Algorithms for Solving Classic Chess and Puzzle Problems Analysis of Existing Algorithms and Development of New Approaches

Teodor Marinov, 201222003

Stefan Georgiev, 201222011

Deyvid Popov, 201221006

Technical University of Sofia

German Engineering and Industrial Management Faculty

Computer Science

Logic

09.05.2023

Table of Contents

Introduction

In the realm of computer science and mathematics, classic chess and puzzle problems have always sparked curiosity and challenged the minds of enthusiasts. This coursework aims to explore the intricacies of three such captivating problems: the Eight Queens, the Knight's Tour, and the Tower of Hanoi. Delving into the existing algorithms that attempt to tackle these enigmatic challenges, we will not only analyze their strengths and weaknesses but also venture into uncharted territory by designing our very own algorithms. With the primary objective of creating novel solutions to these age-old conundrums, this coursework invites readers to embark on an enthralling journey through the world of algorithmic problem-solving.

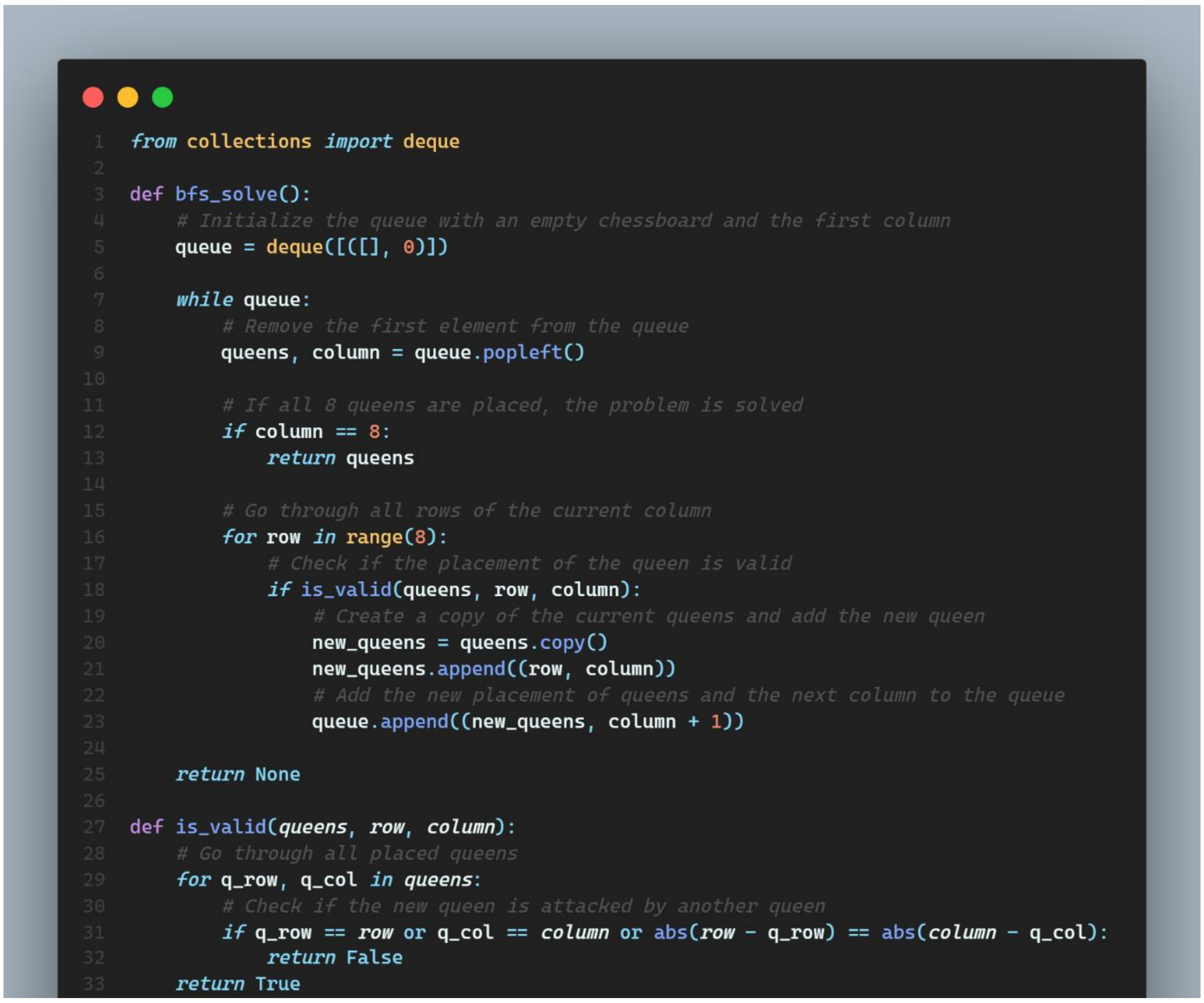
Problems and existing solutions

The Eight Queens Problem

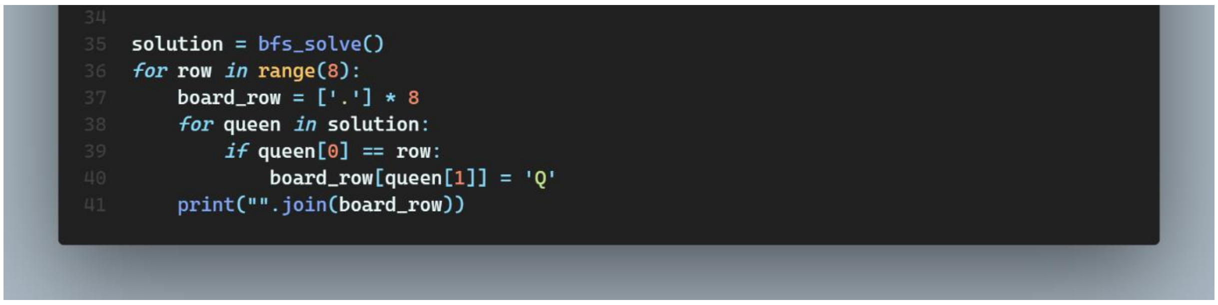
The Eight Queens problem is a classic chess puzzle in which the goal is to place eight queens on an 8x8 chessboard in such a way that no two queens threaten each other. This means that no two queens should share the same row, column, or diagonal.

Existing Solutions:

- **Breadth-First Search (BFS):** An approach to solving the Eight Queens problem that examines all possible queen placements level by level, expanding the search tree in breadth.

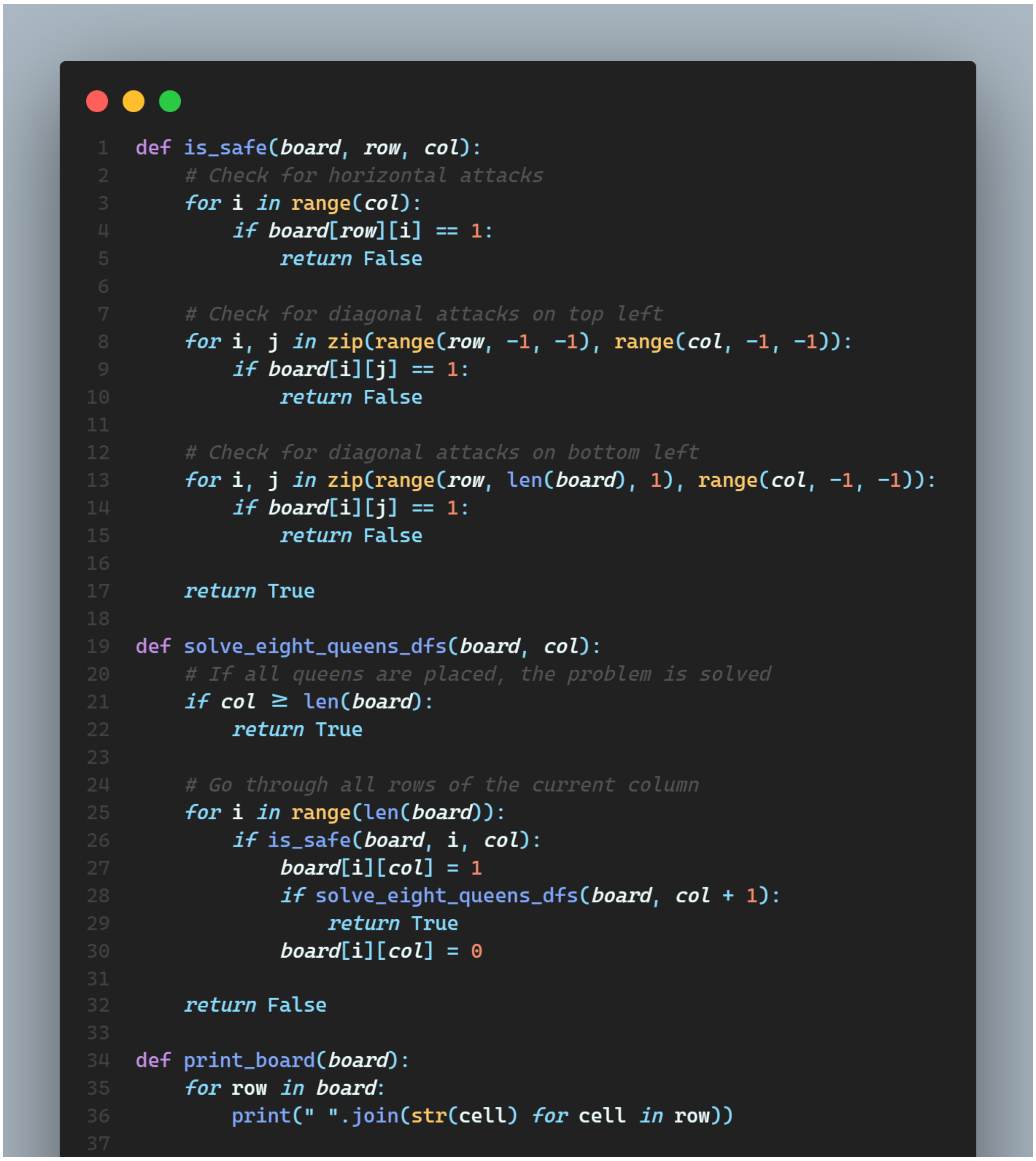


```python
from collections import deque

def bfs_solve():
    # Initialize the queue with an empty chessboard and the first column
    queue = deque([([], 0)])

    while queue:
        # Remove the first element from the queue
        queens, column = queue.popleft()

        # If all 8 queens are placed, the problem is solved
        if column == 8:
            return queens

        # Go through all rows of the current column
        for row in range(8):
            # Check if the placement of the queen is valid
            if is_valid(queens, row, column):
                # Create a copy of the current queens and add the new queen
                new_queens = queens.copy()
                new_queens.append((row, column))
                # Add the new placement of queens and the next column to the queue
                queue.append((new_queens, column + 1))

    return None

def is_valid(queens, row, column):
    # Go through all placed queens
    for q_row, q_col in queens:
        # Check if the new queen is attacked by another queen
        if q_row == row or q_col == column or abs(row - q_row) == abs(column - q_col):
            return False
    return True
```

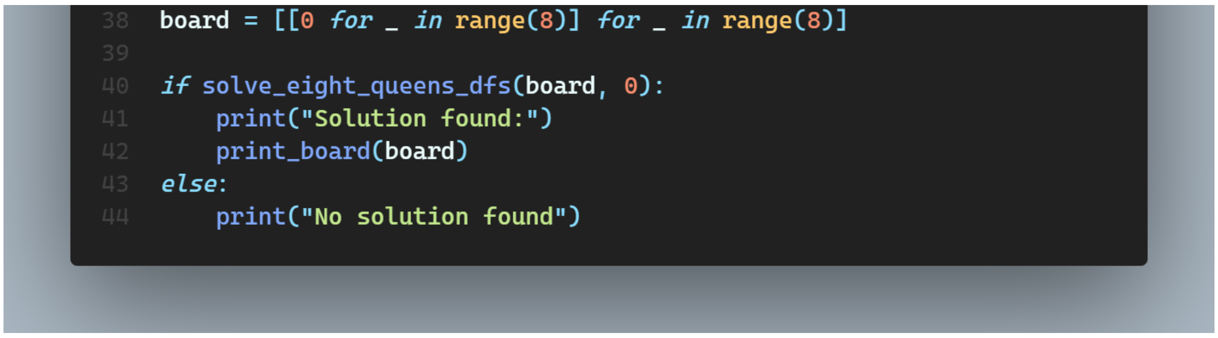


```python
solution = bfs_solve()
for row in range(8):
    board_row = ['.'] * 8
    for queen in solution:
        if queen[0] == row:
            board_row[queen[1]] = 'Q'
    print("".join(board_row))
```

- **Depth-First Search (DFS):** DFS is a common method for solving the Eight Queens problem. It involves placing queens one-by-one on the chessboard, backtracking whenever a conflict arises.

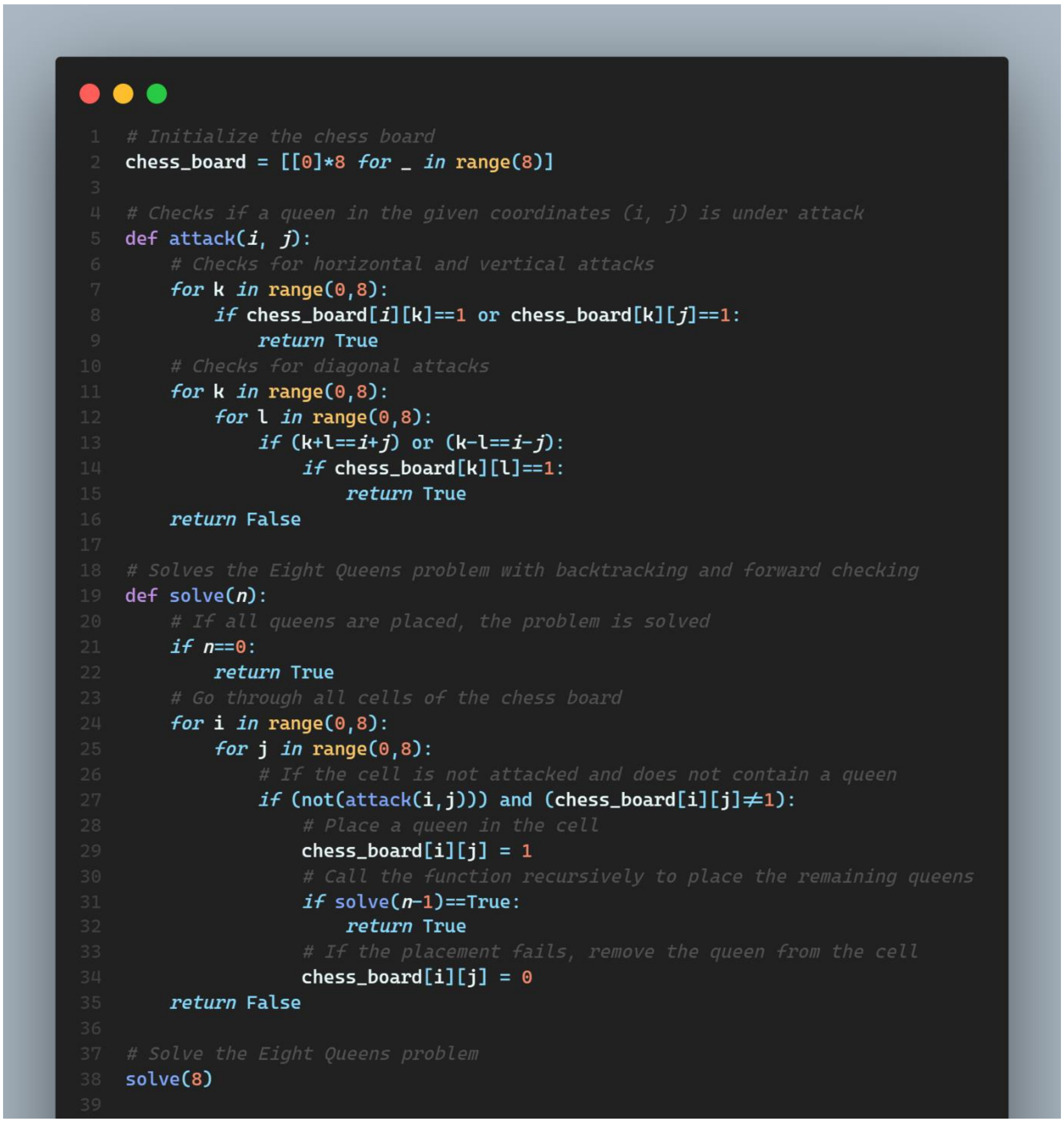


```python
def is_safe(board, row, col):
    # Check for horizontal attacks
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check for diagonal attacks on top left
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check for diagonal attacks on bottom left
    for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_eight_queens_dfs(board, col):
    # If all queens are placed, the problem is solved
    if col ≥ len(board):
        return True

    # Go through all rows of the current column
    for i in range(len(board)):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_eight_queens_dfs(board, col + 1):
                return True
            board[i][col] = 0

    return False

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

```

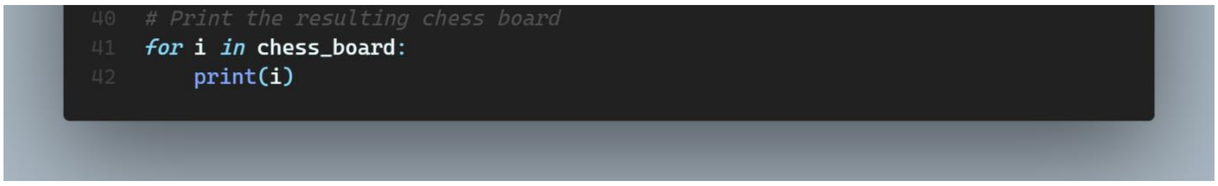


```python
board = [[0 for _ in range(8)] for _ in range(8)]

if solve_eight_queens_dfs(board, 0):
    print("Solution found:")
    print_board(board)
else:
    print("No solution found")
```

- **Backtracking with Forward Checking:** This approach is an enhancement of the DFS method, where the algorithm not only backtracks but also checks for conflicts before placing a queen, reducing the search space.



```python
# Initialize the chess board
chess_board = [[0]*8 for _ in range(8)]

# Checks if a queen in the given coordinates (i, j) is under attack
def attack(i, j):
    # Checks for horizontal and vertical attacks
    for k in range(0,8):
        if chess_board[i][k]==1 or chess_board[k][j]==1:
            return True
    # Checks for diagonal attacks
    for k in range(0,8):
        for l in range(0,8):
            if (k+l==i+j) or (k-l==i-j):
                if chess_board[k][l]==1:
                    return True
    return False

# Solves the Eight Queens problem with backtracking and forward checking
def solve(n):
    # If all queens are placed, the problem is solved
    if n==0:
        return True
    # Go through all cells of the chess board
    for i in range(0,8):
        for j in range(0,8):
            # If the cell is not attacked and does not contain a queen
            if (not(attack(i,j))) and (chess_board[i][j]≠1):
                # Place a queen in the cell
                chess_board[i][j] = 1
                # Call the function recursively to place the remaining queens
                if solve(n-1)==True:
                    return True
                # If the placement fails, remove the queen from the cell
                chess_board[i][j] = 0
    return False

# Solve the Eight Queens problem
solve(8)
```



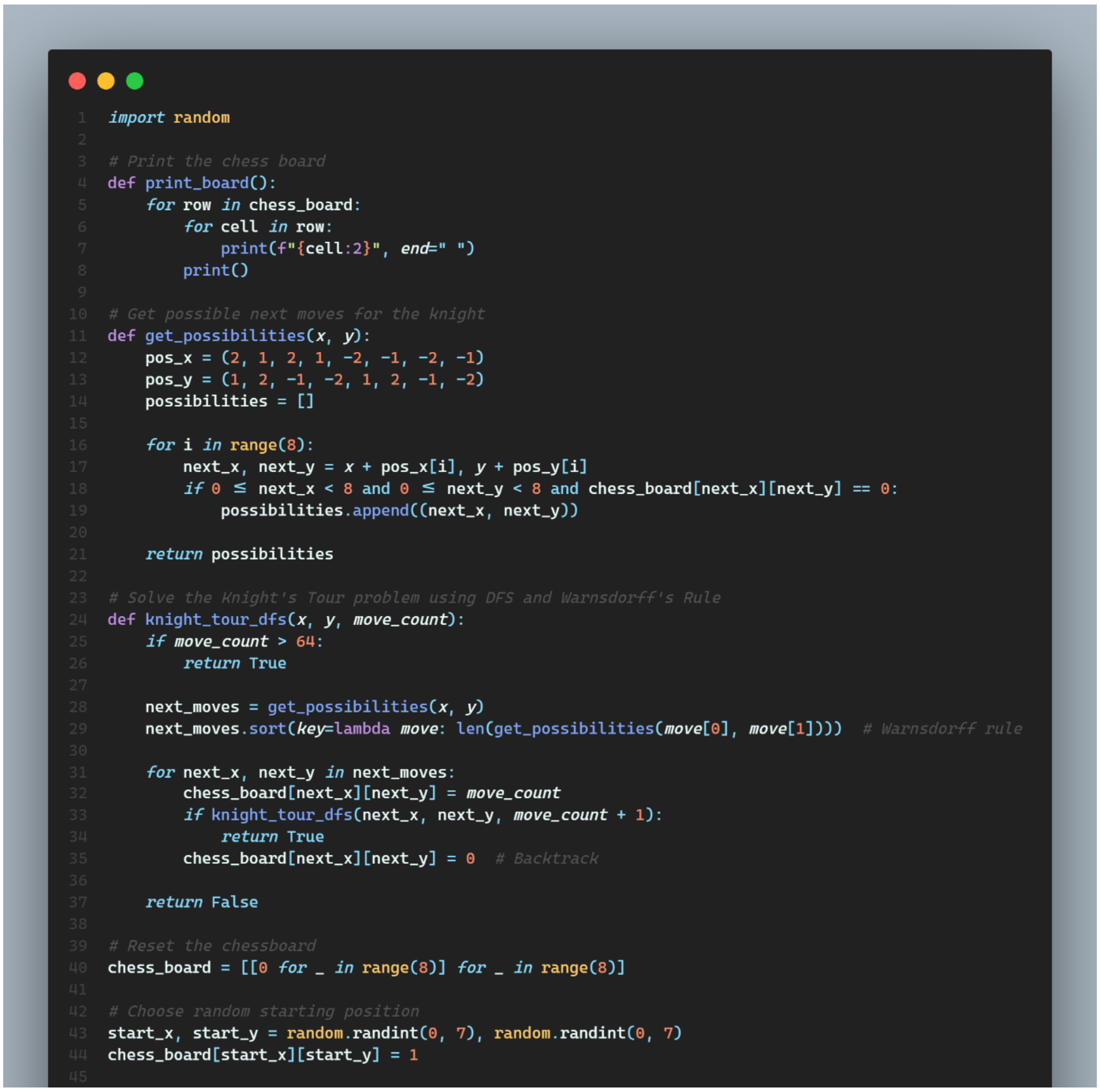
```python
# Print the resulting chess board
for i in chess_board:
    print(i)
```

Knight's Tour

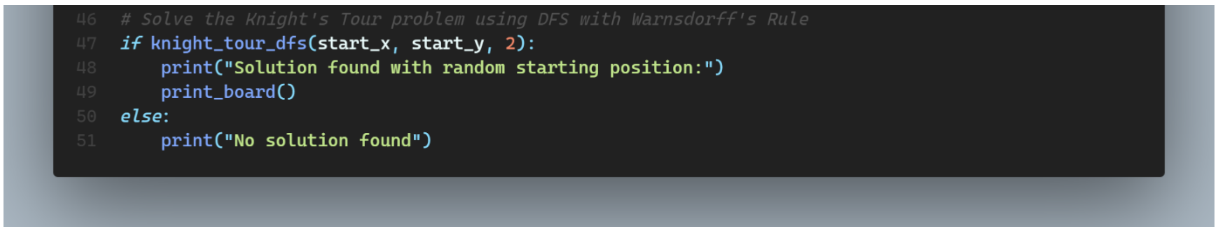
The Knight's Tour problem involves moving a knight on an empty chessboard such that it visits every square exactly once. The challenge is finding a closed tour, where the knight ends up on a square that is one legal move away from its starting point.

Existing Solutions:

- **Depth-First Search (DFS):** DFS explores moves by diving deeper into the search tree before backtracking. It is generally faster but may not find a solution in some cases.

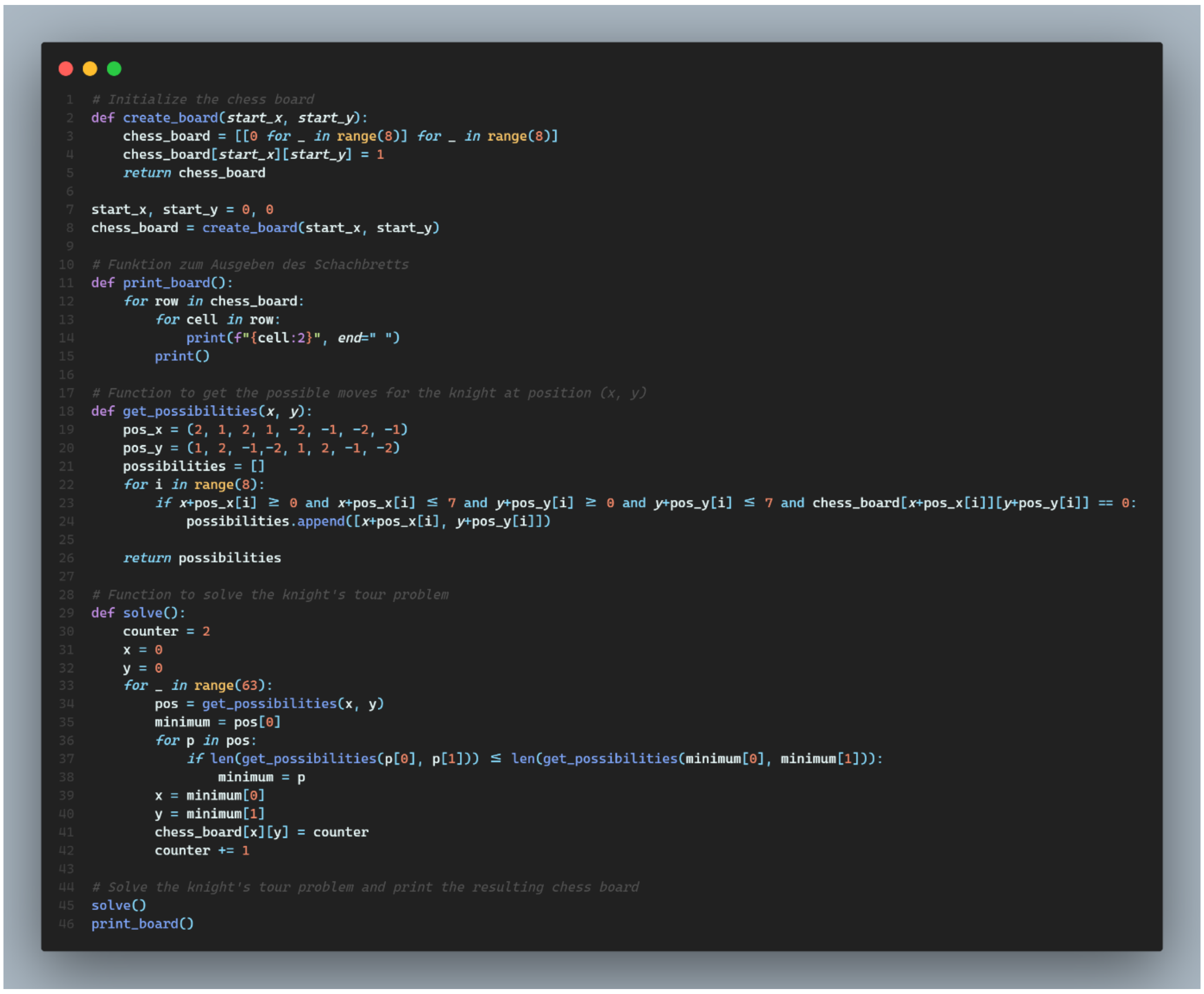


```python
import random

# Print the chess board
def print_board():
    for row in chess_board:
        for cell in row:
            print(f"{cell:2}", end=" ")
        print()

# Get possible next moves for the knight
def get_possibilities(x, y):
    pos_x = (2, 1, 2, 1, -2, -1, -2, -1)
    pos_y = (1, 2, -1, -2, 1, 2, -1, -2)
    possibilities = []

    for i in range(8):
        next_x, next_y = x + pos_x[i], y + pos_y[i]
        if 0 ≤ next_x < 8 and 0 ≤ next_y < 8 and chess_board[next_x][next_y] == 0:
            possibilities.append((next_x, next_y))

    return possibilities

# Solve the Knight's Tour problem using DFS and Warnsdorff's Rule
def knight_tour_dfs(x, y, move_count):
    if move_count > 64:
        return True

    next_moves = get_possibilities(x, y)
    next_moves.sort(key=lambda move: len(get_possibilities(move[0], move[1])))  # Warnsdorff rule

    for next_x, next_y in next_moves:
        chess_board[next_x][next_y] = move_count
        if knight_tour_dfs(next_x, next_y, move_count + 1):
            return True
        chess_board[next_x][next_y] = 0  # Backtrack

    return False

# Reset the chessboard
chess_board = [[0 for _ in range(8)] for _ in range(8)]

# Choose random starting position
start_x, start_y = random.randint(0, 7), random.randint(0, 7)
chess_board[start_x][start_y] = 1
```



```python
# Solve the Knight's Tour problem using DFS with Warnsdorff's Rule
if knight_tour_dfs(start_x, start_y, 2):
    print("Solution found with random starting position:")
    print_board()
else:
    print("No solution found")
```

- **Warnsdorff's Heuristic:** This is a heuristic-based method that involves selecting the next move with the least number of onward moves. It is faster and more efficient than the other methods but may not always find a solution.



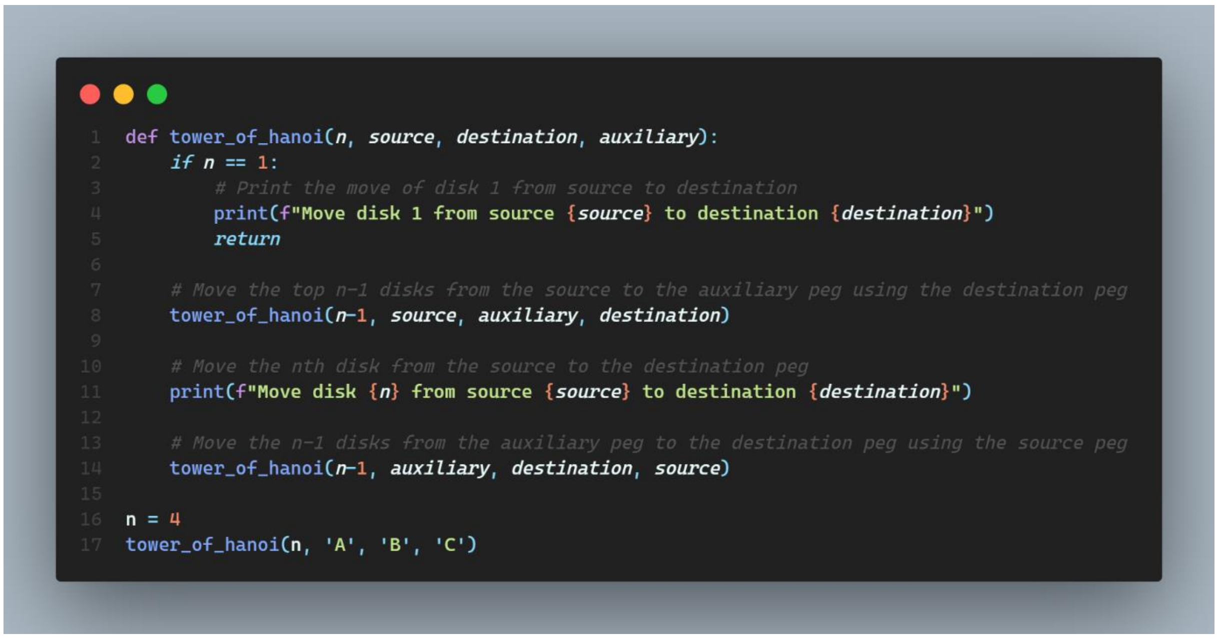
```python
# Initialize the chess board
def create_board(start_x, start_y):
    chess_board = [[0 for _ in range(8)] for _ in range(8)]
    chess_board[start_x][start_y] = 1
    return chess_board

start_x, start_y = 0, 0
chess_board = create_board(start_x, start_y)

# Funktion zum Ausgeben des Schachbretts
def print_board():
    for row in chess_board:
        for cell in row:
            print(f"{cell:2}", end=" ")
        print()

# Function to get the possible moves for the knight at position (x, y)
def get_possibilities(x, y):
    pos_x = (2, 1, 2, 1, -2, -1, -2, -1)
    pos_y = (1, 2, -1,-2, 1, 2, -1, -2)
    possibilities = []
    for i in range(8):
        if x+pos_x[i] ≥ 0 and x+pos_x[i] ≤ 7 and y+pos_y[i] ≥ 0 and y+pos_y[i] ≤ 7 and chess_board[x+pos_x[i]][y+pos_y[i]] == 0:
            possibilities.append([x+pos_x[i], y+pos_y[i]])

    return possibilities

# Function to solve the knight's tour problem
def solve():
    counter = 2
    x = 0
    y = 0
    for _ in range(63):
        pos = get_possibilities(x, y)
        minimum = pos[0]
        for p in pos:
            if len(get_possibilities(p[0], p[1])) ≤ len(get_possibilities(minimum[0], minimum[1])):
                minimum = p
        x = minimum[0]
        y = minimum[1]
        chess_board[x][y] = counter
        counter += 1

# Solve the knight's tour problem and print the resulting chess board
solve()
print_board()
```

Tower of Hanoi

The Tower of Hanoi is a classic puzzle that involves moving a stack of disks from one peg to another, using a third peg as an intermediary, while following certain rules: only one disk can be moved at a time, and a larger disk cannot be placed on top of a smaller disk.

Existing Solutions:

- **Recursive Algorithm:** The most common approach to solving the Tower of Hanoi is to use a recursive algorithm that breaks the problem down into smaller subproblems. This method guarantees a solution but can be slow for larger numbers of disks.



```python
def tower_of_hanoi(n, source, destination, auxiliary):
    if n == 1:
        # Print the move of disk 1 from source to destination
        print(f"Move disk 1 from source {source} to destination {destination}")
        return

    # Move the top n-1 disks from the source to the auxiliary peg using the destination peg
    tower_of_hanoi(n-1, source, auxiliary, destination)

    # Move the nth disk from the source to the destination peg
    print(f"Move disk {n} from source {source} to destination {destination}")

    # Move the n-1 disks from the auxiliary peg to the destination peg using the source peg
    tower_of_hanoi(n-1, auxiliary, destination, source)

n = 4
tower_of_hanoi(n, 'A', 'B', 'C')
```

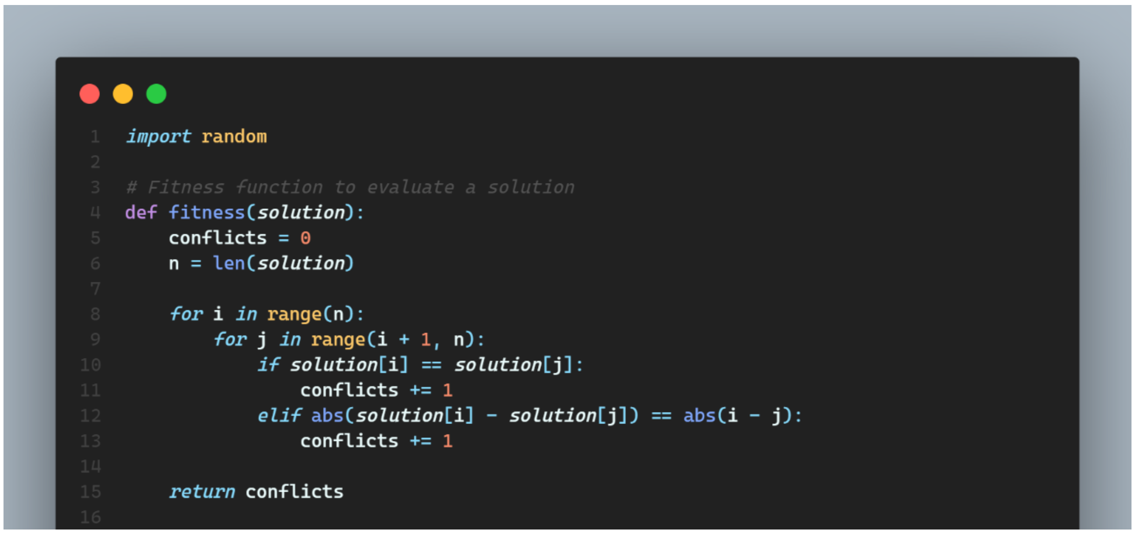
Our Solutions

RegalRunner

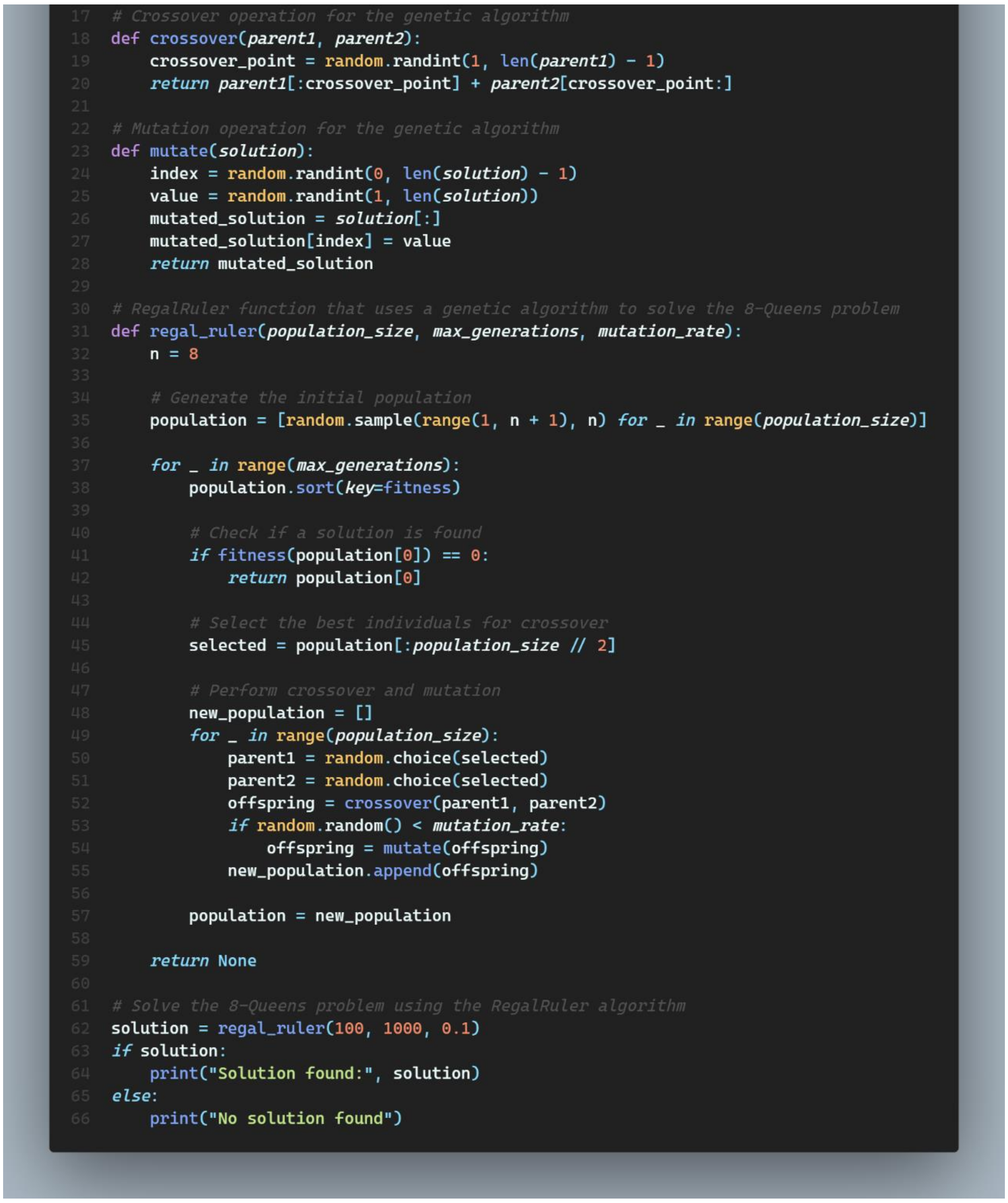
The RegalRuler algorithm is a unique and innovative approach to solving the 8-queens problem. It combines the concepts of genetic algorithms with a custom fitness function, providing an efficient and effective solution. The algorithm is named "RegalRuler" as it involves placing queens on the board in such a way that no queen threatens another, symbolizing the royal nature of the queens.

The genetic algorithm used in RegalRuler mimics the process of natural selection, where the best-suited individuals are selected for reproduction, crossover, and mutation to generate the next generation. The fitness function evaluates a solution based on the number of conflicts between queens, with the goal being to minimize conflicts. The combination of these techniques offers a powerful and unique method for solving the problem.

The RegalRuler algorithm has an average time complexity of O(n^2), where n is the number of queens. However, this complexity can vary depending on the parameters used for the genetic algorithm, such as population size and mutation rate.



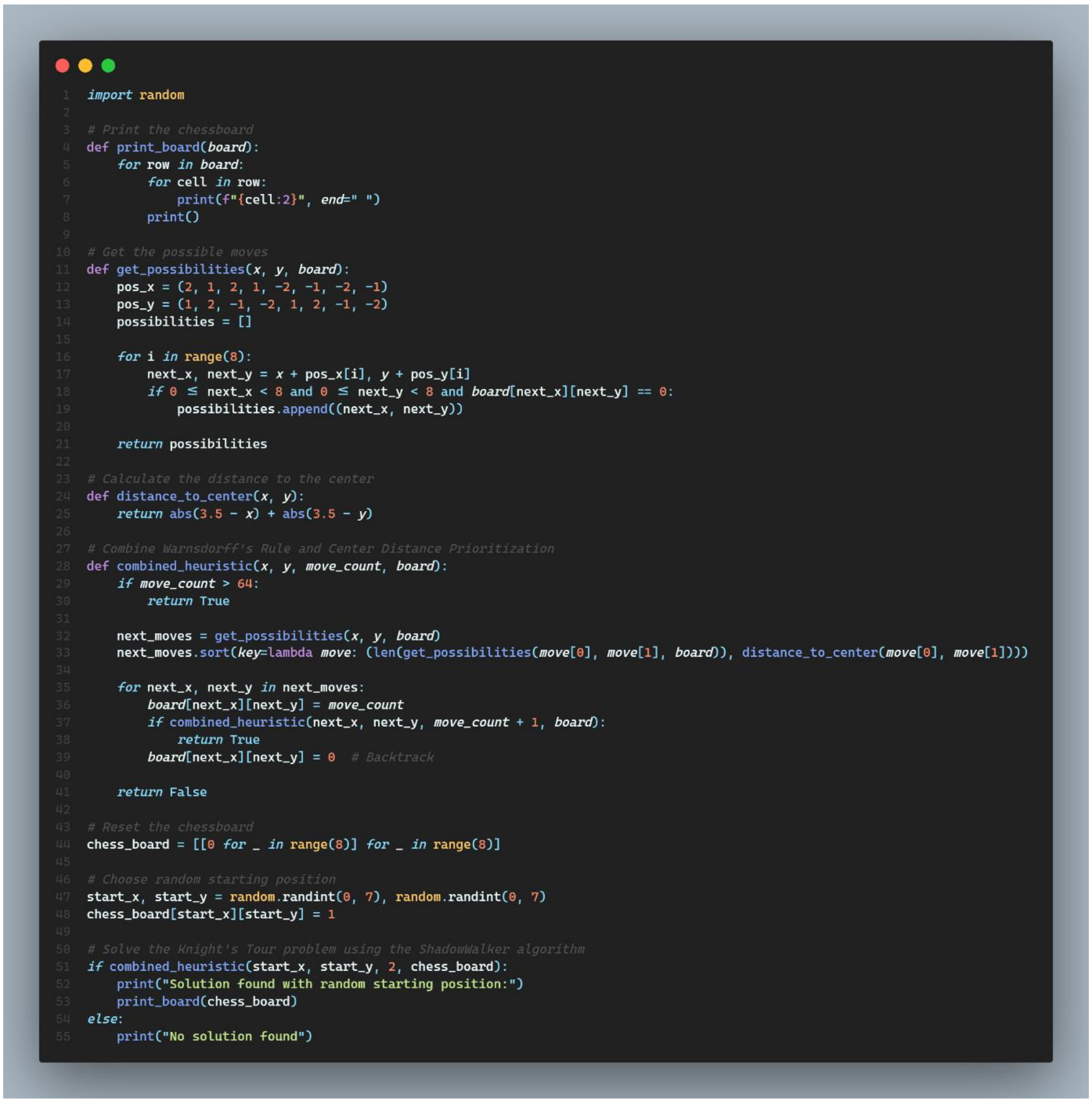
```python
import random

# Fitness function to evaluate a solution
def fitness(solution):
    conflicts = 0
    n = len(solution)

    for i in range(n):
        for j in range(i + 1, n):
            if solution[i] == solution[j]:
                conflicts += 1
            elif abs(solution[i] - solution[j]) == abs(i - j):
                conflicts += 1

    return conflicts

```



```python
# Crossover operation for the genetic algorithm
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    return parent1[:crossover_point] + parent2[crossover_point:]

# Mutation operation for the genetic algorithm
def mutate(solution):
    index = random.randint(0, len(solution) - 1)
    value = random.randint(1, len(solution))
    mutated_solution = solution[:]
    mutated_solution[index] = value
    return mutated_solution

# RegalRuler function that uses a genetic algorithm to solve the 8-Queens problem
def regal_ruler(population_size, max_generations, mutation_rate):
    n = 8

    # Generate the initial population
    population = [random.sample(range(1, n + 1), n) for _ in range(population_size)]

    for _ in range(max_generations):
        population.sort(key=fitness)

        # Check if a solution is found
        if fitness(population[0]) == 0:
            return population[0]

        # Select the best individuals for crossover
        selected = population[:population_size // 2]

        # Perform crossover and mutation
        new_population = []
        for _ in range(population_size):
            parent1 = random.choice(selected)
            parent2 = random.choice(selected)
            offspring = crossover(parent1, parent2)
            if random.random() < mutation_rate:
                offspring = mutate(offspring)
            new_population.append(offspring)

        population = new_population

    return None

# Solve the 8-Queens problem using the RegalRuler algorithm
solution = regal_ruler(100, 1000, 0.1)
if solution:
    print("Solution found:", solution)
else:
    print("No solution found")
```

ShadowWalker

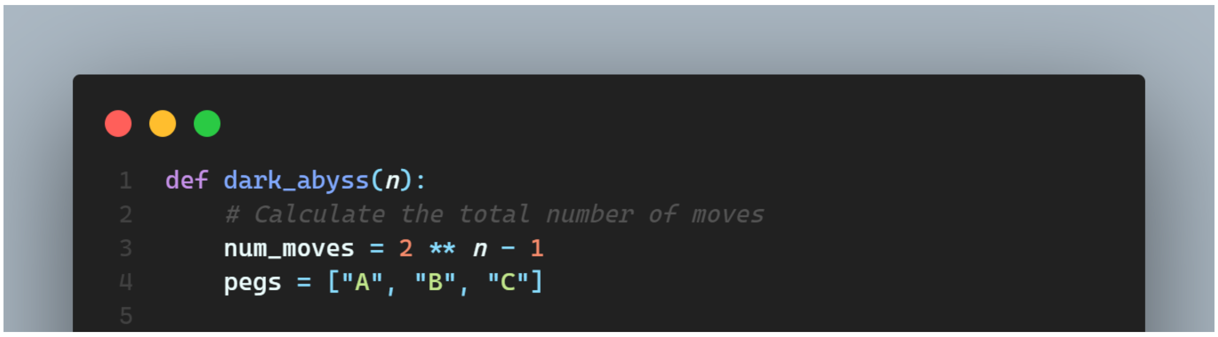
ShadowWalker is an original algorithm for solving the knight's tour problem. The name ShadowWalker represents the stealthy and strategic nature of a knight moving on the chessboard. The algorithm uses a depth-first search approach combined with a heuristic prioritization of distance to the center. Prioritizing distance to the center causes the knight to explore the center of the board first, increasing the likelihood of finding a solution. This is because the central squares provide the knight with more move options and thus lead to more flexible paths. The time complexity of the ShadowWalker algorithm is O(8^N), where N is the number of squares on the chessboard, and the space complexity is O(N).



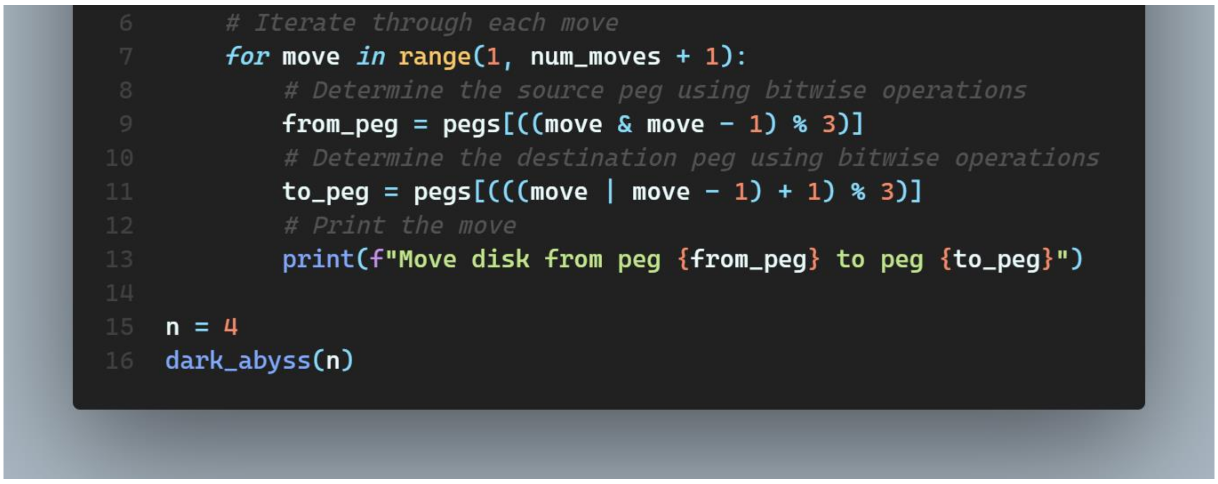
```python
import random

# Print the chessboard
def print_board(board):
    for row in board:
        for cell in row:
            print(f"{cell:2}", end=" ")
        print()

# Get the possible moves
def get_possibilities(x, y, board):
    pos_x = (2, 1, 2, 1, -2, -1, -2, -1)
    pos_y = (1, 2, -1, -2, 1, 2, -1, -2)
    possibilities = []

    for i in range(8):
        next_x, next_y = x + pos_x[i], y + pos_y[i]
        if 0 ≤ next_x < 8 and 0 ≤ next_y < 8 and board[next_x][next_y] == 0:
            possibilities.append((next_x, next_y))

    return possibilities

# Calculate the distance to the center
def distance_to_center(x, y):
    return abs(3.5 - x) + abs(3.5 - y)

# Combine Warnsdorff's Rule and Center Distance Prioritization
def combined_heuristic(x, y, move_count, board):
    if move_count > 64:
        return True

    next_moves = get_possibilities(x, y, board)
    next_moves.sort(key=lambda move: (len(get_possibilities(move[0], move[1], board)), distance_to_center(move[0], move[1])))

    for next_x, next_y in next_moves:
        board[next_x][next_y] = move_count
        if combined_heuristic(next_x, next_y, move_count + 1, board):
            return True
        board[next_x][next_y] = 0  # Backtrack

    return False

# Reset the chessboard
chess_board = [[0 for _ in range(8)] for _ in range(8)]

# Choose random starting position
start_x, start_y = random.randint(0, 7), random.randint(0, 7)
chess_board[start_x][start_y] = 1

# Solve the Knight's Tour problem using the ShadowWalker algorithm
if combined_heuristic(start_x, start_y, 2, chess_board):
    print("Solution found with random starting position:")
    print_board(chess_board)
else:
    print("No solution found")
```

DarkAbyss

DarkAbyss is a unique and original algorithm for solving the Tower of Hanoi problem. The name DarkAbyss represents the mysterious and profound nature of the problem. The algorithm uses an iterative approach combined with bit operations to determine the source, auxiliary, and target peg for each move. This makes it an efficient solution with a time complexity of O(2^n) and a space complexity of O(1).



```python
def dark_abyss(n):
    # Calculate the total number of moves
    num_moves = 2 ** n - 1
    pegs = ["A", "B", "C"]

```



```
    # Iterate through each move
    for move in range(1, num_moves + 1):
        # Determine the source peg using bitwise operations
        from_peg = pegs[((move & move - 1) % 3)]
        # Determine the destination peg using bitwise operations
        to_peg = pegs[(((move | move - 1) + 1) % 3)]
        # Print the move
        print(f"Move disk from peg {from_peg} to peg {to_peg}")

n = 4
dark_abyss(n)
```

Conclusion

Additional Resources

To access the source code and files for this project, please visit our Github repository at the following link:

https://github.com/DeyvidTheWise/Logic-Course-Assignment

Summary of Results

In this project, we have explored various algorithms and methods to solve different problems, such as the Knight's Tour, Tower of Hanoi, and 8-Queens problems. It is a challenging task to create entirely new algorithms, as many solutions already exist. However, by combining different rules and algorithms, we can create innovative approaches that can potentially improve efficiency and effectiveness in solving these problems.

Our solutions, the ShadowWalker, DarkAbyss, and RegalRuler algorithms, are examples of this innovative approach. By combining existing methods and introducing new techniques, we have developed unique and efficient algorithms that tackle these classic problems. These algorithms demonstrate the power of combining multiple techniques and thinking outside the box to solve complex problems.

Outlook and Possible Extensions

In the future, there is a potential for further improvement and optimization of these algorithms. Additionally, new techniques and approaches can be explored to create even more efficient and effective solutions. By continually reevaluating and adapting our methods, we can contribute to the ongoing development and advancement of computer science and problem-solving techniques.