INTRODUCTION TO ARTIFICIAL INTELLIGENCE (AI) AI101B

N- QUEENS PROBLEM

Project Report

BACHELOR OF TECHNOLOGY IN COMPUTER SCIENCE (AIML)

Section : B

SUBMITTED BY:

NAME: ITY GAUR

UNIVERSITY ROLL NO: 202401100400101

SUBMITTED TO:

MR. ABHISHEK SHUKLA

Introduction

The N-Queens problem is an old combinatorial puzzle consisting of putting N queens on an N×N chessboard such that no two queens attack each other. That is:

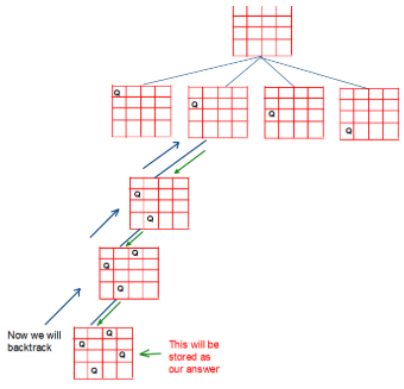
No two queens occupy the same row.

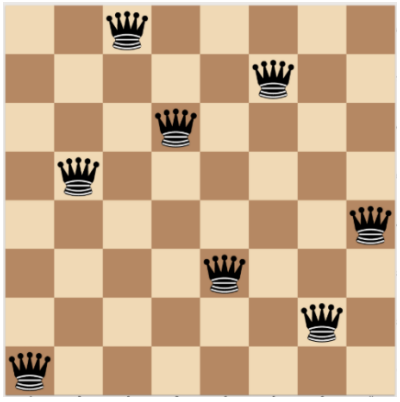
No two queens occupy the same column.

No two queens occupy the same diagonal.

This is a very standard problem in artificial intelligence and optimization because of its complexity and several potential solutions. It has numerous methods with which it can be solved, such as using backtracking, constraint satisfaction, or metaheuristic algorithms. As N becomes larger, it is harder to solve the problem, making this an ideal case study for methods of optimization.

In this report, we concentrate on finding the solution using the Hill Climbing algorithm with random restarts, a heuristic method that is meant to effectively find a valid configuration. The method provides an effective alternative to brute-force methods by making incremental improvements to a solution until a conflict-free configuration is achieved. Through the use of random restarts, the algorithm prevents being trapped in local optima, raising its success rate.





Methodology

Approach: Hill Climbing with Random Restarts

Hill Climbing is a local search algorithm that tries to locate an improved solution by making small adjustments to the current state. It tends to get stuck in local optima—locations where no direct improvement can be made. In order to correct this, we apply random restarts, which reset the algorithm to a fresh random configuration when it gets stuck. This is to ensure that the algorithm is not stuck in a suboptimal solution and has a greater chance of reaching a valid placement.

Steps Traversed in Our Solution

1. **Generate a Random Board:** The board is given as a list of length N, with each index denoting a column and the value stored at each index denoting the row where the queen is to be placed. The initial board setup is randomly created to ensure variety in queens' starting positions.

2. **Assess the Board (Conflict Calculation):** The count of conflicting pairs of queens is calculated. Queens conflict if: They are on the same row.They are on the same diagonal (major and minor diagonals).The evaluation function gives a measure for estimating the quality of the current board setup.

3. Local Search for Improvement: For every column, the algorithm attempts to move the queen to a new row and computes the new number of conflicts. If a better solution (with less conflicts) is obtained, it is accepted and kept as the new state. If no improvement is possible, the algorithm moves on to the next column.This process is repeated iteratively, improving the solution step by step.

4. Random Restart Mechanism: If no better move is discovered for the whole board, a new random board is created, and the algorithm begins again.This ensures that the algorithm does not become trapped in local optima, thus making it more robust.The number of restarts permitted is fixed to guarantee computational efficiency.

Solution Validation:

The algorithm runs until a conflict-free solution is achieved. If no solution is reached after a specified number of tries, the algorithm stops and indicates failure.

The last configuration is then output, graphically illustrating the positioning of queens on the board.

Advantages of This Method

- Quicker than brute-force strategies: In contrast to exhaustive search methods, Hill Climbing rapidly converges to a solution by constantly enhancing the state of the board.
- Memory-efficient: The algorithm only needs to store the current board state, so it can be used with larger N.
- Practical for larger N: Although backtracking algorithms will have problems with bigger chessboards, this heuristic algorithm can frequently solve more quickly.
- Challenges and Limitations
- Local optima: The algorithm can get stuck in a state where there is no way to improve
- immediately, so random restarts are needed.

- No guarantee of success: In contrast to exhaustive search algorithms, Hill Climbing does not necessarily find a solution within a reasonable number of steps.
- Dependent on initial configuration: The algorithm's efficiency greatly depends on the quality of the initial random board.

This method achieves an optimum trade-off between efficiency and accuracy and is thus an effective way of solving the N-Queens problem, particularly when working with large board sizes for which brute-force algorithms would prove to be impractical.

CODE OF THE PROBLEM:

```
import random  # For generating random board configurations
def random_board(n):
    """Generate a random board where each queen is placed in a random row."""
    return [random.randint(0, n-1) for _ in range(n)]

def count_conflicts(board):
    """Count the number of attacking queen pairs (same row or diagonal)."""
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i+1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def hill_climbing(n, max_restarts=100):
    """Solve N-Queens using hill climbing with random restarts."""
    for _ in range(max_restarts):  # Try up to max_restarts times
        board = random_board(n)  # Start with a random board

        while count_conflicts(board) > 0:  # Keep improving until conflicts = 0
            best_board = board[:]  # Copy current board
            best_conflicts = count_conflicts(board)

            for col in range(n):  # Try moving each queen
                original_row = board[col]  # Save original position
                for row in range(n):  # Try placing in a different row
                    if row == original_row:
                        continue  # Skip if it's the same row

                    board[col] = row  # Move queen
                    new_conflicts = count_conflicts(board)

                    if new_conflicts < best_conflicts:  # If it's better, keep it
                        best_board = board[:]
                        best_conflicts = new_conflicts
```

```
            board[col] = original_row  # Reset to original row before next try

        if best_conflicts < count_conflicts(board):
            board = best_board  # Keep the improved board
        else:
            break  # No better move found, restart

    if count_conflicts(board) == 0:
        return board  # Solution found

  return None  # No solution found after max_restarts

def print_board(board):
  """Print the board with 'Q' for queens and '.' for empty spaces."""
  if board is None:
    print("No solution found.")
    return

  n = len(board)
  for row in range(n):
    print(" ".join("Q" if board[col] == row else "." for col in range(n)))
  print()  # Blank line for readability

def main():
  """Ask the user for a board size and solve N-Queens."""
  n = int(input("Enter chessboard size: "))
  solution = hill_climbing(n)

  print(f"\nSolution for {n}-Queens:")
  print_board(solution)

if __name__ == "__main__":
  main()
```

OUTPUTS:

```
    """Print the board with 'Q' for queens and '.' for empty spaces."""
    if board is None:
        print("No solution found.")
        return

    n = len(board)
    for row in range(n):
        print(" ".join("Q" if board[col] == row else "." for col in range(n)))
    print()  # Blank line for readability

def main():
    """Ask the user for a board size and solve N-Queens."""
    n = int(input("Enter chessboard size: "))
    solution = hill_climbing(n)

    print(f"\nSolution for {n}-Queens:")
    print_board(solution)

if __name__ == "__main__":
    main()
```

```
Enter chessboard size: 5

Solution for 5-Queens:
. . . Q .
Q . . . .
. . Q . .
. . . . Q
. Q . . .
```

```
        print("No solution found.")
        return

    n = len(board)
    for row in range(n):
        print(" ".join("Q" if board[col] == row else "." for col in range(n)))
    print()  # Blank line for readability

def main():
    """Ask the user for a board size and solve N-Queens."""
    n = int(input("Enter chessboard size: "))
    solution = hill_climbing(n)

    print(f"\nSolution for {n}-Queens:")
    print_board(solution)

if __name__ == "__main__":
    main()
```

```
Enter chessboard size: 9

Solution for 9-Queens:
. . . . . Q . . .
. . Q . . . . . .
. . . . . . . . Q
. . . Q . . . . .
Q . . . . . . . .
. . . . . . . Q .
. Q . . . . . . .
. . . . Q . . . .
. . . . . Q . . .
```

REFERENCES OF THE IMAGES :

1) The images are taken from takeuforward to make the reader understand the code better : https://images.app.goo.gl/ZAE7KfhuigySAvmV6
2) The screenshot of the output is from google colab: https://colab.research.google.com/drive/1rK1f8VgMZ0roXLpBrH3B3Lp14qb7fdTJ