtracking represents larger number of solutions for each state than genetic in short time , for optimal solutions in a short running time backtracking is recommended , while if you focus on efficiency and algorithm that can quickly converge to good solutions so Genetic algorithm is your choice .

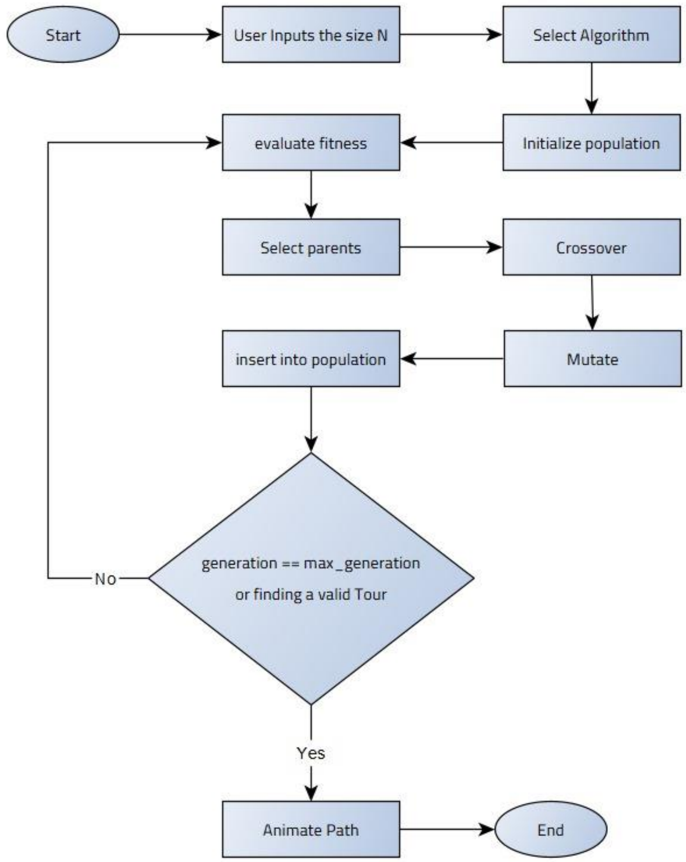
5 . Diagrams

1. Flowcharts

Backtracking

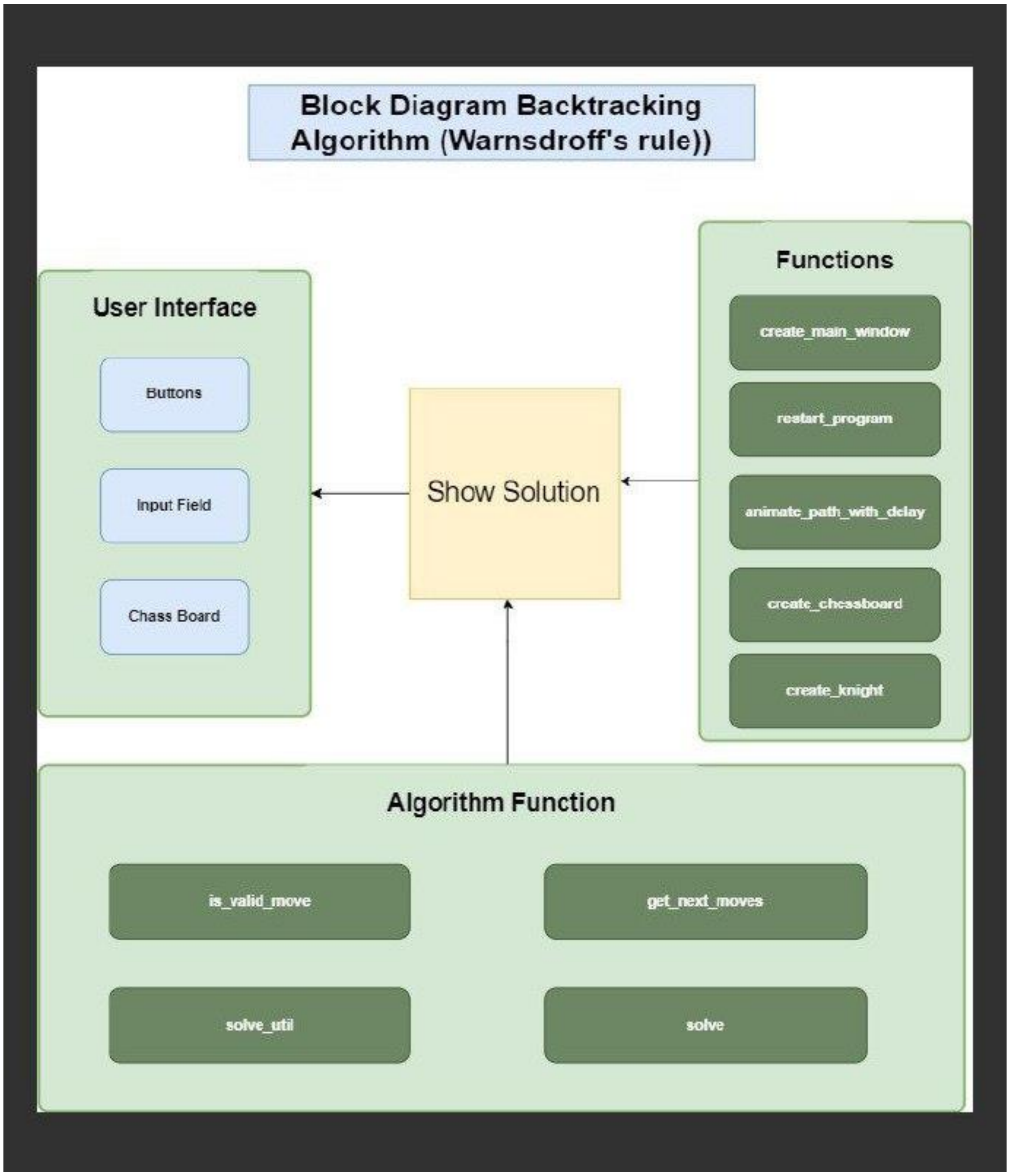


Genetic flowchart :

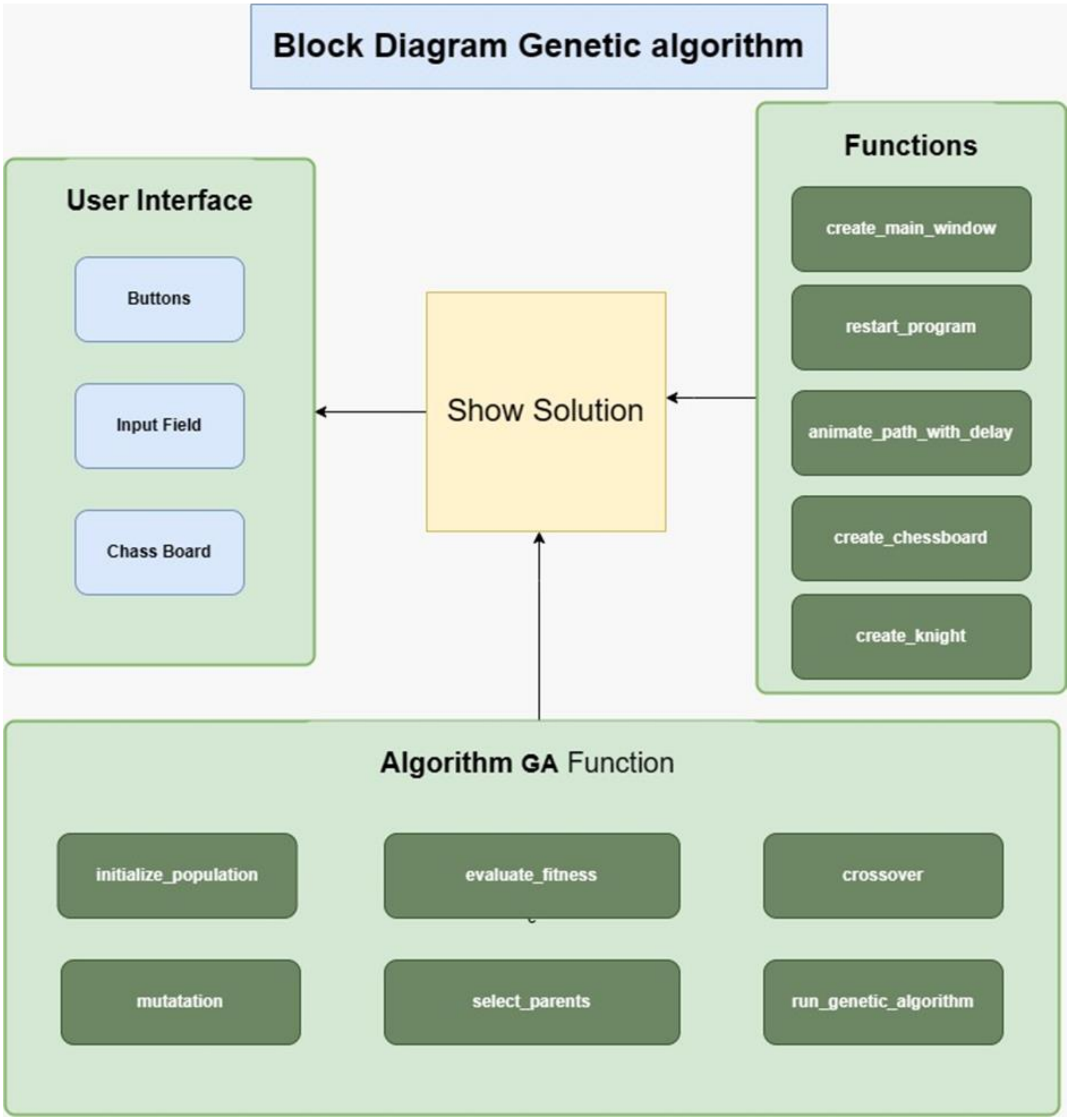


2.Block diagram

Backtracking :



Block diagram Genetic:



3. Use case diagram